Philipps Universität Marburg

Eden
Parallel Functional Programming with Haskell

Rita Loogen

Philipps-Universität Marburg, Germany

Joint work with
Yolanda Ortega Mallén, Ricardo Peña
Alberto de la Encina, Mercedes Hildalgo Herrero, Christóbal Pareja,
Fernando Rubio, Lidia Sánchez-Gil, Clara Segura, Pablo Roldan Gomez
(Universidad Complutense de Madrid)

Jost Berthold, Silvia Breitinger, Mischa Dieterle, Thomas Horstmeyer,
Ulrike Klusik, Oleg Lobachev, Bernhard Pickenbrock, Steffen Priebe, Björn Struckmeier
(Philipps-Universität Marburg)

CEFP Budapest 2011

Marburg /Lahn

Overview

- Lectures I & II (Thursday)
  - Motivation
  - Basic Constructs
  - Case Study: Mergesort
  - Eden TV – The Eden Trace Viewer
  - Reducing communication costs
  - Parallel map implementations
  - Explicit Channel Management
  - The Remote Data Concept
  - Algorithmic Skeletons
    - Nested Workpools
    - Divide and Conquer

- Lecture III: Lab Session (Friday Morning)

- Lecture IV: Implementation
  - Layered Structure
  - Primitive Operations
  - The Eden Module
  - The Trans class
  - The PA monad
  - Process Handling
  - Remote Data

Materials

Materials

- **Lecture Notes**
- **Slides**
- **Example Programs (Case studies)**
- **Exercises**

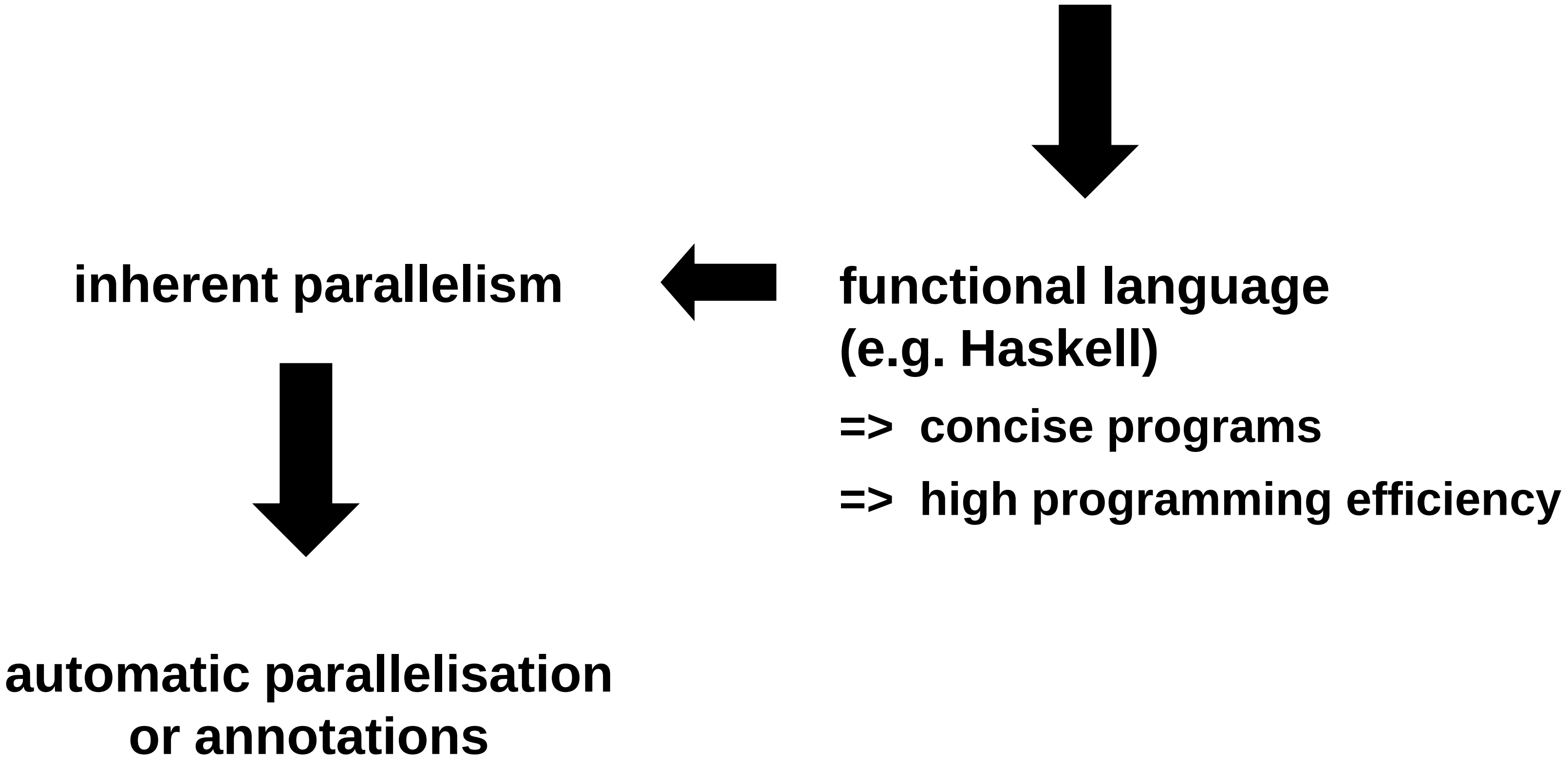
are provided via the Eden web page

www.informatik.uni-marburg.de/~eden

Navigate to CEFP!

Motivation

Our Goal



Parallel programming at a high level of abstraction

functional language
(e.g. Haskell)

=> concise programs

=> high programming efficiency

inherent parallelism

automatic parallelisation
or annotations

Our Approach

Parallel programming at a high level of abstraction

parallelism control

- » **explicit processes**
- » **implicit communication**
- » **distributed memory**
- » **…**

+

functional language
(e.g. Haskell)

=> concise programs

=> high programming efficiency

Eden = Haskell + Parallelism

www.informatik.uni-marburg.de/~eden

Basic Constructs

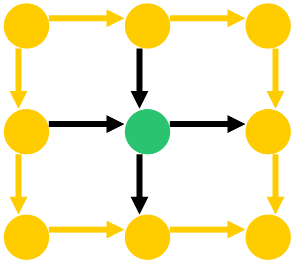
Eden

= Haskell + Coordination

parallel programming at a high level of abstraction

- process definition

```
process       ::    (Trans a, Trans b) =>  (a -> b) -> Process a b
gridProcess   =     process (\ (fromLeft,fromTop) ->
                                 let    ...    in   (toRight, toBottom))
```



process outputs computed by concurrent threads, lists sent as streams

- process instantiation

```
( # )                  ::     (Trans a, Trans b) =>  Process a b -> a -> b
(outEast, outSouth)   =   gridProcess   #   (inWest,inNorth)
```

Derived operators and functions

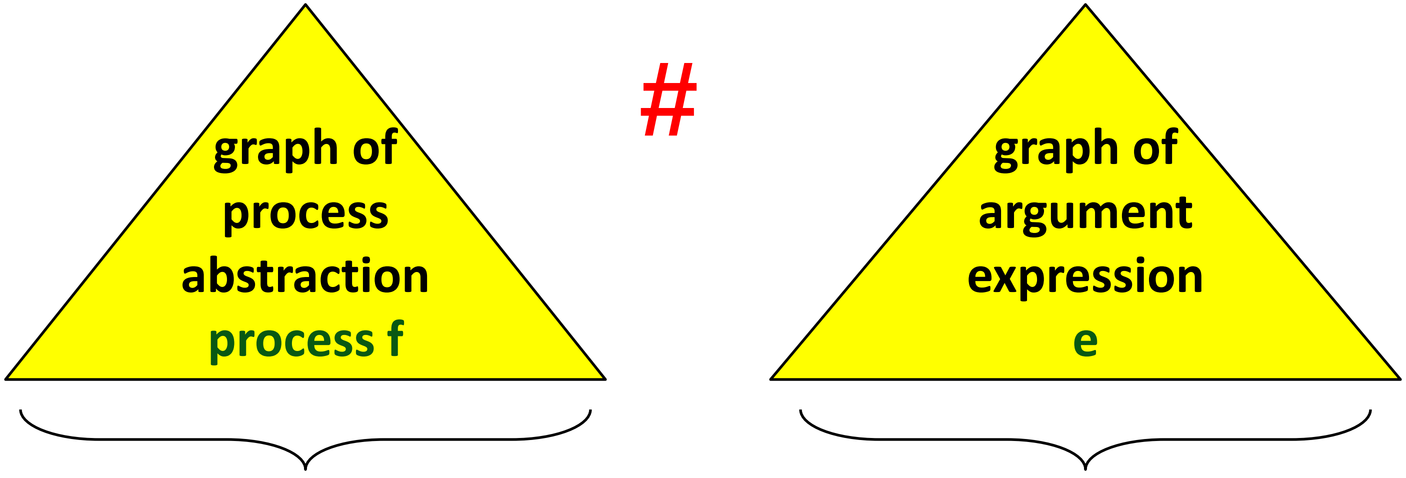
- **Parallel function application**
  - Often, process abstraction and instantiation are used in the following combination

```
($#)  :: (Trans a, Trans b) =>  (a -> b) ->  a  -> b
f $# x = process f # x   -- ($#) = (#) . process
```

- **Eager process creation**
  - Eager creation of a series of processes

```
spawn   ::  (Trans a, Trans b) =>
            [Process a b] -> [a] -> [b]
spawn    =  zipWith (#)  -- ignoring demand control

spawnF  ::  (Trans a, Trans b) =>
            [a -> b] -> [a] -> [b]
spawnF   =  spawn . (map process)
```

Evaluating f $# e



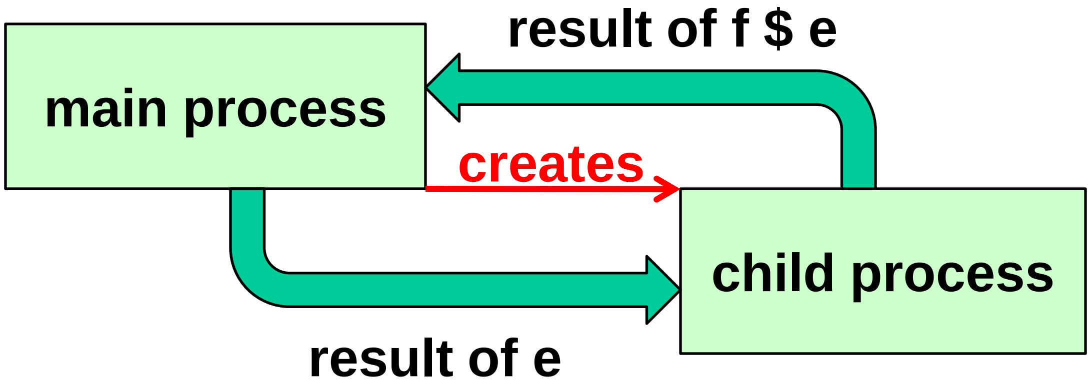
graph of process abstraction **process f**

#

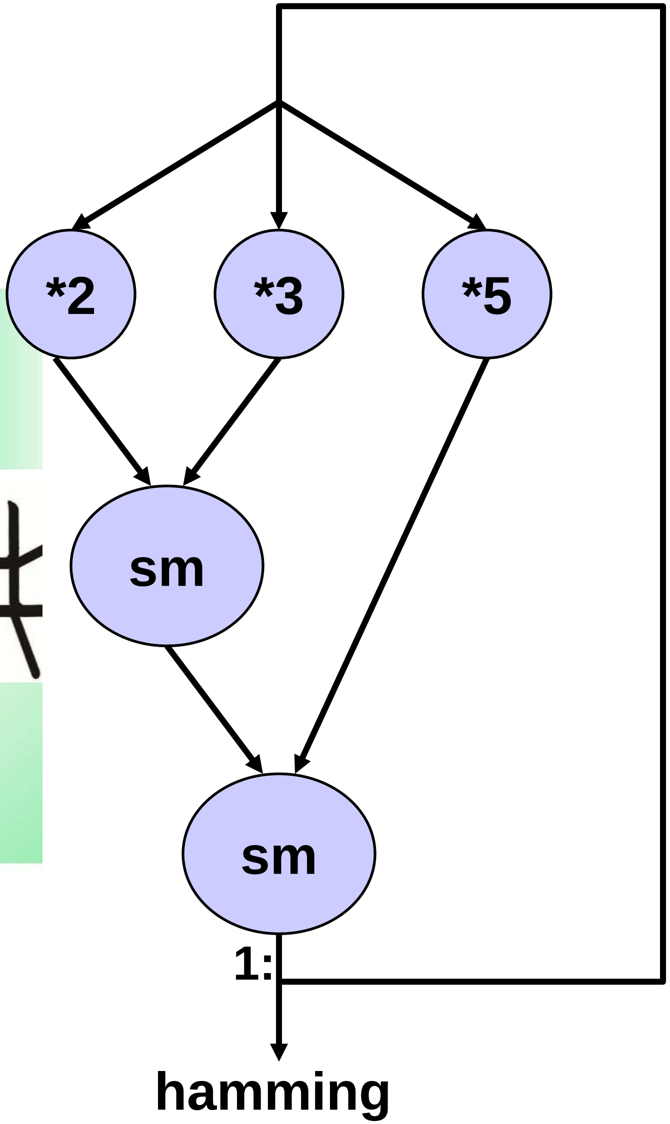
graph of argument expression **e**

will be evaluated
by new child process
on remote PE

will be evaluated
in parent process
by new concurrent thread
and sent to child process



Defining process nets
Example: Computing Hamming numbers



```
import Control.Parallel.Eden

hamming  :: [Int]
hamming
  = 1: sm ((uncurry sm) $#
             (map (*2) $# hamming,
              map (*3) $# hamming))
             (map (*5) $# hamming)


sm ::   [Int] -> [Int] -> [Int]
sm []      ys    = ys
sm xs      []    = xs
sm (x:xs) (y:ys)
   | x <  y      = x : sm xs (y:ys)
   | x == y      = x : sm xs  ys
   | otherwise   = y : sm (x:xs) ys
```

Questions about Semantics

- **simple denotational semantics**
  - **process abstraction -> lambda abstraction**
  - **process instantiation -> application**

  ➔ value/result of program, but no information about execution, parallelism degree, speedups /slowdowns

- **operational**
  1. **When will a process be created?**
     When will a process instantiation be evaluated?
  2. **To which degree will process in-/outputs be evaluated?**
     Weak head normal form or normal form or ...?
  3. **When will process in-/outputs be communicated?**

Answers

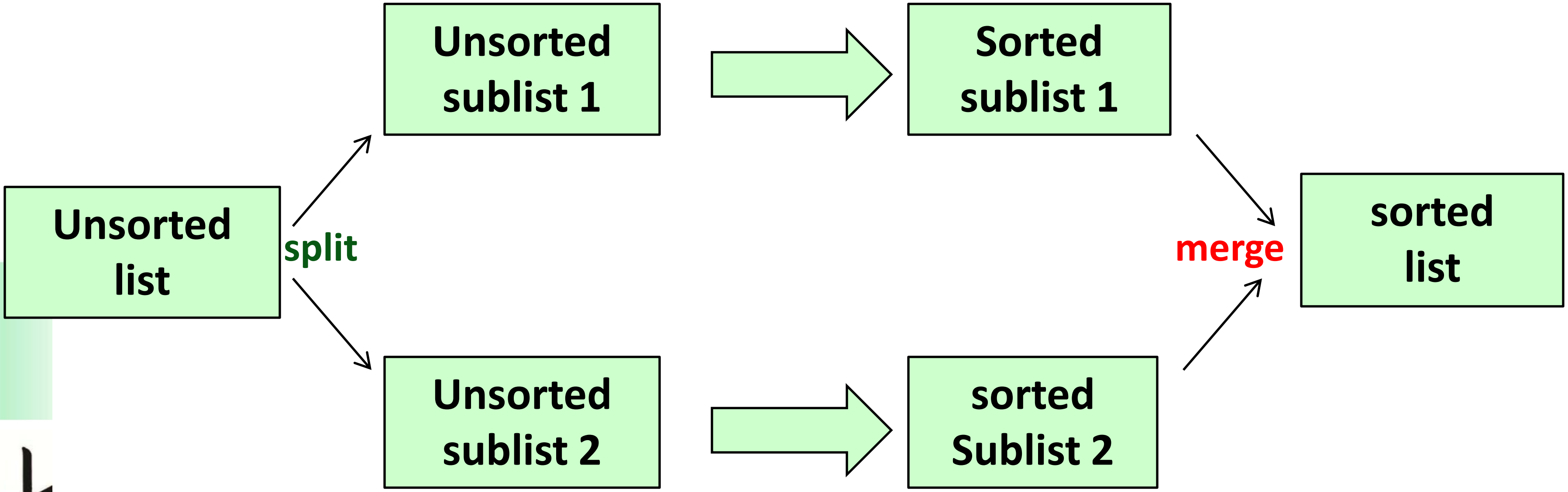
	Lazy Evaluation (Haskell)	Eden
1. When will a process be created? When will a process instantiation be evaluated?	only if and when its result is demanded	only if and when its result is demanded
2. To which degree will process in-/outputs be evaluated? Weak head normal form or normal form or ...?	WHNF (weak head normal form)	normal form
3. When will process in-/outputs be communicated?	only if demanded: request and answer messages necessary	eager (push) communication: values are communicated as soon as available

Lazy evaluation vs. Parallelism

- **Problem:** Lazy evaluation ==> distributed sequentiality
- Eden‘s approach:
  - eager process creation with spawn
  - eager communication:
    - normal form evaluation of all process outputs (by independent threads)
    - push communication, i.e. values are communicated as soon as available
  - explicit demand control by sequential strategies (Module Control.Seq):
    - rnf, rwhnf ... :: Strategy a
    - using :: a -> Strategy a -> a
    - pseq :: a -> b -> b (Module Control.Parallel)

Case Study: Merge Sort

Case Study: Merge Sort



Unsorted list

split

Unsorted sublist 1

Unsorted sublist 2

Sorted sublist 1

sorted Sublist 2

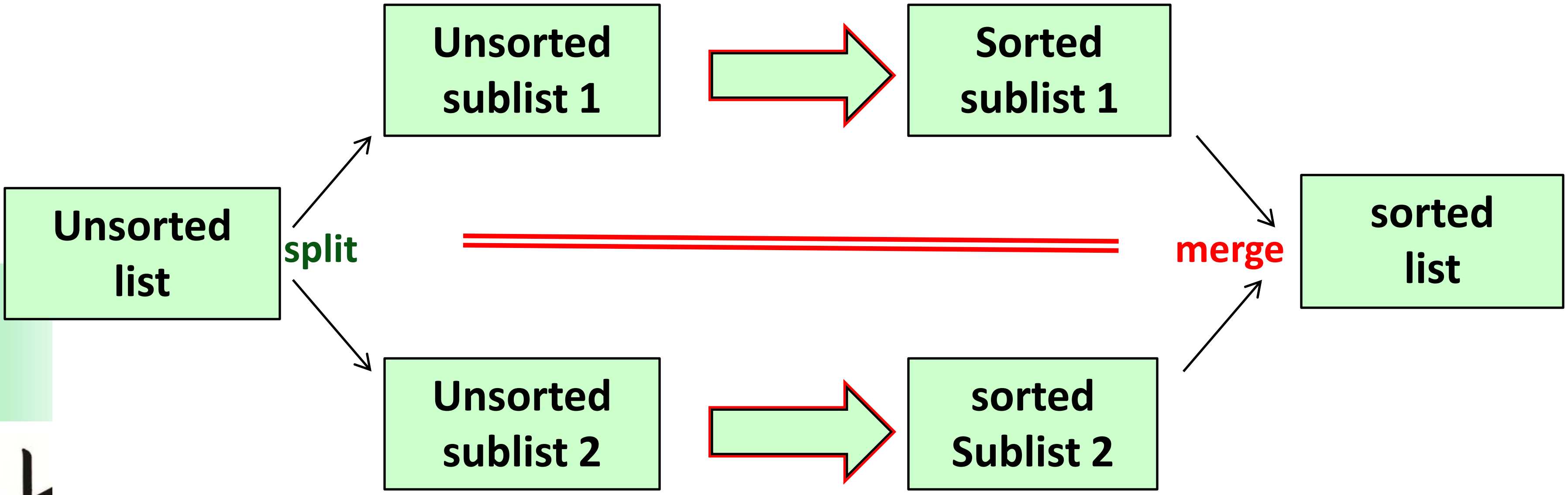
merge

sorted list

Haskell Code:

```
mergeSort          :: (Ord a, Show a) => [a] -> [a]
mergeSort []       =  []
mergeSort [x]      =  [x]
mergeSort xs       =  sortMerge (mergeSort xs1)  (mergeSort xs2)
                      where   [xs1,xs2] = unshuffle 2 xs
```

Example: Merge Sort parallel



Unsorted list

split

Unsorted sublist 1

Unsorted sublist 2

Sorted sublist 1

sorted Sublist 2

merge

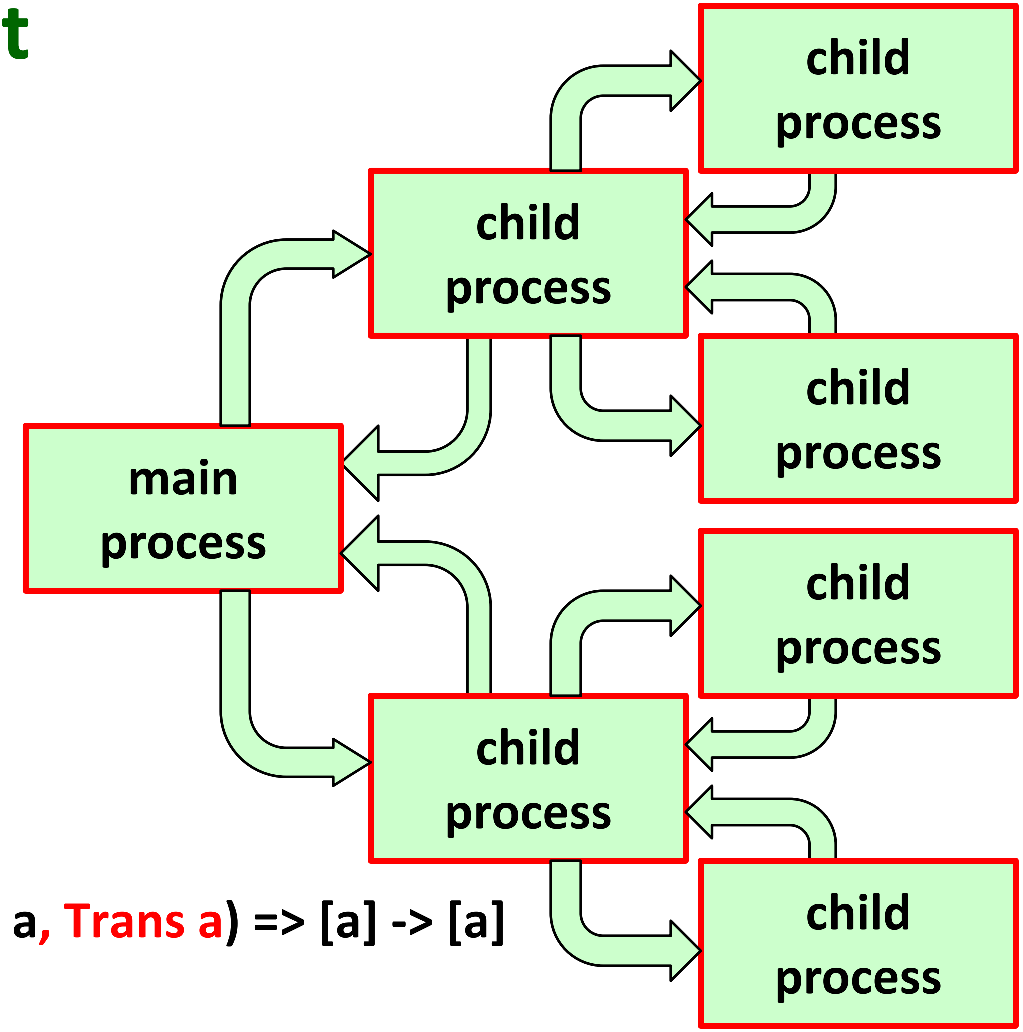
sorted list

Eden Code (simplest version):

```
parMergeSort      :: (Ord a, Show a, Trans a) => [a] -> [a]
parMergeSort []   =  []
parMergeSort [x] =  [x]
parMergeSort xs  =  sortMerge (parMergeSort  $#  xs1)  (parMergeSort  $#  xs2)
                          where   [xs1,xs2] = unshuffle 2 xs
```

Example: Merge Sort

Process net

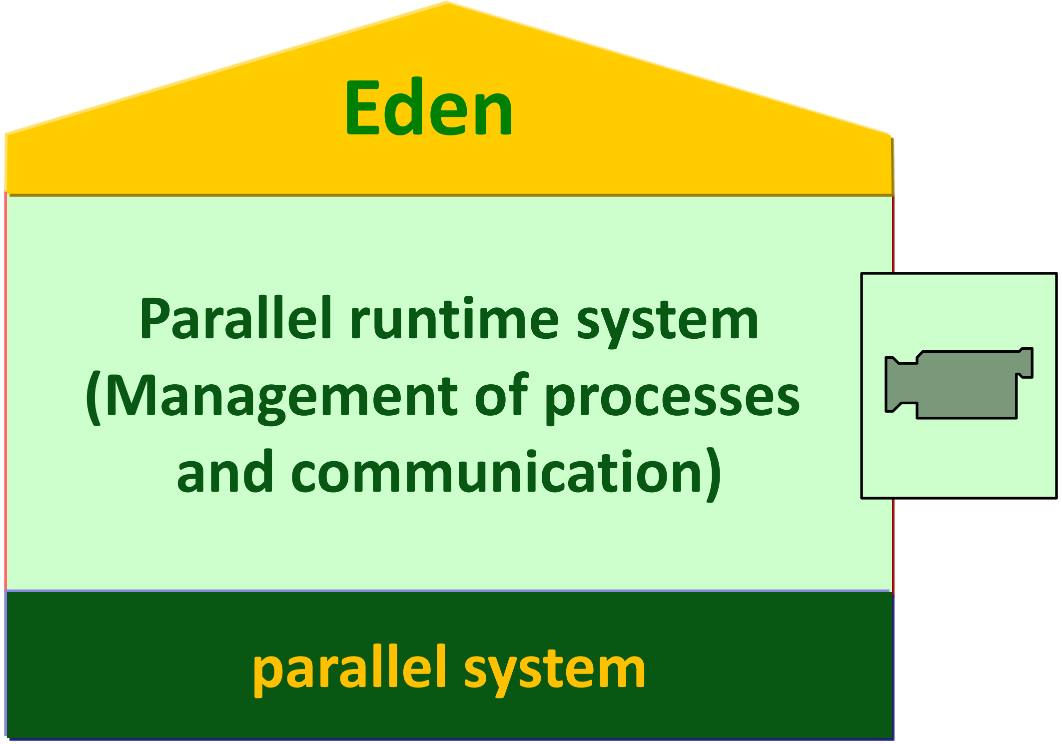


Eden Code (simplest version):

```
parMergeSort     :: (Ord a, Show a, Trans a) => [a] -> [a]
parMergeSort []  =  []
parMergeSort [x] =  [x]
parMergeSort xs  =  sortMerge (parMergeSort  $#  xs1)  (parMergeSort  $#  xs2)
                        where   [xs1,xs2] = unshuffle 2 xs
```

EdenTV: The Eden Trace Viewer Tool

The Eden-System



Eden

Parallel runtime system
(Management of processes
and communication)

parallel system

EdenTV

Compiling, Running, Analysing Eden Programs

Set up environment for Eden on Lab computers by calling

```
edenenv
```

Compile Eden programs with

```
ghc –parmpi   --make –O2 –eventlog   myprogram.hs
```

or

```
ghc –parpvm  --make –O2 –eventlog   myprogram.hs
```

If you use pvm, you first have to start it.
Provide pvmhosts or mpihosts file
Run compiled programs with

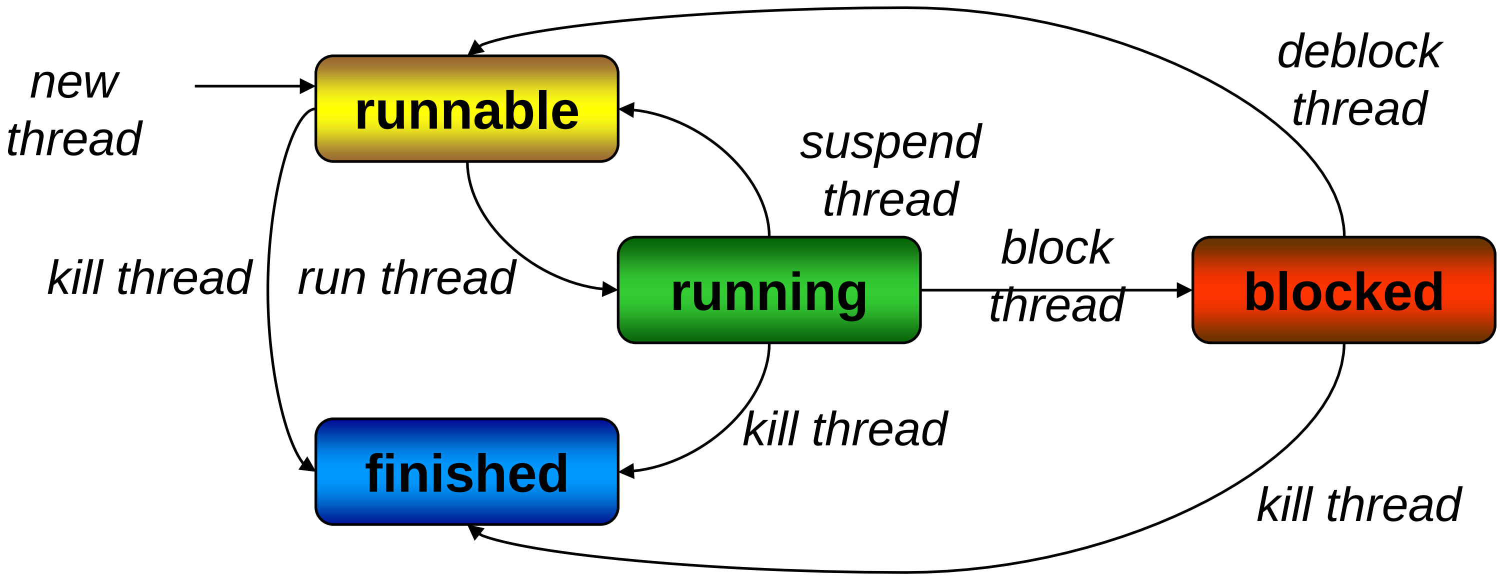
```
myprogram <parameters>   +RTS   –ls    -N<noPe>  -RTS
```

View activity profile (trace file) with

```
edentv    myprogram_..._-N4_-RTS.parevents
```

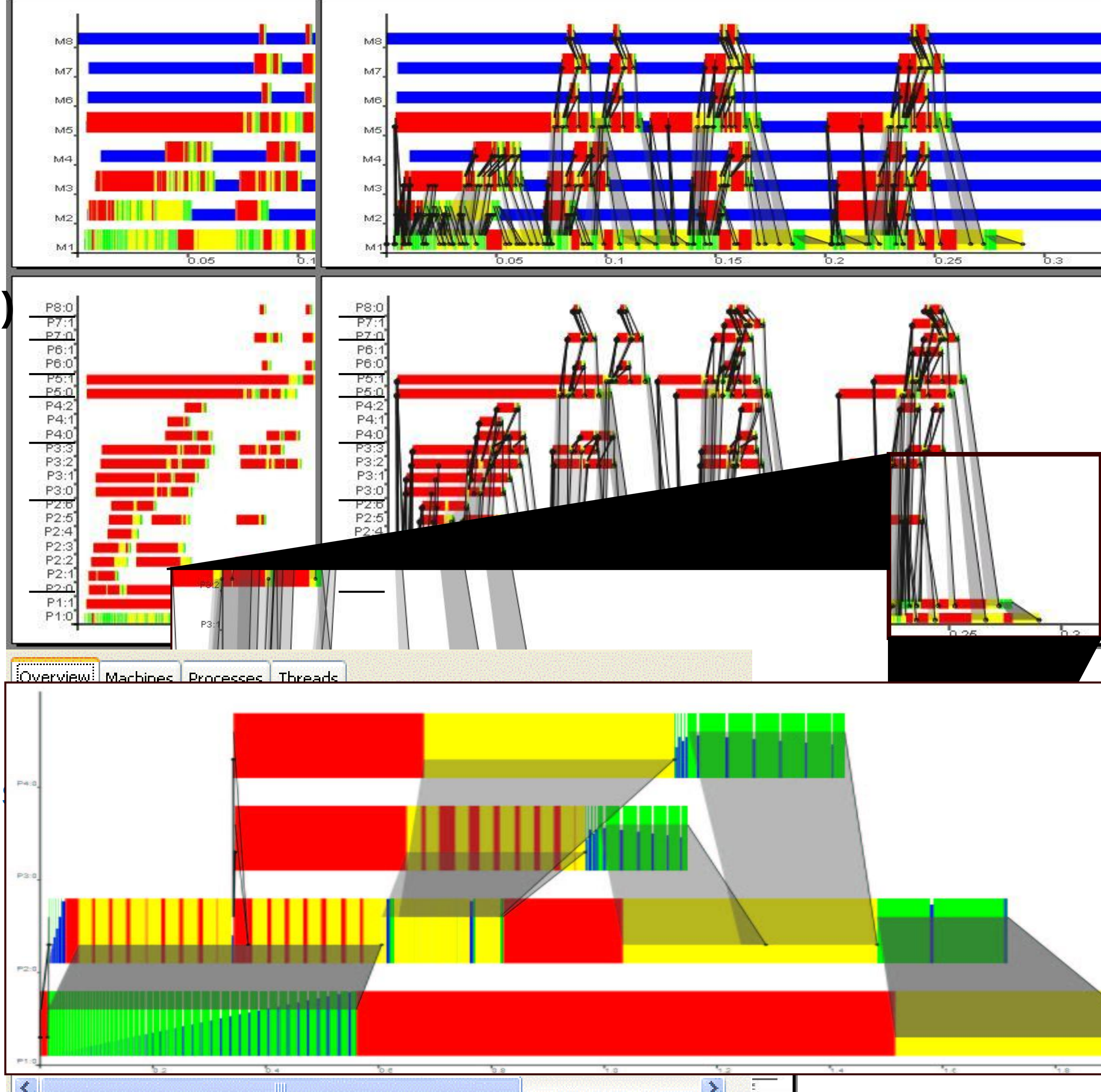
Eden Threads and Processes

- **An Eden process comprises several threads (one per output channel).**
- **Thread State Transition Diagram:**



EdenTV

- **Diagrams**:
  - Machines (PEs)
  - Processes
  - Threads
- **Message Overlays**
  - Machines
  - Processes
- **zooming**
- **message stream**
- **additional infos**
- **...**



EdenTV Demo

Case Study: Merge Sort continued

Example: Activity profile of parallel mergesort

<u>**Program run, length of input list: 1.000**</u>

<u>**Observation**</u>**:**

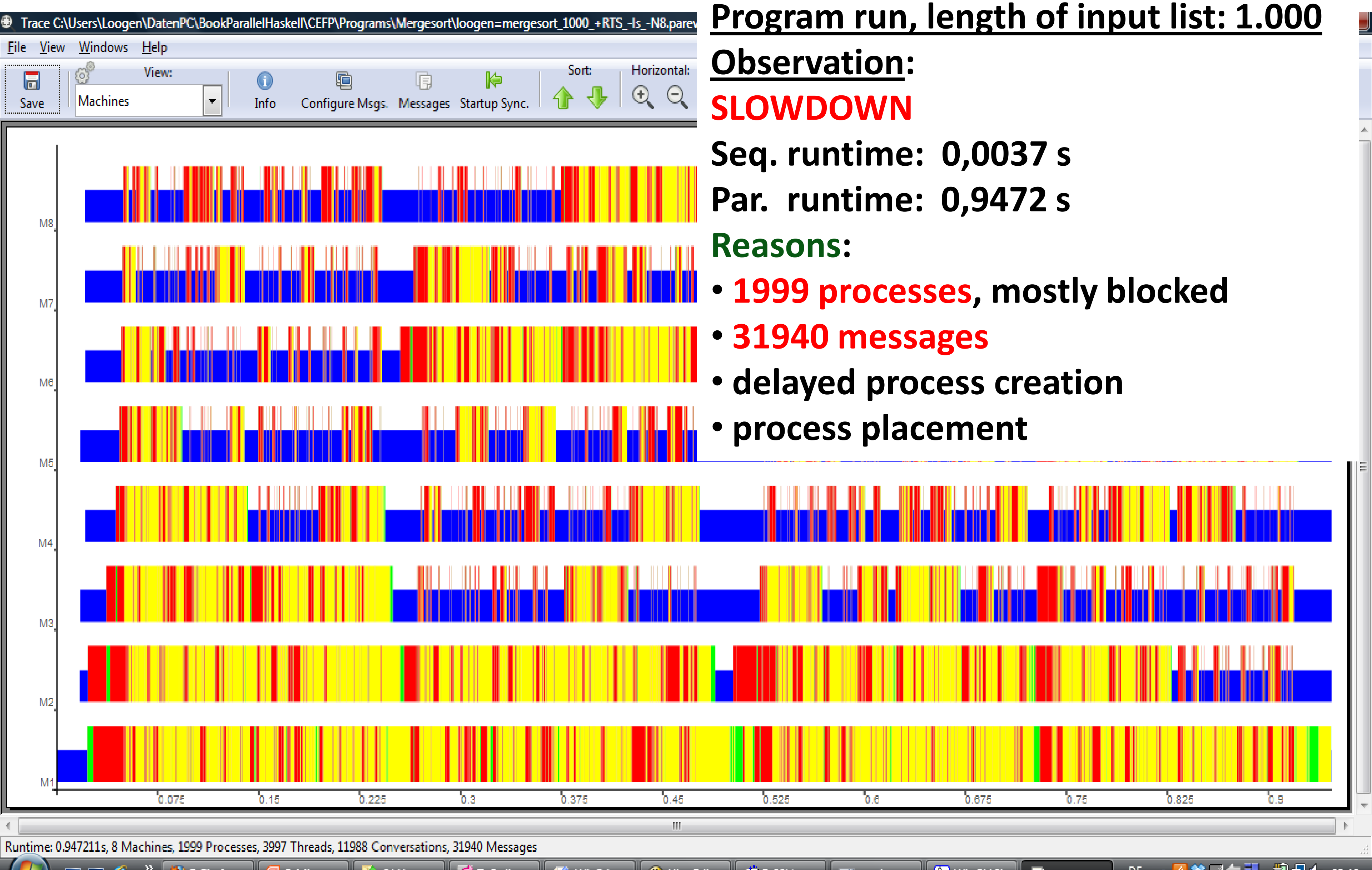
SLOWDOWN

Seq. runtime: 0,0037 s

Par. runtime: 0,9472 s

Reasons:

- **1999 processes, mostly blocked**
- **31940 messages**
- **delayed process creation**
- **process placement**



Trace C:\Users\Loogen\DatenPC\BookParallelHaskell\CEFP\Programs\Mergesort\loogen=mergesort_1000_+RTS_-ls_-N8.parev

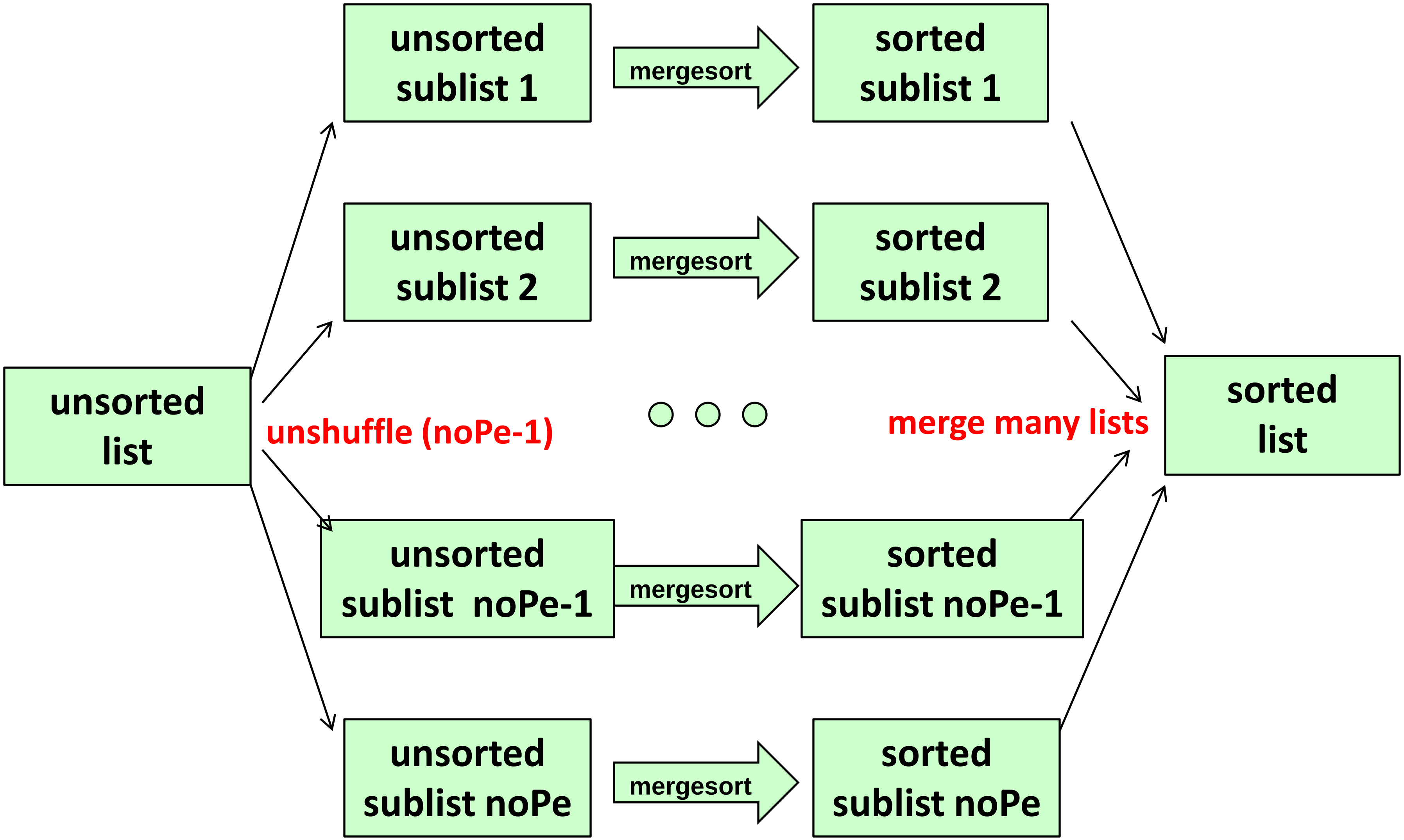
Runtime: 0.947211s, 8 Machines, 1999 Processes, 3997 Threads, 11988 Conversations, 31940 Messages

How can we improve our parallel mergesort? Here are some rules of thumb.

1. **Adapt the total number of processes to the number of available processor elements (PEs), in Eden: noPe :: Int**
2. **Use eager process creation functions spawn or spawnF.**
3. **By default, Eden places processes round robin on the available PEs. Try to distribute processes evenly over the PEs.**
4. **Avoid element-wise streaming if not necessary, e.g. by putting the list into some „box“ or by chunking it into bigger pieces.**

THINK PARALLEL!

Parallel Mergesort revisited



unsorted list

unshuffle (noPe-1)

unsorted sublist 1

mergesort

sorted sublist 1

unsorted sublist 2

mergesort

sorted sublist 2

unsorted sublist noPe-1

mergesort

sorted sublist noPe-1

unsorted sublist noPe

mergesort

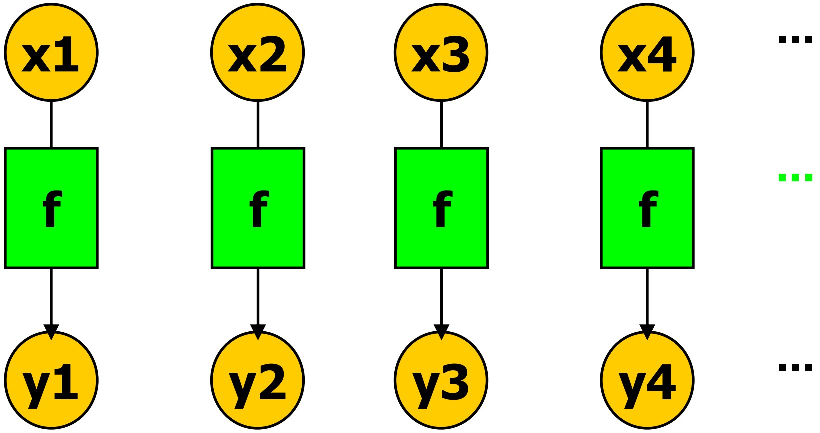
sorted sublist noPe

merge many lists

sorted list

A Simple Parallelisation of map

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



```
parMap ::  (Trans a, Trans b) =>
           (a -> b) -> [a] -> [b]
parMap f   = spawn (repeat (process f))
```

1 process per list element

Alternative Parallelisation of mergesort - 1st try

Eden Code:

```
par_ms  :: (Ord a, Show a, Trans a) => [a] -> [a]
par_ms  xs
  = head $ sms $ parMap mergeSort
                               (unshuffle (noPe-1) xs))
```

```
sms :: (NFData a, Ord a) => [[a]] -> [[a]]
sms []            = []
sms xss@[xs]      = xss
sms (xs1:xs2:xss) = sms (sortMerge xs1 xs2) (sms xss)
```

→ Total number of processes = noPe

→ eagerly created processes

→ round robin placement leads to 1 process per PE

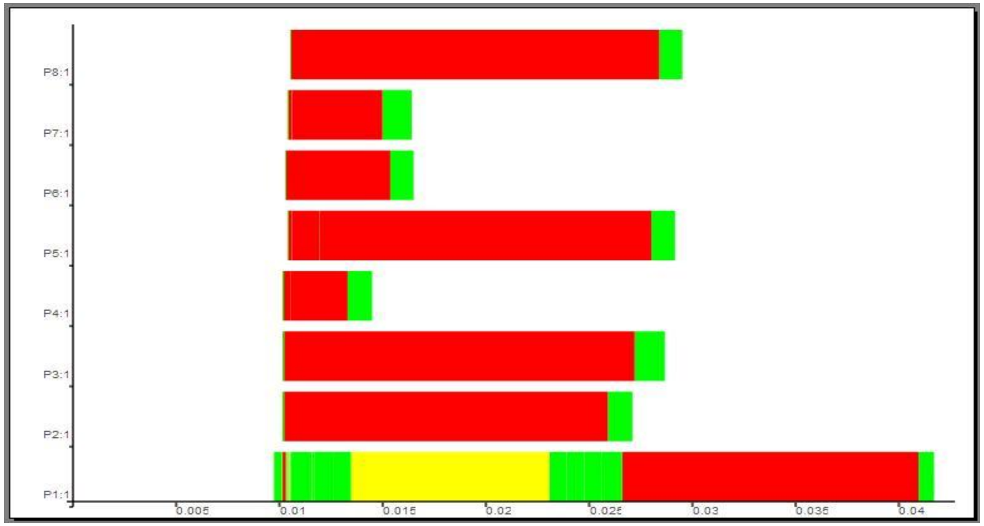
but maybe still too many messages

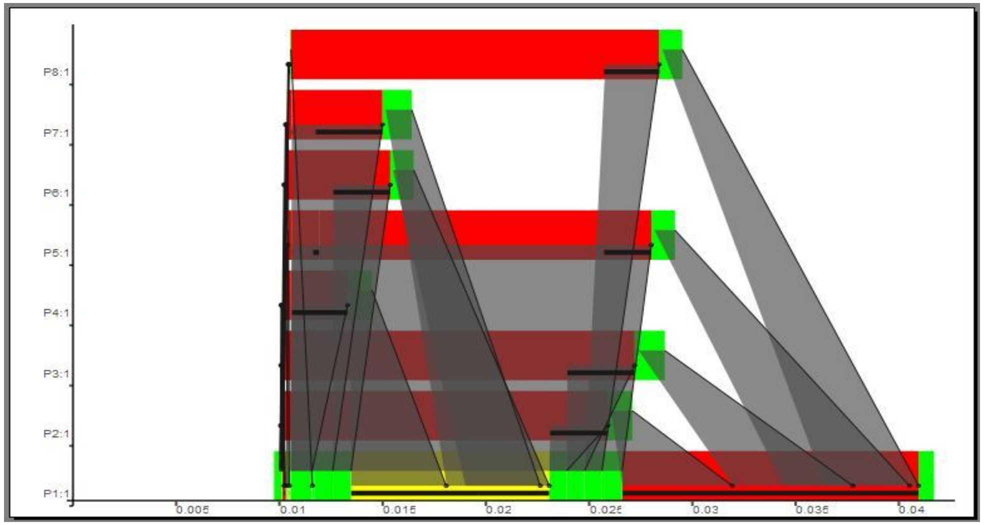
Resulting Activity Profile (Processes/Machine View)

<u>Previous results for input size 1000</u>

Seq. runtime: 0,0037 s

Par. runtime: 0,9472 s





- **Input size 1.000**
- **seq. runtime: 0,0037**
- **par. runtime: 0,0427 s**

- **8 Pes, 8 processes, 15 threads**
- **2042 messages**

Much better, but still

SLOWDOWN

Reason:

Indeed too many messages

Reducing Communication Costs

Reducing Number of Messages by Chunking Streams

Split a list (stream) into chunks:

```
chunk :: Int -> [a] -> [[a]]
chunk size [] = []
chunk size xs = ys : chunk size zs
  where (ys,zs) = splitAt size xs
```

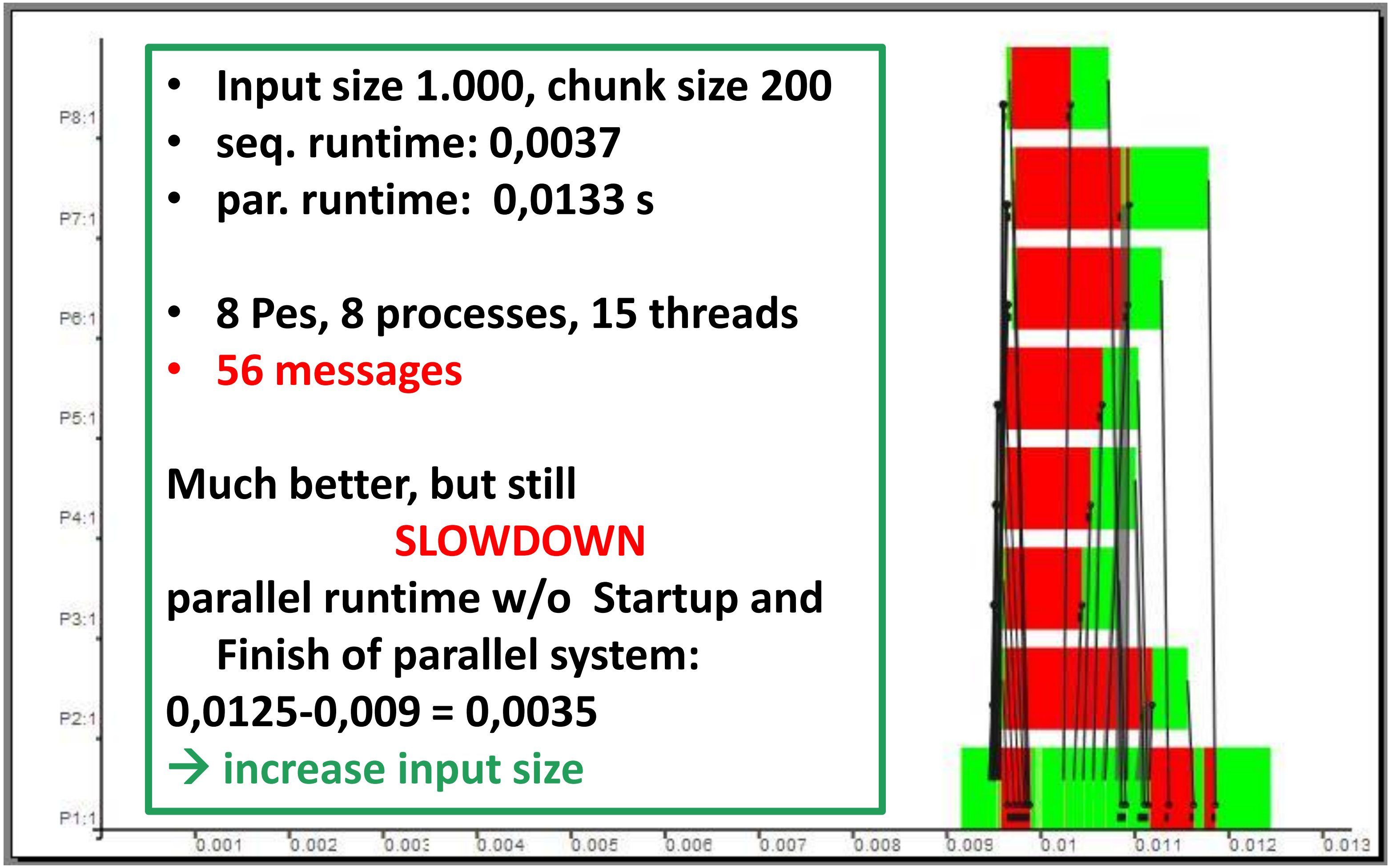
Combine with parallel map-implementation of mergesort:

```
par_ms_c  :: (Ord a, Show a, Trans a) =>
             Int ->        -- chunk size
             [a] -> [a]
par_ms_c size xs
  = head $ sms $ map concat $
     parMap ((chunk size)  .  mergeSort . concat)
            (map (chunk size)(unshuffle (noPe-1) xs)))
```

Resulting Activity Profile (Processes/Machine View)

Previous results for input size 1000

Seq. runtime: 0,0037 s
Par. runtime I: 0,9472 s
Par. runtime II: 0,0427 s



- Input size 1.000, chunk size 200
- seq. runtime: 0,0037
- par. runtime: 0,0133 s

- 8 Pes, 8 processes, 15 threads
- 56 messages

Much better, but still

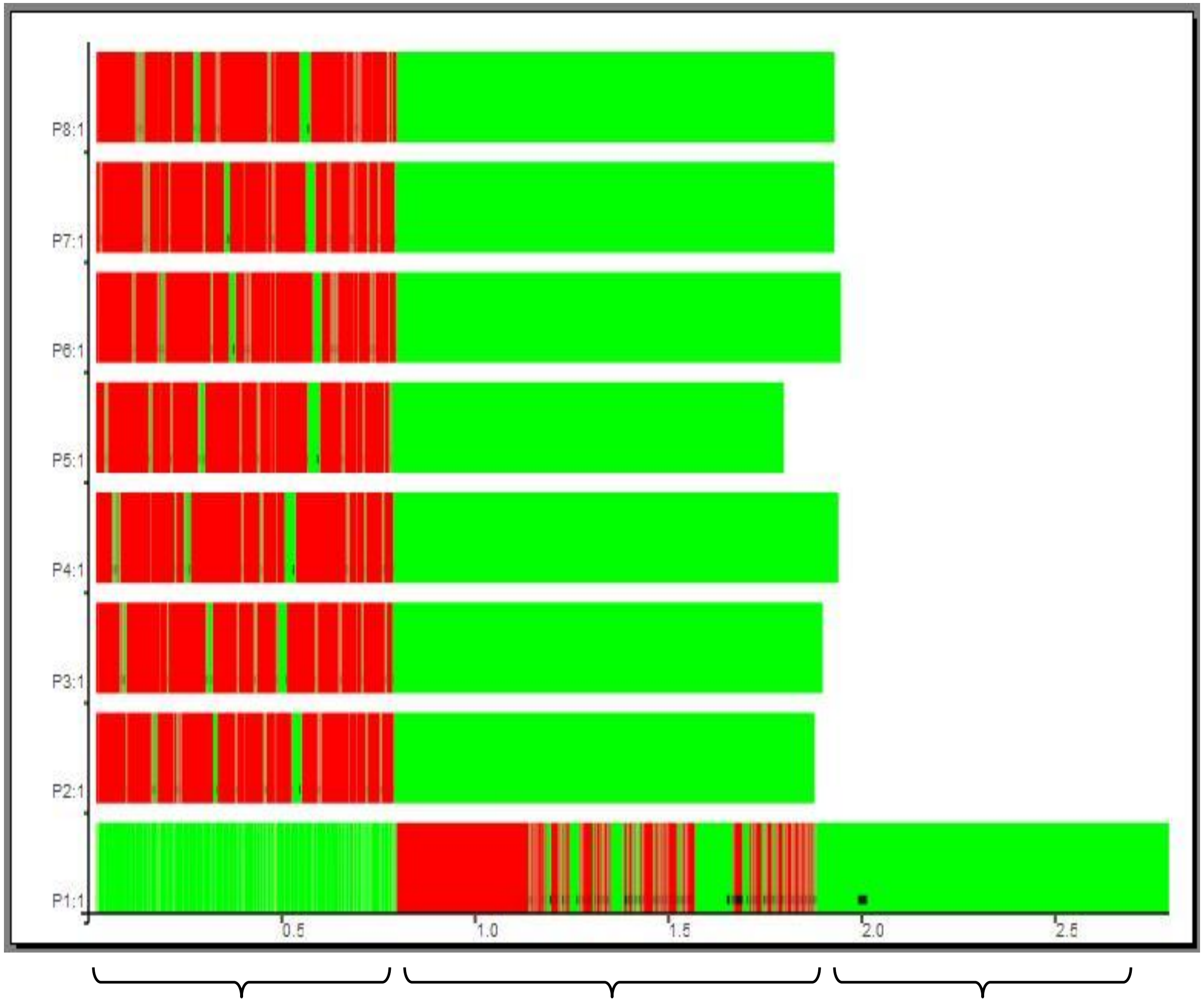
SLOWDOWN

parallel runtime w/o Startup and Finish of parallel system:

0,0125-0,009 = 0,0035

→ increase input size

Activity Profile for Input Size 1.000.000



unshuffle map mergesort merge

- Input size 1.000.000
- Chunk size 1000
- seq. runtime: 7,287 s
- par. runtime: 2,795 s

- 8 Pes, 8 processes, 15 threads
- 2044 messages

→ speedup of 2.6 on 8 PE

Further improvement

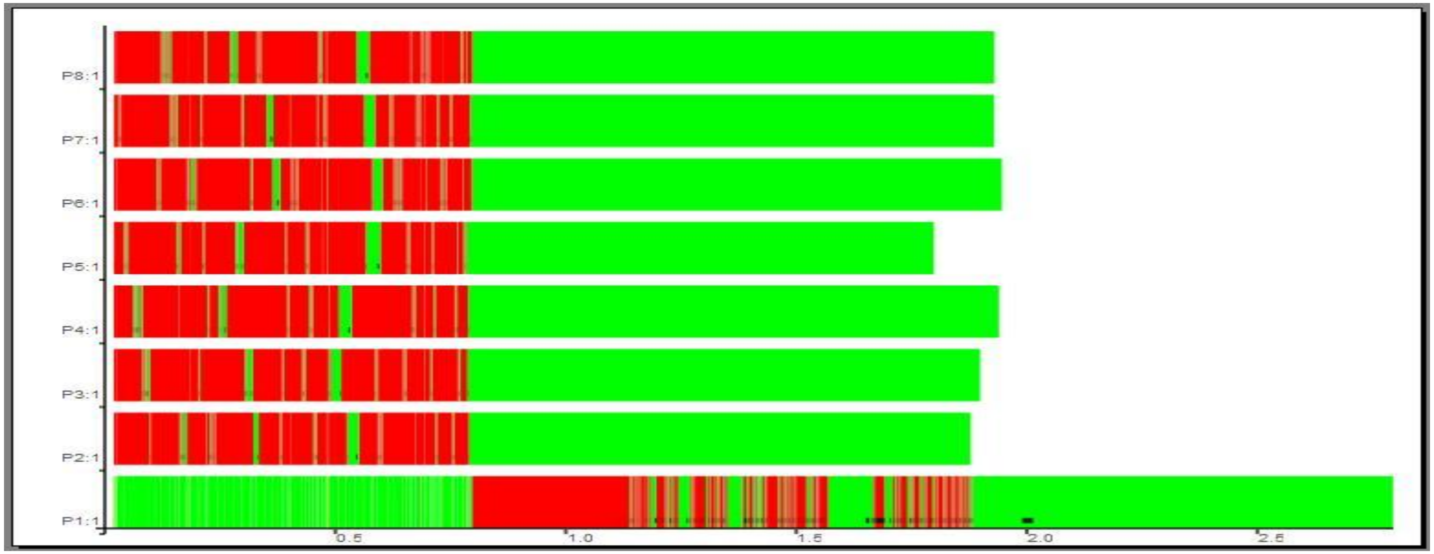
Idea: Remove input list distribution by local sublist selection:

```
par_ms_c  :: (Ord a, Show a, Trans a) =>
             Int -> [a] -> [a]
par_ms_c size xs
  = head $ sms $ map concat $
     parMap ((chunk size) .  mergeSort . concat)
            (map (chunk size)(unshuffle (noPe-1) xs)))
```

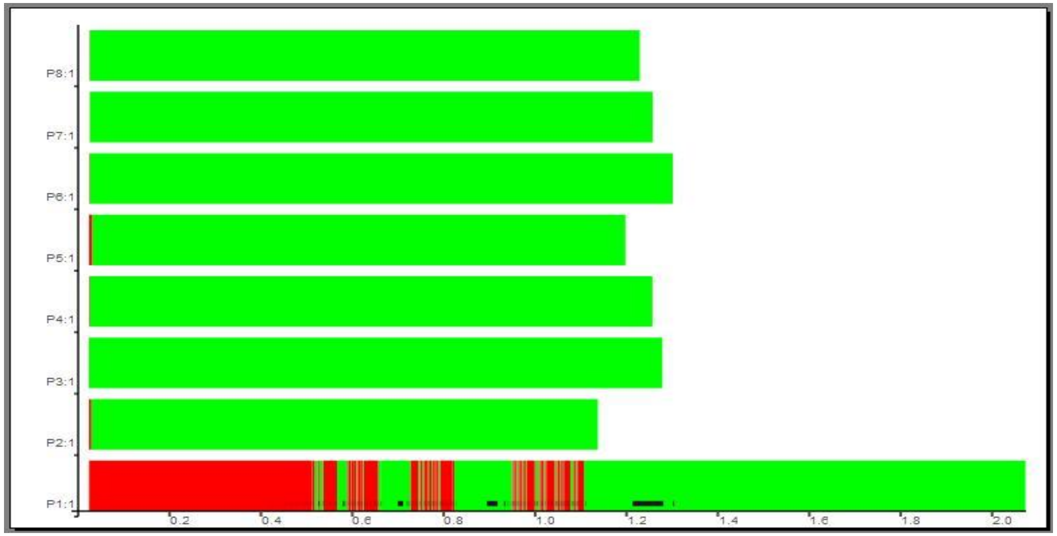
→

```
par_ms  :: (Ord a, Show a, Trans a) =>
             Int -> [a] -> [a]
par_ms_b size xs
  = head $ sms $ map concat $
     parMap (\ i -> (chunk size (mergeSort
                 ((unshuffle (noPe-1) xs)!!i))))
            [0..noPe-2]
```

Corresponding Activity Profiles



→

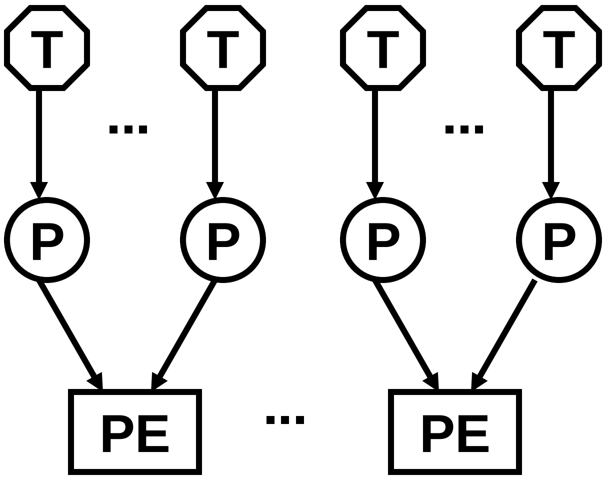


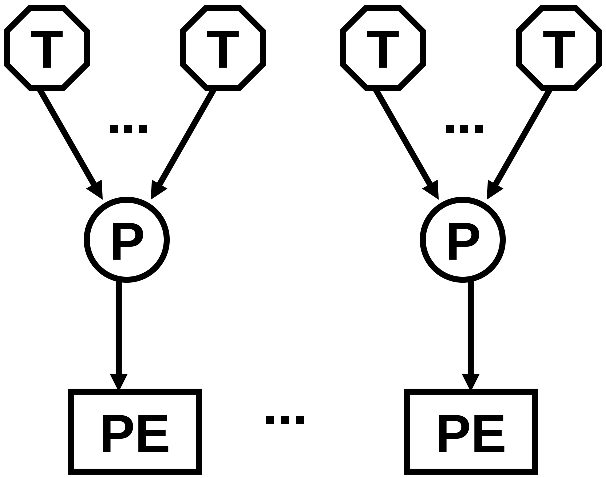
- **Input size 1.000.000**
- **Chunk size 1000**
- **seq. runtime: 7,287 s**
- **par. runtime: 2,795 s**
- **new par. runtime: 2.074 s**
- **8 Pes, 8 processes, 15 threads**
- **1036 messages**

→ speedup of 3.5 on 8 PEs

Parallel map implementations

Parallel map implementations: parMap vs farm





```
parMap   :: (Trans a, Trans b) =>
            (a -> b)  -> [a] -> [b]
parMap  f xs
   =   spawn (repeat  (process f)) xs
```

```
farm  ::  (Trans a, Trans b) =>
          ([a] -> [[a]]) -> ([[b]] -> [b]) ->
          (a -> b) -> [a] -> [b]
farm distribute combine f xs
     = combine (parMap (map f)
                       (distribute xs))
```

Process farms

```
farm :: (Trans a, Trans b) =>
        ([a] -> [[a]]) ->    -- distribute
        ([[b]] -> [b]) ->    -- combine
        (a->b) -> [a] -> [b]
farm distribute combine f xs
  =  combine . (parMap (map f)) . distribute
```

1 process
per sub-tasklist
with static
task distribution

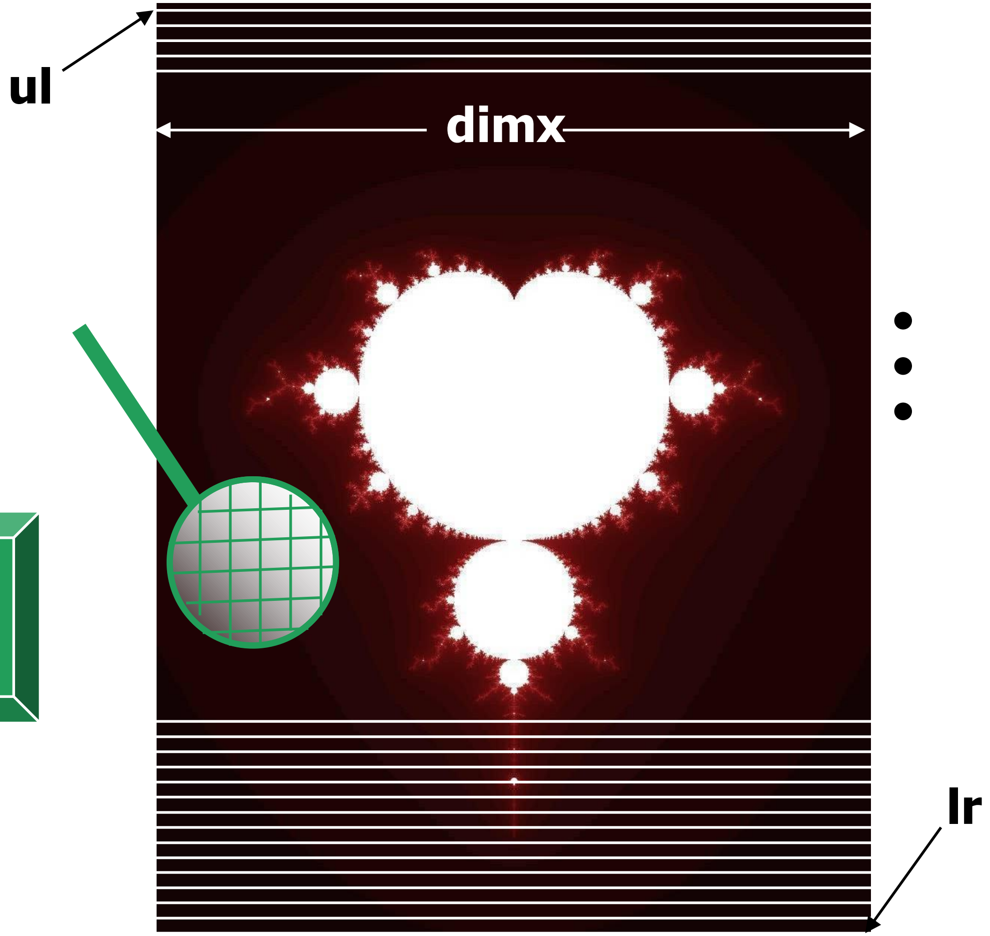
`Choose e.g.`

- `distribute = unshuffle  noPe / combine = shuffle`
- `distribute = splitIntoN noPe / combine = concat`

1 process
per PE
with static
task distribution

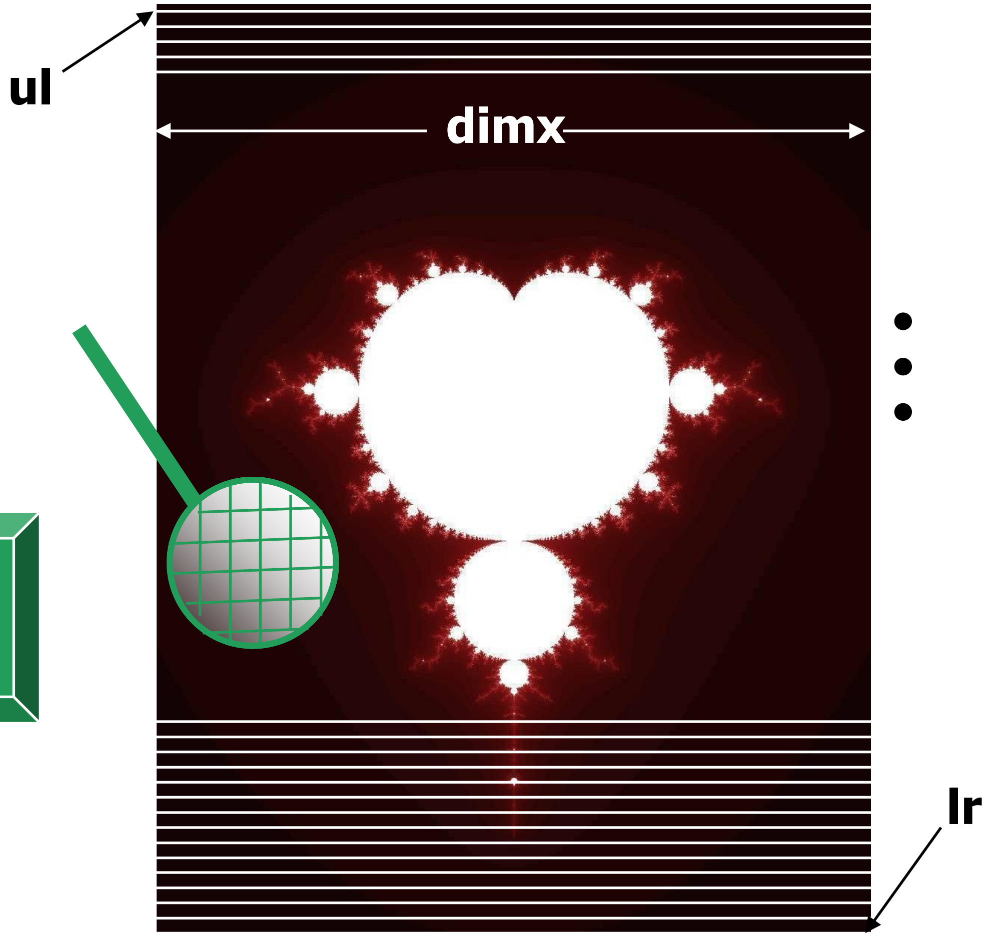
Example: Functional Program for Mandelbrot Sets

Idea: parallel computation of lines



```
image :: Double -> Complex Double -> Complex Double -> Integer -> String
image threshold ul lr dimx
 = header ++ (concat $ map xy2col lines  )
 where
   xy2col ::[Complex Double] ->  String
   xy2col line    = concatMap (rgb.(iter threshold (0.0 :+ 0.0) 0)) line
  (dimy, lines)   = coord ul lr dimx
```

Example: Parallel Functional Program for Mandelbrot Sets



Idea: parallel computation of lines

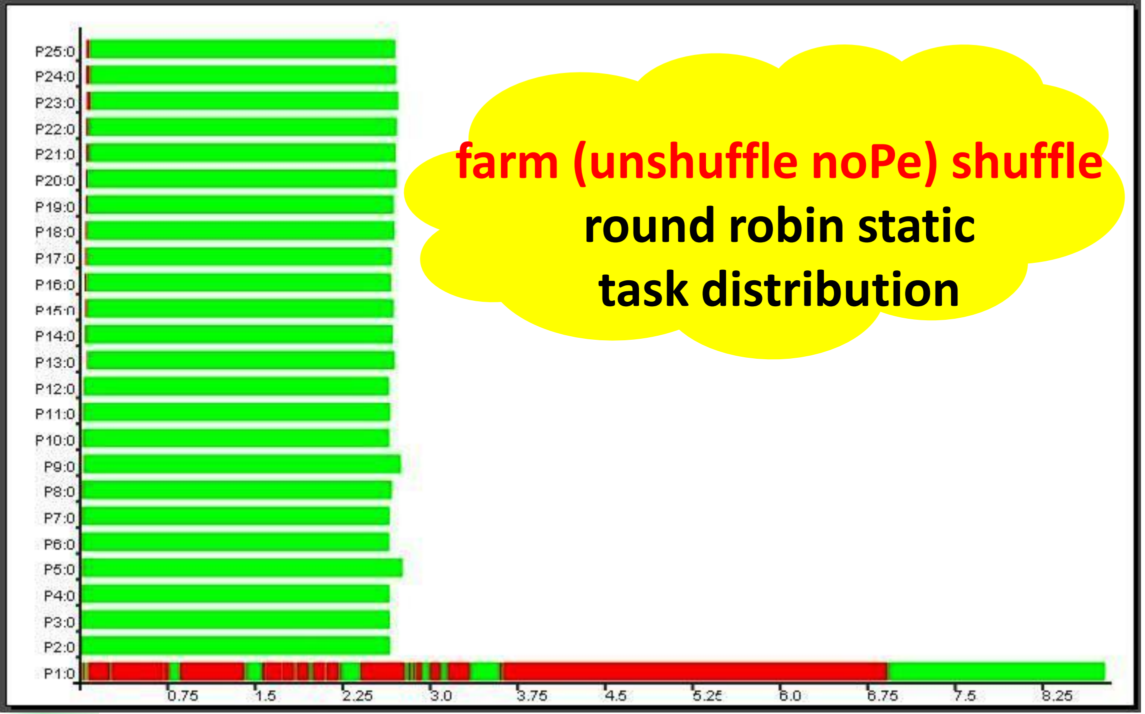
```
image :: Double -> Complex Double -> Complex Double -> Integer -> String
image threshold ul lr dimx
 = header ++ (concat $ map xy2col lines)
 where
   xy2col ::[Complex Double] ->  String
   xy2col line    = concatMap (rgb.(iter threshold (0.0 :+ 0.0) 0)) line
  (dimy, lines)   = coord ul lr dimx
```

Replace map by
farm (unshuffle noPe) shuffle
or farmB (splitIntoN noPe) concat

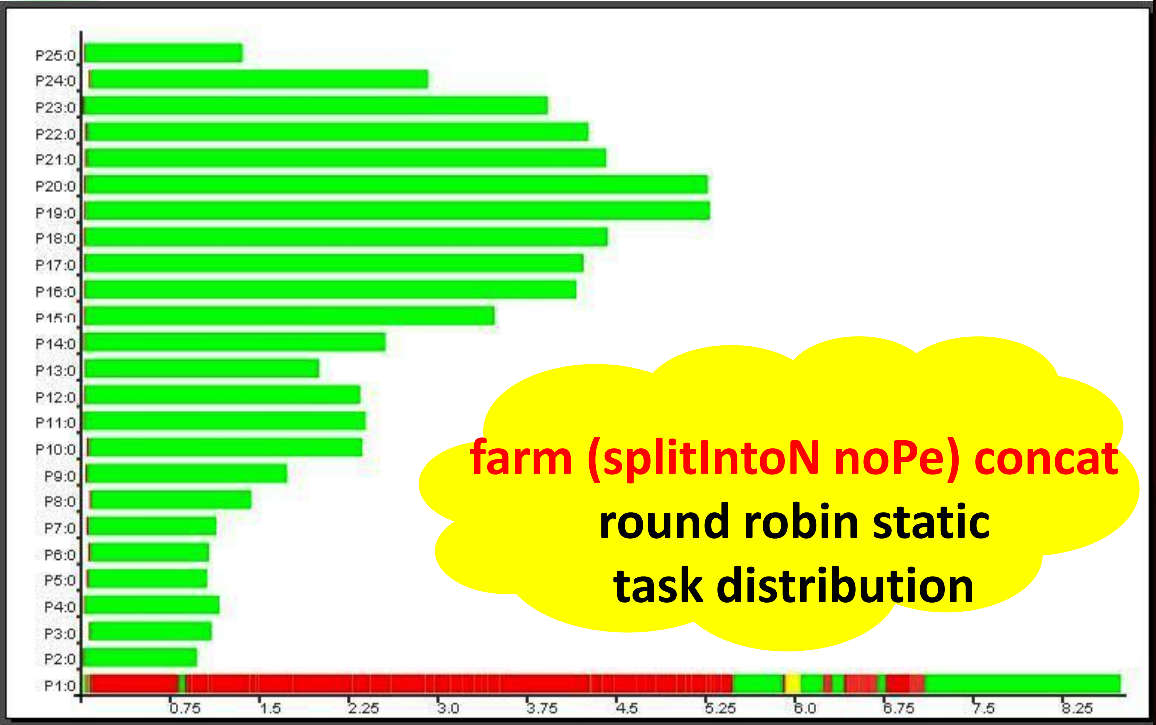
Mandelbrot Traces

Problem size: 2000 x 2000

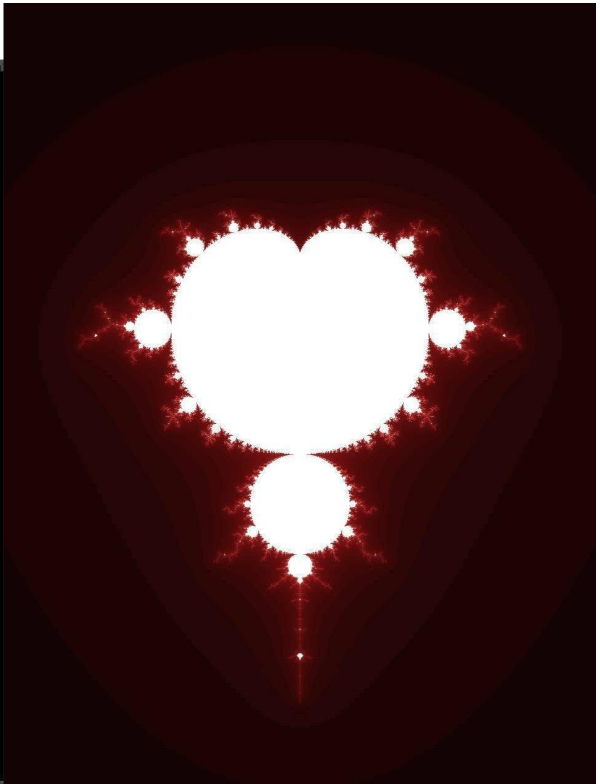
Platform: Beowulf cluster
Heriot-Watt-University, Edinburgh
(32 Intel P4-SMP nodes @ 3 GHz 512MB RAM, Fast Ethernet)



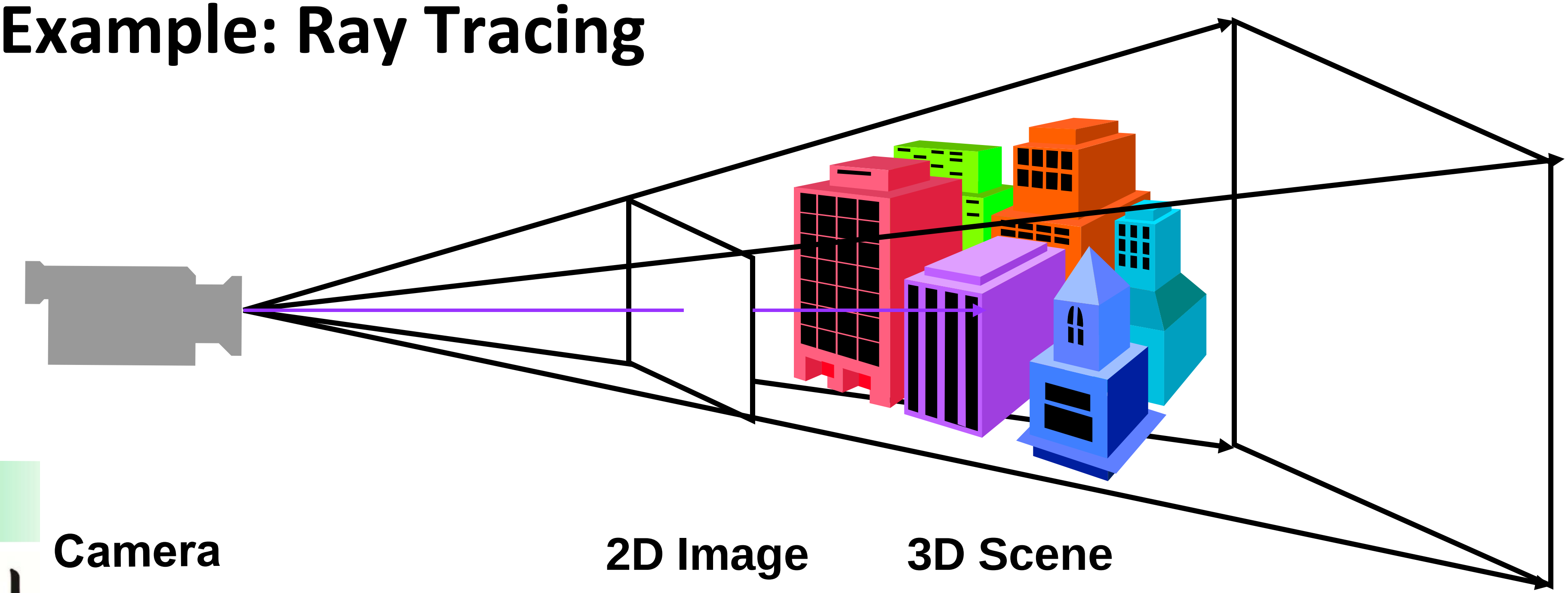
farm (unshuffle noPe) shuffle
round robin static task distribution



farm (splitIntoN noPe) concat
round robin static task distribution



Example: Ray Tracing



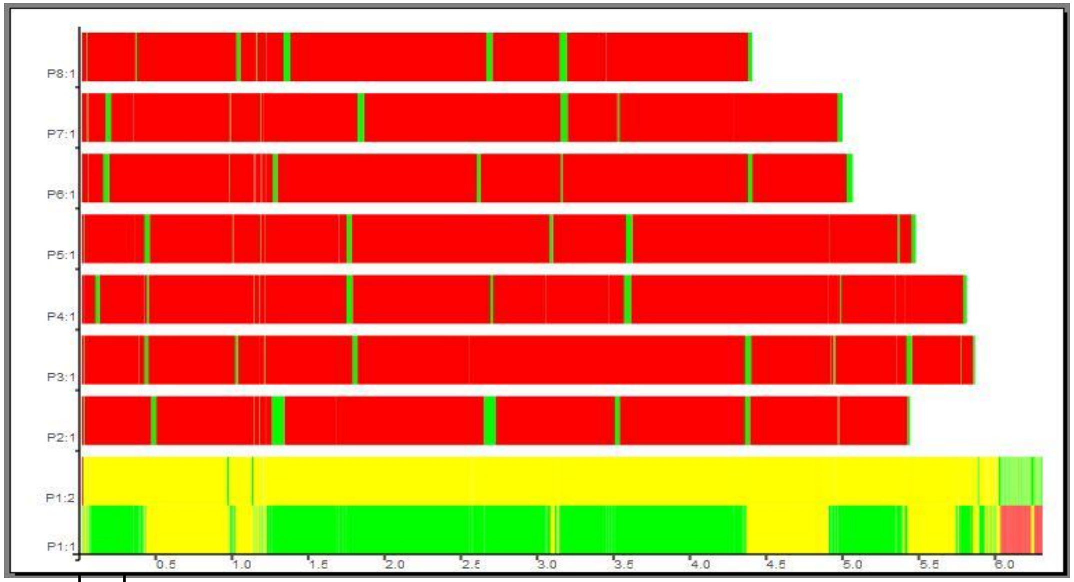
```
rayTrace  ::  Size -> CamPos -> [Object] ->  [Impact]
rayTrace  size  cameraPos scene   = findImpacts allRays scene
  where allRays = generateRays size cameraPos

findImpacts ::  [Ray] -> [Object] -> [Impact]
findImpacts rays objs = map (firstImpact objs) rays
```

Reducing Communication Costs by Chunking

Combine chunking with parallel map-implementation:

```
chunkMap :: Int  -> (([a] -> [b]) -> ([[a]] -> [[b]]))
                 -> (a -> b) -> [a] -> [b]
chunkMap size mapscheme f xs
  = concat (mapscheme (map f) (chunk size xs))
```

Raytracer Example:
Element-wise Streaming vs Chunking



Input size 250

Runtime: 6,311 s

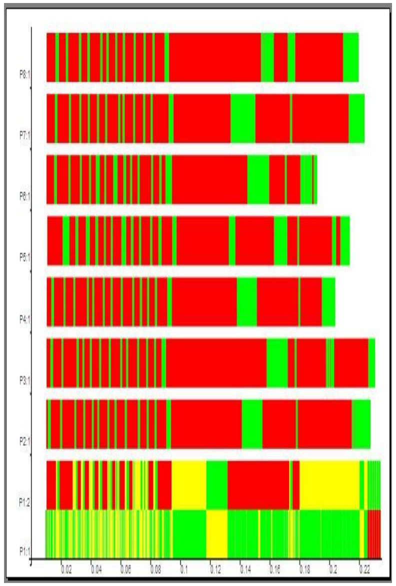
8 PEs

9 processes

17 threads

48 conversations

125048 messages



Input size 250

Chunk size 500

Runtime: 0,235 s

8 PEs

9 processes

17 threads

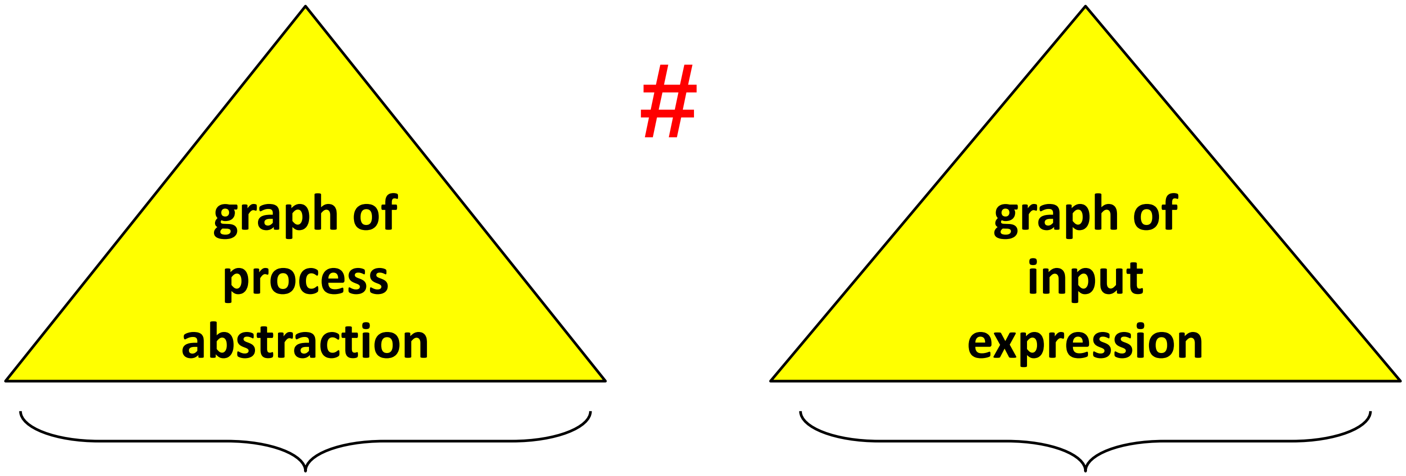
48 conversations

548 messages

Communication vs Parameter Passing

Process inputs

- **can be communicated:** `f $# inp`
- **can be passed as parameter** `(\ () -> f inp) $# ()`
  () is dummy process input



graph of process abstraction

#

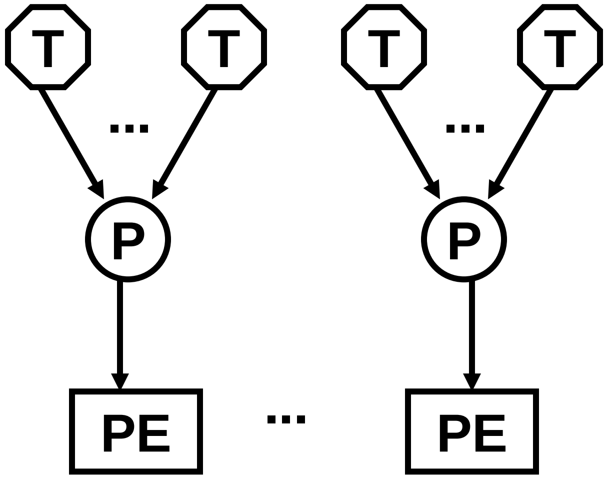
graph of input expression

**will be packed (serialised)
and sent to remote PE
where child process is created
to evaluate this expression**

**will be evaluated in parent process
by concurrent thread
and then sent to child process**

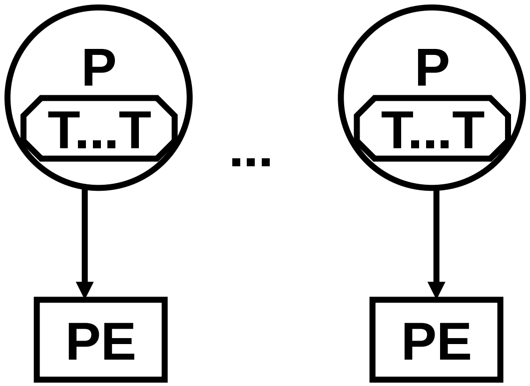
Farm vs Offline Farm

Farm



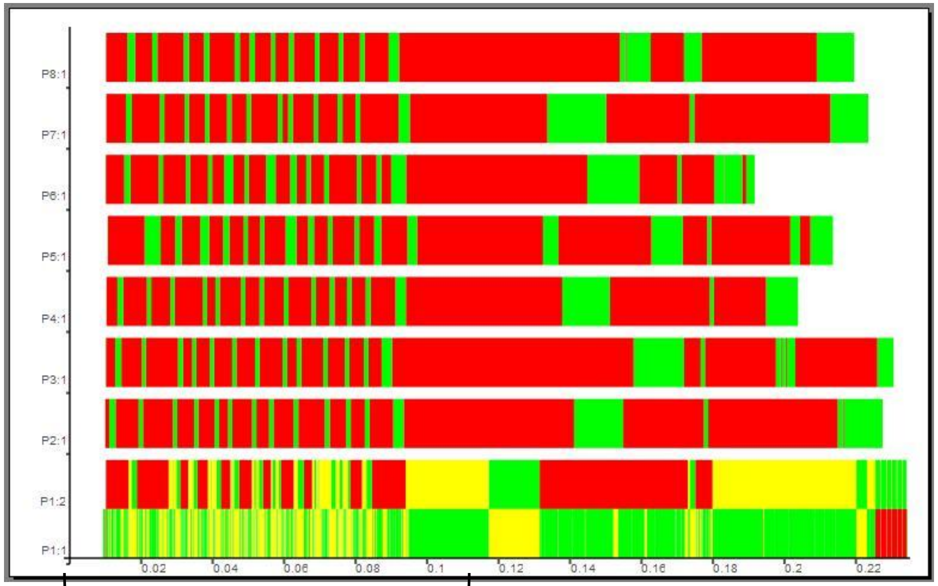
```
farm ::  (Trans a, Trans b) =>
         ([a] -> [[a]]) -> ([[b]] -> [b]) ->
         (a -> b) -> [a] -> [b]
farm distribute combine f xs
   = combine (parMap (map f)
                          (distribute xs))
```

Offline Farm

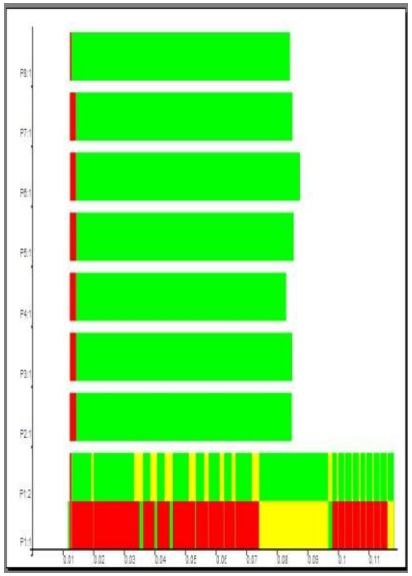


```
offlineFarm   :: (Trans a, Trans b) =>
          ([a] -> [[a]]) -> ([[b]] -> [b]) ->
          (a -> b) -> [a] -> [b]
offlineFarm  distribute combine  f xs
  =  combine $
       spawn
         (map (rfi (map f)) (distribute xs) )
         (repeat ())

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

Raytracer Example: Farm vs Offline Farm



Input size 250
Chunk size 500
Runtime: 0,235 s
8 PEs
9 processes
17 threads
48 conversations
548 messages



Input size 250
Chunk size 500
Runtime: 0,119 s
8 PEs
9 processes
17 threads
40 conversations
290 messages

Eden: What we have seen so far

- Eden extends Haskell with parallelism
  - explicit process definitions and implicit communication
  - → control of process granularity, distribution of work, and communication topology
  - implemented by extending the Glasgow Haskell Compiler (GHC)
  - tool EdenTV to analyse parallel program behaviour
- rules of thumb for producing efficient parallel programs
  - number of processes ~ noPe
  - reducing communication
    - chunking
    - offline processes: parameter passing instead of communication
- parallel map implementations

Schemata	task decomposition	task distribution
parMap	regular	static: process per task
farm	regular	static: process per processor
offlineFarm	regular	static: task selection in processes
workpool	irregular	dynamic

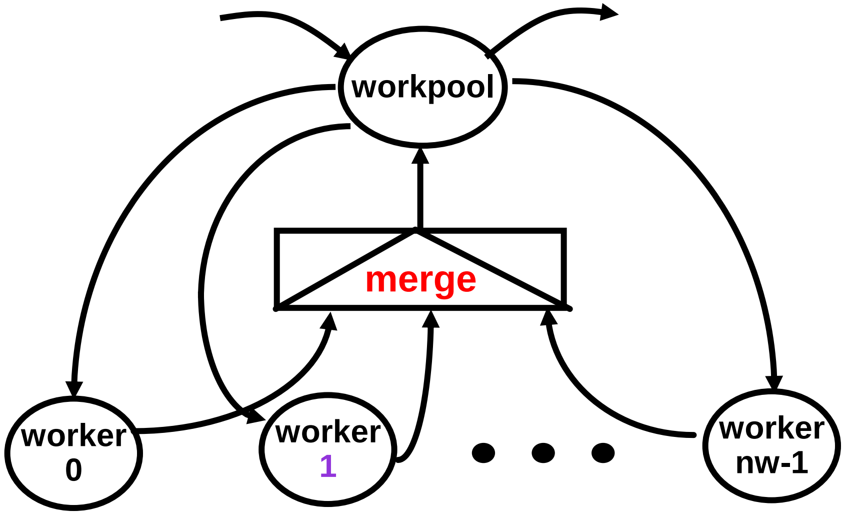
Overview Eden Lectures

- Lectures I & II (Thursday)
  - Motivation
  - Basic Constructs
  - Case Study: Mergesort
  - Eden TV – The Eden Trace Viewer
  - Reducing communication costs
  - Parallel map implementations
  - Explicit Channel Management
  - The Remote Data Concept
  - Algorithmic Skeletons
    - Nested Workpools
    - Divide and Conquer
- Lecture III: Lab Session (Friday Morning)
- Lecture IV: Implementation (Friday Afternoon)
  - Layered Structure
  - Primitive Operations
  - The Eden Module
  - The Trans class
  - The PA monad
  - Process Handling
  - Remote Data

Many-to-one Communication: merge

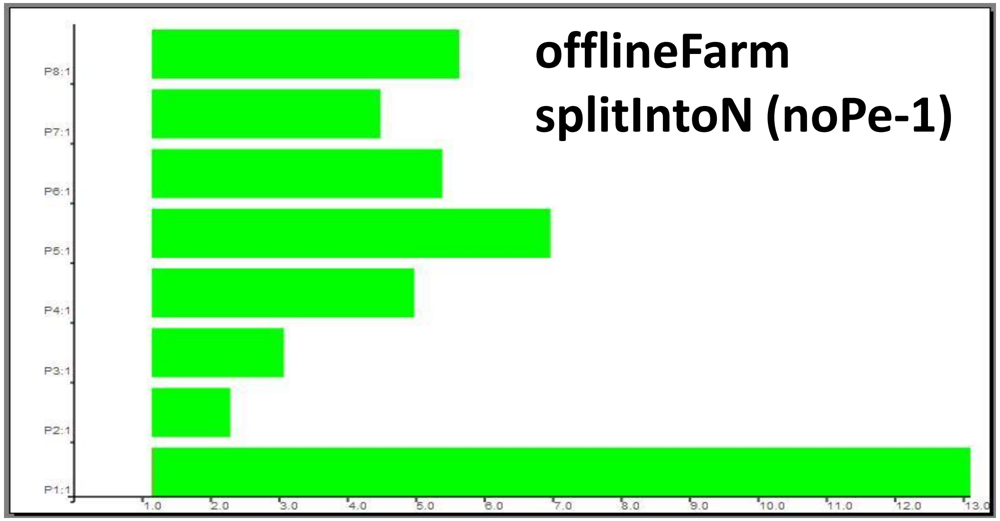
Using non-deterministic merge function: `merge :: [[a]] -> [a]`

Workpool or Master/Worker Scheme



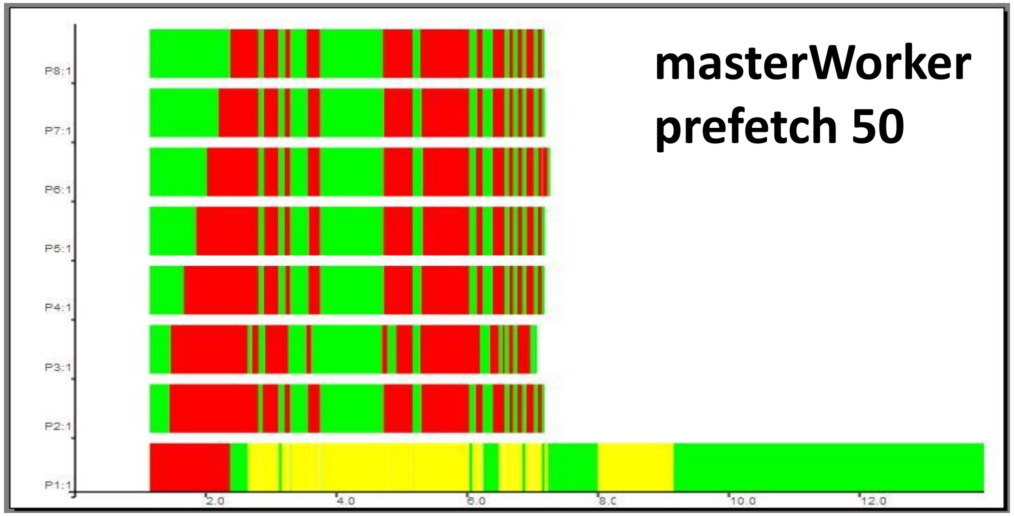
```
masterWorker :: (Trans a, Trans b) => Int -> Int -> (a->b) -> [a] -> [b]
masterWorker nw prefetch f tasks = orderBy fromWs reqs
 where  fromWs      = parMap (map f) toWs
        toWs        = distribute np tasks reqs
        reqs        = initReqs ++ newReqs
        initReqs    = concat (replicate prefetch [0..nw-1])
        newReqs     = merge [[i | r <- rs] | (i,rs) <- zip [0..nw-1] fromWs]
```

Example: Mandelbrot revisited!



offlineFarm
splitIntoN (noPe-1)

Input size 2000
Runtime: 13,09 s
8 PEs
8 processes
15 threads
35 conversations
1536 messages



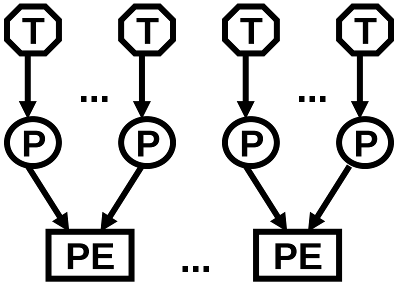
masterWorker
prefetch 50

Input size 2000
Runtime: 13,91 s
8 PEs
8 processes
22 threads
42 conversations
3044 messages

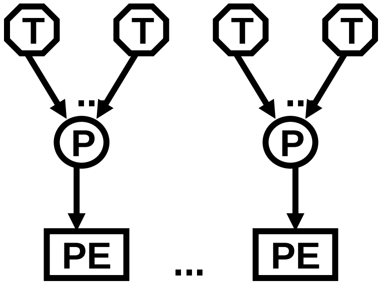
Parallel map implementations

- **static task distribution:**

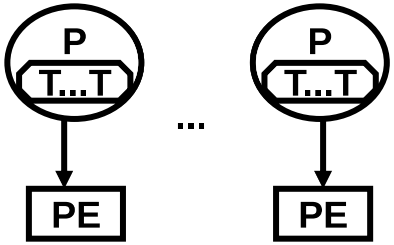
parMap



farm

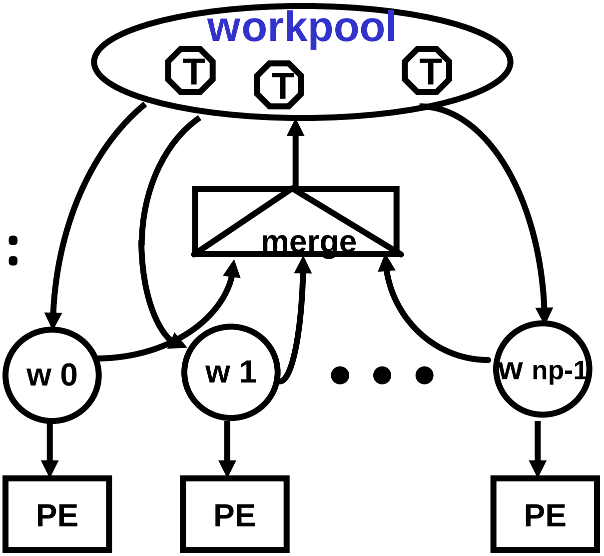


offlineFarm



increasing granularity

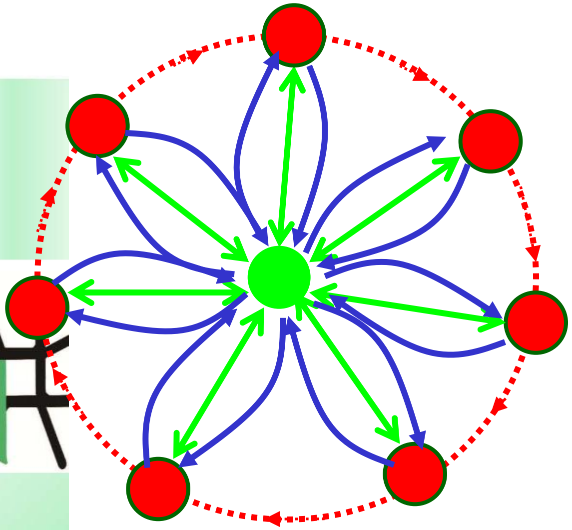
- **dynamic task distribution:**



Explicit Channel Management

Explicit Channel Management in Eden

Example: Definition of a process ring

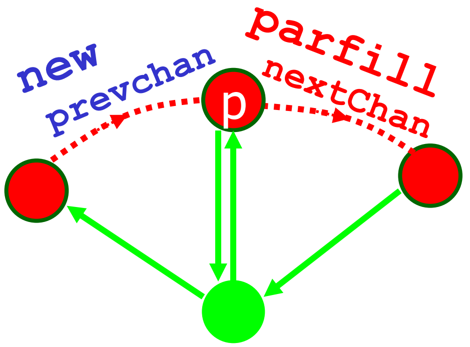


```
ring :: (Trans i,Trans o,Trans r) =>
   ((i,r) -> (o,r))  -> -- ring process fct
   [i] -> [o]           -- input-output fct

ring f is = os
   where
    (os, ringOuts)
       = unzip [process f # inp |
                 inp <- lazyzip is ringIns]

    ringIns          = rightRotate ringOuts
    rightRotate xs   = last xs : init xs
```

rightrotate

Problem: only indirect ring connections via parent process

Explicit Channels in Eden



- Channel generation

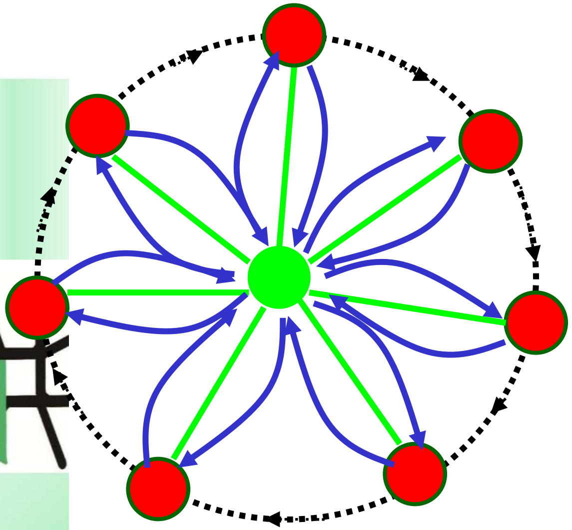
```
new ::     Trans a =>
  (ChanName a -> a -> b) -> b
```

- Channel usage

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

```
plink ::
  (Trans i,Trans o, Trans r) =>
  ((i,r) -> (o,r)) ->
   Process (i,ChanName r)
           (o,ChanName r)
plink f = process fun_link
 where
  fun_link (fromP, nextChan)
  = new (\ prevChan prev ->
     let
      (toP, next)
         = f (fromP, prev)
     in
       parfill nextChan next
               (toP, prevChan)
          )
```

Ring Definition with explicit channels



```
ring :: (Trans i,Trans o,Trans r) =>
   ((i,r) -> (o,r))  -> -- ring process fct
   [i] -> [o]           -- input-output fct

ring f is = os
   where
    (os, ringOuts)
      = unzip [process f # inp |
               inp <- lazyzip is ringIns]

    ringIns            = rightRotate ringOuts
    rightRotate xs     = last xs : init xs
```

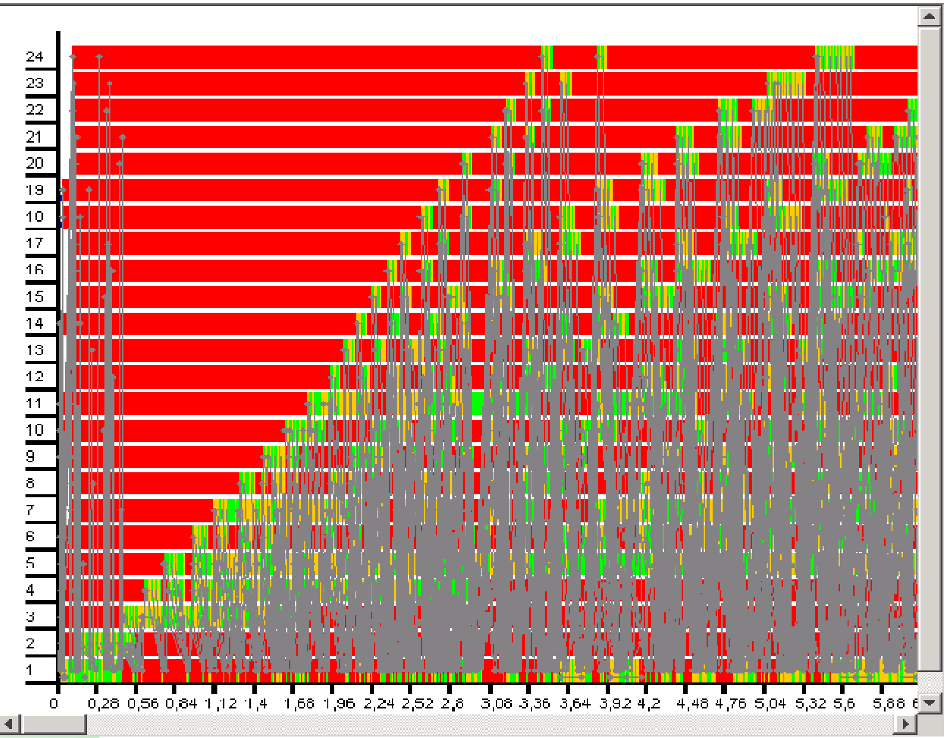
rightrotate

leftrotate

plink

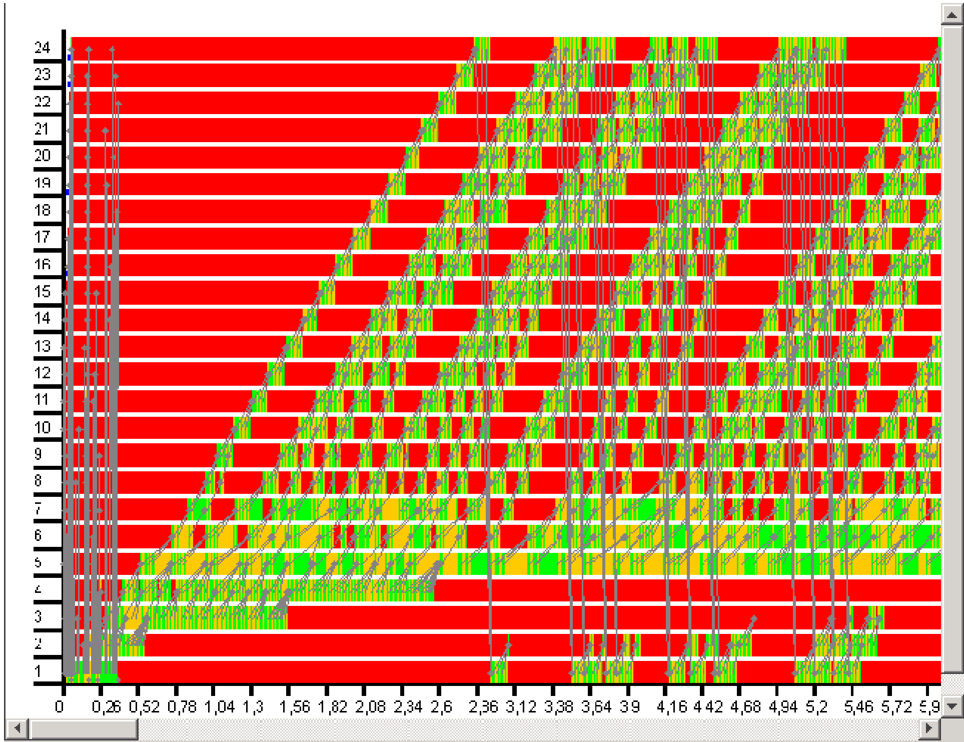
Problem: only indirect ring connections via parent process

Traceprofile Ring
Implicit vs explicit channels



ring with implicit channels - All communications go through generator process (number 1).

Ring with explicit channels – Ring processes communicate directly.

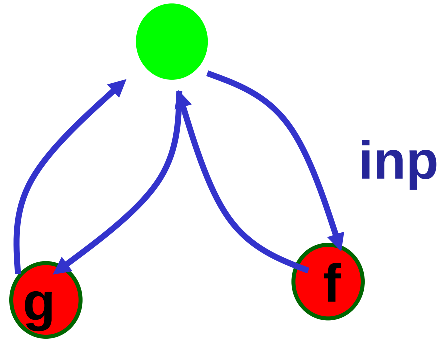


The Remote Data Concept

The „Remote Data“-Concept

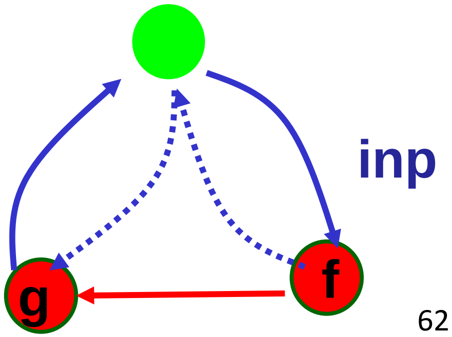
- **Functions:**
  - Release local data with release :: a -> RD a
  - Fetch released data with fetch :: RD a -> a

- **Replace**
  - (process g # (process f # inp))



with

- process (g o fetch) # (process (release o f) # inp)



Ring Definition with Remote Data

```
ring :: (Trans i,Trans o,Trans r) =>
  ((i,r) -> (o,r))  -> -- ring process fct
  [i] -> [o]           -- input-output fct

ring f is = os
  where
   (os, ringOuts)
      = unzip [process f_RD # inp |
               inp <- lazyzip is ringIns]

   f_RD (i, ringIn) = (o, release ringOut)
      where (o, ringOut) = f (i, fetch ringIn)

   ringIns          = rightRotate ringOuts
   rightRotate xs   = last xs : init xs
```

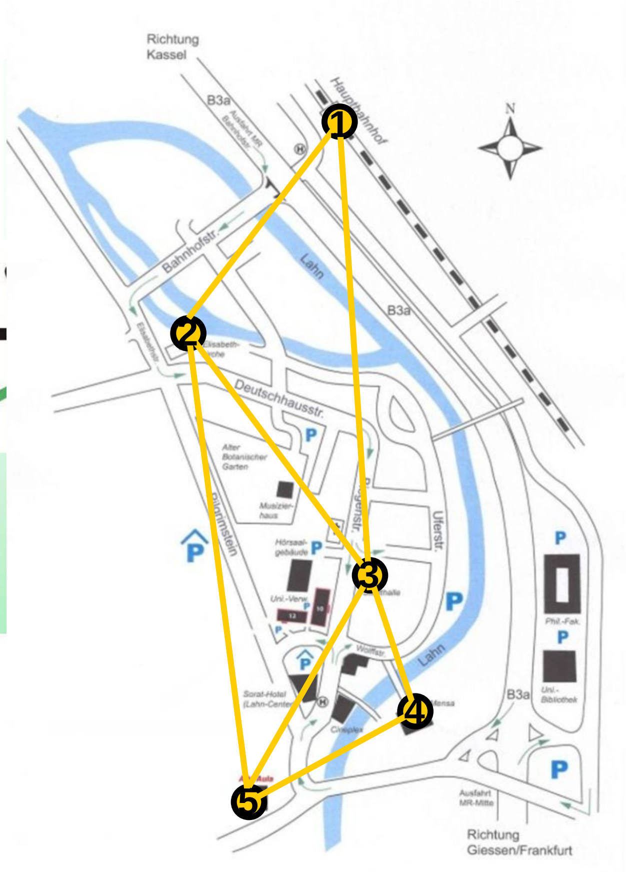
type [RD r]

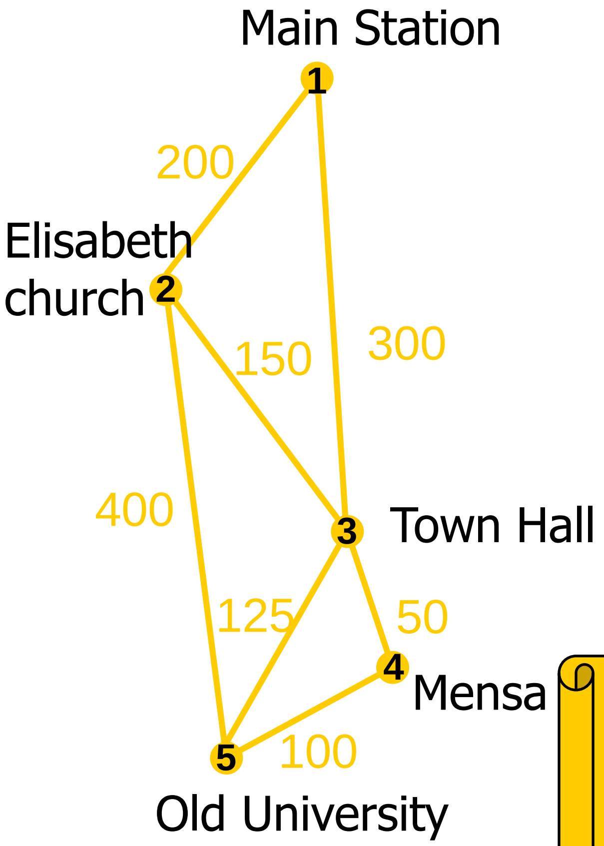
Implementation of Remote Data with dynamic channels

```
-- remote data
type RD a = ChanName (ChanName a)


-- convert local data into corresponding remote data
release :: Trans a ⇒ a → RD a
release x = new (\ cc c → parfill c x cc)


-- convert remote data into corresponding local data
fetch :: Trans a ⇒ RD a → a
fetch cc = new (\ c x → parfill cc c x)
```

Example: Computing Shortest Paths

Map -> Graph -> Adjacency matrix/ Distance matrix

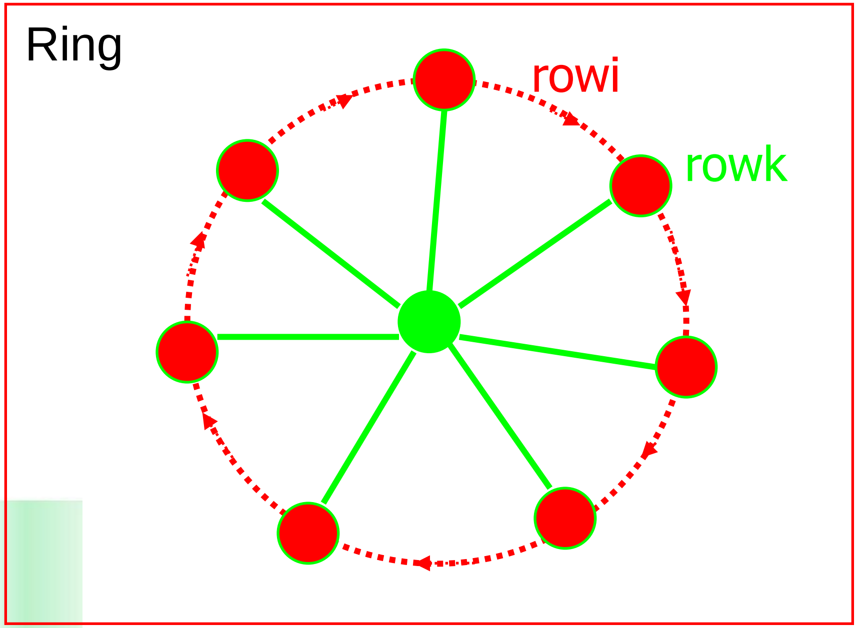




$$\begin{pmatrix} 0 & 200 & 300 & \infty & \infty \\ 200 & 0 & 150 & \infty & 400 \\ 300 & 150 & 0 & 50 & 125 \\ \infty & \infty & 50 & 0 & 100 \\ \infty & 400 & 125 & 100 & 0 \end{pmatrix}$$

Compute the shortest way from A to B für arbitrary nodes A and B!

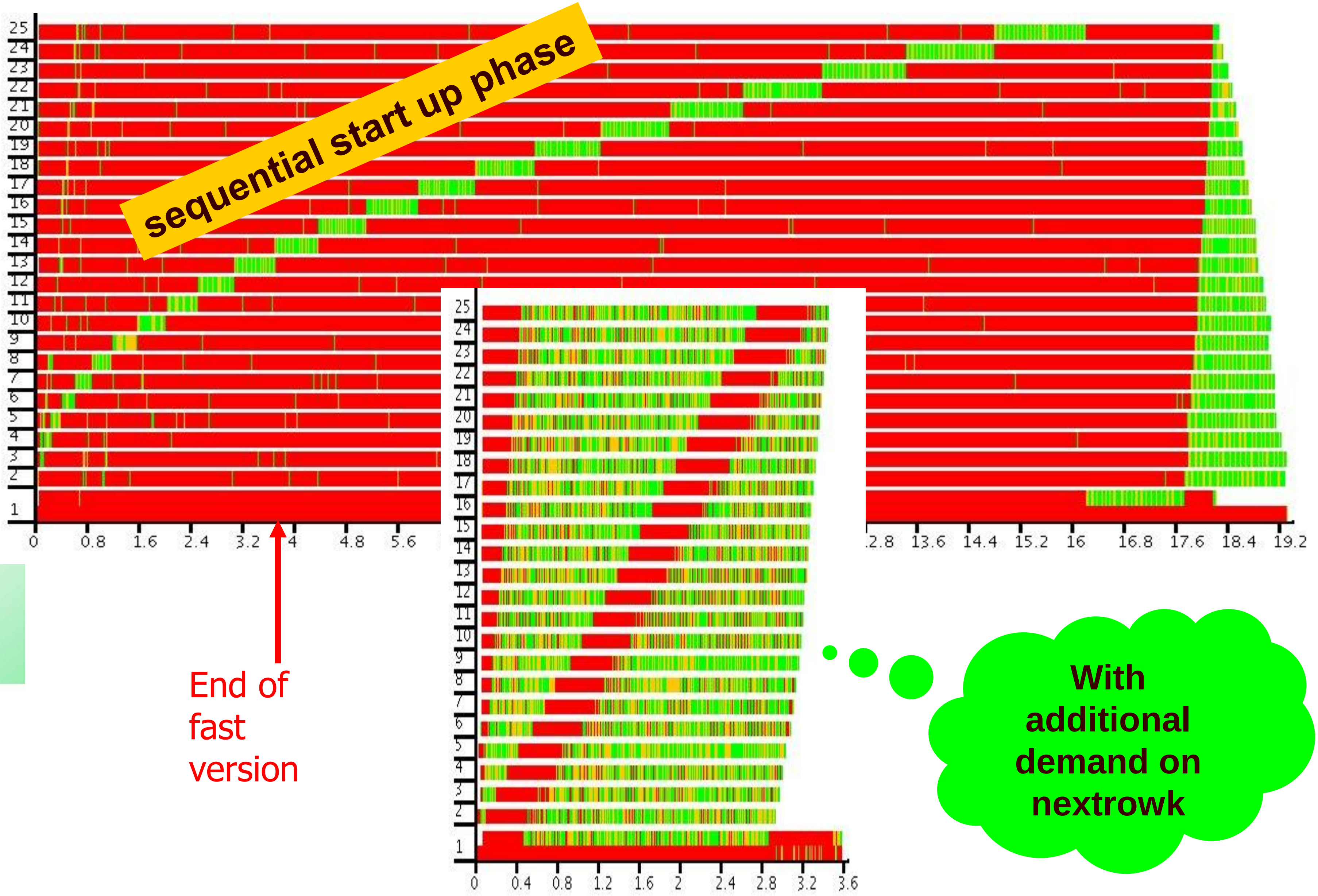
Warshall‘s algorithm in process ring



Force evaluation of nextrowk by inserting
`rnf nextrowk `pseq``
before call of ring_iterate

```
ring_iterate :: Int -> Int -> Int ->
                       [Int] -> [[Int]] -> ( [Int], [[Int]])
ring_iterate size k i rowk (rowi:xs)
    | i > size   =  (rowk, [])     -- End of iterations
    | i == k     =  (rowR, rowk:restoutput) -- send own row
    | otherwise =  (rowR, rowi:restoutput) -- update row
 where
 (rowR, restoutput) = ring_iterate size k (i+1) nextrowk xs
  nextrowk | i == k     = rowk -- no update, if own row
           | otherwise = updaterow rowk rowi (rowk!!(i-1))
```

Traces of parallel Warshall



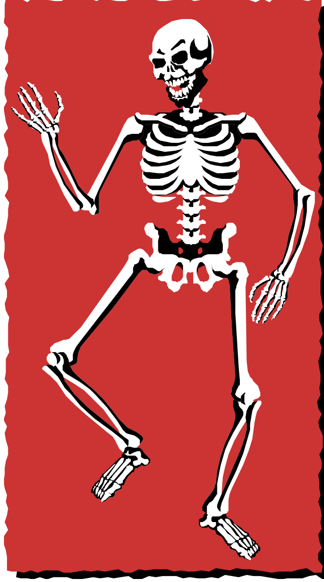
sequential start up phase

End of fast version

With additional demand on nextrowk

(Advanced) Algorithmic Skeletons

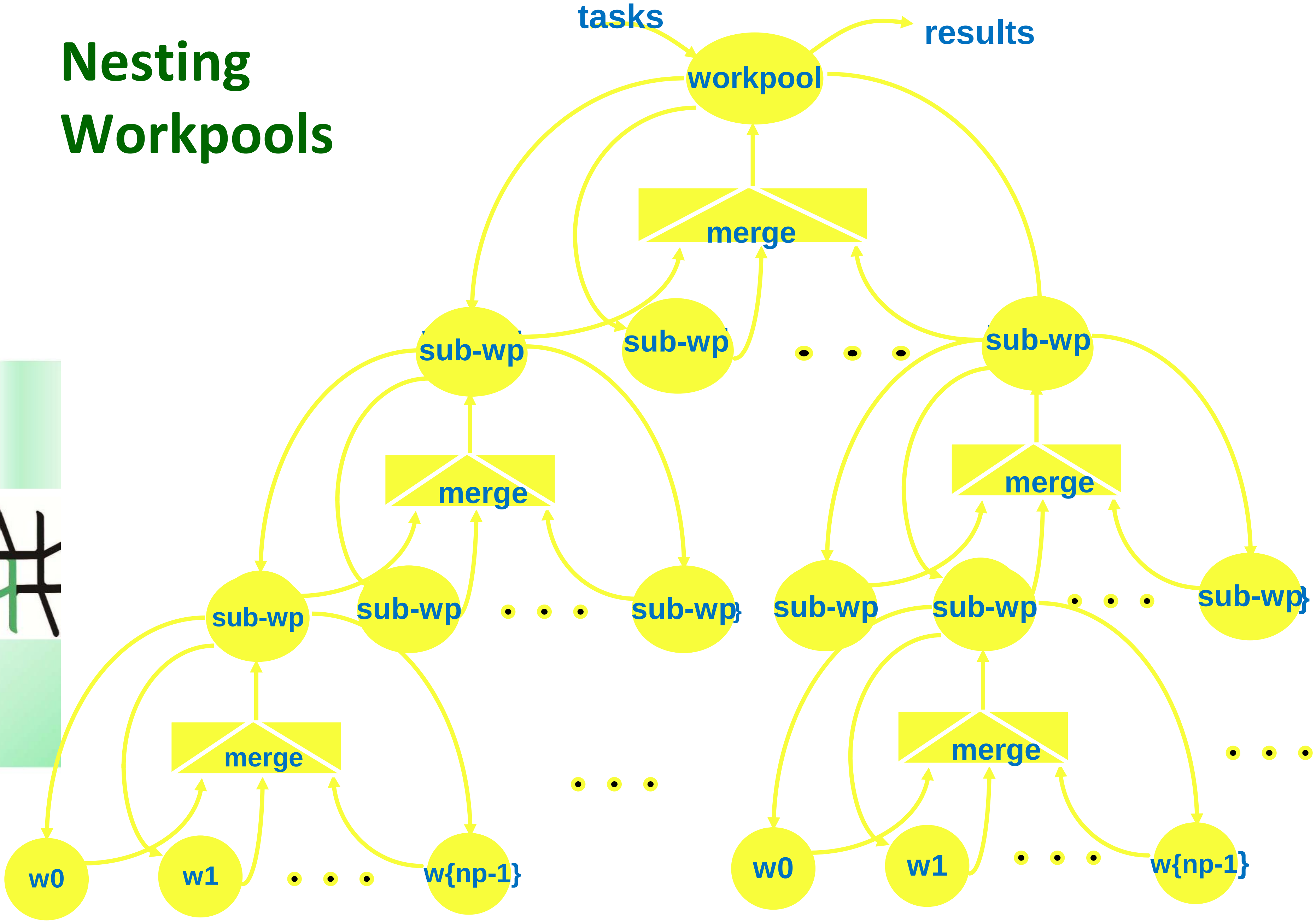
Algorithmic Skeletons



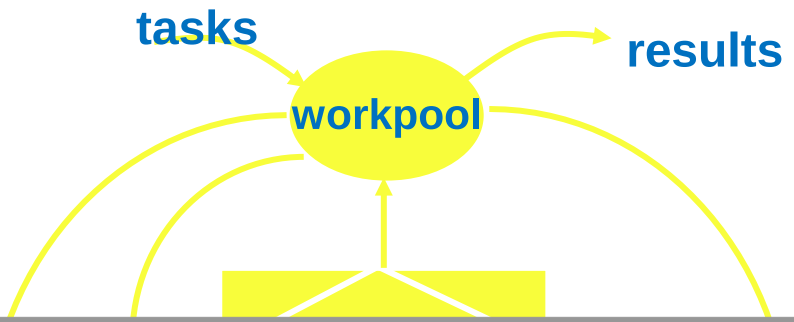
- patterns of parallel computations
  => in Eden:
  parallel higher-order functions
- typical patterns:
  - parallel maps and master-worker systems:
    parMap, farm, offline_farm, mw (workpoolSorted)
  - map-reduce
  - topology skeletons: pipeline, ring, torus, grid, trees ...
  - divide and conquer
- in the following:
  - nested master-worker systems
  - divide and conquer schemes

See Eden‘s Skeleton Library

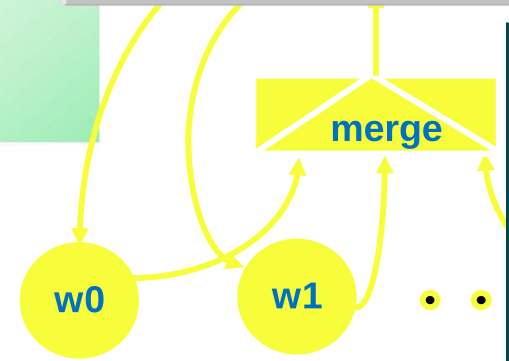
Nesting Workpools

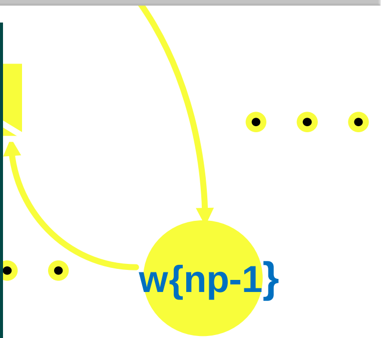


Nesting Workpools

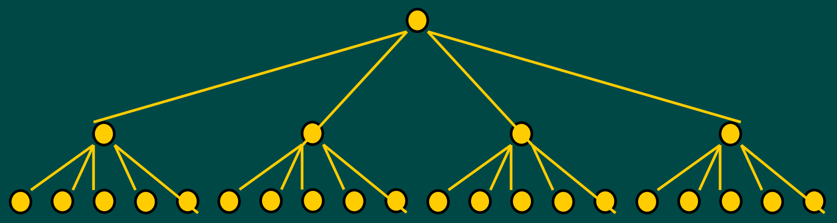


```
wpNested :: (Trans a, Trans b) =>
            [Int] -> [Int] ->   -- branching degrees/prefetches
                                -- per level
            ([a] -> [b]) ->     -- worker function
            [a] -> [b]          -- tasks, results
wpNested ns pfs wf = foldr fld wf (zip ns pfs)
  where
    fld :: (Trans a, Trans b) =>
           (Int,Int) -> ([a] -> [b]) -> ([a] -> [b])
    fld (n,pf) wf = workpool' n pf wf
```

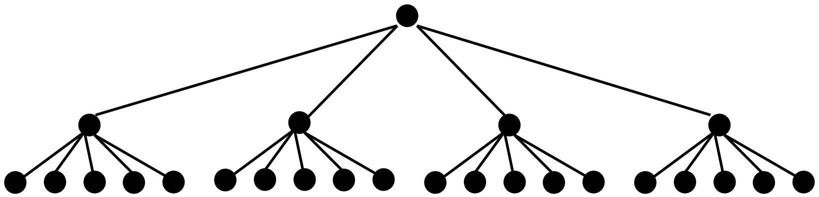


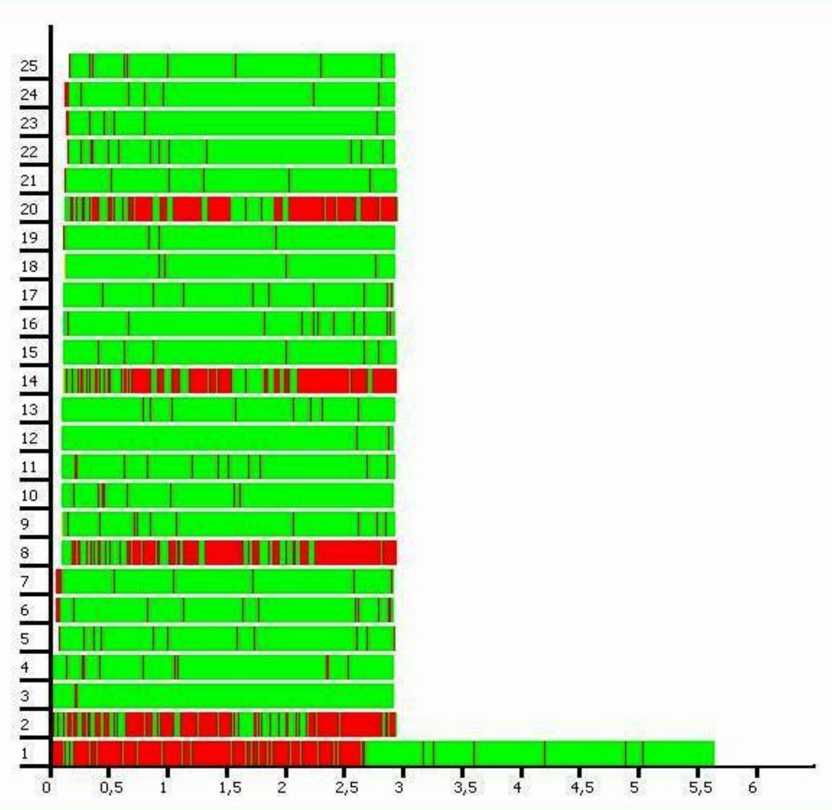


wpnested [4,5] [64,8] yields



Hierarchical Workpool





Mandelbrot Trace

1 master
4 submasters
20 workers

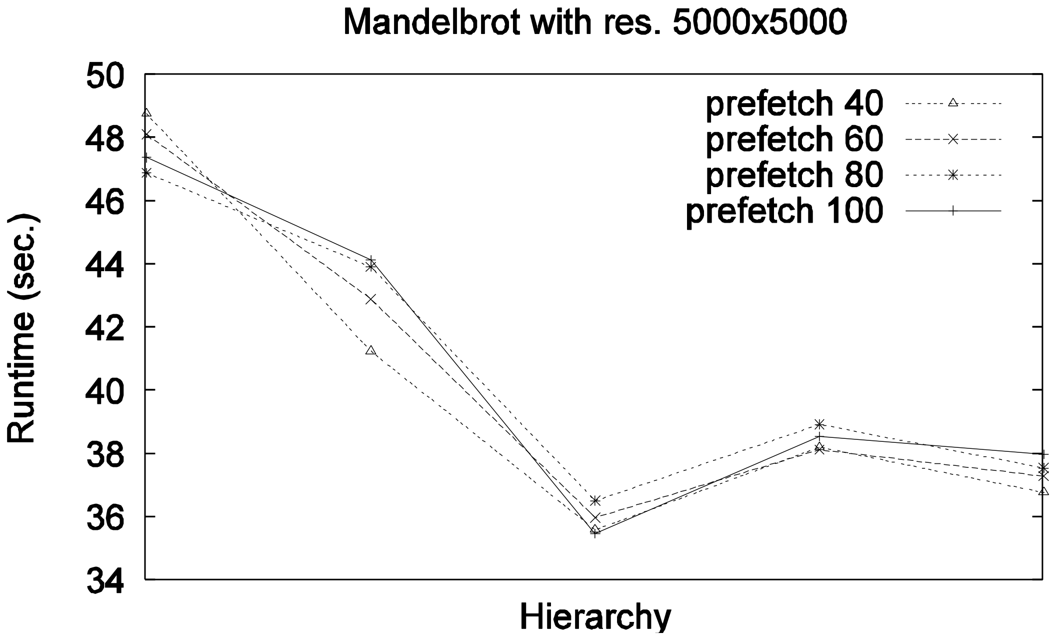
faster result collection
via hierarchy
-> better overall runtime

Problem size: 2000 x 2000

Platform: Beowulf cluster Heriot-Watt-University, Edinburgh
(32 Intel P4-SMP nodes @ 3 GHz, 512MB RAM, Fast Ethernet)

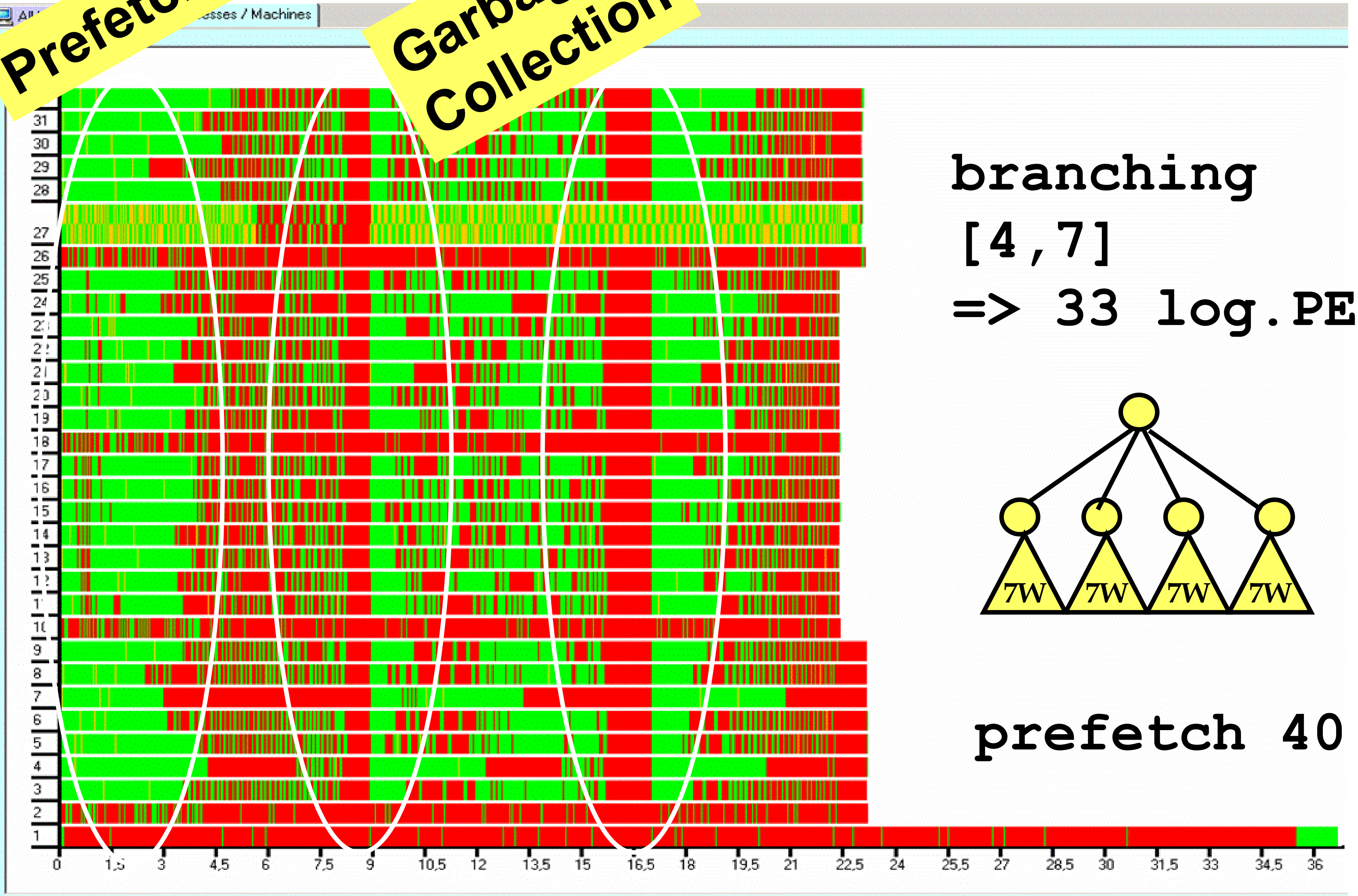
Experimental Results

- **Mandelbrot set visualisation**
- **. . . for 5000 5000 pixels, calculated line-wise (5000 tasks)**
- **Platform: Beowulf cluster Heriot-Watt-University**
  (32 Intel P4-SMP nodes @ 3 GHz, 512MB RAM, Fast Ethernet)



	[31]	[2,15]	[4,7]	[6,5]	[2,2,4]
logical PEs	32	33	33	37	35

1-Level-Nesting Trace



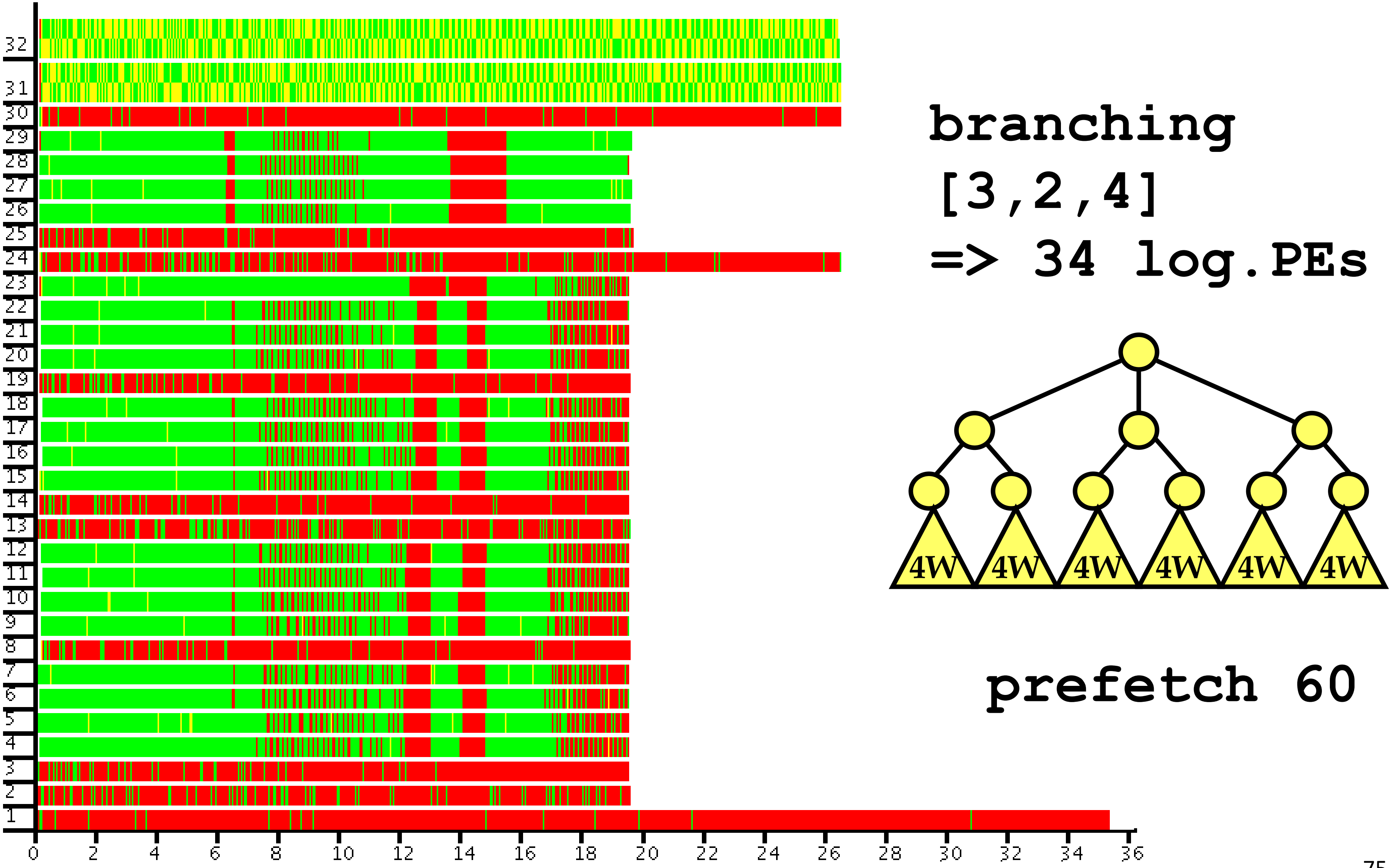
Prefetch

Garbage Collection

branching [4,7] => 33 log.PEs

prefetch 40

3-Level-Nesting Trace



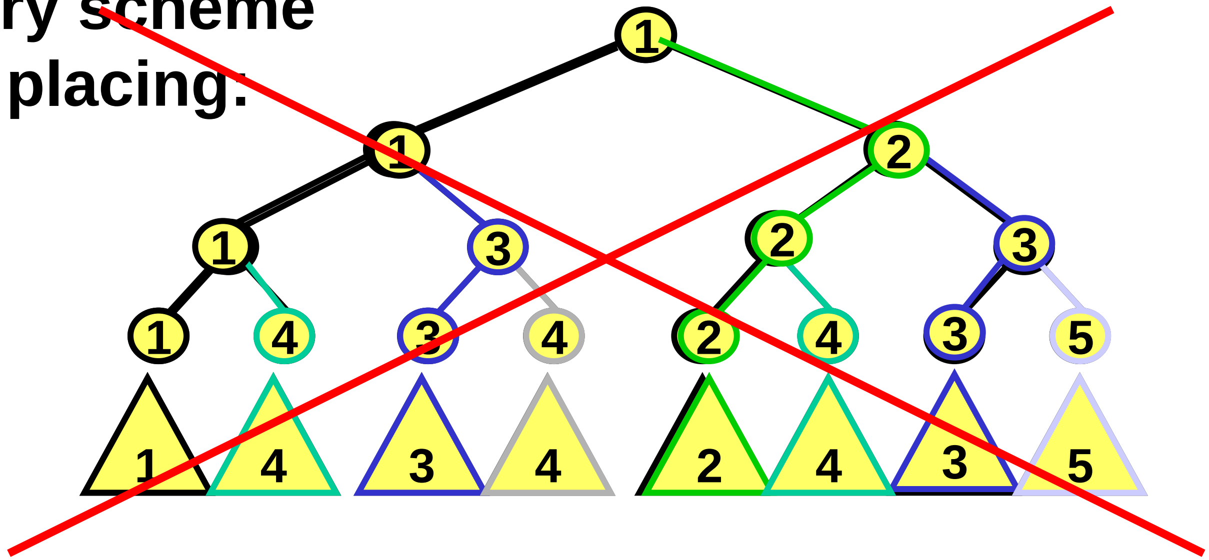
branching [3,2,4] => 34 log.PEs

prefetch 60

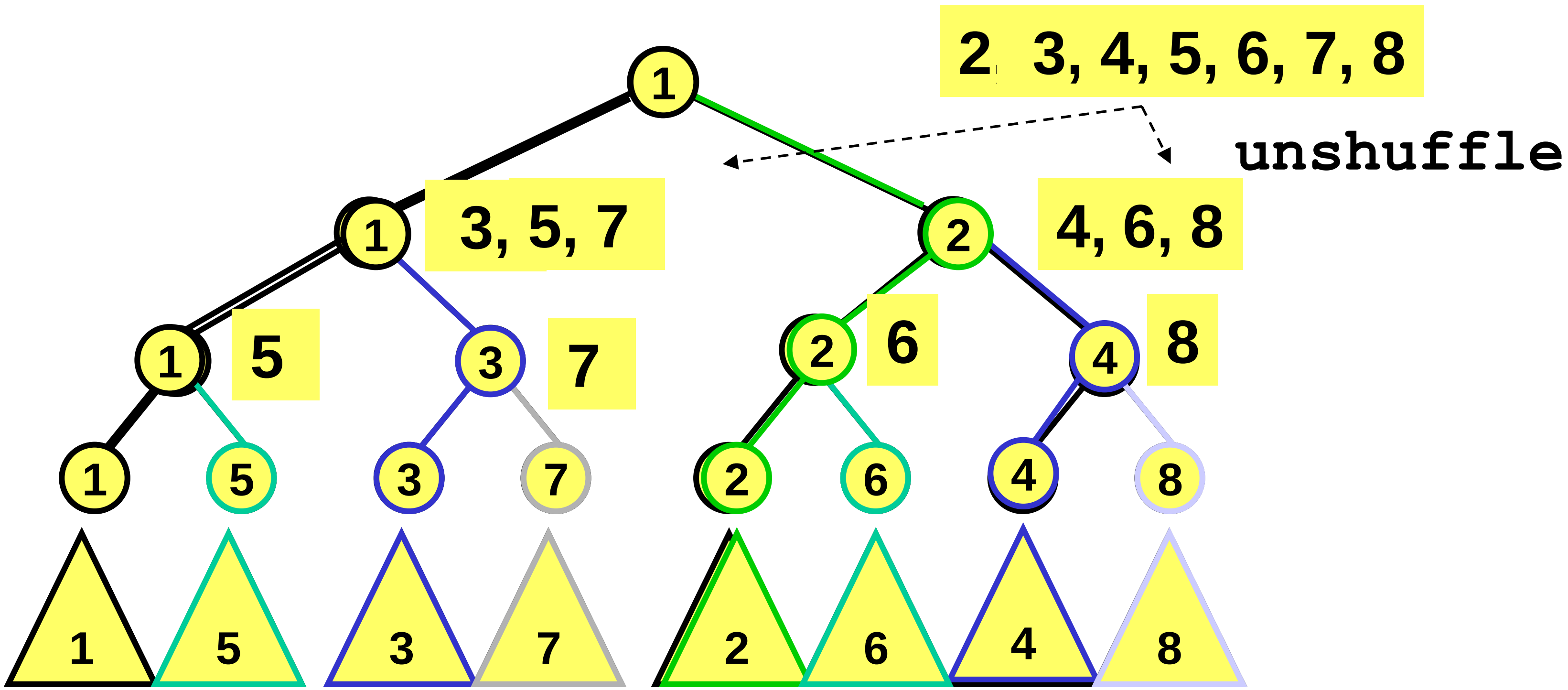
Divide-and-conquer

```
dc :: (a->Bool) -> (a->b) -> (a->[a]) -> ([b]->b) -> a->b
dc trivial solve split combine task
   =    if trivial task then solve task
        else combine (map rec_dc (split task))
   where rec_dc = dc trivial solve split combine
```

regular binary scheme with default placing:



Explicit Placement via Ticket List



Regular DC-Skeleton with Ticket Placement

```
dcNTickets :: (Trans a, Trans b) =>
           Int -> [Int] -> ...   -- branch degree / tickets / ...
dcNTickets k [] trivial solve split combine
    = dc trivial solve split combine
dcNTickets k tickets trivial solve split combine x
    = if trivial x then solve x
      else    childRes `pseq` rnf myRes `pseq`  -- demand control
              combine (myRes:childRes ++ localRess )
    where   childRes  = spawnAt childTickets childProcs procIns
            childProcs = map (process . rec_dcN) theirTs
            rec_dcN ts = dcNTickets k ts trivial solve split combine
            -- ticket distribution
            (childTickets, restTickets) = splitAt (k-1) tickets
            (myTs: theirTs) = unshuffle k restTickets
            -- input splitting
            (myIn:theirIn) = split x
            (procIns, localIns) = splitAt (length childTickets) theirIn
            -- local computations
            myRes = dcNTickets k myTs trivial solve split combine myIn
            localRess = map (dc trivial solve split combine) localIns
```

Regular DC-Skeleton with Ticket Placement

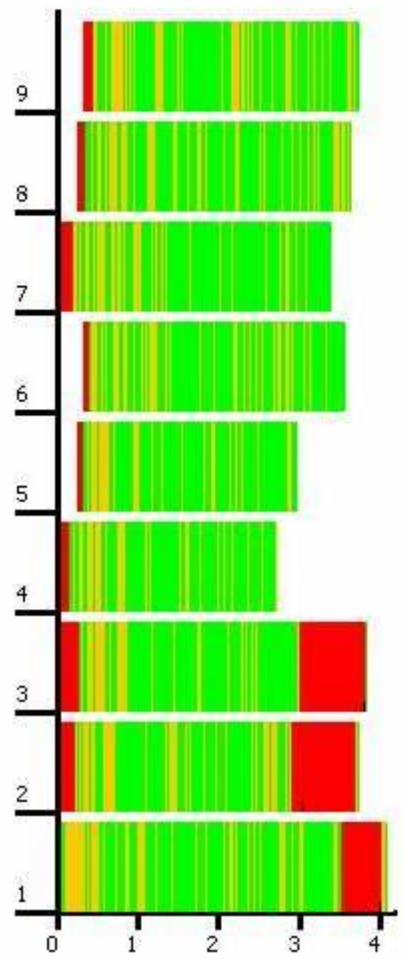
```
dcNTickets :: (Trans a, Trans b) =>
        Int -> [Int] -> ...    -- branch degree / tickets / ...
dcNTickets k [] trivial solve split combine
   = dc trivial solve split combine
dcNTickets k tickets trivial solve split combine x
   = i

  wh

                                                                 ne

          -- input splitting
          (myIn:theirIn) = split x
          (procIns, localIns) = splitAt (length childTickets) theirIn
          -- local computations
          myRes = dcNTickets k myTs trivial solve split combine myIn
          localRess = map (dc trivial solve split combine) localIns
```

- arbitrary, but fixed branching degree
- flexible, works with
    - too few tickets
    - double tickets
- parallel unfolding controlled by ticket list

Case Study: Karatsuba

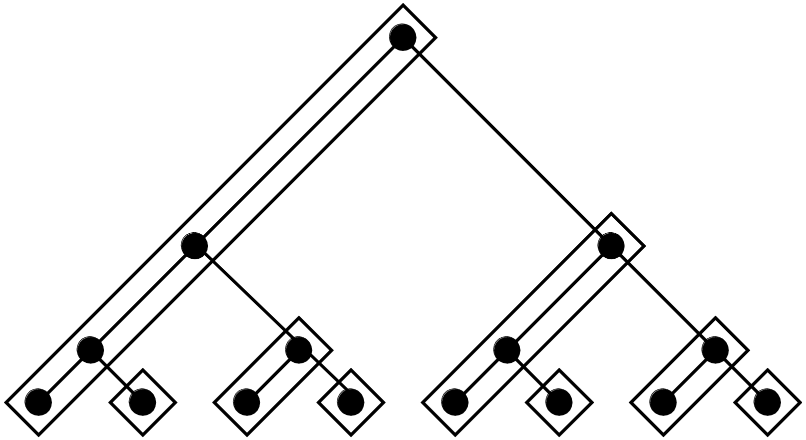
- multiplication of large integers
- fixed branching degree 3
- complexity $O(n^{\log_2 3})$,
  combine complexity $O(n)$

- Platform: LAN (Fast Ethernet),
  7 dual-core linux workstations,
  2 GB RAM
- input size: 2 integers with 32768 digits each

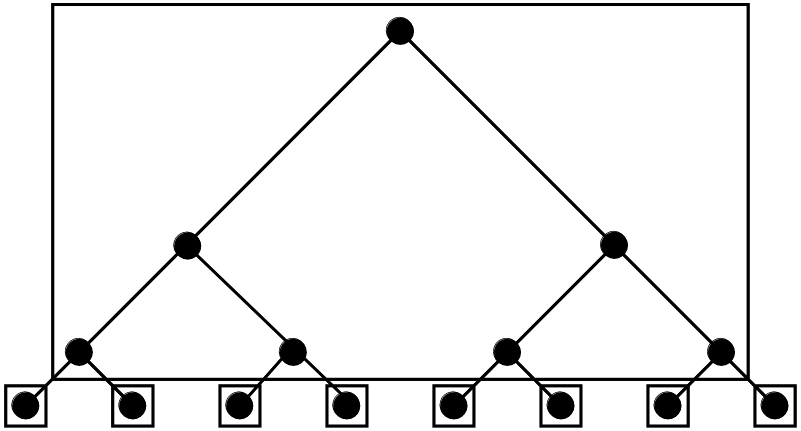


Divide-and-Conquer Schemes

- **Distributed expansion**



- **Flat expansion**

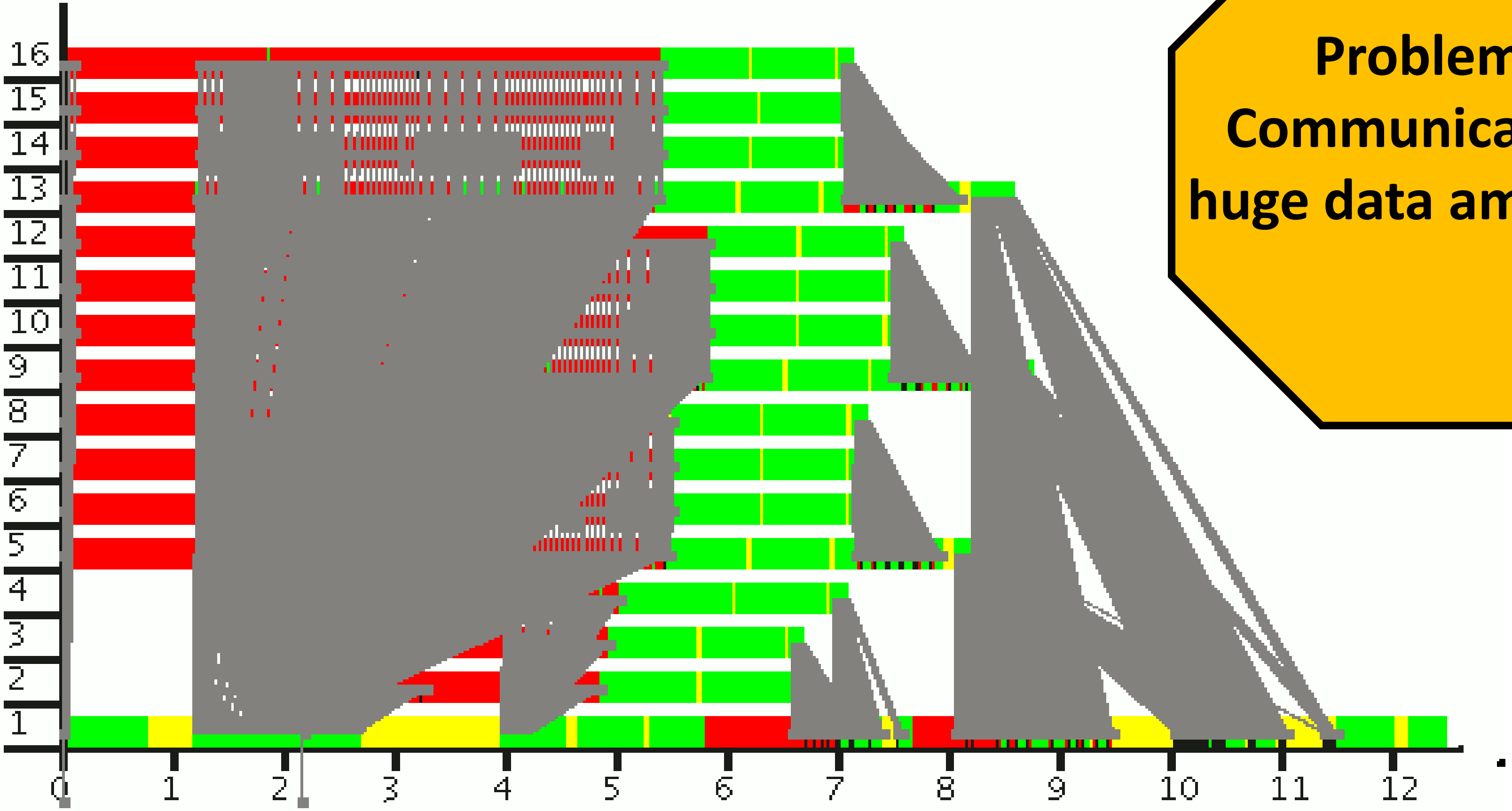


Divide-and-Conquer Using Master-Worker

```
divConFlat :: (Trans a,Trans b, Show b, Show a, NFData b) =>
          ((a->b) -> [a] -> [b])
           -> Int  -> (a->Bool) -> (a->b) -> (a->[a]) -> ([b]->b) -> a -> b
divConFlat parallelMapSkel depth trivial solve split combine x
 = combineTopMaster (\_ -> combine) levels results
 where
   (tasks,levels) = generateTasks depth trivial split x
   results          = parallelMapSkel  dcSeq  tasks
   dcSeq            = dc trivial solve split combine
```

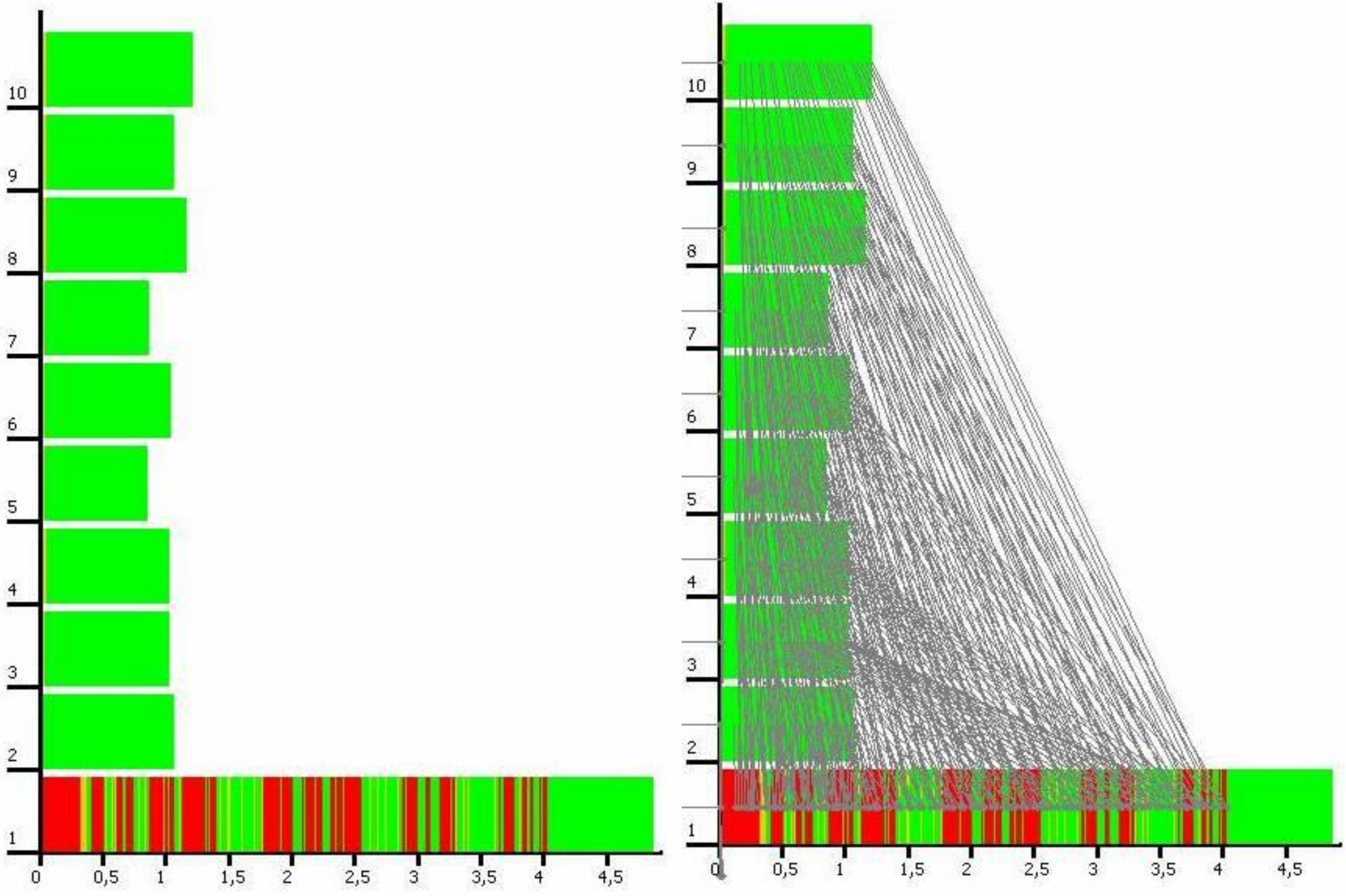
Case Study: Parallel FFT

- frequency distribution in a signal, decimation in time
- 4-radix FFT, input size: 4^{10} complex numbers
- Platform: Beowulf Cluster Edinburgh



Problem: Communicating huge data amounts

Using Master-Worker-DC



Intermediate Conclusions

- **Eden enables high-level parallel programming**
- **Use predefined or design own skeletons**
- **Eden‘s skeleton library provides a large collection of sophisticated skeletons:**
  - **parallel maps: parMap, farm, offlineFarm …**
  - **master-worker: flat, hierarchical, distributed …**
  - **divide-and-conquer: ticket placement, via master-worker …**
  - **topological skeletons: ring, torus, all-to-all, parallel transpose …**

Eden Lab Session

- Download the exercise sheet from
  http://www.mathematik.uni-marburg.de/~eden/?content=cefp
- Choose one of the three assignments and download the corresponding sequential program:
  sumEuler.hs (easy) - juliaSets.hs (medium) - gentleman.hs (advanced)
- Download the sample mpihosts file and modify it to randomly chosen lab computers nylxy with xy chosen from 01 up to 64
- Call `edenenv` to set up the environment for Eden

- Compile Eden programs with

```
ghc   –parmpi   --make –O2 –eventlog   myprogram.hs
```

- Run compiled programs with

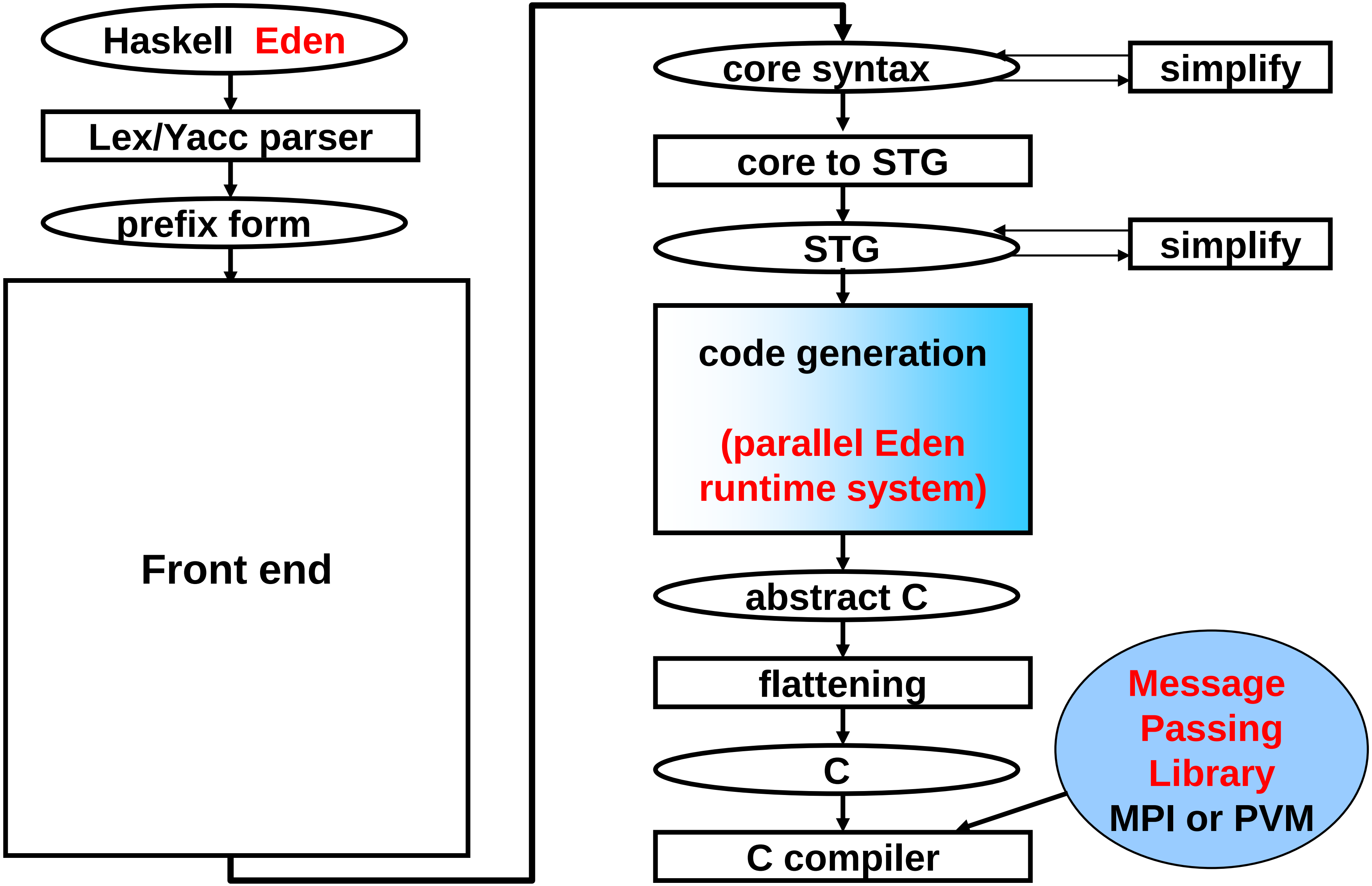
```
myprogram <parameters>   +RTS   –ls    -Nx  -RTS   with x=noPe
```

- View activity profile (trace file) with

```
edentv    myprogram_..._-N..._-RTS.parevents
```

Eden‘s Implementation

Glasgow Haskell Compiler
& Eden Extensions



Haskell **Eden**

Lex/Yacc parser

prefix form

Front end

core syntax

simplify

core to STG

STG

simplify

code generation

(parallel Eden runtime system)

abstract C

flattening

C

C compiler

Message Passing Library

MPI or PVM

Eden‘s parallel runtime system (PRTS)

Modification of GUM, the PRTS of GpH (Glasgow Parallel Haskell):

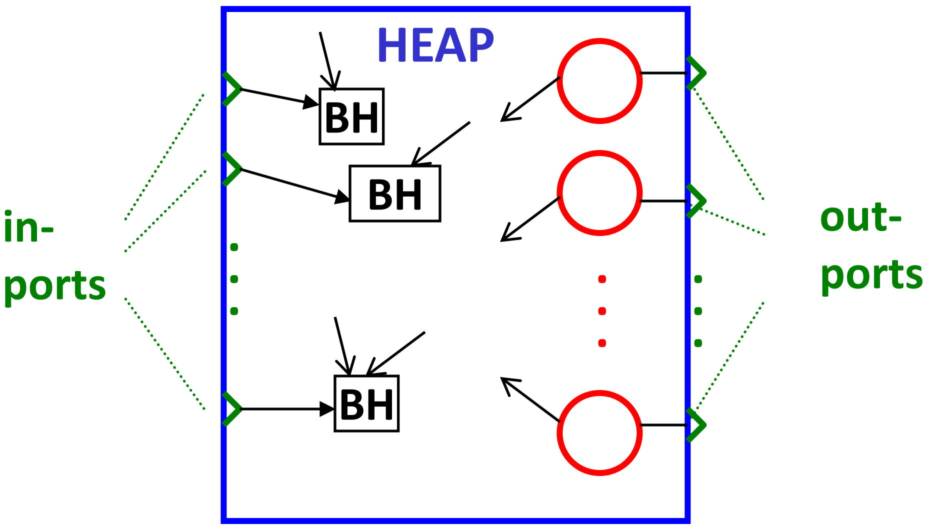
- Recycled
  - Thread management: heap objects, thread scheduler
  - Memory management: local garbage collection
  - Communication: graph packing and unpacking routines

- Newly developed
  - Process management: runtime tables, generation and termination
  - Channel management: channel representation, connection, etc.

- Simplifications
  - no „virtual shared memory“ (global address space) necessary
  - no globalisation of unevaluated data
  - no global garbage collection of data

DREAM: DistRibuted Eden Abstract Machine

- **abstract view of Eden‘s parallel runtime system**
- **abstract view of process:**



○ **Thread represented by TSO (thread state object) in the heap**

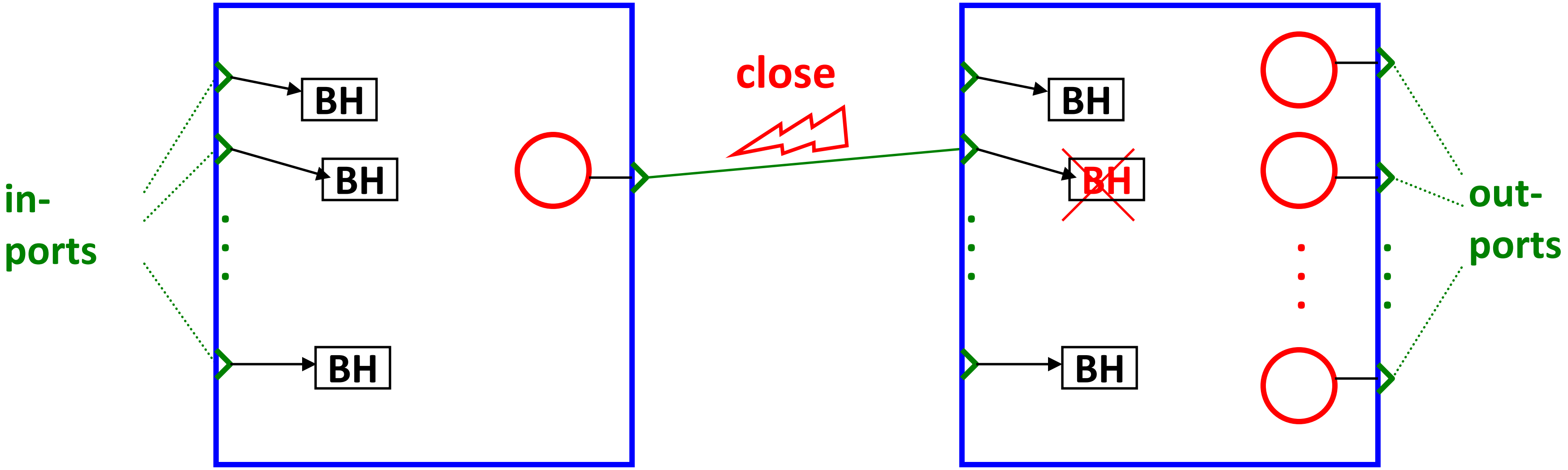
BH **black hole closure, on access threads are suspended until this closure is overwritten**

Garbage Collection and Termination

- no global address space
- local heap
- inports/outports

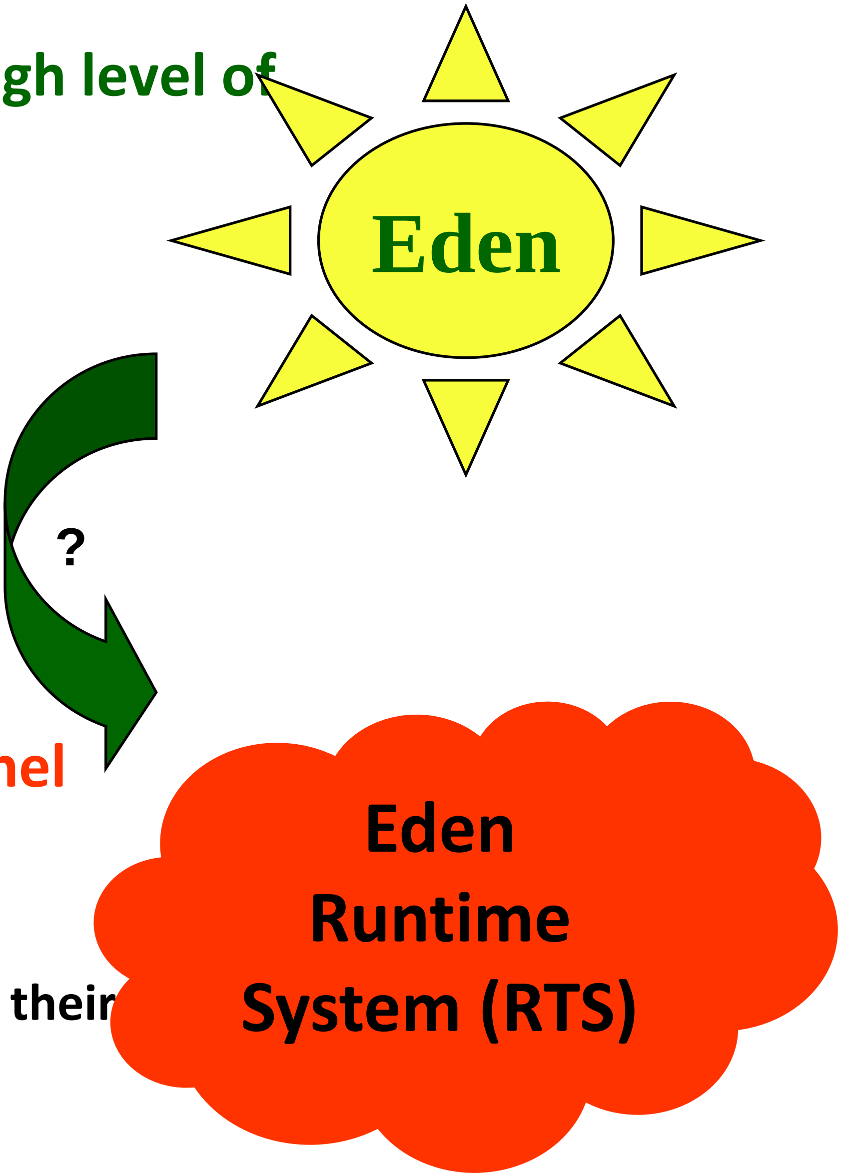
- no need for global garbage collection
- local garbage collection
- outports as additional roots

  → inports can be recognised as garbage



Implementation of Eden

- Parallel programming on a high level of abstraction
  - explicit process definitions
  - implicit communication



Eden

?

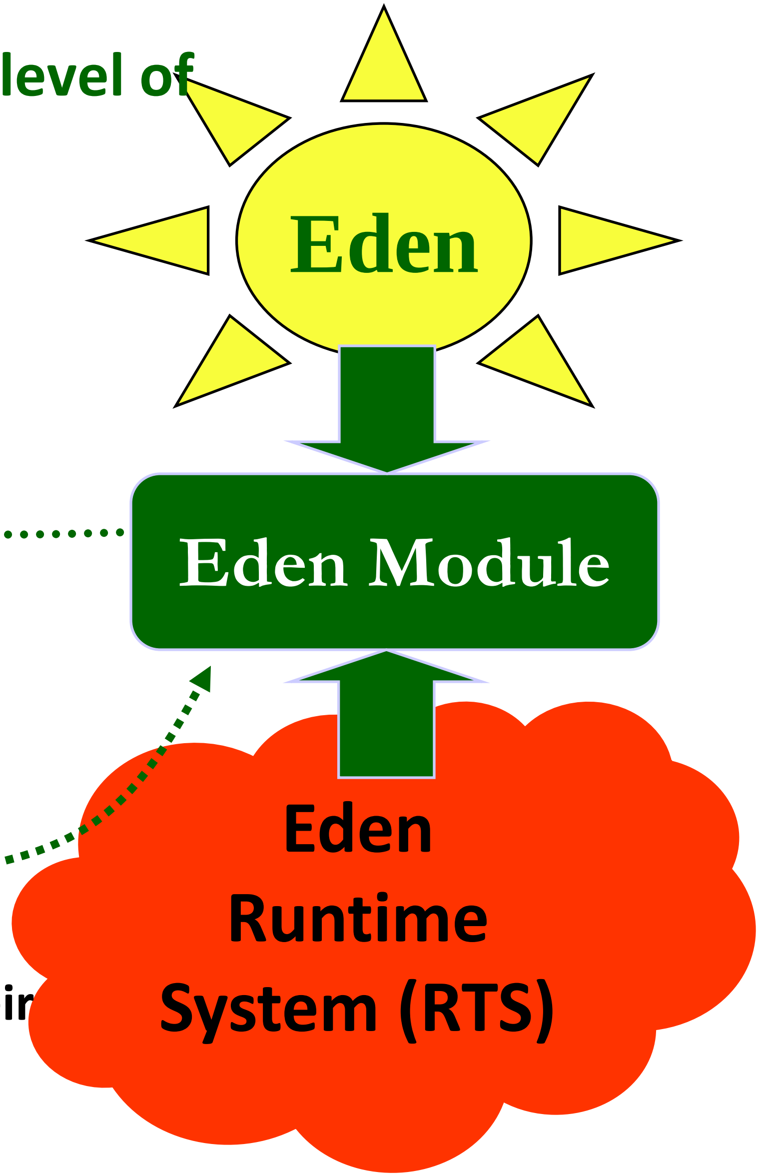
Eden Runtime System (RTS)

- Automatic process and channel management
  - Distributed graph reduction
  - Management of processes and their interconnecting channels
  - Message passing

Implementation of Eden

- Parallel programming on a high level of abstraction
  - explicit process definitions
  - implicit communication

- Automatic process and channel management
  - Distributed graph reduction
  - Management of processes and their interconnecting channels
  - Message passing

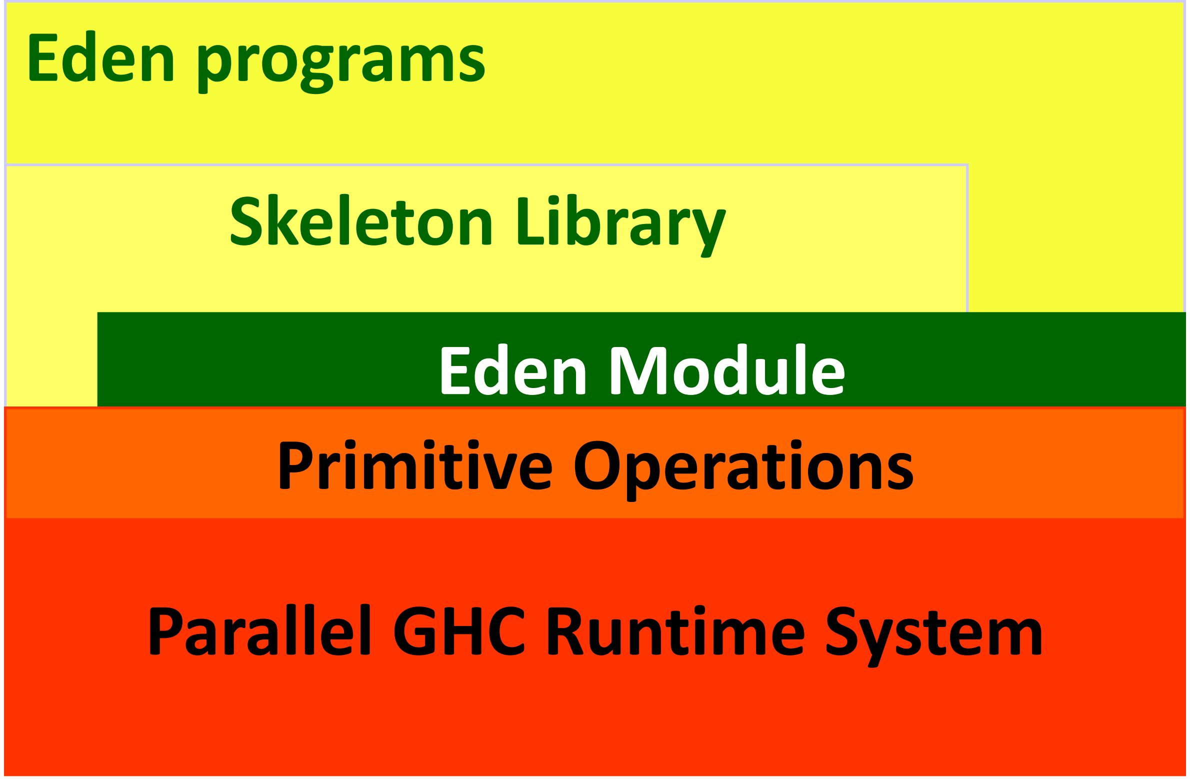


Eden

Eden Module

Eden Runtime System (RTS)

Layer Structure


Eden programs

Skeleton Library

Eden Module

Primitive Operations

Parallel GHC Runtime System

Parprim – The Interface to the Parallel RTS

Primitive operations provide the basic functionality :

- channel administration
  - → primitive channels (= inports) — data ChanName' a = Chan Int# Int# Int# (PE, Process, Inport)
  - → create communication channel(s) — createC :: IO (ChanName' a, a)
  - → connect communication channel — connectToPort :: ChanName' a -> IO ()
- communication
  - → send data — sendData :: Mode -> a -> IO ()
  - → modi — data Mode = Connect | Stream | Data | Instantiate Int
- thread creation — fork :: IO () -> IO ()
- general — noPE, selfPE :: Int

The Eden Module

Type class **Trans**

- transmissible data types
- overloaded communication functions

  for lists (-> streams): **write :: a -> IO ()**

  and tuples (-> concurrency): **createComm :: IO (ChanName a, a)**

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

explicit definitions
of process, (#) and **Process**
as well as **spawn**

explicit **channels**

- newtype ChanName a
  = Comm (a -> IO ())

Type class Trans

```
class NFData a => Trans a where
  write     :: a -> IO ( )
  write x   =  rnf x `pseq` sendData Data x

  createComm :: (ChanName a, a)
  createComm =  do (cx, x) <- createC
                   return (Comm (sendVia cx), x)


sendVia :: ChanName’ a -> a -> IO ()
sendVia ch d =  do connectToPort ch
                   write d
```

Tuple transmission by concurrent threads

```
instance (Trans a, Trans b)  =>  Trans (a,b) where
         createComm        =  do    (cx,x) <- createC
                                    (cy,y) <- createC
                                    return   (Comm (write2 (cx,cy)),
                                              (x,y))

write2 ::  (Trans a, Trans b) =>
           (ChanName' a, ChanName' b) ->   (a,b) -> IO ()
write2  (c1,c2)  (x1,x2)     =  do    fork (sendVia c1 x1)
                                      sendVia c2 x2
```

Stream Transmission of Lists

```
instance Trans a => Trans [a]  where
   write l@[]      =  sendData Data l
   write (x:xs)    =  do  (rnf x `pseq` sendData Stream x)
                          write xs
```

The PA Monad

Improving control over parallel activities:

```
newtype PA a = PA { fromPA :: IO a }
instance Monad PA where
  return b         = PA $ return b
  (PA ioX) >>= f = PA $ do x <- ioX
                              fromPA $ f x

runPA :: PA a -> a
runPA = unsafeperformIO . fromPA
```

Remote Process Creation

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

output channel(s)

channel for returning input channel handle(s)

```
process   :: (a -> b) -> Process a b
process f = Proc  f_remote
   where    f_remote   (Comm sendResult) inCC
              = do  (sendInput, invals) = createComm
                    connectToPort inCC
                    sendData Data sendInput
                    sendResult (f invals)
```

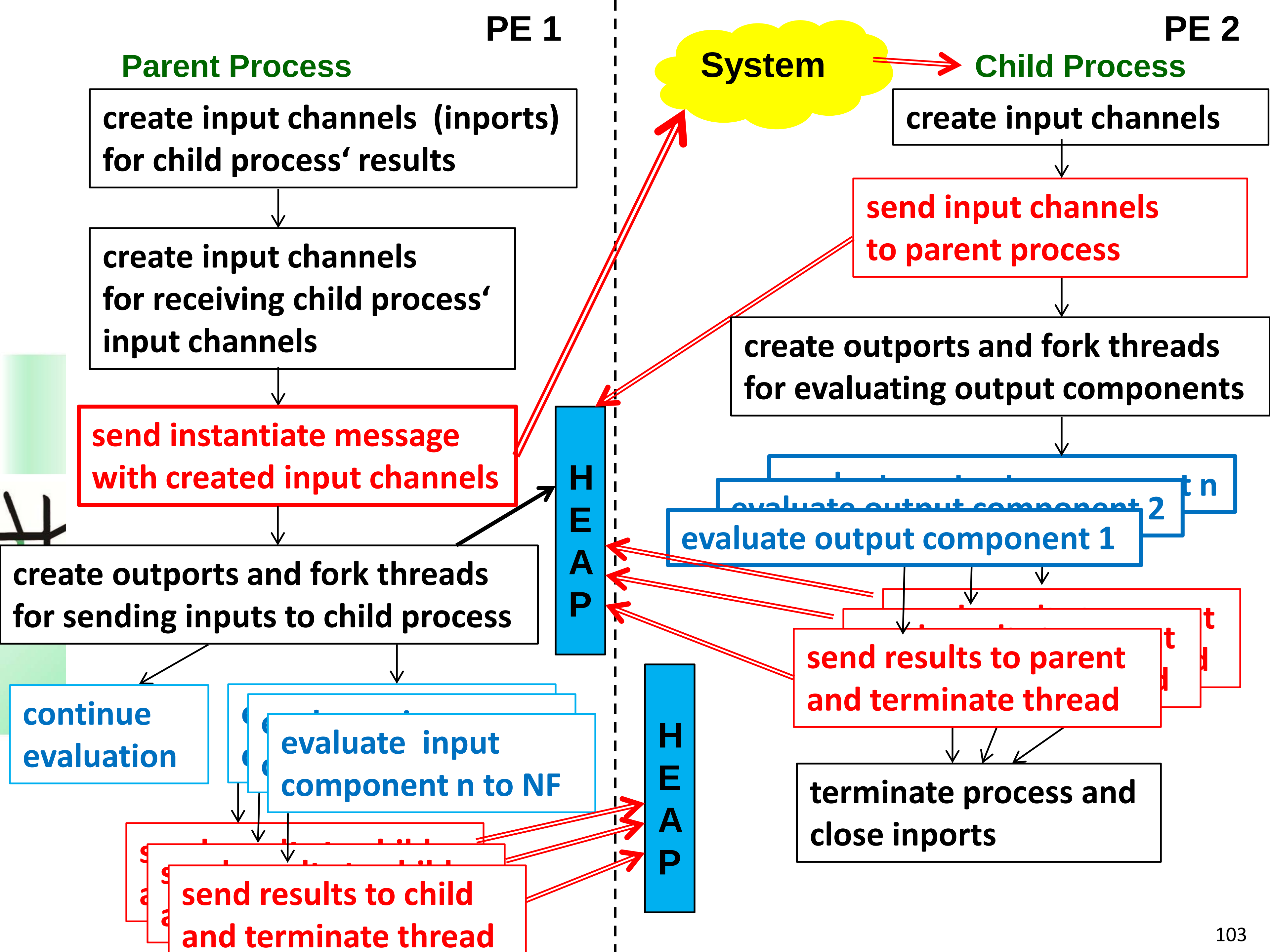
Process Output

Process Instantiation

```
( # ) :: (Trans a, Trans b) => Process a b -> a -> b
pabs # inps
  = unsafePerformIO $ instantiateAt  0  pabs inps

instantiateAt :: (Trans a, Trans b) =>
                 Int -> Process a b -> a -> IO b
instantiateAt pe (Proc f_remote) inps
 = do  (sendresult, result)   <- createComm
       (inCC, Comm sendInput) <- createC
       sendData  (Instantiate pe)
                 (f_remote sendresult inCC)
       fork  (sendInput inps)
       return result
```

output channel(s)

channel for returning input channel handle(s)



PE 1

Parent Process

create input channels (inports) for child process‘ results

create input channels for receiving child process‘ input channels

send instantiate message with created input channels

create outports and fork threads for sending inputs to child process

continue evaluation

evaluate input component n to NF

send results to child and terminate thread

HEAP

HEAP

PE 2

System

Child Process

create input channels

send input channels to parent process

create outports and fork threads for evaluating output components

evaluate output component n

evaluate output component 2

evaluate output component 1

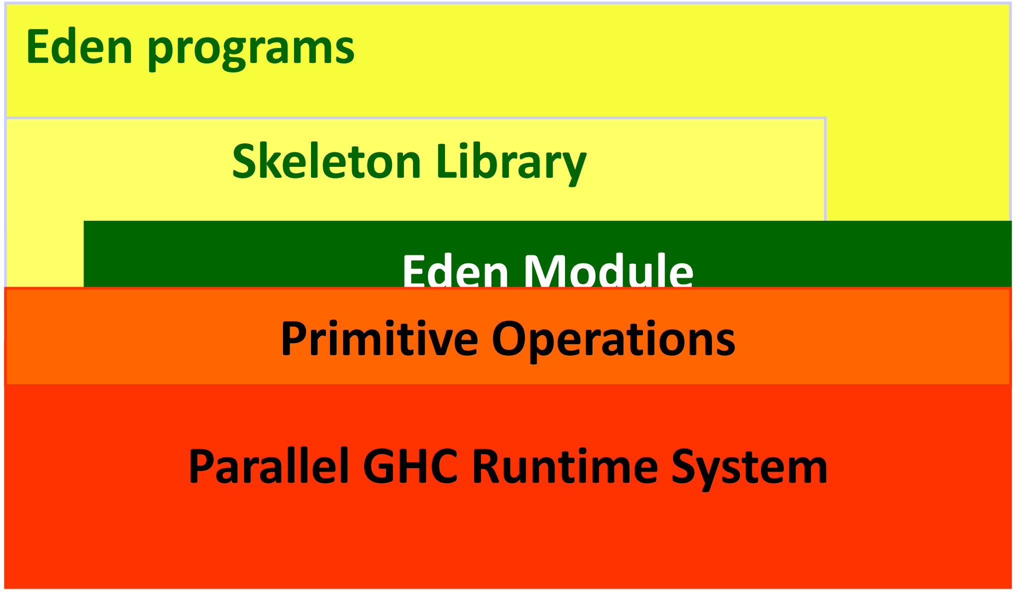
send results to parent and terminate thread

terminate process and close inports

Conclusions of Lecture 3

Layered implementation of Eden

- More flexibility
- Complexity hidden
- Better Maintainability
- Lean interface to GHC RTS



Conclusions

www.informatik.uni-marburg.de/~eden

- Eden = Haskell + Coordination
- Explicit Process Definitions
- Implicit Communication (Message Transfer)
- Explicit Channel Management
  -> arbitrary process topologies
- Nondeterministic Merge
  -> master worker systems with dynamic load balancing
- Remote Data
  -> pass data directly from producer to consumer processes
- Programming Methodology: Use Algorithmic Skeletons
- EdenTV to analyse parallel program behaviour
- Available on several platforms

More on Eden

PhD Workshop tomorrow 16:40-17:00

Bernhard Pickenbrock:
Development of a multi-core implementation of Eden