Dynamic Task Generation and Transformation within a Nestable Workpool Skeleton

Steffen Priebe*

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans-Meerwein-Straße, D-35032 Marburg, Germany
priebe@mathematik.uni-marburg.de

Abstract. Within a classical workpool skeleton a master process employs a set of worker processes to solve tasks contained in a task pool. In contrast to the usual statically fixed task set some applications generate tasks dynamically. Additionally often the need for dynamic task pool transformation arises, for example to combine newly generated partial tasks to form full tasks. We present an *extended workpool skeleton* for the parallel Haskell dialect *Eden* which provides both features and employs careful stream-processing and a termination detection mechanism. We also show how to nest the skeleton to alleviate the bottleneck a single master presents. Furthermore we demonstrate its efficiency by its fruitful use for the parallelisation of a DNA sequence alignment algorithm.

1 Introduction

Uneven task sizes arise naturally from many problems and are often an obstacle for parallelisation. Within parallel dialects of *Haskell* [1] the classical static distribution schemes (like parallel *map*) can hardly establish load-balance given unevenly sized tasks; therefore dynamic task distribution schemes are used. The well-known *workpool* scheme (also known as farm, master-worker, or client-server) [2] is mostly used to compensate for such irregularly sized tasks: A master administrates a statically fixed task pool out of which tasks are gradually assigned to currently idle workers, leading to a balanced workload. Such a scheme is often expressed as a high-level code template, known as a *skeleton* [3, 4].

Some applications however expose their full task set only successively as the computation proceeds and need therefore a more general workpool skeleton whose worker processes are allowed to generate new tasks dynamically. Then a task will not only produce a result, but possibly also a set of new tasks for the global task pool. This introduces the problem of termination detection, which was not a problem before since a statically fixed task number makes it easy to determine termination. Now special care has to be taken to account for a dynamically growing and shrinking task pool. Emptiness of the the task pool does no longer mean that there is no more work to do, since new work may still be created by active workers.

* Supported by *Evangelisches Studienwerk Villigst e.V.*

To make things even more complicated, dynamically created tasks may be incomplete (e. g. due to limited local data) and need to be combined with other incomplete tasks before submission to a worker. Therefore means have to be provided for traversing and transforming the task pool on the fly. But since the task pool is often modelled as a lazy list and woven into a network of interdependent streams, one has to be extra careful during a transformation. When combining partial tasks, deadlocks can easily occur by searching for not yet existent partial partner tasks; additionally, all usual techniques (like delayed pattern matching and incremental functions) when dealing with lazy lists have to be considered.

All this means extra work for the master process, which worsens the bottleneck it already presents. One way to alleviate this is exchanging the single master process by a tree of master and submaster processes distributing the administration load. This corresponds to a nested workpool, in which a workpool is given other workpools as worker processes.

Contributions.

- We present a new workpool skeleton for the parallel Haskell dialect *Eden* [5] which firstly enables worker processes to dynamically generate additional tasks for the task pool (together with the needed termination detection) and secondly permits dynamic transformation of the task pool (Sect. 2.2). A function aiding safe task pool transformation is also described.
- Via folding the workpool skeleton is then nested to reduce the administrative load of the master process (Sect. 2.3).
- We show the usefulness of the new workpool by applying it to the parallel alignment of DNA sequences (Sect. 3).

2 The Workpool Skeleton for Eden

2.1 A Short Glance at Eden

Eden extends Haskell with means for relocating the evaluation of a function application to other network nodes, enabling the evaluation of multiple expressions in parallel. A function embedded in a *process abstraction* by applying

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

can be run in parallel to the continuing evaluation of its parent expression on another processor by applying its arguments to a special application operator

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

New processes are placed round-robin on available nodes. There is no shared memory, all data exchanges happen via communication based on PVM [6] message passing within Eden's implementation. The `Trans` context ensures that only those values can be communicated for which corresponding low-level communication routines exist. Not only finite values but also infinite lists (known as *streams*) can be transmitted. These are sent piecewise with each stream element being demanded strictly. For merging a set of streams into a single stream a nondeterministic `merge :: [[a]]->[a]` function is predefined. Stream communication plays a vital part in the following workpool skeleton.

2.2 The Workpool Skeleton

Now we present a workpool skeleton which provides dynamic task generation and task pool transformation. The basic scheme works as follows: A master process keeps a pool of tasks which are distributed to a set of worker processes on request. When a worker receives a task, it solves it and sends back the result together with a request for new work. This way each worker is busy most of the time and load balance is kept as tasks are assigned depending on the current work distribution.

We extend the basic scheme by two new features, dynamic task generation and task pool transformation: *Firstly*, when a worker processes a task new tasks may arise. These will be sent back to the master and appended to the global task pool, preserving task order. *Secondly*, sometimes it is helpful to be able to process and transform the task pool. A given transformation function `tt` will be applied to the task pool to combine incomplete tasks and replace them by complete ones, possibly changing task order. The resulting basic interaction scheme is shown in Fig. 1. All connections shown are stream connections; the thick pointers touching the worker processes Ⓦ are interprocess connections while all others reside within the master process. The full code for the extended workpool is shown in Fig. 2. Parameters are: The number of processors available, the number of advance requests for each worker, the worker function, the transformation function for the task pool, and finally a set of initial tasks. At first, the workpool demands the first cons of the list of worker processes via `touch` to trigger their creation using the predefined parallel zip function `eagerInstList`, then the `results` are given back. The main body is divided into two parts:

The *stream part* defines the parallel stream network according to Fig. 1. A set of `workerProcs` is created, which apply the worker function `f` to their input and attach their id number to the result as a request for new work. Their input `toWorkers` is a list of streams, each of which contains `tasks` for the corresponding worker according to its `requests`. The `initialRequests` are built based on an interleaved sequence (each of size `prefetch`) of worker numbers and provides an initial supply of tasks for each worker. The worker's outputs are merged to

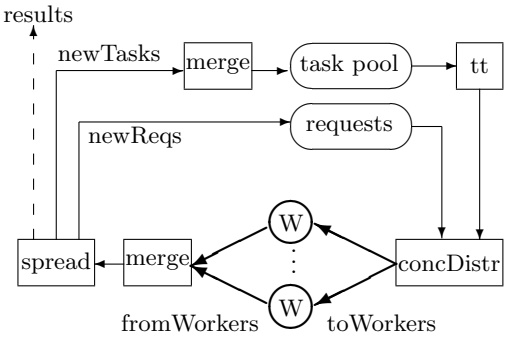


Fig. 1. Stream interconnections of workpool (seen from master process)

```
wpool :: (Trans t, Trans r) =>
         Int -> Int -> ([t] -> [(r, [t])]) ->
         (([t],[t],[t],Int) -> ([t],[t],[t],Int)) ->
         [t] -> [r]
wpool np prefetch f tt initialTasks =
  (touch fromWorkers) `seq` results
  where touch []    = ()       -- Demand first constructor to
        touch (_:_) = ()       -- initiate worker creation

        -- 1) Stream ------------------------------------------
        fromWorkers = eagerInstList workerProcs toWorkers
        workerProcs = [process (zip [n,n..] . f) | n<-[1..np]]

        toWorkers   = concDistr requests [1..np] tasks
        requests    = initialReqs ++ newReqs
        initialReqs = concat (replicate prefetch [1..np])

        taskpool                 = initialTasks ++ (merge newTasks)
        (_, _, tasks, _)         = tt (taskpool, [], [], 0)
        workerstream             = merge fromWorkers
        (newReqs, (x, newTasks)) = spread workerstream

        -- 2) State -------------------------------------------
        ([], _, results, _) = terminate
                              ([], length initialTasks, [], 0)
                              workerstream

        terminate (is,t,rs,r) ( (_,(res,ntasks)) :ws)
          | t' >  r' = terminate (is', t', res:rs, r') ws
          | t' == r' = (is', t', reverse (res:rs), r')
          | t' <  r' = error ("Will never happen.")
          where ([], is', _, n) = tt (is++ntasks, [], [], 0)
                t'              = t + n
                r'              = r + 1
        terminate _           []
          = error "Workerstream empty!"

concDistr :: Eq a => [a] -> [a] -> [b] -> [[b]]
concDistr unsortedKeys allKeys vals =
  where vals'  = zip unsortedKeys vals
        result = [ [v | (uk,v) <- vals', uk == k] | k <- allKeys]

-- TH splice creates: spread :: [(a,(b,[c]))] -> ([a], ([b], [[c]]))
$(do let empty      = ListE []
     let structure  = TupE [empty, TupE [empty, empty]]
     let spread_fct = mkSpread structure
     return spread_fct)
```

Fig. 2. Workpool with dynamic task generation and task pool transformation

```
-- Call in Fig.2    spread []               = ([], ([], []))
-- creates two      spread ((a,(b,c)):rest) = (a:as,(b:bs,c:cs))
-- clauses:           where (as,(bs,cs)) = spread rest
mkSpread :: Exp -> [Dec]
mkSpread struct = [FunD "spread" clauses] where
  clauses = [Clause pat1 body1 [], Clause pat2 body2 [ValD pat b []]]
  pat1    = [ListP []]
  body1   = NormalB (buildE struct (repeat (ListE [])))
  pat2    = [ConP "GHC.Base::" [pat2', VarP "rest"]]
  pat2'   = buildP struct [VarP [c] | c <- ['a','b'..]]
  body2   = NormalB (buildE struct lists)
  lists   = [AppE (AppE (ConE "GHC.Base::") (VarE [c]))
             (VarE [c,'s']) | c <- ['a','b'..]]
  pat     = buildP struct [VarP [c,'s'] | c <- ['a','b'..]]
  b       = NormalB (AppE (VarE "spread") (VarE "rest"))
buildE :: Exp -> [Exp] -> Exp;   buildP :: Exp -> [Pat] -> Pat
buildE (TupE vs) ls = TupE (fst (traverse vs ls)) where
  traverse ((ListE []):rest) (l:ls) = (l : r, ls')
    where (r, ls')   = traverse rest ls
  traverse ((TupE ws):rest) ls      = ((TupE rec):r,ls'')
    where (rec, ls') = traverse ws ls
          (r, ls'')  = traverse rest ls'
  traverse [] ls                    = ([], ls)
```

Fig. 3. Template Haskell generation of `spread` for any tuple nesting (`buildP` omitted)

a single `workerstream` which is `spread` to yield a tuple of streams instead of a stream of tuples. Fig. 3 shows how Template Haskell [7] is used to flexibly create the needed version of `spread`. New requests are appended to the list of pending requests while new tasks are added to the task pool which gets transformed by `tt`. One could in principle also extract the `results` out of `spread` via `x`; but this would result in non-termination since after processing all tasks the master would wait forever for further worker messages containing new tasks.

Therefore the *state part* has been introduced to care for termination detection and result accumulation. The `terminate` function carries a state consisting of a set of incomplete tasks, the number of complete tasks in the task pool, the accumulated results, and the number of accumulated results. In addition to the continuous evaluations in the stream part, `terminate` traverses `workerstream` a second time in a stepwise fashion. For every answer from a worker process, `terminate` will run `tt` on the incomplete tasks extended by the received new tasks and update its `t` counter accordingly. The result counter `r` is incremented by 1, as every answer delivers exactly one result. If then the new counters `t'` and `r'` are equal, which means that for every complete task issued to the task pool a result has been received, the workpool terminates giving back the reversed list of results. If, on the other hand, `t' > r'`, then the remaining incomplete tasks together with the new counters and the result list will be used for a tail-recursive

call to `terminate`. The remaining case `t' < r'` can never happen since every step will yield only one result.

When constructing a proper task pool transformation function `tt` for the workpool one has to be careful because:

- In the stream part `tt` is applied once to a stream of tasks while in the state part it is applied many times to a finite task pool. It has to behave correctly in both situations.
- As interdependent task and result streams are used it is necessary to produce as much output as possible with as few inputs as possible. Therefore delayed matching (via the lazy matching operator `~` or selection functions `head` and `tail`) and the earliest possible production of results should be used.
- Transformation often means combination or comparison which implies searching the task pool. As the task pool is potentially infinite one runs the risk of searching for (and then blocking on) not yet existent tasks.

As `tt` will often in some way have to combine incomplete tasks to complete ones, we present in Fig. 4 a predefined function `ttransform` for doing this while taking some care of the aforementioned dangers. One has to provide only two arguments to `ttransform` to get a full version of `tt`: Firstly, a predicate `cp`, which checks whether a given task is *complete* or not. Secondly, a function `co` which takes a set of mixed complete and incomplete tasks and tries to *combine* as many incomplete tasks as possible. Its results are the already complete tasks together

```
ttransform,ttransform2 :: (t -> Bool) ->              -- complete, cp
                          ([t] -> ([t],[t],Int)) ->   -- combine,  co
                          ([t], [t], [t], Int) -> ([t], [t], [t], Int)
ttransform cp co old@(tasks, incomplete, complete, n)       -- Step 1
  = if (not (null incomplete))
    then let (ct,it,d) = co incomplete
         in if (not (null ct))
            then ttransform  cp co (tasks,it,ct++complete,n+d)
            else ttransform2 cp co old
    else ttransform2 cp co old

ttransform2 cp co (t:ts, incomplete, complete, n)           -- Step 2
  | cp t      = (tts1,iis1,  t:ccs1, d1+1)
  | otherwise = (tts2,iis2,ct++ccs2, d2+d)
    where (tts1,iis1,ccs1,d1) = ttransform cp co
                                (ts, incomplete, complete, n)
          (ct,it,d)           = co (t:incomplete)
          (tts2,iis2,ccs2,d2) = ttransform cp co (ts, it, complete, n)
ttransform2 cp co ([],incomplete, complete, n) = ([], it, ct, n+d)
  where (ct,it,d) = co incomplete
```

Fig. 4. Higher-order function `ttransform` for task pool transformation

with the newly completed tasks, the currently not combinable incomplete tasks, and the number of newly generated complete tasks.

To avoid the above mentioned danger of blocking when trying to find partners for incomplete tasks we will make only a single traversal over the task list and use an accumulator to carry not yet combined tasks with us. For that purpose `ttransform` carries a state argument consisting of the remaining task stream, the accumulator, a stream of complete tasks (its result), and the number of new complete tasks (needed to correct termination detection counters). `ttransform` is divided in two steps: The first step postpones any matching on the input task stream and tries to combine incomplete tasks inside the accumulator as long as possible. Only if that fails, it matches the first task of the task stream and acts depending on its completeness. Complete tasks are immediately passed to the output stream while incomplete ones are tried to be completed. By considering data dependencies the user has to make sure that enough complete or completable tasks are generated in the right order by his application.

2.3 The Nested Workpool

A growing number of workers or tasks induces heavy traffic at the master process which then apparently quickly becomes a bottleneck for the whole workpool scheme. This can be avoided by having more independent workers which manage a buffer of tasks for themselves. In other words: We will replace each worker by another workpool for local task distribution. Fig. 5 shows the code for such a nested workpool with an even arbitrary nesting depth ≥ 1. For depth 1 the previously defined workpool is returned. The depth is controlled by the (equal) length of the first three argument lists which contain the number of workers (or submasters respectively), the prefetch, and the task transformation function for each level of the workpool tree. The nesting itself works by folding the zipped arguments for each level with the `wpool` function. The worker function `f` is used to close the workpool tree with a set of worker leafs. Note the use of `repeat`: No submaster will migrate tasks to masters above him, therefore newly created tasks will only be sent by the worker leafs to their respective master processes. Fig. 6 shows an example call of the nested workpool together with the resulting

```
wpN :: (Trans t, Trans r) =>
  [Int] -> [Int] ->                                -- #workers, prefetches
  [(([t],[t],[t],Int) -> ([t],[t],[t],Int))] -> -- transformations
  ([t] -> [(r,[t])]) ->                            -- worker function
  [t] -> [r]                                       -- tasks, results
wpN ns pfs tts f initTasks = results where
  (results,_)      = unzip ((foldr fld f (zip3 ns pfs tts)) initTasks)
  fld (n,pf,tt) wf = \ts -> zip (wpool n pf wf tt ts) (repeat [])
```

Fig. 5. Nested workpool

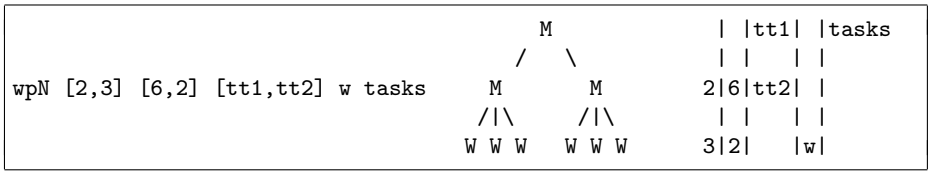


Fig. 6. Two-level example call of `wpN` with process tree and argument distribution

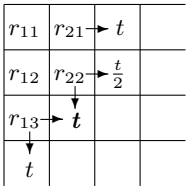
process tree; additionally for each argument it is shown to which level it applies. The termination detection of `wpool` fits smoothly into this nesting.

3 Case Study: Parallel Sequence Alignment

We have used the extended workpool of Sect. 2.2 to parallelise the alignment of DNA sequences via the linear Needleman-Wunsch [8] algorithm. Although not being very efficient, the algorithm serves as a good example for wavefront parallelism [9]: Within a matrix structure the algorithm exhibits diagonal wavefront dependencies (see Fig. 7) which can be expressed as tasks for execution via the extended workpool. More specifically: Each block depends on its two left and upper neighbours in the matrix. Therefore each result produces incomplete tasks for its right and lower (not yet computed) neighbours. Two of these incomplete tasks will then be combined in the task pool to form a new complete task. Only elements of the first row and the first column can be computed given only one of their respective neighbours.

Fig. 8 shows on the left the relative speedup of the parallel sequence alignment algorithm using the extended workpool. All measurements were taken on a cluster of nine Linux PCs connected via 100 Mbit ethernet. The PCs are not completely identical, but this is compensated by the dynamic task distribution of the workpool. Sequences of length 10.000 have been tested with a varying block partitioning. The figure shows that a medium task granularity (block size 500) has paid off the most in our experiments. Larger tasks result in task shortage, while smaller tasks induce too much administrative overhead due to their large number. The nested workpool cannot be used to reduce that overhead, since tasks of different subworkpools would have to be combined. We are aware that our unoptimised implementation of a suboptimal algorithm is slower than modern imperative alignment solutions; it nevertheless serves as a good example of wavefront parallelism for our workpool.

Fig. 8 shows on the right an activity diagram for the execution of the parallel sequence alignment on nine processors (length 10.000, block size 500). Each row represents the activity of one processor during execution, starting on the left and ending on the right at around 45 seconds. White areas represent phases of inactivity or blocking on not yet available data (combined for better visibility), while black areas represent active computation or communication. The lowest row (processor 1) contains the master process which shows constant activity in



Creation of task t_{23}:

1) t_{13} ends, yields r_{13}, releases t_{14} and $(\frac{t}{2})_{23}$ into the task pool
2) t_{22} ends, yields r_{22}, releases $(\frac{t}{2})_{23}$ and $(\frac{t}{2})_{32}$ into the task pool
3) `tt` creates t_{23} in the task pool
4) t_{23} will later release $(\frac{t}{2})_{24}$ and $(\frac{t}{2})_{33}$ into the task pool

Fig. 7. Task creation for parallel sequence alignment ($\frac{t}{2}$ represents an incomplete task)

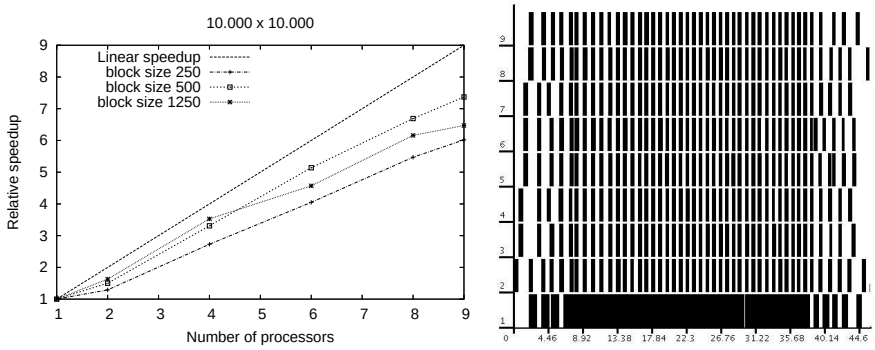


Fig. 8. Relative speedups and activity diagram (sequences of length 10.000, 9 nodes)

distributing and combining tasks. The remaining rows show the activity of the worker processes. These are evenly loaded with tasks. Both start and end phase of the computation show clearly the growing and shrinking task availability induced by the diagonal wavefront traversal of the matrix described in Fig. 7.

4 Related Work

An older survey by Stephens [10] describes approaches to stream programming in general. Kahn et al. already described in [11] a model of functional processes communicating via streams. Also the big complex of Dataflow languages [12] has to be mentioned in the context of stream programming. In [2] we have already shown a basic workpool skeleton for Eden which we have extended by dynamic task creation, task pool transformation, and termination detection in this work. Martínez and Peña describe in [13] another workpool scheme for including dynamic task creation. Their approach also introduces state for the master and worker processes aiming at branch-and-bound algorithms which is not covered by our approach. However, only the master is allowed to create new tasks; furthermore they implement the scheme via continuous state updates and do not offer task pool transformation. Regarding our application we have not been able to find another application of parallel functional languages to DNA sequence alignment. A non-parallel application of Haskell, however, has been described in [14].

5 Conclusion

We have developed a new generalised workpool skeleton for the parallel Haskell dialect Eden by adding two features for dynamic task handling: Firstly, worker processes can generate tasks and insert them into the global task pool dynamically. This requires a more complicated termination detection which we have solved by a counting mechanism. Secondly, the master process is enabled to traverse and to transform the task pool to cope e. g. with incomplete tasks. To ease the definition of such functions we have given a function for task pool transformation. We then presented a way to nest the workpool skeleton to lower the administrative load of the master process by introducing additional submasters. Finally we have applied the skeleton to parallel DNA sequence alignment which yielded good relative speedups.

Acknowledgments. The author thanks Rita Loogen for carefully reading the paper and Hans-Philipp Annen, Simon Göbel, and Simon Wiesler for their work on the parallel sequence alignment.

References

1. Peyton Jones, S., et al.: Haskell 98: A Non-strict, Purely Functional Language (2003) See: `http://www.haskell.org/onlinereport`.
2. Klusik, U., Loogen, R., Priebe, S., Rubio, F.: Implementation Skeletons in Eden: Low-Effort Parallel Programming. In: IFL 2000, Aachen, LNCS 2011. (2001)
3. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press (1989)
4. Rabhi, F.A., Gorlatch, S.: Patterns and Skeletons for Parallel and Distributed Computing. Springer-Verlag (2003)
5. Loogen, R., Ortega-Mallén, Y., Peña, R.: Parallel Functional Programming in Eden. Journal of Functional Programming, Special Issue on Functional Approaches to High-Performance Parallel Programming (2004)
6. Oak Ridge National Laboratory: Parallel Virtual Machine (2003) See: `http://www.csm.ornl.gov/pvm/pvm_home.html`.
7. Sheard, T., Peyton Jones, S.: Template Meta-programming for Haskell. In: Haskell Workshop 2002, ACM Press (2002)
8. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. (1970)
9. Anvik, J., et al.: Generating Parallel Programs from the Wavefront Design Pattern. In: Proceedings of the 7th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'02). (2002)
10. Stephens, R.: A survey of stream processing. Acta Informatica **34** (1997) 491–541
11. Kahn, G., MacQueen, D.: Coroutines and Networks of Parallel Processes. In: IFIP 77, North Holland (1977)
12. Hanna, J.R.P., Johnston, W.M., Millar, R.J.: Advances in dataflow programming languages. ACM Computing Surveys **36** (2004) 1–34
13. Martínez, R., Peña, R.: Building an Interface Between Eden and Maple: A Way of Parallelizing Computer Algebra Algorithms. In: IFL 2003, Edinburgh. (2004)
14. Giegerich, R., Kurtz, S., Weiller, G.: An Algebraic Dynamic Programming Approach to the Analysis of Recombinant DNA Sequences. In: Proceedings of Workshop on Algorithmic Ascpects of Advanced Programming Languages, Paris. (1999)