JFP **15** (4): 1–45, 2005. © 2005 Cambridge University Press
DOI: 10.1017/S0956796805005526 Printed in the United Kingdom

Parallel functional programming in Eden

RITA LOOGEN
Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, Hans-Meerwein-Straße, D-35032 Marburg, Germany
(*e-mail:* `loogen@mathematik.uni-marburg.de`)

YOLANDA ORTEGA-MALLÉN and RICARDO PEÑA-MARÍ
Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, C/ Juan del Rosal 8, E-28040 Madrid, Spain
(*e-mail:* `{yolanda,ricardo}@sip.ucm.es`)

Abstract

Eden extends the non-strict functional language Haskell with constructs to control parallel evaluation of processes. Although processes are defined explicitly, communication and synchronisation issues are handled in a way transparent to the programmer. In order to offer effective support for parallel evaluation, Eden's coordination constructs override the inherently sequential demand-driven (lazy) evaluation strategy of its computation language Haskell. Eden is a general-purpose parallel functional language suitable for developing sophisticated skeletons – which simplify parallel programming immensely – as well as for exploiting more irregular parallelism that cannot easily be captured by a predefined skeleton. The paper gives a comprehensive description of Eden, its semantics, its skeleton-based programming methodology – which is applied in three case studies – its implementation and performance. Furthermore it points at many additional results that have been achieved in the context of the Eden project.

1 Introduction

The exploitation of parallelism is a long pursued – and not yet convincingly met – goal in programming. There is a trade-off between the efficient exploitation of parallelism and the simplicity of the corresponding programs: the more control a language has on process management, and on communication and synchronisation aspects, the more complex and longer – and the less amenable for reasoning – are the resulting programs. Imperative parallel programming is a good example for this.

Functional programming means expressing algorithms at a high level of abstraction, thereby substantially simplifying the task of programming and increasing the programmer's productivity. Abstraction, expressiveness, referential transparency, and a clear semantic model lead to concise programs which can be developed in a short time as well as analysed or optimised with powerful formal methods.

Research in parallel functional programming tries to provide these advantages in the context of parallel program development as well. Following the idea of declarative programming, the main task of a parallel programmer should be to specify *what* has to be evaluated in parallel and *not how* the parallel evaluation has to be organised. Consequently, programmers should not deal with *low level* details of process management such as process creation and placement, communication and synchronisation, but instead concentrate on the decomposition of their problems into parallel tasks.

During the last decades many parallel functional languages have been designed and investigated, see e.g. the overviews given in Hammond & Michaelson (1999) and by Trinder *et al.* Pointon (2002). Most languages extend a pure functional computation language like Haskell or ML by a high-level coordination language. The coordination extensions range from purely implicit approaches where the exploitation of parallelism is transparent to the programmer and managed by a sophisticated compiler and runtime system, to completely explicit approaches where the programmer has to explicitly define the parallel behaviour, i.e. thread creation and/or communication and synchronisation. Moreover, parallel functional languages may differ in the supported paradigm: data parallelism or task parallelism.

In this paper we describe *Eden*, an extension of the non-strict functional language Haskell, covering its design, semantics, programming methodology, and implementation. Parallel programming at a high level of abstraction is achieved in Eden by defining processes explicitly, but at the same time keeping communication actions implicit. The programmer only specifies which data a process depends on. Sending and receiving data is performed automatically by the underlying Eden parallel runtime system. Thus, Eden can be classified as a semi-explicit approach to parallel functional programming. Task and data parallelism paradigms can be modelled in Eden, and high-level parallelism abstractions like skeletons can be defined. They simplify the task of parallel programming substantially.

Eden does not try to compete in performance with the combination of an imperative language like C or C++ and a parallel library such as OpenMp (Chandra, 2000), PVM (1993) or MPI (1997). These approaches achieve high performance at the price of investing a rather high effort in programming. Eden's intended users are in the first place functional programmers, which are willing to trade some performance for easier and shorter programming. As a consequence, the comparisons we provide in the related work section (see section 6) are between Eden and other parallel functional languages.

Another question that may be raised is why have we chosen a lazy language such as Haskell as the host language for a parallel extension. At first sight, eagerness appears to be more suitable than laziness for parallelism. In fact, the lazy option is a more challenging one. The main reason has been that we wanted to preserve all the advantages that lazy languages, and Haskell in particular, offer to programmers: non-strict functions, demand-driven evaluation, infinite objects, and monads, especially the state transformer monad and the IO monad which allow mutable arrays and purely functional IO, respectively. As an added value, we can use infinite lazy streams to model process communication. Circular topologies of processes connected by streams can be easily expressed as a set of mutually recursive equations, which do not deadlock as it would be the case in a strict language. We will show some examples of circular topologies in section 3.

The paper is meant as a wrap-up of the Eden project. It sums up the main design and implementation decisions, provides examples of skeletal parallel programming in Eden, and points at papers describing further achievements of the Eden project. Most of the information has been previously published in several workshop and conference papers. However, a comprehensive journal publication on Eden is overdue. The paper is intended to be self-contained, but a look into the rest of the given references will provide a deeper insight into details.

Eden's syntax, design decisions and the kernel part of its operational semantics are described in the next section. The subsequent section presents the skeleton-based programming methodology recommended for Eden programmers and shows skeleton definitions in Eden. In section 4, an overview of Eden's implementation is given. Section 5 discusses three case studies where the skeleton methodology is applied to parallelise functional programs. Runtime measurements show the speedups achieved by the Eden system for these example programs. The next section gives pointers to additional achievements of the Eden project, and Eden is compared with other parallel functional languages. Finally, conclusions are drawn.

2 The language

The parallel functional language Eden (Breitinger *et al.*, 1997b) extends the non-strict functional language Haskell (Peyton Jones & Hughes, 1999) with syntactic constructs for defining processes.

2.1 Basic constructs

Processes are defined by using the function

```
process :: (Trans a, Trans b) => (a -> b) -> Process a b
```

which embeds a function of type `a -> b` into a *process abstraction* of type `Process a b`, where `Process` is a new type constructor. The type class `Trans` will be explained later. A process abstraction `process (\x -> e)` of type `Process a b` defines the behaviour of a process having the parameter `x` with type `a` as input and the expression `e` with type `b` as output. The main difference between functions and process abstractions is that the latter, when instantiated, are executed in parallel.

Processes are created by using the infix instantiation operator

```
( # ) :: (Trans a, Trans b) => Process a b -> a -> b
```

which provides a process abstraction with actual input parameters. The evaluation of a *process instantiation* `(process (\x -> e1)) # e2` leads to the dynamic creation of a process together with its interconnecting communication channels. The instantiating or *parent process* is responsible for evaluating `e2` and for sending the resulting value v_2 via an implicitly generated channel to the new *child process* which evaluates the application `(\x -> e1)` v_2 and returns the result via another implicitly generated channel.

Eden is explicit about processes and their incoming and outgoing data, but it abstracts from the transfer of these data between processes and the necessary synchronisation. Thus, an Eden program defines a system of processes which exchange data via *unidirectional channels* which connect one writer to exactly one reader. The data transfers are automatically done by the system and need not be managed by the programmer. Communication channels are modelled by head-strict lazy lists, as in stream-based I/O. The predefined type class `Trans` declared by

```
class NFData a => Trans a where (...)
```

provides functions used internally for the transmission of values on communication channels. In principle, arbitrary values can be communicated. Their types must be instances of this type class. Corresponding instance declarations will automatically be derived by the compiler. The type class `Trans` is a subclass of the class `NFData` (Normal Form Data) because all process outputs are evaluated to normal form before being sent. In particular, communication is not demand-driven. Values are sent to receiver processes without prior requests until a receiver notifies that it does not need input values any more. In general terms, the system will *push* instead of *pull* information. Evaluation to normal form and eager communication deviate from Haskell's demand-driven evaluation, but are essential to support parallelism (Klusik *et al.*, 2001a).

Example 1 (*mergesort*)

The following function creates a parallel sorting network which transforms an input stream into a sorted output stream by subsequently merging sorted sublists with increasing length:

```
mergesort :: (Ord a, Trans a) => [a] -> [a]
mergesort []   =  []
mergesort [x]  =  [x]
mergesort xs   =  sortmerge (process mergesort # xs1)
                            (process mergesort # xs2)
         where (xs1,xs2) = unshuffle xs
```

Streams with at least two elements are split into two sub-streams using the function `unshuffle`. The sub-streams are sorted by recursive instantiations of `mergesort` processes. The sorted sublists are coalesced into a sorted result list using the function `sortmerge` which is an ordinary Haskell function like `unshuffle` (both functions are not shown here). The context `Ord a` ensures that an ordering is defined on type `a`. The process system generated when `mergesort` is applied to a list with more than two elements is a binary tree. ◁

If the output of a process is a tuple, an independent thread will be created for each component of the tuple and the result of its evaluation will be sent on a separate channel. The connection points of channels to processes are called *inports* on the receiver side and *outports* on the sender side. There is a one-to-one correspondence between the threads and the outports of a process while data that is received via the inports is shared by all threads of a process. Only the first level of tuple outputs will be evaluated concurrently, i.e. (e_1, e_2, e_3) yields three threads while $(e_1, (e_2, e_3))$

would be evaluated by two threads. Analogously, several threads will be created in a parent process for tuple inputs of a child process.

Example 2 (*several outputs*)

The following expression is a simple process abstraction defining a process with two input streams and two output streams:

```
process (\ (xs,ys) -> (zipWith (+) xs ys, zipWith max xs ys))
```

It has the type `(Num a, Ord a) => Process ([a],[a]) ([a],[a])`. When instantiated with a pair of lists, the process will deliver two outputs produced by two independent threads. The first thread computes the element-wise sum of the input streams while the second thread compares the input streams element-wise and outputs the maximum values. Both threads share the input streams. ◁

Process abstraction and instantiation could have been coalesced into a single binary operator

```
(#')    :: (Trans a, Trans b) => (a -> b) -> a -> b
f #' x =  let pf = process f in pf # x
```

which is a parallel version of the Haskell operator `($):: (a -> b) -> a -> b` for function application, and Eden could have been defined with only this operator. We prefer, however, the clear distinction between process abstractions and functions, which is also reflected in the new type `Process a b`. Process instantiations and abstractions represent two different sides of process creation: the parent and the child side, respectively. While process instantiation represents process creation on the parent side, process abstractions represent the definition of a child process, independently from its creation. In section 2.4, the operational semantics of Eden is, however, discussed for a core language with the operator `#'` only.

2.2 *Parallelism vs. laziness*

Laziness has many advantages over eager evaluation, such as demand-driven evaluation, infinite data structures, and the natural handling of partially available data. The latter is especially important for parallel evaluation, since communication channels can be modelled by lazy lists and circular topologies of processes connected by such lists can be created. The demand-driven evaluation within processes is important to avoid unnecessary computations. In particular, we will avoid unnecessary blocking on non-available input, and consuming available but non-needed data.

Nevertheless, lazy evaluation is changed to eager evaluation in two cases: processes are eagerly created, and they produce their output even if it is not demanded. Eager process creation means that process instantiations in local definitions (`let`- and `where`-blocks) are treated in a special way: instead of creating a closure representation in the heap, the process is immediately created. These modifications aim at increasing the parallelism degree and at speeding up the distribution of the computation. Even though, it is sometimes still necessary to produce additional demand in order to unfold certain process systems. In many cases the programmer may experience *distributed sequentiality*, because demand-driven (lazy) evaluation activates the parallel evaluation only when its result is already needed to continue the overall computation, i.e. it immediately waits for the result. This situation is illustrated by the next example:

Example 3 (*parallel map*)
Replacing the function application in the `map` function:

```
map        :: (a -> b) -> [a] -> [b]
map f xs   =  [f x | x <- xs]
```

by a process instantiation, leads to a simple parallel `map` function, in which a different process is created for each element of the input list:

```
map_par_1        :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
map_par_1 f xs =  [process f # x | x <- xs]
```

The process abstraction `process f` determines that the input parameter `x`, as well as the result value, will be transmitted via channels. The types `a` and `b` should therefore belong to the class `Trans`.
The problem with this definition is that for instance the expression `sum (map_par_1 square [1..10])` will create 10 processes, but only one after the other as demanded by the `sum` function which sums up the elements of a list of numbers. Consequently, the computation will not speed up by "parallel" evaluation, but slow down because of the process creation overhead added to the sequential evaluation. ◁

Fortunately, it is easy to impose additional demand on expressions, e.g. by using evaluation strategies (Trinder *et al.*, 1998). Evaluation strategies (or simply strategies) are functions which control the evaluation of expressions without producing a result value[1]. They are applied by means of a function `using`, which first applies the strategy to the input, and then returns the value of the input:

```
type Strategy a = a -> ()

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

The evaluation itself is enforced by the operator `seq :: a -> b -> b` – defined in the Haskell prelude – which evaluates its first argument to weak head normal form (whnf) and then returns its second argument. As above, it is usually introduced to overrule laziness for performance reasons.

Example 4 (*parallel map with demand control*)
The traversal of the spine of a list is achieved by the strategy `spine` defined as follows:

[1] Originally, they have been introduced to specify parallel behaviour of programs. However, we only use them to control sequential evaluation.

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

With this strategy, the map_par function of Example 3 can be modified in such a way that all processes are created as soon as there is demand for the evaluation of the list.

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

The spine strategy eagerly evaluates the spine of the process instantiation list. In this way all processes will immediately be created. ◁

The function map_par defines a basic scheme of parallel evaluation which eagerly creates a set of independent processes. Such parallelism abstractions are an important part of the Eden methodology and we elaborate on them in section 3.

2.3 Coordination aspects

In this subsection we provide an informal introduction to Eden’s semantics. A more formal treatment is given in section 2.4.

Concurrent activities (processes and threads) are initiated when a process instantiation is evaluated. We answer the following questions:

- When will a process instantiation be evaluated (a process be created)?
- To which degree will output expressions be evaluated?
- When will the results of output expressions be transmitted between processes?

Process creation. Local definitions of process instantiations in `let` or `where` blocks, i.e. bindings of the form `outp = pabs # inps`, lead to the immediate creation of a process when they are accessed during the evaluation. To look at this the other way round, the instantiation of a process will be postponed when it does not appear at “top-level”, namely when it occurs, for instance,

- as the body of a lambda abstraction or function definition: The instantiation takes place as soon as the abstraction or function is fully applied and evaluated.
- as a component of a data structure: Its instantiation takes place as soon as the corresponding component of the data structure is evaluated.
- in a branch of a case analysis: It will only be instantiated if this branch is selected and evaluated.

A new process has a thread for each of its outports; moreover, the *parent* process will initiate a thread to serve each of the inports of the newly created *child* (communication from parent to child).

Eden has been designed to run in distributed settings, therefore a common shared memory is not assumed. The concept of a virtually shared global graph is avoided, to save the administration costs while paying the price of possibly duplicating work. All bindings needed for the evaluation of the free variables in a process body will be copied from the parent to the child. However, when the evaluation of the process body depends on a value to be communicated from some other process, the process creation is delayed until the necessary communications and instantiations have taken place.

Example 5 (*Delayed process creation*)
In the Eden expression

```
let  x_out = (process x) # x_in   -- process X
in   combine x_out
              ((f x_out)  # y_in)  -- process Y
```

process X will be created when the let-expression is evaluated, while the second process instantiation `(f x_out) # y_in)` yielding process Y is only performed when it is needed by `combine`. If process Y is created before process X has completely delivered its output, process Y's creation must be postponed until X's output is available, because channel inports cannot be copied from parent to child heaps. ◁

Communication. Outport expressions are evaluated to normal form, except for expressions with a function type, which are evaluated to weak head normal form. In that latter case it is mandatory to copy – from the producer to the consumer – all the bindings needed for the evaluation of the free variables in the abstraction. As in the case of process creation, this copy can take place only if there is no dependency on pending communications.

Results of the evaluation of outports are sent to the connected inports as soon as they are available. A channel is closed when the output value has been completely transmitted to the receiver.

Synchronisation. If a thread needs some input value that has not been received, the thread is suspended until the corresponding producer sends the desired data. Notice that communication through channels has non-blocking sending, but blocking reception, and that process synchronisation is achieved exclusively by exchanging data through the communication channels.

Termination. The execution of an Eden process is controlled by the evaluation of its outports, so that execution will end as soon as the process has no more outports or when its output is detected to be unnecessary (during garbage collection). Upon termination of a process, its inports are closed immediately. Then the corresponding outports will be closed in the producer processes, so that termination cascades through the process network.

2.4 Formal semantics

The non-strict semantics of the Haskell subset is preserved in Eden. Considering just the input-output behaviour, process abstractions and instantiations could be identified with function definitions (λ-abstractions) and applications, respectively. However, this view completely neglects parallelism and ignores issues like process

Syntax		Restricted Syntax	
E ::=	x	x	identifier
|	$\lambda x.E$	$\lambda x.E$	λ-abstraction
|	$E_1 E_2$	$x_1 x_2$	application
|	E_1 `#'` E_2	x_1 `#'` x_2	process instantiation
|	`let` $\{x_i = E_i\}_{i=1}^n$ `in` E	`let` $\{x_i = E_i\}_{i=1}^n$ `in` x	local declaration

Fig. 1. Eden core syntax.

creation and communication. In the following we define an operational semantics in the style of Baker-Finch *et al.* (2000) which is based on Launchbury's *natural* semantics for lazy evaluation (Launchbury, 1993). The semantics handles process creation and communication and is precise about expression scheduling and evaluation order. Therefore it is, for instance, suitable for measuring the amount of speculative computation produced during program execution.

For simplicity, we define the semantics for a core language consisting of the untyped λ-calculus just extended with local definitions (`let`) and the derived process instantiation operator (`#'`). The (abstract) syntax, based on variables $x \in Var$ and expressions $E \in Exp$, is given in Figure 1.

Following Launchbury (1993), we assume a general renaming of variables which avoids name clashes during expression evaluation. Moreover, the language is normalised to a restricted syntax (see Figure 1) where all subexpressions, except for the body of λ-abstractions, are replaced by variables defined in `let`-expressions. An expression $(E_1\ E_2)$ with non-variable sub-expressions E_1 and E_2 will e.g. be normalised to `let` $x = \tilde{E}_1; y = \tilde{E}_2$ `in` $(x\ y)$, where $\tilde{E}_1$ and $\tilde{E}_2$ are the results of normalising E_1 and E_2. In contrast to Launchbury, we replace not only the argument expression in applications by variables, but also the functional expression as well as the body of let-expressions. In this way, all subexpressions of any expression may be shared and will be evaluated at most once. Additionally, the semantic definition of evaluating an application (lazily) is simplified.

In Launchbury (1993), closures are modelled as variable-to-expression bindings which are collected in a heap representing the program space. In Baker-Finch *et al.* (2000), such bindings are also used to model threads, which share a unique heap and are executed by the available processors. Due to the distributed nature of Eden, each process is represented by a separate heap. Distinct variables $c \in \mathscr{C}han$ are introduced to represent communication channels, where $\mathscr{C}han$ denotes the set of channel identifiers. A process is represented by a pair $\langle p, H\rangle$, where p is a process identifier and H is the bindings heap. As each binding is considered a potential thread, a label indicates the thread's state: $x \overset{\alpha}{\mapsto} e$, where $\alpha ::= I|A|B$ corresponds to *Inactive* (either not yet demanded or already completely evaluated), *Active* (or demanded), and *Blocked* (demanded but waiting for the value of another binding), respectively. Channel identifiers can appear on either side of a binding. On the left-hand side, a channel identifier represents an outport of the corresponding process. A channel identifier on the right-hand side denotes an inport of the process.

$$H + \{x \overset{I}{\mapsto} v\} : \theta \overset{A}{\mapsto} x \quad \longrightarrow \quad H + \{x \overset{I}{\mapsto} v, \theta \overset{A}{\mapsto} v\} \qquad \textbf{(value)}$$

$$\text{if } E \notin \mathit{Val},\ H + \{x \xmapsto{IAB} E\} : \theta \overset{A}{\mapsto} x \quad \longrightarrow \quad H + \{x \xmapsto{AAB} E, \theta \overset{B}{\mapsto} x\} \qquad \textbf{(demand)}$$

$$H : x \overset{A}{\mapsto} x \quad \longrightarrow \quad H + \{x \overset{B}{\mapsto} x\} \qquad \textbf{(blackhole)}$$

$$\text{if } E \notin \mathit{Val},\ H + \{x \xmapsto{IAB} E\} : \theta \overset{A}{\mapsto} x\,y \quad \longrightarrow \quad H + \{x \xmapsto{AAB} E, \theta \overset{B}{\mapsto} x\,y\} \ \textbf{(app-demand)}$$

$$H + \{x \overset{I}{\mapsto} \lambda z.E\} : \theta \overset{A}{\mapsto} x\,y \quad \longrightarrow \quad H + \{x \overset{I}{\mapsto} \lambda z.E, \theta \overset{A}{\mapsto} E[y/z]\} \qquad \textbf{(}\beta\textbf{-reduction)}$$

$$H : \theta \overset{A}{\mapsto} \texttt{let } \{x_i = E_i\} \texttt{ in } x \quad \longrightarrow \quad H + \{y_i \overset{I}{\mapsto} \sigma(E_i)\}_{i=1}^{n} + \{\theta \overset{A}{\mapsto} \sigma(x)\} \ \textbf{(let)}$$

$$\text{where } \mathit{fresh}(y_i)\ (1 \leq i \leq n) \text{ and } \sigma := [y_1/x_1, \ldots, y_n/x_n]$$

Fig. 2. Local transition rules.

In the following, we will use $x, y, z \in \mathit{Var}$ for "ordinary variables", $c \in \mathscr{C}\mathit{han}$ for channels, where $\theta \in \mathit{Var} \cup \mathscr{C}\mathit{han}$, and p and q for process identifiers.

The semantics consists of a two-level transition system: the lower level handles local effects within processes, while the upper level describes the effects global to the whole system (the set of all parallel processes), like process creation and data communication. In order to avoid writing multiple similar transition rules, we allow a binding to appear with several labels, corresponding to the different possibilities admitted by the rule. Thus, if for instance $x \xmapsto{IAB} E$ appears on the left-hand side of a rule, and $x \xmapsto{ABA} E'$ on the right-hand side, this means that the thread corresponding to the closure $x \mapsto E$ becomes active in the case it was either inactive or blocked, while it becomes blocked if it was previously active. Besides, notation $H + \{x \overset{\alpha}{\mapsto} E\}$ means that the heap H is extended with the binding $x \overset{\alpha}{\mapsto} E$, while $H : x \overset{\alpha}{\mapsto} E$ means that the binding for x is the one which guides the application of the corresponding rule.

2.4.1 *Local process evolution*

Local transitions express the reduction of an active thread in the context of a single process. This internal activity affects only the corresponding heap: bindings may be created, modified, blocked or activated. The evaluation of an expression terminates when a weak head normal form (whnf) value ($v \in \mathit{Val}$) has been reached. Local transitions take the form $H : x \overset{A}{\mapsto} E \longrightarrow H'$, which is read as "the evaluation of the active thread $x \overset{A}{\mapsto} E$ transforms the heap $H + \{x \overset{A}{\mapsto} E\}$ into H'". The rules given in Figure 2 express how lazy evaluation progresses under demand.

Whenever the evaluation of an expression finishes, the resulting whnf value is shared with other bindings. By applying the rule **(value)**, the value is copied and bound to the demanding variable. The corresponding binding remains active, because bindings blocking on it must be unblocked before the binding is deactivated. Unblocking and deactivation is performed by scheduling rules which will be

$$\frac{\begin{array}{l}\{H_p^{(i,1)} + H_p^{(i,2)} : \theta_p^i \overset{A}{\mapsto} E_p^i \longrightarrow H_p^{(i,1)} + K_p^{(i,2)} \\ \quad \mid H_p = H_p^{(i,1)} + H_p^{(i,2)} + \{\theta_p^i \overset{A}{\mapsto} E_p^i\} \text{ and } \theta_p^i \overset{A}{\mapsto} E_p^i \in \mathcal{ET}(H_p)\}_{i=1}^{n_p}\end{array}}{\langle p, H_p\rangle \overset{par}{\Longrightarrow}_p \langle p, (\cap_{i=1}^{n_p} H_p^{(i,1)}) \cup (\cup_{i=1}^{n_p} K_p^{(i,2)})\rangle}$$

Fig. 3. Rule **parallel-p**: concurrent thread evolution within a single process p.

introduced later. The rule **(demand)** handles the case where the demand is issued before the value has been obtained; then, the demanding binding is blocked while the demanded one is activated (or remains blocked if it was already blocked). The rule **(blackhole)** deals with cyclic dependencies. In an application, demand is propagated to the variable corresponding to the abstraction (rule **(app-demand)**). The application of the obtained abstraction is specified by the rule **(β-reduction)**. A local declaration introduces new bindings in the heap (rule **(let)**); all of them are labelled as inactive, as they have not been demanded yet. To avoid name clashes, the local variables x_i are renamed by fresh variables y_i using the substitution σ.

All these local evolutions are considered to occur simultaneously, entwined in a parallel (global) step. At the lower level, we consider the evolution of parallel threads inside a process with a common heap H_p. The corresponding rule **(parallel-p)** is given in Figure 3. Let $\mathscr{ET}(H_p)$ denote the set of active threads in process p that may evolve (as defined in Subsection 2.4.3), and $n_p = |\mathscr{ET}(H_p)|$ be the number of "evolutionary" threads. All threads share a common heap H_p and modify this heap without any interference. Therefore, we can decompose the common heap, for each thread i with $1 \leqslant i \leqslant n_p$, into three parts: $H_p^{(i,1)}$ is the part of H_p that remains unchanged during the application of the corresponding local rule, while $H_p^{(i,2)}$ is the part that will be modified into $K_p^{(i,2)}$. The third part is the active thread binding which must be a member of $\mathscr{ET}(H_p)$. The parallel execution of the active threads keeps all parts of the heap which are not changed at all $(\cap_{i=1}^{n_p} H_p^{(i,1)})$ and adds every modification that has been done by any rule $(\cup_{i=1}^{n_p} K_p^{(i,2)})$.

2.4.2 Global system evolution

The upper level defines global transitions between process systems (S), i.e. sets of processes. A global transition takes the general form:

$$\{\langle p, H_p\rangle\}_{p\in S} \overset{\diamond}{\Longrightarrow} \{\langle p, H_p'\rangle\}_{p\in S\cup S'}$$

where each heap H_p (associated to a process p in S) is transformed to H_p', while new processes (in S') may be created. The diamond $\diamond$ is a place-holder for the name of the rule. As a first global transition we consider the parallel evolution of all the processes within the system S, shown in Figure 4.

After each process has internally evolved, the following tasks have to be done at the system level: process creation, interprocess communication and state management (thread unblocking and deactivation). In general, these tasks imply multiple single

$$\frac{\{\langle p, H_p\rangle \overset{par}{\Longrightarrow}_p \langle p, H'_p\rangle\}_{\langle p, H_p\rangle \in S}}{S \overset{par}{\Longrightarrow} \{\langle p, H'_p\rangle\}_{\langle p, H_p\rangle \in S}}$$

Fig. 4. Rule **(parallel)**: parallel process evolution.

Process creation:

$$\begin{aligned}(S, \langle p, H + \{\theta \overset{\alpha}{\mapsto} x\texttt{\#'}y\}\rangle) \xrightarrow{pc} (S,\ &\langle p, H + \{\theta \overset{B}{\mapsto} c_1, c_2 \overset{A}{\mapsto} y\}\rangle,\\ &\langle q, \eta(\mathtt{nh}(x, H)) + \{c_2 \overset{A}{\mapsto} \eta(x)\, z, z \overset{B}{\mapsto} c_1\}\rangle)\end{aligned}$$

if $noChan(\mathtt{nh}(x, H)) = True$ and q, z, c_1, c_2 are fresh (channel) variables / identifiers and substitution η replaces all variables by fresh ones.

Fig. 5. Process creation rule.

steps, each involving at most two processes. Let S be a process system, and $\diamond$ the name of a rule, for each single-step rule $S \overset{\diamond}{\longrightarrow} S'$ we can define a *multi-step rule* $S \overset{\diamond}{\Longrightarrow} S'$ satisfying:

1. $S \overset{\diamond}{\longrightarrow}^* S'$ and,
2. there is no S'' such that $S' \overset{\diamond}{\longrightarrow} S''$.

The application of a single-step rule $\diamond$ to a binding in some process, may enable the application of the same rule $\diamond$ to other bindings – in the same or in other processes – but it can never disable applications of rule $\diamond$ which were enabled before the former application.

Process creation. When evaluating E_1#'E_2, a new child process q is created and fed with the value of E_2 by its parent process p via an input channel. It evaluates $E_1\ E_2$ and returns the result (to its parent) via an output channel. The following diagram illustrates this:

$$\begin{tikzcd}
 & & \boxed{p} \arrow[dd, shift right, "E_2"'] \\
\boxed{p} \arrow[r, "E_1\#'E_2"] & {} & \\
 & & \boxed{q} \arrow[uu, shift right, "E_1E_2"']
\end{tikzcd}$$

The process creation can only take place, if the body E_1 does not depend on a channel variable, i.e. on some value which has not been communicated yet. The rule for process creation is given in Figure 5. Speculative parallelism is achieved by applying this rule to inactive bindings.

For a normalised expression x#'y, the argument y is evaluated by the parent (p), while the body x as well as the application, $x\,y$, are evaluated by the new-born child (q). Two channels are introduced: one is used to communicate the value of the argument from the parent to the child process; the result of the child is returned to the parent process via the other channel. The two bindings with the new channel

Communication:

$$
\begin{array}{lll}
(S, & \langle p, H_p + \{c \overset{\alpha}{\mapsto} v\}\rangle, & \langle q, H_q + \{\theta \overset{B}{\mapsto} c\}\rangle) \quad \xrightarrow{com} \\
(S, & \langle p, H_p\rangle, & \langle q, H_q + \{\theta \overset{A}{\mapsto} \eta(v)\} + \eta(\mathsf{nh}(v, H_p))\rangle)
\end{array}
$$

if $noChan(\mathsf{nh}(v, H_p)) = True$ and η introduces fresh names for all variables.

Fig. 6. Value communication rule.

variables on their left-hand side are active and will be evaluated if there are enough resources.

The initial heap of the child process contains all the bindings needed for the evaluation of the dependent variables in the process body; these are copied, in an inactive state, from the parent to the child heap by the function nh (needed heap): $\mathsf{nh}(x, H)$ collects all the bindings in H that are reachable from x. A renaming η with fresh variables is applied to avoid name clashes. As mentioned before, a process creation is blocked if there is a dependency on values that have to be communicated. A predicate *noChan* checks whether the heap needed by the process body does not depend on a channel variable.

The process creation rule introduces new bindings and modifies only the one corresponding to the `#'`-expression. As a consequence, the creation of a process cannot disable the creation of other processes. On the contrary, it may even bring new top-level `#'`-expressions. Even then, the number of processes that can be created in one multi-step is always finite, and thus, the corresponding multi-step rule $\overset{pc}{\Longrightarrow}$, which carries out every possible process creation, is well defined.

Communication. The rule for value communication between processes is given in Figure 6. In this simple calculus, values are always abstractions, and it is mandatory to copy – from the producer's heap to the consumer's heap – all the bindings needed for the evaluation of the dependent variables in the communicated abstraction. Again, this copy can only take place if the abstraction does not depend on pending communications; a renaming substitution (η) is applied to the transferred heap, and bound variables are replaced by fresh variables.

Although a communication may enable additional ones, this never leads to an infinite number of communications (in one system step) because there is always only a finite number of communication channels in the system. The corresponding multi-step rule $\overset{com}{\Longrightarrow}$, which carries out every possible communication, is therefore well defined. The order of communications is not relevant, because variables that are already bound to values are not affected by communications.

Scheduling. Once all the enabled process creations and communications have been done, the following tasks have to be executed:

- Unblocking bindings depending on a variable bound to a whnf value meanwhile.
- Deactivating bindings to values in whnf.
- Blocking process creations that could not be executed.

WHNF unblocking:

$$(S, \langle p, H + \{x \overset{A}{\mapsto} v, \theta \overset{B}{\mapsto} E_B^x\}\rangle) \xrightarrow{wUnbl} (S, \langle p, H + \{x \overset{A}{\mapsto} v, \theta \overset{A}{\mapsto} E_B^x\}\rangle)$$

WHNF deactivation:

$$(S, \langle p, H + \{\theta \overset{A}{\mapsto} v\}\rangle) \xrightarrow{deact} (S, \langle p, H + \{\theta \overset{I}{\mapsto} v\}\rangle)$$

blocking process creation:

$$(S, \langle p, H + \{\theta \overset{IA}{\mapsto} x\#'y\}\rangle) \xrightarrow{bpc} (S, \langle p, H + \{\theta \overset{B}{\mapsto} x\#'y\}\rangle)$$

Fig. 7. Rules for scheduling.

The corresponding rules are given in Figure 7, where E_B^x denotes an expression that is *immediately blocked on the variable* x, i.e. either x or $x\,y$, with y being an arbitrary variable.

The sequential execution of the rules in Figure 7 gives a new global rule:

$$\overset{Unbl}{\Longrightarrow} = \overset{wUnbl}{\Longrightarrow}; \overset{deact}{\Longrightarrow}; \overset{bpc}{\Longrightarrow}.$$

For each multi-step rule in $\overset{Unbl}{\Longrightarrow}$ it can be proven: (1) that a single step never disables any other rule application enabled before, and (2) that the number of steps is always finite.

The global system evolves by applying each of the global transition rules that have been introduced so far:

$$\overset{sys}{\Longrightarrow} = \overset{comm}{\Longrightarrow}; \overset{pc}{\Longrightarrow}; \overset{Unbl}{\Longrightarrow}$$

The order of rule applications is not arbitrary; a communication may enable a pending process creation, but not the other way around, because when a new process is created no communication can take place without at least a local **value** transition.

Finally, each transition step of the system is defined as follows:

$$\Longrightarrow = \overset{par}{\Longrightarrow}; \overset{sys}{\Longrightarrow}.$$

2.4.3 Speculative parallelism

The evaluation of an Eden program may give rise to different computations. The exact amount of speculative parallelism depends on the number of available processors, the scheduler decisions and the speed of basic instructions. Hence, the execution of a program may range from reducing the speculation to the minimum – only what is effectively demanded is computed – to expanding it to the maximum – every speculative computation is carried out. While the former would be equivalent to executing the program on a single processor with the scheduler giving priority to the demand originated by the main expression, the latter would correspond to having an unlimited set of processors for evaluating the output of every generated process. Moreover, if a reduction sequence for a program expression E is defined as a – finite or infinite – sequence of configurations

$$\langle p_0, \{main \overset{A}{\mapsto} E\}\rangle \Longrightarrow^{*} \langle p_0, H + \{main \overset{I}{\mapsto} v\}\rangle, \langle p_1, H_1\rangle, \ldots, \langle p_n, H_n\rangle \Longrightarrow^{*}$$

then, in a *minimal* semantics, the final configuration is the first one where the main variable becomes inactive, while in the case of a *maximal* semantics, the execution continues until every process that has been created is finished or blocked, i.e. until there are not active threads in the system.

Minimal semantics. We need a way to give preference to demands originating from the main expression. An auxiliary function pre (preference) not shown here collects all bindings that are demanded by a variable x (or by a channel c) for its immediate evaluation within a process $\langle p, H\rangle$ and a system S. For a minimal evaluation, we start from the variable *main*:

$$\mathsf{pm}(S) = \mathsf{pre}(main, \langle p_0, H_0\rangle, S)$$

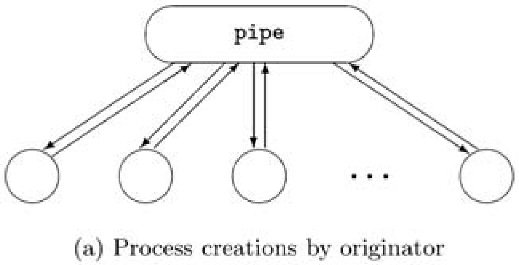
where $\langle p_0, H_0\rangle$ is the main process, i.e. the one which contains the variable *main*. Finally, we define the set of *evolutionary threads* of a heap H (used in the rule **parallel-p**) as $\mathscr{E}\mathscr{T}_{min}(H) = H \cap \mathsf{pm}(S)$. Notice that in this semantics a process can be speculatively created, but its body will be evaluated only if its output is demanded.

Maximal semantics In this case, all the active bindings in the system evolve in parallel at each step. We simply define $\mathscr{E}\mathscr{T}_{max}(H)$ as the set of all active bindings in H.

A first version of this operational semantics was presented in Hidalgo-Herrero & Ortega-Mallén (2001). In Hidalgo-Herrero & Ortega-Mallén (2002) it was re-elaborated and extended with streams for communication and the language features introduced in the next subsection: dynamic channels and nondeterminism. For more details, such as correctness proofs, examples and applications, the reader is referred to Hidalgo-Herrero (2004). A denotational semantics for Eden has been defined in Hidalgo-Herrero & Ortega-Mallén (2003). A more detailed version extended with communication streams is given in Hidalgo-Herrero (2004). This semantics addresses three different aspects: (1) functionality: the final value computed; (2) parallelism: the process system topology and its corresponding interactions generated by the computation; and (3) distribution: the degree of speculation.

2.5 Extra-functional features

To make programming in Eden more convenient and to improve the expressive power of the language, two additional constructs have been added to the language: *dynamic reply channels* which simplify the creation of complex communication topologies and reactive process systems, and *many-to-one communication* using a non-deterministic fair `merge` process.



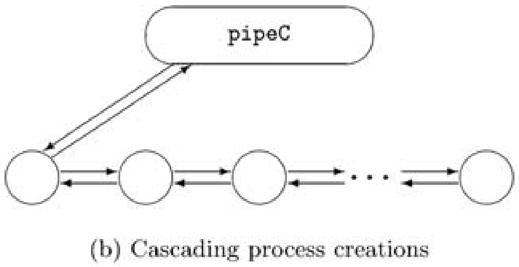


Fig. 8. Pipeline topologies.

2.5.1 Dynamic channels

With the Eden constructs introduced up to now, communication channels are only established between parent and child processes during process creation. This results in purely hierarchical process topologies.

Example 6 (*Pipeline*)
Consider the following straightforward definition of a process pipeline[2]:

```
pipe :: Trans a => [a -> a] -> a -> a
pipe []     vals   = vals
pipe (p:ps) vals   = pipe ps (process p # vals)
```

The process evaluating a `pipe` application will create all processes of the pipeline and consequently, the topology shown in Figure 8(a) will be produced. Data is passed from one pipeline stage to the next via the parent process, which causes a big communication overhead and contradicts the intention of the programmer. In an alternative definition each pipeline process creates its successor process.

```
pipeC :: Trans a => [a->a] -> a -> a
pipeC [] vals = vals
pipeC ps vals = process (generatePipe ps) # vals

generatePipe :: Trans a => [a->a] -> a -> a
generatePipe [p]    vals = p vals
generatePipe (p:ps) vals = (process (generatePipe ps)) # (p vals)
```

[2] An equivalent, much shorter definition of `pipe` would be `pipe = flip (foldl ((#).process))` but for didactic reasons we prefer the explicit version.

The cascading pipe `pipeC` is defined using the auxiliary function `generatePipe` which will be executed by each pipeline process causing the creation of a successor process only if its list parameter contains at least two elements. Note that `generatePipe` will always be called with a non-empty list parameter. Evaluating an application of `pipeC` yields the process topology shown in Figure 8 (b). This is still not an optimal realisation of a pipeline, because the pipeline results must be passed through all pipeline stages before reaching the originating process, but it is the best we can achieve with only tree-shape process topologies. ◁

To establish direct channel connections between arbitrary processes, Eden has been extended with *dynamic channel creation.* An Eden process may explicitly generate a new *dynamic input channel* whose name can be communicated to another process. The receiving process may then either use the name to return some information directly to the sender process (*receive and use*), or pass the channel name further on to another process (*receive and pass*). Both possibilities exclude each other, and a runtime error will occur if a channel name is used more than once.

Eden introduces a unary type constructor `ChanName` for the names of dynamically created channels. Moreover, it adds an operator

```
new :: Trans a => (ChanName a -> a -> b) -> b
```

Evaluating an expression `new (\ ch_name ch_vals -> e)` has the effect that a new channel name `ch_name` is declared as reference to the new input channel via which the values `ch_vals` will eventually be received in the future. The scope of both is the body expression `e`, whose value is the result of the whole expression. The channel name must be passed to another process to establish the direct communication. A process receiving a channel name `ch_name`, and wanting to reply through it, uses the function

```
parfill :: Trans a => ChanName a -> a -> b -> b
```

Evaluating an expression `parfill ch_name e1 e2` means: Before `e2` is evaluated, a new concurrent thread for the evaluation of `e1` is generated, whose normal form result is transmitted via the dynamic channel. The result of the overall expression is `e2`. The generation of the new thread is a side effect. Its execution continues independently from the evaluation of `e2`.

Example 7 (*Pipeline, continued*)

By passing a dynamic reply channel through the pipeline the last process can directly send the final results to the originator process. This yields the intended process topology shown in Figure 9

```
pipeD :: Trans a => [a->a] -> a -> a
pipeD [] vals = vals
pipeD ps vals = new (\ chan res ->
                    (process (generatePipeD ps chan)) # vals
                    `seq` res)

generatePipeD :: Trans a => [a->a] -> ChanName a -> a -> ()
generatePipeD [p]    c vals = parfill c (p vals) ()
generatePipeD (p:ps) c vals = (process (generatePipeD ps c)) # (p vals)
```

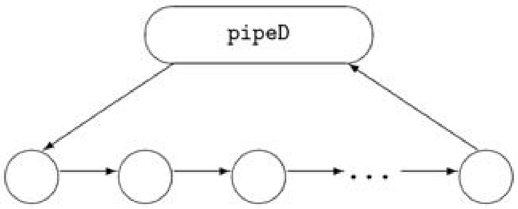


Fig. 9. Intended pipeline topology.

It is obvious that the amount of communications is substantially reduced in this version. ◁

The ring topology described in section 3 is another example where the direct dynamically-established channel connections increase the performance substantially.

Although dynamic channels are a non-functional feature, they do not extend the expressiveness of Eden. They have been introduced to improve the efficiency of programs, but any program that correctly uses dynamic channels can be rewritten into a program that shows the same input/output behaviour with only hierarchical communication. This can be seen from the following argumentation. There will always be a “first” dynamic channel whose channel name has been passed through a purely “static” network, i.e. without using dynamic channel connections. By introducing new reverse static channel connections along the path of the channel name leads to an indirect connection between the sender and the receiver of the dynamic channel which can be used instead for message transfer. Thus the dynamic channel can be eliminated from the program. Using this method, all dynamic channels can be systematically eliminated from a program, when they are used correctly. Consequently, dynamic channels do not extend the expressiveness of the language, but they are an important optimisation technique. However, the problem with extra-functional features, such as this, is that they allow the programmer to write subtly broken programs that would not have been possible before.

2.5.2 Merge

Many-to-one communication is an essential feature for many parallel applications, but, unfortunately, it introduces non-determinism and, in consequence, spoils the purity of functional languages. In Eden, the predefined process abstraction

```
merge :: Trans a => Process [[a]] [a]
```

is used to instantiate a process which does a fair merging of several input streams into a single (non-deterministic) output stream. The incoming values are passed to the output stream in the order in which they arrive. A `merge` process can profitably be used to react quickly to requests coming in an unpredictable order from a set of processes. This is the only way to enable dynamic load balancing of parallel programs in a master-worker scheme:

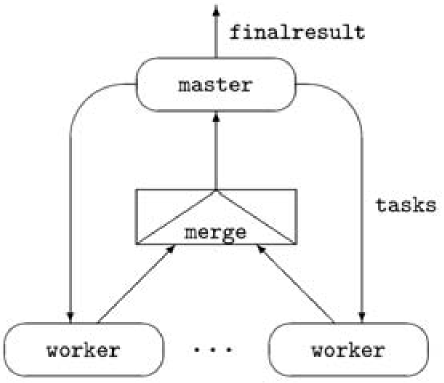


Fig. 10. Master-worker process topology.

```
(finalresult,taskss) = master (merge # [worker # tasks | tasks <- taskss])
```

The resulting process topology[3] shown in Figure 10 is utilised in the replicated workers skeleton described in section 3. Note that the above definition is recursive, because the master and the workers mutually depend on each other.

Although `merge` is of great worth, because it is the key to specify many reactive systems, one has to be aware that functional purity and its benefits are lost when `merge` is being used in a program. Trying to encapsulate the non-deterministic `merge` into a monad in order to isolate the non-functional behaviour would severely restrict its usability, since mutually recursive value-passing as seen above can only be expressed with special monadic fixpoint operators. Functional purity can still be preserved in most portions of an Eden program. In particular, it is possible to use sorting in order to force a particular order of the results returned by a `merge` process.

The Eden constructs are summarised in Figure 11. Sections 3 and 5 give evidence that a powerful and flexible extension for parallelism has been achieved with only five new functions.

3 Skeleton-based programming methodology

Skeletons (Cole, 1989) provide commonly used patterns of parallel evaluation and simplify the development of parallel programs, because they can simply be used in a given application context. A good example is the well-known `map` function, which applies its argument function to each element of a given list. As each of these calculations is independent, the evaluation of each element of the result list can be done in parallel (see Examples 3 and 4). Normally, a skeleton can be implemented in several ways. Implementations may differ in the process topology created, in the granularity of tasks, in the load balancing strategy or in the target architecture used to run the program. So, the implementation hides many details from the potential user, and also determines the efficiency of the program. One of the main

[3] The communication between `merge` and the processes feeding it is always direct.

Predefined types

```
class NFData a => Trans a where (...)
data (Trans a, Trans b) => Process a b = ...
```

Process abstraction and instantiation functions:

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

Dynamic reply channels:

```
type ChanName a = ...
new     :: Trans a => (ChanName a -> a -> b) -> b
parfill :: Trans a =>  ChanName a -> a -> b  -> b
```

Many-to-one communication:

```
merge   :: Trans a => Process [[a]] [a]
```

Fig. 11. Eden constructs.

characteristics of skeletons is that it is possible to predict the efficiency of their applications. This can be done by providing a *cost model* for a particular skeleton implementation. A cost model is just a formula stating the predicted parallel time of the algorithm, which will be parameterised by some constants that may depend either on the problem to be solved, on the underlying parallel architecture, or on the runtime system (RTS).

For a functional programmer, a skeleton is nothing but a polymorphic higher-order function which can be applied with different types and parameters. Thus, programming with skeletons follows the same principle as programming with higher-order functions (in fact the principle used in any abstraction): *To define each concept once and to reuse it many times.*

Eden is one of the few functional languages in which skeletons can be both *used* and *implemented*. In other approaches like Darlington *et al.* (1993) or Michaelson *et al.* (2001), the creation of new skeletons is considered as a system programming task or even as a compiler construction task. Skeletons are implemented by using imperative languages and parallel libraries. Therefore, these systems offer a closed collection of skeletons which the application programmer can use, but without the possibility of creating new ones, so that adding a new skeleton usually implies a considerable effort. Describing both the functional specification and the parallel implementation of a skeleton in the same language context has several advantages. First, it constitutes a good basis for formal reasoning and correctness proofs. Second, it provides much flexibility, as skeleton implementations can easily be adapted to special cases, and if necessary, new skeletons can even be introduced by the programmer himself. In this section we present a selection of typical Eden skeletons. For details on Eden skeletons (their specification, implementation and cost models), the reader is referred elsewhere to Loogen *et al.* (2002).

```
map_farm :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
map_farm  = farm noPe unshuffle shuffle

farm :: (Trans a, Trans b) =>
        Int -> (Int->[a]->[[a]]) -> ([[b]]->[b]) -> (a->b) -> [a] -> [b]
farm np unshuffle shuffle f tasks =
        shuffle (map_par (map f) (unshuffle np tasks))
```

Fig. 12. Farm skeleton.

3.1 Task farms

In most parallel implementations of the well-known `map` function, the input list is considered as a task queue that can be processed using several processor elements (PEs). In Examples 3 and 4 we have already developed a straightforward parallelization of `map`, `map_par`, which creates a new process for each task. This simple approach is not always well-suited, especially in the presence of many fine-grained or irregular tasks. Alternative parallel implementations of `map` use a fixed number of worker processes, each processing a subset of tasks.

The main process of the *farm implementation* creates as many processes as processors are available, distributes the tasks evenly amongst the processes, and collects the results. Each process applies the parameter function to each task it receives, and sends the results back to the main process. The number `np` of workers, and the distribution and collection functions `unshuffle` and `shuffle` are parameters of `farm` (see Figure 12). The `map_par` skeleton is used to create as many processes as the number of task lists into which the original list is decomposed.

`noPe` is an Eden constant giving the number of available processors. Different strategies to split the work into the different processes can be used provided that `(shuffle . unshuffle n) xs == xs` holds for every list `xs`. The farm implementation is appropriate when task granularity is uniform, and when an even distribution of tasks amongst all the processors can be achieved.

In Eden's skeleton library there is a variant of `map_farm` in which the list of tasks is passed to each worker as a free variable instead of through a channel. This may imply the duplication of work (see section 2.4) but nevertheless this approach often reduces the total execution time, as the amount of communication is much lower. When the evaluation of the task list is cheaper than communicating the evaluated list (or parts of it), it is better to allow the workers to evaluate the list of tasks on their own and to select their part of it. The resulting skeleton is called *self-service farm implementation* of `map`.

3.2 Replicated workers

Load balancing is a crucial issue when developing parallel programs. A bad load balance will cause poor speedups for an otherwise elegant parallel algorithm. The farm implementation is appropriate only if the different tasks in the list can be guaranteed to have a regular granularity. Besides irregular task granularity, a non-homogeneous processor architecture or additional load on some processors may require to distribute work on demand. In this case, a new task will be assigned to a process only when it has finished its previous task. Thus, some processors may solve a few complex tasks while others solve many small ones. The amount of work done by each processor will be approximately equal.

This idea gives rise to the *replicated workers* skeleton (Klusik *et al.*, 2002). Initially, the manager assigns one or more tasks to each of the workers. By assigning several tasks, idle time between tasks is minimised. Each time a worker finishes a task, it sends an acknowledgement message to the manager including the result, and then a new task (if available) is assigned to that process. The computation finishes when the manager has received all the results. The programmer cannot predict the order in which processes are going to finish their works, as this depends on runtime issues. By using the reactive process `merge`, results from different processes can be received by the manager in the order in which they arrive. Thus, if each result contains the identity of the sender process, the list of merged results can be scrutinised to know who has sent the first message, and a new task can be assigned to it. Notice that this approach could not be used in a purely functional language, as process `merge` is not functional (see section 2.5.2).

The input parameters of the skeleton shown in Figure 13 are: (1) the number of worker processes to be used; (2) the size of workers' pre-fetching buffer; (3) the worker function; and (4) the list of tasks.

Notice that the output of the list of workers, `outss`, is used in two different ways:

1. An instance of `merge` is applied to it in order to obtain a list `unordResult` containing the order in which the results are generated. This list is used by `distribute` to assign new tasks to processors which have delivered a result.
2. The final result is obtained by applying `sortMergeByTid` to it, which is a simple Haskell function (not shown) merging the workers lists (each of them already sorted) into a single list sorted by task identity.

Sorting the results guarantees that, seen from the outside, the skeleton is completely deterministic. In order to implement `map`, a worker is created for every processor.

3.3 Divide-and-conquer

Divide-and-conquer is a well-known scheme in sequential programming: The problem is *split* into one or more smaller subproblems. Once they are recursively solved, their results are *combined* to produce the solution of the original problem. The splitting process stops when a subproblem *is trivial* enough to be *solved* without recursively invoking the function. The Haskell version of this scheme is the following polymorphic higher-order function:

```
dc :: (a -> Bool) -> (a -> b) -> (a -> [a]) -> (a -> [b] -> b) -> a -> b
dc is_trivial solve split combine x
   | is_trivial x = solve x
   | otherwise    = combine x children
   where children = map (dc is_trivial solve split combine) (split x)
```

```
map_rw :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
map_rw =  rw noPe 2

rw :: (Trans a, Trans b) => Int -> Int -> (a -> b) -> [a] -> [b]
rw np prefetch f tasks =  results
 where
   results     = sortMergeByTid outss
   outss       = [(worker f i) # input | (i,input) <- zip [0..np-1] inputs]
   inputs      = distribute tasksAndIds (initReqs ++ map owner unordResult)
   tasksAndIds = zip [1..] tasks
   initReqs    = concat (replicate prefetch [0..np-1])
   unordResult = merge # outss

   distribute []      _      = replicate np []
   distribute (e:es) (i:is) = insert i e (distribute es is)
       where insert 0     e ~(x:xs) = (e:x) : xs
             insert (n+1) e ~(x:xs) = x: (insert n e xs)

data (Trans b) => ACK b = ACK Int Int b
worker :: (Trans a, Trans b) => (a->b) -> Int -> Process [(Int,a)] [ACK b]
worker f i = process  (map f')
             where f' (id_t,t) = ACK i id_t (f t)
```

Fig. 13. Replicated workers skeleton.

Notice that the resulting call tree may be non-homogeneous, and that trivial solutions may appear at any level of the tree. The easiest way to parallelise the `dc` scheme in Eden is to replace `map` by `map_par` and to stop the parallel unfolding at a given level `d`. A dynamic tree of processes will then be created with each process connected to its parent. An additional integer parameter `d` determines the maximum level after which no more child processes are generated, and the sequential version is used instead. We call the resulting skeleton `dc_par`.

Using the `rw` implementation of `map` allows however a better control over process granularity and distribution, and a better load balance. The process creation overhead will be decreased as well, since only one process per processor will be created. The original task is split up to a given depth, and a subtask is created for every subtree at this depth. The list of subtasks is given to a `map_rw` skeleton in which the worker function is just the sequential algorithm (see Figure 14). To be able to appropriately combine the results returned by the parallel processes, the tree shape of splitting the task must be saved as well. Function `generateTasks` does the splitting job, while function `combineTop` combines the results level-wise from the leaves of the tree to the top. (These are simple Haskell definitions which are not shown here.) Notice that the initial splitting and the final combination are done in the manager processor, while solving the leaves is done in the worker processors. Due to the laziness of the language, part of the splitting and of the combination can

```
dc_rw :: (Trans a,Trans b) =>
         Int -> (a -> Bool) -> (a -> b) -> (a -> [a]) -> (a -> [b] -> b)  ->
         a -> b
dc_rw d is_trivial solve split combine x = combineTop combine levels results
  where
    (tasks,levels) = generateTasks d is_trivial split x
    results        = map_rw (dc is_trivial solve split combine) tasks

data Tree a = Node a [Tree a]

generateTasks :: Int -> (a -> Bool) -> (a -> [a]) -> a -> ([a], Tree a)

combineTop    :: (a -> [b] -> b) -> (Tree a) -> [b] -> b
```

Fig. 14. Divide-and-conquer skeleton.

be done in parallel with solving the leaves. In any case, this load should be small enough to avoid a bottleneck in the manager processor.

3.4 Replicated workers with dynamic task creation

Many problems are well-suited for a replicated workers implementation. For instance, searching problems in which a huge-space state-tree has to be searched in parallel to find one (or: the optimal) solution. In these problems the initial set of tasks is small (it may even be only one initial task) but the number of tasks increases as long as subproblems are solved. We have investigated several variants of the basic replicated workers scheme in which workers are allowed to dynamically generate new tasks. These variants are appropriate to solve depth-first and branch-and-bound search problems. To complicate things, in some of these problems, the workers must maintain an internal state (for instance, the cost of the best solution found up to now) so that the result of the task at hand depends not only on the task itself but also on the worker state. The manager may update the state from time to time when new data are received from other workers.

We sketch a skeleton called *stateful replicated workers* implementing all these features. The details of this skeleton can be found in (Martínez & Peña, 2004). Figure 15 shows only the type of `strw` in Eden. We maintain the parameters of the stateless replicated workers skeleton seen before: the number of workers to be created by the skeleton, and the size of the prefetch buffer. The next four parameters are the problem-dependent functions delivered to the skeleton. The implementation of `strw` follows similar patterns to those of the basic skeleton presented in Figure 13. A new main concern is termination, as tasks are now dynamically created. The manager has to ensure that it triggers termination, only if no more new tasks will be created. This is handled by the function `distribute` which has now eight parameters instead of two in the basic version (see Figure 13). The purposes of `distribute` are manifold:

```
strw  :: (Trans tsk, Trans act, Trans res, Trans wl) =>
      Int ->                                        -- no. of PE
      Int ->                                        -- buffer size
      (inp -> Int -> ([wl],[tsk],ml)) ->            -- split function
      (wl -> tsk -> [act] -> (res,wl)) ->           -- worker function
      (ml -> res -> Int -> ([[act]],[tsk],ml)) ->   -- combine function
      (ml -> result) ->                             -- result function
      inp ->                                        -- skeleton input
      result                                        -- skeleton result
```

Fig. 15. The type of the Eden skeleton `strw` (stateful replicated workers).

1. It detects when a worker has finished a task and assigns a new one, as in the basic `rw` skeleton.
2. It computes the list of pending updates for each individual worker and combines it with the newly assigned task.
3. It detects termination. To this aim, it controls the number of tasks generated by the skeleton, the number of tasks distributed to workers, and the number of results received from workers. Termination can be triggered, as soon as these numbers are equal.

3.5 Ring

Many parallel algorithms arrange processes in a unidirectional ring, where each process – apart from sending and receiving values to and from the parent – is connected to only two neighbours: the previous link, from which it receives values, and the next link, to which it sends values. By using dynamic channels to provide direct connections between processes, the `ring` skeleton defined in Figure 16 creates the desired topology. Each ring process `pring` receives an input from the parent, and a channel name used to send values to its successor in the ring. It produces an output sent to the parent, and a channel name used to receive inputs from its predecessor in the ring.

The parameters of the skeleton are the number `n` of ring processes, a function `distribute` to distribute the input data to the ring processes, a function `combine` to combine the outputs produced by the ring processes into a final result, a function `f` to be performed in each ring process and the input data. As the type of function `f` shows, each ring process receives data of type `a` from the parent, and data of type `[c]` from its predecessor process. It produces output of type `[c]` for its successor and a result of type `b` for the parent. Function `mzip` is a more lazy variant of the standard function `zip`.

4 Implementation

Eden's compiler[4] has been developed by extending the Glasgow Haskell Compiler (GHC, 1993). The GHC has been chosen as the basis of our compiler because of its

[4] Available at http://www.mathematik.uni-marburg.de/~eden

```
ring :: (Trans a, Trans b, Trans c) =>
        Int -> (Int -> a -> [a]) -> ([b] -> b) -> ((a,[c]) -> (b,[c]))
        -> a -> b
ring n distribute combine f input = combine toParent
  where
    (toParent, chans) = unzip (map_par (pring f) inputs)
    inputs            = mzip toChildren preds
    toChildren        = distribute n input
    preds             = last chans : init chans -- rotate chans

-- each individual process in the ring
pring ::(Trans a, Trans b, Trans c) =>
         ((a,[c]) -> (b,[c])) -> (a,ChanName [c]) -> (b,ChanName [c])
pring f (fromParent,nextChan) = new (\ (prevChan, previous) ->
                let (toParent, next) = f (fromParent, previous)
                in  parfill nextChan next (toParent,prevChan))
```

Fig. 16. Ring skeleton.

efficiency and portability. Moreover, Haskell features and extensions supported by the GHC may be used in Eden.

4.1 Extending the Glasgow Haskell Compiler

The GHC translates Haskell programs into abstract C-Code which is flattened into proper C using macro definitions of the runtime system. A standard C compiler (Gnu) translates the resulting C code into native code which is finally linked with the runtime system code to give the executable (see Figure 17). The GHC runtime system (RTS) implements an abstract graph reduction machine, called the STG machine (Peyton Jones, 1992). The compilation into STG-code is a chain of transformations, finally resulting in a C representation (abstract C) (Peyton Jones, 1996).

The GHC allows to extend the functionality of its runtime system by defining additional primitive operations, i.e. functions directly implemented in C by the compiler. They provide basic atomic actions which are performed directly in the runtime system. The Eden compiler uses the eight primitive operations shown in Figure 18 to provide the additional functionality needed by the Eden constructs. The type class `Trans`, the type constructors `Process` and `ChanName` as well as the new Eden functions like `process` and `#` have been defined in a special module, called the *Eden module*, which has to be imported by every Eden program. The definitions are based on the Eden-specific primitive operations. To give an impression of how the Eden module realizes parallelism control, we show the definition for process abstraction (see Figure 19) which defines how a new process is set up in a remote environment.

A process abstraction of type `Process a b` is implemented by a function `f_remote` which will be evaluated remotely by a corresponding child process. The function's parameter are two channel names: the first `outDCs` (of type `ChanName b`) is a channel for sending its output while the second `chanDC` (of type `ChanName (ChanName a)`)

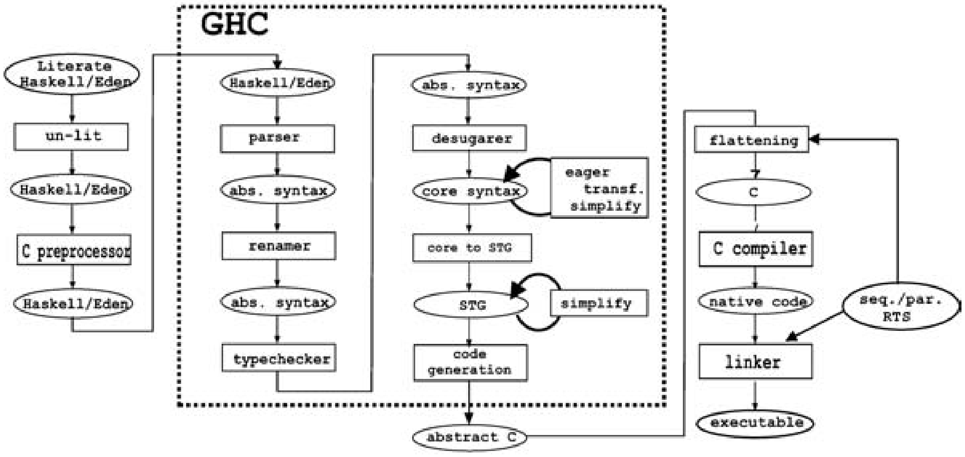


Fig. 17. Overview of the GHC with extensions for Eden.

1.	`createProcess#`	request process instantiation on another processor
2.	`createDC#`	create (dynamic) communication channel
3.	`setChan#`	connect communication channel in the proper way
4.	`sendHead#`	send head element of a list on a communication channel
5.	`sendVal#`	send single value on a communication channel
6.	`noPe#`	determine number of processing elements in current setup
7.	`selfPE#`	determine own processor identifier
8.	`merge#`	nondeterministic merge of a list of outputs into a single input

Fig. 18. Primitive operations for Eden.

```
data (Trans a, Trans b) =>
     Process a b = Proc (ChanName b -> ChanName (ChanName a) -> ())

process :: (Trans a, Trans b) => (a -> b) -> Process a b
process f = Proc f_remote
  where f_remote outDCs chanDC
          =  let (inDCs, invals) =  createDC invals
             in  writeDCs chanDC inDCs `fork` (writeDCs outDCs (f invals))
```

Fig. 19. Haskell definition of Eden process abstraction.

is an administrative channel to return the names of input channels to the parent process. The exact number of channels which are established between parent and child process does not matter in this context, because the operations on dynamic

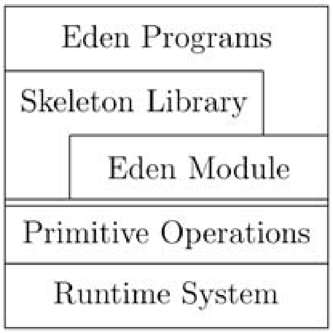


Fig. 20. Layer structure of the Eden system.

channels are overloaded. The definition of `process` shows that the remotely evaluated function, `f_remote`, creates its input channels via the function `createDC` which is a wrapper function for the corresponding primitive operation. Moreover, a function `writeDCs` which is defined using the primitive operations `setChan#`, `sendHead#`, and `sendVal#`, is used twice: the dynamically created input channels of the child, `inDCs`, are sent to the parent process via the channel `chanDC` and the results of the process determined by evaluating the expression `(f invals)` are sent via the channels `outDCs`.

Note that, although the language definition introduces `merge` as a process abstraction, the current implementation provides it as a function `merge :: Trans a => [[a]] -> [a]` implemented by a primitive operation. Thus, a `merge` is tightly coupled with the receiver process, enabling direct channel connections from the producer processes to the receiver and avoiding any additional process creation overhead.

The main advantage of using primitive operations and the Eden module is that Eden programs can be passed through the GHC front-end without any changes. In particular, the original type inference algorithm checks the types of Eden programs. An additional transformation on the core language level, called the *eager transformation*, moves top-level process instantiations into a strict context where they will be evaluated eagerly, as described in section 2.3. The most important changes of the GHC concern the back end of the compiler and mainly the runtime system (RTS). The necessary modifications have been designed as orthogonal additions to the existing implementation. The implementation re-uses simplified kernel parts of the parallel functional RTS GUM, the implementation of GpH (Trinder *et al.*, 1996).

The layered implementation of Eden, shown in Figure 20, achieves more flexibility and improves the maintainability of this highly complex system (Berthold *et al.*, 2003). By lifting aspects of the RTS into the Eden module, basic work-flows can be defined on a high level of abstraction. The layer structure makes the development of extensions much easier once the RTS support is implemented. It takes complexity out of the low-level RTS and simplifies its maintenance.

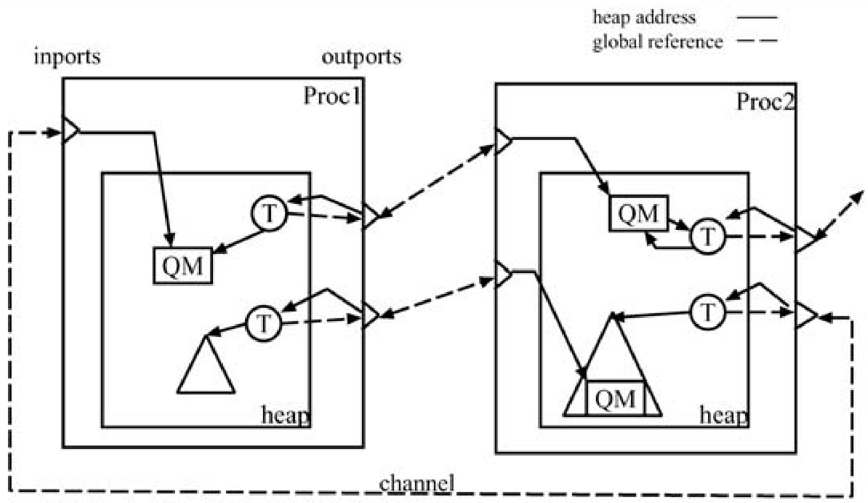


Fig. 21. DREAM instances and their interconnections.

4.2 DREAM – The DistRibuted Eden Abstract Machine

Eden's RTS is an implementation of the DREAM abstract machine (Breitinger *et al.*, 1997a), a parallel version of the sequential STG-machine. Each Eden process is executed by its own instance of an extended STG-machine. It consists of one or more concurrent threads of control. These threads, which evaluate different output expressions, are independent of each other, but use a common heap with shared information. Input is also shared among all threads in a process. As explained before, the channels are represented by their ends which are called *inport* on the receiver side and *outport* on the sender side. The inport points at heap locations where the incoming data should be stored. Until the data arrives, a *Queue-Me* closure (QM) blocks any demanding thread. Every thread is associated with its own outport via which it will send the result of its computation. Each inport knows from which sending thread (referred to by its outport) it will receive the values (see Figure 21). The necessary information is kept in an *inport table* mapping inport ids to the heap addresses of Queue-Me closures, and an *outport table* mapping outport ids to the destination inport.

In contrast to the operational semantics shown in Subsection 2.4, the formalisation of the DREAM concept makes these port connections explicit in the state of a process. The generated process system is a collection of inter-related DREAM instances. The state of a process includes information common to all threads and the states of the threads. The shared part includes the heap and the inport table. The state of a *thread* comprises the state of an STG-machine and the associated outport referencing the connected inport.

We do not go further into the details of the DREAM model, the interested reader is referred to the original paper (Breitinger *et al.*, 1997a).

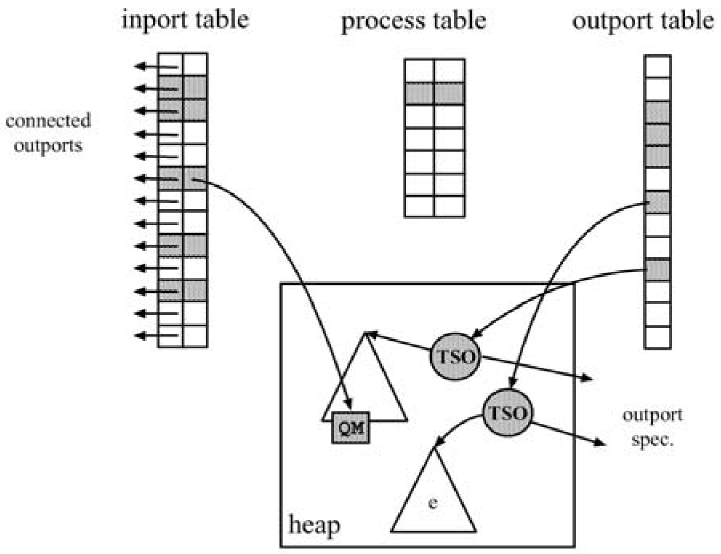


Fig. 22. Runtime tables per PE.

4.3 The parallel runtime system

To map arbitrary process systems on a finite machine, multiple processes have to be evaluated in an interleaved manner on the same processor element (PE). For the purpose of keeping process creation as cheap as possible, the RTS provides only one instance of DREAM per PE, which can execute several Eden processes concurrently. Processes executed on the same PE share the scheduler and the runtime tables. Thus, there is only one instance of the inport and outport tables per PE (see Figure 22). The *inport table* maps locally unique inport identifiers to the heap addresses of the corresponding Queue-Me closures and to the global references to the connected outports. The outport references are used for the propagation of termination information. The *outport table* maps locally unique outport identifiers to the corresponding address of the thread state object (TSO) which represents a thread in the heap. The outport table is used for system management, i.e. garbage collection, termination, error detection, etc. A process table provides for each process the number of inports and the number of threads, which is equal to the number of outports.

The 1:1 channels allow to notify and in consequence to terminate the sending thread when an inport is closed. Thus, local garbage collection (GC) may have global effects. When the last thread of a process terminates, the whole process is terminated and its remaining inports are closed. Otherwise the inports would be kept alive until the next GC.

Eden uses a fair round-robin scheduling of threads to guarantee that threads waiting for input are not blocked too long. To reduce the communication costs, several messages addressed to the same PE can be put together into a single packet.

This leads to a dynamic adaptation of the message granularity in the runtime system (Berthold, 2004).

For the moment, Eden's RTS supports two modes to map processes to processors, which can be chosen by the user for each execution. *Round-robin mode*: If several processes are instantiated from a PE p, they are mapped to consecutive processors starting with $p + 1$. *Random mode*: Each processor maps instantiated processes to randomly chosen processors. Notice that the round-robin mode allows the programmer to control the mapping of processes to some extent, as it may be achieved that different processes will be placed on different processors. This is, for example, the case for the farm and replicated workers skeletons introduced in section 3. Eden's primitive for process instantiation allows explicit process placement and we are currently experimenting with this feature to extend skeletons with explicit placement instructions.

To sum up, the overall setting is to have one instance of the executable on each PE, one of which is called the *mainPE* as it starts the execution by evaluating the expression `main`. For the inter-processor communication, either PVM (1993) or MPI (1997) can be used. As only very basic message passing operations are used, they could readily be substituted by any other message passing library. The current compiler uses PVM. The interested reader is referred to Breitinger *et al.* (1998), Klusik *et al.* (1999) and Berthold *et al.* (2003) for details on the implementation of the parallel Eden RTS.

5 Case studies and performance results

This section is devoted to three case studies which make use of the skeletons previously introduced. The main objective here is to show how easy it is to express parallel algorithms in Eden, once you are provided with a library of skeleton implementations. A second objective is to show performance results in the form of relative speedup curves for some concrete examples. This means that the reference sequential program is the parallel Eden program running on a single processor. So, the graphics will show how efficiently the algorithm scales with the number of processors.

The case studies have been chosen as representatives of the kind of parallel problems that can be solved in Eden. The first one, called *pair interaction*, is a typical systolic example in which a ring of processes performs a regular and highly synchronised computation. The second one is Karatsuba's algorithm for multiplying very large integers. It is a parallel *divide and conquer* problem with irregular parallelism, both in the created topology and in the task granularity. This is due to the different length of the initial numbers. The main difficulty here is to map the tree-shaped process topology to a fixed number of processors. The third case study parallelises Buchberger's algorithm from the field of computer algebra. Its granularity is irregular. Additionally, the number of parallel tasks is not known in advance, as they are dynamically created. Therefore the stateful replicated workers skeleton is applied.

The experiments for the first two case studies have been performed with a Beowulf cluster at the University of St. Andrews. Nodes are 450 MHz Pentium II running Linux RedHat 5.2, with 348 MB of DRAM and connected through a CISCO 2984 G full duplex 100 Mb/s fast Ethernet switch, the latency being $\delta = 142\,\mu$s. So, it is a low cost environment with high latencies. The third case study has been measured on a local Beowulf cluster with five processors.

5.1 Pair interactions

Let us assume that we want to determine the force undergone by each particle in a set of n atoms. The total force vector f_i acting on each atom x_i, is

$$f_i = \sum_{j=1, j\neq i}^{n} F(x_i, x_j)$$

where $F(x_i, x_j)$ denotes the attraction or repulsion between atoms x_i and x_j. This constitutes an example of pairwise interactions. For a parallel algorithm, we may consider n independent tasks, each devoted to compute the total force acting on a single atom. Thus, task i handles atom x_i and computes $\{F(x_i, x_j) \mid j \in \{1, \ldots, n\}, j \neq i\}$.

A separate process for each task would generate a big overhead, when dealing with a large set of particles. Therefore, we partition the set of atoms into as many subsets as the number of available processors. Each processor must compute the interaction between each of its local particles and all the rest. This is a quadratic computation. In order to minimise the amount of communications, we arrange the processors into a ring and make the atoms information flowing around the ring. In this way, after $n - 1$ rounds, all the processors will complete the computation of the interactions. At the first iteration, each processor will compute the forces between the local particles assigned to it. Then, at each subsequent iteration, it will receive a new set of particles, and compute the forces between its own particles and the new ones, adding the forces to the ones already computed in the previous iterations:

```
force    :: [Atom] -> [ForceVec]
force xs =  ring noPe splitIntoN concat (force' noPe) xs

force'   :: Int -> ([Atom],[[Atom]]) -> ([ForceVec],[[Atom]])
force' np (local,ins) =  (total,outs)
   where outs         = take (np - 1) (local : ins)
         total        = foldl1' f forcess
         f acums news = zipWith addForces acums news
         forcess      = [map (faux ats) local | ats <- (local:ins)]
         faux xs y    = sumForces (map (forcebetween y) xs)
         sumForces l  = foldl' addForces nullvector l
```

Function `splitIntoN` distributes the n particles to the `noPe` processors; function `forcebetween` computes the forces betweeen two single particles; the list `forcess` has type `[[ForceVec]]`, and each of its blocks represents the forces undergone by the local particles caused by the particles of a foreign block; function `addForces :: ForceVec -> ForceVec -> ForceVec` simply adds two forces.

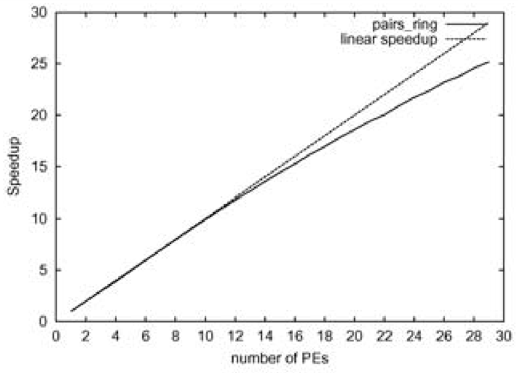


Fig. 23. Speedups of pair interactions.

Figure 23 shows the speedups determined with up to 30 Beowulf nodes for an input size of 7000 particles. We have obtained a relative speedup of 25 with 30 processors. This corresponds to an efficiency of 83,3% with 30 PE. The execution time of the parallel program in one processor was 194.86 seconds. Notice that the total communications of each process are in O(n), while its computations are in O($n^2/$`noPe`).

5.2 Karatsuba's algorithm

Karatsuba's algorithm (e.g. see Horowitz & Sahni 1978) computes the product of two large integers using a divide-and-conquer approach. Let us assume that a long integer is represented in base b as a list of "digits" d_i, where for all i we have $0 \leqslant d_i < b$. If two large integers x and y are to be multiplied, the algorithm works as follows:

- Let n be half of the length of the longest of the two integers.
- Let $x_1 = x/b^n$, $x_2 = x \,\mathbf{mod}\, b^n$, $y_1 = y/b^n$ and $y_2 = y \,\mathbf{mod}\, b^n$.
- Let $u = x_1 y_1$, $v = x_2 y_2$, $w = (x1 + x2)(y1 + y2)$.
- The result of the multiplication is $ub^{2n} + (w - u - v)b^n + v$.

Notice that it is not necessary to perform any division to obtain x_1, x_2, y_1 and y_2. It is enough to cut the lists representing x and y into two halves. The multiplication with b^n and b^{2n} only needs appending zeros to the corresponding list. Therefore, only three multiplications are needed, i.e. three subproblems of length $n/2$ are generated when splitting a problem. Combining the subresults has a cost in $O(n)$. This leads to an overall complexity in $O(n^{log_2 3})$.

This algorithm perfectly fits into a divide-and-conquer scheme. Notice that the granularity of the subtasks may not be uniform as the three multiplications are possibly applied to integers of different lengths. Also, trivial nodes may appear at any depth in the tree. The implementation of the Karatsuba algorithm in terms of

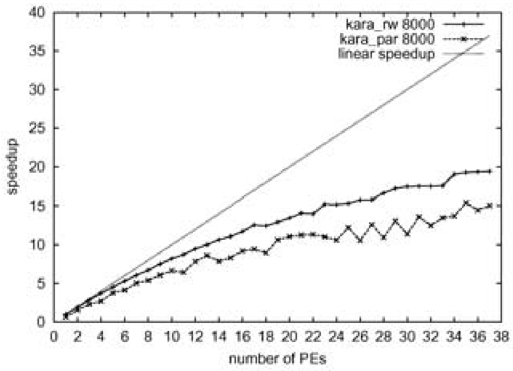


Fig. 24. Speedups of the Karatsuba algorithm.

the divide-and-conquer skeleton dc_rw is as follows:

```
type LongInteger = [Int]
karat       :: LongInteger -> LongInteger -> LongInteger
karat i1 i2 =  dc_rw depth trivial solve split combine (i1,i2)
    where depth = ceiling (logBase 3 (10*noPe))
          trivial (i1,i2) = null i1 || null i2
          solve _         = []
          split   :: ([Int],[Int]) -> [([Int],[Int])]
          combine :: ([Int],[Int]) -> [[Int]] -> [Int]
```

where the code of split and combine follows the patterns described above.

Both the dc_rw and the dc_par version of the divide-and-conquer skeleton have been tested for the same input data. The execution time of the parallel program in one processor was 440 seconds. The speedups obtained for both skeletons can be seen in Figure 24. As expected, the naïve implementation of the skeleton is worse and also more irregular than the other, the main reason being that the load balance is poorer and more random. Moreover, the overhead for creating processes is greater. For the dc_rw skeleton, we have obtained a relative speedup of 20 with 38 PE. This corresponds to an efficiency of 52,6% with 38 PE. A reason for this sub-optimal speedup can be found in Amdahl's law (Amdahl, 1967), as there is an inherent and not negligible sequential part in this algorithm: the initial computation of the subtasks and the final combination of the results.

5.3 Gröbner bases

Gröbner bases computation is a computer algebra algorithm with plenty of applications in commutative algebra, geometry and systems theory. The problem can be explained in the following terms: Given a finite set of polynomials $F = \{f_1, \ldots, f_r\}$ in n indeterminates $x_1, \ldots, x_n$, a Gröbner basis is another finite set of polynomials $G = \{g_1, \ldots, g_t\}$ determining the same ideal and satisfying an additional canonical property. The ideal I determined by a set S of polynomials, denoted $I = \langle S \rangle$ is the

```
function Buchberger ($F = \{f_1, \dots, f_r\}$) return $G$
  $G := F$; $P := \{(f_i, f_j) \mid f_i, f_j \in F, i \neq j\}$;
  while $P \neq \emptyset$ do
    $(f, g) \leftarrow chooseAPair\ (P)$; $P := P - \{(f, g)\}$
    $S(f, g) \overset{G}{\longrightarrow} * h$ such that $h$ is reduced w.r.t. $G$
    if $h \neq 0$ then
      $P := P \cup \{(u, h) \mid u \in G\}$;
      $G := G \cup \{h\}$
    end if
  end while
  return $G$
end function
```

Fig. 25. Buchberger's sequential algorithm computing a Gröbner basis.

smallest set containing S and closed under polynomial addition and product:

$$\langle S \rangle \stackrel{\text{def}}{=} \{\sum_{f_i \in S} u_i f_i \mid u_i \in P[x_1, \dots, x_n]\}$$

with $P[x_1, \dots, x_n]$ being the set of all polynomials in n indeterminates.

Given a finite set F of polynomials, there exists an algorithm by B. Buchberger (Adams & Loustaunau, 1994) which computes a Gröbner basis G for the ideal I determined by F. It makes intensive use of two elementary steps: computation of the *S-polynomial* of two polynomials f and g, denoted $S(f, g)$, and *reduction* of a polynomial f to normal form with respect to a set G of polynomials, denoted $f \overset{G}{\longrightarrow} * h$. These operations involve linear combinations of polynomials. The sequential Buchberger algorithm is shown in Figure 25.

It has been proven that the algorithm always terminates and that its cost is in $O(msp)$, where m and s are the worst case values for the length of the reduction chains of S-polynomials and the cardinality of G, respectively. Value p is the number of pairs in the final G. It is *a priori* unknown and depends on the form of the initial polynomials in F. In the worst case, p can be exponential in the cardinality of F.

The parallel algorithm surveyed here has been run in a hybrid Eden-Maple system by using an interface developed as part of the Eden project. The idea for the parallel version of Buchberger's algorithm is to compute the reduction to normal form $S(f, g) \overset{G}{\longrightarrow} * h$ in parallel for different pairs (f, g). The order in which such pairs are chosen is not important for the correctness and the termination of the algorithm. The granularity of such decomposition is large enough to justify the communication of the polynomials f and g. This problem perfectly fits the stateful replicated worker skeleton `strw` described in section 3.4. So, the strategy chosen has been to have a *manager* process communicating pairs (f, g) to a fixed set of *worker* processes, and getting back the results h of such reductions. If the result is 0, the manager will just move to the next pair. Otherwise, the manager will compute additional pairs and add them to the list of pending pairs.

$$\begin{array}{lll}
\texttt{split}\ F\ \texttt{np} & \stackrel{\text{def}}{=} & ([\overbrace{F,\ldots,F}^{\texttt{np}}], [(f_i, f_j) \mid f_i, f_j \in F, i \neq j], F) \\
\texttt{wf}\ G\ (f,g)\ [h_1,\ldots,h_r] & \stackrel{\text{def}}{=} & (\texttt{res}, G') \quad \text{where} \\
 & & G' = G \cup \{h_1,\ldots,h_r\} \\
 & & \texttt{res} = \begin{cases} [\,] & \text{, if } S(f,g) \xrightarrow{G'} * \, 0 \\ [h] & \text{, if } S(f,g) \xrightarrow{G'} * \, h \neq 0 \end{cases} \\
\texttt{combine}\ G\ \texttt{res}\ \texttt{np} & \stackrel{\text{def}}{=} & ([\overbrace{\texttt{res},\ldots,\texttt{res}}^{\texttt{np}}],\ \texttt{tsks},\ G') \quad \text{where} \\
 & & (\texttt{tsks}, G') = \begin{cases} ([\,], G) & \text{,if } \texttt{res} = [\,] \\ ([(u,h) \mid u \in G], G \cup \{h\}) & \text{,if } \texttt{res} = [h] \end{cases} \\
\texttt{result}\ G & \stackrel{\text{def}}{=} & G
\end{array}$$

Fig. 26. Problem-dependent functions of `strw` for Gröbner problem.

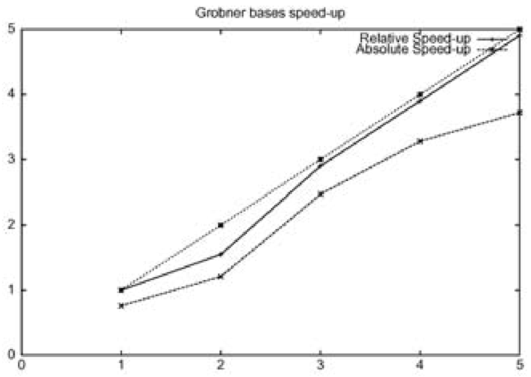


Fig. 27. Speedups of the parallel algorithm computing a Gröbner basis.

Each Eden worker process has an auxiliary Maple process to which it delegates the computation of $S(f,g) \xrightarrow{G} * \, h$. Maple systems usually provide a sub-library to compute Gröbner bases sequentially. However, they also provide the elementary steps of the algorithm as individual functions. In particular, there exists a function called `spoly` computing the S-polynomial of two given polynomials, and a function called `normalf` computing the normal form of a polynomial with respect to a set of polynomials. By looking at the sequential algorithm of Figure 25, it is easy to define the four problem-dependent functions. They are shown in Figure 26.

We have run the skeleton, the problem-dependent functions and the Eden-Maple interface in a small Beowulf cluster with five processors. The absolute and relative speedups are shown in Figure 27. The reference sequential version for the absolute speedup is Buchberger's algorithm written completely in Maple. Its sequential time was 212 sec. Notice that the Eden version running in one processor is about 32% slower (280 sec.) than the pure Maple version. We find this overhead acceptable as the combination of the two languages, one of them functional, the interface itself, which communicates Eden and Maple via a Unix pipe, and the Eden RTS, which creates threads and processes, constitute enough reasons for it. The relative speedup is however rather good: 4,91 with 5 PE, or 98,2% efficiency with 5 PE. This means that the skeleton `strw` provides a very good load balance and that there are no sequential bottlenecks in the algorithm.

The three case studies have shown that programs can easily be parallelised using pre-defined skeletons and that good speedups can be achieved offhand.

6 Related work

This section is divided into three parts. First, we describe Eden-specific related work, i.e. we shortly sketch further achievements of the Eden project which are out of the scope of this paper. Then, we give an overview of other parallel or concurrent functional languages. Finally, we discuss Eden in the context of these other approaches, thereby pointing at similarities and explaining essential differences.

6.1 Further achievements

In this paper, we have given a survey on Eden's design, the semantics, the programming methodology, the implementation, and some runtime experiments. Further work has been done on program analysis, profiling, automatic skeleton selection by meta-programming techniques, and an interface between Eden and the Maple system which has been used in the case study on Gröbner bases. In this subsection, we want to point the interested reader at the corresponding publications.

Analyzing Eden programs. The GHC follows the principle "compilation by program transformation" (Peyton Jones, 1996). Some optimizing transformations may however affect the semantics of Eden programs, in particular in the presence of non-determinism. Fortunately, the critical GHC transformations tend to reduce the degree of non-determinism in Eden programs, i.e. the number of different behaviours (Pareja *et al.*, 2001). *Non-determinism analyses* have been investigated by Peña and Segura (2004).

Another Eden-specific analysis tries to find out whether incoming data will immediately be transferred to other processes without any local evaluation depending on them. This is e.g. the case for the cascading pipe described in section 2.5 (see Figure 8 (b)). Automatically *bypassing* such roundabout ways improves the overall communication topology (Klusik *et al.*, 2000).

Automatic skeletons. Haskell has recently been extended with compile-time meta-programming facilities, called Template Haskell (Sheard & Peyton-Jones, 2002). A system for automatically deriving Eden implementation skeletons from high-level skeleton specifications has been described in Hammond *et al.* (2003). The approach uses Template Haskell to automatically transform high-level skeletons into good parallel implementations on the basis of static cost information.

Profiling. It is not easy to write efficient parallel programs, or to reason about their runtime behaviour. Some kind of feedback is needed in order to check whether programs are well-parallelised or to understand the reasons of possible inefficiencies. PARADISE (PARAllel DIstribution Simulator for Eden) is a simulator tool for profiling Eden programs (Hernández *et al.*, 2000). It substantially modifies GranSim (Hammond *et al.*, 1995), a corresponding simulator for Glasgow parallel Haskell (GpH). Unfortunately, the paradise simulator is only available for an older version of the Eden compiler (version 3.02) which is no longer supported. Instead, the Eden runtime system has been instrumented by trace outputs which are protocolled per processor and combined into a single trace file after program termination. This trace file can then be analysed with a separate trace-viewer tool (Roldán Gómez, 2004) which displays interactive diagrams of machine, process and thread activities, as well as their communication. Trace viewing allows to observe and analyse the *real* parallel runtime behaviour of Eden programs, while PARADISE only *simulated* parallel execution.

Eden-Maple interface. The Eden-Maple interface briefly mentioned in section 5.3 allows the parallelization of computer algebra algorithms, which are known to be computation-intensive. The idea is to use Eden as a coordination layer running on top of multiple Maple systems, all of them running in parallel in the available processors. The computation intensive functions are kept in the Maple processes and these are called from the Eden layer. The interface and its use are described in detail in Martínez & Peña (2004)[5].

6.2 The spectrum of parallel functional languages

As functional languages offer good opportunities for parallelism due to the freedom in the evaluation order of their subexpressions, there have been many different approaches to parallel functional programming. Comprehensive overviews have been given in Hammond & Michaelson (19999) and by Trinder *et al.* (2002).

The exploitation of implicit parallelism is challenging, but time has shown that implicit parallelism is simply *too much* to be exploited effectively. There is a high risk to produce a large number of fine-grained parallel activities in such a way that the benefits of parallelism are lost in creating and communicating processes. For this reason, many of the more recent approaches rely on the programmer to decide which expressions deserve the effort of creating a parallel process for their evaluation. The *degree of explicitness* chosen in the various proposals is however different. We distinguish between the following two language groups:

Transformational languages: In a parallel transformational system, inputs are transformed to outputs functionally depending on them. The main purpose of parallelism is to speed up the computation. The programmer adds special expressions to a purely functional program, either written as annotations interspersed in the text

[5] The sources of the interface are available at `http://dalila.sip.ucm.es/~ricardo`.

or provided as specialised *wiring* functions, which abstractly specify where and when processes should be created. The denotational semantics of a program with these specialised expressions is (almost) the same as the semantics without them. An important semantic difference might be the order of evaluating subexpressions. We refine this group further into three subgroups:

Annotated languages: These are languages like *Concurrent Clean* (Nöcker *et al.*, 1991), *Glasgow parallel Haskell* (GpH) (Trinder *et al.*, 1998), or *Caliban* (Kelly, 1989).

Skeleton languages: Here we include most of the work done on skeletons. Typical examples are: Bratvold (1993), Darlington *et al.* (1993), Bacci *et al.* (1999), Herrmann (2000), Hamdan (2000) and Michaelson *et al.* (2001).

Data-parallel languages: Examples are *NESL* (Blelloch, 1996), *pH* (Nikhil *et al.*, 1995), *Sisal* (Gaudiot *et al.*, 2001) and *SaC* (Scholz, 1996).

Reactive languages Reactive systems are different from transformational ones: usually they do not have clear notions of inputs and outputs or even of termination. The purpose of parallelism, often called concurrency in this context, is to maintain a set of separate tasks interacting with an external environment. Of course, reactive constructs can also be used to express parallel transformational systems but the set of possible systems is wider than in the previous group. Non-determinism inevitably appears in these systems and the referential transparency of functional languages may be lost.

Typically, languages in this group offer constructs not only for the creation of processes but also for communicating and synchronising them. Most parallelism issues are treated explicitly on a low level of abstraction. Languages in this group are, for example, *FACILE* (Giacalone *et al.*, 1990),*Concurrent ML* (Reppy, 1991), *Erlang* (Wikström, 1994), *Concurrent Haskell* (Peyton Jones *et al.*, 1996) and its distributed variant *Glasgow distributed Haskell (GdH)* (Pointon *et al.*, 2001).

Classifying Eden into these groups poses some difficulties discussed in the following.

6.3 Discussion

Considering only process abstractions and instantiations, Eden could be classified into the group of transformational languages, in particular into the sub-group of annotated languages, because its basic coordination constructs can be viewed as special annotations. Whereas GpH and Concurrent Clean use parallelism annotations indicating only *potential* parallelism, which need not be exploited by the runtime system, Eden's process instantiation will definitely lead to the creation of a new process. This gives the programmer direct control over parallel evaluation and an indirect control over data distribution. Caliban introduces separate wiring constructs which are combined with application code to construct static process networks. As in Eden, processes and their topology are explicitly handled in Caliban, but the focus is on extracting the static process topology during compile time and on computing an optimal mapping on the parallel destination architecture. In contrast to Caliban, Eden – and also GpH and Concurrent Clean – supports the dynamic creation of parallel threads or processes, which gives the flexibility needed for handling irregular or dynamically evolving parallel systems.

Skeletons can be defined in Eden as abstractions over lower-level definitions of process systems, but Eden is not a skeleton language in the usual sense, because there are no pre-defined skeletons with a specialised implementation. Most skeleton languages work with pre-defined, specially supported sets of skeletons, and much care has been taken to identify minimal sets of skeletons (Cole, 2003). New classes of applications may, however, require new skeletons to obtain the best performance. Therefore, in our opinion, the best *minimum* set of skeletons is the one which is required to introduce and control parallelism in the host language. Higher-level patterns can then be constructed from this basic mechanism as we have shown in section 3.

Data-parallelism can be expressed in Eden, as the Eden skeleton *map* and other show. Also, data distribution can be specified by the programmer by controlling the free variables of process abstractions and the data communicated to processes through channels. Eden is, however, neither a data-parallel language, because it has no special support for data-parallel distribution primitives.

Because of `merge`, Eden can also be classified into the reactive languages group. Note, however, that a lot of typical low-level features of coordination languages such as sending and receiving messages are missing in Eden. In this respect, we agree with S. Gorlatch, who argues in a recent note (Gorlatch, 2004) that low-level coordination constructs like send and receive should be considered harmful and should be hidden inside higher-level schemes like collective communication operations or skeletons. Note that the `merge` process enables Eden to express reactive systems like the replicated-workers skeleton which implements a dynamic load balancing scheme. It is not possible to express this scheme in pure functional languages.

Summarising, Eden – a language with processes as first class values – perfectly fits in both subgroups of our classification.

Many publications on Eden, like Klusik *et al.* (2001b), Peña & Rubio (2001) and Loogen *et al.* (2002) show good runtime performance and speedups, most of them obtained on a high-latency Beowulf cluster. Comparative measurements of Eden and its sibling language GpH in Loidl *et al.* (2001) showed that the explicit process model favoured by Eden gave better parallel performance for coarse-grained applications running on a Beowulf cluster. The subsequent journal article (Loidl *et al.*, 2003) additionally considers the parallel functional ML-based language PMLS (Michaelson *et al.*, 2001) and compares the three languages with respect to their coordination constructs, runtime performance and programmer productivity. All three languages aim to provide higher-level models of parallelism, with the objective of reducing programmer overhead. In contrast to GpH and Eden, PMLS is a strict skeleton language which provides a set of pre-defined skeletons with associated parallel behaviours. As said before, skeleton languages are less flexible than general-purpose parallel languages like GpH and Eden, but will cause the lowest programmer overhead when applying a known scheme covered by the pre-defined skeletons. As anticipated, PMLS showed the lowest runtimes for the three benchmarks considered in the paper. Eden also showed good speedup results, but worse absolute sequential and parallel runtimes than GpH. The latter was due to different versions of the Glasgow Haskell compiler whose runtime system was substantially revised from version 3 (underlying the Eden compiler) to version 4 (underlying the GpH system). The article also contains comparative results with C+PVM benchmarks for parallel matrix multiplication. While the runtime of the sequential C program is a factor 4–6 less than the runtimes of the functional programs, the code size increases by the same factor. The speedup values progress in a similar way to the functional programs, but the ratio of lines-of-code of the parallel program versus the sequential program, which somehow indicates the development costs of the parallel programs is about 1:4 for the C + PVM programs and varies between 1:1.1 and 1:1.5 for the parallel functional programs. Thus, there is a clear trade-off between performance and productivity.

7 Conclusions

Eden has been designed in such a way that programmers get a reasonable degree of control over parallelism, but without low-level coordination constructs, which would make programs longer and more difficult to understand. Eden can be seen as a compromise between several extreme alternatives in the design space of parallel functional languages, e.g. implicitness vs. explicitness, laziness vs. eagerness, or determinism vs. nondeterminism.

Eden programs are not intended to be written always from scratch. In a way similar to the rich set of higher-order functions provided by Haskell's prelude, a rich set of skeletons is provided by Eden's library. They cover many common patterns of parallel algorithms such as *parallel map*, *parallel divide-and-conquer*, *map-and-reduce*, *parallel search*, *iterate-until* and others, as well as typical process topologies like grids, tori, rings, pipelines, and the like. For the vast majority of problems, the task of the Eden programmer is to choose or to adapt one of the predefined skeletons and to instantiate it with appropriate problem-dependent parameters. Only a few problems may require the explicit definition of process abstractions and instantiations in the program text. Even in this case, the recommended methodology is to try to separate the problem-dependent aspects from the coordination aspects. The latter can then be embedded in a polymorphic problem-independent skeleton which can be added to Eden's library to be used in future programs. This separation of concerns is also useful for reasoning, documentation, and testing purposes.

A big advantage of Eden is the replicated workers skeleton, which provides dynamic load balancing and often yields substantial performance gains in comparison with purely static schemes like tasks farms. Eden gets this additional expressive power in comparison to other transformational functional languages like GpH or Concurrent Clean from its non-deterministic `merge` process.

No special parallel hardware is required to run Eden programs. A couple of personal computers, connected by a mini-hub or by ethernet, and running Linux + PVM is sufficient. In the near future it will be more and more common to have desktop computers with two or four processors. Eden provides an easy way of exploiting the power of these machines. Computation-intensive Haskell programs will easily be converted into parallel Eden programs.

When compared to their counterpart programs directly written in C + PVM, there will be some runtime overhead due to the more sophisticated runtime system. This is the typical trade-off between low-level control and high-level expressiveness. The main advantage of the latter lies in the higher productivity of programmers, as programs can be easier developed, verified and maintained. Consequently, we can state that. in Eden, parallel functional programs which show good speedups can be obtained with low effort.

Acknowledgements

The authors would like to thank the following organisations for funding the work presented here: the Spanish Ministries of Education and Science (MEC) and of Science and Technology (MCYT) as well as the Madrid Region for financing the projects 06T/033/96, TIC97-0672 and TIC2000-0738, and the German Academic Exchange Service (DAAD), its spanish counterpart and the British Council for supporting the Spanish-German Acciones Integradas: HA1995-142B, HA1996-0147, HA1997-0107 as well as the British-Spanish Accion Integrada HB1999-0102 and the British-German ARC project 313-ARC-XV-01.

Moreover, we wish to thank all current and past members of the Eden working groups in Madrid and Marburg, who contributed in many respects to the work presented here. We enumerate them in alphabetical order: Jost Berthold, Silvia Breitinger, Alberto de la Encina, Ralf Freitag, Luis Antonio Galán, Mercedes Hidalgo, Ulrike Klusik, Rafael Martínez, Manuel Núñez, Pedro Palao, Cristóbal Pareja, Mari Plümacher, Steffen Priebe, Pablo Roldán, Fernando Rubio, Clara Segura and Nils Weskamp. Many fruitful discussions and impulses also came from our project partners in Scotland and Munich, in particular Kevin Hammond, Greg Michaelson, Phil Trinder, and Hans-Wolfgang Loidl.

We are grateful to the anonymous referees as well as to Jost Berthold, Mercedes Hidalgo-Herrero, Fernando Rubio and Steffen Priebe for valuable comments on a preliminary version of this paper.

References

Adams, W. W. and Loustaunau, P. (1994) *An Introduction to Gröbner Bases.* American Mathematical Society.

Amdahl, G. M. (1967) Validity of the single processor approach to achieving large-scale computing capabilities. *AFIPS Conference Proceedings 1967, Vol. 30*, pp. 483–485. Thompson Books.

Bacci, B., Gorlatch, S., Lengauer, C. and Pelagatti, S. (1999) Skeletons and transformations in an integrated parallel programming environment. *PACT'99 – International Conference on Parallel Architecture and Compilations Techniques: Lecture Notes in Computer Science 1662*, pp. 13–27. Springer.

Baker-Finch, C., King, D. J., Hall, J. G. and Trinder, P. W. (2000) An operational semantics for parallel lazy evaluation. *ICFP'00 – International Conference on Functional Programming*, pp. 162–173. ACM.

Berthold, J. (2004) Dynamic chunking in Eden. *Implementation of Functional Languages, IFL 2003, Revised Selected Papers: Lecture Notes in Computer Science 3145.* Springer.

Berthold, J., Klusik, U., Loogen, R., Priebe, S. and Weskamp, N. (2003) High-level process control in Eden. *Euro-Par 2003 – Parallel Processing: Lecture Notes in Computer Science 2790*, pp. 732–741. Springer.

Blelloch, G. E. (1996) Programming parallel algorithms. *Comm. ACM*, **39**(3), 85–97.

Bratvold, T. A. (1993) A skeleton-based parallelising compiler for ML. *IFL'93 – 5th International Workshop on the Implementation of Functional Languages*, pp. 23–33. University of Nijmegen.

Breitinger, S., Klusik, U., Loogen, R., Ortega-Mallén, Y. and Peña Marí, R. (1997a) DREAM – the DistRibuted Eden Abstract Machine. *Selected Papers of IFL '97 – International Workshop on the Implementation of Functional Languages: Lecfture Notes in Computer Science 1467*, pp. 250–269. Springer.

Breitinger, S., Loogen, R., Ortega-Mallén, Y. and Peña Marí, R. (1997b) The Eden Coordination Model for Distributed Memory Systems. *HIPS'97 – Workshop on High-level Parallel Programming Models*, pp. 120–124. IEEE Press.

Breitinger, S., Klusik, U. and Loogen, R. (1998) From (Sequential) Haskell to (Parallel) Eden: An implementation point of view. *PLILP'98 – Programming Languages: Implementation, Logics and Programs: Lecture Notes in Computer Science 1490*, pp. 318–334 Springer.

Chandra, R. (2000) *Parallel Programming in OpenMP*. Morgan Kaufmann.

Cole, M. (1989) *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press.

Cole, M. I. (2003) *The eSkel Reference Manual*. University of Edinburgh. `http://www.dcs.ed.ac.uk/home/mic/eSkel/eSkelmanual.ps`.

Darlington, J., Field, A. J., Harrison, P. G., Kelly, P. H. J., Sharp, D. W. N., Wu, Q. and While, R. L. (1993) Parallel programming using skeleton functions. *PARLE'93 – Parallel Architectures and Languages Europe: Lecture Notes in Computer Science 694*, pp. 146–160. Springer.

Gaudiot, J.-L., DeBoni, T., Feo, J., Böhm, W., Najjar, W. and Miller, P. (2001) The Sisal Project: Real world functional programming. *Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques, and Run Time Systems Languages: Lecture Notes in Computer Science 1808*, pp. 45–72 Springer.

GHC (1993) *Glasgow Haskell Compiler*. `http://www.haskell.org/ghc`.

Giacalone, A., Mishra, P. and Prasad, S. (1990) Operational and Algebraic Semantics for Facile: A Symmetric Integration of Concurrent and Functional Programming. *International Colloquium on Automata, Languages and Programming (ICALP 90): Lecture Notes in Computer Science 443*. Springer.

Gorlatch, S. (2004) Send-Receive Considered Harmful: Myths and Realities of Message Passing. *ACM Trans. Pogram. Lang. & Syst.* **26**(1), 47–56.

Hamdan, M. (2000) *A Combinational Framework for Parallel Programming Using Algorithmic Skeletons*. PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University.

Hammond, K. and Michaelson, G. (eds). (1999) *Research Directions in Parallel Functional Programming*. Springer.

Hammond, K., Loidl, H. W. and Partridge, A. (1995) Visualising granularity in parallel programs: a graphical winnowing system for Haskell. In: Bohm, A. P. W. and Feo, J. T., editors, *HPFC'95, High Performance Functional Computing*, pp. 208–221.

Hammond, K., Berthold, J. and Loogen, R. (2003) Automatic Skeletons in Template Haskell. *Parallel Process. Lett.* **13**(3), 413–424.

Hernández, F., Peña, R. and Rubio, F. (2000) From GranSim to Paradise. *SFP'99 – Scottish Functional Programming Workshop*, pp. 11–19. Trends in Functional Programming. Intellect.

Herrmann, C. (2000) *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions.* PhD thesis, University of Passau.

Hidalgo-Herrero, M. and Ortega-Mallén, Y. (2001) A distributed operational semantics for a parallel functional language. *SFP'00 – Scottish Functional Programming Workshop*, pp. 101–116. Trends in Functional Programming, Vol. 2. Intellect.

Hidalgo-Herrero, M. and Ortega-Mallén, Y. (2002) An operational semantics for the parallel language Eden. *Parallel Process. Lett.* **12**(2), 211–228.

Hidalgo-Herrero, M. and Ortega-Mallén, Y. (2003) Continuation semantics for parallel Haskell dialects. *APLAS'03 – the First Asian Symposium on Programming Languages and Systems: Lecture Notes in Computer Science 2895*, pp. 303–321. Springer.

Hidalgo-Herrero, M. (2004) *Formal Semantics for a Parallel Functional Language.* PhD thesis, Universidad Complutense de Madrid. (In spanish.)

Horowitz, E. and Sahni, S. (1978) *Fundamentals of Computer Algorithms.* Pitman.

Kelly, P. H. J. (1989) *Functional Programming for Loosely-Coupled Multiprocessors.* Research Monographs in Parallel and Distributed Computing. MIT Press.

Klusik, U., Ortega-Mallén, Y. and Peña Marí, R. (1999) Implementing Eden – or: Dreams Become Reality. *IFL'98 – International Workshop on the Implementation of Functional Languages: Lectuure Notes in Computer Science 1595*, pp. 103–119. Springer.

Klusik, U., Peña, R. and Segura, C. (2000) Bypassing of Channels in Eden. *SFP'99 – Scottish Functional Programming Workshop*, pp. 2–10. Trends in Functional Programming. Intellect.

Klusik, U., Loogen, R. and Priebe, S. (2001a) Controlling Parallelism and Data Distribution in Eden. *SFP'00 – Scottish Functional Programming Workshop*, pp. 53–64. Trends in Functional Programming, Vol. 2. Intellect.

Klusik, U., Loogen, R., Priebe, S. and Rubio, F. (2001b) Implementation Skeletons in Eden – Low-Effort Parallel Programming. *IFL'00 – International Workshop on the Implementation of Functional Languages: Lecture Notes in Computer Science 2011*, pp. 71–88. Springer.

Klusik, U., Peña, R. and Rubio, F. (2002). Replicated Workers in Eden. *CMPP'00 – Constructive Methods for Parallel Programming*. Advances in Computation: Theory and Practice. Nova Science Books and Journals.

Launchbury, J. (1993) A natural semantics for lazy evaluation. *POPL'93 — ACM Symposium on Principles of Programming Languages*, pp. 144–154. ACM.

Loidl, H.-W., Klusik, U., Hammond, K., Loogen, R. and Trinder, P. W. (2001) GpH and Eden: Comparing two parallel functional languages on a Beowulf cluster. *SFP'00 – Scottish Functional Programming Workshop*, pp. 39–52. Trends in Functional Programming, Vol. 2. Intellect.

Loidl, H.-W., Rubio, F., Scaife, N. R., Hammond, K., Klusik, U., Loogen, R., Michaelson, G. J., Horiguchi, S., Peña Marí, R., Priebe, S. M., Rebón Portillo, A. J. and Trinder, P. W. (2003) Comparing parallel functional languages: Programming and performance. *Higher-order & Symbolic Computation*, **16**(3), 203–251.

Loogen, R., Ortega-Mallén, Y., Peña, R., Priebe, S. and Rubio, F. (2002) Parallelism abstractions in Eden. In: Rabhi, F. A. and Gorlatch, S., editors, *Patterns and Skeletons for Parallel and Distributed Computing*, pp. 95–128. Springer.

Martínez, R. and Peña, R. (2004) Building an interface between Eden and Maple: A way of parallelizing computer algebra algorithms. *Implementation of Functional Languages, IFL 2003, Selected PapersLecture Notes in Computer Science 3145*. Springer.

Michaelson, G., Scaife, N., Bristow, P. and King, P. (2001) Nested algorithmic skeletons from higher order functions. *Parallel Algorithms and Applications*, **16**, 181–206. (Special Issue on High Level Models and Languages for Parallel Processing.)

MPI (1997) *MPI-2: Extensions to the Message-Passing Interface.* `http://www-unix.mcs.anl.gov/mpi`.

Nikhil, R. S., Arvind, Hicks, J., Aditya, S., Augustsson, L., Maessen, J.-W. and Zhou, Y. (1995) *pH Language Reference Manual.* Technical report CSG Memo 369, Laboratory for Computer Science, M.I.T.

Nöcker, E., Smetsers, J., van Eekelen, M. and Plasmeijer, M. J. (1991) Concurrent Clean. *PARLE'91 – Parallel Architectures and Languages Europe: Lecture Notes in Computer Science 505*, pp. 202–219. Springer.

Pareja, C., Peña, R., Rubio, F. and Segura, C. (2001) Optimizing Eden by Transformation. *SFP'00 – Scottish Functional Programming Workshop*, pp. 13–26. Trends in Functional Programming, Vol. 2. Intellect.

Peña, R. and Segura, C. (2004) Non-determinism Analysis in a Parallel-Functional Language. *J. Funct. Program.* To appear.

Peña, R. and Rubio, F. (2001) Parallel Functional Programming at Two Levels of Abstraction. *PPDP'01 – International Conference on Principles and Practice of Declarative Programming*, pp. 187–198. ACM.

Peyton Jones, S. L. (1992) Implementing lazy functional languages on stock hardware: the Spineless Tagless G-Machine. *J. Funct. Program.*, **2**(2), 127–202.

Peyton Jones, S. L. (1996) Compiling Haskell by program transformation: a report from the trenches. *ESOP'96 – European Symposium on Programming: Lecture Notes in Computer Science 1058*, pp. 18–44. Springer.

Peyton Jones, S. L. and Hughes, J. (1999) *Haskell 98: A Non-strict, Purely Functional Language.* Available at `http://www.haskell.org/`.

Peyton Jones, S. L., Gordon, A. and Finne, S. (1996) Concurrent Haskell. *POPL'96 – ACM Symposium on Principles of Programming Languages*, pp. 295–308. ACM.

Pointon, R. F., Trinder, P. W. and Loidl, H.-W. (2001) The design and implementation of Glasgow distributed Haskell. *IFL'00 – International Workshop on the Implementation of Functional Languages: Lecture Notes in Computer Science 2011*, pp. 101–116. Springer.

PVM (1993) *Parallel Virtual Machine Reference Manual, Version 3.2.* University of Tennessee. `http://www.csm.ornl.gov/pvm/`.

Reppy, J. H. (1991) CML: a Higher-Order Concurrent Language. *PLDI'91 – Programming Languages Design and Implementation*, pp. 293–305. ACM.

Roldán Gómez, P. (2004) *The Eden Trace Viewer Tool.* Diploma Thesis, Philipps-University Marburg and Universidad Complutense de Madrid.

Scholz, S.-B. (1996) *Single Assignment C – Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen (in German).* PhD thesis, Institut für Informatik und praktische Mathematik, Universität Kiel.

Sheard, T. and Peyton-Jones, S. (2002) Template Meta-programming for Haskell. *Proc. of the Haskell Workshop*, pp. 1–16. ACM.

Trinder, P. W., Hammond, K., Mattson Jr., J. S., Partridge, A. S. and Peyton Jones, S. L. (1996) GUM: a Portable Parallel Implementation of Haskell. *PLDI'96 – Programming Language Design and Implementation*, pp. 78–88. ACM.

Trinder, P. W., Hammond, K., Loidl, H.-W. and Peyton Jones, S. L. (1998) Algorithm + Strategy = Parallelism. *J. Funct. Program.* **8**(1), 23–60.

Trinder, P. W., Loidl, H.-W. and Pointon, R. F. (2002) Parallel and Distributed Haskells. *J. Funct. Program.* **12**(4&5), 469–510.

Wikström, C. (1994) Distributed Programming in Erlang. In: Hong, H., editor, *PASCO'94: First International Symposium on Parallel Symbolic Computation*, pp. 412–421. World Scientific.