1 Computation by circuits

Model: abstraction of reality allowing qualitative and quantitative reasoning

Examples:
- atom
- biological cell
- galaxy
- Kepler’s universe
- Newton’s universe
- Einstein’s universe
- ...
Computation by circuits
Computation models and algorithms

Computation model: abstract computing device to reason about computations and algorithms
Examples:
- scales+weights (for “counterfeit coin” problems)
- Turing machine
- von Neumann machine (“ordinary computer”)
- JVM
- quantum computer
- ...

Algorithm: a finite description of a (usually infinite) set of computations on different inputs
Assumes a specific computation model and input/output encoding
Algorithm’s running time (worst-case) $T : \mathbb{N} \rightarrow \mathbb{N}$

$f(n), g(n) \geq 0 \quad n \rightarrow \infty$

$g = O(f)$: "$g$ grows at the same rate or slower than $f$"
$g = O(f) \iff \exists C : \forall n_0 : \forall n \geq n_0 : g(n) \leq C \cdot f(n)$

In words: we can scale $f$ up by a specific (possibly large) constant, so that $f$ will eventually overtake and stay above $g$

$g = o(f)$: "$g$ grows (strictly) slower than $f$"
$g = o(f) \iff \forall c : \exists n_0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)$

In words: even if we scale $f$ down by any (however small) constant, $f$ will still eventually overtake and stay above $g$

Overtaking point depends on the constant!
Exercise: $\exists n_0 : \forall c : \forall n \geq n_0 : g(n) \leq c \cdot f(n)$ — what does this say?
**Computation by circuits**

**Computation models and algorithms**

\[ g = \Omega(f): \text{"}g \text{ grows at the same rate or faster than } f\text{"} \]

\[ g = \omega(f): \text{"}g \text{ grows (strictly) faster than } f\text{"} \]

\[ g = \Omega(f) \iff f = O(g) \quad g = \omega(f) \iff f = o(g) \]

\[ g = \Theta(f): \text{"}g \text{ grows at the same rate as } f\text{"} \]

\[ g = \Theta(f) \iff g = O(f) \text{ and } g = \Omega(f) \]

Note: an algorithm is **faster**, when its complexity grows **slower**

Note: the “equality” in \( g = O(f) \) is actually set membership. Sometimes written \( g \in O(f) \), similarly for \( \Omega \), etc.

---

Example usage: sorting an array of size \( n \)

All good comparison-based sorting algorithms run in time \( O(n \log n) \)

If only pairwise comparisons between elements are allowed, no algorithm can run faster than \( \Omega(n \log n) \)

Hence, comparison-based sorting has complexity \( \Theta(n \log n) \)

If we are not restricted to just making comparisons, we can often sort in time \( o(n \log n) \), or even \( O(n) \)

---

**Example usage: multiplying \( n \times n \) matrices**

All good algorithms run in time \( O(n^3) \), where \( n \) is matrix size

If only addition and multiplication between elements are allowed, no algorithm can run faster than \( \Omega(n^3) \)

Hence, \((+, \times)\) matrix multiplication has complexity \( \Theta(n^3) \)

If subtraction is allowed, everything changes! The best known matrix multiplication algorithm (with subtraction) runs in time \( O(n^{2.373}) \)

It is conjectured that \( O(n^{2+\epsilon}) \) for any \( \epsilon > 0 \) is possible – open problem!

Matrix multiplication cannot run faster than \( \Omega(n^2 \log n) \) even with subtraction (under some natural assumptions)

---

\[ f(n), g(n) \geq 0 \quad n \to \infty \]

The maximum rule: \( f + g = \Theta(\max(f, g)) \)

Proof: for all \( n \), we have

\[ \max(f(n) + g(n)) \leq f(n) + g(n) \leq 2 \max(f(n) + g(n)) \]
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

Directed acyclic graph (dag), fixed number of inputs/outputs
Models oblivious computation: control sequence independent of the input
Computation on varying number of inputs: an (infinite) circuit family
May or may not admit a finite description (= algorithm)

In a circuit family, node indegree/outdegree may be bounded (by a constant), or unbounded: e.g. two-argument vs $n$-argument sum
Elementary operations:
- arithmetic/Boolean/comparison
- each (usually) constant time

size = number of nodes
depth = max path length from input to output
Other uses of circuits:
- arbitrary (non-oblivious) computation can be thought of as a circuit that is not given in advance, but revealed gradually
- timed circuits with feedback: systolic arrays

A comparison network is a circuit of comparator nodes

Input/output: sequences of equal length, taken from a totally ordered set
Examples:
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 finitely described family of sorting (or merging) networks 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**

**BUBBLE-SORT**

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

**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**!

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

Proof. “Only if”: trivial. “If”: by contradiction. Assume a given network does not sort input $x = (x_1, \ldots, x_n)$, where $(x_1, \ldots, x_n) \mapsto (y_1, \ldots, y_n)$, and there 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 $x_i \leq x_j \Rightarrow X_i \leq X_j$, therefore each $X_i$ follows the same path through the network as $x_i$.

$(X_1, \ldots, x_n) \mapsto (Y_1, \ldots, Y_n)$ where $Y_k = 1 > 0 = Y_l$.

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:

- a sorting network by checking only $2^n$ input sequences, instead of a much larger number $n! \approx (n/e)^n$
- 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 by recursion:

- 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: $(u_2, v_1), (u_3, v_2), \ldots$

$sizemerging(OEM(n', n'')) \leq 2 \cdot sizemerging(OEM(n'/2, n''/2)) + O(n) = O(n \log n)$

depthmerging(OEM(n', n'')) $\leq$ depthmerging(OEM(n'/2, n''/2)) $+ 1 = O(\log n)$

Correctness proof of odd-even merging: induction, zero-one principle

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

**Inductive step**: Inductive hypothesis: odd, even merge both work correctly

Let the input consist of 0s and 1s. We have for all $k$, $l$:

$\langle 0^{\lceil k/2 \rceil}11\ldots \rangle, \langle 0^{\lceil l/2 \rceil}11\ldots \rangle \mapsto \langle 0^{\lceil k/2 \rceil + \lceil l/2 \rceil}11\ldots \rangle$ in the odd merge

$\langle 0^{\lceil k/2 \rceil}11\ldots \rangle, \langle 0^{\lceil l/2 \rceil}11\ldots \rangle \mapsto \langle 0^{\lceil k/2 \rceil + \lceil l/2 \rceil}11\ldots \rangle$ in the even merge

$\langle [k/2] + [l/2] \rangle - ([k/2] + [l/2]) =$

\[
\begin{cases}
0,1 & \text{result sorted: } \langle 0^{k+l/2}11\ldots \rangle \\
2 & \text{single pair wrong: } \langle 0^{k+l/2-1}10111\ldots \rangle
\end{cases}
\]

The final stage of comparators corrects the wrong pair

$\langle 0^{k}11\ldots \rangle, \langle 0^{l}11\ldots \rangle \mapsto \langle 0^{k+l}11\ldots \rangle$
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 by recursion:
- sort \( \langle x_1, \ldots, x_{\lfloor n/2 \rfloor} \rangle \)
- sort \( \langle x_{\lceil n/2 \rceil + 1}, \ldots, x_n \rangle \)
- merge results by OEM(\( \lceil n/2 \rceil, \lfloor n/2 \rfloor \))

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

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

A bitonic sequence: \( \langle x_1 \geq \cdots \geq x_m \leq \cdots \leq x_n \rangle \) \( 1 \leq m \leq n \)

Bitonic merging: sorting a bitonic sequence
When \( n = 1 \) we are done, otherwise by recursion:
- sort bitonic \( \langle x_1, x_3, \ldots \rangle \mapsto \langle u_1 \leq u_2 \leq \cdots \leq u_{\lceil n/2 \rceil} \rangle \)
- sort bitonic \( \langle x_2, x_4, \ldots \rangle \mapsto \langle v_1 \leq v_2 \leq \cdots \leq v_{\lfloor n/2 \rfloor} \rangle \)
- compare pairwise: \( (u_1, v_1), (u_2, v_2), \ldots \)

Exercise: prove correctness (by zero-one principle)
Note: cannot exchange \( \geq \) and \( \leq \) in definition of bitonic!

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

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

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

\[
\text{size} 19 \quad \text{depth} 6
\]

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

\[
\text{size} 12 \quad \text{depth} 3
\]
Bitonic merge sorting [Batcher: 1968]

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

- sort \( \langle x_1, \ldots, x_{\lfloor n/2 \rfloor} \rangle \mapsto \langle y_1 \geq \cdots \geq y_{\lfloor n/2 \rfloor} \rangle \) in reverse
- sort \( \langle x_{\lfloor n/2 \rfloor + 1}, \ldots, x_n \rangle \mapsto \langle y_{\lfloor n/2 \rfloor + 1} \leq \cdots \leq y_n \rangle \)
- 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

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

Goals: scalability, portability, predictability

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)

Some elements of a BSP computer can be emulated by others, e.g.
- external memory by local memory + network communication
- barrier synchronisation mechanism by network communication

Communication network parameters:
- \( g \) is communication gap (inverse bandwidth), worst-case time for a data unit to enter/exit the network
- \( l \) is latency, worst-case time for a data unit to get across the network

Every parallel system can be (approximately) described by \( p, g, l \)

Network efficiency grows slower than processor efficiency and costs more energy: \( g, l \gg 1 \). 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 \( \text{proc} \) in superstep \( \text{sstep} \):

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

For the whole BSP computer in one superstep \( \text{sstep} \):

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

For the whole BSP computation with \( \text{sync} \) supersteps:

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

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:

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

Conventions:

- problem size $n \gg p$ (slackness)
- input/output in external memory, counts as one-sided communication

BSP algorithm design: minimising $\text{comp}$, $\text{comm}$, $\text{sync}$

Main principles:

- computation load balancing: ideally, $\text{comp} = O\left(\frac{\text{seq work}}{p}\right)$
- data locality: ideally $\text{comm} = O\left(\frac{\text{input/output}}{p}\right)$
- coarse granularity: ideally, $\text{sync}$ function of $p$ not $n$ (or better, $O(1)$)

Data locality exploited, network locality ignored!

---

Broadcasting:

- initially, one designated processor holds a value $a$
- at the end, every processor must hold a copy of $a$

Combining (complementary to broadcasting):

- initially, every processor $r$ holds a value $a^{(r)}$, $0 \leq r < p$
- at the end, one designated processor must hold $a^{(0)} \cdot \ldots \cdot a^{(p-1)}$ 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

\[
\text{comp} = O(p) \quad \text{comm} = O(p) \quad \text{sync} = O(1)
\]
From now on, cost components will be shaded when they are optimal. More precisely, $\text{cost} = O(f(n,p))$ means that for a given problem

- $\text{cost} = O(f(n,p))$ for the given algorithm on all inputs
- $\text{cost} = \Omega(f(n,p))$ for any algorithm on some inputs

This implies $\text{cost} = \Theta(f(n,p))$ for the given algorithm on the worst-case input, but also $\Omega(f(n,p))$ for any algorithm on its own worst-case input (which might be different).

**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 sends it to a sleeping processor, waking it up

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

$\text{comp} = O(\log p)$

$\text{comm} = O(\log p)$

$\text{sync} = O(\log p)$

**Array broadcasting:**

- Initially, one designated processor holds array $a = \langle a_0, \ldots, a_{n-1} \rangle$, $n \geq p$
- At the end, every processor must hold a copy of array $a$

**Combining** (complementary to broadcasting):

- Initially, every processor $r$ holds array $a^{(r)} = \langle a_0^{(r)}, \ldots, a_n^{(r)} \rangle$, $n \geq p$
- At the end, one designated processor must hold $a_0^{(0)} \bullet \cdots \bullet a_{n-1}^{(n-1)}$ for a given associative operator $\bullet$ (e.g. $+$)

Effectively, $n$ independent instances of broadcasting/combining

By symmetry, we only need to consider broadcasting

**Two-phase array broadcast:**

- Partition array into $p$ blocks of size $n/p$
- Scatter blocks, then total-exchange blocks

$\text{comp} = O(n)$

$\text{comm} = O(n)$

$\text{sync} = O(1)$

Enables concurrent access to external memory (in blocks of size $\geq p$)

Concurrent reading: one processor reads then broadcasts

Concurrent writing, resolved by arbitrary associative operator $\bullet$: one processor combines then writes
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

3D array network
$p = q^2$ processors
degree 6

diameter $3/2 \cdot p^{1/3} = 3/2 \cdot q$

2D array network
$p = q^2$ processors
degree 4

diameter $p^{1/2} = q$

Butterfly network
$p = q \log q$ processors
degree 4

diameter $\approx \log p \approx \log q$
Parallel computation models
Network routing

**Hypercube network**

\( p = 2^q \) processors

degree \( \log p = q \)

diameter \( \log p = q \)

<table>
<thead>
<tr>
<th>Network</th>
<th>Degree</th>
<th>Diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D array</td>
<td>2</td>
<td>( \frac{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>4</td>
<td>( \log p )</td>
</tr>
<tr>
<td>Hypercube</td>
<td>( \log p )</td>
<td>( \log p )</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

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: \([\text{Valiant: 1980}]\)
- 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

<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

The balanced binary tree dag

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

1 input, \(n\) outputs

size \(n - 1\)

depth \(\log n\)

A generalisation of broadcasting/combining

Can be defined top-down (input at root, outputs at leaves) or bottom-up

Sequential work \(O(n)\)

From now on, we always assume that a problem’s input/output is stored in the external memory; reading/writing will also refer to the external memory
Basic parallel algorithms
Balanced tree and prefix sums

Parallel balanced tree computation

$\text{tree}(n)$

Partition $\text{tree}(n)$ into
- one top block, isomorphic to $\text{tree}(p)$
- a bottom layer of $p$ blocks, each isomorphic to $\text{tree}(n/p)$

For top-down computation:
- a designated processor is assigned the top block; the processor reads block’s input, computes the block, writes block’s $p$ outputs
- every processor is assigned a different bottom block; each processor reads block’s input, computes the block, writes $n/p$ block’s outputs

For bottom-up computation, reverse the steps

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

For other problems, we may not be so lucky to get a fully-optimal BSP algorithm. 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 subject to optimality in $\text{comp}$

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 $\bullet$ be an operator, and assume

- operator $\bullet$ computable in size/depth $O(1)$
- operator $\bullet$ associative: $a \bullet (b \bullet c) = (a \bullet b) \bullet c$

Examples: numerical $+, \cdot$, min, max, Boolean $\land, \lor, \ldots$

Let $\epsilon$ be identity element for operator $\bullet$ (can be introduced formally if missing)

The prefix sums problem

$$a = [a_0, \ldots, a_{n-1}]$$
$$b_{-1} = \epsilon \quad b_i = a_i \bullet b_{i-1} \quad 0 \leq i < n$$

$$b_0 = a_0$$
$$b_1 = a_0 \bullet a_1$$
$$b_2 = a_0 \bullet a_1 \bullet a_2$$
$$\ldots$$

$$b_{n-1} = a_0 \bullet a_1 \bullet \cdots \bullet a_{n-1}$$

Sequential work $O(n)$ by trivial circuit of size $n - 1$, depth $n - 1$

The prefix circuit

$Ladner, Fischer: 1980$

$\text{prefix}(n)$

where $a_{k:l} = a_k \bullet a_{k+1} \bullet \cdots \bullet a_l$

The underlying dag is called the prefix dag

The prefix circuit (contd.)

$\text{prefix}(n)$

$n$ inputs
$n$ outputs
size $2n - 2$
depth $2 \log n$
Basic parallel algorithms
Balanced tree and prefix sums

Parallel prefix computation

Dag \textit{prefix}(n) consists of

- a top subtree similar to bottom-up \textit{tree}(n)
- transfer of values from top subtree to bottom subtree
- a bottom subtree similar to top-down \textit{tree}(n)

Both trees can be computed by the previous algorithm

Transfer stage: communication cost \( O(n/p) \)

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

Slackness \( n \geq p^2 \)

Application: \textit{generic first-order linear recurrence}

\[
\begin{align*}
\begin{bmatrix} 1 \\ c_i \end{bmatrix} &= \begin{bmatrix} 1 & 0 \\ a_i & b_i \end{bmatrix} \begin{bmatrix} 1 \\ c_{i-1} \end{bmatrix} \quad 0 \leq i < n \\
\end{align*}
\]

Let \( C_i = \begin{bmatrix} 1 \\ c_i \end{bmatrix}, \quad A_i = \begin{bmatrix} 1 & 0 \\ a_i & b_i \end{bmatrix} \)

\[
\begin{align*}
C_0 &= A_0 C_{-1} \\
C_1 &= A_1 A_0 C_{-1} \\
C_2 &= A_2 A_1 A_0 C_{-1} \\
& \quad \vdots \\
C_{n-1} &= A_{n-1} \ldots A_1 A_0 C_{-1}
\end{align*}
\]

Computing the generic first-order linear recurrence:

- suffix products of \([A_{n-1}, \ldots, A_0]\) with \(2 \times 2\) matrix multiplication
- each suffix product multiplied by \(C_{-1}\)

Resulting circuit: size \( O(n) \), depth \( O(\log n) \)

Similarly, we can compute generic first-order linear recurrence for any operators \(\oplus, \odot\), where

- operators \(\oplus, \odot\) computable in size/depth \(O(1)\)
- operator \(\ominus\) associative: \(a \ominus (b \oplus c) = (a \ominus b) \oplus c\)
- operator \(\odot\) associative: \(a \odot (b \ominus c) = (a \odot b) \ominus c\)
- operator \(\odot\) (left-)distributive over \(\oplus\): \(a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c)\)

Examples of suitable \(\oplus\) and \(\odot\):

- numerical \(+\) and \(\cdot\)
- numerical \(\min\) and \(\div\); numerical \(\max\) and \(+\)
- Boolean \(\land\) and \(\lor\); Boolean \(\lor\) and \(\land\)
Basic parallel algorithms
Balanced tree and prefix sums

Application: polynomial evaluation
\[ a = [a_0, \ldots, a_{n-1}] \]
\[ y = a_0 + a_1 \cdot x + \ldots + a_{n-2} \cdot x^{n-2} + a_{n-1} \cdot x^{n-1} \]
Evaluating the polynomial:
- \(1, x, x^2, \ldots, x^{n-1}\) by prefix product with operator \(\cdot\)
- sum \(y\) by bottom-up balanced binary tree with operator \(+\)

Resulting circuit: size \(O(n)\), depth \(O(\log n)\)

Basic parallel algorithms
Balanced tree and prefix sums

Application: polynomial evaluation by Horner’s rule
\[ a = [a_0, \ldots, a_{n-1}] \]
\[ y = a_0 + a_1 \cdot x + \ldots + a_{n-2} \cdot x^{n-2} + a_{n-1} \cdot x^{n-1} \]
Evaluating the polynomial:
- generic first-order linear recurrence over \([d_{n-1}, \ldots, d_0]\) and \([x, \ldots, x]\)

Resulting circuit: size \(O(n)\), depth \(O(\log n)\)

Basic parallel algorithms
Balanced tree and prefix sums

Application: binary addition via Boolean logic
\[ x + y = z \]
Define bit arrays \(u = [u_{n-1}, \ldots, u_0]\), \(v = [v_{n-1}, \ldots, v_0]\)
\[ u_i = x_i \land y_i \quad v_i = x_i \lor y_i \quad 0 \leq i < n \]
Arrays \(u, v\) can be computed in size \(O(n)\), depth \(O(1)\)
We then compute
\[ z_0 = v_0 \]
\[ c_0 = u_0 \]
\[ z_1 = v_1 \lor c_0 \]
\[ c_1 = u_1 \land (v_1 \lor c_0) \]
\[ \ldots \]
\[ z_{n-1} = v_{n-1} \lor c_{n-2} \]
\[ c_{n-1} = u_{n-1} \land (v_{n-1} \lor c_{n-2}) \]
\[ z_n = c_{n-1} \]
Resulting circuit has size and depth \(O(n)\)
Equivalent to a ripple-carry adder. Can we do better?
Basic parallel algorithms

Balanced tree and prefix sums

c_{-1} = 0 \quad c_{i} = u_{i} \lor (v_{i} \land c_{i-1})

Generic first-order linear recurrence with inputs \( u, v \) and operators \( \lor, \land \)

Compute \( c \) in size \( O(n) \), depth \( O(\log n) \)

Then compute \( z \) in extra size \( O(n) \), depth \( O(1) \)

Resulting circuit has size \( O(n) \), depth \( O(\log n) \)

Equivalent to a carry-lookahead adder

Basic parallel algorithms

Fast Fourier Transform and the butterfly dag

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

Assume \( n = 2^r \) \quad Let \( m = n^{1/2} = 2^s \)

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

Matrices \( A, B \) are vectors \( a, b \) written out as \( m \times m \) matrices

\[
B_{s,t} = \sum_{u,v} \omega^{(ms+t)(mu+1)} A_{u,v} = \sum_{u,v} \omega^{msv+tv+mut} A_{u,v} = \sum_{v} (\omega^{m})^{sv} \cdot \omega^{tv} \cdot \sum_{u} (\omega^{m})^{tu} A_{u,v}
\]

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 \( m \)

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

\[
F_{m,\omega^{m}}(A)_{t,v} = \sum_{u} (\omega^{m})^{tu} A_{u,v}
\]

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

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

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:

\[
F_{n,\omega}(a) = F_{n,\omega} \cdot a = b, \quad \text{where } F_{n,\omega} = [\omega^{ij}]_{i,j=0}^{n-1}
\]

\[
\begin{bmatrix}
1 & 1 & \cdots & 1 \\
1 & \omega & \omega^{2} & \cdots & \omega^{n-1} \\
1 & \omega^{2} & \omega^{4} & \cdots & \omega^{n-2} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{n-1} & \omega^{n-2} & \cdots & \omega
\end{bmatrix}
\]

\[
\sum_{j} \omega^{ij} a_{j} = b_{i} \quad i, j = 0, \ldots, n - 1
\]

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

Applications: digital signal processing (amplitude vs frequency);

polynomial multiplication; big integer multiplication

Basic parallel algorithms

Fast Fourier Transform and the butterfly dag

The Fast Fourier Transform (FFT) algorithm (contd.)

We have \( B = F_{m,\omega^{m}}(T_{m,\omega}(F_{m,\omega^{m}}(A))) \), thus DFT of size \( n \) in four steps:

- \( m \) independent DFTs of size \( m \)
- transposition and twiddle-factor scaling
- \( m \) independent DFTs of size \( m \)

We reduced DFT of size \( n = 2^{2r} \) to DFTs of size \( m = 2^s \). Similarly, can reduce DFT of size \( n = 2^{2r+1} \) to DFTs of sizes \( m = 2^s \) and \( 2m = 2^{s+1} \).

By recursion, we have the FFT circuit

\[
\text{size}_{\text{FFT}}(n) = O(n) + 2 \cdot n^{1/2} \cdot \text{size}_{\text{FFT}}(n^{1/2}) = O(1 \cdot n \cdot 1 + 2 \cdot n^{1/2} \cdot n^{1/2} + 4 \cdot n^{3/4} \cdot n^{1/4} + \cdots + \log n \cdot n \cdot 1) = O(n + 2n + 4n + \cdots + \log n \cdot n) = O(n \log n)
\]

\[
\text{depth}_{\text{FFT}}(n) = 1 + 2 \cdot \text{depth}_{\text{FFT}}(n^{1/2}) = O(1 + 2 + 4 + \cdots + \log n) = O(\log n)
\]
The underlying dag is called butterfly dag.

The FFT circuit and the butterfly dag (contd.)

Dag \( \text{bfly}(n) \) consists of:

- a top layer of \( n^{1/2} \) blocks, each isomorphic to \( \text{bfly}(n^{1/2}) \)
- a bottom layer of \( n^{1/2} \) blocks, each isomorphic to \( \text{bfly}(n^{1/2}) \)

The data exchange pattern between the top and bottom layers corresponds to \( n^{1/2} \times n^{1/2} \) matrix transposition.

Parallel butterfly computation

To compute \( \text{bfly}(n) \):

- every processor computes \( n^{1/2}/p \) blocks from the top layer
- every processor computes \( n^{1/2}/p \) blocks from the bottom layer

In each layer, the processor reads the total of \( n/p \) inputs, performs \( O(n \log n/p) \) computation, then writes the total of \( n/p \) outputs.
Basic parallel algorithms
Fast Fourier Transform and the butterfly dag

Parallel butterfly computation (contd.)

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

\[ \begin{array}{cccccccccccccc}
  a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 \\
  b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 \\
  a_8 & a_9 & a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\
  b_8 & b_9 & b_{10} & b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\
\end{array} \]

\[ \text{bfly}\left( \frac{n}{2} \right) \]

\[ \text{bfly}\left( \frac{n}{2} \right) \]

\[ \text{bfly}\left( \frac{n}{2} \right) \]

\[ \text{bfly}\left( \frac{n}{2} \right) \]

\[ \text{comp} = O\left( \frac{n \log n}{p} \right) \]

\[ \text{comm} = O\left( n/p \right) \]

\[ \text{sync} = O(1) \]

Slackness \( n \geq p^2 \)

Basic parallel algorithms
Ordered grid

Parallel ordered 2D grid computation

\[ \text{grid}_2(n) \]

Partition into a \( p \times p \) grid of blocks, each isomorphic to \( \text{grid}_2(n/p) \)

Arrange blocks as \( 2p - 1 \) anti-diagonal layers: \( \leq p \) independent blocks in each layer

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: triangular linear system; discretised PDE via Gauss–Seidel iteration (single step); 1D cellular automata; dynamic programming

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

Basic parallel algorithms
Ordered grid

Parallel ordered 2D grid computation (contd.)

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

- every block assigned to a different processor (some processors idle)
- the processor reads the \( 2n/p \) block inputs, computes the block, and writes back the \( 2n/p \) block outputs

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

\[ \text{comm} = (2p - 1) \cdot O(n/p) = O(n) \]

\[ \text{comp} = O(n^2/p) \]

\[ \text{comm} = O(n) \]

\[ \text{sync} = O(p) \]

Slackness \( n \geq p \)
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 \( b \)

\( a = \text{"DEFINE"} \quad b = \text{"DESIGN"} \quad LCS(a, b) = \text{"dein"} \)

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

LCS computation by dynamic programming [Wagner, Fischer: 1974]

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

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

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

\[
lcs(\text{"DEFINE"}, \text{"DESIGN"}) = 4
\]

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

Data dependence in the table corresponds to the 2D grid dag

Parallel LCS computation

The 2D grid algorithm solves the LCS problem (and many others) by dynamic programming

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

\( \text{comm} \) is not scalable (i.e. does not decrease with increasing \( p \))

Can scalable \( \text{comm} \) be achieved for the LCS problem?

Parallel LCS computation

Solve the more general semi-local LCS problem:

- each string vs all substrings of the other string
- all prefixes of each string against all suffixes of the other string

Divide-and-conquer on substrings of \( a, b \): \( \log p \) recursion levels

Each level assembles substring LCS from smaller ones by parallel seaweed multiplication

Base level: \( p \) semi-local LCS subproblems, each of size \( n/p^{1/2} \)

Sequential time still \( \mathcal{O}(n^2) \)
Parallel ordered 3D grid computation

**grid\(_3(n)\)**
- Partition into \(p^{1/2} \times p^{1/2} \times p^{1/2}\) grid of blocks, each isomorphic to \(grid\(_3(n/p^{1/2})\)\)
- Arrange blocks as \(3p^{1/2} - 2\) anti-diagonal layers: \(\leq p\) independent blocks in each layer

Open problem: can we achieve \(\text{comp} = O\left(\frac{n^2}{p}\right), \text{sync} = O(\log p)\)?

### Communication vs synchronisation tradeoff

**Parallel LCS computation (cont.)**

- Communication vs synchronisation tradeoff
- **Parallelising normal** \(O(n \log n)\) seaweed multiplication: [Krusche, T: 2010]
  - \(\text{comp} = O\left(\frac{n^2}{p}\right)\)
  - \(\text{comm} = O\left(\frac{n}{p^{1/2}}\right)\)
  - \(\text{sync} = O(\log^2 p)\)

- Special seaweed multiplication [Krusche, T: 2007]
  - Sacrifices some \(\text{comp}\), \(\text{comm}\) for \(\text{sync}\)
  - \(\text{comp} = O\left(\frac{n^2}{p}\right)\)
  - \(\text{comm} = O\left(\frac{n \log p}{p^{1/2}}\right)\)
  - \(\text{sync} = O(\log p)\)

- Open problem: can we achieve \(\text{comm} = O\left(\frac{n}{p^{1/2}}\right), \text{sync} = O(\log p)\)?
Basic parallel algorithms

Discussion

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

- Typically, realistic slackness requirements: $n \gg p$

The goals:
- $\text{comp} = \text{comp}_{\text{opt}} = \text{comp}_{\text{seq}}/p$
- $\text{comm}$ should scale down with increasing $p$
- $\text{sync}$ should be 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: array of $n$ nodes

- Each node contains data and a pointer to (= index of) successor node
- Nodes may be placed in array in an arbitrary order

Logical structure linear: $\text{head, succ(head), succ(succ(head)),…}$
- a pointer can be followed in time $O(1)$
- no global ranks/indexing/comparison

Pointer jumping at node $u$

- Let $\bullet$ be an associative operator, computable in time $O(1)$
- $v \leftarrow \text{succ}(u)$
- $\text{succ}(u) \leftarrow \text{succ}(v)$
- $a \leftarrow \text{data}(u)$
- $b \leftarrow \text{data}(v)$
- $\text{data}(u) \leftarrow a \bullet b$

Pointer $v$ and data $a$, $b$ are kept, so that pointer jumping can be reversed:
- $\text{succ}(u) \leftarrow v$
- $\text{data}(u) \leftarrow a$
- $\text{data}(v) \leftarrow b$
Further parallel algorithms
List contraction and colouring

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

![Diagram](image)

Data \(a, b\) are kept, 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, reversing contraction

Application: list ranking

![Diagram](image)

Node’s **rank**: distance from **head**

\(\text{rank}(\text{head}) = 0, \text{rank}('\text{succ}(\text{head})) = 1, \ldots\)

The **list ranking** problem: each node to hold its rank

![Diagram](image)

Note the solution should be independent of the merging order

Application: list ranking (contd.)

Each intermediate node during contraction/expansion represents a contiguous sublist in the original list

Contraction phase: each node \(u\) holds length \(l(u)\) of corresponding sublist

Initially, \(l(u) \leftarrow 1\) for each node \(u\)

Merging \(v, w\) into \(u\): \(l(u) \leftarrow l(v) + l(w)\), keeping \(l(v), l(w)\)

Fully contracted list: single node \(t\) holding \(l(t) = n\)

Expansion phase: each node holds

- length \(l(u)\) of corresponding sublist (as before)
- rank \(r(u)\) of the sublist’s starting node

Fully contracted list: single node \(t\) holding

\(l(t) = n\)  \(r(t) \leftarrow 0\)

Un-merging \(u\) to \(v, w\): restore \(l(u)\), \(l(v)\), then

\(r(v) \leftarrow r(u)\)  \(r(w) \leftarrow r(v) + l(v)\)

After full expansion: each node \(u\) holds

\(l(u) = 1\)  \(r(u) = \text{rank}(u)\)
Application: list prefix sums

Initially, each node \( u \) holds value \( a_{\text{rank}(u)} \)

Let \( \bullet \) be an associative operator with identity \( \epsilon \)

The list prefix sums problem: each node \( u \) to hold prefix sum

\[
a_{0:\text{rank}(u)} = a_0 \bullet a_1 \bullet \cdots \bullet a_{\text{rank}(u)}
\]

Note the solution should be independent of the merging order

Expansion phase: each node holds

\( \bullet \)-sum \( l(u) \) of corresponding sublist (as before)

\( \bullet \)-sum \( r(u) \) of all nodes before the sublist

Fully contracted list: single node \( t \) holding

\[
l(t) = a_{0:n-1} \quad r(t) \leftarrow \epsilon
\]

Un-merging \( u \) to \( v, w \): restore \( l(u), l(v), l(w) \), then

\[
r(v) \leftarrow r(u) \quad r(w) \leftarrow r(v) \bullet l(v)
\]

After full expansion: each node \( u \) holds

\[
l(u) = a_{\text{rank}(u)} \quad r(u) = a_{0:\text{rank}(u)}
\]

In general, only need to consider the contraction phase (expansion by symmetry)

Sequential contraction: always merge head with succ(head), time \( O(n) \)

Parallel contraction must be based on local merging decisions: a node can be merged with either its successor or predecessor, but not both

Therefore, we need either node splitting, or efficient symmetry breaking
Further parallel algorithms
List contraction and colouring

Wyllie’s mating [Wyllie: 1979]

Split every node, label copies “forward” and “backward”

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

Parallel list contraction by Wyllie’s mating

In the first round, every processor

- inputs $n/p$ nodes (not necessarily contiguous in input list), overall $n$ nodes forming input list across $p$ processors
- performs node splitting and labelling
- merges mating pairs; each merge involves communication between two processors; the merged node placed arbitrarily on either processor
- outputs the resulting $\leq 2n/p$ nodes (not necessarily contiguous in output list), overall $n$ nodes forming output lists across $p$ processors

Subsequent rounds similar

Parallel list contraction by Wyllie’s mating (contd.)

Parallel list contraction:

- perform $\log n$ rounds of Wyllie’s mating, reducing original list to $n$ fully contracted lists of size 1
- 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” or “backward”

For each node, labelling independent with probability $\frac{1}{2}$

A node mates with probability $\frac{1}{2}$, hence on average $\frac{n}{2}$ nodes mate

Merge mating node pairs, obtaining a new list of expected size $\frac{3n}{4}$

Moreover, the new list has size $\leq \frac{15n}{16}$ with high probability (whp), i.e. with probability exponentially close to 1 (as a function of $n$)

$$\text{Prob}(\text{new size} \leq \frac{15n}{16}) \geq 1 - e^{-n/64}$$
Parallel list contraction by random mating

In the first round, every processor
- inputs $\frac{n}{p}$ nodes (not necessarily contiguous in input list), overall $n$ nodes forming input list across $p$ processors
- performs node randomisation and labelling
- merges mating pairs; each merge involves communication between two processors; the merged node placed arbitrarily on either processor
- outputs the resulting $\leq \frac{n}{p}$ nodes (not necessarily contiguous in output list), overall $\leq \frac{15n}{16}$ nodes (whp), forming output list across $p$ processors

Subsequent rounds similar, on a list of decreasing size (whp)

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
Parallel list contraction by block mating

In the first round, every processor
- inputs $\frac{n}{p}$ nodes (not necessarily contiguous in input list), overall $n$ nodes forming input list across $p$ processors
- participates in construction of distribution graph and communicating it to the designated processor

The designated processor collects distribution graph, computes and distributes labels

Parallel list contraction by block mating (contd.)

Continuing the first round, every processor
- receives its label from the designated processor
- merges mating pairs; each merge involves communication between two processors; the merged node placed arbitrarily on either processor
- outputs the resulting $\leq \frac{n}{p}$ nodes (not necessarily contiguous in output list), overall $\leq \frac{3n}{4}$ nodes, forming output list across $p$ processors

Subsequent rounds similar, on a list of decreasing size

The list $k$-colouring problem: given a linked list and an integer $k > 1$, assign a colour from $\{0, \ldots, k-1\}$ to every node, so that in each pair of adjacent nodes, the two colours are different

Using list contraction, $k$-colouring for any $k$ can be done in

$$\text{comp} = O(n/p) \quad \text{comm} = O(n/p) \quad \text{sync} = O(\log p)$$

Is list contraction really necessary for list $k$-colouring?

Can list $k$-colouring be done more efficiently?

For $k = p$: we can easily (how?) do $p$-colouring in

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

Can this be extended to any $k \leq p$, e.g. $k = O(1)$?
Further parallel algorithms
List contraction and colouring

Deterministic coin tossing

Given a $k$-colouring, $k > 6$
Consider every node $v$. We have $\text{col}(v) \neq \text{col}(\text{succ}(v))$.
If $\text{col}(v)$ differs from $\text{col}(\text{succ}(v))$ in $i$-th bit, re-colour $v$ in
- $2i$, if $i$-th bit in $\text{col}(v)$ is 0, and in $\text{col}(\text{succ}(v))$ is 1
- $2i + 1$, if $i$-th bit in $\text{col}(v)$ is 1, and in $\text{col}(\text{succ}(v))$ is 0
Model assumption: can find lowest nonzero bit in an integer in time $O(1)$
After re-colouring, still have $\text{col}(v) \neq \text{col}(\text{succ}(v))$
Number of colours reduced from $k$ to $2 \left\lceil \log k \right\rceil \ll k$

comp, comm: $O(n/p)$

Further parallel algorithms
List contraction and colouring

Parallel list contraction and colouring

Parallel list colouring by deterministic coin tossing

Reducing the number of colours from $p$ to 6: need $O(\log^* p)$ rounds of deterministic coin tossing
The iterated log function

$\log^* k = \min r : \log \ldots \log k \leq 1$ (r times)

Number of particles in observable universe: $10^{81} \approx 2^{270}$

$\log^* 2^{270} = 1 + \log^* 270 \leq 1 + \log^* 512 = 1 + \log^* 2^9 = 2 + \log^* 9 \leq 2 + \log^* 16 = 2 + \log^* 2^4 = 3 + \log^* 4 = 3 + \log^* 2^2 = 4 + \log^* 2 = 5 + \log^* 1 = 5$

$\log^* 2^{65536} = \log^* 2^{2^{22}} = 5$

Further parallel algorithms
List contraction and colouring

Parallel list colouring by deterministic coin tossing (contd.)

Initially, each processor reads a subset of $n/p$ nodes
- partially contract the list to size $O(n/\log^* p)$ by $\log_{4/3} \log^* p$ rounds of block mating
- compute a $p$-colouring of the resulting list
- reduce the number of colours from $p$ to 6 by $O(\log^* p)$ rounds of deterministic coin tossing

comp, comm: $O(\frac{n}{p} + \frac{n}{p \log^* p} \cdot \log^* p) = O(n/p)$

sync: $O(\log^* p)$

Further parallel algorithms
List contraction and colouring

Parallel list colouring by deterministic coin tossing (contd.)

We have a 6-coloured, partially contacted list of size $O(n/\log^* p)$
- select node $v$ as a pivot, if $\text{col}(\text{pred}(v)) > \text{col}(v) < \text{col}(\text{succ}(v))$; no two pivots are adjacent or further than 12 nodes apart
- re-colour all pivots in one colour
- from each pivot, 2-colour the next $\leq 12$ non-pivots sequentially; we now have a 3-coloured list
- reverse the partial contraction, maintaining the 3-colouring

We have now 3-coloured the original list

$\text{comp} = O(n/p)$  \hspace{1cm} $\text{comm} = O(n/p)$  \hspace{1cm} $\text{sync} = O(\log^* p)$  \hspace{1cm} $n \geq p^4$
Further parallel algorithms

Sorting

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

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 [Shi, Schaeffer: 1992]

Every processor

- reads a subarray of $a$ of size $n/p$ and sorts it sequentially
- selects from it $p$ samples from index 0 at regular intervals $n/p^2$
  defining $p$ equal-sized, contiguous blocks in subarray

A designated processor

- collects all $p^2$ samples and sorts them sequentially
- selects from them $p$ splitters from index 0 at regular intervals $p$
  defining $p$ unequal-sized, non-contiguous buckets in array $a$
- broadcasts the splitters

We will need to prove that bucket sizes, although not uniform, are still well-balanced ($\leq 2n/p$)

$$\begin{align*}
\text{comp} = O\left(\frac{n \log n}{p}\right) & \quad \text{comm} = O\left(\frac{n}{p}\right) & \quad \text{sync} = O(1)
\end{align*}$$

$n \geq p^3$
Further parallel algorithms

Sorting

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

Proof (sketch). Relative to a fixed bucket \( B \), a block \( b \) is
- low, if lower boundary of \( b \) is \( \leq \) lower boundary of \( B \)
- high otherwise

A bucket may only intersect
- \( \leq 1 \) low block per processor, hence \( \leq p \) low blocks overall
- \( \leq p \) high blocks overall

Therefore, bucket size \( \leq (p + p) \cdot \frac{n}{p^2} = \frac{2n}{p} \)

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 with sorting, we assume the comparison model

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

Sequential work \( O(n) \) by median sampling \[Blum+: 1973\]

Selection by median sampling \[Blum+: 1973\]

Proceed in rounds. In each round:
- partition array \( a \) into subarrays of size 5
- in each subarray, select median e.g. by 5-element sorting
- select median-of-medians by recursion: \((n, k) \leftarrow (n/5, n/10)\)
- find rank \( l \) of median-of-medians in array \( a \) by linear search

If \( l = k \), return \( a_l \); otherwise, eliminate elements on "wrong side" of \( l \) and set new target rank for next round:
- if \( l < k \), eliminate all \( a_l \leq a_i \); for next round \( n \leftarrow n - l - 1 \), \( k \leftarrow k - l - 1 \)
- if \( l > k \), eliminate all \( a_i \geq a_l \); for next round \( n \leftarrow l, k \) unchanged
Further parallel algorithms
Selection

Claim: Each round removes \( \geq \frac{3n}{10} \) of elements of a

Proof (sketch). We have \( \frac{n}{5} \) subarrays

In at least \( \frac{1}{2} \cdot \frac{n}{5} \) subarrays, subarray median \( \leq a_I \)

In every such subarray, three elements \( \leq \) subarray median \( \leq a_I \)

Hence, at least \( \frac{1}{2} \cdot \frac{3n}{10} = \frac{3n}{10} \) elements \( \leq a_I \)

Symmetrically, at least \( \frac{3n}{10} \) elements \( \geq a_I \)

Therefore, in a round, at least \( \frac{3n}{10} \) elements are eliminated

Data reduction rate is exponential

\[ T(n) \leq T\left(\frac{n}{2}\right) + T(n - \frac{3n}{10}) + O(n) = T\left(\frac{2n}{10}\right) + T\left(\frac{7n}{10}\right) + O(n), \]

therefore

\[ T(n) = O(n) \]

Parallel selection by median sampling (contd.)

A designated processor

- adds up local ranks to determine global rank of median-of-medians
- compares it against target rank to determine direction of elimination
- broadcasts info on this direction

Every processor

- performs elimination on its subarray
- writes remaining elements

\( \leq \frac{3n}{4} \) elements remain overall in array \( a \)

Subsequent rounds similar, on an array of decreasing size, with target rank adjusted as necessary
Parallel selection by regular sampling (generalised median sampling)

In the first round, every processor

- reads a subarray of \( a \) of size \( n/p \)
- selects from it \( s = O(1) \) samples from rank 0 at regular rank intervals \( n/sp \), defining \( s \) equal-sized, non-contiguous blocks in subarray

A designated processor

- collects all \( sp \) samples
- selects from them \( s \) splitters from rank 0 at regular rank intervals \( p \), defining \( s \) unequal-sized, non-contiguous buckets in array \( a \)
- broadcasts the splitters

Every processor

- determines rank of every splitter in local subarray

Parallel selection by accelerated regular sampling

In median sampling, we maintain \( s = 2 \) (sample 0 and median); array shrinks exponentially

Varying \( s \) helps reduce the number of rounds: as array shrinks, we can afford to increase sampling frequency; array will shrink superexponentially

Parallel selection:

- perform \( O(\log \log p) \) rounds of regular sampling (with increasing frequency) and elimination, reducing original array to size \( n/p \)
- a designated processor collects the remaining array and performs selection sequentially

Technical details omitted

\[
\text{comp} = O(n/p) \quad \text{comm} = O(n/p) \quad \text{sync} = O(\log \log p) \quad n \gg p
\]
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 $\text{conv} A$.

Definition of convexity requires arithmetic on coordinates, hence we assume the arithmetic model.

Further parallel algorithms
Convex hull

$d = 2$

Fundamental arithmetic primitive: signed area of a triangle

Let $a_0 = (x_0, y_0), a_1 = (x_1, y_1), a_2 = (x_2, y_2)$

$$\Delta(a_0, a_1, a_2) = \frac{1}{2} \begin{vmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix} = \frac{1}{2} ((x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0))$$

$$\Delta(a_0, a_1, a_2) \begin{cases} < 0 & \text{if } a_0, a_1, a_2 \text{ clockwise} \\ = 0 & \text{if } a_0, a_1, a_2 \text{ collinear} \\ > 0 & \text{if } a_0, a_1, a_2 \text{ counterclockwise} \end{cases}$$

An easy $O(1)$ check: $a_0$ is to the left/right of directed line from $a_1$ to $a_2$?

All of $A$ is to the left of every edge of $\text{conv} A$, traversed counterclockwise.

Further parallel algorithms
Convex hull

The (discrete) convex hull problem

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

Output (a finite representation of) $\text{conv} a$.

More precisely, must output each $k$-dimensional face of $\text{conv} a$, $1 \leq k < d$.

E.g. in 3D: vertices, edges and facets.

Output must be structured:

- in 2D, all vertex-edge incidence pairs; every vertex should “know” its two neighbours
- for general $d$, all incidence pairs between $k$-D and $k+1$-D faces

Further parallel algorithms
Convex hull

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 edges from vertex to vertex to obtain sorted output
Further parallel algorithms

Convex hull

The discrete convex hull problem

\( d = 2: \) \( \leq n \) vertices, \( \leq n \) edges, output size \( \leq 2n \)

\( d = 3: \) \( O(n) \) vertices, edges and facets, output size \( O(n) \)

\( d > 3: \) much bigger output...

Claim: for general \( d \), \( \text{conv } A \) contains \( O(n^{\lfloor d/2 \rfloor}) \) faces of various dimensions

Hence

- for \( d = 4, 5 \) output size \( O(n^2) \)
- for \( d = 6, 7 \) output size \( O(n^3) \)
- ...

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

Sequential work \( O(n \log n) \) by Graham’s scan (2D) or mergehull (2D, 3D)

Claim. An \( \epsilon \)-approximation for \( A \) is an \( \epsilon \)-net for \( A \)
The converse does not hold!

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, independently of \( d \).
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. Sort points of \( A \) in circular order they appear on the convex hull.
Put every \( n/r \)-th point into \( E \). We have \( |E| \leq r \).

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 2\( p \) splitters can be assumed to be in convex position (like any \( \epsilon \)-net), and therefore define a splitter polygon with at most 2\( p \) edges.
Each vertex of splitter polygon defines a bucket: the subset of set \( a \) visible when sitting at this vertex (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 \( O(n \log n) \).

Proof: [Brönnimann, Goodrich: 1995]

Claim. A 1/r-approximation for A of size \( O(r^3(\log r)^{O(1)}) \) exists and can be computed in sequential work \( O(n \log r) \).

Proof: [Matoušek: 1992]

Better approximations are possible, but are slower to compute

Further parallel algorithms

Parallel 3D hull computation by\textit{ 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\textit{ 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-spaces not containg 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 a bucket
- scans its hull and sends each point to the appropriate bucket
- receives the points of its bucket and computes their convex hull sequentially
- writes the bucket hull back to external memory

\[
\text{comp} = O\left(\frac{n \log n}{p}\right) \quad \text{comm} = O(n/p) \quad \text{sync} = O(1)
\]
Parallel matrix algorithms
Matrix-vector multiplication

\[ A: n\text{-matrix} \quad b, c: n\text{-vectors} \]

The \textbf{matrix-vector multiplication} problem

\[ A \cdot b = c \]

\[ c_i = \sum_j A_{ij} \cdot b_j \ (0 \leq i, j < n) \]

Consider elements of \( b \) as inputs and of \( c \) as outputs

Elements of \( A \) are considered to be problem parameters, do not count as inputs (motivation: iterative linear algebra methods)

Overall, \( n^2 \) \textbf{elementary products} \( A_{ij} \cdot b_j = c_i^l \)

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

The \textbf{matrix-vector multiplication circuit}

\[ c \leftarrow 0 \]

For all \( i, j: \) \( c_i \leftarrow c_i^j \leftarrow A_{ij} \cdot b_j \)

(adding each \( c_i^j \) to \( c_i \) asynchronously)

\( n \) input nodes of outdegree \( n \), one per element of \( b \)

\( n^2 \) \textbf{computation nodes} of in- and outdegree 1, one per elementary product

\( n \) output nodes of indegree \( n \), one per element of \( c \)

\( A \) size \( O(n^2) \), depth \( O(1) \)

Parallel matrix-vector multiplication

Partition computation nodes into a regular grid of \( p = \frac{p^1}{2} \cdot \frac{p^1}{2} \) square \( \frac{n}{p^{1/2}} \)-blocks

Matrix \( A \) gets partitioned into \( p \) square \( \frac{n}{p^{1/2}} \)-blocks \( A_{IJ} \) \( (0 \leq I, J < p^{1/2}) \)

Vectors \( b, c \) each gets partitioned into \( p^{1/2} \) linear \( \frac{n}{p^{1/2}} \)-blocks \( b_J, c_I \)

Overall, \( p \) \textbf{block products} \( A_{Ij} \cdot b_j = c_i^l \)

\[ c_I = \sum_{0 \leq J < p^{1/2}} c_i^J \ \text{for all} \ I \]
Parallel matrix algorithms
Matrix-vector multiplication

Parallel matrix-vector multiplication (contd.)

\[ c \leftarrow 0 \]

For all \( i, J \):

\[ c \downarrow \leftarrow c_{ij} \leftarrow A_{ij} \cdot b_j \]

\[
\begin{array}{c}
A
\end{array}
\begin{array}{c}
\downarrow
\end{array}
\begin{array}{c}
\uparrow
\end{array}
\begin{array}{c}
b
\end{array}
\begin{array}{c}
\rightarrow
\end{array}
\begin{array}{c}
c
\end{array}
\]

\[ A \cdot b \rightarrow c \]

\[ j \]

\[ i \]

\[ A, B, C: n\text{-matrices} \]

The matrix multiplication problem

\[ A \cdot B = C \]

\[ C_{ik} = \sum_j A_{ij} \cdot B_{jk} \quad (0 \leq i,j,k < n) \]

Overall, \( n^3 \) elementary products \( A_{ij} \cdot B_{jk} = C^i_{jk} \)

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

Parallel matrix algorithms
Matrix multiplication

The matrix multiplication circuit

\[ C_{ik} \leftarrow 0 \]

For all \( i, j, k \):

\[ C^i_{ik} \leftarrow C^i_{ik} \leftarrow A_{ij} \cdot B_{jk} \]

(adding each \( C^i_{ik} \) to \( C_{ik} \) asynchronously)

2n input nodes of outdegree \( n \), one per element of \( A, B \)

\( n^2 \) computation nodes of in- and outdegree 1, one per elementary product

\( n \) output nodes of indegree \( n \), one per element of \( C \)

size \( O(n^3) \), depth \( O(1) \)

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

Slackness: \( \frac{n}{p^{1/2}} \geq p^{1/2} \) required by array concurrent write, hence \( n \geq p \)
Parallel matrix algorithms
Matrix multiplication

Parallel matrix multiplication
Partition computation nodes into a regular grid of \( p = p^{1/3} \cdot p^{1/3} \cdot p^{1/3} \)
\( n \frac{p}{p^{2/3}} \)-blocks
Matrices \( A, B, C \) each gets partitioned into \( p^{2/3} \) square \( n \frac{p}{p^{2/3}} \)-blocks \( A_{IJ}, B_{JK}, C_{IK} \) \( (0 \leq I, J, K < p^{1/3}) \)
Overall, \( p \) block products \( A_{IJ} \cdot B_{JK} = C_{IK} \)
\( C_{IK} = \sum_{0 \leq J < p^{1/2}} C^J_{IK} \) for all \( I, K \)

Initialise \( C \leftarrow 0 \) in external memory
Every processor
- is assigned \( I, J, K \)
- reads blocks \( A_{IJ}, B_{JK} \), and computes \( C^J_{IK} \leftarrow A_{IJ} \cdot B_{JK} \)
- updates \( C_{IK} \leftarrow C^J_{IK} \) in external memory; concurrent writing resolved by operator ‘+’ (recall array broadcast/combine)

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

Slackness: \( \frac{n^2}{p^{2/3}} \geq p^{1/3} \) required by array concurrent write, hence \( n \geq p^{1/2} \)

Parallel matrix algorithms
Matrix multiplication

Parallel matrix multiplication (contd.)

Theorem. Computing the matrix multiplication dag requires communication \( \Omega\left(\frac{n^3}{p^{2/3}}\right) \) per processor

Proof: (discrete) volume vs surface area
Let \( V \) be the subset of nodes 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 \( \text{comm} \geq |A| + |B| + |C| \geq 3(|A| \cdot |B| \cdot |C|)^{1/3} \geq 3|V|^{2/3} \geq
3\left(\frac{n^3}{p}\right)^{2/3} = \frac{3n^2}{p^{2/3}} \), hence \( \text{comm} = \Omega\left(\frac{n^3}{p^{2/3}}\right) \)

Note that this is not conditioned on \( \text{comp} = O\left(\frac{n^3}{p}\right) \)
Parallel matrix algorithms
Matrix multiplication

The optimality theorem only applies to matrix multiplication by the specific $O(n^3)$-node dag
Includes e.g.
- numerical matrix multiplication with only `+`, `·` allowed
- Boolean matrix multiplication with only `∨`, `∧` allowed
Excludes e.g.
- numerical matrix multiplication when `−` also allowed
- Boolean matrix multiplication when `if/then` also allowed

Alexander Tiskin (Warwick)
Advanced Topics in Algorithms
169 / 235

Parallel matrix algorithms
Fast matrix multiplication

2-matrix multiplication: standard circuit

$A \cdot B = C$

$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}$

$B = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$

$C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$

Let $A$, $B$, $C$ be over a ring: `+`, `−`, `·` allowed on elements

$D^{(0)} = (A_{00} + A_{11}) \cdot (B_{00} + B_{11})$

$D^{(1)} = (A_{10} + A_{11}) \cdot B_{00}$

$D^{(3)} = A_{11} \cdot (B_{10} - B_{00})$

$D^{(5)} = (A_{10} - A_{00}) \cdot (B_{00} + B_{01})$

$C_{00} = D^{(0)} + D^{(3)} - D^{(4)} + D^{(6)}$

$C_{10} = D^{(1)} + D^{(3)}$

$C_{01} = D^{(2)} + D^{(4)}$

$C_{11} = D^{(0)} - D^{(1)} + D^{(2)} + D^{(5)}$

$A_{00}, \ldots$: either ordinary elements or square blocks; 8 multiplications

Alexander Tiskin (Warwick)
Advanced Topics in Algorithms
170 / 235

Parallel matrix algorithms
Fast matrix multiplication

$N$-matrix multiplication: bilinear circuit

- certain $R$ linear combinations of elements of $A$
- certain $R$ linear combinations of elements of $B$
- $R$ pairwise products of these combinations
- certain $N^2$ linear combinations of these products, each giving an element of $C$

Bilinear circuits for matrix multiplication:
- standard: $N = 2$, $R = 8$, combinations trivial
- Strassen: $N = 2$, $R = 7$, combinations highly surprising!
- sub-Strassen: $N > 2$, $N^2 < R < N^{\log_2 7} \approx N^{2.81}$

Elements of $A$, $B$, $C$: either ordinary elements or square blocks
Block-recursive matrix multiplication

Given a scheme: bilinear circuit with fixed $N$, $R$

Let $A$, $B$, $C$ be $n$-matrices, $n \geq N$ \quad $A \cdot B = C$

Partition each of $A$, $B$, $C$ into an $N \times N$ regular grid of $n/N$-blocks

Apply the scheme, treating

- each ‘+’ as block ‘+’, each ‘−’ as block ‘−’
- each ‘·’ as recursive call on blocks

Resulting recursive bilinear circuit:

- size $O(n^\omega)$, where $\omega = \log_N R < \log_N N^3 = 3$
- depth $\approx 2 \log n$

Sequential work $O(n^\omega)$

---

Parallel block-recursive matrix multiplication

At each level of the recursion tree, the $R$ recursive calls are independent, hence the recursion tree can be computed breadth-first

At recursion level $k$:

- $R^k$ independent block multiplication subproblems

In particular, at level $\log_R p$:

- $p$ independent block multiplication subproblems, therefore each subproblem can be solved sequentially on an arbitrary processor
Parallel matrix algorithms

Fast matrix multiplication

Parallel block-recursive matrix multiplication (contd.)

In recursion levels 0 to \( \log_R p \), need to compute elementwise linear combinations on distributed matrices

Assigning matrix elements to processors:

- partition \( A \) into regular \( \frac{n}{p^{1/\omega}} \)-blocks
- distribute each block evenly and identically across processors
- partition \( B, C \) analogously (distribution identical across all blocks of the same matrix, need not be identical across different matrices)

E.g. cyclic distribution

Linear combinations of matrix blocks in recursion levels 0 to \( \log_R p \) can now be computed without communication

Alexander Tiskin (Warwick)
Advanced Topics in Algorithms

Parallel block-recursive matrix multiplication (contd.)

Each processor inputs its assigned elements of \( A, B \)

Downsweep of recursion tree, levels 0 to \( \log_R p \):

- linear combinations of blocks of \( A, B \), no communication

Recursion levels below \( \log_R p \): \( p \) block multiplication subproblems

- assign each subproblem to a different processor
- a processor collects its subproblem’s two input blocks, solves it sequentially, then redistributes the subproblem’s output block

Upsweep of recursion tree, levels \( \log_R p \) to 0:

- linear combinations giving blocks of \( C \), no communication

Each processor outputs its assigned elements of \( C \)

\[
\begin{align*}
comp &= O\left(\frac{n^3}{p}\right) \\
comm &= O\left(\frac{n^2}{p^{1/\omega}}\right) \\
sync &= O(1)
\end{align*}
\]

Alexander Tiskin (Warwick)
Advanced Topics in Algorithms

Parallel matrix algorithms

Boolean matrix multiplication

Boolean matrix multiplication

Let \( A, B, C \) be Boolean \( n \)-matrices: ‘\( \lor \)’, ‘\( \land \)’, ‘if/then’ allowed on elements

\[
A \land B = C
\]

\[
C_{ik} = \lor_j A_{ij} \land B_{jk} \quad 0 \leq i, j, k < n
\]

Overall, \( n^3 \) elementary products \( A_{ij} \land B_{jk} \)

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

BSP costs in bit operations:

\[
\begin{align*}
comp &= O\left(\frac{n^3}{p}\right) \\
comm &= O\left(\frac{n^2}{p^{1/3}}\right) \\
sync &= O(1)
\end{align*}
\]

Alexander Tiskin (Warwick)
Advanced Topics in Algorithms
Fast Boolean matrix multiplication

\[ A \land B = C \]

\[ A'_{ij} \leftarrow A_{ij} \text{ where } 0, 1 \text{ are treated as integers} \]

\[ B'_{jk} \leftarrow B_{jk} \text{ where } 0, 1 \text{ are treated as integers} \]

Compute \( A' \cdot B' = C' \mod n + 1 \) using a Strassen-like algorithm

\[ C_{ik} \leftarrow \text{"} C'_{jk} \neq 0 \mod n + 1 \text{"} \]

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

BSP costs:

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

Parallel Boolean matrix multiplication by regular decomposition

The following algorithm is impractical, but of theoretical interest, because 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) [Szemerédi: 1978]

\[ K = K(\epsilon), \text{ where } \epsilon \text{ is the "degree of random-likeness"} \]

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

We shall call this the regular decomposition of a Boolean matrix

Parallel Boolean matrix multiplication by regular decomposition (contd.)

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

- \( A^{(1)} \land B^{(1)} = C^{(1)} \), where \( A^{(1)} \) is sparse
- \( A^{(2)} \land B^{(2)} = C^{(2)} \), where \( B^{(2)} \) is sparse
- \( A^{(3)} \land B^{(3)} = C^{(3)} \), where \( C^{(3)} \) is sparse
- \( C = C^{(1)} \lor C^{(2)} \lor C^{(3)} \)

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

Parallel Boolean matrix multiplication by regular decomposition (contd.)

\( A \land B = \overline{C} \)

Partition \( ijk \)-cube into a regular grid of \( p^3 = p \cdot p \cdot p \) cubic \( \frac{n}{p} \)-blocks

\( A, B, C \) each gets partitioned into \( p^2 \) square \( \frac{n}{p} \)-blocks \( A_{IJ}, B_{JK}, C_{IK} \)

\( 0 \leq I, J, K < p \)

---

Parallel matrix algorithms
Triangular system solution

Let \( L \) be an \( n \)-matrix, \( b, c \) be \( n \)-vectors

\( L \) is lower triangular: 

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

\( L \cdot b = c \)

The triangular system problem: given 

\( L, c, \) find \( b \)

---
Parallel matrix algorithms
Triangular system solution

Forward substitution

\[ L \cdot b = c \]

\[
\begin{align*}
L_{00} \cdot b_0 & = c_0 \\
L_{10} \cdot b_0 + L_{11} \cdot b_1 & = c_1 \\
L_{20} \cdot b_0 + L_{21} \cdot b_1 + L_{22} \cdot b_2 & = c_2 \\
& \quad \vdots \\
L_{n-1,0} \cdot b_0 + & \quad \vdots \\
L_{n-1,n-1} \cdot b_1 + & \quad \vdots \\
& \quad \vdots \\
L_{n-1,n-1} \cdot b_{n-1} & = c_{n-1}
\end{align*}
\]

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

Symmetrically, an upper triangular system solved by back substitution

Parallel matrix algorithms
Triangular system solution

Block-recursive forward substitution

\[ L \cdot b = c \]

\[
\begin{bmatrix}
L_{00} & L_{01} \\
L_{10} & L_{11}
\end{bmatrix}
\begin{bmatrix}
b_0 \\
b_1
\end{bmatrix}
= 
\begin{bmatrix}
c_0 \\
c_1
\end{bmatrix}
\]

Recursion: two half-sized subproblems

\[
\begin{align*}
L_{00} \cdot b_0 & = c_0 \\
L_{10} \cdot b_0 + L_{11} \cdot b_1 & = c_1 \\
& \quad \vdots \\
L_{n-1,0} \cdot b_0 + & \quad \vdots \\
& \quad \vdots \\
& \quad \vdots \\
L_{n-1,n-1} \cdot b_{n-1} & = c_{n-1}
\end{align*}
\]

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

Parallel forward substitution by 2D grid

Assume \(L\) is predistributed as needed, does not count as input

\[
\begin{bmatrix}
\end{bmatrix}
\]

Pivot node:

\[
\begin{align*}
s & \rightarrow L[i,j]^{-1} \cdot (c - s) \\
L[i,j]^{-1} \cdot (c - s)
\end{align*}
\]

Update node:

\[
\begin{align*}
s & \rightarrow s + L[i,j] \cdot b \\
b & \rightarrow b
\end{align*}
\]

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

Parallel block-recursive forward substitution

Assume \(L\) is predistributed as needed, does not count as input

At each level, the two recursive subproblems are dependent, hence recursion tree must be computed depth-first

At recursion level \(k\):

- sequence of \(2^k\) triangular system subproblems, each on \(n/2^k\)-blocks

In particular, at level \(\log p\):

- sequence of \(p\) triangular system subproblems, each on \(n/p\)-blocks
- total \(p \cdot O((n/p)^2) = O(n^2/p)\) sequential work, therefore each subproblem can be solved sequentially on an arbitrary processor
Parallel matrix algorithms
Triangular system solution

Parallel block-recursive 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

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

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

\[
\begin{align*}
\text{comp} &= O(n^2/p) \quad \text{comm} = O(n) \quad \text{sync} = O(p)
\end{align*}
\]

Parallel matrix algorithms
Generic Gaussian elimination

Application: solving a linear system
\( Ax = b \)

If LU decomposition of \( A \) is known: \( Ax = LUx = b \)
Solve triangular systems \( Ly = b \) then \( Ux = y \), obtaining \( x \)
LU decomposition of \( A \) can be reused for multiple right-hand sides \( b \)

Let \( A, L, U \) be \( n \)-matrices

**LU decomposition** of \( A: A = L \cdot U \)

\[
A = \begin{bmatrix} L & \cdot \\ \cdot & U \end{bmatrix}
\]

\( L \) is unit lower triangular: \( L_{ij} = \begin{cases} \text{arbitrary} & \text{below diagonal} (i > j) \\ 1 & \text{on diagonal} (i = j) \\ 0 & \text{above diagonal} (i < j) \end{cases} \)

\( U \) is upper triangular: \( U_{ij} = \begin{cases} 0 & \text{below diagonal} (i > j) \\ \text{arbitrary} & \text{on/above diagonal} (i \leq j) \end{cases} \)

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

Block generic Gaussian elimination

**LU decomposition**: \( A = L \cdot U \), also returns \( L^{-1}, U^{-1} \)

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

Compute \( A_{00} = L_{00} \cdot U_{00} \), also \( L_{00}^{-1}, U_{00}^{-1} \)

\( L_{10} \leftarrow A_{10} \cdot U_{00}^{-1}, \quad U_{01} \leftarrow L_{00}^{-1} \cdot A_{01} \)

\( \tilde{A}_{11} = A_{11} - L_{10} \cdot U_{01} = A_{11} - A_{10} A_{00}^{-1} A_{01} \) (Schur complement of \( A_{11} \))

\[
\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & \cdot \\ \cdot & \tilde{A}_{11} \end{bmatrix} \begin{bmatrix} U_{00} & U_{01} \\ \cdot & I \end{bmatrix}
\]

Compute \( \tilde{A}_{11} = L_{11} \cdot U_{11} \), also \( L_{11}^{-1}, U_{11}^{-1} \), then return \( L^{-1}, U^{-1} \):

\[
L^{-1} \left[ \begin{bmatrix} L_{00}^{-1} \\ -L_{11}^{-1} L_{10} L_{00}^{-1} \end{bmatrix} \right], \quad U^{-1} \left[ \begin{bmatrix} U_{00}^{-1} - U_{00}^{-1} U_{01} U_{11}^{-1} \\ U_{11}^{-1} \end{bmatrix} \right]
\]
Parallel matrix algorithms

Generic Gaussian elimination

Block generic Gaussian elimination (contd.)

A₀₀, . . . : either ordinary elements or blocks, can be applied recursively

Recursion base: 1 × 1 matrix \( A = 1 \cdot A \)

Assumption: pivot elements nonzero (respectively pivot blocks nonsingular):

- \( A₀₀ \neq 0 \) (respectively \( \det A₀₀ \neq 0 \))
- \( \bar{A}_{11} \neq 0 \) (respectively \( \det \bar{A}_{11} \neq 0 \))

Hence no pivoting required

In practice, pivots must be sufficiently large. Holds for some special classes of matrices: diagonally dominant; symmetric positive definite.

Recursive generic Gaussian elimination

Let \( A \) be an \( n \times n \) matrix

\[
A = \begin{bmatrix}
A₀₀ & A₀₁ \\
A₁₀ & A₁₁
\end{bmatrix}
\]

\( A = LU \) by block generic Gaussian elimination on \( A \), then on \( \bar{A}_{11} \)

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

Parallel recursive generic Gaussian elimination

At each level, the two recursive subproblems are dependent, hence recursion tree must be computed depth-first

At recursion level \( k \):

- sequence of \( 2^k \) LU decomposition subproblems, each on \( \frac{n}{2^k} \)-blocks

In particular, at level \( \frac{1}{2} \cdot \log p \):

- sequence of \( p^{1/2} \) LU decomposition subproblems, each on \( \frac{n}{p^{1/2}} \)-blocks

- total \( p^{1/2} \cdot O\left(\left(\frac{n}{p^{1/2}}\right)^3\right) = O\left(\frac{n^3}{p}\right) \) sequential work, therefore each subproblem can be solved sequentially on an arbitrary processor

Iterative generic Gaussian elimination

Let \( A \) be an \( n \times n \) matrix

\[
A = \begin{bmatrix}
A₀₀ & A₀₁ \\
A₁₀ & A₁₁
\end{bmatrix}
\]

\( A = LU \) by block generic Gaussian elimination on \( A \), then on \( \bar{A}_{11} \)

Sequential work \( O(n^3) \)
Parallel matrix algorithms

Generic Gaussian elimination

Parallel recursive generic Gaussian elimination (contd.)

Level \(\frac{1}{2} \cdot \log p\): threshold to switch from parallel to sequential computation

Recursion levels 0 to \(\frac{1}{2} \cdot \log p\):

- block generic LU decomposition using parallel matrix multiplication

Threshold recursion level \(\frac{1}{2} \cdot \log p\):

- a designated processor reads the subproblem’s input block, solves it sequentially, and writes the output blocks

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

Parallel matrix algorithms

Gaussian elimination with pivoting

Pivoting permutes rows/columns of input matrix to remove the assumptions of generic Gaussian elimination, ensuring that:

- pivot elements are always nonzero
- pivot blocks are always nonsingular

Continuous tradeoff between \(\text{comm}\) and \(\text{sync}\)

Controlled by parameter \(\alpha\), \(1/2 \leq \alpha \leq 2/3\)

\(\alpha = 1/2\): \(\text{comm}\) and \(\text{sync}\) as for 3D grid

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

\(\alpha = 2/3\):

- \(\text{comm}\) goes down to that of matrix multiplication
- \(\text{sync}\) goes up accordingly

\[
\text{comp} = O\left(\frac{n^3}{p}\right) \quad \text{comm} = O\left(\frac{n^2}{p^{2/3}}\right) \quad \text{sync} = O\left(p^{2/3}\right)
\]
Let $A, P, L, U$ be $n$-matrices.

**PLU decomposition of $A$:** $P \cdot A = L \cdot U$

- $P$ is a permutation matrix:
  - all elements 0 or 1
  - exactly one 1 in every row and column
- $L$ is unit lower triangular,
- $U$ is upper triangular

The **PLU decomposition** problem: given $A$, find $P, L, U$

---

**Block Gaussian elimination with column pivoting (contd.)**

Let $A$ be an $m \times n$ matrix, $m \geq n$

$A = \begin{bmatrix} A_0 & A_1 \\ A_{10} & A_{11} \end{bmatrix}$

$P$ is a permutation such that $|A_0'|$ is largest across $A$

$P$ is an $m \times m$ permutation matrix

$L_00$ is $n \times n$ unit lower triangular, $U_00$ is $n \times n$ upper triangular
Parallel matrix algorithms
Gaussian elimination with pivoting

Iterative Gaussian elimination with column pivoting
Let $A$ be an $n \times n$ matrix
\[
A = \begin{bmatrix}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{bmatrix}
\]
\[
PA = LU \text{ by block Gaussian elimination with column pivoting on } A,
\]
Sequential work $O(n^3)$

Recursive Gaussian elimination with column pivoting
Let $A$ be an $n \times n$ matrix
\[
A = \begin{bmatrix}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{bmatrix}
\]
\[
PA = LU \text{ by block Gaussian elimination with column pivoting on } A,
\]
Sequential work:
- $O(n^3)$ using standard matrix multiplication
- $O(n^\omega)$ using fast (Strassen-like) matrix multiplication

Parallel recursive Gaussian elimination with column pivoting
At each level, the two recursive subproblems are dependent, hence recursion tree must be computed depth-first
At recursion level $k$:
- sequence of $2^k$ PLU decomposition subproblems, each on $\frac{n}{2^k} \times n$ blocks
In particular, at level $\log p$:
- sequence of $p$ PLU decomposition subproblems, each on $\frac{n}{p} \times n$-blocks
- total $p \cdot O\left(\frac{n^3}{p}\right) = O\left(\frac{n^3}{p}\right)$ sequential work, therefore each subproblem can be solved sequentially on an arbitrary processor

Threshold recursion level $\log p$:
- a designated processor reads the subproblem’s input block, solves it sequentially, and writes the output blocks

\[
\text{comp} = O\left(\frac{n^3}{p}\right) \quad \text{comm} = O(n^2) \quad \text{sync} = O(p)\]
Parallel matrix algorithms
Gaussian elimination with pivoting

Parallel recursive Gaussian elimination with column pivoting (contd.)
Alternative: all recursion levels computed in parallel
Level \( \log p \): threshold to switch from normal to fine-grained computation
Recursion levels 0 to \( \log p \):
- block PLU decomposition using parallel matrix multiplication
Recursion levels \( \log p \) to \( \log n \):
- block PLU decomposition on partitioned matrix, using broadcast of pivot subrows and \( p \) instances of sequential matrix multiplication
Recursion base at level \( \log n \):
- column PLU decomposition; pivot selected by balanced binary tree

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

Parallel graph algorithms
Algebraic path problem

Semiring: a set \( S \) with addition \( \oplus \) and multiplication \( \odot \)
Addition commutative, associative, has identity \( \Box \)
\[ a \oplus b = b \oplus a \quad a \oplus (b \oplus c) = (a \oplus b) \oplus c \quad a \oplus \Box = \Box \oplus a = a \]
Multiplication associative, has annihilator \( \Box \) and identity \( \mathbb{I} \)
\[ a \odot (b \odot c) = (a \odot b) \odot c \quad a \odot \Box = \Box \odot a = \Box \quad a \odot \mathbb{I} = \mathbb{I} \odot a = a \]
Multiplication distributes over addition
\[ a \odot (b \oplus c) = a \odot b \oplus a \odot c \quad (a \oplus b) \odot c = a \odot c \oplus b \odot c \]
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; \( \Box \) by the zero matrix
- \( \odot \) given by matrix multiplication; \( \mathbb{I} \) by the unit matrix
Some specific semirings:

<table>
<thead>
<tr>
<th>$S$</th>
<th>$\oplus$</th>
<th>$\ominus$</th>
<th>$\odot$</th>
<th>$\boxplus$</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 $a$: $a^* = \bigoplus a \ominus a^2 \ominus a^3 \ominus \cdots$

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

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

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 $i \to j$

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

Tropical $A^*$: the graph’s all-pairs shortest paths
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 \( A_{ij} + A_{0j} \), if that gives a shortcut from \( i \) to \( j \)
\[
A'_{11} \leftarrow A_{11} \oplus A_{10} \odot A_{01} = \min(A_{11}, A_{10} + A_{01})
\]
Continue elimination on reduced matrix \( A'_{11} \)
Generic Gaussian elimination in disguise
Sequential work \( O(n^3) \)

Block Floyd–Warshall algorithm
\[
A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}, \quad \quad A^* = \begin{bmatrix} A''_{00} & A''_{01} \\ A''_{10} & A''_{11} \end{bmatrix}
\]
Recursion: two half-sized subproblems
\[
A'_{00} \leftarrow A_{00} \quad A'_{10} \leftarrow A_{00}A_{10} \quad A'_{11} \leftarrow A_{11} \oplus A_{10}A_{01}
\]
\[
A''_{11} \leftarrow (A'_{11})^* \quad \text{by recursion}
\]
\[
A''_{10} \leftarrow A''_{11}A_{10} \quad A''_{01} \leftarrow A''_{11}A_{01} \quad A''_{00} \leftarrow A_{00} \oplus A'_{01}A''_{11}A'_{10}
\]
Block generic Gaussian elimination in disguise
Sequential work \( O(n^3) \)

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 \( \alpha \): \( 1/2 \leq \alpha \leq 2/3 \)
\[
\text{comp} = O(n^3/p) \quad \text{comm} = O(n^2/p^\alpha) \quad \text{sync} = O(p^\alpha)
\]
The all-pairs shortest paths problem: the algebraic path problem over the tropical semiring

\[
\begin{array}{c|ccc}
S & + & 0 & \oplus \\
\hline
\text{tropical} & \mathbb{R}_{\geq 0} \cup \{+\infty\} & \min & +\infty \\
\end{array}
\]

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: \(a \oplus a = \min(a, a) = a\)

Let \(A\) be a matrix of size \(n\) over the tropical semiring, defining a weighted directed graph

\[
A_{ij} = \text{length of the edge } i \rightarrow j
\]

\(A_{ij} \geq 0\) \(A_{ii} = \odot = 0\) \(0 \leq i, j < n\)

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

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

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

\(A_{ij}^* = \text{length of the shortest path } i \leadsto j \text{ (of any size)}\)

The all-pairs shortest paths problem:

\[
A^* = I \oplus A \oplus A^2 \oplus \cdots = I \oplus A \oplus A^2 \oplus \cdots \oplus A^n = (I \oplus A)^n = A^n
\]

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 graph algorithms
All-pairs shortest paths

## 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!

---

### 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 $A_{ij}^\leq k = \text{length of the shortest path } i \rightarrow j \text{ of size } \leq k$

Let $A_{ij}^k = \text{length of the shortest path } i \rightarrow j \text{ of size exactly } k$

We have $A^k = A^\leq k = A^0 \oplus \cdots \oplus A^k$

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

---

### Mult-Dijkstra

- **$\text{comm} = O(n^2)$**
- **$\text{sync} = O(1)$**

### Coming next

- **$\text{comm} = O(n^2/p^{2/3})$**
- **$\text{sync} = O(\log p)$**
Parallel all-pairs shortest paths by selective path doubling (contd.)

Now let $A$ have arbitrary (nonnegative or negative) edge lengths. We still assume there are no negative-length cycles.

All processors compute $A, A^{\leq \frac{3}{2} + \cdots}, A^{\leq p + \cdots}$ by $\leq 2 \log p$ rounds of parallel sparse-by-dense matrix $\odot$-multiplication

Let $A^{-q} = A^{-p} \oplus A^{-2p} \oplus \cdots \oplus A^{-p^2}$

Consider $A = 0, \ldots, A = p^2$ and $A = (p) - q, \ldots, A = (p)$. The total number of non-$\Box$ 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 $2n^2$ non-$\Box$ elements.