Modular and Non-Invasive Distributed Memory Parallelization

Rui Carlos Gonçalves João Luís Sobral
Departamento de Informática, Universidade do Minho
{rgoncalves,jls}@di.uminho.pt

Abstract

This paper presents an aspect-oriented library to support parallelization of Java applications for distributed memory environments, using a message-passing approach. The library was implemented using AspectJ language, and aims to provide a set of mechanisms to make easier to parallelize applications, as well as to solve well known problems of parallelization, such as lack of modularity and reusability. We compare the advantages of this method over the traditional approach, and we discuss differences to recent approaches that address the same problem. Results show benefits over other approaches, and, in most of cases, a competitive performance.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Design, Languages, Performance

1. Introduction

Over the last decades, the power delivered by computers has been increasing. However, in recent years, this increase usually means more cores, instead of higher clock rates, as in the past. This new reality moves the tasks of improve performance of applications from chip manufacturers to software developers. To get advantage of these machines, applications must be prepared to them, and this is a complex task, usually reserved for experts in parallel computing.

Current approaches to parallelize applications present several problems. The ability to deal with legacy code is essential, as scientist will be resilient to embrace the parallelization of their applications if that requires to rewrite thousands of lines of code. Moreover, in new applications, parallelism related code should not invade basic application code (we refer to this code as *domain specific code* (DSC)). Parallelization is a crosscutting concern, as base application code is mixed with parallelism, and this concern is spread among several modules (tangling and scattering in aspect-oriented terminology) [4, 6, 12]. This results in lack of modularity, and consequently it is more difficult to develop and maintain code, as this does not allow independent development, and there are less opportunities to reuse code.

A particularly hard problem is the development and reuse of data partitioning strategies for distributed memory systems. Effectively exploiting parallelism requires the partitioning of application data into chunks that can be processed in parallel, and managing dependencies among data. Such code is usually a considerable part of the total distributed memory code and it is very prone to errors. Some existing systems provide a set of fixed data partitioning strategies, but that results in invasive source code changes, and they are hard to adapt for a specific case.

Aspect-Oriented Programming (AOP) [7] has been successfully applied to improve modularity in problems like logging [8], as well as in several well known design patterns [5]. In this paper we explore AspectJ [8] to provide reusable and customizable implementations of code recurrently needed in parallel applications for distributed memory architectures. The library includes a set of classes that provides common strategies to partition regular data structures.

2. The Library's Design

The library implements features that are often needed when parallelizing an application for distributed memory. We aim to improve productivity and make parallelization process easier and feasible by non experts in parallel programming. The library was developed with the following major goals:

Modularity and Reusability. The different concerns should be implemented in different modules, and should not invade DSC. This enables code reutilization, and thus reduces development times and improves productivity, reduces bugs (by reusing code already tested) and reduces maintenance times (by localizing code at a single place).

Incrementality. When the same base algorithm can be used in sequential and parallel versions, developers should be able to start with sequential code, and incrementally add parallelization code, without requiring significant changes to previous code. We may need several versions of the application to run in different environments (sequential version, multithreaded version, distributed memory version, or even hybrid versions, that mix some of previous). Although the parallel version is usually able to run as a sequential application, parallel code adds an overhead, and thus, versions with only needed features enabled are useful. If all versions share the same base code, and are obtained adding only new features, it is easier to maintain them.

Pluggability. The parallelization should be easily removable. As we mentioned in the previous point, we may need to maintain several versions of an application. If parallelization features are provided by pluggable modules, we can generate different versions just enabling the needed modules.

Extensibility and Flexibility. The developers should be able to adapt the operators provided to fit their needs, as well as add new operators. It is almost impossible to develop a tool that fits all applications, thus developers may need to add new operators, or adapt existing ones.

Performance. The library should provide a performance similar to traditional approaches. Although our goal is not improve applications' performance, the library should provide a competitive performance.

To accomplish these goals, we developed a library of abstract aspects, with abstract pointcuts that allow us to change the behavior of an application. We use lexical pointcuts, instead of annotation-based, as the later would require changes in the DSC to add the parallelization, and they can not provide support for including parallelization specific code (*e.g.*, to support a customized data partition strategy).

It is assumed that applications will run in several processes, that each of them will have an *id*, and that process with id 0 will be the master process (and the others are slaves). The library is well-suited to *single program multiple data* (SPMD) applications, where all processes run the same program, but each process deals only with its own data.

2.1 Operations Provided

2.1.1 Data Partition

There are two basic data partition operations. *Scatter* takes an array stored on master process, divides the array according to a specific partition strategy, and sends parts of the array to other processes. *Gather* does the opposite, *i.e.*, the master process receives data from other processes, and concatenates it in a single array. We provide two variants of the pointcut, to perform the operation before or after a join point.

Applications may have different needs concerning the indexes of data in the arrays when they are distributed. In some applications the position of data is not relevant. However, there are applications where indexes of data are important, as computation depends on them. We refer to these as local and global array indexes. Therefore, in the former, when sending data to slave processes, we fill arrays continuously, starting from position 0. In the later we keep the original data indexes, possibly creating holes in the array.

We support both local and global array indexes. The former is implemented in aspect `DataDistributionPart` and the latter in aspect `DataDistributuion`. When using local array indexes, we can have to hold both the complete array and an additional array to keep only its part of the array, with a local view. `DataDistributionPart` aspect provide three additional pointcuts, to specify when to change to global/local view, or when to switch view.

2.1.2 Data Reduction

A data reduction operation takes one array of data per process, and, for each array index, applies a function that merges values of that index from all arrays into one. We provide two types of reduction operations: one that collects the result of the reduction only on master process, and other that sends the result to all processes. Again, there are two versions of each, to specify whether operation should be performed before or after the join point. These operations are provided by `Reduce` aspect.

2.1.3 Execution Restriction

In SPMD applications, all processes execute the same program, but each one with local data. When we partitioned data, we may need to keep original indexes, leading to arrays where only some positions are filled. Thus, we may need to restrict execution of a method only to indexes of data stored locally.

We support this functionality in `ExecutionControl` aspect by providing advices that restrict execution of a method to processes with specific *ids*, or to processes where some predicate is met.

When a method that returns a value is executed only at one process, the returned value may be needed at all processes. To deal with these situations, we provide variants of this operator that broadcast the value returned by one process to others.

2.2 Data Partition Strategies

When parallelizing an application using a SPMD approach, we have to divide data among processes. In most of cases we have to divide arrays (or similar data structures), and there are some recurring strategies to partition arrays.

We provide classes that implement two common partition strategies: block and cyclic. In the former, we allocate a contiguous block to each process, all blocks with a similar size (it is implemented by class `Block`). In the latter, we allocate one position to each process in a cyclic way (it is implemented by class `Cycle`). We can easily extend these classes to provide slightly different variants of provided partitions (as we did do in one of the case studies).

We provide an abstract class with an API that provides information about how an array is partitioned. The API has methods to provide the start and the end index of each block, the size of each block, as well as the sum of sizes of all blocks allocated to the same process, and the node that holds a specific block or array index. There are also some methods that allow to iterate over blocks allocated to the same process.

2.3 Implementation Challenges

The most complex challenge that we have to deal with was context information. We need to know which array and which partition strategy to use in a scatter operation, or which array and which operation to use in a reduction operation, for example.

AspectJ allows specification of context information in pointcuts, that is available to be used in the advice. However, it presents some limitations in this point. For example, in an object oriented (OO) application, it is usual to have data that is needed by a method stored in the object's fields. In AspectJ, we can specify the object as context information, but we can not specify a field of the object. As we can specify methods' parameters, we could overcome this limitation passing the array as a parameter, however, in several situations, this would force significant changes in code structure, with a negative impact in code quality. Moreover, partitions or reduction operations usually can not be derived from context information.

To solve these issues, concrete aspects, besides extending the abstract aspect and defining pointcuts to specify where we want to use an operator, also have to implement some abstract methods. This implies to use different aspects to apply an operator with different parameters.

Another challenge was the specification of points where we want to execute an operator. AspectJ provides a wide range of join points, however it was not enough for our needs. Sometimes we needed more fine grained join points, to allow to intervene in loops, for example. Therefore, currently to use this library developers may have to refactor the code. Nevertheless, these refactorings are mainly extractions of code blocks to new methods, a tasks that can be done almost automatically by IDEs.

3. Example of Use

To develop and test this library, we studied JGF benchmarks [1]. This is an well known set of benchmarks, that had already been used in previous works [2, 4, 6, 13]. It provides sequential and parallel versions (Java threads, and message passing, mpiJava [11] based) of applications, giving us a comparison base for our work.

In this section we show how the library presented was used to parallelize the JGF Series benchmark. We only mention key details of the implementation. The complete source code, as well as the parallelization of the other JGF benchmarks and the library's code, is available at `http://alfa.di.uminho.pt/~rgoncalves/aj-mpi/`.

The JGF Series application computes the firsts n Fourier coefficients of the function $f(x) = (x + 1)^x$ in the interval $[0, 2]$. Figure 1 shows the most relevant parts of the method.

```
1  class SeriesTest {
2    double[][] TestArray;
3    void buildTestData() {
4      TestArray=new double[2][array_rows];
5    }
6    void Do() {
7      //...
8      coef0();
9      //...
10     for(int i=1; i<array_rows; i++) {
11       coefs(omega, i);
12     }
13   }
14   //...
15 }
```

Figure 1. JGF Series benchmarck.

```
1  aspect DataDistributionSeries extends DataDistribution {
2    protected static double[][] array;
3    protected static Partition partition;
4    after(SeriesTest obj):
5        execution(void SeriesTest.buildTestData()) && target(obj) {
6      array=obj.TestArray;
7      partition=new Block(obj.TestArray[0].length,Main.getNParts());
8    }
9    protected pointcut gatherAfter22():execution(* SeriesTest.Do());
10   protected Partition getPartition() { return partition; }
11   protected Object getArray() { return array; }
12 }
```

Figure 2. Data distribution aspect for Series benchmark.

The method has a loop that performs independent computations on each coefficient (line 11) and stores the results in an array (`TestArray`).

To parallelize the application, we divide the array using a block partition and we only call the method `coefs` in the process that holds the corresponding array position. At the end of method `Do`, we gather all the data on the master process.

Figure 2 shows the aspect that specifies the pointcut after which the gather operation will be executed (line 9). It also shows an advice where we specify the partition and the array that stores the results (lines 4-8). This is done after the method that initializes the data, and the information is later used by the methods that provide the context information to the gather operation (`getPartition` (line 10) and `getArray` (line 11)).

Figure 3 shows the aspect that restricts the execution of methods `coefs` and `coef0` to the process that holds the values. We need two aspects because we have two different conditions. The aspect `ExecutionControlSeriesMaster` extends the aspect `ExecutionControlMaster` (an extension of `ExecutionControl` aspect, that already specifies the restriction condition as the process's id be equal to 0). The method `getId` (line 12) returns the *id* of the process that contains index *i* of the partitioned array.

4. Discussion

To evaluate our approach, we implemented all the JGF MPI benchmarks. We compare the versions developed using our approach with JGF MPI versions. Table 1 summarizes our conclusions.

```
public aspect ExecutionControlSeries {
  public static aspect ExecutionControlSeriesMaster
      extends ExecutionControlMaster {
    protected pointcut condExeVoid():
      call(private void SeriesTest.coef0();
  }
  public static aspect ExecutionControlId
      extends ExecutionControl {
    protected pointcut condExeIdVoidArg(int i):
      call(private void SeriesTest.coefs(double,int))
      && args(..,i) && within(jgf.section2.series.*);
    protected int getId(int i) {
      return DataDistributionSeries.aspectOf().
          getPartition().getNodePosition(i);
    }
  }
}
```

Figure 3. Execution restriction aspect for Series benchmark.

	JGF	Lib
Modular / Reusable	no	yes
Incremental	no	yes
Pluggable	no	yes
Extensible / Flexible	no (yes)	yes

Table 1. Comparison of traditional (JGF) with our (Lib) approach.

Regarding modularity and reusability, as we already mentioned, the traditional approach is not a good solution, because it mixes DSC with parallelization code. On the other hand, our approach was able to separate several parallelization concerns from the DSC. The data partition and the control flow (execution restriction) related code are in different aspects. Thus, it is easier to reuse the different features in different applications.

Moreover, the strategy to partition data is also specified in a separated module. This makes easier to change the partition strategy, which could be useful to test and choose the best strategy for a specific application, or, when the best strategy depends on the target platform, to provide versions using different strategies.

Concerning incrementality, our versions were developed using the sequential code provided by JGF. In some cases, we needed to refactor code, mainly to expose join points (*e.g.*, extract a block of code to a new method) or context information (*e.g.*, move a variable to an object field). Nevertheless, usually these refactorings neither deteriorate the quality of code, nor result in significant changes in the structure of the original code. A more detailed discussion on the types of refactorings needed can be found in [4]. The only exception was the SOR benchmark, that requires a different algorithm to allow efficient parallelization.

Although we only address distributed memory environments, we could use our library together with another library that uses a similar approach to provide shared memory parallelism [3]. Therefore, we could start with a sequential version, and incrementally develop a multithreaded version, a distributed memory version, and hybrid version, allowing the developer to adapt the application to the hardware.

Application	4*JGF	4*Lib	16*JGF	16*Lib
CryptC	3,54	3,61	7,80	8,01
SeriesC	3,11	3,13	12,26	12,27
SparseMatmultC	2,11	2,00	6,85	6,48
LUFactC	2,22	1,83	2,85	0,20
SORC	1,53	1,71	2,93	2,77
MDB	3,78	3,98	9,94	9,02
RayTracerB	3,82	3,87	14,13	14,33

Table 2. Speedups of JGF and our (Lib) versions.

With the traditional approach, as parallel code is mixed with DSC, we cannot unplug the parallelization. Therefore, developers have to maintain different versions of code. With our approach, versions are obtained enabling/disabling aspects.

Our library showed to be extensible and flexible enough to parallelize all JGF benchmarks. We were able to add new partition strategies, based on the existing ones, as well as provide specialized versions of the operators that increased code reuse (*e.g.*, `ExecutionControlMaster`, where we already have defined that only the master process executes the method, thus avoiding the need to define a method each time we extend this aspect). Although it was not necessary, we could have added new operators. When the operators cannot be used, when we need features specific of one application, or when the library operators do not provide enough performance, we are able to develop customized aspects, that can be used together with the library operators.

Furthermore, the aspects act mainly as *glue*, specifying when and how an operator shall be applied. The implementation of the operators is in a separated class that implements a specific API. We provide classes implementing that API, but developers can use their own classes, with different strategies to implement the operators, or using other technologies than MPI.

Performance. To evaluate performance, we compared the speedup of the versions developed using our approach and the MPI version provided by JGF. Table 2 presents the speedups relative to the sequential versions, when running the applications with 4 and 16 processes. For each application, we collected the average of four execution times. It was used a cluster of 4 bi-Xeon 5130 machines (a total of 16 cores, 4 per machine), Myrinet 10 Gbps network, with JDK 1.5_3, mpiJava 1.2.5, and MPICH 1.2.7.

We do not provide results for Monte Carlo benchmark, as in all tries to run the application (both our and JGF version), it aborted due to MPI errors. The results show that our approach was able to provide a similar performance to the traditional approach in most of the JGF benchmarks (the only exception was the LUFact benchmark, where with our approach we are not able to make a code optimization that results in the need of an additional communication operation).

5. Related Work

Several works have addressed the problems that parallel programming faces. We focus on approaches that used AOP.

Harbulot and Gurd were the firsts to explore the use AspectJ to separate concerns in parallel applications [6]. Later, other works explored AspectJ [3, 10, 12, 14], however, the first only addresses shared memory environments, and the others present flexibility limitations, as they require the code to be developed according to some predefined rules (*e.g.*, class have to implement some interface), thus they can hardly deal with legacy code and force developers to think in parallelism when they are developing sequential DSC.

Other works have used aspects in a different way: instead of using them directly, they provide a domain specific language to generate AspectJ code [4, 13]. This approach has the advantage of hiding AOP technology from developer, while eliminating composition problems of AspectJ [9]. But it also has disadvantages. The first approach presents problems with context information, which forces the data needed by operators to be part of the parameters of some method. As in OO applications we often use object fields to store data, this limitation can imply changes in source code that degrade the design of the application. Moreover, developer has to specify for each application how data is partitioned. In the second approach, these problems are addressed, as the ability to deal with data is improved. However, to extend the provided operators we need knowledge about AOP, about implementation of the language, and about tools used to process the language, which may make this task difficult. Additionally, although these languages are able to deal with a wide range of applications, in more complexes cases, they may also require knowledge of AspectJ technology to use them.

6. Conclusion

This paper presents a library that provides operators to parallelize applications in a modular way, using an AOP approach. It allows a higher reuse of code, as well as incremental development of parallel applications. It is compatible with other tools that provide multithreaded versions of applications with a similar approach, making easier to develop and maintain sequential, multithreaded, distributed memory, and hybrid versions of the same application. It shows flexibility that allows adaptation of operators provided in order to fit the needs of a wide range of applications.

It was able to parallelize all JGF benchmarks, with only minor changes in sequential code, that do not change the structure of code, while providing a competitive performance with traditional approach. It was able to deal with limitations detected in previous attempts to apply AOP to solve crosscutting in parallelization, namely regarding context information.

Nevertheless, this approach also presents some disadvantages. First, it requires knowledge of AOP and AspectJ language in particular. But probably the biggest disadvantage is the fact that the relation between parallel code and DSC is not obvious. Whereas in traditional approach we see the code where it is executed, in our approach parallelization code is not visible in DSC. An IDE like Eclipse with AspectJ plug-in may be useful to minimize this problem. Moreover, there are no visible *contracts* in sequential code to tell developers of sequential code which parts of it they must not change in order to parallelization code continues working correctly. Future work should address these points.

Acknowledgments

This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-010152 and FCOMP-01-0124-FEDER-011413.

References

[1] J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. A benchmark suite for high performance java. *Concurrency: Practice and Experience*, 12(6):81–88, 1999.

[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pages 519–538, 2005.

[3] C. Cunha, J. Sobral, and M. Monteiro. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *AOSD '06*, pages 134–145, 2006.

[4] R. Gonçalves and J. Sobral. Pluggable parallelisation. In *HPDC '09*, pages 11–20, 2009.

[5] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02*, 2002.

[6] B. Harbulot and J. R. Gurd. Using aspectj to separate concerns in parallel scientific java code. In *AOSD '04*, pages 121–131, 2004.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97*, pages 220–242, 1997.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *ECOOP '01*, pages 327–354, 2001.

[9] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A disciplined approach to aspect composition. In *PEPM '06*, pages 68–77, 2006.

[10] M. Maia, P. Maia, N. Mendonca, and R. Andrade. An aspect-oriented programming model for bag-of-tasks grid applications. In *CCGRID '07*, pages 789–794, 2007.

[11] mpiJava. `http://www.hpjava.org/mpiJava.html`.

[12] J. Sobral. Incrementally developing parallel applications with aspectj. In *IPDPS '06*, pages 95–104, 2006.

[13] J. Sobral and M. Monteiro. A domain-specific language for parallel and grid computing. In *DSAL '08*, 2008.

[14] J. Sobral, C. Cunha, and M. Monteiro. Aspect oriented pluggable support for parallel computing. In *VECPAR '06*, pages 93–106, 2007.