Implementation Skeletons in Eden: Low-Effort Parallel Programming*

Ulrike Klusik[1], Rita Loogen[1], Steffen Priebe[1], and Fernando Rubio[2]

[1] Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans Meerwein Straße, D-35032 Marburg, Germany
{klusik,loogen,priebe}@mathematik.uni-marburg.de

[2] Universidad Complutense de Madrid, Departamento de Sistemas Informáticos y Programación, Facultad de Ciencias Matemáticas, E-28040 Madrid, Spain
fernando@sip.ucm.es

Abstract. *Algorithmic skeletons* define general patterns of computation which are useful for exposing the computational structure of a program. Being general structures they qualify as a target for parallelisation, which is most often carried out by providing specialised, non-portable, low-level parallel implementations (*architectural skeletons*) of each algorithmic skeleton for different platforms. In the paper we introduce an intermediate layer of *implementation skeletons* for the parallel functional language Eden. These are portable high-level skeletons which simplify the design of parallel programs substantially. Runtime experiments on a network of workstations and on a Beowulf cluster have shown that even on such *high-latency* parallel platforms good speedups can be obtained.

1 Introduction

The inherent parallelism of functional programs often leads to many fine-grained tasks, while, due to costly communication and fast processors, conventional parallel machines depend on coarse-grained tasks to deliver speedups. A parallel functional language has to bridge that gap, either in an implicit or explicit way. Ideally, the programmer should not be bothered with the low-level details of parallel execution. But often speedups can only be achieved when one is able to control the costs introduced by processes, communication, and data distribution. Therefore it is necessary to find a level of abstraction which gives programmers enough control to implement their parallel algorithms efficiently (including granularity issues) and at the same time frees them from the low-level details of process management.

The parallel functional language Eden[1] [3, 4] provides such a level of abstraction. Eden is explicit about processes and their input and output data, but abstracts from the communication of data between processes and the synchronisation required. The Eden implementation is a freely available[1] distributed

* Work supported by the DAAD (Deutscher Akademischer Austauschdienst), by the spanish project CICYT-TIC97-0672, and by the Acción Integrada HB 1999-0102

[1] URL: http://www.mathematik.uni-marburg.de/inf/eden

implementation that maps process structures directly to the underlying architecture giving the programmer more control over work and data distribution.

In this paper we show methods to support the construction of efficient parallel programs in Eden. This is a first step towards a programming methodology for Eden programmers which shows how to use the language effectively. Our approach is based on the concept of *skeletons* [5]. A skeleton is a high-level parallel programming construct (often a polymorphic higher-order function) which represents a common parallel computation pattern. Associated with a skeleton are usually specialised, efficient low-level implementations for various parallel machines called *architectural skeletons*. We propose to consider an intermediate layer of *implementation skeletons* between the high-level algorithmic skeletons and the low-level architectural skeletons. As a case study we introduce various parallel Eden implementation skeletons for the well-known algorithmic skeleton `map`. Being able to define the abstract specification of a skeleton as well as its parallel implementations in the same declarative language gives a solid basis for proving the correctness and other properties of skeleton implementations. We present runtime results of several realistic benchmark programs which have been parallelised using the implementation skeletons on a network of workstations and on a Beowulf cluster. These architectures are examples of extremely high-latency, low-bandwidth parallel machines, favouring coarse-grained computation with minimal communication. Our test cases have been a ray tracer, a computer algebra algorithm for solving linear equation systems and the calculation of Mandelbrot sets. Although it is difficult to achieve speedups in the presence of high communication costs, Eden provides enough control to cope with this obstacle.

The main contributions of this paper are the introduction of parallel implementation skeletons which enable parallel programming with low effort in Eden (and in every other language in which such skeletons can be expressed) and the presentation of runtime results which reveal reasonable absolute speedups on high-latency systems for the non-trivial benchmark programs mentioned above.

The next section introduces the key features of Eden which are necessary to understand the implementation skeletons defined in Section 3. Section 4 presents the results of our experiments. The paper finishes with a discussion of related work and conclusions.

2 Eden

Eden [3] extends the lazy functional language Haskell [18] with syntactic constructs for *explicitly* defining processes [4]. Eden's process model provides direct control over process granularity, data distribution and communication topology.

Defining Processes. An Eden process maps inputs `in_1`, ..., `in_m` to outputs `out_1`, ..., `out_n`. Its behaviour is specified by a *process abstraction* with `Process` being a newly defined type constructor:

```
p :: Process (it_1,...,it_m) (ot_1,...,ot_n)
p =  process (in_1,...,in_m) -> (out_1,...,out_n)
     where equation_1 ... equation_r
```

Example 1. A function `f :: a -> b` can be embedded into a process abstraction via the function `mkProc`:

```
mkProc   :: (a -> b) -> Process a b
mkProc f =  process x -> f x
```

The function argument will be communicated to a process generated using `mkProc f`. ◁

Processes are dynamically created using process instantiations. A *process instantiation* provides a process abstraction with actual input parameters. Its evaluation leads to the creation of a process together with its interconnecting communication channels. Processes communicate via *unidirectional channels* which connect one writer to exactly one reader. We use the operator

```
# :: (Transmissible a, Transmissible b) =>
     Process a b -> a -> b
```

for process instantiation. The context `Transmissible a` ensures that functions for the transmission of values of type `a` are available. In the equation

```
(out_1,..., out_n) = p # (inexp_1,..., inexp_m)
```

a process abstraction `p` is instantiated with a tuple of input expressions, yielding a tuple of outputs.

Example 2. The higher-order function `map`

```
map      :: (a -> b) -> [a] -> [b]
map f xs =  [f x | x <- xs]
```

can be lifted to a parallel setting by using a process abstraction instead of a function as first parameter. This process abstraction is instantiated with each element of the list of input data yielding a list of results:

```
parMap      :: (Transmissible a, Transmissible b) =>
               Process a b -> [a] -> [b]
parMap p xs =  [p # x | x <- xs]
```

The function `parMap` can be combined with `mkProc` to construct a parallel map which has (apart from the type context) the same type as `map`:

```
map_par :: (Transmissible a, Transmissible b) =>
           (a -> b) -> [a] -> [b]
map_par =  parMap . mkProc
```

Preserving the original type of `map` makes it easy to parallelise sequential programs by replacing appropriate occurrences of `map` by `map_par`. ◁

A predefined nondeterministic process `merge` is provided for *many-to-one communication* in process systems. The `merge` process takes a list of input streams and merges the incoming values in the order in which they arrive into a single output stream.

Evaluating Processes. Each Eden process evaluates its output expressions to normal form and sends the results on the corresponding communication channels. Lists are transmitted as streams. This means that in two respects additional demand is introduced in favour of parallelism:

1. Evaluation to normal form is used for process outputs instead of evaluation to weak head normal form (WHNF).
2. Communication is not demand-driven. Values are sent to the receiver process without that the latter has to request for them. In general terms, Eden employs pushing instead of pulling of information.

To achieve an early instantiation of a process system it is necessary to impose an appropriate demand on the expression describing the process system [13]. This can be achieved by using *evaluation strategies* [21]. A `parMap` version which eagerly creates all its processes when evaluated to WHNF uses the strategy `spine` to force the evaluation of the result list's spine:

```
parMap      :: (Transmissible a, Transmissible b) =>
               Process a b -> [a] -> [b]
parMap p xs =  [p # x | x <- xs] `using` spine
```

3 Implementation Skeletons

Often a given algorithm can be parallelised in many ways. The most suitable parallelisation depends heavily on the kind of inherent parallelism (granularity) and the characteristics of the parallel machine executing the program. In this section we identify three alternative parallelisations of the higher-order function `map`. The parallelisations are defined in a machine-independent way as *implementation skeletons*, a new concept we introduce in the following.

An *algorithmic skeleton* is a higher-order scheme that abstracts from the details of a (parallel) algorithm and defines the general pattern of a (parallel) computation [5]. Well-known skeletons are e.g. the divide-and-conquer scheme and the data-parallel process farm. In skeletal programming one often distinguishes between three aspects of skeletons [9]:

– the higher-order function defining a general computation scheme with inherent parallelism (the proper algorithmic skeleton)
– different parallel implementations for different target architectures (architectural skeletons)
– a cost (performance) model to estimate the execution time.

In this paper we concentrate on the first two aspects. In our context different parallel implementations are not related to different architectures. Therefore the notion of architectural skeleton is inappropriate and we introduce the term implementation skeleton instead. An *implementation skeleton* is an architecture-independent scheme that describes a parallel implementation of an algorithmic skeleton on a higher level of abstraction than an architectural skeleton, the latter usually being optimized for a special target architecture. Often parallel implementations of an algorithmic skeleton are only described informally and hidden from the user of the skeleton (see e.g. [7, 17]). In Eden, it is possible to describe the algorithmic skeleton and its parallel implementations in the same language context. This constitutes a good basis for formal reasoning and correctness proofs, although this is not further elaborated in this paper.

In most parallel implementations of `map` the input list is seen as a task queue that can be processed using several processor elements (PEs). We have already seen a straightforward parallelisation of `map`, `parMap`, in Section 2. `parMap` creates a new process for each task. This simple approach is not always well suited, especially in the presence of fine-grained or irregular tasks. Alternative parallelisations of `map` found in the literature use a fixed number of worker processes which process a subset of tasks each. They differ in the way the tasks are distributed among the worker processes:

- *static task distribution*
  `farm`: an equal number of tasks is sent to each worker
  `direct mapping`: each worker computes the task list, and selects part of it
- *dynamic task distribution* (distribution on demand)
  `workpool`: send a new task to a worker only when it has finished a task.

A visualisation of the parallelisations `parMap`, `farm` and `direct mapping` is given in Fig. 1. These schemes can be used for regular problems with equally complex subtasks while the workpool scheme is better suited for problems where the task granularity is irregular and load balancing problems can occur.

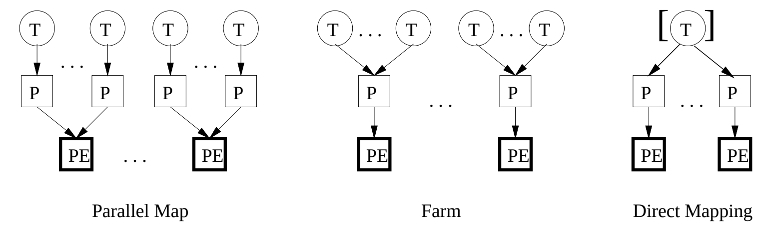


Fig. 1. Implementation skeletons for problems with regular task granularity

In the following we define implementation skeletons for these different methods of parallelising the map function. Note that the simple skeleton `map_par` has already been defined in Section 2. The original type interface of `map` is always retained, although this is not essential for implementation skeletons. It has, however, the advantage that the parallelisation of calls to `map` simply consists of replacing `map` by one of the implementation skeletons.

3.1 Static Task Distribution: Farm

While the simple `parMap` defined in Section 2 creates a new process for each task, the *farm* scheme creates only a finite number `np` of processes and assigns every process *more than one* task. It uses parameter functions `distribute` and `combine` to distribute the tasks to the processes and to combine the results. The following Eden function defines the farm scheme:

```
farm :: (Transmissible a,Transmissible b) =>
        Int -> (Int->[a]->[[a]]) -> ([[b]]->[b]) ->
        Process [a] [b] -> [a] -> [b]

farm np distribute combine proc tasks
     =  combine (parMap proc (distribute np tasks))
```

This definition is very general as it takes the number of processes and functions for distributing the tasks and combining the results as parameters. The process creation is done using `parMap`. A round robin task distribution is defined by the function `unshuffle` which on n PEs, assigns every n-th task (starting with offset i) to the i-th process (see Fig. 2). Assuming regular task granularities or at least a random distribution of task granularities, and assuming that the number of tasks is much larger than the number of processes, every process gets a representative set of tasks and an unequal work distribution is avoided. Note that `unshuffle` is incremental in the sense that it supplies every process with at least a part of its input as soon as possible and its definition allows for a parallel access to the sublists. The function `shuffle` in Fig. 2 is a counterpart of the function `unshuffle` which collects the results and restores the original order. It also works incrementally. When the order of the list elements need not be maintained, one can apply *faster* merging functions or even Eden's nondeterministic fair `merge` process.

```
unshuffle          :: Int -> [a] -> [[a]]
unshuffle n  xs  =  [ takeEach n (drop i xs) | i <- [0..n-1] ]
  where takeEach :: Int -> [a] -> [a]
        takeEach n []     = []
        takeEach n (x:xs) = x : takeEach n (drop (n-1) xs)

shuffle     :: [[a]] -> [a]
shuffle []  =  []
shuffle xxs =  (map head xxs) ++ shuffle (map tail xxs)
```

Fig. 2. Distribution and Composition Functions

Another implementation skeleton for `map` can now be defined using the `farm` scheme and the round robin task distribution defined by `unshuffle`. The `farm` scheme is applied to the number `noPE` of processor elements available, where `noPE` is a constant provided by the Eden runtime system. The process placement in the Eden system will place one process on every processor element.

```
map_farm :: (Transmissible a,Transmissible b) =>
            (a -> b) -> [a] -> [b]
map_farm =  (farm noPE unshuffle shuffle) . (mkProc . map)
```

There are two differences to the simple parallelisation shown in Section 2: Firstly, the total number of processes is fixed and set to the number `noPE` of processor elements. By bundling tasks this scheme increases the granularity and creates less processes than the *parMap* approach, saving process creation and communication costs. Secondly, *farm* can save memory compared to the simple parallelisation. If there are data structures which are used by all tasks, then *parMap* will have multiple copies of the data per PE[2], while in the *farm* approach there will only be a single copy on each PE.

In the following we discuss the possibility of replacing the communication of tasks by the recomputation of the task list in each process. We call this scheme *direct mapping*, because the original computation is simply divided into as many parts as there are PEs and the parts are directly mapped onto the PEs.

3.2 Static Task Distribution: Direct Mapping

Like `farm`, the direct mapping scheme creates one process per PE each working on a subpart of the task list. But each process lazily evaluates the whole task list and selects its own subpart of it. This local task list evaluation and selection saves a considerable amount of communication. The price one has to pay is the partial recomputation of the task list by each process. But especially on systems with high communication costs, this putative overhead pays off. Replacing communication by recomputation is a well-known parallel programming technique. The direct mapping scheme makes this technique available to parallel functional programmers in an easy to handle way.

The trick is to pass the task list to the processes not as input but as a parameter. Each process can then compute the part of the task list determined by its unique process identifier `pid` and the total number of processes `np`:

```
dm :: (Transmissible b) =>
      Int -> (Int->[a]->[[a]]) -> ([[b]]->[b]) ->
      ([a]->Process () [b]) -> [a] -> [b]

dm np distribute combine proc tasks
   =  combine [ proc (extract pid np tasks) # ()
              | pid <- [0..np-1] ] `using` spine
   where extract i np ts = (distribute np ts) !! i
```

A corresponding direct mapping implementation skeleton for `map` which uses the functions `unshuffle` and `shuffle` for distributing tasks and combining results is defined by:

[2] Sometimes this copying can be avoided by declaring the data at top level. In this special case the compiler, as an extension of the Glasgow Haskell Compiler (GHC), will keep only one copy of the data in each PE.

```
map_dm  :: (Transmissible b) =>
           (a -> b) -> [a] -> [b]
map_dm  =  (dm noPE unshuffle shuffle) . (rfi . map)

rfi     :: (Transmissible b) =>
           (a -> b) -> a -> Process () b
rfi f x =  process () -> (f x)
```

Similar to `mkProc`, the function `rfi` (*remote function invocation*) maps a function to a process abstraction. The function argument is however not communicated as in `mkProc` but passed as a parameter to the process abstraction. As in the `farm` scheme the tasks have to be of equal complexity to avoid a load imbalance. Tasks of different complexity can be dealt with by employing a workpool which is described next.

3.3 Dynamic Task Distribution: Workpool

For so called *irregular parallelism*, i.e. tasks of different granularity, we need an approach different to the ones presented before, which were static in the sense that every process was working on a predefined subset of tasks. To cope with extremely variable task granularities one usually employs a *workpool* with a set of tasks which are dynamically assigned to free worker processes. In our version, every PE hosts one worker process, which iteratively 1) receives a task from the workpool, 2) computes the result and 3) sends back the result. The results are interpreted as new requests for work and composed to produce the overall result. The flow of data together with their types is shown in Fig. 3.

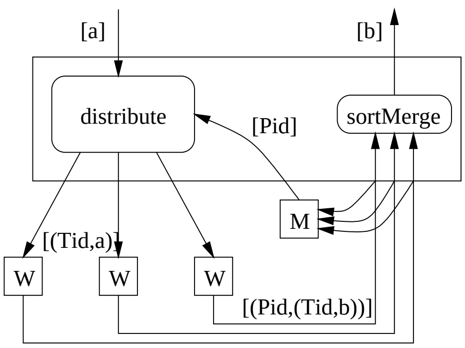


Fig. 3. The workpool scheme

The Eden program which implements the workpool is shown in Fig. 4. The `tasks` are spread across the worker processes using the `distribute` function defined in Fig. 5. The distribution depends on which workers have delivered the result of their previous computation. Initial requests `initialReqs` are generated depending on the `prefetch` argument to avoid workers from being idle while waiting

```
type Pid = Int  -- Process Ids
type Tid = Int  -- Task Ids
workpool :: (Transmissible a, Transmissible b) =>
            Int -> Int -> (Pid -> Process [(Tid,a)] [(Pid,(Tid,b))])
            -> [a] -> [b]
workpool np prefetch proc tasks
   = sortMerge (map (map (\ (pid,res) -> res)) fromWorkers)
     where fromWorkers   = [ proc pid # ts | (pid,ts) <- toWorkers]
           toWorkers     = zip [0..np-1] taskQueues
           taskQueues    = distribute np requests numberedtasks
           numberedtasks = zip [0..] tasks
           requests      = initialReqs ++ newReqs
           initialReqs   = concat (replicate prefetch [0..np-1])
           newReqs       = merge # (map (map (\ (pid,res) -> pid))
                                                 fromWorkers)
```

Fig. 4. Workpool

to receive a new task. For the programs measured to date, a prefetch argument of 2 has been sufficient. More elaborate versions for this workpool scheme with dynamic task creation and termination detection have been presented in [14]. Using the workpool process scheme of Fig. 4 one can define the following implementation skeleton for `map`:

```
map_wp  :: (Transmissible a,Transmissible b) =>
           (a -> b) -> [a] -> [b]
map_wp  =  (workpool noPE 2) . workerProc

workerProc  :: (a -> b) -> Int -> Process [(Int,a)] [(Int,(Int,b))]
workerProc f pid
     =  process numberedTasks -> map solveTask numberedTasks
        where solveTask (nr,t) = (pid, (nr, f t))
```

Note that the workpool scheme is not only worthwhile for irregular problems but also when working on non-uniform systems with processors of different speed. For our measurements we used however only uniform systems with equally equipped processors.

3.4 Increasing Task Granularity

A well-known technique to increase the granularity in fine-grained problems is to put several tasks together to form *chunks* of tasks or *macro-tasks*. Working with coarse-grained macro-tasks instead of fine-grained tasks reduces the communication overhead substantially and thus improves the runtime behaviour on high-latency distributed systems. The following function embeds implementation skeletons into ones with increased task granularity.

```
distribute                   :: Int -> [Int] -> [a] -> [[a]]
distribute np rs     []     =  replicate np []
distribute np []     ts     =  unshuffle np ts
distribute np (r:rs) (t:ts) =  (take r tqs) ++
                               ((t:(tqs!!r)) : (drop (r+1) tqs))
                               where tqs =  distribute np rs ts

sortMerge     :: [[(Int,a)]] -> [a]
sortMerge xss =  sm (\ x y -> (fst x) < (fst y)) (sortMerge odds)
                                                 (sortMerge evens)
  where [odds,evens]         =  unshuffle 2 xss
        sm                   :: (a->a->Bool) -> [a] -> [a] -> [a]
        sm  p  []     ys     =  ys
        sm  p  xs     []     =  xs
        sm  p  (x:xs) (y:ys) =  if (p x y) then (x : sm p xs (y:ys))
                                else (y : sm p (x:xs) ys)
```

Fig. 5. Auxiliary functions for the workpool

```
macro :: Int -> (Int -> [a] -> [[a]]) -> ([[b]] -> [b])
             -> (([a] -> [b]) -> [[a]] -> [[b]])
             -> (a -> b) -> [a] -> [b]
macro size decompose compose mapscheme f xs
      =  compose (mapscheme (map f) (decompose size xs))
```

The parameter function `decompose` is meant to divide the input list into subparts of a given `size`. The parameter function `compose` should be the counterpart of `decompose`, i.e. the equation `compose (chunk size xs) == xs` should be valid for finite lists `xs`. Especially the `parMap` scheme can take profit from macro-tasks as the number of processes is reduced to the number of macro-tasks:

```
map_par_macro = macro size chunk concat map_par
  where chunk :: Int -> [a] -> [[a]]
        chunk k [] = []
        chunk k xs = (take k xs) : chunk k (drop k xs)
```

For an input list with length k this scheme will produce $\lceil k/\texttt{size}\rceil$ processes where `size` should be defined globally. Using `unshuffle` for chunking, although `unshuffle` will not produce chunks of the given `size` but `size` chunks, and `shuffle` for combining will make the macro task version of `parMap` identical to the `farm` scheme:

`macro noPE unshuffle shuffle map_par = map_farm`.

4 Experimental Results

Eden has been implemented by extending the Glasgow Haskell Compiler, version 3.02. The Eden runtime system consists of a set of multi-threaded abstract machines, each of which is mapped to a separate processor element (PE) on a multi-computer system [2]. The system can be used with the message passing libraries MPI (Message Passing Interface) [20] or PVM (Parallel Virtual Machine) [8]. A special Eden prelude module has been constructed which implements much of Eden's functionality in Eden itself.

In the following we will explore the implementation skeletons in the context of several examples on a network of workstations (NOW) and discuss their efficiency. Afterwards we show a few additional measurements on a Beowulf cluster[3] located at the University of St. Andrews. The two systems, which both represent low-cost high-performance parallel computing platforms, have the following characteristics:

System	CPU (MHz)	OS	Mem	#Nodes	Ethernet	Latency
NOW	UltraSparc IIi (300)	Solaris 2.6	128MB	6	10MBit	$282\mu s$
Beowulf	Pentium II (450)	Linux RH 6.2	384MB	64	100MBit	$142\mu s$

The latency on both systems is relatively high, favouring coarser grained parallel programs. All experiments (including determination of the latencies) were done under PVM 3.4.2. To rule out speedup gains due to the usage of more memory in the parallel setting (less garbage collections compared to the sequential version), we used only 40 MB on each processor node. This keeps the time for garbage collection under 5% of the sequential runtime.

The examples have been compiled with full optimization (flag `-O2`). We show the runtimes of different parallelisations and give the *absolute speedups*, i.e. comparing parallel runtimes with the runtime of the corresponding purely sequential programs containing no parallel machinery. The runtimes are given as the average over five runs to rule out erroneous measurements due to temporary network traffic caused by other services. We compare the effectiveness of the implementation skeletons for tasks with equal or varying granularity. A more complete presentation of the experiments described in this section will be contained in the forthcoming thesis [12].

4.1 Regular Tasks: a Simple Ray Tracer

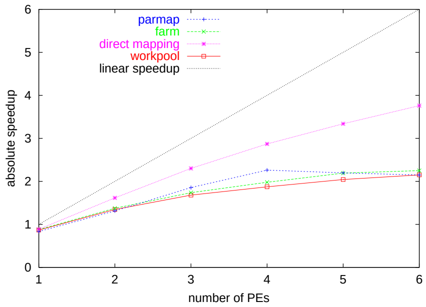
Given a scene consisting of 3D objects and a camera, a ray tracer calculates a 2D image of the scene. For every pixel of the output image, the ray tracer shoots a ray into the scene and tests whether it impacts with any object of the scene. In this simple version object properties like reflection and transparency are neglected. Therefore a pixel gets the color of the object its corresponding ray hits first; if it hits none at all, it gets the color of the background. The scene itself is maintained as an unordered list of objects, so that the effort to compute the nearest impact with the objects is nearly the same for all rays. The main function of the ray tracer is `top`:

```
top :: ScreenSize -> CPos -> [Objects] -> [Impact]
top detail cameraPos scene
  = map (firstImpact scene) allRays
    where allRays = generateRays detail cameraPos
```

[3] EPSRC Research Grant No. GR/M 32351

This simple algorithm contains fine-grained parallelism with tasks of the same time complexity and depends only on medium-sized globally shared data (the scene). The top level parallelism is inherent in the call of the function `map` which can be parallelised using the four implementation skeletons introduced in the previous sections.

Results. For the runtime measurements we used a screen of 400×400 pixels with a scene consisting of 17 objects, so the shared data was relatively small. From the runtimes and absolute speedups (shown in Fig. 6) we see that the *direct mapping scheme* performs best. The reason is that for this version only half of the communication is needed, as the rays for each process are given in a compact format.



runtimes (in seconds)				
# PEs	parmap	farm	dm	workpool
1	65.36	64.58	64.02	67.96
2	42.13	41.26	35.10	43.20
3	33.76	32.64	24.64	30.56
4	30.28	28.65	19.76	25.08
5	27.76	25.88	16.98	25.80
6	26.38	25.20	15.08	26.40

sequential runtime: 54.78s

Fig. 6. Absolute speedups and runtimes of ray tracer on the NOW

All the other schemes take nearly the same time. *parMap* performs worst, as for each task we have to copy the scene within the process abstraction. The difference is not big, as the scene is still small. The *workpool* (with prefetch 2) on one PE is slower than the other schemes due to additional computations caused by the sequence numbers added to the tasks to be able to reconstruct them in the correct order. Moreover, more work for the task distribution is done in this scheme.

4.2 Irregular Tasks: An Exact Linear System Solver

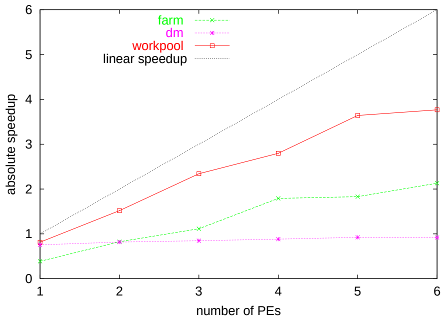
The algorithm LinSolv discussed in this section finds an exact solution of a linear system of equations of the form $Ax = b$ where $A \in \mathbb{Z}^{n\times n}, b \in \mathbb{Z}^n$. LinSolv uses a *multiple homomorphic images* approach which consists of the following three stages: map the input data into several homomorphic images; compute the solution in each of these images; and combine the results of all images to a result in the original domain ('lifting'). The original `LinSolv` program was written by Hans-Wolfgang Loidl [15].

The advantage of this approach is that calculations over arbitrary precision integers are quite expensive, thus the main computation can be performed cheaper in the smaller domains $\mathbb{Z}_p$ ($\mathbb{Z}$ modulo p), where it is possible to work with the standard integers. Each $\mathbb{Z}_p$ is determined by a different prime number p. The computations on these domains can be performed in parallel and be expressed in a map-like fashion. Cramer's Rule was chosen as the basic linear system solver as it computes the exact solution and provides again a high potential for parallelism. The solution is computed by

$$x_{pj} = \det A'_{pj} / \det A_p, \quad j = 1, \ldots, n$$

where the matrix A'_{pj} is A_p (A modulo p) with the j-th column replaced by vector b_p (b modulo p). Each of the determinants can be computed independently. In the algorithm we have to take care that we do not take prime numbers for which $\det A_p$ is zero. We need to filter out these *unlucky* prime numbers after computing the determinant in the homomorphic image. This results in tasks with two different granularities: small ones, only computing $\det A_p$, and big ones, computing n determinants more.

Results. The measurements were taken for linear systems of various sizes, with elements not greater than 2000. As an example we show the results for a linear system with dimension 8. For this system we get only few tasks as 21 lucky primes are sufficient to find the exact solution. If there are more PEs than tasks it would be necessary to also parallelise the computation of the homomorphic solutions, e.g. by computing the determinants in parallel. The runtimes and speedups are given in Fig. 7.



runtimes (in seconds)			
#PEs	farm	dm	workpool
1	240.96	123.84	115.04
2	113.54	114.60	61.70
3	84.04	110.58	40.00
4	52.30	106.12	33.48
5	51.18	101.48	25.72
6	43.96	101.92	24.86

sequential runtime: 93.66s

Fig. 7. Absolute speedups and runtimes of LinSolv on the NOW

Although there are only few tasks, the *workpool* version delivers the best results. This has two reasons: first of all because the load can be balanced better, and what is probably more significant is that not too many primes are generated without real demand. The runtimes of the *farm* on one processor are unusually high, because the main process produces prime numbers without demand, and this thread gets the same amount of time as the local process solving the equations and the thread computing the final lifting phase. The problem of the *direct mapping* version is the recomputation of the prime list in each worker process.

4.3 Compensating Irregularity: Mandelbrot Sets

Mandelbrot sets can be used to generate fractal graphics. For each pixel of the screen, it is necessary to compute its color (in fact, its distance to the Mandelbrot set), which depends on the coordinates of the pixel. This computation is performed for each pixel by iterating a function until a good approximation is obtained, or until a maximum number of iterations has been reached. For some pixels it is sufficient to compute a few iterations, while for others the maximum number of iterations is needed. Therefore the problem's inherent pixel-wise parallelism is of irregular granularity.

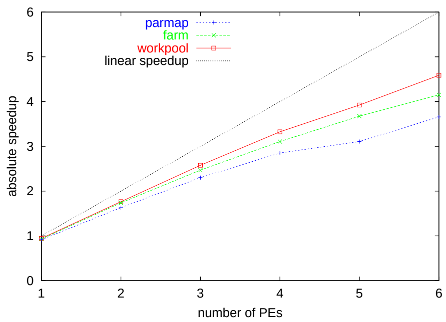
```
mandel :: Int -> Int -> [Int] -- Using a (sizeX x sizeY) screen
mandel sizeX sizeY = map calcPix (range (0,0) (sizeX-1,sizeY-1))

range  :: (Int,Int) -> (Int,Int) -> [(Int,Int)]
range (x1,y1) (x2,y2) = [(x,y) | x <- [x1..x2], y <- [y1..y2]]

calcPix       :: (Int,Int) -> Int
calcPix (x,y) =  loop (firstApprox (x,y)) 0
```

The much too fine granularity has to be coarsened by grouping pixels into larger tasks. Time-consuming pixels may be clustered with this approach, which bears the risk of a serious load imbalance.

Results. We have generated Mandelbrot sets for a 1024×1024 screen, using 1000 as maximum number of iterations. Fig. 8 shows the results on the NOW.



runtimes (in seconds)			
# PEs	parmap	farm	workpool
1	235.00	231.10	227.92
2	131.94	123.74	121.92
3	93.42	87.13	83.44
4	75.44	69.20	64.70
5	69.22	58.54	54.83
6	58.80	51.82	46.88

sequential runtime: 215.06s

Fig. 8. Absolute speedups and runtimes of Mandelbrot on the NOW

As expected, even when using a row-wise partitioning the *workpool* scheme is the best suited for the original inherently irregular problem. But *parMap* and *farm* can also produce good speedups if we use a different partitioning. When rows are combined with `unshuffle` these schemes can be nearly as good as *workpool*. The reason is that the round robin distribution with `unshuffle` often compensates the irregularity of the rows. There is a big difference in the computation needed for different rows, but neighbouring rows usually need similar times. Hence, unshuffling the tasks leads to bigger tasks of a more regular granularity, giving *workpool* only a small advantage over the other schemes.

4.4 Measurements on a Beowulf Cluster

We had the opportunity to repeat some of our measurements on the St. Andrews Beowulf cluster. As can be seen from the absolute speedups achieved on up to ten PEs for the ray tracer with the same input parameters as before (see Fig. 9) and for the LinSolv example with the 8×8 input matrix on up to 25 PEs (see Fig. 10), the implementation skeletons show the same relative performance in comparison to each other. The direct mapping scheme performs best for the regular tasks of the ray tracer, while the workpool scheme copes best with the irregularity within the LinSolv program.

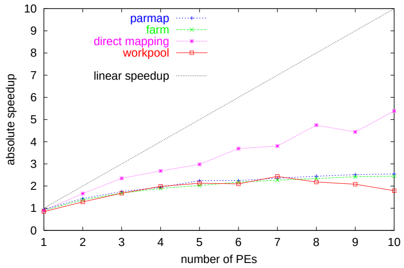


Fig. 9. Absolute speedups of the simple ray tracer on the Beowulf cluster

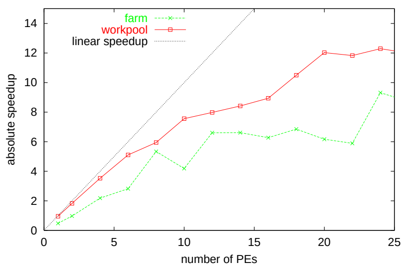


Fig. 10. Absolute speedups of LinSolv with 8×8 input matrix on the Beowulf cluster

4.5 Summary

In this section we have considered examples with inherent data parallelism. The problems can be distinguished by the number of tasks and their granularity. We had two problems with fine-grained parallelism: the ray tracer and the Mandelbrot sets. LinSolv is coarse-grained as there are only a few expensive tasks. To observe the impact of load balancing we had two problems with varying task granularity: In Mandelbrot between 0 and 1000 iterations are needed per pixel, whereas in LinSolv there are two different complexities. The complete characterization is shown in Table 1.

problem	granularity	decomposition	best scheme
mandelbrot	fine	irregular	workpool
raytracer	fine	regular	direct mapping
linsolv	coarse	irregular	workpool

Table 1. Characterization of the problems

The *workpool* is the best choice if the task granularity is varying, although the maintenance of the *workpool* is not for free. If all tasks have the same granularity the *direct mapping* yields the best results, because we can save communications by implicitly passing the task description as a parameter. As a consequence of increasing the task granularity by the use of macro-tasks in `parMap`, the *parMap* and *farm* schemes show a very similar behaviour. The results obtained on the Beowulf cluster show that the direct mapping and the workpool scheme behave equally well on large numbers of processor elements.

4.6 Further Measurements on the Beowulf Cluster

Additional measurements on the St. Andrews Beowulf cluster with up to 61 processor nodes have shown that Eden achieves a good scalability for complex problems. We have studied a more complex ray tracer for spheres and the LinSolv program explained before. The spheres ray tracer differs from the ray tracer described in Subsection 4.1 in that it is restricted to spherical objects, but takes into account reflection and light conditions. The original version of this algorithm has been ported from the Impala suite of implicitly parallel programs[4].

Spheres Ray Tracer Results. Fig. 11 shows the speedups obtained for the ray tracer program with a direct mapping parallelisation for screens with dimension 350 (sequential runtime: 213.42s) and 500 pixels (sequential runtime: 435.01s). The largest speedup is 27 on 37 PEs for the larger screen size. The speedup increases almost linearly until about 40 processor elements. Then a dramatic decrease is observed which is due to an increase of the startup time of the Eden system on the Beowulf cluster. The startup problems on more than 40 processors have not been solved yet. Nevertheless these results are very promising for what is a relatively immature implementation.

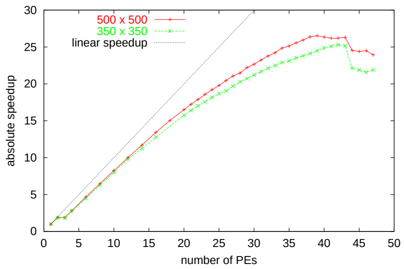


Fig. 11. Absolute speedups of spheres ray tracer on the Beowulf cluster

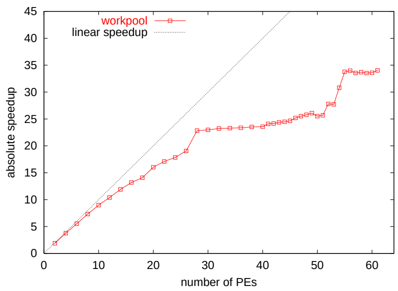


Fig. 12. Absolute speedups of LinSolv with 14 × 14 input matrix on the Beowulf cluster

LinSolv Results. Fig. 12 shows the speedups obtained for the LinSolv algorithm described in Subsection 4.2. The measurements used a sparse matrix with dimension 14 (sequential runtime: 552.59s) and a workpool parallelisation. Measurements were taken for up to 61 PEs and show good performance: The largest speedup is 35 on 61 PEs.

[4] URL: `http://www.csg.lcs.mit.edu/impala/`

5 Related Work

In the last decade a great variety of different approaches to the combination of functional programming and parallelism has evolved. A comprehensive collection of introductory chapters and surveys of current research projects has been published by Hammond and Michaelson [10]. Besides many other approaches to efficient parallel functional programming the concept of *algorithmic skeletons* [5] has received a lot of attention. For efficiency, skeletons are often implemented in a low-level language different from the language in which they are used. Therefore skeleton implementations tend to be a "black box" for the programmer, which can make them hard to use in an efficient way. Well-known approaches to introduce skeletons in a parallel language include: *Darlington et al.* [6], P^3L [17], *Skil* [1], and others. As Eden, Skil allows to design new skeletons in the language itself, the main difference being that Skil is based on the imperative host language C [19]. More closely related to our work are the following approaches:

In *GAPML* [9] Michaelson, Hamdam et al. extend an ML compiler by machinery which automatically searches the given program for higher-order functions which are suitable for parallelisation. During compilation these are replaced by efficient low-level implementations written in C and MPI. In *HaskSkel* [11], Hammond and Rebón Portillo combine the evaluation strategies of *GpH* [21] with Okasaki's Edison library [16] (which provides efficient implementations of data structures) to implement parallel skeletons directly in GpH.

6 Conclusion

We have introduced four implementation skeletons for the higher-order function `map` which are based on the schemes *parMap*, *farm*, *direct mapping*, and *workpool*. Depending on problem characteristics like degree and granularity of inherent parallelism, static or dynamic evolution of parallelism etc. the programmer can choose the appropriate skeleton to obtain an efficient parallel program. The schemes have proven to be helpful during program construction and efficient during program execution.

This is supported by runtime results measured with the Eden system on two different high-latency parallel machines. The current implementation contains the basic functionality without optimisations, but it is already mature enough to deliver reasonable speedups with several examples on a conventional network of workstations and on a Beowulf cluster.

Acknowledgements

The authors thank Ricardo Peña, Kevin Hammond, Hans-Wolfgang Loidl, Phil Trinder and Robert Pointon for their collaboration and helpful comments on previous versions of this paper. Special thanks go to Kevin Hammond and Álvaro Rebón Portillo for giving us the opportunity to work with the St. Andrews Beowulf Cluster.

References

1. G. H. Botorog and H. Kuchen. Efficient Parallel Programming with Algorithmic Skeletons. In *Europ. Conf. on Par. Proc. (Euro-Par)*, LNCS 1123. Springer, 1996.
2. S. Breitinger, U. Klusik, and R. Loogen. From (sequential) Haskell to (parallel) Eden: An Implementation Point of View. In *Principles of Declarative Programming, PLILP'98*, LNCS 1490. Springer, 1998.
3. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden — Language Definition and Operational Semantics. TR 96-10, Philipps-Univ. Marburg, 1996.
4. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden Coordination Model for Distributed Memory Systems. In *High-Level Parallel Progr. Models and Supportive Env. (HIPS)*. IEEE Press, 1997.
5. M. Cole. *Algorithmic Skeletons: Structure Management of Parallel Computations.* MIT Press, 1989. Research Monographs in Parallel and Distributed Computing.
6. J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel Programming Using Skeleton Functions. In *Parallel Architectures and Languages Europe*. Springer, 1993.
7. J. Darlington, Y.-K. Guo, H. To, and J. Yang. Functional Skeletons for Parallel Coordination. In *Euro-Par*, LNCS 966. Springer, 1995.
8. A. Geist, A. Beguelin, J. Dongarra, and W. Jiang. *PVM: Parallel Virtual Machine.* MIT Press, 1994.
9. M. Hamdan. *A Combinatorial Framework for Parallel Programming Using Algorithmic Skeletons.* PhD thesis, Heriot-Watt Univ. Edinburgh, 2000.
10. K. Hammond and G. Michaelson. *Research Directions in Parallel Functional Programming.* Springer-Verlag, 1999.
11. K. Hammond and Á. R. Portillo. HaskSkel: Algorithmic Skeletons in Haskell. In *Impl. of Functional Languages (IFL'99)*, LNCS 1868. Springer-Verlag, 2000.
12. U. Klusik. *An Efficient Implementation of the Parallel Functional Language Eden on Distributed-Memory System.* PhD thesis, University of Marburg, 2001. In prep.
13. U. Klusik, R. Loogen, and S. Priebe. Controlling Parallelism and Data Distribution in Eden. In *Trends in Functional Programming, Vol. 2, (SFP 2000)*. Intellect, 2001.
14. U. Klusik, R. Peña, and F. Rubio. Replicated Workers in Eden. *Workshop on Constructive Methods for Par. Progr. (CMPP)*. TR-MIP-0007, Passau Univ., 2000.
15. H. W. Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Dept. of Computing Science, University of Glasgow, 1997. TR-1998-7.
16. C. Okasaki. An Overview of Edison. In *ICFP 2000 (Haskell Workshop)*, 2000.
17. S. Pelagatti. *Structured Development of Parallel Programs.* Taylor & Francis, 1998.
18. J. Peterson and K. Hammond (eds.). Report on the programming language Haskell: a non-strict, purely functional language, version 1.4. TR YALEU/DCS/RR-1106, Yale Univ., 1997.
19. T. Richert. Skil: Programming with Algorithmic Skeletons — A Practical Point of View. In *Proc. of the 12th Int. Workshop on Impl. of Funct. Lang. (IFL'00)*, Aachener Informatik-Berichte 00-7, RWTH Aachen, 2000.
20. The MPI Forum. MPI: A Message-Passing Interface Standard. TR, Univ. of Tennessee, Knoxville, 1994.
21. P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), 1998.