A Skeleton Library

Herbert Kuchen

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

Abstract. Today, parallel programming is dominated by message passing libraries such as MPI. Algorithmic skeletons intend to simplify parallel programming by increasing the expressive power. The idea is to offer typical parallel programming patterns as polymorphic higher-order functions which are efficiently implemented in parallel. The approach presented here integrates the main features of existing skeleton systems. Moreover, it does not come along with a new programming language or language extension, which parallel programmers may hesitate to learn, but it is offered in form of a library, which can easily be used by e.g. C and C++ programmers. A major technical difficulty is to simulate the main requirements for a skeleton implementation, namely higher-order functions, partial applications, and polymorphism as efficiently as possible in an imperative programming language. Experimental results based on a draft implementation of the suggested skeleton library show that this can be achieved without a significant performance penalty.

1 Introduction

Today, parallel programming of MIMD machines with distributed memory is typically based on message passing. Owing to the availability of standard message passing libraries such as MPI [1] [GLS99], the resulting software is platform independent and efficient. Typically, the SPMD (single program multiple data) style is applied, where all processors run the same code on different data. Conceptually, the programmer often has one or more distributed data structures in mind, which are manipulated in parallel. Unfortunately, the mentioned message passing approach does not support this view of the computation. The programmer rather has to split the conceptually global data structure into pieces, such that every processor receives one (or more) of them and cares about all computations which correspond to the locally available share of data. In the syntax of the final program, there is no indication that all these pieces belong together. The combined distributed data structure only exists in the programmer's mind. Thus, the programming level is much lower than the conceptual view of the programmer. This causes several disadvantages. First, the programmer often has to fight against low-level communication problems such as deadlocks and starvation which could be substantially reduced and often eliminated by using a more

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

expressive approach. Moreover, the local view of the computation makes global optimizations very difficult. One reason is that such optimizations require a cost model of the computation which is hard to provide for general message passing based computations.

Many approaches try to increase the level of parallel programming and to overcome the mentioned disadvantages. Few of them could gain significant acceptance by parallel programmers. It is impossible to mention all high-level approaches to parallel programming here. Let us just focus on a few particularly interesting ones.

Bulk synchronous parallel processing (BSP) [SHM97] is a restrictive model where a computation consists of a sequence of *supersteps*, i.e. independent parallel computations followed by a global communication and a barrier synchronization. BSP has been successfully applied to several data-parallel application problems, but owing to its restrictive model it cannot easily be used for irregularly structured problems.

An even higher programming level than BSP is provided by *algorithmic skeletons*, i.e. typical parallel programming patterns which are efficiently implemented on the available parallel machine and usually offered to the user as higher-order functions, which get the details of the specific application problem as argument functions. Thus, a parallel computation consists of a sequence of calls to such skeletons, possibly interleaved by some local computations. The computation is now seen from a global perspective. Several implementations of algorithmic skeletons are available. They differ in the kind of host language used and in the particular set of skeletons offered. Since higher-order functions are taken from functional languages, many approaches use such a language as host language [Da93,KPS94,Sk94]. In order to increase the efficiency, imperative languages such as C and C++ have been extended by skeletons, too [BK96,BK98,DPP97,FOT92].

Depending on the kind of parallelism used, skeletons can be classified into *task parallel* and *data parallel* ones. In the first case, a skeleton (dynamically) creates a system of communicating processes. Some examples are `pipe`, `farm` and `divide&conquer` [DPP97,Co89,Da93]. In the second case, a skeleton works on a distributed data structure, performing the same operations on some or all elements of this structure. Data parallel skeletons, such as `map`, `fold` or `rotate` are used in [BK96,BK98,Da93,Da95,DPP97,KPS94].

Although skeletons have many advantages, they are rarely used to solve practical application problems. One of the reasons is that there is not a common system of skeletons. Each research group has its own approach. The present paper is the result of a lively discussion within the skeleton community on a *standard set of skeletons*. By agreeing on some common set of skeletons, the acceptance of skeletons shall be increased. Moreover, this will facilitate the exchange of tools such as cost analyzers, optimizers, debuggers and so on, and it will boost the development of new tools.

The approach described in the sequel incorporates the main concepts suggested in the discussion and found in existing skeleton implementations. In particular, it provides task as well as data parallel skeletons, which can be combined based on the *two-tier model* taken from P^{3}L [DPP97]. In general, a computation consists of nested task parallel constructs where an atomic task parallel computation maybe sequential or data parallel. Purely data parallel and purely task parallel computations are special cases of this model.

Apart from the lack of standardization, another reason for the missing acceptance of algorithmic skeletons is the fact that they typically are provided in form of a new programming language. However, parallel programmers typically know and use Fortran, C, or C++, and they hesitate to learn new languages in order to try skeletons. Thus, an important aspect of the presented approach is that skeletons are provided in form of a library. Language bindings for the mentioned, frequently used languages will be provided. The C++ binding is particularly elegant, and the present paper will focus on this binding. The reason is that the three important features needed for skeletons, namely higher-order functions (i.e. functions having functions as arguments), partial applications (i.e. the possibility to apply a function to less arguments than it needs and to supply the missing arguments later), and polymorphism, can be implemented elegantly and efficiently in C++ using operator overloading and templates, respectively [St00]. Thus, the C++ binding does not cause the skeleton library to have a significant disadvantage compared to a corresponding language extension. For a C binding, the type system needs to be bypassed using questionable features like void pointers in order to simulate polymorphism (just as the C binding of MPI).The price is a loss of type safety.

The skeleton library can be implemented in various ways. The implementation considered in the present paper is based on MPI and inherits hence its platform independence.

This paper is organized as follows. In Section 2, we present the main concepts of the skeleton library. Section 3 contains experimental results. In Section 4 we conclude.

2 The Skeleton Library

2.1 Data Parallel Skeletons

Data parallelism is based on a *distributed data structure* (or several of them). This data structure is manipulated by operations (like `map` and `fold`, explained below) which process it as a whole and which happen to be implemented in parallel internally. These operations can be interleaved with sequential computations working on non-distributed data. In fact, the programmer views the computation as a sequence of parallel operations. Conceptually, this is almost as easy as sequential programming. Communication problems like deadlocks and starvation cannot occur. Currently, two distributed data structures are offered by the library, namely:

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

where `E` is the type of the elements of the distributed data structure. Other distributed data structures such as distributed lists may be added in the future. By instantiating the template parameter `E`, arbitrary element types can be generated. This shows one of the major features of distributed data structures and their operations. They are *polymorphic.* A distributed data structure is split into several partitions, each of which is assigned to one processor participating in the data parallel computation. Currently, only block partitioning is supported. Other schemes like cyclic partitioning may be added later.

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

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

`A.mapIndexInPlace(g)` applies a binary function `g` to each index position i and the corresponding array element $\mathtt{A}_i$ of a distributed array `A` and replaces $\mathtt{A}_i$ by $\mathtt{g}(i,\mathtt{A}_i)$. `A.fold(h)` combines all the elements of `A` successively by an associative binary function `h`. E.g. `A.fold(plus)` computes the sum of all elements of `A` (provided that `E plus(E,E)` adds two elements). The full list of computation skeletons including other variants of `map` and `fold` as well as different versions of `zip` and `scan` (parallel prefix) can be found in [Ku02a,Ku02b].

Communication consists of the exchange of the partitions of a distributed data structure between all processors participating in the data parallel computation. In order to avoid inefficiency, there is no implicit communication e.g. by accessing elements of remote partitions like in HPF [HPF93] or Pooma [Ka98]. Since there are no individual messages but only coordinated exchanges of partitions, deadlocks and starvation cannot occur. The most frequently used communication skeleton is

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

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

Moreover, there are operations which allow to access attributes of the local partition of a distributed data structure, e.g. `get`, `getFirstCol`, and `getFirstRow` (see Fig. 1) fetch an element of the local partition and the index of the first locally available row and column, respectively. These operations are no skeletons but frequently used when implementing an argument function of a skeleton.

At first, skeletons like `fold` and `scan` might seem equivalent to the corresponding MPI collective operations `MPI_Reduce` and `MPI_Scan`. However, they are more powerful due to the fact that the argument functions of all skeletons can be *partial applications* rather than just C++ functions. A skeleton essentially defines some parallel algorithmic structure, where the details can be fixed

```
 1 inline int negate(const int a){return -a;}
 2
 3 template <class C, int n>
 4 C sprod(const DistributedMatrix<C,n,n>& A,
 5          const DistributedMatrix<C,n,n>& B, int i, int j, C Cij){
 6   C sum = Cij;
 7   for (int k=0; k<A.getLocalRows(); k++)
 8     sum += A.get(i,k+A.getFirstCol()) * B.get(k+B.getFirstRow(),j);
 9   return sum;}
10
11 template <class C, int n>
12 DistributedMatrix<C,n,n> matmult(DistributedMatrix<C,n,n> A,
13                                  DistributedMatrix<C,n,n> B){
14   A.rotateRows(negate);
15   B.rotateCols(negate);
16   DistributedMatrix<C,n,n> R(0,A.getBlocksInCol(),A.getBlocksInRow());
17   for (int i=0; i< A.getBlocksInRow(); i++){
18     R.mapIndexInPlace(curry(sprod<C,n>)(A)(B));
19     A.rotateRows(-1);
20     B.rotateCols(-1);}
21   return R;}
```

Fig. 1. Gentleman's algorithm with skeletons.

by appropriate argument functions. With partial applications as argument functions, these details can depend themselves on parameters, which are computed at runtime. Consider the code fragment in Fig. 1 taken from [Ku02b]. It is the core of Gentleman's algorithm for matrix multiplication (see e.g. [Qu94]). The idea is that two $n \times n$ matrices `A` and `B` are split into a matrix of $m \times m$ partitions (where $m = n/\sqrt{p}$ and p is the number of processors). Initially, the partitions of `A` and `B` are shifted cyclically in horizontal and vertical direction, respectively. More precisely, a partition in row i (column j) is shifted i (j) positions to the left (up) (lines 14,15). Then, the result matrix is initialized with zeros (line 16). The core of the algorithm is the repeated local matrix multiplication at each processor (line 18) followed by a cyclic shift of `A` and `B` by one position in horizontal and vertical direction, respectively. Note that the local multiplication at each processor (line 18) is achieved by partially applying the scalar product function `sprod` to `A` and `B`. The auxiliary function `curry` is used to transform the C++ function `sprod` in such a way that it can be partially applied, i.e. applied to less arguments than it actually needs. Note that `sprod` requires five arguments; two of them are provided by the partial application resulting in a function which needs three more arguments, and such a function is exactly what `mapIndexInPlace` expects as an argument. `R.mapIndexInPlace(curry(sprod<C,n>)(A)(B))` will apply `curry(sprod<C,n>)(A)(B)` to every row index i, column index j, and the corresponding element of $\mathtt{R}_{i,j}$ at position (i, j), i.e. it will provide the three missing arguments of `sprod`. Partial applications are frequently used in the example applications used in section 3. The "magic" curry function has been taken from the C++ template library Fact [St00].

2.2 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 with 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.

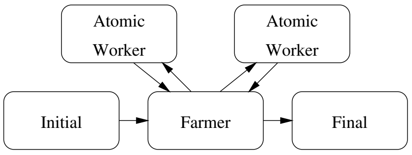
Besides pipelines, the skeleton library offers farms and parallel composition. In a farm, a farmer process accepts a sequence of inputs and assigns each of them to one of several workers. Farms are convenient for applications which require some load balancing such as divide & conquer algorithms. The parallel composition works similar to the farm. However, each input is forwarded to every worker.

Each task parallel skeleton has the same property as an atomic process, namely it accepts a sequence of inputs and produces a sequence of outputs. This allows the task parallel skeletons to be arbitrarily nested. Task parallel skeletons like pipeline and farm are provided by many skeleton systems. The two-tier model and the concrete formulation of the pipeline and farm skeletons in our library have been taken from P^3L [DPP97].

In the example in Fig. 2, a pipeline of an initial atomic process, a farm of two atomic workers, and a final atomic process is constructed. In the C++ binding, there is a class for every task parallel skeleton. All these classes are subclasses of the abstract class `Process`. A task parallel application proceeds in two steps. First, a process topology is created by using the constructors of the mentioned class. This process topology reflects the actual nesting of skeletons. Then, this system of processes is started by applying method `start()` to 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. When constructing an atomic process, the argument function of the constructor tells how each input is transformed into an output value. Again, such a function can be either a C++ function or a partial application. In Fig. 2, worker i multiplies all inputs by $i + 1$. The initial and final atomic processes are special, since they do not consume inputs and produce outputs, respectively.

2.3 Global vs. Local View

The matrix multiplication example (see Fig. 1) demonstrates that the programmer has a global view of the computation. Distributed data structures are manipulated as a whole. If you compare this to a corresponding program which



```
#include ‘‘Skeleton.h"

static int current = 0;
static const int numworkers = 2;

int* init(){if (current++ < 99) return &current;
            else return NULL;}

int times(int x, int y){return x * y;}

void fin(int n){cout << ‘‘result: ‘‘ << n << endl;}

int main(int argc, char **argv){
    InitSkeletons(argc,argv);

  // step 1: create a process topology (using C++ constructors)
    Initial<int>       p1(init);
    Process*           p2[numworkers];
    for (int i=0; i<numworkers; i++)
       p2[i] = new Atomic<int,int>(curry(times)(i+1),1);
    Farm<int,int>      p3(p2,numworkers);
    Final<int>         p4(fin);
    Pipe               p5(p1,p3,p4);

  // step 2: start the system of processes
    p5.start();

    TerminateSkeletons();}
```

Fig. 2. Task parallel example application.

uses MPI directly (see [Ku02b]), you will note that the latter is based on a local view of the computation. A typical piece of code is the following:

```
int A[m][m], B[m][m], R[m][m];
...
for (r=0; r<sqrtp; r++)
  for (k=0;k<m;k++)
    for (l=0;l<m;l++)
      for (q=0;q<m;q++)
        R[k][l] +=  A[k][q] * B[q][l];
```

It implements the multiplication of the local matrices. There are no global matrices here. A, B, R are just the locally available partitions of the global data structures, which only exist in the programmer's mind. This has several consequences. First, the index computations are different. In the above piece of code, all array indices start at 0, while the indices are usually global when using skeletons. For partition (i,j), they start at $(i*m, j*m)$ (where m is the number of rows and columns in each partition). If the actual computation depends on the global index, the global view is more convenient and typically more efficient. Otherwise the local view is preferable. Without skeletons, there is no choice: only the local view is possible. However for skeletons, we can provide the best of both worlds. We just have to add access operations, which support the local view or a combination of global and local view in order to provide the most convenient operations. In the matrix multiplication example, we could replace the code for the scalar product for instance by:

```
template <class C, int n>
C sprod(const DistributedMatrix<C,n,n>& A,
         const DistributedMatrix<C,n,n>& B, int i, int j, C Cij){
  C sum = Cij;
  for (int k=0; k<A.getLocalRows(); k++)
     sum += A.getGlobalLocal(i,k) * B.getLocalGlobal(k,j);
  return sum;}
```

Here, i and j are global indices, while k is a local one.

In general, the local view has the advantage that the computation may be more easily optimized to the special needs at each processor. Consider Gaussian elimination (see e.g. [Ku02a,Ku02b]). From a global perspective, one would have to apply the pivot operation to every element of the matrix. For the columns to the left of the pivot column this computation would be redundant, and it could be easily avoided based on a local perspective. For such situations, the skeleton library provides specialized versions of the corresponding skeletons, which do not refer to single elements of a distributed data structure but to every partition as a whole. This allows similar optimizations than the local view at the price of lesser elegance. In the above situation the skeleton `mapPartitionInPlace` could be used instead of `mapIndexInPlace` in order to process a partition as a whole.

3 Experimental Results

Since the skeleton library has been implemented on top of MPI, it will hardly be possible to outperform hand-written C++/MPI code. Thus, the question is whether the implementation of the skeletons causes some loss of performance and how large this will be. In order to investigate this, we have implemented a couple of example programs in both ways, with skeletons and using MPI directly. In particular, we have considered the following kernels of parallel applications:

Matrix Multiplication based on the algorithm of Gentleman [Qu94]:
this example uses distributed matrices and the skeletons `mapIndexInPlace`, `rotateRows`, and `rotateCols`.

All Pairs Shortest Paths based on matrix computations:
this uses essentially the same skeletons as matrix multiplication.

Gaussian Elimination: The matrix is split horizontally into partitions of several rows. Repeatedly, the pivot row is broadcasted to every processor and

Table 1. Runtimes for different benchmarks with and without skeletons.

		$p = 4$			$p = 16$		
example	n	skel.	MPI	quotient	skel.	MPI	quotient
matrix multiplication	256	0.459	0.413	1.11	0.131	0.124	1.06
	512	4.149	3.488	1.19	1.057	0.807	1.31
	1024	35.203	29.772	1.18	8.624	6.962	1.24
shortest paths	1024	393.769	197.979	1.99	93.825	44.761	2.10
Gaussian elimination	1024	13.816	9.574	1.44	7.401	4.045	1.83
FFT	2^{18}	2.127	1.295	1.64	0.636	0.403	1.58
samplesort	2^{18}	1.599	†	-	0.774	†	-

the pivot operation is executed everywhere. This example mainly uses `mapPartitionInPlace` and `broadcastPartition`.

FFT: This example is a variant of the FFT algorithm shown in [Qu94]. It uses `mapIndexInPlace` and `permutePartition`.

Samplesort: (see [Qu94]) This well-known sorting algorithm uses the skeletons `mapPartition`, `gather`, and `allToAll`.

It is important to note that our implementation of the skeleton library only uses `MPI_Send` and `MPI_Recv`, respectively, but refrains from collective operations, since we want to keep the implementation as portable as possible. In particular, we are able to port the library to any other message passing platform by changing only a few lines, even if this platform does not provide collective operations. Interestingly, this is the reason why the skeleton-based implementation of samplesort also works for large problem sizes, while the direct MPI implementation based on the rich set of collective operations happened to crash for medium problem sizes already (probably due to buffering problems). Table 1 shows the results for the above benchmarks on a Siemens hpcLine running RedHat Linux 7.2 [PC^2]. Columns 4, 5, 7, and 8 contain the runtimes (in seconds) with and without skeletons (i.e. using MPI directly), respectively. p is the number of processors, n is the problem size (#rows, #elements). Columns 5 and 8 show that the skeleton-based versions are between 1.1 and 2.1 times slower than their MPI-based counterparts. This is mainly caused by the overhead for parameter passing introduced by the higher-order functions. This overhead can be reduced by extensive inlining. Moreover, the mentioned global optimizations for skeletons have not yet been implemented. The scalability of the skeletons is similar to that of MPI.

4 Conclusions and Future Work

We have shown that it is possible to provide algorithmic skeletons in form of a library rather than within a new programming language. This will facilitate their use for typical parallel programmers. The library smoothly combines the main features of existing skeletons. In particular, it provides task parallel skeletons generating a system of communicating processes as well as data parallel skeletons working in parallel on a distributed data structure. Task and data parallelism are combined based on the two-tier model. The C++ binding of the skeleton library has been presented. Moreover, experimental results for some draft implementation based on MPI show that the higher programming level can be gained without a significant performance penalty. Communication problems like deadlocks and starvation are avoided since there are no individual messages but coordinated systems of messages for each skeleton.

As future work, tools like cost analyzers, optimizers, and debuggers will have to be developed for the standard skeleton library.

Acknowledgements. The presented skeleton library is the result of many stimulating discussions with Murray Cole, Sergei Gorlatch, Gaétan Hains, Quentin Miller, Susanna Pelagatti, Jörg Striegnitz and others. Their influence is gratefully acknowledged.

References

BK96. G.H. Botorog, H. Kuchen: Efficient Parallel Programming with Algorithmic Skeletons, Proceedings of EuroPar '96, LNCS 1123, Springer, 1996.

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

Co89. M. I. Cole: *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press, 1989.

DPP97. M. Danelutto, F. Pasqualetti and S. Pelagatti Skeletons for Data Parallelism in p3l , EURO-PAR '97, Springer LNCS 1300, pp. 619-628, 1997.

Da93. J. Darlington, A. J. Field, P. G. Harrison et al: Parallel Programming Using Skeleton Functions, in *Proceedings of PARLE '93*, LNCS 694, Springer, 1993.

Da95. J. Darlington, Y. Guo, H. W. To, J. Yang: Functional Skeletons for Parallel Coordination, in *Proceedings of EURO-PAR '95*, LNCS 966, Springer, 1995.

FOT92. I. Foster, R. Olson, S. Tuecke: Productive Parallel Programming: The PCN Approach, in *Scientific Programming*, Vol. 1, No. 1, 1992.

GLS99. W. Gropp, E. Lusk, A. Skjellum: Using MPI, MIT Press, 1999.

HPF93. High Performance Fortran Forum: High Performance Fortran Language Specification, Scientific Programming, Vol. 2(1), 1993.

Ka98. S. Karmesin et al.: Array Design and Expression Evaluation in POOMA II. ISCOPE 1998, pp. 231-238.

KPS94. H. Kuchen, R. Plasmeijer, H. Stoltze: Efficient Distributed Memory Implementation of a Data Parallel Functional Language, *PARLE*, LNCS 817, 1994.

Ku02a. H. Kuchen: A Skeleton Library, Technical Report, Univ. Münster, 2002.

Ku02b. H. Kuchen: The Skeleton Library Web Pages, http://danae.uni-muenster.de/lehre/kuchen/Skeletons/

PC^2. PC^2: http://www.upb.de/pc2/services/systems/psc/index.html.

Qu94. M. J. Quinn: *Parallel Computing: Theory and Practice*, McGraw Hill, 1994.

SHM97. David B. Skillicorn, Jonathan M. D. Hill, W. F. McColl: Questions and answers about BSP, Scientific Programming 6(3): 249-274, 1997.

Sk94. D. Skillicorn: Foundations of Parallel Programming, Cambridge U. Press, 1994.

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