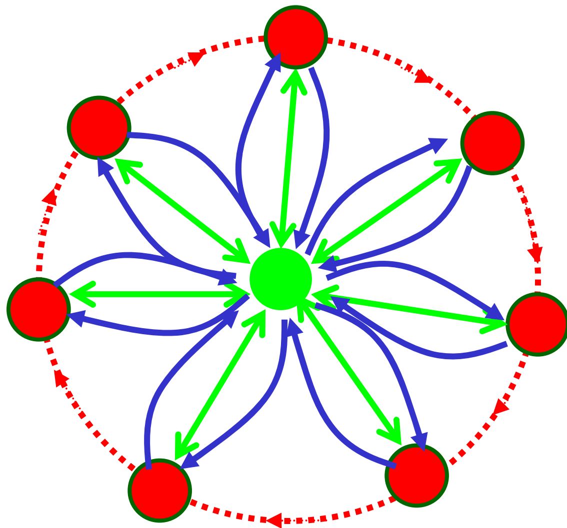
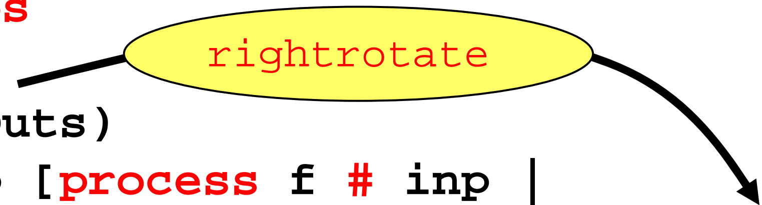
Motivation für dynamische Kanäle in Eden

Beispiel: Definition eines Prozessrings

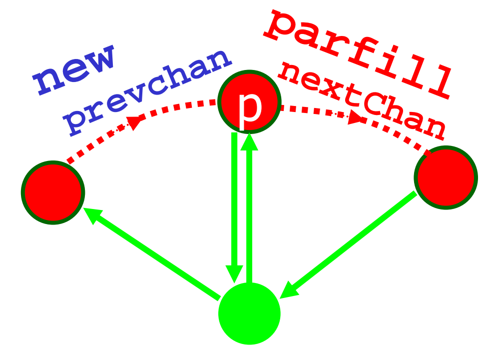


```
ring :: (Trans i,Trans o,Trans r) =>
   ((i,r) -> (o,r))  -> -- ring process fct
   [i] -> [o]           -- input-output fct

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

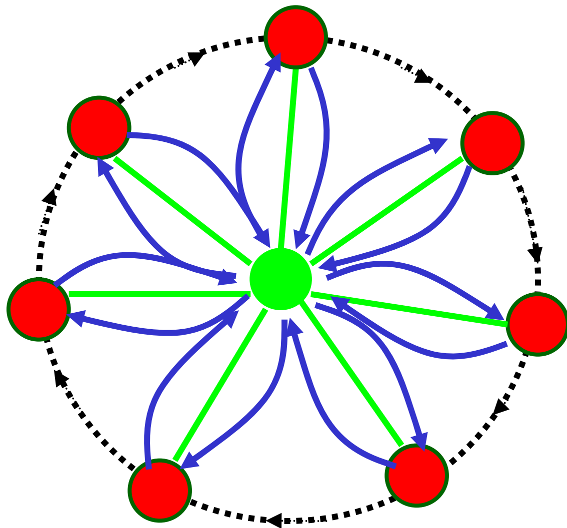
```
new ::     Trans a =>
  (ChanName a -> a -> b) -> b
```

- 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
       (to, next)
          = f (fromP, prev)
      in
        parfill nextChan next
                 (toP, prevChan)
          )
```

Dynamisches ~~Statisches~~ Ring Skelett

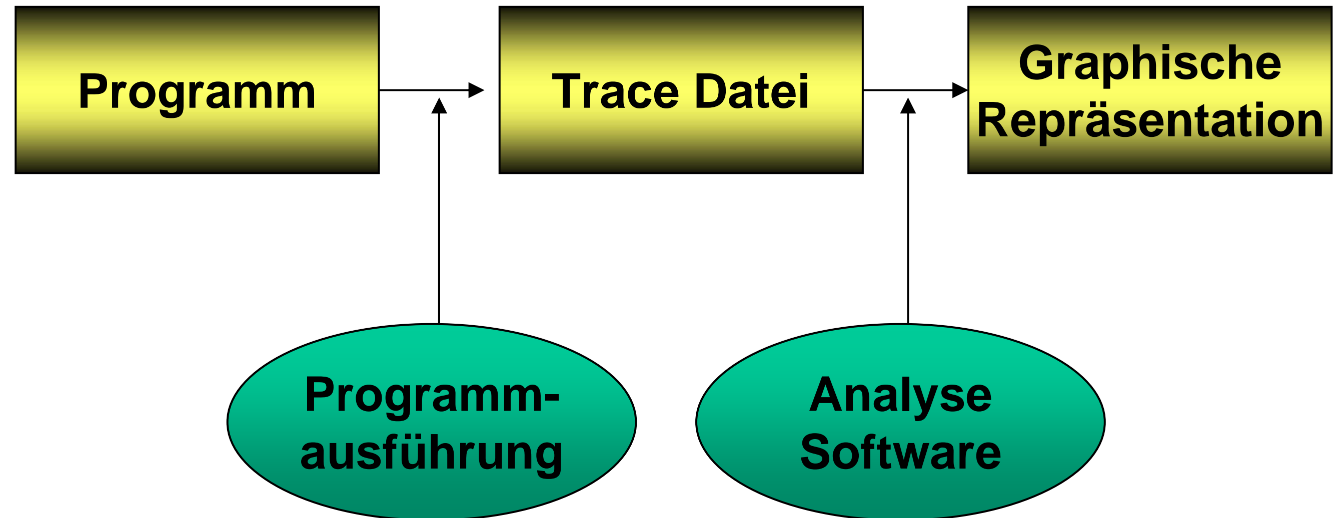


```
ring :: (Trans i,Trans o,Trans r) =>
   ((i,r) -> (o,r))  -> -- ring process fct
   [i] -> [o]           -- input-output fct

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

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

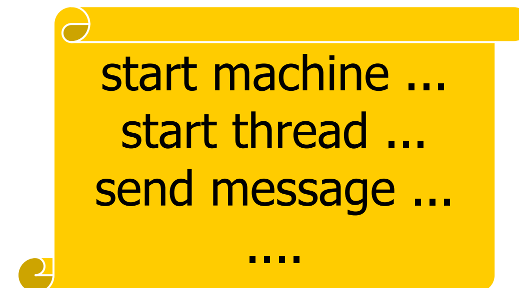
rightrotate / leftrotate

Problem: Ringverbindungen nur indirekt über Elternprozess

Profiling



1. Instrumentierung des PRTS
   → Trace Generierung

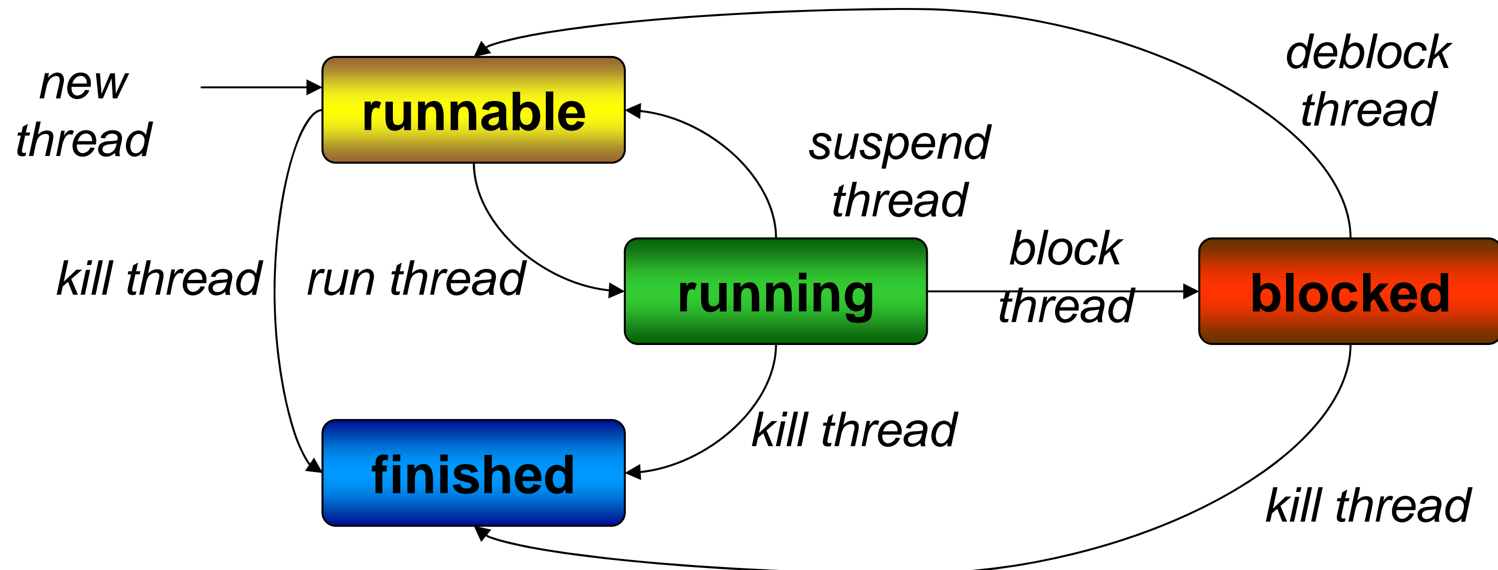


2. Eden Trace Viewer (EdenTV)
   → Trace Analyse und Repräsentation

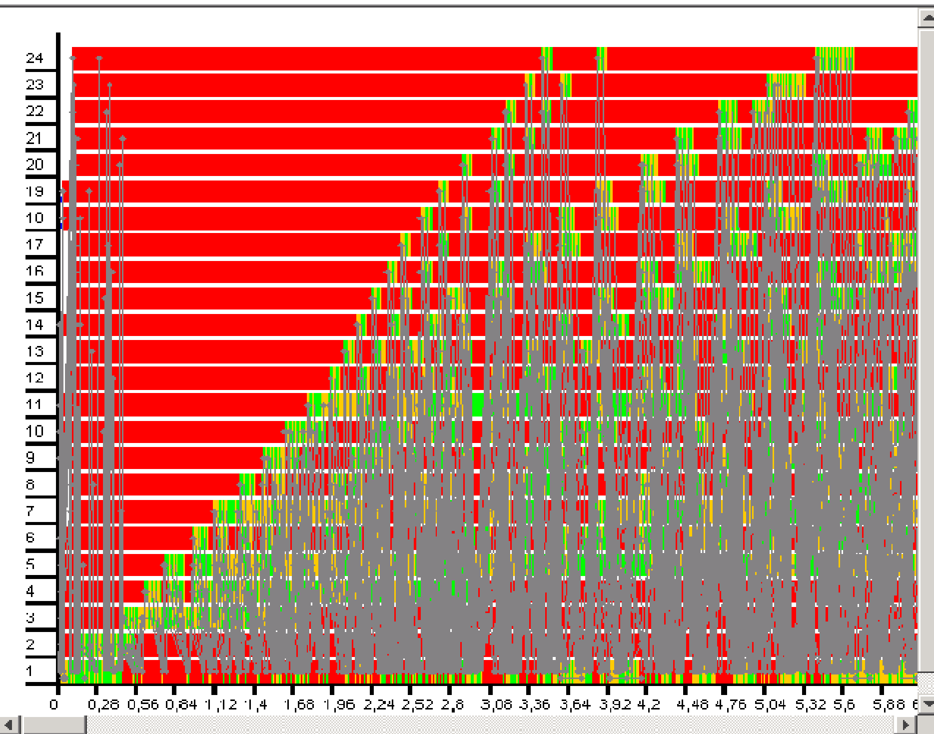
Eden Threads und Prozesse

- Ein Prozess besteht aus einer Menge von Threads (Berechnungsfäden).

- Thread State Transition Diagram:

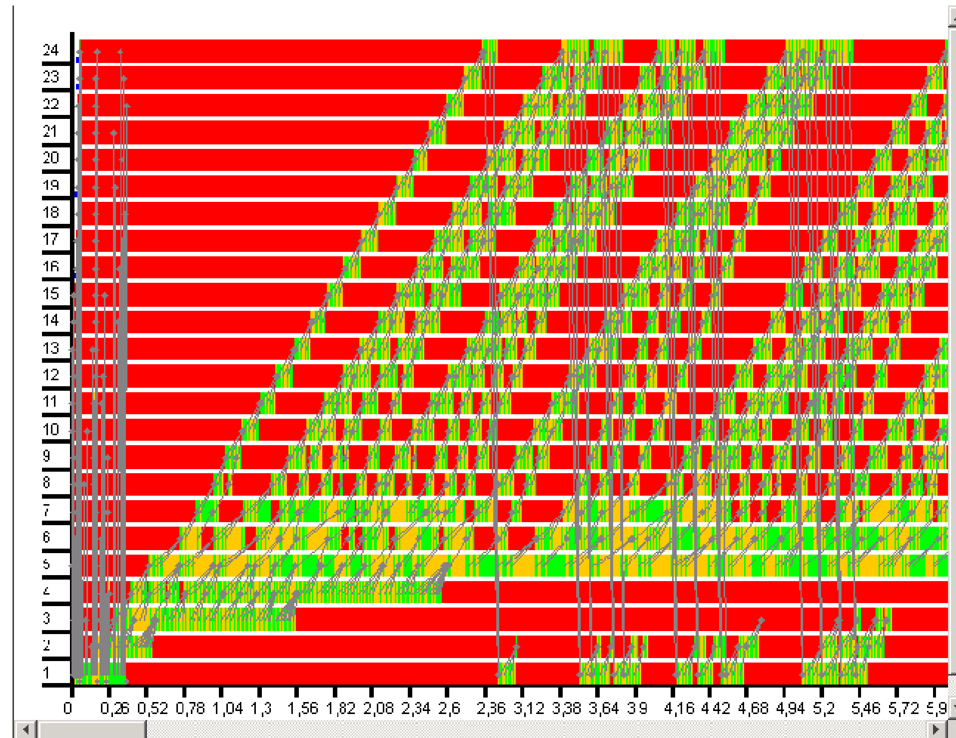


Traceprofile
Statisches vs dynamisches Ringskelett



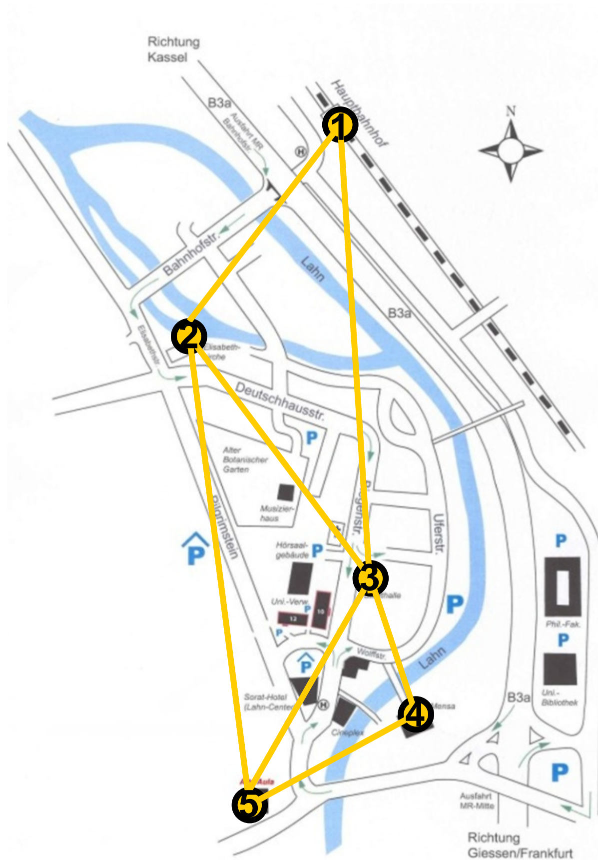
Statisches Ringskelett - Alle Kommunikationen laufen über den Generator-prozess (Nummer 1).

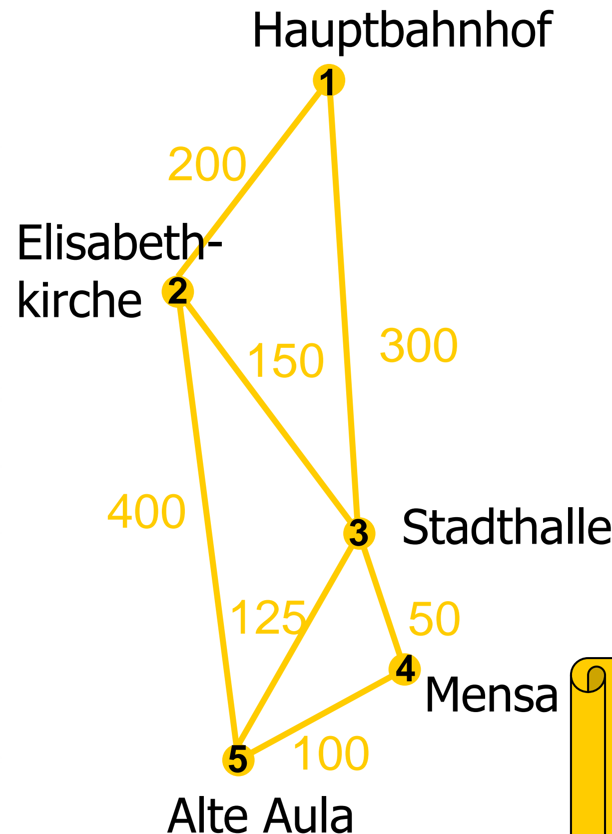
Dynamisches Ringskelett – Ringprozesse kommunizieren direkt.



Weiteres Beispielproblem: Kürzeste Wege

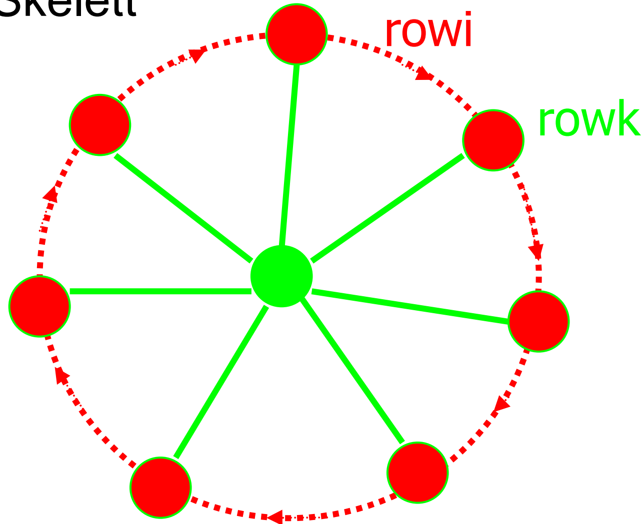
Landkarte -> Graph -> Adjazenzmatrix/ Entfernungsmatrix





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

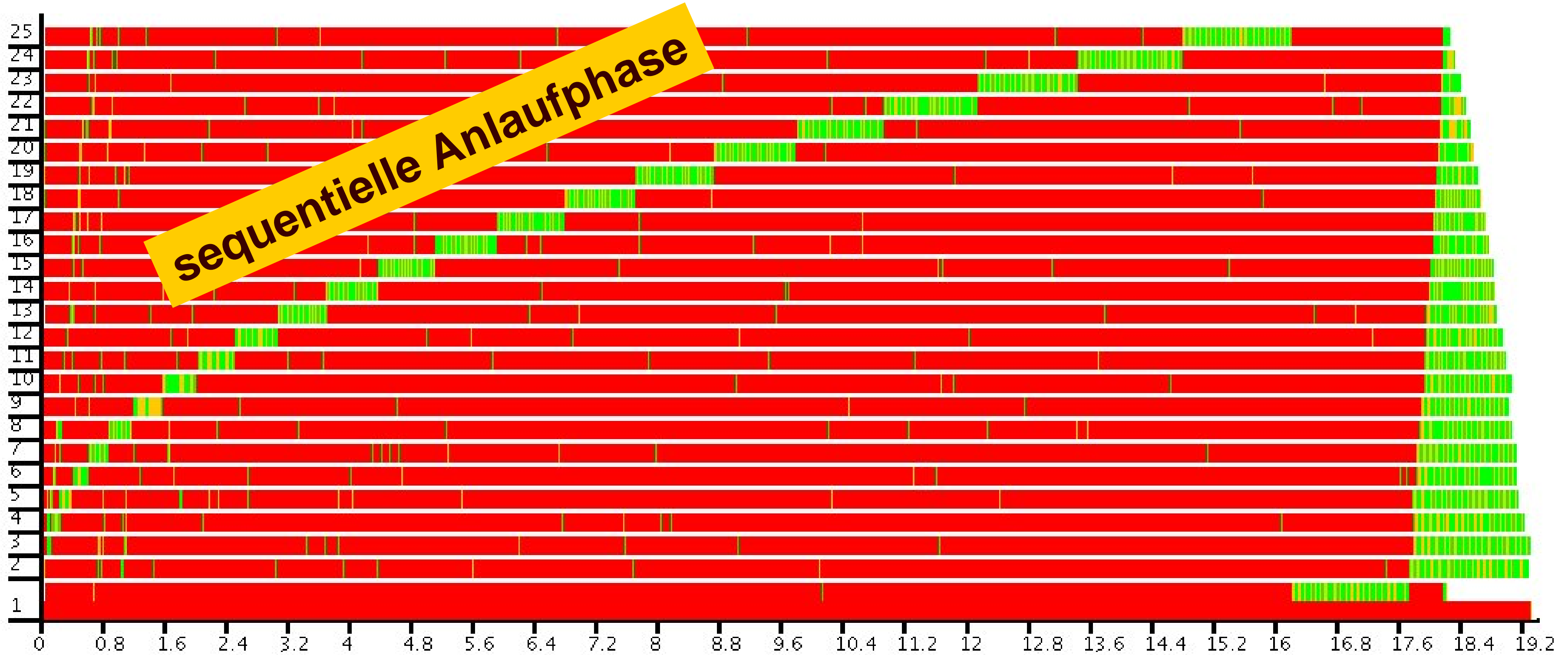
Wie lang ist der kürzeste Weg von A nach B für beliebige Knoten A und B?



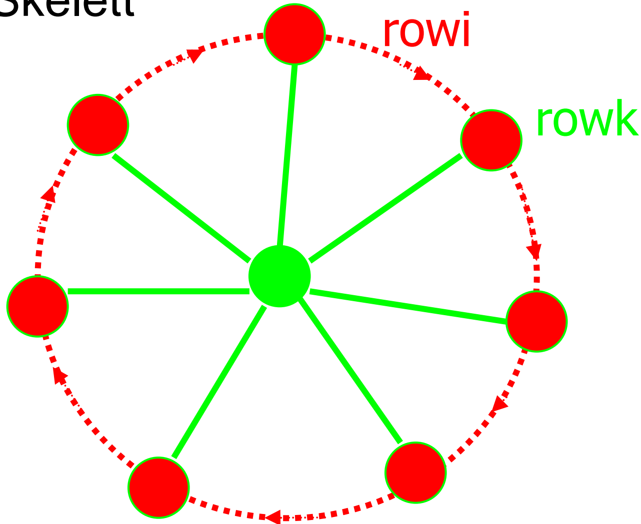
Warshalls Algorithmus in Prozessring

```
ring_iterate :: Int -> Int -> Int ->
                    [Int] -> [[Int]] -> ([Int], [[Int]])
ring_iterate size k i rowk (rowi:xs)
    | i > size   =  (rowk, [])     -- Ende der Iterationen
    | i == k     =  (rowR, rowk:restoutput) -- sende eigene row
    | otherwise  =  (rowR, rowi:restoutput) -- aktualisiere 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))
```

Trace zu parallelem Warshall



Probleme: Datenabhängigkeit und fehlender Auswertungsbedarf.

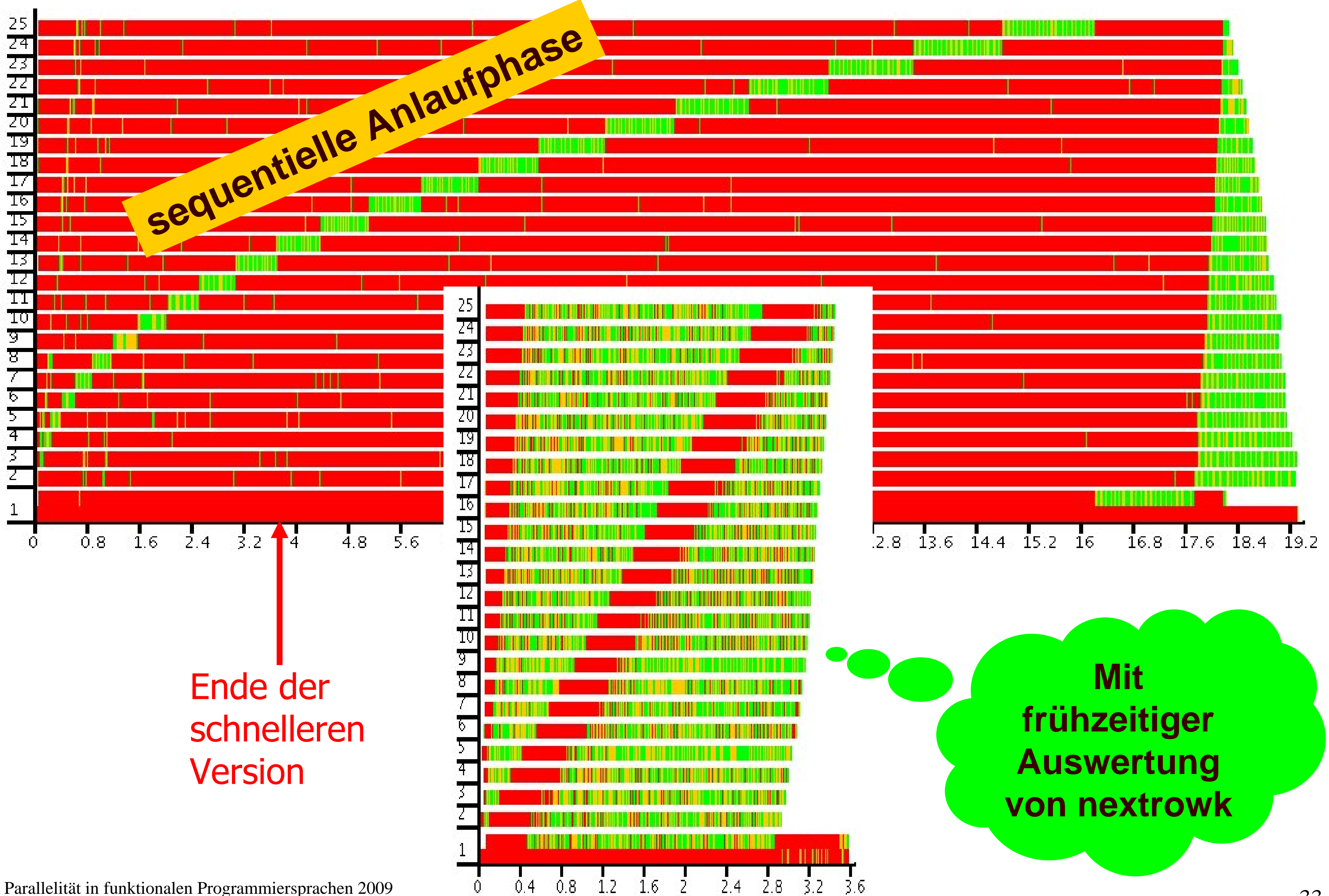


Warshalls Algorithmus in Prozessring

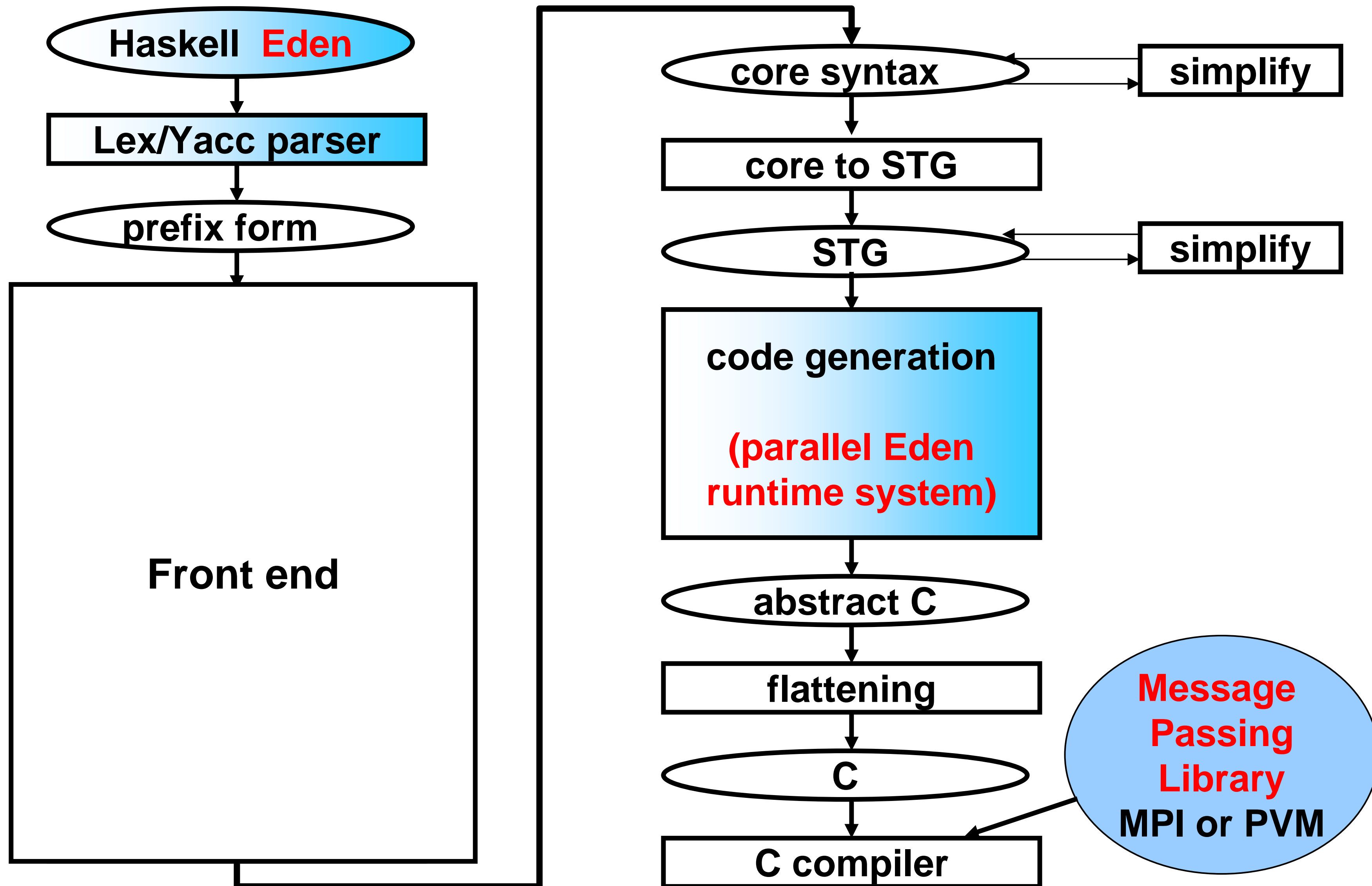
Erzwinge frühzeitige Auswertung von nextrowk

```
ring_iterate :: Int -> Int -> Int ->
                     [Int] -> [[Int]] -> ([Int],
ring_iterate size k i rowk (rowi:xs)
    | i > size   =  (rowk, [])    -- Ende der Iteration
    | i == k     =  (rowR, rowk:restoutput) -- sende eigene row
    | otherwise =  (rowR, rowi:restoutput) -- aktualisiere 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 zu parallelem Warshall



Glasgow Haskell Compiler
& Eden Erweiterungen



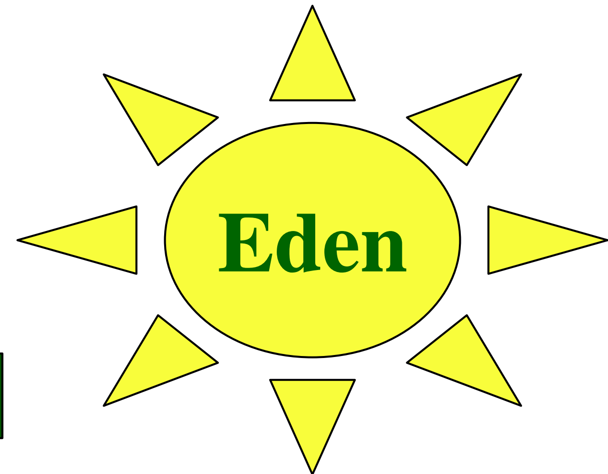
Edens paralleles Laufzeitsystem

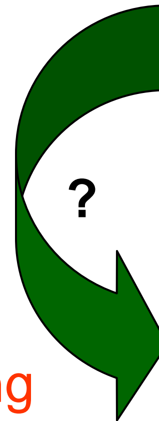
Modifikation von GUM, dem Laufzeitsystem von GpH (Glasgow Parallel Haskell):

- **Wiederverwendung**
  - **Threadverwaltung:** **Heap Objekte, Thread Scheduler**
  - **Speicherverwaltung:** **lokale Garbage Collection**
  - **Kommunikation:** **Graph Pack- und Entpack-Routinen**
- **Neuentwicklung**
  - **Prozessverwaltung:** **Laufzeittabellen, Erzeugung und Termination, Lastbalancierung und Arbeitsverteilung**
  - **Kommunikationsroutinen:** **adaptive Kommunikation, Bypassing**
- **Vereinfachungen**
  - **kein „virtual shared memory“ (globaler Adreßbereich)**
  - **keine Globalisierung unausgewerteter Daten**

Implementierung von Eden

- Parallele Programmierung auf hoher Abstraktionsebene
  - explizite Prozessdefinitionen
  - implizite Kommunikation

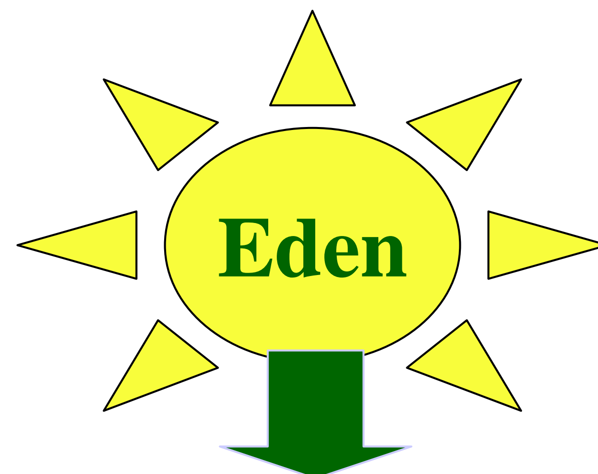




- Automatische Prozessverwaltung
  - verteilte Graphreduktion
  - Verwaltung von Prozessen und Kommunikationen

Ansatz

- Parallele Programmierung auf hoher Abstraktionsebene
  - explizite Prozessdefinitionen
  - implizite Kommunikation



- Automatische Prozessverwaltung
  - verteilte Graphreduktion
  - Verwaltung von Prozessen und Kommunikationen

Ebenenstruktur

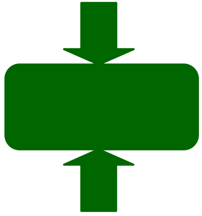
Edenprogramme

Skelettbibliotheken

Eden Modul

Primitive Operationen

Paralleles GHC Laufzeitsystem



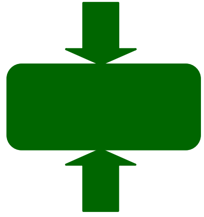
Das Eden Modul

Typklasse **Trans**
- enthält alle übertragbaren Datentypen
- überladene Kommunikationsfunktionen für Listen (-> streams) und Tupel

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

explizite Definitionen von **process**, **(#)** und **Process**

explizite **Kanäle**
- type ChanName a = [ChanName’ a]
- data ChanName’ a = Chan Int# Int# Int#
- createDC :: a -> (ChanName a, a)
- writeDC :: ChanName’ a -> b -> ( )

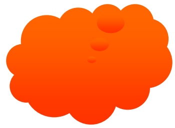


Die Typklasse Trans

```
class NFData a => Trans a where
   sendChan     :: a -> ( ) ;  sendChan x    = rnf x `seq` sendVal x
   tupsize      :: a -> Int;  tupsize _ = 1
   writeDCs     :: ChanName a -> a -> ( )
   writeDCs (cx:_) x = writeDC cx x
```

```
instance Trans a => Trans [a]
 where
  sendChan x = sendStream x

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

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



Schnittstelle zum parallelen Laufzeitsystem

Primitive Operationen stellen Basisfunktionalität bereit:

- remote process creation — **createProcess#**

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

- channel administration
  - →create communication channel(s) — **createDC#**
  - →connect communication channel — **setChan#**

```
writeDC chan a
 = setChan chan
    `seq` sendChan a
```

- communication
  - →send single value — **sendVal#**
  - →send head element of a list — **sendHead#**
- general — **noPE#, selfPE#**

merge#