Optimizing Sequences of Skeleton Calls

Herbert Kuchen

University of Münster, Department of Information Systems
Leonardo Campus 3, D-48159 Münster, Germany
kuchen@uni-muenster.de

Abstract. Today, parallel programming is dominated by message passing libraries such as MPI. Algorithmic skeletons intend to simplify parallel programming by their expressive power. The idea is to offer typical parallel programming patterns as polymorphic higher-order functions which are efficiently implemented in parallel. Skeletons can be understood as a domain-specific language for parallel programming. In this chapter, we describe a set of data parallel skeletons in detail and investigate the potential of optimizing sequences of these skeletons by replacing them by more efficient sequences. Experimental results based on a draft implementation of our skeleton library are shown.

1 Introduction

Today, parallel programming of MIMD machines with distributed memory is typically based on message passing. Owing to the availability of standard message passing libraries such as MPI [1] [GL99], the resulting software is platform independent and efficient. However, the programming level is still rather low and programmers have to fight against low-level communication problems such as deadlocks. Moreover, the program is split into a set of processes which are assigned to the different processors. Like an ant, each process only has a local view of the overall activity. A global view of the overall computation only exists in the programmer's mind, and there is no way to express it more directly on this level.

Many approaches try to increase the level of parallel programming and to overcome the mentioned disadvantages. Here, we will focus on *algorithmic skeletons*, i.e. typical parallel programming patterns which are efficiently implemented on the available parallel machine and usually offered to the user as higher-order functions, which get the details of the specific application problem as argument functions (see e.g. [Co89,BK98,DP97]).

Object-oriented programmers often simulate higher-order functions using the "template" design pattern [GH95], where the general algorithmic structure is defined in an abstract superclass, while its subclasses provide the application-specific details by implementing certain template methods appropriately. In our approach, there is no need for static subclasses and the corresponding syntactic

[1] We assume some familiarity with MPI and C++.

overhead for providing them, but these details are just passed as arguments. Another way of simulating higher-order functions in an object-oriented language is to use the "command" design pattern [GH95]. Here, the "argument functions" are encapsulated by command objects. But still this pattern causes substantial overhead.

In our framework, a parallel computation consists of a sequence of calls to skeletons, possibly interleaved by some local computations. The computation is now seen from a global perspective. As explained in [Le04], skeletons can be understood as a domain-specific language for parallel programming. Several implementations of algorithmic skeletons are available. They differ in the kind of host language used and in the particular set of skeletons offered. Since higher-order functions are taken from functional languages, many approaches use such a language as host language [Da93,KP94,Sk94]. In order to increase the efficiency, imperative languages such as C and C++ have been extended by skeletons, too [BK96,BK98,DP97,FO92].

Depending on the kind of parallelism used, skeletons can be classified into *task parallel* and *data parallel* ones. In the first case, a skeleton (dynamically) creates a system of communicating processes by nesting predefined process topologies such as `pipeline`, `farm`, `parallel composition`, `divide&conquer`, and `branch&bound` [DP97,Co89,Da93,KC02]. In the second case, a skeleton works on a distributed data structure, performing the same operations on some or all elements of this data structure. Data-parallel skeletons, such as `map`, `fold` or `rotate` are used in [BK96,BK98,Da93,Da95,DP97,KP94].

Moreover, there are implementations offering skeletons as a library rather than as part of a new programming language. The approach described in the sequel is based on the skeleton library introduced in [Ku02,KC02,KS02] and on the corresponding C++ language binding. As pointed out by Smaragdakis [Sm04], C++ is particularly suited for domain-specific languages due to its meta-programming abilities. Our library provides task as well as data parallel skeletons, which can be combined based on the *two-tier model* taken from $\mathrm{P^3L}$ [DP97]. In general, a computation consists of nested task parallel constructs where an atomic task parallel computation can be sequential or data parallel. Purely data parallel and purely task parallel computations are special cases of this model. An advantage of the C++ binding is that the three important features needed for skeletons, namely higher-order functions (i.e. functions having functions as arguments), partial applications (i.e. the possibility to apply a function to less arguments than it needs and to supply the missing arguments later), and parametric polymorphism, can be implemented elegantly and efficiently in C++ using operator overloading and templates, respectively [St00,KS02].

Skeletons provide a global view of the computation which enables certain optimizations. In the spirit of the well-known Bird-Meertens formalism [Bi88,Bi89] [GL97], algebraic transformations on sequences of skeleton calls allow one to replace a sequence by a semantically equivalent but more efficient sequence. The investigation of such sequences of skeletons and their transformation will be the core of the current chapter. Similar transformations can be found in

[Go04,GL97,GW99,BG02]. There, you can also find correctness proofs of the transformations. All these transformations are expressed in terms of classical map, fold, and scan operations on lists, abstracting from data-distribution issues. These transformations are very elegant, but unfortunately exchanging lists between processors causes a substantial overhead for marshaling and unmarshaling. For this reason, we consider skeletons working on distributed arrays and matrices rather than on lists or sequences of values. Arrays and matrices are data structures frequently used in parallel programming, since their partitions can be easily shipped to other processors and since their elements can be efficiently accessed. Thus, skeletons working on them are particularly important in practice. Our skeletons explicitly control the data distribution, giving the programmer a tighter control of the computation. Moreover, our skeletons partly update an array or matrix in place. This is typically more efficient than generating a new data structure like the usual and well-known skeletons for lists do it. Updates in place are possible, since we are in an imperative / object oriented setting rather than in a functional language.

This chapter is organized as follows. In Section 2, we present the main concepts of our skeleton library and explain its skeletons in detail. In Section 3, we investigate the optimization of sequences of these skeletons and show some experimental results. Finally, in Section 4 we conclude.

2 The Skeleton Library

The skeleton library offers data parallel and task parallel skeletons. Data parallelism is based on a *distributed data structure*, which is manipulated by operations processing it as a whole and which happen to be implemented in parallel internally. Task parallelism is established by setting up a system of processes which communicate via streams of data. Such a system is not arbitrarily structured but constructed by nesting predefined process topologies such as farms and pipelines. Moreover, it is possible to nest task and data parallelism according to the mentioned two-tier model of P^{3}L, which allows atomic task parallel processes to use data parallelism inside. Here, we will focus on data parallelism and on the optimization of sequences of data parallel skeletons. Details on task parallel skeletons can be found in [KC02].

2.1 Overview of Data Parallel Skeletons

As mentioned above, data parallelism is based on a *distributed data structure* (or several of them). This data structure is manipulated by operations such as `map` and `fold` (explained below) which process it as a whole and which happen to be implemented in parallel internally. These operations can be interleaved with sequential computations working on non-distributed data. In fact, the programmer views the computation as a *sequence* of parallel operations. Conceptually, this is almost as easy as sequential programming. Synchronization and communication problems like deadlocks and starvation cannot occur, since each skeleton encapsulates the passing of low-level messages in a safe way. Currently, two main distributed data structures are offered by the library, namely:

```
template <class E> class DistributedArray{...}
template <class E> class DistributedMatrix{...}
```

where `E` is the type of the elements of the distributed data structure. Moreover, there are variants for sparse arrays and matrices, which we will not consider here. By instantiating the template parameter `E`, arbitrary element types can be generated. This shows one of the major features of distributed data structures and their operations in our framework. They are *polymorphic.* Moreover, a distributed data structure is split into several partitions, each of which is assigned to one processor participating in the data parallel computation. Currently, only block partitioning is supported. Future extensions by other partitioning schemes are planned.

Roughly, two classes of data parallel skeletons can be distinguished: computation skeletons and communication skeletons. *Computation skeletons* process the elements of a distributed data structure in parallel. Typical examples are the following methods in class `DistributedArray<E>`:

```
void mapIndexInPlace(E (*f)(int,E))
E fold(E (*f)(E,E))
```

`A.mapIndexInPlace(g)` applies a binary function `g` to each index position i and the corresponding array element $\mathtt{A}_i$ of a distributed array `A` and replaces $\mathtt{A}_i$ by $\mathtt{g}(i,\mathtt{A}_i)$. `A.fold(h)` combines all the elements of `A` successively by an associative binary function `h`. E.g. `A.fold(plus)` computes the sum of all elements of `A` (provided that `E plus(E,E)` adds two elements).

In order to prevent C++ experts from being annoyed, let us briefly point out that the types of the skeletons are in fact more general than shown above. E.g. the real type of `mapIndexInPlace` is:

```
template <class T> void mapIndexInPlace(T f)
```

Thus, the parameter `f` can not only be a C++ function of the mentioned type but also a so-called function object, i.e. an object representing a function of the corresponding type. In particular, such a function object can represent a partial application as we will explain below.

Communication consists of the exchange of the partitions of a distributed data structure between all processors participating in the data parallel computation. In order to avoid inefficiency, there is no implicit communication e.g. by accessing elements of remote partitions like in HPF [Ko94] or Pooma [Ka98], but the programmer has to control communication explicitly by using skeletons. Since there are no individual messages but only coordinated exchanges of partitions, deadlocks cannot occur. The most frequently used communication skeleton is

```
void permutePartition(int (*f)(int))
```

`A.permutePartition(f)` sends every partition $\mathtt{A}_{[j]}$ (located at processor j) to processor `f(`j`)`. `f` needs to be bijective, which is checked at runtime. Some other communication skeletons correspond to MPI collective operations, e.g. `allToAll`, `broadcastPartition`, and `gather`. `A.broadcastPartition(j)`, for instance, replaces every partition of `A` by the one found at processor `j`.

Moreover, there are operations which allow one to access attributes of the local partition of a distributed data structure, e.g. `get`, `getLocalGlobal`, and `isLocal` (see Fig. 1) fetch an element of the local partition (using global or local indexing explained below, or combinations of both) and check whether a considered element is locally available, respectively. These operations are not skeletons but frequently used when implementing an argument function of a skeleton.

At first, skeletons such as `fold` and `scan` might seem equivalent to the corresponding MPI collective operations `MPI_Reduce` and `MPI_Scan`. However, they are more powerful due to the fact that the argument functions of all skeletons can be *partial applications* rather than just C++ functions. A skeleton essentially defines some parallel algorithmic structure, where the details can be fixed by appropriate argument functions. With partial applications as argument functions, these details can depend themselves on parameters, which are computed at runtime. For instance, in

```
template <class E> E plus3(E x, E y, E z){return x+y+z;}
...
int x = ... some computation ...;
A.mapIndexInPlace(curry(plus3)(x));
```

each element $\mathtt{A}_i$ of the distributed array `A` is replaced by `plus3(x,`i`,`$\mathtt{A}_i$`)`. The "magic" `curry` function has been taken from the C++ template library Fact [St00], which offers functional-programming capabilities in C++ similar to FC++ [Sm04,MS00]. `curry` transforms an ordinary C++ function (such as `plus3`) into a function (object) which can be partially applied. Here, the computed value of `x` is given directly as a parameter to `plus3`. This results in a binary function, and this is exactly, what the `mapIndexInPlace` skeleton needs as parameter. Thus, `mapIndexInPlace` will supply the missing two arguments. Hence, partial applications as arguments provide the application-specific details to a generic skeleton and turn it into a specific parallel algorithm.

The code fragment in Fig. 1 taken from [Ku03] shows how skeletons and partial applications can be used in a more interesting example. It is a parallel implementation of (simplified) Gaussian elimination (ignoring potential division by 0 problems and numerical aspects). For the moment, this example shall only give you a flavor of the approach. We suggest that you return to the example after having read Subsections 2.2 and 2.3, which explain the skeletons in detail. Then, you will be able to completely understand the code.

The core is the function `gauss` in lines 11-16. In line 12 a $p\times(n+1)$ distributed matrix `Pivot` is constructed and initialized with 0.0. According to the two last arguments of the constructor, it is split into $p \times 1$ partitions, i.e. each of the p processors gets exactly one row. The $n \times (n+1)$ coefficient matrix `A` is assumed

```
 1 double copyPivot(const DistributedMatrix<double>& A,
 2                  int k, int i, int j, double Aij){
 3   return A.isLocal(k,k) ? A.get(k,j)/A.get(k,k) : 0.0;}

 4 void pivotOp(const DistributedMatrix<double>& Pivot,
 5              int rows, int firstrow, int k, double** LA){
 6   for (int l=0; l<rows; l++){
 7     double Alk = LA[l][k];
 8     for (int j=k; j<=Problemsize; j++)
 9       if (firstrow+l == k) LA[l][j] = Pivot.getLocalGlobal(0,j);
10       else  LA[l][j] -=  Alk * Pivot.getLocalGlobal(0,j);}}

11 void gauss(DistributedMatrix<double>& A){
12   DistributedMatrix<double> Pivot(p,n+1,0.0,p,1);
13   for (int k=0; k<Problemsize; k++){
14     Pivot.mapIndexInPlace(curry(copyPivot)(A)(k));
15     Pivot.broadcastRow(k);
16     A.mapPartitionInPlace(curry(pivotOp)(Pivot,n/p,A.getFirstRow(),k));}}
```

Fig. 1. Gaussian elimination with skeletons.

to be split into $p \times 1$ partitions consisting of n/p consecutive rows. For instance, it could have been created by using the constructor:

```
DistributedMatrix<double> A(n,n+1,& init,p,1);
```

where the function `init` tells how to initialize each element of `A`. In each iteration of the loop in lines 13-16, the pivot row is divided by the pivot element and copied to a corresponding row of the matrix `Pivot` (line 14). In fact, the `mapIndexInPlace` skeleton tries to do this on every processor, but only at the processor owing the row it will actually happen. `copyPivot` sets all other elements to 0.0. The pivot row is then broadcast to every other row of matrix `Pivot` (line 15). Thus, every processor now has the pivot row. This row is then used by the corresponding processor to perform the pivot operation at every locally available element of matrix `A` (line 16). This is done by applying the `mapPartitionInPlace` skeleton to the pivot operation. More precisely, it is applied to a partial application of the C++ function `pivotOp` to the `Pivot` matrix and to three other arguments, which are a bit technical, namely to the number n/p of rows of `A` each processor has, to the index of the first row of the local partition of `A`, and to the index k of the pivot column (and row). The remaining fourth argument of `pivotOp`, namely the local partition of `A`, will be supplied by the `mapPartitionInPlace` skeleton. When applied to the mentioned arguments and to a partition of `A`, `pivotOp` (lines 4-10) performs the well-known pivot operation:

$$\mathtt{A}_{l,j} = \begin{cases} \mathtt{A}_{k,j}/\mathtt{A}_{k,k}, & \text{if local row } l \text{ is the pivot row} \\ \mathtt{A}_{l,j} - \mathtt{A}_{l,j} \cdot \mathtt{A}_{k,j}/\mathtt{A}_{k,k}, & \text{otherwise} \end{cases}$$

for all elements $\mathtt{A}_{l,j}$ of the local partition. Note that `pivotOp` does not compute $\mathtt{A}_{k,j}/\mathtt{A}_{k,k}$ itself, but fetches it from the j-th element of the locally available row of `Pivot`. Since all rows of matrix `Pivot` are identical after the broadcast, the index of the row does not matter and we may take the first locally available row (in fact the only one), namely the one with local index 0. Thus, we find $\mathtt{A}_{k,j}/\mathtt{A}_{k,k}$ in `Pivot.getLocalGlobal(0,j)`. Note that 0 is a local index referring to the local partition of `Pivot`, while j is a global one referring to matrix `Pivot` as a whole. Our skeleton library provides operations which can access array and matrix elements using local, global, and mixed indexing. The user may pick the most convenient one. Note that parallel programming based on message passing libraries such as MPI only provides local indexing.

It is also worth mentioning that for the above example the "non-standard" map operation `mapPartitionInPlace` is clearly more efficient (but slightly less elegant) than more classic map operations. `mapPartitionInPlace` manipulates a whole partition at once rather than a single element. This allows one to ignore such elements of a partition which need no processing. In the example, these are the elements to the left of the pivot column. Using classic map operations, one would have to apply the identity function to these elements, which causes substantial overhead.

2.2 Data Parallel Skeletons for Distributed Arrays

After this overview of data parallel skeletons, let us now consider them in more detail. This will set up the scene for the optimizations of sequences of these skeletons, which will be considered in Section 3. We will focus on skeletons for distributed arrays here. Skeletons for distributed matrices are very similar, and we will only sketch them in the next subsection. The following operations are needed to start and terminate a skeleton-based computation, respectively.

`void InitSkeletons(int argc, char* argv[])`
: `InitSkeletons` needs to be called before the first skeleton is used. It initializes the global variables `sk_myid` and `sk_numprocs` (see below). The parameters `argc` and `argv` of `main` have to be passed on to `InitSkeletons`.

`void TerminateSkeletons()`
: `TerminateSkeletons` needs to be called after the last skeleton has been applied. It terminates the skeleton computation cleanly.

The following global variables can be used by skeleton-based computations:

`int sk_myid`
: This variable contains the number of the processor on which the considered computation takes place. It is often used in argument functions of skeletons, i.e. in functions telling how a skeleton should behave on the locally available data.

`int sk_numprocs`
: contains the number of available processors.

All skeletons for distributed arrays (DAs) are methods of the corresponding class

```
template <class E> class DistributedArray {...}
```

It offers the public methods shown below. For every method which has a C++ function as an argument, there is an additional variant of it which may instead use a partial application with a corresponding type. As mentioned above, a partial application is generated by applying the function `curry` to a C++ function or by applying an existing partial application to additional arguments. For instance, in addition to

```
void mapInPlace(E (*f)(E))
```

(explained below) there is some variant of type

```
template <class F>
void mapInPlace(const Fct1<E,E,F>& f)
```

which can take a partial application rather than a C++ function as argument. Thus, if the C++ functions `succ` (successor function) and `add` (addition) have been previously defined, all elements of a distributed array `A` can be incremented by one either by using the first variant of `mapInPlace`

```
A.mapInPlace(succ)
```

or by the second variant

```
A.mapInPlace(curry(add)(1))
```

Of course, the second is more flexible, since the arguments of a partial application are computed at runtime. `Fct1<A,R,F>` is the type of a partial application representing a unary function with argument type `A` and result type `R`. The third template parameter `F` determines the computation rule needed to evaluate the application of the partial application to the missing arguments (see [KS02] for details). The type `Fct1<A,R,F>` is of internal use only. The user does not need to care about it. She only has to remember that skeletons may have partial applications instead of C++ functions as arguments and that these partial applications are created by applying the special, predefined function `curry` to a C++ function and by applying the result successively to additional arguments. Of course, there are also types for partial applications representing functions of arbitrary arity. In general, a partial application of type Fct$i$<A$_1$,...,A$_n$,R,F> represents a n-ary function with argument types A$_1$,...,A$_n$ and result type `R`, for $i \in \mathbb{N}$.

Some of the following methods depend on some preconditions. An exception is thrown, if they are not met. p denotes the number of processors collaborating in the data parallel computation on the considered distributed array.

Constructors.

`DistributedArray(int size, E (*f)(int))`
creates a distributed array (DA) with `size` elements. The i-th element is initialized with f(i) for $i = 0, \ldots, \texttt{size} - 1$. The DA is partitioned into

equally sized blocks, each of which is given to one of the p processors. More precisely, the j-th participating processor gets the elements with indices $j \cdot \texttt{size}/p, \ldots, (j+1) \cdot \texttt{size}/p - 1$, where $j = 0, \ldots, p-1$. We assume that p divides `size`.

`DistributedArray(int size, E initial)`
: This constructor works as the previous. However, every array element is initialized with the same value `initial`. There are more constructors, which are omitted here in order to save space.

Operations for Accessing Attributes of a Distributed Array. The following operations access properties of the local partition of a distributed array. They are not skeletons themselves, but are frequently used when implementing argument functions of skeletons.

`int getFirst()`
: delivers the index of the first locally available element of the DA.

`int getSize()`
: returns the `size` (i.e. total number of elements) of the DA.

`int getLocalSize()`
: returns the number of locally available elements.

`bool isLocal(int i)`
: tells whether the `i`-th element is locally available.

`void setLocal(int i, E v)`
: sets the value of the `i`-th locally available element to `v`. Note that the index `i` is referring to the local partition and not to the DA as a whole.
Precondition: $0 \leq \texttt{i} \leq \texttt{getLocalSize}() - 1$.

`void set(int i, E v)`
: sets the value of the `i`-th element to `v`.
Precondition: $j \cdot \texttt{size}/p \leq \texttt{i} < (j+1) \cdot \texttt{size}/p$, if the local partition is the j-th partition of the DA $(0 \leq j < p)$.

`E getLocal(int i)`
: returns the value of the `i`-th locally available element.
Precondition: $0 \leq \texttt{i} \leq \texttt{getLocalSize}() - 1$.

`E get(int i)`
: delivers the value of the `i`-th element of the DA.
Precondition: $j \cdot \texttt{size}/p \leq \texttt{i} < (j+1) \cdot \texttt{size}/p$, where the local partition is the j-th partition of the DA $(0 \leq j < p)$.

Obviously, there are auxiliary functions which access a distributed array via a global index and others which access it via an index relative to the start of the local partition. Using global indexing is usually more elegant, while local indexing can be sometimes more efficient. MPI-based programs use local indexing only. Our skeleton library offers both, and it even allows to mix them as in the Gaussian elimination example (Fig. 1, lines 9 and 10). All these methods and most skeletons are inlined. Thus, there is no overhead for function calls when using them.

Map Operations. The `map` function is the most well-known and most frequently used higher-order function in functional languages. In its original formulation, it is used to a apply an unary argument function to every element of a list and to construct a new list from the results. In the context of arrays and imperative or object oriented programming, this original formulation is of limited use. The computation related to an array element A_i often not only depends on A_i itself but also on its index i. In contrast to functional languages, imperative and object oriented languages allow also to update the element in place. Thus, there is no need to construct a new array, and time and space can be saved. Consequently, our library offers several variants of `map`, namely with and without considering the index and with and without update in place. In fact, `mapIndexInPlace` turned out to be the most useful one in many example applications. Some of the following variants of `map` require a template parameter `R`, which is the element type of the resulting distributed array.

`template <class R> DistributedArray<R> map(R (*f)(E))`
returns a new DA, where element i has value `f(get(`i`))`
($0 \leq i \leq$ `getSize`$() - 1$). It is partitioned as the considered DA.

`template <class R> DistributedArray<R> mapIndex(R (*f)(int,E))`
returns a new DA, where element i has value `f(`i`,get(`i`))`
($0 \leq i \leq$ `getSize`$() - 1$). It is partitioned as the considered DA.

`void mapInPlace(E (*f)(E))`
replaces the value of each DA element with index i by `f(get(`i`))`,
where $0 \leq i \leq$ `getSize`$() - 1$.

`void mapIndexInPlace(E (*f)(int,E))`
replaces the value of each DA element with index i by `f(`i`,get(`i`))`,
where $0 \leq i \leq$ `getSize`$() - 1$.

`void mapPartitionInPlace(void (*f)(E*))`
replaces each partition P of the DA by `f(`P`)`.

Zip Operations. `zip` works on two equally partitioned distributed arrays of the same size and combines elements at corresponding positions. Just as for `map`, there are variants taking the index into account and variants updating one of the arrays in place. In the following `<class E2>` and `<class R>` are additional template parameters of the corresponding method.

```
DistributedArray<R> zipWith(const DistributedArray<E2>& b,
                                                  R (*f)(E,E2))
```
returns a new DA, where element i has value `f(get(`i`),b.get(`i`))` and
$0 \leq i \leq$ `getSize`$() - 1$. It is partitioned as the considered DA.

`void zipWithInPlace(DistributedArray<E2>& b,E (*f)(E,E2))`
replaces the value of each DA element with index i by `f(get(`i`),b.get(`i`))`,
where $0 \leq i \leq$ `getSize`$() - 1$.

```
DistributedArray<R> zipWithIndex(const DistributedArray<E2>& b,
                                              R (*f)(int,E,E2))
```
returns a new DA, where element i has value `f(`i`,get(`i`),b.get(`i`))` and
$0 \leq i \leq$ `getSize`$() - 1$. It is partitioned as the considered DA.

`void zipWithIndexInPlace(const DistributedArray<E2>& b,`
`E (*f)(int,E,E2))`

replaces the value of each DA element with index i by `f(`i`,get(`i`),b.get(`i`))`, where $0 \leq i \leq \texttt{getSize}() - 1$.

Both, `map` and `zipWith` (and their variants) require no communication.

Fold and Scan. `fold` (also known as reduction) and `scan` (often called parallel prefix) are computation skeletons, which require some communication internally. They combine the elements of a distributed array by an associative binary function.

`E fold(E (*f)(E,E))`

combines all elements of the DA by the associative binary operation `f`, i.e. it computes `f(get(0),f(get(1),...f(get(`n`-2),get(`n`-1))...))` where n = `getSize()`. Precondition: `f` is associative (this is not checked).

`void scan(E (*f)(E,E))`

replaces every element with index i by `f(get(0),f(get(1),...f(get(`i`-1),get(`i`))...))`, where $0 \leq i \leq \texttt{getSize}() - 1$.
Precondition: `f` is associative (this is not checked).

Both, `fold` and `scan`, require $\Theta(\log\ p)$ messages (of size $size(E)$ and $n/p \cdot size(E)$, respectively) per processor [BK98].

Communication Operations. As mentioned already, communication skeletons rearrange the partitions of a distributed data structure. This is done in such a way that the index range assigned to each processor is not changed. Only the corresponding values are replaced by those from another partition.

`void permutePartition(int (*f)(int))`

replaces every partition `f(`i`)` by partition i where $0 \leq i \leq p - 1$.
Precondition: $\texttt{f} : \{0, \ldots, p-1\} \rightarrow \{0, \ldots, p-1\}$ is bijective. This is checked at runtime.

`void permute(int (*f)(int))`

replaces every element with index `f(`i`)` by the element with index i where $0 \leq i \leq \texttt{getSize}() - 1$.
Precondition: $\texttt{f} : \{0, \ldots, \texttt{getSize}()-1\} \rightarrow \{0, \ldots, \texttt{getSize}()-1\}$ is bijective. This is checked at runtime.

`void broadcastPartition(int i)`

replaces every partition of the DA by partition `i`.
Precondition: $0 \leq i \leq p - 1$.

`void broadcast(int index)`

replaces every element of the DA by the element with index `i`.
Precondition: $0 \leq i \leq \texttt{getSize}() - 1$.

`void allToAll(const DistributedArray<int *>& Index, E dummy)`
: uses parts of the local partition to replace parts of each other partition. `Index` is a DA of type int$[p+1]$ indicating which part goes where. Depending on `Index`, some processors may get more elements than others. The unused elements of each partition are filled with `dummy`.

Another communication skeleton, `gather`, is a bit special, since it does not rearrange a distributed array, but it copies it to an ordinary (non-distributed) array. Since all non-distributed data structures and the corresponding computations are replicated on each processor participating in a data parallel computation [BK98], `gather` corresponds to the MPI operation `MPI_Allgather` rather than to `MPI_Gather`.

`void gather(E b[])`
: copies all elements of the considered DA to the non-distributed array `b`.

`permute` and `permutePartition` require a single send and receive per processor, while `broadcast`, `broadcastPartition`, and `gather` need $\Theta(\log\ p)$ communication steps (with message sizes $size(E)$ and $n/p \cdot size(E)$, respectively). `allToAll` requires $\Theta(p)$ communication steps.

Other Operations.

`DistributedArray<E> copy()`
: generates a copy of the considered DA.

`void show()`
: prints the DA on standard output.

2.3 Data Parallel Skeletons for Distributed Matrices

The operations for distributed matrices (DM) are similar to those for distributed arrays. Thus, we will sketch them only briefly here. The constructors specify the number of rows and columns, tell how the elements shall be initialized, and give the number of partitions in vertical and horizontal direction. E.g. the constructor

```
DistributedMatrix(int n, int m, E x0, int v, int h)
```

constructs a distributed `n`×`m` matrix, where every element is initialized with `x0`. The matrix is split into `v`×`h` partitions. As for DAs, there are other constructors which compute the value of each element depending on its position using a C++ function or function object.

The operations for accessing attributes of a DM now have to take both dimensions into account. Consequently, there are operations delivering the number of (local or global) rows and columns, the index of the first locally available row and column, and so on. As shown in the Gaussian-elimination example, the operations for accessing an element of a DM can use any combination of local and global indexing. For instance, `M.get(`i`,`j`)` fetches (global) element $\mathtt{M}_{i,j}$ of DM

`M`, while `M.getLocalGlobal(`l`,`j`)` accesses the element in local row l and global column j.

There are also straightforward variants of map, zip, fold, and scan operations as well as communication operations such as permutation, broadcast, and all-to-all, e.g.

```
void permutePartition(int (*f)(int,int), int (*g)(int,int))
```

replaces every partition $P_{i',j'}$ by partition $P_{i,j}$, where $i' = \mathtt{f}(i,j)$ and $j' = \mathtt{g}(i,j)$. It is checked at runtime whether the resulting mapping is bijective.

Additionally, there are rotation operations, which allow to rotate the partitions of a DM cyclically in vertical or horizontal direction. In fact, these are special cases of `permutePartition`, e.g.

```
void rotateRows(int (*f)(int))
```

cyclically rotates the partitions in row i by $\mathtt{f}(i)$ partitions to the right. Negative values for $\mathtt{f}(i)$ correspond to rotations to the left.

3 Optimization of Sequences of Skeletons

As explained above, the user of the data parallel skeletons has a global view of the parallel computation. An advantage of this global view is that it simplifies global optimizations compared to the local view of MPI-based processes. In particular it enables algebraic transformations of sequences of skeletons, much in the spirit of the well-known Bird-Meertens formalism (see e.g. [Bi88,Bi89,GL97]). For instance

```
A.mapIndexInPlace(f);
A.mapIndexInPlace(g);
```

can be replaced by

```
A.mapIndexInPlace(curry(compose)(g)(f));
```

where `compose` is assumed to be a C++ implementation of a composition function, for instance by:

```
template <class C1, class C2, class C3>
inline C3 compose(C3 (*f)(C2), C2 (*g)(C1), C1 x){return f(g(x));}
```

In the sequel, we will discuss a few such transformations, however without attempting to generate a complete set of them. The idea is to consider a few optimizations and check out their impact based on some experimental results for a few small benchmarks. The final aim will be to create an automatic optimizer which detects sequences of skeletons and replaces them by some semantically equivalent but more efficient alternative sequences of skeletons, as in the above example. However, this aim is far beyond the scope of this chapter. The mentioned optimizer will need a cost model of each skeleton in order to estimate the

example	n	original	combined	speedup
`permutePartition` ∘ `permutePartition`	2^7	37.00 μs	16.43 μs	2.25
`mapIndexInPlace` ∘ `mapIndexInPlace`	2^7	0.69 μs	0.40 μs	1.73
`mapIndexInPlace` ∘ `fold`	2^7	144.82 μs	144.60 μs	1.00
`mapIndexInPlace` ∘ `scan`	2^7	144.52 μs	144.48 μs	1.00
`multiMapIndexInPlace`	2^7	0.67 μs	0.65 μs	1.03
`mapIndexInPlace` ∘ `permutePartition`	2^7	16.60 μs	31.43 μs	0.53
`permutePartition` ∘ `permutePartition`	2^{19}	10.76 ms	5.32 ms	2.02
`mapIndexInPlace` ∘ `mapIndexInPlace`	2^{19}	2.79 ms	1.59 ms	1.75
`mapIndexInPlace` ∘ `fold`	2^{19}	2.97 ms	2.67 ms	1.11
`mapIndexInPlace` ∘ `scan`	2^{19}	6.01 ms	5.83 ms	1.03
`multiMapIndexInPlace`	2^{19}	2.77 ms	2.62 ms	1.06
`mapIndexInPlace` ∘ `permutePartition`	2^{19}	6.68 ms	5.72 ms	1.17

Table 1. Runtimes on 4 processors for sequences of skeletons working on a distributed array of n elements and corresponding speedups.

$n \setminus p$	4	8	16
16	1.64	1.91	1.89
128	1.71	1.92	1.70
1024	1.48	1.81	1.72

Table 2. Speedups for the combination of two rotations of a $n \times n$ matrix depending on the number p of processors.

runtimes of alternative sequences of skeletons. A corresponding cost model for most of the mentioned skeletons can be found in [BK98]. Other cost models for skeletons and similar approaches like the bulk synchronous parallel (BSP) model [SH97] are presented in e.g. [MC95,SD98]. A BSP computation is a sequence of so-called supersteps each consisting of a computation and a global communication phase. This restricted model facilitates the estimation of costs. Our data parallel skeletons are similarly easy to handle.

The transformation in the above example is an instance of a more general pattern, where a sequence of the same skeleton is replaced by a single occurrence of this skeleton. This pattern can also be applied to e.g. other variants of `map`, `permutePartition`, `rotateRows`, and `rotateCols`.

We have compared the runtimes of the original and the transformed sequence of skeleton calls on the IBM cluster at the University of Münster. This machine [ZC03] has 94 compute nodes, each with a 2.4 GHz Intel Xeon processor, 512 KB L2 Cache, and 1 GB memory. The nodes are connected by a Myrinet and running RedHat Linux 7.3 and the MPICH-gm implementation of MPI.

Fig. 2 and (partly) Table 1 show the speedups which could be achieved for different array sizes and numbers of processors. It is interesting to note that the problem size and the number of processors have little influence on the speedup.

By composing two permutation skeletons, the expected speedup of about 2 could be obtained. Combining two rotations of a distributed matrix leads to speedups between 1.5 and 1.9 (see Table 2). There are more opportunities for

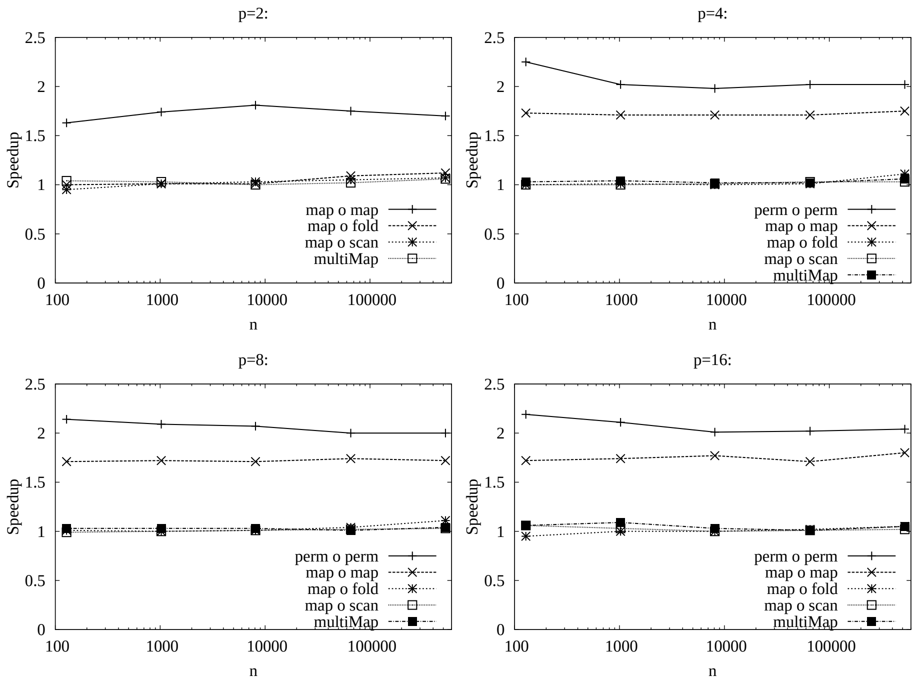

Fig. 2. Speedups for the combination of skeletons on 2, 4, 8, and 16 processors. On 2 processors, the combination of permutations does not make sense. "map" stands for `mapIndexInPlace` and "perm" for `permutePartition`.

combining sequences of communication skeletons as shown in Table 3. Transformations involving rotations are shown for distributed matrices rather than arrays. Since these transformations can be rarely applied in practice (as explained below), we do not consider them further here. Moreover, we omit the corresponding correctness proofs, since the transformations are rather straightforward.

Combining two computation skeletons such as `map` corresponds to *loop fusion* (see e.g. [ZC90]). For `mapIndexInPlace` this roughly means that internally

```
for(int i = 0; i<localsize; i++)
   a[i] = f(i+first,a[i]);
for(int i = 0; i<localsize; i++)
   a[i] = g(i+first,a[i]);
```

is replaced by

```
for(int i = 0; i<localsize; i++)
   a[i] = g(i+first,f(i+first,a[i]));
```

Obviously, one assignment and the overhead for the control of one loop are saved. It is well-known that loop fusion is only possible if the considered loops do not

original	combined

A.permute(f);
A.permute(g);
	A.permute(curry(compose)(g)(f));
	(analogously for permutePartition)

A.permute(f);
A.broadcast(k);
	A.broadcast(f^{-1}(k));
	(analogously for permutePartition)

A.broadcast(k);
A.permute(f);
	A.broadcast(k);
	(analogously for permutePartition)

A.broadcast(k);
A.broadcast(i);
	A.broadcast(k);

M.permutePartition(f,g);
M.rotateRows(h);
	M.permutePartition(curry(compose)(h')(f),g);

M.rotateRows(h);
M.permutePartition(f,g);
	M.permutePartition(curry(compose)(f)(h'),g);

M.permutePartition(f,g);
M.rotateCols(h);
	M.permutePartition(f,curry(compose)(h')(g));

M.rotateCols(h);
M.permutePartition(f,g);
	M.permutePartition(f,curry(compose)(g)(h'));

M.broadcast(i,j);
M.rotateRows(h);
	M.broadcast(i,j);
	(analogously for rotateCols)

M.rotateRows(h);
M.broadcast(i,j);
	M.broadcast(h'$^{-1}$(i),j);
	(analogously for rotateCols)

M.rotateRows(f);
M.rotateRows(g);
	M.rotateRows(curry(compose)(g)(f));
	(analogously for rotateCols)

M.rotateRows(f);
M.rotateCols(g);
	M.permutePartition(f',g');

M.rotateCols(f);
M.rotateRows(g);
	M.permutePartition(g',f');

Table 3. Transformations for sequences of communication skeletons working on a distributed array `A` and a distributed matrix `M`, respectively. $\mathtt{h}'(x) = x + \mathtt{h}(x)\ \mathit{mod}\ m$, where m is the number of partitions in each row (or column). `f` and `g` are analogously transformed to `f'` and `g'`, respectively.

depend on each other. When combining two calls of `mapIndexInPlace`, this is typically the case. Note however that `f` and `g` may be partial applications having array `a` as parameter. In this case, loop fusion cannot (easily) be applied. Moreover, loop fusion is not always a good idea. For instance, in some cases the caching behavior may deteriorate to an extent that the saved overhead for loop control is outweighed. In our experiments (see Fig. 2), we could achieve a speedup of about 1.7, when composing two occurrences of map (in fact `mapIndexInPlace`). However, this is only possible for simple argument functions. If each call to the argument function causes a complex computation, the advantage of loop fusion almost disappears. If we choose an argument function, where each call causes a loop of 1000 iterations to be executed (instead of one iteration as in Fig. 2), the speedup decreases to about 1.05 (see Table 4).

A similar kind of loop fusion can also be performed when combining different skeletons, e.g. `mapIndexInPlace` and `fold` or `scan` (see Fig. 2). However, this

k	n	original	combined	speedup
1	2^7	0.69 μs	0.40 μs	1.73
1000	2^7	21.23 μs	19.79 μs	1.07
1	2^{19}	2.79 ms	1.59 ms	1.75
1000	2^{19}	83.00 ms	80.12 ms	1.04

Table 4. Runtimes for the combination of two occurrences of `mapIndexInPlace` working on a distributed array with n elements on 4 processors depending on the number k of iterations executed for each call to the argument function of `mapIndexInPlace`.

requires not only a transformation of the application program, but the skeleton library also has to be extended by the required combined skeletons. For instance,

```
A.mapIndexInPlace(f);
result = A.fold(g);
```

is replaced by

```
result = A.mapIndexInPlaceFold(f,g);
```

in the application program, and the new skeleton `mapIndexInPlaceFold` with obvious meaning is added to the library. Combining `map` and `scan` is done analogously. As Fig. 2 shows, speedups of up to 1.1 can be obtained for big arrays, and a bit less for smaller arrays. Again, a simple argument function was used here. With more complex argument functions, the speedup decreases. The same approach can be applied for fusing:

- two calls to `zipWith` (and variants),
- `zipWith` and `map`, `fold`, or `scan`,
- a communication skeleton (such as `permute`, `broadcast` and `rotate`) and a computation skeleton (such as `map`, `fold`, and `scan`) (and vice versa).

Let us consider the latter in more detail. If implemented in a clever way, it may (in addition to loop fusion) allow one to *overlap communication and computation* and hence provide a new source for improvements. For the combination of `mapIndexInPlace` and `permutePartition`, we observed a speedup of up to 1.25 (see Fig. 3). Here, the speedup heavily depends on a good balance between computation and communication. If the arrays are too small and the amount of computation performed by the argument function of `mapIndexInPlace` is not sufficient, the overhead of sending more (but smaller) messages does not pay off, and a slowdown may occur. Also if the amount of computation exceeds that of communication, the speedup decreases again. With an optimal balance between computation and communication (and a more sophisticated implementation of the `mapIndexInPlaceFold` skeleton) speedups up to 2 are theoretically possible.

We have investigated a few example applications (see [Ku03]) such as matrix multiplication, FFT, Gaussian elimination, all pairs shortest paths, samplesort, TSP (traveling salesperson problem), and bitonic sort in order to check which of the optimizations discussed above can be applied more or less frequently in

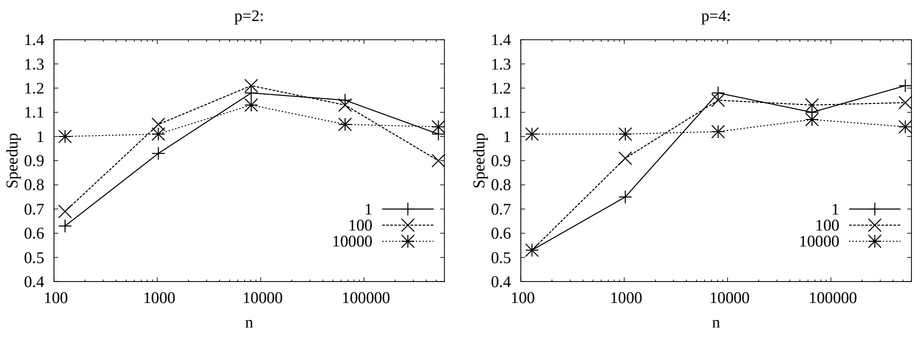

Fig. 3. Speedups for the combination of `mapIndexInPlace` and `permutePartition` on 2 and 4 processors depending on the number of iterations (1-10000) of a simple loop that each call to the argument function requires.

practice. Unfortunately, there was no opportunity to combine communication skeletons. This is not too surprising. If such a combination is possible, it is typically performed by the programmer directly, and there is no need for an additional optimization. However automatically generated code as well as programs written by less skilled programmers might still benefit from such transformations. Moreover, we did find opportunities for combining map and fold, e.g. in the TSP example [KC02], and also for combining two occurrences of map. In our examples, there was no instance of a combination of map and scan. However, you can find one in [Go04]. Very often, there were sequences consisting of a communication skeleton followed by a computation skeleton or vice versa as in our `mapIndexInPlace`$\circ$`permutePartition` example (see Fig. 2). This is not surprising, since this alternation of communication and computation is what the BSP approach [SH97] is all about; and using data parallel skeletons is somehow similar to BSP.

In several examples, we also found sequences of maps, which were manipulating different arrays. Here, the above transformation could not be applied. However, since we noted the importance of this pattern in practice, we have added a new skeleton `multiMap` (with the usual variants such as `multiMapIndexInPlace`) to our library. `multiMap` combines two maps working on two equally partitioned distributed arrays (or matrices) of the same size. Internally, this results in a similar loop fusion as for the combination of maps working on the same array. For instance

```
A.mapIndexInPlace(f);
B.mapIndexInPlace(g);
```

can be replaced by: `multiMapIndexInPlace(A,f,B,g);`

In our example code, this transformation caused a speedup of up to 1.08 (see Fig. 2). Since `multiMap` is not particularly nice, it is supposed to be used by an optimizer only, and not explained to the user.

4 Conclusions and Future Work

We have explained the data parallel skeletons of our C++ skeleton library in detail. Moreover, we have considered the optimization of sequences of these skeletons. When combining sequences of communication skeletons such as permutations, broadcasts, and rotations, we observed the expected speedup. Unfortunately, there are little opportunities in practical applications, where communication skeletons can be successfully fused. For computation skeletons such as map and fold and, in particular, for sequences of communication and computation skeletons, the situation is different. Practical applications often contain such sequences, such that the corresponding transformation can be applied. This essentially leads to some kind of loop fusion, and, in the case of combined communication and computation skeletons, to overlapping communication and computation. Speedups of a few percent could be achieved this way.

In the future, we plan to develop an automatic optimizer, which replaces sequences of skeletons by semantically equivalent but more efficient sequences. The purpose of the presented case study was to check out the potential of such an optimizer.

Acknowledgements

I would like to thank the anonymous referees for lots of helpful suggestions.

References

[BG02] Bischof, H., Gorlatch, S.: Double-Scan: Introducing and Implementing a New Data-Parallel Skeleton. In Monien, B., Feldmann, R., eds.: Euro-Par'02. LNCS 2400, Springer-Verlag (2002) 640-647

[Bi88] Bird, R.: Lectures on Constructive Functional Programming, In Broy, M., ed.: Constructive Methods in Computing Science, NATO ASI Series. Springer-Verlag (1988) 151-216

[Bi89] Bird, R.: Algebraic identities for program calculation. The Computer Journal 32(2) (1989) 122-126

[BK96] Botorog, G.H., Kuchen, H.: Efficient Parallel Programming with Algorithmic Skeletons. In Bougé, L. et al., eds.: Euro-Par'96. LNCS 1123, Springer-Verlag (1996) 718-731

[BK98] Botorog, G.H., Kuchen, H.: Efficient High-Level Parallel Programming, Theoretical Computer Science 196 (1998) 71-107

[Co89] Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press (1989)

[Da93] Darlington, J., Field, A.J., Harrison T.G., et al: Parallel Programming Using Skeleton Functions. PARLE'93. LNCS 694, Springer-Verlag (1993) 146-160

[Da95] Darlington, J., Guo, Y., To, H.W., Yang, J.: Functional Skeletons for Parallel Coordination. In Hardidi, S., Magnusson, P.: Euro-Par'95. LNCS 966, Springer-Verlag (1995) 55-66

[DP97] Danelutto, M., Pasqualetti, F., Pelagatti S.: Skeletons for Data Parallelism in p3l. In Lengauer, C., Griebl, M., Gorlatch, S.: Euro-Par'97. LNCS 1300, Springer-Verlag (1997) 619-628

[FO92] Foster, I., Olson, R., Tuecke, S.: Productive Parallel Programming: The PCN Approach. Scientific Programming 1(1) (1992) 51-66

[GH95] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1995)

[GL99] Gropp, W., Lusk, E., Skjellum, A.: Using MPI. MIT Press (1999)

[GL97] Gorlatch, S., Lengauer, C.: (De)Composition Rules for Parallel Scan and Reduction, 3rd Int. Working Conf. on Massively Parallel Programming Models (MPPM'97). IEEE (1997) 23-32

[Go04] Gorlatch, S.: Optimizing Compositions of Components in Parallel and Distributed Programming (2004) In this volume

[GW99] Gorlatch, S., Wedler, C., Lengauer, C.: Optimization Rules for Programming with Collective Operations, 13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing (1999) 492-499

[Ka98] Karmesin, S., et al.: Array Design and Expression Evaluation in POOMA II. ISCOPE'98 (1998) 231-238

[KC02] Kuchen, H., Cole, M.: The Integration of Task and Data Parallel Skeletons. Parallel Processing Letters 12(2) (2002) 141-155

[Ko94] Koelbl, C.H., et al.: The High Performance Fortran Handbook. Scientific and Engineering Computation. MIT Press (1994)

[KP94] Kuchen, H., Plasmeijer, R., Stoltze, H.: Efficient Distributed Memory Implementation of a Data Parallel Functional Language. PARLE'94. LNCS 817, Springer-Verlag (1994) 466-475

[KS02] Kuchen, H., Striegnitz, J.: Higher-Order Functions and Partial Applications for a C++ Skeleton Library. Joint ACM Java Grande & ISCOPE Conference. ACM (2002) 122-130

[Ku02] Kuchen, H.: A Skeleton Library. In Monien, B., Feldmann, R.: Euro-Par'02. LNCS 2400, Springer-Verlag (2002) 620-629

[Ku03] Kuchen, H.: The Skeleton Library Web Pages. http://danae.uni-muenster.de/lehre/kuchen/Skeletons/

[Le04] Lengauer, C.: Program Optimization in the Domain of High-Performance Parallelism. (2004) In this volume

[MC95] W. McColl: Scalable Computing. In van Leuwen, J., ed.: Computer Science Today, LNCS 1000, Springer-Verlag (1995) 46-61

[MS00] McNamara, B., Smaragdakis, Y.: Functional Programming in C++. ICFP'00. ACM (2000) 118-129

[SD98] Skillicorn, D., Danelutto, M., Pelagatti, S., Zavanella, A.: Optimising Data-Parallel Programs Using the BSP Cost Model. In Pritchard, D., Reeve, J.: Euro-Par'98. LNCS 1470, Springer-Verlag (1998) 698-703

[SH97] Skillicorn, D., Hill, J.M.D., McColl, W.: Questions and Answers about BSP. Scientific Programming 6(3) (1997) 249-274

[Sk94] Skillicorn, D.: Foundations of Parallel Programming. Cambridge U. Press (1994)

[Sm04] Smaragdakis, Y.: A Personal Outlook of Generator Research (2004) In this volume

[St00] Striegnitz, J.: Making C++ Ready for Algorithmic Skeletons. Tech. Report IB-2000-08, http://www.fz-juelich.de/zam/docs/autoren/striegnitz.html

[ZC90] Zima, H.P., Chapman, B.M.: Supercompilers for Parallel and Vector Computers. ACM Press/Addison-Wesley (1990)

[ZC03] ZIV-Cluster: http://zivcluster.uni-muenster.de/