Incrementally Developing Parallel Applications with AspectJ

J. L. Sobral
Departamento de Informática, Universidade do Minho
4700 - 057 Braga, Portugal
jls@di.uminho.pt

Abstract[1]

This paper presents a methodology to develop more modular parallel applications, based on aspect oriented programming. Traditional object oriented mechanisms implement application core functionality and parallelisation concerns are plugged by aspect oriented mechanisms. Parallelisation concerns are separated into four categories: functional or/and data partition, concurrency, distribution and optimisation. Modularising these categories into separate modules using aspect oriented programming enables (un)pluggability of parallelisation concerns. This approach leads to more incremental application development, easier debugging and increased reuse of core functionality and parallel code, when compared with traditional object oriented approaches. A detailed analysis of a simple parallel application - a prime number sieve - illustrates the methodology and shows how to accomplish these gains.

1. Introduction

When developing parallel applications the programmer is faced with concerns of traditional programming but she/he has also to manage concerns specific to parallel applications, namely she/he must specify how tasks are partitioned among available resources and when these tasks communicate and synchronise. In traditional approaches these concerns are mixed in application components: each module includes code to implement core (sequential) functionality, as well as code to support parallel execution (parallel code). This results in code tangling and scattering, increasing development complexity. It becomes difficult to reuse and/or share existing sequential components and to debug or improve core functionality, since each module includes parallelism related concerns and once these concerns are included in source code they are not easily removed. Also, parallel code is spread into several components (i.e., it cuts across multiple modules), making harder to understand overall parallelism structure and to develop parallel code that can be reused in several parallel applications. Parallel applications are also populated with optimisation code to improve application performance for some classes of target platforms.

Development and maintenance costs of parallel applications can be reduced if parallelisation concerns can be added in a modular way, with the fewest possible changes to the source code. Several key benefits might arise from this approach. First, a large amount of sequential code would become easier to reuse in parallel applications. Second, parallel code would be easier to maintain and reuse, since it would be concentrated in separate modules, making it easier to understand the parallelism structure and to reuse parallel code in several applications. Third, it would be feasible to have both parallel and sequential versions of the same code, since going from one version to the other is just a matter of plugging or unplugging parallelisation modules.

The same strategy can also be realised for optimisation code; however, this type of code is generally more oriented to specific range of target platforms. A modular specification of optimisation code, without spreading the code into several components would make it possible to have several modular optimisations and plug or unplug each optimisation in a particular target platform.

AspectJ [7], a Java extension to support Aspect Oriented Programming (AOP), implements the concept of separation of concerns [8][22]. AOP programming makes it possible to localise within a single module code that would be scattered across multiple components with other programming paradigms. Increased modularity was observed in the implementation of several design patterns [13] using AspectJ.

[1] Work supported by PPC-VM project POSI/CHS/47158/2002, funded by Portuguese FCT (POSI) and by European funds (FEDER).

1-4244-0054-6/06/$20.00

In parallel applications and particularly in object oriented parallel applications, parallelisation concerns cut across several object classes. AOP can help to modularise such concerns. The main idea is to model the core functionality with traditional object oriented mechanisms and to model parallelisation concerns as additional aspect oriented modules, that can be (un)plugged or exchanged from core functionality. This model allows the programmer to concentrate on a specific concern, simplifying the development, since each concern is treated in a modular (and incremental) way. The code also becomes easier to understand since each concern is localised and isolated in a single block of code.

The rest of this paper is organised as follows. Section 2 discusses related research on object oriented parallel programming and AOP. Section 3 presents a brief AspectJ overview. Section 4 presents our design methodology and how it is realised using AOP. Section 5 presents a case study, a prime number sieve. Section 6 shows performance results and section 7 concludes the paper with suggestions for future work.

2. Related work

In the beginning of the nineties integration of objects and concurrency was a very active field in the research community. One of the most relevant works was ABCL [1], which provided 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. There are two alternatives for an asynchronous method invocation: when no return value is required the client object can proceed while the remote object executes the requested method; when a return value is required the client provides a variable, called future, to store the return value. If the client attempts to use this variable before its value becomes available it will be automatically blocked, until the value is computed.

Latter attempts the integrate objects and concurrency are based on extensions to sequential object oriented languages [4][9]. These extensions are based on the same primitives, but active and passive objects can coexist in the same application. On these extensions new language constructs are introduced that indicate which objects are active and which calls can be performed asynchronously [11][17]. These approaches require source code modifications to introduce parallelisation statements, resulting in tangled code, where partition, concurrency and distribution issues are mixed with object definitions. Our approach uses equivalent concurrency constructors but moves parallelism related issues to (un)pluggable modules.

OpenMP [18] uses annotations to introduce parallelisation concerns. Annotations soften the transition from sequential to parallel code and allow unpluggability of parallelisation concerns, since annotations can be ignored by the compiler for a strict sequential execution. However, parallel code is not modularised, since annotations are spread through functional code, becoming difficult to reuse.

Reflective systems or meta-level architectures [10][23] provided an early attempt to separate concurrency and distribution issues from core functionality. However, these approaches yield inefficient code, due to reification and meta-level code is difficult to understand and reuse. Probably the most successful separation between core functionality and parallelisation concerns was achieved in functional languages, where parallelism structure is expressed by a skeleton, using high order functions [6][12][19]. Generative patterns approach [14] follows a similar path. Parallel code is generated and the programmer must fill the provided hooks with core functionality. Aspect oriented programming differs from these previous approaches since is uses a different way to compose core functionary and parallel code.

The work in [15] was one of the roots of aspect oriented programming, by proposing a domain-specific language that separates core functionality from concurrency and distribution issues. AspectJ is a more mature and general-purpose language and was used recently in [16][21] to introduce distribution concerns into sequential applications. In [2], an attempt is made to move all parallelism related issues into a single module. Our methodology was influenced by all these previous efforts but differs from them in that we use a more fine-grained decomposition of parallelisation concerns, using a set of modules that can be (un)plugged or switched. This leads to a higher reuse potential and more incremental application development.

3. Overview of AspectJ

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

With static crosscutting it is possible to introduce instance variables or methods into a class, without changing the base class source code. It is also possible to declare a class to implement an interface or to extend another class. Figure 1 presents a point class and Figure 2 presents an aspect that changes class *Point*, to implement interface *Serializable*, and to include an additional method, called *migrate*. For simplicity we do not strictly follow the AspectJ syntax.

```
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);
  }
}
```

Figure 1 - Point class

```
public aspect Static {

  declare parents: Point implements Serializable;

  public void Point.migrate(String node) {
    System.out.println("Migrate to " + node);
  }
}
```

Figure 2 - Example of a static crosscutting aspect

With dynamic crosscutting it is possible to replace object creations, method calls or instance variable accesses, using the *around* construct, or add behaviour before or after the event. In aspect code a special keyword, *proceed*, indicates where/when the original event executes. Figure 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 method's signature to intercept.

```
public aspect Logging {

  void around(void Point.move*()) {
    System.out.println("Move called");
    proceed();     // proceed the original call
  }
}
```

Figure 3 - Example of a dynamic crosscutting aspect

Each event (i.e., object creation or method call) is called joinpoint and a set of joinpoints is called pointcut (i.e., *void Point.move*()*). Reusable aspects can be developed using abstract pointcuts or interfaces. In both cases the abstract aspect only refers to abstract pointcut(s) or to interface(s). Each concrete reuse refines the abstract aspect by specifying concrete pointcuts or concrete classes that implement the interface.

An aspect construct specifies a crosscutting concern in a modular way, although it results in code executed in several places, using a process called weaving in AOP. This process, at compile time, composes aspect code into the target classes. For instance, the around code in Figure 3 would be called in both methods of *Point* class.

4. Proposed Methodology

The programming model is based on a parallel object oriented language, where the base concurrency mechanism is asynchronous method invocations. In this type of method call, the client can proceed while the server processes the requested method in parallel. An asynchronous method invocation can be seen as a high-level equivalent of a message send in message passing interfaces like MPI. The most relevant difference from existing parallel object oriented languages [17] is that this type of method calls and object distribution are specified in a more centralised and modular way.

In our methodology, traditional object oriented mechanisms are used to implement application core functionality and parallelisation concerns are implemented using aspect oriented mechanisms. Parallelisation concerns are separated into four categories: functional or/and data partition, concurrency, distribution and optimisation. Each concern is implemented into a different module, which can be plugged or unplugged from application core functionality.

Core functionality is developed or reused from an existing application, and specifies the application base functionality, i.e., what the application is supposed to do. Partition modules specify how work is performed in an efficient way, using several processing elements; they implement either a functional or a data partition. Concurrency modules specify parallel execution among tasks and synchronisation requirements. Distribution modules perform object distribution among available resources and remote method invocations. Optimisation modules mainly tune application performance for a particular platform, optimising work distribution and inter-object communication.

Each parallelisation concern (i.e., aspect) intercepts object creations and/or method calls (i.e., joinpoints) and specifies an enhanced behaviour, replacing the event or including additional functionality before or after event execution. Intercepted events are specified in a quantified form (i.e., all calls to a specific method) and other modules are oblivious of the additional concern [5].

In this methodology the programmer must first design the parallel application to support a high level of modularity. This design phase mainly consists on developing the core functionality to provide adequate joinpoints to compose with parallelisation concerns.

Core functionality is implemented using traditional object oriented abstractions, in which client objects request task execution to server objects. It has two main goals: to expose application core functionality and to provide adequate functionality to support parallelisation by plugging partition, concurrency and distribution

aspects. A partition aspect intercepts joinpoints from the core functionality and performs work/data partition, composing object instances/method calls of core classes. This module introduces joinpoints that can be intercepted by concurrency, distribution and optimisation aspects. The concurrency module specifies events from partition or core functionality that are executed in parallel as well as synchronisation requirements. The distribution aspect specifies how objects are distributed among available computing resources and how they communicate. Finally, an optimisation aspect can be introduced. Each of these parallelisation concerns is detailed in the following subsections.

4.1 Partition

In object oriented parallel applications functional parallelism can be modelled by calling several methods in parallel, in the same or in multiple objects. Data parallel applications can be modelled by calling the same method in multiple objects in parallel, where each object contains a different piece of data. Two base mechanisms work together to achieve these types of parallelism: object duplication and method call split.

The first mechanism transparently replaces a single object in the core functionality by a set of objects (Figure 4). Objects created in aspect code are called aspect managed objects, since their lifetime is managed by the partition aspect. Objects in this set can compute in parallel (concurrency concern) and can be distributed among processing nodes (distribution concern). Object duplication is implemented by intercepting object creations or method calls in core functionality, by means of aspect code. When additional data is required to execute the requested task the partition aspect also specifies how and when each object in the set interacts with other objects to gather necessary data.

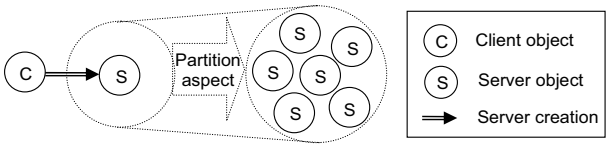

Figure 4 – Object duplication through aspect code.

The second mechanism splits a method call into several calls that can be processed in parallel, by one or several objects (Figure 5). This mechanism performs a data partition, in which data sent in a method call (as method parameters) is partitioned or replicated and forwarded to multiple objects. This mechanism can also distribute data across several objects, forwarding a piece of data to each object in the set.

These two mechanisms work together to achieve a partition module. Object duplication creates several object instances, while method call split transforms a single method call into several calls that can be executed in parallel by these aspect managed object instances.

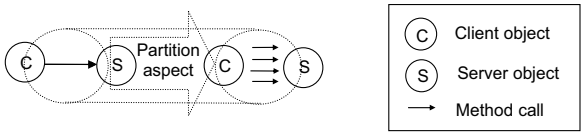

Figure 5 – Method call split through aspect code.

A single aspect specifies work partition code, keeping the core functionality oblivious from partition concerns, transparently changing object creation/method call semantics in core functionality. Object duplication is specified by intercepting the creation of objects and method split calls are specified by intercepting method calls, but it is also possible to perform object creations when intercepting method calls (e.g., in divide and conquer algorithms).

The design of core functionary to transparently support modular partition requires that classes from core functionality provide method(s) to process a subset of the data. The partition aspect must deal with data dependencies, replicating data and/or performing additional interactions among aspect managed objects to retrieve the relevant data. For instance, in iterative applications the full data set can be initially distributed into several objects in a block fashion. Iterations are broadcasted to all aspect managed objects. Between iterations, the partition code must exchange updated data among objects, required for the next iteration.

4.2 Concurrency

Concurrency is based on asynchronous method calls. In Java these calls can be implemented by spawning a new thread to perform the requested method call. Implementation of asynchronous method calls can be specified in aspect code that transparently changes the semantics of method calls. Concurrency aspect can intercept events either in core functionality or in partition code.

Asynchronous method invocations may also require synchronisation code to protect shared objects, avoiding data races and to ensure a specific execution order. Synchronisation code is also placed in concurrency aspect and it resorts to synchronised blocks and monitors provided by Java.

Partition and concurrency code are specified in separate modules. The main idea is first to develop a partition module and next to develop the concurrency module, i.e., partition and concurrency concerns are placed in their respective aspects. This way it is possible to (un)plug concurrency for debugging and it also helps to avoid the inheritance anomaly problem [20]. However, it

also means that the program must be valid without concurrency, something similar to OpenMP [18]. It is also possible to merge partition and concurrency into a single module; however, additional benefits arise when these concerns can be modularised in different aspects.

Separation between partition and concurrency code requires more design effort, since the partition code must provide adequate joinpoints to allow composition with concurrency code. One example is future type method calls. It is possible to design the partition code in a way that the concurrency module can transparently introduce this type of method calls [3].

4.3 Distribution

Our programming model is based on distributed objects: each object is a unit of distribution. The key idea is to implement code related to object distribution into another module. There are two main benefits from this approach: partition and concurrency modules can be developed without dealing with object distribution issues (i.e., they are developed for a single processor/shared memory machine) and it becomes easier to switch among underlying middleware implementations for distribution concerns, such as CORBA, Java RMI and MPI.

There are several alternatives to implement object distribution; one of them is Java Remote Method Invocations (RMI). However, various changes are required in the source code to use Java classes as remote classes within RMI. With AOP this code can be isolated in a single aspect. It is also possible to use a combination of middleware implementations, for example, using MPI for performance critical parts, and Java RMI in the remainder parts of the application.

Distribution requires an aspect that intercepts both sides of a method call: in the caller object to transparently redirect the call to the distribution middleware; in the called object node to receive the call and forward it to the target object.

Distribution aspect is also responsible by the selection of the most adequate node for a particular object instance. Several policies can be implemented in this aspect (e.g., random, round-robin).

4.4 Optimisation

Optimisation code can make the parallel code very hard to understand since it generally suffers from two classic problems: code tangling and code scattering. Code tangling arises since the optimisation code is mixed with the class base functionality. Code scattering arises since one optimisation may be implemented in multiple classes.

Aspects provide a way to modularise optimisations, becoming easier to experiment various alternative optimisations, by plugging or unplugging each optimisation aspect. However, only optimisations based in joinpoints can be modularised by aspects. Examples are: thread pools, cache objects, communication packing and replicated computation.

5. Case Study – Prime Number Sieve

This case study shows how to apply the proposed methodology to a particular case study, using AspectJ, and illustrates several ideas presented in previous section. The presented code samples are written in a simplified version of AspectJ to enhance code readability and ease of understanding.

The case study is a prime number sieve to calculate all primes up to a predefined number. The candidate numbers are placed in a list in increasing order. First, all multiples from the lowest number are removed from the list (initially all multiples of 2). Afterwards, all multiplies of the next lowest number are removed (multiples of 3 in the second step). The process is repeated up to the largest number. All numbers remaining in the list are primes.

A parallel prime number sieve pre-calculates the primes up to the square root of the largest number and distributes these numbers by a set of objects (i.e., processes) connected in a pipeline structure. Each object filters multiples of a range of prime numbers, receiving numbers to filter from the previous pipeline element and sending to the next element the pipeline, numbers that still to be filtered. Each number that reaches the end of the pipeline is a prime number. The next sub-sections detail how this application can be designed and implemented following the proposed methodology.

5.1 Core Functionality

The object oriented sieve is based on a two-step filtering. First, it calculates all primes up to the square root of the maximum number. Afterwards, the rest of the numbers are placed in a list and divided by all the first calculated primes. These steps are executed, respectively, in object constructor and in method *filter*. The object constructor indicates the range of primes to filter and the method *filter* removes all non-prime numbers from the list of candidate numbers. The following code provides a skeleton of these two methods:

```
public class PrimeFilter {

  // calculates primes between [pmin,pmax]
  public PrimeFilter(int pmin, int pmax);

  // remove non-primes from num list
  public void filter(int num[]);

}
```

The following code uses this core functionality to calculate all primes up to predefined maximum:

```
void public static void main(String[] arg) {
  int list[] = ... // place numbers in the list
  PrimeFilter p = new PrimeFilter(2, sqrt(Max) );
  p.filter(list);  // filters the list
}
```

Figure 6 presents an interaction diagram corresponding to the above code. It is a fully functional sequential version of the prime number sieve for a single processor machine. The next subsections show how to develop modules to add parallelisation concerns to this core functionality.

Figure 6 – Core functionality of prime number sieve

5.2 Partition Module

The overall structure of partition code is presented in Figure 7; it consists of three parts: (1) object duplication to use a pipeline of prime filters instead of a single filter; (2) method call split to divide a large pack of numbers into smaller packs that can be processed in parallel and (3) method call forward to propagate a method call made to the first object in the pipeline to all objects in the pipeline (3). These tasks are modularised within *Partition* aspect, whose code sketch is presented in Figure 8.

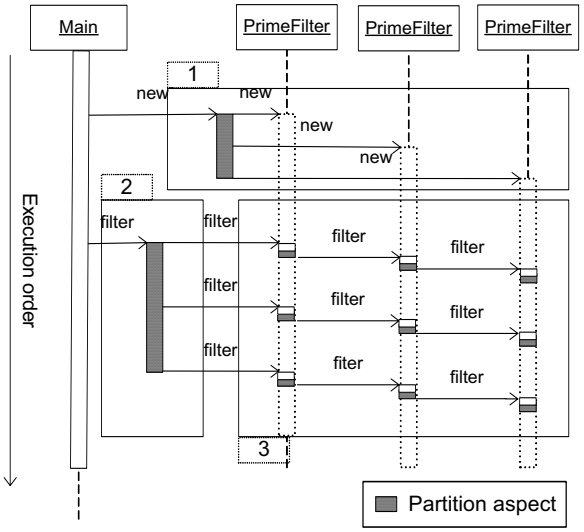

Figure 7 – Interaction diagram after weaving core functionality and partition aspect

```
aspect Partition {

  // pointer to the next pipeline element
  HashMap next = new HashMap();
  ... // other local variables and functions

  // prime filter object duplication
  PrimeFilter around (PrimeFilter.new(..)) ... {
    PrimeFilter p, oldp = null;
    // create pipeline filters in reverse order
    for(int i=numberOfDuplicates; i>0; i--) {
      // create filter with specific parameters
1     p=new PrimeFilter(/*..*/);
      next.put(p,oldp);  // save filter sequence
      oldp = p;
    }
    return(p); // return first pipeline element
  }

  // filter method call split
  void around (PrimeFilter.filter(..)) ... {
    PrimeFilter p = ... // get target object
2   packs = ... // split into a set of packs
    for(int i=0; i<numberOfPacks; i++) {
      p.filter(packs[i]);
    }
  }

  // forward calls among pipeline elements
  void around (PrimeFilter.filter(int num[]) ){
    PrimeFilter p = ... // get target object
3   proceed(num);  // invoke filter(num)
    if (next.get(p)!=null))  // if has next
      (next.get(p)).filter(num); // invoke
  }

}
```

Figure 8 – Partition code sketch

Partition aspect consists of three parts. The *next* variable registers prime filter pipeline sequence.

The first part (1 - *around (PrimeFilter.new()*) carries out object duplication. It changes the semantics of the *PrimeFilter* constructor to create a predefined number of filters. The first prime filter created is returned to the client (i.e., call to *PrimeFilter.new* in core functionality). This pointcut only intercepts object creations in the core functionality.

The second part (2 - *around (PrimeFilter.filter(..))*) changes the semantics of calls to *filter* method in core functionality, performing method split. It takes a pack of numbers to filter, received as a parameter, splits this pack into a set of packs and performs a series of *filter* calls, each call with a different pack.

The last part (3 - *around (PrimeFilter.filter(int num[])),* propagates *filter* calls, performed in the first pipeline element to all pipeline elements. This code also applies recursively to the *filter* method (i.e., also intercepts aspect calls to method *filter*, see Figure 7, block 3).

In the core functionality a **single** *PrimeFilter* filters a **single** pack of numbers. After applying the partition module, a **pipeline** of *PrimeFilters* receives **several** packs of numbers that can be processed in parallel. This change has been performed without changing the core

functionality. The partition aspect completely modularises and encapsulates the pipeline functionality. Also, partition code can be unplugged to improve or debug core functionality.

After having modularised the pipeline functionality, the natural step is to develop a reusable module encapsulating this type of partition. Common behaviour of a pipeline includes the three steps presented before:

1 creation of pipeline elements, including the management of next associations;
2 split of a core functionality call into several calls;
3 forward of method calls among all pipeline elements.

In AspectJ we can use abstract pointcuts or marker interfaces to develop reusable aspects. Figure 9 presents the code sketch of an abstract pipeline using marker interfaces. Two main refactorings were performed: references to pipeline elements became marker interfaces and each method parameter became a generic Object. Concrete pipeline elements must implement the *Pipe* interface in a concrete aspect that inherits from this abstract aspect.

```
abstract aspect PipelineProtocol {

  // generic pipeline element
  public interface Pipe {
    // o = pipe parameters, i = rank
    public void Pipe(Object o, int i);
    public void compute(Object o);
  }

  // pipe object duplication
  Object around (Pipe.new(..)) ... {
    Pipe oldp = null;
    ... // create pipeline elements as before
  }

  // method call split
  void around ( Pipe.compute(Object t) ) ... {
    Pipe p = ... // get target object
    ... // generate several compute calls
  }

  ... // call forward among pipeline elements

}
```

Figure 9 – Reusable code sketch for a pipeline

In a simple farming parallelisation each filter has ALL the primes up to the square root of the maximum number and each pack of numbers can be processed by ANY *PrimeFilter* (Figure 10). This requires two changes to the code in Figure 8: the *PrimeFilter* constructor parameters are broadcasted to all duplicated objects (change in line *new PrimeFilter* of block 1 and the *next* variable becomes a vector) and each call to the filter method is forwarded to a single filter (change in *if (next.get(p)!=null))* statement in block 3 to select one *PrimeFilter* in the set). In this particular case block 2 does not require any changes.

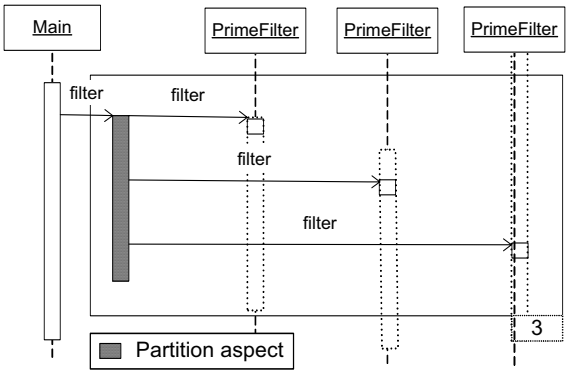

Figure 10 – Farm parallelisation strategy

5.3 Concurrency and Distribution Code

Concurrency can be added to the previous code by spawning a new thread to execute each *filter* method call. However, each *PrimeFilter* object must be protected against concurrent invocations to avoid data races, since its implementation is not thread safe. Figure 11 shows an interaction diagram after weaving the core functionality with partition and concurrency aspects and Figure 12 shows the code of concurrency aspect.

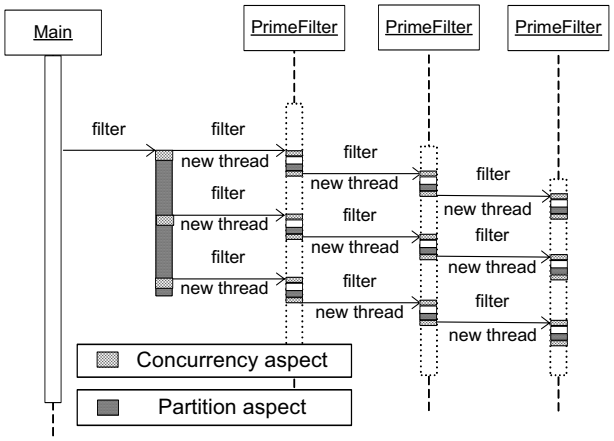

Figure 11 – Interaction diagram after weaving partition and concurrency aspects

```
aspect Concurrency {
  void around( PrimeFilter.filter(..) ) {
    (new Thread() { // execution in a new thread
      public void run() { proceed(); }
    }).start();  // starts the thread
  }
  void around( PrimeFilter.filter(..) ) {
    // synchronise thread access
    synchronized(/* target object */) {
      proceed();
    }
  }
}
```

Figure 12 – Concurrency aspect

Introducing the concurrency aspect allows the application to take advantage of parallel shared memory machines. However distribution code is required to use distributed memory machines.

Distribution concerns can be implemented with Java RMI. This module creates remote object instances and redirects local calls to remote objects instances (Figure 13). Four code changes are required to use Java RMI:

1. Distributed objects must implement an interface, this interface must extend the *Remote* interface and all its methods must throw a *RemoteException*.
2. Each remote object must be instantiated, exported to be externally available and registered in a name server to allow external references to it;
3. Client objects must contact the name server to obtain a initial reference to a remote object;
4. Remote method calls must include a *try { ... } catch* statement to deal with *RemoteExceptions*;

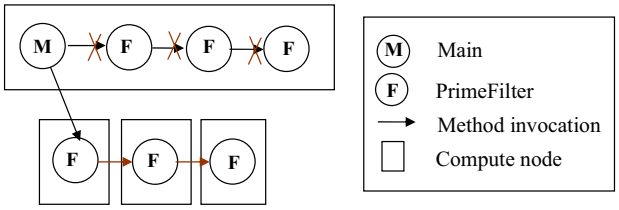

Figure 13 – Distribution concern

The first two are server side modifications, whereas the last two are client side modifications. Without using AOP these modifications involve changes in several places in code. Using AOP it is possible to concentrate these changes in a single module (Figure 14). Additionally, this module can be (un)plugged to provide support for distributed memory machines.

```
01 aspect Distribution {
02
03   // Registers servers in RMI (server side)
04   public static void PrimeFilter.main() {
05     PrimeFilter ps = new PrimeFilter(...);
06     ...    // registers object in name server
07   }
08
09   // catch PrimeFilter.new and associate
10   // with a remote one (client side)
11   int count=0;  // number of servers created
12   PrimeFilter around(PrimeFilter.new()) ... {
13     String name=new String("PS"+(++count));
14     remote=findRemoteObject(name, ...);
15     ...  // associate remote to the local ref.
16   }
17
18   // redirect calls to remote objects (client)
19   void around (PrimeFilter.filter(int num[])){
20     ... // exception handler logic
21     remote.filter(num); // redirects call
22     ... // exception handler logic
23   }
25 }
```

Figure 14 – RMI distribution code

The Prime filter class is declared to implement the required interfaces (code modification 1) and a new interface IPrimeFilter is introduced due to RMI requirements (code for both modification is not presented, but it is also modularised within distribution aspect). The method *PrimeFilter.main* implements modification 2 (lines 03-07). When a prime filter is created a remote JVM is contacted to provide a remote instance. The name of each remote instance is automatically generated, using the name PS<instance number> (modification 3, lines 09-16). Lines 18-23 redirect local calls to remote instances (modification 4).

Code to use different middleware can also be developed. For example, when using a message passing library the remote method invocation is performed through a message send and the main method contains a loop to receive incoming messages and call the method *filter*. Figure 15 presents a simplified example, using the Java MPP (Message Passing Package) library, which is based on the new Java nio package. It is also possible to develop a hybrid implementation, using MPP and RMI.

In these two examples object distribution is performed explicitly (in main method) but it is also possible to modularise more automatic object distributions, by using object factories and changing the way remote servers are contacted on object creations.

```
aspect DistributionMPP {

  public static void PrimeFilter.main() {
    PrimeFilter ps = new PrimeFilter(...);
    while(!end) {  // receive "filter" messages
      comm.recv(buf,  buf.length, ...);
      ps.filter(buf);
  }
  // catch PrimeFilter.new
  PrimeFilter around(PrimeFilter.new(..)) ... {
    ...
  }

  // redirect calls of filter to remote object
  void around (PrimeFilter.filter(int num[]) {
      comm.send(num, num.length, ...);
  }

}
```

Figure 15 – MPP distribution code

6. Performance Evaluation

The first test evaluates the overhead introduced by an AOP based approach. It compares the performance of a hand coded RMI version of the prime sieve against an AspecJ version, developed using the presented methodology. This test was performed on seven dedicated dual processor Xeon 3.2 GHz (Hyper-thread enabled), 1 GB DDR2 400, running Linux CentOS 4.1, connected through a Gigabit Ethernet. The maximum prime number was set to 10.000.000 and there are 50 messages of

100.000 numbers (only odd numbers are sent to the pipeline). Presented values are median of five executions, using JVM 1.5.0_03 and AJDT 1.3.0. Figure 16 shows that the performance penalty introduced by the proposed methodology is very small (less than 5%). It mainly occurs due to code that is no longer inlined in object classes but placed in separated classes by the AspectJ compiler.

The prime sieve does not scale well using a pipeline parallelisation strategy. This is due to the amount of messages generated among filters (each message must cross all pipeline elements). However, this was one of the reasons to select the prime filter for this test, since the aspect code executes in multiple places in the code (see Figure 11).

The second test compares execution times (Figure 17) of AspectJ versions using combinations of modules presented in section 5 (Table 1). These versions were obtained by plugging or unplugging one or several modules. When using a single machine (i.e., using only 2 filters) the best parallel application is obtained without including the distribution aspect; this application is targeted for shared memory systems. However, this version cannot take advantage of more than 4 filters, since it does not include distribution code. The farm strategy is better than a pipeline partition strategy in all cases. The MPP middleware leads to lower execution times since it introduces lower communication overhead, when compared to Java RMI. Fortunately, using our methodology it is easier to exchange the parallelisation strategy. We also present results using a dynamic farm parallelisation. In this application the dynamic farm only introduces a small improvement since there are not load imbalances in a normal farming strategy (i.e., with a static work allocation). The dynamic farm is an example where we were not able yet to separate partition from concurrency issues.

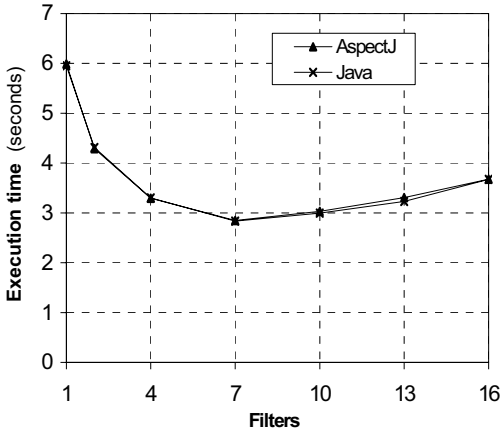

Figure 16 - Performance of Java versus AspectJ

	Partition	Concurrency	Distribution
FarmThreads	Farm	Yes	No
PipeRMI	Pipeline	Yes	RMI
FarmRMI	Farm	Yes	RMI
FarmDRMI	Dynamic Farm		RMI
FarmMPP	Farm	Yes	MPP

Table 1 – Tested module combinations

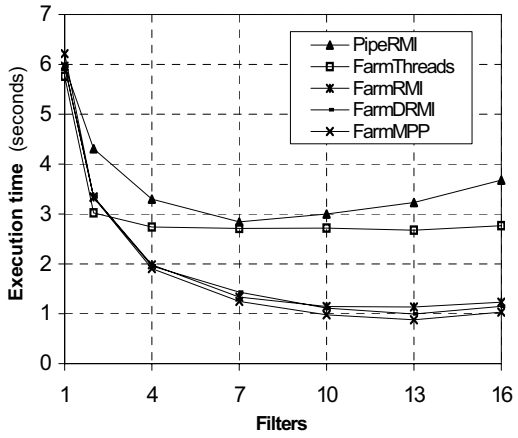

Figure 17 - Performance of AspectJ versions

Conclusion

This article presents a design methodology to develop parallel applications, based on the modularisation of core functionality, partition, concurrency, distribution and optimisation code. By separating these concerns applications become easier to develop and maintain, since code is more modular and easier to reuse. In this methodology, most of the code is developed using sequential programming; concurrency and distribution aspects are introduced in a later development phase and can be (un)plugged on the fly. Also, it is possible to exchange one aspect by another, for example, exchanging a pipeline by a farm partition.

AOP requires fewer changes to existing code to work in a parallel environment than current alternatives, namely templates and skeletons. This is due to the larger number of alternatives to compose core functionality with parallelisation code. With AOP it becomes feasible to develop a working core application and to incrementally add parallelisation concerns. Additionally, these concerns can be unplugged at any development phase, which also makes debugging easier. In our experience, most of the changes are performed in a single module and even when

the required modifications break the modularity they can be incrementally propagated to the rest of the modules.

The article presented a simple case study. However, we have developed parallelisation strategies for the three most common categories: pipeline, farm with separable dependencies and heartbeat. Our felling is that code reuse is high, since moving from a parallel application to another using the same parallelisation strategy is performed by copying the parallelisation aspects and updating these modules to the new application. Additionally, it is possible to develop reusable abstract aspects to model the common behaviour. Currently, experiments are also ongoing with other benchmarks with more complex task dependencies. We are also developing a domain-specific aspect library for parallel computing, based on reusable aspects.

Acknowledgments

Thanks to Miguel Monteiro for his comments on this paper. Also thanks to all PPC-VM members for their contribution to this work.

References

[1] A. Yonezawa. ABCL: an Object-Oriented Concurrent System, MIT Press, 1990.

[2] B. Harbulot, J. Gurd. Using AspectJ to Separate Concerns in Parallel Scientific Java Code, *Third International Conference on Aspect Oriented Software Development (AOSD'04)*, Lancaster, UK, March 2004.

[3] C. Cunha, J. Sobral, M. Monteiro, Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms, *Fifth International Conference on Aspect Oriented Software Development AOSD'06*, Bonn, Germany, March 2006.

[4] E. Dekel (Ed). Java on Clusters, *Special Issue of Journal of Parallel and Distributed Computing, 60(10)*, October 2000.

[5] E. Filman, D Friedman. Aspect-Oriented Programming is Quantification and Obliviousness, *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, Minneapolis, October 2000.

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

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, Getting Started with AspectJ. *Communications of the ACM, 44(10)*, October 2001

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin. Aspect Oriented Programming, *ECOOP '97*, 1997

[9] G. Wilson (Ed). Parallel Programming Using C++, MIT Press, 1996.

[10] H. Masuhara, A. Yonezawa. Design and Partial Evaluation of Meta-Objects for a Concurrent Reflective Language, *ECOOP'98*, July 1998.

[11] J. Briot. R. Guerraoui. K. Lohr. Concurrency and Distribution in Object Oriented Programming, *ACM Comp Surveys*, 30(3), Sept. 1998.

[12] J. Darlington, Y. Guo, H. To, J. Yang. Parallel Skeletons for Structured Composition, *PPoPP'95*, Santa Clara, USA, 1995.

[13] J. Hannemann, G. Kiczales. Desing Pattern Implementation in Java and AspectJ, *OOPSLA '02*, November 2002.

[14] 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.

[15] C. Lopes, D: A Language Framework for Distributed Computing, Ph.D. thesis, College of Computer Science, Northeastern University, Boston, USA, November 1997.

[16] M. Ceccato, P. Tonella. Adding Distribution to Existing Applications by means of Aspect Oriented Programming, In *Proc. of the 4th IEEE Int. Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, Chicago Illinois, USA, September 2004.

[17] M. Philippsen. A Survey of Concurrent Object-Oriented Languages, *Concurrency: Practice and Experience, 10(12),* August 2000.

[18] OpenMP architecture review board, OpenMP Application Program Interface, Version 2.5, May 2005, www.openmp.org.

[19] P. Trinder, K. Hammond, H. Loidl, S. Jones. Algorithm + Strategy = Parallelism, *Journal of Functional Programming, 8(1)*, January 1998.

[20] S. Matsuoka, A. Yonezawa, 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)*, MIT press, 1993.

[21] S. Soares, E. Loureiro, P. Borba. Implementing Distribution and Persistence Aspects With AspectJ, *OOPSLA '02,* November 2002.

[22] T. Elrad, R. E. Filman, A. Bader. Aspect Oriented Programming, *Communications of the ACM, 44(10)*, October 2001

[23] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte H. Tezuka, H. Konaka, M. Maeda, K. Kubota. Design and Implementation of Metalevel Architecture in C++ - MPC++ Approach, *Workshop on Reflection and Metalevel Architecture*, April 1996.