Aspect-Oriented Support for Modular Parallel Computing

João L. Sobral
Departamento de Informática
Universidade do Minho
Campus de Gualtar
4710-057 Braga PORTUGAL
jls@di.uminho.pt

Miguel P. Monteiro
Escola Superior de Tecnologia
Instit. Politécnico de Castelo Branco
Avenida do Empresário
6000-767 Castelo Branco PORTUGAL
mmonteiro@di.uminho.pt

Carlos A. Cunha
Escola Superior de Tecnologia
Instit. Politécnico de Viseu
Campus de Repeses
3504-510 Viseu PORTUGAL
cacunha@di.estv.ipv.pt

ABSTRACT
In this paper, we discuss the benefits of using aspect-oriented programming to develop parallel applications. We use aspects to separate parallelisation concerns into three categories: partition, concurrency and distribution. The achieved modularisation enables us to assemble a variety of platform specific parallel applications, by composing combinations of (reusable) aspect modules into domain-specific core functionality. The approach makes it feasible to develop parallel applications of a higher complexity than that achieved with traditional concurrent object oriented languages.

Keywords
Parallel computing, skeletons, aspect composition.

1. INTRODUCTION
There is a growing demand for parallel applications. The increasingly popular multi-core CPU architectures require concurrent programming to effectively leverage the underlying parallel processing capabilities. However, concurrent programming may introduce overhead in application execution time when running on a single core system. Grid systems [6] connect worldwide computing resources, delivering significant computing power. However, the systems are extremely complex to program, due to the intrinsic heterogeneity of computing resources, network latencies and bandwidths. Grid systems are built as clusters of clusters of multiprocessors machines, whose processors can be multi-core CPUs.

Traditional parallel applications suffer from classic tangling problems [10], as parallelisation concerns cut across multiple application modules. Core functionality (i.e., domain specific logic) is usually mixed with parallelisation concerns. Such concerns include work partition into parallel tasks, concurrent execution of these tasks; synchronisation of parallel accesses to shared data structures (to avoid data races) and distributed execution of the tasks. Code related to these concerns is tangled in application code, which makes it harder to understand, reuse and evolve core functionality and parallel code. Traditional parallel programming is mainly focused on performance issues. Aspect-oriented programming (AOP) [10] contributes to conciliate between high level and high performance computing, by modularising the above concerns more effectively [9].

This paper presents an approach to develop parallel applications in which parallelisation concerns are modularised into various aspects. Section 2 overviews the approach. Section 3 focuses on the development of reusable parallelisation concerns and section 4 discusses benefits and how to support composition of the aspects. Section 5 concludes the paper.

2. MODULAR PARALLELISATION CONCERNS
Our approach entails modularising parallelisation concerns with AOP, to increase general modularity and reuse potential in parallel applications. The base application must be amenable for parallelisation, since our focus is more on modularising *existing* parallel applications than parallelising sequential applications. It is harder to transform sequential applications into parallel ones than to modularise current parallel applications. The greater suitability of AOP to achieve such transformations of sequential applications relative to other approaches is as yet unproven.

Our approach involves using traditional object oriented mechanisms to implement application core functionality and implementing parallelisation concerns with AOP. Parallelisation concerns are grouped in thee categories: functional or/and data partition, concurrency and distribution. Each concern is implemented in its own aspect module that can be (un)plugged from application core functionality.

Core functionality expresses domain specific logic; by specifying what the application is supposed to do. Partition modules specify how work is performed in an efficient way, using several processing elements. Concurrency modules manage execution of parallel tasks, including synchronisation requirements. Distribution modules assign objects to available resources and manage remote method invocations.

The programmer is in charge of dividing parallel code into these concerns. In some parallel applications, core functionality code is the same as the sequential code. However, in intrinsically parallel applications (i.e., in parallel applications where there is no sequential equivalent) our core functionality module contains the domain-specific logic.

The partition module transparently replicates objects and manages method calls to the replicated objects. Replicated objects are managed by the partition aspect, which also manages their life-cycle (these are called aspect managed objects). The partition aspect controls the way a method call is executed into such objects. Usual partitions include *farming*, *divide and conquer*, *pipeline* and *heartbeat* [2]. For instance, in a pipeline partition, aspect-managed objects are organised in a pipeline sequence and each method call is successively executed by all pipeline objects. Partition module can introduce joinpoints that can be intercepted by other aspects.

The concurrency module specifies asynchronous method invocations: the caller object is allowed to proceed while the called object executes the requested method. In Java, this can be achieved by using a new thread to perform the requested method. Asynchronous method invocations may also require synchronisation to protect shared objects, to avoid data races and to ensure a specific execution order. Synchronisation code is also placed in the concurrency aspect and it resorts to synchronised block constructs and monitors provided by Java.

Partition and concurrency code is deployed in separate modules. The main idea is first to develop a partition module and next to develop the concurrency module. This way it is possible to (un)plug concurrency for debugging purposes and it also helps to avoid the inheritance anomaly problem [12], as partition code can be reused independently of concurrency constraints.

Our approach is based on distributed objects, which can be deployed across machines. Object distribution concerns are implemented in their own module as well. We identify two main benefits: (1) partition and concurrency modules can be developed without taking into account object distribution issues (i.e., they are developed for a single processor/shared memory machine) and (2) it is easier to switch between underlying middleware implementations for distribution concerns, such as CORBA, Java RMI and MPI.

A simple example (using Java and AspectJ) of the intended separation of the above concerns is described next. The Java Grande Forum (JGF) RayTracer [14] is a benchmarking application that renders an image of sixty spheres. This benchmark is already amenable for parallelisation. Actually, JGF provides both sequential and parallel versions of this application. Core functionality is very close to the JGF sequential version. It creates a *RayTracer* object, initialises it with a scene to render and then sends it a message to render the scene, specifying image size by means of an *Interval* object:

```
RayTracer rt = new RayTracer();
Scene sc = ... // create scene to render
rt.initialise(sc);
Interval interval = new Interval(0,500);
Image result = rt.render(interval);
```

The concrete partition aspect for this example intercepts the creation of the *RayTracer* object and creates a set of aspect-managed *RayTracer* objects. The concrete partition aspect broadcasts the call to method *initialise* to all the new *RayTracer* objects. It also broadcasts the call of method *render,* using a different interval for each *RayTracer* object and joins partial results produced by each one (this is a typical implementation of a simple farm partition):

```
RayTracer farm[]=new RayTracer[numberOfWorkers];

RayTracer around() : call (RayTracer.new()) {
   for(i=0; i<numberOfWorkers; i++)
      farm[i] = new RayTracer();
   return(farm[0]);
}

void around(/*scene*/) :
               call (RayTracer.initialise(..)) {
   for(int i=0; i<numberOfWorkers; i++)
      farm[i].initialise(/*scene*/);
}
```

```
Image around(/* interval */) :
                    call (RayTracer.render(..)) {
   for(int i=0; i<numberOfWorkers; i++)
      res[i] = farm[i].render(/* subinterval*/);
   ... //join sub-images saved in res array
   return(/*merged subimages*/);
}
```

In the above example, array *farm* stores the references to aspect-managed objects. Element zero of the array is used as the group front-end: it executes calls to *RayTracer* objects that are not intercepted by the partition aspect. We chose to perform explicit calls (i.e., *new*, *initialise* and *render* calls), instead of calling *proceed* to allow specific advising of joinpoints introduced by this partition aspect. Also note that all advices also must include *!within(...)* to avoid recursive advices (the *within* pointcut designator (PCD) is omitted above for simplicity). This aspect can be deployed with *pertarget* instantiation to support advices on multiple *RayTracer* instances.

The concurrency aspect spawns a new thread for each call to *initialise* (code for an asynchronous call of method *render* is also possible but trickier, as it involves the creation of a future object):

```
void around() : call (RayTracer.initialise()) {
   (new Thread() {
      public void run() {
         proceed();
      }
   }).start();
}
```

Distribution aspects redirect local object creations/calls to remote object instances. The caller local aspect plays the role of traditional proxies. However, a fake local object is required due to type system compliance. The following code presents a sketch of the code for the ray tracer benchmark:

```
RayTracer around() : call (RayTracer.new()) {
   // request object creation to remote factory
   // associate remote object to local fake
   return(/*fake local object*/);
}

void around() : call (RayTracer.initialise()) {
   // redirect call to remote object
}

Image around() : call (RayTracer.render(...)) {
   // redirect call to remote object
   return(/*remotely rendered image*/);
}
```

Table 1 presents combinations of these modules and their purpose. By modularising parallelisation concerns into multiple aspects it is possible to manage multiple configurations of a parallel application and to deploy the one that more adequately matches the target platform. When the target platform is a single processor machine, only core functionality is deployed. On multiprocessor machines, the concurrency module is included as well. However, we keep the choice over whether we include the partition module, depending on the type of parallel application (e.g., in branch and bound applications partition module usually is not required).

Table 1. Deployable parallel applications

Partition Module	Concurrency Module	Distribution module	Purpose
No	No	No	Tidy up core functionality, debugging, single processor machines
Yes	No	No	Tidy up partition strategy, debugging
No / Yes	Yes	No	Shared memory parallel machines (SMP/Multi-core)
Yes	Yes	Yes	Distributed memory machines/Grids
No	No / Yes	Yes	Distributed application

Aspect precedence is of particular importance in this approach. Partition code has the highest precedence. Contrary to the effect achieved by AspectJ's precedence mechanism, advice of partition code is *always* the first to execute, including after advice. This is required since partition code may introduce new objects and method calls that can be intercepted by other aspects. In the previous ray tracer benchmark, both concurrency and distribution aspects should be applied to aspect-managed ray tracer objects.

Order of precedence of concurrency and distribution determines the difference between client activated and server activated threads [11]. Both precedences achieve asynchronous remote method invocations. When concurrency has higher priority, threads are created to perform all remote communication in a separate local thread running on a client node, to hide network latencies and to increase network bandwidth usage.

The precedence rules ensure that parallelisation concerns are composed in the right order. However, in this composition model, each aspect only applies to joinpoints introduced by one previous module (i.e., with the next higher precedence, in the precedence list). For instance, it does not make sense to apply the distribution module to jointpoints from the core functionality and from the partition aspect. Replacing the explicit method calls in aspects by calls to *proceed* can enforce this behaviour; however, our experience reveals that this becomes trickier, since it depends on the particular implementation of the aspect weaver. Another alternative is to use the *within* PCD to specifically advise a single module, but it negatively affects the flexibility to assemble parallel applications, as it reduces the range of possible compositions.

A description of the approach, to modularise parallelisation concerns, from a parallel computing perspective, as well as a detailed case study, including performance figures, can be found in [17].

3. REUSABLE MODULES

Having modularised partition, concurrency and distribution concerns, the next step is to build reusable modules for these concerns. The modules are abstract aspects that are developed on the basis of abstract pointcuts and marker interfaces.

Reusable implementations based on abstract pointcuts follow the template advice idiom [7] that is extensively used in [8]. An abstract aspect defines the reusable crosscutting implementation. Reusable code is applied to a case-specific situation by creating a concrete aspect that inherits the logic from the abstract aspect, usually a set of abstract pointcuts and methods. The pointcuts are defined by the concrete aspects to specify the case-specific joinpoints. Inherited methods configure the logic defined in the abstract aspect by defining case-specific logic that binds the reusable part to the case-specific part.

Marker interfaces are used to implement mixin composition [3]. Concrete aspects introduce implementation of marker interfaces declared in the abstract aspect to case-specific classes. Joinpoints originating from implementing classes are captured by the aspects to apply the concern logic as with other. Marker interfaces are preferable when the aspect implements a class based role.

The reusable partition module combines the two above approaches. Each reusable aspect declares marker interfaces to specify the classes whose instances are replicated (i.e., which classes give rise to aspect-managed instances). Abstract pointcuts are concretised to specify how method calls are executed by the aspect-managed instances.

The reusable partition module implements the functionality to transparently replicate objects and redirect/broadcast method calls to aspect-managed objects. Method calls that are not captured by the aspect run on a special object, the group proxy. The structure of the reusable aspect (not shown here) is similar to the partition code as presented in previous section. The main difference is that in this case we replace references to class *RayTracer* by interfaces, abstract pointcuts and calls to *proceed*. Figure 1 presents an example of the reuse of the partition aspect. Objects of class *RayTracer* are replicated into all available processing elements. Calls to method *initialise* are broadcasted to all objects in this set (pointcut *broadcastCall*). Calls to method *render* are also executed by all elements in the set but each object receives a different argument (computed by method *scatter*, which is called by aspect *ObjectGridProtocol*). For simplicity, code implementing the merging of results of method *render* is not shown.

```
aspect Partition extends ObjectGridProtocol {

   declare parents: RayTracer implements Grid1D;

   pointcut void broadcastCall() :
               call(* RayTracer.initialise()));

   // calculates parameters of each scatterCall
   Vector scatter(Object arg) {
      Vector v = new Vector();
      ... // splits arg into sub-intervals
      return(v);
   }
   pointcut scatterCall(..) :
            call (* RayTracer.render(..)) ...;
}
```

Figure 1 – Example of reusing of partition concerns

A complete description of the reusable aspects for partition can be found in [16].

Reusable components for concurrency concerns provide functionality to perform calls in separate threads and to manage synchronisation among running threads [5]. Figure 2 presents an example in which separate threads call method *initialise*.

```
public aspect Oneway extends OnewayProtocol {
   protected pointcut onewayMethodExecution() :
      (execution(* RayTracer.initialise(..)));
```

Figure 2 Reuse of OnewayProtocol

Distribution concerns can be modularised using aspects [15][17]. However, building *reusable* abstract aspects for distribution seems to be harder. Traditional approaches require additional tools to automate the generation of distribution code. Aspects can improve on such tools by (1) reducing the amount of generated code, (2) avoiding invasive changes on the original classes [4] and (3) by using code templates [19]. Development of reusable modules in the form of abstract aspects may require an extensive usage of introspection features of Java and AspectJ, due to the specificity of Java RMI.

4. COMPOSING (REUSABLE) MODULES

Aspect composition provides a particularly interesting subject to parallel computing. Composition of partition strategies has been well studied on skeleton-based approaches [13]. The idea is to combine partition strategies to achieve more sophisticated parallelisations. One common strategy is to have a two-level farm parallelisation, where each worker is also a farmer (with its own workers). Such a partition strategy is of particular importance to large-scale parallel applications, in which a single level farm risks becoming a bottleneck. With aspect-oriented (AO) modules this composition can be implemented by specifying aspects (e.g., a partition) that act on joinpoints introduced by another aspect. In our example in Figure 1, this would mean that each *RayTracer* object in the set would also be a set of objects. Composing partition aspects can be done by capturing just the joinpoints originating from a specific aspect (e.g., the first-level partition aspect), which can be implemented using the AspectJ *within* PCD. A simple implementation based on non-reusable aspects requires the duplication of partition code presented in section 2:

```
aspect partitionLevel1 {
   RayTracer around(): call (RayTracer.new()) {
      // same as before
   }
   void around(): call (RayTracer.initialise()){
      // same as before
   }
   Image around(): call (RayTracer.render(...)){
      // same as before
   }
}

aspect partitionLevel2 {
   RayTracer around(): call (RayTracer.new())
                  && within(partitionLevel1) {
      // same as before
   }
   void around(): call RayTracer.initialise())
                  && within(partitionLevel1) {
      // same as before
   }
   Image around(): call (RayTracer.render(...)){
                  && within(partitionLevel1) {
      // same as before
   }
```

Composing concurrency and distribution aspects is also of interest. When using a two-level farming it may be more efficient to only distribute objects of the first level of the farm. This type of composition closely matches the architecture of clusters of SMP machines. To achieve such composition, the distribution aspect should be applied only to the first-level partition aspect. This can be also achieved including *within(partitionLevel1)* in all pointcuts.

Composing reusable aspects is harder. For instance, when using marker interfaces it is not possible to distinguish from aspect-managed instances of level1 and level2 farm. This requires the programmatic support of associations between objects and aspects. A similar problem occurs when explicit method calls are replaced by *proceed* to develop reusable aspects, no longer being possible to advise a particular aspect by means of the *within* designator.

5. RELATED WORK

Skeletons and templates are alternative ways to achieve the separation between core functionally and parallelisation strategies. In functional languages, the parallelisation strategy can be modelled by higher-order functions that accept functions as parameters [13]. Templates and generative patterns provide generic classes for the parallelisation strategy that can be refined to include the core functionality [1]. CO_2P_3S [18] provides an example of such a system: code that models the parallelisation strategy is generated and the user must provide application-dependent sequential hook methods.

Skeletons and templates are very close to our AO approach in that both have a similar goal: to modularise the parallelisation strategy from core functionally. One main difference between skeletons and reusable AOP modules is how parallelisation strategies and core functionality are composed together to yield a parallel application. In the former approach, core functionality must be decomposed into code fragments to fill the hooks provided by the skeleton/template. In AOP approaches this composition is based on joinpoints, which results in less invasive changes to the core functionality. The advantage of AO parallel programming is due to the richer set of mechanisms available to perform compositions between core functionality and parallelisation strategies. On the other way, templates and skeletons have the advantage to enforce stricter rules for compositions and the correct (syntactic) composition can be checked at compile-time (for example ensuring that parallelisation modules are stacked in the right order).

6. CONCLUSION

This paper discusses an AO approach to modularise parallelisation concerns, namely object partitioning, concurrency management and distribution. The approach leverages the superior compositional capabilities of AOP to obtain a higher reuse potential from parallelisation concerns.

Composing non-reusable aspects can be performed using current AOP capabilities. However, an adequate model to compose reusable aspects in a general way is left for future work.

7. ACKNOWLEDGMENTS

This work is supported by PPC-VM (Portable Parallel Computing based on Virtual Machines) project POSI/CHS/47158/2002, funded by Portuguese FCT (POSI) and by European funds (FEDER). Miguel P. Monteiro is partially supported by FCT under project SOFTAS (POSI/EIA/60189/2004).

8. REFERENCES

[1] Aldinucci, M., Danelutto, M., Teti, P., *An advanced environment supporting structured parallel programming in Java*, Future Generation Computing Systems, vol. 19, 2003.

[2] Andrews, G., *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison Wesley, 2000.

[3] Bracha G., Cook W., *Mixin-Based Inheritance*. ECOOP/ OOPSLA 1990, Ottawa, Canada, October 1990.

[4] Ceccato, M., P. Tonella, P., *Adding Distribution to Existing Applications by means of Aspect Oriented Programming*, IEEE SCAM'04, September 2004.

[5] Cunha, C., Sobral, J., Monteiro, M., *Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms*, AOSD'06, Bonn, Germany, March 2006.

[6] Foster, I., Kesselman, C., *The GRID2 Blueprint for a New Computing Infrastructure*, Morgan Kauffman, 2004.

[7] Hanenberg, S., Schmidmeier, A., Unland, R., *AspectJ Idioms for Aspect-Oriented Software Construction*, 8th EuroPLoP, Irsee, Germany, June 2003.

[8] Hannemann, J., Kiczales, G., *Design Pattern implementation in Java and in AspectJ*, OOPSLA 2002, Seattle, USA, November 2002.

[9] Harbulot, B., Gurd, J., *Using AspectJ to Separate Concerns in Parallel Scientific Java Code*, AOSD 2004, Lancaster, UK, March 2004.

[10] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J., Irwin J., *Aspect-Oriented Programming*. ECOOP'97, Jyväskylä, Finland, June 1997.

[11] Lea, D., *Concurrent Programming in Java*, Second edition, Addison-Wesley, 1999.

[12] Matsuoka S., Yonezawa A., *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*. In Research Directions in Concurrent Object-Oriented Programming (Agha G., Wegner P., et al., editors), pp. 107-150, MIT press, 1993.

[13] Rabhi, F., Gorlatch, S. (ed): *Patterns and Skeletons for Parallel and Distributed Computing*, Springer, 2003.

[14] Smith, A., Bull, J., Obdrzálek, J., *A Parallel Java Grande Benchmark Suite*, Supercomputing 2001, Denver, USA, November 2001.

[15] Soares, S., Loureiro, L., Borba, P., *Implementing Distribution and Persistence Aspects With AspectJ*, OOPSLA '02, Seatle, USA, November 2002.

[16] Sobral, J., Cunha, C., Monteiro, M., *Aspect-Oriented Pluggable Support for Parallel Computing*, to be presented at VecPar'06, Rio de Janeiro, Brasil, June 2006.

[17] Sobral, J., *Incrementally Developing Parallel Applications with AspectJ*, to be presented at IEEE IPDPS'06, Rhodes, Greece, April 2006.

[18] Tan, K., Szafron, D., Schaeffer, J., Anvik, J. MacDonald, S., *Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment*, PPoPP'03, San Diego, California, USA, June 2003.

[19] Tilevich, E., Urbanski, S., Smaragdakis, Y., Fleury, M., *Aspectizing Server-Side Distribution*, IEEE ASE 2003, Montreal, Canada, October 2003.