Pluggable Parallelisation

Rui C. Gonçalves, João L. Sobral
Departamento de Informática
Universidade do Minho
4710-057 Braga,
PORTUGAL

[rgoncalves, jls]@di.uminho.pt

ABSTRACT

This paper presents the concept of pluggable parallelisation that allows scientists to develop "sequential like" codes that can take advantage of multi-core, cluster and grid systems. In this approach parallel applications are developed by plugging parallelisation patterns/idioms into scientific codes (e.g., "sequential like" codes), softening the move from sequential to parallel programming and promoting the separation between domain specific code and parallelisation issues. Pluggable parallelisation combines three characteristics: 1) parallelisation is performed from "outside to inside", localising parallelisation concerns into well defined modules, reducing changes required to the domain specific code and avoiding invasive parallelisation of base code; 2) control view is separated from data view promoting a stronger separation of concerns which improves reuse of parallelisation concerns across platforms and enables fine-grained refinements; and 3) abstractions can be composed, supporting the development of more complex patterns based on fine-grained features. This paper presents the concept of pluggable parallelisation and shows how some well-known parallelisation strategies can be implemented in this approach. Results show that this is a feasible approach and performance is competitive with traditional parallel programming.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming - *Parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features – *Concurrent programming structures.*

General Terms

Performance, Design and Languages.

Keywords

Parallel programming, non-invasive parallelisation, separation of concerns.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HPDC'09, June 11–13, 2009, Munich, Germany.
Copyright 2009 ACM 978-1-60558-587-1/09/06...$5.00.

1. INTRODUCTION

Parallel computing is being pushed to the mainstream by the advent of multi-core machines and grid systems. The number of cores on every desktop machine (currently 2 or 4 cores per machine) will keep increasing. This requires a massive migration of current sequential applications to this new reality, introducing a strong pressure for new parallel programming paradigms that can help on this move.

```
public class JGFLUFactBench
          extends Linpack implements JGFSection2{

  public static int nprocess;
  public static int rank;

  public void JGFinitialise() throws MPIException{
    int r_count,z_count;
    int p_ldaa;
    n = datasizes[size];
    ipvt = new int [ldaa];
    p_ldaa = (ldaa + nprocess - 1) / nprocess;
    rem_p_ldaa = (p_ldaa*nprocess) - ldaa;
    /* ... */

    if(rank==0) {
      long nl = (long) n;
      ops = (2.0*(nl*nl*nl))/3.0 + 2.0*(nl*nl);
      norma = matgen(a,lda,n,b);
    }

    if(rank==0) {
      for(int i=0;i<a.length;i++){
       if(r_count==0) {
         for(int l=0;l<a[0].length;l++){
           buf_a[z_count][l] = a[i][l];
         }
         z_count++;
       } else {
         MPI.Send(a,i,1,MPI.OBJECT,r_count,10);
       }
       buf_list[i] = z_count - 1;
      }
    } else {  // rank!=0
      for(int i=0;i<real_p_ldaa;i++){
        MPI.Recv(buf_a,i,1,MPI.OBJECT,0,10);
      }
    }
  }
```

Figure 1. LuFact MPI-Based parallelisation

The paradigm with the highest probability of success would resemble traditional sequential programming and should maximise reuse of existing sequential codes. Unfortunately, currently most parallel programming languages require extensive and invasive source code changes to enable codes to take

© ACM, 2009. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in Proceedings of the 18th ACM international symposium on High performance distributed computing http://dx.doi.org/10.1145/2162024.2162034

advantage of parallel systems. For instance, in MPI the programmer must insert code to perform data partitioning among MPI processes and to coordinate the execution and data moves among processes. As a consequence, parallelisation issues become tangled with domain specific code [8] and changes are non-reversible. In large scale applications this can limit application modularity and maintainability, and consequently, the ability to evolve application code in an independent manner.

We illustrate these issues by presenting code a snippet of the JGF MPI implementation of the LuFact [15] (Figure 1), a Java version of the popular Linpack benchmark.

The code is shown in three colours (fonts): i) the basic functionality (LuFact) in black; ii) MPI control related issues in red (grey) and iii) data partition issues in blue (italic). These three concerns are tangled, harming modularity, understandability, etc.

This paper proposes an approach that addresses these issues by supporting a traditional "sequential like" style of programming. Parallelisation is performed by "plugging" a set of parallelisation patterns/idioms that transform "sequential like" codes to take advantage of parallel systems (e.g., parallel codes). This approach modularises parallelisation issues into a set of transformations, enabling independent development of domain-specific and parallelisation issues, and reducing changes required to the basic code. These are key features to promote a broader usage of parallel processing, as most software companies will only have a few experts in parallel computing, although most programmers would be more familiar with a "sequential like" style of programming. Moreover, it promotes reuse of legacy code.

This article is organised as follows. Section 2 presents the pluggable parallelisation approach. Section 3 performs an evaluation of this approach with two case studies, including performance results. Section 4 discusses the limitations of the approach and section 5 presents related work. Section 6 concludes the paper presenting directions for further research.

2. PLUGGABLE PARALLELISATION

Pluggable parallelisation addresses the development of parallel applications by providing abstractions to transform domain-specific code into parallel code. The process starts with a "sequential like" code that is transformed into a parallel code by using a set of parallel programming abstractions. In this process the programmer can also provide additional parallelisation specific code.

This "transformation oriented" view presents several advantages over more traditional approaches. Parallelisation is performed "from outside to inside" minimising changes required to the base code and can be applied to legacy code that cannot be made parallel with parallelising compilers. Moreover, keeping these transformations in separate modules enables independent development: it is easier to maintain/change parallel code and programmers that write domain-specific code can be oblivious of parallelisation issues.

This approach introduces several questions:

1) What kind of transformations to support?

2) How to specify transformations?

3) How to combine transformations to achieve more complex ones?

4) How to express common parallel patterns in this approach?

5) What are the main limitations of this type of approach?

The next sub-section contributes to answer to the first and second questions. The following sub-sections focus on the third and fourth questions. The discussion and related work sections discuss the main limitations of this approach.

2.1 Parallel Programming by Transformation

For several years we have been converting sequential Java codes to their equivalent parallel counterparts. One interesting source of case studies was the Java Grande Forum (JGF) benchmark [15]. This benchmark includes well known computational kernels that exist in many scientific codes (some kernels were simply taken from the SciMark benchmark). This benchmark includes parallel versions (e.g., MPI based) of most of these well known sequential codes. The main question was: *how could a programmer transform sequential versions into parallel equivalents without directly inserting parallelisation code into the source, avoiding tangling presented in Figure 1?*

Parallelisation of the JGF benchmarks follows a SPMD model: all nodes execute the same sequential code, but each node works on a different part of the data and certain operations are performed by a subset of nodes. Additional data moves among nodes may be required, if each part of data can not be processed independently from others. This "natural style" of moving from sequential to parallel codes was confirmed by requesting undergraduate students, in parallel computing courses, to develop parallel versions of sequential Java codes. They tended to follow an approach similar to the one used in the JGF benchmarks.

Based on this set of experiences, we devised a set of programming abstractions that could help to perform code transformations for parallelisation. Identified transformations specify actions like:

1. Block *B* of sequential code executes on all nodes;

2. Block *B* of code executes on nodes *N* where condition *C* holds;

3. Data structure *D* is partitioned among nodes using strategy *S*;

4. Update remote data at execution point *E*.

Non-invasively applying these transformations to scientific codes requires a way to identify, *from outside* (i.e., through a unique global name), blocks of code *B*, data structures *D* and execution points *E*. For this purpose we resorted to an object-oriented language as the basic language. Object-oriented languages offer a richer set of abstractions (e.g., classes, methods, fields), besides traditional structured programming abstractions. As a consequence, they provide a richer set of alternatives to identify program elements.

We implement transformations upon the concept of object aggregate [1][17]. Object aggregates are objects that are replicated on all nodes (i.e., each node has a local instance of the object). The idea is to transform object creations in the sequential code into creations of object aggregates in the parallel code (i.e., transformed code). Subsequent method calls (originally in the sequential code) can be performed by the local aggregate representative (by default) or by all elements of the aggregate. It is possible to restrict method executions to specific aggregate

elements. For that purpose a condition can be provided that can depend on the node identifier, method parameters and object data.

Operations of type 1) and 2) can be mapped into method executions that are (conditionally) executed by aggregate elements. Operations of type 3) and 4) are expressed by operations controlling how data stored in objects is distributed / updated among aggregate elements at specific execution points.

Operations of type 1) and 2) manage task coordination (i.e., control view) and operations of type 3) and 4) manage data distribution and dependence issues (i.e., data distribution view). The proposed approach promotes the management of these issues as separate concerns, by managing these two views as separate as possible. The control flow view specifies how a sequential control flow is transformed into multiple control flows and how to coordinate parallel control flows. Data distributions specify how data is partitioned among parallel tasks and what data moves are required to keep data consistent. This separation introduces several benefits: 1) it allows a more incremental development and understanding (e.g. first understanding the control flow and then the data distribution strategy); 2) it is conceptually possible to change one independently of the other, although we foresee that the common case will be changing data distributions (e.g., some parallelisation patterns differ only in data dependencies, requiring specific data distributions and moves).

Figure 2 illustrates this model. The sequential code creates a single object instance *obj1*. Method *call a* is performed on this instance, which issues a method *call b* to the same instance. After applying transformations of type 1), the resulting parallel code creates an aggregate of *obj1* (one object instance per node) and *call a* executes on all nodes. Method *call b* is also issued on all nodes, since it is nested inside *call a*. But, in this specific example, we explicitly restricted the execution of *call b* to the object on node 0, by composing a transformation of type 2).

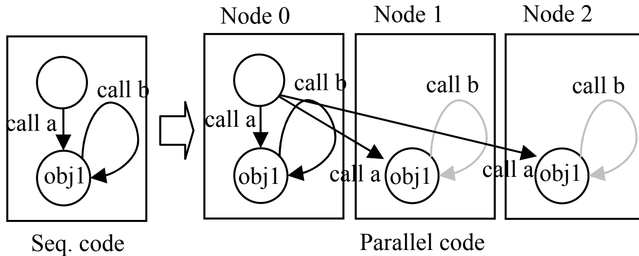

Figure 2. Parallelisation process based on object aggregates

By default each object aggregate manages its data in an independent manner (i.e., each aggregate has a local copy of the data). The same holds for static data allocated in static classes. We support several transformations that can change the default behaviour. The idea is to specify how data is distributed among aggregate elements. We support BLOCK and CYCLIC data partitions and global and local view of data. In the former, data indexes refer to the local partition, in the latter data indexes are relative to the global data structure. Figure 3 illustrates this transformation. There are transformations to inject code to change from one view to the other, during execution, and to update remote data. Moreover, since we start with a sequential code we allocate a shadow of the data in node 0. This is usually required since data initialisation performed in the sequential code usually needs to be performed in node with identification 0.

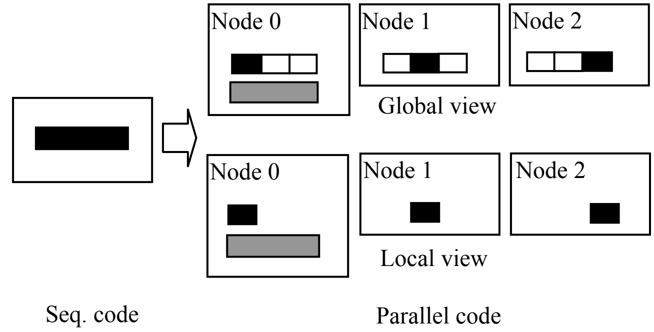

Figure 3. Global and local view of data

We also support non-SPMD code through remote objects and asynchronous method calls. In the former, an object in the sequential code can be placed on a remote node (chosen by the run-time system or specified by the programmer). In the latter, the method call is performed in a new thread of control.

The remainder of this section gives an overview of the template-based syntax used to express these transformations and presents common transformations. The two next sub-sections overview currently supported control and data transformations. The last sub-section discusses cases where these two views are more tightly coupled and presents sample code.

2.1.1 Template-Based Syntax

Code transformations are expressed in a template-based syntax (similar to C++ templates). Transformations can be applied to classes, methods (including parameters and return values) and data fields: these are the types of parameters in templates. A template-based syntax also supports the specification of composition of abstractions through template nesting (described in subsection 2.2). Table 1 presents relevant transformations currently supported.

Table 1. Transformations currently supported

Transformation	Description
Separate<T>	Instances of class T can be created on any cluster node
GridSeparate<T>	Instances of class T can be created on any grid node
Replicate<T>	Create an instance to class T on each available resource (e.g. node)
Broadcast<T,M>	Execute method call M on all aggregate elements (e.g., nodes)
CondExe<T,M,C>	Restrict execution of method M to places where condition C holds
Async<T,M,E>	Spawn a new thread to execute M, wait for the spawned thread at E
Partitioned<T,D, [BLOCK,CYCLE]>	Distribute the data field D, of class T, using a partition strategy
ChangeView<T,E,D, [Local|Global]>	Change field D from/to local/global view at execution point E
Scatter*/Gather* <T,E,D>or<T,E,M>	Update copy of field D or method parameter M at execution point E

Transformations have the following generic syntax:

TemplateName<
Class T
*[,Method M]**
*[,ExecutionPoint E]**
[,Condition C]
[,DataField D]
>

Class T specifies the class where the transformation applies, replacing instances of class T by a new type compatible version of the class. One example is *Replicate<T>* that specifies that one instance of class T should be created on each available computing resource (e.g., class *T* becomes an object aggregate). For compactness, in this article, we generally omit template parameter class T, although it is important to understand that transformations generally change the implementation of class T, so this parameter must be explicitly provided.

Method M and *ExecutionPoint E* are syntactically similar (both specify method executions) but they are used for different purposes. The former is used to specify blocks of code (e.g., a method) and the latter is used to specify execution points to plug code generated by the transformation. One example is the Async<T,M,E> template, that spawns a new thread to execute method *M* and waits for thread completion on execution point *E* (generally another method execution).

Parallelisation patterns may require case specific code. *Method M* can also be used for this purpose. In this case, the definition of M is given along with the template, instead of being defined in the sequential code.

Condition C specifies that a transformation takes place only on certain conditions. This is particularly important in SPMD code since some operations are frequently performed by a subset of tasks. One example is *CondExe<class, "someMethod", "iAmRoot?">*, which just executes *someMethod* if calls to *iAmRoot* return true.

2.1.2 Control Flow Transformations

These intend to transform a single control flow in the "sequential like" code into multiple control flows running in parallel. The most basic transformation is to replicate the same task on multiple nodes, which is achieved using *Broadcast<T, M>*. This executes the method *M* in all resources (e.g., nodes). It can only be applied to static methods. For non-static methods an additional *Replicate<T>* is required to create an instance of class *T* on every node before performing the broadcast. A more loosely coupled task creation can be specified by *Async<T, M>*, which spawns a new task to execute the method M. Some transformations require more than one method parameter. For instance, a refined Async can specify a second method (i.e., execution point) to wait for the spawned threads on a different execution point (e.g., *Async<T, M, E>*). Two other currently supported templates have the purpose of limiting/synchronising the execution of parallel control flows. *Barrier<T, E>* inserts a barrier at the specified execution point. *CondExe<T, M, C>* limits execution of method M to aggregate elements (or compute nodes) where condition C holds. Note that *Replicate<T>*, *Broadcast<T, M>* and *CondExe<T, M, C>* implement transformations of type 1 and 2 presented in section 2.1.

2.1.3 Data Distribution View

Data transformations support the specification of data partitions and updates among nodes (or aggregate elements, if the data is not stored in static fields). For this purpose we provide a set of pre-defined data partitions, including block and cyclic, that can be used of-the-shelf or extended to address application specific needs. *Partitioned<T, D, partitionType>* specifies that data field *D* will be partitioned among aggregate elements according to *partitionType* policy. The point where data is actually scattered or gathered (or reduced) is specified using templates *Scatter<T, E, D>* and *Gather <T, E, D>*. The reduce template requires an additional parameter specifying the reduce operator (this is another case where a method can be provided along with the template specification). These operations implement transformations of type 3 and 4 introduced in section 2.1. Conceptually, it is possible to provide a set of collective operations equivalent to MPI. The difference is that these manage information stored in class/object fields and the template parameter specifies the place in the code where the operation is inserted (by means of execution point *E*). More fine-grained data moves are supported by developing case specific templates (currently we must resort to AspectJ [11] to write these templates). One of such examples is the use of point-to-point messages (e.g., the JGF SOR benchmark). In that case a new template can be developed to perform the required data move. Our library of data partitions provides functions to access to local data (see Figure 3) within these templates.

2.1.4 Interplay of Transformations

Control flow transformations keep the data centralised, performing data initialisation at the root node. This behaviour is modified in three ways. 1) if data is initialised in an object constructor (e.g., the *Replicate* template implicitly calls the object initialisation method on each aggregate member); 2) when some data structure is used as a method parameter (e.g., in the *Broadcast* template all data is sent by value); 3) when data initialisation methods are broadcasted (or called from a broadcasted method).

Figure 4 illustrates how data distribution transformations change this default behavior in the JGF Series benchmark. Transformations are specified into a separate module, but for compactness and understandability purposes these were inserted as comments in the basic code, showing where the transformation will inject the parallelisation code.

```
class SomeClass {
...
  // Partitioned<SomeClass,TestArray,BLOCK>
  double TestArray[] = new ...
...
  void Do() {
    // ScatterBefore<SomeClass,Do(),TestArray>
    // ChangeView<SomeClass,Do(),TestArray,LOCAL>
    ...
    for (int i = 0; i < TestArray.length; i++)
      TestArray[i] = someComputation(/*.. */);
    }
    // GatherAfter<SomeClass,Do(),TestArray>
    // ChangeView<SomeClass,Do(),TestArray,GLOBAL>
  }
}
```

Figure 4. JGF Series data distribution view

The *Partitioned<SomeClass,TestArray,BLOCK>* injects code to support the block-wise distribution of *TestArray* among aggregate elements. *ScatterBefore<SomeClass,Do(),TestArray>* injects code to update each partition before the execution of method *Do* (in this case, the template injects a MPI_Scatter function in the sequential code). *ChangeView<...,Do(),TestArray,LOCAL>* makes each process to switch to the local view of data. By changing to the local data view, after that execution point, the variable *TestArray* refers to the local block of data (e.g., the *TestArray.length* is the size of the local block). The reverse operations are performed at the end of method *Do*: data is again collected in the master and the view is changed to global. Note that there are means to change the data view independently from the scatter and gather operations. This allows performing scatter/gather and change view operations in different execution points, which, in some cases, may lead to more efficient programs, by scattering/gathering the data before it is actually needed.

Figure 5 presents the complete example, now including the control view. In this case, *SomeClass* is transformed into an object aggregate (by applying the *Replicate* template) and calls to the *Do* method are executed by all members of the aggregate (by applying the *Broadcast* template). An additional *Separate* transformation can distribute instances of *SomeClass* across nodes of a cluster.

Core functionality

```
class SomeClass {
  double[] TestArray = ... // initialise array
  void Do() {
    for (int i = 0; i < TestArray.length; i++)
      TestArray[i] = someComputation(/*.. */);
  }
}
```

Parallelisation (data view)

```
Partitioned<SomeClass,TestArray,BLOCK>
ScatterBefore<SomeClass,Do,TestArray>
GatherAfter<SomeClass,Do,TestArray>
```

Parallelisation (control view)

```
Replicate<SomeClass>
// all aggregate elements execute method Do
Broadcast<SomeClass,Do>
```

Figure 5. Example of separation of data and control view.

2.2 Composing Transformations

Composition issues can arise when multiple transformations have impact in the same place in the sequential code (e.g., a method). In addition, building complex parallelisation and the ability to extend existing templates requires the composition of several code transformations.

The composition model builds upon an incremental development process: each transformation generates new code that can be transformed by another template. The key point is that each template might have impact on additional methods or execution points introduced by transformations previously applied. This enables the generation of different parallel code by composing transformations in different orders. Thus, a small set of templates can be used to generate a larger range of parallel programs. Moreover, additional tools can be provided to ensure that only valid compositions are allowed.

Figure 6 presents the code generated (for a shared memory machine) by applying a sequence of *Replicate*, *Broadcast* and *Async* transformations. Each call to *someMethod* is executed by a new thread on each aggregate element.

"Sequential like" code

```
public class SomeClass {
   void someMethod () { … }
}

SomeClass f = new SomeClass();
f.someMethod();
```

Generated parallel code

```
...
for (int i=0; i<numOfReplicas; i++) {
  agg.add(new SomeClass());     // Replicate
  new Thread() {                // Async
    void run() {
      agg.elementAt(i).someMethod();
    }
  }.start();
}
```

Figure 6 - Code resulting from the application of the sequence of transformations: *Replicate<SomeClass>*, *Broadcast<SomeClass, "someMethod">* and *Async< SomeClass, "someMethod">*.

In this case, the *Replicate* template injects code to create an aggregate of instances of *SomeClass*. The default behaviour would be to issue the call to *someMethod* only in the aggregate representative. By applying the *Broadcast* transformation to *someMethod* it will be executed in all aggregate members. The *Async* template acts upon the result of the *Broadcast* template, by issuing all method calls in a new thread (i.e., it also has impact on method calls generated by the *Broadcast* template).

A different parallel program can be generated by changing the order of *Broadcast* and *Async* transformations. In that case, a single thread will sequentially perform the call to each aggregate element, since first the asynchronous call is applied and then the broadcast.

A composition of transformations is specified by nesting templates. The composition in the example of Figure 6 would be specified as *Async<Broadcast<Replicate<SomeClass>,…>,…>)*. This syntax may become cumbersome for complex compositions. One problem is the lack of line breaks. This issue is solved by introducing a syntax that allows to *store* each program transformation into a variable and introducing one additional template parameter that specifies the program where the transformation applies. A special name (e.g., MAIN) represents the original "sequential like" code. Under this approach, we could write the transformation of Figure 6 as:

*prog1 = Replicate<**MAIN**,SomeClass>*

*prog2 = Broadcast<**prog1**,SomeClass,"someMethod">*

*prog3 = Async<**prog2**,SomeClass,"someMethod">*

This syntax introduces the possibility to specify a transformation tree (always starting in the "sequential like" program) instead of a single transformation chain. This allows a partial ordering of transformations (enough for many applications). Transformations acting on common methods/execution points should be completely ordered. Otherwise, the composition may produce an unpredictable result as it must select an order to apply transformations (e.g., the final result could be implementation dependent).

Template extensibility is also based on composition of templates (although we also provide means to build a new template from scratch, by using code templates in AspectJ [11]). To extend a template we compose additional functionality to that template. For instance, we could define a new template to implement a Farm based on *Replicate*, *Scatter* and *Gather*:

Farm<Class T, Method compute, DataField field>{

prog1= Replicate<T>

prog2 = Broadcast<prog1, T, compute>

prog3 = Scatter<prog2, T, compute, field>

prog4 = Gather<prog3, T, compute, field>

}

The last composition issue is related to the generation of code for specific target platforms. The proposed approach targets the generation of efficient code for a wide range of architectures, including multi-core, clusters, computational grids and systems composed of combinations of these. One common way to take advantage of clusters of multi-core machines is to use a mix of MPI and OpenMP. For this purpose the approach supports the *Separate* template to specify cluster-aware transformations and *GridSeparate* to specify transformations for computational grids. For instance, an aggregate of objects distributed through the nodes of a cluster, with a second level of inner aggregates can be specified by *Replicate<Separate<Replicate <SomeClass>>>*. This inner aggregate can more efficiently take advantage of multi-core processors by communicating through shared memory.

2.3 Expressing Common Patterns

This section revises some well known parallelisation patterns and outlines how they can be supported in the proposed approach. Patterns are directives, providing guidelines for solving classes of parallelisation problems. They are collections of solutions and there is no "one solution fits all". The proposed approach does not force a particular solution for each problem. Instead, it provides a set of parallelisation patterns that can be composed to address each specific case. The purpose of this section is to illustrate how to implement *one specific variant* of each of these patterns, namely, it shows how Farm, Pipeline, Divide & Conquer and Heartbeat can be plugged into "sequential-like" codes.

2.3.1 Farm

In the farm parallelisation the data is divided into independent parts, which are processed in parallel by several workers, and joined after processing.

The farm pattern can be plugged into "sequential like" code by transforming a single object instance into an aggregate of objects. This requires the specification of the class to be replicated, the method to process each task and split and merge functions (e.g., *Farm<Class T, Method compute, Method split, Method join*>). It can be implemented by composing *Replicate* with *Broadcast*, and providing methods to split and join the data, in a similar way to *Scatter* and *Gather* functions. However, in this case the *Scatter* and *Gather* act on method parameters and return value, instead of acting on data stored in object fields (although using an object field to store the data is also possible, as shown in section 2.2).

Figure 7 shows a farm pattern applied to the JGF RayTracer. Class *T* was replaced by the RayTracer class and the *compute* method becomes calls to *render* method.

Core functionality

```
RayTracer rt = new RayTracer();
Interval interval = new Interval(0,500);
int Result[] = rt.render(interval);
```

Farm parallelisation

```
Vector<Interval> split(Interval in) {
   … // split in into sub-intervals
}
int[] join(Vector<int[] in) {
   … // join rendered sub-images
}
Separate<Farm<RayTracer, render, split, join> >
```

Figure 7 - JGF RayTracer parallelisation.

This case shows how it is possible to provide case specific code to inject in the parallel code. Methods *split* and *join*, defined in the parallelisation, specify how to divide the *Interval* method parameter and how to merge the resulting integer array.

2.3.2 Pipeline

A pipeline consists of a chain of processes working in parallel on different parts of data. Each part of data is successively processed by all processes in the chain.

A pipeline can be plugged into sequential code by replacing an instance of a class by a pipeline of elements of the same class. Additional split and join methods can be used to divide the original data into independent pieces and to merge the processed pieces (e.g., *Pipe<Class T, Method compute[, Method split, Method join]*>. Another way to implement a pipeline, when the sequential code includes a chain of method calls, is to use the *Async* pattern.

2.3.3 Divide & Conquer

This pattern addresses problems that are recursively divided into simpler sub-problems that can be solved in parallel. The "sequential like" code where the pattern applies can be intrinsically divide & conquer (e.g., problems that are sequentially solved in a recursive manner). In this case, the parallelisation pattern spawns a new parallel task on each recursive call, using the *Async* template (with future type synchronisation [3], an approach similar to fork & join frameworks [12][14], but avoiding invasive changes in sequential code).

We illustrate this pattern (Figure 8) with the classic Fibonacci function (this is an example of how the proposed approach can support non-SPMD code).

Core functionality

```
public class Fib {
  long value;
  public Fib(long val) { value = val; }
  public long compute() {
    if (value <=1) return(value);
    else{
      Fib f1 = new Fib(value-1);
      Fib f2 = new Fib(value-2);
      Long r1 = f1.compute();
      Long r2 = f2.compute();
      return(r1.longValue()+r2.longValue());
    }
  }
}
```

Parallelisation

```
Async<Fib,"compute","Long.longValue">
Separate<Fib>
```

Figure 8 - Parallel computation of Fibonacci numbers.

Calls to *compute* methods are made asynchronously through the *Async* template. Calls to *Long.longValue* (i.e., the unboxing function that transforms an object Long into a long value) provide the execution point where a fake return value is replaced by the result of the computation. We also specify that instances of the *Fib* class can be placed on remote resources, by making them *Separate* objects, otherwise the application would run on a single machine.

A variant of Divide & Conquer is the search for the best solution (e.g., N-Queens). Recursive calls are only issued if they could lead to a better solution. This data dependence can be addressed by additionally plugging appropriate template to conditionally *Broadcast* data field at certain execution points.

Core functionality

```
public class Linpack {
...
 final int dgefa(double a[][],
         int lda, int n, int ipvt[]) {
  double[] col_k, col_j;
  double t;
  int j, k, kp1, l, nm1;
  int info;

  // gaussian elimination with partial pivoting
  info=0;
...
    // find l = pivot index
    l=idamax(n-k, col_k, k, 1)+k;
    ipvt[k]=l;

    col_k=calcMults(col_k, n, k, kp1, l);
    if(col_k[l]!=0)     {
      for(j=kp1; j<n; j++)
       reduceColumn(a, n, col_k, j, k, kp1, l);
    }
...
  info=calcInfo(a, n, info);
  return info;
 }
}
```

Parallelisation code (control view)

```
Separate<Replicate<Linpack>>
Broadcast<Linpack,
  'int dgefa(double[][], int, int, int[])'>
CondExe<Linpack,
  'double[] calcMults(double[] col_k, int n,
       int k, int kp1, int l)', 'a.getPart(k)'>
CondExe<Linpack,
  'int calcInfo(double[][] a, int n,
                 int info)','a.getPart(n-1)'>
CondExe<Linpack,
  'int idamax(int n,double dx[],int dx_off,
                int incx)','a.getPart(dx_off)'>
CondExe<Linpack,
  'void reduceColumn(double[][] a, int n,
               double[] col_k, int j, int k,
               int kp1, int l)','a.getPart(j)'>
```

Parallelisation code (data view)

```
Partitioned<'Linpack.a',[CYCLE][*]>
ScatterBefore<Linpack,
  'int dgefa(double[][], int, int, int[])',
  'double[][] Linpack.a'>
GatherAfter<Linpack,
  'int dgefa(double[][], int, int, int[])',
  'double[][] Linpack.a'>
```

Figure 9 – JGF LuFact parallelisation.

2.3.4 Heartbeat

Heartbeat patterns are generally applied to problems solved iteratively. This type of pattern is addressed by executing (i.e. broadcasting) the method that computes iterations on all nodes. Additional data moves (at each iteration) can be injected into the sequential code through data distribution transformations.

The JGF LuFact example of a typical heartbeat application (Figure 9), it is a Java version of the popular Linpack benchmark. In this example we extracted three blocks of code into methods in order to support conditional execution by means of *CondExe*. Before and after executing the *dgefa* method we need to scatter and gather values of matrix *a*. Note that in this case a method that returns a value is conditionally executed. That value is automatically broadcasted to all aggregate elements (e.g., by *CondExe*).

The parallelisation of the LUFact is very close to the parallelisation of the computation of All-Pairs Shortest Paths (ASP) [2]. This makes it attractive to develop a template that can be used in both cases. Figure 10 presents that template. It creates an aggregate of *ClassT*, broadcast execution of *method2BCast* to all aggregate elements, conditionally executes *method2CEx* when *condEx* is true and *field2Dist* is distributed among aggregate elements using the *partType*. This template is enough for the ASP application, but for LUFact it only implements statements in italics from Figure 9. As such, in this case, three additional *condExe* templates are also applied.

```
HearbeatBC<ClassT, method2BCast,
   method2CEx, condEx, field2Dist, partType> {

 Separate<Replicate<ClassT>>
 Broadcast<ClassT, method2BCast>
 CondExe<ClassT, method2CEx, condEx>

 Partitioned< field2Dist, partType >
 ScatterBefore<ClassT, method2BCast,field2Dist>
 GatherAfter< ClassT, method2BCast, field2Dist>
}
```

Figure 10 – Template for JGF LuFact and ASP parallelisation.

Another important point about code in Figure 9 is that a shared memory version of the LUFact can be efficiently derived by ignoring transformations that implement the data distribution.

3. PERFORMANCE EVALUATION

This section evaluates the proposed approach with two case studies from the Java Grande Forum (JGF) [15] and presents performance results. JGF includes benchmarks in sequential, concurrent (i.e., Java threads) and parallel (Java MPI) variants. This section describes developed parallel versions of Crypt and LU factorisation, which are in the Farm and Heartbeat category. Parallel versions of the other benchmarks were developed in a similar way, as they are also in these pattern categories.

The Crypt benchmark encrypts and decrypts a byte array. The processing is performed in the method *Do* of *IDEATest*. This application is parallelised by processing parts of the byte array in parallel, which was performed with the *Replicate* template to create one instance of *IDEATest* on each node and executing the method *Do* on all nodes. *Scatter* and *Gather* templates divide the byte array among workers (field *plain1*) and gather the processed results (field *plain2*). The *CondExe* template was used to ensure that some data initialisations were performed only at node 0.

The LUFact performs a LU factorisation through an iterative algorithm (e.g., Heartbeat), requiring the broadcast of a matrix column at each iteration (different for each iteration). We followed a similar approach to the Crypt benchmark, by replicating instances of class *Linpack* and executing the *dgefa* method on all nodes. This example was presented in Figure 9.

Performance benchmarks were performed by comparing execution times of parallel versions built with this approach and equivalent hand written parallel versions (MPI based taken from JGF). Table 2 presents the speed-up, relative to the sequential versions, on a cluster of 8 bi-Xeon 5130 machines (a total of 32 cores, 4 per machine) and JDK 1.5_3. The first four benchmarks are from JGF (SOR is red-black successive-over relation, another typical heartbeat and the RayTracer is a typical farm).

Table 2. Speed-up of hand written (HW) parallel applications and built using pluggable parallelisation (PP) on a cluster

Application	4 cores		16 cores		32 cores	
	HW	PP	HW	PP	HW	PP
CryptC	3.54	3.53	7.80	7.56	9.56	9.27
SeriesC	3.11	3.13	12.26	12.29	24.42	24.61
SparseMatmultC	2.11	2.12	6.85	8.28	10.53	19.46
LUFactC	2.22	2.29	2.85	3.03	2.07	2.53
SORC	1.53	1.25	2.93	1.86	2.87	2.49
MDB	3.78	3.74	9.94	10.89	-	-
RayTracerB	3.82	3.88	14.13	14.38	25.64	26.16

Overheads introduced by our approach are generally very low. These are due to code re-factorings (e.g, moving blocks of code to methods) and due to the use of AspectJ. These are generally low as code transformations are made at compile-time and most injected code can be inlined. The amount of overhead depends on method granularity: fewer operations executed at each intercepted execution point represent higher overheads. The sparse matrix multiplication performs better with pluggable parallelisation. Interestingly, in this benchmark, we simply used a standard data partition strategy that seems to provide some advantage over the one used in JGF. Both LuFact and SOR scale poorly due to communication overheads (both require certain amount of communication per iteration).

Table 3 presents the execution times on machine bi-Xeon E5430 (a total of 8 cores). In this case hand written versions are implemented with Java Threads (also provided by the JGF). It should be stressed that shared memory implementations with pluggable patterns share most of the code with the distributed memory implementations and the "sequential like" code is the same for both versions (usually only data distribution issues are not included).

In this case the performance of both versions is also very close. The speed-up of sparse matrix multiplication drops with 8 cores. We are currently investigating this issue but it is probably due to less data locality. There is also some performance difference in MolDyn. In this case the difference is due to generation of fine-grained tasks that impose higher overheads.

Table 3. Speed-up of hand written (HW) parallel applications and built using pluggable parallelisation (PP) on a SMP

Application	4 cores		8 cores	
	HW	PP	HW	PP
CryptC	3.7	4.1	7.0	7.5
SeriesC	3.4	3.7	7.9	7.9
SparseMatmultC	4.1	4.3	8.3	3.0
LUFactC	3.6	3.6	5.7	5.5
SORC	3.7	3.9	5.9	6.7
MDB	2.9	2.4	4.2	3.2
RayTracerB	3.3	3.8	7.5	7.2

4. DISCUSSION

We start this section by comparing the proposed approach against current mainstream programming languages, i.e., MPI and OpenMP (Table 4).

Table 4. Assessment of OpenMP, MPI and pluggable patterns

	OpenMP	MPI	PP
localised / modular parallelisation	no (yes)	no	yes
incremental parallelisation	yes (no)	no	yes
unpluggability	yes	no	yes
code reuse / composition of abstractions	no	no	yes
support for new abstractions	no	no	yes
support for multi-core/ cluster/grids	yes/ no/ no	no/ yes/no	yes/ yes/yes

Both OpenMP and MPI lead to tangled code (e.g., no modular parallelisation), however OpenMP seems better in this respect, as all parallelisation statements can be placed into annotations (except for more complex issues). The use of annotations makes it easy to identify parallelisation-related statements. Tangled code makes it hard to understand MPI programs, as each statement must be tracked either to domain-specific issues or to parallelisation issues. The proposed approach modularises parallelisation issues into transformations.

Incremental parallelisation means that we can start by sequential code and progressively perform the parallelisation, with minor impact on the original code. This is supported in OpenMP by inserting code annotations, although, more complex issues usually require code re-factorings. In this matter, OpenMP and our approach seem to have a similar support.

Unpluggability of parallel code is a nice property of OpenMP since the standard allows a compiler to ignore parallelisation directives. This can also be true even if the program has calls to OpenMP run-time. The standard defines stubs for implementing these run-time libraries on machines that do not support OpenMP. Unpluggability is also supported in our approach.

A weak point of MPI or OpenMP is the lack of support to include new abstractions and to compose instances of parallelisation code into reusable modules. In MPI this is due to the fact that parallelisation code is mixed with domain specific code. In OpenMP this limitation is mainly due to its annotation-based nature that confines the set of abstractions to the set provided by the language and pre-empts the addition of parallelisation specific code in a modular way.

OpenMP only supports shared memory systems and MPI only supports distributed memory systems. Pluggable parallelisation supports both types of target platforms by "plugging" different transformations for each type of target platform. Our *GridSeparate* template supports grid environments.

We performed additional measurements to assess the usability and code reuse of each approach. Table 5 presents the number of non commenting source statements (NCSS) for each benchmark measured with the JavaNCSS tool [19], version 29.50. The NCSS of parallelisation statements were manually collected, following a philosophy similar to the one implemented by the tool. OpenMP data is based on the JGF JOMP implementation, where OpenMP directives are specified as Java comments that are not considered by the NCSS tool. In that case we counted each OpenMP directive as one statement.

Table 5. NCSS of various parallelisation approaches

Application	Base code	JOMP		MPI Java		PP	
	NCSS	NCSS	Grow	NCSS	Grow	NCSS	Grow
Crypt	190	193	2%	242	27%	217	14%
LUFact	239	240	0%	328	37%	262	10%
Series	70	71	1%	115	64%	79	13%
SOR	56	72	29%	155	176%	110	96%
SparseMatmult	60	100	67%	109	82%	74	23%
MD	261	-[1]	-	283	8%	271	4%
RayTracer	240	240	0%	273	14%	259	8%

The OpenMP parallelisation (using JOMP) usually results in a small increase due to OpenMP directives. There are two exceptions: the SOR and the SparseMatMult. The increase in the former is due to the use of a different algorithm for all parallel versions (a version named red-black). The increase in the later is due to code to schedule loop iterations to threads.

The MPI version leads to the highest number of statements on every cases. This is due to the statements to specify data partitioning and coordination among MPI processes. More problematic is that these statements are tangled with the basic functionality, making hard to reuse parallelisation code.

The proposed approach always requires fewer statements than its MPI equivalent implementation. This is due to the reuse of the data partitioning strategy in the library and to the template based syntax. The lower count of SparseMatMult is due to the reuse of a default partitioning strategy, as the data partitioning, in this case, is simpler than the scheduling of loops to threads (the same can also be noticed in the MPI-based implementation, where this case is the one with less increase in statements, when compared with the JOMP implementation). The proposed approach tends to generate a higher number of statements than OpenMP, although it should be stressed that OpenMP does not support distributed memory systems (i.e., these numbers do not include the code required to specify data partition).

Applying pluggable patterns to parallelise legacy code requires that the base code should be amenable for parallelisation. For instance, the sequential JGF version of the SOR does not use the red-black variant, so the parallelism that can be introduced is quite limited (in the case a *complete re-write* was required). Experience showed that in general some code re-factorings are required. The most common is to move a block of code to a method (M2M) to expose a new execution point and/or to name a block of code (as it was performed in LuFact) or to change the place where a certain operation is performed (MMC). These execution points are required to "plug" the parallelisation code into the right places. One less frequent re-factoring is the exposition of context (e.g., the addition of a new parameter to a method (PDP), or to move a variable to an object field (M2OF). Since the parallelisation is performed "from outside to inside" the pluggable parallelisation must have access to context information. In traditional systems context information is "pushed" by calling a programming API (e.g., creating a Farm class). In the proposed approach this information must be "pulled" by the pattern. To preserve modularity, sometimes the required information must be explicitly exposed by making a re-factoring (e.g., it does not make sense to expose a local variable). Table 6 summarises the re-factorings performed on each benchmark. We classify each re-factoring as improving the program structure (G), degrading the structure (B) and neutral (N).

Table 6. Description of re-factorings required to JGF benchmarks

	Expose exec. point	Expose context
Crypt	2xM2M (G)	-
Series	2xM2M (B)	-
SOR	M2M (G), MMC (G)	M2OF (G)
LuFact	3xM2M (2G/B)	-
RayTracer	M2M (G)	2xM2OF (G/B), PDP (G)
SparceMatmult	M2M (G), MMC (N)	-
MD	M2M (G)	-

M2M – Move to Method; MMC - Move Method Call; M2OF – Move Variable to Object field, PDP – Processing Dependent of new Parameter

The current implementation uses AspectJ code templates that are pre-processed by a tool. These issues and implementation details are out of the scope of this paper (some details can be found in [18]). Since we rely on AspectJ as an implementation tool, the supported execution points is a subset of the one provided by AspectJ. The implementation of the *Partitioned* template requires direct processing of the source code, since we needed to keep track of data allocation statements, identifying the size of data allocated to each object field.

[1] The JGF does not include the JOMP implementation of MD

5. RELATED WORK

Recent work focuses on using Aspect Oriented Programming [10] (AOP), namely AspectJ [11], to separate parallelisation concerns from domain specific code [8][16][13], on the development of reusable aspects to implement well known patterns [7][4][17] and on extending AspectJ with a joinpoint model for loops [9]. AspectJ is an alterative to implement parallelisation concerns but it has three significant limitations: 1) it unnecessarily exposes AOP technology to the programmer (e.g., aspects, advices and pointcuts). 2) it lacks a suitable composition model, since aspects were designed to compose with some basic functionality and not to compose one with each other. Composability of abstractions is essential to develop complex patterns. 3) the most important is the lack of powerful constructs to implement static code transformations as it relies too much on a joinpoint model that captures dynamic events. There are no language constructs to address accesses to data arrays (e.g., there is no way to intercept accesses to specific array indexes to implement data transformations). This is essential to parallelise legacy code, where methods share data structures. Probably this is why authors of [8] have written "without completely re-writing LUFact, there is nothing more that can be done using AspectJ". Our approach overcomes these issues by relying on a template based approach that hides AOP technology, providing a model to compose these templates and providing templates to explicitly address data distributions and moves. Separating transformations of control flow from data view makes it more manageable to plug parallelisation into legacy code as in the LUFact.

Java-based skeleton approaches [5][6] are an object-oriented alternative to pluggable parallelisation. These systems provide a set of high-level patterns to implement common parallelisation strategies. Parallelisation is performed "from inside to outside" resulting in invasive and non-reversible changes to the base code. This results in a weak support for legacy code and scientific codes become dependent on a specific parallelisation strategy. As a consequence skeleton systems do not promote a so clear separation between domain-specific code and parallelisation issues. Moreover, there is no support for "sequential like" style: programmers build parallel applications by composing provided skeletons. The proposed approach follows a different philosophy: domain specialists develop their code in a traditional manner and specialists in parallel computing work on *pluggable* parallelisation issues that enable codes to take advantage of parallel systems.

6. CONCLUSION

This paper proposes an approach to develop parallel applications by plugging transformations into "sequential like" code. Code transformations are specified through templates that can be composed to implement more complex patterns. The approach promotes the separation of the control from the data view through the use of a specific set of templates for each purpose.

This approach is able to support a "sequential like" style of programming and to support parallelisation of legacy code by plugging parallelisation issues, requiring fewer changes than competitive approaches. In cases where code re-factorings are necessary, scientific code remains "sequential like" (e.g., domain specific code does not become dependent of the parallelisation, being able to run when patterns are unplugged).

Future work includes applying this technique to codes that require a larger amount of parallel code when moving from sequential to parallel (e.g., parallel sorting), to address other kinds of applications, such as pointer based structures (e.g., graphs) and to investigate how to provide contracts between domain specific code and pluggable parallelisation.

7. ACKNOWLEDGMENTS

This work was supported by AspectGrid (GRID/GRI/81880/2006) and PRIA (UTAustin/CA/0056/2008) funded by Portuguese FCT and European funds (FEDER).

8. REFERENCES

[1] Baduel, L., Baude, F., Caromel, D., Object-Oriented SPMD, IEEE CCGrid2005, Cardiff, May 2005

[2] Bornemann, M., Nieuwpoort, M., Kielmann T., MPJ/Ibis: a Flexible and Efficient Message Passing Platform for Java, EuroPVM/MPI 2005, Sorrento, Italy, September 2005.

[3] Caromel, D., Towards a Method of Object-Oriented Concurrent Programming, Communications of the ACM, 36, 9, Sept. 1993.

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

[5] Danelutto, M., Teti, P., An advanced environment supporting structured parallel programming. Java, FGCS 19 2003

[6] Fernando, J., Sobral, J., Proenca, A., JaSkel: A Java Skeleton-Based Framework for Structured Cluster and Grid Computing, CCGrid'06, Singapore, May 2006

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

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

[9] Harbulot, B., Gurd, J., A Join Point for Loops in AspectJ, AOSD'06, Bonn, Germany, March 2006.

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

[11] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G., An Overview of AspectJ. ECOOP 2001, LNCS, Budapest, Hungary, June 2001

[12] Lee, D., A Java fork/join framework, Java Grande 2000

[13] Maia, M., Maia, P., Mendonça, N., Andrade, R., An Aspect-Oriented Programming Model for Bag-of-Tasks Grid Applications, IEEE CCGrid 07, 2007

[14] Nieuwpoort, R., Kielmann, T., Bal, H., Satin: Efficient Parallel Divide-and-Conquer in Java. Euro-Par 2000

[15] Smith, A., Bull, J., Obdrzálek, J., A Parallel Java Grande Benchmark Suite, SC 2001, Denver, USA, November 2001

[16] Sobral, J., Incrementally Developing Parallel Applications with AspectJ, IEEE IPDPS'06, Rhodes, Greece, April 2006

[17] Sobral, J., Cunha, C., Monteiro, M., Aspect-Oriented Pluggable Support for Parallel Computing, VecPar'06, LNCS, Rio de Janeiro, Brasil, June 2006

[18] Sobral, J., Monteiro, M., A Domain-Specific Language for Parallel and Grid Computing, DSAL08, Belgium, April 2008

[19] http://www.kclee.de/clemens/java/javancss/