Chapter 1

GpH and Eden: Comparing Two Parallel Functional Languages on a Beowulf Cluster

H-W. Loidl[1], U. Klusik[2], K. Hammond[3], R. Loogen[2], P.W. Trinder[1]

Abstract: We investigate two similar but contrasting parallel functional language designs: Eden and GPH. Both languages use the non-strict functional language Haskell as a core expression language, both are implemented as extensions of the high performance Glasgow Haskell Compiler (GHC), and both implementations are available on the same distributed architecture: a Beowulf cluster. This allows an exceptionally pure comparison of the language design characteristics and their impact on parallel performance.

The comparison is illustrated by two parallel symbolic computation benchmarks which expose differences in the communication, process creation, and work distribution mechanisms employed by Eden and GPH. Our results show that the explicit process model favoured by Eden gives good parallel performance for coarse-grained applications running on the Beowulf cluster. In comparison, the current GPH implementation of implicit parallelism yields poorer absolute speedup for these two applications. Further work is needed to determine whether this difference is an implementation artefact or a consequence of the different models employed in each case, though excessively fine thread granularity appears to be a contributing factor.

[1]Department of Computing and Electrical Engineering, Heriot-Watt University, Scotland; Email: {`hwloidl,trinder`}`@cee.hw.ac.uk`

[2]Department of Computer Science, Marburg University, Germany; Email: {`klusik,loogen`}`@mathematik.uni-marburg.de`

[3]School of Computer Science, University of St Andrews, Scotland; Email: `kh@dcs.st-andrews.ac.uk`

1.1 INTRODUCTION

An exciting recent development is the emergence of parallel systems based on *commodity* personal computers. So-called "Beowulf" clusters [RBMS97] are becoming especially popular as low-cost, high-performance systems based on inexpensive (but high latency) LAN technology. In principle, such architectures make parallel computation available to a much wider class of users. In practice, many potential applications fall outside the well studied area of regularly parallel scientific computations. This paper considers two functional language approaches that aim to provide easy parallelisation of such applications, GPH [THM+96] and Eden [BLOP97]. The paper compares the effectiveness of both the languages and their implementations on two archetypical parallel benchmarks.

1.2 GPH AND EDEN

GPH and Eden are both conservative extensions of the non-strict functional language Haskell [PH97]. They take significantly different approaches to parallelism: Eden incorporates explicit process creation and interconnection that can be exploited by the programmer to match the target architecture; GPH uses an annotation-based approach in which process (thread) creation, distribution etc. are under automatic dynamic control. In sharp contrast to conventional approaches, such as C+MPI [MPI94], both languages employ sophisticated dynamic techniques to automatically manage many complex details of parallel programming including communication, task synchronisation and the mapping of tasks to processors. These issues and other high level functional approaches to parallel programming are covered in detail elsewhere [HM99].

The contrasting language approaches are motivated by their design philosophies. GPH specifically targets *transformational* (parallel) programs on a range of tightly-coupled parallel architectures from shared-memory to distributed-memory machines. The Beowulf represents one extreme of this range. The overriding design objective is to achieve acceptable parallelism at minimal cost in introducing changes to the source code of the underlying sequential program. The design goal of Eden is to target both transformational and *reactive* (concurrent) programs on distributed-memory machines as epitomised by the Beowulf architecture. The Eden design philosophy accepts that significant changes to the source code may be required to achieve an effective parallel implementation.

1.2.1 Coordination Constructs in GpH

GPH uses only two primitives to coordinate parallel threads (Table 1.1): parallel composition (`par`) to indicate potential parallelism, and sequential composition (`seq`) to sequence computations. Higher-level forms of parallelism can be described by abstracting over these primitives using normal Haskell function definitions. These abstractions are called *evaluation strategies*: lazy, polymorphic, higher-order coordination functions. For example `rnf` is a strategy that causes

TABLE 1.1. Coordination Constructs

No.	Aspect	GPH	Eden
1.1a	primitives	`par, seq`	process abstractions/instantiations
1.1b	abstractions	evaluation strategies	implementation skeletons

sequential evaluation to normal form, and `parList` is a strategy that indicates that every element of a list should be evaluated in parallel [THLP98]. Strategies are applied by the function `using :: a -> Strategy a -> a`.

Example 1.1. The following parallel GPH variant of the well-known higher-order function `map` permits the parallel evaluation of every element of the result list with the strategy passed as the first argument. The definition clearly separates the specification of the algorithm (`map f xs`) from that of the dynamic (parallel) behaviour (`parList s`).

```
parMap        :: Strategy b -> (a -> b) -> [a] -> [b]
parMap s f xs =  map f xs `using` parList s
```

A corresponding function can be defined in Eden.

1.2.2 Coordination Constructs in Eden

An Eden program defines a system of processes which exchange data on one-to-one communication channels. Processes are defined by *process abstractions* analogous to the definition of functions by λ-abstractions. A process abstraction specifies the mapping of data input via inports to data output via outports. Inports and outports connect communication channels to processes. Communication channels are not autonomous objects, but tightly coupled with processes. Processes and their interconnecting channels are created by the evaluation of *process instantiations* which apply process abstractions to expressions defining their incoming data (Table 1.1). The result of a process instantiation is the tuple of outgoing data of the newly created process. Eden processes use independent *threads* to produce their outputs. For each output a separate thread is created which evaluates the output expression to normal form and sends the result value via the corresponding outport. Lists are transmitted as streams, i.e. element-wise.

Example 1.2. Figure 1.1 shows the definition of a generic master worker system, called *workpool*, in Eden. This system is a simplified version of the general replicated workers scheme (see e.g. [Les93]). It uses a set of worker processes to process a list of tasks. Initially the workpool (master) distributes a fixed amount (`prefetch`) of tasks among the worker processes. Afterwards, it sends a new task to a worker only when it receives a result from this worker. The results of the tasks are returned in the order they are received from the workers. This is implemented using the predefined `merge` process abstraction whose instantiation leads to the creation of a nondeterministic fair merge process. The type class

```
workpool :: (Transmissible task, Transmissible res) =>
            Int ->                -- number of workers
            Int ->                -- prefetch number
            (task -> res) ->      -- solveTask function
            [task] -> [res]
workpool noW pf solveTask tasks = results
 where
   worker      = process tasks -> map solveTask tasks
   -- process abstraction defining worker behaviour
   workerList  = [ worker # ts | ts <- toWorkers]
   -- creation of worker processes by
   -- process instantiations with Eden operator (#)

   fromWorkers = merge # (tagWithPids workerList)
   -- worker outputs are tagged with unique process ids
   -- and collected using nondeterministic merge process
   toWorkers   = distribute noW requests tasks
   -- tasks are distributed to noW workers according to
   -- list of requests which contains process ids

   requests    = initialReqs ++ newReqs
   initialReqs = concat (replicate prefetch [0..noW-1])
   -- define pf initial requests for each worker
   newReqs     = map (\ (pid,res) -> pid) fromWorkers
   -- get process ids of workers delivering results

   results     = map (\ (pid,res) -> res) fromWorkers
   -- get results out of tagged worker outputs
```

FIGURE 1.1. A Workpool in Eden

(`Transmissible a`) provides (implicitly used) functions for the transmission of values of type `a`.

The Eden definition clearly determines which processes are created, how tasks are distributed among processes, and which data must be communicated between these processes and the main program along appropriate communication channels. Note that it is not possible to give a corresponding implementation of this scheme in GPH, since it relies on nondeterministic merge to collate results and to distribute new tasks to the workers. Thus, although many algorithmic skeletons can be defined in GPH [HR99], Eden also allows the definition and implementation of certain skeletons such as the `workpool` above [KLPR00] that cannot be given definitions in GPH.

1.2.3 Strictness Issues

In non-strict languages such as Eden and GPH, it may be necessary to introduce additional strictness in order to obtain good parallel performance. Eden eagerly evaluates channel outputs to increase parallelism by exposing data structures that can then be communicated earlier than otherwise possible. Moreover, in some contexts processes are created eagerly. Similarly, evaluation strategies may be

TABLE 1.2. Process Model

No.	Aspect	GPH	Eden
1.2a	processes&threads model	only threads	processes&threads
1.2b	process creation	—	explicit&mandatory
1.2c	thread creation	implicit&optional	implicit&mandatory
1.2d	proc.&thread relationship	—	fixed:1 thread per outport
1.2e	proc.&thread placmt	implicit&dynamic	implicit&static
1.2f	scheduling	not specified (unfair implemented)	fair

used in GPH programs to increase strictness with the intention of enhancing parallel behaviour.

1.2.4 Process and Thread Models

Processes and threads are key differences between Eden and GPH. Both languages have the same notion of (lightweight) threads: units of computation that share a name (or address) space with other threads, with inter-thread synchronisation and communication both occurring via shared data. In the case of GPH the address space is global across the machine. The concept of a process is, however, rather different in each language. Each Eden process has its own name space, a set of inports and outports, with each of the outports having an associated thread. They are created as a result of dynamic process instantiations. In contrast, there is precisely one process in a GPH program corresponding to the main (possibly state-manipulating) computation.

The languages' approaches to process and thread issues are as follows. The numbers refer to the summary in Table 1.2. Does the language support processes, threads or both (1.2a)? GPH has only threads, but Eden has both threads and processes. Is process and thread creation explicit or implicit, and is it *mandatory* i.e. the runtime-system must create the thread, or *optional*(1.2b,1.2c)? Thread creation in GPH is implicit because `par` is optional. In Eden the creation of processes is explicit and mandatory during process instantiation, but thread creation, e.g. to evaluate each outport expression is implicit. Is there a relationship between processes and threads (1.2d)? In Eden there is a fixed relationship with one thread per outport in a process.

How are the processes placed on PEs (1.2e)? GPH thread placement is dynamic and implicit as they are stolen by idle PEs. Eden process placement is static and implicit, placing processes in round robin manner on the available PEs at the start of a program. Which scheduling strategy is used, i.e. which thread is run on a processor, if more than one runnable thread is available (1.2f)? The current implementation of GPH uses unfair scheduling whereas Eden requires fair scheduling. Unfair scheduling has the advantage of a smaller runtime overhead,

TABLE 1.3. Storage Management

No.	Aspect	GPH	Eden
1.3a	address space	global	disjoint [copying]
1.3b	garbage collection	local&global	local

but runs the danger of indefinitely delaying work, which may be important for the result of the computation. Fair scheduling is needed in order to guarantee the independent progress of all threads in an Eden process and of all Eden processes on the same processor element.

1.2.5 Storage Management

Eden and GPH have different storage models, as discussed below, and summarised in Table 1.3. What memory can a process address (1.3a)? While GPH relies on a globally shared address space, Eden processes have disjoint address spaces. When a process is created, its whole environment is copied and transferred into the local address space of the process. Sharing may be lost by this copying, and this may result in the duplicated evaluation of some subexpressions. The duplication of work can however always be avoided by communicating shared subexpressions. In GPH sharing of subexpressions is always retained and thus, work will never be duplicated. How is the address space garbage collected (1.3b)? GPH garbage collects both the local address space of each PE (independently of other PEs), and returns expired global references to the owning PE. Eden need only perform (independent) local garbage collection because of the separated disjoint address spaces of the processes.

Details of the implementations and in particular of the storage management models can be found in [THM$^+$96, BKL98, Klu00].

1.2.6 Communication and Synchronisation Model

Both languages combine the notions of communication and synchronisation: threads and processes synchronise when they communicate. The communication models have many other similarities, only point-to-point communication is employed[4], functions or partial applications can be transmitted for remote evaluation, and the compiler guarantees that communication is type safe.

There are, however, some important differences between the communication models, as discussed below and summarised in Table 1.4. Is the communication medium explicit in the language (1.4a)? In GPH communication is implicit, occurring when threads share variables. Eden combines channel communication, with shared variables in process abstractions that are implicitly copied during a process instantiation. Are communication actions explicit in the language (1.4b)?

[4]In Eden, many-to-one communication can however be modelled by the nondeterministic `merge` process.

TABLE 1.4. Communication Model

No.	Aspect	GPH	Eden
1.4a	comm. mechanism	implicit:shared vars	explicit:channels & implicit:shared vars
1.4b	communication	implicit	implicit
1.4c	evaluation	semi-eager	eager
1.4d	work duplication	no	yes

In both languages, communication actions occurs implicitly. In GPH, e.g. the value of a shared variable is communicated when a thread on a remote PE demands it. In Eden, e.g. the value of an outport is transmitted to the connected inport.

How eagerly is data communicated (1.4c)? In lazy languages it is natural to send only the data demanded which introduces many small messages. High latency networks can be better utilised by communicating a smaller number of bigger messages. To increase the message size GPH is *semi-eager*: it attempts to fill a communications packet with graph 'near' to the demanded node. Eden eagerly communicates data: each process evaluates its output expressions to normal form and sends the result values as soon as they are available. Is work ever duplicated as a result of communication (1.4d)? As discussed in Section 1.2.5, Eden may duplicate work as the result of process creation.

1.3 EXPERIMENTAL RESULTS

This section describes the results we have obtained using two realistic applications: a symbolic linear system solver (discussed in more detail in [Loi97]); and a ray tracer for spheres (discussed in more detail in [HR99]). Both programs are typical symbolic computation programs, operating over complex data structures and potentially generating a large amount of irregular parallelism. We believe that both Eden and GPH are well suited for this class of problems, making use of largely-implicit parallelism and dynamic resource management.

1.3.1 Experimental Framework

Our experiments were carried out on an homogeneous 64-processor Beowulf cluster [RBMS97] consisting of Linux RedHat 6.2 workstations with a 450MHz Pentium II processor, 384MB of DRAM and 8GB of IDE disk. The workstations are connected through a CISCO 2984G full duplex 100Mb/s fast Ethernet switch with a latency of 142μs, measured under PVM 3.4.2. For administrative reasons we could use only a sub-network of up to 16 machines for most of our measurements.

```
rnf noOfPrimes                                  `seq`
parListN noOfPrimes par_sol_strat xList         `par`
parList rnf xs
where
  par_sol_strat :: Strategy [Integer]
  par_sol_strat = \ (p:modDet:pmx) -> rnf modDet `seq`
                                      if modDet /= 0
                                        then parList rnf pmx
                                        else ()
```

FIGURE 1.2. Parallel strategy for LinSolv (GpH version)

1.3.2 LinSolv

Problem Description

The LinSolv algorithm discussed in this section finds an exact solution of a linear system of equations of the form $Ax = b$ where $A \in \mathbb{Z}^{n \times n}, b \in \mathbb{Z}^n$. This algorithm has been discussed in detail in [Loi97, LTH$^+$99]. Here we focus on the difference between the GPH and Eden versions.

Parallel Algorithm

In order to find an exact solution for a given system of equations, LinSolv uses a *multiple homomorphic images* approach. This approach, which is very common in the area of computer algebra, consists of the following three stages: map the input data into several homomorphic images; compute the solution in each of these images; and combine the results of all images to a result in the original domain ('lifting').

For complex, unbounded domains (here $\mathbb{Z}$) this approach offers high sequential performance because the computations in the homomorphic images (here $\mathbb{Z}$ modulo p) are comparatively cheap: they use fixed rather than arbitrary precision arithmetic. Only in the combination phase do expensive arbitrary precision values appear again. In the solution phase we use Cramer's rule, which requires $n + 1$ determinant computations.

This approach also lends itself to parallel computation since the computations of the homomorphic solutions are independent. The basic structure of the algorithm is a parallel map over the homomorphic solutions, modified to work over infinite lists. Additionally, a divide-and-conquer parallel version of the determinant computation is used for matrices larger than 5×5.

Comparison of Eden and GpH versions

GpH version. The GPH version of LinSolv exploits laziness by using an infinite list `xList` representing the results of all homomorphic images together with the prime number (`p`), as the basis of the image, and the value of the determinant of A in that image (`modDet`). Figure 1.2 shows the top level strategy, i.e. only the control code, that is used in the parallelisation of the algorithm. The strategy guesses the number of primes needed to compute the overall result (`noOfPrimes`) and uses a `parListN` strategy to generate data parallelism over the initial segment of `xList`. Using `parList` inside the `par_sol_strat` strategy, which is applied to the solution in every image, causes each component of the homomorphic result to be evaluated in parallel. For each image we need to check whether the determinant of the input matrix is zero. In that case the prime number is "unlucky" and the result in that image cannot be used during lifting. We avoid the computation in that case but keep it in the result list (only `p` and `modDet` will have been evaluated). If some prime numbers turn out to be unlucky the lifting phase will evaluate more results by demanding a so far unevaluated list element. The final strategy application `parList rnf xs` specifies that all elements of the result vector should be combined in parallel.

Eden version. The Eden version is a simple top-level parallelisation using the generic workpool scheme introduced in Figure 1.1. The `workpool` is instantiated by `xList = workpool 2 np getHom primes`, i.e. the list of `primes` is the task list distributed to `np` worker processes which compute the solutions in the corresponding homomorphic images using the function `getHom`. A pre-fetch value of two is chosen to achieve some overlapping of computation and communication from the beginning of the execution. The determinant computation is not parallelized as it delivers excessively fine-grained tasks which cause too much overhead.

Although this version bears the danger of increasing computation by potentially computing an infinite number of solutions (*"unlimited speculation"*), it performs best for small numbers of processors. A more thorough discussion of different parallelisations of LinSolv, including a version where the speculation is restricted algorithmically, can be found in [Klu00].

Performance Results

For the measurements we used a dense input matrix of size 9×9 (sequential runtime 61.16s) and a sparse matrix of size 14×14 (sequential runtime 552.59s).

Figure 1.3 shows the absolute speedups for the GPH and Eden implementations of LinSolv for up to 16 PEs on the Beowulf. In the GPH measurements, and for the small input matrix of dimension 9×9 the speedup evens out after about 6 PEs because of a lack of available work. This can be seen from the number of threads generated: from 2 to 4 PEs it increases from 117 to 208 but after that it reaches the limit of 217. In the cases where sufficient work is available, GPH's thread subsumption mechanism [THM$^+$96] ensures that no excess of fine-grained threads is generated in the divide-and-conquer determinant computation. Overall, the sparse input matrix yields the best results (absolute speedup of 7.5 on 12 PEs). The larger input size causes a larger number of top-level threads to be generated. Since these threads perform relatively little communication the parallel overhead is reduced. This example also shows that our system can handle irregular parallelism, because a sparse matrix will produce parallel threads of widely differing

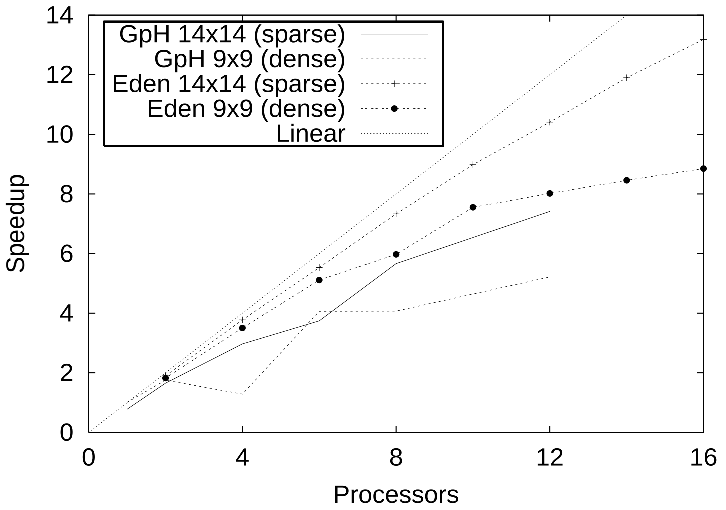

FIGURE 1.3. Absolute Speedups of LinSolv on the Beowulf

computation costs.

The Eden measurements show that the workpool parallelisation achieves good performance and scales up well. When there are fewer homomorphic images needed than processors available (for 9×9 this is around 24 PEs) the runtimes of the version with unlimited speculation can increase, as too many speculative tasks are distributed. From Figure 1.3 we can see that the speedups for the 9×9 matrix start to even out for 10 and more processors, whereas the larger 14×14 matrix produces sufficient parallelism for larger numbers of processors. We have developed another parallel Eden version which scales up much better for increasing numbers of processor elements, as it exploits additionally a parallel homomorphic vector computation. For the number of processor elements considered in Figure 1.3 its performance is similar to the version with unlimited speculation, and we omit its results here. As in GPH, the larger sparse problem delivers the best results with an absolute speedup of 13 (8.8 for the smaller problem) on 16 PEs. We believe the improved performance is due to the greater amount of total work which induces coarser-grained tasks, and ameliorates startup and finalisation costs.

1.3.3 Raytracer

Algorithm. The Raytracer program calculates a 2D image of a given scene of 3D objects by tracing all rays in a given grid, or window. Since the computations for all pixels are independent of each other the top level code, consisting of two maps over the rows and columns of the window, can be parallelised by using two *parallel maps*. In practice the picture sizes are about several hundred rows and

```
ray :: Int -> Int -> [Sphere] -> [[String]]
ray x y scene = shuffleN outps
    where outps = [ (process i -> f_dm i) # void | i <- [0..np-1]]
                        'using' spine
          f_dm n _  = map outer_f (takeEach np (drop n sizes_y))
          outer_f i = map (\ j -> (rgb (tracepixel scene i j))) sizes_x
```

FIGURE 1.4. Parallel direct-mapping version of the Raytracer (Eden version)

columns, creating hundreds of threads for the outer `map` alone. In order to achieve good granularity in the inner loop, sequences of pixels are collected into chunks and processed together.

In order to minimize the communication overhead, the processes can initially be given all necessary data, like the scene and the viewpoint, to select the lines to be computed by themselves. We call this version *direct-mapping*. If `np` processors are available, `np` processes are created. Process `i` ($0 \leq$ `i` $\leq$ `np-1`) then computes all result lines `i+j*np` with `j` ≥ 0. In this version no dynamic distribution of tasks is required after the startup phase. Even more importantly, the tasks are provided with all necessary data at startup and hence subsequently perform minimal communication. Provided that the amount of work does not vary too much between the tasks, this should yield an even work distribution.

Implementation. In GPH we have implemented the parallel map version. The Eden implementation of the direct-mapping version is shown in Figure 1.4, with `f_dm` representing the work to be executed by one processor and the function `outer_f` computing one line. The code uses two auxiliary functions: `takeEach`, which extracts each *n*-th element from a given list; and `shuffleN` which merges a list of lists, element per element in a round-robin fashion. Note that the following equation holds:
`xs==shuffleN [takeEach np (drop i xs) | i<-[0..np-1]]`.

The processes are created in a list of process instantiations. The sequential strategy `spine`[5] is used to drive an eager process creation, i.e. to create the processes before the outport values are needed.

Performance Results. For the measurements we used images with 350×350 pixels (sequential runtime: 213.42s) and 500×500 pixels (sequential runtime: 435.01s). Figure 1.5 shows the absolute speedups for the GPH and Eden implementations of the Raytracer. The Eden version achieves the best speedups with the direct-mapping version. The reconstruction of the picture from the list of lists delivered by each worker in this version takes only about 4*s* (for a 350×350 picture) and about 8*s* (for a 500×500 picture). The discontinuity of the speedups between two and three PEs is due to the following reason: When there are three or more PEs, the Eden skeleton reserves one PE to reconstruct the picture, while the remaining PEs perform the raytracing. The only sources of overhead in the par-

[5]In evaluation strategies notation: `spine = seqList r0`

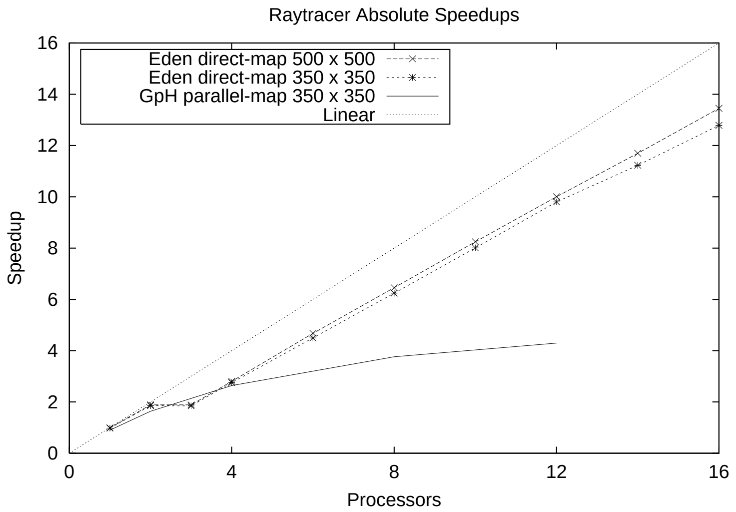

FIGURE 1.5. Absolute Speedups for the Raytracer

allel version are the partial recomputation of the lines (this should be marginal), the time needed for restructuring the result, and possibly some bottleneck in the communication network. The program yields an absolute speedup of 12.8 and 13.4 on 16 PEs for the smaller and larger window size, respectively. For larger numbers of PEs (not shown here) the maximal speedup is 27 on 37 PEs for the larger window size. This is significantly better than the results achieved for a parallel map version under GPH. In this case we achieved an absolute speedup of only 4.3 on 12 PEs. In this version parallel performance is severely hampered by the generation of several thousand threads.

Overall, the better parallel performance of Eden in Figure 1.5 can be attributed to two features, one of the algorithm and one of the language. Firstly, the parallel map version is both finer-grained and requires more communication than the direct-mapping version. Because of the roughly balanced load between the threads in the direct-mapping version, the potential benefit of evening out an imbalanced load in the parallel map version does not apply. Secondly, the Eden version guarantees that all input data of a remote task is fully evaluated. This avoids the sharing of input data between parallel threads, which can lead to a large amount of communication. In the GPH version it is more difficult to achieve a good data distribution.

1.4 CONCLUSIONS

We have used two realistic benchmarks to compare two contrasting parallel functional languages on a Beowulf cluster. Eden uses explicit process notations, whereas GPH uses a more implicit annotation-based approach. Our (preliminary) results show that the Eden implementation gives good absolute speedups of up to 13 on 16 processors for LinSolv and 13.5 on 16 processors for a direct-mapping Raytracer, whereas the GPH implementation achieves only a maximum speedup of 7.5 on 12 processors for LinSolv and 4.3 for a parallel-map Raytracer.

There are several explanations for the performance variations between Eden and GPH. Firstly, the Raytracer has a fairly regular and static **process structure**. This can be readily mapped either directly to a small number of explicit Eden processes, or to a similar number of GPH threads using an appropriate evaluation strategy. The more implicit approach favoured by GPH does, however, encourage additional sub-expressions to be annotated, as we have done here. In this case, these annotations yield excessively fine grained threads. Similarly, the more deeply nested irregular parallelism in LinSolv, which seems better suited to the GPH approach, cannot be exploited effectively in this setting because the granularity of the inner threads is too fine. Secondly, recent experiments have shown that our implementation of **global memory management** in GPH [Loi00] can carry high overheads. We are currently working to reduce this overhead. Thirdly, the GPH implementation may be performing **excess communication**, for example in fetching data lazily when eager data fetching would be possible.

Comparing the programming styles that have been used in the GPH and Eden versions of the benchmark programs, the most striking difference is that GPH makes it easy to parallelise functional programs by annotating the algorithmic code by parallel control structures, while Eden encourages designing a program around a certain parallel process structure. As a result, it is easy to obtain *some* parallelism from an arbitrary program using GPH with appropriate evaluation strategies. In particular, any improvement of data locality in the algorithm has to be done indirectly in the GPH version. When tuning the performance of the program, however, it can be easier to improve speedups using a skeleton-based Eden version of the program.

We are encouraged by the Eden results given here, which show that good *absolute* parallel speedups can be achieved using functional programming languages even for very high-latency distributed machines such as Beowulf clusters, and are motivated to improve GPH performance to match this, if possible.

ACKNOWLEDGEMENTS

The authors would like to thank the following organisations for funding the work presented here: the Austrian Academy of Sciences (APART 624), UK's Engineering and Physical Sciences Research Council (grants EPSRC GR/L 93379 and EPSRC GR/M 32351), the British Council and the DAAD (grant 1097). We also would like to express our thanks to Ricardo Peña Mari, Robert Pointon, Steffen

Priebe, Álvaro J. Rebón Portillo, Fernando Rubio Diez, and the anonymous referees for helping us to improve the quality of the paper.

REFERENCES

[BKL98] S. Breitinger, U. Klusik, and R. Loogen. From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View. In *Programming Languages: Implementations, Logics and Programs (PLILP'98)*, LNCS, Vol. 1490, pages 318–334, Pisa, Italy, September 1998. Springer-Verlag.

[BLOP97] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña Mari. The Eden Coordination Model for Distributed Memory Systems. In *Workshop on High-level Parallel Programming Models (HIPS'97)*, pages 120–124, Geneva, Switzerland, April 1997. IEEE Computer Science Press.

[HM99] K. Hammond and G. Michaelson. *Research Directions in Parallel Functional Programming*. Springer-Verlag, 1999.

[HR99] K. Hammond and A.J. Rebon Portillo. HaskSkel: Algorithmic Skeletons for Haskell. In *Intl. Workshop on the Impl. of Funct. Langs. (IFL'99)*, LNCS, Vol. 1868, pages 181–198, Lochem, The Netherlands, Sep. 1999. Springer-Verlag.

[KLPR00] U. Klusik, R. Loogen, S. Priebe, and F. Rubio Diez. Process Schemes in Eden — Efficient Parallel Programming With Low Effort. In *Intl. Workshop on the Impl. of Funct. Langs. (IFL'00)*, Aachener Informatik-Berichte No. 00-7, 2000. Draft Proc.

[Klu00] U. Klusik. *An Efficient Implementation of the Parallel Functional Language Eden on Distributed-Memory System*. PhD thesis, University of Marburg, November 2000. In preparation.

[Les93] B.P. Lester. *The Art of Parallel Programming*. Prentice Hall, 1993.

[Loi97] H-W. Loidl. LinSolv: a Case Study in Strategic Parallelism. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, Sep., 1997.

[Loi00] H-W. Loidl. Investigating the Memory Management in a Parallel Graph Reducer. In *Intl. Workshop on the Impl. of Funct. Langs. (IFL'00)*, Aachener Informatik-Berichte No. 00-7. RWTH Aachen, Sep. 2000. Draft Proc.

[LTH+99] H-W. Loidl, P.W. Trinder, K. Hammond, S.B. Junaidu, R.G. Morgan, and S.L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11:701–752, 1999.

[MPI94] MPI: A Message-Passing Interface Standard. *Intl. Journal of Supercomputing Applications*, 8(3/4), 1994.

[PH97] J.C. Peterson and K. Hammond, editors. *Report on the Non-Strict Functional Language, Haskell, Version 1.4*. April 1997.

[RBMS97] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *IEEE Aerospace*, 1997.

[THLP98] P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. Functional Programming*, 8(1):23–60, Jan. 1998.

[THM+96] P.W. Trinder, K. Hammond, J.S. Mattson, A.S. Partridge, and S.L. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Prog. Lang. Design and Implementation (PLDI'96)*, pages 78–88. ACM, May 1996.