Model-Driven Transformations for Mapping Parallel Algorithms on Parallel Computing Platforms

Ethem Arkin[1], Bedir Tekinerdogan[2]

[1]Aselsan MGEO, Ankara, Turkey
earkin@aselsan.com.tr
[2]Bilkent University, Dept. of Computer Engineering, Ankara, Turkey
bedir@cs.bilkent.edu.tr

Abstract. One of the important problems in parallel computing is the mapping of the parallel algorithm to the parallel computing platform. Hereby, for each parallel node the corresponding code for the parallel nodes must be implemented. For platforms with a limited number of processing nodes this can be done manually. However, in case the parallel computing platform consists of hundreds of thousands of processing nodes then the manual coding of the parallel algorithms becomes intractable and error-prone. Moreover, a change of the parallel computing platform requires considerable effort and time of coding. In this paper we present a model-driven approach for generating the code of selected parallel algorithms to be mapped on parallel computing platforms. We describe the required platform independent metamodel, and the model-to-model and the model-to-text transformation patterns. We illustrate our approach for the parallel matrix multiplication algorithm.

Keywords: Model Driven Software Development, Parallel Computing, High Performance Computing, Domain Specific Language, Tool Support.

1 Introduction

The famous Moore's law which states that the performance of the processing power doubles every eighteen months is coming to an end due to the physical limitations of a single processor [11]. To keep increasing the performance of the processing power the current trend is towards applying parallel computing on multiple nodes. Unlike serial computing in which instructions are executed serially, multiple processing elements are used to execute the program instructions in parallel. An important challenge in parallel computing is the mapping of the parallel algorithm to the parallel computing platform. The mapping of the algorithm requires the analysis of the algorithm, writing the code for the algorithm and deploying it on the nodes of the parallel computing parallel computing platform. This mapping can be done manually in case we are dealing with a limited number of processing nodes. However, the current trend shows the dramatic increase of the number of processing nodes for parallel computing platforms with now about hundreds of thousands of nodes providing petascale to exascale level processing power [8]. As a consequence mapping the parallel algorithm to computing platforms has become intractable for the human parallel computing engineer.

Once the mapping has been realized in due time the parallel computing platform might need to evolve or change completely. In that case the overall mapping process must be redone from the beginning requiring lots of time and effort.

In this paper we provide a model-driven approach for both the mapping of parallel algorithms to parallel computing platform, and the evolution of the parallel computing platform. In

essence our approach is based on the model-driven architecture design paradigm that makes a distinction between platform independent models and platforms specific models or code. We provide a platform independent metamodel for parallel computing platform and define the model-to-model transformation patterns for realizing the platform specific parallel computing platforms. Further we provide the model-to-text transformation patterns for realizing the code from the platform specific models.

The remainder of the paper is organized as follows. In section 2, we describe the problem statement. Section 3 presents the implementation approach for mapping the parallel algorithm to parallel computing platform by the help of model transformations. Section 4 presents the related work and finally we conclude the paper in section 5.

2 Problem Statement

To define a feasible mapping the parallel algorithm needs to be analyzed and a proper configuration of the given parallel computing platform is required to meet the corresponding quality requirements for power consumption, efficiency and memory usage. To illustrate the problem we will use the parallel matrix multiplication algorithm [10]. The pseudo code of the algorithm is shown inFig.1a. The matrix multiplication algorithm recursively decomposes the matrix into subdivisions and multiplies the smaller matrices to be summed up to find the resulting matrix. The algorithm is actually composed of three different sections. The first serial section is the multiplication of subdivision matrix elements (line 3), which is followed by a recursive multiplication call for each subdivision (line 5-15). The final part of the algorithm defines the summation of the multiplication results for each subdivision (line 13-16).

Given a physical parallel computing platform consisting of a set of nodes, we need to define the mapping of the different sections to the nodes. In this context, the logical configuration is a view of the physical configuration that defines the logical communication structure among the physical nodes. Typically, for the same physical configuration we can have many different logical configurations [2]. An example of a logical configuration is shown inFig.1b. In this paper we assume that a feasible logical configuration is selected and the mapping of the code need to be realized.

```
1.  Procedure Matrix-Multiply(A, B, s):
2.  if s=1 then
3.  C = A * B
4.  endif
5.  P0 = Matrix-Multiply(A00, B00, s-1)
6.  P1 = Matrix-Multiply(A01, B10, s-1)
7.  P2 = Matrix-Multiply(A00, B01, s-1)
8.  P3 = Matrix-Multiply(A01, B11, s-1)
9.  P4 = Matrix-Multiply(A10, B11, s-1)
10. P5 = Matrix-Multiply(A11, B10, s-1)
11. P6 = Matrix-Multiply(A10, B01, s-1)
12. P7 = Matrix-Multiply(A11, B11, s-1)
13. C00 = P0 + P1
14. C01 = P2 + P3
15. C10 = P4 + P5
16. C11 = P6 + P7
```

a) b)

Fig.1.Matrix Multiplication Algorithm (a) to be mapped on (b) logical configuration platform

Fig.2 shows an example of a manually written C code for the matrix multiplication algorithm. The code is implemented using the MPI [12], a widely used parallel programming library. For simplicity, we assume that a 2x2 physical configuration is selected. Hence, the example code is defined for a four node logical configuration. Before starting the code it is required to initialize the MPI configuration and related variables (line 3). For succinctness we have omitted the code in the figure. The algorithm will run in parallel on four nodes. To distinguish among the nodes the variable rank defines four different ids including 0, 1, 2, and 3. From line 4 to 8

the code for node 0 is defined which sends the sub matrices to the other nodes (1,2,3). Lines 9 to 14 define the code for receiving the matrices in node 1. A similar code is implemented for the nodes 2 and 3 (not shown in the figure). Line 16 defines a so-called barrier to let the process wait until all the sub-matrices have been distributed and received by all the nodes. After the distribution of the sub-matrices to the nodes, each node runs the code as defined in line 17-18 and, as such, multiplies, the received sub-matrices. Once the multiplication is finalized the results are submitted to node 0, which is shown in line 19-22 for node 1 (code for node 2 and 3 is not shown). Line 23 to 25 defines again the collection of the results in node 0. Line 27 defines again a barrier to complete this process. Finally in line 28 to 33 the results are summed in node 0 to compute the resulting matrix C.

```
1.  #include "mpi.h"
2.  int main
3.  {//MPI initializations
4.  if(rank == 0) {
5.  MPI_Isend(A_0_0, 4, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &request);
6.  MPI_Isend(B_0_0, 4, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &request);
7.  MPI_Isend(A_0_1, 4, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &request);
8.  MPI_Isend(B_1_0, 4, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &request);}
9.  if(rank == 1) {
10. MPI_Irecv(A_0, 4, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &request);
11. MPI_Irecv(B_0, 4, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &request);
12. MPI_Irecv(A_1, 4, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &request);
13. MPI_Irecv(B_1, 4, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &request);}
14. ...
15. MPI_Barrier(MPI_COMM_WORLD);
16. //SERIAL SECTION PART (RUN ON ALL NODES)
17. C_0 = A_0 * B_0;
18. C_1 = A_1 * B_1;
19. if(rank == 1) {
20. MPI_Isend(C_0, 4, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD, &request);
21. MPI_Isend(C_1, 4, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD, &request);
22. }
23. if(rank == 0) {
24. MPI_Irecv(P_0, 4, MPI_DOUBLE, 2, MPI_ANY_TAG, MPI_COMM_WORLD, &request);
25. MPI_Irecv(P_1, 4, MPI_DOUBLE, 2, MPI_ANY_TAG, MPI_COMM_WORLD, &request);}
26. ...
27. MPI_Barrier(MPI_COMM_WORLD);
28. // SERIAL SECTION PART (RUN ON FIRST NODE)
29. if(rank == 0) {
30. C00 = P 0 + P 1
31. C01 = P_2 + P_3
32. C10 = P_4 + P_5
33. C11 = P_6 + P_7 }
34. MPI_Finalize();}
```

Fig.2.Example parallel code of the matrix multiplication algorithm code

After the code implementation, we can allocate and deploy the developed code to the nodes of the parallel computing platform. In our example we have assumed a simple configuration consisting of four nodes. Here we could easily decide on the strategy for sending, receiving and collecting the data over the nodes. However, one can imagine easily that the code for the larger configurations such as in petascale and exascale becomes dramatically larger, the strategy for the data distribution will be much more difficult [4] and likewise the effort to implement the code will be much higher. Because of the size and complexity implementing the code is not trivial and can become easily error-prone. In case of platform evolution or change the whole code needs to be substantially adapted or even rewritten from scratch.

3 Implementation Approach

To support the implementation and deployment of the code for the parallel computing algorithm on the parallel computing platform we propose a model-driven development approach. The approach integrates the conventional analysis of parallel computing algorithms with the

model-driven development approaches. The overall approach is shown inFig.3. In the first step of the approach the parallel computing algorithm is analyzed to define and characterize the sections that need to be allocated to the nodes of the parallel computing platform. In the second step, the plan is defined for allocating the algorithm sections to the corresponding nodes of the logical computing platform. In the third step the code for each serial section is manually implemented. The fourth step includes the implementation or reuse of predefined model transformations to generate the code for parallel sections. The final step includes the deployment of the code on the physical configuration platform. The details of the steps are described in the following sub-sections.

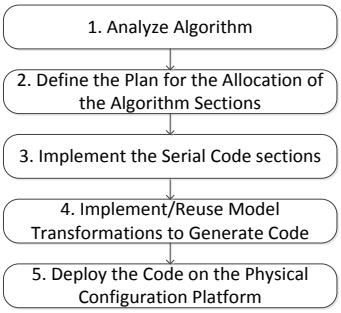


Fig.3.Approach for Generating/Developing and Deployment of Parallel Algorithm Code

3.1 Analyze Algorithm

The analysis of the parallel algorithm identifies the separate sections of the algorithm and characterizes these as serial or parallel sections. Here, a section is defined as a coherent set of instructions in the algorithm. A serial section defines the part of the algorithm that needs to run serially on nodes without interacting with other nodes. A parallel section defines the part of the algorithm that runs on each node and interacts with other nodes. For example the matrix multiplication algorithm (Fig.1a) has four main sections as shown in Table 1.

Table 1.Analysis of algorithm sections

NO	Algorithm Section	Section Type
1	Distribute the sub-matrices	PAR
2	C = A * B	SER
3	Collect matrix multiply results	PAR
4	C00 = P0 + P1 C01 = P2 + P3 C10 = P4 + P5 C11 = P6 + P7	SER

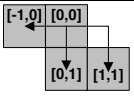
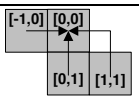
The first section defines the distribution of the sub-matrices to the different nodes. This section is characterized as a parallel section (PAR). The second section is characterized as serial (SER) and defines the set of instructions for the multiplication of the sub-matrices. The third section is a parallel section and defines the collection of the results of the matrix multiplications. Finally, the fourth section is characterized as serial and defines the summation of the result to derive the final matrix.

3.2 Define the Plan for the Allocation of the Algorithm Sections

The next step of the implementation approach is to define the plan for mapping the algorithm sections to logical configurations. Usually many different logical configurations can be derived for a given parallel algorithm and parallel computing platform. We refer to our earlier paper [2] in which we define the overall approach for deriving feasible logical configuration alternatives with respect to speed-up and efficiency metrics. In this paper we assume that a

feasible logical configuration has been selected and elaborate on the generation of the implementation of the algorithm sections.

Table 2.Plan for allocating sections to nodes

NO	Algorithm Section	Section Type	Plan
1	Distribute the sub-matrices	PAR	
2	C = A * B	SER	Run on each node
3	Collect matrix multiply results	PAR	
4	C00 = P0 + P1 C01 = P2 + P3 C10 = P4 + P5 C11 = P6 + P7	SER	Run on each node

The allocation of the sections to the nodes depends on the type of the sections. The plan for the matrix multiplication algorithm is shown in the fourth column of Table 2. Here we assume that each serial section runs on each node (section 2 and 4). The plan for allocating the parallel sections is defined as a pattern of nodes. The rectangles represent the nodes; the arrows represent the interactions (distribution or collection) among the nodes. Further, each node is assigned an id defining the coordinate of the node in the logical configuration. For section 1 the distribution of the data is represented as a pattern of four nodes in which the dominating node is the node with coordinate (0, 0). The arrows in the pattern show the distribution of the sub-matrices from the dominating node to the other nodes. For section 3 the pattern represents the collection of the results of the multiplications to provide the final matrix.

In the given example we have assumed a logical configuration consisting of four nodes. Of course for larger configurations defining the allocation plan becomes more difficult. Hereby, the required plan is not drawn completely but defined as a set of patterns that can be used to generate the actual logical configuration. For example, scaling the patterns of Table 2can be used to generate the logical configuration ofFig.1b. For more details about the generation of larger logical configurations from predefined patterns we refer to our earlier paper [2].

3.3 Implement the Serial Code Sections

Once the plan for allocating the algorithm sections to the logical configuration is defined we can start the implementation of the algorithm sections. Hereby, the code for the serial sections is implemented manually.

Table 3.Implementation of the serial sections

NO	Algorithm Section	Implementation
1	Distribute the sub-matrices	Will be generated
2	C = A * B	C0 = A_0 * B_0 C1 = A_1 * B_1
3	Collect matrix multiply results	Will be generated
4	C00 = P0 + P1 C01 = P2 + P3 C10 = P4 + P5 C11 = P6 + P7	C00 = P_0 + P_1 C01 = P_2 + P_3 C10 = P_4 + P_5 C11 = P_6 + P_7

The code for the parallel sections are generated using the model-transformation patterns as defined in the next sub-section. The third column of Table 3 shows the implementation of the serial sections of the matrix multiplication algorithm. Note that the implementation is alignment with the complete implementation of the algorithm as shown in Fig.2.

3.4 Model Transformations

After analyzing the algorithm, implementing the code for serial algorithm sections and defining the plan for mapping these sections to the logical configuration, the code for the parallel sections will be generated. To support platform independence this code generation process is realized in two steps using model-to-model transformation and model-to-text transformation. These transformation steps are described below.

Model-to-Model Transformation.

For different parallel computing platforms, there are several parallel programming languages such as, MPI, OpenMP, MPL, CILK [15]. According to the characteristic of the parallel computing platforms, different programming languages can be selected. Later on in case of changing requirements a different platform might need to be selected. To cope with the platform independence and the platform evolution problem we apply the concepts as defined in the Model-Driven Architecture (MDA) paradigm [13]. Accordingly, we make a distinction between platform independent models (PIM), platform specific models (PSM) and the source code. The generic model-to-model transformation process is shown in Fig.4.

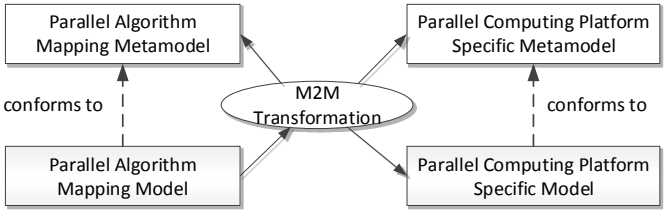


Fig.4.Model-to-model transformation.

Here the transformation process takes as input a platform independent model called, parallel algorithm mapping model. This model defines the mapping of the algorithm sections to the logical configuration. The model conforms to the parallel algorithm mapping metamodel which we will explain later in the section. The output of the transformation process is a platform specific model, called parallel computing platform specific model. Similarly this model conforms to its own metamodel, which typically represents the model of the language of the platform (e.g. MPI metamodel). The platform specific model will be later used to generate the code using model-to-text transformation patterns.

```
Algorithm: 'entity' name = ID '{'(sections+=Section*) '}';
Section:'abstract entity' name = ID '{'(sections+=Section*) '}';
SerialSection:
  'entity' name = ID ('extends' superType = [Section])? '{'code = STRING '}';
ParallelSection:
  'entity' name = ID ('extends' superType = [Section])? '{'tiles += Pattern*'}';
LogicalConfiguration:'entity' name = ID '{'(tiles+=Tile*)'}';
Tile:'abstract entity' name = ID '{''}';
Core:'entity' name = ID ('extends' superType = [Tile])?'{'i = INTj = INT '}';
Pattern:'entity' name = ID ('extends' superType = [Tile])?
'{' tiles += Tile*dominationgs += Tilecomms += Communication*
xsize = INTysize = INT'}';
Communication:'entity' name = ID '{'
    from = Coreto = Corelegsize = INTfromData = DatatoData = Data '}';
```

Fig.5.Concrete Syntax of the Parallel Algorithm Mapping Metamodel (PAMM)

The grammar for the parallel algorithm mapping metamodel is defined in *XText* in the Eclipse IDE and shown in Fig.5. Here, Algorithm consists of Sections, which can be either a *ParallelSection* or *SerialSection*. Each section can itself have other sections. In the grammar the serial sections are related to code implementations in the code block. The parallel sections include the data about the mapping plan that is determined with the logical configuration. *LogicalConfiguration* consists of *Tile* entity which can be either a single *Core* (processing

unit) or a *Pattern* with tiles and communications between these tiles. The assets related with the logical configuration with cores and patterns compose the plan for mapping algorithm to logical configuration.

Fig.6 shows, for example, the parallel algorithm mapping model for the matrix multiplication algorithm. In the figure two serial sections *MultiplyBlock* and *SumBlock* are defined. In the *MultiplyBlock* section the matrices are divided into sub-matrices and scattered by using the B2S pattern. The B2S pattern is a predefined pattern in the toolset indicating the pattern for section 1 as defined in the fourth column of Table 2. This multiply block also contains a Multiply serial section which contains the serial implementation of the multiply operation. In the *SumBlock* section, the resulting matrices are gathered by the pattern *B2G* which is predefined for section 3 as shown in the fourth column of Table 2. The *SumBlock* serial section contains the serial code for summation of the resulting sub-matrices.

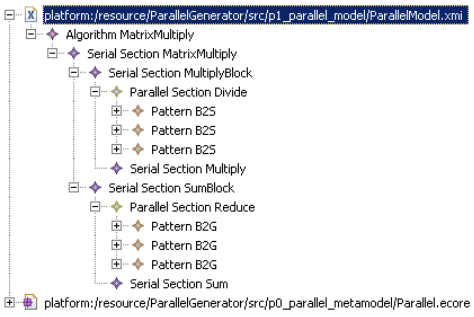


Fig.6.Parallel Algorithm Mapping Model for the Matrix Multiplication Algorithm

Once the platform independent parallel algorithm mapping model is defined we can transform it to the required platform specific model. We assume, for example, that the aim is to generate a MPI model. Fig.7shows the grammar of the MPI metamodel that is again defined using XText. In the metamodel each MPI model consists of a group of entities, which include MPISection, Process, Node, and Communication. Each section consists of processes and communication among these processes. Each Process allocates to a Node. Each communication defines the destination and target process.

```
MpiModel:'entity' name = ID '{'(groups+=MpiGroup*)'}';
MpiGroup:'entity' name = ID '{'(sections+=MpiSection*)(nodes+=Node)'}';
MpiSection:'entity' name = ID '{'
    (sections+=MpiSection*)(processes+=Process*)
    (communications+=Communication*)code = STRING'}';
Process:'entity' name = ID '{'rank = INTallocates=Node'}';
Node:'entity' name = ID '{''}';
Communication:'entity' name = ID '{'from = Processto = Process '}';
```

Fig.7.Grammar of the MPI Metamodel

The model-driven transformation rules refer to elements of both the PAMM and the parallel computing platform specific metamodel, in this case the MPI Metamodel. The M2M transformation rules are implemented using the ATL [1] transformation language. The transformation rules are shown in Fig.8. As shown in the figure we have implemented four different rules which define the transformations of mapping patterns to MPI sections, cores to processes and communications to MPI communications.

The rule *Algorithm2MpiModel*, is defined as the main rule of the transformation. The rule *Pattern2Section* transforms the algorithm pattern sections to *MpiSection* within the *MpiGroup*. The rule *Core2Process* transforms the cores as defined in the patterns to the processes in *MpiSection*. Each process is transformed from the core with the data of rank calculated from

the index values of the core. Similarly, *Comm2Comm* transforms the communications that are defined in the patterns, to the communications in *MPISection*.

```
1.  rule Algorithm2MpiModel {
2.    from algorithm: ParallelModel!Algorithmtompimodel: MpiModel!MpiModel (
3.  name<-algorithm.name, groups<-OrderedSet{mpiGroup}),
4.    mpiGroup: MpiModel!MpiGroup(name<-algorithm.name,
5.     sections<-algorithm.getPatterns())}
6.  rule Pattern2Section {
7.    from pattern: ParallelModel!Patternto section: MpiModel!MpiSection (
8.      name<-pattern.name,processes <- pattern.getCores(),
9.      communications<-pattern.getCommunications())}
10. rule Core2Process {from core: ParallelModel!Core to process: MpiModel!Process (
11.     rank<-core.i.mod(core.getGlobalSize())*core.getGlobalSize() +
12.           core.mod(core.getGlobalSize()),)}
13. rule Comm2Comm {from p_communication:ParallelModel!Communication
14. to communication : MpiModel!Communication (
15.     from<-p_communication.from, to<-p_communication.to,)}
```

Fig.8.Transformation rules from PAMM to MPI metamodel

The MPI model which is the result of the model-to-model transformation is shown in Fig.9. The MPI model includes the *MpiSection* with processes that will run on each node, communications from a destination process to target process and the serial code section implementation. This MPI model is now ready for model-to-text transformation to generate the final MPI source code.

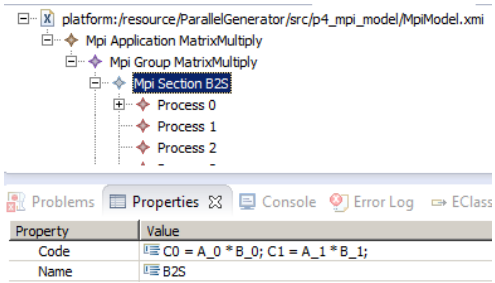


Fig.9.Part of the MPI model generated by model-to-model transformation

Model-to-Text Transformation

The generated PSM includes the mapping of the processes specific to the parallel computing platform. Subsequently, this PSM is used to generate the source code. The model-to-text transformation pattern for this is shown inFig.10.

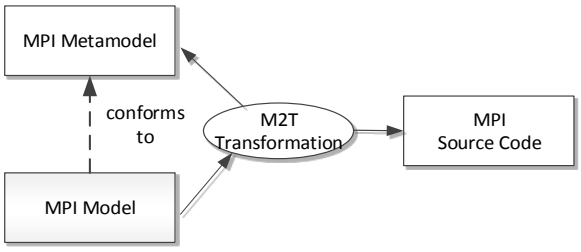


Fig.10.Example model transformation chain of MPI model

Fig.11 shows the implementation of the model-to-text transformation for which we used the XPand [18] transformation language. To map the sections to the parallel computing platform, for each section the communication operations for the data is generated for target and destination process ranks (line 6 to 11). Subsequently, the serial code implementation is imported to the source code in line 13. For each section, a barrier code is implemented to synchronize the section processes (line 14). The resulting code of the transformation is the code as defined in Fig.2.

```
1.  «IMPORT mpi»
2.  … // MPI initializations and type definitions
3.  «FOREACH groups AS group»
4.  «FOREACH group.sections AS section»
5.  «FOREACH section.communications AS comm»
6.  if(rank == «comm.from.rank») {
7.  MPI_Isend(«comm.fromData.name»,«comm.fromData.size»,MPI_«comm.fromData.type»,
8.            «comm.to.rank»,«comm.from.rank»,MPI_COMM_WORLD,&request);}
9.  if(rank == «comm.to.rank») {
10. MPI_Irecv(«comm.toData.name»,«comm.fromData.size»,MPI_«comm.toData.type»,
11.           «comm.from.rank»,MPI_ANY_TAG,MPI_COMM_WORLD,&request);}
12. «ENDFOREACH»
13. «section.code»
14. MPI_Barrier(MPI_COMM_WORLD);
15. «ENDFOREACH»«ENDFOREACH»
16. … // Final code
```

Fig.11. Transformation template from MPI metamodel to MPI source code

3.5 Deploy Code on Physical Configuration

The resulting code of the previous steps needs to be deployed on the physical configuration. The deployment can be done manually or using tool support in case of large configurations. In the literature various tools can be found which concern the automatic deployment of the code to the nodes of a parallel computing platform. We refer to, for example, [8][15][4] for further details.

4 Related Work

Several papers have been published in the domain of model-transformations for parallel computing. Palyart et. al. [14] propose an approach for using model-driven engineering in high performance computing. They focus on automated support for the design of a high performance computing application based on the distinction of different domain expertise like physical configuration, numerical computing, application architecture etc.

Bigot and Perez [3] adopt HLCM a hierarchical and generic component model with connectors originally designed for high performance applications. The authors represent on their experience with metamodeling and model transformation to implement HLCM. Gamatié et al. [7] introduced the GASPARD design framework systems that use model transformations for massively parallel embedded systems. They refined the MARTE models based on Model Driven Engineering paradigm. They provide tool support to automatically generate code with high-level specifications. Taillard et.al [16] implemented a graphical framework for integrating new metamodels to GASPARD framework. They used MDE paradigm to generate OpenMP, Fortran or C code.

Similar to our approach the above studies generate source code for high performance computing. The main difference of our approach is focus on the mapping of algorithm sections to parallel computing platforms.

5 Conclusion

In this paper we have described the model transformations needed to implement the mapping of a parallel algorithm to a parallel computing platform. In alignment with the MDA paradigm the approach is based on separating the platform independent parallel computing model from the platform specific parallel computing model and the source code. The model transformations do not only helps the parallel programming engineer to generate code but it also provides support for easier portability in case of platform evolution. We have illustrated the approach for the MPI platform but the approach is generic. In our future work we will elaborate on the application of model-driven approaches to parallel computing platform and focus on optimizing the values for metrics which are important for mapping parallel algorithms to parallel computing platforms.

References

1. ATL: ATL Transformation Language. http://www.eclipse.org/atl/
2. Arkin, E., Tekinerdogan, B., Imre, K. Model-Driven Approach for Supporting the Mapping of Parallel Algorithms to Parallel Computing Platforms. *Proc. of the ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems*. (2013)
3. Bigot, J., Perez, C. On Model-Driven Engineering to implement a Component Assembly Compiler for High Performance Computing. JournéessurlIngénierieDirigée par les Modèles, IDM 2011. (2011)
4. Cumberland, D., Herban, R., Irvine, R., Shuey, M., and Luisier, M. Rapid parallel systems deployment: techniques for overnight clustering. In *Proceedings of the 22nd conference on Large installation system administration conference* (LISA'08). USENIX Association, Berkeley, CA, USA, 49-57. (2008)
5. Foster, I. Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. *Addison-Wesley Longman Publishing Co.*, Inc., Boston, MA, USA. (1995)
6. Frank, M.P. The physical limits of computing. *Computing in Science &Engineering* , vol.4, no.3, pp.16,26, May-June 2002. (2002)
7. Gamatié, A., Le Beux, S., Piel, E., Ben Atitallah, R., Etien, A., Marquet, P., and Dekeyser, J. A Model-Driven Design Framework for Massively Parallel Embedded Systems. *ACM Trans. Embed. Comput. Syst.* 10, 4, Article 39. (2011)
8. Hoffmann, A., Neubauer, B. Deployment and configuration of distributed systems. In *Proceedings of the 4th international SDL and MSC conference on System Analysis and Modeling* (SAM'04), Daniel Amyot and Alan W. Williams (Eds.). Springer-Verlag, Berlin, Heidelberg, 1-16. (2004)
9. Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hiller, J., Karp, S., Keckler, S., Klein, D., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, R.S., Yelick, K., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hiller, J., Keckler, S., Klein, D., Williams, R.S., and Yelick, K., *Exascale Computing Study: Technology Challenges in Achieving Exascale Systems*. DARPA. (2008)
10. Li, K.Scalable parallel matrix multiplication on distributed memory parallel computers. *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International* , vol., no., pp.307,314. (2000)
11. Moore, G.E.Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE* , vol.86, no.1, pp.82,85. (1998)
12. MPI: A Message-Passing Interface Standart, version 1.1. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.
13. Object Management Group (OMG). Model Driven Architecture (MDA), ormsc/2001-07-01.
14. Palyart, M., Lugato, D., Ober, I., and Bruel, J. MDE4HPC: an approach for using model-driven engineering in high-performance computing. In *Proceedings of the 15th international conference on Integrating System and Software Modeling* (SDL'11), Iulian Ober and Ileana Ober (Eds.). Springer-Verlag, Berlin, Heidelberg, 247-261. (2011)
15. Stawinska, M., Kurzyniec, D., Stawinski, J., Sunderam, V., Automated Deployment Support for Parallel Distributed Computing, *Parallel, Distributed and Network-Based Processing, 2007. PDP '07. 15th EUROMICRO International Conference on* , vol., no., pp.139,146. (2007)
16. Taillard, J., Guyomarc'h, F.,Dekeyser, J. A Graphical Framework for High Performance Computing Using An MDE Approach. In *Proc. of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing* (PDP '08). IEEE Computer Society, Washington, DC, USA, 165-173. (2008)
17. Talia, D. Models and Trends in Parallel Programming. *Parallel Algorithms and Applications* 16, no. 2: 145-180. (2001)
18. Xpand, Open Architectureware. http://wiki.eclipse.org/Xpand.
19. Zheng, G., Kakulapati, G., Kale, L.V. BigSim: a parallel simulator for performance prediction of extremely large parallel machines. *Parallel and Distributed Processing Symposium, 2004. Proc.. 18th International* , vol., no., pp.78,, 26-30. (2004)