AOmpLib: An Aspect Library for Large-Scale Multi-Core Parallel Programming

Bruno Medeiros
Departamento de Informática/CCTC
Universidade do Minho
Braga, Portugal
brunom@di.uminho.pt

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

Abstract— **This paper introduces an aspect-oriented library aimed to support efficient execution of Java applications on multi-core systems. The library is coded in AspectJ and provides a set of parallel programming abstractions that mimics the OpenMP standard. The library supports the migration of sequential Java codes to multi-core machines with minor changes to the base code, intrinsically supports the sequential semantics of OpenMP and provides improved integration with object-oriented mechanisms. The aspect-oriented nature of library enables the encapsulation of parallelism-related code into well-defined modules. The approach makes the parallelisation and the maintenance of large-scale Java applications more manageable. Furthermore, the library can be used with plain Java annotations and can be easily extended with application-specific mechanisms in order to tune application performance. The library has a competitive performance, in comparison with traditional parallel programming in Java, and enhances programmability, since it allows an independent development of parallelism-related code.**

Keywords- Java; Aspect-oriented programming; parallel programming, OpenMP

I. Introduction

The wide availability of multi-core systems reinforces the need for parallel programming languages that improve programmer productivity. It is expected that in the next decades the number of cores on those systems will continue to increase. Contrary to previous evolutions in computer architectures, multi-core systems require new software programming techniques to support the management of parallelism-related code. This adds extra complexity to the software development process since programmers must be concerned with core (e.g., domain) functionality implementation and with techniques to effectively exploit parallelism across target platforms.

Introducing new parallel programming languages makes the adaptation process more difficult since programmers must rewrite the huge base of legacy code (e.g., sequential code). Extending an existing language with new parallel programming constructs (e.g., using a sequential-like language as the base language) provides a smooth transition to multi-core enabled applications and promises to make the migration of the huge base of legacy sequential code easier.

OpenMP is an example of such kind of approach. First, parallelisation can be performed incrementally, by progressively including pragma annotations into the base code. This supports the parallelisation of legacy code with minor changes to the code base. Second, the parallelism-related code can be localised into well-defined statements (the pragma annotations) making it easier to identify them, improving understandability and maintainability. Third, it is possible to develop programs with a sequential semantics (i.e., programs can be valid if annotations for parallelisation are ignored). These features can help development, since most of the domain-specific code can be developed and tested by running applications sequentially, even in later development stages.

However, OpenMP has several limitations for large-scale applications: i) pragmas can become spread across many application modules and, thus, modular programming is not feasible once the design decisions concerning the parallelism exploitation strategy are not encapsulated into well-defined modules; ii) more sophisticated parallelisation strategies must resort to explicit calls to OpenMP library routines and to additional parallelism-specific code. For instance, programmers may have to resort to threads id to provide application-specific loop scheduling or to attach code for performance tuning. This defeats the support for incremental development and sequential semantics: programmers must invasively include new code into base programs, code than can be difficult to unplug to enforce sequential execution.

Another OpenMP hurdle is the lack of support for the Java language: some early efforts (like JOMP [1]) are no longer supported. Moreover, the OpenMP programming approach is not suited for object-oriented applications since it is not compatible with features of modern object oriented languages that are extensively used in large-scale Java applications, namely annotations, interfaces, abstract classes and inheritance. The root of this problem 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 [2]. Support for parallelism in object-oriented frameworks built on top of Java becomes even more complex since the methods implementing parallelism concerns can be accidentally overridden in concrete framework instances.

The AOmpLib approach aims to promote a smooth transition of Java programmers to multi-core programming by providing a library of pluggable modules that enable parallel execution. Java programmers can start by writing domain specific code in plain Java (or reuse existing sequential code) and later can compose the base program with aspect modules from (or extended from) the AOmpLib in order to enable parallel execution.

The AOmpLib is inspired in OpenMP but relies on Aspect-Oriented Programming (AOP) techniques [3] to overcome OpenMP limitations for large-scale Java applications by providing:

- a library of aspects modules implementing most common used OpenMP abstractions, which can be composed with a base program either through plain Java annotations or through AspectJ pointcuts; sequential semantics and incremental development are intrinsically supported since aspects can be (un)plugged to/from a given base program at any time;
- better compatibility with inheritance and with the usage of Java interfaces by using AOP pointcuts to bind aspect modules to interfaces;
- an easy way to attach new aspects into a given base program, enabling the development of application specific aspects in order to tune performance. Moreover, the library can be easily extended/changed to handle application specific mechanisms.

The main benefit of the AOmpLib approach is the support for incremental and independent development of large-scale, multi-core enabled Java applications. Programmers start with platform-independent Java code and later implement and compose aspect modules in order to introduce parallelism in a non-intrusive manner. The AOmpLib provides a library of aspect modules that mimics OpenMP constructs and that can be reused/tuned across applications and target platforms.

The next section provides a more detailed discussion of the problems of traditional parallel programming for large-scale object-oriented applications. Section III describes the execution model, the supported programming abstractions and presents a simple example of the usage of the library. Section IV presents an overview of the current library implementation and Section V presents evaluation results. Section VI compares the work with other research efforts and Section VII concludes the paper.

II. Limitations of Traditional Approaches

To illustrate the problem of traditional parallel programming techniques this section uses the MolDyn benchmark, an example taken from the Java Grande Forum (JGF) [4]. The MolDyn benchmark provides a computational kernel typical in molecular dynamics simulation codes [5][6]. The JGF benchmark provides a sequential and parallel version (based on Java threads, a.k.a., JGF MT) of each benchmark.

Figure 1 shows a simplified class diagram. The class MD contains all the information about the simulation. The method *runiters* implements the iterations of the simulation. The class *Particle* contains 9 variables representing the position, velocity and force for all coordinates in a three dimensional space. The method *force* calculates the force of a specific particle with the remaining particles.

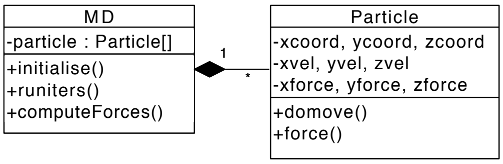


Figure 1. Simplified MolDyn class diagram

Figure 2 shows a code snippet of the base program (i.e., sequential version). For a pre-defined number of steps (*movemx*) it updates the particles position and computes the new force on each particle based on these new positions.

```
class md {
 static particle one [] = …;
 int movemx = …;

 int mdsize = …;

 void runiters(){
   for (move=0; move<movemx; move++) {
     for (i=0; i<mdsize; i++) {
       one[i].domove(); /* move particles */
     }
     for (i=0; i<mdsize; i++) {
       one[i].force(i);  /* compute forces */
     }
     … // other simulation steps
   }
 }
}

class particle {
 double xcoord, xvelocity, xforce;

 void domove() {
    xcoord = xcoord + xvelocity + xforce;
    … // other computations
 }
 void force(int x) {
   for (i=x+1; i<mdsize; i++) {
      forcex = …
      md.one[i].xforce = md.one[i].xforce - forcex;
      … // other computations
   }
 }
}
```

Figure 2. JGF MolDyn sequential implementation

In order to enable parallel execution the JGF MT benchmark (invasively) changes the base implementation of both *MD* and *Particle* classes. Figure 3 shows a simplified code that illustrates the JGF parallelisation. In the approach taken all threads execute the *runiters* method, which was moved to the *run* method of the *mdRunner* class. Thread creation code is shown in red. In this approach each thread will compute the forces acting on a subset of particles. The code implementing this decision is shown in blue (in this case a cyclic load-distribution approach was taken). There is a data race in the computation of the particles' force fields due to third's Newton law use (i.e., forces are symmetric). To deal with this data race, each thread uses a local force array, requiring a change in the particle implementation (shown in green).

```
 class md {
  static particle one [] = …;
  int movemx = …;

  int mdsize = …;
  int nthreads =…;

  void runiters(){

   /* spawn threads */
   for(int i=1;i< nthreads;i++) {
      thobjects[i] = new mdRunner(i);
      th[i] = new Thread(thobjects[i]);
      th[i].start();
    }
    ...
   for(int i=1;i< nthreads;i++) {
      th[i].join();
    }
}

 class mdRunner implements Runnable {
  int id;

  void run() {
    for (move=0;move<movemx;move++) {
      for (i=0;i<mdsize;i++) {
        one[i].domove(i);  /* move particles */
      }
      /* cyclic distribution */
     for (i=id;i<mdsize;i+=nthreads) {
        one[i].force(i);  /* compute forces */
      }
  }
 }

 class particle {
  double xcoord, xvelocity,  xforce;
  double [] sh_forcex;
  double [][] sh_forcex2;

  void domove(int part_id) {
     xcoord = xcoord + xvelocity + sh_forcex[part_id];
  }
  void force(int x) {
    for (i=x+1;i<mdsize;i++) {
      //md.one[i].xforce = md.one[i].xforce - forcex;
      sh_forcex2[id][i] = sh_forcex2[id][i] - forcex;
    }
  }
 }
```

Figure 3. JGF MolDyn multi-thread implementation

This example illustrates the problems of introducing parallelism into large-scale Java applications: parallelism related code is scattered across many base classes (*MD* and *Particle* in this case). This results in invasive changes to the base program and the design decisions become scattered across multiple code blocks (e.g., red, blue and green code reflect three design decisions: task creation, work distribution and dependence management). This kind of approach is not usable in systems with a large number of classes. Note that OpenMP could partially solve the issue, but it would not avoid the code in green.

This kind of approach also pre-empts the usage of object-oriented mechanisms. For instance, there are many types of forces among particles, which can be managed by extending the class particle overriding the method *force*. In those cases the parallelism related code could be lost if a new implementation is provided for that method.

A more complex problem is the usage of Java interfaces. The particle class could be an interface with many implementations (this is for instance the case of the LAMMPS package that provides many different Particle implementations where the user selects the implementation that best suits his needs). In such cases the code should be injected in all classes implementing the interface. A more severe problem is that the user cannot provide a specific implementation of the Particle class.

The AOmpLib solves all these issues by modularising parallelism related code into aspect modules. Those modules are attached to the base code using the AOP pointcut mechanism, which also supports pointcuts defined over Java interfaces and bindings that are retained over the class hierarchy. The next section describes the AOmpLib.

III. Design and Implementation of AOmpLib

This section describes a library of aspect modules that currently supports the approach. The section starts by presenting the execution model and how it is supported by aspect oriented programming techniques [3]. Later it describes the implemented parallelism constructs and how those concerns are composed. The last subsection provides a case study.

A. *Execution Model*

The AOmpLib execution model is inspired in OpenMP and on Concurrent Object-Oriented Languages (COOL) [2]. The main source of parallelism in AOmpLib are parallel regions, as in OpenMP. The execution starts with a single activity, designated master thread, that creates a new team of threads when enters into a parallel region. When the threads in a team encounter a work-sharing construct the work is divided among them. Synchronisation constructs enforce execution constrains among the threads in the team. Data sharing constructs specify how heap allocated objects are shared among threads.

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 AOmpLib each mechanism acts upon a set of method calls in the base program (i.e., a joinpoint in AOP terminology). Thus, like in COOL languages all parallelism related constructs are bound to class interfaces, more specifically to method executions. A parallel region is the context of a method execution. When the master thread performs such method execution a new team of threads is created, where each thread will execute the method and implicitly synchronise when the method execution ends. Additional synchronisation among the threads in the team can occur on method executions performed in the calling context of a parallel region. For instance, methods declared as *critical* behave as *synchronized* Java methods and enforce execution in mutual exclusion.

In AOmpLib multiple statements are grouped by moving those statements into an externally visible 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 (e.g., the mechanism can be applied to a method with a given name and signature). 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 methods call are often automatically made inline by the compiler, avoiding the overhead of an additional procedure call.

A consequence of the previous rule is the refactoring of loops into methods, which are a very important source of parallelism in scientific applications. To inject parallelism in loop executions using independently developed pluggable modules, external modules must be able to act upon the loop iteration range. The AOmpLib follows a strategy similar to TBB by exposing the loop iteration space as method parameters. These methods are called *for methods* and expose the loop iteration range in their first three integer parameters (start value, end value and step). These methods are part of the parallelisation API, since when refactoring loops into methods they become part of the class interface. Based on this approach, those methods can be annotated with properties that enable the support of an execution model similar to *for* work sharing constructs in OpenMP. Exposing *for* loops as methods enables the development of pluggable modules that can exploit different loop scheduling approaches (e.g., static versus dynamic). Furthermore, it is the key to enable efficient load distribution policies by plugging aspect modules that rewrite the iteration range according to the thread id that executes the method.

Data sharing among activities is only possible through heap allocated objects, since local variables and method parameters are intrinsically local in Java. Heap allocated objects are shared by default in parallel regions unless they are declared thread local objects. In such cases additional constructs provide control on how/when values are copied to/from shared memory. The memory consistency model layers on top of the Java weak consistency model. An activity can locally cache shared values until a synchronisation primitive is executed.

B. Aspect-Oriented Programming

The Java language is one of the most widely used programming languages and it includes support for concurrent programming through Java threads and other concurrency abstractions. However, it is cumbersome to develop parallel applications with Java threads as it generates more verbose programs than an OpenMP-like implementation would generate (as it was shown in section 2). Several authors [1][7] proposed extensions to Java to support OpenMP. Some approaches explore Java 5 annotations introducing an alternative to traditional OpenMP directives. AspectJ [8], an aspect-oriented extension to Java, provides an easy way to associate semantic actions to those annotations. Moreover, aspect-oriented techniques also offer the possibility to encapsulate in the form of reusable modules many concurrency patterns and mechanisms [9].

The current AOmpLib (v1.0) is built on top of the AspectJ language and supports both annotation and pointcut style of programming. Annotations provide the easiest way to use the library (although, also more limited). For instance, a parallel region can be expressed by placing the *@Parallel* annotation in the corresponding method definition. The pointcut style involves the creation of an aspect module that extends the abstract aspect *ParallelRegion*. For instance, the aspect in Figure 4 specifies that executions of *someMethod()* are parallel regions (e.g., they will be executed by a team for threads).

```
public aspect MyParallelRegion extends ParallelRegion
{
 pointcut parallelMethod() : call (void someMethod());
}
```

Figure 4. Concrete aspect for a parallel region

The pointcut style is more powerful than annotation style since aspect specific methods can be overridden to customise the abstract aspect for the specific situation. It also enables the development of pluggable modules, since aspects can be deployed at load time and, thus, they keep the base program free from parallelism related statements.

The pointcut style natively supports OO mechanisms, since a pointcut can act upon all implementations of a method (including overriding methods) and also can act upon Java interfaces (e.g., all methods implementing a given interface).

This paper illustrates AOmpLib mechanisms with the annotation style. Annotations avoid writing the above aspect, since the library provides aspects implementations for annotations. For instance, Figure 5 shows the aspect that acts upon all methods that are annotated with *@Parallel*. The annotation style, however, suffers from limitations similar to OpenMP concerning modularity.

```
aspect ParallelAnnotation extends ParallelRegion {
 pointcut parallelMethod() : call (@Parallel * *(*));
}
```

Figure 5: Aspect parallel region capturing annotated methods

The AOmpLib uses AspectJ mechanisms to rewrite the base program. The aspect compiler (called weaver) rewrites the base program to include the code provided in the aspect implementation. In this specific case, annotated methods will be rewritten to include code to create a team of threads and to synchronise that team at the end of the method execution. Current AspectJ implementations perform rewrites at compile-time (or at load time) introducing very low run-time overhead on most cases.

Aspects are specified into separate modules (e.g., an AspectJ module is specified in a way similar to a Java class) and can be configured for each concrete usage. In the annotation style such configuration is performed though additional annotation parameters. In the pointcut style, it is performed by overriding methods of the base (abstract) aspect. For instance, a parallel region with 4 activities can be created with *@Parallel(threads=4)*, or by defining the method *int numThreads() { return(4); }* in the concrete aspect of Figure 4.

C. Programming Abstractions

Table 1 summarises the currently supported set of programming abstractions. Parallel regions are the main source of parallelism (e.g., methods annotated with *@Parallel*). All threads in the team execute the code inside those parallel regions. *For methods* (e.g., methods providing loop iteration space in their first three parameters) can be annotated with the *@For* work-sharing construct. The library provides three different loop-scheduling alternatives: static by blocks, static cyclic and dynamic. The *@Ordered* construct is only supported within the calling context of a *for method*.

@Parallel[(threads=n)]
@For[(schedule=[staticBlock | staticCyclic | dynamic)]
@Task @TaskWait
@FutureTask @FutureResult
@Ordered
@Critical[(id=name)]
@BarrierBefore @BarrierAfter
@Reader @Writer
@Single
@Master
@ThreadLocalField[(id=name)] @Reduce[(id=name)]

Table 1. Supported OpenMP abstractions

The @Task spawns a new parallel activity to execute the annotated method. This construct can also be used outside the parallel region. In both cases an additional method can be defined to act as the join point between the spawning and the spawned activity. @FutureTasks is similar but targets methods with a return value. Those methods must return an object with getter/setter methods that act as synchronisation points (specified by @FutureResult).

@Critical restricts the execution of a method to a single activity at once. The library provides an implementation of this mechanism to replace the Java built-in *synchronized* mechanism. In Java each object holds its own lock, which is used to ensure the exclusive access to methods or regions declared as *synchronized*. The library implementation enables the use of a specific lock that can be shared among multiple type-unrelated objects (as in the OpenMP). In order to identify a particular lock in the application, the parameter *id* should be settled or, otherwise, the lock of the object where the annotation is defined is used (as in plain Java).

The lock id has particular importance to improve composability of annotations. A common strategy to reduce the amount of synchronisation is to use separate locks within the same object. This allows more than one thread to execute methods in the same object but these methods are organised into disjoint sets, each set controlled by a different lock. For this purpose, more than one lock per object is required, which is possible through the use of lock ids. The pointcut style does not have such problem since each aspect instance can use a different lock. To support a single lock across the entire parallel region the library provides two different pointcuts: *criticalUsingCapturedLock* and *criticalUsingSharedLock.* The first uses a lock per target object, the later uses a single lock per aspect.

A @Barrier establishes a synchronisation point where threads synchronise. Each thread, reaching one of those points blocks waiting for the remaining threads in the same team to arrive. The library provides two different annotations, inserting the barrier before or after the method execution. The barrier has the scope of a teams of threads, in a way similar to OpenMP (this contrasts with @Critical whose scope is all threads in the system).

The readers / writer synchronisation mechanism differentiates accesses for reading and writing purposes. It allows multiple readers, but a single exclusive writer. This implementation requires two hook points to specify accesses for reading and writing. The annotation-based implementation uses the @Reader and @Writer annotations.

The @Single and @Master mechanisms conditionally execute a method call, by a single or by the master thread in the team. Both mechanisms can also be applied to methods returning a value. In such case the result is propagated to all threads in the team.

Variables on the stack are local to threads. However, it is sometimes desirable that global variables – e.g. object fields – are instantiated per thread and not per object. This is generally made when objects are exposed to concurrent activities to avoid unnecessary synchronisation, either due to sharing of memory references between threads or required to ensure data consistency. Thread local mechanism enables the declaration of variables local to a thread.

In the library current implementation each thread local object field is initialised with the value of the field outside the thread local context, if the first thread access is a read operation. Otherwise, the thread local value is not initialised, since the first thread access is for writing. The interface of the implementation of thread local uses the *threadLocalFieldRead* and *threadLocalFieldWrite* pointcuts to specify the points where variables are read or updated. In the annotation style a single annotation (@ThreadLocalField) replaces both of these join points

In same cases it is necessary to reduce thread local copies to a single value, in order to compute the object global value (e.g., as if the thread local mechanism was not applied). This is usually performed when the value is requested outside the thread local context (e.g., by the main thread) or when a thread that owns a copy terminates. In the pointcut style the concrete aspect must implement an abstract method that merges two thread local objects into a single object and must also specify the join point where the reduction is performed.

In the case of annotations, thread local objects must implement the *reducer* interface, which provides a method to merge two thread local objects into a single object. An additional annotation (@Reduce) specifies the point where values should be reduced. The optional id parameter can be used to distinguish among several thread local fields.

In certain situations it is necessary to provide parallelism specific code. In this approach specific aspect modules can provide such code. One example is a mechanism that conditionally executes a method call according to some condition (e.g. method parameters). To enable the development of such code the id of each activity in the team can be gathered inside parallel regions by extending an abstract aspect and calling the method *getThreadId()*.

D. Composition of Aspect Modules

The OpenMP standard supports several combined constructs (e.g. a single directive for a *parallel region* and *for* work-sharing). In the AOmpLib 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.

The library also supports nested parallel regions. In such cases, multiple aspects extending the base parallel region aspect can be included in the build (or as a alternative, multiple nested *@Parallel* annotations can be provided).

E. Linpack Case Study

This section illustrates the use of the library by showing details of the parallelisation of the JGF LUFact. This case study is based on the Java version of the highly popular Linpack benchmark [10]. Figure 6 presents a sketch of the code after refactoring (the *dgefa* function), which involved the creation of two new methods that comprise well defined phases of the algorithm: i) the creation of a new method (*interchange*) and ii) a new *for method* (*reduceAllCols*). The more computational demanding part of this algorithm is the row elimination. For each column k of the matrix it performs a column-by-column multiplication (i.e., vector multiplication) of the pivot col_k with every other column, starting at column k+1.

```
int dgefa(double a[][], int lda, int n, int ipvt[]) {
  // gaussian elimination with partial pivoting
  ...
  // for each column
  for(k=0; k<nm1; k++) {
    ...

    // find l = pivot index
    l = idamax(n-k,col_k,k,1) + k;
    ipvt[k] = l;

    if (col_k[l] != 0) {

      // interchange if necessary
      interchange(col_k, k, l);

      // compute multipliers
      t = -1.0/col_k[k];
      dscal(n-(kp1),t,col_k,kp1,1);

      // row elimination with column indexing
      reduceAllCols(kp1,n,1 /* other parameters omitted */);
    }
  }
  ...
}
```

```
// reduce all columns from startc to endc
void reduceAllCols(int startc, int endc, int is /* ... */}) {

  ...
  for (int j = startc; j < endc; j+=is) {

    ... // reduceColumn(a, n, col_k, j, k, kp1, l);
    daxpy(n-(kp1),t,col_k,kp1,1,col_j,kp1,1);
  }
}


void interchange(double[] col_k, int k, int l) {
  ...
}
```

Figure 6. Java Linpack benchmark after refactoring

The parallelisation of this case study (Figure 7) was performed by making the execution of the method *dgefa* a parallel region and applying the *for* work-sharing construct over the *reduceAllCols* method calls. This case also includes the usage of 4 barrier points (1 before and 3 after method calls) and 2 master directives.

```
aspect ParallelLinpack extends ParallelRegion {

  pointcut parallelMethod():
     call(int Linpack.dgefa(..));

  pointcut scheduleForStatic():
     call(void reduceAllCols(..));

  pointcut master():
     call(void Linpack.interchange(..))
   || call(void Linpack.dscal(..));

  pointcut barrierBefore():
     call(void Linpack.interchange(..))

  pointcut barrierAfter():
     call(void reduceAllCols(..))
   || call(void Linpack.interchange(..))
   || call(void Linpack.dscal(..));

}
```

Figure 7. Parallelisation of the Java Linpack benchmark

Figure 8 shows the same example with annotations (for clarity, the body of each method was omitted).

In this case, when using annotations no additional code is required, besides the introduction of annotations in the bse program. The base program can be complied with a standard Java compiler. The AOmpLib is only required to import annotation definitions. In this case, the command line for program execution must specify the use of the aspect weaver as java agent in order to perform load-time weaving of the required AOmpLib aspects.

```
@Parallel
int dgefa( double a[][], int lda, int n, int ipvt[]) {
  ...
}


@For
@BarrierAfter
void reduceAllCols(int startc, int endc, int is /* ... */}) {
  ...
}


@Master
@BarrierBefore
@BarrierAfter
void interchange(double[] col_k, int k, int l) {
  ...
}


@Master
@BarrierAfter
void dscal(int n, double da, double dx[] /* ... */) {
  ...
}
```

Figure 8. Parallelisation of the Java Linpack benchmark with annotations

IV. Implementation Overview

This section describes the implementation of three provided mechanisms: *parallel regions* and *for* work-sharing constructs with static and dynamic scheduling. Those mechanisms were selected due to their implementation simplicity. The implementation of many other mechanisms shares this simplicity but a few implementations are more intricate and their description is out of the scope of this paper since it requires a deeper understanding of AspectJ mechanism.

The parallel region aspect spawns a new set of threads to execute the specified method body (Figure 9). The aspect declares an *abstract pointcut* that is defined in each mechanism usage (see Figure 4 for a concrete usage of this abstract aspect).

```
public abstract aspect ParallelRegion {

   abstract pointcut parallelMethod();

   void around() : parallelMethod() {

       for(int i=0; i< numberOfThreads -1; i++) {

           threads[i] = new Thread() {

               void run() {
                   initialiseMyId();
                   proceed();
               }

           }
           thread[i].start();
       }

       proceed();
       joinAllspawnedThreads(); // wait for other threads
   }

}
```

Figure 9: Implementation of a parallel region aspect module

The aspect intercepts each method call declared as a parallel region (*around* advice statement in the figure) and spawns the number of threads defined for the given region. Each thread initialises its local id (a thread local variable) and starts the execution of the original method (*proceed* statement). After spawning all threads, the master thread executes itself the parallel region, also calling the *proceed* statement. After the method completion the master thread waits for all spawned threads to finish.

The *for* work-sharing construct with static scheduling gathers the *for* method parameters and updates the loop iteration range according the thread id (see Figure 10 for a simplified implementation, note that the actual expressions to compute the lower and upper bounds are slightly more complex). For simplicity the chunk size was defined as one.

```
abstract pointcut forMethod(int,int);

 void around(int i0, int in) : forMethod(int,int) {
   int myId = getThreadId();
   int lowerlimit = myid*(in-i0)/numberOfThreads;
   int upperLimit = (myid+1)*(in-i0)/numberOfThreads;
   proceed(lowerLimit,upperLimit);
 }
}
```

Figure 10: Implementation of a for (static scheduling)

The *around* advice body gathers the first two method parameters (the loop iteration space begin and end), and calls the original method with thread specific parameters.

The dynamic *for* work-sharing works as follows (Figure 11): intercepts the *for* method, calculates the number of iterations within the *for* and assigns an initial task to each thread. While there are tasks to perform, threads will execute them and request for more. 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 synchronised Java routine is used.

```
void around(int i0,int in,int is) : dynamicfor(i0,in,is) {
  int number_of_tasks = (in-i0)/is;
  int iteration = getTask();
  while(iteration < number_of_tasks) {
     proceed(iteration,iteration+chunk,incr);
     iteration = getTask();
  }
  // call barrier
}
```

Figure 11: Implementation of a for (dynamic scheduling)

The current AspectJ weaver and the JIT compiler can inline most of the code specified in separated methods. In certain cases, refactoring loops into methods can even provide a performance improvement, since the compiler can use better heuristics to decide when it is worth to inline a call (there is however some performance penalty when methods can be overridden in derived classes). Figure 12 illustrates the weaving process by showing the code that would be generated from the code of Figure 6 after applying the aspect module of Figure 7 (actually this code is similar to the code from JGF benchmark implemented with Java threads).

```
int dgefa( double a[][], int lda, int n, int ipvt[]) {
    for(int i=0; i<nthreads-1; i++) {
        threads[i] = new Thread() {
            void run() {
                initialiseThreadId();
                original_dgefa(/* parameters omitted */);
            }
        }
    }
    original_dgefa(/* parameters omitted */)
    joinAllspawnedThreads(); // wait for other threads
}

final int original_dgefa(/* ... */) {
  ... // original dgefa code is placed here
}

final void reduceAllCols(int i0, int in, /* ... */) {
    int myId = getThreadId();
    int lowerlimit = myid*(in-i0)/numberOfThreads;
    int upperLimit = (myid+1)*(in-i0)/numberOfThreads;
    original_reduceAllCols(lowerlimit, upperLimit /* ... */);
}

final void original_reduceAllCols(/* ... */)  {
  ... // original code is placed here
}
```

Figure 12: Sketch of the code generated from Java Linpack

The original methods are rewritten into new methods (*orignal_dgefa* and *original_reduceAllCols*), which are called inside new generated methods. These new methods replace the old ones in client calls and include the code provided in the advice, replacing the *proceed* keyword with the method called.

V. EVALUATION

This section presents an evaluation of the proposed collection, by providing high level benchmarks, using the Java Grande Forum (JGF) benchmarks. The benchmarks were performed on two machines, a typical desktop machine and a server machine:

1. Intel i7 (with total of four 3.2 GHz cores, sharing a 8MB L3 cache) with 8 GB of RAM. This machine runs Lion OS and Sun JVM 1.6.0_43 64-bit Server VM.
2. Intel Dual Xeon X5650 (with total of two six core processors at 2.66 GHz, each processor with sharing a 12MB L3 cache) with 12 GB of RAM. This machine runs Cent OS 5.3 and Sun JVM 1.6.0_43 64-bit Server VM.

The benchmarks were performed by running JGF section2 and section3 benchmarks. Results are presented for 8 threads in the first machine and 24 threads on the second machine (note: both machines support hyper-threading, thus, these are the number of threads that provide the best overall speedups across those benchmarks). In all benchmarks the performance difference between the JGF version and the AOmpLib version is less than 1% (Figure 13). Note that both LUFact and SOR benchmarks scale poorly due to the lack of locality of memory accesses. The minor differences in other benchmarks are explained by overheads introduced by aspects.

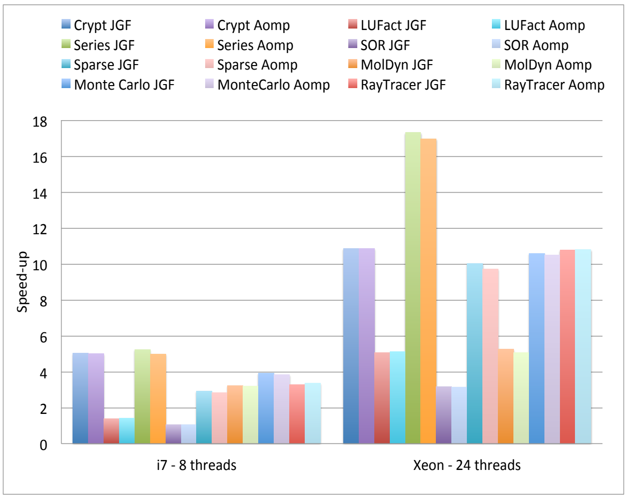


Figure 13. Speed-up with Java threads (JGF) and the proposed approach (Aomp).

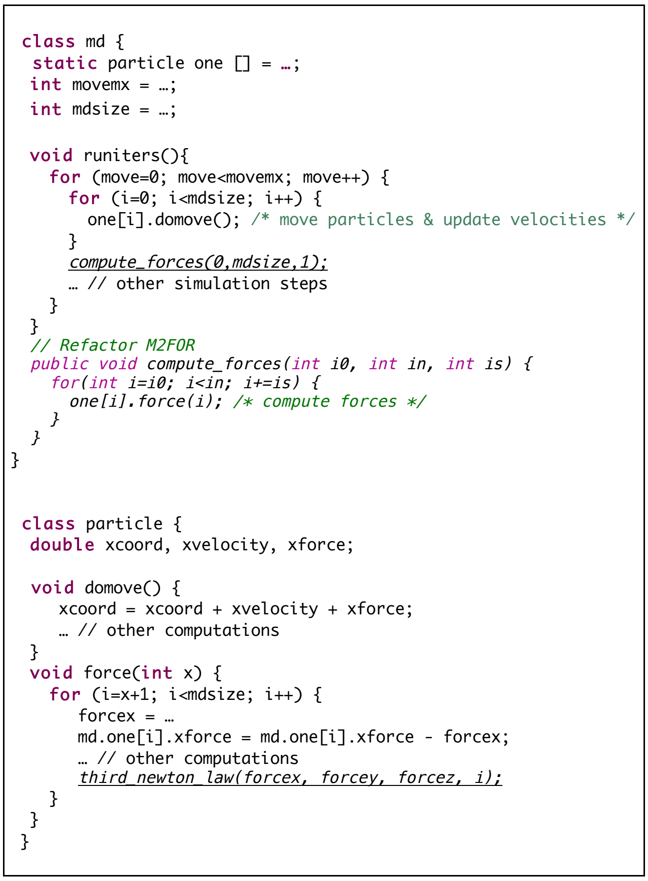
Programmability is very hard to assess, even among sequential programming community. In order to provide an indication of the AOmpLib effectiveness, Table 3 shows the refactoring and abstractions required for each benchmark. Refactoring is classified in two types: i) M2M-Move to method; ii) M2FOR – Move to *for* method. Abstractions used are as follows: PR-Parallel region; FOR-for method; BR- barrier; MA – master; TLF-Thread Local Field, CS – Case specific.

Table 2. Refactoring and abstractions used

	Refactorings	Abstractions
Crypt	M2FOR, M2M	PR, FOR (block)
LUFact	M2FOR, M2M	PR, FOR (block), 4xBR, 2xMA
Series	M2FOR, M2M	PR, FOR (block)
SOR	M2FOR, M2M	PR, FOR (block), BR
Sparse	M2FOR, M2M	PR, FOR (Case Specific), CS
MolDyn	M2FOR, 3xM2M	PR, FOR (cyclic), 2xTLF
MonteCarlo	M2FOR, M2M	PR, FOR (cyclic),
RayTracer	M2FOR	PR, FOR (cyclic), TLF

Note that most benchmarks require a few refactorings (two in most cases) and that the *Sparse* benchmark requires a case specific *for* scheduling strategy and a case specific aspect.

Refactoring can improve the base code in most of the cases, since it frequently encapsulates a set of well-defined steps into a given method. Figure 14 shows two of the required refactorings for the MolDyn case study from Figure 2. Methods *compute_forces* and *third_newton_law* are two of the required refactoring that reflect well-defined steps of the algorithm.



```
class md {
  static particle one [] = …;
  int movemx = …;
  int mdsize = …;

  void runiters(){
    for (move=0; move<movemx; move++) {
      for (i=0; i<mdsize; i++) {
        one[i].domove(); /* move particles & update velocities */
      }
      compute_forces(0,mdsize,1);
      … // other simulation steps
    }
  }
  // Refactor M2FOR
  public void compute_forces(int i0, int in, int is) {
    for(int i=i0; i<in; i+=is) {
      one[i].force(i); /* compute forces */
    }
  }
}

class particle {
  double xcoord, xvelocity, xforce;

  void domove() {
     xcoord = xcoord + xvelocity + xforce;
     … // other computations
  }
  void force(int x) {
    for (i=x+1; i<mdsize; i++) {
       forcex = …
       md.one[i].xforce = md.one[i].xforce - forcex;
       … // other computations
       third_newton_law(forcex, forcey, forcez, i);
    }
  }
}
```

Figure 14. JGF MolDyn sequential implementation after refactoring

One main benefit of the proposed approach is the ability to quickly (and independently) test new parallelisation approaches. The JGF version uses a thread local array of forces. Besides, JGF two other approaches are possible: i) the use of a critical region on force update; ii) the use of a lock per particle.

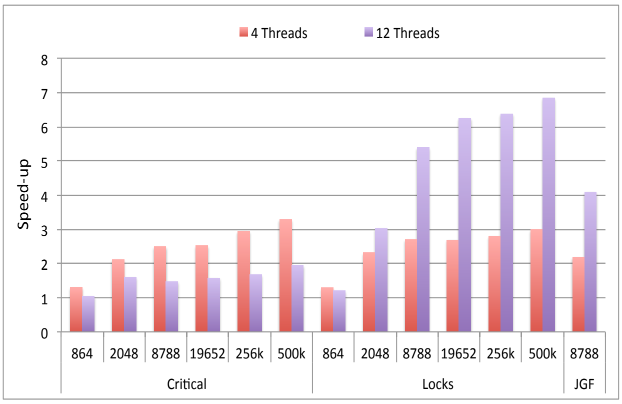


Figure 15. Performance of different JGF MolDyn parallelisations

Figure 15 presents performance results obtained with these two variants, for several numbers of particles (the JGF performs a simulation with 8788 particles). From the figure it is possible to observe that using a lock per particle provides better performance than the JGF base implementation for 12 threads. Moreover, for larger number of particles (256k and 500k) and a small number of threads the critical region approach is the best strategy. This test shows one key point in the proposed approach: multiple parallelisation approaches can be experimented (and simultaneously supported) without modifying the base program.

VI. Related work

Traditional parallel programing languages do not promote modular and independent development. However, in certain domains, such as in linear algebra, common parallelism exploitation patterns have been already encapsulated into libraries (e.g., Basic Linear Algebra Libraries provided by vendors like Intel (MKL) and AMD (ACML)). Thus, the library can include many optimisations from that domain and the final user transparently benefits from those optimised implementations.

Unfortunately, in most domains it is not possible to pack those optimised implementations into libraries. Thus programmers must develop their own implementations. With traditional parallel programming approaches, those parallelisation concerns are mixed up with domain specific concerns, making it impossible to develop modular strategies and reuse those strategies across applications.

OpenMP provides a clean separation between domain specific code and parallelisation concerns (OpenMP pragmas). However, only simple parallelisation concerns are covered by the approach. To develop more sophisticated approaches the programmer must frequently resort to explicit parallel code that relies on threads ids. Proposals for OpenMP for Java, such as JOMP [1], share these problems. AOmpLib overcomes those limitations by supporting an annotation style of programming for usage in simple cases and by supporting the more advanced pointcut based style for more complex parallelism patterns.

Aspect oriented programming has been previously explored to separate parallelisation concerns from the domain specific code [11] and to encapsulate different concerns into separate modules [12]. [13] provides a template-based language aiming to encapsulate different parallelisation concerns into different modules. The work in [9] showed how several concurrency patterns and mechanisms could be encapsulated into reusable aspect modules. AOmpLib differs from these previous approaches by providing a library of aspect modules that mimics the OpenMP standard in Java.

[14] proposed a joint point model for loops, which could avoid having to refactor for loops into methods. However, moving loops into the object API (i.e., by creating *for* methods) is important to promote independent development since the parallelisation modules depend on this explicit API. Intel Parallel Task Library [15] is an example of a system where there is a special construct to express loops. Exposing loops at the object interface level is the key to enable aspect modules that implement loop scheduling strategies.

[16] introduced the concept of asynchronous advice, a technique to delay the execution of the code associated to a pointcut. The idea is similar to delay execution of certain blocks of code, which can also be used to introduce parallelism.

VII. Conclusion

This paper describes the AOmpLib approach, which provides a library of aspect modules that mimics the OpenMP standard in Java. The proposed constructs are closely integrated into object-oriented systems: tasks and synchronisation points are defined at object boundary (e.g., method calls).

The library is fully implemented as reusable aspect modules and supports two different programming styles: annotations and traditional AOP aspect extension through pointcuts.

AOmpLib is a step further in the direction of providing modular and reusable parallelisation concerns. Nevertheless, the base code should be suitable for parallelisation. Programmers should select an algorithm to express domain specific concerns that supports parallelisation. There can also be parallelisation blockers in certain codes that pre-empt the exploitation of parallelism. Thus, designing the base domain-specific code should be a shared task between scientists and computer science specialists to ensure that a proper algorithm is selected and that no parallelisation blockers are introduced in the coding of the algorithm.

The library is being successfully applied to many Java frameworks, enabling the independent development of parallelism modules. One of such cases is the JECoLi

(Java Evolutionary Computation Library) that implements the main metaheuristic optimisation algorithms [17][18]. This case study is an illustrative usage of AOmpLib in a large-scale Java framework.

Current work includes the investigation of the feasibility of this approach in more irregular algorithms (e.g., graph based) and the optimisation of several mechanisms in the collection (e.g., by using concurrency features introduced in Java 7).

Acknowledgment

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 projects FCOMP-01-0124-FEDER-011413 and FCOMP-01-0124-FEDER-010152.

References

[1] J. Bull and M. Kambites, "JOMP—an OpenMP-like interface for Java", Proceedings of the ACM 2000 conference on Java Grande, California, June, 2000. doi: 10.1145/337449.337466.

[2] A. Yonezawa and M. Tokoro, (ed), "Object-Oriented Concurrent Programming", MIT Press, 1987.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin, "Aspect-Oriented Programming" ECOOP'97, Jyväskylä, Finland, June, 1997. doi: 10.1007/BFb0053381

[4] J. Smith, J. Bull and J. Obdrzálek, "A Parallel Java Grande Benchmark Suite", Supercomputing Conference (SC 2001), Denver, Nov. 2001. doi: 10.1145/582034.582042.

[5] B. Hess, C. Kutzner, D. Spoel and E. Lindahl, "Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation", J. Chem. Theory Comput. 4, 3 (March 2008) pp 435-447. doi: 10.1021/ct700301q.

[6] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan and K. Schulten, "Namd2: Greater scalability for parallel molecular dynamics", Journal of Computational Physics", 151, 1, May 1999, pp 283-312. doi: 10.1006/jcph.1999.6201.

[7] M. Klemm, R. Veldema, M. Bezold and M. Philippsen, "A Proposal for OpenMP for Java", Second International Workshop on OpenMP (IWOMP 2006), Reims, France, June, 2006. doi: 10.1007/978-3-540-68555-5_33.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold, "An Overview of AspectJ", ECOOP 2001. LNCS. Budapest, Hungary, June, 2001. doi: 10.1007/3-540-45337-7_18.

[9] C. Cunha, J. Sobral and M. Monteiro, "Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms", AOSD'06, Bonn, Germany, March, 2006. doi: 10.1145/1119655.1119674.

[10] www.netlib.org/benchmark/linpackjava

[11] B. Harbulot and J. Gurd, "Using AspectJ to Separate Concerns in Parallel Scientific Java Code", AOSD 2004, Lancaster, UK, March 2004. doi: 10.1145/976270.976286.

[12] J. Sobral, "Incrementally developing parallel applications with AspectJ", 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 06), Greece, Rhodes, April 2006. doi: 10.1109/IPDPS.2006.1639352.

[13] R. Gonçalves and J. Sobral, "Pluggable Parallelization", 18th ACM international symposium on High Performance Distributed computing, (HPDC 09), Munique, June 2009. 11-20. doi: 10.1145/1551609.1551614.

[14] B Harbulot, J. Gurd "A join point for loops in AspectJ" In Proceedings of the 5th international conference on Aspect-oriented software development (AOSD '06). ACM, New York, NY, USA, 63-74. doi: 10.1145/1119655.1119666

[15] D. Leijen, W. Schulte and S. Burckhardt, "The design of a task parallel library" In Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA '09). ACM, New York, NY, USA, 227-242. doi: 10.1145/1640089.1640106.

[16] D. Ansaloni, W. Binder, A. Villazón and P. Moret, "Parallel dynamic analysis on multicores with aspect-oriented programming". In Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD '10). ACM, New York, NY, USA, 1-12. doi: 10.1145/1739230.1739232

[17] http://darwin.di.uminho.pt/jecoli

[18] J. Pinho, J. Sobral and M. Rocha, "Parallel evolutionary computation in bioinformatics applications", May 2013. doi: 10.1016/j.cmpb.2012.10.001.