Skeletons

- A number of patterns recur frequently in the body of parallel algorithms.

  These patterns are composed of

  – computations and

  – the interactions between them.

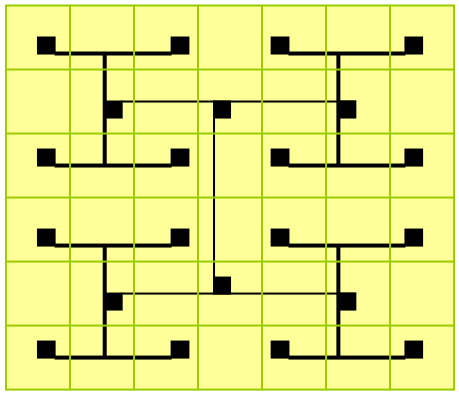
  The patterns can be conceptually abstracted from the details of the activities they control.
- This leads to (algorithmic) skeletons which are
  - higher-order functions
  - with an associated parallel evaluation strategy and
  - a cost (performance) model to estimate the execution time
- Campbell's classification:
  – divide and conquer (recursively partitioned)
  – task queue (work pool, master worker)
  – systolic (pipeline)

Divide and Conquer

- **higher order function:**

```
d&c  :: (a-> Bool) -> (a->b) ->   (a -> [a]) ->   ([b] -> b) ->   a -> b
d&c     trivial          solve       divide          conquer       p
   =  if (trivial p) then solve p
      else conquer (map (d&c trivial solve divide conquer) (divide p))
```

- **parallel implementations:**
  - **idealised implementation on tree of processors**
  - **H-tree implementation of binary d&c on a grid:**



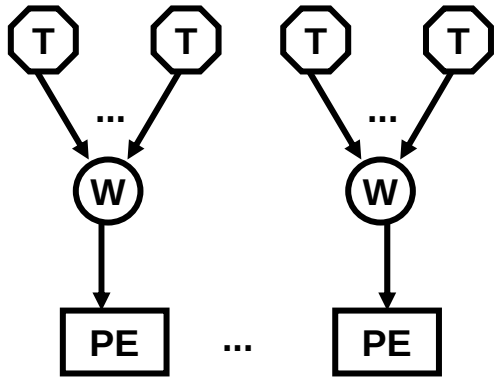
Task Queue - Farm - Master/Worker

- **higher order function:**

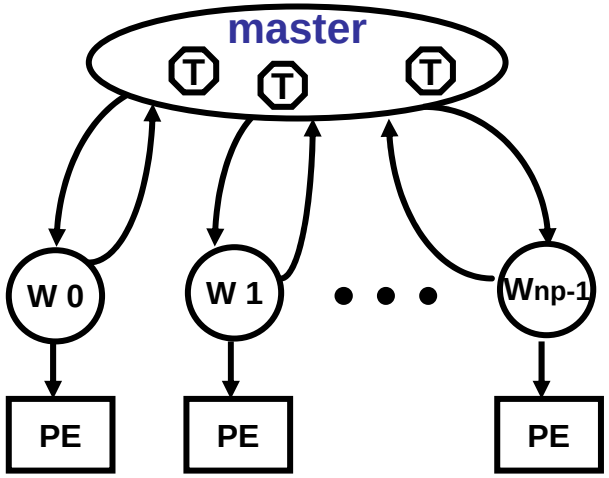
  ```
  farm          :: (a -> b -> c) -> a -> [b] -> [c]
  farm f env    = map (f env)
  ```

- **implementation:**

  – **static task distribution:**



  – **dynamic task distribution:**



- **cost model** (static distr.): $t_{farm} = t_{setup} + n/p\ (t_{solve} + 2t_{comm})$

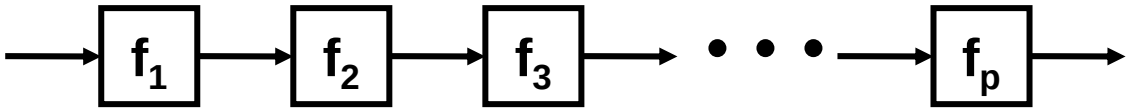
Systolic Scheme - Pipelining

- **higher order function:**

```
pipe    :: [ [a] -> [a] ]  -> [a] -> [a]
pipe    = foldr (.)  id
```

- **implementation:**

  linear pipeline of processes



- **cost model**: $t_{pipe} = t_{setup} + (t_{stage} + t_{comm})(p + n-1)$

Skeletal Programming

- **fixed number of higher-order skeletons (algorithmic skeletons)**
- **different highly optimized implementations for different target architectures (architectural skeletons)**

- **Programming Methodology:**
  - **choose suitable skeletons**
  - **compose them in a program**
  - **estimate its expected performance => cost model**
  - **make changes in design, if necessary => transformation rules**

History

- **origin:**

  PhD thesis of M. Cole (Univ. of Edinburgh, 1988)

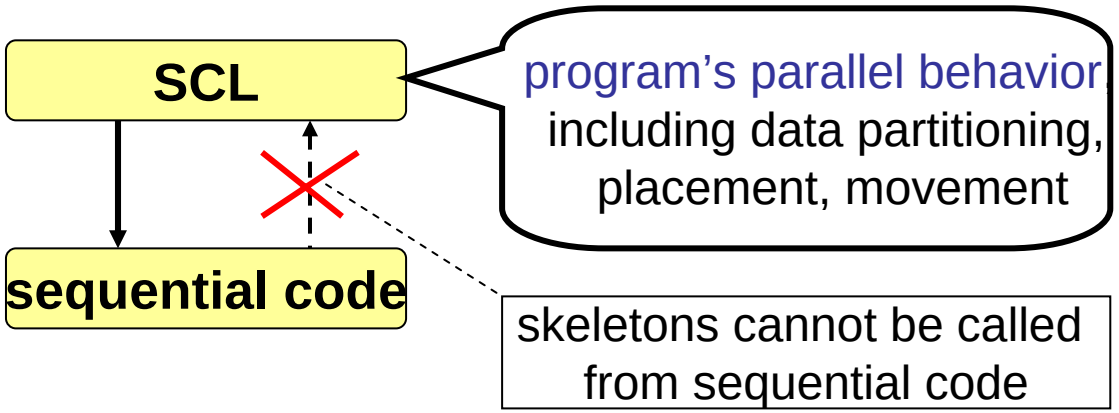
  few complex skeletons:

  - **FDDC (fixed degree divide & conquer)**
  - **IC (iterative combination)**
  - **TQ (task queue)**

- **first systems:**
  - **P^3L Pisa Parallel Programming Language [Pelagatti et al. 1995] based on imperative computation language**
  - **SCL Structured Coordination Language [Darlington et al. 1995] based on functional computation language**

SCL - Structured Coordination Language

layered skeletal approach:



- **provides lower level of detailed control through explicitly distributed arrays ParArray with skeletons:**

```
partition :: Partition_pattern -> SeqArray index a
                                  -> ParArray index (SeqArray index a)
gather    :: Partition_pattern -> ParArray index (SeqArray index a)
                                  -> SeqArray index a
align     :: ParArray index a -> ParArray index b -> ParArray index (a,b)
```

- **higher level skeletons:**
  - **elementary skeletons (data parallel operations over distributed arrays)**
  - **computational skeletons (parallel control flow)**
  - **communication skeletons**

SCL Skeletons

- **elementary**
  - map :: (a -> b) -> ParArray index a -> ParArray index b
  - imap :: (index -> a -> b) -> ParArray index a -> ParArray index b
  - fold :: (a -> a -> a) -> ParArray index a -> a
  - scan :: (a -> a -> a) -> ParArray index a -> ParArray index a

- **computational**
  - farm :: (a -> b -> c) -> a -> ParArray index b -> ParArray index c

    farm f e = map (f e)

  - spmd :: [(ParArray index a -> ParArray index a, index -> a -> a)] ->
    ParArray index a -> ParArray index a

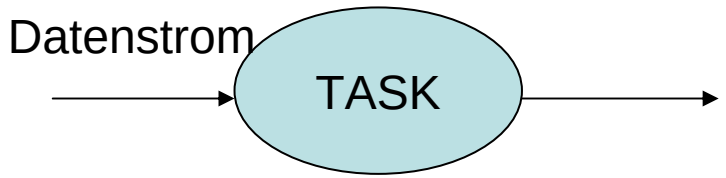
    spmd [ ] = id

    spmd ((gf, lf) : fs) = (spmd fs) . gf . (imap lf)
  - ...

- **communication**
  - rotate :: Int -> ParArray index a -> ParArray index a
  - ...

Standardisierung
– Skelettbibliotheken für MPI –

- **H. Kuchen (Univ. Münster, 2002)**
  - **Skelette als C++ Bibliothek auf der Basis von MPI**
    - **Polymorphie über Templates simulieren**
    - **Funktionen höherer Ordnung mittels überladener Operatoren**
  - **Unterscheidung**
    - **datenparallele Skelette**
      - **Manipulation verteilter Datenstrukturen**
      - **Berechnungsskelette: map, fold, zip, scan ...**
      - **Kommunikationsskelette: rotate, broadcast, permute, gather ...**
    - **kontrollparallele Skelette**
      - **pipe, farm, d&c, search**
      1. **Erzeuge Prozesstopologie**
      2. **starte Tasks (parallele Prozesse)**



Standardisierung
– Skelettbibliotheken für MPI II –

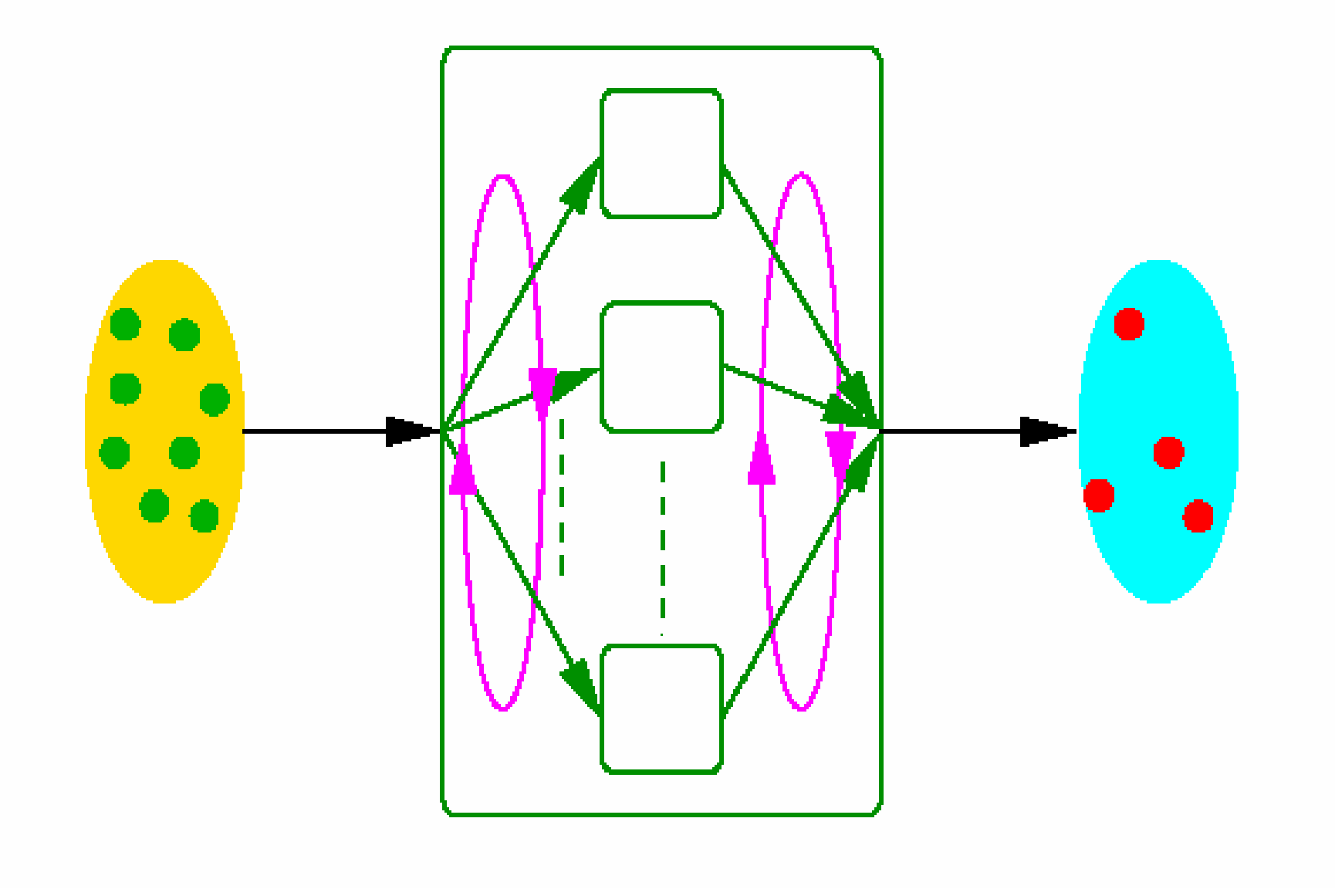
- **M. Cole, A. Benoit (Univ. of Edinburgh, 2004)**
  - **eSkel – Edinburgh Skeleton Library**

Ziel: Erweiterung von kollektiven Operationen (einfache Skelette) in MPI um algorithmische Skelette

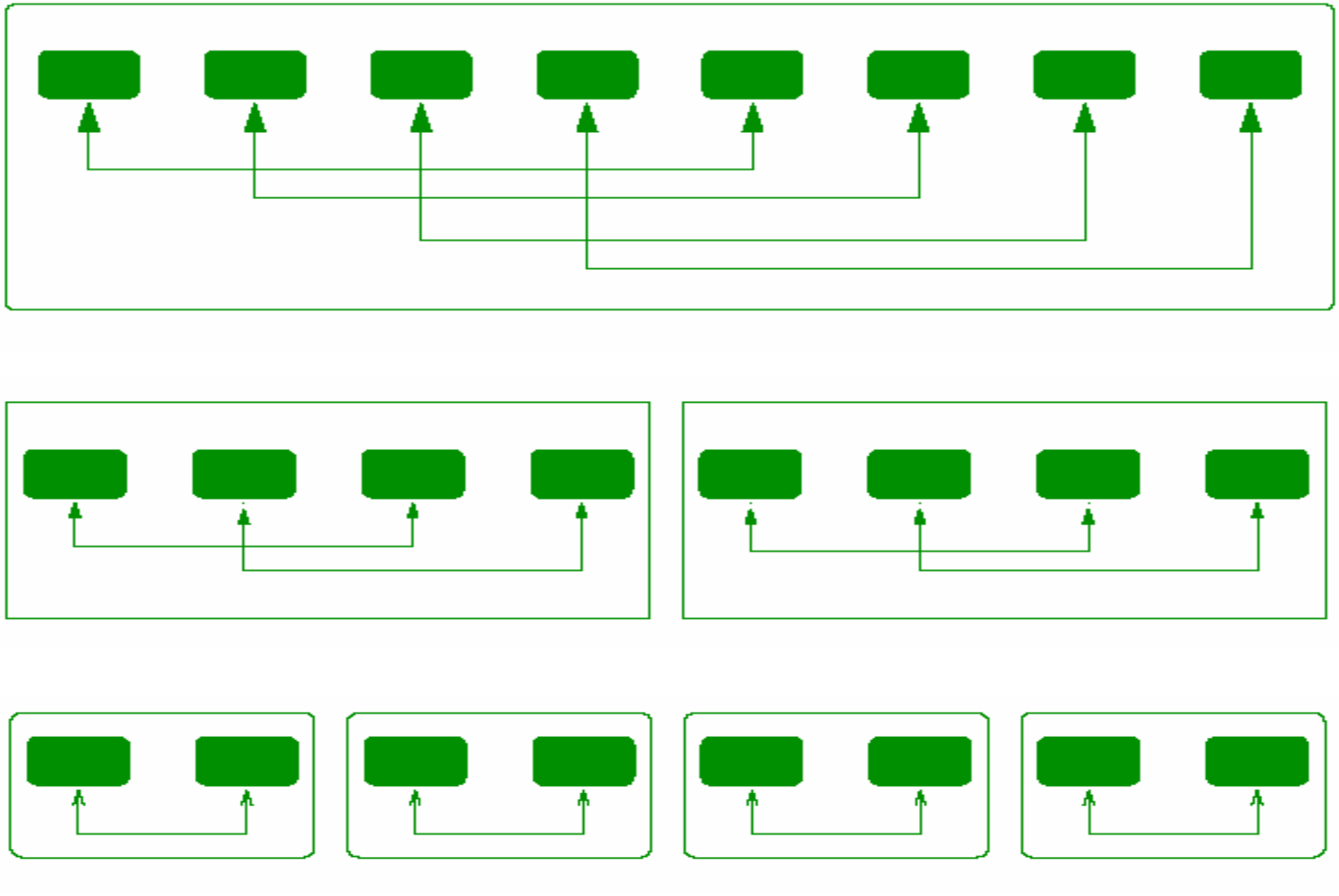
Zur Zeit: 5 Skelette

1. **pipeline**
2. **farm**
3. **deal (wie farm, ohne farmer mit zyklischer Taskverteilung)**
4. **haloswap -> iterative Approximation**
   - **Schleife mit**
     - **local update**
     - **check for termination**
5. **butterfly -> divide & conquer in hypercube**

Deal in Pipeline



Butterfly



Pipeline Skelett

```
void Pipeline (int ns, Amode_t amode[],
                    eSkel_molecule_t * (*stages[])(eSkel_molecule_t ), int col,
                    Dmode t dmode, spread t spr[], MPI_Datatype ty[],
                    void  *in, int inlen, int inmul, void  *out, int outlen,
                    int  *outmul, int outbuffsz, MPI_Comm comm)
```

=> 15 Parameter:

- **3 Parameter für Pipeline Eingaben**
- **4 Parameter für Pipeline Ausgaben**
- **3 Parameter für Pipeline-Stufen-Spezifikation**
- **4 Parameter für Schnittstellen & Modi**
- **1 Parameter für Kommunikator**

Transformation of Skeletal Programs

- **Problem: composition of skeletons**
  - **try to predict impact on performance -> cost model required**
  - **develop transformation rules**

- **[Gorlatch, Lengauer 1997]:**
  (De)Composition Rules for Parallel Scan and Reduction,
  Workshop on Massively Parallel Programming Models
  - **Expressiveness:**
    How much ground does the class of (almost-)homomorphisms cover?
  - **Implementation:**
    How can the homomorphism skeleton be implemented efficiently on parallel computers?
  - **Composition:**
    Are certain compositions of standard homomorphisms good candidates for new homomorphic skeletons? How can these be optimized further?
  - **Decomposition:**
    Can a more complex homomorphism be decomposed into simpler homomorphisms, with the result of improved performance?
  - **Performance:**
    How portable are skeleton implementations?

Homomorphisms

- **Functions are defined on non-empty finite lists, with list concatenation ++ as constructor.**

- **Definition: Function h on lists is a homomorphism iff there exists a binary operator $\otimes$ such that, for all lists xs and ys:**

  $$h\ (xs\ ++\ ys) = h\ xs \otimes h\ ys$$

- **$\otimes$ is necessarily associative, because ++ is associative.**

- **Examples**:
  - **Mapping:** map f $[x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n]$
  - **Reduction:** red $(\oplus)$ $[x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \ldots \oplus x_n$
  - **Scanning:**

    scan $(\oplus)[x_1, x_2, \ldots, x_n] = [x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \ldots \oplus x_n]$

    with associative operator $\oplus$

Properties of Homomorphisms

- **Normal form:** **Function h is a homomorphism iff it can be factored into the composition**

  $$h = red\ (\otimes)\ \circ\ map\ f \qquad with \qquad f\ a = h\ [a]$$

  for every element a and $\otimes$ is from the definition of homomorphisms.

- **Promotion property:** $h \circ red\ (++) = red\ (\otimes) \circ map\ h$

- **Transformations: Functional programs consisting of homomorphisms can be transformed using semantics-preserving equational rules, e.g.**

  $$prog = h_n \circ h_{n-1} \circ .... \circ h_1$$

  function composition --> synchronisation (avoidable?)

Scan-Reduce Composition

Let $\text{scanred}\ (\otimes,\oplus) := \text{red}\ (\oplus) \circ \text{scan}\ (\otimes)$.

We assume that $\otimes$ **distributes over** $\oplus$: $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$.

An almost-homomorphism is a function that becomes a homomorphism when tupled with one or more auxiliary functions.

When tupled with function red scanred becomes the homomorphism scanred':

$$\text{scanred'}\ (\otimes,\oplus)\ x := (\text{scanred}\ (\otimes,\oplus)\ x\ ,\ \text{red}\ (\otimes)\ x)$$

For arbitrary binary, associative operators $\otimes$ **and** $\oplus$**, such that** $\otimes$ **distributes over** $\oplus$:

$$\text{red}\ (\oplus) \circ \text{scan}\ (\otimes) = \pi_1 \circ \text{red}\ (\langle\oplus,\otimes\rangle) \circ \text{map pair}$$

where $\text{pair}\ x := (x,x)$

and $(s_1,r_1)\ \langle\oplus,\otimes\rangle\ (s_2,r_2) := (s_1 \oplus (r_1 \otimes s_2),\ r_1 \otimes r_2)$.

Scan - Scan - Composition

For arbitrary binary, associative operators $\otimes$ and $\oplus$, where $\otimes$ distributes over $\oplus$,

$$\text{scan}\,(\oplus) \circ \text{scan}\,(\otimes) = \text{map}\,\pi_1 \circ \text{scan}\,(\langle\oplus,\otimes\rangle) \circ \text{map pair}$$

This result can be derived using the auxiliary function

$$\text{inits}\,[x_1, x_2, \ldots, x_n] = [\,[x_1], [x_1, x_2], \ldots, [x_1, x_2, \ldots, x_n]\,]$$

and the following identities

$$\begin{aligned}
\text{map}\,(f \circ g) &= \text{map}\,f \circ \text{map}\,g \\
\text{scan}\,(\otimes) &= \text{map}\,(\text{red}\,(\otimes)) \circ \text{inits} \\
\text{inits} \circ \text{map}\,f &= \text{map}\,(\text{map}\,f) \circ \text{inits} \\
\text{inits} \circ \text{scan}\,(\otimes) &= \text{map}\,(\text{scan}\,(\otimes)) \circ \text{inits}
\end{aligned}$$

Hypercube Implementation

- **Properties:**
  - **An elementary operation takes one unit of computation time.**
  - **Communication links are bidirectional:**

    Two neighboring processors can send messages of size m to each other simultaneously in time

    $$t_s + m\, t_w,$$

    where t_s is the start-up time and t_w is the per-word transfer time.
  - **A processor is allowed to send/receive messages on only one of its links at a time.**
  - **The computation time it takes to split or concatenate lists within a processor is ignored.**

- **Time for scan on hypercube using powerlist implementation:**

  $$\log p\ (t_s + m\,(t_w + 2))$$

  where p is the number of processor elements and
  m is the number of list elements per processor element.
- **Time for red:** $\log p\ (t_s + m\,(t_w + 1))$

Performance Evaluation

- **Scan-Scan Performance:**

time ignored

$$\underbrace{\text{scan } (\oplus) \circ \text{scan } (\otimes)}_{2 \log p \ (t_s + m\,(t_w + 2))} = \underbrace{\text{map } \pi_1 \circ \text{scan } (\langle \oplus, \otimes \rangle) \circ \text{map pair}}_{\log p \ (t_s + 2\,m\,(t_w + 3))}$$

$$2 \log p \ (t_s + m\,(t_w + 2)) > \log p \ (t_s + 2\,m\,(t_w + 3))$$

$$\text{if} \quad t_s > 2m$$

- **Scan-Red Performance:**

$$\underbrace{\text{red } (\oplus) \circ \text{scan } (\otimes)}_{\log p \ (2\,t_s + m\,(2\,t_w + 3))} = \underbrace{\pi_1 \circ \text{red } (\langle \oplus, \otimes \rangle) \circ \text{map pair}}_{\log p \ (t_s + m\,(2\,t_w + 3))}$$

$$\log p \ (2\,t_s + m\,(2\,t_w + 3)) > \log p \ (t_s + m\,(2\,t_w + 3))$$

!