Efficient Parallel Algorithms

Dr. Alexander Tiskin

Department of Computer Science
University of Warwick
http://www.dcs.warwick.ac.uk/~tiskin

30 lectures in Spring Term
Exam in Summer Term (results at the end of June)
Website: http://www.dcs.warwick.ac.uk/~tiskin/teach/epa.html
Forum: http://forums.warwick.ac.uk/
departments/computer-science/ugyear3/cs329/
Everybody welcome to participate!

Literature

No core textbook — lecture handouts are provided

Reading list:

For general background in algorithms:

Further reading:
Course outline (V): Parallel matrix algorithms

19 Matrix-vector and matrix-matrix multiplication

20 Triangular system solution

21 Gaussian elimination

Course outline (VI): Parallel graph algorithms

22 Algebraic path problem

23 All-pairs shortest paths

Dr. Alexander Tiskin (Warwick)
Efficient Parallel Algorithms

Part I

Computation by circuits

1 Computation models and algorithms

2 The circuit model

3 The comparison network model

4 Naive sorting networks

5 The zero-one principle

6 Efficient merging and sorting networks
Computation models and algorithms

Model: abstraction of reality allowing qualitative and quantitative reasoning

E.g. atom, galaxy, biological cell, Newton’s universe, Einstein’s universe...

Computation model: abstract computing device to reason about computations and algorithms

E.g. scales+weights, Turing machine, von Neumann machine (“ordinary computer”), JVM, quantum computer...

An algorithm in a specific model: input $\rightarrow$ (computation steps) $\rightarrow$ output

Input/output encoding must be specified

Algorithm complexity (worst-case): $T(n) = \max_{\text{input size}=n}$ computation steps

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

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

Example usage: $O(n^3), \Omega(n \log n), n^{2+o(1)}, \Theta(1)$

All good matrix multiplication algorithms are $O(n^3)$

No comparison-based sorting algorithm is better than $\Omega(n \log n)$

$n^{2+o(1)}$ is worse than $n^2$, but better than $n^{2.000001}$

$\Theta(1)$ can mean any specific constant

Asymptotic growth classes: $O(f), o(f), \Omega(f), \omega(f), \Theta(f)$

$g = O(f)$: "$g$ grows at the same rate or slower than $f$"

- up to a constant factor
- for sufficiently large $n$

$g = o(f) \iff \exists C : \exists n_0 : \forall n \geq n_0 : g(n) \leq C \cdot f(n)$

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

$g = \omega(f)$: "$g$ grows (strictly) slower than $f$"

$g = \Omega(f) \iff \forall c : \exists n_0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)$

In other words: even if we scale $f$ down by any (even very small) constant, it will still eventually overtake and stay above $g$
The circuit model

Basic special-purpose parallel model: a circuit

\[ a^2 + 2ab + b^2 \]
\[ a^2 - b^2 \]

Directed acyclic graph (dag)
Fixed number of inputs/outputs

Oblivious computation: control sequence independent of the input

The comparison network model

A comparison network is a circuit of comparator nodes

\[ x \triangleright y \]
\[ x \sqcap y \]
\[ x \triangleright y \]
\[ x \sqcap y \]

denotes

\[ \sqcap = \min \]
\[ \triangleright = \max \]

The input and output sequences have the same length

Examples:

<table>
<thead>
<tr>
<th>n</th>
<th>size</th>
<th>depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

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

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

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

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

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

What is the complexity of oblivious sorting?
Naive sorting networks

**BUBBLESORT**

- Size: \(n(n-1)/2 = O(n^2)\)
- Depth: \(2n - 3 = O(n)\)

**BUBBLESORT** (8)

- Size: 28
- Depth: 13

**INSERTIONSORT**

- Size: \(n(n-1)/2 = O(n^2)\)
- Depth: \(2n - 3 = O(n)\)

**INSERTIONSORT** (8)

- Size: 28
- Depth: 13

Identical to **BUBBLESORT**!

---

**The zero-one principle**

**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 \(x = (x_1, \ldots, x_n)\)

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

Let \(X_i = \begin{cases} 
0 & \text{if } x_i < y_k \\
1 & \text{if } x_i \geq y_k
\end{cases} \)

and run the network on input \(X = (X_1, \ldots, X_n)\)

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

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

We have \(k < l\) but \(Y_k > Y_l\), so the network does not sort 0s and 1s.

---

**The zero-one principle**

The zero-one principle applies to sorting, merging and other comparison problems (e.g. selection).

It allows 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 \(\binom{n}{n'} = \binom{n}{n''}\)
Efficient merging and sorting networks

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

How fast can we merge obliviously?

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

**Odd-even merging**

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

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

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

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

---

Efficient merging and sorting networks

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

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

**Inductive step:** By the inductive hypothesis, we have for all $k, l$:

\[
\langle 0^{[k/2]}11\ldots \rangle, \langle 0^{[l/2]}11\ldots \rangle \mapsto \langle 0^{[k/2]+[l/2]}11\ldots \rangle
\]

\[
\langle 0^{[k/2]}11\ldots \rangle, \langle 0^{[l/2]}11\ldots \rangle \mapsto \langle 0^{[k/2]+[l/2]}11\ldots \rangle
\]

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

\[
(\lfloor k/2 \rfloor + \lfloor l/2 \rfloor) - (\lfloor k/2 \rfloor + \lfloor l/2 \rfloor) =
\]

\[
\begin{cases}
0,1 & \text{result sorted: } \langle 0^{k+l}11\ldots \rangle \\
2 & \text{single pair wrong: } \langle 0^{k+l-1}1011\ldots \rangle
\end{cases}
\]

The final stage of comparators corrects the wrong pair

---

Efficient merging and sorting networks

\[OEM(n', n'')\]

- size $O(n \log n)$
- depth $O(\log n)$

\[n' \leq n''\]

\[OEM(4, 4)\]

- size 9
- depth 3

---

Efficient merging and sorting networks

Sorting an arbitrary input $\langle x_1, \ldots , x_n \rangle$ [Batcher: 1968]

When $n = 1$ we are done, otherwise

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

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

\[\text{depth}_{OEMSORT}(n) \leq \text{depth}_{OEMSORT}(n/2) + \text{depth}_{OEM}(n) = \text{depth}_{OEMSORT}(n/2) + O(\log n) = O((\log n)^2)\]
Bitonic sequence: \( \langle x_1 \geq \cdots \geq x_m \leq \cdots \leq x_n \rangle \)

Bitonic merging: sorting a bitonic sequence

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

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

Correctness proof: by zero-one principle (exercise)

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

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

\[
\text{size}_{BM}(n) = O(n \log n) \quad \text{depth}_{BM}(n) = O(\log n)
\]
Efficient merging and sorting networks

Both $OEMSORT$ and $BITONICSORT$ have size $\Theta(n(\log n)^2)$

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

$AKS$ sorting

[Altaï, Komlós, Szemerédi: 1983]

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

Uses sophisticated graph theory ($expander$)

Asymptotically optimal, but has huge constant factors

Part II

Parallel computation models

- The PRAM model
- The BSP model
- Network topologies
- Oblivious routing
- Randomised routing
The PRAM model


![PRAM diagram]

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

---

The BSP model

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

![BSP diagram]

Realistic general-purpose parallel model

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

BSP computation: sequence of parallel supersteps

Asynchronous computation/communication within supersteps (includes data exchange with external memory)
Synchronisation before/after each superstep
Cf. CSP: parallel collection of sequential processes

Compositional cost model

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

For the whole BSP computer in one superstep \( sstep \):
- \( \text{comp}(sstep) = \max_{0 \leq proc < p} \text{comp}(sstep, proc) \)
- \( \text{comm}(sstep) = \max_{0 \leq proc < p} \text{comm}(sstep, proc) \)
- \( \text{cost}(sstep) = \text{comp}(sstep) + \text{comm}(sstep) \cdot g + l \)

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

For the whole BSP computation with \( sync \) supersteps:
- \( \text{comp} = \sum_{0 \leq sstep < sync} \text{comp}(sstep) \)
- \( \text{comm} = \sum_{0 \leq sstep < sync} \text{comm}(sstep) \)
- \( \text{cost} = \sum_{0 \leq sstep < sync} \text{cost}(sstep) = \text{comp} + \text{comm} \cdot g + \text{sync} \cdot l \)

BSP computation: scalable, portable, predictable
BSP algorithm design: minimising \( \text{comp} \), \( \text{comm} \), \( \text{sync} \)
Main principles:
- load balancing minimises \( \text{comp} \)
- data locality minimises \( \text{comm} \)
- coarse granularity minimises \( \text{sync} \)

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

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

\( p = q^3 \) processors

degree 6

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

Butterfly network

\( p = q \log q \) processors

degree 4

diameter \( \approx \log p \approx \log q \)

2D array network

\( p = q^2 \) processors

degree 4

diameter \( p^{1/2} = q \)
Network topologies

*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>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>
<tr>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

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

Oblivious routing

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

E.g. 1-relation = permutation

Once we can route any permutation in \( k \) steps, we can route any

\( h \)-relation in \( h k \) steps

In the worst case, we may always be forced to make \( \Omega(\text{diameter}) \) steps

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

Assume *distributed* routing: no global information

*Oblivious routing*: path determined only by source and destination

E.g. *greedy routing*: always take shortest path

"Hot spots": for \( h = 1 \), degree \( d \), any oblivious routing method may be forced to make \( \Omega(p^{1/2}/d) \) steps

Many practical patterns force "hot spots" on traditional networks

Routing means “sorting” by destination address

Routing based on sorting networks (non-oblivious for individual packets!)

Processors correspond to wires

Links correspond to comparators (possibly one-to-many)

Each stage of routing corresponds to a stage of sorting

<table>
<thead>
<tr>
<th>Network</th>
<th>Degree</th>
<th>Diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>OEMSORT/BSORT</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": for \( h = 1 \), can always route packets in \( O(\text{diameter}) \) steps

Not practical due to messy wiring
Randomised routing

Two-phase randomised routing: [Valiant, 1980]
- send every packet to random intermediate destination
- forward every packet to final destination

Both phases oblivious (e.g. greedy)

Hot spots very unlikely

2D array, butterfly, hypercube:
  for $h = 1$, $O(\text{diameter})$ steps with high probability

Hypercube: $O(\text{diameter})$ steps even for $h = \log p$

Justifies the BSP model

Randomised routing

BSP implementation: processes placed at random, communication delayed until end of superstep

All packets with same source and destination sent together, hence message overhead absorbed in $l$

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

Actual values of $g$, $l$ obtained by benchmarking

www.bsp-worldwide.org/implmnts/oxtool

Part III

Basic parallel algorithms

- Broadcast/combine
- Balanced tree and prefix sums
- Fast Fourier Transform and the butterfly dag
- Ordered grid
- Discussion
The **broadcasting** problem:
- initially, one designated processor holds a value $a$
- at the end, every processor must hold a copy of $a$

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

By symmetry, we only need to consider broadcasting

Binary tree broadcast:
- initially, only designated processor is awake
- processors are woken up in $\log p$ rounds
- in every round, every awake processor makes a copy of $a$ and send it to a sleeping processor, waking it up

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

**Direct broadcast:**
- designated processor makes $p - 1$ copies of $a$ and sends them directly to destinations

*comp $O(p)$  comm $O(p)$  sync $O(1)$*

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

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

(effectively, $n$ independent instances of broadcasting/combining)
**Broadcast/combine**

*Two-phase array broadcast:*
- partition array into $p$ blocks of size $n/p$
- *scatter* blocks, then *total-exchange* blocks

```
A   B
C   D
```

comp $O(n)$  comm $O(n)$  sync $O(1)$

**Balanced tree and prefix sums**

Parallel balanced tree computation

From now on, we always assume that a problem’s input/output is stored in the external memory.

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

```
A   B   C   D
```

```
A B C D
A B C D
A B C D
A B C D
```

Sequential work $O(n)$

```
comp $O(n/p)$  comm $O(n/p)$  sync $O(1)$
```

**Balanced tree and prefix sums**

The *balanced binary tree dag:*
- a generalisation of broadcasting/combining
- can be defined top-down (root the input, leaves the outputs) or bottom-up

```
tree(n)
1 input
n outputs
size $n - 1$
depth $\log n$
```

For bottom-up computation, reverse the steps

$n \geq p^2$

```
comp $O(n/p)$  comm $O(n/p)$  sync $O(1)$
```
The described parallel balanced tree algorithm has:

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

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

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

---

**The prefix circuit**

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

[Diagram of the prefix circuit]

where \( a_{k:i} = a_k \cdot a_{k+1} \cdot \ldots \cdot a_i \), and “*” is a dummy value.

The underlying dag is called the **prefix dag**.
Balanced tree and prefix sums

Parallel prefix computation

The dag $\text{prefix}(n)$ consists of

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

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

$n \geq p^2$

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

Balanced tree and prefix sums

$x + y = z$

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

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

Let $u_i = x_i \land y_i \quad v_i = x_i \lor y_i \quad 0 \leq i < n$

Arrays $u = \langle u_{n-1}, \ldots, u_0 \rangle$, $v = \langle v_{n-1}, \ldots, v_0 \rangle$ can be computed in size $O(n)$ and depth $O(1)$

\[
\begin{align*}
z_0 &= v_0 & c_0 &= u_0 \\
z_1 &= v_1 \oplus c_0 & c_1 &= u_1 \lor (v_1 \land c_0) \\
\vdots & & \vdots \\
z_{n-1} &= v_{n-1} \oplus c_{n-2} & c_{n-1} &= u_{n-1} \lor (v_{n-1} \land c_{n-2}) \\
z_n &= c_{n-1}
\end{align*}
\]

Resulting circuit has size and depth $O(n)$. Can we do better?
Efficient Parallel Algorithms

Fast Fourier Transform and the butterfly dag

Let $\omega, \omega^2, \ldots, \omega^{n-1} \neq 1 \quad \omega^n = 1$

Such an $\omega$ is called a primitive root of unity of degree $n$, exists for every $n$ in complex numbers

Let $F_{n, \omega} = [\omega^i j]_{i,j=0}^{n-1}$

The Discrete Fourier Transform problem: $F_{n, \omega}(a) = F_{n, \omega} \cdot a = b$

\[
\begin{bmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\
1 & \omega^3 & \omega^4 & \cdots & \omega^{n-2} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{n-1} & \omega^{n-2} & \cdots & \omega \\
\end{bmatrix}
\begin{bmatrix}
a[0] \\
a[1] \\
a[2] \\
\vdots \\
a[n-1] \\
\end{bmatrix}
= \begin{bmatrix}
b[0] \\
b[1] \\
b[2] \\
\vdots \\
b[n-1] \\
\end{bmatrix}
\]

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

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

Let $A[u, v] = a[mu + v]$  $B[s, t] = b[ms + t]$  $s, t, u, v = 0, \ldots, m - 1$

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

We have $B[s, t] = \sum_{u, v} \omega^{(ms + tu + mtu)} A[u, v] = \sum_{u, v} \omega^{mv} \cdot \omega^{tv} \cdot \sum_{u} (\omega^m)^u A[u, v]$

We have $B = F_{m, \omega^m}(T_{m, \omega}(F_{m, \omega^m}(A)))$, where

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

$F_{m, \omega^m}(A) = F_{m, \omega^m} \cdot A$  $F_{m, \omega^m}(A)[t, v] = \sum_{u} (\omega^m)^u A[u, v]$

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

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

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

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 $m$

By recursion, we have the FFT circuit

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

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

The FFT circuit

The underlying dag is called butterfly dag
Fast Fourier Transform and the butterfly dag

The FFT circuit and the butterfly dag (contd.)

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

Applications: Fast Fourier Transform; sorting bitonic sequences

Ordered grid

The ordered 2D grid dag

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

Applications: Gauss–Seidel iteration (single step); triangular system solution; dynamic programming; 1D cellular automata

Sequential work $O(n^2)$
Ordered grid

Parallel ordered 2D grid computation

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

Consists of \( p^2 \) regular blocks, each isomorphic to \( \text{grid}_2(n/p) \)

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

\[
\begin{array}{c}
\text{comp: } (2p - 1) \cdot O((n/p)^2) = O(p \cdot n^2/p^2) = O(n^2/p) \\
\text{comm: } (2p - 1) \cdot O(n/p) = O(n) \\
n \geq p
\end{array}
\]

Ordered grid

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 \( b \)

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

\[ \text{LCS}(a, b) = \text{“dein”} \]

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

LCS computation by dynamic programming

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

\[
lcs(\text{“”}, \text{“”}) = 0 \quad lcs(a, \text{“”}) = \begin{cases} \max(lcs(a, b), lcs(a, b\beta)) & \text{if } \alpha \neq \beta \\ lcs(a, b) + 1 & \text{if } \alpha = \beta \end{cases}
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
* & d & e & f & i \\
\hline
* & 0 & 0 & 0 & 0 & 0 \\
d & 0 & 1 & 1 & 1 & 1 \\
e & 0 & 1 & 2 & 2 & 2 \\
s & 0 & 1 & 2 & 2 & 2 \\
i & 0 & 1 & 2 & 3 & 3 \\
g & 0 & 1 & 2 & 3 & 3 \\
n & 0 & 1 & 2 & 3 & 4 \\
\hline
\end{array}
\]

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

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

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

Parallel LCS computation

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

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

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

The semi-local LCS algorithm (details omitted) [Krusche, Tiskin, 2007]

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

Ordered grid

The ordered 3D grid dag

\(\text{grid}_3(n)\)

nodes arranged in an \(n \times n \times n\) grid
edges directed top-to-bottom, left-to-right, front-to-back
\(\leq 3n^2\) inputs (to front/left/top faces)
\(\leq 3n^2\) outputs (from back/right/bottom faces)
size \(n^3\) depth \(3n - 2\)
Applications: Gauss–Seidel iteration; Gaussian elimination; dynamic programming; 2D cellular automata
Sequential work \(O(n^3)\)

Parallel ordered 3D grid computation

\(\text{grid}_3(n)\)

Consists of \(p^{3/2}\) regular blocks, each isomorphic to \(\text{grid}_3(n/p^{1/2})\)
The blocks can be arranged into \(3p^{1/2} - 1\) layers orthogonal to the main diagonal, with \(O(p)\) independent blocks in each layer

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

Parallel ordered 3D grid computation (contd.)

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

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

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

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

\(n \geq p^{1/2}\)
Discussion

Typically $n \gg p$; $\text{comp}$, $\text{comm}$, $\text{sync}$ are functions of $n, p$

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

The challenges:
- efficient (optimal) algorithms
- good (sharp) lower bounds

Further parallel algorithms

List contraction and colouring

Linked list: $n$ nodes, each contains data and a pointer to successor

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

Primitive list operation: pointer jumping

The original node data $a$, $b$ and the pointer to $b$ are kept, so that the pointer jumping operation can be reversed
List contraction and colouring

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

\[
\begin{array}{ccc}
\circ & \rightarrow & a \\
\circ & \rightarrow & a \bullet b \\
\end{array}
\]

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

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

**List expansion**: restore the original list by reversing the contraction

Dr. Alexander Tiskin (Warwick)
Efficient Parallel Algorithms
93 / 182

Application: list ranking (contd.)

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

Contraction phase: for each node keep the length of its sublist

Initially, each node assigned 1

Merging operation: \( k, l \rightarrow k + l \)

In the fully contracted list, the node contains value \( n \)

Dr. Alexander Tiskin (Warwick)
Efficient Parallel Algorithms
94 / 182

Application: list ranking (contd.)

Expansion phase: for each node keep

- the rank of the starting node of its sublist
- the length of its sublist

Initially, the node (fully contracted list) assigned \((0, n)\)

Un-merging operation: \((s, k), (s + k, l) \leftarrow (s, k + l)\)

In the fully expanded list, a node with rank \( i \) contains \((i, 1)\)
List contraction and colouring

Application: list prefix sums

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

$\bullet \omega \rightarrow a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow a_7$

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

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

$b_0 \rightarrow b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4 \rightarrow b_5 \rightarrow b_6 \rightarrow b_7$

Note the solution should be independent of the merging order!

List contraction and colouring

Application: list prefix sums (contd.)

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

Contraction phase: for each node keep the $\bullet$-sum of its sublist

Initially, each node assigned $a_i$;

Merging operation: $u, v \rightarrow u \bullet v$

In the fully contracted list, the node contains value $b_{n-1}$

List contraction and colouring

Application: list prefix sums (contd.)

From now on, we only consider pure list contraction (the expansion phase is obtained by symmetry).

Sequential work $O(n)$ by always contracting at the list’s head.

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

Therefore, we need either node copying, or efficient symmetry breaking.

Expansion phase: for each node keep

- the $\bullet$-sum of all nodes before its sublist
- the $\bullet$-sum of its sublist

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

Un-merging operation: $(t, u), (t \bullet u, v) \leftarrow (t, u \bullet v)$

In the fully expanded list, a node with rank $i$ contains $(b_{i-1}, a_i)$

We have $b_i = b_{i-1} \bullet a_i$
List contraction and colouring

**Wyllie’s mating**  
[Wyllie, 1979]

Copy every node, label one copy ("forward") and the other ("backward")

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

--

**Random mating**  
[Miller, Reif, 1985]

Label every node ("forward") or ("backward") independently with probability 1/2

A node has a mate with probability 1/2, hence on average $n/2$ nodes have mates

Merge mating pairs, obtaining a new list of expected size $3n/4$

--

Parallel list contraction by Wyllie’s mating

Initially, each processor reads a subset of $n/p$ nodes

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

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

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

$n \geq p$

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

--

Parallel list contraction by random mating

Initially, each processor reads a subset of $n/p$ nodes

- reduce list to expected size $n/p$ by $\log_{4/3} p$ rounds of random mating
- collect the current list in a designated processor and contract sequentially

Total work $O(n)$, optimal but randomised with high probability (whp)

Formally, $\exists C, d > 0 : \text{Prob}(\text{total work} \leq C \cdot n) \geq 1 - 1/n^d$

$n \geq p^2 \cdot \log p$

\[
\begin{align*}
\text{comp } & O(n/p) \text{ whp} \\
\text{comm } & O(n/p) \text{ whp} \\
\text{sync } & O(\log p)
\end{align*}
\]
Deterministic block mating

Contract local chains (if any)

Build distribution graph:
- complete weighted directed graph on \( p \) nodes
- \( \text{weight}(i,j) = |\{u \rightarrow v : u \in \text{proc}_i, v \in \text{proc}_j\}| \)

Each processor holds the corresponding node’s outgoing edges

Collect distribution graph in a designated processor

By a sequential greedy algorithm, label every processor node “forward” or “backward”, so that
\[
\sum_{i \in F, j \in B} \text{weight}(i,j) \geq \frac{1}{4} \cdot \sum_{i,j} \text{weight}(i,j)
\]

Distribute labels back to processors

By construction of the labeling, at least \( n/2 \) nodes have mates

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

Parallel list contraction by deterministic block mating

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

- reduce list to size \( n/p \) by \( \log_{4/3} p \) rounds of deterministic block mating
- collect the current list in a designated processor and contract sequentially

Total work \( O(n) \), optimal and deterministic

\[ n \geq p^3 \]

\[
\begin{align*}
\text{comp } O(n/p) & \quad \text{comm } O(n/p) & \quad \text{sync } O(\log p)
\end{align*}
\]
List contraction and colouring

List k-colouring: assign a colour from 0, ..., k − 1 to every node, so that no two adjacent nodes have the same colour

Require \( \text{comp } O(n/p), \text{comm } O(n/p) \)

Note: k-colouring can be done for any k in \( \text{sync } O(\log p) \) by list contraction

\( p \)-colouring can be easily done in \( \text{sync } O(1) \)

Is faster \( k \)-colouring possible for \( k \leq p \)? For \( k = O(1) \)?

Deterministic coin tossing [Cole, Vishkin, 1986]

Given an initial \( k \)-colouring represented in binary

Consider every node \( v \). We have \( \text{colour}(v) \neq \text{colour} \left( \text{next}(v) \right) \).

If \( \text{colour}(v) \) differs from \( \text{colour} \left( \text{next}(v) \right) \) in \( i \)-th bit, re-colour \( v \) in

- \( 2i \), if \( i \)-th bit is 0 in \( \text{colour}(v) \) and 1 in \( \text{colour} \left( \text{next}(v) \right) \)
- \( 2i + 1 \), if \( i \)-th bit is 1 in \( \text{colour}(v) \) and 0 in \( \text{colour} \left( \text{next}(v) \right) \)

Results in a \( 2 \log k \)-colouring of the list for \( k > 6 \)

List contraction and colouring

Parallel list 3-colouring by deterministic coin tossing:

- compute a \( p \)-colouring
- reduce the number of colours from \( p \) to 6 by deterministic coin tossing: \( O(\log^* k) \) rounds
  \[
  \log^* k = \min r : \log \ldots \log k \leq l \quad (r \text{ times})
  \]

- select node \( v \) as a pivot, if \( \text{colour}(\text{prev}(v)) > \text{colour}(v) < \text{colour}(\text{next}(v)) \). No two pivots are adjacent or further than 12 nodes apart
- from each pivot, re-colour the succeeding run of at most 12 non-pivots sequentially in 3 colours

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

Sorting and convex hull

Sorting \( \langle x_0, \ldots, x_{n-1} \rangle \)

May assume all \( x_i \) are distinct (otherwise, attach unique tags)

Sequential work \( O(n \log n) \) e.g. by mergesort

Parallel sorting based on an AKS sorting network

\( \text{comp } O(\frac{n \log n}{p}), \text{comm } O(\frac{n \log n}{p}), \text{sync } O(\log n) \)
Every processor
- reads a subarray of size \( n/p \) and sorts it sequentially
- selects from its subarray \( p + 1 \) regular samples

A designated processor
- collects all \( p(p + 1) \) samples and sorts them sequentially
- selects from the sorted samples \( p + 1 \) regular splitters

The designated processor broadcasts the splitters
Every processor
- receives the splitters and is assigned a bucket
- scans its subarray and sends each element to the appropriate bucket
- receives the elements of its bucket and sorts them sequentially
- writes the sorted bucket back to external memory

\[ n \geq p^3 \]
\[
\text{comp } O\left(\frac{n \log n}{p}\right) \quad \text{comm } O(n/p) \quad \text{sync } O(1)
\]
Sorting and convex hull

\( X = \{x_0, \ldots, x_{n-1}\} \subseteq \mathbb{R}^d \)

The (discrete) convex hull problem: find vertices of \( \text{conv} \ X \)

Output must be ordered: every vertex must “know” its neighbours

- \( d = 2 \): two neighbours per vertex; output size \( 2n \)
- \( d = 3 \): on average, \( O(1) \) neighbours per vertex; output size \( O(n) \)

Sequential work \( O(n \log n) \) by “gift wrapping”

In higher dimensions, each vertex potentially has a large number of neighbours; typical output size is much higher than \( \Omega(n) \)

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

Claim: Convex hull in \( \mathbb{R}^2 \) is at least as hard as sorting

Proof. Let \( x_0, \ldots, x_{n-1} \in \mathbb{R} \)

To sort \( \langle x_0, \ldots, x_{n-1} \rangle \):

- compute \( \text{conv} \{ (x_i, x_{2i}) \in \mathbb{R}^2 : 0 \leq i < n \} \)
- follow the neighbour links to obtain sorted output

Can the convex hull problem be solved at the same BSP cost as sorting?

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

Proof. Generalised “gift wrapping”.

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

Proof. Take every \( r \)-th point on the convex hull of \( X \).
Sorting and convex hull

\[ X \subseteq \mathbb{R}^3 \quad |X| = n \quad \epsilon = 1/r \]

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

Proof. [Matoušek, 1992]

(Better approximations exist, but are harder to compute)

Parallel convex hull computation by \textit{generalised regular sampling}

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

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

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

The designated processor broadcasts the splitters

Every processor

- reads a subset of \( n/p \) points and computes its convex hull
- selects from the hull a \( 1/p \)-approximation of \( O(p^3 \log p) \) points as \textit{samples}

The global union of all samples is a \( 1/p \)-approximation for \( X \)

A designated processor

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

The set of splitters is an \( 1/p \)-net for a \( 1/p \)-approximation for \( X \). Therefore, it is a \( 2/p \)-net for \( X \).
Matrix algorithms

Part V

Matrix algorithms

19 Matrix-vector and matrix-matrix multiplication

20 Triangular system solution

21 Gaussian elimination

Matrix-vector and matrix-matrix multiplication

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

The matrix-vector multiplication problem: $A \cdot b = c$

$c[i] = \sum_j A[i,j] \cdot b[j]$

$0 \leq i, j < n$

Sequential work $O(n^2)$

The matrix-vector multiplication circuit

$c[i] \leftarrow 0$

$c[i] \leftarrow A[i,j] \cdot b[j]$

$0 \leq i, j < n$

size $O(n^2)$, depth $O(1)$
Matrix-vector and matrix-matrix multiplication

Parallel matrix-vector multiplication

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

Partition the square of elementary product nodes $A[i,j] \cdot b[j]$ into $p$ regular square blocks

Matrix $A$ gets partitioned into $p$ square blocks $A[[i,j]]$ of size $n/p^{1/2}$

Vectors $b, c$ each gets partitioned into $p^{1/2}$ linear blocks $b[j], c[i]$ of size $n/p^{1/2}$

Let $A, B, C$ be matrices of size $n$

The matrix multiplication problem: $A \cdot B = C$

$C[i, k] = \sum_j A[i,j] \cdot B[j, k]$

$0 \leq i, j, k < n$

Sequential work $O(n^3)$

Individual asynchronous updates to $c[i]$ add up to the correct final value $n \geq p$

\[\text{comp } O(n^2/p)\quad\text{comm } O(n/p^{1/2})\quad\text{sync } O(1)\]
Matrix-vector and matrix-matrix multiplication

The matrix multiplication circuit

\[ C[i, k] \leftarrow 0 \]

\[ C[i, k] \leftarrow A[i, j] \cdot B[j, k] \]

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

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

Parallel matrix multiplication

Partition the cube of elementary product nodes \( A[i, j] \cdot B[j, k] \) into \( p \) regular cubic blocks

Matrices \( A, B, C \) each gets partitioned into \( p^{2/3} \) square blocks \( A[i, j], B[j, k], C[i, k] \) of size \( n/p^{1/3} \)

Parallel matrix multiplication (contd.)

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

Every processor

- is assigned to compute a block product \( A[i, j] \cdot B[j, k] = C^[i, k] \)
- reads blocks \( A[i, j], B[j, k] \), and computes \( C^[i, k] \)
- updates block \( C[i, k] \) in external memory by adding to each element the corresponding element of \( C^[i, k] \)

Individual asynchronous updates to \( C[i, k] \) add up to the correct final value

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

\[ \text{comp } O(n^3/p) \quad \text{comm } O(n^2/p^{2/3}) \quad \text{sync } O(1) \]
Matrix-vector and matrix-matrix multiplication

Theorem. Computing the matrix multiplication dag by regular block partitioning is asymptotically optimal.

Proof: \( \text{comp } O(n^3/p), \text{ sync } O(1) \) trivially optimal; optimality of \( \text{comm } O(n^2/p^{2/3}) \) proved by discrete isoperimetry (volume vs surface area).

Let \( V \subseteq \mathbb{Z}^3 \) be the set of nodes computed by a certain processor.

For at least one processor, we have \( |V| \geq n^3/p \).

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

Arithmetic-geometric mean inequality:
\[
(|A| \cdot |B| \cdot |C|)^{1/3} \leq \frac{1}{3} \cdot (|A| + |B| + |C|)
\]

Loomis–Whitney inequality:
\[
|V|^2 \leq |A| \cdot |B| \cdot |C|
\]

We have \( \text{comm} \geq |A| + |B| + |C| \geq 3(|A| \cdot |B| \cdot |C|)^{1/3} \geq 3|V|^{2/3} \geq 3(n^3/p)^{2/3} = 3n^2/p^{2/3} \).

Hence \( \text{comm} = \Omega(n^2/p^{2/3}) \).

Matrix-vector and matrix-matrix multiplication

Let \( A, B, C \) be numerical matrices of size \( n \).

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

Primitives \(+, -, \cdot\) on matrix elements.

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

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

The number of linear operations turns out to be irrelevant. The asymptotic work is determined by \( N \) and \( R \).

Let \( \omega = \log_2 R < \log_2 N^3 = 3 \).

Resulting dag has size \( O(n^\omega) \), depth \( \approx 2 \log n \).

Sequential work \( O(n^\omega) \).
Matrix-vector and matrix-matrix multiplication

Some specific instances of Strassen-type scheme:

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N^3$</th>
<th>$R$</th>
<th>$\omega = \log/N$ $R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>7</td>
<td>2.81</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>23</td>
<td>2.85</td>
</tr>
<tr>
<td>5</td>
<td>125</td>
<td>100</td>
<td>2.86</td>
</tr>
<tr>
<td>48</td>
<td>110592</td>
<td>47216</td>
<td>2.78</td>
</tr>
</tbody>
</table>

Best known $\omega \approx 2.38$, with astronomical $N$, $R$

[Strassen, 1969]

[Dr. Alexander Tiskin (Warwick)]

Efficient Parallel Algorithms

141 / 182

Matrix-vector and matrix-matrix multiplication

Parallel Strassen-type matrix multiplication

Assign matrix elements to processors by the cyclic distribution:

- partition $A$ into regular blocks of size $p^{1/2}$
- distribute each block identically, one element per processor
- partition $B$, $C$ analogously (distribution identical across all blocks of the same matrix, but need not be identical across different matrices)

The cyclic distribution allows us to perform linear operations on matrix blocks of size at least $p^{1/2}$ without communication

---

Parallel Strassen-type matrix multiplication (contd.)

At the top level, $R$ parallel recursive calls. Recursion tree unfolded breadth-first

<table>
<thead>
<tr>
<th>level</th>
<th>tasks</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>$R$</td>
<td>$n/N$</td>
</tr>
<tr>
<td>2</td>
<td>$R^2$</td>
<td>$n/N^2$</td>
</tr>
<tr>
<td>...</td>
<td>$\log/R$ $p$</td>
<td>$n/\log/N$ $p$</td>
</tr>
<tr>
<td>...</td>
<td>$\log$/N $n$</td>
<td>$R^{\log/N}$ $n$</td>
</tr>
</tbody>
</table>

The tasks at each level are independent and can be done in parallel

$\text{comp } O(n^\omega/p)$  $\text{comm } O(n^2/p^{1/2}/\omega)$: opt?  $\text{sync } O(1)$
Matrix-vector and matrix-matrix multiplication

Let $A$, $B$, $C$ be Boolean matrices of size $n$

**Boolean matrix multiplication:** $A \land B = C$

Primitives $\lor$, $\land$, $\text{if/then}$ on matrix elements

$C[i, k] = \lor_j A[i, j] \land B[j, k]$

$0 \leq i, j, k < n$

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

Parallel Boolean matrix multiplication

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

**Regularity Lemma:** in any Boolean matrix, one can select $c = O(1)$ rows and $c = O(1)$ columns, so that $c^2$ resulting submatrices are random-like

[Szemerédi, 1978]

More precisely: $c = f(\epsilon)$, where $\epsilon$ is the “degree of random-likeness” (in our setup, a function of $p$). Function $f(\epsilon)$ is huge, but independent of $n$.

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

Parallel Boolean matrix multiplication (contd.)

$A \land B = C$

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

If $A, B, C$ are arbitrary, by Regularity Lemma we have the *three-way regular decomposition* $C = C_1 \lor C_2 \lor C_3$, such that

- $A_1 \land B_1 = C_1$, where $A_1$ has very few 1s
- $A_2 \land B_2 = C_2$, where $B_2$ has very few 1s
- $A_3 \land B_3 = C_3$, where $C_3$ has very few 0s

All the above matrices are composed of $O(1)$ (more precisely, $f(\epsilon)$) random-like submatrices

Partition the cube of elementary product nodes $A[i, j] \land B[j, k]$ into $p^3$ regular cubic blocks

Matrices $A$, $B$, $C$ each gets partitioned into $p^2$ square blocks $A[[i, j]]$, $B[[j, k]]$, $C[[i, k]]$ of size $n/p^{2/3}$
Matrix-vector and matrix-matrix multiplication

Parallel Boolean matrix multiplication (contd.)

Every processor

- is assigned to compute a “slab” of $p^2$ block products
  $A[i,j] \land B[j,k] = C^{(j)}[i,k]$ for a fixed $j$ and all $i, k$
- reads blocks $A[i,j], B[j,k]$ and computes all $C^{(j)}[i,k]$
- computes the three-way regular decompositions of all $C^{(j)}[i,k]$ and determines the submatrices having very few 0s or 1s

Recompute $A \land B = C$, using the knowledge of the blocks’ three-way regular decompositions (details omitted). We save on communication of submatrices by only sending the positions of 0s or 1s, whichever are fewer.

\[ \text{comp } O(n^\omega \cdot 2^{O(\log p)}) : -0 \quad \text{comm } O(n^2/p) \quad \text{sync } O(1) \]

Dr. Alexander Tiskin (Warwick)
Efficient Parallel Algorithms
149 / 182

Triangular system solution

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

$L$ is lower triangular: $L[i,j] = \begin{cases} 0 & 0 \leq i < j < n \\ \text{arbitrary} & \text{otherwise} \end{cases}$

The triangular system problem: given $L, c$, find $b$ such that $L \cdot b = c$

\[ \sum_j L[i,j] \cdot b[j] = c[i] \quad 0 \leq j \leq i < n \]

Sequential work $O(n^2)$

Dr. Alexander Tiskin (Warwick)
Efficient Parallel Algorithms
151 / 182

Triangular system solution

Forward substitution

$L \cdot b = c$

\[
\begin{align*}
    b[0] & \leftarrow L[0,0]^{-1} \cdot c[0] \\
    b[1] & \leftarrow L[1,1]^{-1} \cdot (c[1] - L[1,0] \cdot b[0]) \\
    \vdots \\
    b[i] & \leftarrow L[i,i]^{-1} \cdot (c[i] - \sum_{j<i} L[i,j] \cdot b[j]) \\
    \vdots \\
    b[n-1] & \leftarrow L[n-1,n-1]^{-1} \cdot (c[n-1] - \sum_{j<i} L[n-1,j] \cdot b[j])
\end{align*}
\]

Parallel forward substitution by 2D grid dag

Assume $L$ is predistributed as needed, does not count as input

Sequential work $O(n^2)$

Dr. Alexander Tiskin (Warwick)
Efficient Parallel Algorithms
152 / 182
Block forward substitution (divide-and-conquer)

\[ L \cdot b = c \]

\[
\begin{bmatrix}
L_{11} & \bullet \\
L_{21} & L_{22}
\end{bmatrix}
\begin{bmatrix}
b_1 \\
b_2
\end{bmatrix} =
\begin{bmatrix}
c_1 \\
c_2
\end{bmatrix}
\]

\[ L_{11} \cdot b_1 = c_1 \]
\[ L_{22} \cdot b_2 = c_2 - L_{21} \cdot b_1 \]

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

Parallel block forward substitution (contd.)

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

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

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

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

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

Gaussian elimination

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

\[ L \text{ is unit lower triangular: } L[i,j] = \begin{cases} 0 & 0 \leq i < j < n \\ 1 & 0 \leq i = j < n \\ \text{arbitrary} & \text{otherwise} \end{cases} \]

\[ U \text{ is upper triangular: } U[i,j] = \begin{cases} 0 & 0 \leq j < i < n \\ \text{arbitrary} & \text{otherwise} \end{cases} \]

The \(LU\) decomposition problem: given \(A\), find \(L\), \(U\) such that \(A = L \cdot U\)

\[ A[i,k] = \sum_j L[i,j] \cdot U[j,k] \]

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

Generic Gaussian elimination
\[
\begin{bmatrix}
  a_{11} & a_{12} \\
  a_{21} & a_{22}
\end{bmatrix} = \begin{bmatrix}
  1 & \bullet \\
  a_{21}/a_{11} & 1
\end{bmatrix} \cdot \begin{bmatrix}
  a_{11} & a_{12} \\
  \bullet & a_{22} - a_{12} \cdot a_{21}/a_{11}
\end{bmatrix}
\]
Assume no pivoting required: \( a_{11} \neq 0 \)
Sequential work \( O(n^3) \)
Parallel generic Gaussian elimination: 3D grid (details omitted)

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

Gaussian elimination

Block generic Gaussian elimination (divide-and-conquer)
\[
A = L \cdot U
\]
\[
\begin{bmatrix}
  A_{11} & A_{12} \\
  A_{21} & A_{22}
\end{bmatrix} = \begin{bmatrix}
  L_{11} & \bullet \\
  L_{21} & L_{22}
\end{bmatrix} \cdot \begin{bmatrix}
  U_{11} & U_{12} \\
  \bullet & U_{22}
\end{bmatrix}
\]
\[A_{11} = L_{11} \cdot U_{11}\]
\[U_{12} \leftarrow L_{11}^{-1} \cdot A_{12}\]
\[L_{21} \leftarrow A_{21} \cdot U_{11}^{-1}\]
\[A_{22} - L_{21} \cdot U_{12} = L_{22} \cdot U_{22}\]
Assume \( L_{11}^{-1}, U_{11}^{-1} \) exist and are available

Gaussian elimination

Block generic Gaussian elimination (contd.)
Now need to produce \( L^{-1}, U^{-1} \) as extra outputs
\[
L^{-1} \leftarrow \begin{bmatrix}
L_{11}^{-1} & \bullet \\
-L_{22}^{-1} \cdot L_{21} \cdot L_{11}^{-1} & L_{22}^{-1}
\end{bmatrix}
\]
\[
U^{-1} \leftarrow \begin{bmatrix}
U_{11}^{-1} & -U_{11}^{-1} \cdot U_{21} \cdot U_{22}^{-1} \\
\bullet & U_{22}^{-1}
\end{bmatrix}
\]
Sequential work \( O(n^3) \), allows use of Strassen-type schemes

Gaussian elimination

Parallel block generic Gaussian elimination
At each level, two recursive calls in a sequence
Recursion tree unfolded depth-first

<table>
<thead>
<tr>
<th>level</th>
<th>tasks</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>( n )</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>( n/2 )</td>
</tr>
<tr>
<td>2</td>
<td>( 2^2 )</td>
<td>( n/2^2 )</td>
</tr>
<tr>
<td>( \cdots )</td>
<td>( p^\alpha )</td>
<td>( n/p^\alpha ) (for any ( \alpha ))</td>
</tr>
<tr>
<td>( \cdots )</td>
<td>( \log n )</td>
<td>1</td>
</tr>
</tbody>
</table>
Gaussian elimination

Parallel block generic Gaussian elimination (contd.)

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

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

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

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

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

Part VI

Graph algorithms

Gaussian elimination

Parallel LU decomposition (contd.)

In particular:

- $\alpha = 1/2$
  \[ \text{comp} O(n^3/p) \quad \text{comm} O(n^2/p^{1/2}) \quad \text{sync} O(p^{1/2}) \]
  Cf. 2D grid
- $\alpha = 2/3$
  \[ \text{comp} O(n^3/p) \quad \text{comm} O(n^2/p^{2/3}) \quad \text{sync} O(p^{2/3}) \]
  Cf. matrix multiplication

Graph algorithms

- Algebraic path problem
- All-pairs shortest paths
Semiring: a set $S$ with addition $\oplus$ and multiplication $\odot$

- $\oplus$ commutative, associative, identity $0$
  
  \[
  a \oplus b = b \oplus a \\
  a \oplus (b \oplus c) = (a \oplus b) \oplus c \\
  a \oplus 0 = 0 \oplus a = a
  \]

- $\odot$ associative, annihilator $0$, identity $1$
  
  \[
  a \odot (b \odot c) = (a \odot b) \odot c \\
  a \odot 0 = 0 \odot a = 0 \\
  a \odot 1 = 1 \odot a = a
  \]

- $\odot$ distributes over $\oplus$
  
  \[
  a \odot (b \oplus c) = a \odot b \oplus a \odot c \\
  (a \oplus b) \odot c = a \odot c \oplus b \odot c
  \]

- In general, no $\ominus$ or $\oslash$!

Some particular semirings:

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

Given a semiring $S$, the set of all square matrices of size $n$ over $S$ is itself a semiring, with

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

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

A semiring is closed, if

- an infinite sum $a_1 \oplus a_2 \oplus a_3 \oplus \cdots$ is always defined
- infinite sums are commutative, associative and distributive

In particular:

- a numerical infinite sum is often undefined (the series diverges)
- a Boolean infinite sum can be defined as 1 if at least one $a_i = 1$, otherwise 0
- a tropical infinite sum can be defined as the terms’ greatest lower bound $\inf \{a_i\} = \max \{l \mid \forall i : l \leq a_i\}$ ($= \min \{a_i\}$ if it exists)

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

- the numerical semiring is not closed
- the Boolean and tropical semirings are closed

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

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

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

In particular:
- numerical $A^* = I + A + A^2 + \cdots = (I - A)^{-1}$, if the sum is defined

The algebraic path problem in a closed semiring can be interpreted via a weighted directed graph on $n$ nodes, defined by adjacency matrix $A$

$A[i,j]$: length of the edge $i \rightarrow j$

In particular:
- Boolean $A^*$ corresponds to the graph’s transitive closure
- tropical $A^*$ corresponds to the graph’s all-pairs shortest paths

### Block Floyd–Warshall algorithm

Partition $A$ and $A^*$ into regular half-sized blocks

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad A^* = \begin{bmatrix} \bar{A}_{11} & \bar{A}_{12} \\ \bar{A}_{21} & \bar{A}_{22} \end{bmatrix}$$

$$\bar{A}_{11} \leftarrow A_{11}^* \quad \bar{A}_{22} \leftarrow A_{22}^*$$

$$\bar{A}_{12} \leftarrow \bar{A}_{11}A_{12} \quad \bar{A}_{21} \leftarrow \bar{A}_{12}A_{21}$$

$$\bar{A}_{21} \leftarrow A_{21}\bar{A}_{11} \quad \bar{A}_{12} \leftarrow A_{12}\bar{A}_{22}$$

$$\bar{A}_{22} \leftarrow A_{22} \oplus A_{21}\bar{A}_{11}A_{12} \quad \bar{A}_{11} \leftarrow \bar{A}_{11} \oplus A_{21}\bar{A}_{22}A_{12}$$

Block generic Gaussian elimination in disguise

Sequential work $O(n^3)$

### Parallel algebraic path computation

Similar to LU decomposition by block generic Gaussian elimination

- the 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 performs the task sequentially

Works even when some lengths are negative (assuming $A^*$ still exists)

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

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

Parallel algebraic path computation (contd.)

In particular:

\[ \alpha = \frac{1}{2} \]

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

Cf. 2D grid

\[ \alpha = \frac{2}{3} \]

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

Cf. matrix multiplication

All-pairs shortest paths

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

| \( S \) | \( \mathbb{R}_{\geq 0} \cup \{+\infty\} \) | \( + \) | \( 0 \) | \( \odot \) | \( 1 \) |
|---|---|---|---|
| tropical | \( \min \) | \( +\infty \) | \( + \) | \( 0 \) |

We continue to use the generic notation \( \oplus, \odot \) (to avoid confusion over matrix \( \odot \))

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

All-pairs shortest paths

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

\[ A[i, j] : \text{length of the edge } i \rightarrow j \]

\[ A[i, j] \geq 0 \quad A[i, i] = \odot = 0 \quad 0 \leq i, j < n \]

The length of a path is the semiring product of all its edge lengths

The size of a path is its total number of edges (by definition, at most \( n \))

\[ A^k[i, j] : \text{length of the shortest path } i \rightsquigarrow j \text{ of size } \leq k \]

\[ A^n[i, j] : \text{length of the shortest path } i \rightsquigarrow j \text{ (of any size)} \]

The all-pairs shortest paths problem:

\[ A^n = I \oplus A \oplus A^2 \oplus \cdots \]

\[ = I \oplus A \oplus A^2 \oplus \cdots \oplus A^n \]

\[ = (I \oplus A)^n = A^n \]

Dijkstra’s algorithm [Dijkstra, 1959]

Computes single-source shortest paths by the greedy method

Fix node \( u \) as the source. Compute all shortest paths originating at \( u \) in order of increasing length: to nearest node, to second nearest node, etc.

Every time a new shortest path \( u \rightsquigarrow v \) is obtained:

- consider all outgoing edges \( v \rightarrow w \)
- if current shortest path \( u \rightsquigarrow w \) is longer than path \( u \rightsquigarrow v \rightarrow w \), then assign the latter to be the new shortest path \( u \rightsquigarrow w \)

Sequential work \( O(n^2) \) to compute single-source shortest paths

All-pairs shortest paths can be computed by running Dijkstra’s algorithm independently from every node as the source

Sequential work \( O(n^3) \)
All-pairs shortest paths

Parallel all-pairs shortest paths

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

For Dijkstra’s algorithm, it is essential that the edge lengths are nonnegative

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

Path doubling

Compute $A, A^2, A^4 = (A^2)^2, A^8 = (A^4)^2, \ldots, A^n = A^* \text{ by log } n \text{ rounds of tropical matrix multiplication}$

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

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 $A^{\leq k}[i,j]$: length of the shortest path $i \leadsto j$ of size $\leq k$

Let $A^{= k}[i,j]$: length of the shortest path $i \leadsto j$ of size exactly $k$

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

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

Parallel all-pairs shortest paths (contd.)

In the above algorithm, use of multiple Dijkstra requires that all edge lengths in A are nonnegative. We now remove this restriction.

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

All processors compute A, A≤3+···, (≤3)2+···, ..., A≤n = A* by at most 2 log p rounds of parallel sparse-by-dense tropical matrix multiplication


Consider A=0, ..., A= p and A= (p) = A= (p) + A=2p + ... + A= p. The total number of non- elements in these matrices is at most n2, on average n2/p per matrix. Hence, for some q ≤ p, matrices A=q and A= p−q have together at most 2 n2/p non- elements.


All processors compute (A= p) by sequential selective path doubling

All processors compute (A= p) ⊗ A= p = A* by parallel matrix multiplication

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