Software & Systems Modeling (2019) 18:2907–2935
https://doi.org/10.1007/s10270-018-00705-w

REGULAR PAPER

ParDSL: a domain-specific language framework for supporting deployment of parallel algorithms

Bedir Tekinerdogan[1] · **Ethem Arkin**[2]

Received: 10 October 2017 / Revised: 18 November 2018 / Accepted: 23 November 2018 / Published online: 17 December 2018

Abstract
An important challenge in parallel computing is the mapping of parallel algorithms to parallel computing platforms. This requires several activities such as the analysis of the parallel algorithm, the definition of the logical configuration of the platform and the implementation and deployment of the algorithm to the computing platform. However, in current parallel computing approaches very often only conceptual and idiosyncratic models are used which fall short in supporting the communication and analysis of the design decisions. In this article, we present ParDSL, a domain-specific language framework for providing explicit models to support the activities for mapping parallel algorithms to parallel computing platforms. The language framework includes four coherent set of domain-specific languages each of which focuses on an activity of the mapping process. We use the domain-specific languages for modeling the design as well as for generating the required platform-specific models and the code of the selected parallel algorithm. In addition to the languages, a library is defined to support systematic reuse. We discuss the overall architecture of the language framework, the separate DSLs, the corresponding model transformations and the toolset. The framework is illustrated for four different parallel computing algorithms.

Keywords Model-driven software development · Parallel programming · High-performance computing · Domain-specific language · Architecture framework

1 Introduction

It is now increasingly acknowledged that the processing power of a single processor has reached the physical limitations, and serial computing has thus reached its limits. To increase the performance, the current trend is toward applying parallel computing on multiple nodes typically including many CPUs. In contrast to serial computing in which instructions are executed serially, in parallel computing multiple processing elements are used to execute the program instructions simultaneously. To benefit optimally from the parallel computing, power parallel algorithms can be used that are executed simultaneously on multiple nodes. However, for optimal performance it is required that the parallel algorithm is properly mapped to the parallel computing platform. This requires several activities such as the analysis of the parallel algorithm, the definition of the logical configuration of the platform, and the implementation and deployment of the algorithm to the computing platform. These activities can be performed manually for small platforms but soon become intractable in case the size of the platform increases. Hence, to support these activities, it is important to adopt proper modeling approaches. On the one hand, modeling is important for representing a blueprint that can be used to support the communication among the stakeholders, support the reasoning about the design decisions during the mapping process, and for analyzing the design alternatives. On the other hand, modeling can be also used for supporting the automation process of the corresponding activities and as such increase the faster development and analysis of the system.

In current parallel computing approaches, however, there does not seem to be standard modeling approaches for supporting the design and analysis of parallel computing design models. Most approaches seem either to adopt

Communicated by Dr. Jeff Gray.

✉ Bedir Tekinerdogan
bedir.tekinerdogan@wur.nl

Ethem Arkin
earkin@aselsan.com.tr

[1] Information Technology, Wageningen University, Wageningen, The Netherlands

[2] Aselsan, Ankara, Turkey

conceptual modeling approaches in which the parallel computing elements are represented using idiosyncratic models or are generally low level and machine specific [1]. Other approaches borrow for example models from embedded and real-time systems and try to adapt these for parallel computing. The lack of a clear and precise modeling approach with first class abstractions for parallel computing impedes the solutions for analyzing, designing and communicating the design decisions.

To support the deployment of parallel algorithms, it is necessary to provide the appropriate precision level of modeling. Models are different in nature and quality, and different classifications of models have been provided in the literature. Mellor et al. [2] make a distinction between three kinds of models, depending on their level of precision. Based on this, a model can be considered as a Sketch, as a Blueprint, or as an Executable. According to [3], an executable model is a model that has everything required to produce the desired functionality of a single domain. Executable models are more precise than sketches or blueprints and can be interpreted by model compilers. A similar classification of models is defined by Fowler et al. [4] who suggests a distinction based on three levels of models, namely Conceptual Models, Specification Models and Implementation Models. In model-driven software development, the concept of models can be considered as executable models as defined by the above characterization of Mellor et al. [2]. Hereby, models are not mere documentation but become "code" that are executable and that can be further used to generate even more refined models or code. This contrasts with model-based software development in which models are used as blueprints at the most [2].

Developing domain-specific languages (DSLs) is an important approach for supporting model-driven software development and herewith the application of executable models. To model the particular concerns of parallel computing and to support the automation of the activities, we propose ParDSL, a domain-specific language framework consisting of four coherent set of domain-specific languages (DSLs) each of which focuses on a different aspect of the mapping process. For each DSL, we describe the abstract syntax and the concrete syntax. The DSLs are used in the generation of platform-specific models and the code of selected parallel algorithms that needs to be mapped on parallel computing platforms. Both the languages and the transformations are implemented using the Epsilon language. We illustrate the language framework for supporting the mapping of four different parallel algorithms to parallel computing platforms.

The remainder of the article is organized as follows. In Sect. 2, we describe the background on parallel computing and domain-specific languages. Section 3 discusses the required modeling concerns for parallel computing. Section 4 presents the domain-specific language framework and the included four DSLs. Section 5 presents the model transformations using the DSLs. Section 6 discusses the toolset that implements the language framework. Section 7 presents the evaluation of the DSL framework. Section 8 presents the related work, and finally we conclude the article in Sect. 9.

2 Background

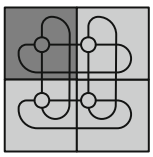
2.1 Parallel computing

The famous Moore's law states that the number of transistors on integrated circuits and likewise the performance of processors doubles approximately every 18 months [5]. Since its introduction in 1965, the law seems to have quite accurately described and predicted the developments of the processing power of components in the semiconductor industry [6]. Although Moore's law is still in effect, currently it is recognized that increasing the processing power of a single processor has reached the physical limitations [1]. Hence, to increase the performance the current trend is toward applying parallel computing on multiple nodes. Here, unlike serial computing in which instructions are executed serially, multiple processing elements are used to execute the program instructions simultaneously. To benefit further from the parallel computing, power parallel algorithms can be used that are executed simultaneously on multiple nodes. In general, a parallel algorithm can be mapped in different alternative ways to the processing nodes and research has been carried out to optimize the algorithm and the mapping process. This problem has gained even more attention with the dramatic increase in the processing nodes to tens and hundreds of thousands of nodes providing processing performance from petascale to exascale levels [7]. Once the feasible mapping is selected, the parallel algorithm needs to be transformed to the target parallel computing models such as MPI, OpenMP, MPL and CILK [8].

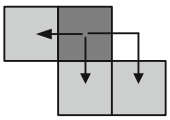
To define a feasible mapping, the parallel algorithm needs to be analyzed and a proper configuration of the given parallel computing platform is required to meet the corresponding quality requirements for power consumption, efficiency and memory usage. To illustrate this allocation problem, we will use, as an example, the parallel matrix multiplication algorithm for which the pseudo code is shown in Fig. 1a. We have selected this algorithm since it is popular and easy to understand in the parallel computing domain. 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 algo-

```
// Matrix Multiplication
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) *Serial Matrix Multiplication Algorithm*



(b) *Physical Configuration*



(c) *Logical Configuration*

```
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();}
```

(d) *Parallel Implementation of Matrix Multiplication Algorithm*

Fig. 1 Matrix multiplication algorithm

rithm 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. The processing units can be constructed within a configuration defined by the parallel computing platform architect as the *physical configuration* of nodes (Fig. 1b). The parallel algorithm is mapped to the parallel computing platform that defines the physical configuration of nodes. To reason about the mapping of the parallel algorithm, we adopt the term *logical configuration*, which represents a view of the physical configuration that defines the logical communication structure among the physical nodes (Fig. 1c). Once the logical configuration is defined, the corresponding code is implemented and deployed on the nodes (Fig. 1d).

For very large topologies including a large number of cores, the logical topology cannot be drawn on the same scale. Instead, for representing the topology in a more succinct way the topology can be defined as a regular pattern that can be built using *tiles*. Tiles as such can be considered as the basic building blocks of the logical configuration. The tile notation is used for addressing groups of processing elements that form a neighborhood region on which processes and communication links are mapped. The smallest part of a tile is a *processing element* (core).

Tiles can be used to construct the logical configuration using *scaling* that can be defined as the composition of the larger structure from the smaller tiles. In general, we can distinguish among different primitive tiles which can be constructed in different ways. The selected tiling configuration will be dependent on the required communication patterns of the algorithm that will be explained in the next subsection. Examples of primitive tiles are shown in Fig. 2 [9, 10].

Each primitive tile defines the structure among the nodes but initially does not describe the dynamic behavior among these nodes. Hence, after defining the primitive tiles, we need to define the dynamic behavior among the nodes. This is

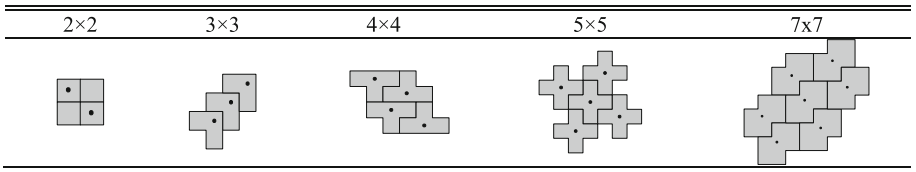


Fig. 2 Primitive tiling examples

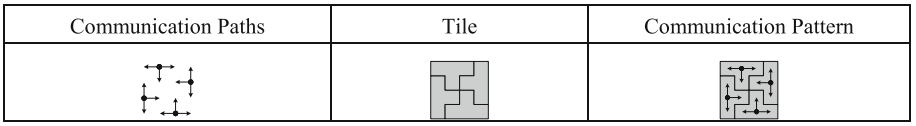


Fig. 3 Communication patterns constructed with tile and matching communication paths

defined using communication patterns for each tile configuration. A communication pattern includes communication paths that consist of a source node, a target node and a route between the source and target nodes. An example communication pattern is shown in Fig. 3.

For a given physical configuration, we can derive many different logical configurations. Each logical configuration will perform differently with respect to different quality concerns. Hereby, important metrics are *speedup* and *efficiency* [6]. *Speedup* refers to how much a parallel algorithm is faster than a corresponding sequential algorithm. *Efficiency* defines how well the processors are utilized in executing the algorithm.

2.2 Domain-specific languages

General-purpose languages (GPL), such as C, C# and Java, can be applied to arbitrary problem domains. In contrast, a domain-specific language (DSL) is a tailor-made language for a specific problem domain [11]. DSLs provide particular abstractions that are suitable for one particular problem domain. In the literature [12–16], several different benefits have been described for the development and use of DSLs among which increasing the development productivity. Presumably the amount of DSL code that needs to be written is substantially smaller compared to the use of general-purpose languages. A DSL can also support the communication among domain experts by providing a clear and precise language focused on the particular domain. In case a DSL abstracts away from the underlying technology platform, it can also support platform independence [17]. Hereby, the specifications in DSLs can be used to generate code for platform-specific execution environments. Further, using DSLs can increase the quality of the created product due to the removal of unnecessary degrees of freedom for programmers, the avoidance of duplication of code, and the consistent automation of repetitive work by the execution DSL engineering engine [16]. Finally, since DSLs are more semantically rich than GPL programs, a more elaborate and precise verification and validation of the domain concerns can be carried out. Several other benefits could be derived for

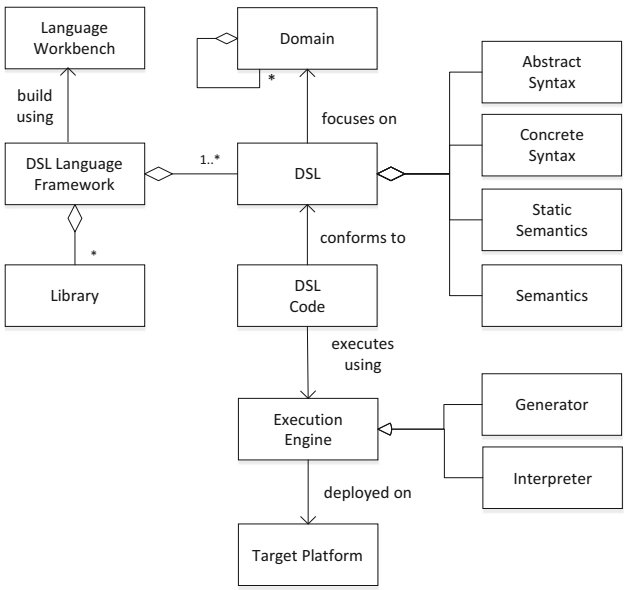


Fig. 4 Basic elements of DSLs

adopting DSLs. In general, multiple of these listed benefits together lead to the decision for adopting DSLs.

According to Voelter, a DSL is "a language that is optimized for a given class of problems, called a domain. It is based on abstractions that are closely aligned with the domain for which the language is built" [16]. Fowler defines a DSL as "a computer language that's targeted to a particular kind of problem, rather than a general-purpose language that's aimed at any kind of software problem" [3]. Several other definitions can be found in the literature, but there seems to be a general agreement that a domain-specific language is focused on a particular domain. Because of the focus on a particular domain, DSLs are usually small languages. The application of a systematic, disciplined, quantifiable approach to the development, use and maintenance of these languages is usually called *software language engineering* [18]. A number of concepts need to be known when considering the development of domain-specific languages. We summarize these in Fig. 4.

A DSL runs on a *target platform* and is assumed to be something we cannot change (significantly) during the DSL

development process [16]. The *execution engine* bridges the gap between the DSL and the platform and can be an *interpreter* or a *generator*. An interpreter is a program running on the target platform that loads a DSL program and then acts on it. A generator takes the DSL program and transforms it into an artifact that can run directly on the target platform. Usually, a distinction is made between *external versus internal DSLs*. An external DSL is represented in a separate language to the main programming language it's working with. An internal DSL is represented within the syntax of a general-purpose language [3, 16].

A DSL consist of the following main elements. The *abstract syntax* defines the vocabulary of concepts provided by the language and how they may be combined to create models. The *concrete syntax* defines the notation that facilitates the presentation and construction of models or programs in the language. It can be visual or textual. *Well-formedness rules (static semantics)* defines additional constraint rules on abstract syntax that are hard or impossible to express in standard syntactic formalisms of the abstract syntax. *Semantics* includes the definition of the meaning of the concepts in the abstract syntax.

Typically, we can distinguish two different roles: the *language engineer* and the *language user*. The language engineer is responsible for creating the language. The language user is the person who uses the language to develop applications.

One important design decision when developing domain-specific languages is the use of corresponding libraries. In fact, most programming languages have an associated core library which is made available by all implementations of the language. The inclusion of libraries helps to reduce the complexity of the language itself and in addition provides additional reuse of recurring program structures. The library can be considered separate from the language or be treated as part of the integrated whole.

A *language workbench* is an environment designed to help people create new DSLs, together with high-quality tooling required to use those DSLs effectively [3, 16]. A language workbench provides a set of tools for supporting the language engineer in creating a DSL.

Transformations are defined to transform DSL code written in a concrete syntax to another model representation or to (programming language) code that can be executed on a specific platform.

A platform consists of software building blocks that provide functions to implement the DSL's semantics in a specific system environment. The platform consists of generic platform artifacts, such as programming languages and frameworks (for example based on the Enterprise JavaBeans (EJB) technology or Microsoft.NET), as well as DSL-specific platform artifacts (i.e., parts of the software platform that need to be implemented or integrated into a generic platform just to support the DSL). DSL-to-platform transformations are conducted by a generator component and result in generated artifacts (such as Java, C#, or Ruby code for example) based on the respective platform (see, e.g., [19–21]). However, after transforming DSL code written in a concrete syntax to platform-specific artifacts, the remaining transformations are typically executed by standard tools of the target platform, such as compilers or interpreters (examples are a C compiler or a Tcl interpreter).

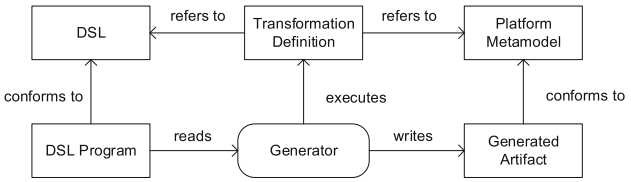


Fig. 5 Model transformation pattern

DSL code written in a concrete syntax can be transformed to another model representation or to code that can be executed on a particular platform. A DSL to platform transformation pattern is shown in Fig. 5. The generator takes as input a DSL program and transforms it into a generated artifact by using predefined transformation definition. The DSL program conforms to the DSL (grammar), while the generated artifact conforms to the platform metamodel. Typically, this is an example of a model-to-model transformation [22]. Alternatively, the generator could directly generate code, a model-to-text transformation.

3 Modeling concerns for parallel computing

In this section, we discuss the motivation for adopting domain-specific languages for mapping parallel algorithms to parallel computing platforms. In Sect. 3.1, we present the requirements for modeling in parallel computing. In Sect. 3.2, we discuss the need for automated support.

3.1 Modeling requirements for parallel computing

Usually, it is not trivial to map the parallel algorithm to the parallel computing platform due to the size and complexity of the algorithm and the platform on which the algorithm is mapped. As such, the problem of mapping a parallel algorithm to parallel computing platform cannot be reduced to the level of programming only. To communicate and analyze the design decisions regarding the mapping process, it is important to define useful modeling and design abstractions. To derive the requirements for modeling in parallel computing, we need to consider the activities that are involved in parallel computing. As shown in Fig. 6, we identify the following important activities with respect to the mapping of parallel algorithm to the parallel computing platform.

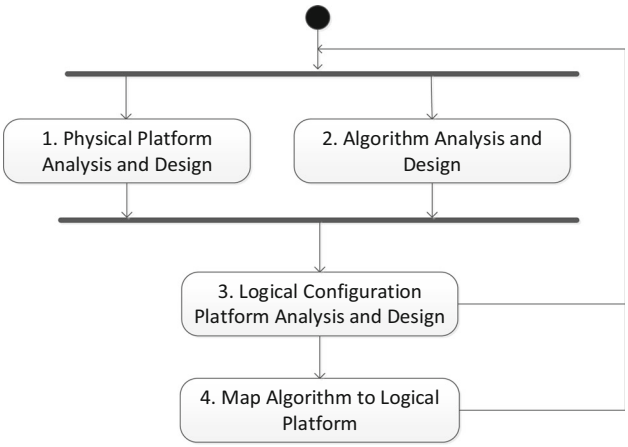


Fig. 6 Lifecycle for mapping parallel algorithm to parallel computing platform

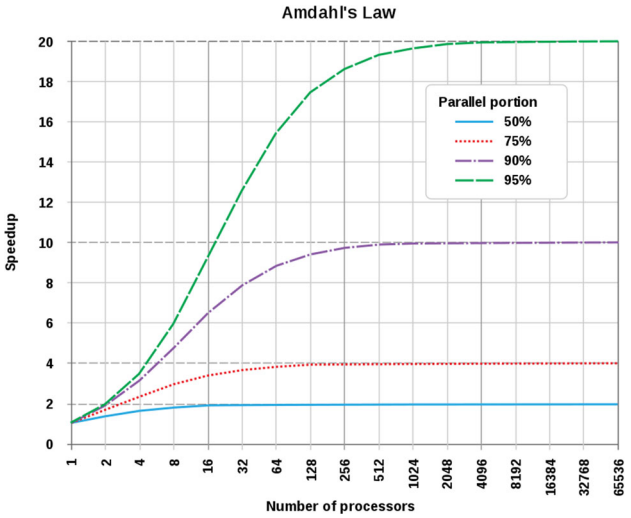


Fig. 7 Amdahl's law illustrated for four different algorithms with different ratios of parallel sections

3.1.1 Modeling the physical computing platform

Usually, the parallel computing platform is large and consists of thousands or tens of thousands of nodes. In current practices, the parallel computing platform is rarely modeled and usually only the scale is provided. Since the parallel computing algorithm needs to be mapped on the physical computing platform, it is important to represent the platform explicitly and reason about the mapping process.

3.1.2 Modeling the decomposition of parallel algorithm

The parallel computing algorithm has a direct impact on the speedup and efficiency of the computation. Each parallel algorithm usually includes both serial and parallel sections. Serial sections can only run on one node, while parallel sections can be allocated to multiple nodes and as such executed in parallel. In this context, the Amdahl's law defines that the serial parts of an algorithm (program) limit the overall speedup of the computation. Amdahl's formula is defined as follows [23]:

$$S_{(n)} = \frac{1}{T_s + T_p/n}$$

where T_s is the time needed to execute the serial part of the algorithm, T_p is the execution time of the parallel part of the algorithm and n is the number of processors.

Many different parallel algorithms exist which have different properties and different ratio of serial and parallel parts. Figure 7 shows, for example, the speedup of four different algorithms with different parallel sections. Since the Amdahl's law has a clear impact on the speedup, it is important to analyze a parallel algorithm and represent the decomposition including parallel and serial sections.

3.1.3 Modeling the logical computing platform

As stated before, the logical computing platform defines the communication structure among the physical nodes. For the same physical configuration platform, many different logical configuration platforms can be defined. Since different logical configuration platforms will require different communication structures among the physical nodes, the communication overhead will be also different. On its turn, this will impact the overall speedup of the computation. The Amdahl's law represents an upper limit of the speedup given a parallel computing algorithm with its parallel code ratio and the number of nodes in the physical configuration. In practice the real speedup will be lower than the predicted value based on Amdahl's formula because of the parallel communication overhead. The challenge as such is to define a logical configuration which is close to the theoretical limit as defined by the Amdahl's formula. Thus, separate from the modeling approach for physical configuration platform, it is also important to provide models for reasoning about the possible logical configuration platforms.

3.1.4 Modeling the mapping of parallel algorithm to the code

Once the logical configuration has been defined, the corresponding code needs to be implemented. Two important issues can be identified here. First of all, the code for the parallel computing algorithm needs to be implemented based on the selected logical configuration. Second, the developed code should be allocated and deployed on the nodes of the parallel computing platform. For developing the code, the decomposition of the algorithm into serial and parallel parts

will be used. Further, the allocation as defined in the logical configuration will be used to realize the deployment over the physical nodes. To communicate about the mapping of the parallel algorithm to the code, it is important to represent this explicitly.

3.2 Need for automated support

As discussed above, the mapping of the parallel 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 hundreds of thousands of nodes providing petascale to exascale level processing power [7]. As a consequence, mapping the parallel algorithm to computing platforms has become intractable for the human parallel computing engineer. With the increased complexity also evolution of the requirements need to be considered. Once the mapping has been realized, in due time the parallel computing platform might need to evolve or change completely, or different algorithms might be required. In that case, the overall mapping process must be redone requiring lots of time and effort.

After the code implementation, we can allocate and deploy the developed code to the nodes of the parallel computing platform. In our example of Fig. 1, 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 easily imagine 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 [7] and likewise the effort to implement the code will be much higher. Because of the size and complexity, implementing the code as such 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.

4 Domain-specific language framework

In the previous sections, we have described the needs for modeling in parallel computing and indicated that DSLs substantially support the realization of these goals. To this end, we describe ParDSL, the domain-specific language framework that integrates four different languages and a common library for specifying physical configuration, algorithm decomposition, logical configuration and algorithm-to-code mapping. Each DSL addresses specific concerns of a particular domain. In the following subsections, we first describe the adopted approach for developing the DSL framework and then discuss each DSL in more detail.

4.1 Approach for developing DSL

It appears that the systematic development of DSLs is not tackled in depth. Most of the DSL research seems to have focused on case studies and experience reports for the development of individual DSLs, design approaches and implementation techniques for DSLs, and the integration of DSLs with other software development approaches, such as programming languages for embedding DSLs, model-driven software development or component-based software development [11].

To address the needs for a systematic development approach for DSLs, Strembeck and Zdun [11] conducted and analyzed many projects in which they built different types of DSLs. From these experiences, they have identified and described the common activities that are needed when engineering a DSL. They distinguish among the following four main activities with their sub-activities:

1. Define the core language model of the DSL

   The goal of this sub-process is the identification and integration of domain abstractions that will form the basis of the DSL. For complex domains, it is recommended to follow a domain analysis method, such as domain-driven design for the identification of domain abstractions [24].

2. Define the DSL language elements' behavior

   The behavior definition of a DSL determines how the language elements of the DSL interact to produce the behavior intended by the DSL designers. For this, language model elements are selected, the required behavior of these elements is modeled, and the DSL's behavior is checked.

3. Define the concrete syntax(es) of the DSL

   Symbols for language model elements are defined together with the DSL production/composition rules. The DSL concrete syntax is defined and checked with respect to correctness and completeness from the point of view of domain experts.

4. Integrate DSL artifacts with the platform/infrastructure

   DSL artifacts are mapped to the platform. For this, all features needed for the platform are identified and implemented and the required DSL-to-platform transformations are defined.

According to Strembeck and Zdun [11], these activities form a micro-process that can be tailored to various influ-

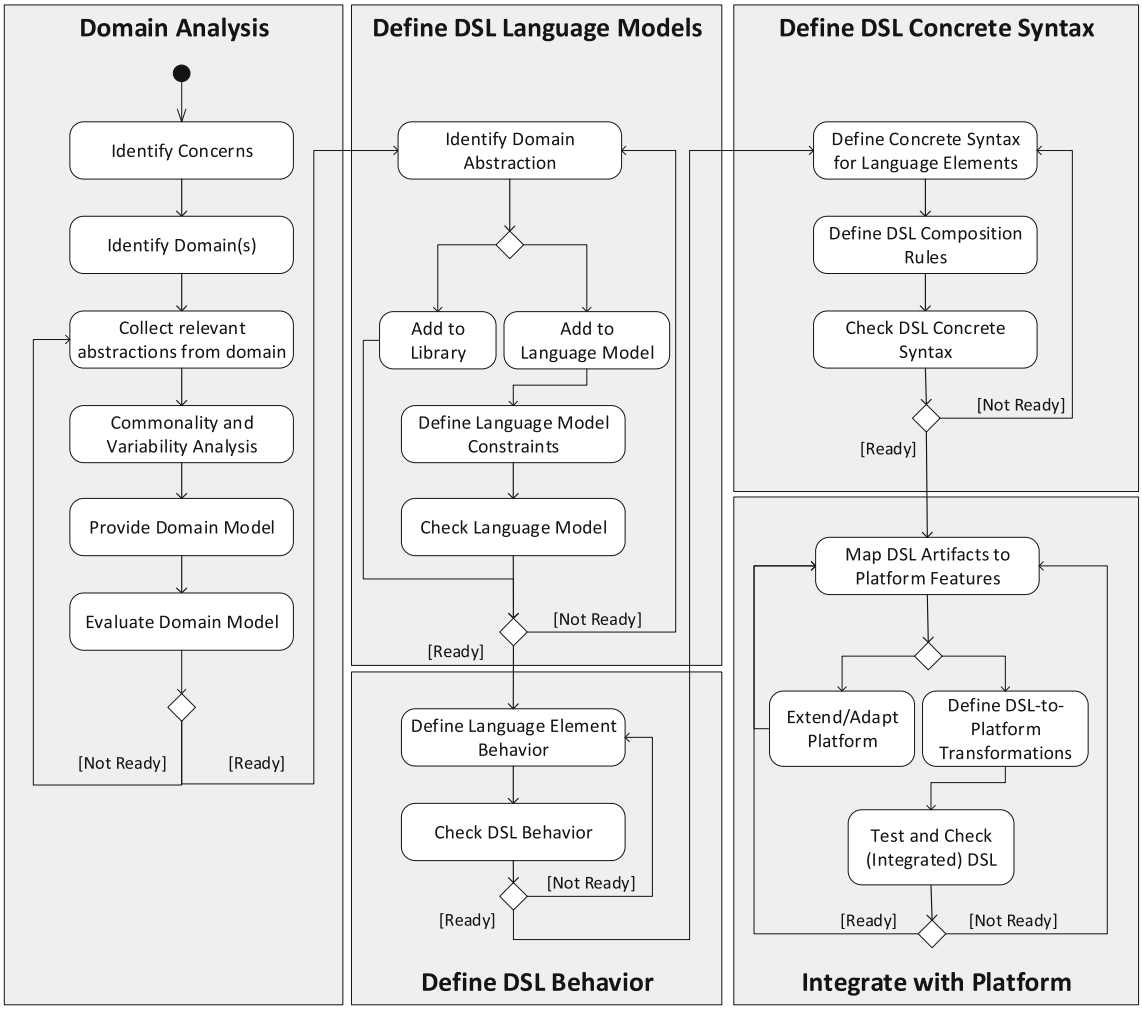


Fig. 8 Approach for developing DSL framework

encing factors of an actual DSL project. In essence, we have tailored and adopted the above steps to develop the DSL framework. The process that we have followed is shown in Fig. 8. The DSL development was an explorative and iterative process that followed after our earlier studies on parallel computing [25, 26].

For deriving the important concerns for mapping parallel tasks to parallel computing platforms, we have carried out a thorough domain analysis process. Domain analysis can be defined as the process of identifying, capturing and organizing domain knowledge about the problem domain with the purpose of making it reusable when creating new systems [27]. Conventional domain analysis methods consist generally of the activities *Domain Scoping* and *Domain Modeling*: *Domain Scoping* identifies the domains of interest, the stakeholders, and their goals, and defines the scope of the domain. *Domain Modeling* is the activity for representing the domain, or the *domain model*. The domain model can be represented in different forms such as object-oriented language, algebraic specifications, rules, conceptual models or a DSL. The scope of the domain that we focus on is shaped by the four different concerns that we have described before. Hence, the domain scope for our purposes included the literature on parallel computing in general, such as for example [8, 28–30]. Once the domain model is ready it can be used to develop the DSLs (*Define DSL Language Models*). Hereby, each domain abstraction is selected and checked whether it will be part of the language or a separate library. In case it is decided to reserve its realization for the library, this will be implemented later. Otherwise it will be part of the DSL. The language model constraints (static semantics) are defined and the overall language model is checked with respect to completeness and correctness. Iterations might be required to enhance and/or complete the language model. The next step is the definition of the DSL behavior, which determines how the language elements of the DSL interact to produce the behavior intended by the DSL designers. Subsequently, the concrete syntax of the DSLs are defined, together with the

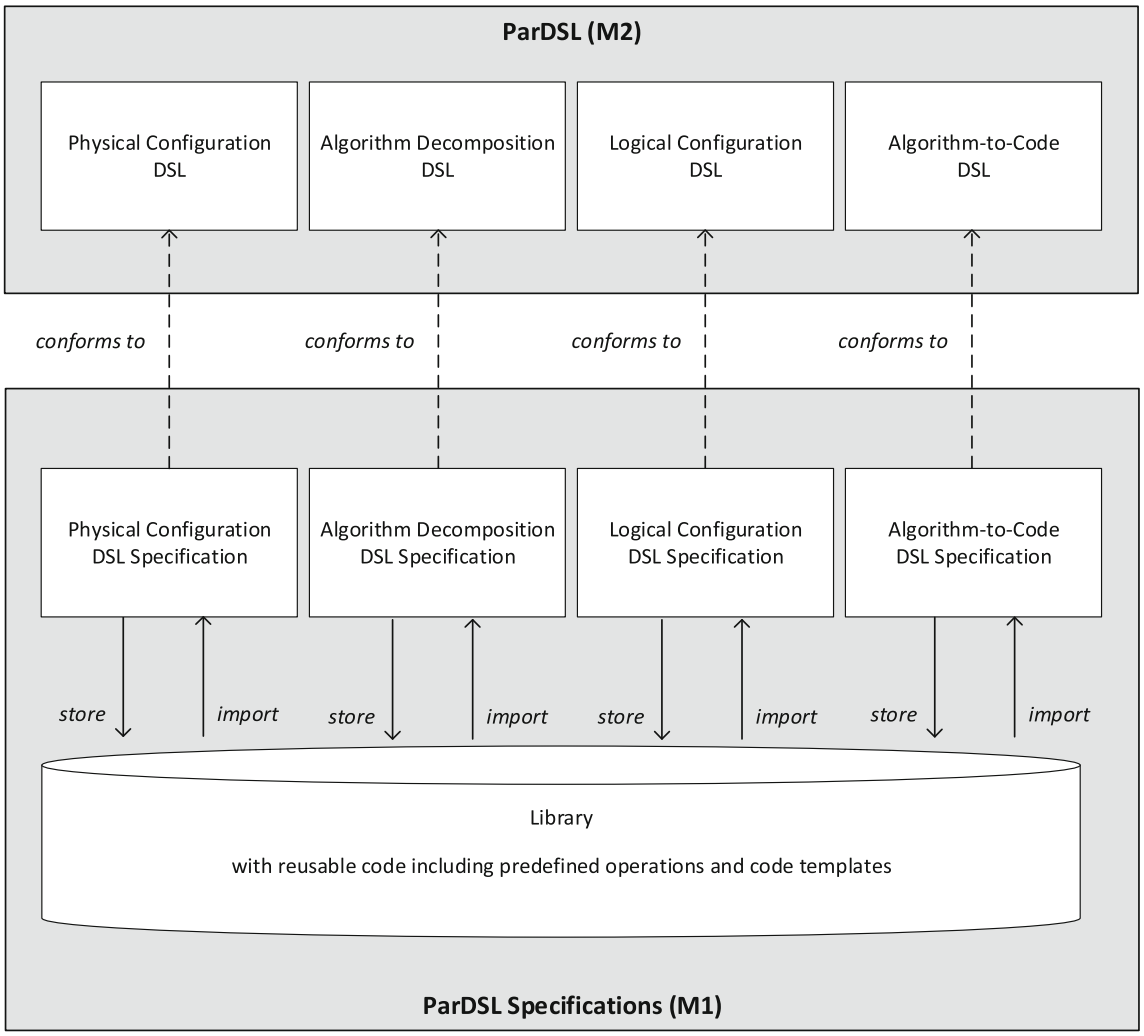


Fig. 9 ParDSL architecture: language framework consisting of 4 core DSLs together with Library of reusable program elements

composition rules and the corresponding checks. The final activity is the definition of the model transformations based on the DSL.

The resulting DSL framework is shown in Fig. 9. As stated before the framework consists of 4 DSLs and a reusable library. In the following subsections we describe each DSL and the library separately.

4.2 Physical configuration DSL

The Physical Configuration DSL is constructed to define the physical parallel computing platform features for mapping the parallel algorithm. The DSL defines the explicit notations for Node, Processing Unit, Network, Memory Bus and Memory. The result of the domain analysis process and the abstract syntax is given in Fig. 10. *Physical Configuration* includes the *Nodes* and the *Network* among nodes of the computing platform. Network has attribute *topology* that represents the physical architecture topology and performance attributes like *bandwidth*, *throughput*, and *latency*. *Node* includes *Processing Units*, *Memory* and *Bus*. Processing Unit has *clock rate* and *precision* attributes. *Bus* has attribute *latency* for delivering data from memory to processing units. Memory has attribute *size* for storage size.

Based on the abstract syntax Fig. 11 shows the provided corresponding grammar specification (left), an example physical configuration specification (right top), and an example physical configuration that is put in library (right bottom).

One of the key challenges in developing a DSL is defining the proper scope of the DSL. We have deliberately chosen to have a generic enough DSL that can cover a broad set of computing platforms. The DSL framework focuses on supporting the mapping of parallel computing algorithms to parallel computing platforms. The provided DSLs support this process and help to find the feasible deployment alternative. In essence for the physical computing platform, the domain experts only need to provide the properties of computing

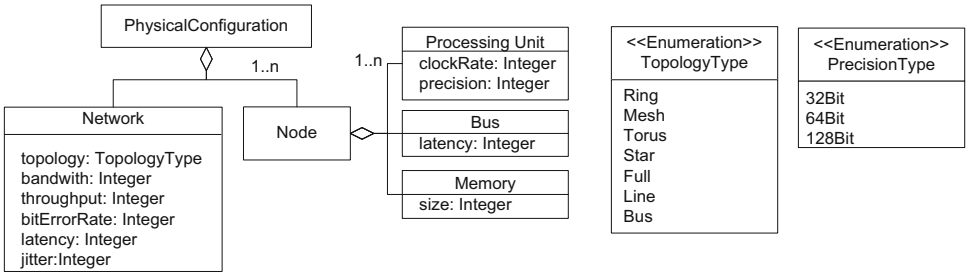


Fig. 10 Physical configuration DSL abstract syntax

DSL (M2)

```
01. grammar PhysicalConfiguration
02. Model:
03.  imports+=ImportType*
04.  packages+=PackageType
05.  types+=TypeDef*
06.  physicalConfigurations+=PhysicalConfiguration*;
07. ImportType:
08.  'import' package=[PackageType] ';';
09. PackageType:
10.  'package' name=ID ';';
11. PhysicalConfiguration:
12.  'physicalconfiguration' name=ID '{'
13.     'nodes' ':' nodes+=[NodeType] '['size=INT']' (','
    nodes+=[NodeType] '['size=INT']')* ';'
14.     'network' ':' network=[NetworkType] ';'
15.  '}';
16. TypeDef:
17.  NodeType | MemoryType | BusType |
18.  ProcessingUnitType | NetworkType;
19. NetworkType:
20.  'network' name=ID '{'
21.     'topology'':' topology=TopologyType ';'
22.     'bandwith'':' bandwith=INT bunit=BwithUnit';'
23.     'throughput'':' throughput=INT ';'
24.     'bitErrorRate'':' berate=INT '%' ';'
25.     'latency'':' latency=INT 'usec' ';'
26.     'jitter'':' jitter=INT ';'
27.  '}';
28. enum TopologyType:
29.  Ring='Ring'|Mesh='Mesh'|Torus='Torus'|
    Star='Star'|
30.  Full='Full'|Line='Line'| Bus='Bus';
31. NodeType:
32.  'node' name=ID '{'
33.     'processingunits'':' pus+=[ProcessingUnitType]
    '['size=INT']' (',' pus+=[ProcessingUnitType]
    '['size=INT']')* ';'
34.     'memory' ':' memory=[MemoryType] ';'
35.     'bus' ':' bus=[BusType] ';'
36.  '}';
37. ProcessingUnitType:
38.  'processingunit' name=ID '{'
39.     'clockRate'':' rate=DOUBLE unit=ClockRateType ';'
40.     'precision'':' precision=PrecisionType ';'
41.  '}';
42. MemoryType:
43.  'memory' name=ID '{'
44.     'size' ':' size+=MemorySize ';'
45.  '}';
46. MemorySize: size=INT unit=MemorySizeUnit;
47. DOUBLE returns EDouble: '-'? INT? '.' INT;
```

Example Specification (M1)

```
01. import PowerPCLibrary;
02. package Cluster;
03. node ClusterHost {
04.   processingunits : PowerPC_450[4];
05.   memory : DDR2_4GB;
06.   bus : PowerPc_Bus;
07. }
08. network ClusterNet {
09.   topology : Torus;
10.   bandwith : 1 GBit;
11.   throughput : 10;
12.   bitErrorRate : 20%;
13.   latency : 500 usec;
14.   jitter : 10;
15. }
16. physicalconfiguration
17.   ClusterComputer {
18.     nodes : ClusterHost[100];
19.     network : ClusterNet;
20. }
```

Example Library Specification (M1)

```
01. package PowerPCLibrary;
02. processingunit PowerPC_450 {
03.   clockRate : 850.00 MHz;
04.   precision : 32Bit;
05. }
06. bus PowerPc_Bus {
07.   latency : 300 usec;
08. }
09. memory DDR2_4GB {
10.   size : 4 GByte;
11. }
```

Fig. 11 Grammar specification of the physical configuration DSL (left), and example specification of physical configuration using the DSL (right)

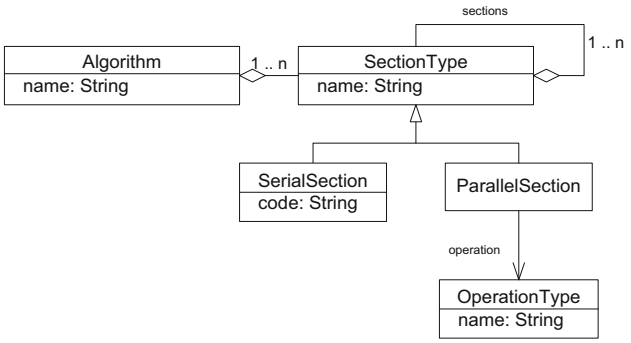


Fig. 12 Algorithm decomposition DSL abstract syntax

nodes and the network configuration. The DSL can be further enriched with more refined computing platform details (e.g., deep memory hierarchies, temporary buffer space and parallel filesystems). This would make the DSL more specific and hence less reusable. If needed though the DSL can be easily extended for this functionality. Furthermore, the specific details or customizations of the DSL can be realized using generators that generate the platform specific concerns. However, it should be noted that the DSL framework has been designed as a coherent set of DSLs that are aligned to each other. A change to the Physical Configuration DSL or any other DSL in the framework would require checking and enhancing the other DSLs in the coherent language framework.

4.3 Algorithm decomposition DSL

To implement a parallel algorithm, it is necessary to decompose the algorithm into separate sections and define which sections are serial and which can be run in parallel. Further, each section of an algorithm realizes an operation, which is a reusable abstraction of a set of instructions. For serial sections the operation can be custom to the algorithm. Parallel sections typically include common operations that can be allocated and run on parallel on different nodes. Hereby, we can identify for example the primitive operations *Scatter* for distributing data to other nodes, *Gather* for collecting data from nodes, *Broadcast* for broadcasting data to other nodes, etc. Given these abstractions of serial and parallel sections, in principle we could thus decompose each given algorithm into these elements. Figure 12 shows the corresponding Algorithm Decomposition DSL abstract syntax. Figure 13 shows

DSL (M2)

```
grammar
com.arkin.parallel.dsl.AlgorithmDecomposition with
org.eclipse.xtext.common.Terminals
generate algorithmDecomposition
"http://www.arkin.com/parallel/dsl/AlgorithmDecompos
ition"
Model:
 imports+=ImportType*
 packages+=PackageType
 sections+=SectionType*
 operations+=OperationType*
 algorithms+=Algorithm;
ImportType:'import' package=PackageType ';';
PackageType:'package' name=ID ';';
Algorithm:
 'algorithm' name=ID '{'
   sections+=SectionType*
 '}';
SectionType:
 Section | ParallelSection | SerialSection;
SectionBase: 'section' name=ID;
Section:
 SectionBase '{'
   sections+=SectionType*
 '}';
ParallelSection:
 'parallel' SectionBase '{'
   sections+=SectionType*
   opers+=Operation*
 '}';
SerialSection:
 'serial' SectionBase '{'
   'code' '{'
     text+=STRING
   '}'
 '}';
Operation: 'operation' operation+=OperationType ';';
OperationType: 'operation' name=ID ';';
```

Example Specification (M1)

```
import ParallelSecLib;
package AllPair;
algorithm AllPair
{
 parallel section Gather
 parallel section Exchange
 parallel section Scatter
 serial section Compute {
   code {
     "//compute pair force"
   }
 }
}
```

Example Library Specification (M1)

```
package ParallelSecLib;
operation Gather;
operation Scatter;
operation Collect;
operation Distribute;
operation Exchange;
operation Broadcast;
parallel section Gather {
   operation Gather;
}
parallel section Exchange {
   operation Exchange;
}
parallel section Scatter {
   operation Scatter;
 }
```

Fig. 13 Algorithm decomposition DSL and example specification

Table 1 Common operations for parallel algorithms as included in the library

Operation	Description
Gather	Each dominating node gets data from its dominated nodes in a pattern
Scatter	Each dominating node sends data to its dominated nodes in a pattern
Collect	A dominating node gets data from other dominating nodes
Distribute	A dominating node sends data to other dominating nodes
Exchange	All dominating nodes exchange data with each other dominating nodes
Broadcast	A dominating node sends data to all other nodes. In general, the broadcast operation consists of distribute and scatter operations
Serial	Serial code block that runs on a single node

the grammar specification based on the abstract syntax and an example specification for the matrix multiplication algorithm.

In the abstract syntax, *Algorithm* consists of *Sections* which are either a *Serial Section* or a *Parallel Section*. The parallel sections are related to an Operation that is an abstraction of well-known operations (such as, for example, *gather* and *scatter*). An analysis of the parallel algorithms in the literature shows that these use abstract well-known operations [10] which can be used to describe the parallel algorithm. In principle, we could define these operations also as elements and thus keywords in the provided DSL. However, we have chosen to provide succinct DSLs that can be easy to learn and reuse. The various set of operations that are reusable will be typically defined in the separate library. Table 1 shows, for example, six operations that form part of different parallel algorithms. These operations are predefined, implemented and stored in the library so that these can be used when describing existing or new parallel algorithms. Table 2 shows an example set of parallel algorithms with the implemented operations of Table 1. For example, the parallel algorithm *Matrix Transpose* can be defined as a combination of *Gather, Exchange* and *Scatter* operations. The *Matrix Multiply* algorithm consists of the operation *Distribute, Serial, Collect* and *Serial.* Similar to these algorithms other parallel algorithms can be in essence defined as consisting of these predefined operations.

4.4 Logical configuration DSL

Figure 14 shows the abstract syntax of the logical configuration DSL for modeling the logical configuration of the parallel computing architecture. *Logical Configuration* includes a set of *Tiles* which are either a *Pattern* or a *Core*.

Table 2 Example parallel algorithms expressed using abstract operations

Algorithm	Operations
Matrix Transpose	Gather; Exchange; Scatter
Matrix Multiply	Distribute; Serial (multiply); Collect; Serial (sum)
Array Increment	Distribute; Collect; Serial (increment)
Complete Exchange	Scatter; Exchange; Gather
Map Reduce	Scatter; Serial (custom); Gather

Tile is an abstract class with the index values (i and j) within the container pattern. *Core* includes the actual index values for the tile. *Pattern*, which implements *Operation* defined in the library, consists of tiles recursively where some of the tiles are labeled as *dominating* nodes. The recursive construction of patterns with tiles derives larger logical configurations with respect to the scaling process we refer to our earlier paper [25]. For very large configurations such as in exascale computing it is not feasible to draw this on the same scale. Instead we define the configuration as consisting of a set of tiles which are used to generate the actual logical configuration. *Pattern* also includes the *Communication* definitions. The speedup and efficiency values are calculated according to the communication definitions with respect to the tiling. Figure 15 shows the grammar specification based on the abstract syntax, and an example specification.

4.5 Algorithm-to-code DSL

Once the algorithm decomposition and the corresponding logical configuration plan have been defined, the implementation of the algorithm can be started. In fact, the algorithm-to-code DSL is defined as a combination of algorithm decomposition and logical configuration DSLs. The *Parallel Section* class includes the pattern implementations with respect to the logical configuration. For *Serial Section*, the code block is implemented manually and inserted into the section after generation of the parallel section patterns. Figure 16 shows the abstract syntax for the Algorithm-to-Code DSL definitions for the algorithm-to-code DSL. Figure 17 shows the grammar specification together with the example specification.

5 Model transformations using ParDSL

As stated before the adoption of DSLs has several benefits including support for communication among domain experts, higher productivity, support for quality validation and verification, and automation. In essence each DSL can be used to address one of the four corresponding concerns and likewise

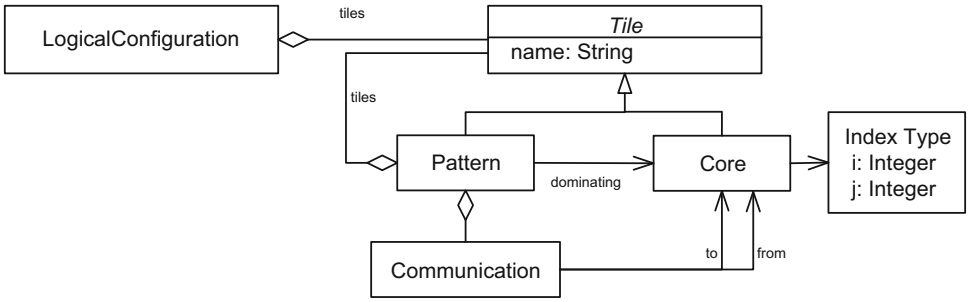


Fig. 14 Logical configuration DSL abstract syntax

DSL (M2)

```
01. grammar
    com.arkin.parallel.dsl.LogicalConfiguration
    with org.eclipse.xtext.common.Terminals
01. generate logicalConfiguration
    "http://www.arkin.com/parallel/dsl/LogicalCon
    figuration"
02. Model:
03.  imports+=ImportType*
04.  packages+=PackageType
05.  logicalconfigurations+=LogicalConfiguration*
06. ;
07. ImportType:'import' package=PackageType ';';
08. PackageType:'package' name=ID ';';
09. LogicalConfiguration:
10.  'logicalconfiguration' name=ID '{'
11.    tiles+=Tile*
12.  '}';
13. Tile:
14.  (many?='dominating')? tile=(Pattern|Core);
15. Pattern:
16.  'pattern' name=ID '{'
17.  index+=Index
18.  tiles+=Tile*
19.  comms+=Communication*
20.  implements+=OperationType
21.  '}';
22. Core: 'core' index=Index;
23. Index: 'index' i=INT j=INT;
24. Communication:
25.  'communication' 'from' from=Core 'to'
    to=Core;
26. OperationType: 'operation' name=ID ';';
```

Example Specification (M1)

```
01. import ParallelLibPatterns;
02. package AllPair;
03. logicalconfiguration AllPair
04. {
05.   pattern Size2_Gather
06.   pattern Size2_Exchange
07.   pattern Size2_Scatter
08. }
```

Example Library Specification (M1)

```
01. package ParallelLibPatterns;
02. pattern Size2_Scatter {
03. index 0 0
04. dominating core index 0 0
05. core index 0 1
06. core index 1 0
07. core index 1 1
08. communication from core index 0 0
    to core index 0 1
09. communication from core index 0 0
    to core index 1 0
10. communication from core index 0 0
    to core index 1 1
11. operation Distribute
12. }
```

Fig. 15 Logical configuration DSL abstract syntax and EOL grammar specification

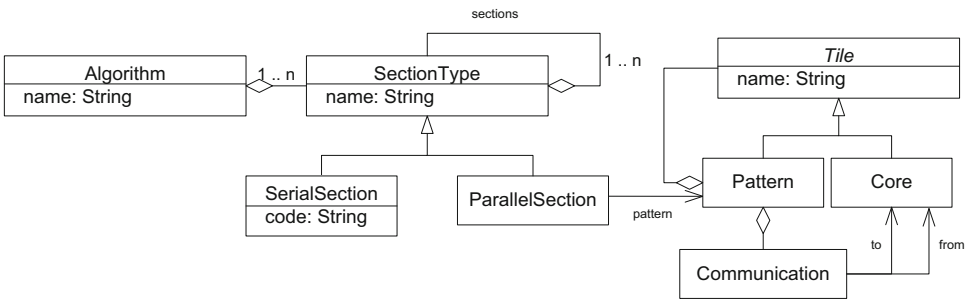


Fig. 16 Algorithm-to-code DSL abstract syntax

provide the benefits that a DSL in general offers. However, since the four DSLs have been developed as a coherent set within the DSL framework the most benefit will be achieved in case these are used in combination. In addition, it should be noted that we have shown the manual usage of the DSLs to specify the four concerns and did not use the full benefit of the DSLs enabling automated processing. For this we will use model transformation techniques as it has been defined in the model-driven development community [15, 16]. Figure 18 shows the process for the integrated usage of the DSLs

DSL (M2)

```
grammar com.arkin.parallel.dsl.AlgorithmToCode with org.eclipse.xtext.common.Terminals
generate algorithmToCode "http://www.arkin.com/parallel/dsl/AlgorithmToCode"
Model:
 imports+=ImportType*
 packages+=PackageType
 algorithms+=Algorithm
;
ImportType:'import' package=PackageType ';';
PackageType:'package' name=ID ';';
Algorithm:
 'algorithm' name=ID '{'
    sections+=SectionType*
 '}';
SectionType:
 Section | ParallelSection | SerialSection;
SectionBase: 'section' name=ID;
Section:
 SectionBase '{'
    sections+=SectionType*
 '}';
ParallelSection:
 'parallel' SectionBase '{'
    sections+=SectionType*
    parallelsections+=Pattern*
 '}';
SerialSection:
 'serial' SectionBase '{'
    'code' '{'
       text+=STRING
    '}'
 '}';
Tile:
 (many?='dominating')? tile=(Pattern|Core);
Pattern:
 'pattern' name=ID '{'
 index+=Index
 tiles+=Tile*
 comms+=Communication*
 implements+=OperationType
 '}';
Core: 'core' index=Index;
Index: 'index' i=INT j=INT;
Communication:
 'communication' 'from' from=Core 'to' to=Core;
OperationType:'operation' name=ID ';';
```

Example Specification (M1)

```
import ParallelSecLib
package AllPair;
algorithm AllPair
{
 parallel section Gather {
    index 0 0
    dominating core index 0 0
    core index 0 1
    core index 1 0
    core index 1 1
    communication from core index 0 1 to core index 0 0
    communication from core index 1 0 to core index 0 0
    communication from core index 1 1 to core index 0 0
    operation Gather
 }
 parallel section Exchange {
    ...
 }
 parallel section Scatter {
    ...
 }
 serial section Compute {
    code {
       "//compute pair force"
    }
 }
}
```

Fig. 17 Algorithm-to-code DSL abstract syntax

and the adoption of model transformations for supporting the selection of feasible parallel algorithm deployments.

In the figure, three different actors have been defined including system engineer, parallel algorithm engineer, and domain engineer. These actors represent roles and could be played by one or multiple physical persons. The *system engineer* is the person who will be using the physical configuration DSL to describe the physical configuration of the computing platform. The *parallel algorithm engineer* is an expert in analyzing a parallel algorithm and describing it using the Algorithm Decomposition DSL. Finally, the *domain engineer* is the person who can define reusable programming elements to support the overall process. In the process of Fig. 18, these are the initiating manual steps for defining the feasible deployments. The developed models by the three stakeholders are in fact relatively independent and the transition overhead is minimized. However, in case the parallel algorithm engineer requires additional reusable patterns, rules or templates, then the domain engineer needs to enhance the reusable assets with the required patterns, rules or templates.

The subsequent activities of the process are automated and use a set of model-to-model and model-to-text transformations using the corresponding generators, including *Logical Configuration Plan Generator*, *Algorithm-to-Code Model Generator* and *Code Generator*. These support the automated generation of the logical configuration plan, algorithm to code model, and the code. We explain each of these steps in more detail in the following subsections.

5.1 Logical configuration plan generator

The Logical Configuration Plan Generator takes as input the Physical Configuration, Algorithm Decomposition Model

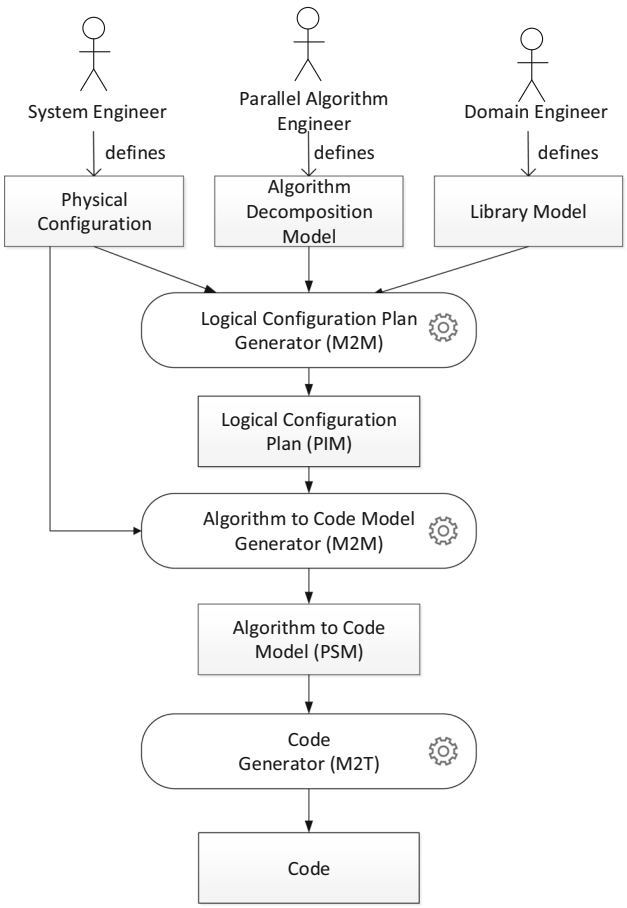


Fig. 18 Domain-specific language framework transformation chain

and Library Model to generate the Logical Configuration Plan. The transformation is carried out in two steps. First, the Algorithm Decomposition Model is merged with the Physical Configuration to generate an algorithm model which includes the physical computing platform features like number of processing units and dimension size of the platform. In the second step, the logical configuration is generated from the algorithm model by using the operation patterns defined in the library model.

The logical configuration plan defines the number of processors and the required primitive tiles and communication patterns that will be selected from the reusable library. The primitive tiles and communication patterns are selected based on the *scale factors* that are calculated using the logical configuration size. The scale factor is the ratio of a logical configuration size to another logical configuration size. For example, a 6×6 logical configuration has a scale factor of 3 to a 2×2 logical configuration. Hereby, we can construct a 6×6 logical configuration using a 3×3 logical configuration whereby each node consists of a 2×2 logical configuration. To calculate all the scale factors of a logical configuration, we adopt *prime factorization*. Prime factorization is the decomposition of a composite number into smaller primitive numbers. The primitive tiles with the primitive size numbers can be scaled to larger logical configuration by using prime factors as scale factors. For example, if we have a 12×12 torus topology, the prime factors of 12 are 2, 2 and 3. As such, we can use a 3×3, and two 2×2 primitive tiles to construct the entire logical topology.

The pseudo code for the logical configuration plan generator is shown in Fig. 19. To generate the logical configuration plan, each parallel section is processed through transformation steps. First the prime factors are calculated to get the scale factors of the physical configuration (line 4). For each factor the patterns which can be used for the given operation of the section are found (line 7) from the library. According to the scaling strategy, the logical configuration patterns are created bottom up or top down (lines 8–13). After selecting the scaling strategy, the pattern is scaled to the physical configuration size (line 14), and then the communications are set for the scaled pattern (line 15).

5.2 Algorithm to code model generator

After the generation of the logical configuration, the algorithm to code model which includes the implementation for each algorithm section must be generated. The parallel sections are generated with respect to the logical configuration plan. The serial sections are implemented manually by the parallel algorithm engineer. Thus, this step of the transformation step can be considered as a semi-automatic transformation process.

Figure 20 shows the pseudo code for algorithm to code model generator. The transformation steps are done for each section of the algorithm. If the algorithm section is parallel, the logical configuration pattern is transformed into the algorithm code pattern (lines 4–6). The structural elements of the algorithm code pattern are correlated to the logical configuration plan pattern. If the algorithm section is serial, the manual code parts are inserted to the serial section code part (lines 8–10).

5.3 Code generator

Code generator generates the target source code of the parallel algorithm from the algorithm-to-code model. This transformation is a model-to-text transformation, which uses code templates. For different platforms, we can define different code templates. We have used the Xtext template language to generate the source codes for the parallel algorithms. The code is generated according to the patterns defined in the DSL framework using loop structures and the code parts are written according to the targeted platform. The coding effort as such may change according to the target

```
1.  function Logical Configuration Plan Generator
2.    for each parallel section in algorithm sections
3.    do
4.      Calculate Prime Factors ()
5.      for each factor
6.      do
7.        Find pattern for section operation
8.        if scaling = UP then
9.          create pattern bottom up
10.       end
11.       if scaling = DOWN then
12.         create pattern top down
13.       end
14.       Scale pattern
15.       Set communications
16.     done
17.   done
```

Fig. 19 Pseudo code for the logical configuration plan generator

```
1.  function Algorithm to Code Generator
2.    for each section in algorithm
3.    do
4.      if section is type of parallel then
5.        transform logical configuration pattern
6.          to algorithm code pattern
7.      end
8.      if section is type of serial then
9.        insert serial section code part
10.     end
11.   done
```

Fig. 20 Pseudo code for the algorithm to code model generator

platform; however, the required effort is less than manually writing the code.

In this article, we adopt the MPI programming model as an example. We could have used a different programming model; this would only require changing the implementation of the code templates.

The template for model-to-text transformation rules to generate MPI code is depicted in Fig. 21. The template uses two template operations including section code generation (lines 13–16) and pattern code generation (lines 17–29). The main template part includes initial definitions (lines 1–3) and MPI initializations (line 4). After the initializations, for each section the section code generation operation is called (lines 6–9). The section code generation operation calls pattern code generation operation. Within the pattern code generation operation, the codes for data communications is generated using MPI send and receive methods (lines 19–26). The serial code parts for the tiles are inserted in the final part of the template (lines 27–29).

6 Implementation and toolset

Each DSL has two basic actors, the language engineer who develops the DSL, and the language user who uses the developed DSL. In this section, we describe the toolset for both the language engineer and language user. In Sect. 5.1, we present the adopted tools for developing ParDSL by the language engineer. Section 5.2 describes the tools that are used by the language users, i.e., domain experts, to derive feasible deployment alternatives.

6.1 Language engineer toolset

For developing the DSL framework, we have used the Eclipse framework [31] including the Xtext Language Generator [32], Emfatic [33] and EuGENia [34] tools. Xtext Language Generator is used to generate textual domain specific language. Emfatic is used to define the visual syntax definitions, and EuGENia is used to generate visual syntax using models from metamodels. Figure 22 shows the activities of the Language Engineer using the Language Engineering Toolset to create the Algorithm to Platform Mapping Toolset. To develop a DSL, the language engineer writes first the grammar using the Xtext language tool. The grammar definitions (see Sect. 4) form an input for the Xtext Language Generator which extracts the ecore models to be used for generating the visual syntax. In parallel, the Xtext Language Generator generates the textual DSL plug-in for the language users toolset. The language engineer also defines the visual syntax using the Emfatic tool [33]. Emfatic is a text editor supporting navigation, editing and conversion of Ecore models, using a compact and human-readable syntax [33]. Emfatic updates the ecore models using the required tag definitions for Graphical Modeling Framework (GMF) [35]. The updated ecore models are provided to EuGENia [34] and EuGENia generates the visual syntax plug-in. EuGENia is a tool that automatically generates models needed to implement a GMF

```
1.  #include "mpi.h"
2.  int main(int argc, char *argv[])
3.  {
4.  // MPI Initializations
5.  // GENERATE CODE FOR EACH SECTION
6.  [% for(section in algorithm.sections) { %]
7.  [%"Section ".print();section.name.println();%]
8.  [%=section.code(algorithm.size)%]
9.  [% } %]
10. MPI_Finalize();
11. return 0;
12. }
13. // SECTION CODE GENERATION
14. [%@template operation Section code(size:Integer) { %]
15. [%for(pattern in section.patterns) {%]
16. [%=pattern.code(size)%][%}%]
17. // PATTERN CODE GENERATION
18. [% @template operation Pattern code(size:Integer) { %]
19. [%for(commu in self.comm) {%]
20. [%var fromRank = commu.fromCore.rank(size); %]
21. [%var toRank = commu.toCore.rank(size); %]
22. if(rank == [%=fromRank%])
23. MPI_Isend(buffer, 1, MPI_DOUBLE, [%=toRank%], [%=fromRank%], MPI_COMM_WORLD, &request);
24. if(rank == [%=toRank%])
25. MPI_Irecv(buffer, 1, MPI_DOUBLE, [%=fromRank%], MPI_ANY_TAG, MPI_COMM_WORLD, &request);
26. [%}%]
27. [%for(tile in self.tiles) {%]
28. [%if(tile.isTypeOf(algorithmtocode!Pattern)) {%]
29. [%=tile.code(size)%][%}%][%}%][%}%]
```

Fig. 21 Part of the code template for generating MPI code

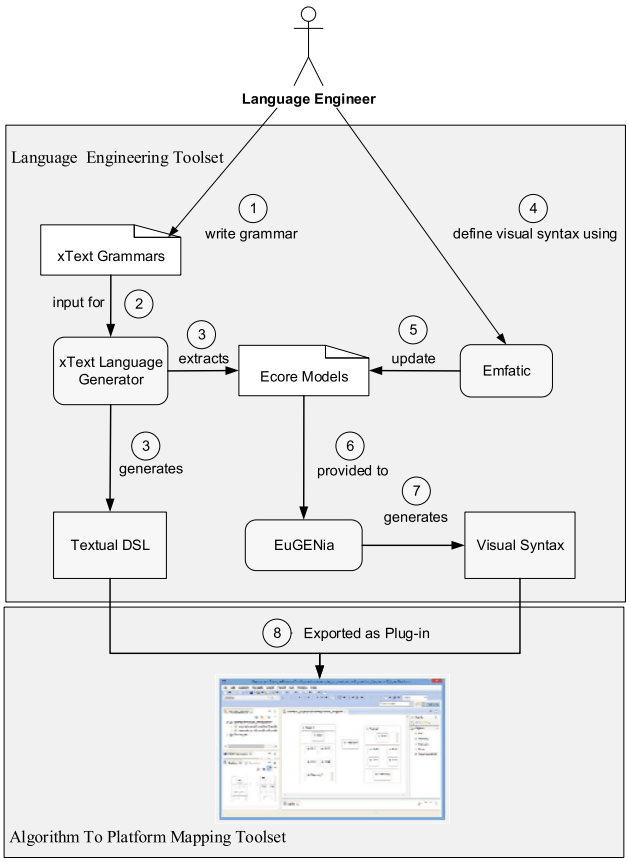


Fig. 22 Activities of the language engineer

editor from a single annotated Ecore metamodel. Finally, both the textual DSL and visual syntax is exported as Eclipse plug-in to be used by the language users.

6.2 Language user toolset

The language user toolset includes the tools for realizing the four concerns including modeling the physical computing platform and modeling the algorithm decomposition. Further it provides tools for generating the logical configuration and the algorithm deployment code. Figure 23 shows the conceptual model of the toolset for the language users. As it can be observed, there are three different language users, that is, the domain engineer, the system engineer and the parallel algorithm engineer. The domain engineer is responsible for defining reusable language elements and storing this in the library using the *Library Definition Tool*. The system engineer is responsible for defining the physical configuration using the *Physical Configuration Editor*. Finally, the parallel algorithm engineer is responsible for defining the algorithm decomposition using the *Algorithm Decomposition Editor*. The output of these activities, that is, *Library Ecore Model, Physical Configuration Model* and *Algorithm Decomposition Model* are provided as an input to the *Transformation Chain* which generates the required code. In the following, we explain each tool in more detail.

6.2.1 Library definition toolset

Figure 24 shows the *Library Definition Toolset* including four panels: (1) Primitive Tile Definition Panel, (2) Pattern Definition Panel, (3) Operation Definition Panel and (4) Review Panel. The *Primitive Tile Definition Panel* shows the primitive tile that is being defined or selected from the library. The *Pattern Definition Panel* shows the communication patterns defined for each primitive tile. The *Operation Definition Panel* shows the adopted operations for the selected primitive

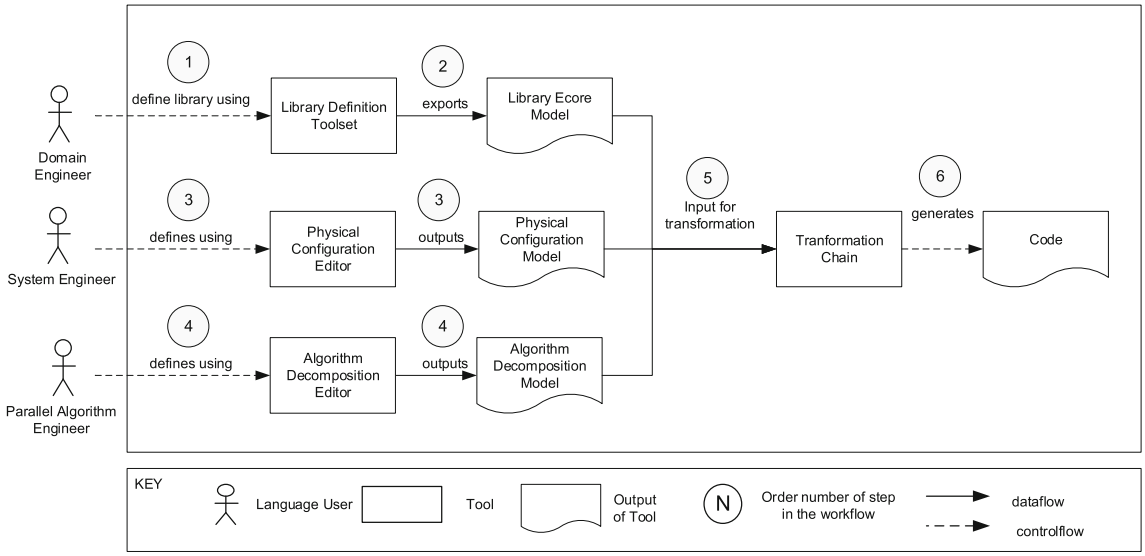


Fig. 23 Conceptual model of the tools for realizing ParDSL

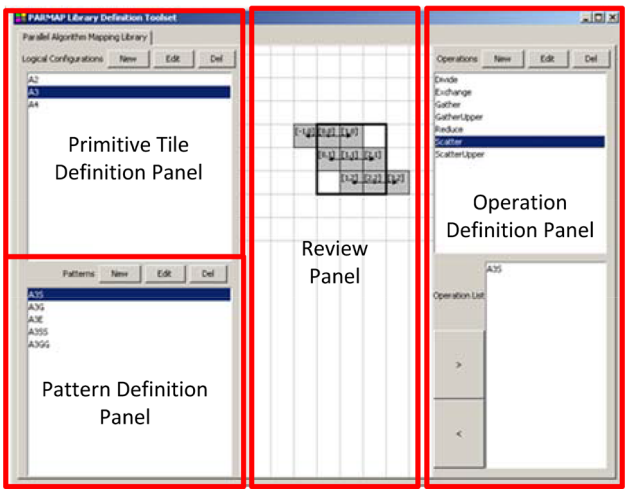


Fig. 24 Library definition toolset

tile and at the bottom the assigned communication pattern. Finally, the *Review Panel* shows the visual representation of either the selected primitive tile or selected communication pattern. The example shows the selection of the primitive tile named "A3," for which the "A3E" communication pattern has been selected. The Review Panel shows the visual representation of the A3E communication pattern. The defined primitive tiles, patterns and operations can be stored in the library so that these can be reused when defining the mapping of parallel algorithm to a configuration.

6.2.2 Physical configuration editor

Figure 25 shows the *Physical Configuration Editor* with its four panels: (1) Project Explorer, (2) Outline Overview, (3) Editor Panel and (4) Palet Panel. Project Explorer shows the defined projects to define different physical configuration models. Outline Overview shows the outline of the editing physical configuration. In the Editor Panel, System Engineer can edit the physical configuration using the DSL structures which can be selected and easily added to the model by drag and drop. The Palet Panel includes these DSL structures. In the example, a physical configuration with two nodes and a network is defined. Each node has 4 processing units and a memory with a bus. The DSL framework that we present is mainly textual, so that the scalability can be handled easily by changing the number of elements in the DSL specification. The tool, however, also provides a visual editor for defining the physical configuration which can be used if found practical.

6.2.3 Algorithm decomposition editor

Algorithm Decomposition Editor is used to define the algorithm decomposition by the parallel algorithm engineer. The Eclipse Ecore Exeed Editor [36] is used to define the algorithm decomposition. Figure 26 shows the Algorithm Decomposition Editor. The figure shows the Matrix Multiply algorithm consists of two main sections. The first section, named as *MultiplyBlock*, includes a parallel section that implements *Scatter* operation in library, and a serial section which a serial multiplication code will be implemented manually. The second section, named as *SumBlock*, includes a parallel section that implements *Gather* operation in library, and a serial section which the results will be summed to generate the result.

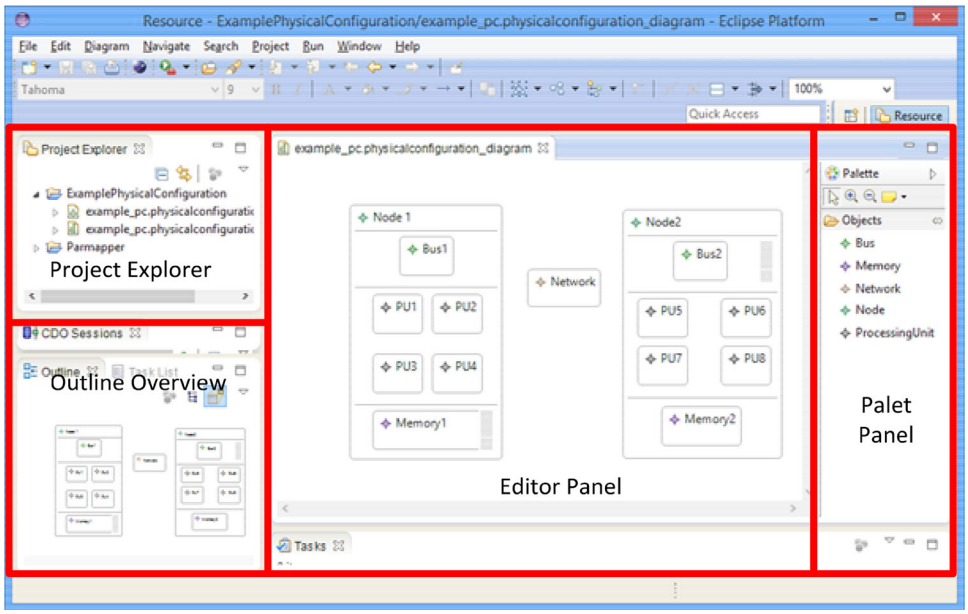


Fig. 25 Physical configuration editor

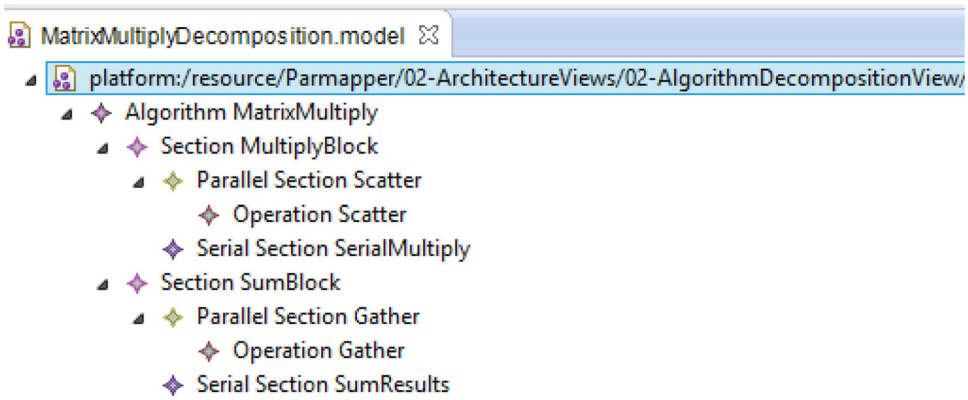


Fig. 26 Algorithm decomposition editor

6.2.4 Transformation toolset

After the definition of the Algorithm Decomposition Model and the Physical Configuration Model, the transformation chain is executed. Transformation Toolset implements the model transformations explained in Sect. 4. The Epsilon Transformation Language (ETL), Epsilon Merging Language (EML) and Epsilon Generation Language (EGL) [37] are used for defining the transformation rules. In the transformation chain, the Logical Configuration Model and Algorithm-To-Code Model are generated in order. The Logical Configuration Model includes the patterns that are used in the parallel sections of the algorithm decomposition model with respect to the physical configuration model. The core definitions and communication definitions are generated for each parallel section operation. In a subsequent transformation, Algorithm-To-Code model is generated including the code implementation for each serial section together with the pattern definitions for each parallel section. This model is ready for the model-to-text transformation to generate the final parallel code.

When we transform the Algorithm-To-Code model to the final code, a C code using MPI [38] is generated as shown in Fig. 27. 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 lines 4 to 8, the code for node 0 is defined which sends the sub-matrices to the other nodes (1, 2, 3). Lines 9–14 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

Fig. 27 Example parallel code of the matrix multiplication algorithm code

```
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();}
```

node 1 (code for node 2 and 3 is not shown). Line 23–25 defines again the collection of the results in node 0. Line 27 defines again a barrier to complete this process. Finally, in line 28–33 the results are summed in node 0 to compute the resulting matrix C.

7 Evaluation of the DSL framework

In our earlier studies, we have presented the problem of the deployment of parallel algorithms to parallel computing platforms [25, 26]. Hereby, we have evaluated several algorithms to execute on multicore computer, a grid simulation environment and a real-world grid platform. We have used the Turkish Science e-Infrastructure (TRUBA) High Performance and Grid Computing platform. We have run our algorithms on 16 nodes which has Intel Xeon i5-2690 processors. The physical platform is configured with a total 64 cores using Simple Linux Utility for Resource Management (SLURM). The logical configuration alternatives of the algorithms are generated using the physical configuration size of 64 cores (8×8). The results of the measured values for evaluation of four different algorithms are shown in Fig. 28. The implementation codes are generated using the transformation rules explained in Sect. 5. To achieve the results, we have run the program 10 times for each alternative and took the mean value of these runs [26]. The implementation codes are verified to run on such a grid system, but still it is needed to evaluate for larger physical configuration sizes. For this reason, we have also created a simulation environment using SimGrid [19] and verified the generated codes.

The focus of these studies was on the allocation problem itself, and we did not introduce any domain specific language. Thereby we have evaluated our approach with respect to the model transformations and effectiveness of the derived feasible deployment alternatives. These studies showed that the presented approach was feasible and is beneficial for deriving feasible deployment alternatives. In this article, we have explicitly focused on the design and development of DSLs and the overall integration of these languages, which we introduced for the first time. The evaluation of the DSL framework requires a separate discussion independent of the overall deployment approach. The main question here is to which extent the DSL framework is effective and supports the end user in describing the required models for solving the algorithm deployment problem.

We have used the matrix multiplication algorithm during the development of ParDSL. In addition, we have applied our approach to three other representative set of parallel computing algorithms to justify the external validity. These algorithms are Array Increment [8], N-Body All Pair [10] and Matrix Transpose [39] algorithms. The pseudo codes for these algorithms are shown in Fig. 29.

Figure 30 shows the example library specifications for physical configuration library, logical configuration library and algorithm library. The specifications for the physical configuration and algorithm decompositions are shown in Fig. 31. The physical configuration and algorithm decom-

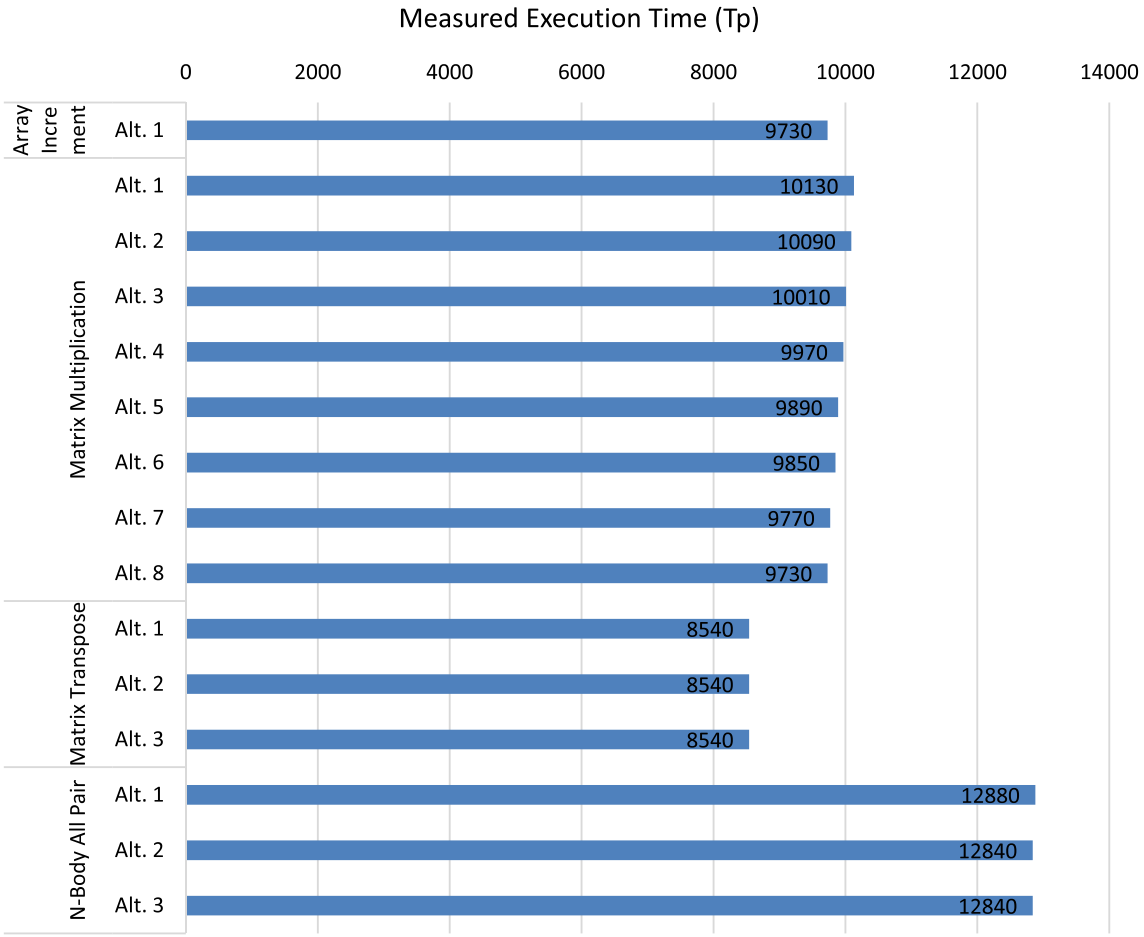


Fig. 28 Measured execution time for four different algorithms with generated alternative implementations

ArrayIncrement

```
1.  Procedure ArrayInc(A[], n):
2.  if n=1 then
3.    *A += 1
4.  else
5.    ArrayInc(A, n/2)
6.    ArrayInc(A+n/2, n)
7.  endif
```

MatrixTranspose

```
1.  Procedure MatrixTranspose(A[n,m]):
2.  for j=0 to m-1
3.    for i=0 to n-1
4.      Copy all blocks of A of
5.        P(n + i, m - j) to T1
6.      Send T1 to P(n + i, m - j)
7.      Receive T2 from P(n - i, m + j)
8.      Copy T2 to A
9.    endfor
10. endfor
```

All-Pair

```
1.  Procedure All-Pair:
2.  for i=1 to n-1
3.       Collect data to the dominating nodes from the dominated nodes
4.  endfor
5.  for i=n-1 down to 1
6.       Exchange the selected data between dominating nodes
7.       Distribute data from dominating nodes to the dominated nodes
8.  endfor
```

Fig. 29 Parallel computing algorithms that were used to test ParDSL

position specifications are used to transform into algorithm to code specifications and real code as shown in the transformation chain of Fig. 18. After the transformations, the algorithm to code DSL specification and real code is generated. The example representation for the array increment algorithm of algorithm to code specification and MPI code is shown in Fig. 32.

Physical Configuration Library

```
01. package PowerPCLibrary;
02.
03. processingunit PowerPC_450 {
04.   clockRate : 850.00 MHz;
05.   precision : 32Bit;
06. }
07. bus PowerPc_Bus {
08.   latency : 300 usec;
09. }
10. memory DDR2_4GB {
11.   size : 4 GByte;
12. }
```

Algorithm Library

```
01.  package algorithmlib;
02.
03.  operation Gather;
04.  operation Scatter;
05.  operation Collect;
06.  operation Distribute;
07.  operation Exchange;
08.  operation Broadcast;
09.  parallel section Gather {
10.     operation Gather;
11.  }
12.  parallel section Scatter {
13.     operation Scatter;
14.  }
15.  parallel section Collect {
16.     operation Collect;
17.  }
18.  parallel section Distribute {
19.     operation Distribute;
20.  }
21.  parallel section Exchange {
22.     operation Exchange;
23.  }
24.  parallel section Broadcast {
25.     operation Broadcast;
26.  }
```

Logical Configuration Library

```
01. package LogicalConfigurationLibrary;
02. pattern Size2_TypeA_Gather {
03.  index 0 0
04.  dominating core index 0 0
05.  core index 0 1
06.  core index 1 0
07.  core index 1 1
08.  communication from core index 1 0 to  core
    index 0 0
09.  communication from core index 1 1 to  core
    index 0 1
10.  operation Gather;
11. }
12. pattern Size2_TypeB_Gather {
13.  ...
14.  operation Gather;
15. }
16. pattern Size2_TypeA_Scatter {
17.  ...
18.  operation Scatter;
19. }
20. pattern Size2_TypeB_Scatter {
21.  ...
22.  operation Scatter;
23. }
24. pattern Size2_TypeA_Collect {
25.  ...
26.  operation Collect;
27. }
28. pattern Size2_TypeB_Collect {
29. ...
30.  operation Collect;
31. }
32. pattern Size2_TypeA_Distribute {
33.  ...
34.  operation Distribute;
35. }
36. pattern Size2_TypeB_Distribute {
37.  ...
38.  operation Distribute;
39. }
40. // Other Size and Type patterns
41. ...
```

Fig. 30 Library specifications (M1)

An important aspect in the evaluation of the DSL framework is also the end-user perspective. However, since the DSL framework is novel it does not have yet a sufficient number of trained users to provide a statistical valid result. As such, we have not been able to conduct formal interviews with questionnaires to assess the effectiveness of the DSLs. Instead, we have looked at theoretical studies in the literature that provide an approach for evaluating DSLs from multiple aspects. There are indeed a number of relevant studies which can be used to assess novel DSLs [14, 18, 40, 41]. We applied the Framework for Qualitative Assessment of DSLs (FQAD) which has been recently published [18]. The framework is based on the ISO/IEC 25010:2011 standard, and defines a set of quality characteristics for evaluating a DSL including: Functional suitability, Usability, Reliability, Maintainability, Productivity, Extensibility, Compatibility, Expressiveness, Reusability, and Integrability. We have considered each perspective and evaluated each DSL of ParDSL.

Functional suitability refers to the degree to which a DSL is fully developed and likewise includes all the necessary functionality. On the other hand, DSL should not include functionality that is not in the domain. According to the multiple case studies that we have carried out, we can claim that ParDSL scores satisfactory on this point. We have been able to specify all the problem specific functionality that is needed for deriving feasible deployment alternatives for all the four parallel algorithms that we have considered (Matrix Multiplication, ArrayIncrement, All-Pair and MatrixTranspose.).

Usability of a DSL is the degree to which a DSL can be used by specified users to achieve specified goals. To analyze usability, we have conducted informal interviews with members at the company including senior engineers and senior faculty members who are working in the domains related to parallel computing. These persons were asked to assess the overall usability of the four DSLs of ParDSL. It was indicated that the languages are relatively simple but expressive and can be relatively easily learned. The users mentioned that the separation of the languages for addressing different concerns was helpful. The current users were using a single language in which they have to cope with the different concerns in the same program which increased the complexity and reduced the understandability and maintainability. The provided DSLs of ParDSL by themselves were found easy to understand, and it was indicated that these could indeed be adopted by different stakeholders. This was considered as an important benefit. Yet, the lack of documentation, besides of

Physical Configuration

```
01.  import PowerPCLibrary;
02.  package Cluster;
03.  node ClusterHost {
04.    processingunits : PowerPC_450[4];
05.    memory : DDR2_4GB;
06.    bus : PowerPc_Bus;
07.  }
08.  network ClusterNet {
09.    topology : Torus;
10.    bandwith : 1 GBit;
11.    throughput : 10;
12.    bitErrorRate : 20%;
13.    latency : 500 usec;
14.    jitter : 10;
15.  }
16.  physicalconfiguration
17.    ClusterComputer {
18.      nodes : ClusterHost[100];
19.      network : ClusterNet;
20.  }
```

Algorithm Decomposition

ArrayIncrement

```
01.  import algorithmlib;
02.  package algorithms;
03.  algorithm ArrayIncrement {
04.    section Distribute {
05.      parallel section Scatter;
06.    }
07.    section Increment {
08.      serial section IncrementSerial {
09.        code {
10.          "A += 1;"
11.        }
12.      }
13.      parallel section Gather;
14.    }
15.  }
```

All-Pair

```
01.  import algorithmlib;
02.  package algorithms;
03.  algorithm All-Pair {
04.    parallel section Gather;
05.    section ExchangeScatter {
06.      parallel section Exchange;
07.      parallel section Scatter;
08.    }
09.  }
```

MatrixTranspose

```
01.  import algorithmlib;
02.  package algorithms;
03.  algorithm MatrixTranspose {
04.    parallel section Gather;
05.    parallel section Scatter;
06.  }
```

Fig. 31 Algorithm implementations (M1)

a corresponding PhD thesis, was found weak. For the shift to a new but better platform, such as ParDSL additional training and planning the organization will be required. We plan to improve the usability in our future work by addressing the required documentation concerns.

Reliability of a DSL is defined as the property of a language that aids producing reliable programs. ParDSL has been developed using the Eclipse environment. Each language has an editor with the default editing features including autocomplete. The languages are defined according to the well-defined principles [16] and the languages have precise semantics. The Eclipse editor provides all the required instruments for debugging and handling code errors. In our earlier studies, we have already provided a full quantitative analysis on the reliability of the allocation process [25, 26].

Extensibility refers to the degree to which a language has mechanisms for adding new features. The language framework with the separate DSLs that we have presented has a low degree of extensibility because we do not provide embedded mechanisms to users for extending the languages with new functionality. In case of novel functionality that requires the change of a DSL then the DSL needs to be redeveloped. This by itself is not a very difficult problem for language developer, but it is a concern for language users. On the other hand, since we believe that the functionality of the language framework is high the need for extendibility will be low. The library that we have provided however has high extendibility because predefined patterns and code can be stored easily in the library thereby extending the overall functionality.

Integrability defines how easily a DSL can be integrated with other languages and modeling tools. We have developed the language framework as a plug-in within the Eclipse framework. The Eclipse platform allows the developer to extend Eclipse applications like the Eclipse IDE with additional functionalities via plug-ins. Integration of the language with other languages can be done using the Exlipse IDE and the overall OSGI component model.

Extensibility defines the degree to which a language has general mechanisms for users to add new features. For adding new features to the languages, the language developer toolset can be used and the language can be easily extended. This is of course also the benefit of the Eclipse and the related tools that we used as a language workbench.

Maintainability defines the degree to which a language is easy to maintain. DSLs can be altered and new concepts and concept extensions can be added. The language framework has been carefully design regarding the separation of concerns principle. This has led to a modular structure of the overall language framework as well as within the DSLs. Maintenance also requires that the languages are understand-

Algorithm To Code

```
01. algorithm ArrayIncrement {
02. section ArrayIncrement {
03. section DistributeBlock {
04. serial section DecomposeSerial {
05. parallel section Scatter {
06.  /***** DSL SECTION Size2_TypeA_Scatter
    START *****/
07. pattern Size2_TypeA_Scatter {
08. index 0 0
09. dominating pattern Size2_TypeA_Scatter
    {
10. index 0 0
11. dominating  core index 0 0
12.  core index 0 1
13.  core index -1 0
14.  core index 1 1
15. }
16. pattern Size2_TypeA_Scatter {
17. ...
18. }
19. pattern Size2_TypeA_Scatter {
20. ...
21. }
22. pattern Size2_TypeA_Scatter {
23. ...
24. }
25.  communication from    core index 0 0
26.  to    core index -2 0
27.  communication from    core index 0 0
28.  to    core index 0 2
29.  communication from    core index 0 0
30.  to    core index 2 2
31. }
32.  /*****  DSL SECTION
    Size2_TypeA_Scatter  STOP ******/
33.
34.  /***** DSL SECTION Size2_TypeA_Scatter
    START *****/
35. ...
36.  /*****  DSL SECTION
    Size2_TypeA_Scatter  STOP ******/
37. }}}
38. section IncrementBlock {
39. serial section IncrementSerial {
40. }
41. parallel section Gather {
42.  /***** DSL SECTION Size2_TypeA_Gather
    START *****/
43. ...
44.  /*****  DSL SECTION Size2_TypeA_Gather
    STOP ******/
45.
46.  /***** DSL SECTION Size2_TypeA_Gather
    START *****/
47. ...
48.  /*****  DSL SECTION Size2_TypeA_Gather
    STOP ******/
49. }}}}
```

Code

```
01. #include <stdio.h>
02. #include "mpi.h"
03. int main(argc, argv)
04.  int argc;
05.  char *argv[]; {
06.  int rank, size;
07.  MPI_Init( &argc, &argv );
08.  MPI_Comm_size( MPI_COMM_WORLD, &size );
09.  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
10.  MPI_Status status;
11.  MPI_Request request;
12.  typedef double DOUBLE;
13.  /*MPI SECTION Size2_TypeA_Scatter  START*/
14.  DOUBLE* Dt1_0_0_0;
15.  if(rank == 0) Dt1_0_0_0 = (DOUBLE
    *)malloc(1*sizeof(DOUBLE));
16.  DOUBLE* Dt1_0_0_2;
17.  if(rank == 8) Dt1_0_0_2 = (DOUBLE
    *)malloc(4*sizeof(DOUBLE));
18.  DOUBLE* Dt1_0_2_0;
19.  if(rank == 2) Dt1_0_2_0 = (DOUBLE
    *)malloc(4*sizeof(DOUBLE));
20.  DOUBLE* Dt1_0_2_2;
21.  if(rank == 10) Dt1_0_2_2 = (DOUBLE
    *)malloc(4*sizeof(DOUBLE));
22.  if(rank == 0) {
23.     MPI_Isend(Dt1_0_0_0, 1, MPI_DOUBLE, 2, 0,
    MPI_COMM_WORLD, &request);
24.  }
25.  if(rank == 2) {
26.     MPI_Recv(Dt1_0_2_0+3, 1, MPI_DOUBLE, 0,
    MPI_ANY_TAG, MPI_COMM_WORLD, &status);
27.  }
28.  if(rank == 0) {
29.     MPI_Isend(Dt1_0_0_0, 1, MPI_DOUBLE, 8, 0,
    MPI_COMM_WORLD, &request);
30.  }
31.  if(rank == 8) {
32.     MPI_Recv(Dt1_0_0_2+0, 1, MPI_DOUBLE, 0,
    MPI_ANY_TAG, MPI_COMM_WORLD, &status);
33.  }
34.  if(rank == 0) {
35.     MPI_Isend(Dt1_0_0_0, 1, MPI_DOUBLE, 10, 0,
    MPI_COMM_WORLD, &request);
36.  }
37.  if(rank == 10) {
38.     MPI_Recv(Dt1_0_2_2+0, 1, MPI_DOUBLE, 0,
    MPI_ANY_TAG, MPI_COMM_WORLD, &status);
39.  }
40.  MPI_Barrier(MPI_COMM_WORLD);
41. /*MPI SECTION Size2_TypeA_Scatter  STOP*/
42. /*MPI SECTION Size2_TypeA_Scatter  START*/
43. ...
44. /*MPI SECTION Size2_TypeA_Scatter  STOP*/
45. /*MPI SECTION Size2_TypeA_Gather  START*/
46. ...
47. /*MPI SECTION Size2_TypeA_Gather  STOP*/
48. /*MPI SECTION Size2_TypeA_Gather  START*/
49. ...
50. /*MPI SECTION Size2_TypeA_Gather  STOP*/
51.  MPI_Finalize();
52.  return 0;
53. }
```

Fig. 32 Algorithm-to-code and generated code DSL implementations for array increment algorithm

able. The DSLs in the language framework directly model the concepts as defined in parallel computing and hence can be easily understood. For maintainability, it is also important that new features can be added easily. This relates also the extensibility criteria that we have discussed above.

Productivity of a DSL refers to the degree to which a language promotes programming productivity. Productivity is a characteristic related to the amount of resources expended by the user to achieve specified goals. Productivity can be increased because of different reasons in ParDSL. First of all, the languages are high-level domain-specific languages that adopt the concepts as defined in the parallel computing domain. Hence, the required physical computing and the parallel computing algorithm can be developed quite quickly. Further, ParDSL provides the generators that automate the development of the logical configurations as well as the code that is required. Deriving the space of possible logical configurations for a larger physical computation platform would take a lot of time and in general simply be not tractable for the human engineer. Productivity in this context is largely achieved. Finally, productivity is also supported since an important aspect of the ParDSL framework is the adoption

of a library that includes reusable patterns and code for modeling the parallel computing system.

Compatibility defines the degree to which a DSL is compatible to the domain and development process. The language framework that we have provided is agnostic to the development process. In fact, after the physical computing platform and the decomposition of the parallel algorithm are defined, most of the process is automated.

Reusability is defined as the degree to which a language constructs can be used in more than one language. The specifications that are defined in ParDSL can be reused in the overall environment. This counts also for the generated artifacts. In addition, the library includes the elements that can be imported and reused in the specifications of the four different languages. If needed, new elements can be added to the library for increasing the number of reusable elements.

Expressiveness is defined as the degree to which a problem solving strategy can be mapped into a program naturally. In other words, expressiveness is the relation between the program and what the programmer has in mind. Hereby it is important that there is a one-to-one correspondence between concepts and their representation in the language. Further the right abstraction level must be selected so as not to use too generic or too specific concepts. The DSLs that have been developed are based on a thorough domain analysis whereby we have modeled each concept in the corresponding metamodels of the languages. We can state that there is indeed a one-to-one correspondence between the identified concepts and the representations in the language. The abstraction level of the elements has been carefully defined to be usable for the developer. The language elements are expressive enough to specify the problem, while too specific details are handled by the generators.

8 Related work

There are several modeling approaches for parallel computing. Some of the approaches adopt existing UML profiles which are used for similar computing platforms like embedded systems, real-time systems. Gamatié et al. [29] define a model-driven design framework, named as GASPARD, that generates code from MARTE models for massively parallel embedded systems. GASPARD supports the engineers for formal verification, simulation and hardware synthesis. Etien et al. [42] presents a model-driven transformation approach to design large-scale parallel systems with large languages by decomposing the complex transformations into smaller transformations within GASPARD framework. They use the separation of concerns principle and define a notation of localized transformations. Yu et al. [21] use GASPARD2 framework to model high-performance embedded systems. They address the correctness of the design using a formal validation tool. Model transformations are used as a bridge between UML models and validation methods. Baklouti et al. [43] defines a solution to generate VHDL parallel configuration from a model using MARTE profile (MARTE2VHDL). Elhaji et al. [44] describe a methodology to model system level Network-on-Chip (NoC) design. They use MARTE to VHDL transformation (MARTE2VHDL) to generate source code. Rodrigues et al. [45] give an approach to generate OpenCL code from models using MARTE profile.

Some of the approaches are defined based on a domain-specific language for parallel computing. Palyart et al. [46] introduces MDE4HPC approach and the High Performance Computing Modeling Language (HPCML) to help in dealing with the complexity of the design by abstracting platform dependent details. The structural and behavioral aspects are defined and the code generation is done with metamodels that conform to these aspects. I a another study Palyart et al. [30] use MDE4HPC approach for high-performance numerical computations. Zhen et al. [47] introduce the graphical model-driven toolset, called ParDT, which enables modeling the parallel application and transforming the model to source code. The toolset supplies an Eclipse-based graphical modeling environment and translator to generate MPI source codes. Mengmeng et al. [48] introduces a visual modeling toolset for parallel application development. The Parallel Application Visual Modling (PAVM) toolset is based on DSL tools and supported by a model checker and code generator. Arora et al. [49] define an approach to develop message passing applications. They developed generative programming tools with a domain-specific language, named Hi-PaL. The source codes are compared to the manually written codes. Jacob et al. [50] discuss a framework PPModel which is designed and implemented as a graphical modeling tool for Eclipse users and a domain-specific language named as tPPModel for non-Eclipse users to facilitate the separation, mapping and execution. Hernandez et al. [51] use domain-specific modeling for automating the development of scientific applications. Pazel and Tibbitts [52] developed a visual development environment tool BladeRunner to generate MPI applications. They provide notations representing higher level abstractions of MPI artifacts. DeVito et al. [53] defined Liszt, a domain-specific language, for generating parallel code that run on clusters, SMP and GPU. The solution is specialized for partial differential equation solving and can transform model to text to generate source code to support portability to heterogeneous computer platforms. SPIRAL [54] is another tool for code generation using model-to-text transformations. SPIRAL introduces a model-driven code generation approach rather than a DSL framework and is specialized for digital signal processing algorithms. Franchetti et al. [55] used the approach to generate discrete Fourier transformation algorithm that run on multicore platforms.

Table 3 Comparison of modeling approaches

Modeling approach	Property			
	Representation format	Adopted viewpoints	Model maturity	Model transformation
GASPARD	Visual	7 viewpoints	Executable	M2M, M2T
MDE4HPC	Visual	4 viewpoints	Executable	M2M, M2T
MARTE2VHDL	Visual	3 viewpoints	Executable	M2M, M2T
MARTE2OpenCL	Visual	7 viewpoints (2 customized)	Executable	M2M, M2T
ParDT	Visual	3 viewpoints	Executable	M2T
PAVM	Visual	1 viewpoint	Executable	M2T
Hi-PaL	Textual	1 viewpoint	Executable	M2M, M2T
PPModel	Visual	1 viewpoint	Executable	M2T
BladeRunner	Visual	1 viewpoint	Executable	M2T
Liszt	Textual	1 viewpoint	Executable	M2T
SPIRAL	Textual	1 viewpoint	Executable	M2T
SIMPAR	Visual	1 viewpoint	Blueprint	M2T
ParDSL	Visual, textual	4 viewpoints	Executable	M2M, M2T

Some approaches adopt UML-based modeling methodologies. Prasad and Gupta [56] have presented a UML-based modeling framework, SIMPAR, for performance evaluation, estimation and prediction of HPC systems. They use UML activity diagrams to model the computation, communication and operations of the parallel applications. They have stated that the UML is insufficient for all HPC modeling needs. Pllana and Fahringer [57] defined a custom UML profile for performance oriented parallel systems. They extended the UML building blocks to describe the important message passing and shared memory paradigms in parallel programming. Fahringer et al. [58] propose the Teuta toolset for UML-based performance modeling of distributed and parallel applications. The tool also supports semantic-based model checking and model traversing for generation of different model representations. The performance predictions are done by using a simulation model.

Table 3 shows the comparison of the modeling approaches that we have described above. Hereby four different properties are distinguished including representation format, adopted viewpoints, model maturity and model transformation. The representation format can be either textual or visual. Adopted viewpoints represent the number of viewpoints that is used in the approach. A viewpoint represents the template for defining models based on particular concerns. Model maturity can be either blueprint or executable. A blueprint is a detailed model that can be used by human engineers to derive more refined models. An executable is a model that can be compiled and used by model compilers. Finally model transformations can be model-to-model or model-to-text transformations.

From the table, we can observe that the approaches except Hi-Pal, Liszt and SPIRAL adopt a visual representation format. For large-scale physical configurations however a textual executable representation would be required. ParDSL provides both a textual and visual representations which can be transformed to each other easily. Hi-PaL, Liszt and SPIRAL provide textual representation format, but only adopt a single viewpoint.

In the ParDSL framework, we have separated the four concerns for mapping parallel algorithms to parallel computing platforms and likewise developed four smaller DSLs. Each DSL can be used by the corresponding stakeholder. The system engineer will be responsible for developing the physical configuration model, the library engineer for developing the DSL library and the transformation modules, and finally, the parallel algorithm engineer is the high-level end user will define the decomposition of the parallel algorithm. Separating the concerns and the roles in the corresponding DSLs substantially increases the usability and the productivity.

Almost all of the approaches focus on executable models to generate source code, some of them also use model-to-model transformations to generate intermediate level models. In this sense, ParDSL is complementary to the other approaches.

9 Conclusion

An important challenge in parallel computing is the mapping of parallel algorithms to parallel computing platforms. This requires several activities such as the analysis of the parallel algorithm, the definition of the logical configuration of

the platform and the implementation and deployment of the algorithm to the computing platform. However, in current parallel computing approaches very often only conceptual and idiosyncratic models are used which fall short in supporting the communication and analysis of the design decisions.

In this article, we present a domain-specific language framework for providing explicit and precise models to support the activities for mapping parallel algorithms to parallel computing platforms. The language framework includes a coherent set of four domain-specific languages each of which focuses on a particular activity of the mapping process. The languages can be used by different stakeholders to model the physical configuration, the decomposition of the parallel algorithm, the logical configuration and the mapping of the algorithm to the code. In essence each DSL can be used to address one of the four corresponding concerns and likewise provide the benefits that a DSL in general offers. The four DSLs however are also a coherent set of DSLs to support the overall purpose of optimizing and easing the deployment of parallel algorithms to parallel computing platforms. As such, the most benefit will be achieved in case the four DSLs are used in combination. For this, we have defined logical configuration plan generator, the algorithm to code model generator and the code generator. With these generators we were able to automate the overall process thereby supporting the selection of a feasible deployment alternative. The framework has been evaluated using four different representative set of parallel algorithms and a qualitative analysis based on a DSL evaluation framework. The presented DSL framework has covered all the important concerns for mapping parallel algorithms to parallel computing platforms and also paved the way for further research.

The DSL framework that we have presented considers algorithms that are general purpose which have not been optimized for a particular computation platform yet. The DSL framework that we present, the provided systematic approach and the corresponding tools help to support the finding of the feasible deployment of the given parallel algorithm to the computation platform. The approach does not directly focus on specific algorithms like the Canon's algorithm which have been specifically developed as an optimized parallel algorithm that can be used on torus topologies. The general matrix multiplication algorithm, for example, does not consider a specific computing platform. With our approach, given the details of the computation platform, the parallel algorithm, can be analyzed and decomposed into serial and parallel sections, which are then mapped to the given physical computation platform. A future research that can be considered is the more advanced analysis of algorithms that also take communication patterns into account. This would be a contribution by its own which we think is complementary to our work.

In the presented approach, the selection of the feasible deployment configuration is defined at the application development time. This is because we generate code that is customized for the provided parallel algorithm and the properties of the parallel computation platform. In the code, the adopted physical configuration parameters are fixed, but if needed the code can be easily regenerated. An interesting issue would be perhaps to decide on the number of nodes or cores to use during run-time. This could be considered complementary to our approach.

In this paper, we did not consider the application domain context but focused on mapping parallel algorithms to parallel computing platforms in particular. The selected algorithms are well known in the parallel computing domain and helped to understand the application of domain-specific languages for mapping parallel algorithms to parallel computing platforms. In the future, we will also focus on the broader application domain context [59, 60] and also enhance our framework to model more precise metrics like hybrid communication environments and power consumption concerns.

This article has applied the DSL development and model-driven development domains to the parallel computing domain to solve a particular problem, that is, the deployment of parallel algorithms to parallel computing platforms. Similar to the development of any other DSL or DSL framework, we have scoped the domain for our purposes and it is hard to conclude that the DSLs in the framework that we have proposed are fixed. Different requirements might have a different impact on each of the DSLs and the overall design of the DSL framework as a coherent set of DSLs. Thus, in this perspective our work could be considered as a selected focus and experience in developing a DSL framework in the parallel computing domain. On the other hand, we believe that the DSL and model-driven framework should be definitely further explored and applied in the parallel computing domain to refine the problems that we have addressed and to elaborate and solve other problems in the parallel computing domain.

References

1. Frank, M.P.: The physical limits of computing. Comput. Sci. Eng. **4**(3), 16–26 (2002)
2. Mellor, S.J., Scott, K., Uhl, A., Weise, D.: MDA Distilled: Principle of Model Driven Architecture. Addison Wesley, Reading (2004)
3. Fowler, M.: Domain-Specific Languages. Addison Wesley, Boston (2010)

4. Fowler, M., Scott, S., Booch, G.: UML Distilled, Object Oriented Series. Addison-Wesley, Reading (1999)
5. Moore, G.E.: Cramming more components onto integrated circuits. Proc. IEEE **86**(1), 82–85 (1998)
6. Aizcorbe, A.M., Kortum, S.S.: Moore's law and the semiconductor industry: a vintage model. Scand. J. Econ. **107**(4), 603–630 (2005)
7. 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., Yelick, K.: Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. DARPA (2008)
8. Talia, D.: Models and trends in parallel programming. Parallel Algorithms Appl. **16**(2), 145–180 (2001)
9. Baransel, C., İmre, K.M.: A parallel implementation of Strassen's matrix multiplication algorithm for wormhole-routed all-port 2D torus networks. J. Supercomput. **62**(1), 486–509 (2012)
10. İmre, K.M., Baransel, C., Artuner, H.: Efficient and scalable routing algorithms for collective communication operations on 2D all-port torus networks. Int. J. Parallel Program. **39**(6), 746–782. Springer Netherlands, ISSN: 0885-7458 (2011)
11. Strembeck, M., Zdun, U.: An approach for the systematic development of domain specific languages. Softw. Pract. Exp. **39**, 1253–1292 (2009)
12. Challenger, M., Kardas, G., Tekinerdogan, B.: A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems. Softw. Qual. J. **24**, 755–795 (2015)
13. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Longman Publishing Co., Inc., Boston (2009)
14. Mernik, M., Heering, J., Sloane, A.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4), 316–344 (2005)
15. Stahl, T., Voelter, M.: Model-Driven Software Development. Addison-Wesley, Boston (2006)
16. Voelter, M.: DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. CreateSpace Independent Publishing Platform (2013)
17. Tekinerdogan, B., Bilir, S., Abatlevi, C.: Integrating platform selection rules in the model driven architecture approach, pp. 159–173. Model Driven Archit., Springer (2005)
18. Kahraman, G., Bilgen, S.: A framework for qualitative assessment of domain-specific languages. SoSym J. **14**, 1505–1526 (2013)
19. Casanova, H., Giersch, A., Legrand, A., Quinson, M., Suter, F.: Versatile, scalable, and accurate simulation of distributed applications and platforms. J. Parallel Distrib. Comput. **74**(10), 2899–2917 (2014)
20. ISO/IEC 42010:2007.: Recommended practice for architectural description of software-intensive systems (ISO/IEC 42010) (2011)
21. Yu, H., Gamatié, A., Rutten, É., Dekeyser, J.L.: Safe design of high-performance embedded systems in an MDE framework. Innov. Syst. Softw. Eng. **4**(3), 215–222 (2008)
22. Bézivin, J.: On the unification power of models. Softw. Syst. Model. **4**(2), 171–188 (2005)
23. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18–20), AFIPS Press, Reston, VA, 1967, pp. 483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California, Solid-State Circuits Newsletter, IEEE, vol. 12, no. 3, pp. 19, 20, Summer (2007)
24. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, Boston (2004)
25. Arkin, E., Tekinerdogan, B., Imre, K.: Model-driven approach for supporting the mapping of parallel algorithms to parallel computing platforms. In: Proceedings of the ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (2013)
26. Arkin, E., Tekinerdogan, B., Imre, K.: Systematic approach for deriving feasible mappings of parallel algorithms to parallel computing platforms. Concurr. Comput. Pract. Exp. **29**, 21 (2016)
27. Arrango, G.: Domain analysis methods. In: Schäfer, W., Prieto-Díaz, R., Matsumoto, M. (eds.) Software Reusability, pp. 17–49. Ellis Horwood, New York (1994)
28. Gustafson, J.L.: Reevaluating Amdahl's law. Commun. ACM **31**(5), 532–533 (1988)
29. Gamatié, A., Le Beux, S., Piel, É., BenAtitallah, R., Etien, A., Marquet, P., Dekeyser, J.-L.: A model-driven design framework for massively parallel embedded systems. ACM Trans. Embed. Comput. Syst. **10**(4), 1–36 (2011)
30. Palyart, M., Ober, I., Lugato, D., Bruel, J.M.: HPCML: a modeling language dedicated to high-performance scientific computing. In: Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and CLoud computing, p. 6. ACM (2012)
31. Eclipse.: http://www.eclipse.org (2015)
32. Xtext.: http://www.eclipse.org/Xtext (2015)
33. Emfatic.: http://www.eclipse.org/emfatic (2015)
34. EuGENia.: http://www.eclipse.org/epsilon/doc/eugenia (2015)
35. Graphical Modeling Framework (GMF).: http://www.eclipse.org/gmf-tooling (2015)
36. Exeed.: http://www.eclipse.org/epsilon/doc/exeed (2015)
37. Epsilon.: http://www.eclipse.org/epsilon (2015)
38. MPI: A Message-Passing Interface Standart, version 1.1. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html (2013)
39. Tekinerdogan, B., Demirli, E.: Evaluation framework for software architecture viewpoint languages. In: Proceedings of Ninth International ACM Sigsoft Conference on the Quality of Software Architectures Conference (QoSA 2013), Vancouver, Canada, pp. 89–98, 17–21 June 2013
40. Kosar, T., Oliveira, N., Mernik, M., Pereira, M.J.V., Crepinsek, M., Cruz, D., Henriques, P.R.: Comparing general-purpose and domain-specific languages: an empirical study. Comput. Sci. Inf. Syst. **7**(2), 247–264 (2010)
41. Oliveira, N., Pereira, M.J.V., Henriques, P.R., Cruz, D.: Domain-specific languages: a theoretical survey. In: Proceedings of INForum, CoRTA2009 (Compilers, Programming Languages, Related Technologies and Applications), Lisbon, Portugal, pp. 35–46 (2009)
42. Etien, A., Muller, A., Legrand, T., Paige, R.F.: Localized model transformations for building large-scale transformations. Softw. Syst. Model. **14**, 1189–1213 (2013)
43. Baklouti, M., Ammar, M., Marquet, P., Abid, M., Dekeyser, J.L.: A model-driven based framework for rapid parallel SoC FPGA prototyping. In: 22nd IEEE International Symposium on Rapid System Prototyping (RSP), pp. 149–155. IEEE (2011)
44. Elhaji, M., Boulet, P., Zitouni, A., Meftali, S., Dekeyser, J.L., Tourki, R.: System level modeling methodology of NoC design from UML-MARTE to VHDL. Des. Autom. Embed. Syst. **16**(4), 161–187 (2012)
45. Rodrigues, A.W.O., Guyomarc'h, F., Dekeyser, J.L.: An MDE approach for automatic code generation from UML/MARTE to OpenCL. Comput. Sci. Eng. **15**(1), 46–55 (2013)
46. Palyart, M., Lugato, D., Ober, I., Bruel, J.M.: MDE4HPC: an approach for using model-driven engineering in high-performance computing. In: Ober, I., Ober, I. (eds.) Proceedings of the 15th International Conference on Integrating System and Software Modeling (SDL'11). Springer (2011)

47. Zhen, X., Jizhou, S., Ce, Y., Huabei, W., Xiaojing, M., Shanjiang, T.: A visual model-driven rapid development toolsuite for parallel applications. In: 2009 WRI World Congress on Computer Science and Information Engineering, vol. 7, pp. 479–483. IEEE (2009)
48. Mengmeng, W., Ce, Y., Jizhou, S., Chao, S., Jinyan, C.: Visual modeling for parallel programming based on DSL. In: 2011 IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom), pp. 623–627. IEEE (2011)
49. Arora, R., Bangalore, P., Mernik, M.: Raising the level of abstraction for developing message passing applications. J. Supercomput. **59**(2), 1079–1100 (2012)
50. Jacob, F., Gray, J., Carver, J.C., Mernik, M., Bangalore, P.: PPModel: a modeling tool for source code maintenance and optimization of parallel programs. J. Supercomput. **62**(3), 1560–1582 (2012)
51. Hernández, F., Bangalore, P., Reilly, K.: Automating the development of scientific applications using domain-specific modeling. In: Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications, pp. 50–54. ACM (2005)
52. Pazel, D.P., Tibbitts, B.R.: Intentional MPI programming in a visual development environment. In: Proceedings of the 2006 ACM Symposium on Software Visualization, pp. 169–170. ACM (2006)
53. DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Darve, E.: Liszt: a domain specific language for building portable mesh-based PDE solvers. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, p. 9. ACM (2011)
54. Puschel, M., Moura, J.M., Johnson, J.R., Padua, D., Veloso, M.M., Singer, B.W., Chen, K.: SPIRAL: code generation for DSP transforms. Proc. IEEE **93**(2), 232–275 (2005)
55. Franchetti, F., Puschel, M., Voronenko, Y., Chellappa, S., Moura, J.M.: Discrete Fourier transform on multicore. IEEE Signal Process. Mag. **26**(6), 90–102 (2009)
56. Prasad, G., Gupta, P.: SIMPAR: a portable object-oriented simulation-science-based metamodel framework for performance modeling, prediction, and evaluation of HPC systems. In: Defense and Security, pp. 264–275. International Society for Optics and Photonics (2004)
57. Pllana, S., Fahringer, T.: UML based modeling of performance oriented parallel and distributed applications. In: Proceedings of the Winter Simulation Conference, vol. 1, pp. 497–505. IEEE (2002)
58. Fahringer, T., Pllana, S., Testori, J.: Teuta: tool support for performance modeling of distributed and parallel applications. In: Computational Science-ICCS 2004, pp. 456–463. Springer, Berlin Heidelberg (2004)
59. Arkin, E., Tekinerdogan, B.: Architectural view driven model transformations for supporting the lifecycle of parallel applications. In: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, pp. 40–49 (2015)
60. Tekinerdogan, B., Arkin, E.: Architecture framework for modeling the deployment of parallel applications on parallel computing platforms. In: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, pp. 185–192 (2015)

Bedir Tekinerdogan is full professor and chairholder of the Information Technology group at Wageningen University in The Netherlands. He has more than 25 years of experience in information technology and software engineering. He is the author of more than 200 peer-reviewed scientific papers. He has been active in dozens of national and international research and consultancy projects with various large software companies whereby he has worked as a principal researcher and leading software/system architect. He graduated more than 50 M.Sc. students, supervised around 20 Ph.D. students and developed around 20 academic computer science courses. He has also been very active in scientific conferences and has organized more than 50 conferences/workshops on important software engineering and computer science research topics. His current research interests include systems engineering, software engineering, information technology and parallel computing.

Ethem Arkin is principal software architect and software team lead at ASELSAN in Turkey. He has worked in more than 10 defence industry projects with the focus on command and control systems. He has completed his Ph.D. in parallel computing and M.Sc. in artificial neural networks at Hacettepe University in Turkey. His current research interests include software architecture, software product lines, software development methodologies, high-performance computing and big data analytics.