Parallel Algorithms

Two closely related models of parallel computation.
Circuits

- Logic gates (AND/OR/not) connected by wires
- important measures
  - number of gates
  - depth (clock cycles in synchronous circuit)

PRAM

- P processors, each with a RAM, local registers
- global memory of M locations
- each processor can in one step do a RAM op or read/write to one global memory location
- synchronous parallel steps
- not realistic, but explores "degree of parallelism"

Essntially the same models, but let us focus on different things.

Circuits

- Logic gates (AND/OR/not) connected by wires
- important measures
  - number of gates
  - depth (clock cycles in synchronous circuit)
- bounded vs unbounded fan-in/out
- $AC(k)$ and $NC(k)$: unbounded and bounded fan-in with depth $O(\log^k n)$ for problems of size n
- $AC(k) \subset NC(k) \subset AC(k+1)$ using full binary tree
- $NC = \cup NC(k) = \cup AC(k)$

Addition

- consider adding a_i and b_i with carry c_{i-1} to produce output s_i and next carry c_i
- Ripple carry: $O(n)$ gates, $O(n)$ time

- Carry lookahead: $O(n)$ gates, $O(\log n)$ time
- preplan for late arrival of c_i.
- given a_i and b_i, three possible cases for c_i
  - if $a_i = b_i$, then $c_i = a_i$ determined without c_{i-1}: *generate* $c_1 = 1$ or *kill* $c_i = 0$
  - otherwise, *propogate* $c_i = c_{i-1}$
  - write $x_i = k, g, p$ accordingly
- consider 3×3 "multiplication table" for effect of two adders in a row. pair propagates previous carry only if both propagate.

		x_i: k	x_i: p	x_i: g
x_{i-1}	k	k	k	g
	p	k	p	g
	g	k	g	g

- Now let $y_0 = k$, $y_i = y_{i-1} \times x_i$
- constraints "multiplication table" by induction

		x_i: k	x_i: p	x_i: g
y_{i-1}	k	k	k	g
	p	k	never	g
	g	k	g	g

- conclude: $y_i = k$ means $c_i = 0$, $y_i = g$ means $c_i = 1$, and $y_i = p$ never happens
- so problem reduced to computing all y_i in parallel

Parallel prefix

- Build full binary tree
- two gates at each node
- pass up product of all children
- pass down product of all x_i preceding leftmost child
- works for any associative function

PRAM

various conflict resolutions (CREW, EREW, CRCW)

- $CRCW(k) \subset EREW(k+1)$
- $NC = \cup CRCW(k)$

PRAMs simulate circuits, and vice versa

- So NC well-defined

differences in practice

- EREW needs $\log n$ to find max (info theory lower bound)
- CRCW finds max in constant time with n^2 processors
  - Compare every pair
  - If an item loses, write "not max" in its entry
  - Check all entries
  - If item is max (not overwritten), write its value in answer
- in $O(\log\log n)$ time with n processors
  - Suppose k items remain
  - Make k^2/n blocks of n/k items
  - quadratic time max for each: $(k^2/n)(n/k)^2 = n$ processors total
  - recurrence: $T(k) = 1 + T(k^2/n)$
  - $T(n/2^i) = 1 + T(n/2^{2i})$
  - so $\log\log n$ iters.

parallel prefix

- using n processors

list ranking EREW

- next pointers $n(x)$
- $d(x) += d(n(x))$; $n(x) = n(n(x))$.
- by induction, sum of values on path to end doesn't change

0.1 Work-Efficient Algorithms

Idea:

- We've seen parallel algorithms that are somewhat "inefficient"
- do more total work (processors times time) than sequential
- Ideal solution: arrange for total work to be proportional to best sequential work
- *Work-Efficient Algorithm*
- Then a small number of processors (or even 1) can "simulate" many processors in a fully efficient way

- Parallel analogue of "cache oblivious algorithm"—you write algorithm once for many processors; lose nothing when it gets simulated on fewer.

Brent's theorem

- Different perspective on work: count number of processors actually working in each time step.
- If algorithm does x total work and critical path t
- Then p processors take $x/p + t$ time
- So, if use $p = x/t$ processors, finish in time t with efficient algorithm

Work-efficient parallel prefix

- linear sequential work
- going to need $\log n$ time
- so, aim to get by with $n/\log n$ processors
- give each processor a block of $\log n$ items to add up
- reduces problem to $n/\log n$ values
- use old algorithm
- each processor fixes up prefixes for its block

Work-efficient list ranking

- harder: can't easily give contiguous "blocks" of $\log n$ to each processor (requires list ranking)
- However, assume items in arbitrary order in array of $\log n$ structs, so *can* give $\log n$ distinct items to each processor.
- use random coin flips to knock out "alternate" items
- shortcut any item that is heads and has tails successor
- requires at most one shortcut
- and constant probability every other item is shortcut (and indepdendent)
- so by chernoff, 1/16 of items are shortcut out
- "compact" remmaining items into smaller array using parallel prefix on **array** of pointers (ignoring list structure) to collect only "marked" nodes and update their pointers
- let each processoor handle $\log n$ (arbitrary) items

- $O(n/\log n)$ processors, $O(\log n)$ time
- After $O(\log\log n)$ rounds, number of items reduced to $n/\log n$
- use old algorithm
- result: $O(\log n \log\log n)$ time, $n/\log n$ processors
- to improve, use faster "compaction" algorithm to collect marked nodes: $O(\log\log n)$ time randomized, or $O(\log n/\log\log n)$ deterministic. get optimal alg.
- How about deterministic algorithm? Use "deterministic coin tossing"
- take all local maxima as part of ruling set.

Euler tour to reduce to parallel prefix for computing depth in tree.

- work efficient

Expression Tree Evaluation: plus and times nodes
Generalize problem:

- Each tree edge has a label (a, b)
- meaning that if subtree below evaluates to y then value $(ay + b)$ should be passed up edge

Idea: pointer jumping

- prune a leaf
- now can pointer-jump parent
- problem: don't want to disconnect tree (need to feed all data to root!)
- solution: number leaves in-orde
- three step process:
  - shunt odd-numbered left-child leaves
  - shunt odd-number right-child leaves
  - divide leaf-numbers by 2

Consider a tree fragment

- method for eliminating all left-child leaves
- root q with left child p (product node) on edge labeled (a_3, b_3)
- p has left child edge (a_1, b_1) leaf ℓ with value v
- right child edge to s with label (a_2, b_2)

- fold out p and ℓ, make s a child of q
- what label of new edge?
- prepare for s subtree to eval to y.
- choose a, b such that $ay + b = a_3 \cdot [(a_1 v + b_1) \cdot (a_2 y + b_2)] + b_3$

0.2 Sorting

CREW Merge sort:

- merge to length-k sequences using n processors
- each element of first seq. uses binary search to find place in second
- so knows how many items smaller
- so knows rank in merged sequence: go there
- then do same for second list
- $O(\log k)$ time with n processors
- total time $O(\sum_{i \leq \lg n} \log 2^i) = O(\log^2 n)$

Faster merging:

- Merge n items in A with m in B in $O(\log \log n)$ time
- choose $\sqrt{n} \times \sqrt{m}$ evenly spaced fenceposts α_i, β_j among A and B respectively
- Do all $\sqrt{nm} \leq n + m$ comparisons
- use concurrent OR to find $\beta_j \leq \alpha_i \leq \beta_j + 1$ in constant time
- parallel compare every α_i to all $\sqrt{m}$ elements in (β_j, β_{j+1})
- Now α_i can be used to divide up both A and B into consistent pieces, each with $\sqrt{n}$ elements of A
- So recurse: $T(n) = 2 + T(\sqrt{n}) = O(\log \log n)$

Use in parallel merge sort: $O(\log n \log \log n)$ with n processors.

- Cole shows how to “pipeline” merges, get optimal $O(\log n)$ time.

Connectivity and connected components

Linear time sequential trivial.

Directed

Squaring adjacency matrix

- $\log n$ time to reduce diameter to 1
- mn processors for first iter, but adds edges
- so, n^3 processors
- improvements to mn processors
- But "transitive closure bottleneck" still bedevils parallel algs.

Undirected

Basic approach:

- Sets of connected vertices grouped as stars
- One root, all others parent-point to it
- Initially all vertices alone
- Edge "live" if connects two distinct stars
- Find live edges in constant time by checking roots
- For live edge with roots $u < v$, connect u as child of v
- May be conflicts, but CRCW resolves
- Now get stars again
  - Use pointer jumping
  - Note: may have chains of links, so need $\log n$ jumps
- Every live star attached to another
- So number of stars decreases by 2
- $m + n$ processors, $\log^2 n$ time.

Smarter: interleave hooking and jumping:

- Maintain set of rooted trees
- Each node points to parent
- Hook some trees together to make fewer trees
- Pointer jump (once) to make shallower trees

- Eventually, each connected component is one star

More details:

- "top" vertex: root or its children
- each vertex has label
- find root label of each top vertex
- Can detect if am star in constant time:
  - no pointer double reaches root
- for each edge:
  - If ends both on top, different components, then hook smaller component to larger
  - may be conflicting hooks; assume CRCW resolves
  - If star points to non-star, hook it
  - do one pointer jump

Potential function: height of live stars and tall trees

- Live stars get hooked to something (star or internal)
- But never hooked to leaf. So add 1 to height of target
- So sum of heights doesn't go up
- But now, every unit of height is in a tall tree
- Pointer doubling decreases by 1/3
- Total heigh divided each time
- So $\log n$ iterations

Summary: $O(m+n)$ processors, $O(\log n)$ time.
Improvements:

- $O((m+n)\alpha(m,n)/\log n)$ processors, $\log n$ time, CRCW
- Randomized $O(\log n)$, $O(m/\log n)$ processors, EREW

0.3 Randomization

Randomization in parallel:

- load balancing
- symmetry breaking
- isolating solutions

Classes:

- NC: poly processor, polylog steps
- RNC: with randomization. polylog runtime, monte carlo
- ZNC: las vegas NC
- immune to choice of R/W conflict resolution

Sorting

Quicksort in parallel:

- n processors
- each takes one item, compares to splitter
- count number of predecessors less than splitter
- determines location of item in split
- total time $O(\log n)$
- combine: $O(\log n)$ per layer with n processors
- problem: $\Omega(\log^2 n)$ time bound
- problem: $n \log^2 n$ work

Using many processors:

- do all n^2 comparisons
- use parallel prefix to count number of items less than each item
- $O(\log n)$ time
- or $O(n)$ time with n processors

Combine with quicksort:

- Note: single pivot step inefficient: uses n processors and $\log n$ time.

- Better: use $\sqrt{n}$ simultaneous pivots
- Choose $\sqrt{n}$ random items and sort fully to get $\sqrt{n}$ intervals
- For all n items, use binary search to find right interval
- recurse
- $T(n) = O(\log n) + T(\sqrt{n}) = O(\log n + \frac{1}{2}\log n + \frac{1}{4}\log n + \cdots) = O(\log n)$

Formal analysis:

- consider root-leaf path to any item x
- argue total number of parallel steps on path is $O(\log n)$
- consider item x
- claim splitter within $\alpha\sqrt{n}$ on each side
- since prob. not at most $(1 - \alpha\sqrt{n}/n)^{\sqrt{n}} \leq e^{-\alpha}$
- fix $\gamma, d < 1/\gamma$
- define $\tau_k = d^k$
- define $\rho_k = n^{(2/3)^k}$ $(\rho_{k+1} = \rho_k^{2/3})$
- note size ρ_k problem takes $\gamma^k \log n$ time
- note size ρ_k problem odds of having child of size $> \rho_{k+1}$ is less than $e^{-\rho_k^{1/6}}$
- argue at most d^k size-ρ_k problems whp
- follows because probability of d^k size-ρ_k problems in a row is at most
- deduce runtime $\sum d^k \gamma_k = \sum (d\gamma)^k \log n = O(\log n)$
- note: as problem shrinks, allowing more divergence in quantity for whp result
- minor detail: "whp" dies for small problems
- OK: if problem size $\log n$, finish in $\log n$ time with $\log n$ processors

Maximal independent set

trivial sequential algorithm

- inherently sequential
- from node point of view: each thinks can join MIS if others stay out
- randomization breaks this symmetry

Randomized idea

- each node joins with some probability
- all neighbors excluded
- many nodes join
- few phases needed

Algorithm:

- all degree 0 nodes join
- node v joins with probability $1/2d(v)$
- if edge (u, v) has both ends marked, unmark lower degree vertex
- put all marked nodes in IS
- delete all neighbors

Intuition: d-regular graph

- vertex vanishes if it or neighbor gets chosen
- mark with probability $1/2d$
- prob (no neighbor marked) is $(1 - 1/2d)^d$, constant
- so const prob. of neighbor of v marked—destroys v
- what about unmarking of v's neighbor?
- prob(unmarking forced) only constant as argued above.
- So just changes constants
- const fraction of nodes vanish: $O(\log n)$ phases
- Implementing a phase trivial in $O(\log n)$.

Prob chosen for IS, given marked, exceeds $1/2$

- suppose w marked. only unmarked if higher degree neighbor marked
- higher degree neighbor marked with prob. $\leq 1/2d(w)$
- only $d(w)$ neighbors
- prob. any superior neighbor marked at most 1/2.

For general case, define good vertices

- good: at least 1/3 neighbors have lower degree
- prob. no neighbor of good marked $\leq (1 - 1/2d(v))^{d(v)/3} \leq e^{-1/6}$.
- So some neighbor marked with prob. $1 - e^{-1/6}$
- Stays marked with prob. 1/2
- deduce prob. good vertex killed exceeds $(1 - e^{-1/6})/2$
- Problem: perhaps only one good vertex?

Good edges

- any edge with a good neighbor
- has const prob. to vanish
- show half edges good
- deduce $O(\log n)$ iterations.

Proof

- Let V_B be bad vertices; we count edges with both ends in V_B.
- direct edges from lower to higher degree d_i is indegree, d_o outdegree
- if v bad, then $d_i(v) \leq d(v)/3$
- deduce
$$\sum_{V_B} d_i(v) \leq \frac{1}{3} \sum_{V_B} d(v) = \frac{1}{3} \sum_{V_B} (d_i(v) + d_o(v))$$
- so $\sum_{V_B} d_i(v) \leq \frac{1}{2} \sum_{V_B} d_o(v)$
- which means indegree can only “catch” half of outdegree; other half must go to good vertices.
- more carefully,
  - $d_o(v) - d_i(v) \geq \frac{1}{3}(d(v)) = \frac{1}{3}(d_o(v) + d_i(v))$.

- Let V_G, V_B be good, bad vertices
- degree of bad vertices is

$$\begin{aligned}
2e(V_B,V_B)+e(V_B,V_G)+e(V_G,V_B) &= \sum_{v\in V_B} d_o(v)+d_i(v) \\
&\le 3\sum(d_o(v)-d_i(v)) \\
&= 3(e(V_B,V_G)-e(V_G,V_B)) \\
&\le 3(e(V_B,V_G)+e(V_G,V_B)
\end{aligned}$$

Deduce $e(V_B,V_B)\le e(V_B,V_G)+e(V_G,V_B)$. result follows.

Derandomization:

- Analysis focuses on edges,
- so unsurprisingly, pairwise independence sufficient
- not immediately obvious, but again consider d-uniform case
- prob vertex marked $1/2d$
- neighbors $1,\ldots,d$ in increasing degree order
- Let E_i be event that i is marked.
- Let E'_i be E_i but no E_j for $j<i$
- A_i event no neighbor of i chosen
- Then prob eliminate v at least

$$\begin{aligned}
\sum \Pr[E'_i\cap A_i] &= \sum \Pr[E'_i]\Pr[A_i \mid E'_i] \\
&\ge \sum \Pr[E'_i]\Pr[A_i]
\end{aligned}$$

- Wait: show $\Pr[A_i \mid E'_i]\ge \Pr[A_i]$
  - true if independent
  - measure $\Pr[\neg A_i \mid E'_i]\le \sum \Pr[E_w \mid E'_i]$ (sum over neighbors w of i)
  - measure

$$\begin{aligned}
\Pr[E_w \mid E'_i] &= \frac{\Pr[E_w\cap E']}{\Pr[E'_i]} \\
&= \frac{\Pr[(E_w\cap\neg E_1\cap\cdots)\cap E_i]}{\Pr[(\neg E_1\cap\cdots)\cap E_i]} \\
&= \frac{\Pr[E_w\cap\neg E_1\cap\cdots \mid E_i]}{\Pr[\neg E_1\cap\cdots \mid E_i]} \\
&\le \frac{\Pr[E_w \mid E_i]}{1-\sum_{j\le i}\Pr[E_j \mid E_i]} \\
&= \Theta(\Pr[E_w])
\end{aligned}$$

(last step assumes d-regular so only d neighbors with odds $1/2d$)

- But expected marked neighbors $1/2$, so by Markov $\Pr[A_i] > 1/2$
- so prob eliminate v exceeds $\sum \Pr[E'_i] = \Pr[\cup E_i]$
- lower bound as $\sum \Pr[E_i] - \sum \Pr[E_i \cap E_j] = 1/2 - d(d-1)/8d^2 > 1/4$
- so $1/2d$ prob. v marked but no neighbor marked, so v chosen
- Generate pairwise independent with $O(\log n)$ bits
- try all polynomial seeds in parallel
- one works
- gives deterministic NC algorithm

with care, $O(m)$ processors and $O(\log n)$ time (randomized)
LFMIS P-complete.

Perfect Matching

We focus on bipartite; book does general case.
Last time, saw detection algorithm in $\mathcal{RNC}$:

- Tutte matrix
- Sumbolic determinant nonzero iff PM
- assign random values in $1, \ldots, 2m$
- Matrix Mul, Determinant in $\mathcal{NC}$

How about finding one?

- If unique, no problem
- Since only one nozero term, ok to replace each entry by a 1.
- Remove each edge, see if still PM in parallel
- multiplies processors by m
- but still $\mathcal{NC}$

Idea:

- make unique minimum **weight** perfect matching
- find it

Isolating lemma: [MVV]

- Family of distinct sets over $x_1, \ldots, x_m$
- assign random weights in $1, \ldots, 2m$
- Pr(unique min-weight set)$\geq 1/2$
- Odd: no dependence on number of sets!
- (of course $< 2^m$)

Proof:

- Fix item x_i
- Y is min-value sets containing x_i
- N is min-value sets not containing x_i
- true min-sets are either those in Y or in N
- how decide? Value of x_i
- For $x_i = -\infty$, min-sets are Y
- For $x_i = +\infty$, min-sets are N
- As increase from $-\infty$ to ∞, single transition value when both X and Y are min-weight
- If only Y min-weight, then x_i in every min-set
- If only X min-weight, then x_i in no min-set
- If both min-weight, x_i is *ambiguous*
- Suppose no x_i ambiguous. Then min-weight set unique!
- Exactly one value for x_i makes it ambiguous given remainder
- So Pr(ambiguous)$= 1/2m$
- So Pr(any ambiguous)$< m/2m = 1/2$

Usage:

- Consider tutte matrix A
- Assign random value 2^{w_i} to x_i, with $w_i \in 1, \ldots, 2m$
- Weight of matching is $2^{\sum w_i}$
- Let W be minimum sum
- Unique w/pr 1/2

- If so, determinant is odd multiple of 2^W
- Try removing edges one at a time
- Edge in PM iff new determinant/2^W is even.
- Big numbers? No problem: values have poly number of bits

NC algorithm open.
For exact matching, P algorithm open.