Comparing and Optimising Parallel Haskell Implementations for Multicore Machines

J. Berthold [ad1] , S. Marlow [b2], K. Hammond [c3], and A. D. Al Zain [d4]
[a]Philipps-Universität Marburg, Germany [b]Microsoft Research Ltd. Cambridge, UK
[c]School of Computer Science, University of St Andrews, UK
[d]School of Mathematics and Computer Sciences, Heriot-Watt University Edinburgh, UK
[1]berthold@informatik.uni-marburg.de, [2]simonmar@microsoft.com,
[3]kh@cs.st-and.ac.uk, [4]a.d.alzain@hw.ac.uk

***Abstract*—In this paper, we investigate the differences and tradeoffs imposed by two parallel Haskell dialects running on multicore machines. GpH and Eden are both constructed using the highly-optimising sequential GHC compiler, and share thread scheduling, and other elements, from a common code base. The GpH implementation investigated here uses a physically-shared heap, which should be well-suited to multicore architectures. In contrast, the Eden implementation adopts an approach that has been designed for use on distributed-memory parallel machines: a system of multiple, independent heaps (one per core), with inter-core communication handled by message-passing rather than through shared heap cells. We report two main results. Firstly, we report on the effect of a number of optimisations that we applied to the shared-memory GpH implementation in order to address some performance issues that were revealed by our testing: for example, we implemented a work-stealing approach to task allocation. Our optimisations improved the performance of the shared-heap GpH implementation by as much as 30% on eight cores. Secondly, the shared heap approach is, rather surprisingly, not superior to a distributed heap implementation: both give similar performance results.**

I. Introduction

Haskell [1] is the dominant non-strict purely functional programming language, with a highly active research community. In the ten years since the release of the Haskell 98 standard, it has come to be very widely used in academia, and is beginning to find real-world uses. There is even (from November 2008) an O'Reilly textbook [2]. Unlike many other general-purpose programming languages, Haskell was originally designed with parallelism in mind. The use of the term "non-strictness" rather than the more common "laziness" reflects this preoccupation, and the first complete Haskell implementation (at Glasgow, in 1989) was actually intended to execute on the GRIP parallel graph-reducing architecture [3]. Since then, several other parallel implementations of Haskell dialects have been produced, including implementations of pH [4], GpH [5] and Eden [6]. Early parallel Haskell implementations targeted novel architectures, including the GRIP novel architecture [3] and the Monsoon dataflow machine [7], as well as conventional shared-memory and distributed-memory machines. Most of the recent efforts have, however, taken advantage of the widespread availability of small- to medium-scale clusters of commodity uni- or dual-processors [8], or aim to exploit large-scale computational Grids, distributed over a wide geographical area [9]. Such systems generally work best with coarse-grained, evenly-partitioned parallel programs, with limited communication volumes. However, the recent hardware focus on multicore architectures for mainstream desktop and laptop computing means that fine-grained communication-intensive parallel computing is becoming increasingly affordable, and attention is therefore turning once more to large shared-memory and even dataflow machines.

This paper considers two modern parallel Haskell implementations for current multicore systems. Both are based on the high-quality sequential Glasgow Haskell Compiler (GHC) [10], [11]. One uses a physically shared heap, explicitly intended for multicore/shared-memory systems [12], to implement the mostly-implicit GpH dialect [13]. The other uses message passing to implement a distributed heap for the more explicit Eden dialect [6]. This distributed heap is then mapped to physically-shared memory on a multicore machine. We are particularly interested in the tradeoffs between the distributed- and shared-heap models. While this paper sheds some light on the factors affecting this choice, showing rather surprisingly that there is little difference in performance between the two models on eight to sixteen cores, we have by no means told the whole story here. We believe that the ongoing push towards greater numbers of cores in shared-memory systems is likely to tip the balance further over the coming years in the direction of distributed heaps.

The main novel contributions of this paper are as follows:

- We compare the performance of shared-heap (GpH) and distributed-heap (Eden) parallel implementations of Haskell on commodity multicore machines (eight and sixteen cores).
- Our work underlines the importance of adequate tools for parallel profiling. In this paper we exploit a custom approach to profiling, that we have used pending official support for profiling in GHC.
- We report on various optimisations for the shared-heap GpH implementation, covering performance issues uncovered by our profiling results.
- We attempt to shed some light on the best way forward for parallel Haskell implementations on multicore machines. Our aim is to provide consistently good performance with a transparent, and easy-to-use programming

model. Our results suggest a number of ways we can move towards that goal.

II. Eden and GpH: Two Parallel Dialects of Haskell

We have based our comparison of parallel Haskell multicore implementations and programming models around two general-purpose parallel Haskell dialects, GpH and Eden. Both dialects allow the programmer to express both data and task parallelism. The main differences between them lie in the way that parallel computation is specified, and in the degree of explicit programmer control that they allow over parallel execution. In this paper, we consider the use of parallel *skeletons* in Eden, and the use of the similar parallel *evaluation strategies* in GpH. In both models, the programmer delegates task creation, allocation and communication to the implementation. The GpH implementation has more freedom to control the precise dynamic behaviour of the program, and the use of evaluation strategies allows more programmability. However, this carries a potential performance cost, especially for more regular parallel problems.

A. Algorithmic Skeletons, Eden

The Eden dialect of Haskell [6] originally targetted distributed computing systems. It provides an explicit process notation in a non-strict setting, and allows communication of fully-evaluated data between processes. In this paper, we adopt a programming model for Eden based on the use of "algorithmic skeletons" [14], [15], that build on the basic Eden `process` construct to provide higher-level parallelism. Algorithmic skeletons are used to give an abstract description of the structure of an algorithm as a higher-order function. Usually, such skeletons are analogous to commonly-used sequential functions such as `map` or `fold`, but are given specific parallel implementations (and often special names). The potential parallelism can be exploited by a hidden parallel implementation, whereas the specification remains purely sequential. There is a long history of research on the use of such skeletons in Eden [16], [17]. We use some simple examples to illustrate this basic idea:

```
parMap ::
   (Trans a,Trans b) =>
   (a->b)  ->  [a] -> [b]

parReduce ::
   (Trans a,Trans b) =>
   (a -> a -> a) -> a -> [a] -> a

parMapReduce ::
   (Trans a,Trans b, Trans c, Trans d) =>
   (c->[(d,a)]) -> (d->[a]->b) -> [c]->[(d,b)]

masterWorker::
   (Trans a,Trans b) =>
   (a -> ([a],b)) -> [a] -> [b]

ring ::
   (Trans i,Trans o, Trans r) =>
   (a -> [i]) -> ([o] -> b) ->
   ((i,[r]) -> (o,[r])) -> a -> b
```

Here, the parallel `parMap` skeleton is semantically equivalent to a standard higher-order `map` function, but differs operationally in that it applies its function argument to each element of the list argument *in parallel*. Similarly, `parReduce` is a parallel variant of a standard `fold(l)` function. In the spirit of Google MapReduce [18], `parMapReduce` first applies a parallel `map`, producing a list of key-value pairs from every input item, and then reduces each set of values for one key across these intermediate results. `masterWorker` (in the shown version) describes a group of worker processes that collectively process a large, and dynamically changing, set of irregularly-sized tasks, under the control of a coordinating master process. It can implement a parallel map, backtracking, and branch-and-bound (when extended by a master state) [19]. Finally, the `ring` skeleton illustrates a different class of skeletons. Rather than capturing algorithmic structure, *topology skeletons*, explicitly represent the parallel interaction between processes in a regular topology [20], [21], [22].

1) Implementing Skeletons in Eden.: Eden allows *process abstractions* to be defined using the `process` function. Process abstractions created in this way can then be *instantiated* (i.e. executed) on remote processors, using the `#` function or more basic internals (IO-monadic [23], [22] and providing more explicit control).

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

Eden Processes are encapsulated units of computation, assumed to have their own private heap and to communicate inputs and results solely via *channels*. Data types communicated over them has to be instance of the `Trans` class. All values are reduced to *normal form* prior to sending, which implies that processes (and any skeletons built from them) may impose additional strictness requirements. Overloading in `Trans` implements Eden's communication semantics: If the input or output of a process is a tuple, each component of the tuple is evaluated by its own independent thread and communicated independently. Top-level lists are evaluated and communicated element-by-element. Values of any other type (whether structured or not) will be communicated in single messages.

For example, we can implement `parReduce` in Eden as:

```
parReduce f ntr list = foldl' f ntr rs where
     rs = spawn (repeat (process (foldl' f ntr))) ls
     ls = splitIntoN noPE list
spawn:: (Trans a, Trans b) => [Process a b]->[a]->[b]
 -- kicks off processes with their inputs
spawn ps inputs = unsafePerformIO (zipWithM
    [ instantiateAt 0 p i | (p,i) <- zip ps inputs ])
```

Although this particular implementation is not technically difficult, it does require deeper understanding of the underlying Eden parallel implementation, and is thus best seen as a systems programming task. The point, however, is that unlike the encapsulated libraries we commonly find for imperative languages (e.g. [24], [25], [26]), Eden skeleton implementations are still amenable to customisation.

B. Evaluation Strategies, Glasgow Parallel Haskell (GpH)

Semi-explicit parallelism: Glasgow parallel Haskell (GpH, [5]) has been studied since the late 1980s [3]. GpH provides semi-explicit data and task parallelism, based on a single parallel coordination construct, `par`:

```
par :: a -> b -> b
```

The `par` function returns the value of its second argument, having recorded the first argument as a "spark", i.e. a closure that could subsequently be evaluated in parallel if there are available processor resources. The implementation of GpH uses a dynamic work-stealing approach, where idle processors steal sparks from busy ones. These (extremely lightweight) sparks may then be turned into threads for execution on the stealing processor. Haskell98's `seq` operator can be used to provide similar sequential control, forcing evaluation of its first argument before returning the second argument as its result.

```
seq :: a -> b -> b
```

Evaluation Strategies: By using normal higher-order functional programming, higher-level parallel programming constructs can be defined just from these two simple primitive constructs. These "evaluation strategies" [27] can then be applied to normal program expressions. In this way, the functional specification (the expression) is separated from the specification of parallel behaviour (the evaluation strategy). Since strategies are normal higher-order functions, they can be easily combined into more complex forms, and end-users can easily define tailor-made strategies, building on existing standard strategies, where appropriate.

It is easy to define constructs that are similar to algorithmic skeletons. For example, we can define a strategy that evaluates the elements of a list in parallel:

```
parList :: Strategy a -> Strategy [a]
parList s [] = ()
parList s (x:xs) = s x `par` parList s xs
```

Here, the first argument to `parList` is a strategy that is used on the elements of the list, so a strategy that evaluates the outer two levels of a list in parallel might be defined as:

```
parList2 = parList (parList rwhnf)
```

The main advantage of the GpH approach is that it can dynamically adapt the parallel computation to the state and load of nodes in the parallel system. In previous work, we have found that it is possible to obtain good parallel utilisation and performance, especially for large numbers of fine-grained threads on low-latency machines (SMPs and multicores). Moreover, for coarser granularities, spark and thread migration limits the problems that can be caused by uneven explicit work allocation on higher latency machines. Subject to obtaining good data locality, this can give advantages over explicit placement and control in terms of improved workload distribution and performance.

III. Eden and GpH Implementations

A. Shared Memory Implementation of GpH in GHC

From 2004, GHC has supported a shared-memory implementation of GpH [12]. The GHC runtime system supports both GpH and Concurrent Haskell, and can run programs that use both models. The runtime system implements Concurrent Haskell threads (hereafter just "Haskell threads") using a system of lightweight threads multiplexed onto a small number of heavyweight OS threads in order to achieve real parallelism on a multiprocessor, while still keeping the overheads of concurrency low. The parallel runtime system is built around the notion of a *capability*. A capabilty represents the resources for running a Haskell computation. The number of capabilities in the system is fixed at program startup time, and is typically chosen to be equal to the number of physical processor cores on the machine. The number of capabilities equates to the number of Haskell threads that can be running simultaneously at any one time. GHC's capabilities correspond precisely to the Eden and GUM Processing Elements (PEs) described below, and every capability manages its own local pool of Haskell threads and GpH sparks. It is the resposibility of the scheduler to allocate Haskell threads to capabilities. Sparks and threads, all sharing the global heap, are moved between capabilities at runtime, determined by an appropriate scheduling policy and runtime parameters.

B. Distributed-Memory Implementations

Eden and GpH have been implemented for clusters as extensions to the Glasgow Haskell compiler, GHC [10], [11] and historically share parts of their infrastructure. It comprises a few changes to the front-end, but major modifications to the runtime environment [23]. When run in parallel, each PE runs a sequential copy of the GHC runtime. Multiple PEs are linked into a parallel system by using a message-passing communication system, which has been designed to allow plug-in replacement of different message-passing libraries. Typically, it uses either the standard PVM or MPI libraries (for which shared-memory implementations exist), but specialist implementations have also been constructed. And, in the context of this paper, we can use such middleware or specialised implementations on shared memory systems, to use shared memory instead of the network for communication behind the scenes! So it is perfectly possible to run the distributed-memory implementations on a multicore.

Eden implementations require an explicit remote task creation mechanism and channel communication mechanisms between the PEs. Both are exposed to the Haskell level via primitive operations. The proper constructs of Eden are implemented as a Haskell module on top of these more basic primitives [23]. To synchronise communication between the PEs, placeholders in the heap are used, which will be replaced by arriving message data (computation subgraph structures, serialised into one or more packets for transmission). Especially the data transfer support in distributed-heap implementations (often common code shared with Eden and GUM implementations in the past), is the essential obstacle for the maintenance of systems for distributed memory. GUM implementations of GpH add passive work distribution mechanisms (thus requiring support for spark management and respective protocols), virtual shared memory by global addressing, and weighted reference counting for global garbage collection. The latter are, however, well-understood general concepts [28].

IV. Optimisations and Runtime Analyses

A. Optimising shared-memory GpH in GHC

During our analyses of the benchmarks we discovered some performance bottlenecks and areas for improvement in GHC's shared-memory parallelism implementation. In the following sections we detail each of the problems we discovered, and outline solutions (where we have them).

1) Improving the GC barrier.: Versions of GHC up to 6.8.x used a stop-the-world sequential garbage collection (GC). Although a parallel collector has recently been implemented [29], it is still stop-the-world, and this causes a performance bottleneck. The default GC parameters set the size of the allocation area to 0.5MB. Each capability has its own allocation area, and whenever an area becomes full up, all capabilities must stop in order to GC. The cost of this barrier can be high: since GC checks happen only when a context-switch occurs, and Haskell threads only check to see whether a context-switch is required once they have allocated a certain amount of memory (currently 4k), threads that are allocating slowly will context-switch infrequently, and the GC barrier will therefore be delayed. The adverse effect of the barrier can be seen in our performance measurements in Section V, where we compare the original against an optimised barrier synchronisation in the GHC runtime, which shows slightly better performance.

However, there is need for a more substantial modification: simply reducing the frequency of young-generation collections by increasing the size of the allocation areas had a massive effect on runtime and core utilisation. One solution to the GC synchronisation problem would be to implement a concurrent garbage collector. Typically, however, concurrent GC adds some overhead to the mutator, since it must synchronise with the collector. An alternative, and effective, GC strategy is to allow capabilities to collect their own heaps independently [30], while managing shared global heap that is collected much less frequently. We plan to implement some form of this approach in the future.

2) Work-stealing for sparks.: Each capability maintains a local pool of sparks, where sparks are either created using the `par` combinator or by evaluation strategies. If a capability has no threads to run, but a non-empty spark pool, it can choose a spark from the pool, turn it into a thread and evaluate it. This on its own does not create parallelism; a work-offloading mechanism is also required. The simple work-balancing scheme implemented in GHC 6.8.x was to *push* surplus work (threads and sparks) to idle capabilities by polling in the scheduler. This scheme has a severe drawback: since load-balancing only happens when the scheduler is running, there might be a significant delay between the work being created and it being made available for execution on an idle capability. We have adopted a traditional solution to this problem: the spark pool is implemented using a lock-free work-stealing queue [31], and idle capabilities can steal sparks from the spark pools of other capabilities. This scheme of work *pulling* is precisely the original GpH idea, eliminates any hand-shaking when sharing work, and enables much more effective dynamic load-balancing. Using this optimisation, we were able to obtain substantial performance improvements. In our work, the work pulling scheme is applied to sparks, surplus threads are still pushed actively to other capabilities. Work pulling could also be applied to threads, which might have a further positive impact on performance in some cases.

3) Work duplication and black-holing.: The GHC runtime system implements an abstract graph reduction engine, which lazily reduces the computation graph of a functional program, i.e. enters every graph node for evaluation at most once. This technique nicely implements work sharing by shared graph nodes. Furthermore, it enables a low-effort concurrency synchronisation: thanks to referential transparency, it is completely safe to evaluate the same expression in multiple threads. However, in the case of a parallel program, this will waste resources that could be used to do useful work. The runtime system therefore uses synchronisation nodes ("black holes") in the graph, to avoid duplicate evaluation and suspend the second one of the evaluating duplicate threads. While such duplicate work is typically uncommon in concurrent programs, the domain of parallel programming knows examples that do trigger duplicate evaluation to a large degree (using shared global data in embarassingly parallel problems).

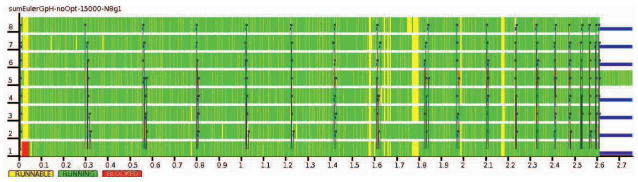
That said, an optimisation of the sequential GHC system turns out to be a disadvantage for parallel programs: GHC currently does "lazy black-holing"; that is, it only marks a thunk as being under evaluation at context-switch time, rather than immediately. In the sequential setting, this can mean that some thunks are never "black-holed", so giving a potential performance win. Unfortunately, in the parallel setting, it substantially enlarges the time window in which two or more duplicate threads might evaluate the same think, and thus waste considerable time. One solution to this is to revert to the earlier "eager black-holing" approach: where each thunk is marked immediately as soon as it is entered. In this case, a second or third thread that attempts to evaluate a thunk that is already under evaluation will be suspended until the first thread to enter the thunk completes. Surprisingly, our preliminary measurements suggest that, on current processor architectures, this carries little performance disadvantage over lazy black-holing.

4) Using one thread to run sparks.: When a spark is activated, this is currently done by creating a new thread in which to evaluate the sparked closure. While threads are cheap in GHC, there is still a certain amount of overhead associated with this thread creation and its subsequent destruction, and the context-switch between one spark being completed and the next being activated. The solution used by Harris and Singh [32] is to dedicate one thread on each capability to just running sparks. Our solution is similar, but simpler, and involves less book-keeping: when there are sparks to run, the scheduler creates a *spark thread* whose job it is to continuously evaluate sparks until there are no more sparks available (on any Capability), and then exit. This strategy avoids problems that

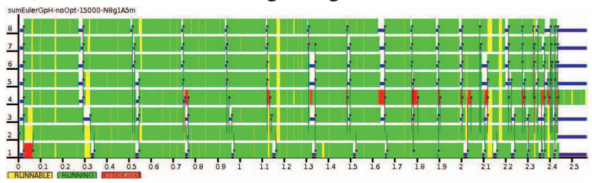
Program version and runtime system	Runtime
GpH in plain GHC-6.9	2.75 *sec.*
GpH in plain GHC-6.9, big allocation area	2.58 *sec.*
GpH, above + improved GC synchronisation	2.44 *sec.*
GpH, above + work stealing for sparks	2.3 *sec.*
Eden-6.8.3, 8 PEs running under PVM	2.24 *sec.*

Fig. 1. Parallel runtimes of the sumEuler program for `[1..15000]`

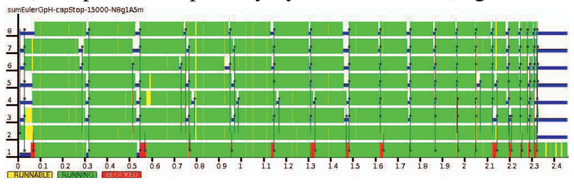
a) Unmodified GHC-6.10



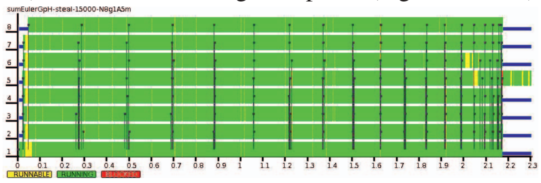
b) same as above, using a big allocation area



c) improved Capability synchronisation (big alloc. area)



d) with work stealing for sparks (big alloc. area)



e) Eden version using PVM

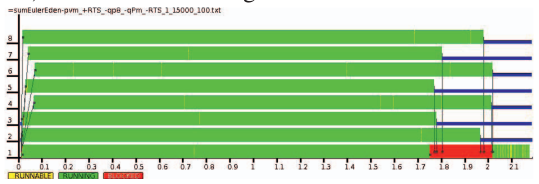


Fig. 2. Runtime traces of `sumEuler [1..15000]`: GpH versions and Eden (Intel 8-core MS Research Cambridge)

arise when a spark blocks (say on a black hole); the scheduler will simply create another spark thread. Spark threads are not special in any way, and the runtime system does not have to track which threads are spark threads. A spark thread can also exit if it finds that there are other threads to run on the current Capability, since spark threads should give up the CPU for higher priority threads (for simplicity, GHC's scheduler does not provide priorities).

V. Measurements

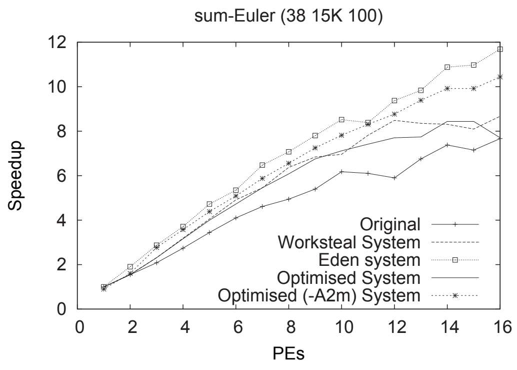
We have measured runtimes, and analysed the runtime behaviour, for some small test programs which represent typical parallelisation problems: transformation and reduction; a regular bag-of-tasks problem; and a genuine parallel algorithm which uses a ring structure. For each application, we compare an Eden program with a comparable GpH program on the same multicore machine. Measurements have been carried out on the following platforms:

- an Intel-based 8-Core machine (2× Intel Xeon Quad-Core @ 1.86GHz) with 16GB RAM, at MS Research Cambridge;
- an AMD-based 16-Core machine (4× AMD Opteron QuadCore @ 2.3GHz) with 132GB RAM, at LMU Munich.

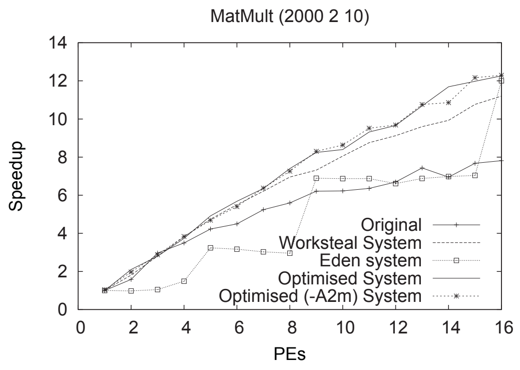
A simple map-reduce operation: Our first test program sums up values of the Euler-function φ for all numbers smaller than a given limit n. The Euler function $\varphi(k)$ (computed naïvely in our test program) calculates how many $j < k$ are relatively prime to k. So we compute $\sum_{k=1}^{n} \varphi(k) = \sum_{k=1}^{n} |\{j < k | (j,k) = 1\}|$, or in Haskell syntax: `sum ( map phi [1..n])` where `phi n = length (filter (relprime n) [1..(n-1)])`. The Eden program can use a ready-made `parMapReduce` skeleton; the GpH program can apply several variants of splitting the input into sublists, and use a `parList rnf` to force the result of applying the above function to each sublist.

The runtime behaviour of this function on 8 nodes can be analysed from the timeline diagrams shown in Fig. 2.[1] All versions of the program check the result using a second sequential computation, that is obvious at the end of each trace. The diagrams show the activity of the 8 capabilities on an 8-core machine over time, indicating whether a Haskell computation is being run (green), whether the capability is runnable, but is waiting for system work or synchronisation (yellow), whether all threads are blocked (red), or whether it is idle (small blue). As Trace a) shows, in the default setting, synchronisation between the capabilities (mainly for garbage collection) seriously affects the runtime. Trace b) shows a run with an enlarged allocation area, which improves the runtime. For trace c), we additionally optimised the synchronisation mechanism in the RTE, to obtain a further slight improvement (there is much more effect without the larger allocation area). Trace d) shows the effect of replacing the spark-pushing mechanism by a (non-synchronising) spark-stealing mechanism. This eliminates all idle periods for the capabilities. Trace

[1]For the purpose of our optimisations, we have instrumented the threaded GHC runtime system and can now use the EdenTV visualisation tool for GpH programs.



Speedups for sumEuler
(problem size 15000)



Speedups for matrix multiplication
(size: 2000×2000 elements)

Fig. 3. Relative speedup for sumEuler and matrix programs (AMD 16-core LMU Munich)

e) shows a comparable Eden program running eight virtual machines (with sub-optimal static load balance) under PVM on the same 8-core. This gives the best runtime of all five options.

Dense matrix multiplication: Matrix multiplication is a classical parallelisation example, applied in various areas of scientific computing, and easily parallelised in different ways (with respective implications for data locality and synchronisation). A GpH parallelisation essentially consists of annotating certain parts of the result matrix for parallel evaluation. In the program we have measured, *regular blocks of the result* are turned into sparks. The block size, i.e. the spark granularity, is tunable by a parameter. The advantage of sparking blocks over a straight-forward row-parallel version is the reduced data-dependence: a block of the result only depends on a subset of both input matrices, whereas each row depends on the whole second input matrix. The Eden implementation equally works with blocks of the result matrix. Implementing Cannon's algorithm[33], the processes communicate in a toroid topology (using a respective skeleton) and exchange respective blocks of the input matrices in sequence with computing subresults. Communication is reduced to a minimum. Our measurements show that both programs yield comparable results and fair speedup. Runtimes and traces are given in Figure 4.

As can be seen from traces a) and b), the unmodified GHC is unable to use the eight cores equally well, and suffers from frequent synchronisation steps for garbage collection (reduced in b) by the large allocation area). Trace c) with work stealing shows the best runtime, and good core usage. To our surprise, the Eden/distributed memory implementation can even profit from using more virtual machines than we had actual cores. We assign this result to improved cache and data separation in the different heaps leading to less frequent garbage collections. Trace d) shows the execution of a 3×3 blockwise computation, on 9 virtual PEs, even better runtime is achieved in trace (e) using a 4×4 division and 17 virtual PEs (on eight physical cores).

Fig. 3 shows speedups for all five versions of these two programs, on another machine, an AMD-based 16-core. Eden runtimes have been measured from a 32-bit binary. In general, 32-bit binaries perform better, and the distributed-heap implementation does not require large address space. For fairness, We give relative speedups.

Parallel shortest paths: Finally, we have performed another set of measurements for a genuinely parallel algorithm: an all-pairs shortest-path computation using a process ring for optimised communication (adapted from [34]). Initialised with a row of the adjacency matrix, each process computes the minimum distances from the corresponding node to every other node by updating its row continuously using the other rows received from, and forwarded to, the ring. These row updates depend on each previous row, but nevertheless can be pipelined to speed up the computation. The equivalent GpH program sparks an evaluation for each row in advance and relies on the runtime system efficiently synchronising concurrent evaluations. Fig. 5 shows relative speedups for the GpH program and different GHC runtime system optimisations, and the Eden program.

The measurements reveal the importance of eager black-holing for parallel evaluation in a shared heap. While the Eden version with clearly structured communication and data separation shows a good speedup, GpH versions of the program cannot profit from additional resources when executed in parallel, unless eager black-holing is used. This effect is most apparent in the workstealing system, where work is distributed among the capabilities efficiently without synchronisation, leading even to a slowdown. In all, the parallel performance of the GpH versions is disappointing and flattens out very soon. Additionally, the runtimes that can be achieved vary significantly due to the non-deterministic scheduling behaviour. An analysis of runtime traces indicates that the bad performance is due to excessive concurrency and/or inappropriate thread-sharing policies, but a more thorough investigation of this effect is necessary. GpH generally does not offer the appropriate level of control for genuinely parallel algorithms with regular process topologies for optimised communication (such as the

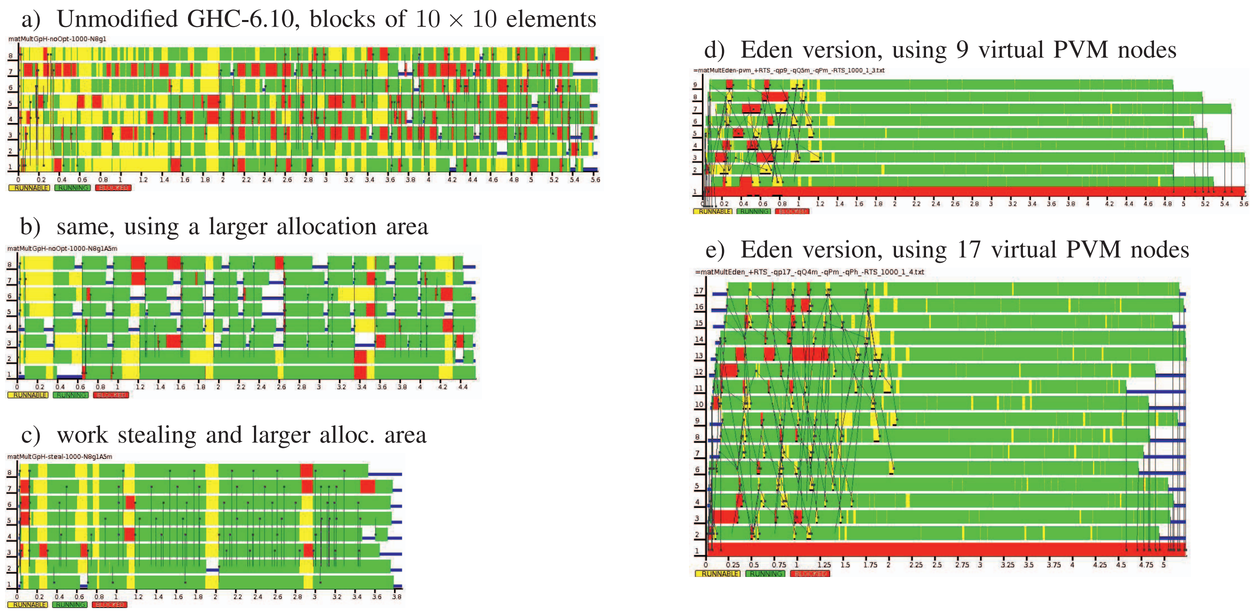
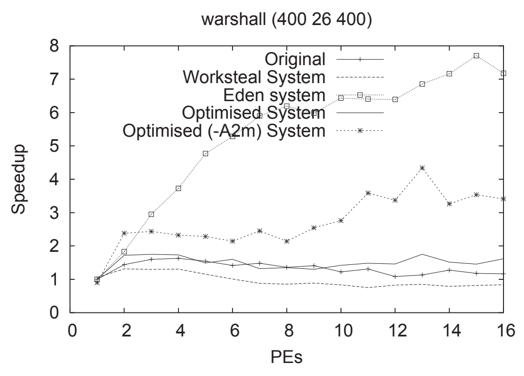


Fig. 4. Traces of matrix multiplication, 1000 × 1000 elements: GpH and Eden (Intel 8-core MS Research Cambridge)



Speedups for shortest-paths-algorithm (400 nodes)

Fig. 5. Relative speedup for shortest-paths program (AMD 16-core LMU Munich)

ring structure used here, that was created using a topology skeleton in the Eden version).

VI. Conclusions and Future Work

A. *Distributed or shared heaps?*

At first glance, it may seem somewhat curious to map a distributed memory programming model with a message-passing implementation onto a multi-core machine, where data may be exchanged between cores simply using shared memory. However, the choice between a distributed or shared-heap model is far from simple. Some of the factors that affect the relative performance of these two models are as follows:

- The shared-heap model has zero communication cost, eliminating a significant class of performance problems at one stroke. The programmer is thus freed from having to think about communication costs, and is more likely to be able to achieve a parallel speedup.
- The shared-heap model implies some synchronisation in order to perform garbage collection, and the cost of this synchronisation becomes more acute as the number of processors increases. The overhead can be reduced by using a semi-distributed heap model: each process has a private heap, with a shared global heap that is collected infrequently [30]. However, this re-introduces a form of communication overhead, because accessing data on another processor requires that data to be moved into the global heap.
- Taking advantage of multiple processors in a shared-heap garbage collector requires a parallel collector, which is a difficult engineering task in itself [29]. In contrast, garbage collection is perfectly scalable in the distributed-heap model, because each processor can collect its own portion of the heap completely independently.
- The distributed-heap model is unlikely to suffer from bottlenecks induced by the system's cache-coherency protocols, because each processor is operating almost exclusively on its own private memory.

So to summarise: the shared-heap model eliminates communication costs, but introduces a number of other overheads. Some of these overheads can be mitigated and engineered around, but it seems unlikely that the shared-heap model will scale to a large number of cores. Future systems with larger numbers of cores are likely to exploit non-uniform memory architectures, with more hierarchical cache designs. In such systems it makes sense to match the heap model to the memory architecture, using per-processor local heaps.

B. *Current Picture and Future Work*

We have compared two implementation approaches for two different parallel Haskell dialects on multicore systems: one approach uses shared-memory directly; the other, less obviously, relies on mapping message-passing to shared memory, and a more explicit programming model. Our results may be somewhat counter-intuitive to those who are not familiar with parallelism: We have achieved good speedups with a distributed-memory Eden implementation, using stock middleware and without special optimisation efforts.

Directed by the results of this comparison, our work has concentrated on optimising the shared-memory implementation of GpH in GHC. We are pleased to have achieved substantially better results for some programs, to have identified some crucial implementation-level optimisations using our custom tracing tool for the analysis of the runtime behaviour of GpH programs. Further optimisation areas for the GpH implementation (and shared-memory parallelism in GHC in general) have been identified, and we anticipate that further investigation of potential overheads and ongoing work on the multicore support in GHC will lead to better results.

We expect, however, that the speedups that can be achieved on eight or 16 cores will not scale when future systems with more cores are used. The solution may be to use structured implementations, either using multiple levels in a more sophisticated shared memory implementation, or combining aspects of our shared memory and message-passing implementations into a single implementation. In order to properly support such a hybrid approach, it may be necessary to develop a multi-level programming model, for example, by specifying new evaluation strategies/skeletons that expose parallelism at multiple hierarchical levels. Clearly, there is significant work to be done, if large-scale multicore architectures are to be exploited effectively. The work described in this paper sheds light on some of the issues that must be addressed by future parallel implementations.

Acknowledgements: We would like to thank Hans-Wolfgang Loidl for the opportunity to run a number of experiments on the 16-core machine at LMU, Munich.

References

[1] Peyton Jones, S., Hughes, J., *et al.*: Haskell 98: A Non-strict, Purely Functional Language (1999) Available at http://www.haskell.org/.

[2] O'Sullivan, B., Stewart, D., Goerzen, J.: Real World Haskell: Code You Can Believe In. O'Reilly (2008) ISBN 10: 0-596-51498-0.

[3] Peyton Jones, S., Clack, C., Salkild, J., Hardie, M.: Grip - a high-performance architecture for parallel graph reduction. In: Intl. Conf. on Functional Programming Languages and Computer Architecture (FPCA'87). LNCS 274, Springer (1987)

[4] Nikhil, R., Arvind, Hicks, J., Aditya, S., Augustsson, L., Maessen, J.W., Zhou, Y.: pH Language Reference Manual. Technical Report CSG Memo 369, Laboratory for Computer Science, MIT (January 1995)

[5] Trinder, P., Hammond, K., Mattson Jr., J., Partridge, A., Peyton Jones, S.: GUM: a Portable Parallel Implementation of Haskell. In: PLDI'96, ACM Press (1996)

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

[7] Traub, K.R., Papadopoulos, G.M., Beckerle, M.J., Hicks, J.E., Young, J.: Overview of the monsoon project. In: Proceedings of the 1991 IEEE International Conference on Computer Design. (1991) 150–155

[8] Trinder, P.W., Loidl, H.W., Pointon, R.F.: Parallel and distributed haskells. J. of Functional Prog. **12**(4, 5) (2002) 469–510

[9] Al Zain, A., Trinder, P., Loidl, H.W., Michaelson, G.: Evaluating a High-Level Parallel Language (GpH) for Computational GRIDs. IEEE Transactions on Parallel and Distributed Systems **19**(2) (2008) 219–233

[10] GHC: The Glasgow Haskell Compiler. http://www.haskell.org/ghc

[11] Peyton Jones, S., Hall, C., Hammond, K., Partain, W., Wadler, P.: The Glasgow Haskell Compiler: a Technical Overview. In: Proc. JFIT (Joint Framework for Information Technology) Technical Conference, Keele, UK (March 1993) 249–257

[12] Harris, T., Marlow, S., Peyton Jones, S.: Haskell on a Shared-Memory Multiprocessor. In Leijen, D., ed.: Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, ACM Press (September 2005)

[13] Roe, P.: Parallel programming using functional languages. Technical Report CSC 91/R3, Department of Computing Science, University of Glasgow (1991)

[14] Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge(MA), USA (1989)

[15] Cole, M.: Algorithmic Skeletons. In Hammond, K., Michaelson, G., eds.: Research Directions in Parallel Functional Programming. Springer-Verlag (1999) 289–304

[16] Klusik, U., Loogen, R., Priebe, S., Rubio, F.: Implementation Skeletons in Eden — Low-Effort Parallel Programming. In: IFL'00 — Intl. Workshop on the Implementation of Functional Languages. LNCS 2011, Aachen, Germany, Springer (September 2000)

[17] Loogen, R., Ortega-Mallén, Y., Peña, R., Priebe, S., Rubio, F.: Parallelism Abstractions in Eden. In: Patterns and Skeletons for Parallel and Distributed Computing. Springer (2003)

[18] Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM **51**(1) (2008) 107–113

[19] Berthold, J., Dieterle, M., Loogen, R., Priebe, S.: Hierarchical Master-Worker Skeletons. In Hudak, P., Warren, D., eds.: PADL'08 — Practical Aspects of Declarative Languages. Springer LNCS 4902, San Francisco, USA (January 2008)

[20] Berthold, J., Loogen, R.: The impact of dynamic channels on functional topology skeletons. In Tiskin, A., Loulergue, F., eds.: HLPP 2005 — 3rd International Workshop on High-level Parallel Programming and Applications, Coventry, UK (2005) Proceedings.

[21] Berthold, J., Loogen, R.: Skeletons for recursively unfolding process topologies. In Joubert, G.R., Nagel, W.E., Peters, F.J., Plata, O.G., Tirado, P., Zapata, E.L., eds.: Proceedings of ParCo 2005. Volume 33 of NIC. (2005)

[22] Berthold, J.: Explicit and implicit parallel functional programming: Concepts and implementation. PhD thesis, Philipps-Universität Marburg, Germany (June 2008) http://archiv.ub.uni-marburg.de/diss/z2008/0547/.

[23] Berthold, J., Loogen, R.: Parallel coordination made explicit in a functional setting. In Horváth, Z., Zsók, V., eds.: 18th Intl. Symposium on the Implementation of Functional Languages (IFL 2006). Springer LNCS 4449, Budapest, Hungary (2007)

[24] Kuchen, H.: The Münster Skeleton Library Muesli. Universität Münster, URL: http://www.wi.uni-muenster.de/PI/forschung/Skeletons/index.php (2007)

[25] Benoit, A.: ESkel — The Edinburgh Skeleton Library. University of Edinburgh, URL: http://homepages.inf.ed.ac.uk/abenoit1/eSkel/ (2007)

[26] Danelutto, M.: The parallel programming library Muskel. Universita di Pisa, http://www.di.unipi.it/\textasciitildemarcod/Muskel/Home.html (2007)

[27] Trinder, P., Hammond, K., Loidl, H.W., Peyton Jones, S.: Algorithm + Strategy = Parallelism. J. of Functional Programming **8**(1) (1998) 23–60

[28] Loidl, H.W.: The virtual shared memory performance of a parallel graph reduce. In: CCGRID, IEEE Computer Society (2002) 311–318

[29] Marlow, S., Harris, T., James, R.P., Peyton Jones, S.L.: Parallel generational-copying garbage collection with a block-structured heap. In Jones, R., Blackburn, S.M., eds.: ISMM, ACM (2008) 11–20

[30] Doligez, D., Leroy, X.: A concurrent, generational garbage collector for a multithreaded implementation of ML. In: POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM (1993) 113–123

[31] Chase, D., Lev, Y.: Dynamic circular work-stealing deque. In Gibbons, P.B., Spirakis, P.G., eds.: SPAA, ACM (2005) 21–28

[32] Harris, T., Singh, S.: Feedback directed implicit parallelism. In Hinze, R., Ramsey, N., eds.: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, (ICFP'07), Freiburg, Germany, ACM (October 2007)

[33] Quinn, M.: Parallel Computing. McGraw-Hill (1994)

[34] Plasmeijer, M., van Eekelen, M.: Functional Programming and Parallel Graph Rewriting. Addison-Wesley, Reading, Massachusetts, USA (1993)