Foundations of parallel algorithms

PRAM model

Time, work, cost

Self-simulation and Brent’s Theorem

Speedup and Amdahl’s Law

NC

Scalability and Gustafsson’s 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

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

Literature


  Addison-Wesley, 1992.

[CLR] Cormen, Leiserson, Rivest: Introduction to Algorithms,

RAM (Random Access Machine) [PPP 2.1]
programming and cost model for the analysis of sequential algorithms

Example: Computing the global sum of N elements
\[
t = t_{\text{load}} + t_{\text{store}} + \sum_{i=2}^{N} (2t_{\text{load}} + t_{\text{add}} + t_{\text{store}} + t_{\text{branch}}) = 5N - 3 + \Theta(N)
\]

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

The RAM model (2)
Algorithm analysis: Counting instructions
\[
s = d(0)
do\ i = 1, N-1\ s = s + d(i)
end do
\]

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 \( x_0, x_1, \ldots, x_{n-1} \) 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

 recursive calls: parallel time \( T(n/2) \)
 with base case: load operation, time \( O(1) \)

Divide phase: trivial, time \( O(1) \)

Combine phase: addition, time \( O(1) \)

Use induction or the master theorem \([CLR 4]\) \( T(n) \leq T(n/2) + O(1) \)

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

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

```c
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

```c
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 \( O(\log p) \)

- \( M_0, M_1, \ldots, M_p \)

**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

→ estimation based on model and pseudocode operations
→ results for large problem sizes, large # processors

(b) empirical analysis

→ measurements based on implementation
→ 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)

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

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

\( A \) is cost-effective if \( w_A(n) = \Theta(c_A(n)) \).
Asymptotic analysis for global sum computation

<table>
<thead>
<tr>
<th>problem size $n$</th>
<th># processors $p$</th>
<th>time $t(p,n)$</th>
<th>work $w(p,n)$</th>
<th>cost $c(p,n) = t \cdot p$</th>
</tr>
</thead>
</table>

Example: seq. sum algorithm

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

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

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

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

Cost: $n/\log n \cdot O(n) = O(n)$ 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 $O([q/p])$ 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 $\leq [q/p]$
- 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

Trading concurrency for cost-effectiveness

**Making the parallel sum algorithm cost-optimal:**

Instead of $n$ processors, use only $n/\log n$ 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!

**Consequences of self-simulation**

- RAM = 1-processor PRAM simulates $p$-processor PRAM in $O(p)$ time steps.
  - $\rightarrow$ RAM simulates $\Lambda$ with cost $c_{\Lambda}(n) = p_{\Lambda}(n)t_{\Lambda}(n)$ in $O(c_{\Lambda}(n))$ time.
    - (Actually possible in $O(w_{\Lambda}(n))$ time.)

Even with arb. many processors $\Lambda$ cannot be simulated any faster than $t_{\Lambda}(n)$.

For cost-optimal $\Lambda$, $c_{\Lambda}(n) = \Theta(t_{\Lambda}(n))$

**Exercise**

$p$-processor PRAM can simulate one step of $\Lambda$ requiring $p_{\Lambda}(n)$ processors in $O(p_{\Lambda}(n)/p)$ 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 $c_{\Lambda}(n) = \omega(w_{\Lambda}(n))$?
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
\begin{equation}
O \left( t_A(n) + \frac{w_A(n)}{p} \right)
\end{equation}
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, ...

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
\[ SU_{rel}(p,n) = \frac{t_A(1,n)}{t_A(p,n)} \]
\[ t_S(n) \leq t_A(1,n) \rightarrow SU_{rel}(p,n) \geq SU_{abs}(p,n). \] [PPP p.44 typo!]

Preferably used in papers on parallelization to “nice” performance results.

The relative efficiency of parallel algorithm \( A \) is the ratio
\[ EF(p,n) = \frac{t_A(1,n)}{p \cdot t_A(p,n)} \]
\[ EF(p,n) = SU_{rel}(p,n) / p \in [0,1] \]

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
\[ SU_{abs}(p,n) = \frac{t_S(n)}{t_A(p,n)} \]
If \( S \) is an optimal algorithm for \( P \), then
\[ SU_{abs}(p,n) = \frac{t_S(n)}{t_A(p,n)} \leq p \frac{t_S(n)}{c_A(n)} \leq p \]
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 \( S > p \) may exist only for small \( n \).

Speedup curves

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

\begin{itemize}
  \item \textbf{trivially parallel} (e.g., matrix product, LU decomposition, ray tracing)
  \item \textbf{work-bound algorithms} \( \rightarrow \) linear \( SU \in \Theta(p) \), work-optimal
  \item \textbf{tree-like task graphs} (e.g., global sum / max)
  \item \textbf{communication-bound} \( \rightarrow \) sublinear \( SU = 1 / fn(p) \)
\end{itemize}

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

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 \( p \).

Proof of Amdahl’s Law

\[
SU_{rel} = \frac{T(1)}{T(p)} = \frac{T(1)}{T_A + T_{Ap}(p)}
\]

Assume perfect parallelizability of the parallel part \( A^p \),

that is, \( T_{Ap}(p) = (1 - \beta)T(p) = (1 - \beta)T(1)/p \):

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

Remark:

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

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

\[
\rightarrow \text{total work } w_A(n) = w_{A^s}(n) + w_{A^p}(n)
\]

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 \( \beta \) with

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

the relative speedup of \( A \) with \( p \) processors is limited by

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

NC

Recall complexity class \( \mathcal{P} \):

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

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

“Nick’s class” \( \mathcal{NC} \):

\[\mathcal{NC} \text{ = set of problems solvable on a PRAM in polylogarithmic time } O(\log^k n) \text{ for some } k \geq 0\]

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 \( n > 10^{12} \).

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]

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

<table>
<thead>
<tr>
<th>Processors</th>
<th>Clock cycles</th>
<th>Time</th>
<th>( \text{SU}_{rel} )</th>
<th>( \text{SU}_{abs} )</th>
<th>( \text{EF} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>460118</td>
<td>1.84</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1621738</td>
<td>6.49</td>
<td>1.00</td>
<td>0.28</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>408622</td>
<td>1.63</td>
<td>3.97</td>
<td>1.13</td>
<td>0.99</td>
</tr>
<tr>
<td>16</td>
<td>105682</td>
<td>0.42</td>
<td>15.35</td>
<td>4.35</td>
<td>0.96</td>
</tr>
<tr>
<td>64</td>
<td>29950</td>
<td>0.12</td>
<td>54.15</td>
<td>15.36</td>
<td>0.85</td>
</tr>
<tr>
<td>256</td>
<td>10996</td>
<td>0.04</td>
<td>147.48</td>
<td>41.84</td>
<td>0.58</td>
</tr>
<tr>
<td>1024</td>
<td>6460</td>
<td>0.03</td>
<td>251.04</td>
<td>71.23</td>
<td>0.25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processors</th>
<th>Clock cycles</th>
<th>Time</th>
<th>( \text{SU}_{rel} )</th>
<th>( \text{SU}_{abs} )</th>
<th>( \text{EF} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>4600118</td>
<td>18.40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>16202152</td>
<td>64.81</td>
<td>1.00</td>
<td>0.28</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>4054528</td>
<td>16.22</td>
<td>4.00</td>
<td>1.13</td>
<td>1.00</td>
</tr>
<tr>
<td>16</td>
<td>1017844</td>
<td>4.07</td>
<td>15.92</td>
<td>4.52</td>
<td>0.99</td>
</tr>
<tr>
<td>64</td>
<td>258874</td>
<td>1.04</td>
<td>62.59</td>
<td>17.77</td>
<td>0.98</td>
</tr>
<tr>
<td>256</td>
<td>69172</td>
<td>0.28</td>
<td>234.23</td>
<td>66.50</td>
<td>0.91</td>
</tr>
<tr>
<td>1024</td>
<td>21868</td>
<td>0.09</td>
<td>740.91</td>
<td>210.36</td>
<td>0.72</td>
</tr>
</tbody>
</table>

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 \( T^N_A(p,n), T^M_S(n) \) in seconds

absolute, machine-uniform speedup of \( A \):
\[
\text{SU}_{abs}(p,n) = \frac{T^M_S(n)}{T^N_A(p,n)}
\]

parallelization slowdown of \( A \):
\[
\text{SL}(n) = \frac{T^M_S(1,n)}{T^N_A(p,n)}
\]

Hence,
\[
\text{SU}_{abs}(p,n) = \frac{\text{SU}_{rel}(p,n)}{\text{SL}(n)}
\]

absolute, machine-nonuniform speedup = \( \frac{T^M_S(p,n)}{T^N_A(p,n)} \)

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

Scalability

For machine \( N \) with \( p \leq p_A(n) \),
\[ t_A(p,n) = O(c_A(n)/p) \]
and thus \( \text{SU}_{abs}(p,n) = \frac{T^M_S(n)}{c_A(n)} \).

\( \rightarrow \) 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 \cite{Rao,Kumar’87}

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

$$\text{EF}(p, n) = \frac{T_M^A(1,n)}{p \cdot T_M^A(p,n)} = \frac{SU_{rel}(p,n)}{p}$$

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

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 $\varepsilon$.

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

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

**Proof of Gustafsson’s Law**

Scaled speedup for $p = n > 1$:

$$SU_{rel}(n) = \frac{T_n(1)}{T_n(n)} = \frac{T_A + w_{Ap}(n)}{T_A + T_{Ap}}$$

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

$$SU_{rel}(n) = \frac{\alpha + (1 - \alpha)n}{n} = n - (n - 1)\alpha$$

Yields better speedup predictions for data-parallel algorithms.

**Fundamental PRAM algorithms**

- reduction \(\sqrt{\text{see parallel sum algorithm}}\)
- prefix-sums
- list ranking

Oblivious (PRAM) algorithm:

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 \cite[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 $x_0, x_1, \ldots$
- daily balances: $(0, x_0), (x_0 + x_1, x_0 + x_1 + x_2, \ldots)$

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 $y_i$

$\rightarrow$ 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:

\[
\begin{align*}
\textbf{real} & \quad a : \text{array} [0..N-1]; \\
\textbf{int} & \quad \text{stride}; \\
\text{stride} & \leftarrow 1; \\
\textbf{while} & \quad \text{stride} < N \textbf{ do} \quad \textbf{forall} \ i : [0..N-1] \textbf{ in parallel do} \quad \textbf{if} \ i \geq \text{stride} \quad \textbf{then} \quad a[i] \leftarrow a[i-\text{stride}] + a[i]; \\
\text{stride} & \leftarrow \text{stride} \times 2; \quad \text{(* finally, sum in } a[N-1] *)
\end{align*}
\]

Parallel prefix sums (4)

Odd/even parallel prefix

P_{oddeven}(n) = P_{oddeven}(n/2): odd/even

\[
\begin{align*}
\text{EREW, } 2\log n - 2 \text{ time steps, work } 2n - \log n - 2, \text{ cost } \Theta(n\log n)
\end{align*}
\]

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

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 prefix-sums problem can be solved on a \( n \)-processor EREW PRAM in \( \Theta(\log n) \) time steps and cost \( \Theta(n) \).
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 \( \rightarrow \) next
- that I splice
- out of the list

Every step
+ doubles #lists
+ halves lengths
\( \rightarrow [\log_2 n] \) steps

Not work-efficient!

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 \( [\log_2 \text{max. depth}] \) 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

For Weak/Common/Arbitrary CRCW PRAM:
handle concurrent writes with auxiliary array \( \Lambda \) of pairs.
CRCW processor \( i \) should write \( x_i \) into location \( l_i \):
EREW processor \( i \) writes \( (l_i, x_i) \) to \( \Lambda[i] \)
Sort \( \Lambda \) on \( p \) EREW processors by first coordinates in time \( O(\log p) \)

Processor \( j \) inspects write requests \( \Lambda[j] = (l_j, x_j) \) and \( \Lambda[j-1] = (l_{j-1}, x_{j-1}) \)
and assigns \( x_j \) to \( l_j \) 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.

Broadcasting with selective reduction (BSR) PRAM

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

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

Asynchronous PRAM

No common clock
No uniform memory access time
Sequentially consistent shared memory
Delay model

Idealized multicompiler: point-to-point communication costs time $t_{\text{msg}}$.

Cost of communicating a larger block of $n$ bytes:

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

Assumption: network not overloaded; no conflicts occur at routing

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

accounts for hardware and software overhead

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

depends on the network bandwidth.

BSP model

Bulk-synchronous parallel programming

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

- MIMD
- SPMD

$h$-relation models communication pattern / volume

fan-in, fan-out of $P_i$

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

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

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

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:

$$
2 \cdot o + L \text{ if the network capacity is not exceeded}
$$
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 $P_0$ to $P_1, P_2, P_3$

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 $G = \text{gap per word}$ to model block communication

Communication of an $n$-word-block:

\[ t_n = (n-1)g + L + 2o \quad \text{with the LogP-model:} \]
\[ t'_n = o + (n-1)G + L + o \quad \text{with the LogGP-model:} \]

Summary

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

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