Int J Parallel Prog (2017) 45:283–299
DOI 10.1007/s10766-016-0416-7

Data Parallel Algorithmic Skeletons with Accelerator Support

Steffen Ernsting[1] · **Herbert Kuchen**[1]

Received: 6 August 2015 / Accepted: 21 March 2016 / Published online: 31 March 2016

Abstract Hardware accelerators such as GPUs or Intel Xeon Phi comprise hundreds or thousands of cores on a single chip and promise to deliver high performance. They are widely used to boost the performance of highly parallel applications. However, because of their diverging architectures programmers are facing diverging programming paradigms. Programmers also have to deal with low-level concepts of parallel programming that make it a cumbersome task. In order to assist programmers in developing parallel applications *Algorithmic Skeletons* have been proposed. They encapsulate well-defined, frequently recurring parallel programming patterns, thereby shielding programmers from low-level aspects of parallel programming. The main contribution of this paper is a comparison of two skeleton library implementations, one in C++ and one in Java, in terms of library design and programmability. Besides, on the basis of four benchmark applications we evaluate the performance of the presented implementations on two test systems, a GPU cluster and a Xeon Phi system. The two implementations achieve comparable performance with a slight advantage for the C++ implementation. Xeon Phi performance ranges between CPU and GPU performance.

Keywords High-level parallel programming · Algorithmic skeletons · GPGPU · Hardware accelerators

✉ Steffen Ernsting
s.ernsting@uni-muenster.de

Herbert Kuchen
kuchen@uni-muenster.de

[1] University of Muenster, Leonardo-Campus 3, 48149 Muenster, Germany

1 Introduction

Nowadays, multi-core and many-core processors are ubiquitous. Both the growing complexity of applications and the growing amount of data lead to high demand for high performance. In the last few years we can observe a trend towards hardware accelerators such as graphics processing units (GPUs) or the Intel Xeon Phi. Both GPUs and Xeon Phi [1] are many-core processors comprising hundreds or thousands of cores on a single chip and promising to deliver high performance in teraflops-scale.

Programming GPUs is very challenging because of their specialized nature for highly parallel, high throughput computing. Fully exploiting their computing capabilities still requires programmers to deal with intrinsic low-level concepts such as e.g. memory allocation in device memory and data transfer to or from main memory. Programming multi-GPU systems is even more challenging because the programmer is responsible for managing multiple buffers and data transfer between GPUs. These low-level concepts constitute a high barrier to efficient development of parallel applications and also make it a tedious and error-prone task.

The Intel Xeon Phi is based on a x86-compatible multiprocessor architecture. Programming the Xeon Phi therefore is a much more convenient task than programming GPUs because existing parallelization software and tools such as e.g. MPI and OpenMP can be utilized. Nevertheless, programs written for the Xeon Phi processor cannot be run on GPUs and vice versa. Porting existing applications from Xeon Phi to GPUs or vice versa can be very complex and typically involves a lot of effort.

With algorithmic skeletons [2,3] Cole has proposed an approach to structured high-level parallel programming. Algorithmic skeletons can be considered as high-level tools that encapsulate well-defined, frequently recurring parallel and distributed programming patterns, thereby hiding low-level details and also encouraging a structured way of parallel programming. The high level of abstraction ensures portability: skeletons can be implemented for various architectures. Thus, programs that utilize algorithmic skeletons are innately portable between various architectures (that are supported).

In this paper, we present a comparison of two implementations of the *Muenster Skeleton Library* (Muesli), one in C++ and one in Java, in terms of library design and programmability, and performance. A skeleton library allows for platform independent development of parallel applications. Supported computing architectures include (multi-core) CPUs and GPUs (C++ and Java), as well as Xeon Phi (C++ only). We also present four benchmark applications in order to draw a performance comparison between the two presented implementation approaches as well as between the supported computing architectures. The four benchmark applications include matrix multiplication, N-Body computations, shortest paths, and ray tracing.

The remainder of this paper is structured as follows. Section 2 introduces the *Muenster Skeleton Library*, briefly pointing out the underlying concepts. The implementation of data parallel skeletons with accelerator support is presented in Sect. 3 where we point out some implementation aspects that both implementations have in common and where they distinguish from each other. The two implementations are evaluated, in terms of performance, in Sect. 4. Related work is discussed in Sect. 5 and finally, Sect. 6 concludes the paper and gives a short outlook to future work.

2 The Muenster Skeleton Library Muesli

The C++ library Muesli provides algorithmic skeletons as well as distributed data structures for shared and distributed memory parallel programming. It is built on top of MPI and OpenMP. Thus it provides efficient support for multi- and many-core computer architectures as well as clusters of both. A first implementation of data parallel skeletons with GPU support using CUDA [4] was presented in [5]. A pure Java implementation of data parallel skeletons (CPU only) was presented in [6].

Conceptually, we distinguish between data parallel and task parallel skeletons. Data parallel skeletons such as *map*, *zip*, and *fold* are provided as member functions of distributed data structures, including a one-dimensional array, a two-dimensional matrix, and a two-dimensional sparse matrix [7].[1] Communication skeletons such as *permutePartition* assist the programmer in dealing with data that is distributed among several MPI processes. Task parallel skeletons represent well-known process topologies, such as *Farm*, *Pipeline* (Pipe), *Divide and Conquer* (D&C) [8] and *Branch and Bound* (B&B) [9]. They can be arbitrarily nested to create a process topology that defines the overall structure of a parallel application. The algorithm-specific behavior of such a process topology is defined by some particular user functions that describe the algorithm-specific details.

In Muesli, a user function is either an ordinary C++ function or a functor, i.e. a class that overrides the function call operator. Due to memory restrictions, GPU-enabled skeletons must be provided with functors as arguments, CPU skeletons can take both functions and functors as arguments. As a key feature of Muesli, the well-known concept of *Currying* is used to enable partial application of user functions [10]. A user function requiring more arguments than provided by a particular skeleton can be partially applied to a given number of arguments, thereby yielding a "curried" function of smaller arity, which can be passed to the desired skeleton. On the functor side, additional arguments are provided as data members of the corresponding functor.

3 Implementation of Data Parallel Skeletons with Accelerator Support

The implementation of data parallel skeletons with GPU support has already been presented in [5] (C++) and [6,11] (Java). In this paper, we want to point out implementation aspects that both implementations have in common and aspects that are different for the two implementations. Additionally, we briefly explain how the Xeon Phi coprocessor is supported by the C++ implementation. For a more detailed description of each implementation please refer to the above mentioned papers.

3.1 Distributed Data Structures

As we already mentioned in Sect. 2, data parallel skeletons in Muesli are provided as member functions of distributed data structures. As the name suggests, distributed data

[1] In this paper, we focus on the data structures array and matrix. The sparse matrix currently does not provide accelerator skeletons.

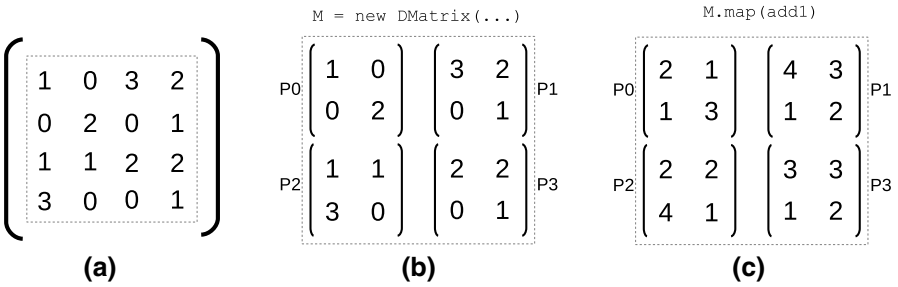

Fig. 1 **a** Global view of a distributed matrix. **b** Local view of a distributed matrix. **c** Local view after applying the Map skeleton

structures are distributed among several MPI processes, each maintaining a partition of the entire data structure. The concept of such parallel data structures is common to both implementations. We distinguish between the *global view* that considers the entire data structure, and the *local view* that considers the decomposition into local partitions (see Fig. 1a, b). Programmers may concentrate on the *global view* because all data parallel skeletons process the entire data structure, meaning all elements of a data structure (see Fig. 1c). However, when designing their programs, they must keep in mind that data is physically separated in distributed memory. Currently, the data structures `DArray` (one-dimensional) and `DMatrix` (two-dimensional) provide skeletons with accelerator support.

3.2 Parallelization

For our C++ implementation of data parallel skeletons with accelerator support we make use of a multi-tier approach based on MPI, OpenMP, and CUDA for parallelization. The code can be separately compiled for each supported platform. MPI is used for inter-node (distributed memory) parallelization. The CPU and Xeon Phi versions utilize OpenMP and the GPU version utilizes CUDA for intra-node (shared memory) parallelization.

In our Java implementation we make use of MPJ Express [12] (MPJE, a Java implementation of MPI) and Aparapi [13,14] for parallelization. Aparapi is an API for data parallel Java that translates at runtime suitable Java (byte)code into OpenCL [15] code that can be executed on GPUs. It does not (yet) deliver the performance of Java bindings such as jCuda [16,17], JOCL [18], or JogAmp's JOCL [19], but instead provides good programmability, which is ideal for high-level approaches such as algorithmic skeletons [20]. At present Aparapi is restricted to primitive data types and includes only limited support for simple Java objects. However, because of incompatible memory layouts between Java and OpenCL the performance will be poor. Future releases may address this issue. For additional information on how we integrate MPJE and Aparapi please refer to [6,11].

Xeon Phi Support For the GPU version computational offloading is employed: the main code runs on the host system and specific compute-intensive tasks (in this case

the skeletons) are offloaded to the GPU. Data has to be transferred explicitly. For the Xeon Phi version, the code runs in so-called native mode, where the entire application runs on the coprocessor. The advantage of this approach is that data transfer to the coprocessor's memory is handled implicitly. For applications with a high portion of sequential code this can be disadvantageous, because the Xeon Phi is mainly designed to run highly parallel code and performs poorly running sequential code. However, most applications based on data parallel skeletons are highly parallel by nature and involve only a small fraction of sequential code.

3.3 Data Parallel Skeletons

On the basis of the `mapInPlace` skeleton of the distributed array we want to briefly explain how the data parallel skeletons are implemented. Listing 1 shows the C++ implementation for both the CPU and Xeon Phi variant (lines 2–9) and the GPU variant (lines 11–22) of this skeleton.

```
1  // CPU and Xeon Phi variant of the mapInPlace Skeleton
2  template <typename MapFunctor>
3  void mapInPlace(MapFunctor& f)
4  {
5    #pragma omp parallel for
6    for (int i = 0; i < nLocal; i++) {
7      setLocal(i, f(in_data[i]));
8    }
9  }
10 // GPU variant of the mapInPlace Skeleton
11 template <typename MapFunctor>
12 void mapInPlace(MapFunctor& f)
13 {
14   // upload data first
15   upload();
16   // map
17   for (int i = 0; i < num_gpus; i++) {
18     cudaSetDevice(i);
19     mapKernel<<<blocks, threads>>>(in_data, out_data, nLocal, f);
20     f.notify();
21   }
22 }
```

Listing 1 C++ implementation of the `mapInPlace` skeleton for CPUs, Xeon Phis, and GPUs.

On the CPU (Xeon Phi, respectively) side, the main part is a parallel for-loop that iterates over the local partition in order to call the user function `f` (which actually is not a function but a functor) for each element of the data structure (lines 5–8). On the GPU side we need to first transfer data to the GPUs' memory (line 15). After that, the map kernel can be launched on each GPU (lines 17–21). The map kernel straightforwardly maps the user function to each element of each local GPU partition, such that finally every element of the distributed data structure is processed according to the user function. The call of the `notify()` function in line 20 originates from the mechanism that allows for additional arguments to the user function. It will be

explained in detail in the next section. Listing 2 shows the Java implementation of the `mapInPlace` skeleton.

```
1 public void mapInPlace(MapKernel kernel) {
2   kernel.init(in_data, out_data);
3   kernel.execute(Range.create(nLocal));
4   kernel.dispose();
5 }
```

Listing 2 Java implementation of the `mapInPlace` skeleton for CPUs and GPUs.

Because of the use of Aparapi as the parallelization tool, the `mapInPlace` skeleton is very simple in the Java version. The user provides the user function in terms of a so-called `MapKernel`, which is an abstract class that incorporates all the functionality that is necessary for Aparapi to generate OpenCL code and run it on the GPU. The kernel is initialized with input and output data in line 2 and launched in line 3. Disposing the kernel (line 3) ensures synchronization with the host thread. In the next section we will give detailed information about this abstract kernel class and how it is to be extended by the user.

In order to give an example of the interaction between shared and distributed memory parallelization, Listing 3 schematically presents the `fold` skeleton. According to the two-tier parallelization, first, each process calculates a local result by folding its local partition of the data structure (line 2). This is done in parallel by CPU or GPU threads, depending on the execution mode. In the next step all the local results are shared among the processes (lines 4–5) to finally calculate a global result which is then returned by the skeleton (line 7).

```
1 int fold(FoldFunction f) {
2     int localResult = localFold(f, localPartition);
3
4     int[] localResults = new int[numProcs];
5     allgather(localResult, localResults); // simplified
6
7     return localFold(f, localResults);
8 }
```

Listing 3 Scheme of the `fold` skeleton.

3.4 Providing the User Function

In order to provide a skeleton with a user function, the user has to implement a functor. Listing 4 exemplarily shows the abstract class `MapFunctor` that is to be extended by the user. The actual user function is implemented in terms of the function call operator (line 6). In case of a map functor, this function takes a value of type `IN` as argument and returns a value of type `OUT`. The preprocessor macro `MSL_UFCT`[2] expands to `__host__ __device__` when compiled for GPU platforms via the NVIDIA compiler.

[2] `MSL_UFCT` is a contraction for `MSL_USERFUNCTION`.

It tells the compiler to generate code that can be run on the host system (i.e. on the CPU) as well as on the GPU. When compiled with any other compiler, this macro expands to the empty word. A concrete functor that extends the abstract class `MapFunctor` will be presented in Listing 6.

```
1 template <typename IN, typename OUT>
2 class MapFunctor : public FunctorBase
3 {
4 public:
5   // To be implemented by the user.
6   MSL_UFCT virtual OUT operator() (IN value) const = 0;
7   virtual ~MapFunctor();
8 };
```

Listing 4 Class `MapFunctor` to be extended by the user.

On the Java side, the user has to implement a functor in terms of a so-called kernel (see Listing 5). This is due to the way how Aparapi generates OpenCL code from Java code. Due to the restriction to primitive data types, these kernels have to implemented for a specific (primitive) type (in this case `int`). The map functionality is provided with the `run` method within which the map function is applied. Data to be processed (arrays `in` and `out` in line 2) must be a member of the kernel in order for Aparapi to be able to handle data transfer to GPU memory through JNI. Initialization of the map kernel is handled by the map skeleton (see Listing 2, line 2). The important part for the user is the abstract method `mapFunction` in line 4 that is to be implemented in order to provide the functionality of the user function. A concrete kernel that extends the abstract class `MapKernel` will be presented in Listing 8.

```
1  public abstract class MapKernel extends Kernel {
2      protected final int[] in, out;
3      // To be implemented by the user.
4      public abstract int mapFunction(int value);
5      public void run() {
6          int gid = getGlobalId();
7          out[gid] = mapFunction(in[gid]);
8      }
9      // Called by skeleton implementation.
10     public void init(DIArray in, DIArray out) {
11       this.in = in.getLocalPartition();
12       this.out = out.getLocalPartition();
13     }
14 }
```

Listing 5 Abstract class `MapKernel` to be extended by the user.

Additional Arguments for the User Function The arguments of a user function are determined by the skeleton that calls this particular function. The map skeleton for instance passes only the current value to the user function. For most applications, however, it is crucial to enable the user function to access further data. In former versions of Muesli, this mechanism was accomplished through the concept of *Currying* [10] where function pointers are incorporated into functors that hold additional arguments. Because in CUDA it is restricted to take the address of a `__device__` function [21] it was not feasible to simply adopt this feature for the

skeletons with GPU support. Having the user directly implementing a functor anyway immediately suggests to simply add additional arguments in terms of a functor's data members as shown in Listing 6. This works well for data types whose size in bytes can be correctly calculated by the `sizeof`-operator, e.g. primitives and simple structures. This is usually not the case for classes with pointer data members such as for instance the classes `DArray` and `DMatrix`.

```
struct AddN : public MapFunctor<int, int>
{
  int n;

  AddN(int n_) : n(n_)
  {
  }

  MSL_UFCT int operator()(int value) const
  {
    return value + n;
  }
};
```

Listing 6 Adding primitive arguments to a functor.

Adding arguments of such complex types is problematic: pointer data that are to be accessed from a GPU device requires to be uploaded to GPU memory first. In a multi-GPU setting, due to disjoint memory of multiple GPUs, this leads to holding multiple pointers, one for each GPU and one pointing to the host data in main memory. When accessing these pointers on the GPUs, we need to take care that each GPU accesses the correct pointer (that points to an address in its memory). For that reason, the functors in Muesli are implemented similarly to the observer pattern [22] (see Fig. 2). Listing 7 shows the implementation in C++ of the abstract classes `ArgumentType` and `FunctorBase`. In order to serve as an argument type, a class needs to extend the abstract class `ArgumentType`, i.e. override the `update` function. This function notifies a concrete observer, that it needs to update the current pointer, so that it points to an address in the correct GPU memory. On the functor side, the user has to register concrete observers (i.e. `Arguments`) with the help of the `addArgument` function. At first glance, this may seem to be complex. However, users need to deal with this procedure only when defining their own argument types. For the distributed data structures `DArray` and `DMatrix` Muesli provides some kind of proxy classes `LArray` and `LMatrix`[3] that implement this behavior described above. The reason for the use of proxy classes instead of implementing this behavior directly in the classes `DArray` and `DMatrix` is that functors must be passed by value to the CUDA kernels. This would involve a copy of the entire data structure to be made by the copy constructor every time the functor was passed to a CUDA kernel (in case of 4 GPUs 4 copies would be made). The classes `LArray` and `LMatrix` instead are just shallow copies that store the pointers only.

[3] The `L` in `LArray` and `LMatrix` stands for "local" and is thought to denote that only the local partition of a distributed data structure can be accessed locally.

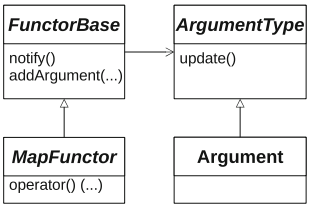

Fig. 2 UML class diagramm for functor classes in Muesli

```
class ArgumentType
{
public:
  virtual void update();
  virtual ~ArgumentType();
};

class FunctorBase
{
public:
  void notify();
  void addArgument(ArgumentType* arg);
  virtual ~FunctorBase();
protected:
  std::vector<ArgumentType*> args;
};
```

Listing 7 Base class for functors enables additional arguments.

On the Java side, passing additional arguments is much simpler. However, here, arguments are restricted to primitive data types and arrays of primitive data types. The user can simply add an argument to the user function in terms of a data member of the kernel class. Aparapi handles memory allocation and data transfer to the GPU memory. A simple user function with an additional argument is exemplarily shown in Listing 8.

```
public class AddN extends MapKernel {
    protected int n;

    public AddN(int n) {
        this.n = n;
    }

    public int mapFunction(int value) {
        return value+n;
    }
}
DIArray A = new DIArray(...);
A.map(new AddN(2));
```

Listing 8 Adding additional arguments to a kernel.

4 Experimental Results

In order to demonstrate how the presented data parallel skeletons with accelerator support perform, we have implemented four benchmark applications: matrix multiplication, N-Body computations, shortest paths, and ray tracing. There are two test systems on which the benchmarks were conducted: The first system is a multi-node GPU cluster. Each node is equipped with two Intel Xeon E5-2450 (Sandy Bridge) CPUs with a total of 16 cores and two NVIDIA Tesla K20x GPUs. The second system is a Xeon Phi system including 8 Xeon Phi 5110p coprocessors. For each of the first three benchmarks, we considered six configurations:

- *two CPU configurations C++ CPU*, *Java CPU*
- *three GPU configurations C++ GPU*, *C++ multi-GPU*, and *Java GPU*
- *one Xeon Phi configuration C++ Xeon Phi*

For the *CPU* configurations, 16 threads per node are employed. For the *GPU* configurations *C++ GPU* and *Java GPU* a single GPU per node has been utilized. For the *C++ multi-GPU* configuration two GPUs per node have been utilized. Each of these configurations was run on multiple nodes, ranging from 1 to 16 nodes. For the *C++ Xeon Phi* configuration up to 8 Xeon Phis have been utilized. In this case one Xeon Phi processor corresponds to one node.

Each of the first three benchmark applications is implemented just twice: one C++ version and one Java version. In order to run the benchmark with different hardware configurations, we can simply compile the same application for multiple architectures. Due to the restriction to primitive data types on the Java side, we have implemented the ray tracing benchmark only for the C++ version. For that reason, we considered only the four C++ configurations for this benchmark.

4.1 Matrix Multiplication

For the matrix multiplication benchmark, we have implemented Cannon's algorithm [23] for multiplying two $n \times n$-matrices. It is based on a checkerboard block decomposition and assumes the matrices to be partitioned into p submatrices (local partitions) of size $m \times m$, where p denotes the number of processes and $m = n/\sqrt{p}$. Initially the submatrices of A and B are shifted cyclically in horizontal and vertical direction, respectively (see Fig. 3a). Submatrices of row i (column j) are shifted i (j) positions to the left (upwards). After the initial shifting, the first submatrix multiplication takes place (see Fig. 3b). The grey boxes indicate one pair of row and column to calculate the dot product from. This is done in parallel by all processes for each element of C. Within each step, a submatrix multiplication takes place followed by a row and column shift of A and B, respectively. In total, $\sqrt{p}$ steps are required until each process has calculated one submatrix of the result matrix C.

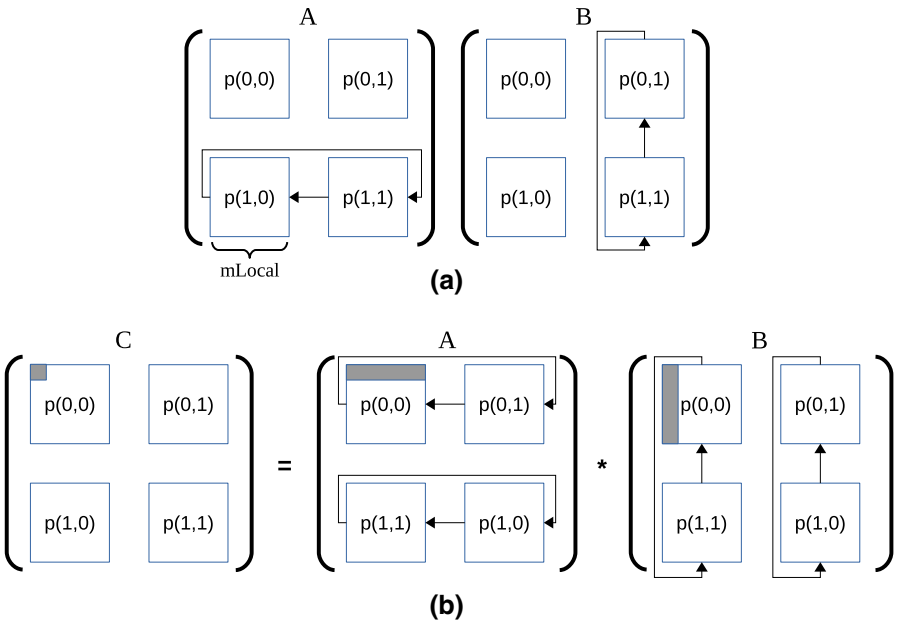

Fig. 3 Cannon's algorithm: **a** Initial shifting of submatrices. **b** Intermediate result of first submatrix multiplication and stepwise shifting of submatrices

```
template <typename T>
DMatrix<T>& matmult(DMatrix<T>& A, DMatrix<T>& B, DMatrix<T>* C) {
  // Initial shifting.
  A.rotateRows(&negate);
  B.rotateCols(&negate);

  for (int i = 0; i < A.getBlocksInRow(); i++) {
    DotProduct<T> dp(A, B);
    // Submatrix multiplication.
    C->mapIndexInPlace(dp);

    // Stepwise shifting.
    A.rotateRows(-1);
    B.rotateCols(-1);
  }
  return *C;
}
```

Listing 9 C++ Implementation of Cannon's algorithm using data parallel skeletons.

```
1  public DFMatrix matmult(DFMatrix A, DFMatrix B, DFMatrix C) {
2      // Initial shifting.
3      A.rotateRows(negate);
4      B.rotateCols(negate);
5
6      for (int i = 0; i < A.getBlocksInRow(); i++) {
7          DotProduct dp = new Dotproduct(A, B);
8          // Submatrix multiplication.
9          C.mapIndexInPlace(dp);
10
11         // Stepwise shifting.
12         A.rotateRows(-1);
13         B.rotateCols(-1);
14     }
15     return C;
16 }
```

Listing 10 Java Implementation of Cannon's algorithm using data parallel skeletons.

The implementation of the algorithm is presented in Listings 9 (C++) and 10 (Java). The initial shifting is performed by the communication skeletons `rotateRows` and `rotateCols`, respectively, in lines 3-4 (Java: 2-3). When called with a function or functor as argument, these skeletons calculate the number of positions each submatrix has to be shifted by applying the functor to the row and column indices of the submatrices, respectively. According to the function/functor `negate`, a submatrix of row i (column j) is shifted i (j) positions to the west (north). When called with the argument `-1` (lines 10-11, Java: 9-10), submatrices of row i (column j) are shifted one position to the west (north). In line 8 (Java: 7) the submatrix multiplication is performed by the `mapIndexInPlace` skeleton. It is called with a dot product functor presented in Listings 11 (C++) and 12 (Java). The suffix `Index` indicates that amongst the element itself also the indices of that element are passed to the user function. The suffix `InPlace` denotes that the skeleton works in-place, i.e. elements are overridden.

```
1  template <typename T>
2  struct DotProduct : public MapIndexFunctor<T, T> {
3    // Additional arguments
4    LMatrix<T> A, B;
5
6    dotproduct(DMatrix<T>& A_, DMatrix<T>& B_)
7      : A(A_), B(B_)
8    {
9    }
10
11   // User function
12   MSL_UFCT T operator()(int row, int col, T Cij) const
13   {
14     T sum = Cij;
15     for (int k = 0; k < this->mLocal; k++) {
16       sum += A[row][k] * B[k][col];
17     }
18     return sum;
19   }
20 };
```

Listing 11 Map kernel that calculates the dot product (C++).

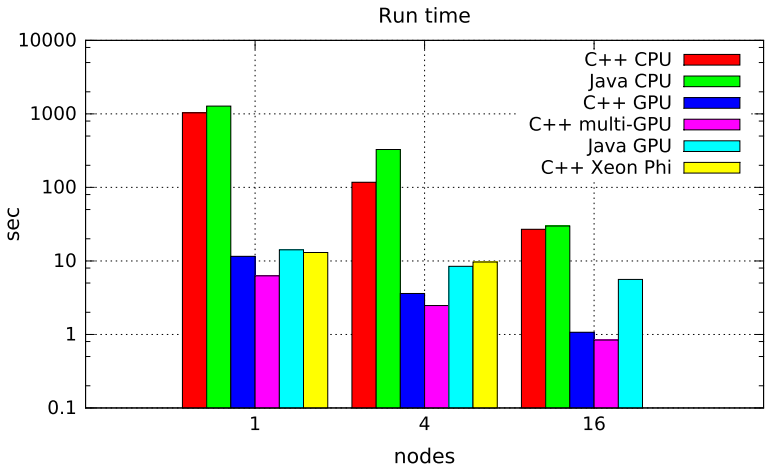

Fig. 4 Results of the Matrix multiplication benchmark with n = 8192. Run time (in seconds) is given on a logarithmic scale

```
1  class DotProduct extends MapIndexInPlaceKernel {
2      // Additional arguments
3      protected float[] A, B;
4
5      public Dotproduct(DFMatrix A, DFMatrix B) {
6          super();
7          this.A = A.getLocalPartition();
8          this.B = B.getLocalPartition();
9      }
10
11     // User function
12     public float mapIndexFunction(int row, int col, float Cij) {
13         float sum = Cij;
14         for (int k = 0; k < mLocal; k++) {
15             sum += A[row * mLocal + k] * B[k * mLocal + col];
16         }
17         return sum;
18     }
19 }
```

Listing 12 Map kernel that calculates the dot product (Java).

The dot product of corresponding rows of matrix A and columns of matrix B is calculated in lines 14-16 (Java: 12-14).

Performance results are reported in Fig. 4. As expected, the *C++ multi-GPU* configuration clearly performs best, followed by the *C++ GPU* configuration. The *Java GPU* and *C++ Xeon Phi* configurations are on a similar level, but clearly staying behind the C++ (multi-)GPU configurations. The two CPU configurations *C++ CPU* and *Java CPU* are trailing behind, with the C++ version having a slight edge over the Java version. Comparing the GPU configurations with the CPU configurations, one can observe speedups of about 90x on a single node. On higher node counts, the speedups decrease. This is due to CPU cache effects that result in super-linear speedups (about 40x on 16 nodes compared to a single node) for the *CPU* configurations.

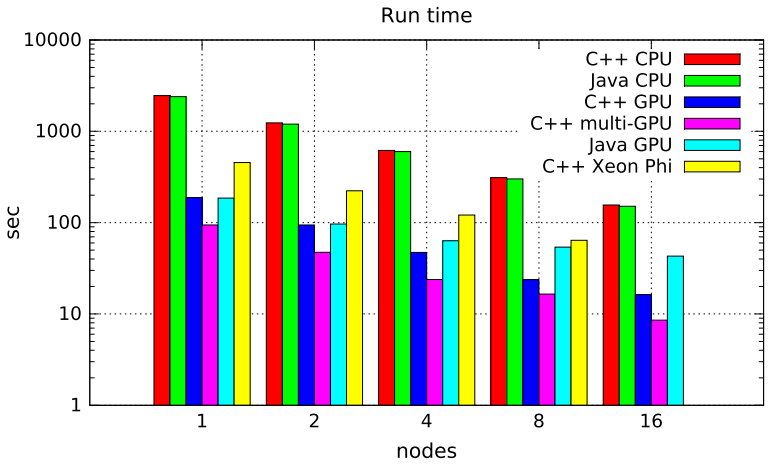

Fig. 5 Results of the N-Body benchmark with n = 500.000 over 10 time steps. Run time (in seconds) is given on a logarithmic scale

4.2 N-Body Computations

Performance results for the N-Body benchmark are reported in Fig. 5. Analogous to the matrix multiplication benchmark, the *C++ multi-GPU* configuration has a clear advantage over the other configurations. For lower node counts (1-2 nodes), the *C++ GPU* and *Java GPU* configurations are on the same level. However, for higher node counts (4-16) the C++ version delivers higher scalability, thus achieving better speedups and providing better performance. The *C++ Xeon Phi* configuration is about 2-3 times slower than the single GPU versions *C++ GPU* and *Java GPU*. The CPU configurations *C++ CPU* and *Java CPU* are on the same level for all node counts. However, they cannot compete with the accelerator configurations that achieve speedups of about 10-12x compared to the CPU configurations.

4.3 Shortest Paths

Performance results for the shortest paths benchmark are reported in Fig. 6. The results from this benchmark are very similar to the Matrix multiplication benchmark. For higher node counts super-linear speedups are noticeable. Again, this is likely due to cache effects. For the *Java GPU* configuration, however, higher node counts result in strongly decreasing speedups. The speedup when shifting from 4 to 16 nodes is only about 1.06, which is very close to no speedup at all. There is also close to no speedup recognizable for the *C++ Xeon Phi* configuration when shifting from 1 to 4 nodes.

4.4 Ray Tracing

Performance results for the ray tracing benchmark are reported in Fig. 7. Again, as expected, the *C++ multi-GPU* configuration has a clear edge over the other config-

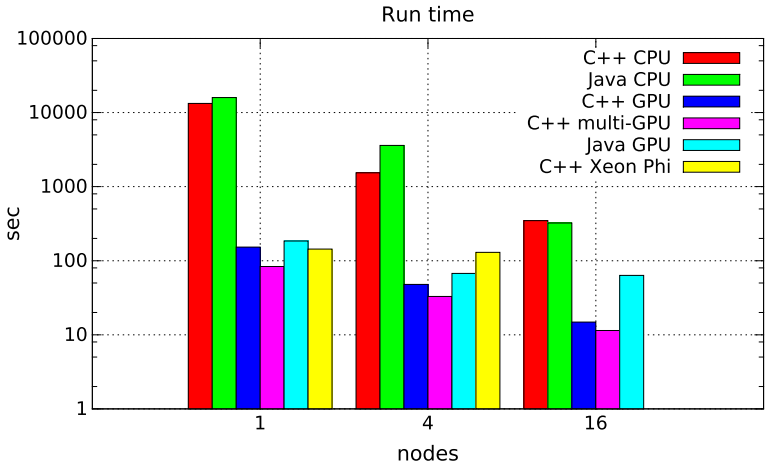

Fig. 6 Results of the Shortest paths benchmark with n = 8192. Run time (in seconds) is given on a logarithmic scale

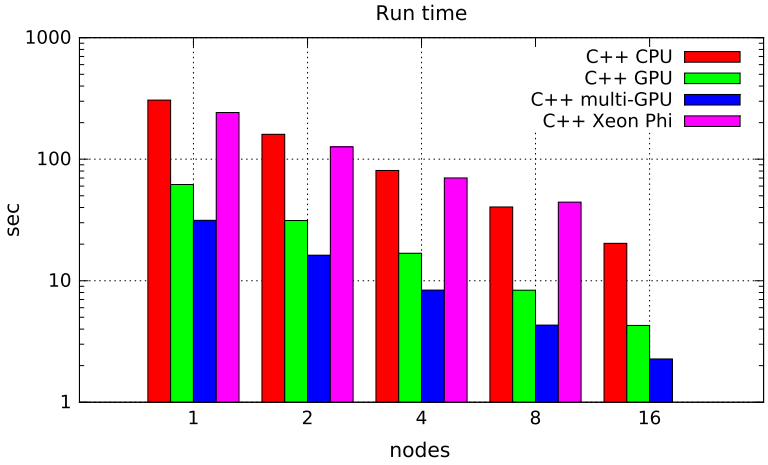

Fig. 7 Results of the Ray tracing benchmark with an image size of 2048 × 2048. Run time (in seconds) is given on a logarithmic scale

urations, being almost twice as fast as the *C++ GPU* configuration. Until up to 4 nodes, the *C++ Xeon Phi* configuration is about 10–20% faster than the *C++ CPU* configuration. At node count 8, the CPU configuration is even faster than the Xeon Phi configuration. This is due to the lack of auto-vectorization that was not feasible for this benchmark application. Inter-node speedups are close to ideal for each considered configuration.

All in all the results show that, performance-wise, there is not a big gap between the C++ and the Java implementation. Nevertheless, the multi-GPU configuration of the C++ implementation clearly outperforms any other configuration, which is not surprising. The Xeon Phi performance strongly depends on (auto)vectorization [24] and is only slightly better than the CPU performance.

5 Related Work

SkelCl [25] and SkePU [26] are C++ skeleton frameworks targeting multi-core, multi-GPU systems. While SkelCl is exclusively built on top of OpenCL, SkePU provides support for both CUDA and OpenCL. Additionally, SkePU allows for heterogeneous execution as well as performance-aware dynamic scheduling and load balancing. Both frameworks are currently limited to multi-core, multi-GPU systems and do not support distributed memory systems such as clusters. FastFlow [27] is a C++ framework providing high-level parallel programming patterns. It supports heterogeneous shared memory platforms equipped with hardware accelerators such as GPUs, Xeon Phi, and Tilera TILE64 as well as clusters of such platforms. As of this writing all three frameworks are limited to the C++ programming language and do not provide a Java implementation of their skeletons. To the best of our knowledge they also do not provide the functionality to arbitrarily add arguments to the user functions that are passed to the skeletons.

6 Conclusion

We have presented the implementation of data parallel skeletons with accelerator support in C++ and Java. It provides a high-level approach to simplify parallel and distributed programming. Applications developed with these skeletons are portable across a variety of platforms, including CPUs, GPUs and the Xeon Phi as well as clusters of such platforms. Programmers may specify whether they want to run a program either on CPUs only or with accelerator support. On the Java side, features such as e.g. its huge standard library, garbage collection, and reflection also contribute to making (parallel) programming more comfortable.

The benchmark results show that the Java and the C++ implementation offer comparable performance. However, there are still some restrictions on the Java side. Because of the restriction to primitive data types there is no opportunity to implement generic data structures, which results in code bloat. Also the restriction to single GPU systems needs to be addressed in future releases.

The Xeon Phi is sort of in-between CPUs and GPUs, performance-wise and in terms of programmability. Thanks to the Xeon Phi's support for existing parallelization tools and frameworks, support for this platform could seamlessly be added to the (C++) skeletons. Additional support for the Xeon Phi platform within the Java implementation will be targeted in the future.

References

1. Intel Corp: Intel Xeon Phi Coprocessor—The Architecture (Website). https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner. Accessed Jan 2016
2. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge (1989)
3. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput. **30**(3), 389–406 (2004)

4. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue **6**(2), 40–53 (2008)
5. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters. Int. J. High Perform. Comput. Netw. **7**(2), 129–138 (2012)
6. Ernsting, S., Kuchen, H.: Data parallel skeletons in java. In: Proceedings of the International Conference on Computational Science (ICCS), pp. 1817–1826. Omaha, Nebraska, USA (2012)
7. Ciechanowicz, P.: Algorithmic skeletons for general sparse matrices on multi-core processors. In: Proceedings of the 20th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS), pp. 188–197 (2008)
8. Poldner, M., Kuchen, H.: Skeletons for divide and conquer algorithms. In: Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN). ACTA Press (2008)
9. Poldner, M., Kuchen, H.: Algorithmic skeletons for branch and bound. In: Proceedings of the 1st International Conference on Software and Data Technology (ICSOFT), vol. 1, pp. 291–300 (2006)
10. Kuchen, H., Striegnitz, J.: Higher-order functions and partial applications for a C++ skeleton library. In: Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande, pp. 122–130. ACM (2002)
11. Ernsting, S., Kuchen, H.: Java implementation of data parallel skeletons on GPUs. In: Proceedings of the International Conference on Parallel Computing, ParCo 2015. Publication status, Edinburgh (2015) In press
12. Shafi, A., Carpenter, B., Baker, M.: Nested parallelism for multi-core HPC systems using Java. J. Parallel Distrib. Comput. **69**(6), 532–545 (2009)
13. Frost, G.: A parallel API. http://developer.amd.com/tools-and-sdks/opencl-zone/aparapi/ (2011). Accessed Jan 2016
14. Aparapi Github pages. https://aparapi.github.io/. Accessed Jan 2016
15. OpenCL Working Group: The OpenCL Specification, Version 1.2. (2011)
16. jCuda Website. http://jcuda.org. Accessed Jan 2016
17. Yan, Y., Grossman, M., Sarkar, V.: JCUDA: a programmer-friendly interface for accelerating java programs with Cuda. In: Euro-Par 2009 Parallel Processing, Lecture Notes in Computer Science, pp. 887–899. Springer (2009)
18. jOCL Website. http://jocl.org. Accessed Jan 2016
19. JogAmp Website. http://jogamp.org. Accessed Jan 2016
20. Docampo, J., Ramos, S., Taboada, G.L., Expósito, R.R., Touriño, J., Doallo, R.: Evaluation of java for general purpose GPU computing. In: 27th International Conference on Advanced Information Networking and Applications Workshops, pp. 1398–1404. Barcelona, Spain (2013)
21. Nvidia Corp: NVIDIA CUDA C Programming Guide 7.5. Nvidia Corporation (2015)
22. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc, Boston (1995)
23. Quinn, M.J.: Parallel Programming in C with MPI and OpenMP. McGraw-Hill Education Group, New York (2003)
24. Intel Corp: Vectorization Essentials (Website). https://software.intel.com/en-us/articles/vectorization-essential. Accessed Jan 2016
25. Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL—a portable skeleton library for high-level GPU programming. In: HIPS '11: Proceedings of the 16th IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments, Anchorage, AK, USA (2011)
26. Enmyren, J., Kessler, C.W.: SkePU: a multi-backend skeleton programming Library for multi-GPU systems. In: Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications. HLPP '10, pp. 5–14. ACM, New York, NY, USA (2010)
27. Aldinucci, M., Torquati, M., Drocco, M., Peretti Pezzi, G., Spampinato, C.: An Overview of FastFlow: Combining Pattern-Level Abstraction and Efficiency in GPGPUs. In: GPU Technology Conference (GTC 2014). San Jose, CA, USA (2014)