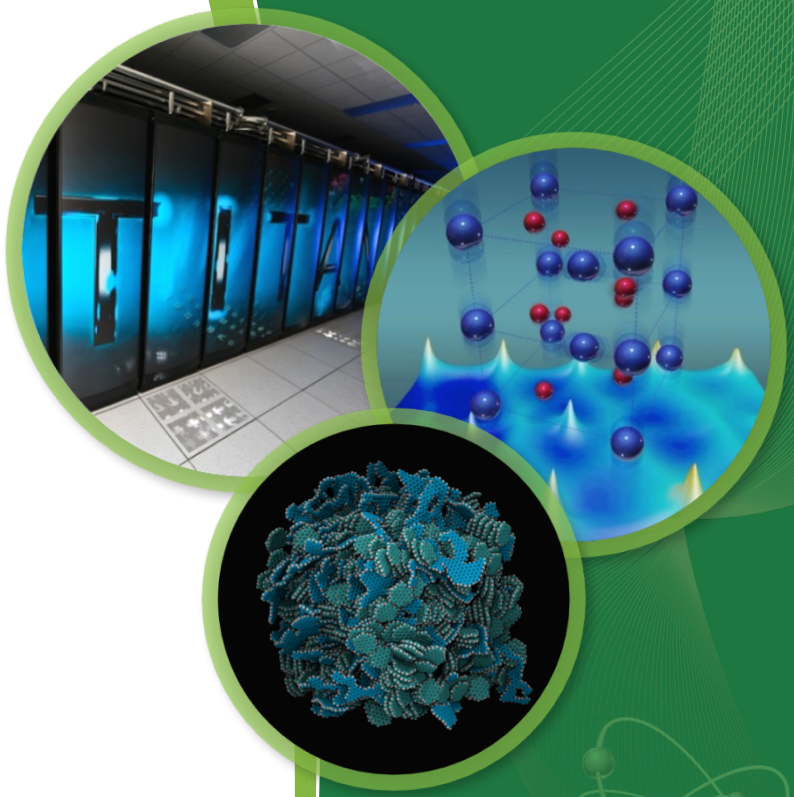
Parallelizing Single Source Shortest Path with OpenSHMEM

Ferrol Aderholdt

Jeffrey A. Graves

Manjunath Gorentla Venkata



ORNL is managed by UT-Battelle
for the US Department of Energy

OAK RIDGE National Laboratory

Motivation to Revisit Parallel Single Source Shortest Path

- The convergence of BigCompute and BigData
  - HPC-centric systems will be used for both compute and data intensive applications
  - High-Performance parallelization of Data Analytics is of growing importance
- Explore PGAS for single source shortest path (SSSP)
  - OpenSHMEM for irregular, graph-based algorithms
    - Graph datasets and algorithms are irregular and have poor locality
    - Programming model suitable for graph-based workloads
      - Lightweight semantics: fast communication/AMOs
      - Graph workloads are irregular and have poor locality

Single Source Shortest Path

- SSSP: the discovery of paths from an origin vertex to all other vertices
  - Can be used by many different graph-based algorithms
    - Betweeness centrality, all-pairs shortest path
- Can be categorized into two types
  - Label-setting (Dijkstra's)
    - Calculate the correct minimum distance value before moving on
    - More difficult to parallelize due to dependencies
  - Label-correcting (Bellman-Ford)
    - Continually update distance values until global convergence
    - More easily parallelizable due to few dependencies

Dijkstra's Algorithm (Serial)

- Undirected/directed graphs with non-negative weights
- Maintained data structures: distance and path arrays; priority queue
- O(|E| + |V| log |V|) when using an efficient heap

```
Require: G, a weighted graph
Require: src, a source vertex
 1: function DIJKSTRA(G, src)
 2:     for v in V(G) do
 3:         distance[v] ← ∞
 4:         path[v] ← null
 5:         Queue.push(v, dst)
 6:     end for
 7:     Queue.decrease(src, 0)
 8:     while Queue is not empty do
 9:         v ← Queue.pop()
10:         for u in Γ(v) do
11:             RELAX(v, u, edge(v, u))
12:         end for
13:     end while
14:     return distance, path
15: end function
```

```
1: function RELAX(v, u, e)
2:     tmp ← distance[v] + weight(e)
3:     if distance[u] > tmp then
4:         distance[u] ← tmp
5:         path[u] ← v
6:         Queue.decrease(u, distance[u])
7:     end if
8: end function
```

Dijkstra's Algorithm (Serial)

- Undirected/directed graphs with non-negative weights
- Maintained data structures: distance and path arrays; priority queue
- O(|E| + |V| log |V|) when using an efficient heap

Initialize data structures

```
Require: G, a weighted graph
Require: src, a source vertex
 1: function DIJKSTRA(G, src)
 2:     for v in V(G) do
 3:         distance[v] ← ∞
 4:         path[v] ← null
 5:         Queue.push(v, dst)
 6:     end for
 7:     Queue.decrease(src, 0)
 8:     while Queue is not empty do
 9:         v ← Queue.pop()
10:         for u in Γ(v) do
11:             RELAX(v, u, edge(v, u))
12:         end for
13:     end while
14:     return distance, path
15: end function
```

```
1: function RELAX(v, u, e)
2:     tmp ← distance[v] + weight(e)
3:     if distance[u] > tmp then
4:         distance[u] ← tmp
5:         path[u] ← v
6:         Queue.decrease(u, distance[u])
7:     end if
8: end function
```

Dijkstra's Algorithm (Serial)

- Undirected/directed graphs with non-negative weights
- Maintained data structures: distance and path arrays; priority queue
- O(|E| + |V| log |V|) when using an efficient heap

```
Require: G, a weighted graph
Require: src, a source vertex
 1: function DIJKSTRA(G, src)
 2:     for v in V(G) do
 3:         distance[v] ← ∞
 4:         path[v] ← null
 5:         Queue.push(v, dst)
 6:     end for
 7:     Queue.decrease(src, 0)
 8:     while Queue is not empty do
 9:         v ← Queue.pop()
10:         for u in Γ(v) do
11:             RELAX(v, u, edge(v, u))
12:         end for
13:     end while
14:     return distance, path
15: end function
```

Loop over Priority Queue until all vertices are visited

```
1: function RELAX(v, u, e)
2:     tmp ← distance[v] + weight(e)
3:     if distance[u] > tmp then
4:         distance[u] ← tmp
5:         path[u] ← v
6:         Queue.decrease(u, distance[u])
7:     end if
8: end function
```

Dijkstra's Algorithm (Serial)

- Undirected/directed graphs with non-negative weights
- Maintained data structures: distance and path arrays; priority queue
- O(|E| + |V| log |V|) when using an efficient heap

```
Require: G, a weighted graph
Require: src, a source vertex
 1: function DIJKSTRA(G, src)
 2:     for v in V(G) do
 3:         distance[v] ← ∞
 4:         path[v] ← null
 5:         Queue.push(v, dst)
 6:     end for
 7:     Queue.decrease(src, 0)
 8:     while Queue is not empty do
 9:         v ← Queue.pop()
10:         for u in Γ(v) do
11:             RELAX(v, u, edge(v, u))
12:         end for
13:     end while
14:     return distance, path
15: end function
```

```
1: function RELAX(v, u, e)
2:     tmp ← distance[v] + weight(e)
3:     if distance[u] > tmp then
4:         distance[u] ← tmp
5:         path[u] ← v
6:         Queue.decrease(u, distance[u])
7:     end if
8: end function
```

Return result

Parallelizing Dijkstra's Algorithm

- Challenges
  - Dependency on guaranteeing correct value when visiting a vertex
    - Requires synchronization at each step
  - Requires parallel/distributed priority queue
    - Can be difficult to make efficient
- Approach
  - Naïve graph partitioning with uniform partitioning
  - Portion of priority queue is symmetric
    - Enables a priori determination of read/write operations to symmetric memory

Parallelizing Dijkstra's Algorithm (2)

Require: G, a weighted graph
Require: src, a source vertex

1: **function** ParallelDijkstra(G, src, v_{start}, v_{end})
2: **for** $v_{start} \leq v \leq v_{end}$ in $V(G)$ **do**
3: $distance[v] \leftarrow \infty$
4: $path[v] \leftarrow null$
5: $Queue.push(v, dst)$
6: **end for**
7: $Queue.decrease(src, 0)$
8: **while** $Queue$ is not empty **do**
9: $v \leftarrow$ FindMin($rank$)
10: **for** u in $\Gamma(v)$ **do**
11: Relax($v, u,$ edge(v, u))
12: **end for**
13: **end while**
14: **return** $distance$, $path$
15: **end function**

1: **function** Relax(v, u, e)
2: $tmp \leftarrow distance[v] +$ weight(e)
3: **if** $distance[u] > tmp$ **then**
4: $distance[u] \leftarrow tmp$
5: $path[u] \leftarrow v$
6: $Queue.decrease(u, distance[u])$
7: **end if**
8: **end function**

1: **function** FindMin($rank$)
2: $p, v \leftarrow Queue.peek()$
3: put(shared$[rank]$, (p, v), Master)
4: **if** $rank =$ MASTER **then**
5: $p, v \leftarrow$min$(shared)$
6: **end if**
7: bcast(p, v)
8: **if** $v = Queue.peek()$ **then**
9: $Queue.pop()$
10: **end if**
11: **return** v
12: **end function**

Parallelizing Dijkstra's Algorithm (2)

Require: G, a weighted graph
Require: src, a source vertex

1: **function** ParallelDijkstra(G, src, v_{start}, v_{end})
2: **for** $v_{start} \leq v \leq v_{end}$ in $V(G)$ **do**
3: $distance[v] \leftarrow \infty$
4: $path[v] \leftarrow null$
5: $Queue.push(v, dst)$
6: **end for**
7: $Queue.decrease(src, 0)$
8: **while** $Queue$ is not empty **do**
9: $v \leftarrow$ FindMin($rank$)
10: **for** u in $\Gamma(v)$ **do**
11: Relax($v, u,$ edge(v, u))
12: **end for**
13: **end while**
14: **return** $distance$, $path$
15: **end function**

1: **function** Relax(v, u, e)
2: $tmp \leftarrow distance[v] +$ weight(e)
3: **if** $distance[u] > tmp$ **then**
4: $distance[u] \leftarrow tmp$
5: $path[u] \leftarrow v$
6: $Queue.decrease(u, distance[u])$
7: **end if**
8: **end function**

1: **function** FindMin($rank$)
2: $p, v \leftarrow Queue.peek()$
3: put(shared[$rank$], (p, v), Master)
4: **if** $rank =$ MASTER **then**
5: $p, v \leftarrow$min($shared$)
6: **end if**
7: bcast(p, v)
8: **if** $v = Queue.peek()$ **then**
9: $Queue.pop()$
10: **end if**
11: **return** v
12: **end function**

Parallelizing Dijkstra's Algorithm (2)

Require: G, a weighted graph
Require: src, a source vertex

1: **function** ParallelDijkstra(G, src, v_{start}, v_{end})
2: **for** $v_{start} \leq v \leq v_{end}$ in $V(G)$ **do**
3: $distance[v] \leftarrow \infty$
4: $path[v] \leftarrow null$
5: $Queue.push(v, dst)$
6: **end for**
7: $Queue.decrease(src, 0)$
8: **while** $Queue$ is not empty **do**
9: $v \leftarrow$ FindMin($rank$)
10: **for** u in $\Gamma(v)$ **do**
11: Relax($v, u,$ edge(v, u))
12: **end for**
13: **end while**
14: **return** $distance$, $path$
15: **end function**

1: **function** Relax(v, u, e)
2: $tmp \leftarrow distance[v] +$ weight(e)
3: **if** $distance[u] > tmp$ **then**
4: $distance[u] \leftarrow tmp$
5: $path[u] \leftarrow v$
6: $Queue.decrease(u, distance[u])$
7: **end if**
8: **end function**

1: **function** FindMin($rank$)
2: $p, v \leftarrow Queue.peek()$
3: put(shared$[rank]$, (p, v), Master)
4: **if** $rank =$ MASTER **then**
5: $p, v \leftarrow$min$(shared)$
6: **end if**
7: bcast(p, v)
8: **if** $v = Queue.peek()$ **then**
9: $Queue.pop()$
10: **end if**
11: **return** v
12: **end function**

Parallelizing Dijkstra's Algorithm (2)

Require: G, a weighted graph
Require: src, a source vertex

1: **function** ParallelDijkstra(G, src, v_{start}, v_{end})
2: **for** $v_{start} \leq v \leq v_{end}$ in $V(G)$ **do**
3: $distance[v] \leftarrow \infty$
4: $path[v] \leftarrow null$
5: $Queue.push(v, dst)$
6: **end for**
7: $Queue.decrease(src, 0)$
8: **while** $Queue$ is not empty **do**
9: $v \leftarrow$ FindMin($rank$)
10: **for** u in $\Gamma(v)$ **do**
11: Relax($v, u,$ edge(v, u))
12: **end for**
13: **end while**
14: **return** $distance$, $path$
15: **end function**

1: **function** Relax(v, u, e)
2: $tmp \leftarrow distance[v] +$ weight(e)
3: **if** $distance[u] > tmp$ **then**
4: $distance[u] \leftarrow tmp$
5: $path[u] \leftarrow v$
6: $Queue.decrease(u, distance[u])$
7: **end if**
8: **end function**

1: **function** FindMin($rank$)
2: $p, v \leftarrow Queue.peek()$
3: put(shared$[rank]$, (p, v), Master)
4: **if** $rank =$ MASTER **then**
5: $p, v \leftarrow$min$(shared)$
6: **end if**
7: bcast(p, v)
8: **if** $v = Queue.peek()$ **then**
9: $Queue.pop()$
10: **end if**
11: **return** v
12: **end function**

Parallel Dijkstra's Algorithm

- Improvement:
  - Reduce update operations per call to FindMin()
    - Only put vertex from local priority queue to "Master" (i.e., PE 0) if value changed from previous iteration

Serial Bellman-Ford

- Undirected/directed graphs with positive or negative weights
- O (|V| * |E|)

```
Require: G, a weighted graph
Require: src, a source vertex
 1: function BellmanFord(G, src)
 2:     for v in V(G) do
 3:         distance[v] ← ∞
 4:         path[v] ← null
 5:     end for
 6:     distance[src] ← 0
 7:     for k = 1 to |V| − 1 do
 8:         for v in V(G) do
 9:             for u in Γ(v) do
10:                 Relax(v, u, edge(v, u))
11:             end for
12:         end for
13:     end for
14:     return distance, path
15: end function
```

```
1: function Relax(v, u, e)
2:     tmp ← distance[v] + weight(e)
3:     if distance[u] > tmp then
4:         distance[u] ← tmp
5:         path[u] ← v
6:     end if
7: end function
```

Serial Bellman-Ford

- Undirected/directed graphs with positive or negative weights
- O (|V| * |E|)

Initialize data structures

```
Require: G, a weighted graph
Require: src, a source vertex
 1: function BellmanFord(G, src)
 2:     for v in V(G) do
 3:         distance[v] ← ∞
 4:         path[v] ← null
 5:     end for
 6:     distance[src] ← 0
 7:     for k = 1 to |V| − 1 do
 8:         for v in V(G) do
 9:             for u in Γ(v) do
10:                 Relax(v, u, edge(v, u))
11:             end for
12:         end for
13:     end for
14:     return distance, path
15: end function
```

```
1: function Relax(v, u, e)
2:     tmp ← distance[v] + weight(e)
3:     if distance[u] > tmp then
4:         distance[u] ← tmp
5:         path[u] ← v
6:     end if
7: end function
```

Serial Bellman-Ford

- Undirected/directed graphs with positive or negative weights
- O (|V| * |E|)

```
Require: G, a weighted graph
Require: src, a source vertex
 1: function BellmanFord(G, src)
 2:     for v in V(G) do
 3:         distance[v] ← ∞
 4:         path[v] ← null
 5:     end for
 6:     distance[src] ← 0
 7:     for k = 1 to |V| − 1 do
 8:         for v in V(G) do
 9:             for u in Γ(v) do
10:                 Relax(v, u, edge(v, u))
11:             end for
12:         end for
13:     end for
14:     return distance, path
15: end function
```

Continually loop over vertices until complete

```
1: function Relax(v, u, e)
2:     tmp ← distance[v] + weight(e)
3:     if distance[u] > tmp then
4:         distance[u] ← tmp
5:         path[u] ← v
6:     end if
7: end function
```

Serial Bellman-Ford

- Undirected/directed graphs with positive or negative weights
- O (|V| * |E|)

```
Require: G, a weighted graph
Require: src, a source vertex
 1: function BellmanFord(G, src)
 2:     for v in V(G) do
 3:         distance[v] ← ∞
 4:         path[v] ← null
 5:     end for
 6:     distance[src] ← 0
 7:     for k = 1 to |V| − 1 do
 8:         for v in V(G) do
 9:             for u in Γ(v) do
10:                 Relax(v, u, edge(v, u))
11:             end for
12:         end for
13:     end for
14:     return distance, path
15: end function
```

```
1: function Relax(v, u, e)
2:     tmp ← distance[v] + weight(e)
3:     if distance[u] > tmp then
4:         distance[u] ← tmp
5:         path[u] ← v
6:     end if
7: end function
```

Return result

Parallelizing Bellman-Ford

- Relatively simple to parallelize:
  - Naïve uniform graph partitioning
  - Each PE operates on its own partition
- Challenges
  - Global shared data structures (distance and path)
    - Retain consistency through AMOs
  - Synchronization
    - Collective operation (barrier) at end of each iteration

Parallelizing Bellman-Ford (2)

Require: G: a weighted partitioned graph
Require: src: a source Vertex

```
1:  function parallelBellmanFord(G, src)
2:      for v ∈ local V(G) do
3:          distance[v] = ∞
4:          path[v] = null
5:      end for
6:      if src is local then
7:          distance[src] = 0
8:      end if
9:      for k = 1 to |V| − 1 do
10:         for v in local V(G) do
11:             for u in Γ(v) do
12:                 Relax(v, u, edge(v, u))
13:             end for
14:         end for
15:         Barrier
16:     end for
17:     return distance, path
18: end function
```

```
1: function Relax(v, u, e)
2:     d_u ← get(distance[u])
3:     tmp ← distance[v] + weight(e)
4:     if d_u > tmp then
5:         atomic_cswap(distance[u], tmp)
6:         atomic_cswap(path[u], v)
7:     end if
8: end function
```

Parallelizing Bellman-Ford (2)

```
Require: G: a weighted partitioned graph
Require: src: a source Vertex
 1: function ParallelBellmanFord(G, src)
 2:     for v ∈ local V(G) do
 3:         distance[v] = ∞
 4:         path[v] = null
 5:     end for
 6:     if src is local then
 7:         distance[src] = 0
 8:     end if
 9:     for k = 1 to |V| − 1 do
10:         for v in local V(G) do
11:             for u in Γ(v) do
12:                 Relax(v, u, edge(v, u))
13:             end for
14:         end for
15:         Barrier
16:     end for
17:     return distance, path
18: end function
```

Becomes localized to the partition

```
 1: function Relax(v, u, e)
 2:     d_u ← get(distance[u])
 3:     tmp ← distance[v] + weight(e)
 4:     if d_u > tmp then
 5:         atomic_cswap(distance[u], tmp)
 6:         atomic_cswap(path[u], v)
 7:     end if
 8: end function
```

Parallelizing Bellman-Ford (2)

```
Require: G: a weighted partitioned graph
Require: src: a source Vertex
 1: function ParallelBellmanFord(G, src)
 2:     for v ∈ local V(G) do
 3:         distance[v] = ∞
 4:         path[v] = null
 5:     end for
 6:     if src is local then
 7:         distance[src] = 0
 8:     end if
 9:     for k = 1 to |V| − 1 do
10:         for v in local V(G) do
11:             for u in Γ(v) do
12:                 Relax(v, u, edge(v, u))
13:             end for
14:         end for
15:         Barrier
16:     end for
17:     return distance, path
18: end function
```

Becomes localized to the partition

```
1: function Relax(v, u, e)
2:     d_u ← get(distance[u])
3:     tmp ← distance[v] + weight(e)
4:     if d_u > tmp then
5:         atomic_cswap(distance[u], tmp)
6:         atomic_cswap(path[u], v)
7:     end if
8: end function
```

Use of atomics to guarantee consistency for globally shared distance and path structures

Parallelizing Bellman-Ford (3)

- Improvements
  - Reduce atomics by removing non-contributing edges from graph and labelling internal vertices during graph reading and partitioning
    - Non-contributing edges: self-loops
  - Reduce number of atomics by combining distance/path structure into one
    - Split 64-bit value by some ratio depending on the maximum weight in the graph
  - Label a vertex as active if it is updated
    - An active vertex needs to update its adjacent vertices
  - End algorithm execution when convergence occurs
    - Replace barrier with all-reduce on convergence
  - Take advantage of one-sided operations
    - Remove synchronization requirements (asynchronous operation)

Parallelizing Bellman-Ford (4)

- Convergence for asynchronous approach
  1. Atomic-based: uses atomic add to implement a counter
     - PE 0 used as base
     - Add 1 to increment counter, add -1 to decrement counter
  2. Get-based: locally set *convergence* flag, get all neighbors' flag
     - Loop from PE 0 to PE n
     - If convergence flag not set, assume not converged
       - Useful for power-law graphs

Evaluation

- Testbed
  - Titan at OLCF
  - Cray SHMEM
  - Fully loaded each node (16 PEs per node)

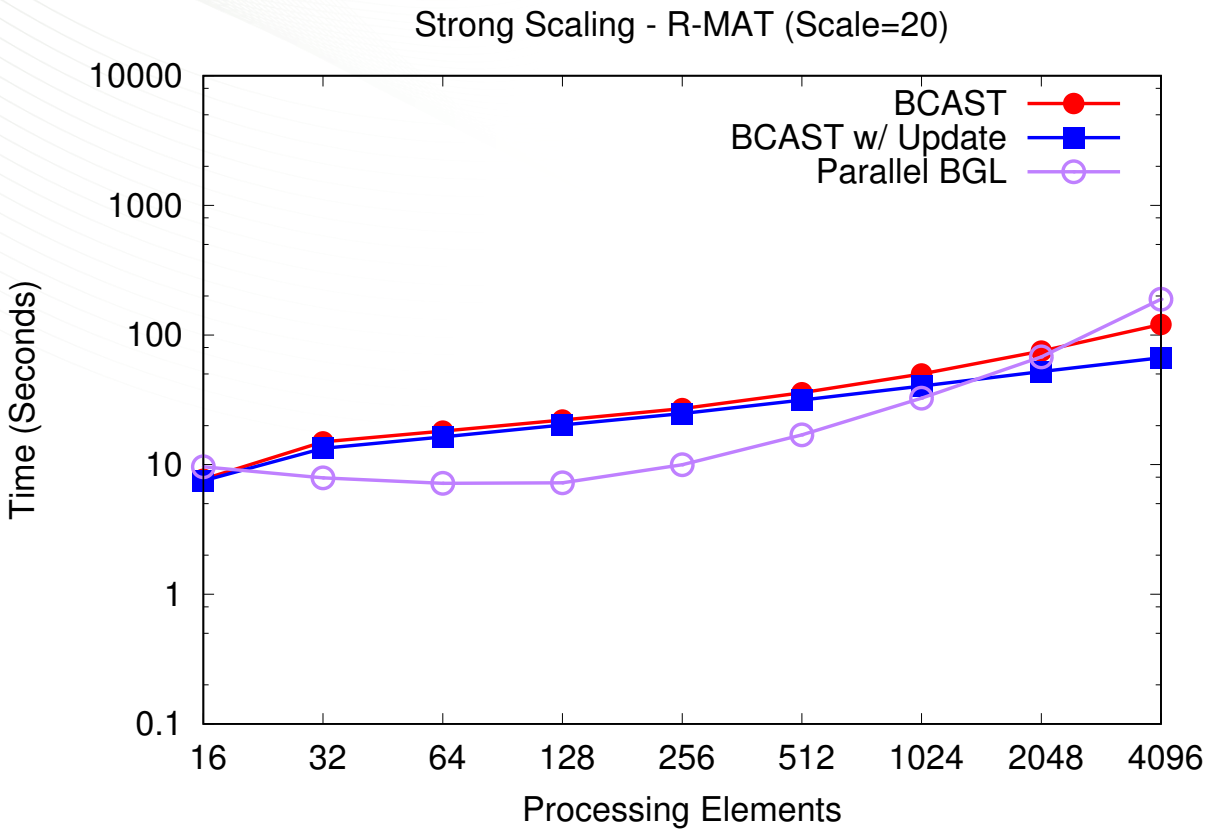
	Facebook	Twitter	LiveJournal	Road-CA	Road-TX
Vertices	4,039	81,306	4,847,571	1,971,281	1,393,383
Edges	176,468	2,420,766	68,993,773	5,516,784	3,805,842
Clustering Coef.	0.6055	0.5653	0.2742	0.0464	0.0470
Diameter	8	7	16	849	1054

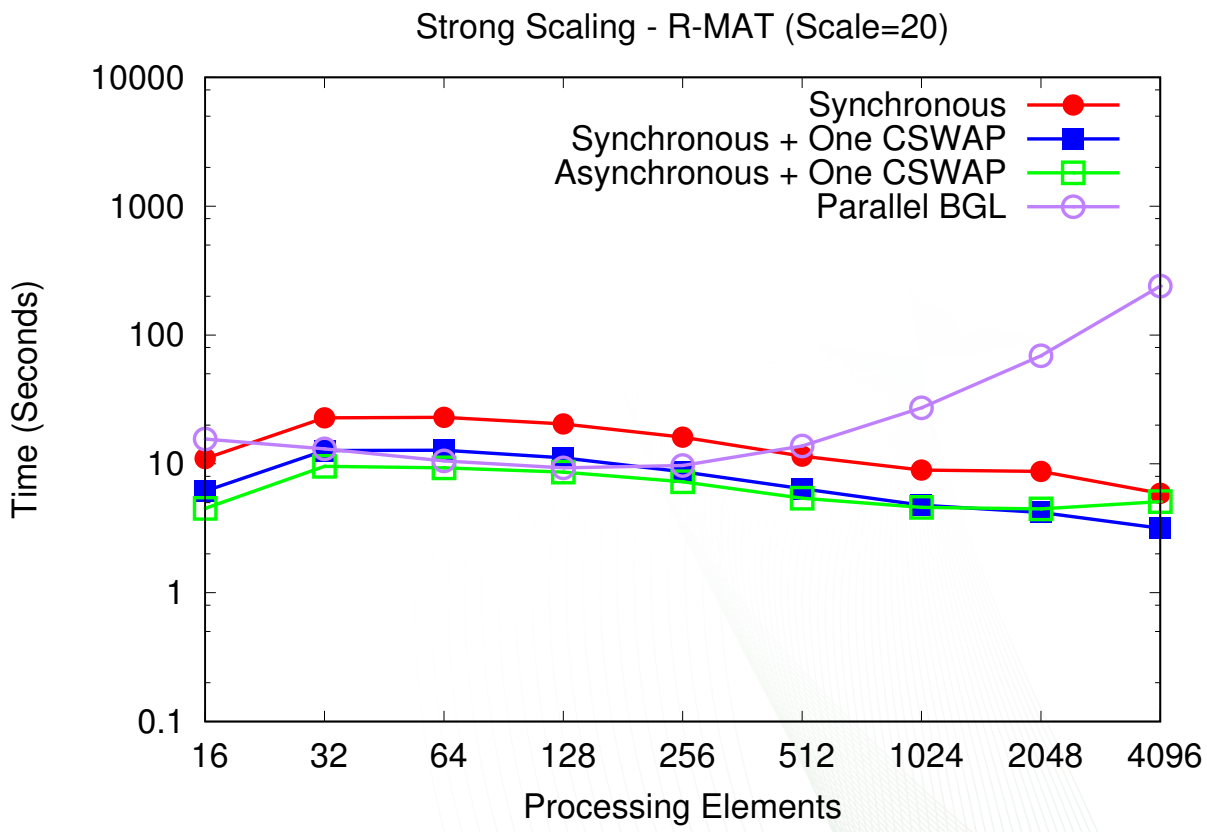
- Workloads
  - Synthetic graphs (R-MAT and Small-World graphs)
    - R-MAT: power-law graph, average of 16 edges per vertex
    - Small-world: regular graph with 1024 edges per vertex
  - Real-world Graphs (Road Maps, Facebook, Twitter, LiveJournal)
- Compared against Parallel Boost Graph Library (Parallel BGL)
  - Cray MPICH

Parallel BGL

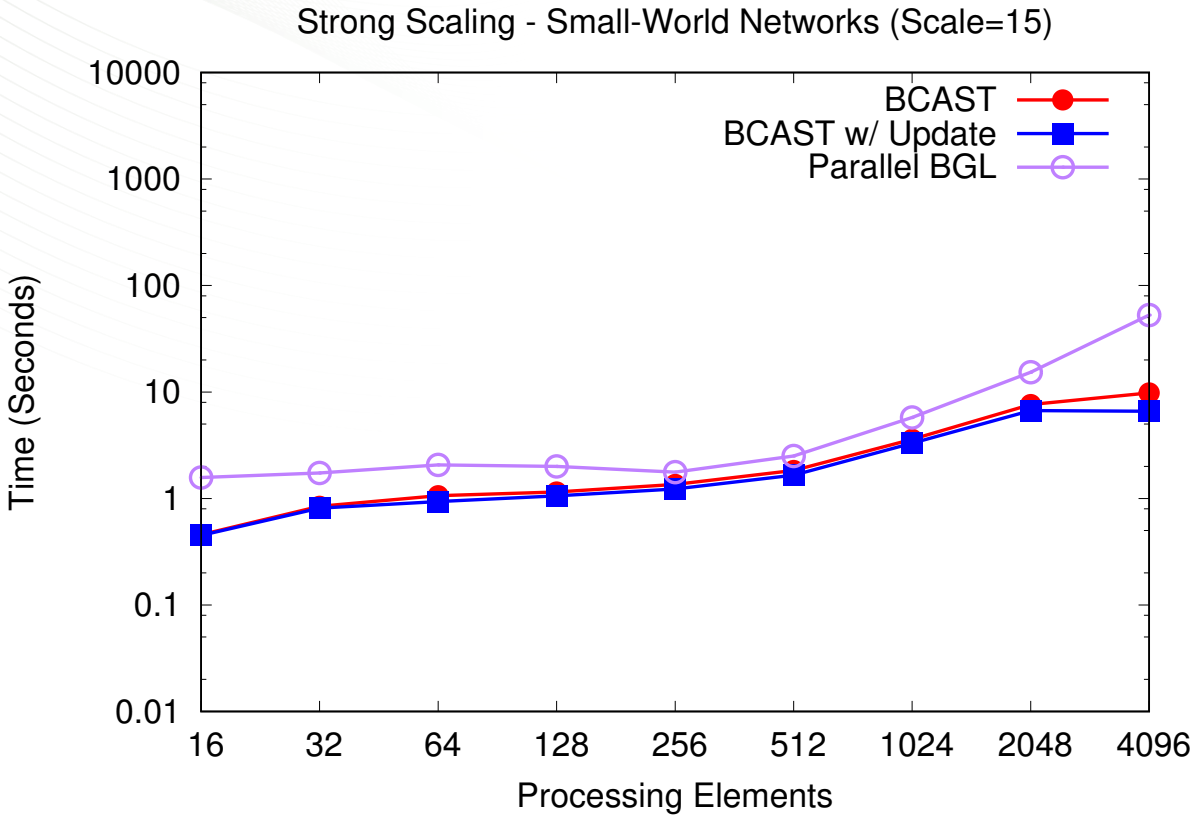
- MPI-based
  - Collectives (All-reduce)
- Dijkstra's Algorithm:
  - Breadth first search with Parallel BGL's distributed Priority Queue
- No native support for Bellman-Ford
  - Use Delta-Stepping implementation with delta set to maximum weight
    - Algorithmically equivalent
- Uniform graph partitioning

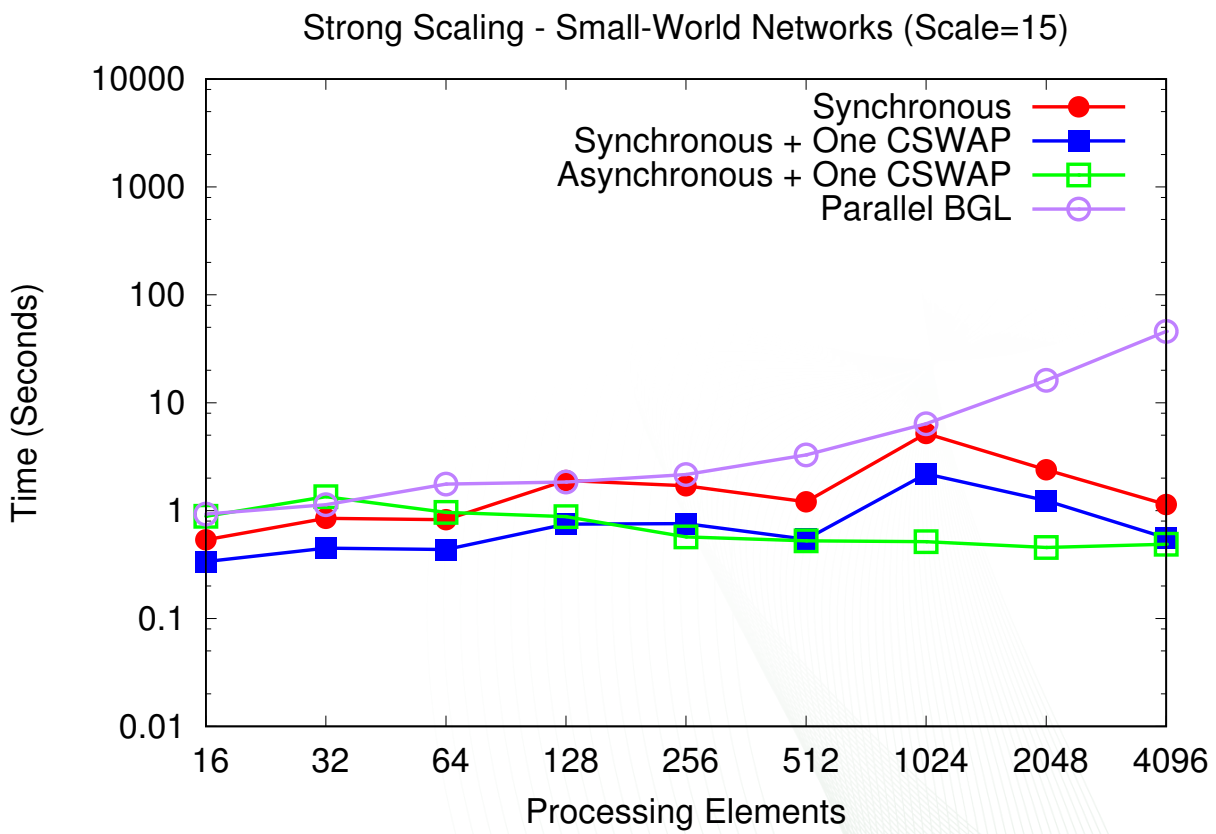
R-MAT Graph



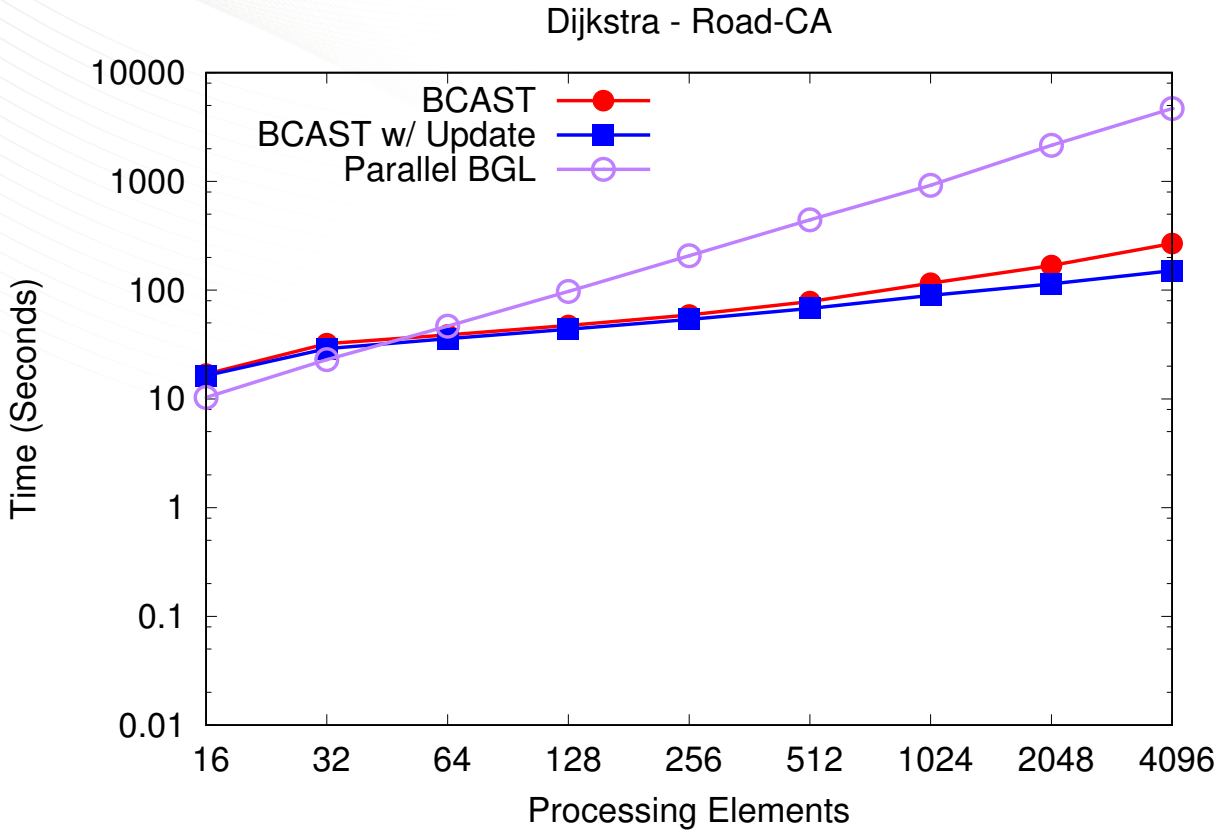


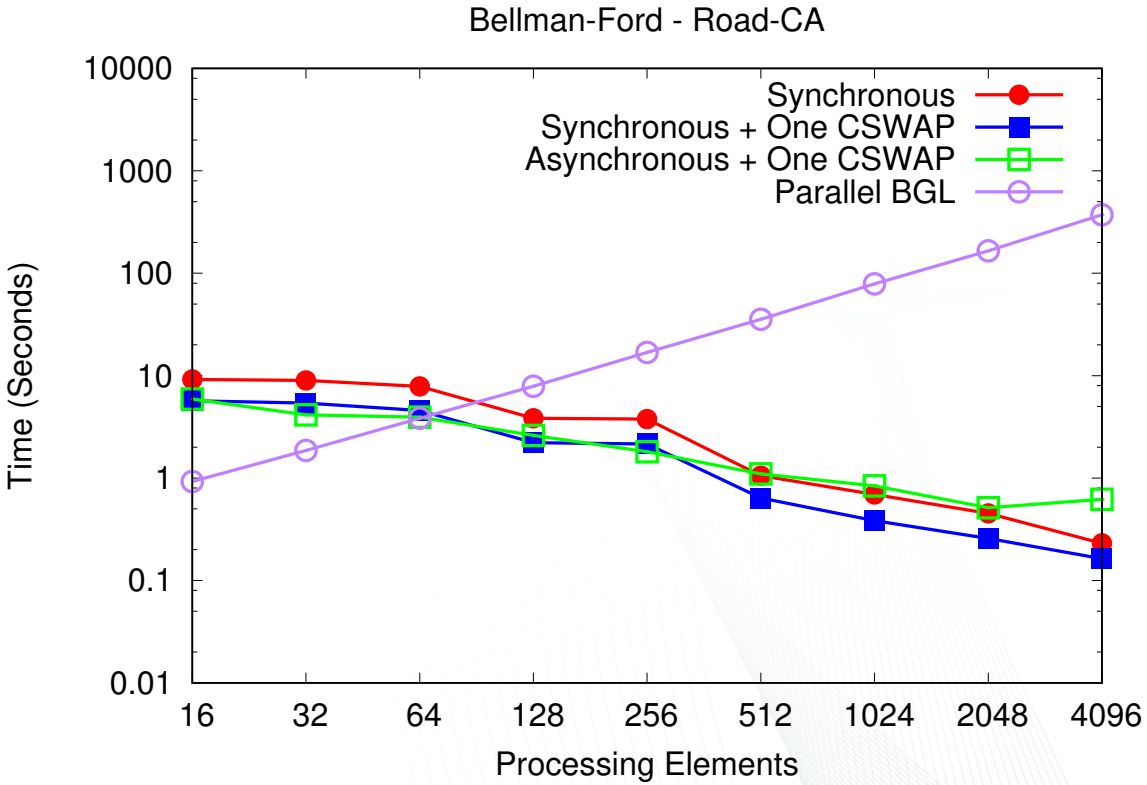
Small-World Graph



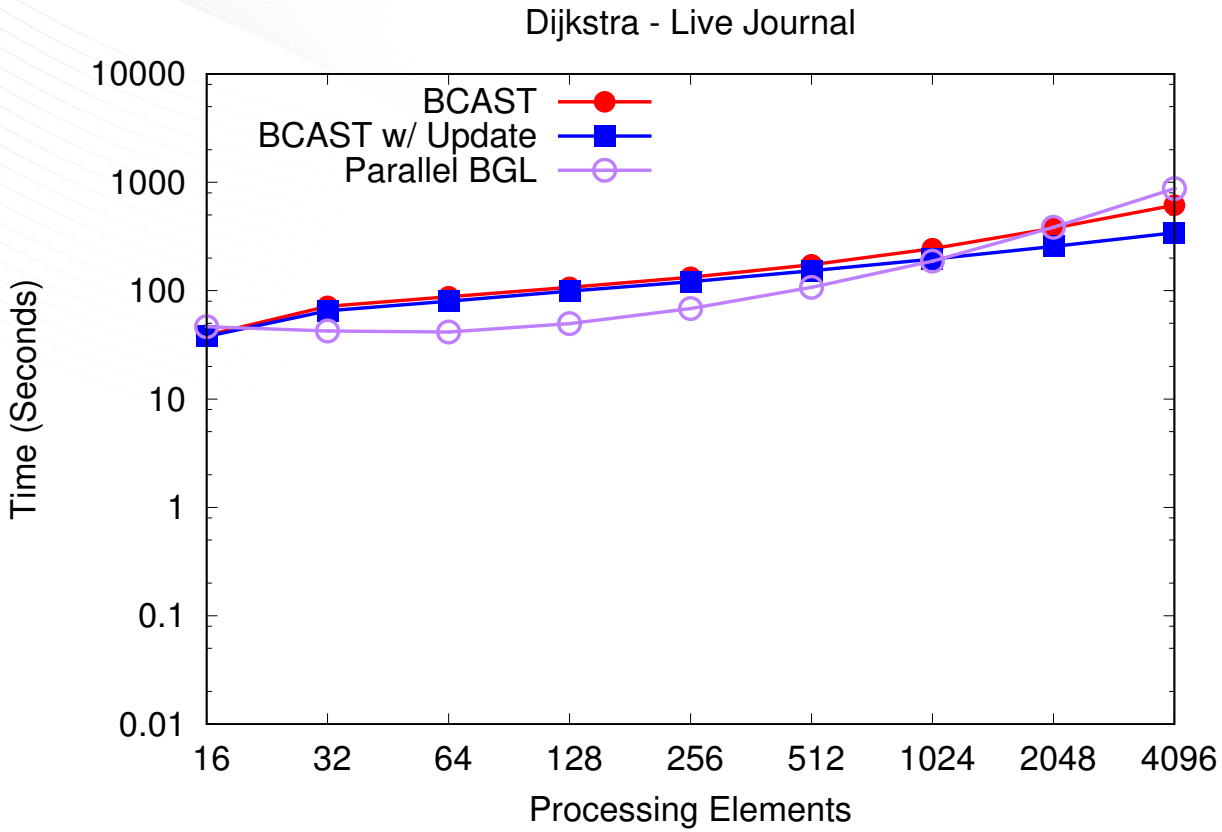


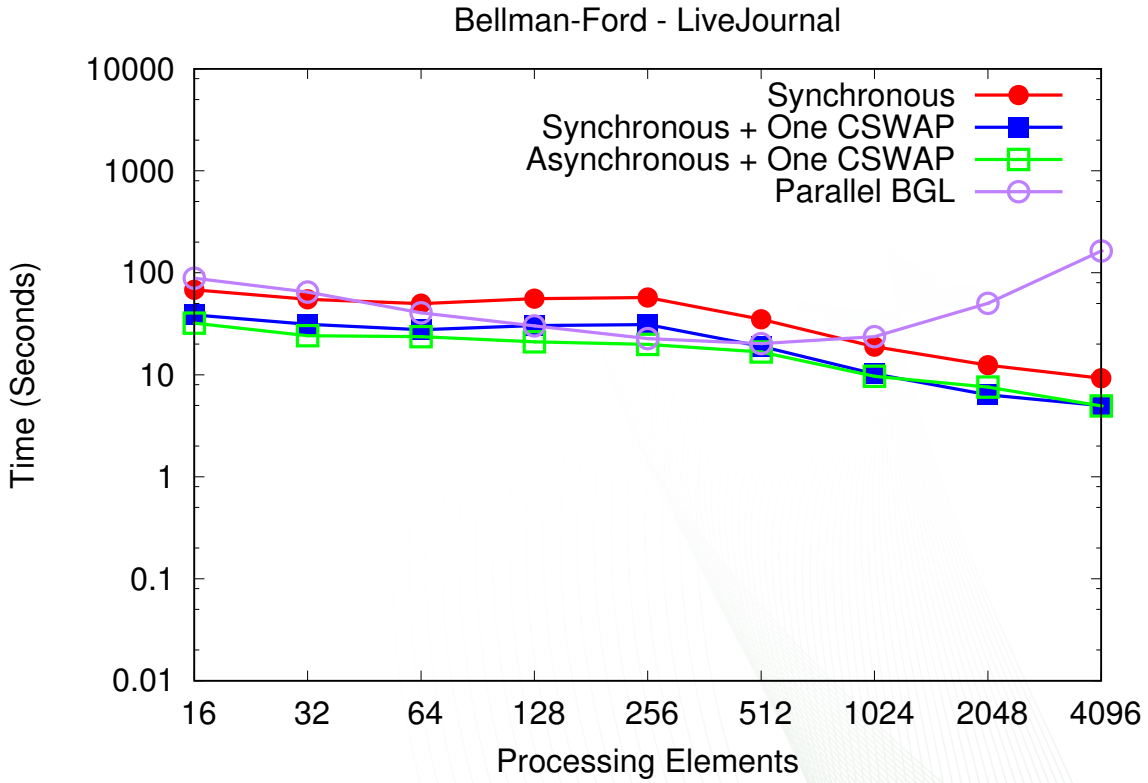
Real-World: California Road Map





Real-World: LiveJournal





Conclusions

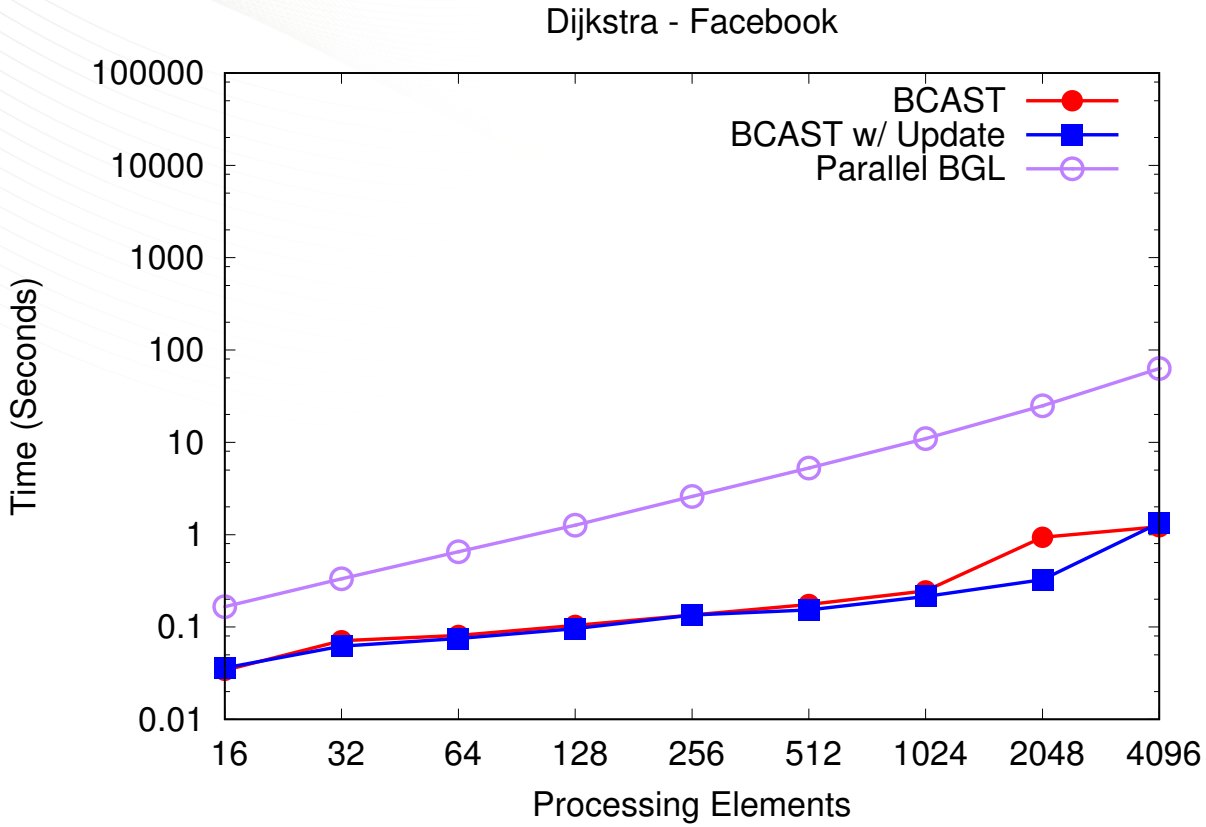
- The parallelization of Dijkstra's algorithm is difficult to achieve scalability
  - Works best for balanced workloads (i.e., small-world graphs, road maps)
  - Forced synchronization removes many advantages from OpenSHMEM
- The parallel version of Bellman-Ford scales, but not perfectly
  - Synchronous approach useful for balanced workloads and power-law graphs
  - Currently, the asynchronous approach does not result in major improvement at larger scales for power-law graphs
- Initial OpenSHMEM prototypes scale better than Parallel BGL

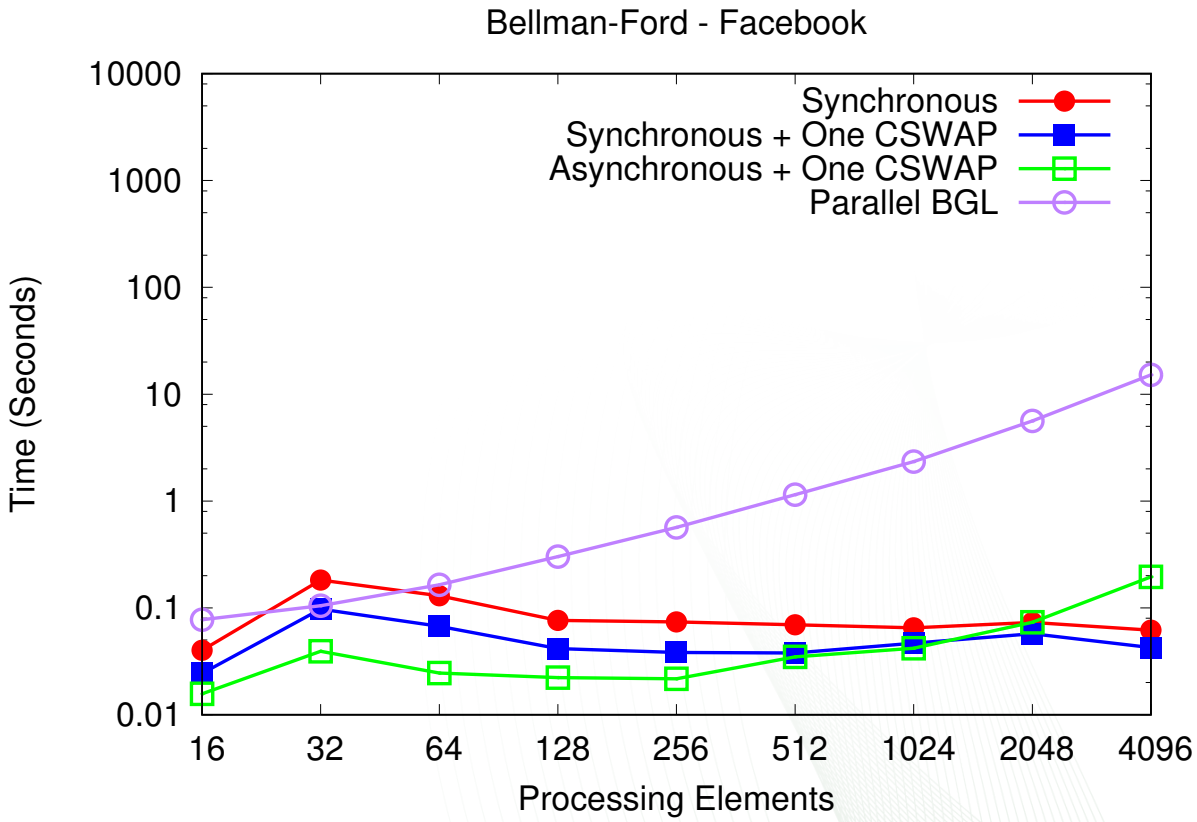
Possible Next Steps

- Explore these algorithms perform on GPUs with OpenSHMEM
  - What will be required to improve the scalability?
- Improve the efficiency of the implementation
  - Frontiers
  - Scalable convergence determination for the asynchronous approach

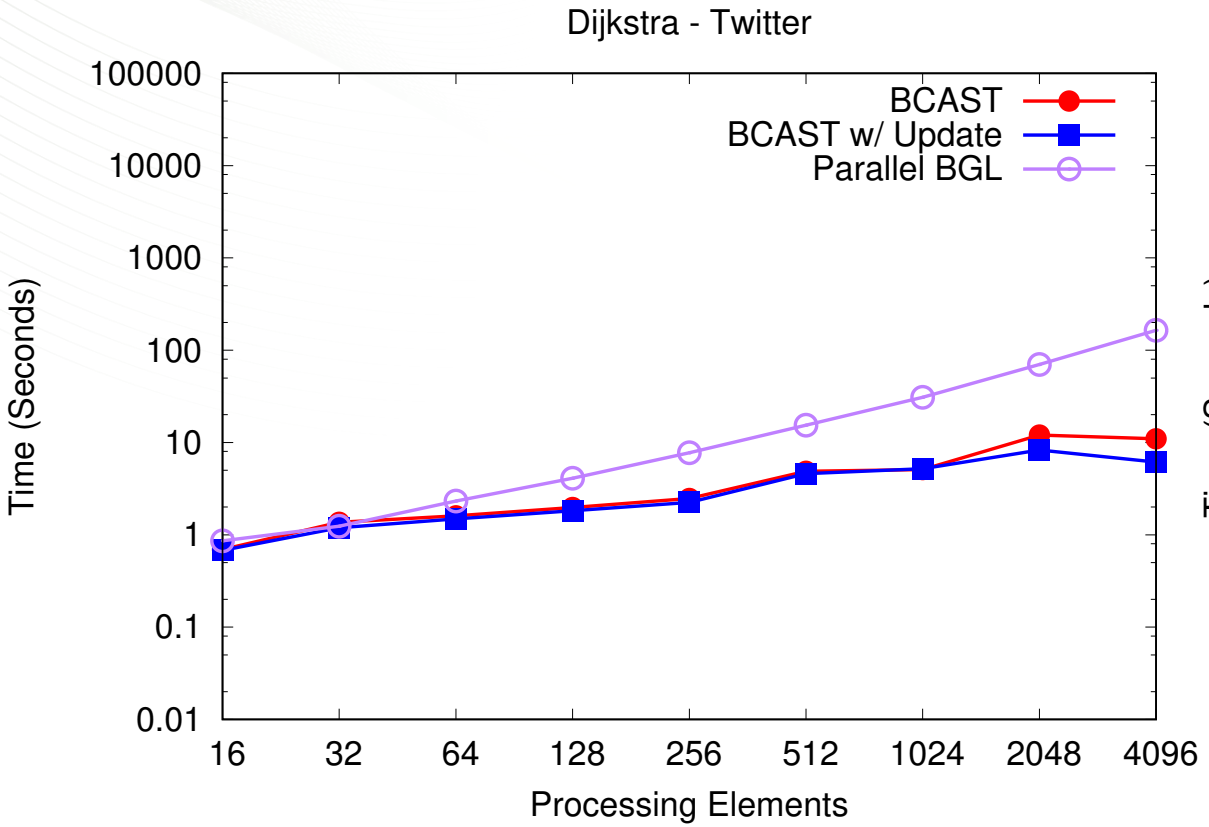
Questions??

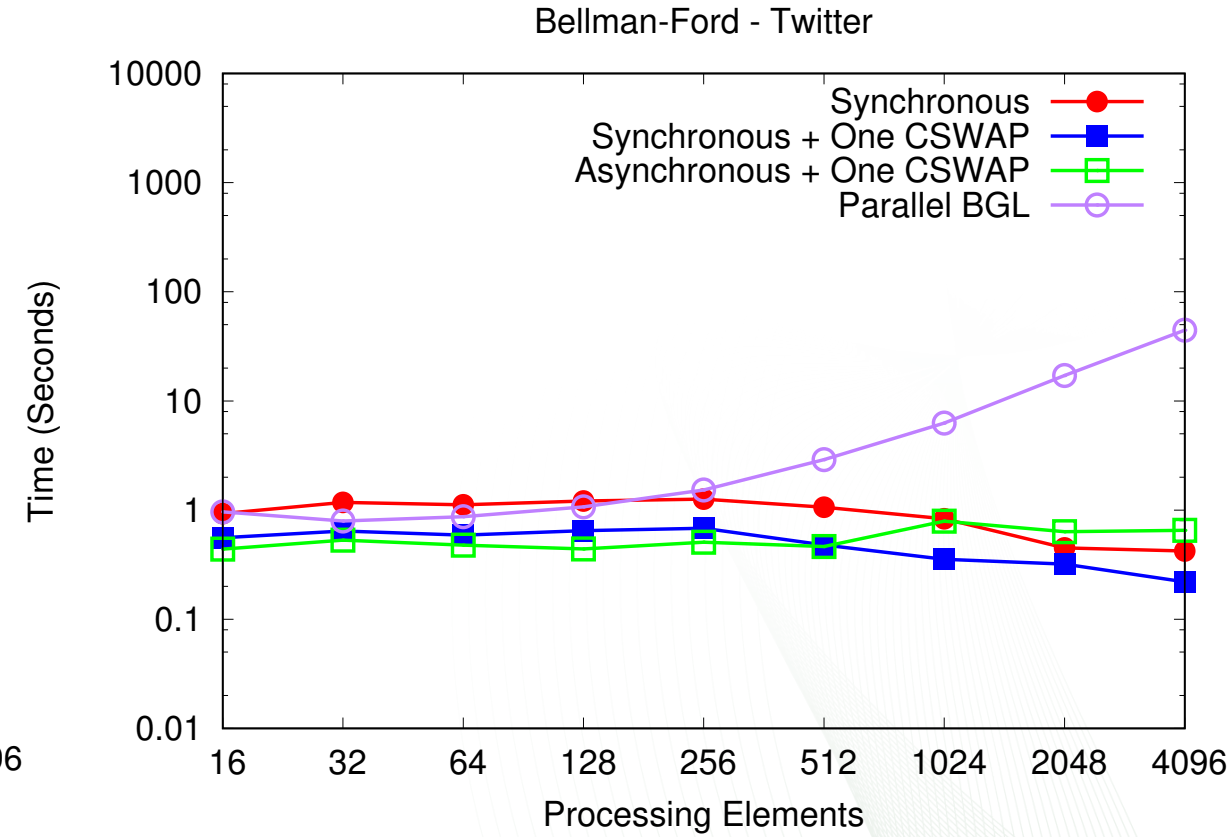
Real-World: Facebook





Real-World: Twitter





Real-World: Texas Road Map