Parallel Graph Algorithms

Aydın Buluç
ABuluc@lbl.gov
http://gauss.cs.ucsb.edu/~aydin/
Lawrence Berkeley National Laboratory

CS267, Spring 2014
April 15, 2014

Slide acknowledgments: K. Madduri, J. Gilbert, D. Delling, S. Beamer

Graph Preliminaries

Define: \textbf{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)

$D = \text{diameter}$ (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, PHAST, Floyd-Warshall
  C. Ranking: Betweenness centrality
  D. Maximal Independent Sets: Luby’s algorithm
  E. Strongly Connected Components
- Wrap-up: challenges on current systems
Routing in transportation networks

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


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

- 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

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
Graph-theoretic problems in social networks

- Targeted advertising: clustering and centrality
- Studying the spread of information

Network Analysis for Neurosciences

Graph-theoretical models are used to predict the course of degenerative illnesses like Alzheimer’s.

Vertices: ROI (regions of interest)
Edges: structural/functional connectivity

Some disease indicators:
- Deviation from small-world property
- Emergence of “epicenters” with disease-associated patterns

Research in Parallel Graph Algorithms

<table>
<thead>
<tr>
<th>Application Areas</th>
<th>Methods/Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Social Network Analysis</td>
<td>Find central entities Community detection Network dynamics</td>
</tr>
<tr>
<td>WWW</td>
<td>Marketing Social search</td>
</tr>
<tr>
<td>Computational Biology</td>
<td>Gene regulation Metabolic pathways Genomics</td>
</tr>
<tr>
<td>Scientific Computing</td>
<td>Graph partitioning Matching Coloring</td>
</tr>
<tr>
<td>Engineering</td>
<td>VLSI CAD Route planning</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph Algorithms</th>
<th>Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traversal</td>
<td>GPUs, FPGAs</td>
</tr>
<tr>
<td>Shortest Paths</td>
<td>x86 multicores</td>
</tr>
<tr>
<td>Connectivity</td>
<td>Massively multithreaded architectures (Cray XMT, Sun Niagara)</td>
</tr>
<tr>
<td>Max Flow</td>
<td>Multicore clusters (NERSC Hopper)</td>
</tr>
<tr>
<td></td>
<td>Clouds (Amazon EC2)</td>
</tr>
</tbody>
</table>

Characterizing Graph-theoretic computations

Input data

Problem: Find **

- paths
- clusters
- partitions
- matchings
- patterns
- orderings

Graph kernel

- traversal
- shortest path algorithms
- flow algorithms
- spanning tree algorithms
- topological sort
- ……

Factors that influence choice of algorithm

- graph sparsity (m/n ratio)
- static/dynamic nature
- weighted/unweighted, weight distribution
- vertex degree distribution
- directed/undirected
- simple/multi/hyper graph
- problem size
- granularity of computation at nodes/edges
- domain-specific characteristics

Graph problems are often recast as sparse linear algebra (e.g., partitioning) or linear programming (e.g., matching) computations
Lecture Outline

- Applications
- Designing parallel graph algorithms
- Case studies:
  A. Graph traversals: Breadth-first search
  B. Shortest Paths: Delta-stepping, PHAST, Floyd-Warshall
  C. Ranking: Betweenness centrality
  D. Maximal Independent Sets: Luby's algorithm
  E. Strongly Connected Components
- Wrap-up: challenges on current systems

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.

Building blocks of classical PRAM graph algorithms

- Prefix sums
- Symmetry breaking
- Pointer jumping
- List ranking
- Euler tours
- Vertex collapse
- Tree contraction
  [some covered in the “Tricks with Trees” lecture]
Data structures: graph representation

Static case
• Dense graphs \((m = \Theta(n^2))\): adjacency matrix commonly used.
• Sparse graphs: adjacency lists, compressed sparse matrices

Dynamic
• representation depends on common-case query
• Edge insertions or deletions? Vertex insertions or deletions? Edge weight updates?
• Graph update rate
• Queries: connectivity, paths, flow, etc.
• Optimizing for locality a key design consideration.

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

Graph representations

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

2D checkerboard distribution

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

Per-processor local graph representation

Flatten Sparse matrices

9 vertices, 9 processors, 3x3 processor grid
High-performance graph algorithms

- Implementations are typically array-based for performance (e.g. CSR representation).
- Concurrency = minimize synchronization (span)
- Where is the data? Find the distribution that minimizes inter-processor communication.
- Memory access patterns
  - Regularize them (spatial locality)
  - Minimize DRAM traffic (temporal locality)
- Work-efficiency
  - Is (Parallel time) * (# of processors) = (Serial work)?

Lecture Outline

- Applications
- Designing parallel graph algorithms
- Case studies:
  A. Graph traversals: Breadth-first search
  B. Shortest Paths: Delta-stepping, PHAST, Floyd-Warshall
  C. Ranking: Betweenness centrality
  D. Maximal Independent Sets: Luby’s algorithm
  E. Strongly Connected Components
- Wrap-up: challenges on current systems
Parallel BFS Strategies

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

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

Bottom-up BFS algorithm

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

Direction Optimization:
- Switch from top-down to bottom-up search
- When the majority of the vertices are discovered.
[Read paper for exact heuristic]

Performance observations of the level-synchronous algorithm

When the frontier is at its peak, almost all edge examinations “fail” to claim a child
Replace scalar operations

Multiply -> select
Add -> minimum

Select vertex with minimum label as parent

parents:

parents:
Insight!

**Breadth-First Search**

- Small-world graphs are harder to partition
- Block for many graph algorithms

Breadth-First Search (BFS) is a key building

*Top-Down*

examined unnecessarily

*Bottom-Up*

Buluç

Direction-Optimizing Breadth-First Search

Hybrid of Top-Down & Bottom-Up

for all

$
\begin{align*}
\text{In-node} & \\
\text{6-way} & \\
\text{BT} & \\
& \\
\text{& growing} & \\
\text{BT} & \\
\end{align*}
$

Update
distances/parents
for
unvisited
vertices.

Find
owners
of
the
current
frontier’s
adjacency

**BFS Strong Scaling**

- NERSC Hopper (Cray X6, Gemini interconnect AMD Magny-Cours)
- Hybrid: In-node 6-way OpenMP multithreading
- Kronecker (Graph500): 4 billion vertices and 64 billion edges.

**Direction optimizing BFS with 2D decomposition**

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

- Applications
- Designing parallel graph algorithms
- Case studies:
  A. Graph traversals: Breadth-first search
  B. Shortest Paths: Delta-stepping, PHAST, Floyd-Warshall
  C. Ranking: Betweenness centrality
  D. Maximal Independent Sets: Luby’s algorithm
  E. Strongly Connected Components
- Wrap-up: challenges on current systems

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, $\Delta$ - stepping algorithm

$\Delta$ - stepping algorithm

- Label-correcting algorithm: Can relax edges from unsettled vertices also
- “approximate bucket implementation of Dijkstra”
- For random edge weights $[0,1]$, runs in $O(n+m+D \cdot L)$ where $L = \text{max distance from source to any node}$
- Vertices are ordered using buckets of width $\Delta$
- Each bucket may be processed in parallel
- Basic operation: $\text{Relax (} e(u,v) \text{)}$
  
  $d(v) = \min \{d(v), d(u) + w(u,v)\}$

- $\Delta < \min w(e)$: Degenerates into Dijkstra
- $\Delta > \max w(e)$: Degenerates into Bellman-Ford

$\Delta$ - stepping algorithm: illustration

- $\Delta = 0.1$ (say)

One parallel phase

```
while (bucket is non-empty)
  i) Inspect light ($w < \Delta$) edges
  ii) Construct a set of “requests” (R)
  iii) Clear the current bucket
  iv) Remember deleted vertices (S)
  v) Relax request pairs in R
Go on to the next bucket
```

Buckets

```

<table>
<thead>
<tr>
<th>Bucket</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0.01</td>
<td>0.15</td>
<td>0.65</td>
<td>0.02</td>
<td>0.18</td>
<td>0.23</td>
<td>0.56</td>
</tr>
</tbody>
</table>
```

$\Delta$-stepping algorithm:

```
\text{array} (d)

\begin{align*}
\text{Bucket 0:} & \quad \{1, 0.01, 0.15, 0.65\} \\
\text{Bucket 1:} & \quad \{0.02, 0.18, 0.23, 0.56\}
\end{align*}
```
Δ - stepping algorithm: illustration

One parallel phase
while (bucket is non-empty)
   i) Inspect light (\(w < \Delta\)) 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

One parallel phase
while (bucket is non-empty)
  i) Inspect light \((w < \Delta)\) 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

Buckets

0 1 2

\[
\begin{array}{ccccccc}
\hline
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
0 & \infty & 0.01 & \infty & 0.15 & \infty & \infty \\
\hline
\end{array}
\]
Too many phases in high diameter graphs: Level-synchronous breadth-first search has the same problem.

**PHAST – hardware accelerated shortest path trees**

Preprocessing: Build a contraction hierarchy
- order nodes by importance (highway dimension)
- process in order
- add shortcuts to preserve distances
- assign levels (ca. 150 in road networks)
- 75% increase in number of edges (for road networks)

**Complexity:**

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

PHAST – hardware accelerated shortest path trees

One-to-all search from source s:
- Run forward search from s
- Only follow edges to more important nodes
- Set distance labels \( d \) of reached nodes

From top-down:
- linear sweep without priority queue
- reorder nodes, arcs, distance labels by level
- accesses to \( d \) array becomes contiguous (no jumps)
- parallelize with SIMD (SSE instructions, and/or GPU)

PHAST – hardware accelerated shortest path trees

Replacing Dijkstra
- process all nodes \( u \) in reverse level order:
  - check incoming arcs \( (v,u) \) with \( \text{lev}(v) > \text{lev}(u) \)
  - Set \( d(u) = \min\{d(u), d(v) + w(v,u)\} \)

PHAST – performance comparison

<table>
<thead>
<tr>
<th>Input: Europe/USA Road Networks</th>
<th>Europe</th>
<th>USA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra M1-4</td>
<td>1104.52</td>
<td>618.18</td>
</tr>
<tr>
<td></td>
<td>380.40</td>
<td>280.17</td>
</tr>
<tr>
<td>Dijkstra M2-6</td>
<td>288.81</td>
<td>177.58</td>
</tr>
<tr>
<td></td>
<td>229.00</td>
<td>167.77</td>
</tr>
<tr>
<td>PHAST M1-4</td>
<td>19.47</td>
<td>23.01</td>
</tr>
<tr>
<td>PHAST M2-6</td>
<td>7.20</td>
<td>8.27</td>
</tr>
<tr>
<td>PHAST M4-12</td>
<td>4.03</td>
<td>5.03</td>
</tr>
<tr>
<td>GPFAST GTX 480</td>
<td>2.90</td>
<td>4.75</td>
</tr>
<tr>
<td>GPU implementation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Edge weights: estimated travel times

Edge weights: physical distances
**PHAST – hardware accelerated shortest path trees**

- Specifically designed for road networks
- Fill-in can be much higher for other graphs (Hint: think about sparse Gaussian Elimination)

Road networks are (almost) planar.

Planar graphs have $O(\sqrt{n})$ separators.

Good separators lead to orderings with minimal fill.

---

**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

It turns out a previously overlooked recursive version is more parallelizable than the triple nested loop
All-pairs shortest-paths problem

Floyd-Warshall ported to GPU

Naïve recursive implementation

The right implementation (Matrix multiply)

Communication-avoiding APSP in distributed memory

Bandwidth: $W_{D25D}(n,p) = O(n^2/\sqrt{p})$

Latency: $S_{D25D}(p) = O(\sqrt{cp} \log^2(p))$

Optimal for any memory size!
Lecture Outline

- Applications
- Designing parallel graph algorithms
- Case studies:
  A. Graph traversals: Breadth-first search
  B. Shortest Paths: Delta-stepping, PHAST, Floyd-Warshall
  C. Ranking: Betweenness centrality
  D. Maximal Independent Sets: Luby’s algorithm
- Strongly Connected Components
- Wrap-up: challenges on current systems

Betweenness Centrality (BC)

- Centrality: Quantitative measure to capture the importance of a vertex/edge in a graph
  - degree, closeness, eigenvalue, betweenness
- Betweenness Centrality
  \[ BC(v) = \sum_{s \neq w \neq t} \frac{\sigma_s(v)\sigma_t(w)}{\sigma_s(w)} \]
  ( \( \sigma_s(w) \): No. of shortest paths between \( s \) and \( t \))
- Applied to several real-world networks
  - Social interactions
  - WWW
  - Epidemiology
  - Systems biology

Algorithms for Computing Betweenness

- All-pairs shortest path approach: compute the length and number of shortest paths between all \( s-t \) pairs (\( O(n^3) \) time), sum up the fractional dependency values (\( O(n^2) \) space).
- Brandes’ algorithm (2003): Augment a single-source shortest path computation to count paths; uses the Bellman criterion; \( O(mn) \) work and \( O(m+n) \) space on unweighted graph.

\[
\delta(v) = \sum_{n: \text{head}(v) \neq n} \frac{\sigma_s(v)\sigma_t(w)}{\sigma_s(w)} (1 + \delta(w))
\]

Dependency of source on \( v \).

Number of shortest paths from source to \( v \).

Parallel BC algorithms for unweighted graphs

- High-level idea: Level-synchronous parallel breadth-first search augmented to compute centrality scores
- Two steps (both implemented as BFS)
  - traversal and path counting
  - dependency accumulation

Shared-memory parallel algorithms for betweenness centrality

Exact algorithm: \( O(mn) \) work, \( O(m+n) \) space, \( O(nD+nm/p) \) time.

Improved with lower synchronization overhead and fewer non-contiguous memory references.

Parallel BC Algorithm Illustration

1. **Traversal step**: visit adjacent vertices, update distance and path counts.

Level-synchronous approach: The adjacencies of all vertices in the current frontier can be visited in parallel.
1. **Traversal step**: at the end, we have all reachable vertices, their corresponding predecessor multisets, and D values.

2. **Accumulation step**: Can also be done in a level-synchronous manner.

Level-synchronous approach: The adjacencies of all vertices in the current frontier can be visited in parallel.

### Parallel BC Algorithm Illustration

**Parallel BC Algorithm Illustration**

1. **Traversal step**: at the end, we have all reachable vertices, their corresponding predecessor multisets, and D values.

2. **Accumulation step**: Can also be done in a level-synchronous manner.

### Distributed memory BC algorithm

Work-efficient parallel breadth-first search via parallel sparse matrix-matrix multiplication over semirings

Encapsulates three levels of parallelism:
1. columns(B): multiple BFS searches in parallel
2. columns($A^T$)+rows(B): parallel over frontier vertices in each BFS
3. rows($A^T$): parallel over incident edges of each frontier vertex

---

Lecture Outline

- Applications
- Designing parallel graph algorithms
- Case studies:
  A. Graph traversals: Breadth-first search
  B. Shortest Paths: Delta-stepping, PHAST, Floyd-Warshall
  C. Ranking: Betweenness centrality
  D. Maximal Independent Sets: Luby’s algorithm
  E. Strongly Connected Components
- Wrap-up: challenges on current systems

Sequential Maximal Independent Set Algorithm

1. \( S = \text{empty set}; \)
2. for vertex \( v = 1 \) to \( n \) {
3.  if (\( v \) has no neighbor in \( S \)) {
4.    add \( v \) to \( S \)
5.  }
6. }

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.

Sequential Maximal Independent Set Algorithm

1. \( S = \text{empty set}; \)
2. for vertex \( v = 1 \) to \( n \) {
3.  if (\( v \) has no neighbor in \( S \)) {
4.    add \( v \) to \( S \)
5.  }
6. }

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

\[ S = \{1\} \]
Sequential Maximal Independent Set Algorithm

1. $S = \emptyset$;
2. for vertex $v = 1$ to $n$
   3. if ($v$ has no neighbor in $S$)
      4. add $v$ to $S$
   5. 
6. 

$S = \{1, 5\}$

Parallel, Randomized MIS Algorithm

1. $S = \emptyset$; $C = V$;
2. while $C$ is not empty
   3. label each $v$ in $C$ with a random $r(v)$;
   4. for all $v$ in $C$ in parallel
      5. if $r(v) < \min(r($neighbors of $v$))$
         6. move $v$ from $C$ to $S$;
         7. remove neighbors of $v$ from $C$;
   8. 
9. 
10. 

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


Sequential Maximal Independent Set Algorithm

1. $S = \emptyset$;
2. for vertex $v = 1$ to $n$
   3. if ($v$ has no neighbor in $S$)
      4. add $v$ to $S$
   5. 
6. 

$S = \{1, 5, 6\}$

work $\sim O(n)$, but span $\sim O(n)$

Parallel, Randomized MIS Algorithm

1. $S = \emptyset$; $C = V$;
2. while $C$ is not empty
   3. label each $v$ in $C$ with a random $r(v)$;
   4. for all $v$ in $C$ in parallel
      5. if $r(v) < \min(r($neighbors of $v$))$
         6. move $v$ from $C$ to $S$;
         7. remove neighbors of $v$ from $C$;
   8. 
9. 
10. 

$S = \{\}$ $C = \{1, 2, 3, 4, 5, 6, 7, 8\}$
Parallel, Randomized MIS Algorithm

1. \( S = \text{empty set}; \ C = V; \)
2. \( \text{while } C \text{ is not empty } \)
3. \( \text{label each } v \text{ in } C \text{ with a random } r(v); \)
4. \( \text{for all } v \text{ in } C \text{ in parallel } \)
5. \( \text{if } r(v) < \min( r(\text{neighbors of } v) ) \) \}
6. \( \text{move } v \text{ from } C \text{ to } S; \)
7. \( \text{remove neighbors of } v \text{ from } C; \)
8. \( \}
9. \( \}
10. \)

\[ S = \{1, 5\} \]
\[ C = \{6, 8\} \]
Parallel, Randomized MIS Algorithm

1. \( S = \text{empty set}; \ C = V; \)
2. while \( C \) is not empty {
3. \( \text{label each } v \text{ in } C \text{ with a random } r(v); \)
4. \( \text{for all } v \text{ in } C \text{ in parallel} \{
5. \text{if } r(v) < \min( r(\text{neighbors of } v) \) \}
6. \text{move } v \text{ from } C \text{ to } S; \\
7. \text{remove neighbors of } v \text{ from } C; \\
8. \}
9. \}
10. }

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

work \( \approx O(n \log n) \), but span \( \approx O(\log n) \)

Strongly connected components (SCC)

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

Lecture Outline

- Applications
- Designing parallel graph algorithms
- Case studies:
  A. Graph traversals: Breadth-first search
  B. Shortest Paths: Delta-stepping, PHAST, Floyd-Warshall
  C. Ranking: Betweenness centrality
  D. Maximal Independent Sets: Luby’s algorithm
  E. Strongly Connected Components
- Wrap-up: challenges on current systems

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.
**Fleischer/Hendrickson/Pinar algorithm**

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

**Lemma:** $\text{FW}(v) \cap \text{BW}(v)$ is a unique SCC for any $v$. For every other SCC $s$, either
  (a) $s \subseteq \text{FW}(v) \cap \text{BW}(v)$,
  (b) $s \subseteq \text{BW}(v) \setminus \text{FW}(v)$,
  (c) $s \subseteq V \setminus (\text{FW}(v) \cup \text{BW}(v))$.

$\text{FW}(v)$: vertices reachable from vertex $v$.
$\text{BW}(v)$: vertices from which $v$ is reachable.

**Improving FW/BW with parallel BFS**

**Observation:** Real world graphs have giant SCCs

Finding $\text{FW}(\text{pivot})$ and $\text{BW}(\text{pivot})$ can dominate the running time with $\text{span} = O(N)$

**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

---

**Lecture Outline**

- Applications
- Designing parallel graph algorithms
- Case studies:
  A. Graph traversals: Breadth-first search
  B. Shortest Paths: Delta-stepping, PHAST, Floyd-Warshall
  C. Ranking: Betweenness centrality
  D. Maximal Independent Sets: Luby’s algorithm
  E. Strongly Connected Components
- Wrap-up: challenges on current systems

---

**The locality challenge**

“Large memory footprint, low spatial and temporal locality impede performance”

Serial Performance of “approximate betweenness centrality” on a 2.67 GHz Intel Xeon 5560 (12 GB RAM, 8MB L3 cache)

Input: Synthetic R-MAT graphs (# of edges $m = 8n$)

---

**Graph**

- No Last-level Cache (LLC) misses
- $O(m)$ LLC misses
- $\sim 5X$ drop in performance
**The parallel scaling challenge**

"Classical parallel graph algorithms perform poorly on current parallel systems"

- Graph topology assumptions in classical algorithms do not match real-world datasets
- Parallelization strategies at loggerheads with techniques for enhancing memory locality
- Classical "work-efficient" graph algorithms may not fully exploit new architectural features
  - Increasing complexity of memory hierarchy, processor heterogeneity, wide SIMD.
- Tuning implementation to minimize parallel overhead is non-trivial
  - Shared memory: minimizing overhead of locks, barriers.
  - Distributed memory: bounding message buffer sizes, bundling messages, overlapping communication w/ computation.

**Conclusions**

- Best algorithm depends on the input.
- Communication costs (and hence data distribution) is crucial for distributed memory.
- Locality is crucial for in-node performance.
- Graph traversal (BFS) is fundamental building block for more complex algorithms.
- Best parallel algorithm can be significantly different than the best serial algorithm.