# Chapter 4

# Parallelism Abstractions in Eden

**Rita Loogen, Yolanda Ortega, Ricardo Peña,
Steffen Priebe, and Fernando Rubio**[1]

The parallel functional programming language Eden extends Haskell with expressions to define and instantiate process systems. These extensions allow also the easy implementation of skeletons as higher-order functions. Parallel programming is possible in Eden at two levels: The first level is the most abstract one and it is appropriate for building parallel applications at low effort on top of the predefined skeletons. At the lower level, the programmer instantiates processes explicitly, being able to create new skeletons, and also to build applications with irregular parallelism for which there is not an appropriate skeleton to apply. In this chapter, we present several skeletons covering a wide range of parallel structures. For each skeleton, one or more implementations in Eden are given, together with their corresponding cost models. We also show examples of application programming, including predicted and actual results on a Beowulf.

## 4.1 Introduction

Two important abstractions have contributed to create a reliable programming methodology for industrial-strength programs. These are *functional abstraction* (which has received different names in programming languages, such as procedure, subroutine, function, etc), and *data abstraction* (also with different names such as abstract data type, object, package or simply module). In both abstractions two different pieces of information are distinguished:

[1]Work partially supported by the spanish project TIC2000-0738, Spanish-British Acción Integrada HB 1999-0102 and a German-British ARC cooperation funded by the German Academic Exchange Service (DAAD).

- The *specification* defines its external behaviour. It contains all the information needed by a potential user of the abstraction.

- The *implementation* determines its efficiency. In general, there can be several implementations for the same specification.

Several *algorithmic schemes* have been identified to solve different problem families in sequential programming. For instance, there exist the greedy method, the dynamic programming method or the divide and conquer method. Analogously, *parallel* algorithms can be classified into families, so that all members of a family are solved by using the same scheme. The abstraction of this scheme is what we call an *algorithmic skeleton*, or simply a *skeleton* [Col89].

Its *specification* describes at least the values returned by the skeleton for each possible input, i.e. its functional behaviour. But usually it also describes the family of problems to which the skeleton is applicable. For instance, there exists a parallel *divide and conquer* skeleton useful for problems for which a function `split` (to divide a problem into subproblems), and a function `combine` (to combine the sub-results) exist. In fact the skeleton solves the same problem family as the sequential *divide and conquer* scheme. We take the position that, as part of the specification, a sequential algorithm solving the family of problems should be provided. Frequently, this sequential algorithm is actually used by the implementations in some of the parallel processes.

Normally, a skeleton can be implemented in several different ways. *Implementations* may differ in the process topology created, in the granularity of the tasks, in the load balancing strategy or in the target architecture used to run the program. So, the implementation hides many details to the potential user, and also determines the efficiency of the program.

One of the main characteristics of skeletons is that it should be possible to predict the efficiency of each implementation. This can be done by providing a *cost model* together with each implementation. A cost model is just a formula stating the predicted parallel time of the algorithm [Ham00]. To build this formula, the implementor has to consider all the activities which take place in the *critical path* of the algorithm. This includes the initial sequential actions needed to put at work all the processors of the parallel machine, the maximum of the individual times needed by the processors, and the final sequential actions, which take place between finishing the last subtask and delivering the final result. Cost models will be parameterized by some constants that may depend either on the problem to be solved, on the underlying parallel architecture, or on the runtime system (RTS) being used.

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

Eden [BLOP96, BLOMP97] is one of the few functional languages in which skeletons can be both *used* and *implemented*. In other approaches, 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.

In Section 4.2 we introduce the features of Eden that are used in the skeleton definitions in Section 4.3. Section 4.4 presents several example applications which are parallelized using the skeletons. Runtime results show that the skeleton-based parallelization leads to reasonable speedups on a Beowulf cluster. Moreover, the actual runtime results correspond to the ones predicted by the cost models of the skeletons. The chapter finishes with a discussion of related work and conclusions.

## 4.2 Eden's Main Features

Eden [BLOP96, BLOMP97] extends the lazy functional language Haskell [PH99] by syntactic constructs for *explicitly* defining processes. Eden's process model provides direct control over process granularity, data distribution and communication topology.

### 4.2.1 Basic Constructs

A *process abstraction* expression `process x -> e` of type `Process a b` defines the behaviour of a process having the formal parameter `x::a` as input and the 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.

A *process instantiation* uses the predefined infix operator

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

to provide a process abstraction with actual input parameters. The context `Transmissible a` ensures that functions for the transmission of values of type `a` are available.

The evaluation of an expression `(process x -> e1) # e2` leads to the dynamic creation of a process together with its interconnecting communication channels. The instantiating or *parent process* will be responsible for evaluating and sending `e2` via an implicitly generated channel, while the new *child process* will evaluate the application `(\ x -> e1) e2` and return the result via another implicitly generated channel. The instantiation protocol deserves some attention: (1) Expression `e1` together with its whole environment is *copied*, in the current evaluation state, to a new processor, and the child process is created there to evaluate the expression `(\ x -> e1) e2` where `e2` must be remotely

received. (2) Expression `e2` is eagerly evaluated in the parent process. The resulting full normal form data is communicated to the child process as its input argument. (3) The normal form of the value `(\ x -> e1) e2` is sent back to the parent. For input or output tuples, independent concurrent threads are created to evaluate each component.

Processes communicate via *unidirectional channels* which connect one writer to exactly one reader. Once a process is running, only fully evaluated data objects are communicated. The only exceptions are lists, which 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 input which is not available yet, are temporarily suspended. This is the only way in which Eden processes synchronize.

*Example 4.1* 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` skeleton, in which a different process is created for each element of the input list:

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

The process abstraction `pf` wraps the function application `(f x)`. It determines that the input parameter `x` as well as the result value will be transmitted on channels. Therefore both types `a` and `b` must belong to the class `Transmissible`.

The `spine` strategy is used to eagerly evaluate the spine of the process instantiation list. In this way all processes are immediately created. Strategies [THLP98] are functions which control the evaluation of expressions without producing a result value. They are applied by means of the function `using`, that first applies the strategy to the input, and then returns the value of the input:

```
using x s = s x `seq` x

spine        :: [a] -> ()
spine []     = ()
spine (_:xs) = spine xs
```

`map_par` is an essential primitive skeleton used to eagerly create a set of independent processes. More sophisticated parallel implementations of `map`, built on top of it, will be presented in the following section (see also [KLPR01, PR01]). ◁

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. Thus, in general, the number of processes cannot be determined at compile time.

Eden is based on Haskell, a non-strict functional language. Non-strictness, implemented by using lazy evaluation of expressions, is a key point in our approach. For instance, in Eden it is possible to create circular topologies of processes connected by lists. In an eager language this will simply lead to a deadlock, as a process cannot completely evaluate its output because it will probably need its whole input, which is still being produced by another process, which in turn will demand its whole input, and so on. Another interesting feature of non-strictness in our Eden programs is that they can be automatically converted to a working sequential program in Haskell just by replacing processes by functions (a small syntactic change as seen above). From Haskell's point of view, our process topologies will then simply look like a set of mutually recursive functions.

Lazy evaluation is changed to eager evaluation in two cases: processes are eagerly instantiated, 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. The concept of a virtually shared global graph does not exist. Each process evaluates its outputs autonomously.

### 4.2.2 Many-to-one Communication

Many-to-one communication is an essential feature for some parallel applications, but it spoils the purity of functional languages, as it introduces nondeterminism. In Eden, the predefined process abstraction

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

is used to instantiate processes which *fairly* merge lists of input streams into single (non-deterministic) output streams. The incoming values are passed to the output stream in the order in which they arrive. In this way `merge` provides *many-to-one communication*. It can profitably be used to react quickly to requests coming in an unpredictable order from a set of processes.

Even though the skeletons presented are deterministic, some of them are required to immediately react to requests for work coming from a group of worker 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. A non-determinism analysis [PS01] detects the expressions which are sure to be deterministic even in presence of `merge` instantiations.

### 4.2.3 Dynamic channels

An Eden process may generate a new *dynamic input channel* and send a message containing the channel's name to another process. The receiving process may

then either use the 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, and a runtime error occurs if not appropriately used.

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

```
new (ch_name, chan) e
```

This declares a new channel name `ch_name` as reference to the new input channel `chan`, which represents future input. 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 channel name `ch_name`, and wanting to reply through it, uses an expression `ch_name !* e1 par e2` . 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`, 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 chapter— it is possible to create the same topologies without using dynamic channels, the main difference being that some channels will connect the intended processes through intermediate threads in other processes. By using dynamic channels those will be *direct* connections. In this sense, in this paper this feature can be seen as an optimization using a low-level construct provided by the language rather than as a radically new concept.

### 4.2.4 Eden Implementation

Eden's compiler[2] has been developed by extending the Glasgow Haskell Compiler (GHC) [PHH+93, Pey96], in order to reuse its efficiency and portability. Eden's runtime system (RTS) is an implementation of the DREAM abstract machine [BKL+98b] on top of a message passing library. In the current compiler, both PVM [GBDJ94] and MPI [MPI94] can be used. Therefore, the compiler can be ported to any architecture where GHC and either PVM or MPI are available.

Eden compiler has been developed by modifying two parts of GHC. Firstly, the front-end has been extended to deal with the new Eden constructions. The modification is done in such a way that the extra constructions are hidden after the parser. The idea is that the constructions are translated into applications of predefined Eden functions, that will be used to connect GHC with Eden RTS. By doing so, it is not necessary to modify the compilation process of GHC, as its internal constructions are still the same. Eden extensions only appear again at runtime, when its predefined functions are invoked. Eden RTS extends GHC RTS in order to implement the DREAM abstract machine, and this is

[2]Freely available at http://www.mathematik.uni-marburg.de/inf/eden

done by modifying GUM, the implementation of GpH [THJ+96]. See [BKL98a, KOMP99] for more details about Eden implementation.

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 round-robin mode allows the programmer to control somehow the mapping of processes, as he/she can achieve that different processes will be placed on different processors.

The number of processors is provided by the integer constant `noPe`. It can be used to adapt the number of processes to the number of available processors.

## 4.3 Skeletons in Eden

Skeletons and alternative parallel implementations of them can easily be defined in a higher-order functional language with explicit parallelism like Eden. 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 many typical data-parallel, task-parallel and systolic skeletons in Eden, and discuss alternative parallel implementations of these skeletons together with appropriate cost models.

### 4.3.1 Cost Models

The cost models presented in this section are an adaptation of classical cost models appearing in the literature (see e.g. [Ham00]). In Eden, the parallel computation starts and finishes always in the main process, from which other processes may be instantiated. In some cases, all the child processes are created from the same parent process. In other cases, a child process creates another child process, which in turn creates others, and so on. The cost models below take into account the creation and termination of processes in the critical path, i.e. the processes created from the beginning of the main process until all processors are computing in parallel, and the activities from the end of the last child until the main process terminates.

We will use the parameters shown in Figure 4.1 and combine some of them into higher-level parameters. The local costs, i.e. CPU time, to pack or to unpack a message with $nw$ words for sending or receiving, respectively, is given by

$$time(nw) = \lambda + \beta * nw.$$

| **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 set up the child process |
| **Architecture dependent parameters** | |
| $P$ | number of processors |
| $\delta$ | latency of a message, from start sending to start receiving |
| $\lambda$ | start-up fixed CPU cost for sending or receiving a message |
| $\beta$ | per-word CPU cost for sending or receiving a message |

Figure 4.1: Parameters of the cost models

As we do not distinguish between packing and unpacking messages, the CPU time for packing or unpacking a message going to a child process is

$$t_{unpackI} = t_{packI} = time(nwI)$$

In the same way, the time for packing or unpacking an output coming from a child is

$$t_{unpackO} = t_{packO} = time(nwO)$$

*Example 4.2* In the `map_par` skeleton defined in Section 4.2 a process is created for each element in the list. If the number of tasks is greater than the number of processors, several tasks will be evaluated in each processor. For the cost model, we assume a round-robin distribution of processes onto processors and a uniform granularity of tasks:

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

The first formula describes the critical path determining the parallel time. This path consists of a startup phase taking time $L_{init}$, an intermediate phase with time $t_{processor}$ where all processes work in parallel, and a final shutdown phase with time $L_{final}$. Before the last worker starts computing, $P$ processes must be created and $P$ messages must be packed in the parent. The remaining $N - P$ process creations and message packing are interleaved in the same processor as the last worker, i.e. they are in the critical path, so we have attributed these costs to $L_{init}$. After the last worker finishes, an output message must first arrive to the parent (hence the $\delta$ latency) and then be unpacked. The most heavily loaded processor will get $\lceil \frac{N}{P} \rceil$ tasks. We are assuming that the remaining manager costs (i.e. the reception of the $N - 1$ remaining messages) are outside the critical path. ◁

### 4.3.2 Data Parallel Skeletons

Data-parallel skeletons define global operations over large data structures, where the individual operations on single elements or substructures of the data structure are performed in parallel. The simplest data-parallel skeleton is `map` which applies the same function `f` on different elements of a distributed data structure, in our case a list. Another skeleton we will consider, is `map and reduce`, a combination of a `map` and a `fold`.

In a data-parallel language, the sequential code of each process is assumed to be stored at every processor, and the global data already distributed according to the programmer's declaration for the data structure. So, the programmer is not concerned with process creation and/or communication. These take place implicitly when needed by the algorithm. In Eden, the computation starts in a single process and the programmer is responsible for instantiating processes and for specifying the distribution of data between them. This is so because Eden is not a data-parallel, but a task-parallel language. Therefore, the following skeletons can be seen as an approximation of how to express data parallelism in a task-parallel language.

#### Map

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 Section 4.2 we have already shown 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 tasks subset.

**Farm Implementation.** 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 of workers `np`, and the distribution and collection functions `unshuffle` and `shuffle` are parameters of `farm`. The `map_par` skeleton is used to create as many processes as the number of task lists into which the original list is distributed.

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

farm :: (Transmissible a, Transmissible 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))
```

`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` holds.

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 order to place the processes on different processors, the round-robin mode of the RTS should be used. Moreover, to improve the load balance, the length of the task list should be much higher than the number of available processors, so that it is not relevant the fact that one processor may receive one task in excess of those of other processors. Alternatively, the number of tasks should be a multiple of the number of processors.

In the cost model for `map_farm` the costs of shuffling and unshuffling are added and only one process is created per processor:

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

$t_{unshuffle_1}$ is the time needed to distribute one element of the task list. It is multiplied by `P`, because a task must be delivered for each worker. We are assuming that the CPU time not shown in the parent (i.e. unshuffling and shuffling the rest of the tasks) is small and does not affect the critical path. If this were not the case, we would assign a separate processor to the main process and these times will then be outside the critical path. The price to be paid is devoting a single processor to the parent. For that purpose, we introduce the following variants of the `map_farm` skeleton:

```
map_farm_1 :: (Transmissible a,Transmissible b) =>
              (a->b) -> [a] -> [b]
map_farm_1  = farm (noPe-1) shuffle unshuffle

map_farm_thr :: (Transmissible a,Transmissible b) =>
                Int -> (a->b) -> [a] -> [b]
map_farm_thr thr = if noPe > thr then map_farm_1 else map_farm
```

`map_farm_1` devotes a separate processor to the main process, while `map_farm_thr` behaves like `map_farm` or `map_farm_1` depending on a threshold parameter. These variants can be defined in a similar way for all the skeletons. In some of the algorithms presented in Section 4.4 the threshold variant of the skeletons has been used.

**Self-service Farm Implementation.** Sometimes duplicating work helps reducing the total execution time, as communications can be reduced a lot. In the `map` case, when the evaluation of the task list is cheaper than communicating the evaluated list, it is better to allow the workers to evaluate the list of tasks on their own and to select their part of it. This can be done by providing the workers with parameters instead of input channels:

```
ssf :: Transmissible b =>
       Int -> (Int->[a]->[[a]]) -> ([[b]]->[b]) -> (a->b) -> [a] -> [b]
ssf np shuffle unshuffle f tasks
  = shuffle [(worker f ts) # () | ts <- unshuffle np tasks]
    where worker f tasks = process () -> map f tasks
```

The difference between the cost model of `map_farm` and the one of `map_ssf` is that now $t_{packI}$ and $t_{unpackI}$ disappear, and that the cost for unshuffling the tasks is attributed to the workers:

$$\begin{aligned} t_{map\_ssf} &= L_{init} + t_{worker} + L_{final} \\ L_{init} &= Pt_{create} + \delta \\ L_{final} &= \delta + t_{unpackO} + t_{shuffle_1} \\ t_{worker} &= t_{unshuffle_{\lceil \frac{N}{P} \rceil}} + t_{\#} + \lceil \frac{N}{P} \rceil (t_f + t_{packO}) \end{aligned}$$

**Replicated Workers Implementation.** The load balance obtained using the farm or self-service schemes can be poor in three cases: (1) When the granularity of the tasks is not uniform; (2) when the processors' architecture is irregular; and (3) when the programs share CPU time with other programs in the same processors.

In all these three situations distributing work on demand helps to improve substantially the load balance. A new task is assigned to a process only when it has finished its previous task. This idea gives rise to the *replicated workers* skeleton [KPR01]. 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 task, it sends an acknowledgment message to the manager including the result, and then a new task (if any is available) is assigned to that process. The computation finishes when the manager has received all the results.

The programmer cannot predict in advance the order in which processes are going to finish their works, as this depends on runtime issues. By using the process `merge`, acknowledgments from different processes can be received by the manager in the order in which they arrive. Thus, if each acknowledgment contains the identity of the sender process, the list of merged results can be scrutinized to know who has sent the first message, and a new task can be assigned to it. Notice that this approach can not be used in a purely functional language, as process `merge` is not functional (see Section 4.2.2).

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)

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

Notice that the output of the list of workers (`outsChildren`) is used in two different ways: (i) `merge` is applied to it in order to obtain a list `unordResult` containing the order in which the results are generated, so that it can be used by `distribute` to distribute a new task to each processor as soon as it finishes its previous tasks; and (ii) it is used to obtain the final result by applying `sortMerge` to it, 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 fact, ignoring the first two parameters, its semantics is that of `map`. In order to implement `map`, a worker is created for every processor, and a prefetch parameter of 2 is used, as this value uses to be the best one in general. The reason is that, with a smaller value communications and computations cannot overlap, and with bigger values the load balance could be worse, as there are more tasks not distributed on demand.

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

The cost model for `map_rw` is the following:

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

The considerations made for the `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 with an average computing cost $t_{comp}$.

**Fixed Shared Data Structures.** When there exists a fixed data structure that has to be shared by all the tasks, it does not make sense to send such a structure each time a new task is released. Instead, it should be sent only once to each process, and all the tasks of the same process should share it. This cannot be done with the implementations presented so far, but the solution is quite simple: the new implementations need an extra parameter (the shared data) that is sent to the workers through an independent channel. In the case of the replicated workers the implementation only requires the following modification:

```
rw_FD :: (Transmissible a,Transmissible b, Transmissible fixed) =>
      Int -> Int -> fixed -> (fixed -> a -> b) -> [a] -> [b]
rw_FD np prefetch fixed f tasks =  results     where
  outsChildren = [(worker_FD f i) # (fixed,inputs) |
                   (i,inputs) <- zip [0..np-1] inputss]
  ...
worker_FD :: (Transmissible a, Transmissible b, Transmissible fixed) =>
          (fixed -> a -> b) -> Int -> Process (fixed,[(Int,a)]) [ACK b]
worker_FD f i = process (fixed,ts) -> map f' ts
  where f' (id_t,t) = ACK i id_t (f fixed t)
```

and these modifications are analogous for `farm`. The only difference with `rw` is that now it is necessary to have an extra parameter for the fixed structure, and it has to be used appropriately. The difference in the cost models is that the workers have now an extra cost unpacking the shared data, while the cost of packing it $P$ times has to be added to $L_{init}$. The advantage is that now the cost associated to $t_{packI}$ and $t_{unpackI}$ will be smaller, as the tasks are smaller because the full fixed data structure is not sent with each task.

### Map and Reduce

The sequential specification of this classical scheme is a combination of a map and a fold function:

```
mr :: (a -> b) -> (b -> b -> b) -> b -> [a] -> b
mr f g e tasks = foldl g e (map f tasks)
```

where the first parameter is the function `f` to be applied by the map, while the second is a binary commutative and associative function `g` with a *neutral* element `e`.

**Farm Implementation.** In a straightforward approach this scheme could be parallelized by first applying in parallel the map step, and then folding the results, thereby using the strict variant `foldl'` of `fold`. More parallelism and less communication can be achieved, because the folding parameter `g` is an associative and commutative function with neutral element `e`. The results computed in each processor can be folded together locally before the global folding is done, i.e. the folding step is also parallelized, and the communications are reduced, as only one element is returned by each worker, instead of a sublist.

```
mr_PM :: (Transmissible a, Transmissible b) =>
        Int -> (Int -> [a] -> [[a]]) ->
        (a -> b) -> (b -> b -> b) -> b -> [a] -> b
mr_PM np unshuffle f g e tasks = foldl' g e results
  where results = [(worker_PM f g e) # mtasks
                   | mtasks <- unshuffle np tasks] `using`  spine
worker_PM f g e = process tasks -> foldl' g e (map f tasks)
```

Notice that an `unshuffle` function is provided, but not the corresponding `shuffle`: due to the associative and commutative properties of the parameter function `g`, the order in which the results are combined does not matter.

**Self-service Implementation.** In many situations (see e.g. Section 4.4.2) it can be done even better in case the list of tasks can be easily generated by each worker. In those cases, the self-service approach can be used. Thus, each worker can select its tasks, so that the communications are reduced:

```
mr_SSI :: Transmissible b => Int -> (Int -> [a] -> [[a]]) ->
          (a->b) -> (b->b->b) -> b -> [a] -> b
mr_SSI np unshuffle f g e tasks = foldl' g e results
  where results = [(worker f g e mtasks) # ()
                   | mtasks <- unshuffle np tasks] `using`  spine
        worker f g e tasks = process () -> foldl' g e (map f tasks)
```

As in the `map` case, the number of processes depends on the number of processors available. A predefined `unshuffle` function is provided to distribute the inputs in a round-robin fashion:

```
map_reduce_ssi :: Transmissible b => (a->b) -> (b->b->b) -> b -> [a] -> b
map_reduce_ssi =  mr_SSI noPe unshuffle

unshuffle :: Int -> [a] -> [[a]]
unshuffle n xs = [takeEach n (drop i xs) | i <- [0..n-1]]
  where takeEach :: Int -> [a] -> [a]
        takeEach n [] = []
        takeEach n (x:xs) = x : takeEach n (drop (n-1) xs)
```

The cost model for `map_reduce_ssi` is the following:

$$
\begin{array}{lcl}
t_{map\_reduce\_ssi} & = & L_{init} + t_{worker} + L_{final} \\
L_{init} & = & Pt_{create} + \delta \\
L_{final} & = & \delta + t_{unpackO} + t_{foldl_P} \\
t_{worker} & = & t_{\#} + t_{extract} + \lceil \frac{N}{P} \rceil (t_f + t_{packO}) + t_{fold_{\lceil \frac{N}{P} \rceil}}
\end{array}
$$

### 4.3.3 Task Parallel Skeletons

In contrast to data-parallel skeletons where the source of parallelism is the distribution of data between processors and the application of the same operation to all portions of the data, here the source of parallelism is the decomposition

of a task into different subtasks which can be done in parallel. These subtasks need not be identical.

The first skeleton we describe in this section, *divide and conquer*, is the parallel counterpart of the well-known sequential scheme. The parallelism comes from the fact that the different subtasks into which a given task is split, can be solved in parallel. In the second skeleton, the *pipeline*, different stages of a sequential computation can be done in parallel if they work on different elements of a continuous stream of data.

### Divide and Conquer

The sequential specification of this scheme is:

```
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 may be non-homogeneous, and that trivial solutions may appear at any level of the tree.

**Naïve Implementation.** The easiest way to parallelize the `dc` scheme is to replace `map` by `map_par`. The following implementation uses this approach, but stops the parallel unfolding at a given level `d`. A dynamic tree of processes is created with each process connected to its parent. The integer parameter `d` determines the maximum level after which no more child processes are generated, and the sequential version is used instead. The implementation is as follows:

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

Notice that in this implementation 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 use a distribution function of processes into processors and another distribution function of granularities into processes. The first distribution is known, as it only depends 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.

**Farm and Replicated Workers Implementations.** These implementations use the `farm` and `rw` implementations of `map`, in order to have a better control over process granularity and distribution, and to achieve a better load balance. Notice that, by using `rw` the load balance will be improved, even if the granularities of the different tasks are different. Also, the process creation overhead will be decreased, as 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_farm` (or better a `map_rw`) skeleton in which the function of the workers is just the sequential algorithm. In order to be able to appropriately combine the results returned by the parallel processes, an explicit tree of arguments must be generated when splitting the initial task. We only present the `dc_rw` skeleton. The corresponding `dc_farm` skeleton can be obtained by replacing `map_rw` by `map_farm`. Functions `generateTasks` and `combineTop` are simple Haskell definitions not shown.

```
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 4.3.2), 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 4.4.3 both `dc_par` and `dc_rw` are used and compared for a typical divide and conquer algorithm.

### Pipeline

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.

**Implementation.** A naïve parallelization of this scheme instantiates a different process to evaluate each of the pipeline stages. This can be expressed in Eden in several ways. For instance, in the following one, each process in the pipe creates its successor:

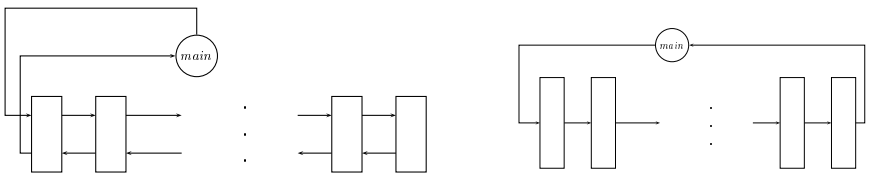


Figure 4.2: Topology generated with `pipe_naive` (left), and the desired pipeline using `pipeD` (right)

```
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)
```

However, this definition has a subtle problem: It does not achieve the desired topology because the last process of the pipe cannot send the values directly to the main process, as topologies in Eden are hierarchical by default (see Figure 4.2(left)). The solution is the use of Eden's *dynamic channels* facility (see Section 4.2) to establish a direct data connection between the last and the main process. The main 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 4.2(right). The similarities between both versions are remarkable. In fact, in [PRS01] we give a method that, given as specification a hierarchical program using only process abstractions and instantiations, derives non-hierarchical implementations using dynamic channels.

```
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. We assume $F > P$ and the round-robin mode for the RTS.

### 4.3.4 Systolic Skeletons

Systolic programs are those in which processes alternate parallel computation and global synchronization steps. Depending on the concrete problems, they can be classified as data parallel or task parallel. We first present the `iterUntil` skeleton, that iterates a parallel computation until a convergence condition is met, and then the `torus` and `ring` skeletons, in which processes communicate respectively using a torus or a ring topology. In these skeletons, the sequential specification is the same program as the parallel one, replacing the Eden runtime value `noPe` by 1.

#### Iterate Until

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. There are as many workers as processors. The difference with a `farm` or a `rw` skeleton is that 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 algorithms is solving linear systems by the *conjugate gradient* or the *Jacobi relaxation* methods [Qui94].

The manager is initialized with data of type `inp` (the problem input) and a manager local state of type `ml`. Each worker is initialized with data of type `wl` (worker local state) and one initial task of type `t`. At each iteration, each worker computes a sub-result of type `sr` which is transmitted to the manager, and a new local state which is used for its next computation. The manager combines the sub-results and, either produces a new set of tasks and a new local manager state, or it terminates with a result of type `r`. The Eden skeleton receives the following parameters:

- A `split` function to be used by the manager in order to compute the initial state and the initial task of each worker, and its own local state. It receives an integer telling into how many pieces the input should be split.

- The function `wf` to be used by the workers: given a local worker state and a task, it generates a sub-result and a new local state for the next round.

- The function `comb` to be used by the manager to combine the sub-results of the workers: it produces either the final result or a new list of tasks and a new local manager state for the next round.

- The input data of the problem, of type `inp`.

The Eden source code is the following:

```
iterUntil :: (Transmissible wl, Transmissible t, Transmissible sr) =>
             (inp -> Int -> ([wl],[t],ml)) ->     -- split function
             (wl -> t -> (sr, wl)) ->             -- worker function
             (ml -> [sr] -> Either r ([t],ml)) -> -- combine function
             inp -> r
iterUntil split wf comb x = result
  where (result, moretaskss)  = manager comb ml (transpose' srss)
        srss                  = map_par (worker wf) (zip wlocals taskss)
        taskss                = transpose' (initials : moretaskss)
        (wlocals,initials,ml) = split x noPe

manager :: (ml -> [sr] -> Either r ([t],ml)) -> ml -> [[sr]] -> (r, [[t]])
manager comb ml (srs : srss) = case comb ml srs of
            Left  res      -> (res, [])
            Right (ts,ml') -> let (res',tss) = manager comb ml' srss
                              in  (res',ts:tss)

worker ::  (wl -> t -> (sr, wl)) -> (wl, [t]) -> [sr]
worker wf (local, [])  = []
worker wf (local,t:ts) = sr : worker wf (local',ts)
  where (sr, local') = wf local t

transpose' = foldr (mzipWith' (:)) (repeat [])

mzipWith' f (x:xs) ~(y:ys) = f x y
mzipWith' f _ _  = []
```

The cost model is the following:

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

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.

### Torus

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 sends 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 defined below creates the desired toroidal topology by properly connecting the

inputs and outputs of the different `ptorus` processes. Each process 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 whole source code of the skeleton is the following:

```
torus :: (Transmissible a,Transmissible b,Transmissible c,Transmissible d)
         => Int -> (Int -> c -> [[c]]) -> ([[d]] -> d) ->
            ((c,[a],[b]) -> (d,[a],[b])) -> c -> d
torus np dist comb f input = comb outssToParent   where
  toChildren = dist np input
  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)

mzip3 (x:xs) ~(y:ys) ~(z:zs) = (x,y,z) : mzip3 xs ys zs
mzip3 _ _ _ = []
```

Notice that the size of the torus is a parameter that will usually depend on the number of available processors (the value of `np` will usually be $\lfloor\sqrt{\texttt{noPe}}\rfloor$), and that a function `dist` is used to distribute the input data to the `ptorus` processes, while `comb` is used to produce the final output from the subresults of the torus. The third 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, and produces results `[a]` and `[b]` for its neighbors and a final result `d` for its parent. Functions `mzip3`, `mzipWith` and `mzipWith3` are just lazier versions of functions of the `zip` family, the difference being that our functions use irrefutable patterns for most of its parameters, as shown for `mzip3`.

The cost model is the following:

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

$N$ 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 that $P = n \times m$, i.e. each worker has a separate processor. We are also assuming that a separate processor is devoted to the manager, that is, the number of processors is $P + 1$.

### Ring

A (unidirectional) ring can be considered a particular case of a torus, where each process —apart from sending and receiving values to/from the parent— is connected only to two neighbors: the previous link, from which it receives messages, and the next link, to which it sends messages. By using dynamic channels to provide direct connections between links, the `ring` function creates the desired topology. Each `pring` receives an input from the parent, and a channel name to be used to send values to the next link, and produces an output to the parent and a channel name to be used to receive inputs from the previous link. The whole source code of the skeleton is as follows:

```
ring :: (Transmissible a,Transmissible b,Transmissible c) =>
         Int -> (Int -> a -> [a]) -> ([b] -> b) ->
         ((a,[c]) -> (b,[c])) -> a -> b
ring n dist comb f input = comb toParent  where
  (toParent,nexts) = unzip outss
  outss      = [(pring f) # ins | ins <- inss]
  inss       = mzip toChildren prevs
  toChildren = dist n input
  prevs      = last nexts : init nexts

-- each individual process in the ring
pring ::(Transmissible a,Transmissible b,Transmissible c) =>
         ((a,[c]) -> (b,[c])) -> Process (a,ChanName [c]) (b,ChanName [c])
pring f = process (fromParent,nextChan) -> out
 where out= new (prevChan, prev)
              let (toParent,next) = f (fromParent,prev)
              in nextChan !* next par (toParent,prevChan)
```

The number of links is provided by the programmer as a parameter of the skeleton. Similarly to the torus, a function `dist` is used to distribute the input data to the `pring` processes, while `comb` combines in a final result the outputs produced by each link. The computation to be performed at each link is represented by a function $f$, which receives an initial datum of type `a` from the parent and a datum `[c]` from the previous link and produces output `[c]` for the next link and a local result of type `b` for the parent. Function `mzip` is just a lazier version of function `zip`.

The cost model for the ring is very similar to the one for the torus:

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

where $P$ is the number of links (each in a separate processor) and $N$ is the number of rounds each link does, i.e. the maximum length of lists `[c]`, and $t_{comp}$ is the cost of the worker function in each round.

## 4.4 Application Parallel Programming

In this section we present the results obtained for several examples, which are typical instances of the previously defined skeletons. For each example, both actual and predicted relative speedups are shown. The experiments have been performed in a 64-processor Beowulf cluster at the University of St. Andrews. Nodes are 450MHz Pentium II running Linux RedHat 5.2, with 348MB of DRAM and connected through a CISCO 2984G full duplex 100Mb/s fast Ethernet switch, being the latency $\delta = 142\mu$s. So, it is a low cost environment with high latencies. Eden RTS was running on top of PVM 3.4.2. All the processors of the Beowulf cluster are equal, so that the main potential sources of load imbalances come from the algorithms. Due to administrative reasons, it has not been possible to use all the processors in the tests.

### 4.4.1 Ray Tracing — Map

Given a scene consisting of 3D objects, and given the position of the camera, a ray tracer calculates a 2D image of the scene. For every pixel of the output image, the ray tracer shoots a ray into the scene and tests whether it impacts with any object of the scene. When an impact is found, the ray is reflected and the colour of the intersection point is computed based on the strength of the ray and on the texture of the object's material. The code is based on the Id version, that is part of the Impala suite [Imp01] of parallel benchmarks. The program was translated to Haskell by the group developing the GPH language.

The main function of the program is `ray`, which receives as parameters the size of the window `x` × `y` and the scene `world` consisting of a list of spheres. The computation is performed by two nested maps, applying a function `tracepixel` to each of the pixels of the window. Function `camparams` computes the parameters depending on the camera position:

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

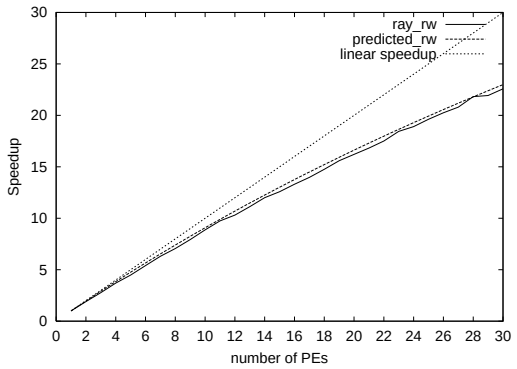


Figure 4.3: Speedups of the ray tracer

```
sizes_y = [0..y-1]
f world i j = tracepixel world i j firstray scrnx scrny
(firstray, scrnx, scrny) = camparams x y
```

This algorithm can be easily parallelized exploiting the outer `map` of the main function, and creating a task for each line of the window. Notice that the scene `world` is needed in all the tasks. Thus, we should use the versions of the `map` skeleton which incorporate an extra fixed parameter, to guarantee that the scene is communicated only once per processor:

```
ray x y world = map_rw_FD world do_line sizes_y    where ...
```

The speedups obtained for a $350 \times 350$ window and a scene of 640 spheres can be seen in Figure 4.3. The sequential execution time was 176.99 seconds. The speedups are quite good, the only inefficiency being a sequential bottleneck of 1.3 seconds while distributing tasks and combining the results.

### 4.4.2 Euler Numbers — Map and Reduce

The Euler number of a given value $x$ is the number of integers smaller than $x$ that are relatively prime to $x$. We are interested in computing the sum of the Euler numbers of the first $n$ numbers. This problem has recently been proposed in [TLP01] to compare the way in which different parallel languages based on Haskell are used. The sequential version is trivial:

```
sumEuler  :: Int -> Int
sumEuler n = sum (map euler [n,n-1..1])

euler   :: Int -> Int
euler x = length (filter (relprime x) [1..(x-1)])
```

Notice that the problem fits the map and reduce scheme, as the `euler` function is mapped while `sum` folds all the results into a single one. Moreover, the list

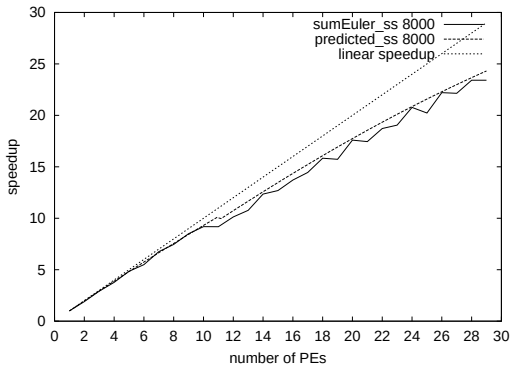


Figure 4.4: Speedups of sumEuler

of tasks can be trivially computed by each worker, reducing the communication overheads. Therefore, the parallelization is straightforward:

```
sumEuler n = map_reduce_ss euler (+) 0 [n,n-1..1]
```

The granularity of the `euler` function depends directly on the value of the input parameter. Thus, it is important to take care of the distribution of tasks between processes. The granularity decreases as the input value decreases. So, unshuffling the tasks in a round robin fashion gives a good distribution, as Figure 4.4 shows. The measurements were performed for a problem size of 8000, and the sequential time was 80.39 seconds. The only inefficiency in the parallelization is the time needed to create and initialize the worker processes, that is 0.015 seconds per processor.

### 4.4.3 Karatsuba Algorithm — Divide and Conquer

The Karatsuba algorithm [HS78] 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 $x_1 = x/b^n$, $x_2 = x \bmod b^n$, $y_1 = y/b^n$ and $y_2 = y \bmod b^n$.

- Let $u = x_1 * y_1$, $v = x_2 * y_2$, $w = (x1 + x2) * (y1 + y2)$.

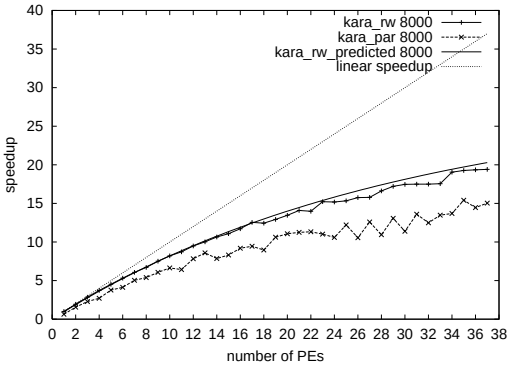


Figure 4.5: Speedup for the Karatsuba algorithm

- The result of the multiplication is $u * b^{2*n} + (w - u - v) * b^n + v$.

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 product 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. The implementation of the Karatsuba algorithm in terms of the divide and conquer skeleton 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)
```

Both the `dc_rw` and the `dc_par` version of the divide and conquer skeleton have been tested for the same input data, whose sequential execution time is 440 seconds. The speedups predicted and obtained can be seen in Figure 4.5. 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. The prediction of the `dc_rw` behavior is quite accurate. No prediction is given for the `dc_par` version owing to the lack of an accurate cost model.

### 4.4.4 Conjugate Gradient - Iterate Until

The gradient conjugate method is an iterative method used to find approximate solutions of linear systems $Ax = b$ in which the coefficient matrix $A$ is positive definite. In each iteration, the current solution $x$ is improved using the function

$$x(t) \;=\; x(t-1) \;+\; s(t)\,d(t)$$

where $d$ is the direction vector to the solution, and $s$ is the size of the scalar step. Each iteration requires the following computations:

$$(1)\quad g(t) \;=\; Ax(t-1) \;-\; b$$

$$(2)\quad d(t) \;=\; -\,g(t) \;+\; \frac{g(t)^T g(t)}{g(t-1)^T g(t-1)}\,d(t-1)$$

$$(3)\quad s(t) \;=\; -\,\frac{d(t)^T g(t)}{d(t)^T A\,d(t)}$$

$$(4)\quad x(t) \;=\; x(t-1) \;+\; s(t)\,d(t)$$

where in the first iteration $x(0)$, $d(0)$ and $g(0)$ are initialized to the zero vector, and $g(0)$ to $-b$. With this method, the values of $x$ are guaranteed to converge in at most $n$ iterations, being $n$ the number of equations of the system. As each step is in $O(n^2)$, the algorithm is in $O(n^3)$.

This algorithm fits well the `iterUntil` skeleton: It consists of several steps; each step can be parallelized; and the current step must completely finish before starting the next one.

The Eden code parallelizes the product of $A$ and $x$ and also the product of $A$ and $d$, as these are the time consuming parts of the algorithm. So, in each iteration of the algorithm, there are two steps to be performed in parallel. This can be included in the `iterUntil` skeleton by using an `Either` type to mark which step of the iteration is to be computed. The source code is the following:

```
type Input  = (Matrix,Vector,Vector,Vector,Vector)
type Task   = Either Vector Vector              -- d or x
type SubResult = Either Vector Vector           -- A d or A x
type LocalW = (Matrix,Vector)                   -- A_i and b_i
type LocalM = (Vector,Vector,Vector,Double,Int) -- d,g,x,gg,iterations

cg :: Int -> Matrix -> Vector -> Vector
cg a b = cg' a b nOs b (map negate b) where nOs = replicate (length b) 0
cg' a b x d g = iterUntil split f_it comb (a,b,x,d,g)   where
  split :: Input -> Int -> ([LocalW],[Task],LocalM)
  split (a,b,x,d,g) np = (splitIntoN np (zip a b), replicate np (Left d),
                          (d,g,x,prVV g g,length b))
  f_it :: LocalW -> Task -> (SubResult,LocalW)
  f_it l t = (f_it' l t,l)  -- The local state does not change
  f_it' (ai,bi) (Right x) = Right (zipWith (-) (prMV ai x) bi)  -- g
  f_it' (ai,bi) (Left d) = Left (prMV ai d)                     -- A d

  comb :: LocalM -> [SubResult] -> Either Vector ([Task],LocalM)
  comb (d,g,x,gg,cont) srs@(Left _:_)  = ...
  comb (d,g,x,gg,cont) srs@(Right _:_) = ...
```

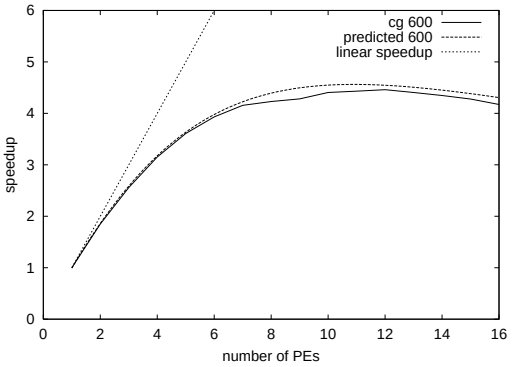


Figure 4.6: Speedups of the conjugate gradient algorithm

where `prVV` multiplies vectors, and `prMV` multiplies a matrix and a vector. Function `comb` performs the corresponding computations depending on the step of the iteration. That is, perfoms the computations (1) and (2) of the specification of the problem, or the computations (3) and (4).

Figure 4.6 shows the speedups obtained for a system of 600 equations, the sequential execution time being 684 seconds. The speedups do not scale well. The curve flattens out for more than four processor elements. The reason for these results is the bad computation-communication ratio. In each iteration, the main process communicates $O(n * p)$ values, while the computation of the workers are in $O(n^2/p)$. Thus, the communication costs increase with more processor elements while the computation decreases. Good speedups can only be expected if $n \gg p^2$.

The reasons why the speedups are worse than in other languages are two: (1) the packing and unpacking routines of Eden are not optimized yet, so the communications overheads are comparatively higher; and (2) there are not multicasting facilities to reduce the communications when a processor needs to send the same information to several processors. We hope to overcome these two limitations in the near future.

### 4.4.5 Matrix Multiplication — Torus

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:

```
type Matrix = [[Int]]
prMM :: Matrix -> Matrix -> Matrix
prMM   m1 m2 = prMMTr m1 (transpose m2)
prMMTr m1 m2 = [[sum (zipWith (*) row col) | col <- m2 ] | row <- m1]
```

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, a first approach could

be to generate $p$ tasks, each one evaluating $n/p$ rows of the resulting matrix. As the granularity of the tasks is very regular, the corresponding Eden program uses the simple `map_par` skeleton:

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

where `splitIntoN n xs` splits `xs` into `n` nearly equal size sublists.

The communications of the main process are in $O(n^2 * p)$ and the computation of each process will be in $O(n^3/p)$. Note that this is a similar ratio like the one for the conjugate gradient and so the speedup curve will rapidly flatten out for a fixed input size when increasing the number of processors (see Figure 4.8).

Gentleman's algorithm [Gen78] can be used to decrease the communication overhead. The matrices are distributed block-wise to the processes which are organized in 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 sub-matrices of the first matrix are rotated from left to right in the torus, while those of the second matrix are rotated from top to bottom. Each process computes a rectangular block of the final matrix, as depicted in Figure 4.7. The algorithm needs $\sqrt{p}$ iterations, where $p$ denotes the total number of processes in the torus. In each iteration, a process computes the product of its sub-matrices, adds this element-wise to its intermediate result block and communicates its sub-matrices with its neighbor processes.

To instantiate the torus skeleton one only needs 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 multiplications performed by each process is the size of the torus.

```
prMM_torus :: Matrix -> Matrix -> Matrix
prMM_torus m1 m2 = torus torusSize split combine (mult torusSize) (m1,m2)
  where torusSize = (floor . sqrt . fromInt) noPe
        combine   = concat . (map (foldr (zipWith (++)) (repeat [])))
        split     = ...
-- 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 `split` is a simple Haskell function that splits the matrices into blocks and shifts them appropriately to have matching torus inputs.

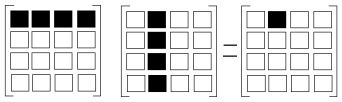

Figure 4.7: Matrix multiplication using blocks

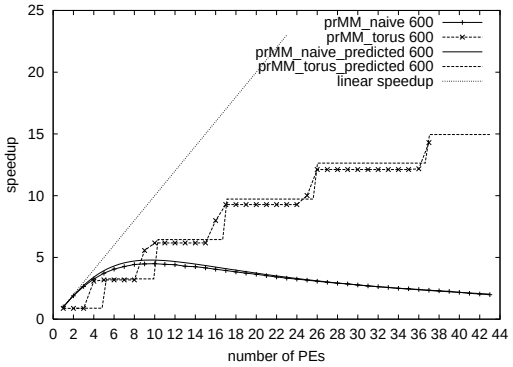


Figure 4.8: Speedups of matrix multiplication

The computation of each process is still in $O(n^3/p)$, while the communication overhead of the main process reduces to $O(n^2)$. The communication in each process is however now in $O(n^2/\sqrt{p})$. Another drawback is that a perfect square number of processes is needed to form the torus.

Figure 4.8 shows the speedup profiles for the two Eden versions of matrix multiplication using square matrices of size $600 \times 600$, the sequential execution time being 221 seconds. It can be seen that the first parallelization only scales well up to 8 processors and then flattens out. The predicted speedup of this 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 $L_{init}$. Thus, the communication overhead increases linearly with the number of processors.

The torus version scales much better and the prediction is also quite accurate in all points but in 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 be easily adjusted to take this fact into account.

The main reason why the torus scales better than the simple approach 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. The total communication cost incurred at the beginning of the computation is the same.

### 4.4.6 Pair Interactions — Ring

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

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

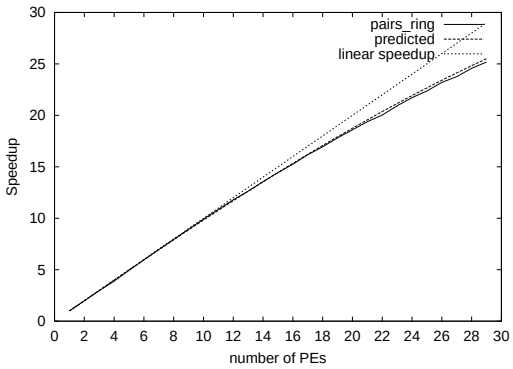


Figure 4.9: Speedups of pair interactions

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 $i$ is given the datum $x_i$ and computes $\{F(x_i, x_j) \mid i \neq j\}$. It is however inconceivable to have a separate process for each task when dealing with a large set of particles, as it is usually the case. Therefore, we distribute the atoms in as many subsets as the number of processors available. We use a ring structure, so that all the data can flow around. In the first iteration, each process will compute the forces between the local particles assigned to it. Then, in each iteration it will receive a new set of particles, and it will 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' np) 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
```

Figure 4.9 shows the speedups obtained using 7000 particles, the sequential execution time being 194.86 seconds. The total communications of each process are in $O(n)$, while its computations are in $O(n^2/p)$, $n$ being the number of particles and $p$ the number of processors. As the communications are now negligible, and the work to be done in the parent process is also minimal, the main inefficiency is the time needed in the parent to create and initialize the child processes: 0.03 seconds per child.

## 4.5 Related Work and Conclusions

Well-known approaches to introduce skeletons in a parallel language include: *Darlington et al.* [DFH+93], $P^3L$ [Pel98], *Skil* [BK96], and others. As Eden, Skil allows to design new skeletons in the language itself, the main difference being that Skil is based on the imperative host language C.

In *PMLS* [SMH01] Scaife et al. extend an ML compiler by machinery which automatically searches the given program for higher-order functions which are suitable for parallelisation. During compilation these are replaced by efficient low-level implementations written in C and MPI. In *HaskSkel* [HR99], Hammond and Rebón Portillo combine the evaluation strategies of *GpH* [THLP98] with Okasaki's Edison library [Oka00] (which provides efficient implementations of data structures) to implement parallel skeletons in GpH. Other funcional languages with parallel facilities are Concurrent Clean [Kes95] and Caliban [HM99, Chapter 14]. These languages would be appropriate for the definition of skeletons as they have an explicit notion of process. Nevertheless, not much work has been done in this direction.

The main differences between Eden and more traditional skeleton-based languages are two: (1) Eden is functional while the vast majority of skeleton implementation languages are imperative, and (2) skeletons can be implemented and used within the same language. In other approaches, skeletons are often implemented in a low-level language different from the language in which they are used.

The advantages of (1) can be experienced from the skeletons presented in this chapter. The whole code is included for most of them, and these code portions are rather short. Typically they fit is less than half a page. This is a consequence of the higher level of abstraction of functional languages compared to imperative ones. This higher level also extends to the coordination features. Compared to an implementation by using a message passing library such as MPI, less details are given. For instance, neither explicit message sending/receiving, nor initialization/termination routines need to be called.

The advantages of (2) are also evident: Eden, as a skeleton-based language, is easily *extensible*. The programmer may create new skeletons at any time, or modify the existent ones, and immediately use them in the same program text. Thus, Eden serves both as an application and as a system language, yielding a high degree of flexibility for the programmer. In other approaches, skeleton creation is a hard task and it is normally considered as a specialized system activity, or as part of the compiler contruction. Application programmers are not allowed to create new skeletons.

Of course, everything comes at a price. Eden, as a system language, offers to the programmer less opportunities for optimization than other lower-level languages. For instance, the packing conventions of Eden for communicating streams are often not convenient for some applications. Also, the lack of broadcasting facilities may lead to higher overheads (e.g. see Section 4.4.5).

All the speedups reported here are *relative* to the time of the same parallel program running in a single processor. So, absolute speedups, i.e. speedups

relative to the best sequential version of the algorithm, written for instance in C, are expected to be lower. This will be due, of course, to the constant factor between a Haskell implementation and one done in C (this factor has been reported to be around 4 in [HFea96]). But also, lower *relative* speedups than those of an implementation written, for instance, in C + MPI can be expected. These will be due to the lower overheads introduced by MPI with respect to our RTS, which have been remarked in the precedent paragraph.

So, we do not claim to achieve optimal speedups with Eden. Our divise can be summarized in the following sentence: *acceptable speedups at low effort*. If someone aims at better speedups, then a different language, and probably more effort, would be needed.

Parallel applications in Eden can also be done by explicitly instantiating processes. This corresponds to doing sequential functional programming with explicit recursion. Sometimes this is appropriate, but an experienced functional programmer will try to use higher-order functions, i.e. skeletons, as much as possible in order to reduce the amount of work and the possibility of making mistakes. Nevertheles, explicit process instantiation is not forbidden. A complex application could use both available skeletons and explicit instantiation. Even new skeletons could be defined and used in the program. This gives the programmer complete flexibility about the use of the parallel facilities of the language.

In this chapter several typical data-parallel, task-parallel and systolic skeletons have been considered in Eden. Each skeleton has been first specified by a sequential function and then implemented in parallel, some of them in several different ways. Cost models for predicting the execution time of the implementations have been defined. Several example programs have been parallelized using the skeletons and measured on a Beowulf cluster with several dozens of processing elements. The experiments have shown the flexibility and the efficiency of skeletal programming in Eden. The predictions of the cost models have been accurate.

# Bibliography


[BK96] G. H. Botorog and H. Kuchen. Efficient Parallel Programming with Algorithmic Skeletons. In *EuroPar*, LNCS 1123, pages 718 – 731. Ecole Normale Supérieure de Lyon, Springer Verlag, 1996.

[BKL98a] S. Breitinger, U. Klusik, and R. Loogen. From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View. In PLILP'98, pages 318–334. LNCS 1490, Springer-Verlag, 1998.

[BKL+98b] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Peña. DREAM: the Distributed Eden Abstract Machine. In *Selected Papers of Implementation of Functional Languages, IFL'97. St. Andrews, Scotland*, pages 250–269. LNCS 1467. Springer-Verlag, 1998.

[BLOMP97] 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.

[BLOP96] S. Breitinger, R. Loogen, Y. Ortega Mallén, and R. Peña. Eden: Language Definition and Operational Semantics. Technical Report, Bericht 96-10, revised version, Philipps-Universität Marburg, Germany, 1996.

[Col89] M. Cole. *Algorithmic Skeletons: Structure Management of Parallel Computations*. MIT Press, 1989. Research Monographs in Parallel and Distributed Computing.

[DFH+93] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel Programming Using Skeleton Functions. In *Parallel Architectures and Languages Europe*. Springer, 1993.

[GBDJ94] A. Geist, Ad. Beguelin, J. Dongarra, and W. Jiang. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[Gen78] W. M. Gentleman. Some Complexity Results for Matrix Computations on Parallel Computers. *Journal of the ACM*, 25(1):112–115, Jan 1978.

[Ham00] M. Hamdan. *A Combinational Framework for Parallel Programming Using Algorithmic Skeletons*. PhD thesis, Department of Computing and Electrical Engineering. Heriot-Watt University, 2000.

[HFea96] P. H. Hartel, M. Feeley, and M. Alt et al. Benchmarking Implementations of Functional Languages with "Pseudoknot", a Float-Intensive Benchmark. *Journal of Functional Programming*, 4(6):621–655, July 1996.

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

[HR99] K. Hammond and A. J. Rebón Portillo. HaskSkel: Algorithmic Skeletons for Haskell. In *Implementation of Functional Languages (IFL'99), Selected Papers*, LNCS 1868, Lochem, The Netherlands, September 1999. Springer-Verlag.

[HS78] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Pitman, 1978.

[Imp01] Impala. Impala – (IMplicitly PArallel LAnguage Application Suite). <URL:`http://www.csg.lcs.mit.edu/impala/`>, July 2001.

[Kes95] M. Kesseler. Constructing skeletons in Clean: The bare bones. In A. P. Wim Bohm and John T. Feo, editors, *High Performance Functional Computing*, pages 182–192, April 1995.

[KLPR01] 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, IFL 2000*. LNCS 2011, 2001.

[KOMP99] U. Klusik, Y. Ortega-Mallén, and R. Peña. Implementing Eden - or: Dreams Become Reality. In *Selected Papers of Implementation of Functional Languages, IFL'98, London, Sept. 1998*, pages 103–119. LNCS 1595. Springer-Verlag, 1999.

[KPR01] U. Klusik, R. Peña, and F. Rubio. Replicated Workers in Eden. In *Constructive Methods for Parallel Programming (CMPP'2000). To appear*. Nova Science, 2001.

[MPI94] MPI Forum. MPI: A Message-passing Interface Standard. International Journal of Supercomputer Applications, 8(3/4), 1994.

[Oka00] C. Okasaki. An Overview of Edison. In *Haskell Workshop*, 2000.

[Pel98] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor and Francis, 1998.

[Pey96] S. L. Peyton Jones. Compiling Haskell by Program Transformations: A Report from the Trenches. In *ESOP'96*. LNCS 1058, Springer-Verlag, 1996.

[PH99] S. L. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98*. URL http://www.haskell.org, February 1999.

[PHH+93] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Inf. Technology, Keele*, pages 249–257, 1993.

[PR01] R. Peña and F. Rubio. Parallel Functional Programming at Two Levels of Abstraction. In *Principles and Practice of Declarative Programming (PPDP01)*. ACM Press, September 2001.

[PRS01] R. Peña, F. Rubio, and C. Segura. Deriving Non-Hierarchical Process Topologies. In *Draft Proceedings of the 3rd Scottish Functional Programming Workshop*, 2001.

[PS01] R. Peña and C. Segura. Non-Determinism Analysis in a Parallel-Functional Language. In *Selected papers of Implementation of Functional Languages, IFL00.*, pages 1–18. LNCS 2011. Springer-Verlag, 2001.

[Qui94] M. J. Quinn. *Parallel Computing*. McGraw-Hill, 1994.

[SMH01] N. Scaife, G. Michaelson, and S. Horiguchi. Comparative Cross-Platform Results from a Parallelizing SML Compiler. In *Draft Proceedings of Implementation of Functional Languages, IFL'01*, Stockholm (Sweden), 2001.

[THJ+96] P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 1996.

[THLP98] P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), 1998.

[TLP01] P. W. Trinder, H. W. Loidl, and R. Pointon. Parallel and Distributed Haskells. *Journal of Functional Programming*, 2001. To appear.