Noname manuscript No.
(will be inserted by the editor)

Comparing Low-Pain and No-Pain Multicore Haskells

M. KH. Aswad · P. W. Trinder · A. D. Al-Zain · G. J. Michaelson · J. Berthold

the date of receipt and acceptance should be inserted later

Abstract Multicore and NUMA architectures are becoming the dominant processor technology and functional languages are theoretically well suited to exploit them. In practice, however, implementing effective high level parallel functional languages is extremely challenging.

This paper is a systematic programming and performance comparison of four parallel Haskell implementations on a common multicore architecture. It provides a detailed analysis of the performance, and contrasts the programming effort that each language requires with the parallel performance delivered. The study uses 15 'typical' programs to compare a 'no pain', i.e. entirely implicit, parallel implementation with three 'low pain', i.e. semi-explicit, language implementations.

We report detailed studies comparing the parallel performance delivered. The comparative performance metric is speedup which normalises against sequential performance. We ground the speedup comparisons by reporting both sequential and parallel runtimes and efficiencies for three of the languages. To measure the programming effort required by each language we record the number of programs improved and the relative and absolute program changes required to coordinate the parallelism.

The results of the study are encouraging and, on occasion, surprising. We find that fully implicit parallelism as implemented in FDIP cannot yet compete with semi-explicit parallel approaches. Semi-explicit parallelism shows encouraging speedup for many of the programs in the test suite. Languages with implementations designed for distributed memory architectures perform surprisingly well given their high message-passing costs. This leads us to speculate that, as the number of cores grow, implementations with some form of independent heap will outperform those with shared heaps.

Aswad, Trinder, Zain, Michaelson
School of Mathematics and Computer Sciences, Heriot-Watt University,
Edinburgh, UK
E-mail: {mka19,P.W.Trinder,A.D.AlZain,G.Michaelson}@hw.ac.uk

Berthold
Datalogisk Institut, University of Copenhagen,
E-mail: berthold@diku.dk

1 Introduction

Physical limits of semiconductor technology and improved manufacturing technologies are driving processor technology towards multi and many cores. This hardware trend has engendered much interest in functional languages, as their statelessness makes them well suited to exploit multi and many cores, and matching interest in the functional community in developing technologies to exploit the new hardware, e.g. [8].

The key advantage of a referentially transparent language is that the implementation has considerable freedom of execution order while preserving program semantics. The benefits of a functional paradigm for parallel evaluation have been recognised for forty years, e.g. [42], and there has been sustained effort to realise this potential. A good survey of the concepts and history of parallel functional programming is available in [17], and a comprehensive survey of parallel Haskells in [41].

Typically a parallel functional program must not only specify the *computation* i.e. a correct and efficient algorithm, it must also specify the *coordination* e.g. how the program is partitioned, how parts of the program are placed on processors, or how they communicate and synchronise. Most parallel functional languages aim to combine high level coordination sublanguages with their high level computation language. A range of high level coordination models have been used including data parallelism e.g. [9, 6], *semi-explicit* models e.g. [26,40], coordination languages e.g. [34], and algorithmic skeletons e.g. [30,20]. The ultimate extreme is to make coordination entirely *implicit*, typically using either profiling as in [19] or parallel iteration as in [16]. The slogan associated with languages with high-level coordination is 'Low Pain Parallelism' and with implicit languages is 'No Pain Parallelism'. The challenge, then, is to produce robust portable language implementations that deliver good parallel performance from such high-level, or vanishing, coordination specifications.

This paper provides a snapshot of multicore functional programming by systematically comparing four parallel Haskell implementations on a common multicore architecture. The comparison contrasts the programming effort required to specify coordination with the parallel performance delivered in each language. The comparison uses 15 programs carefully selected, i.e. without regard for their inherent parallelism, from parts of the nofib benchmark suite [32]. In consequence, the results reflect the multicore performance that might be expected for a 'typical' set of Haskell programs (Section 4).

The parallel Haskells and associated implementations studied are depicted in Figure 1. We compare a 'no pain' approach, Feedback Directed Implicit Parallelism (FDIP) [19], with three 'low pain', i.e. semi-explicit languages [24] (Section 2). The semi-explicit Haskells are Eden [26] and two implementations of Glasgow parallel Haskell (GpH) [40], namely GHC-SMP an optimised shared memory implementation integrated in GHC from version 6.6 onwards [5], and GpH-GUM a message-passing implementation designed for shared and distributed memory architectures [39] (Section 3).

We articulate the experiment design in Section 4. Although the parallel Haskell implementations all share the same optimising Glasgow Haskell Compiler technology (GHC), each uses a different version, and hence the performance comparisons are based on speedups, which normalise against different sequential performance. We establish a baseline for the speedup comparisons by reporting sequential and parallel runtimes and efficiencies for three of the languages (Section 5).

We report detailed parallel performance and programming effort studies focusing on the number of programs improved, speedups delivered, and program changes re-

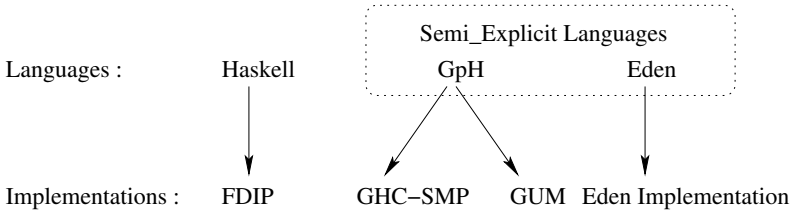


Fig. 1 The Parallel Haskells and Implementations Studied

quired to coordinate parallel evaluation (Section 6). We make a study comparing the scalability, the programming effort required, and the parallel performance achieved, in each language (Section 7). We conclude by summarising the key results and discussing their implications (Section 8).

This paper extends [2] as follows. It presents a more detailed language comparison including coverage of recent parallel Haskells in Section 2. It presents a more detailed language implementations comparison in Section 3. It presents additional sequential and parallel performance results together with a more detailed analysis in Section 5.

2 Parallel Haskell Language Comparison

The parallel Haskells studied are depicted with their implementations in Figure 1. The FDIP implicit approach we consider supports GHC Haskell, but as Haskell is well-known this section focuses on GpH and Eden, the two semi-explicit Haskell extensions that are compared in the remainder of the paper. The languages supported by the GpH implementations have small differences, and we denote them as GpH-GUM and GpH-SMP.

In fact FDIP supports the Concurrent Haskell [35] superset of Haskell, and both Eden and GpH-SMP also support concurrency, i.e. multiple stateful (IO) threads, in addition to the stateless parallel threads. More precisely, GpH-SMP is a superset of Concurrent Haskell. However concurrency is not used in our study and so our language comparison focuses on parallelism only.

FDIP, Eden, and both GpH implementations all rely on sophisticated runtime support. In contrast a number of parallel Haskells have emerged very recently that lift coordination aspects to the language level. These include the `Par` Monad [29], HdpH [27] and Meta-Par [15]. Cloud Haskell [13] is similar in lifting *distributed* coordination to the Haskell level. These languages are developing rapidly, and once the rate of development slows a systematic study comparing the languages with built-in parallelism with those with language-level parallelism would be worthwhile.

We illustrate the coordination extensions by using them to parallelise the Boyer nofib program [32]. This is a Haskell version of the relatively naive "Boyer" theorem proving benchmark. The program rewrites a given input term according to a given set of lemmas in an attempt to produce the value `True` meaning that the original term was a valid theorem. The Boyer benchmark typifies symbolic computations that are a common Haskell application area. Figure 2 shows the key top-level function where, in a simple extension to the original program, the input size `n` is passed as a parameter.

```
test :: Int -> Bool
test n = all test0 (take n (repeat (Var X)))
```

Fig. 2 Haskell Top-level Boyer function

2.1 Glasgow parallel Haskell (GpH)

GpH [40] is a modest extension of Haskell98 with parallel (`par`) and sequential (`pseq`) composition as coordination primitives (see Figure 3). Denotationally, both constructs are projections onto the second argument. Operationally `pseq` causes the first argument to be evaluated before the second and `par` indicates that the first argument may be executed in parallel. The latter operation is called the "sparking" of parallelism and is used in different variants in many parallel languages. The runtime-system, however, is free to ignore any available parallelism. In this model the programmer only has to expose expressions in the program that can usefully be evaluated in parallel. The runtime-system manages the details of the parallel execution such as thread creation, communication etc.

```
par  :: a -> b -> b           -- parallel composition
pseq :: a -> b -> b           -- sequential composition

type Strategy a = a -> ()     -- type of evaluation strategies

using :: a -> Strategy a -> a -- strategy application

rwhnf :: Strategy a           -- reduce to weak head normal form

class NFData a where          -- class of reducible types
  rnf :: Strategy a           -- reduce to normal form

parList :: Strategy a -> Strategy [a]
                        -- Apply a strategy in parallel
                        -- to every element of a list
parList strat []   = ()
parList strat (x:xs) = strat x `par` (parList strat xs)
```

Fig. 3 Coordination in GpH: Evaluation Strategies

Experience of implementing non-trivial programs in GpH shows that the unstructured use of `par` and `seq` operators can lead to rather obscure programs. This problem can be overcome with *evaluation strategies*: lazy, polymorphic, higher-order functions controlling the evaluation degree and the parallelism of an expression. They provide a clean separation between coordination and computation. The driving philosophy behind evaluation strategies is that it should be possible to understand the computation specified by a function without considering its coordination.

Figure 3 shows both basic, and composite original evaluation strategies. A new formulation of evaluation strategies has recently been developed [28], but the programs

here are parallelised using the original strategies. The `using` construct applies a strategy to an expression. The basic strategy `rwhnf` reduces an expression to weak head normal form (WHNF). The overloaded basic strategy `rnf` reduces an expression to normal form (NF), and is instantiated for all major types. As functions, strategies can be combined using the power of the language, e.g. composed or passed as arguments. For example `parList` applies strategy `strat` to every element of a list in parallel.

```
test :: Int -> Int -> Bool
test n m = all (&& True) res
   where xs  = take n (repeat (Var X))
         xs1 = splitAtN m xs
         res = map (all test0) xs1 `using` parList rnf

splitAtN :: Int -> [a] -> [[a]]
splitAtN n [] = []
splitAtN n xs = ys : splitAtN n zs
                where (ys,zs) = splitAt n xs
```

Fig. 4 GpH Top-level Boyer function

Figure 4 shows the GpH parallelisation of the top-level Boyer `test` function, and works as follows. The input list is bound to a variable `xs`, and then split into $\lceil (n/m) \rceil$ chunks and bound to `xs1`. In the programs measured in the remainder of the paper there are 8 chunks. Next the condition (`all test0`) is mapped over the chunks to give a list of intermediate results `res`. It is this mapping that is parallelised (`` `using` parList rnf ``). The final stage is to combine the intermediate results `all (&& True) res`.

The parallelisation illustrates some interesting points. In this program, just 1 of the 52 functions in the 300 line program changes. This is the case for many, but not all, programs. Exceptions include Sphere and Hidden where parallelism is introduced in more than one function. The parallel paradigm is chunked data parallelism. That is, the parallelism is determined by the underlying data structure, and to obtain suitable thread granularity, the program has been changed to aggregate the input. In other programs it is possible to introduce parallelism without changing the algorithmic or computational part of the program, e.g. [23].

2.2 Eden

Eden [26] extends Haskell with syntactic constructs to explicitly define and instantiate processes. In contrast to the other languages, such direct Eden programming exposes parallel tasks at the language level, and requires the programmer to manage them using the control mechanisms provided in the language. In practise however, Eden provides libraries of skeletons [26,3] and many programs, including all of the nofib suite here, can be parallelised using them.

Eden supports a distributed memory parallel paradigm. That is, processes share no values, and communicate only by messages. It might be thought that such a paradigm

```
newtype Process a b = ...

-- process abstraction (language construct)
process :: (Trans a, Trans b) => (a->b) -> Process a b

-- process instantiation
(#) :: (Trans a, Trans b) => Process a b -> a -> b
-- operating on lists of process abstractions and input
spawn :: (Trans a, Trans b) => [Process a b] -> [a] -> [b]

-- non-deterministic merge process
merge :: Process [[a]] [a]
```

Fig. 5 Basic Coordination Constructs in Eden

would not be suitable for parallelism on shared-memory multicore architectures, however recent results have shown good performance [5], as indeed do the results in Sections 5, 6, 7. We return to this issue in Section 8.2.

In direct usage Eden is explicit about process creation and about the communication topology, but implicit about other control issues such as sending and receiving messages, and process placement. Granularity is under the programmer's control because he/she decides which expressions must be evaluated as parallel processes, and also some control of the load balancing is possible at the program level.

Eden provides process abstraction and process instantiation for coordination as shown in Figure 5. The expression `process (\ x -> e)` of type `Process a b` denotes a *process abstraction* constructed from the function `\ x -> e` of type `a -> b`. A process can be *instantiated* using the infix `(#)` operator. Each time an expression `e1 # e2` is evaluated, a new process is created to evaluate the application of the function `e1` to the argument `e2`. Once instantiated, the new process will be executed in parallel with the instantiating caller. We refer to the new process as the child process, and to the instantiating process as the parent.

Parent and child process do not share any data, they communicate by exchanging messages through (implicit) communication channels created on process instantiation. Therefore, the argument and result types of the function embedded in a process abstraction have to be instances of the type class `Trans`. This type class provides functions for data transmission between parent and child process. If the argument or result of a process is a tuple, several channels will be created – one for each element of the tuple.

The instantiation semantics specifies where each expression is evaluated, and the communication pattern: (1) The process abstraction `e1`, together with its whole environment, is copied *in the current evaluation state* to another processor, and the child process is created there to evaluate the application of the function `e1` to `e2`. The child process will receive the argument `e2` from the parent through a channel or several channels if `e2` is a tuple. (2) the argument expression `e2` is eagerly evaluated in the parent process. The resulting normal form data is communicated to the child process through the channel(s) as its input argument. (3) The child process sends the result of the function application to the parent process over the result channel(s).

Once a process has been created, only fully evaluated data objects are communicated. The only exception are lists: they are transmitted in a *stream*-like fashion, i.e.

element by element. Each list element is first evaluated to normal form and then transmitted. Processes trying to access input not yet available are temporarily suspended. This is the only synchronising mechanism in Eden.

To instantiate a whole set of processes on different input, a special `spawn` function is provided, which is denotationally equivalent to `zipWith (#)`, but creates all processes at once, when the computation demands one of their results.

Algorithmic skeletons abstract common patterns of parallel evaluation into higher order functions [10]. They simplify the development of parallel programs by hiding coordination details from the programmer, and may provide ready-made parallel cost models. Eden supports a range of skeletons [26,3], and some of these have been used to parallelise the nofib programs. Appendix A presents and discusses the Eden implementation of a basic master-worker skeleton that is used to parallelise several of the nofib suite, including Boyer.

```
test n m f = all (&& True) res
  where
   xs  = take n (repeat (Var X))
   xs1 = splitAtN  m  xs
   res = parallelMap (all test0) xs1
   parallelMap = mw np pf
   np = noPe
   pf = min 100 maxpf
   maxpf = max 2 (n `div` (m*np*f)))
splitAtN :: Int -> [a] -> [[a]]
splitAtN n [] =... -- see earlier code
```

Fig. 6 Eden Top-level Boyer function

Figure 6 shows the Eden skeleton-based parallelisation of the top-level Boyer `test` function, and works as follows. As before, the input list is chunked into `xs1` and the intermediate results combined by `all (&& True) res` in the final stage. The mapping of (`all test0`) over the chunks is parallelised using a master worker skeleton `mw`, as specified in Appendix A. The master worker skeleton has a master process that manages the parallel evaluation, and a set of worker processes. The number of workers is determined by `np`, the first skeleton parameter. The master distributes tasks, i.e. the function applied to some chunk, to the workers and collects the results. To hide the communications latency between completing a task, and the next task arriving from the master, each worker prefetches a number of tasks, determined by `pf` the second skeleton parameter.

In the Boyer `test` function the number of tasks to prefetch is specified by the f parameter, calculated from the total number of tasks. That is, the number of tasks to prefetch is set to $\frac{1}{f}$th of the average number of tasks per worker, but at least 2 and not more than 100 tasks. The average number of tasks is computed as list length (n) divided by chunk size (m) and number of workers (np), that is $\left\lfloor \frac{n}{m \cdot np \cdot f} \right\rfloor$. As in GpH

Language/ Description	**Haskell (FDIP)**	**GpH**	**Eden**
Classification	Implicit	Semi-explicit	Semi-explicit
Evaluation Order	Normal Order	Normal or Mixed	Mixed
Methodology	FDIP Tools	Evaluation Strategies	Direct or Skeletons
Process Model & Creation	Speculative Threads	Optional Threads	Explicit Processes, Mandatory Creation
Thread Placement	Implicit	Implicit & Dynamic	Implicit & Static
Communication Channels	Implicit	Implicit	Implicit & Explicit

Table 1 Language-level Comparison of Parallel Haskells

the paradigm is chunked data parallelism, and just one out of 52 functions has been parallelised, although this time an algorithmic skeleton is used.

2.3 Language Comparison

Table 1 summarises the language level differences in coordination specification in the three parallel Haskells. Much of the table summarises aspects outlined above. However a key distinction between the languages is that while FDIP preserves normal order evaluation of pure expressions, GpH may not, and Eden does not. GpH preserves normal order evaluation if every evaluation strategy added is no more strict than the embedding function. However it is often useful to be more strict, e.g. speculatively evaluating expressions in the anticipation that they will be used. While Eden processes preserve some normal order evaluation, e.g. of expressions within the body of a process, they are more strict than the corresponding function, e.g. they eagerly evaluate their arguments.

As an entirely implicit implementation, FDIP provides the highest level of coordination abstraction, GpH an intermediate level and Eden the lowest in direct usage. That is, Eden is the most explicit about coordination behaviour, but as we shall see in Section 7, the use of appropriate skeletons can raise the level of abstraction.

3 Parallel Haskell Implementation Comparison

The four parallel Haskell implementations studied are depicted in Figure 1. All of the parallel Haskells support high-level coordination, and rely on sophisticated implementations to effectively manage a vast array of low-level coordination issues typically including task placement, communication, synchronisation, and storage management. All four implementations perform parallel graph reduction [36]. No simple models have ever been constructed of such systems, and their performance is often extremely hard

to analyse. Indeed this is why profiling tools are an essential aid to understand parallel behaviour when tuning the parallel performance of programs written in this class of language.

3.1 Feedback Directed Implicit Parallelism (FDIP)

FDIP supports the full Concurrent Haskell language compiled with traditional optimisations and including I/O operations and synchronisation as well as pure computation. Parallelism is introduced and controlled in FDIP in a four stage process [19] as follows.

Firstly an example execution of the program is profiled. Secondly the profile trace is analysed as a dependency graph of computations to identify useful sources of parallelism. Given the large number of potential computations, or *thunks*, in almost any Haskell program, the challenge is to identify thunks that are simultaneously independent of other thunks, demanded by the program, and with large thread granularity. The third stage is to recompile the program to automatically introduce parallelism at the identified program sites. Finally sophisticated mechanisms are introduced into the runtime system to manage the threads introduced at these sites. These include treating the parallel threads as speculative, and managing load with work stealing.

A simulated limit study shows the potential of FDIP to produce substantial amounts of parallelism for many programs, e.g. utilising at least 8 cores for 40%, and at least 2 cores for 80%, of the 20 nofib programs studied. However the multicore performance is disappointing with only 5 programs out of 20 delivering a speedup of more than 10% [19].

3.2 GHC-SMP

Since 2004 the Glasgow Haskell Compiler (GHC) has supported a shared-memory implementation of GpH, and we term this implementation GHC-SMP. The shared memory implementation is evolving rapidly, and the precise version we describe here and measure in later Sections is based on GHC 6.10.1. The GHC runtime system implements Concurrent 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 overheads of concurrency low. The parallel runtime system is built around the notion of `capability`. A capability represents the resources for running a Haskell computation. 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. It is the responsibility of the scheduler to allocate Haskell threads to capabilities, and Haskell threads may migrate between capabilities at runtime depending on the scheduling policy and runtime parameters. Although the GHC 6.10.1 implementation measured here distributes work eagerly, later unreleased versions gain improved performance by adopting a lazy work stealing approach [5].

The capability holds all of the private state that a worker needs to execute Haskell code. The capability has its own allocation area so allocation proceeds without expensive per-object synchronisation [18]. GHC 6.10 supports both parallel and sequential garbage collection, and the measurements in the following sections use the former. In

this scheme, when memory is exhausted all cores cease reduction and perform garbage collection in parallel.

GHC is under development and published performance results are sparse. Performance results comparing adapted versions of GHC 6.10.1 on an 8-core machine are reported in [5], together with the identification of a number of areas for improvement.

3.3 GUM

Graph-reduction on a Unified Machine-model (GUM) is a portable, parallel runtime environment for GpH [39]. As the name suggests GUM is designed for both shared and distributed memory architectures. It implements a Distributed Shared Memory (DSM) [12] model of parallel graph reduction on a distributed, but virtually shared, graph. Graph segments are communicated in a message passing architecture, using standard communication libraries like PVM [37] or MPI [31], to provide an architecture neutral and portable runtime environment.

The unit of computation in GUM is a lightweight thread, and each logical PE is an operating system process that co-schedules multiple lightweight threads. Threads are automatically synchronised using the graph structure, and each PE maintains a pool of runnable threads. Parallelism is initiated by the *par* combinator: when an expression `x 'par' e` is evaluated, the heap object referred to by the variable `x` is *sparked*, and then `e` is evaluated. By design, sparking a reducible expression (a *thunk*) is a relatively cheap operation, and sparks may freely be discarded if they become too numerous. If a PE is idle, a spark may be converted to a thread and executed. Threads are more heavyweight than sparks as they must record the current execution state.

GUM uses dynamic, decentralised, and blind load management. The load distribution mechanism is designed for homogeneous architectures with uniform PE speed and communication latency, and works as follows. If (and only if) a PE has no runnable threads, it creates a thread to execute from a spark in its spark pool, if there is one. If there are no local sparks, then the PE sends a FISH message to a PE chosen at random seeking to steal work. If the PE that receives a FISH has a useful spark, it sends a message to the PE that originated the FISH containing the sparked thunk packaged with nearby graph. The originating PE unpacks the graph, and adds the newly-acquired thunk to its local spark pool. To maintain the shared virtual graph, a message is then sent to record a reference to the new location of the thunk.

GUM delivers good performance for a range of benchmark and real applications on a variety of parallel architectures, including conventional shared and distributed-memory architectures [23]. The results reported in Sections 5, 6.3, and 7 extend earlier shared memory results with *the first multicore* performance results for GUM. GUM's performance is also comparable with other mature parallel functional languages and with conventional parallel paradigms [22].

3.4 Eden Implementation

Similar to the GUM implementation of GpH, the Eden implementation uses coordinated execution of multiple GHC runtime system instances and message passing internally. No actual changes are made to the compiler front-end, but the runtime environment is extended substantially [4]. Each PE in a GUM or Eden system runs a

sequential copy of the GHC runtime system. Multiple PEs communicate by message-passing, and the communication layer has been designed to allow plug-in replacement of different message-passing libraries, including PVM, MPI, or a custom implementation for shared memory. This paper considers an Eden implementation based on GHC 6.8 using PVM as a message passing library. A native shared-memory implementation is available from version 7.4, but not considered here.

Eden implements explicit remote task creation mechanism and channel-based communication mechanisms between PEs. Both are exposed to the Haskell level via primitive operations. Eden language-level constructs are implemented as a Haskell module on top of these more basic primitives [4]. To synchronise communication between the PEs, placeholders in the heap are used, which will be replaced by arriving message data, i.e. computation subgraph structures, serialised into one or more packets for transmission. This is an important difference between the Eden implementation and both GHC-SMP and FDIP: Eden processes construct and maintain *completely independent sub-heaps*, whereas GUM maintains a virtual shared graph by maintaining references between distributed heaps and reference counting for garbage collection.

Eden's distributed memory parallel performance is widely reported and shows excellent runtimes, speedup and scaleup, e.g. [21, 33, 25]. Eden's distributed memory performance is also comparable with other mature parallel functional languages, and the explicit process model gives performance advantages for applications with coarse thread granularity [22]. The logical separation of heaps between Eden computation units (processes) pays off on multicore platforms, showing competitive performance e.g. for coordinating large symbolic computations [1], as reported in a direct comparison with an optimised multi-threaded GHC 6.8 [5].

3.5 Implementation Comparison

Table 2 summarises the implementation level differences between the four parallel Haskells. All four implementations perform parallel graph reduction. While an arbitrary number of Eden processes can be dynamically created, each process is mandatory. In contrast the other implementations support dynamic techniques including thread subsumption, sparking, and the creation of optional or speculative threads. Eden also uses eager work distribution: newly created processes are pushed out to available PEs, while the other implementations are lazy and idle PEs steal work, i.e. thunks. FDIP and GpH-GUM are both careful not to duplicate work by evaluating the same thunk more than once, but work may be duplicated in GHC-SMP or Eden.

A key distinction between the implementations is the heap model: while FDIP and GHC-SMP have shared heaps, GUM maintains a virtual shared heap, and Eden supports distributed independent heaps, the latter two supported by message passing. Message passing is essential for distributed systems but initially seems enormously expensive compared with shared memory access. That is, subgraphs transmitted between heaps must be must be serialised into, and deserialised from, messages, and computationally expensive message-passing libraries invoked.

However the independent heaps maintained by GUM and Eden convey four significant advantages for shared-memory systems like multicores. Firstly, while the cores in shared heap implementations like FDIP and GHC-SMP must synchronise to garbage

Description	FDIP	GHC-SMP	GUM	Eden Impl.
GHC Version	GHC 6.6	GHC 6.10	GHC 4.06	GHC 6.8
Evaluation Model	Parallel Graph Reduction	Parallel Graph Reduction	Parallel Graph Reduction	Parallel Graph Reduction
Granularity Control	Dynamic	Dynamic	Dynamic	Static
Synch. Unit	Thunk Locking	Thunk Locking	Thunk Locking	Channel Locking
Work Distribution	Work Stealing	Work Pushing	Work Stealing	Dynamic Process Placement
Work Duplication	Not Possible	Possible	Not Possible	Possible
Heap	Shared Heap	Shared Heap	Virtual Shared Heap	Distr. Heap
GC	Synchronised & Sequential	Synchronised & Sequential/ Parallel	Independent & Parallel	Independent & Parallel

Table 2 Implementation-level Comparison of Parallel Haskells

collect, GUM and Eden cores can collect independently and hence in parallel [1]. Secondly, synchronisation is confined to limited shared memory areas, essentially the communication buffers. Thirdly, synchronisation granularity is often large, i.e. on large messages, rather than on individual thunks or memory locations. Finally cache coherency issues are reduced as tasks do not share caches [1]. We discuss the performance implications of the heap designs further in Section 8.2.

Although both FDIP and GHC-SMP use dependent stop-the-world GC, such a design is not inherent. An implementation that maintains some form of thread-private heap, e.g. [11], would enable independent garbage collection and offer many of the advantages outlined above, without incurring the high communication costs of message passing. Indeed we argue that some form of independent heaps will be essential as multicores evolve towards many cores.

4 Experiment Design

We compare the performance of the four parallel Haskells using the 15 programs from the 'real' and 'spectral' sections of the nofib benchmark suite [32] listed in the summary Table 10. The 'real' and 'spectral' sections of the nofib suite are carefully designed to be representative of small Haskell programs, i.e. around 300 source lines of code. The programs are a substantial subset of the 20 multicore benchmarks used in [19] that are in turn carefully selected to be representative. Of the five programs not measured, two are not nofib benchmarks, and three (`cacheprof`, `calendar` and `fibheaps`) are too

[1] Indeed, even the collection of references between GUM heaps is asynchronous.

small to benefit from parallel execution, i.e. where the input cannot be sized to give a runtime of 3s or more on current hardware. Crucially, other than to exclude short programs, the programs are not selected *a priori* for having obvious parallel structure. Hence our results reflect the multicore performance that might be expected for a set of 'typical' small Haskell programs.

To parallelise the programs in Eden and GpH the programs were first time and space profiled to identify computationally expensive functions, and these were parallelised. A variety of parallelisations were investigated for each program and the best selected for the given input, and hence input size. The same GpH program is evaluated under GpH-SMP and GpH-GUM, and the Eden program introduces an appropriate skeleton. Example GpH and Eden parallelisations of the Boyer benchmark are discussed in 2.

All programs are measured on the same input, and with the same heap size. We follow the common practice of increasing input size in many cases to match improvements in processor technology since the benchmarks were established in 1992. The best parallel performance is reported for each system. For Eden, GpH-SMP and GpH-GUM the best performance is obtained on 8 cores, but for FDIP it is obtained on 4 cores as discussed in Section 7.2.

Parallel executions vary due to factors like non-deterministic scheduling, for example in the presence of scheduling accidents the following runtimes (seconds) may be observed: 35.5, 36.0, 72.8, 34.5, 38.7. With such noisy data many believe that the median of 36.0 better represents the sample than the mean of 43.5. We follow this common practice and report measurements that are the median from three executions. To summarise relative metrics like speedup and % program changes we report geometric means, also following common practice.

The parallel implementations are all based on the GHC compiler, but use different versions of it. The FDIP approach uses GHC 6.6, GpH-SMP uses GHC 6.10.1, GpH-GUM uses GHC 4.06, and Eden uses GHC 6.8. As a research platform GHC evolves, and typically the sequential execution time of programs is improved by later versions of the compiler. To address the issue of varying sequential performance, the primary comparative measure is the speedup relative to the corresponding optimised sequential GHC compiler, e.g. GpH-SMP speedups are relative to GHC 6.10.1. This measure substantially normalises against sequential performance and is grounded by runtime measurements in Section 5.

The programs are all measured on common multicore architectures, namely eight core machines comprising two quad-cores. The GpH-SMP, GpH and Eden measurements are for Intel Xeon 5410 cores running at 2.33GHz, with a 1998 MHz front-side bus 6144 KB and 8GB RAM running under Linux Fedora 7. The FDIP measurements are for Intel Xeon X5350 running at 2.66GHz with 4GB RAM running under Windows Server 2003 R2 x64 service pack 2.

5 Runtime Comparison

As the parallel implementations use different versions of the GHC compiler (Section 4), this section provides a baseline for the speedup measurements in the following sections by comparing the runtimes and efficiencies of the GpH-SMP, GpH-GUM and Eden parallel implementations on 1, 2, 3, 4, 6, and 8 cores. FDIP is excluded from these measurement as an implementation is not publicly available.

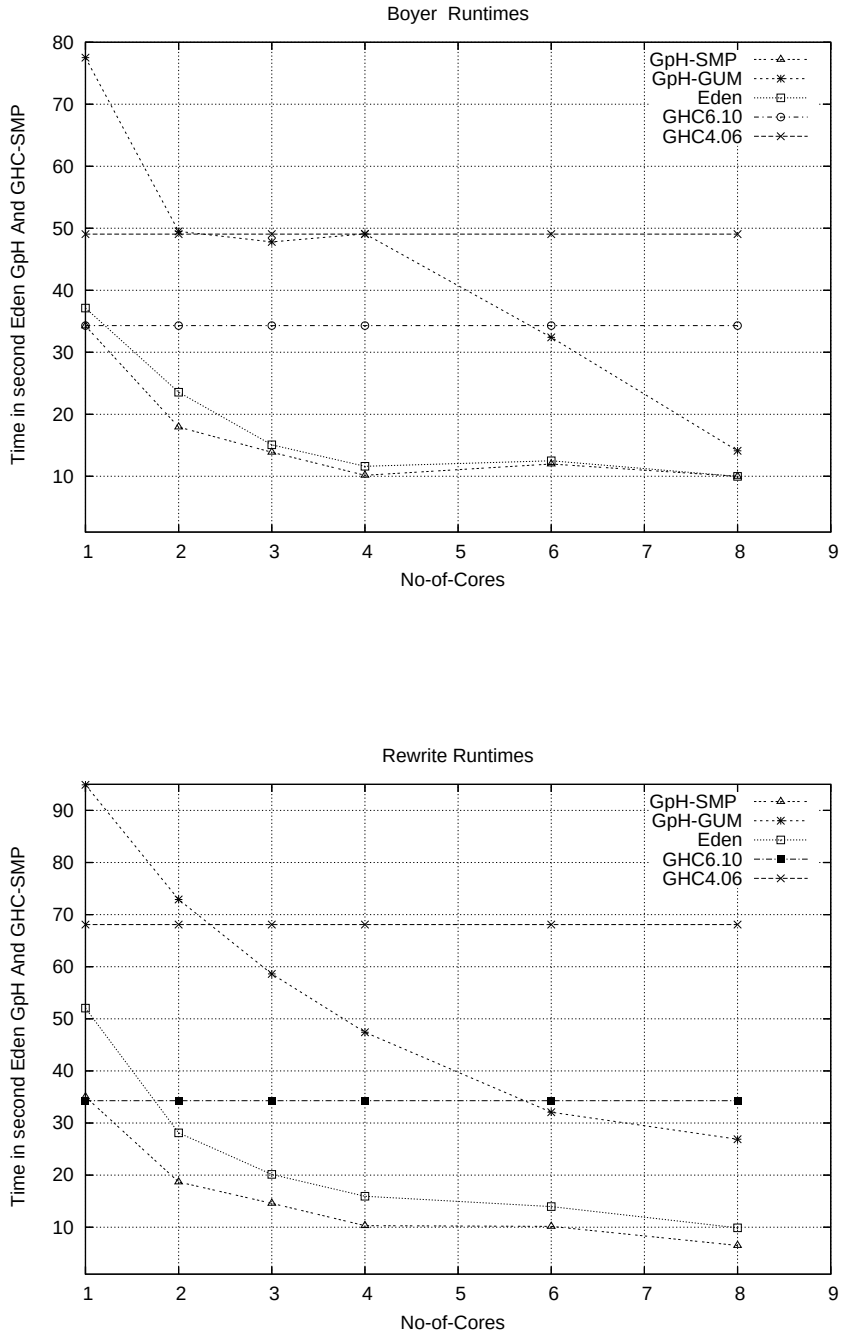


Fig. 7 Runtime Comparison of Parallel Haskells (Boyer/Rewrite)

Program	Sequential			1 Core		
	GHC 6.10	**GHC 4.06**	**GHC 6.8**	**GpH SMP**	**GpH GUM**	**Eden**
Boyer	34.3	49.3	36.7	34.1	77.52	37.1
Clausify	31.1	51.2	29.1	30.4	78.7	29.3
Fft2	35.8	75.7	48.6	35.9	80.9	49.2
Rewrite	34.3	68.1	46.8	35.1	94.9	52.05
Geometric Mean	**33.8**	**60.1**	**39.5**	**33.8**	**82.7**	**40.9**

Table 3 Sequential Runtime Comparison (seconds).

Table 3 summarises the single core runtimes of 4 nofib programs that deliver good speedup. Complete comparative results for all programs in all Haskells will be presented and discussed later, e.g. in Figure 10 and Table 10. To facilitate comparison, the inputs for the programs are sized to give sequential GHC 6.10 runtimes of approximately 35s. Columns 2–4 of the table report optimised sequential runtimes for the compiler instances extended by the parallel Haskell implementations, and these form the basis for the absolute speedup calculations in the remainder of the paper. The remaining columns of the table report the 1 core parallel runtimes for each implementation. We make the following observations.

- The sequential runtimes vary by as much as a factor of 2.1: Fft2 under GHC 6.10 takes 35.8s, and under GHC 4.06 takes 75.7s, but typical variation is less.
- The mean sequential runtimes show that GHC 4.06 is the slowest on a single core, and 1.8 (60.1/33.8) times slower than GHC 6.10. This reflects recent GHC performance improvements. GHC 6.8 is 17% (39.5/33.8) slower than GHC 6.10. Longer runtimes for GHC 4.06 and GHC 6.8 give GpH-GUM, and to a lesser extent Eden, an advantage in the following speedup measurements as the compute time is relatively large compared with communication time.
- It is generally anticipated that a parallel language implementation will introduce some sequential overhead compared with optimised sequential execution. The overhead is termed *sequential efficiency* and represents the additional costs of parallel execution, e.g. launching a single virtual PE and synchronising on closures. The efficiency is a function of both the parallel program and the architecture, and is typically around 80% [39].
  Comparing columns 3 and 6, and columns 4 and 7, of Table 3 shows that this expectation is met for GpH-GUM and Eden, with mean sequential efficiencies of 72% (60.1/83.0) and 97% (39.5/40.9) respectively. Surprisingly, comparing columns 2 and 5 shows that although GpH-SMP is slower than GHC 6.10 for two programs (Fft2 and Rewrite) it is faster for the other two programs, and the mean runtimes indicate a sequential efficiency of 100%. The GHC-SMP implementation of GpH-SMP has been carefully designed for efficiency, and while a sequential efficiency of 95% is anticipated, we have yet to explain such exceptional performance.

Table 4 summarises the runtimes of the same 4 programs on 8 cores. We make the following observations.

- On 8 cores the variation in runtimes is at most a factor of 4.1 (26.9/6.5), between GpH-SMP and GpH-GUM Rewrite, but is typically rather less.

	SMP	GUM	Eden
Boyer	10.0	14.1	10.1
Clausify	4.7	11.5	5.1
Fft2	13.1	45.8	17.7
Rewrite	6.5	26.9	9.9
Geometric Mean	**8.0**	**21.1**	**9.7**

Table 4 8 Core Parallel Runtime Comparison (seconds).

– The mean 8-core runtimes show that, for this collection of programs, GpH-SMP remains fastest, Eden is just 21% (9.7/8.0) slower, and GpH-GUM slowest by a factor of 2.6 (21.1/8.0).

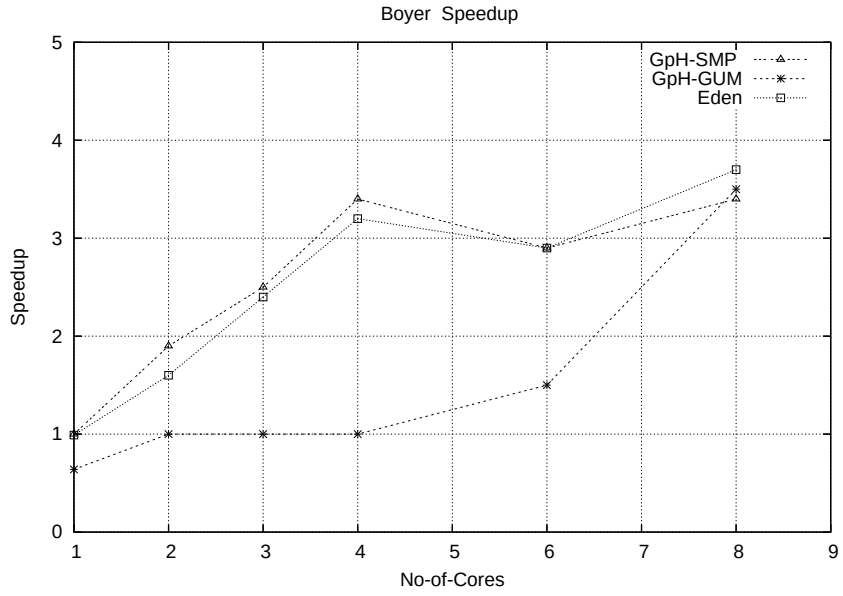
For comparative purposes Figures 7 and 8 show the runtimes and absolute speedups of two of the programs from Table 3, namely Boyer and Rewrite. Boyer and Rewrite are chosen as they give good performance in three of the languages. In Figures 7 and 8 the programs are measured on 1, 2, 3, 4, 6, and 8 cores, and we make the following observations.

– For both programs the runtime curves are broadly similar for all implementations. For GpH-SMP and Eden the curves are very similar, and while the Eden is a little ($< 36\%$) slower on 1 core, the 8 core results are very similar. This is reflected in the speedup graphs where Eden has better 8-core speedups.
– For both programs in all three implementations *scale*, i.e. the runtimes fall as cores are added. The only exceptions are for Boyer between 2 and 4 cores under GpH-GUM and between 4 and 6 cores under GpH-SMP and Eden. This is in marked contrast to FDIP, where the best performance may be achieved under 2, 3 or 4 cores [19], and we shall return to this point in Section 7.2.
– Reflecting the runtime curves, the speedup curves for both programs are broadly similar, and for GpH-SMP and Eden very similar.
– The absolute speedup on a single core reflects the sequential efficiencies of the implementations.

6 Programming Effort and Performance Results

This section investigates the parallel performance of the four parallel Haskells in conjunction with the programming effort required to achieve that performance. Parallel performance is measured as *absolute* speedup over optimised sequential execution (GHC), i.e. not relative to the single core parallel execution. The programming effort is measured using logical source lines of code (SLOC) [14], both as an absolute number and as a percentage of program length. For our purposes SLOC has the advantages of simplicity and relatively wide use. We discuss issues with measuring programming effort, and alternative measures in Section 8.2. We also record the parallel paradigm applied in GpH and Eden.

The following subsections report the programming effort and performance of each language, and the absolute speedups achieved for all 15 programs in the four languages are depicted in Figure 10 and summarised in Table 10. Section 7 then makes a comparison of the approaches.



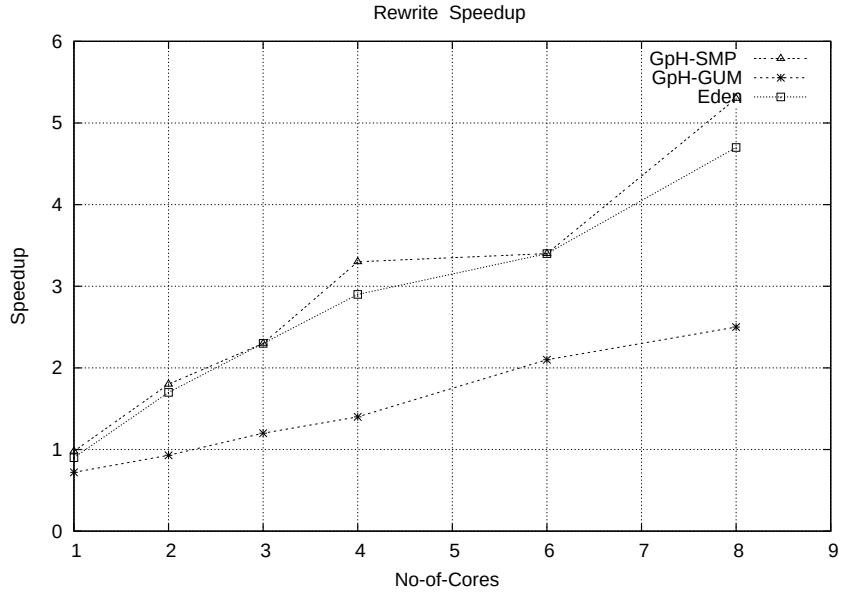


Fig. 8 Absolute Speedup Comparison of Parallel Haskells (Boyer/Rewrite)

6.1 FDIP Multicore Performance

FDIP is entirely implicit, and so no programmer effort is expended other than in profiling and using a special compiler. Similarly the programmer does not need to identify and apply some parallel paradigm. The FDIP performance results reported in this paper are based on the ICFP'07 paper [19], augmented with some additional results from the authors. Where the other parallel Haskells are measured on 8 cores, FDIP performs better on 4 cores than on 8 and hence Table 5 follows [19] in reporting the programs that are improved on 4 cores. It shows that FDIP speeds up only 20% (3/15) of the programs, with a mean speedup of 1.4, and maximum speedup of 1.8.

Program Name	**Speedup**	**Lines Of Code**
Hidden	1.82	316
Atom	1.27	57
Simple	1.27	1053
Geometric Mean	**1.4**	

Table 5 FDIP Programs Improved (4 Cores)

Automatically extracting good parallel performance is acknowledged to be a challenging problem. However some of the reasons for the relatively poor performance of FDIP are that the implementation is immature compared with the other systems and has some known technical problems [19]. Specifically, the profiling simulation ignores several crucial aspects of parallel coordination, namely contention within the GHC runtime system; the locking overheads; and finally the overheads of sparking work and the cache effects of moving data from a 'sparking' core to one running work speculatively.

6.2 GpH-SMP Multicore Performance

Table 6 reports the programming effort and parallel performance of programs improved by GpH-SMP on 8 cores. As a semi-explicit parallel language, GpH requires the programmer to identify a suitable parallel paradigm and introduce evaluation strategies to apply it. Introducing the parallelism requires changing an average of just 9 lines in each program, i.e. 3.2% of the code, and we discuss this further in Section 7.1.

The table shows that GpH-SMP improves more than half of the programs, i.e. 53% (8/15). The mean speedup is 2.9, with a best speedup of 6.6 for Clausify. It is impressive that 3 of the programs achieve speedups of 4 or more on 8 cores, i.e. a parallel efficiency of 50% or more.

6.3 GpH-GUM Multicore Performance

Table 7 reports the programming effort and parallel performance of programs improved by GpH-GUM on 8 cores. Only 12 of the 15 programs are attempted for GpH-GUM as Compress, Hidden and Primetest import modules not available in GHC 4.06.

Program Name	**Spdup**	**Lines Code**	**Lines Chgd**	% **Chgd**	**Paradigm**
Clausify	6.6	101	6	6	Chunked Data Parallelism
Rewrite	5.3	408	14	3	Chunked Data Parallelism
Sphere	4.0	332	12	4	Nested Data Parallelism
Boyer	3.4	295	9	3	Chunked Data Parallelism
Fft2	2.7	705	13	2	Data Parallelism
Primetest	2.0	112	15	13	Chunked Data Parallelism
Hidden	1.8	316	6	2	Nested Data Parallelism
Para	1.2	274	3	1	Data Parallelism
Geometric Mean	**2.9**		**9**	**3.2**	

Table 6 GpH-SMP Programs Improved (8 Cores)

As before, GpH requires the programmer to identify a suitable parallel paradigm and apply it. Introducing the parallelism requires changing an average of just 10 lines of each of these programs, i.e. 3.4% of the code, and we discuss this further in Section 7.1.

The table shows that GpH-GUM improves 42% (5/12) of the programs. The mean speedup is 2.6, with a best speedup of 4.5 for Clausify.

6.4 Eden Multicore Performance

Table 8 reports the programming effort and parallel performance of programs improved by Eden on 8 cores. Eden requires that the programmer identify a suitable parallel paradigm and introduce an appropriately parameterised algorithmic skeleton to exploit it. This set of programs all use the master-worker skeleton discussed in Section 2.2, but some do so directly, while others like Boyer and Rewrite chunk the input to improve thread granularity. Introducing the parallel coordination requires changing an average of just 10 lines in each program, again just 3.6% of the program text.

The table shows that Eden improves a slightly smaller fraction, i.e. 40% (6/15), of the programs than GpH-SMP. The maximum speedup of 6.2 is similar to GpH-SMP (6.6), and the mean speedup is slightly greater, 3.1. It is impressive that 4 of the programs achieve speedups of 3.7 or more on 8 cores, , i.e. a parallel efficiency of 46% or more.

Program Name	Spdup	Lines Code	Lines Chgd	% Chgd	Paradigm
Clausify	4.5	101	6	6	Chunked Data Parallelism
Boyer	3.5	295	9	3	Chunked Data Parallelism
Rewrite	2.5	408	14	3	Chunked Data Parallelism
Sphere	1.8	332	12	4	Nested Data Parallelism
Fft2	1.7	705	13	2	Data Parallelism
Geometric Mean	**2.6**		**10**	**3.4**	

Table 7 GpH-GUM Programs Improved (8 Cores)

Program Name	Spdup	Lines Code	Lines Chgd	% Chgd	Paradigm
Clausify	6.2	101	7	7	Data Parallelism
Rewrite	4.7	408	15	4	Chunked Data Parallelism
Boyer	3.7	295	14	5	Chunked Data Parallelism
Fft2	3.7	705	11	2	Data Parallelism
Compress	1.6	109	3	2	Data Parallelism
Sphere	1.5	332	7	2	Data Parallelism
Geometric Mean	**3.1**		**8**	**3.2**	

Table 8 Eden Programs Improved (8 Cores)

7 Comparative Study

This section compares the best parallel performance of the four Haskell languages and the programming effort required to achieve that performance. Table 9 summarises the key metrics from section 6.

7.1 Programming Effort Comparison

As a purely implicit approach, FDIP requires minimal programmer effort, simply the execution of a profiling run. In contrast GpH and Eden both require programmer effort

Description	FDIP*	GpH-SMP	GpH-GUM	Eden
No. Programs Measured	15	15	12	15
No. Programs Improved	3	8	5	6
% Programs Improved	20%	53%	42%	40%
No. Lines Changed	0	9	10	8
% Code Changed	0	3.2%	3.4%	3.2%
Geometric Mean Speedup	1.4*	2.9	2.6	3.1
* Performance on 4 Cores				

Table 9 Comparative Multicore Performance Summary

to time profile the program, to insert evaluation strategies or skeletons, and to tune the parallel performance. Tables 6, 7, and 8 show that the scale of the program changes is on average small in both absolute and relative terms, e.g. representing just 10 lines or 3.4% of the program text in both languages. We conclude that, for these relatively simple programs, using existing Eden skeletons represents a similar level of coordination abstraction to evaluation strategies in GpH.

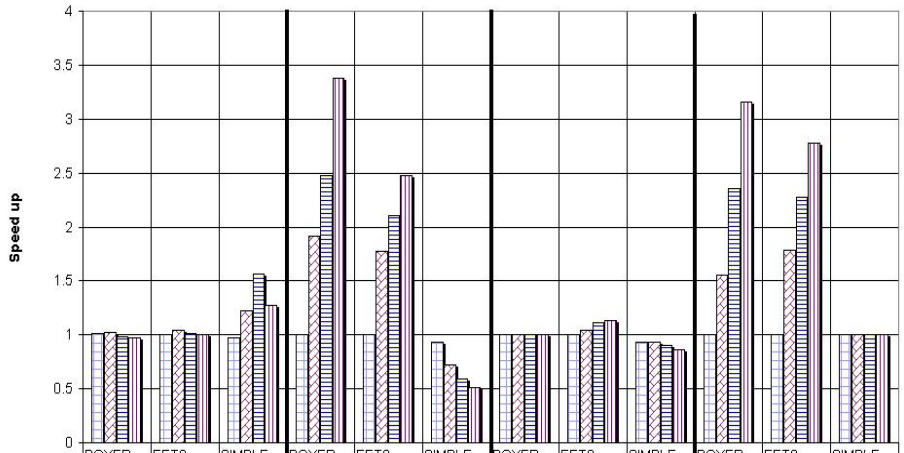
The results also illustrate that in both GpH and Eden some programs are easier to parallelise than others. That is, the scale of program changes induced by parallelisation may vary significantly in both absolute and relative terms. For example Table 6 shows that in GpH the number of lines changed may vary from 3 to 15, and the percentage of program text may vary from 1% to 13%. Similarly, Table 8 shows that in Eden the number of lines changed may vary from 3 to 15, and the percentage of program text may vary from 2% to 7%.

Although in absolute terms the changes required to parallelise the programs are small, the Source Lines of Code (SLOC) metric does not reflect the programmer effort expended on understanding the program, on sequential time/space profiling, and on investigating alternative parallelisations. While time/space profiling is a fast and routine activity, the key intellectual challenge is to understand the computational structure of a program written by another programmer. Some, like Clausify, are simple but others, like SCS, are far more complex. The time to comprehend programs was not measured systematically, but the mean time is roughly estimated at several days. Of course this effort would be reduced if the programmer is parallelising a program he/she wrote. Once the computational structure of the program is understood only half a working day is required to introduce and tune the parallelism. We return to this issue in Section 8.2.

The parallel paradigms used in the improved programs are all forms of data parallelism, sometimes combined with *chunking* to increase thread granularity, or *nesting* to introduce additional parallelism. Section 2 outlines the chunking data parallelism in the Boyer program.

7.2 Scalability

A key property of a parallel implementation in *scalability*, i.e. whether performance increases as processing elements are added. We have already seen the scalability of the GpH-SMP, GpH-GUM and Eden implementations up to 8 cores in the discussion of Figure 7.



	FDIP			GpH-SMP			GpH-GUM			EDEN		
	BOYER	FFT2	SIMPLE	BOYER	FFT2	SIMPLE	BOYER	FFT2	SIMPLE	BOYER	FFT2	SIMPLE
1 PE	1.01	1	0.97	1.0	1.00	0.93	1	1.00	0.93	1.0	1.0	1
2 PE	1.02	1.04	1.22	1.9	1.77	0.72	1	1.04	0.93	1.6	1.8	1
3 PE	0.98	1.01	1.56	2.5	2.11	0.59	1	1.11	0.9	2.4	2.3	1
4 PE	0.97	1	1.27	3.4	2.48	0.51	1	1.13	0.86	3.2	2.8	1

Fig. 9 Comparing the Performance Scalability of Parallel Haskells on 4 Cores.

Figure 9 provides a more detailed analysis for three programs (Boyer, FFT2 and Simple) in each language implementation on 1, 2, 3 or 4 cores. Each program gives good performance on at least one implementation. The figure shows that in GpH-SMP, GpH-GUM and Eden the performance of programs that speedup, i.e. Boyer and FFT2, improves steadily as cores are added. In contrast FDIP delivers the best speedup for Simple on 3 cores. This is not an isolated result: the 5 programs delivering speedups under FDIP reported in [19] deliver maximum speedup twice on 3 cores, and three times on 4 cores.

Furthermore, FDIP ceases to scale beyond 4 cores [38], and this is illustrated by the 4 core performances of Boyer, Simple and FFT2 in Figure 9, which are uniformly better than the 8 core performances reported in Figure 10. The reasons for this have not been established, but are likely to be either lock contention or low-level memory effects, e.g. disrupting caches when transferring threads between cores.

7.3 Performance Comparison

A complete comparison of the 8 core absolute speedups achieved for all 15 programs in the four language implementations is depicted in Figure 10 and summarised in Table 10.

The performance price of FDIP's purely implicit approach is high, and it is the least effective of the languages surveyed here. It improves the fewest number of programs: 3 out of 15 on 4 cores (Table 5), and 2 out of 15 on 8 cores (Figure 10). Moreover the mean and maximum speedup are both relatively small at 1.4 and 1.8 respectively on 4 cores. However, a mean speedup of 1.4 on 4 cores shows parallel efficiency approaching

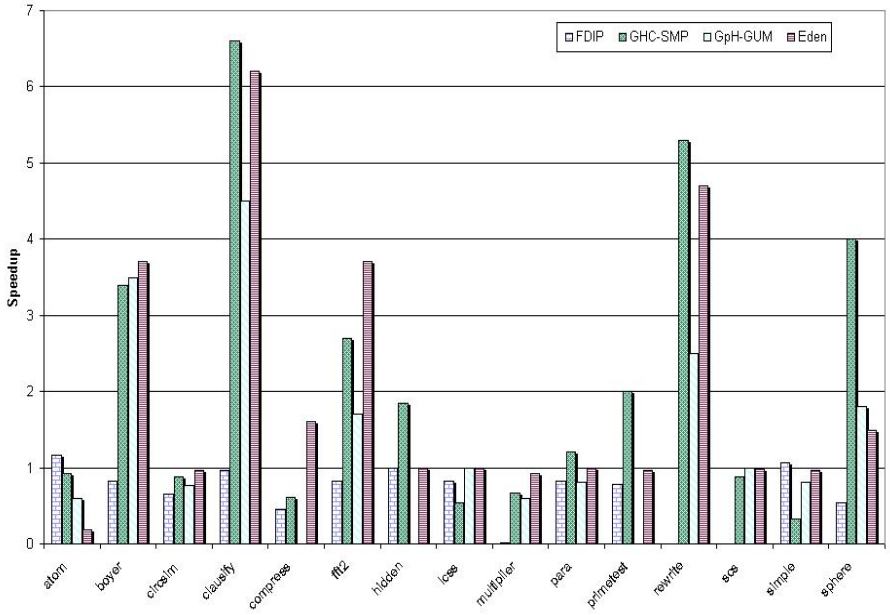


Fig. 10 Performance Comparison of Parallel Haskells (8 cores)

that of the semi-explicit implementations, i.e. speedups of approximately 3.0 on 8 cores. However FDIP parallelism scales both irregularly, and only to a limited extent. That is, FDIP does not deliver significant performance gains beyond 4 cores, and it is hard to predict how many cores will deliver the maximum performance as outlined in section 7.2.

The performance of GpH-SMP and Eden is broadly similar. The mean speedups are similar: 2.9 for GpH-SMP and 3.1 for Eden, as are the maximum speedups: 6.6 for GpH-SMP and 6.2 for Eden. However Eden improves a smaller percentage of the programs: 40% (6/15) compared with 53% (8/15) for GpH-SMP.

The performance of GpH-GUM is marginally worse than GpH-SMP and Eden with mean speedup of 2.6 and maximum speedup of 4.5. GpH-GUM improves an intermediate percentage of programs, i.e. 42% (5/12).

We analyse the implications of these relative performances in section 8.2.

8 Conclusion

8.1 Summary

The preceding sections report a systematic comparison of four functional multicore technologies. The study reflects a snapshot of parallel Haskell technologies and compares the programming effort each variant requires with the parallel performance delivered.

We contrast a 'no pain' approach with three 'low pain' approaches, and start by outlining and comparing the approaches at both language (Section 2) and implementa-

tion (Section 3) levels. We present the design of an experiment that uses 15 programs carefully selected from the representative parts of the nofib suite and hence our results reflect the multicore performance that might be expected for a typical set of Haskell programs (Section 4).

Although the parallel Haskell implementations all use GHC, they each use a different version, and hence the primary performance comparison metric is speedup which normalises against corresponding sequential performance. To ground the speedup comparisons we report sequential and parallel runtimes and efficiencies for three of the languages. We find that sequential runtimes vary by as much as a factor of 2.1, and 8-core runtimes by as much as a factor of 4.1. On a single core GpH-SMP is fastest and GpH-GUM slowest, and sequential efficiencies vary between 74% and 100%. Finally runtime and speedup graphs show that GpH-SMP, GpH-GUM and Eden parallel performance scales, i.e. runtimes fall consistently as cores are added (Section 5).

We report detailed parallel performance and programming effort studies (Section 6), and make a comparative study with the following key results (Section 7).

- FDIP's purely implicit approach requires minimal programmer effort. In contrast GpH and Eden both require programmer effort to understand the program's computational structure, to profile it, to insert parallel coordination, and to tune the parallel performance. As the languages provide high levels of coordination abstraction the program changes are small, on average no more than 4.3% of the program text in both languages. We conclude that Eden skeletons represent a similar high level of coordination abstraction to evaluation strategies in GpH (Section 7.1).
- While GpH-SMP, GpH-GUM and Eden all scale consistently up to 8 cores, FDIP does not scale beyond 4 cores and may deliver best performance on 3 or 4 cores (Section 7.2).
- The performance price of FDIP's purely implicit approach is high: it improves the fewest number of programs (just 3 out of 15) and the mean and maximum speedup are both relatively small at 1.4 and 1.8 respectively on 4 cores (Section 7.3).
- All three semi-explicit approaches improve approximately half of the programs, and the performance of GpH-SMP and Eden is broadly similar with mean and maximum speedups of approximately 3.5 and 6.5. GpH-GUM performance is marginally worse with mean speedup of 2.6 and maximum speedup of 4.5. (Section 7.3).

8.2 Discussion

As multicores become the dominant processor technology it is crucial that functional languages realise their theoretical potential to exploit them effectively. Our study reflects some of the technologies emerging to do so, namely four multicore Haskell implementations, and the results have a number of implications for the field.

It is clear that purely implicit parallelism remains an elusive goal. The FDIP approach speeds up fewer programs, with smaller speedups, and doesn't scale well. While it is not clear that the scaling issues with FDIP are fundamental, the move towards many cores will make scalability a crucial property for languages and implementations.

We have used Source Lines of Code (SLOC) to measure the "pain" inflicted by the semi-explicit approaches, as discussed in Section 6. SLOC is a simple and not uncommon measure of the effort to write, or convert a program from one form to another. However it does not capture the cognitive effort in understanding the computational

behaviour of the original program, and in the cycles of refactoring and testing required to ensure semantic consistency, as outlined in Section 7.1. Measuring cognitive effort is highly problematic, not least because it is deeply individual and subjective. Nevertheless it might be possible to capture more meaningful measures of program change, for example by using a refactoring tool [7] to both profile programmer behaviour and bound the number of refactoring steps. An alternative might be to prove the equivalence of the original and refactored programs, and count the proof steps.

It might be seen as discouraging that, even in the low pain languages, only half of the programs deliver good speedups (Table 10), and that the mean parallel efficiencies are only around 45% (Tables 6, 7, and 8). However recall that these programs were neither designed to be parallel, nor selected for their inherent parallelism. While some algorithms will remain inherently sequential, it is likely with thoughtful design a far higher percentage of programs can be effectively parallelised. Moreover the implementations are evolving fast and will deliver greater parallel efficiencies in the future.

Interestingly Eden, with an implementation designed for distributed memory architectures, performs fractionally better than GpH-SMP which is designed for multicores. Similarly the mean speedups of GpH-GUM, with an architecture designed for both distributed and shared memory systems, are within 10% of the GpH-SMP results. These implementations must have significant advantages to outweigh the massive communication and synchronisation overheads incurred by serialising heap, calling expensive communication libraries, and deserialising heap.

We argue that *the key reason for the good performance of Eden and GUM is the maintainenance independent heaps* using a message-passing architecture. Independent heaps convey three significant advantages for shared-memory systems like multicores. Independent heaps enable cores to garbage collect independently; they confine synchronisation to both limited and large-grain memory areas, i.e. the message buffers; and they simplify cache coherency issues (Section 3.5). We further predict that as multicore scale to many cores the advantages of independent heaps will be greatly magnified, and that some form of thread-private heap, e.g. [11], will be essential on these architectures.

There are many encouraging signs for parallel functional languages. The GpH and Eden semi-explicit approaches deliver effective high level coordination, and hence require very small program changes, and perhaps only half a working day to introduce and tune the parallelism for a known program. The level of interest in addressing the challenges of parallelism is reflected not only in the 4 languages compared here, but also in the rapid emergence and evolution of shared and distributed memory parallel Haskells outlined in Section 2. Most of these languages, like `Par` Monad [29], HdpH [27], Meta-Par [15] provide language-level, rather than built-in parallelism, and once they are stable a systematic study comparing the languages with built-in parallelism with those with language-level parallelism would be worthwhile.

Acknowledgements

Thanks to Rita Loogen, Tim Harris, Simon Marlow, and our anonymous reviewers for constructive feedback. This work has been supported by the European Union grants RII3-CT-2005-026133 SCIEnce: Symbolic Computing Infrastructure in Europe, IST-2011-287510 RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software, and by the UKs Engineering and Physical Sciences Research Council grant

EP/G055181/1 HPC-GAP: High Performance Computational Algebra and Discrete Mathematics.

References

1. Al Zain, A., Berthold, J., Hammond, K., Trinder, P., Michaelson, G., Aswad, M.: Low-Pain, High-Gain Multicore Programming in Haskell: Coordinating Irregular Symbolic Computations on MultiCore Architectures. In: ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP'09). ACM Press (2009)
2. Aswad, M., Trinder, P., Al Zain, A., Michaelson, G., Berthold, J.: Low pain vs no pain multi-core Haskells. In: Z. Horváth, V. Zsók, P. Achten, P. Koopman (eds.) Trends in Functional Programming, 10th International Symposium, Komárno, Slovakia, 2009, Revised Selected Papers (TFP'09), pp. 49–64. Intellect, Bristol (2010)
3. Berthold, J., Dieterle, M., Loogen, R., Priebe, S.: Hierarchical master-worker skeletons. In: PADL, pp. 248–264 (2008)
4. Berthold, J., Loogen, R.: Parallel coordination made explicit in a functional setting. In: Z. Horváth, V. Zsók (eds.) 18th Intl. Symposium on the Implementation of Functional Languages (IFL 2006), Springer LNCS 4449. Budapest, Hungary (2007)
5. Berthold, J., Marlow, S., Hammond, K., Al Zain, A.: Comparing and Optimising Parallel Haskell Implementations for Multicore Machines. In: Tomoya Enokido et al. (ed.) 3rd Int. Workshop on Advanced Distributed and Parallel Network Applications (ADPNA'09). IEEE, New York (2009). (previously presented at IFL'08)
6. Blelloch, G.: Programming Parallel Algorithms. Communications of the ACM **39**(3), 85–97 (1996)
7. Brown, C.: Tool support for refactoring Haskell programs. Ph.D. thesis, University of Kent (2009)
8. Chakravarty, M.M.T. (ed.): ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP'09). ACM Press (2009)
9. Chakravarty, M.M.T., Leshchinskiy, R., Jones, S., Keller, G., Marlow, S.: Data Parallel Haskell: a status report. In: Workshop on Declarative Aspects of Multicore Programming (DAMP'07). ACM Press (2007)
10. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. Ph.D. thesis, University of Edinburgh (1988). Also published in book form by Pittman/MIT, 1989
11. Domani, T., Goldshtein, G., Kolodner, E.K., Lewis, E., Petrank, E., Sheinwald, D.: Thread-local heaps for java. In: SIGPLAN Not, pp. 76–87. ACM Press (2002)
12. Distributed Shared Memory Home Pages. WWW page (2006). `http://www.ics.uci.edu/ javid/dsm.html/`
13. Epstein, J., Black, A.P., Peyton-Jones, S.L.: Towards Haskell in the cloud. In: Haskell '11, Tokyo, Japan, pp. 118–129. ACM (2011)
14. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach. PWS (1998)
15. Foltzer, A., Kulkarni, A., Swords, R., Sasidharan, S., Jiang, E., Newton, R.: A meta-scheduler for the par-monad: Composable scheduling for the heterogeneous cloud. In: ICFP '12, Copenhagen, Denmark. ACM (2012)
16. Grelck, C., Scholz, S.B.: SaC – from High-Level Programming with Arrays to Efficient Parallel Execution. Parallel Processing Letters **13**(3), 401–412 (2003)
17. Hammond, K., Michaelson, G. (eds.): Research Directions in Parallel Functional Programming. Springer (1999)
18. Harris, T., Marlow, S., Jones, S.: Haskell on a Shared-Memory Multiprocessor. In: Haskell '05, pp. 49–61. ACM, New York, NY, USA (2005)
19. Harris, T., Singh, S.: Feedback directed implicit parallelism. In: R. Hinze, N. Ramsey (eds.) ICFP'07, pp. 251–264. ACM (2007)
20. Herrmann, C., Lengauer, C.: $\mathcal{HDC}$: A Higher-Order Language for Divide-and-Conquer. Parallel Processing Letters **10**(2–3), 239–250 (2000)
21. 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. Springer, Aachen, Germany (2000)

22. Loidl, H.W., Rubio Diez, F., Scaife, N., Hammond, K., Klusik, U., Loogen, R., Michaelson, G., Horiguchi, S., Peña-Marí, R., Priebe, S., Rebon Portillo, A., Trinder, P.: Comparing Parallel Functional Languages: Programming and Performance. Higher-order and Symbolic Computation **16**(3), 203–251 (2003)
23. Loidl, H.W., Trinder, P., Hammond, K., Junaidu, S., Morgan, R., Peyton Jones, S.: Engineering Parallel Symbolic Programs in GPH. Concurrency — Practice and Experience **11**(12), 701–752 (1999)
24. Loogen, R.: Programming Language Constructs. In: K. Hammond, G. Michaelson (eds.) Research Directions in Parallel Functional Programming, pp. 63–91. Springer-Verlag (1999)
25. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R., Priebe, S., Rubio, F.: Parallelism Abstractions in Eden. In: Patterns and Skeletons for Parallel and Distributed Computing. Springer (2003)
26. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel Functional Programming in Eden. Functional Programming (2005)
27. Maier, P., Trinder, P.: Implementing a high-level distributed-memory parallel Haskell in Haskell. In: IFL 2011, Lawrence, Kansas, USA. Springer (2012). To appear
28. Marlow, S., Maier, P., Loidl, H.W., Aswad, M.K., Trinder, P.W.: Seq no more: Better strategies for parallel Haskell. In: Haskell '10, Baltimore, USA, pp. 91–102. ACM (2010)
29. Marlow, S., Newton, R., Peyton-Jones, S.L.: A monad for deterministic parallelism. In: Haskell '11, Tokyo, Japan, pp. 71–82. ACM (2011)
30. Michaelson, G., S., H., Scaife, N., Bristow, P.: A Parallel SML compiler based on algorithmic skeletons. Journal of Functional Programming **15**(4), 615–650 (2005)
31. MPI-2: Extensions to the Message-Passing Interface. Tech. rep., University of Tennessee, Knoxville (1997)
32. Partain, W.: The nofib Benchmark Suite of Haskell Programs. In: Glasgow Workshop on Functional Programming, Workshops in Computing, pp. 195–202. Springer-Verlag, Ayr, Scotland (1992)
33. Peña-Marí, R., Rubio, F., Segura, C.: Deriving non-hierarchical process topologies. In: K. Hammond, S. Curtis (eds.) 3rd Scottish Functional Programming Workshop (SFP01), selected papers, *Trends in Functional Programming*, vol. 3. Intellect (2001)
34. Peterson, J., Trifonov, V., Serjantov, A.: Parallel Functional Reactive Programming. In: PADL'00 — Practical Aspects of Declarative Languages, LNCS 1753, pp. 16–31. Springer-Verlag (2000)
35. Peyton Jones, S., Gordon, A., Finne, S.: Concurrent Haskell. In: POPL'96 — Symposium on Principles of Programming Languages, pp. 295–308. ACM, St Petersburg, Florida (1996)
36. Peyton Jones, S., Lester, D.: Implementation of Functional Programming Languages. Prentice Hall (1987)
37. Parallel Virtual Machine Reference Manual. University of Tennessee (1993)
38. Singh, S.: Private Communication about FDIP Performance (2008)
39. Trinder, P.W., Hammond, J., Mattson, J., Partridge, A., Peyton Jones, S.L.: GUM: a portable parallel implementation of Haskell. In: PLDI 1996: Proc ACM SIGPLAN 1996 Conf. on Programming Language Design and Implementation, pp. 79–88. ACM Press, New York, NY, USA (1996)
40. Trinder, P.W., Hammond, K., Loidl, H.W., Jones, P.: Algorithm + Strategy = Parallelism. Journal of Functional Programming **8**, 23–60 (1998)
41. Trinder, P.W., Loidl, H.W., Pointon, R.F.: Parallel and distributed Haskells. Journal of Functional Programming **12**(4&5), 469–510 (2002)
42. Wegner, P.: Programming Languages, Information Structures and Machine Organisation. McGraw-Hill, New York (1971)

A Eden Master Worker Skeleton

Figure 11 shows the implementation of a relatively simple master worker skeleton as an illustration of the Eden skeleton implementation discussed in Section 2.2. Without discussing all details of this implementation, its essential workings are as that tasks are distributed to `n` worker processes, which apply the worker function `wf` to each task and return a pair consisting of the worker number and the result of the task evaluation to the master process, i.e.

the process evaluating `mw`. The worker numbers are interpreted as requests for new tasks. The master uses a function `distribute` to send tasks to the workers according to the (`n*prefetch`) requests initially created and the ones received from the workers. The version shown here returns results in a non-deterministic order, and should be used only in context of a subsequent reduction with a commutative operation (like the logical `and` in the Boyer example).

```
mw :: (Trans t, Trans r) => Int -> Int -> (t -> r) -> [t] -> [r]
mw n prefetch wf tasks = ress
where (reqs, ress) = (unzip . merge) (spawn workers inputs)
-- workers :: [Process [t] [(Int,r)]]
   workers  = [process (zip [i,i..] . map wf) | i <- [0..n-1]]
   inputs   = distribute n tasks (initReqs ++ reqs)
   initReqs = concat (replicate prefetch [0..n-1])
-- task distribution according to worker requests
   distribute :: Int -> [t] -> [Int] -> [[t]]
   distribute np tasks reqs = [taskList reqs tasks n | n<-[0..np-1]]
   where taskList (r:rs) (t:ts) pe | pe == r = t:(taskList rs ts pe)
                                   | otherwise = taskList rs ts pe
         taskList _ _ _ = []
```

Fig. 11 Eden Master-Worker Skeleton (Static Task Pool)

B Comparative Absolute Speedups of Parallel Haskells

The following tables summarises the best absolute speedups obtained on 8 Cores for each of the 15 programs in each of the four parallel Haskells, as visualised in Figure 10.

Program Name	FDIP	GpH-SMP	GpH-GUM	Eden
Atom	1.2	0.9	0.6	0.6
Boyer	0.8	3.4	3.5	3.7
Circsim	0.7	0.9	0.8	1.0
Clausify	1.0	6.6	4.5	6.2
Compress	0.4	0.6	Not tested	1.6
FFT2	0.8	2.7	1.7	3.7
Hidden	1.0	1.8	Not tested	1.0
Lcss	0.8	0.5	0.9	1.0
Multiplier	0.0	0.7	0.9	0.9
Para	0.8	1.2	0.9	1.0
Primetest	0.8	2.0	Not tested	1.0
Rewrite	0.0	5.3	2.5	4.7
Scs	0.0	0.9	1.0	1.0
Simple	1.1	0.3	0.9	1.0
Sphere	0.5	4.0	1.8	1.5

Table 10 Comparative Absolute Speedups of Parallel Haskells on 8 Cores.