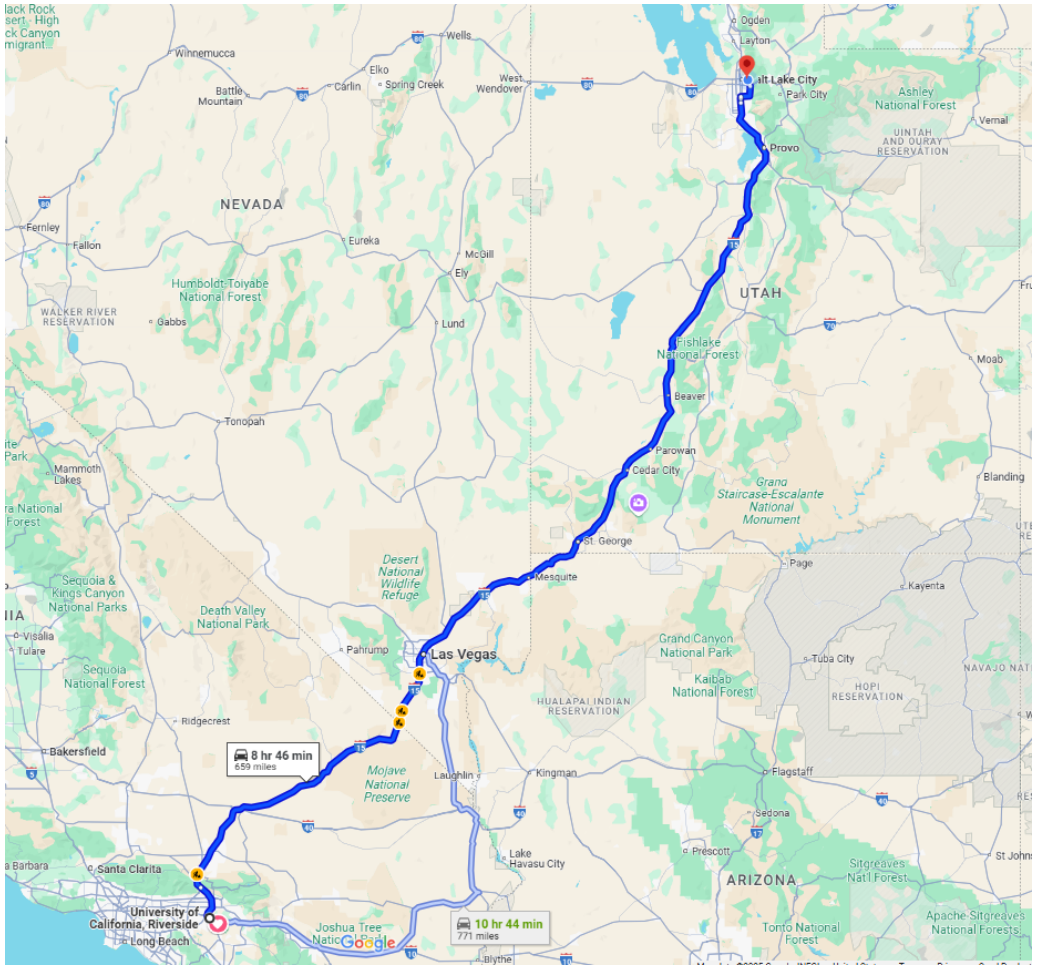
Parallel Contraction Hierarchies can be Efficient and Scalable

Zijin Wan [1], Xiaojun Dong [1], Letong Wang [1], Enzuo Zhu [2],
Yihan Sun [1], Yan Gu [1]
[1] University of California, Riverside
[2] University of California, Davis

Why is Navigation on Large Map So Fast?

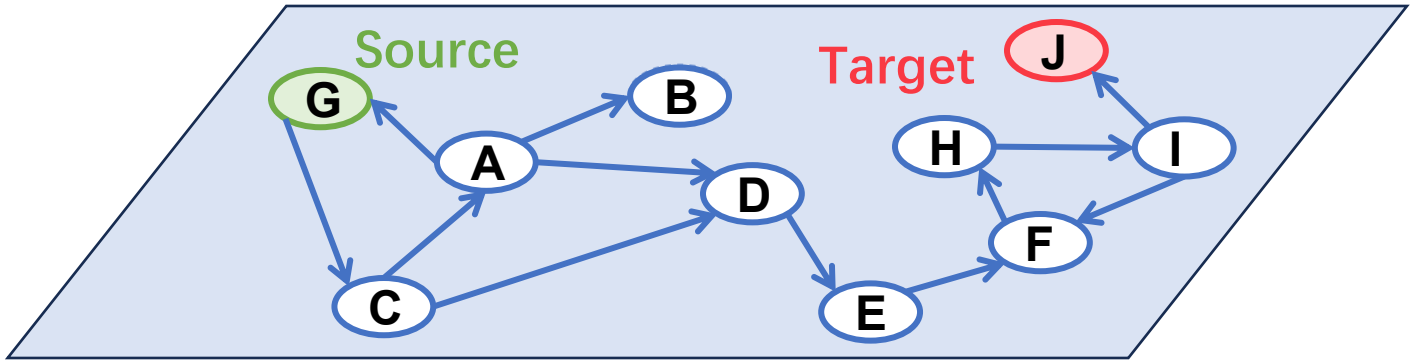
- **Route planning from Riverside to SLC only takes less than one second.**



Contraction Hierarchies (CH)

Point-to-point Shortest Path (PPSP)

- Input: a graph $G = (V, E, w)$, with positive edge weight function w: $e \mapsto \mathbb{R}^+$ and a pair of vertices $s, t \in V$. Let $n = |V|, m = |E|$
- Output: the **minimum total weight path** from source s to target t



Point-to-point Shortest Path (PPSP)

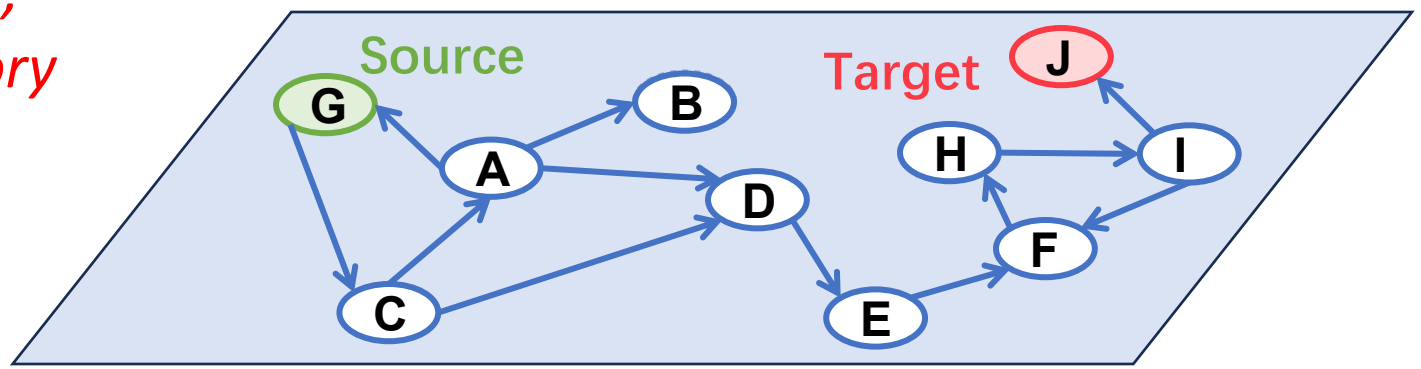
- The classic solution: Dijkstra's algorithm
- The parallel version: Δ-stepping [MS03]

North American Road Graph:
87 million vertices
113 million edges

Algorithm	Preprocessing time	Query time
Dijkstra [1959]	-	7.20 s
Δ-stepping [DGSZ'21]	-	0.320 s

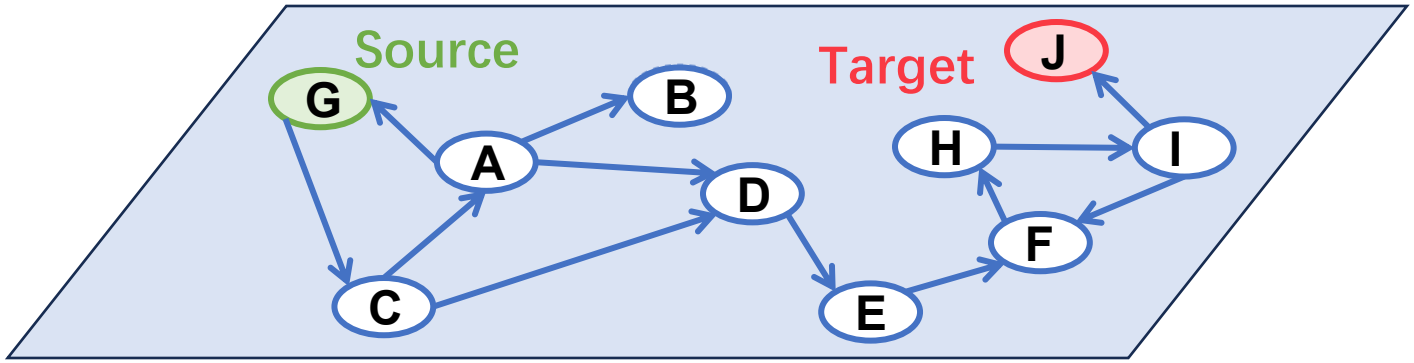
22.5x speedup

96-core machine,
1.5TB main memory



Contraction Hierarchies (CH) [1]

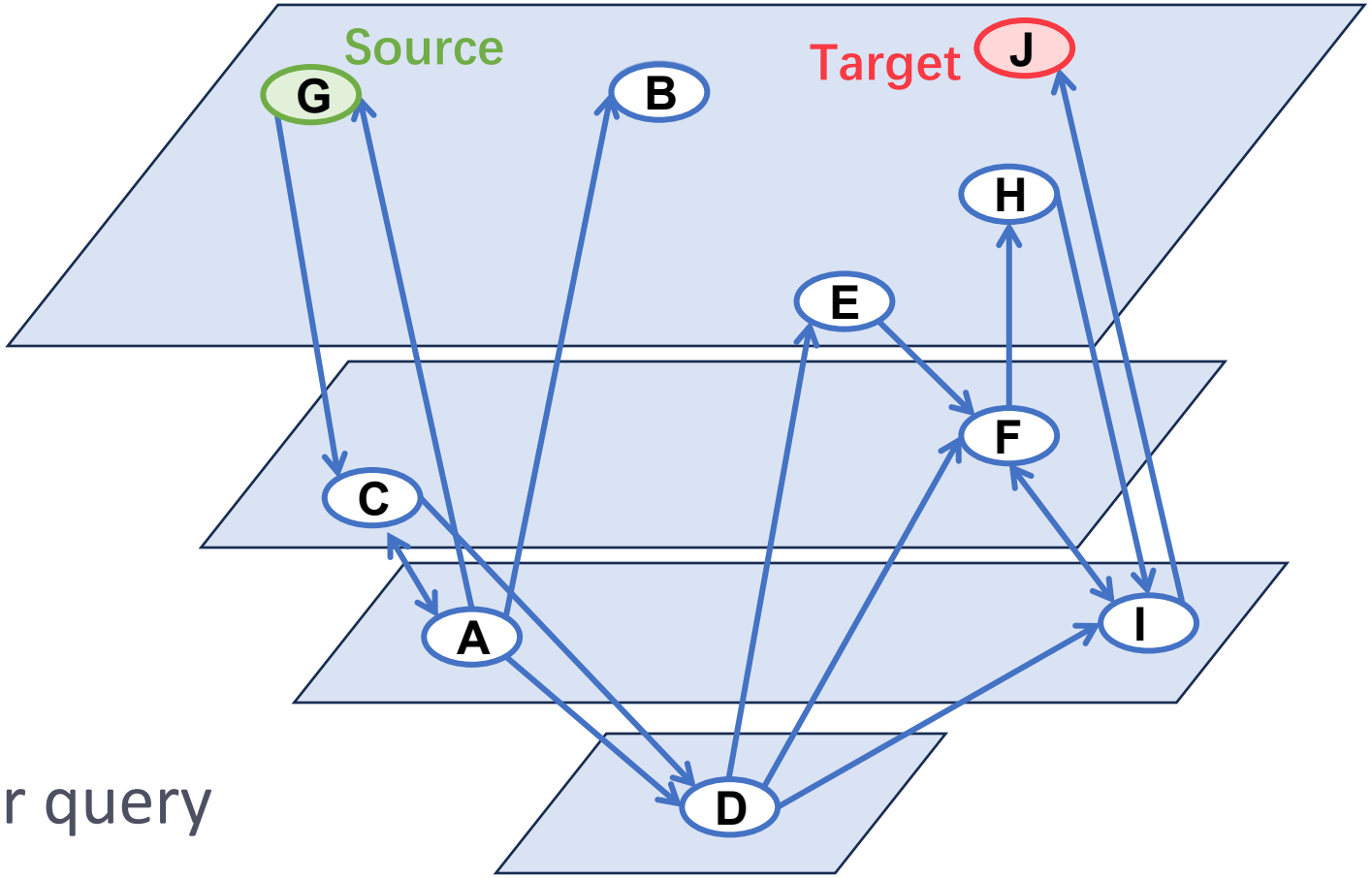
- Two-phases approach



[1] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In Proceedings of the International Conference on Experimental Algorithms (WEA), pages 319–333, 2008.

Contraction Hierarchies (CH) [1]

- **Construction:** Build a **hierarchical graph** G_{CH} that preserves the shortest distances
- **Query: Fast bidirectional searches** on G_{CH} that only traverse downward vertices
- Much fewer vertices explored
- Reduced search space => Faster query



Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In Proceedings of the International Conference on Experimental Algorithms (WEA), pages 319–333, 2008.

CH: Fast Queries, Slow Preprocessing

- Query time of CH is 10^3–10^5 times faster than classic algorithms.
- However, this comes at the cost of **slow preprocessing.**
- Existing parallel solution is only **up to 4.4×** faster than the sequential one.

North America Road Graph:
87 million vertices
113 million edges

96-core machine, 1.5TB main memory

	Algorithm	Preprocessing time	Query time	
	Dijkstra [1959]	-	7.20 s	
	Δ-stepping [DGSZ'21]	-	0.320 s	
CH-based	RoutingKit [JEA'08]	2466 s	79.1 µs	*Sequential baselines*
CH-based	PHAST [IPDPS'11]	1341 s	138 µs	*Sequential baselines*
CH-based	CH-Constructor [2024]	1527 s	317 µs	*Parallel baselines*
CH-based	OSRM [SIGSPATIAL'11]	307 s	163 µs	*Parallel baselines*
CH-based	Ours [ICS'25]	23.1 s	93.3 µs	

Our Contribution

- We propose the first **truly scalable and efficient** parallel CH algorithm.
- On this graph, our algorithm:
  - Achieves a 58× speedup over the best sequential baseline
  - Delivers competitive query performance

North America Road Graph:
87 million vertices
edges

96-core
1.5TB m

Why is CH construction so costly?

CH-bas	PHAST [IPDPS'11]	1341 s	138 µs
	CH-Constructor [2024]	1527 s	317 µs
	OSRM [SIGSPATIAL'11]	307 s	163 µs
	Ours [ICS'25]	23.1 s	93.3 µs

tial baselines

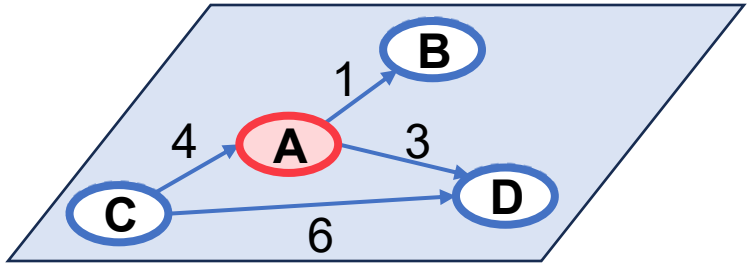
Parallel baselines

CH: Vertex Contraction

- **Contract a vertex A**
  - Move A to the CH

Contraction Hierarchies G_{CH}

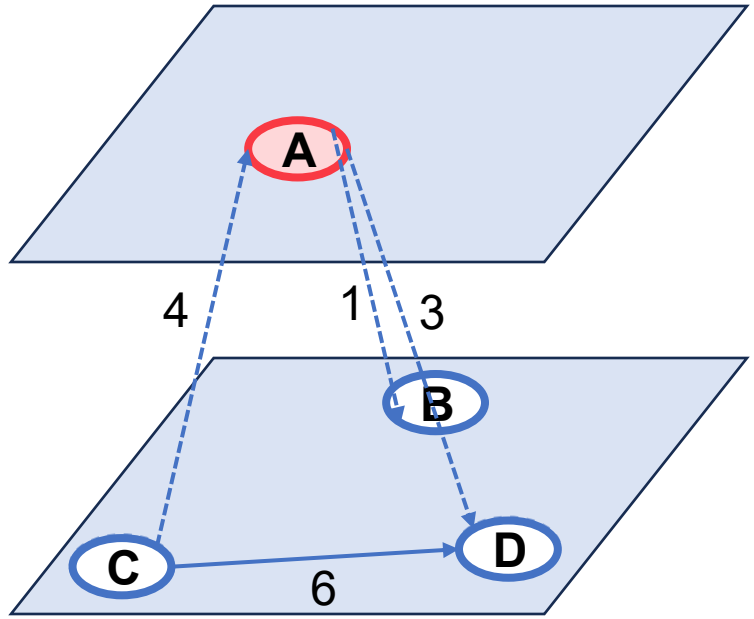
Overlay Graph G_O



CH: Vertex Contraction

- **Contract a vertex A**
  - Move A to the CH

Contraction Hierarchies G_{CH}



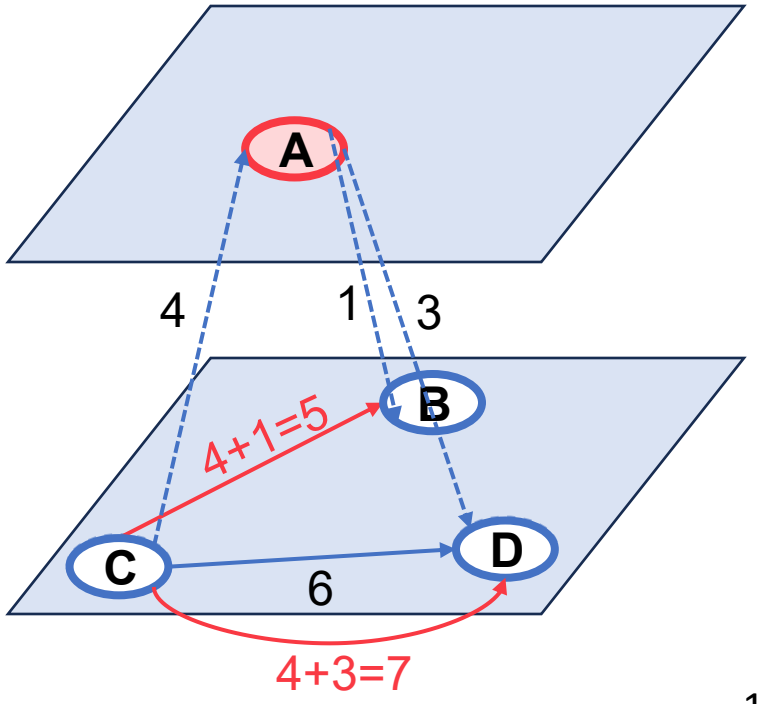
Overlay Graph G_O

CH: Vertex Contraction

- **Contract a vertex A**
  - Move A to the CH
  - Add **shortcuts** in the overlay graph to preserve shortest path distances
  - (run a shortest path algorithm to determine whether/how shortcuts should be added)

Contraction Hierarchies G_{CH}

Overlay Graph G_O

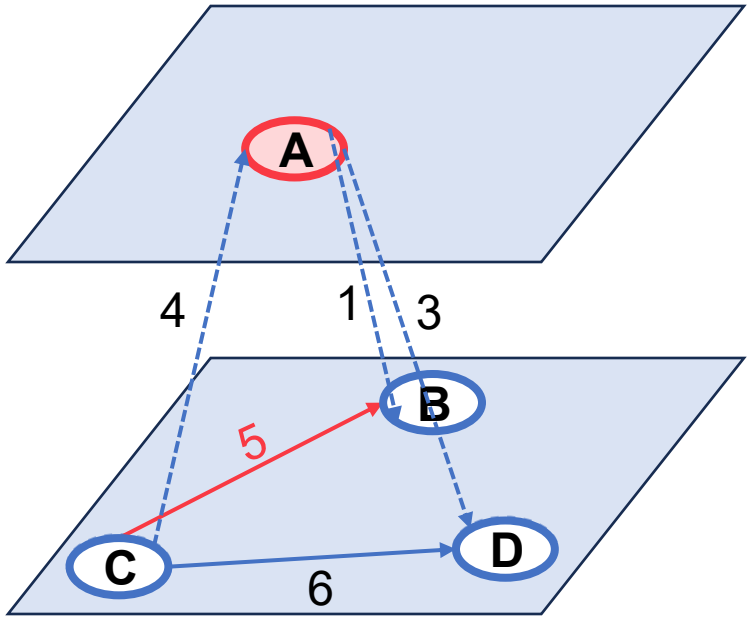


CH: Vertex Contraction

- **Contract a vertex A**
  - Move A to the CH
  - Add **shortcuts** in the overlay graph to preserve shortest path distances
  - (run a shortest path algorithm to determine whether/how shortcuts should be added)
- **Contract the vertices one by one**
  - Until the overlay graph becomes empty.

Contraction Hierarchies G_{CH}

Overlay Graph G_O

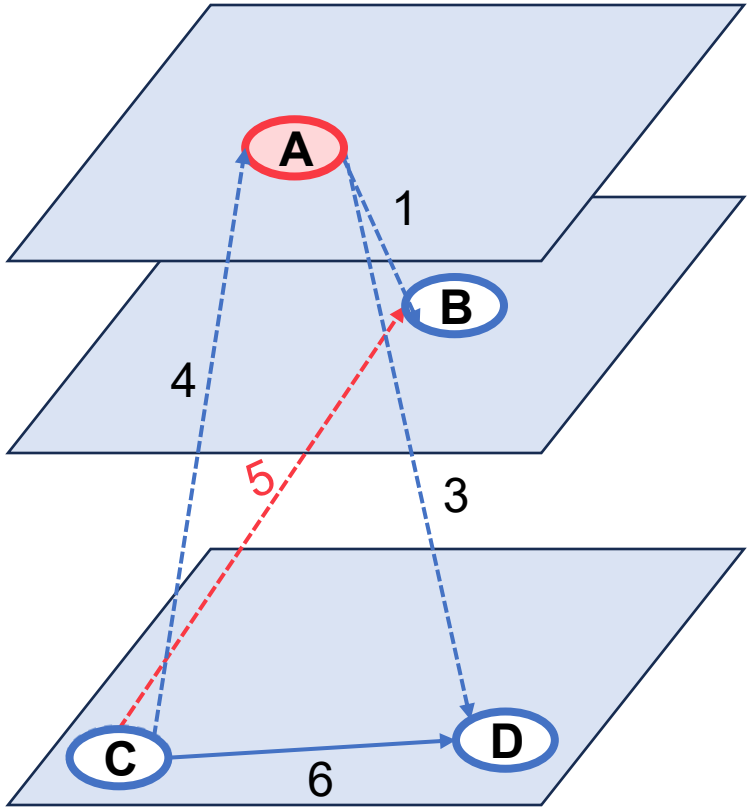


CH: Vertex Contraction

- **Contract a vertex A**
  - Move A to the CH
  - Add **shortcuts** in the overlay graph to preserve shortest path distances
  - (run a shortest path algorithm to determine whether/how shortcuts should be added)
- **Contract the vertices one by one**
  - Until the overlay graph becomes empty.

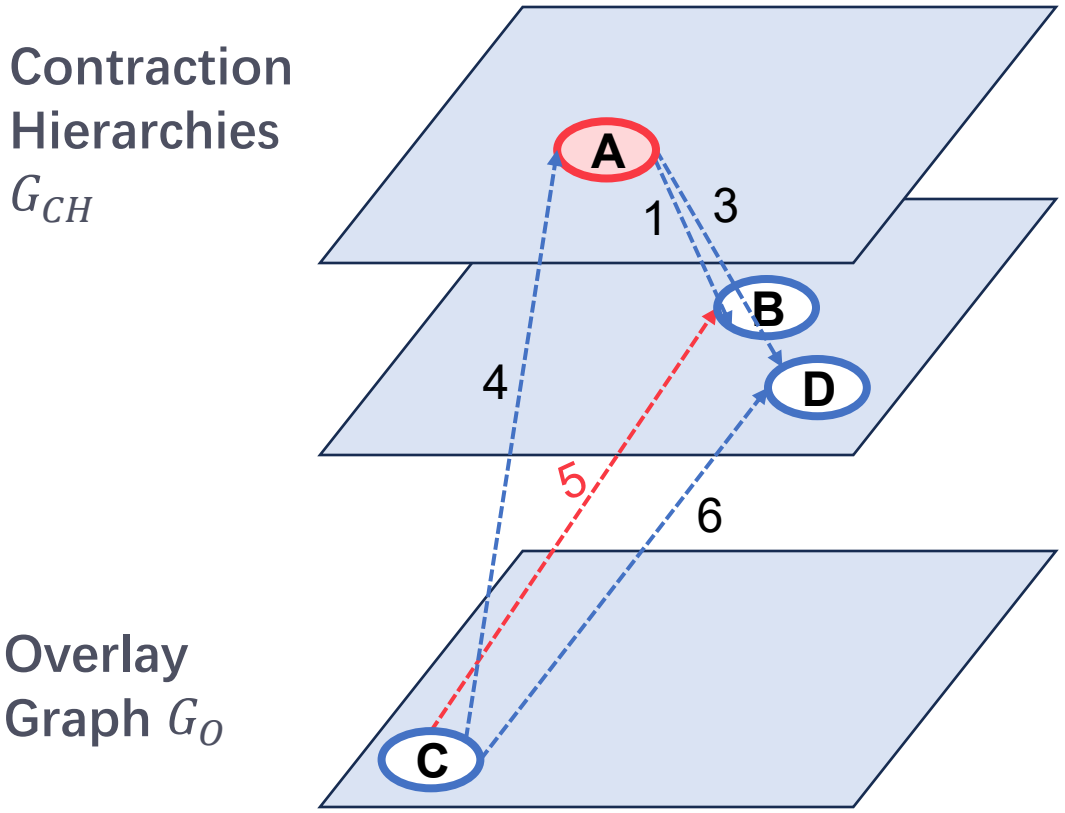
Contraction Hierarchies G_{CH}

Overlay Graph G_O



CH: Vertex Contraction

- **Contract a vertex A**
  - Move A to the CH
  - Add **shortcuts** in the overlay graph to preserve shortest path distances
  - (run a shortest path algorithm to determine whether/how shortcuts should be added)
- **Contract the vertices one by one**
  - Until the overlay graph becomes empty.

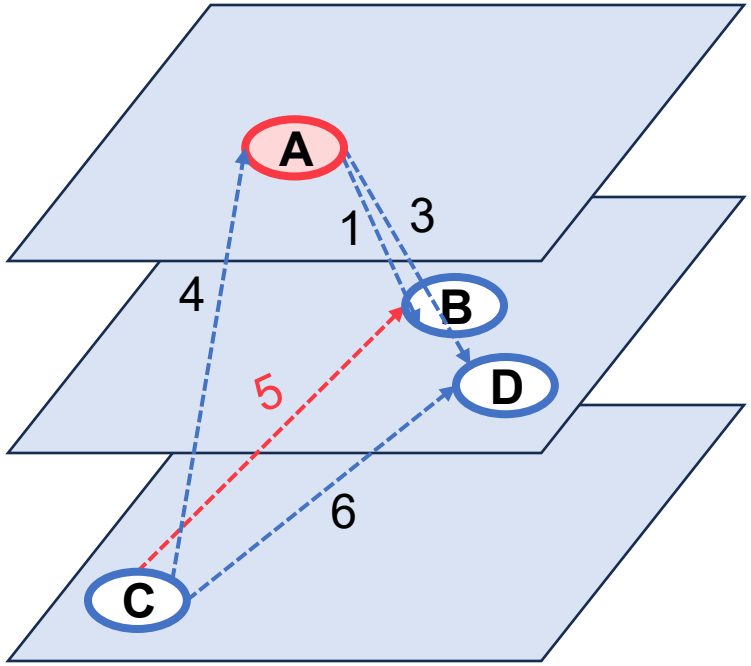


CH: Vertex Contraction

- **Contract a vertex A**
  - Move A to the CH
  - Add **shortcuts** in the overlay graph to preserve shortest path distances
  - (run a shortest path algorithm to determine whether/how shortcuts should be added)
- **Contract the vertices one by one**
  - Until the overlay graph becomes empty.
- Co… …kes the … …rtices smaller => faster queries!

In What Order?

Contraction Hierarchies G_{CH}

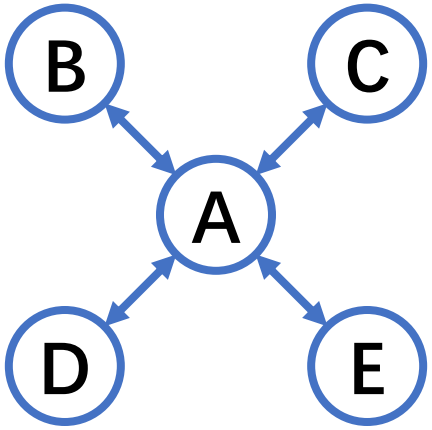


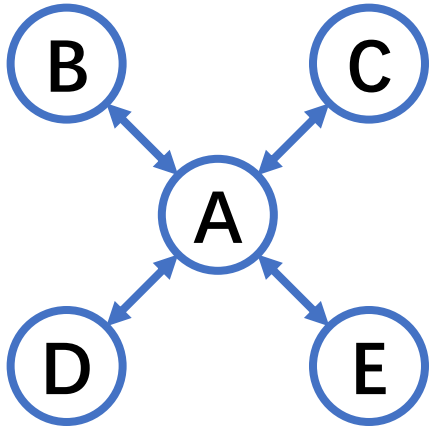
Overlay Graph G_O

Contraction Order: Vertex Scoring

- Each vertex is given a **score**, denoting the cost of contracting the vertex

Score(v) = # of **changed-edges** if contracting v
= {# of **shortcuts added** by contracting v} – {# of v's **adjacent edges**}



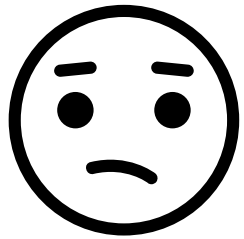


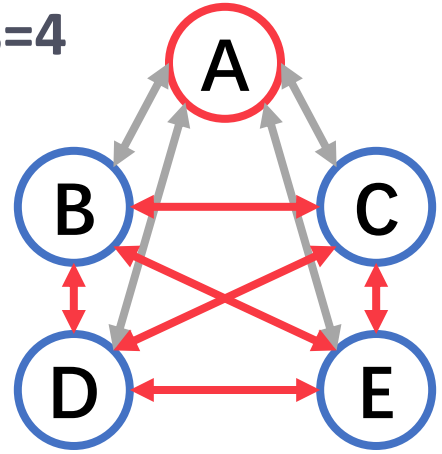
Contraction Order: Vertex Scoring

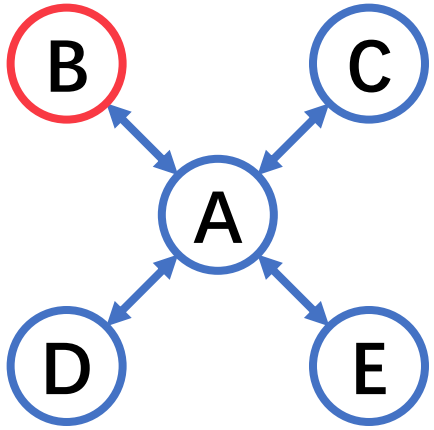
- Each vertex is given a **score**, denoting the cost of contracting the vertex

Score(v) = # of **changed-edges** if contracting v
= {# of **shortcuts added** by contracting v} – {# of v's **adjacent edges**}

Contracting A
Removing 8 edges, adding 12 shortcuts
Score(A) = 12-8=4







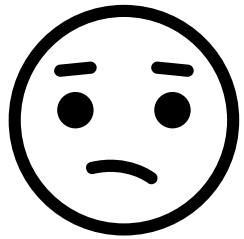
Contraction Order: Vertex Scoring

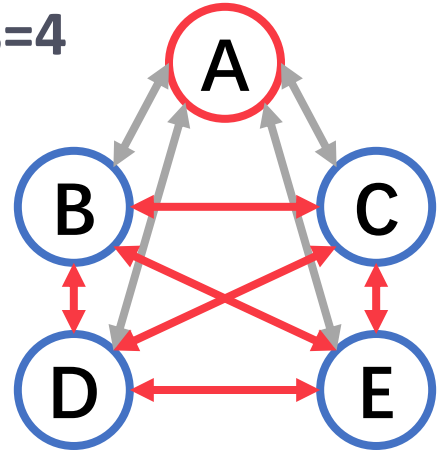
- Each vertex is given a **score**, denoting the cost of contracting the vertex

Score(v) = # of **changed-edges** if contracting v
= {# of **shortcuts added** by contracting v} – {# of v's **adjacent edges**}

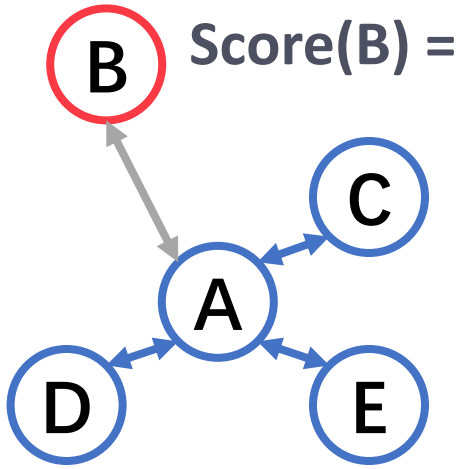
- Contract the vertex with the **lowest score** first!

Contracting A
Removing 8 edges, adding 12 shortcuts
Score(A) = 12-8=4



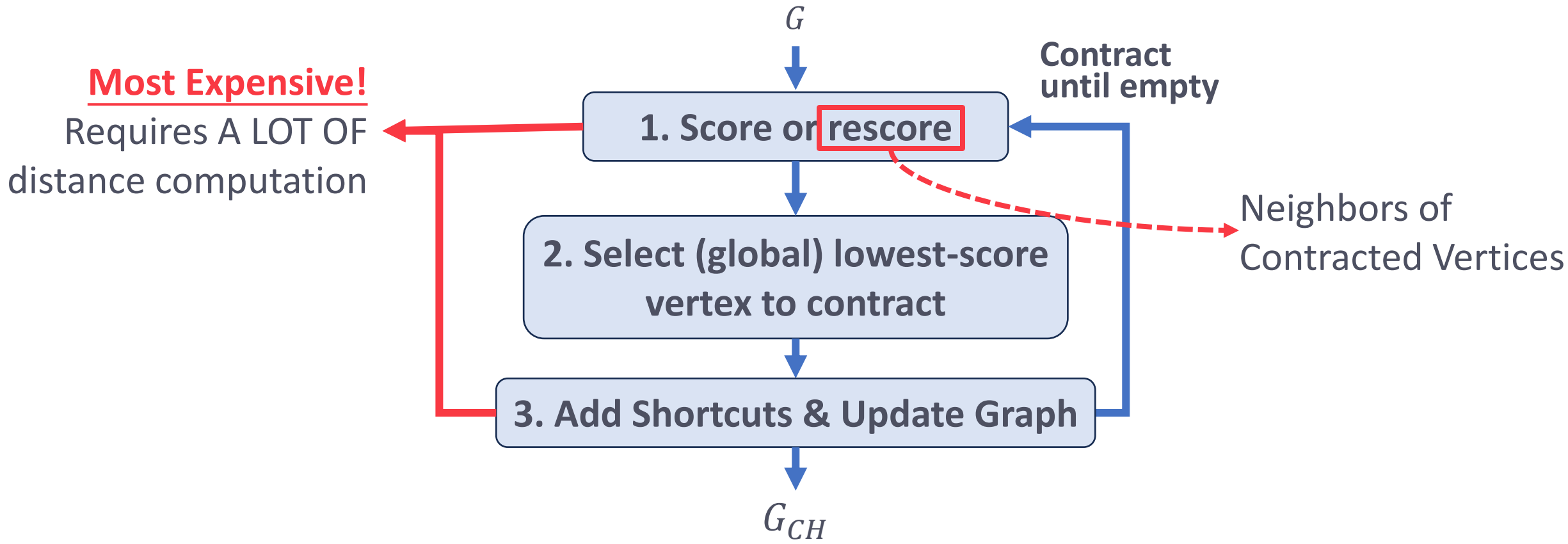


Contracting B:
Removing 2 edge, adding 0 shortcut
Score(B) = 0-2=-2



Sequential Algorithms to build a CH

- **Reasons that CH construction is slow:**



How to parallelize it?

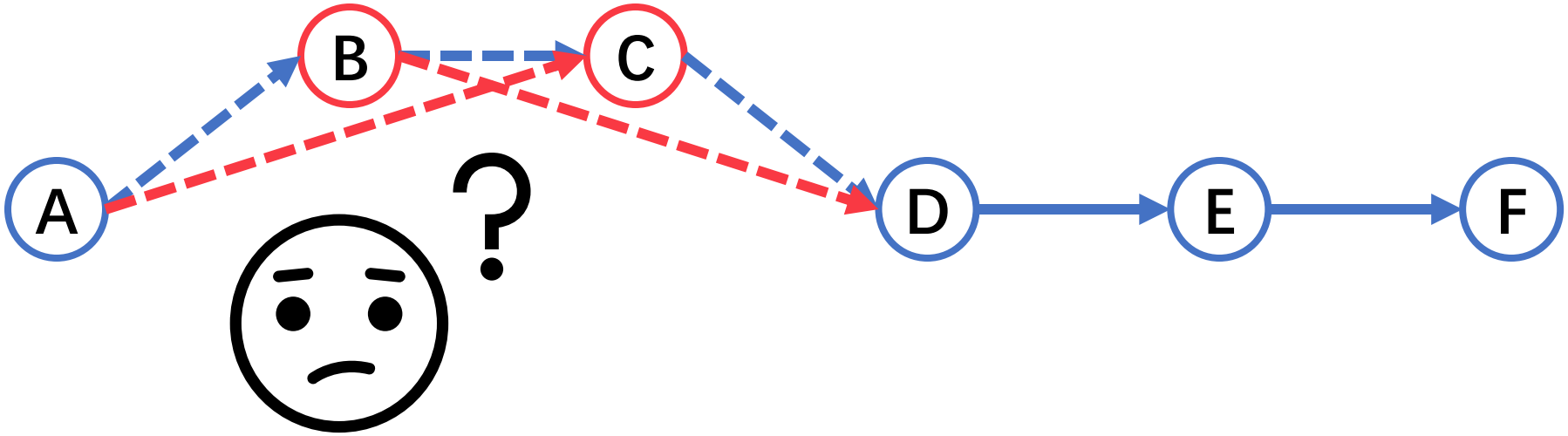
How to contract in parallel?

- Can we contract arbitrary vertices in parallel?

How to contract in parallel?

Key observations [Vetter'09]

1. Neighboring vertices cannot be contracted in parallel
2. low-score vertices should still be prioritized

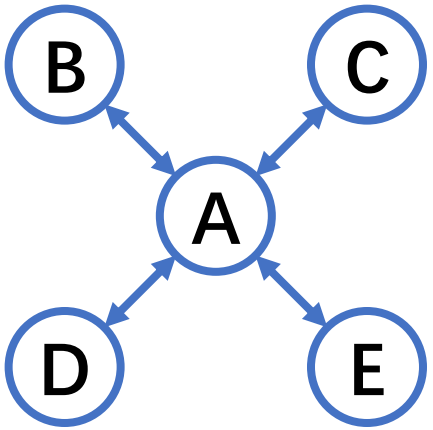


How to contract in parallel?

Key observations [Vetter'09]

1. **Neighboring vertices cannot be contracted in parallel**
2. **low-score vertices should still be prioritized**

Strategy: Contract **all** vertices with **the lowest score among their neighbors** in parallel!

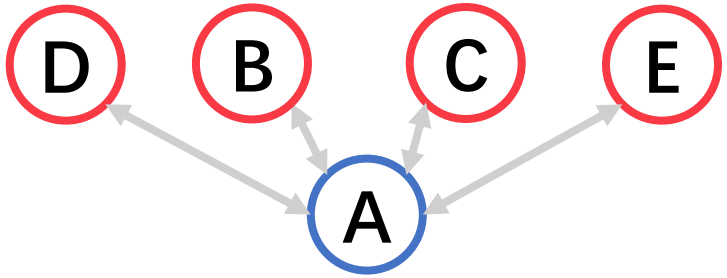


How to contract in parallel?

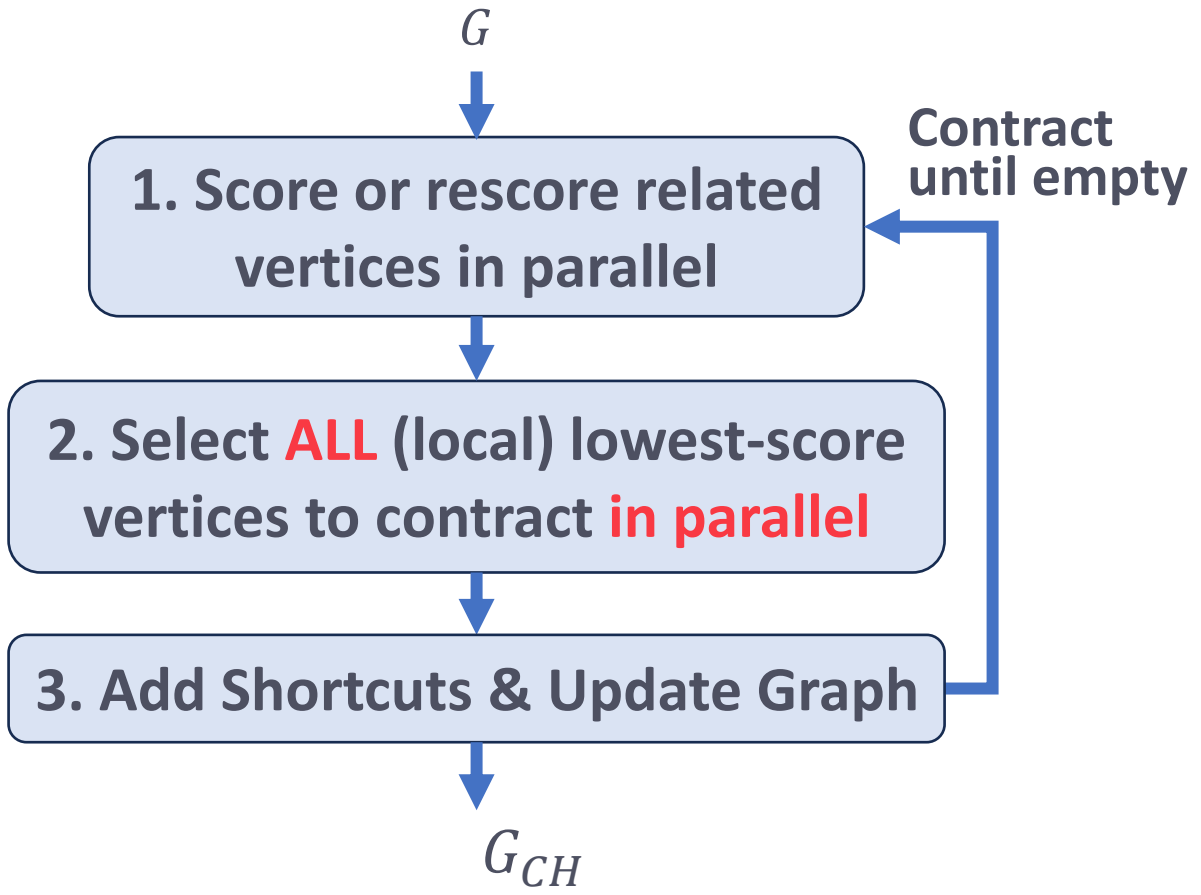
Key observations [Vetter'09]

1. **Neighboring vertices cannot be contracted in parallel**
2. **low-score vertices should still be prioritized**

Strategy: Contract **all** vertices with **the lowest score among their neighbors** in parallel!



Parallel Contraction Hierarchy [Vetter'08]

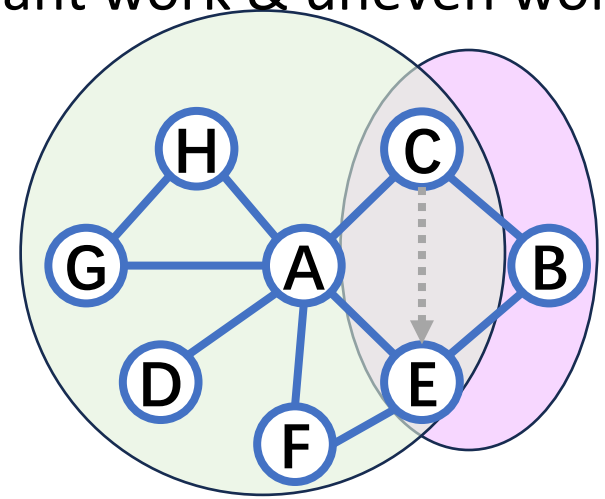


What makes the parallel algorithm slow?

Parallelizing the scoring for the vertices

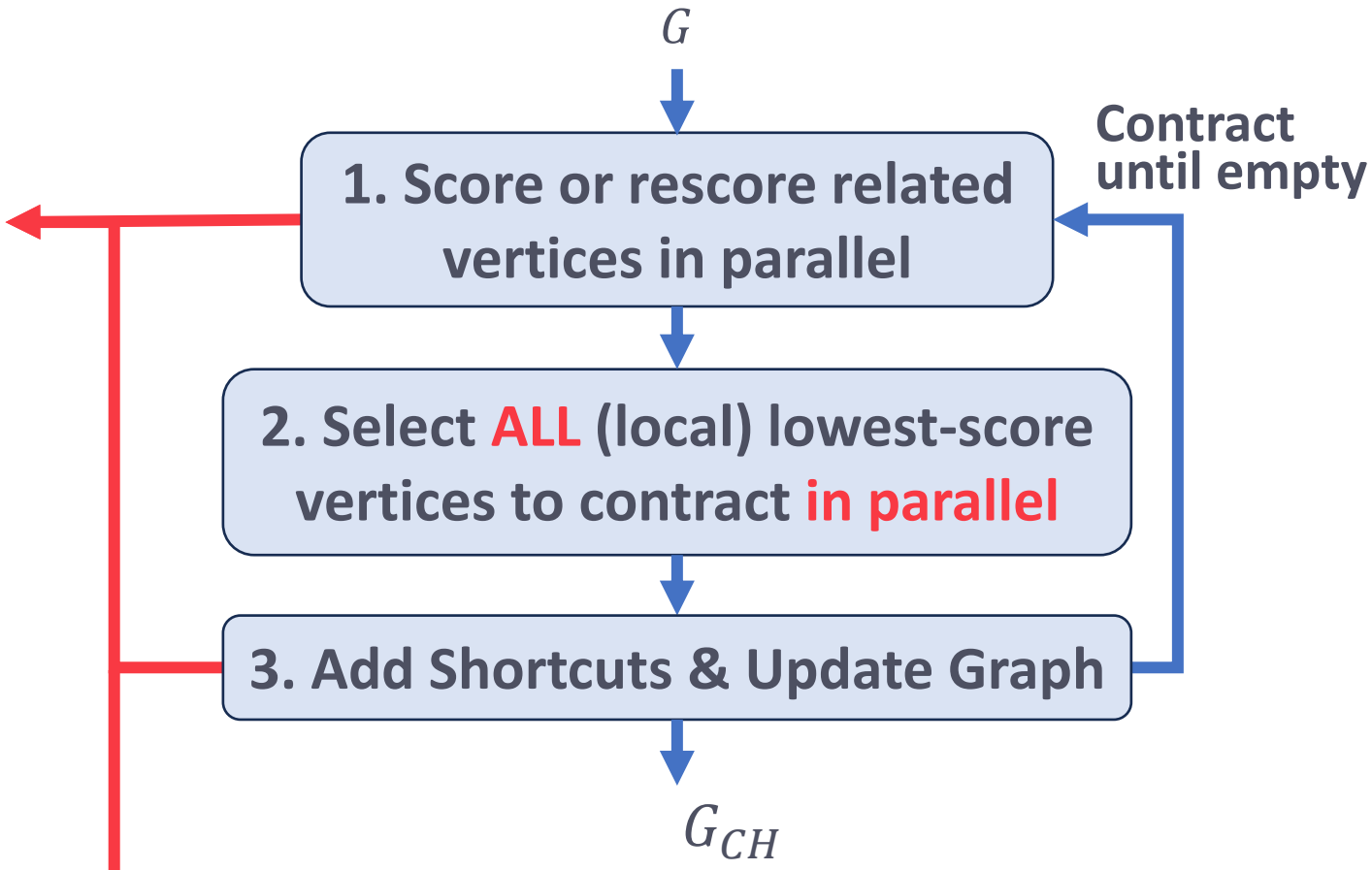
Challenge 1: score & shortcut multiple vertices in parallel

A LOT OF distance computation
Redundant work & uneven workload



Challenge 2: Maintaining the graph dynamically in parallel

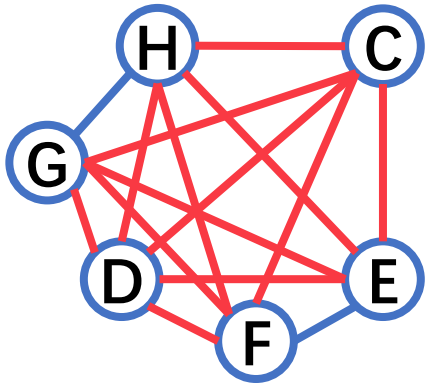
The graph changes every round



Parallelizing the scoring for the vertices

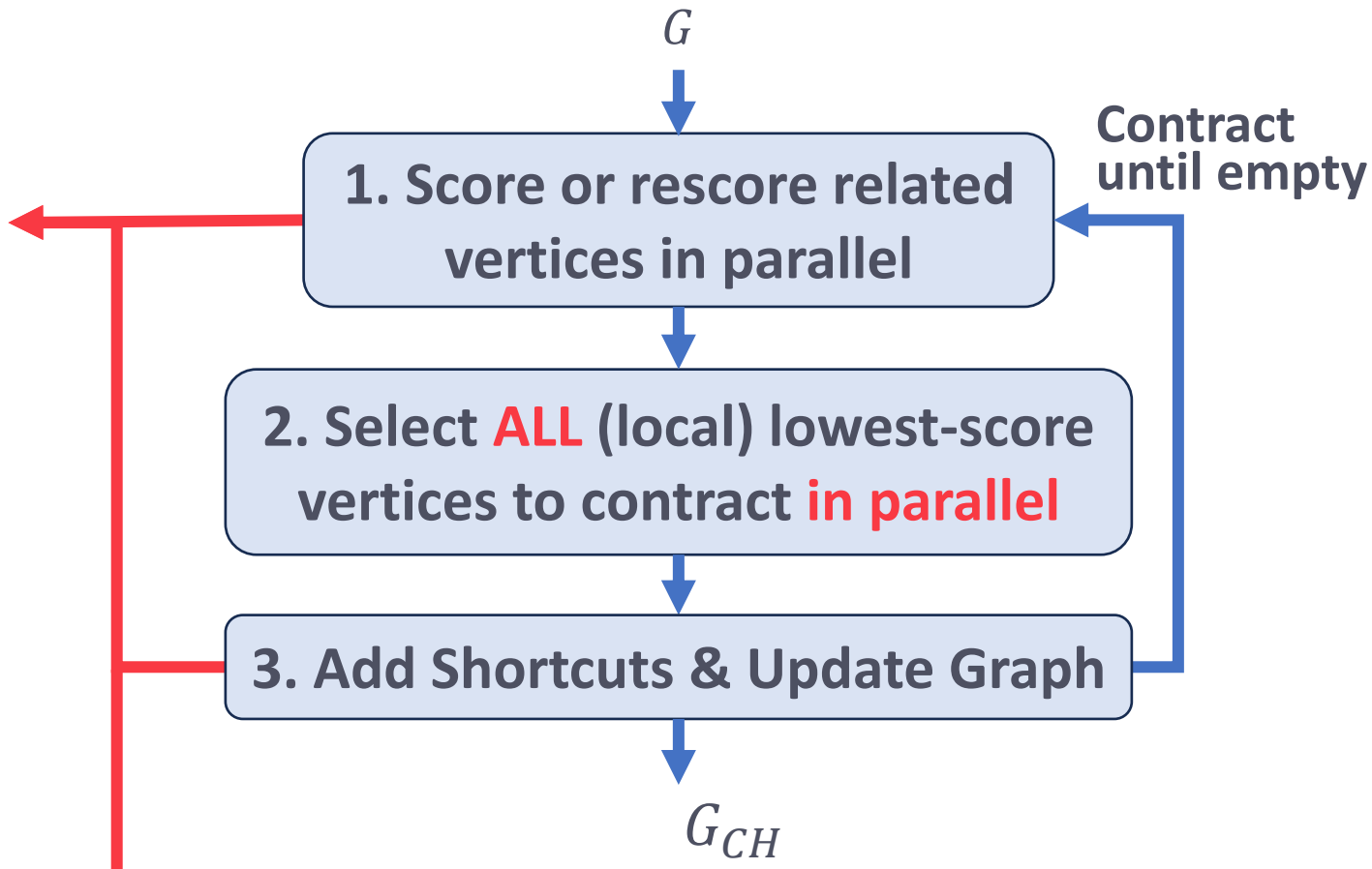
Challenge 1: score & shortcut multiple vertices in parallel

A LOT OF distance computation
Redundant work & uneven workload



Challenge 2: Maintaining the graph dynamically in parallel

The graph changes every round

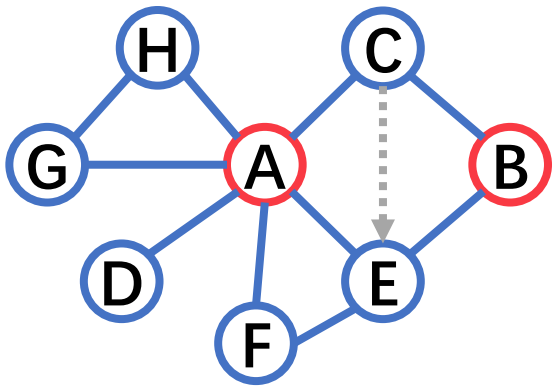


SPoCH: Scalable Parallelization of Contraction Hierarchies

Parallelizing the distance calculations

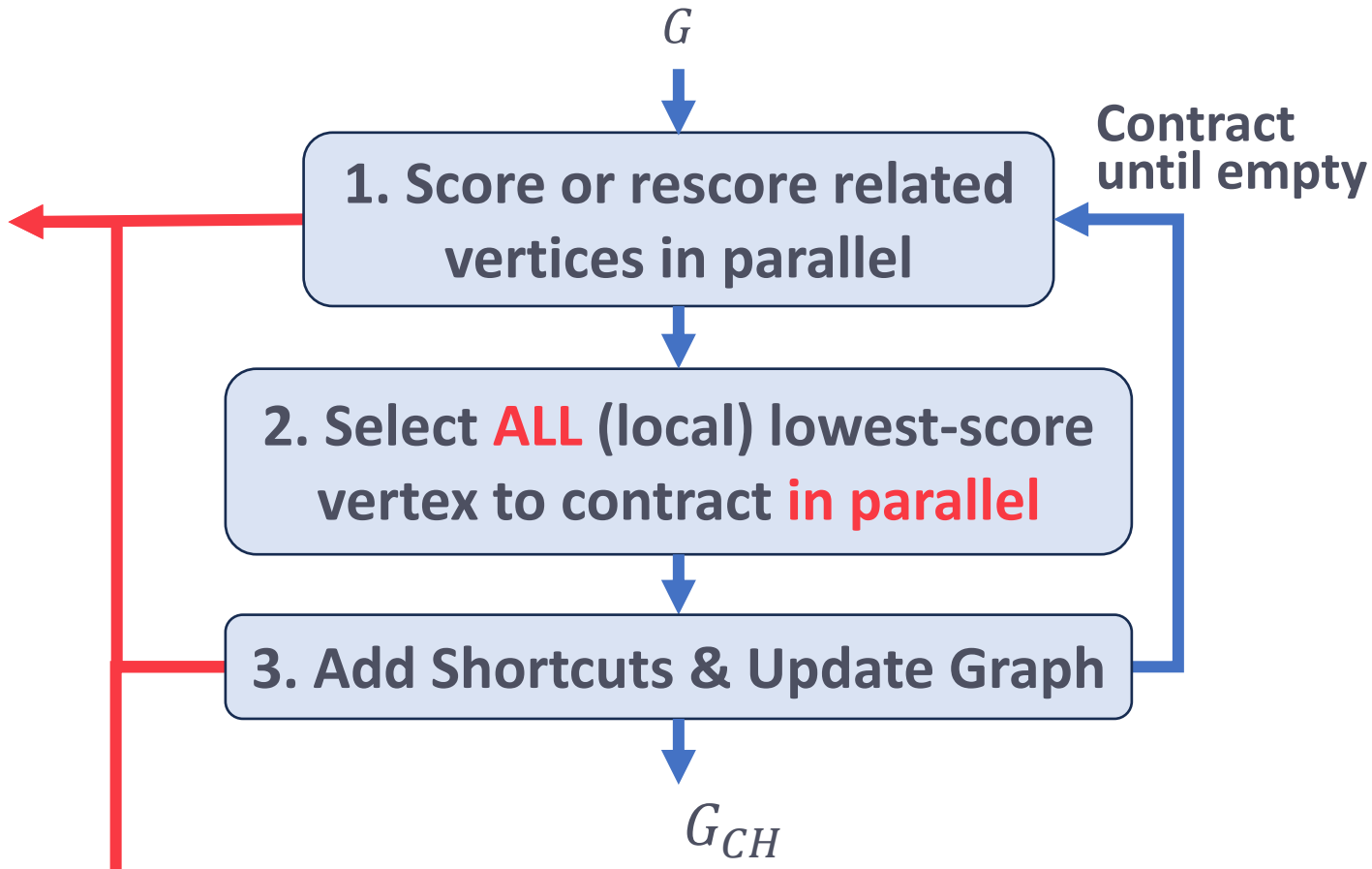
Challenge 1: score & shortcut multiple vertices in parallel

Smaller tasks → better parallelism
Enables **batching** and **memoization**



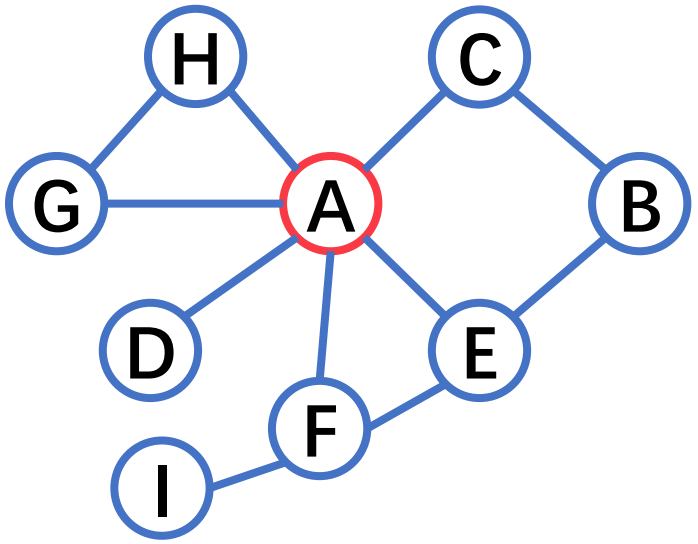
Challenge 2: Maintaining the graph dynamically in parallel

Efficient **parallel data structure** to manage graph updates



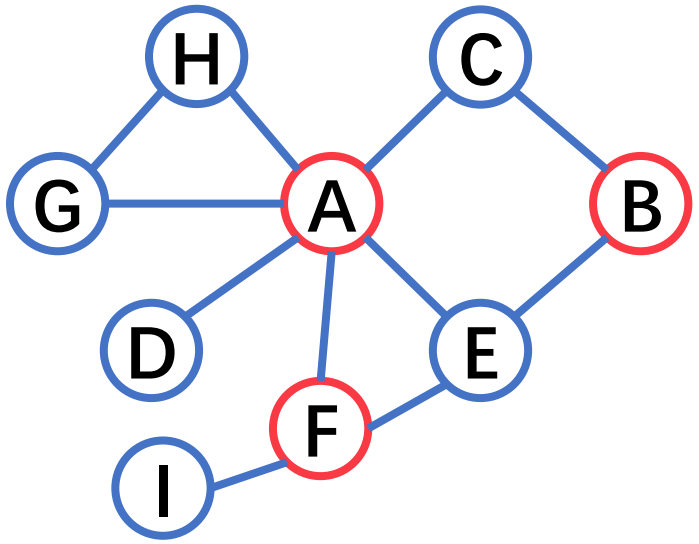
Improving parallelism and reducing work by parallelizing the distance computations

- In each round, each vertex to be (re)scored will start a search from each neighbor and compute the distances to all other neighbors



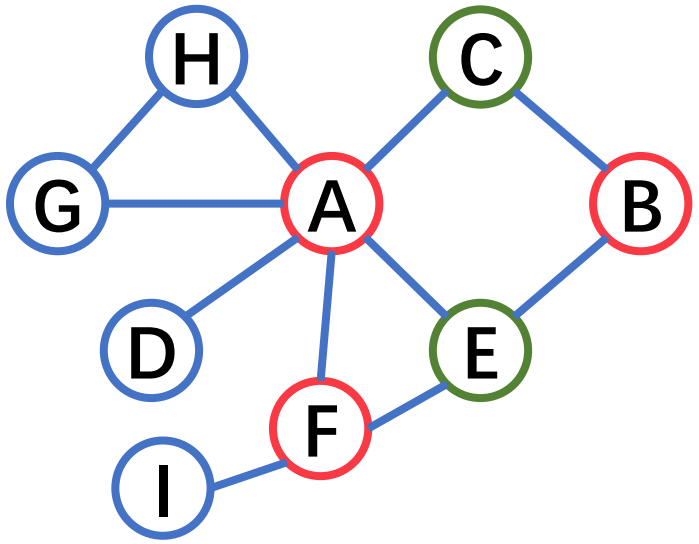
Improving parallelism and reducing work by parallelizing the distance computations

- For multiple vertices to be scored in one round, we will gather all the sources, and apply only one search for each source vertex, in parallel
- Much better parallelism due to smaller task granularity!



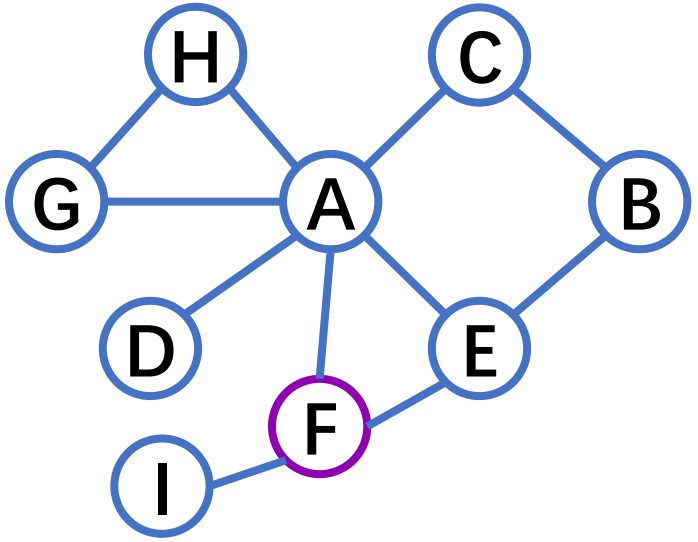
Improving parallelism and reducing work by parallelizing the distance computations

- For multiple vertices to be scored in one round, we will gather all the sources, and apply only one search for each source vertex, in parallel
- Much better parallelism due to smaller task granularity!
- Redundant work saved!
- Details on how to efficiently achieve so are in our paper



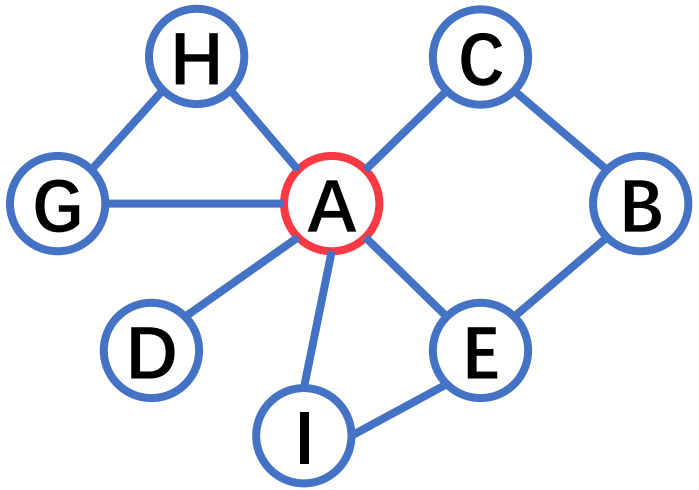
Saving the work across rounds!

- Assume F is contracted in this round,



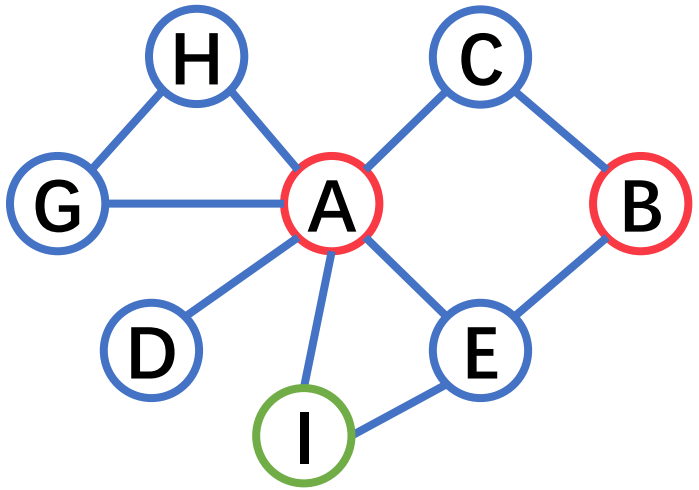
Saving the work across rounds!

- Assume F is contracted in this round, and A needs to be rescored
- If we have the distances computed before, then not all A's neighbor needs to be recomputed!
- Indeed, for this example, we only need to search from vertex I

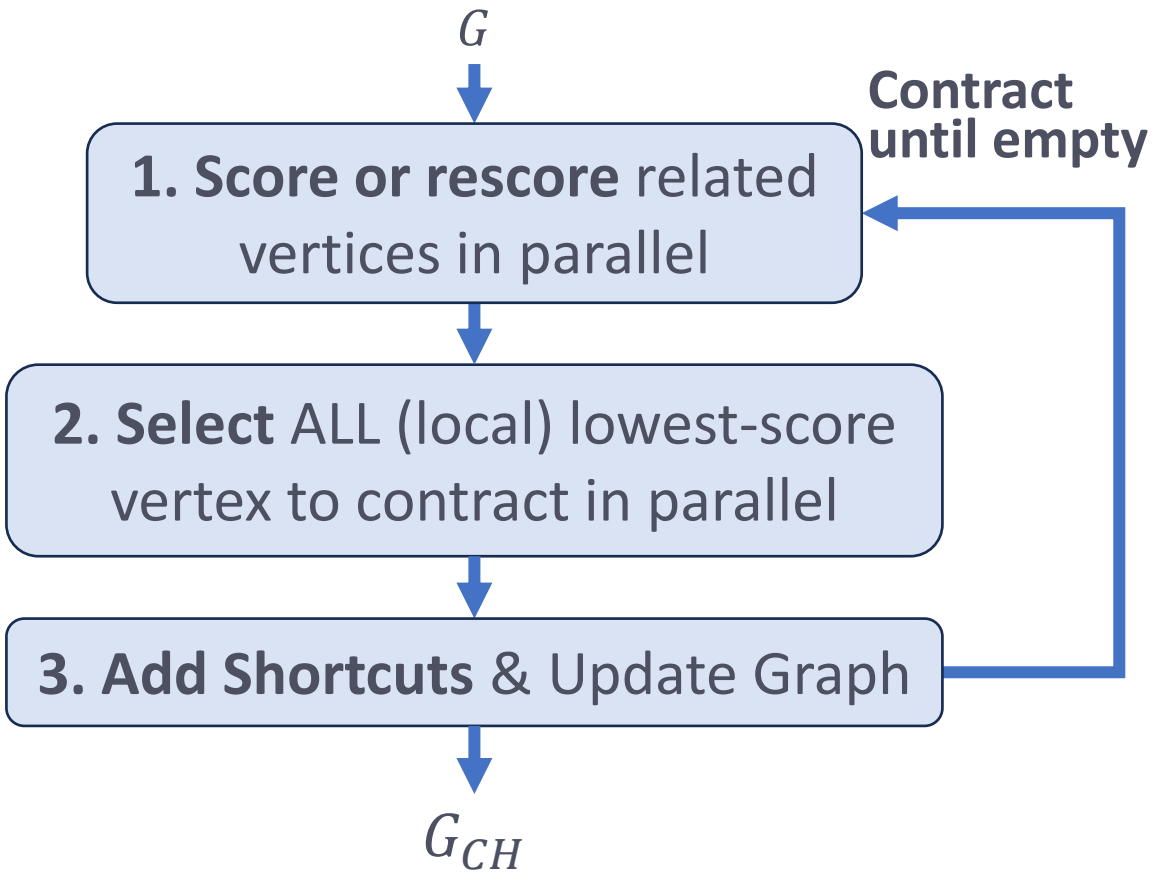


Saving the work across rounds!

- Assume F is contracted in this round, and A needs to be rescored
- We need a parallel hash table to memoize the distances
- This approach can accelerate the sequential algorithm!

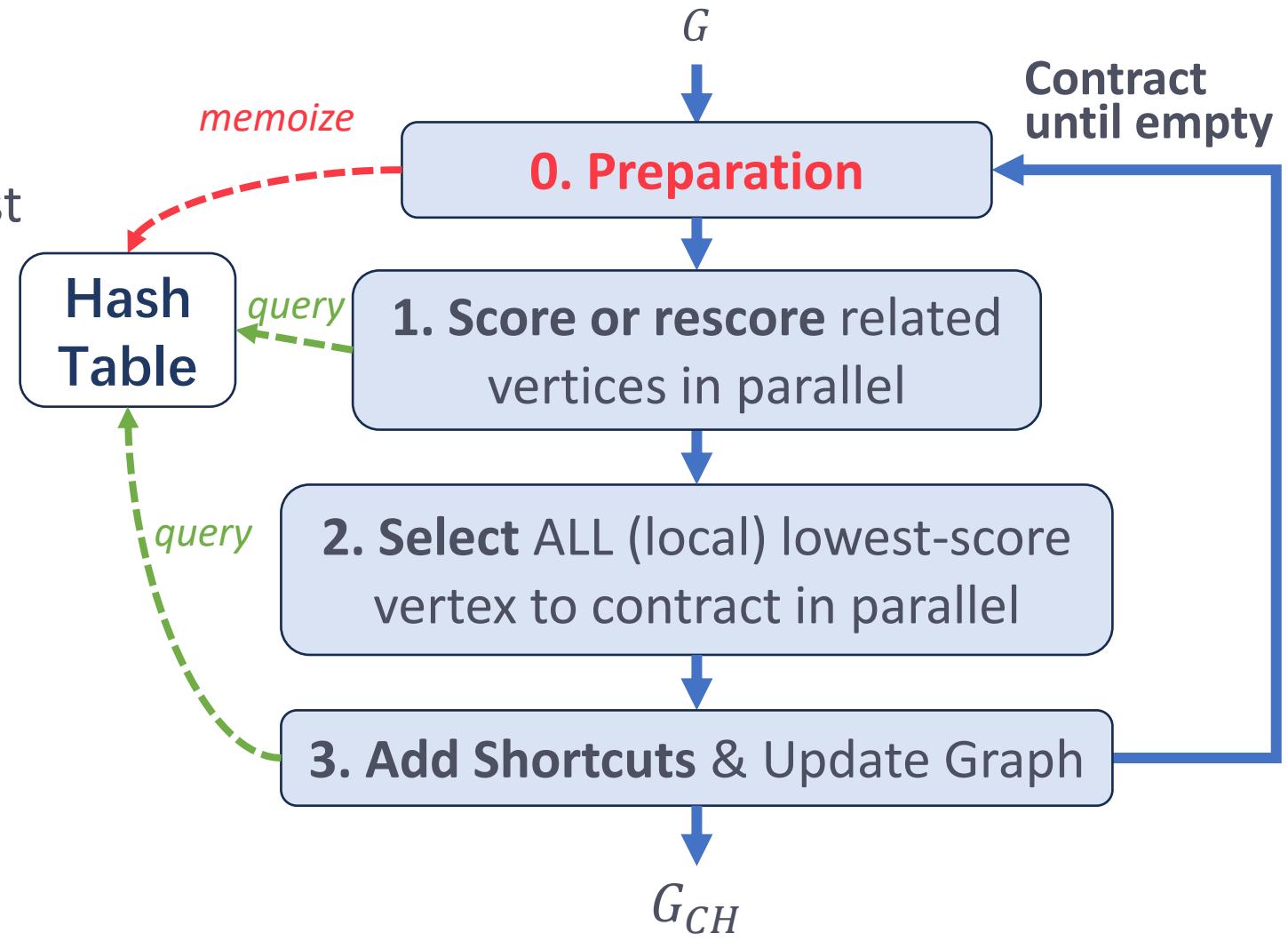


Parallel Contraction Hierarchy [Vetter's]



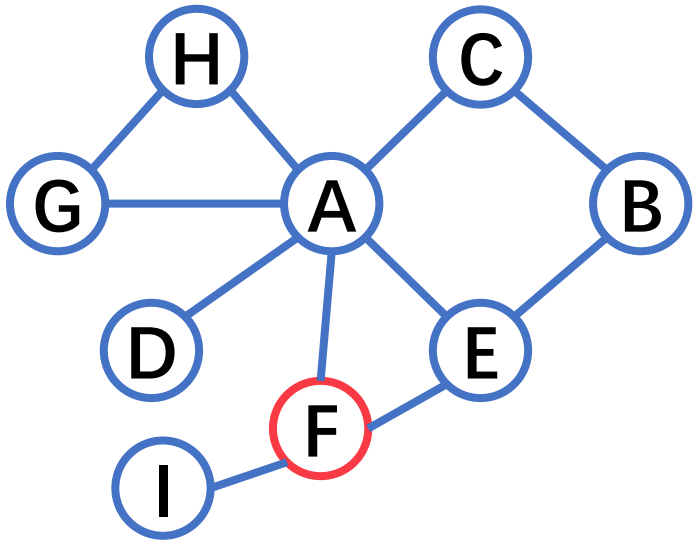
Parallel Contraction Hierarchy [This Paper]

- Add a step 0, **Preparation**, at the beginning of each round:
  - Compute all necessary shortest distances in a **source-centric** strategy.
  - **Memoize** the results in the hash table.
- Step 1 and 3:
  - **Perform shortest distance queries** to decide whether or not a shortcut should be inserted.



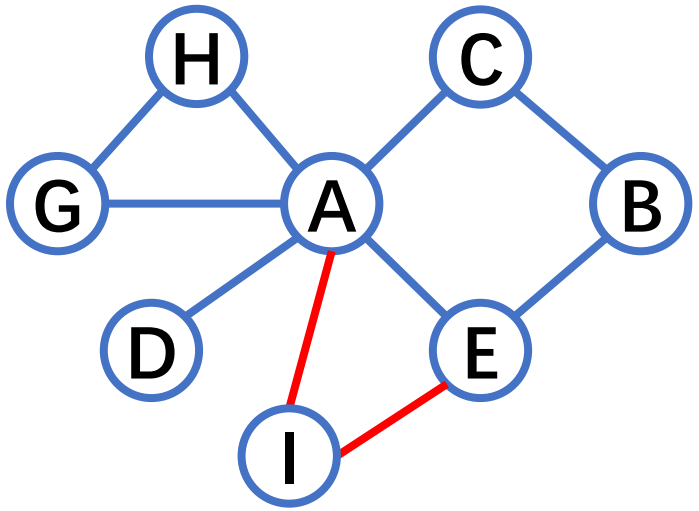
Challenges in Parallel Data Structure Design

- Needed in two places in the algorithm
- 1. Memoizing the calculated distances
- 2. Maintain edge updates



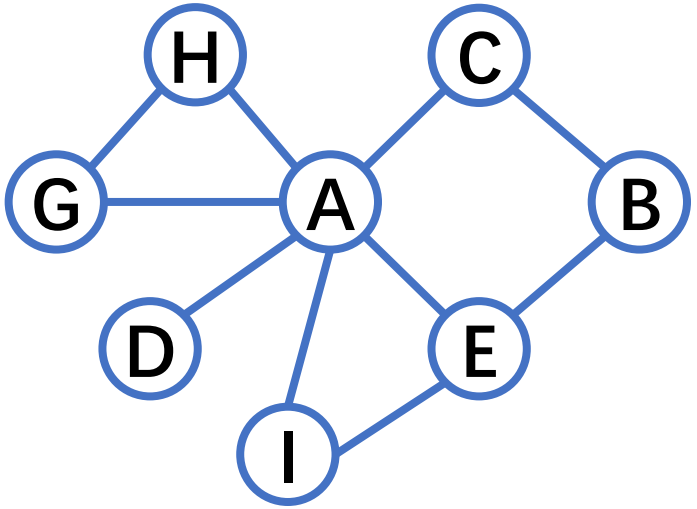
Challenges in Parallel Data Structure Design

- Needed in two places in the algorithm
- 1. Memoizing the calculated distances
- 2. Maintain edge updates



Algorithmic Challenges

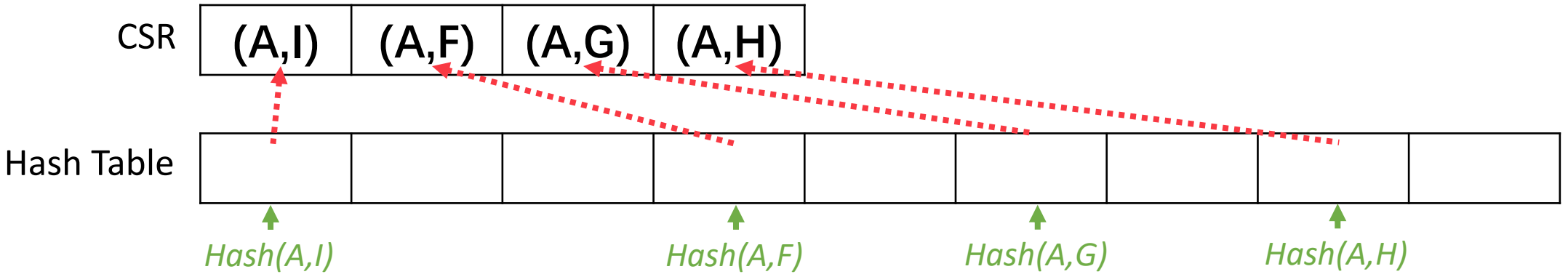
- What is the interface for this data structure?
- It should support: (1) inserting edges (u, v, w), and (2) list all neighbors from a vertex u
- Static case (only for (2)): compressed sparse row (CSR)



Our algorithmic highlights

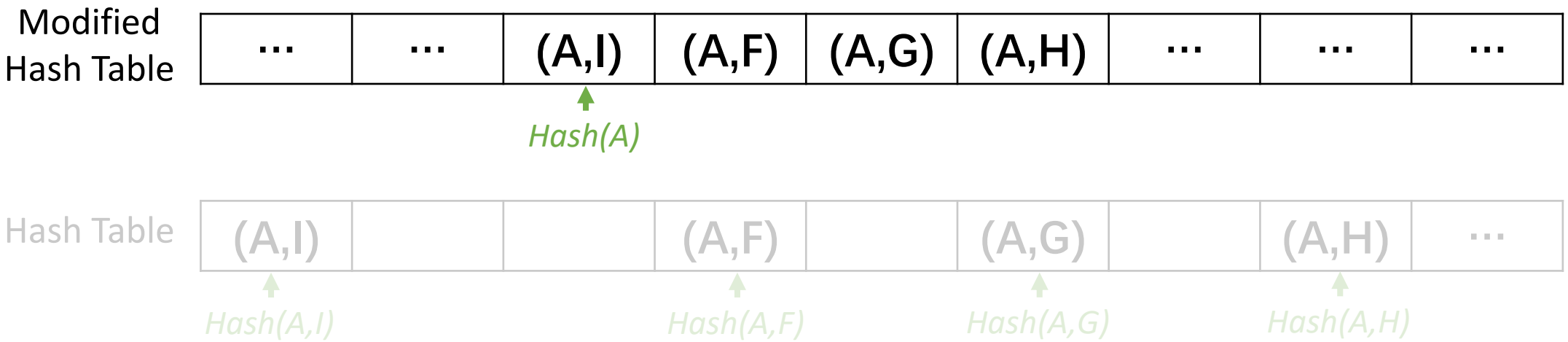
- Still keep the CSR for the original graph for efficiently traversing the original edges
- Maintain the inserted edges using a **phase-concurrent hash table**[1]
- For each edge (u, v, w) to be inserted, we consider u as the key while (v, w) as the value

[1] Julian Shun and Guy E Blelloch. 2014. Phase-concurrent hash tables for determinism. In ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). 96–107.



Our algorithmic highlights

- Still keep the CSR for the original graph for efficiently traversing the original edges
- Maintain the inserted edges using a **phase-concurrent hash table**[1]
- For each edge (u, v, w) to be inserted, we consider u as the key while (v, w) as the value
- **Lazily merge** the hash table to the CSR to amortize the cost



Summary of Our Algorithm

1) **Source-Centric** strategy: batch shortcuts sources and run shortest path searches in parallel for better load balance.

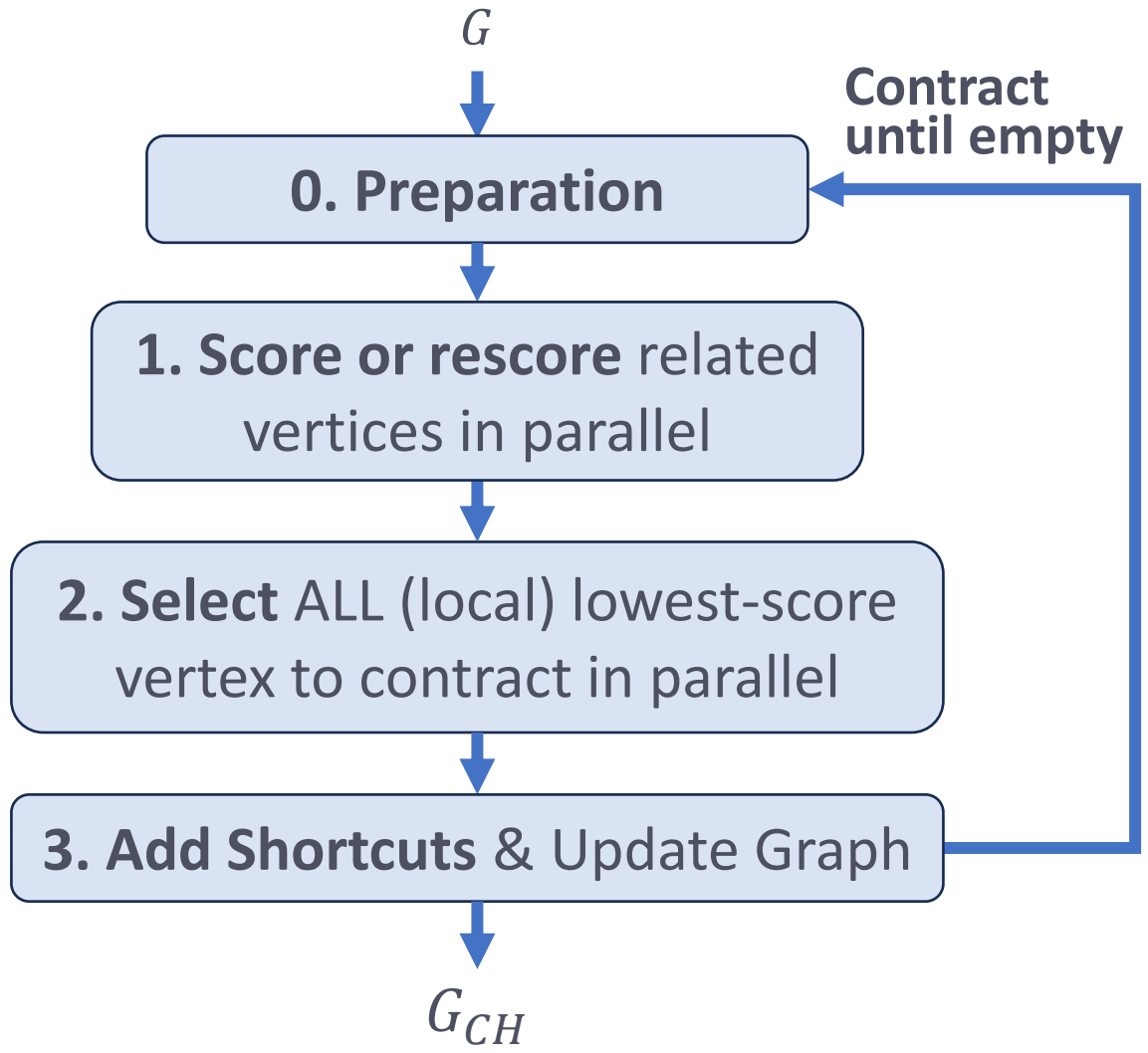
Better Load Balance

2) **CH Framework Redesign:** Add a **Preparation** step to compute shortest distances. Making step 1 & 3 query-only.

Reduce Redundancy

3) **Dynamic Graph Structure:** Efficiently support concurrent shortcut insertions and fast edge list traversal.

Efficient Overlay graph maintenance



Experiments

Experiment Setup

- **A 96-core with four Intel Xeon Gold 6252 CPUs**
  - 192 hyper-threads
  - 1.5TB main memory and 36MB*4 L3 cache
  - C++ codes compiled with clang 14.0.6 using ParlayLib with -O3 flag
- **5 implementations tested:**
  - **Ours:** SPoCH (Scalable Parallelization of Contraction Hierarchies) [ICS ’25]
  - **OSRM:** Open Source Routing Machine [GIS ’11] (parallel)
  - **CC:** an opensource parallel CH construction implementation (parallel)
  - **RK:** a C++ library that provides advanced route planning functionality [WEA ’08] (sequential)
  - **PHAST:** Hardware-Accelerated Shortest Path Trees [IPDPS ’11] (sequential)
- **Our code : https://github.com/ucrparlay/Parallel-Contraction-Hierarchy**

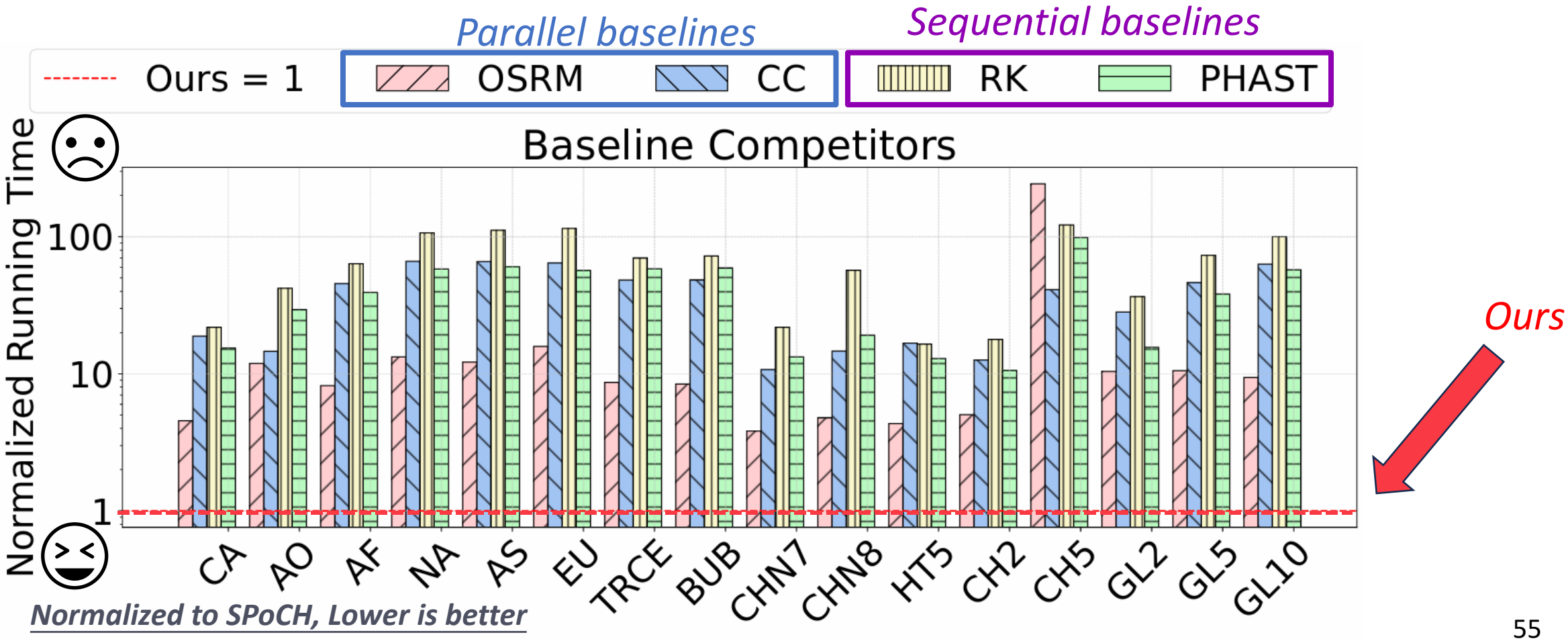
- ParlayLib: https://github.com/cmuparlay/parlaylib

Benchmark datasets

- **16 graphs tested**
  - **6 road graphs**: Central America (CA), Australia Oceania (AO), Africa (AF), North America (NA), Asia (AS), Europe (EU)
  - **4 synthetic graphs**: two path graphs and two 2D triangular meshes
  - **6 k-NN graphs**: Humidity and Temperature with k=5 (HT5), Chemical with k=2,5 (CH2,CH5),and GeoLife with k=2,5,10 (GL2,GL5,GL10)
  - 3.3 to 96 million vertices, 4.2 to 249 million edges
  - Synthetic and k-NN graphs have edge weight randomly assigned from 1 to 32
  - Road graphs have edge weight provided in the dataset

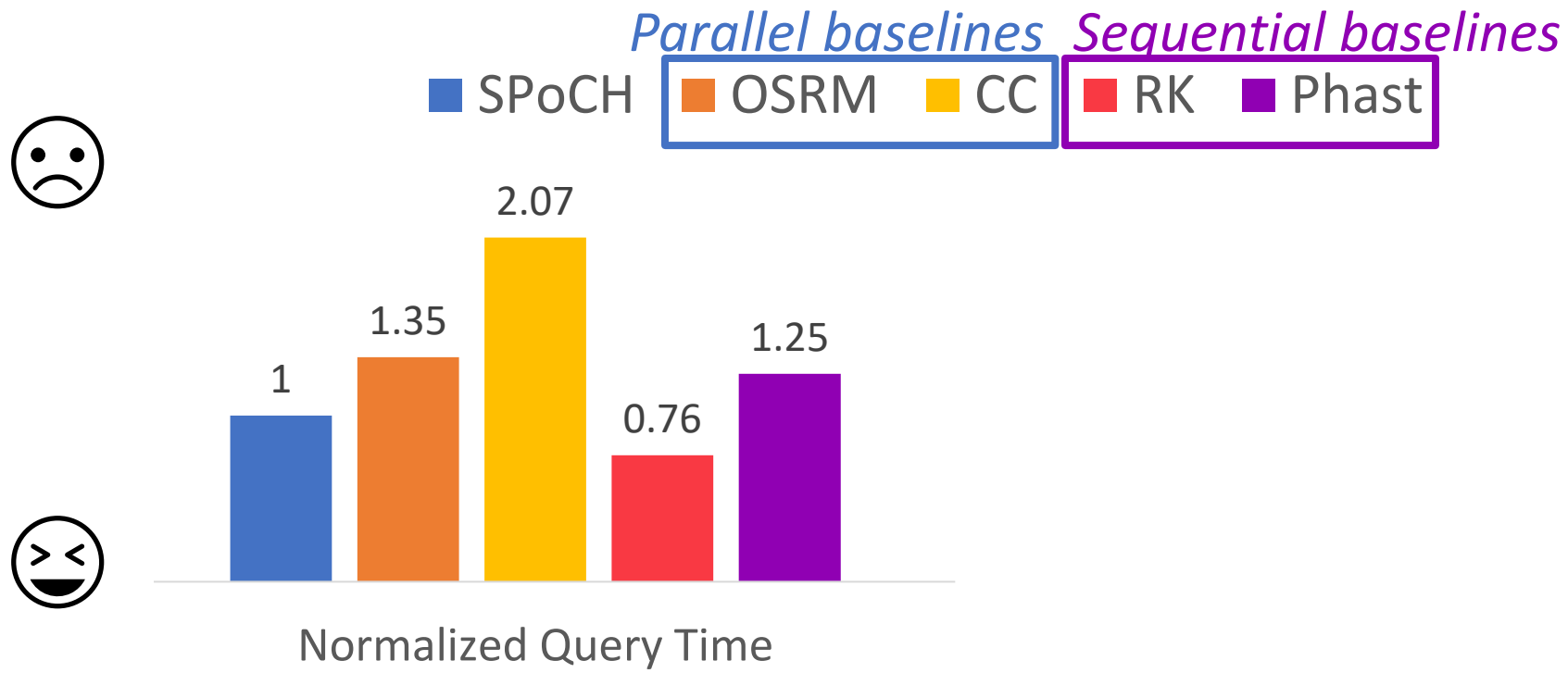
SPoCH outperforms all existing algorithms

- SPoCH achieves **11–68× speedups over the best sequential baseline**
- SPoCH achieves **3.8–41× speedups over the best parallel baseline**



SPoCH doesn't sacrifice CH quality

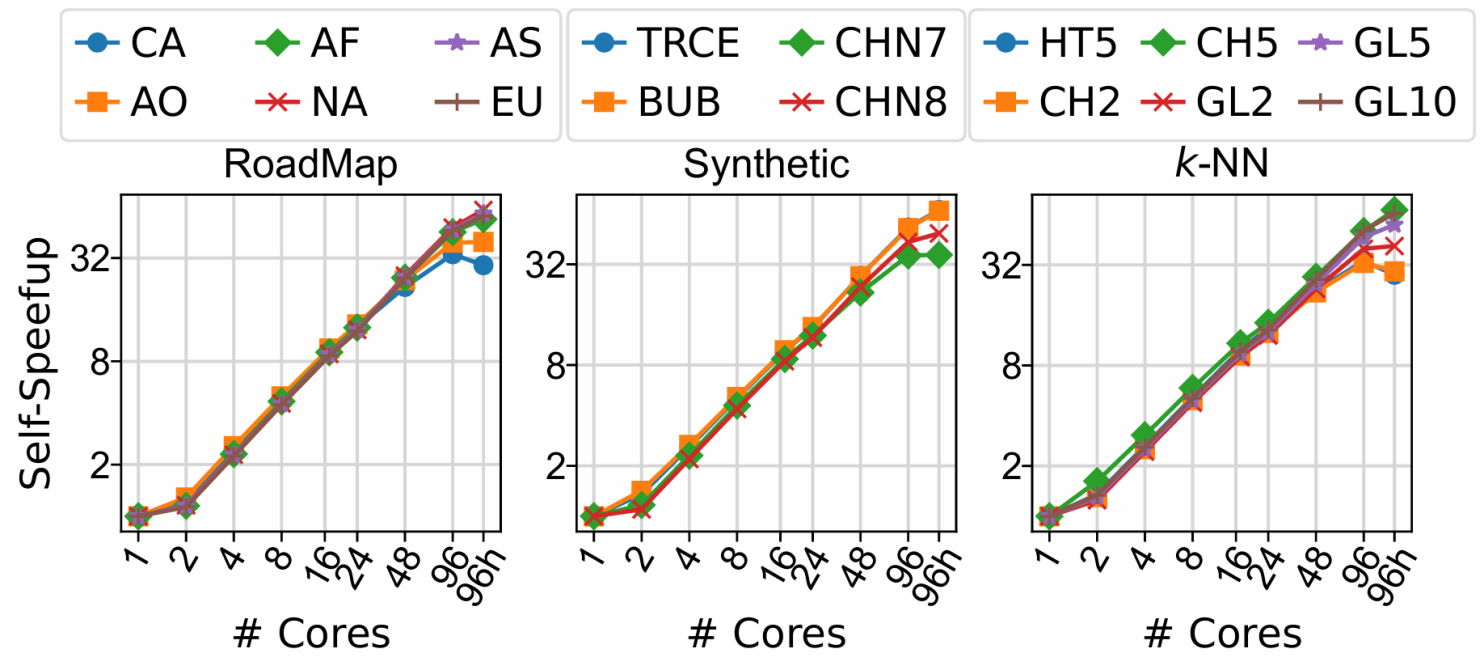
- SPoCH maintains competitive query performance
- SPoCH maintains similar space usage (number of edges in the CH)



Lower is better

SPoCH shows good self-scalability

- Self-relative speedups of SPoCH on the three graph categories
- SPoCH scales well as we increase the number of cores.



Takeaway

- Contraction Hierarchies (CH) and Vetter's parallel CH
  - CH is one of the most important algorithm for PPSP on sparse networks. It offers very fast queries, but its construction is costly.
  - Existing parallel algorithms suffer from **load balance**, **redundant computations**, and **lack of support for parallel edge updates**.
- Scalable Parallelization of Contraction Hierarchies (SPoCH)
  - SPoCH significantly **outperforms all existing sequential and parallel baselines in construction time**, without compromising CH quality.
  - The new **Preparation** step and the **source-centric strategy** for shortcut computations greatly improves parallelism and reduces work.
  - The **parallel data structure** enables efficient concurrent updates and traversal for some containers used in the algorithm.

Thanks

- Our code and dataset are publicly available here: https://github.com/ucrparlay/Parallel-Contraction-Hierarchy