Implementing an OpenMP-like Standard with AspectJ

Bruno Medeiros
Universidade do Minho
Braga, Portugal
brunom@di.uminho.pt

João L. Sobral
Universidade do Minho
Braga, Portugal
jls@di.uminho.pt

Abstract

This paper presents an aspect-oriented library, coded in AspectJ, that aims to mimic the OpenMP standard for multicore programming in Java. Building the library on top of AspectJ intrinsically supports the sequential semantics of OpenMP. The library enables the use of parallelism related constructors in object-oriented systems due to better compatibility with inheritance, making it more suitable to introduce parallelism into object-oriented frameworks. However, it requires more program refactoring than OpenMP directives.

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 Algorithms, Performance, Design, Languages

Keywords Parallel programming, Aspect-oriented programming, OpenMP

1. Introduction

Multicore systems are widely available in desktop machines and it is expected that the number of cores on those systems will continue to increase over the next decades. Contrary to previous evolutions in computer architectures, multicore systems require new software programming techniques to support parallel task's specificities. This adds extra complexity to the programmer's task since he must be concerned with core (e.g, domain) functionality implementation and with techniques to effectively exploit parallelism.

OpenMP [1] is becoming the effective standard for multicore systems. In OpenMP parallelism is expressed as a set of directives. The standard is currently supported by the main compilers for Fortran and C/C++ (e.g., GNU gcc). One important feature of the standard is the support for the sequential semantics: it is possible to develop parallel programs whose sequential execution is correct, by ignoring the OpenMP directives. This provides two main benefits: i) the base functionality can be developed by ignoring the OpenMP directives making it easier to develop domain specific code; and ii) incremental development is possible by starting with a sequential program and progressively making refactoring and introducing OpenMP directives.

MISS'13, March 25, 2013, Fukuoka, Japan.

The Java language is one of the most popular programming languages due to its object-oriented features, such as inheritance and polymorphism. Unfortunately, the OpenMP standard does not support Java. There were some attempts to extend the standard, such as JOMP [2], but those do not integrate well with the object-oriented philosophy. One of the most problematic aspects is the intrinsic conflict between inheritance and concurrency constrains: parallelism directives may not be retained across the inheritance chains. This conflict was identified a long time ago and reported as the inheritance anomaly [3]. Support for parallelism in object-oriented frameworks built on top of Java becomes more complex since the methods implementing parallelism concerns can be accidentally overridden in concrete framework instance.

AspectJ is a promising language to support parallelism in Java due to its power to encapsulate concurrency concerns [4]: sequential semantics is intrinsically supported since aspects can be unplugged. Moreover, aspects can act on joinpoints defined by Java interfaces, becoming a promising approach to introduce parallelism into object-oriented frameworks as those rely heavily on the usage of interfaces as extension points [5]. This paper describes an ongoing effort to develop an AspectJ library to support an OpenMP-like standard in Java. The research challenges are:

- How to implement OpenMP directives in AspectJ?
- How to improve the integration of parallelism constructs into object-oriented languages?
- What kind of refactoring is required in order to introduce parallelism?

The rest of the paper provides a short overview of the OpenMP standard, describes the library already developed and discusses the approach taken.

2. OpenMP Overview

OpenMP is based upon *parallel regions*, *work-sharing* constructs, *synchronisation* and *data environment* directives.

The key construct in OpenMP is the parallel region. In the OpenMP model execution starts with a single thread. When that thread encounters a parallel region it becomes the master thread of a newly created team of threads. All threads in the team will execute the parallel region and synchronise at the end of the parallel region. The number of threads in the team is implementation dependent (in most cases it is equal to the number of the cores in the system). Each thread in the team has a unique thread identification (available through a library call) that can be used to assign different tasks to each thread in the team.

Work-sharing constructs are used in a parallel region to divide the work among the threads in the team. The *for* directive divides the range of loop iterations among threads. This directive supports several alternatives to schedule loop iterations among threads in the

team. A static scheduling assigns loop iterations using each thread identification, while a dynamic scheduling assigns loop iterations during execution on a first serve basis. The *sections* directive specifies multiple code blocks that are executed by different threads in the team. The *task* directive specifies a block of code that executes in parallel with the thread that encountered the block. It is mostly used for applications with recursive tasks.

Frequently a parallel region consists of a single work-sharing region (e.g., a single *for*). For those cases OpenMP provides combined directives to support parallel regions of a single *for* or *section*.

Synchronisation directives control (e.g., synchronise) the execution of a team of threads. The *master* directive indicates a code section that is only executed by the master thread. The *single* directive indicates that a single thread should execute the given block. Other directives include a *critical* section and a *barrier*. The former enforces mutual exclusion generally to avoid data races, the latter enforces a barrier point where all threads in the team should arrive before proceeding. It is possible to enforce a specific section of a loop to execute in the logical order (iteration space order) using the *ordered* construct.

Data environment directives control the data scope inside a parallel region. Data is either shared by all threads in the team or private to each thread. Shared data supports clauses *firstprivate* and *lastprivate* to control, respectively, the initial and final data values (e.g., when entering in the region and after region execution). A *reduction* clause specifies how to combine thread private values into a single value at the end of a given region.

3. The Aspect Library

Encapsulating OpenMP directives into a library of aspects provides two benefits: i) no changes to the base code are required in order to apply a mechanism; ii) a mechanisms can act upon all classes implementing a given Java interface. The former supports the sequential semantics of the OpenMP standard and also provides support for incremental development. The latter, provides support for extensible object-oriented frameworks since an aspect can be attached to interfaces that can have impact on classes that are later added to the framework.

The OpenMP standard requires that every directive should be applied to a single statement. In C/C++ this often requires enclosing the desired code block into brackets in order to group multiple statements into a logical block.

In the developed OpenMP aspect library (OmpLib) each mechanism acts upon a set of method calls in the base program (e.g., joinpoints). Thus multiple statements must be grouped by moving those statements into a method, making it possible to assign a unique name to a code block in order to ”plug” the required aspects. Grouping statements into methods is also more consistent with the object-oriented philosophy: concurrency constructs are applied at object boundary and thus, can be retained when the method is overridden in subclasses. Moreover, many modern IDEs, such as the Eclipse, provide support for this kind of refactoring. In modern systems this refactoring does not impose any performance penalty since method calls can often be automatically inlined by the compiler, avoiding the overhead of an additional method call.

A consequence of the previous rule is the refactoring of *for* loops into methods. In the current library those loops must be implemented by a method whose first three arguments are, respectively, the loop variable start value, end value and incremental step. Note that OpenMP also enforces several constrains in the supported *for* loops.

The developed aspect library is based on abstract aspects. Thus, in order to apply a given aspect to a concrete code base developers must extend the abstract aspect to provide application-specific pointcuts. The next subsections discuss the implementation of several mechanisms.

3.1 Parallel Region

The parallel region is the key abstraction in OpenMP to introduce parallelism since it creates a team of threads to execute a given code section in parallel. In the developed OmpLib a parallel region is given by the context of a method execution: the method call is executed in parallel by a new team of threads. The *parallel region aspect* creates the required additional threads in the team and each newly created thread will execute the original method call (by calling *proceed*).

```
01: protected pointcut parallelRegion() :
02: (
03:  call(// Method)
04:  && within(//Class name)
05: );
06:
07:
08: void around() : parallelRegion(){
09:  // Create a local barrier
10:  Thread threads [] = new Thread [numThreads];
11:  for(int i = 1; i < numThreads; i++){
12:     Runnable t = new Runnable(){
13:        public void run(){
14:        try{
15:            proceed();
16:            // call barrier
17:            } catch(..){..}
18:      }  }
19:    threads[i] =  new Thread(t);
20:    threads[i].start();
21:  }
22:  proceed();
23:  // call barrier
24:  // Terminate threads
```

Figure 1. Parallel region.

If the programmer wants to have a given method *f()* running in parallel, he would fill the parallel region pointcut (line 01 of fig. 1) with the signature of the method *f()*. When the aspect *parallelRegion()* intersects the method *f()* call, it creates a new team of threads (line 10) and a runnable with a task (line 15), in this case the method *f()*. The thread master will assign the created runnable to the each thread (line 19) and start its execution. After all threads have started their tasks, the master thread will also start its task (line 22). After finishing its task each thread calls a barrier which will wait for the remaining threads. When all threads' tasks finish the master thread ends threads execution (line 24).

The parallel region aspect must also maintain global information about the parallel region. Specifically, it must assign a unique id to each thread in the team starting in zero (the master thread). Moreover, the OpenMP standard allows the implementation of nested parallel regions. Nesting complicates the aspect implementation since the a thread can become the master of different parallel regions during the execution. Moreover, there is data specific to each parallel region (e.g., the number of threads in that team and a barrier shared by all threads in the team). Thus, in order to support nested parallel regions the aspect must keep track of the entrance/exit in each parallel region. A new context, shared by all threads, is created when entering into a nested parallel region (lines 09-10) and removed when exiting the parallel region.

The OpenMP standard supports an *if* clause in the parallel region directive. This clause provides a mechanism to restrict the creation of a team of threads to certain conditions. This clause can be trivially implemented by the *if* pointcut designator.

3.2 Work-sharing Constructs

The library provides three implementations of the *for* construct: static (in round-robin or block manner), dynamic and guided scheduling. Each implementation is tuned for one specific type of loop scheduling. However, dynamic and guided are implemented using a similar strategy.

Static scheduling divides the range of loop iterations statically among the threads in the team. In the OmpLib the iteration range is exposed in the three first method parameters. The aspect implementation calls *proceed* changing those method parameters to thread specific values, in order to assign different iterations of the loop space to each thread. This implementation provides the lowest overhead since the scheduling is statically performed (e.g., loop ranges are statically assigned in the code).

```
01: pointcut parallelfor(int a, int b, int c):
02: (
03:     call(private forMethod (int, int, int, ...))
04:     && args(a,b,c,..)
05: );
06:
07:
08: void around (int begin, int end, int inc)  :
09:     parallelfor(begin,end,inc){
10:     int new_begin = //
11:     int new_end =   //
12:     int new_incr = //
13:
14:     proceed(new_begin, new_end, new_incr);
15:
16:}
```

Figure 2. Static for.

The code illustrated in figure 2 is a simplified view of the *for* work-sharing with static scheduling. When a programmer wants a given loop iteration space to be split across different threads, he will create a method to represent the *for*, as already described. Then, he will fill the *parallelfor* pointcut with the name of the method resulting from the design rule application. When the *for* method is called by each thread, the library will intercept this call, and assign a range to each thread (line 14 of figure 2). The set of iterations assigned to each thread in the team is computed based on the thread id and on the type of static scheduling selected (round-robin or block).

The dynamic *for* works as follows: intercepts the *for* method, calculates the number of iterations within the *for* (line 02 of figure 3) and assigns a initial task to each thread (line 03). While there are tasks to perform (line 05), threads will execute them (line 07) and request for more (line 08). Each thread, after finishing its work, will call a barrier. The method *getTask()* stores in a local variable the current *for* iteration, incrementing the iteration number at each call, and returning a value that corresponds to the task that the thread will execute. To ensure that the same task will not be assigned to different threads, a *synchronized* Java routine is used. This routine guarantees that the synchronized block of code is only executed by one thread at the time. Guided scheduling differs from dynamic scheduling in the way that tasks are initially created and executed: initially larger tasks are generated and smaller tasks are used to balance the work assigned to each thread near the end of the loop execution.

The *section* construct is implemented in a similar way to a static scheduling: each thread in the team executes a different method call.

The *task* work-sharing construct is not yet implemented.

```
01: void around(int a,int b,int c) : dynamicfor(a,b,c){
02: int number_of_tasks = (b-a)/c;
03: int iteration = getTask();
04:
05: while(iteration < number_of_tasks)
06:     {
07:     proceed(iteration,iteration+chunk,incr);
08:     iteration = getTask();
09:     }
10:
11: // call barrier
12:
```

Figure 3. Dynamic for.

3.3 Combined Constructs

The OpenMP standard supports several combined constructs (e.g. a single directive for a *parallel region* and *for* work-sharing). In the OmpLib those combined constructs can be implemented by creating a new abstract aspect enclosing several aspects as inner aspects. For instance, a *parallel for* can be implemented by creating an abstract aspect encompassing both the *parallel region* and *for* aspects as inner aspects.

3.4 Synchronisation Constructs

Critical and *barrier* constructs are implemented by surrounding a *proceed* with the corresponding primitive synchronisation. In the OpenMP standard those constructs can optionally provide a name in order to created multiple instances of a mechanism (e.g., a barrier with a given name). In the OmpLib each instance of the abstract aspect uses a different lock/barrier. Thus, a behavior similar to OpenMP can be obtained by creating a concrete aspect for each context (e.g., name). Thus, the set of joinpoints specified by the concrete pointcut will share an instance of the synchronisation mechanism.

The *master* and *single* mechanisms share the implementation strategy. In the former, only the thread with id 0 will call *proceed*, while in the latter the call *proceed* is executed by the first thread performing the method call.

The *ordered* clause in the *for* work-sharing is implemented in a way similar to *single* . Each thread in the team will wait until the thread with the previous id has reached the region.

3.5 Data Environment

The OpenMP standard supports shared and private variables. When entering into a logic block (e.g., parallel region or work-sharing construct) each variable defined outside that region can become private to each thread or can be shared among all threads in the team.

In the OmpLib those regions are always method bodies and thus, only object (data fields) may be shared among threads. The default behaviour in OmpLib it to share those objects among all threads in the system.

In order to provide thread private object data fields (i.e., thread local values) accesses to those fields is required to be performed through getter and setter methods. Thus a field with name XXX is accessed through methods named getXXX and setXXX. This is required in order to: i) provide the required joinpoint when reading and writing the data field; ii) generate different joinpoints for each data field. The latter is necessary in order to differentiate accesses to different data fields.

The OmpLib provides an aspect that implements a thread local variable by intercepting calls to these getter and setter methods. One aspect extension is required for each private data field, specifying one pointcut for the getter call and another for the setter call. This is required in order to avoid the use of refection which would

impose a prohibitive overhead. To use a single aspect for all thread local values would require the usage of thisJointPoint to differentiate among getters/setters of different data fields and an additional map to associate each getter with the corresponding setter method.

The initial value of a private field can be gathered from the context surrounding the region (*firstprivate* clause in OpenMP). For this purpose the concrete aspect must provide a way to retrieve that value (e.g., a method call that gets/sets the non-private value).

The OpenMP standard supports a *reduction* operation that merges all thread private values into a single value that will be visible outside the region. The reduction operation is provided by implementing an abstract method providing the reduction operation.

3.6 Composing Data Environment with Work-sharing

Private and reductions can be composed with both parallel region and work-sharing constructs. The AspectJ implementation enables the reutilisation of a single implementation of data environment clauses in all mechanisms. Composition is simply attained by using the same pointcuts in both implementations.

4. Case Study

The OmpLib was validated by implementing all benchmarks from the JGF implementations [6]. This case study focus on the MD benchmark. MOLDYN (MD) is a molecular dynamics simulation algorithm where a set of particles interact under a Lennard-Jones potential [7]. There is a force acting on each particle that will affect its position and velocity. The force calculation (figure 4) is the big part of the computation time, since it involves two loops: an outer loop on all system particles (line 03 figure 4) and an inner loop (line 04) which ranges from the current particle till the total number of particles. Every inner loop iteration updates the force between particle pairs within a given radius. At the end of the loop each particle updates it own force [6] and the process is repeated for a number of time steps.

```
01: void forceCalculation(...){
02:
03:  for(int i = 0; i < NUM_PARTICLES;i++)
04:   for(int j = i + 1; j < NUM_PARTICLES; j++){
05:    d = distance_between(i,j);
06:    if(d < radius){
07:      apply_3_newtons_law(j); // Force j -> i
08:      acumulated_forces = // Force i -> j
09:    }
10:  updateForce(acumulated_forces,i);
11:}
```

Figure 4. MOLDYN: Force calculation.

In order to understand the granularity level of the tasks that will be performed by threads we focus on force calculation function. The application time profiling identified this function as the most time consuming task. Thus, the calculation of the interaction between a particle and the remaining particles in the system became a parallel task by transforming the outer loop into a parallel for. Furthermore, there is a data dependency between parallel tasks, since a race condition could happen during third Newtons Law application (line 07 figure 4) and during the particle force update (line 10). In both situations there is the possibility that one thread is updating the force of a particle that was assigned to another parallel task. Therefore mutual exclusion must be ensured, so that each thread can complete its update and leave the data in a consistent state for others.

The first step to parallelise the force calculation was to apply the *for* design rule to the outer loop, the result is presented in figure 5.

```
01: void forceCalculationFor(int begin,int end,int step){
02:   for(int i = begin; i < end; i+=step)
03:    for(int j = i + 1; j < NUM_PARTICLES; j++){
04:       ...
05:       }
06:  updateForce(acumulated_forces,i);
07: }
08: ...
09: void moldyn_simulation(...){
10:  ...
11:  forceCalculationFor(0,NUM_PARTICLES, 1, ...);
12:
13:}
```

Figure 5. MOLDYN: For rule design application.

After applying the design rule, it is necessary to fill the pointcuts responsible for the parallel region (line 02 of figure 6) and the scheduling used in the parallel *for* (line 06 of figure 6) routines of the OmpLib. Finally, it was also filled the pointcut to ensure the mutual exclusion (line 11 of figure 6)

```
01: public aspect MOLDYN extends OmpLib
02:   pointcut parallel_region () :
03:   call(static void forceCalculationFor(..))
04:   && within(//MOLDYN);
05:
06:    pointcut for_block (int begin,int end,int inc) :
07:    call(static void parallelfor(int,int,int,..))
08:    && args(begin ,end,inc ,..)
09:    && within(//MOLDYN);
10:
11:    pointcut critical() :
12:    call     (// apply_3_newtons_law) ||
13:    call (//updateForce(acumulated_forces,i));
14:
```

Figure 6. MOLDYN: Filled pointcuts.

5. Results

This section compares the performance of Java threads version (i.e., shared memory) MOLDYN benchmark. The evaluation was performed by a series of tests using two machine configurations: i) AMD Opteron Processor 6174, 2.2 GHz, 12 MB Cache L3, with 24 cores; ii) Intel Nehalem CPU Hex-Core, 2.67 GHz, 12 MB Cache L3, corresponding to 12 cores and 24 threads. Each test performed 25 different measurements, considering the median of these measurements.

The reported results include two distinct parallel versions, named intrusive and non-intrusive (OmpLib). In the non-intrusive version, these routines were implemented using the proposed OmpLib (our approach), while on the intrusive version they were manually inserted on the base code (this version is provided in the JGF implementation). Finally, the time difference between both approaches was registered, thus obtaining the AspectJ approach overhead. The absolute values are illustrated in figures 7 and 8.

Note that the same static thread scheduling was used in both approaches (intrusive and non-intrusive).

Analysing the results for the static scheduling in both AMD and HEX machines the proposed approach, for the MD parallelisation has approximately the same performance. The difference between non-intrusive and intrusive approaches vary from -0,636 to 1,32 seconds representing respectively -2,58% and 2,50% of the intrusive execution time in the Hex machine benchmarks, and from -1,214 to 1,8445 seconds (-2,01% and 2,81 %) in the AMD machine.

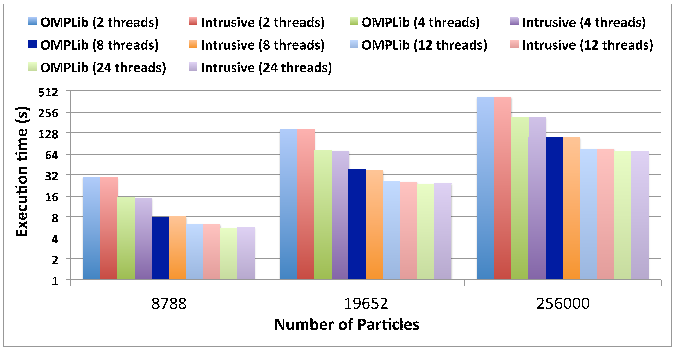

Figure 7. Hex : Static for overhead.

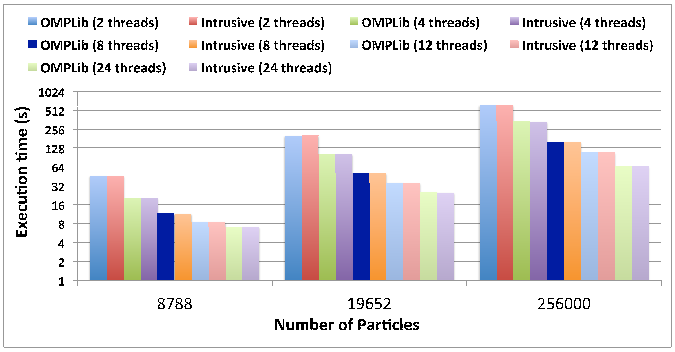

Figure 8. AMD : Static for overhead.

6. Conclusion

This paper presents an ongoing effort to develop an aspect library that mimics the OpenMP standard. This library better fits into object-oriented systems, since parallelism concerns are specified at class interface level. Although, the library requires some degree of refactoring in order to be applied to a given code base. The library has been tested in several benchmarks from the Java Grande Forum [6] and provides a performance similar to implementations using traditional techniques (e.g., Java threads). Currently the library provides most of the OpenMP functionality to Java programmers. Future work includes the implementation of task-related constructs, that have been added more recently to OpenMP and the implementation of strategies to reduce the programming overhead of refactoring loop constructs.

Acknowledgments

This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-010152 and FCOMP-01-0124-FEDER-011413.

References

[1] www.openmp.org

[2] J. M. Bull and M. E. Kambites. 2000. JOMP-an OpenMP-like interface for Java. In Proceedings of the ACM 2000 conference on Java Grande (JAVA '00). ACM, New York, NY, USA, 44-53. http://doi.acm.org/10.1145/337449.337466

[3] S. Matsuoka and A. Yonezawa. 1993. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Research directions in concurrent object-oriented programming, Gul Agha, Peter Wegner, and Akinori Yonezawa (Eds.). MIT Press, Cambridge, MA, USA 107-150.

[4] Carlos A. Cunha, J. Sobral, and M. Monteiro. 2006. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In Proceedings of the 5th international conference on Aspect-oriented software development (AOSD '06). ACM, New York, NY, USA, 134-145. http://doi.acm.org/10.1145/1119655.1119674

[5] J. van Gurp and J. Bosch. 2001. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. Softw. Pract. Exper. 31, 3 (March 2001), 277-300. http://dx.doi.org/10.1002/spe.366

[6] L.A. Smith, J.M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In Proceedings of the 2001 ACM/IEEE conference on Supercomputing, Supercomputing '01, page8, New York, USA, November 2001. ACM. http://doi.acm.org/10.1145/582034.582042

[7] J. E. Lennard-Jones. Cohesion. In Proceedings of the Physical Society, volume 43, 461-482, 1931. http://dx.doi.org/10.1088/0959-5309/43/5/301