ERCIS

European Research Center for Information Systems

Working Paper No. 7

P. Ciechanowicz, M. Poldner & H. Kuchen

The Münster Skeleton Library Muesli – A Comprehensive Overview

ERCIS – European Research Center for Information Systems

Editors J. Becker, K. Backhaus, H. L. Grob, B. Hellingrath, T. Hoeren, S. Klein, H. Kuchen, U. Müller-Funk, U. W. Thonemann, G. Vossen

University of Münster Germany

Working Papers

ERCIS – European Research Center for Information Systems
Editors: J. Becker, K. Backhaus, H. L. Grob, B. Hellingrath, T. Hoeren, S. Klein, H. Kuchen, U. Müller-Funk, U. W. Thonemann, G. Vossen

Working Paper No. 7

The Münster Skeleton Library *Muesli* - A Comprehensive Overview

Philipp Ciechanowicz, Michael Poldner, Herbert Kuchen

ISSN 1614-7448

cite as: Ciechanowicz, P.; Poldner, M.; Kuchen, H.: The Münster Skeleton Library *Muesli* - A Comprehensive Overview. In: Working Papers, European Research Center for Information Systems No. 7. Eds.: Becker, J. et al. Münster 2009.

Table of Contents

List of Figures

- 4

Chapter 1

Introduction

Parallel programming of MIMD machines with distributed memory is typically based on standard message passing libraries such as MPI [1], which leads to platform independent and efficient software. However, the programming level is rather low, and the development of a parallel application can be a quite complicated task: Programmers have to think about how to decompose the problem and distribute it across the collaborating processes, develop a communication protocol to synchronize processes by passing messages, and eventually integrate the partial solutions into a final one. Due to the low programming level, programmers have to fight against low-level communication problems such as deadlocks, starvation, or overlapping communication and computation. 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 globel 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. Debugging such applications borders on looking for the famous needle in the haystack since errors do not occur deterministically. Although debuggers for parallel applications have been developed, implementing a parallel program remains tedious and error-prone.

For this reasons many approaches have been suggested which provide a higher level of abstraction and an easier program development. The skeletal approach to parallel programming proposes that typical communication and computation patterns for parallel programming should be offered to the user as predefined and application independent components which can be combined and nested by the user. These components are referred to as algorithmic skeletons [2, 3, 4, 5, 6, 7, 8]. By providing application-specific parameters to these skeletons, the user can adapt them to the considered parallel application without bothering about low-level implementation details such as synchronization, interprocessor communication, load balancing, and data distribution. Efficient implementations of many skeletons exist, such that the resulting parallel application can be almost as efficient as one based on low-level message passing.

From a conceptual point of view, skeletons can be divided into data parallel and task parallel ones. Data parallel skeletons such as *map* and *fold* operate on a distributed data structure and manipulate its elements in parallel [8, 9, 10, 11, 12, 13]. Task parallel skeletons create a system of processes communicating via streams of data by nesting predefined process topologies such as *pipeline*, *farm*, *branch&bound*, and *divide&conquer* [2, 3, 5, 8, 11, 14].

Skeletons can be understood as domain-specific languages 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 [11, 13, 15]. In order to increase the efficiency, imperative languages such as C and C++ have been extended by skeletons, too [8, 9, 10, 16, 17, 18]. Moreover, there are implementations offering skeletons as

a library rather than as part of a new programming language [3, 6, 16].

The Edinburgh Skeleton Library *eSkel* [3, 5, 14, 19, 20] is a structured parallel programming library written in C on top of MPI. In its current version eSkel provides the following skeletons: Pipeline, Farm, Butterfly (Divide & Conquer), Deal (sort of Farm) and HaloSwap. The library is built on top of two fundamental concepts, namely the *nesting mode* and the *interaction mode*. The former can either be set to *transient* or *persistent* and affects the nesting of (different or identical) skeletons. The latter can either be set to *implicit* or *explicit* and is used to control the temporal and spatial interaction of each skeleton. eSkel does not support any data parallel skeletons.

The MaLLBa [2] library offers skeletons for solving optimization problems. It has been developed by three working groups at Malaga, La Laguna and Barcelona and offers exact, heuristic and hybrid solving techniques. In order to solve a problem exactly, skeletons for Divide & Conquer, Branch & Bound and Dynamic Programming are supported. Heuristic techniques are facilitated by skeletons for hill climbing, metropolis, simulated annealing, tabu search, genetic and memetic algorithms. Hybrid approaches combine exact and heuristic techniques, e.g. genetic algorithms and simulated annealing, branch and bound and simulated annealing, or genetic algorithms and branch and bound. Just like eSkel, MALLBA does not support any data parallel skeletons at all.

One of the few libraries to support both task and data parallel skeletons is the Pisa Parallel Programming Language P^{3}L [21, 22, 23]. The computational model underlying P^{3}L provides a few primitive skeletons, abstracting both task and data parallelism, and the ability to nest them to express complex application structures. The language provides two task parallel skeletons in terms of a Farm and a Pipe and numerous data parallel skeletons like *map*, *reduce*, and *scan*. Additionally, P^{3}L offers so-called control skeletons such as Loop and Seq. Since P^{3}L does not support different data structures, all data parallel skeletons are limited to work on arrays of data elements.

Lithium [24, 25] is skeleton library written in Java and includes common task and data parallel skeletons such as *Pipeline*, *Farm*, *Map*, *Reduce* and *Divide and Conquer*. *Muskel* [26] is a pure Java/RMI skeleton library derived from *Lithium* that targets workstation clusters, networks, and grids. Both libraries are implemented exploiting (macro) data flow technology, rather than the more usual skeleton technology relying on the use of implementation templates.

Moreover, the skeleton library SkeTo [27] is to be mentioned. The resourch group from Tokyo focuses on implementing different data structures for data parallel skeletons like lists, trees, and (sparse) matrices [28]. Task parallel skeletons are not provided. As a unique feature, SkeTo offers a fusion based optimization mechanism which merges successive function calls into a single one such that the execution time can be reduced significantly. Although a data structure for sparse matrices is supported, this data structure is not as flexible as our approach since neither the data type, the compression nor the distribution scheme for the submatrices (SkeTo calls them *blocks*) can be user-defined (cf. Section 3.4).

The Data parallel Template Library (*DatTel*) [16, 17] extends the Standard Template Library (STL) of the C++ standard by parallel skeletons. An application based on *DatTel* can run on both distributed and shared memory machines without any changes to the source code. In [29, 30], skeleton-based programming models for grid computing are presented. The system discussed in [30] is based on Java and its Remote Method Invocation (RMI) API. The *HOC-SA* approach [29] provides skeletons in a similar manner. However, this system is designed to be used in the Open Grid Services Architectur (OGSA) [31]. The skeletal parallelism homepage [32] contains links to virtualy all groups and projects working on skeletons.

The approach described in the sequel is an enhancement of the skeleton library introduced in [5, 6, 33] and based on the results of our latest research [34, 35, 36, 37, 38, 39, 40, 41]. The Münster Skeleton Library *Muesli* offers data and task parallel skeletons in terms of a program library implemented in C++ and uses the current standard message passing interface MPI to

facilitate the communication between processes. Based on the two-tier model taken from P^{3}L [8], atomic task parallel processes can internally be data parallel. Programs developed with the Münster skeleton library are platform independent (due to MPI) and can be executed on every system for which an MPI implementation and a C++ compiler exist. *Muesli* is intended to form some sort of additional abstraction layer. From a programmer's perspective, the whole inter-process communication is completely transparent, since these things are hidden and taken care of inside our skeleton library. Thus, the programmer of a parallel application is freed from bothering with low-level implementation details such that the probability of error is reduced to a minimum. In short, the benefits of our skeleton library are:

- Sequential programming style. Developing a parallel application is almost as easy as developing a serial one. This is due to the fact that the programmer needs not bother about coordinating processes since these details are taken care of inside our skeleton library.
- Easy development of parallel applications. Since *Muesli* is provided as a C++ library, every programmer familiar with this language can develop a parallel application with marginal effort. Thus, there is no need to learn language extensions or even a new programming language at all.
- Safe development of parallel applications. Since the whole communication is encapsulated inside our skeleton library, the programmer needs not know anything about MPI routines. Thus, synchronization and space allocation problems, starvation, and deadlocks cannot occur.

The remainder of this paper is structured as follows. In Section 2 the key concepts and fundamental ideas of *Muesli* are described, such as polymorphic types, higher-order functions, partial applications, and serialization. Section 3 presents the different distributed data structures (distributed array, distributed matrix, and distributed sparse matrix) and the corresponding data parallel skeletons provided by *Muesli* in detail. Section 4 introduces the atomic task parallel processes and task parallel skeletons. Finally, Section 5 discusses selective implementaion details.

Chapter 2

Concepts

The following sections describe key concepts and fundamental ideas underlying the implementation of our skeleton library. First of all, we recapitulate polymorphic types and their implementation in C++, since they are extensively used throughout our library (cf. Section 2.1). Next, two concepts which emerged from the mathematical study of functions are presented, namely higher-order functions and partial applications (cf. Sections 2.2 and 2.3). Both of them are essential in order to understand the idea of algorithmic skeletons. Finally, we describe how our library automatically serializes user-defined types (cf. Section 2.4)

2.1 Polymorphic Types

Type polymorphism is a programming language feature which allows to build generic programs, i.e. programs that are independent of concrete types. This property is especially useful when building container classes which store a certain number of elements of arbitrary type `E`. Since one cannot anticipate the concrete type to be used with the data structure, without polymorphic types one would have to provide different variants for each function. For sure, this approach is far from being elegant. But even more disadvantageous is the fact, that the implementation will not work with new types added later on. While this is hardly acceptable, type polymorphism allows one to implement only a single generic variant of a function which will also work with future types.

C++ supports parametric polymorphism by means of so-called *templates* [42]. A template definition consists of a list of type variables followed by the definition of a function, a class member function, or a class. Here is an example for a function template to compute the maximum of two values:

```
template<class E> E max(E a, E b) {
  return (a < b) ? b : a;
}
```

The template parameter `E` is a placeholder for any built-in or user-defined type. The substitution with concrete types is transparent to the user. `max` can be used as if it was an ordinary function. For instance, the compiler automatically generates, i.e. instantiates, `int max(int, int)` when calling `max(5, 3)`. Different instances of `max` are distinguished by the overloading mechanism of C++.

Our library uses polymorphic types within all distributed data structures. Here, the user can

define the type of the elements to be stored by instantiating the appropriate type. All integral numerical types of C++ as well as structs and unions, which are defined using these integral numerical types, are supported. Letting the user choose the appropriate type is important due to the following reasons: Firstly, this approach avoids unnecessary waste of memory. Allocating space for a **`long double`** variable where **`short`** would suffice is hardly efficient. Secondly, using a fixed type would force the user to make downcasts when using our data structure. Since this is rather unaesthetic we decided to use type polymorphism in order to avoid inefficiencies and provide maximum flexibility.

2.2 Higher-Order Functions

Higher-order functions are functions that take functions as arguments and/or return functions as result. Although the term is not embraced in the C++ community, the Standard Template Library (STL) contains lots of examples such as `for_each` and `transform` [43]. The template mechanism and the chance to overload the parenthesis operator prove to be powerful toys to express higher-order functions in C++. Here is a simple example for a higher-order function that takes a unary function `f` and a value `x` of type `E` as arguments and applies `f` to `x` twice:

```
template<class E, class F>
E applyTwice(F f, E x) {
  return f(f(x));
}
```

The template parameter `F` is a placeholder for arbitrary C++ types that support the function call syntax such as ordinary function pointers and classes that provide a parenthesis operator:

```
class FunctionalClass {
  inline int operator()(int x) const {
    return x * x;
  }
}
```

The overloaded function call operator permits objects of type `FunctionalClass` to be used as if they were ordinary C++ functions. As the name suggests, such objects are often referred to as *functional objects* or simply *functors*.

Our library makes heavy use of higher-order functions, since they are an essential requirement for algorithmic skeletons. Prior to explaining skeletons in detail, we will introduce another important feature of our library which is closely coupled with higher-order functions, namely partial applications.

2.3 Partial Applications

Passing less than n arguments to an n-ary function is called *partial application*. Semantically, partially applying an n-ary function to k arguments with $k < n$ means binding its first arguments to some fixed values, thereby yielding a $(n - k)$-ary function. From a theoretical viewpoint, the key concept to enable partial application is *currying* [33]. Currying has its roots in the mathematical study of functions where it has been shown that it is sufficient to restrict attention to unary functions: Every function $f\colon A_1 \times \ldots \times A_n \to R$ can be turned into a unary function $g\colon A_1 \to \ldots \to A_n \to R$ with $\to$ being associative to the right. g is called the curried form of

f. Passing a single argument of type A_1 to g results in a unary function to which we may pass a value of type A_2 to obtain another unary function to which we may pass a value of type A_3 to retrieve yet another unary function and so on until we obtain a unary function that takes a value of type A_n which returns a value of type R. Of course, $g(a_1) \ldots (a_n)$ leads to the same result as $f(a_1, \ldots, a_n)$. Although g is unary, we may think of it as being n-ary with the special property that it is capable of taking its arguments one at a time: $g(A_1) \ldots (A_k)$ with $k < n$ yields a valid result, namely a unary function. Semantically, this unary function is the curried form of an $(n-k)$-ary function h that is defined as follows:

$$h\colon A_{k+1} \times \ldots \times A_n \to R$$

Thus, we retrieve the curried form of f, whereas the first k arguments have been bound to $a_1, \ldots, a_k$:

$$h(a_{k+1}, \ldots, a_n) = f(\underbrace{a_1, \ldots, a_k}_{\text{constant}}, a_{k+1}, \ldots, a_n)$$

Consider Listing 1 for a more concrete example. In order to use partial applications, we first need to include the header file `curry.h` which defines the `curry` function (cf. line 1). The binary function `f` to curry is defined in lines 3 - 5 and simply returns the sum of the given **`int`** arguments `a` and `b`. The important part of this example is shown in lines 8–10. Here, the binary function `f` is transformed into a partial application `g` by calling `curry(f)` (cf. line 8). Then, `g` is applied to the argument 3 by calling `g(3)` (cf. line 9). This returns a function `h` whose definition is also given in Listing 1. The difference between `f` and `h` is, that the first argument of `f` is preset to 3 such that `h` consumes one argument less than `f`. In other words, `f` is only partially evaluated by presetting `a` to 3 thus returning the partial application `h`. Finally, `h` is applied to the argument 4 (cf. line 9). By doing so, `x` gets the value 7.

```
1  #include "curry.h"                          // partial application of f:
2                                              //
3  int f(int a, int b) {                       // int h(int b) {
4    return a + b;                             //   return 3 + b;
5  }                                           // }
6
7  void main(int argc, char** argv) {
8    Fct2<int, int, int, int (*)(int, int)> g = curry(f);
9    Fct1<int, int, int (*)(int)> h = g(3);
10   int x = h(4);
11 }
```

Listing 1: Partial applications and currying with Muesli.

Partial applications are implemented using the functional classes `Fct0` ... `Fct6` defined in the file `curry.h`. The number denotes how many parameters the functor expects. Again, consider the binary function `f`. The function call `curry(f)(3)` curries the function and applies the argument 3 to it, thus returning a `Fct1` object. These objects come into play when using a skeleton function which expects a partial application as an argument. Let's take, for example, the following skeleton function implemented for the distributed array:

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

The function sets each element a_i of the distributed array to `f`(a_i). In order to do so, the user must pass the functional object `f` which is of type `Fct1<E, R, F>&`. The template parameter `E` denotes

the type of the parameter of the function, `R` denotes its return type and `F` denotes the type of the function, for example `int (*)(int)`. Fortunately, one needs not pass `F` in most cases since the compiler can derive it by means of the other template parameters.

2.4 Serialization

Object serialization is an important issue in the context of data storage and transmission due to the fact that the objects to be interchanged among task or data parallel skeletons often are of dynamic size or contain pointer structures. In this case, it is necessary to write the object data to a contiguous memory block before sending them over the network, and to restore the object at receivers' side. In contrast to languages such as (Object) Pascal or Java, C++ does not inherently support object serialization. To perform this type of operation, *Muesli* provides the abstract class `MSL_Serializable`.

```
class MSL_Serializable {
    public:
        MSL_Serializable() {}
        virtual ~MSL_Serializable() {}
        virtual void reduce(void* pBuffer, int bufferSize) = 0;
        virtual void expand(void* pBuffer, int bufferSize) = 0;
        virtual int getSize() = 0;
};
```

Each class, whose instances represent objects which have to be transmitted serialized, must be derived from `MSL_Serializable` and implement the inherited methods `reduce`, `expand`, and `getSize`. Objects which are not derived from `MSL_Serializable` are supposed to be already serialized, such as pointerless C++ structures or basic data types. In this case, there is no need for an additional serialization (the object data is already stored in a continous memory block), and the objects can be directly transmitted to the receiver. Further implementation details are discussed in section 5.1 and 6.2.

Chapter 3

Distributed Data Structures

3.1 Concepts

3.1.1 Skeletons

In *Muesli*, data parallelism is based on a distributed data structure. A distributed data structure is split into several partitions, each of which is assigned to exclusively one process participating in the data parallel computation. The data struture is manipulated by operations which process it as a whole and are internally implemented in parallel. Nevertheless, these operations can be interleaved with sequential computations working on non-distributed data. The programmer views the computation as a sequence of parallel operations. Conceptually, this is almost as easy as sequential programming. Communication problems such as deadlocks or starvation cannot occur. Currently, three distributed data structures are offered by our library, namely:

```
template<class E> class DistributedArray  { ... }
template<class E> class DistributedMatrix { ... }
template<class E, class S, class D> class DistributedSparseMatrix { ... }
```

The template parameter `E` denotes the type of the elements of the distributed data structure, the parameters `S` and `D` are explained in Section 3.4. By instantiating the parameter `E`, arbitrary element types can be generated as pointed out in Section 2.1. Note that a (sparse) matrix is not the same as an array of arrays, since it allows an arbitrary partitioning into submatrices while the latter would only allow to build blocks of sequences of rows or columns, each consisting of a full array. Moreover, there are additional operations for (sparse) matrices such as horizontal or vertical rotations which only make sense for a matrix.

The main objective when designing our distributed data structures was to support data parallel skeletons such as *fold*, *map*, *zip*, and their variants. Skeletons can best be described as abstract computational patterns which relieve the programmer of a parallel application from low-level problems such as synchronizing processes or preventing deadlocks. Each skeleton expects a function pointer or a partial application *f* as an argument which concretizes its abstract behaviour. The skeletons are defined as follows:

- *fold* reduces all elements of the distributed data structure by repeatedly applying *f* to them. *f* needs to be associative and bijective.
- *map* replaces each element of the distributed data structure by the result of applying *f* to it.

- *scan* replaces each element of the distributed data structure by the result of folding all previous elements with *f*.
- *zip* combines two structurally identical data structures by merging corresponding elements with *f*.

`f` can either be a partial application or an ordinary C++ function pointer. In the latter case, the function pointer must comply with the following syntax: ⟨*return type*⟩ `(*`⟨*function name*⟩`)` `(`⟨*list of parameter types*⟩`)`. The syntax may seem a little odd but is easily explained. Let's consider, for example, the signature of the skeleton *mapInPlace* defined in the class `DistributedArray<E>`:

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

Let `A` be a distributed array and `f` be a function of type `E (*)(E)`. `A.mapIndexInPlace(f)` applies `f` to each element a_i of the distributed array and replaces a_i by `f`(a_i). In other words, the *map* skeleton iterates over each element of the distributed array and applies the given function `f` to it. `f` in turn expects one argument of type `E` which represents the element `f` is applied to. Obviously, `f` must return a value which again is of type `E`. Note that the signature of `f` must exactly match the one defined in the signature of `mapInPlace`. Otherwise, the code cannot be compiled.

At first, some skeletons such as *fold* or *scan* might seem equivalent to the corresponding MPI collective operation `MPI_Reduce` or `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 ordinary functions. The ability to pass user-defined functions in order to manipulate the whole data structure in parallel is the main distinctive feature of our implementation. This concept is very powerful, since the user is not bound to predefined functions but rather can define her own. For each skeleton, there is a variant which expects a partial application as parameter. Due to the C++ overloading mechanism, we could give it the same name and the same "remaining signature". Thus, the user does not have to bother whether or not the skeleton is used with a function or a partial application as argument. If a skeleton has more than one argument function, any combination of functions and partial applications as arguments is possible.

As already mentioned, all skeletons are offered in different variants regarding the signature of the argument function. The type of the necessary function can be deduced from the suffix of the skeleton function. The full list of data parallel skeleton functions including all variants of *fold*, *map*, *scan*, and *zip* are presented in Sections 3.2–3.4.

Index denotes, that the argument function of the skeleton must provide additional parameter(s) for the global index(es) of the current element. In case of the distributed array, one parameter for the global index is sufficient. In case of the distributed (sparse) matrix, two parameters must be provided, namely one for the global row and one for the global column index. Thus, the argument function may incorporate the index(es) in its calculations but is of course not obliged to do so. Since these skeletons return a new distributed data structure as opposed to overwriting elements, all of them are annotated with the **`const`** modifier. The following example creates a distributed array `A` with 10 elements and initializes each of them with 1 (cf. Section 3.2). Then, a second distributed array `B` is created whose elements are initialized with the result of applying `f` to the corresponding elements of `A`:

```
double f(double value, int rowIndex, int columnIndex) {
  return 2 * value + rowIndex - columnIndex;
}

void main(int argc, char** argv) {
  DistributedArray<double> A(10, 1);
  DistributedArray<double> B = A.mapIndex(&f);
}
```

InPlace denotes, that the value of each element is overwritten with the result of applying the skeleton argument function `f` to each element as opposed to creating a new distributed data structure and returning a reference to it. Since these skeletons overwrite elements as opposed to creating a new distributed data structure, all of them return **`void`**. The following example creates a distributed matrix `A` of size 4×4 and initializes each element with 0. The distributed matrix is divided into partitions of size 2×2. Then, each element is replaced by applying `f` to it:

```
double f(double value) {
  return 2 * value;
}

void main(int argc, char** argv) {
  DistributedMatrix<double> A(4, 4, 0, 2, 2);
  A.mapInPlace(&f);
}
```

IndexInPlace denotes, that the skeleton has both of the above explained properties: It must provide additional parameter(s) for the global index(es) of the current element and overwrites each element with the result of applying the given function `f` to the element. Since these skeletons overwrite elements, all of them return **`void`**. The following example creates a distributed sparse matrix `A` of size 4×4 and partitions it into submatrices of size 2×2 (cf. Section 3.4). Then, each element is replaced by applying `f` to it:

```
double f(double value, int rowIndex, int columnIndex) {
  return value * (rowIndex + columnIndex);
}

void main(int argc, char** argv) {
  DistributedSparseMatrix<double> A(4, 4, 2, 2);
  A.mapIndexInPlace(&f);
}
```

All data parallel skeletons are provided in terms of member functions of a distributed data structure. Since each skeleton conceptually answers a different purpose, we classify them into communication skeletons, computation skeletons, and combined skeletons:

- Communication skeletons perform typical parallel communication patterns such as broadcasting an element to all partitions or collection all elements from each partition. Communication skeletons do not perform any computation tasks.
- Computation skeletons perform typical parallel computation patterns such as applying a given function to each element or combining two data structures. Computation skeletons do not perform any communication tasks.
- Combinded skeletons perform both parallel computation as well as parallel communication patterns such as folding all elements.

3.1.2 Local vs. Global View

When distributing a data structure among a certain number of processes, each process only stores a part of the whole data structure. This part locally available to a single process is often referred to as *partition*. The global data structure only exists in the programmer's mind. However,

the programmer wants to access the distributed data structure in whole and is not interested in the local view of each process. To overcome this problem, it is reasonable to distinguish between a *global* and a *local* view:

Global From a programmer's perspective, this is the default view when working with our data structures. In general, the programmer is not interested in how the data structure is decomposed into partitions and which partition is assigned to which process. All these details are intended to be completely transparent to the user of our library, such that one can write parallel programs in a sequential way.

Local As already mentioned, the whole data structure is decomposed into several partitions and exclusively assigned to one process. By doing so, each partition can only access its locally stored elements. To prevent unnecessary index calculations, these elements are always accessed with their *local* index. However, each partition is able to compute the global index of each element, since each partition stores its starting index with respect to the whole data structure.

Figure 3.1 illustrates the difference between a local and a global view point by means of the 4×4 distributed sparse matrix A (cf. Figure 3.1a). The matrix is decomposed into four partitions of size 2×2 and distributed among four processes. Thus, each process p_i is exclusively assigned one partition (cf. Figure 3.1b). From a global perspective, accessing the element $a_{0,0}$ will yield 1 for each process. However, from a local perspective, accessing the element $a_{0,0}$ will yield a different result for each process, since each process returns the element in the upper left corner of its local partition. Thus, p_1 will return 1, p_2 will return 0, p_3 will return 6, and p_4 will return 3.

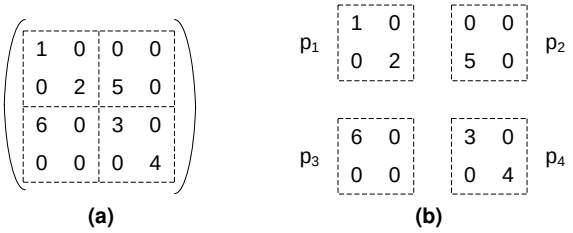


Figure 3.1: (a) Global view of the sparse matrix A. (b) Local view of the sparse matrix A after distributing the partitions among $np = 4$ processes. p_i denotes process i.

By default, all indexes used by skeletons and auxiliary functions are expected to be global ones. If this is not the case, the corresponding function has the suffix `Local` as part of its name. There are also functions which expect a global index for the first and a local index for the second argument, and vice versa. In this case, the function has the suffix `GlobalLocal` and `LocalGlobal`, respectively.

3.1.3 Multi-Core Processing

All of our distributed data structures not only work on multi-processor architectures with a distributed memory, but efficiently make use of current multi-core processors using a shared memory architecture. This feature has been implemented using OpenMP, which is an abbreviation for Open Multi-Processing [44, 45]. As the name suggests, OpenMP is an API which has been developed to support the development of parallel applications for shared memory architectures. Essentially, the API consists of a couple of compiler directives and library routines which makes it very easy to use. The main advantages of OpenMP are as follows:

Simplicity There is no need for dealing with message passing and all its common errors such as starvation or deadlocks as with MPI. Instead, OpenMP creates a team of threads and distributes the work among them. After the parallel region has been executed, all threads are destroyed.

Incremental parallelization In general, one can speed up existing programs by simply inserting OpenMP directives without needing to reorganize the code dramatically. Thus, one can parallelize a program in a step-by-step manner with little risk to introduce new bugs.

Portability Due to the fact that many of the major IT companies were involved during the specification of OpenMP, the API is supported by virtually all C++ compilers[1]. If the compiler for whatever reasons does not support OpenMP it will nevertheless compile the code and simply ignore the OpenMP directives by treating them as comments.

Section 5.5 shows how OpenMP directives can speed up the execution of certain functions using the example of the *mapInPlace* skeleton. Furthermore, Section 3.4.5 presents some experimental benchmarks which show the scalability behaviour of all OpenMP enhanced skeletons of the distributed sparse matrix.

3.2 Distributed Array

The class `DistributedArray<E>` can be used to distribute an array of length `size` among `np` processes. The following constraints apply:

- The number `np` of used processes must divide the size of the distributed array without remainder, i.e. $\texttt{size} \bmod \texttt{np} = 0$.
- The number `np` of used processes must be a power of 2. Otherwise, the following functions and all its variants cannot be used: `broadcast`, `fold`, `gather`, `permute`, `rotate`, and `show`.

3.2.1 Constructors

The following constructors can be used to create a new distributed array. None of them requires any communication, since initially, all processes have access to the whole data structure. Each process then autonomously decides which part of the data structure has to be copied and stored locally. Constructors annotated with $\star$ are enhanced by OpenMP directives (cf. Section 3.1.3).

```
DistributedArray(int size)
```

Creates a distributed array with `size` elements where $\texttt{size} > 1$.

```
DistributedArray(int size, E initial)                                    ⋆
```

Creates a distributed array with `size` elements where $\texttt{size} > 1$. Each element is initialized with the given value `initial`.

[1] In order to make use of the OpenMP parallelization, one needs to enable this compiler feature. This is usually done by providing the option *-openmp*.

`DistributedArray(int size, const E* b)` ⋆

Creates a distributed array with `size` elements where `size` > 1. Each element is initialized with the corresponding element of the given array `b`. Obviously, the length of `b` must match the given argument `size`. Let lb be the length of `b`. If $lb \geq$ `size` the last $lb -$ `size` elements of `b` are ignored. If $lb <$ `size`, the programm will exit.

`DistributedArray(int size, E (*f)(int))` ⋆

Creates a distributed array with `size` elements where `size` > 1. Each element a_i is initialized with `f`(i). Note that `f` can take the global index of the current element into account.

`template<class F> DistributedArray(int size, const Fct1<int, E, F>& f)` ⋆

Variant for partial applications (see above).

`DistributedArray(const DistributedArray<E>& B)` ⋆

Copy constructor. Creates a new distributed array by means of the given distributed array `B`.

3.2.2 Skeletons

The following sections provide a complete list of all skeletons currently implemented for the distributed array. Skeletons annotated with ⋆ are enhanced by OpenMP directives (cf. Section 3.1.3).

3.2.2.1 Computation Skeletons

`inline template<class R> DistributedArray<R> map(R (*f)(E)) const` ⋆

Creates a new distributed array b with the same size as the original distributed array a and initializes each element b_i with `f`(a_i). Note that b may store a different type `R` than a does.

```
inline template<class R, class F> DistributedArray<R> map(                ⋆
    const Fct1<E, R, F>& f) const
```

Variant for partial applications (see above).

```
inline template<class R>                                                   ⋆
    DistributedArray<R> mapIndex(R (*f)(int, E)) const
```

Creates a new distributed array b with the same size as the original distributed array a and initializes each element b_i with `f`(i, a_i). Note that b may store a different type `R` than a does and that `f` may take the global index of the current element into account.

```
inline template<class R, class F> DistributedArray<R> mapIndex(                ⋆
    const Fct2<int, E, R, F>& f) const
```

Variant for partial applications (see above).

```
inline void mapIndexInPlace(E (*f)(int, E))                                    ⋆
```

Sets each element a_i of the distributed array to `f`(i, a_i).

```
inline template<class F>                                                       ⋆
    void mapIndexInPlace(const Fct2<int, E, E, F>& f)
```

Variant for partial applications (see above).

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

Sets each element a_i of the distributed array to `f`(a_i).

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

Variant for partial applications (see above).

```
inline void mapPartitionInPlace(void (*f)(E*))
```

Replaces the whole partition, i.e. all locally stored elements, by applying the given function `f` to them.

```
inline template<class F>
    void mapPartitionInPlace(const Fct1<E*, void, F>& f)
```

Variant for partial applications (see above).

```
inline template<class E2, class R> DistributedArray<R>                         ⋆
    zipWith(const DistributedArray<E2>& b, R (*f)(E, E2)) const
```

Creates a new distributed array c with the same size as the original distributed array a where each element is set to `f`(a_i, b_i). Thus, the given distributed array `b` must also be of length `size`. If `b` is smaller, the program will exit. If `b` is larger, surplus elements will be ignored.

```
inline template<class E2, class R, class F> DistributedArray<R>                ⋆
    zipWith(const DistributedArray<E2>& b, const Fct2<E, E2, R, F> f) const
```

Variant for partial applications (see above).

```
inline template<class E2, class R> void zipWithIndex(                    ⋆
    const DistributedArray<E2>& b, R (*f)(int, E, E2)) const
```

Creates a new distributed array c with the same size as the original distributed array a where each element c_i is set to $\texttt{f}(i, a_i, b_i)$. Thus, the given distributed array `b` must also be of length `size`. If `b` is smaller, the program will exit. If `b` is larger, surplus elements will be ignored.

```
inline template<class E2, class R, class F> DistributedArray<R>          ⋆
    zipWithIndex(const DistributedArray<E2>& b,
    const Fct3<int, E, E2, R, F>& f) const
```

Variant for partial applications (see above).

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

Sets each element a_i of the distributed array to $\texttt{f}(i, a_i, b_i)$. Thus, the given distributed array `b` must also be of length `size`. If `b` is smaller, the program will exit. If `b` is larger, surplus elements will be ignored.

```
inline template<class E2, class F> void zipWithIndexInPlace(             ⋆
    const DistributedArray<E2>& b, const Fct3<int, E, E2, E, F>& f) const
```

Variant for partial applications (see above).

```
inline template<class E2> void zipWithInPlace(                           ⋆
    const DistributedArray<E2>& b, E (*f)(E, E2))
```

Sets each element a_i of the distributed array to $\texttt{f}(a_i, b_i)$. Thus, the given distributed array `b` must also be of length `size`. If `b` is smaller, the program will exit. If `b` is larger, surplus elements will be ignored.

```
inline template<class E2, class F> void zipWithInPlace(                  ⋆
    const DistributedArray<E2>& b, const Fct2<E, E2, E, F> f)
```

Variant for partial applications (see above).

3.2.2.2 Communication Skeletons

```
void allToAll(const DistributedArray<int*>& indexes, E dummy)
```

Each collaborating processor sends a block of elements to every other processor. The beginnings of all blocks are specified by the given distributed array `indexes`. Thus, `indexes` must be of length `size/np`. If `indexes` is smaller, the program will exit. If `indexes` is larger, surplus elements are simply ignored. The received blocks are concatenated without gaps in arbitrary order. If a processor receives less elements than its local partition can accomodate, the remaining elements are filled with the given argument `dummy`. The programm will exit, if a processor receives more elements than it can accomodate.

void broadcast(**int** index)

Sends the element at the given global index to all processors where $0 \leq$ index $<$ size. Afterwards, each element of the distributed array stores the same value. If index < 0 or index $\geq$ size, an IllegalPartitionException will be thrown.

void broadcastPartition(**int** blockIndex)

Broadcasts the block with the given index to all processors. Afterwards, each block of the distributed array stores the same values. Note that $0 \leq$ blockIndex $<$ size/np must hold. Otherwise, an IllegalPartitionException will be thrown.

void gather(E* b) **const**

Transforms a distributed array to an ordinary array by copying each element to the given array b. Obviously, b must at least be of length size. If b is smaller, the program will exit. If b is larger, surplus elements are left unchanged.

inline void permute(**int** (*f)(**int**))

Permutes the elements of the distributed array according to the given function f. For this reason, f must be bijective and return the new global index for each element a_i with $0 \leq i < size$.

inline template<**class** F> **void** permute(**const** Fct1<**int, int,** F>& f)

Variant for partial applications (see above).

inline void permutePartition(**int** (*f)(**int**))

Permutes whole partitions of the distributed array according to the given function f. For this reason, f must be bijective and return the ID of the new process p_i to store the partition with $0 \leq i < np$.

inline template<**class** F> **void** permutePartition(**const** Fct1<**int, int,** F>& f)

Variant for partial applications (see above).

3.2.2.3 Combined Skeletons

None of the following *map* skeleton variants increases the expressiveness of our skeleton library, but are merely provide as an optimization in case one uses the *map* skeleton and directly afterwards another skeleton, e.g. *fold* or *scan* [46].

```
inline void fold(E (*f)(E, E)) const
```
⋆

Reduces all elements of the distributed array to a single one by successively applying the given binary function `f` to them. `f` must be associative and commutative.

```
inline template<class F> E fold(const Fct2<E, E, E, F>& f) const
```
⋆

Variant for partial applications (see above).

```
inline E mapIndexInPlaceFold(E (*g)(int, E), E (*f)(E, E)) const
```
⋆

Reduces all elements of the distributed array to a single one by successively applying the given binary function `f` to them. Prior to folding each new element a_i, the given function `g` is applied to it. The effects of applying `g` to each element are *not* persistent, i.e. the elements of a will not be altered. `f` must be associative and commutative in order to return a correct result.

```
inline template<class F1, class F2> E mapIndexInPlaceFold(
    const Fct2<int, E, E, F1>& g, const Fct2<E, E, E, F2>& f) const
```
⋆

Variant for partial applications (see above).

```
inline template<class F1, class F2> E mapIndexInPlaceFold(
    E (*g)(int, E), const Fct2<E, E, E, F2>& f) const
```
⋆

Variant for mixed arguments where `g` is an ordinary function pointer and `f` a partial application (see above).

```
inline template<class F1, class F2> E mapIndexInPlaceFold(
    const Fct2<int, E, E, F1>& g, E (*f)(E, E)
```
⋆

Variant for mixed arguments where `g` is a partial application and `f` an ordinary function pointer (see above).

```
inline void mapIndexInPlacePermutePartition(E (*f)(int, E), int (*g)(int))
```

Sets each element a_i of the distributed array to `f`(i, a_i) and permutes the locally stored elements by help of the given function `g`. For each of the `np` processes, `g` is used to calculate the ID of the new process which will store the elements. For this reason, `g` must be bijective. Otherwise, an `IllegalPermuteException` is thrown.

```
inline template<class F> void mapIndexInPlacePermutePartition(
    const Fct2<int, E, E, F1>& f, const Fct1<int, int, F2>& g) const
```

Variant for partial applications (see above).

```
inline void mapIndexInPlaceScan(E (*g)(int, E), E (*f)(E, E)) const
```

Sets each element a_i of the distributed array to $\texttt{f}(i, a_i)$ and subsequently replaces each element a_i by the result of reducing the elements $a_0 \ldots a_i$ by successively applying the given binary function `g` to them.

```
inline template<class F> void mapIndexInPlaceScan(
    const Fct2<int, E, E, F>& g, const Fct2<E, E, E, F>& f) const
```

Variant for partial applications (see above).

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

Replaces each element a_i of the distributed array by the result of folding $a_0 \ldots a_i$ using the given function `f`. `f` must be associative and commutative.

```
inline template<class F> void scan(const Fct2<E, E, E, F>& f) const
```

Variant for partial applications (see above).

3.2.3 Auxiliary Functions

The following auxiliary functions can be used to access and modify properties of the local partition of a distributed array. They are no skeletons, but they are typically used in user-defined skeleton argument functions. Functions annotated with $\star$ are enhanced by OpenMP directives (cf. Section 3.1.3).

```
DistributedArray<E> copy() const                                    ⋆
```

Copies the distributed array by using a copy constructor and returns a reference to the new distributed array.

```
DistributedArray<E> copyWithGap(int size, E dummy) const            ⋆
```

Copies the distributed array by copying each partition to a partition of the new distributed array. The new distributed array is of length `size` where $\texttt{size} > 1$. If the new distributed array is smaller than the original distributed array, surplus elements will get lost. If it is larger, missing elements are filled with the given argument `dummy`.

```
E get(int index) const
```

Returns the value of the element at the given global index where $0 \leq \texttt{index} < \texttt{size}$.

int getFirst() **const**

Returns the global index of the first element of the local partition.

E getLocal(**int** index) **const**

Returns the value of the element at the given local index of the local partition. Note that $0 \leq$ index $<$ getLocalSize() must hold. Otherwise, the program will exit.

int getLocalSize() **const**

Returns the number of elements of the local partition of the distributed array.

int getSize() **const**

Returns the total number of elements of the distributed array.

bool isLocal(**int** index) **const**

True, if the given global index corresponds to an element in the local partition of the distributed array. False otherwise.

void set(**int** index, E value)

Sets the element at the given global index of the distributed array to the given value where $0 \leq$ index $<$ size.

void setLocal(**int** index, E value)

Sets the element at the given local index of the local partition to the given value where $0 \leq$ index $<$ getLocalSize().

void show() **const**

Prints the values of the distributed array to standard output. Alternatively, one can use the operator <<.

3.3 Distributed Matrix

The class DistributedMatrix<E> can be used to distribute a matrix of size n $\times$ m among np processes. For this purpose the matrix is decomposed into several blocks. The size of each block is derived from the given parameters n, m, rows and cols such that each block is of size (n / rows) $\times$ (m / cols). The following constraints apply:

- rows · cols = np, i.e. the number of blocks must be equal to the number of processors used. Otherwise, the program will exit when accessing certain elements of the distributed matrix.
- The number of processors used must be a power of 2. Otherwise, the following functions and all its variants cannot be used: allToAll, broadcast, fold, gather, permute, rotate, scan, and show.
- Both n mod rows = 0 and m mod cols = 0 must hold.

Internally, blocks are identified using both a unique ID or the coordinates within the group of collaborating processes. The ID is assigned row-wise from left to right and starts with 0. A similar scheme is used by the distributed sparse matrix (cf. Section 3.4.1.1).

3.3.1 Constructors

The following constructors can be used to create a new distributed matrix. None of them requires any communication, since initially, all processes have access to the whole data structure. Each process then autonomously decides which part of the data structure has to be copied and stored locally. Constructors annotated with ⋆ are enhanced by OpenMP directives (cf. Section 3.1.3).

```
DistributedMatrix(int n, int m, int rows, int cols)
```

Creates a new distributed matrix of size n × m, all elements are left uninitialized. The parameters rows and cols determine into how many rows and columns the matrix is split up. Note that n ≥ 1, m ≥ 1, 1 ≤ rows ≤ n and 1 ≤ cols ≤ m must hold.

```
DistributedMatrix(int n, int m, E initial, int rows, int cols)                ⋆
```

Creates a new distributed matrix of size n × m and initializes each element with the given parameter initial. The parameters rows and cols determine into how many rows and columns the matrix is split up. Note that n ≥ 1, m ≥ 1, 1 ≤ rows ≤ n and 1 ≤ cols ≤ m must hold.

```
DistributedMatrix(int n, int m, const E** a, int rows, int cols)              ⋆
```

Creates a new distributed matrix of size n × m, each element is initialized with the corresponding element of the given matrix a. Obviously, a must at least be of size n × m. If a is smaller, the program will exit. If a is larger, surplus elements are simply ignored. The parameters rows and cols determine into how many rows and columns the matrix is split up. Note that n ≥ 1, m ≥ 1, 1 ≤ rows ≤ n and 1 ≤ cols ≤ m must hold.

```
DistributedMatrix(int n, int m, E (*f)(int, int), int rows, int cols)        ⋆
```

Creates a new distributed matrix of size n × m, each element $a_{i,j}$ is initialized by evaluating the given function f(i, j). The parameters rows and cols determine into how many rows and columns the matrix is split up. Note that n ≥ 1, m ≥ 1, 1 ≤ rows ≤ n and 1 ≤ cols ≤ m must hold.

```
template<class F> DistributedMatrix(int n, int m,                    ⋆
    Fct2<int, int, E, F>& f, int rows, int cols)
```

Variant for partial applications (see above).

```
DistributedMatrix(const DistributedMatrix<E>& B)                     ⋆
```

Copy constructor. Creates a new distributed matrix by means of the given distributed matrix `B`.

3.3.2 Skeletons

The following sections provide a complete list of all skeletons currently implemented for the distributed matrix. Skeletons annotated with ⋆ are enhanced by OpenMP directives (cf. Section 3.1.3).

3.3.2.1 Computation Skeletons

```
template<class R> inline DistributedMatrix<R> map(R (*f)(E)) const   ⋆
```

Creates a new distributed matrix b with the same size as the original distributed matrix a and initializes each element $b_{i,j}$ with $\mathtt{f}(a_{i,j})$. Note that b may store a different type `R` than a does.

```
template<class R, class F> inline DistributedMatrix<R> map(          ⋆
    const Fct1<E, R, F>& f) const
```

Variant for partial applications (see above).

```
template<class R> inline DistributedMatrix<R> mapIndex(              ⋆
    R (*f)(int, int, E)) const
```

Creates a new distributed matrix b with the same size as the original distributed matrix a and initializes each element $b_{i,j}$ with $\mathtt{f}(i, j, a_{i,j})$. Note that b may store a different type `R` than a does and that `f` may take the global index of the current element into account.

```
template<class R, class F> inline DistributedMatrix<R> mapIndex(     ⋆
    const Fct3<int, int, E, R, F>& f) const
```

Variant for partial applications (see above).

```
inline void mapIndexInPlace(E (*f)(int, int, E))                     ⋆
```

Sets each element $a_{i,j}$ of the distributed matrix to $\mathtt{f}(i, j, a_{i,j})$. Note that `f` may take the global index of the current element into account.

```
template<class F> inline void                                          ⋆
    mapIndexInPlace(const Fct3<int, int, E, E, F>& f)
```

Variant for partial applications (see above).

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

Sets each element $a_{i,j}$ of the distributed matrix to $\mathtt{f}(a_{i,j})$.

```
template<class F> inline void mapInPlace<const Fct1<E, E, F>& f        ⋆
```

Variant for partial applications (see above).

```
inline void mapPartitionInPlace(void (*f)(E**))
```

Replaces all locally stored elements by applying the given function `f` to them.

```
template<class F> inline void
    mapPartitionInPlace(const Fct1<E**, void, F>& f)
```

Variant for partial applications (see above).

```
template<class E2, class R> inline DistributedMatrix<R>                ⋆
    zipWith(const DistributedMatrix<E2>& b, R (*f)(E, E2)) const
```

Creates a new distributed matrix c with the same size as the original distributed matrix a where each element is set to $\mathtt{f}(a_{i,j}, b_{i,j})$. Thus, the given distributed array `b` must be of the same size and distribution as `a`.

```
template<class E2, class R, class F> inline DistributedMatrix<R>       ⋆
    zipWith(const DistributedMatrix<E2>& b, const Fct2<E, E2, R, F>& f) const
```

Variant for partial applications (see above).

```
template<class E2, class R> inline DistributedMatrix<R> zipWithIndex(  ⋆
    const DistributedMatrix<E2>& b, R (*f)(int, int, E, E2)) const
```

Creates a new distributed matrix c with the same size as the original distributed matrix a where each element is set to $\mathtt{f}(i, j, a_{i,j}, b_{i,j})$. Thus, the given distributed array `b` must be of the same size and distribution as `a`.

```
template<class E2, class R, class F> inline DistributedMatrix<R>       ⋆
    zipWithIndex(const DistributedMatrix<E2>& b,
    const Fct4<int, int, E, E2, R, F>& f) const
```

Variant for partial applications (see above).

```
template<class E2> inline void zipWithIndexInPlace(                    ⋆
    const DistributedMatrix<E2>& b, E (*f)(int, int, E, E2))
```

Sets each element $a_{i,j}$ of the distributed matrix fo $\texttt{f}(i, j, a_{i,j}, b_{i,j})$. Thus, the given distributed array `b` must be of the same size and distribution as `a`.

```
template<class E2, class F> inline void zipWithIndexInPlace(           ⋆
    const DistributedMatrix<E2>& b, const Fct4<int, int, E, E2, E, F>& f)
```

Variant for partial applications (see above).

```
template<class E2> inline void zipWithInPlace(                         ⋆
    const DistributedMatrix<E2>& b, E (*f)(E, E2))
```

Sets each element $a_{i,j}$ of the distributed matrix fo $\texttt{f}(a_{i,j}, b_{i,j})$. Thus, the given distributed array `b` must be of the same size and distribution as `a`.

```
template<class E2, class R, class F> inline DistributedMatrix<R>       ⋆
    zipWithInPlace(const DistributedMatrix<E2>& b,
    const Fct4<int, int, E, E2, R, F>& f) const
```

Variant for partial applications (see above).

3.3.2.2 Communication Skeletons

```
void broadcast(int rowIndex, int colIndex)
```

Replaces each element of the distributed matrix with the element with the given global indexes. Therefore, $0 \leq$ `rowIndex` $<$ `n` and $0 \leq$ `colIndex` $<$ `m` must hold. Otherwise, an `IllegalPartitionException` is thrown.

```
void broadcastPartition(int blockIndexRow, int blockIndexCol)
```

Replaces each block of the distributed matrix with the block with the given indexes. Note that $0 \leq$ `blockIndexRow` $<$ `n`$/$`rows` and $0 \leq$ `blockIndexCol` $<$ `m`$/$`cols` must hold. Otherwise, an `IllegalPartitionException` is thrown.

```
void gather(E** b) const
```

Transforms a distributed matrix to an ordinary matrix by copying each element $a_{i,j}$ to the given matrix `b`. Obviously, `b` must at least be of size `n` $\times$ `m`. If `b` is smaller, the program will exit. If b is larger, surplus elements are left unchanged.

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

Permutes the blocks of a distributed matrix according to the given functions `f` and `g`. Both functions will be passed the indexes of the locally available block and must return the new block index. While `f` must return the new row block index, `g` must do the same for the new column block index. Note that `f` must return a value between 0 and `getBlocksInRow()` and that `g` must return a value between 0 and `getBlocksInCol()`. In other words, `f` and `g` in combination must create a bijective relation. Otherwise, an `IllegalPermuteException` is thrown.

```
template<class F> inline void permutePartition(
    const Fct2<int, int, int, F>& f, int (*g)(int, int))
```

Variant with `f` being a partial application (see above).

```
template<class F> inline void permutePartition(
    int (*f)(int, int), const Fct2<int, int, int, F>& g)
```

Variant with `g` being a partial application (see above).

```
template<class F1, class F2> inline void permutePartition(
    const Fct2<int, int, int, F1>& newRow, const Fct2<int, int, int, F2>& newCol)
```

Variant for both `f` and `g` being partial applications (see above).

```
inline void rotateCols(int (*f)(int colIndex)
```

Rotates the partitions of a distributed matrix cyclically in vertical direction. The number of steps depends on the given function `f` which calculates this number for each column. Negative numbers correspond to cyclic rotations upwards, whereas positive numbers correspond to cyclic rotations downward.

```
template<class F> inline void rotateCols(const Fct1<int, int, F>& f)
```

Variant for partial applications (see above).

```
inline void rotateCols(int steps)
```

Rotates the partitions of a distributed matrix cyclically in vertical direction. The number of steps depends on the given parameter `steps`. Negative numbers correspond to cyclic rotations upwards, whereas positive numbers correspond to cyclic rotations downward.

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

Rotates the partitions of a distributed matrix cyclically in horizontal direction. The number of steps depends on the given function `f` which calculates this number for each row. Negative numbers correspond to cyclic rotations to the left, whereas positive numbers correspond to cyclic rotations to the right.

```
template<class F> inline void rotateRows(const Fct1<int, int, F>& f)
```

Variant for partial applications (see above).

```
inline void rotateRows(int steps)
```

Rotates the partitions of a distributed matrix cyclically in horizontal direction. The number of steps depends on the given parameter `steps`. Negative numbers correspond to cyclic rotations to the left, whereas positive numbers correspond to cyclic rotations to the right.

3.3.2.3 Combined Skeletons

```
inline E fold(E (*f)(E, E)) const                                    ⋆
```

Reduces all elements of the distributed matrix to a single one by successively applying the given binary function `f` to them. `f` must be associative and commutative.

```
template<class F> E fold(const Fct2<E, E, E, F>& f) const            ⋆
```

Variant for partial applications (see above).

3.3.3 Auxiliary Functions

The following auxiliary functions can be used to access and modify properties of the local partition of a distributed array. They are no skeletons, but they are typically used in user-defined skeleton argument functions. Functions annotated with ⋆ are enhanced by OpenMP directives (cf. Section 3.1.3).

```
inline E get(int rowIndex, int colIndex)
```

Returns the element with the given global indexes.

```
inline int getBlocksInCol() const
```

Returns the number of blocks per column of the distributed matrix, i.e. `cols`.

```
inline int getBlocksInRow() const
```

Returns the number of blocks per row of the distributed matrix, i.e. `rows`.

inline int getCols() **const**

Returns the number of columns of the distributed matrix, i.e. m.

inline int getFirstCol() **const**

Returns the global column index of the first column of the locally stored block.

inline int getFirstRow() **const**

Returns the global row index of the first row of the locally stored block.

inline E getGlobalLocal(**int** rowIndex, **int** colIndex)

Returns the element with the given indexes. Note that rowIndex is interpreted as a global index whereas colIndex is interpreted as a local index. Thus, $0 \leq$ rowIndex $<$ n and $0 \leq$ colIndex $<$ getLocalCols() must hold. A NonLocalAccessException is thrown otherwise.

inline E getLocal(**int** rowIndex, **int** colIndex)

Returns the element with the given indexes. rowIndex and columnIndex are interpreted as local indexes. Thus, $0 \leq$ rowIndex $<$ getLocalRows() and $0 \leq$ colIndex $<$ getLocalCols() must hold. A NonLocalAccessException is thrown otherwise.

inline int getLocalCols() **const**

Returns the number of columns of the locally stored block, i.e. n / rows.

inline E getLocalGlobal(**int** rowIndex, **int** colIndex)

Returns the element with the given index. rowIndex is a local index and that colIndex is a global index. Thus, $0 \leq$ rowIndex $<$ getLocalRows() and $0 \leq$ colIndex $<$ m must hold. Otherwise a NonLocalAccessException is thrown.

inline int getLocalRows() **const**

Returns the number of rows of the locally stored block.

inline int getRows() **const**

Returns the number of rows of the distributed matrix, i.e. n.

```
inline bool isLocal(int rowIndex, int colIndex) const
```

True, if the element with the given global indexes is stored locally. False otherwise.

```
inline int setLocal(int rowIndex, int colIndex, E value)
```

Sets the element with the given indexes to the given `value`. Note that `rowIndex` and `colIndex` are interpreted as local indexes. Thus, $0 \leq$ `rowIndex` $<$ `getLocalRows()` and $0 \leq$ `colIndex` $<$ `getLocalCols()` must hold.

```
inline void show() const
```

Prints the values of the distributed matrix to standard output. Each line is surrounded by parantheses, elements are separated by blanks. Alternatively, one can use the operator `<<`.

3.4 Distributed Sparse Matrix

Sparse matrices play an important role in numerical analysis: The discretization of partial differential equations with the finite element method or the description of graphs by means of an adjacency matrix often results in a matrix primarily populated with zeros. Especially for very large matrices it is beneficial to use special data structures that take advantage of its sparse structure. Thus, operations can be performed faster while simultaneously consuming less memory compared to standard data structures. Our data structure for general sparse matrixes provides the following core features:

- Support for various data parallel skeletons such as *map*, *fold*, and *zip* in terms of member functions in order to manipulate the data structure in parallel.
- Support for arbitrary compression schemes. The user can define how the sparse matrix is compressed. In fact, the user has the option to extend our predefined compression schemes and use her own.
- Support for arbitrary distribution schemes. The user can define how the sparse matrix is distributed across the processors. This load balancing mechanism is highly flexible, since again our predefined schemes can be extended.
- Besides supporting multi-processor architectures with a distributed memory, our data structure also makes use of multi-core processors with a shared memory architecture such that some skeletons and auxiliary functions can be executed even faster.

At first glance, our data structure for distributed sparse matrices competes with approved and established software libraries for numerical computations such as LINPACK [47] and its successor LAPACK [48]. These libraries offer many powerful routines for solving arbitrary systems of linear equations, least-square and eigenvalue problems. While the former has been designed to run on supercomputers in the 1970s, the latter has been developed to run on modern shared memory computer architectures. More recently, the ScaLAPACK [49] project has redesigned numerous LAPACK routines to facilitate numerical computations even on distributed memory systems. Obviously, our data structure cannot compete with the aforementioned libraries and does, in fact, not intend to do so. Our library is not designed to solve linear systems of equations but focuses

on the support of data parallel algorithmic skeletons for sparse matrices. Although LAPACK as well as ScaLAPACK can handle band matrices, none of them supports general sparse matrices or user-defined functions, i.e. skeletons, at all. The SPARSKIT [50] tool package, which has been explicitly designed to work with general sparse matrices, can indeed convert between a vast number of different storage schemes but also lacks the support for skeletons.

3.4.1 Concepts

When designing our distributed data structure for general sparse matrices we aimed at providing a highly flexible construct which is straightforward to use and easy to extend. The following subsections describe the key concepts used to implement this flexibility. Some, but definitely not all implementation details are covered in Chapter 5.

3.4.1.1 Submatrices

One of our main objectives was to implement a data structure for general sparse matrices which supports arbitrary compression schemes, not merely a single one. This is important in conjunction with the fact that all compression schemes must perform a tradeoff between the access time for a single element and the space needed for storing the compressed matrix. For this reason, the global sparse matrix of size $n \times m$ is split up into multiple submatrices of size $r \times c$ with $n, m, r, c \in \mathbb{N}$. Usually, $r \ll n$ and $c \ll m$. Both parameters must be defined by the user. This concept is similar to the one used in the Block Sparse Row compression scheme [50]. However, our implementation uses this approach only to divide the global matrix into multiple submatrices, not to compress them. The actual compression is handled by the submatrices, i.e. each submatrix is responsible for compressing locally available elements. How this compression takes place can be defined by the user (cf. Section 3.4.1.2).

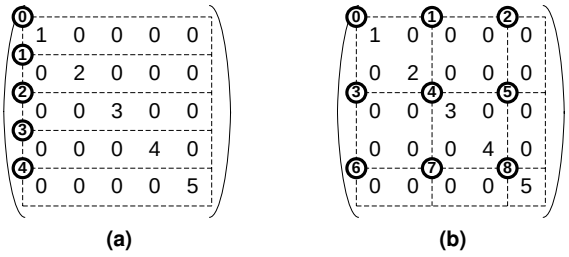


Figure 3.2: Splitting up a 5×5 sparse matrix into multiple submatrices using (a) $r = 1$ and $c = 5$ and (b) $r = c = 2$. The encircled number in the upper left corner of a submatrix shows its ID.

Our approach offers a lot of flexibility regarding the partitioning of the sparse matrix. For example, if $r = 1$ and $c = m$, the matrix is split into n submatrices such that each submatrix stores a single row (cf. Figure 3.2a). A column-wise partitioning would be achieved by setting $r = n$ and $c = 1$. This flexibility comes in handy when the user often needs to rotate rows or columns, as all required elements are stored locally in a single submatrix such that no communication overhead is involved. Furthermore, note that neither r nor c need to divide n and m without remainder. If $n \bmod r \neq 0$ or $m \bmod c \neq 0$ the size of the submatrices at the edge of the matrix is automatically adjusted accordingly (cf. Figure 3.2b). Due to internal reasons, each submatrix is assigned a unique ID which is depicted by the encircled number in the upper left corner. The usage of this ID will be explained in more detail in Sections 5.3 and 5.4.

3.4.1.2 Compression Scheme

As already mentioned, one of our main objectives was to implement a data structure which supports arbitrary compression schemes. This is important since all compression schemes perform a tradeoff between access time and storage space. For example, the CRS scheme has asymptotic access costs of $O(\log n)$, since the elements are stored row-wise in a sorted list, and needs $O(nnz)$ storage space with nnz being the number of non-zero elements [50]. In contrast, BSR has asymptotic assess costs of $O(1)$, since blocks are always stored entirely if containing more than one non-zero element, and needs $O(nnzb \cdot r \cdot c)$ storage space with $nnzb$ being the number of non-zero blocks. As the choice of the appropriate compression scheme depends on the user's application it is reasonable to leave this decision to her.

Currently, both compression schemes mentioned above are implemented. Note that our design can easily be extended to provide arbitrary compression schemes, e.g. Jagged Diagonal (JDD), Symmetric Skyline (SSK), Linpack Banded (BND) [50] etc. This is especially useful, if the user wants to control the tradeoff between access time and storage space more precisely than this is done by our default implementations. Again, this approach offers a high degree of flexibility, since the user can choose between different compression schemes and even implement and use her own (cf. Section 5.3).

3.4.1.3 Distribution Scheme

After partitioning the sparse matrix into multiple submatrices these submatrices need to be distributed among all collaborating processors. Rather than using a fixed distribution mechanism we decided to implement a more flexible approach. In analogy to the compression scheme choosing an appropriate distribution scheme is left to the user. This can be very useful since the user knows best which parts of the matrix are not sparse such that this feature can be used as a flexible and convenient load balancing mechanism.

Currently, four distribution schemes are implemented, namely the *Block Distribution*, where processes are assigned contiguous blocks of submatrices, the *Round Robin Distribution*, where submatrices are assigned to processes in a cyclic way, the *ColumnDistribution*, where submatrices are assigned to processes column-wise, and the *RowDistribution*, where submatrices are assigned to processes row-wise. Note that arbitrary distribution schemes can easily be extended such that the implemented load balancing mechanism is more suitable for the user's application (cf. Section 5.4).

3.4.2 Constructors

The following constructors can be used to create a new distributed sparse matrix. None of them requires any communication, since initially, all processes have access to the whole data structure. Each process then autonomously decides which part of the data structure has to be copied and stored locally. Constructors annotated with $\star$ are enhanced by OpenMP directives (cf. Sections 3.1.3 and 5.5).

```
DistributedSparseMatrix(int n, int m, int r, int c)
```

Creates an empty distributed sparse matrix of size `n` $\times$ `m`. The parameters `r` and `c` determine the size of the submatrices in which the whole sparse matrix is divided into. Note that submatrices which are located at the right and/or lower edge of the sparse matrix are smaller

if n mod r $\neq 0$ and/or m mod c $\neq 0$. This is automatically taken care of inside the data structure. Furthermore, note that n ≥ 1, m ≥ 1, $1 \leq$ r $\leq$ n and $1 \leq$ c $\leq$ m must hold.

```
DistributedSparseMatrix(int n, int m, int r, int c, E** a)
```

Creates an empty distributed sparse matrix of size n $\times$ m and initializes its values with the corresponding values of the given matrix a. Obviously, a must be at least of size n $\times$ m. If a is smaller, the program will exit. If a is greater, all surplus elements will be ignored. The parameters r and c determine the size of the submatrices in which the whole sparse matrix is divided into. Note that submatrices which are located at the right and/or lower edge of the sparse matrix are smaller if n mod r $\neq 0$ and/or m mod c $\neq 0$. This is automatically taken care of inside the data structure. Furthermore, note that n ≥ 1, m ≥ 1, $1 \leq$ r $\leq$ n and $1 \leq$ c $\leq$ m must hold.

```
DistributedSparseMatrix(const DistributedSparseMatrix<E, S, D>& B)          *
```

Copy constructor. Creates a new distributed sparse matrix by means of the given distributed sparse matrix B.

3.4.3 Skeletons

The following sections provide a complete list of all skeletons currently implemented for the distributed sparse matrix. Skeletons annotated with * are enhanced by OpenMP directives (cf. Section 3.1.3).

3.4.3.1 Computation Skeletons

```
DistributedSparseMatrix<E, S, D> map(E (*f)(E a)) const
```

Creates a new distributed sparse matrix B by means of the copy constructor and initializes each non-zero element $b_{i,j}$ by applying the given function f to the corresponding element $a_{i,j}$, i.e. replaces $b_{i,j}$ by f$(a_{i,j})$ if $a_{i,j} \neq 0$. The return value is a pointer to the new distributed sparse matrix B. Note that the distributed sparse matrix A is left unchanged.

```
template<class E2> DistributedSparseMatrix<E2, S, D>* map(
    Fct1<E, E2, F> f) const
```

Version for partial applications (see above).

```
DistributedSparseMatrix<E, S, D>* mapIndex(
    E (*f)(E a, int rowIndex, int columnIndex)) const
```

Creates a new distributed sparse matrix B by means of the copy constructor and initializes each non-zero element $b_{i,j}$ by applying the given function f to the corresponding element $a_{i,j}$, i.e. replaces $b_{i,j}$ by f$(a_{i,j}, i, j)$ if $a_{i,j} \neq 0$. The return value is a pointer to the new distributed sparse matrix B. Note that the distributed sparse matrix A is left unchanged and that f may take the global index of the current element into account.

```
template<class F, class E2> DistributedSparseMatrix<E2, S, D>* mapIndex(
    Fct3<E, int, int, E2, F> f) const
```

Version for partial applications (see above).

```
void mapIndexInPlace(E (*f)(E a, int rowIndex, int columnIndex))
```
⋆

Successively applies the given function `f` to all non-zero elements, i.e. replaces each element $a_{i,j}$ by `f`$(a_{i,j}, i, j)$ if $a_{i,j} \neq 0$.

```
template<class F> void mapIndexInPlace(Fct3<E, int, int, E, F> f)
```
⋆

Version for partial applications (see above).

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

Successively applies the given function `f` to all non-zero elements, i.e. replaces each element $a_{i,j}$ by `f`$(a_{i,j})$ if $a_{i,j} \neq 0$.

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

Version for partial applications (see above).

```
void multiply(const E* const b, E* const x) const
```
⋆

Multiplies the distributed sparse matrix A with the given vector `b` and stores the result in the given vector `x`, i.e. computes $A \cdot$ `b` $=$ `x`. Obviously, `b` must at least be of length `m` and `x` at least be of length `n`. If either or both of them are larger, surplus elements are left unchanged. If either or both of them are smaller, the program will exit.

```
template<class E2, class S2> DistributedSparseMatrix<E, S, D>* zip(
    const DistributedSparseMatrix<E2, S2, D>& B, E (*f)(E a, E2 b))
```

Creates a new distributed sparse matrix C by means of the copy constructor and initializes each non-zero element $c_{i,j}$ by merging the corresponding elements of the distributed sparse matrices A and B by means of the given function `f`, i.e. replaces each element $c_{i,j}$ by `f`$(a_{i,j}, b_{i,j})$ if $a_{i,j} \neq 0$ or $b_{i,j} \neq 0$. Note that `B` may store elements of type `E2` and that its submatrices may be of type `S2`. For this function to complete successfully, both matrices must have the same size and be distributed identically across the processes, i.e. $n_A = n_B$, $m_A = m_B$, $r_A = r_B$ and $c_A = c_B$.

```
template<class E2, class E3, class S2> DistributedSparseMatrix<E3, S, D>*
    zip(const DistributedSparseMatrix<E2, S2, D>& B,
    Fct2<E, E2, E3, F> f) const
```

Version for partial applications (see above).

```
template<class E2, class S2> DistributedSparseMatrix<E, S, D>* zipIndex(
    const DistributedSparseMatrix<E2, S2, D>& B,
    E (*f)(E a, E2 b, int rowIndex, int columnIndex))
```

Creates a new distributed sparse matrix C by means of the copy constructor and initializes each non-zero element $c_{i,j}$ by merging the corresponding elements of the distributed sparse matrices A and B by means of the given function `f`, i.e. replaces each element $c_{i,j}$ by $\texttt{f}(a_{i,j}, b_{i,j}, i, j)$ if $a_{i,j} \neq 0$ or $b_{i,j} \neq 0$. Note that `f` may take the global index of the current element into account, that `B` may store elements of type `E2` and that its submatrices may be of type `S2`. For this function to complete successfully, both matrices must have the same size and be distributed identically across the processes, i.e. $n_A = n_B$, $m_A = m_B$, $r_A = r_B$ and $c_A = c_B$.

```
template<class E2, class E3, class F, class S2> DistributedSparseMatrix
    <E3, S, D>* zipIndex(const DistributedSparseMatrix<E2 ,S2, D>& B,
    Fct4<E, E2, int, int, E3, F> f) const
```

Version for partial applications (see above).

```
template<class E2, class S2> void zipIndexInPlace(                    ⋆
    const DistributedSparseMatrix<E2, S2, D>& B,
    E (*f)(E a, E2 b, int rowIndex, int columnIndex))
```

Merges each non-zero element $a_{i,j}$ with its corresponding element $b_{i,j}$ of the given matrix `B`, i.e. replaces each element $a_{i,j}$ by $\texttt{f}(a_{i,j}, b_{i,j}, i, j)$ if $a_{i,j} \neq 0$ or $b_{i,j} \neq 0$. Note that `f` may take the global index of the current element into account, that `B` may store elements of type `E2` and that its submatrices may be of type `S2`. For this function to complete successfully, both matrices must have the same size and be distributed identically across the processes, i.e. $n_A = n_B$, $m_A = m_B$, $r_A = r_B$ and $c_A = c_B$.

```
template<class E2, class E3, class F, class S2> void zipIndexInPlace(  ⋆
    const DistributedSparseMatrix<E2, S2, D>& B,
    Fct4<E, E2, int, int, E3, F> f)
```

Version for partial applications (see above).

```
template<class E2, class S2> void zipInPlace(                         ⋆
    const DistributedSparseMatrix<E2, S, D>& B, E (*f)(E a, E2 b))
```

Merges each element $a_{i,j}$ with its corresponding element $b_{i,j}$ of the given matrix `B`, i.e. replaces each element $a_{i,j}$ by $\texttt{f}(a_{i,j}, b_{i,j}, i, j)$ if $a_{i,j} \neq 0$ or $b_{i,j} \neq 0$. Note that `B` may store elements of type `E2` and that the submatrices of `B` may be of type `S2`. For this function to complete successfully, both matrices must have the same size and be distributed identically across the processes, i.e. $n_A = n_B$, $m_A = m_B$, $r_A = r_B$ and $c_A = c_B$.

```
template<class E2, class E3, class F, class S2> void zipInPlace(      ⋆
    const DistributedSparseMatrix<E2, S2, D>& B, Fct2<E, E2, E3, F> f)
```

Version for partial applications (see above).

3.4.3.2 Communication Skeletons

`void getColumn(int index, E* const column) const` ⋆

Copies the column with the given global `index` into the given array `column`. The length of the given array must at least be `n`. If `column` is smaller, the program will exit. If `column` is larger, surplus elements are left unchanged. If the given index is out of bounds, i.e. `index` < 0 or `index` $\geq$ `n`, an `IndexOutOfBoundException` is thrown.

`void getRow(int index, E* const row) const` ⋆

Copies the row with the given global `index` into the given array `row`. The length of the given array must at least be `m`. If `row` is smaller, the program will exit. If `row` is larger, surplus elements are left unchanged. If the given index is out of bounds, i.e. `index` < 0 or `index` $\geq$ `n`, an `IndexOutOfBoundException` is thrown.

`void rotateColumn(int index, int steps)`

Rotates the column with the given index cyclically up (`steps` < 0) or down (`steps` > 0). If `steps` $= 0$, no rotation will be performed. The parameter `steps` controls the increment, i.e. how many positions each element of the given column is shifted up or down. If the given column index is out of bounds, i.e. `index` < 0 or `index` $\geq$ `m`, an `IndexOutOfBoundsException` is thrown.

`void rotateColumns(int (*f)(int columnIndex))`

Rotates all rows of the sparse matrix be means of the given function `f`. `f` is expected to return the number of steps the column with the given `columnIndex` has to be rotated. If `steps` < 0, the corresponding column will be rotated up. If `steps` > 0, the corresponding column will be rotated down. If `steps` $= 0$, no rotation will be performed. The parameter `steps` controls the increment, i.e. how many positions each element of the given column is shifted up or down. If the given `columnIndex` is out of bounds, i.e. `columnIndex` < 0 or `columnIndex` $\geq$ `m`, an `IndexOutOfBoundsException` is thrown.

`template<class F> void rotateColumns(Fct1<int, int, F> f)`

Version for partial applications (see above).

`void rotateRow(int index, int steps)`

Rotates the row with the given index cyclically to the left (`steps` < 0) or to the right (`steps` > 0). If `steps` $= 0$, no rotation will be performed. The parameter `steps` controls the increment, i.e. how many positions each element of the given row is shifted to the left or right. If the given row index is out of bounds, i.e. `rowIndex` < 0 or `index` $\geq$ `n`, an `IndexOutOfBoundsException` is thrown.

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

Rotates all rows of the sparse matrix be means of the given function `f`. `f` is expected to return the number of steps the row with the given `rowIndex` has to be rotated. If `steps` < 0, the corresponding row will be rotated to the left. If `steps` > 0, the corresponding row will be rotated to the right. If `steps` $= 0$, no rotation will be performed. The parameter `steps` controls the increment, i.e. how many positions each element of the given row is shifted to the left or right.If the given `rowIndex` is out of bounds, i.e. `rowIndex` < 0 or `rowIndex` $\geq$ `n`, an `IndexOutOfBoundsException` is thrown.

```
template<class F> void rotateRows(Fct1<int, int, F> f)
```

Version for partial appliations (see above).

3.4.3.3 Combined Skeletons

```
E fold(E (*f)(E a, E b)) const                                                    ⋆
```

Folds all non-zero elements of the distributed sparse matrix into a single value by repeatedly applying the given function `f` to them. `f` must be an associative and commutative binary function.

```
template<class F> E fold(Fct2<E,E,E,F> f) const                                   ⋆
```

Version for partial applications (see above).

```
E foldIndex(E (*f)(E a, E b, int rowIndex, int columnIndex)) const                ⋆
```

Folds all non-zero elements of the distributed sparse matrix into a single value by repeatedly applying the given function `f` to them. `f` must be an associative and commutative function and may take the global index of the given element `b` into account.

```
template<class F> E foldIndex(Fct4<E,E,int,int,F> f) const                        ⋆
```

Version for partial appliations (see above).

3.4.4 Auxiliary Functions

The following auxiliary functions can be used to access and modify properties of the local partition of a distributed sparse matrix. They are no skeletons, but they are typically used in user-defined skeleton argument functions. Functions annotated with $\star$ are enhanced by OpenMP directives (cf. Section 3.1.3).

int getC() **const**

Returns the number of columns of a submatrix, i.e. c.

int getColumnCount() **const**

Returns the number of columns of the distributed sparse matrix, i.e. m.

E getElement(**int** rowIndex, **int** columnIndex) **const**

Returns the value of the element with the given indexes. Note that $0 \leq$ rowIndex $<$ n and $0 \leq$ columnIndex $<$ m must hold. Otherwise, an IndexOutOfBoundsException is thrown.

int getElementCount() **const** ⋆

Returns the total number of non-zero elements nnz of the distributed sparse matrix.

int getElementCountInColumn(**int** index) **const** ⋆

Returns the total number of non-zero elements of the column with the given index. Note that $0 \leq$ index $<$ m must hold. Otherwise, an IndexOutOfBoundsException is thrown.

int getElementCountInRow(**int** index) **const** ⋆

Returns the total number of non-zero elements of the row with the given index. Note that $0 \leq$ index $<$ n must hold true. Otherwise, an IndexOutOfBoundsException is thrown.

E getElementLocal(**int** rowIndex, **int** columnIndex)

Returns the value of the element with the given indexes. In contrast to getElement this function does not broadcast the element to all collaborating processes but can rather be used to acces local elements of the sparse matrix. The function is for example used in the copy constructor. Note that $0 \leq$ rowIndex $<$ n and $0 \leq$ columnIndex $<$ m must hold. Otherwise, an IndexOutOfBoundsException is thrown.

double getFillRate() **const**

Returns the ratio of n·m and the total number of non-zero elements nnz.

int getR() **const**

Returns the number of rows of a submatrix, i.e. r.

int getRowCount() **const**

Returns the number of rows of the distributed sparse matrix, i.e. n.

bool isStoredLocally(**int** idSubmatrix) **const**

True, if the submatrix with the given ID is currently stored locally. False otherwise.

void print() **const**

Prints the whole distributed sparse matrix to standard output. Each line is surrounded by square brackets, elements are separated by blanks.

void print(**int** rowIndex, **int** columnIndex, **int** rows, **int** columns) **const**

Prints a section of the distributed sparse matrix to standard output. Each line is surrounded by square brackets, elements are separated by blanks. The section is defined by a starting element and the number of rows and columns to output from this position. If rowIndex and/or columnIndex are out of bounds, i.e. rowIndex < 0, rowIndex $\geq$ n, columnIndex < 0 or columnIndex $\geq$ m, or the section contains elements which are, i.e. rowIndex + rows $\geq$ n or columnIndex + columns $\geq$ m, an IndexOutOfBoundsException is thrown. Furthermore, note that the number of rows and columns to be printed are interpreted as absolute values such that the starting element is the element in the upper-left corner of the section to print.

void setElement(**int** rowIndex, **int** columnIndex, E value)

Sets the element at the given position to the given value where $0 \leq$ rowIndex $<$ n and $0 \leq$ columnIndex $<$ m.

void setPrecision(**int** value)

Sets the number of decimal places to output when printing the distributed sparse matrix to the given value.

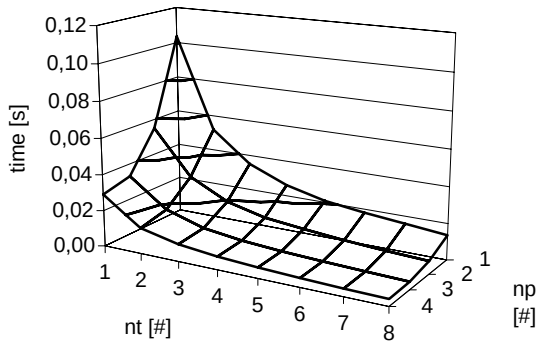
3.4.5 Results

In order to prove the efficiency of our implementation we have conducted some experimental benchmarks on a workstation cluster at the University of Münster. The cluster consists of 4 compute nodes, each equipped with two 2.66 GHz Intel Xeon Quad-Core processors (E5430) and 32 GB RAM. The nodes are connected by a 10 Gbit Ethernet and are running under CentOS 5.2 using the MPICH2 1.0.7 implementation of MPI-2 [1, 51]. The test set-up can be described as follows: We have used a sparse matrix from the Matrix Market [52] with $n = m = 90,449$ and $nnz = 1,921,995$ to measure the runtime of each skeleton function. To reduce measurement uncertainties we have averaged all results over 100 test runs. The function used with the *map* skeletons computed the sine of the given value. The *fold* skeletons used a function to sum up all elements, whereas the *zip* skeletons used a function which returned the maximum of two

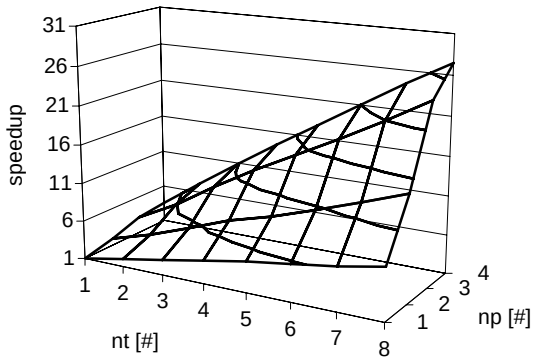
values. The parameters used for our data structure were `E` = **`double`**, `S` = `CrsSubmatrix`, `D` = `RoundRobinDistribution`, and `r` = `c` = 10. Note that the runtime of each skeleton is mainly influenced by the given user-defined function whereas the scalability behaviour is fixed by our implementation.

The results show that all skeletons exhibit reasonable scalability when increasing the number np of processors and the number nt of threads, respectively. However, increasing nt does not necessarily speed up a program. This is due to the fact that, when working with multiple threads, synchronization in terms of critical sections is sometimes mandatory, since all threads on a single processor access the same shared memory. For this reason, the skeletons *map*, *mapIndex*, *zip*, and *zipIndex* have not been enhanced with OpenMP directives, since the additional overhead is too high. In general, scalability is only ensured by increasing np while increasing nt can speed up the computation in a lot of cases, but lacks scalability in some of them.

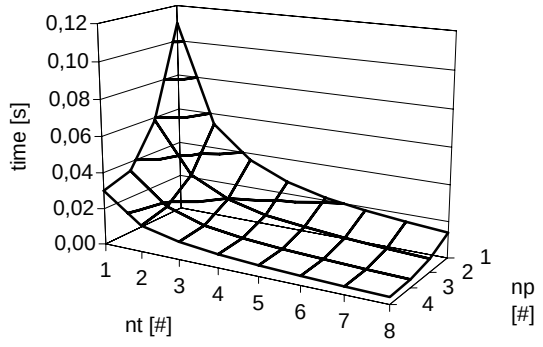
The next thing to mention is the performance of the skeletons which have the suffix `Index` as part of their name compared to the skeletons which have not. Obviously, the former perform slightly worse than the latter. This can be explained by the fact that the `Index` skeletons need to perform additional computations, since they need to calculate and pass the global indexes of the current element to the function supplied by the user. However, this additional overhead is only marginal and its ratio will decrease as the complexity of the user-defined function increases, since applying the given function to each element of the sparse matrix will take much more time than setting each element of the data structure.



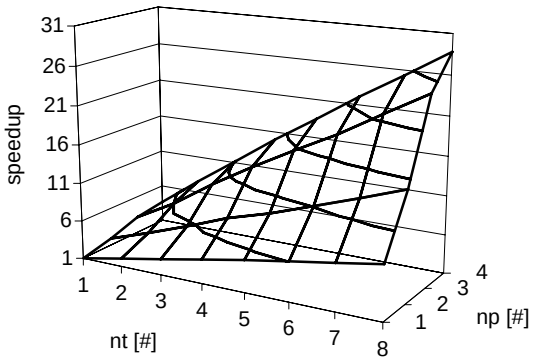
(a) Runtime of the *fold* skeleton.



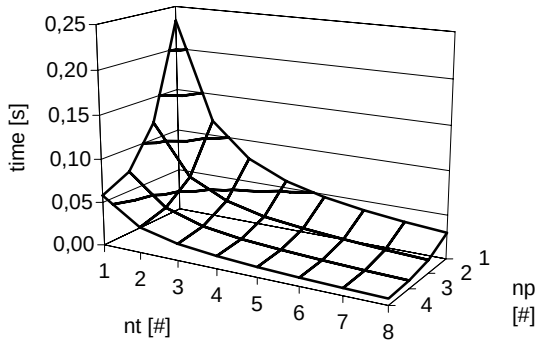
(b) Speedup of the *fold* skeleton.



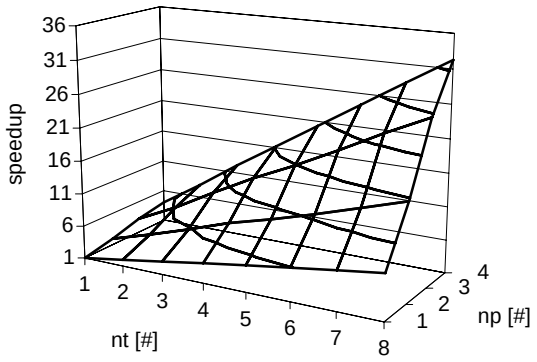
(c) Runtime of the *foldIndex* skeleton.



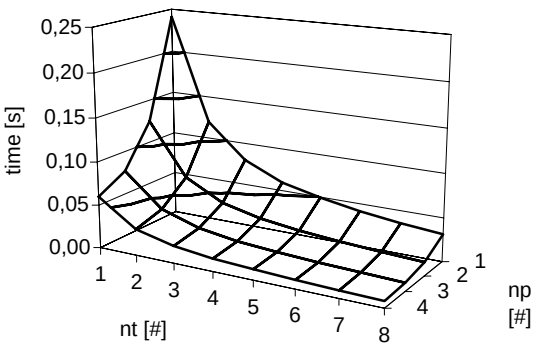
(d) Speedup of the *foldIndex* skeleton.



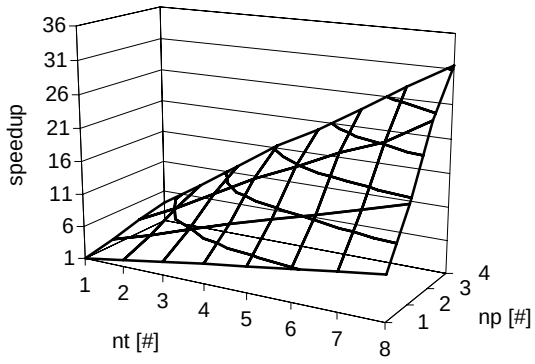
(e) Runtime of the *mapIndexInPlace* skeleton.



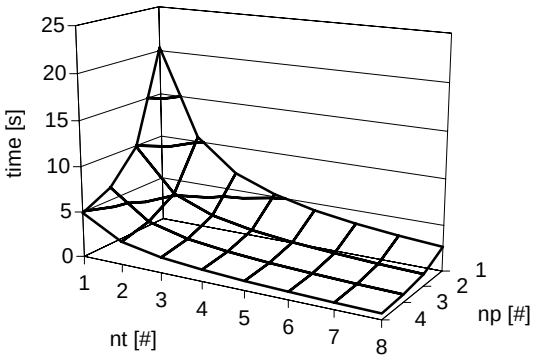
(f) Speedup of the *mapIndexInPlace* skeleton.



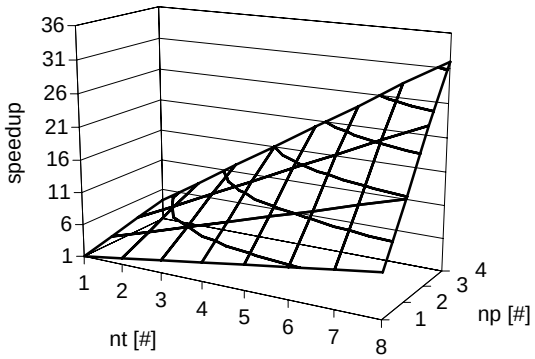
(g) Runtime of the *mapInPlace* skeleton.



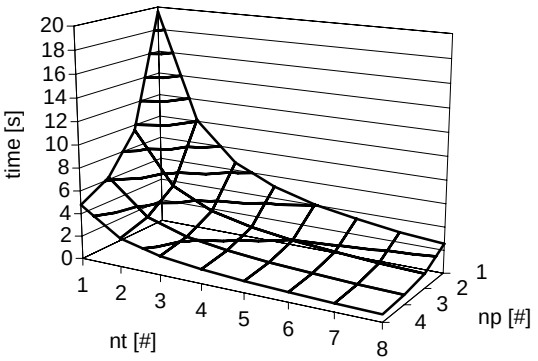
(h) Speedup of the *mapInPlace* skeleton.



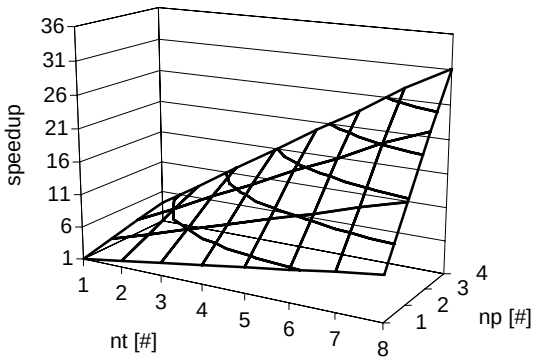
(i) Runtime of the *zipIndexInPlace* skeleton.



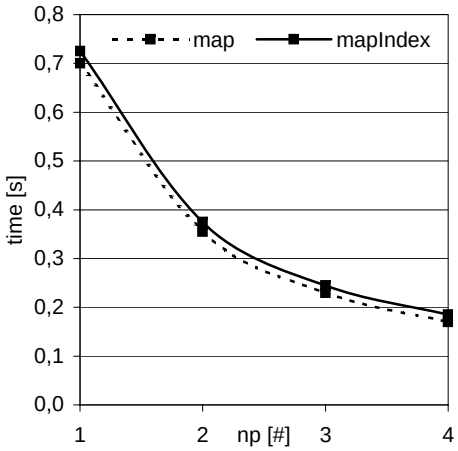
(j) Speedup of the *zipIndexInPlace* skeleton.



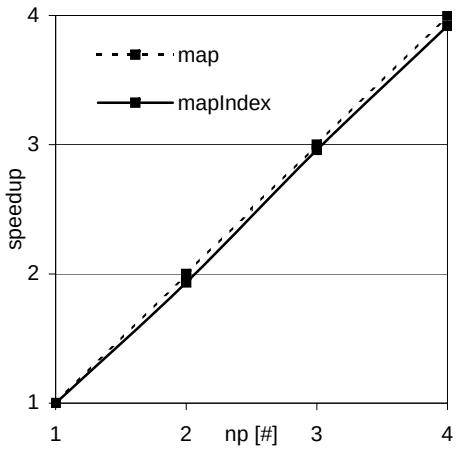
(k) Runtime of the *zipInPlace* skeleton.



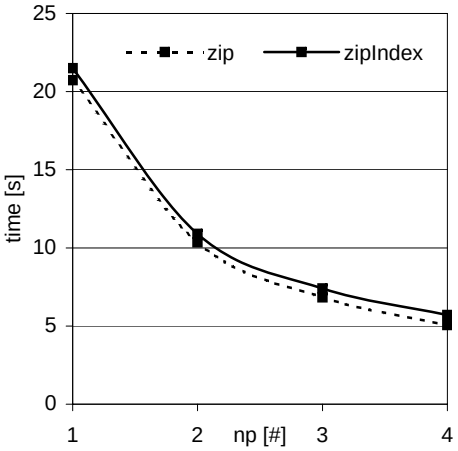
(l) Speedup of the *zipInPlace* skeleton.



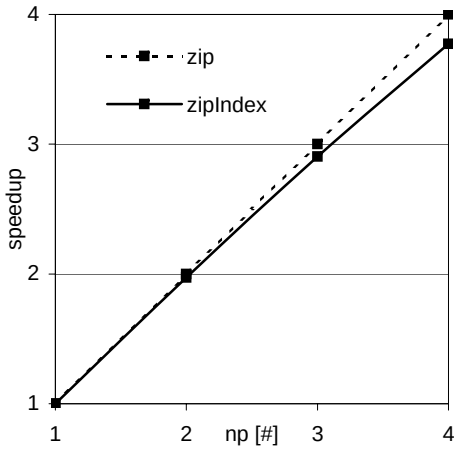
(m) Runtime of the *map* and the *mapIndex* skeletons.



(n) Speedup of the *map* and the *mapIndex* skeletons.



(o) Runtime of the *zip* and the *zipIndex* skeletons.



(p) Speedup of the *zip* and the *zipIndex* skeletons.

Figure 3.1: Test results of the benchmarks conducted on a multi-core cluster using up to $np = 4$ processors and $nt = 8$ threads. Note that the skeletons *map*, *mapIndex*, *zip*, and *zipIndex* were only benchmarked depending on np, since they are not enhanced by OpenMP directives.

Chapter 4

Task Parallel Skeletons

Most parallel applications are data parallel, and they can be handled with data parallel skeletons alone. However, in some cases more structure is required. Consider for instance an image processing application where a picture is first improved by applying several filters, then edges are detected, and finally objects formed by these edges are identified possibly by comparing them to a data base of known objects. Here, the mentioned stages could be connected by a pipeline where each stage processes a sequence of pieces of the picture and delivers its results to the next stage. Each stage could internally use data parallelism resulting in a two tier model, where the computation is first structured by task parallel skeletons like the mentioned pipeline and where atomic task parallel computations can be data parallel.

Task parallel skeletons provided by the *Muesli* skeleton library create a system of processes communicating via streams of data by nesting and combining predefined process topologies such as `Pipeline`, `Farm`, `DivideAndConquer`, and `BranchAndBound`. Moreover, *Muesli* provides some predefined building blocks which represent atomic task parallel computations, such as `Atomic`, `Filter`, `Initial` and `Final`. An atomic building block can be considered as a sort of *primitive skeleton*, since it can be provided with a problem specific argument function, which is internally applied to elements taken from a data stream. The argument function tells how each input value is transformed into an output value. This function can either be a C++ function or a partial application. However, in contrast to an algorithmic skeleton, an atomic building block does not represent a parallel programming pattern.

Both, a task parallel skeleton and an atomic building block, are represented as a class, which mainly provides a constructor and a method `start` to actually start the computation of the skeleton and all nested skeletons. Since all task parallel skeletons and basic building blocks offer the same interface, each of them can be represented by an instance of a subclass of the abstract class `Process`, which provides this interface (and a few auxiliary methods used internally in the skeleton implementation). Each task parallel skeleton has the same property as an atomic building block, namely it accepts a sequence of inputs and produces a sequence of outputs. This allows the task parallel skeletons to be arbitrarily nested. Most task parallel skeletons provided by *Muesli* consume at least one stream of input values and produce at least one stream of output values. An exception are the `Initial` and `Final` process provided by *Muesli*. They represent the source and the sink of a stream, respectively. Thus, an `Initial` process does not consume

any inputs, but only generates output values. In contrast, a `Final` process only consumes input values and does not produce any output values.

A task parallel application works in two steps. First, a process topology is created by using the constructors of the mentioned classes. This process topology reflects the actual nesting of skeletons. Then, the system of processes is started by calling the `start` method of the outermost skeleton. Internally, every atomic process will be assigned to a processor. For an implementation on top of SPMD, this means that every processor will dispatch depending on its rank to the code of its assigned process.

In the following, all task parallel algorithmic skeletons and their underlying building blocks provided by Muesli are described in detail.

4.1 Atomic Buildung Blocks

In the sense of Cole [3], an algorithmic skeleton describes the overall structure of an algorithm or a typical parallel programming pattern with gaps left for the definitions of problem specific procedures and declarations. An *atomic building block* can be seen as a sort of primitive skeleton, since it offers an interface for passing problem specific procedures and declarations, but neither represents a class of algorithms (and its underlying computation scheme) such as divide and conquer, nor a parallel programming pattern. An atomic building block refers to an atomic task parallel process within the process system which simply generates data streams or transforms elements of data streams by applying a user defined function. However, just like an algorithmic skeleton, such a component can be used as a building block for farms and pipelines. In the following, the atomic building blocks provided by *Muesli* are described in detail.

4.1.1 Initial

The `Initial` process represents the source of the data stream, and thus, it does not consume any input values, but only generates output values. An output value typically describes a problem that is to be solved by the parallel application. An output value is automatically routed to a successor of the `Initial` process corresponding to the process topology defined by the user.

```
template <class O>
class Initial: public Process {
    public:
        Initial(O* (* f)(Empty))
        void start()
}
```

The output values can be generated from scratch or read by an external source such as a file or a data base. How this primitive skeleton exactly generates the output values is described by the argument function `f`, which is passed to the constructor. The function `f` has no explicit arguments. The dummy argument `Empty` is inserted in order to simplify currying. However, the implementation of such an argument function will typically use some side effects like reading from a file in order to produce a pointer to an output value of type `O`. Internally, the `Initial` process repeatedly calls `f`, until `f` returns a `NULL` pointer, which indicates the end of the output stream and triggers the termination of the overall process system. In this case, all successors of the `Initial` process are informed that the output stream ends and the `Initial` skeleton terminates.

4.1.2 Final

In contrast to the `Initial` process, the `Final` process represents the sink of the data stream. This building block does not produce any output values, but only consumes input values. An input value represents the result of the computation which takes place within the process system.

```
template <class I>
class Final:  public Process {
    public:
        Final(void (* f)(I))
        void start()
}
```

The argument function `f` describes, what exactly has to be done with an input value of typ `I`. This function is internally applied to each of the received input values, until the data stream ends. Although the argument function `f` of the constructor does not explicitly return a result, `f` will typically cause some side effects such as writing results to a file.

4.1.3 Atomic

An object of the class `Atomic` represents a non-nested task parallel computation within the process system.

```
template <class I, class O>
class Atomic: public Process {
    public:
        Atomic(O* (* f)(I*), int n)
        void start()
}
```

This construct is able to consume input streams from different predecessors and generates at least one output stream. If several input streams are available, a fair reception of messages is guaranteed. The skeleton selects a stream and takes a new input value from it. An input value is transformed into an output value by applying the argument function `f`. The input and output values are of the data type `I` and `O`, respectively. An output value is automatically routed to a successor of the `Atomic` process corresponding to the user defined process topology. The function `f` is repeatedly applied to every value in all input streams. To avoid copying overhead, only a pointer to the input and output value is passed to and returned from `f`, respectively. The second argument `n` specifies the number of processors which shall be used for this process. In case of a purely task parallel computation, the only reasonable value for `n` is `1`. However, remember that the two-tier model allows data parallel computations inside of the `Atomic` process. In such a case `n` determinates the number of processors which collaborate in the nested data parallel computation.

4.1.4 Filter

As well as the `Atomic` process, the `Filter` process represents a non-nested task parallel computation within the process system. However, the mentioned `Atomic` process is restrictive in the sense that for each input value exactly one output value is produced. A `Filter` removes this restriction. For each input, an arbitrary number of output values (including `0`) can be generated.

```
template <class I, class O>
class Filter: public Process {
    public:
        Filter(void (* f)(Empty), int n)
        void start()
}

template <class I> I* MSL_get()

template <class O> void MSL_put(O* value)
```

The `Filter` process is able to consume and produce multiple input and output streams, respectively. In contrast to the `Atomic` process, the argument function `f` of the constructor is not repeatedly applied to each input value, but it is called only once. The argument function `f` of the `Filter` process has a dummy argument of type `Emtpy` (rather than type `void (* f)()`). This dummy argument is inserted in order to avoid technical difficulties when calling `Filter` with a partial application as argument. It is assumed that `f` uses the special auxiliary function `MSL_get()` in order to fetch the next input value of type `I` from an automatically selected stream and `MSL_put()` to route the computed output value of type `O` to one of the successors corresponding to the process topology. Note that the functions `MSL_get()` and `MSL_put()` do not need any additional information about where to fetch input and to deliver output values due to the fact that this information is obtained from the user defined process topology. This removes a nasty source of errors compared to usual message passing.

4.2 Skeletons

Besides a farm and a pipeline skeleton, which can be used to generate complex process systems, *Muesli* provides a divide and conquer as well as a branch and bound skeleton. In the following section, the application of each skeleton is described in detail. The focus is mainly on the skeleton interfaces rather than on implementation details and design aspects, because the latter is the content of several papers on which the current implementation of *Muesli* bases.

4.2.1 Branch & Bound

Branch and bound [53] is a well-known and frequently applied approach to solve certain optimization problems, among them integer and mixed-integer linear optimization problems [53] and the well-known traveling salesman problem [54]. Many practically important but NP-hard planning problems can be formulated as (mixed) integer optimization problems, e.g. production planning, crew scheduling, and vehicle routing. Branch and bound is often the only practically successful approach to solve these problems exactly.

A branch and bound algorithm searches the complete solution space of a given problem for the best solution. Due to the exponentially increasing number of feasible solutions, their explicit enumeration is often impossible in practice. However, the knowledge about the currently best solution, which is called *incumbent*, and the use of *bounds* for the function to be optimized enables the algorithm to search parts of the solution space only implicitly. During the solution process, a pool of yet unexplored subsets of the solution space, called the *work pool*, describes the current status of the search. Initially there is only one subset, namely the complete solution space, and the best solution found so far is infinity (for minimization problems). The unexplored subsets are represented as nodes in a dynamically generated search tree which initially only contains the root, and each iteration of the branch and bound algorithm processes one such node. This tree

is called the *state-space tree*. Each node in the state-space tree has associated data, called its *description*, which can be used to determine, whether it represents a *solution* and whether it has any successors.

Typically, a branch and bound problem is solved by applying a small set of basic rules. While the signature of these rules is always the same, the concrete formulation of the rules is problem dependent. Starting from a given problem, subproblems with pairwise disjoint state spaces are generated using an appropriate *branching rule*. The value of the best solution of a generated subproblem can be estimated by applying a *bounding rule*. Using a *selection rule*, the subproblem to be branched from next is chosen from the work pool. Last but not least subproblems with non-optimal or inadmissible solutions can be eliminated using an *elimination rule*.

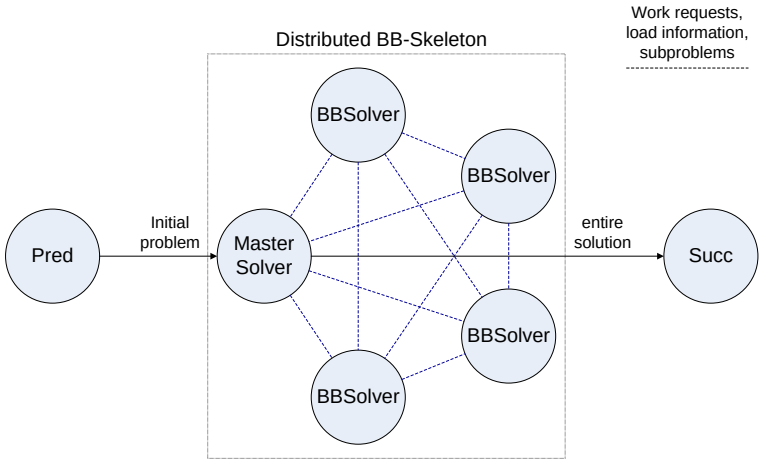


Figure 4.1: A fully distributed branch and bound skeleton

The `BranchAndBound` skeleton provided by *Muesli* offers a computation scheme for branch and bound algorithms to the user as predefined parallel component. Figure 4.1 illustrates the design of the skeleton. It consists of a set of peer solvers (in our example, $n = 5$ solvers are used), which exchange subproblems, work requests, and load information messages. A subproblem corresponds to an unexplored subset of the solution space. The work pool is distributed among the solvers, so that each of the solvers processes subproblems from its own local pool. Exactly one of the solvers, called the *master solver* serves as an interface to the branch and bound skeleton. The master solver receives a new optimization problem from a predecessor and delivers the solution to a successor. The master solver is able to consume multiple input streams and generates at least one output stream. The input and output values taken from and written to a stream are of the same type `I`. If several input streams are available, the master solver repeatedly selects one, takes a new optimization problem from the selected stream and initiates its solution. A fair reception of initial problems is guaranteed. The best solution is automatically routed to a successor of the `BranchAndBound` skeleton corresponding to the user defined process topology. In contrast to the divide and conquer skeleton (cf. 4.2.2), the `BranchAndBound` skeleton processes only one optimization problem at a time in order to avoid memory problems.

```
template <class I>
class BranchAndBound: public Process {
    public:
        BranchAndBound(I** (*branch)(I*,int*),
            void (*bound)(I*),
            bool (*betterThan)(I*,I*),
            bool (*isSolution)(I*),
            int (*getLowerBound)(I*),
            int d, int size)
        void start()
}
```

The user has to provide the BranchAndBound skeleton with five basic operators: branch, bound, betterThan, isSolution, and getLowerBound. The branch operator has to implement a branching rule, that is, given an initial problem representing an unexplored subset of the solution space, branch has to generate subproblems with pairwise disjoint state spaces. The bound operator has to implement a bounding rule in order to estimate a given subproblem. The corresponding lower bound must be returned by getLowerBound. betterThan is used to compare two estimated subsets of the solution space, and isSolution is used to discover, whether its argument is a solution to the problem or not. The basic operators are described in detail later. In addition to the argument functions, the user has to pass two int parameters d and size. The parameter d specifies the degree of the state space tree and describes the maximum number of subproblems generated by branch. The actual number of subproblems generated by branch may vary depending on the problem and must not exceed d, but may be lesser than d. The argument size determines the number of solvers (including the master solver) which are used to solve the optimization problems in parallel. The solvers are automatically generated and connected by the BranchAndBound skeleton. If the skeleton only consists of a single solver there is no need for any load balancing. In this case, all communication parts are bypassed to speed up the computation.

Each worker repeatedly executes two phases: a communication phase and a solution phase. Let us first consider the communication phase. In order to avoid that computation time is wasted with the solution of irrelevant subproblems, new best solutions are distributed among the solvers directly. If a solver has received new incumbents, it stores the best and discards the others. Subproblems whose lower bounds are worse than the incumbent are removed from the work pool. From time to time, the solvers exchange information about the quality and the number of the problems stored in their local pools. This knowledge sharing triggers the load distribution and is used to continuously mix the work pools of the solvers in order to provide each solver with worthwhile problems. This guarantees that the problems with the best lower (upper) bounds are distributed periodically among the solvers which significantly speeds up the overall computation. The solution phase starts with selecting an unexamined subproblem from the work pool. The work pool is organized as a heap and the selection rule implements a best-first search strategy. The selected problem is decomposed into m subproblems by applying branch. For each of the subproblems, we proceed as follows. First, we check, whether it is solved. If a new best solution is detected, we update the local incumbent and broadcast it. A worse solution is discarded. If the subproblem is not yet solved, the bound function is applied to compute a new lower (upper) bound. After this, we check again whether the problem is solved in order to support branch and bound algorithms which produce solutions by applying bound. If required, new incumbents are stored locally and distributed again. An unsolved subproblem is only stored in the work pool, if its lower bound is actually better than the incumbent. In this case, solving this subproblem may (but needs not) lead to a new incumbent. Otherwise, the best solution to the subproblem cannot be better than the best solution found so far, so that the subproblem can be discarded.

In the following, the basic operators, which have to be implemented by the user, are described in detail.

```
I** (*branch)(I* p, int* size)
```

The branch operation describes how to divide a given problem p of type I into subproblems of the same type. A problem corresponds to an unexplored subset of the solution space which has to be divided into subproblems with pairwise disjoint state spaces. To avoid copying overhead, a pointer to the problem is passed to this operation. The user has to generate the subproblems and prepare an array of pointers to the subproblems. The return value of this method is a pointer to this array of pointers. The subproblems should be generated by the new operator. If the array contains pointers to local variables, an access violation error is to be expected at runtime. Due to the fact that the number of generated subproblems may vary depending on the problem passed to branch, the skeleton needs an

additional information about the size of the array of subproblems. For this reason, `branch` offers a second parameter `int* size`. By setting `*size` to the number of generated problems, the skeleton is informed about the length of the array returned by `branch`. The memory management is undertaken by the skeleton. Thus, the user does not have to care about the deallocation of memory for the passed problem or the generated subproblems. However, the user is responsible for deleting any other object which is created within the body of `branch`.

```
void (*bound)(I* p)
```

The `bound` method is used to compute a lower (upper) bound for the best solution to a given minimization (maximization) problem `p` of type `I`. To avoid copying overhead, a pointer to the problem is passed to this operation. This method is internally used to prune the work pool and thus to search only parts of the solution space by comparing the computed bound with the incumbent. If the best solution found so far is better (in the sense of `betterThan`) than the lower (upper) bound for the best solution which can be found by solving `p`, then it is not necessary to consider `p` any more and the problem can be discarded.

```
bool (*betterThan)(I* p1, I* p2)
```

This method compares two given problems `p1` and `p2` of type `I` and has to deliver `true`, iff the lower (upper) bound for the best solution to problem `p1` is better than the lower (upper) bound for the best solution to `p2` in case of a minimization (maximization) problem. Otherwise `false` must be returned. Internally, this method is used to maintain the local work pool which is implemented as a heap. The root of the heap refers to the problem with the lowest bound in the work pool. Thus, the selection rule provided by the skeleton implements a best-first search strategy. Moreover, this method is used to compare a problem with the incumbent in order to decide if the problem can be discarded.

```
bool (*isSolution)(I* p)
```

The `isSolution` operator has to return `true`, if its argument `p` represents a solution of the optimization problem. Otherwise `false` must be returned. Again, only a pointer to the problem is passed to the method in order to avoid copying overhead.

```
int (*getLowerBound)(I* p)
```

This method simply has to return a lower (upper) bound of the value of the best solution of the problem `p`. Internally, it is used to create load information messages. In the current implementation the bounds are represented as `int` values.

4.2.2 Divide & Conquer

Divide and conquer is a common computation paradigm, in which the solution to a problem is obtained by dividing the original problem into smaller subproblems and solving the subproblems recursively. Then, solutions for the subproblems must be combined to form the final solution of the entire problem. Simple problems are solved directly without dividing them further.

Examples of divide and conquer computations include various sorting methods such as mergesort and quicksort, computational geometry algorithms such as the construction of the convex hull, combinatorial search such as constraint satisfaction techniques, graph algorithmic problems such as graph coloring, numerical methods such as the Karatsuba multiplication algorithm, and linear algebra such as Strassen's algorithm for matrix multiplication. Some applications require solving several divide and conquer problems in sequence. Examples here are the 2D or 3D triangulation of several geometric objects, matrix chain multiplication problems, in which parts of the chain can be computed independently from each other, or factorization of several large numbers.

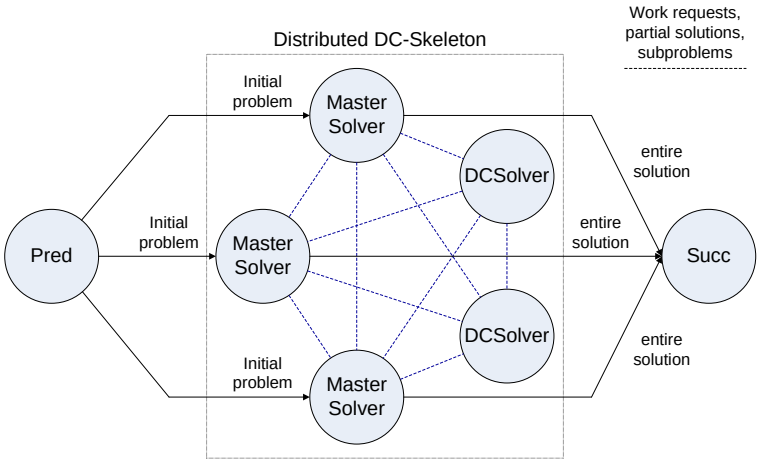


Figure 4.2: A fully distributed divide and conquer skeleton for stream processing

The `StreamDC` skeleton provided by *Muesli* is based on an implementation scheme for divide and conquer and offers it to the user as predefined parallel component. Figure 4.2 illustrates the design of the skeleton. It consists of a set of peer solvers, which exchange subproblems, partial solutions, and work requests. In our example, $n = 5$ solvers are used. The work pool and the solution pool, which are used to maintain the subproblems and partial solutions, are distributed among the solvers, and each of the solvers processes subproblems and partial solutions from its own local pools. If a solver finds it own work pool empty, it sends a work request to a randomly selected neighbor corresponding to the given internal topology, which triggers the load distribution. If the work pool of the receiver is not empty, it selects a subproblem from the work pool which is expected to be big and delegates it to the sender. Exactly `n` of the solvers serve as an interface to the skeleton, which are referred to as *master solvers*. A master solver receives a new divide and conquer problem from a predecessor and delivers the solution to its successor. By using more than one master solver, the skeleton is able to process several divide and conquer problems at a time.

```
template <class I, class O>
class StreamDC: public Process {
    public:
        StreamDC(I** (*divide)(I*),
                 O* (*combine)(O**),
                 O* (*solve)(I*),
                 bool (*isSimple)(I*),
                 int d, int size, int probs)
        void start()
}
```

The user has to provide the skeleton with four basic operators: `divide`, `combine`, `isSimple`, and `solve`. If `isSimple` indicates that a problem is simple enough, it can be solved directly by applying `solve`. Otherwise, the problem is divided into subproblems by calling `divide`. Solutions

of subproblems can be combined to the solution of the corresponding parent problem by applying `combine`. The basic operators are described in detail later. In addition to the argument functions, the user has to pass three `int` parameters `d`, `size`, and `n`. The parameter `d` specifies the degree of the divide and conquer tree and describes how many subproblems are generated by `divide` and how many partial solutions are required by `combine`. The argument `size` determines the number of solvers (including master solvers) which are used to solve the divide and conquer problems in parallel. The solvers are automatically generated and connected by the `StreamDC` skeleton. Thus, the solvers are not visible to the user. If the skeleton only consists of a single solver there is no need for load balancing. In this case, all communication parts are bypassed to speed up the computation. Finally, `n` declares how many master solvers should be used by the skeleton, and thus, how many divide and conquer problems may be solved in parallel by the skeleton at a time. Note that `n` must be greater than `0` and less or equal than `size`. The number of master solvers corresponds to the number of entrances and exits of this skeleton. If only one big divide and conquer problem has to be solved, the only reasonable value for `n` is `1`. Using more than one master solver is recommended, if a sequence of divide and conquer problems must be processed. In this way it is possible to adapt the number of problems to be solved in parallel to the available distributed memory offered by the solvers, as well as to speed up the computation significantly.

Each master solver is able to consume multiple input streams and generates at least one output stream. If several input streams are available, a master solver repeatedly selects one, takes a new divide and conquer problem from the selected stream and initiates its solution. A fair reception of messages is guaranteed. The input and output values taken from and written to a stream are of the data type `I` and `O`, respectively. The solution of a divide and conquer problem is automatically routed to a successor of the `StreamDC` skeleton corresponding to the user defined process topology.

A detailed explanation of the design and implementation of the `StreamDC` skeleton and its components can be found in [37, 38, 39]. In the following, the basic operators, which have to be implemented by the user, are described in detail.

```
I** (*divide)(I*)
```

The `divide` operation describes how to divide a given problem of type `I` into subproblems of the same type. To avoid copying overhead, a pointer to the problem is passed to this operation. The user has to generate the subproblems and prepare an array of pointers to the subproblems. The array size must match the parameter `d`, which specifies the degree of the divide and conquer tree. The return value of this method is a pointer to this array of pointers. The subproblems should be generated by the `new` operator. If the array contains pointers to local variables, an access violation error is to be expected at runtime. The memory management is undertaken by the skeleton. Thus, the user does not have to care about the deallocation of memory for the passed problem or the generated subproblems. However, the user is responsible for deleting any other object which is created within the body of `divide`.

```
O* (*combine)(O**)
```

The `combine` operation specifies the strategy of combining partial solutions of type `O` to the corresponded solution of the parent problem, which is of type `O` as well. To avoid copying overhead, only a pointer to an array is passed to this method, which contains pointers to the partial solutions. The size of this array is implicitely given by the parameter `d` which determines the degree of the divide and conquer tree. The user has to create the parent solution and return a pointer to it. As for the `divide` operator, the user should not return a pointer to a local variable in order to avoid memory access violation errors. The memory

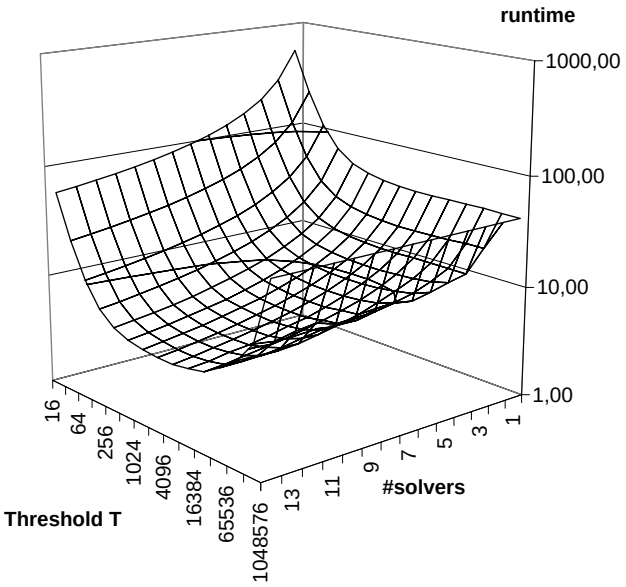


Figure 4.3: Karatsuba runtimes depending on T

management for the partial solutions and the return value is undertaken by the skeleton. Thus, the user does not have to care about the deallocation of memory for the passed problem or the generated subproblems. However, the user is responsible for deleting any other object which is created within the body of `divide`.

`O* (*solve)(I*)`

The `solve` operator typically implements a sequential divide and conquer algorithm and is applied on problems which are identified as simple problems by `isSimple`. The problem to be solved is referenced by a pointer `I*` which is passed to this method by the skeleton. The return value of `solve` is a pointer `O*` to the corresponding solution. The user has to avoid returning a pointer to a local variable due to memory access violation errors. As for `divide` and `combine`, the memory management is undertaken by the skeleton. Thus, the user does not have to care about the deallocation of memory for the passed problem or the generated solution, but only for objects which are created within the body of `solve` for any other reason. It is possible to solve the problem *in place* if the problem and its solution are of the same type `T` (which is the case for sorting problems, for instance). In this case, the pointer to the problem references the solution as well, and can be returned by `solve`. If the pointer to the problem is not equal to the pointer returned by this method, the problem is assumed not to be required any more, and the memory for this problem is freed by the skeleton.

`bool` `(*isSimple)(I*)`

This function indicates whether a problem is simple enough to be solved directly or it needs to be divided it into subproblems. Given a pointer to the problem of type `I`, this method has to return `true` for simple problems and `false` for problem which have to be divided further. Even though this is a very simple method which typically can be implemented in a very few lines of code, it greatly influences the runtime of the considered application. Thus, it must be carefully implemented by the user.

Dividing problems and maintaining them in a work pool introduces overhead. It appears reasonable to solve subproblems locally by calling a sequential (divide and conquer) algorithm

at a time when the subproblem sizes have reached a certain threshold T. Unfortunately, the specific value of T is problem dependent and therefore hard to predict reliably by the user. If T is large, only few big subproblems are generated and distributed among the solvers, which can lead to an unbalanced load distribution. Thus, the threshold T has to be chosen small enough such that a sufficient number of subproblems is generated to ensure a good load balancing. Dividing a problem causes costs for subproblem generation, combination, and maintenance. Thus, T has to limit the total number of generated subproblems. In either case, T determines the depth of the divide and conquer tree and the number of leaf nodes stored in the work pool, i.e. subproblems, which are solved sequentially. Using the Karatsuba multiplication algorithm for big integers [55] as example, Fig. 4.3 reveals that a bad choise of T typically involves high performance penalties. If only one solver is used, it is recommended to enable a purely sequential computation by adjusting T to the size of the initial problem. In this case, the initial problem is instantly identified as simple enough to solve it directly by the user defined `solve` operator. Thus, there is no need for a `divide` or `combine` operator call at all, and the work and solution pool are bypassed as well. The more solvers are used, the more subproblems must be generated to assure a good load balancing. In the considered example application, where we have generated and multiplied two numbers with $2^{20} = 1048576$ digits for each test run, $T = 4096$ is a good choise, that is, a problem of multiplicating two numbers with $T = 4096$ digits is identified as a simple problem and thus solved sequentially with `solve`. This leads to 6561 subproblems distributed among the solvers.

4.2.3 Pipe

The `Pipe` skeleton represents a logical predecessor-successor-relationship between skeletons. Building a pipeline of skeletons is the easiest form of coordination in *Muesli*.

```
class Pipe: public Process {
    public:
        Pipe(Process& p1, Process& p2)
        Pipe(Process& p1, Process& p2, Process& p3)
        ...
        void start()
}
```

The constructor `Pipe` creates a pipeline of skeletons. The constructor is overloaded and the constructor which is called depends on the number of arguments that are passed to it. The processes passed to the constructor may be basic building blocks such as `Initial` or `Final`, skeletons such as `DivideAndConquer` or `BranchAndBound`, or nested skeletons such a `Farm` or even another `Pipe`. The `Pipe` skeleton connects the passed skeletons and enables the data transfer between them, including a fair load balancing. The output values of `p1` are routed to `p2`, and, in the case of the second constructor, the output values of `p2` are routed to `p3`. For this reason, each `Process` (except for `Initial` and `Final`) offers at least one entrance to which a data stream can be sent, as well as at least one exit, from which a data stream is sent. Each entrance and exit is mapped to a processor which is part of the skeleton or building block, repectively. The `Pipe` skeleton connects each exit of `p1` with each entrance offered by `p2`. `p2` and `p3` are connected in the same manner (Fig. 4.4). The entrances and exits of the pipeline are represented by the entrances of the first process and the exits of the last process passed to the constructor. If the first and last process in the pipeline are an `Initial` and `Final` process, respectively, the pipeline neither has an entrance nor an exit.

Note, that the interface of two adjacent skeletons within the pipeline must match the data type of the exchanged data. If, for instance, `p1` generates output values of type `T` at its exits, `p2` must

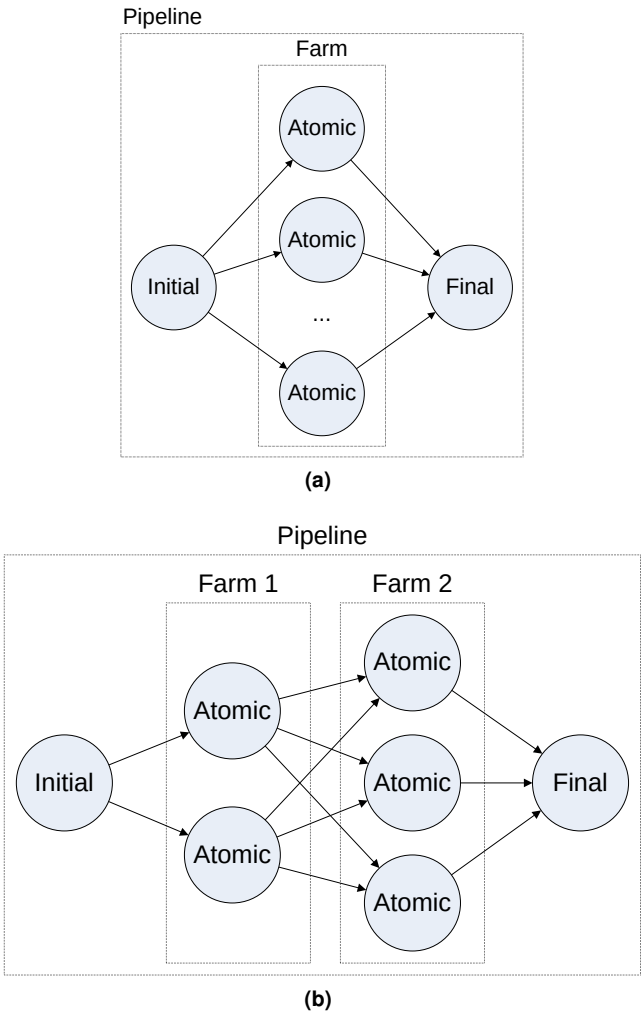


Figure 4.4: A farm skeleton (left) and a pipeline of farms (right).

accept input values of type `T` at its entrances as well. Otherwise a compiler time error will occur. Due to the fact that all entrances and exits of two adjacent skeletons are connected to each other, it is possible to connect two farms of different sizes, or farms with basic building blocks. The `Pipe` skeleton can be used to combine simple skeletons to a more complex skeleton. By passing a pipeline to the constructor of `Farm` or another `Pipe` skeleton, complex process topologies can be generated (Fig. 4.4).

4.2.4 Farm

A `Farm` consists of a set of peer worker processes, which solve problems independently from each other. The use of a farm is expedient to speed up the overall computation, if a multiplicity of problems of the same type have to be solved. In this case, the problems are automatically distributed among the workers, so that they can be processed in parallel.

In many farm implementations, the internal farm topology is based on a *master/worker* scheme. In this case, a farm consists of a *farmer* process and several *worker* processes. The farmer accepts a sequence of tasks from some predecessor process and propagates each task to a worker. The worker executes the task and delivers the result back to the farmer who propagates it to a

successor process. As we have shown in [34, 36], this design comes with several disadvantages such as high communication costs, which quickly results in a bottleneck situation at the farmer process. In *Muesli*, a fully distributed implementation scheme is used (Fig. 4.4a). The *farmer* process is omitted, and the skeleton only consists of a set of peer worker processes. This approach reduces the overhead for the propagation of messages and omits a potential source of a bottleneck in the user defined process system.

```
template <class I, class O>
class Farm: public Process {
    public:
        Farm(Process& worker, int n)
        void start()
}
```

The constructor `Farm` takes an arbitrary process (an atomic task parallel process or another skeleton) and generates additional `n-1` copies of it, so that the farm finally consists of `n` workers. Each of the workers serves as an interface to the `Farm`. Thus, the type of the input and output values `I` and `O`, which are sent to and from the farm, must match the types of the input and output values of worker processes. The entrances and exits of the farm result from the entrances and exits of each worker.

The `Farm` skeleton can be arbitrarily nested with pipelines or other farms. For instance, it is possible to construct a pipeline of farms, as depicted in Fig. 4.4b. A sequence of farms may, for instance, make sense, if each worker of the first farm uses one processor, while each worker of the second farm performs a data parallel computation on several processors. Note that the sequence of the output values produced by the farm is non-deterministic because the workers produce its outputs indepentendly from each other (Fig. 4.4).

Chapter 5

Selected Implementation Aspects

5.1 Serialization

Object serialization is an important issue in the context of data storage and transmission due to the fact that the objects to be interchanged among task or data parallel skeletons often have a dynamic size or contain pointer structures. In this case, it is necessary to write the object data to a contiguous memory block before sending it over the network, and to restore the object at the receivers' site. In contrast to languages such as (Object) Pascal or Java, C++ does not inherently support object serialization. Due to the lack of runtime metadata, serialization is a difficult feature to implement in C++. Several approaches to overcome this problem have been proposed, among them C++ language extentions and C++ serialization libraries based on XML and/or binary formats. The C++ dialect *opC++* extends the standard C++ language with additional concepts such as reflection and serialization. *Xparam* [56] is a general-purpose tool for parameter handling in C++. It allows object serialization and deserialization in a format that is human-readable and -writeable, and is unaffected by issues of word-size and endianity. *Sweet Persist* is a C++ serialization library that provides serialization of objects to and from XML and binary formats. It requires Microsoft Visual Studio 2005 (MSVC 8.0) and Boost [57, 58]. Another library that provides serialization is *GenSerial* [59] which currently only works in Visual C++ .NET 2003.

To perform object serialization in a platform independent manner, *Muesli* provides the abstract class `MSL_Serializable`.

```
class MSL_Serializable {
    public:
        MSL_Serializable() {}
        virtual ~MSL_Serializable() {}
        virtual void reduce(void* pBuffer, int bufferSize) = 0;
        virtual void expand(void* pBuffer, int bufferSize) = 0;
        virtual int getSize() = 0;
};
```

An important objective is to avoid runtime overhead for basic data types or types that do not require serialization. Each class, whose instances represent objects which have to be transmitted serialized, must be derived from `MSL_Serializable` and implement the inherited methods `reduce`, `expand`, and `getSize`. Objects which are not derived from `MSL_Serializable` are supposed to be already serialized, such as pointerless C++ structures or basic data types. In this case, there is no need for an additional serialization (the object data is already stored in a contiguous memory block), and the objects can be directly transmitted to the receiver.

For the serialized communication operations, *Muesli* allocates a contiguous memory block corresponding to the size of the object to be serialized. The method `reduce` is used to write the object data to the buffer, whereas `expand` reads the object data from the buffer and restores the object. The concrete size of the buffer which is required to store the serialized object has to be calculated by the method `getSize`. *Muesli* internally allocates a send buffer matching this size and calls `reduce` to write the data into the buffer immediately before sending out the packet. The address and the size of the send buffer are passed to `reduce` by the parameters `pBuffer` and `bufferSize`, respectively, which can be used by the user to copy the required object data to the buffer. The method `expand` is called immediately after receiving a serialized message. Here, `pBuffer` and `bufferSize` identify the address and size of the receive buffer, which stores the serialized object data. The object can be restored by the user from the receive buffer by reading the object data in the same sequence in which the data has been written to the send buffer. Please note, that the send and receive buffer cannot be reused because each send and receive operation create a new buffer for the data transmission. After a send and receive operation has been completed, the memory for the send and receive buffer, respectively, is deallocated immediately. That is, when restoring an object with `expand` it is strongly recommended to copy the data explicitly from the receive buffer and not to point into the address space of the receive buffer. Otherwise memory access violation errors can occur.

The key for our serialization mechanism is the detection of an inheritance relationship at compile time [60]. This problem can be solved with the aid of the `sizeof` operator, which is surprisingly powerful because it can be applied to any expression irrespective of its complexity. `sizeof` returns the size of an expression without evaluating it at runtime.

The detection of conversions is based on the application of `sizeof` to a function which is overloaded two times. The first implementation of this function accepts a type which can be converted to `U`, the second one accepts everything else. Then, the overloaded function is applied on a temporary element of type `T`. If the first function is selected by the compiler, `T` can be converted to `U`, otherwise the second function is called. In order to detect which function is applied, the overloaded functions are provided with a return type of different size. The differentiation is then carried out with `sizeof`.

At first, we generate two types of different size. In general, basic data types such as `char` and `long double` have a different size. However, this property is not guaranteed by the C++ standard, so we use this simple scheme instead:

```
typedef char Small;             // sizeof(Small) = 1
class Big { char dummy[2]; };   // sizeof(Big) > 1
```

Moreover, we need two overloaded functions. The first one accepts a type `U`, the second one accepts everything else:

```
static Small Test(U);
static Big Test(...);
```

Passing a C++ object to a function with an ellipse[1] leads to an undefined result. However, the function is neither applied nor implemented, because `sizeof` does not evaluate the arguments of the function. In the next step, the `Test` function is applied to a temporary element of type `T`, and the return value is passed to the `sizeof` operator. A problem is to create an object of type `T`, if the standard constructor of `T` is private. This problem is solved with a simple dummy function

```
static T MakeT();
```

[1] denotes a non-specified number of parameters

so that we can apply:

```
const bool exists = sizeof(Test(MakeT())) == sizeof(Small);
```

Please remember, that `sizeof` does not evaluate any arguments. Moreover, `MakeT` and `Test` do not do anything and are not even implemented.

This is all provided as a class template in order to hide the details of the type derivation. The constant `sameType` of the class `MSL_Conversion` is used to detect, whether `T` and `U` have the same type.

```
#define MSL_IS_SUPERCLASS(T, U)
        (MSL_Conversion<const U*, const T*>::exists &&
        !MSL_Conversion<const T*, const void*>::sameType)

template <class T, class U>
class MSL_Conversion {
  private:
    typedef char Small;
    class Big { char dummy[2]; };
    static Small Test(U);
    static Big Test(...);
    static T MakeT();
  public:
    enum { exists = sizeof(Test(MakeT())) == sizeof(Small) };
    enum { sameType = false };
};

template <class T>
class MSL_Conversion<T,T> {
  public:
  enum { exists = true, sameType = true};
};
```

`MSL_IS_SUPERCLASS` determines the convertibility of `const U*` to `const T*`. The operation `MSL_IS_SUPERCLASS(T,U)` returns `true`, iff `U` is derived from `T` or if `T` and `U` are of the same type.

Internally, *Muesli* provides implementations of a serialized `MSL_Send` and `MSL_Receive` operation used for transmitting instances of classes derived from `MSL_Serializable`, as well as non-serialized `MSL_Send` and `MSL_Receive` communication operations for already serialized objects. Within the skeleton source code, only the wrapper methods `MSL_Send` and `MSL_Receive` are used, which are each replaced by the required serialized or non-serialized operation at compile time in order to avoid performance penalties at runtime. A problem is, that the signature of both, the serialized and non-serialized implementation of the `MSL_Send` (`MSL_Receive`) operation, is identical:

```
MSL_Send(ProcessorNo destination, Data* pData, int tag)
```

Thus, `MSL_Send` (`MSL_Receive`) cannot be overloaded directly. The solution of this problem is to extend the signature of these functions by an additional type parameter. A simple template proves helpful:

```
template <bool v>
struct MSL_Bool2Type { enum {value = v}; };
```

MSL_Int2Type creates a different type for each integer constant, which is passed to this template. This is due to the fact that different template instances are different types. Therefore, MSL_Int2Type<**true**> differs from MSL_Int2Type<**false**>. The type generating value v is stored in the enum variable value.

By extending the signature of the serialized and non-serialized send and receive operation, the compiler is able to distinguish between both implementations, and MSL_Send as well as (MSL_Receive) can be overloaded. This is shown at the example of MSL_Send in the following. MSL_Receive is implemented in the same way.

```
// for objects to be serialized
template <class Data>
inline void MSL_Send(ProcessorNo destination,
                     Data* pData, int tag,
                     MSL_Int2Type<true>) {
    int size = pData->getSize();
    void* buffer = malloc(size);
    if (buffer == NULL)
        std::cout << "OUT OF MEMORY ERROR" << std::endl;
    pData->reduce(buffer,size);
    MPI_Send(buffer, size, MPI_BYTE, destination, tag, MPI_COMM_WORLD);
    free(buffer);
}

// for already serialized objects
template <class Data>
inline void MSL_Send(ProcessorNo destination,
                     Data* pData, int tag,
                     MSL_Int2Type<false>) {
    MPI_Send(pData, sizeof(Data), MPI_BYTE, destination,
                                        tag, MPI_COMM_WORLD);
}
```

The corresponding wrapper for MSL_Send is:

```
template <class Data>
inline void MSL_Send(ProcessorNo destination,
                     Data* pData, int tag = MSLT_MYTAG) {
    if (destination == MSL_UNDEFINED)
        throws(UndefinedDestinationException());
    MSL_Send(destination, pData, tag,
        MSL_Int2Type<MSL_IS_SUPERCLASS(MSL_Serializable, Data)>());
}
```

The expression

```
MSL_Int2Type<MSL_IS_SUPERCLASS(MSL_Serializable, Data)>())
```

is reduced by the compiler to MSL_Int2Type<**true**> if the class Data is derived from MSL_Serializable, otherwise the result is MSL_Int2Type<**false**>. Thus,

```
MSL_Send(destination, pData, tag,
         MSL_Int2Type<MSL_IS_SUPERCLASS(MSL_Serializable, Data)>());
```

is replaced (due to the inline operator) by the required implementation depending on Data at compile time.

5.2 DistributedSparseMatrix<E,S,D>

Having established the conceptual basis in Section 2 this section describes the implementation details of our data structure. Most of the aforementioned flexibility is implemented using three core mechanisms of C++, namely template parameters, inheritance, and polymorphism. Listing 2 shows the class definition of our data structure introducing the template parameters `E`, `S` and `D`. These template parameters are used to flexibly define the data type of the matrix elements (`E`), the compression scheme of the submatrices (`S`), and their distribution scheme (`D`). All three parameters are provided with default values in case the user does not want to provide specific ones. Here, the default values are **`double`** for the type, `CrsSubmatrix<E>` for the compression scheme and `RoundRobinDistribution` for the distribution scheme. Both of the two classes are explained in detail in the following subsections.

```
1 template<class E = double,
2          class S = CrsSubmatrix<E>,
3          class D = RoundRobinDistribution>
4 class DistributedSparseMatrix {...};
```

Listing 2: Definition of the class `DistributedSparseMatrix` showing the three template parameters `E`, `S` and `D`.

As with all our task and data parallel skeletons, our data structure is defined in the file `Muesli.h`[2]. Using it is very convenient, one simply has to include the header file with the **`#include`** compiler directive. Listing 3 shows how to declare a distributed sparse matrix using default values for the template parameters and how to declare a matrix using specific arguments. The classes `BsrSubmatrix` and `BlockDistribution` are explained in detail in the following subsections.

Besides providing support for algorithmic skeletons our data structure offers functions for rotating its rows and columns, to multiply the matrix with a row vector and to write the whole or a part of the sparse matrix to standard output. The constructors of our data structure can be used to create a distributed sparse matrix by means of a given two-dimensional array of type `E`, to create an empty matrix which is successively filled by calls to the function `setElement` (cf. Section 5.3), or to create a new distributed sparse matrix by copying an existing matrix.

```
1 #include "Muesli.h"
2
3 void main(int argc, char** argv) {
4   int n = 100, m = 100, r = 10, c = 10;
5   DistributedSparseMatrix< > A(n, m, r, c);
6   DistributedSparseMatrix<double, BsrSubmatrix<double>,
7     BlockDistribution> B(n, m, r, c);
8 }
```

Listing 3: Declaration of two distributed sparse matrices. Matrix A uses default arguments, whereas matrix B is instantiated using specific arguments.

[2]Note that due to the usage of template parameters, separating the source code into a header and a source file is not possible. In fact, we use the inclusion model such that the header file contains all source code.

5.3 Submatrix<E>

As already mentioned in Section 3.4.1.1, the whole sparse matrix is partitioned into multiple submatrices. Each submatrix is responsible for storing and compressing its local elements such that arbitrary compression schemes can easily be supported. In order to do so, we provide an interface in terms of an abstract class which is defined in the file `Submatrix.h`. Listing 4 shows most of its implementation details omitting only trivial attributes and functions.

Apart from attributes for its ID, the number of non-zero elements and the local dimension of the submatrix (cf. line 2) the class provides a vector from the C++ Standard Template Library [43] in order to store locally available elements (cf. line 3). Again, the type of these elements is passed as a template argument. The most important part of this class is shown in lines 5 and 6. Here, two pure virtual functions[3] are declared in order to access locally stored elements of the submatrix. Both of them expect local row and column indexes, respectively and must be overridden in any concrete subclass as they are responsible for handling the implemented compression scheme. However, sometimes it is useful to bypass the compression scheme and simply work on the raw values stored by the submatrix. This can be done using the functions `getElementLocal` and `setElementLocal` (cf. lines 8 and 9). Both of them expect an index referring to a position in the `values` array declared in line 3. In order to determine the local row and column indexes of an element which is accessed using these functions, one can use the functions `getColumnIndexLocal` and `getRowIndexLocal`, respectively (cf. lines 11 and 12). Note that both of them again are declared as pure virtual. The `init` functions declared in lines 14–18 are used to initialize the submatrix: The first one initializes an empty submatrix, the second one initializes the submatrix by means of the given two-dimensional array `matrix`, and the third one only expects a single value and its coordinates to initialize the submatrix. Again, it is mandatory to override these functions.

```
1  template<class E = double> class Submatrix {
2    int id, nnz, nLocal, mLocal;
3    std::vector<E> values;
4
5    virtual E    getElement(int row, int col)  const  = 0;
6    virtual void setElement(int row, int col, E val)  = 0;
7
8    E getElementLocal(int index)  const;
9    E setElementLocal(int index, E val);
10
11   virtual int getColumnIndexLocal(int index) const  = 0;
12   virtual int getRowIndexLocal(int index)    const  = 0;
13
14   virtual void init(int id, int nLocal, int mLocal) = 0;
15   virtual void init(int id, int nLocal, int mLocal,
16     E** const matrix)                               = 0;
17   virtual void init(int id, int nLocal, int mLocal,
18     E val, int row, int col)                        = 0;
19 };
```

Listing 4: Definition of the class `Submatrix`. For the sake of clarity trivial attributes and functions are omitted.

By using mechanisms such as polymorphism and dynamic binding our data structure can offer a lot of flexibility. This additional layer of abstraction is very useful, since we can internally work with the abstract class `Submatrix`. Thus, extending our data structure with arbitrary user-defined compression schemes is very easy: Simply extend a class from the class `Submatrix` and

[3] The keyword **virtual** is used to dynamically bind the function such that the compiler can invoke the correct function at runtime. The suffix $= 0$ is used to denote that a function is abstract and must be overridden in a conrete subclass.

implement all pure virtual functions. In order to get started directly, we provide two default implementations of the Compressed Row Storage and the Block Sparse Row compression schemes. The corresponding classes are named `CrsSubmatrix` and `BsrSubmatrix`, respectively.

5.4 Distribution

As already mentioned, after dividing the whole sparse matrix into multiple submatrices these are distributed among all collaborating processes (cf. Section 3.4.1.3). Another important feature of our data structure is the ability to user-define this distribution scheme as opposed to implementing a fixed one. This is very important and useful since the distribution scheme determines which process is responsible for storing which submatrix. Conceptually, this can be seen as a load balancing mechanism and comes in handy if the processors of a multiprocessor system exhibit different clock rates such that faster processors can handle more submatrices. The distribution scheme can also be used to assign all submatrices of a row and/or column to a single process. Thus, rotating rows and/or columns can be performed much faster since all elements are locally available to a process such that function calls to MPI routines are not necessary. As the user knows best about the characteristics of her application, it is at best to leave the decision how to distribute the submatrices to her.

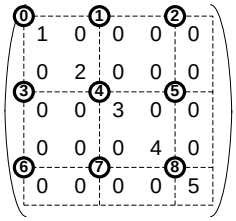
```
1  class Distribution {
2    int n, m, r, c, np;
3
4    void init(int n, int m, int r, int c, int np) {...}
5
6    virtual int getIdProcess(int idSubmatrix) const = 0;
7  };
```

Listing 5: Definition of the class `Distribution`. For the sake of clarity trivial attributes and functions are omitted.

Again, our approach is implemented using polymorphism and inheritance. We provide an interface in terms of an abstract class defined in the file `Distribution.h` (cf. Listing 5). As already mentioned, each submatrix is assigned a unique ID. This ID is now used to assign each submatrix to one of the np processes with $np \in \mathbb{N}$ (cf. Figure 5.1). The mapping between the ID of the submatrix and the ID of the process is performed by the pure virtual function `getIdProcess` (cf. line 4). This function expects the ID of a submatrix and returns the ID of the process responsible for storing the submatrix with the given ID. Obviously, this function must be overridden in any concrete subclass. In order to perform the mapping the function may use the variables `n`, `m`, `r`, `c`, and `np` which are passed using the `init` function (cf. lines 2 and 3). Currently, we provide four default distribution schemes which are implemented in the classes `BlockDistribution`, `ColumnDistribution`, `RoundRobinDistribution`, and `RowDistribution`:

- The class `BlockDistribution` can be used to assign whole blocks of contiguous submatrices to each process. If the number of submatrices ns can be divided by the number of processes np without remainder, each process receives a block of $\frac{ns}{np}$ submatrices. Otherwise, the last processes are assigned one submatrix less than the first processes (cf. Figure 5.1).
- The class `ColumnDistribution` can be used to assign complete columns of submatrices to each process. The assignment is performed in an alternating way, i.e. after each process has been assigned a complete column, the next column is assigned to the first process again etc.

- The class `RoundRobinDistribution` can be used to assign submatrices to processes in an alternating way. A process is responsible for a certain submatrix, if the following formula holds true: $\texttt{idProcess} = \texttt{idSubmatrix} \bmod \texttt{np}$.
- The class `RowDistribution` can be used to assign complete row of submatrices to each process. The assignment is performed in an alternating way, i.e. after each process has been assigned a complete row, the next row is assigned to the first process again etc.



(a)

distribution class	p_0	p_1
`BlockDistribution`	0, 1, 2, 3, 4	5, 6, 7, 8
`ColumnDistribution`	0, 3, 6, 2, 5, 8	1, 4, 7
`RoundRobinDistribution`	0, 2, 4, 6, 8	1, 3, 5, 7
`RowDistribution`	0, 1, 2, 6, 7, 8	3, 4, 5

(b)

Figure 5.1: Distributing nine submatrices among two processes using different classes for the distribution. p_i denotes process i with $i \in \{0, 1\}$.

5.5 Enhanced Skeletons

One of the primary goals of OpenMP is to exploit *data parallelism* which is also known as *loop-level parallelism*. This form of parallelism naturally comes into play when executing loops. The main idea behind parallelizing a loop is to distribute the loop passes among all collaborating threads such that the execution time is roughly divided by the number of threads. Obviously, this is only possible if there do not exist any *data dependencies*, i.e. if the result of executing an arbitrary loop pass does not depend on the result of executing a former loop pass. This section shows how OpenMP directives are used to speed up the execution of certain functions on the example of the *mapInPlace* skeleton (cf. Listing 6). Note that some implemented skeletons are enhanced in this way such that the execution on a multi-core processor will be much faster than compared to a single-core processor.

Recall that the *mapInPlace* skeleton replaces each element of the sparse matrix by the result of applying the given function `f` to it. Thus, the given function `f` must accept and return elements of type `T`. Since the *mapInPlace* skeleton only changes the state of the sparse matrix and does not return a new one, its return type is **`void`** (cf. line 1). The following two lines declare some variables used for the number of locally stored submatrices (`ns`), the number of elements of the current submatrix (`ne`), a pointer to the current submatrix (`submatrix`), and the value of the current element (`value`).

Let us first explain the function without the OpenMP directive by simply ignoring the line starting with **`#pragma`** (cf. line 7). Basically, the skeleton iterates over all locally stored submatrices, reads the value of each element, and overwrites it with the result of applying `f` to it. The iteration over the locally stored submatrices is performed by the loop in line 8. The first thing we do when entering a new loop pass is to store a pointer to the current submatrix since it will be used quite often later on (cf. line 9). Note that `submatrices` is a global variable of type `std::vector<Submatrix<T>*>` which contains all locally stored submatrices. Next, we need to determine the number of elements stored by the current submatrix in order to iterate over them. Since this information is stored by each submatrix, we can access it via the function `getElementCountLocal` (cf. line 10). The following loop is used to iterate over all elements of the current submatrix (cf. line 12). The loop index `j` can be used to access a local element of the current submatrix and store its value in `value` (cf. line 13). Note that by doing so, we bypass the

implemented compression scheme and work directly on the raw values, which is much faster (cf. Section 5.3). If `value` is zero, we do not need to do anything and may simply continue with the next iteration. If `value` is non-zero, we apply the given function `f` to it and store the result by again using the loop index `j` as a local index of the current submatrix (cf. line 16).

```
1  void mapInPlace(T (*f)(T)) {
2    int ne;
3    int ns = getSubmatrixCount();
4    Submatrix* submatrix;
5    T value;
6
7    #pragma omp parallel for private(ne, submatrix, value)
8    for(int i = 0; i < ns; i++) {
9      submatrix = submatrices.at(i);
10     ne = submatrix->getElementCountLocal();
11
12     for(int j = 0; j < ne; j++) {
13       value = submatrix->getElementLocal(j);
14
15       if(value != 0) {
16         submatrix->setElementLocal(f(value), j);
17 } } } }
```

Listing 6: Complete source code of the *mapInPlace* skeleton. Note the OpenMP directive in line 7.

Let us now consider how this function can be improved with an OpenMP directive. First of all, we observe that the execution of the outer loop in line 8 is suitable for parallelization, since there are no data dependencies inside the loop body. This can be achieved by inserting the OpenMP `parallel for` directive[4] just before the loop (cf. line 7). When encountering such a directive, the runtime environment creates a thread team and distributes the loop passes between them. For example, if a team of two threads encounters a loop with ten iterations, each thread is assigned five iterations: The first thread executes the iterations with indexes zero to four, the second thread executes the iterations with indexes five to nine.[5] The number of threads created can be user-defined via the optional clause `num_threads`.[6] If omitted, the runtime environment creates one thread for each available processor. The directive is used in conjunction with the optional **`private`** clause which is absolutely vital here. In order to understand its purpose, we need to recapitulate that OpenMP is an API developed for shared memory programming. In such an environment, all threads have access to so-called *shared variables*. Per default, all variables used inside a `parallel` region are shared, i.e. there is only a single variable in memory which is accessed by all threads. Without the **`private`** clause, the variables declared in lines 2–5 would be shared variables which might cause strange and irreproducible behaviour. Imagine the following scenario: Thread t_1 enters the outer loop, stores the number of locally available elements of the current submatrix, and prepares to execute the inner loop (cf. lines 10–12). Now imagine a second thread t_2 which overwrites the value of `ne` before t_1 enters the inner loop. Since `ne` is a shared variable, all threads read from and write to the same memory location, such that t_1 might read a different value when initializing the inner loop than it previously wrote to memory. Obviously, this is not the intended behaviour. In order to correctly parallelize the loop, each thread must have its own, i.e. private copy of each variable used inside the loop body. This behaviour can be achieved by using the **`private`** clause which expects a comma separated list of variables such that each thread has a private copy of the variables declared inside the list.

[4] In general, all OpenMP pragmas have the form **#pragma** omp <directive> [clauses] where each clause may have a comma separated list of arguments.

[5] This schedule is called *static* and is the default behaviour. Other schedules are *dynamic*, *guided*, and *runtime*.

[6] The number of threads created by a parallel region can also be controlled by means of the OpenMP function omp_set_num_threads. If set to 1, OpenMP is effectively disabled.

One question remains unanswered: Why don't we also parallelize the inner loop in line 12? This is due to the fact that creating and destroying a thread team is not totally for free. If we would also try to parallelize the inner loop, every time a thread belonging to the thread team of the outer loop encounters a nested `for` directive, it would create a new thread team and destroy it after executing the loop. Summing up the additional overhead, it is very likely that this approach is even slower than the sequential one. Anyway, if the outer loop already created a single thread for each available processor, there are no more processors left to execute threads in parallel. In general, parallelizing nested `for`-loops only pays off if the outer loop does not use all available processors. If this is the case, parallelizing inner loops can speed up a program, although it will introduce additional overhead by repeatedly creating and destroying thread teams.

Chapter 6

Case Studies

In the following sections, we give two short examples, which demonstrate how Muesli can be used for parallel programming.

6.1 Combining Task and Data Parallelism

The following example is intended to show the main features of our skeleton library and how to use them. We are going to cover currying, partial applications, skeleton topologies, and how to combine task and data parallel skeletons. The example is intended to run on $np = 6$ processes and uses the task parallel skeleton *Pipe* in combination with the atomic building blocks *Initial*, *Atomic*, and *Final* as well as our distributed data structure for general sparse matrices.

The implemented skeleton topology is depicted in Figure 6.1. The *Initial* process creates ten numbers ranging from 0 to 9 and sends each of them to the *Atomic* process. This process uses each number received from the *Initial* process to create a distributed sparse matrix. Since the *Atomic* process internally consists of four processes, it is used to perform some data parallel computations. Afterwards, the result of these computations is sent to the *Final* process which simply outputs it on the console. At this point it is clear why we use 6 processes: One process is used by each the *Initial* and the *Final* process and four processes are used by the *Atomic* process.

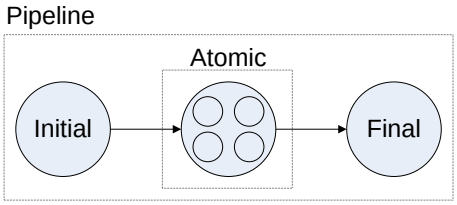


Figure 6.1: Skeleton topology used for our example.

Let us now have a look at the corresponding source code (cf. Listing 7). Prior to using our skeleton library, the user first has to include the header file `Muesli.h` (cf. line 1). By doing so, all task and data parallel skeletons described in the previous sections can be used without including any further header files. The `main` function of our program can be found in lines 35–45. First of all, we must call the function `InitSkeletons` before we can instantiate any skeletons (cf. line 37). This is of outmost importance, since, as the name suggests, the function performs some crucial tasks such as initializing the MPI environment, determining the number of processes and

its ID, respectively. Now, we are able to create our skeletons (cf. lines 39–42). First of all, we create the `Initial` process. As explained in Section 4.1.1, the process must be parameterized with the problem type it is going to create. Moreover, the process must be passed a pointer to a function which creates the problems. Since each of our problems consists of a single integer, we parameterize the process with **`int`**. The function used for creating the problems is defined in lines 9–17 and is expected to return a pointer to the problem it just created. Note that the return type of the `createProblem` function must be a pointer to the type the `Initial` process has been parameterized with. This is no coincidence, but in fact mandatory. In order to indicate that all problems have been created, `createProblem` may simply return `NULL` (cf. line 16).

Next, we create the `Atomic` process (cf. line 40). As explained in Section 4.1.3, the process must be parameterized with the problem type it is going to receive and with the solution type it is going to produce. Both parameters are set to **`int`**, since we expect to receive integer values from the `Initial` process and send integer values to the `Final` process. Again, it is important that you provide the process with the correct types, since otherwise your program will not compile. Besides parameterizing the process, we need to pass two arguments to it: A pointer to a function which is executed to transform a problem into a solution and the number of processes which are used internally. As already mentioned above, the `Atomic` process will use four internal processes in order to execute some data parallel skeletons. These finally come into play in the function `duplicate` which is passed as an argument to the `Atomic` process. The function is defined in lines 19–29 and, according to the parameterization of the process, is expected to return a solution of type **`int`**`*` for each problem of type **`int`**`*` it receives. Upon receiving a problem, the function creates a new distributed sparse matrix `A` of size 4×4 and divides it into submatrices of size 2×2 (cf. line 21). Thus, the whole sparse matrix is split into four submatrices such that each of the four `Atomic` processes stores one submatrix. Next, the elements in the upper left and the lower right corner of the sparse matrix are set to 1 (cf. lines 23 and 24). Then, we apply the data parallel skeleton *mapInPlace* to the sparse matrix (cf. line 25 and Section 3.4.3). The skeleton expects a pointer to a function and will replace each element of the sparse matrix by applying the given function to it. However, we are passing a partial application (cf. Section 2.3) rather than an ordinary function pointer. Note that the destinction between a partial application and a function pointer is irrelevant to you, since each function expecting a function pointer is overloaded such that it can also be passed a partial application. Partial applications are created by calling the `curry` function. This function expects a function pointer as the sole argument and transforms the given function into a partial application of type `FctX` where `X` is a placeholder for the number of arguments the given function expects. Thus, calling `curry(multiply)` returns an anonymous partial application of type `Fct2`. However, this partial application, let's call it f, cannot be used by the *mapInPlace* skeleton, since the skeleton expects a function of type `Fct1`, whereas f is of type `Fct2`. In order to overcome this incompatibility, we need to pass an **`int`** argument to f, thus transforming f into a partial application of type `Fct1`. This is done by applying the argument `*problem` to f. The resulting partial application, let's call it g, is created by replacing the first argument of `multiply` by `*problem`. The function `multiply` is defined in line 7 and simply returns the product of the given arguments `a` and `b`. Thus, g returns the product of `*problem` and the argument `b`, since `a` is replaced by `*problem`. By doing so, `A.mapInPlace(curry(multiply)(*problem))` replaces each element of the sparse matrix by the result of multiplying the element with the problem received. In order to verify the correctness of our implementation, we create the solution returned by `duplicate` by means of the skeleton `fold` which reduces all values of the sparse matrix into a single one by repeatedly applying the given function to all elements (cf. line 26 and Section 3.4.3). The `add` function used with the skeleton is defined in line 6 and simply returns the sum of the given values `a` and `b`. By doing so, the `duplicate` function doubles the value of the given argument.

Prior to creating the `Pipe` skeleton, we need to create the `Final` process which is responsible for receiving the solutions created by the `Atomic` process (cf. line 41). As explained in Section 4.1.2, the process must be parameterized with the solution type it is going to receive. The parameter is set to **`int`**, since we expect to receive integer values from the `Atomic` process. Again, it is important that you provide the process with the correct types, since otherwise your program will not compile. As an argument, the process expects a pointer to a function which is

executed upon receiving a solution. This function is defined in lines 31–33 and simply prints the solution received to standard output.

Finally, we create the `Pipe` skeleton (cf. Section 4.2.3) and pass the formerly created process `initial`, `atomic`, and `final` as arguments (cf. line 42). The whole computation is started by calling the function `start` defined by the `Pipe` skeleton. This should output the numbers 0, 2, 4, 6, 8, 10, 12, 14, 16, and 18. To shut down our library, you should call the function `TerminateSkeletons` (cf. line 46). This will finalize the MPI environment and free any previously allocated resources.

```
1  #include "Muesli.h"
2
3  static int numberOfProblems = 10;
4  static int currentProblem   =  0;
5
6  int add(int a, int b)      { return a + b; }
7  int multiply(int a, int b) { return a * b; }
8
9  int* createProblem(Empty dummy) {
10   if(currentProblem < numberOfProblems) {
11     int* problem = new int;
12     *problem = currentProblem++;
13     return problem;
14   }
15
16   return NULL;
17 }
18
19 int* duplicate(int* problem) {
20   int* solution = new int;
21   DistributedSparseMatrix<int> A(4, 4, 2, 2);
22
23   A.setElement(0, 0, 1);
24   A.setElement(3, 3, 1);
25   A.mapInPlace(curry(multiply)(*problem));
26   *solution = A.fold(add);
27
28   return solution;
29 }
30
31 void receiveSolution(int* solution) {
32   std::cout << "solution: " << *solution << std::endl;
33 }
34
35 int main(int argc, char* argv[]) {
36   try {
37     InitSkeletons(argc, argv, MSL_SERIALIZED);
38
39     Initial<int>    initial(createProblem);
40     Atomic<int,int> atomic(duplicate, 4);
41     Final<int>      final(receiveSolution);
42     Pipe            pipe(initial, atomic, final);
43
44     pipe.start();
45
46     TerminateSkeletons();
47   } catch(Exception&) {
48     std::cout << "exception" << std::endl << std::flush;
49 } }
```

Listing 7: Example demonstrating the main features of Muesli.

6.2 Mergesort

An important issue in the context of divide and conquer algorithms is object serialization due to the fact that the subproblems generated by divide are typically smaller w.r.t. the size of their representation than the problem that has been divided. The following example demonstrates the implementation of MSL_Serializable and the application of the StreamDC skeleton at the example of the well-known Mergesort algorithm. The implemented skeleton topology is depicted in Figure 4.2. The example is intended to run on $np = 7$ processes and generates an *Initial*, a *StreamDC*, and a *Final* skeleton, which are combined with a *Pipe*. The `Initial` process generates divide and conquer problems and routes them to the `StreamDC` skeleton, which processes them in parallel. The StreamDC skeleton internally consists of five solvers, from which three are master solvers. That is, the StreamDC skeleton is able to process three problems in parallel at the same time. The solutions are then routed to the `Final` process.

Listing 8 shows the corresponding source code. First of all, the user has to include the header file `Muesli.h` (cf. line 1). By doing so, all task and data parallel skeletons described in the previous sections can be used without including any further header files. An unsorted integer array is represented as an instance of the class `Problem`, which stores the array of integer values and the size of the array (cf. lines 7–36). Moreover, it is derived from `MSL_Serializable` and implements the methods `getSize`, `reduce`, and `expand`. `getSize` returns the size of the buffer which is required to store the serialized object. In our case, we have to store the attribute `size` and all values of the array. Thus, `getSize` returns $sizeof(int) + sizeof(int) * size$. A contiguous memory block corresponding to this size is passed to the method `reduce`, which copies the attribute `size` and all values of the array to the buffer. In contrast, `expand` extracts this data in the same sequence in which the data was written to the buffer. The first value read from the buffer is the attribute `size`. This value is used to generate an integer array, which is than filled with the remaining values stored in the buffer.

The `main` function of our program can be found in lines 128–144. First of all, the function `InitSkeletons` must be applied in order to initialize *Muesli*. Then, the process topology is created using C++ constructors. The `Initial` process is provided with a user defined function `init`. The *Initial* process internally calls the *init* method, which creates five randomly generated integer arrays of the size $N = 2^{20} = 1048576$. These arrays are internally passed to the Initial process and then routed to the StreamDC skeleton. In order to indicate that all problems have been created, `init` may simply return `NULL` (cf. line 54). The StreamDC skeleton is provided with four user defined functions `divide`, `combine`, `solve`, and `isSimple`. The `isSimple` operator is applied to an unsorted array in order to test whether it is simple enough to sort it directly by `solve`. In our implementation, the test is based on the size of the array which is compared to a threshold $T = 1024$. If the array size is less or equal than T, the `solve` operator is applied and the array is sorted sequentially (cf. lines 83–88). Otherwise, the array is divided into two half-size subarrays by calling `divide` (cf. lines 90–104). Thus, each of the initial problems is divided into a total of 1024 subproblems. The `combine` operator (cf. lines 106–122) takes an array of two sorted subarrays selected by the StreamDC skeleton and merges them together. A `Final` process is responsible for receiving the solutions created by the `StreamDC` skeleton.

Finally, we create the `Pipe` skeleton (cf. Section 4.2.3) and pass the formerly created process `initial`, `dc`, and `final` as arguments (cf. line 136). The whole computation is started by calling the function `start` defined by the `Pipe` skeleton. To shut down *Mueslie* the function `TerminateSkeletons` (cf. line 141) must be applied. This will finalize the MPI environment and free any previously allocated resources.

```
#include "Muesli.h"

static int D = 2;               // degree of the divide and conquer tree
static int N = 1048576;         // problem size
static int THRESHOLD = 1024;    // arrays of this size are simple
static int current = 0;
static int numOfProblems = 5;

class Problem: public MSL_Serializable {
public:
  int size;
  int* array;

  Problem() {
  }

  virtual ~Problem() {
    delete[] array;
  }

  inline int getSize() {
    return sizeof(int) + sizeof(int)*size;
  }

  void reduce(void* pBuffer, int bufferSize) {
    int* adr1 = (int*) memcpy(pBuffer,&(this->size),sizeof(int));
    adr1++;
    int len = (this->size)*sizeof(int);
    int* adr2 = (int*) memcpy(adr1,this->array,len);
  }

  void expand(void* pBuffer, int bufferSize) {
    int* adr = (int*) pBuffer;
    int size = *adr;
    this->size = size;
    this->array = new int[size];
    adr++;
    for (int i=0; i<size; i++) {
      this->array[i] = *adr;
      adr++;
    }
  }
}; // EOC Problem

Problem* init(Empty dummy) {
  if (current < numOfProblems) {
    Problem* prob = new Problem();
    prob->size = N;
    prob->array = new int[N];
    for (int i=0; i<N; i++) {
      prob->array[i] = rand()\%N;
    }
    current++;
    return prob;
  }
  else {
    return NULL;
  }
}

```

```
61 bool isSimple(Problem* p) {
62   return (p->size <= THRESHOLD);
63 }
64
65 void merge(int* a, int left, int mid, int right) {
66   int size = right-left;
67   int* temp = new int[size];
68   int i=0, j=0, k=0;
69   while ((left+i<mid) && (mid+j<right)) {
70     if(a[left+i]<a[mid+j]) {
71       temp[k++] = a[left+(i++)];
72     }
73     else {
74       temp[k++] = a[mid+(j++)];
75     }
76   }
77   while (left+i<mid) {
78     temp[k++] = a[left+(i++)];
79   }
80   while (mid+j<right) {
81     temp[k++] = a[mid+(j++)];
82   }
83   for (i=0; i<size; i++) {
84     a[left+i] = temp[i];
85   }
86   delete[] temp;
87 }
88
89 void mergesort(int* a, int left, int right) {
90   if (right-left>1) {
91     int mid = (left+right)/2;
92     mergesort(a, left, mid);
93     mergesort(a, mid, right);
94     merge(a, left, mid, right);
95   }
96 }
97
98 Problem* solve(Problem* p) {
99   int* a = p->array;
100  int size = p->size;
101  mergesort(p->array,0,p->size);
102  return p;
103 }
104
105 Problem** divide(Problem* p) {
106  int pSize = p->size;
107  Problem** result = new Problem*[2];
108  result[0] = new Problem();
109  result[0]->size = pSize/2;
110  result[0]->array = new int[pSize/2];
111  for (int i=0; i<pSize/2; i++) {
112    result[0]->array[i] = p->array[i];
113  }
114  result[1] = new Problem();
115  result[1]->size = (pSize+1)/2;
116  result[1]->array = new int[(pSize+1)/2];
117  for (int i=(pSize+1)/2; i<pSize; i++) {
118    result[1]->array[i-(pSize+1)/2] = p->array[i];
119  }
120  return result;
121 }
```

```
122 Problem* combine(Problem** p) {
123   int sizeP0 = p[0]->size, sizeP1 = p[1]->size, size = sizeP0 + sizeP1;
124   Problem* result = new Problem();
125   result->size = size;
126   result->array = new int[size];
127   int i=0, j=0, k=0;
128   while (i < sizeP0 && j < sizeP1) {
129     if (p[0]->array[i] <= p[1]->array[j]) {
130       result->array[k++] = p[0]->array[i++];
131     }
132     else {
133       result->array[k++] = p[1]->array[j++];
134     }
135   }
136   if (i == sizeP0) {
137     while (j < sizeP1) {
138       result->array[k++] = p[1]->array[j++];
139     }
140   }
141   else {
142     while (i < sizeP0) {
143       result->array[k++] = p[0]->array[i++];
144     }
145   }
146   return result;
147 }
148
149 void fin(Problem prob) {
150   // do something with the sorted array ...
151 }
152
153 int main(int argc, char* argv[]) {
154   try {
155     InitSkeletons(argc,argv);
156
157     // create the process topology
158     Initial<Problem>          initial(init);
159     StreamDC<Problem,Problem> dc(divide,combine,solve,isSimple,D,5,3);
160     Final<Problem>            final(fin);
161     Pipe                      pipe(initial,dc,final);
162
163     // start the process topology
164     pipe.start();
165
166     TerminateSkeletons();
167   }
168   catch(Exception&) {
169     std::cout << "Exception" << std::endl << std::flush;
170   }
171 }
```

Listing 8: A task parallel implementation of Mergesort.

References

[1] MPI. Message Passing Interface Forum. `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`, 2008.

[2] E. Alba and F. Almeida. MaLLBa: A Library of Skeletons for Combinatorial Search. In *Proceedings of the Euro-Par '02*, volume 2400 of *LNCS*, pages 927–932. Springer, 2002.

[3] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[4] J. Darlington, Y. Guo, H.To, and J. Yang. Parallel Skeletons for Structured Composition. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 19–28. ACM Press, 1995.

[5] H. Kuchen and M. Cole. The Integration of Task and Data Parallel Skeletons. In *Parallel Processing Letters*, volume 12(2), pages 141–155, 2002.

[6] H. Kuchen. A Skeleton Library. In *Euro-Par '02*, volume 2400 of *LNCS*, pages 620–629. Springer, 2002.

[7] K. Matsuzaki, K. Emoto, H. Iwasaki, and Z. Hu. A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In *Proceedings of 1st international Conference on Scalable Information Systems (INFOSCALE)*, 2006.

[8] S. Pelagatti. Task and Data Parallelism in P3L. In F.A. Rabhi and S. Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing*, pages 155–186. Springer, 2003.

[9] G. H. Botorog and H. Kuchen. Efficient Parallel Programming with Algorithmic Skeletons. In *Proceedings of the Euro-Par '96*, volume 1123 of *LNCS*, pages 718–731. Springer, 1996.

[10] G. H. Botorog and H. Kuchen. Efficient High-Level Parallel Programming. In *Theoretical Computer Science*, volume 196, pages 71–107, 1998.

[11] J. Darlington, A.J. Field, and P.G. Harrison. Parallel Programming Using Skeleton Functions. In *Proceedings of Parallel Architectures and Languages Europe (PARLE)*, volume 694 of *LNCS*. Springer, 1993.

[12] J. Darlington, Y. Guo, and H.W. To. Functional Skeletons for Parallel Coordination. In *Proceedings of Euro-Par'95*, volume 966 of *LNCS 966*. Springer, 1995.

[13] H. Kuchen, R. Plasmeijer, and H. Stoltze. Efficient Distributed Memory Implementation of a Data Parallel Functional Language. In *Proceedings of the PARLE '94*, volume 817 of *LNCS*, pages 466–475. Springer, 1994.

[14] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. In *Parallel Computing*, volume 30(3), pages 389–406, 2004.

[15] D. Skillicorn. *Foundations of Parallel Programming*. Cambridge U. Press, 1994.

[16] H. Bischof, S. Gorlatch, and R. Leshchinskiy. DatTel: A Data-parallel C++ Template Library. In *Parallel Processing Letters*, volume 13(3), pages 461–472, 2003.

[17] H. Bischof. *Systematic Development of Parallel Programs Using Skeletons*. PhD thesis, Westfälische Wilhelms-Universität Münster, 2005.

[18] I. Foster, R. Olson, and S.Tuecke. Productive Parallel Programming: The PCN Approach. In *Scientific Programming*, volume 1(1), pages 51–66, 1992.

[19] A. Benoit, M. Cole, J. Hillston, and S. Gilmore. Flexible Programming with eSkel. In *Proceedings of the Euro-Par '05*, volume 3648 of *LNCS*, pages 761–770. Springer, 2005.

[20] M. Cole. The Skeletal Parallelism Web Page. `http://homepages.inf.ed.ac.uk/mic/Skeletons/`, 2008.

[21] M. Danelutto, R.D. Meglio, S. Orlando, and S. Pelagatti. A Methodology for the Development and the Support of Massively Parallel Programms. In *Future Generation Computer Systems*, volume 8, pages 205–220. Elsevier, 1992.

[22] B. Bacci, M. Danelutto, S. Orlando, and S. Pelagatti. P3L: A Structured High Level Programming Language and its Structured Support. In *Concurrency: Practice and Experience*, volume 7(3), pages 225–255. John Wiley & Sons, 1995.

[23] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for Data Parallelism in P3L. In *Proceedings of the Third International Euro-Par Conference on Parallel Processing*, volume 1300 of *LNCS*, pages 619–628. Springer, 1997.

[24] M. Aldinucci, M. Danelutto, and P. Teti. An Advanced Environment Supporting Structured Parallel Programming in Java. In *Future Generation Computer Systems*, volume 19, pages 611–626. Elsevier, 2003.

[25] M. Danelutto and P. Teti. Lithium: A Structured Parallel Programming Environment in Java. In *Proceedings of Computational Science (ICCS)*, volume 2330 of *LNCS*, pages 844–853. Springer, 2002.

[26] M. Aldinucci, M. Danelutto, and P. Dazzi. Muskel: A Skeleton Library Supporting Skeleton Set Expandability. In *Scalable Computing: Practice and Experience*, volume 8(4), pages 325–341. SWPS, 2007.

[27] K. Matsuzaki, K. Kakehi, and H. Iwasaki. A Fusion-Embedded Skeleton Library. In *Proceedings of the 10th International Euro-Par Conference on Parallel Processing*, volume 3149 of *LNCS*, pages 644–653. Springer, 2004.

[28] Y. Karasawa and H. Iwasaki. Parallel Skeletons for Sparse Matrices in SkeTo Skeleton Library. *Information Processing Society of Japan (IPSJ)*, 4:167–181, 2008.

[29] M. Alt, J. Dünnweber, J. Müller, and S. Gorlatch. HOCs: Higher-order Components for Grids. In *Component Models and Systems for Grid Applications*. Springer, 2004.

[30] M. Alt. *Using Algorithmic Skeletons for Efficient Grid Computing with Predictable Performance*. PhD thesis, Westfälische Wilhelms-Universität Münster, 2007.

[31] The Globus Alliance Web Pages. `http://www.globus.org/`, 2008.

[32] M. Cole. The Skeletal Parallelism Web Page. `http://homepages.inf.ed.ac.uk/mic/Skeletons/`, 2008.

[33] H. Kuchen and J. Striegnitz. Higher-Order Functions and Partial Applications for a C++ Skeleton Library. In *Joint ACM Java Grande & ISCOPE Conference*, volume 3648, pages 122–130. John Wiley & Sons, 2002.

[34] M. Poldner and H. Kuchen. Scalable Farms. In *Proceedings of Parallel Computing (ParCo)*, 2005.

[35] M. Poldner and H. Kuchen. Algorithmic Skeletons for Branch and Bound. In *Proceedings of the International Conference on Software and Data Technology (ICSOFT)*, pages 204–219. Springer, 2008.

[36] M. Poldner and H. Kuchen. On Implementing the Farm Skeleton. In *Parallel Processing Letters*, volume 18(1), pages 204–219, 2008.

[37] M. Poldner and H. Kuchen. Skeletons for Divide and Conquer Algorithms. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*. ACTA Press, 2008.

[38] M. Poldner and H. Kuchen. Optimizing Skeletal Stream Processing for Divide and Conquer. In *Proceedings of the 3rd International Conference on Software and Data Technologies (ICSOFT)*, pages 181–189. INSTICC Press, 2008.

[39] M. Poldner and H. Kuchen. Task Parallel Skeletons for Divide and Conquer. In *Proceedings of the 25.Workshop of the Working Group Programming Languages and Computing Concepts of the German Computer Science Association GI, Bad Honnef*, 2008.

[40] P. Ciechanowicz, S. Dlugosz, H. Kuchen, and U. Müller-Funk. Exploiting Training Example Parallelism With a Batch Variant of the ART 2 Classification Algorithm. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, pages 195–201. ACTA Press, 2008.

[41] H. Kuchen, M. Poldner, and P.Ciechanowicz. The Skeleton Library Web Pages. `http://www.wi.uni-muenster.de/PI/forschung/Skeletons/index.php`, 2008.

[42] D. Vandevoorde and N. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2003.

[43] D. Musser and A. Saini. *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison Wesley, 1995.

[44] R. Chandra. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[45] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, pages 46–55, 1998.

[46] H. Kuchen. Optimizing Sequences of Skeleton Calls. In *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 254–273. Springer, 2004.

[47] J. Dongarra. *LINPACK User's Guide*. Society for Industrial and Applied Mathematics, 1987.

[48] E. Angerson, Z. Bai, and J. Dongarra. LAPACK: A Portable Linear Algebra Library for High-Performance Computers. In *Supercomputing*, pages 2–11, 1990.

[49] J. Choi, J. Dongarra, and R. Pozo. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memoryconcurrent Computers. *Frontiers of Massively Parallel Computation*, pages 120–127, 1992.

[50] Y. Saad. SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, 1990.

[51] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. MIT Press, 1999.

[52] R. Boisvert, R. Pozo, and K. Remington. Matrix Market: A Web Resource for Test Matrix Collections. *The Quality of Numerical Software: Assessment and Enhancement*, pages 125–137, 1997.

[53] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. Wiley & Sons, 1999.

[54] J.D.C. Little, K.G. Murty, D.W. Sweeny, and C. Karel. An Algorithm for the Traveling Salesman Problem. In *Operations Research*, volume 11, pages 972–989, 1963.

[55] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. volume 145(2), pages 293–294. Doklady Akademii Nauk SSSR, 1962.

[56] R. Maor and M. Brand. The xparam home page. `http://xparam.sourceforge.net/`, 2008.

[57] Boost C++ Libraries. `http://www.boost.org/`, 2008.

[58] R. Ramey. Boost Serialization Library. `http://rrsd.com/boost/`, 2008.

[59] The GenSerial Library. `http://genserial.sourceforge.net/`, 2008.

[60] A. Alexandrescu. *Modernes C++ Design*. mitp, 2003.

Working Papers, ERCIS

Nr. 1 Becker, J.; Backhaus, K.; Grob, H. L.; Hoeren, T.; Klein, S.; Kuchen, H.; Müller-Funk, U.; Thonemann, U. W.; Vossen, G.; European Research Center for Information Systems (ERCIS). Gründungsveranstaltung Münster, 12. Oktober 2004. October 2004.

Nr. 2 Teubner, R. A.: The IT21 Checkup for IT Fitness: Experiences and Empirical Evidence from 4 Years of Evalutation Practice. March 2005.

Nr. 3 Teubner, R. A.; Mocker, M.: Strategic Information Planning – Insights from an Action Research Project in the Financial Services Industry. June 2005.

Nr. 4 Gottfried Vossen, Stephan Hagemann: From Version 1.0 to Version 2.0: A Brief History Of the Web. January 2007.

Nr. 5 Hagemann, S.; Letz, C.; Vossen, G.: Web Service Discovery – Reality Check 2.0. July 2007.

Nr. 6 Hoeren, T.; Vossen, G.: The Role of Law in an Electronic World Dominated by Web 2.0. 2008.

ERCIS – European Research Center for Information Systems
Universität Münster
Leonardo-Campus 3 ▪ 48149 Münster ▪ Germany
Tel: +49 (251) 83-38100 ▪ Fax: +49 (251) 83-38109
info@ercis.org ▪ http://www.ercis.org/

ISSN 1614-7448