ON IMPLEMENTING THE FARM SKELETON

MICHAEL POLDNER and HERBERT KUCHEN
Department of Information Systems, University of Münster, D-48149 Münster, Germany

ABSTRACT
Algorithmic skeletons intend to simplify parallel programming by providing a higher level of abstraction compared to the usual message passing. Task and data parallel skeletons can be distinguished. In the present paper, we will consider several approaches to implement one of the most classical task parallel skeletons, namely the farm, and compare them w.r.t. scalability, overhead, potential bottlenecks, and load balancing. We will also investigate several communication modes for the implementation of skeletons. Based on experimental results, the advantages and disadvantages of the different approaches are shown. Moreover, we will show how to terminate the system of processes properly.

Keywords: Parallel programming, algorithmic skeletons, farm, MPI.

1. Introduction
Today, parallel programming of MIMD machines with distributed memory is typically based on message passing. Owing to the availability of standard message passing libraries such as MPI\textsuperscript{a} [10], the resulting software is platform independent and efficient. However, the programming level is still rather low and programmers have to fight against low-level communication problems such as deadlocks. Moreover, the program is split into a set of processes which are assigned to the different processors. Like an ant, each process only has a local view of the overall activity. A 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.

Many approaches try to increase the level of parallel programming and to overcome the mentioned disadvantages. Here, we will focus on algorithmic skeletons, i.e. typical parallel-programming patterns which are efficiently implemented on the available parallel machine and usually offered to the user as higher-order functions, which get the details of the specific application problem as argument functions (see e.g. [3,4,19]). The skeletal parallelism homepage [6] contains links to virtually all groups and projects working on skeletons.

In our framework, a parallel computation consists of a sequence of calls to skele-

\textsuperscript{a}We assume some familiarity with MPI and C++.
tons. Several implementations of algorithmic skeletons are available. They differ in the kind of host language used and in the particular set of skeletons offered. Since higher-order functions are taken from functional languages, many approaches use such a language as host language [7,12,20]. In order to increase the efficiency, imperative languages such as C and C++ have been extended by skeletons, too [2,3,9,19].

Depending on the kind of parallelism used, skeletons can be classified into task parallel and data parallel ones. In the first case, a skeleton (dynamically) creates a system of communicating processes by nesting predefined process topologies such as pipeline, farm, parallel composition, divide&conquer, and branch&bound [1,4,5,7,11,19]. In the second case, a skeleton works on a distributed data structure, performing the same operations on some or all elements of this data structure. Data-parallel skeletons, such as map, fold or rotate are used in [2,3,7,8,12,19].

Moreover, there are implementations offering skeletons as a library rather than as part of a new programming language [5,14] The approach described in the sequel is based on the skeleton library introduced in [11,13,14] and on the corresponding C++ language binding. Skeletons can be understood as domain-specific languages for parallel programming.

Our library provides task as well as data parallel skeletons, which can be combined based on the two-tier model taken from P³L [19]. In general, a computation consists of nested task parallel constructs where an atomic task parallel computation can be sequential or data parallel. Purely data parallel and purely task parallel computations are special cases of this model. An advantage of the C++ binding is that the three important features needed for skeletons, namely higher-order functions (i.e. functions having functions as arguments), partial applications (i.e. the possibility to apply a function to less arguments than it needs and to supply the missing arguments later), and parametric polymorphism, can be implemented elegantly and efficiently in C++ using operator overloading and templates, respectively [13,21].

In the present paper, we will focus on task-parallel skeletons in general and on the well-known farm skeleton in particular. Conceptually, a farm consists of a farmer and several workers. The farmer accepts a sequence of tasks from some predecessor process and propagates each task to a worker. The worker executes the task and delivers the result back to the farmer who propagates it to some successor process (which may be the same as the predecessor). This specification suggests a straightforward implementation leading to the process topology depicted in Fig. 1. The problem with this simple approach is that the farmer may become a bottleneck, if the number of workers is large. Another disadvantage is the overhead caused by the propagation of messages. Consequently, it is worth considering different implementation schemes avoiding these disadvantages. In the present paper, we will consider a variant of the classical farm where the farmer is divided into a dispatcher and a collector as well as variants where these building blocks have (partly) been omitted.

Moreover, we investigate which MPI communication modes are most appropriate for implementing skeletons. Finally, we will present a scheme to shutdown the system of processes cleanly, even for nested and possibly cyclic process topologies.

The rest of the paper is organized as follows. In Section 2, we show how task-