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.

KEY WORDS
Algorithmic Skeletons, parallelism, divide and conquer

1 Introduction
Parallel programming of MIMD machines with distributed memory is error-prone and time consuming. Typically, this is due to the fact that commonly implemented techniques are based on message passing which leads to a comparatively low programming 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, 6, 9, 15, 18, 20, 23]. By providing application-specific parameters to these skeletons, the user can adapt them to the considered parallel application without bothering about low-level implementation details such as synchronization, interprocessor communication, load balancing and data distribution. Efficient implementations of many skeletons exist, such that the resulting parallel application can be almost as efficient as one based on low-level message passing. Algorithmic skeletons can roughly be classified into data parallel and task parallel ones. Data parallel skeletons [4, 18, 19] 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 [2, 8, 15, 18, 24, 25] 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. Simple problems are solved directly without dividing them further. Examples of divide and conquer computations include various sorting methods such as mergesort and quicksort, computational geometry algorithms such as the construction of the convex hull, combinatorial search such as constraint satisfaction techniques, graph algorithmic problems such as graph coloring, numerical methods such as the Karatsuba multiplication algorithm, and linear algebra such as Strassen’s algorithm for matrix multiplication.

In the present paper we will consider different implementation and design issues of task parallel divide and conquer skeletons in the context of the skeleton library Muesli [18, 24, 25]. Muesli is based on MPI [21] internally 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.

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 DC-Skeleton. Section 3 contains experimental results demonstrating the strength of the distributed design. In
Problem $N$

Sub $T$

Solution

... ... ...

work

pool

solution 

pool

successor

by the solution of the corresponding parent problem. In the

initial solutions, which can be combined, they can be replaced

divide

provide the skeleton with four basic operators: divide, combine, isSimple, and solve. If isSimple indi-
cates that a problem is simple enough, it can be solved
directly by applying solve. Otherwise, the problem is di-
vided into subproblems by calling divide. Solutions of
subproblems can be combined to the solution of the corre-
sponding 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). Un-
processed subproblems are stored in a work pool and partial
solutions are maintained in a solution pool. In the begin-
ning the work pool only contains the initial problem, which
is of size $N$, and the solution pool is empty. In each iter-
ation one such problem is selected from the workpool cor-
responding 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 $d$ sub-
problems 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 mem-
ory problems.

The order in which solutions are stored in the solution pool
depends on the implemented traversal strategy. It is recom-
mended to combine partial solutions as soon as possible in
order to free memory. If the solution pool contains $d$ par-
tial solutions, which can be combined, they can be replaced
by the solution of the corresponding parent problem. In the

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 implementa-
tion 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 indi-

cates that a problem is simple enough, it can be solved
directly by applying solve. Otherwise, the problem is di-

vided into subproblems by calling divide. Solutions of

subproblems can be combined to the solution of the corre-
sponding 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). Un-

processed subproblems are stored in a work pool and partial

solutions are maintained in a solution pool. In the begin-

ning the work pool only contains the initial problem, which

is of size $N$, and the solution pool is empty. In each iter-

ation one such problem is selected from the workpool cor-

responding 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 $d$ sub-

problems 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 mem-

ory problems.

The order in which solutions are stored in the solution pool
depends on the implemented traversal strategy. It is recom-
mended to combine partial solutions as soon as possible in
order to free memory. If the solution pool contains $d$ par-
tial 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 solu-
tion pool only contains the solution of the initial problem.

2.1 Adjusting the number of generated problems

Dividing problems and maintaining them in a workpool in-

troduces overhead. Thus, it appears reasonable to solve

subproblems locally by calling a sequential (divide and

conquer) algorithm at a time when the subproblem sizes

have reached a certain threshold $T$. Unfortunately, the spe-
cific value of $T$ is problem dependent and therefore hard to
predict. If $T$ is large, only few big subproblems are gener-
ated and distributed among the processors, which can lead
to an unbalanced load distribution. Thus, the threshold $T$
has to be chosen small enough such that a sufficient num-
ber of subproblems is generated to ensure a good load bal-
ancing. Dividing a problem causes costs for subproblem
generation, combination, and maintenance. Thus, $T$ has to
limit the total number of generated subproblems. In either
case, $T$ determines the depth of the divide and conquer tree
and the number of leaf nodes stored in the work pool, i.e.
subproblems, which are solved sequentially. The impact of
the choice of $T$ will be investigated in Section 3.

2.2 Master/Worker design

The simplest approach to implement a divide and conquer
skelton 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 mas-
ter, which distributes problems to the workers and receives
subproblems and solutions from them. When a worker re-
ceives 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