Design and Analysis of Parallel Programs

TDDE35 Lecture 3-4

Christoph Kessler

PELAB / IDA
Linköping University
Sweden

Background reading: C. Kessler, "*Design and Analysis of Parallel Algorithms – An Introduction*", Compendium TDDE65/TDDD56 Chapter 2. (c) 2019-2024
https://www.ida.liu.se/~TDDE65/handouts.shtml login: parallel

Outline

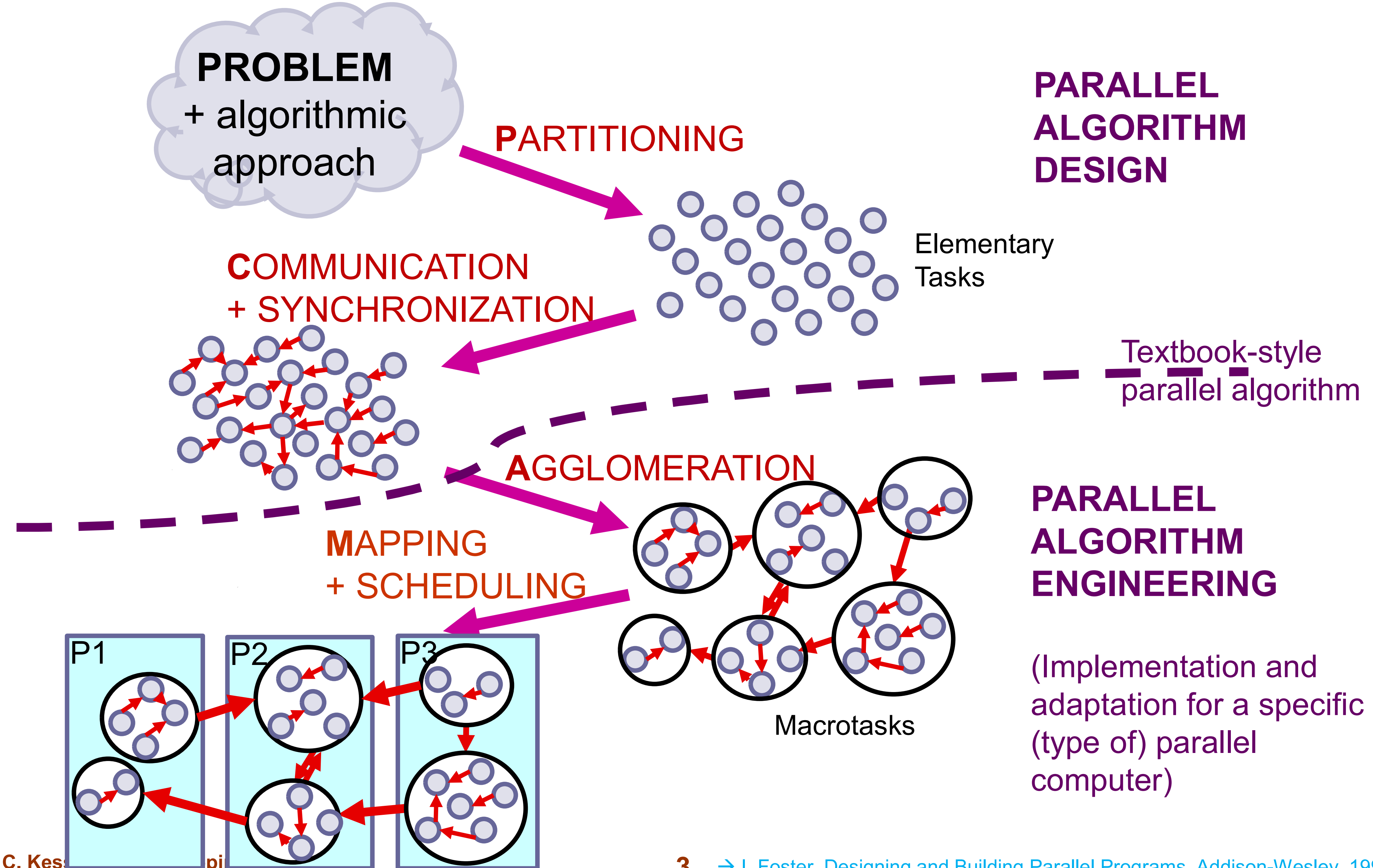
Design and analysis of parallel algorithms

- Foster's PCAM method for the design of parallel programs
- Parallel cost models
- Parallel work, time, cost
- Parallel speed-up; speed-up anomalies
- Amdahl's Law
- Fundamental parallel algorithms: Parallel prefix, List ranking

+ TDDD56: Parallel Sorting Algorithms

+ TDDE65: Parallel Linear Algebra and Linear System Solving

Foster's Method for Design of Parallel Programs ("PCAM")



→ I. Foster, Designing and Building Parallel Programs. Addison-Wesley, 1995.

Parallel Cost Models

A Quantitative Basis for the Design of Parallel Algorithms

Christoph Kessler, IDA,
Linköping University

Parallel Computation Model
= Programming Model + Cost Model

\+ abstract from hardware and technology

\+ specify basic operations, when applicable

\+ specify how data can be stored

→ analyze algorithms before implementation independent of a particular parallel computer → $T = f(n, p, ...)$

→ focus on most characteristic (w.r.t. influence on exec. time) features of a broader class of parallel machines

Programming model

- shared memory / message passing,
- degree of synchronous execution

Cost model

- key parameters
- cost functions for basic operations
- constraints

Parallel Computation Models

Shared-Memory Models

- PRAM (Parallel Random Access Machine) [Fortune, Wyllie '78]
  including variants such as Asynchronous PRAM, QRQW PRAM
- Data-parallel computing
- Task Graphs (Circuit model; Delay model)
- …

Message-Passing Models

- BSP (Bulk-Synchronous Parallel) Computing [Valiant'90]
  including variants such as Multi-BSP [Valiant'08]
- MPI (programming model)
  + Delay-model or LogP-model (cost model)
- Synchronous reactive (event-based) programming e.g. Erlang
- Dataflow programming
- …

Cost Model

Cost model: should

+ explain available observations

+ predict future behaviour

+ abstract from unimportant details $\rightarrow$ generalization

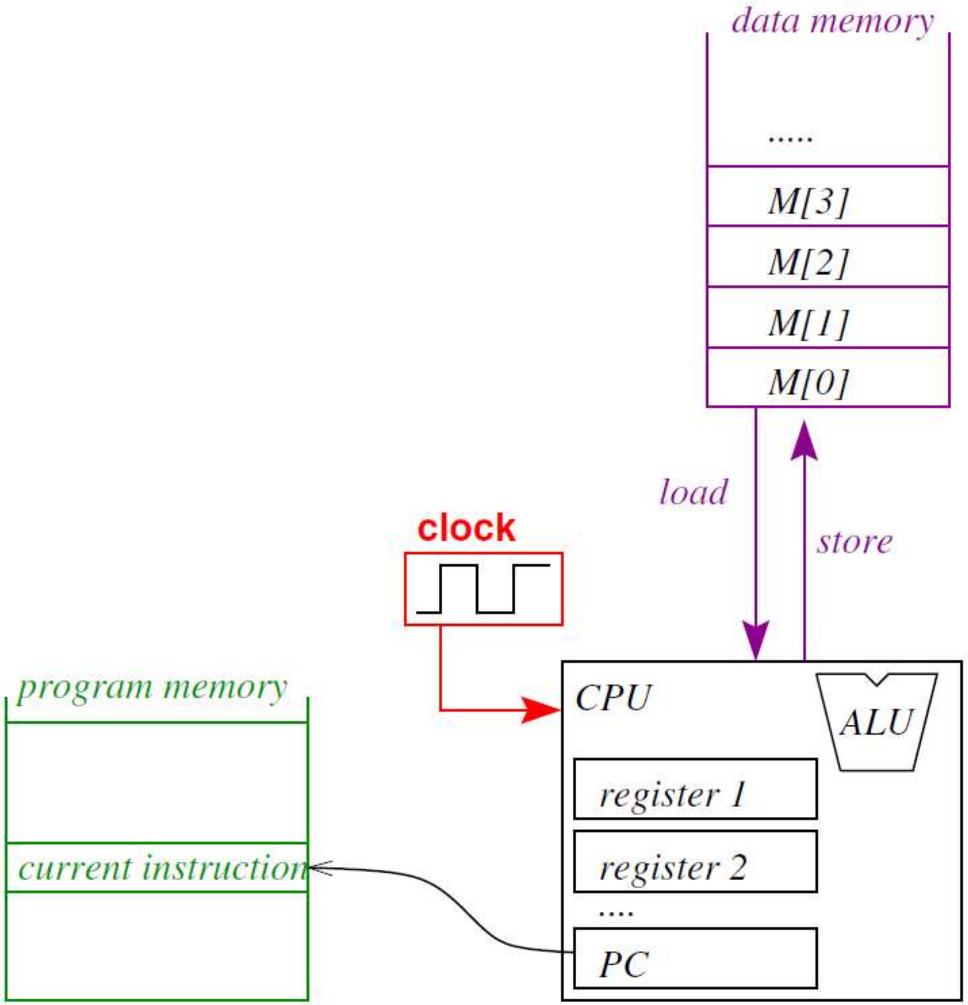
Simplifications to reduce model complexity:

- use idealized multicomputer model
  ignore hardware details: memory hierarchies, network topology, ...
- use scale analysis
  drop insignificant effects
- use empirical studies
  calibrate simple models with empirical data
  rather than developing more complex models

Flashback to DALG, Lecture 1:
The RAM (von Neumann) model for sequential computing

RAM (Random Access Machine)

programming and cost model for the analysis of sequential algorithms



Basic operations (instructions):
- Arithmetic (add, mul, …) on registers
- Load
- Store
- Branch

Simplifying assumptions
for time analysis:
- All of these take 1 time unit
- Serial composition adds time costs
  T(op1;op2) = T(op1)+T(op2)

C.

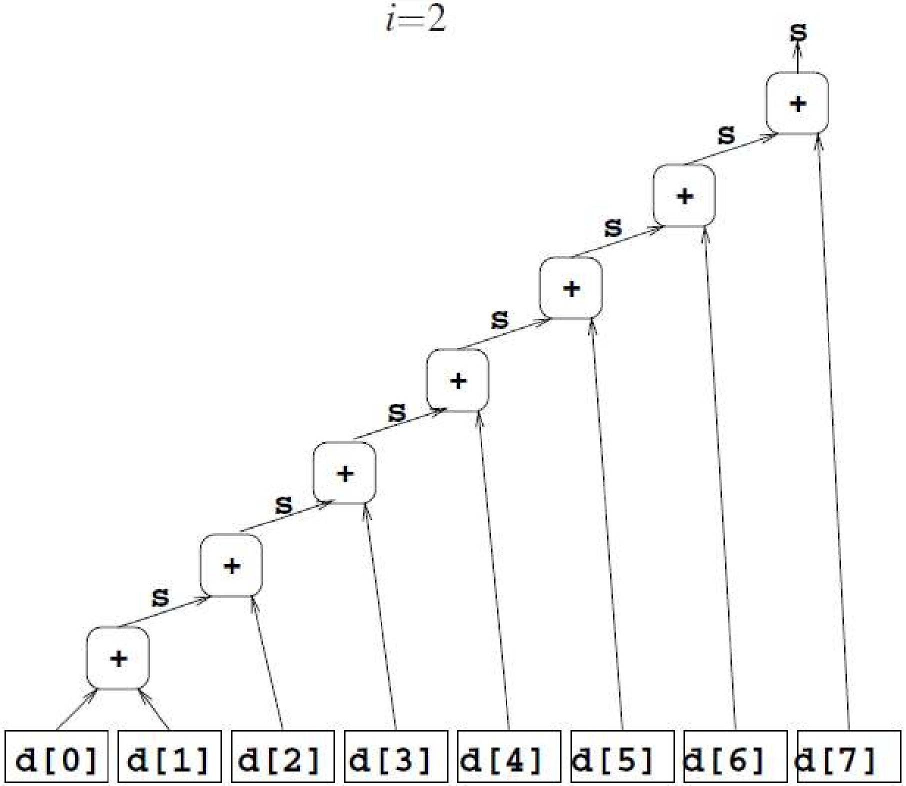
Analysis of sequential algorithms: RAM model (Random Access Machine)

Algorithm analysis: Counting instructions

Example: Computing the global sum of N elements

```
s = d[0];
for (i=1; i<N; i++)
      s = s + d[i];
```

$$t = t_{load} + t_{store} + \sum_{i=2}^{N} (2t_{load} + t_{add} + t_{store} + t_{branch}) = 5N - 3 \in \Theta(N)$$



← *Data flow graph*,
showing dependences
(precedence constraints)
between operations

C. → arithmetic circuit model, directed acyclic graph (DAG) model

The PRAM Model – a Parallel RAM

Parallel Random Access Machine [Fortune/Wyllie'78]

p processors

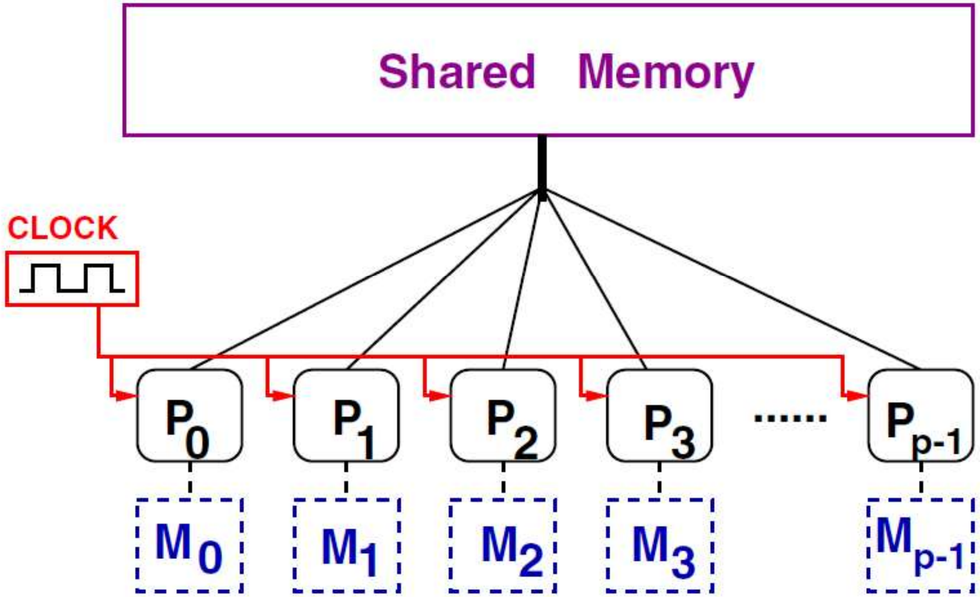
- MIMD
- common clock signal
- arithm./jump: 1 clock cycle

shared memory

- uniform memory access time
- latency: 1 clock cycle (!)
- concurrent memory accesses
- sequential consistency

private memory (optional)

- processor-local access only



Remark

PRAM model is very idealized,
extremely simplifying / abstracting from real parallel architectures:

unbounded number of processors:

abstracts from scheduling overhead

The PRAM cost model has only 1 machine-specific parameter: the number of processors

local operations cost 1 unit of time

every processor has unit time memory access to any shared memory location:

abstracts from communication time, bandwidth limitation, memory latency, memory hierarchy, and locality

$\rightarrow$ focus on pure, fine-grained parallelism

→ Good for rapid prototyping of parallel algorithm designs:

A parallel algorithm that does not scale under the PRAM model does not scale well anywhere else!

PRAM Variants

Exclusive Read, Exclusive Write (EREW) PRAM

concurrent access only to different locations in the same cycle

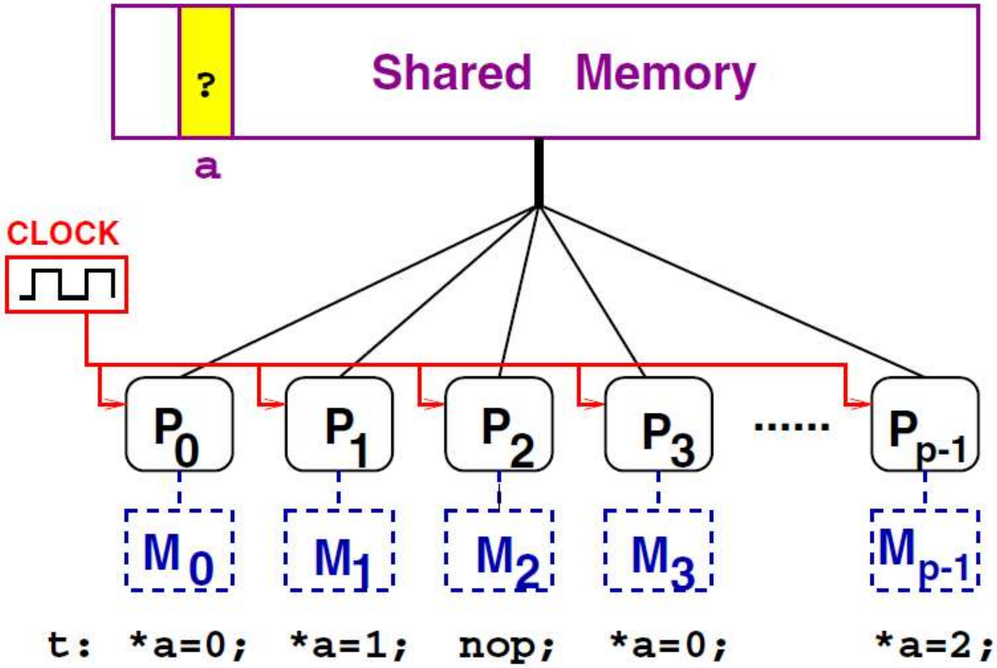
Concurrent Read, Exclusive Write (CREW) PRAM

simultaneous reading from *or* writing to same location is possible:

Concurrent Read, Concurrent Write (CRCW) PRAM

simultaneous reading from *or* writing to same location is possible:

- Weak CRCW
- Common CRCW
- Arbitrary CRCW
- Priority CRCW
- Combining CRCW
  (global sum, max, etc.)



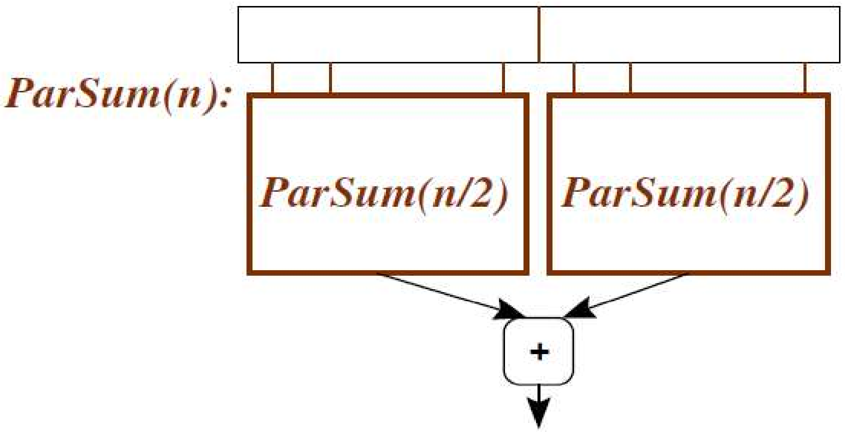
No need for ERCW ...

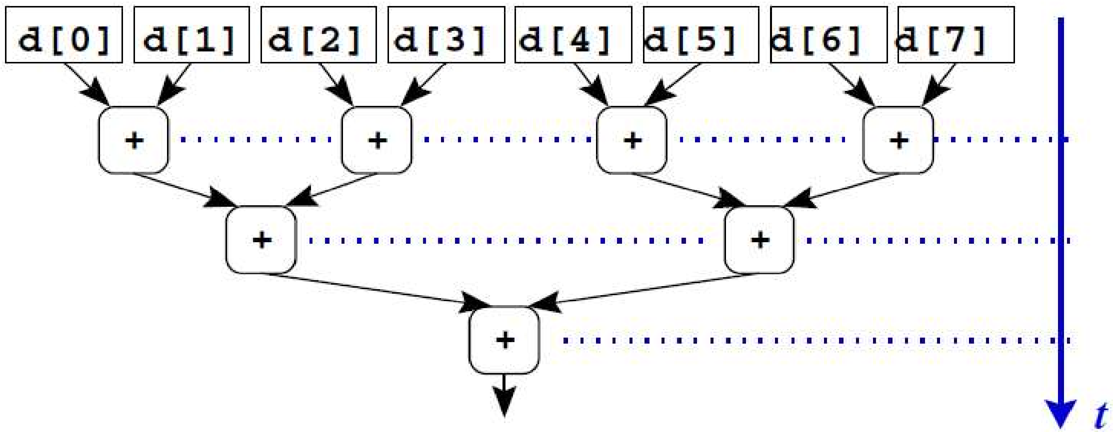
Divide&Conquer Parallel Sum Algorithm in the PRAM / Circuit (DAG) cost model

Given n numbers $x_0, x_1, \ldots, x_{n-1}$ stored in an array.

The global sum $\sum_{i=0}^{n-1} x_i$ can be computed in $\lceil \log_2 n \rceil$ time steps on an EREW PRAM with n processors.

Parallel algorithmic paradigm used: Parallel Divide-and-Conquer





Divide phase: trivial, time $O(1)$
Recursive calls: parallel time $T(n/2)$
with base case: load operation, time $O(1)$
Combine phase: addition, time $O(1)$

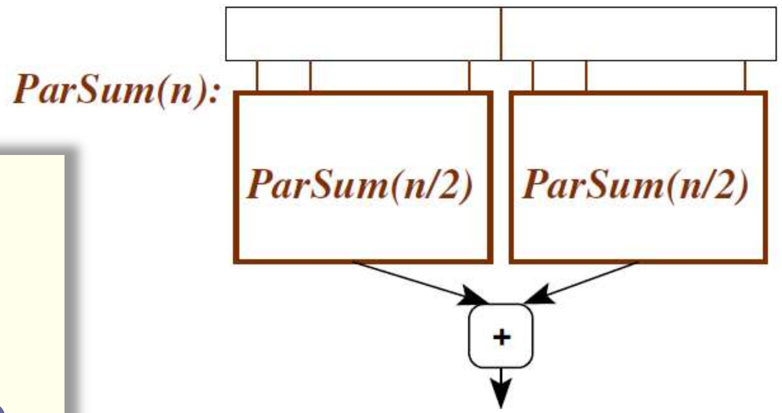
Recurrence equation for parallel execution time:

$$\begin{cases} T(n) = T(n/2) + O(1) \\ T(1) = O(1) \end{cases}$$

Use induction or the master theorem [Cormen+'90 Ch.4] $\rightarrow T(n) \in O(\log n)$

Recursive formulation of DC parallel sum algorithm in some programming model

Implementation e.g. in **Cilk**: (shared memory)




```
cilk int parsum ( int *d, int from, int to )
{
    int mid, sumleft, sumright;
    if (from == to)  return d[from];    // base case
    else {
        mid = (from + to) / 2;
        sumleft = spawn parsum ( d, from, mid );
        sumright = spawn parsum( d, mid+1, to );
        sync;
        return sumleft + sumright;
    }
}
```

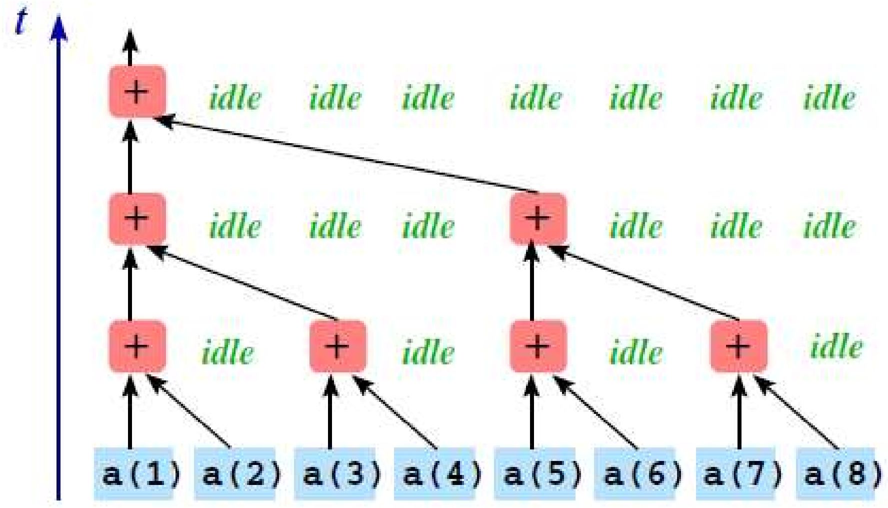
Fork-Join execution style:
a single task starts,
tasks spawn child tasks for
independent recursive calls,
and synchronize with them

```
// The main program:

main()
{
  ...
  parsum ( data, 0, n-1 );
  ...
}
```

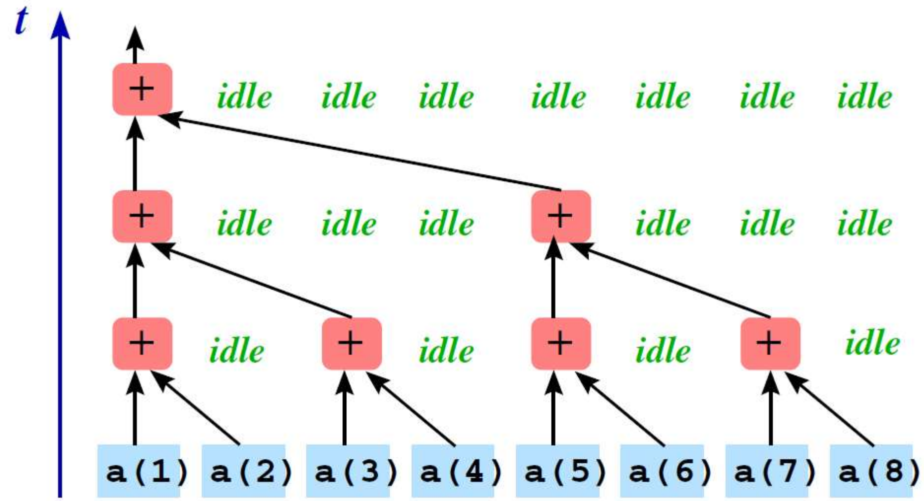
Iterative formulation of DC parallel sum in EREW-PRAM model


```
int sum(sh int a[], sh int n)
{
  int d, dd;
  int ID = rerank();
  d = 1;
  while (d<n) {
    dd = d; d = d*2;
    if (ID%d==0)  a[ID] = a[ID] + a[ID+dd];
  }
}
```



Circuit / DAG model

- Independent of how the parallel computation is expressed, the resulting (unfolded) task graph looks the same.



- **Task graph** is a directed acyclic graph (DAG) G=(*V*,*E*)
  - Set *V* of vertices: elementary tasks (taking time 1 resp. *O*(1) each)
  - Set *E* of directed edges: dependences (partial order on tasks) (v_1,v_2) in *E* → v_1 must be finished before v_2 can start
- **Critical path** = longest path from an entry to an exit node
  - Length of critical path is a lower bound for parallel time complexity
- **Parallel time** can be longer if number of processors is limited
  → *schedule tasks* to processors such that dependences are preserved - by programmer (SPMD execution) or run-time system (fork-join execution)

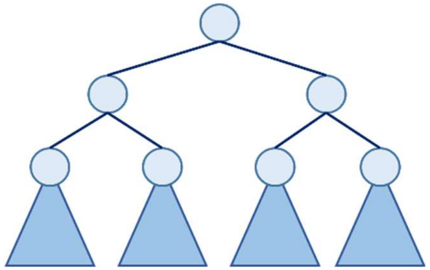
For a fixed number of processors … ?

- Usually, $p << n$
- Requires scheduling the work to p processors

(A) manually, at algorithm design time:

- Requires **algorithm engineering**
- E.g. for parallel sum:

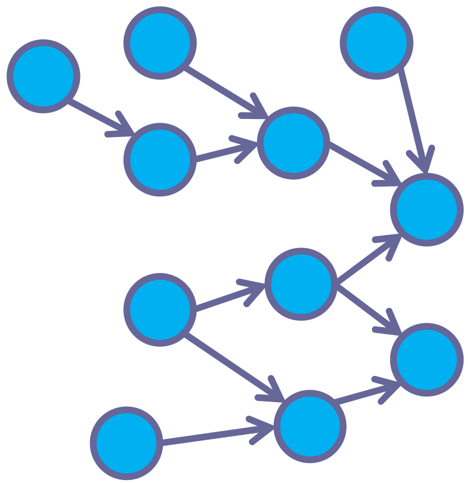
  stop the parallel divide-and-conquer
  e.g. at subproblem size n/p
  and switch there to sequential divide-and-conquer
  (= task agglomeration)



  - Step 0. Partition the array of n elements in p slices of n/p elements each (= domain decomposition)
  - Step 1. Each processor calculates a local sum for one slice, using the sequential sum algorithm, resulting in p partial sums (intermediate values)
  - Step 2. The p processors run the parallel algorithm to sum up the intermediate values to the global sum.

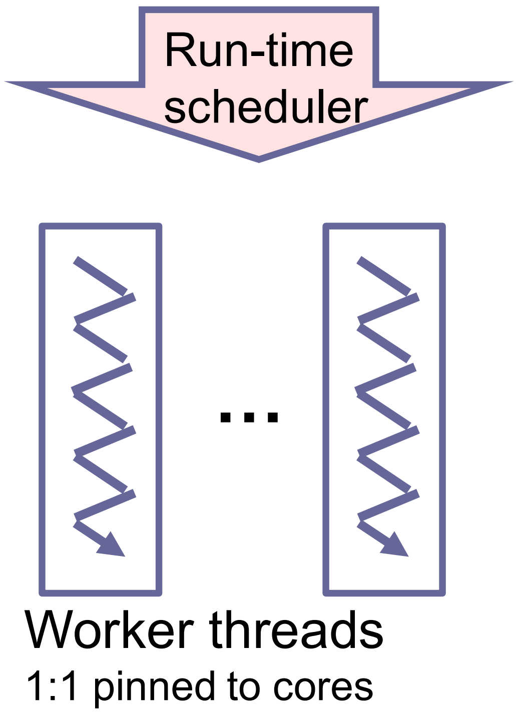
For a fixed number of processors ... ?

- Usually, $p << n$
- Requires scheduling the work to p processors



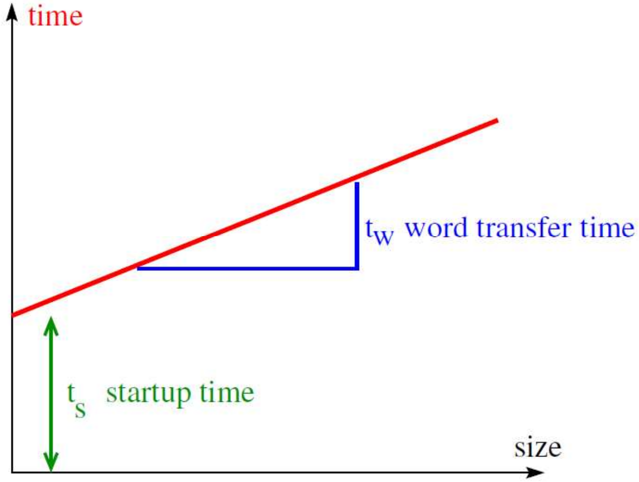
(B) automatically, at run time:

- Requires a **task-based runtime system** with dynamic scheduler
  - Each newly created task is dispatched at runtime to an available worker processor
    - run-time overhead ☹
  - Dynamic load balancing ☺
    - Central task queue where idle workers fetch next task to execute
    - Local task queues + Work stealing – idle workers steal a task from some other processor



Delay Model

Idealized multicomputer: point-to-point communication costs overhead t_{msg}.



Cost of communicating a larger block of n bytes:

time $t_{msg}(n) =$ sender overhead + latency + receiver overhead + n/bandwidth

$$=: \quad t_{startup} + n \cdot t_{transfer}$$

LINEAR

Assumption: network not overloaded; no conflicts occur at routing

$t_{startup}$ = startup time (time to send a 0-byte message)

accounts for hardware and software overhead.

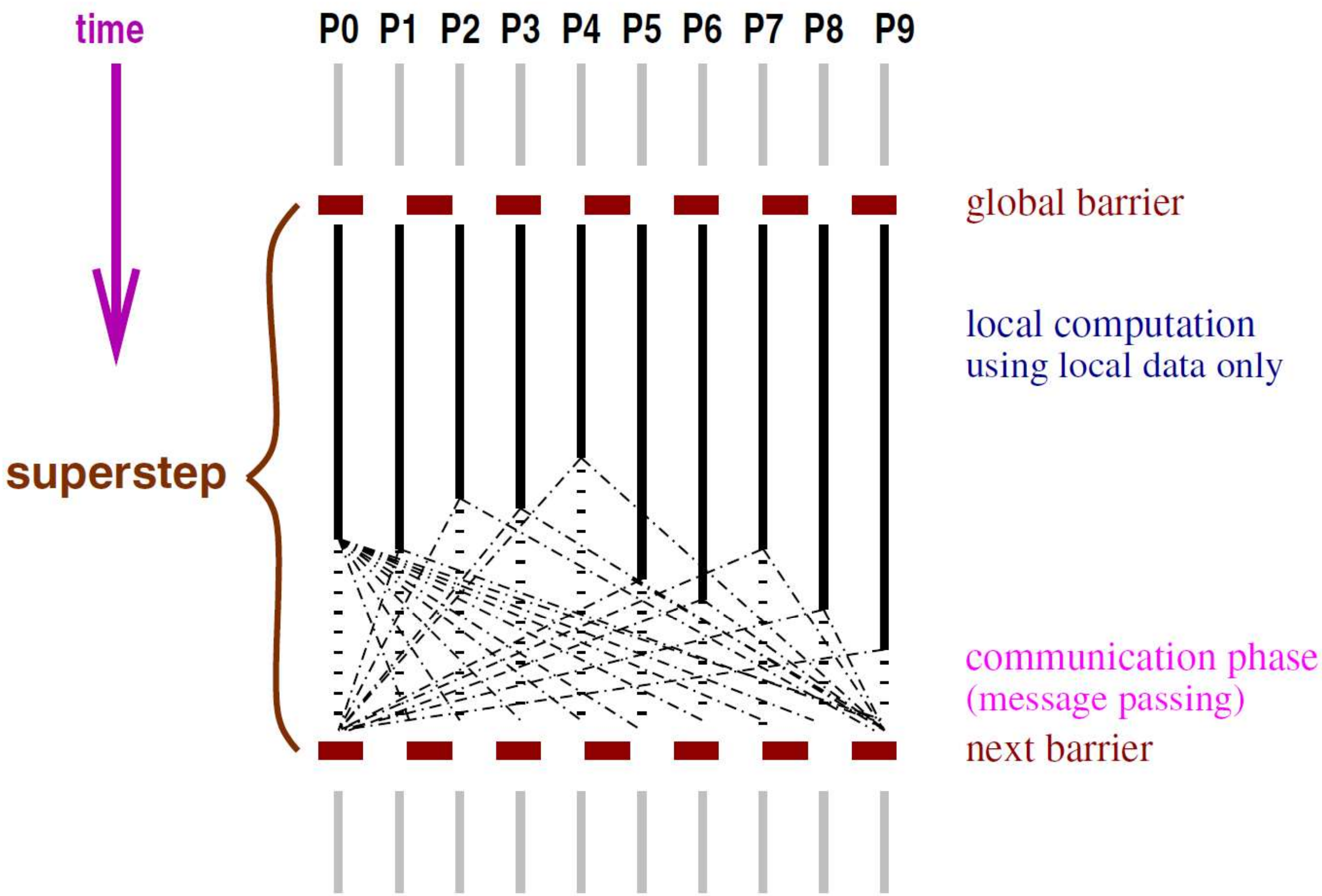
$t_{transfer}$ = transfer rate, send time per word sent.

depends on the network bandwidth.

BSP-Model

Bulk-synchronous parallel programming [Valiant'90] [McColl'93]

BSP computer = abstract message passing architecture (p, L, g, s)



MIMD

SPMD

h-relation models communication pattern / volume

h_i [words] = comm. fan-in, fan-out of P_i

$h = \max_{1 \le i \le p} h_i$

$t_{step} = w + hg + L$

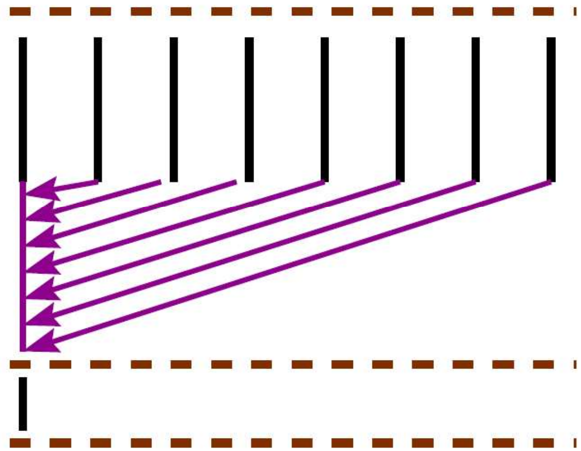
BSP program = sequence of supersteps, separated by (logical) barriers

BSP Example:

Global Maximum (NB: non-optimal algorithm)

Compute maximum of n numbers $A[0,\dots,n-1]$ on BSP(p, L, g, s):

```
// A[0..n-1] distributed block-wise across p processors
step
    // local computation phase:
    m ← −∞;
    for all A[i] in my local partition of A {
        m ← max (m, A[i]);
    // communication phase:
    if myPID ≠ 0
        send ( m, 0 );
    else      // on P0:
        for each i ∈ {1,...,p−1}
            recv ( m_i, i );
step
    if myPID = 0
        for each i ∈ {1,...,p−1}
            m ← max(m, m_i);
```



Local work:

$\Theta(n/p)$

Communication:

$h = p - 1$

(P_0 is bottleneck)

$$t_{step} = w + hg + L$$

$$= \Theta\left(\frac{n}{p} + pg + L\right)$$

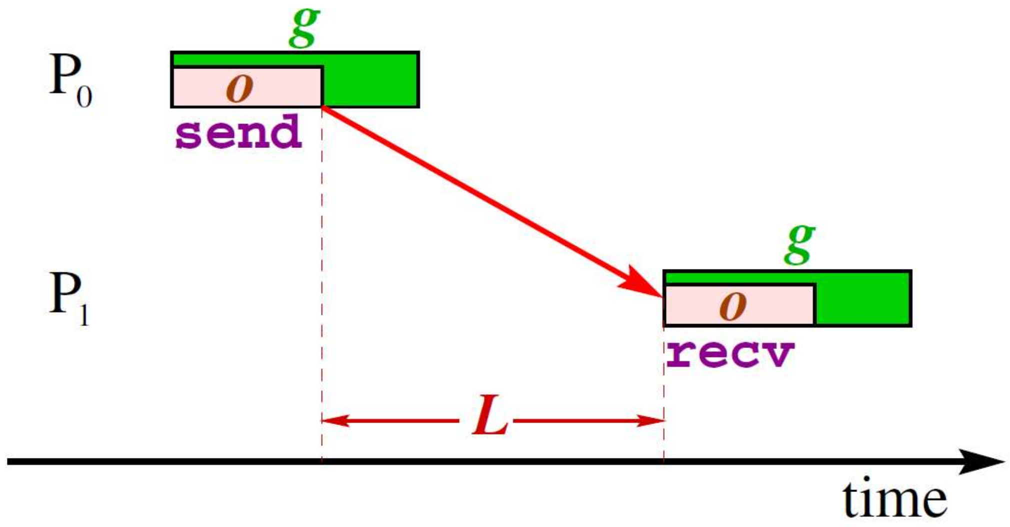
LogP Model → TDDE65

LogP model [Culler et al. 1993]

for the cost of communicating small messages (a few bytes)

4 parameters:

- latency L
- overhead o
- gap g (models bandwidth)
- processor number P



abstracts from network topology

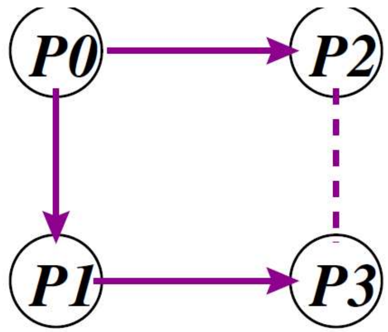
gap g = inverse network bandwidth per processor:
Network capacity is L/g messages to or from each processor.

L, o, g typically measured as multiples of the CPU cycle time.

transmission time for a small message:

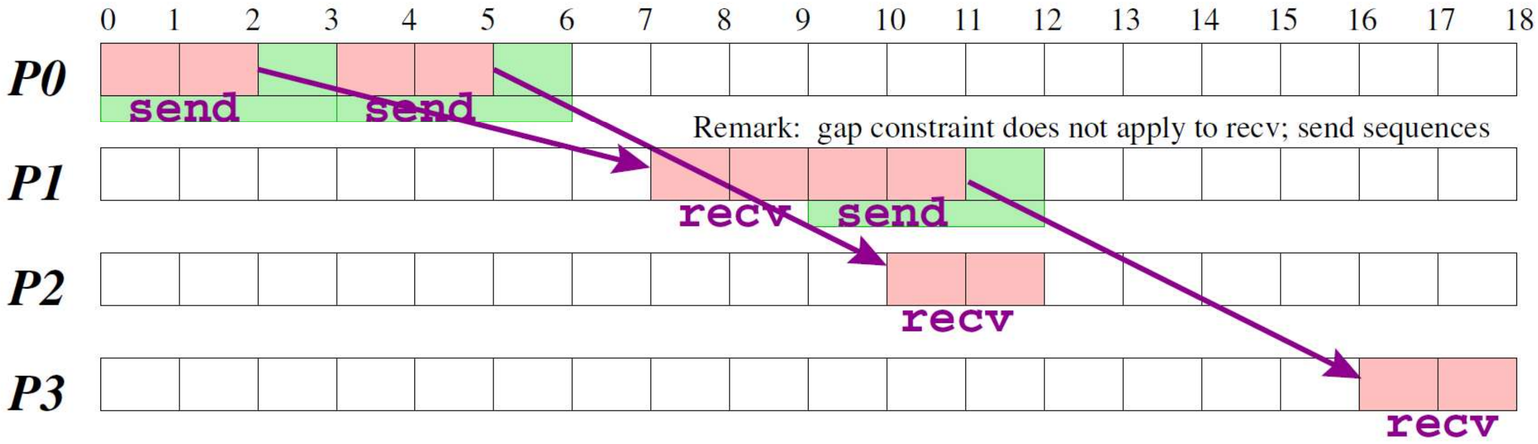
$2 \cdot o + L$ if the network capacity is not exceeded

LogP Model: Example → TDDE65



Example: Broadcast on a 2-dimensional hypercube

With example parameters $P = 4, \; o = 2\mu s, \; g = 3\mu s, \; L = 5\mu s$



it takes at least $18\mu s$ to broadcast 1 byte from $P0$ to $P1, P2, P3$

Remark: for determining time-optimal broadcast trees in LogP, see [Papadimitriou/Yannakakis'89], [Karp et al.'93]

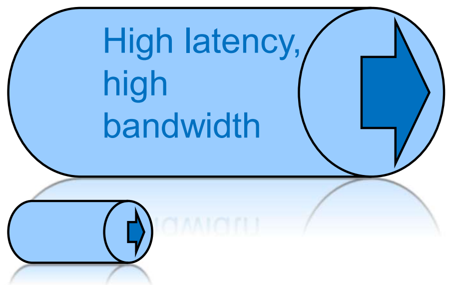
Analysis of Parallel Algorithms

Christoph Kessler, IDA,
Linköping University

Analysis of Parallel Algorithms

Performance metrics of parallel programs

- **Parallel execution time**
  - Counted from the start time of the earliest task to the finishing time of the latest task
- **Work** – the total number of performed elementary operations
- **Cost** – the product of parallel execution time and #processors
- **Speed-up**
  - the factor by how much faster we can solve a problem with *p* processors than with 1 processor, usually in range (0…*p*)
- **Parallel efficiency** = Speed-up / #processors, usually in (0…1)
- **Throughput** = #operations finished per second
- **Scalability**
  - does speedup keep growing well also when #processors grows large?



Analysis of Parallel Algorithms

Asymptotic Analysis

- Estimation based on a cost model and algorithm idea (pseudocode operations)
- Discuss behavior for large problem sizes, large #processors

Empirical Analysis

- Implement in a concrete parallel programming language
- Measure time on a concrete parallel computer
  - Vary number of processors used, as far as possible
- More precise
- More work, and fixing bad designs at this stage is expensive

Parallel Time, Work, Cost

problem size n

\# processors p

time $t(p,n)$

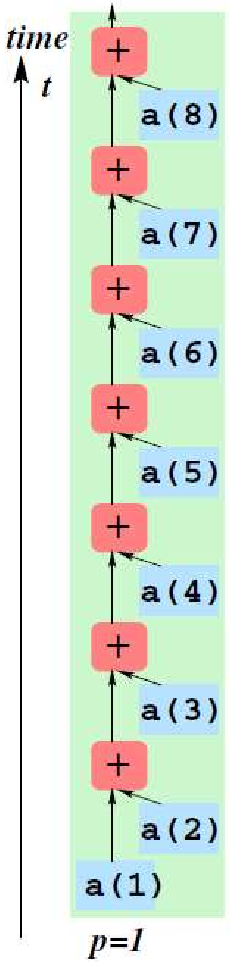
work $w(p,n)$

cost $c(p,n) = t \cdot p$

Example:
seq. sum algorithm

```
s = a(1)
do i = 2, n
  s = s + a(i)
end do
```

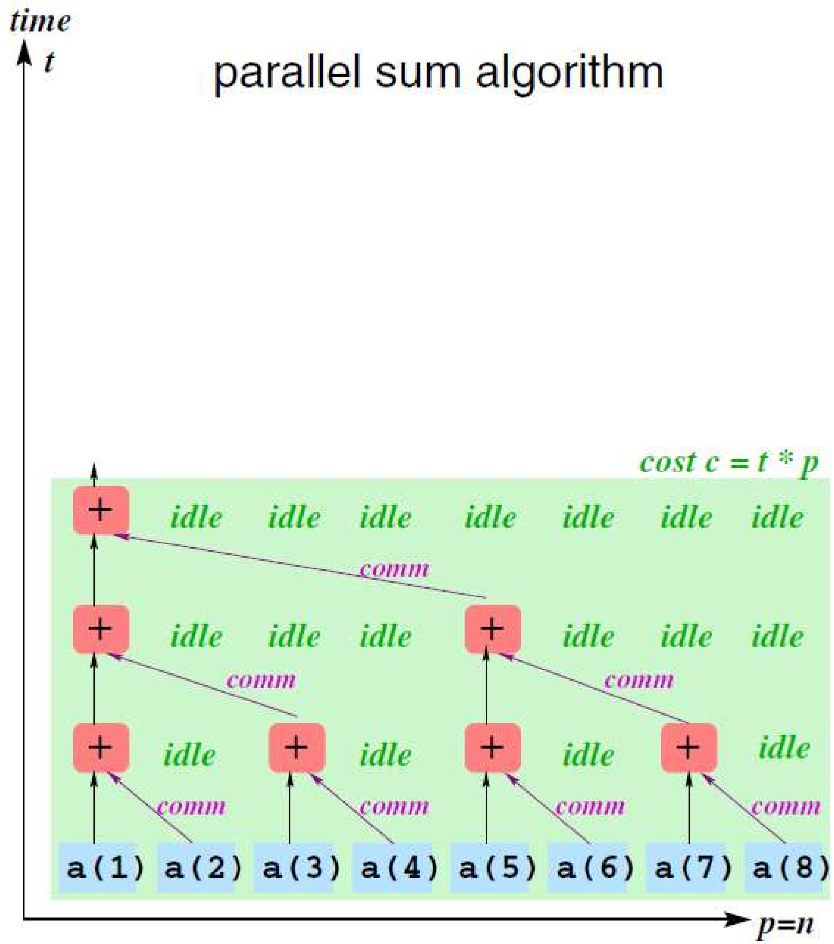
$n-1$ additions
n loads
$O(n)$ other



$$t(1,n) = t_{seq}(n) = O(n)$$

$$w(1,n) = O(n)$$

$$c(1,n) = t(1,n) \cdot 1 = O(n)$$



$$t(n,n) = O(\log n)$$

$$w(n,n) = O(n)$$

$$c(n,n) = O(n \log n)$$

par. sum alg. *not* cost-effective!

Parallel work, time, cost

parallel work $w_A(n)$ of algorithm A on an input of size n
= max. number of instructions performed by all procs during execution of A, where in each (parallel) time step as many processors are available as needed to execute the step in constant time.

parallel time $t_A(n)$ of algorithm A on input of size n
= max. number of parallel time steps required under the same circumstances

parallel cost $c_A(n) = t_A(n) * p_A(n)$ $\rightarrow c_A(n) \geq w_A(n)$
where $p_A(n) = \max_i p_i(n)$ = max. number of processors used in a step of A

Work, time, cost are thus *worst-case* measures.

$t_A(n)$ is sometimes called the depth of A
(cf. circuit model of (parallel) computation)

$p_i(n)$ = number of processors needed in time step i, $0 \leq i < t_A(n)$,
to execute the step in constant time. Then, $w_A(n) = \sum_{i=0}^{t_A(n)} p_i(n)$

Work-optimal and cost-optimal

A parallel algorithm A is asymptotically work-optimal iff $w_A(p,n) = O(t_{seq}(n))$

A parallel algorithm A is asymptotically cost-optimal iff $c_A(p,n) = O(t_{seq}(n))$

Making the parallel sum algorithm cost-optimal:

Instead of n processors, use only $n/\log_2 n$ processors.

First, each processor computes sequentially the global sum of "its" $\log n$ local elements. This takes time $O(\log n)$.

Then, they compute the global sum of $n/\log n$ partial sums using the previous parallel sum algorithm.

Time: $O(\log n)$ for local summation, $O(\log n)$ for global summation

Cost: $n/\log n \cdot O(\log n) = O(n)$ *linear!*

→ TDDD56

This is an example of a more general technique based on Brent's theorem.

Speedup

Consider problem $\mathcal{P}$, parallel algorithm A for $\mathcal{P}$

T_s = time to execute the best serial algorithm for $\mathcal{P}$
on one processor of the parallel machine

$T(1)$ = time to execute parallel algorithm A on 1 processor

$T(p)$ = time to execute parallel algorithm A on p processors

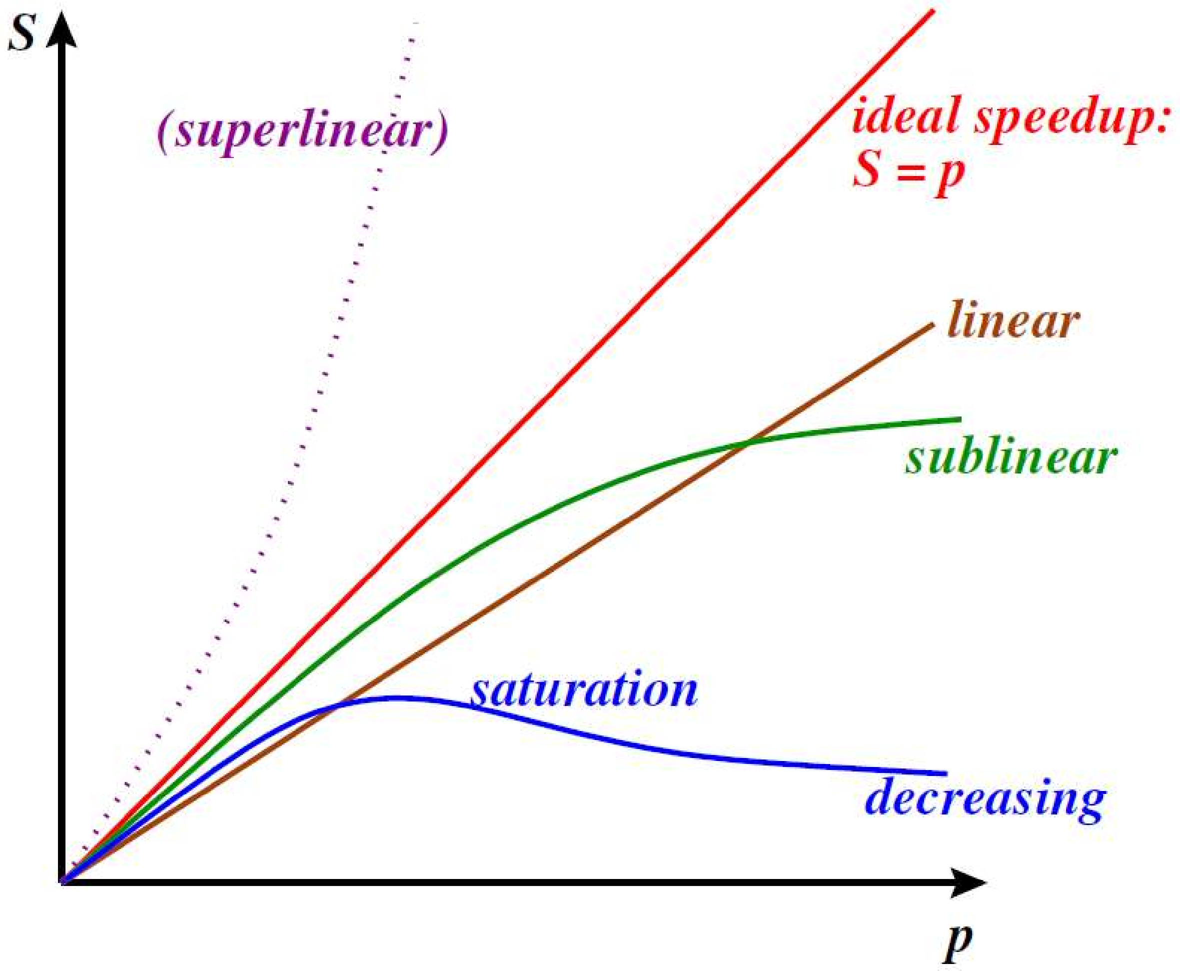
Absolute speedup $S_{abs} = \frac{T_s}{T(p)}$

Relative speedup $S_{rel} = \frac{T(1)}{T(p)}$

$S_{abs} \leq S_{rel}$

LINKÖPING UNIVERSITY

Speedup



trivially parallel
(e.g., matrix product, LU decomposition, ray tracing)
$\rightarrow$ close to ideal $S = p$

work-bound algorithms
$\rightarrow$ linear $S \in \Theta(p)$, work-optimal

tree-like task graphs
(e.g., global sum / max)
$\rightarrow$ sublinear $S \in \Theta(p/\log p)$

communication-bound
$\rightarrow$ sublinear $S = 1/fn(p)$

Most papers on parallelization show only relative speedup

(as $S_{abs} \leq S_{rel}$, and best seq. algorithm needed for S_{abs})

Amdahl's Law: Upper bound on Speedup

Consider execution (trace) of parallel algorithm A:

sequential part A^s where only 1 processor is active

parallel part A^p that can be sped up perfectly by p processors

$\rightarrow$ total work $w_A(n) = w_{A^s}(n) + w_{A^p}(n)$, time $T = T_{A^s} + \frac{T_{A^p}}{p}$,

Amdahl's Law

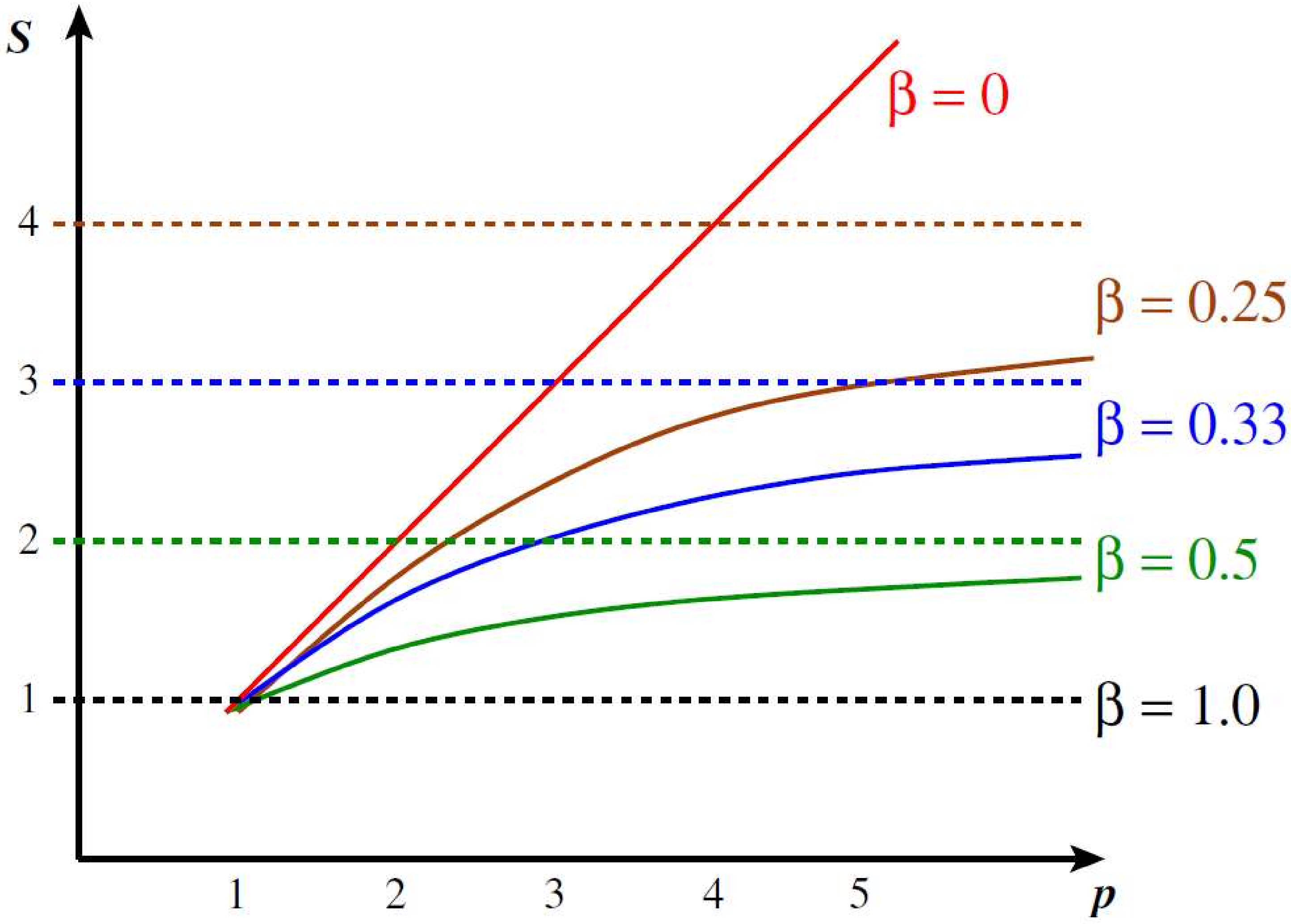
If the sequential part of A is a *fixed* fraction of the total work irrespective of the problem size n, that is, if there is a constant β with

$$\beta = \frac{w_{A^s}(n)}{w_A(n)} \leq 1$$

the relative speedup of A with p processors is limited by

$$\frac{p}{\beta p + (1 - \beta)} < 1/\beta$$

Amdahl's Law



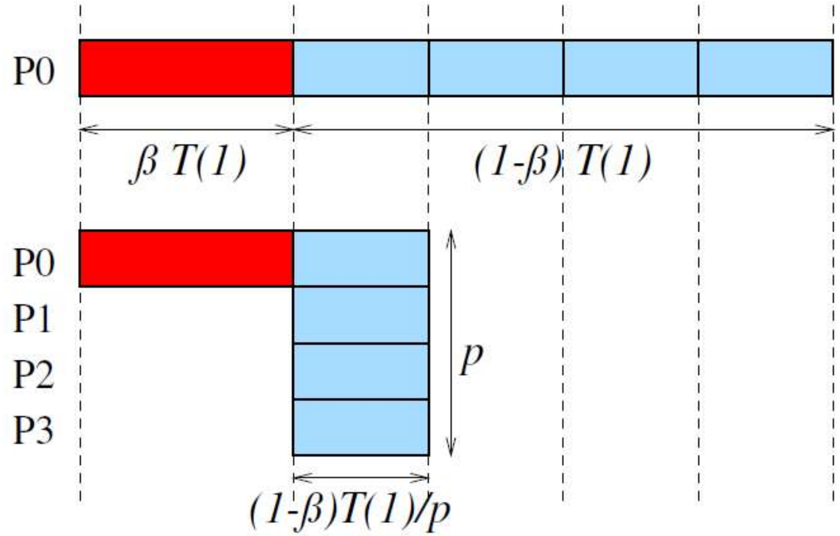
$$S(p) = \frac{p}{\beta p + (1-\beta)} < 1/\beta$$

Proof of Amdahl's Law

$$S_{rel} = \frac{T(1)}{T(p)} = \frac{T(1)}{T_{A^s} + T_{A^p}(p)}$$

Assume perfect parallelizability of the parallel part A^p,
that is, $T_{A^p}(p) = (1-\beta)T(p) = (1-\beta)T(1)/p$:

$$S_{rel} = \frac{T(1)}{\beta T(1) + (1-\beta)T(1)/p} = \frac{p}{\beta p + 1 - \beta} \leq 1/\beta$$



Remark:

C. For most parallel algorithms the sequential part is *not* a fixed fraction.

Remarks on Amdahl's Law

Not limited to speedup by parallelization only!

Can also be applied with other optimizations

e.g. SIMDization, instruction scheduling, data locality improvements, ...

Amdahl's Law, general formulation:

If you speed up a fraction $(1-\beta)$ of a computation by a factor p,

the overall speedup is $\frac{p}{\beta p + (1-\beta)}$, which is $< \frac{1}{\beta}$.

Implications

- Optimize for the common case.
  If $1-\beta$ is small, optimization has little effect.
- Ignored optimization opportunities (also) limit the speedup.
  As $p \longrightarrow \infty$, speedup is bound by $\frac{1}{\beta}$.

Speedup Anomalies

Speedup anomaly:

An implementation on p processors may execute faster than expected.

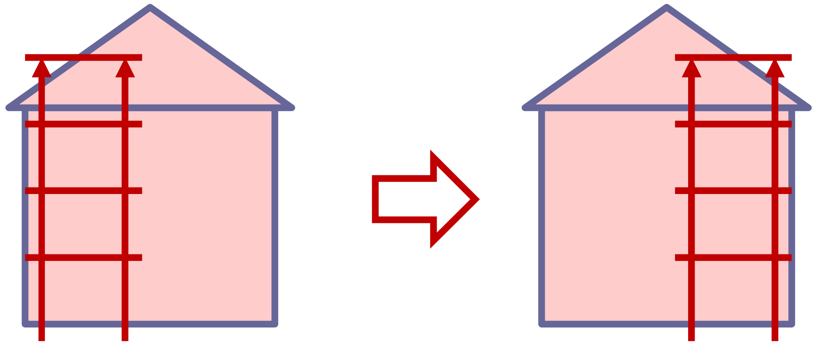
Superlinear speedup

speedup function that grows faster than linear, i.e., in $\Omega(p)$

Possible causes:

- cache effects
- search anomalies

Real-world example: move scaffolding



Speedup anomalies may occur only for fixed (small) range of p.

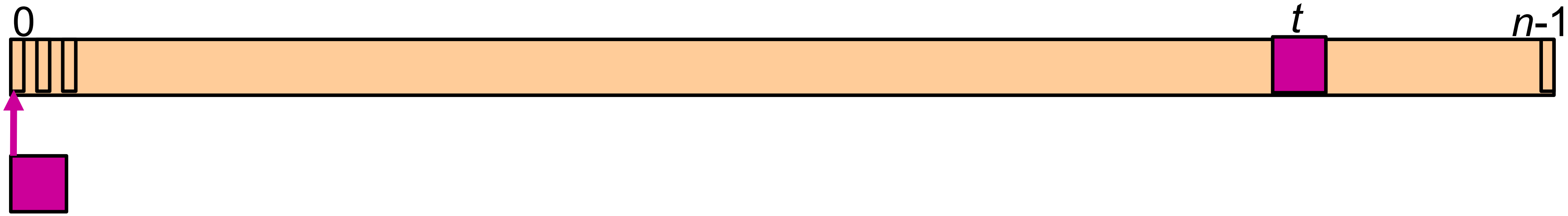
Theorem:

There is no absolute superlinear speedup for arbitrarily large p.

Search Anomaly Example: Simple string search

Given: Large unknown string of length *n*,
pattern of constant length $m << n$
Search for *any* occurrence of the pattern in the string.

Simple sequential algorithm: Linear search



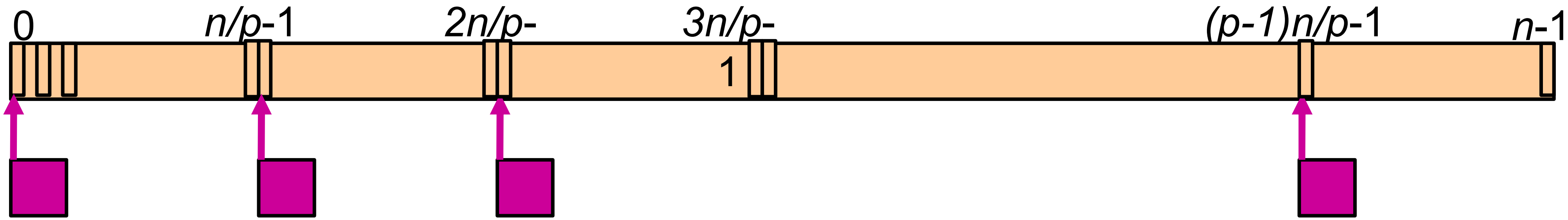
Pattern found at first occurrence at position *t* in the string after *t* time steps or not found after *n* steps

Parallel Simple string search

Given: Large unknown shared string of length *n*,
pattern of constant length *m* << *n*

Search for *any* occurrence of the pattern in the string.

Simple parallel algorithm: Contiguous partitions, linear search



Case 1: Pattern not found in the string

→ measured parallel time n/p steps

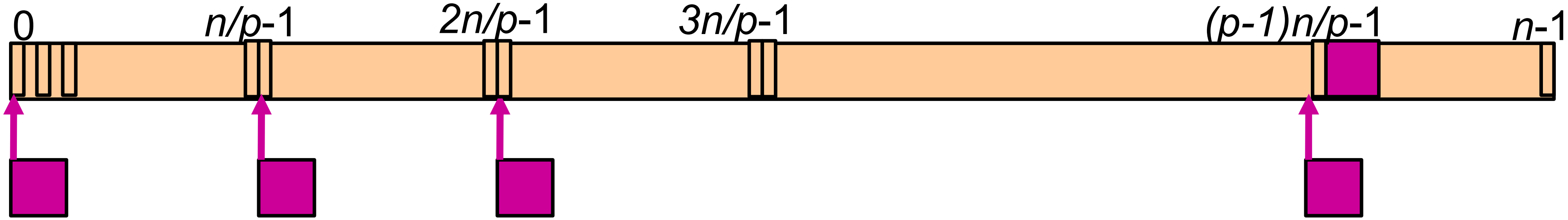
→ speedup = n / (n/p) = p ☺

Parallel Simple string search

Given: Large unknown shared string of length *n*,
pattern of constant length *m* << *n*

Search for *any* occurrence of the pattern in the string.

Simple parallel algorithm: Contiguous partitions, linear search

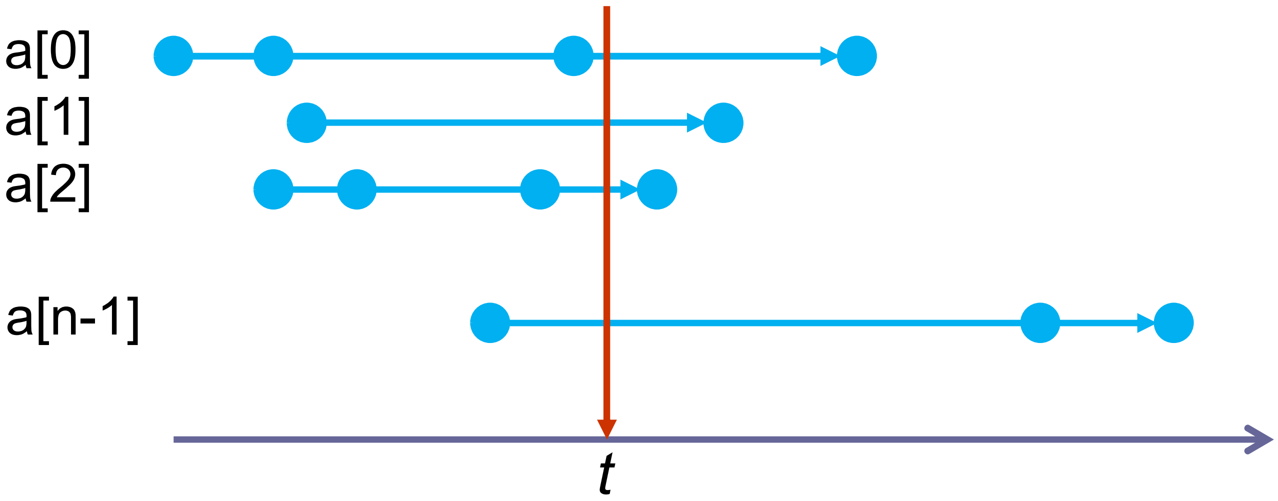


Case 2: Pattern found in the first position scanned by the last processor
→ measured parallel time 1 step, sequential time *n-n/p* steps
→ observed speedup *n-n/p*, "superlinear" speedup?!?

But, …
… this is not the worst case (but the best case) for the parallel algorithm;
… and we could have achieved the same effect in the sequential algorithm, too, by altering the string traversal order

Simple Analysis of Cache Impact

- Call a variable (e.g. array element) ***live***
  between its first and its last access in an algorithm's execution
  - Focus on the large data structures of an algorithm (e.g. arrays)
- **Working set** of algorithm A at time t
  $WS_A(t) = \{ v\text{: variable } v \text{ live at } t \}$
- Worst-case **working set size** / **working space** of A
  $WSS_A = \max_t | WS_A(t) |$
- Average-case working set size / working space of A
  … $= \mathrm{avg}_t | WS_A(t) |$



Simple Analysis of Cache Impact

- Call a variable (e.g. array element) ***live***
  between its first and its last access in an algorithm's execution
  - Focus on the large data structures of an algorithm (e.g. arrays)
- **Working set** of algorithm A at time t
  $WS_A(t) = \{ v\text{: variable } v \text{ live at } t \}$
- Worst-case **working set size** / **working space** of A
  $WSS_A = \max_t | WS_A(t) |$
- Average-case working set size / working space of A
  … $= \text{avg}_t | WS_A(t) |$
- **Rule of thumb**: Algorithm A has good (last-level) cache locality
  if $WSS_A < 0.9 * SizeOfLastLevelCache$
  - Assuming a fully associative cache with perfect LRU impl.
  - Impact of the cache line size not modeled
  - 10% reserve for some "small" data items
    (current instructions, loop variables, stack frame contents, …)

☺ Allows realistic performance prediction for simple regular algorithms

☹ Hard to analyze *WSS* for complex, irregular algorithms

Further fundamental parallel algorithms

Parallel prefix sums

Parallel list ranking

… as time permits …

Christoph Kessler, IDA,
Linköping University

Data-Parallel Algorithms

- One task (virtual processor) associated with each data element
  Agglomeration + mapping to hardware processors by the compiler

- Problems of size N
  solved usually in time $O(1)$ or $O(\log N)$ using N processors

Some data-parallel algorithms

- Parallel sum $\surd$
- Prefix sums (partial sums)
- Radix sort
- Parsing a regular language
- Parallel combinator reduction
- List ranking (finding the end of a parallel linked list, list prefix sums etc.)
- Matching components of two lists

The Prefix-Sums Problem

Given: a set S (e.g., the integers)

a binary associative operator $\oplus$ on S,

a sequence of n items $x_0, \ldots, x_{n-1} \in S$

compute the sequence y of *prefix sums* defined by

$$y_i = \bigoplus_{j=0}^{i} x_j \text{ for } 0 \le i < n$$

An important building block of many parallel algorithms! [Blelloch'89]

typical operations $\oplus$:

integer addition, maximum, bitwise AND, bitwise OR

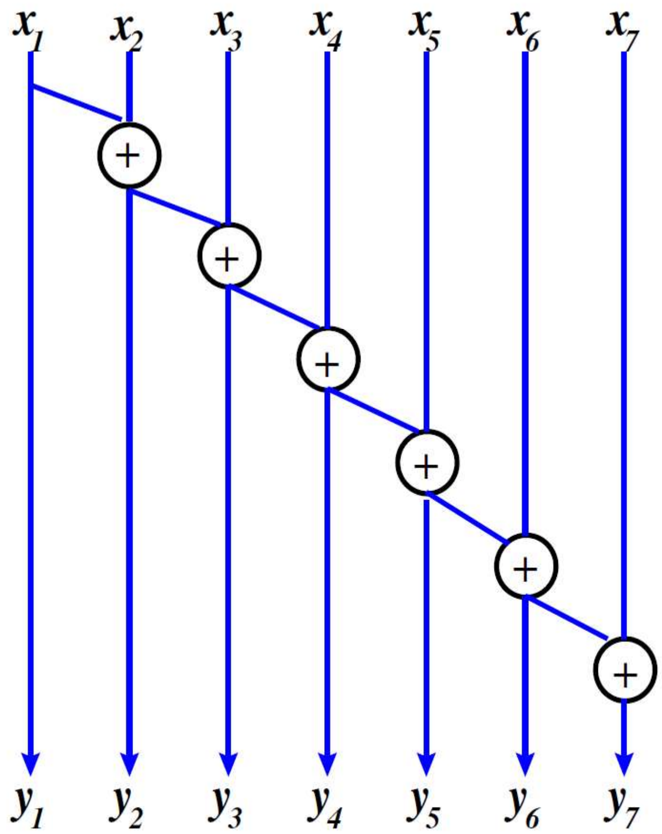
Example:

bank account: initially 0$, daily changes x_0, x_1, ...

$\rightarrow$ compute daily balances: (0,) x_0, $x_0 + x_1$, $x_0 + x_1 + x_2$, ...

Sequential prefix sums algorithm

```
void seq_prefix( int x[], int n, int y[] )
{
  int i;
  int ps;  // i'th prefix sum
  if (n>0) ps = y[0] = x[0];
  for (i=1; i<n; i++) {
    ps += x[i];
    y[i] = ps;
  }
}
```



if run in parallel on n virtual processors:

time $\Theta(n)$, work $\Theta(n)$, cost $\Theta(n^2)$

Task dependence graph: linear chain of dependences

$\rightarrow$ seems to be inherently sequential — how to parallelize?

Parallel prefix sums algorithm 1
A first attempt…

Naive parallel implementation:

apply the definition,

$$y_i = \bigoplus_{j=0}^{i} x_j \text{ for } 0 \leq i < n$$

and assign one processor for computing each y_i

$\rightarrow$ parallel time $\Theta(n)$, work and cost $\Theta(n^2)$

But we observe:

a lot of redundant computation (common subexpressions)

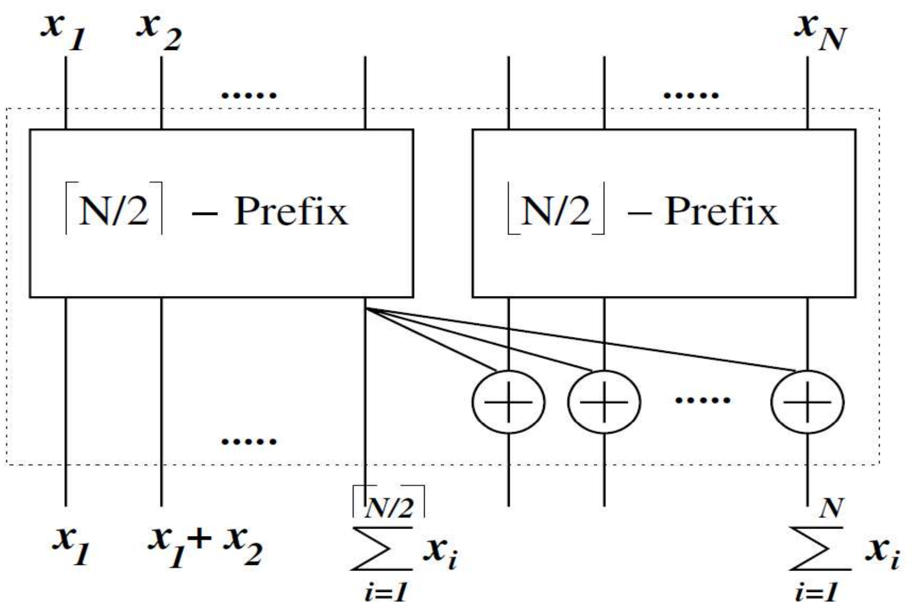
Parallel Prefix Sums Algorithm 2:
Upper-Lower Parallel Prefix

Algorithmic technique: parallel divide&conquer

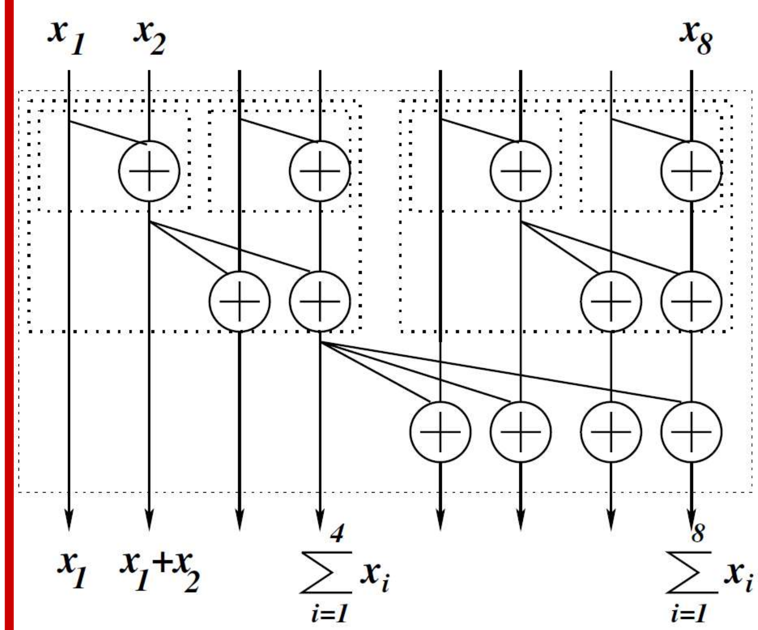
We consider the simplest variant, called Upper/lower parallel prefix:

recursive formulation:

N–prefix is computed as



Upper/lower parallel prefix, unfolded for $N = 8$:



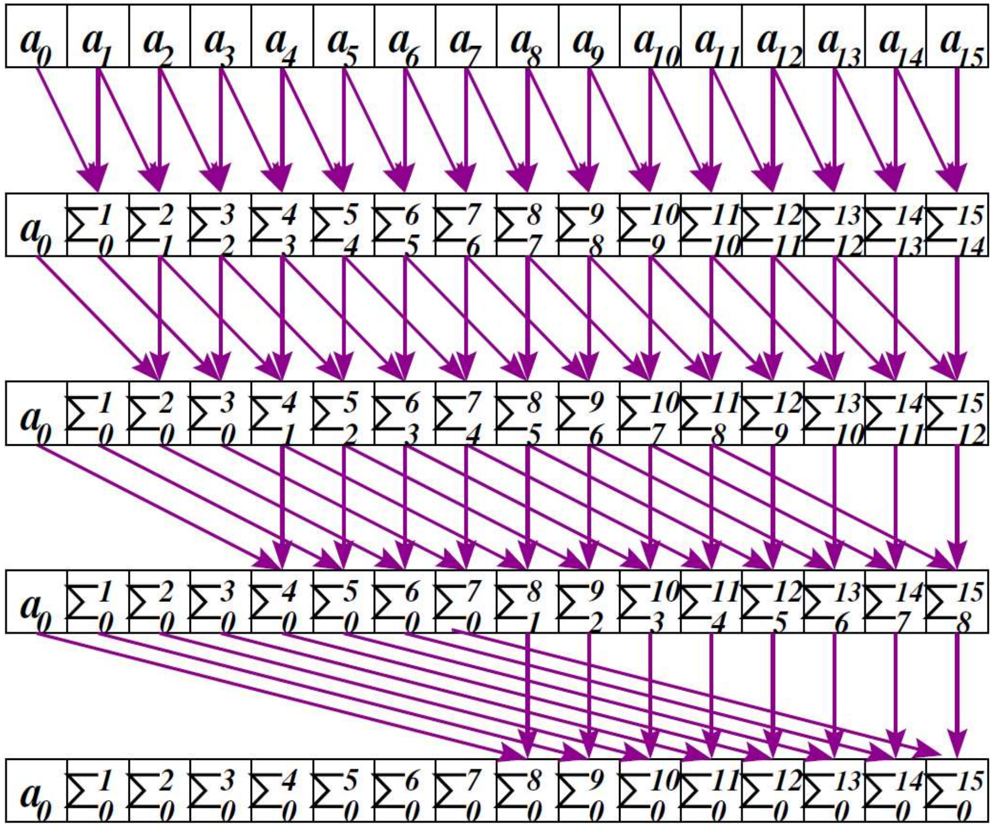
Parallel time: $\log n$ steps, work: $n/2 \; \log n$ additions, cost: $\Theta(n \log n)$

c. Not work-optimal! And needs concurrent read...

Parallel Prefix Sums Algorithm 3:
Recursive Doubling (for EREW PRAM)

[Hillis, Steele '86]

EREW (exclusive read, exclusive write) prefix sums algorithm:



Work: $\Theta(n \log n)$:-(

iterative formulation
in data-parallel pseudocode:

real a : **array**$[0..N-1]$;
int stride;

stride $\leftarrow$ 1;
while stride $< N$ **do**

  forall i : $[0..N-1]$ **in parallel do**

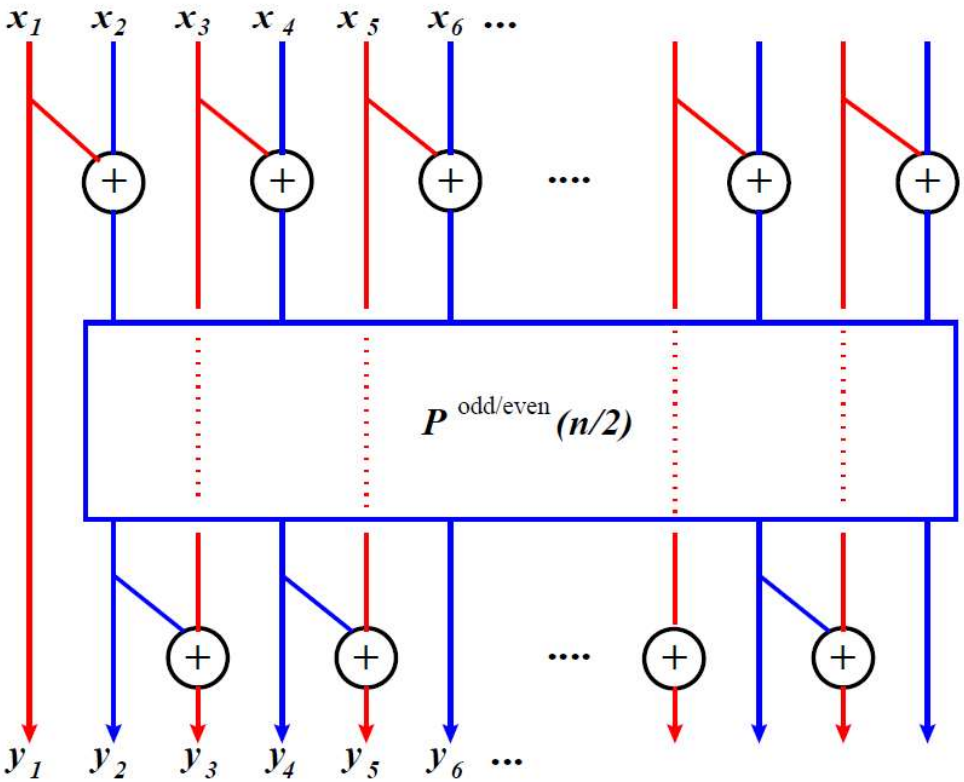
    if $i \geq$ stride **then**
      $a[i] \leftarrow a[i-\text{stride}] + a[i]$;

  stride := stride * 2;

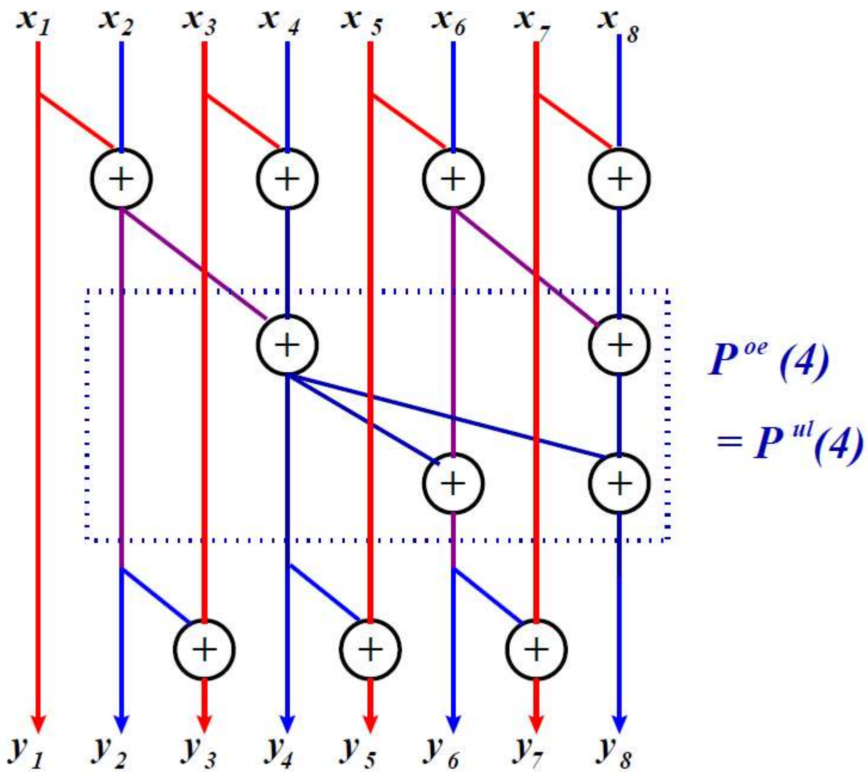
(* finally, sum in $a[N-1]$ *)

Parallel Prefix Sums Algorithm 4:
Odd-Even Parallel Prefix

Recursive definition: $P^{oe}(n)$:



Example: $P^{oe}(8)$ with
base case $P^{oe}(4)$



EREW, $2\log n - 2$ time steps, work $2n - \log n - 2$, cost $\Theta(n \log n)$

Not cost-optimal! But may use Brent's theorem...

Parallel Prefix Sums Algorithm 5
Ladner-Fischer Parallel Prefix Sums (1980)

Odd-Even Parallel Prefix Sums algorithm
after work-time rescheduling:

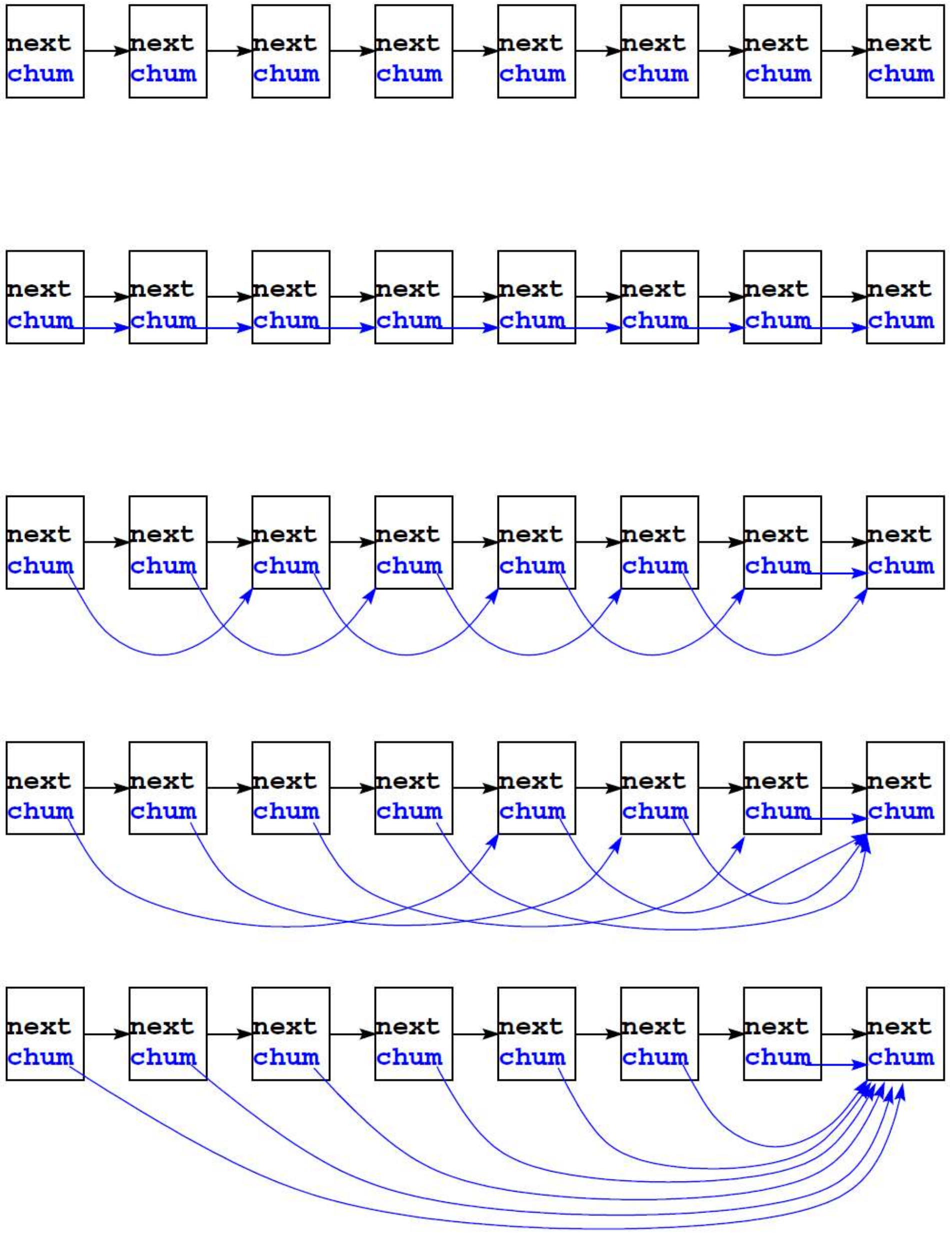
cost-optimal (cost $\Theta(n)$) if using $\Theta(n/\log n)$ virtual processors only

Parallel List Ranking (1)

Parallel list: (unordered) array of list items (one per proc.), singly linked

Problem: for each element, find the end of its linked list.

Algorithmic technique: recursive doubling, here: "pointer jumping" [Wyllie'79]



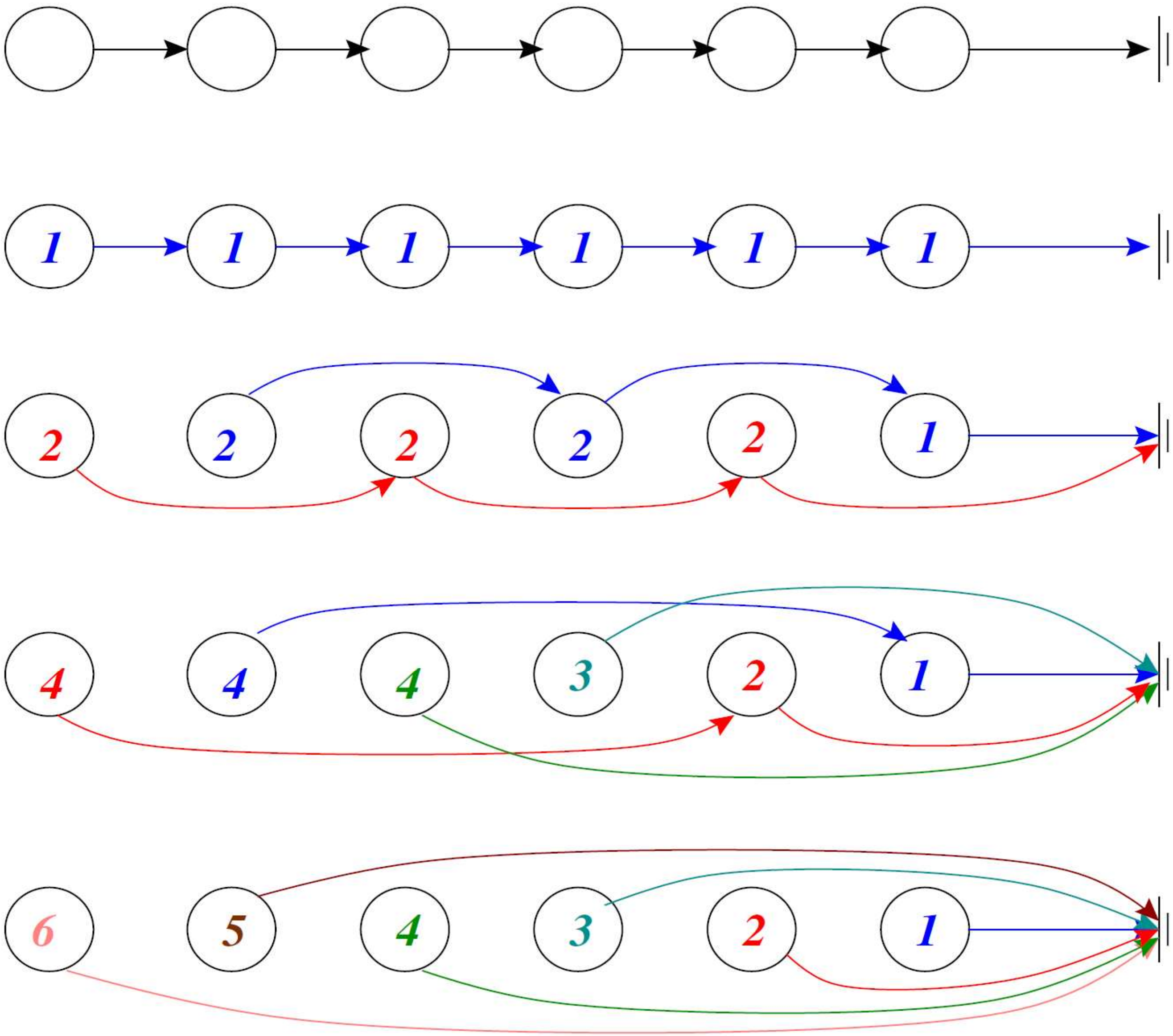
The algorithm in pseudocode:

forall k in $[1..N]$ **in parallel do**
    chum$[k] \leftarrow$ next$[k]$;
    while chum$[k] \neq$ null
            and chum$[$chum$[k]] \neq$ null **do**
        chum$[k] \leftarrow$ chum$[$chum$[k]]$;
    od
od

lengths of chum lists halved in each step
$\Rightarrow \lceil \log N \rceil$ pointer jumping steps

Parallel List Ranking (2)

Extended problem: compute the rank = distance to the end of the list



By pointer jumping:

in each step:

to my own
distance value,
I add the distance
of my →chum
that I splice
out of the list

Every step
+ doubles #lists
+ halves lengths

→ $\lceil \log_2 n \rceil$ steps

Not work-efficient!

Parallel List Ranking (3)

NULL-checks can be avoided by marking list end by a self-loop.

Pointer jumping algorithm for list ranking, implementation in Fork:

```
wyllie( sh LIST list[], sh int length )
{
  LIST *e; // private pointer
  int nn;

  e = list[$$];    // $$ is my processor index
  if (e->next != e)  e->rank = 1;  else e->rank = 0;
  nn = length;
  while (nn>1) {
    e->rank = e->rank + e->next->rank;
    e->next = e->next->next;
    nn = nn>>1;  // division by 2
  }
}
```

Summary

Parallel computation model = programming model + performance model

$\rightarrow$ quantitative basis for design and analysis of parallel algorithms

Use simple performance models (PRAM, Delay, BSP)
early in the design process.

Refine performance model at later stages (BSP, LogP, LogGP)
and conduct simple experiments to derive model parameters

During implementation, compare performance to predictions by the model
$\rightarrow$ may identify implementation errors and improve quality.

Questions?

Christoph Kessler, IDA,
Linköping University

Further Reading

See the TDDE65/TDDD56 Compendium!

Also:

C. Kessler, J. Keller: Models for Parallel Computing: Review and Perspectives. *PARS-Mitteilungen* **24**, Gesellschaft für Informatik, Dec. 2007, ISSN 0177-0454.

On PRAM model and Design and Analysis of Parallel Algorithms

- J. Keller, C. Kessler, J. Träff: ***Practical PRAM Programming.*** Wiley Interscience, New York, 2001.
- J. JaJa: ***An introduction to parallel algorithms.*** Addison-Wesley, 1992.
- D. Cormen, C. Leiserson, R. Rivest: ***Introduction to Algorithms,*** Chapter 30. MIT press, 1989.
- H. Jordan, G. Alaghband: ***Fundamentals of Parallel Processing.*** Prentice Hall, 2003.
- W. Hillis, G. Steele: Data parallel algorithms. *Comm. ACM* **29**(12), Dec. 1986. Link on TDDC78 / TDDD56 course homepage.