Chapter 5

Controlling Parallelism and Data Distribution in Eden

Ulrike Klusik, Rita Loogen, and Steffen Priebe[1]

Abstract: The parallel functional language Eden uses explicit processes to export computations to other processor elements and to achieve parallelism. As Eden is based on the non-strict functional language Haskell, this raises the question in which way and to which degree the lazy evaluation strategy of Haskell should be transferred to the parallel setting. A modification is needed, as a completely demand driven evaluation would not lead to real parallelism but only to distributed sequentiality. The non-existence of a global shared memory for all processes raises a second question, namely, how shared data should and can be distributed across the available processing elements. In general, one has to choose between data duplication via communication and work duplication by recomputation. In total we will discuss the interaction of laziness, parallelism and data distribution in Eden. We explain the evaluation model underlying Eden's parallel implementation and justify corresponding design decisions.

5.1 INTRODUCTION

The lazy evaluation strategy of non-strict functional languages delays the evaluation of expressions as long as possible. Thus in a parallel version of such a language effects associated with these expressions are also delayed. Depending on the kind of effect (recognizing an expression as executable in parallel, creating a process), parallelism will be more or less degenerated. This makes it necessary to abandon lazy evaluation to a certain degree.

In Glasgow Parallel Haskell (GpH) [12] explicit evaluation strategies have been introduced to control the dynamic behaviour of evaluations and in partic-

[1]Department of Mathematics and Informatics, Marburg University, Germany
Email: {`klusik,loogen,priebe`}`@mathematik.uni-marburg.de`,
Supported by the DAAD (Deutscher Akademischer Austauschdienst)

ular to impose early additional demand in favour of parallelism. Although the programmer controls the amount of potential parallelism with this approach, the actual parallelism remains under the control of the runtime system. Concurrent Clean [8] takes a similar approach providing low-level annotations to start remote and locally interleaved evaluations. The latter can be used to introduce additional demand. Tremblay and Gao [11] also investigated possibilities to introduce additional demand: result data structures are traversed in a breadth-first manner leading to parallel head-strict speculation, which evaluates different levels of a data structure speculatively in parallel.

The parallel functional language Eden [4] takes a different approach where the programmer completely controls the parallelism and the data distribution. Eden is explicit about processes and their incoming and outgoing data but abstracts from the transfer of these data between processes and the necessary synchronisation. An Eden program defines a system of processes which exchange data on communication channels. The latter are modeled by head-strict lazy lists, as in stream-based I/O. Eden extends the non-strict functional language Haskell [9] but deviates from lazy evaluation in the following cases:

- Evaluation to normal form is used for process outputs instead of evaluation to weak head normal form.
- Communication is not demand-driven. Values are sent to the receiver process without prior requests until the receiver notifies that it does not need these input values any more. In general terms, the system will push instead of pull information.
- Process instantiations (expressions whose evaluation leads to the creation of a process) in local definitions (let- and where-blocks) are processed in a special way: instead of creating a closure representation in the heap, the process is immediately created.

In the following we discuss how to take advantage of these modifications for parallel programming. We show that in spite of the above changes, evaluation strategies are still necessary to produce early demand for the unfolding of certain process systems. Additionally we discuss the pros and cons of distributing data versus distributing computations.

5.2 EDEN

Eden extends the non-strict functional language Haskell [9] by syntactic constructs for *explicitly* defining processes, whose actions (like communication, synchronization) are, however, handled by the runtime system [4]. Thus, Eden's process model provides direct control over process granularity, data distribution and communication topology but frees the programmer from having to cope with low-level details. We will introduce the language by example and describe only constructs relevant for this paper. A complete description including the operational semantics can be found in [3].

```
multp    :: (Num a, Transmissible a) =>
            a -> Process [a] [a]
multp n =  process ns -> map (*n) ns

mergeThree :: (Ord a, Transmissible a) =>
               Process ([a],[a],[a])    [a]
mergeThree =  process (xs, ys, zs) -> sm xs (sm ys zs)
  where
    sm []      ys     = ys
    sm xs      []     = ys
    sm (x:xs) (y:ys) | x <  y     = x : sm xs (y:ys)
                     | x == y     =     sm (x:xs) ys
                     | otherwise = y : sm (x:xs) ys
hamming :: [Int]
hamming =  1: mergeThree # ( (multp 2) # hamming,
                             (multp 3) # hamming,
                             (multp 5) # hamming )
```

FIGURE 5.1. Eden program computing Hamming numbers

Syntax. The main Eden coordination constructs are *process abstractions* and *process instantiations*. Process abstractions define the behaviour of processes, while process instantiations are used to create processes by instantiating process abstractions with input values. Processes communicate via unidirectional one-to-one communication channels. Inports and outports connect channels to processes. A type context `Transmissible a` provides implicitly used functions for the transmission of values of type `a`.

Example: The example program given in Figure 5.1 defines a process network `hamming` which computes the sorted sequence of *Hamming numbers*, i.e. all numbers $2^i 3^j 5^k$ with $i, j, k \geq 0$. The functions `multp` and `mergeThree` define *process abstractions* which determine the behaviour of the processes as a mapping from incoming to outgoing data (transmitted via inports and outports). A process created using `multp` multiplies each element of its input list with a given number. A process created using `mergeThree` merges three sorted input streams into a single stream, removing duplicates at the same time.

The process network is defined by the function `hamming`. For $i = j = k = 0$, we get the first Hamming number 1. The sequence of Hamming numbers is multiplied by 2, 3 and 5 using processes created by process instantiations `((multp` i`) # hamming)` ($i \in \{2,3,5\}$) with `#` being the instantiation operator for processes. The resulting subsequences are merged by another process created by `(mergeThree # (...))`. □

Semantics. Denotationally, process abstractions and instantiations can be identified with function definitions (λ-abstractions) and applications, i.e. in particular, the non-strict semantics of Haskell is preserved in Eden. However, this view neglects parallelism. If one wants to argue about parallelism a closer look at the operational semantics is necessary. Operationally, concurrent activities (processes and threads) are initiated when a process instantiation is evaluated. Each process consists of as many threads as it has outports, because for each outport a thread is created that evaluates the outport expression. This general description leaves the following questions open:

1. When should a process instantiation be evaluated (a process be created)?
2. To which degree should outport expressions be evaluated?
3. When should the results of outports be transmitted between processes?

Eden's operational semantics [3] answers these questions clearly as follows:

1. Local definitions of process instantiations in `let` or `where` blocks, i.e. equations of the form `outp = pabs # inps`, are immediately evaluated when they are accessed during the evaluation, i.e. they immediately lead to the creation of a process.
2. outport expressions are evaluated to normal form, except for expressions with a function type, which are evaluated to weak head normal form.
3. Results of the evaluation of outports are sent to the connected inports as soon as they are available. Lists are transmitted as streams, i.e. elementwise.

This semantics served as a solid basis for Eden's parallel implementation [2]. The recent re-formulation [6] in the style of [1] coincides with the original definition in [3] but is more precise about expression scheduling and evaluation order. The modifications are only related to our parallel extensions of Haskell and do not change the non-strict semantics of the computation language. The Eden operational semantics yields the same value of an expression as the denotational semantics sketched above. It will evaluate at least the same or even more subexpressions than the lazy evaluation strategy employing a set of additional threads for the latter. In the following section, we will discuss alternative answers to the above questions and justify the design decisions that have been taken in Eden.

5.3 PARALLELISM CONTROL

Let us consider the following conservative answers to the above questions which obey the demand-driven (lazy) evaluation strategy of Haskell:

1. A process instantiation is only evaluated when there is demand for its output.
2. All output expressions are evaluated to weak head normal form (WHNF). Further evaluation takes only place when the receiver/consumer of the output demands this. In a distributed setting demands will be propagated using request messages.
3. Results of evaluations are transmitted on outports, when the receiver process demands them.

main	Process “mergeThree”	Process “multp 2”
m = (take 10 hamming) (take 10 (1:mt)) 1:(take (10-1) mt) 1:(take 9 mt)		
$\text{MERGETHREE}_{\text{CREATE}}$	mt = sm mp2 (sm mp3 mp5)	
	$(\text{MULTP 2})_{\text{CREATE}}$	mp2 = map (*2) hamming
hamming = m1:m2 m1 = 1 m2 = mt		$\text{HAMMING}_{\text{REQWHNF}}$
$\text{HAMMING}_{\text{DELIVERWHNF}}$		map (*2) (m1:m2) (m1*2):map (*2) m2 mp2 = mp21:mp22 mp21 = (m1 * 2) mp22 = map (*2) m2
	sm (mp21:mp22) (sm mp3 mp5)	$\text{MP2}_{\text{DELIVERWHNF}}$
⋮	$(\text{MULTP 3})_{\text{CREATE}}$ ⋮	⋮

TABLE 5.1. Distributed sequentiality

Example: Consider the computation of the first 10 Hamming numbers using the Eden program given in Figure 5.1: `main = (take 10 hamming)`, where `take` is a Haskell prelude function. The evaluation will proceed as shown in Table 5.1. This sequence clearly shows that the demand driven evaluation does not permit any parallelism. At each time at most one process is active and the other processes are waiting for demand. □

To overcome this problem, it is necessary to break the completely demand-driven evaluation at some points. As a first step, we replace the evaluation of outport expressions to WHNF by a normal form evaluation, because in many cases the whole output of a process is needed for the overall computation; precomputing it will save time later.

5.3.1 Normal Form Evaluation of Output Expressions

The normal form evaluation of all process outputs will keep the processes evaluating their output expressions independently of an explicit demand of these values. This policy introduces speculative evaluations and it is important to incorporate a scheme for terminating processes the results of which will not be consumed.

Example: Consider again the Hamming example. The normal form evaluation of outputs does not change the sequence of computation steps shown before, because the main process is blocked waiting for the WHNF result of the `mergeThree` process which has to wait for the WHNF result of the `(multp 2)` process. Only after all the subprocesses have been created and more hamming numbers have been produced, can the multiplication processes proceed independently. Thus the startup phase will proceed sequentially, but as soon as all processes have been created they will start to produce new results continuously. □

The example shows that the normal form evaluation of process outputs will lead to a real parallel evaluation, as soon as all the processes have been created. However, a problem is the many request messages that have to be sent to distribute the process results to the places where they will be consumed. Such request messages may lead to an enormous communication overhead, as for each data item that is transferred between processes, two messages will be sent: the request message and the answer message delivering the data value. Request messages can completely be avoided by replacing the demand-driven communication by an eager communication.

5.3.2 Eager Communication

An eager communication scheme means that process outputs are sent to the receiver processes as soon as the result values are available. No explicit request message is necessary. Each process will automatically deliver its results. This policy bears the danger of flooding a receiver's heap. The runtime system has to take care of this by throttling a producer process whose results are not consumed fast enough.

Example: Consider again the above Hamming evaluation. The results of the `multp` processes are e.g. immediately sent to the `mergeThree` process. This eliminates request messages and supports the progress of the evaluation. □

For the Hamming example, replacing the communication scheme and outputs' evaluation is sufficient to initiate a parallel evaluation of the processes. The reason is that the process instantiations only occur in communication contexts. Note that further processes are only created after the first process produced the WHNF of its result, because otherwise the computation within the `mergeThree` process does not proceed. Thus, the creation of subsequent processes is delayed until the previously created processes deliver the WHNF of their results. This leads to bad startup times and is usually not intended by the programmers. Therefore it is necessary to go one step further and allow the early evaluation of process instantiations without an explicit demand for their results.

5.3.3 Early Evaluation of Process Instantiations

To understand the need for early process creation, we will parallelise the well-known function `map` in such a way that each element of the result list is evaluated by a separate process each being immediately created when the parallel map is applied:

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

A straightforward definition of a parallel `map` is:

```
parMap1   :: (Transmissible a, Transmissible b) =>
             Process a b -> [a] -> [b]
parMap1 p =  map (p #)
```

However, lazy evaluation of the expression `(parMap1 p [i1,i2,...,im])` will yield the WHNF `((p #) i1) : parMap1 p [i2,...,im]` without creating any process. The expression `((p #) i1)` is not even a process instantiation, but only an application which creates an instantiation. The other list elements `[i2,...,im]` all remain completely untouched. Obviously it is necessary to put additional demand on the list and its elements.

At a first glance, it seems possible to solve the problem *solely* by using *evaluation strategies* [12], which are lazy higher-order functions forcing sequential (or even parallel) evaluation of an expression to a wide range of degrees using the combinators `seq` and `par`. For example, the strategy `rnf` evaluates the given expression sequentially to normal form. A strategy is applied via the `using` function. However, applying evaluation strategies for imposing additional demand on the incomplete `parMap1` evaluation does not solve the problem, because the expression `(parMap1 p [i1,i2,...,im]) 'using' rnf` will wait for the termination of a process before creating the next one; the flow of control will never split. Although all processes will be created eventually, one will experience only *distributed sequentiality* instead of real parallelism.

As a solution, the early evaluation of process instantiations in `let` or `where` blocks has been introduced. It allows the programmer to mark process instantiations that should be evaluated eagerly. With the following definition, evaluation of a `parMap` application to WHNF will lead to the creation of the first process:

```
parMap2 p []      = []
parMap2 p (i:ins) = let o = p # x
                    in  o : parMap2 p ins
```

The list comprehension version of `parMap`

```
parMap2 :: (Transmissible a, Transmissible b) =>
           Process a b -> [a] -> [b]
parMap2 p ins = [ p # i | i <- ins ]
```

has the same effect due to its transformation by the compiler. It provides a more comfortable notation. By applying the `spine` strategy, which traverses the list structure without touching the list elements, the initially intended parMap behaviour will be obtained:

```
parMap :: (Transmissible a, Transmissible b) =>
          Process a b -> [a] -> [b]
parMap p ins = [ p # i | i <- ins ] 'using' spine
```

5.3.4 Summary

The following table sums up the discussion of the last subsections. In the first step the outport evaluation was changed from WHNF to NF evaluation, because the output of a process is usually needed completely. The second step included changing the transmission of results from a demand-driven to an eager model, resulting in less communication. The last step dealt with the early instantiation of processes using evaluation strategies only for imposing additional demand.

Question	5.3 (Haskell)	5.3.1	5.3.2	5.3.3 (Eden)
Creation time	demand	demand	demand	*speculative*
Outport evaluation	WHNF	*NF*	NF	NF
Result transmission	demand	demand	*eager*	eager

5.4 DATA DISTRIBUTION

Eden process systems are distributed, i.e. each process is a closed entity with its own environment which is shared only by its threads. During its creation, the whole environment of a process abstraction is copied, possibly including non-evaluated expressions which may then be evaluated more than once by different processes. Recomputation of globally shared expressions can be avoided by evaluating them only once and communicating their values explicitly to the sharing processes. Sometimes recomputation is more efficient than communication and in such a case it is better to simply include the expression in the process' environment.

5.4.1 Parameter Passing and Communication

Consider the following function `fun2proc` which maps a function to a process abstraction:

```
fun2proc     :: (a -> b -> c) -> a -> Process b c
fun2proc f x =  process y -> f x y
```

The parameter function `f` takes two arguments. The first is passed implicitly as a function parameter to the process abstraction, the second will explicitly be communicated to the process. Thus, in an instantiation:

```
(fun2proc fexpr xarg) # yarg
```

the newly created process will get a copy of the (maybe unevaluated) expressions `fexpr` and `xarg` and evaluate them on demand, while the generator process evaluates the expression `yarg` to normal form and sends it to the new process.

Let `h :: a -> b` be a function that should be converted into a process abstraction. There are two alternatives for passing the parameter to this function:

1. communication:
```
mkProc   :: (a->b) -> Process a b
mkProc h =  process x -> h x
```

```
map_farm      :: (Transmissible a,Transmissible b) =>
                  (a -> b) -> [a] -> [b]
map_farm f xs =  shuffle (parMap (mkProc (map f))
                                 (unshuffle noPE xs))
unshuffle      :: Int -> [a] -> [[a]]
unshuffle n [] =  replicate n []
unshuffle n xs
   | len < n    = map (:[]) xs ++ replicate (n-len) []
   | otherwise  = zipWith (:) (take n xs) xss
               where xss = unshuffle n (drop n xs)
                     len = length xs
shuffle     :: [[b]] -> [b]
shuffle []  =  []
shuffle xss =  (map head nonEmptyxss) ++
               shuffle (map tail nonEmptyxss)
            where nonEmptyxss = filter (not.null) xss
```

FIGURE 5.2. Parallel `map` via process farm

This corresponds to using the function `h_c :: () -> a -> b` with `h_c () = h` as a parameter function in `fun2proc`.

2. environment:

```
rfi     :: (a->b) -> a -> Process () b
rfi h x =  process () -> h x
```

   This implements a *remote function invocation* of `h` and corresponds to using `h_e :: a -> () -> b` with `h_e x () = h x` as a parameter function in `fun2proc`.

It depends on the function `h` and its context which version is the best to choose. To investigate the difference between parameter passing via the environment and explicit communication via ports, we consider two parallel versions of the function `map`, in addition to the one we have already discussed in Section 5.3.3. In both parallelisations we generate a finite number `noPE` (globally defined) of worker processes. Each process has to process a part of the input list. The parallelisations vary in the following respect:

process farm: The main process evaluates the input list, divides it into parts (`unshuffle`), communicates the parts to the corresponding workers, and combines the results (`shuffle`).

direct mapping: The whole input list is passed as a parameter to each worker process which `extracts` the part it has to process by itself.

Figures 5.2 and 5.3 show the Eden programs for these two so-called *process schemes*.

```
map_dm :: (Transmissible a,Transmissible b) =>
          (a -> b) -> [a] -> [b]
map_dm f xs
  = shuffle
      [ (rfi (((map f).(extract pid noPE)) xs)) # ()
      | pid <- [0,1..np-1] ] `using` spine
extract         :: Int -> Int -> [a] -> [a]
extract i np xs = (unshuffle np xs) !! i
```

FIGURE 5.3. Parallel `map` via direct mapping

5.4.2 Experimental Results

We show the runtimes and speedups obtained with a simple ray tracer that has been parallelized using the two process schemes described in the previous section.

Problem Description. Given a scene consisting of 3D objects and a camera, a raytracer calculates a 2D image of the scene. For every pixel of the output image, the raytracer shoots a ray into the scene and tests whether it impacts with any object of the scene. In this simple version object properties like reflection and transparency are neglected. Therefore a pixel gets the color of the object its corresponding ray hits first; if it hits none at all, it gets the color of the background. The scene itself is maintained as an unordered list of objects, so that the effort to compute the nearest impact with the objects is nearly the same for all rays. The main function of the raytracer is `rt`:

```
rt :: ScreenSize -> CPos -> [Objects] -> [Impact]
rt detail cameraPos scene
  = map (firstImpact scene) allRays
    where allRays = generateRays detail cameraPos

firstImpact :: [Objects] -> Ray -> Impact
```

The parallelisations simply replace the function `map` by the parallel versions `map_farm` and `map_dm` shown in Figures 5.2 and 5.3. To avoid the network from being flooded with lots of small messages, the parallel farm version additionally packed the rays for transmission into packets of a fixed size.

Results. Eden's implementation extends the Glasgow Haskell Compiler, version 3.02. The Eden runtime system consists of a set of multi-threaded abstract machines, each of which is mapped to a separate processor element (PE) on a multi-computer system [2]. The programmer can choose between the message passing libraries MPI (Message Passing Interface) [10] and PVM (Parallel Virtual Machine) [5]. The Eden implementation is architecture-independent in that it can be ported to any system where the Glasgow Haskell compiler can be installed and where at least one of the above message passing libraries is available. We made our measurements on a network of workstations (NOW) consisting of 6 UltraSparc IIi (300Mhz) each with 128 MByte memory and 2MB cache, running Solaris 2.6. PVM 3.4.2 has been used as message passing library based on TCP/IP.

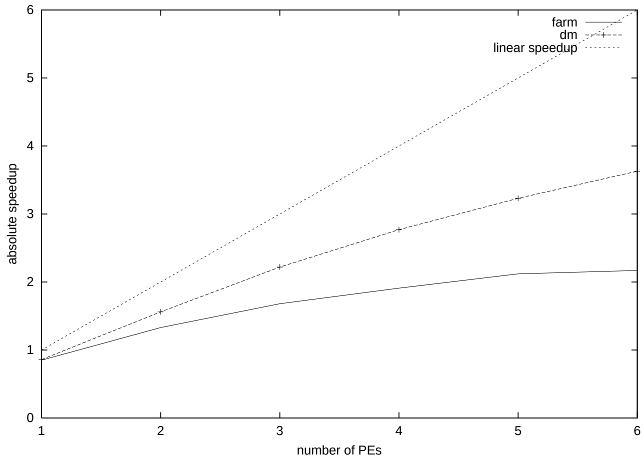

FIGURE 5.4. Absolute speedups of the ray tracer

	1	2	3	4	5	6
seq.	54.78					
farm	64.58	41.26	32.64	28.65	25.88	25.2
dm	64.02	35.1	24.64	19.76	16.98	15.08

TABLE 5.2. Runtimes (given in seconds) of the ray tracer

For the runtime measurements we used a screen of 400×400 pixels with a scene consisting of 17 objects, so the shared data was relatively small. Experiments showed that a packetsize of 200 yielded the best runtimes. From the runtimes (shown in Table 5.2) and the absolute speedups (shown in Figure 5.4) we see that the *direct mapping scheme* was clearly better than the *process farm*. The reason is that for this version less communication is needed, as the rays for each process are passed implicitly during process creation and are not communicated explicitly as it is done in farm.

This example reveals the typical relationship between the schemes that has been observed with several other benchmark programs in this measurement environment [7]. The high-latency of the communication network causes a high overhead for the farm scheme that outweighs the partial re-evaluation of the task list in the direct mapping scheme.

5.5 CONCLUSIONS

Eden's parallel operational semantics keeps a reasonable balance between the demand-driven evaluation of its computation language Haskell and the more eager semantics of the coordination constructs. In the parallel context the control of a program's dynamic behaviour, e.g. by the use of evaluation strategies, becomes more important than in the sequential world. In Eden, evaluation strategies are only used for controlling the evaluation degree of expressions. The parallel behaviour is defined by Eden constructs. In this paper, we have justified the design decisions underlying the parallel functional language Eden. We have discussed several design alternatives and shown how the distribution of work and data can explicitly be controlled in Eden programs. A more formal semantic treatment is necessary to analyze these features in more detail. We hope to take profit from the work presented in [1] and [6] for that purpose.

REFERENCES

[1] C. Baker-Finch, D. J. King, and P. W. Trinder. An Operational Semantics for Parallel Lazy Evaluation. In *ACM Int. Conf. on Functional Programming*, 2000.

[2] S. Breitinger, U. Klusik, and R. Loogen. From (sequential) Haskell to (parallel) Eden: An Implementation Point of View. In *International Symposium on Programming Languages: Implementations, Logics, Programs (PLILP)*, LNCS 1490. Springer, 1998.

[3] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden – Language Definition and Operational Semantics. Technical Report 96-10, Philipps-Universität Marburg, 1996.

[4] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden coordination model for distributed memory systems. In *High-Level Parallel Programming Models and Supportive Environments (HIPS)*. IEEE Press, 1997.

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

[6] M. Hidalgo-Herrero and Y. Ortega-Mallén. A Distributed Operational Semantics for a Parallel Functional Language. In *Proc. of 2nd Scottish Functional Programming Workshop*. St Andrews, 2000.

[7] U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Process Schemes in Eden – Efficient Parallel Programming With Low Effort. In *Workshop on the Implementation of Functional Languages*, Aachener Informatik-Berichte No. 00-7. Aachen, 2000.

[8] E. Nöcker, J. Smetsers, M. Eekelen, and M. J. Plasmeijer. Concurrent Clean. In *PARLE '91, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms*, LNCS 505. Springer, 1991.

[9] J. Peterson and K. Hammond (eds.). Report on the programming language Haskell: a non-strict, purely functional language, version 1.4. Technical Report YALEU/DCS/RR-1106, Yale University, 1997.

[10] The MPI Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, 1994.

[11] G. Tremblay and G. Gao. The impact of laziness on parallelism and the limits of strictness analysis. In A. P. W. Bohm and J. T. Feo, editors, *High Performance Functional Computing Proceedings*, pp. 119–133, Apr. 1995.

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

Contents