A Graphical Framework for High Performance Computing Using An MDE Approach

Julien Taillard, Frédéric Guyomarc ', Jean-Luc Dekeyser

A Graphical Framework For High Performance Computing Using An MDE Approach

Julien Taillard
LIFL and INRIA-Futurs
University of Lille
France
Julien.Taillard@lifl.fr

Frédéric Guyomarc'h
IRISA
University of Rennes
France
Frederic.Guyomarch@irisa.fr

Jean-Luc Dekeyser
LIFL and INRIA-Futurs
University of Lille
France
Jean-Luc.Dekeyser@lifl.fr

Abstract

In this paper, we present a framework for Shared Memory Architectures that make design of parallel applications easier. We use the Model-Driven Engineering (MDE) approach and integrate new metamodels in Gaspard for each step of the design flow. The targeted model is an OpenMP metamodel, from which we immediately derive a source code in OpenMP Fortran or OpenMP C. This approach based on models allows a better reuse and also gives a better and more hierarchic view of the application so that it can better fit the architecture.

1 Introduction

Thanks to advancements in technology, the number of cores in processors have increased in recent years. This has caused shared memory computers to become common and has made parallel programming more attractive for non specialists.

Parallel language evolves continuously but it does not facilitate programming, code reuse and maintainability. The use of visual modeling, like the Unified Modeling Language [1], which is a standard, can help users to design parallel applications. High abstraction level modeling allows to have a model independent of any language: from this high abstraction level, different languages can be then targeted. Moreover visual modeling will imply a raise in the abstraction level at which the application is designed. It could help to increase productivity while implementation details will be managed by the code generation not by the designers. Another point to help to raise productivity is that component-based approach promotes component reuse. Once a component has been designed for a service or a computation, it could be reused in any other application with the same need.

An important point to obtain a high performance application is the distribution of the tasks on the execution platform. Since distribution could have a great influence on the application performance, a mechanism to express the distribution has been introduced in the parallel language. For example, a `DISTRIBUTE` directive is used in High Performance Fortran [2] to distribute the data over processors. This directive allocates each array element to an *owner*. Then the *owner-compute rule* is used: the *owner* has to execute each program block which modifies the owned element. Such a mechanism is needed in a high level model to express tasks distribution.

In this paper, we present a Model-Driven Engineering approach for High Performance Computing in a framework called Gaspard. This approach allows to model application and hardware architecture. Then, with the help of a distribution mechanism, the placement of the tasks onto the processor is done. Afterwards code generation is made: from a high level model, and through model transformation, different languages are targeted. We will focus on OpenMP Fortran.

This paper is organized as follow: section 2 presents how the Model Driven Engineering could be use in the High Performance Computing field, then Gaspard is presented and how to make parallel code generation is explained. Section 4 presents a comparison between generated code and hand-written code. Finally, section 5 concludes and gives some further works.

2 Usage of MDE for High Performance Computing

Although the model approach and High Performance Computing are two different fields, the latter can take advantage of modeling to make design, reuse and parallel programming easier. In this section, the Model Driven Engineering (MDE) and how it could be used in the High Performance Computing (HPC) field is presented.

2.1 Model Driven Engineering (MDE) overview

The Model Driven Engineering [3] (MDE) approach requires the use of models in each level of conception. Starting from a high abstraction level, models are refined to a targeted model. Typically, the high level models contain domain specific concepts while lowest levels contain technical aspects. The MDE is based on two concepts: model and transformation.

A model is an abstract view of the modeled system according to a certain point of view. Models information are structured according to the metamodel to which they conform. A metamodel defines the available concepts. For instance, in a metamodel of object-oriented language such as Java, concepts like *Class*, *Interface* and *Method* should be available.

The other important concept is the transformation between models. In an MDE process, all the inputs and the outputs are models. Transformation consists of describing how concepts in the input(s) metamodel(s) are converted into the concepts in the output(s) metamodel(s). It is described in *transformation rules*. Then a tool called a *transformation engine* executes these rules in order to create the output model(s).

An Object Management Group (OMG) standard called Query/View/Transformation [4] (QVT) defines a declarative and an imperative language to write the transformations. Unfortunately transformation tools are not yet mature and no complete implementation of QVT is available. Some others transformations languages are available like ATL [5] or Kermeta [6].

One of the incarnations of MDE is the Model Driven Architecture [7] (MDA) which is based on the OMG's standard. The separation between high level models and technical aspects makes the change of technology and the reuse of high level models quite simple.

2.2 Using the MDE for HPC

The use of the MDE approach for HPC should allow to raise the abstraction level and to permit a higher productivity while it should ease programming and encourage model reuse.

One of the main goals in today's language design is to raise productivity. New languages such as Chapel [8] or Fortress [9] are developed in this intention. These languages want to permit more abstraction to be used in the program specification and to improve program performance. This leads to the specification of all the application's potential concurrency. Thus compilers have to deal with this potential parallelism. Considering that a specification is a kind of model even if it is not graphical but textual one. An MDE approach could also reach this goal.

Another interesting point to raise productivity is to reuse the same specification of an application to target different computers. An application model made without any reference to the computing resource onto it will run could be reuse for any computers.

2.2.1 Visual programming

Visual programming should help to reach a high abstraction level while implementation details will not be modeled. The use of the de facto standard Unified Modeling Language (UML) will simplify programming. Manipulation of visual elements is easier than coding. Communication between developers should be simplified since they can access the global model and design each part independently of each others.

The use of a component based approach could also allow a better use of models.

2.2.2 Component based approach

Although Component Based Software Engineering (CBSE) is a widely studied domain, there is not a singular definition of a component. Each component model like CCM, EJB, or UML 2.0 has its own definition. A component can generally be defined [10] by a name, an interface and code. The code implements the services provided, or operations performed. The interface specifies what the component needs as inputs and its produced outputs. In the HPC field, inputs are data used by the computation and outputs are data produced by the component computation.

Modeling of a system in component based approach is comparable to making a component assembly.

The advantages of a component based approach is the reuse of components in different models as much as the ease to replace a component by another.

2.2.3 Related work

A use of model approach for the HPC has already been studied. Pllana and Fahringer [11] have made a model approach for High Performance Computing based on UML activity diagram. It allows to model classical constructs of parallel language like `SEND`, `RECEIVE`, `PARALLEL`. This approach is close to programming language because it includes technical aspects.

Our goal is to model applications without any technical restriction or any language reference.

3 Gaspard: Implementation of an MDE approach for HPC

Gaspard (Graphical Array Specification for Parallel and Distributed Computing) is an Integrated Development En-

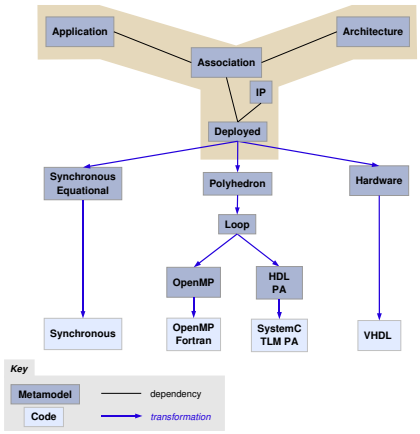

Figure 1. The Gaspard Y chart

vironment (IDE) for Multiprocessor System On Chip co-modeling and High Performance Computing. It is based on the Model Driven Engineering, more precisely on the Y-Chart (Figure 1).

Gaspard uses a component based approach using UML 2.0 [1] components. It proposes a UML profile, allowing designers to model both applications and their architecture.

The methodology is the following: application models and hardware architecture models are designed independently. Then the mapping of application on hardware is done by the user. It implies the placement of tasks on processing units and the data on memory. Finally, through some abstraction levels, specific code is generated for the targeted system. This separation between domain specific concept and the technical aspect simplifies model reuse. The appearance of a new language will introduce a new targeted language but the high level model will be reused.

Gaspard has different targets. The *Synchronous* target allows to make model verification in synchronous language such as Lustre. The *SystemC* target is able to make System On Chip co-simulation. The *VHDL* one permits to generate an hardware accelerator for a part of the application. This hardware accelerator can be implemented on Field-Programmable Gate Array (FPGA).

This papers deals with Shared Memory architectures. Therefore we produce OpenMP code [12], a standard for this targeted architecture. We use the *Single Program Multiple Data* (SPMD) approach: each processor has the same code, parametrized with the processor number. As shared memory architecture are targeted, data placement are not taken into account. Only task distributions over processors are managed.

The expression of parallelism in Gaspard is based on Array-OL, detailed in the following section. Then, the high level model is explained. Finally, we present how to generate automatic code from such models.

3.1 Array-OL

Array Oriented Language [13] (Array-OL) is a specification language allowing to express all the parallelism of a multidimensional application, including the data parallelism, in order to allow an efficient distributed scheduling of this application on a parallel architecture. It is a *data dependence expression* language which allows to express the true data dependencies. Thus any schedule respecting these dependencies will be a valid schedule.

Array-OL allows the expression of task parallelism and data parallelism.

Task-parallelism The task-parallelism is expressed by a directed acyclic graph (DAG) where each node is a task and each edge represents a multidimensional array. These multidimensional arrays may have one infinite dimension that is generally used to represent time. At the execution of a task, the input arrays are consumed while the output arrays are produced. Using this directed acyclic graph, the execution of the different tasks can be scheduled. In Figure 2, tasks A and B could be done in parallel before the execution of C.

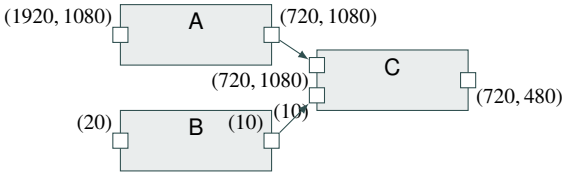

Figure 2. Task-parallelism

Data-parallelism A data-parallel repetition of a task is specified in a repetition task with a repetition space. All the repetitions of the repeated task are independent. Thus repetitions can be executed in parallel.

A repeated task works on *patterns* which are sub-arrays of the inputs and outputs of the repetition. A pattern is called a *tile* when it is considered as a set of point in an array. The considered tiles are sets of regularly spaced point of an array and the tiles themselves are regularly spaced in the array. The description of the regular spacing of the points of a tile is called *fitting* and the description of the regular spacing of the tiles in the array is called *paving*. The complete description of the tiling of an array by tiles, called a *tiler*, necessitates the description of the shape of the pattern, the fitting, the paving,

an origin and a *repetition space*. The repetition space gives the number of tiles. Figure 3 shows a tile which correspond to a (2,3) pattern positioned on an array of (6,4).

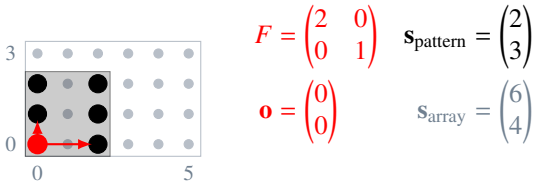

Figure 3. A sparse tile aligned on the axes of the array.

From an origin element ($\vec{r_q}$) in the array, a pattern has to be extracted. The fitting matrix is used to compute the regularly spaced point of the tile in the array. Equation to compute each tile element ($\vec{x}$) is given by (1), where $\vec{D}$ is the pattern shape, F the fitting matrix and $\vec{x_d}$ is any vector inside the pattern shape.

$$\forall \vec{x_d},\ \vec{0} \leq \vec{x_d} < \vec{D},\ \vec{x} = (\vec{r_q} + F \times \vec{x_d}) \mod \vec{m} \quad (1)$$

For each repetition in the repetition space ($\vec{Q}$), the origin element ($\vec{r_q}$) is built relatively to one origin vector ($\vec{o}$). Its coordinates are given by the following equation (2), which is a combination between paving matrix (P), the repetition index ($\vec{x_q}$) and the origin vector ($\vec{o}$).

$$\forall \vec{x_q},\ \vec{0} \leq \vec{x_q} < \vec{Q},\ \vec{r_q} = (\vec{o} + P \times \vec{x_q}) \mod \vec{m} \quad (2)$$

Figure 4 represents the complete description of a data-parallel task. A task called *Hfilter* is repeated (240,1080,∞). It consumes patterns with a shape of 13 and produces patterns with a shape of 3, which means it reads a one dimensional array of size 13 and writes one of size 3 for each repetition.

Detailed information about Array-OL can be found in [14].

3.2 Gaspard profile

The Gaspard model is at the highest level. Models are made at this level. In order to benefit from a visual modeling interface and some well known tools, the Gaspard models are designed in UML with the help of a profile. A profile allows to add semantics to the UML model. The Gaspard profile [15, 16] introduces stereotypes which allow application, hardware and allocation modeling. It is a subset of the OMG standard MARTE [1] (Modeling and Analysis of

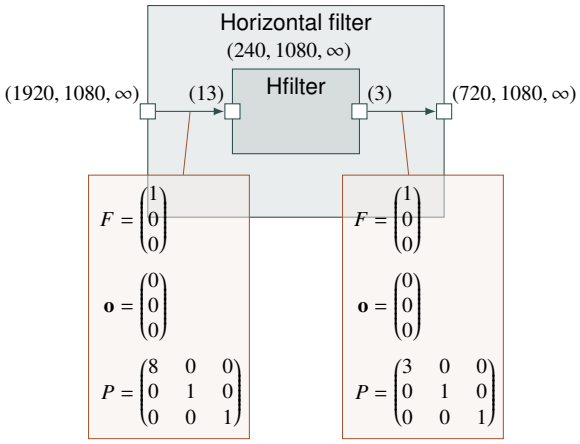

Figure 4. Data-parallelism

Real-Time and Embedded systems) profile. Gaspard profile is based on the Array-OL specification. In this section, we will briefly introduce few concepts of Gaspard, mandatory for the rest of the paper.

3.2.1 Application - Hardware architecture modeling

Gaspard models are composed of basic components: the *ElementaryComponents* (also called elementary tasks). An ElementaryComponent has no structure, it is a *black box* deployed on a function or on an Intellectual Property (IP). As Gaspard deals with all the parallelism of the application, it implies that the elementary tasks are sequential. Then components can be composed by component instances to make a *compound component*. An instance in a compound component is repeated if there are multiple identical and independent tasks. A repeated instance has a repetition space and is connected to other components via *tilers*. In Figure 5, which presents a row-column matrix multiplication, an instance *dP* of the component *dotProduct* is repeated (4,4) to produce each scalar of the output matrix. These basic concepts are used for both application modeling and hardware architecture modeling. It allows to make compact modeling of regular applications and hardware architecture.

3.2.2 Allocation

Once application and hardware architecture have been modeled, allocation of application on hardware has to be specified. Allocation has to be defined by the users, therefore it must be modeled too. We have to specify how applications will be executed on the execution platform. A *TaskAllocation* stipulates the allocation of an application on to a specific in-

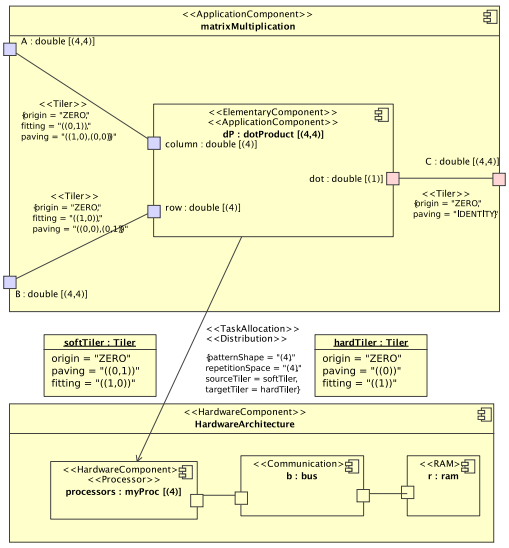

Figure 5. row-column multiplication mapped on a multiprocessor

stance of a processor. *Distribution* allows to map a repeated software task onto repeated computing elements. It is based on the Array-OL concepts. A *Distribution* is composed of two tilers, a repetition space (*rS*) and a pattern shape (*pS*). A pattern (with a *pS* size) is filled with a tiler which expresses how to fit repetition tasks into pattern. Then pattern elements are disseminated with a tiler which indicates on which repetition of processor each software repetition is distributed. Such process is repeated *rS* times. It is a powerful way to express regular allocation. Detailed explanation of the distribution mechanism could be found in [17].

High Performance Fortran [2] distribution could be done with such a mechanism. Classical distribution such as *Block, cyclic, k-cylic* can be modeled. Figure 5 shows the distribution of the *dotProduct* instance on a four processor model. Each processor will execute a column of computation. This distribution is equivalent to the following HPF distribution of the output array C:

```
!HPF$ PROCESSOR P(4)
!HPF$ DISTRIBUTE C(*,BLOCK)
```

3.3 From a Gaspard2 model to OpenMP code

We focused on the generation of parallel code for High Performance Computing. From a Gaspard model, the goal is to make a valid and efficient code generation.

For the generation, we considered that a thread is associated to a single processor. As optimized sequential tasks are used and knowing that those tasks have a good management of the processor pipeline, threads switching will penalize performance.

In agreement with the MDE approach, a few abstraction levels have been defined. Through these abstraction levels, which are detailed below, an architecture is targeted and specific code is generated. The use of some abstraction levels allow to decompose the transformation into small transformations which are easier to debug than a big one. Thus, from a certain abstraction level, different languages could be targeted. For instance, from the loop abstraction level, OpenMP and SystemC simulation are generated.

Transformation from a Gaspard model to OpenMP code is now presented step by step in the following sections. For each abstraction level, a metamodel has been defined.

3.3.1 Gaspard model to polyhedron model

During the transformation, models have to become closer to the targeted language. The first step is to produce a model representing the application on to the hardware architecture. At the high level, a Gaspard model is composed by different models:

- Application model
- Hardware architecture model
- Allocation model

These models are unified to produce a unique model. Hardware architecture model is the same as the one in the Gaspard high level. Allocation information is given to the software tasks. As models are still repetitive, it has to be kept in a compact way. The expression of the mapping of repeated tasks on repeated processors are expressed in the polyhedral model. The polyhedral model is used to solve the distribution on multiple processors. Polyhedrons are well known and several tools already use it [18]. Allocation information is transformed into polyhedron where processor number is a parameter of the polyhedron. Software tasks are parametrized by their corresponding polyhedron which express the mapping of the repeated tasks over the repeated computing resources at which they are linked to.

Polyhedron is build automatically from distribution information. It is composed of equality which come from the fitting and paving equations (1,2), and inequations to landmark polyhedrons (software repetition space, processors repetition space).

3.3.2 Polyhedron model to Loop model

Once we have a model with distribution in the polyhedral model, the next step is to generate a loop expression from each polyhedron. This is a usual problem studied since decades. CLooG (Chunky Loop Generator), written by Cédric Bastoul [18, 19], focused on the problem of scanning polyhedrons. CLooG is used to transform polyhedrons into loops.

Figure 6 shows a polyhedron generated from a block distribution of the *dotProduct* instance on four processors. Each processor (processor number is supposed to be p0) will execute one computation column. This polyhedron is given to CLooG which generates a loop to scan the polyhedron.

$$\begin{cases} p0 \geq 0, \quad 3 - p0 \geq 0 \\ -p0 + 0 * q0 + 1 * d0 + 0 = 0 \\ -x0 + 0 * q0 + 1 * d0 + 0 = 0 \\ -x1 + 1 * q0 + 0 * d0 + 0 = 0 \\ q0 \geq 0, \quad 3 - q0 \geq 0 \\ d0 \geq 0, \quad 3 - d0 \geq 0 \\ x0 \geq 0, \quad 3 - x0 \geq 0 \\ x1 \geq 0, \quad 3 - x1 \geq 0 \end{cases}$$

```
DO x1=0,3
     S1(d0 = p0,q0 = x1,x0 = p0)
END DO
```

Figure 6. Polyhedron generated and Fortran code generated by CLooG

3.3.3 Loop model to OpenMP model

A metamodel, which allows to model classical constructs of a procedural language, has been developed. It is inspired by the ANSI C Yacc grammar [20]. OpenMP statements have been added to this metamodel. The goal of this model is to use the same model for Fortran and C. From an OpenMP model, we are able to generate OpenMP Fortran or OpenMP C.

Generated code manages parallelism and control loop to distribute tasks repetition over processors. When an elementary task is used, a call to the adapted subroutine/function is done.

The transformation between the loop model and the OpenMP model requires different operations:

- scheduling of the task
- generation of synchronization barriers
- code generation for tiler computations

Scheduling the task is limited to an analysis of the data dependencies of the DAG. A task can be scheduled when all the data required by the task have been produced. A synchronization barrier is needed when a task needs data produced by another task. The algorithm is as follow: while there is a task to schedule, schedule tasks that could be done now, then a synchronization is made (waiting for data production of each task) then others tasks can be schedule.

Code generation for the tilers consists of generating a code to fit *tiles*. This code is generated only when they are no others solutions (for the complex tiles). In most of the cases, when patterns are regular and axes parallel, this generation is not used and is replaced by the adapted tiles in the original array.

3.3.4 Code generation

The last transformation is a model to code transformation. As the OpenMP model is close to a programming language, the code generation is just a "pretty printer".

4 Experimentations

In this paper, experimentations are made on a 3Ghz bi-Xeon dual core processor, running Linux with a total of 2 Gb of shared memory. We run the program over four threads unless precised otherwise.

In order to illustrate the use of this approach, we have modeled a classical operation: the matrix multiplication. In this study, we have use the GotoBLAS optimized library [21]. Five algorithms have been compared:

- Row column multiplication
- matrix multiplication by block
- matrix multiplication by block using sequential GotoBLAS
- Parallel GotoBLAS
- Sequential GotoBLAS

Computations are made with dense square matrices of size {2000,2000} and with rectangular matrices of size {3000,2000}×{4000,3000} . The first step of the study is to model the program.

4.1 Matrix multiplication modeling

Our benchmark programs consist of two matrix initializations and the call of the routine for the matrix-matrix multiplication. Row-column multiplication is build with the same expression as shown in Figure 5, where the *dotProduct* is deployed on the *DDOT* function. This very simple algorithm has a major drawback: it does not respect the memory hierarchy, and thus we expect a lot of cache misses using this model. That is why we also propose a basic model for the matrix multiplication using different levels of blocks (see Figure 7).

The *MatrixMultiplication* component takes as input two matrices ({2000,2000} for the square case) and produces an output matrix of the right size. It is composed of one

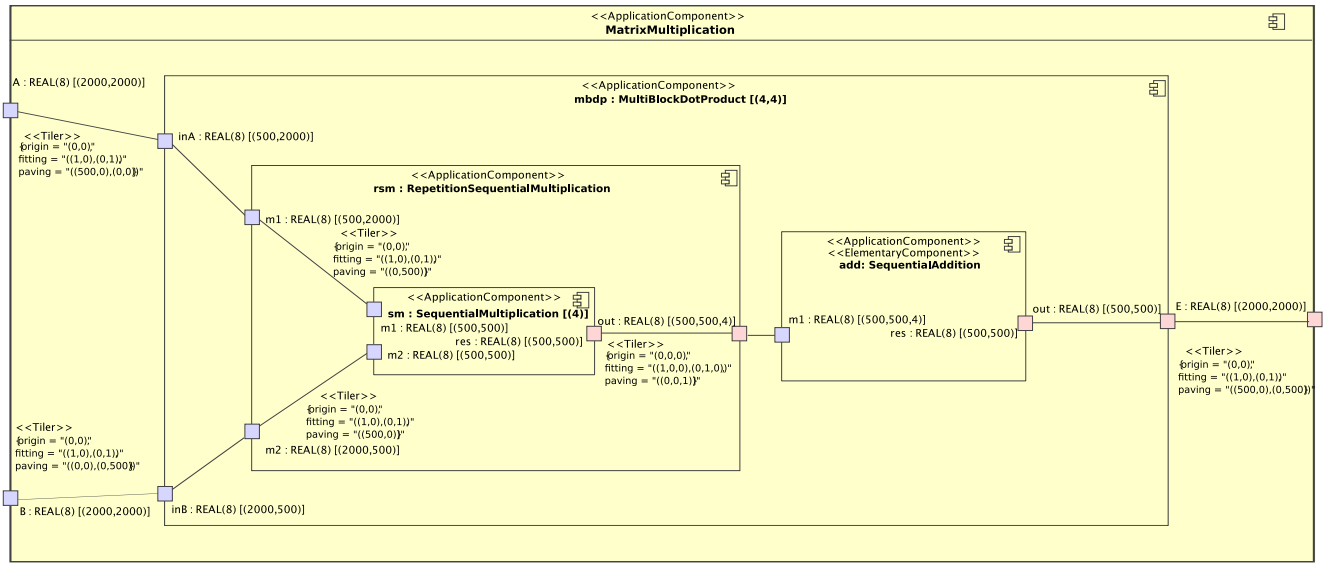

Figure 7. Model of matrix multiplication by block

instance of *MultiBlockDotProduct* which is repeated {4,4} times. This repeated task is distributed over the processors: each processor deals with a {2,2} computation block.

Then *MultiBlockDotProduct* runs sequentially. It should be handle by an Elementary Task (ET), provided by an optimized library. Here we use the *DGEMM* routine from GotoBLAS.

We also propose a model of this algorithm using different levels of block on each thread to decrease the cache-misses. Our model here has two levels. We describe here the first level, composed by two component instances: *rsm* instance of *RepetitionSequentialMultiplication* and *add* instance of SequentialAddition. The *RepetitionSequentialMultiplication* is also composed of a repetitive instance. *SequentialAddition*[1]is an *ElementaryComponent* which makes reduction of {500,500,4} arrays into {500,500} arrays.

The Figure 7 shows only the first layer of the block hierarchy. The real model we use goes down to the scalar multiplication with 2 levels but it can not be easily shown in a unique readable figure.

4.2 Results

Results shown in Figure 8 and in Figure 9 are the average of 100 consecutive executions.

The four first lines show a parallel multiplication (on four threads) whereas the last line is the sequential time on one processor for the optimized hand-written library. The execution time decreases for the more complex algorithms, which means algorithm 2 and 3 take a better advantage of the

[1]We use here an elementary task because reduction concepts are not yet available in the Gaspard framework.

Algorithm	Best execution time	Average
Row-column algorithm	0:21.11	0:21.60
Block multiplication	0:12.59	0:13.17
Block multiplication with GotoBLAS task	0:01.25	0:01.26
Parallel GotoBLAS	0:01.03	0:01.05
Sequential GotoBLAS	0:03.34	0:03.39

Figure 8. Square matrices - Execution times on four threads - Average is made on 100 executions

Algorithm	Best execution time	Average
Row-column algorithm	1:53.79	1:57.87
Block multiplication	0:32.46	0:33.52
Block multiplication with GotoBLAS task	0:03.97	0:04.04
Parallel GotoBLAS	0:02.91	0:02.96
Sequential GotoBLAS	0:09.89	0:10.03

Figure 9. Rectangular matrices - Execution times on four threads - Average is made on 100 executions

hardware. The best execution time while using Gaspard is when we use sequential optimized elementary task when the task belongs to only one process. That is how we recommend to use Gaspard, as a framework dealing with the parallelism, and rely on optimized libraries for the sequential parts. Such an approach is suitable for coarse grain parallelization. Tasks are distributed over threads and use optimized components.

5 Conclusion and further work

We have presented a model driven approach for metacomputing, it should simplify the writing, maintenance and reuse of applications. And it can be used by non specialists, without knowing parallel programming, to generate parallel programs easily. Gaspard generates OpenMP Fortran code but generating OpenMP C code is immediate if the feature is needed because of the metamodels. The last model is actually extremely close to the generated code. The design framework is a full graphic environment based on Eclipse and MagicDraw, inside which we can create the application model, architecture model and the association between both. The generated code is good enough if the algorithm is well adapted to the target architecture, but can not compete with the state-of-art hand-written libraries. Therefore Gaspard models should use these optimized sequential libraries as building blocks, and deal only with the parallel part of the application. This could help the parallelization of applications using intensive computations, like in numerical simulations or more generally in scientific computations.

The next step of this work will be to extend the range of architectures addressed and we plan to manage distributed memory architectures using Message Passing Interface (MPI). This is in fact a platform widely used in the scientific computing field, and our generated codes could then be executed on many new powerful calculators, especially on clusters or even on grids.

An other important improvement of our framework is an extension of the application metamodel to unlock a few limitations of ArrayOL. First one concerns the loops. In order to model iterative algorithms, we need to express convergence loops, which means we have no idea about the iteration space before the execution.

The second one is to add some dynamicity in the tilers: for the moment, shapes are given at the design phase and their size is fixed. For the needs of linear algebra algorithms, we plan to add parametrized tilers which allow to adapt the tiler during the iterations. This typically occurs when we program an algorithm like the QR factorization using Householder reflections where the dimension of the reflection space decreases when the algorithm progresses.

All the approaches will be validated on a 3D electromagnetic simulation code which has been developed in L2EP. This code is highly hierarchic and implies all the classic steps of software for modeling and simulation of physical phenomena: discretization of the structure, solving non linear equations with a Newton-like algorithm, solving linear equations with preconditioned Krylov or multigrid solvers. Then our framework becomes also a tool to study algorithms comportment. In fact, because of its high modular structure, we can easily change an algorithm by another: this is just changing the algorithm instantiation. Finally we can imagine to integrate these algorithms choices inside Gaspard itself: the choices would be guide by the architecture model.

References

[1] Object Management Group, Inc., ed.: UML 2 Superstructure (Available Specification). `http://www.omg.org/cgi-bin/doc?ptc/2004-10-02` (2004)

[2] High Performance Fortran Forum: High Performance Fortran language specification, version 2.0. Rice University, Houston, TX (1997)

[3] Planet MDE: Model Driven Engineering (2007) `http://planetmde.org`.

[4] Object Management Group, Inc.: MOF Query / Views / Transformations. `http://www.omg.org/docs/ptc/05-11-01.pdf` (2005) OMG paper.

[5] Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM, Montego Bay, Jamaica (2005)

[6] Muller, P.A., Fleurey, F., Vojtisek, D., Drey, Z., Pollet, D., Fondement, F., Studer, P., Jézéquel, J.M.: On executable meta-languages applied to model transformations. In: Model Transformations In Practice Workshop, Montego Bay, Jamaica (2005)

[7] Board, O.A.: Model driven architecture (MDA). Technical Report ormsc/2001-07-01, OMG (2001)

[8] Callahan, D., Chamberlain, B.L., Zima, H.P.: The Cascade High Productivity Language. In: 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments, IEEE Computer Society (2004) 52–60

[9] Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessn, J.W., Ryu, S., Jr., G.L.S., Tobin-Hochstadt, S.: The Fortress Language Specification Version 1.0 Beta. Technical report, Sun Microsystems, Inc. (2007)

[10] Lau, K.K., Wang, Z.: A Taxonomy of Software Component Models. In: 31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'05), IEEE (2005)

[11] Pllana, S., Fahringer, T.: On Customizing the UML for Modeling Performance-Oriented Applications. In: UML. (2002) 259–274

[12] OpenMP Architecture Review Board: OpenMP application program interface. Technical report (2005) http://www.openmp.org/drupal/mp-documents/spec25.pdf.

[13] Soula, J., Marquet, P., Dekeyser, J.L., Demeure, A.: Compilation principle of a specification language dedicated to signal processing. In: Sixth International Conference on Parallel Computing Technologies, PaCT 2001, Novosibirsk, Russia, Lecture Notes in Computer Science vol. 2127 (2001) 358–370

[14] Boulet, P.: Array-OL revisited, multidimensional intensive signal processing specification. Research Report RR-6113, INRIA (2007)

[15] Cuccuru, A., Dekeyser, J.L., Marquet, P., Boulet, P.: Towards UML 2 extensions for compact modeling of regular complex topologies. In: MoDELS/UML 2005, ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica (2005)

[16] Ben Atitallah, R., Boulet, P., Cuccuru, A., Dekeyser, J.L., Honoré, A., Labbani, O., Le Beux, S., Marquet, P., Piel, E., Taillard, J., Yu, H.: Gaspard2 uml profile documentation. Technical Report 0342, INRIA (2007)

[17] Boulet, P., Marquet, P., Piel, E., Taillard, J.: Repetitive Allocation Modeling with MARTE. In: Forum on specification and design languages (FDL'07), Barcelona, Spain (2007) Invited paper.

[18] Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques, Juan-les-Pins (2004) 7–16

[19] document, W.W.W.: (Cloog home page) URL: http://www.cloog.org.

[20] document, W.W.W.: (Ansi c yacc grammar) URL: http://www.lysator.liu.se/c/ANSI-C-grammar-y.html.

[21] Goto, K., van de Geijn, R.: On Reducing TLB Misses in Matrix Multiplication. Technical Report TR-2002-55, The University of Texas at Austin, Departement of Computer Sciences (2002) FLAME Working Note #9.