Algorithmen / Algorithms II

Peter Sanders

Exercise:

Daniel Seemaier, Tobias Heuer

Institute of Theoretical Informatics

Web:

`http://algo2.iti.kit.edu/AlgorithmenII_WS20.php`

10 Parallel Algorithms

Appetizer.

More in the lecture with the same name

as well as in the Vertiefungsfach “parallel processing”

(Parallelverarbeitung)

Why Parallel Processing

Speedup: p machines processing a problem in parallel may solve it faster by a factor of up to p. However, too many cooks spoil the broth $\rightsquigarrow$ algorithms need good coordination

Power saving: Two processors with half the clock rate need less energy than one processor on its limit.
(Power $\approx$ voltage $\cdot$ clock rate)

Memory limitation of single processors

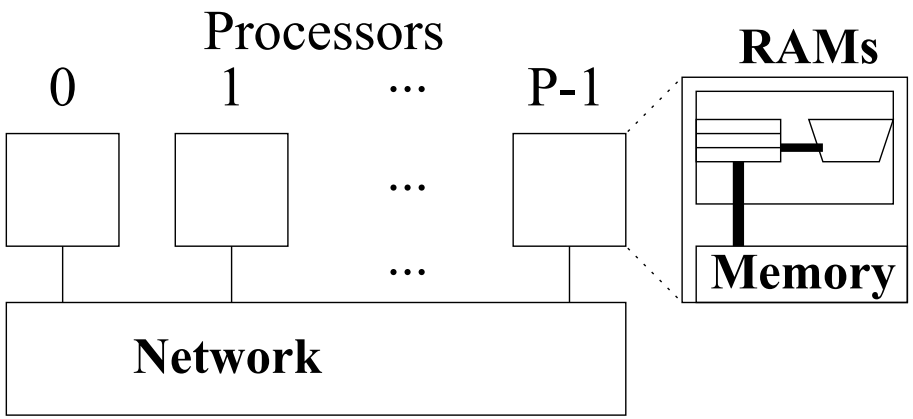
Reduction in communication: If data is distributed it can also be processed using the same distribution

Parallel Processing at the ITI Sanders

- Massively parallel sorting, Michael Axtmann
- Massively parallel graph algorithms, Sebastian Lamm
- Big-Data framework Thrill, Timo Bingmann
- Shared memory data structures, Tobias Maier & Marvin Williams
- (Hyper)Graph partitioning, Tobias Heuer & Daniel Seemaier
- Communication efficient algorithms, Lorenz Hübschle-Schneider
- SAT-Solving and planning problems, Dominik Schreiber
- Geometric algorithms, Daniel Funke

10.1 Model

Distributed-Memory Parallel Computer



- Processors are RAMs
- Asynchronous parallel processing
- Message passing for interaction

Cost Model for Message Exchanges

Each PE can

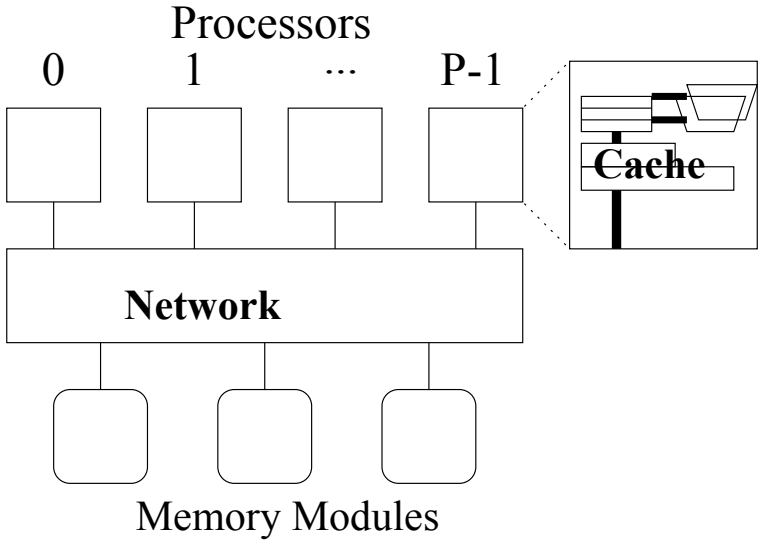
simultaneously send and receive at most one message.

Time for a message exchange of size ℓ

$$T_{\text{comm}}(\ell) = \alpha + \ell\beta$$

- Full-duplex
- Point-to-point
- Fully connected topology
- In general $\alpha \gg \beta$ – Alternative: blockwise communication
  analogous to the external memory model

(Shared Memory) Multicore-Model – Why not



- Unclear how to obtain scalable parallelism
- Unclear cost measure for memory access conflicts
- Good strategy for parallel programming: Develop distributed, implement shared memory

⇝ distributed memory model covers entire spectrum from multi-core smartphones to supercomputers and we are still close to cloud, sensor, and peer-to-peer networks

Describing Parallel Algorithms

Same pseudocode as always.

Single Program Multiple Data principle.

Use processor index to break symmetry.

Procedure helloWorldParallel

writeLineAtomic "Hallo, I am PE " iProc " out of " p "processing elements"

```
Hallo, I am PE 0 out of 3 processing elements
Hallo, I am PE 2 out of 3 processing elements
Hallo, I am PE 1 out of 3 processing elements
```

Analysis of Parallel Algorithms

Essentially only one additional parameter: p.

Calculate running time $T(I,p)$.

Problem: Interpretation.

Work: $W = pT(p)$ is a cost measure.

Span: $T_\infty = \inf_p T(p)$ a measure for scalability

(Absolute) Speedup: $S = T_{\text{seq}}/T(p)$.

Use best known sequential algorithms.

Relative speedup $S_{\text{rel}} = T(1)/T(p)$ is usually different!

Efficiency: $E = S/p$. Target: $E \approx 1$ or at least $E = \Theta(1)$.

(meaningful cost measure?)

"Superlinear speedup": $E > 1$. (possible?).

10.2 Example: Associative Operations (=Reduction)

Theorem 1. *Let $\oplus$ be an associative operator that can be calculated in constant time. Then,*

$$\bigoplus_{i<p} x_i := (\cdots((x_0 \oplus x_1) \oplus x_2) \oplus \cdots \oplus x_{p-1})$$

can be computed in time $\mathrm{O}(\log p)$

Examples: $+$, $\cdot$, max, min, ... (e.g. ? non-commutative?)

Basic concept for $p = 2^k$ (w.l.o.g.?)

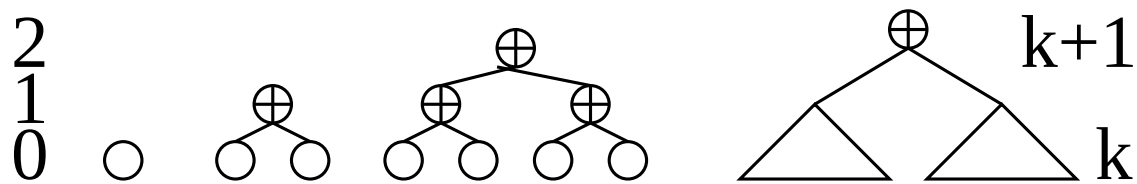
Induction over k:

$k = 0$: trivial

$k \rightsquigarrow k+1$:

$$\bigoplus_{i<2^{k+1}} x_i = \underbrace{\overbrace{\bigoplus_{i<2^k} x_i}^{\text{Depth } k} \oplus \overbrace{\bigoplus_{i<2^k} x_{i+2^k}}^{\text{Depth } k \text{ (IH)}}}_{\text{Depth } k+1}$$

■



Pseudocode

PE index $i \in \{0, \ldots, p-1\}$

// Input x_i located on PE i

active := 1

$s := x_i$

for $0 \leq k < \lceil \log p \rceil$ **do**

  if active **then**

    if bit k of i **then**

      sync-send s to PE $i - 2^k$

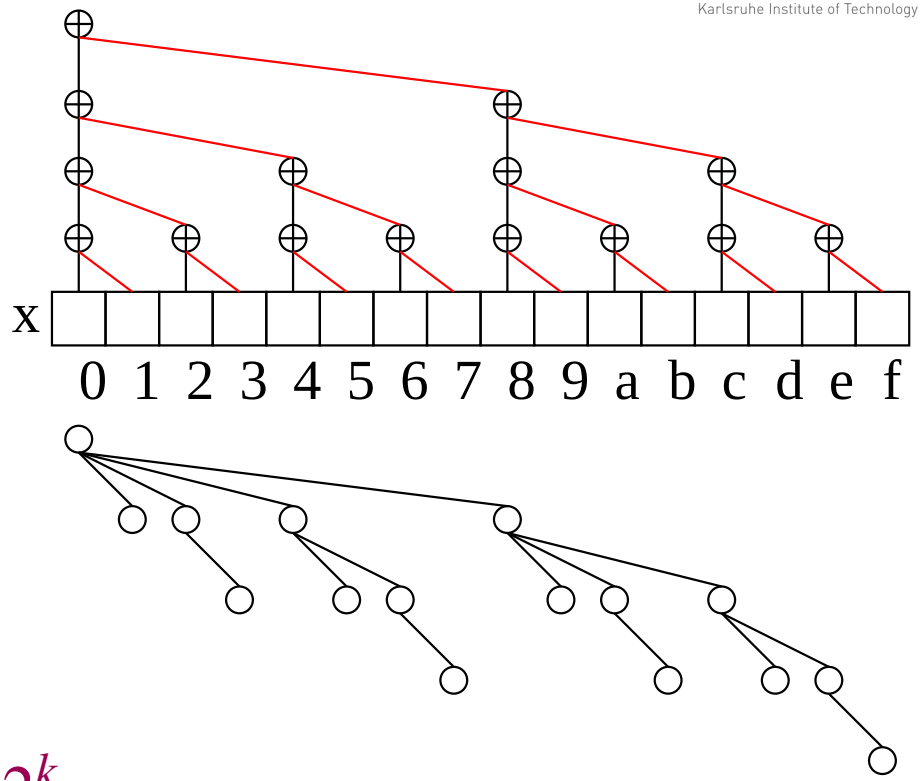
      active := 0

    else if $i + 2^k < p$ **then**

      receive s' from PE $i + 2^k$

      $s := s \oplus s'$

// result is in s on PE 0



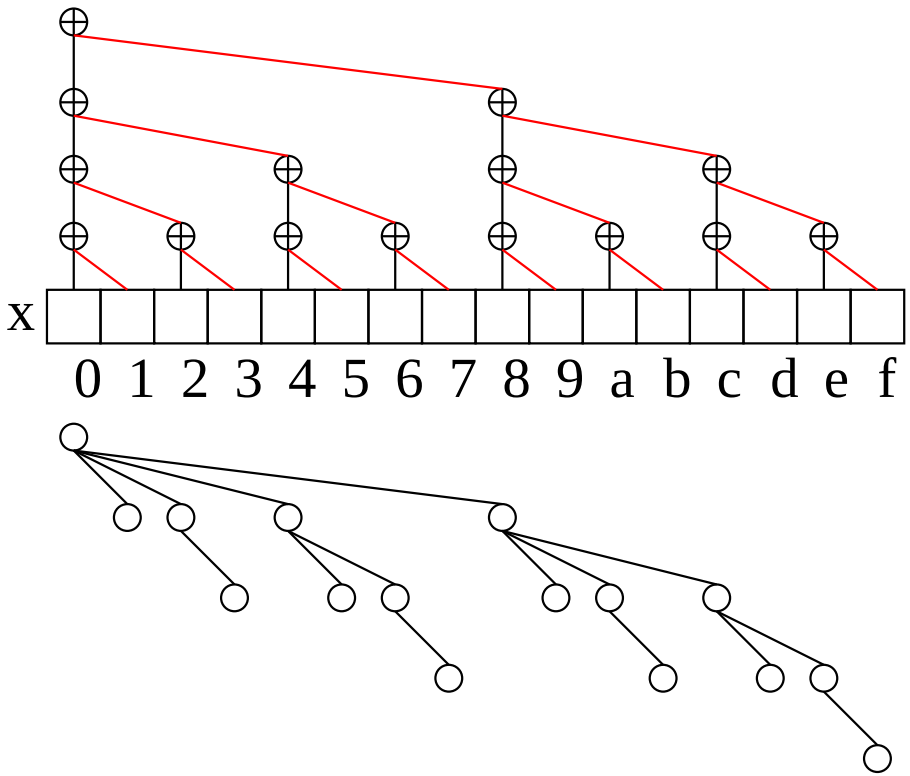
Analysis

n PEs

Time $\mathrm{O}(\log n)$

Speedup $\mathrm{O}(n/\log n)$

Efficiency $\mathrm{O}(1/\log n)$



Less is More (Brent's Principle)

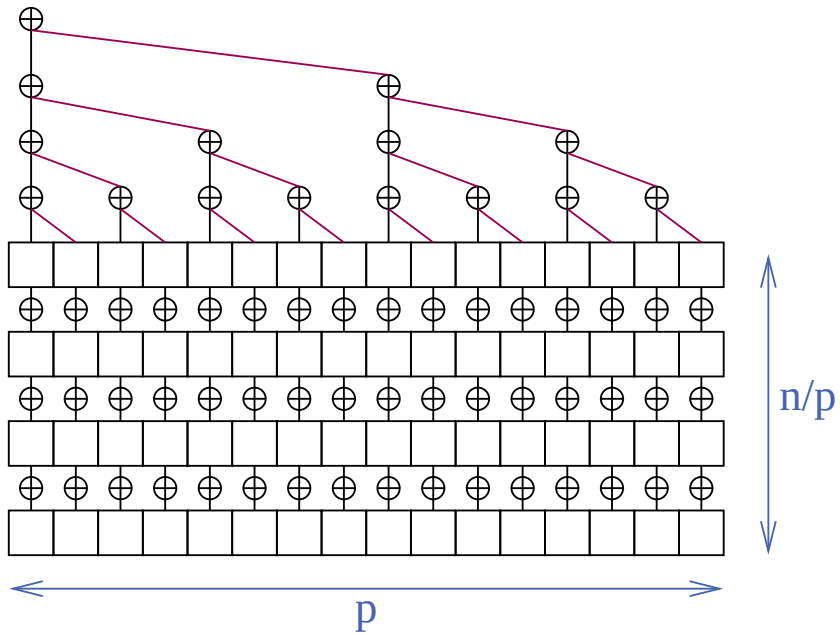
p PEs

Each PE adds

n/p elements sequentially.

Then, parallel sum

over p partial sums.



Time $T_{\text{seq}}(n/p) + \Theta(\log p)$

Efficiency

$$\frac{T_{\text{seq}}(n)}{p(T_{\text{seq}}(n/p) + \Theta(\log p))} = \frac{n}{p(n/p + \Theta(\log(p)))} = \frac{1}{1 + \Theta(p\log(p))/n}$$

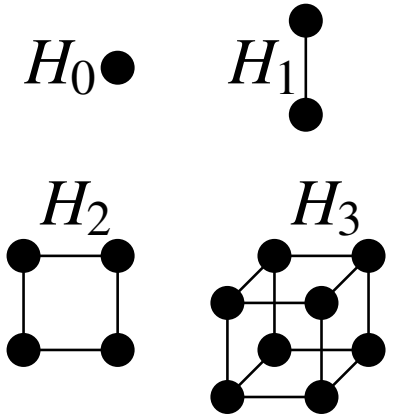
≈ 1 if $n \gg p \log p$

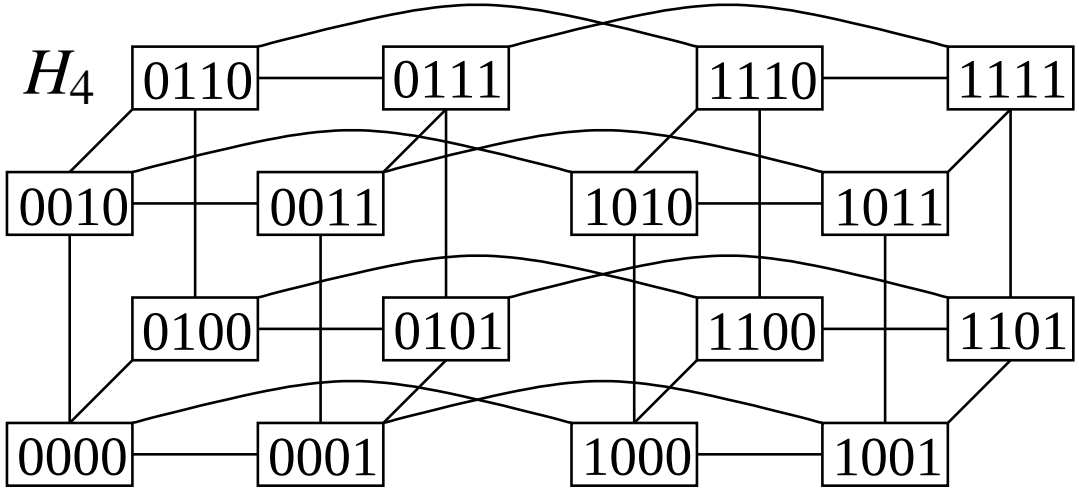
Discussion of Reduction

- Binary tree results in logarithmic running time
- Useful in most models
- Brent's principle: Inefficient algorithms become efficient by reducing the number of processors

Hypercube

hypercube



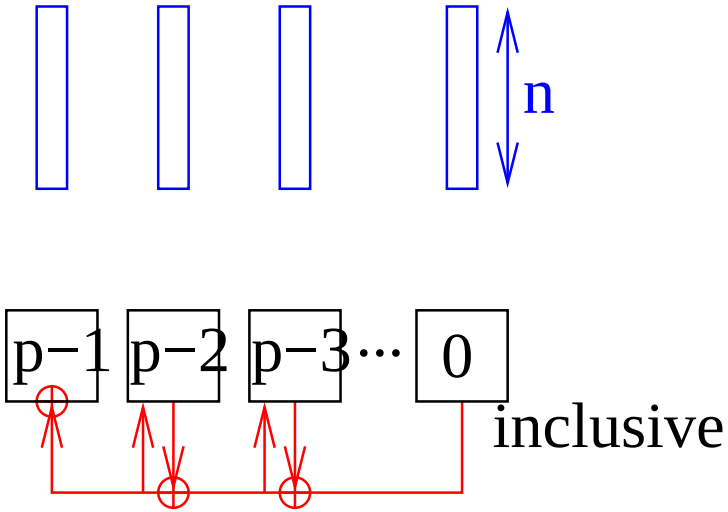


Prefix Sums

Requirement

$$x@i := \bigoplus_{i' \leq i} m@i'$$

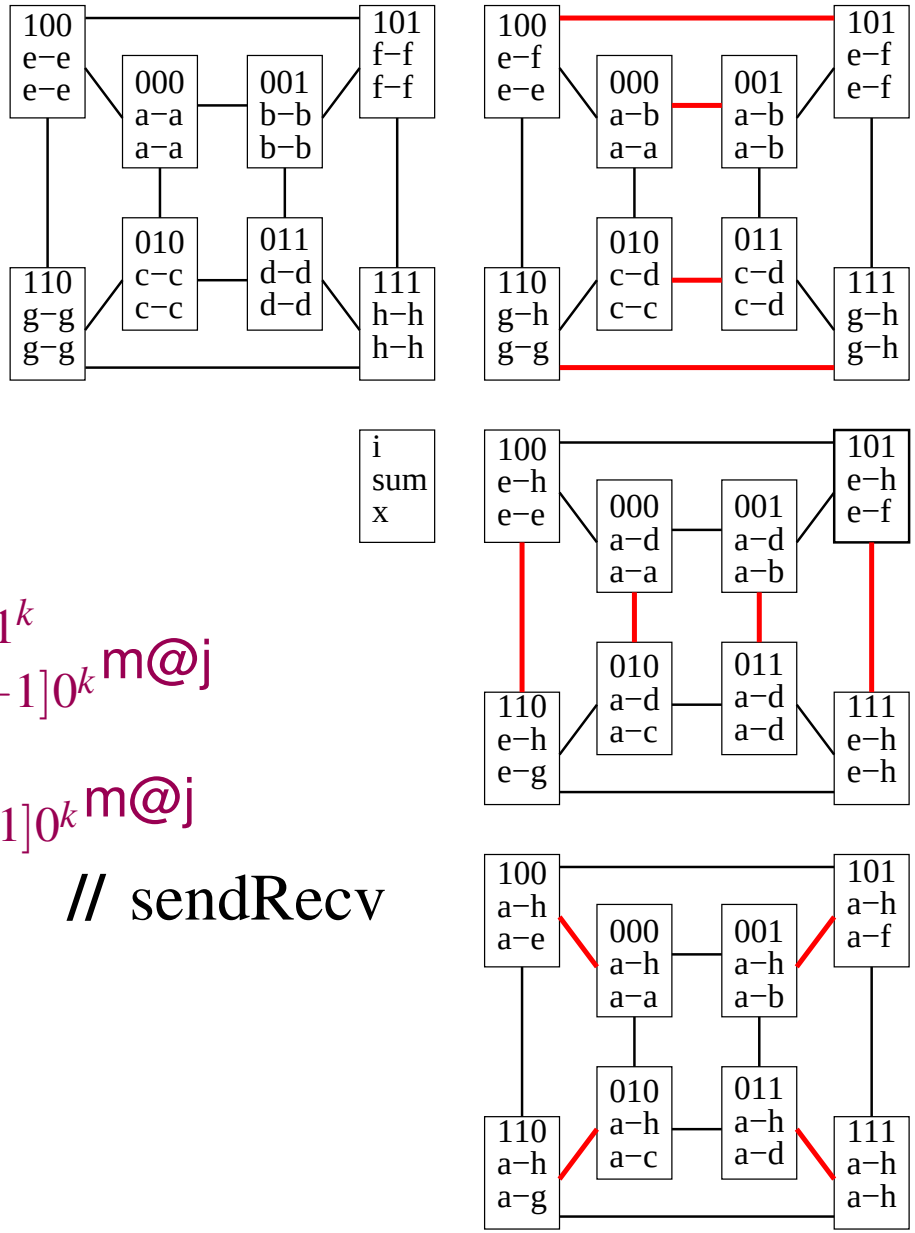
(on PE i) objects of length ℓ



inherently sequential ???

Hypercube Algorithm

// view PE index i as a
// d-bit bit array
Function hcPrefix(m)
  $x := \sigma := m$
  for $k := 0$ **to** $d-1$ **do**
    invariant $\sigma = \bigoplus_{j=i[k..d-1]0^k}^{i[k..d-1]1^k} \mathsf{m@j}$
    invariant $x = \bigoplus_{j=i[k..d-1]0^k}^{i} \mathsf{m@j}$
    $y := \sigma @ (i \text{ xor } 2^k)$ // sendRecv
    $\sigma := \sigma \oplus y$
    if $i[k] = 1$ **then** $x := x \oplus y$
  return x



Analysis

$$T_{\text{prefix}} = \mathrm{O}((\alpha + \ell\beta)\log p)$$

Problem:

Not optimal for $\ell\beta > \alpha$ (already seen for the operation reduction)

see lecture “Parallel Algorithms”

$\rightsquigarrow \mathrm{O}(\alpha \log P + \ell\beta)$

10.3 Sorting

- Parallel quicksort
- Parallel multi-way mergesort
- Not here: binary mergesort, radixsort,...

Parallel Quicksort

Sequential (simplified)

Procedure qSort$(d[], n)$

  if $n = 1$ **then return**

  select a pivot v

  reorder the elements in d such that

$$d_0 \cdots d_{k-1} \leq v < d_k \cdots d_{n-1}$$

  qSort($[d_0, \ldots, d_{k-1}]$, k)

  qSort($[d_{k+1}, \ldots, d_{n-1}]$, $n-k-1$)

Beginners' Parallelization

Parallelization of recursive calls.

$$T_{\text{par}} = \Omega(n)$$

- Very limited speedup
- Bad for distributed memory

Theoreticians' Parallelization

For simplification: $n = p$.

Idea: Also parallelize the partitioning

1. One PE picks a pivot (e.g., uniform at random).
2. Broadcast
3. Local comparison
4. Enumerate "'smaller"' elements (prefix sum)
5. Redistribute data
6. Split processors
7. Recurse in parallel

Theoreticians' Parallelization

// Let $i \in 0..p-1$ and p denote the 'local' PE index and partition size

Procedure theoQSort(d, i, p)

  if $p = 1$ **then return**

  r:= random element from $0..p-1$ // same value in entire partition

  $v := d@r$ // broadcast pivot

  $f := d \leq v$ // 1 iff d is on left side, 0 otherwise

  $j := \sum_{k=0}^{i} f@k$ // prefix sum, count elements on left side

  $p' := j@(p-1)$ // broadcast, result is border index

  if f **then** send d to PE $j-1$

  else send d to PE $p'+i-j$ // $i - j = \sum_{k=0}^{i} d@k > v$

  receive d

  if $i < p'$ **then** join left partition; qsort(d, i, p')

  else join right partition; qsort$(d, i-p', p-p')$

Example

pivot $v = 44$

PE Nummer	0	1	2	3	4	5	6	7
Nr. d. Elemente $\leq$ Pivot	0		1		2	3		4
Nr. d. Elemente $>$ Pivot		0		1			2	
Wert vorher	44	77	11	55	00	33	66	22
Wert nachher	44	11	00	33	22	77	55	66
PE Nummer	0+0	0+1	0+2	0+3	0+4	5+0	5+1	5+2

```
int pQuickSort(int item, MPI_Comm comm)
{ int iP, nP, small, allSmall, pivot;
  MPI_Comm newComm; MPI_Status status;
  MPI_Comm_rank(comm, &iP); MPI_Comm_size(comm, &nP);

  if (nP == 1) { return item; }
  else {
    pivot = getPivot(item, comm, nP);
    count(item <= pivot, &small, &allSmall, comm, nP);
    if (item <= pivot) {
      MPI_Bsend(&item,1,MPI_INT,   small - 1     ,8,comm);
    } else {
      MPI_Bsend(&item,1,MPI_INT,allSmall+iP-small,8,comm);
    }
    MPI_Recv(&item,1,MPI_INT,MPI_ANY_SOURCE,8,comm,&status);
    MPI_Comm_split(comm, iP < allSmall, 0, &newComm);
    return pQuickSort(item, newComm);}}
```

```
/* determine a pivot */
int getPivot(int item, MPI_Comm comm, int nP)
{   int pivot   = item;
    int pivotPE = globalRandInt(nP);/* from random PE */
    /* overwrite pivot by that one from pivotPE */
    MPI_Bcast(&pivot, 1, MPI_INT, pivotPE, comm);
    return pivot;
}

/* determine prefix-sum and overall sum over value */
void
count(int value,int *sum,int *allSum,MPI_Comm comm,int nP)
{   MPI_Scan(&value, sum, 1, MPI_INT, MPI_SUM, comm);
    *allSum = *sum;
    MPI_Bcast(allSum, 1, MPI_INT, nP - 1, comm);
}
```

Analysis

- Per recursion level:
  - $2\times$ broadcast
  - $1\times$ prefixsum
  - $\rightsquigarrow$ Time $\mathrm{O}(\alpha \log p)$
- Expected recursion depth: $\mathrm{O}(\log p)$

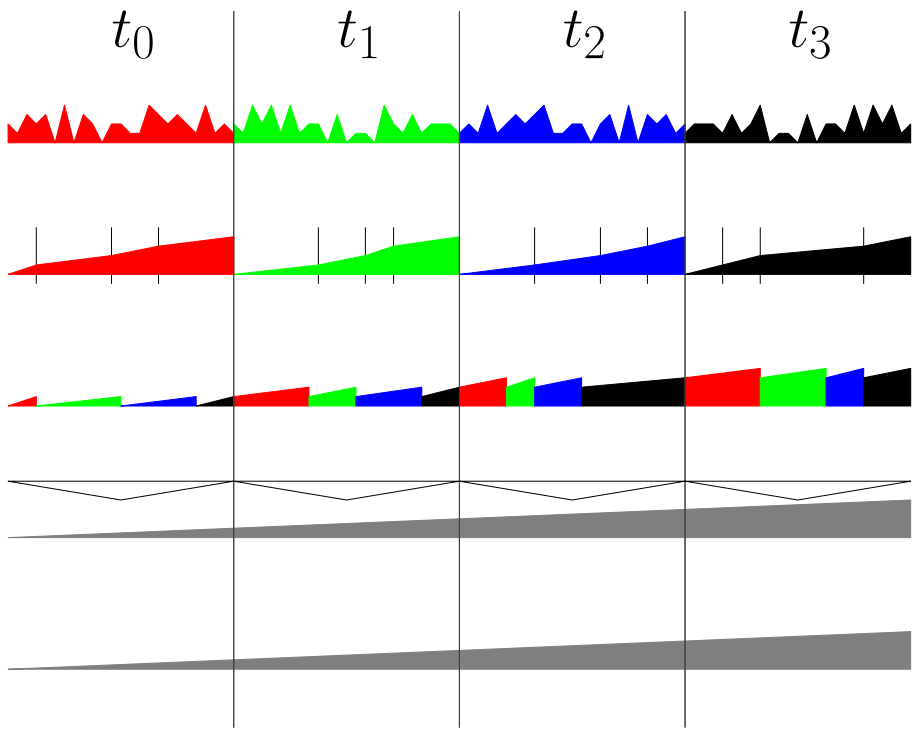
Expected total running time: $\mathrm{O}(\alpha \log^2 p)$

Generalization for $n \gg p$ is straightforward?

- Each PE has in general "small" and "large" elements.
- Cannot be divided equally
- Prefix sum still useful
- Bottom line: Time $\mathrm{O}\left(\frac{n \log n}{p} + \log^2 p\right)$ possible
- $\Omega(\log p)$ transfers of each element may be a problem on distributed memory machines
  $\rightsquigarrow \cdots \rightsquigarrow$ time $\mathrm{O}\left(\frac{n}{p}(\log n + \beta \log p) + \alpha \log^2 p\right)$

Parallel Sorting with Multi-Way Merging

1. p processors sort n/p local elements each
2. Find pivots p_i such that n/p elements are between two adjacent pivots
3. Processor i sorts all elements between pivots p_i and p_{i+1} with a multi-way merge.

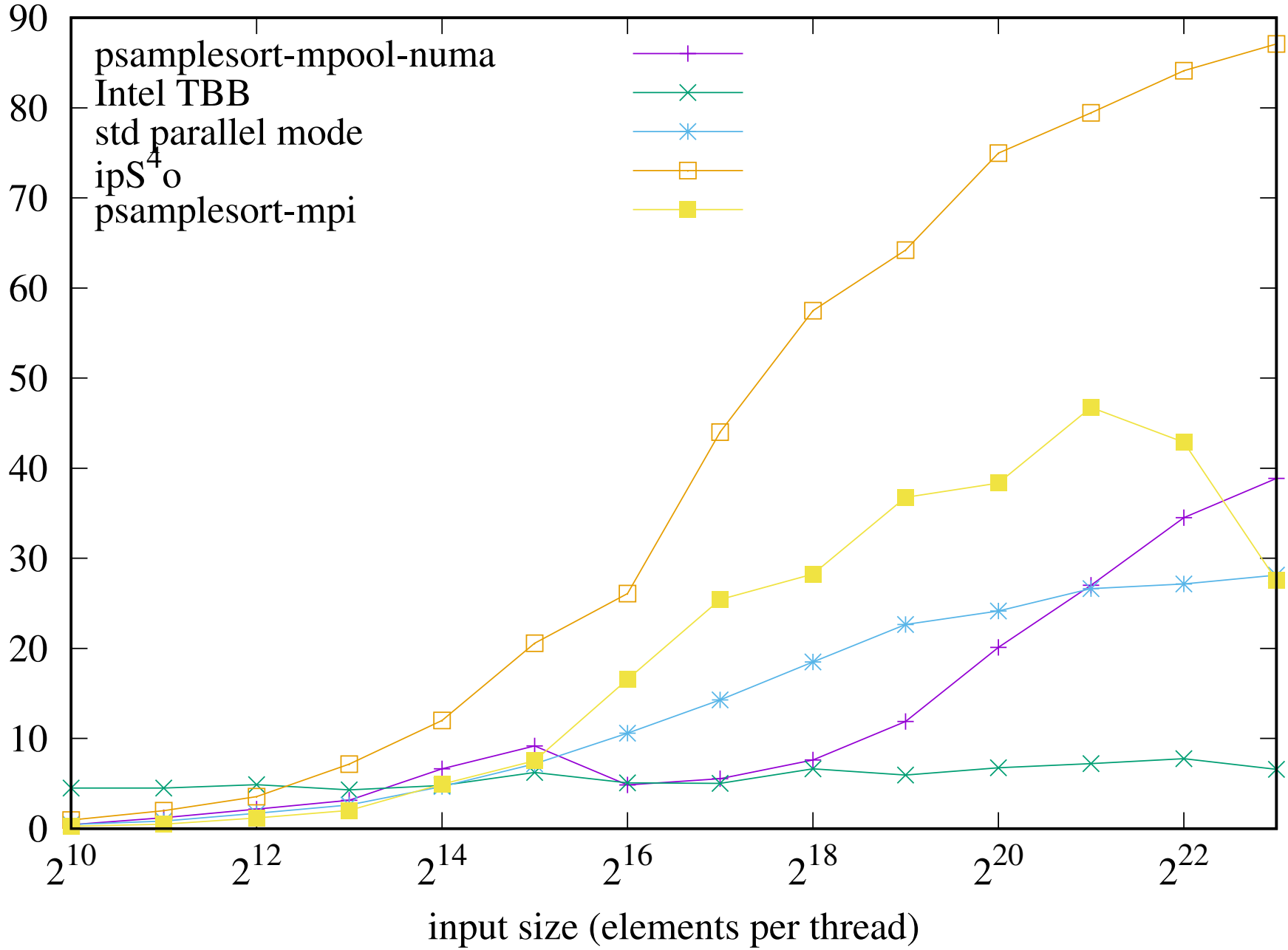


More details in lecture “Parallel Algorithms”

More on Parallel Sorting

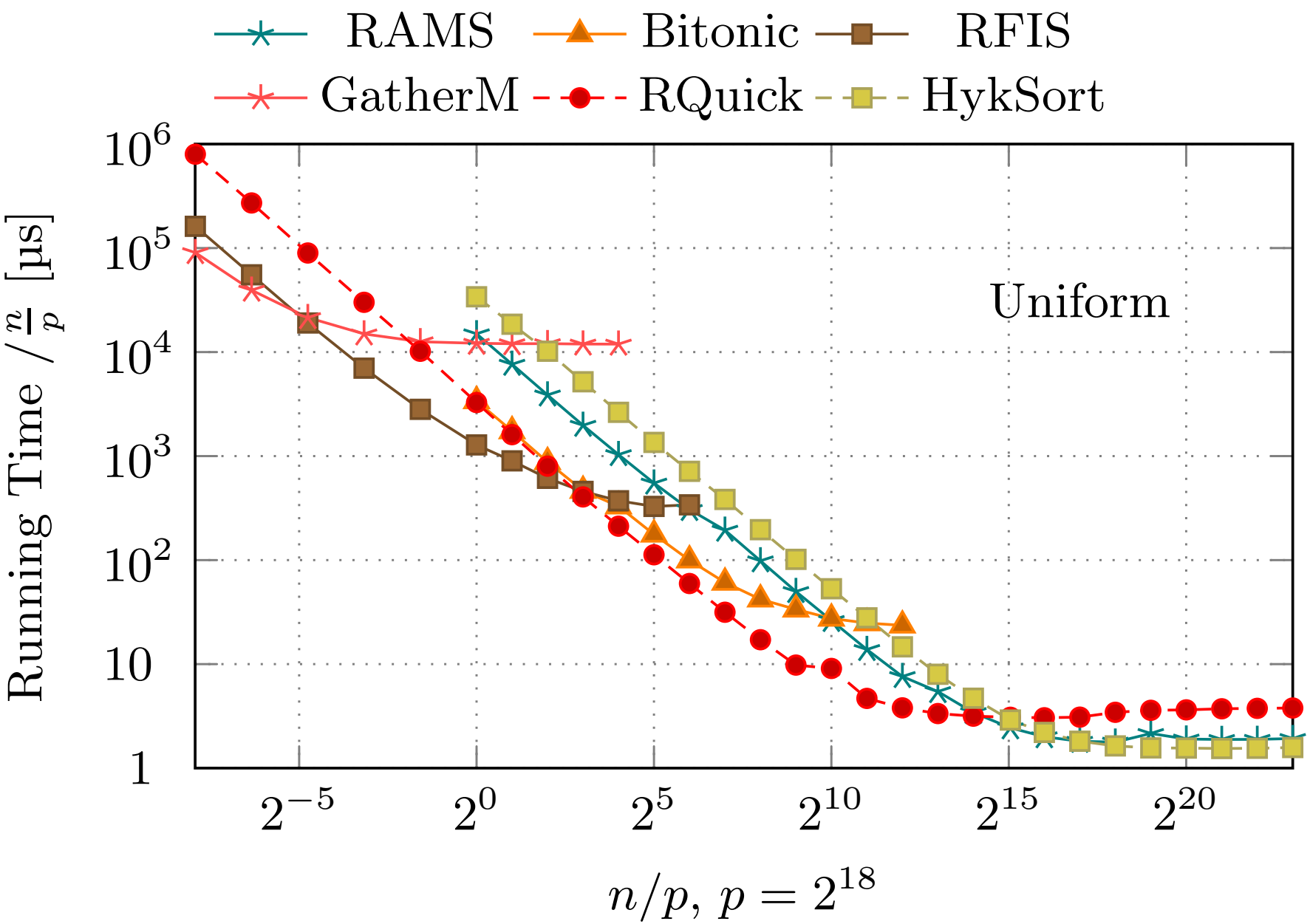
- Theoreticians' algorithm with time $\mathrm{O}(\log p)$ for $p = n$
- Practical algorithms with time $\mathrm{O}(\log p)$ for $n = \mathrm{O}\left(\sqrt{p}\right)$ – if speed is more important than efficiency, e.g., for the base case
- Sample sort: k-way generalization of quicksort
- Parallel external sorting

Experiments Speedup on 4× Intel E7-8890 v3 – 72 cores



Measurements

Axtmann Sanders ALENEX 2017



More on Parallel Algorithms – Parallelization of the Basic Toolbox ?

Parallel hash tables: possible, usually expensive $\rightsquigarrow$ [Maier S Dementiev 16]

Priority queues, search trees: similar problem. Most practical: batched updates$\rightsquigarrow$ [Hübschle-Schn. S Müller 16][Akhremtsev S 16]

BFS: OK for graphs with small diameter

DFS: $\approx$ inherently not parallelizable

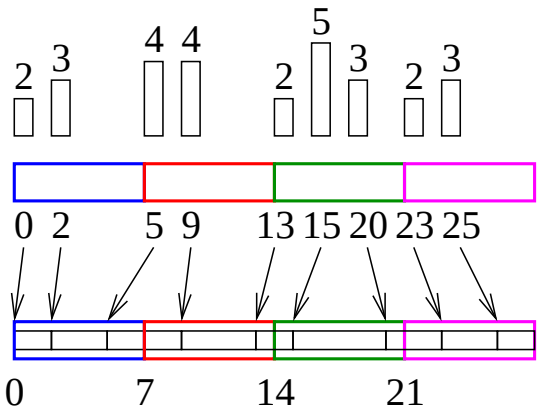
Shortest paths: similar BFS. But all-to-all, precomputation OK, multiobjective OK [Mandow S 13]

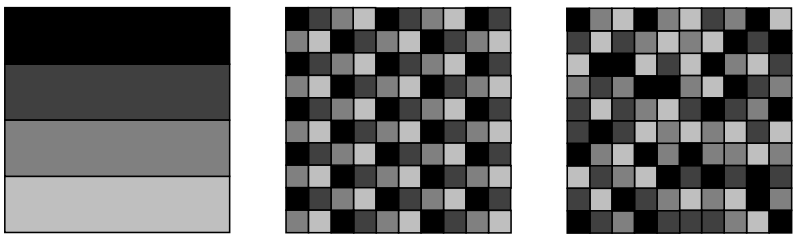
MST: Yes! But large constant factors?

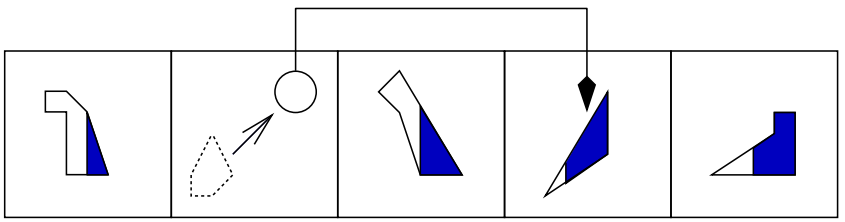
Optimization methods: dynamic programming OK. Greedy??? LP is hard, some metaheuristics OK

More on Parallel Algorithms – Parallelization of the Basic Toolbox

- Collective communication: reduction, prefix sum, in general message exchange, gossiping,. . .
- Load balancing
  - static/dynamic
  - known/unknown job sizes
  - centralized/completely distributed







processed by: =PE 0 =PE 1 =PE 2 =PE 3