Separation of Concerns in Parallel Applications with Class Refinement

Matheus Almeida and João Luís Sobral

Departamento de Informática
Universidade do Minho,
Braga, Portugal

Abstract. Parallel programming is becoming increasingly important since the popularization of multi-core processors. Traditional programming techniques that take advantage of these processors lack structure in the sense that the parallelization artefacts are mixed with the base code. This leads to problems in reusing, debugging and maintaining both the base code and the parallelization code. This paper presents and compares a new approach to separate those concerns. This approach is based on the concept of Object-Oriented Programming inheritance and it is called Class Refinement. Since the concepts and abstractions are similar to those on Object-Oriented, the learning curve is much smaller than using, for instance, the Aspect Oriented (AOP) approach.
We show that the performance overhead of using Class Refinement is close to the AOP approach and minimal compared to the traditional programming style.

1 Introduction

The solution adopted to overcome the problems of the increase of frequency in processors [1] is to integrate into a single CPU a set of independent processing units (*cores*). With this approach, processor designers no longer need to raise clock frequencies to increase computational power. The trend is the continue increase of the number of cores.

The older variant of parallel computing but still very important today is related to distributed computing (eg.: Cluster) where the computation is performed across a number of nodes connected by a network. The most important benefits of this approach are the large number of nodes that can be interconnected and the fact that each node can be composed by commodity hardware making it a low cost solution.

Both multi-core and cluster computing require a different programming style from sequential programming, as programmers need to specify parallel activities within applications. Thus, the development of parallel applications requires knowledge of traditional programming and expertise in parallel execution concerns (eg.: data partition, thread/process communication and synchronization). Generally, these two concerns are mixed because the code that supports the parallel execution is *injected* into the core functionality (coded sequentially),

INForum 2010 - II Simpósio de Informática, Luís S. Barbosa, Miguel P. Correia (eds), 9-10 Setembro, 2010, pp. 67–78

resulting in *tangled code*. The lack of structure of this approach also leads to scattered code since the code to enable parallel execution is spread over different *classes/modules* of the *base/domain* code. The main drawbacks of that approach are mostly noticed in the greater effort that is necessary to understand both the parallel structure of the program and the base algorithm and in the difficulty to reuse or debug functionalities.

Previous studies [2,3] argue the separation of the core functionality from the parallelization structure which allows :

1. better maintenance and reuse of the core functionality, reducing or eliminating the problem of *code tangling* and *scattering*;
2. easier understanding of the parallel structure and better reuse of the parallel code;
3. enhancement of the parallelization structure by promoting incremental development.

Aspect Oriented Programming [4] aims to separate and modularize crosscutting concerns that are not well captured by the Object-Oriented (OO) approach. It was already used successfully to separate the parallelization structure from the base/domain code [2, 3, 5]. The experience gained led us to investigate the use of Class Refinement to achieve a similar goal. The main purpose is to ease the migration of programmers since the rules and abstractions are similar to the ones found in Object-Oriented programming (OOP).

The remainder of this document is structured as follows. Section 2 gives an overview and a comparison of the techniques studied in this paper to separate concerns in parallel applications. The next section shows the implementation of a case study and the overhead caused by the separation of concerns. The conclusion and the future work are presented in Section 4.

2 Tangled, AOP and Class Refinements

This section introduces the problems with the traditional approach for the parallelization of applications and compares two other approaches that allow separation of concerns. AOP and Class Refinement allow the separation of the parallelization into well defined modules promoting modularization [6–8]. Both approaches need that the base code exposes *entry points* where additional code can be appended. In other words, some functionalities in the base code have to be separated into methods to support the attachment of code in the new units of modularity. When those *entry points* are not available, *code refactoring* is needed.

2.1 Traditional Approach

Traditional techniques to parallelize applications are invasive. Program 1 illustrates the problem of invasive modification by showing the simplified cluster

oriented parallelization of a molecular dynamics simulation [9] that will be detailed in section 3. In black it can be seen the base code and in red (italic) the parallelization statements.

Once the domain code is populated with artefacts regarding the parallelization concerns, modularity is lost. This doesn't allow, for instance, to change the parallelization to match other target platforms (eg.: Shared Memory) or to perform incremental development to enhance the parallelization or the domain code. Both codes are glued and dependent on each other.

```
public class MD {
   Particle [] one; // Vector with all particles
   int mdsize; // Problem size (number of particles)
   int movemx; // Number of interactions

    //Declare auxiliary variables to MPI parallelization
    double [] tmp_xforce;
    double [] tmp_yforce;
    double [] tmp_zforce;
    ...
    public void runiters throws MPIException {

       for (move = 0; move < movemx; move++) { // Main loop
          for (i = 0; i < mdsize; i++) {
             one[i].domove(side); // move the particles and
          }                       // update velocities
    ...
    MPI.COMM_WORLD.Barrier();
    computeForces(MPI.COMM_WORLD.Rank(),MPI.COMM_WORLD.Size());
    MPI.COMM_WORLD.Barrier();
    for (i = 0; i < mdsize; i++) {  //Copy forces to temp vector
       tmp_xforce[i] = one[i].xforce;  // to use in MPI operation
       tmp_yforce[i] = one[i].yforce;
       tmp_zforce[i] = one[i].zforce;
     }
     //Global reduction
     MPI.Allreduce(tmp_xforce,0,tmp_xforce,0,mdsize,MPI.DOUBLE,MPI.SUM);
     MPI.Allreduce(tmp_yforce,0,tmp_yforce,0,mdsize,MPI.DOUBLE,MPI.SUM);
     MPI.Allreduce(tmp_zforce,0,tmp_zforce,0,mdsize,MPI.DOUBLE,MPI.SUM);

     //Update forces based in reducted values
     //Scale forces and calculate velocity
```

Program 1: MD cluster based parallelization.

2.2 AOP Technique

Aspect Oriented Programming [4] aims to separate and modularize crosscutting concerns that are not well captured by the Object-Oriented (OO) approach. Aspects are units of modularization that encapsulate code that otherwise would be scattered and tangled with the base code.

Crosscutting concerns can be either static or dynamic. Static crosscutting allows the redefinition of the static structure of a type hierarchy. For instance, fields of a class can be added or an arbitrary interface can be implemented. For dynamic crosscutting, AOP introduces the concepts of *join point* and *advice*. Join point is a well defined place in the base code where arbitrary behaviour can be attached and advice is the piece of code to execute when a join point is reached. A set of join points can be defined by the use of the pointcut designator that allows the use of logical operators to define an arbitrary point in the program flow.

In program 2 an Aspect example is shown. This aspect traces all calls to the method *Deposit* defined in *Bank* class that have one argument of type *int* (line 3). Before that method being called (line 5), a piece of *advice* is executed. Lines 6 and 7 correspond to the advice.

Weaving is the mechanism responsible to compile the aspects. It merges both the aspects and the base classes. This process can be done either in the compilation phase or during class loading.

In the remainder of this paper whenever we'll use the term AOP we will be referring to the most mature and complete AOP implementation, AspectJ [10].

```
1  public aspect Logging {
2      int call = 0;
3      pointcut deposit() : call (void Bank.Deposit(int));
4
5      before() : deposit() {
6         Logger.log(...);
7         call ++;
8     }
9 }
```

Program 2: AOP logging example. AspectJ syntax.

2.3 Class Refinement Technique

Object-Oriented languages allow the extension of classes by means of inheritance. The new class (subclass) inherits a subset or the complete set of the superclass state and behaviour. The subclass has the ability to override that behaviour or introduce a new one. This mechanism can be seen as a layer where additional,

more specific behaviour can be attached. Thus, the fact that the subclass needs to be instantiated does not solve entirely the problem of separation of concerns. The access of the new behaviour or state defined in the subclass has to be done explicitly by using, for instance, the name of the subclass. Clearly, this solution does not scale if we want to encapsulate others parallelization mechanisms, each in its own module (eg.: subclass) because it is required to make changes to client modules to compose them.

Batory [11] proposed that refinement "*is a functionality addition to a program, which introduces a conceptually new service, capability, or feature, and may affect multiple implementation entities*". Others definitions are more restrictive and thus specific to a particular area[1].

In this study, we define Class Refinement as the ability to extend a class by means of inheritance but instead of creating a new scope (subclass), the refined class takes the name of the original class. In terms of implementation, the original class is rewritten to include the modifications defined in the refinements. Composition order becomes important since refinements are class rewritings (note: the composition order is also important in traditional OO inheritance, although it is implicitly specified by the inheritance chain and method lookup).

```
1  public class Logging refines Bank {
2     int call = 0;
3     @Override
4     public void Deposit(int value){
5         Logger.log(...);
6         call ++;
7         super.Deposit(value);
8      }
9  }
```

Program 3: Class Refinement example.

In program 3 the same example is given but using the Class Refinement approach. The similarities with Object-Oriented inheritance are huge and besides the word *refines* in Line 1, this piece of code could belong to a Bank's subclass. The main difference is that the class *Logging* does not need to be explicitly instantiated because it will rewrite the class Bank. This means that calls to the *Deposit* method in a Bank object will trigger the execution of lines 5 and 6 when the refinement is applied to the base code.

For the rest of this paper, we'll present Class Refinements implemented with GluonJ [12] (although, we do not strictly follow GluonJ's syntax). GluonJ supports refinements by means of inheritance using Java annotations. This allows the use of a standard Java compiler (eg.: javac) to compile both the base code

[1] One example belongs to formal methods (eg.: Refinement Calculus)

and the refinements. When more than one refinement is applied, it is necessary to define an order of composition because different refinements can be applied to the same class (eg.: two refinements can override the same method). Refinements are applied in load-time by the GluonJ mechanism [13] using the order defined in a specific container called *Glue class*.

2.4 Comparison

In this section we present a comparison among the approaches previously discussed to separate concerns against the traditional (tangled) parallel programming approach. We compare each approach using a set of properties that we think are fundamental to a major acceptance by the community and to build better and modular applications.

	$ClassRefinement$	$AOP(AspectJ)$	$Traditional$
$OO-Like$	Yes	No	Yes
$Modularity$	$Good$	$Good$	$Poor$
$Context$	$VeryGood$	$VeryGood$	$Excellent$
$Composition$	$Good$	$Average$	$Poor$
$Reusability$	$Average$	$VeryGood$	$Poor$
$Usability$	$VeryGood$	$Hard$	$Excellent$
$Performance$	$VeryGood$	$VeryGood$	$Excellent$
$UnanticipatedEvolutions$	Yes	Yes	No

Table 1: Comparison of Approaches.

Being *OO-Like* is important for a major acceptance from the community. The AOP approach is the weaker in this case because programmers need to learn new concepts different from Object-Oriented Programming and it also requires an Aspect compiler (eg.: ajc). Class Refinement, on the other hand, shares the same concepts with the Object-Oriented approach and it can be compiled with a standard compiler like *javac*.

Class Refinement and AOP allow improvements in modularity since new concerns are localized in new units of modularity (Refinement and Aspect). Traditional parallel programming techniques present poor modularity due to the mentioned code tangling.

Context information is the ability to access behaviour or information defined in the base code. Since method overriding is the finest grain to change the base class, Class Refinements can access almost everything except local information in methods. For instance, method overriding doest not allow to reuse parts of the overridden method. The same situation happens with AOP but with the difference that some context information can be retrieved using reflection (eg.: *thisJoinPoint*) bringing performance penalties. On the other hand, the tangled

approach, where code can be inserted anywhere, has excellent context information access.

In terms of composition, the order in which the refinements and the aspects are applied plays an important rule and can be tricky. Nevertheless, both approaches are superior to the tangled version, as it is not possible to compose code that is not made in a modular manner. Composition using Class Refinements is better than with AOP since we explicitly specify which refinements must be applied to the base code. For instance, in GluonJ, a specific *Glue class* specifies the set and order of refinements. AOP lacks such kind of explicit composition step and clear composition rules.

Re-usability is a topic still in research in the case of Class Refinements. First implementations of reusable mechanisms using Class Refinement share the problem of explicitly defining the name of the class to refine making it an average solution. There are reusable implementations of concurrency mechanisms implemented using AOP [14] and the base code can be reused as well. In the tangled approach, the base code and the parallelization structure cannot be reused because they are intrinsically glued.

The tangled version is the easiest to use because it is the simplest approach. Class Refinement, since it is OO-like and based on inheritance, borrows most of its concepts in Object-Oriented Programming, making it easier to learn and use. AOP is the hardest because it introduces new concepts (eg.: join-point model, pointcuts) and it even changes some Object-Oriented properties (eg.: methods with body in Interfaces).

The change of the parallelization to match a new target platform can be seen as an unanticipated evolution. The tangled version is the weakest because its hard to reuse the base code. The same does not happen to the other two approaches as we have been seen.

In terms of performance, in section 3.2 we'll compare the approaches in more detail.

3 Case Study

We present the parallelization of a Molecular Dynamics (MD) algorithm that makes part of the Java Grande Forum [9] benchmark suite. Molecular Dynamics are important algorithms to simulate the interaction of microscopical particles (eg.: atoms) in a great variety of fields (eg.: Physics, Biology or Medicine).

Figure 1 shows the main steps of a generic MD algorithm. The first step is to assign particles its initial position. The algorithm then iterates until some condition is met (eg.: number of iterations). At each iteration, it is calculated the force on each particle due to the interaction with all others particles (main computational cost). The next step is to determine the new position of each particle and increment the time step.

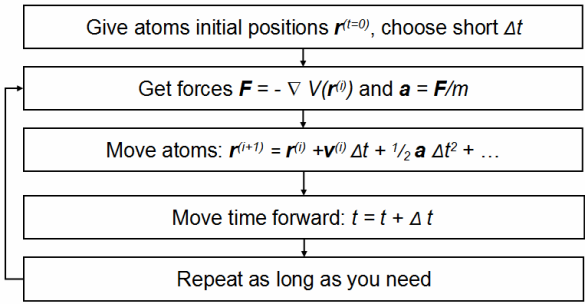


Fig. 1: Typical MD Algorithm [15]

3.1 Implementation

Figure 2 shows a simplified class diagram. The class **MD** contains all the information about the simulation including references to all particles. The method *runiters* implements the iterations of the simulation. The class **Particle** contains 9 variables representing the position, velocity and force for all coordinates in 3 dimensional space. The method force calculates the force for that specific particle with all others particles in the simulation.

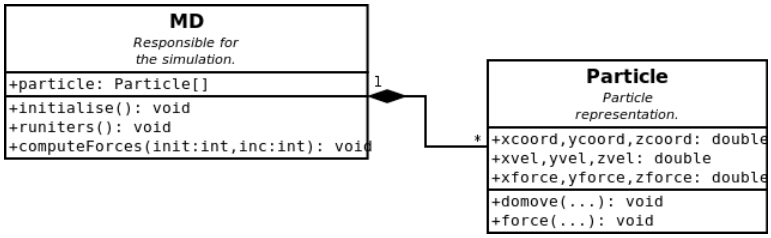


Fig. 2: Simplified MD class diagram.

To implement the shared memory parallelization using Class Refinement, the refinements listed in program 4 and 5 were created. The parallel algorithm implemented is based on the idea that the computation of the forces can be done in parallel, where each computation unit (Thread) calculates the forces for a subset of the total number of particles. When all of these computation units end, the result is merged (reduce operation).

The refinement of the MD class introduces a new data structure that will save temporary calculations before the reduce operation in the method *computeForces*.

The refinement of the class Particle is needed to use the new data structure created in the refinement *RefMD* that saves temporary computation of the forces.

```
public class RefMD refines MD {
  public static double[][] lforcex;
  ...//same structure to forcey and forcez

  @Override
  public void runiters(){
     //initialise new data structures
     //call original runiters to initialise data structures
     //of the original simulation
     super.runinters();
}

@Override
public void computeForces(...){
   //Spawn threads to compute forces in parallel
   //Join threads and reduce the values calculated in parallel

}
```

Program 4: MD refinement.

```
public class RefParticle refines Particle {

@Override
public void setForceX(...){
    //save in a new data structure created in RefMD
}

//Same to other components of the Force (y and z)
}
```

Program 5: Particle refinement.

To implement the distributed memory version, the Message Passing Interface (MPI) library was used to handle the creation, communication and synchronization of processes. Since the MPI parallelization is based in the Single Process Multiple Data (SPMD) methodology, only one refinement was needed and is presented in program 6.

The algorithm is the same as the shared memory version but instead of threads, the computational units are processes that have their own memory space and communicate through messages. The refinement *RefMPI* overrides the method *computeForces* to allow partitioning the computation. When each process ends, information is interchanged to continue the algorithm with updated values.

To take advantages of modern clusters that have hundreds or thousands of nodes where each node is composed by multi-core processors, a Hybrid version can be seen as the computation using Distributed and Shared memory parallelization. The creation of an Hybrid is just a matter of composing the refinements in the right order as shown in program 7. The first refinement being applied is *RefMD* and the last *RefMPI*. Thereafter, the behaviour defined in the refinement of the distributed memory version is the first being executed and then the behaviour in *RefMD*.

```
public class RefMPI refines MD {
 @Override
    public void computeForces(int init, int inc){
       super.computeForces(mpiRank,mpiSize);

       //Interchange information with MPI_AllReduce
       //Update state

       }
}
```

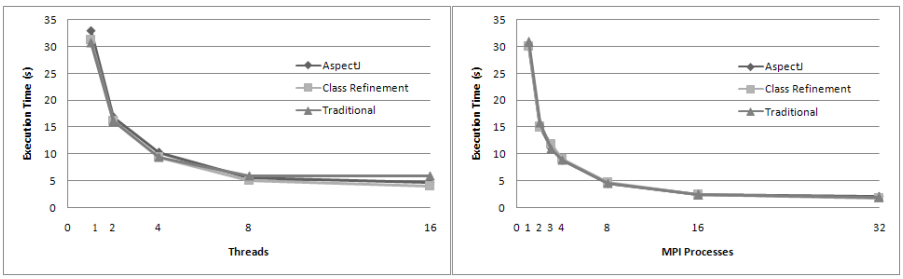
Program 6: MD refinement for MPI.

```
  applyRefinement
     RefMD,
     RefParticle,
     RefMPI
```

Program 7: Composition of Refinements.

3.2 Benchmarks

The benchmarks measure the execution time of three implementations of the algorithm explained in section 2 in both shared memory (Figure 3a) using multi-threads and distributed memory (Figure 3b) using MPI. As we expected because



(a) Shared memory (b) Distributed memory

Fig. 3: Benchmarks

the mechanism of class rewriting, the overhead caused by the Class Refinement mechanism is virtually zero and similar to the AOP approach. The difference for 8 threads can be explained by the use of concurrency mechanisms presented in *Java 1.6* that perform better for high number of threads compared to the implementation used in the JGF benchmark (*Java1.2*) (executors with thread pool). Similar results were obtained for the distributed memory parallelization. The differences in execution time are minimal in the 3 approaches.

4 Conclusion

This paper presented a new approach to solve the problem of separation of concerns in the parallelization of applications. It is based in Object-Oriented inheritance to ease the migration of programmers and to be compatible with standard compilers.

We presented a comparison among different approaches to identify the advantages and disadvantages of each methodology. There is no clear winner but the conclusions are important to understand what are the main important properties that must be presented in a system to allow a better and most complete separation of concerns. AOP and Class Refinement allow the creation of a new unit of modularity, thus allowing to deal with the inclusion of new concerns or with unanticipated changes in a modular way. Both of the approaches showed similar performance.

The main drawbacks of AOP are the need to learn new concepts and the fact that the compilation is done by a specific compiler. The Class Refinement approach is better in this case because it shares the same concepts with Object-Oriented inheritance and the compilation is done with a standard compiler.

The case study illustrated the use of Class Refinement and the benefits from its use compared to regular OO inheritance. The ability to compose the refinements and to choose what refinements must be applied in load-time is a great advantage compared to other approaches. The benchmarks showed that the overhead of using Class Refinement and AOP is little compared to traditional and invasive approaches.

Current work includes the implementation of larger case studies and optimized reusable mechanisms for parallel computing based on Class Refinement.

In the longer term, the creation of a new tool or the optimization of an existent one [12] that implements the concept of Class Refinement is an option.

5 Acknowledgements

This work was supported by the project Parallel Refinements for Irregular Applications (UTAustin/CA/0056/2008) funded by Portuguese FCT and European funds.

References

1. G. Koch, "Discovering multi-core : Extending the benefits of moore's law," *Technology@Intel Magazine*, 2005.
2. R. C. Gonçalves and a. L. Sobral, Jo "Pluggable parallelisation," in *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, (New York, NY, USA), pp. 11–20, ACM, 2009.
3. J. Sobral, "Incrementally developing parallel applications with aspectj," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10 pp.–, April 2006.

4. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97 - Object-Oriented Programming - 11th European Conference*, vol. 1241, pp. 220–242, June 1997.
5. B. Harbulot and J. R. Gurd, "Using aspectj to separate concerns in parallel scientific java code," in *AOSD 2004 Conference*, 2004.
6. D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.
7. D. L. Parnas, "Designing software for ease of extension and contraction," *Software Engineering, IEEE Transactions on*, vol. SE-5, no. 2, pp. 128–138, 1979.
8. E. W. Dijkstra, *A Discipline of Programming*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1978.
9. L. A. Smith, J. M. Bull, and J. Obdrzálek, "A parallel java grande benchmark suite," in *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, (New York, NY, USA), pp. 8–8, ACM, 2001.
10. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "Getting started with aspectj," *Commun. ACM*, vol. 44, no. 10, pp. 59–65, 2001.
11. Y. Smaragdakis and D. Batory, "Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 2, pp. 215–255, 2002.
12. M. Nishizawa and S. Chiba, "A small extension to java for class refinement," in *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, (New York, NY, USA), pp. 160–165, ACM, 2008.
13. S. Chiba and M. Nishizawa, "An easy-to-use toolkit for efficient java bytecode translators," in *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, (New York, NY, USA), pp. 364–376, Springer-Verlag New York, Inc., 2003.
14. C. A. Cunha, J. a. L. Sobral, and M. P. Monteiro, "Reusable aspect-oriented implementations of concurrency patterns and mechanisms," in *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, (New York, NY, USA), pp. 134–145, ACM, 2006.
15. K. Nordlund, "Md algorithm." "http://en.wikipedia.org/wiki/File:Mdalgorithm.PNG".