

# Hierarchical Master-Worker Skeletons

Jost Berthold, Mischa Dieterle, Rita Loogen, and Steffen Priebe

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik  
Hans Meerwein Straße, D-35032 Marburg, Germany  
{berthold,dieterle,loogen,priebe}@informatik.uni-marburg.de

**Abstract.** Master-worker systems are a well-known and often applicable scheme for the parallel evaluation of a pool of tasks, a *work pool*. The system consists of a master process managing a set of worker processes. After an initial phase with a fixed amount of tasks for each worker, further tasks are distributed in reply to results sent back by the workers. As this setup quickly leads to a bottleneck in the master process, the paper investigates techniques for hierarchically nesting the basic master-worker scheme. We present implementations of hierarchical master-worker skeletons, and how to automatically calculate parameters of the nested skeleton for good performance.

Nesting master-worker systems is nontrivial especially in cases where new tasks are dynamically created from previous results (typically breadth- or depth-first tree search algorithms). We discuss how to handle dynamically growing pools in a hierarchy and present a declarative implementation for nested master-worker systems with dynamic task creation.

The skeletons are experimentally evaluated with two typical test programs. We analyse their runtime behaviour and the effects of different hierarchies on runtimes via trace visualisations.

## 1 Introduction

Parallellising an algorithm implemented as a functional program starts by identifying a set of largely independent evaluations. These *tasks* have to be assigned to nodes of a parallel computer, to gain high speedups by simultaneous evaluation. If the tasks are *regular* and their *number* is statically known, mapping them to the parallel nodes is trivial. The everyday situation, however, faces us with *irregular* tasks of varying and unknown complexity. The *static* task distribution should be replaced by a *dynamic* one.

The *master-worker* scheme is a parallel skeleton for a task pool with dynamic task distribution. A master process distributes tasks to a set of subordinate worker processes, and collects the results. Many-to-one communication enables the master to evenly supply a new task to each worker every time it sends back a result. Worker idle-time in the period between sending a result and receiving a new task can

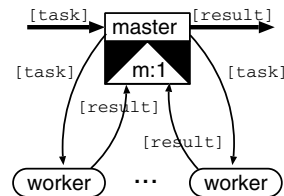


Fig. 1. Master-worker scheme

be avoided by pre-assigning a configurable amount (*prefetch*) of tasks to all workers. The prefetch parameter determines the behaviour of the skeleton, between a completely dynamic (prefetch 1) and a completely static distribution ( $\text{prefetch} \geq \frac{\text{no. of tasks}}{\text{no. of workers}}$ ).

So far, we have assumed a statically fixed task pool, which, in essence, results in a parallelised map function with dynamic assignment. Again, more realistic are *dynamic* settings where results might imply additional new tasks at runtime. This changes the scene completely: Tasks are not only irregular and of unknown number, but also carry an unknown 'task productivity'. This weakens the influence of the prefetch parameter.

A master-worker scheme essentially relies on a double functionality of the master process: it is responsible for collecting (possibly large) results, and it emits new tasks to idle workers. When a large number of workers is used, the single master process quickly becomes a bottleneck which paralyses the whole scheme. On the other hand, using more coarse-grained work requests, and consequently tasks, would restrict the dynamic adaption to the workload. As a remedy, we conceptually investigate techniques to nest the basic master-worker skeleton in a *master-worker hierarchy*. The master process at the top distributes tasks to several lower submasters, each of which manages a (smaller) worker set of its own, or possibly another level of submasters in a deeper hierarchy.

The hierarchical master-worker system as a whole is tree-shaped, with worker processes at the leaves and submasters as the inner nodes. The optimal hierarchy layout depends on the nature of the tasks, and on the number and performance of processing elements (PEs). The basic skeleton mechanism of tasks and requests remains the same at all tree levels, but at higher levels of the tree, skeleton parameters and distribution policies have to be adjusted to achieve good performance. In the case of a dynamic task pool, another question we investigate is whether submasters at one level should forward new tasks to upper levels, or keep them for their own worker set. A simple hierarchical scheme for master-worker systems has been presented in [7]. While we concentrate on the hierarchies, the focus of [7] has been a modified master process, which enables a transformation of the dynamically evolving task queue considering global information.

The paper is organised as follows: Section 2 presents non-hierarchical and hierarchical master-worker skeletons for a static task pool; the essential mechanism for nesting the basic skeleton, and how to automatically compute suitable skeleton parameters. In Section 3, we extend the skeleton for the case of dynamic task sets, and show the more complex nesting mechanisms needed for this skeleton variant. Each section includes experiments with an example application, discussing the behaviour for different hierarchy layouts and prefetch values. Section 4 discusses related work, Section 5 concludes.

## 2 Static Task Pools

In this section, we consider master-worker systems with a *static* task pool, i.e. no tasks are created during processing. The task pool is a list of tasks which can also

---

```

mw :: (Trans t, Trans r) => Int -> Int -> (t -> r) -> [t] -> [r]
mw n prefetch wf tasks = ress
  where (reqs, ress) = (unzip . merge) (spawn workers inputs)
        -- workers    :: [Process [t]] [(Int,r)]
        workers      = [process (zip [i,i..] . map wf) | i <- [0..n-1]]
        inputs       = distribute n tasks (initReqs ++ reqs)
        initReqs     = concat (replicate prefetch [0..n-1])

-- task distribution according to worker requests
distribute :: Int -> [t] -> [Int] -> [[t]]
distribute np tasks reqs = [taskList reqs tasks n | n<-[0..np-1]]
  where taskList (r:rs) (t:ts) pe | pe == r = t:(taskList rs ts pe)
        | otherwise = taskList rs ts pe
        taskList _ _ _ = []

```

---

Fig. 2. Eden master-worker skeleton with a static task pool

be provided as a stream, the total number of tasks does not have to be known in advance. System termination depends, however, on closing this task stream.

## 2.1 The Basic Master-Worker Skeleton

We perform our experiments in the parallel Haskell extension Eden [4] which allows to specify many different variants of the general master-worker schemes in an elegant and concise way. Figure 2 shows the Eden implementation of the basic master-worker skeleton. The task pool `tasks` is distributed to `n` worker processes, which, for each task, apply the worker function `wf` and return a pair consisting of the worker number and the result of the task evaluation to the master process, i.e. the process evaluating `mw`. The worker numbers are interpreted as requests for new tasks. The master uses a function `distribute` to send tasks to the workers according to the (`n*prefetch`) requests initially created and the ones received from the workers.<sup>1</sup> Care must be taken that `distribute` is *incremental*, i.e. it can deliver partial result lists without the need to evaluate requests not yet available. The skeleton uses the following Eden functions:

- `process :: (Trans a, Trans b) => (a -> b) -> Process a b`  
wraps a function into a *process abstraction* which shifts function evaluation to a remote processing element. The `Trans` context ensures the existence of internal communication functions.
- `spawn :: [Process a b] -> [a] -> [b]`  
starts *processes* on remote machines eagerly.
- `merge :: [[r]] -> [r]`  
nondeterministically merges a set of streams into a single one.

<sup>1</sup> Because the input for Eden processes is evaluated by concurrent threads in the generator process, separate threads for each worker evaluate the local function `tasklist`.

An additional merge phase would be necessary to restore the initial task order for the results. This can be accomplished by adding tags to the task list, and passing results through an additional function `mergeByTags` (not shown) which merges the result streams from all workers (each sorted by tags, thus less complex than a proper sorting algorithm). We will not go into further details.

In the following, we will investigate the properties and implementation issues of hierarchical master-worker skeletons. As proclaimed in the introduction, this should enable us to overcome the bottleneck in the master when too many workers must be served.

## 2.2 Nesting the Basic Master-Worker Skeleton

To simplify the nesting, the basic skeleton `mw` is modified in such a way that it has the same type as its worker function. We therefore assume a worker function `wf :: [t] -> [r]`, and replace the expression `(map wf)` in the worker process definition with `wf`. This leads to a slightly modified version of `mw`, denoted by `mw'` in the following. An elegant nesting scheme (taken from [7]) is defined in Figure 3. The parameters specify the branching degrees and prefetch values per level, starting with the root parameters. The length of the parameter lists determines the depth of the generated hierarchical system.

---

```

mwNested :: (Trans t, Trans r) =>
  [Int] -> [Int] -> -- branching degrees/prefetches per level
  ([t] -> [r]) -> -- worker function
  [t] -> [r] -- tasks, results
mwNested ns pfs wf = foldr fld wf (zip ns pfs)
  where fld :: (Trans t, Trans r) =>
    (Int,Int) -> ([t] -> [r]) -> ([t] -> [r])
    fld (n,pf) wf = mw' n pf wf

```

---

**Fig. 3.** Static nesting with equal level-wise branching

The nesting is achieved by folding the zipped branching degree and prefetches lists, using the proper worker function, of type `[t] -> [r]`, as the starting value. The folding function `fld` corresponds to the `mw'` skeleton applied to the branching degree and prefetch value parameters taken from the folded list and the worker function produced by folding up to this point.

The parameters in the nesting scheme above allow to freely define tree shape and prefetch values for all levels. As the `mw` skeleton assumes the same worker function for all workers in a group, it generates a regular hierarchy, one cannot define different branching or prefetch within the same level. It is possible to define a version of the static nestable work pool which is even more flexible (not considered here), yet more simple skeleton interfaces are desirable, to provide access to the hierarchical master-worker at different levels of abstraction. We can

define an interface that automatically creates a regular hierarchy with reasonable parameters for a given number of available processing elements.

```

mwNest :: (Trans t, Trans r) =>
    Int -> Int -> Int -> Int -> (t -> r) -> [t] -> [r]
mwNest depth level1 np basepf f tasks
    = let nesting = mkNesting np depth level1
        in mwNested nesting (mkPFs basepf nesting) (map f) tasks

```

In this version, the parameter lists are computed from a given base prefetch, nesting depth and top-level branching degree by auxiliary functions. These fewer parameters provide simple control of the tree size and shape, and prefetch adjusted to the task granularity.

Auxiliary function `mkNesting` computes a regular nesting scheme from the top-level branching degree `level1` and the nesting `depth`, which appropriately maps to `np`, the number of processing elements (PEs) to use. It calculates the branching list for a hierarchy, where all intermediate levels are binary. The number of workers per group depends on the number of remaining PEs, rounded up to make sure that all PEs are used. Please note that this possibly places several worker processes on the same PE. Workers sharing the same PE will appear as slow workers in the system, but this should be compensated by the dynamic task distribution unless the prefetch is too high.

$$l_d = \left[ \begin{array}{c} \text{total \# subm.s} \\ np - \overbrace{l_1 \cdot (2^{d-1} - 1)} \\ \underbrace{l_1 \cdot 2^{d-2}} \\ \text{\# lowest subm.s} \end{array} \right] \Rightarrow \text{Branching list: } \underbrace{l_1 : 2 : 2 : \dots : l_d}_{d \text{ levels}}$$

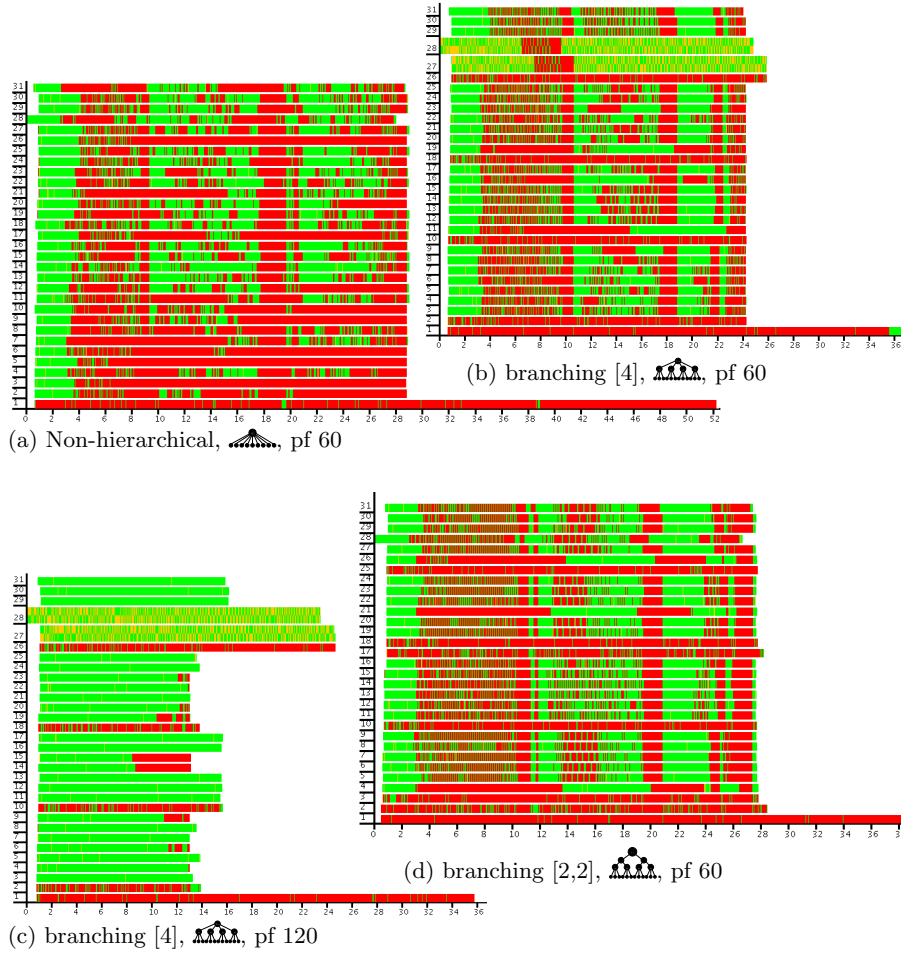
A central problem for the usage of the nested scheme is the choice of appropriate prefetch values per level, specified by the second parameter. A submaster with  $m$  workers requiring prefetch  $p$  should receive a prefetch of at least  $m \cdot p$  tasks to be able to supply  $p$  initial tasks to its child processes. Given a worker (leaf) prefetch of `pf` and a branching list  $[l_1, \dots, l_{d-1}, l_d]$ , this leads to the following minimum prefetch at the different levels:

$$\left[ \prod_{j=k}^{d-1} l_j * pf \mid k \in [1 \dots d-1] \right] = [(l_2 \cdot l_3 \cdot l_4 \cdot pf), (l_3 \cdot l_4 \cdot pf), (l_4 \cdot pf), pf]$$

A reserve of one task per child process is added to this minimum, to avoid the submaster running out of tasks, since it directly passes on the computed prefetch amount to its children. The list of prefetch values is computed by a `scanr1`.

### 2.3 Experimental Results

We have tested the presented nesting scheme with different branching and prefetch parameters, with an application that calculates a Mandelbrot set



**Fig. 4.** Mandelbrot traces, with different nesting and varying prefetch

visualisation of  $5000 \times 5000$  pixels. All experiments use a Beowulf cluster of the Heriot-Watt University Edinburgh, 32 Intel P4-SMP nodes at 3 GHz with 512 MB RAM and Fast Ethernet. The *timeline diagrams* in Figure 4 visualise the process activity over time for program runs with different nesting and prefetch. Blocked processes are red (dark), and active/runnable processes green/yellow (light).

*Flat vs. Hierarchical Master-worker System.* The hierarchical system shows better runtime behaviour than the flat, i.e. non-hierarchical version. Although fewer PEs are available for worker processes, the total runtimes decrease substantially. Figure 4(a) shows a trace of the non-hierarchical master-worker scheme. Many worker processes are blocked most of the time. In a hierarchical version with a single additional level comprising four submasters, shown in (b), workers finish

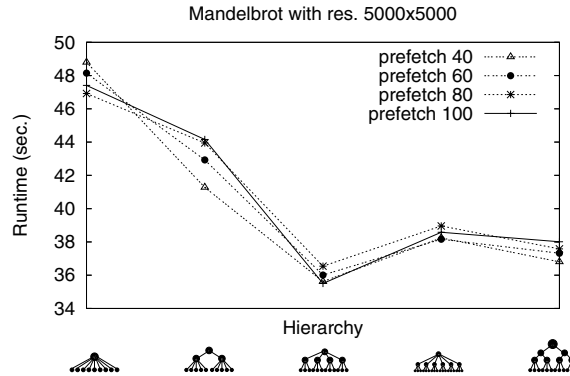


Fig. 5. Runtimes for various hierarchies and prefetch values

faster. Due to the regular structure of the hierarchy, some of the workers in the last branch share the same PE. Nevertheless, the system is well-balanced, but not completely busy. The dynamic task distribution of the master-worker inherently compensates load imbalance due to slower workers or irregular tasks.

*Load Balance and Prefetch Values.* In Figure 4(c), we have applied the same nesting as in (b), but we increased the prefetch value to 120. Small prefetch values lead to very good load balancing, especially PEs occupied by several (and therefore slow) workers do not slow down the whole system. On the other hand, low prefetch lets the workers run out of work sooner or later. Consequently, it is better to correlate prefetch values with the worker speed. Higher prefetch values (like 120) reduce the worker idle time, at the price of a worse load balance, due to the almost static task distribution.

*Depth vs. Breadth.* Figure 4(d) shows the behaviour of a nested master-worker scheme with *two* levels of submasters. It uses 2 submasters at the higher level, each serving two submasters. From our experiments, we cannot yet identify clear pros and cons of employing *deeper* hierarchies. Comparing runs with one and two additional submaster-levels, runtime and load balancing behaviour are almost the same, the advantage of the one-level hierarchy in Figure 4(b) remains rather vague. As shown in Figure 5, a broad flat hierarchy reveals the best total runtimes. However, the submasters will as well become bottlenecks when serving more and more child processes. Therefore, deeper hierarchies will be advantageous on bigger clusters with hundreds of machines.

*Garbage Collection and Post-Processing.* Another phenomenon can be observed in traces (a), (b) and (d): If the prefetch is small, relatively short inactivity at the tree root can make the whole system run out of work and lead to global inactivity. In this case, the reason are garbage collections in the master process, which make

all the submasters and workers run out of tasks. The effect is intensified by higher top-level branching, and compensated by higher prefetch (c).

Additional experiments have shown that the bottleneck in the master process is mainly caused by the size of the result data structures, collected and stored in the master’s local heap. This causes the long post-processing phases that can be observed in our traces. Moreover, since new requests are processed together with the result values, request processing is slowed down in the master processes. Using the same setup as in the previous experiments but replacing worker results with empty lists, the master has no problems to keep all workers busy, even with small prefetch values and no hierarchy.

### 2.4 Result Hierarchies

The hierarchy throttles the flow of results and thus helps shorten the post-processing phases. Therefore, the hierarchical master-worker skeletons show better total runtimes, as shown in Figure 5. The skeleton can further be optimised by decoupling result and request communication. Results may be much larger and hence more time consuming to be sent than simple requests of type `Int`. When requests are sent concurrently and directly to the master, they can be received and processed faster, speeding up work distribution. This, however, applies only if the master is not too busy collecting results.

In this section, we consider a skeleton which collects the results hierarchically to unload the master, but sends requests and tasks *directly* from the master to the workers. The inner processes of the process tree collect the results from their child processes, but also serve as additional workers. The result streams of inner workers are merged with the collected ones, and forwarded to their parent process. To speed up the work distribution, we additionally separate the task distributor functionality of the master from its collector role (also proposed in [6]), which is only a minor change to the previous result-hierarchical skeleton. The result-collecting master creates a distributor process and redirects the direct connections between master and workers to exchange tasks and results. The resulting process structure is depicted in Figure 6.

Figure 7 shows traces for the non-hierarchical skeleton with concurrent request handling, with and without a separate distributor process, and a variant which collects results in a hierarchy, with 4 submasters. As shown in trace (a), concurrent request handling alone does not improve the skeleton performance.

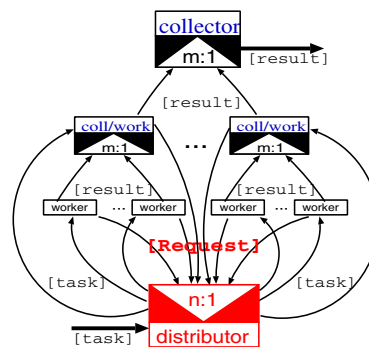
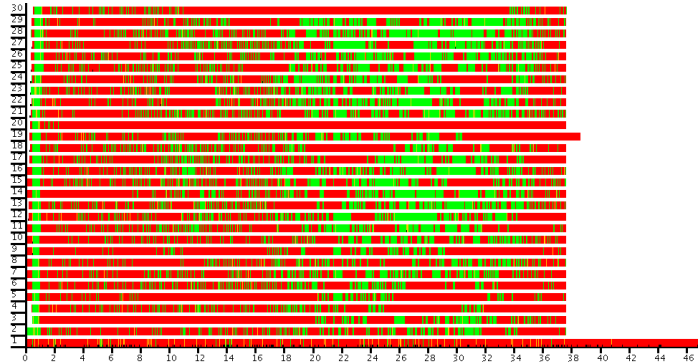

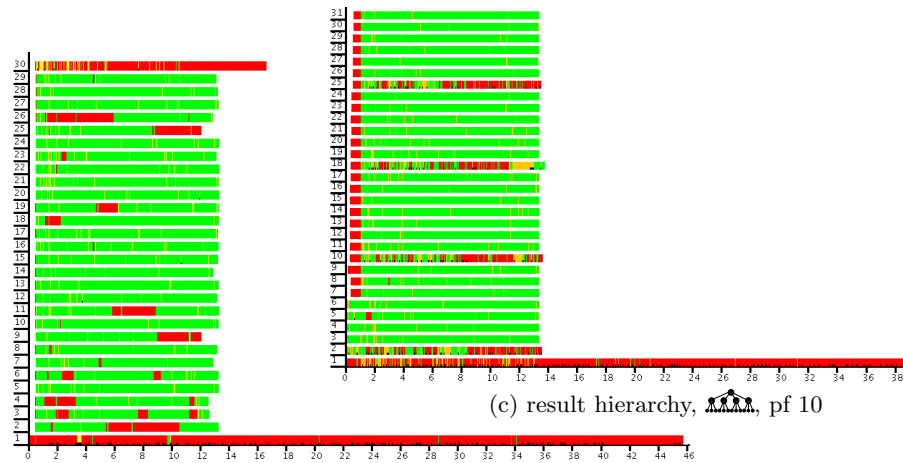



Fig. 6. Result-hierarchical scheme with separation of distributor and collector





(a) Concurrent request handling, non-hierarchical, , pf 20



(b) separate distributor, non-hierarchical , pf 20

(c) result hierarchy, , pf 10

**Fig. 7.** Mandelbrot traces, different result-hierarchical skeletons

Separating the distributor (trace (b), top bar shows distributor) already creates an almost steady workload for the workers, but exposes the same long post-processing. A result hierarchy (trace (c), without separate distributor) shortens the post-processing phase to some extent, while keeping the same positive effect on worker load.

### 3 Dynamic Creation of New Tasks

Except for some problems consisting of independent tasks which are trivial to parallelise, e.g. mandelbrot, ray tracing and other graphical problems, many problems deliver tasks containing inherent data dependencies. Thus, the task pool is not completely known initially, or it depends on other calculation results

to be fully defined. This is the case when the problem space is built hierarchically, as a tree structure or following other, more complex, patterns.

### 3.1 The Dynamic Master-Worker Skeleton

The elementary master-worker skeleton can easily be extended to enable the dynamic creation of additional tasks within the worker processes. In the version shown in Figure 8, the worker processes deliver a list of new tasks with each result, and the master simply adds the new tasks to its task queue. A straightforward extension would be to let the master filter or transform the task queue, considering global information (main point of investigation in [7]).

---

```

mwDyn :: (Trans t, Trans r) => Int -> Int -> (t -> (r,[t])) -> [t] -> [r]
mwDyn n prefetch wf initTasks = finalResults
  where -- identical to static task pool except for the type of workers
        (reqs, ress) = (unzip . merge) (spawn workers inputs)
        workers      = [process (zip [i,i..] . map wf) | i <- [0..n-1]]
        inputs       = distribute n tasks (initReqs ++ reqs)
        initReqs     = concat (replicate prefetch [0..n-1])
        -- additions for task queue management and termination detection
        tasks        = initTasks ++ newTasks
        initNumTasks = length initTasks
        (finalResults, newTasks) = tdetect ress initNumTasks

-- task queue control for termination detection
tdetect :: [(r,[t])] -> Int -> ([r], [t])
tdetect ((r,[]):ress) 1 = ([r], []) -- final result
tdetect ((r,ts):ress) numTs = (r:moreRes, ts ++ moreTs)
  where (moreRes, moreTs) = tdetect ress (numTs-1+length ts)

```

---

Fig. 8. Eden master-worker skeleton with a dynamic task pool

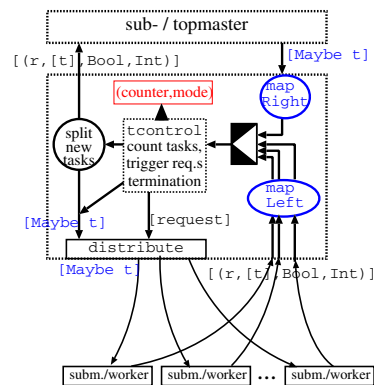
The static task pool version terminates as soon as all the tasks have been processed. With dynamic task creation, explicit termination detection becomes necessary, because the task list contains a reference to potential new tasks. In the skeleton shown in Figure 8, a function `tdetect` keeps track of the current number of tasks in process. It is essential that the result list is extracted via `tdetect` and that the evaluation of this function is driven by the result stream. As long as new tasks are generated, the function is recursively called with an updated task counter, initialised to the length of the skeleton input.<sup>2</sup> As soon as the result of the last task arrives, the system terminates by closing the tasks list and, via `distribute`, the input streams of the worker processes.

<sup>2</sup> The reader might notice that the initial task list now has to have fixed length. This skeleton cannot be used in a context where the input tasks arrive as a stream.

### 3.2 Nesting the Dynamic Task Pool Version

It would be possible to apply the simple nesting scheme from Section 2 to the dynamic master-worker skeleton `mwDyn`. However, newly created tasks would always remain in the lower submaster-worker level because the interface of `mwDyn` only passes results, but not tasks, to upper levels. For nesting, the dynamic master-worker scheme `mwDyn` has to be generalised to enable a more sophisticated task management within the submaster nodes.

Each submaster receives a task stream from its master and a result stream including new tasks from its workers. It has to produce task streams for its workers and a result stream including new tasks for its master (see Figure 9). Sending back all dynamically generated tasks is a waste of bandwidth, when they might be needed in the local subtree. A portion of the generated new tasks can be kept locally, but surplus tasks must be passed up to the next level. Furthermore, sending a result should *not* automatically be interpreted as a request for a new task, since tasks kept locally can compensate for solved tasks. Finally, global information about tasks in process is needed at the top-level, to decide when to terminate the system. The Eden code for the submaster in Figure 10 shows the necessary changes:



**Fig. 9.** Submaster functionality in the dynamic master-worker hierarchy

- The input stream for submasters and workers has type `Maybe t`, where the value `Nothing` serves as a termination signal, propagated downwards from the top level.
- The output stream of submasters (and workers) now includes information about the number of tasks kept locally, and a `Bool` flag, indicating the request for a new task, leading to type `[(r, [t], Bool, Int)]`.
- The incoming list `initTasks` for submasters is a stream, which has to be *merged* with the stream of worker answers, and processed by a central control function `tcontrol`. The origin of each input to `tcontrol` is indicated by tags `Left` (worker answers) and `Right` (parent input), using the Haskell sum type `Either (Int, (r, [t], Bool, Int)) (Maybe t)`
- All synchronisation is concentrated in the task control function `tcontrol`. It both controls the local task queue, passes new requests to `distribute`, and propagates results (and a portion of generated tasks) to the upper levels.

The heart of the dynamic master-worker hierarchy is the function `tcontrol`, shown in Figure 11. It maintains two counters: one for the amount of tasks

---

```

mwDynSub :: (Trans t, Trans r) =>
  Int -> Int -> ([Maybe t] -> [(r,[t],Bool,Int)])
  -> [Maybe t] -> [(r,[t],Bool,Int)]
mwDynSub n pf wf initTasks = finalResults where
  fromWorkers = spawn workers inputs
  -- worker    :: [Process [Maybe t] [(Int,(r,[t],Bool,Int))]]
  workers     = [process (zip [i,i..] . wf) | i <- [0..n-1]]
  inputs      = distribute n tasks (initReqs ++ reqs)
  initReqs    = concat (replicate pf [0..n-1])
  -- task queue management
  ctrlInput  = merge (map Right initTasks : map (map Left) fromWorkers)
  (finalResults, tasks, reqs) = tcontrol (n*pf+n) (False,0,0) ctrlInput

```

---

Fig. 10. Eden submaster for nested dynamic master-worker skeleton

that have been generated and passed up to this submaster, to decide whether a request must be sent up, and the overall task count in the subtree below.

Tasks sent by the parent are simply enqueued in the local task queue. Tasks generated by workers are `split` into a part that is kept local, and a part that is passed upwards. The nested task pools can be seen as a system of interdependent *buffers*, and both buffer-underruns and buffer-overruns will spoil the skeleton performance. This is relatively easy for a static task list: the exchange of tasks and results between different buffers is limited, and the `prefetch` parameter defines the maximum buffer size. In the extension for dynamically growing task

---

```

tcontrol _ (_,_,0) ((Right Nothing):_) -- from above, final termination
  = ([,repeat Nothing,[])
tcontrol pf (even,local,numTs) ((Right (Just t)):ress) -- task from above
  = let (moreRes, moreTs, reqs) = tcontrol pf (even, local ,numTs+1) res
      in (moreRes, (Just t):moreTs, reqs)
-- from i below, (result, new tasks, flag, no. of retained tasks)
tcontrol pf (even,local,numTs) ((Left (i,(r,ts,wantNew,tBelow))):ress)
  = let (localTs,fatherTs,evenAct) = split numTs pf ts even
      newLocal = length localTs + local
        - if wantNew && not newTasksForMe then 1 else 0
      newNumTs = numTs-1 + length localTs + tBelow
      (moreRes, moreTs, reqs)
        = tcontrol pf (evenAct, newLocal, newNumTs) res
      newreqs = if wantNew then i:reqs else reqs
      newTasksForMe = local + length localTs == 0 && wantNew
  in ((r, fatherTs, newTasksForMe, heldBelow + lenlocalTs):moreRes,
      (map Just localTs) ++ moreTs, newreqs)

```

---

Fig. 11. Control function for submaster of Figure 10

pools, more sophisticated policies are needed instead of mechanically forwarding new tasks and requests.

To achieve a roughly even level of tasks in each submaster, the task pool size is limited by two thresholds (sometimes called low and high watermark [3]). When too few tasks are locally generated, additional new tasks must be requested from the upper level, while all surplus tasks must be forwarded to upper levels. In our version, `tcontrol` emits requests when all self-generated tasks have been assigned, thereby trying to maintain its initial local task buffer size, given by the `prefetch` parameter. The `split` function decides how many tasks to hold in the subtree below a submaster. If a sufficient amount of self-generated tasks fills the subtree below the node (overall task count `numTs`), all generated tasks are forwarded to the upper level. The `split` function we use (not shown) splits generated tasks one half each, until the total task count exceeds the double `prefetch` for the whole subtree below. Different heuristics can be configured by exchanging the `split` function, and minor changes in `tcontrol`.

The top-level master in the nesting scheme for a dynamic task pool works similar to the submasters we have described, but of course cannot forward tasks to the outside. A separate top-level master has to be defined.

```
topMaster :: (Trans t, Trans r) =>
  Int -> Int -> ([Maybe t] -> [(r,[t],Bool,Int)]) -> [t] -> [r]
```

Besides termination detection, the former `tdetect` function now takes the role of `tcontrol` in the submaster processes, also incorporating the more sophisticated request handling we have introduced in the `tcontrol` function. Further changes are the adaption of the worker function to the `Maybe` type interface and the termination signal `Nothing` for all submasters upon termination.

### 3.3 Experimental Results

The skeletons that support dynamic task creation have been tested with a special test application: a program which computes all satisfying variable assignments for a particular logical formula (i.e. it is a specialised SAT problem solver). Tasks are incomplete variable assignments, and the workers successively assign a value to a new variable and partially evaluate the result. An assignment that already yields false is immediately discarded, true results are counted, yet unspecified results are new tasks returned to the level above.

The test program runs the satisfiability check for a formula which disjoins *all* conjunctions of  $n$  logical variables where  $k$  variables are negated (yielding  $\binom{n}{k}$  disjoint monoms). In the underlying *search tree* of the SAT problem, each node has at most two child nodes, but for particular formulas, many subproblem nodes in the search tree can immediately be discarded. Using a formula of 200 variables and 1 negation tests the skeleton behaviour for such a broad sparse tree. Especially with sparse search trees, it is challenging for the load balancing strategy to evenly fill the process hierarchy with tasks, avoiding idle times. Many tasks can be discarded early, but the test for 200 variables is complex. In contrast, a test with negated 8 variables out of 16 shows the performance for a dense tree

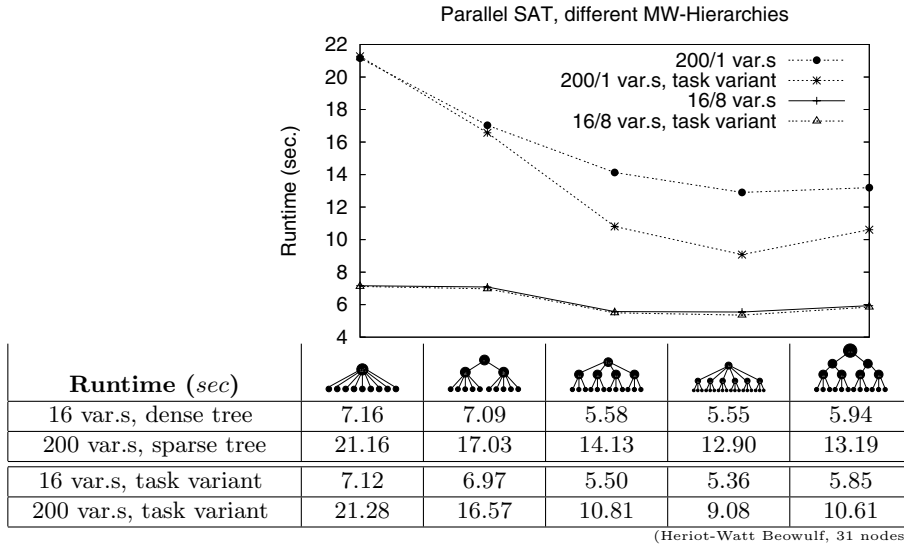


Fig. 12. Experiments using skeletons for dynamic task pool

with very small tasks. Runtimes have been compared for the basic skeleton, for hierarchies with one level of two, four and six submasters, and for a binary hierarchy with two levels.

*Flat vs. Hierarchical Skeleton:* In general, variants of hierarchical master-worker schemes perform better than the non-hierarchical skeleton in our test runs. However, when testing a formula with only 16 variables, tasks are numerous, but very simple. For this variant, hierarchical skeletons yield smaller runtime gains.

*Depth vs. Breadth:* The runtime comparison in Figure 12 shows that, in our setup of 31 machines, broader hierarchies with only one level perform better than the binary two-level hierarchy. The variant with 6 submasters yields the best results, whether sparse or dense decision trees. Measurements with a simplified test procedure, where tasks are checked very quickly using additional knowledge about the tested formula, confirm this result: The performance of the skeleton with two-level nesting is slightly worse than for the one-level nestings. Of course, this result again has to be qualified for bigger clusters.

*Prefetch and Forwarding Policy:* Prefetch values have little influence on performance (or trace appearance) for this test program, since there are relatively few tasks in the beginning anyway and many of the generated tasks are held close to the processing units. Higher prefetch values only lead to “bundled” working and idle phases instead of a more steady workload. Using higher prefetches, we also observed periods of global inactivity, again caused by garbage collections of the top-level master.

The partition policy for tasks returned by workers is a crucial property for an even global load balance. Our minimum threshold, the prefetch parameter, is self-suggesting: requests are emitted when locally generated tasks cannot keep the buffer filled. For the maximum threshold, our experiments have confirmed that increasing the high watermark for the split policy hardly produces performance gains. While the very existence of a maximum threshold has principal impact on the load balance (especially in our setup where only few new tasks are created), it is not necessary to add another parameter to the skeleton.

Figure 13 shows a trace for a program run using the best skeleton in our experiment, with six submasters above the leaves, on a sparse decision tree. The workers expose a slow startup phase, since the (relatively few) tasks must first be propagated in all branches. Tasks are well distributed among the different submaster branches, leading to an even workload among the worker processes. Even though some PEs are reserved as submasters, the remaining workers outperform the non-hierarchical skeleton close to factor 2.

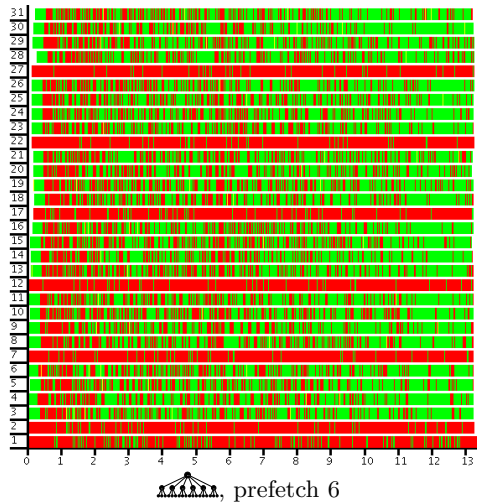


Fig. 13. Trace for SAT solver (200/1 var.)

## 4 Related Work

The commonly used master-worker scheme with a single master managing a set of workers is a well-known scheme which has been used in many different languages [1]. Modifications of this scheme are however more rare, and we are not aware of general hierarchical master-worker schemes like ours.

Driven by the insight that the propagation of messages is expensive in a master-worker scheme, Poldner and Kuchen [6] present a variant where the master is divided into a task distributor (*dispatcher*) and a result *collector*. As described in 2.4, we extended this variant to a skeleton with a hierarchy of collectors and only one distributor. In order to save communication, the dispatcher of Poldner and Kuchen applies a *static* task distribution, and they argue that for a large number of tasks, a roughly even load balance can be expected. However, this contradicts one basic idea of dynamic master-worker skeletons: the intention to balance not only task irregularity, but also potential differences in worker performance.

In [5], Poldner and Kuchen investigate skeletons for branch & bound problems. A centralized master-worker skeleton is compared to a set of distributed workers

connected by a bidirectional ring, without a central master. Distributed workers can establish load balance using a supply-driven or a demand-driven mechanism. In addition to the load balancing problem, the paper addresses branch & bound-specific requirements like fast propagation of updated bounds, and distributed termination detection. In the experiments with two branch & bound algorithms, the distributed skeleton with demand-driven load balancing shows best performance, due to the reduced communication need.

Hippold and Rünger describe *task pool teams* [2], 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.

Various load balancing strategies for divide-and-conquer algorithms are discussed by Nieuwpoort et al., in [8]. The authors experiment with different techniques to exchange tasks between autonomous worker processes, in the context of WAN-connected clusters (hierarchical wide-area systems). Aside from special optimisations to handle different network properties, a basic distinction is made between task *pushing* and *stealing* approaches. Demand-driven work stealing strategies are generally considered advantageous, but must take into account the high latency connections in question. The work pushing strategy speculatively (and blindly) forwards tasks to *random* peers when the amount of local tasks exceeds a prefetch threshold. Contrary to the randomised, or purely demand-driven, task distribution in this work, our skeletons are always based on task-request cycles, and concentrate surplus tasks at higher levels.

## 5 Conclusions

We have given a series of functional implementations of the parallel master-worker scheme. The declarative approach enables a clear conceptual view of the skeleton nesting we have developed.

Starting with a very compact version of the standard scheme, we have given implementations for skeleton nesting, to shift the administrative load to a whole hierarchy of (sub-)masters. The hierarchies have been elegantly expressed as foldings over the modified basic scheme. In the case of a dynamically growing task pool, a termination detection mechanism is needed. Nesting this skeleton is far more complex and needs special code for submasters, especially an appropriate task forwarding policy in the submaster processes.

As our tests show, master-worker hierarchies generally speed up runtime and keep workers busier, avoiding the bottleneck of a flat skeleton. Hierarchy layout and suitable prefetch values, however, have to be chosen carefully, depending on the target architecture and problem characteristics. Our experiments show the



importance of suitable task distribution and task forwarding policies, which we have described and discussed in detail.

We have presented implementations and experiments with a range of hierarchical master-worker variants, and we will continue investigations on some open topics. As ongoing work, we will develop distributed work pools, like the one proposed by Poldner and Kuchen in [5], and compare them to our master-worker hierarchies.

*Acknowledgements.* We greatly appreciate the opportunity to conduct runtime experiments on the Beowulf cluster of the Heriot-Watt University in Edinburgh.

## References

1. Danelutto, M., Pasqualetti, F., Pelagatti, S.: Skeletons for Data Parallelism in P<sup>3</sup>L. In: Lengauer, C., Griehl, M., Gortatch, S. (eds.) Euro-Par 1997. LNCS, vol. 1300, pp. 619–628. Springer, Heidelberg (1997)
2. Hippold, J., Rüniger, G.: Task Pool Teams: A Hybrid Programming Environment for Irregular Algorithms on SMP Clusters. *Concurrency and Computation: Practice and Experience* 18, 1575–1594 (2006)
3. Loidl, H.-W.: Load Balancing in a Parallel Graph Reducer. In: Hammond, K., Curtis, S. (eds.) SFP 2001 — Scottish Functional Programming Workshop, Bristol, UK. *Trends in Functional Programming*, vol. 3, pp. 63–74 (2001) (Intellect)
4. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel Functional Programming in Eden. *Journal of Functional Programming* 15(3), 431–475 (2005)
5. Poldner, M., Kuchen, H.: Algorithmic skeletons for branch & bound. In: Filipe, J., Shishkov, B., Helfert, M. (eds.) ICSOFT (1), pp. 291–300. INSTICC Press (2006)
6. Poldner, M., Kuchen, H.: Scalable farms. In: Joubert, G.R., Nagel, W.E., Peters, F.J., Plata, O.G., Tirado, P., Zapata, E.L. (eds.) ParCo 2005. *Parallel Computing: Current & Future Issues of High-End Computing*, Jülich, Germany. NIC Series, vol. 33, pp. 795–802 (2006)
7. Priebe, S.: Dynamic Task Generation and Transformation within a Nestable Workpool Skeleton. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 615–624. Springer, Heidelberg (2006)
8. van Nieuwpoort, R.V., Kielmann, T., Bal, H.E.: Efficient load balancing for wide-area divide-and-conquer applications. In: PPOPP 2001. *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pp. 34–43. ACM Press, New York (2001)