CS 598: Provably Efficient Algorithms for Numerical and Combinatorial Problems

Part 3: Parallelism in Algorithms

Edgar Solomonik

University of Illinois at Urbana-Champaign

Circuits and PRAM

- Circuits were the first parallel algorithms
  - *depth – execution time*
  - *size – amount of work*
  - *width – number of processors needed*
- The *PRAM* model tries to stay consistent with this view
  - *instead of building dataflow into hardware, simply consider a shared uniform memory*
  - *different PRAM variants permit different concurrent memory access modes*
  - *PRAM types: EREW, CREW, CRCW*
    - *E-exclusive, C-concurrent*
    - *R-read, W-write*
  - *what happens on a concurrent write? more types, e.g. random or highest-priority succeeds, or arbitrary array reduction*

Inner Product in the PRAM Model

- Inner product with n processors
  - *In parallel, compute* $c_i = a_i b_i$ *(EREW-compliant)*
  - *CRCW with array reduction allows* $\sum_i c_i$ *to be done in a single parallel step*
  - *For EREW, require* $log_2(n)$ *parallel steps for reduction tree*
- Inner product with $n/\log_2(n)$ processors
  - *In* $\log_2(n)$ *parallel steps, compute* $c_i = a_i b_i$
  - *For* $k = 1, \cdots, \log_2(n) - 1$, *sum* $n/2^k$ *pairs of elements with* $\min(1, \log_2(n)/2^k)$ *parallel steps, yielding a parallel time of*

$$T(n) = 2\log_2(n) + \sum_{k=1}^{\log_2(\log_2(n))} 2^k = 4\log_2(n) = O(\log(n))$$

Basic Linear Algebra Subroutines (BLAS) in the PRAM Model

- Vector scaling (BLAS 1)
  - *For CREW, suffices to compute compute* $b_i = sa_i$ *in parallel*
  - *For EREW, need to broadcast* s*, requiring* $\log_2(n)$ *parallel steps with* n *processors*
- Matrix-vector multiplication and outer product (BLAS 2)
  - *Corresponds to* n *inner products (with the same vector appearing in all* n*)*
  - *For CRCW with array reduction, can compute in* $O(1)$ *time with* n^2 *processors*
  - *For EREW, can use* $O(n)$ *processors in time* $O(n)$ *by performing* $O(n)$ *independent products and accumulations concurrently* $O(n)$ *times*
  - *For EREW, can use* $O(n/\log n)$ *processors with time* $O(\log n)$ *by binary tree broadcast and reduction*

Work-Depth Model

- The work-depth (or work-time) model keeps track only of total work and algorithm depth/time
  - *Generally, we would like the amount of work W to be no greater than that done in the optimal sequential algorithm*
  - *Given depth D, we would like to use $O(W/D)$ processors to achieve time $O(D)$*
  - *More generally given p processors, would like to achieve time $O(W/p + D)$*
- Its possible to schedule a work-optimal PRAM algorithm so that it uses an asymptotically optimal number of processors
  - *Let PRAM algorithm have D steps with an infinite number of processors, with W_i being the amount of work done in the ith step*
  - *Subdivide the work at each step among the p processors, yielding cost*

$$T(p, D, W) = \sum_{i=1}^{D} \lceil W_i/p \rceil \leq \sum_{i=1}^{D} (\lfloor W_i/p \rfloor + 1) \leq D + W/p$$

  - *For all matrix-multiplication-like BLAS operations, obtain optimal work with respect to the classical (non-Strassen-like) approach, with depth $O(\log n)$*

Numerical Linear Algebra in PRAM

- Standard algorithms for triangular solve and matrix factorizations have polynomial depth
  - *In forward or backward substitution, must solve for* $x_1, \ldots, x_{i-1}$ *before* x_i
  - *Depth of such triangular solve algorithms is* $O(n)$, *work is* $O(n^2)$
  - *For Gaussian elimination (Cholesky/LU) and Householder/Givens/Gram-Schmidt QR depth is* $O(n)$ *and work is* $O(n^3)$
- Polylogarithmic depth algorithms exist for solving linear systems
  - *Triangular matrix inversion can be done recursively with polylogarithmic depth and* $O(n^3)$ *work*
  - *Schemes based on numerical optimization can be used for polylogarithmic depth matrix inversion, but these suffer from numerical instabilities and sensitivity to matrix conditioning*

Recursive Matrix Factorization Depth

- Recursive Cholesky $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T$ has polynomial depth

$$\begin{bmatrix} \boldsymbol{L}_{11} & \\ \boldsymbol{L}_{21} & \boldsymbol{L}_{22} \end{bmatrix} \begin{bmatrix} \boldsymbol{L}_{11}^T & \boldsymbol{L}_{21}^T \\ & \boldsymbol{L}_{22}^T \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} \end{bmatrix}$$

  where we have to solve $\boldsymbol{A}_{11} = \boldsymbol{L}_{11}\boldsymbol{L}_{11}^T$ *before factorizing the Schur complement* $\boldsymbol{A}_{22} - \boldsymbol{A}_{21}\boldsymbol{L}_{11}^{-1}\boldsymbol{L}_{11}^{-T}\boldsymbol{A}_{12} = \boldsymbol{L}_{22}\boldsymbol{L}_{22}^T$

- Recursive triangular inversion $\boldsymbol{S} = \boldsymbol{L}^{-1}$ has logarithmic depth

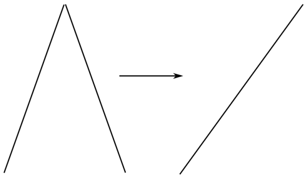
$$\begin{bmatrix} \boldsymbol{L}_{11} & \\ \boldsymbol{L}_{21} & \boldsymbol{L}_{22} \end{bmatrix} \begin{bmatrix} \boldsymbol{S}_{11} & \\ \boldsymbol{S}_{21} & \boldsymbol{S}_{22} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} & \\ & \boldsymbol{I} \end{bmatrix}$$

  where we $\boldsymbol{S}_{11} = \boldsymbol{L}_{11}^{-1}$ *and* $\boldsymbol{S}_{22} = \boldsymbol{L}_{22}^{-1}$ *can be done concurrently, while* $\boldsymbol{S}_{21} = \boldsymbol{S}_{22}\boldsymbol{L}_{21}\boldsymbol{S}_{11}$ *can be done with matrix multiplication which has* $D = O(\log(n))$

Sorting and Parallel Sorting

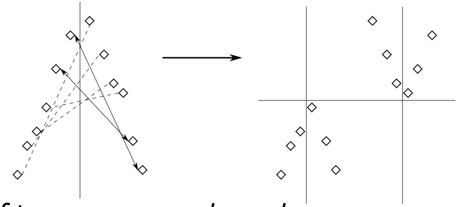
- Parallel sorting within a single shared-memory
  - *given n keys or n key-value pairs, order them in memory contiguously so that the ith smallest key (pair) is in the ith location*
  - *if there are equivalent keys, a stable sort is one that preserves their original ordering*
  - *depending on the type of key, we can work with their bit representation or only perform comparison operations*
- Most sorting algorithms can be classified as *merge-based* or *distribution-based*
  - *merge-based algorithms sort subsequences then merge them, e.g. mergesort, bitonic sort*
    - *sorting small subsequences is parallel and cache-efficient, but merging is challenging*
  - *distribution-based algorithms partition the keys into buckets, then sort the buckets, e.g. quicksort, radix sort*
    - *sorting buckets is parallel and cache-efficient, but partitioning is challenging*

Bitonic Sort



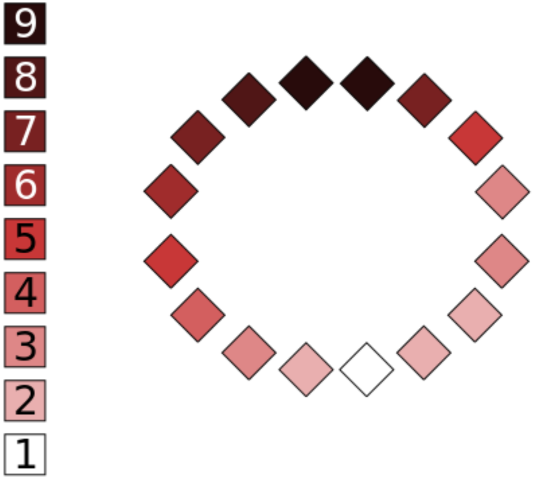
- *bitonic sort recurses like mergesort, and uses a "bitonic merge" to combine subsequences*
- *a bitonic merge is itself recursive and costs* $O(s \log_2 s)$ *to merge to subsequences of size* s
- *the bitonic merge is typically defined with the second subsequence in reverse order from the first*
- *given a sequence like* $(x_1 \leq \cdots \leq x_i \geq \cdots \geq x_{2s})$ *or a shift of such a sequence, it produces an increasing sequence of size* s

Bitonic Merge

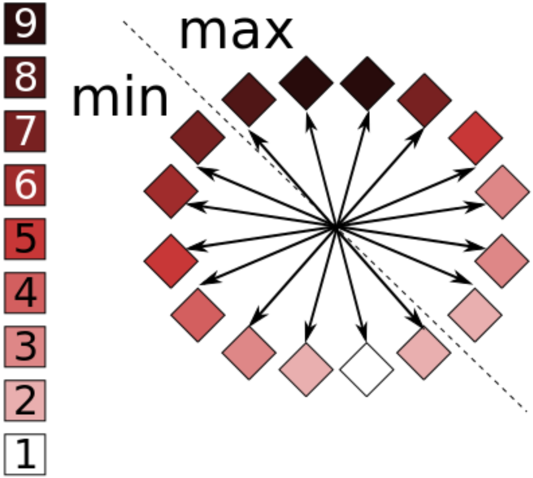


- *the bitonic merge of two reverse order subsequences of size s works as follows*
  - *compare and swap the ith element in the first subsequence with the ith element in the second*
  - *the swaps result in two bitonic sequences, the second has elements greater than any of those in the other*
  - *perform two bitonic merges recursively to merge these subsequences*
- *input may be increasing then decreasing or decreasing then increasing, and we may want an increasing or decreasing output*
- *'increasing' or 'decreasing' is a property of the buffer ordering, each step merges two 'sorted' subsequences*

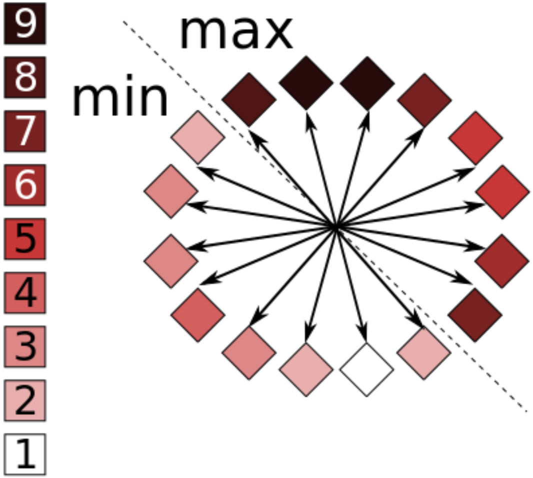
Bitonic sequence as a circle



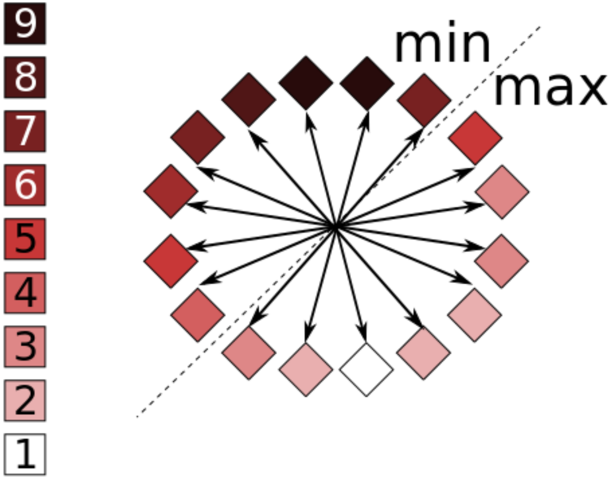
Matching opposite pairs in the circle



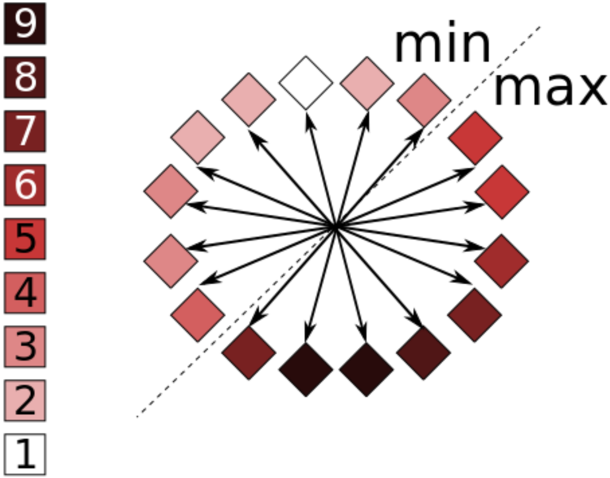
Swapping opposite pairs in the circle



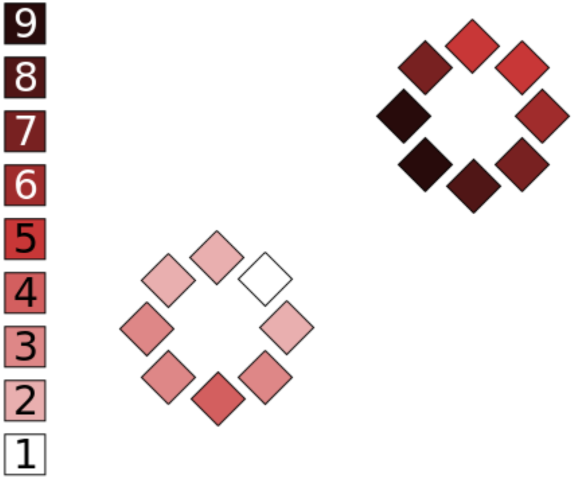
Collecting the min/max into different subsequences



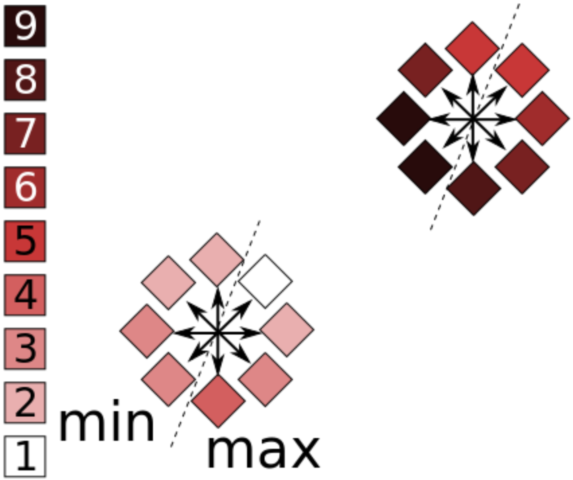
Any partition subdivides smaller/greater halves



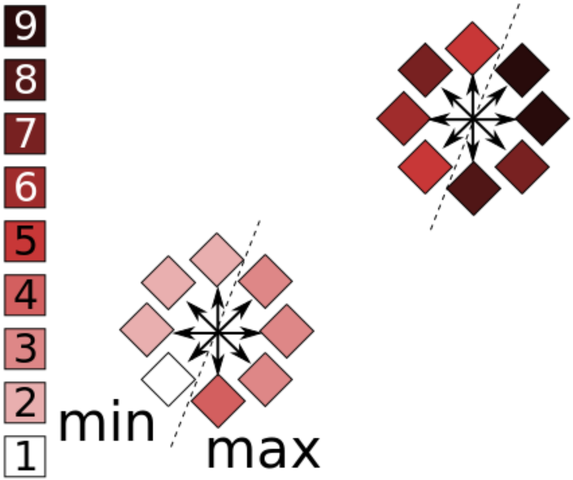
Arranging the two halves into new circles



Swapping opposites again



Continuing with bitonic merge recursively



Bitonic merge

- A *bitonic sequence* is any cyclic shift of the sequence $\{i_0 \leq \cdots \leq i_k \geq \cdots i_{n-1}\}$
  - *each step of bitonic merge partitions the sequence into smaller and greater sets of size $n/2$, both of which are bitonic sequences*
  - *each compare-and-swap acts on elements a distance of $n/2$ away*
  - *these pairings are unaffected by a cyclic shift*
  - *therefore, it suffices to consider swaps on the sequence $S = \{i_0 \leq \cdots \leq i_k \geq \cdots i_{n-1}\}$*
- There exists $l \leq k$, such that the largest $n/2$ elements of (unshifted bitonic sequence) S are the subsequence $\{i_l, \ldots, i_{l+n/2-1}\}$
  - *since every element is compared with one $n/2$ away, all of these will be paired with an element outside of the subsequence*
  - *hence the elements of this subsequence are the larger elements in the $n/2$ comparisons*
  - *any subset of a bitonic sequence is a bitonic sequence*

BFS with Sparse Linear Algebra

- For undirect graph $G = (V, E)$ Breadth First Search (BFS) takes as input a source vertex s and outputs an assignment of vertices to frontiers
  - *Initial frontier $F_0 = \{s\}$, unvisited vertices $U_0 = V \setminus \{s\}$*
  - *Compute F_{i+1} by taking all vertices in U_i adjacent to F_i, set $U_{i+1} = U_i \setminus F_{i+1}$*
  - *Should visit all vertices after at most D iterations, where D is the diameter of G*
- With adjacency matrix $\boldsymbol{A}$ of G, can compute BFS via matrix-vector products
  - *Enumerate vertices as $V \subseteq \{1, \dots, n\}$*
  - *Let $a_{uv} = 1$ if $(u, v) \in E$ and $a_{uv} = 0$ otherwise*
  - *Define starting frontier vector as $\boldsymbol{f}^{(0)}$ to be zero everwhere except $f_s^{(0)} = 1$*
  - *Define unvisited mask vector as $\boldsymbol{u}^{(0)}$ to be one everywhere except $u_s^{(0)} = 0$*
  - $\boldsymbol{f}^{(i+1)} = \boldsymbol{u}^{(i)} \odot (\boldsymbol{A}\boldsymbol{f}^{(i)})$, $\boldsymbol{u}^{(i+1)} = \boldsymbol{u}^{(i)} - \boldsymbol{f}^{(i+1)}$

Sparse Linear Algebra in PRAM

- Sparse-matrix-vector product (SpMV) with m nonzeros (edges) in matrix
  - *In a PRAM CRCW with array reduction, store $\boldsymbol{A}$ in coordinate format, SpMV requires $O(m)$ work and depth $O(1)$*
  - *In PRAM CREW, can use CSR (row-wise) format with each processor working on a row, yielding $O(n+m)$ work and depth $O(d)$ where d is the maximum degree*
- Sparse-matrix-sparse-vector product (SpMSpV) with k nonzeros (frontier vertices) in vector
  - *For sequential algorithm, optimal work w is given by number of nonzeros in the k matrix columns operated on by the sparse vector*
  - *In a PRAM CRCW with array reduction, store $\boldsymbol{A}$ in CSC (column-wise) format, SpMSpV requires $O(w)$ work and depth $O(d)$*
  - *In PRAM CREW, can extract the k columns and sort $O(w)$ entries to convert to CSR format with $n \times k$ matrix, then perform SpMV, yielding work $O(n + w \log w)$ and depth $O(d + \log n)$*

BFS on a PRAM

- Each BFS iteration requires an SpMSpV with an output filter $\boldsymbol{f}^{(i+1)} = \boldsymbol{u}^{(i)} \odot (\boldsymbol{A}\boldsymbol{f}^{(i)})$
  - *Can perform SpMV using CSR format and considering only rows for which $\boldsymbol{u}^{(i)}$ is nonzero, obtaining optimal work modulo input vector sparsity and depth $O(d)$*
  - *Leveraging both sparsity of filter and sparsity of input vector is hard*
  - *Choice of algorithms: push (SpMSpV) or pull (SpMV with output fulter), can be made based on size of frontier relative to size of set of unexplored vertices*
- Different choices of BFS algorithm yield different work/depth
  - *Sequential algorithm has work $O(m)$ since each vertex is visited once, so each edge is traversed once*
  - *Using only SpMV, can minimize depth on CRCW with array reduction to $O(D)$ for diameter D, but require require work $O(mD)$*
  - *Using SpMSpV on CRCW model with CSC layout, can minimize sparse-matrix preoduct work as $O(m)$, but depth is $O(dD)$*
  - *Application of filters with a dense representation requires $O(nD)$ work and depth $O(D)$*

BFS on a PRAM

- Each BFS iteration requires an SpMSpV with an output filter $\boldsymbol{f}^{(i+1)} = \boldsymbol{u}^{(i)} \odot (\boldsymbol{A}\boldsymbol{f}^{(i)})$
  - *Can perform SpMV using CSR format and considering only rows for which* $\boldsymbol{u}^{(i)}$ *is nonzero, obtaining optimal work modulo input vector sparsity and depth* $O(d)$
  - *Leveraging both sparsity of filter and sparsity of input vector is hard*
  - *Choice of algorithms: push (SpMSpV) or pull (SpMV with output fulter), can be made based on size of frontier relative to size of set of unexplored vertices*
- Different choices of BFS algorithm yield different work/depth
  - *Sequential algorithm has work* $O(m)$ *since each vertex is visited once, so each edge is traversed once*
  - *Using only SpMV, can minimize depth on CRCW with array reduction to* $O(D)$ *for diameter* D*, but require require work* $O(mD)$
  - *Using SpMSpV on CRCW model with CSC layout, can minimize sparse-matrix preoduct work as* $O(m)$*, but depth is* $O(dD)$
  - *Application of filters with a dense representation requires* $O(nD)$ *work and depth* $O(D)$

Connectivity in Graphs

- Connectivity seeks to label vertices with a unique label for each connected component
  - *Sequentially, can compute by a sequence of BFS operations with $O(m)$ work*
  - *On a PRAM, BFS may be expensive given graph with large diameter*
- Shiloach and Vishkin (1980) CRCW PRAM algorithm for connectivity
  - *Given a graph with n vertices and m edges, it achieves $O((n + m) \log n)$ work and $O(\log(n))$ depth*
  - *Computes forest where each tree is height one (is a star) and has the vertices of one connected component in G*
  - *Based on hooking (merging two trees by making one a subtree of the other) and shortcutting (pointer chasing on the parent pointer in a tree, to reduce height)*

Shiloach-Vishkin Connectivity Algorithm

Let each node i store ‘parent’ $p(i)$ and perform below steps until convergence

- conditional star hooking

  if $(i, j) \in E$, i *in star, and* $F(i) > F(j)$, *perform* $F(F(i)) \leftarrow F(j)$ *(for every star, some hook may succeed)*

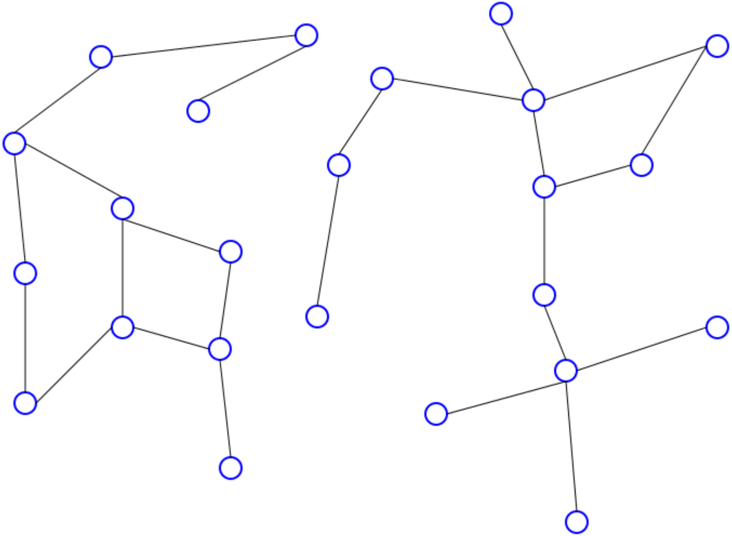
- unconditional star hooking

  if $(i, j) \in E$, i *in star, and* $F(i) \neq F(j)$, *perform* $F(F(i)) \leftarrow F(j)$ *(for every star, some hook succeeds)*

- Shortcutting (pointer chasing)

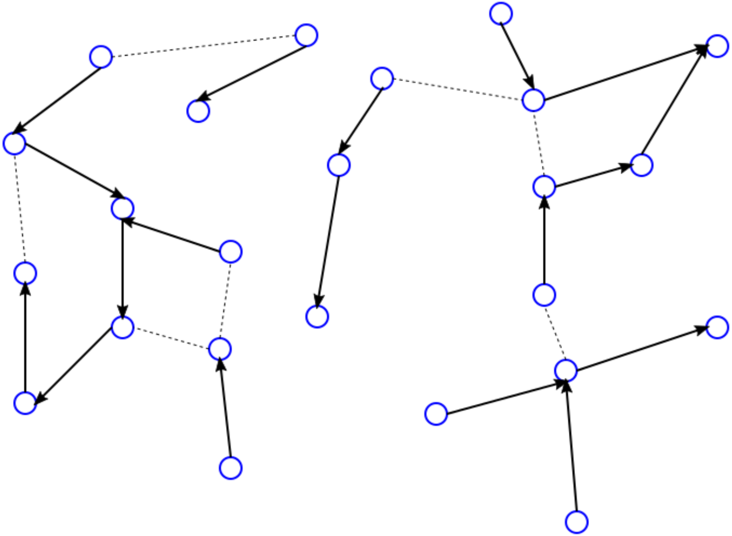
  if i *not in star,* $F(i) \leftarrow F(F(i))$

A graph with two connected components



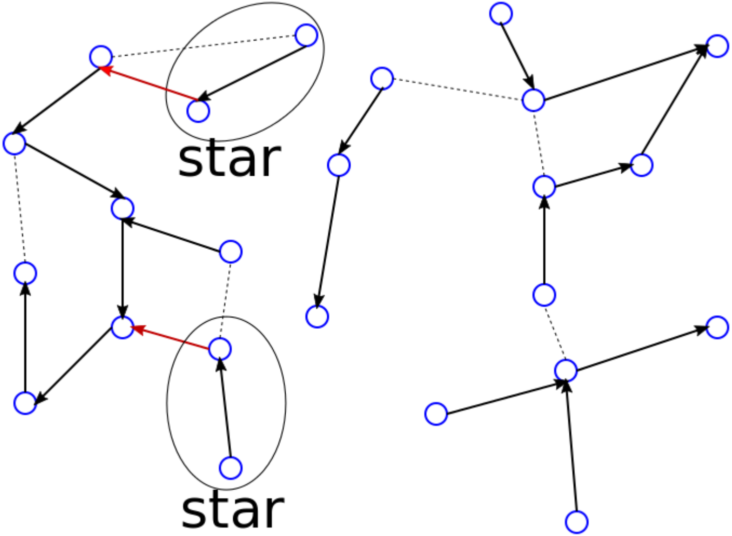
First iteration

1. conditional star hooking



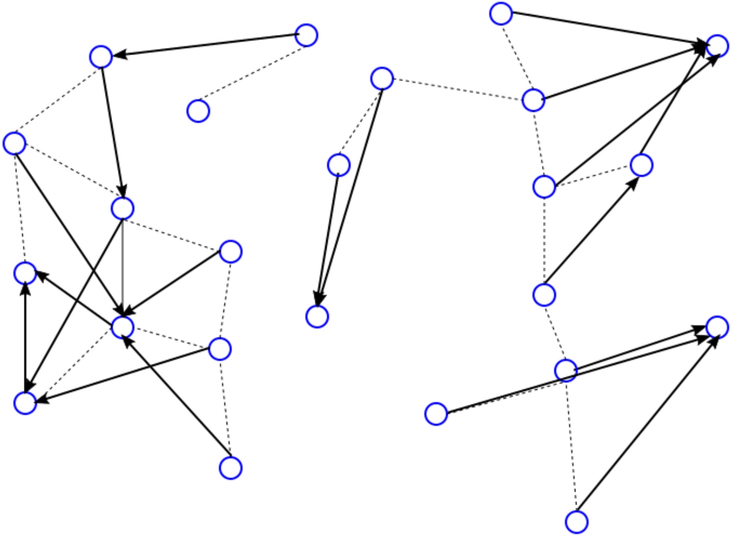
First iteration

2. unconditional star hooking



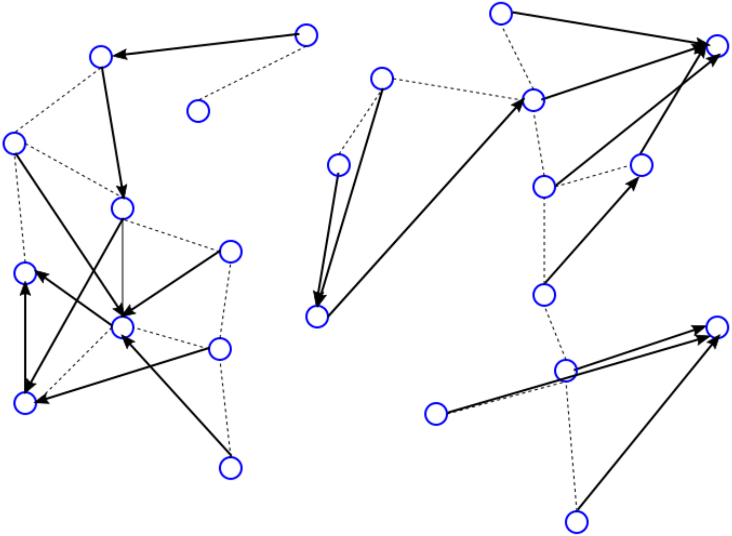
First iteration

3. shortcutting



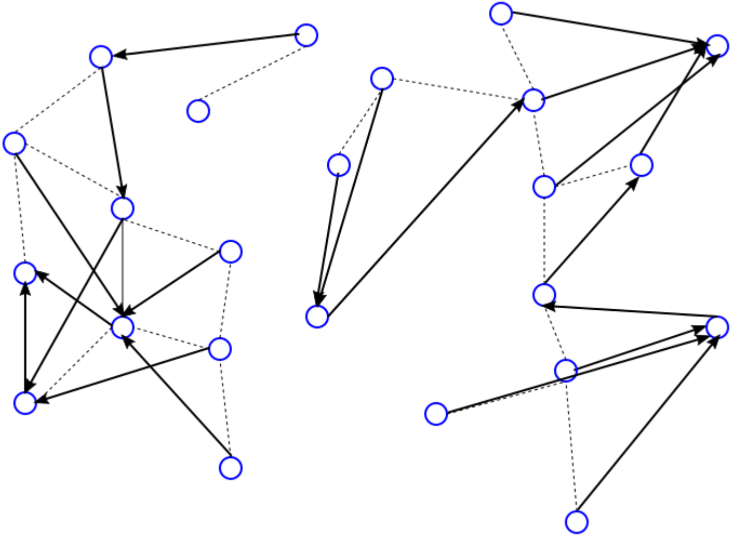
Second iteration

1. conditional star hooking



Second iteration

2. unconditional star hooking



Analysis of parallel tree connectivity

Algorithm converges after $O(\log(n))$ iterations

- Sum of tree heights (starts at n) decreases by a factor of at least $3/2$ every iteration
  - *steps 1 and 2 will hook every star to a tree*
  - *step 3 will decrease the height of every tree by* $3/2$
- Requires $O(n + m)$ work per iteration
  - *hooking steps can be doen via SpMV, with* $O(m)$ *work and* $O(1)$ *depth in CRCW*
  - *pointer chasing can be done in concurrently with* $O(n)$ *work*

Shortest Paths

- Given a positive weight function $w : E \to \mathbb{R}^+$, compute shortest distances from a source vertex s to all other vertices
  - *For unweighted graph, suffices to run BFS*
  - *Classical sequential approach (Dijkstra's algorithm) achieves work $O(m)$*
  - *Bellman-Ford algorithm requires $O(mD)$ work and handles negative weights*
  - *Bellman-Ford can be phrased via SpMV operations*
- Bellman-Ford can be expressed as matix-vector products on the tropical (min–plus) semiring, using SpMV/SpMSpV
  - *Let $a_{uv} = w(e)$ if $e = (u, v) \in E$ and $a_{uv} = \infty$ otherwise*
  - *Semiring vector addition is $\boldsymbol{w} = \boldsymbol{u} \oplus \boldsymbol{v} \Rightarrow w_i = \min(u_i, v_i)$*
  - *Semiring matrix-vector product is defined to be* $\boldsymbol{w} = \boldsymbol{A} \otimes \boldsymbol{v} \Rightarrow w_i = \min_j(a_{ij} + v_j)$
  - *Define starting frontier vector as $\boldsymbol{f}^{(0)}$ to be ∞ everwhere except $f_s^{(0)} = 0$*
  - *Iteration of Bellman-Ford computes new frontier as $\boldsymbol{f}^{(i+1)} = \boldsymbol{f}^{(i)} \oplus (\boldsymbol{A} \otimes \boldsymbol{f}^{(i)})$*

All-Pairs Shortest-Paths

- Given a positive weight function $w : E \to \mathbb{R}^+$, compute shortest distances matrix $\boldsymbol{D}$ containing minimum distances between all pairs of vertices
  - *Using the tropical semiring, $\boldsymbol{D}$ is the closure of $\boldsymbol{A}$, where $\boldsymbol{I}$ is 0 on the diagonal and ∞ everywhere else $\boldsymbol{D} = \boldsymbol{I} \oplus \boldsymbol{A} \oplus \boldsymbol{A}^2 \oplus \cdots \oplus \boldsymbol{A}^{n-1}$*
  - *Setting $\hat{\boldsymbol{A}} = \boldsymbol{I} \oplus \boldsymbol{A}$, we have that $\boldsymbol{D} = \hat{\boldsymbol{A}}^{\otimes n}$, since*

$$d_{ij} = \min_{k_1,\ldots,k_{n-2}} \hat{a}_{ik_1} + \hat{a}_{k_1k_2} + \cdots + \hat{a}_{k_{n-2}j}$$

  - *For a regular ring, using additive inverse, we typically compute the closure of a matrix using $(\boldsymbol{I} - \boldsymbol{A})\boldsymbol{D} = \boldsymbol{I}$, so $\boldsymbol{D} = (\boldsymbol{I} - \boldsymbol{A})^{-1}$*
- Floyd-Warshall algorithm computes achieves $O(n^3)$ work

$$\boldsymbol{D}^{(1)} = \boldsymbol{A}, \quad \boldsymbol{D}^{(i+1)} = \boldsymbol{D}^{(i)} \oplus \left(\boldsymbol{d}_i^{(i)} \otimes \boldsymbol{d}_i^{(i)^T}\right), \quad \boldsymbol{D} = \boldsymbol{D}^{(n)}$$

$O(n)$ *depth due to sequence of* n *rank-*1 *updates.*

Floyd Warshall Algorithm

- $\boldsymbol{D}^{(i)}$ contains the distances of all shortest paths $S^{(i)}$ with at most i edges going through some subset of vertices $\{1, \ldots, i-1\}$
  - *True for $\boldsymbol{D}^{(1)}$ (base case)*
  - *If true for $\boldsymbol{D}^{(i)}$, we have that $\boldsymbol{D}^{(i+1)} = \boldsymbol{D}^{(i)} \oplus \left(\boldsymbol{d}_i^{(i)} \otimes \boldsymbol{d}_i^{(i)T}\right)$*
    - *Paths in $S^{(i+1)} \setminus S^{(i)}$ must go to vertex i via a path in S_i and to another vertex via a path in S_i from vertex i*
    - *$\left(\boldsymbol{d}_i^{(i)} \otimes \boldsymbol{d}_i^{(i)T}\right)$ contains the distances of all paths that could be in $S^{(i+1)} \setminus S^{(i)}$*
  - *Inductive assumption also implies $\boldsymbol{D}^{(n)} = \boldsymbol{D}$*
- A recursive alternative to Floyd-Warshall is given by Gauss-Jordan elimination (Kleene's APSP algorithm)
  - *Let $\hat{\boldsymbol{A}} = \begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} \end{bmatrix}$, do $\begin{bmatrix} \boldsymbol{B}_{11} & \boldsymbol{B}_{12} \\ \boldsymbol{B}_{21} & \boldsymbol{B}_{22} \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}_{11}^* & \boldsymbol{B}_{11} \otimes \boldsymbol{A}_{12} \\ \boldsymbol{A}_{21} \otimes \boldsymbol{B}_{11} & \boldsymbol{A}_{22} \oplus (\boldsymbol{B}_{21} \otimes \boldsymbol{B}_{12}) \end{bmatrix}$ and then compute $\begin{bmatrix} \boldsymbol{A}_{11}^* & \boldsymbol{A}_{12}^* \\ \boldsymbol{A}_{21}^* & \boldsymbol{A}_{22}^* \end{bmatrix} = \begin{bmatrix} \boldsymbol{B}_{11} \oplus (\boldsymbol{A}_{12}^* \otimes \boldsymbol{A}_{21}^*) & \boldsymbol{B}_{12} \otimes \boldsymbol{A}_{22}^* \\ \boldsymbol{A}_{22}^* \otimes \boldsymbol{B}_{21} & \boldsymbol{B}_{22}^* \end{bmatrix}$*

Parallel All-Pairs Shortest-Paths

- Path doubling can be used to obtain polylogarithmic depth
  - *Set* $\boldsymbol{D}^{(1)} = \boldsymbol{I} \oplus \boldsymbol{A}$ *and compute* $\boldsymbol{D}^{(2i)} = \boldsymbol{D}^{(i)} \otimes \boldsymbol{D}^{(i)}$
  - $\boldsymbol{D}^{(i)}$ *contains shortest paths with at most* i *edges, correctness follows by induction*
  - *Naive path doubling has* $O(n^3 \log n)$ *work and* $O(\log n)$ *or* $O(\log^2 n)$ *depth*
- Tiskin (2001) proposed an improvement to achieve $O(n^3)$ cost
  - *Consider the disjoint sum* $\boldsymbol{D}^{(i)} = \boldsymbol{R}^{(i)} \oplus \boldsymbol{S}^{(i)}$ *where* $\boldsymbol{R}^{(i)}$ *contains distances in* $\boldsymbol{D}^{(i)}$ *that go through exactly* i *edges (it becomes sparser as* i *grows)*
  - *Compute* $\boldsymbol{D}^{(2i)} = \boldsymbol{D}^{(i)} \oplus (\boldsymbol{R}^{(i)} \otimes \boldsymbol{D}^{(i)})$*, since each shortest path with at least* $i+1$ *and at most* $2i$ *edges must contain a shortest path with exactly* i *edges*

Parallel (Approximate) Matrix Inversion

- Gauss-Jordan can be used to invert matrix, recursive Cholesky is similar
  - *Both require two recursive calls and $O(1)$ matrix multiplications*
  - *Depth is linear in matrix dimension $D(n) = 2D(n) + O(\log n)$ and $D(1) = O(1)$ so $D(n) = O(n)$*
  - *Work is $W(n) = O(n^3)$ (Strassen-like matrix multiplication gives subcubic cost)*
- Can theoretically invert with polylogarithmic depth via Newton's method
  - *Solve nonlinear equations $\boldsymbol{f}(\boldsymbol{X}) = \mathrm{vec}(\boldsymbol{X}^{-1} - \boldsymbol{A})$ via Newton's method*

$$\boldsymbol{X}^{(i+1)} = \boldsymbol{X}^{(i)} - \boldsymbol{J_f}(\boldsymbol{X}^{(i)})^{-1}\boldsymbol{f}(\boldsymbol{X}^{(i)})$$

  - *Since $\boldsymbol{J_f}(\boldsymbol{X}) = -\boldsymbol{X}^{-1} \otimes \boldsymbol{X}^{-1}$, Newton's iteration can be computed via*

$$\begin{aligned}\boldsymbol{X}^{(i+1)} &= \boldsymbol{X}^{(i)} + \boldsymbol{X}^{(i)} \otimes \boldsymbol{X}^{(i)}\,\mathrm{vec}({\boldsymbol{X}^{(i)}}^{-1} - \boldsymbol{A}) \\ &= \boldsymbol{X}^{(i)} + \boldsymbol{X}^{(i)}({\boldsymbol{X}^{(i)}}^{-1} - \boldsymbol{A})\boldsymbol{X}^{(i)} = (2\boldsymbol{I} - \boldsymbol{X}^{(i)}\boldsymbol{A})\boldsymbol{X}^{(i)}\end{aligned}$$

  - *Quadratic convergence gives good approximation after $O(\log n)$ steps with cost $O(n^3 \log n)$, can use within Gauss-Jordan to get poly-log depth and $O(n^3)$ work*