Int J Parallel Prog (2014) 42:968–987
DOI 10.1007/s10766-013-0269-2

A Scalable Farm Skeleton for Hybrid Parallel and Distributed Programming

Steffen Ernsting · Herbert Kuchen

Received: 2 March 2013 / Accepted: 17 August 2013 / Published online: 4 September 2013
© Springer Science+Business Media New York 2013

Abstract Multi-core processors and clusters of multi-core processors are ubiquitous. They provide scalable performance yet introducing complex and low-level programming models for shared and distributed memory programming. Thus, fully exploiting the potential of shared and distributed memory parallelization can be a tedious and error-prone task: programmers must take care of low-level threading and communication (e.g. message passing) details. In order to assist programmers in developing performant and reliable parallel applications *Algorithmic Skeletons* have been proposed. They encapsulate well-defined, frequently recurring parallel and distributed programming patterns, thus shielding programmers from low-level aspects of parallel and distributed programming. In this paper we take on the design and implementation of the well-known *Farm* skeleton. In order to address the hybrid architecture of multi-core clusters we present a two-tier implementation built on top of MPI and OpenMP. On the basis of three benchmark applications, including a simple ray tracer, an interacting particles system, and an application for calculating the Mandelbrot set, we illustrate the advantages of both skeletal programming in general and this two-tier approach in particular.

Keywords High-level parallel programming · Algorithmic skeletons · Farm skeleton · Shared/distributed memory

S. Ernsting (✉)· H. Kuchen
University of Muenster, Leonardo-Campus 3, 48149 Muenster, Germany
e-mail: s.ernsting@uni-muenster.de

H. Kuchen
e-mail: kuchen@uni-muenster.de

1 Introduction

Today, not only in the field of *High Performance Computing* (HPC) multi-core processors and clusters of multi-core processors are ubiquitous. Even in the consumer segment both the growing complexity of applications and the growing amount of data lead to a high demand for high performance. In order to fully exploit the resources provided by multi-core processors and computer clusters, programmers still have to deal with low-level concepts of parallel and distributed programming. These low-level concepts constitute a high barrier to the efficient development of parallel applications and also make it a tedious and error-prone task. High-level concepts for parallel programming would not only simplify programming, but could also enhance portability of parallel applications by abstracting from the underlying hardware.

With algorithmic skeletons (AS) [1,2] Cole has proposed an approach to address these issues. AS can be considered as high-level tools that encapsulate well-defined, frequently recurring parallel and distributed programming patterns, thus hiding the low-level details of parallel and distributed programming. One of these high-level tools is the *Farm* skeleton. It is a master–slave system, where conceptually a farmer accepts a sequence of tasks and propagates them among the workers. Workers solve their allocated tasks and deliver the corresponding results back to the farmer, who collects all the results and propagates the sequence of results to the subsequent stages within a larger process topology.

The main contributions of this paper are both a two-tier concept for the implementation of the *Farm* skeleton as well as an empirical analysis based on three benchmark applications. This two-tier concept makes use of both inter-node parallelization via message passing as well as intra-node parallelization via multi-threading. Thus, it depicts the hardware configuration of most modern computer clusters, where usually vast numbers of multi-core processors are connected with high-speed network interconnects such as Infiniband. We believe that such a two-tier parallelization becomes more and more important, considering that the amount of cores per single computing node will further increase in the next years. Additionally, taken into account the progress that is made on hardware accelerators such as GPUs and the Intel MIC Compute Accelerator, such multi-tier concepts can be considered vital. Regarding this evolution in HPC hardware, writing efficient and reliable parallel programs may become even more difficult and error-prone, hence more time-consuming. That underlines the necessity for high-level tools that address the multi-tier structure of today's computer clusters.

The remainder of this paper is structured as follows: Sect. 2 introduces the *Muenster Skeleton Library* (Muesli), pointing out its concepts and benefits. In Sect. 3 we consider different strategies for implementing the *Farm* skeleton (with respect to its topology) and also describe its implementation in detail. Benchmark applications based on the *Farm* skeleton as well as experimental results are presented in Sect. 4. Section 5 discusses some related work and finally, Sect. 6 concludes the paper and gives a brief outlook to future work.

2 The Muenster Skeleton Library

There are various approaches towards skeletal parallel programming. Because of its roots in functional programming, the first skeletons were implemented in functional languages [1]. Nowadays, the majority of skeleton frameworks are programmed in imperative and object-oriented languages, for instance, C/C^{++} or Java [3–6]. There are several reasons for this trend. The most striking one, of course, is performance. But imperative and object-oriented languages are also more prevalent, especially among HPC developers. Moreover, emerging programming models for many-core architectures as introduced by frameworks such as CUDA [7] and OpenCL [8] gained high popularity in the last years. These frameworks usually integrate seamlessly into imperative languages, such as C/C^{++}.

Our approach to enabling skeletal programming is called Muesli [3]. Muesli is a C^{++} skeleton library that provides various algorithmic skeletons and distributed data structures for shared and distributed memory parallel programming. For efficient support of multi- and many-core computer architectures as well as clusters of both, it is built on top of MPI [9] and OpenMP [10]. Recently, some efforts were made to also provide some of Muesli's skeletons with support for GPU processing using CUDA [11]. Muesli is intended to relieve programmers from low-level, thus error-prone peculiarities of parallel programming. Because not only task parallel skeletons but also distributed data structures on which data parallel skeletons operate are provided, programmers of parallel applications may draw on a variety of parallel and distributed programming patterns and choose those that suit their applications in a natural way.

Conceptually, we distinguish between task parallel and data parallel skeletons. Task parallel skeletons represent well-defined process topologies such as *Farm*, *Divide and Conquer* (D&C) [12], *Pipeline* (Pipe), and *Branch and Bound* (B&B) [13]. They can be arbitrarily nested to create process topologies that define the overall structure of parallel applications. Their algorithm-specific behavior is defined by user functions that describe the algorithm-specific details. Processes within a topology communicate via streams of input and output data. Parallel applications that include irregular data distributions and/or input dependent communication can greatly benefit from such process topologies. For example, consider a B&B algorithm for solving optimization problems. It is defined by both a branching (i.e. splitting) procedure that splits a problem into two or more subproblems, and a bounding procedure that discards subproblems by computing upper and lower bounds for the solution. In this case, the programmer typically chooses the B&B skeleton as the overall structure of his or her application, and provides this skeleton with the algorithm-specific details, i.e. the problem definition as well as the branching and the bounding procedures. The skeleton implementation ensures that the solution to that specific B&B algorithm is computed in parallel. The programmer is totally relieved from any low-level threading and communication details that a manual parallel implementation of the considered B&B algorithm "by hand" would entail.

However, especially in the field of HPC there are many applications that include massive data parallel computations that greatly benefit from efficient data structures and caching effects. For that reason, our data parallel skeletons, e.g., *Map*, *Zip*, and

Fold, are provided as member functions of distributed data structures (DDS), including a one-dimensional array, a two-dimensional matrix [14], and a two-dimensional sparse matrix [15]. These DDS are divided into local partitions that are each stored and processed by a single process. Data parallel skeletons operate on the DDSs' elements and can also manipulate entire DDSs. By providing a skeleton with a specific user function, programmers define the specific behavior of this skeleton, thus the way the entire data structure or its elements are manipulated. In addition to the data parallel skeletons, the DDSs also include communication skeletons that, for example, are used to exchange local partitions of a DDS in order to redistribute the local partitions. When using a DDS and its skeletons, the programmer must of course be aware of the decomposition into local partitions but the low-level threading and communication details are taken care of the DDS's and the skeleton's implementation.

In Muesli, not only task parallel skeletons can be arbitrarily nested: in order to define sophisticated process topologies for highly complex algorithms, task parallel and data parallel skeletons can be nested as well.

As already stated, skeletons take so-called user functions as arguments that define their algorithm-specific behavior. In Muesli, these user functions can be sequential C^{++} functions as well as functors, i.e. classes that overwrite the function call operator (in a sequential manner).[1] As a key feature of Muesli, the well-known concept of *Currying* is used to enable partial application of user functions [16]: a user function that depends on more arguments than provided by a particular skeleton can be partially applied to a given number of arguments, thereby yielding a "curried" function of smaller arity, which can be passed to the desired skeleton.

3 Farm Skeleton

As pointed out in the previous section, in Muesli, task parallel skeletons are nested to create a process topology that suits the structure of a particular application in a natural way. Processes within a topology communicate via streams of data. The outermost nesting level typically consists of the *Pipe* skeleton. A *Pipe* in turn consists of two or more stages that consume input values and transform them into output values. Each stage's exit point is connected to its succeeding stage's entry point. The very first and the very last stage form an exception as the former does not consume input values and the latter does not produce output values, respectively. This functionality is provided by the *Initial* and *Final* skeletons, which always represent the main entry and exit point of a topology, respectively. Note that a nested pipeline does not have to meet this requirement, as its first and last stage stage do not necessarily represent the main entry and exit points of a topology. At the innermost nesting level, the programmer has to define how input values (taken from the input stream of a particular skeleton) are transformed into output values. Muesli therefore provides the *Atomic* skeleton that, similar to the *Initial* and *Final* skeletons, represents an atomic task parallel computation. An argument function defines how input values are transformed into output values.

[1] Due to memory restrictions, GPU-enabled skeletons must be provided with functors as arguments.

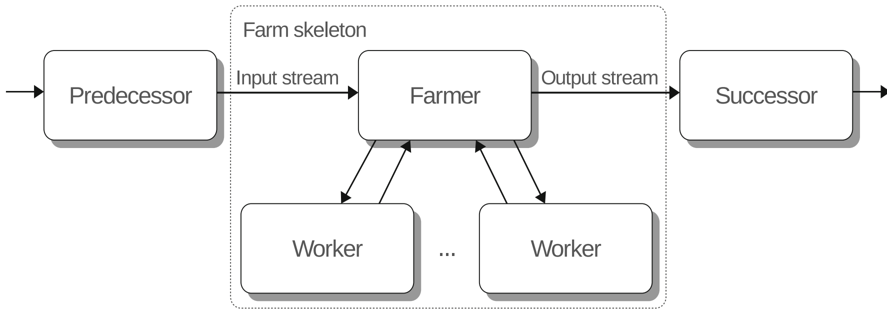

Fig. 1 The *Farm* skeleton

The *Farm* skeleton defines a well-known process topology (depicted in Fig. 1), where a farmer takes input values from a stream and propagates them to its workers. When a worker is served with an input value, it calculates the corresponding output value and redirects it to the farmer in order to receive the next input value. All workers apply the same operation defined by the given user function. The farmer puts the output values produced by its workers back to the output stream as soon as they are available. According to the task parallel skeletons' nesting abilities, a worker can either be represented by an atomic skeleton or be represented by another task parallel skeleton. However, workers of the same farm must be identical and must also provide at least one entry as well as one exit point.

In [17], Poldner and Kuchen consider several approaches to implement the *Farm* skeleton with respect to various topologies. They show that, given the topology depicted in Fig. 1, the farmer might constitute a bottleneck, especially when it has to serve a large number of worker processes. In order to alleviate this bottleneck to a certain degree, they propose a process topology that simply drops the farmer and shifts its functionality to the workers and the preceding stage in the pipeline. Instead of requesting for new input data from a farmer, workers are directly assigned to input data by the preceding stage of the pipeline in a random or round-robin order. Calculated output values are then forwarded to the succeeding stage analogously. Thus, a farm has no single entry and exit point but each of its workers acts as a single entry and exit point itself. This topology is depicted in Fig. 2, where two farms are nested in a pipeline. Representing the first stage of the pipeline, the *Initial* skeleton directly distributes its generated output data among the workers of the first farm (representing the pipeline's second stage). Each worker of the first farm in turn directly distributes its output data among the second farm's workers, which finally forward their output data to the final stage represented by the *Final* skeleton.

Listing 1 displays the implementation of a task parallel example application, that uses the *Farm* skeleton to multiply integer values by two and to print out the result. While this example is somewhat trivial, it is made to illustrate the creation of process topologies within Muesli rather than to constitute a sophisticated parallel algorithm implementation. Here, a pipeline of three stages is constructed. Note that there is a class template for every task parallel skeleton. The first and the last stage (`p1` and `p4`) of the pipeline are represented by the atomic skeletons *Initial* and *Final*. They both are

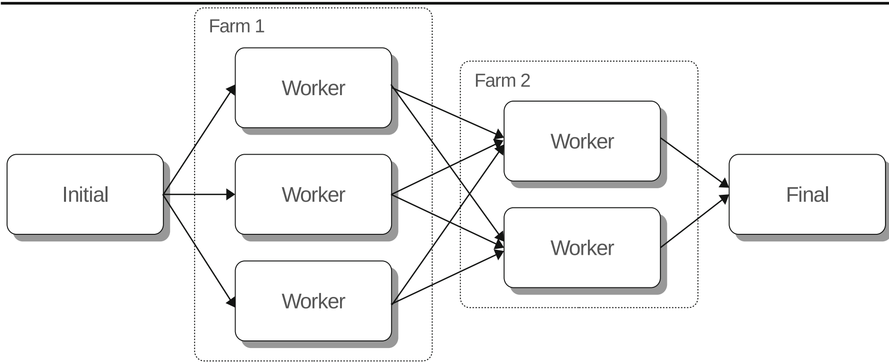

Fig. 2 Pipeline of two optimized farms

instantiated with a specific user function (`init` and `fin`, respectively) as argument. The *Farm* skeleton constitutes the pipeline's middle stage (`p3`). Its constructor expects both an instance of the desired type of workers as well as the number of workers to be employed. In this case, there are three atomic workers represented by the *Atomic* skeleton (`p2`). Their behavior is defined by the given function `multByTwo`. Finally, the task parallel computation is started at the outermost nesting level, i.e. the pipeline process. Also note that, in order to be arbitrarily nested, all class templates representing task parallel skeletons must inherit from the same abstract base class, i.e. the class `Process`.

The example in Listing 1 also illustrates another important aspect of the task parallel skeletons provided by Muesli: the argument function of the *Initial* skeleton returns pointers instead of values. By returning a `null` pointer, the function indicates that there is no input data left. The process topology may then be terminated to ensure that no user data is lost, and a possible subsequent task parallel skeleton computation with a potentially different topology can be started smoothly.

3.1 Implementation Aspects

The *Farm* skeleton with its topology described above is designed to scale on distributed memory machines such as clusters of single-core computers. With a single process of the topology being mapped onto a single MPI process, the *Farm* skeleton additionally inherits support for shared memory computers from MPI. However, with the emergence of hybrid computer architectures including combined shared and distributed memory such as clusters of multi-core computers, it is important for the skeleton topology to reflect this hybrid architecture in some natural way. Creating as many workers as there are cores in total rapidly leads to long idle times for the workers because a single process that generates input data would have to serve all the workers with input data, thus constituting a bottleneck in communication. That is why we have decided to implement a hybrid variant of the *Farm* skeleton. Its implementation details will be described in detail in the following.

As basis for the implementation of the two-tier *Farm* skeleton, we have adopted the above described topology proposed by Poldner and Kuchen. In order to on the

```
int current = 0;
int nWorkers = 3;

int* init() {
    (current++ < 10) ? return new int(current) : return 0;
}

int multByTwo(int value) {return value*2;}

void fin(int value) {printf("result: %d", value);}

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

    // Create a process topology using C++ constructors.
    Initial<int>          p1(init);
    Atomic<int,int>       p2(multByTwo);
    Farm<int,int>         p3(p2, nWorkers);
    Final<int>            p4(fin);
    Pipe                  p5(p1, p3, p4);

    // Start the task parallel calculation.
    p5.start();

    terminateSkeletons();
}
```

Listing 1 Task parallel example application using the *Farm* skeleton.

one hand reflect the hybrid (shared and distributed memory) architecture of modern cluster computers, we had to make major changes to the task parallel skeletons in general and to the *Farm* skeleton in particular. On the other hand, in order to not raise the complexity, we wanted to make as few changes to the overall structure and the interface as possible.

Multi-threaded workers The most important modification to the internal structure of the *Farm* skeleton is that atomic workers of a farm are extended to so-called multi-threaded workers that make use of shared memory parallelization. Instead of sequentially calculating a single output value from a single input value at a certain time, multi-threaded workers now calculate multiple output values from the same number of input values in parallel. By adding this additional parallelization level, each thread of a multi-threaded worker now becomes an atomic worker itself. Hence, a multi-threaded worker is more or less an intermediary for shared memory parallelization. Note that this modification is fully transparent to the programmer: to not shift the responsibility for shared memory parallelization to the programmer, a multi-threaded worker's specific behavior is still defined by the `Atomic` skeleton. Hence, it requires as argument a sequential user function that calculates a single output value from a single input value, i.e. representing an atomic task parallel computation. The multi-threaded worker's implementation ensures the parallel application of the given user function. Listing 2 briefly describes the multi-threaded worker's functionality.

```
void mtWorker() {
  omp_set_nested(1);
  #pragma omp parallel
  {
    #pragma omp single nowait
    {
       while (!finished || !SendPool.empty()) {
        if (checkIncomingMessage()) {
          receive(input);
          if (input == STOP)
            finished = 1;
          else
            putToWorkPool(input);
        }

        if (!SendPool.empty()) {
          takeFromSendPool(output);
          send(output);
        }
      }
    }

    #pragma omp single
    {
       while (!finished || !WorkPool.empty()) {
        if (!WorkPool.empty()) {
          takeFromWorkPool(input);
          #pragma omp parallel for
          for (int i = 0; i < chunkSize; i++) {
            output[i] = atomicUserFunction(input[i]);
          }
          putToSendPool(output);
        }
      }
    }
  }
}
```

Listing 2 Simplified functionality of a multi-threaded worker.

In order to overlap communication and computation, there are two code sections executed in parallel: the first section is responsible for receiving input data from the predecessors and sending output data to the successors within the topology. When receiving input data, data is put to a local work pool. Output data, in turn, is taken from the so-called send pool and is then sent to a successor. Within each step of the `while` loop one message may be received and one message may be sent. If there are no messages to receive, only messages are sent and vice versa. The second section is responsible for taking sets of input values from the work pool, calculating corresponding sets of output values in parallel, and finally putting these sets of output values into the send pool. The parallel calculation is represented by a parallel `for` loop, where each thread applies the `atomicUserFunction` to a single input value at a time. It is important to note that for this kind of parallelization we make use of nested

parallelism provided by OpenMP: the nested `parallel for` pragma may employ all threads of the initial thread team instead of just the single one employed by the preceding `single` pragma. Thus, with the two sections executed in parallel, both the work and the send pool have a single producer and a single consumer and are therefore implemented as a wrapper class for a double-ended queue (`std::deque` $< \mathrm{T} >$) that synchronizes calls to the underlying data structure. Note that also the atomic skeletons *Initial* and *Final* provide the above described functionality of overlapping communication and computation. There is also an obvious alternative implementation for a multi-threaded worker without a work and a send pool. In this case the multi-threaded worker receives a set of input values, calculates in parallel the corresponding set of output values and sends it to one of its successors. As this implementation does not overlap communication and computation, for some applications it might have adverse effects on scalability.

Because a multi-threaded worker needs to have multiple input values at hand in order to calculate multiple output values in parallel (remember that an original farm worker consumes a single input value to produce a single output value), data has to be aggregated at some point: either a sending process must aggregate output data before sending or a receiving process must aggregate input data before calculating new output data in parallel. In order to reduce the number of independent network transfers and work pool accesses, we have decided to aggregate output data at the sending side. Thus, work and send pools store sets of values instead of single values.

Aggregation of output That leads to another major modification to the implementation of the task parallel skeletons: the *Initial* skeleton, always representing the initial stage of a pipeline, aggregates its output values to sets of a given size. Thus, it does not send single values to its successors but sets of values. Internally, these sets are represented by `std::vector` objects provided by the C^{++} standard library, because a vector's size can be dynamically adjusted and the underlying data structure (ordinary array) provides very efficient data access. The size of these sets may be set at compile time or be calculated at runtime. The succeeding stage takes these sets of values as input and calculates the corresponding output sets to be forwarded to its succeeding stage. In conclusion, processes within a topology created by (nested) task parallel skeletons communicate via streams of sets instead of streams of single values. However, this modification is completely transparent to the user as it has no effect on any of the user functions.

4 Benchmarks

In order to investigate how the described *Farm* skeleton performs on hybrid computer architectures with shared and distributed memory, we have implemented three benchmark applications: a simple ray tracer, an interacting particles system, and an application for calculating the Mandelbrot set. All three benchmark applications are based on the process topology depicted in Fig. 3, where a single farm with `n` multi-threaded workers is nested into a pipeline of three stages.

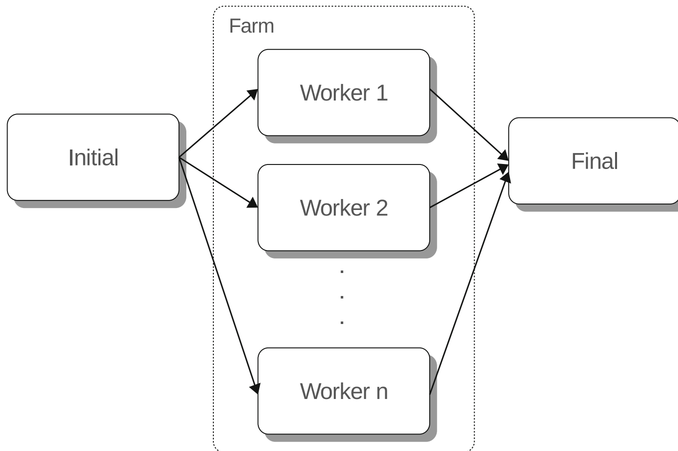

Fig. 3 General process topology for the three benchmark applications

The underlying platform for the benchmarks is called PALMA: a multi-node, multi-core computer cluster. The compute nodes, running CentOS 5.9, provide a six-core dual-socket Intel Xeon Westmere processor with a total of 12 cores and 24 Gb main memory. They are connected through Infiniband.

All benchmark applications were run with varying numbers of workers, ranging from a single worker to 64 workers. Each worker is mapped to a single MPI process occupying a single node of the cluster. For each number of workers, the applications were run with 1, 4, 8, and 12 threads, respectively.

4.1 Ray Tracing

Ray tracing [18] is a well-known rendering technique in computer graphics. It can be used for generating high quality or even photo-realistic 2D images of 3D scenes by calculating the paths of rays of light entering the viewer's eye or the camera. Paths are traced backwards from the viewpoint, through a pixel in the image plane until they intersect with some object in the scene. That is why this technique is also often referred to as backwards ray tracing. Depending on the surfaces of the intersected objects, the corresponding pixel will be colored. This way, a variety of optical phenomena, for instance, reflections, shadows, and dispersion can be simulated, certainly at a high computational cost. Techniques such as spatial anti-aliasing can even further increase the computational complexity of ray tracing. Both the computational complexity and the computational independence make ray tracing amenable to parallel programming.

Implementation Exemplary for all benchmarks, we briefly consider the details of the task parallel skeleton implementation of a simple ray tracer. It is presented in Listing 3.

First of all, a call to the function `initSkeletons` must be made in order to initialize Muesli. Then the 3D scene is created. It describes the settings of the scene, i.e. the camera position and angle as well as the screen position, and contains a number

of objects to be displayed. These objects contain information about their shape, their position in the scene and the material of their surface. In the next step, the process topology depicted in Fig. 3 is created by making use of the constructors of the respecting skeleton classes. As described in Sect. 3, it requires three user functions to provide this general process topology with the algorithm-specific details that describe each process's particular behavior. In this case, these user functions were derived from a sequential ray tracing algorithm [19]. It is important to notice that only very few changes to the sequential code were made in order to decompose the algorithm into three separate units, namely the functions `init`, `castRay`, and `fin`. The `init` function defines the algorithm-specific behavior of the *Initial* skeleton. It produces input values for the farm's workers. When calling this function, it yields the next pixel of the image to be rendered. By successively calling this function, it creates all pixels to be included by the image in a row-wise manner. A pixel is represented by the type `Pixel`. It stores the row and column indices as well as the pixel's color. The algorithm-specific details of ray tracing are encapsulated in the function `castRay`. It defines the algorithm-specific behavior of the farm's workers. Thus, it requires a pixel as argument and, corresponding to the pixel's position in the image, calculates the color of that pixel by performing simple ray tracing. Because each pixel can be independently processed, this can be considered an atomic operation. The function `fin` defines the algorithm-specific behavior of the `Final` skeleton. Thus, it finally stores the colored pixels in the resulting image. The task parallel calculation is started by calling the `start` method provided by the `Pipe` object `p5`. Finally, a call to the `terminateSkeletons` function shuts down Muesli and terminates the application. Figure 4 shows an example image rendered with the above described parallel ray tracing application. It includes an infinite plane, five colored spheres, and one sphere light in the center.

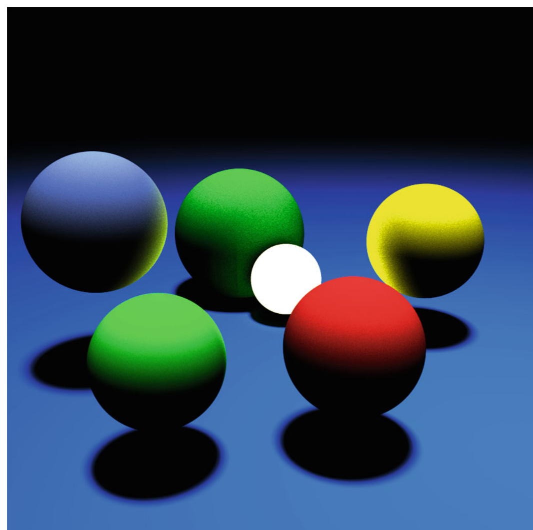

Fig. 4 Example image rendered with the ray tracing benchmark application. It includes an infinite plane, five colored spheres, and one sphere light in the center

```
struct Pixel {
  Color col;
  size_t xPos, yPos;
};

Pixel* init() {return nextPixel();}

Pixel castRay(Pixel& p) {
    sequentialRayTracing(p);
    return p;
}

void fin(Pixel& p) {storePixel(p);}

int main(int argc, char **argv) {
    // Initialize Muesli.
    initSkeletons(argc, argv);

    // Create the 3D scene.
    createScene();

    // Create a skeleton topology using C++ constructors.
    Initial<Pixel>          p1(init);
    Atomic<Pixel, Pixel>    p2(castRay);
    Farm<Pixel, Pixel>      p3(p2,nWorkers);
    Final<Pixel>            p4(fin);
    Pipe                    p5(p1, p3, p4);

    // Start the calculation.
    p5.start();

    // Terminate Muesli.
    terminateSkeletons();
}
```

Listing 3 Implementation of a simple ray tracer using the *Farm* skeleton (partly in pseudo code).

Benchmark results The 3D scene for the ray tracing benchmark includes as objects 100 simple spheres and 10 simple rectangle-shaped lights. The objects are equally distributed among the screen. The rendered image is of size 1024×1024, resulting in a total of 1,048,576 pixels to be colored in terms of ray tracing. As output format we use the ppm format, as it is a very simple image format that is entirely sufficient for our needs. The results of the benchmark are reported in Fig. 5 and Table 1. All in all, the results of the ray tracing benchmark show that the hybrid *Farm* skeleton performs well on hybrid computer architectures with shared and distributed memory. While the inter-node speedups (reading the tables in a row-wise manner) are close to ideal even for large numbers of workers processes, intra-node speedups (reading the tables in a column-wise manner) are close to ideal only for small numbers of threads per worker process. When shifting to larger numbers of threads the speedups slightly decrease. This is due to both an undersized amount of work for equally employing all workers' threads and the slight overhead introduced by the high level of abstraction.

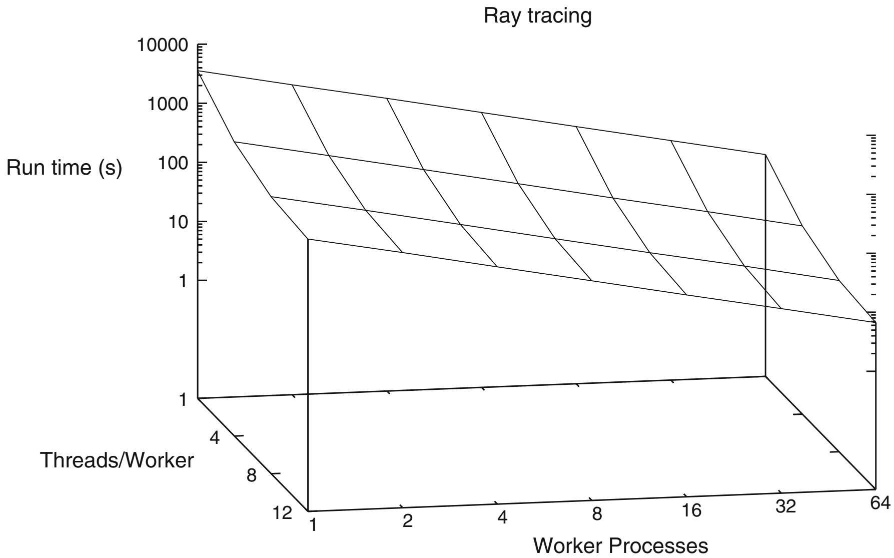

Fig. 5 Run times of the ray tracing benchmark

Table 1 Run times (in seconds) of the ray tracing benchmark

nt/np	1	2	4	8	16	32	64
1	3567.96	1786.83	913.92	457.03	228.41	114.48	57.56
4	969.80	489.17	245.66	122.89	61.07	31.17	15.47
8	494.98	251.48	125.70	62.64	31.44	16.14	7.97
12	412.61	211.43	104.81	52.58	26.25	13.40	6.74

np and nt denote the number of worker processes and the number of threads per worker process, respectively

4.2 Interacting Particles

The problem of interacting particles has many applications ranging from molecular dynamics to galactic dynamics at the other end of the spectrum. A system of interacting particles includes a number of charged particles and a discrete or continuous area, in which all the particles interact with each other. For a number of time steps, each particle in the system interacts with every other particle in the system (except of itself). The interaction of two electrically charged particles is described by the Coulomb law. However, we are not going to explain its details. After each time step, all particles' positions and velocities are updated with respect to the previously calculated interactions.

As previously mentioned, for the implementation of an interacting particles system using the *Farm* skeleton presented in this paper, we build on the process topology depicted in Fig. 3. In this case, the first stage of the pipeline successively yields the particles of the system. Atomic farm workers are responsible for calculating the forces that affect a given particle. Finally, the last stage of the pipeline gathers all particles

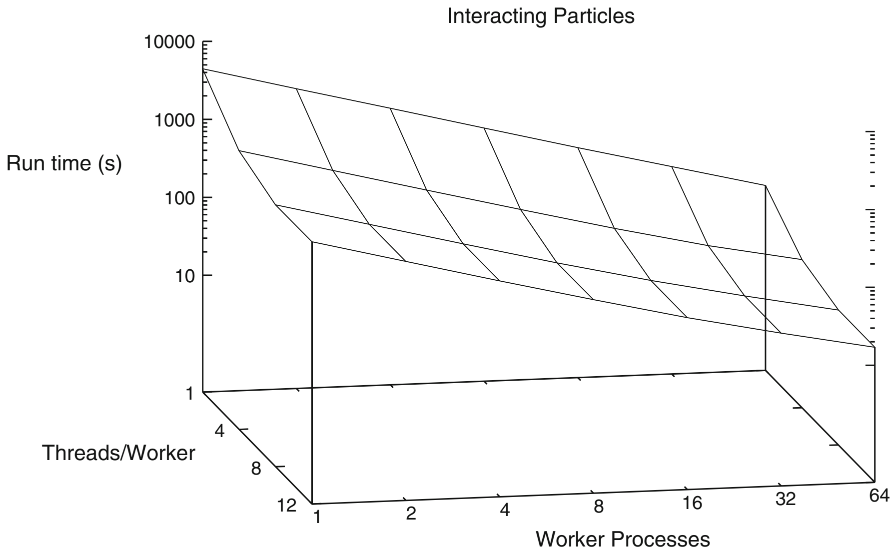

Fig. 6 Run times of the **Muesli** implementation of the interacting particles benchmark

Table 2 Run times (in seconds) of the interacting particles benchmark (**Muesli**)

nt/np	1	2	4	8	16	32	64
1	4458.15	2227.72	1121.73	562.42	283.21	145.40	74.70
4	1194.19	598.76	301.18	152.74	78.44	42.30	25.15
8	726.93	366.06	185.02	94.69	50.70	28.84	17.09
12	734.76	368.58	186.89	96.11	50.74	28.90	17.11

np and nt denote the number of worker processes and the number of threads per worker process, respectively

of the system and updates their positions and velocities. In order to carry out multiple time steps, we have enclosed the task parallel computation by a loop. Within each step of the loop, the entire particle system is broadcasted among all processes in order to provide every participating process with the most up to date particle data. At first glance, this may seem very costly, but compared to the computational complexity of calculating forces to the particles, it turns out to be a negligible overhead.

Benchmark results For this benchmark, we have implemented an interacting particle system of charged plasma particles. For reasons of simplification, we did not take radiation into account, because this would add enormous complexity to the algorithm. The interactions of 100,000 particles are calculated over 20 time steps. The results are reported in Fig. 6 and Table 2.

The overall scaling properties of the interacting particles application are similar to the ray tracing application. The inter-node speedups are close to ideal even for large numbers of workers. However, because the algorithm is memory bound the intra-node speedups are close to ideal only for small numbers of threads per worker

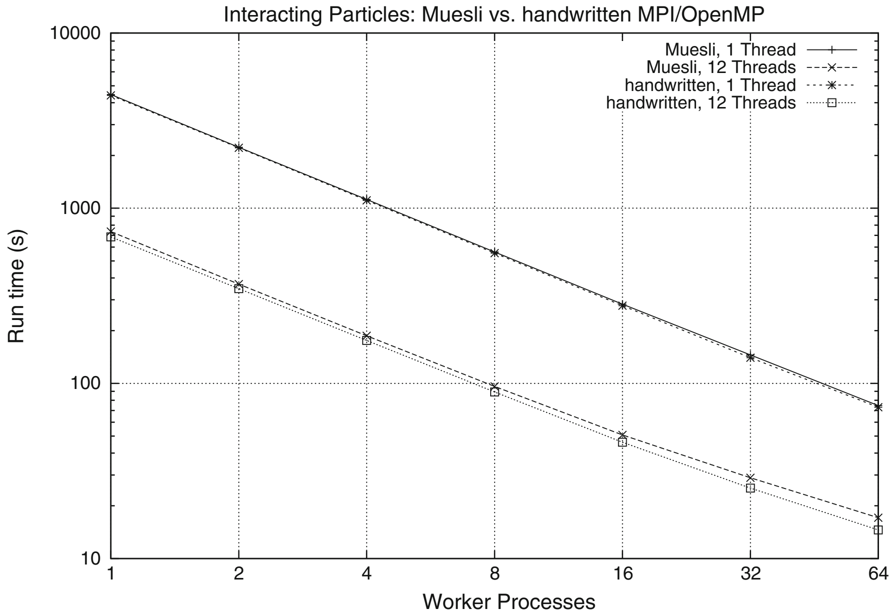

Fig. 7 Comparison between the Muesli and the **handwritten MPI/OpenMP** implementation of the interacting particles benchmark

Table 3 Run times (in seconds) of the interacting particles benchmark (**handwritten MPI/OpenMP**)

nt/np	1	2	4	8	16	32	64
1	4402.60	2210.64	1108.45	553.62	277.89	140.00	73.13
4	1188.61	596.25	299.12	150.62	76.29	39.62	22.04
8	709.10	356.42	179.20	93.18	47.54	25.70	14.96
12	686.09	346.97	175.69	89.34	46.13	25.25	14.57

np and nt denote the number of worker processes and the number of threads per worker process, respectively

and strongly decrease when shifting to larger numbers of threads. For 12 threads there is even a slowdown identifiable with respect to 8 threads. In Fig. 7 we draw a comparison between the Muesli and the handwritten MPI/OpenMP implementation of this benchmark: the handwritten version has only a very slight edge over the Muesli version. Also the scaling properties are nearly the same for both implementations. The results show that there is close to no overhead introduced by the high level of abstraction. For reasons of clarity we have only included run times for 1 and 12 threads, respectively, in Fig. 7. For full details see Tables 2 and 3.

4.3 Mandelbrot Set

The Mandelbrot set is a set of points in the complex plane, whose boundary describes a well-known fractal. It contains all points c of the complex plane for which the

orbit of z_n does not tend to infinity when iterating the quadratic recurrence equation $z_{n+1} = z_n^2 + c$ with $z_0 = 0$. Thus, depending on both the iteration depth as well as the accuracy of computation, the Mandelbrot set potentially involves infinite computational complexity. With its data parallel nature it is naturally suited for parallel programming. Note that for each point of the discrete complex plane the recurrence equation can be independently iterated. However, parallel programmers have to take care of heavy load imbalance because points that are not members of the Mandelbrot set tend to introduce just a very small number of iteration steps. In contrast, for points that are element of the the set, it is necessary to iterate the recurrence equation until the maximum number of iterations is reached.

Similar to the other benchmarks, the task parallel process topology is already defined. Here, the first stage of the pipeline successively yields the points of the discrete complex plane. To address load imbalance, the points are yielded in a block-wise manner so that each block's elements preferably involve similar computational complexity. The farm workers then iterate the recurrence equation for each point, and by that determine wether a given point belongs to the set or not. Finally, in the last stage all points of the discrete complex plane are gathered. Depending on the iteration depth, each point is assigned a color and is stored in an image file, for instance. Unlike the other benchmarks, in this benchmark workers do not require access to additional input data (such as the 3D scene for the ray tracing benchmark). That raises the question whether a purely data parallel implementation would meet the requirements in a more adequate way than a task parallel simulation of data parallelism would do. In order to answer this question, we have implemented a task parallel and a data parallel variant. The data parallel variant was implemented by making use of the distributed matrix (for representing the discrete complex plane) and its data parallel skeletons provided by Muesli. For both applications the size of the discrete complex plane is fixed to 8192×8192. The results are reported in Fig. 8 and Table 4. Figure 9 features a comparison between the task parallel and the data parallel variant. Run times of the data parallel application are summarized in Table 5.

Benchmark results The results of the Mandelbrot benchmark present an overall scaling behavior similar to the other benchmarks: for small numbers of workers and threads per worker both inter- and intra-node speedups are close to ideal. For larger numbers of workers and threads the speedups slightly decrease. However, the overall performance and scaling behavior is considerable. Figure 9 features a comparison between the task parallel and the data parallel variant of the Mandelbrot application.

Due to a fixed data distribution (fixed block-wise decomposition of the matrix), the data parallel implementation does not take care of load imbalance. This behavior is clearly reflected in the results: considering the curve of the single-threaded data parallel variant, it is observable that this curve flattens when shifting from 2 to 4 data parallel workers. At this point, with respect to the structure of the Mandelbrot set, the upper left and lower left blocks of the matrix just introduce very little computational complexity. In contrast to the data parallel implementation, the task parallel implementation naturally takes care of load imbalance, because in this case the block sizes do not depend on the number of participating processes and can therefore be chosen much smaller. This leads to a more balanced distribution (in terms of complexity)

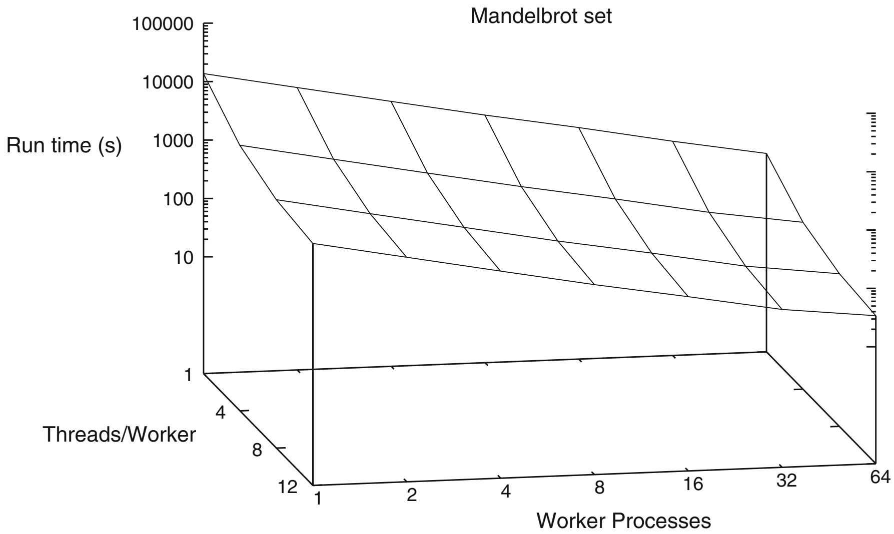

Fig. 8 Run times of the **task parallel** Mandelbrot benchmark

Table 4 Run times (in seconds) of the **task parallel** Mandelbrot benchmark

nt/np	1	2	4	8	16	32	64
1	13707.40	6855.83	3446.18	1744.91	922.09	467.67	250.24
4	3538.90	1770.25	889.86	451.18	240.76	123.43	72.10
8	1800.76	900.79	452.86	229.66	123.52	64.03	41.13
12	1393.78	701.37	352.81	178.74	96.79	50.51	34.23

np and nt denote the number of worker processes and the number of threads per worker process, respectively

of data among the workers and their multiple threads. This in turn leads to higher performance. Considering the 12 thread variants, the task parallel implementation has the edge over the data parallel implementation for all numbers of workers. This is, of course, also due to better load balancing capabilities. Moreover, calculating the Mandelbrot set does not much benefit from cache effects or efficient data structures as other data parallel applications do.

5 Related Work

In [3], Kuchen presents the initial implementation of the *Farm* skeleton within Muesli. As described in Sect. 3, it features a farmer that takes input values and distributes them among the workers. Workers must send their results back to the farmer in order to get the next input value. In [17], Poldner and Kuchen present several optimized process topologies for the *Farm* skeleton. These include a farm with dispatcher, a farm with dispatcher and collector as well as a farm without dispatcher and without collector.

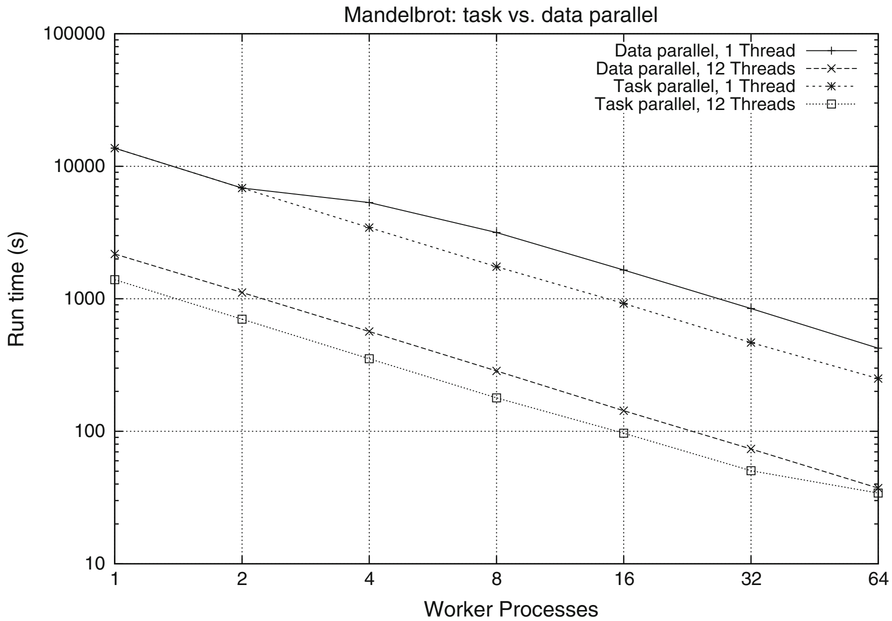

Fig. 9 Comparison between the task parallel and the **data parallel** implementation of the Mandelbrot benchmark

Table 5 Run times (in seconds) of the **data parallel** Mandelbrot benchmark

nt/np	1	2	4	8	16	32	64
1	13708.50	6855.54	5325.42	3161.26	1648.46	845.37	424.33
4	5325.12	3161.63	1648.16	845.11	424.68	212.70	106.77
8	3161.05	1648.12	845.48	424.46	212.72	106.77	53.67
12	2175.59	1116.70	566.56	285.43	143.04	73.54	37.32

np and nt denote the number of worker processes and the number of threads per worker process, respectively

Note that the latter is taken as the base topology for the hybrid farm presented in this paper.

There are also other skeleton libraries that offer a *Farm* skeleton. While eSkel [4] provides the classical *Farm* with a farmer, the P3L [20] *Farm* splits the farmer into a dispatcher and a collector in order to tackle the bottleneck at the farmer process.

Also the Java based skeleton frameworks JaSkel [21] and Skandium [6] provide a *Farm* skeleton. In JaSkel, skeletons are based on inheritance, thus they can be orthogonally and hierarchically composed. Farms can be combined to so-called multi-level farms that can take advantage of modern multi-core clusters. Skandium provides algorithmic skeletons for shared memory programming enabled by Java's built-in threading capabilities. Similar to a farm with a farmer, Skandium's *Farm* skeleton follows a master–slave approach.

To the best of our knowledge, there are no cluster (with shared and distributed memory) results reported for any of the above mentioned *Farm* skeleton implementations.

6 Conclusion

We have presented a pragmatic approach to implementing a scalable, two-tier *Farm* skeleton. With its two-tier concept of shared and distributed memory parallelization it addresses the architecture of modern computer clusters in a natural way. We consider this two-tier concept to be very important, taking into account that the number of cores per processor is expected to further increase in the future. By briefly considering the details of a skeletal implementation of a simple ray tracing algorithm, we have demonstrated the simplicity and handiness of this approach. Moreover, our benchmarks show that various parallel applications can be efficiently implemented with the *Farm* skeleton and also prove its scalability and performance. Comparisons between Muesli and handwritten MPI/OpenMP code show that the overhead introduced by the high level of abstraction is negligible.

In summary, programmers can benefit from using algorithmic skeletons in various ways. Instead of being concerned with the underlying communication and threading details, they may concentrate on the algorithm. By simply choosing for the appropriate process topology (by nesting the corresponding skeletons) and providing the algorithm-specific details in a sequential manner, programmers can rapidly develop parallel applications. The separation of algorithm and communication logic also enables portability. For example, switching the communication protocol only requires the skeletons' implementation to be changed instead of every single application. Programmers can also use algorithmic skeletons for rapidly creating a prototype of a parallel application in order to determine the appropriate overall structure, and to investigate its scaling and performance properties.

For the future work, we plan to provide GPU support for the `Farm` skeleton. Due to its scalable architecture we think that GPU support can be integrated seamlessly and can further speed up parallel applications. In order to address heavy load imbalances, we think of adding load balancing capabilities to the farm's workers. Additionally, we want to enhance other task parallel skeletons such as B&B and D&C with the presented two-tier concept of shared and distributed memory parallelization.

References

1. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge (1989)
2. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput. **30**(3), 389–406 (2004)
3. Kuchen, H.: A skeleton library. In: Proceedings of the 8th International Euro-Par Conference on Parallel Processing, Euro-Par '02, London, UK, pp. 620–629. Springer (2002)
4. Benoit, A., Cole, B., Gilmore, S., Hillston, J.: Flexible skeletal programming with eskel. In: Proceedings of the 11th International Euro-Par Conference, vol. 3648, pp. 761–770. Springer (2005)
5. Matsuzaki, K., Iwasaki, H., Emoto, K., Hu, Z.: A library of constructive skeletons for sequential style of parallel programming. In: Proceedings of the 1st international conference on Scalable information systems, p. 13. ACM (2006)

6. Leyton, M., Piquer, J.: Skandium: multi-core programming with algorithmic skeletons. Euro-micro PDP 2010 (2010)
7. Nvidia Corp.: NVIDIA CUDA C Programming Guide 5.0. Nvidia Corporation (2012)
8. OpenCL Working Group: The OpenCL specification, Version 1.2 (2011)
9. Gropp, W., Lusk, W., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge (1996)
10. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming. Scientific and Engineering Computation. MIT Press, Cambridge (2008)
11. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-gpu systems and clusters. Int. J/ High Perform. Comput. Netw. **7**(2), 129–138 (2012)
12. Poldner, M., Kuchen, H.: Skeletons for divide and conquer algorithms. In: Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN). ACTA Press (2008)
13. Poldner, M., Kuchen, H.: Algorithmic skeletons for branch and bound. In: Proceedings of the 1st International Conference on Software and Data Technology (ICSOFT), vol. 1, pp. 291–300 (2006)
14. Ciechanowicz, P., Kuchen, H.: Enhancing muesli's data parallel skeletons for multi-core computer architectures. In: 12th IEEE International Conference on High Performance Computing and Communications (HPCC), pp. 108–113. IEEE (2010)
15. Ciechanowicz, P.: Algorithmic skeletons for general sparse matrices on multi-Core processors. In: Proceedings of the 20th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS), pp. 188–197 (2008)
16. Kuchen, H., Striegnitz, J.: Higher-order functions and partial applications for a c++ skeleton library. In: Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande, pp. 122–130. ACM (2002)
17. Poldner, M., Kuchen, H.: On implementing the farm skeleton. Parallel Process. Lett. **18**(1), 117–131 (2008)
18. Appel, A.: Some techniques for shading machine renderings of solids. In: Proceedings of the April 30-May 2, 1968, Spring Joint Computer Conference, pp. 37–45. ACM (1968)
19. Farnsworth, M.: Basic ray tracer. http://renderspud.blogspot.de/2012/04/basic-ray-tracer-stage-3.html. Online, accessed 2013–06-13
20. Danelutto, M., Pasqualetti, F., Pelagatti, S.: Skeletons for Data Parallelism in p3l. In: Lecture Notes in Computer Science, vol. 1300, pp. 619–628. Springer (1997)
21. Fernando, A.P.J., Sobral, J.: Jaskel: a java skeleton-based framework for structured cluster and grid computing. In: 6th IEEE International Symposium on Cluster Computing and the Grid, Vol. 5. IEEE Press (2006)