Aspect Oriented Parallel Framework for Java

Bruno Medeiros and João L. Sobral

Departamento de Informática, Universidade do Minho, Braga, Portugal
{brunom,jls}@di.uminho.pt

Abstract. This paper introduces aspect libraries, a unit of modularity in parallel programs with compositional properties. Aspects address the complexity of parallel programs by enabling the composition of (multiple) parallelism modules with a given (sequential) base program. This paper illustrates the introduction of parallelism using reusable parallel libraries coded in AspectJ. These libraries provide performance comparable to traditional parallel programming techniques and enables the composition of multiple parallelism modules (e.g., shared memory with distributed memory) with a given base program.

1 Introduction

OpenMP and MPI are arguably the most relevant instances of the shared memory (SM) and distributed memory (DM) parallel programming paradigms (PPP), respectively. With the increase of clusters of multicore machines it is common to combine MPI with OpenMP to provide a hybrid solution. However, parallelism related concerns (PRC) are known for being crosscutting concerns (CCC) [1], so it is frequent to mix them up with domain application concerns, jeopardizing the application maintenance and evolution. This mixing up of concerns is known by tangling (i.e., code that implements more than one concern) and scattering (i.e., concern that is spread out over multiple modules). Most PPP provide high-level abstractions for a specific programming model. Hence, to exploit clusters of multicores it might be necessary to combine different PPP that leads to more code tangling and scattering issues.

```
void MD (..){
...
~~forces = particles.getForces();~~
globalID = processID + threadID * numProcess;
totalWorkers = numProcess * numThreads
for(pA = globalID ~~pA=0~~;pA < sizeP; pA+=totalWorkers ~~pA++~~)
 for(pB = pA + 1; pB < sizeP; pB++)
  if(distance(pA, pB) < radius){
     forcesAB = callForcesParticles(pA,pB);
     ~~forces[pA] += forcesAB;~~
     ~~forces[pB] -= forcesAB;~~// Newton's 3rd Law
     threadForces[threadID][pA] += forcesAB;
     threadForces[threadID][pB] -= forcesAB;
  }
callThreadBarrier();
threadForceReduction();
callThreadBarrier();
if (threadID == masterThread) processForceReduction();
} ...
```

Fig. 1: MD - Hybrid parallelisation (Processes + Threads).

Figure 1 presents a hybrid parallelisation of a molecular dynamic (MD) simulation to illustrate the tangling and scattering issues. In this code forces between pairs of particles (*pA* and *pB*) within a given radius are calculated and updated. The outer loop iterations are divided by all threads of all processes (line 05) and since the forces' updates might cause data races, each thread uses a local array to save the particles' forces (lines 11 and 12). In the end threads call a local barrier and reduce their work to the master thread (line 15). Lastly, the master thread of each process will perform a force reduction among them (line 17). The black, blue and red lines of code are related to the sequential, multi-thread and multi-process concerns, respectively. The gray lines are sentences that existed in the sequential version but were excluded from the parallel version.

Figure 1 shows that adding parallelism into the sequential code made it more complex and harder to understand. If the developer wants to change the mapping between particles and threads/processes to improve the load balancing, he has to rewrite the code. Moreover, if the optimisations are duplicated elsewhere this means that any modification to its reasoning provokes code changes in different locations of the application. This exposes the inherent problem of the lack of modularity of such solutions. A better design solution is the encapsulation of the threads and processes parallelisation into independent modules.

This paper shows an approach to develop high performance hybrid parallel Java applications without polluting its source code. Initially Java programmers develop their sequential base code and further on add parallelism modules. These modules are added in a non-invasive fashion and their PRC are inserted at compile/load time. The modules are pluggable, which allows testing different types of parallelism without rewriting the base code every time. Performance portability is addressed by supporting SM and DM libraries that can work together or separately to address the specificities of each target platform.

The next section discusses how our parallel libraries address the PRC and presents their implementation that mimics the current mainstream PPP. Section 3 presents performance results. Section 4 compares this work with related work. Finally, section 5 concludes the paper.

2 Parallel Libraries with AspectJ for hybrid parallelism

To solve the PRC problem we provide SM and DM libraries in AspectJ to be used as an extra layer of modularisation, promoting a modular approach. AspectJ [3] provides modules that can be added to the base program without polluting it with CCC. With AspectJ it is possible to modify the static structure of an application as well as its execution flow in a non-invasive fashion. This language allows the capture of join points (using pointcuts), spread across a base program, to add behavior to them (using advices). This behavior is explicitly added (e.g., at compile) through code transformations performed by AspectJ's internal mechanisms, providing a solution to deal with the tangled and scattering problems. Since multiple transformations can be applied to the same base program, the language allows to specify hybrid parallelisations. However, AspectJ restricts the granularity and the type of join points that can be triggered. Therefore, some base programs might need to be adapted to expose potential join points.

Our approach identifies three main components, the base program, the aspect libraries and the concrete aspect, represented in the Figure 2 by blue, red and black colors, respectively. Each library is represented by an abstract aspect that is connected with external APIs (e.g Java Threads, OpenMPI ...) in a reusable and decoupled manner. Thus, programmers can easily interchange between different external APIs. Furthermore, since the libraries will be used with different applications, they cannot depend on join points of a specific application. Thus, these libraries are composed by abstract aspects that encapsulate behavior and state transversal to their sub-aspects and abstract pointcuts without explicitly defined join points. Later on, for each application, those abstract aspects are extended by concrete ones that encapsulate state and behavior specific to the target application. The mapping between the abstract pointcut and the join points to be intercepted is defined in the concrete aspect. The concrete aspect works as a bridge between the core of the library of aspects and the target application. Finally, from the application point of view, in some cases it is necessary to expose join points using our design rules. This kind of approach is known in requirements engineering as *scaffolding*[11] .

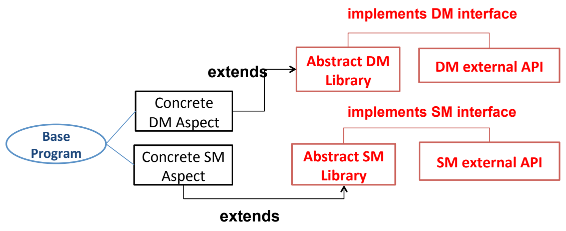

Fig. 2: Aspect Libraries: Overview.

The SM library is influenced by OpenMP and provides many of the its constructs, such as: critical, single, master, barrier, parallel for (dynamic, static ...), tasks and so on. It is possible to specify how objects behave among threads, allowing to declare them as private, shared, to be reduced and so on. The SM library uses a threads-executors pool created after intercepting the main method of the target application. Whenever a thread reaches a parallel region it requests from the initial pool a new team of threads and becomes the master this team.

The DM library will run as many instances as the number of processes requested using the SPMD execution model of MPI. This library provides constructs implemented on top of MPI calls, thus offering constructs such as: -Allreduce, gather, scatter, broadcast and many others. Moreover, offers constructs that are not provided by the MPI standard, for example: parallel for (with static round-robin, static by blocks and dynamic distributions) and distribution of 2D arrays using several strategies.

With the parallel libraries, the developer can use its own data or our data API with extra features, such as: different types of broadcast of a matrix among processes (e.g., by lines ...); accessing arrays with high-level abstractions (e.g., gets/sets ...); pre-programmed reduction functions for arrays and matrixes.

Similar to OpenMP and MPI, our libraries do not check for data dependencies, race conditions, or deadlocks. Nevertheless, the libraries guarantee the correctness of their aspects and advices, and of the user concrete aspects, as long as they follow our designing rules. As far as AspectJ is concerned, with our approach the user only specifies well defined pointcuts and/or inter-type declarations. This reduces the complexity of using the libraries and facilitates its correctness.

2.1 Design rules

In our approach domain experts develop sequential code and apply, if necessary, soft design rules[1] that enable the introduction of PRC and are a key to enable the composition of parallelism modules. The design rules are the same to every application and work with any of our parallel aspect libraries.

Our first design rule states that PRC should be encapsulated into methods. In this manner, PRC can be uniquely identified and additional behavior can be easily (un)pluged. This rule is also applied over loops to be parallelized. Performance-wise, since such methods can be declared as final, the compiler will most-likely inline its calls.

Our rule to data states that PRC objects have to implement our interfaces. Those interfaces are used as market interfaces to identify objects that our libraries should intercept and perform actions (e.g., reductions). The declaration that an object implements our interfaces is coded in the concrete aspect instead of the target object. In this manner objects are not polluted with PRC. This is possible using the inter-type declaration mechanism of AspectJ. In SM this design rule is applied to objects that require to become private to threads, whereas in DM is applied to objects used in data communication among processes.

2.2 Illustrative Example

```
void MD (..){
forceCalculation(0, sizeP, 1, ...);
}
..
void forceCalculation(int begin, int end, int step, ...){
 ...
 forces = particles.getForces();
 for (pA = begin; pA < end; pA += step)
  for(pB = pA + 1; pB < sizeP; pB++)
   if(distance(pA, pB) < radius){
     forcesAB = callForcesParticles(pA,pB);
     forces[pA] += forcesAB;
     forces[pB] -= forcesAB;
   }
} ...
```

Fig. 3: MD - Code transformation.

[1] All the designing rules presented have been formally written and detailed in order to guide the programmer with the transformations and with the aim that in the future they can be automated. However, for reasons of lack of space its formal description was omitted in this article.

Figure 3 illustrates the use of the loop design rule in the sequential version of the code of Figure 1, whereas Figure 4 presents the concrete aspects with the join points that will be intercepted to add the requested behavior by DM and SM libraries. In the concrete aspects the programmer expressed the intentions: - of statically dividing the outer loop iterations within method *forceCalculation()* among processes (line 03 of Figure 4) and among their threads (line 11) and at the end of it performing a data reduction among threads (line 12) and among processes (line 04); - that particles' forces are objects that will became private (line 09) and used during processes communication (line 01) - that the *getForces* method will return a private thread copy (line 13) and that this data will be used in processes communication (line 05) as well.

```
00: public aspect DM_Concrete extends abstract_DM_Library {
01: declare particles.forces implements DMInterface
02: ...
03: public pointcut staticParallelFor (...) : (call (void forceCalculation(...)));
04: public pointcut reduction()             : (call (void forceCalculation(...)));
05: public pointcut commData()              : (call (... getForces()));
06: }
07: ...
08: public aspect SM_Concrete extends abstract_SM_Library {
09: declare particles.forces implements SMInterface
10: ...
11: public pointcut staticParallelFor (...) : (call (void forceCalculation(...)));
12: public pointcut reduction()             : (call (void forceCalculation(...)));
13: public pointcut privateData()           : (call (... getForces()));
14:} ...
```

Fig. 4: Distributed Memory and Shared Memory concrete aspects.

The SM and DM libraries can be use separately or together, such as the case of the MD example of Figure 3. In this hybrid example, after intercepting the main method, the DM and SM libraries will create data related to the processes and their pool of threads, respectively. Since, the object particles implements the SM and DM interfaces, the SM library will create a copy of the particles' forces for each thread and the DM library will save a reference of particles' forces of the master thread. Before entering the *forceCalculation()* method the DM library will intercept its arguments and modify them in order to assign the iterations of its loop (line 08 of Figure 3) to the processes. The SM library will then further divide those iterations by the threads. When the *getForces()* method is intercepted the SM library will caught its object reference, match this reference in an internal *hashmap* and return the correspondent copy assigned to the current thread. After the *forceCalculation()* method finishes its work, the SM library will internally reduce all the forces among threads and save its result in the reference to the particles' forces object corresponding to the thread master. Finally, the DM library will perform, among processes, a global reduction of each master thread result.

3 Performance evaluation

This section evaluates the libraries performance, against Java-based implementations using traditional PPP (i.e., non-modular). The test platform is a cluster with two machines connected by a Gigabit Ethernet. Each machine has two E5-2695v2 processors, each processor with 12 cores connected to a memory bank (a NUMA with 24 physical cores per machine with 48 hyper-threading). The cluster runs Cent OS 6.3, OpenJDK 1.8.0_20 and OpenMPI 1.8.4.

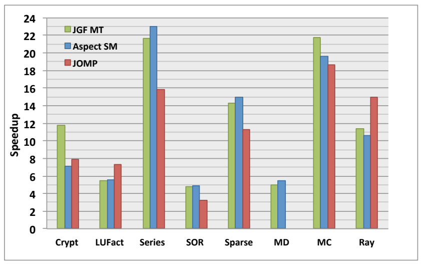

Fig. 5: JGF, JOMP vs AspectJ SM library.

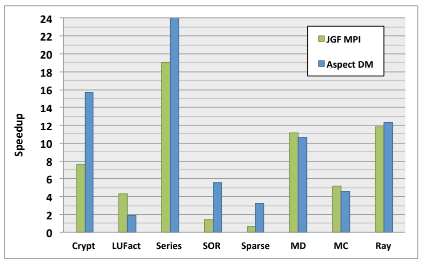

Fig. 6: JGF MPI vs AspectJ DM library.

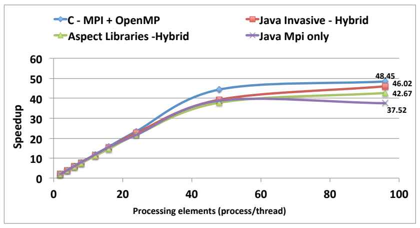

Fig. 7: MD MPI vs Hybrid version.

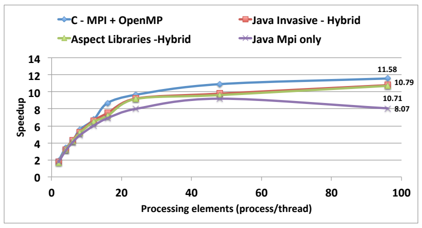

Fig. 8: MM MPI vs Hybrid version.

The first test uses JGF [2] section2 and section3 multi-threaded benchmarks (JGF_MT) as the comparison base. Performance results are the speedup relative to the JGF sequential code and also includes a JOMP[2] version. In most benchmarks the performance is comparable (Figure 5), in some cases our SM library is faster on others is slower. This behavior happens because performance is sensible to many platform details. Overall our library is 1.05x slower due to overheads introduced by aspects and the application of the design rules. The JOMP implementation is 1.2x slower and does not provide a MD implementation. The second test uses JGF MPI benchmarks (JGF_MPI) as the base of comparison. In most benchmarks the performance of our DM library is better (Figure 6) due to a faster implementation than the one provided by the JGF (1.4x faster).

The third test evaluates the impact of composing the SM and DM libraries using two machines (24 cores each with 48 hyper-threading). The Figure 7 compares the performance of the base Java MD (with half a million particles simulation) using only MPI processes against three hybrid versions using: i) Java threads and MPI (non-modular); ii) our SM and DM aspect libraries (modular); iii) using a C version of i) with MPI and OpenMP. The hybrid versions use one MPI process per machine, each composed by multiple threads (from 1 to 48 threads in total). The C version presents the best speedup for 96 processing elements (48x). Our version has a small overhead compared with the non-modular Java version, but both versions have performance close to the C. Using the full processing available the pure MPI version has the worst performance (37x), due to the overhead of inter-process communication. With traditional PPP, moving

[2] The implementation of OpenMP for Java

from this version to a hybrid version requires changes to the base program. In our approach changes are made by simply modifying the parallelism modules to be composed with the base program. The hybrid version uses a static loop scheduling among MPI process and a dynamic loop scheduling among the threads within a process. After a few tests we concluded that this strategy provides the best performance. With our approach, testing various scheduling strategies simply required a change of the pointcut in SM and DM concrete aspects. In contrast, a non-modular design requires invasive changes to the base program (e.g. Figure 1). The MD case study scales well with the number of cores. However, with the last test (Figure 8) that evaluates the performance of a parallel matrix multiplication (using 8192 x 8192 size matrices) it does not scale so well, since it requires more communication among processes. In this test the C version also presents the best performance with 11.58x speedup for 96 processing elements closely follow up by our Java aspect version (10.71x). Finally, the performance of both Java hybrid versions are also better than pure MPI versions.

4 Related work

Although annotations based approaches such as OpenMP and JOMP[3] [4] allow the division between domain concerns and PRC, it is restricted to the basic PRC. Sophisticated approaches required the use of explicit constructs, such as threads ids, object locks and so on. Furthermore, not only those annotations are still tangling with the base code, limiting its composability and modularity properties, but also deal only with SM PRC. Those problems are even worse with MPI libraries where only communication functions are provided (e.g, missing task distribution) which are explicitly added into the source code. Our SM and DM aspect libraries overcome those limitations by providing design rules to make the code parallel-awareness without breaking its sequential semantic. Further on, using concrete aspects as neutral zones where PRC can be expressed using a pointcut based style language. Providing an overall approach that allows to easily compose multiple PPP (e.g. SM and DM) without the need to learn two different programming languages syntaxes and fully decoupling the base code from the parallel code promoting a more cleaner and modularized approach.

Skeleton[4] [6] frameworks provide compositional proprieties, with Lithium [7] being a Java example of such framework. In this kind of framework, it is necessary to create classes that will represent tasks to be done and instantiate a particular skeleton to coordinate the task execution. This approach has two main limitations: 1) the base program is polluted with scaffolding code to redirect execution the skeleton framework; 2) skeletons only encapsulate simple parallelism models (e.g., farm, pipeline ..).

In [1, 9] aspect oriented programming was used to decouple PRC from domain concerns and encapsulate it in separate modules, to do so [9] used a template-

[3] A proposal OpenMP for Java.

[4] Concept proposed to encapsulate the details of a particular parallelism exploitation pattern.

based language. The work in [8] used reusable aspects to encapsulate concurrency patterns. Our work differs from these, by providing libraries (SM and DM) with competitive performance and easy to be composed (hybrids), that mimic OpenMP and MPI constructs for Java. In our approach the join point model for loops of [10] could have been used to avoid applying design rules into parallelizable loops. However, method refactoring of loops promotes independent development since the parallelisation modules depend on this explicit API. Although this approach might appears drastic at first, in reality PPP like Intel Parallel Task Library [5] follow the same strategy and some languages (e.g., R and Haskell) have higher order functions that can be seen as loop encapsulation.

5 Conclusion

The paper presented an alternative modularisation strategy where PRC are encapsulated as aspects modules. We use the potentialities of AspectJ to support composition of multiple aspect libraries with the same base program to provide efficient hybrid solutions. Performance results show that the framework provides a competitive performance comparing with handcrafted approaches. Moreover, the hybrid versions shown to be faster than versions using only DM.

Future work will focus on the introduction of mechanisms to support parallelism using aspects for other platforms (e.g., GPUs and GRID). Furthermore, it is expected the creation of name conventions and preprocessing tools to the automatization of the design rules.

References

1. Sobral, J., Incrementally Developing Parallel Applications with AspectJ, IPDPS06
2. J.Smith, J.Bull and J.Obdrzlek, A Parallel Java Grande Benchmark Suite, SC 2001
3. Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G., An Overview of AspectJ. ECOOP 2001, Springer Verlag LNCS vol. 2072, pp. 327-353
4. J.Bull and M.Kambites, JOMP an OpenMP-like interface for Java, JAVA '00 ACM
5. D. Leijen, W. Schulte, and S. Burckhardt. 2009. The design of a task parallel library SIGPLAN Not. 44, 10 (October 2009), 227-242
6. Cole, D., Algorithmic Skeletons: structured management of parallel computation, Pitman/MIT press, 1989.
7. Aldinucci, M., Danelutto, M., Teti, P., An advanced environment supporting structured parallel programming in Java, Future Gener. Comput. Syst. 19, 5 (2003)
8. Cunha, C., Sobral, J., Monteiro, M., Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms, AOSD06, Bonn, Germany, March 2006.
9. Gonçalves, R. and Sobral, J. 2009. Pluggable parallelisation. in Proceedings of the 18th ACM International Symposium on HPDC 09 Munich, Germany, 11-20
10. Harbulot B., Gurd J. A join point for loops in AspectJ. In Proceedings of the 5th international conference on AOSD '06. ACM, NY, USA, 63-74.
11. Chitchyan, R., Greenwood, P., Sampaio, A., Rashid, A., Garcia, A. F., and da Silva, L. F. Semantic vs. syntactic compositions in AO requirements engineering: an empirical study. In AOSD (2009), K. J. Sullivan, Ed., ACM, pp. 149160.