Parallel Computation Skeletons with Premature Termination Property

Oleg Lobachev

Fachbereich Mathematik und Informatik,
Philipps-Universität Marburg,
D-35032 Marburg
Fax: +49 (0) 6421 28 25466
lobachev@mathematik.uni-marburg.de

Abstract. A parallel computation with early termination property is a special form of a parallel **for** loop. This paper devises a generic high-level approach for such computation which is expressed as a scheme for algorithmic skeletons. We call this scheme `map`+`reduce`, in similarity with the `map-reduce` paradigm. The implementation is concise and relies heavily on laziness. Two case studies from computational number theory support our presentation.

Algorithmic skeletons are parallel algorithm abstractions, which concentrate on the parallelisation and not on the algorithm [1]. We regard here skeletal implementation of a parallel computation with early termination. An imperative equivalent is a **for**−**break** parallel loop. An overview is in Section 1. This paper discusses an implementation of this kind of a loop in lazy parallel functional programming language Eden [2]. We will present this language briefly in Section 2. We devise a classification of new and existing related skeletons in Section 3. We develop there a more special approach. The desired parallel behaviour has a well-known functional counterpart: a `map-reduce` combination with added possibility of premature termination. This premature abort part is new. We find some features of the Eden language useful in this context, as the premature termination is granted for free in our setting. We discuss it in Section 4. Section 5 highlights two primality tests used as examples. We do not discuss their implementation in full detail and focus on parallelisation results. Section 6 discusses further related work. Section 7 concludes and outlines future work.

Contributions. The contributions of this paper are

- Definition of the `map`+`reduce` skeleton scheme for the parallel computation with premature termination.
- Discussion of the importance of laziness in the context of speculative parallelisation and premature termination.
- High-level parallelisation of two primality tests. We were the first to parallelise one of them, the Jacobi sum test.

1 Introduction

Iteration is an important concept in the computation theory, especially in scientific computation. For example, a **for** loop or a **while** loop are ubiquitous in imperative programming. The purely functional counterpart of these constructs is often expressed using recursive calls or corresponding combinators, e. g., `map` and `fold`, or with a contraction, e. g., `iterate`.

A special kind of "repeated computation", as described below, is the goal of this paper. A classical imperative **for** loop can be implemented in parallel in some cases. Assume that the loop is executed n times. If, for some positive integer p, the loop body can be partitioned into $\lceil n/p \rceil$ blocks, and if in each of these blocks single loop iterations do not depend on each other, then the loop can be implemented in parallel on p processors. This scheme is known as *speculative* parallelisation. The first p loop iterations are executed in parallel, then the second p loop iterations follow, until the $\lceil n/p \rceil^{\text{th}}$ block of at most p loop iterations is executed. We will regard the case when computation can be aborted before the loop is over as the parallel functional equivalent of the **for** loop with a conditional **break**.

Such kind of a computation is a quite typical task in computational mathematics. Given some data $x_1, x_2, \ldots$, we apply some function f to them, resulting in $f(x_1), f(x_2), \ldots$, until a predicate p is not satisfied for some $f(x_n)$. See Section 5 for presentation of the particular methods from computational number theory. We abbreviate the processing elements of a parallel system as PEs. Nowadays the PEs are typically cores of a multicore system. Below, we investigate combinations of three distinct features of the algorithmic skeletons: task creation, task balancing, and premature abort.

The first feature—dynamic *internal task creation*—allows creation of new tasks from within the skeleton while the computation is progressing. Essential for this is the opportunity to use intermediate results *inside* the skeleton and to *transform* its task pool. In absence of these facilities, the ability to create tasks from the *final* results is available with a wrapper around standard map skeletons in a lazy functional language. We call this dynamic *external task creation*. This approach is similar to non-monadic I/O in Haskell and will be elaborated on later. If all tasks are known beforehand and no new tasks emerge, *no task creation* takes place.

Task balancing is how the tasks are allocated in the progress of computation in order to maintain a similar load on all PEs. *Dynamic task balancing* is a prime feature of workpools (see below). It is another name for *load* balancing, as in this case we do indeed balance the actual load. Here the decisions are made at runtime. In contrast, we can balance the tasks not while the computation is in progress but beforehand. We call this *static task balancing*; it is implemented in a task farm. Finally, a simple parallel map does no task balancing by itself, but merely creates all tasks simultaneously. Still, the OS could load-balance the simultaneously created threads, however this is beyond the scope of this work.

We discuss the premature abort feature in Sections 3 and 4. The essential idea is to terminate a computation in progress basing on available information.

Twelve combinations of the above distinct features are possible. We will address only these with premature abort property. The no task creating skeletons can be easily upgraded to dynamic external task creation with a wrapper. Author's thesis [3] discusses such an extension for a `farm` skeleton in Chapter 6. Further extensions are known, see, e.g., the `orbit` skeleton [4]. The literature is mainly concerned with three internal task creation versions, see page 5 for a discussion. We define and use here a generic scheme for three *no* task creation versions.

2 Eden

Eden [2] is a parallel Haskell extension that features explicit processes creation. We assume a certain degree of familiarity with Haskell [5]. Laziness is important in Eden, as we will see below.

Eden processes. The parallelism model in Eden is implemented with *processes* that are executed on remote machines. Before that, a process needs to be defined. The *process abstraction* is defined with the smart constructor function `process`. The process abstraction is a mould from which multiple "actual" processes can be obtained. Processes are started with the instantiation operator `#`.

The input is communicated to the started process and the output is communicated back to its parent implicitly. One could say that[1] `(#) ∘ process` in Eden is similar to the `$!` operator from Haskell. Of course, Eden processes have a side effect of a parallel application. Because of the evaluation of data to be sent, `#` is a strict operator.

Why Eden matters. The strong sides of Eden include

- *Distributed memory model.* There is no need for sophisticated virtual shared heap implementations in a distributed setting. Moving the data between Eden heaps on a shared memory machine is essentially copying. An implementation of parallel garbage collection is not required with Eden.
- *No separate skeleton language.* In Eden skeletons are expressed as higher-order functions. There is no separation between skeleton language and application language, which is good for the programmer's flexibility.
- *Usual Haskell traits.* Eden supports everything GHC (the flagship Haskell compiler) supports.

Demand and evaluation control. Haskell is a lazy programming language—expressions are evaluated only if required. The actual GHC implementation determines that Haskell is a demand-driven language: expression is evaluated just when *demand* for its value occurs. Demand on expression that depends on another expression will issue a demand on both expressions. For example: the output of expression value on the display forces demand on the expression.

[1] With ∘ we denote Haskell's composition combinator. It holds `(f ∘ g) x = f (g x)`.

The need for demand is an obstacle to parallelism. If no expressions are evaluated before they are needed, then no parallelism occurs. Thus, most parallel extensions to functional languages with demand-driven evaluation create artificial demand in expressions dealing with parallelism, see, e.g., GpH [6,7]. For the current state of the evaluation strategies' library in Haskell see the paper by S. Marlow et al. [8].

The *transmissible* types in Eden, i.e., types which can be communicated to other machines, must be instances of the `Trans` type class. As Eden semantics forces the evaluation of data before sending, the `Trans` type class instances can be derived only for instances of `NFData`, i.e., types that can be evaluated to normal form. To give an example, the `Int` type is transmissible, it is an instance of `Trans`.

Skeletons in Eden. The Eden application library [9] contains multiple algorithmic skeletons. The skeletons are implemented in Eden as higher-order functions, see [10,11,12]. Related skeleton libraries in other languages were presented in [13,14,15,16,17,18].

Several of Eden skeletons have a type signature similar to that of a `map`. An overview is in Table 1. The simplest one is `parMap`. It has exactly the `map` signature—up to the `Trans` context. So, let

```
type Map a b = (a → b) → [a] → [b],
```

then we can write both `map :: Map a b` and `parMap :: (Trans a, Trans b) ⇒ Map a b`. The semantic equivalent of `parMap f` is `zipWith (#) (repeat $ process f)`. We see that a process is created for each element of the input list. This approach is not efficient in cases where the list length exceeds the number of PEs. The next skeleton relaxes this restriction. We will use the `map`-like skeletons throughout this paper.

A classical approach in parallel computing is *farm* [19,11]. It can be implemented using a *statically task-balanced* parallel map: we divide the input into blocks, which are assigned to PEs. Thus, no more processes than PEs are created, but each process typically processes multiple tasks. This approach works best if the processing time of individual tasks is identical. The type of the `farm` skeleton is the same as that of a `parMap`.

Workpools. The *dynamically task-balanced* parallel map is an example of a master-worker computation scheme—the more generic name is *workpool*. The cost of processing separate elements may vary. Hence, it is better to do load balancing at execution time [20,21]. These skeletons have the same type as the parallel map.

skeleton	task balancing
`parMap`	none
`farm`	static
`workpool`	dynamic

Table 1. An overview of `map`-like skeletons presented here.

We can classify workpools by whether it is possible to *create tasks dynamically* while the computation progresses. This is done using a worker function not of type `a → b`, but of type `a → (b, [a])`. In other words, worker function might produce new tasks, but not necessarily in

```
farm+and :: Trans a ⇒ (a → Bool) → [a] → Bool
farm+and f xs = and $ farm f xs
```

Fig. 1. A simple skeleton with premature termination but without task creation.

each run. Dynamic task creation is possible for both master-worker-based (for Eden implementations see [22,4]) and distributed workpools (*viz.* [23]).

Iteration skeleton. An iteration skeleton `iterUntil` has been defined in [12]. This skeleton allows introduction of further iterations *while* the loop is already being executed. Its sequential imperative counterpart is a **do**–**while** loop. We start a few speculative tasks in parallel and repeat this over and over again. It is possible to produce more tasks during the computation.

This skeleton has dynamic task creation, but no task balancing. We do not elaborate on it, see [12] for details.

3 Skeletons With Premature Termination: The Beginning

Starting point. In case of using a functional lazy programming language and opting for no task creation, we can sketch much of the skeleton with premature termination. Our simple Eden implementation, called here `farm+and`, is shown in Figure 1.[2] We emphasise that this version *does* have task management and can abort the computation early, but it *disallows* internal task creation. The version shown is *very* simple and assumes worker functions of type `a → Bool`. A worker function returning a `False` causes collapse of the skeleton. This happens because of properties of the `and` function [5]. Namely, `and [True, False, ⊥] = False`. The `and` function stops traversing the list, once a `False` occurs. A call to the `farm` skeleton does the actual parallel evaluation. It is straightforward to generalise the `farm+and` skeleton: we replace `and` with a parameter function. We call the resulting skeleton `farm+reduce`. Further generalisation is presented in Section 4.

Related work. Another approach to parallel functional "repeated computation" is to use an advanced `workpool` with dynamic internal task creation. As the name suggests, dynamic load balancing is already supported in this setting. Two possibilities for premature termination exist. One is to extend the skeleton with state management. However, if *global* task pool transformation is available, a stateless version is possible. Such transformations are often used as a generalisation for new tasks placement. It suffices to introduce a new, special task that will be issued by the worker function as a termination signal. The task pool transformation function would simply empty the task pool seeing such a task. So, the purge happens here on the input side. However, our approach is different.

We display a classification of existing `map`-like skeletons with the possibility to abort the computation early in Table 2. We differentiate among them by method of task creation and by the type of task balancing. The skeletons shown here are

[2] Note that strict Haskell syntax disallows such function name, so we use `farmPlusAnd` in the actual implementation.

		task creation	
		internal	external/no
task balancing	no	[12] (`iterUntil`)	`parMap+reduce`
	static	[4]	`farm+reduce`
	dynamic	[22,24,4]	`workpool+reduce`
	abstracted	not known to us	`map+reduce`

Table 2. The `map`-like skeletons with premature termination property.

identified by their names. The skeletons known in the literature are referenced by the appropriate citation. We observe that internal task creation skeletons with dynamic task balancing have been quite intensely studied, since they comprise an extension to the well-known workpool scheme. The skeletons `parMap+reduce` and `workpool+reduce` are equivalents of the `farm+reduce` skeleton with an obvious change in the `map` implementation. In the next section we will generalise these skeletons under an umbrella approach called `map+reduce`.

The key difference between our approach and the naive parallel **while** construct, like parallel_do from Intel Threading Building Blocks library [25], is the `reduce` function, accounting for the termination of the computation. Both in the dynamic task creation workpools and in parallel_do the termination of the computation is implemented with emptying the task queue. Our approach implements the termination by not demanding the results anymore.

4 Map+Reduce

Given the higher-order function `farm+and` from Figure 1, we aim to generalise it. This specific implementation of a parallel **for**-loop turns out to be nothing more than a primitive combination of parallel `map` and a specific `reduce` function. Note that `reduce` is another name for a `fold`. However, we have further requirements for the `reduce` function. These are satisfied by `and`, but not by an arbitrary `reduce`. We call the `map`–`reduce` combination incorporating some parallel `map`-like skeletons and a special `reduce` implementation a `map+reduce` skeleton scheme. This generic scheme is our original contribution.

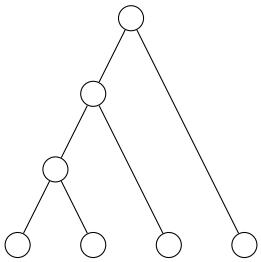

Fig. 2. A standard `reduce`.

The key feature of our Eden implementation should also be granted in a generic `map+reduce` case: if our `reduce` function returns the result before evaluating the whole input list, e.g., `and [True, False,` $\perp$ `] = False`, then the parallel `map` instance still computing further list entries should be terminated. In the following we seek for a formalisation of this issue and develop a generic skeleton using the gained knowledge.

Our required property for `reduce` is *shortcutting*. If some input element produces all the information necessary to produce the final result, then no further input elements need to be evaluated. Figure 2 shows the standard `reduce`, Figure 3 displays such special input element and information propagation in red.

```
map+reduce :: (Trans a, Trans b, Trans c)
           ⇒ (Map a b)                              -- ^ map
           → ((c → b → c) → c → [b] → c)  -- ^ reduce
           → (a → b)                    -- ^ map worker
           → (c → b → c)                -- ^ reduce worker
           → c                          -- ^ reduce zero
           → [a] → c                    -- ^ input and output
map+reduce amap areduce f g z xs = areduce g z $ amap f xs

-- a version with applied worker functions
map+reduce' :: (Trans a, Trans b, Trans c)
            ⇒ ([a] → [b]) -- ^ partially applied map
            → ([b] → c)   -- ^ partially applied reduce
            → [a] → c     -- ^ input and output
map+reduce' = flip (∘)
```

Fig. 4. A generic map+reduce skeleton

Note that it is not possible to encode shortcutting in Haskell's type system, rather like the inability to encode laziness in its type system.

We stress again that map+reduce that we describe here differs from map-reduce skeleton. The latter assumes distributed reduce. Also, the data flow in map-reduce leaves the data distributed between the map and reduce phases. This is one of the main reasons for its success, see [26,27]. However, in our case a special reduce is needed. We use the sequential implementation of reduce and do not employ a distributed reduce with shortcutting. Since we use reduce to control termination, it is acceptable to use sequential implementation: the actual processing is done with the (parallel) map. The time spent in the reduce phase is negligible, compared with the time spent in the parallel map. This is another difference of our map+reduce scheme from map-reduce.

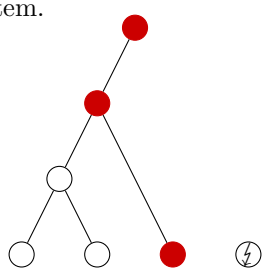

Fig. 3. A standard reduce with shortcutting. The computation at ↯ has no demand.

Following the example in Figure 1, we aim to implement a more generic map+reduce skeleton. In effect, we need to abstract the reduce function. This is done by adding another parameter. The implementation is surprisingly simple, see Figure 4. The map+reduce function receives a skeleton for map (parameter function amap) and an implementation of a reduce (parameter areduce), as well as corresponding parameters (f, g, z), and simply builds them together, such that areduce consumes the output of the amap skeleton.

The version with already applied worker functions, called map+reduce', takes the map and reduce functions, which are already partially applied to their respective arguments. Let paMap be the partially applied map and paReduce be the partially applied reduce. Such partial application yields the types [a] → [b] and [b] → c, appropriately for the paMap and the paReduce functions. For paReduce to consume the output of paMap we need to write

```
farm+reduce' :: Trans a ⇒ (a → Bool) → [a] → Bool
farm+reduce' f = map+reduce' (farm f) and
```

Fig. 5. Implementing `farm+reduce` in a simpler way

```
-- simplified
class (AddMonoid a, MultMonoid a) ⇒ Ring a where
  zero :: a
  unity :: a
  add :: a → a → a
  mult :: a → a → a

instance Ring Int where ...     -- instantiation is trivial
instance Ring Bool where ...
```

Fig. 6. Commutative rings in Haskell.

```
map+reduce' paMap paReduce xs = paReduce (paMap xs)
```

which is written in the point-free style as `flip (∘)`, seen in the bottom of Figure 4. Note, however, that the type of `map+reduce'` is more constrained than that of the composition operator ∘.

The `farm+reduce` skeleton can be implemented using this generic skeleton, as shown in Figure 5. Note that `map+reduce` is a skeleton to build on other skeletons.

A different view on the `map+reduce` functionality is based on streams [28,29,30]. To recapitulate, the idea of a stream is that of a (potentially infinite) lazy list. A stream processing function can handle streams efficiently. The premature termination feature of the `map+reduce` skeleton operates only if both its components are stream processing functions.

Laziness. The `map+reduce` works because of a special property of the `reduce` function that we now examine in detail. We define (commutative) rings in Figure 6. It is not quite the algebraic definition, but it suffices for our purpose. Additionally, it is easily encoded in Haskell's type system. The `zero` (`unity`) is the neutral element w.r.t. `add` (`mult`). Also, `zero` is the absorbing element w. r.t. `mult`. The commutativity is not required here, we just spare two different declarations for left and right `fold`.

We define `reduce`, or, to be more exact, a `fold` for the multiplicative operation, but keep in mind the possibility of a `zero` occurrence. The shortcut case is nothing other than an occurrence of zero in a product-based `fold`. We stress that the only required property to implement premature termination is that the folding operator has a zero absorbing element. It is the left zero in case of the left `fold`:

```
lfold :: (Ring a, Eq a) ⇒ [a] → a
lfold xs = lfoldAcc xs unity
  where lfoldAcc (x:xs) acc
```

```
        | x==zero  = zero -- sic!
        | otherise = lfoldAcc xs (mult acc x)
    lfoldAcc _ acc = acc
```

Assuming the laziness of `mult` in its second argument, i.e., if `mult zero` $\perp$ `= zero` holds, we can write a similar right `fold`:

```
rfold :: Ring a ⇒ [a] → a
rfold (x:xs) = mult x (rfold xs)
rfold [] = unity
```

From now on we base our presentation on the `rfold` code. We have no special comparison with `zero` here, instead `rfold` relies on `mult` to preserve zero and not to evaluate the recursive call. In other words, `zero` needs to make `mult` non-strict in its second parameter. To generalise even more: with `zero'` $\neq \perp$

`mult zero` $\perp$ `= zero'`

should hold. This means, that upon a `zero` occurrence the second parameter of the mult should be not evaluated, but the result of `mult` in this case could be something different from `zero`.

In the above example in Figure 1 the `mult` function was the `&&` operator for booleans and `zero` was `False`.

Connection to **or** *parallelism.* The premature termination case has similarities to the **or** parallelism in parallel logic languages [31]. We regard the deterministic case here. As soon as termination signal is issued, all other running and pending computations are not required anymore. In our case the **or** parallelism may happen only in the outer loop of the computation; the termination signal stops the whole computation.

Byline. The premature termination feature of `map`+`reduce` skeletons is granted for free in a lazy setting if the shortcutting property of the `reduce` holds. The `map`+`reduce` scheme is a new, concise functional implementation of the **for**–**break** imperative pattern.

5 Parallelisation Results

Our test cases originate from computational number theory [32]. The thesis [3] studies such algorithms and their parallelisation in detail.

Both cases presented in this paper are concerned with probabilistic primality testing. Such tests consist of many smaller checks. If one of these checks fails, the complete test can be aborted. All checks are known beforehand.

Speedup. One of the most crucial measures of parallel performance is the speedup. It is defined as the quotient of the sequential and parallel execution time of a program. The ideal speedup is the *linear* speedup. It occurs if the program scales perfectly, e.g., it is 10 times faster on 10 processors, compared to its speed on a single processor. In a contrast to this ideal value, the speedup values computed from the measured execution times are sometimes referenced here as the *observed* speedup values.

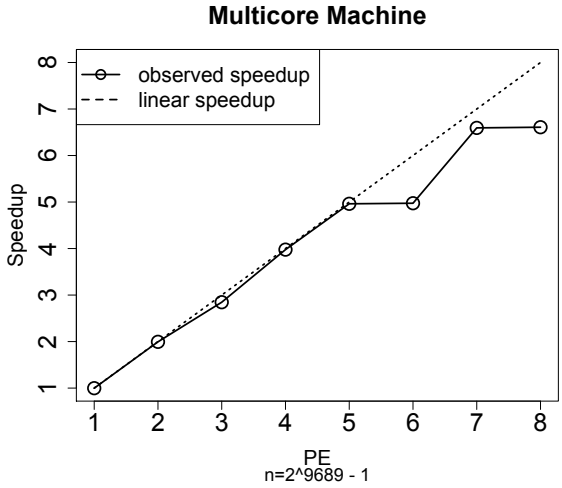

Fig. 7. Speedup of Rabin–Miller test on a multicore machine.

Important for the speedup is the choice of the sequential reference value. In the following, if we use the run time of the sequential pure Haskell program as the sequential value, we obtain the *absolute* speedup. If merely the time measured with a parallel program on a single PE is used, we call it the *relative* speedup.

Setup. The test programs were executed on multiple machines. The "multicore machine", referenced below, is an 8-core dual Intel Xeon platform with 16 GB RAM, running 64 bit CentOS Linux. The "network of workstations" is a set of workstations at the Faculty for Mathematics and Computer Science in Marburg. Each workstation is an Intel Core2Duo machine with 2 GB RAM. Workstations are interconnected with Fast Ethernet and run 64 bit Fedora Linux OS. We used the Eden implementation atop of GHC 6.12. The same GHC version was used to produce the sequential reference binary. Each test program in every configuration was run 5 times, we took the average run time.

Rabin–Miller test. The Rabin–Miller test is a probabilistic primality test that typically involves 20 checks [33,34]. If a check fails, the input number is definitely not a prime. However, a passed check is not a guarantee of primality: the probability of a false positive is 1/4 for an individual check. However, if all checks are satisfied, then we have a very high probability that the input is indeed prime.

The parallel speedup of our implementation is displayed in Figure 7, we used the eight core multicore machine. The speedup is 6.61 on 7 PEs with input size $2^{9689} - 1$. The best speedup value is 6.63 for the input size $2^{11213} - 1$ on the same number of cores. The sequential program in this case took 71.13 seconds to complete. The efficiencies are 0.944 and 0.947 respectively. Note the linear speedup for up to 5 PEs and the erratic behaviour for 6–8 PEs. The reason for the latter is not bad parallelism but problems involving task placement. We always issue 20 tasks corresponding to the checks in the test. In some configurations this may lead to problems. 10 PEs sounds like a good configuration, we show

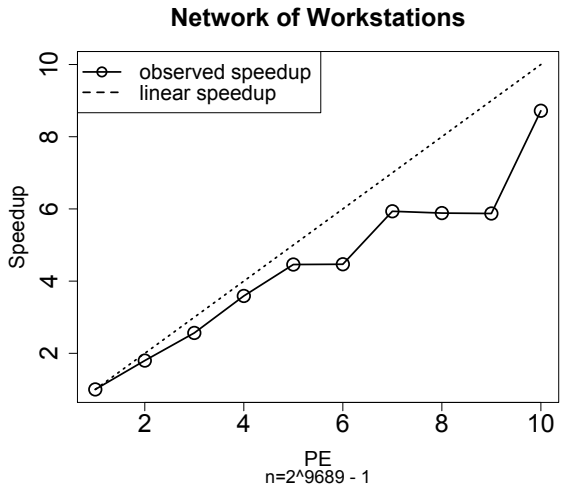

Fig. 8. Speedup of Rabin–Miller test on networked workstations.

the speedup of the same program on a network of workstations in Figure 8. The resulting speedup is 8.72 for 10 PEs. The presented result supports our reasoning on the speedup behaviour: the speedup improves greatly at 10 PEs.

Jacobi Sum Test. The Jacobi sum test, also known as APRCL [35,36], is a sophisticated primality test. In contrast to the Rabin–Miller test, if it acknowledges a number as a prime, then it is indeed a prime number. However, the test may fail in a very few circumstances. See [32] for a detailed presentation of Jacobi sum test.

The key idea of the test is to check for a generalised small Fermat condition in a carefully designed extension field. There are multiple such checks in an individual run of the test. The input for each check can be encoded as a pair of primes (p, q) and the output is written in the output set l_p, which depends on p. We have shown in [3] that the individual checks do not depend on each other. Hence it is possible to run them simultaneously and to merge the resulting sets l_p afterwards.

We applied our `map`+`reduce` scheme to effect the parallelisation of Jacobi sum test. The single checks were expressed in the working function of the `map`, the merge of the l_p sets and the termination control was done in `reduce`. Figure 9 shows the speedup curves. These results were obtained on the eight core Xeon machine described above. We observed a confident, even if not fully linear speedup growth. We used the same compiler. To facilitate dynamic load balancing, a `workpool` skeleton was used in the `map` phase. The reason for this design decision was the extremely different run time of the checks.

In the test implementation we successfully solved some technical problems, both on the mathematical side (e.g., implementation of adjunction) and in parallel computing (e.g., extreme task imbalance in the naive implementation). We do not give the details here.

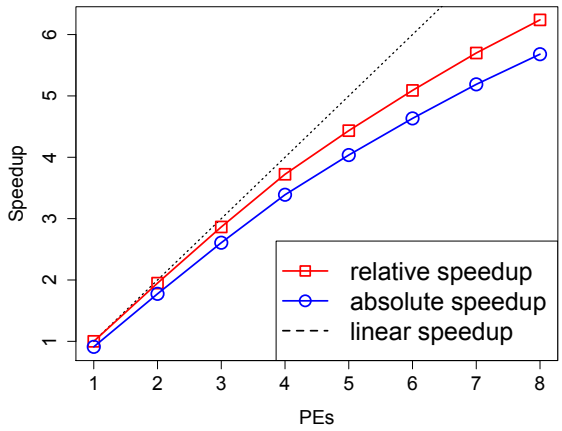

Fig. 9. Speedups for Jacobi sum test. The input was $2^{1279} - 1$. Tests were performed on a multicore machine. Red line denotes relative speedup, blue line shows absolute speedup.

Benefits from `map`+`reduce`. To underline the benefits from our early termination feature, we executed Rabin–Miller test for the input value $2^{9690} - 1$ using the parallel implementation on 7 cores. The test returned the correct negative result and was 3.93 times faster than the same program in the same setting for the input value $2^{9689} - 1$ with the positive result. Note that the non-prime input was almost the double of the prime input, but the termination made the program almost four times faster.

We will present a similar result for the Jacobi sum test using Eden Trace-Viewer [37]. It is possible to instrument Eden binaries with the output of the current parallel activity: each parallel process logs whether it is running, could run, or is blocked in the current moment. These logs, called *traces*, can be *visualised* after the program termination. The visualisation diagram shows the time in the horizontal axis and the PEs in the vertical axis. The processes are visualised as horizontal bars, emphasising the process start and termination. The colour of the bar encodes the state of the process. It corresponds to the traffic light: red is blocked, yellow is runnable, green is running. The message exchange between processes can be visualised with arrows. As expected, the arrow tip designates the receiver.

Two trace visualisations of particular Jacobi sum test executions are shown in Figure 10. The visualisations are displayed with the same time scale. The prime number candidate used as the input was $\approx 2^{607}$. In the top visualisation, the number was composite. We see, the computation terminated in 0.55 seconds. The bottom diagram shows the same test in the same setting, but for a prime number input. It takes 2.5 seconds to complete. Thus, the the premature termination property in a *parallel* implementation of Jacobi sum test benefits us with a speedup of 4.55.

The next example is based on message arrival times from a non-shown trace diagram of the full Jacobi sum test on 24 networked workstations. We regard

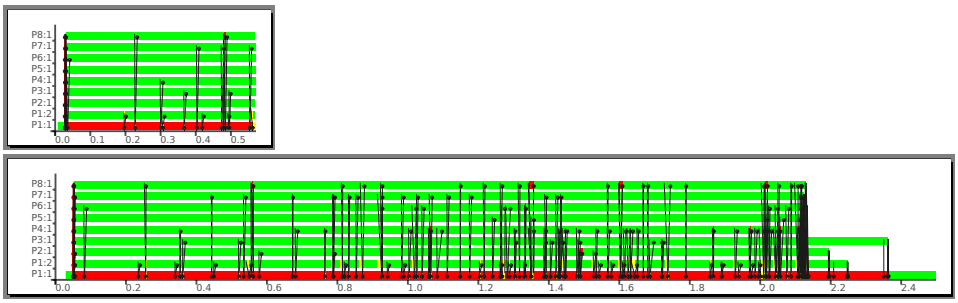

Fig. 10. Two trace visualisations for Jacobi sum test. Top: premature termination, bottom: full run.

a smaller example to simplify the analysis: the larger is the input, the more tasks are issued. When compared with a *parallel* implementation without the premature termination property, our implementation would yield the speedup of 7.34, 5.24, 2.82, or 2.62 for the pivotal task completed at 0.15, 0.21, 0.39, or 0.42 seconds appropriately. Comparing this data with the sequential implementation without premature termination, we would obtain the speedup of 69.11, 49.36, 26.58, or 24.68 appropriately.

6 Further Related Work

We discussed related skeleton approaches in Sections 2 and 3, see also Table 2. The skeletons were initially introduced by Cole [1]. To our knowledge, no one has explicitly addressed premature termination with skeletons. However, the *poison* concept of Hoare's CSP [38,39] is related to our premature abort notion.

Laziness is one of the main features of Haskell [5]. Stream processing functions were regarded by Wray and Fairbairn [29]. One of the first publications on this topic is by Friedman [28].

Rabin–Miller test has been parallelised in [40,41,42]. These papers focus on (parallel) hardware design. In contrast, we suggested a high-level parallelisation with the `map`+`reduce` skeleton scheme. Our approach supports premature termination of computation. We are not aware of `map`+`reduce` skeletons and parallel Jacobi sum test presentations in literature.

7 Conclusions and Future Work

From a decision to implement a skeleton with premature termination we have developed the `map`+`reduce` skeleton scheme. We found it sufficient for the parallel implementation of two primality tests with very good parallel performance. For that purpose the possibility of premature abort was important, but granted for free in a lazy parallel functional setting. We quantified the effect of the premature termination to the factor 4.55 at 8 PEs. Features of lazy evaluation account for the concise representation of the presented skeleton scheme. Parallelisation of the Jacobi sum test is a new result. Generic `map`+`reduce` skeleton scheme is a major part of our contribution.

Future work. We would like to apply the presented technique to further algorithms, search problems seem to be good candidates. Laziness was very important in our implementation, we would like to continue our research on the impacts of laziness on parallelism. Tackling the premature termination in the similar manner as presented here, but in the context of the dynamic task creation is an important future extension of our work.

Acknowledgements. This paper is a revised version of Chapter 5 of author's PhD thesis [3]. I thank Rita Loogen, Yolanda Ortega-Mallén, Paul Tarau and anonymous referees for reading earlier versions of this text and improving the presentation.

References

1. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. Research Monographs in Parallel and Distributed Computing. Pitman (1989)
2. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel Functional Programming in Eden. Journal of Functional Programming **15**(3) (2005) 431–475
3. Lobachev, O.: Implementation and Evaluation of Algorithmic Skeletons: Parallelisation of Computer Algebra Algorithms. PhD thesis, Philipps-Universität Marburg (2011)
4. Brown, C., Hammond, K.: Ever-decreasing circles: a skeleton for parallel orbit calculations in Eden. In: Draft Proceedings. TFP'10 (2010)
5. Peyton Jones, S., ed.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2003)
6. Trinder, P.W., Barry Jr, E., et al.: GPH: an architecture-independent functional language. IEEE Trans. Software Engineering (1999)
7. Marlow, S., Peyton Jones, S., Singh, S.: Runtime support for multicore Haskell. ACM SIGPLAN Notices **44**(9) (2009) 65–78
8. Marlow, S., Maier, P., Loidl, H.W., Aswad, M.K., Trinder, P.: Seq no more: better strategies for parallel Haskell. Haskell'10, ACM (2010) 91–102
9. Eden Group: Eden skeleton library. Software package (2012) Retrieved 13.2.2012.
10. Galán, L.A., Pareja, C., Peña, R.: Functional skeletons generate process topologies in Eden. LNCS 1140, Springer (1996) 289–303
11. Peña, R., Rubio, F.: Parallel Functional Programming at Two Levels of Abstraction. PPDP '01, ACM (2001) 187–198
12. Loogen, R., Ortega-Mallén, Y., Peña, R., Priebe, S., Rubio, F.: Parallelism abstractions in Eden. In Rabhi, F.A., Gorlatch, S., eds.: Patterns and Skeletons for Parallel and Distributed Computing. Springer (2003) 71–88
13. Darlington, J., Field, A., Harrison, P., Kelly, P., et al.: Parallel programming using skeleton functions. PARLE'93, Springer (1993) 146–160
14. Botorog, G.H., Kuchen, H.: Efficient high-level parallel programming. Theoretical Computer Science **196**(1-2) (1998) 71–107
15. Michaelson, G., Scaife, N., Bristow, P., King, P.: Nested Algorithmic Skeletons from Higher Order Functions. Parallel Algorithms and Appl. **16** (2001) 181–206
16. Bischof, H., Gorlatch, S., Leshchinskiy, R.: Generic parallel programming using C++ templates and skeletons. LNCS 3016. Springer (2004) 107–126
17. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P^3L: A structured high-level parallel language, and its structured support. Concurrency: Practice and Experience **7**(3) (1995) 225–255

18. Leyton, M., Piquer, J.M.: Skandium: Multi-core programming with algorithmic skeletons. In: 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, IEEE (2010) 289–296
19. Hey, A.J.G.: Experiments in MIMD parallelism. Future Generation Computer Systems **6**(3) (1990) 185–196
20. Foster, I.: Designing and building parallel programs: concepts and tools for parallel software engineering. Addison–Wesley (1995)
21. Grama, A.Y., Kumar, V., Gupta, A., Karypis, G.: Introduction to parallel computing. Addison–Wesley (2003)
22. Priebe, S.: Dynamic task generation and transformation within a nestable workpool skeleton. In: Euro-Par. LNCS 4128 (2006)
23. Dieterle, M., Berthold, J., Loogen, R.: A skeleton for distributed work pools in Eden. In: FLOPS '10. LNCS 6009. Springer (2010) 337–353
24. Dieterle, M.: Parallele funktionale Implementierung von Master-Worker-Skeletten. Diplomarbeit, Philipps-Universität Marburg (2007) In German.
25. Reinders, J.: Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media, Inc. (2007)
26. Lämmel, R.: Google's mapreduce programming model — revisited. Science of Computer Programming **70**(1) (2008) 1–30
27. Dean, J., Ghemawat, S.: MapReduce: a flexible data processing tool. Communications of the ACM **53** (2010) 72–77
28. Friedman, D.P., Wise, D.S.: CONS should not evaluate its arguments. Automata, Languages, and Programming (1976) 257–281
29. Wray, S.C., Fairbairn, J.: Non-strict languages — programming and implementation. Computer Journal **32**(2) (1989) 142–151
30. Stephens, R.: A survey of stream processing. Acta Inform. **34**(7) (1997) 491–541
31. Conery, J.S., Kibler, D.F.: Parallel interpretation of logic programs. FPCA '81, ACM (1981) 163–170
32. Cohen, H.: A Course in Computational Algebraic Number Theory. Springer (2000)
33. Miller, G.L.: Riemann's hypothesis and tests for primality. Journal of Computer and System Sciences **13**(3) (1976) 300–317
34. Rabin, M.O.: Probabilistic algorithm for testing primality. Journal of Number Theory **12**(1) (1980) 128–138
35. Adleman, L.M., Pomerance, C., Rumely, R.S.: On distinguishing prime numbers from composite numbers. Annals of Mathematics **117**(1) (1983) 173–206
36. Cohen, H., Lenstra, Jr, H.W.: Primality testing and Jacobi sums. Mathematics of Computation **42**(165) (1984) 297–330
37. Berthold, J., Loogen, R.: Visualizing Parallel Functional Program Executions: Case Studies with the Eden Trace Viewer. In: ParCo '07. Parallel Computing: Architectures, Algorithms and Applications, IOS Press (2007)
38. Welch, P.H.: Graceful termination–graceful resetting. In Bakker, A.W.P., ed.: Applying Transputer-Based Parallel Machines, IOS Press (1989) 310–317
39. Brown, N.C.C.: Communicating Haskell processes: Composable explicit concurrency using monads. Communicating Process Architectures (2008) 67–84
40. Hahnel, T.: The Rabin–Miller Prime Number Test on Systola 1024 on the Background of Cryptography. Master's thesis, University of Karlsruhe (1998)
41. Cheung, R.C.C., Brown, A., Luk, W., Cheung, P.Y.K.: A scalable hardware architecture for prime number validation. FPT '04 (2004) 177–184
42. Schmidt, B., Schimmler, M., Schroeder, H.: High-Speed Cryptography. In: Embedded Cryptographic Hardware: Methodologies & Architectures. Nova Science Publishers (2004)