Task Parallel Skeletons for Divide and Conquer

Michael Poldner and Herbert Kuchen

Department of Information Systems, University of Münster
Leonardo Campus 3, D-48149 Münster
{poldner,kuchen}@uni-muenster.de

Abstract. Algorithmic skeletons intend to simplify parallel programming by providing recurring forms of program structure as predefined components. We present a fully distributed task parallel skeleton for a very general class of divide and conquer algorithms for MIMD machines with distributed memory. This approach is compared to a simple master-worker design. Based on experimental results for different example applications such as Mergesort, the Karatsuba multiplication algorithm and Strassen's algorithm for matrix multiplication, we show that the distributed workpool enables good runtimes and in particular scalability. Moreover, we discuss some implementation aspects for the distributed skeleton, such as the underlying data structures and load balancing strategy, in detail. In addition, we present another distributed skeleton which benefits from combining skeletal internal parallelism and stream parallelism. Based on experimental results for matrix chain multiplication problems, we show that this approach enables a better processor load and memory utilization for the engaged solvers, and reduces communication costs.

Key words: Algorithmic Skeletons, parallelism, divide and conquer, stream processing

1 Introduction

Parallel programming of MIMD machines with distributed memory is typically based on standard message passing libraries such as MPI [24], 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 reason 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 [1, 7, 10, 18, 21, 23, 26]. 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, algorithmic skeletons can roughly be classified into data parallel and task parallel ones. Data parallel skeletons [5, 21, 22] process a distributed data structure such as a distributed array or matrix as a whole, e.g. by applying a function to every element or by rotating or permuting its elements. Task-parallel skeletons [3, 9, 18, 21, 27–30] construct a system of processes communicating via streams of data. Such a system is mostly generated by nesting typical building blocks such as farms and pipelines. In the present paper, we will consider task-parallel skeletons for divide and conquer problems.

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. 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, 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 the present paper we will consider different design, implementation, and optimization issues of task parallel divide and conquer skeletons in the context of the skeleton library *Muesli* [21, 27–30]. *Muesli* is build on top of MPI [24] in order to inherit its platform independence. We will show that a master-worker design is less suited to handle divide and conquer problems on distributed memory machines. We have implemented a distributed scheme and present its functionality in detail. We will show that we can achieve a good load balance while minimizing communication costs. This is supported by several test results of three example applications. Moreover, we discuss how to optimize the skeleton for processing streams of divide and conquer problems.

The rest of this paper is structured as follows. In Section 2, we introduce different designs of divide and conquer skeletons in the framework of the skeleton library *Muesli*. Initially, we briefly describe a simple centralized design. Afterwards we will focus on a new fully distributed D&C-Skeleton. Moreover, we discuss how the distributed D&C-Skeleton can be optimized for processing streams of divide and conquer problems. Section 3 contains experimental results demon-

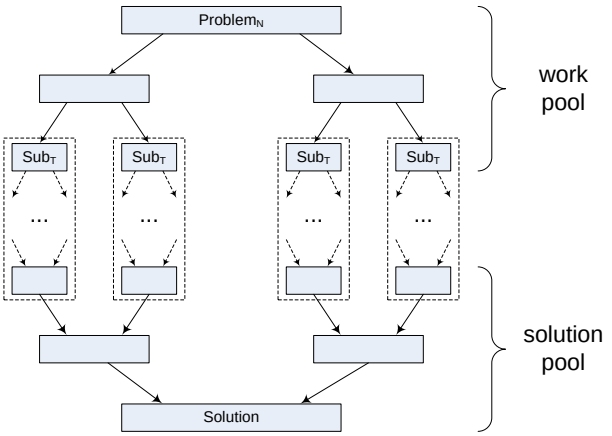

Fig. 1. A divide and conquer tree

strating the strength of the distributed design. In addition, selected experimental results for the stream optimized skeleton are presented. In Section 4 we compare our approach to related work. In Section 5, we conclude and point out future work.

2 Divide and Conquer Skeletons

A divide and conquer skeleton is based on an implementation scheme for divide and conquer and offers it to the user as predefined parallel component. Typically, 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 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). Unprocessed subproblems are stored in a work pool and partial solutions are maintained in a solution pool. In the beginning the work pool only contains the initial problem, which is of size N, and the solution pool is empty. In each iteration one such problem is selected from the workpool 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 workpool, 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 or Strassen algorithm. 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

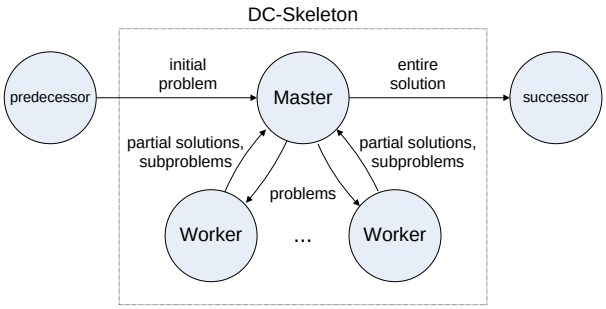

Fig. 2. A master/worker design

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 workpool is empty and the solution pool only contains the solution of the initial problem.

2.1 Master/Worker design

The simplest approach to implement a divide and conquer skeleton is a kind of the master/worker design as depicted in Figure 2. This approach has been used in [1]. The work pool and the solution pool are maintained by the master, which distributes problems to the workers and receives subproblems and solutions from them. When a worker receives a problem, it either solves it or decomposes it into subproblems. The advantage of a single work and solution pool is that it provides a good overall picture of the work still to be done. Moreover, the master knows about all idle workers at any time, which makes it easy to provide each worker with work. The disadvantage is, that accessing the work pool and the solution pool tends to be a bottleneck, as the pools can only be accessed by one worker at a time. This may result in high idle times on the workers' site. Another disadvantage is that the master/worker approach incurs high communication costs, since each subproblem is sent from its producer to the master and propagated to its processing worker. Moreover, the communication time required to send a problem to a worker and to receive in return some subproblems or a solution may be greater than the time needed to do the computation locally. The master's limited memory capacity for maintaining the problems and solutions is another disadvantage of this architecture. As we have shown in [27, 28], a master/worker design is less suited for farms and branch and bound skeletons. Thus, it can be expected that a master/worker design is badly suited to divide and conquer skeletons as well. For this reason, this approach is not considered any further. A promising approach is a fully distributed D&C Skeleton, which is discussed in the following.

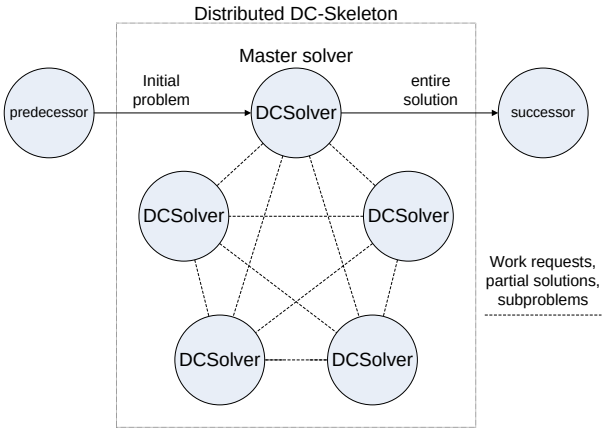

Fig. 3. A distributed design

2.2 A fully distributed D&C-Skeleton

Figure 3 illustrates the design of the distributed divide and conquer skeleton (DCSkeleton) provided by the *Muesli* skeleton library. It consists of a set of peer solvers, which exchange subproblems, partial solutions, and work requests. Several topologies for connecting the solvers are possible. In our implementation, 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 numbers of processors, topologies like torus or hypercube may reduce the communication overhead.

In the example shown in Fig. 3, $n = 5$ solvers are used. Each solver maintains its own local work pool and solution pool. Thus, the work and the solution pool are distributed among the solvers, which enables the skeleton to process D&C problems with much higher memory requirements compared to the ones that can be solved by a skeleton based on a master/worker design. Exactly one of the solvers, the *master solver*, serves as an interface to the DCSkeleton. The *master solver* receives new divide and conquer problems from the predecessor and delivers the solutions to its successor. The code fragment in Figure 4 illustrates the application of our DCSkeleton in the context of the *Muesli* skeleton library. It constructs the process topology shown in Fig. 3.

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 `generateProblem` method returning the initial D&C problem that is to be solved. The constructor `DistributedDC` generates $n = 5$ solvers, which are provided with the four basic operators `divide`, `combine`, `solveSeq`, and `isSimple`. The function `isSimple` has to return `true` if the subproblem size has reached the threshold T, which indicates that the subproblem can be solved sequentially with `solveSeq`. If only one solver is used, it is recommended to enable

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

Fig. 4. Example application using a distributed divide and conquer skeleton.

a purely sequential computation by setting T to the size of the initial problem. 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.

The DC-skeleton consumes a stream of input values and produces a stream of output values. If the *master solver* receives a new D&C problem, the communication with the predecessor is blocked until the received problem is solved. This ensures that the skeleton processes only one D&C problem at a time.

There are different variants for the initialization of the skeleton with the objective of providing each `DCSolver` 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.

Each solver repeatedly executes two overlapping phases: a communication phase and a computation phase. The communication phase includes work requests, delegating problems, and sending solutions. Let us first consider the case in which each solver works locally on the work and solution pool, and no communication due to load balancing issues is needed.

The work pool and the solution pool are implemented in their own classes in order to allow an easy replacement of the underlying data structures and algorithms for e.g. the traversal strategy. In our skeleton, the work pool is implemented as a double ended queue (DEQ). Local work is taken from and written to the head, whereas problems which are delegated within the load distribution are taken from the tail. Thus, the DEQ behaves as a stack for local computations. The solution pool is implemented as a sorted array. If there are only local computations (as it is the case most of the time), it behaves very efficiently as a stack. Thus, we have preferred this solution to a heap.

Subproblems and partial solutions are encapsulated in *Frames*, which are each identified by a unique identifier. Subproblems and their corresponding solutions are marked with the same *id*. The initial problem is marked with $id = 0$. If the solution pool contains a solution with $id = 0$, this indicates that the initial problem has been solved. This problem based termination detection is independent

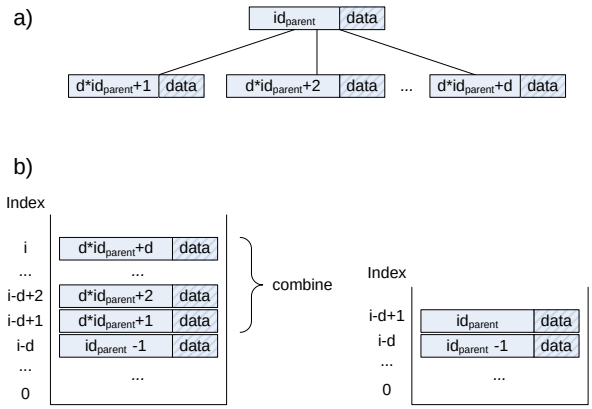

Fig. 5. a) A problem is divided into d subproblems; b) Solution pool represented as sorted array before (left) and after applying combine (right).

from the topology which is used for connecting the solvers. Moreover, it does not need any communication. The *id*s for parent nodes, and child nodes respectively, can for instance be deduced from the formula $parentNodeID = \frac{childNodeID-1}{d}$, with d equal to the degree of the divide and conquer tree (e.g. number of child nodes).

The solver takes and processes only one problem from the work pool per iteration. If a problem is divided by the solver, d new subproblems are generated which are marked with *id*s in ascending order (Fig.5a). These subproblems are successively inserted into the workpool again, starting with the subproblem tagged with the largest id. The subproblem marked with the lowest id is inserted last. Figure 6 illustrates the status of the workpool after the first four iterations with $d = 2$.

If a problem can be directly solved by the solver, the corresponding solution is written to the solution pool. The partial solutions stored in the solution pool are kept sorted by the ids in ascending order. Thus, whenever a solution is pushed to the solution pool, in a first step, the solution is written to the end of the sorted list, and then, a simple insertion sort is applied to preserve the order, if necessary.

Keeping the solution pool sorted enables a fast combination of partial solutions due to the fact that only two elements of the solution pool have to be inspected in order to detect if the top d elements can be combined. Assuming that a problem marked with the identifier id_{parent} is divided into d subproblems, as shown in Figure 5a, the leftmost child node is marked with $d \cdot id_{parent} + 1$, and the rightmost subproblem is marked with $d \cdot id_{parent} + d$. After the subproblems are solved, the solution pool contains solutions which are marked with the same *id*s than the subproblems. The solution of the most right child of id_{parent} is stored at index position i and represents the top element of the stack (Fig. 5b). The top d stack elements can be combined if they emanate from the same

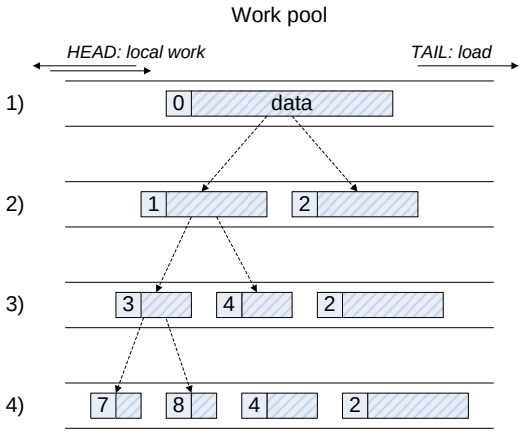

Fig. 6. States of the work pool

parent problem. This is the case iff the subproblem id at index position $i-d+1$ is $d-1$ less than the id at index i. Otherwise, one or more partial solutions are still missing, and `combine` can not be applied yet. If a combination is possible, the partial solutions are removed from, and the solution of the parent problem is written to the solution pool (Fig. 5b). This event triggers another inspection of the top d stack elements and a `combine` call, if applicable.

Let us now consider the communication phase. If several solvers are used, some load balancing mechanism is required. We have implemented a *random stealing* algorithm, which is provably efficient for divide and conquer algorithms in terms of space, time, and communication [11, 32]. In our case, a solver sends a work request to a randomly selected neighbour if its work pool is empty. If the neighbour has more than one subproblem in its work pool, it takes one (from the tail of the `DEQ`) and returns it to the requestor. Otherwise it sends a rejection message, which causes the requestor to ask another neighbour for some work to share. The advantage is that no communication is performed until a solver finds its own work pool empty, and the system behaves well under high loads. Moreover, problems which are taken from the tail of the `DEQ` are expected to be big since they stem from nodes of the upper levels of the divide and conquer tree. Thus, the receiver is supplied with a large amount of work. Obtained problems are processed locally in the same manner as described above. The corresponding solution must be sent back to the neighbour due to the fact that it is required there by `combine`. For this reason the solver records a pair $(problemID, neighbour)$ for each new problem which is received. If a new solution is combined whose id is equal to a previously recorded $problemID$, it is sent back to the corresponding $neighbour$, which stores it in the solution pool. Normally, this triggers an insertion sort call in order to restore the order by id. As a result, there may be combinable solutions deep in the stack, which are not combined immediately. Sooner or later, these solutions are inevitably combined

due to the implemented traversal strategy. However, in order to avoid idle times when the work pool is empty and new work is requested, a solver searches the solution pool for combinable solutions and combines them, if possible.

2.3 Optimizing the DCSkeleton for streams

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 [29]. 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.

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 7, 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.

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. 3) or a farm (fig. 7).

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 8 depicts the utilization of the DCSkeleton within the startup, main,

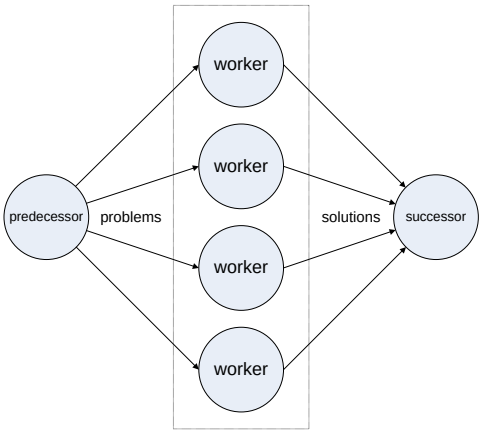

Fig. 7. A farm skeleton

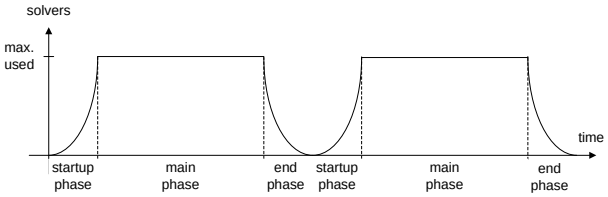

Fig. 8. utilization of the DCSkeleton

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.

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

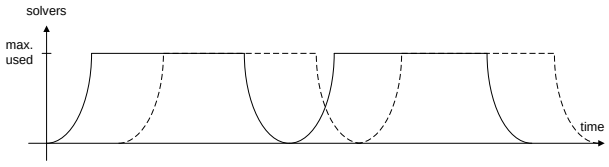

Fig. 9. Overlapped startup, main, and end phases

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 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 9, 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 10 illustrates the design of an optimized 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

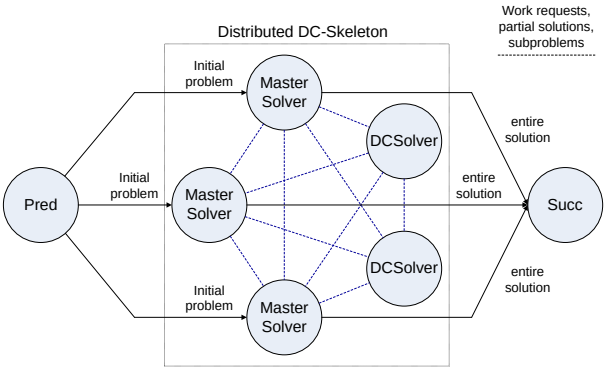

Fig. 10. A fully distributed divide and conquer skeleton for stream processing

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.

3 Applications and Experimental Results

The parallel test environment for our experiments is an IBM workstation cluster [33] of sixteen uniform PCs connected by a Myrinet [25]. 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.

#Solvers	Strassen	Karatsuba	Mergesort
1	295,94	43,52	43,19
2	146,52	22,10	28,25
3	98,13	14,94	24,78
4	73,93	11,41	22,79
5	59,86	9,32	21,91
6	50,58	7,89	21,04
7	42,85	6,92	20,60
8	38,53	6,16	20,03
9	34,92	5,58	19,95
10	32,10	5,14	19,79
11	29,82	4,76	19,58
12	27,72	4,47	19,42
13	26,28	4,18	19,27
14	24,79	3,96	19,15

Table 1. Runtimes for Strassen, Karatsuba, and Mergesort (in seconds).

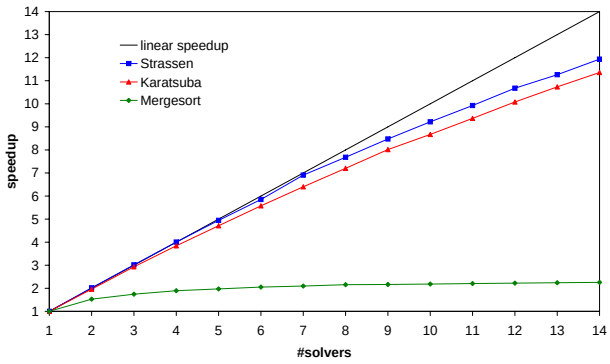

Fig. 11. Speedup for the Mergesort, Karatsuba, and Strassen algorithm.

In order to evaluate the performance and scalability of the distributed divide and conquer skeleton, we have considered three problems with different complexity classes. At first, we have implemented the standard mergesort algorithm [19], which is in $O(N \log N)$, to sort randomly generated integer arrays of size $N = 2^{26} = 67108864$. Moreover, we have implemented the Karatsuba multiplication algorithm for big integers ($O(N^{\log_2 3})$, where $\log_2 3 \approx 1,58$). The Karatsuba algorithm [20] reduces a multiplication of two N-digit integers to three multiplications of $\frac{N}{2}$-digit integers. In our experiments we have generated two numbers with $2^{20} = 1048576$ digits for each test run. Finally, we have implemented 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 4096×4096 integer matrices. The Strassen algorithm [31] reduces a multiplication of two $N \times N$ matrices to seven multiplications of $\frac{N}{2} \times \frac{N}{2}$ matrices. The algorithms differ not only in their complexity classes, but also in the dynamically generated divide and conquer tree, which is of degree $d = 2$ for mergesort, $d = 3$ for Karatsuba, and $d = 7$ for Strassen. Note that the skeleton behaves non-deterministically in the way the load is distributed. Generating only a few big subproblems can lead

a) Mergesort	solver			
	1	2	3	4
processed problems	8193	8192	8191	8191
solved problems	4096	4096	4096	4096
distributed problems	2	1	0	0
received problems	0	1	1	1
# work requests	5	9	17	26
time for solve	4,24	4,24	4,28	4,29
time for combine	4,65	3,57	2,98	2,98
time for divide	1,79	1,39	1,10	1,10
Σ time	10,68	9,19	8,36	8,37

b) Karatsuba	solver							
	1	2	3	4	5	6	7	8
processed problems	3580	3707	3737	3664	3704	3727	3671	3734
solved problems	2466	2469	2490	2492	2446	2443	2391	2486
distributed problems	18	16	10	10	16	15	4	7
received problems	7	9	12	12	12	16	17	11
# work requests	43	57	60	56	95	95	98	58
time for solve	5,19	5,20	5,25	5,24	5,14	5,14	5,24	5,20
time for combine	0,24	0,19	0,18	0,16	0,16	0,15	0,15	0,16
time for divide	0,22	0,18	0,19	0,17	0,18	0,16	0,15	0,17
Σ time	5,65	5,57	5,63	5,58	5,48	5,46	5,54	5,53

c) Strassen	solver							
	1	2	3	4	5	6	7	8
processed problems	2450	2482	2549	2410	2415	2364	2477	2461
solved problems	2100	2128	2184	2123	2109	2065	2027	2071
distributed problems	11	8	7	13	15	18	11	5
received problems	10	12	1	12	12	13	16	12
# work requests	79	88	26	83	73	81	84	74
time for solve	29,56	30,67	30,93	30,22	29,82	29,53	29,76	29,76
time for combine	0,88	0,66	0,65	0,62	0,62	0,60	0,59	0,53
time for divide	2,75	2,01	2,13	2,03	2,03	2,00	2,02	1,84
Σ time	33,19	33,34	33,71	32,87	32,47	32,14	32,37	32,13

Table 2. Distribution of problems, work requests, and computation time for the Mergesort, Karatsuba, and Strassen algorithm

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 Table 1. Figure 11 depicts the corresponding speedups.

Table 2 shows for a typical run the number of subproblems, which are distributed and received by the solvers, as well as the number of work requests, which have been sent by each solver. Moreover, the table shows the number of subproblems, which are processed locally by each solver, as well as the number of simple problems emanating from them, which are solved sequentially. As one can see, only few work requests have been sent. In our skeleton, the *master solver* undertakes the task of dividing the initial problem and combining the entire solution. At this time, all other solvers are idle. Thus, most of the work requests were sent within the startup and the end phase. However, despite of the low number of work requests and distributed subproblems, we noticed a well-balanced work distribution in all considered example applications. This is due to the fact that each solver fetches most problems from its own workpool, such that they require no communication. This is essential for achieving good runtimes and speedups. Note that this not only applies to divide and conquer but also to other skeletons with a similar characteristic such as branch and bound and other search skeletons [27].

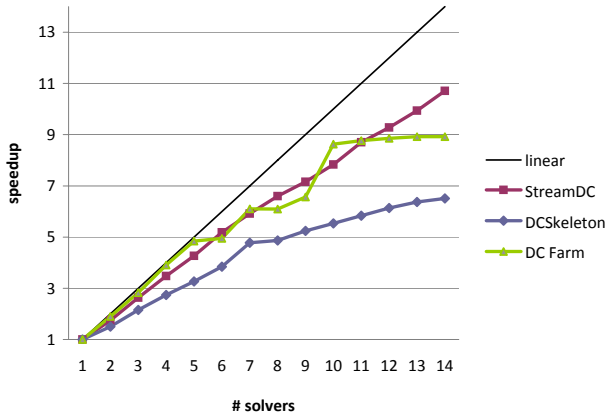

Fig. 12. Speedups for StreamDC, DCSkeleton and a sequential farm processing matrix multiplication problems

As one would expect, the skeleton reaches the lowest speedups for Mergesort. In this case, combining the entire solution takes a good deal of the overall computation time, which cannot be distributed among the solvers. This is supported by Table 2a, which shows the time for `divide`, `solve`, and `combine` consumed by the solvers in case of an equal distribution of the subproblems among the solvers. While the time for solving the subproblems is identically for all solvers, the *master solver* (solver 1) shows clearly higher computation times for `combine` and `divide`. Moreover, solving a problem locally is often faster than delegating it to a solver, because sending and receiving subproblems causes high communication costs. In our case, the best runtimes can be achieved if the threshold T for $6 \leq p \leq 14$ processors is chosen to be $T = 262144$ leading to 256 subproblems. For the Karatsuba and Strassen algorithms, the speedups are clearly better than for Mergesort, because the relation of computation time to communication time is significantly better (i.e. higher).

In order to evaluate the performance and scalability of the StreamDC skeleton, we have considered Strassen's algorithm for matrix multiplication in order to multiply two randomly generated 1024×1024 integer matrices. The stream consists of 20 matrix multiplication problems, which represents single matrix multiplications when solving a matrix chain multiplication problem $A_1 \cdot \ldots \cdot A_n$ [2, 16, 17]. Figure 12 depicts the corresponding speedups for the StreamDC, the DCSkeleton and the farm of sequential DCSkeletons. 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 is caused by a bad load balance due to the fact that the number of problems in the stream is only a little higher than the number of solvers. 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.

4 Related work

Some related work on algorithmic skeletons for divide and conquer can be found in the literature. Recent skeleton libraries such as eSkel [9], skeTo [23], and MaLLBba [1, 12] include skeletons for divide and conquer. The MaLLBa implementation of the divide and conquer skeleton presented in [1] is based on a farm (master-slave) strategy. The distributed approach discussed in [12] offers the same user interface as the MaLLBa skeleton and can be integrated into the MaLLBa framework. The implementation differs considerably from our approach. The work and the solution pool are represented as a pointer based tree structure. Additionally, a queue with the nodes waiting to be explored is kept. Moreover, a global load balancing algorithm has been implemented, which causes high message traffic and requires a complicated communication protocol to cope with starvation problems. Unfortunately, no runtimes of the considered example applications are presented. In [8], Cole suggests to offer `divide` and `combine` as independent skeletons. But this approach has not been implemented in eSkel. The eSkel *Butterfly*-Skeleton [9] 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 due to the group partitioning strategy. The skeTo library [23] only provides data parallel skeletons and is based on the theory of *Constructive Algorithmics*. Restricted data parallel approaches are discussed in [4, 13]. In [13], 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 *Haskel* [14]. A distributed divide and conquer scheme is not considered there. A mixed data and task parallel approach can be found in [6].

5 Conclusions

We have considered two implementation schemes for divide and conquer skeletons. After briefly analyzing a centralized master/worker scheme as used in MaLLBa [1], we have focused on a distributed scheme. Important issues have been the demand-driven work-distribution scheme as well as an efficient approach to the combination of available partial solutions using a sorted array. Based on experimental results for Mergesort, the Karatsuba algorithm, and Strassen's algorithm, we have shown that our approach leads to good runtimes and speedups, and that it minimizes the communication overhead. Only very few problems need

to be exchanged between the different 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

1. E.Alba, F.Almeida, et al.: "MALLBA: A library of Skeletons for combinatorial optimisation", in Euro-Par'02, LNCS 2400, pages 927-932, Springer, 2002.
2. G. Baumgartner: "A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry", In Proceedings of Supercomputing, 2002.
3. A.Benoit, M.Cole, S.Gilmore, J.Hillston: "Flexible Skeletal Programming with eSkel", in Proceedings of Euro-Par'05, LNCS 3648, pages 761-770, Springer, 2005.
4. H.Bischof: "Systematic Development of Parallel Programs Using Skeletons", PhD thesis, Shaker, 2005.
5. G.H.Botorog, H.Kuchen: "Efficient Parallel Programming with Algorithmic Skeletons", in Proceedings of Euro-Par'96, LNCS 1123, pages 718-731, Springer, 1996.
6. Y.Bai, R.Ward: "A Parallel Symmetric Block-Tridiagonal Divide-and-Conquer Algorithm", ACM Transactions on Mathematical Software, Vol. 33, No. 4, Article 25, 2007.
7. M.Cole: "Algorithmic Skeletons: Structured Management of Parallel Computation", Pitman/MIT Press, 1989.
8. M.Cole: "On Dividing and Conquering Independently", in Proceedings of Euro-Par'97, LNCS 1300, pages 634-637, Springer, 1997.
9. M.Cole: "Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. In Parallel Computing 30(3), pages 389-406, 2004.
10. J.Darlington, Y.Guo, H.To, 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.
11. M.Eriksson, C.Kessler, M.Chalabine: "Load Balancing of Irregular Parallel Divide-and-Conquer Algorithms in Group-SPMD Programming Environments", in proceedings of ARCS'06 Workshop on Parallel Systems and Algorithms (PASA'06), Frankfurt, Germany, 2006.

12. J.R.González, C.León, C.Rodríguez: "A Distributed Parallel Divide and Conquer Skeleton", PARA'04, 2004.
13. S.Gorlatch: "N-graphs: scalable topology and design of balanced divide-and-conquer algorithms", Parallel Computing, 23(6), pages 687-698, 1997.
14. C.A.Herrmann: "The Skeleton-Based Parallelization of Divide-and-Conquer Recursions", PhD thesis, Logos-Verlag, 2000.
15. D.Henrich: "Initialization of parallel branch-and-bound algorithms", In proceedings of the 2nd International Workshop on Parallel Processing for Artificial Intelligence (PPAI-93), Elsevier, 1994.
16. T.C. Hu, M.T. Shing: "Computation of matrix chain products I", SIAM Journal on Computing, 11(2):362-373, 1982.
17. T.C. Hu, M.T. Shing: "Computation of matrix chain products II", SIAM Journal on Computing, 13(2):228-251, 1984.
18. H.Kuchen, M.Cole: The Integration of Task and Data Parallel Skeletons, Parallel Processing Letters 12(2), pages 141-155, 2002.
19. D.E.Knuth: "The Art of Computer Programming, Vol. 3: Sorting and Searching", Addison-Wesley, 1973.
20. A.Karatsuba, Y.Ofman: "Multiplikation of multidigit numbers on automata" Doklady Akademii Nauk SSSR, 145(2):293-294, 1962.
21. H.Kuchen: "A Skeleton Library", in Euro-Par'02, LNCS 2400, pages 620-629, Springer, 2002.
22. H.Kuchen: "Optimizing Sequences of Skeleton Calls", in Domain-Spezific Program Generation, LNCS 3016, pages 254-273, Springer, 2004.
23. K.Matsuzaki, K.Emoto, H.Iwasaki, 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.
24. Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
25. The Myricom homepage. http://myri.com/.
26. S.Pelagatti: "Task and Data Parallelism in P3L", in Patterns and Skeletons for Parallel and Distributed Computing, eds. F.A.Rabhi and S.Gorlatch, pages 155-186, Springer, 2003.
27. M.Poldner, H.Kuchen: "Algorithmic Skeletons for Branch & Bound", in proceedings of 1st International Conference on Software and Data Technology (ICSOFT), Vol. 1, pages 291-300, Setubal, Protugal, 2006.
28. M.Poldner, H.Kuchen: "On Implementing the Farm Skeleton", Parallel Processing Letters, Vol. 18, No. 1, pages 117-131, 2008.
29. M.Poldner, H.Kuchen: "Skeletons for Divide and Conquer Algorithms", Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN), ACTA Press, 2008.
30. M.Poldner, H.Kuchen: "Optimizing Skeletal Stream Processing for Divide and Conquer", Proceedings of the 3rd International Conference on Software and Data Technologies (ICSOFT), pages 181-189, INSTICC Press, 2008.
31. V.Strassen: "Gaussian Elimination is not optimal", Numerische Mathematik, 13:354-356, 1969.
32. R.van Nieuwpoort, T.Kielmann, H.Bal: "Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications", Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming PPoPP '01, Vol.36, Issue 7, 2001.
33. Ziv-Cluster. http://zivcluster.uni-muenster.de/.