Parallel Graph Connectivity

Parallel algorithms: lecture 3

15-853, Spring 2018
Outline

- Connectivity
- Parallel BFS
- Random-mate connectivity
- Low-diameter decomposition
- Work-efficient connectivity
- Are parallel graph algorithms practical?
Graph Connectivity

- $G(V, E)$, $n = \#\text{vertices}$, $m = \#\text{edges}$
- Given an undirected graph $G(V, E)$:

  are $s, t, \in V$ connected?

- Sequential algorithm: run BFS or DFS. $O(n + m)$ time
- Nearly linear-work with union-find
Parallel BFS

- BFS(G(V, E), v):
  - Compute a BFS tree rooted at v
  - I.e. compute a parent for all vertices reachable from v
  - Idea: emulate sequential BFS. Run each step in parallel

- How do we compute the next frontier from current frontier?
- edge_map: primitive for traversal
edge_map

- **Input:**
  - $G(V, E)$
  - $U$ (subset of vertices)
  - update: $vtx \times vtx \rightarrow bool$

- **Output:** $\{v \mid (u, v) \in E, u \in U, update(u, v) = true\}$
edge_map

• Input:
  • G(V, E)
  • U (subset of vertices)
  • update: vtx x vtx -> bool
• Output: \([v\mid(u, v) \in E, u \in U, \text{update}(u, v) = \text{true}]\)

Usually implement update s.t. output is a set
edge_map

edge_map(G, U, update) =
  nghs = array(|U|, <>);
  parfor i in [0, |U|]
    v = U[i];
    out_nghs = G[v].out_nghs;
    update_vtx = lambda x. update(v, x);
    nghs[i] = filter(out_nghs, update_vtx);
  return flatten(nghs);

- Runs in
  \[ O(|U| + \sum_{u \in U} \text{deg}_+(u)) \] work
  \[ O(\log n) \] depth
Parallel BFS

\[
\text{BFS}(G(V, E), v) = \\
\quad n = |V|; \\
\quad \text{frontier} = \text{array}(v); \\
\quad \text{visited} = \text{array}(n, 0); \, \text{visited}[v] = 1; \\
\quad \text{parents} = \text{array}(n, -1); \\
\quad \text{update} = \text{lambda} \, (u, v). \\
\quad \quad \text{if} \, (!\text{visited}[v] \ \&\& \ \text{test_and_set}(&\text{visited}[v])) \\
\quad \quad \quad \text{parents}[v] = u; \\
\quad \quad \quad \text{return} \, \text{true}; \\
\quad \quad \text{return} \, \text{false}; \\
\quad \text{while} \, (|\text{frontier}| > 0): \\
\quad \quad \quad \text{frontier} = \text{edge_map}(G, \text{frontier}, \text{update}); \\
\quad \text{return} \, \text{parents};
\]

\[O(m) \ \text{work} \quad O(\text{diam}(G) \log n) \ \text{depth}\]
Parallel BFS for connectivity

- Real world graphs can have high diameter
  - e.g. road networks and meshes

- Sequential dependencies between components
- BFS-based approach can be very fast on social/web graphs*
  - Process the massive component using BFS
  - Process remaining (small) components using LP

* Multistep connectivity, Slota et al. (2014)
Random Mate

- Idea: form a set of disjoint stars and contract
- \#vertices decrease by a constant fraction each round
- Can show this implies $O(\log n)$ rounds w.h.p.

Source: Blelloch and Maggs, 6886-s18, lecture5.2
Random Mate

flip coins (green = heads)

form stars

contract

Source: Blelloch and Maggs, 6886-s18, lecture5.2
Random Mate Implementation

- Use edgelist format \( E = [(1, 3), (1, 4), (3, 1), (4, 1), \ldots] \)
  - \(2m\) edges
- Each iteration: \(O(m)\) work and \(O(\log n)\) depth
- Each iteration reduces \#active vertices by 1/4 in expectation
  - \(O(\log n)\) rounds w.h.p.
  - \(O(m \log n)\) work, \(O(\log^2 n)\) depth in total (both w.h.p.)

Source: Blelloch and Maggs, 6886-s18, lecture5.2

```python
CC_Random_Mate(L, E) =
    if (|E| = 0) return L;
    1. flip coins for all n vertices
    2. For v where flip(v)=tails, hook to an
       arbitrary heads ngh w, set L(v) = w
    3. \(E' = \text{filter}(E, \lambda (u,v).
       \quad \text{return } L(u) != L(v)); // remove self edges
    4. \(L' = \text{CC_Random_Mate}(L, E')\);
    5. For v where flip(v)=tails, set \(L'(v) = L'(w)
       (v\text{ hooked to } w\text{ in step 2})
    return L';
```
Low-diameter decomposition

- Goal: decompose $V$ into a set of clusters s.t.
  - the number of inter-cluster edges is “small”
  - diameter of each cluster is “small” (\(\sim \log(n)\))
- More formally, given a parameter $\beta$, $0 < \beta < 1$

max diameter is $O(\log n/\beta)$
total cut edges $\leq \beta m$
Low-diameter decomposition

• Even more formally, a \((\beta, d)\)-decomposition, \(0 < \beta < 1\) is a partition of \(V\) into \(V_1, \ldots, V_k\) s.t.
  • The shortest path between \(u, v \in V_i\) using only vertices in \(V_i\) is at most \(d\) (strong diameter)
  • The number of edges \((u, v) \in E, u \in V_i, v \in V_j, i \neq j\) is at most \(\beta m\) (few inter-component edges)
Sequential low-diameter decomposition

An interesting sequential algorithm:*  
- Pick an arbitrary vertex  
- Grow a ball around it using BFS. Stop at the first radius \( r \) s.t. the #boundary edges is < than beta * #internal edges  
- Can show that the radius is at most \( O(\log n/\beta) \)  
- LDD = run substep until all vertices are covered

Can you prove this gives a \((\beta, O(\log n/\beta))\)-decomposition?

Finding each ball is parallelizable, but there are sequential dependencies between different balls…

*see: 15-745 Lecture 12 Notes for details
Parallel low-diameter decomposition

- Miller, Peng and Xu give an algorithm that computes an \((\beta, O(\log n/\beta))\)-decomposition in
- \(O(m + n)\) expected work
- \(O(\log^2 n)\) depth w.h.p.

**Idea:** grow balls in parallel from different vertices

*Mimic sequential ball-growing process to ensure strong diameter.*

**Challenge:** how to guarantee that not too many edges are cut and that the maximum radius is \(O(\log n / \beta)\)?

*Use properties of the exponential distribution to ensure bounds on #cut edges and radius.*

Parallel low-diameter decomposition

\[ \text{LDD}(G(V, E), \beta) = \]
\[ n = |V|; \text{num\_finished} = 0; \]
\[ E = \text{array}(n, \lambda i. \text{Exp}(\beta)); \]
\[ C = \text{array}(n, -1); \]
\[ \text{parfor } i \text{ in } [0:n] \]
\[ C[i] = v \text{ in } V \text{ minimizing } (d(v, i) - E[v]); \]
\[ \text{return } C; \]
Parallel low-diameter decomposition

**Equivalently:** compute start times based on $E$, run multi-BFS

```plaintext
LDD(G(V, E), beta) =
    n = |V|; num_finished = 0;
    E = array(n, lambda i.Exp(beta));
    S = array(n, lambda i.max(E) - E[i]);
    C = array(n, (infty, infty));
    num_processed = 0; round = 1;
    while (num_processed < n)
        F = F ∪ {v in V | S[v] < round, C[v] == infty};
        num_processed += |F|;
        update = lambda (u,v).
        if (C[v].snd == infty)
            writeMin(&C[v].fst, S[u]);
        return false;
        edge_map(G, F, update);
    check = lambda (u,v).
    if (C[v].fst == S[u])
        C[v].snd = u;
        return true;
    return false;
    F = edge_map(G, F, check);
    round++;
    return C;
```

1. **Compute start times**
2. **Add ready centers**
3. **Acquire unvisited nghs**
4. **Set ngh’s cluster id if we won**
Parallel low-diameter decomposition

• Strong diameter is $O(\log n/\beta)$

Note: all vertices will start after $\max[E]$ rounds

• What is the maximum of $n$ R.V.’s independently drawn from $\text{Exp}(\beta)$?
Parallel low-diameter decomposition

- What is the maximum of n R.V.’s independently drawn from $Exp(\beta)$?
- Let $\delta_v \sim Exp(\beta)$

\[
\Pr \left[ \delta_{\text{max}} > \frac{k \log n}{\beta} \right] \leq \sum_{v \in V} \Pr \left[ \delta_v > \frac{k \log n}{\beta} \right] \quad \text{(union bound)}
\]

\[
= n \cdot \exp \left( - \beta \cdot \frac{k \log n}{\beta} \right) \quad \text{(cdf of } Exp(\beta)\text{)}
\]

\[
= \frac{1}{n^{k-1}}
\]

- Maximum diameter is $O(\log n / \beta)$ w.h.p.
Parallel low-diameter decomposition

- Claim: each edge is cut (intercluster) with probability $< \beta$

- Arrival times are random variables:

$$T_i = \delta_{\text{max}} - \delta_i + d(i, c)$$

define $\hat{T}_i = \delta_{\text{max}} - T_i = \delta_i - d(i, c)$

Source: 15-750 Spring 2017 notes
Parallel low-diameter decomposition

\[ T_i = \delta_{\text{max}} - \delta_i + d(i, c) \quad \hat{T}_i = \delta_{\text{max}} - T_i = \delta_i - d(i, c) \]

\[ \arg\min_{v \in V} (\delta_{\text{max}} - \delta_v + d(v, c)) = \arg\max_{v \in V} (\delta_v - d(v, c)) \]

**Note:** Both expressions give the center that captures \( c \)
Parallel low-diameter decomposition

\[ \hat{T}_i = T_i - \delta_{\text{max}} = \delta_i - d(i, c) \]

- \( \hat{T}_i - \hat{T}_j < 1 \) is exactly the event that \((u, v)\) is cut!

**Note:** \( \hat{T}_i - \hat{T}_j \sim \text{Exp}(\beta) \) (memoryless property)

\[ \Pr[\hat{T}_i - \hat{T}_j < 1] = 1 - e^{-\beta} < \beta \]

Source: 15-750 Spring 2017 notes
Parallel low-diameter decomposition

• Back to the algorithm:

```
LDD(G(V, E), beta) =
    n = |V|; num_processed = 0;
    E = array(n, lambda i.Exp(beta));
    S = array(n, lambda i.max(E) - E[i]);
    C = array(n, (infty, infty));
    num_processed = 0; round = 1;
    while (num_processed < n)
        F = F ∪ {v in V | S[v] < round, C[v] =}
        num_processed += |F|;
        update = lambda (u,v).
        if (C[v].snd == infty)
            writeMin(&C[v].fst, S[u]);
        return false;
        edge_map(G, F, update);
        check = lambda (u,v).
        if (C[v].fst == S[u])
            C[v].snd = u;
        return true;
        return false;
    F = edge_map(G, F, check);
    round++;
    return C;
```

- $O(n)$ work, $O(\log n)$ depth
- $O(\log n/\beta)$ rounds w.h.p.
- Each round: $O(\log n)$ depth
- edge_map: $O(m)$ work in total
- L2: Acquire unvisited nghs
- L3: Set ngh’s cluster id if we won
Parallel low-diameter decomposition

- MPX algorithm computes an \((\beta, O(\log n / \beta))\)-decomp in
  - \(O(m + n)\) expected work
  - \(O(\log^2 n)\) depth w.h.p. (can be made \(O(\log n \log^* n)\))

See for more details:
Parallel Graph Decompositions Using Random Shifts
Miller, Peng, Xu (SPAA 2013)

Improved Parallel Algorithms for Spanners and Hopsets
Miller, Peng, Vladu and Xu (SPAA 2015)
Work-efficient connectivity

Compute LDD

Contract and Recurse
Contract and Recurse
L’ = [0, 0, 1, 0, 0]

Update labeling based on L’
L[i] = L’[cluster[i]]
Return L

Work-efficient connectivity
Work-efficient connectivity

Connectivity(G(V, E), beta) =
L = LDD(G, beta);
G'(V',E') = Contract(G, L);
if (|E'| == 0)
  return L
L' = Connectivity(G', beta)
L'' = array(n, lambda v.return L'[L'[v]]);)
return L'';

- Assume contraction in $O(m + n)$ work and $O(\log n)$ depth
- $\beta \cdot m$ edges after each round (expected)
  - $m + \beta \cdot m + \beta^2 \cdot m \ldots = O(m)$ work in expectation
- $O(\log n)$ levels w.h.p. => $O(\log^3 n)$ depth w.h.p.
Work-efficient connectivity

- Connectivity can be solved in parallel in
  - $O(m + n)$ expected work
  - $O(\log^3 n)$ depth
- Depth can be improved to $O(\log n \log \log n \log^* n)$
- (Currently) only theoretically efficient connectivity algorithm that is also practical!

See for more details/experiments:
A Simple and Practical Linear-Work Parallel Algorithm for Connectivity
Shun, Dhulipala and Blelloch (SPAA 2014)
Are parallel graph algorithms practical?

- Huge amount of interest in 80s and 90s
- PRAM algorithms: lots of nice work, but hardware wasn’t ready: never saw the gains promised by theory
Are parallel graph algorithms practical?

- Implement in cilk. Run on shared memory multicores, e.g.
  - Dell PowerEdge R930
    - 72-cores (4 x 2.4GHz 18-core E7-8867 v4 Xeon processors)
    - 1TB of main memory
    - Costs less than a mid-range BMW
Example: k-core

- Ideas like work-efficiency matter!
- E.g. work-efficient vs work-inefficient k-core algorithms
- Run on the two largest publicly available graphs:

| Graph          | $|V|$  | $|E|$ (symmetrized) |
|----------------|-------|--------------------|
| Hyperlink2014  | 1.7B  | 124B               |
| Hyperlink2012  | 3.5B  | 225B               |

$\rho =$ number of peeling steps done by the parallel algorithm

$W = O(|E| + \rho|V|)$  
$D = O(\rho \log |V|)$  

Work-inefficient: $O(|E| + |V|)$ expected work  
Work-efficient: $O(\rho \log |V|)$ depth w.h.p.
Example: k-core

Across all inputs:

- Between 4-41x speedup over sequential peeling
- Speedups are smaller on small graphs with large $\rho$
- 2-9x faster than work-inefficient implementation

|V| = 121M
|E| = 3.6B

Friendster
Example: k-core

- Run on the two largest publicly available graphs:

| Graph         | |V|   | |E| (symmetrized) |
|---------------|-----|-----|
| Hyperlink2014 | 1.7B | 124B |
| Hyperlink2012 | 3.5B | 225B |

- On Hyperlink2012 graph our code takes 193s on 72h cores
- 8515s serially => 44x speedup
- Previous best time: 256-node cluster, each with 32 cores
  - 6 minutes to compute approximate k-cores
- We compute exact k-cores
  - 1.8x faster
  - using 113x fewer cores
Example: connectivity

- Run connectivity on Hyperlink2012 graph
- 38.3s on 72h cores
- 2080s serially => 54x speedup
- Very recently, folks from Yahoo (Oath research) presented a new connectivity algorithm that runs in $O(\log n)$ rounds on BSP model
- Their algorithm runs in 341s using:
  - 1000 nodes, 24000 cores and 128Tb of memory

Our algorithm is 8x faster using 128x less memory and 333x fewer cores

- To be fair, their code runs on a graph with 272B vertices and 5.9T edges; out of reach of shared-memory for now…

| Graph       | $|V|$  | $|E|$  |
|-------------|------|-------|
| Hyperlink2012 | 3.5B | 225B  |
Theoretically efficient parallel algorithms

<table>
<thead>
<tr>
<th>Problem</th>
<th>Model</th>
<th>Work</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-First Search</td>
<td>TS</td>
<td>$O(m)$</td>
<td>$O(\text{diam}(G) \log n)$</td>
</tr>
<tr>
<td>Integral-Weight SSSP (weighted BFS)</td>
<td>PW</td>
<td>$O(m)$ expected</td>
<td>$O(\text{diam}(G) \log n)$ w.h.p.*</td>
</tr>
<tr>
<td>General-Weight SSSP (Bellman-Ford)</td>
<td>PW</td>
<td>$O(\text{diam}(G)m)$</td>
<td>$O(\text{diam}(G) \log n)$ w.h.p.</td>
</tr>
<tr>
<td>Single-Source Betweenness Centrality (BC)</td>
<td>FA</td>
<td>$O(m)$</td>
<td>$O(\text{diam}(G) \log n)$</td>
</tr>
<tr>
<td>Low-Diameter Decomposition</td>
<td>TS</td>
<td>$O(m)$ expected</td>
<td>$O(\log^2 n)$ w.h.p.</td>
</tr>
<tr>
<td>Connectivity</td>
<td>TS</td>
<td>$O(m)$ expected</td>
<td>$O(\log^3 n)$ w.h.p.</td>
</tr>
<tr>
<td>Biconnectivity</td>
<td>FA</td>
<td>$O(m)$ expected</td>
<td>$O(\max(\text{diam}(G) \log n, \log^3 n))$ w.h.p.</td>
</tr>
<tr>
<td>Strongly Connected Components</td>
<td>PW</td>
<td>$O(m \log n)$ expected</td>
<td>$O(\text{diam}(G) \log n)$ w.h.p.</td>
</tr>
<tr>
<td>Minimum Spanning Forest</td>
<td>PW</td>
<td>$O(m \log n)$</td>
<td>$O(\log^2 n)$ w.h.p.</td>
</tr>
<tr>
<td>Maximal Independent Set</td>
<td>FA</td>
<td>$O(m)$ expected</td>
<td>$O(\log^2 n)$ w.h.p.</td>
</tr>
<tr>
<td>Maximal Matching</td>
<td>PW</td>
<td>$O(m)$ expected</td>
<td>$O(\log^3 m / \log \log m)$ w.h.p.</td>
</tr>
<tr>
<td>Graph Coloring</td>
<td>FA</td>
<td>$O(m + n)$</td>
<td>$O(\log n + L \log \Delta)$</td>
</tr>
<tr>
<td>$k$-core</td>
<td>FA</td>
<td>$O(m + n)$ expected</td>
<td>$O(\rho \log n)$ w.h.p.</td>
</tr>
<tr>
<td>Approximate Set Cover</td>
<td>PW</td>
<td>$O(m)$ expected</td>
<td>$O(\log^3 n)$ w.h.p.</td>
</tr>
<tr>
<td>Triangle Counting</td>
<td>-</td>
<td>$O(m^{3/2})$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

Based on 30 years of research on parallel algorithms
Theoretically efficient parallel algorithms

<table>
<thead>
<tr>
<th>Problem</th>
<th>(1)</th>
<th>(72h)</th>
<th>(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-First Search</td>
<td>631</td>
<td>12</td>
<td>52</td>
</tr>
<tr>
<td>Integral-Weight SSSP (weighted BFS)</td>
<td>4700</td>
<td>58.1</td>
<td>80</td>
</tr>
<tr>
<td>General-Weight SSSP (Bellman-Ford)</td>
<td>3180</td>
<td>53</td>
<td>60</td>
</tr>
<tr>
<td>Single-Source Betweenness Centrality (BC)</td>
<td>3170</td>
<td>40</td>
<td>79</td>
</tr>
<tr>
<td>Low-Diameter Decomposition</td>
<td>464</td>
<td>12</td>
<td>38</td>
</tr>
<tr>
<td>Connectivity</td>
<td>2080</td>
<td>38.3</td>
<td>54</td>
</tr>
<tr>
<td>Biconnectivity</td>
<td>—</td>
<td>201</td>
<td>—</td>
</tr>
<tr>
<td>Strongly Connected Components</td>
<td>7720</td>
<td>182</td>
<td>42</td>
</tr>
<tr>
<td>Minimum Spanning Forest</td>
<td>—</td>
<td>228</td>
<td>—</td>
</tr>
<tr>
<td>Maximal Independent Set</td>
<td>2210</td>
<td>34</td>
<td>65</td>
</tr>
<tr>
<td>Maximal Matching</td>
<td>11000</td>
<td>140</td>
<td>78</td>
</tr>
<tr>
<td>Graph Coloring</td>
<td>12200</td>
<td>174</td>
<td>70</td>
</tr>
<tr>
<td>$k$-core</td>
<td>8515</td>
<td>193</td>
<td>44</td>
</tr>
<tr>
<td>Approximate Set Cover</td>
<td>3720</td>
<td>104</td>
<td>35</td>
</tr>
<tr>
<td>Triangle Counting</td>
<td>—</td>
<td>1470</td>
<td>—</td>
</tr>
</tbody>
</table>

Running times in seconds on Hyperlink2012

All implementations are theoretically efficient and scalable!