Parallele funktionale Programmierung in Eden

Parallele Programmierung auf hoher Abstraktionsebene

Parallelitätskontrolle

- **explizite Prozesse**
- **implizite Kommunikation** (kein send/receive)
  - Laufzeitsystemkontrolle
  - strombasierte typisierte Kommunikationskanäle
- disjunkte Adressräume, **verteilter Speicher**
- **Nichtdeterminismus**, reaktive Systeme

funktionale Sprache

- polymorphes Typsystem
- Pattern Matching
- Funktionen höherer Ordnung
- lazy evaluation
- ...

Eden (1)

parallele funktionale Sprache

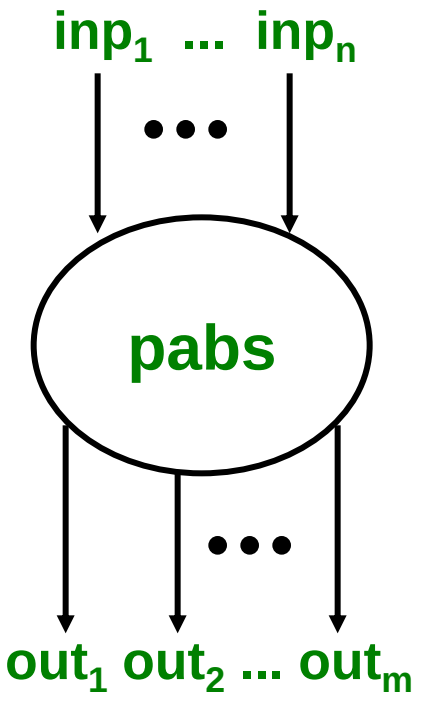
- **Berechnungssprache:** **Haskell**
- **Koordinierungssprache:**
  - + **Prozessabstraktion**
  - + **Prozessinstanzierung**

Typklasse Transmissible
(Trans a, Trans b) =>

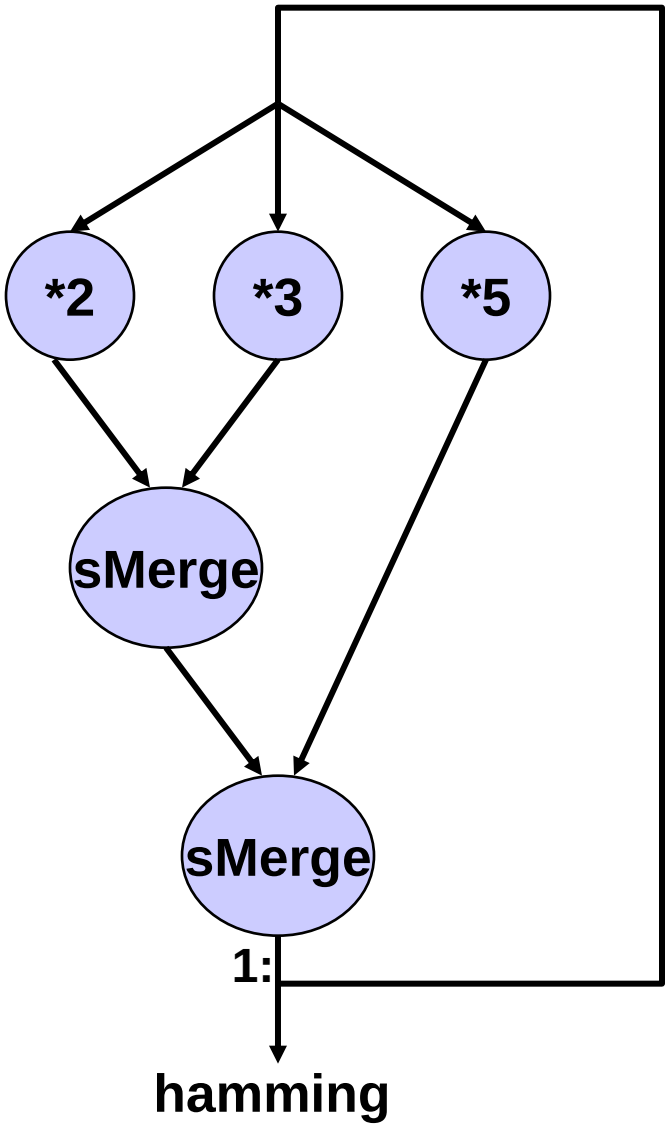
```
process :: (a -> b) -> Process a b
( # )   :: Process a b -> a -> b
```

Häufige Form:

```
pabs :: Process (τ1,...,τn)  (σ1,...,σm)
pabs =  process \ (i1,...,in) -> (o1,...,om)
          where  eqn1 ...  eqnk

pabs # (inp1,...,inpn)   :: (σ1,...,σm)
```



Beispiel: Hamming-Prozessnetz




```
hamming   :: [Int]
hamming   = 1: sMerge #
                 (sMerge #   ((multp 2) # hamming,
                              (multp 3) # hamming)
                  (multp 5) # hamming )

multp     ::  Int -> Process [Int] [Int]
multp n   =   process (map (*n))

sMerge    ::  Process ([Int],[Int]) [Int]
sMerge    =   process \ (xs,ys) -> sm xs ys
              where
                 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
```

Fragen zur Semantik

- denotationell
  - Prozessabstraktion ~> lambda-Abstraktion
  - Prozessinstanzierung ~> Applikation

  ➔ Wert/Ausgabe eines Programms,
  aber keine Information über Ausführung, Parallelitätsgrad, Laufzeiteinsparungen

- operationell

  1. Wann wird ein Prozess erzeugt?
     Wann wird eine Prozessinstanzierung ausgewertet?

  2. Zu welchem Grad werden Prozessausgaben ausgewertet?
     Kopfnormalform oder Normalform oder ...?

  3. Wann werden Prozessausgaben kommuniziert?

Antworten

Bedarfssteuerung	Eden

1. Wann wird ein Prozess erzeugt?
   Wann wird eine Prozessinstanzierung ausgewertet?

nur bei Bedarf für seine Ausgaben	nur bei Bedarf für seine Ausgaben

2. Zu welchem Grad werden Prozessausgaben ausgewertet?
   Kopfnormalform oder Normalform oder ...?

WHNF (Kopfnormalform)	Normalform

3. Wann werden Prozessausgaben kommuniziert?

nur bei Bedarf: Anfrage- und Antwortnachrichten	eager communication: Werte werden ohne Anforderung an Empfänger geschickt

Bedarfssteuerung (lazy evaluation) vs. Parallelität

- **Problem:** **Bedarfssteuerung ==> verteilte Sequentialität**
- **Abhilfe in Eden:**
  - **Bedarf bei Kommunikation („eager communication“):**
    - **Normalformauswertung aller Prozessausgaben (durch unabhängige Threads)**
    - **Kommunikation von Werten erfolgt, sobald diese verfügbar sind**
  - **explizite Bedarfssteuerung über sequentielle Strategien (Modul Seq):**
    - **rnf, rwhnf, spine ... :: Strategy a**
    - **using :: a -> Strategy a -> a**

Parallelles Map

Haskell Definition:

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

Eden-Version (1. Ansatz):

```
parMap          ::  (Trans a, Trans b) =>
                    Process a b -> [a] -> [b]
parMap p [ ]    =   [ ]
parMap p (x:xs) =   (p # x) : parMap p xs
```

parMap p [i_1, i_2, ..., i_n] => (p # i_1) : parMap p [i_2, ... , i_n]

$\mathcal{E}$1 (Kopfnormalform), zunächst keine Prozesserzeugung

Parallelles Map

Haskell Definition:

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

Eden-Version (2. Ansatz):

```
parMap          ::  (Trans a, Trans b) =>
                    Process a b -> [a] -> [b]
parMap p [ ]    =   [ ]
parMap p (x:xs) =   (p # x) : parMap p xs
                          `using`  seqlist r0
```

parMap p [i_1, i_2, ..., i_n] => (p # i_1) : (p # i_2) …: (p # i_n):[]

$\mathcal{E}$2, Auswertung aller Konstruktorknoten ohne direkte Prozess-erzeugung

Parallelles Map

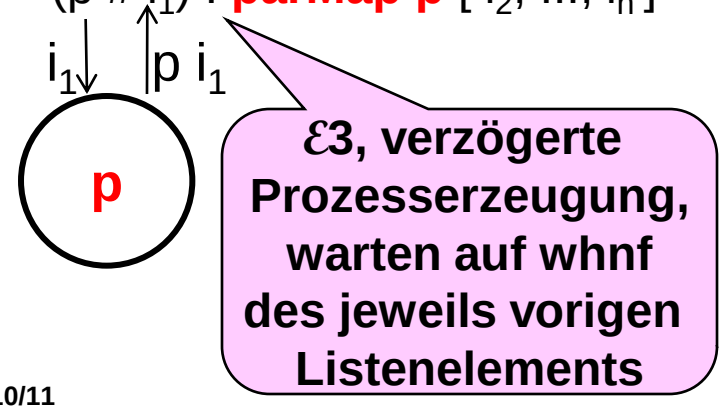
Haskell Definition:

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

Eden-Version (3. Ansatz):

```
parMap          ::  (Trans a, Trans b) =>
                    Process a b -> [a] -> [b]
parMap p [ ]    =   [ ]
parMap p (x:xs) =   (p # x) : parMap p xs
                          `using`  seqlist rwhnf
```

parMap p [i_1, i_2, ..., i_n] => (p # i_1) : **parMap p** [i_2, ..., i_n]



Eden(2)

+ Eager Prozess Instanziierung

```
spawn  ::   (Trans a, Trans b) => [Process a b]  ->  [a]  ->  [b]
-- spawn   = zipWith ( # )  mit Bedarfskontrolle, die alle Prozesse direkt erzeugt
```

Paralleles Map

Haskell Definition:

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

Eden-Version:

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

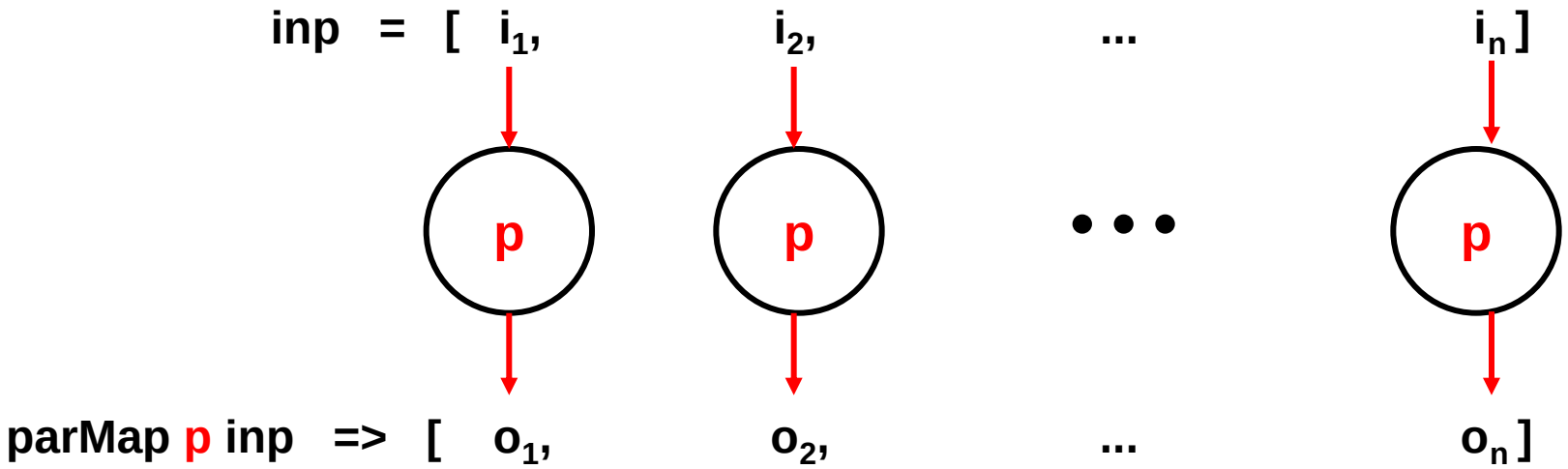
Parallelles Map

Haskell Definition:

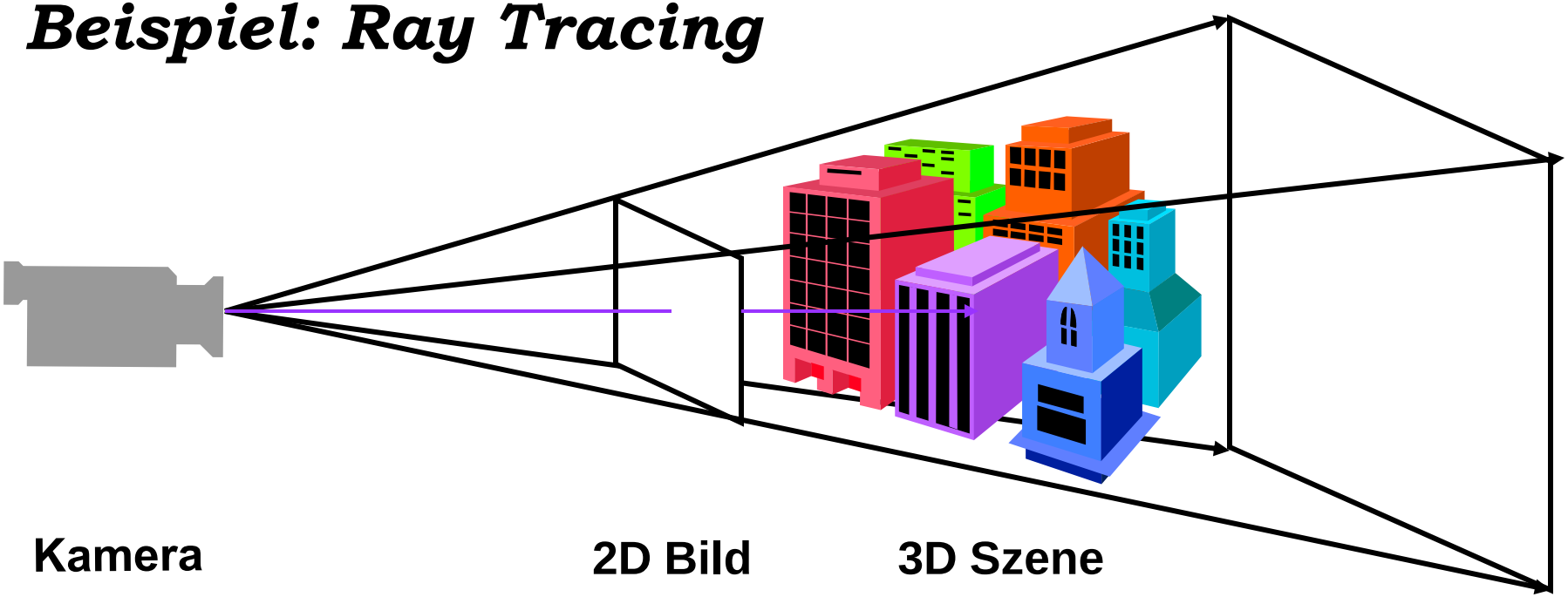
```
map          ::    (a -> b)   -> [a] -> [b]
map  f  [ ]  =     [ ]
map  f  (x:xs)=    (f x) : map  f  xs
```

Parallele Eden-Version (mit map Interface):

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

map_par      ::  (Trans a, Trans b) => (a -> b) -> [a] -> [b]
map_par      =   parMap . process
```



Beispiel: 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
```

Parallelisierung des Ray Tracers mit parMap

- **Erzeuge für jeden Strahl einen Prozess, der den Schnitt des Strahls mit der 3D Szene berechnet:**

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

- **Ersetze**

```
findImpacts rays objs = map (firstImpact objs) rays
```

durch:

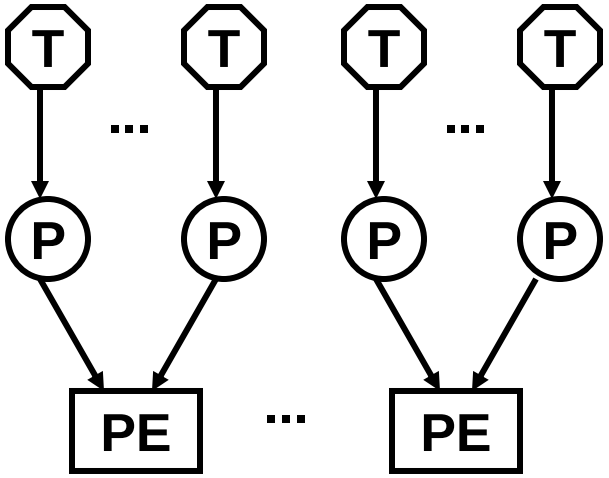
```
findImpacts rays objs = map_par (firstImpact objs) rays
```

⇒ **feine Granularität**

⇒ **besser:**
Erzeuge nur so viele Prozesse wie Prozessoren verfügbar und lasse jeden Prozess mehrere Strahlen bearbeiten

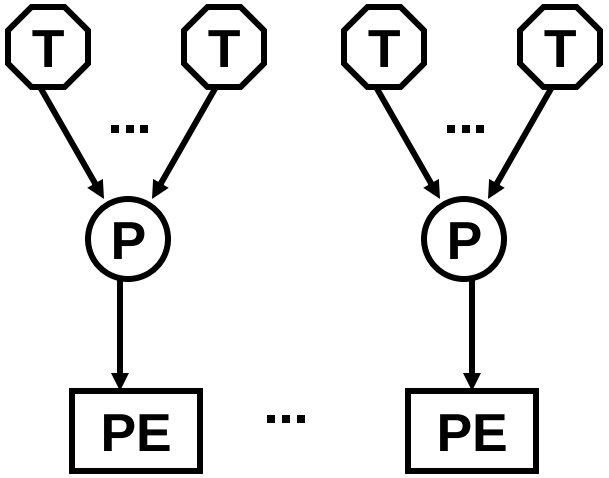
Weitere Prozessschemata: Farm

Paralleles Map




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

parMap p  xs   =   spawn (repeat p) xs
```

Farm




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

Parallelisierung des Ray Tracers mit farm

Ersetze findImpacts rays objs = map_par (firstImpact objs) rays

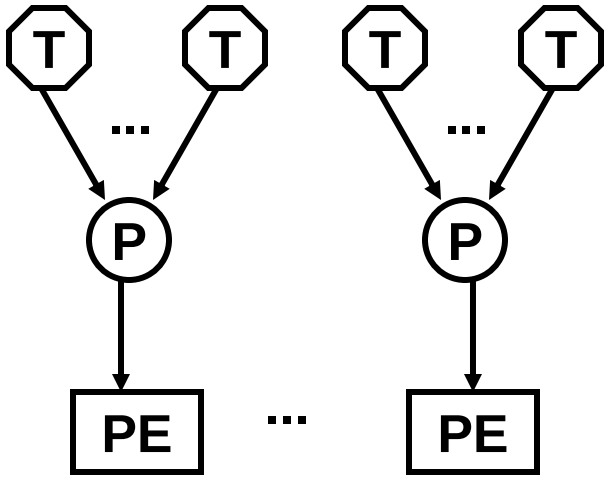
durch: findImpacts rays objs = map_farm (firstImpact objs) rays

```
map_farm      ::  (Trans a, Trans b) => (a -> b) -> [a] -> [b]
map_farm  f   =   farm (unshuffle noPe) shuffle (process (map f))

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

takeEach      ::  Int -> [a] -> [a]
takeEach n []    = []
takeEach n (x:xs) = x : takeEach n (drop (n-1) xs)

shuffle       ::  [[b]] -> [b]
shuffle       =   concat . transpose
```

EdenSkel.lhs

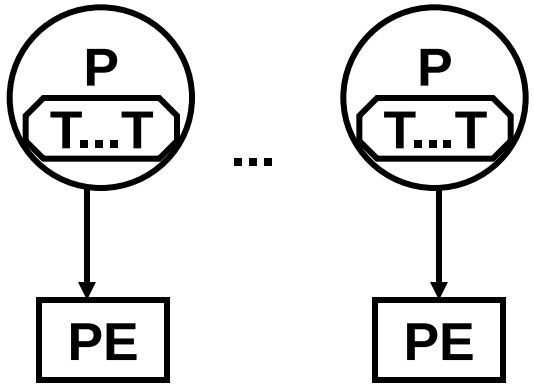
Weitere Prozessschemata: Direct Mapping

Farm




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

Offline Farm




```
offlineFarm      ::    (Trans a, Trans b) =>
     ([a] -> [[a]]) -> ([[b]] -> [b])
     -> ([a] -> Process () [b]) -> [a] -> [b]
offlineFarm  distribute combine  p xs
  =  combine (
        spawn [ p (taskss !! i) | i <- [0..(np-1)]]
                  (replicate np ()))
     where  taskss  = distribute xs
              np = length taskss
```

Parallelisierung des Ray Tracers mit Direct Mapping

Ersetze findImpacts rays objs = map_farm (firstImpact objs) rays

durch findImpacts rays objs = map_offlineFarm (firstImpact objs) rays

```
map_offlineFarm     ::   (Trans a,Trans b) => (a -> b)  -> [a]  -> [b]
map_offlineFarm f xs    =    offlineFarm (unshuffle noPe) shuffle procf xs
   where procf xs = process (\() -> map f xs)
```

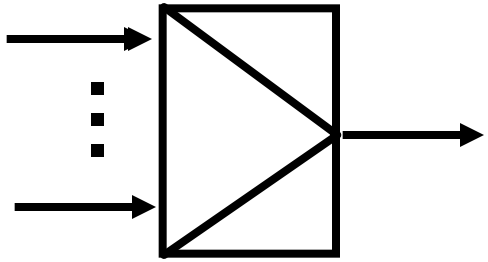
EdenSkel.lhs

Eden (3)

...

+ **m:1 Kommunikation (Nichtdeterminismus)**

```
merge  ::  Trans a => [[a]]  ->  [a]
```



+ **dynamische Antwortkanäle**

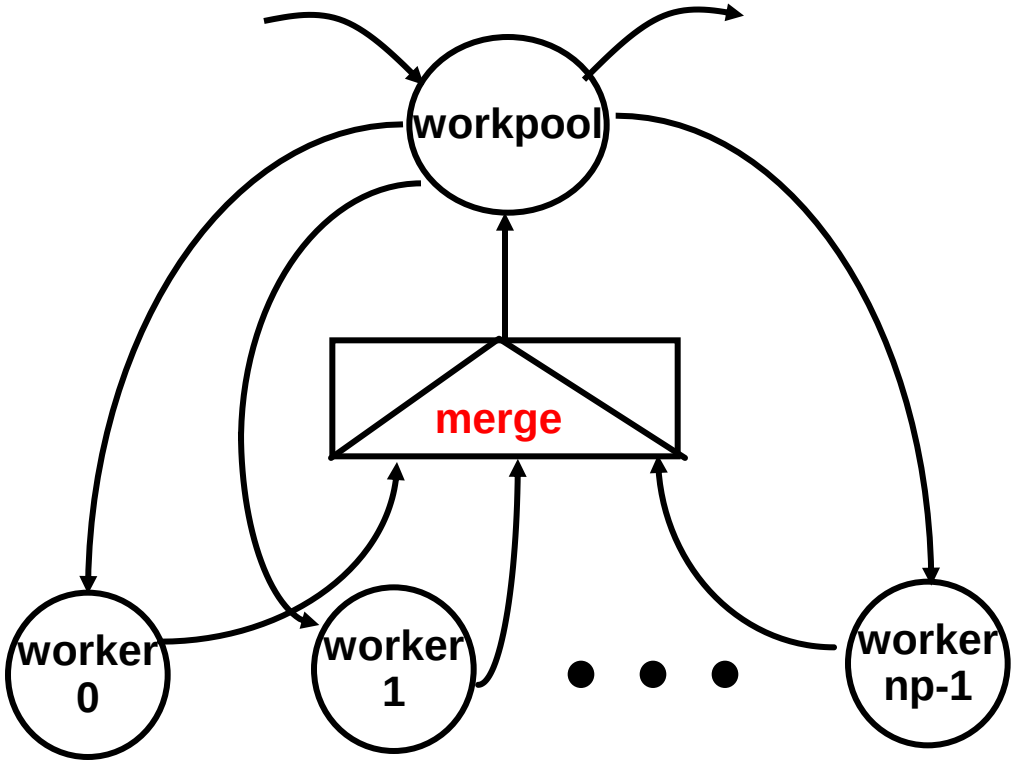
- Kanalerzeugung

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

- Kanalbenutzung

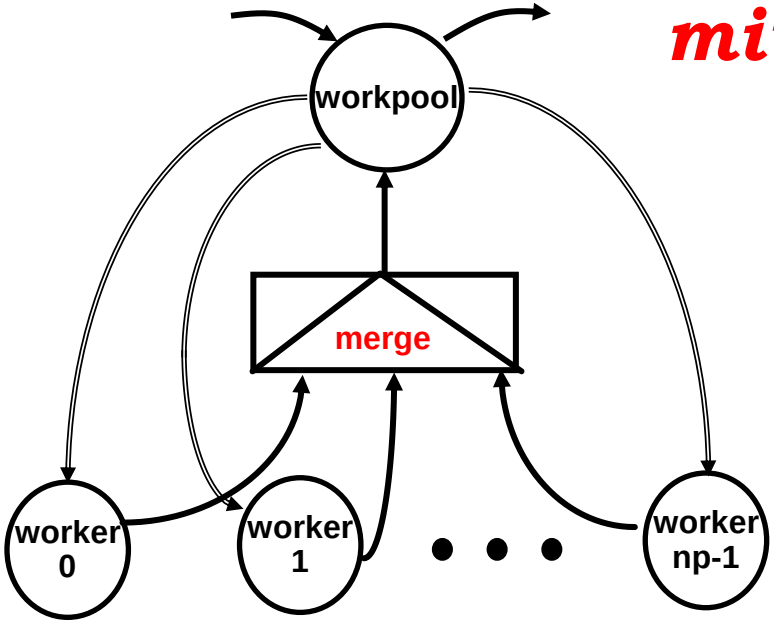
```
parfill :: Trans a =>    ChanName a -> a -> b  -> b
```

Workpool Schema




```
workpool    ::    Int -> Int -> Process [t] [r] -> [t] -> [r]
workpool    np prefetch worker tasks
  = map (\ (id,res) -> res) fromWorkers
    where   fromWorkers     = merge   (tagWithPids [worker # ts | ts <- taskss])
            taskss          = distribute np (initialReqs ++ newReqs) tasks
            initialReqs     = concat (replicate prefetch [0..np-1])
            newReqs         = map (\ (id,res) -> id) fromWorkers
```

Parallelisierung des Ray Tracers mit workpool

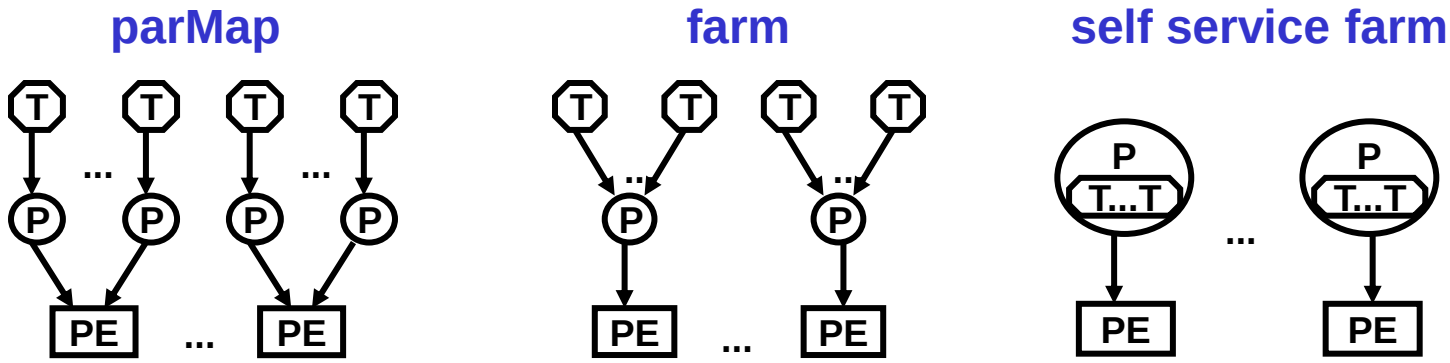



```
workpool :: Int -> Int ->
            Process [t] [r]  -> [t] -> [r]
```

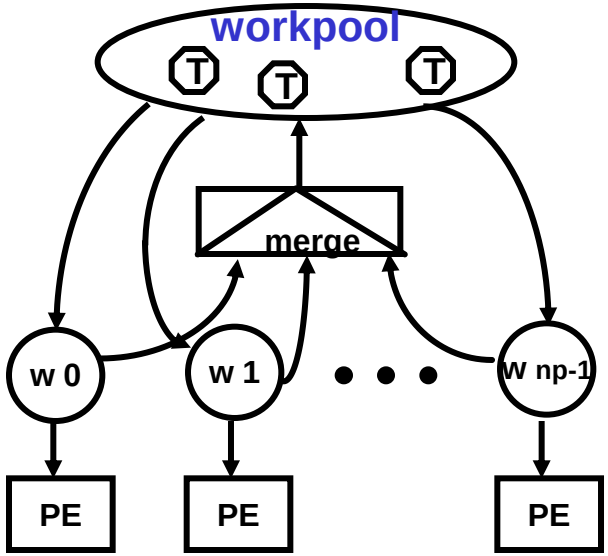
```
findImpacts                 ::  [Ray] -> [Object] -> [Impact]
findImpacts rays objs
  =  mergeByNumber $
       workpool noPe prefetch (firstImpactWorker2 objs)   (zip  [0..]  rays)

firstImpactWorker2          ::  [Object] ->  Process [(Int,Ray)] [(Int, Impact)]
firstImpactWorker2              objs
  = process (map (\ (nr,ray) -> (nr, firstImpact objs ray)))

prefetch = 2 :: Int
```

Prozessschemata-Übersicht

- **statische Aufgabenverteilung:**



- **dynamische Aufgaben-verteilung:**



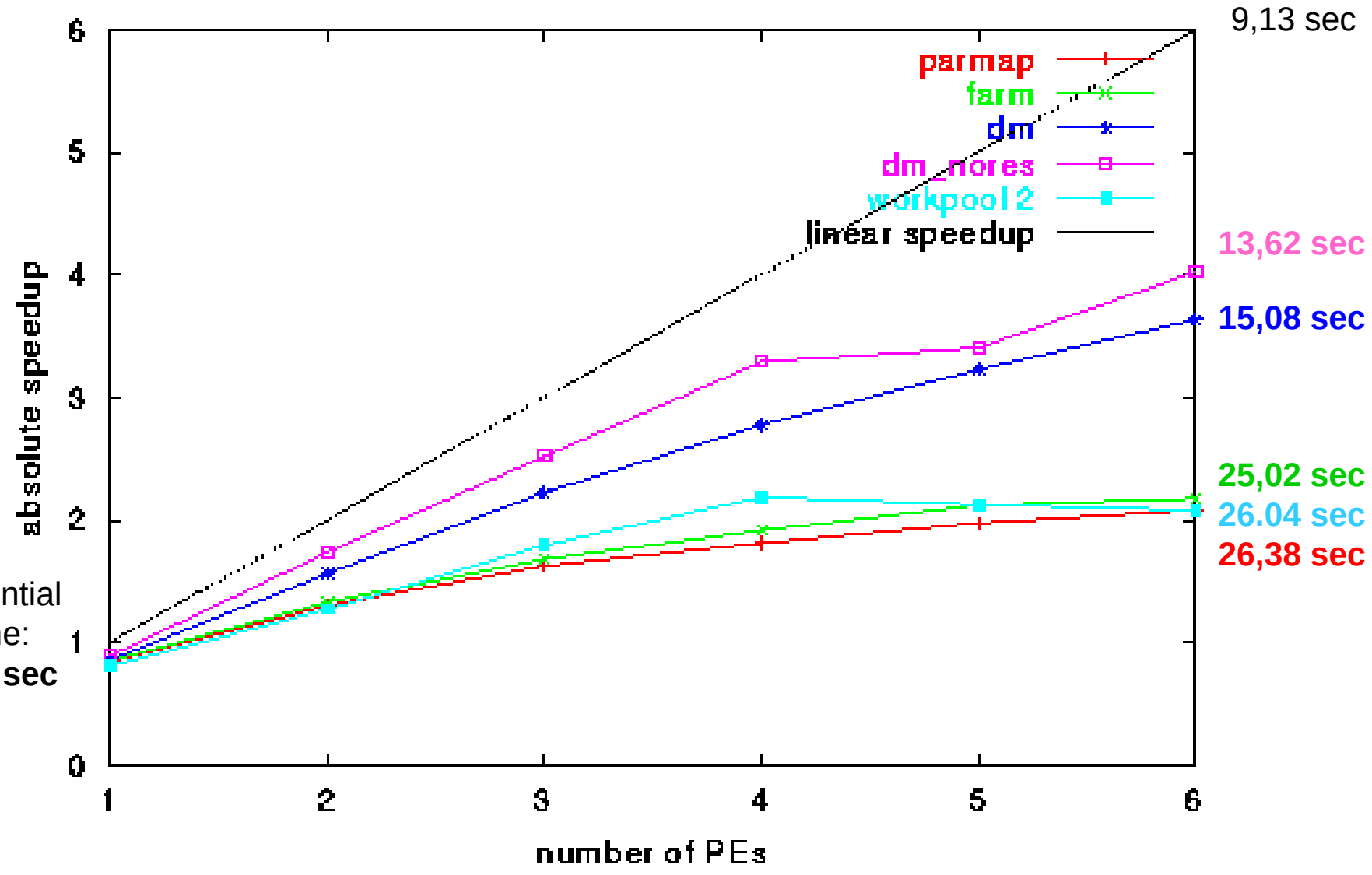
Fazit

- **parallele funktionale Sprache Eden**
  - **explizite Prozeßdefinitionen und implizite Kommunikation**
  - **→ explizite Kontrolle der Prozeßgranularität und Kommunikationstopologie**
  - **implementiert durch Erweiterung des Glasgow Haskell Compilers**

- **Parallelisierungen mit Prozeßschemata**

Schemata	Taskzerlegung	Aufgabenverteilung
parMap	regulär	statisch: Prozess pro Task
farm	regulär	statisch: Prozess pro Prozessor
ssf	regulär	statisch: Taskselektion in Prozessen
workpool	irregulär	dynamisch

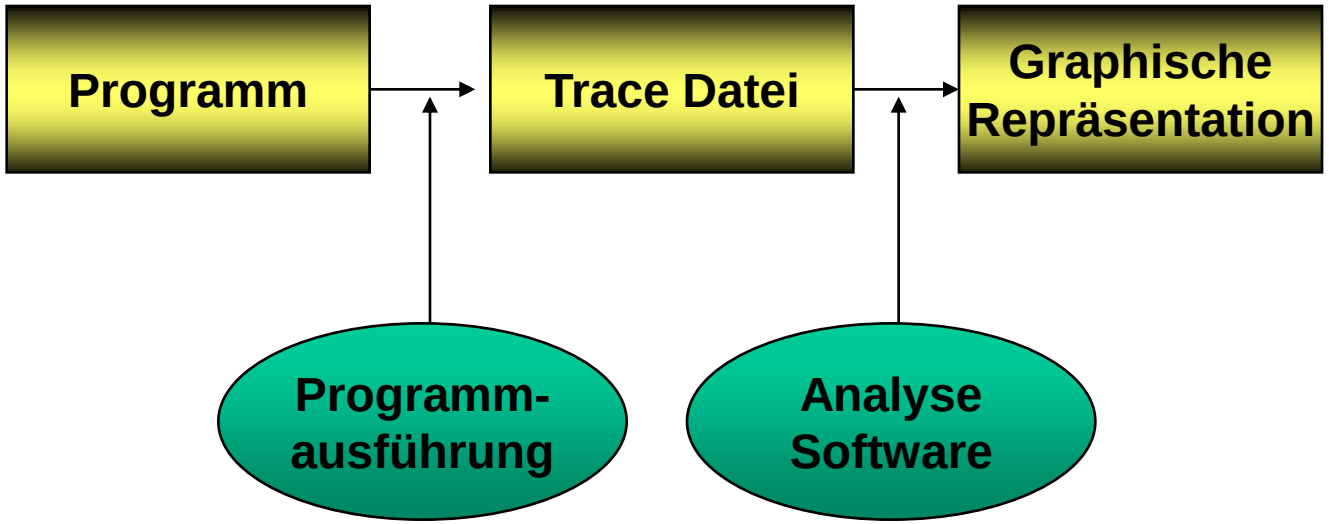
Runtime Results: Ray Tracer

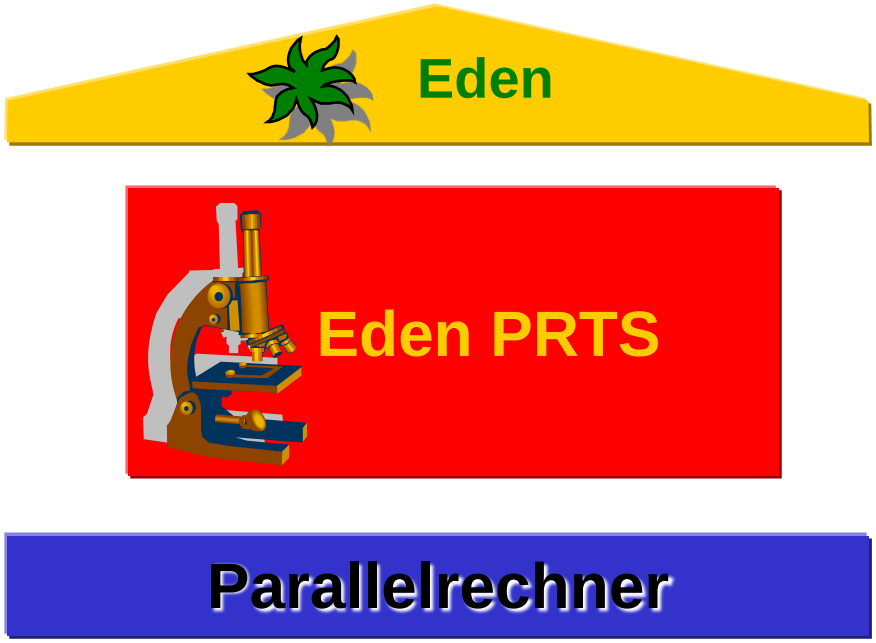


sequential runtime: **54,78 sec**

 Platform: 6 UltraSparc IIi (300 MHz), 128 MB, 2 MB Cache, 100 Mbit Net, Solaris 2.6

Profiling





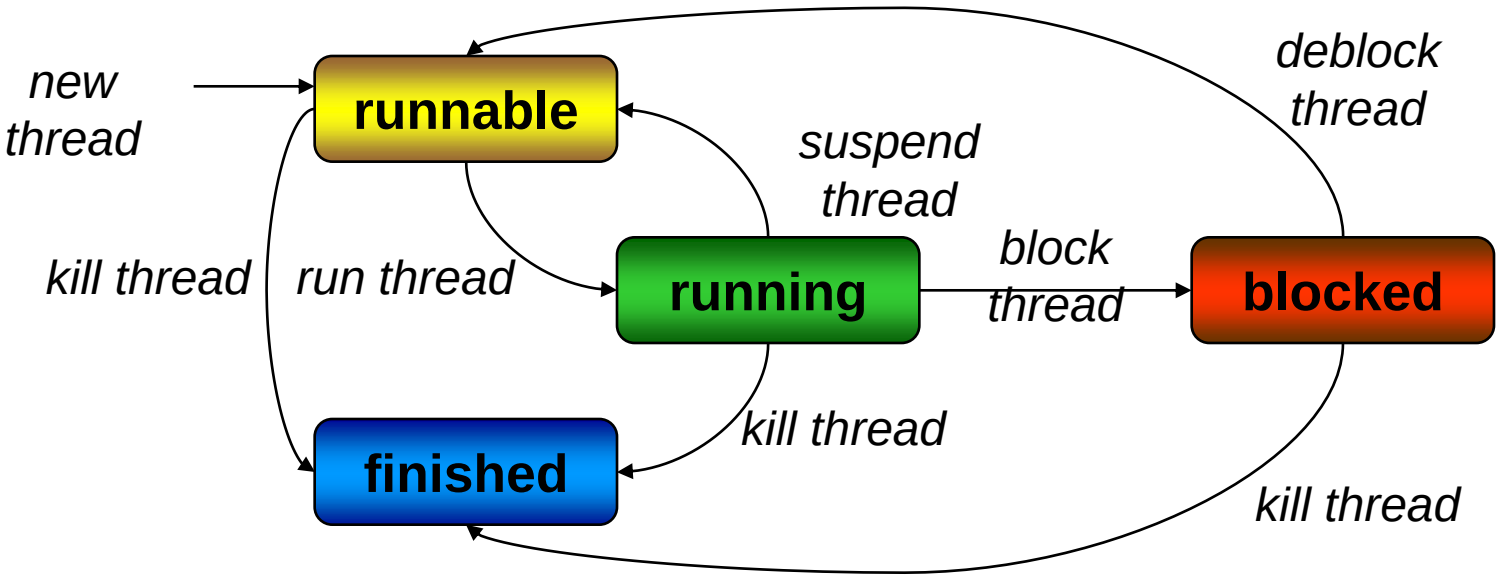
1. Instrumentierung des PRTS
➔ Trace Generierung

start machine ...
start thread ...
send message ...
....

2. Eden Trace Viewer (EdenTV)
➔ Trace Analyse und Repräsentation

Eden Threads und Prozesse

- Ein Prozess besteht aus einer Menge von Threads (Berechnungsfäden).
- Thread State Transition Diagram:

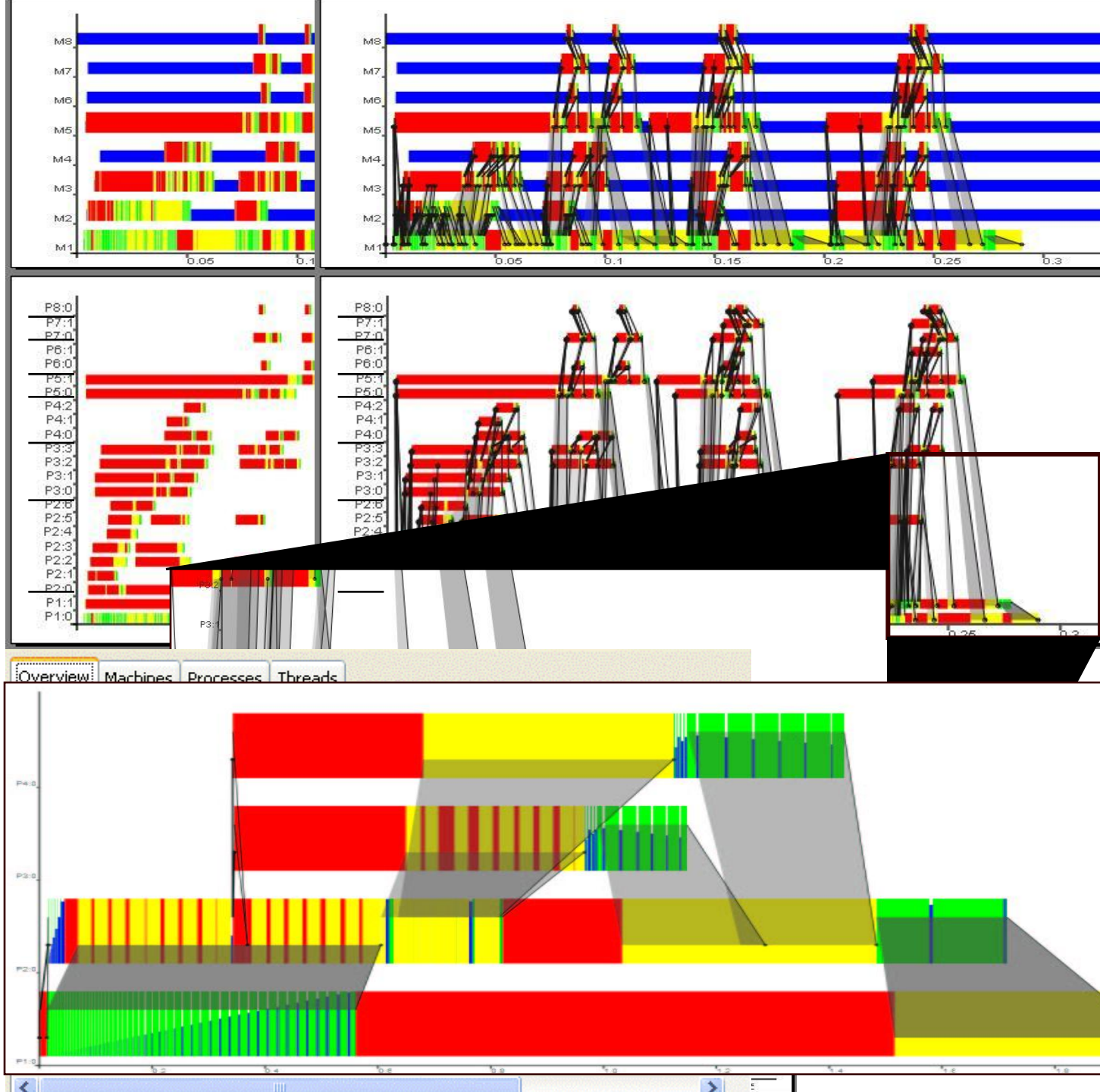


EdenTV

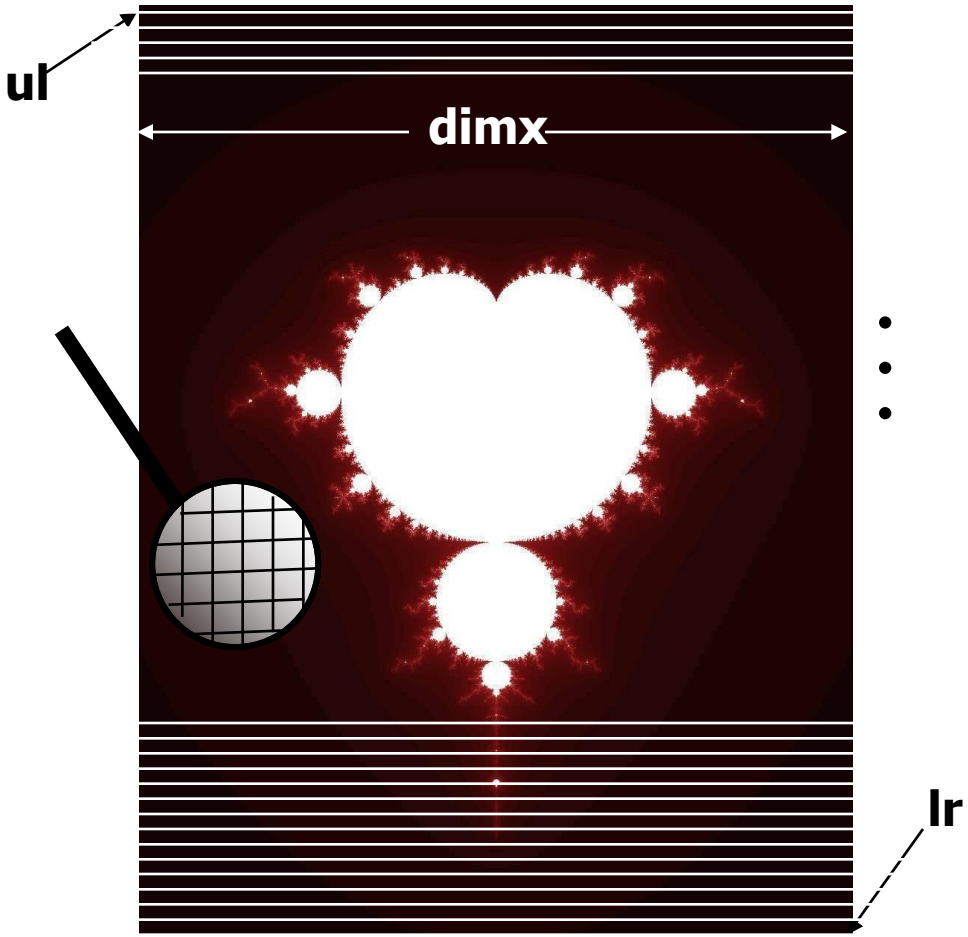
- Diagramme:
  - Maschinen
  - Prozesse
  - Threads

- Nachrichten-einblendung für
  - Maschinen
  - Prozesse

- zooming
- Nachrichten-ströme
- Zusatzinfos
- garbage collection
- ...



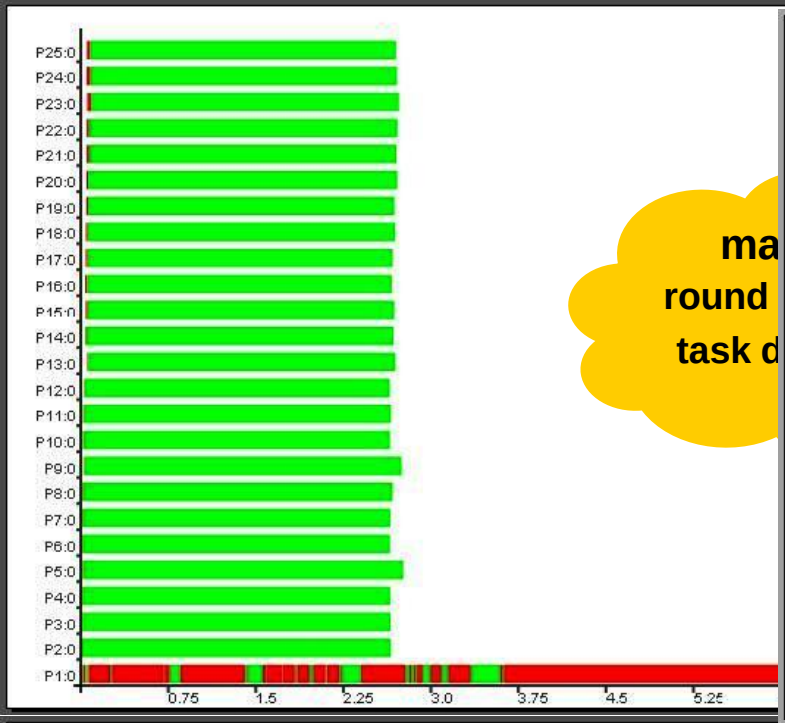
Example: Parallel Functional Program for Mandelbrot Sets



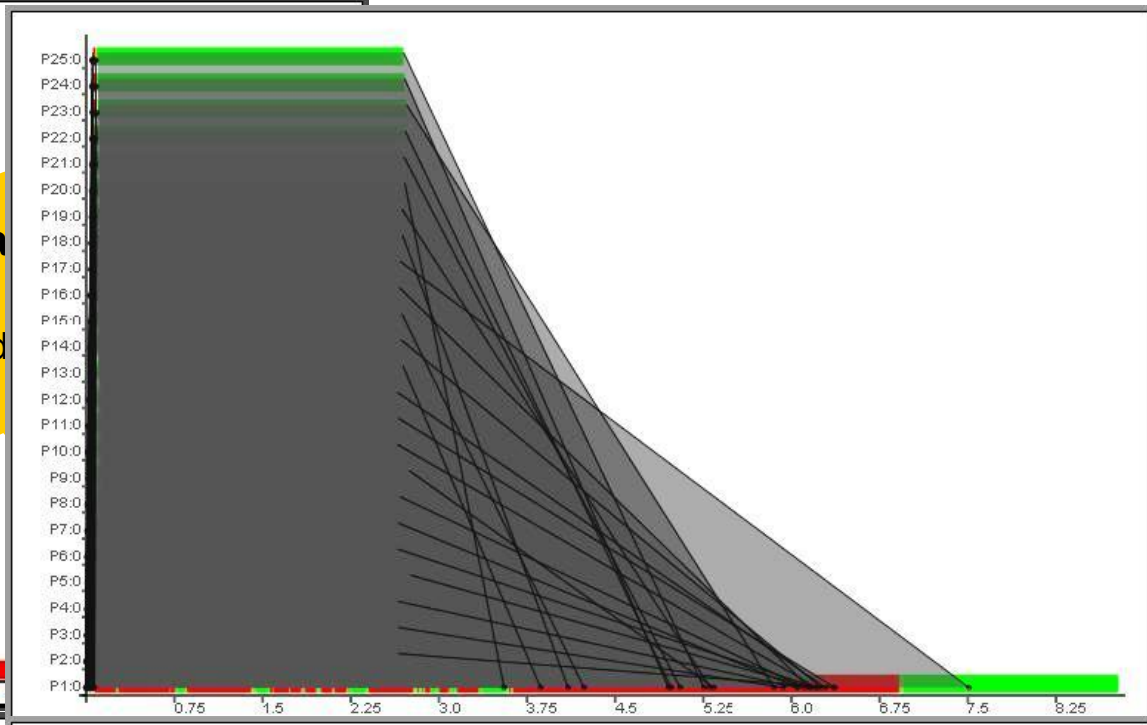
Idea: parallel computation of lines

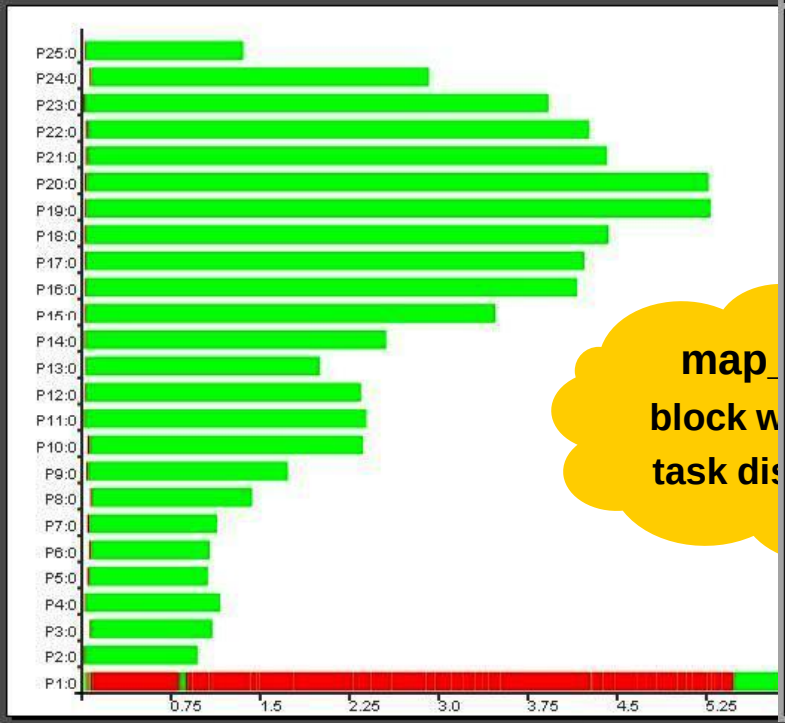
```
image :: Double -> Complex Double -> Complex Double -> Integer -> String
image threshold ul lr dimx
 = header ++ (  concat $ map xy2col lines  )
 where
   xy2col ::[Complex Double] ->  String
   xy2col line    = concatMap (rgb.(iter threshold (0.0 :+ 0.0) 0)) line
    (dimy, lines)  = coord ul lr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

Replace map by map_farm, map_farmB or workpool

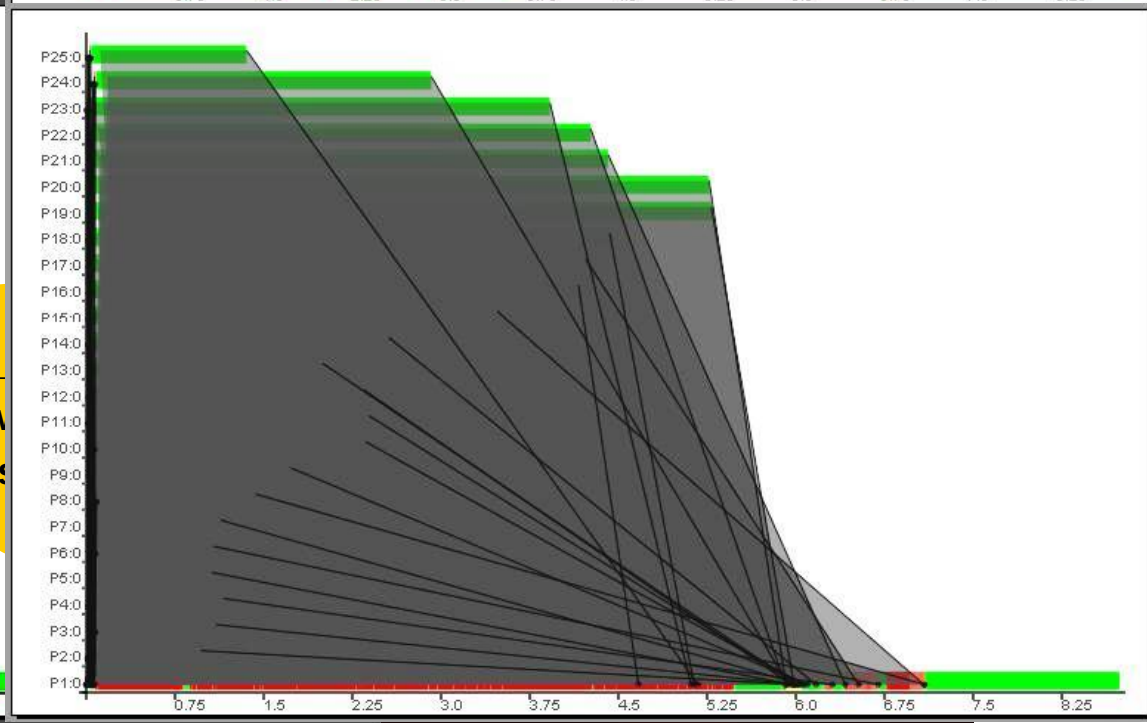


ma
round
task d





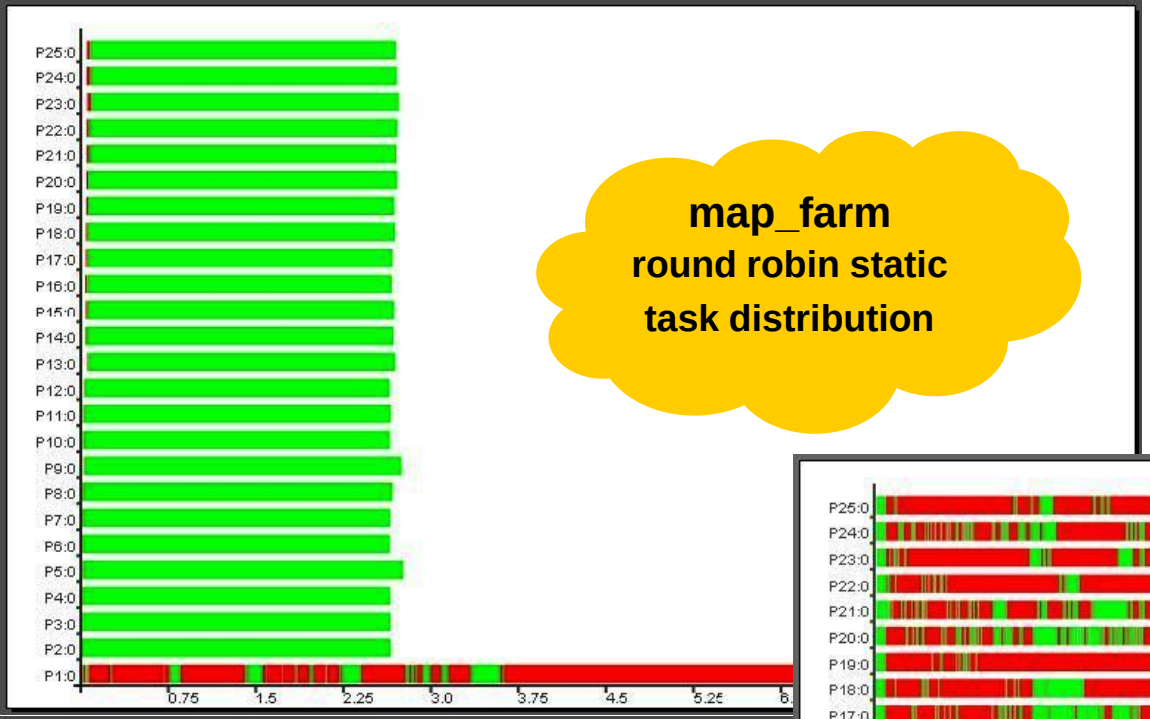
map_
block w
task dis



Mandelbrot Traces

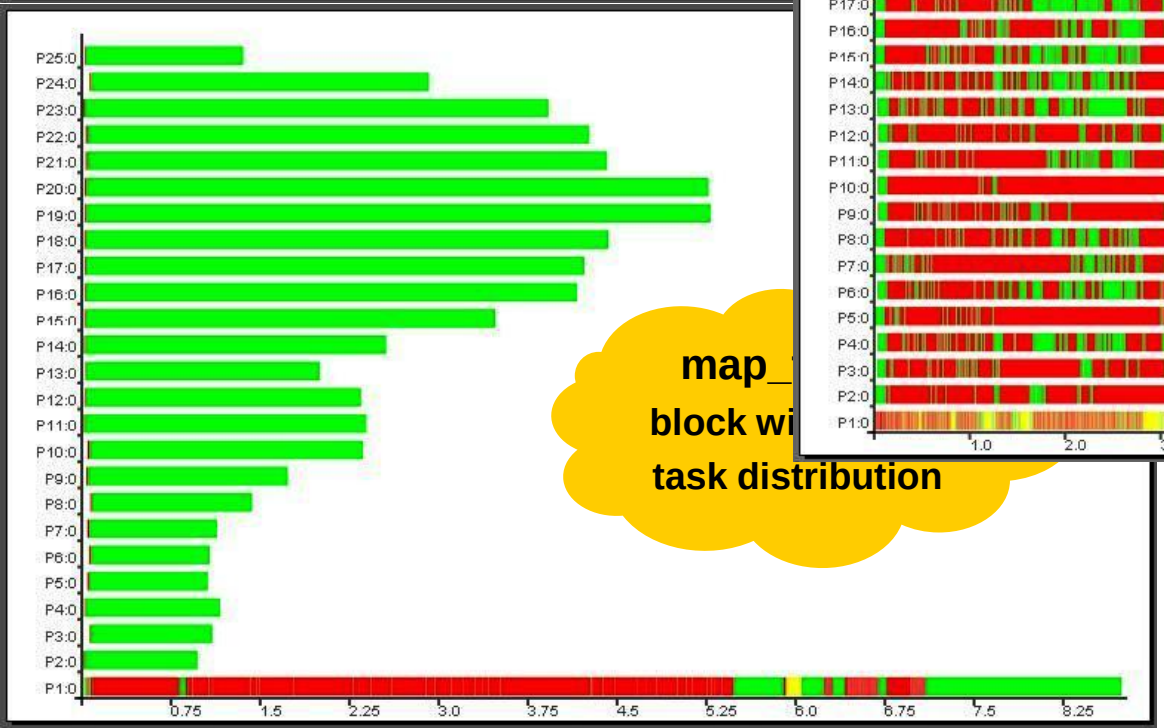
Problem size: 2000 x 2000

Platform: Beowulf cluster Heriot-Watt-University,



map_farm
round robin static task distribution

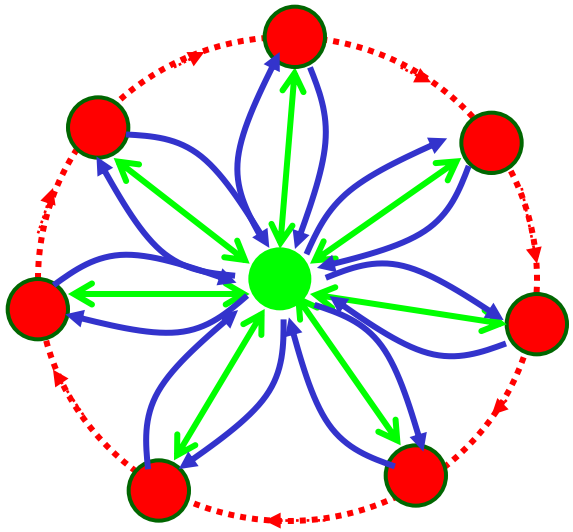
workpool
dynamic task distribution



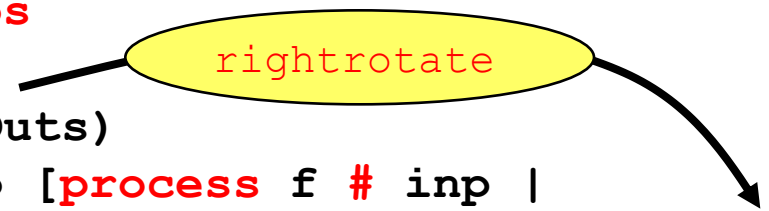
map_
block wi
task distribution

Motivation für dynamische Kanäle in Eden

Beispiel: Definition eines Prozessrings

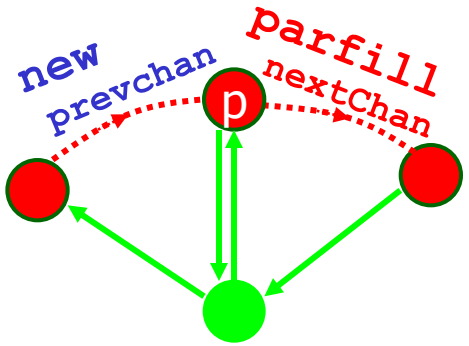


```
ring :: (Trans i,Trans o,Trans r) =>
   ((i,r) -> (o,r))   -> -- ring process fct
   [i] -> [o]            -- input-output fct

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

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



Problem: Ringverbindungen nur indirekt über Elternprozess

Dynamische Kanäle in Eden



- Kanalerzeugung

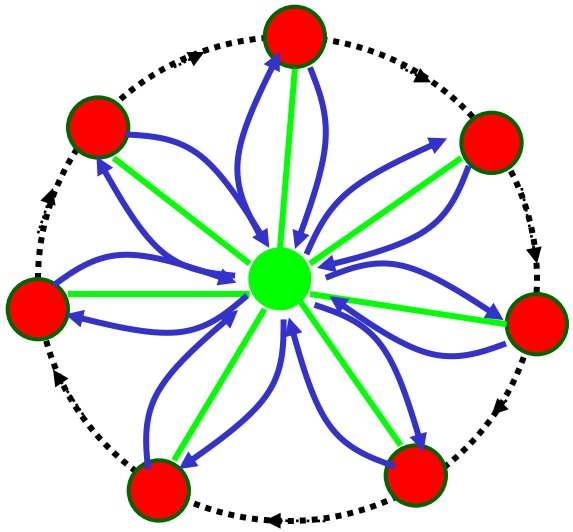
```
new ::     Trans a =>
  (ChanName a -> a -> b) -> b
```

- Kanalverwendung

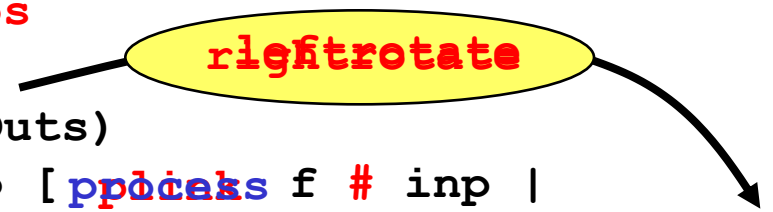
```
parfill :: Trans a =>
  ChanName a -> a -> b  -> b
```

```
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)
        )
```

Dynamisches Statisches Ring Skelett

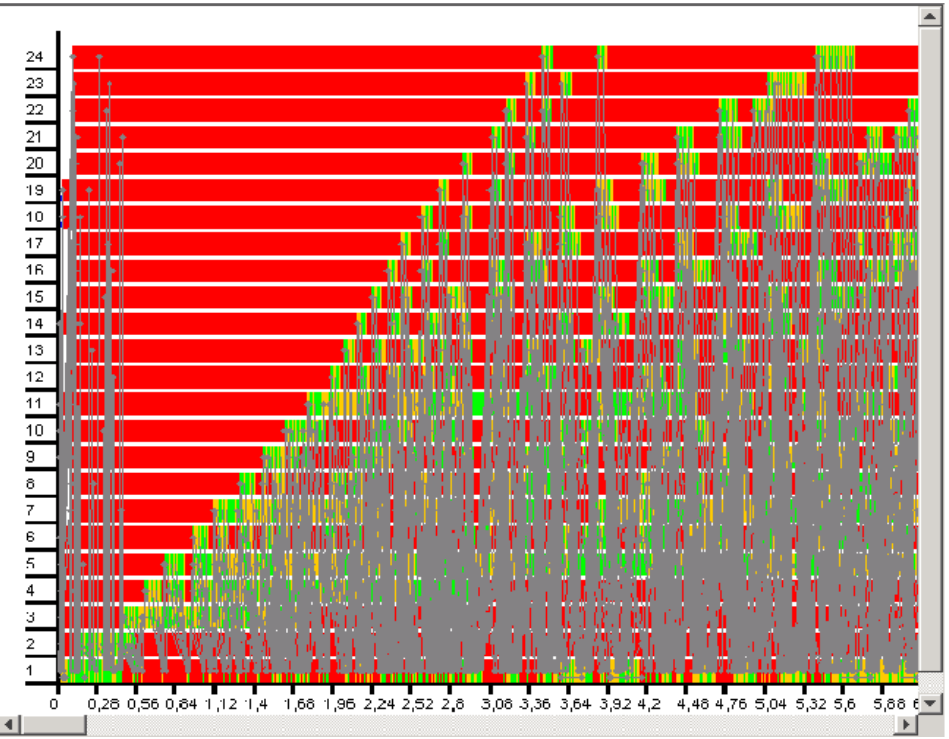


```
ring :: (Trans i,Trans o,Trans r) =>
   ((i,r) -> (o,r))   -> -- ring process fct
   [i] -> [o]            -- input-output fct

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

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



Problem: Ringverbindungen nur indirekt über Elternprozess

Traceprofile
Statisches vs dynamisches Ringskelett

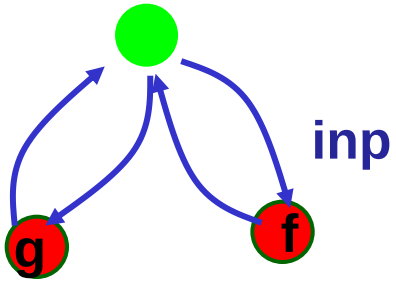


Statisches Ringskelett - **Alle Kommunikationen laufen über den Generator-prozess (Nummer 1).**

Dynamisches Ringskelett – **Ringprozesse kommunizieren direkt.**

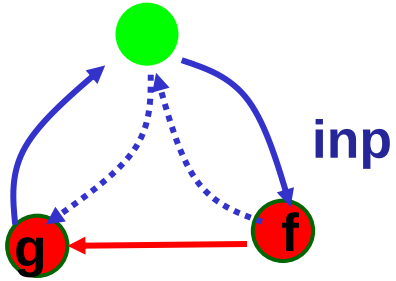
Das „Remote Data“-Konzept

- Funktionen:
  - Freigabe lokaler Daten mittels release :: a -> RD a
  - Holen freigegebener Daten mittels fetch :: RD a -> a
- Ersetze
  - (process g # (process f # inp))



durch

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



Ring Skelett mit 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 <- mzip 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
```

Implementierung mittels dynamischer Kanäle

-- 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)