Aspect-Oriented Pluggable Support for Parallel Computing*

João Luís Sobral[1], Carlos A. Cunha[2], Miguel P. Monteiro[3]

[1]Departamento de Informática, Universidade do Minho, Braga, Portugal
[2]Escola Superior de Tecnologia, Instituto Politécnico de Viseu, Portugal
[3]Escola Superior de Tecnologia, Instituto Politécnico de Castelo Branco, Portugal

Abstract. In this paper, we present an approach to develop parallel applications based on aspect oriented programming. We propose a collection of aspects to implement group communication mechanisms on parallel applications. In our approach, parallelisation code is developed by composing the collection into the application core functionality. The approach requires fewer changes to sequential applications to parallelise the core functionality than current alternatives and yields more modular code. We present the collection and show how the aspects can be used to develop efficient parallel applications.

1 Introduction

The wide use of multithreaded and multi-core architectures requires adequate tools to re-factor current applications to take advantage of this type of platforms. Unfortunately, parallelising compilers do not yet produce acceptable results, forcing programmers to rewrite the application to take advantage of this kind of systems. Moreover, parallelisation concerns become intertwined with application core functionality, increasing complexity and harming maintainability and evolvability.

Tangling concurrency and parallelisation concerns with core functionality was identified as one of the main problems in parallel applications, increasing development complexity and decreasing code reuse [1, 2]. Similar negative phenomenon of code scattering and tangling was identified in traditional object oriented applications [3]. Aspect-Oriented Programming (AOP) was proposed to deal with crosscutting concerns in object-oriented systems, enabling programmers to focus on a single concern code that would otherwise be scattered, tangled with domain-specific logic.

The use of AOP to implement parallel constructs provides the usual benefits of modularisation, namely improved code readability and better chance to attain reusability and (un)pluggability. It also increases the reuse potential of sequential

* 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).

code in parallel applications since aspects can be plugged into existing components without any source code modification. AOP techniques were successful in modularising distribution code [4, 5, 6], middleware features [7], and, in a lesser extent, in isolating parallel code in loop based parallel applications [2].

This paper presents a collection of aspect oriented abstractions for parallel computing, replacing traditional parallel computing constructs, presents several case studies that show how this collection can be used to develop parallel applications. Section 2 presents related work. Section 3 presents a brief AspectJ overview, an AOP extension to Java, which we use to implement the collection. Section 4 describes the collection and outlines its implementation. Section 5 presents several case studies and section 6 presents a performance evaluation. Section 7 concludes the paper.

2 Related Work

We classify related work in two main areas: concurrent object oriented languages (COOL) and approaches to separate parallel code from core functionality.

COOLs received a lot of attention in the beginning of the nineties. ABCL [8] provides active objects to model concurrent activities. Each active object can be implemented by a process and inter-object communication can be performed by asynchronous or synchronous method invocation. Concurrent Aggregates [9] is a similar approach but supports groups of active objects than can work in a coordinated way and includes mechanisms to identify an object within a group. Recent COOLs are based on extensions to sequential object oriented languages [10, 11, 12]. These extensions introduce new language constructs to specify active objects and/or asynchronous method calls. ProActive [13] is an exception since it relies on an implicit wait by necessity mechanism, however, when a more fine grain control is required, an object body should be provided (to replace the default active object body) Object groups, similar to concurrent aggregates, have been recently introduced [14, 15]. With these approaches, introducing concurrency primitives and/ or object groups entails major modifications to source code, yielding tangled code, in which parallelisation concerns are intertwined with core functionality.

One approach to separate core functionality from parallel code is based on *skeletons* where the parallelism structure is expressed through skeletons [16, 17, 18]. In generative patterns [19] these skeletons are generated and the programmer must fill the provided *hooks* with core functionality.

AspectJ was used in [4, 5, 6] to compose distribution concerns into sequential applications. In [2] an attempt is made to move all parallelism related issues into a single aspect and [20] proposes a more fine grain decomposition. [21] presents an collection of reusable implementations of concurrency concerns.

OpenMP [22] introduces concurrency concerns by means of annotations that can be ignored by the compiled in a sequential execution.

Our approach differs from the aforementioned efforts in that we propose a collection of reusable aspects, implementing object groups, to achieve similar goals. We use concurrency constructs equivalent to traditional COOLs but we deploy all code related to parallelism within (un)pluggable aspects. Our approach differs from

skeleton approaches since it uses a different way to compose core functionality and parallel code. Our approach requires less intrusive modifications to the core functionality to achieve a parallel application, yields code with a higher reuse potential and supports (un)plugability of parallelisation concerns.

3 Overview of AspectJ

AspectJ [23, 24] is an extension to Java that includes mechanisms for Aspect Oriented Programming. It supports two types of crosscutting concerns: static and dynamic.

AOP allows the change of the application static structure in a crosscutting way. This includes member introduction and type-hierarchy modification. Inter-type declaration mechanism of AspectJ enables member introduction – i.e. fields, methods and constructors – type-hierarchy modification, by adding super-types and interfaces to specific classes. **Fig. 1** presents a point class and **Fig. 2** presents an Aspect that changes class *Point*, to implement interface *Serializable*, and to include a additional method, called *migrate*.

```
public class Point {
        private int x=0;
        private int y=0;

        public void moveX(int delta) {  x+=delta; }
        public void moveY(int delta) {  y+=delta; }

        public static void main(String[] args) {
                Point p = new Point();
                p.moveX(10);
                p.moveY(5);
        }
}
```

Fig. 1. Sample point class

```
import java.io.*;

public aspect StaticIntroduction {
        declare parents: Point implements Serializable;
        public void Point.migrate(String node) {  System.out.println("Migrate to node" + node);
}
```

Fig. 2. Example of a static crosscutting aspect

With dynamic crosscutting it is possible to capture various kinds of execution events, dubbed *join points*, namely object creation, method calls or accesses to instance fields. The construct specifying a set of interesting join points is a *pointcut*. Pointcut construct specifies a set of join points and collects context information from those join points. The general form of a named pointcut is:

```
<visibility-modifier> pointcut <name>(ParameterList): <pointcut_expression>;
```

The *pointcut_expression* is built by composing pointcut designators, using the operators *&&*, ||, and *!*. AspectJ pointcut designators (PDs) identify a set of join points, by filtering a subset of all join points in the program. PDs matching are three-fold: based on join point kind, based on scope and based on join point context. More detailed information about pointcut designators can be found in [23].

Dynamic crosscutting also enables composing behaviour before, after or instead of each of the captured join points using the *advice* construct.

Advices have the following syntax:

```
[before | after | <Type> around] (<ParameterList>): <pointcut_expression>
    {… // added behaviour }
```

The *before* advice adds the specified behaviour before the execution point associated to the join points quantified by *pointcut_expression*. *around* advices substitute the execution points original code by the added code - is capable of executing the original join point through the *proceed* construct - and *after* adds the new code to the end of the execution point. *pointcut_expression* involves an pointcut-like expression, which can involve a pointcut name and parameters. Objects and values specific to the context of the captured join points can be obtained through *this*, *target* and *args* constructs. **Fig. 3** shows a typical example of a logging aspect, applied to Point class. In this example, on every call to methods moveX or moveY a message is printed on the screen. In this case the wildcard is used to specify a pattern for the call's signature to intercept.

```
public aspect Logging {
    void around(Point obj, int disp) : call(void Point.move*(int)) && target(obj) && args(disp) {
        System.out.println("Move called: target object = " + obj + " Displacement " + disp);
        proceed(obj,disp);                          // proceed the original call
    }
}
```

Fig. 3. Example of a dynamic crosscutting aspect

Modularisation of crosscutting concerns is an achievement that would lead to code reusability. Though it is a necessary condition, it is not a sufficient condition, as only the non case-specific code is reusable. Essential parts of the aspect's behaviour are the same in different join points, whereby other parts vary from join point to join point. Reuse of crosscutting concerns requires the capture of reusable code to abstract aspects, in order to be reused by concrete aspects. Concrete aspects contain the variabilities tailored to a case-specific code base that defines the case-specific join points to be captured in the logic declared by the abstract aspect. Abstract aspects rely on abstract pointcuts and/or marker interfaces. In both cases, the abstract aspect only refers to abstract pointcut(s) or to the interface(s) and is therefore reusable. Each concrete implementation entails creating a concrete sub-aspect that concretises inherited pointcuts by specifying the set of join points specific to a particular system, and by making case-specific types implement the marker interfaces. In addition, aspects can contain their own state and behaviour.

An aspect usually localises code related to a single crosscutting concern. A process called *weaving* enables execution of aspect code in multiple non-contiguous points in the system. Weaving process composes aspect code with multiple classes at compile

time. As an example, the behaviour specified by the around advice in **Fig. 3** will be composed in all base classes that call moveX or moveY methods.

3 Aspect Oriented Collection for Parallel Computing

Our aspect-oriented collection (Table 1) is based on three programming abstractions: *separable/migrable objects*, *asynchronous method calls* and *object aggregates*. Implementing these abstractions by means of aspects makes it possible to transform application core functionality (e.g., sequential, domain-specific, object oriented code) into a parallel application. However, the base code should be amenable for parallelisation, i.e., the amount of parallelism that can be introduced by our aspect collection is limited by application dependencies. Additionally, the composition of our collection with the core functionality requires a set of suitable join points, otherwise the source code must be refactored to expose these join points.

Abstraction	Scope	Description
Separate	Class	Separate object - can be placed in any node
Migrable	Class	Migrable object - can migrate among nodes
Grid1D, Grid2D	Class	Object aggregate in a 1 or 2d GRID
Broadcast/scatter	Aggregate	Broadcast/scatter method among members
Reduction/gather	Aggregate	Reduce/gather method among members
Redirection	Aggregate	Redirect method call to one member (round-robin)
DRedirection	Aggregate	Redirect call to one member (demand-driven)
Barrier	Aggregate	Barrier among aggregate members
OneWay	Method	Spawns a new thread to execute the method
Future	Method	Spawns a new thread and returns a future
Synchronised	Method	Implements object-based mutual exclusion

Table 1. Aspect oriented collection of abstractions for parallel computing

Separable objects are objects that can be placed remote nodes, selected by the run-time system. Migrable objects are similar, but they can migrate to a different node after their creation. These two abstractions are specified through the separable and migrable interfaces using the declare parents AspectJ construct (see section 3).

Asynchronous method calls introduce parallel processing between a client and a server. The client can proceed while the server executes a requested method. Asynchronous calls can be OneWay and Future. One-way calls are used when no return value is required. **Fig. 4** shows the synopsis for the use of one-way calls.

```
public aspect specificOneWay extends OnewayProtocol {
   protected pointcut onewayMethodExecution(Object servant) : <pointcut definition>;
   protected pointcut join() : <pointcut definition>;
}
```

Fig. 4. One-way introduction

Pointcut *onewayMethodExecution* specifies the join points associated to invocation of methods that run into a new parallel task. Pointcut *join* can optionally be used to

specify join points where the main thread blocks, waiting for the termination of the spawned tasks.

Future calls are used for asynchronous calls that require a return value. In typical situations, a variable stores the result of a given method call, which is used in a later phase. Instead of blocking in the method call, the client blocks when the variable that stores the result is actually accessed. **Fig. 5** shows the synopsis for the implementation of futures.

```
public aspect aspectname extends FutureProtocol {
   protected pointcut futureMethodExecution(Object servant): <pointcut definition>;
   protected pointcut useOfFuture(Object servant): <pointcut definition>;
}
```

Fig. 5. Future introduction

Pointcut *futureMethodExecution* indicates the asynchronous method calls and pointcut *useOfFuture* defines the join points where the result of the call is needed. The client blocks on join points captured by *useOfFuture*, in case the methods defined in *futureCall* have not completed execution.

A richer set of primitives for synchronisation is also available [21], namely Java's synchronised methods, barriers and waiting guards, but their description is out of scope of this paper.

Object aggregates are used to transparently represent a set of object instances in the core functionality. An object aggregate deploys one or several object instances in each node (usually one per physical processor/core) and provides additional constructs to access the members of the aggregate. There are two main interfaces to support aggregates: Grid1D and Grid2D; they differ only on the way the internal members of the aggregate are referenced. For instance, a Grid1D aggregate provides two calls: *getAggregateElems()* and *getAggregatetElemId()*. Grid1D and Grid2D aggregates are specified in a way similar to separate objects (i.e., using *declare parents*).

Calls to the original object instance (i.e., calls in the core functionality) are replaced by calls to the first object in the aggregate (called the aggregate representative), but these calls can also be broadcasted, scattered and reduced among member of the aggregate. Broadcasted calls are executed in parallel by all aggregate members, using the same parameters from as core functionality call. Such call returns when all broadcasted calls complete. **Fig. 6** shows the synopsis for the use of broadcasted calls. Pointcut *broadcast* specifies method calls broadcasted to all aggregate members.

```
protected pointcut broadcastMethodExecution(Object servant) : <pointcut definition>;
```

Fig. 6. Broadcasted calls introduction

Scattered calls (**Fig. 7**) are similar to broadcasted calls but they provide a mechanism to specify a different argument for each aggregate member. This is specified by implementing the abstract method *scatter* which returns a vector of several values, one for each call on an aggregate member.

```
protected Vector scatter(Object callParameter) {
    …
}
protected pointcut scatterMethodExecution(Object serv, Object arg) : <pointcut definition>;
```

Fig. 7. Scattered calls introduction

Reduced calls are also similar to broadcasted calls, but they provide a mechanism to combine return values of each aggregate member call. This type of calls is intended to be used instead of broadcasted call, when the call returns a value. In this case a reduction function specifies how to combine the return values of each aggregate member call (**Fig. 8**).

```
protected Object reduce(Vector returnValues) {
    …
}
protected pointcut reduceMethodExecution(Object serv, Object arg) : <pointcut definition>;
```

Fig. 8. Reduced calls introduction

An additional function (scatter/reduce) performs a combination of scatter and reduce calls. Other aggregate functions can redirect a call to one aggregate member in a round-robin fashion (*redirectCall*) or in a demand driven scheme (*dredirectCall*).

Broadcasted, scattered and reduced calls are valid just for object aggregates (e.g., method calls on objects that implement interfaces Grid1D or Grid2D).

Fig. 9 shows a simple application that illustrates the use of this collection of aspects. The object *Filter* in the core functionality (at the left) is transformed into an aggregate and calls to *filter* are broadcasted, in parallel, to all aggregate members. Before filter method execution (*before() execution(* Filter.filter)* statement), each aggregate member displays its Id.

Core functionality	**Parallelisation code**

public class Filter {
  void filter() {
  …
  }
…
Filter f = new Filter();
 	
f.filter();
	declare parents: Filter implements Grid1D;
	
	before() : execution(* Filter.filter(..)) && … {
	  System.out.println("Called on " + getAggregateElemId());
	}
	
	pointcut broadcastMethodExecution(..) : call(* Filter.filter(..));

Fig. 9. Simple application example

4 Case studies

This section presents two representative case studies that illustrate the use of the aspect collection to develop modular parallel applications. The case studies are from the parallel Java Grande Forum Benchmark (JGF) [25]. This benchmark includes several sequential scientific codes and parallel versions of the same applications, using mpiJava (a bind of MPI to Java). Their parallel implementations introduce modifications to the sequential code, intermingling domain specific code with MPI primitives to realise parallel execution. These tangled implementations make it difficult to understand both the parallelisation strategy and the domain specific code. Our approach entails introducing as fewer modifications as possible to domain scientific code, introducing parallelisation logic through non-invasive composition of aspects from the collection. This approach makes the implementation of the parallelisation strategy more modular and explicit.

The first case study is a Successive Over-Relation method (SOR), an iterative algorithm to solve Partial Differential Equations (PDEs). This application is parallelised using a heartbeat scheme, where each parallel task processes part of the original matrix, exchanging information with neighbour elements after each iteration.

The second application is a ray-tracer that renders a scene with 64 spheres. It is parallelised using a farming strategy, where each worker renders a set of image lines.

4.1 Successive Over-Relation

The SOR method is used to iteratively solve a system of PDE equations. The method successively calculates each new matrix element using its neighbour points. A sequential Java program of the JGF method is outlined in **Fig. 10**. This code iterates a number of pre-defined iterations, given by *num_iterations*, over matrix *G*.

In this particular case, the sequential version is not the best version to execute in parallel due the dependencies among calculations. To overcome this limitation the SORrun implementation was changed to use the Red-Black parallel version, becoming more amenable for parallel execution. This strategy was also followed in the JGF parallel benchmark to derive the parallel version of the application.

```
public class SOR {
    …
    public static final void SORrun(double omega, double G[][], int num_iterations) {
        …
        for (int p=0; p<num_iterations; p++) {
                … // performs one iteration
        }
        …
    }
}
```

Fig. 10. JGF SOR sequential code

The sequential code from the JGF does not provide adequate join points to compose with our collection. Our first step is to use the static crosscutting of AspectJ to make this code suitable for composition with parallelisation code (**Fig. 11**). This code introduces two new methods into the SOR class: the *init* method (lines 04-05) initialises the SOR matrix and the *iterate* method (lines 07-08) performs one iteration. In lines 10-17 the original SORrun call is redefined to call these methods. An alternative would be to refactor all the JGF SOR sequential code to use SOR instances, *init* and *iterate* calls.

```
01   double SOR.MyG[][],
02   static int SOR.omega;
03
04   // initialize matrix
05   public void SOR.init(double G[][]) { MyG = G; }
06
07   // performs one iteration
08   public void SOR.iterate() { SORrun(omega, MyG, 1); }
09
10   // redirects SORrun calls to use SOR instances, init call and iterate calls
11   void around(double omega, double G[][], int iterations) call(* SOR.SORrun(..)) && … {
12        SOR.omega = omega;
13        SOR so = new SOR();
14        so.init(G);
15        for(int i=0; i<iterations; i++)
16             so.iterate();
17   }
```

Fig. 11. SOR method core functionality

SOR core functionality can be parallelised through a typical heartbeat strategy. In this strategy, each parallel task iterates over a matrix subset, periodically exchanging boundary information with its neighbours. The parallelisation aspect has four parts: i) creates several SOR objects; ii) assigns a subset of the matrix to each SOR object; iii) performs a call to the *iterate* method on all the objects in the set and iv) exchanges matrix lines among objects after each iteration.

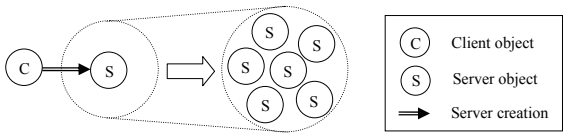

Fig. 12. Transparent creation of several SOR objects

The first step creates an aggregate of SOR objects instead a single object (**Fig. 12**). This is done by specifying that the SOR class implements the Grid1D interface (line 01 in **Fig. 13**). Our system intercepts the creation of SOR instances in the core functionality and creates one SOR object on each node/CPU.

The second step distributes the G matrix among the aggregate, in a block fashion (**Fig. 14**). The code for this step intercepts the *init* method, splits the received matrix into blocks, using the provided scatter method (line 02 in **Fig. 13**) and calls the *init*

method on each object in the set, sending a different block to each element using the scatter method (line 03 in **Fig. 13**). Code for matrix partition (scatter method in line 02 in **Fig. 13**) is a bit tricky to implement since there are lines from the matrix that are replicated in several objects and the first and the last objects receive one line less than other objects. However, this code is also required in a traditional parallel application and, usually, it is tangled with the algorithm core functionality.

```
13  SOR so = new SOR();          01  declare parents: SOR implements Grid1D;

14  so.init(G);                  02  Vector scatter(Object arg) { … }
                                 03  pointcut scatterMethodExecution(..) :
                                                    call (* SOR.init(..)) && ...;
15  for(int i=0; i<iterations; i++)
16      so.iterate();            04  pointcut broadcastMethodExecution(..) :
                                                    call(* SOR.iterate(..)) && …;

                                 05  after() : execution(* SOR.iterate(..)) && … { … }
```

Fig. 13. Parallelisation of the SOR application using our AOP collection

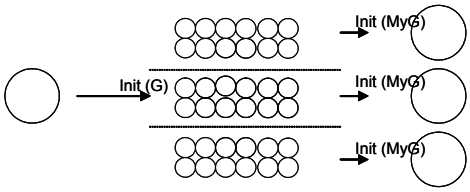

Fig. 14. Matrix distribution among SOR objects

Third, *iterate* method calls are executed by all SOR aggregate objects (**Fig. 15**). Code for this operation implements the broadcast pointcut (line 04 in **Fig. 13**).

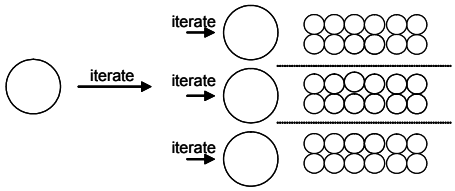

Fig. 15. Iteration distribution among SOR objects

The last step exchanges matrix boundary lines among SOR objects, after an iterate method execution (**Fig. 16** and line 05 in **Fig. 13**).

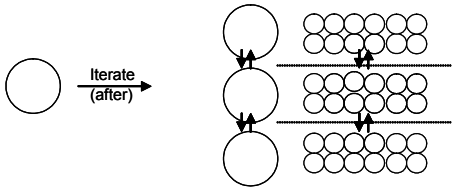

Fig. 16. Boundary exchange among SOR objects

4.2 RayTracer

The JGF RayTrace renders an image of sixty spheres. A simplified version of the JGF sequential code is provided in **Fig. 17**. Method *JGFinitialise* initialises the scene to be rendered and method *JGFapplication* renders the scene. The class *Interval* allows the specification of a subset of the lines to be rendered.

```
public class JGFRayTracerBench extends RayTracer … {
    …
    public void JGFinitialise(){
        …
        scene = createScene();      // create the objects to be rendered
        setScene(scene);            // get lights, objects etc. from scene.
        …
    }

    public void JGFapplication() {
        …
        // Set interval to be rendered to the whole picture
        Interval interval = new Interval(0,width,height,0,height,1);

        render(interval);           // Do the business!
        …
    }
}
```

Fig. 17. JGF RayTracer sequential code

The parallelisation aspect for this benchmark (**Fig. 18**) declares the class *JGFRayTracerBench* to implement the *Grid1D* interface (line 01). Calls to *JGFinitialise* are broadcasted to all aggregate members (line 03) and a call to the *render* method is scattered throughout aggregate elements. The *scatter* function builds a vector with the arguments for each call to one aggregate member. This is the same strategy followed in the JGF parallel version of this application.

```
01  declare parents: RayTracerBench implements Grid1D;
02
03  pointcut broadcastMethodExecution(Object servant) : call(* *. JGFinitialise(..)) && … ;
04
05  Vector scatter(Object arg) {      // calculates the parameters of each call
06      Vector v = new Vector();
07      Interval in = (Interval) arg;
08      …
09      int range = (in.yto-in.yfrom)/workers;  // calculates the render range for each worker
10      for(int i=0; i<workers; i++) {
11          Interval inp = new Interval(0, in.height, in.height, range, range*(i+1), 1);
12          v.add(inp);     // saves the range of each worker
13      }
14      return(v);
15  }
16
17  pointcut scatterMethodExecution(Object serv, Object arg) : call (* *.render(..)) && … ;
18
```

Fig. 18. JGF RayTracer parallelisation aspect

5 Performance Results

This section presents a performance evaluation of the proposed aspect collection. The results presented in this section were measured on an unloaded cluster of 8 dual-Xeon 3.2 GHz machines, with hyper-threading enabled, connected through a 1 Gbit Ethernet. This cluster runs Rocks 4.0.0 and Sun Java JDK 1.5.0_3 in client mode. Presented execution times are median of five executions. Sequential execution times were measured on JGF versions where our parallelisation aspects were unplugged. Speed-up values are relative to these sequential execution times.

Fig. 19 presents the execution time for a SOR (4000x4000 matrix) and a RayTracer (500x500 image) on a single machine. With two aggregate members the ray tracer presents better speed-ups, due to less communication required among tasks. Both of these applications can benefit from hyper-threading (i.e., using more than two aggregate members per node). In this case, higher gains in the SOR can be due to stronger dependencies among matrix elements calculations; leading to higher parallelism when the user performs an explicit parallelisation (e.g., provides more independent tasks, by beans of a higher number of aggregate members).

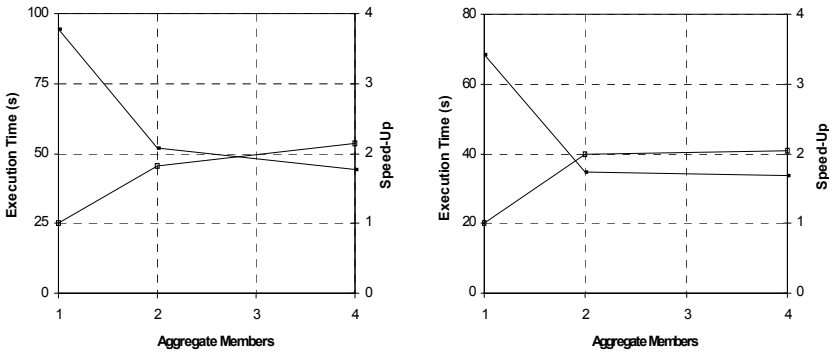

Fig. 19. Execution time and speed-ups for a SOR (at left) a RayTracer (at right).

Fig. 20 presents execution times on 8 cluster nodes. Also in this case the ray tracer presents better speed-ups, due to less communication among tasks. Note that using more than 16 aggregate members leads to a lesser performance improvement, since this additional gain is achieved by using multi-threading capabilities of these processors.

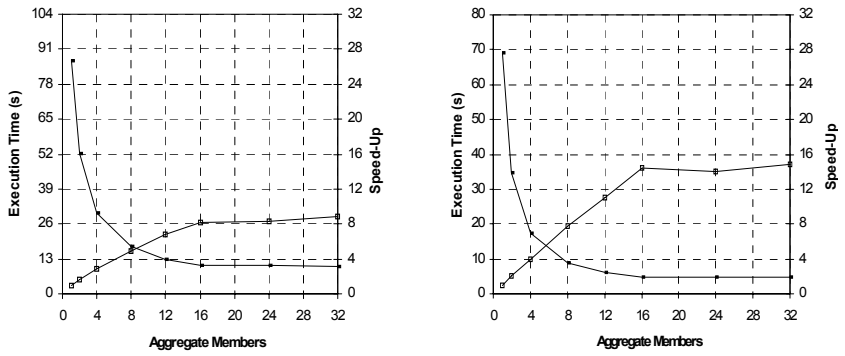

Fig. 20. Execution times and speed-ups for a SOR (at left) and a RayTracer (at right)

Execution times compared to equivalent Java versions (not shown), using MPP (message passing library built on top of Java nio) and Java Threads are within 5% execution time. This overhead is due to the way weaving is performed, which may result in code placed in new classes, instead of being in-lined in the original classes. Scatter and reduce functions can also be an additional source of overhead, since they can require additional data copies.

6 Conclusion

This paper presents a collection of aspects for parallel computing that requires fewer changes to parallelise sequential applications than current alternatives. In addition, it yields parallel object-oriented scientific applications that are more modular and easier to reuse. The collection was successfully applied to several JGF applications.

One of the main drawbacks of this approach is the non object-oriented nature of current scientific applications, as this kind of applications does not provide adequate join point leverage to compose the sequential code with our collection. However, this limitation will have less impact in future as scientific codes become more object oriented. We can partially overcome this limitation by using the static crosscutting mechanisms of AspectJ to introduce the appropriate join points (as in the SOR application).

A second limitation is when the sequential code is note amenable for parallelisation. One solution for this problem is to rewrite the core functionality to allow a more fine grained decomposition. As an example, in the RayTrace we could have a method renderLine which would provide more flexibility to derive the RayTracer parallel version.

Current work includes the extension of this collection to support more orthogonal composition of broadcast, scatter and reduce pointcuts; and a more efficient implementation of these pointcuts on distributed memory machines (e.g., using MPI collective primitives).

References

1. S. Matsuoka, K. Taura, A. Yonezawa: Highly Efficient and Encapsulated Re-use of Synchronisation Code in Concurrent Object-Oriented Languages, OOPSLA '93, Oct. 1993.
2. B. Harbulot, J. Gurd.: Using AspectJ to Separate Concerns in Parallel Scientific Java Code, ACM AOSD'04, Lancaster, UK, March 2004.
3. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin. Aspect Oriented Programming, ECOOP '97, June 1997.
4. S. Soares, E. Loureiro, P. Borba: Implementing Distribution and Persistence Aspects With AspectJ, OOPSLA '02, November 2002.
5. M. Ceccato, P. Tonella. Adding Distribution to Existing Applications by means of Aspect Oriented Programming, 4th IEEE SCAM, September 2004.
6. E. Tilevich, S. Urbanski, Y. Smaragdakis, M. Fleury, Aspectizing Server-Side Distribution, IEEE ASE 2003, Montreal, Canada, October 2003.
7. C. Zhang, H. Jacobsen. Resolving Feature Convolution in Middleware Systems, OOPSLA'04, Vancouver, Canada, October 2004.
8. A. Yonezawa, M. Tokoro, ed, Object-Oriented Concurrent Programming, MIT Press, 1987.
9. A. Chien, V. Karamcheti, J. Plevyak, X. Zhang, Concurrent Aggregates (CA) Language Report - Version 2.0, TR, Dep. Computer Science, University of Illinois, UC, Nov., 1993
10. G. Wilson (Ed). Parallel Programming Using C++, MIT Press, 1996.
11. M. Philippsen. A Survey of Concurrent Object-Oriented Languages, Concurrency: Practice and Experience, 10(12), August 2000.
12. M. Factor, A. Schuster, K. Shagin. A Distributed Runtime for Java: Yesterday and Today, IEEE IPDPS'04, New Mexico, April 2004.
13. F. Baude , L. Baduel, D. Caromel, A. Contes, F. Huet, M. Morel, R. Quilici, Progamming, Composing Deplying for the Grid, in GRID COMPUTING: Software Environments and Tools, Jose C. Cunha and Omer F. Rana (Eds), Springer Verlag, January 2006.
14. J. Maassen, T. Kielmann and H. Bal, GMI: Flexible and Efficient Group Method Invocation for Parallel Programming, Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR-02), Washington DC, March 2002.
15. L. Baduel, F. Baude, D. Caromel, Object-Oriented SPMD, International Symposium on Cluster Computing and the Grid (CCGrid2005), Cardiff, May, 2005.
16. J. Darlington, Y. Guo, H. To, J. Yang. Parallel Skeletons for Structured Composition, PPoPP'95, Santa Clara, USA, 1995.
17. P. Trinder, K. Hammond, H. Loidl, S. Jones. Algorithm + Strategy = Parallelism, Journal of Functional Programming, 8(1), January 1998.
18. F. Rabhi, S. Gorlatch (ed): Patterns and Skeletons for Parallel and Distributed Computing, Springer, 2003.
19. K. Tan, D. Szafron, J. Schaeffer, J. Anvik, S. MacDonald. Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment, PPoPP'03, San Diego, California, USA, June, 2003.
20. J. Sobral, Incrementally Developing Parallel Applications with AspectJ, IEEE IPDPS'06, Rhodes, Greece, April 2006
21. C. Cunha, J. Sobral, M. Monteiro, M., Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms, AOSD'06, Bonn, Germany, March 2006.
22. www.openmp.org
23. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An Overview of AspectJ. ECOOP 2001, Budapest, Hungary, June 2001.
24. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, Getting Started with AspectJ. Communications of the ACM, 44(10), October 2001
25. A. Smith, J. Bull, J. Obdrzálek: A Parallel Java Grande Benchmark Suite, SC'01, Nov. 01.