Using and Extending AspectJ for Separating Concerns in Parallel Java Code

Bruno Harbulot and John Gurd

The University of Manchester

POOSC 2005 – Glasgow, July 2005

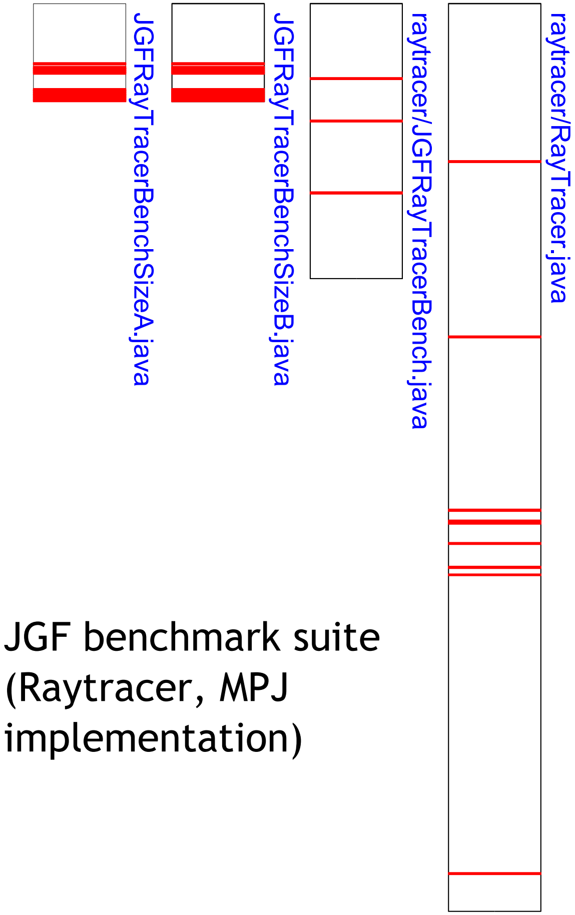
Presentation Outline

- Problem and Approach
- Using AspectJ for Parallelisation
- Extending AspectJ for Parallelisation

Presentation Outline

- Problem and Approach
- Using AspectJ for Parallelisation
- Extending AspectJ for Parallelisation

Problem: Code tangling in scientific software


JGFRayTracerBenchSizeA.java
JGFRayTracerBenchSizeB.java
raytracer/JGFRayTracerBench.java
raytracer/RayTracer.java

JGF benchmark suite (Raytracer, MPJ implementation)

- Statements for parallelism tangled within the numerical algorithm
- Parallelisation cannot be encapsulated in its own class or procedure
- Difficult to extract and reuse numerical algoritm only, in another context

Separation of Concerns

- **Concern**: anything about a software system (feature, requirement, ...)
- "*Separation of concerns*": Dijkstra [Dij76, ch. 27]
- Designing software: separating concerns into units such as procedures, classes, methods, libraries, etc.
- Two concerns **crosscut** each other when their relation implies code-tangling.
- **Crosscutting concern**: concern that crosscuts the main purpose of a unit.

Aspect-Oriented Programming (I) Motivation

- Programming paradigm for encapsulating crosscutting concerns [KLM+97].
- AOP builds on top of other programming paradigms: object-oriented, imperative or functional. It does not supplant them.
- Encapsulate crosscutting concerns into **aspects**.

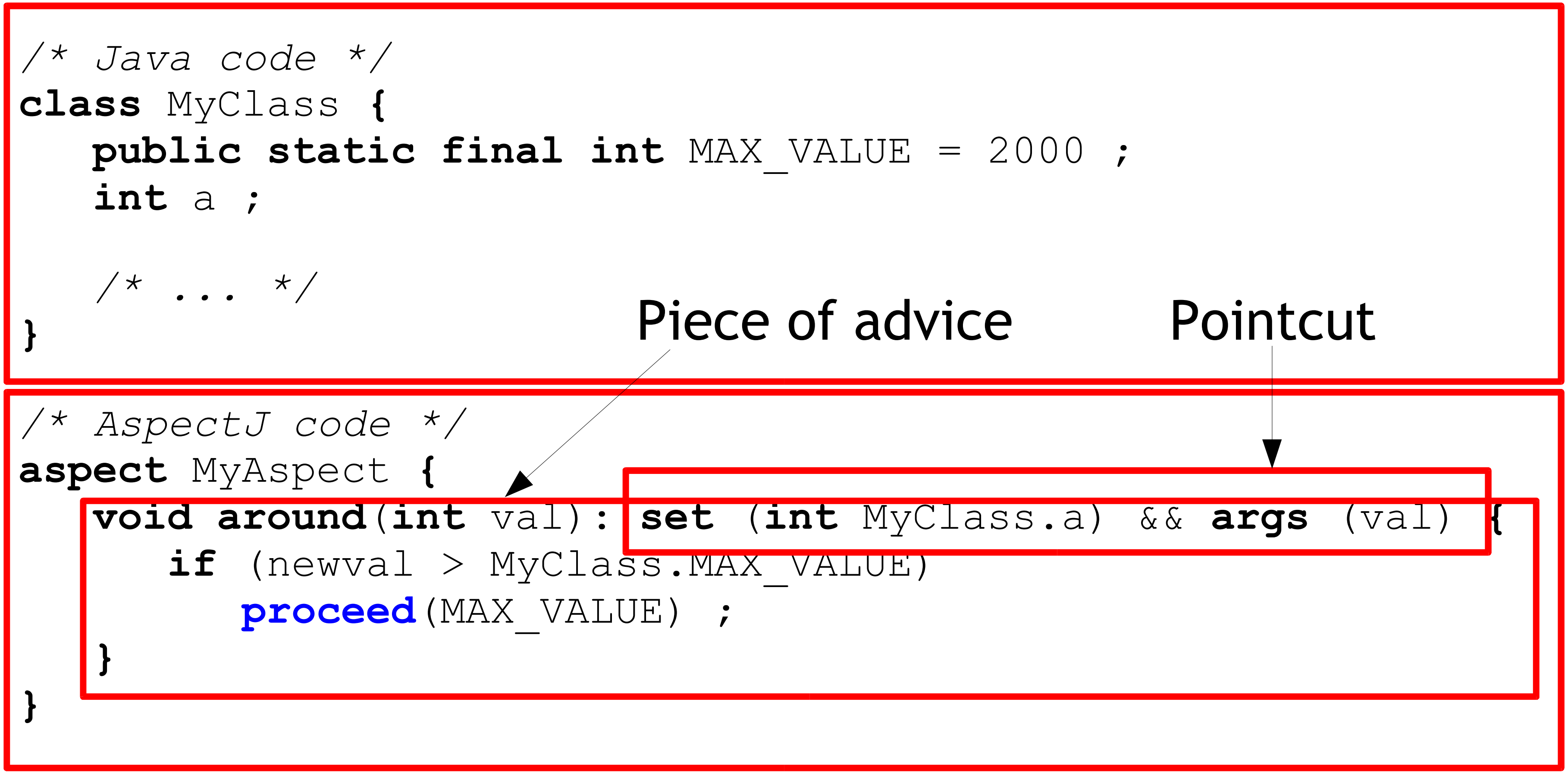
Aspect-Oriented Programming (II) Concepts

- Aspects contain statements of the form: "*[...] whenever condition C arises, perform action A.*" [FF00]
- **Join point**: point in the execution of a program where an aspect might intervene.
- **Pointcut**: expression of a subset of join points (*condition C*)
- **Advice**: piece of code for *action A*.
- Pointcuts and advice encapsulated into **aspects**.

AspectJ

- Aspect-Oriented extension to Java
- Compiles from Java source-code or byte-code
- Defines new constructs for writing aspects (aspect, pointcut, ...)
- Intervenes on object interfaces (field accesses, method calls, instantiation, ...)
- Produces Java byte-code (compatible with Java Virtual Machine specifications)

AspectJ example



```
/* Java code */
class MyClass {
   public static final int MAX_VALUE = 2000 ;
   int a ;

   /* ... */
}
```

Piece of advice

Pointcut

```
/* AspectJ code */
aspect MyAspect {
   void around(int val): set (int MyClass.a) && args (val) {
      if (newval > MyClass.MAX_VALUE)
         proceed(MAX_VALUE) ;
   }
}
```

What we would like to do

- Writing aspects that represent the concern:
  - “parallelise all the loops iterating from 0 to the length of an array of int using MPI”,
  - or “parallelise all the loops iterating over a Collection using Java Threads”.
- Write (aspect) code that does not invade the readability of the numerical code.

Presentation Outline

- Problem and Approach
- Using AspectJ for Parallelisation
- Extending AspectJ for Parallelisation

AspectJ for Parallelisation (I)

- No join point for loops
- Exposing the iteration space as method parameters [HG04]
-
```
void myMethod (..., int iMin, int iMax) {
  for (int i=iMin ; i<iMax ; i++) {...} }
```
-
```
void around(int min, int max):
  call(void *.myMethod(..)) && args(.., min, max) {
    // int t_min, t_max
    new Runnable() {
      public void run() { proceed(t_min, t_max) ; } }
    // execute each instance concurrently
}
```
- Similar aspect for using MPI (if data to be sent exposed as well)

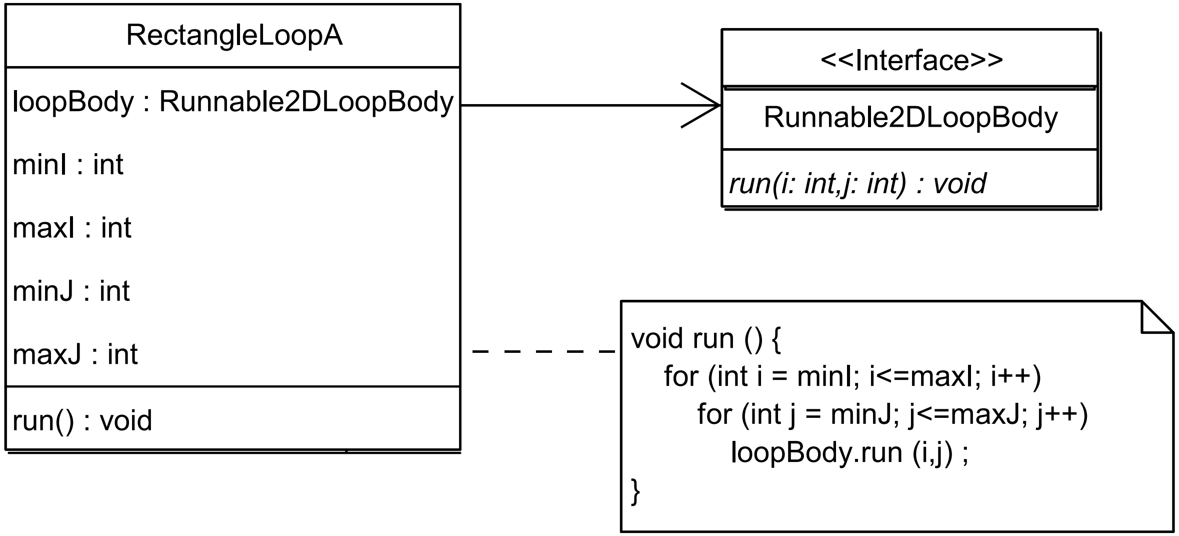
AspectJ for Parallelisation (II)

- AspectJ expects an underlying object-oriented design
- Putting the iteration space outside the method may require substantial refactoring
- Although it works on some examples in the Java Grande Forum benchmark suite, it is almost impossible in others (for example the LU factorisation)

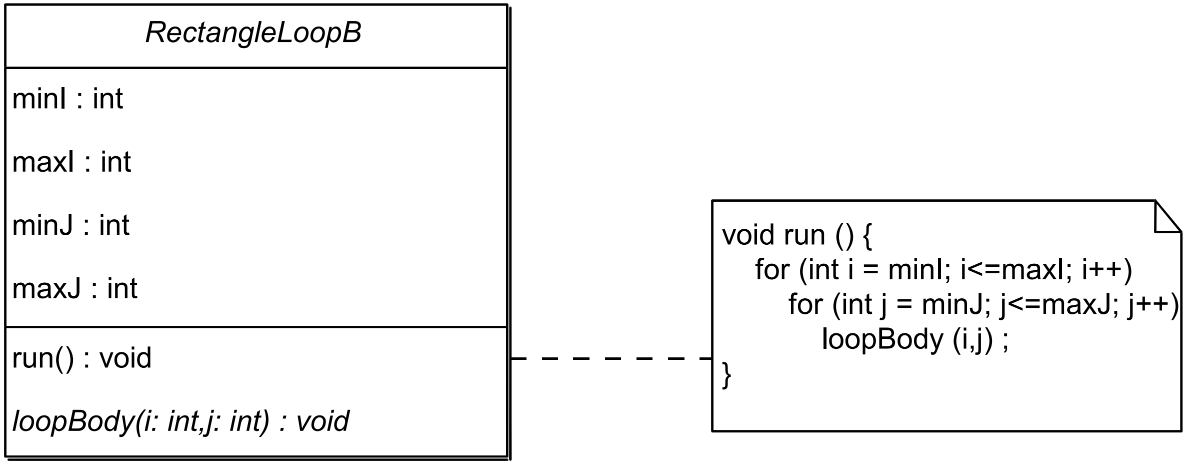
Object-Oriented Models for Loops

- Object-oriented models for “for”-loops [HG04]
- AspectJ can handle these models
- Consist of encapsulating loop information into classes: boundaries and loop-body

Object-Oriented Models for Loops



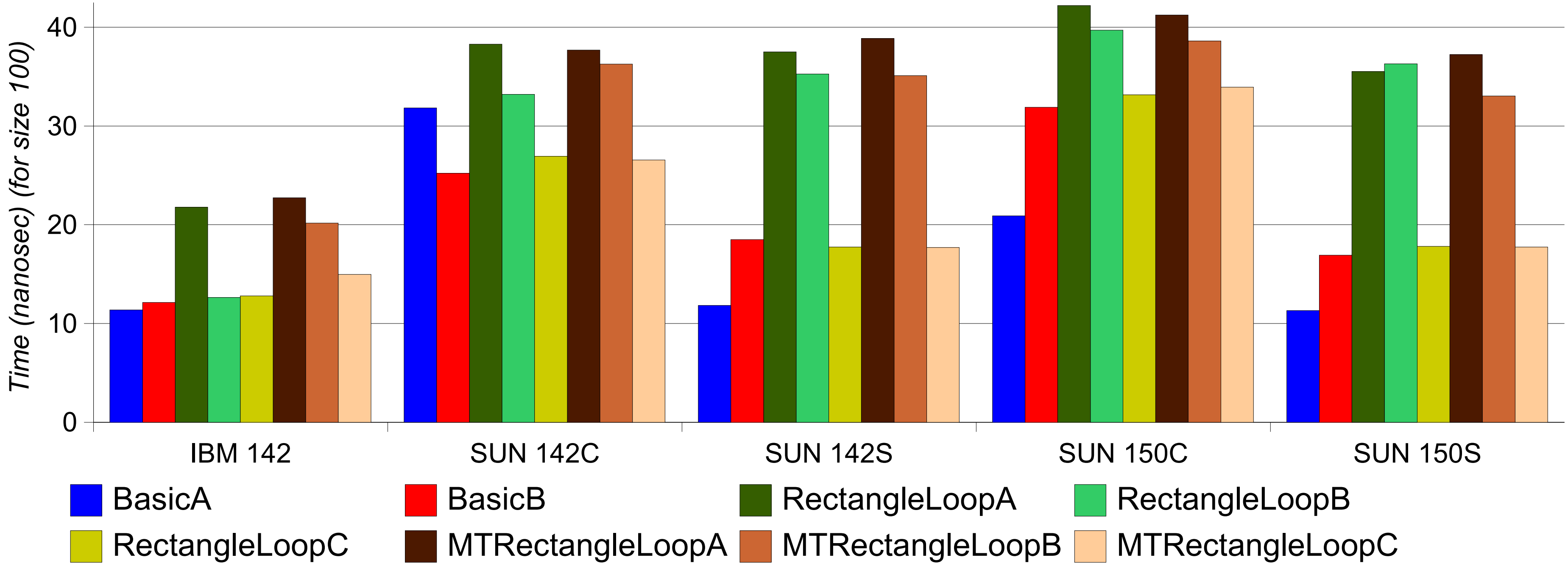
```
void run () {
    for (int i = minI; i<=maxI; i++)
        for (int j = minJ; j<=maxJ; j++)
            loopBody.run (i,j) ;
}
```



```
void run () {
    for (int i = minI; i<=maxI; i++)
        for (int j = minJ; j<=maxJ; j++)
            loopBody (i,j) ;
}
```

Object-Oriented Loops: Overheads

- Performance results depend on the JVM
- Cost of refactoring (here: no parallelism)



Presentation Outline

- Problem and Approach
- Using AspectJ for Parallelisation
- Extending AspectJ for Parallelisation

Join Point for Loops

-
  ```
  TheClass[] array = /* ... */
  for(int i = 0 ; i < array.length ; i++) {
     TheClass obj = array[i] ;
  }
  for (TheClass obj: array) { /* ... */ }
  ```

-
  ```
  Collection c = /* ... */
  for(Iterator it=c.iterator() ; it.hasNext() ;) {
     TheClass obj = (TheClass)it.next() ;
     /* ... */
  }
  for(TheClass obj: c) { /* ... */ }
  ```

Finding loops

- Analysis of the control flow graph
- Finding natural and combined loops
- Based on bytecode representation: it's about recognising the behaviour, not the coding style (e.g. equivalent “while” and “for” loops)

Context Exposure

- Exposing data processed and guiding the execution,
- "Arguments" to the loop,
- Integer range and `Iterator`s,
- Arrays and `Collection`s.
- (Only loop with unique exit nodes to avoid "`break`" statements and irregular iterations)

Context Exposure: Arguments to the loop

- **for** (int i = min; i < max ; i+=stride)
  - **args**(min, max, stride)
- **for** (int i = 0 ; i < array.length ; i+=stride)
  - **args**(min, max, stride, array)
- Iterator it ; **while** (it.hasNext) { it.next() }
  - **args**(it)
- **for** (TheClass obj: collec)
  - **args**(iterator, collec)

Aspects for parallelisation

-
  ```
  void around(int min, int max, int
  stride):
  loop() && args(min, max, stride, ..) {
     /* create runnables */
  }
  ```

- Block scheduling:
  ```
  proceed(tmin, tmax, stride) ;
  ```

- Cyclic scheduling:
  ```
  proceed(min+k, tmax, stride*threads);
  ```

- MPI: access to the array + send/recv

Loop selection

- In AspectJ, the selection is (ultimately) based on a name pattern, for example on the method name or an argument type,
- Loops haven't got names,
- Selection to be made on argument types and on data processed: integer range and Iterators; and especially arrays and Collections. (+`cflow`, `within` and `withincode`)
- `pointcut bytearrayloop(int min,int max,int s,byte[] a): loop() && args(min,max,s,a);`

Implementation using `abc`

- `abc`: AspectBench Compiler (full AspectJ compiler),
- *LoopsAJ*: our extension for `abc` that implements a loop pointcut,
- Analysis capabilities of Soot

Summary

- Parallelisation with AspectJ possible but requires refactoring.
- Join point for loops: meaningful thanks to context exposure, which makes it possible to intervene with the iteration space and data. Refactoring not necessary.
- Both techniques make it possible to have Java base code for numerical concern and aspects for either MPI or Java threads.

References

- Aspect-Oriented Software Development
  http://www.aosd.net/
- [HG04] Harbulot and Gurd. *Using AspectJ to Separate Concerns in Parallel Scientific Java Code*. AOSD 2005
- [HG05] Harbulot and Gurd. *A join point for loops in AspectJ*. FOAL 2005
- [Dij76] Dijkstra. *A Discipline of Programming*.
- [FF00] Filman and Friedman. *Aspect-Oriented Programming is quantification and obliviousness*.
- [KLM+97] Kiczales *et. al.* *Aspect-Oriented Programming*.