Optimizing Molecular Dynamics Simulations with Product Lines

Rui C. Silva
Departamento de Informática
Universidade do Minho
Braga, Portugal

João L. Sobral
Departamento de Informática
Universidade do Minho
Braga, Portugal
jls@di.uminho.pt

ABSTRACT

This paper presents a case study of using product-lines to address the variability of optimization methods and target platform mappings in high-performance molecular dynamics simulations. We use Feature Oriented Programming to incrementally extend the base algorithm by composing performance enhancement features with the core functionality. Developed features encapsulate common optimization methods in molecular dynamics simulations and target platform mappings. The main benefits of the approach are: 1) it promotes an incremental development, where optimizations and mappings are developed incrementally and simultaneously with the core functionality; 2) complex optimizations and mappings can be obtained by composing basic features. The performance of synthesized products is comparable to the performance of products developed with traditional parallel programming techniques. In this approach complex optimizations become easier to develop, by composing basic features, providing a performance advantage over traditional programming techniques.

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

Algorithms, Performance, Design, Languages.

Keywords

Parallel programming, molecular dynamics simulations, product-line, feature oriented programming

1. INTRODUCTION

Molecular dynamics (MD) simulation is an important tool to understand biomolecular functions. These simulations are being used to understand the origin of many diseases and to discover new drugs, among many other applications. These simulations are very computing demanding as the simulation is performed at atom to atom interaction level. Current software for molecular dynamics simulation, e.g., GROMACS [1][2] and NAMD [3][4], resorts to highly optimized methods, including the use of parallel processing.

Unfortunately, with current programming techniques to develop software, the code must be written all-at-once where statements regarding core molecular dynamics simulation are mixed with statements expressing optimizations, parallel execution and platform mappings. This has strong consequences in system development and evolution. For instance, most well known packages required an almost complete rewrite on every new generation of the software. This is due to the lack of modularity in software, as concerns like optimization, parallel execution and mapping into target platforms are mixed with the core functionality. Thus, to change the implementation of one of these concerns we need to perform rewrites over multiple parts of the system.

In this paper we describe an effort to develop a highly modular package for high performance molecular dynamics simulation. The key idea is to provide a basic molecular dynamics implementation and to encapsulate, into pluggable features that refine the basic functionality, the variability of optimizations, parallelization methods and platform mappings. We use Feature Oriented Programming [5][6] to express those features, including a feature model that captures composition rules and constrains.

The rest of this paper is organized as follows. The next section describes, in more detail, the problems of mixing concerns (a.k.a, tangling) in the code and the limitations of current mainstream programming techniques to address separation of concerns. Section 3 presents the basis of molecular dynamics simulations and section 4 presents the developed product line. Section 5 shows performance results and section 6 concludes the paper.

2. PROBLEMS OF TANGLING

We illustrate the problem of mixing concerns by presenting a parallel implementation of a molecular dynamic package, used for benchmarking purposes, that is the base of our study. This benchmark is written in Java and is part of the Java Grande Forum [7].

Figure 1 presents part of the JGF benchmark implementation for distributed memory systems (e.g., a cluster of machines). In this code the core functionality (in black) is mixed with statements implementing the parallelism models and its mapping into a target platform (in italic/red). In this case the parallelization is implemented in MPI [8], a high performance middleware that targets distributed memory systems.

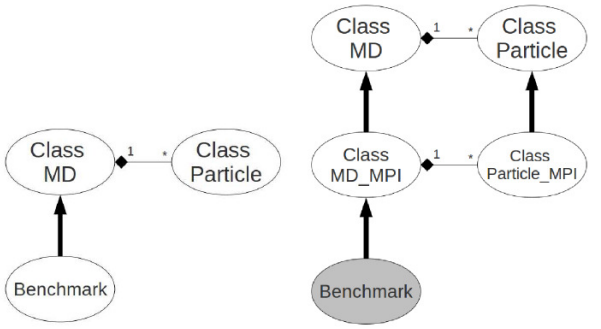
VaMoS '11, 27-JAN-2010, Namur, Belgium

```
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 reduced values
  // Scale forces and calculate velocity
  ...
  }
```

Figure 1 - JGF MD distributed memory parallelization

The problem with this code is that it is written all-at-once: the core molecular dynamics functionality is mixed with statements for parallelization and platform mapping. This has strong implications on system development and evolution. For instance, to develop an implementation for a shared memory system (e.g., a multi-core system) the code must be rewritten, whereas, in a good design, only the red/italic part should be changed. Actually, the JGF benchmark provides three completely disjoint implementations of the MD benchmark: a sequential, a thread-based and a MPI-based. Unfortunately this is the way that most parallel applications are written today. One of most popular molecular dynamics packages, GROMACS, aims to provide high performance on both a single (multi-core) machine and on clusters of machines. Since optimizations applied for each target platform are different, the source code of this package is also well known for having a large number of #ifdef statements to derive the best performing implementation.

Globally, mixing parallelization concerns with the core functionality strongly limits modular and independent development. There is no reuse of the core functionality across target platforms which also limit the evolution of core functionality to cope with new domain requirements. This



(a) Base class representation. (b) Extended class representation.

Figure 2 - Extension of MD through inheritance to cope with parallelization

approach also prevents the use of different parallelization models according to the particular simulation at hand. This becomes unmanageable since implementations mixing together all the concerns do not properly address the variability of execution platforms, parallelization models and domain-specific requirements. This becomes more problematic for long term maintenance due to the overall lack of modularity.

Traditional object-oriented encapsulation mechanisms are not powerful enough to ameliorate the problem. One solution could be to encapsulate each concern through subclassing. In the example of Figure 2 (taken from the JGF MD benchmark) we could introduce the MPI–specific code by extending the core MD class. Generally, each increment (a.k.a. implementation of a concern) extends parent class(es) to define new state and behavior. Thus, program increments are statically defined in a class hierarchy, i.e., the information about parent class(es) is hard coded in the subclass definition. Access to the new behavior defined in the subclass has to be done explicitly by using the subclass. Thus, this strategy does not scale if we intend to encapsulate multiple concerns using subclassing.

As one example of this problem, Figure 2 shows the use of OO inheritance to separate the parallelization concern shown in Figure 1. Figure 2(a) represents the base class hierarchy where class MD contains a set of references to Particles (atoms). The class Benchmark is responsible to instantiate the class MD, to configure the simulation and to call the method that starts the simulation. The use of traditional OO inheritance to encompass parallel execution is shown in Figure 2(b). The creation of two subclasses (class MD_MPI and class Particle_MPI) is needed to introduce the parallelization concerns reusing the core functionality. In order to use the MPI feature, the class Benchmark needs to be changed to instantiate the class MD_MPI. Moreover, Particle instantiations in the original MD class should be changed to instantiate the new Particle_MPI class. Clearly, this solution does not scale if we want to encapsulate several concerns, e.g., match of different target platforms, such as a cluster of machines or a single multi-core system, each in its own module (e.g., subclass) because it is required to make changes to all client modules to compose them.

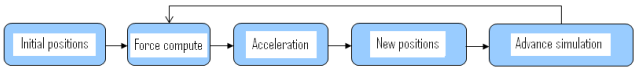


Figure 3 – Steps of a MD simulation algorithm

3. MOLECULAR DYNAMICS SIMULATIONS

Molecular dynamics is a technique of computer simulation where a set of particles (atoms) interact during a certain period of time. The particle movement during the simulation time frame is an approximation of what would happen in the real would. This technique is very useful to understand the behavior of molecules (e.g., proteins, DNA) and to infer their properties.

Simulation of molecular dynamics is performed by computing interactions among particles in the domain in a series of time-steps. For each pair of particles there is an interaction (force) that can be ignored for a relatively long distance, designated by cut-off radius. The force acting on each particle is given by the sum of all interactions with that particle. Figure **3** shows the simulation phases. Initially, positions and velocities are assigned to the particles in the domain. Then, the forces acting on each particle are computed and the resulting particle acceleration. The new position for each particle is then computed based on the particle velocity and on the time step. The process is repeated for a given number simulation steps.

3.1 Optimizations for Single Machine Execution

The most time consuming part of the simulation process is the force computation. Thus, optimization methods focus on this particular simulation phase. Two main optimizations are performed to reduce the amount of computations:

1) divide the domain into cells;

2) use a particle neighbor list.

Both techniques take advantage of the fact that interactions between particles that are beside the cut-off radius can be ignored.

The cell optimization (Figure 4) divides the space domain into cells and assigns particles to cells according to their positions. If the cell size is equal or greater than the cut-off radius, only interactions inside and with neighbor cells need to be computed. Dividing the domain into cells also increases locality of memory accesses on modern computing systems. The idea is to use cells that are small enough to fit in the cache, such that all interactions between particles in a pair of cells can be made by reusing their values in the cache, before advancing to the next pair of cells. The cell optimization also introduces some overhead due bookkeeping necessary to deal with particle moves across cells.

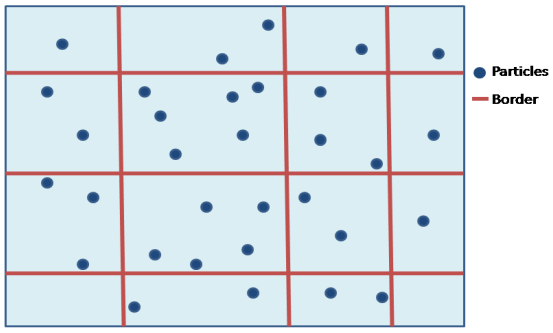


Figure 4 – Cell optimization

The neighbor list optimization (Figure 5) maintains a list of neighbors within the cut-off radius of each particle. For each particle, interactions are computed only with particles on its neighbor list. The list must be updated at the end of each interaction due to particle movement (e.g., a particle may get inside the radius of other particle). To avoid a prohibitive cost of updating the list on every time step, particles on the list can belong to an extra radius. Setting the optimum extra radius implies a trade-off between the cost of updating the list and the additional overhead of computing particle interactions in a wider radius.

The neighbor list optimization is more effective to reduce the amount of computations, when compared with the cell division, as the neighbor is considered at particle level instead of cell level. On the other hand, cell optimization has less bookkeeping and is more "cache friendly". Thus, in certain simulations and/or target platforms the cell optimization can outperform neighbor list optimization.

3.2 Parallel Execution Models

There are three alternative methods to execute molecular dynamics in parallel: 1) divide the particles; 2) divide the force computation and 3) divide the simulation domain into cells.

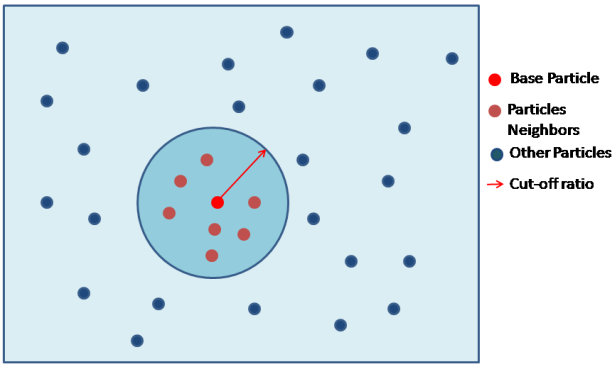


Figure 5 – Neighbor optimization

A particle division assigns a set of particles to each processing element (PE). Each PE is responsible to compute all interactions of particles on his set. This requires that each PE should know the position of all particles in the system. This division can suffer from load imbalance (i.e., PEs do not have equal amounts of work) since the number of interactions for each particle depends on the particles on its neighbor.

The force decomposition divides the force computation among PEs. It avoids the load imbalance of the particle division but also requires that each PE should know the position of all particles in the system.

The cell division is similar to the method used in sequential execution. In this case, each PE is responsible for a cell. This approach can also suffer from load imbalance since cells can have different numbers of particles but it has the advantage of requiring only information on neighbor cells to perform computations.

4. MOLECULAR DYNAMICS PRODUCT-LINE

We engaged the implementation of a molecular dynamics framework using a product line methodology. We used Feature Oriented Programming [5] for that purpose, where the core molecular dynamics algorithm can be extended by a set of optimization/parallelization features. We selected the JGF MD benchmark as our base algorithm implementation since it provides a basic algorithm (non-optimized) for simulation and can also be used for a performance assessment.

4.1 Features and Compositions

Figure 6 shows the developed features, as well as some of supported compositions. Table 1 describes the basic features and Table 2 describes the result of some possible compositions.

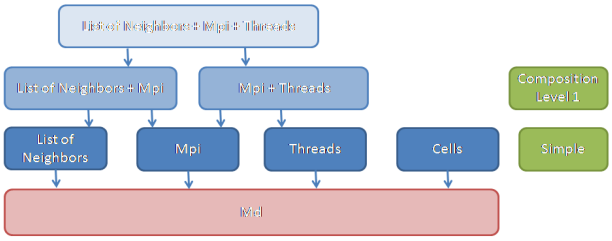


Figure 6 – Basic features and compositions

Table 1 – Description of basic features

Feature Name	Description
Cell	Domain partition into cells
Neighbor	Neighbor list
MPI	Particle partition among MPI process
Threads	Particle partition among Threads

Table 2 – Description of composite features

Composition	Description
Thread + Neighbor	Thread based parallelization with neighbor list
MPI + Neighbor	MPI based parallelization with neighbor list
MPI + Threads	Hybrid parallelization using MPI for inter-machine communication and Threads within a machine
MPI + Threads + Neighbor	MPI + Threads with neighbor optimization

Globally we implemented 4 basic features (Table 1). The Cell and Neighbor implement the corresponding optimizations for single machine (section 3.1). The MPI feature implements the mapping for distributed memory systems (i.e., mapping of execution into clusters of machines using the MPI middleware). The Threads feature implements the mapping to shared memory systems (i.e., multi-core machines, using Java concurrency mechanisms).

Currently we can synthesize 11 different variants of the basic algorithm by composing basic features (and this number is still growing), but only 8 of these are valid. Some of composition constrains are presented in the feature diagram of Figure 7.

The original MD benchmark provides three disjoint implementations: sequential, Thread based and MPI based. We can generate all three versions by composing the base algorithm with the MPI and Threads features. Additionally we support Cell and Neighbor optimization methods and more complex parallelization methods through the composition of basic features. Figure 8 presents the atomically generated application to configure a product of the MD product line. We generated this application using the GUIDSL from the AHEAD tools suite [11].

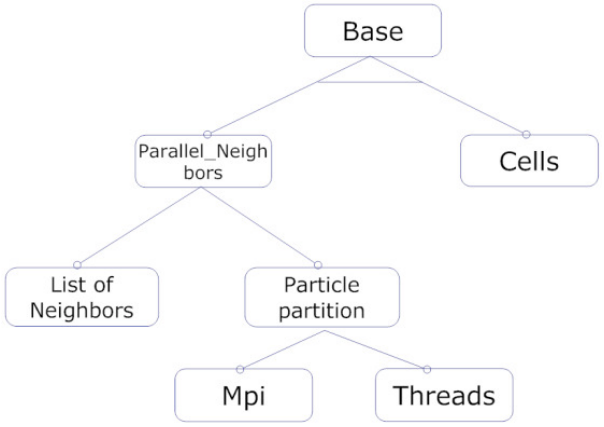


Figure 7 – Feature diagram of the MD product-line

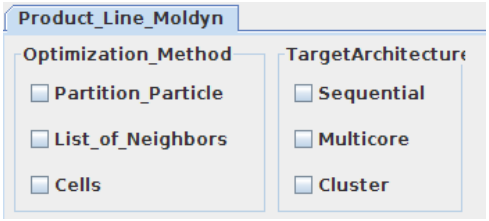


Figure 8 – Application for product selection

4.2 Implementation Overview

We implemented the product line with AspectJ [10], by developing one aspect to implement each feature. Each aspect refines classes MD and Particle to provide the additional feature-specific functionality. New class fields and methods are introduced with AspectJ construct supporting static crosscutting. Changes to the method execution behavior are implemented with call type pointcuts/advices. Figure 9 illustrates the implementation of the MPI feature that provides the support for execution on distributed memory systems (equivalent to code in Figure 1). This aspect introduces new variables into class MD and attaches the required MPI-related code before and after calls the method *computeForces*.

AspectJ provided an effective way to implement individual features, although we needed a tool to manage composition issues (e.g., specification of compatible features, see Figure 7 and Figure 8). Thus, we used the GUIDSL tool from AHEAD. A concrete product line member is specified with AspectJ load-time mechanism, where the list of features (aspects) to apply is specified at application command line. We could also implement features with other approaches like pluggable parallelization [12][13], AHEAD [11] or FeatureHouse [14].

```
public aspect MPI_MD {
   // introduce auxiliary variables to MPI parallelization
  double [] MD.tmp_xforce;
  double [] MD.tmp_yforce;
  double [] MD.tmp_zforce;
  ...
  void around(/*... */) : call(void MD.computeForces(..)) /* ... */ {
        MPI.COMM_WORLD.Barrier();
        proceed(/* ...*/);                  // execute original method call
        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);
       ...
    }
    ...
}
```

Figure 9 – AspectJ implementation of the MPI feature

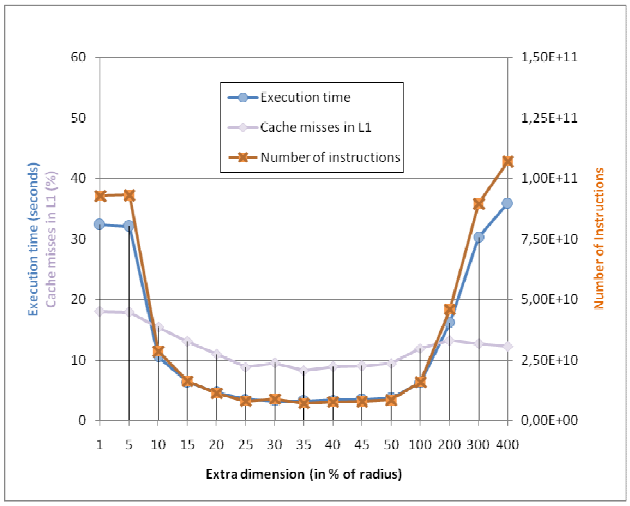


Figure 10 – Impact of neighbor optimization

5. EVALUATION

One key benefit of using a methodology based on product-lines and FOP is to provide an incremental way of developing a complex application. In our case study we were particularly interested in the impact on performance of using the neighbor and cell optimizations, since the original MD code did not provide these optimizations. In our approach these two optimizations are provided as additional pluggable features becoming easy to synthesize a product member with just one of these optimizations. Figure 10 and Figure 11 show a study of the impact on execution time of these optimizations (features).

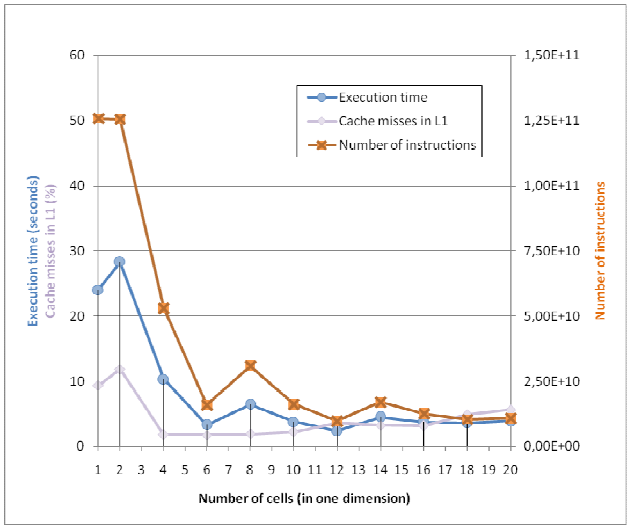


Figure 11 – Impact of cell optimization

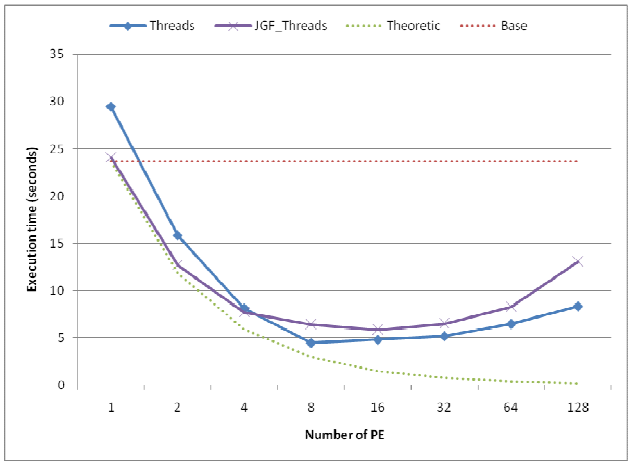


Figure 12 – Thread-based parallelization

Overall the approach allowed us to perform an in-depth study of the impact of each of these features. For instance, the cell feature greatly reduces the cache L1 miss-rate when more than two cells fit in the cache. Both neighbor and cell optimizations strategies reduce the execution time by reducing the number of executed instructions. Note that with traditional programming to perform these kinds of tests we need implement the basic MD algorithm with both optimizations.

The second test compares the performance of our implementations with the ones provided in the JGF. In this case we synthesize products base+Thread and base+MPI that are functionally equivalent to the provided JGF implementations. The Threads implementation (Figure 12) presents a slight performance advantage when compared with the JGF implementation. This advantage is due to the use of more recent concurrency mechanisms provided in Java 5 (e.g., threads pools implemented by executors). The MPI implementations (Figure 13) have identical performance.

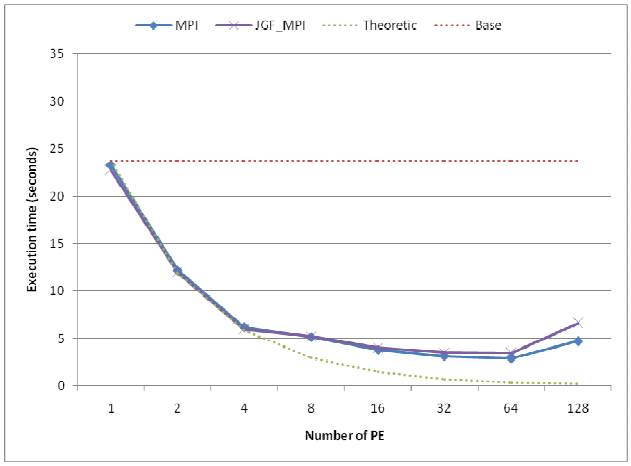


Figure 13 – MPI-based parallelization

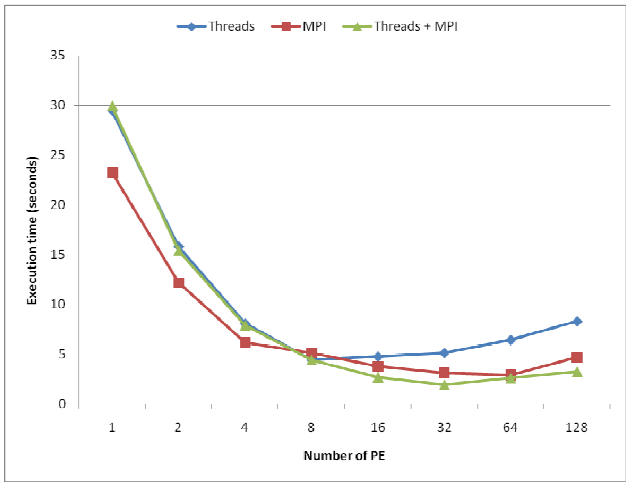


Figure 14 – Hybrid parallelization

One interesting aspect of this study is the synthesis of products with a complexity not originally provided by the JGF benchmark. One of such compositions results from the composition of Thread and MPI (called Hybrid Thread/MPI). Figure 14 shows that, when using more than 8 processing elements, the composition of Thread and MPI features provides better performance than using a one of these features. This advantage is due to hybrid parallelization methods that use MPI for communication inter-machines and Thread parallelization within each machine. It is also interesting to note that there is no single composition that provides the best performance in all cases. Thus, with our approach we can synthesize the best product for each case.

6. CONCLUSION

In this paper we described the implementation of molecular dynamics simulations using a methodology based on product lines. We rely on feature oriented programming to encapsulate, into features, the variability of optimization methods and platform mappings.

In this case study we could incrementally develop features expressing several optimizations. Encapsulating each optimization into a feature also enables an exhaustive study of the impact of each feature (or compositions of features) in the execution time. Performance of synthesized products is similar to hand written implementations, but a performance gain can be provided by developing products by composing multiple optimizations. This becomes even more useful when there is no single composition that provides the best performance in all running conditions.

Currently we are experimenting implementation alternatives to AspectJ. We are also improving the product line with more features, assessing the performance improvements of other optimizations and compositions of features. In the long term we intend to also address other kinds of execution platforms such as GPU and apply the methodology to a widely used molecular dynamics package.

7. ACKNOWLEDGMENTS

This work was supported by AspectGrid, PRIA and GAsPar projects, funded by Portuguese FCT and European funds (FEDER).

8. REFERENCES

[1] http://www.gromacs.org/

[2] Hess, B., Kutzner, C., van der Spoel, D., and Lindahl, E. 2008. Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *J. Chem. Theory Comput.* 4, 3 ((March 2008) 435-447. DOI= http://dx.doi.org/10.1021/ct700301q.

[3] http://www.ks.uiuc.edu/Research/namd/

[4] Kalé, L., Skeel, R., Bh, M., Brunner, R., Gursoy, A., Krawetz, N., Phillips, J., Shinozaki, A, Varadarajan, K., and Schulten, K. 1999. Namd2: Greater scalability for parallel molecular dynamics, Journal of Computational Physics, 151, 1 (May 1999), 283-312. DOI= http://dx.doi.org/10.1006/jcph.1999.6201.

[5] Prehofer, C. 1997. Feature-oriented programming: A fresh look at objects. *In Proceedings of the 11th European Conference on Object Oriented Programming,* (Finland, June 09 - 13, 1997). ECOOP '97. 419-443. DOI= http://dx.doi.org/10.1007/BFb0053389

[6] Batory, D., Cardone, R., and Smaragdakis, Y. 2000. Object-Oriented Frameworks and Product-Lines. *in Proceedings of the First International Conference in Software Product-Line* (Denver, Colorado, USA, August 28-31, 2000). SPLC '00. 227-248.

[7] Smith, J., Bull, J., and Obdrzálek, J. 2001, A Parallel Java Grande Benchmark Suite, in Proceedings of the 2001 ACM/IEEE conference on Supercomputing. (Denver, Nov. 2001), SC '01. DOI= http://dx.doi.org/10.1145/582034.582042

[8] Message Passing Interface Forum. http://www.mpi-forum.org/

[9] Batory, D. Feature Models, Grammars, and Propositional Formulas, *in Proceedings of the 9th International Conference in Software Product-Line* (France, September 26-29, 2005) SPLC '05. 07-20.

[10] G. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. 2001. An Overview of AspectJ. *In Proceedings of the 15th European Conference on Object Oriented Programming* (Budapest, Hungary, June 18-22, 2001) ECOOP '01. 327-353. DOI= http://dx.doi.org/10.1007/3-540-45337-7_18

[11] http://www.cs.utexas.edu/~schwartz/ATS.html

[12] Sobral, J. and Monteiro, M. 2008. A domain-specific language for parallel and grid computing. *in Proceedings of the 2008 AOSD Workshop on Domain-Specific Aspect Languages* (Brussels, Belgium, March 31 - April 04, 2008). DSAL '08. DOI= http://dx.doi.org/10.1145/1404927.1404929

[13] Gonçalves, G. and Sobral, J. 2009. Pluggable parallelization. *in Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing* (Munich, Germany, June 11 - 13, 2009). HPDC '09. 11-20. DOI= http://dx.doi.org/10.1145/1551609.1551614

[14] http://www.fosd.de/fh