Efficient Parallel Algorithms

Alexander Tiskin

Department of Computer Science
University of Warwick
`http://go.warwick.ac.uk/alextiskin`

Computation by circuits

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 $\to$ (computation steps) $\to$ output

Input/output encoding must be specified

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

Computation by circuits
Computation models and algorithms

Algorithm complexity depends on the model

E.g. sorting n items:

- $\Omega(n \log n)$ in the comparison model
- $O(n)$ in the arithmetic model (by radix sort)

E.g. factoring large numbers:

- hard in a von Neumann-type (standard) model
- not so hard on a quantum computer

E.g. deciding if a program halts on a given input:

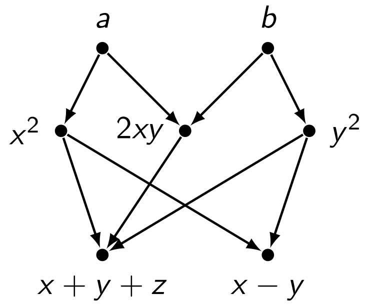
- impossible in a standard (or even quantum) model
- can be added to the standard model as an oracle, to create a more powerful model

Computation by circuits
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

Computation by circuits
The circuit model

Bounded or unbounded fan-in/fan-out

Elementary operations:

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

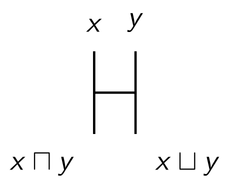
size = number of nodes

depth = max path length from input to output

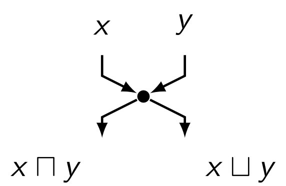
Timed circuits with feedback: systolic arrays

Computation by circuits
The comparison network model

A comparison network is a circuit of comparator nodes



denotes

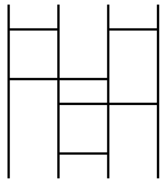


$\sqcap = \min$

$\sqcup = \max$

The input and output sequences have the same length

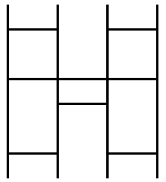
Examples:



$n = 4$

size 5

depth 3



$n = 4$

size 6

depth 3

Computation by circuits

The comparison network model

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

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

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

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

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

What is the complexity of oblivious sorting?

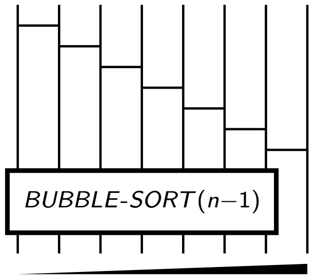
Computation by circuits

Naive sorting networks

BUBBLE-SORT(n)

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

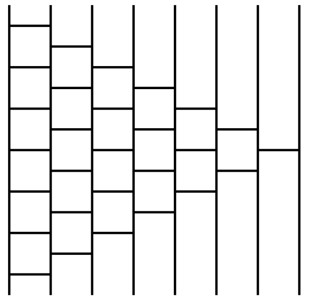
depth $2n - 3 = O(n)$



BUBBLE-SORT(8)

size 28

depth 13



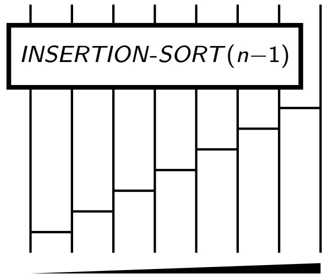
Computation by circuits

Naive sorting networks

INSERTION-SORT(n)

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

depth $2n - 3 = O(n)$

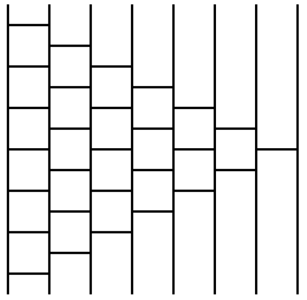


INSERTION-SORT(8)

size 28

depth 13

Identical to *BUBBLE-SORT*!



Computation by circuits

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

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

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 $X = \langle X_1, \ldots, X_n \rangle$

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

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

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

Computation by circuits
The zero-one principle

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

It allows one to test:

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

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

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

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

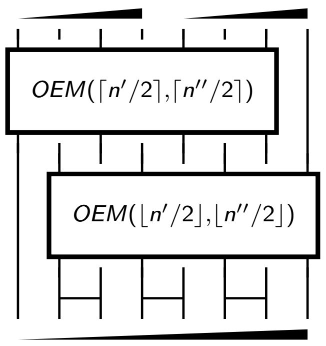
Computation by circuits
Efficient merging and sorting networks

$OEM(n', n'')$

size $O(n \log n)$

depth $O(\log n)$

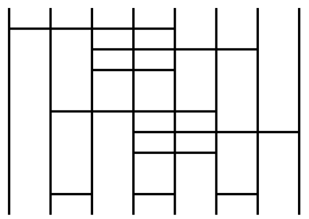
$n' \leq n''$

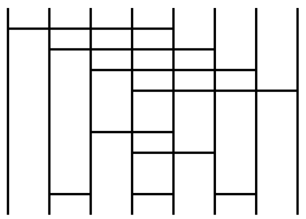

$OEM(\lceil n'/2 \rceil, \lceil n''/2 \rceil)$
$OEM(\lfloor n'/2 \rfloor, \lfloor n''/2 \rfloor)$

$OEM(4, 4)$

size 9

depth 3





Computation by circuits
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:

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

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

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

$$(\lceil k/2 \rceil + \lceil l/2 \rceil) - (\lfloor k/2 \rfloor + \lfloor l/2 \rfloor) = \begin{cases} 0, 1 & \text{result sorted: } \langle 0^{k+l} 11 \ldots \rangle \\ 2 & \text{single pair wrong: } \langle 0^{k+l-1} 1011 \ldots \rangle \end{cases}$$

The final stage of comparators corrects the wrong pair □

Computation by circuits
Efficient merging and sorting networks

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 $OEM(\lceil n/2 \rceil, \lfloor n/2 \rfloor)$

$$size_{OEM\text{-}SORT}(n) \le 2 \cdot size_{OEM\text{-}SORT}(n/2) + size_{OEM}(n) = 2 \cdot size_{OEM\text{-}SORT}(n/2) + O(n \log n) = O\big(n(\log n)^2\big)$$

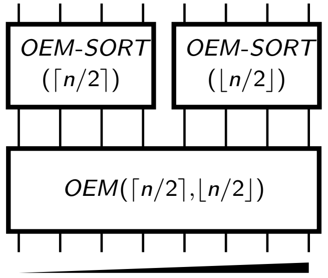
$$depth_{OEM\text{-}SORT}(n) \le depth_{OEM\text{-}SORT}(n/2) + depth_{OEM}(n) = depth_{OEM\text{-}SORT}(n/2) + O(\log n) = O\big((\log n)^2\big)$$

Computation by circuits
Efficient merging and sorting networks

$OEM\text{-}SORT(n)$

size $O\big(n(\log n)^2\big)$

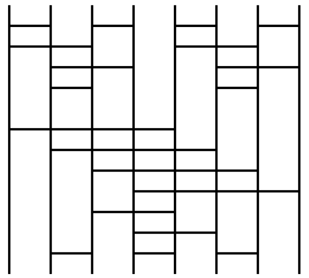
depth $O\big((\log n)^2\big)$



$OEM\text{-}SORT(8)$

size 19

depth 6



Computation by circuits
Efficient merging and sorting networks

A bitonic sequence: $\langle x_1 \ge \dots \ge x_m \le \dots \le x_n \rangle$ $1 \le m \le n$

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), ...

Correctness proof: by zero-one principle (exercise)

(Note: cannot exchange $\ge$ and $\le$ in definition of bitonic!)

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

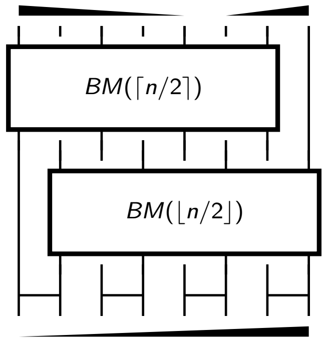
$size_{BM}(n) = O(n \log n)$ $depth_{BM}(n) = O(\log n)$

Computation by circuits
Efficient merging and sorting networks

$BM(n)$

size $O(n \log n)$

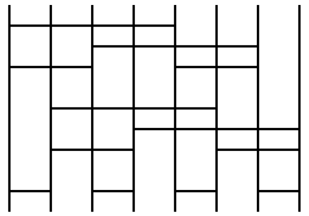
depth $O(\log n)$

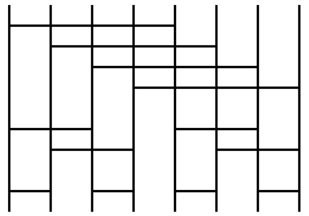


$BM(8)$

size 12

depth 3





Computation by circuits
Efficient merging and sorting networks

Bitonic merge sorting [Batcher: 1968]

When $n = 1$ we are done, otherwise

- sort $\langle x_1, \ldots, x_{\lceil n/2 \rceil} \rangle \mapsto \langle y_1 \geq \cdots \geq y_{\lceil n/2 \rceil} \rangle$ in reverse, recursively
- sort $\langle x_{\lceil n/2 \rceil + 1}, \ldots, x_n \rangle \mapsto \langle y_{\lceil n/2 \rceil + 1} \leq \cdots \leq y_n \rangle$ recursively
- sort bitonic $\langle y_1 \geq \cdots \geq y_m \leq \cdots \leq y_n \rangle$ $\quad m = \lceil n/2 \rceil$ 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 network drawn with "inverted comparators" can be converted into one with only "standard comparators" by a top-down rearrangement

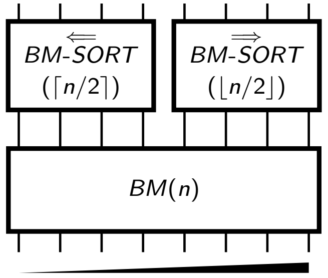
$size_{BM\text{-}SORT}(n) = O\big(n(\log n)^2\big) \quad depth_{BM\text{-}SORT}(n) = O\big((\log n)^2\big)$

Computation by circuits
Efficient merging and sorting networks

BM-SORT(n)

size $O\big(n(\log n)^2\big)$

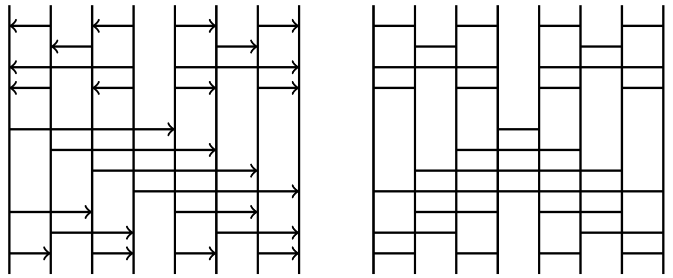
depth $O\big((\log n)^2\big)$


$\overleftarrow{BM\text{-}SORT}$ $(\lceil n/2 \rceil)$
$\overrightarrow{BM\text{-}SORT}$ $(\lfloor n/2 \rfloor)$
$BM(n)$

BM-SORT(8)

size 24

depth 6



Computation by circuits
Efficient merging and sorting networks

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

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

AKS sorting [Ajtai, Komlós, Szemerédi: 1983]

[Paterson: 1990]; [Seiferas: 2009]

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

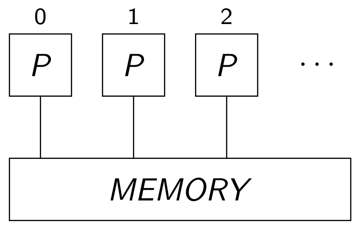
Uses sophisticated graph theory (expanders)

Asymptotically optimal, but has huge constant factors

Parallel computation models
The PRAM model

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

Simple, idealised general-purpose parallel model



Contains

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

Operates in full synchrony

Parallel computation models
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

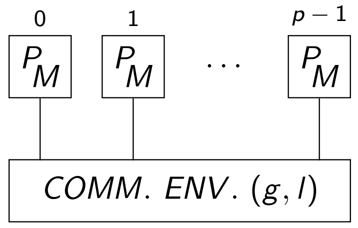
E.g. a linear system solver: $O((\log n)^2)$ steps using n^4 processors :-0

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

Parallel computation models
The BSP model

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

Simple, realistic general-purpose parallel model



Contains

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

Parallel computation models
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

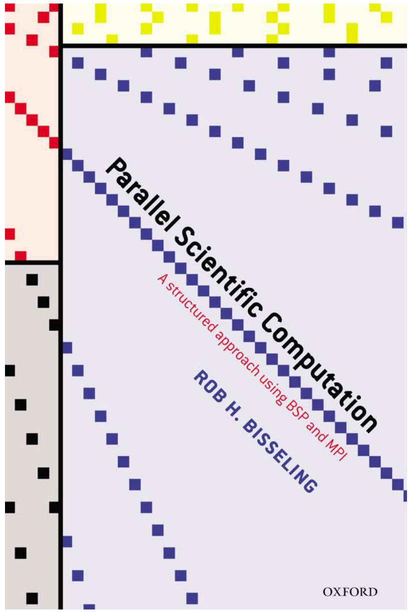
Parameter g corresponds to the network's communication gap (inverse bandwidth) — the time for a data unit to enter/exit the network

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

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

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

Parallel computation models
The BSP model


Parallel Scientific Computation
A structured approach using BSP and MPI
ROB H. BISSELING
OXFORD

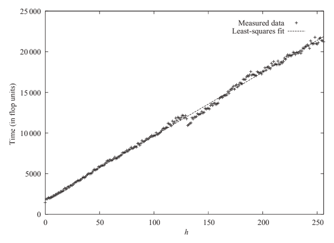


FIG. 1.13. Time of an h-relation on a 64-processor Cray T3E.

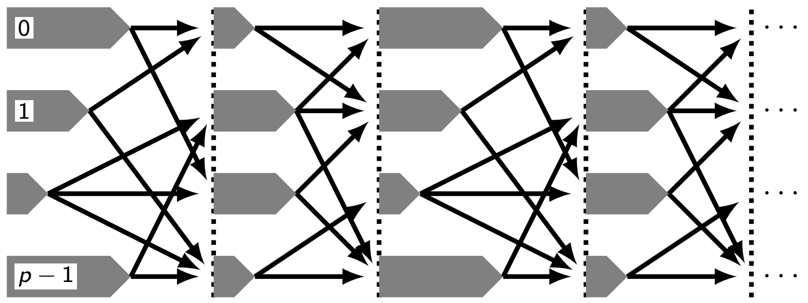
TABLE 1.2. Benchmarked BSP parameters p, g, l and the time of a 0-relation for a Cray T3E. All times are in flop units ($r = 35$ Mflop/s)

p	g	l	$T_{\mathrm{comm}}(0)$
1	36	47	38
2	28	486	325
4	31	679	437
8	31	1193	580
16	31	2018	757
32	72	1145	871
64	78	1825	1440

is a mesh, rather than a torus. Increasing the number of processors makes the subpartition look more like a torus, with richer connectivity.) The time of a 0-relation (i.e. the time of a superstep without communication) displays a smoother behaviour than that of l, and it is presented here for comparison. This time is a lower bound on l, since it represents only part of the fixed cost of a superstep.

Parallel computation models
The BSP model

BSP computation: sequence of parallel supersteps



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

Synchronisation before/after each superstep

Cf. CSP: parallel collection of sequential processes

Parallel computation models
The BSP model

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*:

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

Parallel computation models
The BSP model

For the whole BSP computation with *sync* supersteps:

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

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:

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

Parallel computation models
The BSP model

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)

Parallel computation models
Network routing

BSP network model: complete graph, uniformly accessible (access efficiency described by parameters g, l)

Has to be implemented on concrete networks

Parameters of a network topology (i.e. the underlying graph):

- degree — number of links per node
- diameter — maximum distance between nodes

Low degree — easier to implement

Low diameter — more efficient

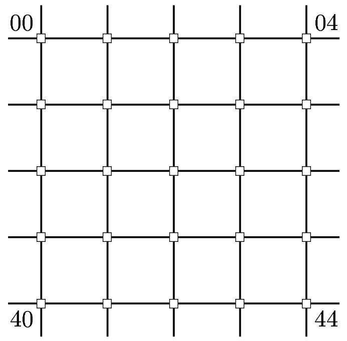
Parallel computation models
Network routing

2D array network

$p = q^2$ processors

degree 4

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



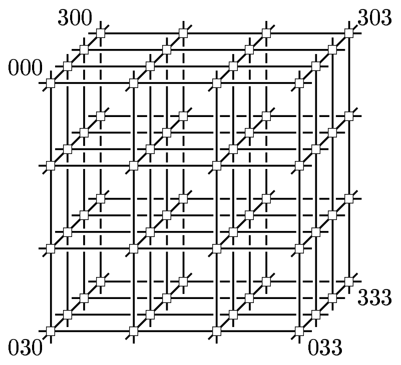
Parallel computation models
Network routing

3D array network

$p = q^3$ processors

degree 6

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



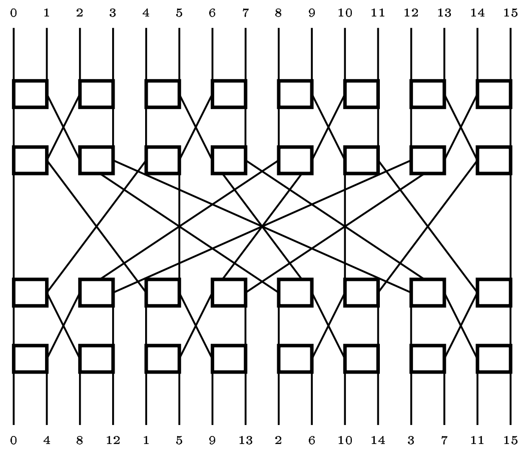
Parallel computation models
Network routing

Butterfly network

$p = q \log q$ processors

degree 4

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



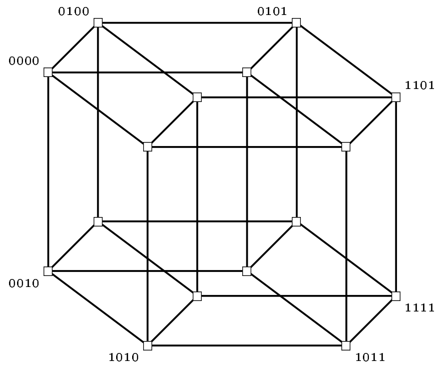
Parallel computation models
Network routing

Hypercube network

$p = 2^q$ processors

degree $\log p = q$

diameter $\log p = q$



Parallel computation models
Network routing

Network	Degree	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

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

Assume distributed routing: no global control

Oblivious routing: path determined only by source and destination

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

Parallel computation models
Network routing

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

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

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

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

Many practical patterns force such "hot spots" on traditional networks

Parallel computation models
Network routing

Routing based on sorting networks

Each processor corresponds to a wire

Each link corresponds to (possibly several) comparators

Routing corresponds to sorting by destination address

Each stage of routing corresponds to a stage of sorting

Such routing is non-oblivious (for individual packets)!

Network	Degree	Diameter
OEM-SORT/BM-SORT	$O((\log p)^2)$	$O((\log p)^2)$
AKS	$O(\log p)$	$O(\log p)$

No “hot spots”: can always route a permutation in $O(\mathit{diameter})$ steps

Requires a specialised network, too messy and impractical

Parallel computation models
Network 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), but non-oblivious overall due to randomness

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

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

Hence constant g, l in the BSP model

Parallel computation models
Network 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

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 running benchmarks

Basic parallel algorithms
Broadcast/combine

The broadcasting problem:

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

The combining problem (complementary to broadcasting):

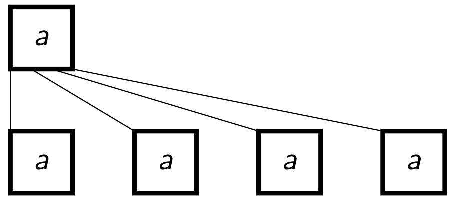
- initially, every processor holds a value a_i, $0 \le i < p$
- at the end, one designated processor must hold $a_0 \bullet \cdots \bullet a_{p-1}$ for a given associative operator $\bullet$ (e.g. $+$)

By symmetry, we only need to consider broadcasting

Basic parallel algorithms
Broadcast/combine

Direct broadcast:

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



$\boxed{\textit{comp } O(p)}$ $\boxed{\textit{comm } O(p)}$ $\boxed{\textit{sync } O(1)}$

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

Basic parallel algorithms
Broadcast/combine

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

$\boxed{\textit{comp } O(\log p)}$ $\boxed{\textit{comm } O(\log p)}$ $\boxed{\textit{sync } O(\log p)}$

Basic parallel algorithms
Broadcast/combine

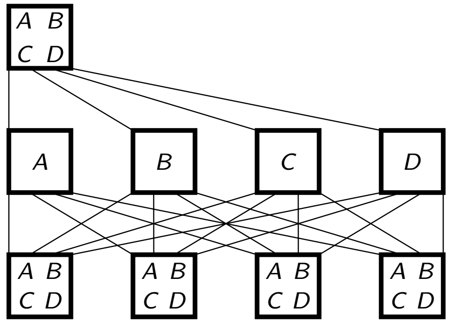
The array broadcasting/combining problem: broadcast/combine an array of size $n \ge p$ elementwise

(effectively, n independent instances of broadcasting/combining)

Basic parallel algorithms
Broadcast/combine

Two-phase array broadcast:

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



$\boxed{comp\ O(n)}$ $\boxed{comm\ O(n)}$ $\boxed{sync\ O(1)}$

Basic parallel algorithms
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

$tree(n)$

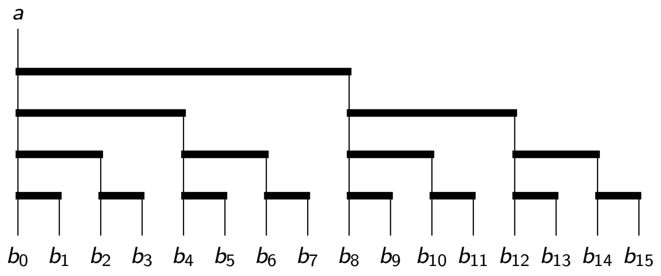
1 input

n outputs

size $n-1$

depth $\log n$

Sequential work $O(n)$



Basic parallel algorithms
Balanced tree and prefix sums

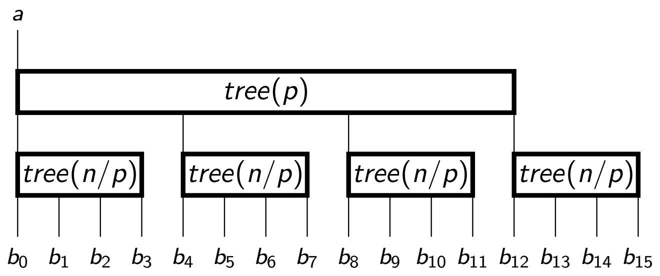
Parallel balanced tree computation

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 $tree(n/p)$

$tree(n)$



Basic parallel algorithms
Balanced tree and prefix sums

Parallel balanced tree computation (contd.)

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

$n \geq p^2$

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

Basic parallel algorithms
Balanced tree and prefix sums

The described parallel balanced tree algorithm is fully optimal:

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

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* $O(1)$!

Basic parallel algorithms
Balanced tree and prefix sums

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

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

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

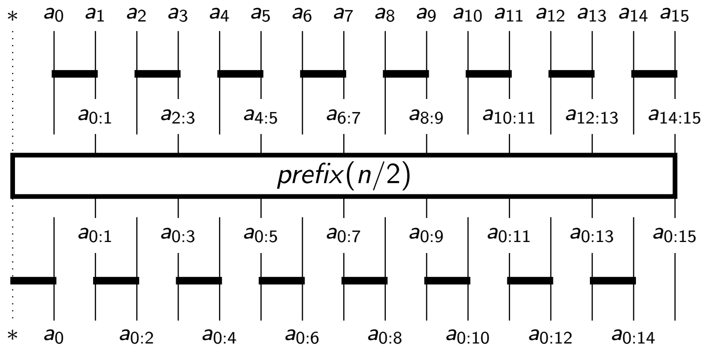
The prefix sums problem:
$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} \mapsto \begin{bmatrix} a_0 \\ a_0 \bullet a_1 \\ a_0 \bullet a_1 \bullet a_2 \\ \vdots \\ a_0 \bullet a_1 \bullet \cdots \bullet a_{n-1} \end{bmatrix}$$

Sequential work $O(n)$

Basic parallel algorithms
Balanced tree and prefix sums

The prefix circuit [Ladner, Fischer: 1980]

prefix(n)



$*$ a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
$a_{0:1}$ $a_{2:3}$ $a_{4:5}$ $a_{6:7}$ $a_{8:9}$ $a_{10:11}$ $a_{12:13}$ $a_{14:15}$
prefix$(n/2)$
$a_{0:1}$ $a_{0:3}$ $a_{0:5}$ $a_{0:7}$ $a_{0:9}$ $a_{0:11}$ $a_{0:13}$ $a_{0:15}$
$*$ a_0 $a_{0:2}$ $a_{0:4}$ $a_{0:6}$ $a_{0:8}$ $a_{0:10}$ $a_{0:12}$ $a_{0:14}$

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

The underlying dag is called the prefix dag

Basic parallel algorithms
Balanced tree and prefix sums

The prefix circuit (contd.)

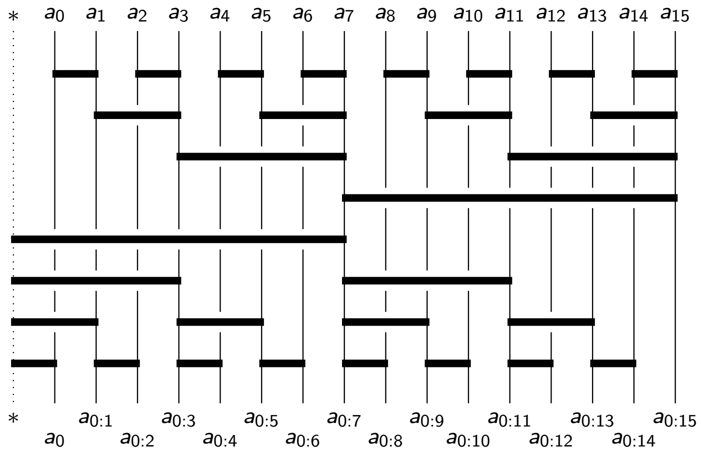
prefix(n)

n inputs

n outputs

size $2n - 2$

depth $2 \log n$



$*$ a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
$*$ $a_{0:1}$ $a_{0:3}$ $a_{0:5}$ $a_{0:7}$ $a_{0:9}$ $a_{0:11}$ $a_{0:13}$ $a_{0:15}$
a_0 $a_{0:2}$ $a_{0:4}$ $a_{0:6}$ $a_{0:8}$ $a_{0:10}$ $a_{0:12}$ $a_{0:14}$

Basic parallel algorithms
Balanced tree and prefix sums

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.

$n \geq p^2$

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

Basic parallel algorithms
Balanced tree and prefix sums

Application: binary addition via Boolean logic

$x + y = z$

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

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

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

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

Basic parallel algorithms
Balanced tree and prefix sums

$x + y = z$

Let $u_i = x_i \wedge y_i \quad v_i = x_i \oplus y_i \quad 0 \leq i < n$

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

$$\begin{array}{ll}
z_0 = v_0 & c_0 = u_0 \\
z_1 = v_1 \oplus c_0 & c_1 = u_1 \vee (v_1 \wedge c_0) \\
\cdots & \cdots \\
z_{n-1} = v_{n-1} \oplus c_{n-2} & c_{n-1} = u_{n-1} \vee (v_{n-1} \wedge c_{n-2}) \\
z_n = c_{n-1} &
\end{array}$$

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

Basic parallel algorithms
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 $c_i = F_{u_i,v_i}(\ldots F_{u_0,v_0}(0))) = F_{u_0,v_0} \circ \cdots \circ F_{u_i,v_i}(0)$

Function composition $\circ$ is associative

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

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 $F_{u_0,v_0} \circ \cdots \circ F_{u_i,v_i}$ 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 $O(\log n)$

Basic parallel algorithms
Fast Fourier Transform and the butterfly dag

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

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

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

$$\textstyle\sum_j \omega^{ij} a_j = b_i \qquad i, j = 0, \ldots, n-1$$

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

Basic parallel algorithms
Fast Fourier Transform and the butterfly dag

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

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

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

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

$B_{s,t} = \sum_{u,v} \omega^{(ms+t)(mu+v)} A_{u,v} = \sum_{u,v} \omega^{msv+tv+mtu} A_{u,v} = \sum_v \left((\omega^m)^{sv} \cdot \omega^{tv} \cdot \sum_u (\omega^m)^{tu} A_{u,v}\right)$, thus $B = \mathcal{F}_{m,\omega^m}(\mathcal{T}_{m,\omega}(\mathcal{F}_{m,\omega^m}(A)))$

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

Equivalent to matrix-matrix product of size m $\qquad \mathcal{F}_{m,\omega^m}(A) = F_{m,\omega^m} \cdot A$

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

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

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

Basic parallel algorithms
Fast Fourier Transform and the butterfly dag

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

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

By recursion, we have the FFT circuit

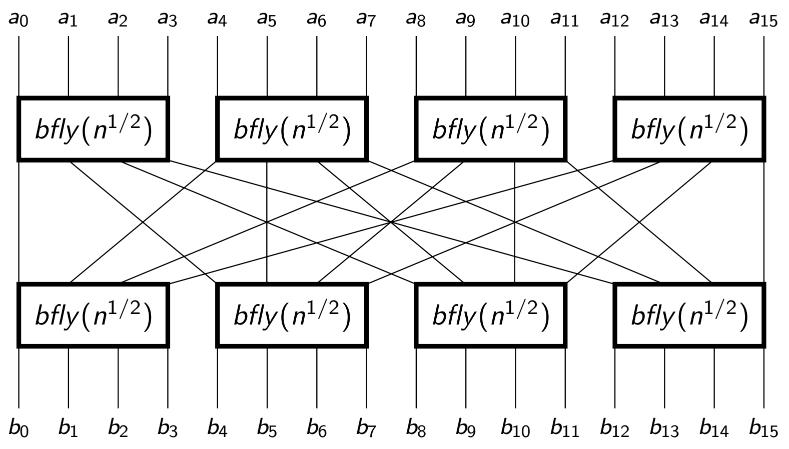
$size_{FFT}(n) = O(n) + 2 \cdot n^{1/2} \cdot size_{FFT}(n^{1/2}) = O(1 \cdot n \cdot 1 + 2 \cdot n^{1/2} \cdot n^{1/2} + 4 \cdot n^{3/4} \cdot n^{1/4} + \cdots + \log n \cdot n \cdot 1) = O(n + 2n + 4n + \cdots + \log n \cdot n) = O(n \log n)$

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

Basic parallel algorithms
Fast Fourier Transform and the butterfly dag

The FFT circuit

$bfly(n)$



a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
$bfly(n^{1/2})$
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14} b_{15}

The underlying dag is called butterfly dag

Basic parallel algorithms

Fast Fourier Transform and the butterfly dag

The FFT circuit and the butterfly dag (contd.)

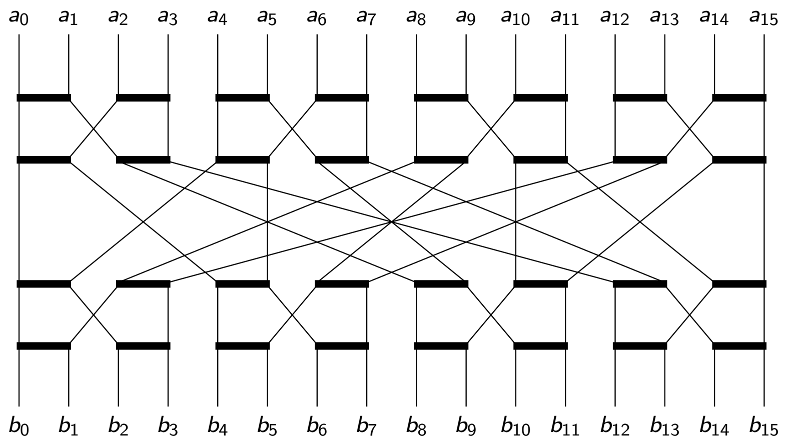
bfly(n)

n inputs

n outputs

size $\frac{n \log n}{2}$

depth $\log n$


a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14} b_{15}

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

Basic parallel algorithms

Fast Fourier Transform and the butterfly dag

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

Basic parallel algorithms

Fast Fourier Transform and the butterfly dag

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

$n \geq p^2$

$\boxed{comp\ O(\frac{n \log n}{p})}$ $\boxed{comm\ O(n/p)}$ $\boxed{sync\ O(1)}$

Basic parallel algorithms

Ordered grid

The ordered 2D grid dag

grid$_2(n)$

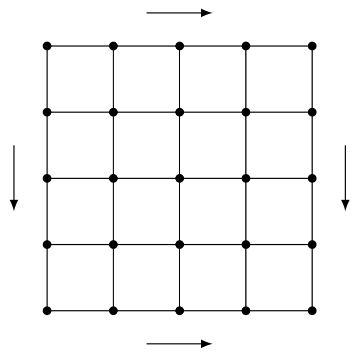
nodes arranged in an $n \times n$ grid

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 $2n - 1$



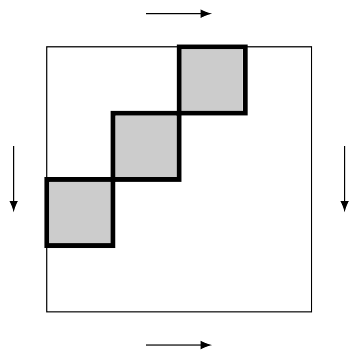
Applications: Gauss–Seidel iteration (single step); triangular system solution; dynamic programming; 1D cellular automata

Sequential work $O(n^2)$

Basic parallel algorithms
Ordered grid

Parallel ordered 2D grid computation

$grid_2(n)$



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

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

Basic parallel algorithms
Ordered grid

Parallel ordered 2D grid computation (contd.)

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

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

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

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

$n \ge p$

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

Basic parallel algorithms
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

$a =$ "define" $b =$ "design"

$LCS(a, b) =$ "dein"

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

Basic parallel algorithms
Ordered grid

LCS computation by dynamic programming

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

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

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

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

	∗	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

$lcs($"define", "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

Basic parallel algorithms
Ordered grid

Parallel LCS computation

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

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

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.

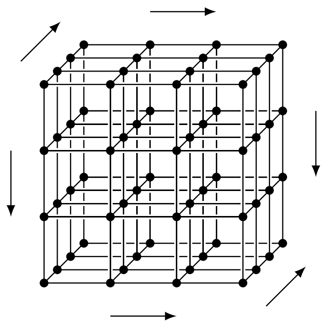
The semi-local LCS algorithm (details omitted) [Krusche, T: 2007]

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

Basic parallel algorithms
Ordered grid

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 $3n - 2$

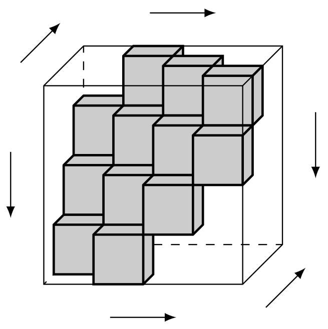
Applications: Gauss–Seidel iteration; Gaussian elimination; dynamic programming; 2D cellular automata

Sequential work $O(n^3)$

Basic parallel algorithms
Ordered grid

Parallel ordered 3D grid computation

$grid_3(n)$



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

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

Basic parallel algorithms
Ordered grid

Parallel ordered 3D grid computation (contd.)

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

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

comp: $(3p^{1/2} - 2) \cdot O\big((n/p^{1/2})^3\big) = O(p^{1/2} \cdot n^3/p^{3/2}) = O(n^3/p)$

comm: $(3p^{1/2} - 2) \cdot O\big((n/p^{1/2})^2\big) = O(p^{1/2} \cdot n^2/p) = O(n^2/p^{1/2})$

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

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

Basic parallel algorithms
Discussion

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

Costs *comp*, *comm*, *sync*: functions of n, p

The goals:

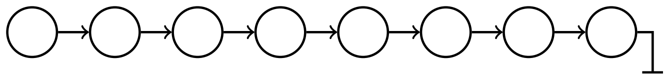
- $comp = comp_{opt} = comp_{seq}/p$
- *comm* scales down with increasing p
- *sync* is a function of p, independent of n

The challenges:

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

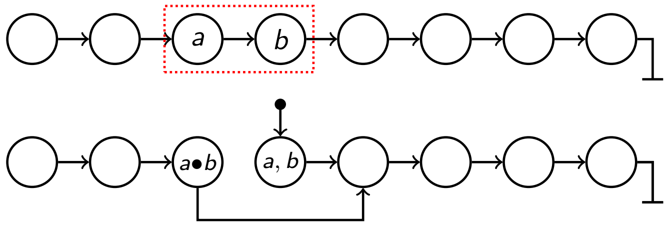
Further parallel algorithms
List contraction and colouring

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



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

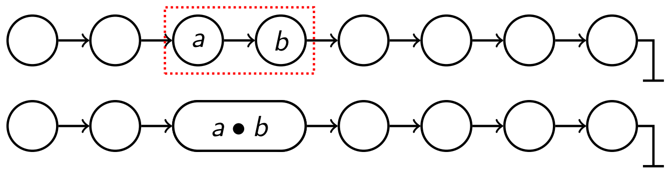
Primitive list operation: pointer jumping



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

Further parallel algorithms
List contraction and colouring

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



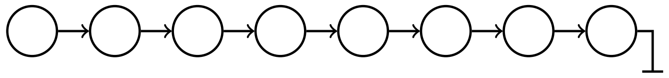
The original a, b are kept implicitly, so that node merging can be reversed

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

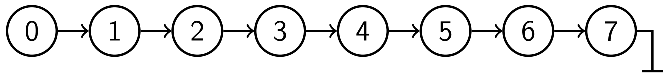
The list expansion problem: restore the original list by reversing the contraction

Further parallel algorithms
List contraction and colouring

Application: list ranking



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



Note the solution should be independent of the merging order!

Further parallel algorithms
List contraction and colouring

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: $k, l \to k + l$

In the fully contracted list, the node contains value n

Further parallel algorithms
List contraction and colouring

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 $(0, n)$

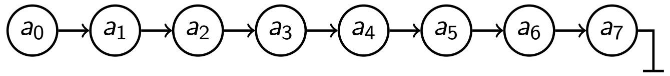
Un-merging operation: $(s, k), (s + k, l) \leftarrow (s, k + l)$

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

Further parallel algorithms
List contraction and colouring

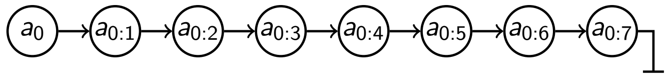
Application: list prefix sums

Initially, each node i contains value a_i



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

The problem: for each node i, find $a_{0:i} = a_0 \bullet a_1 \bullet \cdots \bullet a_i$ by list contraction



Note the solution should be independent of the merging order!

Further parallel algorithms
List contraction and colouring

Application: list prefix sums (contd.)

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

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

Initially, each node assigned a_i

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

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

Further parallel algorithms
List contraction and colouring

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 $b_i = b_{i-1} \bullet a_i$

Further parallel algorithms
List contraction and colouring

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

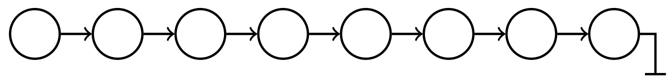
Sequential work $O(n)$ by always contracting at the list's head

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

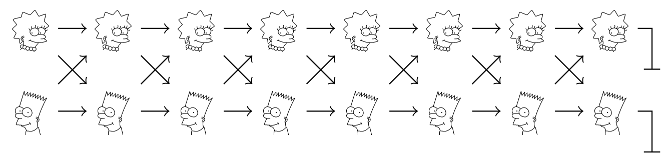
Therefore, we need either node splitting, or efficient symmetry breaking

Further parallel algorithms
List contraction and colouring

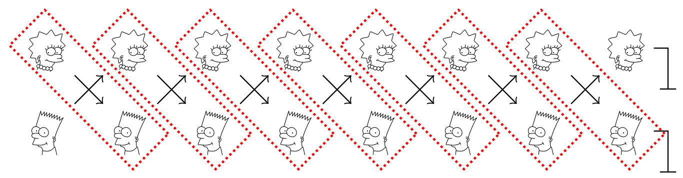
Wyllie's mating [Wyllie: 1979]



Split every node into "forward" node , and "backward" node



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



Further parallel algorithms
List contraction and colouring

Parallel list contraction by Wyllie's mating

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

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

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

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

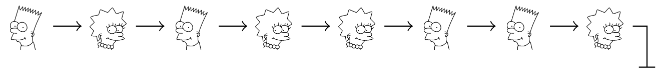
$\boxed{comp\ O(\frac{n \log n}{p})}$ $\boxed{comm\ O(\frac{n \log n}{p})}$ $\boxed{sync\ O(\log n)}$ $n \geq p$

Further parallel algorithms
List contraction and colouring

Random mating [Miller, Reif: 1985]

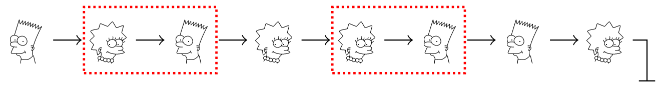
Label every node either "forward" , or "backward"

For each node, labelling independent with probability 1/2



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

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



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

Further parallel algorithms
List contraction and colouring

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 reduced list in a designated processor and contract sequentially

Total work $O(n)$, optimal but randomised

The time bound holds with high probability (whp)

This means "with probability exponentially close to 1" (as a function of n)

$\boxed{comp\ O(n/p)\ \text{whp}}$ $\boxed{comm\ O(n/p)\ \text{whp}}$ $\boxed{sync\ O(\log p)}$

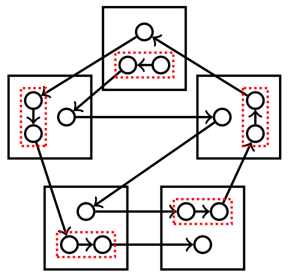
$n \geq p^2 \cdot \log p$

Further parallel algorithms
List contraction and colouring

Block mating

Will mate nodes deterministically

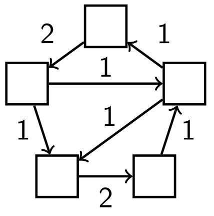
Contract local chains (if any)



Build distribution graph:

- complete weighted digraph on p *supernodes*
- $w(i,j) = |\{u \to v : u \in proc_i, v \in proc_j\}|$

Each processor holds a supernode's outgoing edges



Further parallel algorithms
List contraction and colouring

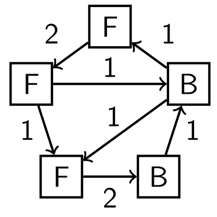
Block mating (contd.)

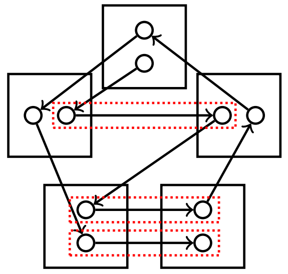
Collect distribution graph in a designated processor

Label every supernode "forward" F or "backward" B, so that $\sum_{i \in F, j \in B} w(i,j) \geq \frac{1}{4} \cdot \sum_{i,j} w(i,j)$ by a sequential greedy algorithm

Scatter supernode labels to processors
By construction of supernode labelling, at least $n/2$ nodes have mates

Merge mating node pairs, obtaining a new list of size at most $3n/4$





Further parallel algorithms
List contraction and colouring

Parallel list contraction by block mating

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

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

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

$\boxed{comp\ O(n/p)}$ $\boxed{comm\ O(n/p)}$ $\boxed{sync\ O(\log p)}$ $n \geq p^3$

Further parallel algorithms
List contraction and colouring

The list k-colouring problem: given a linked list and an integer $k > 1$, assign a colour from $\{0, \dots, k-1\}$ to every node, so that all pairs of adjacent nodes receive a different colour

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

$\boxed{comp\ O(n/p)}$ $\boxed{comm\ O(n/p)}$ $\boxed{sync\ O(\log p)}$

For $k = p$, we can easily achieve (how?)

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

Can we achieve the same for all $k \leq p$? For $k = O(1)$?

Further parallel algorithms
List contraction and colouring

Deterministic coin tossing [Cole, Vishkin: 1986]

Given a k-colouring, $k > 6$; colours represented in binary

Consider every node v. We have $col(v) \neq col(next(v))$.

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

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

After re-colouring, still have $col(v) \neq col(next(v))$

Number of colours reduced from k to $2 \log k \ll k$

Further parallel algorithms
List contraction and colouring

Parallel list 3-colouring by deteministic coin tossing:

- compute a p-colouring
- reduce the number of colours from p to 6 by deteministic coin tossing: $O(\log^* k)$ rounds

$$\log^* k = \min r : \underbrace{\log \ldots \log}_{(r \text{ times})} k \leq 1$$

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

$\boxed{comp\ O(n/p)}$ $\boxed{comm\ O(n/p)}$ $\boxed{sync\ O(\log^* p)}$

Further parallel algorithms
Sorting

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

The sorting problem: arrange elements of a in increasing order

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

Assume the comparison model: primitives $<$, $>$, no bitwise operations

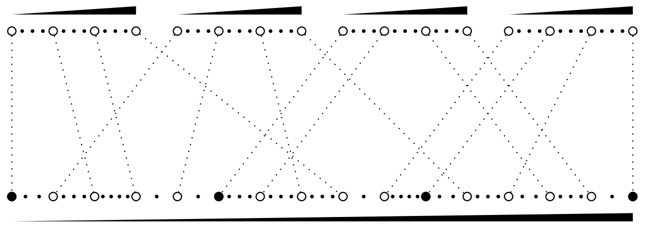
Sequential work $O(n \log n)$ e.g. by mergesort

Parallel sorting based on an AKS sorting network

$\boxed{comp\ O\left(\frac{n \log n}{p}\right)}$ $\boxed{comm\ O\left(\frac{n \log n}{p}\right)}$ $\boxed{sync\ O(\log n)}$

Further parallel algorithms
Sorting

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



Every processor

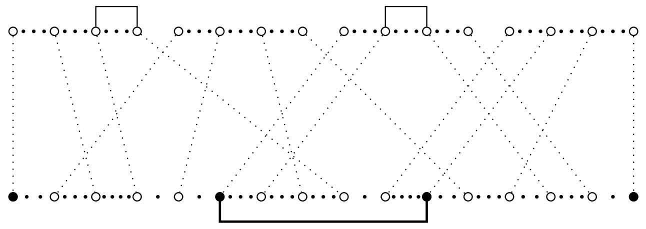
- reads a subarray of size n/p and sorts it sequentially
- selects from its subarray p samples at regular intervals

A designated processor

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

Further parallel algorithms
Sorting

Parallel sorting by regular sampling (contd.)



In each subarray of size n/p, samples define p local blocks of size n/p^2

In the whole array of size n, splitters define p global buckets of size n/p

Further parallel algorithms
Sorting

Parallel sorting by regular sampling (contd.)

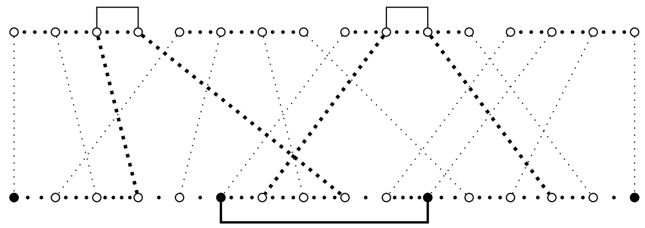
The designated processor broadcasts the splitters

Every processor

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

Further parallel algorithms
Sorting

Claim: each bucket has size $\leq 2n/p$



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

A bucket can intersect $\leq p$ low blocks and $\leq p$ high blocks

Bucket size is at most $(p + p) \cdot n/p^2 = 2n/p$ □

$\boxed{comp\ O(\frac{n \log n}{p})}$ $\boxed{comm\ O(n/p)}$ $\boxed{sync\ O(1)}$ $n \geq p^3$

Further parallel algorithms
Convex hull

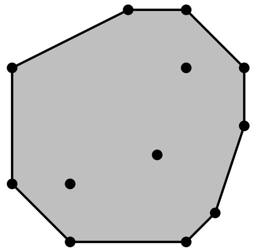
Set $S \subseteq \mathbb{R}^d$ is convex, if for all x, y in S, every point between x and y is also in S

$A \subseteq \mathbb{R}^d$

The convex hull conv A is the smallest convex set containing A

conv A is a polytope, defined by its vertices $A_i \in A$

Set A is in convex position, if every its point is a vertex of conv A



Further parallel algorithms
Convex hull

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

The (discrete) convex hull problem: find vertices of conv a

Output must be ordered: every vertex must "know" its neighbours

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

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

To sort $[x_0, \ldots, x_{n-1}]$:

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

Further parallel algorithms
Convex hull

The discrete convex hull problem

$d = 2$: two neighbours per vertex; output size $2n$

$d = 3$: on average, $O(1)$ neighbours per vertex; output size $O(n)$

Sequential work $O(n \log n)$ by Graham's scan or by mergehull

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

From now on, will concentrate on $d = 2, 3$

Further parallel algorithms
Convex hull

$A \subseteq \mathbb{R}^d$ Let $0 \leq \epsilon \leq 1$

Set $E \subseteq A$ is an ϵ-net for A, if any halfspace with no points in E covers $\leq \epsilon|A|$ points in A

May always be assumed to be in convex position

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

May not be in convex position

Easy to construct in 2D, much harder in 3D and higher

Further parallel algorithms
Convex hull

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

Claim. Union of ϵ-approximations for A', A'' is ϵ-approximation for $A'' \cup A''$

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

Proofs: Easy by definitions. □

Further parallel algorithms
Convex hull

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

Claim. A $1/r$-net for A of size $\leq 2r$ exists and can be computed in sequential work $O(n \log n)$.

Proof. Consider convex hull of A and an arbitrary interior point v

Partition A into triangles: base at a hull edge, apex at v

A triangle is heavy if it contains $> n/r$ points of A, otherwise light

Heavy triangles: for each triangle, take both hull vertices

Light triangles: for each triangle chain, greedy next-fit bin packing

- combine adjacent triangles into bins with $\leq n/r$ points
- for each bin, take both boundary hull vertices

In total $\leq 2r$ heavy triangles and bins, hence taken $\leq 2r$ points □

Further parallel algorithms
Convex hull

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

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

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

Further parallel algorithms
Convex hull

Parallel 2D hull computation by generalised regular sampling

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

Every processor

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

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

A designated processor

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

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

Further parallel algorithms
Convex hull

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

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

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

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

Further parallel algorithms
Convex hull

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

The designated processor broadcasts the splitters

Every processor

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

$\boxed{comp\ O(\frac{n \log n}{p})}$ $\boxed{comm\ O(n/p)}$ $\boxed{sync\ O(1)}$ $\qquad n \geq p^3$

Further parallel algorithms
Convex hull

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

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

Proof: [Brönnimann, Goodrich: 1995] □

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

Proof: [Matoušek: 1992] □

Better approximations are possible, but are slower to compute

Further parallel algorithms
Convex hull

Parallel 3D hull computation by generalised regular sampling

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

Every processor

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

Set of all samples: $1/p$-approximation for set a

A designated processor

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

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

Further parallel algorithms
Convex hull

Parallel 3D 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 splitter polytope with $O(p)$ edges

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

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

Further parallel algorithms
Convex hull

Parallel 3D 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

$\boxed{comp\ O(\frac{n \log n}{p})}$ $\boxed{comm\ O(n/p)}$ $\boxed{sync\ O(1)}$ $n \gg p$

Further parallel algorithms

Selection

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

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

E.g. median selection: $k = n/2$

As before, assume the comparison model

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

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

Further parallel algorithms

Selection

Standard approach to selection: eliminate elements in rounds

In each round:

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

Each time, we eliminate elements on "wrong" side of median-of-medians a_l

Further parallel algorithms

Selection

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

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

Each round removes at least a constant fraction of elements of a

Data reduction rate is exponential

Further parallel algorithms

Selection

More general approach: elimination by regular sampling in rounds

In each round:

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

Selecting samples and splitters:

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

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

Further parallel algorithms
Selection

Elimination by regular sampling (contd.)

Let a_{l^-}, a_{l^+} be adjacent splitters, such that $l^- \leq k \leq l^+$

Splitters a_{l^-}, a_{l^+} define the bucket

- eliminate all a_i outside the bucket

For work-optimality, sufficient to use constant subarray size and constant sampling frequency (as in standard approach)

Since the array size decreases in every round, we can increase the sampling frequency to reduce the number of rounds, while keeping work-optimality

Further parallel algorithms
Selection

Parallel selection

$\boxed{comp\ O(n/p)}$ $\boxed{comm\ O(n/p)}$

$\boxed{sync\ O(\log p)}$ [Ishimizu+: 2002]

$\boxed{sync\ O(\log\log n)}$ [Fujiwara+: 2000]

$\boxed{sync\ O(1)}$ randomised whp [Gerbessiotis, Siniolakis: 2003]

Further parallel algorithms
Selection

Parallel selection by accelerated regular sampling

Main idea: variable sampling frequency in different rounds. As array size decreases, we can afford to increase sampling frequency.

Data reduction rate now superexponential

Selection can be completed in $O(\log\log p)$ rounds

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

$\boxed{comp\ O(n/p)}$ $\boxed{comm\ O(n/p)}$ $\boxed{sync\ O(\log\log p)}$ $n \gg p$

Parallel matrix algorithms
Matrix-vector multiplication

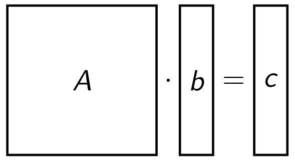
Let A, b, c be a matrix and two vectors of size n

The matrix-vector multiplication problem

$A \cdot b = c$

$c_i = \sum_j A_{ij} \cdot b_j$

$0 \leq i, j < n$



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

Sequential work $O(n^2)$

Parallel matrix algorithms
Matrix-vector multiplication

The matrix-vector multiplication circuit

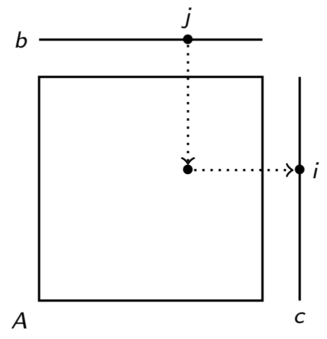
$c_i \leftarrow 0$

$c_i \overset{+}{\leftarrow} A_{ij} \cdot b_j$ ("add to c_i, asynchronously")

$0 \leq i, j < n$

An ij-square of nodes, each representing an elementary product

size $O(n^2)$, depth $O(1)$



Parallel matrix algorithms
Matrix-vector 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 ij-square into a regular grid of $p = p^{1/2} \cdot p^{1/2}$ square blocks

Matrix A gets partitioned into p square blocks A_{IJ} of size $n/p^{1/2}$

Vectors b, c each gets partitioned into $p^{1/2}$ linear blocks b_J, c_I of size $n/p^{1/2}$

$0 \leq I, J < p^{1/2}$

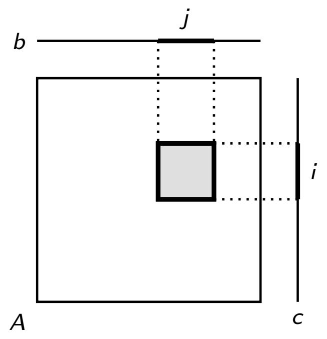
Parallel matrix algorithms
Matrix-vector multiplication

Parallel matrix-vector multiplication (contd.)

$c_I \leftarrow 0$

$c_I \overset{+}{\leftarrow} A_{IJ} \cdot b_J$

$0 \leq I, J < p^{1/2}$



Parallel matrix algorithms
Matrix-vector multiplication

Parallel matrix-vector multiplication (contd.)

Vector c in external memory is initialised by zero values

Every processor

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

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

$\boxed{comp\ O(\frac{n^2}{p})}$ $\boxed{comm\ O(\frac{n}{p^{1/2}})}$ $\boxed{sync\ O(1)}$ $n \geq p$

Parallel matrix algorithms
Matrix multiplication

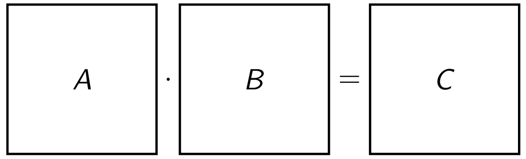
Let A, B, C be matrices of size n

The matrix multiplication problem

$A \cdot B = C$

$C_{ik} = \sum_j A_{ij} \cdot B_{jk}$

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



Overall, n^3 elementary products $A_{ij} \cdot B_{jk}$

Sequential work $O(n^3)$

Parallel matrix algorithms
Matrix multiplication

The matrix multiplication circuit

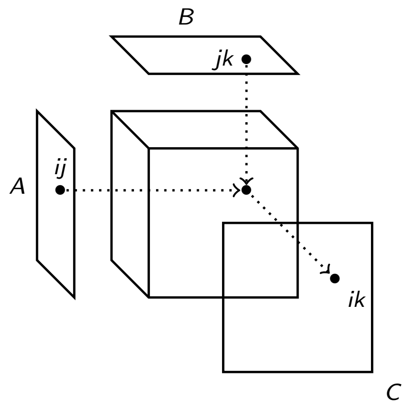
$C_{ik} \leftarrow 0$

$C_{ik} \overset{+}{\leftarrow} A_{ij} \cdot B_{jk}$

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

An *ijk*-cube of nodes, each representing an elementary product

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



Parallel matrix algorithms
Matrix multiplication

Parallel matrix multiplication

Partition the *ijk*-cube into a regular grid of $p = p^{1/3} \cdot p^{1/3} \cdot p^{1/3}$ cubic blocks

Matrices A, B, C each gets partitioned into $p^{2/3}$ square blocks A_{IJ}, B_{JK}, C_{IK} of size $n/p^{1/3}$

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

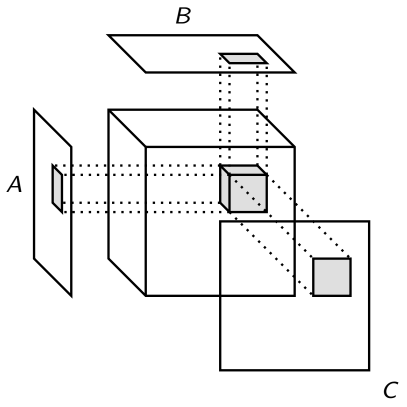
Parallel matrix algorithms
Matrix multiplication

Parallel matrix multiplication (contd.)

$C_{IK} \leftarrow 0$

$C_{IK} \overset{+}{\leftarrow} A_{IJ} \cdot B_{JK}$

$0 \le I, J, K < p^{1/3}$



Parallel matrix algorithms
Matrix multiplication

Parallel matrix multiplication (contd.)

Matrix C in external memory is initialised by zero values

Every processor

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

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

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

Parallel matrix algorithms
Matrix multiplication

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

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

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

Let V be the subset of *ijk*-cube computed by a certain processor

For at least one processor: $|V| \ge \frac{n^3}{p}$

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

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

Loomis–Whitney inequality: $|A| \cdot |B| \cdot |C| \ge |V|^2$

We have *comm* $\ge |A| + |B| + |C| \ge 3(|A| \cdot |B| \cdot |C|)^{1/3} \ge 3|V|^{2/3} \ge 3\left(\frac{n^3}{p}\right)^{2/3} = \frac{3n^2}{p^{2/3}}$, hence *comm* $= \Omega\left(\frac{n^2}{p^{2/3}}\right)$ □

Parallel matrix algorithms
Matrix multiplication

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

Includes e.g.

- numerical matrix multiplication using primitives $+$, $\cdot$
- Boolean matrix multiplication using primitives $\vee$, $\wedge$

Excludes e.g.

- numerical matrix multiplication with extra primitive $-$ (Strassen-like)
- Boolean matrix multiplication with extra primitive `if/then`

Parallel matrix algorithms
Fast matrix multiplication

Recursive block matrix multiplication: $A \cdot B = C$

$$A = \begin{bmatrix} A_{\underline{00}} & A_{\underline{01}} \\ A_{\underline{10}} & A_{\underline{11}} \end{bmatrix} \qquad B = \begin{bmatrix} B_{\underline{00}} & B_{\underline{01}} \\ B_{\underline{10}} & B_{\underline{11}} \end{bmatrix} \qquad C = \begin{bmatrix} C_{\underline{00}} & C_{\underline{01}} \\ C_{\underline{10}} & C_{\underline{11}} \end{bmatrix}$$

$$C_{\underline{00}} = A_{\underline{00}} \cdot B_{\underline{00}} + A_{\underline{01}} \cdot B_{\underline{10}} \qquad C_{\underline{01}} = A_{\underline{00}} \cdot B_{\underline{01}} + A_{\underline{01}} \cdot B_{\underline{11}}$$
$$C_{\underline{10}} = A_{\underline{10}} \cdot B_{\underline{00}} + A_{\underline{11}} \cdot B_{\underline{10}} \qquad C_{\underline{11}} = A_{\underline{10}} \cdot B_{\underline{01}} + A_{\underline{11}} \cdot B_{\underline{11}}$$

8 block multiplications (recursive calls)

Parallel matrix algorithms
Fast matrix multiplication

Strassen's matrix multiplication: $A \cdot B = C$

Let A, B, C be numerical matrices: primitives $+$, $-$, $\cdot$ on matrix elements

$$A = \begin{bmatrix} A_{\underline{00}} & A_{\underline{01}} \\ A_{\underline{10}} & A_{\underline{11}} \end{bmatrix} \qquad B = \begin{bmatrix} B_{\underline{00}} & B_{\underline{01}} \\ B_{\underline{10}} & B_{\underline{11}} \end{bmatrix} \qquad C = \begin{bmatrix} C_{\underline{00}} & C_{\underline{01}} \\ C_{\underline{10}} & C_{\underline{11}} \end{bmatrix}$$

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

$$C_{\underline{00}} = D^{(0)} + D^{(3)} - D^{(4)} + D^{(6)} \qquad C_{\underline{01}} = D^{(2)} + D^{(4)}$$
$$C_{\underline{10}} = D^{(1)} + D^{(3)} \qquad C_{\underline{11}} = D^{(0)} - D^{(1)} + D^{(2)} + D^{(5)}$$

7 block multiplications (recursive calls)

Parallel matrix algorithms
Fast matrix multiplication

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

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

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

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

Parallel matrix algorithms
Fast matrix multiplication

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

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

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

Number of linear operations turns out to be irrelevant

Sequential work $O(n^\omega)$

Parallel matrix algorithms
Fast matrix multiplication

Some specific instances of Strassen-like 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	
...	...	...	...	
HUGE	HUGE	HUGE	2.3755	[Coppersmith, Winograd: 1987]
HUGE	HUGE	HUGE	2.3737	[Stothers: 2010]
HUGE	HUGE	HUGE	2.3727	[Vassilevska–Williams: 2011]

Parallel matrix algorithms
Fast matrix multiplication

Parallel Strassen-like matrix multiplication

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

At level $\log_R p$, we have p independent matrix multiplication tasks

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

Parallel matrix algorithms
Fast matrix multiplication

Parallel Strassen-like matrix multiplication (contd.)

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

Assigning matrix elements to processors:

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

E.g. cyclic distribution

Such distribution allows linear operations on matrix blocks without communication

Parallel matrix algorithms
Fast matrix multiplication

Parallel Strassen-like matrix multiplication (contd.)

Each processor reads its assigned elements of A, B

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

Each processor writes back its assigned elements of C

$\boxed{comp\ O\left(\frac{n^\omega}{p}\right)}$ $\boxed{comm\ O\left(\frac{n^2}{p^{2/\omega}}\right)}$ $\boxed{sync\ O(1)}$

Parallel matrix algorithms
Fast matrix multiplication

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

Proof. By graph expansion, generalises the Loomis-Whitney inequality

[Ballard+:2012]

Parallel matrix algorithms
Boolean matrix multiplication

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

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

Primitives $\vee$, $\wedge$, `if/then` on matrix elements

$C_{ik} = \bigvee_j A_{ik} \wedge B_{jk}$

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

Overall, n^3 elementary products $A_{ij} \wedge B_{jk}$

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

Sequential work $O(n^\omega)$ using a Strassen-like algorithm

Parallel matrix algorithms
Boolean matrix multiplication

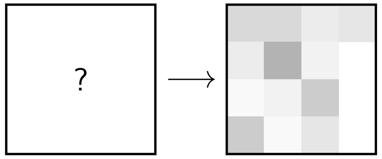
Parallel Boolean matrix multiplication

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

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

$K = K(\epsilon)$, where ϵ is the "degree of random-likeness"

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



We shall call this the regular decomposition of a Boolean matrix

Parallel matrix algorithms
Boolean matrix multiplication

Parallel Boolean matrix multiplication (contd.)

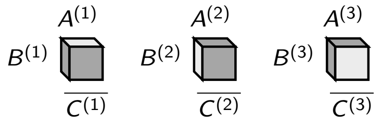
$A \wedge B = C$

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

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

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

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



Matrices $A^{(1)}$, $A^{(2)}$, $A^{(3)}$, $B^{(1)}$, $B^{(2)}$, $B^{(3)}$, $C^{(1)}$, $C^{(2)}$, $C^{(3)}$ can be "efficiently" computed from A, B, C

Parallel matrix algorithms
Boolean matrix multiplication

Parallel Boolean matrix multiplication (contd.)

$A \wedge B = \overline{C}$

Partition the *ijk*-cube into a regular grid of $p^3 = p \cdot p \cdot p$ cubic blocks

Matrices A, B, $\overline{C}$ each gets partitioned into p^2 square blocks A_{IJ}, B_{JK}, $\overline{C}_{IK}$ of size n/p

$0 \leq I, J, K < p$

Parallel matrix algorithms
Boolean matrix multiplication

Parallel Boolean matrix multiplication (contd.)

Every processor

- assigned to compute a "slab" of p^2 cubic blocks $A_{IJ} \wedge B_{JK} = \overline{C}^J_{IK}$ for a fixed J and all I, K
- reads blocks A_{IJ}, B_{JK} and computes $\overline{C}^J_{IJ}$ for all I, K
- computes the three-way regular decomposition for the block product and determines the submatrices having very 1s

$0 \leq I, J, K < p$

Parallel matrix algorithms
Boolean matrix multiplication

Parallel Boolean matrix multiplication (contd.)

Recompute $A \wedge B = \overline{C}$ from block regular decompositions by a Strassen-like algorithm

Communication saved by only sending the positions of 1s

$\boxed{comp\ O\left(\frac{n^\omega}{p}\right)}$ $\boxed{comm\ O\left(\frac{n^2}{p}\right)}$ $\boxed{sync\ O(1)}$ $n \ggg\ggg\ggg\ggg p$:-/

Parallel matrix algorithms
Triangular system solution

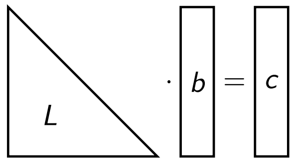
Let L, b, c be a matrix and two vectors of size n

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

$L \cdot b = c$

$\sum_j L_{ij} \cdot b_j = c_i$

$0 \leq j \leq i < n$



The triangular system problem: given L, c, find b

Parallel matrix algorithms

Triangular system solution

Forward substitution

$L \cdot b = c$

$L_{00} \cdot b_0 = c_0$	$b_0 \leftarrow L_{00}^{-1} \cdot c_0$
$L_{10} \cdot b_0 + L_{11} \cdot b_1 = c_1$	$b_1 \leftarrow L_{11}^{-1} \cdot (c_1 - L_{10} \cdot b_0)$
$L_{20} \cdot b_0 + L_{21} \cdot b_1 + L_{22} \cdot b_2 = c_2$	$b_2 \leftarrow L_{22}^{-1} \cdot (c_2 - L_{20} \cdot b_0 - L_{21} \cdot b_1)$
...	...
$\sum_{j:j\le i} L_{ij} \cdot b_j = c_i$	$b_i \leftarrow L_{ii}^{-1} \cdot (c_i - \sum_{j:j<i} L_{ij} \cdot b_j)$
...	...
$\sum_{j:j\le n-1} L_{n-1,j} \cdot b_j = c_{n-1}$	$b_{n-1} \leftarrow L_{n-1,n-1}^{-1} \cdot (c_{n-1} - \sum_{j:j<n-1} L_{n-1,j} \cdot b_j)$

Sequential work $O(n^2)$

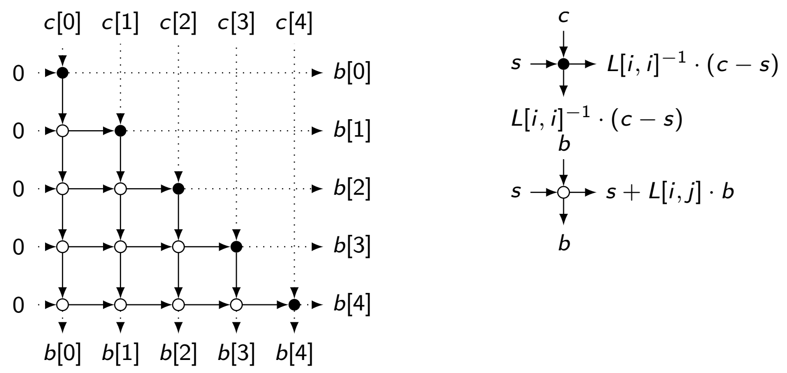
Symmetrically, an upper triangular system solved by back substitution

Parallel matrix algorithms

Triangular system solution

Parallel forward substitution by 2D grid dag

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



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

Parallel matrix algorithms

Triangular system solution

Block forward substitution

$L \cdot b = c$

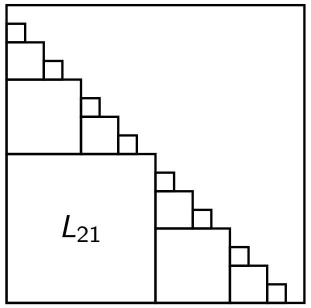
$$\begin{bmatrix} L_{00} & \circ \\ L_{10} & L_{11} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$$

Recursion: two half-sized subproblems

$L_{00} \cdot b_0 = c_0$ by recursion

$L_{11} \cdot b_1 = c_1 - L_{10} \cdot b_1$ by recursion

Sequential work $O(n^2)$



Parallel matrix algorithms

Triangular system solution

Parallel block forward substitution

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

At each level, the two subproblems are dependent, hence recursion tree unfolded depth-first

At level $\log p$, a task fits in a single processor

level	tasks	task size	each task
0	1	n	parallel
1	2	$n/2$	
2	2^2	$n/2^2$	
...			
$\log p$	p	n/p	sequential
...			
$\log n$	n	1	

Parallel matrix algorithms
Triangular system solution

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

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

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

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

Parallel matrix algorithms
Gaussian elimination

Let A, L, U be matrices of size n

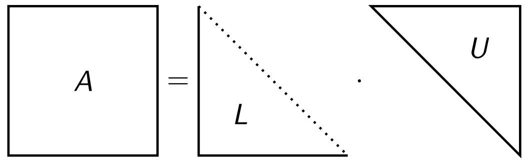
L is unit lower triangular: $L_{ij} = \begin{cases} 0 & 0 \le i < j < n \\ 1 & 0 \le i = j < n \\ \text{arbitrary} & \text{otherwise} \end{cases}$

U is upper triangular: $U_{ij} = \begin{cases} 0 & 0 \le j < i < n \\ \text{arbitrary} & \text{otherwise} \end{cases}$

$A = L \cdot U$

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

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



The LU decomposition problem: given A, find L, U

Parallel matrix algorithms
Gaussian elimination

Generic Gaussian elimination

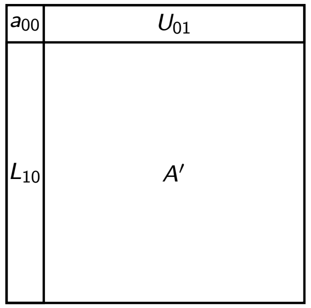
$A = L \cdot U$

$$\begin{bmatrix} A_{00} & A_{0\underline{1}} \\ A_{\underline{1}0} & A_{\underline{1}\underline{1}} \end{bmatrix} = \begin{bmatrix} 1 & \circ \\ L_{\underline{1}0} & L_{\underline{1}\underline{1}} \end{bmatrix} \begin{bmatrix} A_{00} & A_{0\underline{1}} \\ \circ & U_{\underline{1}\underline{1}} \end{bmatrix}$$

First step of elimination: pivot A_{00}

$$\begin{bmatrix} A_{00} & A_{0\underline{1}} \\ A_{\underline{1}0} & A_{\underline{1}\underline{1}} \end{bmatrix} = \begin{bmatrix} 1 & \circ \\ L_{\underline{1}0} & I \end{bmatrix} \begin{bmatrix} A_{00} & A_{0\underline{1}} \\ \circ & A'_{\underline{1}\underline{1}} \end{bmatrix}$$

$L_{\underline{1}0} \leftarrow A_{\underline{1}0} \cdot A_{00}^{-1}$ $\qquad A'_{\underline{1}\underline{1}} \leftarrow A_{\underline{1}\underline{1}} - L_{\underline{1}0} \cdot A_{0\underline{1}}$



Continue elimination on reduced matrix $A'_{\underline{1}\underline{1}} = L_{\underline{1}\underline{1}} \cdot U_{\underline{1}\underline{1}}$

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

Sequential work $O(n^3)$

Parallel matrix algorithms
Gaussian elimination

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

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

Parallel matrix algorithms
Gaussian elimination

Block generic Gaussian elimination

$A = L \cdot U$

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & \circ \\ L_{10} & L_{11} \end{bmatrix} \begin{bmatrix} U_{00} & U_{01} \\ \circ & U_{11} \end{bmatrix}$$

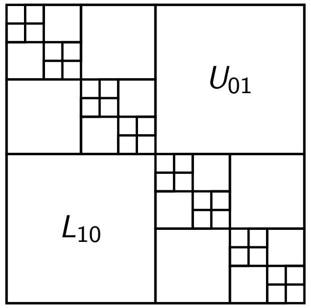
Recursion: two half-sized subproblems

$A_{00} = L_{00} \cdot U_{00}$ by recursion

$U_{01} \leftarrow L_{00}^{-1} \cdot A_{01} \qquad L_{10} \leftarrow A_{10} \cdot U_{00}^{-1}$

$A_{11} - L_{10} \cdot U_{01} = L_{11} \cdot U_{11}$ by recursion

$$L^{-1} \leftarrow \begin{bmatrix} L_{00}^{-1} & \circ \\ -L_{11}^{-1} L_{10} L_{00}^{-1} & L_{11}^{-1} \end{bmatrix} \quad U^{-1} \leftarrow \begin{bmatrix} U_{00}^{-1} & -U_{00}^{-1} U_{10} U_{11}^{-1} \\ \circ & U_{11}^{-1} \end{bmatrix}$$



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

Parallel matrix algorithms
Gaussian elimination

Parallel block generic Gaussian elimination

At each level, the two subproblems are dependent, hence recursion tree unfolded depth-first

At level $\alpha \log p$, $\alpha \geq 1/2$, a task fits in a single processor

level	tasks	task size	each task
0	1	n	parallel
1	2	$n/2$	
2	2^2	$n/2^2$	
...			
$\alpha \log p$	p^α	n/p^α	sequential
...			
$\log n$	n	1	

Parallel matrix algorithms
Gaussian elimination

Parallel block generic Gaussian elimination (contd.)

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

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

Threshold level controlled by parameter 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

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

Parallel matrix algorithms
Gaussian elimination

Parallel LU decomposition (contd.)

In particular:

$\alpha = 1/2$

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

Cf. 2D grid

$\alpha = 2/3$

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

Cf. matrix multiplication

Parallel graph algorithms
Algebraic path problem

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

Addition commutative, associative, has identity $\boxed{0}$

$a \oplus b = b \oplus a \quad a \oplus (b \oplus c) = (a \oplus b) \oplus c \quad a \oplus \boxed{0} = \boxed{0} \oplus a = a$

Multiplication associative, has annihilator $\boxed{0}$ and identity $\boxed{1}$

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

Multiplication distributes over addition

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

In general, no subtraction or division!

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

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

Parallel graph algorithms
Algebraic path problem

Some specific semirings:

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

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

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

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

Boolean closure $a^* = 1 \vee a \vee a \vee a \vee \ldots = 1$

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

In matrix semirings, closures are more interesting

Parallel graph algorithms
Algebraic path problem

A semiring is closed, if

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

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

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

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

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

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

Parallel graph algorithms

Algebraic path problem

Let A be a matrix of size n over a semiring

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

Numerical algebraic path problem: equivalent to matrix inversion

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

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

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

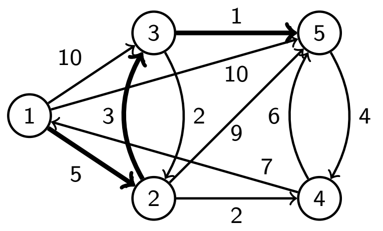
Boolean A^*: the graph's transitive closure

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

Parallel graph algorithms

Algebraic path problem

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



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

Parallel graph algorithms

Algebraic path problem

Floyd–Warshall algorithm [Floyd, Warshall: 1962]

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

Weights may be negative; assume no negative cycles

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

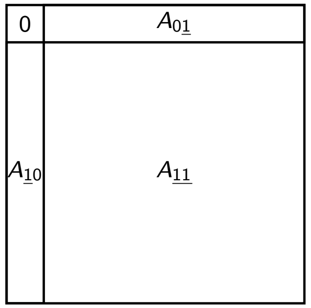
Replace each weight A_{ij}, $i, j \neq 0$, with $A_{i0} + A_{0j}$, if that gives a shortcut from i to j

$A'_{\underline{11}} \leftarrow A_{\underline{11}} \oplus A_{\underline{1}0} \odot A_{0\underline{1}} = \min(A_{\underline{11}}, A_{\underline{1}0} + A_{0\underline{1}})$

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

Generic Gaussian elimination in disguise

Sequential work $O(n^3)$



Parallel graph algorithms

Algebraic path problem

Block Floyd–Warshall algorithm

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \qquad A^* = \begin{bmatrix} A''_{00} & A''_{01} \\ A''_{10} & A''_{11} \end{bmatrix}$$

Recursion: two half-sized subproblems

$A'_{00} \leftarrow A^*_{00}$ by recursion

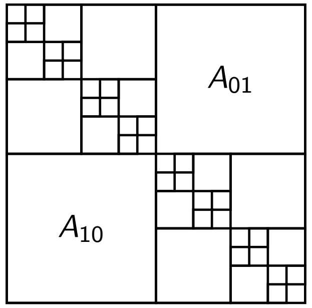
$A'_{01} \leftarrow A'_{00} A_{01} \quad A'_{10} \leftarrow A_{10} A'_{00} \quad A'_{11} \leftarrow A_{11} \oplus A_{10} A'_{00} A_{01}$

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

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

Block generic Gaussian elimination in disguise

Sequential work $O(n^3)$



Parallel graph algorithms
Algebraic path problem

Parallel algebraic path computation

Similar to LU decomposition by block generic Gaussian elimination

Te recursion tree is unfolded depth-first

Recursion levels 0 to $\alpha \log p$: block Floyd–Warshall using parallel matrix multiplication

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

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

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

Parallel graph algorithms
Algebraic path problem

Parallel algebraic path computation (contd.)

In particular:

$\alpha = 1/2$

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

Cf. 2D grid

$\alpha = 2/3$

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

Cf. matrix multiplication

Parallel graph algorithms
All-pairs shortest paths

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

	S	$\oplus$	$\boxed{0}$	$\odot$	$\boxed{1}$
tropical	$\mathbb{R}_{\ge 0} \cup \{+\infty\}$	min	$+\infty$	$+$	0

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

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

Parallel graph algorithms
All-pairs shortest paths

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

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

$A_{ij} \ge 0 \qquad A_{ii} = \boxed{1} = 0 \qquad 0 \le i, j < n$

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

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

A^k_{ij} = length of the shortest path $i \rightsquigarrow j$ of size $\le k$

A^*_{ij} = length of the shortest path $i \rightsquigarrow j$ (of any size)

The all-pairs shortest paths problem:

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

Parallel graph algorithms

All-pairs shortest paths

Dijkstra's algorithm [Dijkstra: 1959]

Computes single-source shortest paths from fixed source (say, node 0)

Ranks all nodes by distance from node 0: nearest, second nearest, etc.

Every time a node i has been ranked: replace each weight A_{0j}, j unranked, with $A_{0i} + A_{ij}$, if that gives a shortcut from 0 to j

Assign the next rank to the unranked node closest to node 0 and repeat

It is essential that the edge lengths are nonnegative

Sequential work $O(n^2)$

All-pairs shortest paths: multi-Dijkstra, i.e. running Dijkstra's algorithm independently from every node as a source

Sequential work $O(n^3)$

Parallel graph algorithms

All-pairs shortest paths

Parallel all-pairs shortest paths by multi-Dijkstra

Every processor

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

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

Parallel graph algorithms

All-pairs shortest paths

Parallel all-pairs shortest paths: summary so far

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

Floyd–Warshall, $\alpha = 2/3$	$\boxed{comm\ O(n^2/p^{2/3})}$	$\boxed{sync\ O(p^{2/3})}$
Floyd–Warshall, $\alpha = 1/2$	$\boxed{comm\ O(n^2/p^{1/2})}$	$\boxed{sync\ O(p^{1/2})}$
Multi-Dijkstra	$\boxed{comm\ O(n^2)}$	$\boxed{sync\ O(1)}$
Coming next	$\boxed{comm\ O(n^2/p^{2/3})}$	$\boxed{sync\ O(\log p)}$

Parallel graph algorithms

All-pairs shortest paths

Path doubling

Compute A, A^2, $A^4 = (A^2)^2$, $A^8 = (A^4)^2$, ..., $A^n = A^*$

Overall, $\log n$ rounds of matrix $\odot$-multiplication: looks promising...

Sequential work $O(n^3 \log n)$: not work-optimal!

Parallel graph algorithms
All-pairs shortest paths

Selective path doubling

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

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

Let $A^{\leq k}_{ij}$ = length of the shortest path $i \rightsquigarrow j$ of size $\leq k$

Let $A^{=k}_{ij}$ = length of the shortest path $i \rightsquigarrow j$ 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-$\boxed{0}$ 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-$\boxed{0}$ elements.

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

Parallel graph algorithms
All-pairs shortest paths

Selective path doubling (contd.)

Compute A, $A^{\leq \frac{3}{2}+\cdots}$, $A^{\leq (\frac{3}{2})^2+\cdots}$, ..., $A^{\leq n} = A^*$

Overall, $\leq \log_{3/2} n$ rounds of sparse-by-dense matrix $\odot$-multiplication

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

Parallel graph algorithms
All-pairs shortest paths

Parallel all-pairs shortest paths by selective path doubling

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

Consider $A^{=0}, \ldots, A^{=p}$. The total number of non-$\boxed{0}$ 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-$\boxed{0}$ elements.

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

All processors compute $(A^{=p})^*$ by parallel multi-Dijkstra, and then $(A^{=p})^* \odot A^{\leq p+\cdots} = A^*$ by parallel matrix $\odot$-multiplication

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

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

Parallel graph algorithms
All-pairs shortest paths

Parallel all-pairs shortest paths by selective path doubling (contd.)

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

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

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

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

Consider $A^{=0}, \ldots, A^{=\frac{p}{2}}$ and $A^{=(p)-\frac{p}{2}}, \ldots, A^{=(p)}$. The total number of non-$\boxed{0}$ 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-$\boxed{0}$ elements.

Parallel all-pairs shortest paths by selective path doubling (contd.)

Every processor

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

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

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