Advanced Topics in Algorithms

Efficient Parallel Algorithms

Alexander Tiskin

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

1 Computation by circuits
2 Parallel computation models
3 Basic parallel algorithms
4 Further parallel algorithms
5 Parallel matrix algorithms
6 Parallel graph algorithms

1 Computation by circuits
2 Parallel computation models
3 Basic parallel algorithms
4 Further parallel algorithms
5 Parallel matrix algorithms
6 Parallel graph algorithms

Computation by circuits

Computation models and algorithms

Model: abstraction of reality allowing qualitative and quantitative reasoning

Examples:

- atom
- biological cell
- galaxy
- Kepler's universe
- Newton's universe
- Einstein's universe
- ...

Computation by circuits
Computation models and algorithms

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

Examples:

- scales+weights (for "counterfeit coin" problems)
- Turing machine
- von Neumann machine ("ordinary computer")
- JVM
- quantum computer
- ...

Computation by circuits
Computation models and algorithms

Computation: input → (computation steps) → output

Algorithm: a finite description of a (usually infinite) set of computations on different inputs

Assumes a specific computation model and input/output encoding

Algorithm's running time (worst-case) $T : \mathbb{N} \to \mathbb{N}$

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

Similarly for other resources (e.g. memory, communication)

Computation by circuits
Computation models and algorithms

$T(n)$ is usually analysed asymptotically:

- up to a constant factor
- for sufficiently large n

$f(n) \geq 0 \quad n \to \infty$

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

Computation by circuits
Computation models and algorithms

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

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

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

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

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

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

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

Overtaking point depends on the constant!

Exercise: $\exists n_0 : \forall c : \forall n \geq n_0 : g(n) \leq c \cdot f(n)$ — what does this say?

Computation by circuits
Computation models and algorithms

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

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

$g = \Omega(f)$ iff $f = O(g)$ $\qquad$ $g = \omega(f)$ iff $f = o(g)$

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

$g = \Theta(f)$ iff $g = O(f)$ and $g = \Omega(f)$

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

Note: the "equality" in $g = O(f)$ is actually set membership. Sometimes written $g \in O(f)$, similarly for Ω, etc.

Computation by circuits
Computation models and algorithms

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

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

Proof: for all n, we have

$\max(f(n) + g(n)) \leq f(n) + g(n) \leq 2\max(f(n) + g(n))$ $\square$

Computation by circuits
Computation models and algorithms

Example usage: sorting an array of size n

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

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

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

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

Computation by circuits
Computation models and algorithms

Example usage: multiplying $n \times n$ matrices

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

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

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

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

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

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

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:

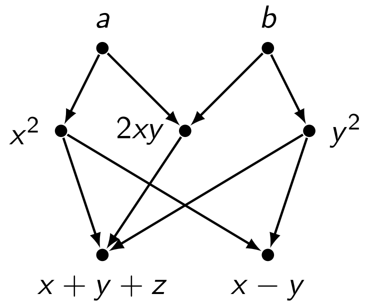
- impossible in a standard (or even quantum) model
- can be added to the standard model as an oracle, to create a more powerful model

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

Models oblivious computation: control sequence independent of the input

Computation on varying number of inputs: an (infinite) circuit family

May or may not admit a finite description (= algorithm)

Computation by circuits
The circuit model

In a circuit family, node indegree/outdegree may be bounded (by a constant), or unbounded: e.g. two-argument vs n-argument sum

Elementary operations:

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

size = number of nodes

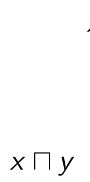
depth = max path length from input to output

Other uses of circuits:

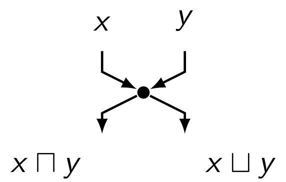
- arbitrary (non-oblivious) computation can be thought of as a circuit that is not given in advance, but revealed gradually
- timed circuits with feedback: systolic arrays

Computation by circuits
The comparison network model

A comparison network is a circuit of comparator nodes



denotes

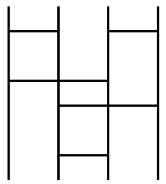


$\sqcap = \min$

$\sqcup = \max$

Input/output: sequences of equal length, taken from a totally ordered set

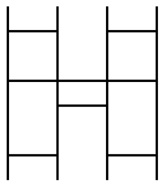
Examples:



$n = 4$

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 finitely described family of sorting (or merging) networks is equivalent to an oblivious sorting (or merging) algorithm

The network's size/depth determine the algorithm's sequential/parallel complexity

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

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

What is the complexity of oblivious sorting?

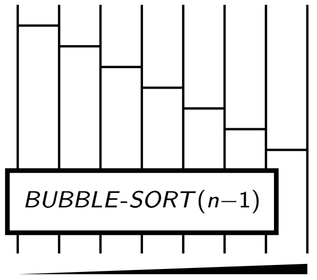
Computation by circuits

Naive sorting networks

BUBBLE-SORT(n)

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

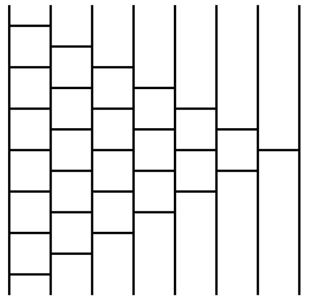
depth $2n - 3 = O(n)$



BUBBLE-SORT(8)

size 28

depth 13



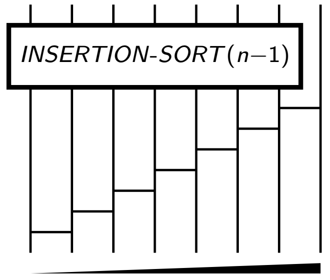
Computation by circuits

Naive sorting networks

INSERTION-SORT(n)

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

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

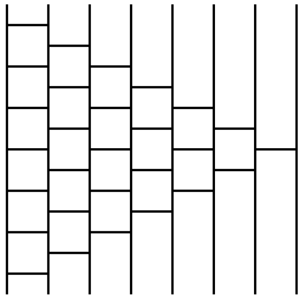


INSERTION-SORT(8)

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?

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

Odd-even merging

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

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

$size(OEM(n', n'')) \le 2 \cdot size(OEM(n'/2, n''/2)) + O(n) = O(n \log n)$

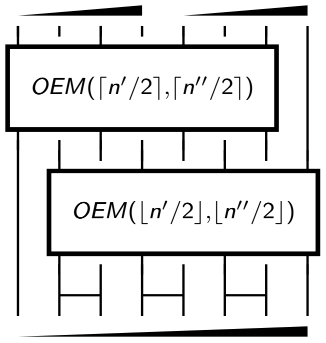
$depth(OEM(n', n'')) \le depth(OEM(n'/2, n''/2)) + 1 = O(\log n)$

Computation by circuits
Efficient merging and sorting networks

$OEM(n', n'')$

size $O(n \log n)$

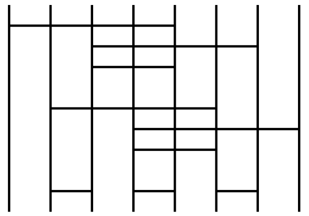
depth $O(\log n)$

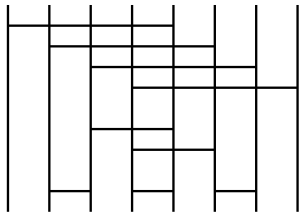

$OEM(\lceil n'/2\rceil, \lceil n''/2\rceil)$
$OEM(\lfloor n'/2\rfloor, \lfloor n''/2\rfloor)$

$OEM(4, 4)$

size 9

depth 3





Computation by circuits
Efficient merging and sorting networks

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

Induction base: trivial (2 inputs, 1 comparator)

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

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

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

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

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

The final stage of comparators corrects the wrong pair

$\langle 0^k 11\dots\rangle, \langle 0^l 11\dots\rangle \mapsto \langle 0^{k+l} 11\dots\rangle$ □

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 by recursion:

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

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

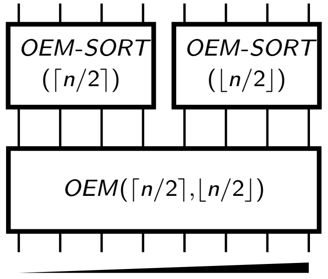
$$depth(OEM\text{-}SORT(n)) \leq depth(OEM\text{-}SORT(n/2)) + depth(OEM(n/2, n/2)) = depth(OEM\text{-}SORT(n/2)) + O(\log n) = O\big((\log n)^2\big)$$

Computation by circuits
Efficient merging and sorting networks

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

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

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

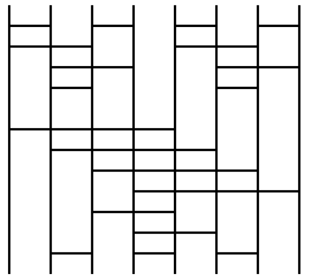


$OEM\text{-}SORT(\lceil n/2 \rceil)$
$OEM\text{-}SORT(\lfloor n/2 \rfloor)$
$OEM(\lceil n/2 \rceil, \lfloor n/2 \rfloor)$

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

size 19

depth 6



Computation by circuits
Efficient merging and sorting networks

A bitonic sequence: $\langle x_1 \geq \dots \geq x_m \leq \dots \leq x_n \rangle$ $\quad 1 \leq m \leq n$

Bitonic merging: sorting a bitonic sequence

When $n = 1$ we are done, otherwise by recursion:

- sort bitonic $\langle x_1, x_3, \dots \rangle \mapsto \langle u_1 \leq u_2 \leq \dots \leq u_{\lceil n/2 \rceil} \rangle$
- sort bitonic $\langle x_2, x_4, \dots \rangle \mapsto \langle v_1 \leq v_2 \leq \dots \leq v_{\lfloor n/2 \rfloor} \rangle$
- compare pairwise: (u_1, v_1), (u_2, v_2), ...

Exercise: prove correctness (by zero-one principle)

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

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

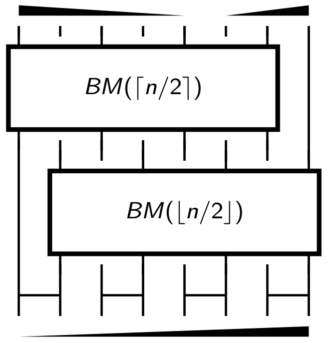
$size(BM(n)) = O(n \log n) \quad depth(BM(n)) = O(\log n)$

Computation by circuits
Efficient merging and sorting networks

$BM(n)$

size $O(n \log n)$

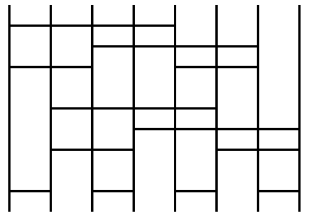
depth $O(\log n)$

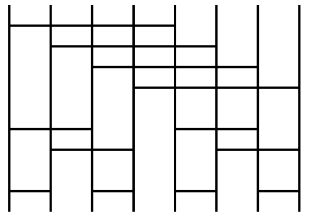


$BM(8)$

size 12

depth 3





Computation by circuits
Efficient merging and sorting networks

Bitonic merge sorting [Batcher: 1968]

When $n = 1$ we are done, otherwise by recursion:

- 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
- sort $\langle x_{\lceil n/2 \rceil+1}, \ldots, x_n \rangle \mapsto \langle y_{\lceil n/2 \rceil+1} \leq \cdots \leq y_n \rangle$
- 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

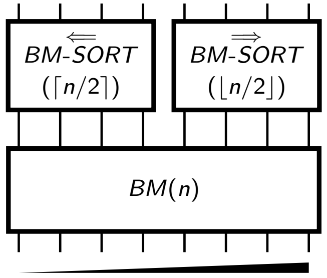
$size(BM\text{-}SORT(n)) = O\big(n(\log n)^2\big)$
$depth(BM\text{-}SORT(n)) = O\big((\log n)^2\big)$

Computation by circuits
Efficient merging and sorting networks

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

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

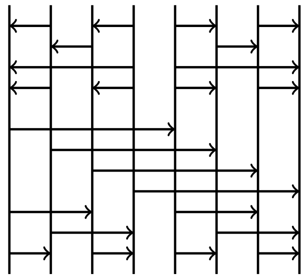
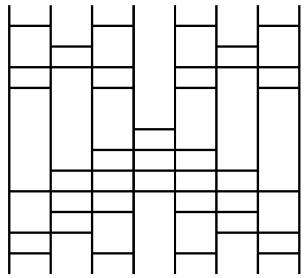
depth $O\big((\log n)^2\big)$


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

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

size 24

depth 6

Computation by circuits
Efficient merging and sorting networks

Both $OEM\text{-}SORT$ and $BM\text{-}SORT$ 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

1 Computation by circuits

2 Parallel computation models

3 Basic parallel algorithms

4 Further parallel algorithms

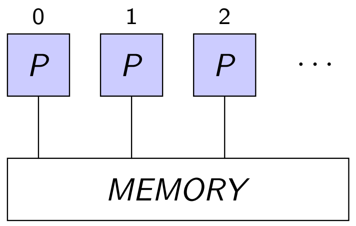
5 Parallel matrix algorithms

6 Parallel graph algorithms

Parallel computation models
The PRAM model

Parallel Random Access Machine (PRAM) [Fortune, Wylie: 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

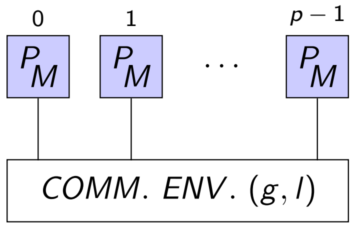
E.g. a linear system solver: $O\big((\log n)^2\big)$ steps using n^4 processors :-O

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



Goals: scalability, portability, predictability

Contains

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

Parallel computation models
The BSP model

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

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

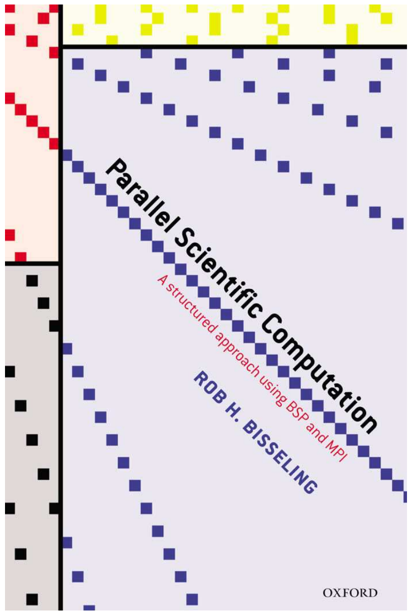
Communication network parameters:

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

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

Network efficiency grows slower than processor efficiency and costs more energy: $g, l \gg 1$. E.g. for Cray T3E: $p = 64$, $g \approx 78$, $l \approx 1825$

Parallel computation models
The BSP model

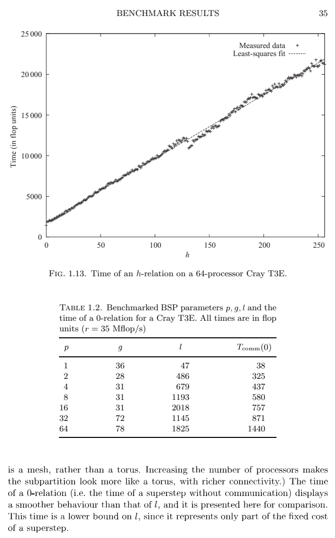


Parallel Scientific Computation

A structured approach using BSP and MPI

ROB H. BISSELING

OXFORD



BENCHMARK RESULTS 35

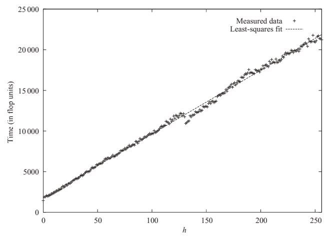


Fig. 1.13. Time of an h-relation on a 64-processor Cray T3E.

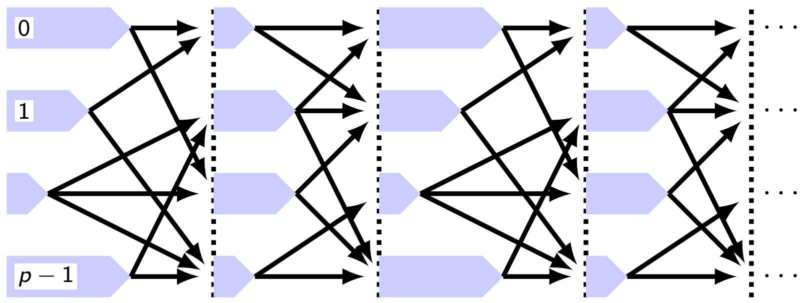
Table 1.2. Benchmarked BSP parameters p, g, l and the time of a 0-relation for a Cray T3E. All times are in flop units ($r = 35$ Mflop/s)

p	g	l	$T_{\text{comm}}(0)$
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:

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

Parallel computation models
The BSP model

Conventions:

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

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

Main principles:

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

Data locality exploited, network locality ignored!

Parallel computation models
The BSP model

BSP software: industrial projects

Google's Pregel	[2010]
Apache Spark (`hama.apache.org`)	[2010]
Apache Giraph (`giraph.apache.org`)	[2011]

BSP software: research projects

Oxford BSP (`www.bsp-worldwide.org/implmnts/oxtool`)	[1998]
Paderborn PUB (`www2.cs.uni-paderborn.de/~pub`)	[1998]
BSML (`traclifo.univ-orleans.fr/BSML`)	[1998]
BSPonMPI (`bsponmpi.sourceforge.net`)	[2006]
Multicore BSP (`www.multicorebsp.com`)	[2011]
Epiphany BSP (`www.codu.in/ebsp`)	[2015]
Petuum (`petuum.org`)	[2015]

Parallel computation models
Standard communication patterns

Broadcasting:

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

Combining (complementary to broadcasting):

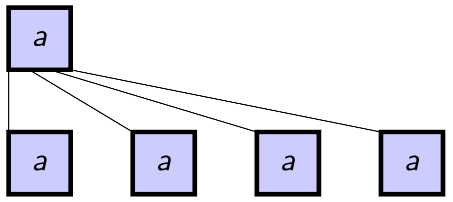
- initially, every processor r holds a value $a^{(r)}$, $0 \leq r < p$
- at the end, one designated processor must hold $a^{(0)} \bullet \cdots \bullet a^{(p-1)}$ for a given associative operator $\bullet$ (e.g. $+$)

By symmetry, we only need to consider broadcasting

Parallel computation models
Standard communication patterns

Direct broadcast:

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



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

Parallel computation models
Standard communication patterns

From now on, cost components will be $\boxed{\text{shaded}}$ when they are optimal

More precisely, $\boxed{cost = O(f(n,p))}$ means that for a given problem

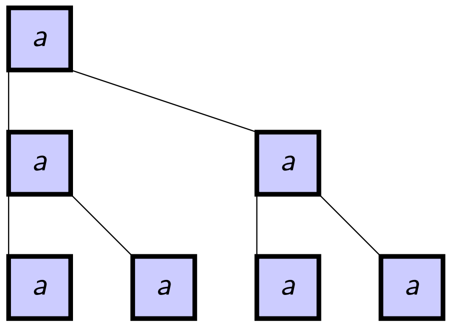
- $cost = O(f(n,p))$ for the given algorithm on all inputs
- $cost = \Omega(f(n,p))$ for any algorithm on some inputs

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

Parallel computation models
Standard communication patterns

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

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

Parallel computation models
Standard communication patterns

Array broadcasting:

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

Combining (complementary to broadcasting):

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

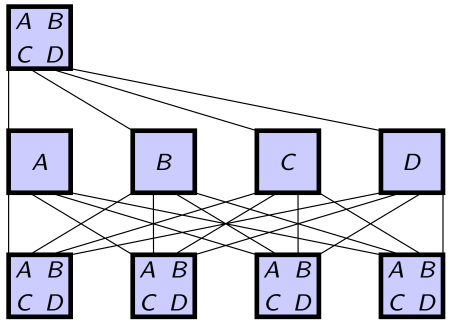
Effectively, n independent instances of broadcasting/combining

By symmetry, we only need to consider broadcasting

Parallel computation models
Standard communication patterns

Two-phase array broadcast:

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



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

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

Concurrent reading: one processor reads then broadcasts

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

Parallel computation models
Network routing

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

Has to be implemented on concrete networks

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

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

Low degree — easier to implement

Low diameter — more efficient

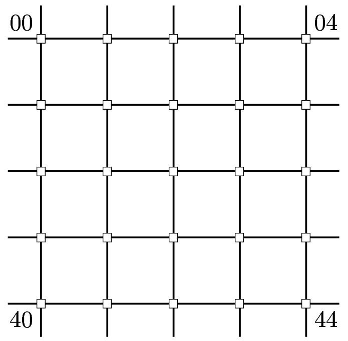
Parallel computation models
Network routing

2D array network

$p = q^2$ processors

degree 4

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



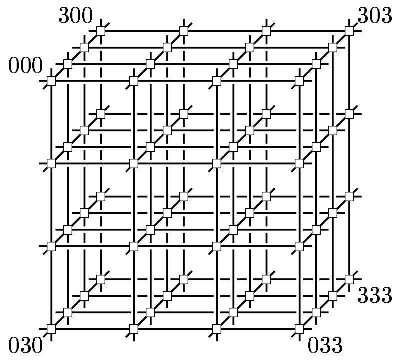
Parallel computation models
Network routing

3D array network

$p = q^3$ processors

degree 6

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



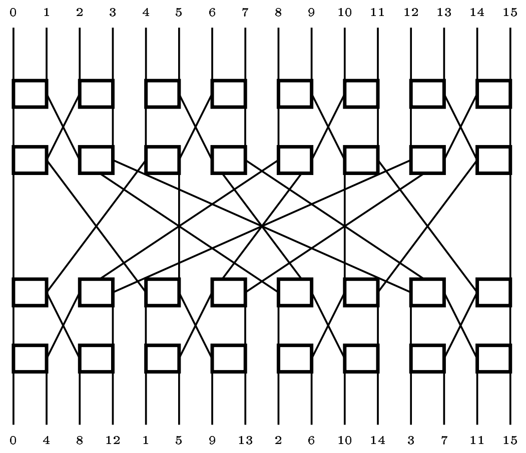
Parallel computation models
Network routing

Butterfly network

$p = q \log q$ processors

degree 4

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



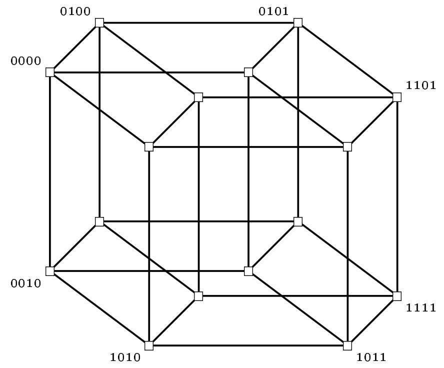
Parallel computation models
Network routing

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 $\Omega(\mathit{diameter})$ steps

Any oblivious routing method may be forced to make $\Omega(p^{1/2}/\mathit{degree})$ 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	$O\big((\log p)^2\big)$	$O\big((\log p)^2\big)$
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 $O(diameter)$ 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

1 Computation by circuits

2 Parallel computation models

3 Basic parallel algorithms

4 Further parallel algorithms

5 Parallel matrix algorithms

6 Parallel graph algorithms

Basic parallel algorithms
Balanced tree and prefix sums

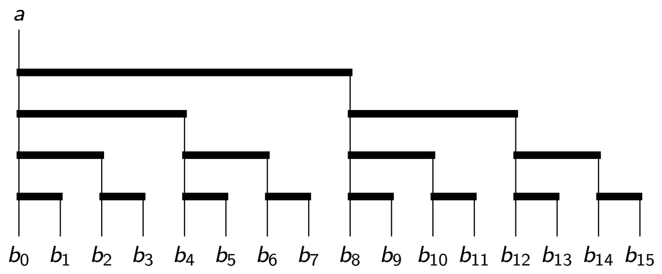
The balanced binary tree dag

tree(n)

1 input, n outputs

size $n - 1$

depth $\log n$



a
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}

A generalisation of broadcasting/combining

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

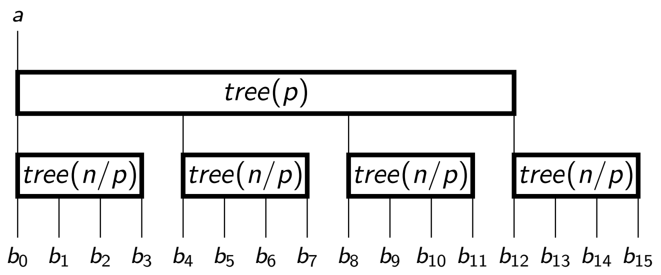
Sequential work $O(n)$

From now on, we always assume that a problem's input/output is stored in the external memory; reading/writing will also refer to the external memory

Basic parallel algorithms
Balanced tree and prefix sums

Parallel balanced tree computation

tree(n)



a
$tree(p)$
$tree(n/p)$
$tree(n/p)$
$tree(n/p)$
$tree(n/p)$
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}

Partition *tree*(n) into

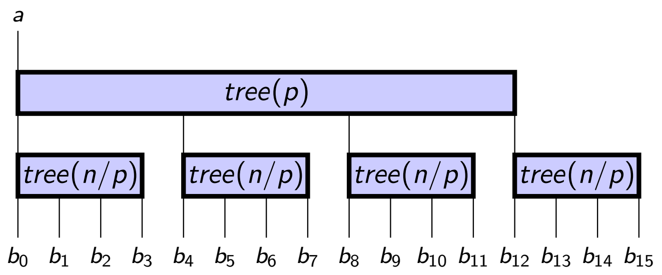
- one top block, isomorphic to *tree*(p)
- a bottom layer of p blocks, each isomorphic to *tree*(n/p)

Basic parallel algorithms
Balanced tree and prefix sums

Parallel balanced tree computation (contd.)

tree(n)

$(p = 4)$



a
$tree(p)$
$tree(n/p)$
$tree(n/p)$
$tree(n/p)$
$tree(n/p)$
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}

For top-down computation:

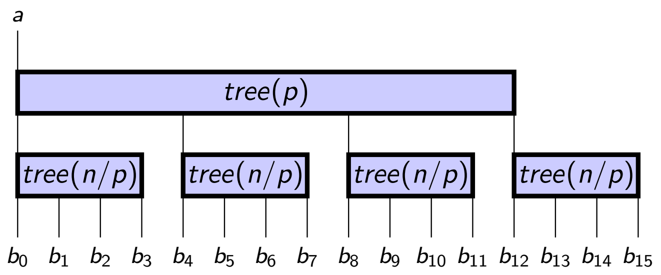
- a designated processor is assigned the top block; the processor reads block's input, computes the block, writes block's p outputs
- every processor is assigned a different bottom block; each processor reads block's input, computes the block, writes n/p block's outputs

For bottom-up computation, reverse the steps

Basic parallel algorithms
Balanced tree and prefix sums

Parallel balanced tree computation (contd.)

tree(n)



a
$tree(p)$
$tree(n/p)$
$tree(n/p)$
$tree(n/p)$
$tree(n/p)$
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}

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

Slackness $n \geq p^2$

Basic parallel algorithms
Balanced tree and prefix sums

The described parallel balanced tree algorithm is fully optimal:

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

For other problems, we may not be so lucky to get a fully-optimal BSP algorithm. However, we are typically interested in algorithms that are optimal in *comp* (under reasonable assumptions).

Optimality in *comm* and *sync* is considered subject to optimality in *comp*

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 operator, and assume

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

Examples: numerical $+$, $\cdot$, min, max, Boolean $\wedge$, $\vee$, ...

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

Basic parallel algorithms
Balanced tree and prefix sums

The prefix sums problem

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

$b_{-1} = \epsilon \quad b_i = a_i \bullet b_{i-1} \quad 0 \leq i < n$

$b_0 = a_0$

$b_1 = a_0 \bullet a_1$

$b_2 = a_0 \bullet a_1 \bullet a_2$

...

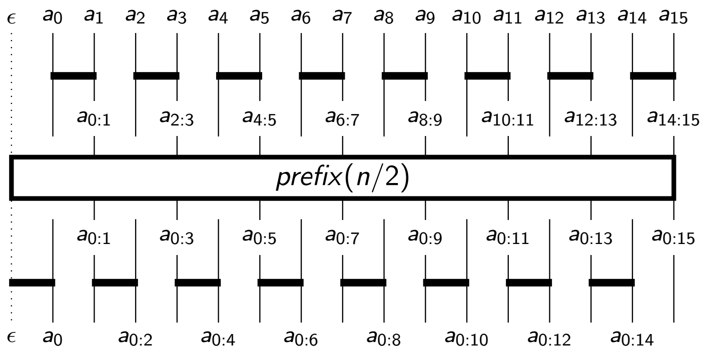
$b_{n-1} = a_0 \bullet a_1 \bullet \cdots \bullet a_{n-1}$

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

Basic parallel algorithms
Balanced tree and prefix sums

The prefix circuit [Ladner, Fischer: 1980]

prefix(n)



ϵ 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$(n/2)$
$a_{0:1}$ $a_{0:3}$ $a_{0:5}$ $a_{0:7}$ $a_{0:9}$ $a_{0:11}$ $a_{0:13}$ $a_{0:15}$
ϵ a_0 $a_{0:2}$ $a_{0:4}$ $a_{0:6}$ $a_{0:8}$ $a_{0:10}$ $a_{0:12}$ $a_{0:14}$

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

The underlying dag is called the prefix dag

Basic parallel algorithms
Balanced tree and prefix sums

The prefix circuit (contd.)

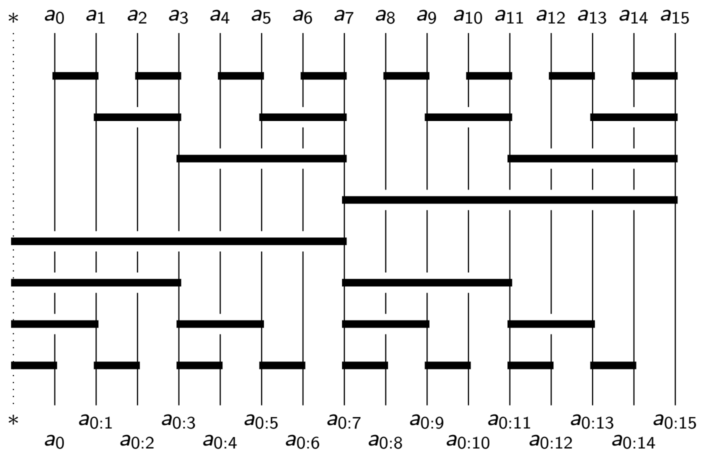
prefix(n)

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

Dag *prefix*(n) consists of

- a top subtree similar to bottom-up *tree*(n)
- transfer of values from top subtree to bottom subtree
- a bottom subtree similar to top-down *tree*(n)

Both trees can be computed by the previous algorithm

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

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

Slackness $n \geq p^2$

Basic parallel algorithms
Balanced tree and prefix sums

Application: generic first-order linear recurrence

$a = [a_0, \ldots, a_{n-1}] \quad b = [b_0, \ldots, b_{n-1}]$

$c_{-1} = 0 \quad c_i = a_i + b_i \cdot c_{i-1} \quad 0 \leq i < n$

$c_0 = a_0$

$c_1 = a_1 + b_1 \cdot c_0$

$c_2 = a_2 + b_2 \cdot c_1$

$\ldots$

$c_{n-1} = a_{n-1} + b_{n-1} \cdot c_{n-2}$

Basic parallel algorithms
Balanced tree and prefix sums

Application: generic first-order linear recurrence (contd.)

$\begin{bmatrix} 1 \\ c_i \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ a_i & b_i \end{bmatrix} \begin{bmatrix} 1 \\ c_{i-1} \end{bmatrix} \quad 0 \leq i < n \qquad \text{Let } C_i = \begin{bmatrix} 1 \\ c_i \end{bmatrix}, A_i = \begin{bmatrix} 1 & 0 \\ a_i & b_i \end{bmatrix}$

$C_0 = A_0 C_{-1}$

$C_1 = A_1 A_0 C_{-1}$

$C_2 = A_2 A_1 A_0 C_{-1}$

$\ldots$

$C_{n-1} = A_{n-1} \ldots A_1 A_0 C_{-1}$

Computing the generic first-order linear recurrence:

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

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

Basic parallel algorithms
Balanced tree and prefix sums

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

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

Examples of suitable $\oplus$ and $\odot$:

- numerical $+$ and $\cdot$
- numerical min and $+$; numerical max and $+$
- Boolean $\wedge$ and $\vee$; Boolean $\vee$ and $\wedge$

Basic parallel algorithms
Balanced tree and prefix sums

Application: polynomial evaluation

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

$y = a_0 + a_1 \cdot x + \ldots + a_{n-2} \cdot x^{n-2} + a_{n-1} \cdot x^{n-1}$

Evaluating the polynomial:

- $1, x, x^2, \ldots, x^{n-1}$ by prefix product with operator $\cdot$
- sum y by bottom-up balanced binary tree with operator $+$

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

Basic parallel algorithms
Balanced tree and prefix sums

Application: polynomial evaluation by Horner's rule

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

$y = a_0 + a_1 \cdot x + \ldots + a_{n-2} \cdot x^{n-2} + a_{n-1} \cdot x^{n-1}$

$y = a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot (\ldots + x \cdot a_{n-1}) \ldots))$

Evaluating the polynomial:

- generic first-order linear recurrence over $[d_{n-1}, \ldots, d_0]$ and $[x, \ldots, x]$

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

Basic parallel algorithms
Balanced tree and prefix sums

Application: binary addition via Boolean logic

$x + y = z$

Inputs x, y and output z represented in binary as bit arrays

$x = [x_{n-1}, \ldots, x_0] \quad y = [y_{n-1}, \ldots, y_0] \quad z = [z_n, z_{n-1}, \ldots, z_0]$

The binary adder problem: given x, y, compute z using bitwise $\wedge$ ("and"), $\vee$ ("or"), $\oplus$ ("xor")

Let $c = [c_{n-1}, \ldots, c_0]$, 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

$x + y = z$

Define bit arrays $u = [u_{n-1}, \ldots, u_0]$, $v = [v_{n-1}, \ldots, v_0]$

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

Arrays u, v can be computed in size $O(n)$, depth $O(1)$

We then compute

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

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

Equivalent to a ripple-carry adder. Can we do better?

Basic parallel algorithms
Balanced tree and prefix sums

$c_{-1} = 0 \quad c_i = u_i \vee (v_i \wedge c_{i-1})$

Generic first-order linear recurrence with inputs u, v and operators $\vee$, $\wedge$

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

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

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

Equivalent to a carry-lookahead adder

Basic parallel algorithms
Fast Fourier Transform and the butterfly dag

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

Applications: digital signal processing (amplitude vs frequency); polynomial multiplication; big integer multiplication

Basic parallel algorithms
Fast Fourier Transform and the butterfly dag

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

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

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

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

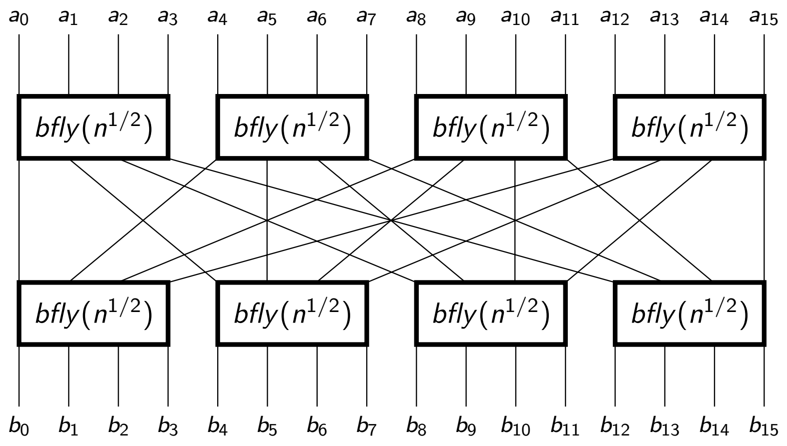
$size_{FFT}(n) = O(n) + 2 \cdot n^{1/2} \cdot size_{FFT}(n^{1/2}) = O(1 \cdot n \cdot 1 + 2 \cdot n^{1/2} \cdot n^{1/2} + 4 \cdot n^{3/4} \cdot n^{1/4} + \cdots + \log n \cdot n \cdot 1) = O(n + 2n + 4n + \cdots + \log n \cdot n) = O(n \log n)$

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

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.)

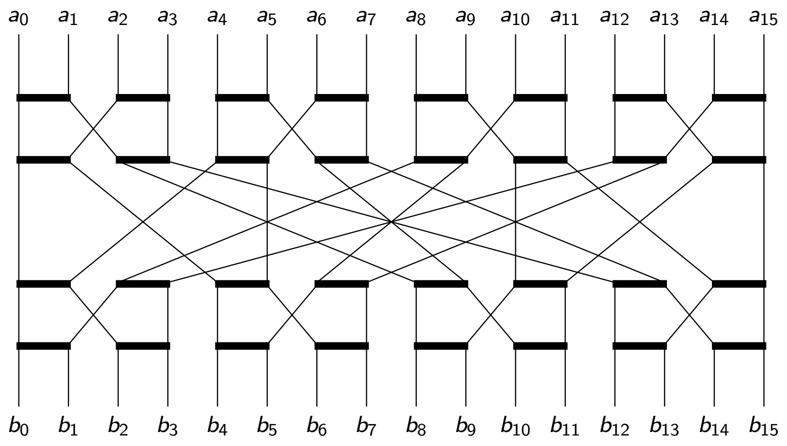
bfly(n)

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}

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):

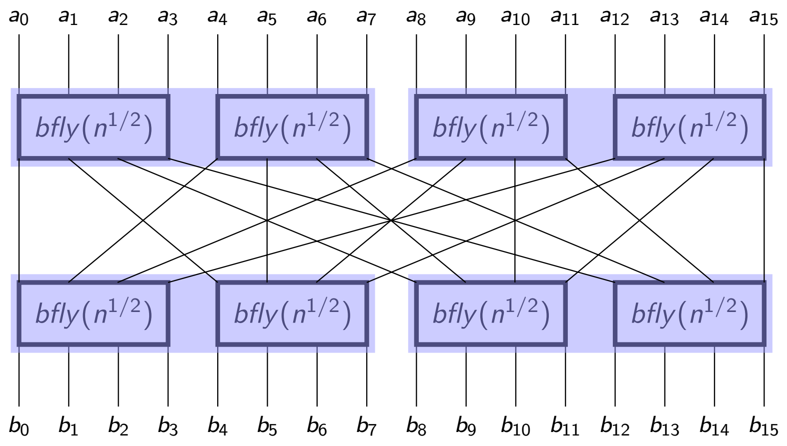
- every processor computes $n^{1/2}/p$ blocks from the top layer
- every processor computes $n^{1/2}/p$ blocks from the bottom layer

In each layer, the processor reads the total of n/p inputs, performs $O(n \log n/p)$ computation, then writes the total of n/p outputs

Basic parallel algorithms
Fast Fourier Transform and the butterfly dag

Parallel butterfly computation (contd.)

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}

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

Slackness $n \geq p^2$

Basic parallel algorithms
Ordered grid

The ordered 2D grid dag

grid$_2(n)$

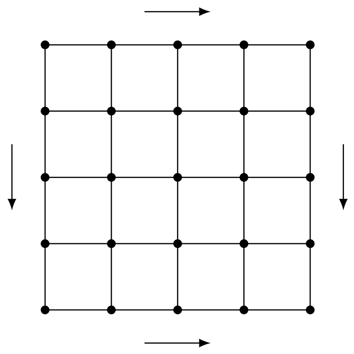
nodes arranged in an $n \times n$ grid

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

Sequential work $O(n^2)$

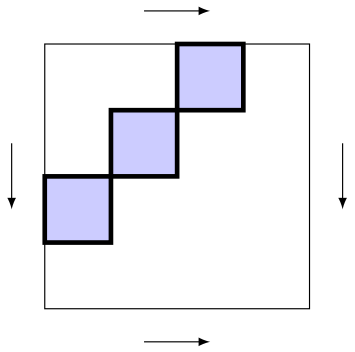
Basic parallel algorithms
Ordered grid

Parallel ordered 2D grid computation

grid$_2(n)$

Partition into a $p \times p$ grid of blocks, each isomorphic to *grid*$_2(n/p)$

Arrange blocks as $2p - 1$ anti-diagonal layers: $\leq p$ 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 block assigned to a different processor (some processors idle)
- the processor reads the $2n/p$ block inputs, computes the block, and writes back the $2n/p$ block outputs

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

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

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

Slackness $n \geq p$

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" $\quad b =$ "DESIGN" $\quad LCS(a, b) =$ "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 [Wagner, Fischer: 1974]

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

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

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

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

	*	D	E	F	I	N	E
*	0	0	0	0	0	0	0
D	0	1	1	1	1	1	1
E	0	1	2	2	2	2	2
S	0	1	2	2	2	2	2
I	0	1	2	2	3	3	3
G	0	1	2	2	3	3	3
N	0	1	2	2	3	4	4

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

$LCS(a, b)$ can be "traced back" through the table at no extra asymptotic cost

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

Basic parallel algorithms
Ordered grid

Parallel LCS computation

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

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

comm is not scalable (i.e. does not decrease with increasing p) :-(

Can scalable *comm* be achieved for the LCS problem?

Basic parallel algorithms
Ordered grid

Parallel LCS computation

Solve the more general semi-local LCS problem:

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

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

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

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

Sequential time still $O(n^2)$

Basic parallel algorithms
Ordered grid

Parallel LCS computation (cont.)

Communication vs synchronisation tradeoff

Parallelising normal $O(n \log n)$ seaweed multiplication: [Krusche, T: 2010]

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

Special seaweed multiplication [Krusche, T: 2007]

Sacrifices some *comp*, *comm* for *sync*

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

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

Basic parallel algorithms
Ordered grid

The ordered 3D grid dag

$grid_3(n)$

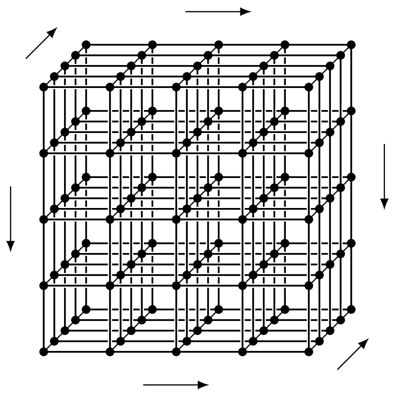
nodes arranged in an $n \times n \times n$ grid

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

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

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

size n^3 depth $3n - 2$



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

Sequential work $O(n^3)$

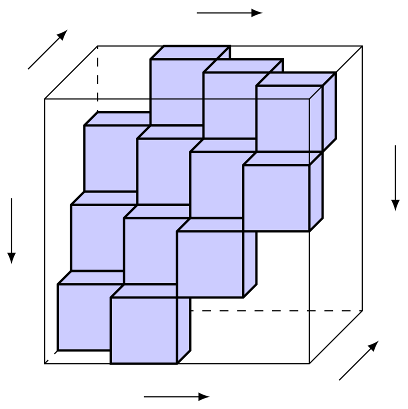
Basic parallel algorithms
Ordered grid

Parallel ordered 3D grid computation

$grid_3(n)$

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

Arrange blocks as $3p^{1/2} - 2$ anti-diagonal layers: $\leq p$ independent blocks in each layer



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

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

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

Basic parallel algorithms
Discussion

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

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

The goals:

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

The challenges:

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

1 Computation by circuits

2 Parallel computation models

3 Basic parallel algorithms

4 Further parallel algorithms

5 Parallel matrix algorithms

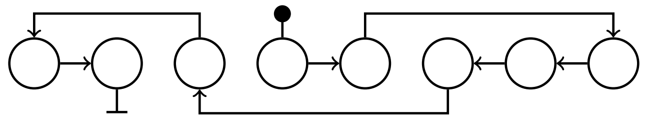
6 Parallel graph algorithms

Further parallel algorithms
List contraction and colouring

Linked list: array of n nodes

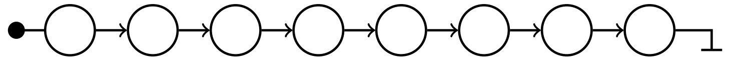
Each node contains data and a pointer to (= index of) successor node

Nodes may be placed in array in an arbitrary order



Logical structure linear: $head, succ(head), succ(succ(head)), \ldots$

- a pointer can be followed in time $O(1)$
- no global ranks/indexing/comparison



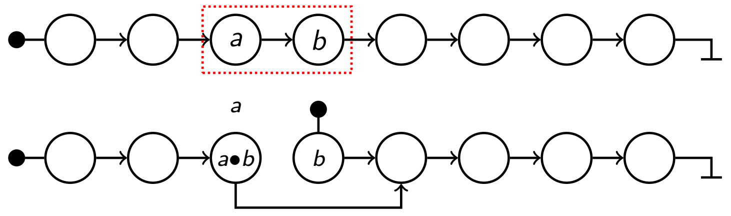
Further parallel algorithms
List contraction and colouring

Pointer jumping at node u

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

$v \leftarrow succ(u)$ $\quad$ $succ(u) \leftarrow succ(v)$

$a \leftarrow data(u)$ $\quad$ $b \leftarrow data(v)$ $\quad$ $data(u) \leftarrow a \bullet b$

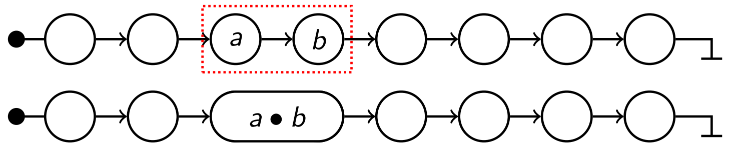


Pointer v and data a, b are kept, so that pointer jumping can be reversed:

$succ(u) \leftarrow v$ $\quad$ $data(u) \leftarrow a$ $\quad$ $data(v) \leftarrow b$

Further parallel algorithms
List contraction and colouring

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



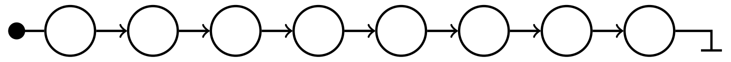
Data a, b are kept, so that node merging can be reversed

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

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

Further parallel algorithms
List contraction and colouring

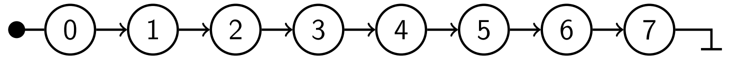
Application: list ranking



Node's rank: distance from *head*

$rank(head) = 0$, $rank(succ(head)) = 1$, ...

The list ranking problem: each node to hold its rank



Note the solution should be independent of the merging order

Further parallel algorithms
List contraction and colouring

Application: list ranking (contd.)

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

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

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

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

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

Further parallel algorithms
List contraction and colouring

Application: list ranking (contd.)

Expansion phase: each node holds

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

Fully contracted list: single node t holding

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

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

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

After full expansion: each node u holds

$l(u) = 1 \quad r(u) = rank(u)$

Further parallel algorithms
List contraction and colouring

Application: list prefix sums

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

$$\begin{tikzcd}
\bullet \arrow[r, dash] & a_0 \arrow[r] & a_1 \arrow[r] & a_2 \arrow[r] & a_3 \arrow[r] & a_4 \arrow[r] & a_5 \arrow[r] & a_6 \arrow[r] & a_7 \arrow[r, dash] & \perp
\end{tikzcd}$$

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

The list prefix sums problem: each node u to hold prefix sum $a_{0:rank(u)} = a_0 \bullet a_1 \bullet \cdots \bullet a_{rank(u)}$

$$\begin{tikzcd}
\bullet \arrow[r, dash] & a_0 \arrow[r] & a_{0:1} \arrow[r] & a_{0:2} \arrow[r] & a_{0:3} \arrow[r] & a_{0:4} \arrow[r] & a_{0:5} \arrow[r] & a_{0:6} \arrow[r] & a_{0:7} \arrow[r, dash] & \perp
\end{tikzcd}$$

Note the solution should be independent of the merging order

Further parallel algorithms
List contraction and colouring

Application: list prefix sums (contd.)

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

Contraction phase: each node u holds the $\bullet$-sum $l(u)$ corresponding sublist

Initially, $l(u) \leftarrow a_{rank(u)}$ for each node u

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

Fully contracted list: single node t with $l(t) = a_{0:n-1}$

Further parallel algorithms
List contraction and colouring

Application: list prefix sums (contd.)

Expansion phase: each node holds

- $\bullet$-sum $l(u)$ of corresponding sublist (as before)
- $\bullet$-sum $r(u)$ of all nodes before the sublist

Fully contracted list: single node t holding

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

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

$r(v) \leftarrow r(u) \quad r(w) \leftarrow r(v) \bullet l(v)$

After full expansion: each node u holds

$l(u) = a_{rank(u)} \quad r(u) = a_{0:rank(u)}$

Further parallel algorithms
List contraction and colouring

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

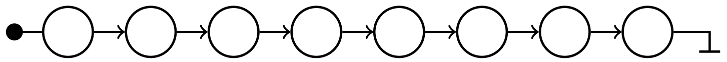
Sequential contraction: always merge *head* with *succ*(*head*), time $O(n)$

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

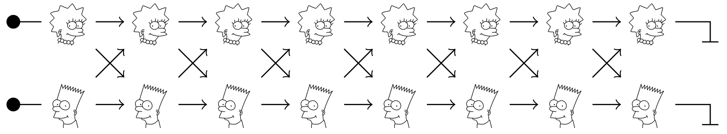
Therefore, we need either node splitting, or efficient symmetry breaking

Further parallel algorithms
List contraction and colouring

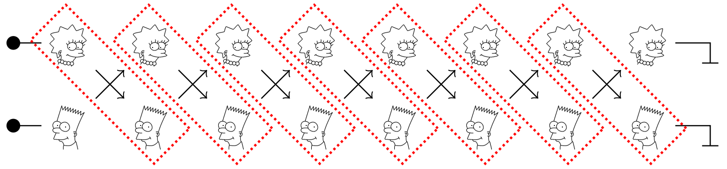
Wyllie's mating [Wylie: 1979]



Split every node, label copies "forward" and "backward"



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

In the first round, every processor

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

Subsequent rounds similar

Further parallel algorithms
List contraction and colouring

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

Parallel list contraction:

- perform $\log n$ rounds of Wyllie's mating, reducing original list to n fully contracted lists of size 1
- select one fully contracted list

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

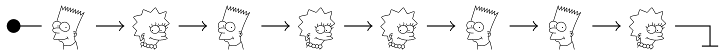
$\boxed{comp = O(\frac{n \log n}{p})}$ $\boxed{comm = O(\frac{n \log n}{p})}$ $\boxed{sync = O(\log n)}$ $n \geq p$

Further parallel algorithms
List contraction and colouring

Random mating [Miller, Reif: 1985]

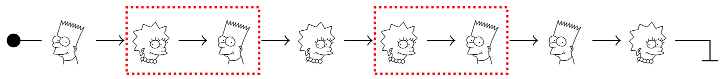
Label every node either "forward" or "backward"

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



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

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



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

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

Further parallel algorithms

List contraction and colouring

Parallel list contraction by random mating

In the first round, every processor

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

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

Further parallel algorithms

List contraction and colouring

Parallel list contraction by random mating (contd.)

Parallel list contraction:

- perform $\log_{16/15} p$ rounds of random mating, reducing original list to size $\frac{n}{p}$ whp
- a designated processor inputs the remaining list, contracts it sequentially

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

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

$n \geq p^2$

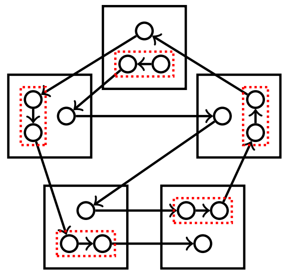
Further parallel algorithms

List contraction and colouring

Block mating

Will mate nodes deterministically

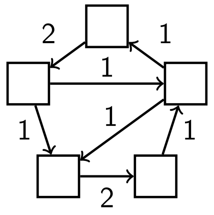
Contract local chains (if any)



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



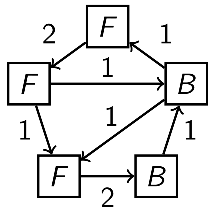
Further parallel algorithms

List contraction and colouring

Block mating (contd.)

Designated processor collects the distribution graph

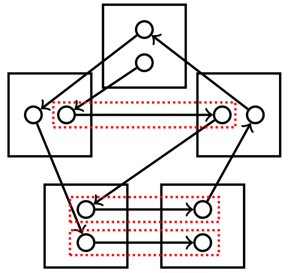
Label every supernode F ("forward") or B ("backward"), so that $\sum_{i \in F, j \in B} w(i,j) \geq \frac{1}{4} \cdot \sum_{i,j} w(i,j)$ by a sequential greedy algorithm



Distribute supernode labels to processors

By construction of supernode labelling, $\geq \frac{n}{2}$ nodes have mates

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



Further parallel algorithms
List contraction and colouring

Parallel list contraction by block mating

In the first round, every processor

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

The designated processor collects distribution graph, computes and distributes labels

Further parallel algorithms
List contraction and colouring

Parallel list contraction by block mating (contd.)

Continuing the first round, every processor

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

Subsequent rounds similar, on a list of decreasing size

Further parallel algorithms
List contraction and colouring

Parallel list contraction by block mating (contd.)

Parallel list contraction:

- perform $\log_{4/3} p$ rounds of block mating, reducing the original list to size n/p
- a designated processor collects the remaining list and contracts it sequentially

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

$\boxed{comp = O(n/p)}$ $\boxed{comm = O(n/p)}$ $\boxed{sync = O(\log p)}$ $n \geq p^4$

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 $\{0, \ldots, k-1\}$ to every node, so that in each pair of adjacent nodes, the two colours are different

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

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

Is list contraction really necessary for list k-colouring?

Can list k-colouring be done more efficiently?

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

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

Can this be extended to any $k \leq p$, e.g. $k = O(1)$?

Further parallel algorithms
List contraction and colouring

Deterministic coin tossing [Cole, Vishkin: 1986]

Given a k-colouring, $k > 6$

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

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

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

Model assumption: can find lowest nonzero bit in an integer in time $O(1)$

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

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

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

Further parallel algorithms
List contraction and colouring

Parallel list colouring by deterministic coin tossing

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

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

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

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

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

Further parallel algorithms
List contraction and colouring

Parallel list colouring by deterministic coin tossing (contd.)

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

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

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

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

Further parallel algorithms
List contraction and colouring

Parallel list colouring by deterministic coin tossing (contd.)

We have a 6-coloured, partially contacted list of size $O(n/\log^* p)$

- select node v as a pivot, if $col(pred(v)) > col(v) < col(succ(v))$; no two pivots are adjacent or further than 12 nodes apart
- re-colour all pivots in one colour
- from each pivot, 2-colour the next ≤ 12 non-pivots sequentially; we now have a 3-coloured list
- reverse the partial contraction, maintaining the 3-colouring

We have now 3-coloured the original list

$\boxed{comp = O(n/p)}$ $\boxed{comm = O(n/p)}$ $\boxed{sync = O(\log^* p)}$ $n \geq p^4$

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

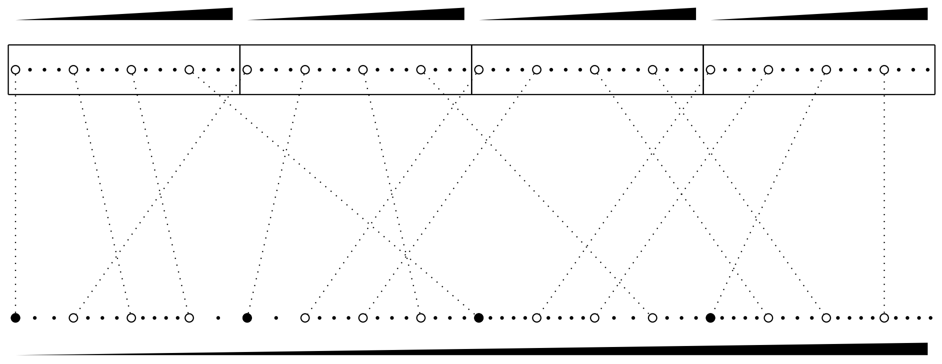
- reads a subarray of a of size n/p and sorts it sequentially
- selects from it p samples from index 0 at regular intervals n/p^2, defining p equal-sized, contiguous blocks in subarray

A designated processor

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

Further parallel algorithms
Sorting

Parallel sorting by regular sampling (contd.)



Further parallel algorithms
Sorting

Parallel sorting by regular sampling (contd.)

Every processor

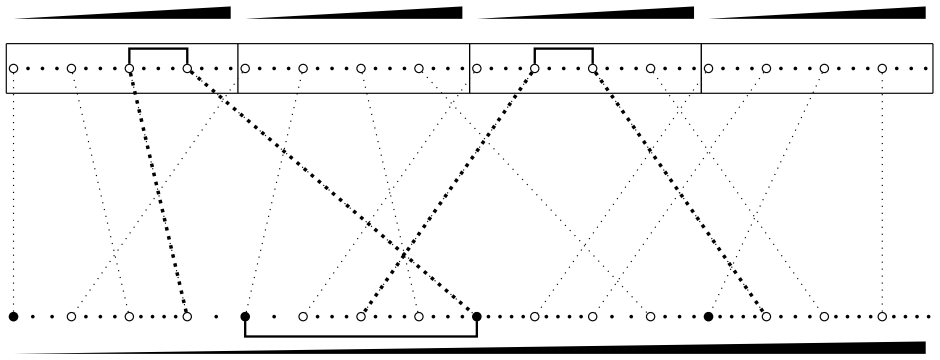
- receives the splitters and is assigned a bucket
- scans its subarray and sends each element to the appropriate bucket
- receives the elements of its bucket and sorts them sequentially
- writes the sorted bucket back to external memory

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

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

Further parallel algorithms
Sorting

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



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, if lower boundary of b is $\leq$ lower boundary of B
- high otherwise

A bucket may only intersect

- ≤ 1 low block per processor, hence $\leq p$ low blocks overall
- $\leq p$ high blocks overall

Therefore, bucket size $\leq (p + p) \cdot n/p^2 = 2n/p$ □

Further parallel algorithms
Selection

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

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

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

As with sorting, we assume the comparison model

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

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

Further parallel algorithms
Selection

Selection by median sampling [Blum+: 1973]

Proceed in rounds. In each round:

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

If $l = k$, return a_l; otherwise, eliminate elements on "wrong side" of l and set new target rank for next round:

- if $l < k$, eliminate all $a_i \leq a_l$; for next round $n \leftarrow n - l - 1$, $k \leftarrow k - l - 1$
- if $l > k$, eliminate all $a_i \geq a_l$; for next round $n \leftarrow l$, k unchanged

Further parallel algorithms
Selection

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

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

In at least $\frac{1}{2} \cdot \frac{n}{5}$ subarrays, subarray median $\leq a_l$

In every such subarray, three elements $\leq$ subarray median $\leq a_l$

Hence, at least $\frac{1}{2} \cdot \frac{3n}{5} = \frac{3n}{10}$ elements $\leq a_l$

Symmetrically, at least $\frac{3n}{10}$ elements $\geq a_l$

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

Data reduction rate is exponential

$T(n) \leq T\left(\frac{n}{5}\right) + T\left(n - \frac{3n}{10}\right) + O(n) = T\left(\frac{2n}{10}\right) + T\left(\frac{7n}{10}\right) + O(n)$, therefore $T(n) = O(n)$

Further parallel algorithms
Selection

Parallel selection by median sampling

In the first round, every processor

- reads a subarray of size n/p, selects the median

A designated processor

- collects all p subarray medians
- selects and broadcasts the median-of-medians

Every processor

- determines rank of median-of-medians in local subarray

Further parallel algorithms
Selection

Parallel selection by median sampling (contd.)

A designated processor

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

Every processor

- performs elimination on its subarray
- writes remaining elements

$\leq 3n/4$ elements remain overall in array a

Subsequent rounds similar, on an array of decreasing size, with target rank adjusted as necessary

Further parallel algorithms
Selection

Parallel selection by median sampling (contd.)

Parallel selection:

- perform $\log_{4/3} p$ rounds of median sampling and elimination, reducing original array to size n/p
- a designated processor collects the remaining array and performs selection sequentially

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

Further parallel algorithms
Selection

Parallel selection by regular sampling (generalised median sampling)

In the first round, every processor

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

A designated processor

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

Every processor

- determines rank of every splitter in local subarray

Further parallel algorithms
Selection

Selection by regular sampling (contd.)

A designated processor

- adds up local ranks to determine global rank of every splitter
- compares these against target rank to determine target bucket
- broadcasts info on target bucket

Every processor

- performs elimination on subarray, keeping elements of target bucket
- writes remaining elements

$\leq 2n/s$ elements remain overall in array a

Subsequent rounds similar, on an array of decreasing size, with target rank adjusted as necessary

Further parallel algorithms
Selection

Parallel selection by accelerated regular sampling

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

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

Parallel selection:

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

Technical details omitted

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

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]

$\boxed{sync = O(\log \log p)}$ [T: 2010]

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

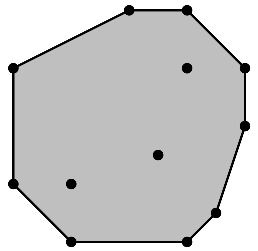
$A \subseteq \mathbb{R}^d$

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

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



Further parallel algorithms
Convex hull

$d = 2$

Fundamental arithmetic primitive: signed area of a triangle

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

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

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

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

All of A is to the left of every edge of conv A, traversed counterclockwise

Further parallel algorithms
Convex hull

The (discrete) convex hull problem

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

Output (a finite representation of) conv a

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

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

Output must be structured:

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

Further parallel algorithms
Convex hull

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

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

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

- compute conv$\{(x_i, x_i^2) \in \mathbb{R}^2 : 0 \leq i < n\}$
- follow the edges from vertex to vertex to obtain sorted output $\square$

Further parallel algorithms
Convex hull

The discrete convex hull problem

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

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

$d > 3$: much bigger output...

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

Hence

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

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

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

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

An ϵ-net may always be assumed to be in convex position

Set $E \subseteq A$ is an ϵ-approximation for A, if for all α, $0 \leq \alpha \leq 1$, any halfspace with $\alpha|E|$ points in E covers $(\alpha \pm \epsilon)|A|$ points in A

An ϵ-approximation may not be in convex position

Both are 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

The converse does not hold!

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, independently of d. □

Further parallel algorithms
Convex hull

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

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, put both hull vertices into E

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

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

In total $\leq 2r$ heavy triangles and bins, hence $|E| \leq 2r$ □

Further parallel algorithms
Convex hull

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

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

Proof. Sort points of A in circular order they appear on the convex hull

Put every n/r-th point into E. We have $|E| \leq r$. □

Further parallel algorithms
Convex hull

Parallel 2D hull computation by generalised regular sampling

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

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 vertex of splitter polygon defines a bucket: the subset of set a visible when sitting at this vertex (assuming the polygon is opaque)

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

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

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

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-spaces 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

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

1 Computation by circuits

2 Parallel computation models

3 Basic parallel algorithms

4 Further parallel algorithms

5 Parallel matrix algorithms

6 Parallel graph algorithms

Parallel matrix algorithms

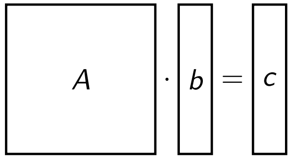
Matrix-vector multiplication

A: n-matrix $\quad b, c$: n-vectors

The matrix-vector multiplication problem

$A \cdot b = c$

$c_i = \sum_j A_{ij} \cdot b_j \; (0 \le i, j < n)$



Consider elements of b as inputs and of c as outputs

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

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

Sequential work $O(n^2)$

Parallel matrix algorithms

Matrix-vector multiplication

The matrix-vector multiplication circuit

$c \leftarrow 0$

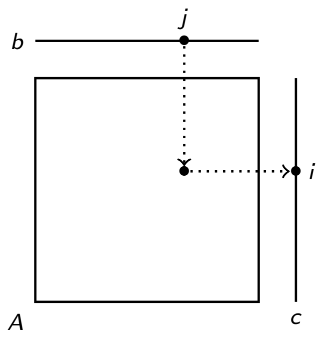
For all i, j: $c_i \overset{+}{\leftarrow} c_j^i \leftarrow A_{ij} \cdot b_j$
(adding each c_j^i to c_i asynchronously)

n input nodes of outdegree n, one per element of b

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

n output nodes of indegree n, one per element of c

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



Parallel matrix algorithms

Matrix-vector multiplication

Parallel matrix-vector multiplication

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

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

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

Overall, p block products $A_{IJ} \cdot b_J = c_I^J$

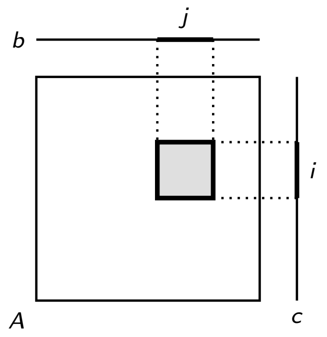
$c_I = \sum_{0 \le J < p^{1/2}} c_I^J$ for all I

Parallel matrix algorithms
Matrix-vector multiplication

Parallel matrix-vector multiplication (contd.)

$c \leftarrow 0$

For all I, J: $c_I \overset{+}{\leftarrow} c_I^J \leftarrow A_{IJ} \cdot b_J$



Parallel matrix algorithms
Matrix-vector multiplication

Parallel matrix-vector multiplication (contd.)

Initialise $c \leftarrow 0$ in external memory

Every processor

- is assigned I, J and block A_{IJ}
- reads block b_J and computes $c_I^J \leftarrow A_{IJ} \cdot b_J$
- updates $c_I \overset{+}{\leftarrow} c_I^J$ in external memory; concurrent writing resolved by operator '+' (recall array broadcast/combine)

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

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

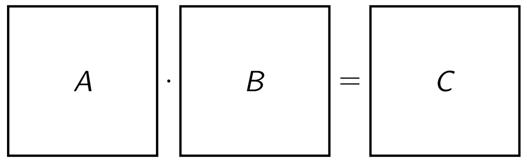
Parallel matrix algorithms
Matrix multiplication

A, B, C: n-matrices

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

Sequential work $O(n^3)$

Parallel matrix algorithms
Matrix multiplication

The matrix multiplication circuit

$C_{ik} \leftarrow 0$

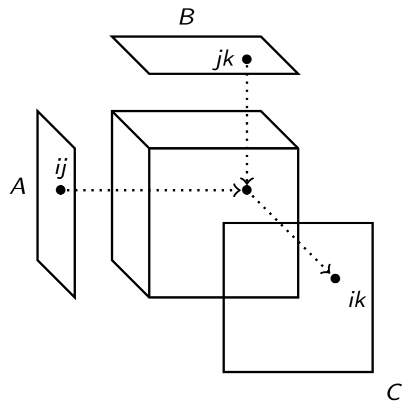
For all i, j, k: $C_{ik} \overset{+}{\leftarrow} C_{ik}^j \leftarrow A_{ij} \cdot B_{jk}$
(adding each C_{ik}^j to C_{ik} asynchronously)

$2n$ input nodes of outdegree n, one per element of A, B

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

n output nodes of indegree n, one per element of C

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



Parallel matrix algorithms
Matrix multiplication

Parallel matrix multiplication

Partition computation nodes into a regular grid of $p = p^{1/3} \cdot p^{1/3} \cdot p^{1/3}$ cubic $\frac{n}{p^{1/3}}$-blocks

Matrices A, B, C each gets partitioned into $p^{2/3}$ square $\frac{n}{p^{1/2}}$-blocks A_{IJ}, B_{JK}, C_{IK} $(0 \leq I, J, K < p^{1/3})$

Overall, p block products $A_{IJ} \cdot B_{JK} = C^J_{IK}$

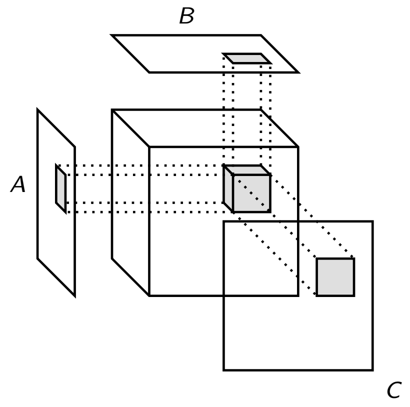
$C_{IK} = \sum_{0 \leq J < p^{1/2}} C^J_{IK}$ for all I, K

Parallel matrix algorithms
Matrix multiplication

Parallel matrix multiplication (contd.)

$C \leftarrow 0$

For all I, J, K: $C_{IK} \overset{+}{\leftarrow} C^J_{IK} \leftarrow A_{IJ} \cdot B_{JK}$



Parallel matrix algorithms
Matrix multiplication

Parallel matrix multiplication (contd.)

Initialise $C \leftarrow 0$ in external memory

Every processor

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

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

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

Parallel matrix algorithms
Matrix multiplication

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

Proof: (discrete) volume vs surface area

Let V be the subset of nodes computed by a certain processor

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

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

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

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

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

Note that this is not conditioned on $comp = O\left(\frac{n^3}{p}\right)$

Parallel matrix algorithms
Matrix multiplication

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

Includes e.g.

- numerical matrix multiplication with only '$+$', '$\cdot$' allowed
- Boolean matrix multiplication with only '$\vee$', '$\wedge$' allowed

Excludes e.g.

- numerical matrix multiplication when '$-$' also allowed
- Boolean matrix multiplication when '`if/then`' also allowed

Parallel matrix algorithms
Fast matrix multiplication

2-matrix multiplication: standard circuit

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

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

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

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

Parallel matrix algorithms
Fast matrix multiplication

2-matrix multiplication: Strassen's circuit

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

Let A, B, C be over a ring: '$+$', '$-$', '$\cdot$' allowed on elements

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

$$D^{(1)} = (A_{10} + A_{11}) \cdot B_{00} \qquad D^{(2)} = A_{00} \cdot (B_{01} - B_{11})$$

$$D^{(3)} = A_{11} \cdot (B_{10} - B_{00}) \qquad D^{(4)} = (A_{00} + A_{01}) \cdot B_{11}$$

$$D^{(5)} = (A_{10} - A_{00}) \cdot (B_{00} + B_{01}) \qquad D^{(6)} = (A_{01} - A_{11}) \cdot (B_{10} + B_{11})$$

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

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

$A_{00}, \ldots$: either ordinary elements or square blocks; 7 multiplications

Parallel matrix algorithms
Fast matrix multiplication

N-matrix multiplication: bilinear circuit

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

Bilinear circuits for matrix multiplication:

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

Elements of A, B, C: either ordinary elements or square blocks

Parallel matrix algorithms
Fast matrix multiplication

Block-recursive matrix multiplication

Given a scheme: bilinear circuit with fixed N, R

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

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

Apply the scheme, treating

- each '$+$' as block '$+$', each '$-$' as block '$-$'
- each '$\cdot$' as recursive call on blocks

Resulting recursive bilinear circuit:

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

Sequential work $O(n^\omega)$

Parallel matrix algorithms
Fast matrix multiplication

Block-recursive matrix multiplication (contd.)

Historical improvements in block-recursive matrix multiplication:

N	N^3	R	$\omega = \log_N R$	
2	8	8	3	standard
2	8	7	2.81	[Strassen: 1969]
3	27	23	$2.85 > 2.81$	
5	125	100	$2.86 > 2.81$	
48	110592	47216	2.78	[Pan: 1978]
...	...	...	...	
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

Block-recursive matrix multiplication (contd.)

Circuit size is determined by the scheme parameters N, R; the number of operations in scheme's linear combinations turns out to be irrelevant

Optimal circuit size unknown: only near-trivial lower bound $\Omega(n^2 \log n)$

Parallel matrix algorithms
Fast matrix multiplication

Parallel block-recursive matrix multiplication

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

At recursion level k:

- R^k independent block multiplication subproblems

In particular, at level $\log_R p$:

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

Parallel matrix algorithms
Fast matrix multiplication

Parallel block-recursive matrix multiplication (contd.)

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

Assigning matrix elements to processors:

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

E.g. cyclic distribution

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

Parallel matrix algorithms
Fast matrix multiplication

Parallel block-recursive matrix multiplication (contd.)

Each processor inputs its assigned elements of A, B

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

- linear combinations of blocks of A, B, no communication

Recursion levels below $\log_R p$: p block multiplication subproblems

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

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

- linear combinations giving blocks of C, no communication

Each processor outputs its assigned elements of C

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

Parallel matrix algorithms
Fast matrix multiplication

Theorem. Computing the block-recursive matrix multiplication dag requires communication $\Omega\left(\frac{n^2}{p^{2/\omega}}\right)$ per processor [Ballard+:2012]

Proof: generalises the Loomis–Whitney inequality using graph expansion (details omitted)

Parallel matrix algorithms
Boolean matrix multiplication

Boolean matrix multiplication

Let A, B, C be Boolean n-matrices: '$\vee$', '$\wedge$', '`if/then`' allowed on elements

$A \wedge B = C$

$C_{ik} = \bigvee_j A_{ik} \wedge B_{jk} \qquad 0 \leq i, j, k < n$

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

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

BSP costs in bit operations:

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

Parallel matrix algorithms
Boolean matrix multiplication

Fast Boolean matrix multiplication

$A \wedge B = C$

$A'_{ij} \leftarrow A_{ij}$ where 0, 1 are treated as integers
$B'_{jk} \leftarrow B_{jk}$ where 0, 1 are treated as integers

Compute $A' \cdot B' = C'$ modulo $n+1$ using a Strassen-like algorithm

$C_{ik} \leftarrow$ "$C'_{jk} \neq 0 \bmod n+1$"

Sequential work $O(n^\omega)$

BSP costs:

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

Parallel matrix algorithms
Boolean matrix multiplication

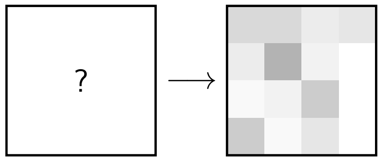
Parallel Boolean matrix multiplication by regular decomposition

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

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

$K = K(\epsilon)$, 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 by regular decomposition (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, at least one of A, B, $\overline{C}$ has few 1s, i.e. is sparse

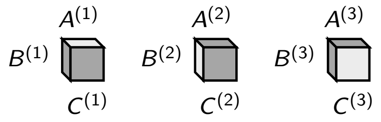
Fix ϵ so that "sparse" means density $\leq 1/p$

Parallel matrix algorithms
Boolean matrix multiplication

Parallel Boolean matrix multiplication by regular decomposition (contd.)

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

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



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

Parallel matrix algorithms
Boolean matrix multiplication

Parallel Boolean matrix multiplication by regular decomposition (contd.)

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

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

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

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

Parallel matrix algorithms
Boolean matrix multiplication

Parallel Boolean matrix multiplication by regular decomposition (contd.)

Consider J-layers of cubic blocks for a fixed J and all I, K

Every processor

- assigned a J-layer for fixed J
- reads A_{IJ}, B_{JK}
- computes $A_{IJ} \wedge B_{JK} = C^J_{IK}$ by fast Boolean multiplication for all I, K
- computes regular decomposition $A^{(1,2,3)}_{IJ} \wedge B^{(1,2,3)}_{JK} = C^{J(1,2,3)}_{IK}$ where $A^{(1)}_{IJ}$, $B^{(2)}_{JK}$, $\overline{C^{J(3)}_{IK}}$ sparse, for all I, K

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

Parallel matrix algorithms
Boolean matrix multiplication

Parallel Boolean matrix multiplication by regular decomposition (contd.)

Consider also I-layers for a fixed I and K-layers for a fixed K

Recompute every block product $A_{IJ} \wedge B_{JK} = C^J_{IK}$ by computing

- $A^{(1)}_{IJ} \wedge B^{(1)}_{JK} = C^{J(1)}_{IK}$ in K-layers
- $A^{(2)}_{IJ} \wedge B^{(2)}_{JK} = C^{J(2)}_{IK}$ in I-layers
- $A^{(3)}_{IJ} \wedge B^{(3)}_{JK} = C^{J(3)}_{IK}$ in J-layers

Every layer depends on $\leq \frac{n^2}{p}$ nonzeros of A, B, contributes $\leq \frac{n^2}{p}$ nonzeros to $\overline{C}$ due to sparsity

Communication saved by only sending the indices of nonzeros

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

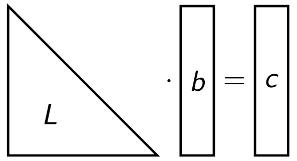
Parallel matrix algorithms
Triangular system solution

Let L be an n-matrix, b, c be n-vectors

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

$L \cdot b = c$

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



Parallel matrix algorithms
Triangular system solution

Forward substitution

$L \cdot b = c$

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

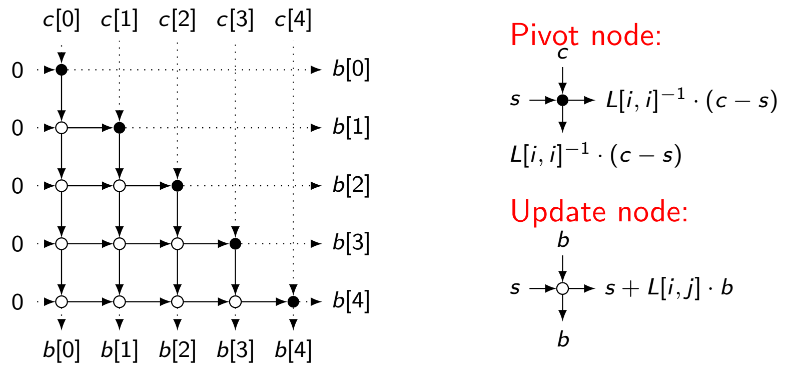
Sequential work $O(n^2)$

Symmetrically, an upper triangular system solved by back substitution

Parallel matrix algorithms
Triangular system solution

Parallel forward substitution by 2D grid

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-recursive forward substitution

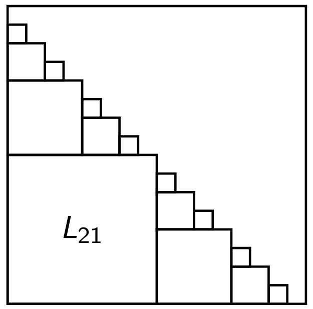
$L \cdot b = c$

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

Recursion: two half-sized subproblems

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

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



Sequential work $O(n^2)$

Parallel matrix algorithms
Triangular system solution

Parallel block-recursive forward substitution

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

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

At recursion level k:

- sequence of 2^k triangular system subproblems, each on $n/2^k$-blocks

In particular, at level $\log p$:

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

Parallel matrix algorithms
Triangular system solution

Parallel block-recursive forward substitution (contd.)

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

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

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

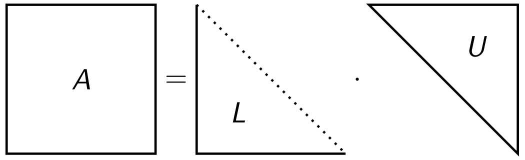
$comm = O(n/p^{1/2}) \cdot \left(1 + 2 \cdot \frac{1}{2} + 2^2 \cdot \frac{1}{2^2} + \ldots\right) + O(n/p) \cdot p = O(n/p^{1/2}) \cdot \log p + O(n) = O(n)$

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

Parallel matrix algorithms
Generic Gaussian elimination

Let A, L, U be n-matrices

LU decomposition of A: $A = L \cdot U$



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

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

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

Parallel matrix algorithms
Generic Gaussian elimination

Application: solving a linear system

$Ax = b$

If LU decomposition of A is known: $Ax = LUx = b$

Solve triangular systems $Ly = b$ then $Ux = y$, obtaining x

LU decomposition of A can be reused for multiple right-hand sides b

Parallel matrix algorithms
Generic Gaussian elimination

Block generic Gaussian elimination

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

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

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

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

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

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & \\ L_{10} & \bar{A}_{11} \end{bmatrix} \begin{bmatrix} U_{00} & U_{01} \\ & I \end{bmatrix}$$

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

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

Parallel matrix algorithms
Generic Gaussian elimination

Block generic Gaussian elimination (contd.)

$A_{00}, \ldots$: either ordinary elements or blocks, can be applied recursively

Recursion base: 1×1 matrix $A = 1 \cdot A$

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

- $A_{00} \neq 0$ (respectively $\det A_{00} \neq 0$)
- $\bar{A}_{11} \neq 0$ (respectively $\det \bar{A}_{11} \neq 0$)

Hence no pivoting required

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

Parallel matrix algorithms
Generic Gaussian elimination

Iterative generic Gaussian elimination

Let A be an $n \times n$ matrix

$$A = \begin{array}{c c} & \begin{array}{cc} (1) & (n-1) \end{array} \\ \begin{array}{c} (1) \\ (n-1) \end{array} & \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \end{array}$$

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

Sequential work $O(n^3)$

Parallel matrix algorithms
Generic Gaussian elimination

Recursive generic Gaussian elimination

Let A be an $n \times n$ matrix

$$A = \begin{array}{c c} & \begin{array}{cc} (n/2) & (n/2) \end{array} \\ \begin{array}{c} (n/2) \\ (n/2) \end{array} & \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \end{array}$$

$A = LU$ by block generic Gaussian elimination on A, treating

- each '$+$' ('$-$', '$\cdot$') as block '$+$' ('$-$', '$\cdot$')
- each LU decomposition as recursive call on blocks

Sequential work:

- $O(n^3)$ using standard matrix multiplication
- $O(n^\omega)$ using fast (Strassen-like) matrix multiplication

Parallel matrix algorithms
Generic Gaussian elimination

Parallel recursive generic Gaussian elimination

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

At recursion level k:

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

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

- sequence of $p^{1/2}$ LU decomposition subproblems, each on $\frac{n}{p^{1/2}}$-blocks
- total $p^{1/2} \cdot O\big((\frac{n}{p^{1/2}})^3\big) = O(\frac{n^3}{p})$ sequential work, therefore each subproblem can be solved sequentially on an arbitrary processor

Parallel matrix algorithms
Generic Gaussian elimination

Parallel recursive generic Gaussian elimination (contd.)

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

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

- block generic LU decomposition using parallel matrix multiplication

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

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

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

Parallel matrix algorithms
Generic Gaussian elimination

Parallel recursive generic Gaussian elimination (contd.)

More generally: threshold level $\alpha \log p$, $1/2 \leq \alpha \leq 2/3$

Recursion levels 0 to $\alpha \log p$:

- block generic LU decomposition using parallel matrix multiplication

Threshold recursion level $\alpha \log p$:

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

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

Parallel matrix algorithms
Generic Gaussian elimination

Parallel recursive generic Gaussian elimination (contd.)

Continuous tradeoff between *comm* and *sync*

Controlled by parameter α, $1/2 \leq \alpha \leq 2/3$

$\alpha = 1/2$: *comm* and *sync* as for 3D grid

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

$\alpha = 2/3$:

- *comm* goes down to that of matrix multiplication
- *sync* goes up accordingly

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

Parallel matrix algorithms
Gaussian elimination with pivoting

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

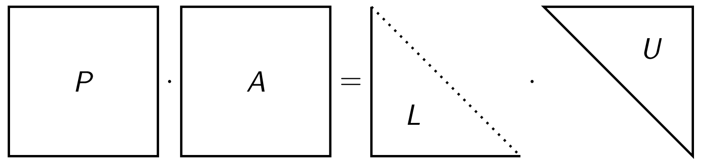
- pivot elements are always nonzero
- pivot blocks are always nonsingular

Parallel matrix algorithms

Gaussian elimination with pivoting

Let A, P, L, U be n-matrices

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



P is a permutation matrix:

- all elements 0 or 1
- exactly one 1 in every row and column

L is unit lower triangular, U is upper triangular

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

Parallel matrix algorithms

Gaussian elimination with pivoting

Block Gaussian elimination with column pivoting

Generalise PLU decomposition to “tall” rectangular matrices

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

$$A = \begin{array}{c c} & (n) \\ (n) & \multirow{2}{*}{$\begin{bmatrix} A_{00} \\ A_{10} \end{bmatrix}$} \\ (m-n) & \end{array} \qquad P \cdot \begin{bmatrix} A_{00} \\ A_{10} \end{bmatrix} = \begin{bmatrix} L_{00} \\ L_{10} \end{bmatrix} \cdot \begin{bmatrix} U_{00} \\ \cdot \end{bmatrix}$$

P is an $m \times m$ permutation matrix

L_{00} is $n \times n$ unit lower triangular, U_{00} is $n \times n$ upper triangular

Parallel matrix algorithms

Gaussian elimination with pivoting

Block Gaussian elimination with column pivoting (contd.)

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

Compute $\begin{bmatrix} P_{00} & P_{01} \\ P'_{10} & P'_{11} \end{bmatrix} \begin{bmatrix} A_{00} \\ A_{10} \end{bmatrix} = \begin{bmatrix} L_{00} \\ L'_{10} \end{bmatrix} \begin{bmatrix} U_{00} \\ \cdot \end{bmatrix}$

$U_{01} \leftarrow L_{00}^{-1}(P_{00}A_{01} + P_{01}A_{11})$

$\bar{A}'_{11} \leftarrow P'_{10}A_{01} + P'_{11}A_{11} - L'_{10}U_{01}$

$$\begin{bmatrix} P_{00} & P_{01} \\ P'_{10} & P'_{11} \end{bmatrix} \begin{bmatrix} A_{00} & A_{01} \\ A_{01} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & \\ L'_{10} & \bar{A}'_{11} \end{bmatrix} \begin{bmatrix} U_{00} & U_{01} \\ \cdot & I \end{bmatrix}$$

Compute $P''_{11}\bar{A}'_{11} = L_{11}U_{11}$

$$\begin{bmatrix} P_{00} & P_{01} \\ P''_{11}P'_{10} & P''_{11}P'_{11} \end{bmatrix} \begin{bmatrix} A_{00} & A_{01} \\ A_{01} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & \\ P''_{11}L'_{10} & L_{11} \end{bmatrix} \begin{bmatrix} U_{00} & U_{01} \\ \cdot & U_{11} \end{bmatrix}$$

Parallel matrix algorithms

Gaussian elimination with pivoting

Block Gaussian elimination with column pivoting (contd.)

$A_{00}, \ldots$: either ordinary elements or blocks, can be applied recursively

Recursion base: $m \times 1$ matrix

$$A = \begin{array}{c c} & (1) \\ (1) & \multirow{2}{*}{$\begin{bmatrix} A_0 \\ A_1 \end{bmatrix}$} \\ (m-1) & \end{array} \qquad P \begin{bmatrix} A_0 \\ A_1 \end{bmatrix} = \begin{bmatrix} A'_0 \\ A'_1 \end{bmatrix} = \begin{bmatrix} 1 \\ L_1 \end{bmatrix} \begin{bmatrix} A'_0 \\ \cdot \end{bmatrix}$$

P is a permutation such that $|A'_0|$ is largest across A

Parallel matrix algorithms
Gaussian elimination with pivoting

Iterative Gaussian elimination with column pivoting

Let A be an $n \times n$ matrix

$$A = \begin{array}{c|cc} & (1) & (n-1) \\ \hline (1) & A_{00} & A_{01} \\ (n-1) & A_{10} & A_{11} \end{array}$$

$PA = LU$ by block Gaussian elimination with column pivoting on A, then on $\bar{A}'_{11}$

Sequential work $O(n^3)$

Parallel matrix algorithms
Gaussian elimination with pivoting

Recursive Gaussian elimination with column pivoting

Let A be an $n \times n$ matrix

$$A = \begin{array}{c|cc} & (n/2) & (n/2) \\ \hline (n/2) & A_{00} & A_{01} \\ (n/2) & A_{10} & A_{11} \end{array}$$

$PA = LU$ by block Gaussian elimination with column pivoting on A, treating

- each '$+$' ('$-$', '$\cdot$') as block '$+$' ('$-$', '$\cdot$')
- each PLU decomposition as recursive call on blocks

Sequential work:

- $O(n^3)$ using standard matrix multiplication
- $O(n^\omega)$ using fast (Strassen-like) matrix multiplication

Parallel matrix algorithms
Gaussian elimination with pivoting

Parallel recursive Gaussian elimination with column pivoting

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

At recursion level k:

- sequence of 2^k PLU decomposition subproblems, each on $\frac{n}{2^k} \times n$ blocks

In particular, at level $\log p$:

- sequence of p PLU decomposition subproblems, each on $\frac{n}{p} \times n$-blocks
- total $p \cdot O(\frac{n^3}{p^2}) = O(\frac{n^3}{p})$ sequential work, therefore each subproblem can be solved sequentially on an arbitrary processor

Parallel matrix algorithms
Gaussian elimination with pivoting

Parallel recursive Gaussian elimination with column pivoting (contd.)

Level $\log p$: threshold to switch from parallel to sequential computation

Recursion levels 0 to $\log p$:

- block PLU decomposition using parallel matrix multiplication

Threshold recursion level $\log p$:

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

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

Parallel matrix algorithms
Gaussian elimination with pivoting

Parallel recursive Gaussian elimination with column pivoting (contd.)

Alternative: all recursion levels computed in parallel

Level $\log p$: threshold to switch from normal to fine-grained computation

Recursion levels 0 to $\log p$:

- block PLU decomposition using parallel matrix multiplication

Recursion levels $\log p$ to $\log n$:

- block PLU decomposition on partitioned matrix, using broadcast of pivot subrows and p instances of sequential matrix multiplication

Recursion base at level $\log n$:

- column PLU decomposition; pivot selected by balanced binary tree

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

Parallel matrix algorithms
Gaussian elimination with pivoting

Parallel recursive Gaussian elimination with column pivoting (contd.)

Discontinuous tradeoff between *comm* and *sync*

Coarse-grained algorithm: *comm* and *sync* as for 2D grid with work and data size $O(n)$ per node

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

Coarse-grained algorithm: *comm* as for matrix multiplication; *sync* becomes a function of n

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

1 Computation by circuits

2 Parallel computation models

3 Basic parallel algorithms

4 Further parallel algorithms

5 Parallel matrix algorithms

6 Parallel graph algorithms

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

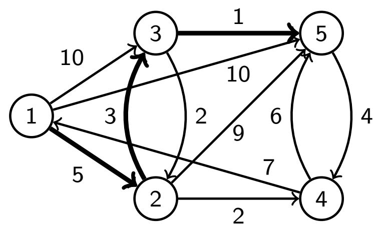
A_{ij} = length of the edge $i \to j$

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 $A_{\underline{00}} = 0$

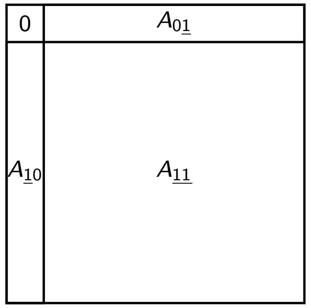
Replace each weight A_{ij}, $i, j \neq 0$, with $A_{i0} + A_{0j}$, if that gives a shortcut from i to j

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

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

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

Recursion: two half-sized subproblems

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

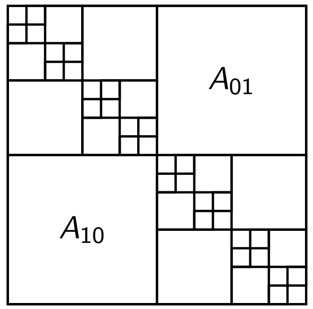
$A'_{\underline{01}} \leftarrow A'_{\underline{00}} A_{\underline{01}} \quad A'_{\underline{10}} \leftarrow A_{\underline{10}} A'_{\underline{00}} \quad A'_{\underline{11}} \leftarrow A_{\underline{11}} \oplus A_{\underline{10}} A'_{\underline{00}} A_{\underline{01}}$

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

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

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: $1/2 \leq \alpha \leq 2/3$

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

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	$\mathbb{R}_{\geq 0} \cup \{+\infty\}$	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 $i \to j$

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

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

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

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

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 $A^{=\frac{k}{2}}$, ..., $A^{=k}$. 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, $A^{\le \frac{3}{2}+\cdots}$, $A^{(\le \frac{3}{2})^2+\cdots}$, $\dots$, $A^{\le p+\cdots}$ by $\le \log_{3/2} p$ rounds of parallel sparse-by-dense matrix $\odot$-multiplication

Consider $A^{=0}, \dots, A^{=p}$. 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 $q \le \frac{p}{2}$, 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 $(A^{=p})^* \odot A^{\le p+\cdots} = A^*$ 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, $A^{\le \frac{3}{2}+\cdots}$, $A^{(\le \frac{3}{2})^2+\cdots}$, $\dots$, $A^{\le p^2+\cdots}$ by $\le 2\log p$ 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 $A^{=0}, \dots, A^{=\frac{p}{2}}$ and $A^{=(p)-\frac{p}{2}}, \dots, A^{=(p)}$. 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 $q \le \frac{p}{2}$, matrices $A^{=q}$ and $A^{=(p)-q}$ have together at most $\frac{2n^2}{p}$ non-$\boxed{0}$ elements.

Parallel graph algorithms
All-pairs shortest paths

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 $(A^{=p})^* \odot A^{\le p} = A^*$ by parallel matrix $\odot$-multiplication

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