Efficient Parallel Algorithms

Alexander Tiskin

Department of Mathematics and Computer Science
St Petersburg University

Model: abstraction of reality allowing qualitative and quantitative reasoning

Examples:

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

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 $\rightarrow$ (computation steps) $\rightarrow$ output

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

Assumes a specific computation model and input/output encoding

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) $T : \mathbb{N} \to \mathbb{N}$

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

Computation by circuits
Computation models and algorithms

Computation: input $\to$ (computation steps) $\to$ 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)

$T(n)$ is usually analysed asymptotically:

$T(n)$ is usually analysed asymptotically:

- up to a constant factor
- for sufficiently large n

$T(n)$ 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: $O(f)$, $o(f)$, $\Omega(f)$, $\omega(f)$, $\Theta(f)$

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

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

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

$g = O(f)$: "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

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

$g = O(f)$: "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"

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

$g = O(f)$: "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

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

$g = O(f)$: "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?

$g = \Omega(f)$: "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 $f = o(g)$

$g = \Omega(f)$: "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 $g = \Omega(f)$

$g = \Omega(f)$: "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.

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

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

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

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

Proof:

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

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

Proof: for all n, we have

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

Example usage: sorting an array of size n

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

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

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

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

Example usage: multiplying $n \times n$ matrices

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

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

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

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)

Algorithm complexity depends on the model

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)

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

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

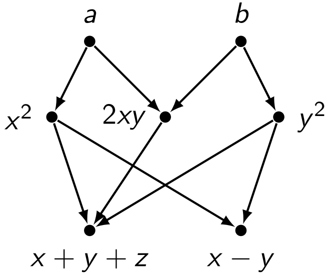
E.g. deciding if a program halts on a given input:

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

Basic special-purpose parallel model: a circuit

$a^2 + 2ab + b^2$

$a^2 - b^2$

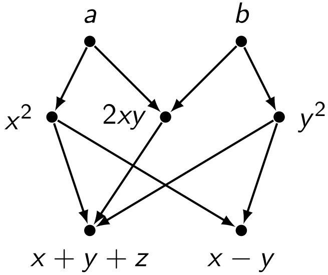


Computation by circuits
The circuit model

Basic special-purpose parallel model: a circuit

$a^2 + 2ab + b^2$

$a^2 - b^2$



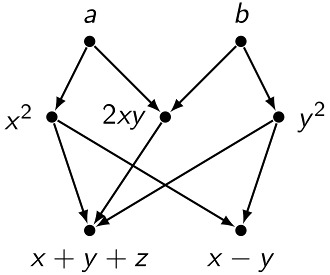
Directed acyclic graph (dag), fixed number of inputs/outputs

Models oblivious computation: control sequence independent of the input

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

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

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

A comparison network is a circuit of comparator nodes

$\sqcap = \min$

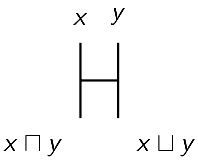
$\sqcup = \max$

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

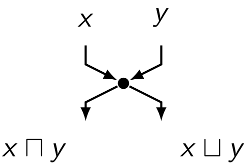
Computation by circuits

The comparison network model

A comparison network is a circuit of comparator nodes



denotes

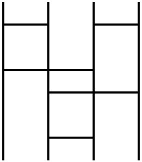


$\sqcap = \min$

$\sqcup = \max$

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

Examples:

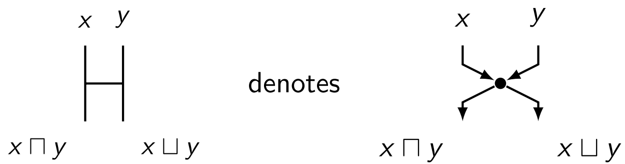


$n = 4$

size 5

depth 3

A comparison network is a circuit of comparator nodes

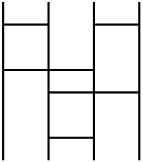


$\sqcap = \min$

$\sqcup = \max$

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

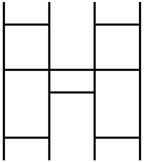
Examples:



$n = 4$

size 5

depth 3



$n = 4$

size 6

depth 3

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

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

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?

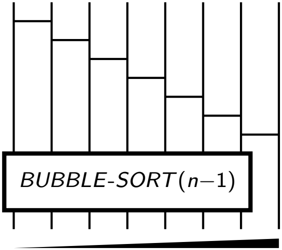
Computation by circuits

Naive sorting networks

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

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

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



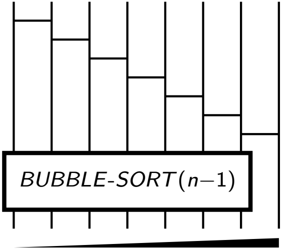
Computation by circuits

Naive sorting networks

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

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

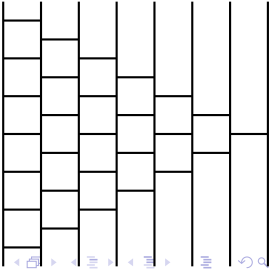
depth $2n - 3 = O(n)$



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

size 28

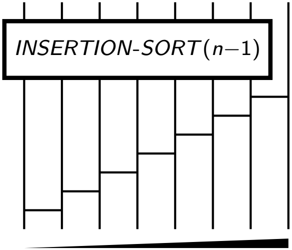
depth 13



INSERTION-SORT(n)

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

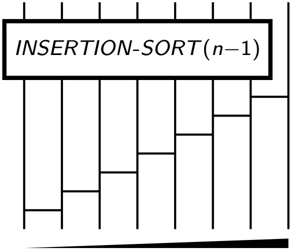
depth $2n-3 = O(n)$



INSERTION-SORT(n)

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

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

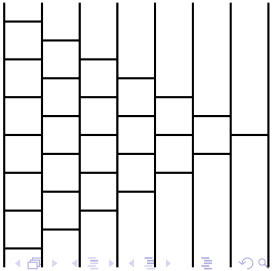


INSERTION-SORT(8)

size 28

depth 13

Identical to *BUBBLE-SORT*!



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

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

Proof.

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.

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

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

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

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 $\square$

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

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 $n! = (1 + o(1))(2\pi n)^{1/2} \cdot (n/e)^n$
- a merging network by checking only $(n' + 1) \cdot (n'' + 1)$ pairs of input sequences, instead of a (typically) very much larger number $\binom{n}{n'} = \binom{n}{n''}$, e.g. for $n = 2n' = 2n''$: $\binom{n}{n'} = (1 + o(1))(\pi n/2)^{-1/2} \cdot 2^n$

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

How fast can we merge obliviously?

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

How fast can we merge obliviously?

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

Odd-even merging

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

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

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

How fast can we merge obliviously?

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

Odd-even merging

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

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

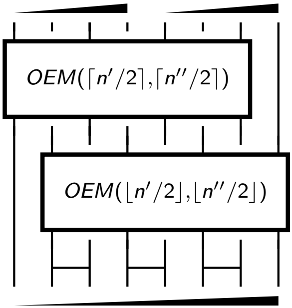
$size(OEM(n', n'')) \leq 2 \cdot size(OEM(n'/2, n''/2)) + O(n) = O(n \log n)$

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

$OEM(n', n'')$

size $O(n \log n)$

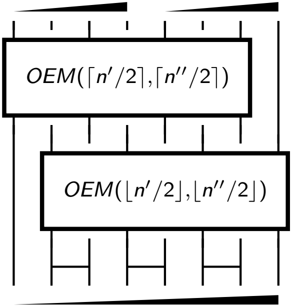
depth $O(\log n)$



$OEM(n', n'')$

size $O(n \log n)$

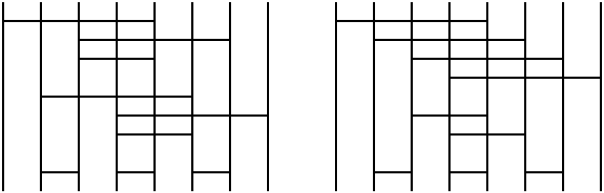
depth $O(\log n)$



$OEM(4, 4)$

size 9

depth 3



Correctness proof of odd-even merging:

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

Induction base: trivial (2 inputs, 1 comparator)

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

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

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

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

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

Induction base: trivial (2 inputs, 1 comparator)

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

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

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

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

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

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

Induction base: trivial (2 inputs, 1 comparator)

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

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

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

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

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

The final stage of comparators corrects the wrong pair

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

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

Odd-even merge sorting [Batcher: 1968]

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

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

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

Odd-even merge sorting [Batcher: 1968]

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

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

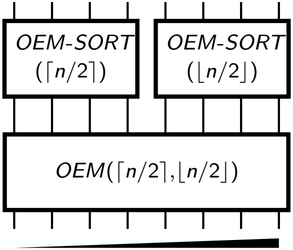
$$size(OEM\text{-}SORT)(n) \leq$$
$$2 \cdot size(OEM\text{-}SORT(n/2)) + size(OEM(n/2, n/2)) =$$
$$2 \cdot size(OEM\text{-}SORT(n/2)) + O(n \log n) = O\big(n(\log n)^2\big)$$

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

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

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

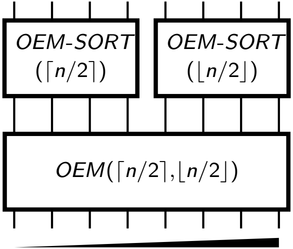
depth $O\big((\log n)^2\big)$



OEM-SORT(n)

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

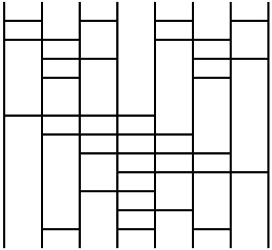
depth $O\big((\log n)^2\big)$



OEM-SORT(8)

size 19

depth 6



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

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

Bitonic merging: sorting a bitonic sequence

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

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

Exercise: prove correctness (by zero-one principle)

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

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

Bitonic merging: sorting a bitonic sequence

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

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

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

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

Bitonic merging: sorting a bitonic sequence

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

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

Exercise: prove correctness (by zero-one principle)

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

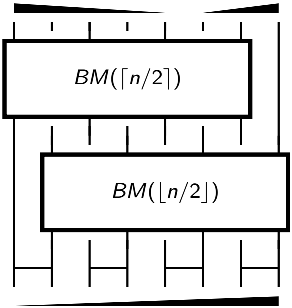
Bitonic merging is more flexible than odd-even merging, since for a fixed n, a single circuit applies to all values of m

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

$BM(n)$

size $O(n \log n)$

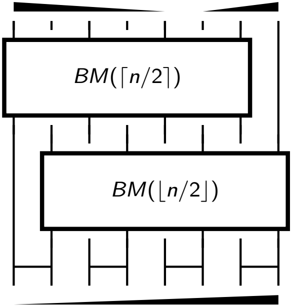
depth $O(\log n)$



$BM(n)$

size $O(n \log n)$

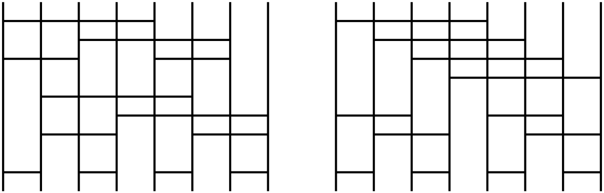
depth $O(\log n)$



$BM(8)$

size 12

depth 3



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 $\langle y_1 \geq \cdots \geq y_m \leq \cdots \leq y_n \rangle$ $\qquad m = \lceil n/2 \rceil$ or $\lceil n/2 \rceil + 1$

Sorting in reverse seems to require "inverted comparators"

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

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

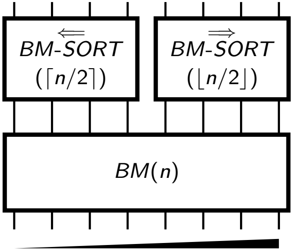
- comparators are actually nodes in a circuit, which can always be drawn using "standard comparators"
- a network drawn with "inverted comparators" can be converted into one with only "standard comparators" by a top-down rearrangement

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

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

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

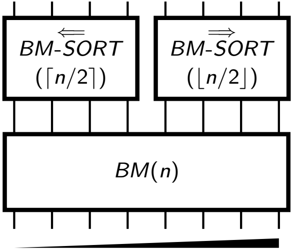
depth $O\big((\log n)^2\big)$



BM-SORT(n)

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

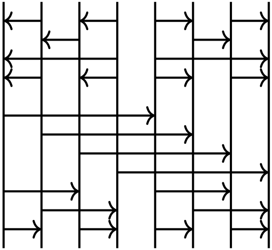
depth $O\big((\log n)^2\big)$

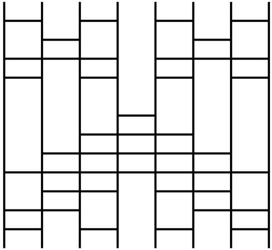


BM-SORT(8)

size 24

depth 6





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

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

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

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

AKS sorting [Ajtai, Komlós, Szemerédi: 1983]

[Paterson: 1990]; [Seiferas: 2009]

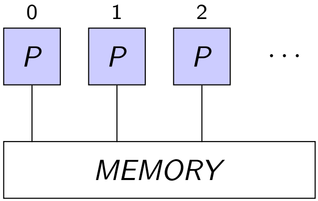
Sorting network: size $O(n \log n)$, depth $O(\log n)$

Uses sophisticated graph theory (expanders)

Asymptotically optimal, but has huge constant factors

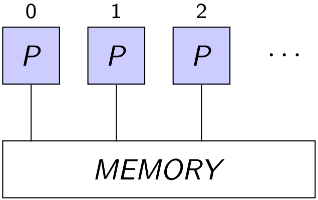
Parallel Random Access Machine (PRAM) [Fortune, Wyllie: 1978]

Simple, idealised general-purpose parallel model



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

Simple, idealised general-purpose parallel model



Contains

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

Operates in full synchrony

PRAM computation: sequence of parallel steps

Communication and synchronisation taken for granted

Not scalable in practice!

PRAM 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

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

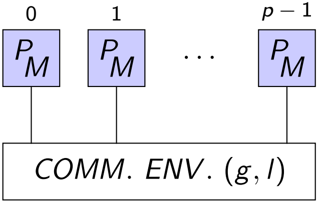
E.g. a linear system solver: $O\big((\log n)^2\big)$ steps using n^4 processors :-O

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

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

Simple, realistic general-purpose parallel model

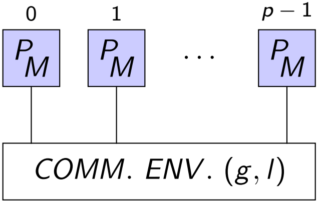
Goals: scalability, portability, predictability



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

Simple, realistic general-purpose parallel model

Goals: scalability, portability, predictability



Contains

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

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

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

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

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

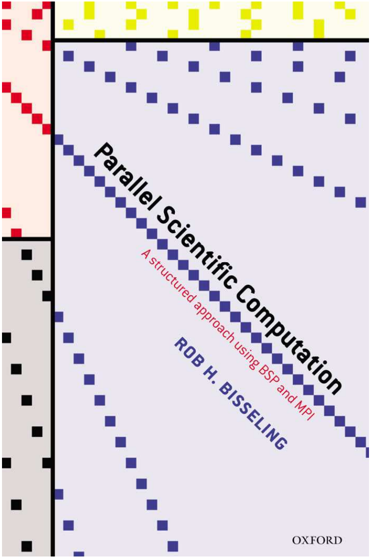
- external memory by local memory + network communication
- barrier synchronisation mechanism by network communication

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: $p = 64$, $g \approx 78$, $l \approx 1825$



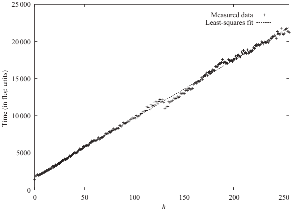


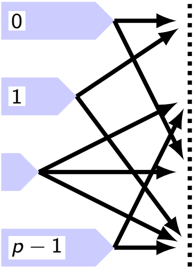
FIG. 1.13. Time of an h-relation on a 64-processor Cray T3E.

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)

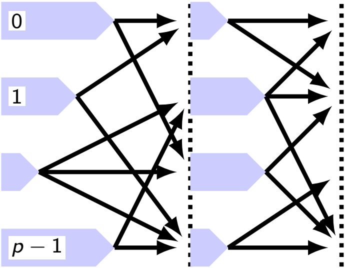
p	g	l	$T_{\rm comm}(0)$
1	36	47	38
2	28	486	325
4	31	679	437
8	31	1193	580
16	31	2018	757
32	72	1145	871
64	78	1825	1440

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.

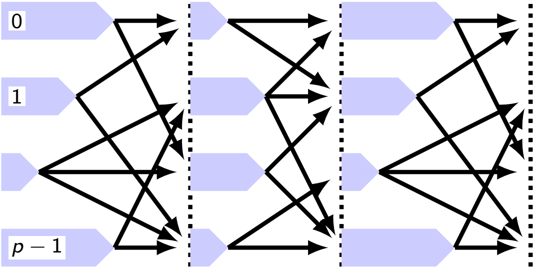
BSP computation: sequence of parallel supersteps



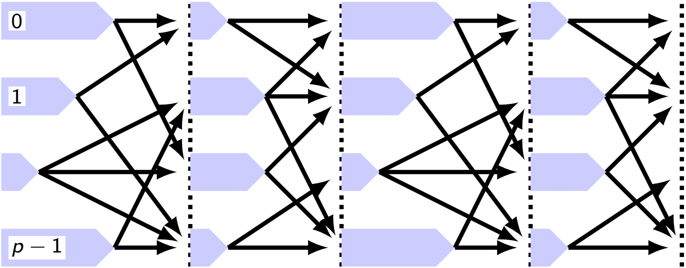
BSP computation: sequence of parallel supersteps



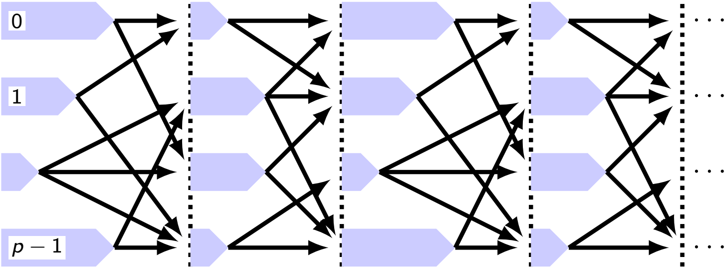
BSP computation: sequence of parallel supersteps



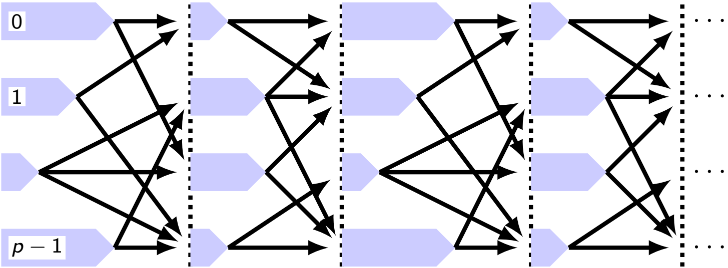
BSP computation: sequence of parallel supersteps



BSP computation: sequence of parallel supersteps



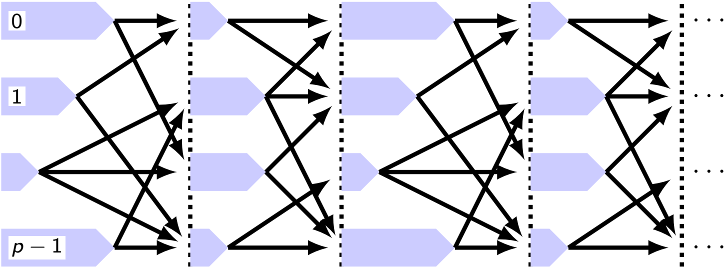
BSP computation: sequence of parallel supersteps



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

Synchronisation before/after each superstep

BSP computation: sequence of parallel supersteps



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

Synchronisation before/after each superstep

Cf. CSP: parallel collection of sequential processes

Compositional cost model

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

- $comp(sstep, proc)$: 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*

Compositional cost model

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

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

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

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

For the whole BSP computation with *sync* supersteps:

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

For the whole BSP computation with *sync* supersteps:

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

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

For the whole BSP computation with *sync* supersteps:

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

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

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

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

BSP algorithm design

Minimising *comp*, *comm*, *sync* as functions of n, p

Conventions:

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

Data locality exploited, network locality ignored

BSP algorithm design (contd.)

Computation balancing

- require work-optimal $comp = O\left(\frac{seq\_work}{p}\right)$

Communication balancing:

- aim for scalable $comm = O\left(\frac{input+output}{p^c}\right)$, $0 < c \leq 1$
- ideally fully-scalable $comm = O\left(\frac{input+output}{p}\right)$

Coarse granularity:

- aim for *sync* independent of n (may depend on p)
- better quasi-flat $sync = O\left((\log p)^{O(1)}\right)$
- ideally flat $sync = O(1)$

BSP software: industrial projects

- Google's Pregel [2010]
- Apache Hama, Spark, Giraph (`apache.org`) [2010–16]

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]

Broadcasting:

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

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
- at the end, one designated processor must hold $\sum_r a_r$
- addition can be replaced by any given associative operator $\bullet$: $a \bullet (b \bullet c) = (a \bullet b) \bullet c$, computable in time $O(1)$

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

Broadcasting:

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

Combining (complementary to broadcasting):

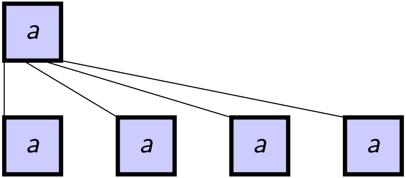
- initially, every processor r holds a value a_r
- at the end, one designated processor must hold $\sum_r a_r$
- addition can be replaced by any given associative operator $\bullet$: $a \bullet (b \bullet c) = (a \bullet b) \bullet c$, computable in time $O(1)$

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

By symmetry, we only need to consider broadcasting

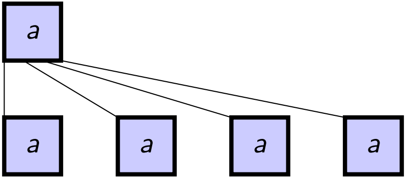
Direct broadcast:

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



Direct broadcast:

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



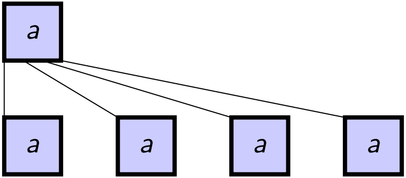
$comp = O(p)$

$comm = O(p)$

$sync = O(1)$

Direct broadcast:

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

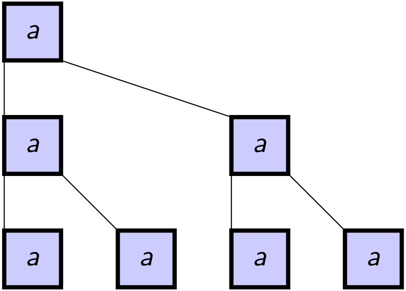


$comp = O(p)$ $comm = O(p)$ $sync = O(1)$

Cost components will be shaded when they are optimal, i.e. cannot be improved by another algorithm (under certain explicit assumptions)

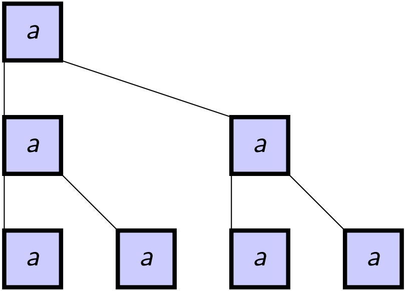
Binary tree broadcast:

- initially, only designated processor is awake
- processors wake up each other 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



Binary tree broadcast:

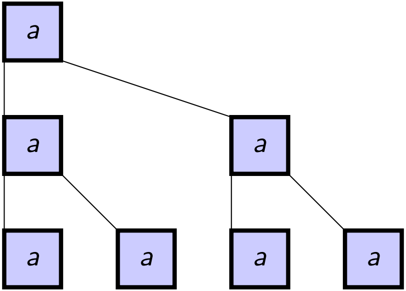
- initially, only designated processor is awake
- processors wake up each other in $\log p$ rounds
- in every round, every awake processor makes a copy of a and send it to a sleeping processor, waking it up



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

Binary tree broadcast:

- initially, only designated processor is awake
- processors wake up each other 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

$comp = O(\log p)$ | $comm = O(\log p)$ | $sync = O(\log p)$

Array broadcasting:

- initially, one designated processor holds array a of size $n \geq p$
- at the end, every processor must hold a copy of the whole array a
- effectively, n independent instances of broadcasting

Array broadcasting:

- initially, one designated processor holds array a of size $n \geq p$
- at the end, every processor must hold a copy of the whole array a
- effectively, n independent instances of broadcasting

Array combining (complementary to array broadcasting):

- initially, every processor r holds an array a_r of size $n \geq p$
- at the end, one designated processor must hold componentwise $\sum_r a_r$
- addition can be replaced by any given associative operator $\bullet$
- effectively, n independent instances of combining

Array broadcasting:

- initially, one designated processor holds array a of size $n \geq p$
- at the end, every processor must hold a copy of the whole array a
- effectively, n independent instances of broadcasting

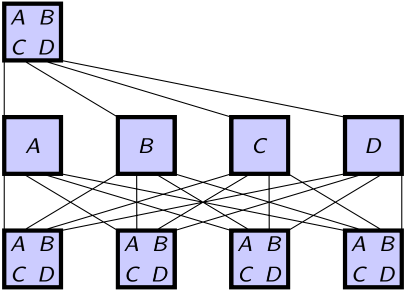
Array combining (complementary to array broadcasting):

- initially, every processor r holds an array a_r of size $n \geq p$
- at the end, one designated processor must hold componentwise $\sum_r a_r$
- addition can be replaced by any given associative operator $\bullet$
- effectively, n independent instances of combining

By symmetry, we only need to consider array broadcasting

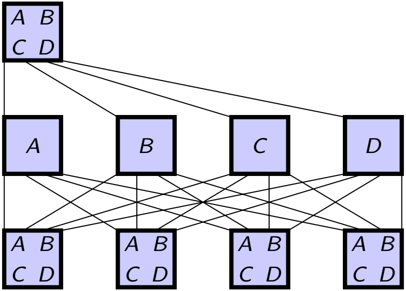
Two-phase array broadcast:

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



Two-phase array broadcast:

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



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

Parallel computation models

Fundamental communication patterns

Array broadcasting/combining enables concurrent access to external memory in blocks of size $\geq p$

Concurrent reading: a designated processor

- reads block from external memory
- broadcasts block

Concurrent writing, resolved by $\bullet$: a designated processor

- combines blocks from each processor by $\bullet$
- writes combined block to external memory

Two-phase array broadcast/combine used as subroutine

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

Has to be implemented on concrete networks

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

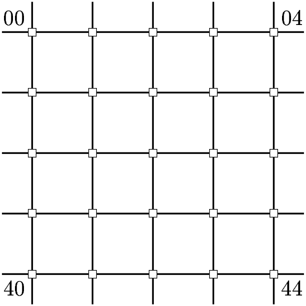
Low diameter — more efficient

2D array network

$p = q^2$ processors

degree 4

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

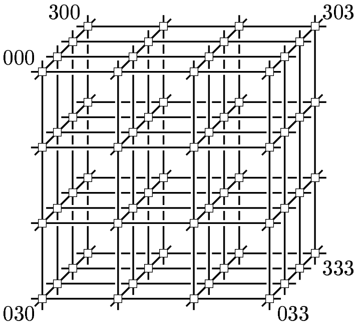


3D array network

$p = q^3$ processors

degree 6

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

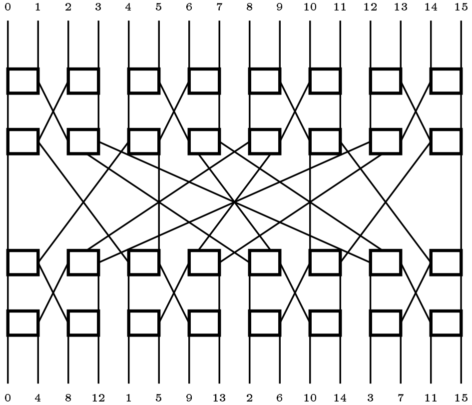


Butterfly network

$p = q \log q$ processors

degree 4

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

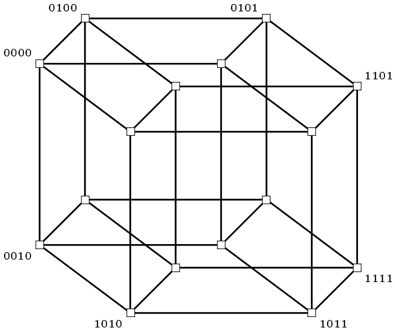


Hypercube network

$p = 2^q$ processors

degree $\log p = q$

diameter $\log p = q$



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

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

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

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

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

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 $\Omega(diameter)$ steps

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 $\Omega(diameter)$ steps

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

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

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

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	$O(\log p)$	$O(\log p)$

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

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

Two-phase randomised routing: [Valiant: 1980]

- send every packet to random intermediate destination
- forward every packet to final destination

Both phases oblivious (e.g. greedy), but non-oblivious overall due to randomness

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

Two-phase randomised routing: [Valiant: 1980]

- send every packet to random intermediate destination
- forward every packet to final destination

Both phases oblivious (e.g. greedy), but non-oblivious overall due to randomness

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

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

Hence constant g, l in the BSP model

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

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

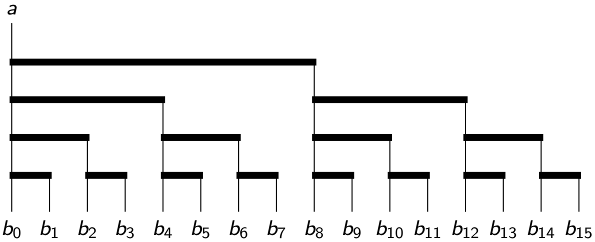
The balanced binary tree circuit:

tree(n)

1 input, n outputs

size $n - 1$

depth $\log n$



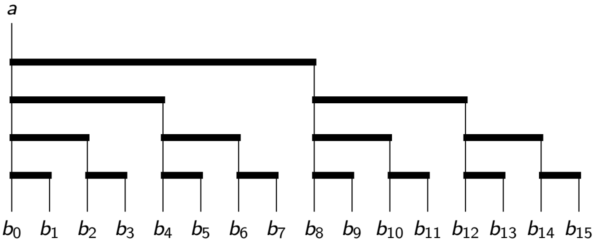
The balanced binary tree circuit:

tree(n)

1 input, n outputs

size $n - 1$

depth $\log n$



Every node computes an arbitrary given operation in time $O(1)$

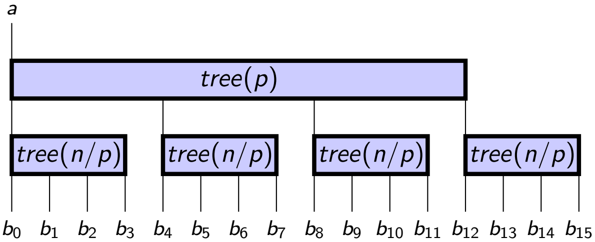
Can be directed

- top-down (one input at root, n outputs at leaves)
- bottom-up (n inputs at leaves, one output at root)

Sequential work $O(n)$

Parallel balanced tree computation, $p = 4$

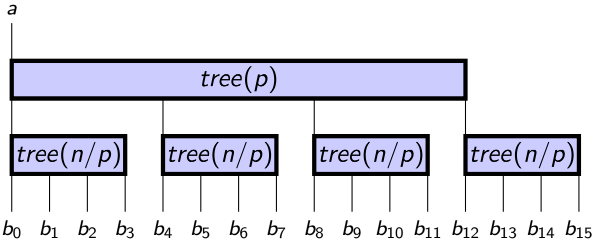
$tree(n)$



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

Parallel balanced tree computation, $p = 4$

tree(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

Partition *tree*(n) into

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

Parallel balanced tree computation (contd.)

For top-down computation, a designated processor

- is assigned the top block
- reads block's input, computes block, writes block's p outputs

Then every processor

- is assigned a different bottom block
- reads block's input, computes block, writes block's n/p outputs

For bottom-up computation, reverse the steps

Parallel balanced tree computation (contd.)

For top-down computation, a designated processor

- is assigned the top block
- reads block's input, computes block, writes block's p outputs

Then every processor

- is assigned a different bottom block
- reads block's input, computes block, writes block's n/p outputs

For bottom-up computation, reverse the steps

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

Required slackness $n \geq p^2$

The described parallel balanced tree algorithm is fully optimal:

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

The described parallel balanced tree algorithm is fully optimal:

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

For other problems, we may not be so lucky 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 "cheat" by running the whole computation in a single processor, sacrificing *comp* and *comm* to guarantee optimal *sync* $= O(1)$

The prefix aggregation problem

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

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

More generally: associative operator $\bullet$ with identity ϵ (introduced formally if missing)

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

$b_0 = a_0$

$b_1 = a_0 \bullet a_1$

$b_2 = a_0 \bullet a_1 \bullet a_2$

$\ldots$

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

The prefix aggregation problem

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

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

More generally: associative operator $\bullet$ with identity ϵ (introduced formally if missing)

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

$b_0 = a_0$

$b_1 = a_0 \bullet a_1$

$b_2 = a_0 \bullet a_1 \bullet a_2$

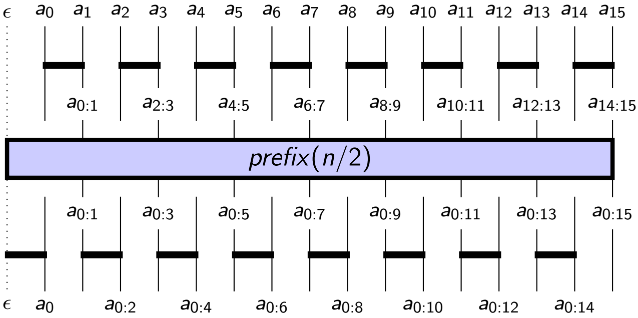
$\ldots$

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

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

The prefix aggregation circuit [Ladner, Fischer: 1980]

prefix(n)



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

The underlying dag is called the prefix dag

The prefix aggregation circuit (contd.)

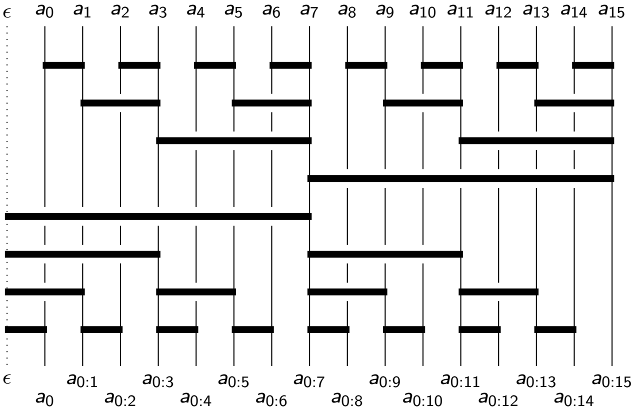
prefix(n)

n inputs

n outputs

size $2n - 2$

depth $2 \log n$



Parallel prefix aggregation

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)

Parallel prefix aggregation

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 $O(n/p)$

Parallel prefix aggregation

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 $O(n/p)$

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

Required slackness $n \geq p^2$

Generic first-order linear recurrence

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

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

$c_0 = a_0$

$c_1 = a_1 + b_1 \cdot c_0$

$c_2 = a_2 + b_2 \cdot c_1$

$\ldots$

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

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

Let $A_i = \begin{bmatrix} 1 & 0 \\ a_i & b_i \end{bmatrix} \quad C_i = \begin{bmatrix} 1 \\ c_i \end{bmatrix} \quad A_i C_{i-1} = \begin{bmatrix} 1 & 0 \\ a_i & b_i \end{bmatrix} \begin{bmatrix} 1 \\ c_{i-1} \end{bmatrix} = \begin{bmatrix} 1 \\ c_i \end{bmatrix} = C_i$

$C_0 = A_0 \cdot C_{-1}$

$C_1 = A_1 A_0 \cdot C_{-1}$

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

$\dots$

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

Computing the generic first-order linear recurrence:

- suffix aggregation (= prefix aggregation in reverse) of $[A_{n-1}, \ldots, A_0]$, with operator defined by 2×2-matrix multiplication
- each suffix aggregate multiplied by C_{-1}
- output obtained as bottom component of resulting 2-vectors

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

Operators $+$, $\cdot$ can be replaced by any given $\oplus$, $\odot$, where

- operators $\oplus$, $\odot$ computable in time $O(1)$
- 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 possible $\oplus$, $\odot$:

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

Polynomial evaluation

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

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

Polynomial evaluation

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

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

Evaluating the polynomial:

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

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

Polynomial evaluation by Horner's rule

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

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

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

$y_0 = a_{n-1}$

$y_1 = a_{n-2} + x \cdot y_0$

$y_2 = a_{n-3} + x \cdot y_1$

$\ldots$

$y_{n-1} = a_0 + x \cdot y_{n-2}$

Generic first-order linear recurrence over $[a_{n-1}, \ldots, a_0]$, $[x, x, \ldots, x]$

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

Binary addition via Boolean logic

$x + y = z \quad x, y, z$ represented as binary 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

Boolean operators as primitives: 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: $x_i + y_i + c_{i-1} = z_i + 2c_i \quad 0 \le i < n$

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

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

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

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

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

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

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

$$\ldots \qquad \ldots$$

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

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

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

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

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

Compute

- c as generic first-order linear recurrence with inputs u, v and operators $\vee$, $\wedge$: size $O(n)$, depth $O(\log n)$
- 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

The integer sorting problem

Given $a = [a_0, \ldots, a_{n-1}]$, arrange elements of a in increasing order

$a_i \in \{0, 1, \ldots, n-1\} \quad 0 \leq i < n$

Elements of a assumed to be distinguishable keys even if values equal

A bucket: subset of keys with equal values

Stable integer sorting: order of keys preserved within each bucket

Sequential work $O(n)$ e.g. by bucket sort or counting sort

Parallel integer sorting

Initially assume $a_i \in \{0, 1, \ldots, \frac{n}{p} - 1\}$, i.e. $\frac{n}{p}$ buckets

Every processor

- reads subarray of a of size n/p
- counts subarray elements in each bucket

A designated processor

- adds subarray counts for each bucket (array combining)
- determines bucket boundaries, broadcasts them (array broadcasting)

Every processor

- writes each element at appropriate offset from bucket boundary

Parallel integer sorting (contd.)

Now consider $a_i \in \{0, 1, \ldots, p-1\}$, i.e. p buckets

Consider keys as pairs: $a_i = \left(a_i \bmod \frac{n}{p}, a_i \operatorname{div} \frac{n}{p}\right)$

Perform 2-fold radix sort on pairs:

- left ("least significant") position
- right ("most significant") position

In each position, perform stable sorting over range $\left\{0, 1, \ldots, \frac{n}{p} - 1\right\}$

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

Required slackness $n \geq p^2$

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 $\omega^n = 1$

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: given complex vector a, compute b, where $F_{n,\omega} \cdot a = b$, and $F_{n,\omega} = \left[\omega^{ij}\right]_{i,j=0}^{n-1}$ is the Fourier matrix

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

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

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: given complex vector a, compute b, where $F_{n,\omega} \cdot a = b$, and $F_{n,\omega} = \left[\omega^{ij}\right]_{i,j=0}^{n-1}$ is the Fourier matrix

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

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

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

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: given complex vector a, compute b, where $F_{n,\omega} \cdot a = b$, and $F_{n,\omega} = \left[\omega^{ij}\right]_{i,j=0}^{n-1}$ is the Fourier matrix

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

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

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

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

The Fast Fourier Transform (FFT) algorithm

[Gauss: 1805; . . . ; Cooley, Tukey: 1965]

Four-step FFT: assume $n = m^2$

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

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

The Fast Fourier Transform (FFT) algorithm

[Gauss: 1805; . . . ; Cooley, Tukey: 1965]

Four-step FFT: assume $n = m^2$

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

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

$B_{s,t} = \sum_{u,v} \omega^{(ms+t)(mu+v)} A_{u,v} = \sum_{u,v} \omega^{msv+tv+mtu} A_{u,v} =$
$\sum_{v} \big((\omega^m)^{sv} \cdot \omega^{tv} \cdot \sum_{u} (\omega^m)^{tu} A_{u,v}\big)$, thus $B = F_{m,\omega^m} \cdot (G_{m,\omega} \circ (F_{m,\omega^m} \cdot A))^T$

$F_{m,\omega^m} \cdot A$ is m independent DFTs of size m on each column of A

$G_{m,\omega} = \left[\omega^{tv}\right]_{t,v=0}^{m-1}$ is the twiddle-factor matrix

Operator $\circ$ is the Hadamard product (elementwise matrix multiplication)

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

$B = F_{m,\omega^m} \cdot (G_{m,\omega} \circ (F_{m,\omega^m} \cdot A))^T$

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

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

We reduced DFT of size $n = m^2$ to DFTs of size m

Similarly, we can reduce DFT of size $n = kl$ to DFTs of sizes k and l

Assume $n = 2^{2^r}$, then $m = 2^{2^{r-1}}$

By recursion, we have the FFT circuit

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

We reduced DFT of size $n = m^2$ to DFTs of size m

Similarly, we can reduce DFT of size $n = kl$ to DFTs of sizes k and l

Assume $n = 2^{2^r}$, then $m = 2^{2^{r-1}}$

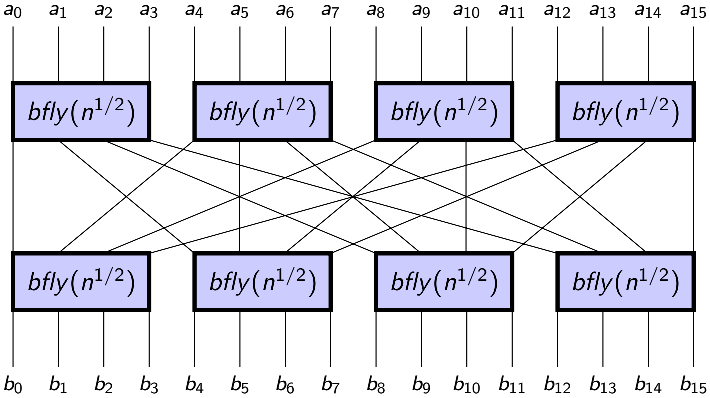
By recursion, we have the FFT circuit

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

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

The FFT circuit

$bfly(n)$



The underlying dag is called butterfly dag

The FFT circuit and the butterfly dag (contd.)

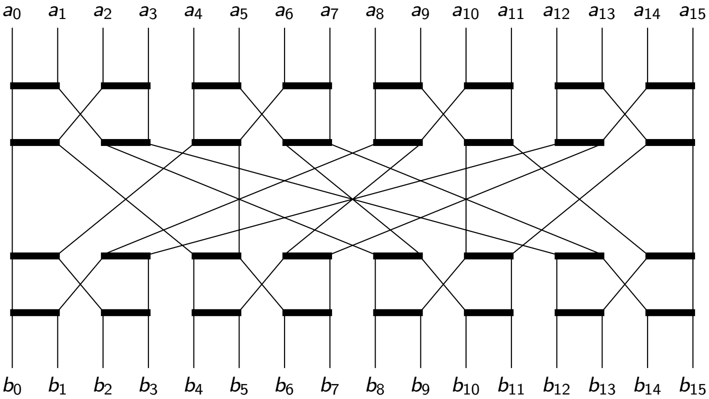
$bfly(n)$

n inputs

n outputs

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

depth $\log n$



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

Parallel butterfly computation

To compute *bfly*(n), every processor

- reads inputs for $\frac{n^{1/2}}{p}$ blocks from top layer; computes blocks; writes outputs
- reads inputs for $\frac{n^{1/2}}{p}$ blocks from bottom layer; computes blocks; writes outputs

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

Parallel butterfly computation

To compute $bfly(n)$, every processor

- reads inputs for $\frac{n^{1/2}}{p}$ blocks from top layer; computes blocks; writes outputs
- reads inputs for $\frac{n^{1/2}}{p}$ blocks from bottom layer; computes blocks; writes outputs

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

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

Required slackness: $n \geq p^2$

Polar coding [Arikan: 2009]

Incorporated in 5G mobile communication standard

Assume binary erasure channel: $x \rightsquigarrow \begin{cases} x & \text{with } \Pr = 1 - \pi \\ ? & \text{with } \Pr = \pi \end{cases}$ $\quad 0 < \pi < 1$

Plain code

$(a_0, a_1, a_2, \ldots) \mapsto (a_0, a_1, a_2, \ldots)$

Recovering each a_i with $p_i = 1 - \pi$

Polar code

$(a_0, a_1) \mapsto (a_0 \oplus a_1, a_1)$, where '$\oplus$' = 'exclusive or' = '$+ \pmod 2$'

Recovering a_0:

$$\left.\begin{array}{l} a_0 \oplus a_1 \\ a_1 \end{array}\right\} \rightsquigarrow (a_0 \oplus a_1) \oplus a_1 = a_0 \text{ with } p_0 = (1-\pi)^2 < 1 - \pi$$

Recovering a_1, conditioned on recovery of a_0:

$$\left.\begin{array}{l} a_0 \oplus a_1 \rightsquigarrow (a_0 \oplus a_1) \oplus a_0 \\ a_1 \end{array}\right\} = a_1 \text{ with } p_1 = 1 - \pi^2 > 1 - \pi$$

Polar code

$(a_0, \ldots, a_{n-1})$ encoded by recursion

Recovering a_i, conditioned on recovery of $a_0, \ldots, a_{i-1}$:

$$p_i \to \begin{cases} 0 & \text{near-useless} \\ 1 & \text{near-perfect} \end{cases} \qquad \frac{1}{n} \sum_{0 \le i < n} p_i = 1 - \pi$$

Encoding:

- frozen bits $a_i = 0$ where $p_i \to 0$
- information bits a_i where $p_i \to 1$

Decoding: successively $a_0, \ldots, a_{n-1}$, substituting known frozen bits

$(a_0, \ldots, a_{n-1})$: $\sim \pi n$ frozen bits, $\sim (1-\pi)n$ information bits

Encoding circuit: $\mathit{bfly}(n)$

- operator $x, y \mapsto x \oplus y$
- size $\frac{n \log n}{2}$, depth $\log n$

Decoding circuit (successive cancellation): $\mathit{bfly}(n)$, traversed in a complex pattern; every node activated three times at different times/directions

- operators $x, y \mapsto x \oplus y$ and $x, x \mapsto x$ (boosting probability of recovering x)
- size $O(n \log n)$, depth $O(n)$

Alternative decoding: belief propagation (work vs parallelism)

Open problem: polar decoding in size $n^{O(1)}$, depth $O(\log n)$?

The ordered 2D grid dag

$grid_2(n)$

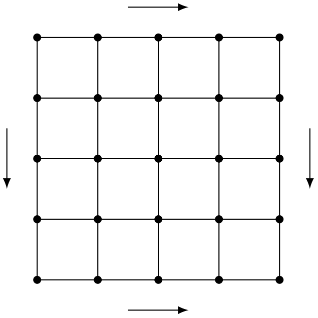
nodes arranged in an $n \times n$ grid

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 $2n - 1$



The ordered 2D grid dag

$grid_2(n)$

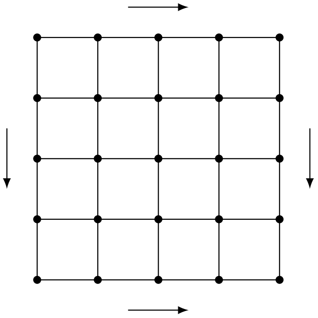
nodes arranged in an $n \times n$ grid

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

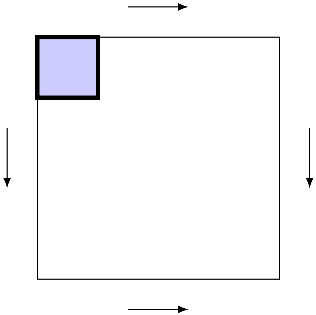
Sequential work $O(n^2)$

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

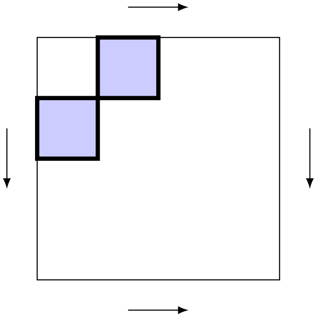


Parallel ordered 2D grid computation

$grid_2(n)$

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

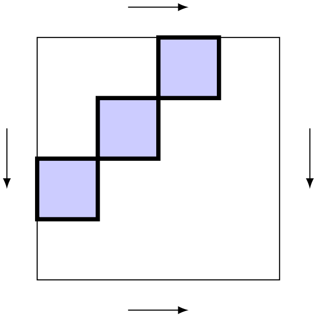


Parallel ordered 2D grid computation

$grid_2(n)$

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

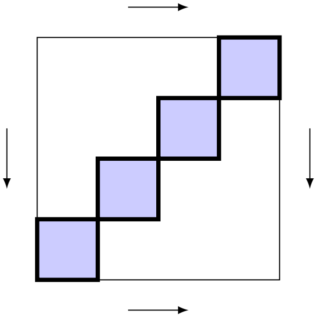


Parallel ordered 2D grid computation

$grid_2(n)$

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

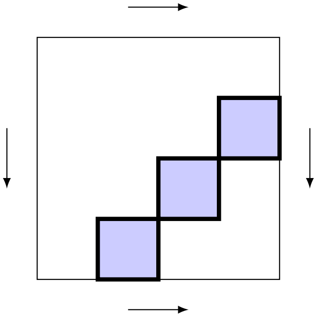


Parallel ordered 2D grid computation

$grid_2(n)$

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

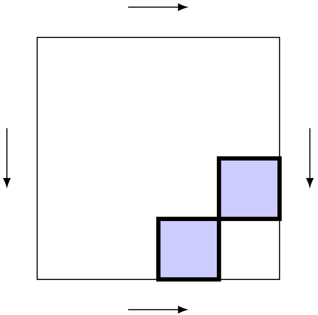


Parallel ordered 2D grid computation

$grid_2(n)$

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

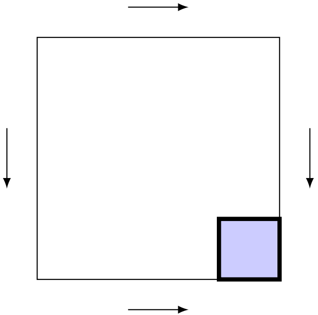


Parallel ordered 2D grid computation

$grid_2(n)$

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



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

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

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

$comp = O(n^2/p)$ $comm = O(n)$ $sync = O(p)$

Required slackness $n \geq p$

The ordered 3D grid dag

$grid_3(n)$

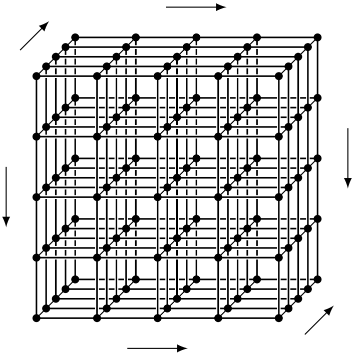
nodes arranged in an $n \times n \times n$ grid

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

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

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

size n^3 depth $3n - 2$



The ordered 3D grid dag

$grid_3(n)$

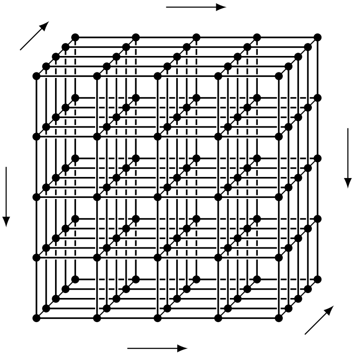
nodes arranged in an $n \times n \times n$ grid

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

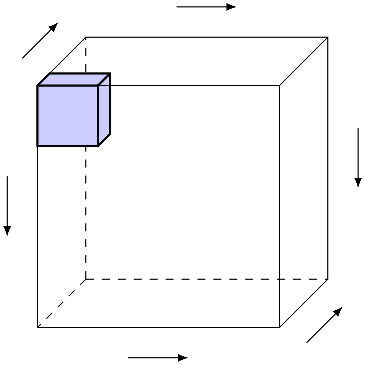
Sequential work $O(n^3)$

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

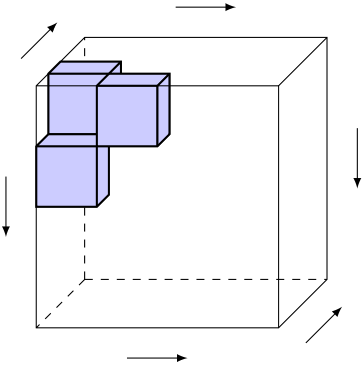


Parallel ordered 3D grid computation

$grid_3(n)$

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

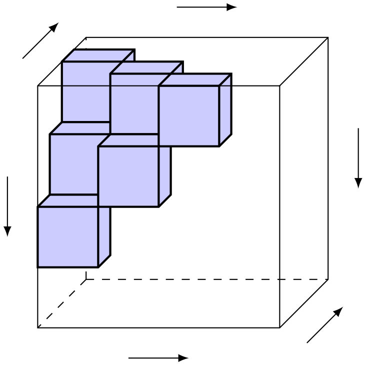


Parallel ordered 3D grid computation

$grid_3(n)$

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

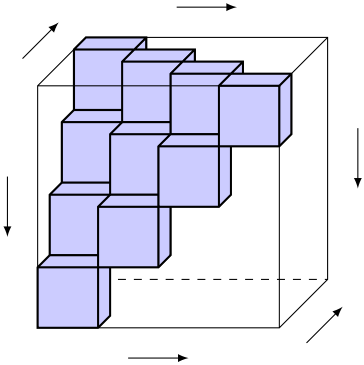


Parallel ordered 3D grid computation

$grid_3(n)$

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

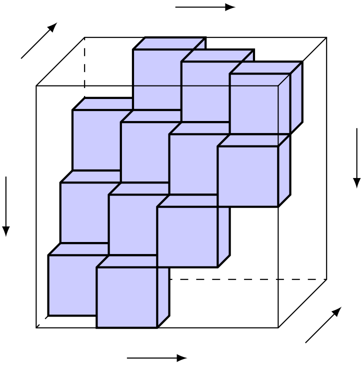


Parallel ordered 3D grid computation

$grid_3(n)$

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

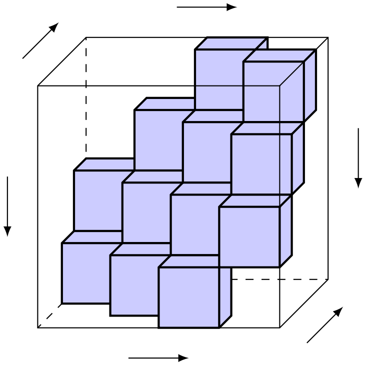


Parallel ordered 3D grid computation

$grid_3(n)$

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

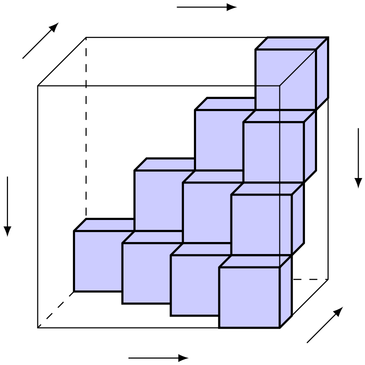


Parallel ordered 3D grid computation

$grid_3(n)$

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

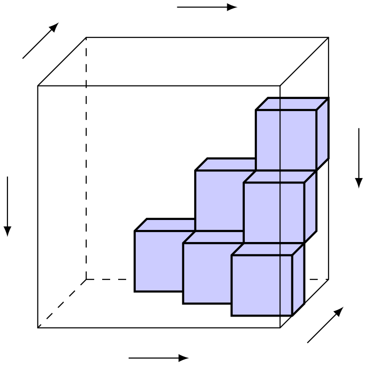


Parallel ordered 3D grid computation

$grid_3(n)$

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

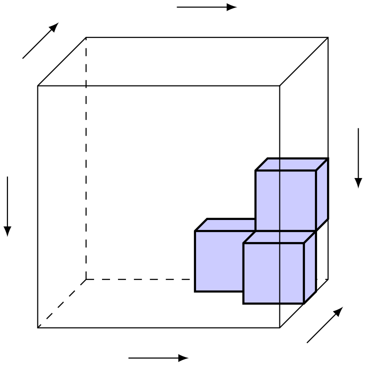


Parallel ordered 3D grid computation

$grid_3(n)$

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

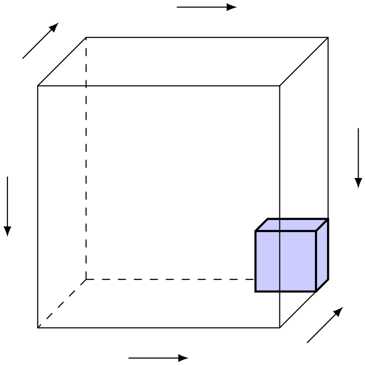


Parallel ordered 3D grid computation

$grid_3(n)$

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



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

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

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

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

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

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

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"

Let a, b be strings of characters

A subsequence of string a is obtained by deleting some (possibly none, or all) characters from a

The longest common subsequence (LCS) problem: find the longest string that is a subsequence of both a and b

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

Let a, b be strings of characters

A subsequence of string a is obtained by deleting some (possibly none, or all) characters from a

The longest common subsequence (LCS) problem: find the longest string that is a subsequence of both a and b

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

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

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

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

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

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

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

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						
E	0						
S	0						
I	0						
G	0						
N	0						

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						
S	0						
I	0						
G	0						
N	0						

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						
I	0						
G	0						
N	0						

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						
G	0						
N	0						

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						
N	0						

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 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 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

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

Parallel LCS computation

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

$comp = O(n^2/p)$ $comm = O(n)$ $sync = O(p)$

Parallel LCS computation

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

$comp = O(n^2/p)$ $comm = O(n)$ $sync = O(p)$

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

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

Parallel LCS computation

Solve the more general semi-local LCS problem:

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

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

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

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

Sequential time still $O(n^2)$

Parallel LCS computation (cont.)

Communication vs synchronisation tradeoff [T: NEW]

$comp = O(n^2/p)$	$comm = O(np^\epsilon)$	$sync = O(\log(1/\epsilon))$

for all $\epsilon > 0$

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

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

Basic parallel algorithms

Discussion

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

Realistic slackness requirements: $n \gg p$, typically $n = \Omega(poly(p))$

Goals:

- $comp = O(comp_{seq}/p)$
- *comm* scales down with increasing p
- *sync* constant or function of p, independent of n

Basic parallel algorithms
Discussion

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

Realistic slackness requirements: $n \gg p$, typically $n = \Omega(poly(p))$

Goals:

- $comp = O(comp_{seq}/p)$
- *comm* scales down with increasing p
- *sync* constant or function of p, independent of n

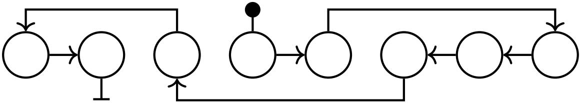
Challenges:

- efficient algorithms: ongoing
- strong lower bounds: recently by Ballard et al, Bilardi et al, others
- further objectives: resilience, privacy
- model evolution: e.g. relax $comp = O(comp_{seq}/p)$ to push down *sync*

Linked list: array of n nodes

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

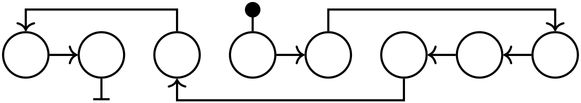
Nodes may be placed in array in an arbitrary order



Linked list: array of n nodes

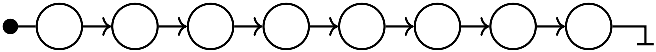
Each node contains data and a pointer to (= index of) successor node

Nodes may be placed in array in an arbitrary order



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

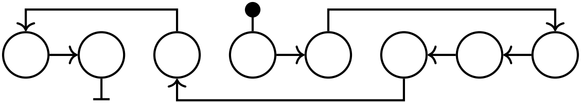
- a pointer can be followed in time $O(1)$
- no global ranks/indexing/comparison



Linked list: array of n nodes

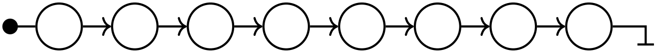
Each node contains data and a pointer to (= index of) successor node

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

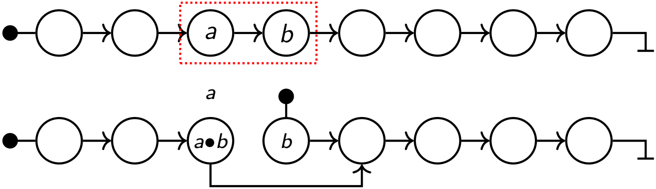


Pointer jumping at node u

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

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

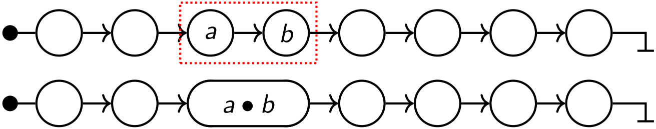
$a \leftarrow data(u) \qquad b \leftarrow data(v) \qquad data(u) \leftarrow a \bullet b$



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

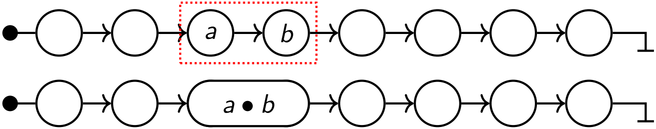
$succ(u) \leftarrow v \qquad data(u) \leftarrow a \qquad data(v) \leftarrow b$

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



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

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



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

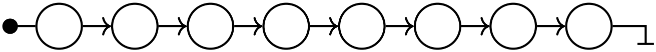
The list contraction problem: reduce the list to a single node by successive node merging (note the result is independent on the merging order)

The list expansion problem: restore the original list by successive node splitting, reversing contraction

Problems solved by list contraction/expansion:

- list ranking
- list prefix aggregation

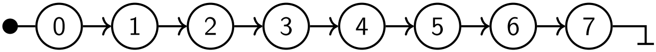
List ranking



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

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

Expansion phase: each node u holds

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

List ranking (contd.)

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

Merging $v, w \mapsto u$: $l(u) \leftarrow l(v) + l(w)$ keep $l(v)$, $l(w)$

Contracted list: node t $l(t) = n$ $r(t) \leftarrow 0$

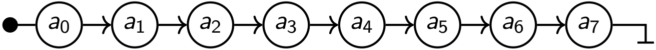
Splitting $u \mapsto v, w$:

restore $l(u)$, $l(v)$ $r(v) \leftarrow r(u)$ $r(w) \leftarrow r(v) + l(v)$

Eventually, for each node u: $l(u) = 1$ $r(u) = rank(u)$

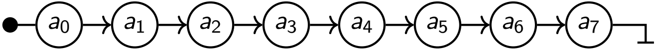
List prefix aggregation

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



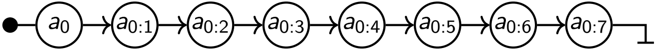
List prefix aggregation

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



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

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



Note the solution should be independent of the merging order

List prefix aggregation (contd.)

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

Contraction phase: each node u holds

- aggregate $l(u)$ of corresponding sublist

Expansion phase: each node u holds

- aggregate $l(u)$ of corresponding sublist (as before)
- aggregate $r(u)$ of list prefix before the sublist

List prefix aggregation (contd.)

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

Merging $v, w \mapsto u$: $l(u) \leftarrow l(v) \bullet l(w)$ keep $l(v)$, $l(w)$

Contracted list: node t $l(t) = b_{n-1}$ $r(t) \leftarrow \epsilon$

Splitting $u \mapsto v, w$:
restore $l(u)$, $l(v)$ $r(v) \leftarrow r(u)$ $r(w) \leftarrow r(v) \bullet l(v)$

Eventually, for each node u: $l(u) = a_{rank(u)}$ $r(u) = b_{rank(u)}$

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

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

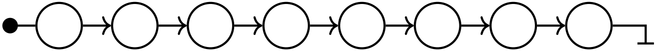
In general, only need to consider contraction phase (expansion by symmetry)

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

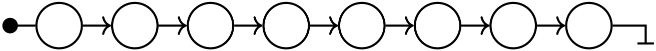
Therefore, we need either node cloning, or efficient symmetry breaking

Wyllie's mating [Wyllie: 1979]

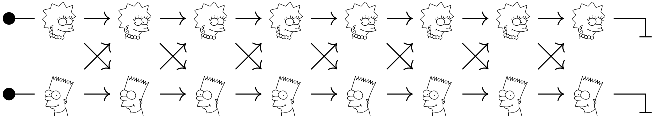


Wyllie's mating

[Wyllie: 1979]

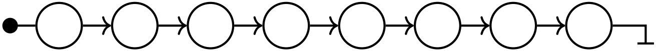


Clone every node, label copies "forward"  and "backward" 

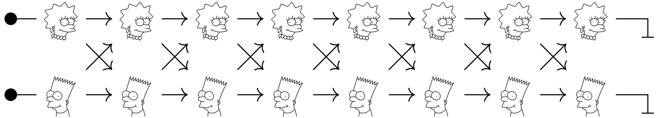


Wyllie's mating

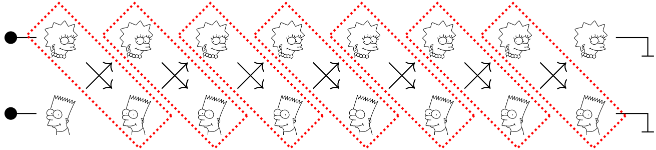
[Wyllie: 1979]



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



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



Parallel list contraction by Wyllie's mating

In the first round, every processor

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

Subsequent rounds similar

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

Parallel list contraction:

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

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

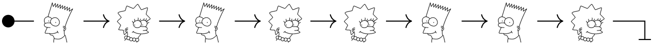
Total work $O(n \log n)$, not optimal vs. sequential work $O(n)$

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

Random mating

[Miller, Reif: 1985]

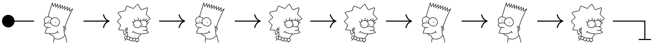
Label every node "forward"  or "backward"  independently with probability $\frac{1}{2}$



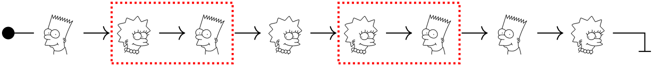
Random mating

[Miller, Reif: 1985]

Label every node "forward"  or "backward"  independently with probability $\frac{1}{2}$



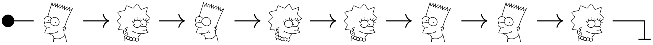
Merge mating node pairs



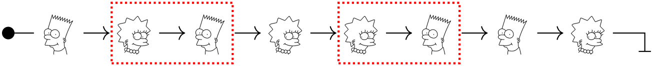
Random mating

[Miller, Reif: 1985]

Label every node "forward"  or "backward"  independently with probability $\frac{1}{2}$



Merge mating node pairs



On average $\frac{n}{2}$ nodes mate, therefore new list has expected size $\frac{3n}{4}$

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

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

Parallel list contraction by random mating

In the first round, every processor

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

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

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

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

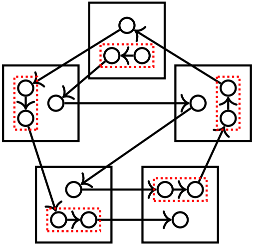
$comp = O(n/p)$ whp | $comm = O(n/p)$ whp | $sync = O(\log p)$

Required slackness $n \geq p^2$

Block mating

Will mate nodes deterministically

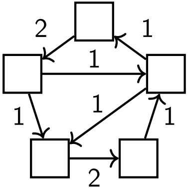
Contract local chains (if any)



Build distribution graph:

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

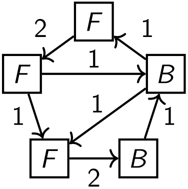
Each processor holds a supernode's outgoing edges



Block mating (contd.)

Designated processor collects the distribution graph

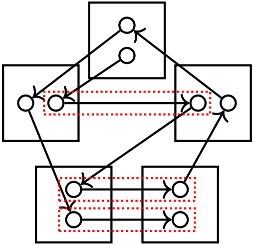
Label every supernode F ("forward") or B ("backward"), so that $\sum_{i \in F, j \in B} w(i,j) \geq \frac{1}{4} \cdot \sum_{i,j} w(i,j)$ by a sequential greedy algorithm



Distribute supernode labels to processors

Merge mating node pairs

By construction of supernode labelling, $\geq \frac{n}{2}$ nodes mate, therefore new list has size $\leq \frac{3n}{4}$



Parallel list contraction by block mating

In the first round, every processor

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

The designated processor collects distribution graph, computes and distributes labels

Parallel list contraction by block mating (contd.)

Continuing the first round, every processor

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

Subsequent rounds similar, on a list of decreasing size

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

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

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

Required slackness $n \geq p^4$

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

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

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

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

Is list contraction really necessary for list k-colouring?

Can list k-colouring be done more efficiently?

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

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

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

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

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

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

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

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

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

Deterministic coin tossing [Cole, Vishkin: 1986]

Given a k-colouring, $k > 6$

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*: $O(n/p)$

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

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: $10^{81} \approx 2^{270}$

$\log^* 2^{270} = \log^* 2^{65536} = \log^* 2^{2^{2^{2^2}}} = 5$

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: $O(\log^* p)$

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

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

The sorting problem

Given $a = [a_0, \ldots, a_{n-1}]$, arrange elements of a in increasing order

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

Assume the comparison model: primitives $<$, $>$, no arithmetic or bit operations on a_i

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

The sorting problem

Given $a = [a_0, \ldots, a_{n-1}]$, arrange elements of a in increasing order

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

Assume the comparison model: primitives $<$, $>$, no arithmetic or bit operations on a_i

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

Parallel sorting based on an AKS sorting network

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

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

Every processor

- reads subarray of a of size n/p and sorts it sequentially
- selects from it p samples from base index 0 at steps n/p^2

Samples define p equal-sized, contiguous blocks in local subarray

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

Every processor

- reads subarray of a of size n/p and sorts it sequentially
- selects from it p samples from base index 0 at steps n/p^2

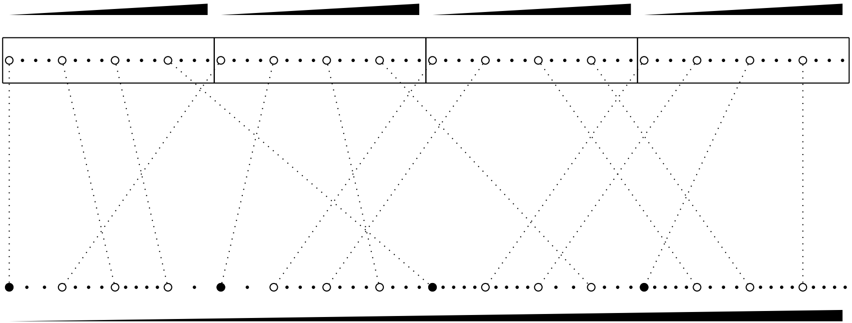
Samples define p equal-sized, contiguous blocks in local subarray

A designated processor

- collects all p^2 samples and sorts them sequentially
- selects from them p splitters from base index 0 at steps p
- broadcasts the splitters

Splitters define p unequal-sized, rank-contiguous buckets in global array a

Parallel sorting by regular sampling (contd.)



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

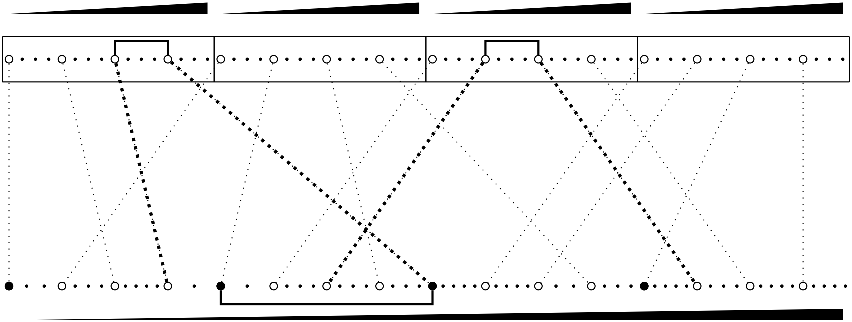
We will need to prove that bucket sizes, although not uniform, are still well-balanced ($\leq 2n/p$)

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

Required slackness $n \geq p^3$

Parallel sorting by regular sampling (contd.)

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



Parallel sorting by regular sampling (contd.)

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 $\leq (p + p) \cdot n/p^2 = 2n/p$ □

The selection problem

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

Selection by median sampling [Blum+: 1973]

Proceed in rounds. In the first 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: $n \leftarrow n/5$, $k \leftarrow n/10$
- find rank l of median-of-medians in array a by linear search

If $l = k$, return a_l; otherwise, eliminate elements on the wrong side of median-of-medians; adjust size and target rank for next round:

- if $l < k$, discard all $a_i \leq a_l$; adjust $n \leftarrow n - l - 1$, $k \leftarrow k - l - 1$
- if $l > k$, discard all $a_i \geq a_l$; adjust $n \leftarrow l$, k unchanged

Subsequents rounds similar, with adjusted n, k

Selection by median sampling (contd.)

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

Selection by median sampling (contd.)

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 $\square$

With each round, array shrinks exponentially

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

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

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 local subarray, discarding elements on wrong side of median-of-medians
- writes remaining elements

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

Subsequents rounds similar, with adjusted n, k

Parallel selection by median sampling (contd.)

Overall algorithm:

- 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

Parallel selection by median sampling (contd.)

Overall algorithm:

- 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

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

Parallel selection by regular sampling (generalised median sampling)

In the first round, every processor

- reads a subarray of size n/p
- selects from it $s = O(1)$ samples from base rank 0 at rank steps $\frac{n}{sp}$

Splitters define s equal-sized, rank-contiguous blocks in local subarray

A designated processor

- collects all sp samples
- selects from them s splitters from base rank 0 at rank steps p
- broadcasts the splitters

Splitters define s unequal-sized, rank-contiguous buckets in global array a

Every processor

- determines rank of every splitter in local subarray

Parallel 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, discarding elements outside target bucket
- writes remaining elements

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

Subsequents rounds similar, with adjusted n, k

Parallel selection by accelerated regular sampling

In the original median sampling, sampling frequency $s = 2$ fixed across all rounds (samples at base rank 0 and local median rank $\frac{n}{2p}$); array shrinks exponentially

We now increase s from round to round, accelerating array reduction; array now shrinks superexponentially

Round 0: selecting samples and determining splitter ranks in time $O\left(\frac{n \log s}{p}\right)$; set $s = 2$, time $O(n/p)$

Round 1: array size $O(n/s)$, we can afford sampling frequency 2^s

Round 2: ...

Parallel selection by accelerated regular sampling

Overall algorithm:

- 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

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

Parallel selection

$comp = O(n/p)$ $comm = O(n/p)$

$sync = O(\log p)$ [Ishimizu+: 2002]

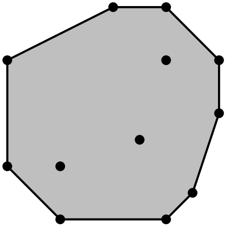
$sync = O(\log \log n)$ [Fujiwara+: 2000]

$sync = O(1)$ whp randomised [Gerbessiotis, Siniolakis: 2003]

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

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 $\operatorname{conv} A$ is the smallest convex set containing A

$\operatorname{conv} A$ 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 $\operatorname{conv} A$

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

$d = 2$

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

The (discrete) convex hull problem

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

Output (a finite representation of) conv a

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

E.g. in 3D: 1D vertices, 2D edges, 3D facets

Output must be structured, i.e. should give

- for $d = 2$, all vertex-edge incidence pairs; every vertex should "know" its both neighbours
- for general d, all incidence pairs between a k-D and a $(k+1)$-D face

The (discrete) convex hull problem (contd.)

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

The (discrete) convex hull problem (contd.)

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

The (discrete) convex hull problem (contd.)

$d = 2$: $\leq n$ 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...

The (discrete) convex hull problem (contd.)

$d = 2$: $\leq n$ 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...

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

$d = 4, 5$: output size $O(n^2)$

$d = 6, 7$: output size $O(n^3)$

...

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

The (discrete) convex hull problem (contd.)

$d = 2$: $\leq n$ 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...

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

$d = 4, 5$: output size $O(n^2)$

$d = 6, 7$: output size $O(n^3)$

...

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

Sequential work $O(n \log n)$: Graham's scan (2D); 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

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

Claim:

ϵ-approximation for A is ϵ-net for A. (The converse does not hold!)

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

ϵ-net for δ-approximation for A is $(\epsilon + \delta)$-net for A

Claim:

ϵ-approximation for A is ϵ-net for A. (The converse does not hold!)

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

ϵ-net for δ-approximation for A is $(\epsilon + \delta)$-net for A

Proof: Easy by definitions; independent of d. □

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

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

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

Claim. A $1/r$-net for A of size $\leq 2r$ exists, 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

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

Claim. A $1/r$-net for A of size $\leq 2r$ exists, 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

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

Claim. A $1/r$-net for A of size $\leq 2r$ exists, 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 $|E| \leq 2r$ □

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

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 $O(n \log n)$.

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

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

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

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

Parallel 2D hull computation by generalised regular sampling

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

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)

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

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 containg any splitters. Therefore, bucket size is at most $2 \cdot (2/p) \cdot n = 4n/p$.

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

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

Requires slackness $n \geq p^3$

Further parallel algorithms

Convex hull

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

Claim: $1/r$-net for A of size $O(r)$ can be obtained in seq time $O(rn \log n)$.
[Brönnimann, Goodrich: 1995]

Claim: $1/r$-approximation for A of size $O\big(r^3 (\log r)^{O(1)}\big)$ can be obtained in seq time $O(n \log r)$. [Matoušek: 1992]

Better approximations are possible, but are slower to compute
[Matoušek: 1992, Mustafa+: 2018]

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

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

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 $2 \cdot (2/p) \cdot n = 4n/p$.

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

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

Requires slackness $n \gg p$

The suffix sorting problem

Given string $a = a_0 \dots a_{n-1}$ $\quad a_i \in \{0, 1, \dots, n-1\} \quad 0 \leq i < n$

Sort all suffixes of a in lexicographic order (implicitly, by returning ranks)

Character sorting: time $O(n)$ e.g. by counting sort

Naive suffix sorting: time $O(n^2)$ by n-fold radix sort, performing character sorting successively in every position from least to most significant

Suffix sorting by DC mod 3 sampling [Kärkkäinen, Sanders: 2003]

Difference cover (DC) modulo 3, aka skew algorithm

Assume no suffix of a is a prefix of another suffix (otherwise, append $-\infty$ as a sentinel)

Denote $a = [01234\ldots]$ $a_i = [i]$

Consider 3-substrings as super-characters: [012], [123], [234], ...

Sort all distinct super-characters by 3-fold radix sort; substitute each by its rank

Suffix sorting by DC mod 3 sampling (contd.)

Sample indices: $i \equiv 0, 1 \bmod 3$, but not 2 mod 3

Sample suffixes: [012...], [123...], ~~[234...]~~, [345...], [456...], ~~[567...]~~, ...

$b =$ [012][345][678]...[123][456][789]...~~[234][567][8 9 10]...~~

String b formed by concatenation of two initial sample suffixes of a, each broken up into super-characters

$length(b) = 2 \cdot n/3 = 2n/3$ super-characters

For comparison purposes, $\{$suffixes of $b\} = \{$sample suffixes of $a\}$

Suffix sorting by DC mod 3 sampling (contd.)

Will sort separately {sample suffixes of a}, {non-sample suffixes of a}

Sort sample suffixes:

- suffix sorting on b by recursion

Comparing non-sample suffixes

- as pairs (character, sample suffix) in time $O(1)$, eg.
  $[2345...] = [2][345...] = ([2], [345...])$ vs
  $[5678...] = [5][678...] = ([5], [678...])$

Sort non-sample suffixes:

- 2-fold radix sort on pairs (character, sample suffix)

Suffix sorting by DC mod 3 sampling (contd.)

We have two ordered sets: {sample suffixes}, {non-sample suffixes}

Comparing any suffixes

- as pairs (super-character, sample suffix) in time $O(1)$, eg.
  $[012\ldots] = [0][123\ldots] = ([012], [123\ldots])$ vs
  $[234\ldots] = [2][345\ldots] = ([234], [345\ldots])$
  $[123\ldots] = [12][345\ldots] = ([123], [345\ldots])$ vs
  $[234\ldots] = [23][456\ldots] = ([234], [456\ldots])$

Merge all suffixes

- comparison-based merging on pairs

Overall running time $T(n) = O(n) + T(2n/3) = O(n)$

Parallel suffix sorting by DC mod 3 sampling

$a = a_0 \dots a_{n-1}$

At the top recursion level, every processor

- reads substring of a of length n/p
- sorts super-characters locally by 3-fold radix sort (or sequential suffix sorting)

The processors collectively

- sort super-characters globally by regular sampling
- form string b
- sort sample suffixes of a by recursion on b
- sort non-sample suffixes of a by 2-fold radix sort

Parallel suffix sorting by DC mod 3 sampling (contd.)

Every processor

- merges sample vs non-sample suffixes locally

The processors collectively

- merge sample vs non-sample suffixes globally by regular sampling

Subsequent recursion levels similar, with n adjusted

Parallel suffix sorting by DC mod 3 sampling (contd.)

Overall algorithm:

- perform $\log_{3/2} p$ recursion levels of suffix sorting by DC mod 3 sampling, obtaining a string of length n/p
- a designated processor collects the resulting string and performs suffix sorting sequentially

Parallel suffix sorting by DC mod 3 sampling (contd.)

Overall algorithm:

- perform $\log_{3/2} p$ recursion levels of suffix sorting by DC mod 3 sampling, obtaining a string of length n/p
- a designated processor collects the resulting string and performs suffix sorting sequentially

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

Suffix sorting by DC mod d sampling

Difference cover (DC) modulo d: set S of integers mod d, such that for all i mod d, there are $j, k \in S$ with $k - j = i$ mod d

Examples:

DC mod 3: $\{0, 1\}$

i	0	1	2
j	0	0	1
k	0	1	0

DC mod 13: $\{0, 1, 4, 6\}$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
j	0	0	4	1	0	1	0	6	6	4	4	6	1
k	0	1	6	4	4	6	6	0	1	0	1	4	0

Suffix sorting by DC mod d sampling (contd.)

Claim: For any d, there is a DC mod d of size $O(d^{1/2})$

[Colbourn, Ling: 2000]

DC mod 3 algorithm can be generalised to DC mod d for any $d \geq 3$

[Kärkkäinen, Sanders: 2003]

Given d, consider d-substrings as super-characters

Fix a DC mod d as sample indices

Sample indices define sample suffixes, sample super-characters

Sort all distinct super-characters by d-fold radix sort; substitute each by its rank

Suffix sorting by DC mod d sampling (contd.)

String b formed by concatenation of $O(d^{1/2})$ initial sample suffixes of a, each broken up into sample super-characters

Overall, b is of length $O(d^{1/2}) \cdot n/d = O(n/d^{1/2})$ super-characters

For comparison purposes, $\{\text{suffixes of } b\} = \{\text{sample suffixes of } a\}$

Sort sample suffixes

- suffix sorting on b by recursion

Sort non-sample suffixes in $<d$ separate subsets according to index mod d

- 2-fold radix sort on a for each non-sample index mod d

Suffix sorting by DC mod d sampling (contd.)

We have $\leq d$ ordered sets of suffixes:

- $\{$sample suffixes$\}$
- $\{$non-sample suffixes at index mod $d = i\}$ for each non-sample i

Comparing any suffixes

- as pairs (super-character, sample suffix) in time $O(1)$

Merge all suffixes

- $\leq d$-way comparison-based merging on pairs

Overall running time $T(n) = O(nd) + T(O(n/d^{1/2})) = O(nd)$

Parallel suffix sorting by accelerated DC mod d sampling

In parallel DC mod 3 sampling, modulus $d = 3$ was fixed across all levels; string shrinks exponentially

Will now increase modulus from each recursion level to the next, accelerating string reduction; string shrinks superexponentially, allowing further increase in modulus while keeping work $O(size \cdot modulus) = O(n)$

Level 0: array size n; can only afford $d = O(1)$

Level 1: array size $O\left(\frac{n}{d^{1/2}}\right)$; can now afford $d^{3/2}$

Level 2: array size $O\left(\frac{n}{d^{1/2} \cdot d^{3/4}}\right) = O\left(\frac{n}{d^{5/4}}\right)$; can now afford $d^{9/4} = d^{(3/2)^2}$

Level 3: array size $O\left(\frac{n}{d^{5/4} \cdot d^{9/8}}\right) = O\left(\frac{n}{d^{19/8}}\right)$; can now afford $d^{27/8} = d^{(3/2)^3}$

...

Level $O(\log\log p)$: array size $O(n/p)$

Parallel suffix sorting by accelerated DC mod d sampling

Overall algorithm:

- perform $O(\log \log p)$ recursion levels of suffix sorting by DC mod d sampling (with increasing d), obtaining a string of length n/p
- a designated processor collects the resulting string and performs suffix sorting sequentially

Parallel suffix sorting by accelerated DC mod d sampling

Overall algorithm:

- perform $O(\log\log p)$ recursion levels of suffix sorting by DC mod d sampling (with increasing d), obtaining a string of length n/p
- a designated processor collects the resulting string and performs suffix sorting sequentially

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

Burrows–Wheeler transform (BWT)

Given string a, compute its permutation $BWT(a)$

- sort all rotations of a lexicographically by suffix sorting
- output final character of each rotation

Characters in a similar (post-)context in a occur consecutively in $BWT(a)$

Similar contexts in $a \Rightarrow$ character runs in $BWT(a)$

String $BWT(a)$ can be efficiently compressed by

- run-length encoding
- move-to-front encoding
- entropy-based encoding (eg. Huffman, arithmetic, FSE)

BWT is the main compression method for genome sequence databases

Burrows–Wheeler transform (contd.)

a = "merry_mary_marry_me$"

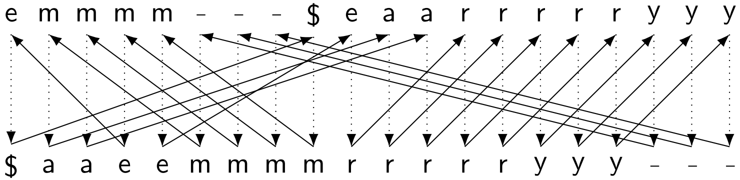
```
$merry_mary_marry_me       rry_me$merry_mary_ma
arry_me$merry_mary_m       ry_marry_me$merry_ma
ary_marry_me$merry_m       ry_mary_marry_me$mer
e$merry_mary_marry_m       ry_me$merry_mary_mar
erry_mary_marry_me$m       y_marry_me$merry_mar
marry_me$merry_mary_       y_mary_marry_me$merr
mary_marry_me$merry_       y_me$merry_mary_marr
me$merry_mary_marry_       _marry_me$merry_mary
merry_mary_marry_me$       _mary_marry_me$merry
rry_mary_marry_me$me       _me$merry_mary_marry
```

$BWT(a)$ = "emmmm___$eaarrrrryyy" = "em4_3$ea2r5y3"

Inverse Burrows–Wheeler transform (Inverse BWT)

Given string $BWT(a)$:

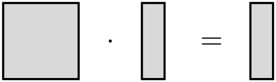
- sort characters of $BWT(a)$ by counting sort
- unfold chain of index mappings in resulting permutation



Permutation BWT is stable: preserves occurence order for each char

The matrix-vector multiplication problem

$A \cdot b = c$



A: predistributed n-matrix

b: input n-vector

c: output n-vector

$c_i = \sum_j A_{ij} \cdot b_j \quad 0 \leq i, j < n$

A assumed to be predistributed, does not count as input (motivation: iterative linear algebra methods)

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

Sequential work $O(n^2)$

The matrix-vector multiplication circuit

$c \leftarrow 0$

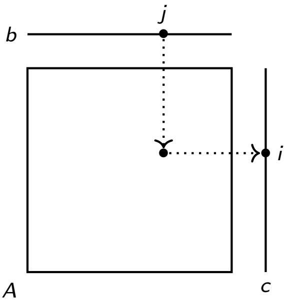
For all i, j: $c_i \overset{+}{\leftarrow} c^i_j \leftarrow A_{ij} \cdot b_j$
(adding each c^i_j to c_i asynchronously)

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

n^2 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-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} $(0 \leq I, J < p^{1/2})$

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

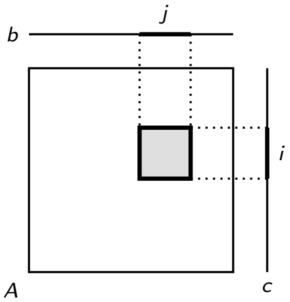
Overall, p block products $A_{IJ} \cdot b_J = c_I^J$

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

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-vector multiplication (contd.)

Initialise $c \leftarrow 0$ in external memory

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 concurrent block writing by array combining)

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

Slackness required $n \geq p$ (as $\frac{n}{p^{1/2}} \geq p^{1/2}$ needed for concurrent write)

Parallel matrix algorithms
Matrix multiplication

The matrix multiplication problem

$A \quad \cdot \quad B \quad = \quad C$

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

A, B: input n-matrices

C: output n-matrix

Parallel matrix algorithms

Matrix multiplication

The matrix multiplication problem

$A \cdot B = C$

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

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

Sequential work $O(n^3)$

A, B: input n-matrices

C: output n-matrix

The matrix multiplication circuit

$C_{ik} \leftarrow 0$

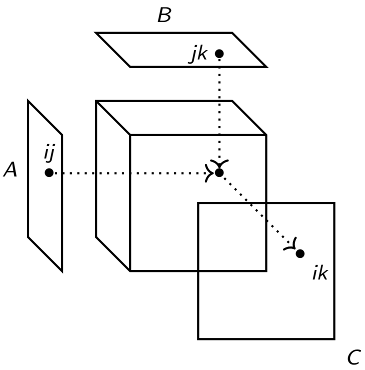
For all i, j, k: $C_{ik} \overset{+}{\leftarrow} C_{ik}^j \leftarrow A_{ij} \cdot B_{jk}$
(adding each C_{ik}^j to C_{ik} asynchronously)

$2n$ 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 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} $(0 \leq I, J, K < p^{1/3})$

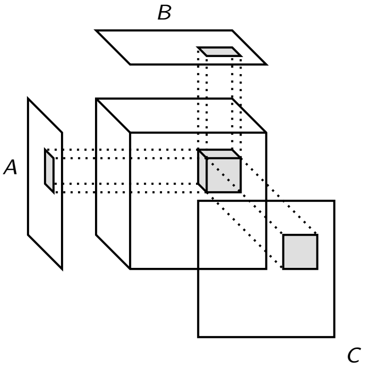
Overall, p block products $A_{IJ} \cdot B_{JK} = C_{IK}^{J}$

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

Parallel matrix multiplication (contd.)

$C \leftarrow 0$

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



Parallel matrix multiplication (contd.)

Initialise $C \leftarrow 0$ in external memory

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 $C_{IK}^{J} \leftarrow A_{IJ} \cdot B_{JK}$
- updates $C_{IK} \stackrel{+}{\leftarrow} C_{IK}^{J}$ in external memory
- concurrent writing resolved by operator $+$ (recall concurrent block writing by array combining)

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 concurrent block writing by array combining)

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

Slackness required $n \geq p^{2/3}$ (as $\frac{n}{p^{1/3}} \geq p^{1/3}$ needed for concurrent write)

Theorem. Computing the matrix multiplication dag requires $comm = \Omega\left(\frac{n^2}{p^{2/3}}\right)$ per processor (no condition on $comp$!)

Theorem. Computing the matrix multiplication dag requires $comm = \Omega\left(\frac{n^2}{p^{2/3}}\right)$ per processor (no condition on $comp$!)

Proof: (discrete) volume vs total projection area

Related to (discrete) volume vs surface area, aka isoperimetry

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

Theorem. Computing the matrix multiplication dag requires $comm = \Omega(\frac{n^2}{p^{2/3}})$ per processor (no condition on *comp*!)

Proof: (discrete) volume vs total projection area

Related to (discrete) volume vs surface area, aka isoperimetry

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: $|A| \cdot |B| \cdot |C| \geq |V|^2$

Theorem. Computing the matrix multiplication dag requires $comm = \Omega\left(\frac{n^2}{p^{2/3}}\right)$ per processor (no condition on *comp*!)

Proof: (discrete) volume vs total projection area

Related to (discrete) volume vs surface area, aka isoperimetry

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 $comm = \Omega\left(\frac{n^2}{p^{2/3}}\right)$ □

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

Includes e.g. the following forms of matrix multiplication:

- numerical, with only operators $+$, $\cdot$ allowed (not operator $-$)
- Boolean, with only operators $\vee$, $\wedge$ allowed (not `if/then`)

2-matrix multiplication: standard circuit

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

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

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

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

2-matrix multiplication: Strassen's circuit

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

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: operators $+$, $-$, $\cdot$ allowed on elements

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

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

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

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

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

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

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

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

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

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

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

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

N-matrix multiplication: bilinear circuit

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

Bilinear circuits for matrix multiplication:

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

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

Block-recursive matrix multiplication

Given a scheme: bilinear circuit with fixed N, R

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

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

Apply the scheme, treating

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

Block-recursive matrix multiplication

Given a scheme: bilinear circuit with fixed N, R

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

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

Apply the scheme, treating

- each '$+$' as block '$+$', each '$-$' as block '$-$'
- each '$\cdot$' 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 $O(n^\omega)$

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]
?	?	?	?	

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 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 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)

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 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

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

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

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

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

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

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

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)

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:

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

Fast Boolean matrix multiplication

$A \wedge B = C$

Convert A, B into integer matrices by treating 0, 1 as integers (requires `if/then` on elements)

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

Convert C into a Boolean matrix by evaluating $C_{jk} \neq 0 \bmod n + 1$

Sequential work $O(n^\omega)$

BSP costs:

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

Parallel Boolean matrix multiplication by regular decomposition

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

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]

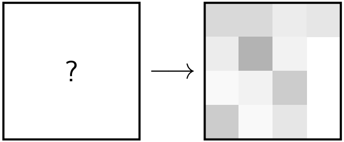
Parallel Boolean matrix multiplication by regular decomposition

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 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

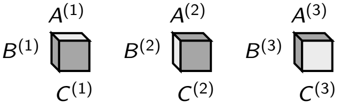
Equivalently, at least one of A, B, $\overline{C}$ has few 1s, i.e. is sparse

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

Parallel 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 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 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 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

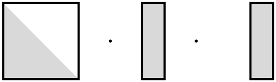
Every layer depends on $\leq \frac{n^2}{p}$ nonzeros of A, B, contributes $\leq \frac{n^2}{p}$ nonzeros to $\overline{C}$ due to sparsity

Communication saved by only sending the indices of nonzeros

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

Triangular system solution

$L \cdot b = c$



L: predistributed n-matrix

c: input n-vector

b: output n-vector

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

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

Forward substitution

$L \cdot b = c$

$L_{00} \cdot b_0 = c_0$

$L_{10} \cdot b_0 + L_{11} \cdot b_1 = c_1$

$L_{20} \cdot b_0 + L_{21} \cdot b_1 + L_{22} \cdot b_2 = c_2$

...

$\sum_{j:j \leq i} L_{ij} \cdot b_j = c_i$

...

$\sum_{j:j \leq n-1} L_{n-1,j} \cdot b_j = c_{n-1}$

Forward substitution

$L \cdot b = c$

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

Forward substitution

$L \cdot b = c$

$L_{00} \cdot b_0 = c_0$ $\qquad$ $b_0 \leftarrow L_{00}^{-1} \cdot c_0$

$L_{10} \cdot b_0 + L_{11} \cdot b_1 = c_1$ $\qquad$ $b_1 \leftarrow L_{11}^{-1} \cdot (c_1 - L_{10} \cdot b_0)$

$L_{20} \cdot b_0 + L_{21} \cdot b_1 + L_{22} \cdot b_2 = c_2$ $\qquad$ $b_2 \leftarrow L_{22}^{-1} \cdot (c_2 - L_{20} \cdot b_0 - L_{21} \cdot b_1)$

$\dots$ $\qquad$ $\dots$

$\sum_{j:j \le i} L_{ij} \cdot b_j = c_i$ $\qquad$ $b_i \leftarrow L_{ii}^{-1} \cdot (c_i - \sum_{j:j<i} L_{ij} \cdot b_j)$

$\dots$ $\qquad$ $\dots$

$\sum_{j:j \le n-1} L_{n-1,j} \cdot b_j = c_{n-1}$ $\qquad$ $b_{n-1} \leftarrow L_{n-1,n-1}^{-1} \cdot (c_{n-1} - \sum_{j:j<n-1} L_{n-1,j} \cdot b_j)$

Sequential work $O(n^2)$

Forward substitution

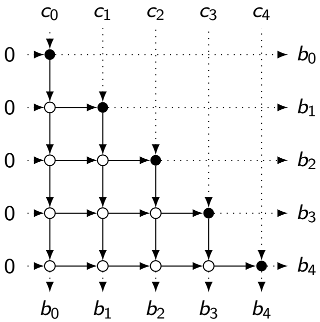
$L \cdot b = c$

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

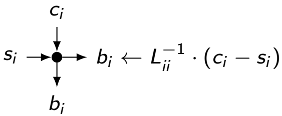
Sequential work $O(n^2)$

Symmetrically, an upper triangular system solved by back substitution

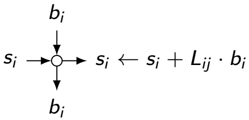
Parallel forward substitution by 2D grid



Pivot node:



Update node:



$comp = O(n^2/p)$ $comm = O(n)$ $sync = O(p)$

Block-recursive forward substitution

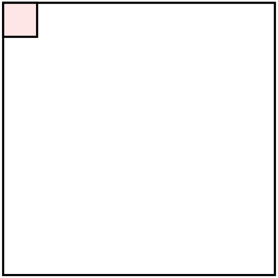
$L \cdot b = c$

$$\begin{bmatrix} L_{\underline{00}} & \\ L_{\underline{10}} & L_{\underline{11}} \end{bmatrix} \cdot \begin{bmatrix} b_{\underline{0}} \\ b_{\underline{1}} \end{bmatrix} = \begin{bmatrix} c_{\underline{0}} \\ c_{\underline{1}} \end{bmatrix}$$

Recursion: two half-sized subproblems

$L_{\underline{00}} \cdot b_{\underline{0}} = c_{\underline{0}}$ by recursion

$L_{\underline{11}} \cdot b_{\underline{1}} = c_{\underline{1}} - L_{\underline{10}} \cdot b_{\underline{1}}$ by recursion



Block-recursive forward substitution

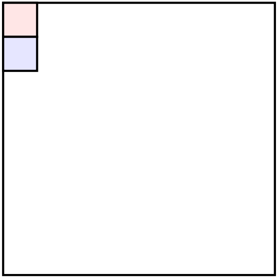
$L \cdot b = c$

$$\begin{bmatrix} L_{\underline{00}} & \\ L_{\underline{10}} & L_{\underline{11}} \end{bmatrix} \cdot \begin{bmatrix} b_{\underline{0}} \\ b_{\underline{1}} \end{bmatrix} = \begin{bmatrix} c_{\underline{0}} \\ c_{\underline{1}} \end{bmatrix}$$

Recursion: two half-sized subproblems

$L_{\underline{00}} \cdot b_{\underline{0}} = c_{\underline{0}}$ by recursion

$L_{\underline{11}} \cdot b_{\underline{1}} = c_{\underline{1}} - L_{\underline{10}} \cdot b_{\underline{1}}$ by recursion



Block-recursive forward substitution

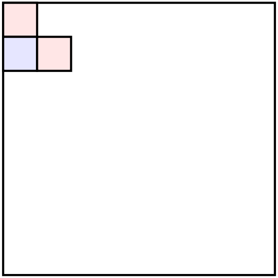
$L \cdot b = c$

$$\begin{bmatrix} L_{\underline{00}} & \\ L_{\underline{10}} & L_{\underline{11}} \end{bmatrix} \cdot \begin{bmatrix} b_{\underline{0}} \\ b_{\underline{1}} \end{bmatrix} = \begin{bmatrix} c_{\underline{0}} \\ c_{\underline{1}} \end{bmatrix}$$

Recursion: two half-sized subproblems

$L_{\underline{00}} \cdot b_{\underline{0}} = c_{\underline{0}}$ by recursion

$L_{\underline{11}} \cdot b_{\underline{1}} = c_{\underline{1}} - L_{\underline{10}} \cdot b_{\underline{1}}$ by recursion



Block-recursive forward substitution

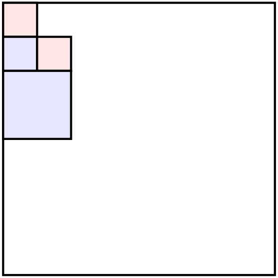
$L \cdot b = c$

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

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



Block-recursive forward substitution

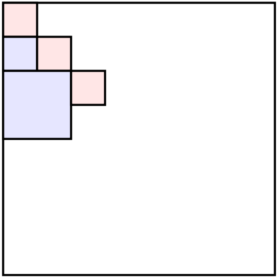
$L \cdot b = c$

$$\begin{bmatrix} L_{\underline{00}} & \\ L_{\underline{10}} & L_{\underline{11}} \end{bmatrix} \cdot \begin{bmatrix} b_{\underline{0}} \\ b_{\underline{1}} \end{bmatrix} = \begin{bmatrix} c_{\underline{0}} \\ c_{\underline{1}} \end{bmatrix}$$

Recursion: two half-sized subproblems

$L_{\underline{00}} \cdot b_{\underline{0}} = c_{\underline{0}}$ by recursion

$L_{\underline{11}} \cdot b_{\underline{1}} = c_{\underline{1}} - L_{\underline{10}} \cdot b_{\underline{1}}$ by recursion



Block-recursive forward substitution

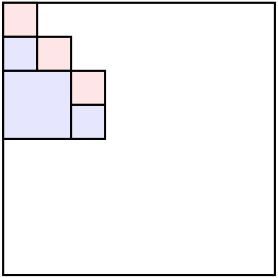
$L \cdot b = c$

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

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



Block-recursive forward substitution

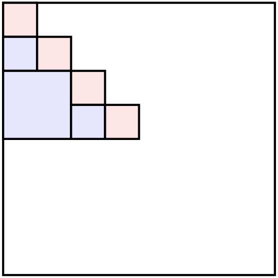
$L \cdot b = c$

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

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



Block-recursive forward substitution

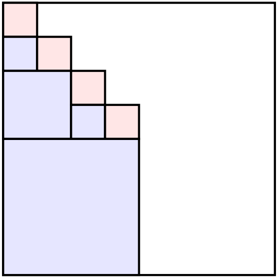
$L \cdot b = c$

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

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



Block-recursive forward substitution

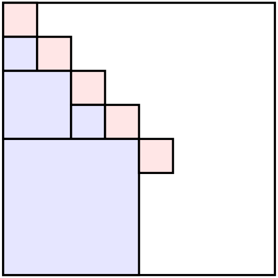
$L \cdot b = c$

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

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



Block-recursive forward substitution

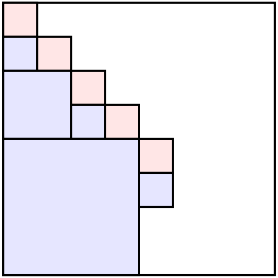
$L \cdot b = c$

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

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



Block-recursive forward substitution

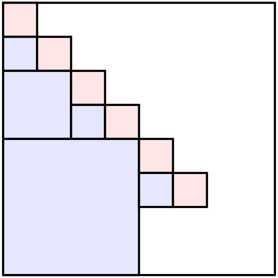
$L \cdot b = c$

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

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



Parallel matrix algorithms
Triangular system solution

Block-recursive forward substitution

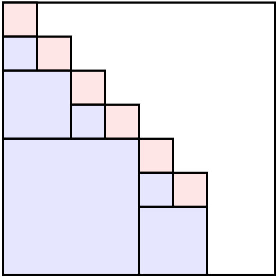
$L \cdot b = c$

$$\begin{bmatrix} L_{\underline{00}} & \\ L_{\underline{10}} & L_{\underline{11}} \end{bmatrix} \cdot \begin{bmatrix} b_{\underline{0}} \\ b_{\underline{1}} \end{bmatrix} = \begin{bmatrix} c_{\underline{0}} \\ c_{\underline{1}} \end{bmatrix}$$

Recursion: two half-sized subproblems

$L_{\underline{00}} \cdot b_{\underline{0}} = c_{\underline{0}}$ by recursion

$L_{\underline{11}} \cdot b_{\underline{1}} = c_{\underline{1}} - L_{\underline{10}} \cdot b_{\underline{1}}$ by recursion



Block-recursive forward substitution

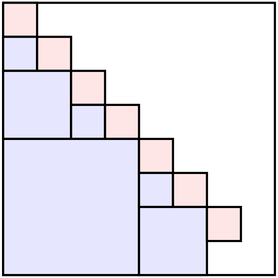
$L \cdot b = c$

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

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



Block-recursive forward substitution

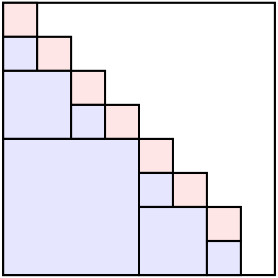
$L \cdot b = c$

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

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



Block-recursive forward substitution

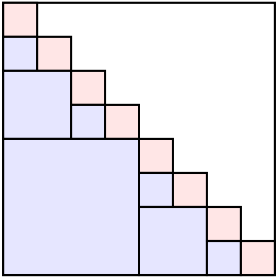
$L \cdot b = c$

$$\begin{bmatrix} L_{\underline{00}} & \\ L_{\underline{10}} & L_{\underline{11}} \end{bmatrix} \cdot \begin{bmatrix} b_{\underline{0}} \\ b_{\underline{1}} \end{bmatrix} = \begin{bmatrix} c_{\underline{0}} \\ c_{\underline{1}} \end{bmatrix}$$

Recursion: two half-sized subproblems

$L_{\underline{00}} \cdot b_{\underline{0}} = c_{\underline{0}}$ by recursion

$L_{\underline{11}} \cdot b_{\underline{1}} = c_{\underline{1}} - L_{\underline{10}} \cdot b_{\underline{1}}$ by recursion



Block-recursive forward substitution

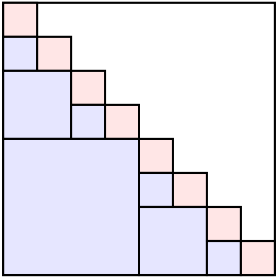
$L \cdot b = c$

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

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



Block-recursive forward substitution

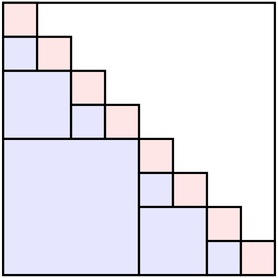
$L \cdot b = c$

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

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 block-recursive forward substitution

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

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 block-recursive forward substitution (contd.)

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

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

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

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

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

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

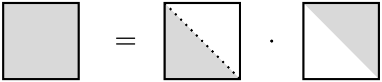
$$comp = O(n^2/p) \cdot \left(1 + 2 \cdot (\tfrac{1}{2})^2 + 2^2 \cdot (\tfrac{1}{2^2})^2 + \ldots\right) + O\left((n/p)^2\right) \cdot p = O(n^2/p) + O(n^2/p) = O(n^2/p)$$

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

$comp = O(n^2/p)$ $comm = O(n)$ $sync = O(p)$

Generic elimination (LU decomposition)

$A = L \cdot U$

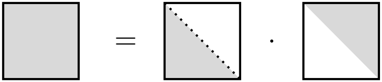


A: input n-matrix

L, U: output n-matrices

Generic elimination (LU decomposition)

$A \quad = \quad L \quad \cdot \quad U$



A: input n-matrix

L, U: output n-matrices

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

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

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

Block generic elimination

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

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

Block generic elimination

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

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

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

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

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

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

Compute $\bar{A}_{\underline{11}} = L_{\underline{11}} \cdot U_{\underline{11}}$ recursively, also $L_{\underline{11}}^{-1}$, $U_{\underline{11}}^{-1}$

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

Block generic elimination (contd.)

Assumption: $\det A_{00} \neq 0$, $\det \bar{A}_{11} \neq 0$, hence no pivoting required

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

Block generic elimination (contd.)

Block-iterative generic elimination with block size r

$$A = \begin{matrix} (r) \\ (n-r) \end{matrix} \overset{\begin{matrix} (r) & (n-r) \end{matrix}}{\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}} = LU \text{ on } A_{00}, \text{ then on } \bar{A}_{11}$$

Sequential work $O(n^3)$

Block-recursive generic elimination

$$A = \begin{matrix} (n/2) \\ (n/2) \end{matrix} \overset{\begin{matrix} (n/2) & (n/2) \end{matrix}}{\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}} = LU \text{ recursively on } A_{00}, \text{ then recursively on } \bar{A}_{11}$$

Sequential work $O(n^3)$ or $O(n^\omega)$ using fast matrix multiplication

Parallel block generic 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 block generic 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

Parallel block generic 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

Parallel block generic 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

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

Parallel block generic 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

Parallel block generic 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

Parallel block generic 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

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

Parallel block generic 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

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

$\alpha = 2/3$:

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

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

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

Elimination with pairwise pivoting [Gentleman, Kung: 1981]

$$T \cdot A = R \qquad \det T \neq 0$$

Elimination with pairwise pivoting [Gentleman, Kung: 1981]

$T \cdot A = R$ $\qquad \det T \neq 0$

$$\begin{bmatrix} 1 & \cdot \\ -a_2/a_1 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} a_1 \\ \cdot \end{bmatrix} \qquad \text{if } a_1 \neq 0$$

$$\begin{bmatrix} \cdot & 1 \\ 1 & \cdot \end{bmatrix} \begin{bmatrix} 0 \\ a_2 \end{bmatrix} = \begin{bmatrix} a_2 \\ \cdot \end{bmatrix} \qquad \text{if } a_1 = 0$$

Iterative GE with pairwise pivoting

Sequential work $O(n^3)$

Elimination by Givens rotations (QR decomposition)

$Q \cdot A = R$ $\qquad Q \cdot Q^T = I$

Parallel matrix algorithms

Gaussian elimination with pivoting

Elimination by Givens rotations (QR decomposition)

$$Q \cdot A = R \qquad\qquad Q \cdot Q^T = I$$

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ \cdot \end{bmatrix}$$

$$c = a_1/(a_1^2 + a_2^2)^{1/2} = \cos\phi$$
$$s = a_2/(a_1^2 + a_2^2)^{1/2} = \sin\phi$$
$$b_1 = (a_1^2 + a_2^2)^{1/2}$$

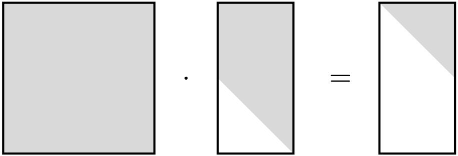
Iterative GE by Givens rotations

Sequential work $O(n^3)$

Block elimination with pairwise pivoting or by Givens rotations

Block-recursive elimination with PP [Schönhage: 1973]

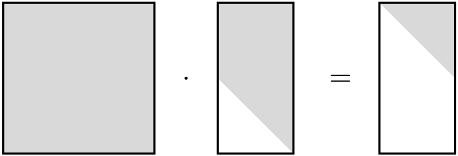
$T \cdot A = R$



Block elimination with pairwise pivoting or by Givens rotations

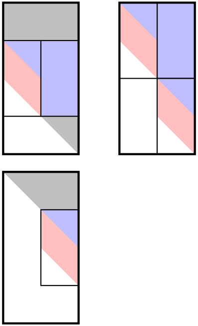
Block-recursive elimination with PP [Schönhage: 1973]

$T \cdot A = R$

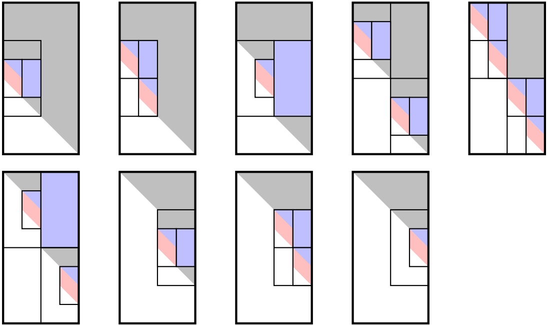


Block elimination with pairwise pivoting or by Givens rotations (contd.)

Recursion depth 1



Recursion depth 2



red: eliminate

blue: update

Skew-block elimination with pairwise pivoting or by Givens rotations

Threshold: $n_0 = n/p^\alpha \qquad 1/2 \leq \alpha \leq 2/3$

Threshold blocks: special distributed elimination

BSP cost similar to generic GE, but needs clever scheduling

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

Elimination with column pivoting, also Householder reflections

Block elimination $comp = O(n^3/p)$ $comm = O(n^2)$ $sync = O(p)$

Fine-grained elimination $comp = O(n^3/p)$ $comm = O(n^2/p)$ $sync = O(n)$

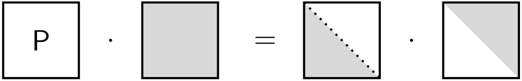
Can we do any better? (Probably not)

Parallel matrix algorithms

Gaussian elimination with pivoting

PLU decomposition problem

$$P \cdot A = L \cdot U$$



A: input n-matrix

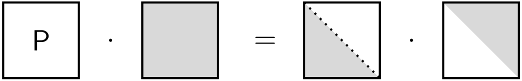
P, L, U: output n-matrices

Parallel matrix algorithms

Gaussian elimination with pivoting

PLU decomposition problem

$P \cdot A = L \cdot U$



A: input n-matrix

P, L, U: output n-matrices

P is a permutation matrix: 0–1 matrix with one nonzero per row/column

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

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

Block elimination with column pivoting

Generalise PLU decomposition to "tall" rectangular matrices

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

$$A = \begin{matrix} \scriptstyle(n) \\ \scriptstyle(m-n) \end{matrix} \overset{(n)}{\begin{bmatrix} A_{00} \\ A_{10} \end{bmatrix}} \qquad P \cdot \begin{bmatrix} A_{00} \\ A_{10} \end{bmatrix} = \begin{bmatrix} L_{00} \\ L_{10} \end{bmatrix} \cdot \begin{bmatrix} U_{00} \\ \cdot \end{bmatrix}$$

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

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

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

Block elimination with column pivoting (contd.)

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

Recursion base: $m \times 1$ matrix

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

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

Block elimination with column pivoting (contd.)

Block-iterative elimination with block size r

$$PA = P \cdot \begin{array}{c} (r) \\ (n-r) \end{array} \overset{\begin{array}{cc} (r) & (n-r) \end{array}}{\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}} = LU \text{ on } \begin{bmatrix} A_{00} \\ A_{10} \end{bmatrix}, \text{ then on updated } \begin{bmatrix} A_{01} \\ A_{11} \end{bmatrix}$$

Sequential work $O(n^3)$

Block-recursive elimination

$$PA = P \cdot \begin{array}{c} (n/2) \\ (n/2) \end{array} \overset{\begin{array}{cc} (n/2) & (n/2) \end{array}}{\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}} = LU \text{ recursively on } \begin{bmatrix} A_{00} \\ A_{10} \end{bmatrix}, \text{ then recursively}$$

on updated $\begin{bmatrix} A_{01} \\ A_{11} \end{bmatrix}$

Sequential work $O(n^3)$ or $O(n^\omega)$ using fast matrix multiplication

Parallel block elimination with column pivoting

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

At recursion level k:

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

In particular, at level $\log p$:

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

Parallel block 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

$comp = O(n^3/p)$ $comm = O(n^2)$ $sync = O(p)$

Parallel block elimination with column pivoting (contd.)

Alternative: no switching to sequential computation

Level $\log p$: threshold to switch to fine-grained parallel 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

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

Parallel block 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

$comp = O(n^3/p)$ | $comm = O(n^2)$ | $sync = O(p)$

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

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

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

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

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

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

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

$\odot$ distributes over $\odot$

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

In general, no subtraction or division!

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

Some specific semirings:

	S	$\oplus$	$\boxed{0}$	$\odot$	$\boxed{1}$
real	$\mathbb{R}$	$+$	0	$\cdot$	1
Boolean	$\{0, 1\}$	$\vee$	0	$\wedge$	1
tropical	$\mathbb{R}^+$	min	$+\infty$	$+$	0

$\mathbb{R}^+ = \mathbb{R}_{\geq 0} \cup \{+\infty\}$

Some specific semirings:

	S	$\oplus$	$\boxed{0}$	$\odot$	$\boxed{1}$
real	$\mathbb{R}$	$+$	0	$\cdot$	1
Boolean	$\{0,1\}$	$\vee$	0	$\wedge$	1
tropical	$\mathbb{R}^+$	min	$+\infty$	$+$	0

$\mathbb{R}^+ = \mathbb{R}_{\geq 0} \cup \{+\infty\}$

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 identity matrix

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

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

Examples

- real: $a^* = 1 + a + a^2 + a^3 + \cdots = \begin{cases} \frac{1}{1-a} & \text{if } |a| < 1 \\ \text{undefined} & \text{otherwise} \end{cases}$
- Boolean: $a^* = 1 \vee a \vee a \vee a \vee \ldots = 1$
- tropical: $a^* = \min(0, a, 2a, 3a, \ldots) = 0$

In matrix semirings, closures are more interesting

A semiring is closed, if

- infinite $a_1 \oplus a_2 \oplus a_3 \oplus \cdots$ (e.g. a closure) always defined
- infinite $\oplus$ commutative, associative
- $\odot$ distributive over infinite $\oplus$

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

A semiring is closed, if

- infinite $a_1 \oplus a_2 \oplus a_3 \oplus \cdots$ (e.g. a closure) always defined
- infinite $\oplus$ commutative, associative
- $\odot$ distributive over infinite $\oplus$

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

Examples

- real semiring not closed: infinite $+$ can be divergent

A semiring is closed, if

- infinite $a_1 \oplus a_2 \oplus a_3 \oplus \cdots$ (e.g. a closure) always defined
- infinite $\oplus$ commutative, associative
- $\odot$ distributive over infinite $\oplus$

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

Examples

- real semiring not closed: infinite $+$ can be divergent
- Boolean semiring closed: infinite $\vee$ is $\exists$

A semiring is closed, if

- infinite $a_1 \oplus a_2 \oplus a_3 \oplus \cdots$ (e.g. a closure) always defined
- infinite $\oplus$ commutative, associative
- $\odot$ distributive over infinite $\oplus$

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

Examples

- real semiring not closed: infinite $+$ can be divergent
- Boolean semiring closed: infinite $\vee$ is $\exists$
- tropical semiring closed: infinite min is inf (greatest lower bound)

A semiring is closed, if

- infinite $a_1 \oplus a_2 \oplus a_3 \oplus \cdots$ (e.g. a closure) always defined
- infinite $\oplus$ commutative, associative
- $\odot$ distributive over infinite $\oplus$

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

Examples

- real semiring not closed: infinite $+$ can be divergent
- Boolean semiring closed: infinite $\vee$ is $\exists$
- tropical semiring closed: infinite min is inf (greatest lower bound)

Matrix closure problem, aka algebraic path problem

Given A: $n \times n$ matrix over a semiring

Compute $A^* = I \oplus A \oplus A^2 \oplus A^3 \oplus \cdots$

Matrix closure problem, aka algebraic path problem

Given A: $n \times n$ matrix over a semiring

Compute $A^* = I \oplus A \oplus A^2 \oplus A^3 \oplus \cdots$

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

Matrix closure problem, aka algebraic path problem

Given A: $n \times n$ matrix over a semiring

Compute $A^* = I \oplus A \oplus A^2 \oplus A^3 \oplus \cdots$

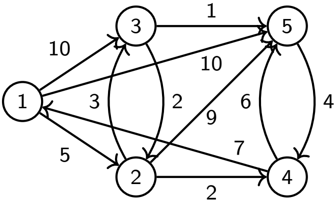
- real: $A^* = I + A + A^2 + \cdots = (I - A)^{-1}$, if nonsingular

Weighted digraph on n nodes: define matrix as

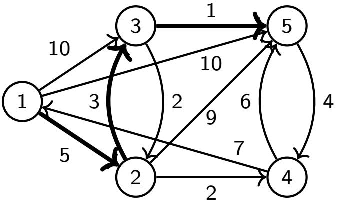
$$A_{ij} = \begin{cases} \boxed{1} = 0 & \text{if } i = j \\ \text{length of edge } i \to j & \text{if edge exists} \\ \boxed{0} = +\infty & \text{otherwise} \end{cases}$$

- Boolean: A^* gives transitive closure
- tropical: A^* gives all-pairs shortest paths

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



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



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

Parallel graph algorithms

Algebraic path problem

Floyd–Warshall algorithm [Floyd, Warshall: 1962]

A: $n \times n$ matrix over closed semiring

First step of elimination: pivot $A_{00} = \boxed{1}$

$A'_{\underline{11}} \leftarrow A_{\underline{11}} \oplus A_{\underline{1}0} \odot A_{0\underline{1}}$

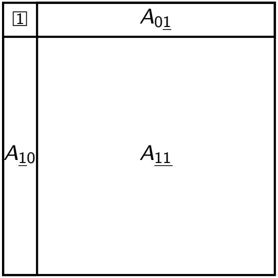
(E.g. replace A_{ij} with $A_{i0} + A_{0j}$, if it gives a shortcut)

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

Generic Gaussian elimination in disguise

Works for any closed semiring

Sequential work $O(n^3)$

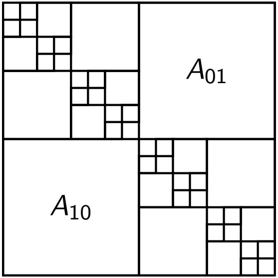


Block Floyd–Warshall algorithm

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

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

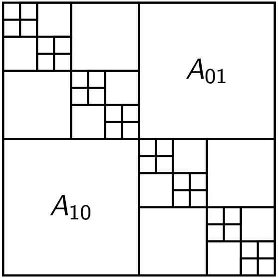
$A'_{\underline{01}} \leftarrow A'_{\underline{00}} A_{\underline{01}}$ $\quad A'_{\underline{10}} \leftarrow A_{\underline{10}} A'_{\underline{00}}$ $\quad A'_{\underline{11}} \leftarrow A_{\underline{11}} \oplus A_{\underline{10}} A'_{\underline{00}} A_{\underline{01}}$

$A''_{\underline{11}} \leftarrow (A'_{\underline{11}})^*$ 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 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 algebraic path computation

Similar to LU decomposition by block generic Gaussian elimination

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

Parallel algebraic path computation

Similar to LU decomposition by block generic Gaussian elimination

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

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

Parallel algebraic path computation (contd.)

In particular:

$\alpha = 1/2$

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

Cf. 2D grid

Parallel algebraic path computation (contd.)

In particular:

$\alpha = 1/2$

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

Cf. 2D grid

$\alpha = 2/3$

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

Cf. matrix multiplication

All-pairs shortest paths (APSP) problem: matrix closure (algebraic path) problem over 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 $+$

All-pairs shortest paths (APSP) problem: matrix closure (algebraic path) problem over 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 $+$

Can be solved by Floyd–Warshall algorithm (ordinary or block)

Also works with negative weights, but no negative cycles

To improve on Floyd–Warshall, we must exploit the tropical semiring's idempotence: $a \oplus a = \min(a, a) = a$

A: $n \times n$ matrix over the tropical semiring, defining a weighted digraph

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

Path size: its total number of edges

A: $n \times n$ matrix over the tropical semiring, defining a weighted digraph

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

Path size: its total number of edges

$(A^k)_{ij}$ = length of shortest path $i \rightsquigarrow j$ among those of size $\leq k$

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

A: $n \times n$ matrix over the tropical semiring, defining a weighted digraph

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

Path size: its total number of edges

$(A^k)_{ij}$ = length of shortest path $i \rightsquigarrow j$ among those of size $\leq k$

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

The APSP problem:

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

APSP by multi-Dijkstra

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:
$A_{0j} \leftarrow A_{0j} \oplus A_{0i} \odot A_{ij}$ for all j not yet ranked

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

APSP by multi-Dijkstra

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:
$A_{0j} \leftarrow A_{0j} \oplus A_{0i} \odot A_{ij}$ for all j not yet ranked

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 $O(n^2)$

APSP by multi-Dijkstra

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:
$A_{0j} \leftarrow A_{0j} \oplus A_{0i} \odot A_{ij}$ for all j not yet ranked

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 $O(n^2)$

APSP: run Dijkstra's algorithm independently from every node as a source, sequential work $O(n^3)$

Parallel APSP 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

Parallel APSP 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

$comp = O(n^3/p)$ $comm = O(n^2)$ $sync = O(1)$

Parallel APSP: summary so far

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

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

Parallel APSP: summary so far

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

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

Path doubling

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

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

...but not work-optimal: sequential time $O(n^3 \log n)$

Sparsified path doubling [Alon+: 1997]

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

Sparsified path doubling [Alon+: 1997]

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

Lex-tropical semiring (aka lexicographic semiring)

- elements are pairs (a, k) $\quad a \in \mathbb{R}^+ \quad k \in \mathbb{Z}^+$
- $\oplus$ is lexicographic min $\quad \boxed{0} = (+\infty, +\infty)$
- $\odot$ is numerical $+$ $\quad \boxed{1} = (0, 0)$

Weighted digraph on n nodes: define matrix as

$$A_{ij} = \begin{cases} \boxed{1} = (0,0) & \text{if } i = j \\ (\text{length of edge } i \to j, 1) & \text{if edge exists} \\ \boxed{0} = (+\infty, +\infty) & \text{otherwise} \end{cases}$$

Sparsified path doubling (contd.)

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

Let $(a,k)|_t = \begin{cases} (a,k) & \text{if } k = t \\ \boxed{0} & \text{otherwise} \end{cases}$

$A^k_{ij}|_\ell = \begin{cases} A^k_{ij} & \text{if realised by a path of size exactly } \ell \leq k \\ \boxed{0} & \text{otherwise} \end{cases}$

$A^k|_\ell$ contains all lengths of shortest paths of size exactly ℓ. May also contain some non-shortest path lengths (where the shortest path is of size $\geq k$), but that does no harm.

Sparsified path doubling (contd.)

We have $A^k = A^k|_0 \oplus \cdots \oplus A^k|_{\frac{k}{2}} \oplus \cdots \oplus A^k|_k$

Consider matrices in $\oplus$-sum $A^k|_{\frac{k}{2}} \oplus \cdots \oplus A^k|_k$

Total density of these $\frac{k}{2}$ matrices is ≤ 1. This is $\leq \frac{2}{k}$ per matrix on average, and hence also for some specific $A^k|_{\frac{k}{2}+\ell}$, $0 \leq \ell \leq \frac{k}{2}$

We have $(I \oplus A^k|_{\frac{k}{2}+\ell}) \odot A^k = A^{\frac{3k}{2}+\ell}$

This is because a shortest path of size $\leq \frac{3k}{2} + \ell$ is either

- of size $\leq k$, or
- $\big($shortest path of size exactly $\frac{k}{2} + \ell\big) \odot$ (one of size $\leq k$)

Sparse-by-dense matrix $\odot$-product: $\leq \frac{2n^2}{k} \cdot n = \frac{2n^3}{k}$ elementary $\odot$-products

Sparsified path doubling (contd.)

Compute matrices A, $A^{\frac{3}{2}+\ell}$, $A^{(\frac{3}{2})^2+\ell'}$, $\ldots$, $A^n = A^*$

Overall, $\leq \log_{3/2} n$ rounds of sparsified path doubling

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

Parallel APSP by sparsified path doubling

All processors collectively

- compute $B = A^{p+\ell}$ by $\leq \log_{3/2} p$ rounds of sparsified path doubling
- select $B|_p$ from B

$B|_p$ is dense, but can be decomposed into a $\odot$-product of sparse matrices

$B|_p = B|_q \odot B|_{p-q} \quad 0 \leq q \leq \frac{p}{2}$

Parallel APSP by sparsified path doubling

All processors collectively

- compute $B = A^{p+\ell}$ by $\leq \log_{3/2} p$ rounds of sparsified path doubling
- select $B|_p$ from B

$B|_p$ is dense, but can be decomposed into a $\odot$-product of sparse matrices

$B|_p = B|_q \odot B|_{p-q} \quad 0 \leq q \leq \frac{p}{2}$

Consider matrix pair $B|_q$, $B|_{p-q}$ for each q

Total density of these $\frac{p}{2}$ pairs is ≤ 1. This is $\leq \frac{2}{p}$ per pair on average, and hence also for some specific pair with a fixed q

Such a q is found sequentially by a designated processor

Parallel APSP by sparsified path doubling (contd.)

Every processor

- selects and writes its shares of $B|_q$, $B|_{p-q}$ from B
- reads whole $B|_q$, $B|_{p-q}$ and combines them to $B|_p = B|_q \odot B|_{p-q}$

All processors collectively

- compute $(B|_p)^*$ by parallel multi-Dijkstra
- compute $(B|_p)^* \odot B = A^*$ by parallel matrix $\odot$-multiplication

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

Parallel APSP by sparsified path doubling (contd.)

Every processor

- selects and writes its shares of $B|_q$, $B|_{p-q}$ from B
- reads whole $B|_q$, $B|_{p-q}$ and combines them to $B|_p = B|_q \odot B|_{p-q}$

All processors collectively

- compute $(B|_p)^*$ by parallel multi-Dijkstra
- compute $(B|_p)^* \odot B = A^*$ by parallel matrix $\odot$-multiplication

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

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

Parallel APSP by sparsified path doubling (contd.)

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

Parallel APSP by sparsified path doubling (contd.)

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

All processors collectively

- compute $B = A^{p^2+\ell}$ by $\leq 2\log_{3/2} p$ rounds of sparsified path doubling

Let $P = \{p, 2p, \ldots, p^2\}$, $P - q = \{p - q, 2p - q, \ldots, p^2 - q\}$ for any q

$B|_P = B|_p \oplus B|_{2p} \oplus \cdots \oplus B|_{p^2}$

All processors collectively

- select $B|_P$ from B

Parallel APSP by sparsified path doubling (contd.)

$B|_P$ is dense, but can be decomposed into a $\odot$-product of sparse matrices

$B|_P = B|_q \odot B|_{P-q} \quad 0 \leq q \leq \frac{p}{2}$

Consider matrix pair $B|_q$, $B|_{P-q}$ for each q

Total density of these $\frac{p}{2}$ pairs is ≤ 1. This is $\leq \frac{2}{p}$ per pair on average, and hence also for some specific pair with a fixed q

Such a q is found sequentially by a designated processor

Parallel APSP by sparsified path doubling (contd.)

Every processor

- selects and writes its shares of $B|_q$, $B|_{P-q}$ from B
- reads whole $B|_q$, $B|_{P-q}$ and combines them to $B|_P = B|_q \odot B|_{P-q}$
- computes $(B|_P)^*$ by $\leq \log_{3/2} n$ rounds of sparsified path doubling (with path sizes multiples of p)

All processors collectively

- compute $(B|_P)^* \odot B = A^*$ by parallel matrix $\odot$-multiplication

Parallel APSP by sparsified path doubling (contd.)

Every processor

- selects and writes its shares of $B|_q$, $B|_{P-q}$ from B
- reads whole $B|_q$, $B|_{P-q}$ and combines them to $B|_P = B|_q \odot B|_{P-q}$
- computes $(B|_P)^*$ by $\leq \log_{3/2} n$ rounds of sparsified path doubling (with path sizes multiples of p)

All processors collectively

- compute $(B|_P)^* \odot B = A^*$ by parallel matrix $\odot$-multiplication

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