Parallel Graph Algorithms

Aydın Buluç

ABuluc@lbl.gov

http://gauss.cs.ucsb.edu/~aydin/

Lawrence Berkeley National Laboratory

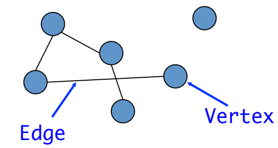
CS267, Spring 2015

March 19, 2015

Slide acknowledgments: A. Azad, S. Beamer, J. Gilbert, K. Madduri

Graph Preliminaries

Define: Graph G = (V,E)
-a set of vertices and a set of edges between vertices



n=|V| (number of vertices)

m=|E| (number of edges)

D=diameter (max #hops between any pair of vertices)

- Edges can be directed or undirected, weighted or not.
- They can even have attributes (i.e. semantic graphs)
- Sequences of edges $<u_1,u_2>, <u_2,u_3>, \ldots ,<u_{n-1},u_n>$ is a **path** from u_1 to u_n. Its **length** is the sum of its weights.

Lecture Outline

- Applications
- Designing parallel graph algorithms
- Case studies:
  - A. **Graph traversals:** Breadth-first search
  - B. **Shortest Paths:** Delta-stepping, Floyd-Warshall
  - C. **Maximal Independent Sets:** Luby's algorithm
  - D. **Strongly Connected Components**
  - E. **Maximum Cardinality Matching**

Lecture Outline

- Applications
- Designing parallel graph algorithms
- Case studies:
  - A. **Graph traversals:** Breadth-first search
  - B. **Shortest Paths:** Delta-stepping, Floyd-Warshall
  - C. **Maximal Independent Sets:** Luby's algorithm
  - D. **Strongly Connected Components**
  - E. **Maximum Cardinality Matching**

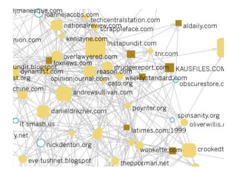
Routing in transportation networks

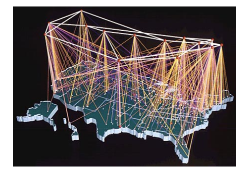
Road networks, Point-to-point shortest paths: 15 seconds (naïve) → **10 microseconds**

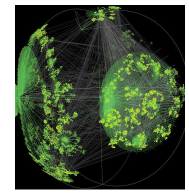
H. Bast et al., "Fast Routing in Road Networks with Transit Nodes", Science 27, 2007.

Internet and the WWW

- The world-wide web can be represented as a directed graph
  - Web search and crawl: traversal
  - Link analysis, ranking: Page rank and HITS
  - Document classification and clustering
- Internet topologies (router networks) are naturally modeled as graphs







Scientific Computing

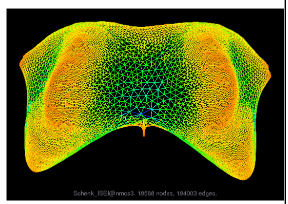
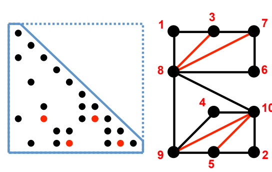


Image source: Yifan Hu, "A gallery of large graphs"

- Reorderings for sparse solvers
  - Fill reducing orderings
    - Partitioning, eigenvectors
  - Heavy diagonal to reduce pivoting (matching)
- Data structures for efficient exploitation of sparsity
- Derivative computations for optimization
  - graph colorings, spanning trees
- Preconditioning
  - Incomplete Factorizations
  - Partitioning for domain decomposition
  - Graph techniques in algebraic multigrid
    - Independent sets, matchings, etc.
  - Support Theory
    - Spanning trees & graph embedding techniques



$G^+(A)$
[chordal]

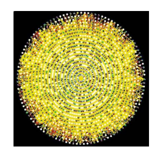
B. Hendrickson, "Graphs and HPC: Lessons for Future Architectures", http://www.er.doe.gov/ascr/ascac/Meetings/Oct08/Hendrickson%20ASCAC.pdf

Large-scale data analysis

- Graph abstractions are very useful to analyze complex data sets.
- Sources of data: petascale simulations, experimental devices, the Internet, sensor networks
- Challenges: data size, heterogeneity, uncertainty, data quality

Astrophysics: massive datasets, temporal variations

Bioinformatics: data quality, heterogeneity



Social Informatics: new analytics challenges, data uncertainty

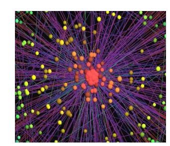
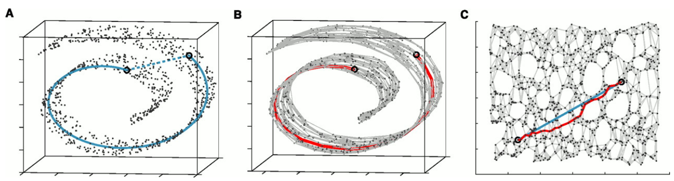


Image sources: (1) http://physics.nmt.edu/images/astro/hst_starfield.jpg (2,3) www.visualComplexity.com

Manifold Learning

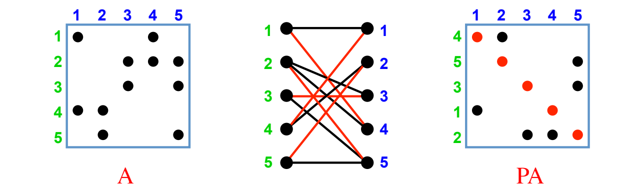
Isomap (Nonlinear dimensionality reduction): Preserves the intrinsic geometry of the data by using the geodesic distances on manifold between all pairs of points

Tools used or desired:
- K-nearest neighbors
- *All pairs shortest paths (APSP)*
- Top-k eigenvalues

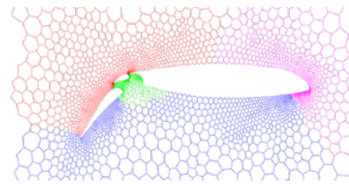


Tenenbaum, Joshua B., Vin De Silva, and John C. Langford. "A global geometric framework for nonlinear dimensionality reduction." Science 290.5500 (2000): 2319-2323.

Large Graphs used in Scientific Discovery



Matching in bipartite graphs: Permuting to heavy diagonal or block triangular form

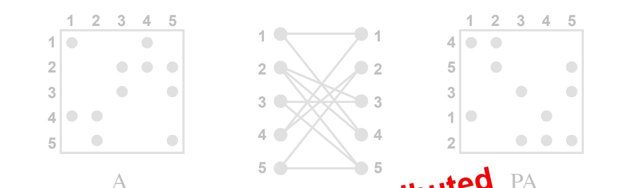


Graph partitioning: *Dynamic load balancing* in parallel simulations

Picture (left) credit: Sanders and Schulz

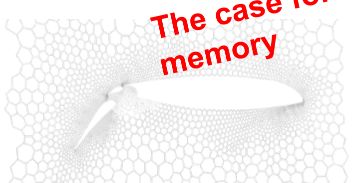
Problem size: as big as the sparse linear system to be solved or the simulation to be performed

Large Graphs used in Scientific Discovery



Matching in bipartite graphs: Permuting to heavy diagonal or block triangular form

The case for distributed memory



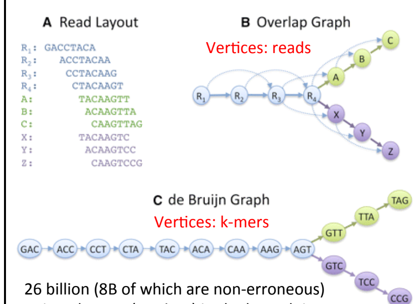
Graph partitioning: *Dynamic load balancing* in parallel simulations

Picture (left) credit: Sanders and Schulz

Problem size: as big as the sparse linear system to be solved or the simulation to be performed

Large Graphs in Scientific Discoveries

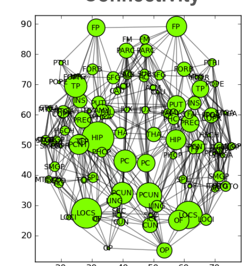
Whole genome assembly



26 billion (8B of which are non-erroneous) unique k-mers (vertices) in the hexaploit wheat genome W7984 for k=51

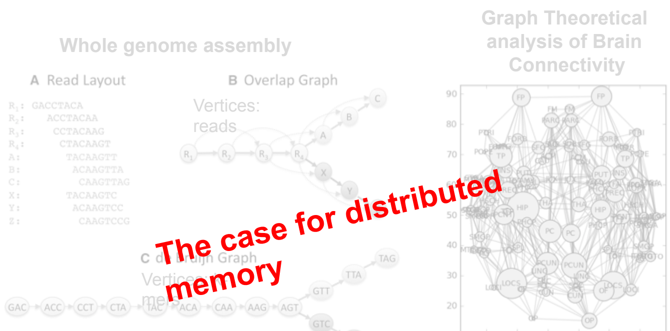
Schatz et al. (2010) Perspective: Assembly of Large Genomes w/2nd-Gen Seq. Genome Res. (figure reference)

Graph Theoretical analysis of Brain Connectivity



Potentially millions of neurons and billions of edges with developing technologies

Large Graphs in Scientific Discoveries



26 billion (8B of which are non-erroneous) unique k-mers (vertices) in the hexaploit wheat genome W7984 for k=51

Potentially millions of neurons and billions of edges with developing technologies

Schatz et al. (2010) Perspective: Assembly of Large Genomes w/2nd-Gen Seq. Genome Res. (figure reference)

Lecture Outline

- Applications
- Designing parallel graph algorithms
- Case studies:
  - A. **Graph traversals:** Breadth-first search
  - B. **Shortest Paths:** Delta-stepping, Floyd-Warshall
  - C. **Maximal Independent Sets:** Luby's algorithm
  - D. **Strongly Connected Components**
  - E. **Maximum Cardinality Matching**

The PRAM model

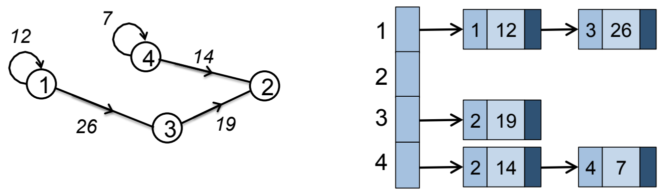
- Many PRAM graph algorithms in 1980s.
- Idealized parallel shared memory system model
- Unbounded number of synchronous processors; no synchronization, communication cost; no parallel overhead
- EREW (Exclusive Read Exclusive Write), CREW (Concurrent Read Exclusive Write)
- Measuring performance: space and time complexity; total number of operations (work)

PRAM Pros and Cons

- Pros
  - Simple and clean semantics.
  - The majority of theoretical parallel algorithms are designed using the PRAM model.
  - Independent of the communication network topology.
- Cons
  - Not realistic, too powerful communication model.
  - Communication costs are ignored.
  - Synchronized processors.
  - No local memory.
  - Big-O notation is often misleading.

Graph representations

Compressed sparse rows (CSR) = cache-efficient adjacency lists



Index into adjacency array	1	3	3	4	6	(row pointers in CSR)

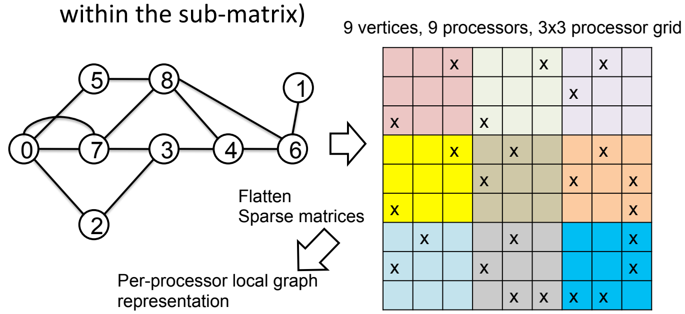
Adjacencies	1	3	2	2	4	(column ids in CSR)
Weights	12	26	19	14	7	(numerical values in CSR)

Distributed graph representations

- Each processor stores the entire graph ("full replication")
- Each processor stores n/p vertices and all adjacencies out of these vertices ("1D partitioning")
- How to create these "p" vertex partitions?
  - Graph partitioning algorithms: recursively optimize for conductance (edge cut/size of smaller partition)
  - Randomly shuffling the vertex identifiers ensures that edge count/processor are roughly the same

2D checkerboard distribution

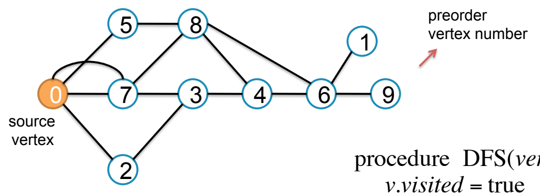
- Consider a logical 2D processor grid (p_r * p_c = p) and the matrix representation of the graph
- Assign each processor a sub-matrix (i.e, the edges within the sub-matrix)



Lecture Outline

- Applications
- Designing parallel graph algorithms
- Case studies:
  - **A. Graph traversals:** Breadth-first search
  - **B. Shortest Paths:** Delta-stepping, Floyd-Warshall
  - **C. Maximal Independent Sets:** Luby's algorithm
  - **D. Strongly Connected Components**
  - **E. Maximum Cardinality Matching**

Graph traversal: Depth-first search (DFS)

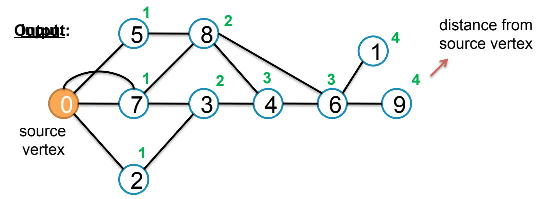


procedure DFS($vertex\ v$)
$\quad v.visited$ = true
$\quad$previsit(v)
$\quad$for all v s.t. $(v,w) \in E$
$\quad\quad$if($!w.visited$) DFS(w)
$\quad$postvisit(v)

Parallelizing DFS is a bad idea: *span(DFS) = O(n)*

J.H. Reif, **Depth-first search is inherently sequential**. Inform. Process. Lett. 20 (1985) 229-234.

Graph traversal : Breadth-first search (BFS)



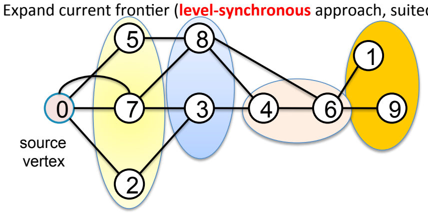
Memory requirements (# of machine words):

- Sparse graph representation: m+n
- Stack of visited vertices: n
- Distance array: n

Breadth-first search is a very important ***building block*** for other parallel graph algorithms such as (bipartite) matching, maximum flow, (strongly) connected components, betweenness centrality, etc.

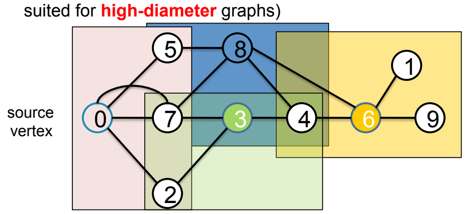
Parallel BFS Strategies

1. Expand current frontier (**level-synchronous** approach, suited for **low diameter** graphs)



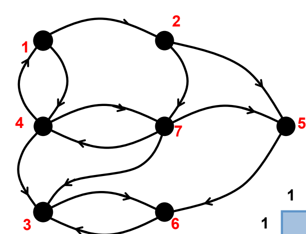
- O(D) parallel steps
- Adjacencies of all vertices in current frontier are visited in parallel

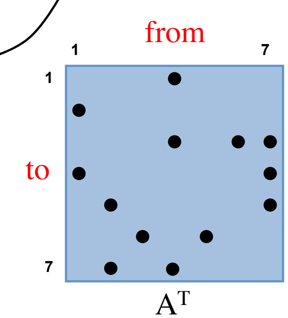
2. Stitch multiple concurrent traversals (Ullman-Yannakakis approach, suited for **high-diameter** graphs)

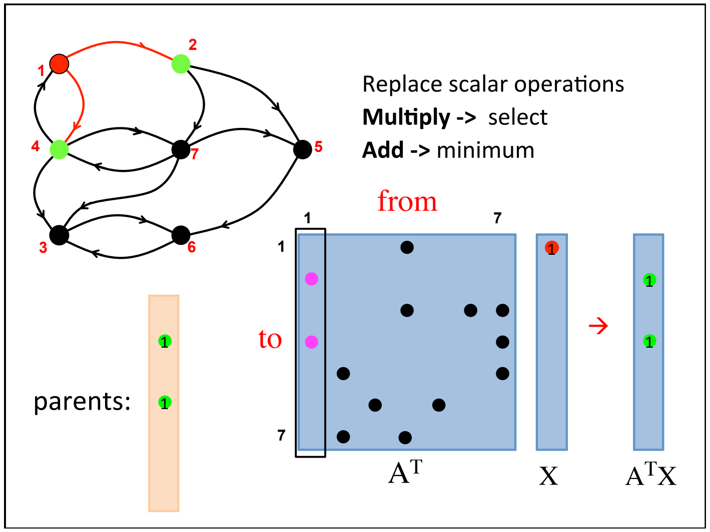


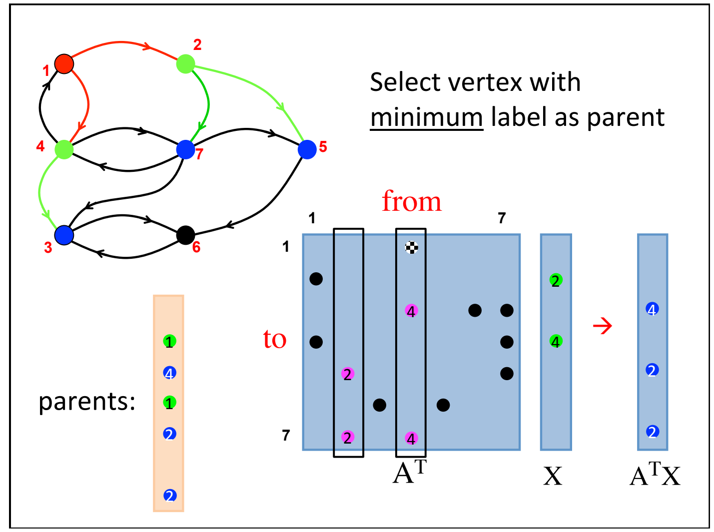
- path-limited searches from "super vertices"
- APSP between "super vertices"

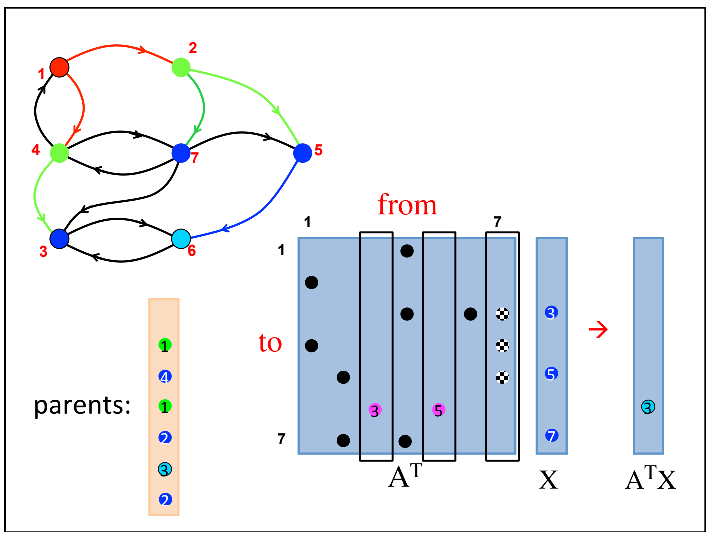
Breadth-first search using matrix algebra

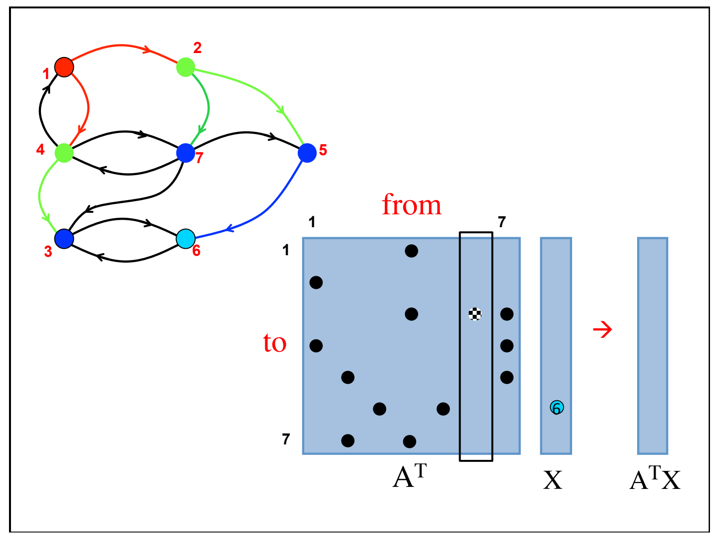




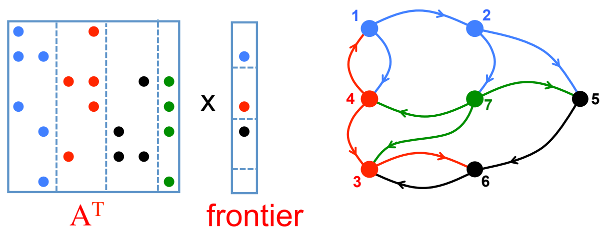

Replace scalar operations
Multiply -> select
Add -> minimum
from
to
parents:
A^T
X
A^TX


Select vertex with
minimum label as parent
from
to
parents:
A^T
X
A^TX


from
to
parents:
A^T
X
A^TX


from
to
A^T
X
A^TX

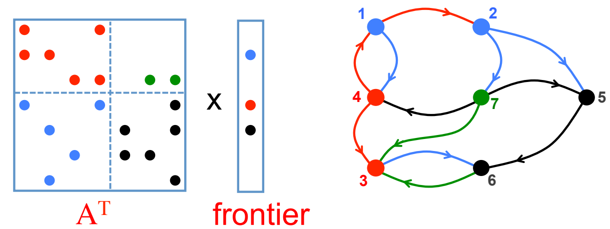
1D Parallel BFS algorithm



ALGORITHM:

1. Find owners of the current frontier's adjacency [computation]
2. Exchange adjacencies via all-to-all. **[communication]**
3. Update distances/parents for unvisited vertices. [computation]

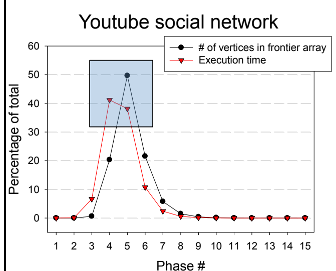
2D Parallel BFS algorithm

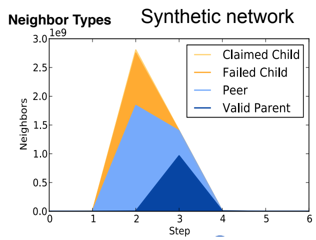


ALGORITHM:

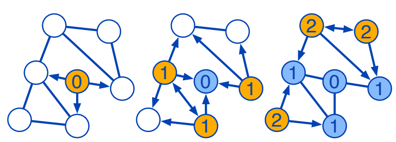
1. Gather vertices in *processor column* **[communication]**
2. Find owners of the current frontier's adjacency [computation]
3. Exchange adjacencies in *processor row* **[communication]**
4. Update distances/parents for unvisited vertices. [computation]

Performance observations of the level-synchronous algorithm



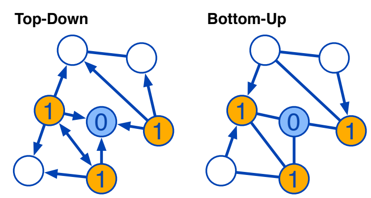


When the frontier is at its peak, almost all edge examinations "fail" to claim a child



Bottom-up BFS algorithm

Classical (top-down) algorithm is optimal in worst case, but pessimistic for low-diameter graphs (previous slide).



for all *v* in frontier attempt to parent ***all*** neighbors(*v*)

for all *v* in unvisited find ***any*** parent (neighbor(*v*) in frontier)

Direction Optimization:

- Switch from top-down to bottom-up search
- When the majority of the vertices are discovered.

[Read paper for exact heuristic]

Scott Beamer, Krste Asanović, and David Patterson, "Direction-Optimizing Breadth-First Search", *Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012

Direction optimizing BFS with 2D decomposition

- Adoption of the 2D algorithm created the *first quantum leap*
- The *second quantum leap* comes from the bottom-up search

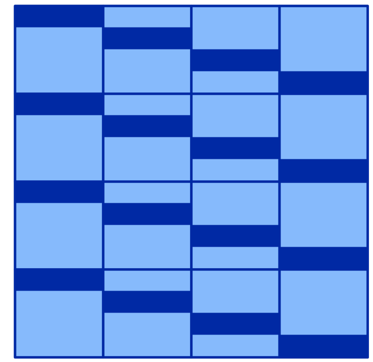
- Can we just do bottom-up on 1D?
- Yes, if you have ***in-network*** fast frontier membership queries
  - IBM by-passed MPI to achieve this [Checconi & Petrini, IPDPS'14]
  - Unrealistic and counter-productive in general

- 2D partitioning reduces the required frontier segment by a factor of p_c (typically $\sqrt{p}$), without fast in-network reductions
- **Challenge:** Inner loop is serialized

Direction optimizing BFS with 2D decomposition

Solution: Temporally partition the work

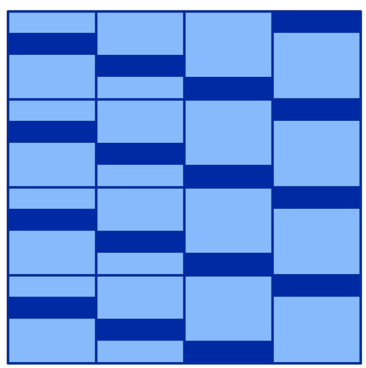
- *Temporal Division* - a vertex is processed by ***at most one processor*** at a time
- *Systolic Rotation* - send completion information to next processor so it knows what to skip



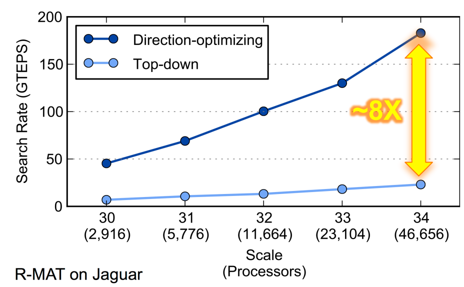
Direction optimizing BFS with 2D decomposition

Solution: Temporally partition the work

- *Temporal Division* - a vertex is processed by ***at most one processor*** at a time
- *Systolic Rotation* - send completion information to next processor so it knows what to skip



Direction optimizing BFS with 2D decomposition

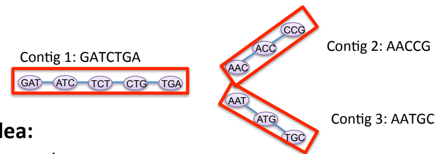


- ORNL Titan (Cray XK6, Gemini interconnect AMD Interlagos)
- Kronecker (Graph500): 16 billion vertices and 256 billion edges.

Scott Beamer, Aydın Buluç, Krste Asanović, and David Patterson, "Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search", *IPDPSW*, 2013

Parallel De Bruijn Graph Traversal

Goal:
- **Traverse** the de Bruijn graph and find UU contigs (chains of UU nodes), *or alternatively*
- find the connected components which consist of the UU contigs.



- **Main idea:**
  - Pick a seed
  - Iteratively extend it by consecutive lookups in the distributed hash table

Parallel De Bruijn Graph Traversal

Assume *one* of the UU contigs to be assembled is:

CGTATTGCCAATGCAACGTATCATGGCCAATCCGAT

Parallel De Bruijn Graph Traversal

Processor P_i picks a random k-mer from the distributed hash table as seed:

CGTATTGCCAAT GCAACGTATC A TGGCCAATCCGAT

P_i knows that forward extension is A

P_i uses the last k-1 bases and the forward extension and forms: CAACGTATCA

P_i does a lookup in the **distributed hash table** for CAACGTATCA

P_i iterates this process until it reaches the "right" endpoint of the UU contig

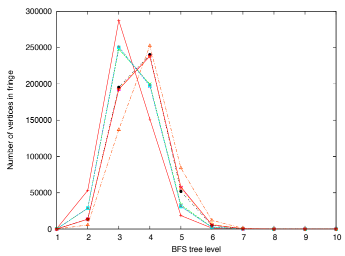
P_i also iterates this process backwards until it reaches the "left" endpoint of the UU contig

Multiple processors on the same UU contig

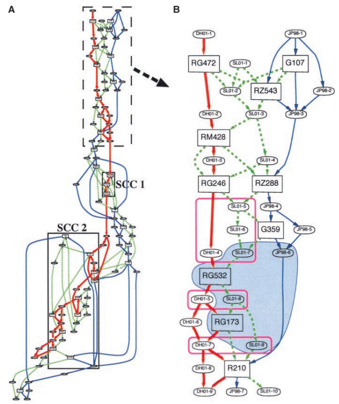
However, processors P_i, P_j and P_t might have picked initial seeds from the same UU contig

- Processors P_i, P_j and P_t have to collaborate and concatenate subcontigs in order to avoid redundant work.
- **Solution:** lightweight synchronization scheme based on a state machine

Moral: One traversal algorithm does not fit all graphs



Low diameter graph (R-MAT)
vs.
Long skinny graph (genomics)



Genetic linkage map, courtesy Yan et al.

Lecture Outline

- Applications
- Designing parallel graph algorithms
- Case studies:
  - A. **Graph traversals:** Breadth-first search
  - B. **Shortest Paths:** Delta-stepping, Floyd-Warshall
  - C. **Maximal Independent Sets:** Luby's algorithm
  - D. **Strongly Connected Components**
  - E. **Maximum Cardinality Matching**

Parallel Single-source Shortest Paths (SSSP) algorithms

- Famous serial algorithms:
  - **Bellman-Ford :** label correcting - works on any graph
  - **Dijkstra :** label setting – requires nonnegative edge weights
- No known PRAM algorithm that runs in sub-linear time and O(m+n log n) work
- Ullman-Yannakakis randomized approach
- Meyer and Sanders, Δ - stepping algorithm

U. Meyer and P.Sanders, Δ - stepping: a parallelizable shortest path algorithm. Journal of Algorithms 49 (2003)

- Chakaravarthy et al., clever combination of Δ - stepping and direction optimization (BFS) on supercomputer-scale graphs.

V. T. Chakaravarthy, F. Checconi, F. Petrini, Y. Sabharwal "Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems ", IPDPS'14

Δ - stepping algorithm

- *Label-correcting* algorithm: Can relax edges from unsettled vertices also
- "approximate bucket implementation of Dijkstra"
- For random edge weighs [0,1], runs in $O(n+m+D\cdot L)$

  where L = max distance from source to any node
- Vertices are ordered using buckets of width Δ
- Each bucket may be processed in parallel
- Basic operation: **Relax (e(u,v))**

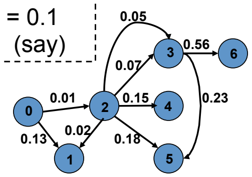
  d(v) = min { d(v), d(u) + w(u, v) }

Δ < min w(e) : Degenerates into Dijkstra

Δ > max w(e) : Degenerates into Bellman-Ford

Δ - stepping algorithm: illustration

Δ = 0.1 (say)



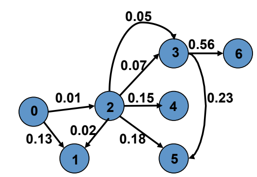
d array

0	1	2	3	4	5	6
∞	∞	∞	∞	∞	∞	∞

Buckets

One parallel **phase**
while (bucket is non-empty)
i) Inspect light (w < Δ) edges
ii) Construct a set of "requests" (R)
iii) Clear the current bucket
iv) Remember deleted vertices (S)
v) Relax request pairs in R
Relax heavy request pairs (from S)
Go on to the next bucket

Δ - stepping algorithm: illustration



d array

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞

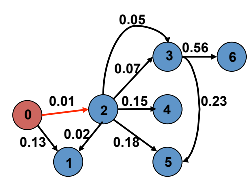
Buckets

0	0						

One parallel **phase**
while (bucket is non-empty)
i) Inspect light (w < Δ) edges
ii) Construct a set of "requests" (R)
iii) Clear the current bucket
iv) Remember deleted vertices (S)
v) Relax request pairs in R
Relax heavy request pairs (from S)
Go on to the next bucket

Initialization:
Insert *s* into bucket, d(s) = 0

Δ - stepping algorithm: illustration



d array

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞

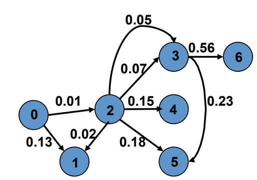
Buckets

0	0						

One parallel **phase**
while (bucket is non-empty)
i) Inspect light (w < Δ) edges
ii) Construct a set of "requests" (R)
iii) Clear the current bucket
iv) Remember deleted vertices (S)
v) Relax request pairs in R
Relax heavy request pairs (from S)
Go on to the next bucket

R	2						
	.01						
S							

Δ - stepping algorithm: illustration



d array

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞

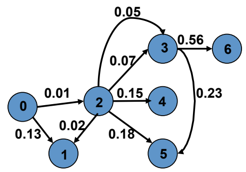
Buckets

0							

One parallel **phase**
while (bucket is non-empty)
i) Inspect light (w < Δ) edges
ii) Construct a set of "requests" (R)
iii) Clear the current bucket
iv) Remember deleted vertices (S)
v) Relax request pairs in R
Relax heavy request pairs (from S)
Go on to the next bucket

R	2						
	.01						
S	0						

Δ - stepping algorithm: illustration



One parallel **phase**
while (bucket is non-empty)
- i) Inspect light (w < Δ) edges
- ii) Construct a set of "requests" (R)
- iii) Clear the current bucket
- iv) Remember deleted vertices (S)
- v) Relax request pairs in R

Relax heavy request pairs (from S)
Go on to the next bucket

d array

0	1	2	3	4	5	6
0	∞	.01	∞	∞	∞	∞

Buckets

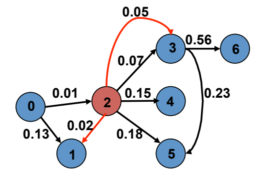
0	2						

R

S

0						

Δ - stepping algorithm: illustration



One parallel **phase**
while (bucket is non-empty)
- i) Inspect light (w < Δ) edges
- ii) Construct a set of "requests" (R)
- iii) Clear the current bucket
- iv) Remember deleted vertices (S)
- v) Relax request pairs in R

Relax heavy request pairs (from S)
Go on to the next bucket

d array

0	1	2	3	4	5	6
0	∞	.01	∞	∞	∞	∞

Buckets

0	2						

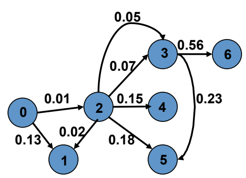
R

1	3					
.03	.06					

S

0						

Δ - stepping algorithm: illustration



One parallel **phase**
while (bucket is non-empty)
- i) Inspect light (w < Δ) edges
- ii) Construct a set of "requests" (R)
- iii) Clear the current bucket
- iv) Remember deleted vertices (S)
- v) Relax request pairs in R

Relax heavy request pairs (from S)
Go on to the next bucket

d array

0	1	2	3	4	5	6
0	∞	.01	∞	∞	∞	∞

Buckets

0							

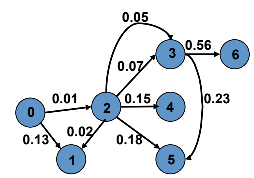
R

1	3					
.03	.06					

S

0	2					

Δ - stepping algorithm: illustration



One parallel **phase**
while (bucket is non-empty)
- i) Inspect light (w < Δ) edges
- ii) Construct a set of "requests" (R)
- iii) Clear the current bucket
- iv) Remember deleted vertices (S)
- v) Relax request pairs in R

Relax heavy request pairs (from S)
Go on to the next bucket

d array

0	1	2	3	4	5	6
0	.03	.01	.06	∞	∞	∞

Buckets

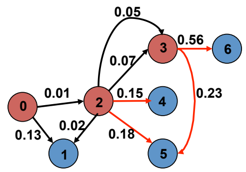
0	1	3					

R

S

0	2					

Δ - stepping algorithm: illustration



One parallel **phase**
while (bucket is non-empty)
i) Inspect light (w < Δ) edges
ii) Construct a set of "requests" (R)
iii) Clear the current bucket
iv) Remember deleted vertices (S)
v) Relax request pairs in R
Relax heavy request pairs (from S)
Go on to the next bucket

d array

0	1	2	3	4	5	6
0	.03	.01	.06	.16	.29	.62

Buckets

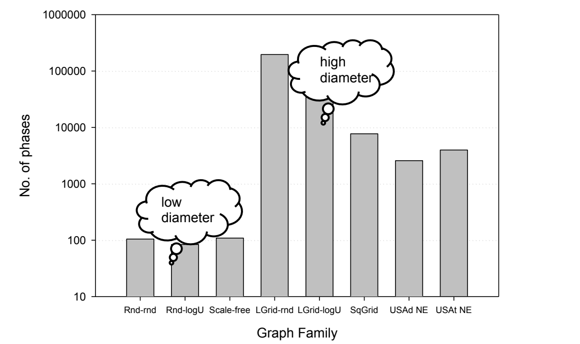
1	4						
2	5						
6	6						

R

S

0	2	1	3			

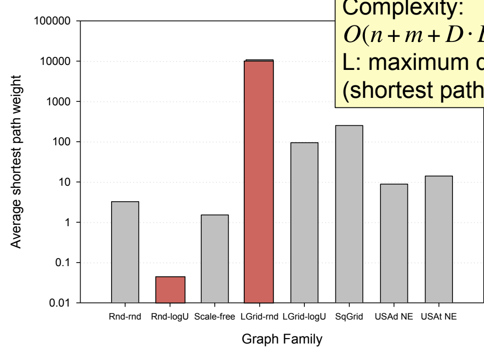
No. of *phases* (machine-independent performance count)



Too many phases in high diameter graphs:
Level-synchronous breadth-first search has the same problem.

Average shortest path weight for various graph families

~ 2^{20} vertices, 2^{22} edges, directed graph, edge weights normalized to [0,1]

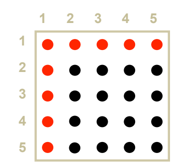


Complexity:
$O(n + m + D \cdot L)$
L: maximum distance (shortest path length)

All-pairs shortest-paths problem

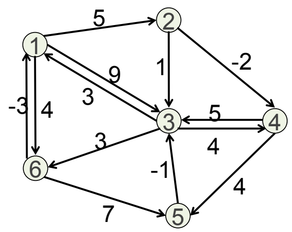
- Input: Directed graph with "costs" on edges
- Find least-cost paths between all reachable vertex pairs
- Classical algorithm: Floyd-Warshall

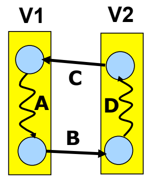
```
for k=1:n    // the induction sequence
  for i = 1:n
    for j = 1:n
        if(w(i→k)+w(k→j) < w(i→j))
            w(i→j):= w(i→k) + w(k→j)
```



k = 1 case

- It turns out a previously overlooked **recursive version** is more parallelizable than the triple nested loop



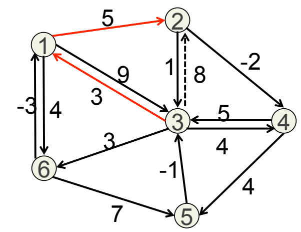


A	B
C	D

+ is "min", × is "add"

```
A = A*;     % recursive call
B = AB;  C = CA;
D = D + CB;
D = D*;     % recursive call
B = BD;  C = DC;
A = A + BC;
```

$$\left[\begin{array}{ccc|ccc} 0 & 5 & 9 & \infty & \infty & 4 \\ \infty & 0 & 1 & -2 & \infty & \infty \\ 3 & \infty & 0 & 4 & \infty & 3 \\ \hline \infty & \infty & 5 & 0 & 4 & \infty \\ \infty & \infty & -1 & \infty & 0 & \infty \\ -3 & \infty & \infty & \infty & 7 & 0 \end{array}\right]$$



$$\begin{bmatrix} \infty \\ 3 \end{bmatrix} \begin{bmatrix} 5 & 9 \end{bmatrix} = \begin{bmatrix} \infty & \infty \\ 8 & 12 \end{bmatrix}$$

C B

The cost of 3-1-2 path

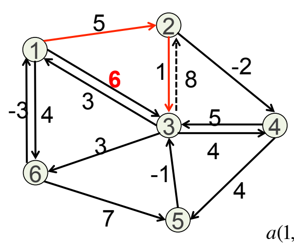
$$a(3,2) = a(3,1) + a(1,2) \xrightarrow{then} \Pi(3,2) = \Pi(1,2)$$

$$\left[\begin{array}{ccc|ccc} 0 & 5 & 9 & \infty & \infty & 4 \\ \infty & 0 & 1 & -2 & \infty & \infty \\ 3 & \mathbf{8} & 0 & 4 & \infty & 3 \\ \hline \infty & \infty & 5 & 0 & 4 & \infty \\ \infty & \infty & -1 & \infty & 0 & \infty \\ -3 & \infty & \infty & \infty & 7 & 0 \end{array}\right]$$

Distances

$$\Pi = \left[\begin{array}{ccc|ccc} 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & \mathbf{1} & 3 & 3 & 3 & 3 \\ \hline 4 & 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 & 6 & 6 \end{array}\right]$$

Parents



D = D*: no change

$$\begin{bmatrix} 5 & 9 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 8 & 0 \end{bmatrix} = \begin{bmatrix} 5 & 6 \end{bmatrix}$$

B D

Path: 1-2-3

$$a(1,3) = a(1,2) + a(2,3) \xrightarrow{then} \Pi(1,3) = \Pi(2,3)$$

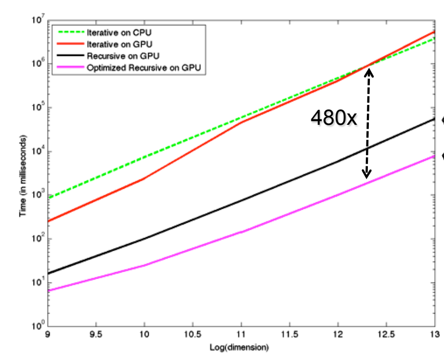
$$\left[\begin{array}{ccc|ccc} 0 & 5 & \mathbf{6} & \infty & \infty & 4 \\ \infty & 0 & 1 & -2 & \infty & \infty \\ 3 & 8 & 0 & 4 & \infty & 3 \\ \hline \infty & \infty & 5 & 0 & 4 & \infty \\ \infty & \infty & -1 & \infty & 0 & \infty \\ -3 & \infty & \infty & \infty & 7 & 0 \end{array}\right]$$

Distances

$$\Pi = \left[\begin{array}{ccc|ccc} 1 & 1 & \mathbf{2} & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 1 & 3 & 3 & 3 & 3 \\ \hline 4 & 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 & 6 & 6 \end{array}\right]$$

Parents

All-pairs shortest-paths problem



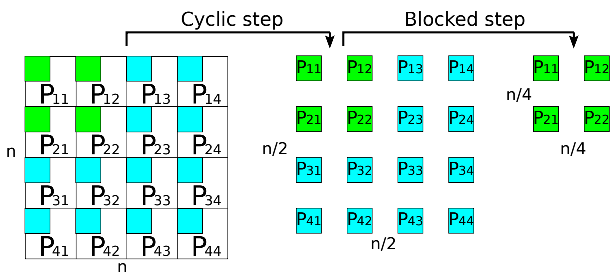
Floyd-Warshall ported to GPU

Naïve recursive implementation

The right primitive (Matrix multiply)

A. Buluç, J. R. Gilbert, and C. Budak. Solving path problems on the GPU. Parallel Computing, 36(5-6):241 - 253, 2010.

Communication-avoiding APSP in distributed memory



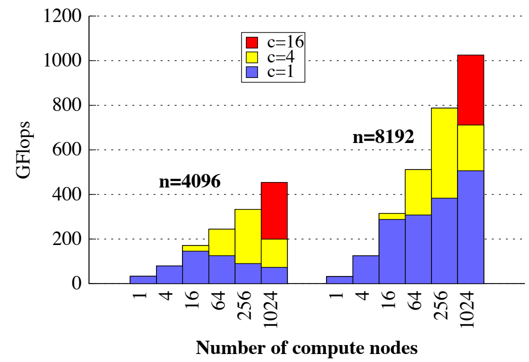
Bandwidth: $W_{\text{bc-2.5D}}(n,p) = O(n^2/\sqrt{cp})$

Latency: $S_{\text{bc-2.5D}}(p) = O\left(\sqrt{cp}\log^2(p)\right)$

c: number of replicas

Optimal for any memory size !

Communication-avoiding APSP in distributed memory



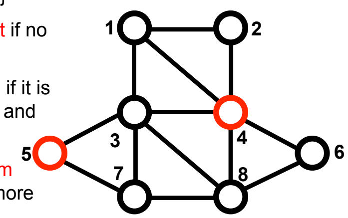
E. Solomonik, A. Buluç, and J. Demmel. Minimizing communication in all-pairs shortest paths. In Proceedings of the IPDPS. 2013.

Lecture Outline

- Applications
- Designing parallel graph algorithms
- Case studies:
  - A. **Graph traversals:** Breadth-first search
  - B. **Shortest Paths:** Delta-stepping, Floyd-Warshall
  - C. **Maximal Independent Sets:** Luby's algorithm
  - D. **Strongly Connected Components**
  - E. **Maximum Cardinality Matching**

Maximal Independent Set

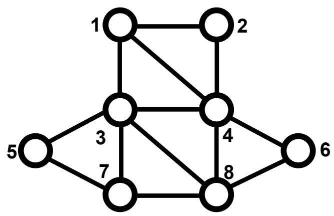
- Graph with vertices V = {1,2,…,n}
- A set S of vertices is independent if no two vertices in S are neighbors.
- An independent set S is maximal if it is impossible to add another vertex and stay independent
- An independent set S is maximum if no other independent set has more vertices
- Finding a *maximum* independent set is intractably difficult (NP-hard)
- Finding a *maximal* independent set is easy, at least on one processor.



The set of red vertices S = {4, 5} is *independent* and is *maximal* but not *maximum*

Sequential Maximal Independent Set Algorithm

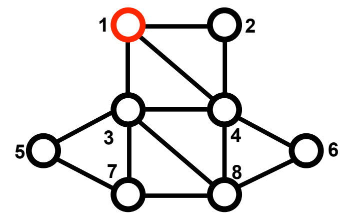
```
S = empty set;
for  vertex v = 1 to n {
    if (v has no neighbor in S) {
        add v to S
    }
}
```



S = { }

Sequential Maximal Independent Set Algorithm

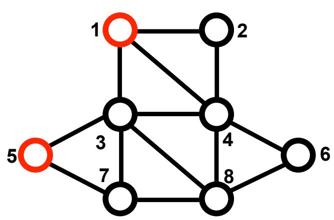
```
S = empty set;
for  vertex v = 1 to n {
    if (v has no neighbor in S) {
        add v to S
    }
}
```



S = { 1 }

Sequential Maximal Independent Set Algorithm

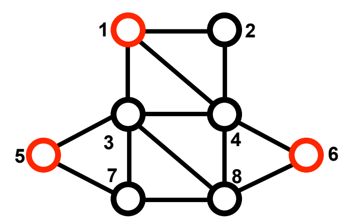
```
S = empty set;
for  vertex v = 1 to n {
    if (v has no neighbor in S) {
        add v to S
    }
}
```



S = { 1, 5 }

Sequential Maximal Independent Set Algorithm

```
S = empty set;
for  vertex v = 1 to n {
    if (v has no neighbor in S) {
        add v to S
    }
}
```

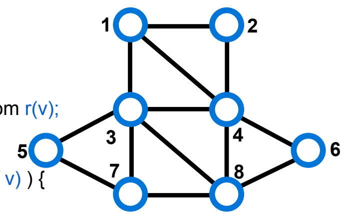


S = { 1, 5, 6 }

work ~ O(n), but *span* ~O(n)

Parallel, Randomized MIS Algorithm

```
S = empty set;  C = V;
while  C  is not empty {
    label each v in C with a random r(v);
    for all v in C in parallel {
        if r(v) < min( r(neighbors of v) ) {
            move v from C to S;
            remove neighbors of v from C;
        }
    }
}
```



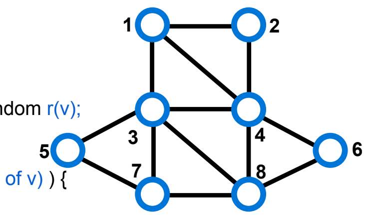
S = { }

C = { 1, 2, 3, 4, 5, 6, 7, 8 }

M. Luby. "A Simple Parallel Algorithm for the Maximal Independent Set Problem". *SIAM Journal on Computing* **15** (4): 1036–1053, 1986

Parallel, Randomized MIS Algorithm

```
S = empty set;  C = V;
while  C  is not empty {
    label each v in C with a random r(v);
    for all v in C in parallel {
        if r(v) < min( r(neighbors of v) ) {
            move v from C to S;
            remove neighbors of v from C;
        }
    }
}
```

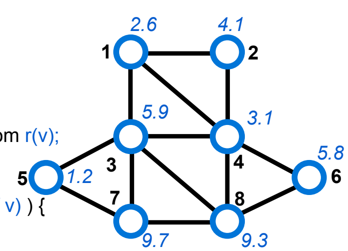


S = { }

C = { 1, 2, 3, 4, 5, 6, 7, 8 }

Parallel, Randomized MIS Algorithm

```
S = empty set;  C = V;
while  C  is not empty {
    label each v in C with a random r(v);
    for all v in C in parallel {
        if r(v) < min( r(neighbors of v) ) {
            move v from C to S;
            remove neighbors of v from C;
        }
    }
}
```

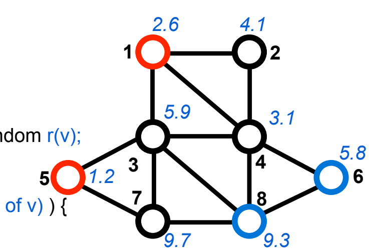


S = { }

C = { 1, 2, 3, 4, 5, 6, 7, 8 }

Parallel, Randomized MIS Algorithm

```
S = empty set;  C = V;
while  C  is not empty {
    label each v in C with a random r(v);
    for all v in C in parallel {
        if r(v) < min( r(neighbors of v) ) {
            move v from C to S;
            remove neighbors of v from C;
        }
    }
}
```

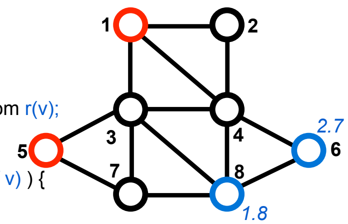


S = { 1, 5 }

C = { 6, 8 }

Parallel, Randomized MIS Algorithm

```
S = empty set;  C = V;
while  C  is not empty {
    label each v in C with a random r(v);
    for all v in C in parallel {
        if r(v) < min( r(neighbors of v) ) {
            move v from C to S;
            remove neighbors of v from C;
        }
    }
}
```

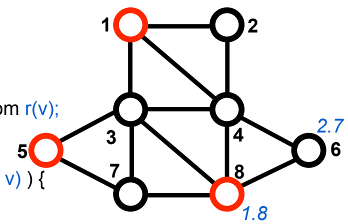


S = { 1, 5 }

C = { 6, 8 }

Parallel, Randomized MIS Algorithm

```
S = empty set;  C = V;
while  C  is not empty {
    label each v in C with a random r(v);
    for all v in C in parallel {
        if r(v) < min( r(neighbors of v) ) {
            move v from C to S;
            remove neighbors of v from C;
        }
    }
}
```

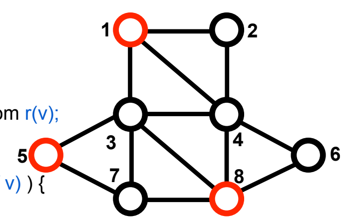


S = { 1, 5, 8 }

C = { }

Parallel, Randomized MIS Algorithm

```
S = empty set;  C = V;
while  C  is not empty {
    label each v in C with a random r(v);
    for all v in C in parallel {
        if r(v) < min( r(neighbors of v) ) {
            move v from C to S;
            remove neighbors of v from C;
        }
    }
}
```



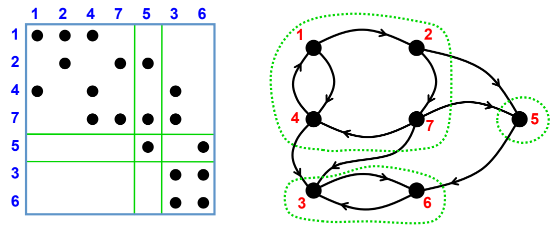
Theorem: This algorithm "very probably" finishes within O(log n) rounds.

***work* ~ O(n log n), but *span* ~O(log n)**

Lecture Outline

- Applications
- Designing parallel graph algorithms
- Case studies:
  - A. **Graph traversals:** Breadth-first search
  - B. **Shortest Paths:** Delta-stepping, Floyd-Warshall
  - C. **Maximal Independent Sets:** Luby's algorithm
  - D. **Strongly Connected Components**
  - E. **Maximum Cardinality Matching**

Strongly connected components (SCC)



- Symmetric permutation to block triangular form
- Find P in linear time by depth-first search

Tarjan, R. E. (1972), "Depth-first search and linear graph algorithms", SIAM Journal on Computing 1 (2): 146–160

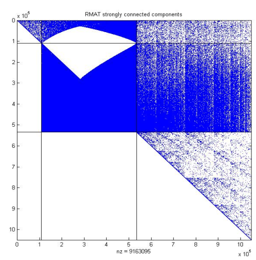
Strongly connected components of directed graph

- Sequential: use depth-first search (Tarjan); work=O(m+n) for m=|E|, n=|V|.
- DFS seems to be inherently sequential.
- Parallel: divide-and-conquer and BFS (Fleischer et al.); worst-case span O(n) but good in practice on many graphs.

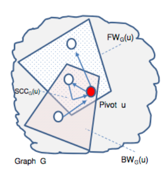
L. Fleischer, B. Hendrickson, and A. Pınar. On identifying strongly connected components in parallel. Parallel and Distributed Processing, pages 505–511, 2000.

Fleischer/Hendrickson/Pinar algorithm

- Partition the given graph into three disjoint subgraphs
- Each can be processed independently/recursively



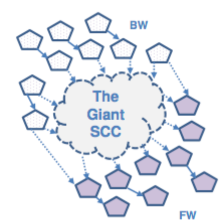
Lemma: FW(v) ∩ BW(v) is a unique SCC for any v. For every other SCC s, either
(a) s ⊂ FW(v)\BW(v),
(b) s ⊂ BW(v)\FW(v),
(c) s ⊂ V \ (FW(v) ∪ BW(v)).



FW(v): vertices reachable from vertex v.
BW(v): vertices from which v is reachable.

Improving FW/BW with parallel BFS

Observation: Real world graphs have giant SCCs



Finding FW(pivot) and BW(pivot) can dominate the running time with span=O(N)

Solution: Use ***parallel BFS*** to limit span to diameter(SCC)

- Remaining SCCs are very small; increasing span of the recursion.
+ *Find weakly-connected components and process them in parallel*

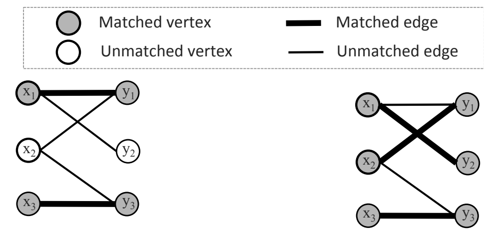
S. Hong, N.C. Rodia, and K. Olukotun. On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-World Graphs. Proc. Supercomputing, 2013

Lecture Outline

- Applications
- Designing parallel graph algorithms
- Case studies:
  - A. **Graph traversals:** Breadth-first search
  - B. **Shortest Paths:** Delta-stepping, Floyd-Warshall
  - C. **Maximal Independent Sets:** Luby's algorithm
  - D. **Strongly Connected Components**
  - E. **Maximum Cardinality Matching**

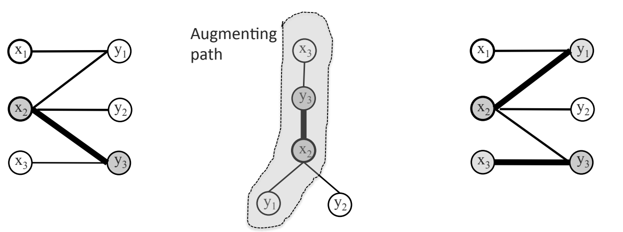
Bipartite Graph Matching

- Matching: A subset *M* of edges with no common end vertices.
  - |M| = Cardinality of the matching M



A Matching (**Maximal** cardinality) **Maximum** Cardinality Matching

Single-Source Algorithm for Maximum Cardinality Matching

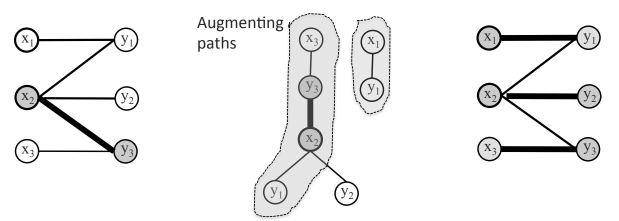


1. Initial matching
2. Search for augmenting path from x_3. stop when an unmatched vertex found
3. Increase matching by flipping edges in the augmenting path

Repeat the process for other unmatched vertices

Multi-Source Algorithm for Maximum Cardinality Matching

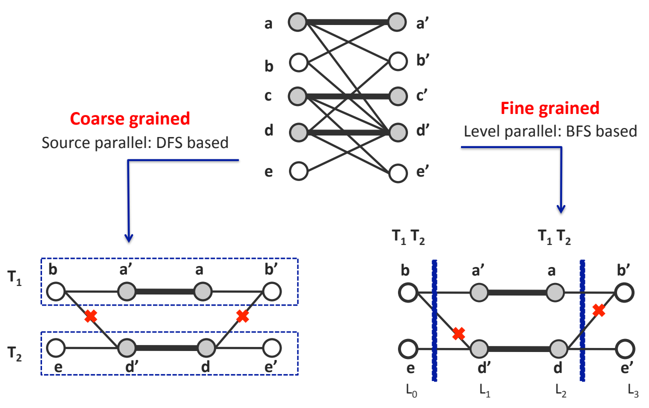
Search Forest



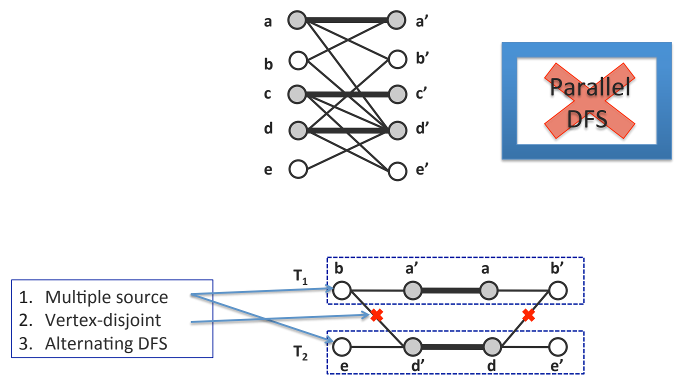
1. Initial matching
2. Search for **vertex-disjoint** augmenting paths from x_3 & x_1. Grow a tree until an unmatched vertex is found in it
3. Increase matching by flipping edges in the augmenting paths

Repeat the process for until no augmenting path is found

Parallelizing Augmenting Path Searches

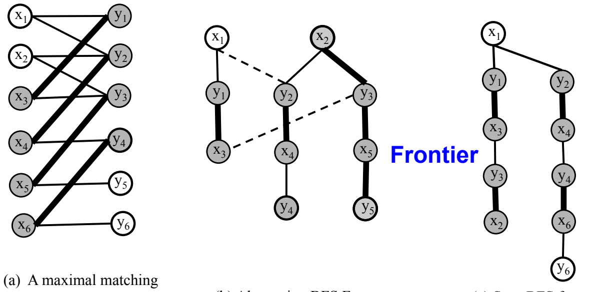


Parallelizing Augmenting Path Searches



Limitation of Current Multi-source Algorithms

Previous algorithms destroy both trees and start searching from x_1 again

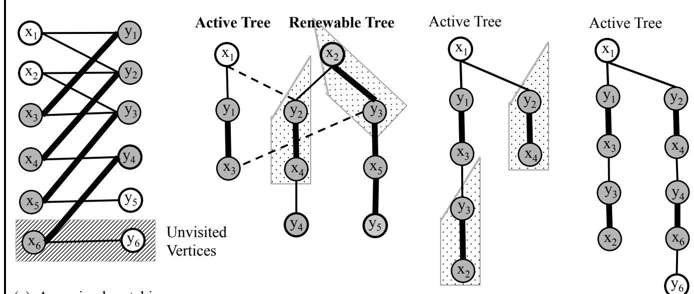


(a) A maximal matching in a Bipartite Graph

(b) Alternating BFS Forest Augment in forest

(c) Start BFS from x_1

Tree Grafting Mechanism



(a) A maximal matching in a Bipartite Graph

(b) Alternating BFS Forest Augment in forest

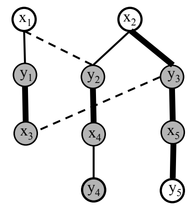
(c) Tree Grafting

(d) Continue BFS

Ariful Azad, Aydin Buluç, and Alex Pothen. A parallel tree grafting algorithm for maximum cardinality matching in bipartite graphs. In Proceedings of the IPDPS, 2015

Parallel Tree Grafting

1. Parallel direction optimized BFS (Beamer et al. SC 2012)
   – Use bottom-up BFS when the frontier is large



Maintain visited array

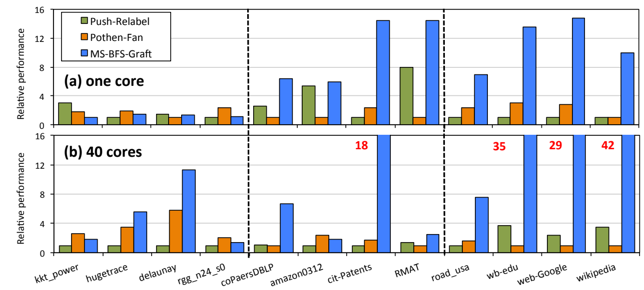
To maintain vertex-disjoint paths, a vertex is visited only once in an iteration.

Thread-safe atomics

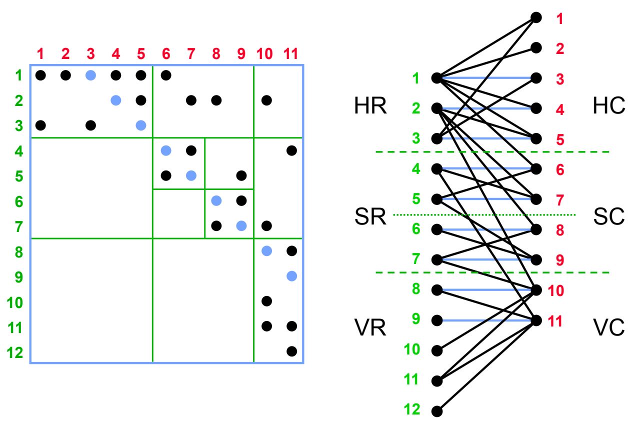
2. Since the augmenting paths are vertex disjoint **we can augment them in parallel**
3. Each renewable vertex tries to attach itself to an active vertex. **No synchronization necessary**

Performance of the tree-grafting algorithm

Pothen-Fan: Azad et al. IPDPS 2012
Push-Relabel: Langguth et al. Parallel Computing 2014



Dulmage-Mendelsohn decomposition



Dulmage-Mendelsohn decomposition

1. ***Find a "perfect matching" in the bipartite graph of the matrix.***
2. ***Permute the matrix to have a zero free diagonal.***
3. ***Find the "strongly connected components" of the directed graph of the permuted matrix.***

Let M be a maximum-size matching. Define:

VR = { rows reachable via alt. path from some ***unmatched row*** }

VC = { cols reachable via alt. path from some ***unmatched row*** }

HR = { rows reachable via alt. path from some ***unmatched col*** }

HC = { cols reachable via alt. path from some ***unmatched col*** }

SR = R – VR – HR

SC = C – VC – HC

Applications of D-M decomposition

- Strongly connected components of directed graphs
- Connected components of undirected graphs
- Permutation to block triangular form for Ax=b
- Minimum-size vertex cover of bipartite graphs
- Extracting vertex separators from edge cuts for arbitrary graphs
- Nonzero structure prediction for sparse matrix factorizations