High-level Process Control in Eden[⋆]

Jost Berthold, Ulrike Klusik, Rita Loogen, Steffen Priebe, and Nils Weskamp

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans Meerwein Straße, D-35032 Marburg, Germany
{berthold,klusik,loogen,priebe,weskamp}@informatik.uni-marburg.de

In: Kosch et al., *Euro-Par 2003 – Parallel Processing* (LNCS 2790), ©2003 Springer.

Abstract. High-level control of parallel process behaviour simplifies the development of parallel software substantially by freeing the programmer from low-level process management and coordination details. The latter are handled by a sophisticated runtime system which controls program execution. In this paper we look behind the scenes and show how the enormous gap between high-level parallel language constructs and their low-level implementation has been bridged in the implementation of the parallel functional language Eden. The main idea has been to implement the process control in a functional language and to restrict the extensions of the low-level runtime system to a few selected primitive operations.

1 Introduction

A growing number of applications demand a large amount of computing power. This calls for the use of parallel hardware including clusters and wide-area grids of computers, and the development of software for these architectures. Parallel programming is however hard. The programmer usually has to care about process synchronisation, load balancing, and other low-level details, which makes parallel software development complex and expensive. Our approach, Eden [3], aims at simplifying the development of parallel software. Eden extends the lazy functional language Haskell [14] by syntactic constructs for *explicitly* defining processes. Eden's process model provides direct control over process granularity, data distribution and communication topology while process creation and placement of processes, necessary communication and synchronisation are automatically managed by the runtime system (RTS), i.e. the implementation of Eden's virtual machine.

In his foreword to [6] Simon Peyton Jones writes: *"Parallel functional programs eliminate many of the most unpleasant burdens of parallel programming. . . . Parallelism without tears, perhaps? Definitely not. . . . The very high-level nature of a parallel functional program leaves a large gap for the implementation to bridge."* The Eden implementation[1] is based on the Glasgow Haskell Compiler (GHC) [13]. By embedding Eden's syntax into Haskell, we could use the front-end of the compiler without any changes. The bulk of the extensions lies in the

[⋆] Work supported by the DAAD (German Academic Exchange Service)

[1] Freely available at www.mathematik.uni-marburg.de/inf/eden and via the GHC CVS repository (see www.haskell.org/ghc).

back end of the compiler and the RTS [2, 7]. Kernel parts of the parallel functional RTS, like thread and memory management, are shared with GUM, the implementation of GpH [19]. Eden's RTS is implemented on top of the message passing library PVM [15].

In this paper we describe how the Eden implementation has been re-organised in layers to achieve more flexibility and to improve the maintainability of this highly complex system. The main idea underlying Eden's new layered implementation shown in Fig. 1 is to lift aspects of the RTS to the level of the functional language, i.e. defining basic workflows on a high level of abstraction and concentrating low-level RTS capabilities in a couple of primitive operations. In this way, part of the complexity has been eliminated from the imperative RTS level.

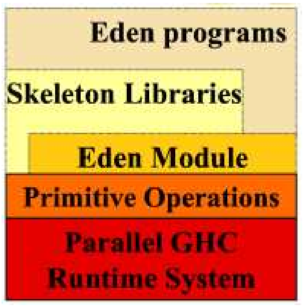

Fig. 1. Layer structure of the Eden system

Eden programmers will typically develop parallel programs using the Eden language constructs, together with parallel skeletons provided in special libraries [9]. This is briefly described in Section 2. Every Eden program must import the Eden module, which contains Haskell definitions of Eden's language constructs, as explained in Section 3. These Haskell definitions use primitive operations which are functions implemented in C that can be accessed from Haskell. The extension of GHC for Eden is mainly based on the implementation of these new primitive operations, which provide the elementary functionality for Eden. The paper ends with a discussion of related work and conclusions.

2 Eden's Main Features

Eden gives the programmer high-level control over the parallel behaviour of a program by introducing the concept of processes into the functional language Haskell [14]. Evaluation of the expression `(process funct) # arg` leads to the creation of a new process for evaluating the application of the function `funct` to the argument `arg`. The argument `arg` is evaluated locally and sent to the newly created process. With the high-level Eden constructs:

– `process :: (Trans a, Trans b) => (a -> b) -> Process a b`
  to transform a function into a process abstraction and
– `( # ) :: (Trans a, Trans b) => Process a b -> a -> b`
  to create a process,

the programmer can concentrate on partitioning the algorithm into parallel subtasks, thereby taking into account issues like task granularity, topology, and data distribution.

Eden processes are eagerly instantiated, and instantiated processes produce their output even if it is not demanded. These deviations from lazy evaluation 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 is avoided, to save the administration costs while paying the price of possibly duplicating work.

Processes communicate via *unidirectional channels* which connect one writer to exactly one reader. When trying to access input which is not available yet, threads are temporarily suspended. The type class `Trans` (short for transmissible) comprises all types which can be communicated and provides overloaded, implicitly used functions for sending values. All primitive types like `Int, Bool, Char` etc., pre- and user-defined algebraic types[2] as well as function and process abstraction types belong to `Trans`.

Example: The following function is at the heart of a simple ray-tracer program. It computes an image with `y` lines and `x` columns of a scene consisting of spheres. The sequential function body of the `ray` function is simply the expression `map (traceLine x world) [0..y-1]`. The parallel version produces the image by several processes each computing a chunk of lines:

```
ray :: Int -> Int -> Int -> [Sphere] ->  [[RGB]]
ray chunk x y world
  = concat ([process (map (traceLine x world)) # linenumbers
             | linenumbers <- splitAtN chunk [0..y-1]]
            `using`  spine)
```

The function `concat` flattens a list of lists into a list, thus removing one level of nested lists — the one introduced by the list of processes. The addendum `` `using` spine`` is needed to produce early demand for the evaluation of the process instantiations. ◁

Many-to-one communication is supported by a predefined process abstraction `merge` which, when instantiated, does a fair merging of input streams into a single (non-deterministic) output stream. Moreover, an Eden process may explicitly generate a new *dynamic input channel* (of type `ChanName a`) and communicate the channel's name to another process, which can use the channel for answering directly. This enables the creation of arbitrary communication topologies despite the tree-like generation of process systems. The primitive operations for dynamic channel creation are also used to establish the channels between parent and child processes during process instantiation (see Section 3).

The task of parallel programming is further simplified by a library of predefined skeletons [9]. Skeletons are higher-order functions defining parallel interaction patterns which are shared in many parallel applications [16]. The programmer can simply use such known schemes to achieve an instant parallelisation of a program. The automatic selection of appropriate skeletons during compile-time using new Haskell meta-programming constructs is discussed in [5]. The recent journal paper [8] shows that Eden achieves good performance in comparison with Glasgow parallel Haskell (GpH) [4] and Parallel ML with Skeletons (PMLS) [10].

[2] For user-defined algebraic types, instances of `Trans` must be derived.

3 A Layered Implementation of Coordination

This section considers the internals of Eden's layered implementation. These are transparent for the ordinary programmer except that every Eden program must import the Eden module. This module contains Haskell definitions of the high-level Eden constructs, thereby making use of the eight primitive operations shown in Fig. 2. The primitive operations implement basic actions which have to be performed directly in the runtime system of the underlying sequential compiler GHC[3]. In the Eden module, every primitive operation is wrapped in

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

Fig. 2. Primitive Operations for Eden

a function[4], thereby providing additional type information, triggering execution and limiting actual use to the desired degree. Information provided by primitives like `selfPE#` or `noPE#` should e.g. only be used for process control. The misuse of such operations may introduce nondeterministic effects.

In the following, we abstract from low-level implementation details like graph reduction and thread management which are well-understood and explained elsewhere [12, 19, 2]. Instead we focus on coordination aspects, in particular process abstractions and process instantiations and their implementation in the Eden module. Communication channels are now explicitly created and installed to connect processes. Primitives provided for handling Eden's dynamic input channels are used for that purpose.

3.1 Channels and Communication

As explained in Section 2, the type class `Trans` comprises all types which can be communicated. It provides an overloaded function `sendChan :: a -> ()` for sending values along communication channels where the channel is known from the context of its application. The additional functions `tupsize` and `writeDCs` shown in Fig. 3 will be explained later. The context NFData (normal form data)

[3] Note that, in GHC, primitive operations and types are distinguished from common functions and types by `#` as the last sign in their names.

[4] With the same name as the primitive operation, except for the `#`.

is needed to ensure that transmissible data can be fully evaluated (using the overloaded function `rnf` (reduce to normal form)) before sending it (using the primitive operation `sendVal#` wrapped by `sendVal`). Lists are transmitted in a *stream*-like fashion, i.e. element by element. For this, `sendChan` is specialized to `sendStream` which first evaluates each list element to normal form and transmits it using `sendHead` (see Fig. 3).

```
class NFData a => Trans a where
  sendChan :: a -> ();               sendChan x = rnf x `seq` sendVal x
  tupsize  :: a -> Int;              tupsize  _ = 1
  writeDCs :: ChanName a -> a -> (); writeDCs (cx:_) x = writeDC cx x
                                  -- default definitions, changed
                                  -- appropriately for tuples and lists
instance Trans a => Trans [a] where
  sendChan x = sendStream x

  sendStream :: Trans a => [a] -> ()
  sendStream []     = sendVal []
  sendStream (x:xs) = (rnf x) `seq` ((sendHead x) `seq` (sendStream xs))
```

Fig. 3. Type Class `Trans` with List Instance

Before any communication can take place, a channel must be created and installed. For this, the functions shown in Fig. 4 are provided. The RTS equivalent to channels is a structure *Port* which contains (1) a specific Port number, (2) the process id (pid), and (3) the processor number, forming a unique port address. At module level, these port addresses (connection points of a channel to a process) are represented by the type `ChanName'`. Objects of this type contain exactly the port identifier triple (see Fig. 4). Note that the higher level type `ChanName a` is a *list* of port identifiers; one for each component of type `a` in case `a` is a tuple.

The function `createDC :: Trans a => a -> (ChanName a, a)` creates a new (dynamic) input channel, i.e. a channel on which data can be received, using the corresponding primitive operation `createDC#`. It yields the channel name which can be sent to other processes and (a handle to) the input that will be received via the channel. If `createDC` is used for tuple types `a`, a list of port identifiers (type `ChanName'`) and a tuple of input handles will be created. To ensure correct typing, `createDC` is always applied to its second output, but will only use it to determine the needed number of channels, using the overloaded function `tupsize` in class `Trans`.

Data transmission is done by the function `writeDC`. This function takes a port identifier and a value, connects the current thread to the given port (`setChan#`) and sends the value using the function `sendChan`. The connection by `setChan#` prior to evaluating and sending values guarantees that the receiver of data mes-

```
type ChanName  a = [ChanName' a]
data ChanName' a = Chan Int# Int# Int#

createDC :: a -> (ChanName a, a)
createDC t = let (I# i#) = tupSize t
             in  case createDC# i# of (# c,x #) -> (c,x)

writeDC  :: ChanName' a -> b -> ()
writeDC chan a = setChan chan `seq` sendChan a
```

Fig. 4. Channel Creation and Communication

sages is always defined when `sendVal#` or `sendHead#` is called. While `writeDC` defines the behaviour of a single thread, the overloaded function `writeDCs` (see Fig. 3) handles tuples in a special way. It takes a list of port identifiers (length identical to the number of tuple components) and creates a thread for each tuple component. The instance declaration of `Trans` for pairs is e.g. as follows:

```
instance (Trans a, Trans b) => Trans (a,b) where
   tupsize _ = 2
   writeDCs (cx:cy:_) (x,y) = writeDC cx x `fork` writeDC cy y
```

The Eden module contains corresponding instance declarations for tuples with up to eight components.

3.2 Process Handling

Subsequently, we will focus on the module definitions for process abstraction and instantiation shown in Fig. 5 and 6. Process creation can be defined on this level, using the internal functions to create channel names and to send data on them, plus the primitive operation `createProcess#` for forking a process on a remote processor.

A process abstraction of type `Process a b` is implemented by a function `f_remote` (see Fig. 5) which will be evaluated remotely by a corresponding child process. It takes two channel names: the first `outDCs` (of type `ChanName b`) is a channel for sending its output while the second `chanDC` (of type `ChanName (ChanName a)`) is an administrative channel to return the names of input channels to the parent process. The exact number of channels which are established between parent and child process does not matter in this context, because the operations on dynamic channels are overloaded. The definition of `process` shows that the remotely evaluated function, `f_remote`, creates its input channels via the function `createDC`. Moreover, `writeDCs` is used twice: the dynamically created input channels of the child, `inDCs`, are sent to the parent process via the channel

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

Fig. 5. Haskell definitions of Eden process abstraction

```
( # ) :: (Trans a, Trans b) => Process a b -> a -> b
pabs # inps = case createProcess (-1#) pabs inps of Lift x -> x
data Lift a = Lift a

createProcess :: (Trans a, Trans b) =>
                 Int# -> Process a b -> a -> Lift b
createProcess on# (Proc f_remote) inps
        = let  (outDCs, outvals) = createDC outvals
               (chanDC, inDCs )  = createDC inDCs
               pinst = f_remote outDCs chanDC
          in outDCs ‘seq‘ chanDC ‘seq‘
             case createProcess# on# pinst of
                 1# -> writeDCs inDCs inps  ‘fork‘ (Lift outvals)
                 _  -> error "process creation failed"
```

Fig. 6. Haskell definitions of Eden process instantiation

`chanDC` and the results of the process determined by evaluating the expression `(f invals)` are sent via the channels `outDCs`[5].

Process instantiation by the operator `( # )` defines process creation on the parent side. To cope with lazy evaluation and to get back control without waiting for the result of the child process, the process results are lifted to an immediately available weak head normal form using the constructor `Lift`. Before returning the result, the Lift is removed. The function `createProcess` takes the process abstraction and the input expression and yields the lifted process result. The placement parameter `on#` is a primitive integer (type `Int#`) which can be used to allocate newly created processes explicitly. The current system does not make use of this possibility, processes are allocated round-robin or randomly on the available processors. The channels are handled using `createDC` and `writeDCs` in the

[5] The prefixes `in` and `out` in channel names in Fig. 5 and 6 reflect the point of view of a child process. Thus, in a process instantiation, the inputs `inps` for the child are written into the channels `inDCs`, which are outputs of the parent process.

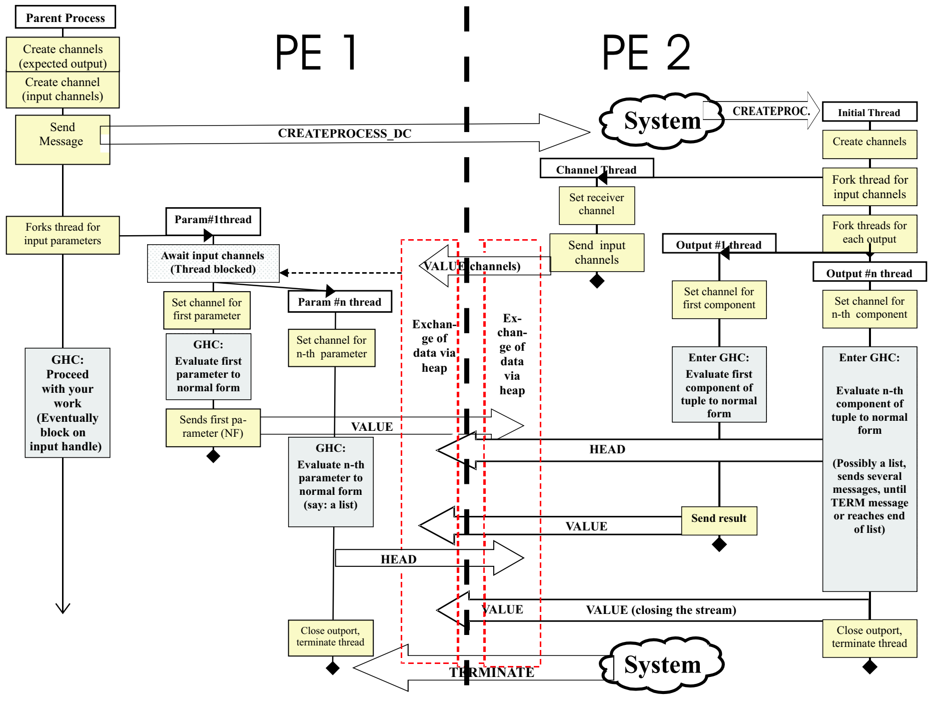
PE 1
PE 2
Parent Process
Create channels (expected output)
Create channel (input channels)
Send Message
CREATEPROCESS_DC
System
CREATEPROC.
Initial Thread
Create channels
Fork thread for input channels
Fork threads for each output
Channel Thread
Set receiver channel
Send input channels
Forks thread for input parameters
Param#1thread
Await input channels (Thread blocked)
VALUE (channels)
Output #1 thread
Output #n thread
Set channel for first parameter
Param #n thread
Set channel for n-th parameter
Exchange of data via heap
Ex-change of data via heap
Set channel for first component
Set channel for n-th component
GHC: Proceed with your work (Eventually block on input handle)
GHC: Evaluate first parameter to normal form
Enter GHC: Evaluate first component of tuple to normal form
Enter GHC: Evaluate n-th component of tuple to normal form (Possibly a list, sends several messages, until TERM message or reaches end of list)
Sends first parameter (NF)
VALUE
HEAD
GHC: Evaluate n-th parameter to normal form (say: a list)
Send result
VALUE
HEAD
VALUE
VALUE (closing the stream)
Close outport, terminate thread
Close outport, terminate thread
System
TERMINATE

Fig. 7. Sequence Diagram for Process Instantiation

same way as on the child side (see the process abstraction). The remote creation of the child process is performed by the primitive operation `createProcess#`.

The whole sequence of actions is shown in Fig. 7, which illustrates the interplay of the codes in Fig. 5 and 6. Process creation is preceded by the creation of new channels (one for every result) plus one additional port to receive channels for input parameters upon creation. The primitive `createProcess#` sends a message `createProcessDC` to a remote processor, which contains these channels and the process abstraction (an unevaluated `Proc f_remote` packed as a subgraph).

The remote processor (PE 2 in Fig. 7) receives this message, unpacks the graph and starts a new process by creating an initial thread. As the first thread in a process, this thread plays the role of a *process driver*. Evaluating the code shown in Fig. 5, it forks a first thread to communicate channels and then evaluates the results, forking one thread for each tuple component but the last, which is evaluated and sent back by the initial thread itself. Thus, one thread is only used to communicate channels, the other threads evaluate the output expressions. Threads block on the created input handles if they need their arguments. As soon as the input arrives, these threads are reactivated by the communication handler when it writes the received values into the heap.

This concludes our discussion of fundamental mechanisms in the Eden system.

4 Related Work and Conclusions

Implementations of parallel functional languages are either based on a parallel abstract machine or on parallel libraries, linked to an ordinary sequential system. Eden is a typical example for the monolithic approach (parallel abstract machine). It is closely related to GpH [19], using the same framework and even sharing a lot of code. Although GpH, combined with evaluation strategies [18] provides comparable control, it generally follows the concept of implicit, or "potential" parallelism, whereas Eden passes parallelism control to the programmer. During the last decade, the extension of functional languages for parallelism moved from implicit to more and more explicit approaches, because it became clear that an effective parallelisation needs some support from the programmer. The *para-functional programming* approach [11], as well as *Caliban* [17], provide explicit control over subexpression evaluation and process placement, going essentially further than our process concept.

Providing a set of parallel skeletons [16] is another way of implementing (more or less explicit) parallelisation facilities. Parallel skeletal languages, as e.g. P^3L [1], provide a fixed set of skeletons, sometimes combined with sophisticated program analysis, as e.g. in PMLS [10]. The skeleton implementation is usually out of reach for the programmer, whereas in Eden, programming a skeleton requires nothing but ordinary parallel functional programming.

By lifting the implementation of explicit low-level process control out of the RTS into the functional language itself, we reach two goals: As new techniques like Grid-Computing evolve, it becomes more and more important for a parallel programming system to provide not only good performance and speed-up behaviour, but also to make parallel programming as simple as possible. We achieve this not only on the highest level, i.e. for the ordinary application programmer, but also for the advanced programmer interested in the development of skeletons or even parallel extensions. An advantage of our layered implementation is that developers can use the high-level layers of module and skeleton library which we have introduced.

With rising demand for efficient compilers and better exploitation of parallel computers, compiler construction is getting much more complex. By the layer concept, we gain advantages on portability, code reuse, extensibility, maintenance, and abstraction on the implementation side. The implementation based on a few primitive operations leads to clean interfaces between the implementation layers, which makes it easier to follow version changes of the underlying sequential compiler. This is the first step to our long-term objective: the design of a generic parallel extension of sequential functional runtime systems, on top of which various parallel functional languages could be implemented.

Acknowledgements. We thank Hans-Wolfgang Loidl, Simon Marlow and Fernando Rubio for their support during the development of the Eden runtime system. Additionally, we are grateful to Phil Trinder and the anonymous referees for helpful comments on this paper.

References

1. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High Level Programming Language and its Structured Support. *Concurrency — Practice and Experience*, 7(3):225–255, May 1995.
2. 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. Springer, 1998.
3. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language Definition and Operational Semantics. Technical report, 1996. Available at http://www.mathematik.uni-marburg.de/inf/eden.
4. Glasgow Parallel Haskell. WWW page. http://www.cee.hw.ac.uk/~dsg/gph/.
5. K. Hammond, J. Berthold, and R. Loogen. Automatic Skeletons in Template Haskell. In *HLPP 2003. Paris, France*, June 2003.
6. K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, 1999.
7. U. Klusik, Y. Ortega-Mallén, and R. Peña Marí. Implementing Eden – or: Dreams Become Reality. In *IFL'98*, LNCS 1595, pages 103–119. Springer, 1999.
8. H.-W. Loidl, F. Rubio Diez, N. Scaife, K. Hammond, U. Klusik, R. Loogen, G. Michaelson, S. Horiguchi, R. Pena Mari, S. Priebe, A. R. Portillo, and P. Trinder. Comparing parallel functional languages: Programming and performance. *Higher-order and Symbolic Computation*, 16(3), 2003.
9. R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Abstractions in Eden. In F. A. Rabhi and S. Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002.
10. G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Appl.*, 16:181–206, 2001.
11. R. Mirani and P. Hudak. First-Class Schedules and Virtual Maps. In *FPCA'95*, pages 78–85. ACM Press, June 1995.
12. S. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *J. of Functional Programming*, 2(2):127–202, 1992.
13. S. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a Technical Overview. In *JFIT'93*, pages 249–257, March 1993. http://www.dcs.gla.ac.uk/fp/papers/grasp-jfit.ps.Z.
14. S. Peyton Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language, 1999. Available at http://www.haskell.org/.
15. PVM: Parallel Virtual Machine. WWW page. http://www.epm.ornl.gov/pvm/.
16. F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002.
17. F. Taylor. *Parallel Functional Programming by Partitioning*. PhD thesis, Department of Computing, Imperial College, London, 1997. http://www.lieder.demon.co.uk/thesis/thesis.ps.gz.
18. P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1):23–60, 1998.
19. P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96*, pages 78–88. ACM Press, May 1996.