Computation by circuits

Model: abstraction of reality allowing qualitative and quantitative reasoning
E.g. atom, galaxy, biological cell, Newton’s universe, Einstein’s universe...

Computation model: abstract computing device to reason about computations and algorithms
E.g. scales+weights, Turing machine, von Neumann machine (“ordinary computer”), JVM, quantum computer...

An algorithm in a specific model: input $\rightarrow$ (computation steps) $\rightarrow$ output
Input/output encoding must be specified
Algorithm complexity (worst-case): $T(n) = \max_{\text{input size}=n}$ computation steps
Computation by circuits

Computation models and algorithms

Algorithm complexity depends on the model

E.g. sorting $n$ items:
- $\Omega(n \log n)$ in the comparison model
- $O(n)$ in the arithmetic model (by radix sort)

E.g. factoring large numbers:
- hard in a von Neumann-type (standard) model
- not so hard on a quantum computer

E.g. deciding if a program halts on a given input:
- impossible in a standard (or even quantum) model
- can be added to the standard model as an oracle, to create a more powerful model

Basic special-purpose parallel model: a circuit

Directed acyclic graph (dag)
Fixed number of inputs/outputs
Oblivious computation: control sequence independent of the input

Bounded or unbounded fan-in/fan-out

Elementary operations:
- arithmetic/Boolean/comparison
- each (usually) constant time

=size\ =\ number\ of\ nodes

depth\ =\ max\ path\ length\ from\ input\ to\ output

Timed circuits with feedback: systolic arrays

A comparison network is a circuit of comparator nodes

The input and output sequences have the same length

Examples:
Computation by circuits

The comparison network model

A merging network is a comparison network that takes two sorted input sequences of length \( n' \), \( n'' \), and produces a sorted output sequence of length \( n = n' + n'' \).

A sorting network is a comparison network that takes an arbitrary input sequence, and produces a sorted output sequence.

A sorting (or merging) network is equivalent to an oblivious sorting (or merging) algorithm; the network's size/depth determine the algorithm's sequential/parallel complexity.

General merging: \( O(n) \) comparisons, non-oblivious
General sorting: \( O(n \log n) \) comparisons by mergesort, non-oblivious

What is the complexity of oblivious sorting?

Naive sorting networks

**INSERTION-SORT**

- **size** \( n(n - 1)/2 = O(n^2) \)
- **depth** \( 2n - 3 = O(n) \)

**INSERTION-SORT** (8)

- **size** 28
- **depth** 13

Identical to **BUBBLE-SORT**!

**BUBBLE-SORT**

- **size** \( n(n - 1)/2 = O(n^2) \)
- **depth** \( 2n - 3 = O(n) \)

**BUBBLE-SORT** (8)

- **size** 28
- **depth** 13

**Zero-one principle**: A comparison network is sorting, if and only if it sorts all input sequences of 0s and 1s.


Assume a given network does not sort input \( x = (x_1, \ldots, x_n) \)

\[
(x_1, \ldots, x_n) \mapsto (y_1, \ldots, y_n) \quad \exists k, l : k < l : y_k > y_l
\]

Let \( X_i = \begin{cases} 0 & \text{if } x_i < y_k \\ 1 & \text{if } x_i \geq y_k \end{cases} \), and run the network on input \( X = (X_1, \ldots, X_n) \)

For all \( i, j \) we have \( x_i \leq x_j \implies X_i \leq X_j \), therefore each \( X_i \) follows the same path through the network as \( x_i \)

\[
(X_1, \ldots, X_n) \mapsto (Y_1, \ldots, Y_n) \quad Y_k = 1 > 0 = Y_l
\]

We have \( k < l \) but \( Y_k > Y_l \), so the network does not sort 0s and 1s.
The zero-one principle applies to sorting, merging and other comparison problems (e.g. selection).

It allows one to test:
1. a sorting network by checking only \(2^n\) input sequences, instead of a much larger number \(n! \approx (n/e)^n\).
2. a merging network by checking only \((n' + 1) \cdot (n'' + 1)\) pairs of input sequences, instead of an exponentially larger number \(\binom{n}{n'} = \binom{n}{n''}\).

General merging: \(O(n)\) comparisons, non-oblivious

How fast can we merge obliviously?

\(\langle x_1 \leq \cdots \leq x_{n'}\rangle, \langle y_1 \leq \cdots \leq y_{n''}\rangle \mapsto \langle z_1 \leq \cdots \leq z_n\rangle\)

**Odd-even merging**

When \(n' = n'' = 1\) compare \((x_1, y_1)\), otherwise

- merge \(\langle x_1, x_3, \ldots \rangle, \langle y_1, y_3, \ldots \rangle \mapsto \langle u_1 \leq u_2 \leq \cdots \leq u_{\lceil n'/2 \rceil + \lceil n''/2 \rceil}\rangle\)
- merge \(\langle x_2, x_4, \ldots \rangle, \langle y_2, y_4, \ldots \rangle \mapsto \langle v_1 \leq v_2 \leq \cdots \leq v_{\lceil n'/2 \rceil + \lceil n''/2 \rceil}\rangle\)
- compare pairwise: \((u_2, v_1), (u_3, v_2), \ldots\)

\(\text{size}_{\text{OEM}}(n', n'') \leq 2 \cdot \text{size}_{\text{OEM}}(n'/2, n''/2) + O(n) = O(n \log n)\)

\(\text{depth}_{\text{OEM}}(n', n'') \leq \text{depth}_{\text{OEM}}(n'/2, n''/2) + 1 = O(\log n)\)

**Correctness proof of odd-even merging (sketch):** by induction and the zero-one principle.

**Induction base:** trivial (2 inputs, 1 comparator)

**Inductive step.** By the inductive hypothesis, we have for all \(k, l\):

\(\langle 0^{\lceil k/2 \rceil}11\ldots, 0^{\lceil l/2 \rceil}11\ldots \rangle \mapsto \langle 0^{\lceil k/2 \rceil + \lceil l/2 \rceil}11\ldots \rangle\)

\(\langle 0^{\lceil k/2 \rceil}11\ldots, 0^{\lceil l/2 \rceil}11\ldots \rangle \mapsto \langle 0^{\lceil k/2 \rceil + \lceil l/2 \rceil}11\ldots \rangle\)

We need \(\langle 0^k11\ldots, 0^l11\ldots \rangle \mapsto \langle 0^{k+l}11\ldots \rangle\)

\([\lceil k/2 \rceil + \lceil l/2 \rceil] - ([k/2] + [l/2]) =
\begin{cases} 
0,1 & \text{result sorted: } \langle 0^{k+l}11\ldots \rangle \\
2 & \text{single pair wrong: } \langle 0^{k+l-1}0111\ldots \rangle
\end{cases}\)

The final stage of comparators corrects the wrong pair.
Computation by circuits
Efficient merging and sorting networks

Sorting an arbitrary input \( \langle x_1, \ldots, x_n \rangle \)

**Odd-even merge sorting**

[Batcher: 1968]

When \( n = 1 \) we are done, otherwise

- sort \( \langle x_1, \ldots, x_{\lceil n/2 \rceil} \rangle \) recursively
- sort \( \langle x_{\lceil n/2 \rceil + 1}, \ldots, x_n \rangle \) recursively
- merge results by \( \text{OEM}(\lceil n/2 \rceil, \lfloor n/2 \rfloor) \)

\[
\text{size}_{\text{OEM-SORT}}(n) \leq 2 \cdot \text{size}_{\text{OEM-SORT}}(n/2) + \text{size}_{\text{OEM}}(n) = 2 \cdot \text{size}_{\text{OEM-SORT}}(n/2) + O(n \log n) = O(n \log n^2)
\]

\[
\text{depth}_{\text{OEM-SORT}}(n) \leq \text{depth}_{\text{OEM-SORT}}(n/2) + \text{depth}_{\text{OEM}}(n) = \text{depth}_{\text{OEM-SORT}}(n/2) + O(\log n) = O((\log n)^2)
\]

A bitonic sequence: \( \langle x_1 \geq \cdots \geq x_m \leq \cdots \leq x_n \rangle \)

**Bitonic merging**: sorting a bitonic sequence

When \( n = 1 \) we are done, otherwise

- sort bitonic \( \langle x_1, x_3, \ldots \rangle \) recursively
- sort bitonic \( \langle x_2, x_4, \ldots \rangle \) recursively
- compare pairwise: \( (x_1, x_2), (x_3, x_4), \ldots \)

Correctness proof: by zero-one principle (exercise)

(Note: cannot exchange \( \geq \) and \( \leq \) in definition of bitonic!)

Bitonic merging is more flexible than odd-even merging, since a single circuit applies to all values of \( m \)

\[
\text{size}_{\text{BM}}(n) = O(n \log n) \quad \text{depth}_{\text{BM}}(n) = O(\log n)
\]

\[
\text{BM}(n)
\]

size \( O(n \log n) \)

depth \( O(\log n) \)

\[
\text{BM}(8)
\]

size 12
depth 3

\[
\text{BM}(\lceil n/2 \rceil)
\]

\[
\text{BM}(\lfloor n/2 \rfloor)
\]
Computation by circuits
Efficient merging and sorting networks

Bitonic merge sorting

[Batcher: 1968]

When \( n = 1 \) we are done, otherwise

- sort \( \langle x_1, \ldots, x_{\lceil n/2 \rceil} \rangle \mapsto \langle y_1 \geq \cdots \geq y_{\lceil n/2 \rceil} \rangle \) in reverse, recursively
- sort \( \langle x_{\lceil n/2 \rceil + 1}, \ldots, x_n \rangle \mapsto \langle y_{\lceil n/2 \rceil + 1} \leq \cdots \leq y_n \rangle \) recursively
- sort bitonic \( \langle y_1 \geq \cdots \geq y_m \leq \cdots \leq y_n \rangle \) \( m = \lceil n/2 \rceil \) or \( \lfloor n/2 \rfloor + 1 \)

Sorting in reverse seems to require “inverted comparators”, however

- comparators are actually nodes in a circuit, which can always be drawn using “standard comparators”
- a network drawn with “inverted comparators” can be converted into one with only “standard comparators” by a top-down rearrangement

\[ \text{size}_{BM-SORT}(n) = O(n(\log n)^2) \]
\[ \text{depth}_{BM-SORT}(n) = O((\log n)^2) \]

Both \( OEM-SORT \) and \( BM-SORT \) have size \( \Theta(n(\log n)^2) \)

Is it possible to sort obliviously in size \( o(n(\log n)^2) \)? \( O(n \log n) \)?

AKS sorting

[Paterson: 1990]; [Seiferas: 2009]

Sorting network: size \( O(n \log n) \), depth \( O(\log n) \)

Uses sophisticated graph theory (expanders)

Asymptotically optimal, but has huge constant factors
Parallel computation models

The PRAM model

Parallel Random Access Machine (PRAM) [Fortune, Wyllie: 1978]

Simple, idealised general-purpose parallel model

Contains
- unlimited number of processors (1 time unit/op)
- global shared memory (1 time unit/access)

Operates in full synchrony

PRAM computation: sequence of parallel steps
Communication and synchronisation taken for granted
Not scalable in practice!

PRAM variants:
- concurrent/exclusive read
- concurrent/exclusive write

CRCW, CREW, EREW, (ERCW) PRAM
E.g. a linear system solver: $O((\log n)^2)$ steps using $n^4$ processors

PRAM algorithm design: minimising number of steps, sometimes also number of processors

The BSP model

Bulk-Synchronous Parallel (BSP) computer [Valiant: 1990]

Simple, realistic general-purpose parallel model

Contains
- $p$ processors, each with local memory (1 time unit/operation)
- communication environment, including a network and an external memory ($g$ time units/data unit communicated)
- barrier synchronisation mechanism ($l$ time units/synchronisation)

Parameter $g$ corresponds to the network’s communication gap (inverse bandwidth) — the time for a data unit to enter/exit the network
Parameter $l$ corresponds to the network’s latency — the worst-case time for a data unit to get across the network

E.g. for Cray T3E: $p = 64$, $g \approx 78$, $l \approx 1825$
Parallel computation models

The BSP model

BSP computation: sequence of parallel supersteps

Asynchronous computation/communication within supersteps (includes data exchange with external memory)

Synchronisation before/after each superstep

Cf. CSP: parallel collection of sequential processes

Compositional cost model

For individual processor proc in superstep sstep:

- \( \text{comp}(sstep, proc) \): the amount of local computation and local memory operations by processor proc in superstep sstep
- \( \text{comm}(sstep, proc) \): the amount of data sent and received by processor proc in superstep sstep

For the whole BSP computer in one superstep sstep:

- \( \text{comp}(sstep) = \max_{0 \leq proc < p} \text{comp}(sstep, proc) \)
- \( \text{comm}(sstep) = \max_{0 \leq proc < p} \text{comm}(sstep, proc) \)
- \( \text{cost}(sstep) = \text{comp}(sstep) + \text{comm}(sstep) \cdot g + l \)

For the whole BSP computation with sync supersteps:

- \( \text{comp} = \sum_{0 \leq sstep < \text{sync}} \text{comp}(sstep) \)
- \( \text{comm} = \sum_{0 \leq sstep < \text{sync}} \text{comm}(sstep) \)
- \( \text{cost} = \sum_{0 \leq sstep < \text{sync}} \text{cost}(sstep) = \text{comp} + \text{comm} \cdot g + \text{sync} \cdot l \)

The input/output data are stored in the external memory; the cost of input/output is included in \( \text{comm} \)

E.g. for a particular linear system solver with an \( n \times n \) matrix:

\( \text{comp} \, O(n^3/p) \quad \text{comm} \, O(n^2/p^{1/2}) \quad \text{sync} \, O(p^{1/2}) \)
Parallel computation models
The BSP model

BSP computation: scalable, portable, predictable
BSP algorithm design: minimising \textit{comp}, \textit{comm}, \textit{sync}

Main principles:
- load balancing minimises \textit{comp}
- data locality minimises \textit{comm}
- coarse granularity minimises \textit{sync}

Data locality exploited, network locality ignored!
Typically, problem size $n \gg p$ (slackness)

Parallel computation models
Network routing

BSP network model: complete graph, uniformly accessible (access efficiency described by parameters $g$, $l$)
Has to be implemented on concrete networks
Parameters of a network topology (i.e. the underlying graph):
- degree — number of links per node
- diameter — maximum distance between nodes

Low degree — easier to implement
Low diameter — more efficient

2D array network
$p = q^2$ processors
degree 4
diameter $p^{1/2} = q$

3D array network
$p = q^3$ processors
degree 6
diameter $3/2 \cdot p^{1/3} = 3/2 \cdot q$
Parallel computation models

Network routing

**Butterfly network**

- \( p = q \log q \) processors
- Degree 4
- Diameter \( \approx \log p \approx \log q \)

![Butterfly Network Diagram]

**Hypercube network**

- \( p = 2^q \) processors
- Degree \( \log p = q \)
- Diameter \( \log p = q \)

![Hypercube Network Diagram]

<table>
<thead>
<tr>
<th>Network</th>
<th>Degree</th>
<th>Diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D array</td>
<td>2</td>
<td>( 1/2 \cdot p )</td>
</tr>
<tr>
<td>2D array</td>
<td>4</td>
<td>( p^{1/2} )</td>
</tr>
<tr>
<td>3D array</td>
<td>6</td>
<td>( 3/2 \cdot p^{1/3} )</td>
</tr>
<tr>
<td>Butterfly</td>
<td>( \log p )</td>
<td>( \log p )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

BSP parameters \( g \), \( l \) depend on degree, diameter, routing strategy

Assume store-and-forward routing (alternative — wormhole)

Assume distributed routing: no global control

Oblivious routing: path determined only by source and destination

E.g. greedy routing: a packet always takes the shortest path

h-relations (h-superstep): every processor sends and receives \( \leq h \) packets

Sufficient to consider permutations (1-relations): once we can route any permutation in \( k \) steps, we can route any h-relation in \( hk \) steps

Any routing method may be forced to make \( \Omega(\text{diameter}) \) steps

Any oblivious routing method may be forced to make \( \Omega(p^{1/2}/\text{degree}) \) steps

Many practical patterns force such “hot spots” on traditional networks
Parallel computation models
Network routing

Routing based on sorting networks
Each processor corresponds to a wire
Each link corresponds to (possibly several) comparators
Routing corresponds to sorting by destination address
Each stage of routing corresponds to a stage of sorting
Such routing is non-oblivious (for individual packets)!

<table>
<thead>
<tr>
<th>Network</th>
<th>Degree</th>
<th>Diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>OEM-SORT/BM-SORT</td>
<td>$O((\log p)^2)$</td>
<td>$O((\log p)^2)$</td>
</tr>
<tr>
<td>AKS</td>
<td>$O(\log p)$</td>
<td>$O(\log p)$</td>
</tr>
</tbody>
</table>

No “hot spots”: can always route a permutation in $O(\text{diameter})$ steps
Requires a specialised network, too messy and impractical

Two-phase randomised routing: [Valiant: 1980]

- send every packet to random intermediate destination
- forward every packet to final destination

Both phases oblivious (e.g. greedy), but non-oblivious overall due to randomness

Hot spots very unlikely: on a 2D array, butterfly, hypercube, can route a permutation in $O(\text{diameter})$ steps with high probability

On a hypercube, the same holds even for a $\log p$-relation

Hence constant $g$, $l$ in the BSP model

BSP implementation: processes placed at random, communication delayed until end of superstep
All packets with same source and destination sent together, hence message overhead absorbed in $l$

<table>
<thead>
<tr>
<th>Network</th>
<th>$g$</th>
<th>$l$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D array</td>
<td>$O(p)$</td>
<td>$O(p)$</td>
</tr>
<tr>
<td>2D array</td>
<td>$O(p^{1/2})$</td>
<td>$O(p^{1/2})$</td>
</tr>
<tr>
<td>3D array</td>
<td>$O(p^{1/3})$</td>
<td>$O(p^{1/3})$</td>
</tr>
<tr>
<td>Butterfly</td>
<td>$O(\log p)$</td>
<td>$O(\log p)$</td>
</tr>
<tr>
<td>Hypercube</td>
<td>$O(1)$</td>
<td>$O(\log p)$</td>
</tr>
</tbody>
</table>

Actual values of $g$, $l$ obtained by running benchmarks
Basic parallel algorithms
Broadcast/combine

The broadcasting problem:
- Initially, one designated processor holds a value \( a \)
- At the end, every processor must hold a copy of \( a \)

The combining problem (complementary to broadcasting):
- Initially, every processor holds a value \( a_i \), \( 0 \leq i < p \)
- At the end, one designated processor must hold \( a_0 \cdot \cdots \cdot a_{p-1} \) for a given associative operator \( \cdot \) (e.g. \( + \))

By symmetry, we only need to consider broadcasting

Direct broadcast:
- Designated processor makes \( p - 1 \) copies of \( a \) and sends them directly to destinations

\[
\begin{array}{ccc}
 a & a & a \\
 a & a & a & a \\
\end{array}
\]

\[
\text{comp } O(p) \quad \text{comm } O(p) \quad \text{sync } O(1)
\]

(from now on, cost components will be shaded when they are optimal, i.e. cannot be improved under reasonable assumptions)

Binary tree broadcast:
- Initially, only designated processor is awake
- Processors are woken up in \( \log p \) rounds
- In every round, every awake processor makes a copy of \( a \) and send it to a sleeping processor, waking it up

In round \( k = 0, \ldots, \log p - 1 \), the number of awake processors is \( 2^k \)

\[
\text{comp } O(\log p) \quad \text{comm } O(\log p) \quad \text{sync } O(\log p)
\]

The array broadcasting/combining problem: broadcast/combine an array of size \( n \geq p \) elementwise

(effectively, \( n \) independent instances of broadcasting/combining)
Basic parallel algorithms

Two-phase array broadcast:

- partition array into \( p \) blocks of size \( n/p \)
- scatter blocks, then total-exchange blocks

Tree (contd.):

- a generalisation of broadcasting/combining
- can be defined top-down (root the input, leaves the outputs) or bottom-up

For bottom-up computation, reverse the steps

\[
\begin{align*}
\text{Sequential work } & O(n) \\
\text{Parallel balanced tree computation (contd.)} & \\
\text{Partition } & \text{tree}(n) \text{ into } \\
\text{one top block, isomorphic to } & \text{tree}(p) \\
\text{a bottom layer of } & p \text{ blocks, each isomorphic to } \text{tree}(n/p)
\end{align*}
\]
The described parallel balanced tree algorithm is fully optimal:

- optimal $\text{comp } O(n/p) = O\left(\frac{\text{sequential work}}{p}\right)$
- optimal $\text{comm } O(n/p) = O\left(\frac{\text{input/output size}}{p}\right)$
- optimal $\text{sync } O(1)$

For other problems, we may not be so lucky. However, we are typically interested in algorithms that are optimal in $\text{comp}$ (under reasonable assumptions). Optimality in $\text{comm}$ and $\text{sync}$ is considered relative to that.

For example, we are not allowed to run the whole computation in a single processor, sacrificing $\text{comp}$ and $\text{comm}$ to guarantee optimal $\text{sync } O(1)$!

Let $\cdot$ be an associative operator, computable in time $O(1)$

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

E.g. numerical $+,-,\min\ldots$

The prefix sums problem:

$$\begin{bmatrix}a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} \mapsto \begin{bmatrix}a_0 \\ a_0 \cdot a_1 \\ a_0 \cdot a_1 \cdot a_2 \\ \vdots \\ a_0 \cdot a_1 \bullet \ldots \bullet a_{n-1} \end{bmatrix}$$

Sequential work $O(n)$

The prefix circuit 

prefix circuit:

$$\text{prefix}(n)$$

where $a_{k:l} = a_k \cdot a_{k+1} \bullet \ldots \bullet a_l$, and "*" is a dummy value

The underlying dag is called the prefix dag
Basic parallel algorithms
Balanced tree and prefix sums

Parallel prefix computation
The dag $\text{prefix}(n)$ consists of
- a dag similar to bottom-up $\text{tree}(n)$, but with an extra output per node (total $n$ inputs, $n$ outputs)
- a dag similar to top-down $\text{tree}(n)$, but with an extra input per node (total $n$ inputs, $n$ outputs)

Both trees can be computed by the previous algorithm. Extra inputs/outputs are absorbed into $O(n/p)$ communication cost.

$n \geq p^2$

$\text{comp } O(n/p)$  $\text{comm } O(n/p)$  $\text{sync } O(1)$

Basic parallel algorithms
Balanced tree and prefix sums

Application: binary addition via Boolean logic

$x + y = z$

Let $x = \langle x_{n-1}, \ldots, x_0 \rangle$, $y = \langle y_{n-1}, \ldots, y_0 \rangle$, $z = \langle z_n, z_{n-1}, \ldots, z_0 \rangle$ be the binary representation of $x$, $y$, $z$
The problem: given $\langle x_i \rangle$, $\langle y_i \rangle$, compute $\langle z_i \rangle$ using bitwise $\land$ (“and”), $\lor$ (“or”), $\oplus$ (“xor”)

Let $c = \langle c_{n-1}, \ldots, c_0 \rangle$, where $c_i$ is the $i$-th carry bit

We have: $x_i + y_i + c_{i-1} = z_i + 2c_i$  $0 \leq i < n$

We have $c_i = u_i \lor (v_i \land c_{i-1})$

Let $F_{u,v}(c) = u \lor (v \land c)$  $c_i = F_{u_i,v_i}(c_{i-1})$

We have $c_i = F_{u_i,v_i}( \ldots F_{u_0,v_0}(0) \ldots ) = F_{u_0,v_0} \circ \cdots \circ F_{u_i,v_i}(0)$

Function composition $\circ$ is associative

$F_{u',v'} \circ F_{u,v}(c) = F_{u,v}(F_{u',v'}(c)) = u \lor (v \land (u' \lor (v' \land c))) = u \lor (v \land u') \lor (v \land v' \land c) = F_{u,v}(v \land u', v \land v')(c)$

Hence, $F_{u',v'} \circ F_{u,v} = F_{u,v}(v \land u', v \land v')$ is computable from $u$, $v$, $u'$, $v'$ in time $O(1)$

We compute $F_{u_0,v_0} \circ \cdots \circ F_{u_i,v_i}$ for all $i$ by $\text{prefix}(n)$

Then compute $\langle c_i \rangle$, $\langle z_i \rangle$ in size $O(n)$ and depth $O(1)$

Resulting circuit has size $O(n)$ and depth $O(\log n)$
Basic parallel algorithms
Fast Fourier Transform and the butterfly dag

A complex number $\omega$ is called a primitive root of unity of degree $n$, if $\omega, \omega^2, \ldots, \omega^{n-1} \neq 1$, and $\omega^n = 1$.

The Discrete Fourier Transform problem: $F_{n,\omega}(a) = F_{n,\omega} \cdot a = b$, where $F_{n,\omega} = [\omega^i j=0]^{n-1}$

$$
\begin{bmatrix}
1 & 1 & 1 & \ldots & 1 \\
1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\
1 & \omega^2 & \omega^4 & \ldots & \omega^{n-2} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{n-1} & \omega^{n-2} & \ldots & \omega
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
a_2 \\
\vdots \\
a_{n-1}
\end{bmatrix}
= 
\begin{bmatrix}
b_0 \\
b_1 \\
b_2 \\
\vdots \\
b_{n-1}
\end{bmatrix}
$$

Sequential work $O(n^2)$ by matrix-vector multiplication

The Fast Fourier Transform (FFT) algorithm ("four-step" version)

Assume $n = 2^r$. Let $m = n^{1/2} = 2^r$.

Let $A_{u,v} = a_{mu+v}$, $B_{s,t} = b_{ms+t}$, $s, t, u, v = 0, \ldots, m-1$.

Matrices $A, B$ are vectors $a, b$ written out as $m \times m$ matrices:

$B_{s,t} = \sum_{u,v} \omega^{(ms+tv)} A_{u,v} = \sum_u \sum_v \omega^{(mu+tv)} A_{u,v}$

Thus, $B = F_{m,\omega^m}(T_{m,\omega}(F_{m,\omega^m}(A)))$

$F_{m,\omega^m}(A)$ is $m$ independent DFTs of size $m$ on each column of $A$

Equivalent to matrix-matrix product of size $m$

$F_{m,\omega^m}(A) = F_{m,\omega^m} \cdot A$

$F_{m,\omega^m}(A)_{v,t} = \sum_u (\omega^m)^{tu} A_{u,v}$

$T_{m,\omega}(A)$ is the transpose of matrix $A$, with twiddle-factor scaling

$T_{m,\omega}(A)_{v,t} = \omega^{tv} \cdot A_{t,v}$

The FFT circuit

The underlying dag is called butterfly dag
Basic parallel algorithms
Fast Fourier Transform and the butterfly dag

The FFT circuit and the butterfly dag (contd.)

\( \text{bfly}(n) \)
- \( n \) inputs
- \( n \) outputs
- size \( \frac{n \log n}{2} \)
- depth \( \log n \)

Applications: Fast Fourier Transform; sorting bitonic sequences (bitonic merging)

Parallel butterfly computation

To compute \( \text{bfly}(n) \):
- every processor is assigned \( n^{1/2}/p \) blocks from the top layer; the processor reads the total of \( n/p \) inputs, computes the blocks, and writes back the \( n/p \) outputs
- every processor is assigned \( n^{1/2}/p \) blocks from the bottom layer; the processor reads the total of \( n/p \) inputs, computes the blocks, and writes back the \( n/p \) outputs

\( n \geq p^2 \)

- \( \text{comp} O(\frac{n \log n}{p}) \)
- \( \text{comm} O(n/p) \)
- \( \text{sync} O(1) \)

Basic parallel algorithms
Ordered grid

The ordered 2D grid dag

\( \text{grid}_2(n) \)
- nodes arranged in an \( n \times n \) grid
- edges directed top-to-bottom, left-to-right
- \( \leq 2n \) inputs (to left/top borders)
- \( \leq 2n \) outputs (from right/bottom borders)
- size \( n^2 \) depth \( 2n - 1 \)

Applications: Gauss–Seidel iteration (single step); triangular system solution; dynamic programming; 1D cellular automata
Sequential work \( O(n^2) \)
Basic parallel algorithms
Ordered grid

Parallel ordered 2D grid computation

\[
grid_2(n)
\]

Consists of a \( p \times p \) grid of blocks, each isomorphic to \( grid_2(n/p) \)

The blocks can be arranged into \( 2p - 1 \) anti-diagonal layers, with \( \leq p \) independent blocks in each layer

\[
\begin{array}{c}
\text{comp} \quad O(n^2/p) \\
\text{comm} \quad O(n) \\
\text{sync} \quad O(p)
\end{array}
\]

Application: string comparison

Let \( a, b \) be strings of characters

A subsequence of string \( a \) is obtained by deleting some (possibly none, or all) characters from \( a \)

The longest common subsequence (LCS) problem: find the longest string that is a subsequence of both \( a \) and \( b \)

\[
a = \text{“define”} \quad b = \text{“design”}
\]

\[
LCS(a, b) = \text{“dein”}
\]

In computational molecular biology, the LCS problem and its variants are referred to as sequence alignment

LCS computation by dynamic programming

Let \( lcs(a, b) \) denote the LCS length

\[
lcs(\varepsilon, \varepsilon) = 0 \quad lcs(\varepsilon, b) = 0
\]

\[
lcs(a, \varepsilon) = \begin{cases} lcs(aalpha, b), & \text{if } \alpha \neq \beta \\ lcs(a, b) + 1, & \text{if } \alpha = \beta \end{cases}
\]

\[
lcs(\text{“define”}, \text{“design”}) = 4
\]

\[
LCS(a, b) \text{ can be “traced back” through the table at no extra asymptotic cost}
\]

Data dependence in the table corresponds to the 2D grid dag
Basic parallel algorithms
Ordered grid

Parallel LCS computation

The 2D grid approach gives a BSP algorithm for the LCS problem (and many other problems solved by dynamic programming)

\[ \text{comp } O\left(\frac{n^2}{p}\right) \quad \text{comm } O(n) \quad \text{sync } O(p) \]

It may seem that the grid dag algorithm for the LCS problem is the best possible. However, an asymptotically faster BSP algorithm can be obtained by divide-and-conquer, via a careful analysis of the resulting LCS subproblems on substrings.

The semi-local LCS algorithm (details omitted) \[ \text{comp } O\left(\frac{n^2}{p}\right) \quad \text{comm } O\left(\frac{n \log p}{p^{1/2}}\right) \quad \text{sync } O(\log p) \]

Alexander Tiskin (Warwick) Efficient Parallel Algorithms 73 / 185

Basic parallel algorithms
Ordered grid

Parallel ordered 3D grid computation

grid_3(n)

Consists of a \( p^{1/2} \times p^{1/2} \times p^{1/2} \) grid of blocks, each isomorphic to \( grid_3(n/p^{1/2}) \)

The blocks can be arranged into \( 3p^{1/2} - 2 \) anti-diagonal layers, with \( \leq p \) independent blocks in each layer

\[ \text{comp } O(n^3/p) \quad \text{comm } O(n^2/p^{1/2}) \quad \text{sync } O(p^{1/2}) \]

Parallel ordered 3D grid computation (contd.)

The computation proceeds in \( 3p^{1/2} - 2 \) stages, each computing a layer of blocks. In a stage:

- every processor is either assigned a block or is idle
- a non-idle processor reads the \( 3n^2/p \) block inputs, computes the block, and writes back the \( 3n^2/p \) block outputs

\[ \text{comp: } (3p^{1/2} - 2) \cdot O\left((n/p^{1/2})^3\right) = O\left(p^{1/2} \cdot n^3/p^{3/2}\right) = O(n^3/p) \]
\[ \text{comm: } (3p^{1/2} - 2) \cdot O\left((n/p^{1/2})^2\right) = O\left(p^{1/2} \cdot n^2/p\right) = O(n^2/p^{1/2}) \]

\[ n \geq p^{1/2} \]

Alexander Tiskin (Warwick) Efficient Parallel Algorithms 75 / 185
Basic parallel algorithms

Discussion

Typically, realistic slackness requirements: \( n \gg p \)

Costs \( \text{comp}, \text{comm}, \text{sync} \): functions of \( n, p \)

The goals:

- \( \text{comp} = \text{comp}_{\text{opt}} = \text{comp}_{\text{seq}} / p \)
- \( \text{comm} \) scales down with increasing \( p \)
- \( \text{sync} \) is a function of \( p \), independent of \( n \)

The challenges:

- efficient (optimal) algorithms
- good (sharp) lower bounds

Further parallel algorithms

List contraction and colouring

Linked list: \( n \) nodes, each contains data and a pointer to successor

Let \( \cdot \) be an associative operator, computable in time \( O(1) \)

Primitive list operation: pointer jumping

The original node data \( a, b \) and the pointer to \( b \) are kept, so that the pointer jumping operation can be reversed

Abstract view: node merging, allows e.g. for bidirectional links

The original \( a, b \) are kept implicitly, so that node merging can be reversed

The list contraction problem: reduce the list to a single node by successive merging (note the result is independent on the merging order)

The list expansion problem: restore the original list by reversing the contraction
Further parallel algorithms
List contraction and colouring

Application: list ranking

The problem: for each node, find its rank (distance from the head) by list contraction

Note the solution should be independent of the merging order!

Application: list ranking (contd.)

With each intermediate node during contraction/expansion, associate the corresponding contiguous sublist in the original list

Contraction phase: for each node keep the length of its sublist
Initially, each node assigned 1
Merging operation: $k, l \rightarrow k + l$
In the fully contracted list, the node contains value $n$

Application: list prefix sums

Initially, each node $i$ contains value $a_i$

Let $\cdot$ be an associative operator with identity $\epsilon$

The problem: for each node $i$, find $a_0; i = a_0 \cdot a_1 \cdot \cdots \cdot a_i$ by list contraction

Note the solution should be independent of the merging order!
Further parallel algorithms
List contraction and colouring

Application: list prefix sums (contd.)

With each intermediate node during contraction/expansion, associate the corresponding contiguous sublist in the original list.

Contraction phase: for each node keep the $\bullet$-sum of its sublist.
Initially, each node assigned $a_i$.

Merging operation: $u, v \rightarrow u \bullet v$
In the fully contracted list, the node contains value $b_{n-1}$.

Expansion phase: for each node keep
- the $\bullet$-sum of all nodes before its sublist
- the $\bullet$-sum of its sublist

Initially, the node (fully contracted list) assigned $(\epsilon, b_{n-1})$

Un-merging operation: $(t, u, v) \rightarrow (t, u \bullet v)$
In the fully expanded list, a node with rank $i$ contains $(b_{i-1}, a_i)$
We have $b_i = b_{i-1} \bullet a_i$

Further parallel algorithms
List contraction and colouring

From now on, we only consider pure list contraction (the expansion phase is obtained by symmetry).
Sequential work $O(n)$ by always contracting at the list's head.
Parallel list contraction must be based on local merging decisions: a node can be merged with either its successor or predecessor, but not with both simultaneously.
Therefore, we need either node splitting, or efficient symmetry breaking.

Wyllie's mating

[Wyllie: 1979]

Split every node into “forward” node $\bigcirc$, and “backward” node $\bigcirc$.

Merge mating node pairs, obtaining two lists of size $\approx n/2$. 

Efficient Parallel Algorithms
87 / 185
Further parallel algorithms
List contraction and colouring

Parallel list contraction by Wyllie’s mating

Initially, each processor reads a subset of \( n/p \) nodes

A node merge involves communication between the two corresponding processors; the merged node is placed arbitrarily on either processor

- reduce the original list to \( n \) fully contracted lists by \( \log n \) rounds of Wyllie’s mating; after each round, the current reduced lists are written back to external memory
- select one fully contracted list

Total work \( O(n \log n) \), not optimal vs. sequential work \( O(n) \)

\[
\text{comp } O\left(\frac{n \log n}{p}\right) \quad \text{comm } O\left(\frac{n \log n}{p}\right) \quad \text{sync } O(\log n) \quad n \geq p
\]

Random mating

[Miller, Reif: 1985]

Label every node either “forward” \( \bigstar \), or “backward” \( \bullet \)

For each node, labelling independent with probability 1/2

\[
\bigstar \rightarrow \bigstar \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bigstar \rightarrow \bigstar
\]

A node mates with probability 1/2, hence on average \( n/2 \) nodes mate

Merge mating node pairs, obtaining a new list of expected size \( 3n/4 \)

More precisely, \( \text{Prob}(\text{new size } \leq 15n/16) \geq 1 - e^{-n/64} \)

Block mating

Will mate nodes deterministically

Contract local chains (if any)

Build distribution graph:

- complete weighted digraph on \( p \) supernodes
- \( w(i, j) = |\{ u \rightarrow v : u \in \text{proc}_i, v \in \text{proc}_j \}| \)

Each processor holds a supernode’s outgoing edges

\( n \geq p^2 \cdot \log p \)
Further parallel algorithms
List contraction and colouring

Block mating (contd.)
Collect distribution graph in a designated processor
Label every supernode “forward” F or “backward” B, so that \( \sum_{i \in F, j \in B} w(i, j) \geq \frac{1}{4} \cdot \sum_{i, j} w(i, j) \)
by a sequential greedy algorithm
Scatter supernode labels to processors
By construction of supernode labelling, at least \( n/2 \) nodes have mates
Merge mating node pairs, obtaining a new list of size at most \( 3n/4 \)

Parallel list contraction by block mating
Initially, each processor reads a subset of \( n/p \) nodes
- reduce list to size \( n/p \) by \( \log_{4/3} p \) rounds of block mating
- collect the reduced list in a designated processor and contract sequentially
Total work \( O(n) \), optimal and deterministic

Deterministic coin tossing [Cole, Vishkin: 1986]
Given a \( k \)-colouring, \( k > 6 \); colours represented in binary
Consider every node \( v \). We have \( col(v) \neq col(next(v)) \).
If \( col(v) \) differs from \( col(next(v)) \) in \( i \)-th bit, re-colour \( v \) in
- \( 2i \), if \( i \)-th bit in \( col(v) \) is 0, and in \( col(next(v)) \) is 1
- \( 2i + 1 \), if \( i \)-th bit in \( col(v) \) is 1, and in \( col(next(v)) \) is 0
After re-colouring, still have \( col(v) \neq col(next(v)) \)
Number of colours reduced from \( k \) to \( 2 \log k \ll k \)
Further parallel algorithms
List contraction and colouring

Parallel list 3-colouring by deterministic coin tossing:
- compute a $p$-colouring
- reduce the number of colours from $p$ to 6 by deterministic coin tossing: $O(\log^* k)$ rounds
  \[
  \log^* k = \min r : \log \ldots \log k \leq 1 \quad (r \text{ times})
  \]
- select node $v$ as a pivot, if $\text{col}(\text{prev}(v)) > \text{col}(v) < \text{col}(\text{next}(v))$. No two pivots are adjacent or further than 12 nodes apart
- from each pivot, re-colour the succeeding run of at most 12 non-pivots sequentially in 3 colours

\[
\text{comp } O(n/p) \quad \text{comm } O(n/p) \quad \text{sync } O(\log^* p)
\]

Further parallel algorithms
Sorting

Parallel sorting by regular sampling [Shi, Schaeffer: 1992]

Every processor
- reads a subarray of size $n/p$ and sorts it sequentially
- selects from its subarray $p$ samples at regular intervals

A designated processor
- collects all $p^2$ samples and sorts them sequentially
- selects from the sorted samples $p$ splitters at regular intervals

$a = [a_0, \ldots, a_{n-1}]$

The sorting problem: arrange elements of $a$ in increasing order
May assume all $a_i$ are distinct (otherwise, attach unique tags)
Assume the comparison model: primitives $<, >$, no bitwise operations
Sequential work $O(n \log n)$ e.g. by mergesort
Parallel sorting based on an AKS sorting network

\[
\begin{align*}
\text{comp } O\left(\frac{n \log n}{p}\right) & \quad \text{comm } O\left(\frac{n \log n}{p}\right) & \quad \text{sync } O(\log n)
\end{align*}
\]
Further parallel algorithms
Sorting

Parallel sorting by regular sampling (contd.)
The designated processor broadcasts the splitters
Every processor
- receives the splitters and is assigned a bucket
- scans its subarray and sends each element to the appropriate bucket
- receives the elements of its bucket and sorts them sequentially
- writes the sorted bucket back to external memory

Claim: each bucket has size \( \leq \frac{2n}{p} \)

Proof (sketch). Relative to a fixed bucket \( B \), a block \( b \) is low (respectively high), if lower boundary of \( b \) is \( \leq \) (respectively \( > \)) lower boundary of \( B \)
A bucket can intersect \( \leq p \) low blocks and \( \leq p \) high blocks
Bucket size is at most \((p + p) \cdot \frac{n}{p^2} = \frac{2n}{p}\)

\[
\begin{align*}
\text{comp } & O\left(\frac{n \log n}{p}\right) \quad \text{comm } O(n/p) \quad \text{sync } O(1) \quad n \geq p^3
\end{align*}
\]

Further parallel algorithms
Convex hull

Set \( S \subseteq \mathbb{R}^d \) is convex, if for all \( x, y \) in \( S \), every point between \( x \) and \( y \) is also in \( S \)
\( A \subseteq \mathbb{R}^d \)
The convex hull \( \text{conv} A \) is the smallest convex set containing \( A \)
\( \text{conv} A \) is a polytope, defined by its vertices \( A_i \in A \)
Set \( A \) is in convex position, if every its point is a vertex of \( \text{conv} A \)

\[ a = [a_0, \ldots, a_{n-1}] \quad a_i \in \mathbb{R}^d \]
The (discrete) convex hull problem: find vertices of \( \text{conv} a \)
Output must be ordered: every vertex must “know” its neighbours
Claim: Convex hull problem in \( \mathbb{R}^2 \) is at least as hard as sorting
Proof. Let \( x_0, \ldots, x_{n-1} \in \mathbb{R} \)
To sort \([x_0, \ldots, x_{n-1}]\):
- compute \( \text{conv}\{ (x_i, x_i^2) \} \in \mathbb{R}^2 : 0 \leq i < n \}
- follow the neighbour links to obtain sorted output
Further parallel algorithms

Convex hull

The discrete convex hull problem

\( d = 2 \): two neighbours per vertex; output size \( 2n \)
\( d = 3 \): on average, \( O(1) \) neighbours per vertex; output size \( O(n) \)
Sequential work \( O(n \log n) \) by Graham’s scan or by mergehull

\( d > 3 \): typically, a lot of neighbours per vertex; output size \( \gg \Omega(n) \)

From now on, will concentrate on \( d = 2, 3 \)

Claim. An \( \epsilon \)-approximation for \( A \) is an \( \epsilon \)-net for \( A \)
Claim. Union of \( \epsilon \)-approximations for \( A', A'' \) is \( \epsilon \)-approximation for \( A' \cup A'' \)
Claim. An \( \epsilon \)-net for a \( \delta \)-approximation for \( A \) is an \( (\epsilon + \delta) \)-net for \( A \)
Proofs: Easy by definitions.

\( d = 2 \quad A \subseteq \mathbb{R}^2 \quad |A| = n \quad \epsilon = 1/r \)
Claim. A 1/r-net for \( A \) of size \( \leq 2r \) exists and can be computed in sequential work \( O(n \log n) \).
Proof. Consider convex hull of \( A \) and an arbitrary interior point \( v \)
Partition \( A \) into triangles: base at a hull edge, apex at \( v \)
A triangle is heavy if it contains \( > n/r \) points of \( A \), otherwise light
Heavy triangles: for each triangle, take both hull vertices
Light triangles: for each triangle chain, greedy next-fit bin packing
- combine adjacent triangles into bins with \( \leq n/r \) points
- for each bin, take both boundary hull vertices
In total \( \leq 2r \) heavy triangles and bins, hence taken \( \leq 2r \) points
Further parallel algorithms
Convex hull

\[ d = 2 \quad A \subseteq \mathbb{R}^2 \quad |A| = n \quad \epsilon = 1/r \]

Claim. If \( A \) is in convex position, then a \( 1/r \)-approximation for \( A \) of size \( \leq r \) exists and can be computed in sequential work \( O(n \log n) \).

Proof. Take every \( n/r \)-th point on the convex hull of \( A \).

Parallel 2D hull computation by generalised regular sampling

\[ a = [a_0, \ldots, a_{n-1}] \quad a_i \in \mathbb{R}^2 \]

Every processor
- reads a subset of \( n/p \) points, computes its hull, discards the rest
- selects \( p \) samples at regular intervals on the hull

Set of all samples: \( 1/p \)-approximation for set \( a \) (after discarding local interior points)

A designated processor
- collects all \( p^2 \) samples (and does not compute its hull)
- selects from the samples a \( 1/p \)-net of \( \leq 2p \) points as splitters

Set of splitters: \( 1/p \)-net for samples, therefore a \( 2/p \)-net for set \( a \)

Parallel 2D hull computation by generalised regular sampling (contd.)

The \( 2p \) splitters can be assumed to be in convex position (like any \( \epsilon \)-net), and therefore define a splitter polygon with at most \( 2p \) edges

Each edge of splitter polytope defines a bucket: the subset of set \( a \) visible when sitting on this edge (assuming the polygon is opaque)

Each bucket can be covered by two half-planes not containing any splitters. Therefore, bucket size is at most \( 2 \cdot (2/p) \cdot n = 4n/p \).

The designated processor broadcasts the splitters

Every processor
- receives the splitters and is assigned 2 buckets
- scans its hull and sends each point to the appropriate bucket
- receives the points of its buckets and computes their hulls sequentially
- writes the bucket hulls back to external memory

\[ \text{comp } O\left(\frac{n \log n}{p}\right) \quad \text{comm } O(n/p) \quad \text{sync } O(1) \]

\( n \geq p^3 \)
Further parallel algorithms
Convex hull

\(d = 3 \quad A \subseteq \mathbb{R}^3 \quad |A| = n \quad \epsilon = 1/r\)

Claim. A \(1/r\)-net for \(A\) of size \(O(r)\) exists and can be computed in sequential work \(O(nn \log n)\).

Proof: \cite{Brönnimann, Goodrich: 1995}

Better approximations are possible, but are slower to compute.

Parallel 3D hull computation by generalised regular sampling

\(a = [a_0, \ldots, a_{n-1}] \quad a_i \in \mathbb{R}^3\)

Every processor
- reads a subset of \(n/p\) points
- selects a \(1/p\)-approximation of \(O(p^3 (\log p)^{O(1)})\) points as \textit{samples}

Set of all samples: \(1/p\)-approximation for set \(a\)

A designated processor
- collects all \(O(p^4 (\log p)^{O(1)})\) samples
- selects from the samples a \(1/p\)-net of \(O(p)\) points as \textit{splitters}

Set of splitters: \(1/p\)-net for samples, therefore a \(2/p\)-net for set \(a\)

Parallel 3D hull computation by generalised regular sampling (contd.)

The \(O(p)\) splitters can be assumed to be in convex position (like any \(\epsilon\)-net), and therefore define a 
\textit{splitter polytope} with \(O(p)\) edges

Each edge of splitter polytope defines a \textit{bucket}: the subset of a visible
when sitting on this edge (assuming the polytope is opaque)

Each bucket can be covered by two half-planes not containing any splitters.
Therefore, bucket size is at most \(2 \cdot (2/p) \cdot n = 4n/p\).

Alexander Tiskin (Warwick) Efficient Parallel Algorithms 114 / 185
Further parallel algorithms
Selection

\[ a = [a_0, \ldots, a_{n-1}] \]

The selection problem: given \( k \), find \( k \)-th smallest element of \( a \)

E.g. median selection: \( k = n/2 \)

As before, assume the comparison model

Sequential work \( O(n \log n) \) by naive sorting

Sequential work \( O(n) \) by successive elimination [Blum+: 1973]

Claim. Each elimination removes \( \geq \) a fraction of \( 3/10 \) of elements of \( a \)

Proof (sketch). In half of all subarrays, the subarray median is on the “wrong” side of the median-of-medians \( a_l \). In every such subarray, two off-median subarray elements are on the “wrong” side of the subarray median. Hence, in a round, at least a fraction of \( 1/2 \cdot (1 + 2)/5 = 3/10 \) elements are eliminated.

Each round removes at least a constant fraction of elements of \( a \)

Data reduction rate is exponential

Further parallel algorithms
Selection

Standard approach to selection: eliminate elements in rounds

In each round:

- partition array \( a \) into subarrays of size 5
- select median in each subarray
- select median of subarray medians by recursion: \((n, k) \leftarrow (n/5, n/10)\)
- find rank \( l \) of median-of-medians in array \( a \)
- if \( l = k \), we are done
- if \( l < k \): eliminate all \( a_i \) that are \( \leq a_l \); in next round, set \( k \leftarrow k - l \)
- if \( l > k \): eliminate all \( a_i \) that are \( \geq a_l \); in next round, \( k \) unchanged

Each time, we eliminate elements on “wrong” side of median-of-medians \( a_l \)

More general approach: elimination by regular sampling in rounds

In each round:

- partition array \( a \) into subarrays
- select a set of regular samples in each subarray
- select a subset of regular splitters from the set of all samples

Selecting samples and splitters:

- if subarray (resp. set of all samples) is small, then we just sort it
- otherwise, we select samples (respectively, splitters) by recursion, without pre-sorting

In standard approach: \( O(n) \) subarrays, each of size \( O(1) \); 3 samples per subarray (median + boundaries); 3 splitters (m-of-ms + boundaries)
Further parallel algorithms
Selection

Elimination by regular sampling (contd.)
Let \( a_l^- \), \( a_l^+ \) be adjacent splitters, such that \( l^- \leq k \leq l^+ \)
Splitters \( a_l^- \), \( a_l^+ \) define the bucket

- eliminate all \( a_i \) outside the bucket

For work-optimality, sufficient to use constant subarray size and constant sampling frequency (as in standard approach)
Since the array size decreases in every round, we can increase the sampling frequency to reduce the number of rounds, while keeping work-optimality

Parallel selection by accelerated regular sampling
Main idea: variable sampling frequency in different rounds. As array size decreases, we can afford to increase sampling frequency.
Data reduction rate now superexponential
Selection can be completed in \( O(\log \log p) \) rounds

- reduce the input array to size \( n/p \) by \( O(\log \log p) \) rounds of accelerated regular sampling (implicit load balancing);
- collect the reduced array in a designated processor and perform selection sequentially

\[
\text{comp } O(n/p) \quad \text{comm } O(n/p) \\
\text{sync } O(\log p) \\
\text{sync } O(\log \log n) \\
\text{sync } O(1) \quad \text{randomised whp}
\]  
[Ishimizu+: 2002]  
[Fujiwara+: 2000]  
[Gerbessiotis, Siniolakis: 2003]
Parallel matrix algorithms

Matrix-vector multiplication

Let $A$, $b$, $c$ be a matrix and two vectors of size $n$.

The matrix-vector multiplication problem

\[ A \cdot b = c \]

\[ c_i = \sum_{j} A_{ij} \cdot b_j \]

Overall, $n^2$ elementary products $A_{ij} \cdot b_j$

Sequential work $O(n^2)$

Alexander Tiskin (Warwick)

Efficient Parallel Algorithms

125 / 185

Parallel matrix algorithms

Matrix-vector multiplication

Parallel matrix-vector multiplication

Assume $A$ is pre-distributed across the processors as needed, does not count as input (motivation: iterative linear algebra methods).

Partition the $n \times n$ square into a regular grid of $p = p_1^2 \cdot p_2^2$ square blocks.

Matrix $A$ gets partitioned into $p$ square blocks $A_{ij}$ of size $n/p_1^2$.

Vectors $b$, $c$ each gets partitioned into $p_1$ linear blocks $b_{i}$, $c_{i}$ of size $n/p_1$.

Overall, $p^2$ elementary products $A_{ij} \cdot b_{j}$.

Sequential work $O(n^2)$

Alexander Tiskin (Warwick)

Efficient Parallel Algorithms

125 / 185

Parallel matrix algorithms

Matrix-vector multiplication

Parallel matrix-vector multiplication (contd.)

\[ c_i \leftarrow 0 \]

\[ c_i \leftarrow A_{i_1} \cdot b_{j_1} \]

\[ 0 \leq i_1, j_1 < n \]

An $n \times n$-square of nodes, each representing an elementary product of size $O(1)$, depth $O(1)$.

Add to $c_i$, asynchronously

Overall, $p^2$ elementary products $A_{ij} \cdot b_{j}$.

Sequential work $O(n^2)$

Alexander Tiskin (Warwick)

Efficient Parallel Algorithms

126 / 185

Parallel matrix algorithms

Matrix-vector multiplication

The matrix-vector multiplication circuit

$A \cdot b = c$

$0 \leq i, j < n$

$A_{ij}$

$b$

$c$

Overall, $p^2$ elementary products $A_{ij} \cdot b_{j}$

Sequential work $O(n^2)$

Alexander Tiskin (Warwick)

Efficient Parallel Algorithms

128 / 185
Parallel matrix algorithms
Matrix-vector multiplication

Parallel matrix-vector multiplication (contd.)
Vector $c$ in external memory is initialised by zero values

Every processor
- is assigned to compute a block product $A_{IJ} \cdot b_J = c_I^J$
- reads block $b_J$ and computes $c_I^J$
- updates $c_I$ in external memory by adding $c_I^J$ elementwise

Updates to $c_I$ add up (asynchronously) to its correct final value

$$\text{comp } O\left(\frac{n^2}{p}\right) \quad \text{comm } O\left(\frac{n}{p^{1/3}}\right) \quad \text{sync } O(1) \quad n \geq p$$

Parallel matrix algorithms
Matrix multiplication

Let $A$, $B$, $C$ be matrices of size $n$
The matrix multiplication problem

$$A \cdot B = C$$
$$C_{ik} = \sum_j A_{ij} \cdot B_{jk}$$
$$0 \leq i, j, k < n$$

Overall, $n^3$ elementary products $A_{ij} \cdot B_{jk}$
Sequential work $O(n^3)$

Parallel matrix algorithms
Matrix multiplication

The matrix multiplication circuit

$$C_{ik} \leftarrow 0$$
$$C_{ik} \leftarrow A_{ij} \cdot B_{jk}$$
$$0 \leq i, j, k < n$$

An $ijk$-cube of nodes, each representing an elementary product size $O(n^3)$, depth $O(1)$

Parallel matrix multiplication
Partition the $ijk$-cube into a regular grid of $p = p^{1/3} \cdot p^{1/3} \cdot p^{1/3}$ cubic blocks
Matrices $A$, $B$, $C$ each gets partitioned into $p^{2/3}$ square blocks $A_{IJ}$, $B_{JK}$, $C_{IK}$ of size $n/p^{1/3}$
$$0 \leq I, J, K < p^{1/3}$$
Parallel matrix algorithms
Matrix multiplication

Parallel matrix multiplication (contd.)

\[ C_{IK} \leftarrow 0 \]

\[ C_{IK} \leftarrow A_{IJ} \cdot B_{JK} \]

\[ 0 \leq I, J, K < p^{1/3} \]

Matrix \( C \) in external memory is initialised by zero values

Every processor

- is assigned to compute a block product \( A_{IJ} \cdot B_{JK} = C'_{IK} \)
- reads blocks \( A_{IJ}, B_{JK} \), and computes \( C'_{IK} \)
- updates \( C_{IK} \) in external memory by adding \( C'_{IK} \) elementwise

Updates to \( C_{IK} \) add up (asynchronously) to its correct final value

\[
\text{comp } O\left(\frac{n^3}{p}\right) \quad \text{comm } O\left(\frac{n^2}{p^{1/3}}\right) \quad \text{sync } O(1)
\]

\[ n \geq p^{1/2} \]

Theorem. Computing the matrix multiplication dag requires communication \( \Omega\left(\frac{n^3}{p^{1/3}}\right) \) per processor

Proof. \( \text{comp } O\left(\frac{n^3}{p}\right), \text{sync } O(1) \) trivially optimal

Optimality of \( \text{comm } O\left(\frac{n^2}{p^{1/3}}\right) \): (discrete) volume vs surface area

Let \( V \) be the subset of \( ijk \)-cube computed by a certain processor

For at least one processor: \( |V| \geq \frac{n^3}{p} \)

Let \( A, B, C \) be projections of \( V \) onto coordinate planes

Arithmetic vs geometric mean: \( |A| + |B| + |C| \geq 3(|A| \cdot |B| \cdot |C|)^{1/3} \)

Loomis–Whitney inequality: \( |A| \cdot |B| \cdot |C| \geq |V|^2 \)

We have \( \text{comm} \geq |A| + |B| + |C| \geq 3(|A| \cdot |B| \cdot |C|)^{1/3} \geq 3|V|^{2/3} \geq 3\left(\frac{n^3}{p}\right)^{2/3} = \frac{3n^2}{p^{1/3}}, \) hence \( \text{comm} = \Omega\left(\frac{n^2}{p^{1/3}}\right) \)
Parallel matrix algorithms
Fast matrix multiplication

Recursive block matrix multiplication: \( A \cdot B = C \)

\[
A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \quad B = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} \quad C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}
\]

\[
C_{00} = A_{00} \cdot B_{00} + A_{01} \cdot B_{10} \\
C_{01} = A_{00} \cdot B_{01} + A_{01} \cdot B_{11} \\
C_{10} = A_{10} \cdot B_{00} + A_{11} \cdot B_{10} \\
C_{11} = A_{10} \cdot B_{01} + A_{11} \cdot B_{11}
\]

8 block multiplications (recursive calls)

Strassen-like matrix multiplication: \( A \cdot B = C \)

Main idea: for certain matrix sizes \( N \), we can multiply \( N \times N \) matrices using \( R < N^3 \) elementary products (\( \cdot \)) and linear operations (+, −):

- some linear operations on elements of \( A \)
- some linear operations on elements of \( B \)
- \( R \) elementary products of the resulting linear combinations
- some more linear operations to obtain \( C \)

Let \( \omega = \log_N R < \log_N N^3 = 3 \)

Strassen’s matrix multiplication: \( A \cdot B = C \)

Let \( A, B, C \) be numerical matrices: primitives +, −, \( \cdot \) on matrix elements

\[
A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \quad B = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} \quad C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}
\]

\[
D^{(0)} = (A_{00} + A_{11}) \cdot (B_{00} + B_{11}) \\
D^{(1)} = (A_{10} + A_{11}) \cdot B_{00} \\
D^{(2)} = A_{00} \cdot (B_{01} - B_{11}) \\
D^{(3)} = A_{11} \cdot (B_{10} - B_{00}) \\
D^{(4)} = (A_{00} + A_{11}) \cdot B_{11} \\
D^{(5)} = (A_{10} - A_{00}) \cdot (B_{00} + B_{01}) \\
D^{(6)} = (A_{01} - A_{11}) \cdot (B_{10} + B_{11}) \\
C_{00} = D^{(0)} + D^{(3)} - D^{(4)} + D^{(6)} \\
C_{01} = D^{(1)} + D^{(3)} \\
C_{10} = D^{(0)} - D^{(1)} + D^{(2)} + D^{(5)} \\
C_{11} = D^{(0)} - D^{(1)} + D^{(2)} + D^{(5)}
\]

7 block multiplications (recursive calls)
Parallel matrix algorithms
Fast matrix multiplication

Some specific instances of Strassen-like scheme:

\[
\begin{array}{cccc}
N & N^3 & R & \omega = \log_N R \\
2 & 8 & 7 & 2.81 \quad \text{[Strassen: 1969]} \\
3 & 27 & 23 & 2.85 \\
5 & 125 & 100 & 2.86 \\
48 & 110592 & 47216 & 2.78 \\
\end{array}
\]

- Huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge huge high
Theorem. Computing the Strassen-like matrix multiplication dag requires communication $\Omega\left(\frac{n^2}{p^{2/\omega}}\right)$ per processor.

Proof. By graph expansion, generalises the Loomis-Whitney inequality \[\text{[Ballard+:2012]}\]

Parallel Boolean matrix multiplication

The following algorithm is impractical, but of theoretical interest, since it beats the generic Loomis-Whitney communication lower bound.

Regularity Lemma: in a Boolean matrix, the rows and the columns can be partitioned into $K$ (almost) equal-sized subsets, so that $K^2$ resulting submatrices are random-like (of various densities) \[\text{[Szemerédi: 1978]}\]

$K = K(\epsilon)$, where $\epsilon$ is the “degree of random-likeness”;

Function $K(\epsilon)$ grows enormously as $\epsilon \to 0$, but is independent of $n$.

We shall call this the regular decomposition of a Boolean matrix.

Parallel Boolean matrix multiplication (contd.)

If $A$, $B$, $C$ random-like, then either $A$ or $B$ has few 1s, or $C$ has few 0s.

Equivalently, $A \land B = \overline{C}$, either $A$, $B$ or $C$ has few 1s.

By Regularity Lemma, we have the three-way regular decomposition:

- $A^{(1)} \land B^{(1)} = \overline{C^{(1)}}$, where $A^{(1)}$ has few 1s
- $A^{(2)} \land B^{(2)} = \overline{C^{(2)}}$, where $B^{(2)}$ has few 1s
- $A^{(3)} \land B^{(3)} = \overline{C^{(3)}}$, where $C^{(3)}$ has few 1s
- $\overline{C} = \overline{C^{(1)}} \lor \overline{C^{(2)}} \lor \overline{C^{(3)}}$

Matrices $A^{(1)}$, $A^{(2)}$, $A^{(3)}$, $B^{(1)}$, $B^{(2)}$, $B^{(3)}$, $C^{(1)}$, $C^{(2)}$, $C^{(3)}$ can be “efficiently” computed from $A$, $B$, $C$. 
Parallel matrix algorithms
Boolean matrix multiplication

Parallel Boolean matrix multiplication (contd.)
\[ A \land B = \overline{C} \]
Partition the \(ijk\)-cube into a regular grid of \(p^3 = p \cdot p \cdot p\) cubic blocks
Matrices \(A, B, \overline{C}\) each gets partitioned into \(p^2\) square blocks \(A_{ij}, B_{jk}, \overline{C}_{ik}\) of size \(n/p\)
\(0 \leq I, J, K < p\)

Every processor
- assigned to compute a “slab” of \(p^2\) cubic blocks \(A_{ij} \land B_{jk} = \overline{C}_{ik}^J\) for a fixed \(J\) and all \(I, K\)
- reads blocks \(A_{ij}, B_{jk}\) and computes \(\overline{C}_{ij}^J\) for all \(I, K\)
- computes the three-way regular decomposition for the block product and determines the submatrices having very 1s
\(0 \leq I, J, K < p\)

Recompute \(A \land B = \overline{C}\) from block regular decompositions by a Strassen-like algorithm
Communication saved by only sending the positions of 1s
\[
\begin{align*}
\text{comp } O(\frac{n^\omega}{p}) & \quad \text{comm } O\left(\frac{n^2}{p}\right) & \quad \text{sync } O(1) & \quad n \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow p & :/ \\
\end{align*}
\]

Parallel matrix algorithms
Triangular system solution

Let \(L, b, c\) be a matrix and two vectors of size \(n\)
\(L\) is lower triangular: \(L_{ij} = \begin{cases} 0 & 0 \leq i < j < n \\ \text{arbitrary} & \text{otherwise} \end{cases}\)
\(L \cdot b = c\)
\[ \sum_j L_{ij} \cdot b_j = c_i \]
\(0 \leq j \leq i < n\)
The triangular system problem: given \(L, c,\) find \(b\)
Forward substitution

\[ L \cdot b = c \]

\[ L_{00} \cdot b_0 = c_0 \]
\[ L_{10} \cdot b_0 + L_{11} \cdot b_1 = c_1 \]
\[ L_{20} \cdot b_0 + L_{21} \cdot b_1 + L_{22} \cdot b_2 = c_2 \]
\[ \ldots \]
\[ \sum_{j \leq i} L_{ij} \cdot b_j = c_i \]
\[ \ldots \]
\[ \sum_{j \leq n-1} L_{n-1,j} \cdot b_j = c_{n-1} \]

Sequential work \( O(n^2) \)

Symmetrically, an upper triangular system solved by back substitution

Block forward substitution

\[ L \cdot b = c \]

\[ \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \]

Recursion: two half-sized subproblems

\[ L_{00} \cdot b_0 = c_0 \text{ by recursion} \]
\[ L_{11} \cdot b_1 = c_1 - L_{10} \cdot b_0 \text{ by recursion} \]

Sequential work \( O(n^2) \)
Parallel block forward substitution (contd.)

Recursion levels 0 to \( \log p \): block forward substitution using parallel matrix-vector multiplication

Recursion level \( \log p \): a designated processor reads the current task’s input, performs the task sequentially, and writes back the task’s output

\[
\text{comp} = O(n^2/p) \cdot (1 + 2 \cdot (\frac{1}{2}^2) + 2^2 \cdot (\frac{1}{2^4}^2) + \ldots) + O((n/p)^2) \cdot p = O(n^2/p) + O(n^2/p) = O(n^2/p)
\]

\[
\text{comm} = O(n/p^{1/2}) \cdot (1 + 2 \cdot \frac{1}{2} + 2^2 \cdot \frac{1}{2^2} + \ldots) + O(n/p) \cdot p = O(n/p^{1/2}) \cdot \log p + O(n) = O(n)
\]

\[
\text{comp} \ O(n^2/p) \quad \text{comm} \ O(n) \quad \text{sync} \ O(p)
\]

Parallel matrix algorithms

Gaussian elimination

Let \( A, L, U \) be matrices of size \( n \)

\( L \) is unit lower triangular:

\[
L_{ij} = \begin{cases} 
0 & 0 \leq i < j < n \\
1 & 0 \leq i = j < n \\
\text{arbitrary} & \text{otherwise}
\end{cases}
\]

\( U \) is upper triangular:

\[
U_{ij} = \begin{cases} 
0 & 0 \leq j < i < n \\
\text{arbitrary} & \text{otherwise}
\end{cases}
\]

\[
A = L \cdot U
\]

\[
A_{ik} = \sum_j L_{ij} \cdot U_{jk}
\]

\[
0 \leq k \leq j \leq i < n
\]

The LU decomposition problem: given \( A \), find \( L, U \)

Generic Gaussian elimination

\[
A = L \cdot U
\]

\[
\begin{bmatrix}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{bmatrix}
= 
\begin{bmatrix}
1 & 0 \\
L_{10} & L_{11}
\end{bmatrix}
\begin{bmatrix}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{bmatrix}
\begin{bmatrix}
1 & 0 \\
0 & U_{11}
\end{bmatrix}
\]

First step of elimination: pivot \( A_{00} \)

\[
\begin{bmatrix}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{bmatrix}
= 
\begin{bmatrix}
1 & 0 \\
L_{10} & I
\end{bmatrix}
\begin{bmatrix}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{bmatrix}
\begin{bmatrix}
1 & 0 \\
0 & A_{11}^{-1}
\end{bmatrix}
\]

\[
L_{10} \leftarrow A_{10} \cdot A_{00}^{-1} \\
A_{11}^{-1} \leftarrow A_{11}^{-1} - L_{10} \cdot A_{10}
\]

Continue elimination on reduced matrix \( A' \)

In every step, we assume \( A_{00} \neq 0 \) (no pivoting, only default pivots)

Sequential work \( O(n^3) \)

Parallel generic Gaussian elimination: 3D grid (details omitted)

\[
\text{comp} \ O(n^3/p) \quad \text{comm} \ O(n^2/p^{1/2}) \quad \text{sync} \ O(p^{1/2})
\]
Block generic Gaussian elimination

\[ A = L \cdot U \]

\[
\begin{bmatrix}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{bmatrix} = \begin{bmatrix}
L_{00} & 0 \\
L_{10} & L_{11}
\end{bmatrix} \begin{bmatrix}
U_{00} & U_{01} \\
0 & U_{11}
\end{bmatrix}
\]

Recursion: two half-sized subproblems

\[ A_{00} = L_{00} \cdot U_{00} \text{ by recursion} \]

\[ U_{01} \leftarrow L_{00}^{-1} \cdot A_{01} \quad L_{10} \leftarrow A_{10} \cdot U_{10}^{-1} \]

\[ A_{11} = L_{10} \cdot U_{01} = L_{11} \cdot U_{11} \text{ by recursion} \]

\[ L^{-1} \leftarrow \begin{bmatrix}
L_{00}^{-1} \\
L_{10}^{-1} L_{11}^{-1} L_{00}^{-1}
\end{bmatrix} \quad U^{-1} \leftarrow \begin{bmatrix}
U_{00}^{-1} & -U_{00}^{-1} U_{01} U_{10}^{-1} \\
0 & U_{11}^{-1}
\end{bmatrix} \]

Sequential work \( O(n^3) \), allows use of Strassen-like schemes

Parallel block generic Gaussian elimination (contd.)

Recursion levels 0 to \( \alpha \log p \): block generic LU decomposition using parallel matrix multiplication

Recursion level \( \alpha \log p \): on each visit, a designated processor reads the current task’s input, performs the task sequentially, and writes back the task’s output

Threshold level controlled by parameter \( \alpha \): \( 1/2 \leq \alpha \leq 2/3 \)

- \( \alpha \geq 1/2 \) needed for \( \text{comp} \)-optimality
- \( \alpha \leq 2/3 \) ensures total \( \text{comm} \) of threshold tasks is not more than the \( \text{comm} \) of top-level matrix multiplication

\[ \text{comp} \; O(n^3/p) \quad \text{comm} \; O(n^2/p^{\alpha}) \quad \text{sync} \; O(p^{\alpha}) \]

Parallel LU decomposition (contd.)

In particular:

\[ \alpha = 1/2 \]

\[ \text{comp} \; O(n^3/p) \quad \text{comm} \; O(n^2/p^{1/2}) \quad \text{sync} \; O(p^{1/2}) \]

Cf. 2D grid

\[ \alpha = 2/3 \]

\[ \text{comp} \; O(n^3/p) \quad \text{comm} \; O(n^2/p^{2/3}) \quad \text{sync} \; O(p^{2/3}) \]

Cf. matrix multiplication
Semiring: a set $S$ with addition $\oplus$ and multiplication $\odot$

Addition commutative, associative, has identity $0$

$$a \oplus b = b \oplus a \quad a \oplus (b \oplus c) = (a \oplus b) \oplus c \quad a \oplus 0 = 0 \oplus a = a$$

Multiplication associative, has annihilator $0$ and identity $1$

$$a \odot (b \odot c) = (a \odot b) \odot c \quad a \odot 0 = 0 \odot a = 0 \quad a \odot 1 = 1 \odot a = a$$

Multiplication distributes over addition

$$a \odot (b \odot c) = (a \odot b) \odot a \odot c \quad (a \odot b) \odot c = a \odot c \odot b \odot c$$

In general, no subtraction or division!

Given a semiring $S$, square matrices of size $n$ over $S$ also form a semiring:

- $\oplus$ given by matrix addition; $0$ by the zero matrix
- $\odot$ given by matrix multiplication; $1$ by the unit matrix

A semiring is closed, if

- an infinite sum $a_1 \oplus a_2 \oplus a_3 \oplus \cdots$ (e.g. a closure) is always defined
- such infinite sums are commutative, associative and distributive

In a closed semiring, every element and every square matrix have a closure

The numerical semiring is not closed: an infinite sum can be divergent

The Boolean semiring is closed: an infinite $\lor$ is 1, iff at least one term is 1

The tropical semiring is closed: an infinite $\min$ is the greatest lower bound

Where defined, these infinite sums are commutative, associative and distributive

### Some specific semirings:

<table>
<thead>
<tr>
<th>$S$</th>
<th>$\oplus$</th>
<th>$\ominus$</th>
<th>$\odot$</th>
<th>$\mathbb{1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>numerical</td>
<td>$\mathbb{R}$</td>
<td>$+$</td>
<td>0</td>
<td>$\cdot$ 1</td>
</tr>
<tr>
<td>Boolean</td>
<td>${0, 1}$</td>
<td>$\lor$</td>
<td>0</td>
<td>$\land$ 1</td>
</tr>
<tr>
<td>tropical</td>
<td>$\mathbb{R}_{\geq 0} \cup {+\infty}$</td>
<td>$\min$</td>
<td>$+\infty$</td>
<td>$+$ 0</td>
</tr>
</tbody>
</table>

We will occasionally write $ab$ for $a \odot b$, $a^2$ for $a \odot a$, etc.

The closure of $a$: $a^* = \overline{\ominus} a \oplus a^2 \oplus a^3 \oplus \cdots$

Numerical closure $a^* = 1 + a + a^2 + a^3 + \cdots = \begin{cases} \frac{1}{1-a} & \text{if } |a| < 1 \\ \text{undefined} & \text{otherwise} \end{cases}$

Boolean closure $a^* = 1 \lor a \lor a \lor a \lor \cdots = 1$

Tropical closure $a^* = \min(0, a, 2a, 3a, \ldots) = 0$

In matrix semirings, closures are more interesting
Parallel graph algorithms
Algebraic path problem

Let $A$ be a matrix of size $n$ over a semiring

The algebraic path problem: compute $A^* = I \oplus A \oplus A^2 \oplus A^3 \oplus \cdots$

Numerical algebraic path problem: equivalent to matrix inversion

$A^* = I + A + A^2 + \cdots = (I - A)^{-1}$, if defined

The algebraic path problem in a closed semiring: interpreted via a weighted digraph on $n$ nodes with adjacency matrix $A$

$A_{ij}$ = length of the edge $i \rightarrow j$

Boolean $A^*$: the graph’s transitive closure

Tropical $A^*$: the graph’s all-pairs shortest paths

**Floyd–Warshall algorithm** [Floyd, Warshall: 1962]

Works for any closed semiring; we assume tropical, all 0s on main diagonal

Weights may be negative; assume no negative cycles

First step of elimination: pivot $A_{00} = 0$

Replace each weight $A_{ij}$, $i, j \neq 0$, with $A_{i0} + A_{0j}$, if that gives a shortcut from $i$ to $j$

$A'_{11} \leftarrow A_{11} \oplus A_{10} \oplus A_{01} = \min(A_{11}, A_{10} + A_{01})$

Continue elimination on reduced matrix $A'_{11}$

Generic Gaussian elimination in disguise

Sequential work $O(n^3)$

Block Floyd–Warshall algorithm

$A = \begin{bmatrix} 0 & 5 & 10 & \infty & 10 \\ \infty & 0 & 3 & 2 & 9 \\ \infty & 2 & 0 & \infty & 1 \\ 7 & \infty & \infty & 0 & 6 \\ \infty & \infty & \infty & 4 & 0 \end{bmatrix}$

$A^* = \begin{bmatrix} 0 & 5 & 8 & 7 & 9 \\ 9 & 0 & 3 & 2 & 4 \\ 11 & 2 & 0 & 4 & 1 \\ 7 & 12 & 15 & 0 & 6 \\ 11 & 16 & 19 & 4 & 0 \end{bmatrix}$

Recursion: two half-sized subproblems

$A'_{00} \leftarrow A_{00}$ by recursion

$A'_{11} \leftarrow A_{11}A'_{10} \quad A'_{10} \leftarrow A_{10}A'_{00} \quad A'_{11} \leftarrow A_{11} + A_{10}A'_{00}A_{01}$

$A''_{11} \leftarrow (A'_{11})^*$ by recursion

$A''_{10} \leftarrow A''_{11}A'_{10} \quad A''_{10} \leftarrow A'_{10}A''_{11} \quad A''_{00} \leftarrow A''_{10} + A''_{11}A''_{10}$

Block generic Gaussian elimination in disguise

Sequential work $O(n^3)$
Parallel graph algorithms

Algebraic path problem

Parallel algebraic path computation
Similar to LU decomposition by block generic Gaussian elimination
Te recursion tree is unfolded depth-first
Recursion levels 0 to $\alpha \log p$: block Floyd–Warshall using parallel matrix multiplication
Recursion level $\alpha \log p$: on each visit, a designated processor reads the current task’s input, performs the task sequentially, and writes back the task’s output
Threshold level controlled by parameter $\alpha$: $1/2 \leq \alpha \leq 2/3$

$\text{comp } O(n^3/p) \quad \text{comm } O(n^2/p^{\alpha}) \quad \text{sync } O(p^{\alpha})$

In particular:

$\alpha = 1/2$

$\text{comp } O(n^3/p) \quad \text{comm } O(n^2/p^{1/2}) \quad \text{sync } O(p^{1/2})$

Cf. 2D grid

$\alpha = 2/3$

$\text{comp } O(n^3/p) \quad \text{comm } O(n^2/p^{2/3}) \quad \text{sync } O(p^{2/3})$

Cf. matrix multiplication

Alexander Tiskin (Warwick)  Efficient Parallel Algorithms  173 / 185

Parallel graph algorithms

All-pairs shortest paths

The all-pairs shortest paths problem: the algebraic path problem over the tropical semiring

$S = \mathbb{R}_{\geq 0} \cup \{+\infty\}$

<table>
<thead>
<tr>
<th>tropical</th>
<th>$\oplus$</th>
<th>$\ominus$</th>
<th>$\odot$</th>
<th>$\boxplus$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{R}_{\geq 0} \cup {+\infty}$</td>
<td>$\min$</td>
<td>$+\infty$</td>
<td>$+$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

We continue to use the generic notation: $\oplus$ for min, $\odot$ for $+$

To improve on the generic algebraic path algorithm, we must exploit the tropical semiring’s idempotence: $a \oplus a = \min(a, a) = a$

$
A_{ij} = \text{length of the edge } i \rightarrow j \\
A_{ij} \geq 0 \quad A_{ii} = \boxplus = 0 \quad 0 \leq i, j < n$

Path length: sum ($\odot$-product) of all its edge lengths

Path size: its total number of edges (by definition, $\leq n$)

$A^k_{ij} = \text{length of the shortest path } i \leadsto j \text{ of size } \leq k$

$A^k_{ij} = \text{length of the shortest path } i \leadsto j \text{ (of any size)}$

The all-pairs shortest paths problem:

$A^* = I \oplus A \oplus A^2 \oplus \cdots = I \oplus A \oplus A^2 \oplus \cdots \oplus A^n = (I \oplus A)^n = A^n$
Parallel graph algorithms
All-pairs shortest paths

Dijkstra’s algorithm [Dijkstra: 1959]

Computes single-source shortest paths from fixed source (say, node 0)
Ranks all nodes by distance from node 0: nearest, second nearest, etc.
Every time a node \( i \) has been ranked: replace each weight \( A_{0j} \), \( j \) unranked, with \( A_{0i} + A_{ij} \), if that gives a shortcut from 0 to \( j \)
Assign the next rank to the unranked node closest to node 0 and repeat

It is essential that the edge lengths are nonnegative

Sequential work \( O(n^2) \)

All-pairs shortest paths: multi-Dijkstra, i.e. running Dijkstra’s algorithm independently from every node as a source
Sequential work \( O(n^3) \)

Parallel all-pairs shortest paths by multi-Dijkstra

Every processor
- reads matrix \( A \) and is assigned a subset of \( n/p \) nodes
- runs \( n/p \) independent instances of Dijkstra’s algorithm from its assigned nodes
- writes back the resulting \( n^2/p \) shortest distances

\[
\begin{array}{ccc}
\text{comp } O(\frac{n^3}{p}) & \text{comm } O(n^2) & \text{sync } O(1)
\end{array}
\]

Parallel all-pairs shortest paths: summary so far

\[
\begin{array}{ccc}
\text{comp } O(\frac{n^3}{p}) & \text{comm } O(\frac{n^2}{p^{2/3}}) & \text{sync } O(\frac{p^{2/3}}{3})
\end{array}
\]

Floyd–Warshall, \( \alpha = 2/3 \)

\[
\begin{array}{ccc}
\text{comp } O(\frac{n^2}{p^{1/2}}) & \text{sync } O(\frac{p^{1/2}}{3})
\end{array}
\]

Floyd–Warshall, \( \alpha = 1/2 \)

\[
\begin{array}{cc}
\text{comp } O(n^2) & \text{sync } O(1)
\end{array}
\]

Multi-Dijkstra

\[
\begin{array}{cc}
\text{comp } O(n^2) & \text{sync } O(1)
\end{array}
\]

Coming next

\[
\begin{array}{cc}
\text{comp } O(\frac{n^2}{p^{2/3}}) & \text{sync } O(\log p)
\end{array}
\]

Path doubling

Compute \( A, A^2, A^4 = (A^2)^2, A^8 = (A^4)^2, \ldots, A^n = A^* \)
Overall, \( \log n \) rounds of matrix \( \odot \)-multiplication: looks promising...
Sequential work \( O(n^3 \log n) \): not work-optimal!
Parallel graph algorithms
All-pairs shortest paths

Selective path doubling

Idea: to remove redundancy in path doubling by keeping track of path sizes
Assume we already have \( A^k \). The next round is as follows.

Let \( A_{ij}^k \) = length of the shortest path \( i \leadsto j \) of size \( \leq k \)
Let \( A_{ij}^\leq k \) = length of the shortest path \( i \leadsto j \) of size exactly \( k \)

We have \( A^k = A^\leq k = A^0 \oplus \cdots \oplus A^k \).

Consider \( A^0, \ldots, A^k \). The total number of non-zero elements in these matrices is at most \( n^2 \), on average \( \frac{2n^2}{k} \) per matrix. Hence, for some \( l \leq \frac{k}{2} \), matrix \( A^{\frac{k}{2}+l} \) has at most \( \frac{2n^2}{k} \) non-zero elements.

Compute \( (I + A^{\frac{k}{2}+l}) \odot A^\leq k = A^{\leq \frac{3k}{2}+l} \). This is a sparse-by-dense matrix product, requiring at most \( \frac{2n^2}{k} \cdot n = \frac{2n^3}{k} \) elementary multiplications.

Parallel all-pairs shortest paths by selective path doubling

All processors compute \( A, A^\leq \frac{3}{2} \cdots, A^p \cdots \) by \( \leq \log_{3/2} p \) rounds of parallel sparse-by-dense matrix \( \odot \)-multiplication

Consider \( A=0, \ldots, A=p \). The total number of non-zero elements in these matrices is at most \( n^2 \), on average \( \frac{n^2}{p} \) per matrix. Hence, for some \( q \leq \frac{p}{2} \), matrices \( A^q \) and \( A^{p-q} \) have together at most \( \frac{2n^2}{p} \) non-zero elements.

Every processor reads \( A^q \) and \( A^{p-q} \) and computes \( A^q \odot A^{p-q} = A^p \)

All processors compute \((A^p)^*\) by parallel multi-Dijkstra, and then \((A^p)^* \odot A^p = A^p\) by parallel matrix \( \odot \)-multiplication

Use of multi-Dijkstra requires that all edge lengths in \( A \) are nonnegative

\[
\text{comp } O(n^3/p), \quad \text{comm } O(n^2/p^{2/3}), \quad \text{sync } O(\log p)
\]
Parallel all-pairs shortest paths by selective path doubling (contd.)

Every processor

- reads $A^{=q}$ and $A^{=(p)-q}$ and computes $A^{=q} \odot A^{=(p)-q} = A^{=(p)}$
- computes $(A^{=(p)})^* = (A^{=p})^*$ by sequential selective path doubling

All processors compute $(A^{=p})^* \odot A^{\leq p} = A^*$ by parallel matrix $\odot$-multiplication

\[
\begin{array}{c}
\text{comp } O(n^3/p) \\
\text{comm } O(n^2/p^{2/3}) \\
\text{sync } O(\log p)
\end{array}
\]