Higher-Order and Symbolic Computation, 16, 203–251, 2003
© 2003 Kluwer Academic Publishers. Manufactured in The Netherlands.

Comparing Parallel Functional Languages: Programming and Performance*

H.-W. LOIDL hwloidl@macs.hw.ac.uk
School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh EH14 4AS, Scotland

F. RUBIO fernando@sip.ucm.es
Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 28040 Madrid, Spain

N. SCAIFE norman@jaist.ac.jp
Japan Advanced Institute for Science and Technology, 1/8 Asahidai, Tatsunokuchi, Nomigun, Ishikawa, 923-1211

K. HAMMOND kh@dcs.st-and.ac.uk
School of Computer Science, University of St. Andrews, KY16 9SS, Scotland

S. HORIGUCHI hori@jaist.ac.jp
Japan Advanced Institute for Science and Technology, 1/8 Asahidai, Tatsunokuchi, Nomigun, Ishikawa, 923-1211

U. KLUSIK klusik@mathematik.uni-marburg.de
R. LOOGEN loogen@mathematik.uni-marburg.de
Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, D-35032 Marburg, Germany

G.J. MICHAELSON greg@macs.hw.ac.uk
School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh EH14 4AS, Scotland

R. PEÑA ricardo@sip.ucm.es
Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 28040 Madrid, Spain

S. PRIEBE priebe@mathematik.uni-marburg.de
Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, D-35032 Marburg, Germany

Á.J. REBÓN alvaro@dcs.st-and.ac.uk
School of Computer Science, University of St. Andrews, KY16 9SS, Scotland

P.W. TRINDER trinder@macs.hw.ac.uk
School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh EH14 4AS, Scotland

Abstract. This paper presents a practical evaluation and comparison of three state-of-the-art parallel functional languages. The evaluation is based on implementations of three typical symbolic computation programs, with performance measured on a Beowulf-class parallel architecture.

*This work is primarily supported by the Austrian Academy of Sciences (APART fellowship 624), the Japan Society for the Promotion of Science (Postdoctoral fellowship P00778), and UK's Engineering and Physical Sciences Research Council (grant nos. GR/L 93379, GR/M 32351 and GR/L 42889).

We assess three mature parallel functional languages: PMLS, a system for implicitly parallel execution of ML programs; GpH, a mainly implicit parallel extension of Haskell; and Eden, a more explicit parallel extension of Haskell designed for both distributed and parallel execution. While all three languages employ a completely implicit approach to communication, each language takes a different approach to specifying and controlling parallelism, ranging from explicit identification of processes as language constructs (Eden) through annotation of potential parallelism (GpH) to automatic detection of parallel skeletons in sequential code (PMLS).

We present detailed performance measurements of all three systems on a widely available parallel architecture: a Beowulf cluster of low-cost commodity workstations. We use three representative symbolic applications: a matrix multiplication algorithm, an exact linear system solver, and a simple ray-tracer. Our results show how moderate speedups can be achieved with little or no changes to the sequential code, and that parallel performance can be significantly improved even within our high-level model of parallel functional programming by controlling key aspects of the program such as load distribution and thread granularity.

Keywords: parallel computation, functional programming, skeletons, implicit parallelism, automatic task decomposition, load balancing, Haskell, ML

1. Introduction

The potential advantages of purely functional programming languages for prototyping and developing parallel programs have long been recognised [8]. The high level of programming abstraction simplifies the task of programming, fosters code reuse and facilitates the development of substantially architecture-independent programs. The absence of side-effects avoids the unnecessary serialisation which is a feature of most conventional programs. For a comprehensive discussion of these issues see [25].

Realising this potential in an effective manner has proved an elusive goal, however. Reducing or eliminating programmer control places considerable emphasis on sophisticated automatic systems for detecting and controlling parallelism, making such systems fairly rare and often only available on a few parallel architectures. A comparison of different implementations of such automatic resource management mechanisms, as presented in this paper, is even rarer—to our knowledge this is the first head-to-head performance comparison of several parallel functional languages on the same parallel architecture.

For this paper, five research groups have cooperated to produce the comparisons. We assess three parallel functional languages: Eden and GpH, both extensions of the standard non-strict functional language Haskell [67], and PMLS, a parallel implementation of the strict functional language ML [56]. The languages all have high-level coordination, represent a range of language and implementation alternatives and are three of the relatively few robust parallel functional language implementations available. Assessment is made on both language and performance levels. We compare the language features available to express parallel coordination, in particular we focus on how parallel tasks are identified and created. In the case of Eden [7], task identification and creation are explicit. In the case of GpH [79], potential parallelism is identified through new language primitives, with tasks created automatically during program execution on the basis of load. In the case of PMLS [55], parallel tasks are identified by automatically detecting instantiations of certain higher-order function templates, skeletons. On the performance level we use three representative symbolic applications that have also been widely studied in the general parallel programming community: a matrix multiplication algorithm, an exact linear system solver, and a simple ray-tracer.

The remainder of this paper is structured as follows: Section 2 discusses general concepts of parallel programming and their importance in the context of a functional language. Section 3 compares the three languages, separating the user-visible language constructs that are needed for expressing parallelism from the implementation of these constructs. Section 4 presents measurements of all three systems for the three example programs mentioned above. We discuss the ease of implementing these programs, the support for performance tuning, and the overall performance achieved on a 32-node Beowulf cluster. Section 5 relates our languages to other parallel functional programming languages. Finally, Section 6 concludes.

2. Parallel functional programming

2.1. *Why parallel functional programming?*

Parallel programming is inherently harder than sequential programming. Traditionally the programmer must not only describe *what* to compute, i.e. a correct algorithm, but also *how* to organise the subcomputations on the target architecture, i.e. effective parallel coordination. Contemporary functional languages have three key properties that make them attractive for parallel programming: they have powerful mechanisms for abstracting over both computation and coordination; they eliminate unnecessary dependencies; and their high-level coordination achieves a largely architecture-independent style of parallelism.

2.1.1. Abstraction. Functional languages have excellent abstraction mechanisms that can be applied to both computation and coordination [33]. Two important abstraction mechanisms are *function composition* and *higher-order functions*. Function composition allows complex problems to be decomposed into simpler sub-functions. Higher-order functions, ones that manipulate other functions, allow new control constructs to be defined as required. Through use of powerful mechanisms such as these, functional programs are typically much shorter than their imperative or object-oriented equivalents.

The principle of abstraction can be carried through to parallel programming, where higher-order functions may be used to form the basis of new parallel programming constructs. Typically, parallel functional programs will abstract over details such as process placement, the timing and volume of communication, and synchronisation issues. More effort can thus be devoted to improving parallel algorithms. High level abstraction of parallel constructs encourages experimentation with alternative parallelisations, which often leads to improved or novel solutions for parallel problems.

2.1.2. Elimination of unnecessary dependencies. The absence of side-effects makes it relatively straightforward to identify potential parallelism. Since the natural method of program construction is by composing functions to the depth required rather than by sequential composition, accidental sequential dependencies are not introduced into the source program. The only source of sequential dependency is that the arguments to a function must be evaluated before they can be used. That is, dependencies are identified solely on the basis of use. Since values do not change once they have been computed, dataflow analysis is not needed to determine usage patterns, even at an inter-procedural level.

2.1.3. Architecture-independence. Good parallel abstractions encourage high-level portability by abstracting over lower level issues. In extreme cases, this abstraction may hide all details of the parallel implementation yielding a model of implicit parallelism. As the low level issues often depend on properties of a specific architecture, a high-level approach is significantly less architecture-dependent than lower-level approaches. The architecture-independence is bought at the price of elaborate language processors: either the compiler or the runtime system or a combination of both must adapt the high-level parallelism for the underlying architecture. By using, at the runtime-system level, standards like PVM [70] or MPI [60], languages can abstract over architecture characteristics. Unlike imperative languages, functional languages enable a high degree of abstraction over such standards through higher order functions and polymorphism.

2.2. *Tasks, processes and threads*

Parallel programming involves the identification and creation of sub-tasks that collectively perform the overall task of the program. These sub-tasks must be allocated to (*placed on*) processors that will execute them in some order. Depending on the system, load balancing may occur by *migrating* sub-tasks between processors at execution time.

In this paper, we will distinguish two levels of parallel tasks: *processes*, relatively heavy-weight tasks whose behaviour is often revealed to the programmer; and *threads*, which are implicit, and which form part of a process.

2.2.1. Task identification and granularity. Tasks may be identified either explicitly by the programmer using some language construct, or implicitly by the system identifying potentially parallel parts of the program. In some cases, the identification may be assisted by the use of *annotations*: programmer instructions that may or may not be exploited by the language implementation. The *granularity*, i.e. the size of the computation, of tasks may thus be determined by the programmer, the compiler, the runtime system or a combination of these.

2.2.2. Task creation. Tasks may be created either statically at initialisation or dynamically during execution of the program. In the latter case, they may be created either immediately they are identified (*eager task creation*) or delayed until they are deemed to be required by the runtime system (*lazy task creation*). When a task is created, it is allocated resources that allow it to execute independently on some parallel processor. In some cases a task may return resources to the system while being suspended, i.e. while it waits for the availability of required data.

2.2.3. Task placement. When a task is created, it is placed on a processor that will execute it. This placement may either be on the basis of static information determined before execution by the compiler, or dynamically, perhaps in response to load information. Static placement usually gives a good balance for regular task structures, in cases where the communication pattern can be determined in advance. Dynamic placement is more appropriate in situations where the task structure is irregular, cannot be pre-determined, or where the structure changes during program execution.

2.2.4. Scheduling and load management. Scheduling is needed to manage the execution of multiple tasks on a single processor. Such scheduling may be required to be *fair*, i.e. guaranteed to execute every available thread eventually. Dynamic rebalancing of workload may also be required, especially for irregular task structures on high-latency systems. Rebalancing is usually achieved by *migrating* tasks, but alternatives are to employ *task subsumption*, in which smaller tasks are merged into larger ones, or to maintain a work-pool of potential tasks, which can be communicated between processors at lower cost than tasks which are already executing. Rebalancing may occur as a result of creating excess work on a single processor, or as a consequence of starvation on some processors, in which case a *task stealing* mechanism may be used.

2.3. Communication

Communication is fundamental to executing parallel tasks. In traditional parallel programming, communication is explicit: the programmer uses explicit message-passing primitives, or communicates through explicitly shared variables, which must normally be protected against concurrent modifications. In the more implicit approaches advocated here, communication occurs as a consequence of shared data dependencies between tasks. The systems use either message passing or shared-memory, as appropriate, and automatically protect the data against concurrent modifications, as required.

2.3.1. Code or data. Traditional parallel systems usually only support data transmission. In a functional setting, it is natural for functions to be transmitted between parallel tasks, and in a non-strict setting, this may extend to partially evaluated or completely unevaluated forms. Although this is no conceptual limitation, the parallel systems discussed here do not perform code migration. Only code pointers are transmitted, as the whole code is usually supposed to reside in all processors. This is sometimes characterised as an SPMD, single program multiple data, approach.

2.3.2. Push or pull. Data may be transmitted either on demand (a pull mechanism) or when produced (a push mechanism). Pulling has the advantage of transmitting only the data that is required, but pushing will require fewer packets to be communicated if most of the data that is transmitted is required, and will reduce the amount of synchronisation that is needed. In some cases, however, large data structures may be transmitted unnecessarily. This leads to *speculative work*, since not all of the data structure may be needed to compute the result value. The optimal approach is application-dependent, but in general a combination of push and pull appears to be ideal.

2.3.3. Communication topology. In the more implicit approaches the topology of processes changes dynamically in response to load balancing demands. In this case, the topology is transparent to the programmer, and it might differ between identical program executions. In more explicit approaches, the programmer can control the topology by connecting processes in the desired way. Topologies such as rings or tori can be explicitly programmed. In contrast to such dynamic approaches some systems use a static topology with the exact number of processes fixed at compile time. Such a static approach is common with libraries for parallel programming or skeletons (see Section 5.1). Note that we make no attempt at matching the topology of the architecture to the topology of the processes, since this would introduce an architecture-dependent aspect to program development.

2.3.4. Data marshalling. Sophisticated data marshalling techniques are employed to automatically pack complex data structures. In some cases, this marshalling extends to graphs as well as hierarchical data structures, and may involve the packing of unevaluated as well as fully evaluated forms (see Section 4.2.4).

2.3.5. Synchronisation. Most systems also employ implicit task synchronisation, when values produced by one task are required by another. A task that requires an uncomputed value may suspend execution awaiting delivery of that value. The task is resumed when the value becomes available. Unlike conventional language approaches, such synchronisation is entirely transparent to the functional programmer, and is handled internally by the runtime system. That is, no explicit communication is required, and no other action is required from the programmer.

3. Language comparison

This section compares the three parallel functional languages PMLS, GPH, and Eden. A comparison of a wider range of functional languages can be found at [54]. The three languages have been chosen for the following reasons. Firstly to be consistent with a high-level computation language we select languages with high-level coordination and exclude languages with imperative or low-level coordination. Secondly the languages represent a range of language designs, e.g. both eager and lazy languages, and with coordination ranging from almost entirely implicit (PMLS) to a language (Eden) in which processes can be manipulated by the programmer. Thirdly, the languages represent a range of implementation designs, e.g. both those with predominantly static coordination (PMLS) and those with predominantly dynamic coordination (GPH). Finally we have selected three of the relatively few robust parallel functional languages available.

In this section we introduce the underlying notions of skeleton-thread- and process-based approaches to parallelism, classify our languages, discuss the user-visible language constructs and the implementations of these languages.

3.1. Language

In this section we introduce the parallelism constructs in each language and compare them in terms of expressiveness and paradigm.

3.1.1. PMLS. Parallel ML with Skeletons (PMLS) is a parallelising compiler for the full purely functional subset of Standard ML, that realises parallelism in higher-order functions (HOFs) as algorithmic skeletons [55]. The PMLS system is based on a purist interpretation of the skeletons "credo", seeking to minimise programmer involvement in identifying and exploiting parallelism.

Skeleton-based approaches define a set of parallel templates or *skeletons* [11]. The programmer writes the program using these skeletons as appropriate. A parallelising compiler can then exploit the rules provided for each skeleton, in order to produce an efficient parallel implementation of the program on the target architecture.

From the functional programmer's perspective, a skeleton is simply a normal higher-order function. Each higher-order function is mapped to a different abstract parallel process topology, with parameters specifying details of the tasks that are to be performed.

Since the only parallel constructions that are available to the programmer are the higher-order functions that have been provided by the language, programmers must design parallel algorithms by adapting the sequential source to these functions. The compiler and runtime system are jointly responsible for setting up the corresponding process topologies, and for mapping processes to processors.

Higher-order functions may be given different behavioural interpretations when compiling for different target architectures. This allows a single higher-order functions to abstract over a range of possible parallel behaviours, which are selected on the basis of concrete details such as communication latency, or the granularity of the tasks to which the function is applied. In essence, skeletons modify behaviours but not values.

As an example Figure 1 shows an implementation of the common higher-order function `map` in PMLS. It applies the function `f` to all the elements of the list `(h::t)`. If `f` converts something of type `'a` to type `'b` then `map f` converts an `'a list` to a `'b list`. If we unfold `map f` across a list `[e1,e2...eN]`, the effect is the evaluation of `[f e1,f e2, ...,f eN]`. There is no interaction between the evaluation of each element, so in principle these evaluations may be carried out in arbitrary order, in particular in parallel.

A common approach to parallelising `map` is to construct a *task farm* skeleton consisting of a farmer processor controlling worker processors pre-loaded with `f`. Given an initial list, the farmer:

– records all workers as free;
– repeatedly:

  • sends an unprocessed list element to a free worker and records it as busy;
  • receives a processed list element from a busy worker and records it as free;

```
fun map f [] = [] |
    map f (h::t) = f h::map f t
val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

Figure 1. Parallel map in PMLS.

– until all list elements have been processed;
– assembles the processed list in the appropriate order.

This approach is self-balancing: no workers sit idle so long as there are more list elements to be processed, and variations in the times to process different elements have minimal impact.

There are various topologies for task farms, for example the linear chain where each processor has a bi-directional connection to its predecessor and successor. The farmer passes unprocessed data down the chain of busy workers to the first free worker, and processed data is passed back up the chain to the farmer. Here, the farmer need not keep track of free and busy workers, and may assemble the final list as processed elements become available.

3.1.1.1. Constructs. The PMLS compiler generates parallel code solely from calls to `map` and `fold`. No other SML constructs are provided or exploited for parallelism. However, the system enables the introduction of new higher-order functions with new skeletons. In some cases, like `fold`, a proof obligation is put on the programmer to ensure correctness of the parallel code: in the case of `fold` the binary operation must be associative.

While the compiler will support the judicious local use of imperative SML constructs, assignments to free variables in arguments to parallelised higher order functions no longer have global effects and defunctionalisation may not preserve evaluation order.

3.1.1.2. Methodology. The programmer need have no conception of parallelism. The compiler will exploit parallelism in explicit uses of `map` and `fold`.

A pre-processor may also be used to synthesise higher-order functions in programs that lack them, using proof planning driven by middle out reasoning [12]. For example Figure 2 shows how, given the function `inc`, this pre-processor can synthesise both `inc1`, defining `inc` in terms of `map`, and `inc2`, defining `inc` in terms of `fold`.

3.1.2. GPH. GPH [79] is a modest conservative extension of Haskell98 [67] realising a thread-based approach to parallelism. *Thread-based approaches* to parallelism allow parallel threads to be created, but do not provide mechanisms to control those threads. Threads are thus managed entirely under runtime-system control. By combining simple thread primitives with higher-order functions, high-level abstractions can be constructed, such as the evaluation strategy approach [79].

```
(* original function *)
fun inc [] = [] |
    inc (h::t) = h+1::inc t

(* first synthesised function, using map *)
val inc1 = map (fn h => h+1)

(* second synthesised function, using foldr *)
val inc2 = foldr (fn h => fn t => h+1::t) []
```

Figure 2. Program synthesis in PMLS.

```
-- basic constructs
par :: a -> b -> b              -- parallel composition
seq :: a -> b -> b              -- sequential composition

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

using :: a -> Strategy a -> a   -- strategy application
using x s = s x `seq` x

rwhnf :: Strategy a             -- reduction to weak head normal form
rwhnf x = x `seq` ()

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

Figure 3. Basic coordination constructs in GPH.

3.1.2.1. Constructs. GPH provides parallel (`par`) and sequential (`seq`) composition as coordination primitives (see Figure 3). Denotationally, both compositions are projections onto the second argument. Operationally `seq` 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.

Experience of implementing non-trivial programs in GPH shows that the unstructured use of `par` and `seq` 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 a Haskell expression. Evaluation strategies provide a clean separation between coordination and computation. The driving philosophy is that it should be possible to understand the computation specified by a function without considering its coordination. Figure 3 shows the basic operations over strategies. A strategy on a value of type `a` is a function from `a` to the nullary value `()` executed purely for effect, and the null value is returned to indicate completion. The `using` construct applies a strategy to a Haskell expression. The basic strategy `rwhnf` reduces an expression to weak head normal form (WHNF), the default in Haskell. The overloaded strategy `rnf` reduces an expression to normal form (NF), i.e. containing no reductions. As there are types that are not reduced to normal form in Haskell, e.g. function types, `rnf` is restricted to types that are reduced to normal form by the `NFData` class which is instantiated for all major types. Because strategies are simply functions, they can be combined or passed as parameters using standard language capabilities.

For example the `parList` strategy in Figure 4 is higher-order, applying the argument strategy `strat` to every element of a list in parallel. This strategy is then used in the GPH implementation of parallel map (`parMap`). Note how the algorithmic code is cleanly separated from the strategy, using the sequential code of `map f xs` unmodified when introducing parallelism.

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

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

Figure 4. Parallel map in GPH.

3.1.2.2. Methodology. GPH programs are developed with an integrated suite of sequential and parallel software tools, based on the Glasgow Haskell Compiler (GHC) [66]. The tools for sequential software development include: the Hugs interpreter, for fast development, the GHC compiler and sequential runtime system for optimising compilation to sequential code; and sequential time and space profilers integrated into GHC [75]. The tools for parallel software development include: the GRANSIM parameterisable parallel simulator [24] for flexible and accurate simulation of the parallel behaviour on a range of parallel machines; the GHC compiler and GUM parallel runtime system for parallel execution on multiprocessors; a set of visualisation tools for both GRANSIM and GUM, visualising the activity of a parallel machine in several levels of detail; prototypes of more detailed parallel profilers [38].

3.1.3. Eden. Eden [7] extends the lazy functional language Haskell by syntactic constructs to explicitly define and instantiate processes. In contrast to the previous techniques, *process-based approaches* like Eden expose parallel tasks at the language level. The programmer must then manage the tasks using the control mechanisms provided in the language. 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 programmer's level.

3.1.3.1. Constructs. Eden provides process abstractions and process instantiations for coordination as shown in Figure 5. The new expression `process x -> e` of a predefined

```
newtype Process a b = ...

-- process abstraction (language construct)
process x -> e :: Process a b

-- process instantiation
(#) :: (Transmissible a, Transmissible b) => Process a b  -> a -> b

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

Figure 5. Basic coordination constructs in Eden.

polymorphic type `Process a b` defines a *process abstraction* having formal parameter `x::a` as input and expression `e::b` as output. Process abstractions of type `Process a b` can be compared to functions of type `a -> b`, the main difference being that the former, when instantiated, are executed in parallel. Additionally, when the output or input expression is a tuple, a separate concurrent thread is created for the evaluation of each tuple element. We will refer to each of them as a channel.

A *process instantiation* is achieved by using the predefined infix operator `(#)`. The context `Transmissible` is needed to guarantee that the elements can be sent through the channels. Each time an expression `e1 # e2` is evaluated, a new process is created to evaluate the application of `e1` to `e2`. We will refer to the latter as the *child* process, and to the owner of the instantiation expression as the *parent* process. The instantiation semantics specifies in which processes these expressions shall be evaluated: (1) Expression `e1` together with its whole environment is *copied* in the current evaluation state to a new processor, and the child process is created there to evaluate the application of `e1` to `e2`, where `e2` must be remotely received. (2) Expression `e2` is eagerly evaluated in the parent process. The resulting full normal form data is communicated to the child process as its input argument.

Once processes are running, 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 full normal form and then transmitted. Processes trying to access input not yet available are temporarily suspended. This is the only synchronising mechanism in Eden.

Figure 6 presents a simple parallel `map` skeleton in Eden, in which a different process is created for every element of the input list. Strategies are used in Eden to influence the evaluation order. In this example, the `spine` strategy is used to eagerly evaluate the spine of the process instantiation list. In this way all processes are immediately created. More sophisticated parallel implementations of `map` have been developed in Eden [40, 41] and some will be discussed in Section 4.

3.1.3.2. Methodology. Like GPH, Eden is based on the Glasgow Haskell Compiler, and can use the same sequential profiling utilities. For parallel profiling Eden provides a simulator called Paradise [28] which is based on GRANSIM, so that tuning the performance of an Eden program is a similar process to that in GPH.

Parallel programming in Eden can be done by explicitly defining and instantiating a process topology. This would be equivalent to sequential functional programming with explicit

```
map_par :: (Transmissible a, Transmissible b) =>
           (a -> b) -> [a] -> [b]
map_par f xs = [pf # x | x <- xs] `using` spine
               where pf = process x -> f x

spine :: Strategy [b]
spine []     = ()
spine (_:xs) = spine xs
```

Figure 6. Parallel map in Eden.

recursion. Sometimes this is appropriate, but an experienced functional programmer will try to use higher-order functions, i.e. skeletons, as much as possible in order to reduce the amount of work and the possibility of making mistakes. In a complex application both methods may be simultaneously needed. The main difference between Eden and more traditional skeleton-based languages, such as PMLS, is the fact that skeletons can be specified within Eden itself. Thus, Eden serves both as a computation and coordination language, providing a high degree of flexibility for the programmer.

3.2. Implementation

In this section we compare the implementations of the languages on arbitrary parallel architectures.

3.2.1. PMLS. The PMLS approach is based on:

- maximising compile-time activity to minimise run-time overheads;
- configuring the virtual topology of the target system to reflect closely the hierarchy of higher-order functions in the program.

While this is relatively inflexible, for example making exploitation of parallelism across condition branches difficult, it often results in very efficient code.

3.2.1.1. Compile time. The PMLS compiler front end parses, elaborates and type checks SML to produce an abstract syntax tree. The ML Kit is used as the front end. The tree is traversed to extract an abstract network showing the nesting hierarchy of higher-order functions. Free variable lifting, or defunctionalisation, is performed to simplify passing free variable bindings to skeletons, and to avoid runtime transmission of closures. The abstract syntax tree and the abstract network are traversed to identify higher-order functions to be realised as skeletons and to generate skeleton network code and MPI registration in C. The resulting abstract syntax tree is translated into Objective Caml for linkage by the OCaml and GNU C compilers with the appropriate skeletons, and skeleton network and MPI registration code.

PMLS skeletons are written in C with MPI. The `map` function is implemented as a task farm and `fold` as a divide-and-conquer network. The skeletons are hybrid and may be run either in parallel or sequentially. Skeletons are coordinated at runtime by generic "Pskel" nodes, dynamically switching between these hybrid modes. Otherwise, skeletons are linked statically with no runtime change of topology. Adopting an SPMD approach, all processors are pre-loaded with all skeletons and functions.

The use of Objective Caml and GNU C to generate native code enables a high degree of portability. PMLS has been ported to a Fujitsu AP3000, IBM SP/2, Cray T3E, networks of UltraSparc workstations, SUN Enterprise and Beowulf clusters, displaying high performance across all platforms as shown in [76].

3.2.1.2. Run time. PMLS generates code to link static skeletons through Pskel nodes. The Objective Caml run-time environment provides garbage collection and appropriate libraries.

At run-time, the Pskel nodes at each level determine their behaviour from the skeleton network. In particular, intermediate Pskel nodes in the hierarchy will switch between parent and child operation if initiated in parallel mode. There is no movement of code or closures at runtime. For further details of the compiler design and implementation see [55].

The single processor efficiency, i.e. the sequential runtime divided by the parallel runtime on 1 processor in percent, of PMLS has been measured as 86% on our Beowulf cluster. The main sources of overhead are slight inefficiencies introduced in program transformation stages, such as extra function calls, and the need to propagate additional information that is used as arguments to the skeletons used for exploiting the parallelism. In a multi-processor setup the worker nodes of the task-farm skeleton used in our measurements exhibit an efficiency of 84%. In this case the main source of overhead is some idle time introduced by blocking communications between nodes in this skeleton. An implementation of a more efficient version, using non-blocking communication wherever possible, is currently in development.

Early versions of PMLS were hampered by inefficiencies in the translation process from SML to Objective Caml. More recent versions employ a set of optimising transformations, allowing fairly similar performance between the output from PMLS and hand-coded Objective Caml. The highest discrepancy in single-processor runtimes we observed in our measurements is 241s (PMLS) versus 195s (OCaml). A slowdown of around 20% is acceptable and is attributable to the remaining inefficiencies in the translation process.

3.2.2. GPH.

3.2.2.1. Compile time. The two additional language constructs of GPH, `par` and `seq`, are treated as built-in functions by the compiler. They are implemented as system-calls in the GUM runtime-system. GPH programs are compiled using almost all of the sequential optimisations in GHC, although care must be taken to preserve `par` and `seq`.

3.2.2.2. Run time. The GUM runtime-system for GPH realises a parallel graph-reduction machine built on top of GHC's sequential STG-machine. To synchronise multiple threads, a thread locks the node of the graph when starting its evaluation, and other threads requesting that data will be added to a blocking queue attached to the locked closure. Access to remote closures is managed by new *FetchMe* nodes, i.e. global indirection nodes. On requesting the contents of such a node a message will be sent to the target processor and the requesting thread will be added to a blocking queue. The details of these synchronisation and communication mechanisms are discussed in [47, Chapter 2].

Being integrated into GHC, GUM makes use of existing analyses and optimisations for efficient sequential execution. A discussion of the design and implementation of GUM is given in [80]. In summary, the additional features to enable parallel execution are:

- sparking of threads, i.e. identified program expressions may be evaluated as independent threads or they may be inlined by other threads, achieving dynamic granularity control as in the lazy task creation mechanism [59];
- multi-threading, i.e. independent threads of control are executed in an interleaved fashion thereby enabling an overlap of computation and communication on each processor;

- virtual shared heap, i.e. the physically distributed heap is treated as a shared heap with global pointers to remote processors, with transparent communication on access of non-local data;
- automatic marshalling of data and work, i.e. when data or work is needed on another processor, a graph structure is automatically serialised, sent to another processor, and unpacked into a graph;
- distributed garbage collection, i.e. weighted reference counting is used to garbage collect global pointers that are not used any more.

In order to assess the overheads of the different systems we have measured key parameters of the runtime-system. One important parameter is the efficiency on 1 processing element (*PE*), i.e. a processor with local memory, measured as sequential runtime divided by 1 PE runtime. For GUM we have previously measured 80%–93% on simple programs [80], and now at least 77% for the programs used here. In a multi-processor execution it turns out that maintaining a virtual shared heap on a distributed memory machine is most expensive. In particular the management of a hash table mapping local heap addresses to global heap address accounts for up to 3.8% of the total execution time, in an earlier version, pre-dating recent improvements in GUM even up to 8%. In comparison, the costs for packing graph structures and communication play only a minor role in the total runtime: less than 1%. The costs for creating parallelism are, by design, very small: creating a spark requires only adding a pointer to an array, and threads are very light (14 bytes for the thread descriptor) with initially small, tunable stacks (1 kB).

A detailed discussion of these overheads in GPH is presented in [49]. This paper separates the overhead into that induced by the thread management, memory management and communication subsystems of GUM. It then focuses on virtual shared memory management, which turns out to be the most expensive part. Several improvements of the basic load balancing mechanism, that we exploit in these measurements, are presented in [48].

3.2.3. Eden.

3.2.3.1. Compile time. Eden extends the optimising Glasgow Haskell Compiler with a few modifications. In Eden, lazy evaluation is changed to eager evaluation in two cases. Firstly, processes are eagerly instantiated when the expression under evaluation demands the creation of a closure of the form `o = e1 # e2`. Secondly, instantiated processes eagerly produce their output expressions and communicate them on channels. These modifications of the standard Haskell semantics are aimed at increasing the degree of parallelism and at speeding up the distribution of the computation, and they are implemented by automatic compile-time transformations. The new expressions provided by Eden, i.e. process abstractions, process instantiations, dynamic channels and `merge` instantiations, are translated into runtime-system calls.

3.2.3.2. Run time. The design of DREAM [6], the parallel abstract graph-reduction machine implementing Eden, is largely similar to GUM. We focus on the differences to GUM:

- In DREAM, the concept of a virtual shared heap does not exist. Each process evaluates its outputs autonomously with respect to other processes. The entire graph needed by a newly instantiated process is copied into its heap *before* it starts running. While this may lead to some duplication of work it reduces the communication overhead of DREAM. Moreover, global garbage collection reduces to the sending of terminate messages to processes whose output has been detected to be garbage during a local garbage collection.
- In contrast to GPH, Eden threads are mandatory. Processes in DREAM and threads in GUM are related as follows: A DREAM process is implemented by several threads, which directly correspond to threads in GUM. These threads run concurrently on the same processor, so that different output values can be independently produced. Threads synchronise on shared graph nodes as in GPH. Special *QueueMe* closures represent input from remote processes which is not available yet. On requesting the contents of such a closure a thread will be blocked until the input arrives.
- Process placement in Eden is controlled by the runtime-system in two different modes that can be set-up at the beginning of the execution: (1) round-robin mode, in which processes are instantiated in consecutively numbered processors, or (2) random mode, where processes are instantiated in randomly chosen processors.

As Eden shares parts of GUM's thread management and communication subsystem, the runtime-system overheads are similar. However, Eden overheads are smaller, as it is not necessary to maintain a virtual shared graph. The single processor performance of the most memory intensive test program is 89%. In general, the main bottlenecks in Eden are due to the packing and unpacking routines, which are not yet optimised. For instance, packing a 600×600 matrix of integers takes 1% of the time required for multiplying it. Moreover, as Eden does not provide multicasting, it is not possible to send the same packet to several processors and pay the packing overhead only once. See [73] for a more detailed description of Eden overheads.

3.3. Summary

Table 1 summarises the language and implementation features of PMLS, GPH, and Eden. On the language level it shows the higher level of abstraction for PMLS, using a skeleton-based approach, which does not require language extensions for parallelism at all, whereas GPH adds combinators to expose parallelism and Eden adds a construct for explicit process creation. On the implementation level PMLS performs sophisticated static analysis and program synthesis in order to generate a sufficient amount of parallelism. Both GPH and Eden rely mostly on a sophisticated runtime-system with dynamic resource management.

To achieve good single processor performance all systems use state-of-the-art sequential compilers for functional languages: GPH and Eden use GHC, and PMLS uses OCaml. In our benchmarks we achieve single processor efficiencies of 77% for GPH, mainly due to using a two-space garbage collector in the current implementation, 89% for Eden (using a better garbage collector), and 84% for PMLS which uses a two-generation garbage collector. We have assessed the overheads in the multi-processor executions for all three languages. The most costly components are in GPH the maintenance of hash tables in the virtual shared

Table 1. Language comparison.

	PMLS	GPH	Eden
Language			
Approach	Skeleton-based	Thread-based	Process-based
Constructs	HOFs	`par/seq`	Proc. abstraction Proc. instantiation
Programming Abstraction	–	Evaluation Strategies	Skeletons
Methodology	–	Simulate, execute, visualise	Define topology and/or skeletons, simulation
Implementation			
Compile-time support	Synthesise HOFs, process network, link skeletons	–	Force strict evaluation of channel data
Run-time support	Skeleton library over distributed heap	Graph-red. over shared heap	Graph-red. over distributed heap

memory management, and in PMLS the usage of blocking communication at certain stages and the single-master, multiple-worker parallel model. The details of these runtime-system measurements for GPH and PMLS, including data obtained from Beowulf and SunSMP machines, will be published in a separate paper [51].

4. Experimental results

This section describes the results we have obtained using three programs: `matMult`, a matrix multiplication algorithm, `linSolv`, an exact linear system solver, and `raytracer`, a simple ray tracer. The parallel algorithms themselves have been explained in more detail in previous papers [46, 53]. In this section we focus on a comparison of the implementations in and the performances achieved with PMLS, GPH, and Eden.

Although rather simple in nature, these programs represent a range of applications we are interested in. In previous studies on developing parallel applications in GPH [53] we have identified the class of symbolic applications, with complex data structures and irregular parallelism, as the most interesting application domain. For pragmatic reasons we had to keep the program size down: ensuring that all three versions implement the same algorithm and produce comparable dynamic structures was the main engineering constraint. Of the three programs in this section the linear system solver, with its multiple homomorphic images approach, fits these characteristics best, with the other programs focusing on different aspects of the execution.

More specifically, matrix multiplication is a well-studied parallel program and serves to relate our approach to that of imperative languages (with concrete language and performance comparisons in Section 6). The linear equation solver exhibits a structure typical for a class of symbolic applications, which is quite different from conventional iteration-based techniques. Since it performs a high amount of heap consumption and creates irregular parallelism, it is closest to typical symbolic applications. The ray tracer is an example of a data-parallel application, and issues of task and computation granularity become important in this context.

4.1. *Experimental framework*

All measurements have been performed on a 32-node Beowulf cluster [72] at Heriot-Watt University, consisting of Linux RedHat 6.2 workstations with a 533 MHz Celeron processor, 128 kB cache, 128 MB of DRAM and a 5.7 GB local IDE disk. The workstations are connected through a 100 Mb/s fast Ethernet switch with a latency of 142 μs, measured under PVM 3.4.2.

4.2. *Matrix multiplication*

4.2.1. Problem description. Given two square matrices of arbitrary precision integers $A, B \in \mathbb{Z}^{n\times n}$, $n \in \mathbb{N}$ find their product, i.e. a matrix $C \in \mathbb{Z}^{n\times n}$ such that $C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j}$.

4.2.2. Parallel algorithms. We start with a sequential algorithm directly implementing the above specification of matrix multiplication, shown in Figure 7. By using an algebraic datatype `Matrix a` to represent matrices as lists of lists we can overload standard arithmetic operations such as multiplication. The main function is `multMatT`, which takes A and B^T, i.e. the transposed matrix B as input. It computes $A*B$ in a double nested list comprehension, computing the rows of the result matrix in the outer comprehension and the elements of a row in the inner comprehension. The function `multVec` computes the sum in the specification above for two vectors of length n.

4.2.2.1. Naive data parallelism. Since each element of the result matrix can be computed independently, we can exploit data parallelism by generating one parallel task for each

```
data (Num a) => Matrix a = M [[a]]

multMat :: (Num a) => Matrix a -> Matrix a -> Matrix a
multMat (M m1) (M m2) = M (multMatT m1 (transpose m2))

multMatT :: (Num a) => [[a]] -> [[a]] -> [[a]]
multMatT m1 m2T = [ [multVec row col | col <- m2T] | row <- m1]

multVec :: (Num a) => [a] -> [a] -> a
multVec v1 v2 = sum (zipWith (*) v1 v2)
```

Figure 7. Sequential `matMult` (Haskell version).

element in the result matrix. However, the excessive number of small computations leads to a very poor performance in general. For example, the GPH implementation of this *naive data parallel* version yields no speedup on up to 16 processors. We do not consider this version any further.

4.2.2.2. Row clustering. The granularity of the naive parallel algorithm can be increased by computing an entire row of the result matrix by one task. Assuming square matrices of size $n \times n$ with integers of average size l in its internal representation, and assuming that integers are multiplied by using the algorithm of Karatsuba and Ofman [35], the computational complexity for each task is $O(n^2 * l^{\log_2 3})$, while the total communication complexity, i.e. the amount of data (in machine words) to be sent, is $O(n^3 * l)$. The latter complexity is due to the fact that each task requires the whole second matrix to compute one final row, and n tasks are created. In order to effectively improve parallel performance, the granularity of the tasks has to be increased by computing as many elements as possible inside each task and the communication has to be minimised.

We can improve the granularity further by computing many rows of the resulting matrix by each task. With perfect load distribution, if p processors are available, p tasks should be created, each one evaluating n/p rows of the resulting matrix. Using such a *row clustering* approach the communication complexity of the main process is $O(n^2 * p * l)$ whereas the computational complexity of each process is $O(n^3 * l^{\log_2 3}/p)$. For large values of n better speedups can be expected, since the computation-communication ratio increases.

4.2.2.3. Block clustering. An alternative form of clustering the data is to partition the input matrices into blocks, performing *block-clustering*, and to perform the basic arithmetic over these blocks rather than over integer values. Figure 8 depicts this partitioning, and indicates that for the computation of one block in the result matrix, only one row of the partitioned matrix A and one column of the partitioned matrix B is needed. In this version the computational complexity of each process is still $O(n^3 * l^{\log_2 3}/p)$ but its communication complexity is only $O(n^2 * l/\sqrt{p})$ as the processors do not require the whole second matrix.

4.2.2.4. Torus topology. All parallel versions so far rely on a broadcast of all data at the beginning of the computation with a communication complexity of $O(n^2 * l * \sqrt{p})$. Therefore, the main process tends to become a bottleneck especially for large numbers of processors. To avoid such a bottleneck we can use a *torus topology* as depicted in Figure 9. Initially each process in the torus receives only its own blocks from matrices A and B.

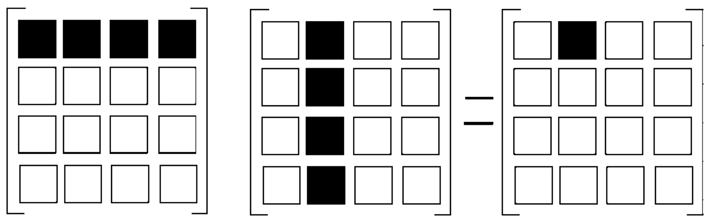

Figure 8. Structure of block-clustering `matMult`.

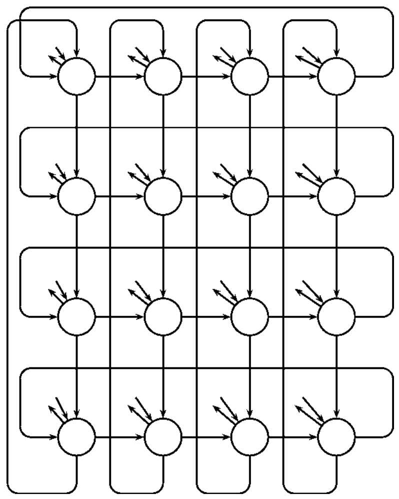

Figure 9. Process topology generated using a torus.

In each step the processor computes the product of both blocks, adds the product to the intermediate result computed so far, and then obtains the next blocks from its neighbours. As shown in Figure 9 the blocks of the first matrix are transmitted from left to right in the torus, while those of the second matrix are transmitted top down. This algorithm is well-known in the literature as Gentleman's algorithm [71]. In this version the communication complexity of the main process is O($n^2 * l$), i.e. it does not depend on the number of processors, and the communication between the processors is spread over the entire execution of the program. The main drawback of this approach is that it requires a perfect square number of processes to form a torus topology.

4.2.3. Implementations.

4.2.3.1. Eden. The *row-clustering* version in Eden creates as many processes as processors available with each of them computing $\frac{n}{p}$ rows of the product matrix. This version, as shown in Figure 10, uses the built-in variable `noPe`, representing the number of available processors. The function `splitIntoN n xs` splits the list `xs` into `n` nearly equal size sublists (see Appendix A for the definition of `splitIntoN` and other auxiliary functions used in this section).

```
multMatTParRow :: (Num a,Transmissible a) =>
                  Matrix a -> Matrix a -> Matrix a
multMatTParRow (M m1) (M m2) = M (concat result)
 where result = map_par multMatT (zip (splitIntoN noPe m1) (repeat m2))
```

Figure 10. Row-clustering `matMult` (Eden version).

The *block-clustering* version in Figure 11 creates `size` × `size` processes, each of them computing a block of the product matrix. In order to reduce the total amount of communication, the typical value of `size` will be $\lfloor\sqrt{\texttt{noPe}}\rfloor$. The main difference to the row-clustering version is the way in which the matrices are split, which is encoded in the `clusterLeft` and `clusterRight` functions. The first function splits matrix A into a list of rows, the second function splits matrix B into a list of columns.

The *torus version* of the algorithm can be expressed in Eden in terms of its general torus skeleton [64]. The main argument of the torus skeleton is the function to be performed by each node in the torus topology (see Figure 9). Each node has three input parameters: one from the parent; one from the left neighbour; and one from the top neighbour. It produces three values: one to the parent; one to the right neighbour; and one to the neighbour below.

With this torus skeleton, the matrix multiplication algorithm `multMatPar` shown in Figure 12 takes the size of the torus, `torusSize`, splits the matrices `m1` and `m2` into blocks `m1ss` and `m2ss`, respectively, thereby pairing them appropriately, and calls the torus skeleton

```
multMatTParBlock :: (Num a, Transmissible a) =>
                    Int -> Matrix a -> Matrix a -> Matrix a
multMatTParBlock size m1 m2 = decluster size result
  where result = map_par multMatT (zip (clusterLeft size m1)
                                       (clusterRight size m2))
```

Figure 11. Block-clustering `matMult` (Eden version).

```
torus :: (Transmissible a,Transmissible b,Transmissible c,Transmissible d) =>
             ((c,a,b)->(d,a,b)) -- Main function in each process
             [[c]] ->           -- Inputs from parent to children
             [[d]]              -- Outputs from children to parent
torus f m = ...

multMatPar :: (Num a, Transmissible a) =>
             Int -> Matrix a -> Matrix a -> Matrix a
multMatPar torusSize m1 m2 = combine results
 where results = torus (multMatPar' torusSize) (zipWith zip m1ss m2ss)
       m1ss    = splitMatrix1 torusSize m1
       m2ss    = splitMatrix2 torusSize m2

-- Function performed by each worker
multMatPar' :: (Num a, Transmissible a) => Int ->
              ((Matrix a, Matrix a), [Matrix a], [Matrix a]) ->
              (Matrix a, [Matrix a], [Matrix a])
multMatPar' size ((sm1,sm2),sm1s,sm2s) = (result,toRight,toBottom)
  where toRight  = take (size-1) (sm1:sm1s)
        toBottom = take (size-1) (sm2':sm2s)
        sm2'     = transpose sm2
        sms      = zipWith (curry multMat2) (sm1:sm1s) (sm2':sm2s)
        result   = foldl1' addMatrices sms
```

Figure 12. Torus version of `matMult` in Eden.

```
multMatPar :: (Num a, NFData a) => Int -> Matrix a -> Matrix a -> Matrix a
multMatPar z m1 m2 = multMat m1 m2
                     `using` \ (M m) -> rnf m1 `seq`
                                       rnf m2 `seq`
                                       parListChunk z rnf m
```

Figure 13. Row-clustering `matMult` (GPH version).

`torus` with the function `multMatPar'` to be applied by the node processes of the torus. The per-node function performs a list of matrix multiplications `sms`—one for each pair of blocks it receives—and sums all products to obtain the `result` which is returned to the parent. Note that the first pair, `(sm1,sm2)`, is received directly from the parent, whereas the other pairs are received from the left and right neighbours as part of `sm1s` and `sm2s`, respectively.

4.2.3.2. GPH. Figure 13 shows a *row-clustering* version of `multMatPar` in GPH. This version uses the sequential matrix multiplication, `multMat`, as shown in Figure 7 without change. All parallelism is defined by a strategy attached to `multMat`. The strategy part of the code first evaluates both input matrices, in order to avoid competition for unevaluated data during the evaluation, and then uses the predefined strategy `parListChunk z rnf m` to fully evaluate chunks of `z` elements in the matrix `m` in parallel.

The *block-clustering* GPH version in Figure 14 implements the algorithm sketched in Figure 8. In contrast to the purely strategic row-clustering version, it uses explicit functions for clustering and declustering the input and result matrices. Note that the code used to multiply the clustered matrices, `multMatT`, is the sequential matrix multiplication overloaded to work on matrices of matrices. The strategy attached to the clustered result matrix guarantees that every block in the clustered result matrix is evaluated in parallel. Such separation of data-layout from computation and reuse of sequential code greatly improves the productivity in our languages, and is in contrast to low-level C-based block-clusterings, where extensive code restructuring is needed to obtain very efficient parallel programs [18].

Based on experiences with different cluster functions, we have developed a generic mechanism for clustering arbitrary user-defined data structures, using formal program transformation to derive data parallel code such as this from the sequential code [52].

4.2.3.3. PMLS. The PMLS implementation uses nested lists for representing matrices, and Objective Caml's arbitrary-precision integer arithmetic library for the operations over

```
multMatPar :: (Num a, NFData a) => Int -> Matrix a -> Matrix a -> Matrix a
multMatPar z m1 m2 =
  decluster (multMatT (cluster z m1) (cluster z (transposeMat m2))
             `using` \ (M m) -> rnf m1 `seq`
                               rnf m2 `seq`
                               parList (parList rnf) m )
```

Figure 14. Block-clustering `matMult` (GPH version).

```
(* vector product call *)
fun inner row col = multVec row col

(* inner map - parallel *)
fun outer BT row = map (inner row) BT

(* outer map - parallel *)
fun multMat A B = map (outer (transpose B)) A
```

Figure 15. Row-clustering `matMult` (PMLS version).

the matrix elements. There is no general overloading of the basic arithmetic functions for matrices as in Haskell.

A straightforward sequential SML algorithm, that uses `map` instances instead of Haskell's list comprehensions, is shown in Figure 15. Since this code uses one of the higher-order functions that is implemented as a parallel skeleton, it can be directly parallelised by the PMLS compiler resulting in a pair of nested `map` skeletons. The outermost `map` in `multMap` computes a list of matrix-vector products by mapping the matrix-vector operation, called `outer`, over the rows of matrix A. The `outer` function computes a list of dot products by mapping the `multVec` function over the columns of matrix B. Note that the entire matrix B is free in `multMat`. The compiler's free-value analysis phase detects this property and generates code to transmit B to the workers prior to running the outer farm.

The parallel `map` skeleton has clustering of data built into it. The clustering size is global to the whole program and set manually, at present. With a clustering parameter of one this algorithm corresponds to the *naive data parallel* version mentioned above. In non-nested mode, with clustering set to a suitable value, the behaviour is identical to the *row-clustering* version. In nested mode, with both `map` skeletons implemented in parallel, the matrix B is only transmitted to the intermediate processors.

Figure 16 shows an approximation of a block-clustering version. The blocks are generated by the `map`'s implicit clustering mechanism. Since PMLS does not provide a user-level mechanism for enforcing absolute placement of data, the quality of the code depends on the ratio of processors to blocks. The best results are achieved if the number of blocks is a multiple of the number of processors. This method is slightly less

```
(* Block map over outer product *)
fun BMmult (A,B) =
  let
    val rows = length A
    val outerAB = outer_product (A,transpose B)
    val AB = map Mdotprod outerAB
  in
    split rows AB
  end
```

Figure 16. Block-clustering `matMult` (PMLS version).

communications-bound than the row-clustering method since the entire matrix B is not transmitted to all the processors.

4.2.4. Performance results. The measurements presented in this section are based on two 200×200 matrices of arbitrary precision integers, none of which is larger than $2^{16} - 1$, i.e. one machine word. For the row- and block-clustering versions Eden uses as many blocks as processors, whereas GPH uses a chunk size of 40. For the row-clustered version PMLS uses blocks of 3 rows, while for the block-clustered version it uses blocks of size 40×40.

The results presented here will be related to the performance of parallel versions implemented in C with PVM and GMP in Section 4.2.5 and in the conclusions (Section 6.2).

Figure 17 summarises the runtimes and Figure 18 the speedups of all versions on our Beowulf cluster. The sequential performance of the strict language, PMLS, is noticeably better than that of the lazy languages, Eden and GPH, with variations of about 26% between the versions of the latter languages.

For all versions the performance tails off fairly early with an increasing number of PEs (processing elements). In general, this is due to the high ratio of communication to computation as elaborated in Section 4.2.2. In Eden the torus topology behaves better than the block clustering version, which in turn is better than row clustering. The torus version shows a small increase in performance even for large numbers of PEs. This is in contrast to e.g. the block-clustering GPH version, which shows good speedups up to 4 PEs but tails off after that. In PMLS the difference in performance between the simple row-clustered and the refined block-clustered version, due to reduced communication, is most pronounced. The amount of communication can be directly linked to the free occurrence of B in the row- (Figure 15) but not in the block-clustered version (Figure 16). Furthermore, PMLS uses a task farm skeleton, as presented in Section 3.1.1, for implementing `map` in parallel. This model achieves a good load balance but limits the scalability of the system

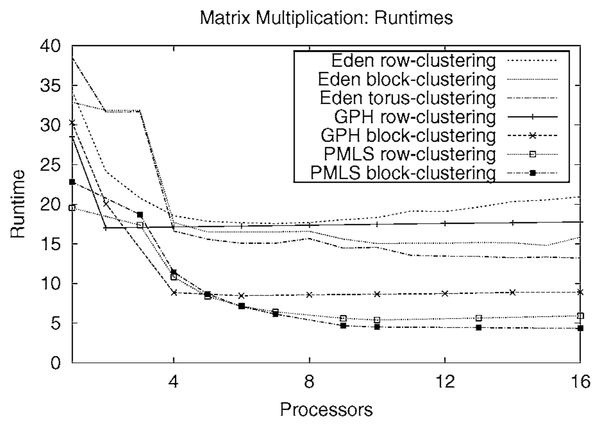

Figure 17. Runtimes of `matMult` on the Beowulf (in seconds).

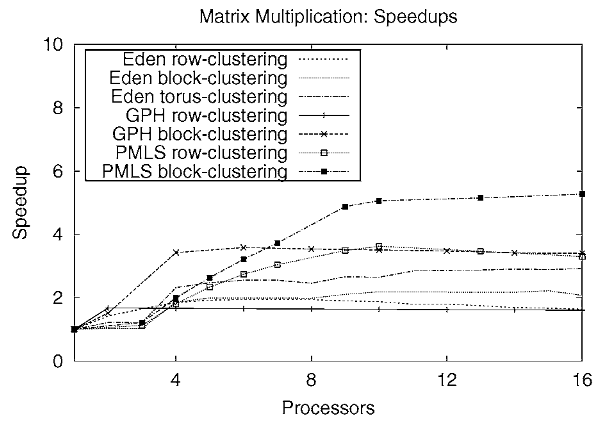

Figure 18. Speedups of `matMult` on the Beowulf.

because the master process becomes a communication bottleneck for large numbers of processors.

One important difference between the implementations of the three languages is the way that data items are packed in order to send them to other processors. In PMLS a generic serialisation routine is used, whereas GPH implements its own graph packing algorithm. As a result, the PMLS version is more portable, but the GPH and Eden versions are in general more efficient. Graph packing could be improved even further by developing specialised packing routines for commonly used data structures, such as lists, thereby reducing packet size and packing time. On a high-latency architecture such as the Beowulf and for communications-bound algorithms such as `matMult` this should yield significant performance improvements.

In summary this example shows how Eden's richer coordination constructs, compared to GPH and PMLS, can be used to improve parallel performance, without having to resort to mechanisms of explicit synchronisation. The higher level of abstraction in GPH and PMLS reduces programming effort for the initial version, but also reduces the amount of programmer control. Although we describe Eden as having the most explicit coordination in this comparison, it is far more implicit than most conventional parallel programming languages.

4.2.5. Comparison with C. The three parallel matrix multiplication algorithms have also been implemented in C+PVM using the GMP (GNU Multi-Precision) library to handle arbitrary sized integers. The program sizes differ substantially from the parallel functional programs. The sequential C matrix multiplication program using the GMP library consists of 156 lines of code (excluding blank lines and comments), while the parallel programs comprise 378 lines of code for the row-clustering algorithm, 436 lines for the block-clustering version and 457 lines for the torus algorithm. The parallel C+PVM programs are a factor of 4 to 5 longer than our parallel functional programs. Table 2 shows some runtimes and

Table 2. Performance results for parallel C+PVM matrix multiplication programs on the Beowulf (runtimes in seconds).

	Row-clustering		Block-clustering		Torus	
No. of PEs	RT	Spdup	RT	Spdup	RT	Spdup
1	5.75	1	5.75	1	5.75	1
4	2.00	2.87	2.00	2.87	1.93	2.98
9	1.36	4.23	1.18	4.87	1.08	5.32
16	1.34	4.29	1.03	5.58	0.79	7.28
25	1.83	3.14	0.97	5.93	0.68	8.46

speedups of the different parallel C+PVM programs for 200 $\times$ 200 matrices of arbitrary precision integers.

The most involved torus version of the program yields the best parallel runtimes and speedups. While the sequential runtime is a factor of 4 to 6 better, the speedup values progress in a similar way as for the functional programs.

4.3. LinSolv

4.3.1. 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 $A \in \mathbb{Z}^{n\times n}$, $b \in \mathbb{Z}^n$, $n \in \mathbb{N}$. In contrast to more common numerical algorithms, which usually produce an approximate solution over floating point numbers for a given accuracy, the algorithm presented here finds an exact solution and works over arbitrary precision integers.

4.3.2. Parallel algorithm. To find an exact solution for a given system of equations, `linSolv` uses a *multiple homomorphic images* approach [43]. This is a common computer algebra approach and consists of the following three stages:

1. map the input data into several homomorphic images,
2. compute the solution in each of these images, and
3. combine the results of all images to a result in the original domain.

Figure 19 depicts this structure for the implementation of `linSolv`. This structure is particularly useful for operations on arbitrary precision integers. In this case the original domain is $\mathbb{Z}$, the set of all integer values, and the homomorphic images are $\mathbb{Z}$ modulo p, written $\mathbb{Z}_p$, with p being a prime number. If the input numbers are very big and each prime number fits into one machine word the basic arithmetic in the homomorphic images is cheap fixed precision arithmetic. Only in the combination phase, when applying a fold-based Chinese Remainder Algorithm (CRA) [45], expensive arbitrary precision arithmetic has to be used to construct the result values. A detailed discussion of several variants of this algorithm is given in [46].

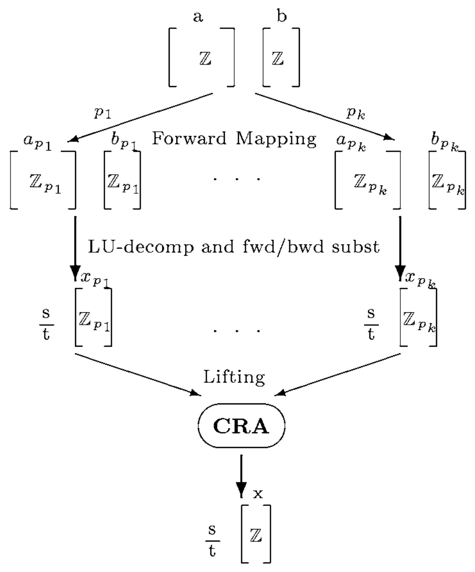

Figure 19. Structure of the `linSolv` algorithm.

The basic parallel structure of the algorithm is one of performing all computations in the homomorphic images in parallel. The Haskell code for the top-level function, which is unchanged for the parallel GPH version, is shown in Figure 20. It uses LU-decomposition followed by forward and backsubstitution to compute the solution `pmx` in the homomorphic image [69]. The main difficulties in the parallel algorithm are two-fold. Firstly, we have to make sure that new results are computed if primes turn out to be "unlucky", i.e. if the determinant of the input matrix *A* in the homomorphic image generated by this prime number is zero. This can be done either using demand-driven evaluation (GPH) or adding explicit code to handle that case (Eden, PMLS). Secondly, we have to avoid a sequential bottleneck in the combination phase at the end. In earlier papers we have experimented with a tree-based Chinese Remainder Algorithm to reduce this bottleneck. However, an analysis of the tree-based code [46] reveals that this algorithm performs much more total computation than a list-based one, due to the more expensive computations at each node of the tree, and we therefore use a list-based CRA in the parallel algorithm.

4.3.3. Implementations.

4.3.3.1. GPH. The parallel GPH version attaches the strategy shown in Figure 21 to the top level expression of the sequential code in the last line of Figure 20. We use an infinite list `xList` representing the results of all homomorphic images together with the prime number,

```
linSolv :: SqMatrix Integer ->                 -- nxn matrix A
           Vector Integer ->                   -- n vector b
           (Vector Integer, Integer, Integer) -- n vector x s.t. A*x=b
linSolv a b =
  let
   {- Step1: forward mapping -}
    ...
   {- Step2: Computation of solutions in Z/p -}
    ...
    -- Infinite list of hom. solutions of a*x=b in Z_p
    xList = map get_homSol primes

    get_homSol :: Integer -> [Integer]
    get_homSol p  =
      let
        b0 = toHom p b
        a0 = toHom p a
        modDet = toHom p (determinant a0)
        pmx = -- inlined version of: homsolv0 p a0 b0
              let
                lua = lu p a0
                (l,u) = split_lu p lua
                y = fwd_subst p l b0
                x = bwd_subst p u y
              in
              x
      in
      p : modDet : pmx

   {- Step3: lifting via list-based CRA -}
    ...
    primeList = projection 0 xList -- primes (bases for the hom ims)
    detList   = projection 1 xList -- dets in all hom ims
    det       = snd (list_cra pBound primeList detList detList)
    x_i i     = snd (list_cra pBound primeList x_i_List detList)
                where x_i_List = projection (i+2) xList
    -- overall solution:
    x = vector (map x_i [0..n-1])
    ...
  in
  x 'using' strat
```

Figure 20. Top level code of the sequential `linSolv` algorithm (Haskell version).

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

Figure 21. Parallel strategy for `linSolv` (GPH version).

```
xList_all = map_rw get_homSol primes

xList = filter lucky xList_all
```

Figure 22. Parallel `linSolv` (Eden speculative version).

as the basis of the image, and the value of the determinant of A in that image. The strategy guesses the number of primes needed to compute the overall result (`noOfPrimes`) and uses a `parListN` strategy to generate data parallelism over that 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 result to be evaluated in parallel. We need to check whether the determinant is zero to avoid redundant computation. This check is done here, rather than when computing `noOfPrimes` to minimise data dependencies in the algorithm. If some prime numbers turn out to be unlucky the `list_cra` will evaluate the additional results by demanding as-yet-unevaluated list elements. The final strategy application `parList rnf x` specifies that all elements of the result should be combined in parallel.

4.3.3.2. Eden. Even though computation in Eden is lazy, communication is eager, except for stream-like lists. Thus, care has to be taken not to send the whole list. To ensure a demand-driven evaluation of homomorphic solutions we use a task farm skeleton as outlined in Section 3.1.1. More specifically, we use the replicated workers paradigm [44]. A manager and a set of worker processes are created, and two tasks are initially released to each of the workers. As soon as any worker finishes a task, it sends the result to the manager, and a new task is delivered to the worker. The computation in the manager is demand-driven and triggered by the availability of result values. As soon as the manager has all the needed results it terminates all the worker processes. Notice that in this *speculative version* the workers may be working speculatively on useless tasks, but only when the useful tasks have already been consumed and hence the degree of speculation is tightly limited. More details about the replicated workers skeleton can be found in [41]. Figure 22 shows the Eden code for the speculative version of `linSolv`. The only modification to the sequential code is the use of a parallel replicated workers map `map_rw` instead of a sequential map over the infinite list of primes.

To avoid the potential waste of resources due to speculation we can implement a *conservative version* as shown in Figure 23. In this version the prime numbers are divided

```
xList_all = map_rw get_homSol primes

xList = filter lucky xList_all
xList_unlucky = filter (not.lucky) xList_all

(p_needed, p_spec) = splitAt ( 1 + toInt noOfPrimes) primes
primes' = p_needed ++ (additional xList_unlucky p_spec)

additional :: [Integer] -> [Integer] -> [Integer]
additional xs ys = zipWith (\ x y -> y) xs ys
```

Figure 23. Parallel `linSolv` (Eden conservative version).

into those known to be needed (p_needed) and those which are only needed if some of the earlier primes are unlucky (p_spec). The function additional adds for each unlucky prime a new prime number to the task list primes'. Note in the definition of additional that due to the demand-driven evaluation the availability of unlucky primes in xs triggers the generation of one result element in ys.

4.3.3.3. PMLS. The PMLS implementation has been developed from the sequential Haskell implementation. Arbitrary length integers are provided by Objective Caml's num library, whereas GPH and Eden use the GNU GMP library. Replacing the default arithmetic for SML with these arbitrary precision routines exposes some limitations of SML's overloading scheme. In direct comparison this step was easier in Haskell.

The main problem in the PMLS implementation, shown in Figure 24, is the handling of *unlucky primes*. Because SML is strict, new primes cannot simply be demanded during

```
(* Solve ax = b modulo p *)
fun gen_xList a b p =
  let
    val (a0,b0) = (matHom p a,vecHom p b)
    val modDet = modHom p (determinant a0)
    val ((iLo,jLo),(iHi,jHi)) = matBounds a
    val pmx =
      fxlist jLo (jHi-jLo+L1)
        (fn j => modHom p (determinant (replaceColumn j a0 b0)))
  in
    p::modDet::pmx
  end

(* Iterative forward mapping phase *)
fun getSols xList [] = xList
  | getSols xList primes =
  let
    val xList' = map (gen_xList aN bN) primes
    val noUnlucky = countUnlucky xList'
    val xList' = filter (not o isUnlucky) xList'
    val primes' = additionalprimes primes noUnlucky
  in
    getSols (xList@xList') primes'
  end
val xList = getSols [] (primesuptomaxprod pBound)

(* Combination via CRA *)
val detList = projection 1 xList
val det = list_cra pBound primes detList detList
fun x_i i =
  let
    val x_i_List = projection (i+2) xList
  in
    list_cra pBound primes x_i_List detList
  end
val x = seqmap x_i (fxlist 0 n (fn x => x))
```

Figure 24. Parallel linSolv (PMLS version).

the evaluation of the `map` skeleton. There are two possible solutions to this problem. Either the homomorphic solution function could generate a new prime upon detecting an unlucky one, as it is done in the conservative Eden version, or the forward-mapping phase could be made iterative with the number of valid solution vectors as a convergent. The second of these was implemented since there are problems with generating unique primes within the `map` instance function. Unfortunately this solution to the problem of unlucky primes results in less efficient parallelism for two reasons.

Firstly, in the iterative solution we introduce sequential synchronisation points at the end of each iteration to exchange data between the processors. This is required to guarantee that all processors, computing an element of the result vector, terminate on the same iteration. This nesting of parallelism inside an iterative structure is a general problem with our methodology. To overcome this problem it would be possible to either broadcast the convergent, introducing additional communication, or to define a special *iterative* skeleton, as it is done in the SKIPPER system [77]. However, we choose a solution that is more general albeit also more costly.

Secondly, the amount of parallelism is drastically reduced by the `map` call during the first iteration of the `getSols` function. Usually, only one or two unlucky primes are found for modest sizes of problems. If the number of unlucky primes is a multiple of the number of processors (including zero) then there is no parallel performance penalty, otherwise there is a minimum of one homomorphic solution time as an overhead. Additionally, the optimal granularity of the `map` call will be different between the iterations, the first phase more efficient with coarser granularity (since there will be the total number of estimated primes to decompose over), the latter with minimal granularity (since there will only be a small number of unlucky primes). We can set the granularity at runtime but this, currently, requires explicit programmer input. An alternative would be to have dynamic behaviour in our skeletons.

4.3.4. Performance results. As inputs for the performance measurements we use a dense 62×62 matrix of arbitrary precision integers. No element in the matrix is larger than $2^{16} - 1$ and the density of the matrix is higher than 90%. The sequential runtimes show PMLS to achieve best single processor performance with 190.8s, followed by GPH with 307.9s, and Eden with 491.7s. We attribute this fairly large difference mainly to algorithmic differences in the code: The PMLS version uses a more efficient forward substitution after LU decomposition in the homomorphic solution phase. This difference, in combination with the lazy evaluation mechanism used in GPH and Eden, leads to a higher heap consumption resulting in higher overall runtime. Furthermore, due to implementation limitations GPH currently has to use a two-space garbage collector, which is known to be less efficient than the generational garbage collector used by GHC for sequential execution (see below). Finally, the difference between Eden and GPH is due to the fact that Eden uses an older version of GHC.

Figure 25 shows the runtimes and Figure 26 shows the relative speedups for the Eden, GPH, and PMLS implementations of `linSolv` for up to 16 PEs on the Beowulf cluster. For the input data used in these measurements a sufficient number of lucky primes are generated to utilise all processors in the machine. Since these top-level threads can compute

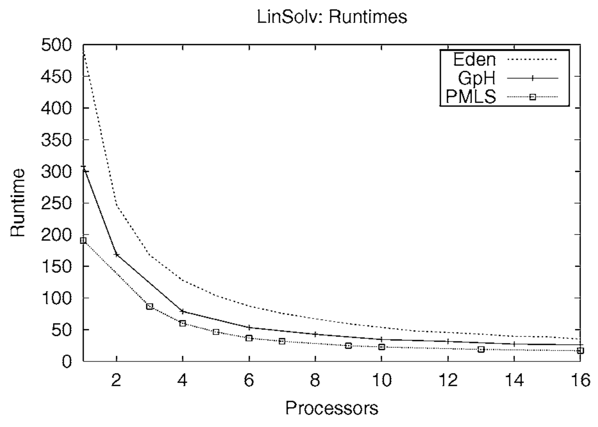

Figure 25. Runtimes of `linSolv` on the Beowulf.

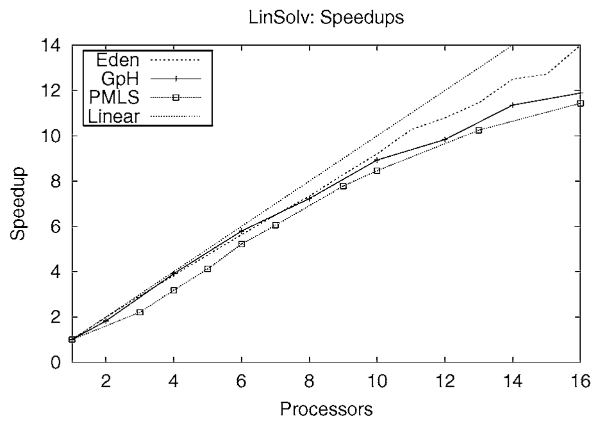

Figure 26. Speedups of `linSolv` on the Beowulf.

their results independently, they perform relatively little communication and the parallel overhead is relatively small giving good parallel efficiency.

A direct comparison of the different languages shows that Eden achieves the best overall speedup on 16 PEs: 14.0, compared to GPH at 11.9 and PMLS at 11.5. However, since Eden has far higher sequential execution time, the PMLS version is the fastest one on 16 PEs. An examination of the activitiy profiles reveals that PMLS's skeleton maintains more parallelism while collecting the data, whereas in GPH this final stage is mostly sequential.

The Eden measurements use the speculative version with the replicated workers skeleton that dynamically sends work to processes. This approach achieves dynamic load distribution without relying on a potentially expensive implementation of a virtual shared heap, as used in GPH, and the measurements show good speedups even beyond 16 PEs.

In examining the dynamic memory management of all systems, we observe that the total heap allocation on all PEs is highest for PMLS: 1052 MB, whereas GPH allocates only 618 MB. However, due to higher maximal heap residency in GPH, processors spend on average 19.0% of the total execution time on garbage collection, whereas in PMLS this percentage is only 11.2%. Measuring the heap fragmentation of both systems as the standard deviation of allocation on each processor we obtain similar values for both systems, 147 MB for PMLS and 157 MB for GPH. This indicates that in `linSolv`, GPH's dynamic memory management does not dramatically increase heap fragmentation.

As these numbers indicate, GPH's garbage collector seems to generate higher overheads than that in PMLS. The main reason is the current usage of a one-generation copying collector, rather than a real generational collector as supported by GHC for sequential compilation [74]. Furthermore, Objective Caml's two-generation collector, as used by PMLS, provides cheap incremental collection for the young generation, which better exploits the additional heap space provided by multiple PEs. The implementation of a mark-and-sweep collector for the older generations is known to be very efficient, too [15]. Another potential reason for this overhead is the weighted reference counting on global pointers in GPH, although this overhead allows PEs to collect local garbage independently, avoiding global synchronisation.

In summary the `linSolv` example demonstrates that for some applications lazy evaluation can reduce the amount of coordination required. Both the conservative Eden and the PMLS versions had to introduce additional coordination to model GPH's demand-driven generation of parallelism and to handle unlucky prime numbers. In Eden the speculative version proved to be faster than the conservative version, but in general such an approach bears the danger of wasting resources. Although the static partitioning and mapping of PMLS is generally less flexible than the approach taken in GPH, the re-use of well-tuned parallel skeletons can compensate for the loss in flexibility in this case. It also induces smaller runtime-overheads e.g. for garbage collection. In terms of speedup the skeleton-based versions in Eden and PMLS are more efficient in collecting the results and achieve the following speedups on 16 PEs: 14.0 (Eden), 11.9 (GPH), 11.5 (PMLS), with PMLS having both the fastest sequential and parallel execution.

4.4. Ray tracer

4.4.1. Problem description. The `raytracer` program calculates a 2D image of a scene of 3D objects by tracing all rays in a grid, or window. In tracing a ray, the intersections with the objects are computed. When an intersection is found, the ray is reflected and the colour of the intersection point is computed based on the strength of the ray and on the texture of the object's material. The code is based on the Id version that was published as a part of the Impala suite [34] of parallel benchmark programs.

4.4.2. Parallel algorithm. Figure 27 shows the top-level function of the sequential Haskell algorithm. The function `ray` takes the size of the window in `x` and `y` dimension and the

```
ray :: Int -> Int -> [Sphere] -> [[((Int, Int), Vector)]]
ray x y world = map do_line sizes_y
    where do_line :: Int -> [((Int,Int), Vector)]
          do_line i = map (\ j -> ((i,j), f i j)) sizes_x
          sizes_x = [0..x-1]
          sizes_y = [0..y-1]
          f i j = tracepixel world lights i j firstray scrnx scrny
          (firstray, scrnx, scrny) = camparams x y
```

Figure 27. Sequential `raytracer` (Haskell version).

`world`, represented as a list of spheres, as input. The computation proceeds as two nested maps, with the outer map operating over the lines of the grid and the inner map, `do_line`, applying the `tracepixel` function to every point in the grid, represented by the coordinates `(i,j)`, returning a vector representing the colour.

We consider two parallel versions of this program. Both versions exploit data parallelism but differ slightly in the way the data is initially distributed.

4.4.2.1. Parallel map. Because the computation to be performed on each pixel, `tracepixel`, is fairly cheap, we do not exploit parallelism in the inner map but instead execute only the outer map in parallel. To achieve good granularity in the outer loop, the computation over several lines are collected into chunks and processed together.

4.4.2.2. Direct map. The direct map version exploits the same kind of data parallelism but differs in its initial distribution of data. Each process is given all necessary data and extracts its own portion of the data by selecting lines in the grid. To improve the granularity of the communication, sub-sequences of pixels are collected into packets. Typically as many tasks as available processors (`noPe`) are generated. To improve the load-balance, task `i` ($0 \leq$ `i` $\leq$ `noPe-1`) computes all result lines `i + j` $*$ `noPe` with `j` ≥ 0. Note that in this version no dynamic distribution of tasks is required after the startup phase. Compared to the parallel map version this should achieve a faster startup of the parallel processes and a better load distribution.

4.4.3. Implementations.

4.4.3.1. PMLS. The PMLS implementation in Figure 28 uses a parallel map and has been developed from the sequential Haskell version in Figure 27.

The function `do_pixel` initiates ray tracing at pixel co-ordinate `(i,j)` and is mapped over the index image `ind`. The same considerations regarding granularity control apply to PMLS. However, the choice whether to do the outer or inner `map` in parallel is determined by the characteristics of the PMLS runtime-system. When two skeletons are direct arguments to each other, as in the example here `(map (map do_pixel))`, there is no advantage in nested implementation since granularity control is performed automatically inside the `map` skeleton. In addition, the PMLS compiler requires all free variables in the function position of a `map`, in this case `do_pixel`, to be sent to the individual processors when initialising

```
fun ray x y world =
  let
    val (firstray, scrnx, scrny) = camparams x y
    fun do_pixel ij =
      let
        val (i,j) = (ij div 1000,ij mod 1000)
      in
        ((i,j),tracepixel world lights (real i) (real j) firstray scrnx scrny)
      end
    val ind = indxs 0 (x - 1) 0 (y - 1)
  in
    map (map do_pixel) ind
  end
```

Figure 28. Parallel `raytracer` (PMLS version).

the skeleton instance. Since the inner `map` has free values which could (potentially) change between successive calls they have to be re-transmitted upon each call. This means that the total amount of data transmitted is significantly less if the outer map is implemented in parallel.

4.4.3.2. GPH. The GPH implementation is based on the parallel map version and uses an additional explicit parameter `chunk` to control the size of the chunks. The code in Figure 29 shows the body of the function `ray` (the local definitions are unchanged), with an evaluation strategy implementing granularity control via clustering. We use the same `parListChunk` strategy as in the row-clustered matrix multiplication.

4.4.3.3. Eden. The Eden implementation uses the direct-map version and is shown in Figure 30. The function `f_dm` represents the work to be executed by one process. In the direct-map version this includes the extraction of its own portion of the input data using the `takeEach` function to combine every *n*-th line of the grid into one chunk.

```
ray :: Int -> Int -> Int -> [Sphere] -> [[((Int, Int), Vector)]]
ray chunk x y world = map do_line sizes_y
                      `using` parListChunk chunk rnf
```

Figure 29. Parallel `raytracer` (GPH version).

```
ray :: Int -> Int -> [Sphere] -> [[((Int, Int), Vector)]]
ray x y scene = shuffleN outps
    where outps = [ (process i -> f_dm i) # void | i <- [0..noPe-1]]
                      `using` seqList r0
          f_dm n _  = map do_line (takeEach noPe (drop n sizes_y))
```

Figure 30. Parallel direct map version of `raytracer` (Eden version).

The processes are created in a list of process instantiations (`outps`). The sequential strategy `seqList r0` is used to drive an eager process creation, creating the processes before the outport values are needed.

4.4.4. Performance results. The measurements in Figures 31 and 32 use a 350 × 350 image with a chunk size of 10 and a scene consisting of 640 spheres as input. The sequential runtimes are: 177.4s for Eden, 163.3s for GPH, and 172.1s for PMLS. For this application the sequential performance of all three versions is fairly similar with a variation of less than

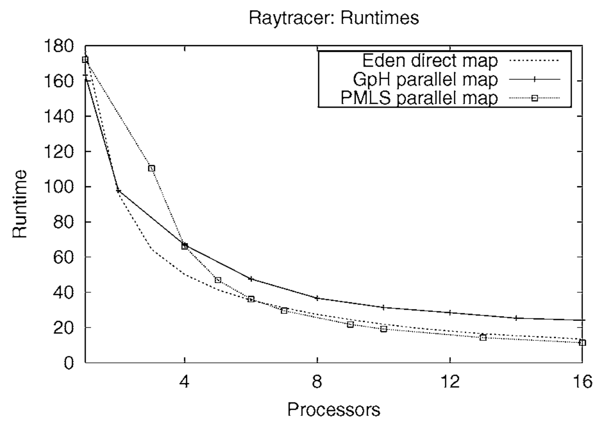

Figure 31. Runtimes for `raytracer` on the Beowulf (in seconds).

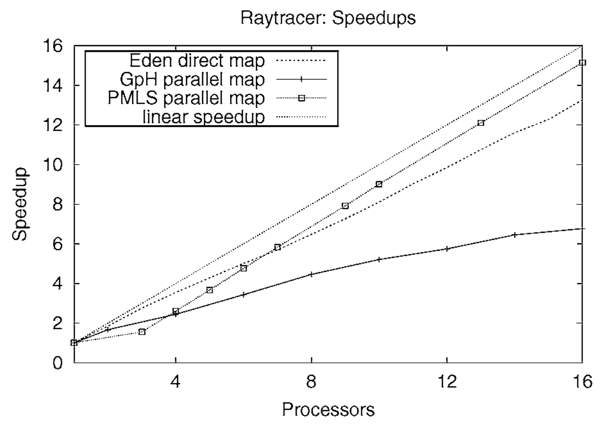

Figure 32. Speedups for `raytracer` on the Beowulf.

10%. This is mainly due to the fact that `raytracer` does not make use of the laziness in the language: all parts of the picture are indeed computed and since there is no interaction they can be computed eagerly. This dynamic program characteristic manifests itself in similar garbage collection overheads for PMLS and GPH: 3.3% and 3.1% as mean over all processors.

For PMLS initial sequential results showed significantly poorer performance than the GPH and Eden versions. This is due to a known limitation of the PMLS compiler. The results reported here required some minimal user interaction during the compilation process. The PMLS group is currently adding an appropriate sequential optimisation step to the compiler.

The rather simple and regular structure of the computation lends itself to a static approach such as the static task farm in PMLS or the direct-map in Eden. The partitioning of the program can be achieved statically and the distribution of work is carried out only once at the beginning of the program. Since the work is fairly evenly distributed, no sophisticated dynamic load balancing is necessary. On the other hand enforcing a fixed data distribution is easier in Eden than in GPH. In general, the more dynamic facilities of GPH are not used in this application. We have experimented with GPH versions that model the Eden approach more closely, but they did not yield any significant performance improvements.

Not surprisingly for an application with a fairly regular structure of parallelism, PMLS performs best in terms of speedup as well as absolute runtime. On 16 processors the runtime is 11.4s, corresponding to a relative speedup of 15.1. The results for Eden, with its slightly more dynamic resource management, are similar: parallel runtime of 13.4s with a speedup of 13.2. GPH pays a higher cost for its dynamic resource management, resulting in a comparatively poor speedup of 6.8 on 16 processors and a parallel runtime of 24.1s.

Another problem we have observed in the GPH version is a potentially poor load distribution where few processors monopolise the entire available parallelism. This is due to a combination of factors: In this program all parallelism is generated on the main processor at the beginning of the computation, and on the Beowulf start-up times between PEs may vary significantly. Moreover this version of the GPH runtime-system does not currently allow tasks to migrate from a loaded PE to an idle PE. Hence the fastest processor(s) sometimes obtain all available work before the slower processors have a chance to send their first work requests. It is possible to crudely control the work distribution by imposing an upper limit on the number of threads that may be alive on one PE, and that is what we used in these measurements. A more recent prototype of a GPH runtime-system, extended with thread migration, achieves similar performance without using such an upper limit on the number of threads [16].

In contrast, for PMLS load-balancing is assumed to be a property of the skeleton implementation. The parallel map skeleton used by all applications has a degree of implicit load-balancing as a result of the processor farming model. This works well in cases like `raytracer` but requires manual tuning for particular instances which can change as execution proceeeds (for example in `linSolv` different balancing strategies are used for the initial and the additional results). Eden's replicated worker skeleton `map_rw` as used in the `linSolv` example provides implicit dynamic load balancing based on the master worker paradigm. Surprisingly, this skeleton is outperformed for the raytracer by a static work distribution where the work list is sent to all processors and the work packages are selected locally within each process.

In summary, the results for `raytracer` underline a general trend in these measurements for Eden and GPH, namely the impact of dynamic resource management overheads on scalability. Eden, which has a lower overhead, performs almost as well as PMLS. However GPH has to maintain a virtual shared heap, and this diminishes parallel performance for larger numbers of processors. In some cases we have observed an overhead of up to 16% of the total execution time, although typical percentages are 3–8% [49]. We are also investigating refined load balancing mechanisms, which show better performance.

5. Related work

For comprehensive overviews on parallel functional programming we refer to [25] and [81]. In this section we focus on comparing our approaches with other implemented systems. Only few implementations have overcome a purely experimental status and concrete head-to-head comparisons of different languages on the same architecture are even rarer. To our knowledge this paper is the first such systematic comparison.

5.1. Skeleton-based approaches

The prospect of implicit parallelism with the use of skeletons has spurred the development of several skeletons-based systems. HDC [29] is a strictly-evaluated subset of Haskell with skeleton-based coordination, in particular support for `fold` and `map`, and several forms of divide-and-conquer. For the Karatsuba algorithm for polynomial multiplication HDC achieves a relative speedup of 363 on 729 processors of a 1024-processor transputer-based Parsytec machine.

A system closely related to GPH is Caliban [36, 78] in which `moreover` clauses, similar to GPH's `using`, can be attached to sequential program source in order to specify parallel behaviours. Expressions are annotated to indicate tasks to be created, and the linkage between the tasks can be specified using normal functions. In the current implementation, the process network is static, with `moreover` clauses being resolved at compile-time and processes being statically mapped to the target topology. A simple raytracer, introduced in [36], has been measured on a 128 processor Fujitsu AP1000, achieving speedups of up to 24 on 35 processors.

Other prominent skeleton-based systems are SCL [13] and P3L [1]. Both use separate coordination languages with small sets of basic skeletons that can be freely nested. The most mature implementation of SCL, SPF, uses Fortran as computation language. Substantial applications such as a Barnes-Hut algorithm have been implemented in SPF [14] and measured on a Fujitsu AP1000. In [63] performance results for P3L on four applications are presented, including a parallel raytracer, obtained with the `SkIE` prototype environment for P3L on a 24-node Meiko CS-2 and an 8-PC Linux cluster.

An active research area in the skeletons community is the nesting of skeletons [21]. In particular, with support for nesting it is possible to construct complex parallel applications by composing and transforming skeletons using given transformation rules and compositional cost models for performance prediction as developed in [2] and [65].

5.2. *Thread- and process-based approaches*

Para-functional programming [32] is the general approach of adding control directives to a functional program in order to specify parallel execution. These control directives allow the programmer to describe detailed schedules of the execution as well as a particular mapping of threads to processors. First-class schedules [58] extend para-functional programming to Haskell, using monads to separate expressions and control directives. These annotations usually describe potential parallelism, in the sense of GPH's `par`, and therefore represent a thread-based approach. Its implementation builds on the concept of futures, as used in MultiLisp [20]. First-class schedules have been implemented by compiling Haskell to the MultiLisp-based operating system STING. Preliminary performance results on a 16 processor Silicon Graphics Challenge shared-memory machine show good speedups for a parallel Barnes-Hut algorithm for solving the n-body problem [57].

ALFL [19] is an LML-like, lazy, implicitly-parallel functional language, implemented on a distributed-memory Intel Hypercube as well as on a shared-memory Encore machine, with performance comparisons between the two architectures.

Concurrent Clean [62, 68] is a lazy language with parallelism annotations. In [37] performance results for three systems are reported: Concurrent Clean on the ZAPP abstract machine; Concurrent Clean on the PABC abstract machine; and a Miranda-like, implicitly-parallel, lazy language, implemented on the abstract HDG machine [39]. All measurements have been performed on (different) transputer networks. In contrast to this paper, no detailed comparison of languages or systems is given. Good speedups are reported for small programs such as nqueens (5.6 on 8 processors) but poorer results for a raytracer (3.9 on 16 processors) in this implementation of Concurrent Clean [37].

5.3. *Other approaches*

One of the most successful data parallel functional languages is NESL [3]. NESL is a strict, strongly-typed, data-parallel language with implicit parallelism and implicit thread interaction. It has been implemented on a range of parallel architectures, including vector machines. A wide range of algorithms has been parallelised in NESL, including a Delaunay algorithm for triangularisation [4], several algorithms for the n-body problem [5], and several graph algorithms. The focus in these papers, however, is on the comparison and improvement of algorithms rather than speedup measurements or a comparison with other languages. Two data-parallel extensions of Haskell have been partially implemented: Data Field Haskell [31] and Nepal [10]. No performance results are available, yet.

SISAL [9] is a first-order, strict functional language with implicit parallelism and implicit thread interaction. Its implementation is based on a dataflow model and it has been ported to a range of parallel architectures. Good absolute performance in comparison to Fortran code is reported in [42].

The pHluid system [17] is a parallel implementation of Id on networks of workstations, using a dataflow model of computation in order to achieve implicit parallelism. Id is polymorphic, higher-order and has a non-strict semantics, implemented via lenient or parallel eager evaluation. A fusion of Id and Haskell, called pH, has been proposed [61] but no implementation is available, yet. On a workstation cluster near-linear (relative) speedups are reported for simple programs such as nqueens [17]. In [22] a rare language and performance comparison of implicitly parallel Id with sequential Haskell on a realistic benchmark program is given.

6. Conclusions

We have compared three state-of-the-art parallel functional programming systems (PMLS, GPH, and Eden) and evaluated their performance on a Beowulf architecture using three symbolic applications: several matrix multiplication algorithms using arbitrary precision arithmetic (`matMult`); an exact linear system solver (`linSolv`); and a simple ray-tracer (`raytracer`).

PMLS, GPH and Eden all aim to support parallel symbolic computations at low programmer cost. While it is relatively straightforward to achieve good (often linear) speedups for *regular*, *numerical* parallel computations, it can be much harder, or even impossible, to achieve the same results for *irregular*, *symbolic* computations, especially those with complex data structures or irregular task structures [53]. Relatively small performance improvements may thus be of much greater significance to users of such systems. At the same time, symbolic application programmers are usually domain experts rather than computer scientists, and are often unwilling or unable to invest major effort in recoding for parallelism. In this section, we will evaluate the three systems in terms of language features, performance, and productivity. We will consider them in order of anticipated programmer effort: namely PMLS, GPH, and Eden.

6.1. Language comparison

All three functional languages aim to provide higher-level models of parallelism, with the objective of reducing programmer overhead. All three abstract over low-level details of communication timing, data structure marshaling (including cyclic graph structures) and synchronisation that must be specified in e.g. C+PVM. Moreover, in all three languages, details of task/thread creation and program decomposition are delegated to the compilation system.

PMLS provides a convenient model of implicit parallelism using *skeletons*—a set of pre-defined higher-order functions with associated parallel behaviours. Since skeletons are partitioned into parallel components and mapped to processing units statically, this approach has the lowest runtime overhead of the three considered here, and where the application structure fits the pre-defined skeletons perfectly, it will also have the lowest programmer overhead. However, such an approach is less flexible than the dynamic approaches taken by Eden and GPH. This is apparent in less regular or longer-running applications, such as `linSolv`, where a regular static structure cannot be determined from the program source.

GPH has a similar philosophy to that of PMLS, aiming to require minimal programmer input in order to achieve acceptable parallel performance. However, it provides more control (if required) over evaluation order, strictness and parallelism, allowing programmable evaluation strategies to be developed. This approach accepts low programmer overhead to enable tuning of the parallel code and to be applicable to a variety of programming styles, but may incur potentially high dynamic overhead. This cost is most apparent in regular applications, where a simple static process to processor mapping could be determined either manually or automatically. In such a case, manual tuning may be needed to extract good parallel performance for GPH, where PMLS might automatically find such a mapping, or it might be straightforward to program such a mapping in Eden. `raytracer` is an example of such simple static mapping.

Finally, of the three languages studied here, Eden provides the greatest control over parallelism, and thus requires the greatest programmer effort. Control is provided over task decomposition, allocation to virtual processors and communication channels. Given sufficient tuning effort, it is possible to develop more sophisticated parallel algorithms, as with the torus version of `matMult` (Section 4.2.2). As with GPH, all PMLS skeletons can be easily replicated [40, 64], with a similar mapping effect. Since all load management details must be explicitly programmed, however, and there is limited support for lazy communication there will be situations where GPH mechanisms cannot be easily replicated, such as using a potentially infinite number of homomorphic images in `linSolv`.

Recognising the value of the skeletons approach for suitable applications, all three languages provide support for such a style. PMLS naturally provides the most direct support, with static process mapping and cost modelling as part of the compilation process. GPH provides a full set of standard skeletons written in Haskell, and using a dynamic cost model and mapping [26]. Haskell's constructor classes are used to abstract over machine models and alternative data structures. Finally, a rich set of skeletons, including some novel branch-and-bound skeletons, has been developed using Eden constructs and used on several parallel machines [40, 64].

6.2. Performance comparison

It is received wisdom that eager evaluation (used for strict function calls) will outperform lazy evaluation (used for non-strict function calls) due to the overhead of recording partial results in the latter case. It follows that fully strict languages should outperform non-strict languages: experimental results suggest that this can be over a factor of 10 in the worst benchmark cases [27].

Similarly, it is argued that full communication should outperform lazy communication, since fewer messages are required in the former case if an entire data structure is communicated. Given that PMLS is fully strict, with strict communication, Eden is non-strict, with strict communication and GPH is non-strict with lazy communication, we would consequently expect PMLS to outperform Eden which should outperform GPH. We would also expect the same ordering on the basis of runtime overheads, but with the possibility of similar overheads for Eden and PMLS. The performance results summarised in Table 3 are therefore somewhat surprising.

For all three benchmarks PMLS achieves the smallest parallel execution times. In the case of `linSolv`, Eden's speedup is higher but sequential execution time is higher, too. GPH achieves similar speedup as PMLS with sequential time between the other two versions. In the case of the `raytracer` (the most regular of the three benchmarks we have considered) PMLS shows even better speedups than Eden or GPH. While mirroring earlier results almost

Table 3. Comparative performance (Seq RT: runtime on a 1 PE parallel machine; Par RT: runtime on a 16 PE parallel machine; Spdup: Speedup on 16 PEs calculated as $\frac{\text{Seq RT}}{\text{Par RT}}$).

	PMLS			GPH			Eden		
	Seq RT	Par RT	Spdup	Seq RT	Par RT	Spdup	Seq RT	Par RT	Spdup
`matMult`	22.8s	4.3s	5.3	30.3s	8.9s	3.4	38.5s	13.2s	2.9
`linSolv`	190.8s	16.6s	11.5	307.9s	25.9s	11.9	491.7s	35.1s	14.0
`raytracer`	172.1s	11.4s	15.1	163.3s	24.1s	6.8	177.4s	13.4s	13.2

Table 4. Comparative performance of matrix multiplication in C (Seq RT: runtime on a 1 PE parallel machine; Par RT: runtime on a 16 PE parallel machine; Spdup: Speedup on 16 PEs calculated as $\frac{\text{Seq RT}}{\text{Par RT}}$).

	PMLS		GPH		Eden		C		
	Seq RT	Spdup	Seq RT	Spdup	Seq RT	Spdup	Seq RT	Par RT	Spdup
`matMult` (rows)	19.5s	3.3	–	–	34.3s	1.7	5.75s	1.34s	4.3
`matMult` (block)	22.8s	5.3	30.3s	3.4	32.9s	2.1	5.75s	1.03s	5.6
`matMult` (torus)	–	–	–	–	38.5s	2.9	5.75s	0.79s	7.3

exactly [26], the GPH performance for the `raytracer` is distinctly disappointing. This has subsequently led us to improve the GPH load distribution mechanism [48].

For comparison, we have re-implemented the `matMult` benchmark in C+PVM using the GNU Multi-Precision library for arbitrary precision arithmetic (Table 4) and the GNU C compiler on the same parallel machine. For the block-parallel version (the only one implemented in all four systems), the speedup results using full C optimisation (-O2) are comparable with those for PMLS: 5.6 on 16 processors. The base sequential performance is, however, a factor of 4 to 6 faster than for the functional languages. This factor between functional and imperative code stems on the one hand from the general overhead of implementing a high-level (lazy) language, and on the other hand the functional versions use list structures rather than in-place arrays. We anticipate that the difference could be further reduced by using e.g. monadic techniques to allow in-place array updates, but at some cost in source code legibility/programmer time. Furthermore, sophisticated type systems, enabling in-place update, are reaching maturity with first programming language implementations becoming available [30].

6.3. *Productivity comparison*

Measuring programmer productivity is notoriously difficult, due to differences in individual ability, prototyping effects, etc. We have therefore chosen to use lines of code as a reasonable

Table 5. Productivity comparison (in lines-of-code).

	PMLS		GPH		Eden		C	
	Seq Size	Par Size	Seq Size	Par Size	Seq Size	Par Size	Seq Size	Par Size
`matMult`	85	25	68	5	68	34	156	301
		(3)				(10)		
`linSolv`	751	13	473	10	473	8	–	–
`raytracer`	410	3	453	7	453	10	–	–

approximation, accepting that the number of lines of code produced by any given trained programmer is roughly constant regardless of the programming language used or the general ability of the programmer. For the three benchmark programs used here, application development was to some degree interleaved with system development. Therefore, a precise separation of the time spent on both activities, as we would have liked to give, was not possible. However, for the matrix multiplication program we note that debugging of the very simple C program, without further tuning, required more than a week already, whereas the parallel versions in PMLS, GPH, and Eden do not require major restructuring of the code. Whilst this does not represent a systematic comparison of parallel programming productivity in functional versus imperative languages, which is not the aim of this paper, we do believe that the following figures provide a realistic picture of expected productivity for typical symbolic applications.

Table 5 gives the number of lines of code for each of the three programs that have been studied here, plus corresponding figures for the arbitrary precision matrix multiplication program in C+PVM, which we discussed in more detail in Section 4.2.5. The counts exclude comments and white space. The parallel code size represents the number of lines that were either changed in or newly written for the parallel version. As expected, these changes are highly significant for the C program (representing some 65% of the total code size), but are generally insignificant for the functional programs (in the worst case, representing 33% of the total code size, for a highly tuned version of the matrix multiplication algorithm). The sequential functional programs are a factor of 2 to 3 times shorter than the C equivalent, with the parallel programs being 4 to 5 times shorter. Clearly, a certain amount of the C code could be reused for other applications, but there is equally clearly a very high entry price to parallel programming in C, especially when complex data structures must be communicated. Although not a major difference, the sequential Haskell code is generally slightly shorter than the SML code. This is mainly a consequence of better standard library support for Haskell, though high-level language features such as overloading and list comprehensions have also been exploited.

Since PMLS is the most implicit approach of the three languages studied, and Eden the most explicit, we would anticipate that PMLS should require least changes with Eden requiring the most changes. While this is generally true, the figures are distorted to some extent by the performance tuning that has been carried out. Although the initial version of the `matMult` in PMLS required only 3 lines, the final tuned version required 25. The corresponding Eden figures are 10 lines and 34 lines, respectively. The GPH code was not tuned, however, and therefore only 5 lines in total were changed. The `linSolv` application showed a reversal of the general result, with Eden requiring fewest changes. This may reflect the poor match between the irregular parallel structure of this application and the standard skeletons/strategies used by the other two systems. It is worth noting that the total number of changed lines is generally small, and that our comparisons must therefore be correspondingly tentative.

We conclude that while C may offer better performance than unoptimised functional code, in this example the difference is less than might be expected. Moreover, the high-level features available in functional code mean that programmer productivity is likely to be much greater than in C.

6.4. Maturity and usability

All three functional language systems discussed here can be rated as *mature research systems*, running a range of parallel benchmark applications on a variety of parallel architectures. Work on GPH began in 1994, and it since has been applied to numerous programs, including the 47,000 line Lolita natural language engineering system [50]. To assist program development, it offers a sophisticated set of profiling tools [23], including ideal and realistic simulation. GPH is publicly available in OpenSource form as part of the GHC compiler project and its most recent version, based on GHC 5.02.3, can be downloaded from [82].

The Eden system is a later development, sharing underlying parallel scheduling and communication infrastructure with the earlier GPH system. It has been tested on a variety of small and medium benchmark applications, but has not yet been applied to large-scale applications, such as Lolita. Both GPH and Eden provide low-level portability by compiling through either C+PVM or C+MPI.

In contrast to the two GHC-based systems, the PMLS system has a more heterogeneous structure, exploiting state-of-the-art implementation technology from several sources. The core system uses the Objective Caml compiler for sequential compilation and calls C+MPI routines for implementing the parallel skeletons. Up-to-date versions of Eden and PMLS are available from the developers on request.

7. Future work

All three systems are under active development. For PMLS the current objectives are to provide a more expressive set of algorithmic skeletons, to optimise the performance of the existing skeletons and to automatically identify skeleton structures in arbitrary code. This work will exploit both dynamic profiling-based performance prediction (which has been found to give good predictions within a narrow range of program characteristics) and automatic program transformation techniques.

The main research direction for GPH is to improve architecture-independence by refining the mechanisms for load balancing and data distribution in order to deal with high-latency machines such as Beowulf clusters. Based on these refinements, research will focus on the development of an adaptive runtime-system capable of automatically tuning its behaviour to suit the characteristics of the parallel machine.

Finally, following the upgrade to conform to the latest sequential GHC compiler, work in Eden will focus on optimisations to reduce communication costs.

Appendix

A. Auxiliary functions

This appendix summarises auxiliary functions, written in Haskell98, which we have used in the body of the paper. Most of these functions modify a data structure so as to define parallelism over this modified data structure. The sequential code can be re-used between GPH and Eden. In fact, the implementations of `matMult` and `raytracer` share a module with basic functions over lists shown in Figure 34.

Figure 33 presents the GPH code for some predefined strategies used in the body of the paper. The strategy `seqListN n s xs` forces the evaluation of the first `n` elements of the list `xs`, applying the strategy `s` to every list element. The strategy `parListChunk c s xs` specifies the evaluation of segments of size `c` of the list `xs` in parallel, applying the strategy `s` to every list element.

Figure 34 summarises the functions, used in the GPH and Eden code, for splitting lists into segments of (almost) equal size and merging them again. This is used for example by the `matMult` and `raytracer` examples to achieve “data clustering”. Note that such clustering is encoded within the skeletons used in PMLS and does not appear explicitly in the user code. The function `splitIntoN n xs` splits the list `xs` into `n` segments of the same size, whereas the function `splitAtN n xs` splits a list into segments of the size `n`. The function `takeEach` extracts each *n*-th element from a given list. It is used in `unshuffleN` to produce a list of lists of every *n*-th element, starting with 0-th, 1-st, 2-nd, etc. element. Thus, `unshuffleN` is an alternative form of clustering, observing the following identity for all *n* that divide the length of the input list:

```
shuffleN . (unshuffleN n) == id
```

```
-- sequentially applies a strategy to the first n elements of a list
seqListN :: (Integral a) => a -> Strategy b -> Strategy [b]
seqListN n strat []      = ()
seqListN 0 strat xs      = ()
seqListN n strat (x:xs) = strat x 'seq' (seqListN (n-1) strat xs)

-- applies a strategy to (sequential) chunks of a list in parallel
parListChunk :: Int -> Strategy a -> Strategy [a]
parListChunk n strat [] = ()
parListChunk n strat xs = seqListN n strat xs 'par'
                          parListChunk n strat (drop n xs)
```

Figure 33. Predefined evaluation strategies.

```
-- Auxiliary functions for splitting and merging lists
bresenham :: Int -> Int -> [Int]
bresenham n p = take p (bresenham1 n)
                where bresenham1 m = (m`div`p):bresenham1 ((m`mod`p)+n)

-- split list into n sublists of (almost) same size
splitIntoN :: Int -> [a] -> [[a]]
splitIntoN n xs = f bh xs
                  where bh = bresenham (length xs) n
                        f [] [] = []
                        f (t:ts) xs = hs : (f ts rest)
                                      where (hs,rest) = splitAt t xs

-- split list into blocks of size n
splitAtN :: Int -> [a] -> [[a]]
splitAtN n [] = []
splitAtN n xs = ys : splitAtN n zs
                where (ys,zs) = splitAt n xs

-- pick every n-th element from a list, starting from 0th elem
takeEach :: Int -> [a] -> [a]
takeEach n [] = []
takeEach n (x:xs) = x : (takeEach n (drop (n-1) xs))

-- list of lists of every n-th element, starting from 0th, 1st, ...
unshuffleN :: Int -> [a] -> [[a]]
unshuffleN n xs = [takeEach n (drop i xs) | i <- [0..n-1]]

-- combine a list of lists generated by unshuffleN
shuffleN :: [[b]] -> [b]
shuffleN ([]:_)  = []
shuffleN xss     = map head xss ++ shuffleN (map tail xss)
```

Figure 34. Functions for splitting and merging lists.

Acknowledgments

The authors would like to thank the following organisations for funding the work presented here: the Austrian Academy of Sciences (APART fellowship 624), the Japan Society for the Promotion of Science (Postdoctoral fellowship P00778), the Spanish Project TIC2000-0738, and UK's Engineering and Physical Sciences Research Council (grants EPSRC GR/L 93379, EPSRC GR/M 32351 and EPSRC GR/L42889). Furthermore, this research was supported by the MRG project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing. Our cooperative work would not have been possible without the travel funding from the following organisations: the British Council and the DAAD (grant 1097), and the Spanish-British Acción Integrada HB 1999-0102.

References

1. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., and Vanneschi, M. P^{3}L: A structured high level programming language and its structured support. *Concurrency—Practice and Experience*, **7**(3) (1995) 225–255.

2. Bacci, B., Gorlatch, S., Lengauer, C., and Pelagatti, S. Skeletons and transformations in an integrated parallel programming environment. In *PACT'99—Intl. Conf. on Parallel Architecture and Compilations Techniques*, Vol. 1662 of LNCS, 1999, pp. 13–27.
3. Blelloch, G. Programming parallel algorithms. *Communications of the ACM*, **39**(3) (1996) 85–97.
4. Blelloch, G., Miller, G., and Talmor, D. Developing a practical projection-based parallel delaunay algorithm. In *Symp. on Computational Geometry*. Philadelphia, PA, 1996, pp. 186–195.
5. Blelloch, G. and Narlikar, G. A Practical comparison of N-body algorithms. In Parallel Algorithms, Vol. 30 of Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1997.
6. Breitinger, S., Klusik, U., Loogen, R., Ortega, Y., and Peña, R. DREAM—the DistRibuted Eden Abstract Machine. In *IFL'97—Intl. Workshop on the Implementation of Functional Languages 1997*. Vol. 1467 of LNCS, St. Andrews, Scotland, 1997, pp. 250–269.
7. Breitinger, S., Loogen, R., Ortega, Y., and Peña, R. The Eden coordination model for distributed memory systems. In *HIPS'97—Workshop on High-Level Parallel Programming Models*. Geneva, Switzerland, 1997, pp. 120–124.
8. Burge, W. *Recursive Programming Techniques*. Addison-Wesley, 1975.
9. Cann, D. Retire fortran? A debate rekindled. *Communications of the ACM*, **35**(8) (1992) 81–89.
10. Chakravarty, M., Keller, G., Lechtchinsky, R., and Pfannenstiel, W. Nepal—Nested data-parallelism in Haskell. In *EuroPar'01—European Conf. on Parallel Processing*, Vol. 2150 of *LNCS*. Manchester, UK, Aug. 28–31, 2001, pp. 524–534.
11. Cole, M.I. *Algorithmic Skeletons: Structured Management of Parallel Computationg*. Research Monographs in Parallel and Distributed Computing. Cambridge, MA: The MIT Press, 1989.
12. Cook, A. Transformation and proof in a parallelising compiler. Ph.D. thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, 2002.
13. Darlington, J., Guo, Y., and To, H. Structured parallel programming: Theory meets practice. In *Research Directions in Computer Science*, R. Milner and I. Wand (Eds.), Cambridge University Press, 1996a.
14. Darlington, J., Guo, Y., To, H., and Yang, J. SPF: Structured parallel fortran. In *PCW'96—Intl. Parallel Computing Workshop*. Kawasaki, Japan, 1996b.
15. Doligez, D. and Leroy, X. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *POPL'93—Symp. on Principles of Programming Languages*. Charleston, SC, Jan. 1993, pp. 113–123.
16. Du Bois, A., Loidl, H.-W., and Trinder, P. Thread migration in a parallel graph reducer. In *IFL'02—Intl. Workshop on the Implementation of Functional Languages*, Vol. 2670 of LNCS. Madrid, Spain, Sept. 16–18, 2002.
17. Flanagan, C. and Nikhil, R. pHluid: The design of a parallel functional language implementation on workstations. In *ICFP'96—Intl. Conf. on Functional Programming*. Philadelphia, PA, May 24–26, 1996, pp. 169–179.
18. Frens, J. and Wise, D. Auto-blocking matrix multiplication, or tracking BLAS3 performance from source code. *PPoPP'97—Symp. on Principles and Practice of Parallel Programming*, **32**(7) (1997) 206–216.
19. Goldberg, B. Multiprocessor execution of functional programs. *Intl. J. of Parallel Programming*, **17**(5) (1988) 425–473.
20. Halstead, R. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, **7**(4) (1985) 106–117.
21. Hamdan, M. A combinational framework for parallel programming using algorithmic skeletons. Ph.D. thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, 2000.
22. Hammes, J., Lubeck, O., and Böhm, W. Comparing Id and Haskell in a Monte Carlo photon transport code. *J. of Functional Programming*, **5**(3) (1995) 283–316.
23. Hammond, K., King, D., Loidl, H.-W., Rebón, A., and Trinder, P. The HasPar performance evaluation suite for GPH: A parallel non-strict functional language. Technical report, 2000.
24. Hammond, K., Loidl, H.-W., and Partridge, A. Visualising granularity in parallel programs: A graphical winnowing system for Haskell. In *HPFC'95—Conf. on High Performance Functional Computing*. Denver, CO, April 10–12, 1995, pp. 208–221.
25. Hammond, K. and Michaelson, G. (Eds.). *Research Directions in Parallel Functional Programming*. Springer, 1999.
26. Hammond, K. and Rebón, A. HaskSkel: Algorithmic skeletons for Haskell. In *IFL'99—Intl. Workshop on the Implementation of Functional Languages*, Vol. 1868 of LNCS. Lochem, The Netherlands, Sept. 7–10, 1999.

27. Hartel, P., Feeley, M., Alt, M., Augustsson, L., Baumann, P., Beemster, M., Chailloux, E., Flood, C., Grieskamp, W., van Groningen, J., Hammond, K., Hausman, B., Ivory, M., Jones, R., Kamperman, J., Lee, P., Leroy, X., Lins, R., Loosemore, S., Röjemo, N., Serrano, M., Talpin, J.-P., Thackray, J., Thomas, S., Walters, P., Weis, P., and Wentworth, P. Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark. *J. of Functional Programming*, **6**(4) 1996.
28. Hernández, F., Peña, R., and Rubio, F. From GranSim to Paradise. In *SFP'00—Scottish Functional Programming Workshop*, Vol. 2 of Trends in Functional Programming. St. Andrews, Scotland, Jul 26–28, 2000, pp. 11–19.
29. Herrmann, C. The skeleton-based parallelization of divide-and-conquer recursions. Ph.D. thesis, University of Passau, 2000.
30. Hofmann, M. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, **7**(4) (2000) 258–289.
31. Holmerin, J. and Lisper, B. Development of parallel algorithms in data field Haskell. In *EuroPar'00—European Conf. on Parallel Processing*, Vol. 1900 of LNCS. Munich, Germany, Aug. 29–Sept. 1, 2000, pp. 762–766.
32. Hudak, P. Para-functional programming. *IEEE Computer*, **19**(8) (1986) 60–70.
33. Hughes, R. Why functional programming matters. *The Computer Journal*, **32**(2) (1989) 98–107.
34. Impala: 2001, Impala—(IMplicitly PArallel LAnguage Application Suite). <URL: `http://www.csg.lcs.mit.edu/impala/`>.
35. Karatsuba, A. and Ofman, Y. Multiplication of multi-digit numbers on automata. *Soviet. Phys. Dokl.*, (7) (1962) 595–596.
36. Kelly, P. *Functional Programming for Loosely-Coupled Multiprocessors*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.
37. Kesseler, M. The implementation of functional languages on parallel machines with distributed memory. Ph.D. thesis, Univ. of Nijmegen, 1996.
38. King, D., Hall, J., and Trinder, P. A strategic profiler for glasgow parallel Haskell. In *IFL'98—Intl. Workshop on the Implementation of Functional Languages*, Vol. 1595 of LNCS. London, UK, Sept. 9–11, 1998, pp. 88–102.
39. Kingdon, H., Lester, D., and Burn, G. The HDG-machine: A highly distributed graph-reducer for a transputer network. *Computer Journal*, **34**(4) (1991) 290–301.
40. Klusik, U., Loogen, R., Priebe, S., and Rubio, F. Implementation skeletons in Eden—Low-effort parallel programming. In *IFL'00—Intl. Workshop on the Implementation of Functional Languages*, Vol. 2011 of LNCS. Aachen, Germany, Sept. 4–7, 2000, pp. 71–88.
41. Klusik, U., Peña, R., and Rubio, F. Replicated workers in Eden. In *CMPP'00—Constructive Methods for Parallel Programming*. Ponte di Lima, Portugal. Nova Science Books, 2001.
42. LANL: 2001, Sisal Performance Data. <URL: `http://www.llnl.gov/sisal/PerformanceData.html`>.
43. Lauer, M. Computing by homomorphic images. In *Computer Algebra—Symbolic and Algebraic Computation*, B. Buchberger, G.E. Collins, R. Loos, and R. Albrecht (Eds.), Springer, 1982, pp. 139–168.
44. Lester, B. *The Art of Parallel Programming*. Prentice-Hall, 1993.
45. Lipson, J.D. Chinese remainder and interpolation algorithms. In *SYMSAM'71—Symp. on Symbolic and Algebraic Manipulation*, 1971, pp. 372–391.
46. Loidl, H.-W. LinSolv: A case study in strategic parallelism. In *Glasgow Workshop on Functional Programming*. Ullapool, Scotland, Sept. 15–17, 1997.
47. Loidl, H.-W. Granularity in large-scale parallel functional programming. Ph.D. thesis, Dept. of Computing Science, Univ. of Glasgow, 1998.
48. Loidl, H.-W. Load balancing in a parallel graph reducer. In *SFP'01—Scottish Functional Programming Workshop*, Vol. 3 of Trends in Functional Programming. Stirling, Scotland, Aug. 22–24, 2001, pp. 63–74.
49. Loidl, H.-W. The virtual shared memory performance of a parallel graph reducer. In *CCGrid/DSM 2002—Intl. Symp. on Cluster Computing and the Grid*. Berlin, Germany, May 21–24, 2002, pp. 311–318.
50. Loidl, H.-W., Morgan, R., Trinder, P.W., Poria, S., Cooper, C., Peyton Jones, S.L., and Garigliano, R. Parallelising a large functional program rr: Keeping LOLITA busy. In *IFL'97—Intl. Workshop on the Implementation of Functional Languages 1997*, Vol. 1467 of *LNCS*. St Andrews, Scotland, Sept. 10–12, 1997, pp. 198–213.

51. Loidl, H.-W., Scaife, N., Michaelson, G., and Trinder, P. Implementation designs for parallel functional languages. In: *PPDP'03—Intl. Conf. on Principles and Practice of Declarative Programming*. Uppsala, Sweden, Aug 27–29, 2003, Submitted.
52. Loidl, H.-W., Trinder, P., and Butz, C. Tuning task granularity and data locality of data parallel GpH programs. *Parallel Processing Letters*, **11**(4) (2001) 471–486.
53. Loidl, H.-W., Trinder, P., Hammond, K., Junaidu, S., Morgan, R., and Peyton Jones, S. Engineering parallel symbolic programs in GPH. *Concurrency—Practice and Experience*, **11**(12) (1999) 701–752.
54. Loogen, R. Programming language constructs. In *Research Directions in Parallel Functional Programming*. Springer, 1999, pp. 63–91.
55. Michaelson, G., Scaife, N., Bristow, P., and King, P. Nested algorithmic skeletons from higher order functions. *Parallel Algorithms and Applications*, **16** (2001) 181–206. Special Issue on High Level Models and Languages for Parallel Processing.
56. Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML (Revised)*. Cambridge, MA: MIT Press, 1997.
57. Mirani, R. High-level abstractions for parallel functional programming. Ph.D. thesis, Yale University, 1996.
58. Mirani, R. and Hudak, P. First-class schedules and virtual maps. In *FPCA'95—Conf. on Functional Programming Languages and Computer Architecture*. La Jolla, CA, June 26–28, 1995, pp. 78–85.
59. Mohr, E., Kranz, D., and Halstead Jr., R. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, **2**(3) (1991) 264–280.
60. MPI: MPI-2: extensions to the message-passing interface. Technical report, Univ. of Tennessee, Knoxville, 1997.
61. Nikhil, R. and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann Publishers, 2001.
62. Nöcker, E., Smetsers, J., van Eekelen, M., and Plasmeijer, M. Concurrent clean. In *PARLE'91—Parallel Architectures and Languages Europe*, Vol. 505 of LNCS. Veldhoven, The Netherlands, 1991, pp. 202–219.
63. Pelagatti, S. Task and data parallelism in P3L. In *Patterns and Skeletons for Parallel and Distributed Computing* F. Rabhi and S. Gorlatch (Eds.), Springer, 2002.
64. Peña, R. and Rubio, F. Parallel functional programming at two levels of abstraction. In *PPDP'01—Intl. Conf. on Principles and Practice of Declarative Programming*. Firenze, Italy, Sept. 5–7, 2001, pp. 187–198.
65. Pepper, P. Deductive derivation of parallel programs. In *Parallel Algorithm Derivation and Program Transformation*. Kluwer Academic Publishers, 1993, pp. 1–53.
66. Peyton Jones, S., Hall, C., Hammond, K., Partain, W., and Wadler, P. The glasgow Haskell compiler: A technical overview. In *Joint Framework for Information Technology Technical Conference*. Keele, UK, 1993, pp. 249–257.
67. Peyton Jones, S., Hughes, J., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C., and Wadler, P. Haskell 98: A non-strict, purely functional language, 1999. Available at <URL: `http://www.haskell.org/`>.
68. Plasmeijer, R., van Eekelen, M., Pil, M., and Serrarens, P. Parallel and distributed programming in concurrent clean. In *Research Directions in Parallel Functional Programming*. Springer, 1999, pp. 323–338.
69. Press, W., Teukolsky, S., Vetterling, W., and Flannery, B. *Numerical Recipes in C: The Art of Scientific Computing*. Chapt. LU Decomposition and Its Applications. Cambridge University Press, 2nd edition, 1992.
70. PVM: Parallel virtual machine reference manual, version 3.2. Univ. of Tennessee, 1993.
71. Quinn, M. *Parallel Computing*. McGraw-Hill, 1994.
72. Ridge, D., Becker, D., Merkey, P., and Sterling, T. Beowulf: Harnessing the power of parallelism in a pile-of-PCs. In *IEEE Aerospace Conference*, 1997, pp. 79–91.
73. Rubio, F. Programación Funcional Paralela Eficiente en Eden. Ph.D. thesis, Universidad Complutense de Madrid, Spain, 2001, in Spanish.
74. Sansom, P. and Peyton Jones, S. Generational garbage collection for Haskell, In *FPCA'93—Functional Programming Languages and Computer Architecture*. Copenhagen, Denmark, June 9–11, 1993, pp. 106–116.
75. Sansom, P. and Peyton Jones, S. Time and space profiling for non-strict, higher-order functional languages. In *POPL'95—Symp. on Principles of Programming Languages*. San Francisco, CA, Jan. 23–25, 1995, pp. 355–366.

76. Scaife, N., Michaelson, G., and Horiguchi, S. Comparative cross-platform performance results from a parallelizing SML compiler. In *IFL'01—Intl. Workshop on the Implementation of Functional Languages*, Vol. 2312 of *LNCS*. Stockholm, Sweden, Sept. 24–26, 2001, pp. 138–154.
77. Serot, J. Tagged-token data-flow for skeletons. *Parallel Processing Letters* **11**(4) (2001) 377–392.
78. Taylor, F. Parallel functional programming by partitioning. Ph.D. thesis, Univ. of London, 1996.
79. Trinder, P., Hammond, K., Loidl, H.-W., and Peyton Jones, S. Algorithm + strategy = parallelism. *J. of Functional Programming* **8**(1) (1998) 23–60.
80. Trinder, P., Hammond, K., Mattson Jr., J., Partridge, A., and Peyton Jones, S. GUM: A portable parallel implementation of Haskell. In *PLDI'96—Programming Language Design and Implementation*. Philadephia, PA, May 21–24, 1996, pp. 78–88.
81. Trinder, P., Loidl, H.-W., and Pointon, R. Parallel and distributed Haskells. *J. of Functional Programming* **12**(4/5) (2002) 469–510.
82. WWW-GPH. Glasgow Parallel Haskell, 2001. WWW page. <URL: `http://www.macs.hw.ac.uk/~dsg/gph/`>.