Philipps Universität Marburg

Skeleton-Based Parallel Programming in Eden

Rita Loogen

Philipps-Universität Marburg, Germany

Joint Work with:

Mischa Dieterle and Thomas Horstmeyer
(Philipps-Universität Marburg)

Jost Berthold
(University of Copenhagen, Denmark)
Yolanda Ortega Mallén and Lidia Sánchez-Gil
(Universidad Complutense de Madrid, Spain)

Overview

- Motivation and Basics
- Algorithmic Skeletons
  - Parallel map implementations
  - Divide and Conquer
- Skeleton Composition
  - Remote data concept
  - Parallel map – parallel reduce
  - Implementing PSRS in Eden
- Conclusions

- Lab Notes

Motivation

Parallel programming at a high level of abstraction

parallelism control

- explicit processes
- implicit communication
- distributed memory
- non-functional features
  - remote data
  - many-to-one communication

+

functional language (→ Haskell)

=> concise programs

=> high programming efficiency

=> higher-order functions

=> laziness

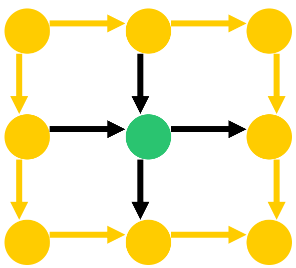
Eden = Haskell + Parallelism

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

Eden = Haskell + Parallelism

parallel programming at a high level of abstraction

- process definition

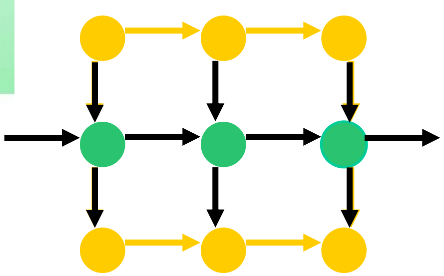


```
process   ::      (Trans a, Trans b) =>  (a -> b) -> Process a b

gridProcess   =     process  gridFunction
gridFunction  (fromLeft,fromTop) =  (toRight, toBottom))
         where       toRight = ...
                     toBottom = ...
```

process outputs computed by concurrent threads, lists sent as streams

- eager creation of processes



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

(outEasts,outSouths) = unzip $
                         spawn (repeat gridProcess)
                              (zip inNorths (inWest:outEasts))
outEast = last outEasts
```

The Eden Module: Control.Parallel.Eden

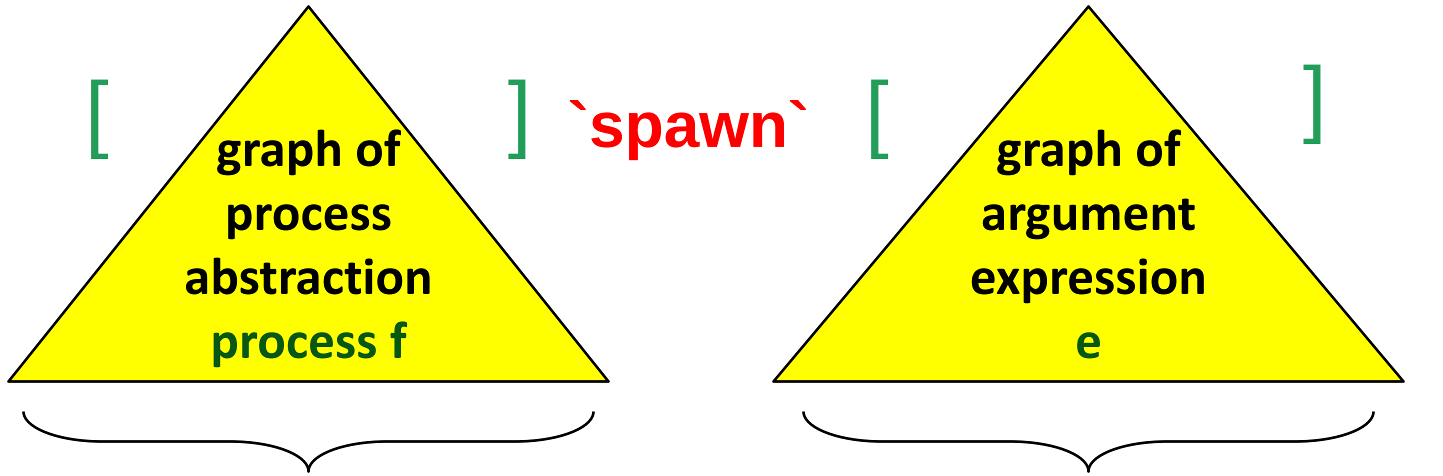
definitions of
process, Process and **spawn**

more features like
e.g. **remote data**

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

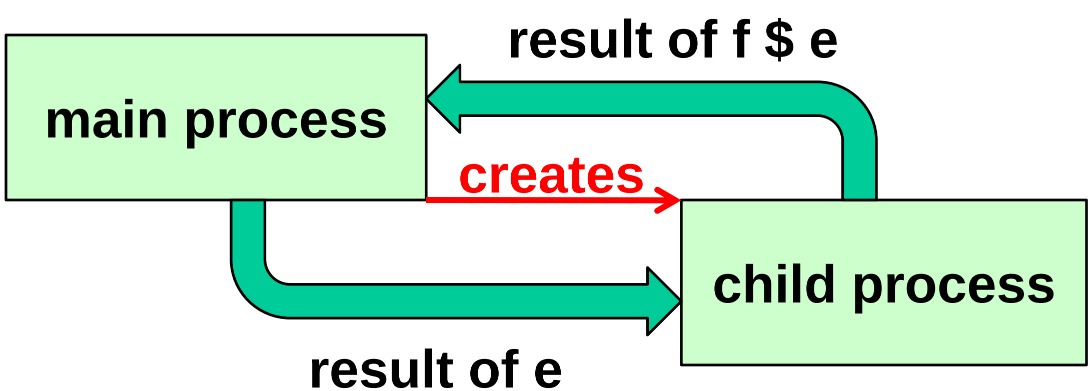
Definition of type class **Trans** which
- contains transmissible data types (most pre-defined types)
- defines (implicitly used) communication functions
  overloaded for lists (-> streams) and tuples (-> concurrency)

Evaluating spawn [process f] [e]



thread(s) in new child process will evaluate(s) the application (f $ e) to normal form and send(s) the result to the parent process

new concurrent thread (s) in parent process evaluate(s) e to normal form and send(s) the result to the child process

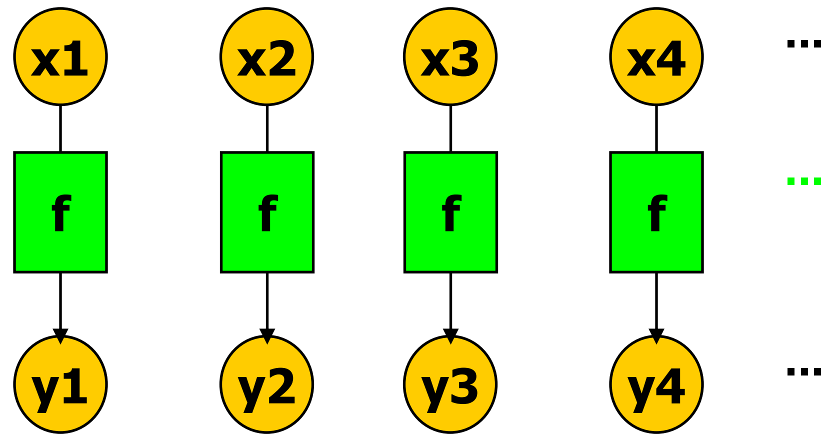


Lazy evaluation vs. Parallelism

- **Problem:** Lazy evaluation ==> distributed sequentiality
- Eden‘s approach:
  - eager process creation with spawn
    - default round robin process placement
    - explicit process placement using spawnAt :: [Int] -> …
  - 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 using sequential strategies (Module Control.Seq):
    - rnf :: NFData a => Strategy a
    - pseq :: a -> b -> b (Module Control.Parallel)

A Simple Parallelisation of map

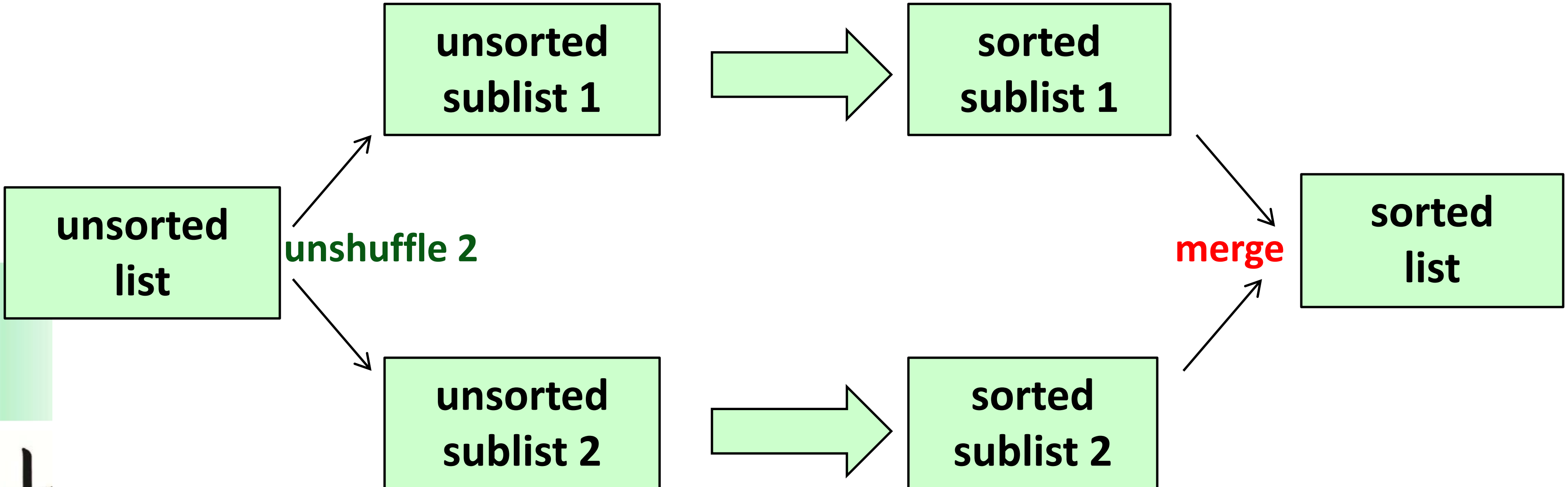
```
map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]
```




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

1 process per list element

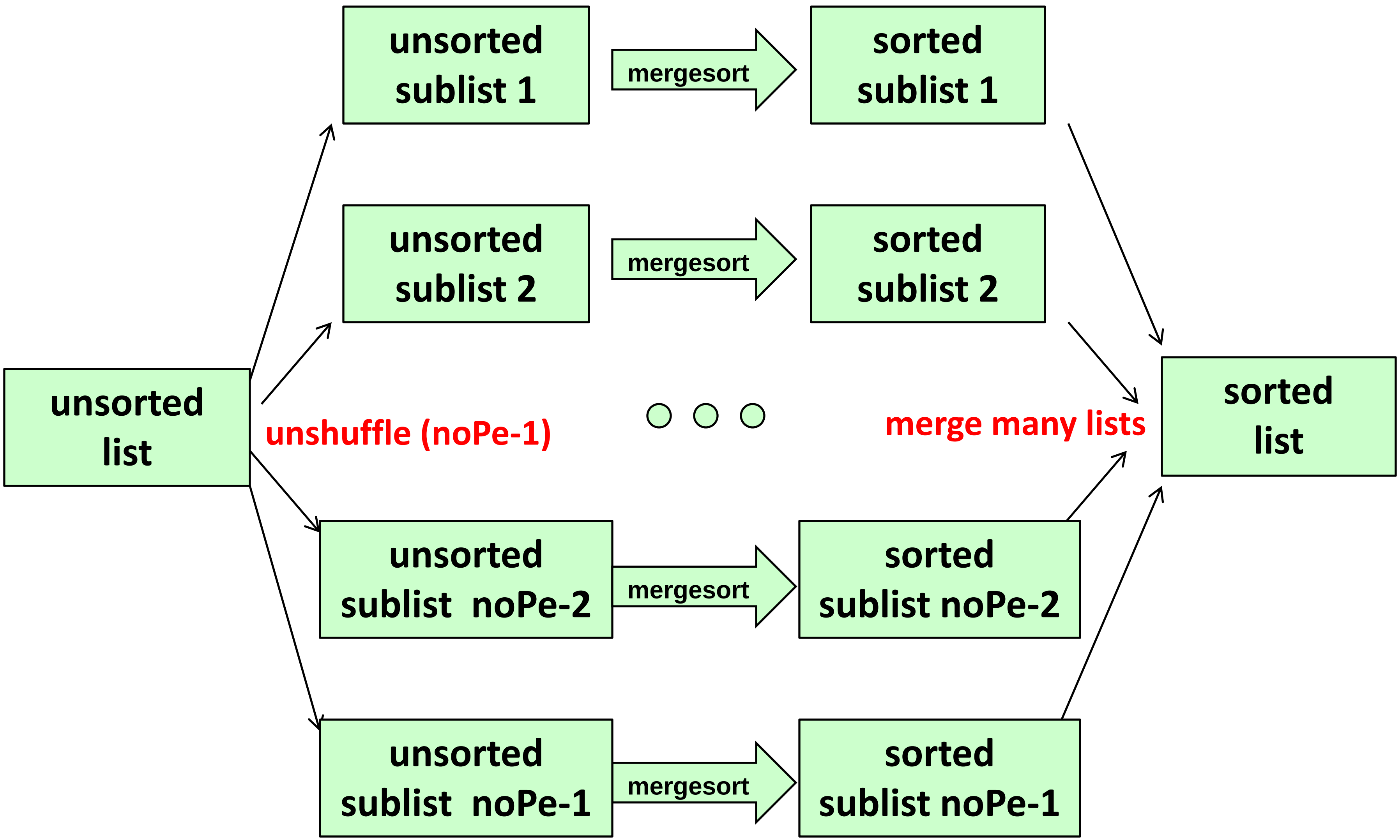
Case Study: Merge Sort



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
```

Parallel Mergesort Using parMap



Eden Code

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

```
mergeAll ::  Ord a => [[a]] -> [a]
mergeAll [xs] = xs
mergeAll xss  = mergeAll (mergePairs xss)

mergePairs :: Ord a => [[a]] -> [[a]]
mergePairs (xs1:xs2:xss)
      = sortMerge xs1 xs2 : mergePairs xss
mergePairs xs = xs
```

→ Total number of processes = noPe

→ eagerly created processes

→ round robin placement leads to 1 process per PE

Eden Program

```
module Main where

import Control.Parallel.Eden
import Control.Parallel.Eden.Map (parMap)
import Control.Parallel.Eden.Auxiliary (unshuffle)
import System.Environment (getArgs)
import System.Random

main :: IO ()
main =  do ins <- getArgs
           let (v:a:xs) = ins
           let rs = randomlist (read a :: Int) 42
           putStrLn (rnf rs `pseq` rnf (ms v rs) `pseq` "Done")

ms :: String -> [Int] -> [Int]
-- sequential mergeSort
ms "seq" xs  = mergeSort xs
-- simple parMap
ms "parMap" xs
  = mergeAll $ parMap mergeSort (unshuffle (noPe-1) xs)
...
```

Compiling, Running, Analysing Eden Programs

1. **Compile** Eden programs on **multicores** with

   ghceden –parcp --make –O2 –eventlog myprogram.hs

   and on **clusters** or multicores with

   ghceden –parmpi --make –O2 –eventlog myprogram.hs

   or **ghceden –parpvm --make –O2 –eventlog myprogram.hs**

2. **Run** compiled programs with

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

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

3. **Analyse** eventlog (trace file) with

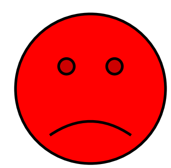
   edentv myprogram_..._-N4_-RTS.parevents

Experimental Results

- For all measurements in this lecture, I have used ghc-eden-7.8.2 on a 64-core machine:

  4 x AMD Opteron(tm) Processor 6378
  (16 Cores, 16MB L3-Cache, 2,4 GHz)
  64 GB DDR3 SDRAM, 1600 MHz

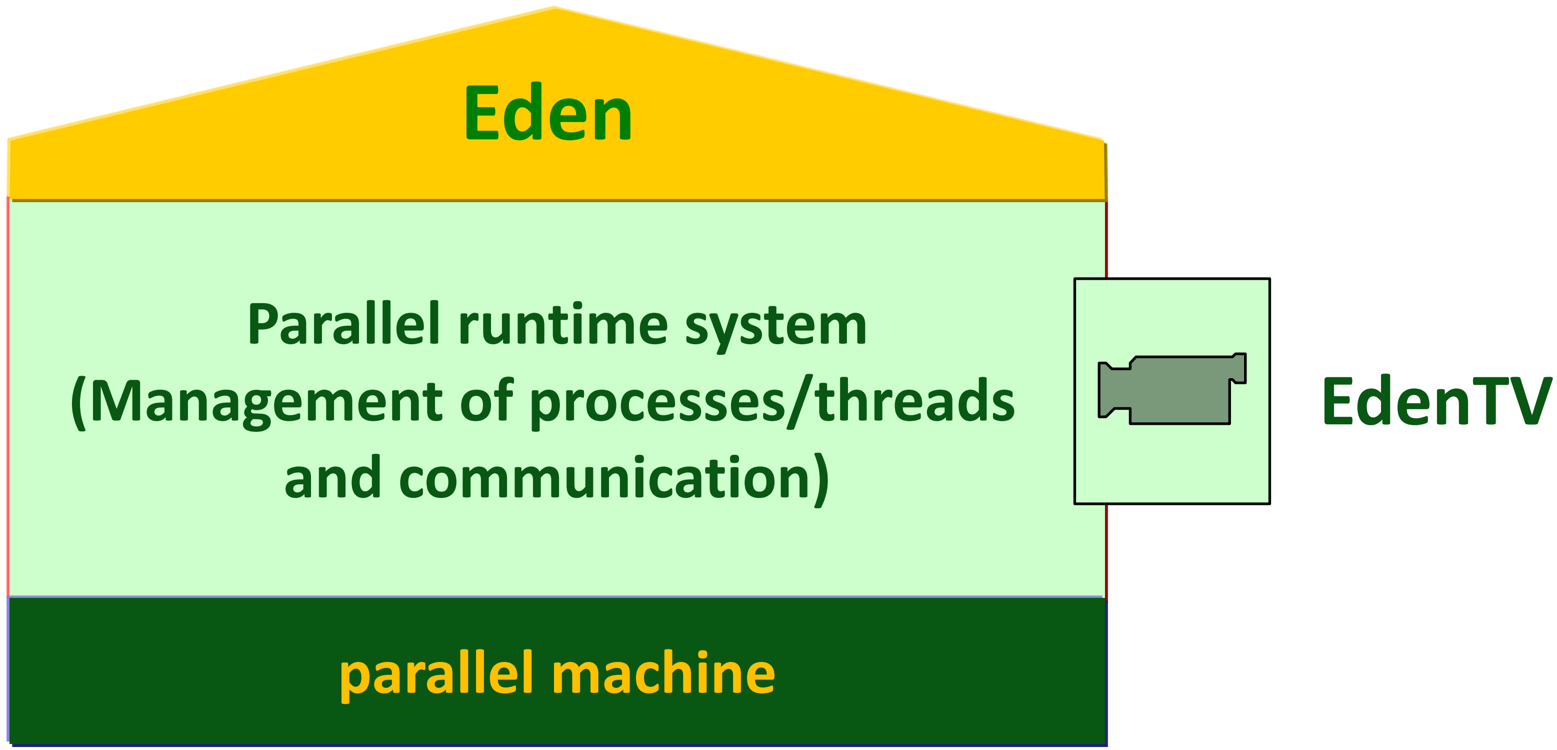
- Runtime Results for parMap-mergesort on 8 cores:
  - Input size 5.000
  - seq. runtime: 0,020 s
  - par. runtime: 0,103 s



SLOWDOWN

- What is going wrong? Use EdenTV to analyse program behaviour.

Eden-TV



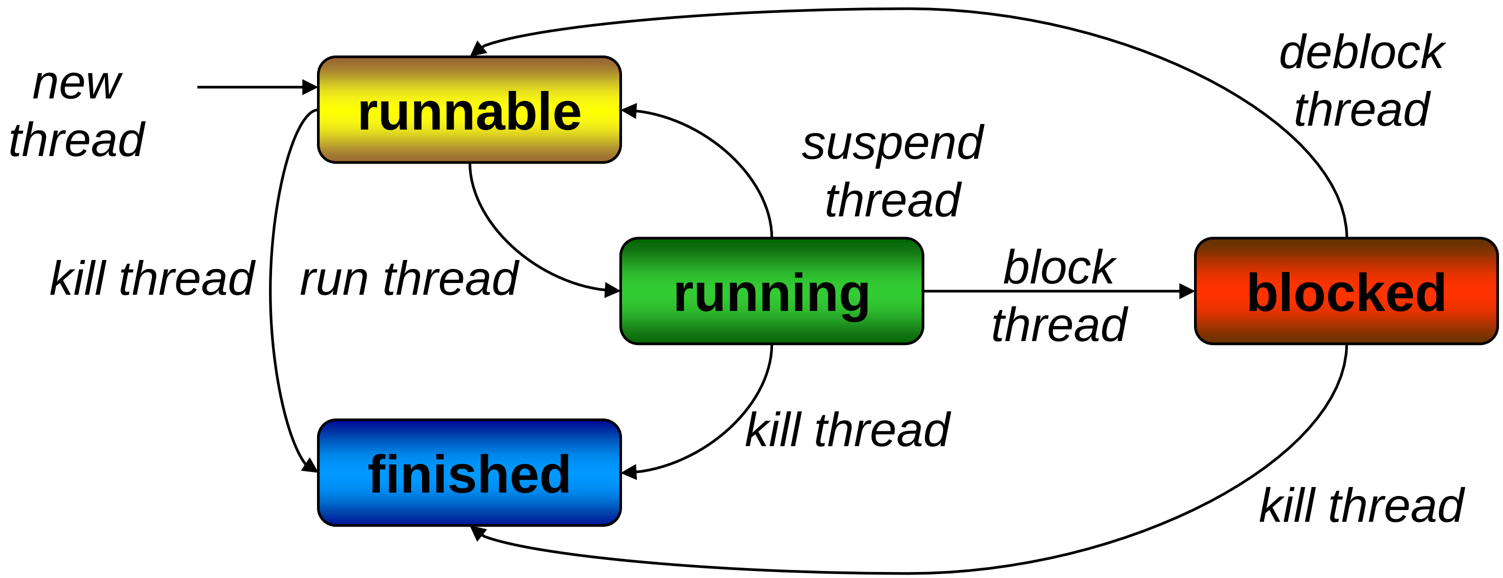
EdenTV provides

- **four different views (activity profiles)**
  Machines (PEs) - Processes - Threads - Processes/Machine
- **message overlays (except for thread profiles)**
- **zooming**

...

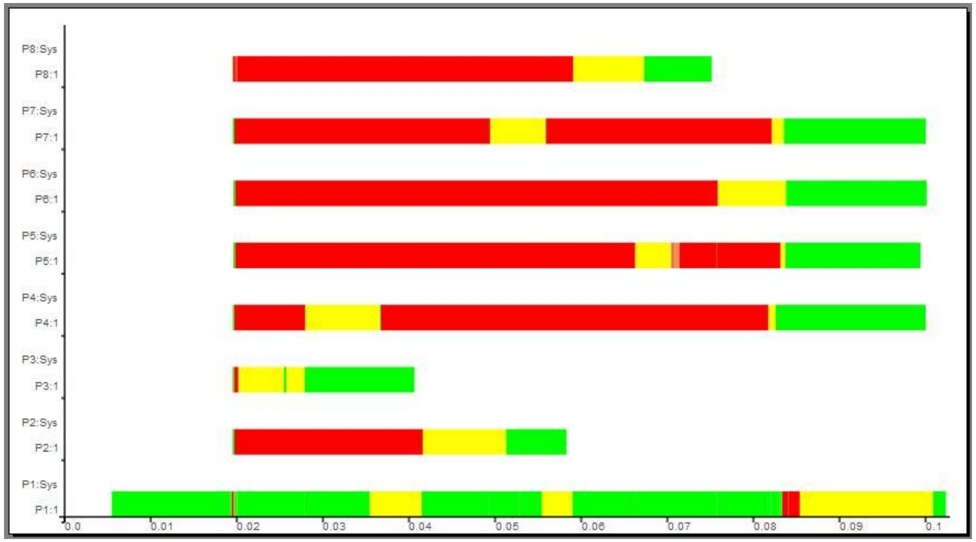
Colour Code Used in Activity Profiles

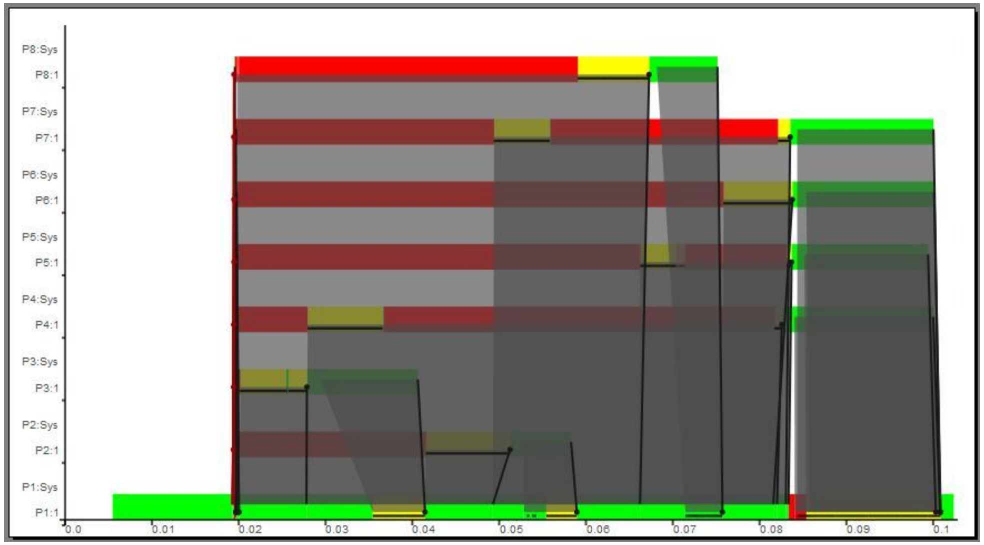
- **An Eden process consists of several threads (one per output channel).**
- **Thread State Transition Diagram:**



- **States of processes and machines are derived from thread states**

EdenTV Activity Profile of Parallel MergeSort (Processes/Machine View)





- **Input size: 5.000**
- **seq. runtime: 0,020 s**
- **par. runtime: 0,103 s**

  SLOWDOWN

- **Additional Infos by EdenTV**
  - **8 Pes**

    8 processes

    15 threads
  - **42 conversations**

    10042 messages

Reason for Slowdown:

Too many messages ?

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
```

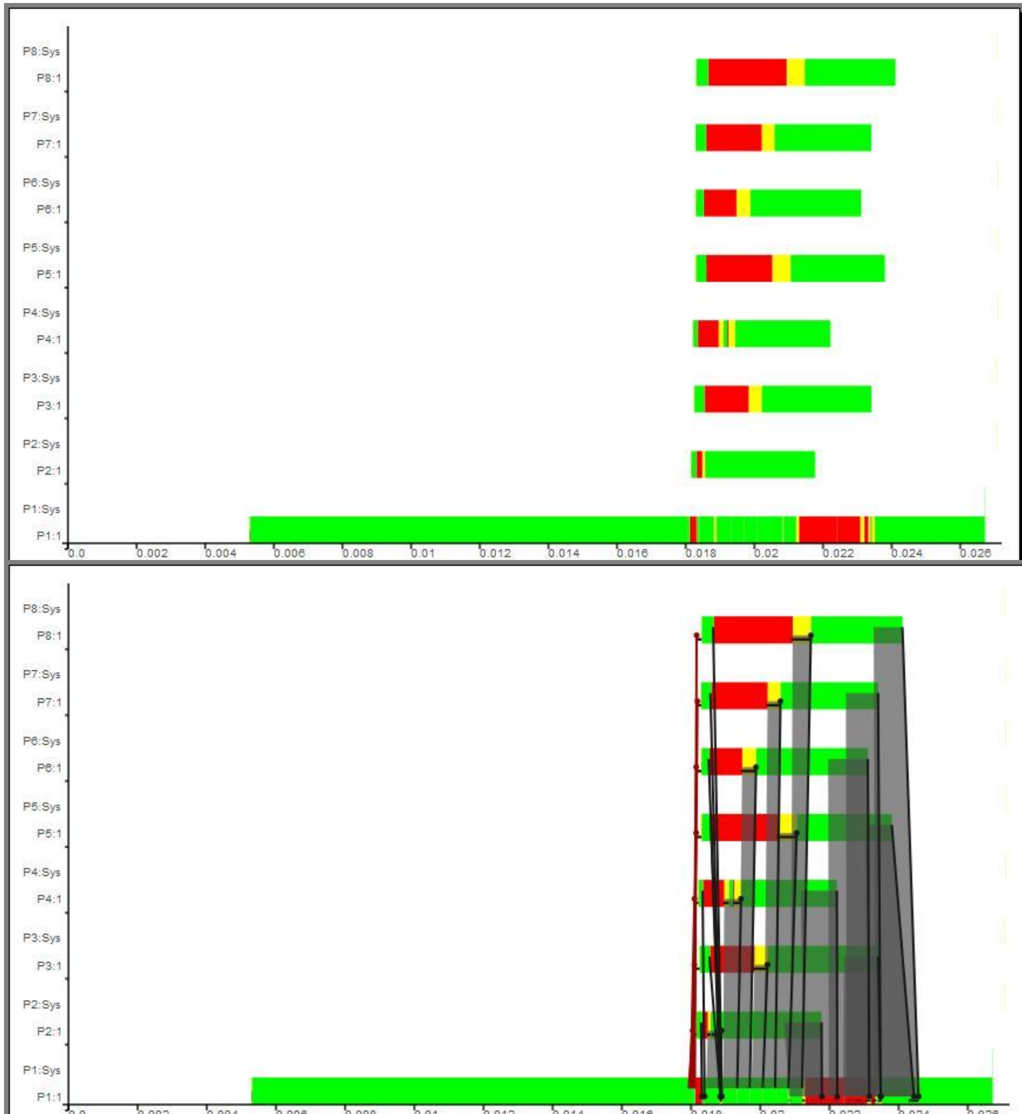
Combine with parallel map-implementation of mergesort:

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

concat = unchunk

Resulting Activity Profile (Processes/Machine View)

Previous results for input size 5000
Seq. runtime: 0,020 s
Par. runtime I: 0,103 s



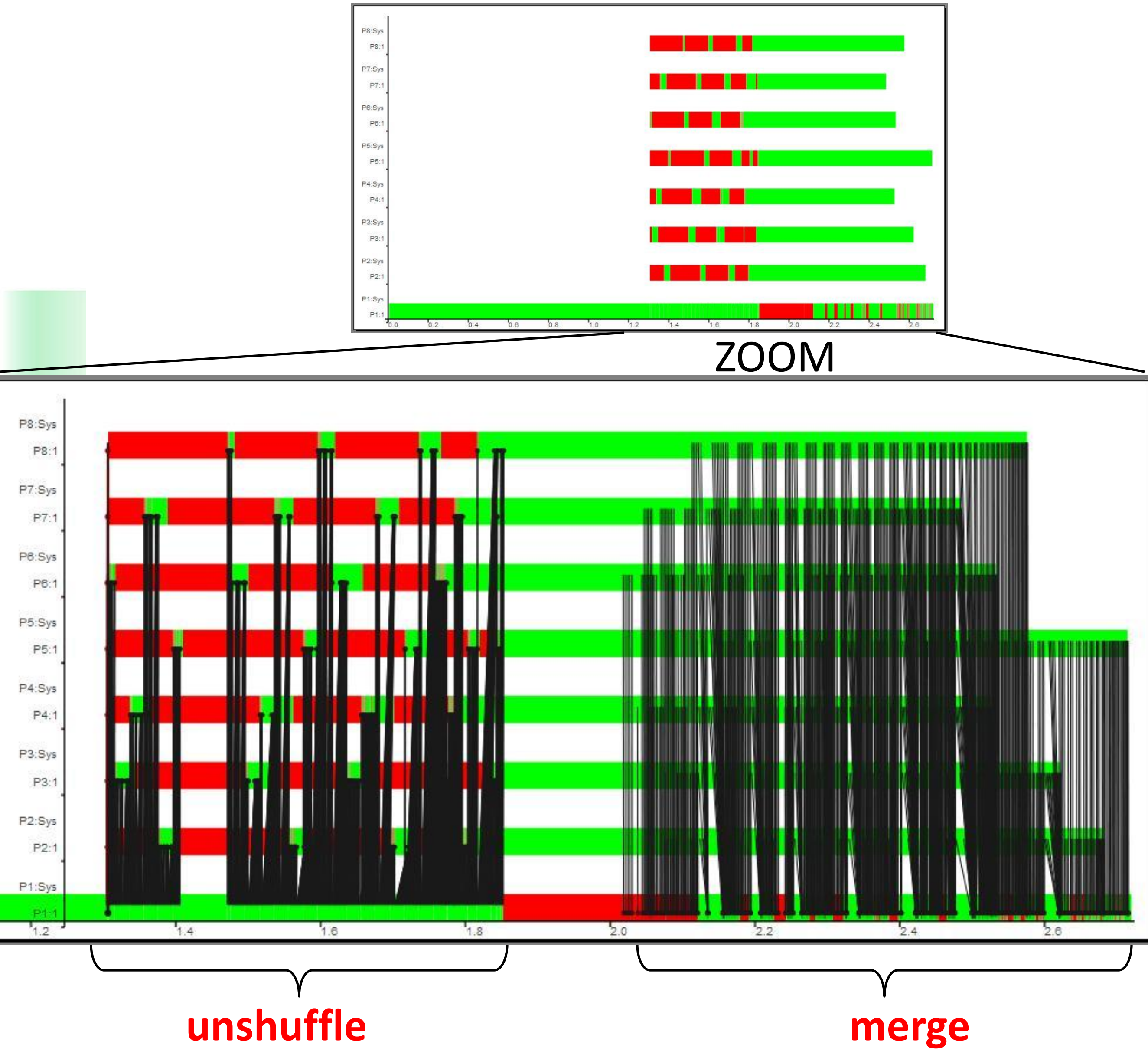
- Input size: 5.000
- Chunk size: 50
- par. runtime: 0,027 s

- Additional Infos by EdenTV
  - 8 Pes
    8 processes
    15 threads
  - 42 conversations
    ~~10042~~ 252 messages

Much better, but still
SLOWDOWN

Time for
- parallel system start up (0,005s)
- random list generation (0,016 s)
dominates runtime.

Activity Profile for Input Size 1.000.000

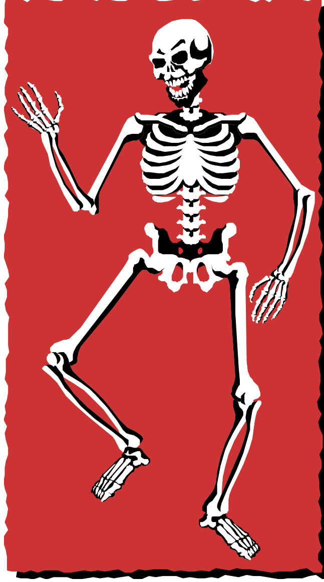


- **Input size 1.000.000**
- **Chunk size 1000**
- **seq. runtime: 6,827 s**
- **list generation: 1,18 s**
- **par. runtime: 2,724 s**

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

→ speedup of parallel sort is 3.66 on 8 PE

Algorithmic Skeletons



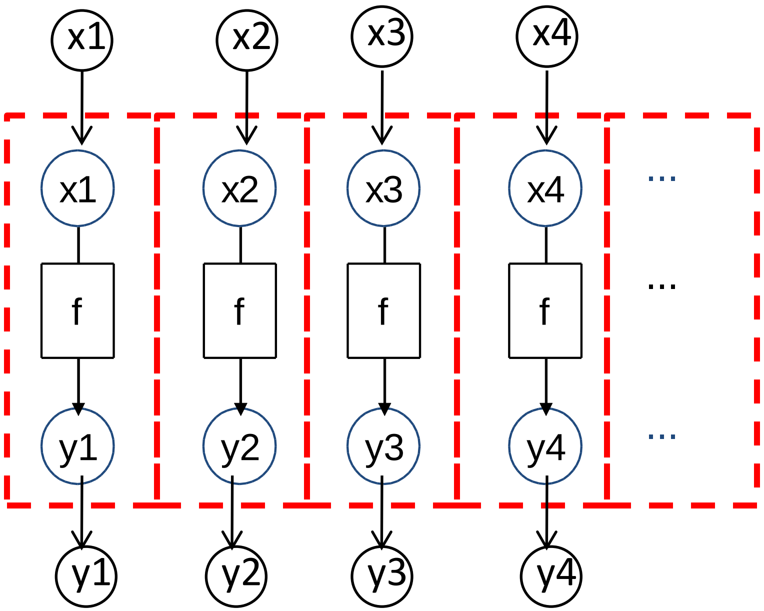
Algorithmic Skeletons

- **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**
  - **divide and conquer**
  - **topology skeletons: pipeline, ring, torus, grid, trees …**

See Eden‘s
Skeleton Library
Control.Parallel.Eden.<...>
with <…> in Map, MapReduce, DivConq, Topology, Workpool, Iteration

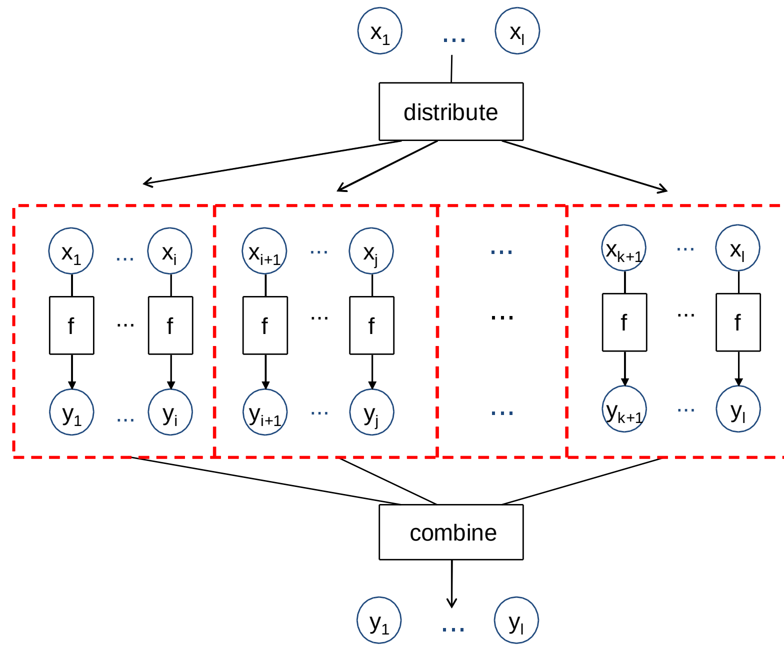
Parallel map implementations: parMap vs farm

parMap




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

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))
```

Distribution and Collection Functions

```
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  np / combine = shuffle`
- `distribute = splitIntoN np / combine = concat`

leading to alternative parallel maps implementations:

```
mapFarmS, mapFarmB ::
        (Trans a, Trans b) =>
        (a -> b) -> [a] -> [b]
mapFarmS = farm (unshuffle  (max (noPe-1) 1)) shuffle
mapFarmB = farm (splitIntoN (max (noPe-1) 1)) concat
```

1 process per PE

Reducing Communication Costs in Skeletons

Techniques:

1. **Chunking**
2. **Offline Processes**

Combine Chunking with Parallel Map:

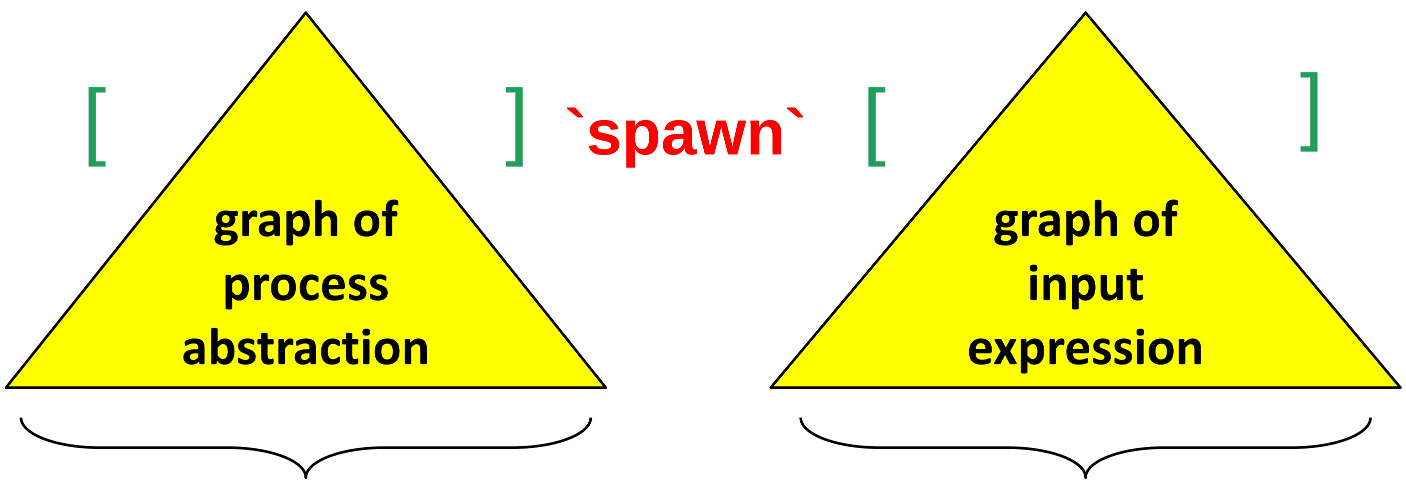
```
chunkMap :: Int  -> (([a] -> [b]) -> ([[a]] -> [[b]]))
                 -> (a -> b) -> [a] -> [b]
chunkMap chunksize mapscheme f xs
  = concat (mapscheme (map f) (chunk chunksize xs))
```

Communication vs Parameter Passing

Process inputs

- can be communicated: spawn [process f] [inp]
- can be passed as parameter spawn [process (\ () -> f inp)] [()]

  () is dummy process input

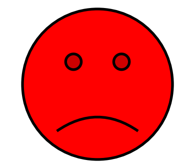


will be packed (serialised) and sent to remote PE where child process is created to evaluate the application of this expression to the input

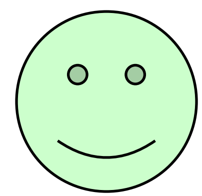
will be evaluated in parent process by concurrent thread and then sent to child process

Offline Processes and Skeletons

- Offline processes run without input or with a trivial input.
- This may cause redundant evaluations, because input expressions are copied without prior evaluation.



- This may save communication costs.

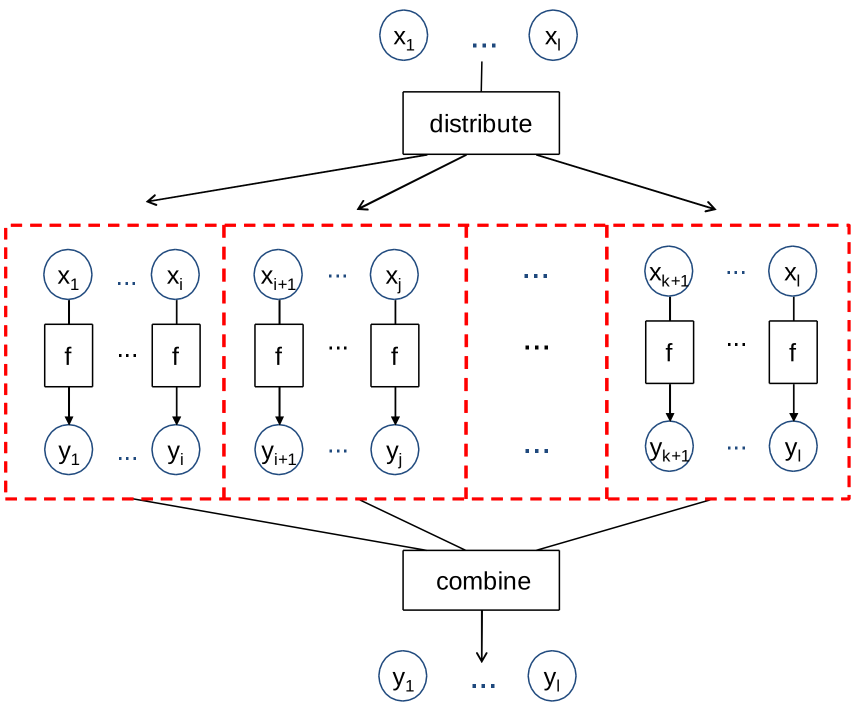


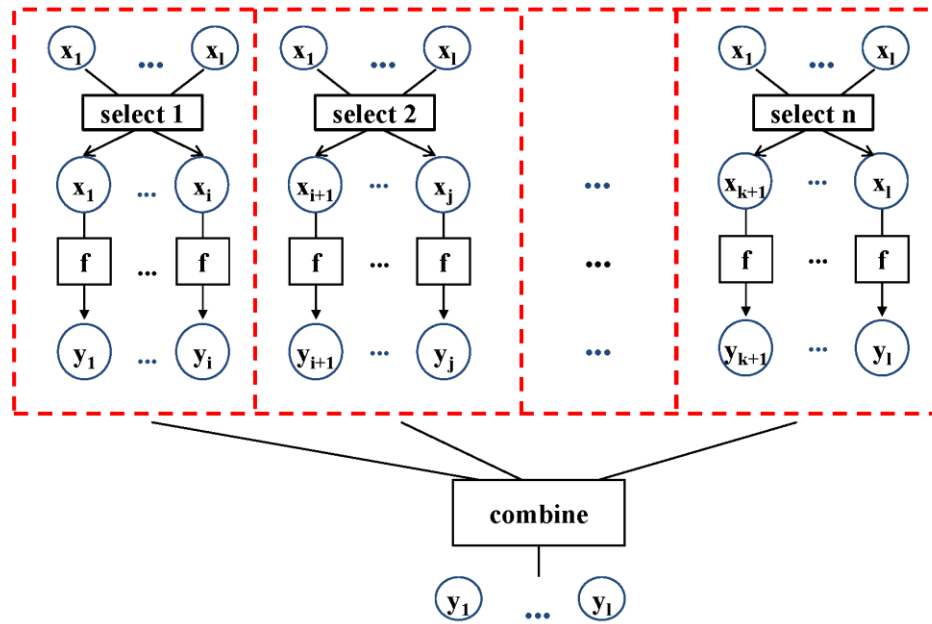
- Offline skeletons use offline processes.
- Offline skeletons are useful, if the input data is not yet evaluated.

Farm vs Offline Farm

```
farm :: (Trans a, Trans b) =>
        ([a] -> [[a]]) -> ([[b]] -> [b]) ->
        (a -> b) -> [a] -> [b]
```

```
offlineFarm :: (Trans a, Trans b) => Int ->
        ([a] -> [[a]]) -> ([[b]] -> [b]) ->
        (a -> b) -> [a] -> [b]
```





Suppress Streaming and/or Input Evaluation

- Streaming for lists or concurrent evaluation of tuples can be avoided by wrapping a box around the input expression:

```
newtype Box a = Box {unBox :: a}

instance Trans  a => Trans  (Box a)
instance NFData a => NFData (Box a)
 where rnf (Box x) = rnf x   -- normal form evaluation
```

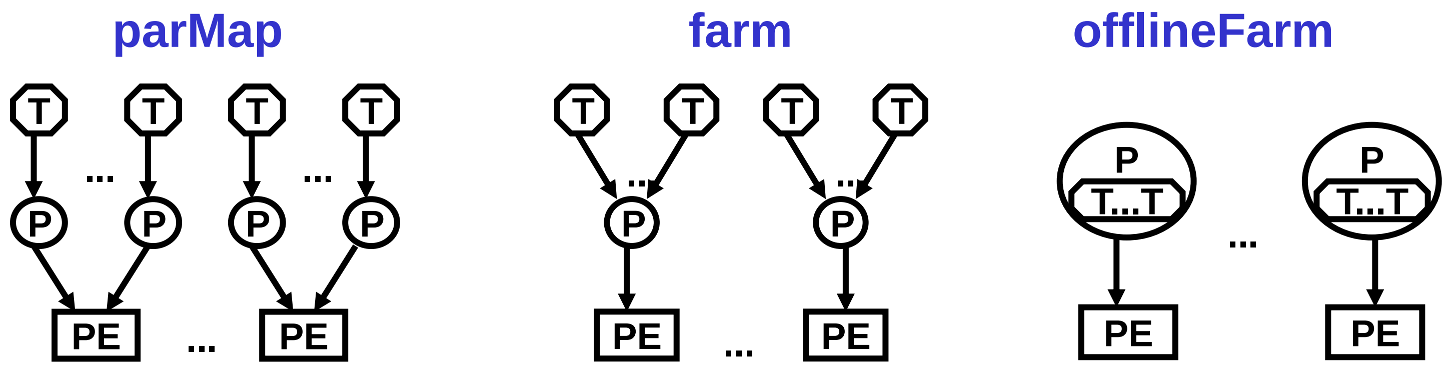
- A simple modification leads to lazy boxes which suppress the evaluation of input expressions before communication:

```
newtype LBox a = LBox {unLBox :: a}

instance Trans  a => Trans  (LBox a)
instance NFData a => NFData (LBox a)
 where rnf (LBox _) = ()     -- suppress evaluation
```

Parallel map implementations

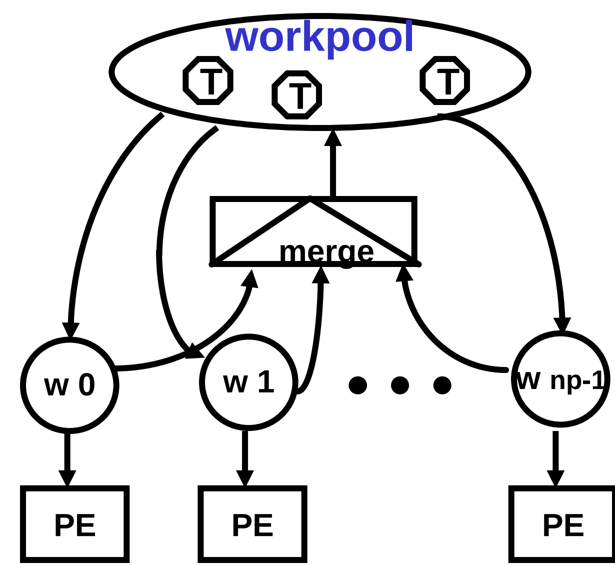
- **static** task distribution / **regular** task decomposition:



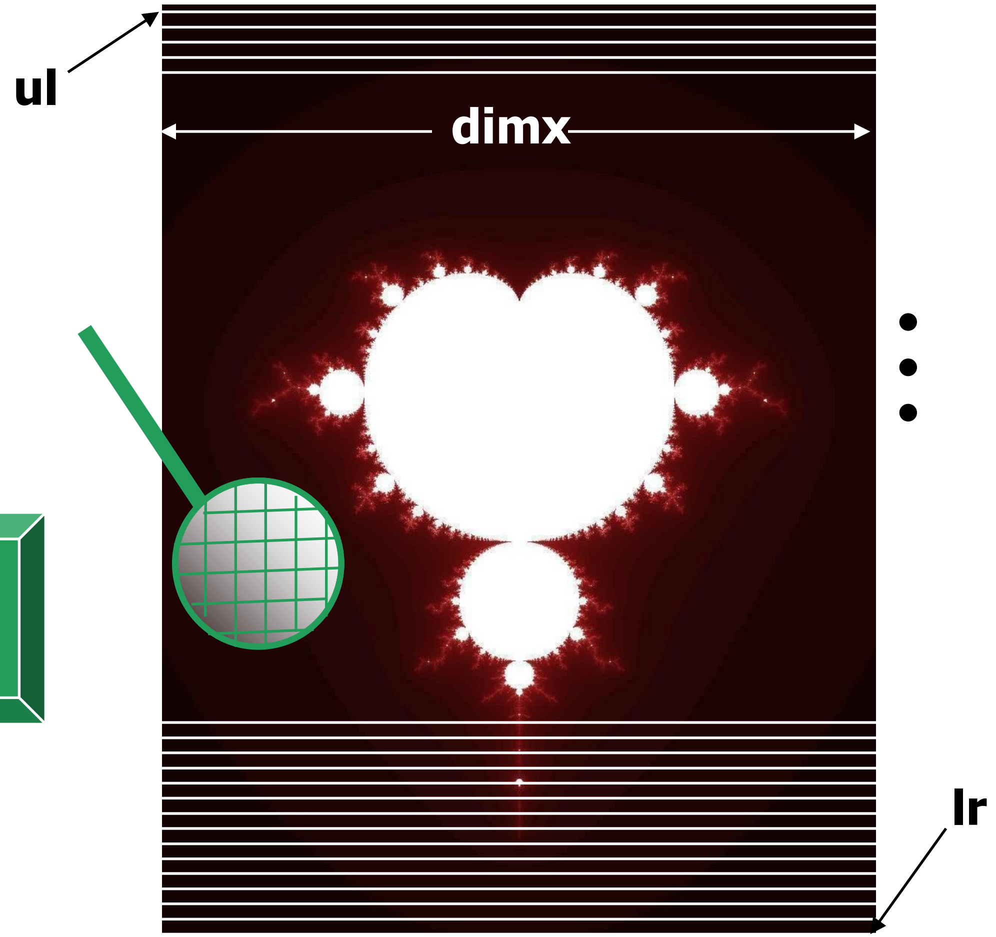
increasing granularity

- **dynamic** task distribution / **irregular** task decomposition:

```
workpoolSorted ::
     Int          -- number of workers
  -> Int          -- prefetch
  -> (a->b)       -- worker function
  -> [a]->[b]  -- input -> output
```



Example: Parallel Functional Program for Mandelbrot Sets



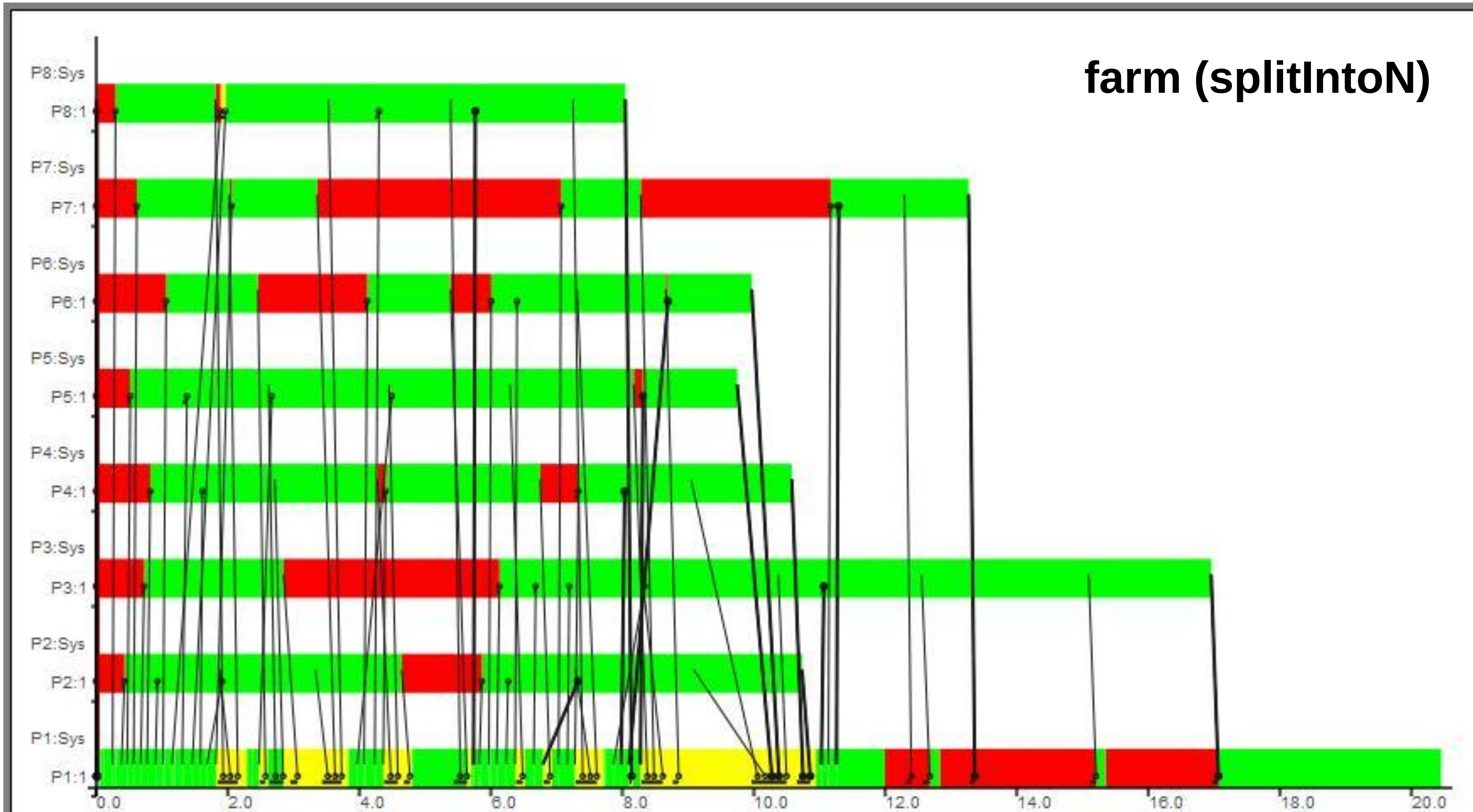
Idea: parallel computation of lines

```
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
parallel map implementation

Mandelbrot Traces

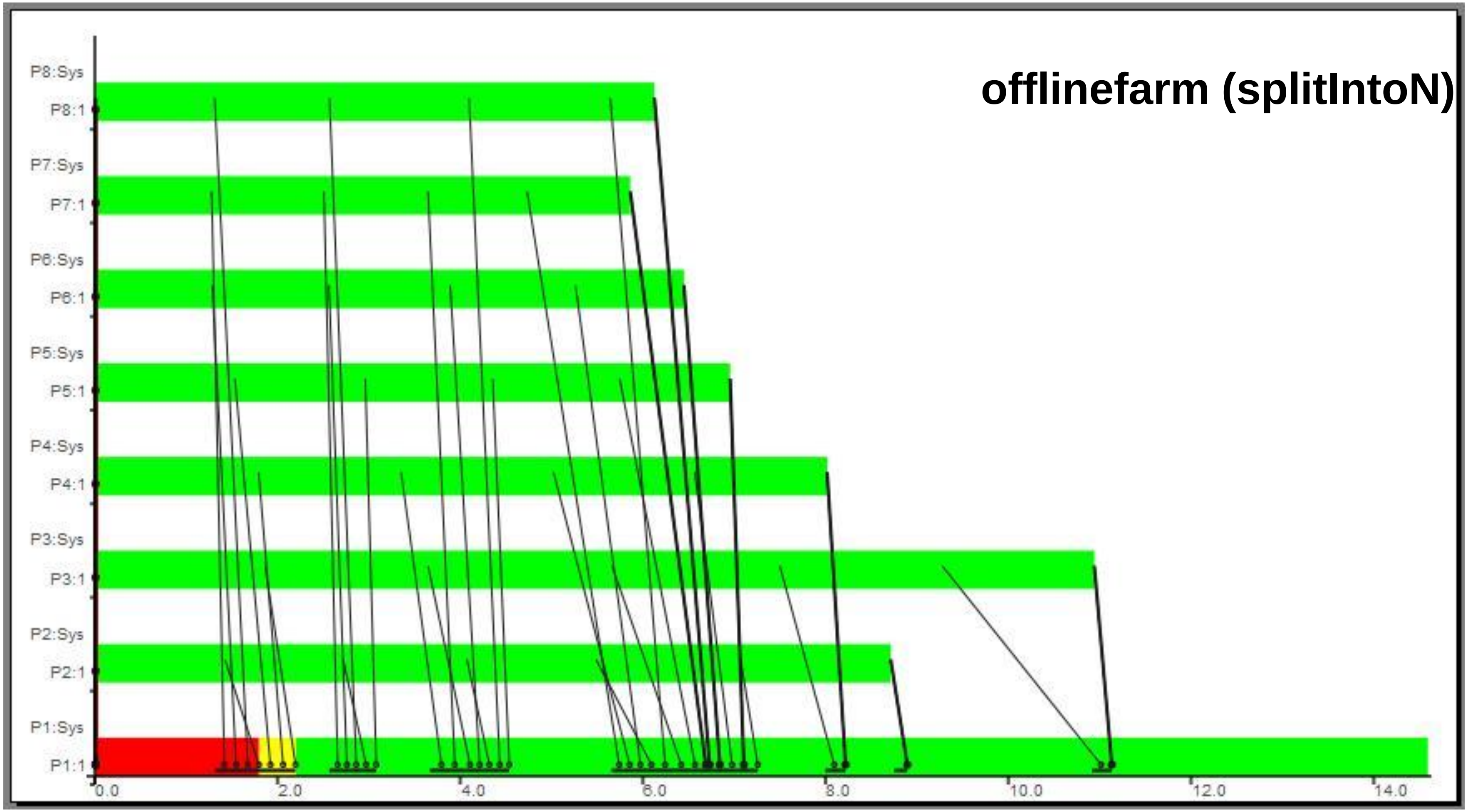
Problem size: 2000 x 2000
Chunking size: 50



20,622 s, 8 Machines, 8 Processes, 23 Threads, 42 Conversations, 116 Messages

Mandelbrot Traces

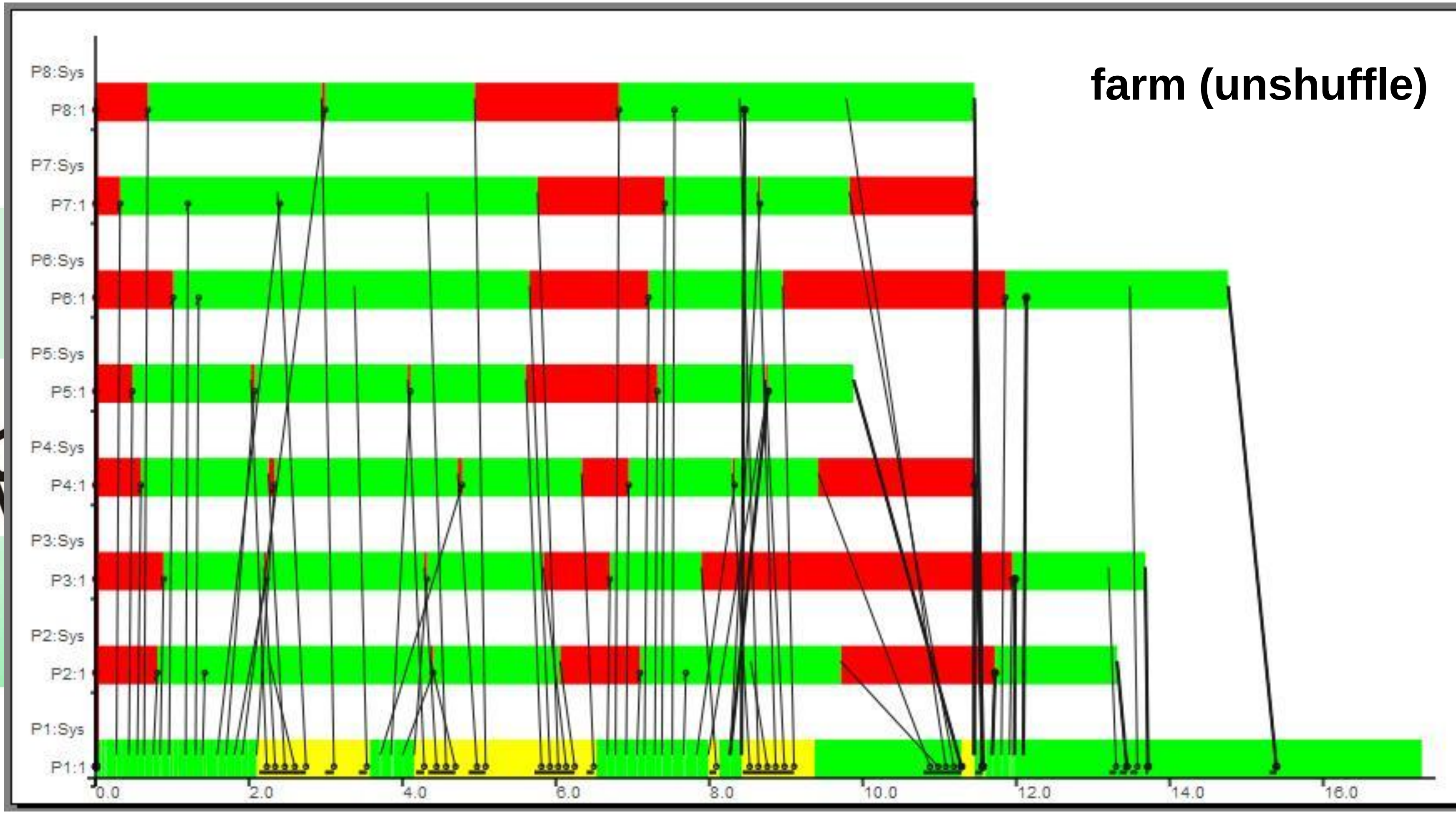
Problem size: 2000 x 2000
Chunking size: 50



14,630 s, 8 Machines, 8 Processes, 23 Threads, 35 Conversations, 72 Messages

Mandelbrot Traces

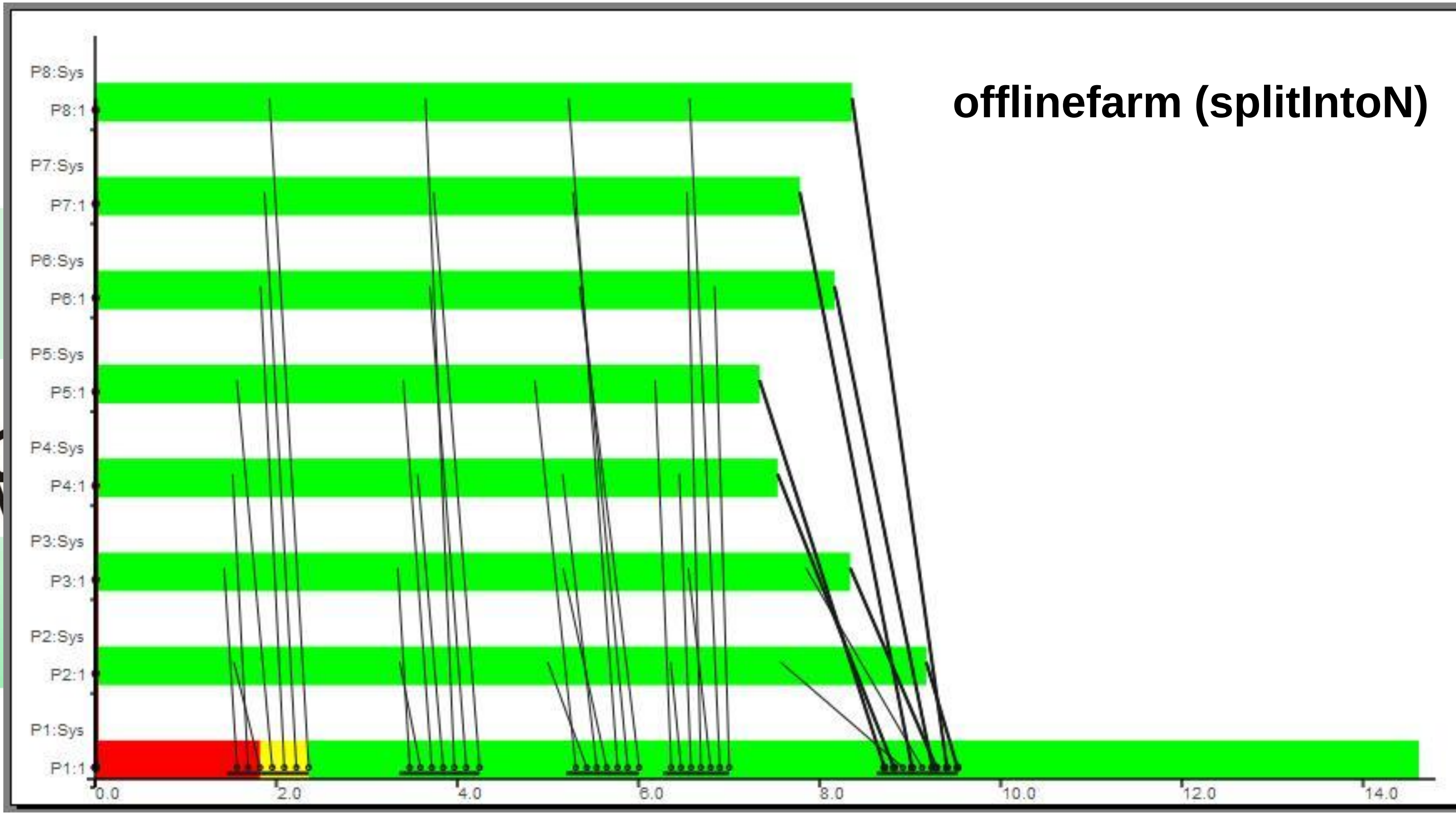
Problem size: 2000 x 2000
Chunking size: 50



17,464s, 8 Machines, 8 Processes, 23 Threads, 42 Conversations, 116 Messages

Mandelbrot Traces

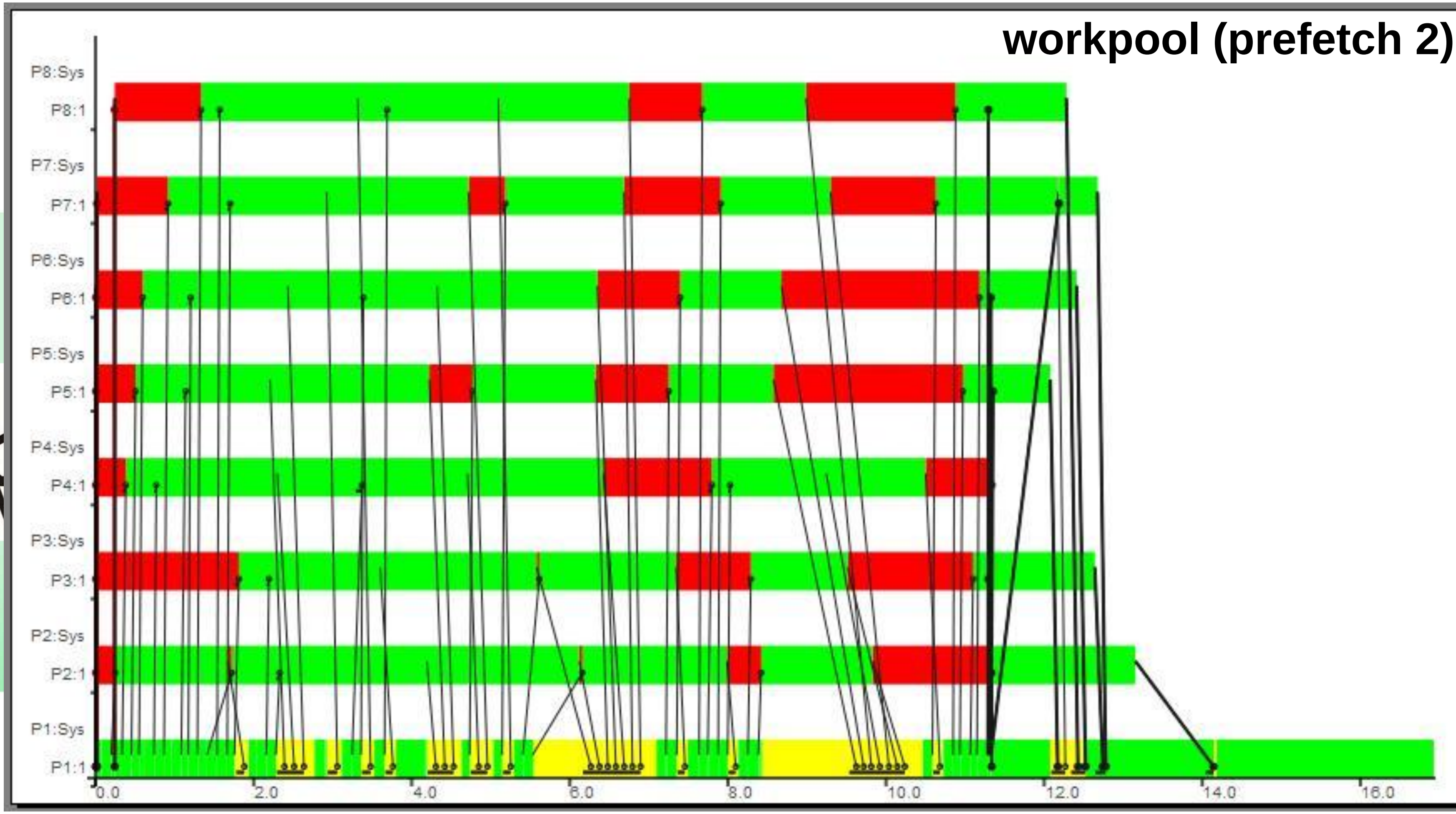
Problem size: 2000 x 2000
Chunking size: 50



14,800 s, 8 Machines, 8 Processes, 23 Threads, 35 Conversations, 72 Messages

Mandelbrot Traces

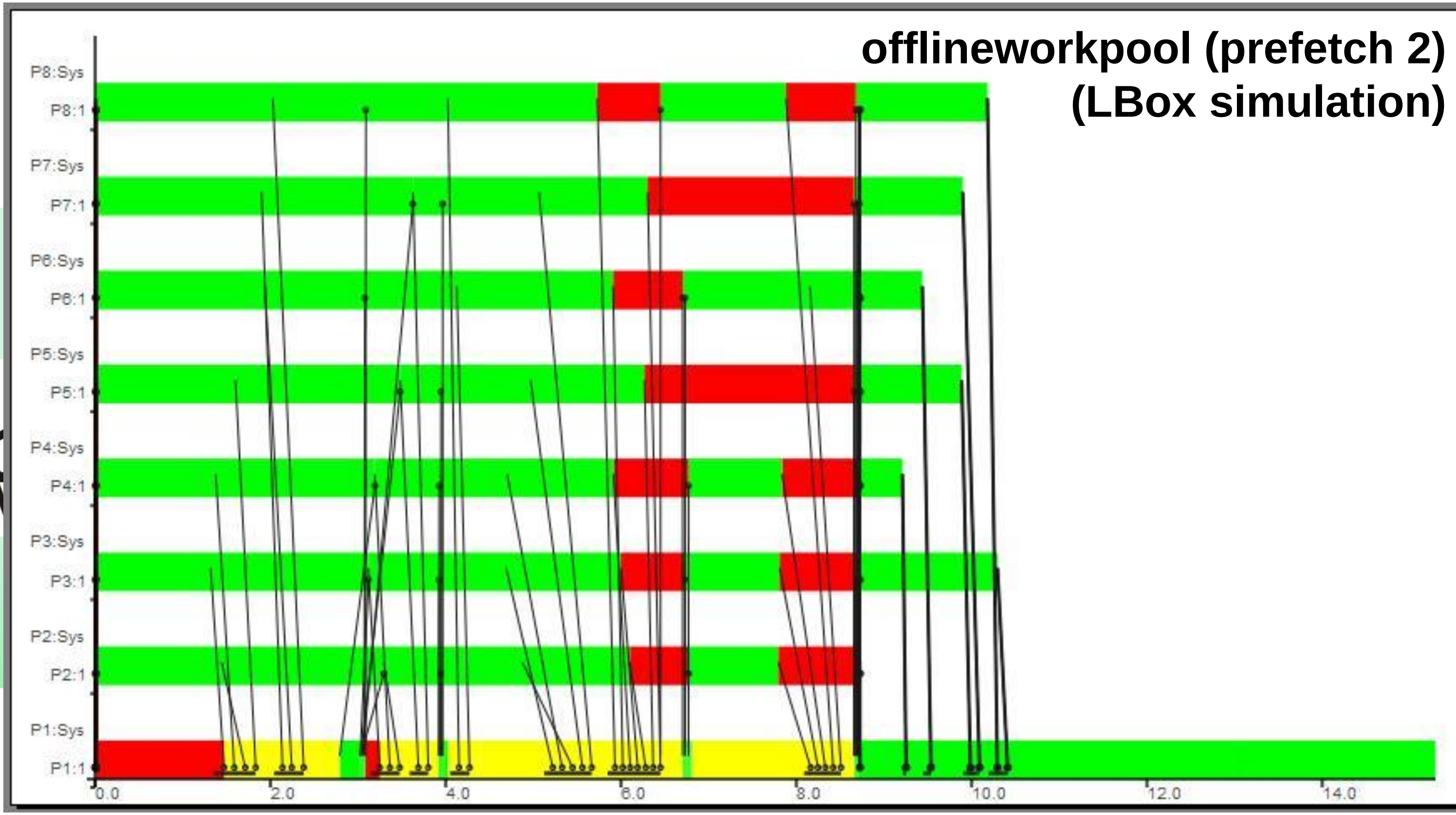
Problem size: 2000 x 2000
Chunking size: 50



16,951s, 8 Machines, 8 Processes, 30 Threads, 42 Conversations, 116 Messages

Mandelbrot Traces

Problem size: 2000 x 2000
Chunking size: 50

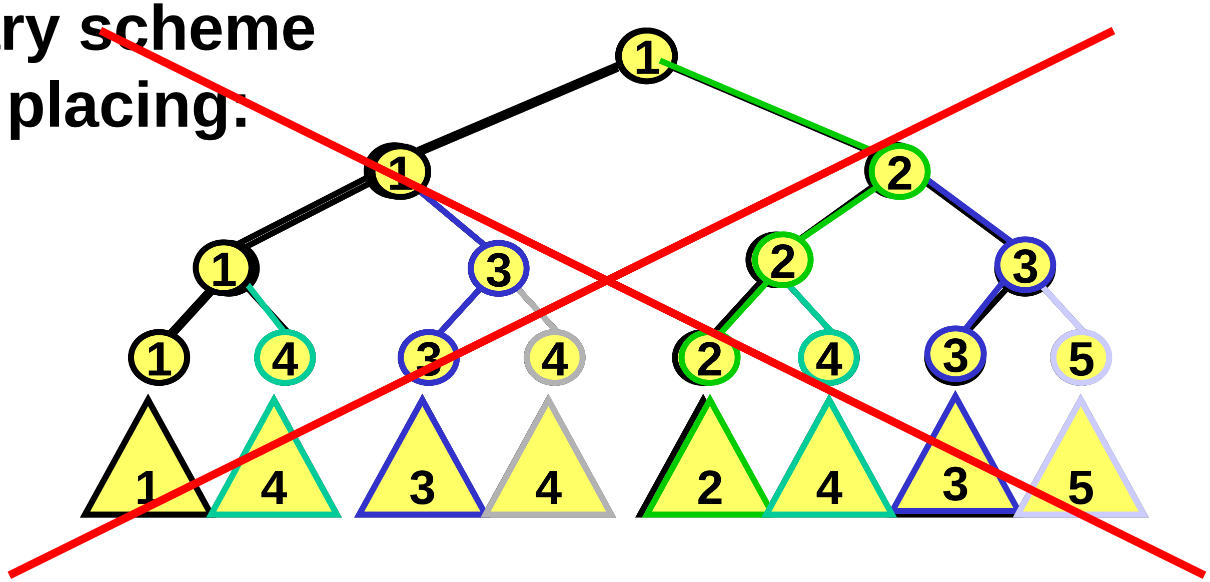


15,291s, 8 Machines, 8 Processes, 30 Threads, 42 Conversations, 116 Messages

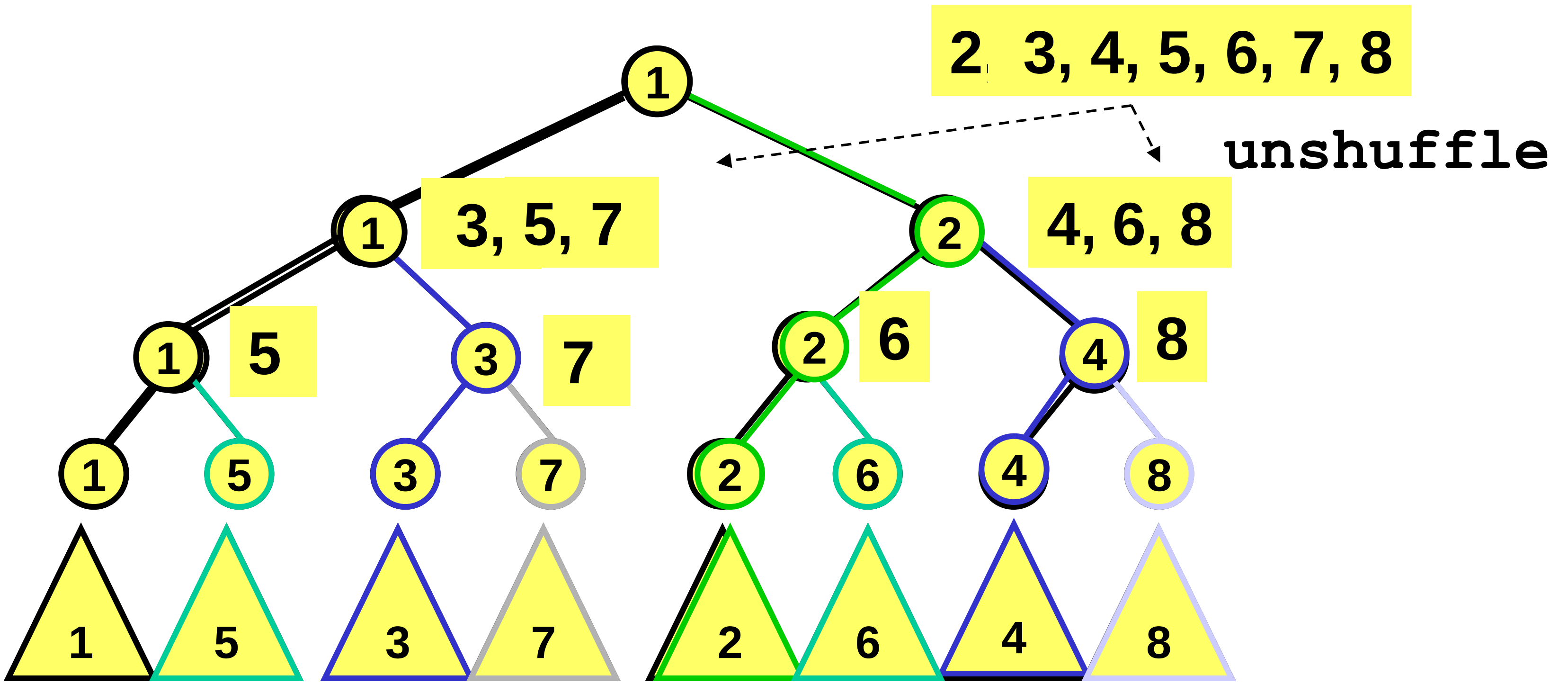
Divide-and-conquer

```
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:



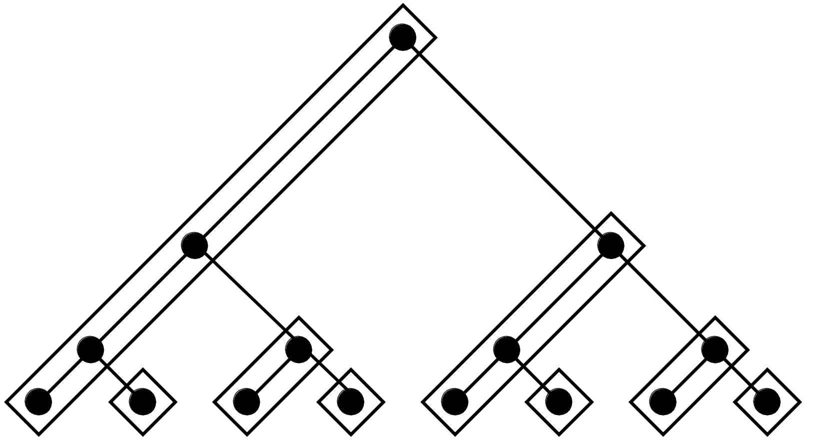
Explicit Placement via Ticket List



Divide-and-Conquer Skeletons

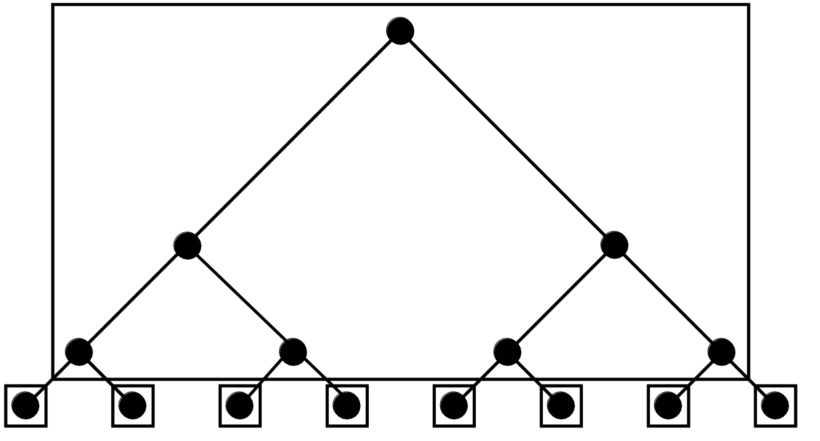
- **Distributed expansion**

```
disDC :: (Trans a, Trans b) =>
    Int         -- branch degree
    -> [Int] -- tickets
    -> ...      -- type of DC
```



- **Flat expansion**

```
flatDC :: (Trans a,Trans b) =>
   ((a->b) -> [a] -> [b])
      -- parallel map skeleton
   -> Int     -- depth
   -> ...     -- type of DC
```



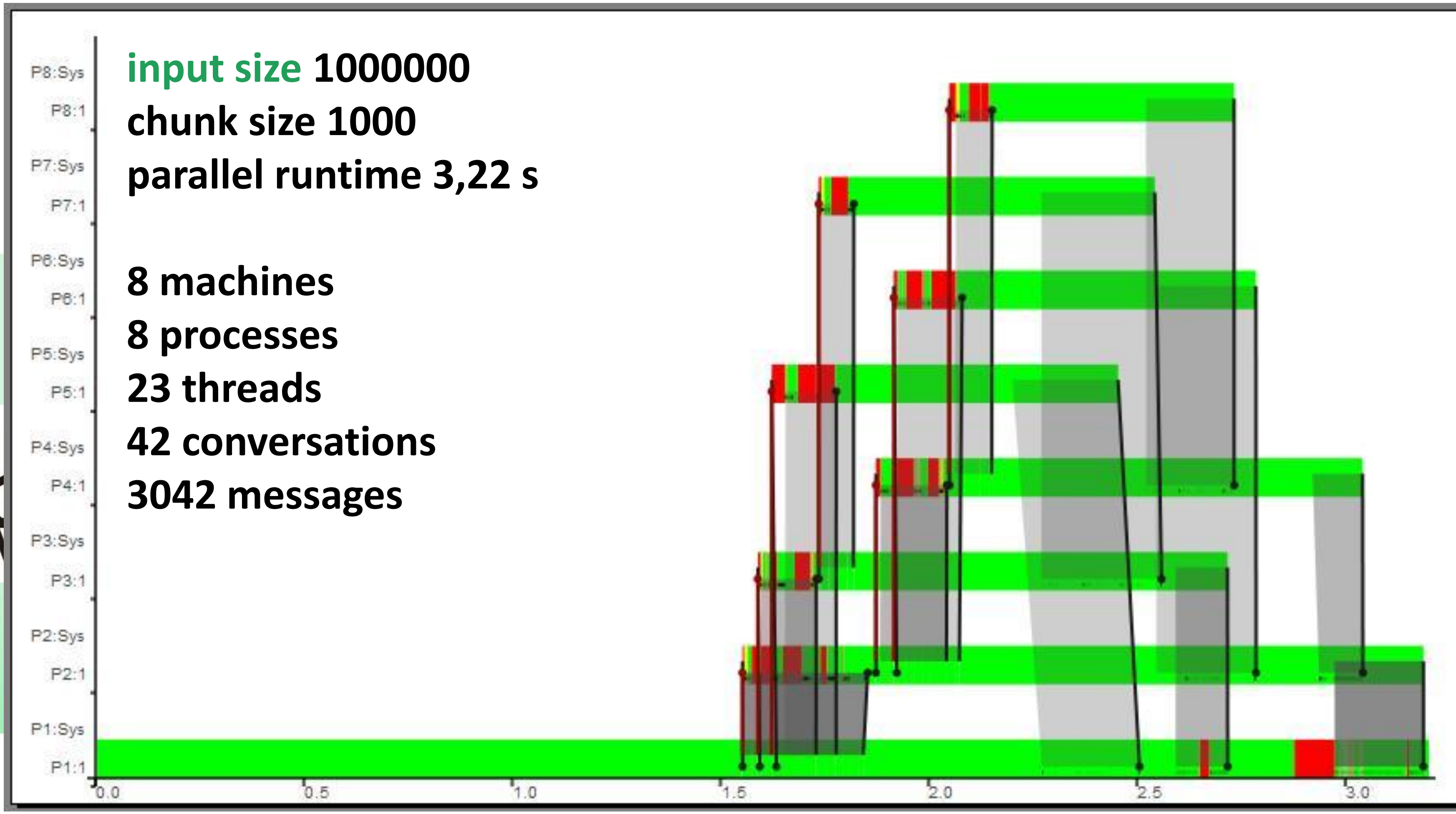
Parallelizing MergeSort Using disDC

```
-- divide and conquer: distributed expansion
ms "disDC" xs n d p
 = concat $ disDC 2 [2..p] triv solve split combine (chunk d xs)
-- disDC does not work with ghc-7.6.2, use dcNtickets_c instead
   where
    threshold   = n `div` p
    triv   xss  = length (concat xss) < threshold
    split       = unshuffle 2
    solve  xss  = (chunk d) . mergeSort .concat $ xss
    combine _  (b1:b2:_)
                = chunk d $ sortMerge (concat b1) (concat b2)

-- divide and conquer: flat expansion with parMap skeleton
ms "flatDC" xs n d p
 = concat $
       flatDC parMap depth triv solve split combine (chunk d xs)
   where
    depth     = floor ((log (fromIntegral p)) / log 2) :: Int
    threshold ...   -- as above
```

Chunking of input and output lists using chunk and concat to unchunk

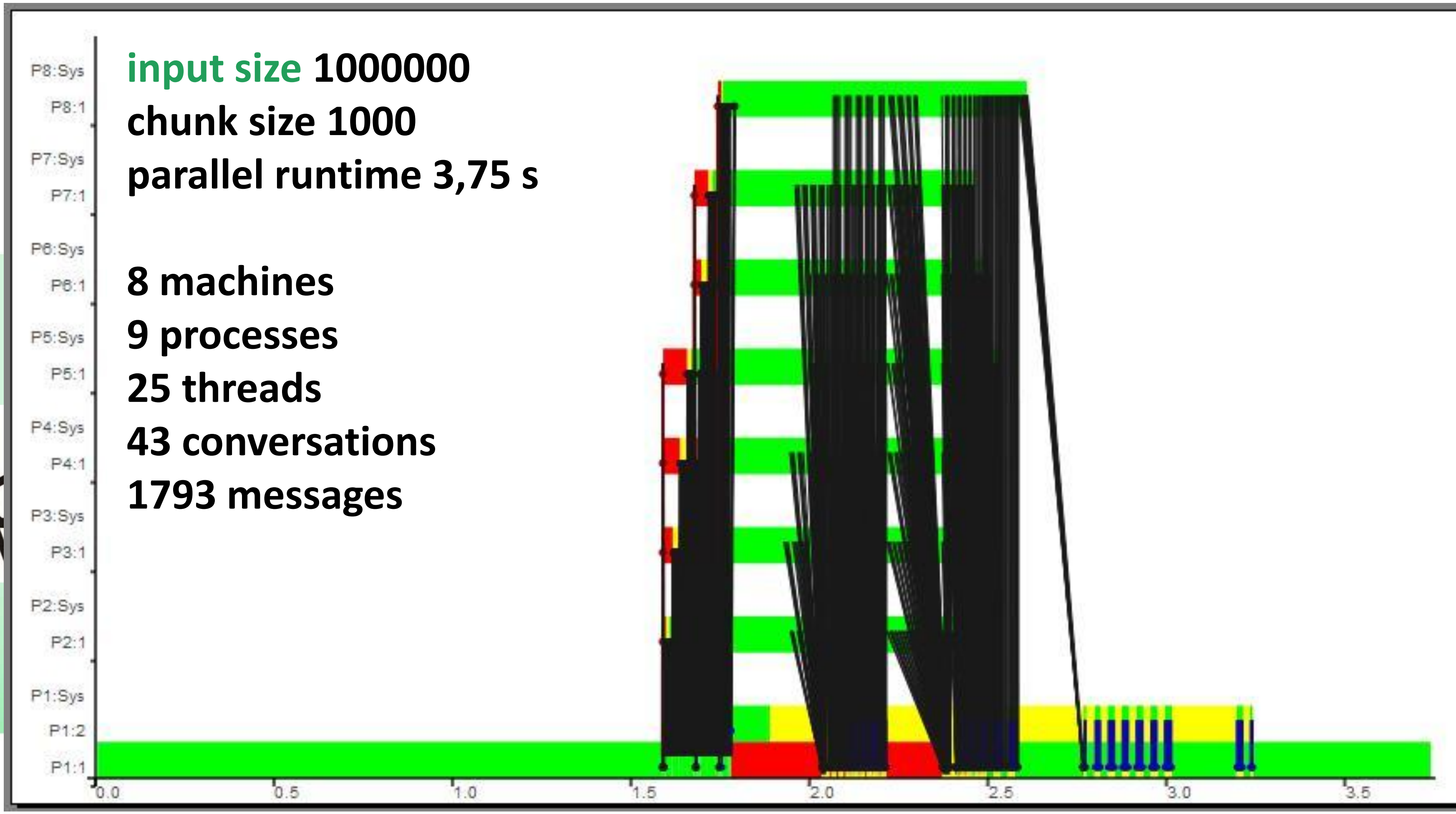
Runtime Behaviour – disDC Skeleton



Runtime Behaviour – flatDC Skeleton

input size **1000000**
chunk size 1000
parallel runtime 3,75 s

8 machines
9 processes
25 threads
43 conversations
1793 messages

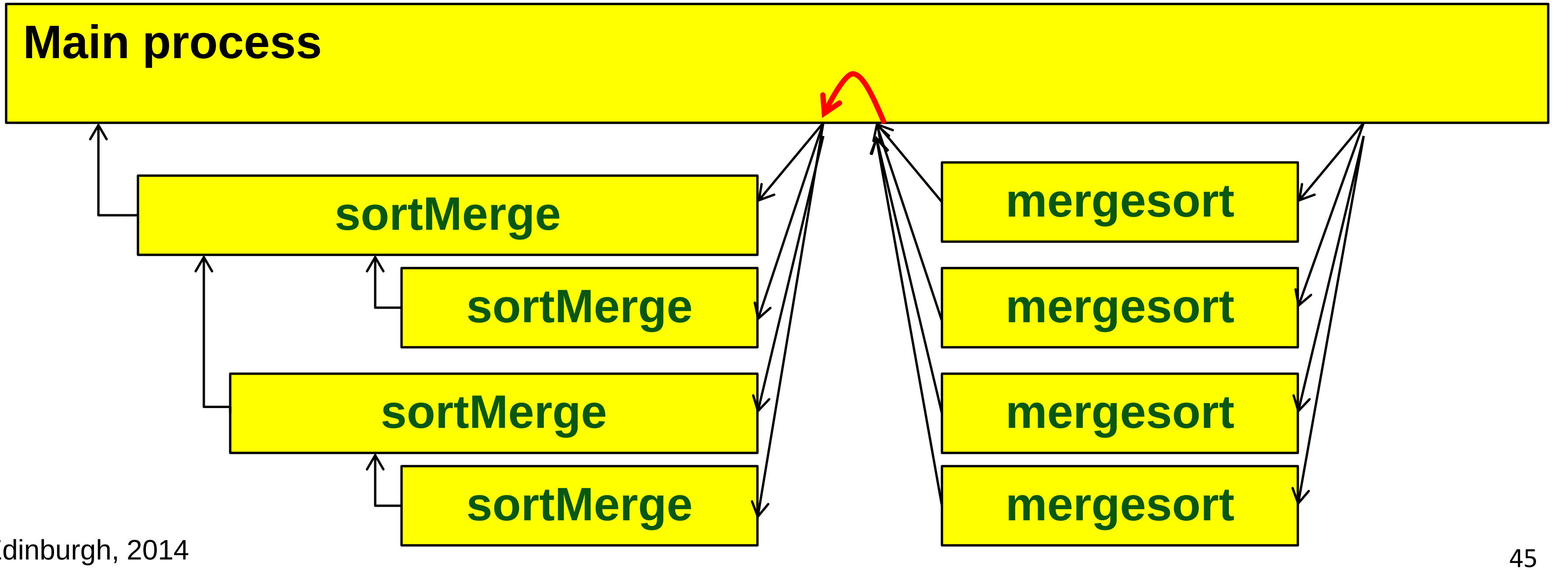


Skeleton Composition

Parallel MapReduce = ParMap → ParRed

- **Parallelisation of mergesort can be seen as a special map-reduce:**

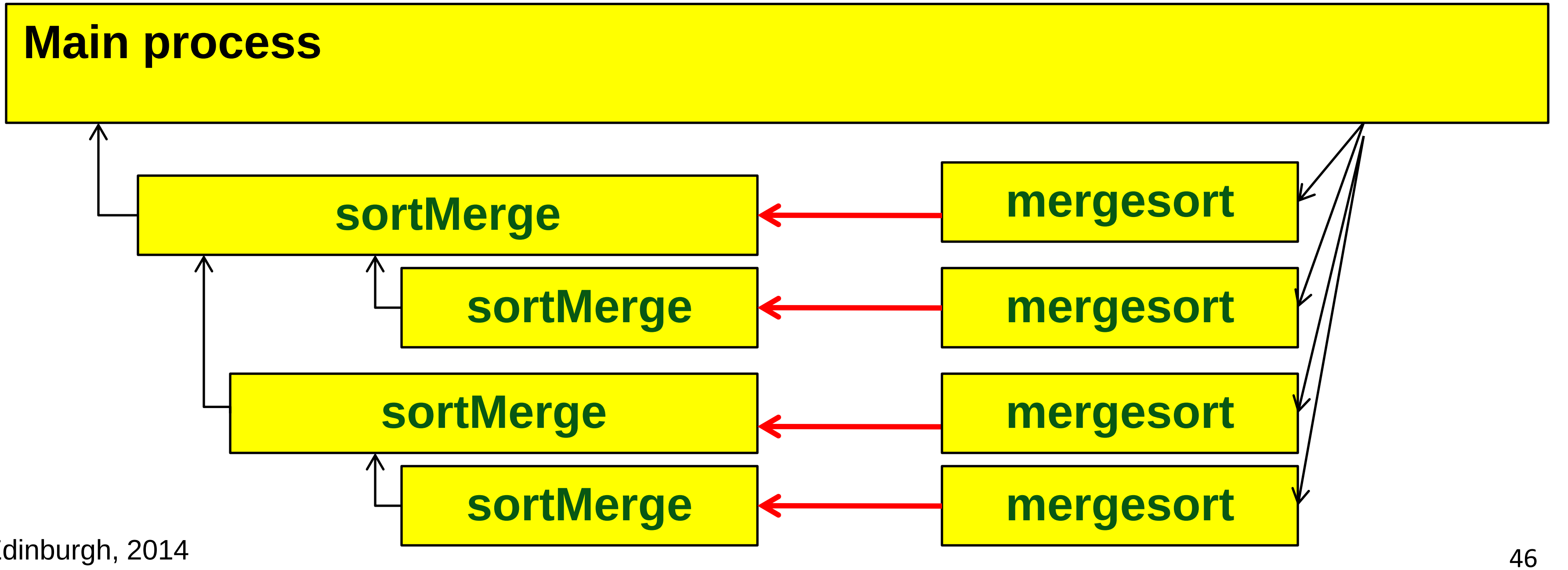
```
parms np xs = (parRed sortMerge) . (parMap mergesort) $
                                          (unshuffle np xs)
```



Parallel MapReduce = ParMap → ParRed

- **Parallelisation of mergesort can be seen as a special map-reduce:**

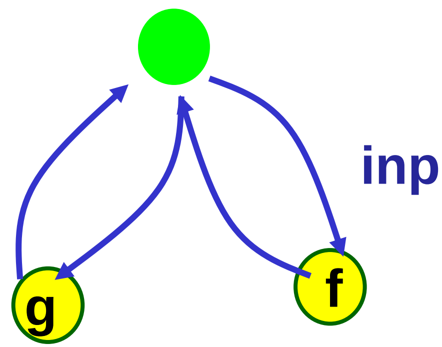
```
parms np xs = (parRed sortMerge) . (parMap mergesort) $
                                          (unshuffle np xs)
```



The „Remote Data"-Concept

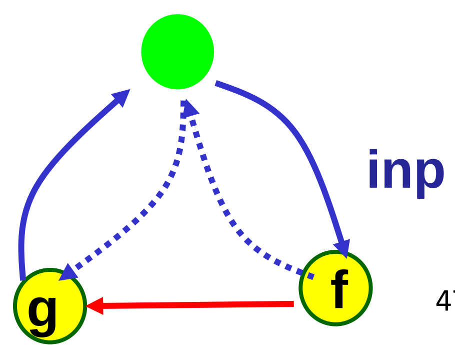
- Functions:
  - Release local data with release :: a -> RD a
  - Fetch released data with fetch :: RD a -> a

- Replace
  - spawn [process g] . spawn [process f] $ [inp]



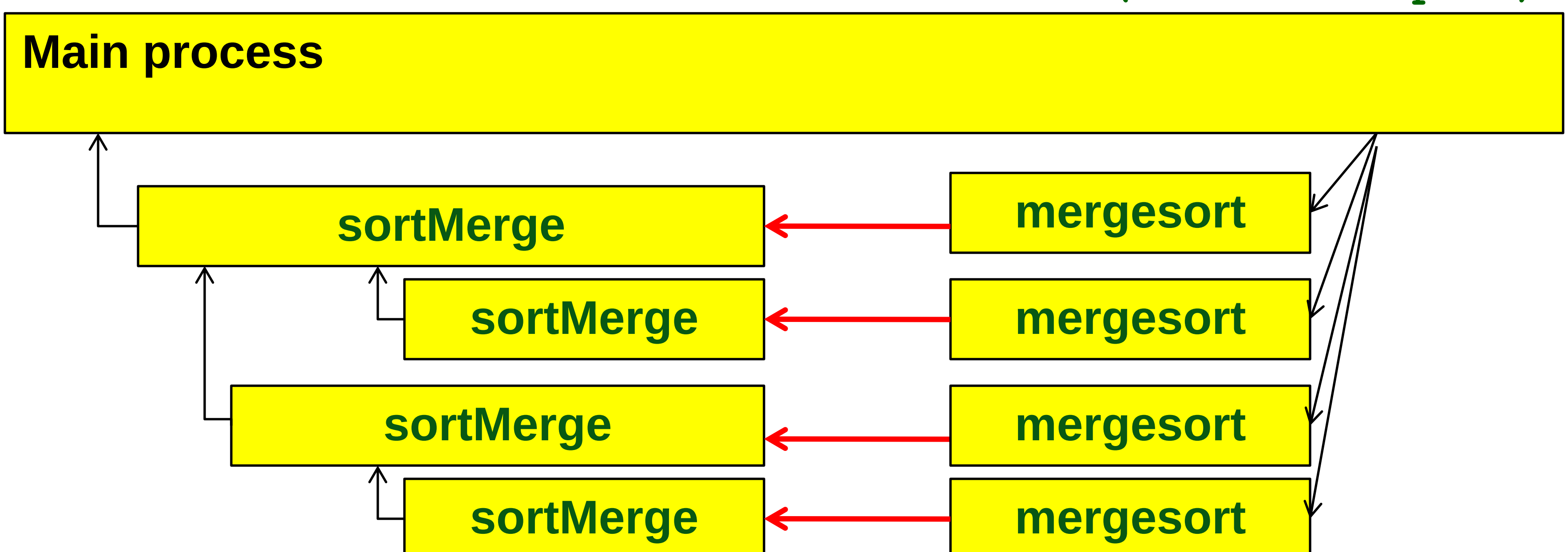
with

- spawn [process (g o fetch)] . spawn [process (release o f)] $ [inp]

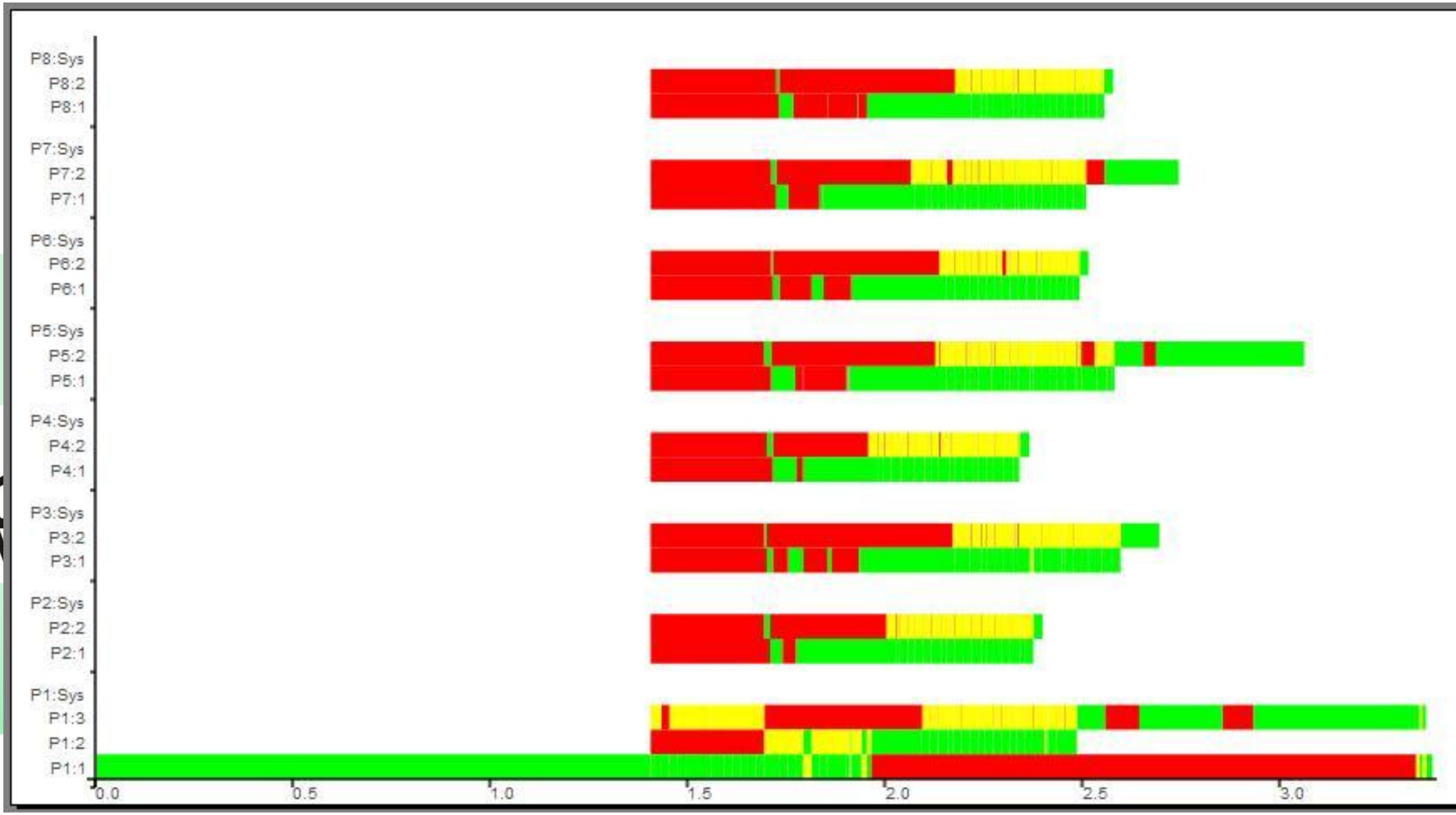


Parallel MapReduce = ParMap → ParRed

```
parRed :: (Trans a) =>
           (a -> a -> a)          -- Reduction function
            -> a                  -- neutral element
            -> [RD a]  -> RD a    -- Input → Output
parms np xs = fetch . (parRed sortMerge) .
                      (parMap (release.mergesort) $
                                          (unshuffle np xs)
```

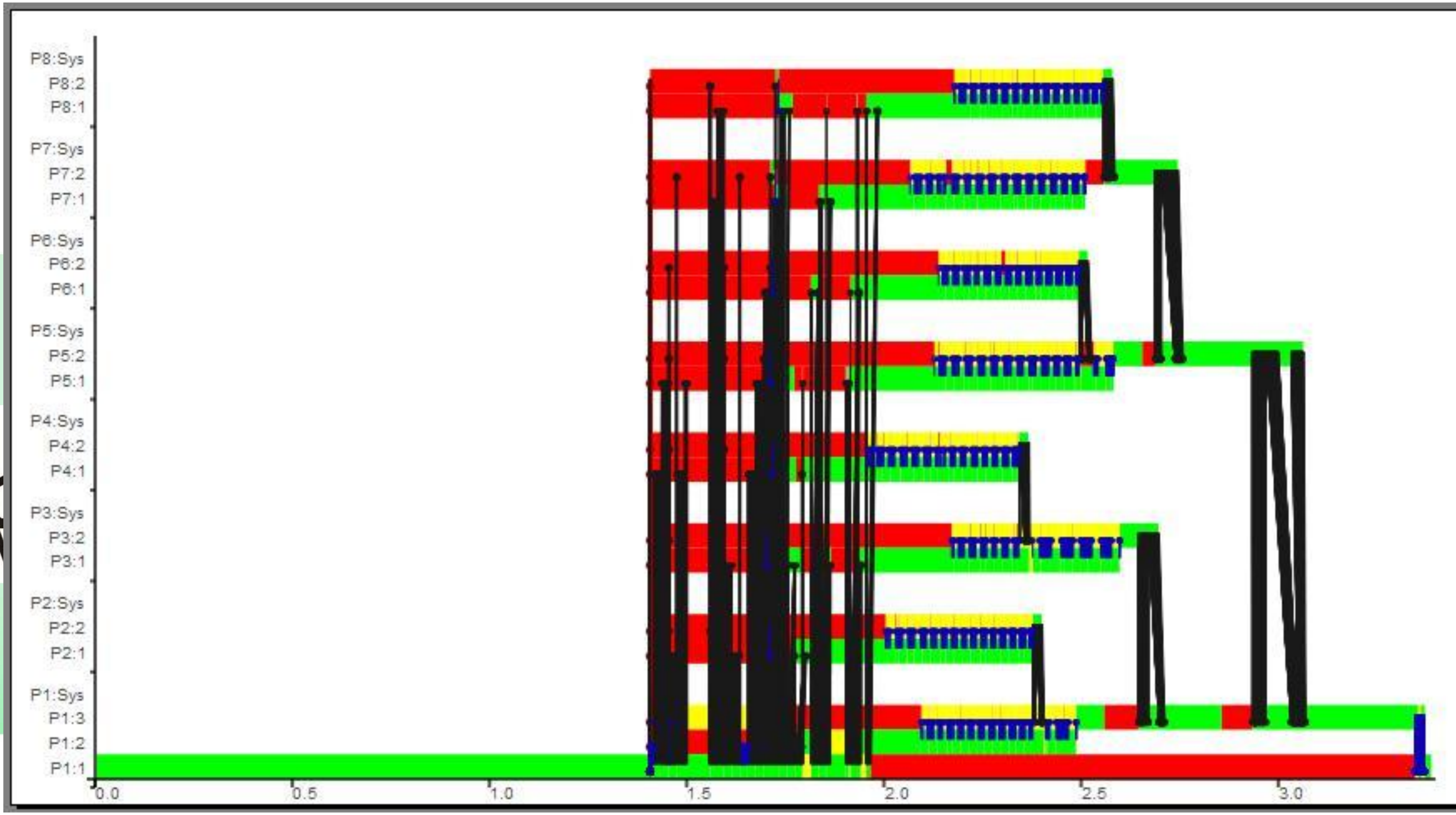


Runtime Behaviour



3,399s, 8 Machines, 17 Processes, 81 Threads, 96 Conversations, 2475 Messages

Runtime Behaviour



3,399s, 8 Machines, 17 Processes, 81 Threads, 96 Conversations, 2475 Messages

PSRS – Parallel Sorting by Regular Sampling

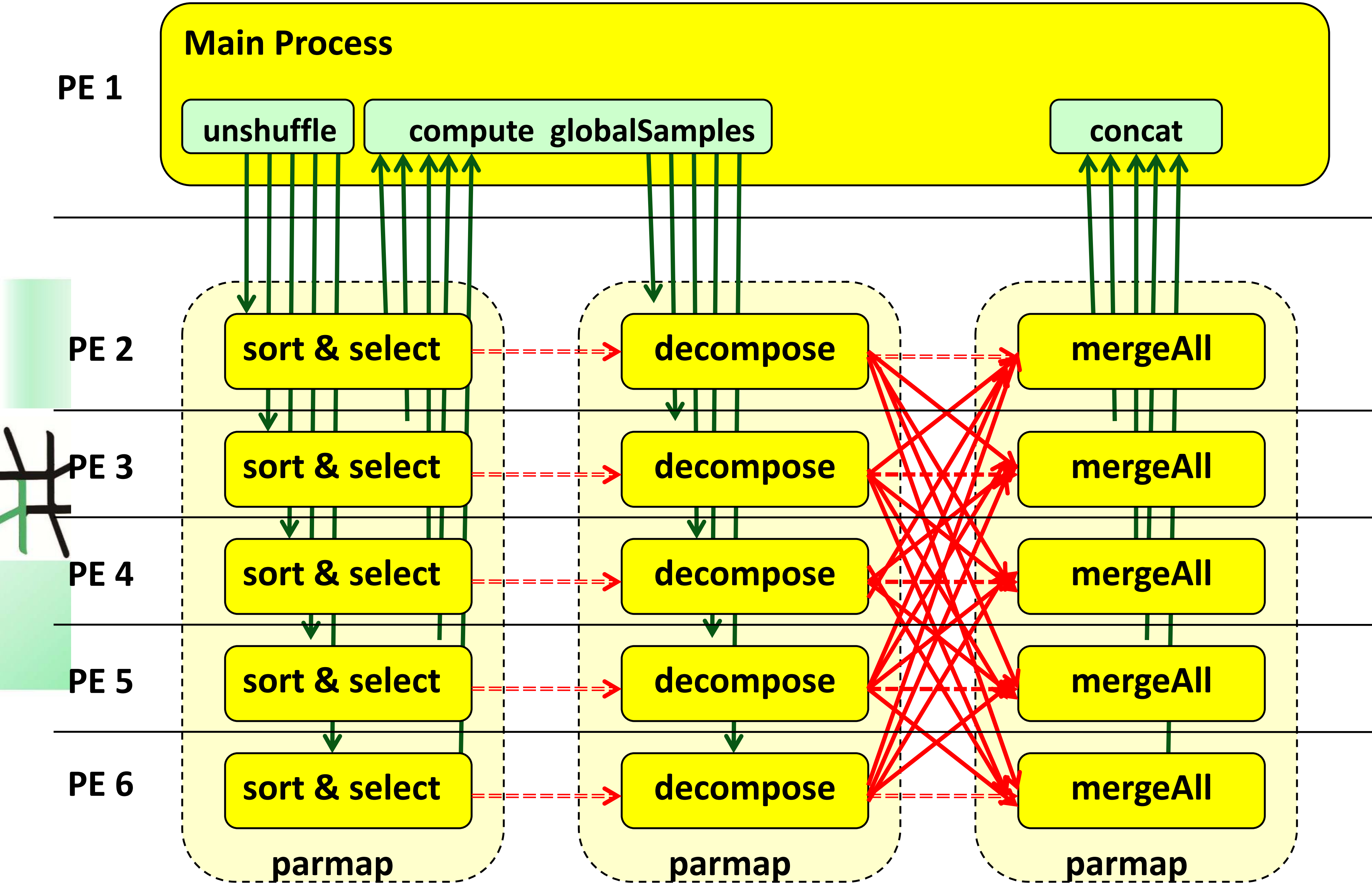
- **4 Phases:**
  1. **split** input list into p equal-sized segments,
     in parallel: **sort segments and select p sample elements** of each segment
  2. **collect and sort all p^2 samples** (p samples from each process), **select (p-1) pivot elements** and broadcast them to all processes
  3. Each process decomposes its segments into p partitions according to the pivot elements and sends the jth partition to process j (1<= j <= p)
  4. Each process merges the p partitions it received

- Complexity: O(n/p log(n)) if $n > p^3$

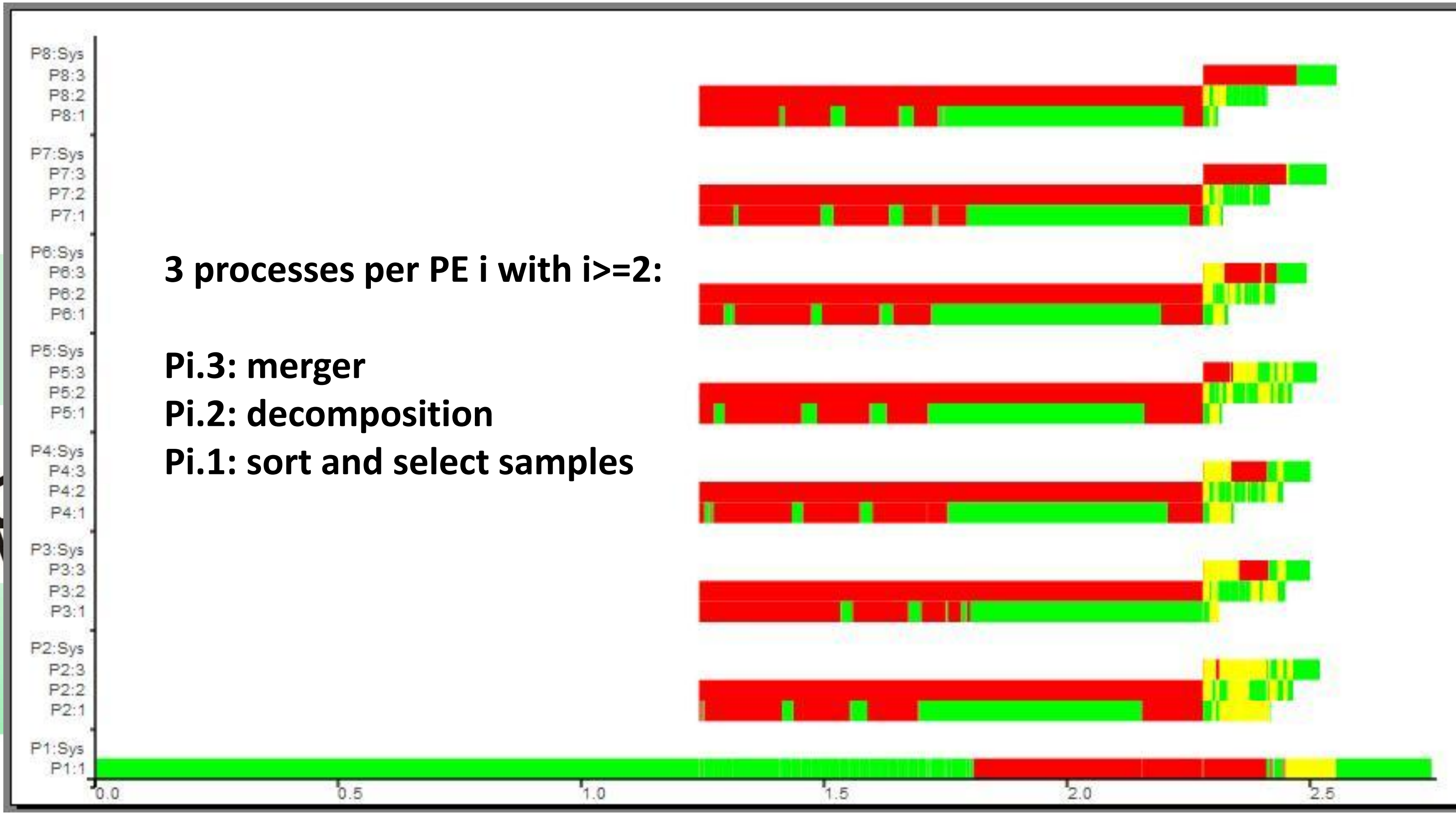
PSRS in Eden

```
psrs :: (Trans a, Ord a) => Int -> [a] -> [a]
psrs p xs = concat results
 where
 -- rdys :: [Rd [a]]                                                    1
 (samples, rdys)
   = unzip $ parMap (\ xs-> let ys = sort xs
                                in (getSamples p ys, release  ys))
                      (unshuffle p xs)
 globalSamples  = getGlobalSamples p . mergeAll $ samples               2
 -- partitions :: [[Rd [a]]]                                            3
 partitions    = parMap (\ (handle, pivots)
                   -> ((map release).(decompose pivots).fetch $
                                                    handle)))
                         (zip rdys (replicate p globalSamples)))
 parts          = transpose partitions
 results        = parMap (mergeAll . (map fetch)) parts                 4
```

PSRS Process Network

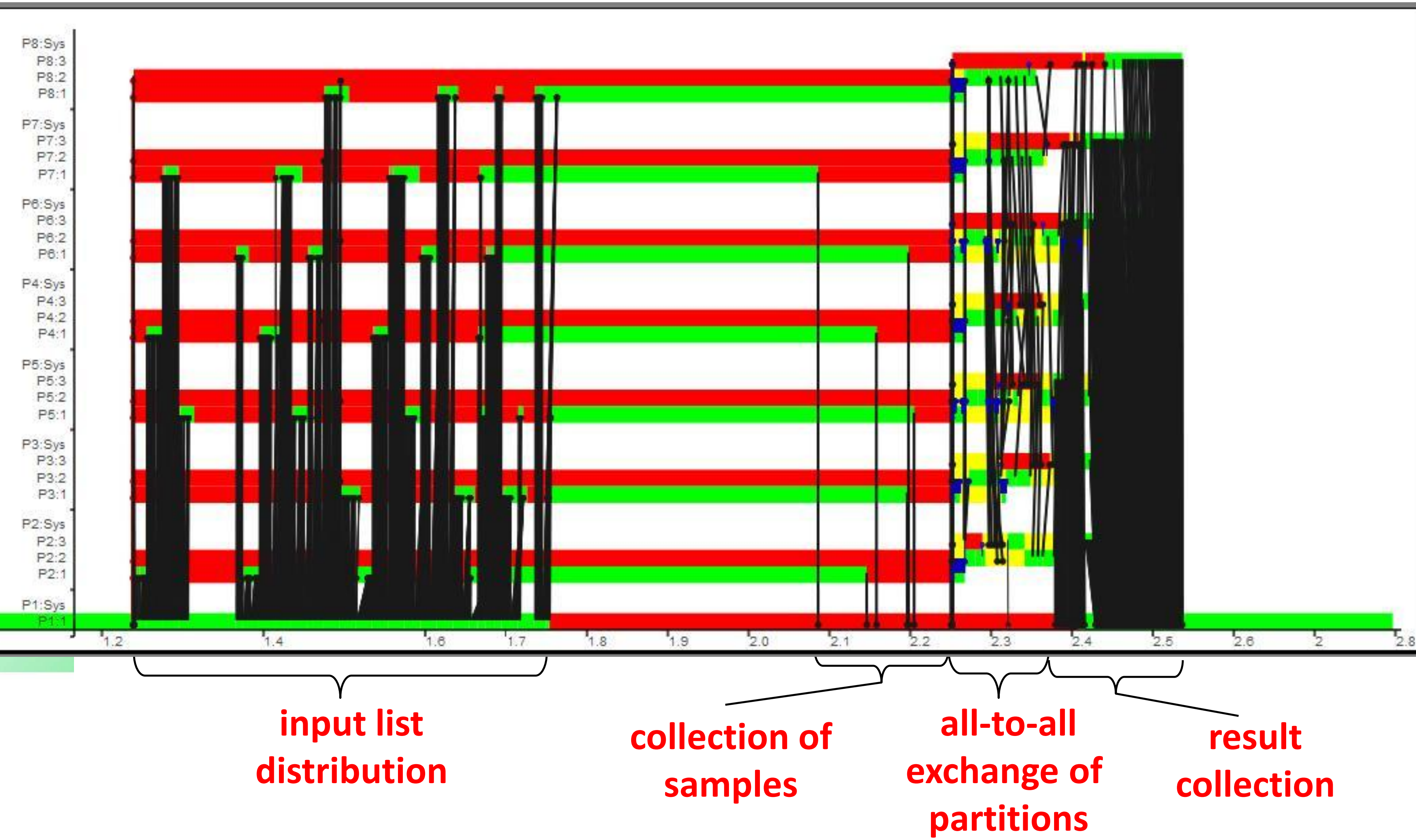


PSRS Runtime Behaviour



2,760s, 8 Machines, 22 Processes, 177 Threads, 210 Conversations, 2311 Messages

PSRS Runtime Behaviour: Communication



2,76s, 8 Machines, 22 Processes, 177 Threads, 210 Conversations, 2311 Messages

Conclusions

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

- Eden = Haskell + Coordination
  - Explicit process definitions
  - Implicit communication (data transfer) defined via type class Trans
  - Remote Data
    -> pass data directly from producer to consumer processes
- Programming Methodology:
  Use or adapt algorithmic skeletons from the skeleton library:
  - parallel maps: parMap, farm, offlineFarm …
  - master-worker: flat, hierarchical, distributed …
  - divide-and-conquer: distributed expansion, flat expansion …
  - topology skeletons: ring, torus, all-to-all, …
  - skeleton iteration

  or design your own skeletons
- Compose skeletons using remote data to implement arbitrary parallel algorithms

Conclusions

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

- **Eden compiler** extends GHC with parallel runtime system
- on **distributed systems**, middleware like MPI and PVM is used for communication
  (→ compile options –parmpi and –parpvm)
- on **multicores**, a special implementation using copying instead of message passing is available
  (→ compile option –parcp)

- **EdenTV** is a powerful tool to analyse the runtime behaviour of Eden programs

Lab Notes

- **Look at exercises.pdf for instructions on how to set up the environment for experiments**
  - **on the lab machines and**
  - **on the beowulf cluster**

- **There are four exercises marked as easy, medium or advanced. Try to do one or two of them.**