Scalable Farms

Michael Poldner [a], Herbert Kuchen [a]

[a]University of Münster, Department of Information Systems, Leonardo Campus 3,
D-48159 Münster, Germany

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 skeleton, namely the farm, and compare them w.r.t. scalability, overhead, potential bottlenecks, and load balancing. Based on experimental results, the advantages and disadvantages of the different approaches are shown.

1. Introduction

Today, parallel programming of MIMD machines with distributed memory is typically based on message passing. Owing to the availability of standard message passing libraries such as MPI [1] [11], 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. [3,4,9]). [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 skeletons. 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 [7,13,18]. In order to increase the efficiency, imperative languages such as C and C++ have been extended by skeletons, too [2,3,9,10].

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` [1,4,5,7,9,12]. 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 [2,3,7–9,13].

Moreover, there are implementations offering skeletons as a library rather than as part of a new programming language. The approach described in the sequel is based on the skeleton library intro-

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

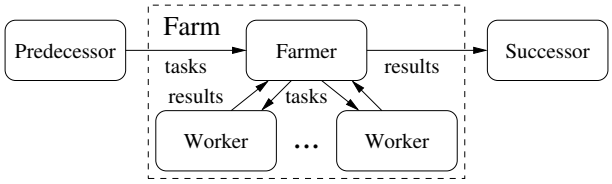


Figure 1. Farm.

duced in [12,14,15] 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 P^{3}L [9]. 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 [14,16,19].

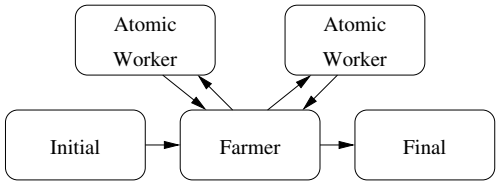
In the present paper, we will focus on task-parallel skeletons in general and on the well-known *farm* skeleton in particular. Conceptually, a farm consists of a *farmer* and several *workers*. The farmer accepts a sequence of tasks from some predecessor process and propagates each task to a worker. 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.

The rest of the paper is organized as follows. In Section 2, we show how task-parallel applications can be implemented using the task-parallel skeletons provided by our skeleton library. The considered variants of the farm skeleton are presented in Section 3. Section 4 contains experimental results for the different approaches. Finally, in Section 5 we conclude and discuss related work. Moreover, we point out future extensions.

2. Task-parallel Skeletons

Our skeleton library offers data parallel and task parallel skeletons. Task parallelism is established by setting up a system of processes which communicate via streams of data. Such a system is not arbitrarily structured but constructed by nesting predefined process topologies such as farms and pipelines. Moreover, there are skeletons for parallel composition, branch & bound, and divide & conquer. Finally, it is possible to nest task and data parallelism according to the mentioned two-tier model of P^{3}L, which allows atomic task parallel processes to use data parallelism inside [9]. Here, we will focus on task-parallel skeletons in general and on the farm skeleton in particular.

In a farm, a farmer process accepts a sequence of inputs and assigns each of them to one of several




```
#include "Skeleton.h"

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

int* init() { if (current++ < 100000) return &current;
            else return NULL;}

int add(int x, int y) { return x + y; }

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

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

    // step 1: create a process topology (using C++ constructors)
    Initial<int>     initial(init);
    Atomic<int,int>  atomicWorker((curry(add)(i),1));
    Farm<int,int>    farm(atomicWorker,numOfWorkers);
    Final<int>       final(fin);
    Pipe             pipeline(initial,farm,final);

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

    TerminateSkeletons();}
```

Figure 2. Task parallel example application.

workers. The parallel composition works similar to the farm. However, each input is forwarded to every worker. A pipeline allows tasks to be processed by a sequence of stages. Each stage handles one task at a time, and it does this in parallel to all the other stages.

Each task parallel skeleton has the same property as an atomic process, namely it accepts a sequence of inputs and produces a sequence of outputs. This allows the task parallel skeletons to be arbitrarily nested. Task parallel skeletons like pipeline and farm are provided by many skeleton systems, see e.g. [5,9].

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

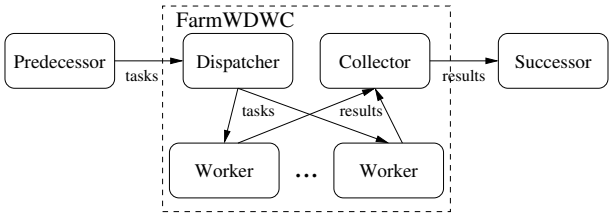


Figure 3. Farm with separate dispatcher and collector.

3. The Farm Skeleton

As pointed out in the introduction, a straightforward implementation of the farm skeleton could be based on the process topology depicted in Fig. 1. It has the advantage that the farmer knows which workers have returned the results of their tasks and are hence idle. Thus, the farmer can forward incoming tasks to the idle workers. However, this approach has the disadvantage that it causes substantial overhead due to the messages which have to be exchanged between farmer and workers. Moreover, the farmer might become a bottleneck, if the number of workers is large. In this case, the farmer will not be able to keep all workers busy, leading to wasted workers. The amount of workers which the farmer can keep busy depends on the sizes of the tasks and the sizes of the messages the farmer has to propagate. For small tasks and large messages, only very few workers can be kept busy.

If on the other hand, there are few workers (with large tasks), the farmer will partly be idle. This problem can be solved by mapping a worker to the same processor as the farmer. Thus, we will not consider this problem further. In general, the aim is to keep all processors as busy as possible and to avoid a waste of resources.

Let us point out that implementing this apparently simple farm skeleton is not as easy as it might seem. The interested reader may have a look at our implementation [17]. Firstly, the process topology is obviously cyclic (see Fig. 1). Thus, one has to be very careful to avoid deadlocks. On the other hand, one has to make sure that the farmer reacts as quickly as possible on newly arriving tasks and on workers delivering their results. For an implementation based on MPI, this means that the simpler synchronous communication has to be replaced by the significantly more complex non-blocking asynchronous communication using `MPI_Wait`. Moreover, special control messages are needed besides data messages in order to stop the computation cleanly at the end. This leads to a quite complicated protocol, which has to be supported by every skeleton. Thus, an obvious advantage of using skeletons is that the user does not have to invent the wheel again and again, but can readily use the typical patterns for parallel programming without worrying about deadlocks, stopping the computation cleanly, or overlapping computation and communication properly.

3.1. Farm with Dispatcher and Collector

A first approach to reduce the load of an overloaded farmer is to split it into a *dispatcher* of work and an independent *collector* of results as depicted in Fig. 3. This variant is implemented in P^3L [9].

In case that distributing tasks needs as much time as collecting results, the farmer can serve twice as many workers as with the previous approach. If, however, distributing tasks is much more work than collecting results or vice versa, little has been gained, since now the dispatcher or the collector will quickly become the bottleneck, while the other will partly be idle. The amount of required messages is unchanged and the corresponding overhead is hence preserved. A disadvantage of this farm variant is that the dispatcher now has no knowledge about the actual load of each worker. Thus,

he has to divide work based on some load independent scheme, e.g. cyclically or by random selection of a worker. Both may lead to an unbalanced distribution of work. However for a large number of tasks, one can expect that the load is roughly balanced. The blind distribution of work could be avoided by sending work requests from idle workers to the dispatcher. But then the dispatcher would have to process as many messages as the farmer in the original approach, and the introduction of a separate collector would be rather useless.

3.2. Farm with Dispatcher

The previous approach can be improved by omitting the collector and by sending results directly from the workers to the successor of the farm in the overall process topology (see Fig. 4). This eliminates the overhead of propagating results. Moreover, the omitted collector can no longer be a potential bottleneck.[2]

3.3. Farm without Dispatcher and Collector

After omitting the collector, one can consider omitting the dispatcher as well (see Fig. 5). In fact, this is possible, provided that the predecessor(s) of the farm in the overall process topology assume(s) the responsibility to distribute its/their tasks directly to the workers. As in the previous subsections, this distribution has to be performed based on a load independent scheme, e.g. cyclically or by random selection.

This approach reduces the overhead for the propagation of messages completely. Moreover, it omits another potential source of a bottleneck, namely the dispatcher. Of course, this new variant of the farm skeleton can be arbitrarily nested with other skeletons. For instance, it is possible to construct a pipeline of the such farms, as depicted in Fig. 5. In such a situation where n workers of the first farm communicate with m workers of the second, it is important to ensure that not all of them start with the same destination. If using a cyclic distribution scheme, worker i could e.g. assign its first task to worker $\lfloor i/m \rfloor$ of the second farm. If the destination is randomly chosen, it has to be ensured that all the random number generators start with a different initial value.

A farm without dispatcher but with collector does not seem to make sense, and it is not considered here.

4. Experimental Results

We have tested the different variants of the farm skeleton for small and big tasks as well as for tasks of variable sizes. A small task simply consists of computing the square of the input, a big task performs two million additions, and the tasks of variable sizes execute $n!$ iterations for some $1 \leq n \leq 10$.

[2] It may however happen that the successor now gets a bottleneck. In this case, the overall process topology needs to be adapted.

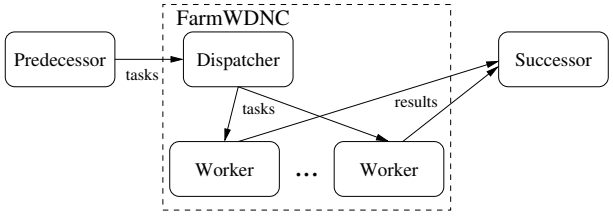


Figure 4. Farm with dispatcher; no collector is used.

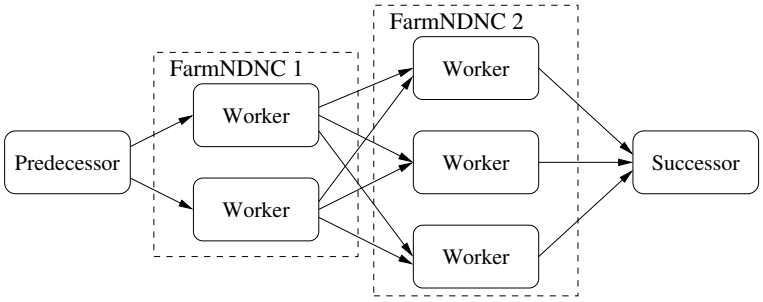


Figure 5. Pipeline of two farms without dispatcher and collector.

The experiments have been carried out on the IBM cluster at the University of Münster. This machine [20] has 94 compute nodes, each with a 2.4 GHz Intel Xeon processor, 512 KB L2 Cache, and 1 GB memory. The nodes are connected by a Myrinet and running RedHat Linux 7.3 and the MPICH-gm implementation of MPI.

For large tasks, all variants are able to keep the workers busy, as one would expect. Thus, all variants need roughly the same amount of time to execute the tasks, as can be seen in Fig. 6 a). However, just comparing the runtimes is not fair, since the different farms need different numbers of processors (unless farmer, dispatcher and collector share the same processor with one worker as mentioned above).Taking this into account, we see that the farmNDNC is the best, followed by the farmWDNC (see Fig. 6 b).

When considering tasks with (strongly) varying sizes (Fig. 7), we note that all approaches are able to keep a small number of workers (here up to 4) busy. If we add more workers, all approaches reach a point where they are no longer able to employ the additional workers. Beyond this point the runtime remains constant, independently of the number of additional workers. As expected, the original farm reaches this situation more quickly than the farmWDWC, the farmWDNC, and the farmNDNC, in this order. Moreover, farms which produce less overhead need less runtime when reaching the limit. When taking into account the number of processors used (rather than the number of workers), the advantages of the farmWDNC and, in particular, the farmNDNC become even more apparent (see Fig. 7 b). Interestingly, the behavior of the different farm variants does not depend significantly on the distribution scheme. Cyclic distribution and random distribution lead to almost identical runtimes. This is due to the fact that the number of tasks was large and the load of the workers has hence been balanced over time (Fig. 7 b). For small numbers of tasks the behavior may depend heavily on the actual mapping of tasks to workers.

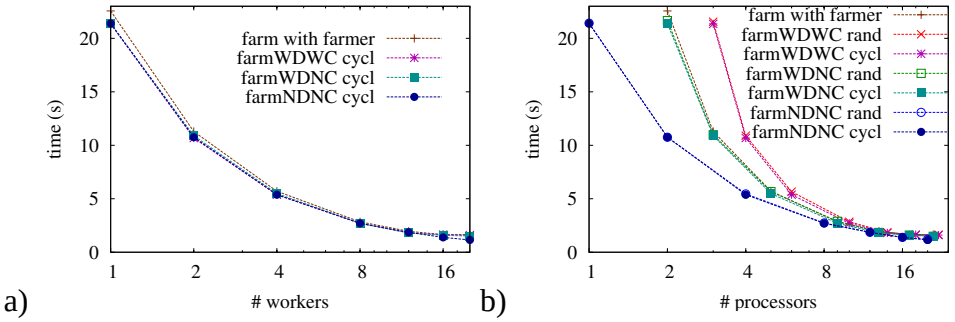


Figure 6. Farm variants with big tasks.

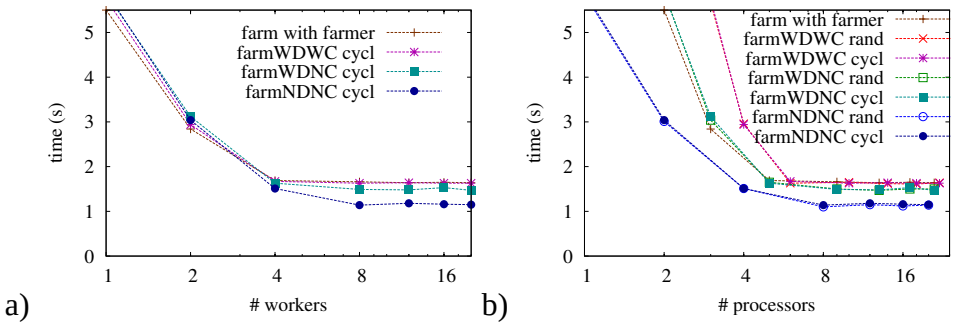


Figure 7. Farm variants with tasks of variable sizes.

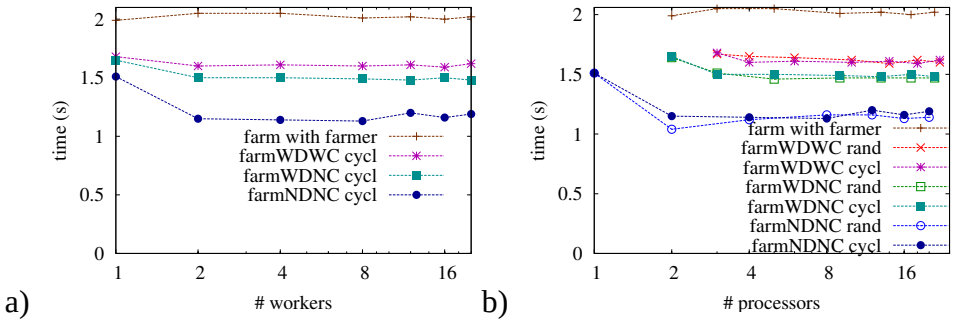


Figure 8. Farm variants with small tasks.

It may be surprising that even the farmNDNC is not able to keep an arbitrary amount of workers busy, although it has no farmer or dispatcher which might get a bottleneck. The reason is that the predecessor(s) of the farm are now responsible for the distribution of tasks to the workers. Unless the predecessor is itself a large farm, it will eventually become a bottleneck as observed in Fig. 7.

For small tasks, it is not worthwhile to employ any worker. The process delivering the small tasks should better execute them itself. If we nevertheless use a farm, it is clear that it is not possible to keep even a single worker busy. All farm variants require hence a more or less constant runtime independent of the number of workers (see Fig. 8). This situation corresponds to the one in the previous experiment when reaching the limit of useful workers. The roughly constant runtimes are not the same for all the skeletons but depend on the overhead caused by the considered variant.

5. Conclusions, Related Work, and Future Extensions

We have considered alternative implementation schemes for the well-known farm skeleton. Besides the classical approach where a farmer distributes work and collects results, we have considered variants where the farmer has been divided into a dispatcher and collector. Moreover, we have investigated variants where the collector and dispatcher have (partly) been omitted. In case of the variant without dispatcher, the predecessor of the farm in the overall process topology is responsible for the distribution of tasks to the workers. As our experimental results and our analysis show, the farm only consisting of workers is the best in terms of scalability and low overhead. It is clearly superior to the farms with dispatcher (and possibly collector). For a large number of tasks, it is also better than the classical farm, where a farmer distributes work. For a very small number of tasks, the classical farm might have advantages in some cases, since it is the only one which takes the actual load of workers

into account when distributing work.

We are not aware of any previous systematic analysis of different implementation schemes for farms in the literature. In the different skeleton projects, typically one technique has been chosen. In the first version of our skeleton library, there was just the classical farm. P^3L [9] uses the farmWDWC while eSkel [5] uses the classical farm with farmer.

As future work, we intend to investigate alternative implementation schemes for other skeletons, e.g. divide & conquer and branch & bound.

References

[1] Alba, E., Almeida, F., et al.: MALLBA: A Library of Skeletons for Combinatorial Search. Euro-Par'02. LNCS 2400, 927-932. Springer Verlag. 2002.

[2] Botorog, G.H., Kuchen, H.: Efficient Parallel Programming with Algorithmic Skeletons. In Bougé, L. et al., eds.: Euro-Par'96. LNCS 1123, 718-731. Springer-Verlag. 1996.

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

[4] Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press. 1989.

[5] Cole, M.: Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. Parallel Computing 30(3), 389-406. 2004.

[6] Cole, M.: The Skeletal Parallelism Web Page. http://homepages.inf.ed.ac.uk/mic/Skeletons/.

[7] Darlington, J., Field, A.J., Harrison T.G., et al: Parallel Programming Using Skeleton Functions. PARLE'93. LNCS 694, 146-160. Springer-Verlag. 1993.

[8] Darlington, J., Guo, Y., To, H.W., Yang, J.: Functional Skeletons for Parallel Coordination. In Hardidi, S., Magnusson, P.: Euro-Par'95. LNCS 966, 55-66. Springer-Verlag. 1995.

[9] Danelutto, M., Pasqualetti, F., Pelagatti S.: Skeletons for Data Parallelism in P^3L. In Lengauer, C., Griebl, M., Gorlatch, S.: Euro-Par'97. LNCS 1300, 619-628. Springer-Verlag. 1997.

[10] Foster, I., Olson, R., Tuecke, S.: Productive Parallel Programming: The PCN Approach. Scientific Programming 1(1) 51-66. 1992.

[11] Gropp, W., Lusk, E., Skjellum, A.: Using MPI. MIT Press. 1999.

[12] Kuchen, H., Cole, M.: The Integration of Task and Data Parallel Skeletons. Parallel Processing Letters 12(2), 141-155. 2002.

[13] Kuchen, H., Plasmeijer, R., Stoltze, H.: Efficient Distributed Memory Implementation of a Data Parallel Functional Language. PARLE'94. LNCS 817, 466-475. Springer-Verlag. 1994.

[14] Kuchen, H., Striegnitz, J.: Higher-Order Functions and Partial Applications for a C++ Skeleton Library. Joint ACM Java Grande & ISCOPE Conference, 122-130. ACM. 2002.

[15] Kuchen, H.: A Skeleton Library. In Monien, B., Feldmann, R.: Euro-Par'02. LNCS 2400, 620-629. Springer-Verlag. 2002.

[16] Kuchen, H.: Optimizing Sequences of Skeleton Calls. in D. Batory, C. Consel, C. Lengauer, M. Odersky (Eds.): Domain-Specific Program Generation. LNCS 3016, 254-273. Springer Verlag. 2004.

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

[18] Skillicorn, D.: Foundations of Parallel Programming. Cambridge U. Press. 1994.

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

[20] ZIV-Cluster: http://zivcluster.uni-muenster.de/.