A Skeleton for Distributed Work Pools in Eden

Mischa Dieterle[1], Jost Berthold[2], and Rita Loogen[1]

[1] Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans Meerwein Straße, D-35032 Marburg, Germany
{dieterle,loogen}@informatik.uni-marburg.de
[2] Datalogisk Institut, University of Copenhagen, Denmark
berthold@diku.dk

Abstract. We present a flexible skeleton for implementing distributed work pools in our parallel functional language Eden. The skeleton manages a pool of tasks (work pool) in a distributed manner using a demand-driven work stealing approach for load balancing. All coordination is done locally within the worker processes. The latter are arranged in a ring topology and exchange additional channels to shortcut communication paths. The skeleton is suited for different types of algorithms, namely simple data parallel ones and standard tree search algorithms like backtracking, and using a global state as needed for branch-and-bound. Runtime experiments reveal a stable runtime behaviour for the different algorithm classes as illustrated by activity profiles (timeline diagrams). Acceptable speedups can be achieved with low effort.

1 Introduction

Parallel evaluation of a large and dynamically evolving pool of tasks (a *work pool*) is a classical parallelisation problem [Fos95]. The common approach is a system with one master process managing the work pool, and a set of worker processes which process the tasks. The master distributes tasks to the workers and collects both the results and possibly created tasks produced and sent by the workers.

With a big number of workers, such a *master-worker* setup quickly leads to a bottleneck in the master process. Consequently, more sophisticated work pool schemes have been proposed, with a focus on optimizing the task-scheduling strategy [Fos95,GGKK03,Qui03]. In Fig. 1, we classify such task-scheduling approaches according to their work allocation policy, the organisation of the work pool and the task distribution strategy.

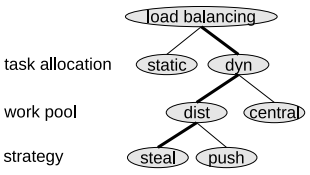


Fig. 1. Classification of Task Scheduling Approaches

Work can be allocated statically or dynamically. While a static scheme certainly reduces the communication overhead, it may lead to load imbalance in the presence of highly irregular tasks or differences in worker performance. For this reason, a *dynamic work allocation strategy* is generally favourable. In contrast

M.Blume, N.Kobayashi, and G.Vidal (Eds.), FLOPS 2010, LNCS 6009, pp.337–353, 2010.
© 2010 Springer-Verlag Berlin/Heidelberg
The original publication is available at www.springerlink.com.

Dieterle, Berthold, Loogen. A Skeleton for Distr. Work Pools in Eden. In:FLOPS'10, LNCS 6009.
© 2010 Springer Berlin/Heidelberg. The original publication is available at www.springerlink.com.

to the classical centralised master-worker scheme, a completely *distributed task pool* avoids the single hot spot in the system (but requires more sophisticated work distribution mechanisms and termination detection) [Qui03]. The master process' role reduces to setting up the system and collecting the results. In such a distributed work pool, a basic distinction can be made between task *pushing* and *stealing* approaches [Qui03]. A work pushing strategy means to speculatively forward surplus tasks to random peers when the amount of local tasks exceeds a given threshold. In a demand-driven work stealing strategy, workers send work request messages to peers when idle. New tasks created by workers can be kept locally until work requests from other workers arrive.

In the following, we present a sophisticated functional implementation of a work pool skeleton where the work pool is managed in a *distributed* manner, and a demand-driven work stealing approach is used for load balancing. All coordination takes place between the worker processes, the master only collects the results. As in [PK06], the worker processes are arranged in a ring topology. This provides an easy way to traverse the whole setup for termination detection, and is also an acceptably fast interconnect for propagating global information. Additional channels are used at runtime to directly pass tasks and requests to peer workers without using the ring. Our paper shows that complex coordination structures can efficiently be implemented in a functional setting yielding a flexible base for a low-effort parallelisation of various algorithm classes. The skeleton is especially useful for solving combinatorial optimisation problems with backtracking or branch-and-bound algorithms. Experiments show stable runtime behaviour for several algorithm classes as illustrated by activity profiles. Our traces show well-balanced workloads. Runtime measurements reveal a good scalability with respect to the number of processors.

After a short introduction to Eden in Section 2, the skeleton is described in Section 3. First we explain the skeleton interface and how to adapt and apply it to typical algorithm classes. Then the functional implementation of the distributed work pool skeleton is presented. Section 4 shows experimental results for two case studies. Related work is discussed in Section 5. The paper ends with conclusions.

2 Eden in a Nutshell

The distributed work pool skeleton has been implemented in the parallel Haskell dialect Eden [LOMP05], which extends Haskell with an explicit notion of processes (function applications evaluated remotely in parallel). The programmer has direct control over evaluation site, process granularity, data distribution and communication topology, but does not have to manage synchronisation and data exchange between processes. The latter are performed by the parallel runtime system through implicit communication channels, transparent to the programmer.

The essential two coordination constructs of Eden are process abstraction and instantiation:

Dieterle, Berthold, Loogen. A Skeleton for Distr. Work Pools in Eden. In:FLOPS'10, LNCS 6009.
© 2010 Springer Berlin/Heidelberg. The original publication is available at www.springerlink.com.

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

The function `process` embeds functions of type `(a -> b)` into *process abstractions* of type `Process a b` where the context `(Trans a, Trans b)` states that both `a` and `b` must be types belonging to the `Trans` class of transmissible values. Evaluation of an expression `(process funct) # arg` leads to the creation of a new process for evaluating the application of the function `funct` to the argument `arg`. The type class `Trans` provides overloaded communication functions for *lists*, which are transmitted as streams, i.e. element by element, and for *tuples*, which are evaluated componentwise by concurrent threads in the same process. An Eden process can thus contain a variable number of threads during its lifetime.

Two additional non-functional features of Eden are essential for performance optimisations: nondeterministic stream merging and explicit communication. Eden's non-deterministic function `merge :: Trans a => [[a]] -> [a]` merges a list of streams into a single stream. It simplifies the specification of control and coordination. Communication channels may be created implicitly during process creation - in this case we call them *static channels* - or explicitly during process evaluation. In the latter case we call them *dynamic channels*. The following functions provide the interface to create and use dynamic channels:

```
new     :: Trans a => (ChanName a -> a -> b) -> b
parfill :: Trans a =>  ChanName a -> a -> b  -> b
```

Evaluating `new (\ name val -> e)`, a process creates a dynamic channel `name` of type `ChanName a` in order to receive a value `val` of type `a`. After creation, the channel should be passed to another process (just like normal data) inside the expression result `e`, which will as well use the eventually received value `val`. Because of Haskell's lazy evaluation, the execution will not block on `val` until that value is actually needed. Evaluating `(parfill name e1 e2)` in the other process has the side-effect that a new thread is forked to concurrently evaluate and send the value `e1` via the channel. The overall result of the expression is `e2`.

In the skeleton, dynamic channels are used to create the ring connections between the processes, as well as shortcut connections between ring processes. The latter are used to bypass (previously) idle workers when sending a new work request and when returning tasks to a requesting worker process. Eden's nondeterministic `merge` function is heavily used to ensure that incoming data can be processed as soon as it is available.

3 Skeleton Definition

The distributed work pool skeleton uses a set of workers to solve a list of initial tasks received from the caller. Each worker holds a local task pool, and maybe a local state. New tasks may be created and added while solving the initial task set. Load balancing is achieved by a demand-driven exchange of surplus tasks.

Dieterle, Berthold, Loogen. A Skeleton for Distr. Work Pools in Eden. In:FLOPS'10, LNCS 6009.
© 2010 Springer Berlin/Heidelberg. The original publication is available at www.springerlink.com.

```
mwRing :: (Trans t, Trans r, Trans s, NFData r') =>
  Int   ->                              -- no of processes
  -- task processing and result post processing
  ([(t,s)] -> [(Maybe (r',s),[t])]) -> -- worker function wf
  ([Maybe (r',s)] -> s -> [r])      -> -- result transform function resTf
  ([[r]] -> [r])                    -> -- result merge function
  -- work pool transformation
  ([t] -> [t] -> s -> [t])          -> -- attach function ttAf
  ([t] -> s -> ([t],[t]))           -> -- split  function ttSplitf
  ([t] -> s -> ([t],Maybe (t,s)))   -> -- detach function ttDf
  -- state comparison function
  (s -> s -> Bool)                  -> -- compare function cpSf
  -- initialisation
  s  -> [t]                         -> -- initial state initS/tasks initTs
  [r]                                  -- results
```

Fig. 2. Interface of the General Distributed Work Pool Skeleton

3.1 Skeleton Interface and Application

Fig. 2 shows the interface of the skeleton, which allows to customise its functionality by a large set of parameter functions. While the last two parameters provide the initial state and task list, the first parameter specifies the number of processes to be created. The skeleton creates a ring of worker processes together with a hierarchy of collector processes. The latter is used to speed-up result post-processing. Three functions determine task processing and result post processing, i.e. the proper worker functionality. The work pool is manipulated with the following three parameter functions of the general skeleton: the task pool transformation and attach function `ttAf` is used to extend the work pool with newly created tasks, the function `ttSplitf` is used to split the work pool when an external work request arrives, and the function `ttDf` detaches a single task for local evaluation. Different selection strategies can be used for serving oneself via `ttDf` and other workers via `ttSplitf`. Finally, the state comparison function is used for branch-and-bound algorithms to select the optimal solution (state).

The following table illustrates how the skeleton functionality is reduced for specific algorithm classes.

Algorithm class	**task pool size**	**post-processing**	**state**	**task pool structure**
parallel transformation (`map`)	fixed at start	no	no	queue
transformation and reduction (`map-reduce`)	fixed at start	yes	no	queue
backtracking (tree search)	dynamic	maybe	no	queue or stack
branch-and-bound (optimum search)	dynamic	yes	yes	priority queue

Dieterle, Berthold, Loogen. A Skeleton for Distr. Work Pools in Eden. In:FLOPS'10, LNCS 6009.
© 2010 Springer Berlin/Heidelberg. The original publication is available at www.springerlink.com.

To show how to parallelise a variety of common data processing patterns, we exemplarily discuss the simplest and the most involved instantiation.

Data-Parallel Transformation. The most simple and very common application of work pool skeletons is the case of a big set of data items processed by a common transformation (using a functionality like the well-known higher-order function `map::(a->b)->[a]->[b]`). In our general distributed work pool skeleton, the worker function simplifies to a transformation `(t -> r)`, since it does not create new tasks, nor does it depend on a system state or environment. We embed such a simple worker function into the type needed by our work pool skeleton using the function `staticWF` and extract the results with the result transformation function `idResTf` before they are returned to the master:

```
staticWF       :: (t->r) -> [(t,())] -> [(Maybe (r,()),[t])]
staticWF wf ts =  [ (Just (wf t,()),[]) | (t,_) <- ts ]
idResTf        :: [Maybe (r,())] -> () -> [r]
idResTf rss _  =  [ r | (Just (r,())) <- l ]
```

The data set can easily be divided and processed in parallel. However, it may have extremely varying complexity for different input data, and thereby needs dynamic load balancing between the working parallel processes. Every worker in the ring receives a subset of the (usually numerous) tasks. Load balancing becomes relevant in the end phase of the computation, when some workers might already be idle, while others still work on the remaining tasks.

Transformation and Reduction. As an immediate extension, a reduction operation (commutative and associative) can be applied to the results, yielding a `map-reduce` skeleton. The reduction is easily realised by the result transform function `resTf`: workers can pre-combine all their results after processing and send only one single result back to the caller. The latter then reduces only few pre-results:

```
mwRingMapReduce :: (Trans t, Trans r) =>
                   (t->r) -> ([r]->[r]) -> [t] -> [r]
mwRingMapReduce wf redF ts
 = mwRing (noPe-1) (staticWF wf)
    (\ rs _-> redF(idResTf rs ())) (redF . merge)
    (\ ts _ _->ts) halfTTSplit topTTD (\_ _->False) () ts
```

Parameters are the worker function `wf`, the reduce function `redF` and the task list `ts`. Note that the type of the reduce function `redF :: [r]->[r]` allows any list transformation including identity or sorting. The constant `noPe :: Int` determines the number of processing elements. The task pool transform and split strategy `halfTTSplit` passes over half of the tasks within the task pool and the taskpool transform and detach function `topTTD` selects the first task to be processed next by the local worker function (both not shown).

Data transformation/reduction and exhaustive tree search (not discussed here) can also be implemented using a hierarchical master-worker systems (as shown and analysed in [BDLP08]). Our distributed work pool skeleton is tailored

© 2010 Springer Berlin/Heidelberg. The original publication is available at www.springerlink.com.

to the more interesting case of tree search problems which look for an *optimal* solution. Note, that ordinary master-worker skeletons are in general not able to handle such optimisation problems.

Tree Search for Optimal Results (Branch-and-Bound). Branch-and-bound algorithms require an internal state (best result yet), and the comparison function of the skeleton interface to decide which branches of the decision tree should be searched further, and which can be discarded because of already known better results. The best result which has previously been found forms the global system state. This system state is included as a parameter and as a result in the worker, yielding the general type: `[(t,s)] -> [(Maybe (r,s),[t])]`. Each time a new (better) result has been found, the new state is propagated through the ring to all worker processes. Delays in this state update mechanism may lead to unnecessary evaluations of suboptimal results and thus should be avoided. It is essential that the ring communication remains responsive under all circumstances.

Branch-and-bound algorithms can use a best-first search strategy, where the task pool is implemented as a priority queue, or a depth-first search strategy, with a stack implementation of the task pool [CP96]. Our skeleton can implement both strategies using appropriate instantiations of the parameter functions `ttDf`, `ttSplitf`, and `ttAf`. In our experiments, we observed a better performance of the depth first search in most cases. The general interface of the skeleton (shown in Fig. 2) must be used for branch-and-bound algorithms.

3.2 Skeleton Implementation

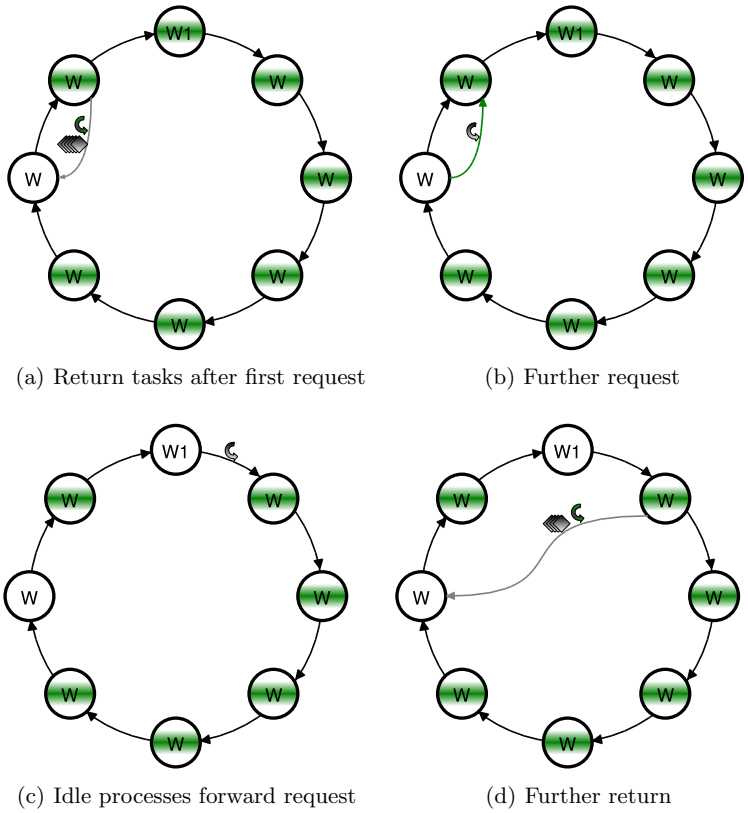
In the following, we describe the full implementation of the skeleton and explain more details of the skeleton parameters on the way.

Global Functionality. In the beginning, all initial tasks are evenly distributed to the worker processes, and the workers work on their local work pools. Newly generated tasks are put into this local pool. When the first worker becomes idle, the demand-driven task exchange starts, following a local round robin strategy [GGKK03]. Fig. 3 shows an exemplary request cycle to illustrate the functionality. Workers with an empty task pool are depicted in white, working processes appear with a coloured (dark) center. The first idle worker sends a work request to its neighbour through the ring. The request of type `Req t` with

```
data Req t =  ...  Other ChanName ([t], ChanName (Req t))  ...
```

contains a dynamic return channel which the receiving process can use to send part of its tasks to the demanding worker (Fig. 3 (a)). The split strategy defined by the parameter function `ttSplitf` determines which tasks will be sent to the requesting process. Together with the task list of type `[t]`, a dynamic request channel for further work requests (type `ChanName (Req t)`) is passed to the requesting worker. If the served worker runs out of work again, this request channel is used to send another work request directly to the process which answered the previous request (Fig. 3 (b)). The worker immediately forwards

Dieterle, Berthold, Loogen. A Skeleton for Distr. Work Pools in Eden. In:FLOPS'10, LNCS 6009.
© 2010 Springer Berlin/Heidelberg. The original publication is available at www.springerlink.com.



(a) Return tasks after first request (b) Further request

(c) Idle processes forward request (d) Further return

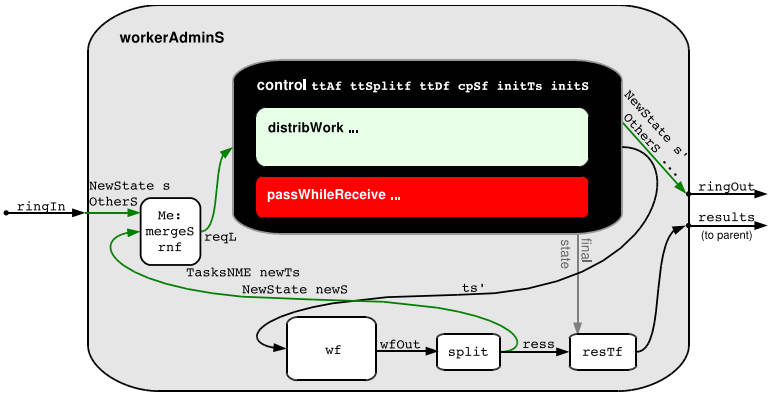
Fig. 3. Snapshots of Local Round Robin Strategy for Task Distribution

the request to its successor in the ring. The request is further passed through the ring (Fig. 3 (c)) until it reaches a worker with spare tasks which again will directly send further work (Fig. 3 (d)) and a new request channel via the return channel included in the request.

Note our notational distinction between request and return channels, which are technically the same. *Request channels* are the channels which transport work requests together with a return channel. The *return channel* is then used by a busy worker process to hand over some of its local tasks to the requesting process. In addition to the tasks, a request channel is supplied, which will be used by the requesting process to send the next work request directly to the process which answered its previous request. Thus, the ring structure is often bypassed via dynamic channels. Nevertheless, the ring is essential when systematically visiting all workers for termination detection.

Worker Functionality. The behaviour of the worker processes is determined by two functions: the task processing *worker function* `wf` and a `control` function

Dieterle, Berthold, Loogen. A Skeleton for Distr. Work Pools in Eden. In:FLOPS'10, LNCS 6009.
© 2010 Springer Berlin/Heidelberg. The original publication is available at www.springerlink.com.




```
data ReqS t s  = ME
               | OtherS (Tag, ChanName([t],Maybe(ChanName(ReqS t s))))
               | TasksNME [t]
               | NewState s   -- carry state inside and between workers

data Tag = Black | White (Int,Int,Int,Int) | None
         -- Tag White carries four counters of incoming/outgoing messages
         -- for two subsequent tours through the ring

workerAdminS :: (Trans t, Trans r, Trans s, NFData r') =>
 ...                   ->  -- passed parameters of general interface
 ChanName [ReqS t s]   ->  -- outgoing ring channel ringOutChan
 [ReqS t s]            ->  -- ring input ringIn
 Bool                  ->  -- first worker? isFirst
 [r]                       -- results to parent
workerAdminS wf resTf ttAf ttSplitf ttDf cpSf initTs initS
             ringOutChan ringIn isFirst
  = parfill ringOutChan ringOut results
  where -- central control: manage local work pool and requests
   (ts', ringOut)      = control ttAf ttSplitf ttDf cpSf
                                 initTs initS isFirst reqL
   reqL                = ME : mergeS rnf [ringIn,localReqs]
   -- task processing and final result transformation
   (ress, localReqs)   = split (wf ts')
   results             = resTf ress finalState
   NewState finalState = last ringOut
```

Fig. 4. Implementation Scheme of Ring-Connected Worker Processes

© 2010 Springer Berlin/Heidelberg. The original publication is available at www.springerlink.com.

which is the heart of the work pool management. Fig. 4 illustrates the internal flow of information and the code of the worker administration function `workerAdminS` which essentially maps a stream `ringIn` of incoming requests with type `ReqS t s` to a stream `ringOut` of outgoing requests and a `results` stream to the parent. The ring output is passed via a dynamic channel `ringOutChan` while the output for the parent is simply returned as the function result.

The request type `ReqS t s` (see Fig. 4) is the type of information passed through the ring, now extended with state information. It covers external and internal requests for task lists of type `t` as well as update information for the state (type `s`). External work requests are identified by the `OtherS` constructor. These include a `Tag` that is needed for distributed termination detection. Requests of the local task processing function are identified by the `TasksNME` constructor which additionally includes a list of newly generated tasks, or by the `ME` constructor which indicates a pure request for new work. State update information identified by the `NewState` constructor will be broadcasted using the ring topology.

The worker function `wf :: [(t,s)] -> [(Maybe (r,s),[t])]` processes a list of task/state pairs and outputs a list of pairs. In the first component a result/state pair may be returned. The second component is a list of newly created tasks. The `Maybe` type allows allows to indicate that no result or no better solution than the already known solution has been found. The output stream of the worker function is split into two streams: a stream of results which is transformed using the parameter function `resTf` and returned to the parent process, and another stream containing the new states and new tasks that have been produced. If existent, the new state `s` is forwarded as `NewState s`, the task list is embedded in a local work request `TaskNMe newTs`. The stream of local work request and new state information is merged with the ring input stream of external work requests and passed to the function `control` (see Fig. 4). Initially, the request stream to a `control` function contains a single `ME` request.

Local Worker Coordination. The central worker function `control` distinguishes between two different modes handled by the functions `distribWork` and `passWhileReceive`. The function `distribWork` is active as long as the local work pool is nonempty, i.e. work requests can be answered with tasks. It is the initial mode of the control function.

```
control .. requests initTasks initState isFirst
  = distribWork ..  requests initTasks initState Nothing ..
distribWork :: Trans t =>
   ...          ->               -- passed parameter functions
   [ReqS t s]   ->               -- requests
   [t] -> s ->                   -- work pool/state
   Maybe(ChanName (ReqS t s)) ->  -- return channel if available
   ... ->                        -- book keeping parameters
   ([t],s,[ReqS t s])            -- new work pool/state,
                                 --    outgoing requests/state infos
```

The function `passWhileReceive` is called when the local task pool is empty and a `Me` request occurs, i.e. the worker itself runs out of work. The first time

© 2010 Springer Berlin/Heidelberg. The original publication is available at www.springerlink.com.

this situation occurs, an `OtherS` work request is sent into the ring, tagged `Black` and containing a newly created return channel. Later requests will be sent on the previously received request channel. Incoming work requests are passed to the next ring process and incoming state information is handled as follows: New states are compared with the current state, and either accepted and passed on to the next ring process or discarded. When new tasks are received via the return channel, control is passed back to the `distribWork` function.

Termination Detection. We have implemented a combination of two standard algorithms: Mattern's "four counter method" [Mat87] and Dijkstra's token algorithm [ED83]. The four counter method counts the number of outgoing and incoming messages per process with a control message circulated twice through the whole ring. We use our work requests for that purpose. Termination is initiated when a work request completes the second ring tour with the same balanced number of sent and received messages as in the first tour. While counting incoming and outgoing messages helps to detect ongoing communications, the tag colour is used to check whether there are still busy workers that may produce additional work.

The main difficulties in the implementation of the distributed work pool skeleton have been to add additional evaluation demand, to ensure liveness of the whole system, and to appropriately merge input data received via many different channels. An example for additional demand is the `mergeS` variant used in the function `workerAdminS` (see Fig. 4). This variant uses a function `rnf` (reduce to normal form strategy) to force the evaluation of stream elements before they are written into the result stream. An additional optimisation of merging will be discussed in the following section. We will not go further into the details of the skeleton implementation. The complete code can be found in [Die07].

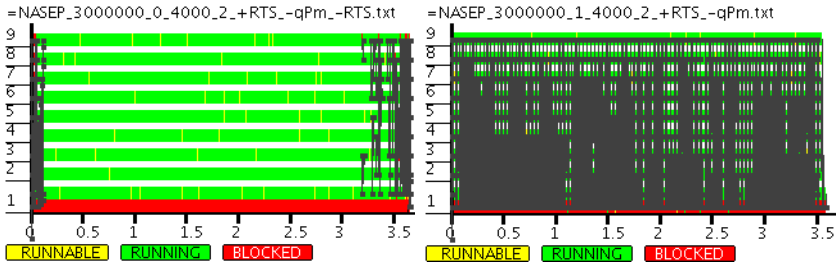
4 Experimental Results

In this section, we present experimental results for typical case studies. We visualise the runtime behaviour using activity profiles and show runtime measurements and speedup figures for the most general case of a branch-and-bound problem.

NAS EP Benchmark. We have compared the skeleton with a simple master-worker-skeleton [LOMP05] using an exemplary transformation problem, the NAS parallel benchmark EP (Embarrassingly Parallel) [BBB+94]. In this benchmark, two-dimensional statistics are accumulated from a large number of Gaussian pseudo-random numbers. Very little communication is required. Fig. 5 visualises the process activities over time for both computations with 3 million numbers on an inhomogeneous local network of 9 Linux workstations. Active phases of the processes are shown in cyan (middle gray), blocked phases in red (dark), and runnable phases in yellow (light). Communication is overlayed, i.e. messages are shown as black arrows from the sending to the receiving process.

In data-parallel transformation problems, the entire task pool is known in advance. With the distributed work pool skeleton, workers are assigned a fixed

Dieterle, Berthold, Loogen. A Skeleton for Distr. Work Pools in Eden. In:FLOPS'10, LNCS 6009.
© 2010 Springer Berlin/Heidelberg. The original publication is available at www.springerlink.com.



(a) Distributed Work Pool Skeleton (b) Simple Master-Worker-Skeleton

Fig. 5. Activity Profiles of NAS EP Benchmark, Input Size 3M, 9 PEs

task subset initially. The activity profile shows that workers are active most of the time. Communication takes place only in the beginning and at the end of the computations. The computation of the simple master-worker-skeleton is very communication-intensive, because the master continuously distributes tasks to the workers. The more sophisticated distributed work pool skeleton has almost no runtime overhead. Load is well balanced in both cases.

Graph Partitioning Problem. The graph partitioning problem is a typical branch-and-bound problem. A graph has to be partitioned into two sub-graphs with an (almost) equal number of nodes where the weight sum of the edges connecting the two subgraphs — the truncation cost — is minimal. The partitioning is incrementally built up by traversing the list of graph nodes and defining sub-problems where each node is assigned to one of the two possible sub-graphs. The actual truncation costs of partial solutions are used to compute a lower bound on the truncation costs of corresponding complete solutions.

The following runtime experiments were carried out on a Beowulf cluster at Heriot-Watt-University, Edinburgh which consists of 32 Intel P4-SMP-processors running at 3 GHz with 512 MB RAM and a Fast Ethernet interconnection. Fig. 6 shows the activity profile when evaluating the graph partitioning problem for a graph with 30 nodes on 31 PEs. On the left hand side, the whole trace is shown. On the right hand side, we see a zoom of the end phase of the same trace. Again, most communication takes place in the beginning to establish the topology and during the final phase, when idle workers send work requests to other workers. The request-reply cycles are clearly visible in the zoomed view. The whole system is well-balanced and all workers are equally loaded. Note that a comparison with the simple master-worker skeleton is not possible, because the latter cannot be used for the implementation of branch-and-bound algorithms.

Two parameters have major impact on the runtime of the parallel program: The cutoff depth (explained below) and the `merge` function. We have examined this impact with experiments focussing on a depth-first branch-and-bound implementation of the graph partitioning problem with graphs consisting of 32 nodes.

Dieterle, Berthold, Loogen. A Skeleton for Distr. Work Pools in Eden. In:FLOPS'10, LNCS 6009.
© 2010 Springer Berlin/Heidelberg. The original publication is available at www.springerlink.com.

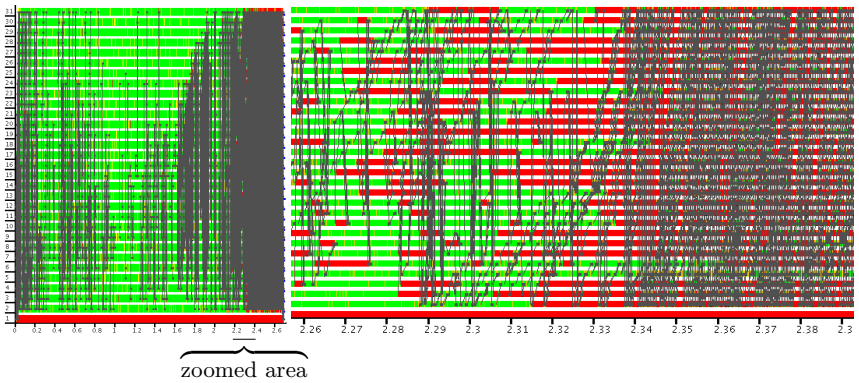


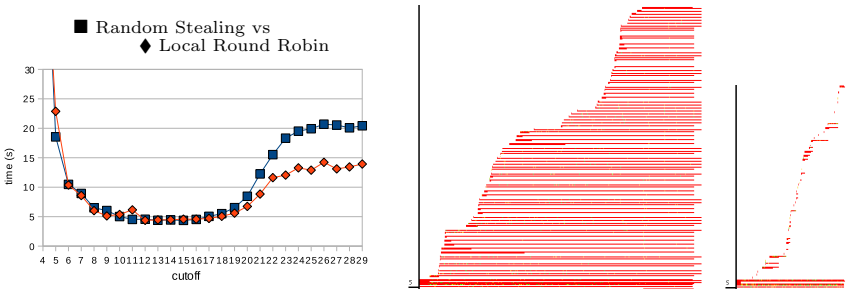
Fig. 6. Graph Partitioning (30 Nodes), Entire Run (Left) and Zoomed End Phase (Right), 31 PEs, Runtime: 2,68 sec

The cutoff Parameter. Tasks evaluating nodes near the root of the search tree usually have a higher complexity than tasks whose nodes are deeper in the tree, i.e. tasks are irregular in their potential to generate new subtasks. Load balancing strategies should take this fact into account, and preferably give away tasks which have a higher "potential". Usually, the tree depth in the decision tree is known, or can be estimated cheaply. Thus, a load balancing strategy may retain tasks when their distance from the root is bigger than a `cutoff`. These small tasks will be solved locally: Passing them to other workers would be more expensive than local evaluation. Although this behaviour is related to task distribution, it is more easily encoded in the worker function. The runtime of the work pool can benefit in two ways from a properly adjusted cutoff parameter. Tasks are only sent to other workers if they have the potential to produce enough work, depending on the remaining subtree depth, thereby reducing communication overhead. In addition, evaluation of tree levels beyond the cutoff depth is done by a simple recursive function, bypassing the control function. The traces in Fig. 6 have been obtained with the (experimentally determined) best cutoff depth.

We have tested the impact of the cutoff depth with two different program versions. One is based on the skeleton described in the previous section, i.e. using a ring of worker processes and implementing a local round robin strategy for task stealing. The second one implements an all-to-all communication topology among the workers and a random strategy for task stealing, i.e. the processes to be asked for tasks are randomly chosen. We made runtime comparisons using these skeleton versions on up to 31 PEs with the cutoff depth ranging from 5 (early cut) to 29 (late cut). The results presented in Fig. 7(a) show an optimal cutoff with 12, 13, 14, or 15 for both program versions. The two versions perform similar in the area of the optimal cutoff values. With higher cutoff values the local round robin version is faster than the random version.

Improving `merge`. At first, our example programs showed a steadily increasing number of active threads per process. The thread activities of a single pro-

Dieterle, Berthold, Loogen. A Skeleton for Distr. Work Pools in Eden. In:FLOPS'10, LNCS 6009.
© 2010 Springer Berlin/Heidelberg. The original publication is available at www.springerlink.com.



(a) Runtimes for varying cutoff par. (b) Threads view: old `merge` (c) new `merge`

Fig. 7. GPP 32 Test Runs on 31 PEs

cess (number 21) in a program run with cutoff value 26 and the original "old" `merge` are illustrated in Fig. 7(b). Each horizontal line represents the life time of a thread. The picture shows many long-living threads within a single process which are blocked most of the time. This is due to the fact that Eden's `merge` which is implemented using the Concurrent Haskell `nmergeIO: [[a]] -> IO [a]` forks one additional thread per input list for concurrently passing through this list; the untouched list elements are written into a single output list. A thread terminates as soon as it reaches the end of its list. Even the final thread will transmit its input list element-by-element into the output list. This approach is acceptable for stream merging. However, each time a finite list is merged with a stream, the number of threads increases by one. In our skeleton, the values sent on dynamic channels (e.g. tasks or work requests) are always merged with the request stream scanned by the worker's `control` function. Thus, the original `merge` implementation causes the number of running threads to increase with the number of requests and replies. We have modified the Concurrent Haskell `nmergeIO`, so that the merger threads can detect this situation and terminate earlier. This implementation dramatically reduces the life time and number of merge threads in the above scenario, as we can see in Fig. 7(c). The overall runtime is reduced because messages are no longer passed through several intermediate threads.

Speedup. If the search tree is immediately cut with cutoff value 0, the task pool contains only the initial node, and the whole branch-and-bound problem is evaluated sequentially. This eliminates most of the overhead of the work pool skeleton compared to a sequential implementation, so we used this cutoff 0 version of the graph partitioning problem to approximate the behaviour and runtime of the sequential algorithm. Fig. 8 shows the almost linear speedup of a series of program runs with the new `merge` version, the cutoff value 13 and the number of processors ranging from 1 up to 29 in comparison with the pseudo-sequential version. The runtime on 1 machine with cutoff 13 and the pseudo-sequential version (cutoff value 0) are very close, they differ less then 1%. Efficiency slightly drops when the number of processors is increased, but it stays above 88%.

Summary. The parallel runtime behaviour of the skeleton has been visualised for the NAS EP benchmark which implements a data-parallel transformation

Dieterle, Berthold, Loogen. A Skeleton for Distr. Work Pools in Eden. In:FLOPS'10, LNCS 6009.
© 2010 Springer Berlin/Heidelberg. The original publication is available at www.springerlink.com.

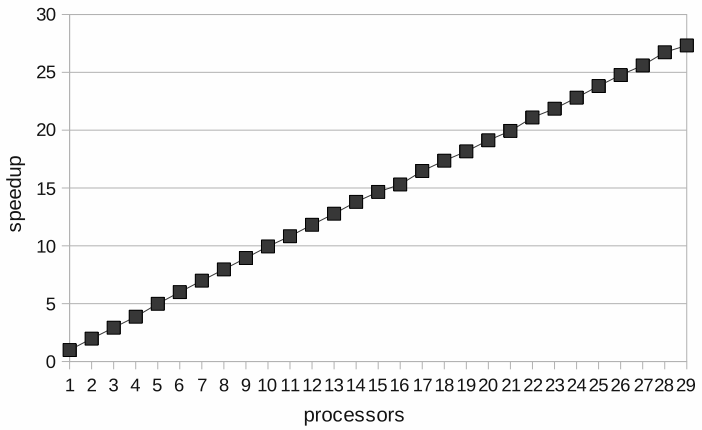


Fig. 8. Speedup for the GPP 32 Problem

problem and for the graph partitioning problem which is a typical branch-and-bound algorithm. Both profiles show that communication concentrates on the start-up and the final computation phase when the worker processes run out of work. Load is well-balanced in both cases. For the NAS EP benchmark, the activity profile has been compared with a profile of a simple master-worker skeleton which shows a higher communication overhead due to the continuous task distribution by the master process. For the graph partitioning problem, it has been shown that the cutoff parameter has a great impact on the runtime. Using a random instead of a local round robin strategy for task stealing makes only a difference for sub-optimal cutoff values, where the random strategy leads to higher runtimes. By improving the implementation of the merge function in the case that a stream is merged with a finite list a substantial reduction of the number and the lifetime of threads and, consequently, of the overall runtime could be achieved. Finally, this led to an almost linear speedup when using an optimal cutoff value and the new merge function.

5 Related Work

The master-worker paradigm has been extensively investigated and many implementations exist. We focus here on other pattern or skeleton approaches and especially on distributed work pool implementations. In the context of the grid computing environment Condor, the MW (Master-Worker) library has been developed [GL06]. MW is tailored for branch-and-bound applications. It implements the basic master-worker system with a central master managing the task pool and a set of worker processes. A special feature that is not supported by our skeleton is the addition and removal of workers during runtime which is especially important in a grid environment. This feature cannot easily be implemented in Eden, because dynamic channels cannot simply be re-directed.

Dieterle, Berthold, Loogen. A Skeleton for Distr. Work Pools in Eden. In:FLOPS'10, LNCS 6009.
© 2010 Springer Berlin/Heidelberg. The original publication is available at www.springerlink.com.

Most skeleton libraries like [Kuc,Ben,Dan] provide master-worker skeletons. The MPI-based skeleton library Muesli [Kuc], e.g. offers a farm, a search and a branch-and-bound skeleton. The farm implements a master-worker system with a dynamic task distribution. The search and branch-and-bound skeletons are especially tailored for the corresponding problem classes. Our distributed work pool skeleton is more general, because it supports all three problem classes. Moreover, it allows a hierarchical result collection. In [PK06], Kuchen and Poldner present a distributed branch-and-bound skeleton based on a distributed work pool. The workers are also arranged in a ring. Two task distribution policies are supported: a supply-driven scheme where workers send their second-best problem to their ring neighbour from time to time, and a demand-driven scheme where work is only distributed if an idle worker requests it.

Hippold and Rünger describe *task pool teams* [HR06], a programming environment for SMP clusters that is explicitly tailored towards irregular problems with strong inter-task dependences. The scheme comprises a set of task pools, each running on its own SMP node, and interacting via explicit message passing. Dynamic task creation by workers, task migration, and distributed task pools with a task stealing mechanism are possible. Locality can be exploited to hold global data on the SMP nodes, while communication between nodes is used for task migration, remote data access, and global synchronisation.

Dorta et al. present a master-worker skeleton implementation in C plus MPI which is tailored for branch-and-bound problems [DLR06]. A master process is used to coordinate the interaction between the worker processes and to keep the information about the currently best solution. A task pushing approach is implemented where the master process determines to which workers a worker should send its newly created tasks. No cutoff parameter is used improve the task granularity. Distributing the whole search tree leads to a high imbalance of task sizes. The profiles show that the workers spend most of the time waiting for new tasks. Moreover, a lack of scalability was observed.

In a previous paper [BDLP08], we have investigated declarative techniques for *hierarchically nesting* master-worker instances. The workers are divided into several groups managed by a hierarchy of sub-masters. The work pool is divided into several sub-pools within the sub-masters. Now we have followed a more radical approach and completely *distributed the work pool* within the worker processes.

6 Conclusions

A distributed work pool skeleton has been implemented in the parallel functional language Eden. Eden's specific features like lazy stream processing, dynamic reply channels, nondeterministic merge are very supportive for the efficient implementation of the complex coordination structure of the skeleton. The skeleton is very general, highly parameterised, and thus applicable to a range of problem classes. Experiments show a stable runtime behaviour, well-balanced work loads and worker activities. Communication overhead mainly occurs in the end phase of executions.

Dieterle, Berthold, Loogen. A Skeleton for Distr. Work Pools in Eden. In:FLOPS'10, LNCS 6009.
© 2010 Springer Berlin/Heidelberg. The original publication is available at www.springerlink.com.

Acknowledgements. We thank the anonymous referees for their helpful comments on a previous version of this paper.

References

[BBB+94] D. Bailey, E. Baszcz, J. Barton, D. Browning, R. Carter, I. Dagum, and et al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA, 1994.

[BDLP08] Jost Berthold, Mischa Dieterle, Rita Loogen, and Steffen Priebe. Hierarchical Master-Worker Skeletons. In *PADL'08 — Practical Aspects of Declarative Languages, LNCS 4902*, pages 248–264. Springer, 2008.

[Ben] Anne Benoit. ESkel — The Edinburgh Skeleton Library. Univ. of Edinburgh 2007, `http://homepages.inf.ed.ac.uk/abenoit1/eSkel/`.

[CP96] J. Clausen and M. Perregaard. On the best search strategy in parallel branch-and-bound - best-first-search vs. lazy depth-first-search. Technical Report 16, University of Copenhagen, 1996.

[Dan] Marco Danelutto. The parallel programming library Muskel. Universita di Pisa 2007, `http://www.di.unipi.it/~marcod/Muskel/Home.html`.

[Die07] Mischa Dieterle. Parallel functional implementation of master worker skeletons. Diploma Thesis, Philipps-Universität Marburg, October 2007. (in german).

[DLR06] I. Dorta, C. Léon, and C. Rodríguez. Performance Analysis of Branch-and-Bound Skeletons. In *14th Euromicro Conf. on Parallel, Distributed, and Network-Based Processing (PDP'06)*. IEEE, 2006.

[ED83] A.J.M. van Gasteren E.W. Dijkstra, W.H.J. Feijen. Derivation of a termination detection algorithm for distributed computations. *Inform. Process. Lett.*, 16(5):217–219, 1983.

[Fos95] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[GGKK03] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Pearson Education, 2003.

[GL06] W. Glankwamdee and J.T. Linderoth. MW: A Software Framework for Combinatorial Optimization on Computational Grids. In E. Talbi, editor, *Parallel Combinatorial Optimization*, pages 239–262. Wiley, 2006.

[HR06] J. Hippold and G. Rünger. Task Pool Teams: A Hybrid Programming Environment for Irregular Algorithms on SMP Clusters. *Concurrency and Computation: Practice and Experience*, 18:1575–1594, 2006.

[Kuc] Herbert Kuchen. The Münster Skeleton Library Muesli. Univ. Münster 2007, `http://www.wi.uni-muenster.de/PI/ forschung/Skeletons/index.php`.

[LOMP05] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[Mat87] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[PK06] Michael Poldner and Herbert Kuchen. Algorithmic skeletons for branch & bound. In *ICSOFT (1)*, pages 291–300. INSTICC Press, 2006.

[Qui03] Michael Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw Hill, 2003.