ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms

Laxman Dhulipala
MIT (Postdoc)
https://ldhulipala.github.io/

Based on joint work with

Changwan Hong and Julian Shun (VLDB’21)
Connected Components

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

Assign vertices labels $L(v)$ s.t. $L(u) = L(v)$ iff there is a path from $u$ to $v$ in $G$
Applications of Connected Components

Clustering
- DBSCAN
- k-Core Hierarchy
- Affinity Clustering
- ...

Other Connectivity Problems
- Spanning Forest
- Biconnectivity
- Approximate Minimum Spanning Forest

Sequential Connectivity Algorithms

❖ Run Breadth-First Search or Depth-First Search:

\[
\text{labels} = [-1, \ldots, -1] \quad \# \text{initialized to a null value}
\]

\[
\text{for } i \text{ in } [0, \mid V\mid):
\]

\[
\text{if } \text{labels}[i] == -1:
\]

\[
\text{BFS}(G, i) \quad \# \text{assign label } i \text{ to visited vertices}
\]

\[
\text{return } \text{labels}
\]

❖ Algorithms run in \(O(n + m)\) time
Parallel BFS for Connectivity

```python
labels = [-1, ..., -1]  # initialized to a null value
for i in [0, |V|):
    if labels[i] == -1:
        ParallelBFS(G, i)  # assign label i to visited vertices
return labels
```

- Real-world graphs can have high diameter (e.g. road networks / meshes)
- Graph could also have many components

$O(m + n)$ work, $O(n)$ depth

Are there low-work, polylog(n) depth connectivity algorithms?
Parallel Connectivity Algorithms

Random-Mate Algorithms

1. Flip coins (green = heads)
2. Form stars

Work-Efficient Algorithms

1. Compute LDD
2. Contract and Recurse

Concurrent Union-Find

1. Contract

Dozens of papers on different approaches to parallel connectivity written over the past few decades!
Goal:
Explore the space of optimizations for parallel (shared-memory) graph connectivity and find the fastest implementation of parallel connectivity
Express several hundred different multicore implementations of connectivity, spanning forest, and incremental connectivity (most of which are new)

Obtain 2.3x average speedup over the fastest existing static multicore connectivity algorithms
Motivation: Direction-Optimizing BFS

Direction-optimization skips over incoming edges in dense traversals once the vertex has already been visited.

Using direction-opt: 0.081425
Without direction-opt: 0.715358
(on the Twitter-Sym graph, 72 cores)

Two-Phase Execution is inspired by direction optimization. It accelerates parallel connectivity algorithms by “skipping” the traversal of certain edges.
Two-Phase Execution

**Sampling Phase**

Compute a partial connectivity labeling while processing edges

Identify the largest component $L_{\text{max}}$ in the partial labeling.

**Finish Phase**

Process all vertices not in $L_{\text{max}}$ using the given finish algorithm to compute a correct connectivity labeling.
def Connectivity(G, sample_opt, finish_opt):
    # Initialize sampling and finish algorithms
    sampling = GetSamplingAlgorithm(sample_opt)
    finish = GetFinishAlgorithm(finish_opt)

    # Initialize labels and perform sampling to
    # obtain a partial connectivity labeling.
    labels = {i -> i | i in [0, |V|]}
    labels = sampling.SampleComponents(G, labels)

    # Identify the largest (most frequent
    # component), L_max
    L_max = IdentifyFrequent(labels)

    # Compute a connectivity labeling from the partial
    # labeling using the finish algorithm.
    labels = finish.FinishComponents(G, labels, L_max)
    return labels
Two-Phase Execution: Example

Input Graph

(i)
Two-Phase Execution: Example

(i) Input Graph

(ii) Sampled Labels

$L_{\text{max}}$
Two-Phase Execution: Example

Finish Step on $v \notin L_{\max}$
Two-Phase Execution: Example

Input Graph

Sampled Labels

Finish Step on $v \notin L_{\text{max}}$

Output Labeling

(i) 

(ii) 

(iii) 

(iv)
Properties of Sampling Methods

Connectivity Labeling

\[ C(u) = C(v) \text{ iff } u \text{ and } v \text{ are in the same component} \]

Partial Connectivity Labeling

\[ C(u) = C(v) \text{ implies that } u \text{ and } v \text{ are in the same component} \]
Properties of Sampling Methods

Let

\[ C = \text{SamplingMethod}(G) \]
\[ C' = \text{Connectivity}(G[C]) \]

A sampling method is **correct** if:

1. \( \forall v \in V, \) either \( C(v) = v \) or \( C(v) = r \) and \( C(r) = r \)
2. \( C'' = \{ C'(C(v)) \mid v \in V \} \) is a connectivity labeling

\( G[C] \) formed by merging all vertices \( v \) with the same label into a single vertex, and only preserving \((u,v)\) edges s.t. \( C(u) \) and \( C(v) \) are distinct (removing duplicate edges)
Properties of Finish Methods

Let

\[ C = \{i \rightarrow i \mid \forall i \in V\} \]

A connectivity algorithm is **monotone** if the algorithm updates the labels s.t. the updated labeling can be represented as the union of two trees in the previous labeling.

I.e., once two vertices are in the same tree, they will always remain in the same tree.
Properties of Finish Methods

A connectivity algorithm operating on a labeling C is **linearizable monotone** if

1. Its operations are linearizable.
2. Every operation in the linearization order preserves monotonicity.

Composing a correct sampling method with a linearizable monotone finish algorithm yields a connectivity labeling.

Next:
Introduce several sampling and finish methods
k-Out Sampling

```python
def kOutSample(G(V,E), labels, k=2):
    edges = {first edge from each vertex} U {sample k-1 edges uniformly at random from each vertex}
    UnionFind(edges, labels)
    Fully compress the components array, in parallel
    return labels
```

Original scheme from Afforest connectivity algorithm (Sutton et al., 2018):

1. Select the first two edges incident to each vertex (in gen. first $k$)

Can yield poor results depending on how vertices in the graph are ordered.
**k-Out Sampling**

Theoretical motivation from Holm et al. (2019):

Suppose each vertex of an arbitrary simple graph on $n$ vertices chooses $k$ random incident edges.

Then the expected number of edges in the original graph connecting different connected components in the sampled subgraph is $O(n/k)$.

Implies that by processing $O(nk)$ edges, only $O(n/k)$ edges need to be examined in the finish stage to compute a correct labeling.

```python
def kOutSample(G(V,E), labels, k=2):
    edges = {first edge from each vertex} U {sample k-1 edges uniformly at random from each vertex}
    UnionFind(edges, labels)
    Fully compress the components array, in parallel
    return labels```
LDD Sampling

```python
def LDDSample(G(V,E), labels, beta=0.2):
    labels = LDD(G, beta)
    return labels
```

Recall theoretical guarantees of LDD:

1. Strong diameter of each cluster is $O(\log n/\beta)$
2. Number of intercluster edges is $O(\beta m)$ in expectation

In practice, after one application of LDD, the resulting clustering often contains a single massive cluster.
def BFSSample(G(V,E), labels, c=5):
    for i in [0, c):
        # Run direction-optimizing BFS from random source.
        s = RandVertex()
        labels = LabelSpreadingBFS(G, s)

        # Check if BFS covered a significant fraction of the
        # vertices.
        freq = IdentifyFrequent(labels)
        if (freq makes up more than 10% of the labels) then:
            return labels

        # otherwise return identity labeling.
        return {i -> i | i in [0, |V|]}

Practical motivation: many real-world graphs contain a single massive
(low-diameter) component which we will find with constant probability.
How do sampling strategies perform in practice?

![Bar chart showing edges in the largest connected component (LCC) for different sampling strategies.](chart.png)
Min-Based and Root-Based Algorithms

A **min-based** algorithm represents connectivity labelings as a collection of disjoint sets (similar to union-find), where all elements in a set are associated with the same label.

A min-based algorithm only updates the label of an element to a new label if the new label is smaller than the previous label.

A **root-based** algorithm is a special type of min-based algorithm which only links sets together by adding a link from the root of one tree to a node in another tree.
Asynchronous Union-Find: Union

```python
def Union(u, v, P):
    p_u = Find(u, P)
    p_v = Find(v, P)
    while (p_u != p_v):
        if (p_u == P[p_u] and CAS(&P[p_u], p_u, p_v)):
            return
        p_u = Find(u, P)
        p_v = Find(v, P)
```

WLOG let $p_u > p_v$

(consistently link high to low or vice versa to prevent cycles)
Asynchronous Union-Find: Find and FindCompress

```python
def FindCompress(u, P):
    # Find the root of u's tree, r. If u
    # is the root, quit.
    r = u
    if (P[r] == r):
        return r
    while (r != P[r]):
        r = P[r]
    # Make the parent of all vertices on
    # the u to r path r (or a smaller id).
    j = P[u]
    while (j > r):
        P[u] = r
        u = j
    return r

def FindNaive(u, P):
    v = u
    while (v != P[v]):
        v = P[v]
    return v
```

FindCompress(u, P)
Asynchronous Union-Find: Splitting and Halving

```python
def FindAtomicSplit(u, P):
    v = P[u]  # parent(u)
    w = P[v]  # grandparent(u)
    while (v != w):
        CAS(&P[u], v, w)
        u = v
    return v

def FindAtomicHalve(u, P):
    v = P[u]  # parent(u)
    w = P[v]  # grandparent(u)
    while (v != w):
        CAS(&P[u], v, w)
        u = P[u]
    return v
```
def Union(u, v, P):
    r_u = u, r_v = v
    while (P[r_u] != P[r_v]):
        # WLOG let P[r_u] > P[r_v].
        if (r_u == P[r_u] and
            CAS(&P[r_u], r_u, P[r_v])):
            # Success: linked the two trees.
            if (CompressOpt != FindNaive):
                Compress(u, P)
                Compress(v, P)
            return
        else:
            # Otherwise shorten path using splice.
            r_u = Splice(r_u, r_v, P)
Concurrent Rem's Algorithm: Splice Options

```python
def HalveAtomicOne(u, x, P):
    v = P[u]  # parent
    w = P[v]  # grandparent
    if (u != w):
        CAS(&P[u], v, w)
    return w

def SplitAtomicOne(u, x, P):
    v = P[u]  # parent
    w = P[v]  # grandparent
    if (u != w):
        CAS(&P[u], v, w)
    return v

def SpliceAtomic(u, x, P):
    p_u = P[u]
    # Try to make u's parent x's parent which
    # could be a node in the other tree.
    CAS(&P[u], p_u, P[x])
    return p_u
```
Concurrent Rem’s Algorithm: Splice Options

```python
def Union(u, v, P):
    r_u = u, r_v = v
    while (P[r_u] != P[r_v]):
        # WLOG let P[r_u] > P[r_v].
        if (r_u == P[r_u] and
            CAS(&P[r_u], r_u, P[r_v])):
            # Success: linked the two trees.
            if (CompressOpt != FindNaive):
                Compress(u, P)
                Compress(v, P)
            return
        else:
            # Otherwise shorten path using splice.
            r_u = Splice(r_u, r_v, P)

def SpliceAtomic(u, x, P):
    p_u = P[u]
    # Try to make u's parent x's parent which
    # could be a node in the other tree.
    CAS(&P[u], p_u, P[x])
    return p_u
```

![Diagram of Rem's Algorithm](image)
Other Min-Based Algorithms

Union-Find Algorithms

- Jayanti-Tarjan (two-try split)
- UF-Early
- UF-Hooks
- UF-Rem-Lock

Liu-Tarjan Algorithms

- Family of min-based algorithms based on shortcutting

Shiloach-Vishkin

Label Propagation
Experiments

Dell PowerEdge R930

❖ 72-cores, 2-way hyper-threaded*
❖ 1TB of main memory
❖ Cost: about 20k USD

Graph Data

❖ Run on a collection of large real-world graphs, including largest publicly available graph (HL12)

<table>
<thead>
<tr>
<th>Graph</th>
<th>n</th>
<th>m</th>
<th>Diam.</th>
<th>Num C.</th>
<th>Largest C.</th>
<th>LT-DC (s)</th>
<th>LT (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RO</td>
<td>23.9M</td>
<td>57.7M</td>
<td>6,809</td>
<td>1</td>
<td>23.9M</td>
<td>0.108</td>
<td>0.241</td>
</tr>
<tr>
<td>LJ</td>
<td>4.8M</td>
<td>85.7M</td>
<td>16</td>
<td>1,876</td>
<td>4.8M</td>
<td>0.101</td>
<td>0.226</td>
</tr>
<tr>
<td>CO</td>
<td>3.1M</td>
<td>234.4M</td>
<td>9</td>
<td>1</td>
<td>3.1M</td>
<td>0.094</td>
<td>0.520</td>
</tr>
<tr>
<td>TW</td>
<td>41.7M</td>
<td>2.4B</td>
<td>23*</td>
<td>1</td>
<td>41.7M</td>
<td>0.115</td>
<td>2.80</td>
</tr>
<tr>
<td>FR</td>
<td>65.6M</td>
<td>3.6B</td>
<td>32</td>
<td>1</td>
<td>65.6M</td>
<td>0.182</td>
<td>6.07</td>
</tr>
<tr>
<td>CW</td>
<td>978.4M</td>
<td>74.7B</td>
<td>132*</td>
<td>23.7M</td>
<td>950.5M</td>
<td>0.534</td>
<td>54.2</td>
</tr>
<tr>
<td>HL14</td>
<td>1.7B</td>
<td>124.1B</td>
<td>207*</td>
<td>129M</td>
<td>1.57B</td>
<td>1.02</td>
<td>101.3</td>
</tr>
<tr>
<td>HL12</td>
<td>3.6B</td>
<td>225.8B</td>
<td>331*</td>
<td>144M</td>
<td>3.35B</td>
<td>1.64</td>
<td>192.5</td>
</tr>
</tbody>
</table>

* (4 x 2.4GHz 18-core E7-8867 v4 Xeon processors)
UF-Rem-CAS with splice/split/halve and no additional compression reliably performs the best across all inputs.
Comparison on WebDataCommons Hyperlink2012

- Fastest ConnectIt algorithm for HL2012 is $3.65 - 41.5x$ faster than existing distributed memory results while using orders of magnitude fewer resources

- Running time without sampling on HL2012 of our fastest algorithm is 13.9 seconds (1.69x speedup using k-Out Sampling)

<table>
<thead>
<tr>
<th>System</th>
<th>Graph</th>
<th>Mem. (TB)</th>
<th>Threads</th>
<th>Nodes</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mosaic [72]</td>
<td>Hyperlink2014</td>
<td>0.768</td>
<td>1000</td>
<td>1</td>
<td>708</td>
</tr>
<tr>
<td>FlashGraph [114]</td>
<td>Hyperlink2012</td>
<td>.512</td>
<td>64</td>
<td>1</td>
<td>461</td>
</tr>
<tr>
<td>GBBS [32]</td>
<td>Hyperlink2012</td>
<td>1</td>
<td>144</td>
<td>1</td>
<td>25.8</td>
</tr>
<tr>
<td>GBBS (NVRAM) [34]</td>
<td>Hyperlink2012</td>
<td>0.376</td>
<td>96</td>
<td>1</td>
<td>36.2</td>
</tr>
<tr>
<td>Galois (NVRAM) [43]</td>
<td>Hyperlink2012</td>
<td>0.376</td>
<td>96</td>
<td>1</td>
<td>76.0</td>
</tr>
<tr>
<td>Slota et al. [99]</td>
<td>Hyperlink2012</td>
<td>16.3</td>
<td>8192</td>
<td>256</td>
<td>63</td>
</tr>
<tr>
<td>Stergiou et al. [101]</td>
<td>Hyperlink2012</td>
<td>128</td>
<td>24000</td>
<td>1000</td>
<td>341</td>
</tr>
<tr>
<td>Gluon [30]</td>
<td>Hyperlink2012</td>
<td>24</td>
<td>69632</td>
<td>256</td>
<td>75.3</td>
</tr>
<tr>
<td>Zhang et al. [113]</td>
<td>Hyperlink2012</td>
<td>≥ 256</td>
<td>262,000</td>
<td>4096</td>
<td>30</td>
</tr>
<tr>
<td><strong>ConnectIt</strong></td>
<td>Hyperlink2014</td>
<td>1</td>
<td>144</td>
<td>1</td>
<td>2.83</td>
</tr>
<tr>
<td></td>
<td>Hyperlink2012</td>
<td>1</td>
<td>144</td>
<td>1</td>
<td>8.20</td>
</tr>
</tbody>
</table>

Table 1: System configurations, including memory (terabytes), num. hyper-threads and nodes, and running times (seconds) of connectivity results on the Hyperlink graphs. The last rows show the fastest ConnectIt times. The fastest time per graph is shown in green.
Comparing No-Sampling with Sampling

<table>
<thead>
<tr>
<th>Grp.</th>
<th>Algorithm</th>
<th>RO</th>
<th>LJ</th>
<th>CO</th>
<th>TW</th>
<th>FR</th>
<th>CW</th>
<th>HL14</th>
<th>HL12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No-Sampling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UF-Early</td>
<td>3.61e-2</td>
<td>3.84e-2</td>
<td>8.63e-2</td>
<td>2.52</td>
<td>1.50</td>
<td>59.8</td>
<td>17.0</td>
<td>32.9</td>
<td></td>
</tr>
<tr>
<td>UF-Hooks</td>
<td>3.37e-2</td>
<td>1.75e-2</td>
<td>2.69e-2</td>
<td>0.390</td>
<td>1.17</td>
<td>6.05</td>
<td>9.37</td>
<td>20.0</td>
<td></td>
</tr>
<tr>
<td>UF-Async</td>
<td>4.02e-2</td>
<td>2.03e-2</td>
<td>3.12e-2</td>
<td>0.426</td>
<td>1.21</td>
<td>7.92</td>
<td>12.2</td>
<td>25.5</td>
<td></td>
</tr>
<tr>
<td>UF-Rem-CAS</td>
<td>2.80e-2</td>
<td>1.27e-2</td>
<td>1.91e-2</td>
<td>0.316</td>
<td>0.902</td>
<td>4.04</td>
<td>6.04</td>
<td>13.9</td>
<td></td>
</tr>
<tr>
<td>UF-Rem-Lock</td>
<td>5.07e-2</td>
<td>1.95e-2</td>
<td>2.84e-2</td>
<td>0.437</td>
<td>1.23</td>
<td>5.64</td>
<td>9.20</td>
<td>19.3</td>
<td></td>
</tr>
<tr>
<td>UF-JTB</td>
<td>6.90e-2</td>
<td>4.49e-2</td>
<td>8.48e-2</td>
<td>0.965</td>
<td>2.76</td>
<td>22.5</td>
<td>36.4</td>
<td>72.1</td>
<td></td>
</tr>
<tr>
<td>Liu-Tarjan</td>
<td>7.40e-2</td>
<td>5.18e-2</td>
<td>6.46e-2</td>
<td>2.78</td>
<td>6.60</td>
<td>30.1</td>
<td>67.1</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>SV</td>
<td>0.138</td>
<td>4.34e-2</td>
<td>5.70e-2</td>
<td>1.65</td>
<td>5.38</td>
<td>21.2</td>
<td>38.5</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>Label-Prop</td>
<td>13.4</td>
<td>4.66e-2</td>
<td>6.37e-2</td>
<td>1.24</td>
<td>4.37</td>
<td>13.4</td>
<td>20.7</td>
<td>46.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>k=0 Sampling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UF-Early</td>
<td>3.25e-2</td>
<td>9.00e-3</td>
<td>8.61e-3</td>
<td>0.117</td>
<td>0.227</td>
<td>2.28</td>
<td>4.77</td>
<td>8.94</td>
<td></td>
</tr>
<tr>
<td>UF-Hooks</td>
<td>3.62e-2</td>
<td>9.18e-3</td>
<td>9.16e-3</td>
<td>0.121</td>
<td>0.230</td>
<td>2.22</td>
<td>3.63</td>
<td>8.51</td>
<td></td>
</tr>
<tr>
<td>UF-Async</td>
<td>3.33e-2</td>
<td>8.97e-3</td>
<td>8.56e-3</td>
<td>0.117</td>
<td>0.228</td>
<td>2.21</td>
<td>3.60</td>
<td>8.49</td>
<td></td>
</tr>
<tr>
<td>UF-Rem-CAS</td>
<td>3.43e-2</td>
<td>8.96e-3</td>
<td>8.62e-3</td>
<td>0.117</td>
<td>0.227</td>
<td>2.15</td>
<td>3.51</td>
<td>8.20</td>
<td></td>
</tr>
<tr>
<td>UF-Rem-Lock</td>
<td>4.45e-2</td>
<td>1.13e-2</td>
<td>1.01e-2</td>
<td>0.138</td>
<td>0.344</td>
<td>2.63</td>
<td>4.33</td>
<td>9.91</td>
<td></td>
</tr>
<tr>
<td>UF-JTB</td>
<td>3.89e-2</td>
<td>9.77e-3</td>
<td>8.80e-3</td>
<td>0.125</td>
<td>0.237</td>
<td>2.43</td>
<td>4.05</td>
<td>9.58</td>
<td></td>
</tr>
<tr>
<td>Liu-Tarjan</td>
<td>6.34e-2</td>
<td>9.90e-3</td>
<td>9.18e-3</td>
<td>0.129</td>
<td>0.374</td>
<td>2.61</td>
<td>6.74</td>
<td>11.5</td>
<td></td>
</tr>
<tr>
<td>SV</td>
<td>5.72e-2</td>
<td>9.72e-3</td>
<td>8.78e-2</td>
<td>0.124</td>
<td>0.237</td>
<td>2.70</td>
<td>5.03</td>
<td>12.5</td>
<td></td>
</tr>
<tr>
<td>Label-Prop</td>
<td>12.6</td>
<td>1.02e-2</td>
<td>9.63e-3</td>
<td>0.124</td>
<td>0.375</td>
<td>2.44</td>
<td>4.75</td>
<td>9.68</td>
<td></td>
</tr>
</tbody>
</table>

- Union-Find algorithms essentially always the fastest
- Sampling does not help much on very sparse graphs (avg degree in RO = 2.41)
- UF-Rem-CAS is consistently the fastest finish algorithm across all settings
- No significant difference between using SplitAtomicOne / HalveAtomicOne / SpliceAtomic
Algorithm Recommendations

- Tuning recommendations based on studying sampling performance on both real-world and synthetic networks (see paper)
Summary: ConnectIt

ConnectIt: framework for static and incremental parallel graph connectivity

- Simple to generate new combinations of sampling and finish algorithms
- Our fastest implementations of connectivity significantly outperform state-of-the-art parallel solutions
- Solutions for connectivity extend to parallel spanning forest and incremental connectivity

Code available as part of GBBS:

github.com/paralg/gbbs