Algorithms for Updating Dynamic Networks

Presented by

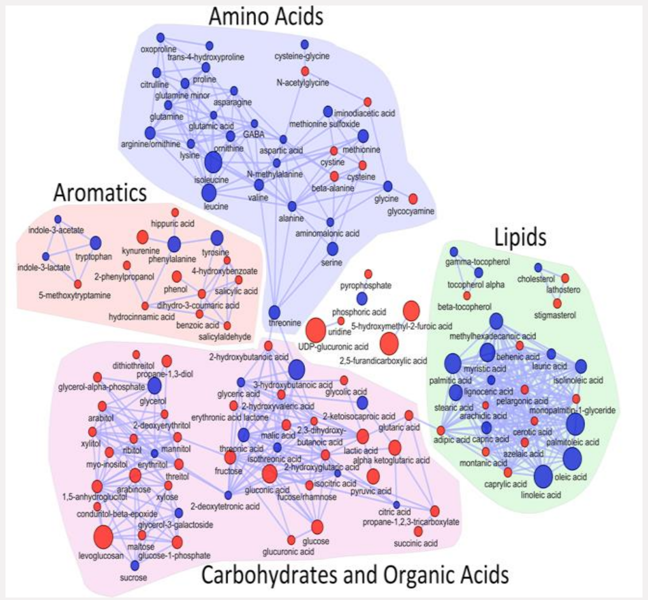
Sriram Srinivasan

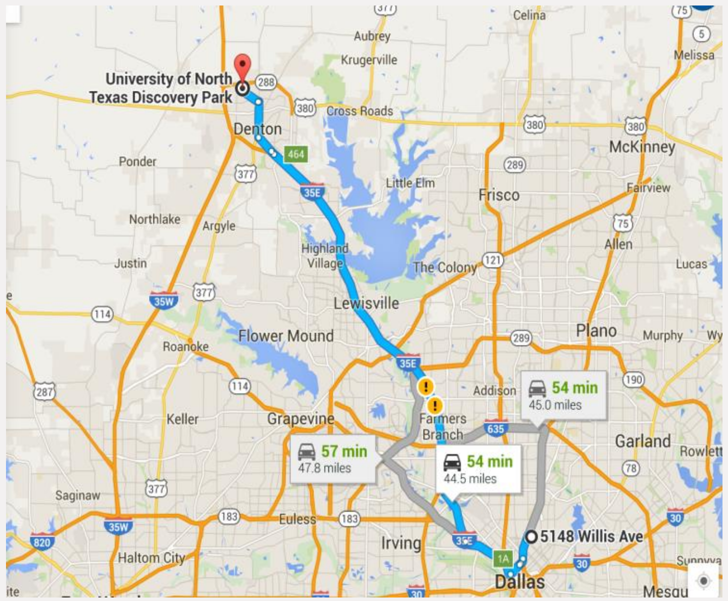
sriramsrinivas@unomaha.edu

github.com/SriramSrinivas/SriramDissertation

Networks

Networks (graphs) are used to model interaction among entities





Dynamic Networks

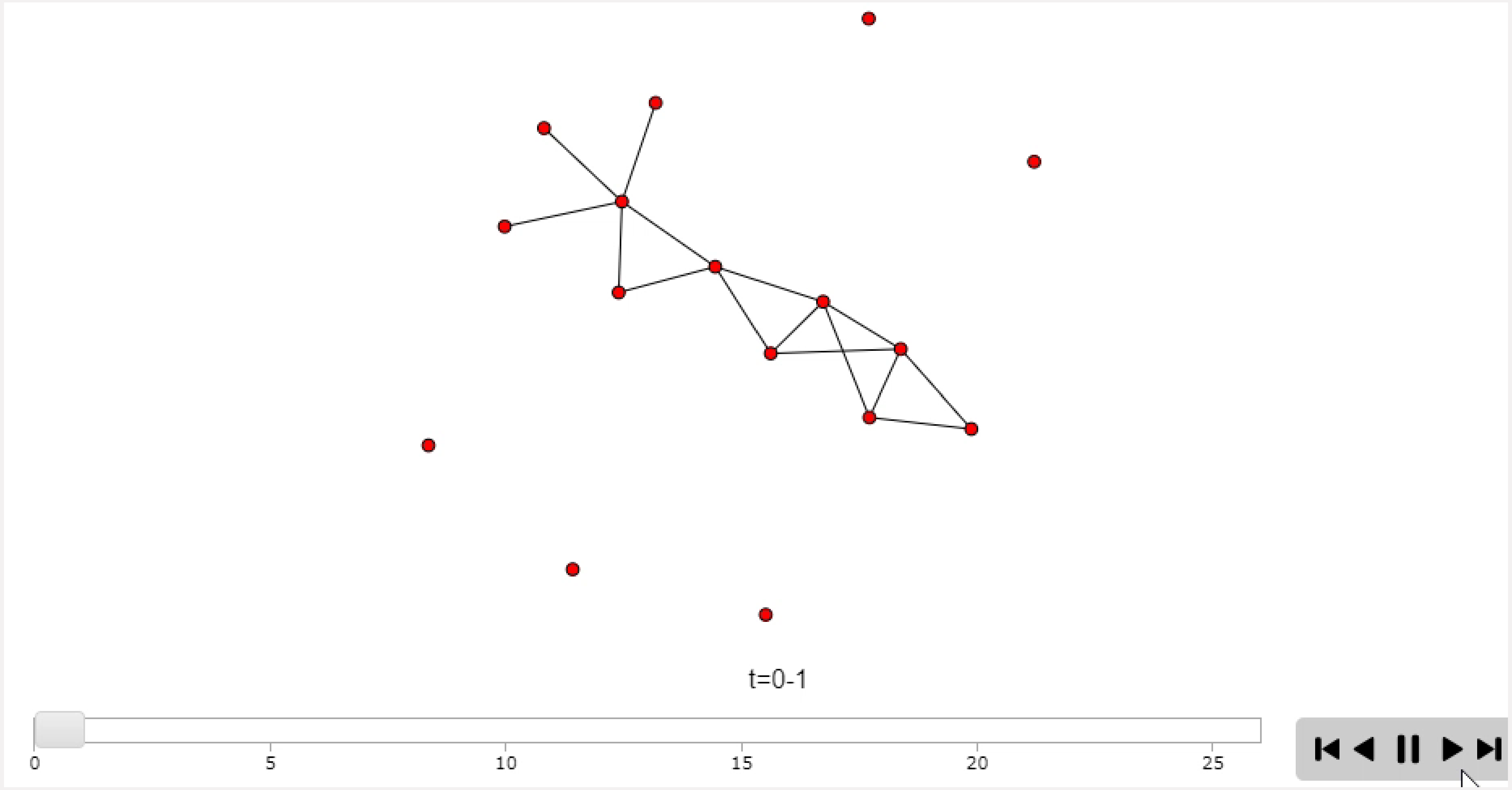
Analyzing properties of the networks can help in understanding the characteristics of the underlying systems

- Communities indicate groups of friends in social networks
- High centrality vertices indicate in important proteins in PPI networks

Networks can evolve with time, so the properties of the dynamic networks have to be updated

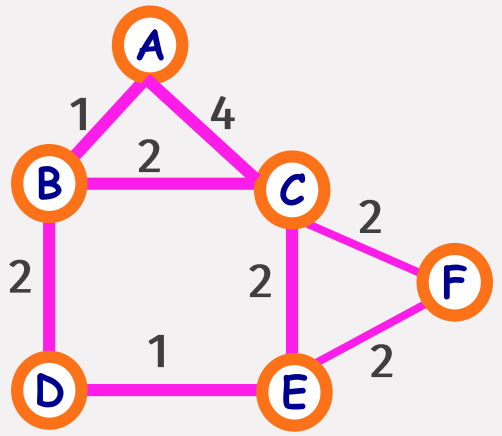
Goal: Update only the part of the network affected by the change, rather than recomputing from scratch

Dynamic Networks Visualization



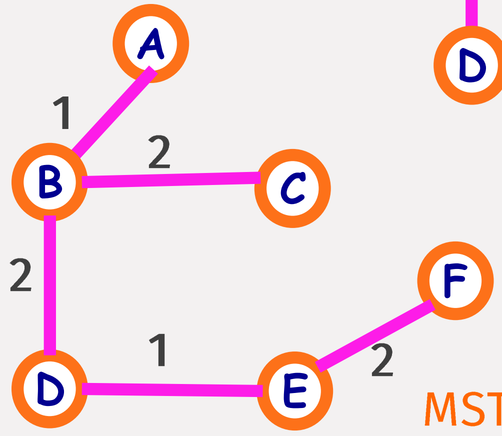
** Simulation was designed using R & d3.js (work still in progress to visualize real-world dynamic networks) **

Two Properties



Minimum Weighted Spanning Tree (MST)

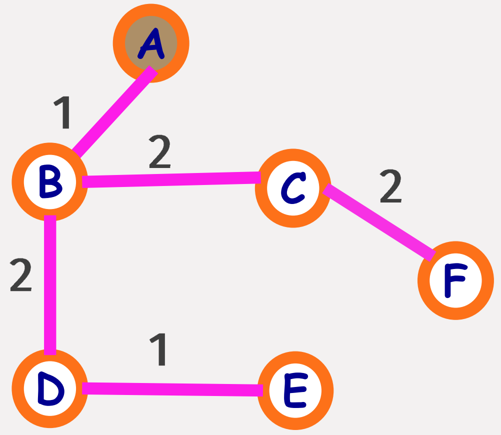
Select a subset of edges from an undirected weighted graph (V,E), such that
(i) all the vertices are connected
(ii) the sum of the total edges is minimized



MST: Key Edges

Single Source Shortest Path (SSSP)

Find the shortest distance of all vertices from a given source vertex



SSSP: Key Edges

The naïve approach

Recompute from scratch

Generate new graph after updates, perform SSSP, MST again

Can we do better than recomputing from scratch?

Related Work

Libraries for dynamic data/graph analysis

e.g., Sandia PHISH, Georgia Tech Stinger

Dynamic graph algorithms

e.g., Ramalingam-Reps, Narvez et al.

Parallel algorithms, implementations for SSSP in static graphs

e.g., Delta-stepping, DSMR

Assume batched updates

Consider a sequence of insertions and deletions

Edge operations considered

Vertex insertions and deletions can be modeled by adding and deleting edges

Observations about graph updates

- Updates may only affect a subgraph and the complete graph need not be analyzed

- Not all updates affect the property updates can be processed in parallel

- Not all updates affect the same subgraph affected subgraphs can be processed in parallel

Template for Parallel Algorithm

Sparsification

Compute only over the edges that affect the property (Key Edges)
Remaining edges accessed for deletion only
Preprocessing before input of changes

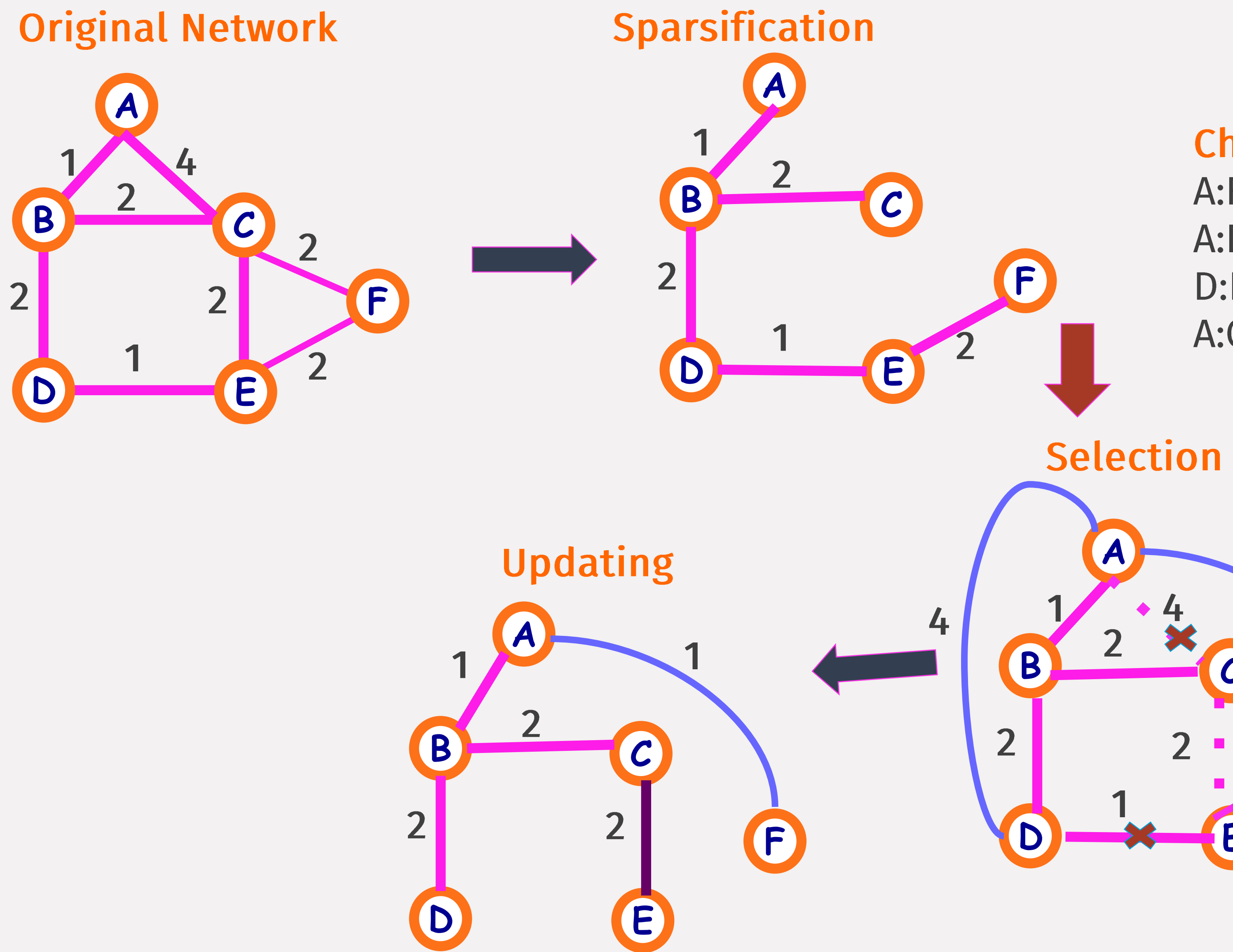
Selection

Identify the added/deleted edges that affect the property
Can be done in parallel for each edge
Edges are not yet added or deleted from the MST

Updating

Update set of key edges according to the changes
Parallel over the edges, but requires multiple iterations
Non-key edges may need to be processed

Updating Minimum Spanning Tree



Change Set
A:F:1 (Ins)
A:D:4 (Ins)
D:E:1 (Del)
A:C:4 (Del)

Issues with Insertion-I

An edge A-B is inserted if there is an edge in path from A to B in the MST that had higher weight

Finding the path between (u,v) for insertion—worst case complexity $O(V+E)$. Can occur for each insertion

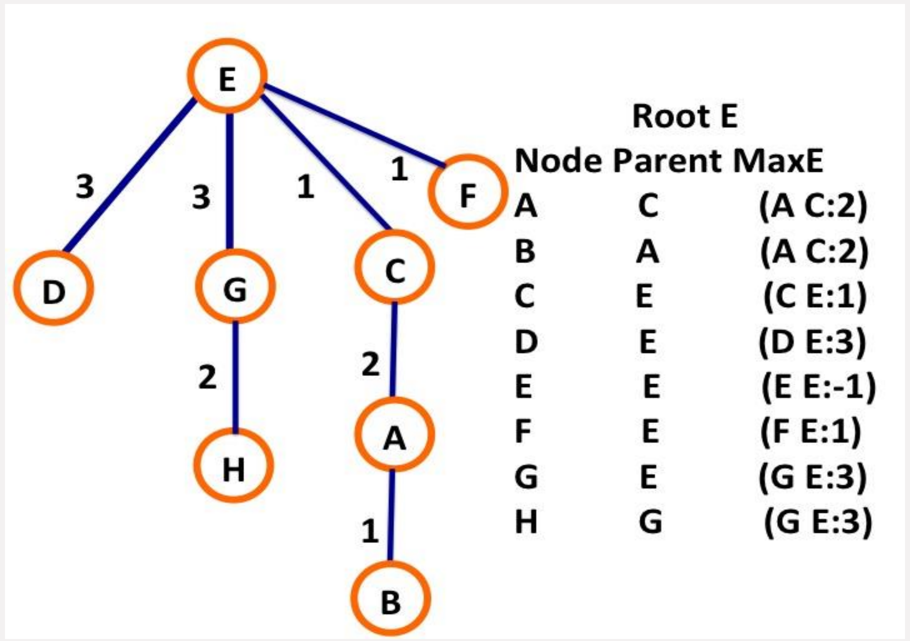
Complexity of simply re-doing the MST $O(ELogV)$

Solution: Store the maximum weighted edges between vertex pairs. Requires $O(V^2)$ storage

Finding Maximum Weighted Edges

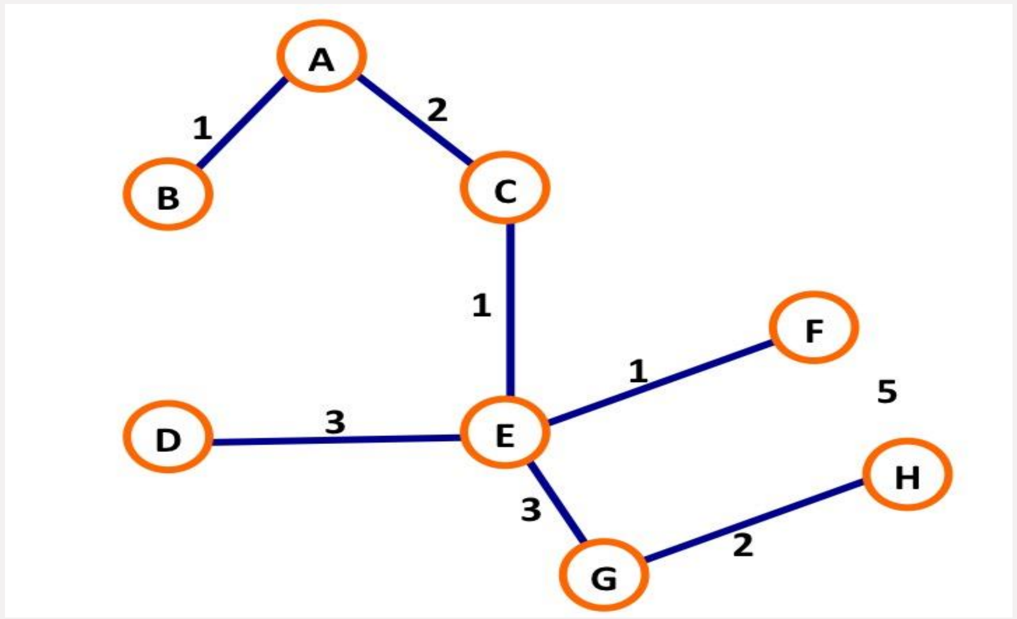
Find path from a designated root to all other vertices

Mark the edges that have maximum weight in these paths

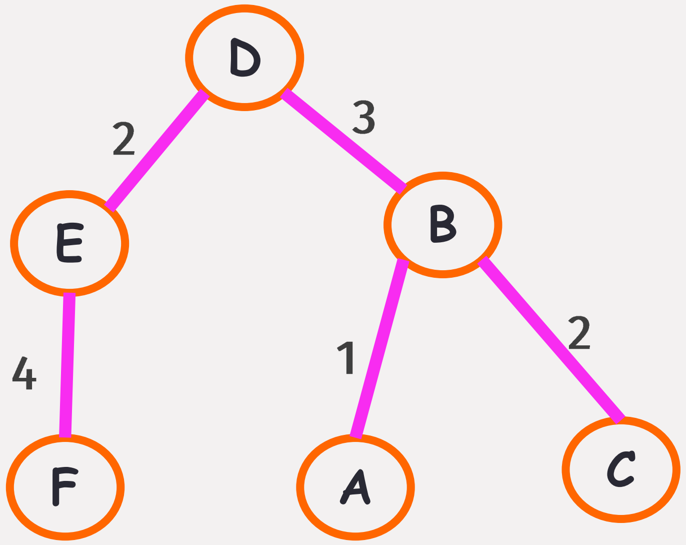


Root E

Node	Parent	MaxE
A	C	(A C:2)
B	A	(A C:2)
C	E	(C E:1)
D	E	(D E:3)
E	E	(E E:-1)
F	E	(F E:1)
G	E	(G E:3)
H	G	(G E:3)



Finding Maximum Weighted Edges



Case 1: (F:C) Max Weight Edges are Different
Max Weight from F:D (E-F) 4
Max Weight from C:D (B-D) 3

Pick the highest weight edge (E-F) 4

Case 2: (A:C) Max Weight Edges are Same
Max Weight from A:D (B-D) 3
Max Weight from C:D (B-D) 3

Find path from A-C and then find max weighted edge B:C 2

If we keep track of the parent, the complexity of this at most O(h); h=height of the tree

Issues with Deletion

Deletion can be done in parallel by simply marking the edge as deleted

Need to reassign component id of disconnected trees to recombine them

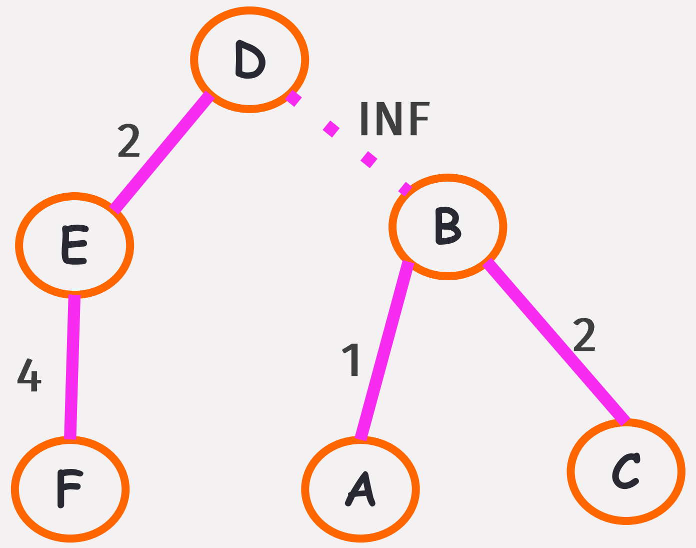
Solution: Mark deleted edges with very high weight (INF)

Apply insertion of remaining edges—reduces to insertion problem

Keep remainder edges in buckets of increasing weight

Once tree is reconnected stop

Deletion and Tree Repair



Delete (D:B)

Edge D-B connecting components (D,E,F) and (A,B,C) is set to INF

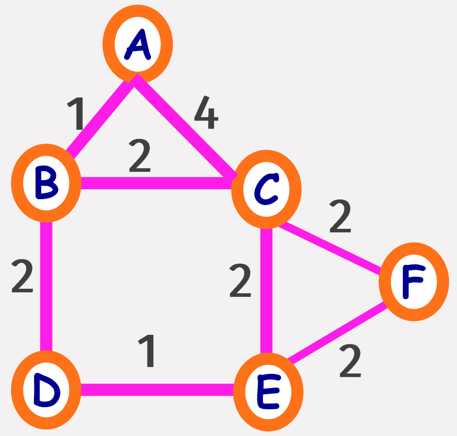
Any edge connecting these two components will identify D-B as the highest weighted edge and replace it.

No need for component reassignment

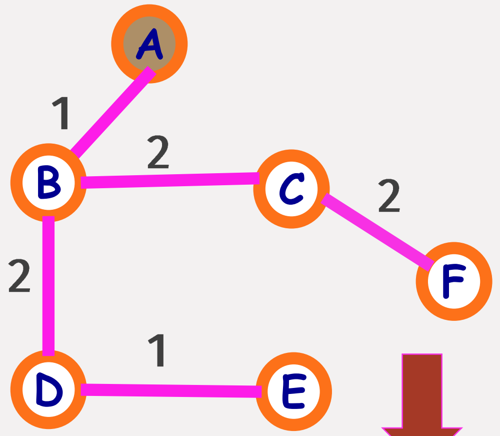
Checking for highest edge will always be under Case 1

Updating Single Source Shortest Path

Original Network

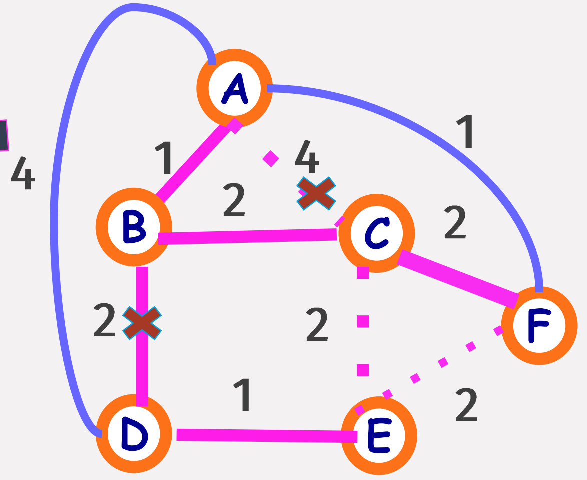


Sparsification

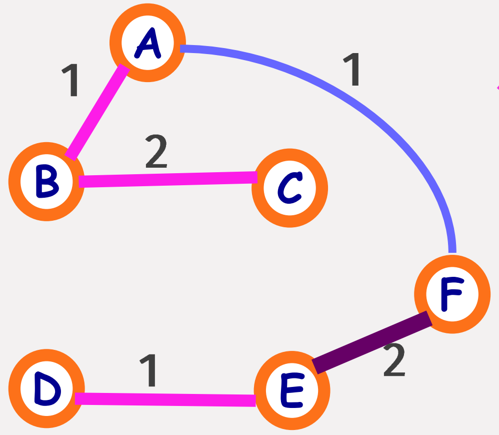


Change Set
A:F:1 (Ins)
A:D:4 (Ins)
D:B:2 (Del)
A:C:4 (Del)

Selection



Updating



Shared-memory parallelization

The Selection step is easy to implement and shows good load balance

The parallel performance of the Updating step is dependent on the number of affected vertices and the size of the subgraphs they alter. Vertex degree distributions can cause further load imbalance.

Asynchronous updates: can process longer paths instead of just neighbors. Reduce number of synchronization steps.

Empirical results

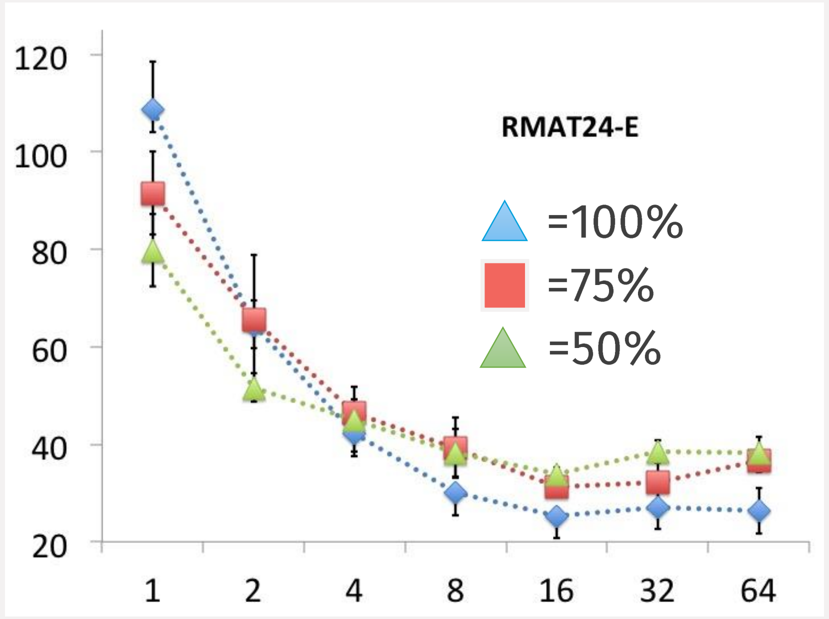
Results on a 36-core Intel Haswell system with 256 GB memory

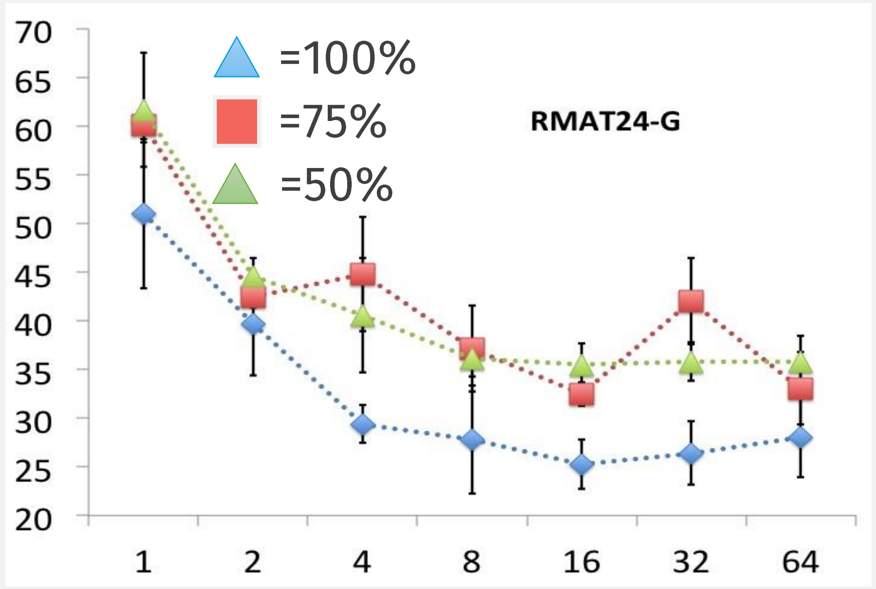
OpenMP implementation

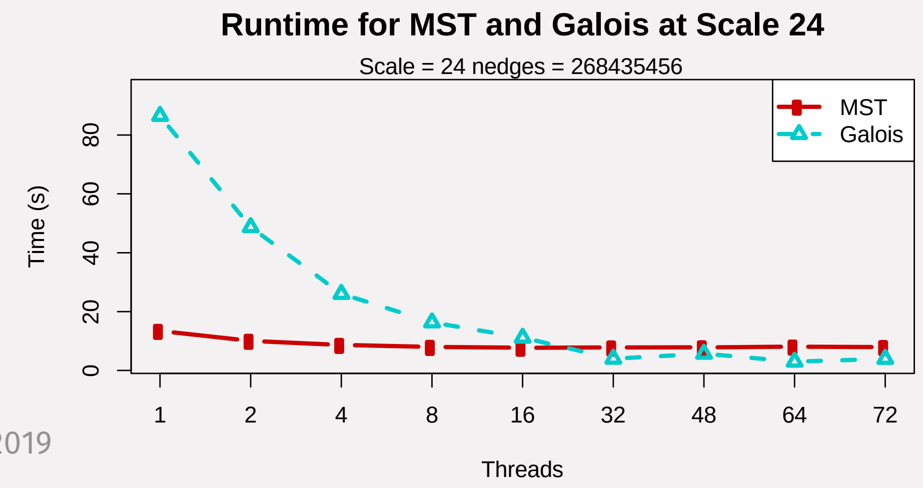
Comparison to SSSP implementation in Galois v2.2.1

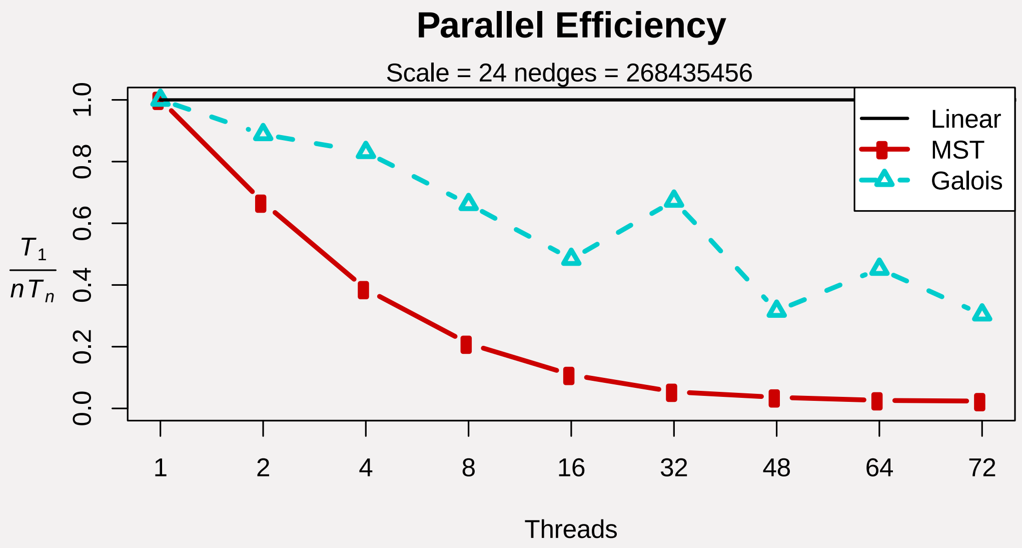
Synthetic RMAT-G (skewed degree distribution) and RMAT-ER (normal degree distribution) graphs, three real-world graphs from SNAP

Updating MST

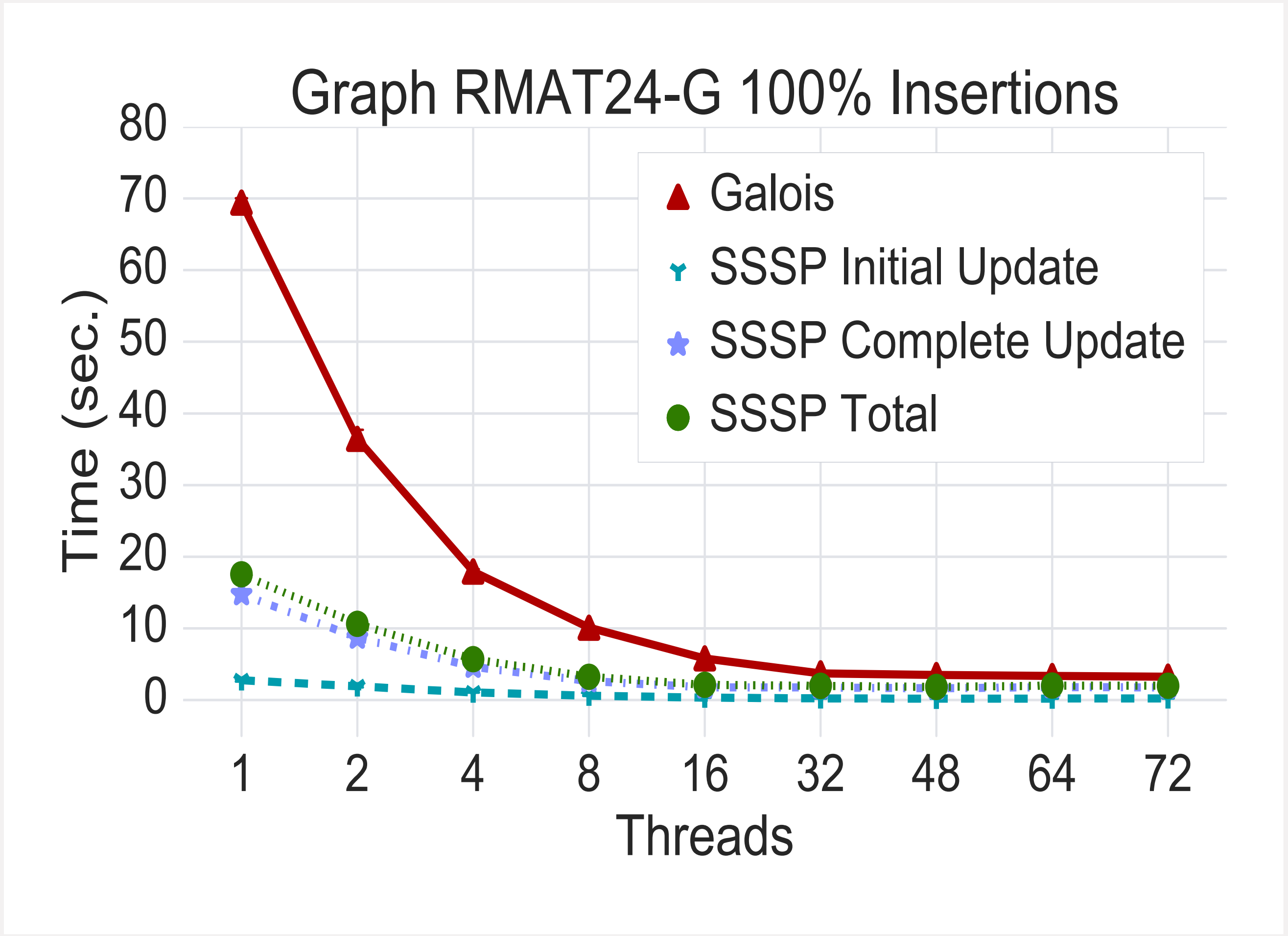






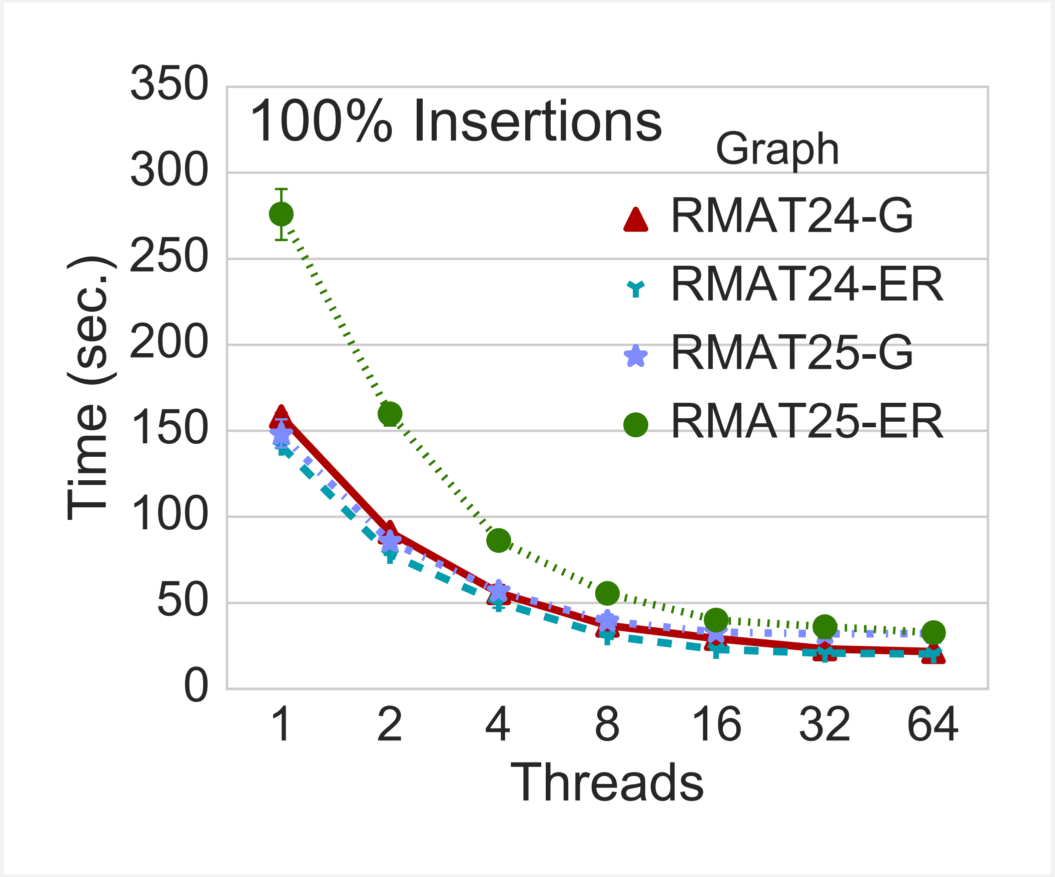


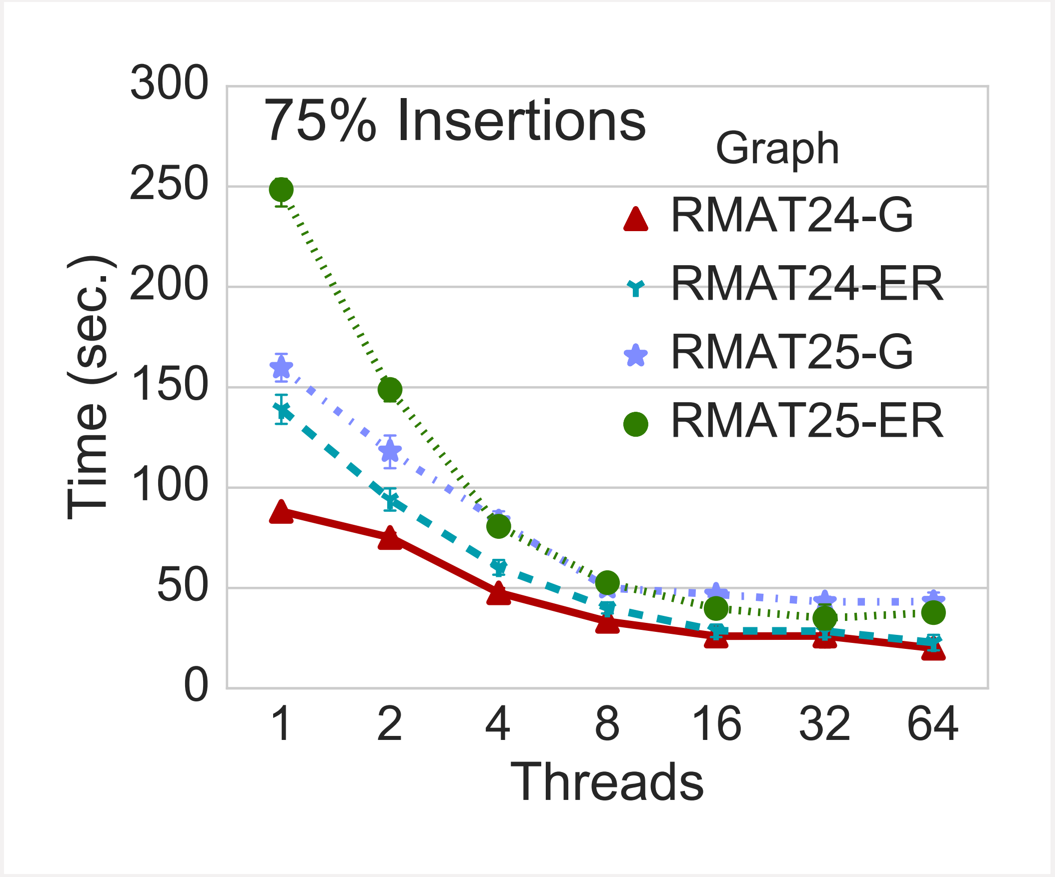
Comparison to recomputation-based approach for SSSP



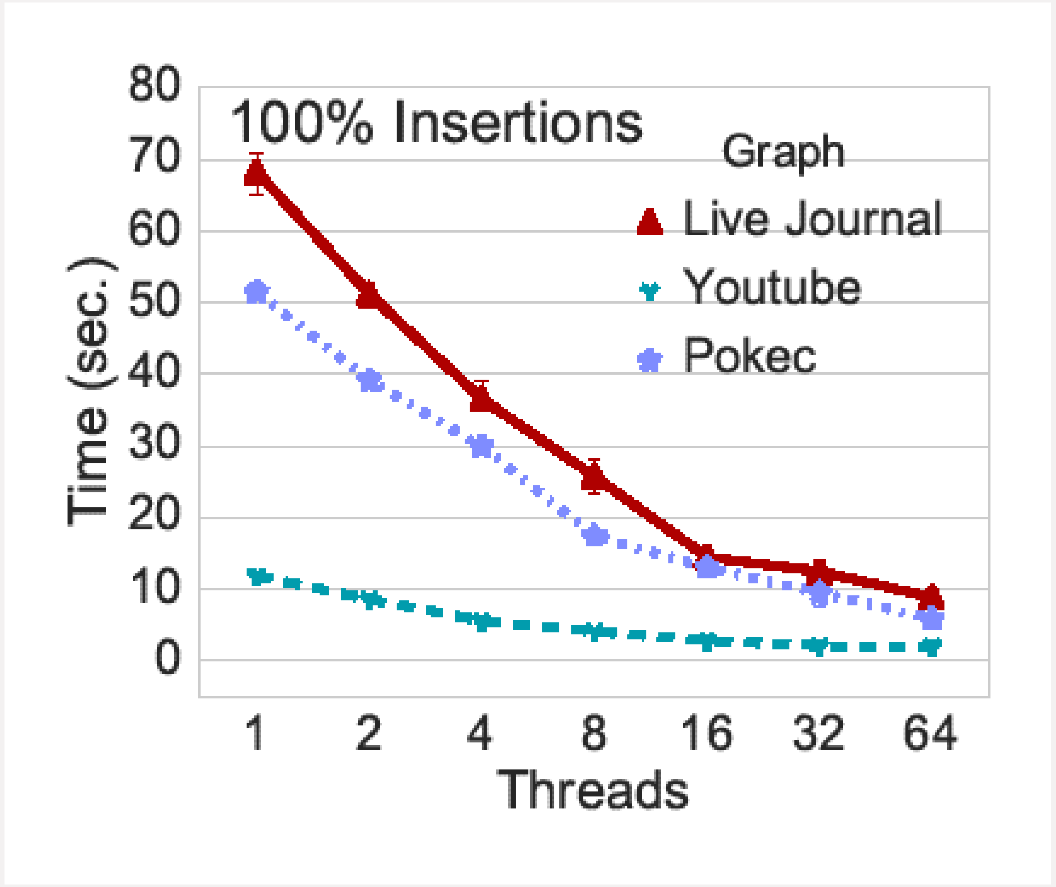
New algorithm is up to 4X faster.

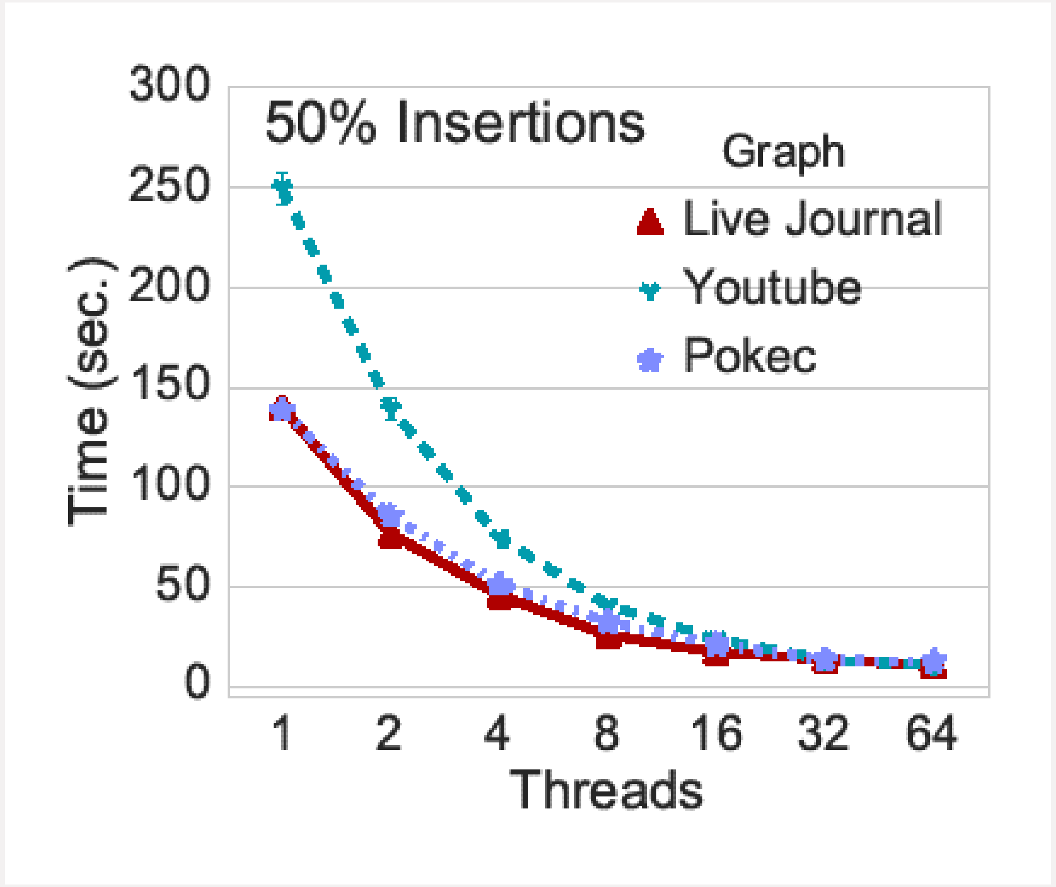
Strong scaling (synthetic graphs) for SSSP



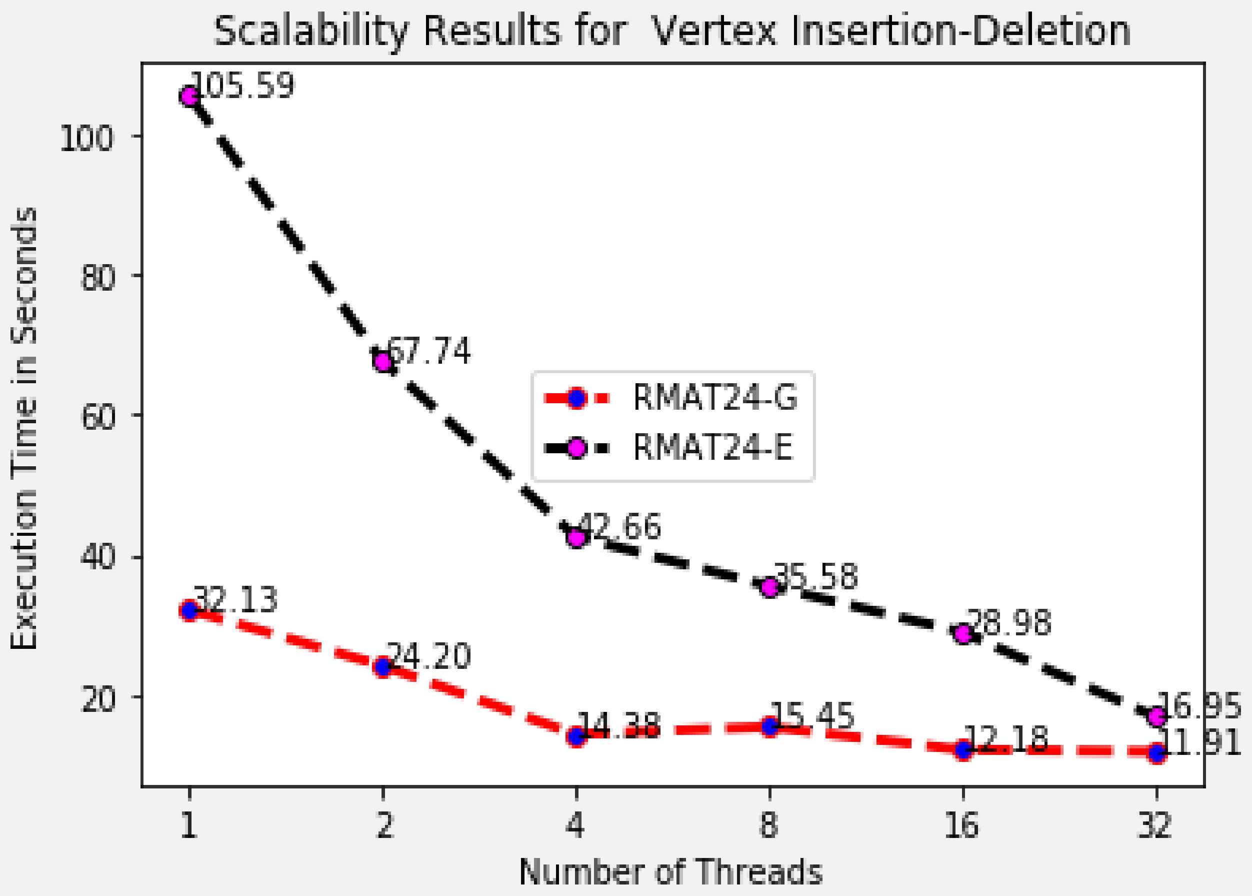


Strong scaling (real-world graphs) for SSSP





Strong scaling (vertex insertion/deletion)



Performance and Scalability

Updating algorithm faster than recomputing for lower number of threads

Recomputing algorithm more scalable

Reasons for low scalability

- Scalability deteriorates at the update phase
- Synchronization between threads updating overlapping subgraphs
- The set of changed edges not known a-priori to identify these overlaps
- In contrast, recomputing algorithms have knowledge of the entire graph

Correctness proof

SSSP :-

Lemma:- Tree obtained by the proposed approach will be a valid SSSP tree for the updated Graph G.

MST :-

Lemma:- After updating MST, sum of edge weights of the updated graph is equal to the sum of the weights of the active key edges in the rooted tree.

Connected Components :-

Lemma:- After updating connected components, if there exists a path between two vertices u and v in updated graph, there also exist a path between those two vertices in the updated rooted tree.

(Complete proof please check dissertation *)

Applications of dynamic SSSP?

Many applications

- Maps and GPS
- Internet routing
- Path planning for robots
- Discrete event simulations
- Centrality analysis in complex networks

Contributions to Date

- A new two-step parallel algorithm for updating the SSSP, MST & Connected Components
  - optimizations to improve scalability
  - optimizations to reduce redundant/wasteful computation
- Correctness proof
- Empirical evaluation to demonstrate speedup over recomputing SSSP & MST from scratch

Research Questions

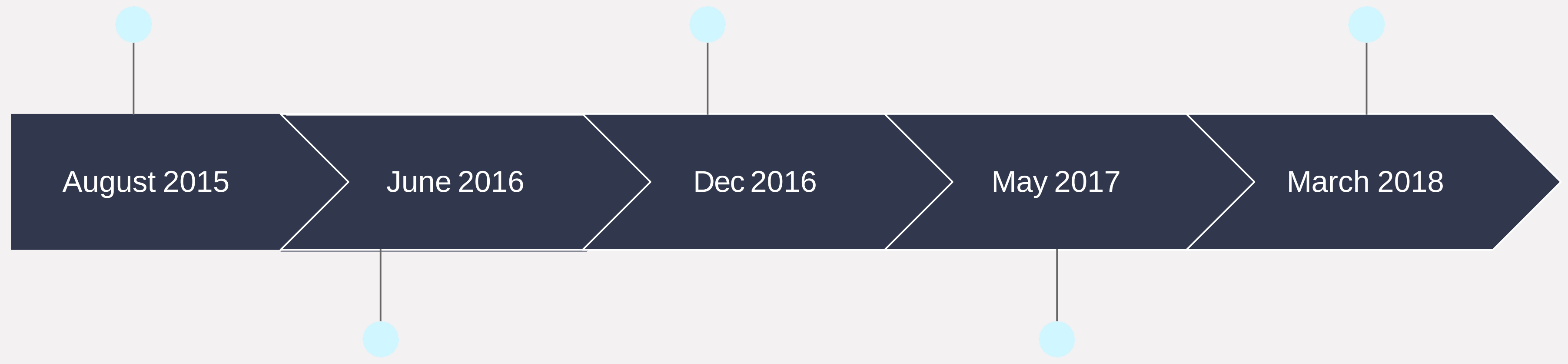
How to improve scalability of updating

- Hybrid: move to recomputing once the edges that affect the property are known
- Asynchrony: increase length of asynchrony during updating
- Hot Spots: identify subgraphs that will be most affected by change, store them separately
- Scheduling Changes: if changes are known, can we schedule them for improved performance

How to measure scalability of updating

- Time: fundamental measure, but affected more by type and amount of changes than size of graph
- TEPS: traversed edges per second, but we are trying to reduce the number of edges traversed
- Others: number of edges updated per change

Timeline



Brainstorming on how to analyze dynamic networks

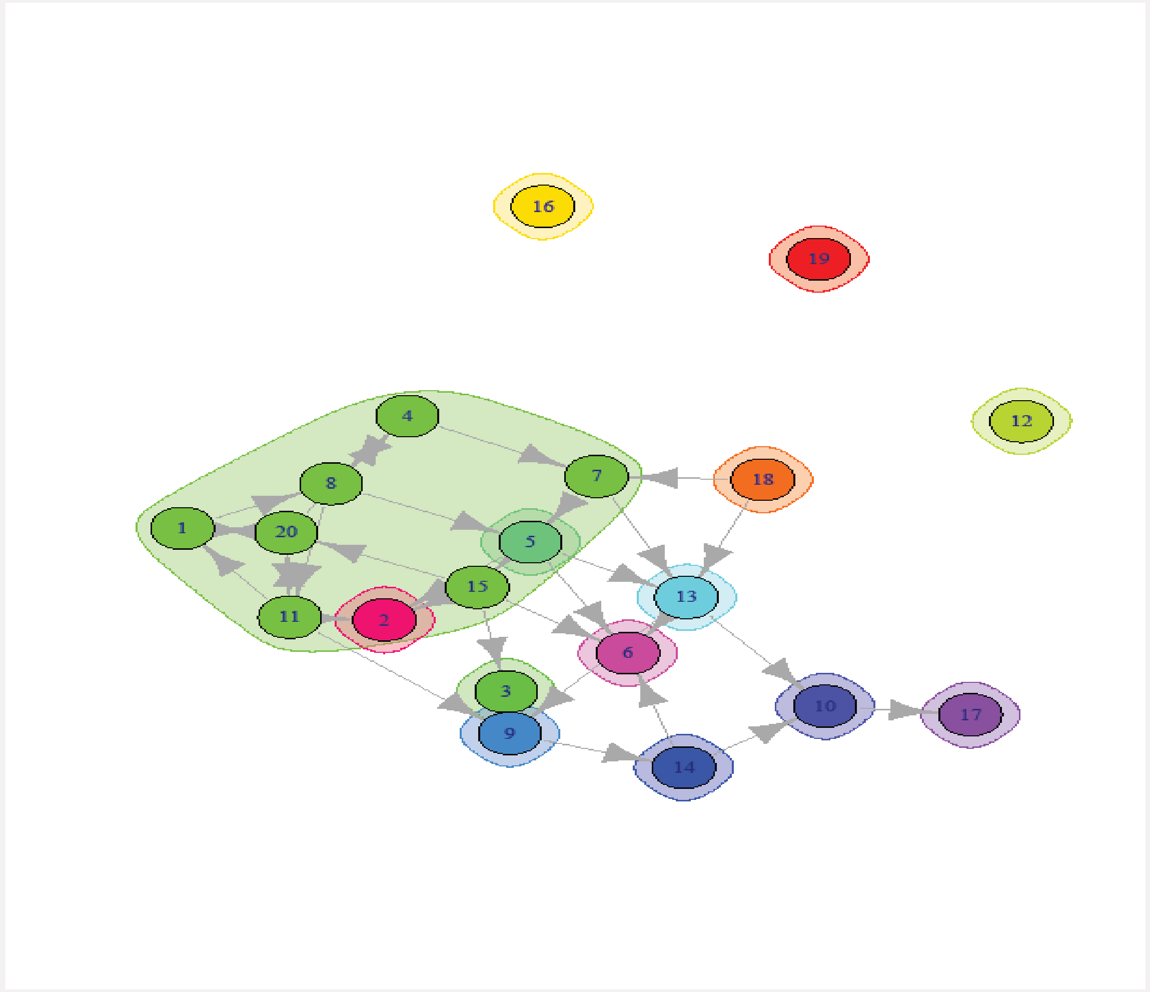
Solved Connected Components. Presented in IPDPS 2016

Brainstorming on how to update MSTand SSSP.

Solved MST, Presented in SIAM CS17, and IPDPS (Ph.D.forum) 2017. IEEE Transactions on Big data Journal (Accepted @ 06/2018)

Solved SSSP, SC18(Accepted @ HIPC 18)

Research Plan- (Task 1, Updating Strongly Connected Components (SCC))

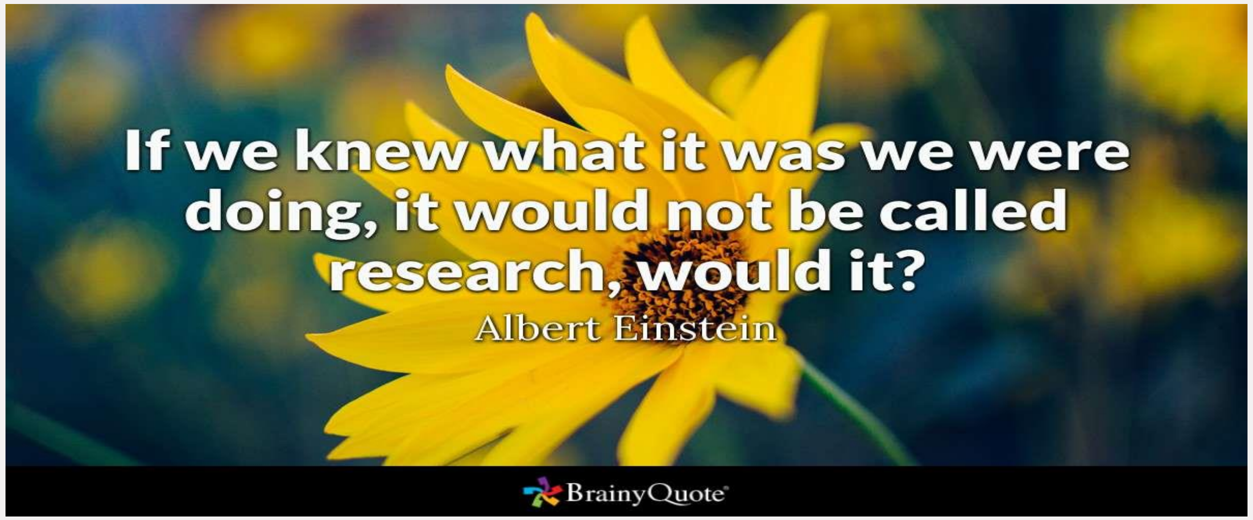


- **Definition:-** Group of vertices in a directed network such that there is a path between all pairs of vertices
- **Research problem** :- Update SCC as network changes i.e. Dynamic Networks
- **Proposal:-** Extend the shared memory implementation of updating SSSP & MST to SCC

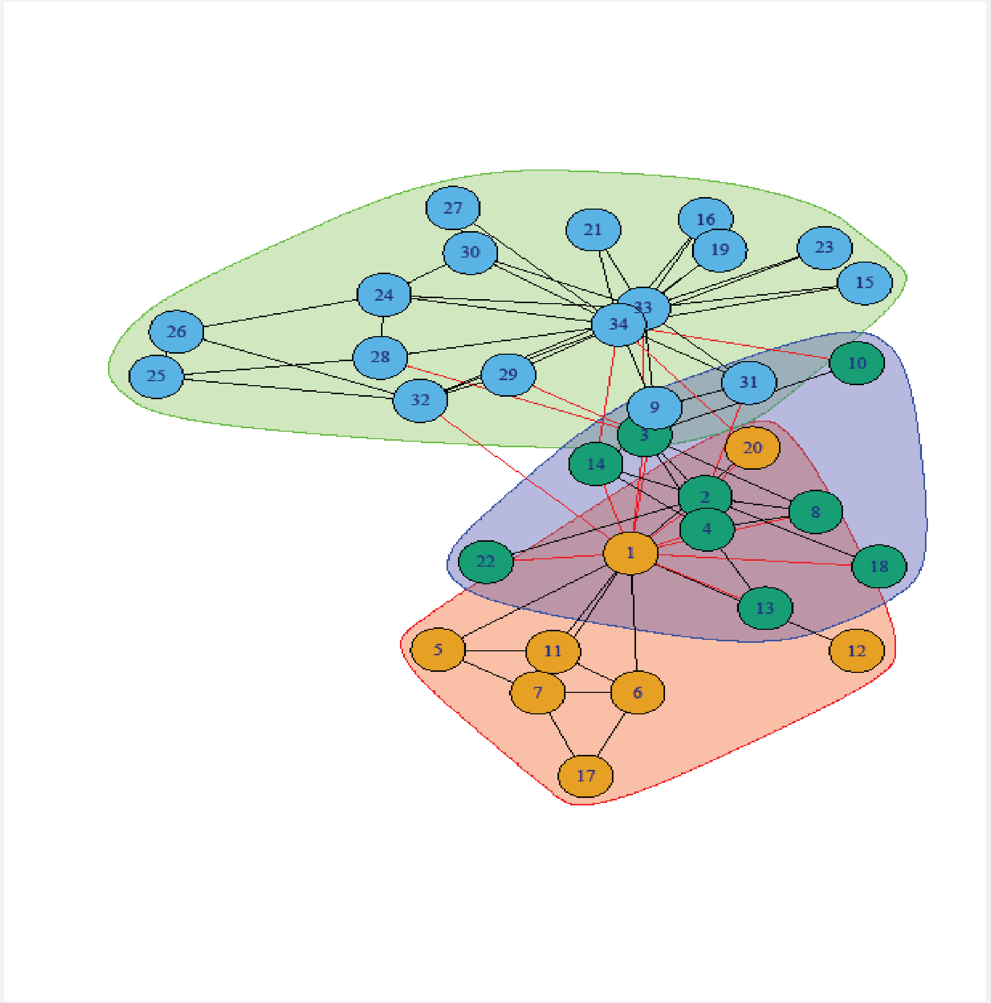
Research Plan- (Task 2, Extending to GPUs)

- Research problem :- Shared memory implementations of updating SSSP, & MST on dynamic networks show limited scalability as thread count increases. GPUs can be a good candidate for updating the network properties to achieve better scalability

- Proposal :-Implement GPU version of updating SSSP on dynamic networks


If we knew what it was we were doing, it would not be called research, would it?
Albert Einstein
BrainyQuote

Research Plan- (Task 3, Shared Memory Implementation of Overlapping Communities)



- **Definition**:- Group of vertices which are strongly connected internally and sparsely connected externally
- **Research problem** :- Scalable implementation of algorithm to detect overlapping communities and measure the quality
- **Proposal**:- Implement shared memory approach for GenPerm

Research Plan- (Task 4, Implementing a shared memory hybrid-Chordal filter)

- Research problem :- Networks are generally loaded with noise, which results in signal corruption. Network filters can reduce noise and size while preserving the significant network structure

- Proposal :- Previous work**sequential hybrid filter showed great results, implementing a multithreaded hybrid-chordal filter can improve performance

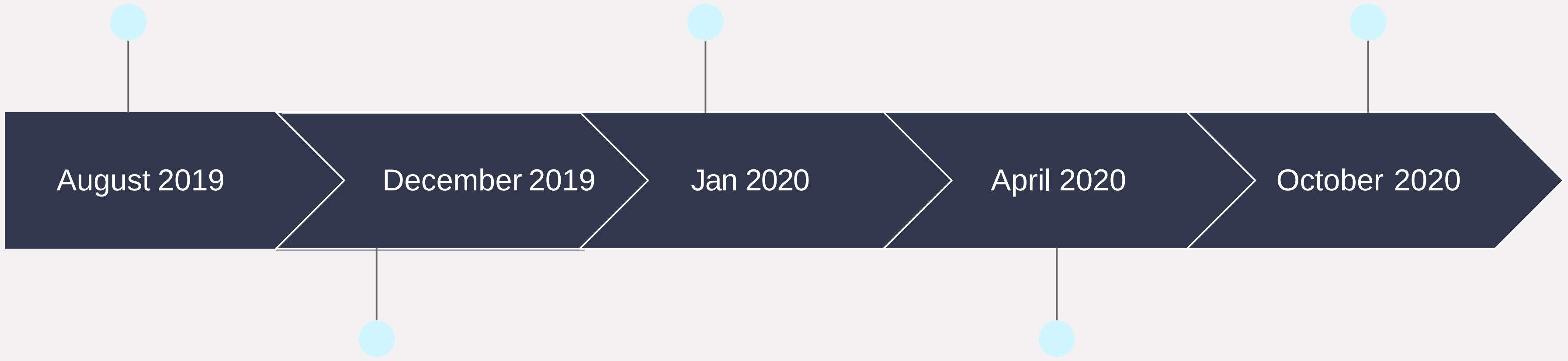
** K. Dempsey, T. Chen, S. Srinivasan, S. Bhowmick and H. Ali, "A structure-preserving hybrid-chordal filter for sampling in correlation networks," *2013 International Conference on High Performance Computing & Simulation (HPCS)*, Helsinki, 2013, pp. 243-250. **

Timeline for Proposed Dissertation Research Plan

Implement shared memory implementation of Updating SCC

Implement shared/distributed Memory Implementation of Overlapping Communities

Update dissertation with the outcomes of all the proposed research, and complete final dissertation exit requirements



Compare results for SCC with other state of the art SCC implementation

Performance comparison with other Overlapping Community Detection algorithm and Start working on GPUs & Hybrid-Chordal filter

Conclusions

New shared-memory algorithm for updating SSSP & MST in dynamic networks

Performance results demonstrate up to a 4X performance improvement over a parallel recomputation-based SSSP code

Plan to extend the general approach to update SCC

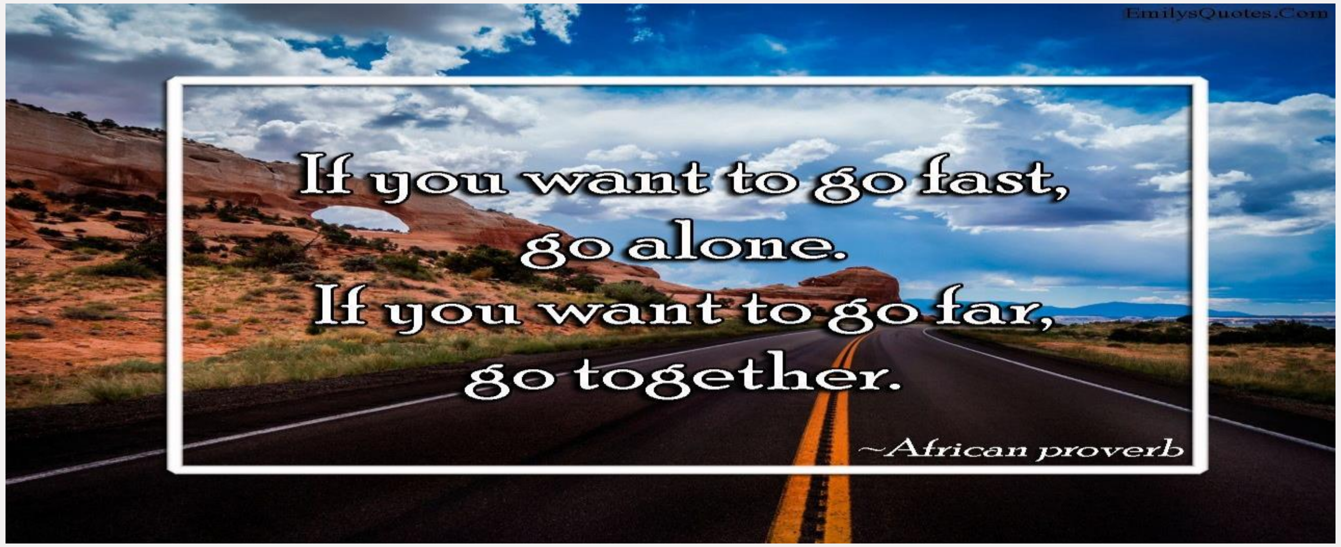
Future GPU and distributed-memory implementations

Acknowledgments & Collaborators

Dr. **Boyana Norris,**
University of Oregon

Dr. **Sajal Das,**
Missouri S&T

Thank you!



Questions?

Corresponding author: Sriram Srinivasan, sriramsrinivas@unomaha.edu

github.com/SriramSrinivas/SriramDissertation