JPPAL: Java Parallel Programming Annotation Library

Edgar Sousa
Departamento de Informática/CCTC
Universidade do Minho
Braga, Portugal
edgar@di.uminho.pt

João L. Sobral
Departamento de Informática/CCTC
Universidade do Minho
Braga, Portugal
jls@di.uminho.pt

Abstract

This paper describes an annotation language for parallel programming. This language is completely implemented as a library of aspects, in AspectJ, and mimics the semantics of the OpenMP standard. We discuss the language implementation and the difficulty in implementing certain constructs of the OpenMP standard, as well as possible workarounds. Performance is close to the one obtained with Java threads, but it requires much less programming effort, although, in certain cases, a refactoring to the original code is required.

Categories and Subject Descriptors D.1.3 [**Programming Techniques**]: Concurrent Programming - *Parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features – *Concurrent programming structures*.

General Terms Performance, Design, Standardization, Languages

Keywords Java annotations, OpenMP, multi-core programming, aspect libraries, AspectJ

1. Introduction

The recent rise of multi-core systems increased the demand for parallel programming languages that can explore the full processing power of these types of architectures. Parallel programming languages must allow the programmer to express a set of parallel activities as well as execution constraints among parallel activities (e.g., to impose a specific order of execution to avoid data races). Additionally, specific guidelines may be given to the run-time system in order to improve performance (e.g., to specify how parallel activities are scheduled into available processing resources).

OpenMP [1] is becoming very popular for programming multi-core platforms. It has many nice features that make it attractive for these kinds of platforms:

- all parallelism related constructs can be placed into code comments, making the parallelization easy to understand, as constructs are easy to track;
- parallelization can be performed in an incremental way, starting by a sequential version and developing the parallel version by inserting new code comments;
- parallelization is reversible as directives can be ignored for a strict sequential execution, making it easier to improve domain specific code.

OpenMP is now supported by most mainstream C/C++ and Fortran compilers (e.g., Intel icc, Gnu gcc). Unfortunately, there is no official "binding" of OpenMP to the Java language. Support for OpenMP in Java has been proposed by several authors [2][3]. These proposals follow an approach close to the OpenMP standard by including programming constructs through code comments with predefined sentinels that mark OpenMP directives.

AspectJ has been showed to provide programming abstractions that can effectively encapsulate common concurrency mechanisms into the form of reusable aspects [4]. These reusable aspects can also be used through Java annotations. Moreover, Java annotations are becoming widely used (for instance, Jboss provides a complete set of annotations to generate persistence using hibernate [5]).

Supporting the OpenMP standard in Java exclusively through annotations becomes very attractive and well integrated into the current trend of the language. Moreover, AspectJ enables the development of reusable aspects to associate semantic actions to those annotations.

In this paper we describe an effort to write an annotation-based domain-specific language for parallel computing, fully implemented through a library of aspects, which would replicate the semantics of the OpenMP standard in Java.

The remainder of the paper is organized as follows. The next section presents an overview of the programming philosophy of OpenMP and its main language constructs. Section 3 introduces the JPPAL library and outlines its implementation. Section 4 discuses this work in terms of performance and feasibility of implementing OpenMP abstractions as reusable aspects and Section 5 concludes the paper.

2. OpenMP Overview

In OpenMP [1] parallel activities are specified through a *parallel region*: a code block that is executed by a team of threads. The specific number of threads is selected by the run-time system (usually equals the number of available processing units). By default the code inside a parallel region is executed by ALL threads in the team. Work sharing constructs are used inside a parallel region to schedule work among threads in the team. The most important work sharing construct is the *for* which schedules iterations of a loop among the threads in the team. Other work sharing constructs include *parallel sections*, which define blocks of code to be executed by different threads in the team and the *task* construct which can be used to schedule recursive task creations among threads. Support for *nested parallel regions* can be provided in a particular implementation but their support is not mandatory in the standard.

Figure 1 presents an excerpt of the parallelization of the JGF Series benchmark [6] with OpenMP (this code, as well as other data relative to OpenMP, was taken from the *JOMP* implementation [2] of JGF benchmarks). The comment *omp parallel* specifies a parallel region, which is executed by a team of threads, and the *omp for* specifies that loop iterations are assigned to the team of threads using some default scheduling policy (e.g., by dividing the number of iterations by the number of threads in the team).

```
void Do() {
  ...
  //omp parallel
  //omp for
  for (int i = 1; i < array_rows; i++) {
    TestArray[0][i] = TrapezoidIntegrate(/*.. */);
    TestArray[1][i] = TrapezoidIntegrate(/*.. */);
  }
  ...
}
```

Figure 1. Illustration of OpenMP *parallel* and *for* construct

In certain cases, thread execution inside a parallel region must be constrained in order to ensure correctness. OpenMP supports several synchronization primitives for this purpose. These include: *master* (the block is executed by the master thread); *single* (the block is executed by a single thread) and *critical* (the block is executed in mutual exclusion), *barrier* (creates a synchronization point among all threads in the team), among others.

Additional constructs are required to define the behavior of local/global variables inside a parallel region. The construct *thread-local* defines a variable that is local to a thread during the execution. Additional parameters in the parallel region construct specify how variables, declared outside the parallel region, behave inside the parallel region. Variables can be *shared* or *private*. In the former, the variable is shared by all threads in the team. In the latter each thread in the team gets a local copy of the variable. In this case, a reduction operation can specify how multiple local values of the variable are reduced to a single value, at the end of the execution of the parallel region. Figure 2 presents an example. In this case, the variable *sum* becomes private to each thread in the team and its final value is the sum of the values of all local *sum* variables.

```
  //omp parallel for reduction(sum:+)
  for (i = 0; i < length; i++)
    sum = sum + data[i];
  ...
```

Figure 2. Illustration of *reduction* operation in OpenMP

3. JPPAL

This chapter describes the created OpenMP-like programming interface for Java, in library form, made of reusable aspects. Similarly to OpenMP, this library, implemented using AspectJ, provides annotations for use in an object-oriented way in Java programs. The following subsections describe the library rationale (3.1) and its implementation (3.2).

3.1 Library Overview

The JPPAL relies on annotations, instead of comments, to apply parallelization mechanisms and it is a library instead of a source code transformation tool. The use of annotations has three main reasons: programming with annotations is natural to Java programmers, removes the need to be knowledgeable about aspect-oriented programming, AspectJ and the internal working of the library, and avoids pointcut fragility [7].

The library implements the most used mechanisms found in OpenMP: *parallel*, *single/master*, *critical*, *barrier* and *for*. Each of those mechanisms gave origin to one annotation (except two for barrier). For each of those mechanisms, aspects were created that look for annotation presence to apply the advice.

3.1.1 User Interface

Each construct is applied to code by tagging a method with the intended annotation. The simplest example, to execute a method by several threads (e.g., a parallel region), is shown in Figure 3. The @Parallel annotation specifies that the method will be executed, in this case, by 4 threads.

```
public class Foo {
  @Parallel(n=4)
  public void someMethod(){
  ...
  }
...
}
```

Figure 3. Example of annotation use

3.1.2 Design Decisions

OpenMP is a specification targeted only at C/C++ and Fortran. So, we decided not to mimic the programming constructs as defined by OpenMP because of the differences between those languages and Java. First, mechanisms apply only to methods, as they are the fundamental part of the application interface. Doing so, one can write a subclass and uses the same annotations on the overriding methods to keep the parallelization. Second, we follow the language spirit when dealing with thread access to variables. Class members are shared and not guaranteed to be up to date unless declared volatile; method arguments and local variables are private. Third, we impose refactoring to expose the required context information (e.g., exposing a value as a method parameter), instead of relying on a pre-processing tool to gather the required information. This results in more explicit APIs as opportunities of parallelism and context information became part of it.

Refactoring The library was tested using the Java Grande benchmark suite [6]. Similarly to Harbulot and Gurd [8] we found that refactoring was needed in order to expose the needed pointcuts for parallelization. Since methods are the smaller unit that annotations can apply to, the main code transformation was to replace a block of code with a method, or as we call it, to give it a name. This can be easily done with an IDE tool like Eclipse.

Figure 4 shows the code after applying this refactoring technique to the Series benchmark of Java Grande (previously presented in Figure 1).

```
void Do() {
  ...
  execFor(1,array_rows,1,omega);
}

void execFor(int start, int end, int step,
                                double omega )  {
  for(int i = start ; i < end; i+= step )   {
    TestArray[0][i] = TrapezoidIntegrate(/*.. */);
    TestArray[1][i] = TrapezoidIntegrate(/*.. */);
  }
}
```

Figure 4. JGF Series after refactoring

By moving the computation loop to a method of its own that exposes the variables controlling the loop (start, end, step), an annotation can be applied to it and the advice can get the proper context information.

3.2 Mechanisms Implementation

3.2.1 Parallel

The *Parallel* annotation declares the start of a region to be executed by several threads simultaneously. It has one mandatory argument, n, the number of threads (in case that $n < 1$, it will use the number of processors, gathered with a Java system call).

The annotation can only be applied to methods with no return value. Method parameters are private to each thread. Java semantics applies in the case of object references: the variable is a private reference for an object that is shared by all threads. On the other side, object fields are shared among threads.

Figure 5 shows the base implementation of this aspect that is active when a method tagged @Parallel is called (pointcut *parallelMethod*), creating n threads (lines 16-18). In this case, $(n - 1)$ *Runnable* objects are created and each one will execute the *proceed* AspectJ statement when the *ThreadPool Executor* runs it (lines 20-28). After proceeding with the method call (line 29), the main thread will wait for the completion of other threads (lines 30-38). The aspect also provides a shared storage area to be used by the threads, which stores information about the parallel zone (master id, group size, etc.), a barrier and a lock.

```
1 import java.util.*;
2 import java.util.concurrent .*;
3
4 public aspect ParallelMethod {
5   public ExecutorService exec = null;
6   public ReentrantLock groupLock;
7   public TournamentBarrier groupBarrier;
8   /* Other parallel region state variables */
9    pointcut   parallelMethod ()  :
10      call ( @Parallel   void   *.*(..) )  ;
11
12   void around (final Parallel pm) :
13      parallelMethod () && @annotation (pm) {
14      groupSize = pm.n() >0 ? pm.n():
15            ... // get available processors
16      if( exec == null && groupSize >1)
17        exec = Executors
18              .newFixedThreadPool(groupSize-1);
19      /* Other initializations */
20      for (int i =1; i<pm.n(); i ++)
21        Runnable r = new Runnable () {
22          public   void   run ()  {
23            myid.get();
24            proceed(pm);
25          }
26        };
27        futures.add(exec.submit (r));
28      }
29      proceed (pm);
30      if(groupSize >1)  {
31        try  {
32          for ( Future <? >  f  :   futs )
33            futures.get ();
34        }   catch   ( Exception   e)  {
35            e. printStackTrace ();
36        }
37        futs.clear ();
38      }
39  }
40 }
```

Figure 5. *ParallelMethod* aspect implementation

3.2.2 For

This is the only work-sharing construct currently supported. At this time it is mandatory that the method it applies has a subclass of *Number* as return type. To be able to use the *@For* annotation, the method must have as its first three parameters three ints, meaning (in order): the loop iterator initial value, end value and step; also the method must be called from inside a parallel region.

When a thread executing a parallel region finds a call to an annotated method with *@For*, it will modify the aforementioned parameters, based on their values and its own id, in order to share the work, proceeding with n calls to the method.

The annotation itself has one argument, reduction, that can be NONE, AVG, MAX, MIN, SUM or PROD, specifying how to combine the individual results from each method call into one.

Figure 6 shows this annotation applied to the Crypt benchmark. In this case, no reduction is required, as no return value is generated by the *execFor* method.

```
@Parallel (n =4)
void cipher_idea(byte[] text1, byte[] text2,
                                  int []  key ){
  //...
  execFor(0,text1.length,8, text1, text2, key );
   //...
}

 @For ( reduction = Reduction . NONE )
Number execFor(int st, int en, int step ,
    byte[] text1, byte[] text2, int[]  key ) {

  for (int i=st; i<en;  i+=step )  {
        //...
  }
  return null;
}
```

Figure 6. Using the @For annotation.

3.2.3 Other Mechanisms

BeforeBarrier & AfterBarrier

These annotations have no arguments and cause the threads to synchronize before (after) entering (leaving) the annotated method.

Master & Single

These annotations enforce method execution by only one thread. With *Single*, the first thread entering the method executes it. With *Master*, only the thread with id 0 executes the method. In both cases, all threads synchronize at the end of the method.

Critical

This annotation marks a section of code that can be executed by one thread at a time. No order between threads is guaranteed.

3.2.4 Implementation Limitations

Parallel method

AspectJ supports multiple instances of an aspect by specifying an aspect instance *percflow*, *perthis* or *pertarget*. To support nested parallel regions one could use one aspect per control flow (e.g., write *public aspect X percflow(somepointcut())* and have a different instance in each join point matched by the pointcut. That instance is accessed by calling the static method *X.aspectOf()*.

However, if one thread is created at that point, a call to *X.aspectOf()* on that thread will return a null object, since a different thread context has been created. For this reason, it is not possible to have nested parallel zones, because currently only one aspect per VM aspect can exist in AspectJ, when running in a multithreaded implementation.

For

The method signature is imposed by the mechanism and does not allow for other return types than Number. Also, there is no load balancing at all, since loop iterations are assigned to threads statically. This can be an issue if some loop iterations take longer than others or if the workload can change during execution. A workaround is to specify a number higher than the number of processors on @Parallel annotations and leave the load balancing to the thread scheduling mechanism of the underlying OS.

Single

It is impossible, with aspect-based tools, to implement a non-blocking version of *Single* mechanism. Figure 7 shows an example why it is impossible. When a thread encounters the call to *doIO()*, its impossible, without maintaining a complex flow analysis, to determine, if it should proceed to the call or skip it.

```
@Single
public   void   doIO (){
    /*...*/
}

@Parallel (0)
public   void   method (){
   for (i =0;i<n;i ++) {
       /*...*/
       doIO();
   }
}
```

Figure 7. Single example

4. Discussion

In this section we present and discuss performance results obtained with JPPAL. The results obtained with two benchmarks from the Java Grande suite show that the library does not cause a significant performance loss.

Values were gathered on a dual Intel Xeon E5420 @ 2.50GHz (QuadCore) machine, with 7GB RAM, running CentOS 3.4 kernel 2.6.9-42.0.2.ELsmp x86_64, Sun Java SE 1.6.0_u11 and AspectJ 1.6.3. Time is in seconds. Table 1 and 2 presents performance results. The first line shows the values obtained with the original multithreaded versions and the second one the values obtained with the library.

Table 1 - Crypt execution times (seconds)

Threads	**1**	**2**	**4**	**8**
JFG Size C	8.93	4.39	2.69	1.14
JPPAL Size C	10.32	5.38	2.73	1.41

Table 2 – Series execution times (seconds)

Threads	**1**	**2**	**4**	**8**
JFG Size C	1307.23	808.39	420.50	220.32
JPPAL Size C	1307.00	809.52	419.38	226.31

Being implemented using AspectJ brings, in this particular case, three immediate advantages. First, the library itself could be distributed in binary form, and using Load Time Weaving, the same binary package can be used on all the applications without recompiling it (akin to shared libraries); second, one can write new aspects for new mechanism applied to the same pointcuts specified by the annotations to improve certain functionality; and third, if needed, anyone can change the library and specify pointcuts instead of annotations in case where the access to the application source code is not possible.

Another important fact is that by using AspectJ, we don't need to use other more complex tools to introduce the extra behavior. The alternative to AspectJ would be parsing and transforming the source code.

On the negative side, we have the lack of AspectJ's ability to target loops. Thus, the base code needs to expose in its interface the suitable points of parallelization. At this time, the only way to accomplish that is to move the loop to a method of its own where parameters are arguments to the method, enabling AspectJ advice to capture and change them. To overcome this limitation when using either pointcut or annotation approach, Java could be extended with true named blocks and then AspectJ could target them. Figure 8 proposes a syntax for this that is able to expose local context.

```
int  foo(){
  //statements
  fooLoopName(int  start,  int  max):{
    for(int  i=start;i<max;i++){
      //loop  body
    }
    //after  loop  statements
  }
  //statements
  return  value;
}
```

Figure 8. Named blocks example

With those facilities, it would be possible to apply annotations directly without refactorings. In this case the code expressed on Figure 9 could be used.

```
int  foo(){
  //some  statements
  //more  statements
  @Parallel(n =DEFAULT)
  @For(reduction =Reduction.NONE )
  fooLoopName(int start, int max):{
    for(int i=start;i<max;i++){
      //loop body
    }
    //after loop statements
  }
   //statements
  return  value;
}
```

Figure 9. Inline annotations on named blocks example

5. Conclusion

This work presents a simple way to express parallelization, by using Java annotations to express the parallelization points. The library does not hinder program gains from parallelization. The JPPAL library does not require previous knowledge of AOP or AspectJ to be able to use it, making it a better choice than Java Threads for parallelization.

On the negative side, we have the lack of AspectJ's ability to target loops. Thus, the base code needs to expose in its interface the suitable points of parallelization. At this time, the only way to accomplish that is to move the loop to a method of is own where these parameters are arguments to the method enabling AspectJ advice to capture and change them (applying the refactoring approach shown on subsection 3.1).

To overcome this limitation, when using either pointcut or annotation approach, Java could be extended with true named blocks and then AspectJ could target them.

As future work the JPPAL library can expand to implement more mechanisms, including mechanism in OpenMP 3.0.

Acknowledgments

This work was supported by the project AspectGrid: Pluggable Grid Aspects for Scientific Applications (GRID/GRI/81880/2006) and Parallel Refinements for Irregular Applications (UTAustin/CA/0056/2008) both funded by Portuguese Fundação para a Ciência e Tecnologia and European funds (FEDER).

References

[1] www.openmp.org

[2] Bull J., Kambites M., JOMP—an OpenMP-like interface for Java, Proceedings of the ACM conference on Java Grande, New York, USA, ACM, 2000.

[3] Klemm M., Veldema R., Bezold M., Philippsen M., A Proposal for OpenMP for Java, OpenMP Shared Memory Parallel Programming Lecture Notes in Computer Science, vol. 4315 (International Workshops IWOMP 2005 and IWOMP 2006), Springer-Verlag, 2008.

[4] Cunha, C., Sobral, J., Monteiro, M., Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms, Proceedings of the 5th international conference on Aspect-Oriented Software Development, Bonn, Germany, ACM, 2006.

[5] http://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single

[6] Smith, A., Bull, J., Obdrzálek, J., A Parallel Java Grande Benchmark Suite, Proceedings of the 2001 ACM/IEEE conference on Supercomputing, Denver, USA, November, ACM, 2001.

[7] Koppen, C., Störzer, M., PCDiff: Attacking the fragile pointcut problem, First European interactive workshop on aspects in software (EIWAS), Berlin, Germany, 2004.

[8] Harbulot, B., Gurd, J., Using Aspectj to separate concerns in parallel scientific Java code, Proceedings of the 3rd international conference on Aspect-oriented software development, New York, USA, ACM, 2004.