OPTIMIZING SKELETAL STREAM PROCESSING FOR DIVIDE AND CONQUER

Michael Poldner and Herbert Kuchen
University of Münster, Department of Information Systems, Leonardo Campus 3, D-48149 Münster, Germany

Keywords: Parallel Computing, Algorithmic Skeletons, Divide & Conquer.

Abstract: Algorithmic skeletons intend to simplify parallel programming by providing recurring forms of program structure as predefined components. We present a new distributed task parallel skeleton for a very general class of divide and conquer algorithms for MIMD machines with distributed memory. Our approach combines skeletal internal task parallelism with stream parallelism. This approach is compared to alternative topologies for a task parallel divide and conquer skeleton with respect to their aptitude of solving streams of divide and conquer problems. Based on experimental results for matrix chain multiplication problems, we show that our new approach enables a better processor load and memory utilization of the engaged solvers, and reduces communication costs.

1 INTRODUCTION

Parallel programming of MIMD machines with distributed memory is typically based on standard message passing libraries such as MPI (MPI, 2008), which leads to platform independent and efficient software. However, the programming level is still rather low and thus error-prone and time consuming. Programmers have to fight against low-level communication problems such as deadlocks, starvation, and termination detection. Moreover, the program is split into a set of processes which are assigned to the different processors, whereas each process only has a local view of the overall activity. A global view of the overall computation only exists in the programmer's mind, and there is no way to express it more directly on this level.

For this reasons many approaches have been suggested, which provide a higher level of abstraction and an easier program development to overcome the mentioned disadvantages. 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 to form the skeletal structure of the entire parallel application. These components are referred to as algorithmic skeletons (E. Alba, 2002; Cole, 1989; J. Darlington, 1995; H. Kuchen, 2002; Kuchen, 2002; K. Matsuzaki, 2006; Pelagatti, 2003). Typically, algorithmic skeletons are offered to the user as higher-order functions, which get the details of the specific application problem as argument functions. In this way the user can adapt the skeletons 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.

Depending on the kind of parallelism used, skeletons can roughly be classified into data parallel and task parallel ones. A data parallel skeleton (G. H. Botorog, 1996; Kuchen, 2002; Kuchen, 2004) works on a distributed data structure such as a distributed array or matrix as a whole, performing the same operations on some or all elements of this data structure. Task parallel skeletons (A. Benoit, 2005; Cole, 2004; H. Kuchen, 2002; Kuchen, 2002; Poldner and Kuchen, 2005; Poldner and Kuchen, 2006) create a system of processes communicating via streams of data by nesting and combining predefined process topologies such as pipeline, farm, parallel composition, divide and conquer, and branch and bound. In the present paper we will consider task parallel divide and conquer skeletons with respect to their aptitude of solving streams of divide and conquer problems.

Divide and conquer is a well known computation paradigm, in which the solution to a problem is obtained by dividing the original problem into smaller subproblems, solving the subproblems recursively, and combining the partial solutions to the final solution. A simple problem is solved directly without dividing it 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 or the Delaunay triangulation, 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. In many cases there is the need of solving multiple divide and conquer problems in sequence. Examples here are the triangulation of several geometric figures, matrix chain multiplication, and factoring of large numbers.

In the present paper we will consider different task parallel divide and conquer skeletons in the context of the skeleton library Muesli (Kuchen, 2002; Poldner and Kuchen, 2005; Poldner and Kuchen, 2006; Poldner and Kuchen, 2008b; Poldner and Kuchen, 2008a), which are used to solve streams of divide and conquer problems. Muesli is based on MPI internally in order to inherit its platform independence. We have implemented a new fully distributed divide and conquer skeleton and compare it to a farm of sequentially working divide and conquer solvers, and to a fully distributed divide and conquer skeleton used in a previous version of Muesli. We will show that our new approach enables a better processor load and memory utilization of the engaged solvers, and reduces communication costs.

The rest of this paper is structured as follows. In section 2, we briefly introduce divide and conquer skeletons and basic notions. In Section 3, we present different parallel implementation schemes of the considered skeletons in the framework of the skeleton library Muesli, and discuss their application in the context of streams. Section 4 contains experimental results demonstrating the strength of our new distributed design. In Section 5 we compare our approach to related work, and finally, we conclude and point out future work in section 6.

2 DIVIDE AND CONQUER SKELETONS

A divide and conquer algorithm solves an initial problem by dividing it into smaller subproblems, solving the subproblems recursively, and combining the partial solutions in order to get the final solution, whereas simple problems are solved directly without dividing them further. The computation can be viewed as a process of expanding and shrinking a tree, in which the nodes represent problem instances and partial solutions, respectively (Fig.1).

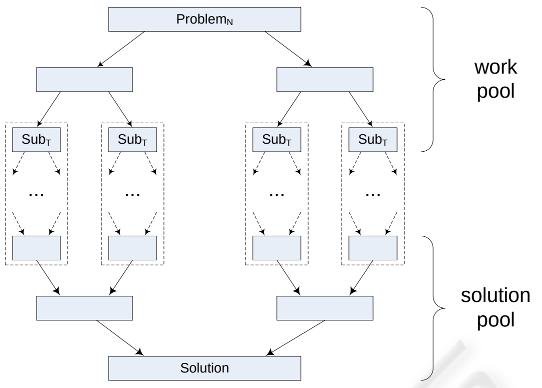

Figure 1: A divide and conquer tree.

A divide and conquer skeleton offers this basic strategy to the user as predefined parallel component. Typically, the user has to supply four application specific basic operators, with which the user can exactly adapt the skeleton to the considered problem: a `divide` operator which describes how the considered (sub)problem can be divided into subproblems, a `combine` operator which specifies how partial solutions can be combined to the solution of the considered parent problem, an `isSimple` operator which indicates if a subproblem is simple enough that it can be solved directly, and last but not least a `solve` operator, which solves a simple problem. During the solution of a concrete divide and conquer problem, the skeleton generates a multitude of subproblems and partial solutions (i.e. the nodes in the divide and conquer tree), which are stored in a *work pool*, and a *solution pool* respectively. In the beginning the work pool only contains the initial problem, and the solution pool is empty. In each iteration one such problem is selected from the work pool corresponding to a particular traversal strategy such as depth first or breadth first. The problem is either divided into d subproblems, which are stored again in the work pool, or it is solved, and its solution is stored in the solution pool. It may happen that a problem of size s is reduced to d subproblems of sizes $s_1, \ldots, s_d$ with $\sum_{i=1}^{d} s_i > s$, e.g. for the Karatsuba multiplication algorithm for large integers or the Strassen algorithm for matrix multiplication. At least in this case, a depth first strategy is recommended in order to avoid memory problems. The order in which solutions are stored in the solution pool depends on the implemented traversal strategy. It is recommended to combine partial solutions as soon as possible in order to free memory. If the solution pool contains d partial solutions, which can be combined, they can be replaced by the solution of the corresponding parent problem. In the end of the computation the work pool is empty and the solution pool only contains the solution of the initial problem.

2.1 A Design for a Fully Distributed Divide and Conquer Skeleton

Figure 2 illustrates the design of the distributed divide and conquer skeleton (DCSkeleton), which has been used in a previous version of our skeleton library *Muesli* (Poldner and Kuchen, 2008b). 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 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. Several topologies for connecting the solvers are possible. The topology for connecting the solvers is exchangeable without having to adapt the load balancing or termination detection algorithm. To simplify matters, in this paper we will consider an all-to-all topology. For larger number of processors, topologies like torus or hypercube may lead to a faster propagation of work to idle processors within the startup phase.

Exactly one of the solvers serves as an interface to the skeleton, which is referred to as master solver. The master solver receives a new divide and conquer problem from the predecessor and delivers the solution to its successor. 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. Moreover, if the number of subproblems generated by the DCSkeleton is limited to one by a corresponding implementation of the isSimple operator, the user can enable a pure sequential computation. 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, and the workpool and solution pool are bypassed as well.

The DCSkeleton consumes a stream of input values and produces a stream of output values. If the master solver receives a new divide and conquer

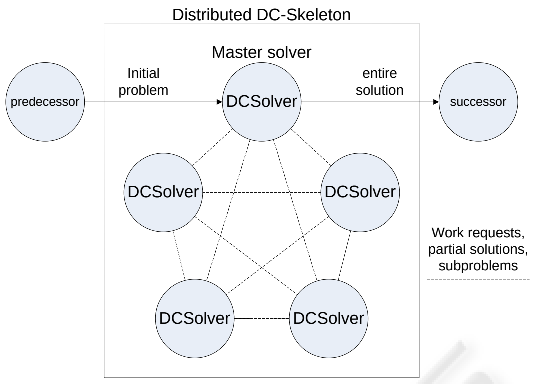

Figure 2: A fully distributed divide and conquer skeleton.

problem, the communication with the predecessor is blocked until the received problem is solved. Thus, the DCSkeleton can only process one divide and conquer problem at a time.

There are different variants for the initialization of the skeleton with the objective of providing each solver with a certain amount of work within the startup phase. Our skeleton uses the most common approach, namely root initialization, i.e. the initial D&C problem is inserted into the local work pool of the master solver. Subproblems are distributed according to the load balancing scheme applied by the solvers.

2.2 Forms of Parallelism

Considering task parallel process systems, two forms of parallelism can be identified. The first one is the skeletal internal parallelism, which follows from processing one single problem by several workers in parallel. The DCSkeleton benefits from skeletal internal parallelism by solving one problem by all engaged solvers in parallel. The second form of parallelism is stream parallelism, which follows from the possibility of splitting up one data stream into many streams and processing these streams in parallel. Somewhere in the process system these streams have to be routed to a common junction point in order to reunite them again. The farm topology depicted in figure 3, which is offered by the *Muesli* skeleton library, benefits from stream parallelism. Each worker of the farm takes a new problem from its own stream, so that several problems can be processed independently from each other within the farm at the same time. In this paper, we consider a farm of DCSkeletons which are each configured to a purely sequential computation as described above.

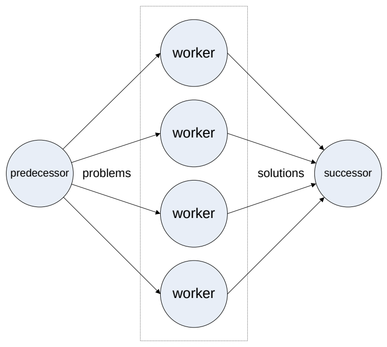

Figure 3: A farm skeleton.

2.3 Streams of Divide and Conquer Problems

Many 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 factoring of several large numbers. Using the *Muesli* skeleton library, the different tasks can be represented as a stream, which is routed to either a single DCSkeleton (fig. 2) or a farm (fig. 3).

MPI is internally based on a two-level communication protocol, the eager protocol for sending messages less than $32KB$ and the rendezvous protocol for larger messages (Poldner and Kuchen, 2008a). For the asynchronous eager protocol the assumption is made that the receiving process can store the message if it is sent and no receive operation has been posted by the receiver. In this case the receiving process must provide a certain amount of buffer space to buffer the message upon its arrival. In contrast to the eager protocol, the rendezvous protocol writes the data directly to the receive buffer without intermediate buffering. This synchronous protocol requires an acknowledgment from a matching receive in order for the send operation to transmit the data. This protocol leads to a higher bandwidth but also to a higher latency due to the necessary handshaking between sender and receiver.

In the following we assume problem sizes greater than $32KB$, such as multiplying at least two 64×64 integer matrices, which enables the rendezvous protocol. Moreover, we act on the assumption that the time between the arrivals of two problems is less than the time for solving it sequentially. Otherwise we are not able to speed up the overall computation because the divide and conquer skeleton cannot be a bottleneck of the process system.

The DCSkeleton processes one divide and conquer problem by N solvers at a time, while the fully distributed memory is available for the solution process. Figure 4 depicts the utilization of the DCSkeleton within the startup, main, and end phase of solving such a problem. In the beginning, a certain amount of work has to be generated by divide calls and distributed among all solvers until each solver is provided with work. For this reason, this skeleton shows high idle times within the startup phase. In particular when the initial problem is divided by the master solver, all remaining solvers are idle. Within the main phase of the computation all solvers are working to full capacity. Moreover, this phase is characterized by low communication costs due to the fact that the solvers predominantly work on their local pools, which is essential to achieve good speedups. Within the end phase, partial solutions have to be collected and combined to parent solutions. At the end, only the master solver combines the entire solution, and all other solvers are idle. Thus, the end phase is characterized by high idle times as well. The duration of the startup and end phase results from both, the complexity of dividing problems and combining partial solutions, and the sizes of the subproblems and partial solutions which have to be sent over the network.

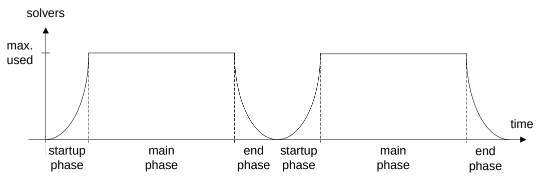

Figure 4: utilization of the DCSkeleton.

Processing streams of divide and conquer problems can be seen as a sequence of several startup, main, and end phases. If the arrival rate of new problems is high, the master solver quickly becomes a bottleneck of the system, because all engaged solvers, which are running idle within an end phase of a computation have to wait for new work which is not delegated to them until the following startup phase. This is caused by the fact that the master solver represents the only interface to the skeleton. The more solvers are used in the skeleton, the faster a problem will be solved in the main phase of the computation. If the arrival rate of new problems is low, the DCSkeleton can be adapted to this rate by adjusting the number of engaged solvers. Thus, the overall time for processing all problems in the stream is the sum of the subtracted times for solving the single problems.

The farm enables the user to solve N problems by N workers in parallel, whereas each of the solvers is represented by a sequentially working DCSkeleton. A DCSkeleton configured to work sequentially has

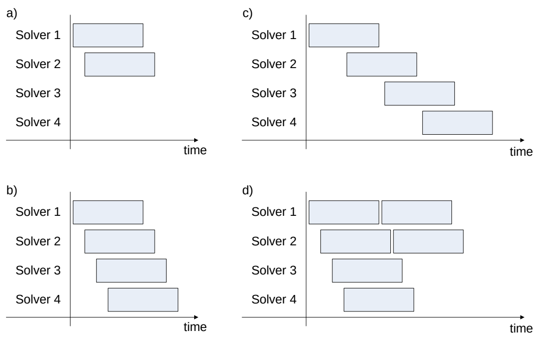

Figure 5: Utilization of workers in a farm.

the advantage that it can speed up the computation by bypassing the work and solution pool, and no divide and combine operator calls as well as load balancing are necessary. The disadvantage is that only the local memory of a solver is available for the solution, so if the problems in the stream are expected to be big, memory problems can occur. If there are $M < N$ problems in the stream, $N - M$ workers of the farm are never supplied with work, which causes high idle times and bad runtimes irrespective of the arrival rate of the problems (Fig. 5a). If the stream contains $M = N$ problems, each worker is sooner or later provided with work (Fig. 5b). However, due to the synchronous rendezvous protocol the solvers are provided with work one after another. Thus, in the beginning many solvers are waiting for work and are idle. Due to the sequential computation these solvers are not able to support other solvers. The same problem occurs in the end of the computation, when the solvers working on the first problems are running idle. The duration of the idle phases depends on the arrival rate of problems. If the rate is high, good runtimes can be achieved due to low idle times. However, a low arrival rate may lead to the situation as shown in figure 5c. In this case, the first problem submitted to a solver is solved before the last solvers of the farm are provided with work. Thus, the number of problems that can be processed in parallel by the farm is less than N. In particular for big problems a bad load balancing is caused if the number of problems in the stream is a little higher than the number of solvers (Fig. 5d). In this case only few problems are delegated to each solver, and the amount of work assigned to each solver differs considerably. Only if the number of problems in the stream is clearly higher than the number of solvers, and the number of solvers is adapted to the arrival rate, the average amount of work each solver has to be done is nearly the same and good runtimes can be expected.

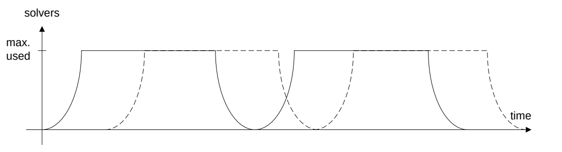

Figure 6: Overlapped startup, main, and end phases.

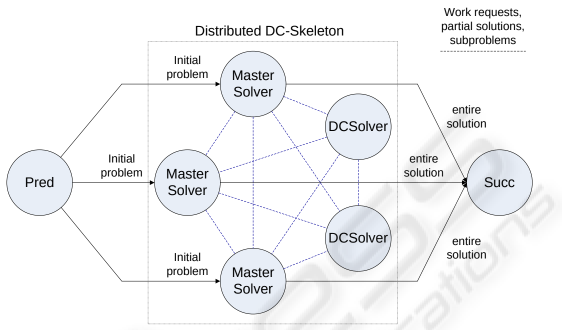

Figure 7: A fully distributed divide and conquer skeleton for stream processing.

2.4 A New Divide and Conquer Skeleton for Stream Processing

The stream processing can be optimized by overlapping phases of high workload with phases of poor workload. In case of the DCSkeleton we find high work load within the main phases and poor workload within the startup und end phases of the computation. If the solution of a divide and conquer problem is in its startup or end phase, only few or even no subproblems exist in the system which can be distributed among the solvers. As shown in figure 6, the phases can be overlapped if more than one divide and conquer problem is processed by the skeleton at a time. If the computation of a solution is in its startup or end phase, the processing of another problem may be in its main phase. This leads to a more balanced processor load due to the fact that the amount of work is increased within the skeleton and thus idle times are reduced. The number of problems which are prepared for load distribution increases linearly with the number of divide and conquer problems solved in parallel. Thus, a less fine-granular decomposition of each divide and conquer problem is necessary to guarantee a sufficient amount of work for all solvers the more divide and conquer problems are solved in parallel. By generating fewer but bigger subproblems the efficiency of the skeleton can be increased not only by reducing the number of divide and combine operator calls, but also by raising the sequential proportion of the computation by applying solve on larger problems. Figure 7 illustrates the design of our new divide and conquer skeleton for stream processing (StreamDC), which is based on the DCSkeleton. In contrast to the DCSkeleton it consists of *n* master solvers receiving new divide and conquer problems from the predecessor. Each solver maintains a multi-part work and solution pool to distinguish between subproblems which emanate from different initial problems. If the workpool stores subproblems which emanate from problems received from another solver, these problems are processed first. In this case, a master solver delegating a subproblem is supplied with its corresponded partial solution more quickly. This speeds up the overall computation time of an internally distributed problem, and an adequate supply of new work to the skeleton is guaranteed by receiving new initial problems from the predecessor more quickly as well. If the arrival rate of new problems is low, this skeleton behaves like a DCSkeleton. An idle master solver sends work requests and receives subproblems from its neighbors. If the arrival rate of problems is high, each of the master solvers receives new problems from its predecessor, which increases the amount of work within the skeleton. For larger number of processors, this leads to a faster propagation of work to idle solvers in the beginning of the computation, and increases the utilization of solvers during the whole computation as shown above. Thus, by applying the StreamDC skeleton with application specific parameters, it can be configured to be a hybrid of a pure stream processing farm and the DCSkeleton. It can be adapted to the arrival rate and the size of the divide and conquer problems which are to be solved so that the distributed memory utilization is improved. For this reason, the StreamDC is able to solve problems, which cannot be solved by a sequential DCSkeleton used in farms due to the lack of memory. In comparison to the DCSkeleton the new StreamDC skeleton benefits from overlapping the startup and end phases of solving single problems by solving several problems in parallel. Moreover, fewer problems must be prepared for load distribution which reduces divide and combine operator calls and increases the sequential part of the computation.

The code fragment in figure 8 illustrates the application of our StreamDC-Skeleton in the context of the *Muesli* skeleton library. It constructs the process topology shown in Fig. 7.

In a first step the process topology is created using C++ constructors. The process topology consists of an initial process, a dc process, and a final process connected by a pipeline skeleton. The initial process is parameterized by a `generateStream` method generating a stream of initial D&C problems that are to be solved. The constructor `StreamDC` generates a total number of $n = 5$ solvers (whereas 3 are master solvers), which are provided with the four basic operators `divide`, `combine`, `solveSeq`, and `isSimple`. The function `isSimple` has to return `true` if the subproblem is simple enough to be solved sequentially by the `solveSeq` operator. The parameter *d* corresponds to the degree of the D&C tree and describes how many subproblems are generated by `divide` and how many subproblems are required by `combine` to generate the solution of the corresponding parent problem.

```
int main(int argc, char* argv[]) {
    InitSkeletons(argc,argv);
    // step 1: create process topology
    Initial<Problem> initial(generateStreamOfProblems);
    StreamDC<Problem,Solution>
        dc(divide, combine, solveSeq, isSimple, d, 5, 3);
    Final<Problem> final(fin);
    Pipe pipe(initial,dc,final);
    // step 2: start process topology
    pipe.start();
    TerminateSkeletons();
}
```

Figure 8: Example application using a distributed divide and conquer skeleton.

3 EXPERIMENTAL RESULTS

The parallel test environment for our experiments is an IBM workstation cluster (ZIV, 2008) of sixteen uniform PCs connected by a Myrinet (Myricom, 2008). Each PC has an Intel Xeon EM64T processor (3.6 GHz), 1 MB L2 cache, and 4 GB memory, running Redhat Enterprise Linux 4, gcc version 3.4.6, and the MPICH-GM implementation of MPI.

In order to evaluate the performance and scalability of the StreamDC skeleton, we have considered Strassen's algorithm for matrix multiplication ($O(N^{\log_2 7})$, where $\log_2 7 \approx 2,808$) in order to multiply two randomly generated 1024×1024 integer matrices. The Strassen algorithm (Strassen, 1969) reduces a multiplication of two $N \times N$ matrices to seven multiplications of $\frac{N}{2} \times \frac{N}{2}$ matrices. The stream consists of 20 matrix multiplication problems, which represents single matrix multiplications when solving a matrix chain multiplication problem $A_1 \cdot \ldots \cdot A_n$ (Baumgartner, 2002; T.C. Hu, 1982; T.C. Hu, 1984). Note that the skeleton behaves non-deterministically in the way the load is distributed. Generating only a few big subproblems can lead to an unbalanced workload and to high idle times. In order to get reliable results, we have repeated each run up to 50 times and computed the average runtimes which are shown in figure 9. Figure 10 depicts the correspond-

	DC Solvers													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
DCSkeleton	**95,94**	**63,71**	**44,60**	**34,99**	**29,38**	**24,98**	**20,09**	**19,70**	**18,29**	**17,35**	**16,43**	**15,64**	**15,06**	**14,74**
2 MS		54,78	41,04	31,33	25,69	21,87	19,14	17,06	15,14	14,10	13,28	12,05	11,65	10,96
3 MS			36,46	29,72	24,61	20,93	18,14	16,35	14,57	13,43	12,70	11,35	10,98	10,40
4 MS				27,61	23,40	20,21	17,66	15,75	14,46	12,97	11,72	11,37	10,35	9,97
5 MS					22,48	19,51	17,23	15,55	13,97	12,61	11,64	11,00	10,20	9,30
6 MS						18,51	16,80	15,40	13,85	12,53	11,48	10,50	9,80	9,15
7 MS							16,22	14,81	13,57	12,48	11,45	10,65	9,77	9,20
8 MS								14,54	13,40	12,27	11,14	10,54	9,84	9,25
9 MS									13,44	12,24	11,21	10,41	9,77	8,96
10 MS										12,24	11,02	10,70	9,93	9,26
11 MS											11,35	10,34	9,66	9,11
12 MS												10,51	9,91	9,14
13 MS													9,81	9,33
14 MS														9,31
StreamDC	**95,94**	**54,78**	**36,46**	**27,61**	**22,48**	**18,51**	**16,22**	**14,54**	**13,40**	**12,24**	**11,02**	**10,34**	**9,66**	**8,96**
DC Farm	**95,95**	**50,40**	**33,88**	**24,54**	**19,81**	**19,40**	**15,72**	**15,74**	**14,62**	**11,12**	**10,95**	**10,83**	**10,76**	**10,75**

Figure 9: Runtimes for StreamDC, DCSkeleton and a sequential farm processing matrix multiplication problems.

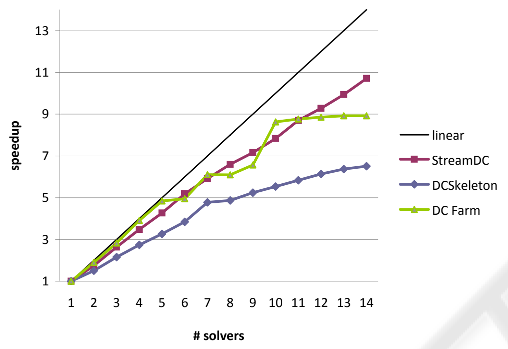

Figure 10: Speedups for StreamDC, DCSkeleton and a sequential farm processing matrix multiplication problems.

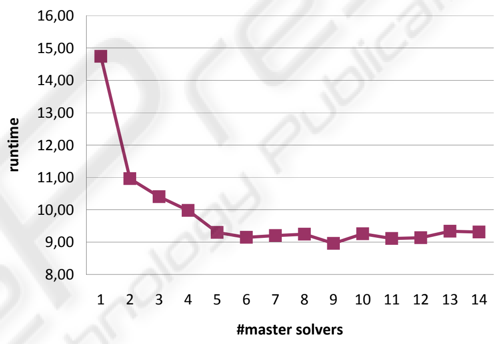

Figure 11: Computation times depending on the number of master solvers.

ing speedups for the StreamDC, the DCSkeleton and the farm of sequential DCSkeletons.
As one would expect, the StreamDC skeleton, which combines stream parallelism with internal task parallelism, is clearly superior to the DCSkeleton, which only provides internal task parallelism. This is by the fact that idle phases are reduced by overlapping the startup and end phases of a solution. Moreover, the number of subproblems is reduced which are prepared for load distribution. Thus, the overhead for `divide` and `combine` operator calls is decreased as well. The speedups for the farm show a kind of stairs effect which can be explained by figure 5d. In this case the solvers are provided with a highly unbalanced amount of work. In contrast to the StreamDC skeleton, the farm is not able to do a load balancing.
Figure 11 shows the impact of additional master solvers used for the computation. The runtime can be decreased significantly by the use of up to five master solvers. Then, the skeleton can not benefit from additional master solvers because of the fact, that all solvers are already busy.

4 RELATED WORK

We are not aware of any previous systematic analysis of different implementation schemes for stream processing divide and conquer skeletons in the literature. In (Aldinucci and Danelutto, 1999) it is shown that any arbitrary composition of stream parallel skeletons can be rewritten into an equivalent ”normal form” skeleton composition, which offers better or equal runtimes compared to the original program. This normal form is defined as a single farm build around a sequential worker code. As we have shown in our analysis and experiments, this conclusion warrants a critical assessment. In particular for streams with few big problems the workload can be highly unbalanced in the farm. Moreover, the solution of big divide and conquer problems may require much more memory than provided by a sequential worker. Thus, such problems can only be solved by several solvers in parallel.

Recent skeleton libraries such as eSkel (Cole, 2004), skeTo (K. Matsuzaki, 2006), and MaLLBba

(E. Alba, 2002; J.R. González, 2004) include skeletons for divide and conquer. The MaLLBa implementation of the divide and conquer skeleton presented in (E. Alba, 2002) is based on a farm (master-slave) strategy, which is inapplicable for streams (Poldner and Kuchen, 2008a). The distributed approach discussed in (J.R. González, 2004) offers the same user interface as the MaLLBa skeleton and can be integrated into the MaLLBa framework. Unfortunately, neither runtimes of the considered example applications are presented nor the design was discussed in the context of streams. In (Cole, 1997), Cole suggests to offer divide and combine as independent skeletons. But this approach has not been implemented in eSkel. The eSkel *Butterfly*-Skeleton (Cole, 2004) is based on *group partitioning* and supports divide and conquer algorithms in which all activity occurs in the divide phase. In contrast to our approach, the number of processors used for the Butterfly skeleton starts from a power of two. This is due to the group partitioning strategy. Note that algorithms like Strassen or Karatsuba produce a number of subproblems which is not a power of two. The skeTo library (K. Matsuzaki, 2006) only provides data parallel skeletons and is based on the theory of Constructive Algorithmics. Restricted data parallel approaches are discussed in (Bischof, 2005; Gorlatch, 1997). In (Gorlatch, 1997), a processor topology called N-*graph* is presented, which is used for a parallel implementation of a divide and conquer skeleton in a functional context. Hermann presents different general and special forms of divide and conquer skeletons in context of the purely functional programming language HDC, which is a subset of *Haskell* (Herrmann, 2000). A mixed data and task parallel approach can be found in (Y. Bai, 2007). However, we are not aware of any implementation of a divide and conquer skeleton which combines stream processing and internal task parallelism.

5 CONCLUSIONS

We have analyzed alternative topologies for processing streams of divide and conquer problems. After introducing the design of the fully distributed DCSkeleton, which was used in a previous version of the skeleton library *Muesli*, we have considered a distributed farm of a sequentially working DCSkeletons as well as a fully distributed DCSkeleton in the context of streams. We suggest combining skeletal internal task parallelism with stream parallelism to achieve both, better memory utilization and a reduction of idle times of the engaged solvers. Moreover, we present a new divide and conquer skeleton optimized for stream processing. By applying the skeleton with application specific parameters, it can be configured to be a hybrid of a pure stream processing farm and the DCSkeleton, and it can range between both extremes. In comparison to the DCSkeleton the new StreamDC skeleton benefits from overlapping the startup and end phases of solving single problems by solving several problems in parallel. The advantage is, that only few problems must be prepared for load distribution which reduces divide and combine operator calls and increases the sequential part of the computation. As we have shown, the new StreamDC skeleton is clearly superior to the DCSkeleton. In comparison to a farm of sequentially working DCSkeletons it offers a better scalability, which is advantageous in particular when only few divide and conquer problems have to be solved. Moreover, the complete sharing of the distributed memory is a great advantage compared to a farm, in which the solvers only have access to their own local memory. Thus, the new StreamDC is able to solve problems, which cannot be solved by a sequential DCSkeleton used in farms due to the lack of memory. In future work we intend to investigate alternative stream based implementation schemes of skeletons for branch and bound and other search algorithms.

REFERENCES

A. Benoit, M. Cole, J. H. S. G. (2005). Flexible skeletal programming with eskel. In *Proc. EuroPar 2005*. LNCS 3648, 761–770, Springer Verlag, 2005.

Aldinucci, M. and Danelutto, M. (1999). Stream parallel skeleton optimization. In *Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems*, MIT, Boston, USA. IASTED/ACTA press.

Baumgartner, G. (2002). A high-level approach to synthesis of high-performance codes for quantum chemistry.

Bischof, H. (2005). Systematic development of parallel programs using skeletons. In *PhD thesis*. Shaker.

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

Cole, M. (1997). On dividing and conquering independently. In *in Proceedings of Euro-Par'97, LNCS 1300, pages 634-637*. Springer.

Cole, M. (2004). Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. In *Parallel Computing 30(3), 389–406*.

E. Alba, F. Almeida, e. a. (2002). Mallba: A library of skeletons for combinatorial search. In *Proc. Euro-Par 2002*. LNCS 2400, 927–932, Springer Verlag, 2005.

G. H. Botorog, H. K. (1996). Efficient parallel programming with algorithmic skeletons. In *Proc. Euro-Par'96*. LNCS 1123, 718–731, Springer Verlag, 1996.

Gorlatch, S. (1997). N-graphs: scalable topology and design of balanced divide-and-conquer algorithms. In *Parallel Computing, 23(6), pages 687-698*.

H. Kuchen, M. C. (2002). The integration of task and data parallel skeletons. In *Parallel Processing Letters 12(2), 141–155*.

Herrmann, C. (2000). The skeleton-based parallelization of divide-and-conquer recursions. In *PhD thesis*. Logos.

J. Darlington, Y. Guo, H. J. Y. (1995). Parallel skeletons for structured composition. In *in Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 19-28*. ACM Press.

J.R. González, C. León, C. R. (2004). A distributed parallel divide and conquer skeleton. In *in proceedings of PARA'04)*.

K. Matsuzaki, K. Emoto, H. I. Z. H. (2006). A library of constructive skeletons for sequential style of parallel programming. In *in proceedings of 1st international Conference on Scalable Information Systems (INFOSCALE)*.

Kuchen, H. (2002). A skeleton library. In *Euro-Par'02*. LNCS 2400, 620–629, Springer Verlag.

Kuchen, H. (2004). Optimizing sequences of skeleton calls. In *Domain-Specific Program Generation*. LNCS 3016, 254–273, Springer Verlag.

MPI (2008). Message passing interface forum, mpi. In *MPI: A Message-Passing Interface Standard*. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

Myricom (2008). The myricom homepage. http://www.myri.com/.

Pelagatti, S. (2003). Task and data parallelism in p3l. In *Patterns and Skeletons for Parallel and Distributed Computing*. eds. F.A. Rabhi, S. Gorlatch, 155–186, Springer Verlag.

Poldner, M. and Kuchen, H. (2005). Scalable farms. In *in proceedings of Parallel Computing (ParCo)*.

Poldner, M. and Kuchen, H. (2006). Algorithmic skeletons for branch & bound. In *in proceedings of 1st International Conference on Software and Data Technology (ICSOFT), Vol. 1, pages 291-300*.

Poldner, M. and Kuchen, H. (2008a). On implementing the farm skeleton. In *Parallel Processing Letters, Vol. 18, No. 1, pages 117-131*.

Poldner, M. and Kuchen, H. (2008b). Skeletons for divide and conquer algorithms. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*. ACTA Press.

Strassen, V. (1969). Gaussian elimination is not optimal. In *Numerische Mathematik, 13:354-356*.

T.C. Hu, M. S. (1982). Computation of matrix chain products i. In *SIAM Journal on Computing, 11(2):362-373*.

T.C. Hu, M. S. (1984). Computation of matrix chain products ii. In *SIAM Journal on Computing, 13(2):228-251*.

Y. Bai, R. W. (2007). A parallel symmetric block-tridiagonal divide-and-conquer algorithm. In *ACM Transactions on Mathematical Software, Vol. 33, No. 4, Article 25*.

ZIV (2008). Ziv-cluster. http://zivcluster.uni-muenster.de/.