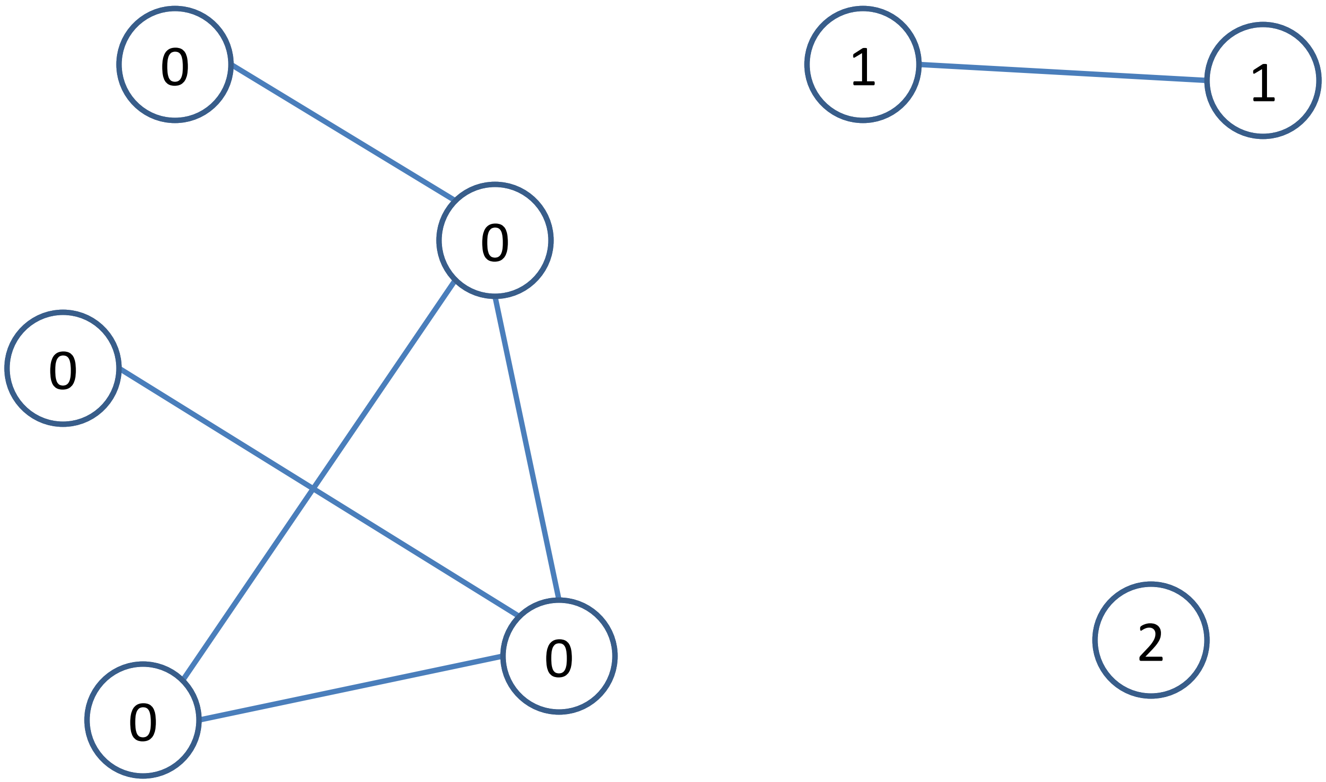
A Simple and Practical Linear-Work Parallel Algorithm for Connectivity

Julian Shun, Laxman Dhulipala, and Guy Blelloch

Connected Component Labeling



Connected Component Labeling

- What are some simple algorithms?
  - Depth-first search
    - Linear work/span
    - Versions of DFS that are parallel are not work-efficient
  - Breadth-first search
    - Linear work
    - Parallelism limited by graph diameter
    - Polylogarithmic span version not work-efficient
  - Spanning forest
    - Good parallelism
    - Practical parallel implementations not linear work

Connected Component Labeling

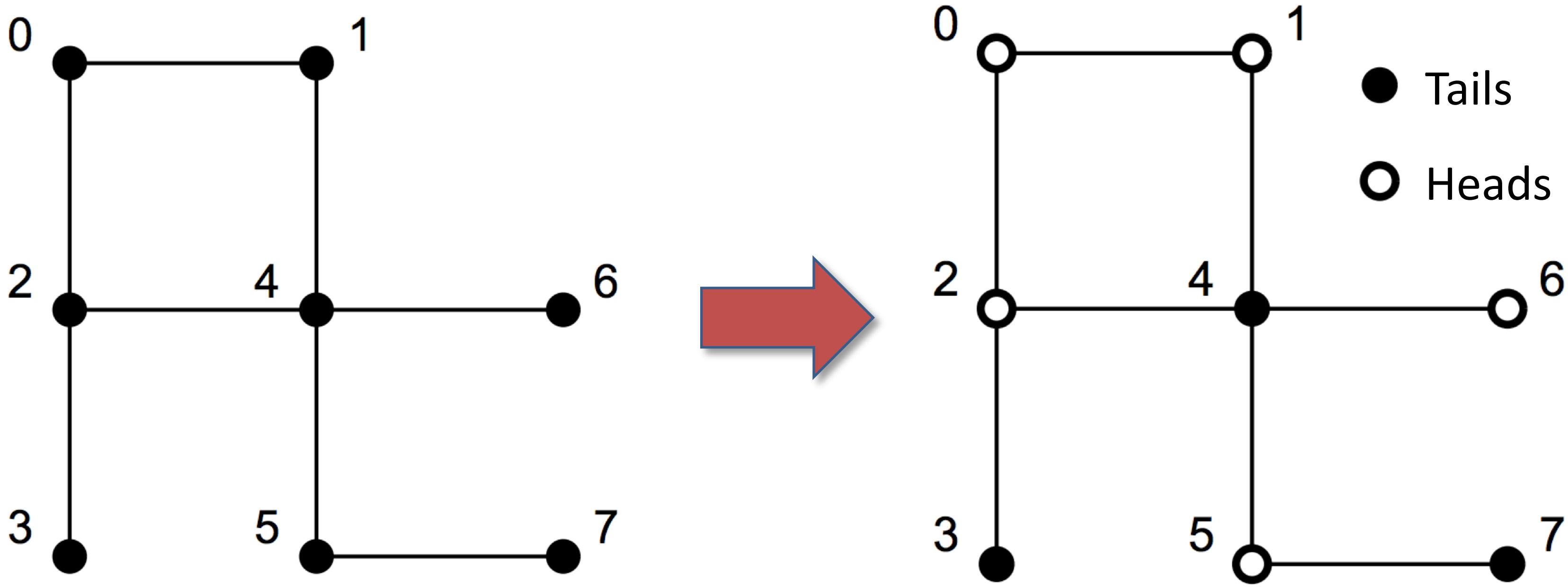
- Parallel (polylogarithmic span) algorithms
  - Shiloach and Vishkin, Awerbuck and Shiloach
    - Combines (contracts) vertices in each iteration
    - O(m log n) work, O(log n) span
  - Reif, Phillips
    - Uses randomization to simplify contraction algorithms
    - O(m log n) work and O(log n) span w.h.p.
    - O(log n) rounds but don't guarantee a constant fraction of edges removed
  - O(m) work algorithms
    - Gazit '91, Halperin/Zwick '96, Cole et al. '96, Poon/Ramachandran '97, Pettie/Ramachandran '02
    - Quite complicated. No one has implemented these

Our Contributions

- Practical parallel connectivity algorithm with linear work and polylogarithmic span
- Experimental evaluation: competitive with existing parallel implementations (that are not linear-work and polylogarithmic span)

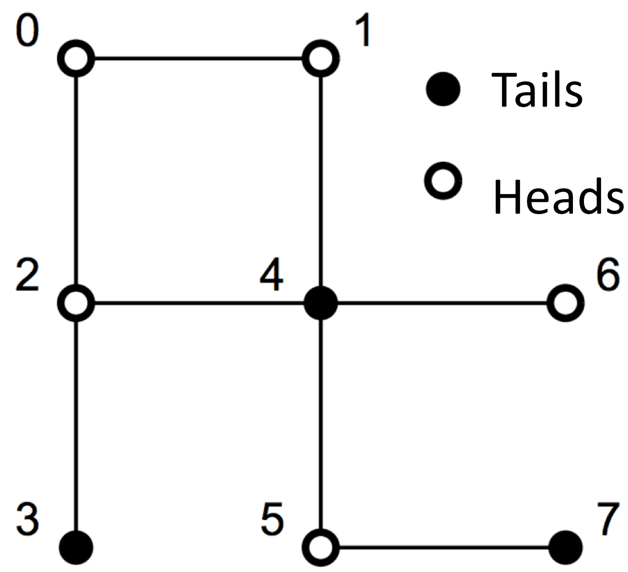
Review: Random Mate

- Idea: Form a set of non-overlapping star subgraphs and contract them
- Each vertex flips a coin. For each Heads vertex, pick an arbitrary Tails neighbor (if there is one) and point to it

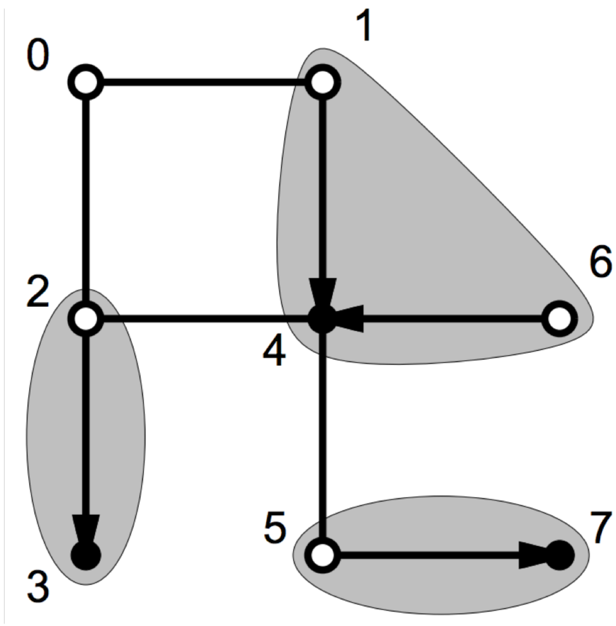


Source: "Parallel Algorithms" by Guy E. Blelloch and Bruce M. Maggs

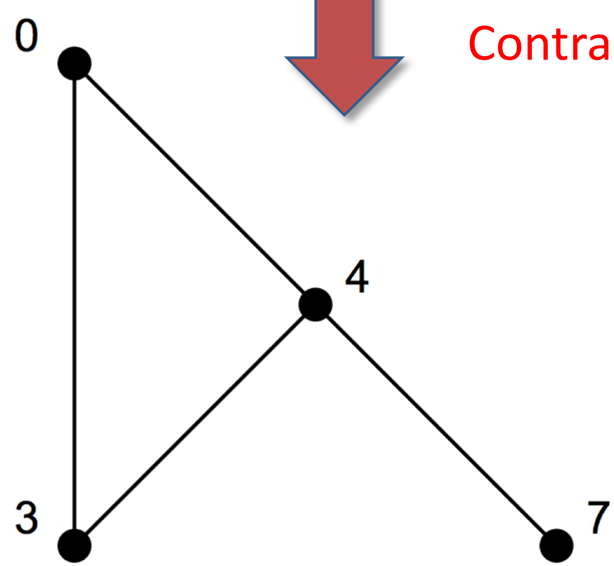
Review: Random Mate



Form stars



Contract



Repeat until each component has a single vertex

Expand vertices back in reverse order with label of neighbor

Source: "Parallel Algorithms" by Guy E. Blelloch and Bruce M. Maggs

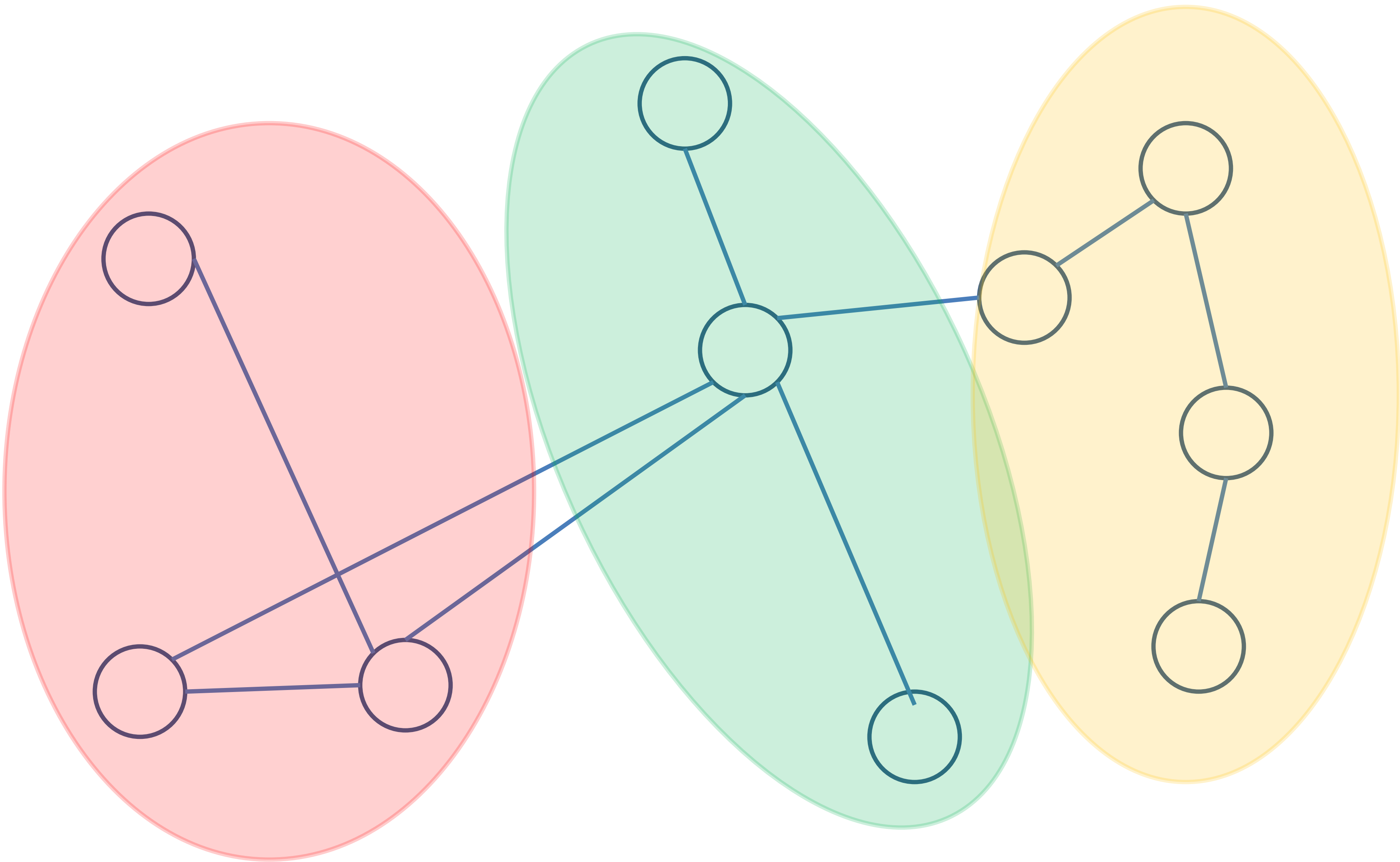
Review: Random Mate Algorithm

```
CC_Random_Mate(L, E)
 if(|E| = 0) Return L //base case
 else
    1.  Flip coins for all vertices
    2.  For v where coin(v)=Heads, hook to arbitrary Tails neighbor w and set
        L(v) = w
    3.  E' = { (L(u),L(v)) | (u,v) ∈ E and L(u) ≠ L(v) }
    4.  L' = CC_Random_Mate(L, E')
    5.  For v where coin(v)=Heads, set L'(v) = L'(w) where w is the Tails neighbor
        that v hooked to in Step 2
    6.  Return L'
```

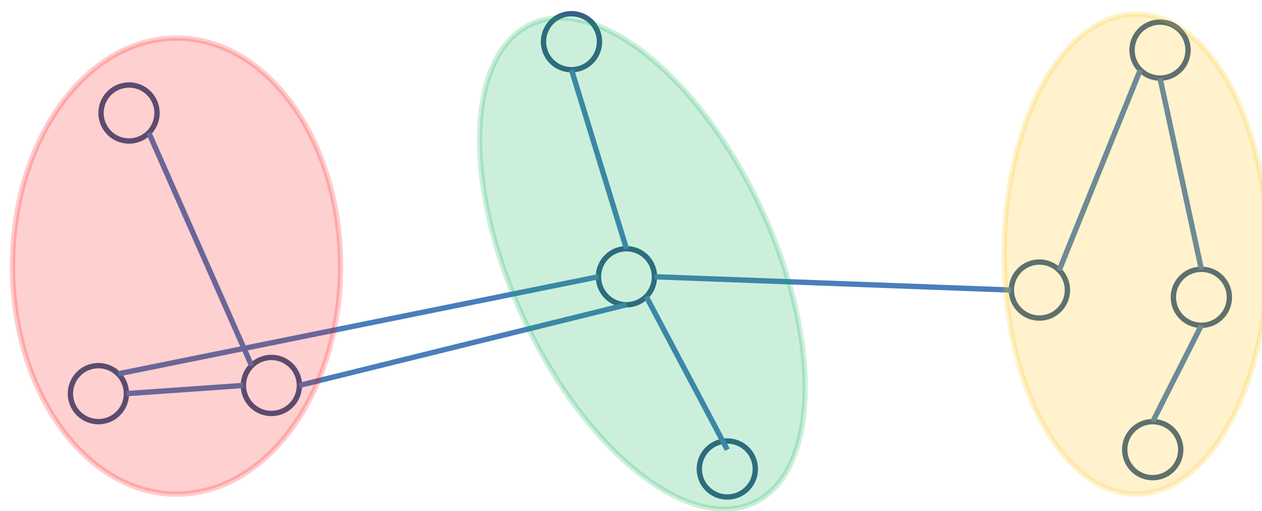
- Each iteration requires O(m+n) work and O(1) span
  - Assumes we do not pack vertices and edges
- Each iteration eliminates 1/4 of the vertices in expectation → O(log n) rounds w.h.p.

W = O(m log n) w.h.p. S = O(log n) w.h.p.

Low diameter decomposition



Low diameter decomposition

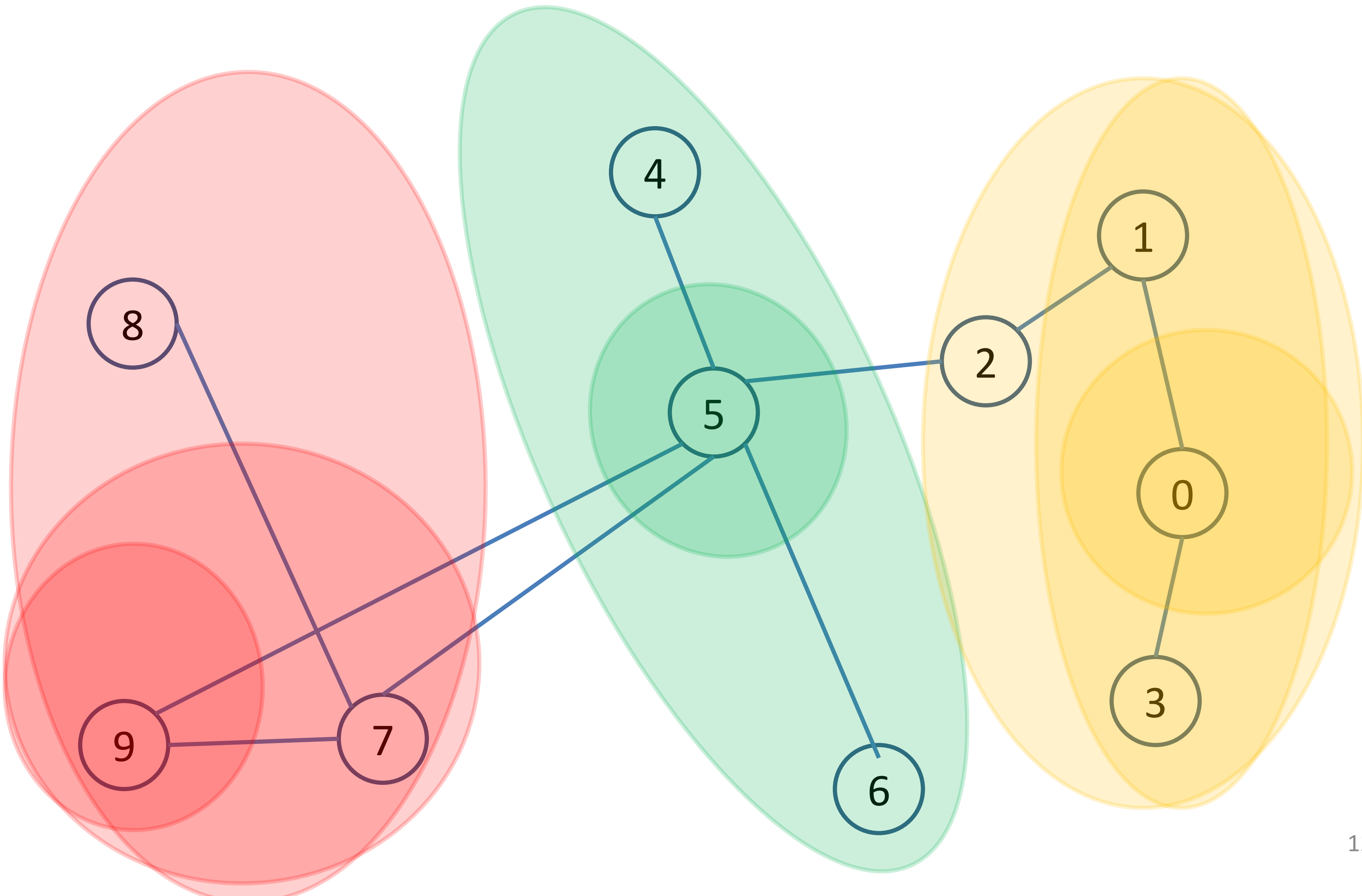


- (β,d)-decomposition ($0 < \beta < 1$) partitions V into $V_1,\ldots,V_k$ such that
  - The shortest path between any two vertices in a partition is at most d
  - The number of inter-partition edges is at most βm
- Used in linear system solvers and metric embeddings

Low diameter decomposition

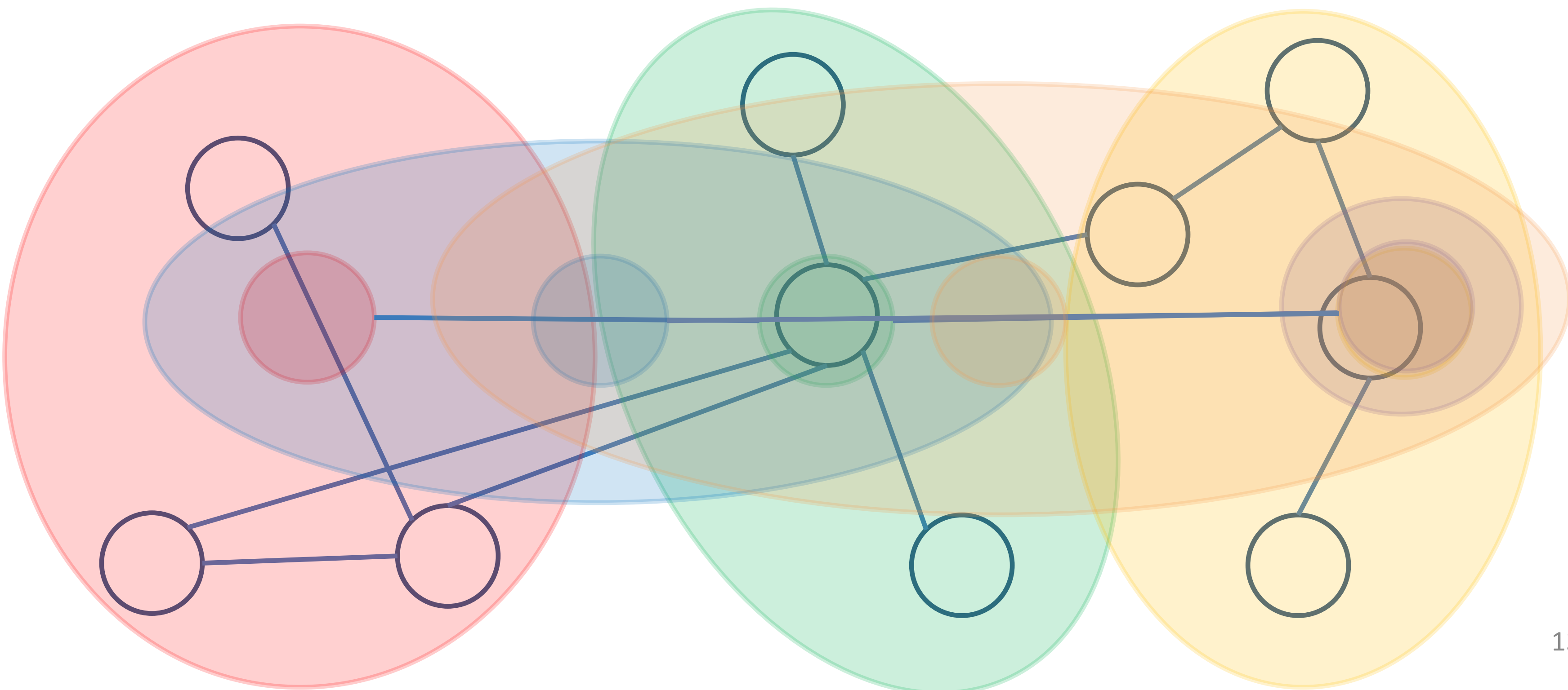
- A (β, O(log n / β))-decomposition can be computed in O(m) expected work and $O(\log^2 n / \beta)$ span w.h.p. [Miller et al. 2013]
  - Start breadth-first searches from vertices with exponentially-distributed (parameter β) start times
    - Each BFS creates a partition containing the source and all vertices explored
    - A BFS does not explore vertices already visited by another BFS
    - All vertices will have started BFS or been explored by time O(log n / β)
  - BFS's are work-efficient and terminate in O(log n / β) iterations.
    - Each iteration requires O(log n) span.
  - Bounding number of inter-partition edges:
    - An edge is inter-partition if the first two BFS's that reach it do so within a one time step of each other
    - Probability that this happens is at most β due to properties of exponential distribution
    - Linearity of expectations gives at most βm edges cut

Low diameter decomposition example



Our Connectivity Algorithm

- Compute a (β, O(log n / β))-decomposition
- Contract each partition into a single vertex
- Recurse



Our Connectivity Algorithm

- Compute a (β, O(log n / β))-decomposition
- Contract each partition into a single vertex
- Recurse

<u>Analysis for β=1/2</u>

- Assume contraction can be done in linear work and in O(log n) span
- m/2 edges remain after each round in expectation
  - Work = O(m) + O(m/2) + … = O(m) in expectation
- O(log n) levels of recursion suffice w.h.p.
  - Span = O(log n) * $O(\log^2 n / \beta) = O(\log^3 n)$ w.h.p.

Contraction

- Contraction can be done in O(log n) span with bookkeeping and parallel prefix sums
  - Intra-partition edges are packed out in O(m) work and O(log n) span
  - Prefix sums: relabel vertices to smaller range
  - Duplicate edges removed using parallel hashing in O(m) work and O(log n) span
    - Not needed theoretically

Improving span

- Each round of BFS can be implemented in O(log* n) span w.h.p. using approximate prefix sum and compaction [Gil-Matias-Vishkin ‘91, Goodrich-Matias-Vishkin ‘94]
  – Improves span of low diameter decomposition to O(log n log* n)
- Recurse for O(log log n) rounds
  – Left with O(m/log n) edges
  – Switch to O(m log n) work, O(log n) span algorithm
- Result: Linear work algorithm with O(log n log log n log* n) span w.h.p.

Low diameter decomposition variants

- Resolving conflicts among BFS’s
  - Decomp-min: breaks ties deterministically
    - Miller et al. showed this produces (β, $O(\log n/\beta)$)-decomposition
    - Uses write-with-min (via compare-and-swap)
    - Requires two phases
  - Decomp-arb: breaks ties arbitrarily
    - We prove (2β, $O(\log n/\beta)$)-decomposition
    - Uses compare-and-swap
    - Requires just a single phase
  - Decomp-arb-hybrid: uses direction-optimizing BFS
    - This is the fastest one and used in the following experimental results

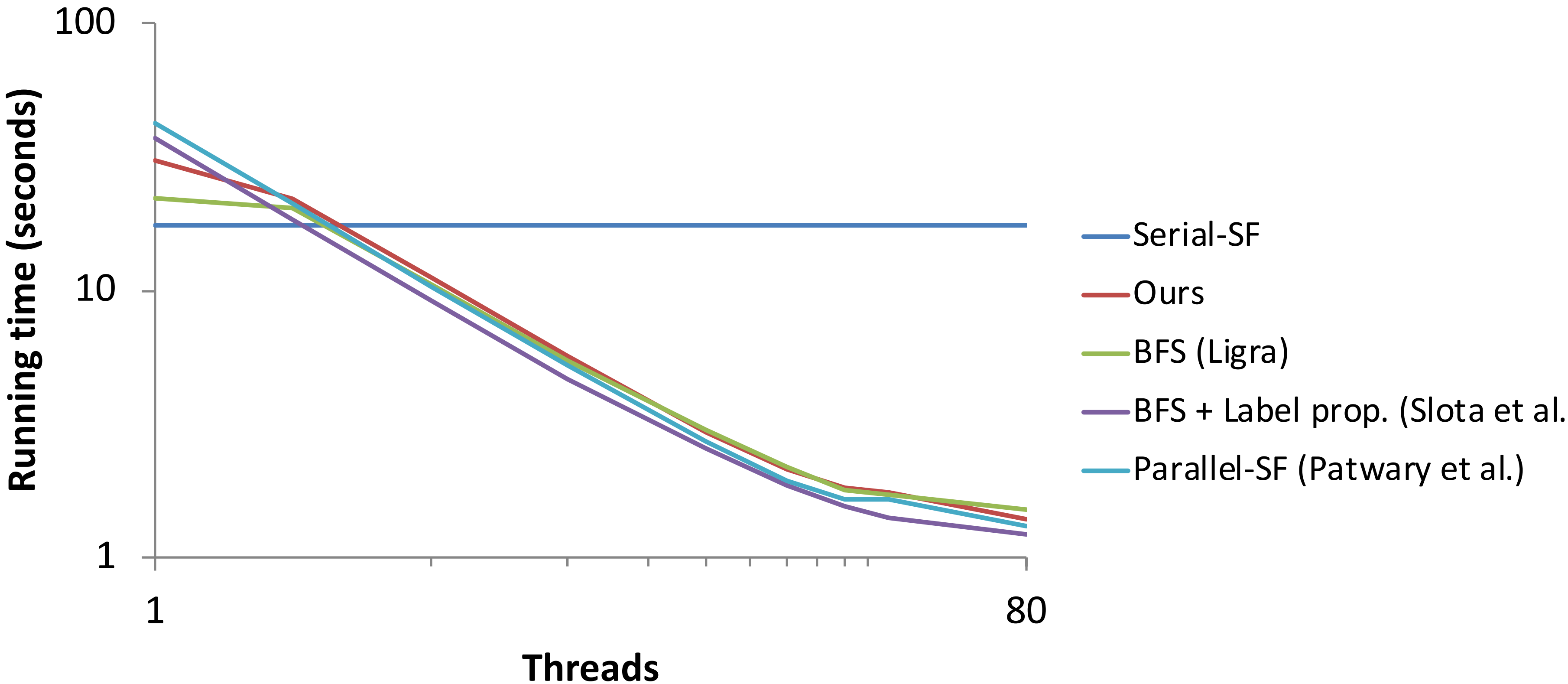
Experiments

- 40-core (with 2-way hyper-threading) Intel Nehalem machine
- Implemented in Cilk Plus
- 3 different implementations, but only showing best one
- Real-world and artificial graphs

Compare to existing implementations

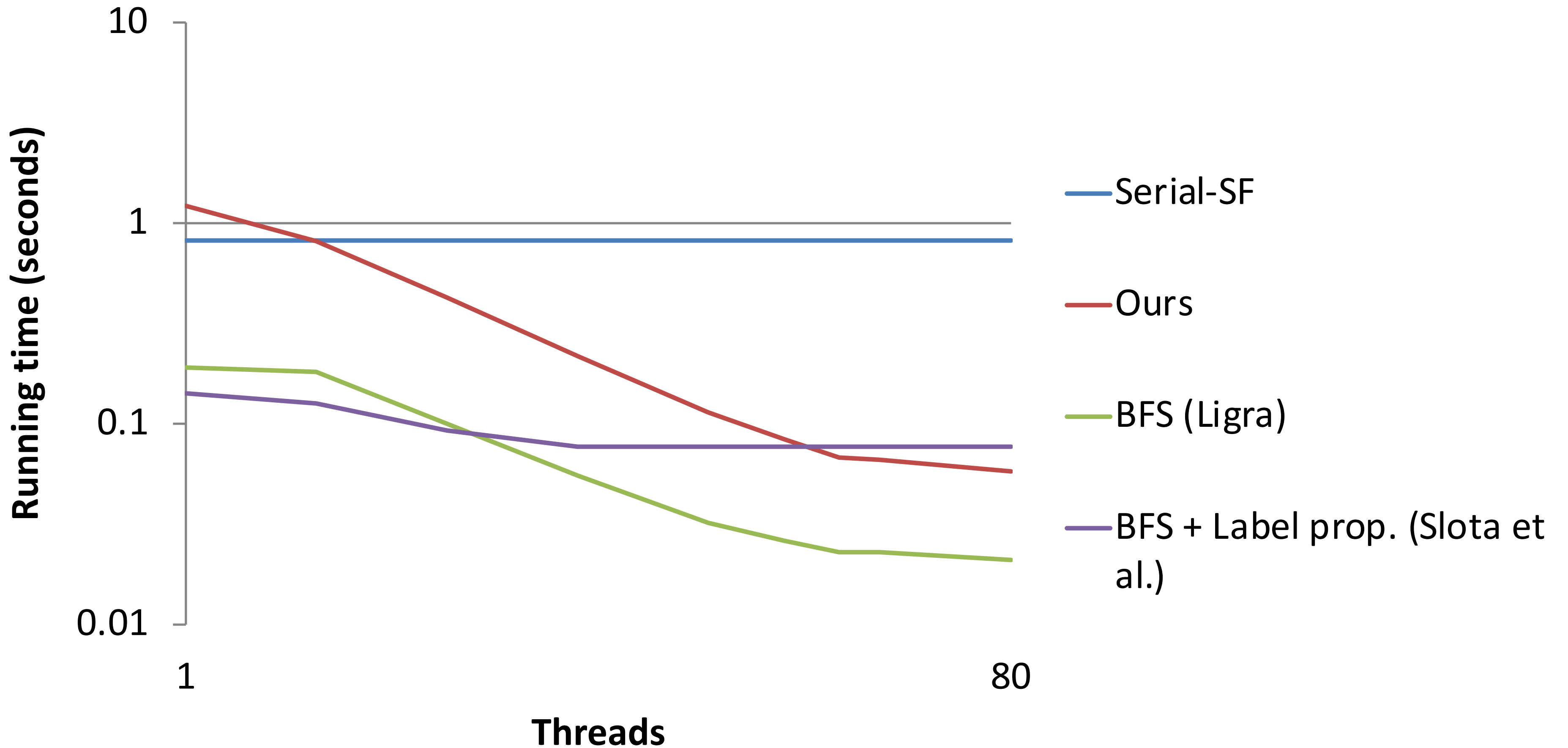
- Existing implementations
  - Sequential spanning forest
  - Parallel spanning forest (Problem Based Benchmark Suite)
  - Parallel spanning forest (Patwary et al.)
  - Parallel BFS (Ligra)
  - Parallel BFS + Label propagation (Slota et al.)
- None provably linear work and polylog span

3D grid graph ($n = 10^8$, $m = 3x10^8$)



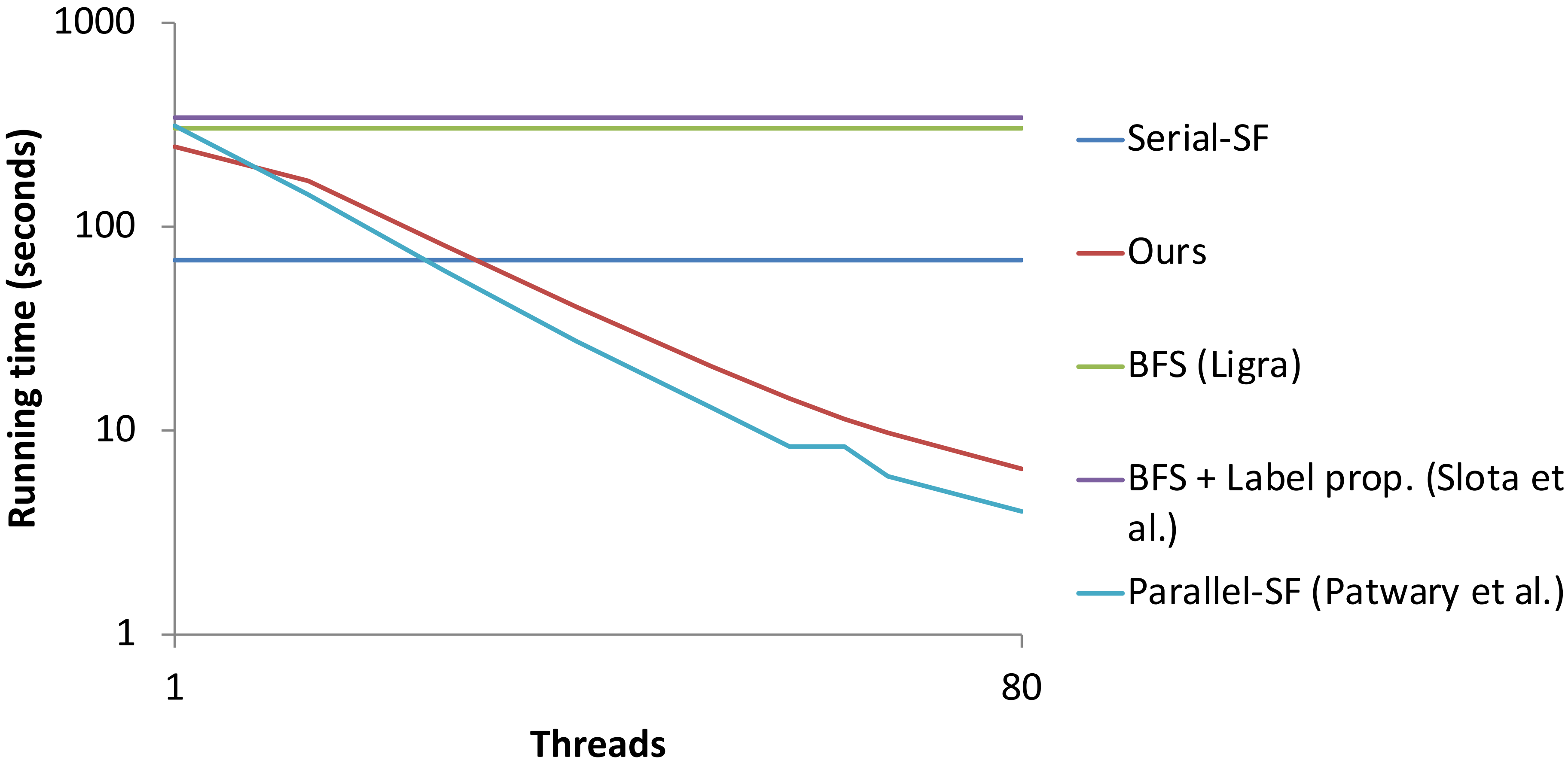
- Competitive with other implementations

com-Orkut graph ($n \approx 3x10^6$, $m \approx 10^8$)



- Fastest implementation uses single BFS

Line graph ($n = 5x10^8$, $m = 5x10^8$)



- Algorithms based on single BFS do poorly

Our algorithm is competitive

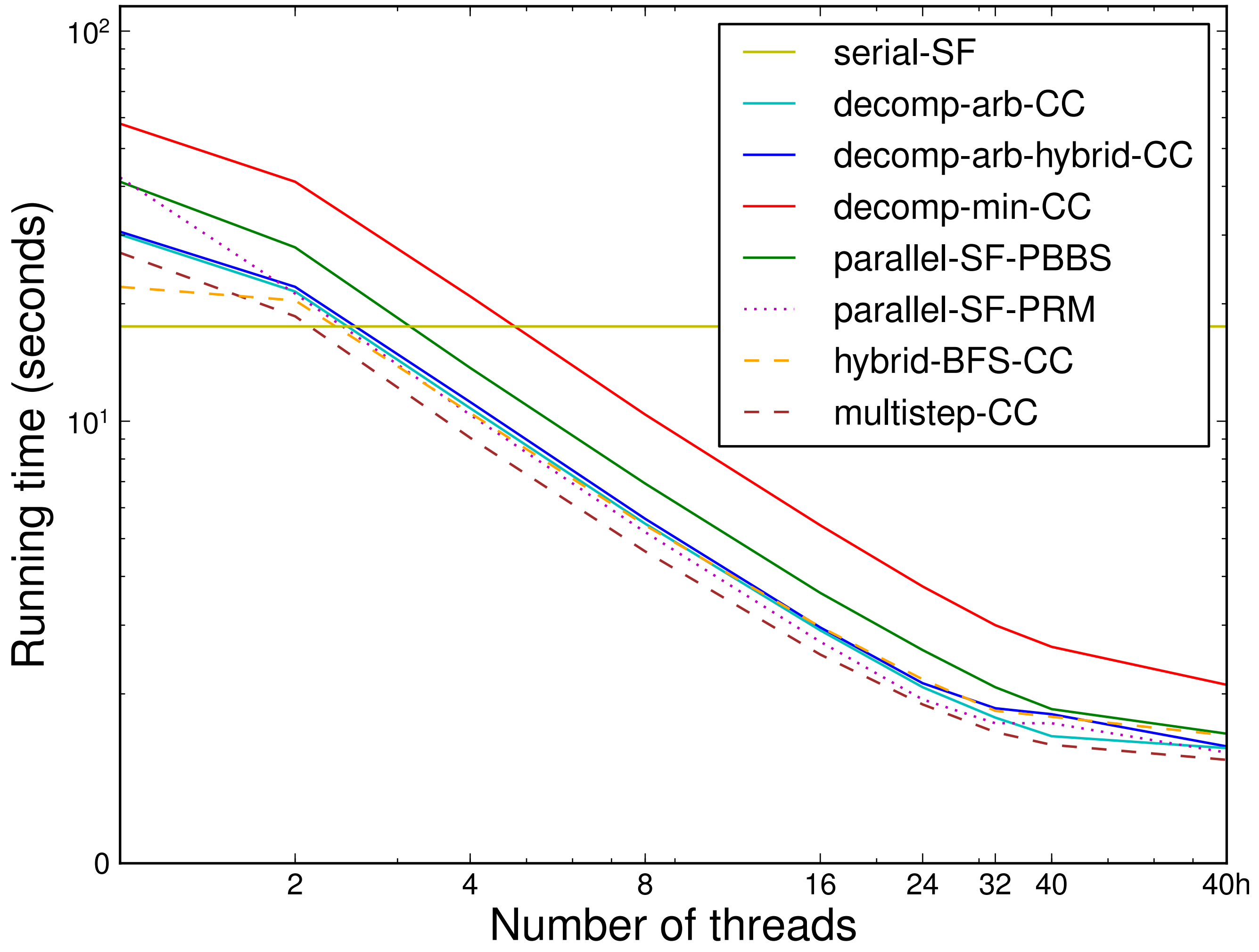
- No "worst-case" inputs
- Performance always close to the fastest implementation for any graph
  - Only at most 70% slower than spanning forest algorithms, and usually much less
  - Can be faster or slower than BFS, depending on graph diameter
- Up to 13x speedup on 40 cores relative to sequential
- 18—39x self-relative speedup

Conclusion

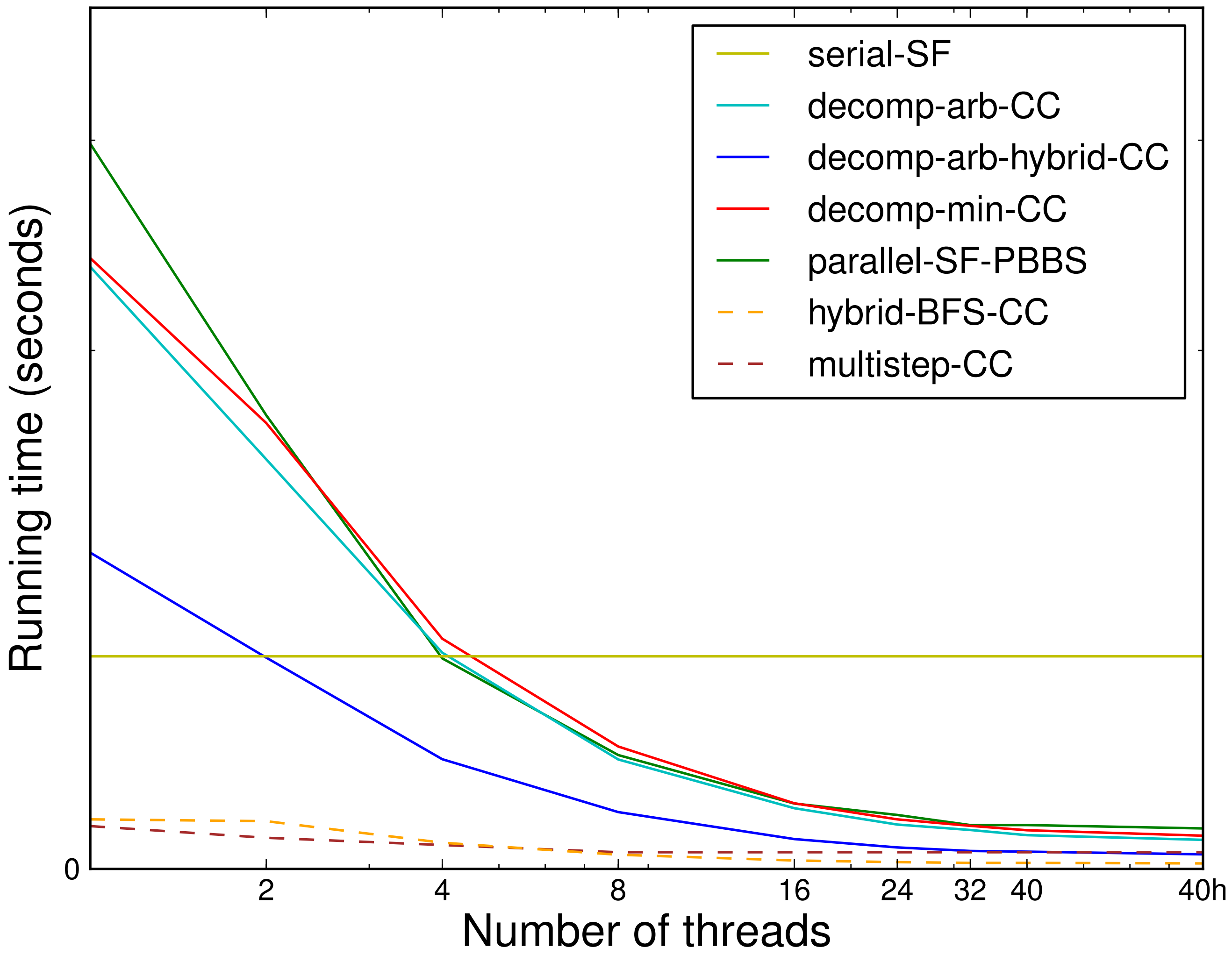
- Simple and practical linear-work, polylog-span connectivity algorithm
  – Can be easily modified to compute spanning forest
- As far as we know, first to be both practical and have linear work and polylog span
- Implementations competitive with existing parallel implementations
- Future direction: Can similar ideas give us a practical linear-work parallel algorithm for minimum spanning forest?

Extra Slides

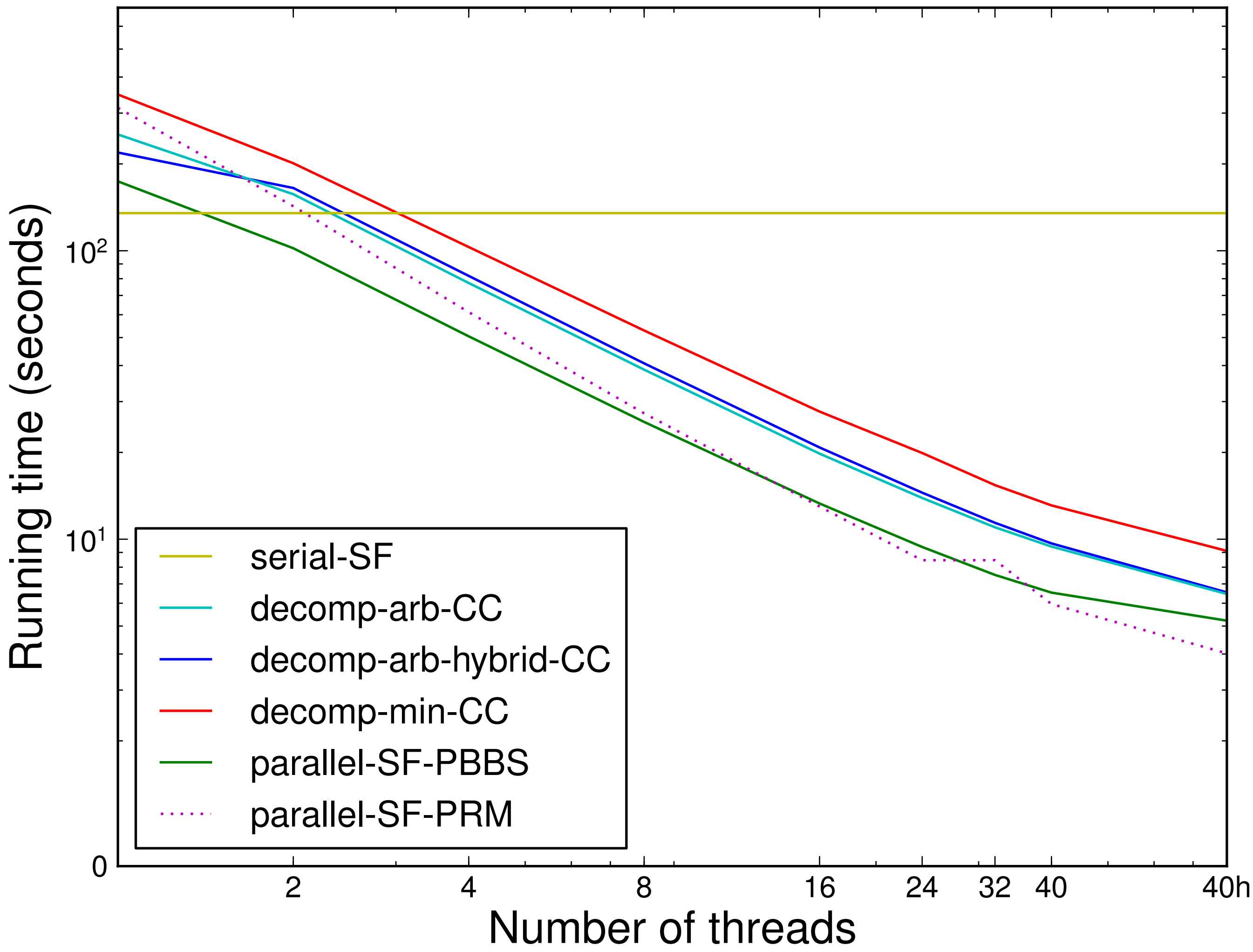
3D grid graph



com-Orkut graph

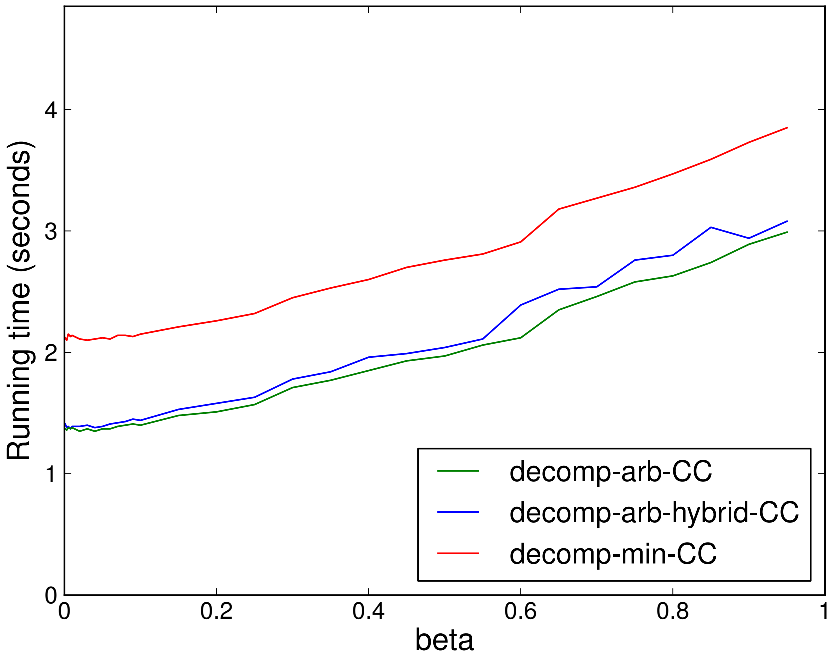


Line graph

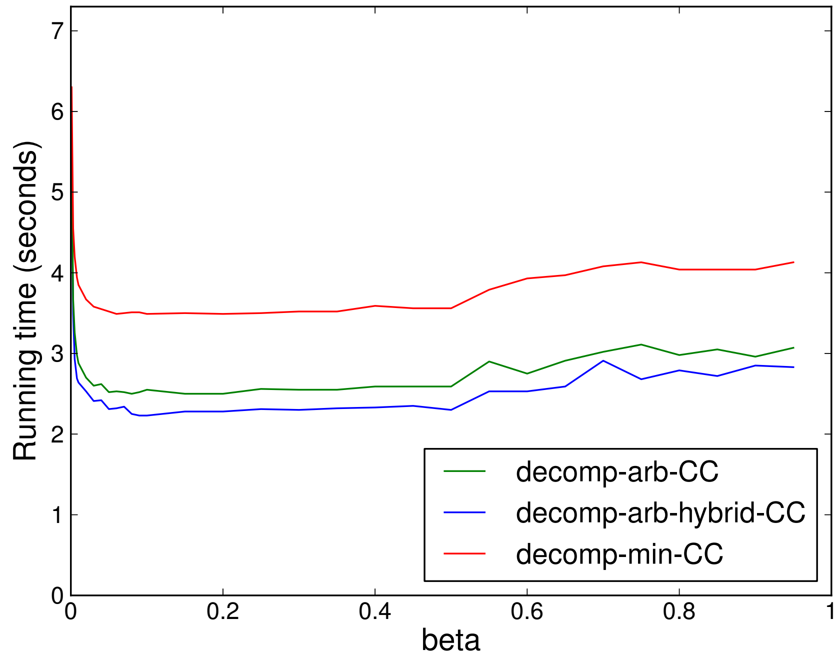

Running time (seconds)
10^2
10^1
0
2
4
8
16
24
32
40
40h
Number of threads
serial-SF
decomp-arb-CC
decomp-arb-hybrid-CC
decomp-min-CC
parallel-SF-PBBS
parallel-SF-PRM

Running time vs β

3D-grid graph



rMat graph



- Running time is similar across wide range of β