

Efficient Parallel Algorithms

Dr. Alexander Tiskin

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

Efficient Parallel Algorithms

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/](http://forums.warwick.ac.uk/departments/computer-science/ugyear3/cs329/)

Everybody welcome to participate!

Literature

No core textbook — lecture handouts are provided

Reading list:

Bisseling. *Parallel Scientific Computation*. 2004.

Jájá. *An Introduction to Parallel Algorithms*. 1992.

Gibbons, Rytter. *Efficient Parallel Algorithms*. 1989.

For general background in algorithms:

Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms*. 2001.

Literature

Further reading:

McColl. *Scalable Parallel Programming (Lecture Notes)*. 1997.

Blelloch, Maggs. *Parallel Algorithms (Survey)*. 1998.

Gibbons, Spirakis (eds). *Lectures on Parallel Computation*. 1993.

Leopold. *Parallel and Distributed Computing*. 2001.

Course outline (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

Course outline (III): Basic parallel algorithms

- 12 Broadcast/combine
- 13 Balanced tree and prefix sums
- 14 Fast Fourier Transform and the butterfly dag
- 15 Ordered grid
- 16 Discussion

Course outline (II): Parallel computation models

- 7 The PRAM model
- 8 The BSP model
- 9 Network topologies
- 10 Oblivious routing
- 11 Randomised routing

Course outline (IV): Further parallel algorithms

- 17 List contraction and colouring
- 18 Sorting and convex hull

- 19 Matrix-vector and matrix-matrix multiplication
- 20 Triangular system solution
- 21 Gaussian elimination

Part I

Computation by circuits

- 22 Algebraic path problem
- 23 All-pairs shortest paths

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

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): $T(n) = \max_{\text{input size}=n} \text{computation steps}$

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

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

$g = o(f)$: "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 other words: even if we scale f down by any (even very small) constant, it will still eventually overtake and stay above g

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

$g = \Omega(f), \omega(f)$: "g grows (at the same rate or) faster than f"

$g = \Theta(f)$: "g grows at the same rate as f"

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

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:

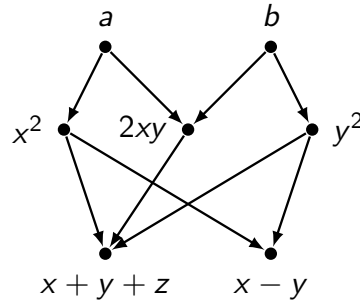
- *impossible* in a standard (or even quantum) model
- can be added to the standard model as an *oracle*, to create a more powerful model

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 circuit model

Bounded or unbounded fan-in/fan-out

Elementary operations:

- arithmetic/Boolean/comparison
- each (usually) constant time

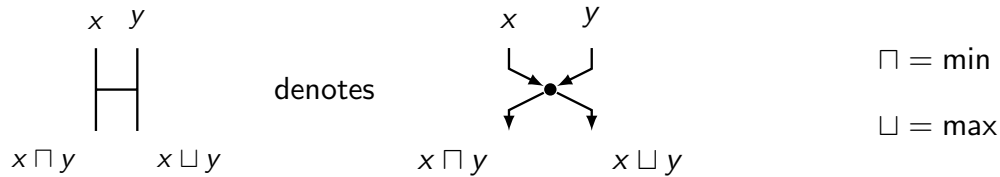
size = number of nodes

depth = max path length from input to output

Timed circuits with feedback: *systolic arrays*

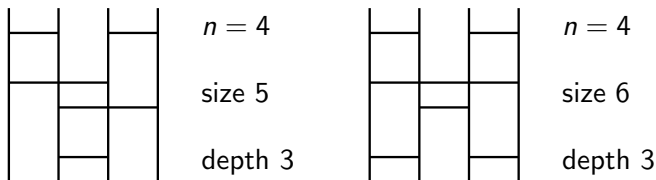
The comparison network model

A *comparison network* is a circuit of *comparator nodes*



The input and output sequences have the same length

Examples:



The comparison network model

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

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

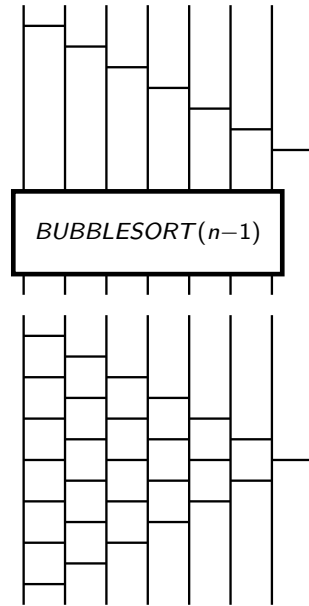
A sorting (or merging) network is equivalent to an oblivious sorting (or merging) algorithm; the network's size/depth determine the algorithm's sequential/parallel complexity

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

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

What is the complexity of oblivious sorting?

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



$BUBBLESORT(8)$
 size 28
 depth 13

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 $x = \langle x_1, \dots, x_n \rangle$

$$\langle x_1, \dots, x_n \rangle \mapsto \langle y_1, \dots, y_n \rangle \quad \exists k, l : k < l : y_k > y_l$$

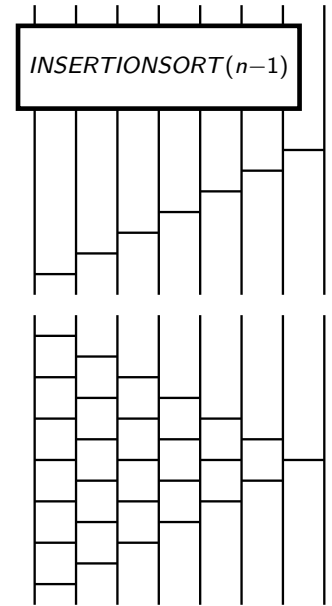
Let $X_i = \begin{cases} 0 & \text{if } x_i < y_k \\ 1 & \text{if } x_i \geq y_k \end{cases}$, and run the network on input $X = \langle X_1, \dots, X_n \rangle$

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

$$\langle X_1, \dots, X_n \rangle \mapsto \langle Y_1, \dots, Y_n \rangle \quad Y_k = 1 > 0 = Y_l$$

We have $k < l$ but $Y_k > Y_l$, so the network does not sort 0s and 1s □

$INSERTIONSORT(n)$
 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

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 $\binom{n}{n'} = \binom{n}{n''}$

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

How fast can we merge obliviously?

$$\langle x_1 \leq \dots \leq x_{n'} \rangle, \langle y_1 \leq \dots \leq y_{n''} \rangle \mapsto \langle z_1 \leq \dots \leq z_n \rangle$$

Odd-even merging

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

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

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

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

$OEM(n', n'')$

size $O(n \log n)$

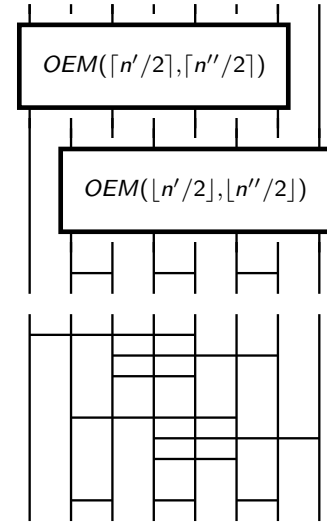
depth $O(\log n)$

$n' \leq n''$

$OEM(4, 4)$

size 9

depth 3



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

Induction base: trivial (2 inputs, 1 comparator)

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

$$\langle 0^{\lceil k/2 \rceil} 11 \dots \rangle, \langle 0^{\lceil l/2 \rceil} 11 \dots \rangle \mapsto \langle 0^{\lceil k/2 \rceil + \lceil l/2 \rceil} 11 \dots \rangle$$

$$\langle 0^{\lfloor k/2 \rfloor} 11 \dots \rangle, \langle 0^{\lfloor l/2 \rfloor} 11 \dots \rangle \mapsto \langle 0^{\lfloor k/2 \rfloor + \lfloor l/2 \rfloor} 11 \dots \rangle$$

$$\text{We need } \langle 0^k 11 \dots \rangle, \langle 0^l 11 \dots \rangle \mapsto \langle 0^{k+l} 11 \dots \rangle$$

$$(\lceil k/2 \rceil + \lceil l/2 \rceil) - (\lfloor k/2 \rfloor + \lfloor l/2 \rfloor) =$$

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

The final stage of comparators corrects the wrong pair □

Sorting an arbitrary input $\langle x_1, \dots, x_n \rangle$

Odd-even merge sorting

[Batcher: 1968]

When $n = 1$ we are done, otherwise

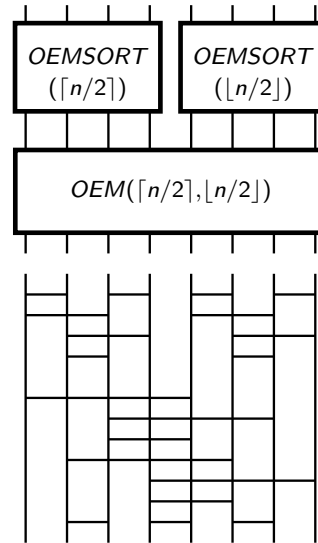
- sort $\langle x_1, \dots, x_{\lceil n/2 \rceil} \rangle$ recursively
- sort $\langle x_{\lceil n/2 \rceil + 1}, \dots, x_n \rangle$ recursively
- merge results by $OEM(\lceil n/2 \rceil, \lfloor n/2 \rfloor)$

$$size_{OEMSORT}(n) \leq 2 \cdot size_{OEMSORT}(n/2) + size_{OEM}(n) = 2 \cdot size_{OEMSORT}(n/2) + O(n \log n) = O(n(\log n)^2)$$

$$depth_{OEMSORT}(n) \leq depth_{OEMSORT}(n/2) + depth_{OEM}(n) = depth_{OEMSORT}(n/2) + O(\log n) = O((\log n)^2)$$

$OEMSORT(n)$
 size $O(n(\log n)^2)$
 depth $O((\log n)^2)$

$OEMSORT(8)$
 size 19
 depth 5



A bitonic sequence: $\langle x_1 \geq \dots \geq x_m \leq \dots \leq x_n \rangle$

Bitonic merging: sorting a bitonic sequence

When $n = 1$ we are done, otherwise

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

Correctness proof: by zero-one principle (exercise)

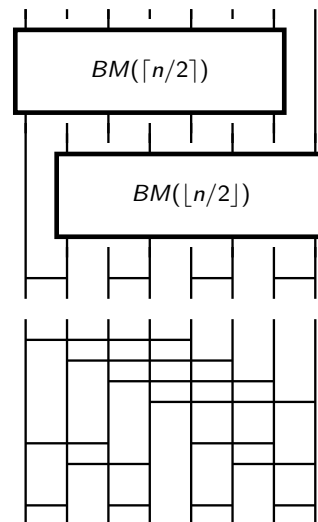
(Note: cannot exchange \geq and \leq in definition of bitonic!)

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

$size_{BM}(n) = O(n \log n) \quad depth_{BM}(n) = O(\log n)$

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

$BM(8)$
 size 12
 depth 3



Bitonic sorting

[Batcher, 1968]

When $n = 1$ we are done, otherwise

- sort $\langle x_1, \dots, x_{\lceil n/2 \rceil} \rangle \mapsto \langle y_1 \geq \dots \geq y_{\lceil n/2 \rceil} \rangle$ in reverse, recursively
- sort $\langle x_{\lceil n/2 \rceil + 1}, \dots, x_n \rangle \mapsto \langle y_{\lceil n/2 \rceil + 1} \leq \dots \leq y_n \rangle$ recursively
- sort bitonic $\langle y_1 \geq \dots \geq y_m \leq \dots \leq y_n \rangle \quad m = \lceil n/2 \rceil \text{ or } \lceil n/2 \rceil + 1$

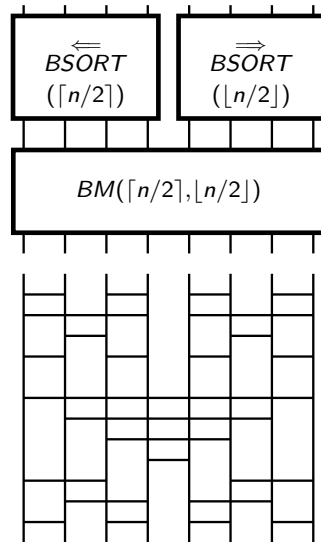
Sorting in reverse seems to require “inverted comparators”, however

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

$size_{BSORT}(n) = O(n(\log n)^2) \quad depth_{BSORT}(n) = O((\log n)^2)$

$BSORT(n)$
 size $O(n(\log n)^2)$
 depth $O((\log n)^2)$

$BSORT(8)$
 size 24
 depth 6



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 [Ajtai, Komlós, Szemerédi: 1983]
 Sorting network: size $O(n \log n)$, depth $O(\log n)$
 Uses sophisticated graph theory (*expanders*)
 Asymptotically optimal, but has huge constant factors

Part II

Parallel computation models

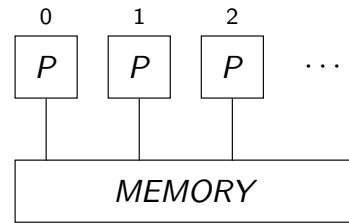
Parallel computation models

- 7 The PRAM model
- 8 The BSP model
- 9 Network topologies
- 10 Oblivious routing
- 11 Randomised routing

The PRAM model

Parallel Random Access Machine (PRAM)

[Fortune, Wyllie, 1978]



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 PRAM model

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

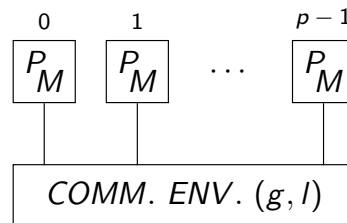
E.g. a linear system solver: $O((\log n)^2)$ steps using n^4 processors

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

The BSP model

Bulk-Synchronous Parallel (BSP) computer

[Valiant, 1990]



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

Some elements of a BSP computer can be emulated by others, e.g.

- external memory by local memory + communication
- barrier synchronisation mechanism by the network

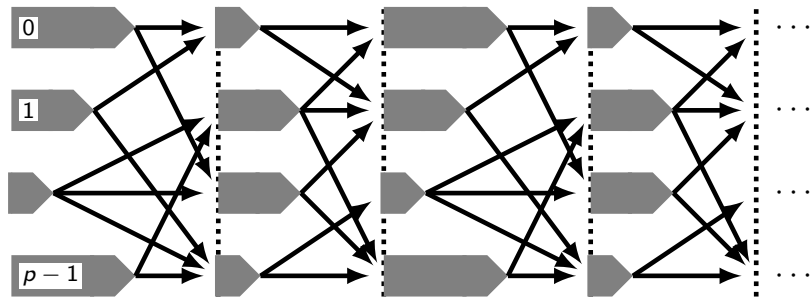
Every parallel computer can be (approximately) described by the parameters p, g, l

E.g. for SGI Origin 2000: $p = 8, g \approx 15, l \approx 4000$

Parameter g corresponds to the network's *communication gap* (inverse bandwidth) — the time for a data unit to enter/exit the network

Parameter l corresponds to the network's *latency* — the worst-case time for a data unit to get across the network

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

For the whole BSP computation with *sync* supersteps:

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

The input/output data are stored in the external memory; the cost of input/output is included in *comm*

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

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

Compositional cost model

For individual processor *proc* in superstep *sstep*:

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

For the whole BSP computer in one superstep *sstep*:

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

BSP computation: scalable, portable, predictable

BSP algorithm design: minimising *comp*, *comm*, *sync*

Main principles:

- load balancing minimises *comp*
- data locality minimises *comm*
- coarse granularity minimises *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

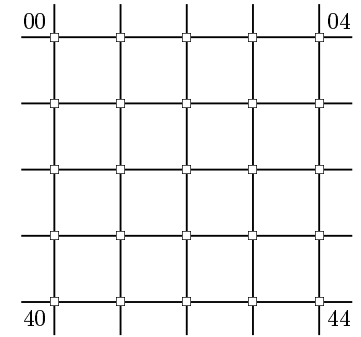
Network topologies

2D array network

$p = q^2$ processors

degree 4

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



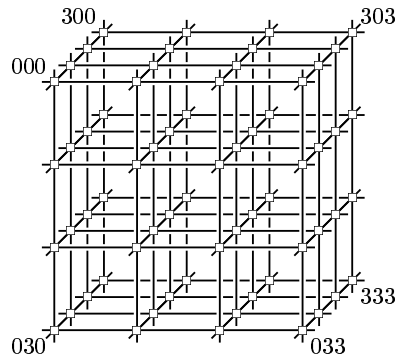
Network topologies

3D array network

$p = q^3$ processors

degree 6

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



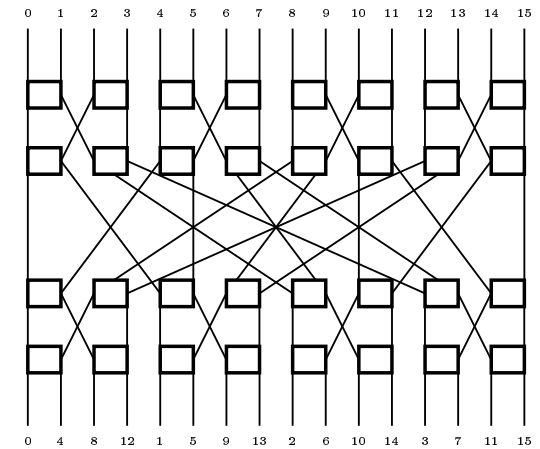
Network topologies

Butterfly network

$p = q \log q$ processors

degree 4

diameter $\approx \log p \approx \log q$

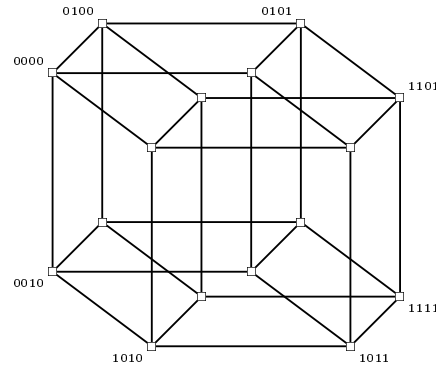


Hypercube network

$p = 2^q$ processors

degree $\log p = q$

diameter $\log p = q$



Summary:

Network	Deg	Diameter
1D array	2	$1/2 \cdot p$
2D array	4	$p^{1/2}$
3D array	6	$3/2 \cdot p^{1/3}$
Butterfly	4	$\log p$
Hypercube	$\log p$	$\log p$
...

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 hk 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

Oblivious routing

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

Network	Degree	Diameter
OEMSORT/BSORT	$O((\log p)^2)$	$O((\log p)^2)$
AKS	$O(\log p)$	$O(\log p)$

No “hot spots”: for $h = 1$, can always route packets in $O(\text{diameter})$ steps

Not practical due to messy wiring

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

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

Network	g	l
1D array	$O(p)$	$O(p)$
2D array	$O(p^{1/2})$	$O(p^{1/2})$
3D array	$O(p^{1/3})$	$O(p^{1/3})$
Butterfly	$O(\log p)$	$O(\log p)$
Hypercube	$O(1)$	$O(\log p)$
...

Actual values of g , l obtained by benchmarking

www.bsp-worldwide.org/implmnts/oxtool

Part III

Basic parallel algorithms

Basic parallel algorithms

- 12 Broadcast/combine
- 13 Balanced tree and prefix sums
- 14 Fast Fourier Transform and the butterfly dag
- 15 Ordered grid
- 16 Discussion

The *broadcasting* problem:

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

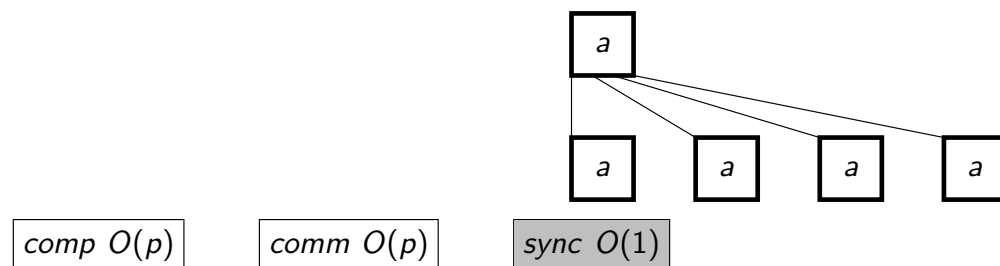
The *combining* problem (complementary to broadcasting):

- initially, every processor holds a value a_i , $0 \leq i < p$
- at the end, one designated processor must hold $a_0 \bullet \dots \bullet a_{p-1}$ for a given associative operator \bullet (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



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

Binary tree broadcast:

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

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

comp $O(\log p)$

comm $O(\log p)$

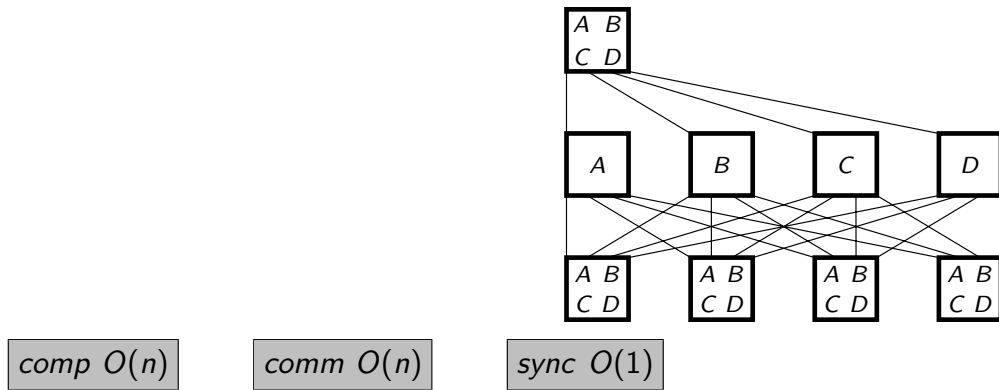
sync $O(\log p)$

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

(effectively, n independent instances of broadcasting/combining)

Two-phase array broadcast:

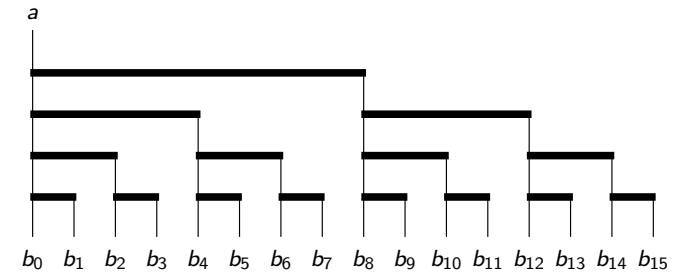
- partition array into p blocks of size n/p
- scatter blocks, then *total-exchange* blocks



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 $\log n$



Sequential work $O(n)$

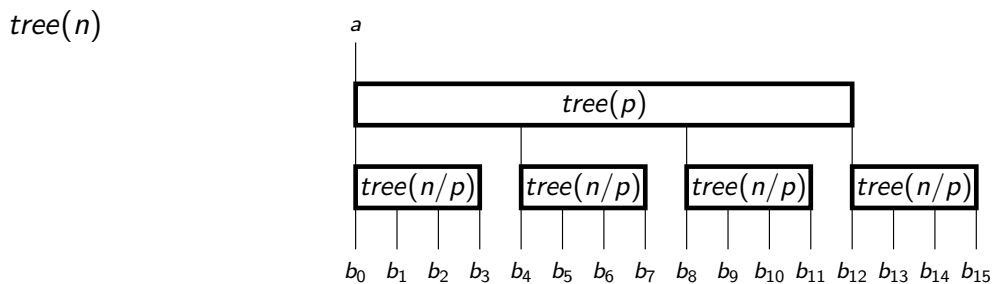
Balanced tree and prefix sums

Parallel balanced tree computation

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

Partition $tree(n)$ into

- one top block, isomorphic to $tree(p)$
- a bottom layer of p blocks, each isomorphic to $tree(n/p)$



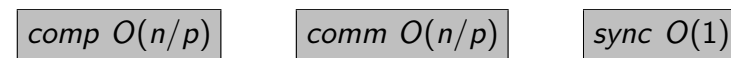
Balanced tree and prefix sums

Parallel balanced tree computation (contd.)

- a designated processor is assigned the top block: it reads the input from external memory, computes the block, and writes the p outputs back to external memory;
- every processor is assigned a bottom block; it reads the input from external memory, computes the block, and writes the n/p outputs back to external memory.

For bottom-up computation, reverse the steps

$n \geq p^2$



The described parallel balanced tree algorithm has

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

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

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

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

E.g. numerical $+$, \cdot , \min ...

The *prefix sums problem*:

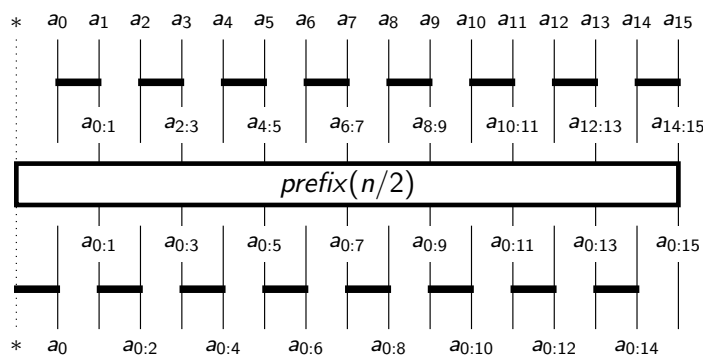
$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} \mapsto \begin{bmatrix} x_0 \\ x_0 \bullet x_1 \\ x_0 \bullet x_1 \bullet x_2 \\ \vdots \\ x_0 \bullet x_1 \bullet \dots \bullet x_{n-1} \end{bmatrix}$$

Sequential work $O(n)$

The *prefix circuit*

[Ladner, Fischer, 1980]

prefix(n)



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

The underlying dag is called the *prefix dag*

The *prefix circuit* (contd.)

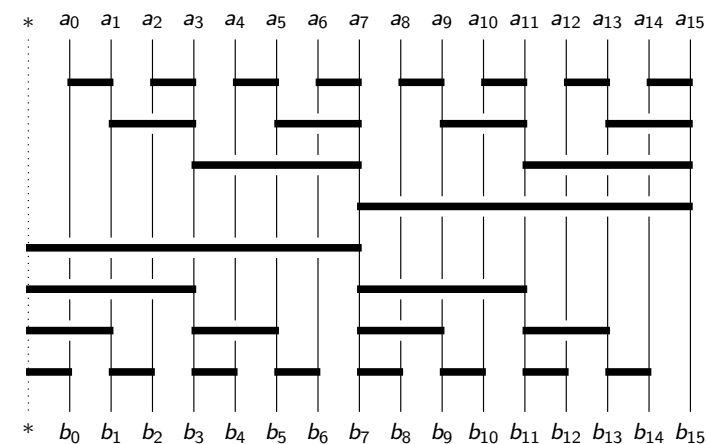
prefix(n)

n inputs

n outputs

size $2n - 2$

depth $2 \log n$



Parallel prefix computation

The dag $prefix(n)$ consists of

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

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

$$n \geq p^2$$

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

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

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

Balanced tree and prefix sums

$$x + y = z$$

Let $c = \langle c_{n-1}, \dots, c_0 \rangle$, 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 $u_i = x_i \wedge y_i \quad v_i = x_i \oplus y_i \quad 0 \leq i < n$

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

$$z_0 = v_0 \qquad c_0 = u_0$$

$$z_1 = v_1 \oplus c_0 \qquad c_1 = u_1 \vee (v_1 \wedge c_0)$$

$$\dots \qquad \dots$$

$$z_{n-1} = v_{n-1} \oplus c_{n-2} \qquad c_{n-1} = u_{n-1} \vee (v_{n-1} \wedge c_{n-2})$$

$$z_n = c_{n-1}$$

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

Application: binary addition via Boolean logic

$$x + y = z$$

Let $x = \langle x_{n-1}, \dots, x_0 \rangle$, $y = \langle y_{n-1}, \dots, y_0 \rangle$, $z = \langle z_n, z_{n-1}, \dots, z_0 \rangle$ be the binary representation of x , y , z

The problem: $\langle x_i \rangle, \langle y_i \rangle \mapsto \langle z_i \rangle$ using \wedge ("and"), \vee ("or"), \oplus ("xor")

Balanced tree and prefix sums

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

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

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

Function composition \circ is associative

$$F_{u', v'} \circ F_{u, v}(c) = F_{u, v}(F_{u', v'}(c)) = u \vee (v \wedge (u' \vee (v' \wedge c))) = u \vee (v \wedge u') \vee (v \wedge v' \wedge c) = F_{u \vee (v \wedge u'), v \wedge v'}(c)$$

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

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

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

Resulting circuit has size $O(n)$ and depth $O(\log n)$

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

Such an ω is called a *primitive root of unity* of degree n , exists for every n in complex numbers

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

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

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

$$b[i] = \sum_j \omega^{ij} a[j] \quad i, j = 0, \dots, n-1$$

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

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

We have $B = \mathcal{F}_{m,\omega^m}(\mathcal{T}_{m,\omega}(\mathcal{F}_{m,\omega^m}(A)))$, 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*

$$size_{FFT}(n) = O(n) + 2 \cdot n^{1/2} \cdot size_{FFT}(n^{1/2}) = O(1 \cdot n \cdot 1 + 2 \cdot n^{1/2} \cdot n^{1/2} + 4 \cdot n^{3/4} \cdot n^{1/4} + \dots + \log n \cdot n \cdot 1) = O(n + 2n + 4n + \dots + \log n \cdot n) = O(n \log n)$$

$$depth_{FFT}(n) = 1 + 2 \cdot depth_{FFT}(n^{1/2}) = O(1 + 2 + 4 + \dots + \log n) = O(\log n)$$

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

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

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

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

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

We have $B = \mathcal{F}_{m,\omega^m}(\mathcal{T}_{m,\omega}(\mathcal{F}_{m,\omega^m}(A)))$, where

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

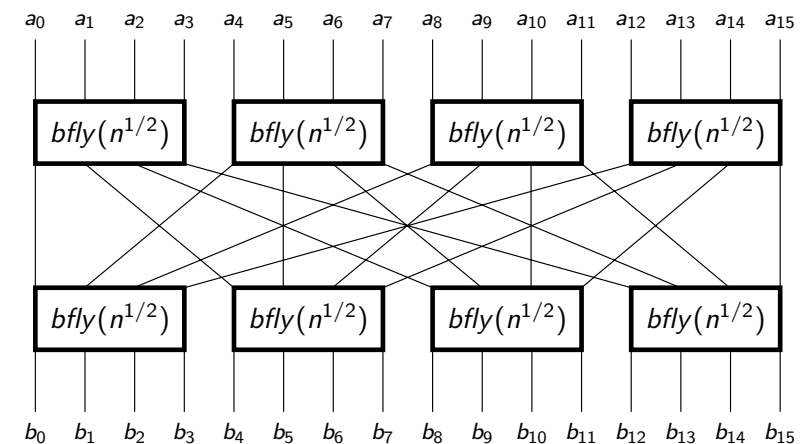
$$\mathcal{F}_{m,\omega^m}(A) = F_{m,\omega^m} \cdot A \quad \mathcal{F}_{m,\omega^m}(A)[t, v] = \sum_u (\omega^m)^{tu} A[u, v]$$

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

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

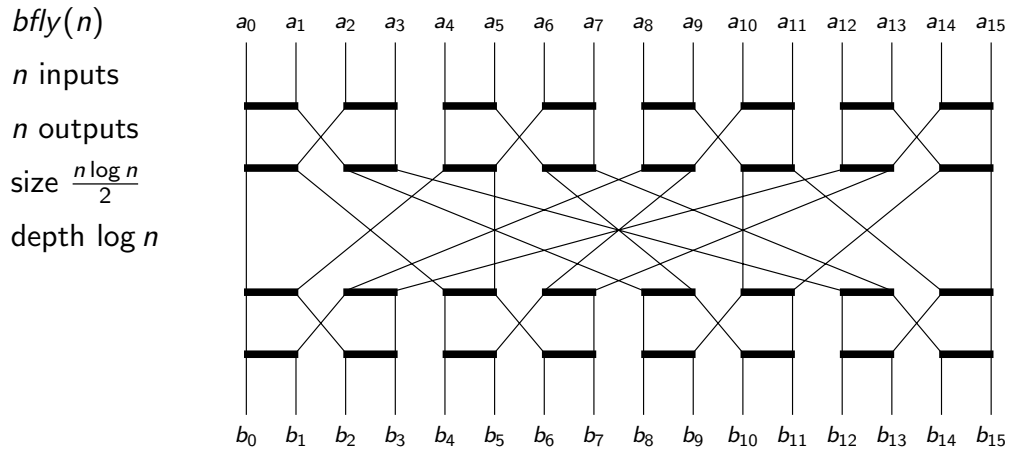
The *FFT circuit*

$bfly(n)$



The underlying dag is called *butterfly dag*

The FFT circuit and the butterfly dag (contd.)



Applications: Fast Fourier Transform; sorting bitonic sequences

Parallel butterfly computation

To compute $bfly(n)$:

- every processor is assigned $n^{1/2}/p$ blocks from the top layer; the processor reads the total of n/p inputs, computes the blocks, and writes back the n/p outputs
- every processor is assigned $n^{1/2}/p$ blocks from the bottom layer; the processor reads the total of n/p inputs, computes the blocks, and writes back the n/p outputs

$$n \geq p^2$$

$$\text{comp } O\left(\frac{n \log n}{p}\right)$$

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

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

The FFT circuit and the butterfly dag (contd.)

Dag $bfly(n)$ consists of

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

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

The ordered 2D grid dag

$grid_2(n)$

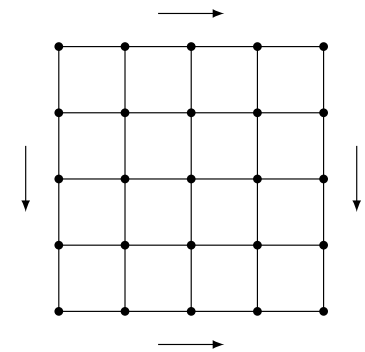
nodes arranged in an $n \times n$ grid

edges directed top-to-bottom, left-to-right

$\leq 2n$ inputs (to left/top borders)

$\leq 2n$ outputs (from right/bottom borders)

size n^2 depth $2n - 1$

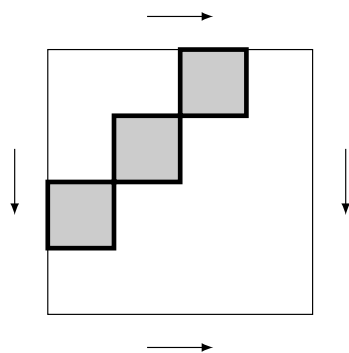


Applications: Gauss–Seidel iteration (single step); triangular system solution; dynamic programming; 1D cellular automata

Sequential work $O(n^2)$

Parallel ordered 2D grid computation

$grid_2(n)$



Consists of p^2 regular blocks, each isomorphic to $grid_2(n/p)$

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

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

$a = \text{"define"}$ $b = \text{"design"}$

$LCS(a, b) = \text{"dein"}$

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

Parallel ordered 2D grid computation (contd.)

The computation proceeds in $2p - 1$ 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/p inputs, computes the block, and writes back the n/p outputs

$comp: (2p - 1) \cdot O((n/p)^2) = O(p \cdot n^2/p^2) = O(n^2/p)$

$comm: (2p - 1) \cdot O(n/p) = O(n)$

$n \geq p$

$comp \ O(n^2/p)$

$comm \ O(n)$

$sync \ O(p)$

LCS computation by *dynamic programming*

Let $lcs(a, b)$ denote the LCS *length*

$lcs(a, \text{""}) = 0$

$lcs(\text{""}, b) = 0$

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

	*	d	e	f	i	n	e
*	0	0	0	0	0	0	0
d	0	1	1	1	1	1	1
e	0	1	2	2	2	2	2
s	0	1	2	2	2	2	2
i	0	1	2	2	3	3	3
g	0	1	2	2	3	3	3
n	0	1	2	2	3	4	4

$lcs(\text{"define"}, \text{"design"}) = 4$

$LCS(a, b)$ 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 approach gives a BSP algorithm for the LCS problem (and many other problems solved by dynamic programming)

$$\text{comp } O(n^2/p)$$

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

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

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]

$$\text{comp } O(n^2/p)$$

$$\text{comm } O\left(\frac{n \log p}{p^{1/2}}\right)$$

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

The ordered 3D grid dag

$grid_3(n)$

nodes arranged in an $n \times n \times n$ grid

edges directed top-to-bottom, left-to-right, front-to-back

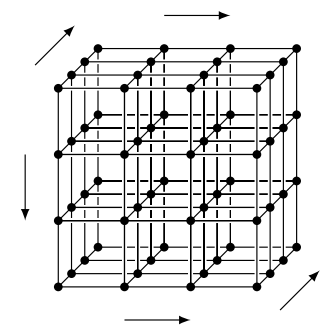
$\leq 3n^2$ inputs (to front/left/top faces)

$\leq 3n^2$ outputs (from back/right/bottom faces)

size n^3 depth $3n - 2$

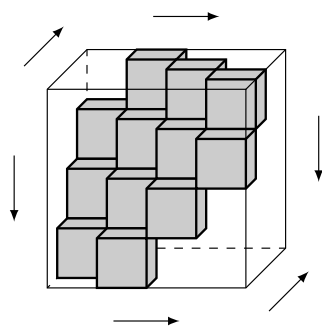
Applications: Gauss-Seidel iteration; Gaussian elimination; dynamic programming; 2D cellular automata

Sequential work $O(n^3)$



Parallel ordered 3D grid computation

$grid_3(n)$



Consists of $p^{3/2}$ regular blocks, each isomorphic to $grid_3(n/p^{1/2})$

The blocks can be arranged into $3p^{1/2} - 1$ layers orthogonal to the main diagonal, with $O(p)$ independent blocks in each layer

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

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

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

$$n \geq p^{1/2}$$

$$\text{comp } O(n^3/p)$$

$$\text{comm } O(n^2/p^{1/2})$$

$$\text{sync } O(p^{1/2})$$

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

The goals:

- $comp = comp_{opt} = comp_{seq}/p$
- $comm$ scales down with increasing p
- $sync$ is a function of p , independent of n

The challenges:

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

Further parallel algorithms

17 List contraction and colouring

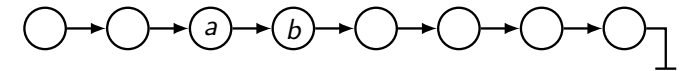
18 Sorting and convex hull

Part IV

Further parallel algorithms

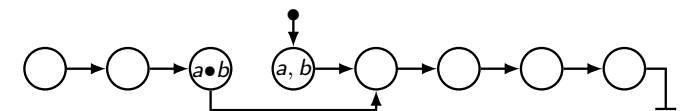
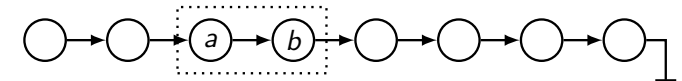
List contraction and colouring

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



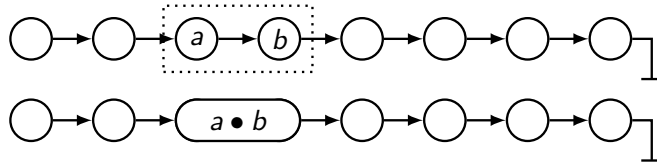
Let \bullet be an associative operator, computable in time $O(1)$

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

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

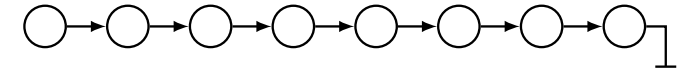


The original a , b are kept implicitly, so that node merging can be reversed

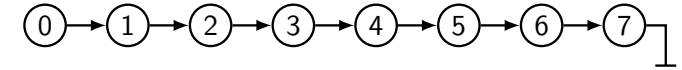
List contraction: reduce the list to a single node by successive merging (note the result is independent on the merging order)

List expansion: restore the original list by reversing the contraction

Application: *list ranking*



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



Note the solution should be independent of the merging order!

Application: *list ranking* (contd.)

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

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

Initially, each node assigned 1

Merging operation: $k, l \rightarrow k + l$

In the fully contracted list, the node contains value n

Application: *list ranking* (contd.)

Expansion phase: for each node keep

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

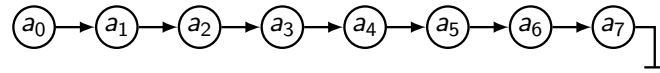
Initially, the node (fully contracted list) assigned $(0, n)$

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

In the fully expanded list, a node with rank i contains $(i, 1)$

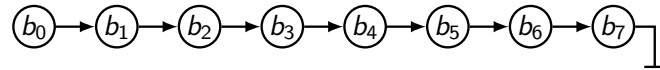
Application: *list prefix sums*

Initially, each node i contains value a_i



Let \bullet be an associative operator with identity ϵ

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



Note the solution should be independent of the merging order!

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}

Application: *list prefix sums (contd.)*

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 (ϵ, 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 $b_i = b_{i-1} \bullet a_i$

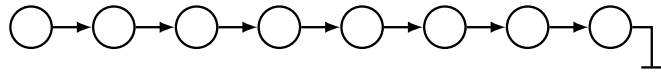
From now on, we only consider pure list contraction (the expansion phase is obtained by symmetry)

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

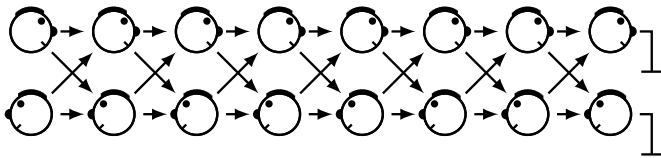
Parallel list contraction must be based on local merging decisions: a node can be merged with either its successor or predecessor, but not with both simultaneously

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

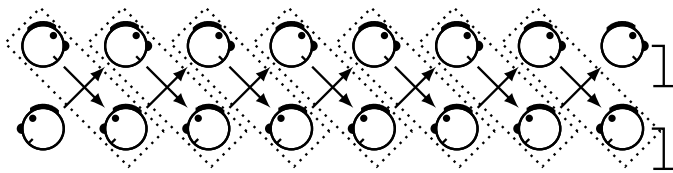
Wyllie's mating [Wyllie, 1979]



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



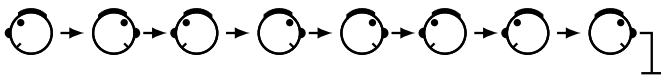
Merge mating pairs, obtaining two lists of size $\approx n/2$



Random mating [Miller, Reif, 1985]

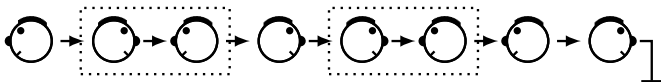
[Miller, Reif, 1985]

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 $O(n)$

$n \geq p$

comp $O(\frac{n \log n}{p})$

comm $O(\frac{n \log n}{p})$

sync $O(\log n)$

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

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

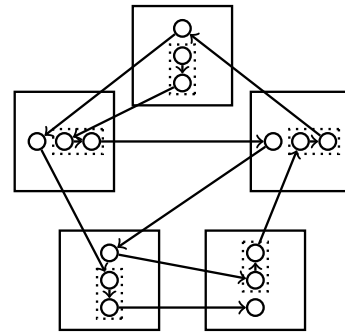
comp $O(n/p)$ whp

comm $O(n/p)$ whp

sync $O(\log p)$

Deterministic block mating

Contract local chains (if any)

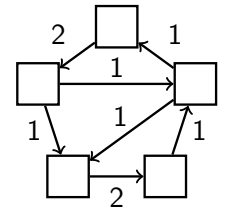


Deterministic block mating (contd.)

Build *distribution graph*:

- complete weighted directed graph on p nodes
- $weight(i, j) = |\{u \rightarrow v : u \in proc_i, v \in proc_j\}|$

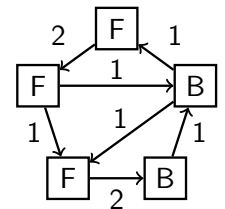
Each processor holds the corresponding node's outgoing edges



Collect distribution graph in a designated processor

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

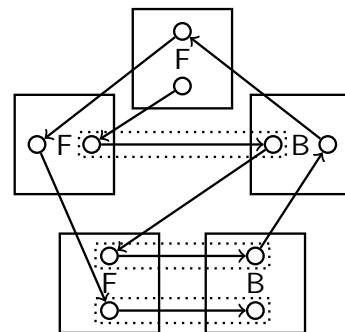
Distribute labels back to processors



Deterministic block mating (contd.)

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

$$n \geq p^3$$

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

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

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

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

Require *comp* $O(n/p)$, *comm* $O(n/p)$

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

p -colouring can be easily done in *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 $colour(v) \neq colour(next(v))$.

If $colour(v)$ differs from $colour(next(v))$ in i -th bit, re-colour v in

- $2i$, if i -th bit is 0 in $colour(v)$ and 1 in $colour(next(v))$
- $2i + 1$, if i -th bit is 1 in $colour(v)$ and 0 in $colour(next(v))$

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

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

$$\log^* k = \min r : \log \dots \log k \leq l$$

(r times)

- select node v as a *pivot*, if $colour(prev(v)) > colour(v) < colour(next(v))$. 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

comp $O(n/p)$

comm $O(n/p)$

sync $O(\log^* p)$

Sorting $\langle x_0, \dots, x_{n-1} \rangle$

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

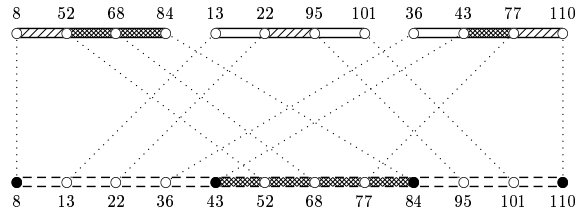
comp $O(\frac{n \log n}{p})$

comm $O(\frac{n \log n}{p})$

sync $O(\log n)$

Parallel sorting by *regular sampling*

[Shi, Schaeffer, 1992]



Every processor

- reads a subarray of size n/p and sorts it sequentially
- selects from its subarray $p + 1$ regular samples

A designated processor

- collects all $p(p + 1)$ samples and sorts them sequentially
- selects from the sorted samples $p + 1$ regular splitters

Parallel sorting by *regular sampling* (contd.)

The designated processor broadcasts the splitters

Every processor

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

$$n \geq p^3$$

$$\text{comp } O\left(\frac{n \log n}{p}\right)$$

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

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

Parallel sorting by *regular sampling* (contd.)

Samples define local *blocks* $[x_i, x_{i+1}]$ of size n/p^2

Splitters define global *buckets* $[[X_k, X_{k+1}]]$

Claim: each bucket has size is at most $3n/p$

Proof. Relative to a bucket $[[X_k, X_{k+1}]]$, a block $[x_i, x_{i+1}]$ can be

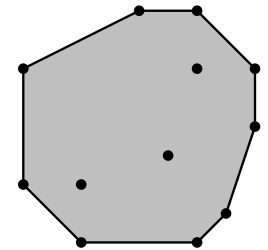
- *outer* if $[x_i, x_{i+1}] \cap [[X_k, X_{k+1}]] = \emptyset$
- *inner* if $[x_i, x_{i+1}] \subseteq [[X_k, X_{k+1}]]$; at most p such blocks overall
- *boundary* otherwise; at most 2 such blocks per processor

Bucket size is at most $p \cdot n/p^2 + 2p \cdot n/p^2 = 3n/p$



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

$$X = \{x_0, \dots, x_{n-1}\} \subseteq \mathbb{R}^d$$



The *convex hull* $\text{conv } X$ is the smallest convex set containing X

$\text{conv } X$ is a *polytope*, defined by its *vertices* $x_i \in X$

Set X is *in convex position*, if every its point is a vertex of $\text{conv } X$

$$X = \{x_0, \dots, x_{n-1}\} \subseteq \mathbb{R}^d$$

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 $d = 2, 3$

$$X = \{x_0, \dots, x_{n-1}\} \subseteq \mathbb{R}^d \quad \text{Let } 0 \leq \epsilon \leq 1$$

Set $A \subseteq X$ is an ϵ -*net* for X , if any halfspace without points in A contains at most $\epsilon|X|$ points in X

Set $A \subseteq X$ is an ϵ -*approximation* for X , if any halfspace with $\alpha|A|$ points in A contains $(\alpha \pm \epsilon)|X|$ points in X

Claim. An ϵ -approximation for X is an ϵ -net for X

Claim. A union of ϵ -approximations for X_1, X_2 is an ϵ -approximation for $X_1 \cup X_2$

Claim. An ϵ -net for a δ -approximation for X is an $(\epsilon + \delta)$ -net for X

Proofs. Easily by definitions. □

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

Proof. Let $x_0, \dots, x_{n-1} \in \mathbb{R}^2$

To sort $\langle x_0, \dots, x_{n-1} \rangle$:

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

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

$$X \subseteq \mathbb{R}^2 \quad |X| = n \quad \epsilon = 1/r$$

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 . □

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

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

Proof. [Brönnimann, Goodrich, 1995] □

Claim. A $1/r$ -approximation for X 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 exist, but are harder to compute)

Parallel convex hull computation by *generalised regular sampling* (contd.)

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

Each splitter edge defines a *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 $2 \cdot (2/p) \cdot n = 4n/p$.

$$X \subseteq \mathbb{R}^3 \text{ (}\mathbb{R}^2 \text{ analogous but simpler)} \quad |X| = n$$

Parallel convex hull computation by *generalised regular sampling*

Every processor

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

The global union of all samples is a $1/p$ -approximation for X

A designated processor

- collects all $O(p^4(\log p)^{O(1)})$ samples
- select from the samples a $1/p$ -net of $O(p)$ points as *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 .

Parallel convex hull computation by *generalised regular sampling* (contd.)

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

$$n \gg p$$

$$\text{comp } O\left(\frac{n \log n}{p}\right)$$

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

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

Part V

Matrix algorithms

19 Matrix-vector and matrix-matrix multiplication

20 Triangular system solution

21 Gaussian elimination

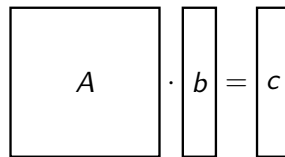
Matrix-vector and matrix-matrix multiplication

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 $O(n^2)$

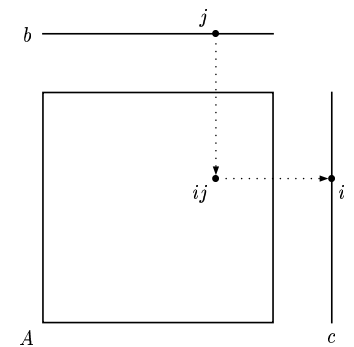
Matrix-vector and matrix-matrix multiplication

The *matrix-vector multiplication circuit*

$$c[i] \leftarrow 0$$

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

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



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 $n/p^{1/2}$

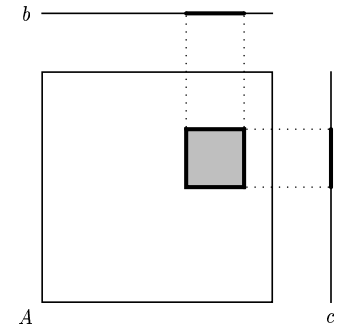
Matrix-vector and matrix-matrix multiplication

Parallel matrix-vector multiplication (contd.)

$$c[[i]] \leftarrow 0$$

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

$$0 \leq i, j < p^{1/2}$$



Matrix-vector and matrix-matrix multiplication

Parallel matrix-vector multiplication (contd.)

Vector c in external memory is initialised by zero values

Every processor

- is assigned to compute a block product $A[[i, j]] \cdot b[[j]] = c^{(i)}[[j]]$
- reads block $b[[j]]$ and computes $c^{(i)}[[j]]$
- updates block $c[[i]]$ in external memory by adding to each element the corresponding element of $c^{(i)}[[j]]$

Individual asynchronous updates to $c[[i]]$ add up to the correct final value

$$n \geq p$$

$$\text{comp } O(n^2/p)$$

$$\text{comm } O(n/p^{1/2})$$

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

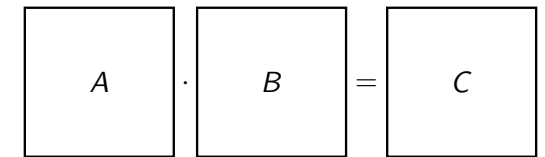
Matrix-vector and matrix-matrix multiplication

Let A, B, C be matrices of size n

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

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

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



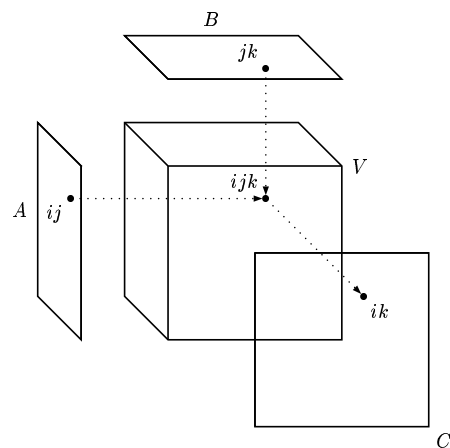
Sequential work $O(n^3)$

The *matrix multiplication circuit*

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

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

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



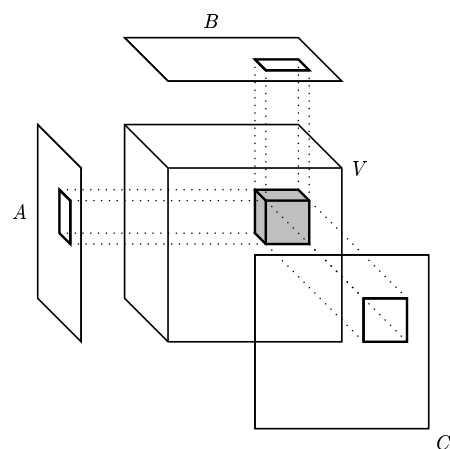
size $O(n^3)$, depth $O(1)$

Parallel matrix multiplication (contd.)

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

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

$$0 \leq i, j, k < p^{1/3}$$



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

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

$$n \geq p^{1/2}$$

$$\text{comp } O(n^3/p)$$

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

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

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

Proof: *comp* $O(n^3/p)$, *sync* $O(1)$ trivially optimal; optimality of *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 $\text{comm} = \Omega(n^2/p^{2/3})$ □

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 optimality theorem only applies to matrix multiplication by the specific $O(n^3)$ -node dag

Corresponds e.g. to

- generic numerical matrix multiplication (primitives $+, \cdot$)
- Boolean matrix multiplication (primitives \vee, \wedge)

Excludes e.g.

- Strassen-type numerical matrix multiplication (primitives $+, -, \cdot$)
- Boolean matrix multiplication (primitives $\vee, \wedge, \text{if/then}$)

Matrices of size $n \geq N$ are partitioned into an $N \times N$ grid of regular blocks, and multiplied recursively:

- some linear operations on the *blocks* of A
- some linear operations on the *blocks* of B
- by recursion, R block products of the resulting linear combinations
- some more linear operations to obtain the blocks of C

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

Let $\omega = \log_N R < \log_N N^3 = 3$

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

Sequential work $O(n^\omega)$

Some specific instances of Strassen-type scheme:

N	N^3	R	$\omega = \log_N R$	
2	8	7	2.81	[Strassen, 1969]
3	27	23	2.85	
5	125	100	2.86	
48	110592	47216	2.78	

Best known $\omega \approx 2.38$, with astronomical N, R
 [Coppersmith, Winograd, 1987]

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*

level	tasks	size
0	1	n
1	R	n/N
2	R^2	n/N^2
...		
$\log_R p$	p	$\frac{n}{N^{\log_R p}} = \frac{n}{p^{\log_R N}} = \frac{n}{p^{1/\omega}}$
...		
$\log_N n$	$R^{\log_N n} = n^{\log_N R} = n^\omega$	1

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

Parallel Strassen-type matrix multiplication (contd.)

Each processor reads its assigned elements of A, B

Recursion levels 0 to $\log_R p$ on the way down: *comm*-free linear operations on blocks of A, B

Recursion level $\log_R p$: we have p independent block multiplication tasks

- assign each task to a different processor
- a processor collects its task's two input blocks, performs the task sequentially, and redistributes the task's output block

Recursion levels $\log_R p$ to 0 on the way up: *comm*-free linear operations on blocks of C

Each processor writes back its assigned elements of C

$comp\ O(n^\omega/p)$

$comm\ O(\frac{n^2}{p^{2/\omega}}): opt?$

$sync\ O(1)$

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

Boolean matrix multiplication: $A \wedge B = C$

Primitives $\vee, \wedge, \text{if/then}$ on matrix elements

$$C[i, k] = \bigvee_j A[i, j] \wedge B[j, k]$$

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

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 ϵ 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.)

$$A \wedge B = C$$

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* $C = C_1 \vee C_2 \vee C_3$, such that

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

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

Parallel Boolean matrix multiplication (contd.)

Partition the cube of elementary product nodes $A[i, j] \wedge B[j, k]$ 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 $n/p^{1/3}$

Parallel Boolean matrix multiplication (contd.)

Every processor

- is assigned to compute a “slab” of p^2 block products $A[[i, j]] \wedge B[[j, k]] = C^{(j)}[[i, k]]$ 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 $A \wedge B = C$, 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.

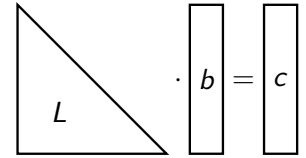
$$\boxed{\text{comp } O(n^\omega \cdot 2^{p^{68}})} \quad \text{: -0} \quad \boxed{\text{comm } O(n^2/p)} \quad \boxed{\text{sync } O(1)}$$

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

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

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

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



Forward substitution

$$L \cdot b = c$$

$$b[0] \leftarrow L[0, 0]^{-1} \cdot c[0]$$

$$b[1] \leftarrow L[1, 1]^{-1} \cdot (c[1] - L[1, 0] \cdot b[0])$$

$$b[2] \leftarrow L[2, 2]^{-1} \cdot (c[2] - L[2, 0] \cdot b[0] - L[2, 1] \cdot b[1])$$

...

$$b[i] \leftarrow L[i, i]^{-1} \cdot (c[i] - \sum_{j:j < i} L[i, j] \cdot b[j])$$

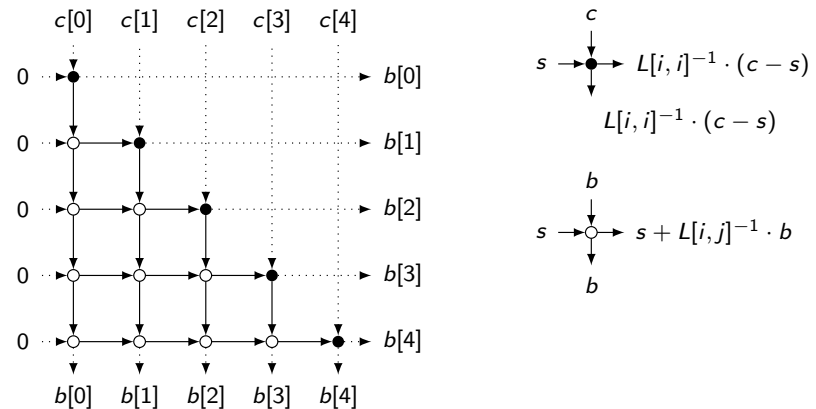
...

$$b[n-1] \leftarrow L[n-1, n-1]^{-1} \cdot (c[n-1] - \sum_{j:j < n-1} L[n-1, j] \cdot b[j])$$

Sequential work $O(n^2)$

Parallel forward substitution by 2D grid dag

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



$$\boxed{\text{comp } O(n^2/p)}$$

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

$$\boxed{\text{sync } O(p)}$$

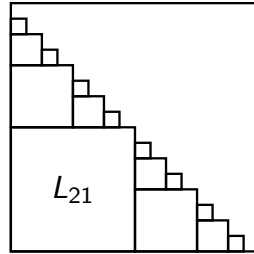
Block forward substitution (divide-and-conquer)

$$L \cdot b = c$$

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

$$L_{11} \cdot b_1 = c_1$$

$$L_{22} \cdot b_2 = c_2 - L_{21} \cdot b_1$$



Sequential work $O(n^2)$

Parallel block forward substitution

Assume L is pre-distributed as needed, does not count as input

At each level, two recursive calls *in a sequence*

Recursion tree unfolded *depth-first*

level	tasks	size
0	1	n
1	2	$n/2$
2	2^2	$n/2^2$
...		
$\log p$	p	n/p
...		
$\log n$	n	1

Parallel block forward substitution (contd.)

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

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

$$\text{comp} = O(n^2/p) \cdot (1 + 2 \cdot (\frac{1}{2})^2 + 2^2 \cdot (\frac{1}{2^2})^2 + \dots) + O((n/p)^2) \cdot p = O(n^2/p) + O(n^2/p) = O(n^2/p)$$

$$\text{comm} = O(n/p^{1/2}) \cdot (1 + 2 \cdot \frac{1}{2} + 2^2 \cdot \frac{1}{2^2} + \dots) + O(n/p) \cdot p = O(n/p^{1/2}) \cdot \log p + O(n) = O(n)$$

$$\text{comp } O(n^2/p)$$

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

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

Let A, L, U be matrices of size n

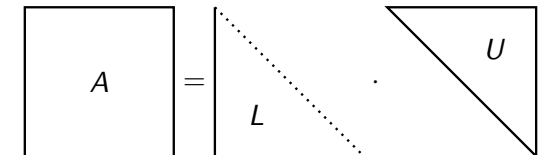
$$L \text{ is unit lower triangular: } L[i, j] = \begin{cases} 0 & 0 \leq i < j < n \\ 1 & 0 \leq i = j < n \\ \text{arbitrary} & \text{otherwise} \end{cases}$$

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

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

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

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



Generic Gaussian elimination

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 1 & \bullet \\ a_{21}/a_{11} & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ \bullet & a_{22} - a_{12} \cdot a_{21}/a_{11} \end{bmatrix}$$

Assume no pivoting required: $a_{11} \neq 0$

Sequential work $O(n^3)$

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

`comp` $O(n^3/p)$

`comm` $O(n^2/p^{1/2})$

`sync` $O(p^{1/2})$

Block generic Gaussian elimination (divide-and-conquer)

$$A = L \cdot U$$

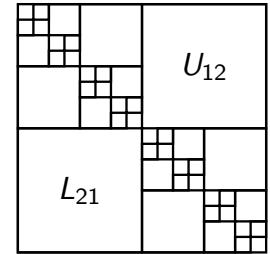
$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \bullet \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ \bullet & U_{22} \end{bmatrix}$$

$$A_{11} = L_{11} \cdot U_{11}$$

$$U_{12} \leftarrow L_{11}^{-1} \cdot A_{12}$$

$$L_{21} \leftarrow A_{21} \cdot U_{11}^{-1}$$

$$A_{22} - L_{21} \cdot U_{12} = L_{22} \cdot U_{22}$$



Assume L_{11}^{-1} , U_{11}^{-1} exist and are available

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

Parallel block generic Gaussian elimination

At each level, two recursive calls *in a sequence*

Recursion tree unfolded *depth-first*

level	tasks	size
0	1	n
1	2	$n/2$
2	2^2	$n/2^2$
...		
$\alpha \log p$	p^α	n/p^α (for any α)
...		
$\log n$	n	1

Parallel block generic Gaussian elimination (contd.)

Recursion levels 0 to $\alpha \log p$ (α 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 α : $1/2 \leq \alpha \leq 2/3$

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

$$\text{comp } O(n^3/p)$$

$$\text{comm } O(n^2/p^\alpha)$$

$$\text{sync } O(p^\alpha)$$

Part VI

Graph algorithms

Parallel LU decomposition (contd.)

In particular:

$$\alpha = 1/2$$

$$\text{comp } O(n^3/p)$$

$$\text{comm } O(n^2/p^{1/2})$$

$$\text{sync } O(p^{1/2})$$

Cf. 2D grid

$$\alpha = 2/3$$

$$\text{comp } O(n^3/p)$$

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

$$\text{sync } O(p^{2/3})$$

Cf. matrix multiplication

Graph algorithms

22 Algebraic path problem

23 All-pairs shortest paths

Semiring: a set S with addition \oplus and multiplication \odot

- \oplus commutative, associative, identity $\mathbb{0}$
 $a \oplus b = b \oplus a$
 $a \oplus (b \oplus c) = (a \oplus b) \oplus c$
 $a \oplus \mathbb{0} = \mathbb{0} \oplus a = a$
- \odot associative, annihilator $\mathbb{0}$, identity $\mathbb{1}$
 $a \odot (b \odot c) = (a \odot b) \odot c$
 $a \odot \mathbb{0} = \mathbb{0} \odot a = \mathbb{0}$
 $a \odot \mathbb{1} = \mathbb{1} \odot a = a$
- \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 !

The *closure* of a : $a^* = \mathbb{1} \oplus a \oplus a^2 \oplus a^3 \oplus \dots$

In particular:

- numerical $a^* = 1 + a + a^2 + a^3 + \dots = \frac{1}{1-a}$, where $|a| < 1$; otherwise, a^* undefined
- Boolean $a^* = 1 \vee a \vee a^2 \vee a^3 \vee \dots = 1$
- tropical $a^* = \min(0, a, 2a, 3a, \dots) = 0$

A semiring is *closed*, if

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

Some particular semirings:

	S	\oplus	$\mathbb{0}$	\odot	$\mathbb{1}$
numerical	\mathbb{R}	$+$	0	\cdot	1
Boolean	$\{0, 1\}$	\vee	0	\wedge	1
tropical	$\mathbb{R}_{\geq 0} \cup \{+\infty\}$	\min	$+\infty$	$+$	0

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

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

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

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 $A^* = I \oplus A \oplus A^2 \oplus A^3 \oplus \dots$

In particular:

- numerical $A^* = I + A + A^2 + \dots = (I - A)^{-1}$, 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

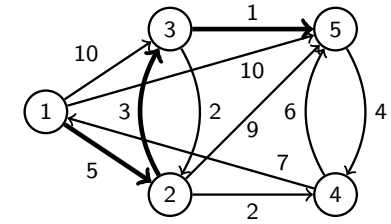
$A[i, j]$: length of the edge $i \rightarrow j$

In particular:

- Boolean A^* corresponds to the graph's *transitive closure*
- tropical A^* corresponds to the graph's *all-pairs shortest paths*

Algebraic path problem

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



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

Algebraic path problem

Block Floyd–Warshall algorithm

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

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

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

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

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

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

Block generic Gaussian elimination in disguise

Sequential work $O(n^3)$

Algebraic path problem

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 α : $1/2 \leq \alpha \leq 2/3$

$$\text{comp } O(n^3/p)$$

$$\text{comm } O(n^2/p^\alpha)$$

$$\text{sync } O(p^\alpha)$$

Parallel algebraic path computation (contd.)

In particular:

$$\alpha = 1/2$$

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

Cf. 2D grid

$$\alpha = 2/3$$

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

Cf. matrix multiplication

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

	S	\oplus	\ominus	\odot	$\mathbb{1}$
tropical	$\mathbb{R}_{\geq 0} \cup \{+\infty\}$	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*: $a \oplus a = \min(a, a) = a$

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

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

$$A[i, j] \geq 0 \quad A[i, i] = \mathbb{1} = 0 \quad 0 \leq i, j < n$$

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)

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

$A^*[i, j]$: length of the shortest path $i \rightsquigarrow j$ (of *any* size)

The *all-pairs shortest paths* problem:

$$\begin{aligned} A^* &= I \oplus A \oplus A^2 \oplus \dots \\ &= I \oplus A \oplus A^2 \oplus \dots \oplus A^n \\ &= (I \oplus A)^n = A^n \end{aligned}$$

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

$comp\ O(n^3/p)$

$comm\ O(n^2)$

$sync\ O(1)$

All-pairs shortest paths

Parallel all-pairs shortest paths: summary so far

$comp\ O(n^3/p)$

Floyd-Warshall, $\alpha = 2/3$

$comm\ O(n^2/p^{2/3})$

$sync\ O(p^{2/3})$

Floyd-Warshall, $\alpha = 1/2$

$comm\ O(n^2/p^{1/2})$

$sync\ O(p^{1/2})$

Multi-Dijkstra

$comm\ O(n^2)$

$sync\ O(1)$

New goal

$comm\ O(n^2/p^{2/3})$

$sync\ O(\log p)$

All-pairs shortest paths

Path doubling

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

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

All-pairs shortest paths

Selective path doubling

Idea: to remove redundancy in path doubling by keeping track of path sizes

Assume we already have A^k . The next round is as follows.

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

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

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

Consider $A^{=\frac{k}{2}}, \dots, A^{=k}$. 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.

Selective path doubling (contd.)

Compute $A, A^{\leq \frac{3}{2}+\dots}, A^{\leq (\frac{3}{2})^2+\dots}, \dots, A^{\leq n} = A^*$ by at most $\log_{3/2} n$ rounds of sparse-by-dense tropical matrix multiplication

Sequential work $2n^3 \left(1 + \left(\frac{3}{2}\right)^{-1} + \left(\frac{3}{2}\right)^{-2} + \dots\right) = O(n^3)$

Parallel all-pairs shortest paths

All processors compute $A, A^{\leq \frac{3}{2}+\dots}, A^{\leq (\frac{3}{2})^2+\dots}, \dots, A^{\leq p+\dots}$ by at most $\log_{3/2} p$ rounds of parallel sparse-by-dense tropical matrix multiplication

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

Every processor reads $A=q$ and $A=p-q$ and computes $A=q \odot A=p-q = A=p$

All processors compute $(A=p)^*$ by parallel multiple Dijkstra

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

comp $O(n^3/p)$

comm $O(n^2/p^{2/3})$

sync $O(\log p)$

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 $A, A^{\leq \frac{3}{2}+\dots}, A^{\leq (\frac{3}{2})^2+\dots}, \dots, A^{\leq p^2+\dots}$ by at most $2 \log p$ rounds of parallel sparse-by-dense tropical matrix multiplication

Let $A^{(p)} = A=p \oplus A=2p \oplus \dots \oplus A=p^2$

Let $A^{(p)-q} = A=p-q \oplus A=2p-q \oplus \dots \oplus A=p^2-q$

Consider $A=0, \dots, A=\frac{p}{2}$ and $A^{(p)-\frac{p}{2}}, \dots, A^{(p)}$. The total number of non-⊙ elements in these matrices is at most n^2 , on average $\frac{n^2}{p}$ per matrix.

Hence, for some $q \leq \frac{p}{2}$, matrices $A=q$ and $A^{(p)-q}$ have together at most $\frac{2n^2}{p}$ non-⊙ elements.

Parallel all-pairs shortest paths (contd.)

Every processor

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

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

Works even when some lengths are negative (assuming A^* still exists, i.e. no negative-length cycles)

comp $O(n^3/p)$

comm $O(n^2/p^{2/3})$

sync $O(\log p)$