Foundations of parallel algorithms

PRAM model

Time, work, cost

Self-simulation and Brent's Theorem

Speedup and Amdahl's Law

NC

Scalability and Gustafssons Law

Fundamental PRAM algorithms
- reduction
- parallel prefix
- list ranking

PRAM variants, simulation results and separation theorems.

Survey of other models of parallel computation
- Asynchronous PRAM, Delay model, BSP, LogP, LogGP

Literature

[PPP] Keller, Kessler, Träff: ***Practical PRAM Programming.*** Wiley Interscience, New York, 2000. Chapter 2.

[JaJa] JaJa: ***An introduction to parallel algorithms.*** Addison-Wesley, 1992.

[CLR] Cormen, Leiserson, Rivest: ***Introduction to Algorithms***, Chapter 30. MIT press, 1989.

[JA] Jordan, Alaghband: ***Fundamentals of Parallel Processing.*** Prentice Hall, 2003.

Parallel computation models (1)

+ 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

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

Programming model

shared memory vs.
message passing

degree of synchronous execution

Cost model

key parameters

cost functions for basic operations

constraints

Parallel computation models (2)

Cost model: should

+ explain available observations

+ predict future behaviour

+ abstract from unimportant details → generalization

Simplifications to reduce model complexity:

use idealized machine model
- ignore hardware details: memory hierarchies, network topology, ...

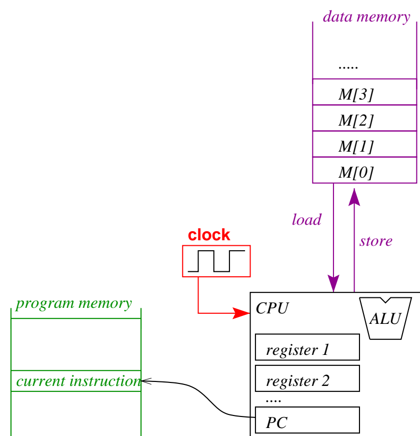
use asymptotic analysis
- drop insignificant effects

use empirical studies
- calibrate parameters, evaluate model

Flashback to DALG, Lecture 1: The RAM model

RAM (Random Access Machine) [PPP 2.1]

programming and cost model for the analysis of sequential algorithms



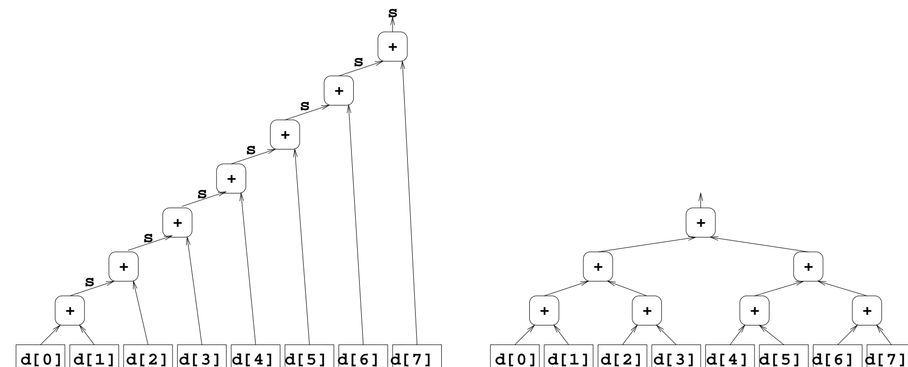
The RAM model (2)

Algorithm analysis: Counting instructions

Example: Computing the global sum of N elements

```
s = d(0)
do i = 1, N-1
   s = s + d(i)
end do
```

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



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

PRAM model [PPP 2.2]

Parallel Random Access Machine [Fortune/Wyllie'78]

p processors

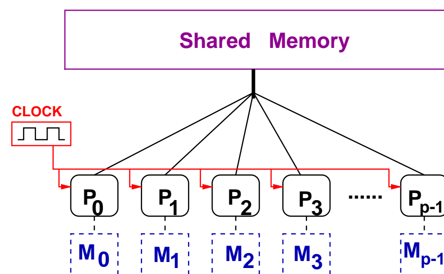
- MIMD
- common clock signal
- arithm./jump: 1 clock cycle

shared memory

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

private memory (optional)

- processor-local access only



PRAM model: Variants for memory access conflict resolution

Exclusive Read, Exclusive Write (EREW) PRAM

concurrent access only to different locations in the same cycle

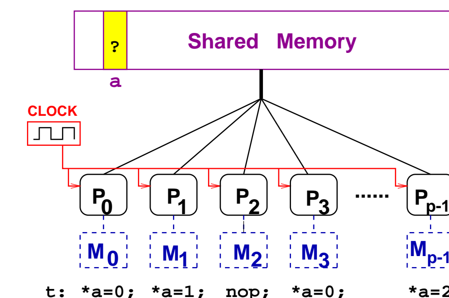
Concurrent Read, Exclusive Write (CREW) PRAM

simultaneous reading from *or* single 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.)



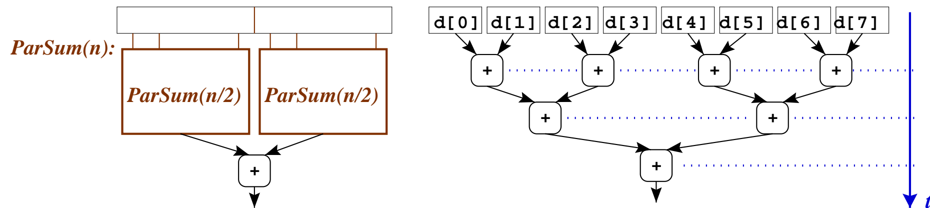
No need for ERCW ...

Global sum computation on EREW and Combining-CRCW PRAM (1)

Given n numbers $x_0, x_1, ..., x_{n-1}$ 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 $O(1)$

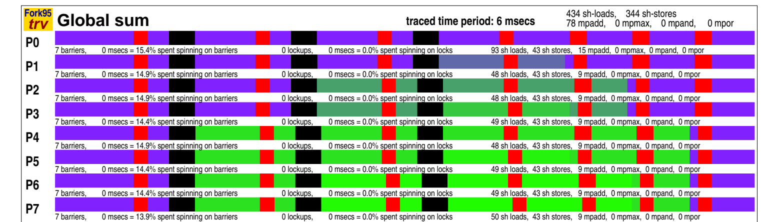
$\rightarrow \quad T(n) = T(n/2) + O(1)$

Use induction or the master theorem [CLR 4] $\rightarrow T(n) \in O(\log n)$

Global sum computation on EREW and Combining-CRCW PRAM (2)

Recursive parallel sum program in the PRAM progr. language Fork [PPP]

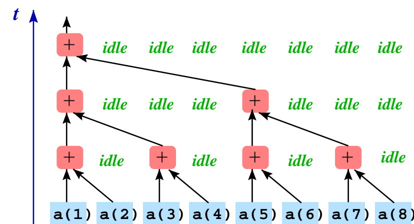
```
sync int parsum( sh int *d, sh int n)
{
  sh int s1, s2;
  sh int nd2 = n / 2;
  if (n==1) return d[0]; // base case
  $=rerank();          // re-rank processors within group
  if ($<nd2)           // split processor group:
     s1 = parsum( d, nd2 );
  else                     s2 = parsum( &(d[nd2]), n-nd2 );
  return s1 + s2;
}
```



Global sum computation on EREW and Combining-CRCW PRAM (3)

Iterative parallel sum program in Fork

```
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];
  }
}
```



On a Combining CRCW PRAM with addition as the combining operation, the global sum problem can be solved in a constant number of time steps using n processors.

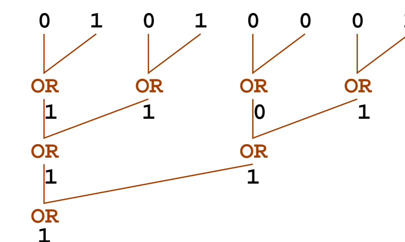
```
 syncadd( &s, a[ID] );   // procs ranked ID in 0...n-1
```

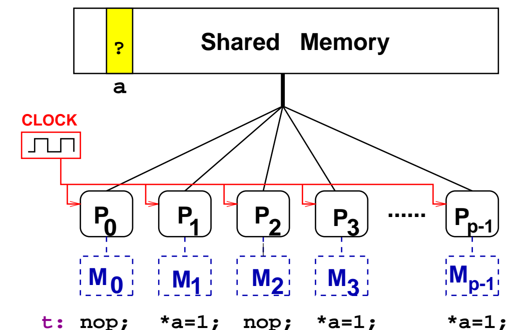

PRAM model: CRCW is stronger than CREW

Example:

Computing the logical OR of p bits

CREW: *time O(log p)*





CRCW: *time O(1)*

```
sh int a = 0;
if (mybit == 1)   a = 1;
```
(else do nothing)

e.g. for termination detection

Analysis of parallel algorithms

(a) asymptotic analysis

$\rightarrow$ estimation based on model and pseudocode operations
$\rightarrow$ results for large problem sizes, large # processors

(b) empirical analysis

$\rightarrow$ measurements based on implementation
$\rightarrow$ for fixed (small) problem and machine sizes

Asymptotic analysis: Work and Time

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
= maximum number of parallel time steps required under the same circumstances.

Work and time are thus *worst-case* measures.

$t_A(n)$ is sometimes called the depth of A
(cf. circuit model, DAG 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, $w_A(n) = \sum_{i=0}^{t_A(n)} p_i(n)$

Asymptotic analysis: Work and time optimality, work efficiency

A is work-optimal if $w_A(n) = O(t_S(n))$

where S = optimal or currently best known sequential algorithm
for the same problem

A is work-efficient if $w_A(n) = t_S(n) \cdot O(\log^k(t_S(n)))$ for some constant $k \geq 1$.

A is time-optimal if any other parallel algorithm for this problem
requires $\Omega(t_A(n))$ time steps.

Asymptotic analysis: Cost, cost optimality

Algorithm A needs $p_A(n) = \max_{1 \leq i \leq t_A(n)} p_i(n)$ processors.

Cost $c_A(n)$ of A on an input of size n
= processor-time product: $c_A(n) = p_A(n) \cdot t_A(n)$

A is cost-optimal if $c_A(n) = O(t_S(n))$
with S = optimal or currently best known sequential algorithm
for the same problem

Work $\leq$ Cost: $w_A(n) = O(c_A(n))$

A is cost-effective if $w_A(n) = \Theta(c_A(n))$.

Asymptotic analysis for global sum computation

problem size n

processors p

time $t(p,n)$

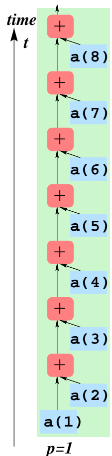
work $w(p,n)$

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

Example:
seq. sum algorithm

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

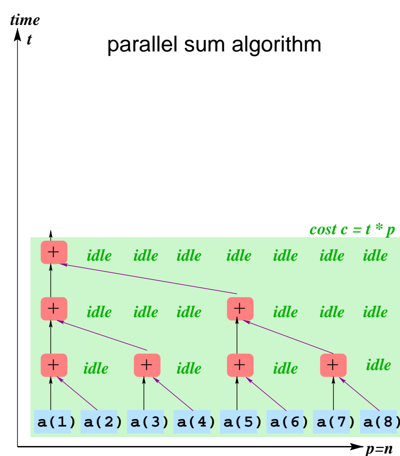
$n-1$ additions
n loads
$O(n)$ other



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

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

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



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

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

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

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

Trading concurrency for cost-effectiveness

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!

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

Self-simulation and Brent's Theorem

Self-simulation (aka work-time scheduling in [JaJa'92])

A model of parallel computation is self-simulating
if a p-processor machine can simulate
one time step of a q-processor machine in $O(\lceil q/p \rceil)$ time steps.

All PRAM variants are self-simulating.

Proof idea for (EREW) PRAM with $p \leq q$ simulating processors:

Divide the q simulated processors in p chunks of size $\leq \lceil q/p \rceil$
assign a chunk to each of the p simulating processors
map memory of simulated PRAM to memory of simulating PRAM
step-by-step simulation, with $O(q/p)$ steps per simulated step
take care of pending memory accesses in current simulated step
extra space $O(q/p)$ for registers and status of the simulated machine

Consequences of self-simulation

RAM = 1-processor PRAM simulates p-processor PRAM in $O(p)$ time steps.

$\rightarrow$ RAM simulates A with cost $c_A(n) = p_A(n) t_A(n)$ in $O(c_A(n))$ time.

(Actually possible in $O(w_A(n))$ time.)

Even with arb. many processors A cannot be simulated any faster than $t_A(n)$.

For cost-optimal A, $c_A(n) = \Theta(t_S(n))$ $\rightarrow$ Exercise

p-processor PRAM can simulate one step of A requiring $p_A(n)$ processors in $O(p_A(n)/p)$ time steps

Self-simulation emulates virtual processors with significant overhead.

In practice, other mechanisms for adapting the granularity are more suitable.

How to avoid simulation of inactive processors where $c_A(n) = \omega(w_A(n))$?

Brent's Theorem

Brent's theorem: [Brent'74]

Any PRAM algorithm A
which runs in $t_A(n)$ time steps and performs $w_A(n)$ work
can be implemented to run on a p-processor PRAM in

$$O\left(t_A(n) + \frac{w_A(n)}{p}\right)$$

time steps.

Proof: see [PPP p.41]

Algorithm design issue: Balance the terms for cost-effectiveness:
$\rightarrow$ design A with $p_A(n)$ processors such that $w_A(n)/p_A(n) = O(t_A(n))$

Note: Proof is non-constructive!
$\rightarrow$ How to determine the active processors for each time step?
$\rightarrow$ language constructs, dependence analysis, static/dynamic scheduling, ...

Absolute Speedup

A parallel algorithm for problem P
S asymptotically optimal or best known sequential algorithm for P.
$t_A(p,n)$ worst-case execution time of A with $p \leq p_A(n)$ processors
$t_S(n)$ worst-case execution time of S

The absolute speedup of a parallel algorithm A is the ratio

$$\mathrm{SU}_{\mathrm{abs}}(p,n) = \frac{t_S(n)}{t_A(p,n)}$$

If S is an optimal algorithm for P, then $\mathrm{SU}_{\mathrm{abs}}(p,n) = \frac{t_S(n)}{t_A(p,n)} \leq p\frac{t_S(n)}{c_A(n)} \leq p$
for any fixed input size n, since $t_S(n) \leq c_A(n)$.

A cost-optimal parallel algorithm A for a problem P has linear absolute speedup.

This holds for n sufficiently large.
"Superlinear" speedup $> p$ may exist only for small n.

Relative Speedup and Efficiency

Compare A with p processors to itself running on 1 processor:

The asymptotic relative speedup of a parallel algorithm A is the ratio

$$\mathrm{SU}_{\mathrm{rel}}(p,n) = \frac{t_A(1,n)}{t_A(p,n)}$$

$t_S(n) \leq t_A(1,n) \;\rightarrow\; \mathrm{SU}_{\mathrm{rel}}(p,n) \geq \mathrm{SU}_{\mathrm{abs}}(p,n)$. [PPP p.44 typo!]

Preferably used in papers on parallelization to "nice" performance results.

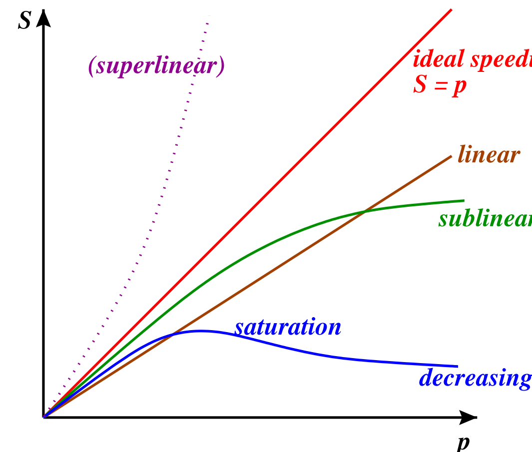
The relative efficiency of parallel algorithm A is the ratio

$$\mathrm{EF}(p,n) = \frac{t_A(1,n)}{p \cdot t_A(p,n)}$$

$\mathrm{EF}(p,n) = \mathrm{SU}_{\mathrm{rel}}(p,n)/p \quad \in [0,1]$

Speedup curves

Speedup curves measure the utility of parallel computing, not speed.



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

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

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

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

Most papers on parallelization show only relative speedup
(as $SU_{abs} \leq SU_{rel}$, and best seq. algorithm needed for SU_{abs})

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.

Amdahl's Law

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 $w_A(n) = w_{A^s}(n) + w_{A^p}(n)$

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

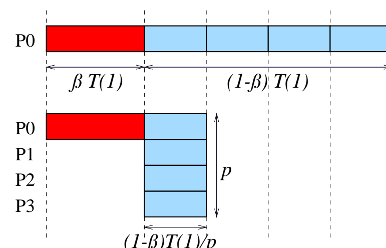
$$\frac{p}{\beta p + (1-\beta)} \leq 1/\beta$$

Proof of Amdahl's Law

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

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

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



Remark:

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

NC

Recall complexity class $\mathcal{P}$:

$\mathcal{P}$ = set of all problems solvable on a RAM in polynomial time

Can all problems in $\mathcal{P}$ be solved fast on a PRAM?

"Nick's class" $\mathcal{NC}$:

$\mathcal{NC}$ = set of problems solvable on a PRAM in
polylogarithmic time $O(\log^k n)$ for some $k \geq 0$
using only $n^{O(1)}$ processors (i. e. a polynomial number)
in the size n of the input instance.

By self-simulation: $\mathcal{NC} \subseteq \mathcal{P}$.

NC - Some remarks

Are the problems in $\mathcal{NC}$ just the well-parallelizable problems?

Counterexample: Searching for a given element in an ordered array
sequentially solvable in logarithmic time (thus in $\mathcal{NC}$)
cannot be solved significantly faster in (EREW)-parallel [PPP 2.5.2]

Are $\mathcal{NC}$-algorithms always a good choice?

Time $\log^3 n$ is faster than time $n^{1/4}$ only for ca. $n > 10^{12}$.

Is $\mathcal{NC} = \mathcal{P}$?

For some problems in $\mathcal{P}$ no polylogarithmic PRAM algorithm is known
$\rightarrow$ likely that $\mathcal{NC} \neq \mathcal{P}$
$\rightarrow$ $\mathcal{P}$-completeness [PPP p. 46]

Speedup and Efficiency w.r.t. other sequential architectures

Parallel algorithm A runs on a "real" parallel machine N
with fixed size p.

Sequential algorithm S for same problem runs on sequential machine M

Measure execution times $T_A^N(p,n)$, $T_S^M(n)$ in seconds

absolute, machine-uniform speedup of A: $\mathrm{SU}_{\mathrm{abs}}(p,n) = \frac{T_S^M(n)}{T_A^M(p,n)}$

parallelization slowdown of A: $\mathrm{SL}(n) = \frac{T_A^M(1,n)}{T_S^M(n)}$

Hence, $\mathrm{SU}_{\mathrm{abs}}(p,n) = \frac{\mathrm{SU}_{\mathrm{rel}}(p,n)}{\mathrm{SL}(n)}$

absolute, machine-nonuniform speedup $= \frac{T_S^M(n)}{T_A^N(n)}$

Used in the 1990's to disqualify parallel processing by comparing to newer superscalars

Example: Cost-optimal parallel sum algorithm on SB-PRAM

$n = 10,000$

Processors	Clock cycles	Time	$\mathrm{SU}_{\mathrm{rel}}$	$\mathrm{SU}_{\mathrm{abs}}$	EF
Sequential	460118	1.84			
1	1621738	6.49	1.00	0.28	1.00
4	408622	1.63	3.97	1.13	0.99
16	105682	0.42	15.35	4.35	0.96
64	29950	0.12	54.15	15.36	0.85
256	10996	0.04	147.48	41.84	0.58
1024	6460	0.03	251.04	71.23	0.25

$n = 100,000$

Processors	Clock cycles	Time	$\mathrm{SU}_{\mathrm{rel}}$	$\mathrm{SU}_{\mathrm{abs}}$	EF
Sequential	4600118	18.40			
1	16202152	64.81	1.00	0.28	1.00
4	4054528	16.22	4.00	1.13	1.00
16	1017844	4.07	15.92	4.52	0.99
64	258874	1.04	62.59	17.77	0.98
256	69172	0.28	234.23	66.50	0.91
1024	21868	0.09	740.91	210.36	0.72

Scalability

For machine N with $p \leq p_A(n)$,
we have $t_A(p,n) = O(c_A(n)/p)$ and thus $\mathrm{SU}_{\mathrm{abs}}(p,n) = p\frac{T_S^M(n)}{c_A^N(n)}$.

$\rightarrow$ linear speedup for cost-optimal A

$\rightarrow$ "well scalable" (in theory) in range $1 \leq p \leq p_A(n)$

$\rightarrow$ For fixed n, no further speedup beyond $p_A(n)$

For realistic problem sizes (small n, small p): often sublinear!
- communication costs (non-PRAM) may increase more than linearly in p
- sequential part may increase with p – not enough work available

$\rightarrow$ less scalable

What about scaling the problem size n with p to keep speedup?

Isoefficiency [Rao,Kumar'87]

measured efficiency of parallel algorithm A on machine M for problem size n

$$\mathrm{EF}(p,n) = \frac{T_A^M(1,n)}{p \cdot T_A^M(p,n)} = \frac{\mathrm{SU}_{\mathrm{rel}}(p,n)}{p}$$

Let A solve a problem of size n' on M with p' processors with efficiency ε.

The isoefficiency function for A is a function of p, which
expresses the *increase in problem size* required for A
to retain a given efficiency ε.

If isoefficiency-function for A linear $\rightarrow A$ well scalable

Otherwise (superlinear): A needs large increase in n to keep same efficiency.

Gustafssons Law

Revisit Amdahl's law:
assumes that sequential work A^s is a constant fraction β of total work.
$\rightarrow$ when scaling up n, $w_{A^s}(n)$ will scale linearly as well!

Gustafssons Law [Gustafsson'88]

Assuming that the sequential work is *constant* (independent of n),
given by seq. fraction α in an *unscaled* (e.g., size $n=1$ (thus $p=1$)) problem
such that $T_{A^s} = \alpha T_1(1)$, $T_{A^p} = (1-\alpha)T_1(1)$,
and that $w_{A^p}(n)$ scales linearly in n,
the *scaled speedup* for $n > 1$ is predicted by

$$SU^s_{rel}(n) = \frac{T_n(1)}{T_n(n)} = \alpha + (1-\alpha)n = n - (n-1)\alpha.$$

The seq. part is assumed to be replicated over all processors.

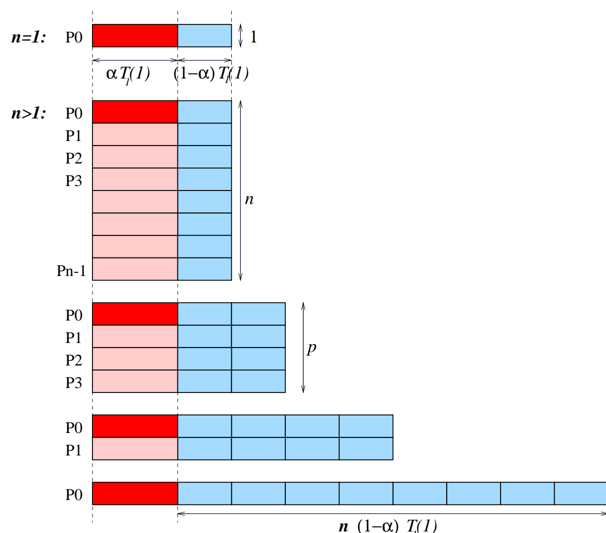
Proof of Gustafssons Law

Scaled speedup for $p = n > 1$:

$$SU^s_{rel}(n) = \frac{T_n(1)}{T_n(n)} = \frac{T_{A^s} + w_{A^p}(n)}{T_{A^s} + T_{A^p}}$$

assuming
perfect parallelizability
of A^p up to $p = n$ processors

$$SU^s_{rel}(n) = \frac{\alpha + (1-\alpha)n}{1} = n - (n-1)\alpha.$$



Yields better speedup predictions for data-parallel algorithms.

Fundamental PRAM algorithms

reduction $\surd$ see parallel sum algorithm

prefix-sums

list ranking

Oblivious (PRAM) algorithm: [JaJa 4.4.1]

control flow ($\rightarrow$ execution time) does not depend on input data.

Oblivious algorithms can be represented as arithmetic circuits
whose shape only depends on the input *size*.

Examples: reduction, (parallel) prefix, pointer jumping;
sorting networks, e.g. bitonic-sort [CLR'90 ch. 28], $\rightarrow$ Lab, mergesort

Counterexamples: (parallel) quicksort

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

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

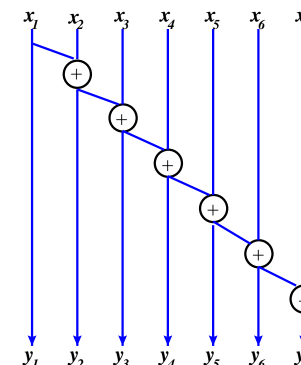
An important building block of many parallel algorithms! [Blelloch'89]

typical operations $\oplus$:
integer addition, maximum, bitwise AND, bitwise OR

Example:
bank account: initially 0$, daily changes x_0, x_1, ...
→ daily balances: (0,) x_0, $x_0 + x_1$, $x_0 + x_1 + x_2$, ...

Sequential prefix sums computation

```
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;
  }
}
```



Task dependence graph:
linear chain of dependences

→ seems to be inherently sequential — how to parallelize?

Parallel prefix sums (1)

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

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

But we observe:
a lot of redundant computation (common subexpressions)

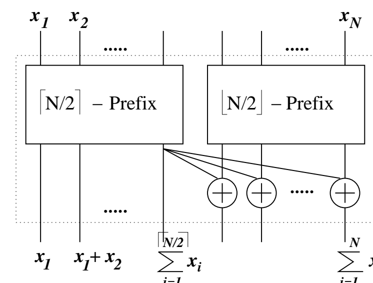
Idea: Exploit associativity of $\oplus$ ...

Parallel prefix sums (2)

Algorithmic technique: parallel divide&conquer

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

recursive formulation:
N–prefix is computed as

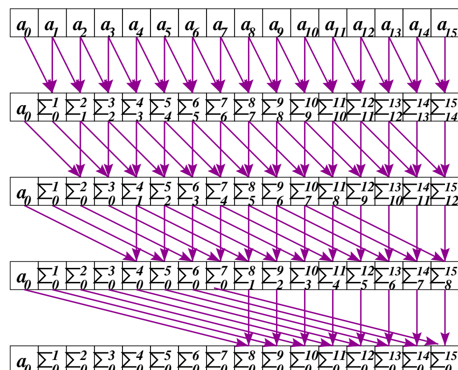


Parallel time: $\log n$ steps, work: $n/2 \log n$ additions, cost: $\Theta(n \log n)$

Not work-optimal! ... and needs concurrent read

Parallel prefix sums (3)

Rework lower-upper prefix sums algorithm for exclusive read:



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**

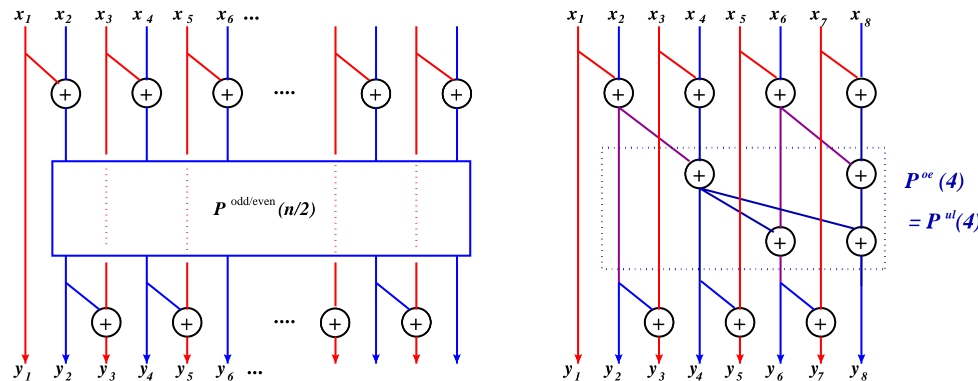
    if $i \geq$stride **then**
      $a[i] \leftarrow a[i-\text{stride}] + a[i]$;

  stride := stride * 2;

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

Parallel prefix sums (4)

Odd/even parallel prefix $P^{oddeven}(n)$:



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 (3)

Ladner/Fischer parallel prefix [Ladner/Fischer'80]

combines advantages of upper-lower and odd-even parallel prefix

EREW, time $\log n$ steps, work $4n - 4.96n^{0.69} + 1$, cost $\Theta(n \log n)$

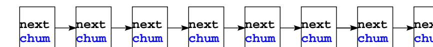
can be made cost-optimal using Brent's theorem:

The prefix-sums problem can be solved on a n-processor EREW PRAM in $\Theta(\log n)$ time steps and cost $\Theta(n)$.

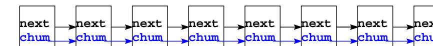
Towards List Ranking

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

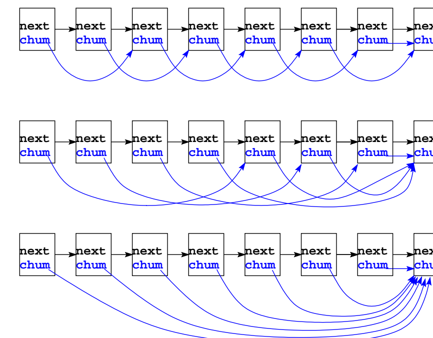
Problem: for each element, find
  the end of its linked list.



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



The algorithm in pseudocode:

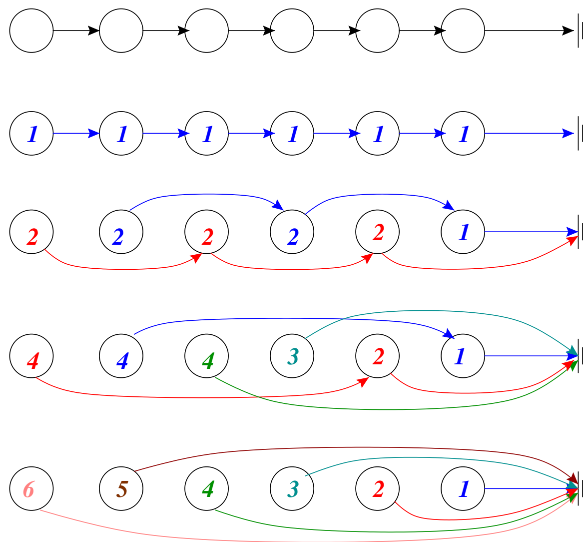


forall k in $[1..N]$ **in parallel do**
  chum$[k] \leftarrow$ next$[k]$;
  while chum$[k] \neq$ null
      and chum[chum$[k]] \neq$ null **do**
    chum$[k] \leftarrow$ chum[chum$[k]]$;
  od
od

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

List ranking

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



Pointer jumping

[Wyllie'79]

EREW

1 step:
to my own distance value, I add distance of my →next that I splice out of the list

Every step
+ doubles #lists
+ halves lengths

→ $\lceil \log_2 n \rceil$ steps

Not work-efficient!

List ranking (2): Pointer jumping

NULL-checks can be avoided by marking list end by a self-loop.
Implementation in Fork:

```
sync 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
  }
}
```

Also for parallel prefix on a list! → Exercise

CREW is more powerful than EREW

Example problem:

Given a directed forest,
compute for each node a pointer to the root of its tree.

CREW: with pointer-jumping technique in $\lceil \log_2$ max. depth$\rceil$ steps

e.g. for balanced binary tree: $O(\log \log n)$;   an $O(1)$ algorithm exists

EREW: Lower bound $\Omega(\log n)$ steps

per step, one given value can be copied to at most 1 other location

e.g. for a single binary tree:

after k steps, at most 2^k locations can contain the identity of the root

A $\Theta(\log n)$ EREW algorithm exists.

Simulating a CRCW algorithm with an EREW algorithm

A p-processor CRCW algorithm can be no more than $O(\log p)$ times faster than the best p-processor EREW algorithm for the same problem.

Step-by-step simulation [Vishkin'83]

For Weak/Common/Arbitrary CRCW PRAM:

handle concurrent writes with auxiliary array A of pairs.

CRCW processor i should write x_i into location l_i:

EREW processor i writes $\langle l_i, x_i \rangle$ to $A[i]$

Sort A on p EREW processors by first coordinates

in time $O(\log p)$ [Ajtai/Komlos/Szemeredi'83], [Cole'88]

Processor j inspects write requests $A[j] = \langle l_k, x_k \rangle$ and $A[j-1] = \langle l_q, x_q \rangle$

and assigns x_k to l_k iff $l_k \neq l_q$ or $j = 0$.

For Combining (Maximum) CRCW PRAM: see [PPP p.66/67]

Simulation summary

EREW $\prec$ CREW $\prec$ CRCW

Common CRCW $\prec$ Priority CRCW

Arbitrary CRCW $\prec$ Priority CRCW

where $\prec$: “strictly weaker than” (transitive)

See [PPP p.68/69] for more separation results.

PRAM Variants [PPP 2.6]

Broadcasting with selective reduction (BSR) PRAM

Distributed RAM (DRAM)

Local memory PRAM (LPRAM)

Asynchronous PRAM

Queued PRAM (QRQW PRAM)

Hierarchical PRAM (H-PRAM)

Message passing models:
  Delay model, BSP, LogP, LogGP → Lecture 4

Broadcasting with selective reduction (BSR)

BSR: generalization of a Combine CRCW PRAM [Akl/Guenther’89]

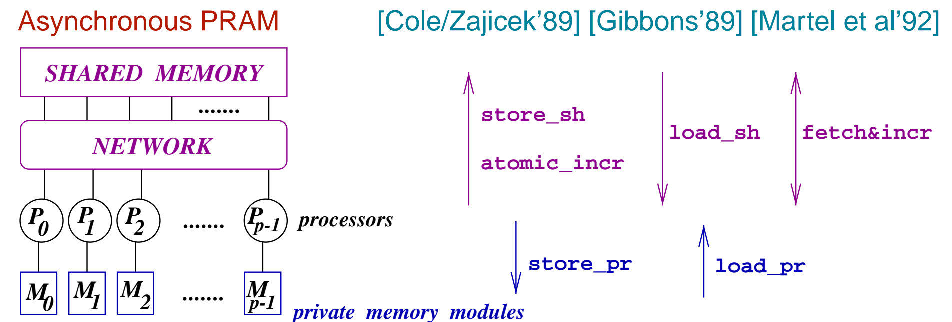
1 BSR write step:

Each processor can write a value to all memory locations (broadcast)

Each memory location computes a global reduction (max, sum, ...)
  over a specified subset of all incoming write contributions (selective reduction)

Asynchronous PRAM

Asynchronous PRAM [Cole/Zajicek’89] [Gibbons’89] [Martel et al’92]



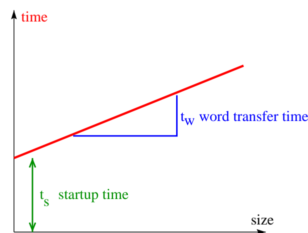
No common clock

No uniform memory access time

Sequentially consistent shared memory

Delay model

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



Cost of communicating a larger block of n bytes:

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

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

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

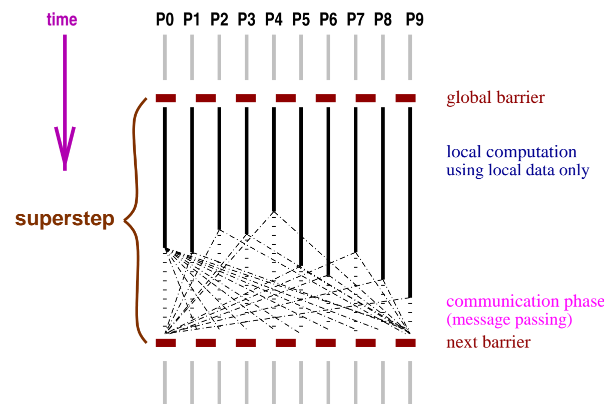
$t_{transfer}$ = transfer rate, send time per word sent

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

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

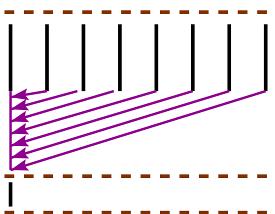
$t_{step} = w + hg + L$

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

BSP example: Global maximum computation (non-optimal algorithm)

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

// $A[0..n-1]$ distributed block-wise across p processors
step
  // local computation phase:
  $m \leftarrow -\infty$;
  for all $A[i]$ in my local partition of A {
    $m \leftarrow$ max $(m, A[i])$;
  // communication phase:
  if myPID $\neq 0$
    send (m, 0);
  else // on P_0:
    for each $i \in \{1, ..., p-1\}$
      recv (m_i, i);
step
  if myPID $= 0$
    for each $i \in \{1, ..., p-1\}$
      $m \leftarrow$ 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 (1)

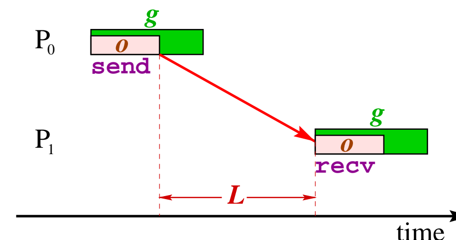
LogP model [Culler et al. 1993]

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



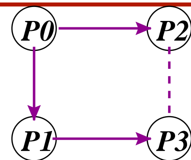
gap g = inverse network bandwidth per processor:
Network capacity is L/g messages to or from each processor.

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

transmission time for a small message:

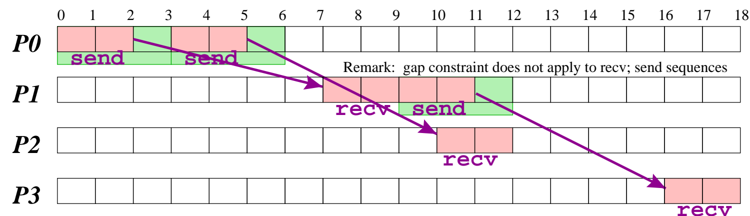
$2 \cdot o + L$ if the network capacity is not exceeded

LogP model (2)



Example: Broadcast on a 2-dimensional hypercube

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



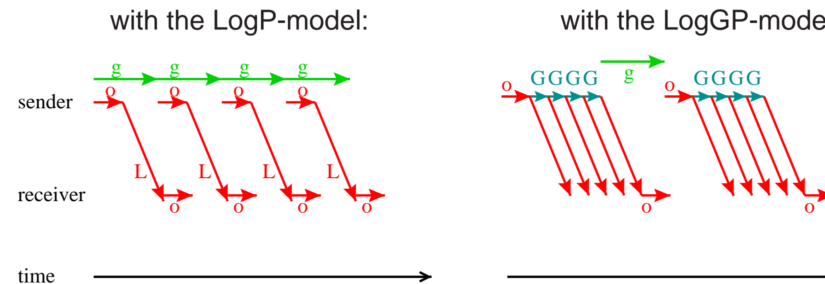
it takes at least $18\mu s$ to broadcast 1 byte from $P0$ to $P1, P2, P3$

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

LogP model (3): LogGP model

The LogGP model [Culler et al. '95] extends LogP by parameter
G = gap per word, to model block communication

Communication of an n-word-block:



$t_n = (n-1)g + L + 2o$

$t'_n = o + (n-1)G + L + o$

Summary

Parallel computation models
- Shared memory: PRAM, PRAM variants
- Message passing: Delay model, BSP, LogP, LogGP
- parallel time, work, cost

Parallel algorithmic paradigms (up to now)
- Parallel divide-and-conquer
  - (includes reduction and pointer jumping / recursive doubling)
- Data parallelism

Fundamental parallel algorithms
- Global sum
- Prefix sums
- List ranking
- Broadcast