# Kapitel 12: Parallelisierung funktionaler Sprachen

Lernziele dieses Kapitels:

1. Entwicklung von Struktur
   - nebenläufige Threads: Multilisp
   - Schleifen mit parallen Aktionen: Sisal
   - parallele Rekursion, Kostenmodell: Nesl
   - Schedule: Para-functional Programming in Haskell
   - Strategien: Glasgow parallel/distributed Haskell
2. Der Skelettansatz: Eden und HDC
3. Metaprogrammierung: MetaOCaml / MPI

# Multilisp

- entwickelt von Robert Halstead am Massachusetts Institute of Technology Anfang 1980er
- basiert auf Scheme: LISP-Dialekt mit Lexical Scoping
- Prinzip: unabhängige Kontrollfäden in einem gemeinsamen Adressraum, sog. *Threads*.
- Parallele Abarbeitung der Threads durch Mehrprozessor-System.
- `future` $X$ erzeugt in einem Thread $T$ einen neuen Thread, dessen Wert $X$ ist.
  - Auswertung von $X$ wird parallel zu $T$ gestartet.
  - (`future` $X$) terminiert sofort in $T$ und liefert eine Referenz.
  - Sobald Auswertung von $X$ beendet, ersetzt der Wert von $X$ die Referenz.
  - Wird der *Wert* (nicht die *Referenz*) von $X$ von $T$ gebraucht (z.B. für +), wird $T$ blockiert solange $X$ noch nicht ausgewertet ist.

# Quicksort in Multilisp

```
(defun qsort (l) (qs l nil))

(defun qs (l rest)
  (if (null l)
      rest
      (let ((parts (partition (car l) (cdr l))))
        (qs (left-part parts)
            (future (cons (car l)
                          (qs (right-part parts)
                              rest)))))))
(defun left-part  (p) (car p))
(defun right-part (p) (cdr p))
```

# Parallele Partitionierung

```
(defun partition (elt lst)
    (if (null lst)
        (bundleparts nil nil)
        (let ((cdrparts (future (partition elt (cdr lst))))
             (if (> elt (car lst))
                 (bundle-parts
                    (cons (car lst) (future (left-part cdrparts)))
                    (future (right-part cdrparts)))
                 (bundle-parts
                    (future (left-part cdrparts))
                    (cons (car lst) (future (right-part cdrparts))
    ))))))))
(defun bundle-parts (x y) (cons x y))
```

# Implementierung von Multilisp

- Garbage Collection mit Baker-Algorithmus
  - umkopieren von Daten bei der Verwendung von oldspace nach newspace
  - Freigabe von oldspace, newspace wird zu oldspace
- Concert Multiprocessor
  - bis zu 32 Motorola 68000 Prozessoren
  - Ausbau 1982: 8 Prozessoren
  - Verbindung der Prozessoren mit dual-ported memory und mit Bus
  - Speedup von fast 8 bei 8 Prozessoren erreichbar
- auch implementiert auf 128-Prozessor Butterfly, nicht so guter Speedup

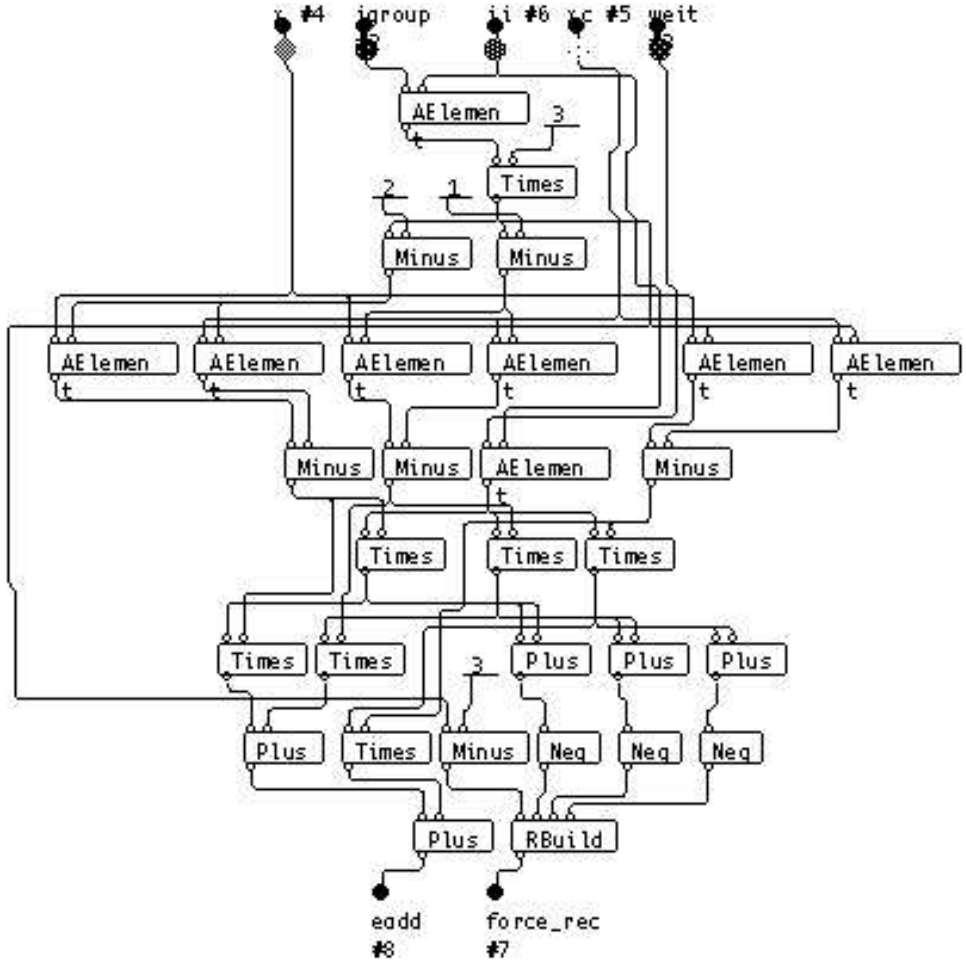
# Sisal

- Streams and Iteration in a Single Assignment Language
- Anwendungsgebiete: numerisches Rechnen, Konkurrenz zu FORTRAN
- Konstrukte: Schleifen, Arrays
- Definition 1983 durch James McGraw u.a., Standard 1990
- Entwickelt: Manchester University, Lawrence Livermore National Laboratory, Colorado State University und DEC
- Zielrechner: diverse Cray's, Datenflußrechner, Transputer und systolische Arrays
- keine Seiteneffekte.
- keine explizite Parallelität, d.h. keine Spezifikation unabhängiger Berechnungen durch den Programmierer $\Rightarrow$ keine Kontrolle, wann auf welchem Prozessor welche Berechnung stattfindet.

# Sisal-Programm zur numerischen Integration

```
for initial
        N       := initial_n;
        Count   := 1;
        H       := (b - a) / double_real(N);
        ...
repeat
        N       := old N * 2;
        Count   := old Count + 1;
        H       := old H / 2.0d0;
        Area    := for i in 1, N
                        ai := a + double_real(i-1) * h;
                        bi := ai + h;
                   returns value of sum trap_area( ai, bi )
                   end for
until abs( area - old area ) < epsilon | Count > 10
```

# Sisal-Zielcode, Datenflussgraph



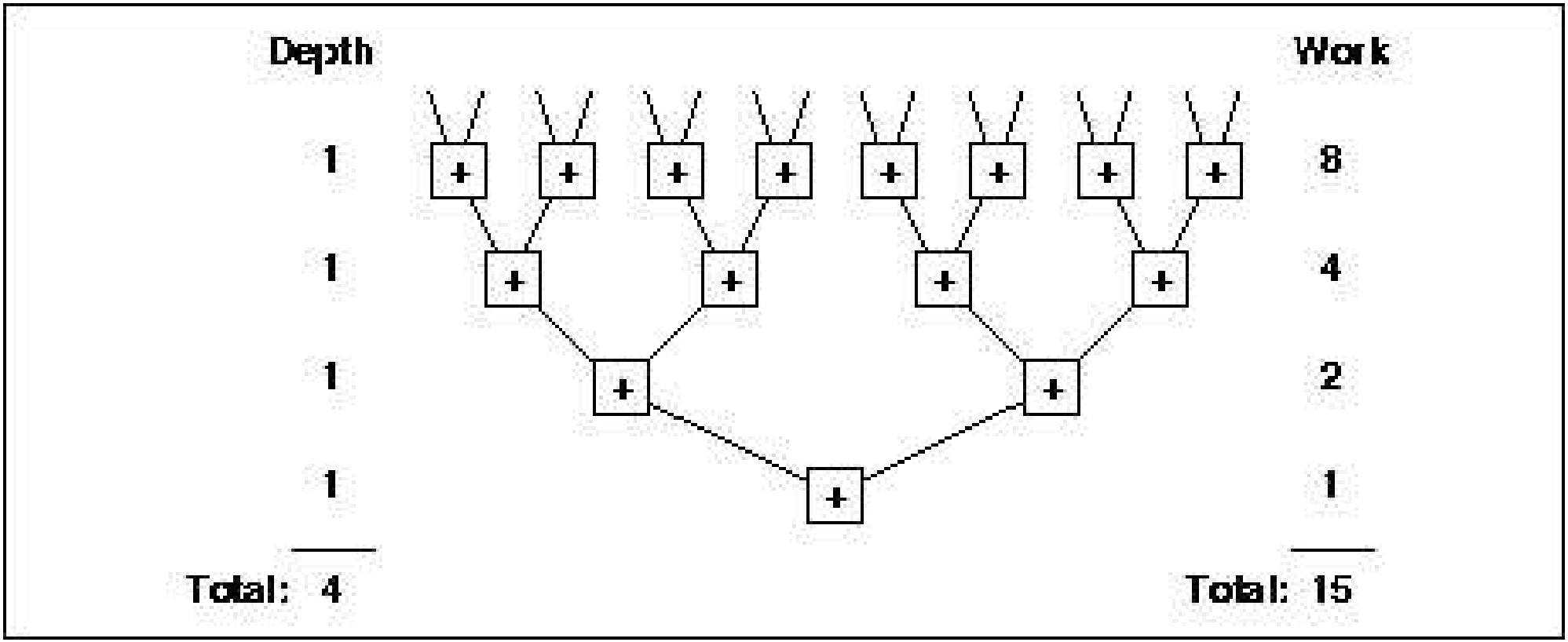
# Nesl

- entwickelt 1994 von Guy Blelloch an der Carnegie Mellon University
- Nested Data-Parallel Language, basiert auf ML und SETL
- Zielplattformen: Crays, Connection Machine, MPI
- explizite Parallelität: Programmierer spezifiziert,
  - was parallel berechnet werden kann,
  - aber nicht: wo wann welche Berechnung stattfindet.
- Taskparallelität mit Rekursion (Divide-and-Conquer)
- Datenparallelität
  - map-Typ: Anwendung einer Funktion auf alle Elemente einer Sequenz
  - reduce-Typ: z.B. Summe der Elemente einer Sequenz

# Quicksort in Nesl

```
function quicksort(a) =
if (#a < 2) then a
else
  let pivot   = a[#a/2];
      lesser  = { e in a | e <  pivot };     Datenparallelität
      equal   = { e in a | e == pivot };          ‘‘
      greater = { e in a | e >  pivot };          ‘‘
      result  = { quicksort(v) : v in [lesser,greater] };
                Taskparallelität
  in result[0] ++ equal ++ result[1];
```

# Nesls Performance-Modell: Work & Depth



# Batcher Sort in Nesl

Work: $\Theta(n \cdot (\log n)^2)$, Depth: $\Theta((\log n)^2)$

```
function bitonic_sort(a) =
if (#a == 1) then a
else let bot = subseq(a,0,#a/2);
         top = subseq(a,#a/2,#a);
         mins = { min(bot,top):bot;top };
         maxs = { max(bot,top):bot;top };
     in flatten({ bitonic_sort(x) : x in [mins,maxs] })

function batcher_sort(a) =
if (#a == 1) then a
else
    let b = { batcher_sort(x) : x in bottop(a) };
    in bitonic_sort(b[0]++reverse(b[1]))
```

# Implementierte Algorithmenklassen

- suchen und sortieren auf Sequenzen
- Baum-und Graphalgorithmen
  - kürzeste Wege
  - Zusammenhangskomponenten
  - maximale unabhängige Menge
- Geometrie
  - konvexe Hülle
- numerisches Rechnen
  - Fourier-Transformation
  - Matrix-Operationen (dense und sparse)
  - N-Körper-Simulation

# Para-functional Programming in Haskell

- entwickelt von Paul Hudak an der Yale University, 1986
- wird übersetzt in eine spezielle Form von MultiLisp

**Scheduled Expressions**

- Kontrolle der Ausführungsreihenfolge
- Syntax: *exp* `sched` *schedule*

**Mapped Expressions**

- Kontrolle des Ausführungsortes (Prozessor)
- Syntax: *exp* `on` *pid*

# Scheduled Expressions

- Labeling von Ausdrücken, auf die scheduled expressions Bezug nehmen.
  Bsp.: $lab@exp$.
- Bezug auf drei Arten von Ereignissen für $lab@exp$
  1. $Dlab$: Anforderung der Auswertung von $exp$
  2. $\hat{}lab$: Beginn der Auswertung von $exp$
  3. $lab\hat{}$: Ende der Auswertung von $exp$
- Komposition von Schedules
  - sequenziell: $s_1.s_2$
    jedes Ereignis in $s_1$ geschieht vor jedem Ereignis in $s_2$
  - parallel: $s_1|s_2$
    Vereinigung der beiden Schedules ohne zusätzliche Constraints zwischen $s_1$ und $s_2$

# Beispiele für Scheduled Expressions

- Spekulative Auswertung der Argumente
  `(e0 m@e1 n@e2) sched Dm|Dn`
- Auswertung der Funktionsvorschrift vor paralleler Auswertung der Argumente
  `(l@e0 m@e1 n@e2) sched l^.(Dm|Dn)`
- Call-by-value-Auswertung
  `o@(l@e0 m@e1 n@e2) sched l.m.n.o`,
  wobei Schedule $s$ Abkürzung für `D`$s$`.`$s$`^`

# Beispiele für Mapped Expressions

- absolut: `(f x on 0) + (g y on 1)`
  `f x` wird auf Prozessor 0 ausgeführt, `g y` auf Prozessor 1
- relativ: `(f x on left self) + (g y on right self)`,
  wobei `left` und `right` Funktionen auf einer Netzwerktopologie, die einen Binärbaum darstellt.

```
left  p  = 2*p
right p  = 2*p+1
parent p = p`div`2
```

# Paralleles Divide-and-Conquer Skelett

```
dc pred basic divide combine = f
 where f x =
  if pred x
     then basic x
     else combine sl@sleft sr@sright  sched  Dsl|Dsr
           where (l,r) = divide x
                 sleft  = f l  on  left self
                 sright = f r  on right self
```

# Glasgow Parallel/Distributed Haskell

**Prozesse vs. Threads**

- Prozesse
  - eigener Adressraum
  - Kommunikation zwischen Prozessen über Mechanismen zum Nachrichtenaustausch
  - Beispiele: MPI/PVM-Bibliotheken, UNIX-Fork, Remote Method Invocation in Java, Occam
- Threads
  - gemeinsamer Adressraum
  - Kommunikation über gemeinsame Variablen
  - Konzepte zur Synchronisation: Semaphore, Monitore
  - Beispiele: C-Cilk, Java-Threads

# Glasgow Parallel Haskell

- ein(!) zusätzliches Sprachkonstrukt: `par`
- denotationelle Semantik

```
par :: a -> b -> b
par x y = y
```

- operationale Semantik
  1. Erzeugung eines Sparks, Ziel: `x` zu WHNF auszuwerten.
  2. Rückgabe des unausgewerteten Ausdrucks `y`.
  3. der Spark wird bei Gelegenheit aktiviert zu einem Thread.
  4. Lastbalancierung: Thread wandert auf anderen Prozessor.
  5. nach Auswertung wird der Graph von `x` durch den reduzierten Graphen ersetzt, sichtbar für `y`.
- Anwendung: Ausdruck `y` enthält `x` als Teilausdruck.

# Beispiel: Fibonacci-Zahlen[a]

```
pfib n
 | n <= 1 = n
 | otherwise = n1 `par` (n2 `seq` n1+n2+1)
                where
                 n1 = pfib (n-1)
                 n2 = pfib (n-2)
```

- `seq` schon Teil von Haskell98
- Berechnung von `n1` parallel zur
  1. Berechnung von `n2`
  2. Auswertung von `n1+n2+1`
- durch rekursive Parallelisierung entsteht ein Baum von Threads.

[a] Beispiel nur zur Erklärung des Prinzips, ineffiziente Mehrfachberechnungen!

# Beispiel: Quicksort

```
quicksortN []     = []
quicksortN [x]    = [x]
quicksortN (x:xs) = losort `par` hisort `par` result
   where
    losort = quicksortN [ y | y<-xs, y<x  ]
    hisort = quicksortN [ y | y<-xs, y>=x ]
    result = losort ++ (x:hisort)
```

Problem: wegen Laziness (keine Auswertung der Argumente von Konstruktoren) erzeugen die Threads für `losort` und `hisort` jeweils nur eine cons-Zelle.

# Quicksort-Beispiel verbessert

```
forceList :: [a] -> ()
forceList [] = ()
forceList (x:xs) = x `seq` forceList xs

quicksortF []     = []
quicksortF [x]    = [x]
quicksortF (x:xs) = (forceList losort) `par`
                    (forceList hisort) `par`
                    losort ++ (x:hisort)
   where
    losort = quicksortF [ y | y<-xs, y<x  ]
    hisort = quicksortF [ y | y<-xs, y>=x ]
```

# Beispiel: nachgebildete Datenparallelität

- alle Prozesse führen die gleiche Berechnung durch
- aber: Anfangszeiten zeitverschoben

```
parMap :: (a->b) -> [a] -> [b]
parMap f [] = []
parMap f (x:xs) = fx `par` fxs `seq` (fx:fxs)
                  where
                   fx  = f x
                   fxs = parMap f xs
```

# Strategien

- es würden für jeden Datentyp forcing-Funktionen benötigt.
- nicht was wir wollen: zu aufwändig, Programm schwer verständlich.
- Lösung: Trennung zwischen Funktionalität und dynamischem Verhalten.

```
type Strategy a = a -> ()

r0, rwhnf :: Strategy a
r0    _ = ()                -- keine Auswertung
rwhnf x = x `seq` ()        -- Auswertung von x zu WHNF
```

# Vererbung von Strategien

```
class NFData a where
 rnf :: Strategy a
 rnf = rwhnf          -- für Basistypen

instance NFData a => NFData [a] where
 rnf [] = ()
 rnf (x:xs) = rnf x `seq` rnf xs

instance (NFData a, NFData b) => NFData (a,b) where
 rnf (x,y) = rnf x `seq` rnf y
```

# Anwendung von Strategien

```
using :: a -> Strategy a -> a
using x s = s x `seq` x

quicksortS []  = []
quicksortS [x] = [x]
quicksortS (x:xs) = losort ++ (x:hisort) `using` strategy
     where
      losort = quicksortS [ y | y<-xs, y<x  ]
      hisort = quicksortS [ y | y<-xs, y>=x ]
      strategy result = rnf losort `par`
                        rnf hisort `par`
                        rnf result
```

# Concurrent Haskell

Benutzer kann zusätzliche I/O-Threads und die Kommunikation zwischen ihnen spezifizieren. Siehe GHC-Bibliothek `Control.Concurrent.*`.

```
forkIO      :: IO () -> IO ThreadId   -- Argument: neuer Thread
                                      -- Ergebnis: zugewiesene ID
myThreadId :: IO ThreadId             -- Abfrage der eigenen ID
```

**Synchronisation und Kommunikation**

```
newEmptyMVar :: IO (MVar a)           -- Erzeuge neues Semaphor
takeMVar     :: MVar a -> IO a        -- Lesen des Inhalts,
                                      -- Blockade solange leer
putMVar      :: MVar a -> a -> IO ()  -- Schreiben des Inhalts
isEmptyMVar  :: MVar a -> IO Bool     -- Test, ob Semaphor leer
```

# Glasgow Distributed Haskell

- verbindet die Features von GpH und Concurrent Haskell.
  - von GpH: funktional transparente Threads,
    können als eigenständige Prozesse angesehen werden.
  - von Concurrent Haskell: zusätzliche I/O-Threads,
    explizite Kommunikation und Synchronisation.
- zusätzlich sind Prozessor-IDs explizit, der Benutzer kann bestimmen, auf welchem Prozessor eine Berechnung stattfindet

# GdH Funktionen

- Abfrage der eigenen virtuellen Prozessor-ID: `myPEId :: IO PEId`
- Abfrage aller virtueller Prozessor-IDs: `allPEId :: IO [PEId]`
- Erzeugung eines Threads auf einem anderen virtuellen Prozessor
  - parallel zum erzeugenden Thread:
    `rforkIO :: IO () -> PEId -> IO ThreadId`
  - mit Suspendierung des erzeugenden Threads:
    `revalIO :: IO a -> PEId -> IO a`

# Ping-Programm in GdH

```
main = do pes <- allPEId
          putStrLn ("PEs = " ++ show pes)
          mapM loop pes
  where loop pe
   = do putStr ("Pinging "++show pe++" ... ")
        (name,ms) <- timeit (revalIO (getEnv "HOST") pe)
        putStrLn ("at "++name++" time="++show ms++"ms")
```

```
PEs = [262344,524389,786442,1048586,1310730]
Pinging 262344 ... at ushas time=0ms
Pinging 524389 ... at bartok time=3ms
Pinging 786442 ... at brahms time=3ms
Pinging 1048586 ... at selu time=2ms
Pinging 1310730 ... at kama time=2ms
```

# Eden

- entwickelt seit 1995 in Marburg (Rita Loogen) und Complutense Madrid
- Erweiterung von Concurrent Haskell
- explizite Parallelität mit Prozessen
- geeignet für Parallelrechner mit verteiltem Speicher
- Features
  - rein funktionale Beschreibung von Prozessen durch sog. *Prozess-Abstraktionen*
  - erzeugen von Prozessen durch Anwendung von Prozessabstraktionen
  - asynchrone Kommunikation zwischen Prozessen über Kommunikationskanäle, beschrieben durch Ströme

# Eden als Sprache mit Prozess-Skeletten

ein Eden-Programm besteht aus zwei Ebenen.

1. Spezifikation der Berechnung als ein Zusammenwirken von Prozessen
   - verbinden von Prozessen durch Nachrichtenkanäle (Ströme)
   - nicht referentiell transparent,
     es gibt nichtdeterministisches Mergen von Strömen
2. rein-funktionale Ebene, auf der das Ein-/Ausgabeverhalten von Prozessen spezifiziert wird

# Beispiel für Prozessabstraktion

- Prozessabstraktion: `process` *input* `->` *output*
- Typ: *input* $::\alpha$, *output* $::\beta$ $\Rightarrow$ Abstraktion `::` `Process` $\alpha$ $\beta$
- die Listen in der Eingabe und Ausgabe des Prozesses sind Ströme für die Eingangs- und Ausgangskanäle

```
merger :: Process ([a],[a]) [a]
merger = process (s1,s2) -> smerge s1 s2
          where smerge [] ys = ys
                smerge xs [] = xs
                smerge (x:xs) (y:ys)
                  = if x<=y then x : smerge xs (y:ys)
                            else y : smerge (x:xs) ys
```

# Beispiel mit Prozessinstanzierung

- Prozessinstanzierung: *Prozessabstraktion # Eingangskanäle*
- das Ergebnis ist ein Tupel von Ausgangskanälen

```
sortNet :: Process [a] [a]
sortNet = process list -> sort list
  where sort []  = []
        sort [x] = [x]
        sort xs  = merger # (sortNet # l1, sortNet # l2)
                   where (l1,l2) = unshuffle xs
        unshuffle []  = ([],[])
        unshuffle [x] = ([x],[])
        unshuffle (x:y:t) = (x:t1,y:t2)
                   where (t1,t2) = unshuffle t
```

# Das $\mathcal{HDC}$-Projekt

- am Lehrstuhl von Prof. Lengauer entwickelt (DFG-Projekt)
- Ziel: High-Level-Alternative zu FORTRAN,
  Programmierer braucht keine Kenntnisse in Parallelprogrammierung
- Fokus aus wissenschaftlichem Interesse
  - Divide-and-Conquer
  - Space-Time-Mapping (Schedule, Allokation) zur Compilezeit
- Vorarbeit: Diplomarbeit Musiol (1996), Makros in C+MPI

# Makros in C+MPI

Beispiel: Teilungsfunktion für Multiplikation von Zahlen `a` und `b`

```
DIVIDE( case 0: COPY_L; break;  /* LOW a / LOW b */
        case 1: DIV_ITER( (DIV(0).a = UNDIV(0).a,     /* LOW a /
                          DIV(0).b = UNDIV(1).b) );     HIGH b   */
                break;
        case 2: DIV_ITER( (DIV(0).a = UNDIV(1).a,     /* HIGH a /
                          DIV(0).b = UNDIV(0).b) );     LOW b    */
                break;
        case 3: COPY_R; break;  /* HIGH a / HIGH b */
      )
```

- Vorteil: sehr effizient
- Nachteile: Makros können nicht rekursiv sein, Programmierung unkomfortabel

# Idee: funktionaler Skelettansatz

- Skelett hat zwei Seiten
  - Quellprogramm: Skelett ist Funktion höherer Ordnung
  - Zielprogramm: Skelett ist paralleles imperatives Schema
- das Programm enthält Aufrufe von Skeletten, z.B. `map`
- die funktionalen Argumente (z.B. $f$ in `map` $f$) können ein Environment haben und auch selbst Skelette aufrufen
  $\Rightarrow$ dynamisch geschachtelte Parallelität
- jedem Skelett ist ein Programmgenerator zugeordnet, der eine parallele Implementierung erzeugt, z.B. in C+MPI (Message Passing Interface)
- der $\mathcal{HDC}$-Compiler entfunktionalisiert das $\mathcal{HDC}$-Programm und verbindet es mit der C+MPI-Implementierung

# Wichtige Unterschiede von $\mathcal{HDC}$ zu Haskell

- $\mathcal{HDC}$ ist strikt, damit eine *exakte* Planung möglich ist, wann auf welchem Prozessor was ausgeführt wird
  ($\rightarrow$ Space/Time-Mapping beim Polytopenmodell)
- $\mathcal{HDC}$ hat keine Benutzer-definierbaren Typklassen und Instanzdeklarationen
- $\mathcal{HDC}$ hat ein eingeschränktes Typsystem (ähnlich ML),
  sonst wäre Entfunktionalisierung nicht möglich
- $\mathcal{HDC}$ hat kein Modulsystem
- Listen werden in $\mathcal{HDC}$ durch Arrays implementiert

# Wichtige Unterschiede von $\mathcal{HDC}$ zu GpH

- viele Skelette anstelle des einzigen Konstrukts `par`;
  nicht so elegant aber mehr Kontrolle über Parallelität
- vorgeplantes Ablaufschema anstelle dynamischem Task-Management
- Referenzzählung als Speicherbereinigungsmechanismus
  (Skelettimplementierungen können in die Speicherverwaltung eingreifen)
- getrennte Heaps auf den Prozessoren anstelle eines verteilten Heaps

# Wichtige Unterschiede von $\mathcal{HDC}$ zu Eden

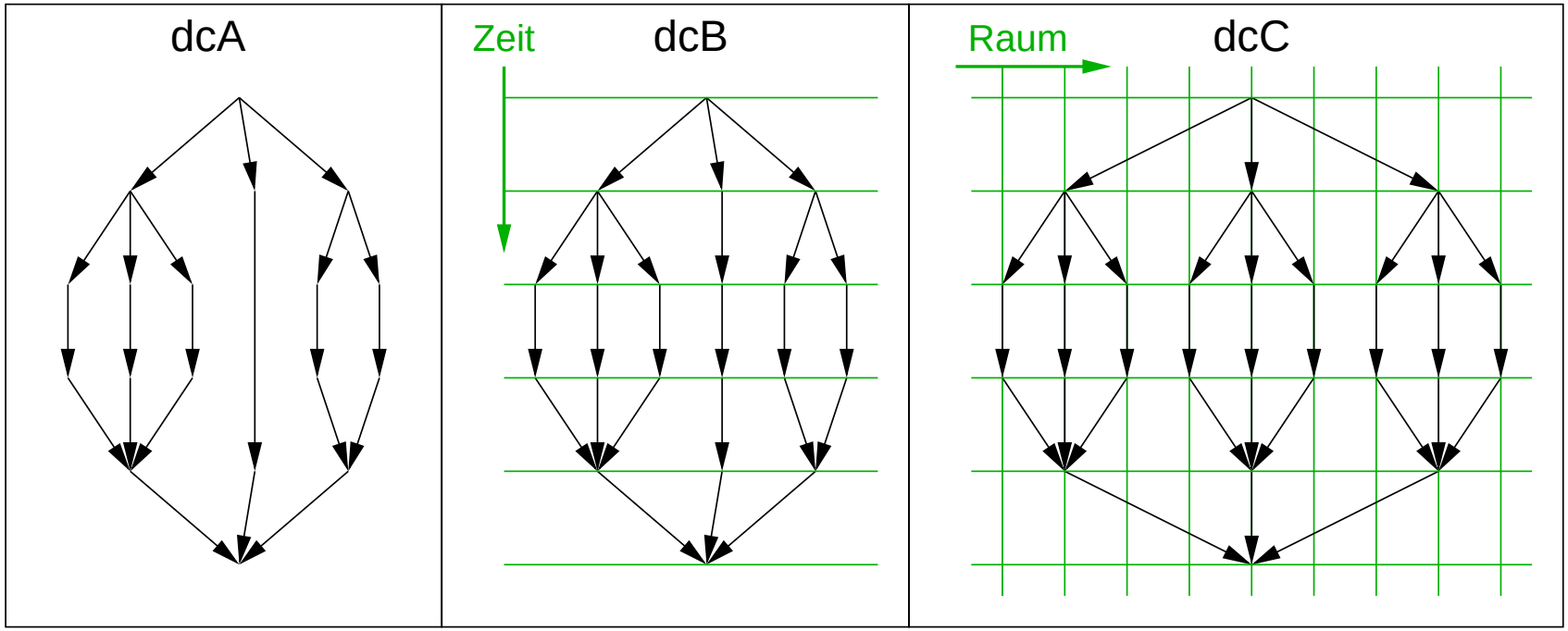
- keine Beschränkung auf eine einzige Koordinationsebene, beliebige Programmteile können Skelette aufrufen
- $\mathcal{HDC}$ erlaubt kein Pipelining mit Strömen
- Skelette sind Funktionen wie andere auch, man kann
  1. ein *Skelett* zunächst im Benutzerprogramm definieren,
  2. später zur Parallelisierung in C+MPI schreiben, ohne die Programme, die dieses Skelett benutzen, ändern zu müssen

# Anwendungsbeispiel: Funktionales *Quicksort*

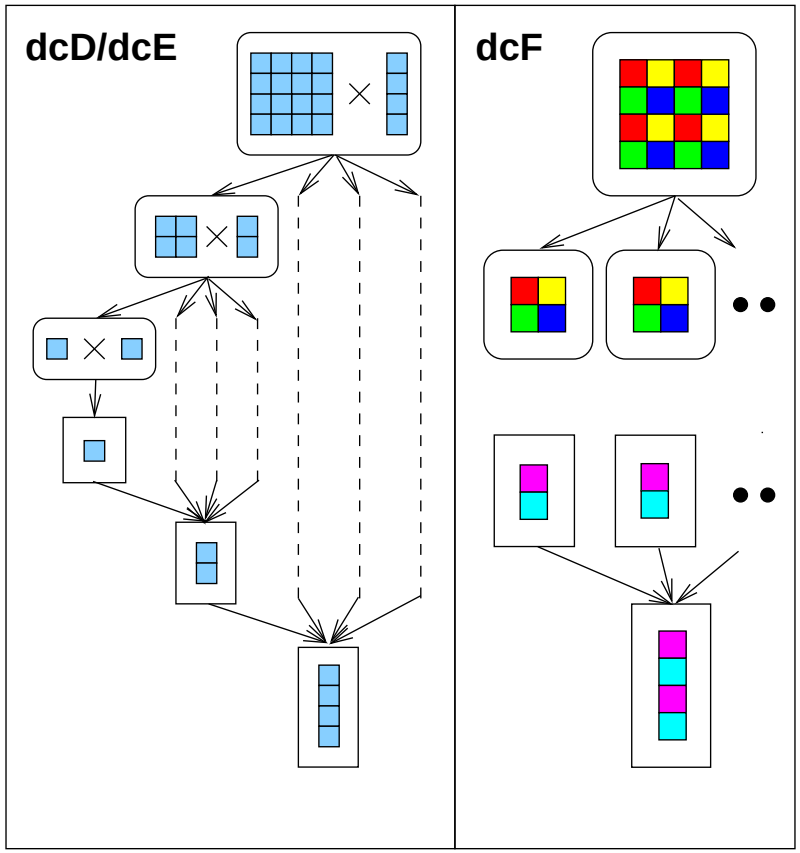
```
qs :: [Int] -> [Int]
qs ys = let p xs = length xs < 2
            b xs = xs
            pivot xs = xs!!0
            d xs = [filter (< pivot xs) xs,
                    filter (> pivot xs) xs]
            c xs [ys,zs]
             = ys ++ (filter (== pivot xs) xs) ++ zs
        in dcA p b d c ys -- Anwendung des DC-Skeletts

parmain :: IO Unit
parmain = get >>= \ x ->
          put (qs (x::[Int]))
```

# Klassifikation von $\mathcal{DC}$

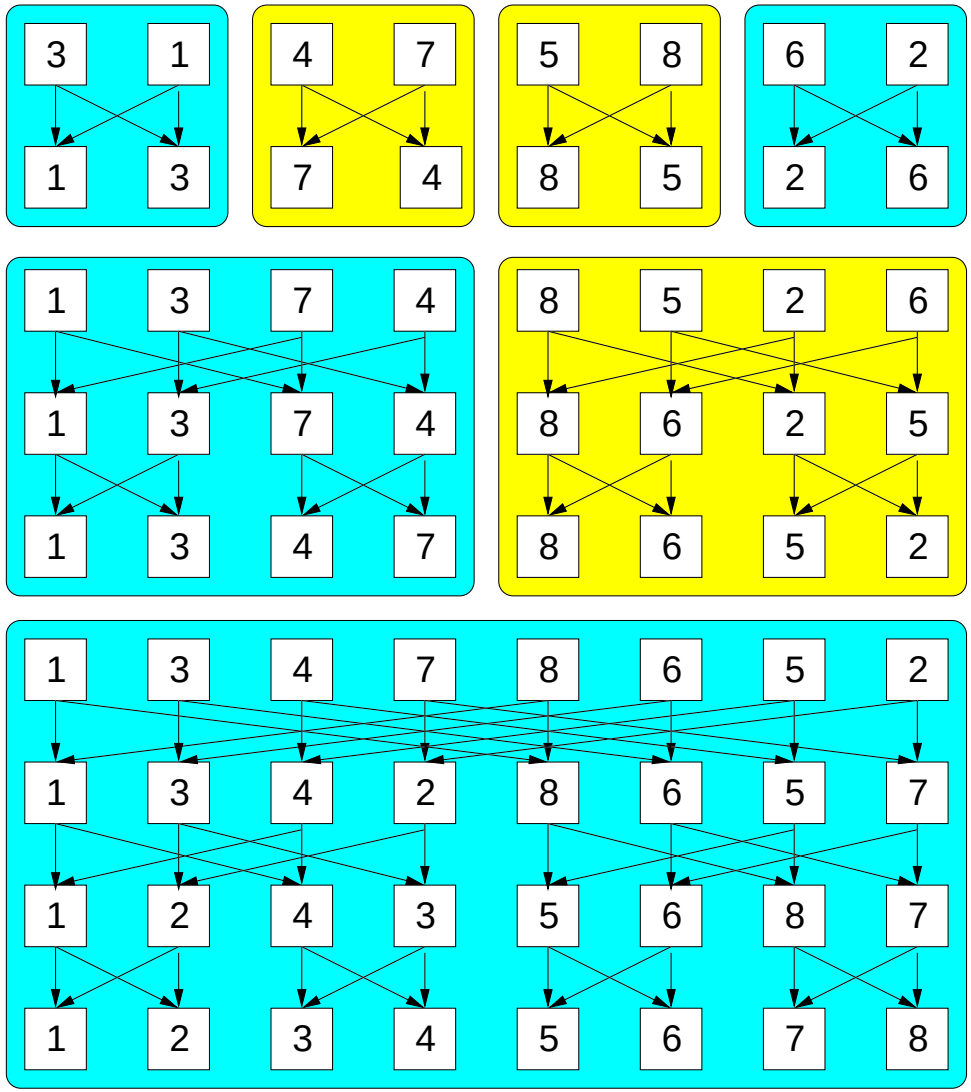



| Skelett | Einschränkung | Anwendung |
|---|---|---|
| `dcA` | — | Quicksort,<br>Maximum Independent Set |
| `dcB` | feste Rekursionstiefe | $n$–Damen Problem |
| `dcC` | fester Teilungsgrad $k$ | Karatsuba Multiplikation ($k$=3) |




| | | |
|---|---|---|
| `dcD` | Blockrekursion | Inversion Dreiecksmatrix ($k$=2) |
| `dcE` | elementweise Operationen | Matrix/Vektor–Multiplikation ($k$=4) |
| `dcF` | Kommunikation korrespondierender Elemente | Schnelle Fourier–Transformation ($k$=2), Bitonisches Mischen ($k$=2), Polynommultiplikation ($k$=3), Matrixmultiplikation ($k$=7) |

# Batcher Sort: Bitonic Merge (`dcF`) in `dcD`

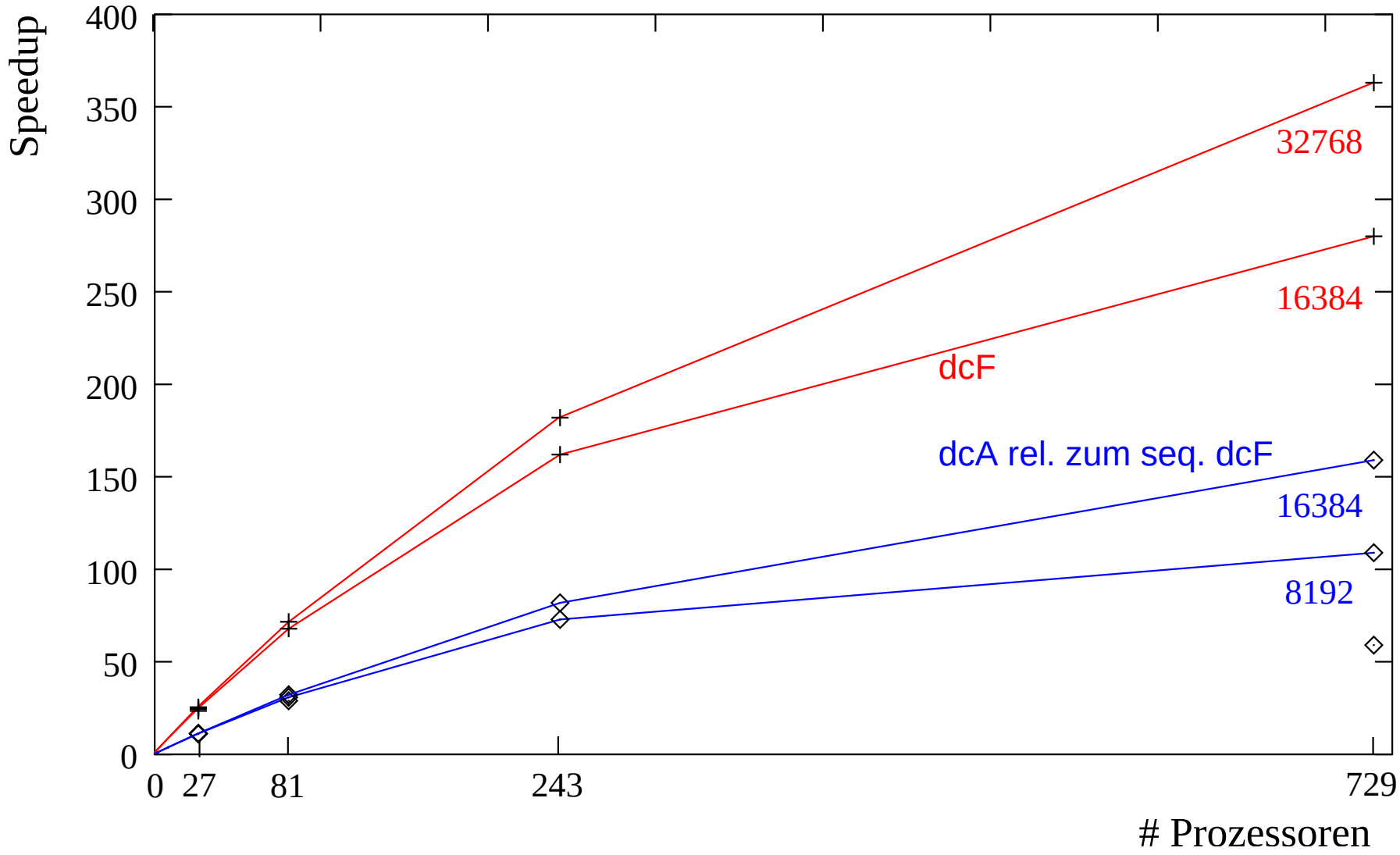


# Batcher Sort: dcF als Argument von dcD

```
batcherSort :: [Int] -> [Int]
batcherSort xs =
 dcD 2 [2] [2] basic divide combine n [xs] !! 0 where
   n = ilog2 (length xs)
   basic x = x
   divide _ [x] = [[left x], [right x]]
   combine lev _ [[x],[y]]
    = let sub = dcF 2 2 2 b d c lev (x ++ reverse y)
      in  [[left sub,right sub]]
            where b x = x
                  d s [x,y] = if s==0 then min x y
                                      else max x y
                  c s l = l!!s
```

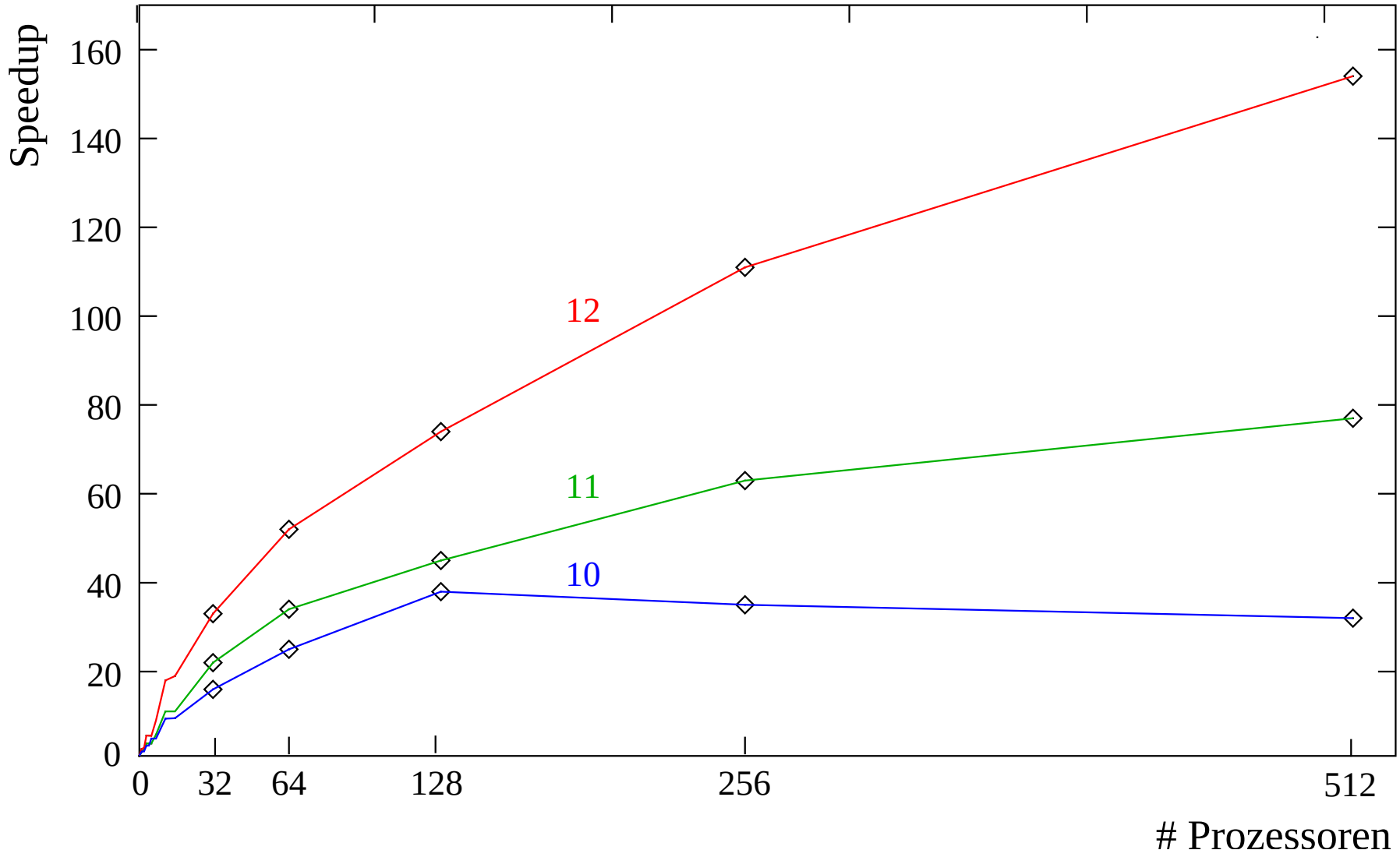
# Performance Karatsuba-Polynomprodukt

auf Parsytec-GCel-1024 in Paderborn



# Performance N-Queens

auf Parsytec-GCel-1024 in Paderborn



# Metaprogrammierung von Parallelitäts-Skeletten

Motivation:

- eigener vollständiger Compiler zu Personal-aufwändig
- möglichst schnelle Verfügbarkeit eines Prototyps
- wenig Wartungsaufwand
- Wiederverwendung existierender Compilertechnologie
- ohne viel Aufwand profitieren von Weiterentwicklungen anderer
- leichte Austauschbarkeit von MetaOCaml oder von MPI (Message Passing Interface)

Preis:

- keine Kontrolle über Details der Repräsentation von Daten
- Verlust der referenziellen Transparenz

# Ablauf der parallelen Metaprogrammierung

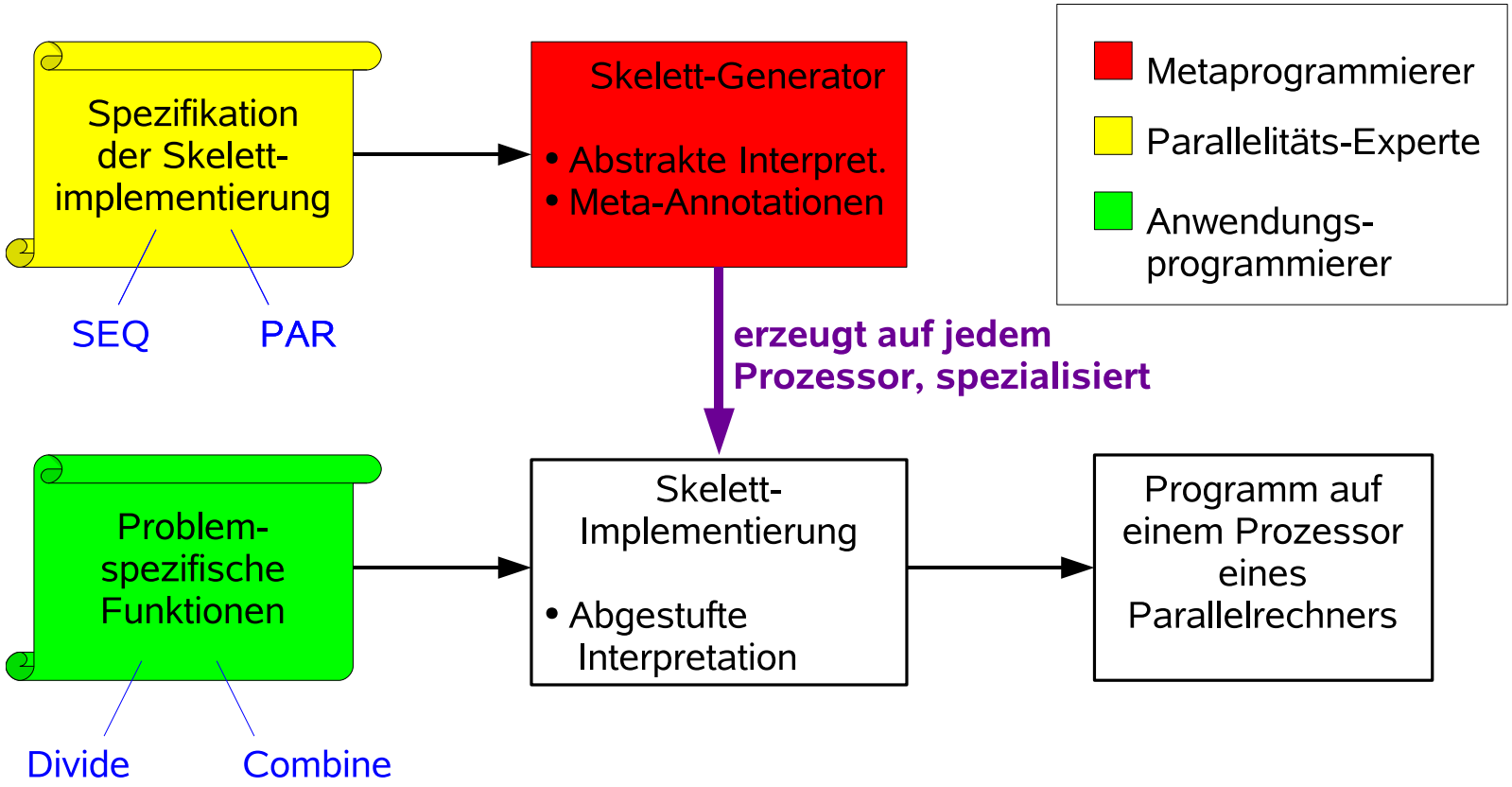
in jedem Prozess / auf jedem Prozessor

1. MetaOCaml-Programm wird durch MPI gestartet
2. Analyse (abstrakte Interpretation) der Spezifikation mit
   - Zuordnung von Prozessidentifier zu Zweig im Call-Tree
   - Berechnung und Einsetzung von MPI-Send/Receive-Aufrufen
3. Programmspezialisierung zur Laufzeit, individuell in jedem Prozess
4. Anwendung des Run-Operators (`.!`) auf das spezialisierte, gestufte MetaOCaml-Program
   - byte-code-Installation: schnelle Codegenerierung, mäßig schneller Code
   - native-code-Installation: mäßig schnelle Codegenerierung, schneller Code
5. (mehrfacher) Start der parallelen Berechung des erzeugten Programms

# Automatische Skelettgenerierung

- Skelett: trennt Implementierung der Struktur von problemspezifischen Funktionen
- Implementierung kann man weiter aufteilen in
  - Parallelitätsaspekte (wo findet was statt, wer kommuniziert wann mit wem?)
  - Metaprogrammieraspekte (Codeerzeugung für sequenzielle/parallele Kompositionen, Kommunikationen)
- Schnittstelle: Spezifikationssprache (algebraischer Datentyp)
  - Atom: sequenzielle Programmteile
  - Seq/Par: sequenzielle/parallele Komposition paralleler Berechnungen
  - Comm: globales Kommunikationsschema

# Generierung von Skeletten für Parallelität



# Spezifikation eines Divide-and-Conquer-Skeletts

```
let rec dc degree basic divide combine depth  =
  if depth=0
  then Atom (fun x -> (.< let (orig,y)=.~x in (orig,basic y) >.))
  else Par (degree, fun mypart ->
        cseq [ Atom (fun x -> .< begin  let (orig,y) = .~x in
                                        buffers.(depth) <- y;
                                        (orig,y)                end >.);
               Comm (degree-1, (fun i -> {source=0; sindex=depth; dest=i+1; dindex=0; ctag=depth }),
                               .<buffers>.);
               Atom (fun x -> .< let q = .~x in
                                 let (orig,y) = if mypart=0 then q else ([],buffers.(0)) in
                                 (y::orig, divide mypart y) >.);
               dc degree basic divide combine (depth-1);
               Atom (fun x -> .< let q = .~x in buffers.(0) <- snd q; q >.);
               Comm (degree-1, (fun i -> {source=i+1; sindex=0; dest=0; dindex=i+1; ctag=depth }),
                               .<buffers>.);
               Atom (fun x -> .< let q = .~x in
                                 if mypart>0 then q
                                             else let (inp::orig,y) = q in
                                                  let res = combine (inp,buffers) in
                                                  (orig,res)
                               >.)
             ])
```