Chapter 1

Low Pain vs No Pain Multi-core Haskells

M. KH. Aswad , P. W. Trinder, A. D. AlZain, G. J. Michaelson [1], J. Berthold [2]

Abstract: Multicores 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 the first programming and performance comparison of functional multicore technologies and reports some of the first ever multicore results for two languages. As such it reflects the growing maturity of the field by systematically evaluating four parallel Haskell implementations on a common multicore architecture. The comparison contrasts the programming effort each language requires with the parallel performance delivered. The study uses 15 'typical' programs to compare a 'no pain', i.e. entirely implicit, parallel language with three 'low pain', i.e. semi-explicit languages.

The parallel Haskell implementations use different versions of GHC compiler technology, and hence 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. Our experiments focus on the number of programs improved, the absolute speedups delivered, the parallel scalability, and the program changes required to coordinate parallelism. The results are encouraging and, on occasion, surprising.

[1]School of Mathematics and Computer Sciences Heriot-Watt University,Edinburgh, UK ; {mka19,P.trinder,ceeatia,G.Michaelson}@hw.ac.uk

[2]FB Mathematik und Informatik Philipps-Universit@at Marburg,D-35032 Marburg, Germany;berthold@mathematik.uni-marburg.de

1.1 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. [1].

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, or how parts of the program are placed on processors. Most parallel functional languages incorporate high level coordination sublanguages and a range of models have been used including data parallelism e.g. [5], *semi-explicit* models e.g. [13], coordination languages e.g. [16], and algorithmic skeletons e.g. [14]. The ultimate extreme is to make coordination entirely *implicit*, typically using either profiling as in [11] or parallel iteration as in [8]. The slogan associated with languages with high-level coordination is 'Low Pain Parallelism' and with implicit languages is 'No Pain Parallelism'.

This paper reflects the current status of multicore functional programming by systematically comparing four parallel Haskell implementations on a common multicore architecture (Section 1.4). We compare a 'no pain' parallel implementation Feedback Directed Implicit Parallelism (FDIP) [11], with three 'low pain', i.e. semi-explicit languages [9] (Section 1.2). The semi-explicit Haskells are Eden [13] and two implementations of Glasgow parallel Haskell (GpH) [22], namely GpH-SMP an optimised shared memory implementation integrated in GHC [4],and GpH-GUM a message-passing implementation designed for shared and distributed memory architectures [21] (Section 1.3).

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 1.5). We report detailed parallel performance and programming effort studies focusing on the number of programs improved, speedups delivered, and program changes required to coordinate parallel evaluation (Section 1.6). We make a study comparing the scalability, the programming effort required, and the parallel performance achieved, in each language (Section 1.7). We conclude by summarising the key results and discussing their implications (Section 1.8).

1.2 PARALLEL HASKELL LANGUAGE COMPARISON

This section outlines GpH and Eden, the two semi-explicit Haskell extensions that are compared in the remainder of the paper. The FDIP implicit approach we consider supports an unchanged Concurrent Haskell [17]. Indeed although both Eden and GpH-SMP also support concurrency i.e. multiple stateful (IO) threads, our our language comparisons focuses on parallelism only.

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

FIGURE 1.1. Sequential Top-level Boyer function

We illustrate the coordination extensions by using them to parallelise the Boyer nofib program [15]. Figure 1.1 shows the key top-level function where, in an obvious extension to the original program, the input size `n` is passed as a parameter.

Glasgow parallel Haskell (GpH) *GpH* is a modest extension of Haskell98 with parallel (`par`) and sequential (`seq`) composition as coordination primitives. *Evaluation strategies* abstract over `par` and `seq` to provide lazy, polymorphic, higher-order functions that control the evaluation degree and the parallelism of an expression [22].

```
test :: 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
```

FIGURE 1.2. GpH Top-level Boyer function

Figure 1.2 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 8 chunks and bound to `xs1`. 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. [12].

Eden Eden [13] 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 [13, 2] 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 would not be suitable for parallelism on shared-memory multicore architectures, however recent results have shown good performance [4], as indeed do the results in Sections 1.5, 1.6, 1.7. We return to this issue in Section 1.8

```
test n m  = all (&& True) res
  where xs = take n (repeat (Var X))
        xs1 =  splitAtN  m  xs
        res = parallelMap (all test0) xs1
          where
            np = noPe
            parallelMap = mw np 1

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

FIGURE 1.3. Eden Top-level Boyer function

Figure 1.3 shows the Eden skeleton-based parallelisation of the top-level Boyer `test` function, and works as follows. The input list is chunked as for GpH into `xs1`, and the condition (`all test0`) is mapped over the chunks to give a list of intermediate results `res`. This mapping is parallelised using the master worker skeleton parameterised by the number of cores and the number of tasks to prefetch: `mw np 1`. As before, the final stage is to combine the intermediate results: `all (&& True) res`. As in GpH, just one out of 52 functions has been parallelised, although this time an algorithmic skeleton is used. As in GpH the paradigm is chunked data parallelism.

Language Comparison Table 1.1 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 language, FDIP has the highest level of coordination

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

TABLE 1.1. Language-level Comparison of Parallel Haskells

abstraction, GpH an intermediate level and Eden the lowest. That is, Eden is most explicit about coordination behaviour, but as we shall see in Section 1.7, the use of appropriate skeletons can raise the level of abstraction.

1.3 PARALLEL HASKELL IMPLEMENTATION COMPARISON

All of the parallel Haskells perform parallel graph reduction [18] and support high-level coordination, relying on sophisticated implementations to effectively manage a vast array of low-level coordination issues typically including task placement, communication, synchronisation, and storage management.

Feedback Directed Implicit Parallelism (FDIP) Parallelism is introduced and controlled in FDIP in a four stage process [11] 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 *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.

GpH-SMP Since 2004 the Glasgow Haskell Compiler (GHC) has supported a shared-memory implementation of GpH. 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. In the remainder of the paper we denote this GpH-SMP. The GHC runtime system implements Concurrent Haskell threads using *capabilities* and a system of lightweight threads multiplexed onto a small number of heavyweight OS threads to achieve real parallelism on a multiprocessor, while still keeping overheads of concurrency low [10]. GHC 6.10 supports both

Description	FDIP	GpH-SMP	GpH-GUM	Eden
GHC Version	GHC 6.6	GHC 6.10	GHC 4.06	GHC 6.8
Evaluation Model	Par. Graph Reduction	Par. Graph Reduction	Par. Graph Reduction	Par. Graph Reduction
Granularity Ctrl	Dynamic	Dynamic	Dynamic	Static
Synchronisation Unit	Thunk Locking	Thunk Locking	Thunk Locking	Channel Locking
Work Distribution	Work Stealing	Work Pushing	Work Stealing	Dynamic Proc. Plcment
Work Duplication	Not Poss.	Possible	Not Poss.	Possible
Heap	Shared Heap	Shared Heap	Virtual Shared Heap	Distr. Heap
GC	Depend. & Sequential	Depend. & Parallel	Indep. & Parallel	Indep. & Parallel

TABLE 1.2. Implementation-level Comparison of Parallel Haskells

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.

GUM Implementation of GpH Graph-reduction on a Unified Machine-model (GUM) is a portable, parallel runtime environment for GpH [21]. As the name suggests GUM is designed for both shared and distributed memory architectures. It implements a Distributed Shared Memory (DSM) 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 [19] to provide an architecture neutral and portable runtime environment.

Eden Implementation The Eden implementation extends GHC making a few changes to the front-end, but major modifications to the runtime environment [3]. When run in parallel, each PE 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. Typically, it uses either PVM or MPI libraries.

Implementation Comparison Table 1.3 summarises the implementation level differences between the four parallel Haskells. 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 think more than one, but work may be duplicated in GpH-SMP or Eden.

A key distinction between the implementations is the heap model: while FDIP and GpH-SMP have shared heaps, GUM maintains a virtual shared heap, and Eden supports distributed independent heaps, both supported by message passing. Message passing is essential for distributed systems but initially seems enormously expensive compared with shared memory access. That is, graph 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 GpH-SMP must synchronise to garbage collect, GUM and Eden cores can collect independently and hence in parallel. 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 1.8.

Although both FDIP and GpH-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. [6], would enable independent garbage collection and offer many of the advantages outlined above, without incurring the high communication costs of message passing.

1.4 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 [15]. 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 [11] 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 small to benefit from parallel execution, i.e. where the input cannot be sized to give a runtime of 3s 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. 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 1.2.

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 1.7. Parallel runtimes are variable and to ameliorate these effects the measurements are based on the median runtimes from three executions.

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 *absolute speedup*, i.e. relative to the corresponding optimised sequential GHC compiler, e.g. GpH-SMP speedups are relative to GHC 6.10.1. The absolute speedups are grounded by comparative runtime measurements in Section 1.5.

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.

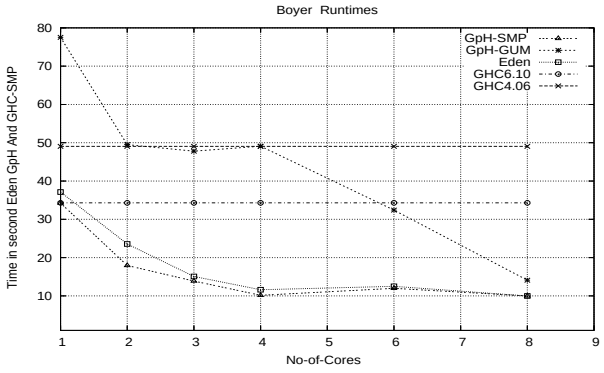


FIGURE 1.4. Runtime Comparison of Parallel Haskells (Boyer)

Program	Sequential			1 Core			8 Core		
	GHC 6.10	GHC 4.06	GHC 6.8	GpH SMP	GpH GUM	Eden	GpH SMP	GpH GUM	Eden
Boyer	34.3	49.3	36.7	34.1	77.52	37.1	10.0	14.1	10.1
Clausify	31.1	51.2	29.1	30.4	78.7	29.3	4.7	11.5	5.1
Fft2	35.8	75.7	48.6	35.9	80.9	49.2	13.1	45.8	17.7
Rewrite	34.3	68.1	46.8	35.1	94.9	52.05	6.5	26.9	9.9
Mean	**33.8**	**61.1**	**40.3**	**33.8**	**83.0**	**41.9**	**8.6**	**24.6**	**10.7**

TABLE 1.3. Sequentialand Parallel Runtime Comparison (seconds).

1.5 RUNTIME COMPARISON

As the parallel implementations use different versions of the GHC compiler (Section 1.4), 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 as an implementation is not available.

Single Core Results Table 1.3 summarises the single core runtimes of 4 nofib programs that deliver good speedup. 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. Columns 5–7 of the table report the 1 core parallel runtimes for each implementation. Columns 8–10 of the table report the 8 cores parallel runtimes for each implementation. We make the following observations.

1 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.

2 The mean sequential runtimes show that GHC 4.06 is the slowest on a single core, and 1.8 (61.1/33.8) times slower than GHC 6.10. This reflects recent GHC performance improvements. GHC 6.8 is 20% (40.3/33.8) 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.

3 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% [21].

Comparing columns 3 and 6, and columns 4 and 7, of Table 1.3 shows that this expectation is met for GpH-GUM and Eden, with mean sequential efficiencies of 74% (61.1/83.0) and 96% (40.3/41.9) 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%. 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.

Eight Core Results Table 1.3 (columns 8–10) summarises the runtimes of the same 4 programs on 8 cores. We make the following observations.

1 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. **2** The mean 8-core runtimes show that, for this collection of programs, GpH-SMP remains fastest, Eden is just 25% (10.7/8.6) slower, and GpH-GUM slowest by a factor of 2.8 (24.6/8.6).

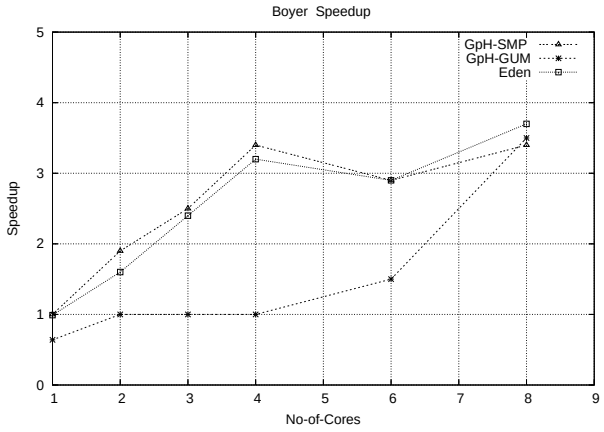


FIGURE 1.5. Absolute Speedup Comparison of Parallel Haskells (Boyer)

Runtime and Speedup Graphs Figures 1.4 and 1.5 compare the runtimes and absolute speedups of Boyer program from Table 1.3. The program is measured on 1, 2, 3, 4, 6, and 8 cores. We make the following observations.

1 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.

2 Three implementations *scale*, i.e. the runtimes fall as cores are added. The only exceptions are 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 [11], and we shall return

Program Name	Speedup	Lines Of Code
Hidden	1.82	316
Atom	1.27	57
Simple	1.27	1053
Mean	**1.5**	

TABLE 1.4. FDIP Programs Improved.

to this point in Section 1.7.

3 Reflecting the runtime curves, the speedup curves for the program are broadly similar, and for GpH-SMP and Eden very similar.

4 The speedup on a single core reflects the sequential efficiencies of the implementations.

1.6 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 and programming effort as logical source lines of code (SLOC) [7], 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 also record the parallel paradigm applied in GpH and Eden. The absolute speedups achieved for all 15 programs in the four languages is depicted in Figure 1.7. The following section compares the approaches.

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 [11], 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 1.4 follows [11] 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 1.5, and maximum speedup of 1.8.

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 because the implementation is a immature compared with the other systems and has some known technical problems [11]. Specifically, the simulation ignores several crucial aspects of parallel coordination, namely contention within the GHC run-time system; the overheads of the shim-lock implementation; and finally the overheads of sparking work and the cache effects of moving data

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 DataParallelism
Boyer	3.4	295	9	3	Chunked DataParallelism
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
Mean	**3.4**		**10**	**4.3**	

TABLE 1.5. GpH-SMP Programs Improved.

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
Sphere	1.8	332	12	4	Nested Data Parallelism
Fft2	1.7	705	13	2	Data Parallelism
Mean	**3.1**		**11**	**3.6**	

TABLE 1.6. GpH-GUM Programs Improved.

from a sparking core to one running work speculatively.

GpH-SMP Multicore Performance Table 1.5 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 10 lines in each program, i.e. 4.3% of the code, and we discuss this further in Section 1.7.

The table shows that GpH-SMP improves more than half of the programs, i.e. 53% (8/15). The mean speedup is 3.4, 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.

GpH-GUM Multicore Performance Table 1.6 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	6.2	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
Mean	**3.6**		**10**	**3.6**	

TABLE 1.7. Eden programs Improved.

As before, GpH requires the programmer to identify a suitable parallel paradigm and apply it. Introducing the parallelism requires changing an average of just 11 lines of each of these programs, i.e. 3.6% of the code, and we discuss this further in Section 1.7. The table shows that GpH-GUM improves 42% (5/12) of the programs. The mean speedup is 3.1, with a best speedup of 4.5 for Clausify.

Eden Multicore Performance Table 1.7 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 1.2, 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 and GpH-GUM. The maximum speedup of 6.2 is similar to GpH-SMP (6.6), and the mean speedup is slightly greater, 3.6. 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.

1.7 COMPARATIVE STUDY

This section compares the best parallel performance of the four Haskell languages and the programming effort required to achieve that performance. Table 1.8 summarises the key metrics from section 1.6.

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 to time profile the program, to insert evaluation strategies or skeletons, and to tune the parallel performance. Tables 1.5, 1.6, and 1.7 show that the scale of the program changes is on average

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	10	11	10
% Code Changed	0	4.3	3.6	3.6
Mean Speedup	1.5*	3.4	3.1	3.6
* Performance on 4 Cores				

TABLE 1.8. Comparative Multicore Performance Summary

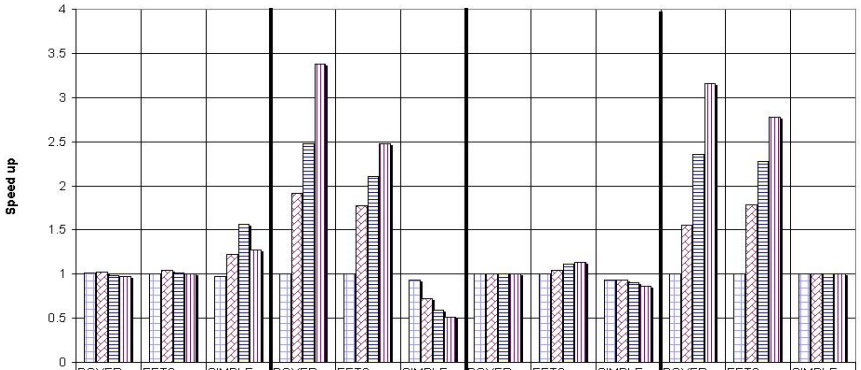
small in both absolute and relative terms, e.g. representing just 11 lines or 4.3% of the program text in both languages. We conclude that, for these relatively simple programs, using existing Eden skeletons represents a similar level of coordination abstraction to evaluation strategies in GpH.

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 1.5 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 1.7 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%.

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. Although the parallelisation changes are small, the source lines of code 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.

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 1.4.

Figure 1.6 provides a more detailed analysis for three programs (Boyer, FFT2 and Simple) in each language 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 [11] deliver maximum speedup twice on 3 cores, and three times on 4 cores.



	FDIP			GpH-SMP			GpH-GUM			EDEN		
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

FIGURE 1.6. Comparing the Performance Scalability of Parallel Haskells on 4 Cores.

Furthermore, FDIP ceases to scale beyond 4 cores [20], and this is illustrated by the 4 core performances of Boyer, Simple and FFT2 in Figure 1.6, which are uniformly better than the 8 core performances reported in Figure 1.7. 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.

Performance Comparison A complete comparison of the 8 core speedups achieved for all 15 programs in the four languages is depicted in Figure 1.7. 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 1.4), and 2 out of 15 on 8 cores (Figure 1.7). Moreover the mean and maximum speedup are both relatively small at 1.5 and 1.8 respectively on 4 cores. However, a mean speedup of 1.5 on 4 cores shows parallel efficiency approaching that of the semi-explicit implementations, i.e. speedups of approximately 3.5 on 8 cores. 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 (Section 1.7).

The performance of GpH-SMP and Eden is broadly similar. The mean speedups are similar: 3.4 for GpH-SMP and 3.6 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.

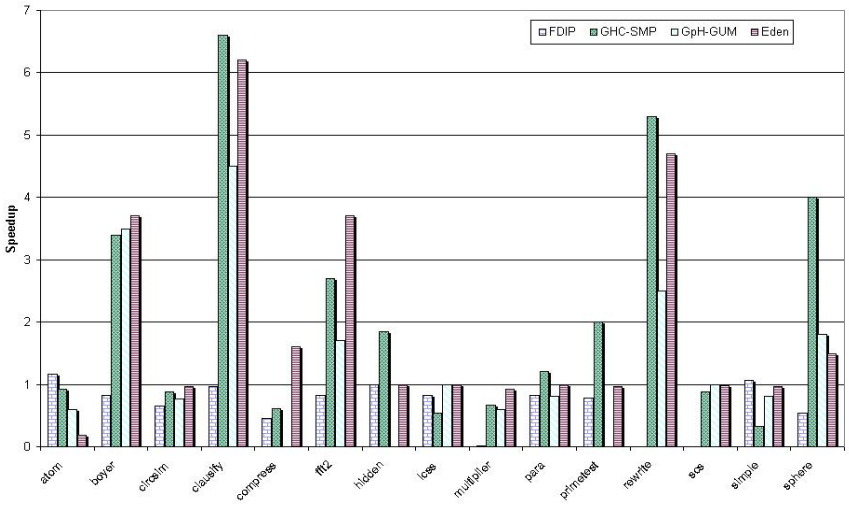


FIGURE 1.7. Performance Comparison of Parallel Haskells (8 cores)

The performance of GpH-GUM is marginally worse than GpH-SMP and Eden with mean speedup of 3.1 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 1.8.

1.8 CONCLUSION

Summary The preceding sections report the first comparison of functional multicore technologies and are some of the first ever GpH-GUM and GpH-SMP multicore results. We contrast a 'no pain' approach with three 'low pain' approaches, and start by outlining and comparing the approaches at both language (Section 1.2) and implementation (Section 1.3) levels. We present the design of an experiment using 15 programs carefully selected to reflect the multicore performance that might be expected for a typical set of Haskell programs (Section 1.4).

Although the parallel Haskell implementations all use GHC, they each use a different version, and hence the primary performance comparison metric is absolute speedups, which normalises against 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 1.5).

We report detailed parallel performance and programming effort studies (Section 1.6), and make a comparative study with the following key results (Section 1.7).

1 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 1.7).

2 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 1.7).

3 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 1.5 and 1.8 respectively on 4 cores (Section 1.7).

4 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 3.1 and maximum speedup of 4.5. (Section 1.7).

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.

It might be seen as discouraging that, even in the low pain languages, only half of the programs deliver absolute speedups, and that the mean parallel efficiencies are only around 45% (Tables 1.5, 1.6, and 1.7). 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 we can expect greater parallel efficiencies in the near 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 four significant advantages for shared-memory systems like multicores. It enables cores to garbage collect independently, it confines synchronisation to both limited and large-grain memory areas, i.e. the message buffers, and simplifies cache coherency issues (Section 1.3). 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. [6], will be essential on these architectures.

There are many encouraging signs for multicore 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 fact that there are 4 multicore Haskells to compare reflects the level of interest in addressing the challenges. The implementations have considerable room for improvement. For example where GpH-GUM and Eden currently use very naive distributed memory implementations, these could be significantly improved for shared-memory multicores, e.g. using shared-memory variants of the communication libraries. Another promising line of future work is to integrate distributed and shared-memory implementations to better exploit the increasingly ubiquitous clusters of multicore architectures.

Acknowledgements Thanks to Rita Loogen, Tim Harris, and Simon Marlow for constructive feedback. This research is supported by European Union Framework 6 grant RII3-CT-2005-026133 SCIEnce.

REFERENCES

[1] A. Al Zain, B. J., H. K., T. P., M. G.J, and A. M.K.C. 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] J. Berthold, M. Dieterle, R. Loogen, and S. Priebe. Hierarchical master-worker skeletons. In *PADL*, pages 248–264, 2008.

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

[4] J. Berthold, S. Marlow, A. Al Zain, , and K. Hammond. Comparing and Optimising Parallel Haskell Implementations on Multicores. In *IFL'08 International Symposia*

on Implementation and Application of Functional Language ,Draft proceeding, Hatfield, UK, 2008.

[5] M. M. T. Chakravarty, R. Leshchinskiy, S. Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In *Workshop on Declarative Aspects of Multicore Programming (DAMP'07)*. ACM Press, 2007.

[6] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for java. In *SIGPLAN Not*, pages 76–87. ACM Press, 2002.

[7] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS, 1998.

[8] C. Grelck and S. B. Scholz. SaC – from High-Level Programming with Arrays to Efficient Parallel Execution. *Parallel Processing Letters*, 13(3):401–412, 2003.

[9] K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer, London, 1999.

[10] T. Harris, S. Marlow, and S. Jones. Haskell on a Shared-Memory Multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, pages 49–61, New York, NY, USA, 2005. ACM.

[11] T. Harris and S. Singh. Feedback directed implicit parallelism. *SIGPLAN Not.*, 42(9):251–264, 2007.

[12] H.-W. Loidl, P. Trinder, K. Hammond, S. Junaidu, R. Morgan, and S. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11(12):701–752, October 1999.

[13] R. Loogen, Y. Ortega-mallén, and R. Pena-marí. Parallel Functional Programming in Eden. *Functional Programming*, 2005.

[14] G. Michaelson, H. S., N. Scaife, and P. Bristow. A Parallel SML compiler based on algorithmic skeletons. *Journal of Functional Programming*, 15(4):615–650, 2005.

[15] W. Partain. The nofib Benchmark Suite of Haskell Programs. In *Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 195–202, Ayr, Scotland, 1992. Springer-Verlag.

[16] J. Peterson, V. Trifonov, and A. Serjantov. Parallel Functional Reactive Programming. In *PADL'00 — Practical Aspects of Declarative Languages*, LNCS 1753, pages 16–31. Springer-Verlag, 2000.

[17] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL'96 — Symposium on Principles of Programming Languages*, pages 295–308, St Petersburg, Florida, Jan. 1996. ACM.

[18] S. Peyton Jones and D. Lester. *Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[19] *Parallel Virtual Machine Reference Manual*. University of Tennessee, Aug 1993.

[20] S. Singh. Private Communication about FDIP Performance, 11 2008.

[21] P. W. Trinder, J. Hammond, J. Mattson, A. Partridge, and S. L. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *PLDI 1996: Proc ACM SIGPLAN 1996 Conf. on Programming Language Design and Implementation*, pages 79–88, New York, NY, USA, 1996. ACM Press.

[22] P. W. Trinder, K. Hammond, H. W. Loidl, and P. Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8:23–60, 1998.