The Journal of Supercomputing (2020) 76:5098–5116
https://doi.org/10.1007/s11227-019-02825-6

Generation of high-performance code based on a domain-specific language for algorithmic skeletons

Fabian Wrede[1] · **Christoph Rieger**[1] · **Herbert Kuchen**[1]

Published online: 27 March 2019

Abstract
Parallel programming can be difficult and error prone, in particular if low-level optimizations are required in order to reach high performance in complex environments such as multi-core clusters using MPI and OpenMP. One approach to overcome these issues is based on *algorithmic skeletons*. These are predefined patterns which are implemented in parallel and can be composed by application programmers without taking care of low-level programming aspects. Support for algorithmic skeletons is typically provided as a library. However, optimizations are hard to implement in this setting and programming might still be tedious because of required boiler plate code. Thus, we propose a domain-specific language for algorithmic skeletons that performs optimizations and generates low-level C++ code. Our experimental results on four benchmarks show that the models are significantly shorter and that the execution time and speedup of the generated code often outperform equivalent library implementations using the Muenster Skeleton Library.

Keywords Algorithmic skeletons · Parallel programming · High-performance computing · Model-driven development · Domain-specific language

1 Introduction

Parallel programming for (HPC) is a difficult endeavor, which requires expertise in different frameworks and programming languages. For example, if the code targets a multi-core cluster environment, the programmer needs to know a framework for

✉ Fabian Wrede
fabian.wrede@uni-muenster.de

Christoph Rieger
christoph.rieger@uni-muenster.de

Herbert Kuchen
herbert.kuchen@uni-muenster.de

[1] Department of Information Systems, University of Muenster, European Research Center for Information Systems (ERCIS), Leonardo-Campus 3, 48149 Muenster, Germany

shared-memory architectures, such as OpenMP [1], for distributed-memory architectures, such as MPI [2], and possibly even for accelerators, such as CUDA [3] or OpenCL [4] for GPUs. Especially if advanced performance tuning is required, basic programming skills might not be sufficient. Moreover, using different frameworks in combination can lead to subtle errors, which are difficult to find and resolve.

One approach to solve this problem is based on predefined, typical parallel programming patterns [5] such as `map`, `fold/reduce`, and `zip`. In addition to these data-parallel skeletons, there are task-parallel skeletons, such as `pipeline` or `farm`, and communication skeletons, such as `gather` or `scatter`. In order to use a general-purpose skeleton, a function providing the application-specific computation logic can be passed as an argument. This function is then executed in parallel on all elements of a data structure or data stream.

Thus, by using algorithmic skeletons, low-level details become transparent to the programmer and he or she does not have to consider them or even does not need to know anything about the underlying frameworks. Moreover, algorithmic skeletons make sure that common parallel programming errors, such as deadlocks and race conditions, do not occur.

There are several libraries, which provide algorithmic skeletons (see Sect. 2). A library implementation of algorithmic skeletons makes certain optimizations hard to implement, such as re-arranging skeleton calls, since on the level of C++ and using a combination of low-level frameworks, relevant and irrelevant features are hard to distinguish at runtime. In the present paper, we show how to avoid these drawbacks by generating C++ code from a dedicated (DSL) model. Moreover, we demonstrate how such a (DSL) can facilitate the efficient development of parallel programs by reducing the language to the essential core features and offering useful validation for models.

Our paper is structured as follows: first, we give an overview about different libraries for algorithmic skeletons in Sect. 2. In Sect. 3, our DSL is described and Sect. 4 shows how the language constructs are transformed into C++ code. The results of four benchmark applications are presented in Sect. 5. In Sect. 6, we conclude and point out the future work.

2 Related work

Algorithmic skeletons for parallel programming are mostly provided as libraries. One instance is the Muenster Skeleton Library (Muesli), a C++ library which is used as a reference for the implementation of the presented approach. Muesli offers distributed data structures, and the skeletons are member functions of them. Data-parallel skeletons implemented in Muesli are, for example, `map`, `fold`, `zip`, `mapStencil` and variants of these. Muesli works on multi-core and multi-GPU clusters [6, 7].

Other well-known libraries are, for example, FastFlow, which focuses on task-parallel skeletons and stream parallelism for multi-core systems [8], and eSkel, which provides skeletons for C and MPI [9].

Two libraries we want to examine more closely here are SkePU2 [10] and SkeTo [11], since they incorporate similar concepts as presented in this paper. SkePU2 includes a source-to-source compiler based on Clang and LLVM. A program written with SkePU can always be compiled into a sequential program. A precompiler transforms the program for parallel execution, e.g., adding the `__device__` keyword to functions. SkePU uses custom C++ attributes, which the precompiler recognizes and transforms accordingly.

SkeTo is a library for distributed-memory environments, which focuses on optimizations such as fusion transformations—i.e., combining two skeleton invocations into one and hence reducing the overhead for function calls and the amount of data which is passed between skeletons. The implementation is based on expression templates [12], a meta-programming technique. By using expression templates it is, for example, possible to avoid temporary variables, which are required for complex expressions.

Lately, the concepts of model-driven and generative software development have gained attraction in academia and practice, mainly because of the expected benefits in development speed, software quality, and reduction in redundant code [13]. In addition, DSLs allow for better reuse and readability of models—targeted at both the modeling domain and user experience—while at the same time reducing the complexity through appropriate abstractions [14]. In the domain of high-performance programming, few approaches have been presented in the literature that adopt DSLs. Almorsy and Grundy [15] have presented a graphical notation to ease the shift from sequential to parallel implementations of existing software for CPU and GPU clusters. Anderson et al. [16] have extended the language Julia which is designed for scientific computing and partly aligned with the MATLAB notation. By applying optimization techniques such as parallelization, fusion, hoisting, and vectorization, the generated code significantly improves the computation. In contrast, our approach focuses on the parallelization on clusters of compute nodes.

There are also DSLs for parallel programming with skeletons or parallel patterns, which are embedded into other languages, or enable the creation of embedded DSLs, such as SPar [17] for C++ or Delite [18] for Scala. SPar uses C++ annotations, and therefore, it is possible to parallelize an existing code base without much effort. Additionally, all features of the host language can easily be used. If a code base already exists, even a less complex stand-alone DSL requires more effort for a re-implementation. However, by reducing the language to core features in a stand-alone DSL the complexity can be easily handled by inexperienced programmers.

Danelutto et al. [19] propose a DSL for designing parallel programs based on parallel patterns, which also allows for optimization and rewriting of the pattern composition. The DSL focuses on the management of non-functional properties such as performance, security, or power consumption. Based on the model and non-functional properties, a template providing an optimized pattern composition for the FastFlow library is generated, which the programmer can use to implement the application.

Since this work presents an own language for parallel programming, we want to highlight that some upcoming and established languages aim to benefit from parallel code execution as well. For example, Chapel has built-in concepts for parallel

programming such as *forall loops* and the *cobegin statement*, which allows for starting independent tasks [20]. Also, the C++ standard includes extensions for parallel programming since version C++17 [21]. For example, the algorithms `for_each` and `transform` are comparable to the presented skeletons `map(InPlace)`. However, these capabilities are restricted to a single (potentially multi-core) node and do not support clusters. Another related approach, which provides a source-to-source compiler, is Bones [22]. It takes sequential C code and transforms it into code for GPUs, but again, it does not support distributed systems.

3 A domain-specific language for high-level parallel programming

Many existing approaches to high-level parallel programming provide parallel constructs in the form of a library. As pointed out in introduction, this causes some limitations such as the difficulty to implement optimizations and a higher entry barrier for inexperienced programmers. Thus, we propose a DSL named *Musket* (Muenster Skeleton Tool for High-Performance Code Generation) to tackle these limitations and generate optimized code.

3.1 Benefits of generating high-performance code

The main drawback of using libraries for high-performance computing is the fact that library calls are included in arbitrarily complex code of a host language such as C++. Besides introducing some performance overhead, a library is always restricted by the host language's syntax. In contrast, important design decisions such the syntax and structure of code can be selected purposefully when building a DSL. For example, different algorithmic skeletons as major domain concept for parallelization can be integrated as keywords in the designed language and recognized by the editing component. Consequently, the program specification is more readable for novice users who want to apply their domain knowledge.

A code generator can easily analyze the DSL features based on a formalized meta-model and produce optimized code for different hardware configurations. In the domain of high-level parallel programming using algorithmic skeletons, parallelism can be built into the structure of the language such that the user does not need to cater for parallelism-specific implementations. Required transformations can be provided by the framework developers, for example when applying an algorithmic skeleton to a distributed data structure. In addition, this level of abstraction increases the readability for users who do not need to know the details of (potentially multiple) target platforms but can focus on the high-level sequence of activities.

With regard to framework developers who are concerned with efficient program execution, DSLs introduce additional flexibility. The abstract syntax of the parallel program can be analyzed and modified in order to optimize the generated high-performance code for the target hardware. In particular, recurring—and potentially inefficient—patterns of high-level user code can be transformed to hardware-specific low-level implementations by applying rewrite rules as described in [23]. For

example, *map fusion* may be applied to combine multiple transformations on the same data structure instead of applying them consecutively (cf. Sect. 4.2).

Moreover, a DSL-based approach can be extended to additional platforms in the future by supplying new generator implementations—without changing the input programs. Compared to customizing compilers, DSL creation frameworks such as Xtext further support in creating usable editing components with features such as syntax highlighting and meaningful model[1] validation [24].

3.2 Language overview

The Musket DSL targets rather inexperienced programmers who want to use algorithmic skeletons to quickly write high-performance programs that run on heterogeneous clusters. Therefore, a syntax similar to C++ was chosen to align with a familiar programming language that is common for high-performance scenarios such as simulating physical or biological systems. However, a Musket model is more structured than an arbitrary C++ program and provides four main sections which are described in more detail in the following.[2] The DSL was created using the Xtext language development framework which uses an EBNF-like grammar to specify the language syntax and derives a corresponding Ecore meta-model [26]. Furthermore, a parser as well as an editor component is generated which integrates with the Eclipse ecosystem for subsequent code generation. Consequently, common features of an (IDE) such as syntax highlighting, auto-completion, and validation are available and have been customized to provide contextual modeling support.

[1] The model-driven software development community prefers the notion DSL *model* rather than DSL *program*.

[2] Due to the lack of space, only the overall structure and the main concepts of the language are presented here and an excerpt of the DSL is given in Listing 2. The full DSL specification can be found in our code repository [25].

```
#config PLATFORM CPU
#config PROCESSES 4
#config CORES 24
#config MODE release

const int dim = 16384;

matrix<float,dim,dim,dist> as = {1.0f};
matrix<float,dim,dim,dist> bs = {0.001f};
matrix<float,dim,dim,dist> cs = {0.0f};

float dotProduct(int i, int j, float Cij){
  float sum = Cij;

  for (int k = 0; k < cs.columnsLocal(); k++) {
    sum += as[[i,k]] * bs[[k,j]];
  }

  return sum;
}

main{
  as.shiftPartitionsHorizontally((int a) -> int {return -a;});
  bs.shiftPartitionsVertically((int a) -> int {return -a;});

  for (int i = 0; i < as.blocksInRow(); ++i) {
    cs.map<localIndex, inPlace>(dotProduct());
    as.shiftPartitionsHorizontally((int a)-> int {return -1;});
    bs.shiftPartitionsVertically((int a) -> int {return -1;});
  }

  as.shiftPartitionsHorizontally((int a) -> int {return a;});
  bs.shiftPartitionsVertically((int a) -> int {return a;});
}
```

Listing 1 Musket model for matrix multiplication.

3.2.1 Meta-information

The header of a Musket model consists of meta-information that guides the generation process. On the one hand, target *platforms* and the compiler optimization *mode* can be chosen for convenient debugging of the program. More important, the configuration of *cores* and *processes* is used by the generator to optimize the code for a distributed execution on a high-performance cluster. For example, the setup of distributed data structures, the parallel execution of skeletons, and the intra-cluster communication of calculation results are then automatically managed. An exemplary model for a matrix multiplication according to the algorithm described in [7] is depicted in Listing 1.

3.2.2 Data structure declaration

Because of the distributed execution of the program, all global data structures are declared upfront and distributed to the different compute nodes. Also, global constants can be defined in this block to easily parametrize the program (lines 6–10).

Musket currently supports several primitive data types (*float, double, integer*, and *Boolean*). *Array* and *matrix* collection types also exist and are defined using the C++ template style, e.g., `matrix<double,512,512,dist> table;`. This definition contains the type and dimension of the collection and also provides a keyword indicating whether the collection should be present on all nodes (*copy*), distributed across the nodes (*dist*, *rowDist*, or *columnDist*), or instantiated depending on the context (*loc*). The explicit distinction lets the user control the partitioning of a data structure by means of a user function (see Subsect. 3.2.3). To simplify the handling of distributed data structures, collections can be accessed either using their global index (e.g., `table[42]`) or the local index within the current partition (e.g., `table[[42]]`). Moreover, primitive and collection types can be composed into custom *struct* types.

3.2.3 User function declaration

The third section of a Musket program consists of custom user functions which specify the behavior to be executed on each node within skeleton calls (such as the `dotProduct` function in lines 12–20). Therefore, a wide variety of calculations such as arithmetic and Boolean expressions can be directly expressed in the DSL. In addition to assignments and skeleton applications, different control structures such as *sequential composition*, *if statements*, and *for loops* are available. Moreover, the modeler can use C++ functions from the standard library or call arbitrary external C++ functions (which are, however, not considered for the optimizations described in Sect. 4.2).

Within functions, users can access globally available data structures (declared in the previous section) or create local variables to store temporary calculation results which are not available to other processes. The sophisticated validation capabilities allow for instant feedback to the user when errors are introduced in the model. For example, type inference aims to statically analyze the resulting data type of expressions or type casts and thus warns the user before vainly starting the generation process.

3.2.4 Main program declaration

Finally, the overall sequence of activities in the program is described in the *main* block (lines 22–34 in Listing 1). Besides the possible control structures and expressions described in the previous paragraphs, *skeleton functions* are the main features to write high-level parallel code. Currently, *map*, *fold*, *gather*, *scatter*, and *shift partition* skeletons are implemented in multiple variants. In general, they are applied to a distributed data structure and may take additional arguments such as the previously defined user functions. For convenience and code readability reasons, the user

can instead specify a lambda abstraction for simple operations, e.g., `(int a) -> int {return -a;}`.

```
1  MainBlock ::=  // cf. Section 3.2.4
2   'main' '{' {MainFunctionStatement} '}'
3
4  MainFunctionStatement ::=
5   MusketControlStructure |  // For loop and if clause variants
6   MusketStatement ';'
7
8  MusketStatement ::=
9   MusketVariable |          // Variable declarations
10  MusketAssignment |        // Assigning values to variables
11  SkeletonExpression |      // Arithmetic and boolean expressions
12  FunctionCall              // Function calls without assignment
13
14 SkeletonExpression ::= CollectionObjectRef '.' Skeleton
15
16 Skeleton:           // available algorithmic skeletons
17  'map'  SkeletonOptions '(' MapFunction ')' |
18  'fold' SkeletonOptions '(' IdentityValue ',' FoldFunction ')' |
19  'mapFold' SkeletonOptions
        '(' MapFunction ',' IdentityValue ',' FoldFunction ')' |
20  'zip' SkeletonOptions '(' ObjectRef ',' UserFunction ')' |
21  'gather'  '(' ')' |
22  'scatter' '(' ')' |
23  'shiftHorizontally' '(' UserFunction ')' |
24  'shiftVertically'   '(' UserFunction ')'
25 // alternative skeleton representations omitted
26
27 SkeletonOptions ::= ['<' SkeletonOption {',' SkeletonOption} '>']
28 SkeletonOption  ::= index | localIndex | inPlace
29
30 MapFunction ::= UserFunction
31 FoldFunction ::= UserFunction
32
33 UserFunction ::=
34  FunctionCall |      // reference to user function (cf. 3.2.3)
35  LambdaFunction      // inline definition of functions (cf. 3.2.4)
```

Listing 2 Excerpt of the Musket DSL in EBNF notation.

An excerpt of the Musket grammar concerning the main program declaration is depicted in Listing 2 using the EBNF notation.[3] A *map* skeleton applies a user function to each element of the data structure (either returning a new collection or updating values in place depending on the *SkeletonOption*). A *fold* skeleton (also known as reduce pattern) takes a user function and the identity value of the operation and folds pairs of elements in the collection into a single value. For performance reasons, both skeletons can be combined into a *mapFold* skeleton (see Sect. 4.2). The *zip* skeleton joins two data structures of the same size using the provided user function. The *gather* and *scatter* skeletons are used to transfer objects with different

[3] The Xtext representation of the full DSL is available in our code repository [25].

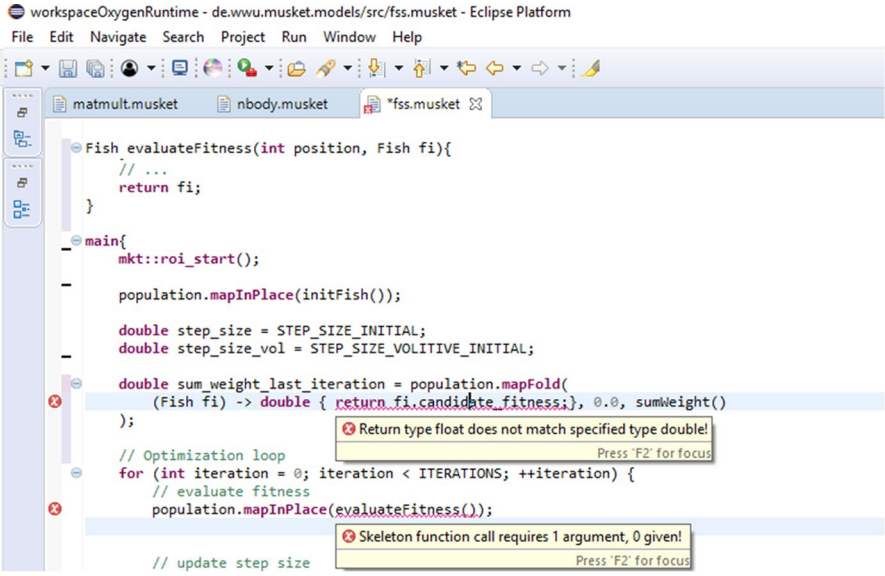

Fig. 1 Integration of custom validation errors in the Eclipse IDE

distribution strategies. Finally, *shift* skeletons can be applied in order to re-distribute rows/columns of distributed matrices between computation nodes.

Again, multiple validators have been implemented to ensure that the types and amount of parameters passed into skeletons match. Meaningful error messages such as depicted in Fig. 1 can be instantly provided while writing the program instead of relying on cryptic failure descriptions when compiling the generated code.

To sum up, the Musket DSL represents a subset of the C++ language in order to handle the complexities of generating parallelism-aware and hardware-optimized code. With only few additions such as distribution modes, local/global collection access, and predefined skeleton functions, a transformation of otherwise regular C++ code into distributed programs which are executable in a cluster environment can be achieved.

4 Code generation for multi-core clusters

In the following section, we demonstrate how certain language constructs are transformed into C++ code. We cover the data structures, data-parallel skeletons, as well as selected specific functions provided by the language. In general, we tried to generate code, which is still readable and makes use of modern features of C++11, 14, and 17.

Further, the way to generate code as described in the following is only one possibility. The approach of using a DSL allows for generating very different implementations to achieve the same behavior. It becomes also possible to consider cost models or descriptions of the target hardware and guide the generation accordingly.

This also includes the generation of code for different architectures. By adding an additional generator, the same language and models can be reused to, for example, generate code for GPUs.

4.1 Data structures

In general, all distributed data structures are represented as wrapper classes around a `std::vector`. Based on the number of processes configured in the model and the distribution type, the size of the local vectors is calculated. Consequently, also for matrices the values are stored only in one vector. When an element in a matrix is accessed, the index is calculated accordingly.

Even though the size of the data structure is known when the code is generated, we decided to use `std::vector` over `std::array`. This is mostly because of the more efficient *move* operation for vectors: for some skeletons, intermediate buffers for sending and receiving data are required and we found vectors to be more efficient when data are moved from temporary buffers to the main vector.

Structs, which are defined in the model, are transformed into C++ structs. Additionally, a default constructor is generated, which initializes all members to default values. Moreover, the generation approach could be used to generate code for different data layouts. At the moment, the data are generated as array of structs, but it could be transformed to struct of arrays or any hybrid representation, which can increase the performance regarding data access and vectorization.

Moreover, there are collection functions, such as `show` and `size`, which can be invoked on data structures. Where possible, these function calls are already evaluated during the generation. For example, the global or local size is known for distributed data structures so that the function call can be replaced by the fixed value.

4.2 Model transformation

The generation approach enables a rewriting step of skeleton calls by performing a model-to-model transformation before the actual generation. In this transformation, certain sequences of skeleton calls can be rewritten in a more efficient way [27]. For example, one or more skeleton calls can be combined through *map fusion*. This is the case for several calls of `map` on the same data structure. The sequence `a.mapInPlace(f); a.mapInPlace(g);` can be joined to `a.mapInPlace(g∘ f);`. For the generated code, this is one less parallel loop, which can save time for synchronization and intermediate data storage.

Also, different skeletons such as `map` and `fold` can be combined. In terms of the presented DSL, `a.mapInPlace(f); x = a.fold(0, g);` can be joined to `x = a.mapFold(f, 0, g);`. In the generated code, this results in one parallel for loop with reduction and a call to `MPI_Allreduce` instead of two loops and the MPI call. Moreover, the intermediate result does not need to be stored in the resulting data structure. However, this transformation would only be valid if `a` was not used in any subsequent skeleton calls. Using static analysis of the model's abstract

syntax tree, such transformations can be specifically targeted, for instance, to optimize specific combinations of skeleton and user function.

4.3 Custom reduction

The implementation of the fold skeleton is based on a straightforward sequence of the OpenMP pragma `#pragma omp parallel for simd reduction` for performing a local reduction in each thread, followed by an `MPI_Allreduce` for combining the local intermediate results. MPI requires a function with the following signature `void f(void *in, void *inout, int *len, MPI_Datatype *dptr)`, which can then be used in reduction operations. By generating this reduction function, it is possible to avoid the combination of a gather operation followed by a second local fold.

4.4 User functions

The generation approach allows for generating user functions in different ways, while they can be expressed at a single point in the model. Moreover, the context in which the function is called can be considered during the generation step, e.g., the function might be generated differently if it is used in a `map` `map_in_place` skeleton. Further examples are the generation for different platforms, e.g., clusters with or without GPUs or generating different functions based on a single user functions as described in Subsect. 4.3. To this respect, the generation approach provides a rather convenient possibility to provide alternative code for the same model.

4.5 Specific Musket functions

There are some additional functions provided by Musket, which are not part of the standard library, such as `rand`. If the `rand` function is used in the model, random engines and distribution objects are generated in the beginning of the main function, so that they can be reused without additional overhead. The actual call to `rand` is generated as `rand_dist[thread_id](random_engines [thread_id])`; thus, it can be used as a part of an expression. Consequently, the DSL conveniently reduces the amount of boilerplate code, since the function can simply be used in the model without, for example, creating an object which creates the random engines on construction.

4.6 Build files

The generation approach offers additional convenience for programmers. In addition to the source and header files, we also generate a CMake file and scripts to build and run the application as well as Slurm job files [28]. Consequently, there is no effort required for the setup and build process, which lowers the entry threshold to parallel programming for inexperienced programmers.

5 Benchmarks

We used four benchmark applications to test our approach: calculation of the Frobenius norm, Nbody simulation, matrix multiplication, and (FSS). In the following subsections, we demonstrate the models, compare them to the C++ implementations with Muesli, and analyze the execution times for both. All execution times are presented in Table 1. The code has been compiled with g++7.3.0 and OpenMPI 3.1.1. Each node of the cluster we have used for the benchmark is equipped with two Intel Xeon E5-2680 v3 CPUs (12 cores each, 30MiB shared L3 cache per CPU and 256KiB L2 cache per core) and 7200MiB memory per node. Hyper-Threading has been disabled.

Table 1 Execution times of the benchmark applications (in seconds)

Benchmark	Execution times (s)				
	Nodes	Cores	Muesli	Musket	Speedup
Frobenius norm	1	1	9.8932	2.0513	4.8228
	1	6	2.1389	0.6906	3.0973
	1	12	1.4890	0.6520	2.2837
	1	18	1.5508	0.6366	2.4361
	1	24	1.6015	0.7240	2.2120
	4	1	2.4793	0.5193	4.7743
	4	6	0.5308	0.2308	2.2996
	4	12	0.3925	0.2034	1.9298
	4	18	0.3943	0.1967	2.0043
	4	24	0.3915	0.1960	1.9970
	16	1	0.6703	0.1364	4.9136
	16	6	0.1497	0.0706	2.1212
	16	12	0.1040	0.0575	1.8093
	16	18	0.1018	0.0538	1.8937
	16	24	0.1060	0.0528	2.0089
Nbody simulation	1	1	7422.7370	7568.5646	0.9807
	1	6	1234.074	1249.3490	0.9878
	1	12	616.9001	624.8900	0.9872
	1	18	411.3290	416.6970	0.9871
	1	24	309.0764	312.6062	0.9887
	4	1	1850.179	1923.3303	0.9620
	4	6	308.7439	312.5329	0.9879
	4	12	154.3643	156.2796	0.9877
	4	18	102.9115	104.2059	0.9876
	4	24	77.7512	78.4770	0.9908
	16	1	473.8626	469.3563	1.0096
	16	6	77.2279	78.1771	0.9879
	16	12	38.6245	39.1056	0.9877
	16	18	25.7638	26.0871	0.9876
	16	24	23.1896	19.6905	1.1777

Table 1 (continued)

Benchmark	Execution times (s)				
	Nodes	Cores	Muesli	Musket	Speedup
Matrix multiplication	1	1	83529.6618	17802.0678	4.6921
	1	6	14726.9833	2430.8756	6.0583
	1	12	7700.4190	1269.3401	6.0665
	1	18	5185.8780	953.9736	5.4361
	1	24	4758.7190	711.8798	6.6847
	4	1	19245.4000	4043.8573	4.7592
	4	6	3578.7240	588.7502	6.0785
	4	12	2086.8870	275.8081	7.5664
	4	18	1549.4470	199.2162	7.7777
	4	24	1376.8620	164.2069	8.3849
	16	1	3655.1590	787.6939	4.6403
	16	6	729.3905	97.0805	7.5133
	16	12	413.3354	44.4174	9.3057
	16	18	252.2129	32.3545	7.7953
	16	24	224.8478	26.2930	8.5516
Fish School Search	1	1	916.3965	762.9235	1.2012
	1	6	158.2332	136.2039	1.1617
	1	12	81.0045	72.4273	1.1184
	1	18	54.8713	52.2537	1.0501
	1	24	45.5823	43.0435	1.0590
	4	1	211.6765	182.9391	1.1571
	4	6	40.1817	33.2093	1.2100
	4	12	20.9549	18.1227	1.1563
	4	18	15.2139	13.7305	1.1080
	4	24	13.5301	12.0980	1.1184
	16	1	54.9478	46.2524	1.1880
	16	6	11.7779	9.4939	1.2406
	16	12	7.1947	6.1798	1.1642
	16	18	5.9784	5.4158	1.1039
	16	24	5.7644	5.2615	1.0956

5.1 Frobenius norm

The calculation of the Frobenius norm for matrices consists of three steps. First, all values are squared, then all values are summed up, and finally, the square root of the sum yields the result. We used a $32{,}768 \times 32{,}768$ matrix with double precision values. The model is presented in Listing 3.

```
1  #config PLATFORM CPU
2  #config PROCESSES 4
3  #config CORES 24
4  #config MODE release
5
6  matrix<double,32768,32768,dist> as;
7
8  main{
9    // init
10   as.map<index, inPlace>((int x, int y, double a) -> double
         {return (double) x + y + 1.5;});
11
12   mkt::roi_start(); // Timer start
13
14   as.map<inPlace>((double a) -> double {return a * a;});
15   double fn = as.fold(0.0, (double a, double b) -> double
         {return a + b;});
16   fn = std::sqrt(fn);
17
18   mkt::roi_end(); // Timer end
19
20   mkt::print("Frobenius norm is %.5f.\n", fn);
21 }
```

Listing 3 Model for Frobenius norm calculation

There is one matrix defined as a distributed data structure. Since all user functions are written as lambda expressions, there is no need for separately defined user functions. Within the main block, the logic for the program is defined. First, in this case, the matrix is initialized with arbitrary values. The calculation of the Frobenius norm is modeled in lines 14–16. The functions `roi_start` and `roi_end` are merely for benchmark purposes and trigger the generation of timer functions. In line 20, the Musket function `print` is used, so that the result is only printed once by the main process.

The results for this benchmark already reveal some interesting insights, even though the complexity of the program is rather low, which is also the reason why the program does not scale very well, when increasing the number of cores per node.

Musket achieves good speedups compared to the Muesli implementation. There are multiple effects that lead to the observed results. First of all, for the generated code GCC is able to vectorize the loops performing the `map` and `fold` operations. Auto-vectorization is, however, not possible for the Muesli implementation. Additionally, Muesli does not consider configurations with one process or one thread as special cases, but relies on the fact that MPI routines also work for one process and that all used OpenMP pragmas are ignored for sequential programs. In contrast, Musket checks the configuration and generates the code accordingly. Consequently, there are no MPI routines in the generated program and less operations are required regarding the management of the data structures, if there is only one process. When

a data structure is created in Muesli, the global and local sizes have to be calculated, the new memory has to be allocated, etc. In Musket, the data structures are defined in the model, and all required information, such as the size, can be generated and therefore need not be calculated during program execution.

To give a perspective on the effort of writing a parallel program, we want to point to the lines of code for the Musket model, the Muesli implementation, and the generated source file. We did not use the lambda notation for Musket for counting the lines of code, since Muesli requires functors in certain situations and in this way the results become more comparable. While the Musket model consists of 20 lines, the Muesli implementation has 45 lines and the generated source code 398. Thus, we conclude that the DSL provides a concise way to express a parallel program.

Another aspect to mention here is the skeleton fusion optimization. The lines 14 and 15 could also be written as `double fn = as.mapFold(square(), 0.0, sum());`, if we assume that the lambda expressions correspond to the respective functions. In the generated code, this would reduce the two loops for the map and fold operations into a single loop. The execution times in Table 1 do not reflect this optimization to keep the results comparable, because Muesli does not offer a combined `mapFold` skeleton. As an example, for a configuration with 4 nodes and 24 cores, the execution time has been 0.07 s with skeleton fusion, which corresponds to a speedup of 2.68 compared to the Musket implementation without skeleton fusion.

5.2 Nbody simulation

In the case of the Nbody simulation with 5,00,000 particles over five time steps, the execution times for both implementations are rather similar. For most configurations, the Musket generated code is slightly slower, while for the configuration with 16 nodes and 24 cores per node there is a speedup of 1.18. The benchmark does not allow for much optimization. Transformations such as skeleton fusion are not applicable, and the user function used in the `map` skeleton contains function calls as well as multiple branches, which prevent efficient vectorization.

In an alternative implementation of the Musket generator, we investigated the effects of inlining of user functions to avoid overhead for function calls. Contrary to intuition, this approach is not blindly applicable as can be seen for the Nbody simulation where inlining has not been advantageous. For a configuration with 4 nodes and 24 cores, the execution time was even 87.08 s compared to 78.48 s for the current approach. We have simulated the behavior of the application with Valgrind's cachegrind and callgrind tools [29]. The number of L3 cache misses as well as the amount of unnecessary data loaded to cache is higher than for the Muesli implementation. Consequently, we refrain from applying loop unrolling and inlining by default and propagate these optimizations to the subsequent compilation step.

In terms of complexity, both implementations have about the same size. The Musket model consists of 77 lines of code, whereas the Muesli program consists of 84 lines. The generated code consists of 335 lines.

5.3 Matrix multiplication

The matrix multiplication benchmark shows a case, in which massive improvements become possible due to the code generation approach. We have performed the matrix multiplication with matrices of 16.384×16.384 single precision values, but first, we have to emphasize again that Musket targets rather inexperienced programmers. The benchmark is set up in such a way that the second matrix for the multiplication is not transposed. Hence, the data is stored row major, but the user function iterates column-wise through the matrix, which leads to inefficient memory accesses. The model is shown in Listing 1 in Sect. 3.

In the Muesli implementation, the compiler is not able to vectorize the calculation and to optimize the memory access. However, this is the case for the generated program, which leads to significant shorter execution times. The speedups for configurations with multiple nodes and cores range between 6.08 and 9.31.

The model has 42 lines of code, while the Muesli implementation has 74. Again, the effort for implementing the benchmark has been reduced. In comparison, the generated code has 542 lines.

5.4 Fish School Search

The FSS benchmark showcases a complex and more real-world example of a parallel program. FSS is a swarm-intelligent meta-heuristic to solve hard optimization problems [30]. The model has 244 lines of code and the Muesli implementation even 623, which is a reduction of about 61%. The generated code consists of 866 lines of code. A detailed discussion of the implementation with Muesli can be found in [31].

```
1  #config PLATFORM CPU
2  #config PROCESSES 4
3  #config CORES 24
4  #config MODE release
5
6  const int NUMBER_OF_FISH = 2048;
7  const int DIMENSIONS = 512;
8
9  struct Fish{
10   array<double,DIMENSIONS,loc> position;
11   double fitness;
12   array<double,DIMENSIONS,loc> candidate_position;
13   double candidate_fitness;
14   array<double,DIMENSIONS,loc> displacement;
15   double fitness_variation;
16   double weight;
17   array<double,DIMENSIONS,loc> best_position;
18   double best_fitness;
19 };
20
21 array<Fish,NUMBER_OF_FISH,dist> population;
22 // [...]
23
24 main{
25   // [...]
26   double sum_weight = population.mapFold(
27       (Fish fi) -> double {return fi.weight;}, 0.0,
28       (double a, double b) -> double {return a + b;});
29   // [...]
30 }
```

Listing 4 Excerpt of the Musket model for Fish School Search

Listing 4 shows excerpts of the FSS model. Since Musket also supports distributed data structures of complex types—which can include arrays—it becomes very convenient to work with. The struct for *Fish* is defined in lines 9–19, and the distributed array is defined in line 21. The fact that complex types are allowed in distributed data structures highlights the benefit of the fused `mapFold` skeleton. For one operator called *collective volitive movement*, it is necessary to calculate the sum of the weight of all fish. Line 26 shows how this can be conveniently done by invoking the `mapFold` skeleton on the `population` array. In the map part, the weight of each fish is taken and in the fold part the sum is calculated. In the generated code, this is efficiently performed in a single parallelized loop. The execution times show a slight improvement for most configurations with speedups up to 1.24.

6 Conclusions and future work

In this paper, we have proposed a DSL for parallel programming, which is based on algorithmic skeletons. Regarding the execution times, the generated code can offer significant speedups (of up to 9) compared to the library-based approach Muesli for most benchmarks and configurations. Furthermore, the benchmark applications have shown that the DSL offers a convenient and concise way to express applications. Based on a single model, it becomes possible to generate different implementations to achieve the same behavior.

For future work on multi-core clusters, this leads to the problem of selecting the best (i.e., fastest) alternative. This could be achieved by considering cost models or descriptions of the target hardware or by comparing alternatives experimentally. At the same time, we are investigating more model transformations to identify performance potentials in the interplay of user functions and skeletons in order to further optimize the generated code.

In addition, current work focuses on implementing a corresponding generator for multi-GPU clusters. Consequently, the modeler can express the desired program using the Musket DSL and generate optimized code for multiple platforms using the same Musket models.

References

1. Chapman B, Jost G, van der Pas R (2008) Using OpenMP: portable shared memory parallel programming. Scientific and engineering computation. MIT Press, Cambridge
2. Gropp W, Lusk E, Skjellum A (2014) Using MPI: portable parallel programming with the message-passing interface. Scientific and engineering computation, 3rd edn. MIT Press, Cambridge
3. Nickolls J, Buck I, Garland M, Skadron K (2008) Scalable parallel programming with CUDA. Queue 6(2):40–53
4. Stone JE, Gohara D, Shi G (2010) OpenCL: a parallel programming standard for heterogeneous computing systems. Comput Sci Eng 12(3):66–73
5. Cole M (1991) Algorithmic skeletons: structured management of parallel computation. MIT Press, Cambridge
6. Ernsting S, Kuchen H (2012) Algorithmic skeletons for multi-core, multi-GPU systems and clusters. Int J High Perform Comput Netw 7(2):129–138
7. Ernsting S, Kuchen H (2017) Data parallel algorithmic skeletons with accelerator support. Int J Parallel Program 45(2):283–299
8. Aldinucci M, Danelutto M, Meneghin M, Torquati M, Kilpatrick P (2010) Efficient streaming applications on multi-core with FastFlow: the biosequence alignment test-bed. In: Chapman B, Desprez F, Joubert GR, Lichnewsky A, Peters F, Priol T (eds) Parallel computing: from multicores and GPU's to petascale, advances in parallel computing, vol 19. IOS Press, Amsterdam
9. Benoit A, Cole M, Gilmore S, Hillston J (2005) Flexible skeletal programming with eSkel. In: Cunha JC, Medeiros PD, (eds) Proceedings of the 11th International Euro-Par Conference on Parallel Processing (Euro-Par '05), Lecture Notes in Computer Science, vol 3648. Springer, pp 761–770
10. Ernstsson A, Li L, Kessler C (2017) SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. Int J Parallel Program 46(1):62–80
11. Matsuzaki K, Emoto K (2010) Implementing fusion-equipped parallel skeletons by expression templates. In: Morazán MT, Scholz S (eds) Implementation and application of functional languages. Springer, Berlin, pp 72–89
12. Veldhuizen T (1995) Expression templates. C++ Rep 7(5):26–31

13. Stahl T, Völter M (2006) Model-driven software development. Wiley, Chichester
14. Mernik M, Heering J, Sloane AM (2005) When and how to develop domain-specific languages. ACM Comput Surv 37(4):316–344
15. Almorsy M, Grundy J (2015) Supporting scientists in re-engineering sequential programs to parallel using model-driven engineering. In: 2015 IEEE/ACM 1st International Workshop on Software Engineering for High Performance Computing in Science, pp 1–8. IEEE
16. Anderson TA, Liu H, Kuper L, Totoni E, Vitek J, Shpeisman T (2017) Parallelizing Julia with a non-invasive DSL. In: Müller P (ed) 31st European Conference on Object-Oriented Programming (ECOOP '17), Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, vol 74. Dagstuhl, Germany, pp 4:1–4:29, 2017
17. Griebler D, Danelutto M, Torquati M, Fernandes LG (2017) SPar: a DSL for high-level and productive stream parallelism. Parallel Process Lett 27(01):1740005
18. Sujeeth AK, Brown KJ, Lee H, Rompf T, Chafi H, Odersky M, Olukotun K (2014) Delite: a compiler architecture for performance-oriented embedded domain-specific languages. ACM Trans Embed Comput Syst (TECS) 13(4s):134
19. Danelutto M, Torquati M, Kilpatrick P, (2016) A DSL based toolchain for design space exploration in structured parallel programming, Procedia Computer Science, vol 80, pp 1519–1530. In: International Conference on Computational Science 2016, ICCS 2016, San Diego, California, USA, 6–8 June 2016
20. Chamberlain BL, Callahan D, Zima HP (2007) Parallel programmability and the chapel language. Int J High Perform Comput Appl 21(3):291–312
21. Standard ISO (2015) Programming languages–technical specification for C++ extensions for parallelism, standard ISO/IEC TS 19570:2015. International Organization for Standardization, Geneva
22. Nugteren C, Corporaal H (2015) Bones: an automatic skeleton-based C-to-CUDA compiler for GPUs. ACM Trans Archit Code Optim (TACO) 11(4):35
23. Steuwer M, Fensch C, Lindley S, Dubach C (2015) Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP '15. ACM, New York pp 205–217
24. Bettini L (2013) Implementing domain-specific languages with Xtext and Xtend. Community experience distilled. Packt Publishing, Birmingham
25. Wrede F, Rieger C (2018) Musket material repository. https://github.com/wwu-pi/musket-material. Accessed 26 Mar 2019
26. Eclipse foundation (2019) The eclipse foundation. Xtext documentation. https://eclipse.org/Xtext/documentation/. Accessed 26 Mar 2019
27. Kuchen H (2004) Optimizing sequences of skeleton calls. In: Lengauer C, Batory D, Consel C, Odersky M (eds) Domain-specific program generation. Lecture notes in computer science, vol 3016. Springer, Berlin, pp 254–274
28. Yoo AB, Jette MA, Grondona M (2003) SLURM: simple linux utility for resource management. In: Feitelson D, Rudolph L, Schwiegelshohn U (eds) Job scheduling strategies for parallel processing. Springer, Berlin, pp 44–60
29. Nethercote N, Seward J (2007) Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07, ACM, New York, NY, USA, pp 89–100
30. Bastos-Filho CJA, de Lima Neto FB, Lins AJCC, Nascimento AIS, Lima MP (2008) A novel search algorithm based on fish school behavior. In: Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC '08). IEEE, pp 2646–2651
31. Wrede F, Menezes B, Kuchen H (2018) Fish school search with algorithmic skeletons. Int J Parallel Program 47(2):234–252