Parallel Functional Programming at Two Levels of Abstraction

Ricardo Peña and Fernando Rubio
Departamento Sistemas Informáticos y Programación
Universidad Complutense de Madrid
Ciudad Universitaria s/n, 28040 Madrid, Spain
Phone: 34 91 394 (4313,4381) Fax: 34 91 394 4602
(ricardo,fernando)@sip.ucm.es

ABSTRACT

The parallel functional language Eden extends Haskell with expressions to define and instantiate process systems. These extensions allow also the easy definition of skeletons as higher-order functions. Parallel programming is possible in Eden at two levels: Recursive programming and higher-order programming. At the lower level, processes are explicitly created by using recursive definitions. In this way, skeletons can be defined. This is very unusual, as most skeleton-based languages use an imperative language to create new skeletons. At the higher level, available skeletons are used to create applications or to define new skeletons on top of the other ones. In this paper, we present five skeletons, most of them well-known, covering a wide range of parallel structures. For each one, several Eden implementations are given, together with their corresponding cost models. Finally, some examples of application programming are shown, including predicted and actual results on a Beowulf cluster.

Keywords

Parallel functional programming, skeletons, cost models.

1. INTRODUCTION

Parallel functional programming is an active area of research. On the one hand, functional languages are less efficient than imperative ones, but on the other hand they offer better opportunities for parallelization due to their absence of side effects and to the large amount of parallelism implicit in their evaluation mode. However, it is not easy to exploit this parallelism due to several practical problems such as controlling granularity data distribution, process placement, and load balancing between processors.

Eden [6] is a parallel extension of the lazy functional language Haskell [29] which provides 'controlled' parallelism, as opposed both to more implicit approaches such as GpH (Glasgow Parallel Haskell) [33] and to more explicit ones such as *Concurrent ML* [32]. In Eden, the programmer decides which expressions are processes and how processes are connected, having explicit control over the communication topology. The remaining aspects such as sending and receiving messages, process placement, data distribution, etc. are implicitly controlled by the runtime system. In GpH, there is no explicit notion of process. The programmer just annotates expressions as candidates for parallel execution and it is the runtime system that decides which ones will be actually executed in a different processor. In *Concurrent ML*, the programmer is responsible for communicating processes by explicitly sending and receiving messages.

Other controlled approaches to parallel functional programming are the so called *skeleton-based* languages [2, 8, 9, 31, 22, 15]. A skeleton [7] corresponds to a parallel *problem solving scheme* or process topology applicable to certain families of problems. Specific functions or processes used in the nodes of the topology can be abstracted as parameters. For the functional programmer, they look just like higher-order functions. In these languages, a library of predefined skeletons is provided, each one with several implementations tuned for different architectures. They are normally implemented at low level by using an *imperative language* and they are the only parallel constructions, so application programmers must express their algorithms in terms of them. This means that, should a new skeleton be needed, then the *system programmer* should add a new piece of imperative code to the library while the *application programmer* should wait for the task to be completed. The skeleton approach is not specific to the functional paradigm. There has been much research along this idea in the imperative field (e.g. see [23, 24, 1]).

Eden can achieve the benefits of having skeletons, the difference with the above mentioned approaches being that skeletons can be implemented in the language *itself*. Thus, programmers do not need to move from one language to another in order to create a new skeleton. They can continuously enlarge the skeleton library with new skeletons specific to different application areas. There is a strong similarity between creating skeletons in Eden and creating new higher-order functions in a sequential functional language. The idea of defining skeletons inside a functional language has also been explored in GpH [14], the problem being that the language did not give enough control to really create the skeletons properly.

	System area	Application area
Higher-order	Skeletons in terms of skeletons	Programs in terms of skeletons
Recursive	Skeletons in terms of processes	Programs in terms of processes

Figure 1: Eden as a two-level, two-area language

Parallel programming in Eden can be done by explicitly instantiating processes. This would be equivalent to doing sequential functional programming with explicit recursion. Sometimes this cannot be avoided, but an experienced functional programmer will try to use higher-order functions as much as possible in order to reduce the amount of work and the possibility of making mistakes. Eden can then be regarded as both a *two-level* and a *two-area* language (see Figure 1):

Recursive programming Processes are defined and instantiated explicitly. In this level, the *application* programmer expresses his algorithm as an *ad-hoc* process topology. This may be needed when no available skeleton nicely meets the algorithm the programmer has in mind. Also in this level, the *system* programmer abstracts a common problem solving scheme and writes it generically as a new skeleton by explicitly defining a process topology.

Higher-order programming The application programmer designs parallel algorithms by instantiating available skeletons. Also system programming is possible at this level if a new skeleton is defined by using previously defined ones.

In a complex application both levels and areas may be simultaneously needed. This is also possible as the language does not pose barriers between them. In this paper, we present five skeletons covering a wide range of parallel structures. For each one, several Eden implementations are given, together with their corresponding cost models. The main contributions of the paper are the following:

- To show how different skeletons, and radically different implementations of them, can be concisely defined in a parallel functional language like Eden. This is the first time that these skeletons are *really* implemented and run in a parallel functional language. This makes Eden a *system* programming language.
- To show some examples of higher-order application programming written on top of the previously defined skeletons. Speedups predicted by the cost models and actual results obtained on a Beowulf cluster are also included.
- To show how all these programming levels (higher-order vs. recursive) and areas (system vs. application) can be smoothly integrated in a single programming language. This forms the kernel of a methodology for doing parallel programming with low effort in a functional framework.

This work can be considered as a continuation of [19, 17], where previous versions of the `map` skeletons were presented.

We do not claim that parallel functional programming (PFP) can achieve better efficiency than its imperative counterpart. The reason is that Eden abstracts many runtime details and so, some low-level optimizations cannot be done at the language level. We rather claim that PFP in Eden can get *acceptable speedups with comparatively less effort*, the reason being exactly the same, i.e. that Eden abstracts many low-level details that an imperative programmer cannot ignore: explicit communication and synchronization, sequence control, process placement, etc. We use the term 'acceptable' as a synonym of 'reasonably good but not necessarily optimal'.

Another important feature of PFP in Eden is that the differences between the sequential and the parallel versions of an algorithm are usually minimal: In the simplest case, transforming a sequential algorithm written in Haskell into a parallel one written in Eden may amount to just replacing some function applications $e_1\ e_2$ by process instantiations $e_1 \# e_2$ (see example of parallel *mergesort* below). If skeletons are being used, the changes may consist of using a parallel version of *map* or of *divide and conquer*, instead of the corresponding sequential versions.

The paper is organized as follows: After this introduction, Section 2 summarizes Eden's main features; Section 3 is the kernel of the paper, in which skeletons in Eden are described; Section 4 shows examples of how to use the skeletons, together with predicted and runtime results for them. The paper ends with a conclusion.

2. EDEN

2.1 Basic Constructions

Eden [6, 5] extends the lazy functional language Haskell by syntactic constructs that explicitly define processes. There exists a new expression `process x -> e` of a predefined type `Process a b` to define a *process abstraction* having variable `x::a` as input and expression `e::b` as output. Process abstractions of type `Process a b` can be compared to functions of type `a -> b`, the main difference being that the former, when instantiated, are executed in parallel. Additionally, when the output (resp. input) expression is a tuple, i.e. `e::(t1,...,tn)` a separate concurrent thread is created for the evaluation of each tuple element (we will refer to each tuple element as a *channel*).

A *process instantiation* is achieved by using the predefined infix operator `(#) :: (Transmissible a, Transmissible b) => Process a b -> a -> b`. Each time an expression `e1 # e2` is evaluated, the instantiating process will be responsible for evaluating and sending `e2`, while a new process is created to evaluate the application `(e1 e2)`. We will refer to the latter as the *child* process, and to the owner of the instantiation expression as the *parent* process. The instantiation protocol deserves some attention in order to explain Eden's semantics:

- Closure `e1` together with all its dependent closures are *copied* unevaluated to a new processor and the child process is created there to evaluate it. Notice that this can lead to duplication of work, but it is useful to increase the parallelism degree.
- Once created, the child process starts producing eagerly its output expression.

- Expression e2 is eagerly evaluated in the parent process, and the result value is eagerly transmitted to the child process. If it is a tuple, an independent concurrent thread is created to evaluate each component.

Once a process is running, only fully evaluated data objects are communicated. The only exceptions are lists, because they accurately represent the way in which communications really take place: they are transmitted in a *stream*-like fashion, i.e. element by element. Each list element is first evaluated to full normal form and then transmitted. Concurrent threads trying to access not yet available input are temporarily suspended. This is the only way in which Eden processes synchronize. To be able to transmit a value, its type must belong to the class `Transmissible`, which includes a function defining how to send a value. Compared to other approaches to PFP, we can summarize Eden's distinctive features as follows:

- Process abstractions in Eden are not just annotations but first class values which can be manipulated by the programmer (i.e. communicated through channels, stored in data structures, and so on). This facilitates the definition of skeletons as higher order functions.
- Process instantiations dynamically create processes. In general, the number of processes cannot be determined at compile time.
- In Eden, the concept of a virtually shared global graph does not exist. Each process evaluates its outputs autonomously with respect to other processes.
- Lazy evaluation is changed to eager in two cases: processes are eagerly instantiated when its closure is allocated, and instantiated processes produce their output even if it is not demanded. These modifications aim at increasing the parallelism degree and at speeding up the distribution of the computation.
- In general, a process is implemented by several threads concurrently running in the same processor, so that different values can be produced independently.

More details about Eden and its implementation can be found in [3, 18].

2.2 A Simple Example

The following Eden program specifies a parallel mergesort, where a parameter `h` controls the granularity by specifying the desired height of the process tree to be generated, as shown below.

```
parMSort :: (Ord a, Transmissible a) =>
            Int -> Process [a] [a]
parMsort h = process xs -> ys
  where ys | h == 0 = seqMsort xs
           | h > 0  = join (parMsort (h-1) # x1)
                           (parMsort (h-1) # x2)
           where (x1,x2) = unshuff xs
seqMsort []  = []
seqMsort [x] = [x]
seqMsort xs  = join (seqMsort x1) (seqMsort x2)
  where (x1,x2) = unshuff xs
unshuff = foldr (\x ~(xs1,xs2) -> (xs2,x:xs1)) ([],[])
```

Notice the difference between parameter `h`, which must be known at instantiation time, and the input `xs`, i.e. the list of values to be sorted, which will be gradually provided during the execution of the process. Ignoring parameter `h`, the similarity between the sequential and the parallel versions is remarkable. While the former makes two function applications —function `seqMsort`— the latter produces two process instantiations —process abstraction `parMsort (h-1)`. Function `unshuff` splits a list into two, and `join` merges two sorted lists into a single one.

2.3 Non-Determinism

Non-determinism is an essential feature for some applications, but it spoils the purity of functional languages. In Eden, the only source of non-determinism is the predefined process abstraction `merge :: Process [[a]] [a]` which is used to instantiate processes *fairly* merging a list of input channels, each one consisting of a list, to produce a single non-deterministic list. The implementation of `merge` immediately copies to its output list any value appearing at any of the input lists. In this way, `merge` can profitably be used to react quickly to requests coming in an unpredictable order from a set of processes.

In Eden parallel algorithms, `merge` is only used because of its reactive nature. Even though the skeletons presented are deterministic, some of them are required to react immediately to requests for work coming from a group of processes. An instantiation of `merge` will propagate these requests as they are being produced.

Functional purity can still be preserved in most portions of an Eden program due to a non-determinism analysis [26] detecting the expressions which are sure to be deterministic even in presence of `merge` instantiations.

2.4 Dynamic Channels

A process may generate a new *dynamic reply channel* and send a message containing its name to another process. The receiving process may then either use the received channel name to return some information 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.

Eden introduces a new unary type constructor `ChanName` for the names of the dymically created reply channels. Moreover, it also adds a new expression `new` (*ch_name*, *chan*) *e* which declares a new channel name *ch_name* as reference to the new input channel *chan*. The scope of both is the body expression *e*. The name should be sent to another process to establish the communication. A process receiving a reply channel name *ch_name*, and wanting to reply through it, uses an expression *ch_name* `!*` e_1 `par` e_2 Before e_2 is evaluated, a new concurrent thread for the evaluation of e_1 is generated, whose normal form result is transmitted via the reply channel. The result of the overall expression is e_2, while the communication through the dynamic channel is a side effect.

Dynamic channels are a non-functional feature and its denotational meaning is a difficult issue, as it needs to take into account the global state of the process system. In most situations —in particular in all the skeletons presented in this paper—, by using only process instantiations it is possible to create the same topologies that could be created by using dynamic channels, except for the fact that some channels will connect the intended processes through intermediate threads in other processes (see Section 3.3). By using dynamic channels, those will be *direct* connections. In this sense, this feature can be seen as an optimization using a low-level construct provided by the language rather than as a radically new concept.

2.5 Runtime System

Eden's compiler[1] has been developed by extending GHC [28, 27] (Glasgow Haskell Compiler), in order to reuse its efficiency and portability. Eden's RTS (Runtime System) is an implementation of the DREAM abstract machine [4] on top of a message passing library. In the current compiler, both PVM [11] and MPI [21] can be used. Therefore, the compiler can be ported to any architecture where GHC and either PVM or MPI are available.

Eden provides no *placement annotations*. However, 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 particular processor p, they are mapped to consecutive processors starting with the one numbered one more than p. *Random mode*: Each processor maps instantiated processes to randomly chosen processors. Notice that the first of them allows the programmer to control somehow the mapping of processes, as he can decide how many processes are created.

3. EDEN SKELETONS

The previous section gave an example of the recursive application area. In this section we concentrate on the skeleton (system) area. The first skeleton (`map`) will be used as the basis for some of the others. Due to lack of space, not all the code is included. The missing details and more explanations can be found in [25].

The definition of a skeleton consists of two parts: (1) A *specification* giving the type of the skeleton and a description of its observable behavior as a higher-order function; and (2) the *implementations*. A skeleton may have several implementations, and for each one two pieces must be provided: An implementation program (the *algorithm*), and a formula describing its expected parallel execution time (the *cost model*).

Cost models describe the parallel time of the algorithm (see e.g. [10]). By parallel time is usually meant the interval between the time the first processor starts to compute and the time the last processor finishes. Cost models will be parameterized by some constants that may depend either on the problem to be solved, or on the parallel architecture or RTS being used. We will use the parameters shown in Figure 2 and combine some of them into higher-level parameters:

$t_{unpackI} = \lambda + \beta.nwI$	CPU time for unpacking an input
$t_{unpackO} = \lambda + \beta.nwO$	CPU time for unpacking an output
$t_{packI} = \lambda + \beta.nwI$	CPU time for packing an input
$t_{packO} = \lambda + \beta.nwO$	CPU time for packing an output

The cost models presented in this section are an adaptation to Eden of classical cost models appearing in the literature. For a survey of cost models see e.g. [13].

[1] The Eden compiler is freely available at http://www.mathematik.uni-marburg.de/inf/eden

3.1 Map Skeleton

The specification is trivial:

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

3.1.1 Naïve Implementation

A process is created for every task in the list. The control of the exact moment in which processes are instantiated can be described by using evaluation strategies [34], that are functions demanding the evaluation of its argument to a certain degree, and returning always `()`. They are applied by means of function `using`, that first applies the strategy to the input value, and then return the value:

```
type Strategy a = a -> ()
using :: a -> Strategy a -> a
using x s = s x `seq` x
```

where `seq` is a Haskell primitive forcing the evaluation of its first parameter to weak head normal form, and then returning its second parameter. As we are interested in instantiating a list of processes, we should demand the spine of the list of process instantiations, using an appropiate strategy:

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

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

If the number of tasks is greater than the number of processors, several tasks will be created in each processor. For the cost model, we assume a round-robin distribution of processes onto processors and a uniform granularity of tasks:

$$\begin{array}{lcl} t_{map\_naive} & = & L_{init} + t_{processor} + L_{final} \\ L_{init} & = & P(t_{create} + t_{packI}) + \delta \\ L_{final} & = & \delta + t_{unpackO} \\ t_{processor} & = & \lceil \frac{N}{P} \rceil (t_{\#} + t_{unpackI} + t_f + t_{packO}) \end{array}$$

The first formula describes the critical path determining the parallel time. Before the last worker starts computing, P processes must be created and P messages must be packed in the parent. After the last worker finishes, an output message must first arrive at the parent (hence the δ latency) then be unpacked. The most heavily loaded processor will receive $\lceil \frac{N}{P} \rceil$ tasks. We are assuming that the remaining manager costs are outside of the critical path.

3.1.2 Farm Implementation

The previous version can be improved by reducing the number of processes to be created. In a `map_farm` a process is created for every processor and tasks are evenly distributed into processors. It is appropriate when task granularity is uniform, and an even distribution of a number of tasks amongst all the processors is desired. The length of the task list must be much higher than the number of available processors in order to improve the load balance. It creates an identical process in each of the available processors, distributes the tasks among them, applies function f to every task and collects the results. In order to instantiate processes correctly, it assumes that the round-robin mode is

Problem dependent parameters	
N	Size of the input
t_f	sequential CPU time for function f
nwI	number of words of input message going to a child
nwO	number of words of output message coming from a child
RTS dependent parameters	
t_{create}	CPU time in a parent processor to create a child process
$t_{\#}$	CPU time in a child processor to create a new process
Architecture dependent parameters	
P	Number of processors
δ	latency of a message, from start sending to start receiving
λ	start-up fixed CPU cost for sending or receiving a message
β	per-word CPU cost for sending or receiving a message

Figure 2: Parameters of the cost models

used by the RTS. The distribution and collection functions are also parameters of the skeleton. The `map_farm` implementation just distributes m tasks between n processors, assuming $m \gg n$, and collects the results. Here is its implementation, where a threshold parameter is added to instruct the skeleton when it must devote a separate processor to the parent.

```
map_farm :: (Transmissible a,Transmissible b) =>
             Int -> (a->b) -> [a] -> [b]
map_farm thr = farm np unshuffle shuffle
  where np | noPe > thr = (noPe-1)
           | otherwise  = noPe
farm :: (Transmissible a, Transmissible b) =>
        Int -> (Int->[a]->[[a]]) -> ([[b]]->[b]) ->
        (a->b) -> [a] -> [b]
farm np unshuffle shuffle f tasks
  = shuffle (map_naive (map f) (unshuffle np tasks))
```

where `noPe` is a constant giving the number of available processors. Different strategies to split the work into the different processes can be used provided that, for every list `xs`, `(shuffle . unshuffle n) xs == xs`. For instance, the following scheme distributes the tasks in a round-robin way:

```
unshuffle n ins
 | lf < n    = take n (map (:[]) firsts ++ repeat [])
 | otherwise = mzipWith' (:) firsts (unshuffle n rest)
 where (firsts, rest) = splitAt n ins
       lf             = length firsts
mzipWith' f [] ys = []
mzipWith' f (x:xs) ~(y:ys) = f x y : mzipWith' f xs ys
shuffle = concat . transpose
```

Notice that it is easy to proof that for every list `xs`, the equation `(shuffle . unshuffle) xs == xs` holds.

The cost model for `map_farm` is shown below.

$$
\begin{aligned}
t_{map\_farm} &= L_{init} + t_{worker} + L_{final} \\
L_{init} &= P(t_{create} + t_{packI} + t_{unshuffle_1}) + \delta \\
L_{final} &= \delta + t_{unpackO} + t_{shuffle_1} \\
t_{worker} &= t_{\#} + \lceil \tfrac{N}{P} \rceil (t_{unpackI} + t_f + t_{packO})
\end{aligned}
$$

Compared to the previous model, in this one the costs of shuffling and unshuffling are added. We are assuming that the CPU time not shown in the parent (i.e. unshuffling and shuffling the rest of the messages) is so negligible that it does not affect the critical path. If this were not the case, we would assign a separate processor to the parent and these times will be then outside the critical path. This is done in some of the algorithms presented below when run with a large number of processors. The price to be paid is devoting a single processor to the parent.

3.1.3 Replicated Workers Implementation

The load balance obtained using the farm scheme can be poor in three cases: (1) When the granularity of the tasks is not uniform; (2) when the processors' architecture is not regular; and (3) when the programs share CPU time with other programs in the same processors.

The solution to the three situations can be the same: Distributing work on demand, i.e. a new task must be assigned to a process only if it is known that it has already finished its previous work. This idea gives rise to the *replicated workers* skeleton [19]. Initially, the manager assigns one or more tasks to each of the workers. By assigning several tasks, idle time between tasks is minimized. Each time a worker finishes a work, it sends an acknowledgment message to the manager including the task result, and a new task (if any is available) is assigned to that process. The computation finishes when the manager has received all the task results.

The programmer cannot predict in advance the order in which processes are going to finish their work, as this depends on runtime issues. By using the process `merge`, acknowledgments from different processes can be received by the manager as soon as they are produced. Thus, if each acknowledgment contains the identity of the sender process, the list of merged results can be scrutinized in order to know who has sent the first message, and a new work 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 2.3).

The skeleton receives as input parameters (1) the number of worker processes to be used; (2) the size of workers' prefetching buffer; (3) the worker function; and (4) the list of tasks.

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

  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)
worker f i = process ts -> map f' ts
  where f' (id_t,t) = ACK i id_t (f t)
```

where sortMerge is a simple Haskell function not shown which merges the workers lists (each of them already sorted) producing a single list sorted by task identity. For this reason, the skeleton is completely deterministic seen from the outside. In order to implement map, a worker is created for every processor, and a prefetch parameter of 2 is used. Also, a threshold parameter must be provided.

```
map_rw :: (Transmissible a,Transmissible b) =>
          Int -> (a -> b) -> [a] -> [b]
map_rw thr | noPe > thr = rw (noPe-1) 2
           | otherwise  = rw noPe 2
```

Assuming $N \gg P$, the cost model for map_rw is the following:

$$\begin{array}{lcl} t_{map\_rw} & = & L_{init} + t_{worker} + L_{final} \\ L_{init} & = & P(t_{create} + t_{packI} + t_{distribute_1}) + \delta \\ L_{final} & = & \delta + t_{unpackO} + t_{sortMerge_1} \\ t_{worker} & = & t_{\#} + \frac{N}{P}(t_{unpackI} + t_{comp} + t_{packO}) \\ t_{comp} & = & \frac{1}{N}\sum_{i=1}^{N} t_{f_i} \end{array}$$

The considerations made in map_farm cost model are also applicable here. In the formula, t_{f_i} represents the sequential CPU time for function f when applied to task i. In $t_{distribute_1}$ we consider accumulated the previous costs of zip, concat and replicate functions for producing one element. Notice that the ceiling operation has disappeared from $\frac{N}{P}$. We are assuming a perfect load balance and it can be considered that every worker receives the exact average number of tasks, each one costing the average computing cost t_{comp}.

3.2 Divide and Conquer Skeleton

The sequential specification of this scheme is also well-known:

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

Notice that the resulting call tree need not be homogeneous and that trivial solutions may appear at any level of the tree.

3.2.1 Naïve Implementation

Here, a dynamic tree of processes is created in which each process is connected to its parent. An integer parameter determines the maximum level after which no more children processes are generated, and the sequential version is used instead. The implementation is as follows:

```
dc_naive :: (Transmissible a, Transmissible b) =>
            Int -> (a->Bool) -> (a->b) -> (a->[a]) ->
            (a -> [b] -> b) -> a -> b
dc_naive 0 trivial solve split combine
  = dc trivial solve split combine
dc_naive d trivial solve split combine x
  | trivial x   = solve x
  | otherwise   = combine x children
 where children = map_naive(dc_naive (d-1) trivial solve
                            split combine) (split x)
```

Notice in this implementation that there is no single manager process, as it happened in previous skeletons, because every child is a parent process of the next process level. The cost model for this implementation should assume a distribution of processes into processors and another distribution of granularity into processes. The first distribution is known, as it depends only on the RTS, but the second one depends on the concrete problem. Thus, the cost model should be able to work with any distribution function. Unfortunately, this is a hard problem in the general case, and the predictions that could be obtained would not be very accurate. We are working on a cost model parameterized by a distribution, but it is not yet satisfactory enough.

3.2.2 Farm and Replicated Workers Implementations

These implementations use the farm and rw implementations of map in order to have a better control over process and granularity distribution and to achieve a better load balance. Also, the overhead will be decreased as only one process per processor will be created. The original task is split up to a given depth and, for every subtree at this depth, a subtask is created. The list of subtasks is given to a map_farm (or better a map_rw) skeleton in which the transformer function is just the sequential algorithm. An explicit tree of arguments must be generated when splitting the initial task in order to be able to combine the results returned by the parallel processes. We present only the dc_rw skeleton. The corresponding dc_farm skeleton can be obtained replacing map_rw by map_farm. Functions generateTasks and combineTop are simple Haskell definitions not shown. As far as we know, this implementation of dc is original to this paper.

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

The cost models for these implementations of dc are mainly those of map_rw and map_farm (see Section 3.1), adding the overheads of creating tasks and combining results in the parent process. To use the models, we must know the number N of tasks generated and the average computation time t_{comp}, which can be easily obtained from the sequential time. In Section *4.2.2* both dc_naive and dc_rw are used and compared for a typical divide and conquer algorithm.

3.3 Pipeline Skeleton

A pipeline consists of a list of stages, where each stage applies a different function to the results obtained in the previous stage. This can be expressed by means of a *folding* function:

```
pipe :: [[a] -> [a]] -> [a] -> [a]
pipe  = foldl1 (flip (.))
```

In order to extract parallelism, we have forced in the type that each function must consume and produce a list.

3.3.1 Implementation

The naïve parallelization of this scheme can be performed by instantiating a different process to evaluate each of the

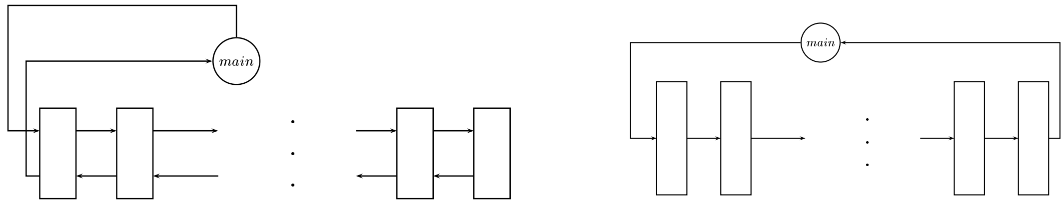

Figure 3: Topology generated with `pipe_naive` (left), and the desired pipeline using `pipe_naiveD` (right)

pipeline stages. This can be expressed in Eden in several ways. For instance, the following one, in which each process in the pipe creates the next process of the pipe:

```
pipe_naive :: Transmissible a => [[a]->[a]] -> [a] -> [a]
pipe_naive fs xs  = (ppipe fs) # xs

ppipe :: Transmissible a => [[a]->[a]] -> Process [a] [a]
ppipe [f]    = process xs -> f xs
ppipe (f:fs) = process xs -> (ppipe fs) # (f xs)
```

This definition does not achieve the desired topology because the last process of the pipe cannot send the values directly to the main process, as a hierarchical topology is created (see Figure 3(left)). In Eden, a *bypassing analysis* [20] tries to detect and correct this problem, but currently this definition is too complex for it. A better definition makes use of Eden's *dynamic channels* facility (see Section 2). The first process creates a dynamic channel that will be used by the last process for sending the final values. Intermediate processes just forward the name of that channel to the next process. The created topology is then that of Figure 3(right). The similarities between both versions are remarkable. In fact, in [25] we give a method to obtain this kind of skeleton using as specification skeletons such as the one above.

```
pipeD :: Transmissible a=>[[a]->[a]] -> [a] -> [a]
pipeD [f] xs = process xs -> f xs
pipeD fs xs
 = new (cn,c) let something = (ppipeD fs) # (xs,cn) in c

ppipeD :: Transmissible a =>
          [[a]->[a]] -> Process ([a], ChanName [a]) ()
ppipeD [f]    = process (xs,cn) -> cn !* (f xs) par ()
ppipeD (f:fs) = process (xs,cn) -> (ppipeD fs)#(f xs,cn)
```

The cost model for `pipeD` is the following:

$$
\begin{array}{lcl}
t_{pipe\_naive} & = & L_{init} + t_{worker} + L_{final} \\
L_{init} & = & F(t_{create} + t_{\#} + t_{packI} + \delta) \\
L_{final} & = & \delta + t_{unpackO} \\
t_{worker} & = & \lceil \frac{F}{P} \rceil N(t_{unpackI} + max\{t_{comp_i}\}_{i=1}^{F} + t_{packO})
\end{array}
$$

where F is the number of functions in the pipe, N the length of the input list and t_{comp_i} the cost of function f_i for processing a single element of the list. We assume $F > P$ and the round-robin mode for the RTS.

3.4 Torus Skeleton

A torus is a well-known two-dimensional topology in which each process is connected to its four neighbors. The first and last processes of each row and column are considered neighbors. In addition, each node has two extra connections to send and receive values to/from the parent. At each round, every worker receives messages from its left and upper neighbors, computes, and then send messages to its right and lower neighbors. Eden's implementation uses lists instead of synchronization barriers to simulate rounds. It also uses dynamic channels to provide direct connections between workers. The `torus` function creates the desired toroidal topology by properly connecting the inputs and outputs of the different `ptorus` processes. Each `ptorus` receives an input from the parent, and two channel names to be used to send values to its siblings, and produces an output to the parent and two channel names to be used to receive inputs from its siblings. The source code follows.

```
torus :: (Transmissible a,Transmissible b,
         Transmissible c,Transmissible d) =>
         ((c,[a],[b]) -> (d,[a],[b])) -> [[c]] -> [[d]]
torus f toChildren = outssToParent   where
 outss   = [[(ptorus f) # outAB | outAB <- outs'] |
                 outs' <- outss']
 (outssToParent,outssA,outssB)=unzip3 (map unzip3 outss)
 outssA' = mzipWith (:) nrows (map last outssA)
                              (map init outssA)
 outssB' = last outssB : init outssB
 outss'  = mzipWith3 mzip3 toChildren outssA' outssB'
 nrows   = length toChildren

-- each individual process of the torus
ptorus ::(Transmissible a,Transmissible b,
         Transmissible c,Transmissible d) =>
         ((c,[a],[b]) -> (d,[a],[b])) ->
         Process (c,ChanName [a],ChanName [b])
                 (d,ChanName [a],ChanName [b])
ptorus f = process (fromParent,outChanA,outChanB) -> out
 where out= new (inChanA, inA) new (inChanB, inB)
        let (toParent,outA,outB) = f(fromParent,inA,inB)
        in outChanA !* outA par outChanB !* outB par
             (toParent,inChanA,inChanB)
```

The number of rows n and of columns m are inferred from the length of the input list. The first parameter of the skeleton is the worker function, which receives an initial datum of type `c` from the parent, a datum `[a]` from the left neighbor and a datum `[b]` from its upper neighbor. It produces results `[a]` and `[b]` for its neighbors and a final result `d` for its parent. Functions `mzip3` and `mzipWith3` are just lazier versions of functions `zip3` and `zipWith3`.

The cost model is the following:

$$
\begin{array}{lcl}
t_{torus} & = & L_{init} + t_{worker} + L_{final} \\
L_{init} & = & P(t_{create} + t_{packC}) + \delta \\
L_{final} & = & \delta + P\ t_{unpackD} \\
t_{worker} & = & t_{\#} + t_{unpackC} + t_{comp} + \\
 & & N(t_{packA} + t_{packB} + t_{unpackA} + t_{unpackB} + t_{comp}) \\
 & & + t_{packD}
\end{array}
$$

where N here is the number of rounds each worker does, i.e. the maximum length of lists [a] and [b], and t_{comp} is the cost of the worker function in each round. It is assumed $P = n \times m$, i.e. each worker has a separate processor. We are also assuming that the not accounted manager costs are not in the critical path, either because they are negligible or because a separate processor is devoted to it —in fact we assume the number of processors to be $P+1$. This topology is used in an application program in Section 4.3.2.

3.5 Iterate-Until Skeleton

This topology is appropriate for parallel algorithms in which a manager iterates until some convergence condition is met. At each iteration, a piece of work is given to each of a set of identical worker processes and a result is collected from each of them. The difference with a `farm` or a `rw` skeleton is that here the tasks sent at each iteration depend on the results of the previous one. Thus, a new iteration cannot start until the whole previous iteration has finished. A typical example of this kind of parallel algorithm is solving linear systems by the *conjugate gradient* or the *Jacobi relaxation* methods [30].

The manager is initialized with a datum of type `a` and each worker with a datum of type `b` and one of the initial tasks of type `t`. At each iteration, each worker computes a subresult of type `sr`, and a new local state of type `b`. The local state is used for its next computation, and is not transmitted to the manager, while the subresult is transmitted. The manager combines the subresults and either produces a new set of tasks and a new local state, or terminates with a result of type `r`. The Eden skeleton receives as input the following parameters:

- The initial local datum for the manager.
- A list of initial data for the workers.
- The list of initial tasks.
- The function to be used by the workers: given a local datum and a task, it generates a subresult and a new local datum for the next round.
- The function to be used by the manager in order to combine the subresults of the worker: it produces either the final result or a new list of tasks and a new local datum for the next round.

The Eden source code is the following:

```
iterUntil :: (Transmissible b, Transmissible t,
              Transmissible sr) =>
             a -> [b] -> [t] -> (b -> t -> (sr,b)) ->
             (a -> [sr] -> Either r ([t],a)) -> r
iterUntil lm ls tasks fworker comb = result   where
  outss           = map_naive (p fworker)(zip ls taskss)
  outssComb       = zipWith comb lms (transpose' outss)
  moreTaskssLms   = [ts | Right ts <- outssComb]
  result          = head [r | Left r <- outssComb]
  (moreTaskss,morelms) = unzip moreTaskssLms
  taskss               = transpose' (tasks:moreTaskss)
  lms                  = lm : morelms

-- Each worker of the skeleton
p :: (local -> task -> (subResult,local)) ->
     (local,[task]) -> [subResult]
p f (local,tasks) = subResults    where
  results                 = zipWith f locals tasks
  locals                  = local : moreLocals
  (subResults, moreLocals) = unzip results
transpose' = foldr (mzipWith' (:)) (repeat [])
```

and the cost model is:

$$
\begin{aligned}
t_{iterUntil} &= L_{init} + I\ t_{worker} + (I-1)t_{parent} + L_{final}\\
L_{init} &= P(t_{create} + t_{packI}) + \delta + t_{\#}\\
L_{final} &= P\ t_{unpackO} + t_{combine}\\
t_{parent} &= P\ t_{unpackO} + t_{combine} + P\ t_{packI} + \delta\\
t_{worker} &= t_{unpackI} + t_{compW} + t_{packO} + \delta
\end{aligned}
$$

where now the computing costs of the workers and of the parent strictly alternate in the critical path. Parameter I is the number of iterations of the algorithm.

3.6 Skeleton Composition

Much of the current research in skeletons is about how they can be composed in different ways while still achieving a good performance for the whole program. The same problem arises in Eden. From the language point of view, composition is trivial as skeletons are just functions. The real problem is how to assign processors to the skeletons in order to achieve good performance. We do not see this problem as an implementation issue but as a program design one. In our opinion, the programmer should be involved in this assignment. In [10] three different ways of composing parallel programs are described: (1) sequentially; (2) in parallel; and (3) concurrently. Applying this classification to the skeleton composition subject, in the first case, the whole machine is used by two different skeletons at different moments. This will pose no problem. In the second case, the machine is split into two or more disjoint subsets and each subset assigned to a different skeleton. The third case is similar but now the subsets need not to be disjoint.

Current Eden features do not support an easy way of splitting the processors into subsets and assigning them to skeletons. We envisage an abstract way of doing this by using a mechanism similar to the *communicator* feature of the MPI library. Every subset of processors could be assigned a unique *group identifier*. A default identifier *all* would identify the whole set of processors. Given this, the only new feature needed is the addition of an optional parameter g (the group identifier) to the instantiation operator #. This will instantiate the new process in the machine subset specified by g. This feature is currently under study.

4. RESULTS

In this section we present the results obtained for three examples using the previously described skeletons (in [25] more examples can be found). For each example, both actual and predicted relative speedups are shown. Notice that the speedups are measured with respect to the functional program running on one processor, not with respect to the most efficient implementation using an imperative program. The reason is that the comparison of the efficiency of Haskell and a language like C is out of the scope of the paper.

The experiments have been performed in a 64-processor[2] Beowulf cluster at the University of St. Andrews. Nodes are 450MHz Pentium II running Linux RedHat 5.2, with

[2]For administrative reasons, it was not possible for us to use all the processors.

348MB of DRAM and connected through a CISCO 2984G full duplex 100Mb/s fast Ethernet switch, and the latency is $\delta = 142\mu$s. So, it can be said that it is a low cost environment with relatively high latencies. Eden RTS was running on top of PVM 3.4.3. For each measurement, the average of three executions has been taken. All the processors of the Beowulf cluster are equal, and our measurements were not sharing CPU time with other programs, so that the only potential sources of load imbalances are those due to the algorithms.

4.1 Mandelbrot Sets

4.1.1 Problem Description

Mandelbrot sets can be used to generate nice fractals. For each pixel of the screen, it is necessary to compute its color (in fact, its distance to the Mandelbrot set), that depends on the coordinates of the pixel. This computation is performed by iterating a function until a good approximation is obtained, or until a maximum number of iterations (4000 in our implementation) has been reached. Hence, for some pixels it is enough to compute a few iterations, while for others 4000 iterations are needed. The sequential algorithm is the following:

```
mandel sizeX sizeY
  = map aPixelOfMandel (range ((0,0),(sizeX,sizeY)))
aPixelOfMandel (x,y) = iterate (firstAprox (x,y)) 0
iterate v its
 | its == maxIts || goodAprox v = extractColor v
 | otherwise = iterate (nextAprox v) (its+1)
```

4.1.2 Parallel Algorithm

As the computation of each pixel can be done in parallel, a parallel map scheme can be used. In order to increase the granularity of the tasks, each line of the screen will be assigned to a task.

This is a highly-parallel algorithm: the subcomputations are completely independent, and the computation vs communication ratio is quite good. Unfortunately, the time complexity of the tasks is rather varying, so that naïve parallelizations can lead to a poor load balance. If we just try to split the screen into p (p being the number of processors) regions and we use a process for each of the regions, we cannot guarantee that the load balancing will be even. Moreover, it is known that time-consuming pixels are usually clustered. Thus, it is not recommended to assign 'neighbor' regions to the same processor. The farm scheme can be used, but care has to be taken in order to decide which `shuffle` and `unshuffle` functions to use, as the speedups can drastically differ. Moreover, the speedup will depend on the Mandelbrot set to be computed, as different sets have different distributions of complex regions. Thus, the cleverest solution is to use `map_rw`, as it balances the load on its own:

```
mandel sizeX sizeY
  = map_rw 2 (oneLineOfMandel sizeY) [1..sizeX]
oneLineOfMandel sizeY x
  = map aPixelOfMandel (range ((x,0),(x,sizeY)))
```

4.1.3 Performance Results

We have generated Mandelbrot sets for a 1024×1024 screen, the sequential runtime being 459 seconds. Figure 4 shows the results obtained using both the `map_rw` and the `map_farm` skeletons. For the `map_farm`, two different strategies for unshuffling the tasks amongst processes have been used. The first one leads to a quite unbalanced load distribution, giving a speed-up of only 11.5 with 32 processors, while the other one behaves nearly the same as the `map_rw`.

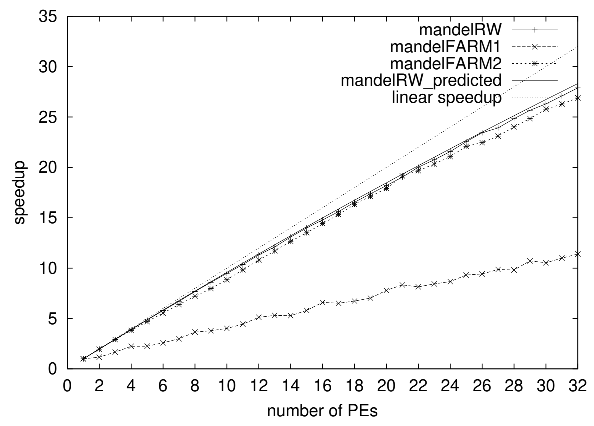

Figure 4: Speedups for Mandelbrot sets

The measurements confirm that skeleton `map_rw` is the best suited for this problem and, in general, for all the problems with varying granularities. Even though the overheads of `map_rw` are slightly greater than the overheads of `map_farm`, the absolute speedup obtained is better, because it provides a much better load-balance. Notice that `map_farm` can approximate `map_rw` in case appropriate `unshuffle` and `shuffle` functions are chosen, but this implies more work for the programmer, and the right functions to be used can be different for different input problems for the same algorithm. Moreover, in case the parallel machine is not solely devoted to our program, the `map_farm` versions cannot rebalance the load, while `map_rw` can do it.

Notice also that the predictions for `map_rw` are accurate, and that the scalability is quite good, the main inefficiency being a sequential bottleneck of two seconds for combining the results of the workers. In the `map_farm` case, no prediction is given, as the cost model assumes the granularity of the tasks to be regular, and it is not true in the case of the Mandelbrot sets.

4.2 Karatsuba Algorithm

4.2.1 Problem Description

The Karatsuba algorithm [16] computes the product of two large integers using a divide and conquer approach. Given n as the length of the integers, the complexity of the naïve strategy for multiplying them is in $O(n^2)$, while the Karatsuba algorithm computes it in $O(n^{log_2 3})$.

If two large integers x and y represented in base b are to be multiplied, the algorithm works as follows:

- Let n be half of the length of the longest of x and y (using the corresponding base representation).
- 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 $u * b^{2*n} + (w - u - v) * b^n + v$.

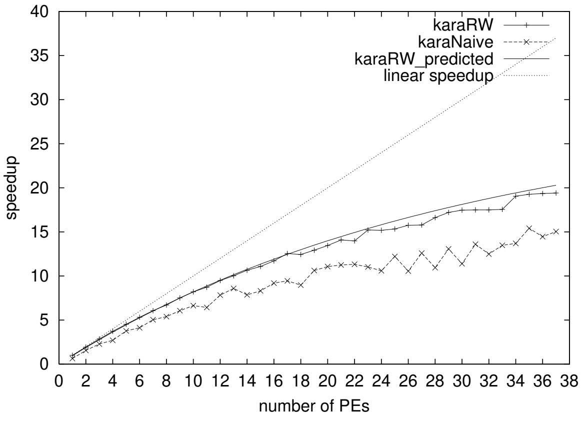

Figure 5: Speedup for Eden Karatsuba

Notice that to obtain x_1, x_2, y_1 and y_2 it is not necessary to perform any division, it is enough to cut the lists representing x and y. Analogously, the multiplication with b^n and b^{2*n} do not need any multiplication, but only adding zeros to the corresponding long integer. Therefore, only three multiplications are needed, that is, only three subproblems are generated when splitting a problem. As three subproblems of half the complexity are generated, and combining the subresults is in $O(n)$, this leads to a complexity in $O(n^{log_2 3})$.

This algorithm fits a divide and conquer scheme, where the granularity of the subtasks can be varying, as the three multiplications are possibly applied to integers of different lengths.

4.2.2 Parallel Algorithm

The implementation of the Karatsuba algorithm in terms of divide and conquer is as follows, where the implementation of the non-shown functions follows the above explanation.

```
type MyInteger = [Int]
karat :: Int -> MyInteger -> MyInteger -> MyInteger
karat depth is1 is2
  = dc_rw depth trivial solve split combine (is1,is2)
```

4.2.3 Performance Results

Both the `dc_rw` and the `dc_naive` version of the divide and conquer skeleton have been tested for the same input data, whose sequential execution time is 440 seconds. The actual and predicted speedups can be seen in Figure 5. 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. Notice that the prediction of the `dc_rw` behavior is quite accurate, while no prediction is given for the naïve version owing to lack of an accurate cost model.

4.3 Matrix Multiplication

4.3.1 Problem Description

The product of an $m \times n$ matrix (M1) and an $n \times p$ matrix (M2), yields an $m \times p$ matrix (M), where M(i,j) is the dot product of the i-th row of M1 and the j-th column of M2:

```
prMMseq m1 m2 = prMMTr m1 (transpose m2)
```

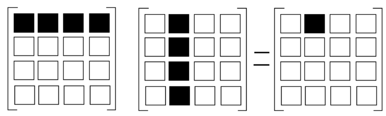

Figure 6: Matrices multiplication using blocks

```
prMMTr m1 m2  = [[prRC row col| col <- m2 ] | row <- m1]
prRC row col  = sum (zipWith (*) row col)
```

4.3.2 Parallel Algorithm

Each element of the resulting matrix can be computed in parallel. If the size of the matrices is $n \times n$, and p processors are available, p tasks should be created, each one evaluating n/p rows of the resulting matrix. Then, the communications of the main process are in $\mathrm{O}(n^2 * p)$, but the computation of each process will be in $\mathrm{O}(n^3/p)$. So, better speedups are expected if $n \gg p$. As the granularity of the tasks is very regular, the corresponding Eden program uses the simple `map_naive` skeleton:

```
prMM :: Matrix -> Matrix -> Matrix
prMM m1 m2 = concat out    where
 out = map_naive (uncurry prMMTr)
      (zip (splitIntoN noPe m1)(repeat (transpose m2)))
```

where `noPe` represents the number of available processors, and `splitIntoN n xs` splits `xs` into `n` nearly equal size sublists.

To decrease the communications, the Gentleman algorithm [12] can be used, where each process computes a rectangular block of the final matrix, as depicted in Figure 6. By doing so, the computation of each process is also in $\mathrm{O}(n^3/p)$, but its communication is only in $\mathrm{O}(n^2/\sqrt{p})$, as they do not require the whole second matrix to be transmitted. The implementation of the above idea can be done using a parallel map approach. However, this implies that the main process has to perform a lot of communications ($\mathrm{O}(n^2 * \sqrt{p})$) at the beginning of the computation, and it can become a bottleneck. A better solution consists of using a torus topology, so that initially each process receives only a portion of the inputs, and it obtains the rest of them from its neighbors: The submatrices of the first matrix are transmitted from left to right in the torus, while those of the second matrix are transmitted from top to bottom. This reduces the communications of the main process to $\mathrm{O}(n^2)$. Notice that there exist an important difference between $\mathrm{O}(n^2 * \sqrt{p})$ and $\mathrm{O}(n^2)$: In the second case the communications do not depend on the number of processors, improving the scalability. The only drawback of the torus approach is that a perfect square number of processes is needed.

The algorithm needs only to define the size of the torus — i.e. $\lfloor \sqrt{P} \rfloor$— to split the matrices into blocks, and to define the function to be applied. The node function just constructs a list of block multiplications —one for each pair of blocks it receives— and then adds up all the products. The number of matrix multiplications performed by each process will be the size of the torus.

```
mult :: Matrix -> Matrix -> Int ->  Matrix
mult m1 m2 torusSize = combine resu   where
 resu = torus (mult' torusSize) (zipWith zip m1ss m2ss)
 m1ss = splitMatrix1 torusSize m1
```

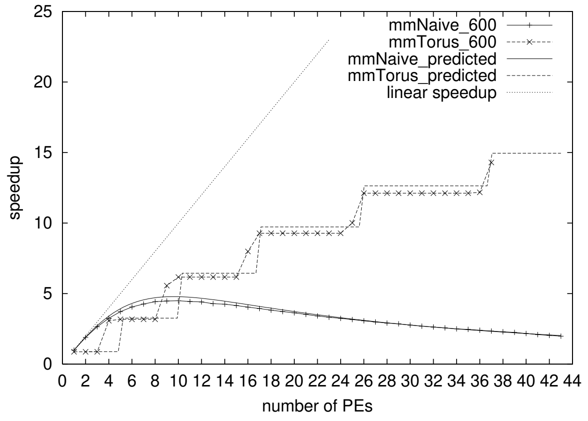

Figure 7: Speedups for matrix multiplication

```
 m2ss = splitMatrix2 torusSize m2

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

where `splitMatrix1` and `splitMatrix2` are simple Haskell functions that split the matrices into blocks (see Figure 6) and shift them in order to appropriately feed the torus.

4.3.3 Performance Results

Figure 7 shows the speedup profiles for the two Eden versions of matrix multiplication using squaring matrices of size 600×600, the sequential execution time being 221 seconds. It can be seen that the naïve version does not scale-up satisfactorily, while the algorithm using the torus scales up much better, as the computation vs communication ratio is also better. The predicted speedup of the naïve version is quite close to the actual speedup obtained. In this case, the dominant parameter of the cost model is t_{packI}, as 2.3 seconds are needed to pack the whole second matrix. This parameter is multiplied by P in L_{init}, so that, as the number of processors increases, the comunication overhead also increases.

For the torus version, the prediction is also quite accurate in all points but the perfect squares. The reason is that the cost model assumes that the main process does not share a processor with a worker, but in our measurements this was not the case for perfect squares. The cost model could easily be adjusted to take this fact into account.

The main reason why the torus scales up better than the naïve version can be seen in the cost model: Now, L_{init} does not depend heavily on the number of processors, because t_{packC} is proportional to $1/P$: as P increases, the block size is smaller, and the total communication cost incurred at the beginning of the computation is the same.

5. CONCLUSION

We have shown that Eden allows a concise expression of different implementation strategies for well-known skeletons such as parallel map, parallel divide and conquer, and pipelines, and that some other topologies such as iterate-until or torus can be easily defined. As a proof of this conciseness, it can be seen in the paper and in [25] that most of the skeletons are implemented in less than half a page of Eden code. Even though process placement annotations are not possible in the language, good load-balance strategies can be defined thanks to the existence of the reactive process `merge`. As a whole, this confirms the usefulness of Eden as a system skeleton-implementation language.

Even easier is to use already-defined skeletons in applications. In this sense, Eden provides the same high-level style of parallel programming as skeleton-based languages. Moreover, as no barriers exist in the language, programmers can freely move from one level of abstraction to another as required by the application, i.e. a part of the algorithm may be skeleton-based while others manipulate processes explicitly. Even, some new application-specific skeletons may be defined and used in the program.

Another claim made at the introduction was that the difference between the sequential and the parallel versions of a functional algorithm are usually small. This can be confirmed in most of the examples presented here (parallel mergesort, Mandelbrot sets, Karatsuba algorithm, naïve matrix multiplication) and in many other not shown (see [25]). Only Gentleman's algorithm for efficient matrix multiplication looks very different from the original sequential one.

Although Eden has only been recently implemented, we have tested our skeletons in a rather large set of examples. Two of the three presented here (Karatsuba and matrix multiplication) are particularly challenging for an abstract language such as Eden and for a high-latency architecture such as a Beowulf. A lot of (costly) communication is performed, and the computation vs communication ratio is not very big. An imperative programmer normally has the opportunity of doing many low-level optimizations (for instance to control the size of the packets, to overlap computation and communication, to efficiently use the cache, and so on) that Eden programmers cannot do. All these issues are under the control of the runtime system which is a general-purpose (as opposed to application-specific) program.

In spite of these limitations, the speedups achieved are quite reasonable, sustaining our claim of *acceptable speedups with low effort*. Also, it is encouraging to notice that our cost models —based on specific knowledge about how our runtime system works, and instantiated for the concrete architecture and problem at hand— accurately predict the actual costs obtained.

6. ACKNOWLEDGMENTS

The authors thank Rita Loogen, Clara Segura, Yolanda Ortega and the anonymous referees for valuable comments on a previous version of the paper. This work was partially supported by the Spanish project TIC2000-0738, and the Spanish-British Acción Integrada HB 1999-0102.

7. REFERENCES

[1] G. H. Botorog and H. Kuchen. Efficient Parallel Programming with Algorithmic Skeletons. In L. Bougue, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proceedings of EuroPar'96*, LNCS 1123, pages 718–731, 1996.

[2] T. A. Bratvold. A Skeleton-Based Parallelising Compiler for ML. In *Implementation of Functional Languages, IFL'93, Nijmegen, The Netherlands*, pages 23–33, September 1993.

[3] S. Breitinger, U. Klusik, and R. Loogen. From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View. In PLILP'98. LNCS 1490, pages 318–334, 1998.

[4] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Peña. DREAM: the Distributed Eden Abstract Machine. In *Implementation of Functional Languages, IFL'97. St. Andrews, Scotland. Selected Papers*, pages 250–269. LNCS 1467. Springer-Verlag, 1998.

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

[6] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language Definition and Operational Semantics. Technical Report, Bericht 96-10. Revised version 1.998, Philipps-Universität Marburg, Germany, 1998.

[7] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research monographs in parallel and distributed computing. Pitman, 1989.

[8] J. Darlington, A. J. Field, P. J. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel Programming using Skeleton Functions. In *PARLE'93: Parallel Architectures and Languages Europe*. Springer-Verlag, Jun. 1993.

[9] J. Darlington, Y. Guo, H. To, and J. Yang. Parallel Skeletons for Structured Composition. In *ACM SIGPLAN Symposium on Principles and Practice of Programming*, pages 19–28. ACM Press, 1995.

[10] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[11] A. Geist, A. Beguelin, J. Dongarra, and W. Jiang. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[12] W. M. Gentleman. Some Complexity Results for Matrix Computations on Parallel Computers. *Journal of the ACM*, 25(1):112–115, Jan 1978.

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

[14] K. Hammond and A. J. Rebón. HasSkel: Algorithmic Skeletons for Haskell. In *Implementation of Functional Languages, IFL'99, Lochem, Sept. 1999. Selected Papers*. LNCS, 2000.

[15] C. Herrmann and C. Lengauer. The HDC Compiler Project. In A. Darte, G.-A. Silber, and Y. Robert, editors, *Proceedings Eighth Int. Workshop on Compilers for Parallel Computers (CPC 2000)*, pages 239–254, 2000.

[16] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Pitman, 1978.

[17] U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Implementation Skeletons in Eden: Low-Effort Parallel Programming. In *Selected papers of Implementation of Functional Languages, IFL2000*, pages 71–88. LNCS 2011, 2001.

[18] U. Klusik, Y. Ortega-Mallén, and R. Peña. Implementing Eden - or: Dreams Become Reality. In *Implementation of Functional Languages, IFL'98, London, Sept. 1998. Selected Papers*, pages 103–119. LNCS 1595, 1999.

[19] U. Klusik, R. Peña, and F. Rubio. Replicated Workers in Eden. In *Constructive Methods for Parallel Programming (CMPP'2000). To appear*. Nova Science, 2001.

[20] U. Klusik, R. Peña, and C. Segura. Bypassing of Channels in Eden. In *Trends in Functional Programming*, pages 2–10. Intellect, 2000.

[21] Message Passing Interface Forum. MPI: A Message-passing Interface Standard. International Journal of Supercomputer Applications, 8(3/4), 1994.

[22] G. Michaelson, A. Ireland, and P. King. Towards a Skeleton Based Parallelising Compiler for SML. In C. Clack, T. Davie, and K. Hammond, editors, *Implementation of Functional Languages, IFL'97, St. Andrews, Scotland*, pages 539–546, 1997.

[23] S. Pelagatti. *A Methodology for the Devolopment and the Support of Massively Parallel Programs*. PhD thesis, Dipartamento di Informatica, University of Pisa, 1993.

[24] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor and Francis, 1998.

[25] R. Peña and F. Rubio. Eden Skeletons. Technical Report. Dep. Sistemas Informáticos y Programación, Univ. Complutense Madrid. Available at http://dalila.sip.ucm.es/funcional/othpapers.html, 2001.

[26] R. Peña and C. Segura. Non-Determinism Analysis in a Parallel-Functional Language. In *Selected papers of Implementation of Functional Languages, IFL2000.*, pages 1–18. LNCS 2011. Springer, 2001.

[27] S. Peyton Jones. Compiling Haskell by Program Transformations: A Report from the Trenches. In *ESOP'96, LNCS 1058*, 1996.

[28] S. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Inf. Technology, Keele*, pages 249–257, 1993.

[29] S. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98*. URL http://www.haskell.org, February 1999.

[30] M. J. Quinn. *Parallel Computing*. McGraw-Hill, 1994.

[31] F. Rabhi. *A Parallel Programming Methodology Based on Paradigms*, pages 239–252. IOS Press, 1995.

[32] J. H. Reppy. CML: A Higher-order Concurrent Language. In *ACM SIGPLAN Conference on Prog. Lang. Design and Implementation*, pages 293–305. ACM Press, 1991.

[33] P. W. Trinder, K. Hammond, J. S. M. Jr., and A. S. Partridge. GUM: a Portable Parallel Implementation of Haskell. In *ACM SIGPLAN PLDI, Philadelphia, USA*. ACM Press, May 1996.

[34] P. W. Trinder, K. Hammond, H. W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), 1998.