Parallel Contraction Hierarchies Can Be Efficient and Scalable

Abstract

Contraction Hierarchies (CH) (Geisberger et al., 2008) is one of the most widely used algorithms for shortest-path queries on road networks. Compared to Dijkstra's algorithm, CH enables orders of magnitude faster query performance through a preprocessing phase, which iteratively categorizes vertices into hierarchies and adds shortcuts. However, constructing a CH is an expensive task. Existing solutions, including parallel ones, may suffer from long construction time. Especially, in our experiments, we observe that existing parallel solutions demonstrate unsatisfactory scalability, and have close performance to sequential algorithms.

This paper presents SPoCH (**S**calable **P**arallelization **o**f **C**ontraction **H**ierarchies), an efficient and scalable CH construction algorithm in parallel. To address the challenges in previous work, our improvements focus on both redesigning the algorithm and leveraging parallel data structures. We implement our algorithm and compare it with the state-of-the-art sequential and parallel implementations on 13 graphs, including road networks, synthetic graphs, and k-NN graphs. Our experiments show that SPoCH achieves 17–131× speedups in CH construction over the best baseline, while maintaining competitive query performance and CH graph size.

1 Introduction

Computing shortest distance is one of the most fundamental graph problems, playing a vital role in various applications, such as navigation on road networks. To accelerate point-to-point distance queries, many existing solutions use two-phase approaches, which preprocess the graph and construct an index—often an auxiliary graph—to facilitate queries. Among the most notable two-phase solutions is the *Contraction Hierarchies (CH)* [37], which is mainly designed for sparse networks such as road networks, and is widely used in practice, for instance in Google Maps. In addition to being used on its own, CH is also a vital component in other approaches for various applications on distance queries, such as Transit Node Routing [13, 15, 16], Hub-Based Labeling [6, 7, 28], and some renowned algorithms including CHASE [19] and PHAST [26]. We refer the audience to the excellent surveys [14, 50, 62] for more background of the state-of-the-art techniques for route planning.

In this paper, we consider a graph $G = (V, E)$ with an edge weight function $w : E \mapsto \mathbb{R}^+$. Given two vertices $s, t \in V$, a distance query asks for the shortest distance from s to t. The idea of CH, as illustrated in Fig. 1(a), is to "contract" the graph into a hierarchy, which is an auxiliary graph G_{CH} that preserves the shortest distances in G, such that distance queries on G_{CH} can be much faster. In each iteration, one vertex u and all its edges are moved from the original graph to an auxiliary graph G_{CH}, forming a *level* in G_{CH}. Additionally, extra edges (***shortcuts***) are added to the original graph, forming the ***overlay graph*** G_O, to preserve the pairwise distances among the remaining vertices. Finally, all vertices in G_{CH} forms the ***contraction hierarchies (CH)***. A distance query requires a bidirectional search on G_{CH}.

Empirically, a distance query on the CH touches far fewer vertices than in the original graph, thereby resulting in much faster query speed. As shown in Tab. 1, for the North American roadmap from OSM [1], queries can be 10^3–10^5 faster than directly using SSSP algorithms, such as Dijkstra [30] and Δ-stepping (a parallel SSSP algorithm) [32, 51]. However, this impressive querying speed comes at a cost—*constructing G_{CH} is expensive*. Existing sequential solutions use 22–41 minutes, and the best parallel solutions need more than 300 seconds. Such long preprocessing time may require significant computation resources, and limit their adaptability to large graphs. Such running time is reasonable for a sequential algorithm—both RoutingKit [29] and PHAST [26] take 100× construction time than Dijkstra, which is acceptable considering the speedup for queries. However, the existing parallel CH algorithms are only 4.3× faster than (sequential) PHAST even on a 96-core machine. The results indicate a significant gap, suggesting the potential for improvements in parallelism in CH construction algorithms.

Algorithms	Preprocessing time	Query time
Dijkstra [30]	-	20.8 s
Δ-stepping* [31]	-	0.538 s
RoutingKit [29]	2466 s	177 μs
PHAST [26]	1341 s	353 μs
CH-Constructor* [23]	1527 s	792 μs
OSRM* [2]	307s	163 μs
Ours*	**23.1 s**	220 μs

Table 1: Preprocessing and point-to-point shortest path query time on the road network North America [1]. The graph has 87 million vertices and 113 million edges. 1 μs = 10^{-6} s. (*) denotes parallel implementations; others are sequential. The selected target is the 50% farthest vertex from each of the sources. Dijkstra [30] and Δ-stepping [31] are sequential and parallel SSSP algorithms for comparison.

In this paper, we propose ***SPoCH*** (**S**calable **P**arallelization **o**f **C**ontraction **H**ierarchies), which supports **scalable and efficient contraction hierarchies construction with high parallelism** without compromising the query performance. The performance gain is mainly from good parallelism, plus additional algorithmic improvements for sequential performance.

Conceptually, parallelizing CH construction is not hard. In 2009, Vetter [64] pointed out that vertices in an independent set (i.e., vertices that do not share edges) can be contracted in parallel, as illustrated in Fig. 1(b). However, despite being studied in many later papers [2, 23, 44, 45, 49], we are unaware of scalable multicore CH implementations. In fact, constructing CH is a sophisticated process with intensive computation. Achieving good performance requires high parallelism in all steps, and careful algorithmic design. We observe two major challenges. First, identifying the vertices to contract involves simulating contractions on many (if not most) vertices, which calculates distances for numerous vertex pairs. Such a process is expensive, requiring high parallelism to enable good performance. The second challenge is to dynamically maintain the graph in parallel, since the graph may experience rapid changes, such as the removal of contracted vertices/edges and the insertion of shortcuts. Existing solutions may sacrifice parallelism (e.g., using locks) to support such dynamic updates. Ideally, a scalable solution would necessitate graph maintenance support with full parallelism.

Contributions. We propose SPoCH, which consists of a new algorithmic framework for CH, and a high-performance implementation [11] (anonymous for now). Our improvement lies in both *algorithm* and *data structure* design with high parallelism. Algorithmically, we introduce the new LocalSearch step in construction to address the high cost of simulated contraction on a large fraction of vertices. This step gathers all the pairs of vertices for distance computation, and processes them in a batch in parallel. Processing

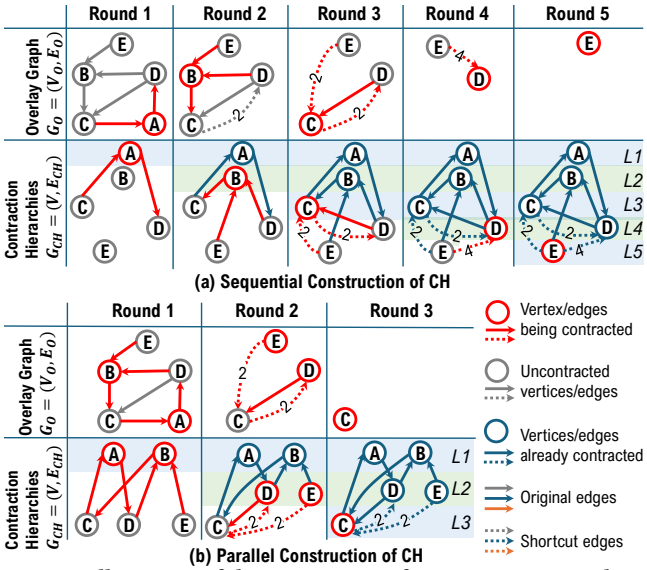

Figure 1: Illustration of the construction of Contraction Hierarchies. For simplicity, we assume the input graph has unit weights, omitting the weight "1" from the graph. The overlay graphs show the state at the beginning of each round. The contraction hierarchies show their state at the end of each round.

them as a batch allows for combining and eliminating repeated computations, offering high potential to exploit parallelism, as well as a *pruning* process to remove suboptimal shortcuts added in previous rounds. Finally, SPoCH memoizes the computed distances in this step to facilitate later computation. We provide more information in Sec. 3.1. Combining these benefits, the LocalSearch step improves both the sequential running time and parallelism.

Another key performance improvement in SPoCH comes from using efficient parallel *data structures*. As mentioned, to get a highly parallel solution, we need to efficiently handle edge updates to the graph in parallel. We use the *phase-concurrent hash table* [60] to buffer newly added shortcuts in each round, which allows for efficient lock-free concurrent insertions of new edges. However, combining these edges to the overlay graph in each round requires scanning the entire hash table array and the edge list of the overlay graph. This may be inefficient in most of the rounds where only a small number of shortcuts are added. Our solution employs a lazy update scheme to delay the combination of the shortcuts to the overlay graph, and still makes the delayed shortcuts visible to future computations. We introduce this technique in Sec. 4. This approach reduces the cost of maintaining the graph from 40.2% to 10.4% of the overall running time on average on all tested graphs.

We compare SPoCH the state-of-the-art sequential and parallel solutions on various graphs. Even the sequential running time of SPoCH is as fast as the sequential baselines. Due to our new design, SPoCH achieves high parallelism and good scalability to large numbers of processors (see Fig. 4). On North America in Tab. 1, SPoCH is 13.3–107× faster in construction than the baselines while achieving similar query performance. On the 13 graphs we tested in Tab. 4, compared to the *fastest baseline on each graph*, SPoCH is 3.63–44.1× faster, with an average of 9.46×. We also conduct in-depth performance studies of the CH algorithms in Sec. 5. For page limit, we omit some details in the main paper, and present a full version (anonymous) at [**?**].

Preliminaries. We focus on the shared-memory setting with fork-join parallelism. The computation starts with one thread. A thread

$G = (V, E)$:	The input graph
$w(u, v)$:	Weight for edge (u, v)
$G_O = (V_O, E_O)$:	The overlay graph
$P[v]$:	The score of vertex v
$N_{\text{in}}(v)$:	In-neighbors of vertex v
$N_{\text{out}}(v)$:	Out-neighbors of vertex v
$N(v)$:	In- and out-neighbors of vertex v
(Unless otherwise specified, we use $N_{\text{in}}(v)$, $N_{\text{out}}(v)$ and $N(v)$ to refer to the neighbors of v on the **overlay graph**)	

Table 2: General notations used for contraction hierarchies.

can `fork` two child software threads to work in parallel. When both children complete, the parent thread continues. Such a computation can be executed by a randomized work-stealing scheduler [12, 20, 22]. We use the atomic operation `compare_and_swap`(p, v_{old}, v_{new}), which checks if the memory location pointed by p has value of v_{old}, and if so, change the value to v_{new}. The function returns *true* if it successfully changes the value, and *false* otherwise. We also use `write_min`(p, v), which reads the memory location pointed to by p and writes value v to it if v is smaller than the current value.

We consider a weighted graph $G = (V, E, w)$ with $n = |V|$, $m = |E|$, and an edge weight function $w : E \to \mathbb{R}^+$. For $v \in V$, we define $N_{\text{in}}(v) = \{u \mid (u, v) \in E\}$ as the ***in-neighbors*** of v, and $N_{\text{out}}(v) = \{u \mid (v, u) \in E\}$ as the ***out-neighbors*** of v. We use $N(v) = N_{\text{in}}(v) \cup N_{\text{out}}(v)$. For a set of vertices $V' \subset V$, we define $N(V')$ as all neighbors of vertices in V'. Note that the main process of the CH construction algorithm is to update the *overlay graph* G_O by removing contracted vertices and incident edges, and inserting shortcuts. Therefore, when we use $N_{\text{in}}(\cdot)$, $N_{\text{out}}(\cdot)$, $N(\cdot)$ in algorithm description, we **always refer to the neighborhood of a vertex on the *overlay graph*.**

2 Contraction Hierarchies

Contraction Hierarchies (CH) [37] was proposed by Geisberger et al. in 2008, building upon the concept of highway hierarchies [46, 57]. CH is one of the most widely adopted techniques in route planning and is used in many systems, such as Google Maps.

Given a graph $G = (V, E)$, the ***Contraction Hierarchies*** $G_{CH} = (V, E_{CH})$ is an auxiliary graph that preserves the pairwise distances of G. Distance queries on G_{CH} can be orders of magnitude faster than on G itself, particularly for sparse graphs such as road networks. The CH algorithm iteratively contracts the "least important" vertex (determined by a user-specified score function) from the graph, and adds its edges to the contraction hierarchy. Additional edges (shortcuts) will be added to the remaining graph (referred to as the overlay graph G_O) to preserve the distances. Each contracted vertex is a *level* in the CH, forming a hierarchy. This process is illustrated in Fig. 1(a). A distance query from s to t is performed by running a bidirectional search from both s and t on the CH. Since this paper focuses on fast construction of CH, we present the details about the query algorithm in the (anonymous) supplemental material for completeness [**?**]. By contracting vertices and adding shortcuts, the hop distances between vertices are greatly reduced, allowing for much faster query performance.

Given its wide range of applications [5, 34, 35, 53, 56], CH has been extensively studied. In this section, we first review the algorithm proposed by Geisberger et al. [37] with the fundamental concepts. Then in Sec. 2.2 we introduce existing parallel solutions. Other related studies are discussed in Sec. 6.

Algorithm 1: Contraction Hierarchies Construction [37]

Input: A graph $G = (V, E)$.
Output: The contraction hierarchy $G_{\text{CH}} = (V, E_{\text{CH}})$.
Maintains: The overlay graph $G_O = (V_O, E_O)$
The score of each vertex $P[\cdot]$

```
1  G_O ← G, E_CH ← ∅
2  Compute the score P[·] for all v ∈ V
3  while V_O ≠ ∅ do
4      v = argmin_{v' ∈ V_O} P[v'] // Select the vertex v with lowest score from V_O
5      for u_1 ∈ N_in(v) do
6          for u_2 ∈ N_out(v) do
7              l ← WPS(u_1, u_2) // Shortest path distance from u_1 to u_2
8              if w(u_1, v) + w(v, u_2) = l then
9                  E_O = E_O ∪ {(u_1, u_2)}
10                 w(u_1, u_2) = w(u_1, v) + w(v, u_2)
11     V_A ← N(v)
12     Remove v and all its edges from G_O = (V_O, E_O) and add them to G_CH
13     Update the scores P[u] for all u ∈ V_A
14 return G_CH
```

2.1 Sequential Solutions

We present the sequential construction algorithm for CH in Alg. 1. The algorithm takes a graph $G = (V, E)$ as the input, and compute the CH for G in $G_{\text{CH}} = (V, E_{\text{CH}})$. As mentioned, the contraction process requires removing (contracting) vertices, and adding shortcuts to preserve the shortest distances. To avoid destroying the input graph, an *overlay graph* $G_O = (V_O, E_O)$ is used to reflect the changes of the graph. A score array $P[\cdot]$ is also maintained for all uncontracted vertices, representing their "importance" and deciding the contracting order.

The algorithm start by computing the initial scores of all vertices (line 2). Then, it iteratively selects and contracts the vertex with the lowest score (line 4) until the overlay graph is empty (line 3). When contracting a vertex v, the algorithm computes the shortest paths (line 7) from each $u_1 \in N_{\text{in}}(v)$ to each $u_2 \in N_{\text{out}}(v)$ by running Dijkstra's algorithm. These shortest path queries are referred to as the *Witness Path Search (WPS)* [37, 44]. If the shortest path length l from u_1 to u_2 is the same as $w(u_1, v) + w(v, u_2)$ (line 8), then v can be on the shortest path from u_1 to u_2. Thus, the shortcut edge (u_1, u_2) with weight l is added to E_O to preserve the distances on E_O (lines 9 and 10). After that, v and all its incident edges are removed from the overlay graph $G_O = (V_O, E_O)$, and are added to the CH G_{CH}. Contracting v may cause the score of all its neighbors to change, which are updated at line 13. Finally, the algorithms moves on to contract the next vertex, until all vertices are contracted.

Vertex Score and Edge Difference. Scoring the vertices is the most time-consuming part of CH construction and it significantly impacts the shortest path query performance in CH. The most widely used score function is the *edge difference* [37]. For a vertex v, the edge difference is the change in the number of edges on the overlay graph G_O after contracting v, i.e., the number of shortcuts added minus the number of edges of v. A vertex with a higher edge difference is considered more important because its contraction has a larger impact on the graph's structure. To compute the number of shortcuts to be added, a ***simulated contraction*** on v is required, which virtually generates and counts the shortcuts. This simulation is very similar to line 5–10 except it does not add edges or modify edge weights in G_O.

In existing work [37, 38], the score function is usually defined as a weighted combination of edge difference and many other measurements, such as vertex degree, hop distance, and hierarchy depth. Among them, edge difference has the highest weight and is usually the most computationally expensive part. Therefore, for simplicity, our algorithm description uses edge difference as the score function. Our algorithmic ideas are independent of the score function.

V_F:	Vertices to be contracted	$V_F = \{v : P[v] < P[u]\ \forall u \in N(v)\}$
V_A:	Vertices that need to recalculate their scores	$V_A = N(V_F)$ in the previous round
E^+:	Shortcuts to be added	
V_S:	Starting points of shortcuts	$V_S = \{v_1 : (v_1, v_2) \in E^+\}$
V_W:	Set of WPS sources	$V_W = V_S \cup N_{\text{in}}(V_S)$

Table 3: Additional notations used in our algorithm.

2.2 Existing Parallel Algorithms

The construction for CH is very costly, so it is natural to consider parallelizing this process, by contracting many vertices in parallel. However, note that we cannot contract arbitrary vertices in parallel. For instance, if two adjacent vertices u and v are contracted together, u may choose to add a shortcut incident on v. Since v is removed from the overlay graph at the same time, the shortcut may not be added correctly, resulting in distances not preserved accurately on the overlay graph. In 2009, Vetter [64] first observed that vertices that do not share edges (i.e., an *independent set*) can be contracted in parallel, and proposed the first parallel CH construction algorithm. In this case, all added shortcuts will be connecting to uncontracted vertices and are thus safe to add to the overlay graph.

Almost all later parallel solutions (e.g., [23, 49]) follow Vetter's high-level idea. In each round, an independent set (IS) of vertices is identified to be contracted. All these vertices will be placed on the same level in the CH. After that, shortcuts will be added normally. This process is repeated until all vertices have been contracted. As with the sequential algorithm that first contracts vertices with the lowest score, the vertices in the IS should also have low scores in general. In Vetter's original algorithm, the IS includes all vertices with the smallest score within its k-hop neighborhood. OSRM [2] specifically uses $k = 2$. Our implementation also follows this idea and uses $k = 1$. An existing GPU algorithm [44] and CH-Constructor [23] find a maximal independent set (MIS) and include all vertices in this MIS with scores smaller than a threshold.

Challenges to Achieve High-Performance Implementations. While Vetter's work reveals the potential parallelism in the algorithm, the idea itself is not sufficient to enable a high-performance solution. Many challenges remain in other parts of the algorithm. We highlight two components here. The first part is the process to (re)compute the scores of vertices. When multiple vertices are contracted together, a large number of simulated contractions are needed, which all involve running WPSes by Dijkstra's algorithm. The second is to update the overlay graph in parallel, since contracting vertices in parallel results in a bulk of new shortcut edges that need to be added to the overlay graph. Both parts (performing WPS and updating the graph) involve expensive computation, and require careful algorithm design and parallelization. In our experiments, we test two existing parallel solutions CH-Constructor [23] and OSRM [2, 49] based on Vetter's algorithm. In tests on a 96-core machine with 13 graphs (Tab. 4), their performance is only 0.266–7.04× faster than a highly-optimized sequential implementation.

In the next section, we present our solution that achieves a scalable and efficient parallel CH construction algorithm.

Algorithm 2: Our Parallel Algorithm for CH Construction

Input: A graph $G = (V, E)$.
Output: A contraction hierarchy $G_{\text{CH}} = (V, E_{\text{CH}})$
Maintains: The overlay graph $G_O = (V_O, E_O)$. Initially $G_O = G$.
The score of each vertex $P[\cdot]$.

1 $E_{\text{CH}} \leftarrow \emptyset$; $V_A \leftarrow V$; $V_W \leftarrow V$
2 **while** $V_O \neq \emptyset$ **do**
*// **Step 1**: Run witness path searches (WPSes) from all vertices in V_W to prepare for score recomputation for all vertices in V_A*
3 LocalSearch(V_W, V_A, G_O)
*// **Step 2**: For each vertex $v \in V_A$, (re)compute its score $P[v]$*
4 Score(V_A, G_O)
*// **Step 3**: Select an independent set of vertices $V_F \subseteq V_O$ to be contracted based on the score array $P[\cdot]$*
5 $V_F \leftarrow$Select(G_O, P)
*// **Step 4**: Contract vertices in V_F; update E_{CH} and G_O accordingly; return LSM as the sources of WPSes in the next iteration*
6 $V_W \leftarrow$Contract(V_F, G_O)
7 $V_A \leftarrow N(V_F)$ *// V_A in the next round includes all neighbors of V_F*
8 **return** Postprocess(V, E_{CH}) *// reindexing*

3 Our Parallel CH Construction Algorithm

In this section, we present SPoCH for generating contraction hierarchies $G_{\text{CH}} = (V, E_{\text{CH}})$ in parallel. As with other parallel CH algorithms, SPoCH also follows the high-level idea from Vetter [64] that in parallel contracts a batch of vertices in an IS. To achieve high parallelism without sacrificing the CH quality, SPoCH uses several novel key techniques that will be discussed in this section.

We present the pseudocode of SPoCH in Alg. 2. It takes a graph $G = (V, E)$ as the input, and returns the contracted graph G_{CH}. Note that vertex set in G_{CH} is the same as in the input graph, the algorithm only needs compute the edge set E_{CH} of the contracted graph. Similar to Alg. 1, our algorithm also maintains the overlay graph $G_O = (V_O, E_O)$ to reflect the changes of the original graph due to contractions.

The main loop of our algorithm (the while-loop in line 2) repeatedly finds an IS of vertices and contract them. We summarize them in five steps. We will briefly outline the high-level idea here, and then elaborate on each step.

One of the key insights in SPoCH is the introduction of the LocalSearch step on line 3. Unlike existing solutions, introducing this step can reduce redundant WPSes, getting more parallel tasks, which overall leads to less computation and better parallelism. With the details of this step described in Sec. 3.1, the overall goal of this step is to preprocess the current overlay graph by running WPSes from a set of vertices V_W, in order to (re)compute the score for a set of vertices V_A. At the beginning, both V_W and V_A are V. In later rounds, V_W will be computed by the Contract step in the previous round, and V_A are the neighbors of vertices contracted in the previous round. In Vetter's algorithm, WPSes are directly performed when recomputing scores in each round. SPoCH separates out this part with new designs, and further utilizes the results for more optimizations.

The next two steps Score and Select will (re-)calculate the score for vertices in V_A, and choose an independent set of vertices V_F to contract. To ensure that selected vertices have low scores, V_F includes all vertices with the lowest score in its neighborhood. Importantly, the Score step makes use of the result from the LocalSearch step to avoid redundant computation.

Finally, the Contract step performs the contraction and updates the overlay graph, which is another major improvement over existing solutions. In this step, all incident edges to V_F are moved to G_{CH}, and the corresponding shortcuts are added to G_O. Unlike existing solutions that sequentialize this process, all these updates in SPoCH are highly parallel. We introduce the algorithmic step in Sec. 3.4, and discuss the parallel data structure support in Sec. 4. In addition, to facilitate the LocalSearch step, the Contract step will generate the set of vertices V_W (the parameter used in the LocalSearch in the next round), which are all vertices from which we will start WPS.

We summarize the notations in our algorithm in Tab. 3. Next, we will elaborate on each step in Alg. 2. Later in Sec. 4, we discuss using efficient data structures to support this algorithm.

3.1 Step 1: Local Search with Memoization

Algorithm 3: Parallel Local Search

1 **Function** LocalSearch$(V_W, V_A, G_O = (V_O, E_O))$
2 **ParallelForEach** s *in* V_W **do**
3 $V_n = \emptyset$ *// the set of vertices that need to be settled by s*
4 **foreach** $u \in N_{out}(s) \wedge u \in V_A$ **do**
5 **foreach** $v \in N_{out}(u) : (v \neq s) \wedge ((s, v) \notin dist)$ **do**
6 $V_n.insert(v)$
7 **[WPS from s]** Run Dijkstra's algorithm starting from s. Terminate the algorithm either all $v \in V_n$ are settled, or θ nodes are settled. Add the shortest distance w from s to all $v \in V_n$ into parallel hashmap $dist$, so that $dist[(s, v)] = w$
8 **foreach** v: $(s, v) \in E_O$ **do**
9 **if** $w(s, v) > dist[(s, v)]$ **then** delete edge (s, v) from E_O

One improvement of SPoCH lies in the stand-alone Local Search with Memoization (LocalSearch) step at the beginning of each round, presented in Alg. 3. This step runs witness path searches (WPSes) to prepare the shortest distances needed during scoring (Sec. 3.2) and contracting (Sec. 3.4). As mentioned in Sec. 2.1, to compute the score of a vertex $v \in V_A$, a simulated contraction on v is needed to determine the shortcuts that would be added if we contract v. For each pair of vertices (u_1, u_2) where $u_1 \in N_{\text{in}}(v)$ and $u_2 \in N_{\text{out}}(v)$, if v is on the shortest paths from u_1 to u_2, then a shortcut is needed. We also use the WPS as described in Sec. 2.1, which runs Dijkstra from u_1 until u_2 is settled to find the shortest distance between them.

Running WPSes is crucial to determine score of a vertex. However, it is also expensive—to compute the edge difference for a vertex v, WPSes are needed on all pairs of $N_{\text{in}}(v) \times N_{\text{out}}(v)$ (Cartesian product). Fully finishing WPSes for all pairs may be expensive, especially when there exists one pair of vertices that are extremely far away. Therefore, most existing systems (including ours) also limit the size of each WPS to be θ, i.e., each Dijkstra's algorithm only search for θ iterations. In this case, some suboptimal shortcuts may be added and make the CH (necessarily) larger. To improve the overall performance, we highlight three key techniques in our LocalSearch step: batching, memoization, and pruning.

Batching. Our first technique is to collect all WPSes needed and process them in a batch. In SPoCH, the Contract step generates a vertex set V_W, which serve as sources of WPSes in the next round. By maintaining them in a batch, we process all WPSes in a batch to avoid redundant computation and achieve better parallelism. For example, if two simulated contractions triggers WPSes from the same vertex, they will be performed only once in the batch. Executing all WPSes as one batch allows us to optimize them as a

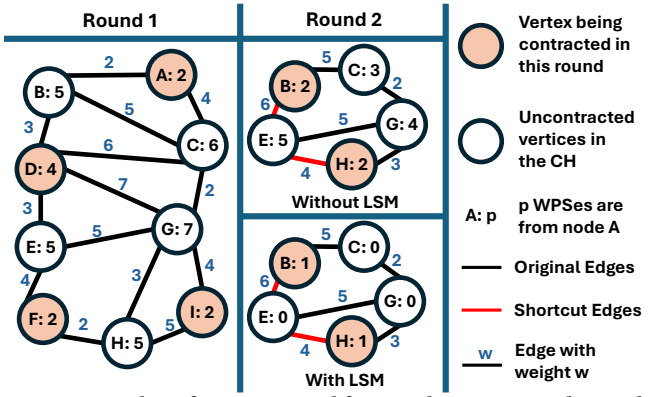

Figure 2: Number of WPSes started from each vertex in each round. Lower is better. In Round 1, the number of WPSes doesn't include LSM optimization. With LSM applied, only one WPS is performed on each vertex. In Round 2, without LSM, the number of WPSes per vertex equals the sum of the neighbors requiring re-scoring and the neighbors to be contracted. For example, in round 2, all three of E's neighbors (B, G and H) require re-scoring as they all have neighbors (D and F) contracted in round 1. Additionally, B and H are to be contracted. Thus, E requires a total of five WPSes in round 2. With LSM, only vertices B and H have each other as new 2-hop neighbors, while all other 2-hop neighbors have their relative distance calculated and memoized in round 1. Thus, only 1 WPS is required for both B and H.

whole, and provides greater opportunity to exploit parallelism.

Memoization. SPoCH uses memoization to avoid computing the shortest distance on the same vertex pair multiple times. In SPoCH, a map *dist* is used to memoize pairwise shortest distances computed in previous WPSes. If a WPS is required on the same pair again, we directly use the memoized results without running another WPS. This information is reused in pruning (described below) and later steps Score and Contract.

Pruning. A byproduct of memoizing the distances in *dist* is to prune redundant edges in the graph. Note that a WPS uses Dijkstra's algorithm, which always computes the true shortest distance between vertices. As the algorithm proceeds, more pairs of vertices (u, v) obtain their true shortest distance $d(u, v)$ by WPSes. If there is an edge between u and v in the current overlay graph (possibly a suboptimal shortcut added in previous rounds) with weight $w > d(u, v)$, we will delete the edge since it is redundant. This optimization effectively reduces the size of the overlay graph, which potentially improves the performance for both construction and query.

The LocalSearch Step. With the three techniques, we now describe the LocalSearch step. The step takes V_W, V_A and G_O as input, and performs WPSes from all vertices in V_W to recompute the scores for all vertices in V_A on the overlay graph G_O. For each WPS from s, we first identify the set of vertices V_n that need to be settled by s. On line 3–6 in Alg. 3, for each $u \in N_{\text{out}}(s)$, if $u \in V_A$, this means that s is a source of WPS because s is an in-neighbor of an affected vertex u. To compute the new score for u, the shortest distances from s to each vertex v as an out-neighbor of u need to be computed. Therefore, we collect $N_{\text{out}}(u)$, the out-neighbors of u, to the set V_n. Note that s can correspond to multiple affected vertices u. In this case, all of their out-neighbors will be gathered in V_n, and only incur one search starting from s. Before adding a vertex v to V_n, we also check if (s, v) has already been memoized in *dist*. If so, v will be skipped.

With all target vertices of WPS originating from s gathered in V_n, We run Dijkstra's algorithm from s to search all vertices in V_n. During the algorithm, the shortest distances from s to each $v \in V_n$ are inserted to *dist*. As mentioned, a WPS terminates either when all vertices in V_n have been settled, or when θ vertices have been settled (so that the overall cost is limited). If the WPS terminates before a vertex $v \in V_n$ is settled, the distance between s and v will be viewed as ∞ and a shortcut between s and v is always generated.

3.2 Step 2: Score

Algorithm 4: Parallel Node Scoring

```
1 Function Score(V_A, G_O = (V_O, E_O))
2     ParallelForEach u in V_A do
3         P[u] ← −|N(u)|
4         foreach v_1 ∈ N_in(u) do
5             foreach v_2 ∈ N_out(u) do
6                 if w(v_1, u) + w(u, v_2) = dist⟨v_1, v_2⟩ ∨ (v_1, v_2) ∉ dist
                    then P[u] ← P[u] + 1
```

The second step computes the scores for each vertex $u \in V_A$, as presented in Alg. 4. In the previous step, we have memoized the distances in the neighborhood of each $u \in V_A$ in a map *dist*. To recompute the score of $u \in V_A$, we first set $P(u)$ to the negative of its number of direct neighbors, i.e., $-|N(u)|$, since $|N(u)|$ edges can be removed from G_O if u gets contracted (line 3). Then, we iterate through each pair of in-neighbors and out-neighbors of u and check if their distance through u is equal to or smaller than the shortest distance stored in *dist*. If so, we add one to $P[u]$ since one shortcut needs to be added.

For simplicity, our psuedocode only computes the edge difference, which is the most important component of the score for each vertex. In practice, many other components are considered in existing work. SPoCH uses a similar score function as in previous work, which is a combination of edge difference, vertex degree, hop distance, and hierarchy depth.

3.3 Step 3: Select

Algorithm 5: Parallel Node Selection

```
1 Function Select(G_O = (V_O, E_O))
2     V_F ← ∅
3     ParallelForEach u in V_O do
4         if (∀v ∈ N(u), P(u) < P(v)) then V_F.insert(u)
5     return V_F
```

The Select step identifies a subset of vertices V_F to contract in parallel, as presented in Alg. 5. We aim to contract multiple vertices in an independent set (i.e., do not share edges) simultaneously while prioritizing those with lower scores. To do this, we select all vertices that has the minimum score in its neighborhood. As shown in line 2–4 in Alg. 5, we process each vertex u in V_O. If u has the lowest score among its neighbors, we insert u into a set V_F. To handle equal scores, we use the label of each vertex to break ties. This approach can find an independent set so that vertices with lower scores are contracted before those with higher scores. Fig. 1(b) presents an example of contracting a graph in parallel. In each round, multiple independent vertices can be contracted together, and the levels can be determined accordingly. The final graph preserves the shortest distance between any two vertices.

Selecting an independent set of vertices ensures that shortcuts are always established between non-contracted vertices. The process is also reasonably fast for contracting vertices. In our exper-

iment, we observe that usually within 10–20 rounds, more than 99% vertices can be contracted. This indicates the high potential of parallelism in this algorithm.

3.4 Step 4: Contract

Algorithm 6: Parallel Node Contraction

```
1  Function Contract(V_F, G_O = (V_O, E_O))
2      V_S ← ∅
3      ParallelForEach u in V_F do
4          Remove u from V_O
5          foreach v ∈ N_in(u) do E_CH↓.insert(⟨v, u, w(v,u)⟩);
6          foreach v ∈ N_out(u) do E_CH↑.insert(⟨u, v, w(u,v)⟩);
7          for v_1 ∈ N_in(u) do
8              for v_2 ∈ N_out(u) do
9                  if w(v_1,u) + w(u,v_2) = dist[(v_1,v_2)] ∨ (v_1,v_2) ∉ dist
                     then
10                     V_S.insert(v_1)
11                     E+.insert(⟨v_1, v_2, w(v_1,u) + w(u,v_2)⟩)
12         Remove the incident edges of u from E_O
13     E_O = E_O ⋃ E+
14     V_W ← ∅
15     ParallelForEach u in V_S do
16         V_W.insert(u)
17         ParallelForEach v ∈ N_in(u) do
18             V_W.insert(v)
19     return V_W
```

With the independent set V_F computed in the previous round, the last step Contract performs the actual contraction, as given in Alg. 6. This step moves all vertices in V_F from V_O to the CH, adds incident edges to E_{CH}, computes and adds the relevant shortcuts to E_O, and finally generates the set V_W as the sources for running WPSes in the next round, which will be used in the next LocalSearch step.

In Alg. 6, we process each vertex $u \in V_F$ in parallel. We first remove each vertex $u \in V_F$ from the overlay vertex set V_O in line 4. Next, as shown in line 4–6 in Alg. 6, we move the incident edges of u to E_{CH}, which contains two subsets: ${E_{\text{CH}}}^{\downarrow}$ and ${E_{\text{CH}}}^{\uparrow}$. When moving a vertex u to the CH, its incoming edges are stored in ${E_{\text{CH}}}^{\downarrow}$, and outgoing edges are stored in ${E_{\text{CH}}}^{\uparrow}$. This separation is necessary because during the shortest path queries, the search from the target only proceeds "backward" by following incoming edges in ${E_{\text{CH}}}^{\downarrow}$ while the search from the source only moves "forward" by following outgoing edges in ${E_{\text{CH}}}^{\uparrow}$. Both searches move "upward" in the CH.

Finally, as shown in line 7–11 in Alg. 6, for each vertex u to be contracted, we enumerate each in-neighbor $v_1 \in N_{\text{in}}(u)$ and out-neighbor $v_2 \in N_{\text{out}}(u)$. We establish a shortcut between v_1 and v_2 if necessary. This is performed by comparing $w(v_1, u) + w(u, v_2)$ with the shortest distance stored in *dist*. Recall that *dist* is a map generated in the LocalSearch step, which buffers shortest distances for relevant vertex pairs. If $w(v_1, u) + w(u, v_2) = dist[(v_1, v_2)]$, then the shortest path between v_1 and v_2 can be via u. A special case is when (v_1, v_2) is not in *dist*, which means that the shortest distance between them was not computed in the LocalSearch step. In both cases, before contracting u, a shortcut needs to be established between v_1 and v_2 with weight $w(v_1, u) + w(u, v_2)$ (line 9). In our implementation, we use the parallel hash map E^+ in Sec. 4.1 to buffer all shortcuts added in this round and later combine it with E_O on line 13. In Sec. 4.2 we will describe other optimizations to reduce the cost to combine E^+ and E_O. Finally, we can also remove all edges incident on $u \in V_F$ from E_O after adding shortcuts. This is because these edges no longer contribute to the reduction of distances to any vertices that remain uncontracted.

During the contraction step, we also generate the set V_W that contains all sources for the WPSes in the next step. When contracting a vertex u, we check all $v_1 \in N_{\text{in}}(u)$ and $v_2 \in N_{\text{out}}(u)$ to see whether a shortcut from v_1 to v_2 needs to be established. If so, the scores for both v_1 and v_2 are affected, since both have their neighborhoods changed. To reconsider the score of v_2, WPSes are needed from v_1 (to find the distance from the new in-neighbor v_1 to all $N_{\text{out}}(v_2)$). To reconsider the score of v_1, WPSes are needed from all in-neighbors of v_1 (to find their distances to the new out-neighbor v_2). Hence, we first use a set V_S to buffer all vertices v_1. At the end of this step, we construct the set V_W, which includes all vertices in V_S and all in-neighbors of them. This set will be used as the sources for the WPSes in the LocalSearch step in the next round.

3.5 Postprocessing

After finishing the contraction of all vertices, we re-index and sort these vertices by their level and score to improve query performance. In the original sequential CH algorithm [37], vertices are reordered based on the order to contract them. In PHAST [3], vertices are re-grouped by levels to perform SSSP queries on CH, where vertices at lower levels always have a lower rank than those in higher levels.

We combine the re-indexing methods from both algorithms and we use vertex scores to approximate the contraction order. For any two vertices, if they are at the same level, they are sorted by score from the lowest to the highest. Otherwise, they are sorted by level from the lowest to the highest.

This re-indexing ensures that the vertices are efficiently organized for query processing, where paths are traversed based on the contraction rank. Once the vertices are re-indexed, we generate the final CH from ${E_{\text{CH}}}^{\uparrow}$ and ${E_{\text{CH}}}^{\downarrow}$.

4 Parallel Data Structures

In Sec. 3, we have presented the algorithmic idea of SPoCH. We note that, however, to support high performance in practice, the implementation relies on efficient data structures to support the algorithm, especially in supporting dynamic updates of the overlay graph. Note that the (overlay) graph is highly dynamic—as the contraction proceeds, the graph consistently undergoes vertex removals (removing contracted vertices), edge removals (removing edges incident on contracted vertices), and edge insertions (adding shortcuts). All of them require high parallelism to enable good performance for the overall CH algorithm. This section provides an overview of the parallel data structures used in our implementation.

Especially, to facilitate graph traversing, the overlay graph is usually maintained by an array-based structure (e.g., the CSR introduced below), which maintains the neighbors of the same vertex in a consecutive chunk in the memory. When multiple vertices are contracted in parallel, shortcuts are also generated by multiple threads. Inserting them into the overlay graph in a concurrent manner is therefore challenging. In existing implementation, OSRM avoids such complication by only contracting vertices with the minimum score in its 2-hop neighborhood. In this case, for a vertex v in the overlay graph, only one contracted vertex may update the neighbor list of v, and this will be performed sequentially. However, by restricting all contracted vertices to be at least two hops away, OSRM can choose much fewer vertices in the independent set in each round. In our experiments, we observe that such re-

striction greatly increased the number of rounds and degraded the performance. In CH-Constructor, a lock-based data structure is used, which also limits parallelism in this step. In this section, we introduce the parallel data structures used in our implementation, such that they support updating the overlay graph in an efficient and flexible way.

4.1 Background of the Used Data Structures

In our implementation, we mainly use two data structures to maintain graphs: Compressed Sparse Row (CSR) [63] and phase-concurrent hash table [60]. In this section, we introduce the background of these two data structures and their interface. Then in Sec. 4.2, we describe how they are used to support our algorithm.

Compressed Sparse Row (CSR) [63]. CSR is an array-based adjacency list format consisting of an `edge` array of size $|E|$ and an `offset` array of size $|V|$. The `edge` array is a concatenation of neighbor lists (along with the corresponding edge weights) for all vertices. Therefore, the neighbor list for each vertex is always contiguous. The `offset` array stores the start position of each vertex's neighborhood in the `edge` array. CSR provides *fast lookups with good cache locality* due to its contiguous memory layout. However, it does not support insertion or deletion without rebuilding the entire edge and offset arrays. Our input graph is given in the CSR format.

Phase-Concurrent Hash Table [60]. The phase-concurrent hash table is an array-based structure that supports efficient operations for an unordered set, including insertions, deletions, and lookups. We use it to support multiple variables in SPoCH.

Unlike traditional concurrent hash tables that allow arbitrary operations of any type to run concurrently, the phase-concurrent hash table only permits operations of the same type to proceed simultaneously. This design ensures both *serialization* and *determinism* (w.r.t. execution order). To insert a key-value pair (k, v) into the hash table H, the key k is first hashed to a random index in H. If this index is occupied, linear probing is performed to find the next available slot. Once an empty index is found, it swaps in the new record using an atomic operation `compare_and_swap`. When multiple threads attempt to modify the same memory location, the atomic operation guarantees that only one thread succeeds. All other threads will fail in the operation and continue linear probing. A special case for insertion is to update the value for the k if it already exists in the hash table. In our implementation where the value can refer to the weight of a given edge, we update the value if the newly inserted value is smaller than the current value stored in the hash table. In particular, when inserting (k, v), if we find that k is already in the table during linear probing, we use an atomic operation `write_min` to update the value and keep the lower one. The atomicity of this operation guarantees that only the minimum value among all concurrent modifications is retained.

Similarly, to look up a record with key k, the hash table hashes k to an initial index i. Then, three cases are considered: 1) If $H[i]$ is empty, then k does not exist in the hash table. 2) If the key at $H[i]$ is k, then the record with key k has been found. 3) If the key at $H[i]$ is not k, the algorithm uses linear probing to continue searching for the key in subsequent indices, until k is found or an empty slot is encountered. To delete a record with key k, the algorithm first locates k in H. If k is present, the corresponding index is marked as a tombstone, which is not treated as empty but will not match any existing keys. If k is not found in H, no further action is needed.

For simplicity, we will refer to the phase-concurrent hash table as the *hash table* in the following part of this section.

4.2 Using Parallel Data Structures in SPoCH

Maintaining the Overlay Graph Dynamically. To facilitate graph traversal, the overlay graph $G_O = (V_O, E_O)$ is maintained using a CSR, such that getting the neighbor list for each vertex is always efficient. The challenge is then to efficiently reflect edge insertions to the static CSR structure.

To efficiently support these updates in the overlay graph, we use an auxiliary edge set E^+ to tentatively maintain the newly added edges. Recall that when a node u is contracted, pairwise edges from $v_1 \in N_{\text{in}}(u)$ to $v_2 \in N_{\text{out}}(u)$ need to be added to the overlay graph if the shortest path from v_1 to v_2 passes through u. Instead of adding them directly to the CSR of the overlay graph, we first use E^+ to buffer them. Once a shortcut is generated (line 11 in Alg. 6), we directly add the new edge to E^+. However, adding these edges to E^+ in parallel is challenging due to contention—although contracting two adjacent vertices is avoided by selecting an IS, it is possible for multiple vertices to be contracted share the same neighbor. In this case, new edges will be concurrently added to this neighbor. Moreover, edges with the same endpoints (u, v) could be added to the overlay graph by two different contracted vertices c_1 and c_2 (i.e., via u–c_1–v and u–c_2–v). Among all the same edges, only the one with the smallest weight should be kept.

To support these updates in the overlay graph, we use a parallel hash table to maintain E_O. Once a shortcut (v_1, v_2) with weight w is generated, we insert the triple (v_1, v_2, w) to the hash table. To insert an edge (v_1, v_2, w), we first check whether the edge (v_1, v_2) exists in the hash table. If so, we use `write_min` to update the weight atomically if w is smaller than the stored weight. Otherwise, we insert (v_1, v_2, w) to the hash table. Here we use v_1 as the key, and we will explain the design choice later in this section.

Note that even if we first collect shortcuts in E^+ and combine them with E_O to avoid frequently updating the CSR, it can still be expensive to do so every round. Especially in later rounds, only a few vertices are contracted, resulting in a small number of shortcuts to add. In this case, updating the CSR for adding a few edges is inefficient. To handle this issue, we combine the two sets in a lazy manner. In particular, we do not clear and combine E^+ with E_O until E^+ gets overloaded (i.e., a constant fraction of the slots are taken). We set the hash table size to the number of edges in the initial CSR. In this case, we can guarantee that when updating the CSR, there must be a comparable number of edge insertions, and thus the cost can be amortized.

However, the belated combination of E^+ and E_O also requires all edges in E^+ to be visible to later graph traversal operations. To do this, when inserting an edge (v_1, v_2, w), we use v_1 as the key and (v_2, w) as the value. Using only the first endpoint as the key allows us to look up the neighbors of a given vertex by linear probing, and also has better cache locality. When traversing all neighbors of u in the overlay graph, we first process all neighbors of u in the CSR E_O. We then further look up u in the hash table for E^+, and use linear probing to get all edges incident on u. These gives the additional neighbors of u w.r.t. shortcuts associated with u that have not yet been incorporated into E_O.

To identify when the hash table is overloaded, at the end of each round, we use random sampling to estimate the size of the hash table. If the hash table is filled up by a constant fraction, a merge of E^+ with E_O is triggered. During this process, we also remove all edges incident to the vertices that have been contracted. In other

words, the edge removals in line 12 of Alg. 6 are also performed lazily until the next update on E_O. Finally, we clear the hash table.

The final edges in the CH, ${E_{\text{CH}}}^{\uparrow}$ and ${E_{\text{CH}}}^{\downarrow}$ are also maintained by the phase-concurrent hash tables. For both of them, only the insertion operation is needed, which can be performed concurrently using atomic operations.

Maintaining the Distance Mapping dist. Another important data structure used in our algorithm is *dist*, which memoizes shortest distances for certain pairs of vertices discovered in WPSes. In our implementation, we also use a phase-concurrent hash table. When a shortest distance between u and v is computed as w, we add (u, v, w) to the hash table, where (u, v) is the key and w is the value. By setting up the mapping in the LocalSearch step, later Score and Contract steps can directly look up the distance instead of performing another simulated contraction.

Similar to the shortcut set E^+, the number of elements in the *dist* map can also change dramatically in different rounds. When most of the slots in the hash table are empty, clearing the map becomes an expensive task in each round. To address this issue, similar to how we maintain E^+, we clear the hash table for *dist* lazily. At the end of the round, we use random sampling to estimate the load factor of the hash table. Only when a constant fraction of the hash table is full do we clear the hash table.

5 Experiments

5.1 Experimental Setup

Environment. We run our experiments on a 96-core (192-hyperthreads) machine with four Intel Xeon Gold 6252 CPUs and 1.5TB of main memory. We implement all algorithms in C++ using ParlayLib for fork-join parallelism and parallel primitives (e.g., sorting). We use `numactl -i all` for parallel tests to interleave the memory pages across CPUs in a round-robin fashion.

Benchmark Dataset. To evaluate the performance of CH algorithms, we create a benchmark dataset containing 13 directed graphs. This dataset includes four road graphs, four synthetic graphs and five k-NN graphs.

- **Road graphs** are the primary use application domain for CH algorithms. Here we use Europe (EU), North America (NA), Asia (Asia), and Africa (AF) from Open Street Map (OSM) [1]. The edge weights are from OSM, which are up to 2^{25}.
- **Synthetic graphs** with relatively low average degree and high diameter are selected because these characteristics are ideal for testing the performance of CH algorithms. Here we use hugebubbles-0020 (BUB) and hugetrace-0020 (TRCE) from [58], which feature 2D adaptively refined triangular meshes. We assign random weights for BUB and TRCE from 0 to 10^5. Chain graphs have 10^7 (CHN7) and 10^8 (CHN8) vertices, and we assign random weights from 1 to 32. Whenever a vertex is contracted in the chain, exactly one shortcut is added. Therefore, chain graphs are primarily used to test the cost related to non-scoring steps.
- ***k*-NN graphs** are used to evaluate the performance among different average degrees. In k-NN graphs, each vertex has k out-going edges pointing to its k-nearest neighbors, excluding itself. As k increases, the complexity of the scoring process in CH construction also increases. We use Chemical with $k = 2, 5$ (CH2, CH5) [33, 65], and GeoLife with $k = 2, 5, 10$ (GL2, GL5, GL10) [65, 70]. Edge weights are randomly assigned from 1 to 32.

When we compare the *average* performance across all graphs, we always report the *geometric mean* number across all graphs.

Baseline Competitors. We call all existing algorithms that we compare to as the ***baselines***. Our implementation is based on the framework in Sec. 3. We compare our implementations with two of the most well-received sequential implementations (RoutingKit and PHAST) and two state-of-the-art parallel open-source implementations (CH-Constructor and OSRM), described as follows.

- **OSRM** [2, 49]: A parallel routing engine designed to calculate the fastest routes between locations in the OpenStreetMap [1].
- **CH-Constructor (CC)** [23]: A parallel CH construction implementation. As shown in Fig. 5, CC always achieves the best performance with around 8 threads. Hence, the reported construction time for CC is on 8 threads instead of 192 threads.
- **RoutingKit (RK)** [4, 29]: An efficient sequential C++ library that applies the CH for fast route planning.
- **PHAST** [3, 26]: A widely used algorithm for parallel single-source shortest paths (SSSP) using CH. Its implementation includes a sequential process for constructing the CH, and a parallel process for running SSSP. In our experiment, we only compare to their construction time, which is sequential.

The basic frameworks of both parallel baselines (CC and OSRM) are based on Vetter [64], which supports node scoring and contracting in parallel. Each time they score a vertex, both algorithms always perform witness searches on all of its neighbors, and they do not record any distance information from these searches. This means that if scoring two vertices involves performing WPSes on the same vertex, both implementations will perform this search twice, which greatly increase the total computation.

Since our main focus is in the performance of constructing CH, we obtain the CH generated from all baselines, and always use the same query algorithm on these CH structures to enable a fair comparison of the CH quality. We choose to use the query algorithm in RK since we observe that it has the best performance in general.

5.2 Overall Performance

To show the efficiency of our algorithm against established baselines, we conducted experiments to compare performance in terms of construction time and query time, which is used as an indicator of graph quality. Tab. 4 presents a comparison among SPoCH and all baselines. In Fig. 3, we normalize the performance of all baselines to that of SPoCH (with SPoCH's numbers always set to 1). For construction, we also include the sequential running times of all parallel implementations to reflect the total work in Tab. 4. For queries, for each graph, we randomly select 100 pairs of vertices, run these queries sequentially, and report the average time and the average number of vertices processed in the queries. As mentioned, we apply the same query algorithm to all baselines. Therefore the query cost can be used as an indicator of the CH quality. Note that a sequential algorithm always chooses the best node to contract in each iteration, while a parallel algorithm may choose to contract multiple vertices together. Conceptually, this means that many of the vertices are contracted earlier than they should be. Therefore, the quality of parallel CH construction may be sacrificed for parallelism.

In the supplemental material, we also report the number of edges in the CH and the number of vertices processed in queries for all baselines. The number of edges in CH reflects the space usage for storing the CH. In general, we observe that all implementations generate similar numbers of edges in the CH. On almost all graphs, the difference is within 10%. The numbers of vertices processed in

	Graph	Road				Synthetic				k-NN				
		AF	NA	AS	EU	TRCE	BUB	CHN7	CHN8	CH2	CH5	GL2	GL5	GL10
	# vertices	33.5M	87.0M	95.7M	131M	16.0M	21.2M	10.0M	100M	4.21M	4.21M	24.9M	24.9M	24.9M
	# edges	44.8M	113M	123M	169M	48.0M	63.6M	10.0M	100M	8.42M	21.0M	49.8M	124M	249M
Build (s)	**Ours***	**7.32**	**23.1**	**23.5**	**32.9**	**27.2**	**38.8**	**1.87**	**15.2**	**0.89**	**107**	**3.00**	**9.67**	**41.1**
	OSRM*	60.0	307	286	521	218	292	6.77	71.2	4.38	28028	30.7	103	391
	CC*	334	1527	1548	2121	1218	1680	19.0	218	11.0	4737	83.3	453	2627
	RK	466	2466	2619	3783	1765	2509	38.7	848	15.5	14048	108	716	4175
	PHAST	287	1341	1421	1865	1470	2053	23.5	285	9.20	11361	46.0	373	2388
	Ours-seq	386	1365	1354	1847	1816	2620	65.4	751	26.1	7454	125	545	2676
	OSRM-seq	503	3246	2839	4101	3817	5273	34.4	390	18.3	>30000	82.4	1062	12082
	CC-seq	553	2907	2604	3574	3658	5002	35.1	423	20.5	17268	133	812	6855
Query (μs)	**Ours***	18.0	101	62.3	172	421	514	5.65	7.39	3.50	253	1.81	10.5	34.0
	OSRM*	28.7	163	97.6	223	454	512	7.94	9.74	3.84	325	2.20	12.5	39.0
	CC*	48.1	317	190	424	910	1010	7.60	9.07	3.33	783	2.11	17.2	85.2
	RK	**13.6**	**79.1**	**47.9**	**102**	**292**	**337**	**4.93**	**6.33**	**2.69**	**176**	**1.52**	**6.25**	**19.0**
	PHAST	26.7	138	84.4	169	430	482	7.73	9.23	4.09	278	2.37	11.8	33.5

Table 4: Comparison of build time (in seconds) and query time (in microseconds) for all tested implementations across all graphs. "Ours" = our algorithm. "OSRM" = Open Source Routing Machine [49]. "CC" = CH-Constructor [23]. "RK" = RoutingKit [29]. "PHAST" = PHAST [26]. Implementations with asterisks (*) are parallel, and their sequential running times (with suffices "-seq") are also included in the table. We report the parallel running time of CC using eight threads, as this configuration yields the best performance due to its limited scalability.

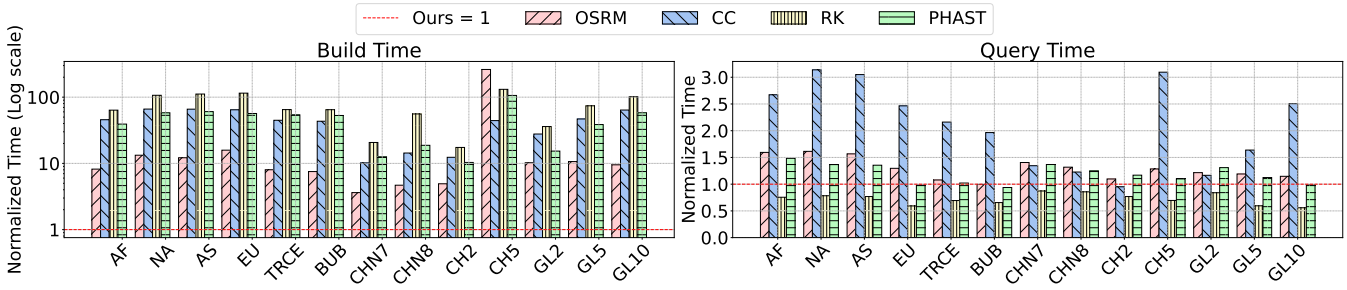

Figure 3: Build time and query time of all tested baselines, normalized to our algorithm. Lower is better. Since all running times are normalized to our algorithm, "Ours" is always equal to one, represented by the horizontal red dotted line.

queries are generally consistent with the query time. Therefore, we focus on analyzing the construction and query time here.

Fig. 3 and Tab. 4 show that our approach achieves significantly better construction performance than all baselines, while maintaining high CH graph quality. Even our sequential construction time is competitive to the highly-optimized sequential baselines, which is within 1.7× slower than RK and 2.9× than PHAST, and can also be faster than them by up to 2.1×. This indicates that our algorithm incurs very small overhead in work to enable parallelism. When running in parallel, the construction time of our algorithm is significantly faster than all baselines. Compared to the best parallel baseline OSRM, SPoCH is 3.6–260× faster in construction, with an average of 10.8× faster. CC suffers from scalability issue in the Reset step in each round (we will discuss more in Sec. 5.3 and Sec. 5.4). As a result, it achieves the best running time with 8 threads, which is close to the performance on sequential algorithms. In general, SPoCH is 10.2–66.2× faster then CC, with an average of 35.9× faster. As shown in Fig. 5, even only considering the time for scoring and contracting, CC is still not competitive to OSRM and is much slower than SPoCH. In general, SPoCH is at least 3.6× faster than all baselines on all graphs. On average across all graphs, SPoCH is 10.8× faster than OSRM, 35.9× faster than CC, 36.4× faster than (sequential) PHAST, and 63.3× than (sequential) RK.

We then analyze the query performance. As mentioned, the quality for a CH generated by a parallel algorithm is generally expected to be lower than those generated sequentially. The results indicate that our algorithm still achieves competitive quality to the best sequential implementation. In general, RK achieves the best query performance in both time and number of vertices visited. SPoCH is competitive to RK, with query times in 1.15–1.79×, and are better than all other baselines in query time, even including the sequential algorithm PHAST. On average, our query time is 1.39× slower than RK, but is 1.18× faster than PHAST, 1.27× faster than OSRM, and 1.95× faster than CC. This indicates the CH generated by our parallel algorithm has close quality to a highly-optimized sequential algorithm, and is better than other parallel versions.

5.3 Scalability

Self-Relative Speedup. We first show the self-relative speedup of our algorithms on all the graphs in Fig. 4. We vary the number of processors from 1 to 96h (192 hyperthreads). Our self-relative speedup is 53.9–61.0× on road graphs, 36.4–67.5× on synthetic graphs and 29.3–68.3× on k-NN graphs. This shows that our algorithm achieves high parallelism on all tested graphs.

Scalability Breakdown and Comparison with CC and OSRM. To validate our claims of improving parallelism in CH preprocessing and to illustrate the limitations of existing state-of-the-art parallel implementations, we test the running times of three parallel algorithms—SPoCH, CC, and OSRM—with cores varying from 1 to 96h (192 hyperthreads) in Fig. 5. We select three graphs: AF, CHN7, and GL5, each from a different category. As mentioned, all parallel implementations roughly follow Vetter's algorithm, so we split the total time into three parts: Score (running WPSes and recomputing vertex scores), Contracting (finding an independent set, contracting them and adding shortcuts), and Others.

CC does not achieve satisfactory scalability, and achieves its best performance with 8 threads. The major issue lies in a Reset step

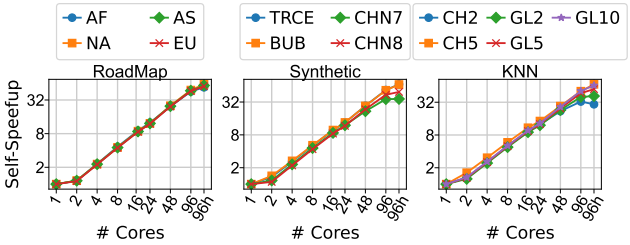

Figure 4: Self-relative speedups of our algorithm on the three graph categories. Higher is better.

that clears arrays at the beginning of each round. This cost increase drastically with the number of threads. To clarify the breakdown between Score and Contract, we also draw a figure for CC without the Others part, referred to as CC′. The sequential running time for the three algorithms are close. However, the running time for both CC and OSRM flattens down after 16 threads. Especially for CC, even without considering the resetting time in the Others part, its time in Contract remains the same or even increases when more than 16 threads are used. This is mainly due to the use of a lock-based structure when adding shortcuts, since using more threads may result in higher contention and further degrade the performance. Our solution using parallel hash table effectively avoids this issue, and remains scalable until using all 192 (hyper)threads.

OSRM has reasonable scalability up to 16 threads, and its running time remains the same when more threads are used. One possible reason lies in the parallel granularity for computing WPS—OSRM employs parallelism on the level of all vertices in V_A (vertices that require score recomputing). In particular, all vertices in V_A are parallelized, but each vertex $v \in V_A$ itself is processed sequentially, which includes a total of $N_{\text{in}}(v)$ executions of Dijkstra's algorithm. In later rounds when most vertices have been contracted, there are only a few vertices in V_A, but their degree may have become large, causing a huge amount of work executed sequentially. Therefore, its poor performance is due to insufficient parallelism in later rounds. This issue is also reflected in Fig. 6, which we discuss in Sec. 5.4. In this case, the WPSes are close to a sequential execution, and do not benefit from having more threads. Our solution that batches all WPS sources and parallelizes them as a whole effectively avoids this issue, and can efficiently utilize more threads.

On the three tested graphs, SPoCH shows high scalability up to 96h, and always achieves better performance with more threads. Finally, the self-relative speedup reaches 53.9× on AF, 36.4× on CHN7, and 55.7× on GL5. All techniques in our algorithm are carefully optimized for high parallelism, such as using lock-free data structures and batching WPSes to run in parallel. Therefore, with a reasonable sequential cost, the good scalability guarantees low parallel running time.

5.4 In-Depth Study for Parallel Algorithms

Recall that parallel CH construction algorithms select an independent set in each round, contract them all in parallel, and repeat until the graph becomes empty. Therefore, the performance and statistics of each round may help to better understand the process of parallel algorithms. In this section, we present an in-depth study for the performance breakdown in each round, for all parallel implementations SPoCH, CC and OSRM. We present the results on four graphs in Fig. 6, which includes the number of vertices processed and accumulated running time up to each round.

Number of Vertices Processed. In Fig. 6, we present the accu-

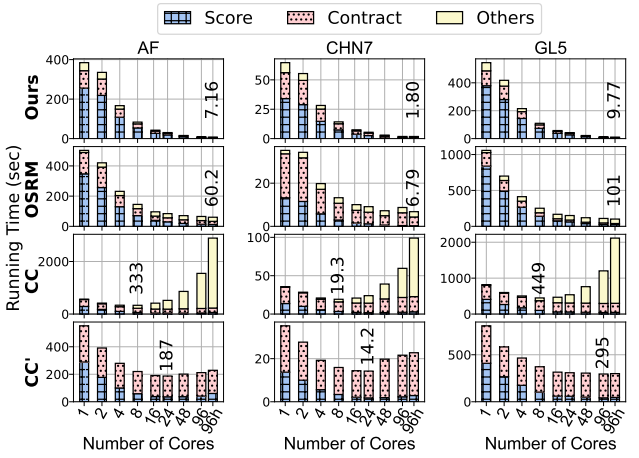

Figure 5: Running time of SPoCH (Ours), OSRM and CH-Constructor (CC) across different core counts. Lower is better. The fastest running time across all core counts is displayed above its bar. "96h" denotes 96 cores with hyper-threading enabled. As the "Others" cost in CC dominates when using more than eight threads, we also report a version of CC that excludes this time, referred to as CC'.

mulated numbers of vertices left in the overlay graph ($|V_O|$), the number of vertices contracted (V_F), as well as the number of WPSes (i.e., the number of executions of Dijkstra's algorithm). We first note that OSRM incurs much more rounds than SPoCH and CC—up to 4× more rounds than SPoCH. The larger number of rounds for OSRM is caused by how the independent set is selected. OSRM selects a vertex when it has the minimum score in its 2-hop neighborhood, while SPoCH uses 1-hop neighborhood. The goal of using 2-hop neighborhood is to avoid using complicated concurrent data structure to deal with shortcut insertion, but it results in a much slower contraction process than SPoCH. On the contrary, SPoCH uses the efficient data structure introduced in Sec. 4, which allows to contract all vertices with minimum score in their 1-hop neighborhood, leading to fewer rounds.

For all three algorithms, the size of the overlay graph decrease quickly. On CHN8, the trend of $|V_O|$ decreases linearly in the log scale. Indeed on a chain, since almost all vertices have the same edge difference, the contraction is mostly determined by the random priority. In this case, a constant fraction of vertices can be selected in expectation [61]. On other graphs, the size of the overlay graph decreases rapidly in the first few rounds, and then slows down. In general, for all implementations, the overlay graph size decreases by more than 99% in the first 10–20 rounds. This implies great potential of parallelism in the construction of CH.

We then compare the number of WPSes executed for each algorithm. As mentioned, one of our efforts is to reduce the number of WPSes performed to optimize the overall performance. SPoCH incurs significantly fewer Witness Path Searches (WPSes) compared to CC and OSRM. In each round, especially in later rounds, SPoCH requires much fewer WPS than the total number of vertices in the overlay graph. This reduction in the number of WPS demonstrates the efficiency of our approach in limiting unnecessary searches.

Running Time Breakdown Compared with CC. We now analyze the accumulated running time across rounds. In the first 10% of the rounds, the running time for both algorithms grows quickly. As mentioned, more than 99% of the vertices may be contracted just in the first 10–20 rounds. As the algorithm proceeds, the overlay graph becomes much smaller. On the four graphs, all algorithms

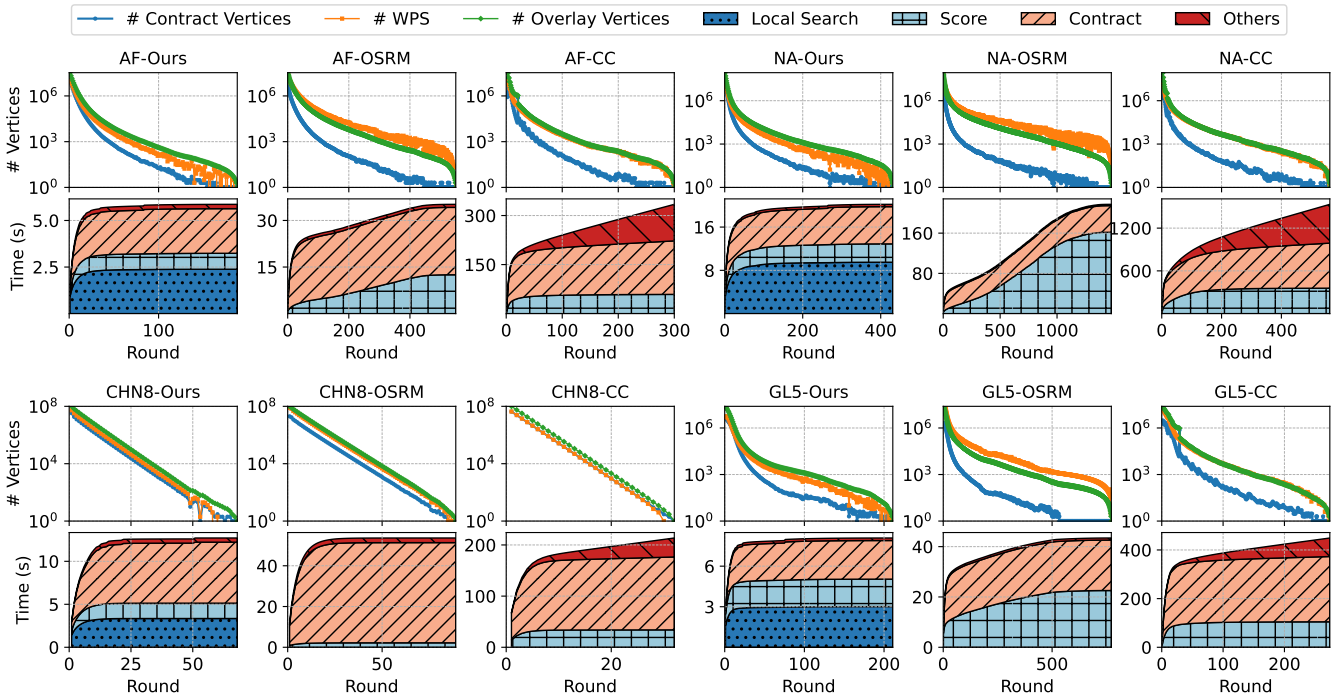

Figure 6: Per-round breakdown of SPoCH (Ours), OSRM and CH-Constructor (CC) on four representative graphs. The x-axis represents the round number. The top row shows the number of contracted vertices (blue circles), the number of WPS searches (orange squares), and the number of remaining overlay vertices (green diamonds) in each round. The bottom row shows the accumulated time for local search (dark blue), scoring (light blue), contracting (coral), and others (red).

reach a small overlay graph with 10^5 within 30 rounds, after which very little computation is required. Indeed, our algorithm spends the majority of time in the first several rounds. As the size of the overlay graph decreases, the running time almost does not increase for the last 90% of rounds. Indeed, all the steps become cheap after a few rounds. This indicates the effectiveness of our solution to maintain the overlay graph—when the overlay graph becomes small, the maintenance cost also shrinks proportionally. This benefit comes from our technique of maintaining the overlay graph in a lazy manner. In later rounds where only a few shortcuts are generated in each round, SPoCH delays the process of combining them into the CSR until it collects a sufficient number of shortcuts. Thus, after 10–20 rounds, there are likely only one or two global merges incurred, leading to high overall performance of SPoCH.

In contrast, the running time for OSRM grows steadily across all rounds. This indicates that even processing a small overlay graph, OSRM may spend a noticeable amount of time. As we discussed in Sec. 5.3, this is likely due to insufficient parallelism in later rounds. For CC, as mentioned, the use of inefficient data structures to maintain shortcuts dramatically increases its running time in the Contract step, leading to overall unsatisfactory performance.

In summary, our design allows for contract a large fraction of vertices in each round, leading to a very fast contraction process. Our new design for the LocalSearch step also reduces the number for WPSes performed. Our data structure also avoids a high cost to maintain a small overlay graph. As a result, SPoCH outperforms both parallel baselines due to new designs in both algorithm

Other Experiments. As mentioned, one of our optimizations is to combine the shortcuts with the overlay graph in a lazy manner, such that the cost of updating the CSR can be amortized. Our experiments show that on average, this reduces the cost of combining the two sets from 40.2% to 10.4% of the overall running time. Due to page limit, we present the figure in the supplemental material.

6 Related Work

Computing the shortest paths on a graph is one of the most well-studied problems in computer science. We refer the audience to the excellent surveys on this topic, including those by Bast et al. [14], Madkoure et al. [50], and Sommer [62]. These surveys all discuss contraction hierarchies (CH) in detail. Meanwhile, we note that the main applications of the CH are on sparse networks such as roadmaps (transition networks). We acknowledge that many interesting algorithms have been proposed for relevant but different problems such as on social networks, queries among all pairs or between a batch of sources and destinations, on dynamic graphs, and time-dependant queries. A brief list of recent work on computing shortest paths include [40, 43, 47, 48, 55, 66–69].

The idea of CH was proposed by Geisberger et al. [37], based on simplifying highway hierarchies [46, 57] and highway node routing [59]. CH has achieved notable success in practical applications, and fostered numerous later studies on relevant problems. Examples include the time-dependent versions [17, 18], parallel and distributed versions [45, 64], on dynamic graphs [53], and more algorithmic optimizations [27, 38, 42]. CH is also used to parallelize SSSP queries, such as in PHAST [3, 26]. Another stream of research is to derive theoretical guarantees for CH [8, 21, 25, 29, 36, 41]. These analyses are mostly parameterized, based on some graph invariants such as tree depths, tree widths, and diameters.

There have been many studies on parallelizing CH construction and relevant techniques. Vetter's work [64] is the earliest and inspired many of the later studies [2, 23, 24, 44, 45, 49]. We reviewed Vetter's approach in Sec. 2.2. OSRM by Luxen and Vetter [2, 49] implements Vetter's approach, and it is considered the SOTA paral-

lel CH construction. CH-Constructor [23] is another open-source software for parallel CH construction. We compared to OSRM and CH-Constructor since they have open-source code available. The others focus on different settings, such as distributed [45], GPU [44], and on edge contraction [24].

We note that there are other shortest-path algorithms that can provide different construction-query trade-offs or are designed for other graph types. Some of these algorithms include transit node routing [16], hub labeling [6, 7], pruned landmark labeling [10], highway labeling [9], and ALT [39]. Many studies have also tried to combine CH and these techniques to trade off among construction time, query speed, and space usage [19, 52, 54]. We believe that the faster CH construction presented in this paper can potentially help further explore such trade-offs, and we leave it as future work.

7 Conclusions and Future Work

In this paper, we propose SPoCH (Scalable Parallelization of Contraction Hierarchies), a parallel algorithm for constructing contraction hierarchies. Our key insights include algorithm redesign to introduce the LocalSearch step, which allows for batching, memoization and pruning, as well as leveraging parallel data structures. In this way, SPoCH effectively reduces the total work and enhances parallelism. Across 13 graphs of varying sizes and average degrees (including road networks, synthetic graphs, and k-NN graphs), SPoCH consistently outperforms four other state-of-the-art sequential and parallel baselines by 3.62–260×, while maintaining a competitive query performance. On a 96-core machine, SPoCH delivers self-relative speedups of 29.3–68.3×. We conduct in-depth experiments to analyze the improvements of our techniques.

References

[1] 2010. OpenStreetMap © OpenStreetMap contributors. https://www.openstreetmap.org/.

[2] 2024. OSRM. https://github.com/Project-OSRM/osrm-backend.

[3] 2024. PHAST. https://github.com/kit-algo/ULTRA-PHAST.

[4] 2024. RoutingKit. https://github.com/RoutingKit/RoutingKit.

[5] Tenindra Abeywickrama, Muhammad Aamir Cheema, and David Taniar. 2016. k-Nearest Neighbors on Road Networks: A Journey in Experimentation and In-Memory Implementation. *Proceedings of the VLDB Endowment (PVLDB)* 9, 6 (2016).

[6] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2011. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms (SEA)*. Springer, 230–241.

[7] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2012. Hierarchical hub labelings for shortest paths. In *European Symposium on Algorithms (ESA)*. Springer, 24–35.

[8] Ittai Abraham, Amos Fiat, Andrew V Goldberg, and Renato F Werneck. 2010. Highway dimension, shortest paths, and provably efficient algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 782–793.

[9] Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata. 2014. Fast shortest-path distance queries on road networks by pruned highway labeling. In *Algorithm Engineering and Experiments (ALENEX)*. SIAM, 147–154.

[10] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 349–360.

[11] Anonymous. 2024. Implementations for Parallel Contraction Hierarchies. https://anonymous.4open.science/r/Parallel-Contraction-Hierarchy-AEE1/.

[12] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. 2001. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)* 34, 2 (2001), 115–144.

[13] Julian Arz, Dennis Luxen, and Peter Sanders. 2013. Transit node routing reconsidered. In *International symposium on experimental algorithms*. Springer, 55–66.

[14] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Mü ller Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. 2016. Route planning in transportation networks. *Algorithm engineering: Selected results and surveys* (2016), 19–80.

[15] Hannah Bast, Stefan Funke, and Domagoj Matijevic. 2006. Ultrafast Shortest-Path Queries via Transit Nodes. *The Shortest Path Problem* 74 (2006), 175–192.

[16] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. 2007. Fast routing in road networks with transit nodes. *Science* 316, 5824 (2007), 566–566.

[17] Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders. 2010. Time-dependent contraction hierarchies and approximation. In *International Symposium on Experimental Algorithms (SEA)*. Springer, 166–177.

[18] G Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. 2013. Minimum time-dependent travel times with contraction hierarchies. *Journal of Experimental Algorithmics (JEA)* 18 (2013), 1–1.

[19] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. 2010. Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. *Journal of Experimental Algorithmics (JEA)* 15 (2010), 2–1.

[20] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 89–102.

[21] Johannes Blum, Stefan Funke, and Sabine Storandt. 2021. Sublinear search spaces for shortest path planning in grid and road networks. *Journal of Combinatorial Optimization* 42, 2 (2021), 231–257.

[22] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (1999), 720–748.

[23] Stefan Bühler and André Nusser. 2024. Contraction Hierarchies Constructor. https://doi.org/10.5281/zenodo.14008202

[24] Zi Chen, Xinyu Ji, Long Yuan, Xuemin Lin, Wenjie Zhang, and Shan Huang. 2024. Parallel Contraction Hierarchies Construction on Road Networks. *IEEE Transactions on Knowledge and Data Engineering* (2024).

[25] Tobias Columbus and Reinhard Bauer. 2009. On the complexity of contraction hierarchies. *Student Thesis, Karlsruhe Inst. Technol., Karlsruhe, Germany* (2009).

[26] Daniel Delling, Andrew V Goldberg, Andreas Nowatzyk, and Renato F Werneck. 2013. PHAST: Hardware-accelerated shortest path trees. *J. Parallel and Distrib. Comput.* 73, 7 (2013), 940–952.

[27] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. 2017. Customizable route planning in road networks. *Transportation Science* 51, 2 (2017), 566–591.

[28] Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2013. Hub label compression. In *International symposium on experimental algorithms*. Springer, 18–29.

[29] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. 2016. Customizable contraction hierarchies. *J. Experimental Algorithmics* 21 (2016), 1–49.

[30] Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959).

[31] Xiaojun Dong, Yan Gu, Yihan Sun, and Letong Wang. 2024. Brief Announcement: PASGAL: Parallel And Scalable Graph Algorithm Library. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[32] Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. 2021. Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 184–197.

[33] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. 2015. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical* 215 (2015), 618–629.

[34] Stefan Funke, André Nusser, and Sabine Storandt. 2015. Placement of loading stations for electric vehicles: No detours necessary! *Journal of Artificial Intelligence Research* 53 (2015), 633–658.

[35] Stefan Funke, Tobias Rupp, André Nusser, and Sabine Storandt. 2019. PATHFINDER: storage and indexing of massive trajectory sets. In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases*. 90–99.

[36] Stefan Funke and Sabine Storandt. 2015. Provable efficiency of contraction hierarchies with randomized preprocessing. In *International Symposium on Algorithms and Computation*. Springer, 479–490.

[37] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental Algorithms (WEA)*. Springer, 319–333.

[38] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. 2012. Exact routing in large road networks using contraction hierarchies. *Transportation Science* 46, 3 (2012), 388–404.

[39] Andrew V Goldberg and Chris Harrelson. 2005. Computing the shortest path: A search meets graph theory. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Vol. 5. 156–165.

[40] Zengyang Gong, Yuxiang Zeng, and Lei Chen. 2024. Querying shortest path on large time-dependent road networks with shortcuts. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 4532–4544.

[41] Michael Hamann and Ben Strasser. 2018. Graph bisection with pareto optimization. *J. Experimental Algorithmics* 23 (2018), 1–34.

[42] Demian Hespe and Peter Sanders. 2019. More Hierarchy in Route Planning Using Edge Hierarchies. In *Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*.

[43] Shuai Huang, Yong Wang, Tianyu Zhao, and Guoliang Li. 2021. A learning-based method for computing shortest path distances on road networks. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 360–371.

[44] Roozbeh Karimi, David M Koppelman, and Chris J Michael. 2020. Fast GPU Graph Contraction by Combining Efficient Shallow Searches and Post-Culling. In *IEEE Conference on High Performance Extreme Computing (HPEC)*. IEEE, 1–7.

[45] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. 2010. Distributed time-dependent contraction hierarchies. In *International Symposium on Experimental Algorithms (SEA)*. Springer, 83–93.

[46] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. 2007. Computing many-to-many shortest paths using highway hierar-

chies. In *Algorithm Engineering and Experiments (ALENEX)*. SIAM, 36–45.

[47] Lingxiao Li, Muhammad Aamir Cheema, Mohammed Eunus Ali, Hua Lu, and David Taniar. 2020. Continuously monitoring alternative shortest paths on road networks. In *Proceedings of the VLDB Endowment (PVLDB)*. Association for Computing Machinery, 2243–2255.

[48] Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. 2020. Fast query decomposition for batch shortest path processing in road networks. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 1189–1200.

[49] Dennis Luxen and Christian Vetter. 2011. Real-time routing with OpenStreetMap data. In *Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems*. 513–516.

[50] Amgad Madkour, Walid G Aref, Faizan Ur Rehman, Mohamed Abdur Rahman, and Saleh Basalamah. 2017. A survey of shortest-path algorithms. *arXiv preprint:1705.02044* (2017).

[51] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152.

[52] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 709–724.

[53] Dian Ouyang, Long Yuan, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees. *Proceedings of the VLDB Endowment (PVLDB)* 13, 5 (2020), 602–615.

[54] Claudius Proissl. 2023. On Improving Contraction Hierarchies Using Distance Group Partitions. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*. SIAM, 38–48.

[55] Yu-Xuan Qiu, Dong Wen, Lu Qin, Wentao Li, Rong-Hua Li, Zhang Ying, et al. 2022. Efficient shortest path counting on large road networks. *Proceedings of the VLDB Endowment (PVLDB)* (2022).

[56] Naroa Coretti Sanchez, Iñigo Martinez, Luis Alonso Pastor, and Kent Larson. 2022. On the simulation of shared autonomous micro-mobility. *Communications in Transportation Research* 2 (2022), 100065.

[57] Peter Sanders and Dominik Schultes. 2006. Engineering highway hierarchies. In *European Symposium on Algorithms*. Springer, 804–816.

[58] Peter Sanders, Christian Schulz, and Dorothea Wagner. 2014. Benchmarking for graph clustering and partitioning. *Encyclopedia of social network analysis and mining Springer* (2014).

[59] Dominik Schultes and Peter Sanders. 2007. Dynamic highway-node routing. In *Workshop on Experimental and Efficient Algorithms (WEA)*. Springer, 66–79.

[60] 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.

[61] Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B Gibbons. 2015. Sequential random permutation, list contraction and tree contraction are highly parallel. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 431–448.

[62] Christian Sommer. 2014. Shortest-path queries in static networks. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 1–31.

[63] William F Tinney and John W Walker. 1967. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE* 55, 11 (1967), 1801–1809.

[64] Christian Vetter. 2009. Parallel time-dependent contraction hierarchies. *Student Research Project* (2009), 134.

[65] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2021. GeoGraph: A Framework for Graph Processing on Geometric Data. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 38–46.

[66] Ziqiang Yu, Xiaohui Yu, Nick Koudas, Yang Liu, Yifan Li, Yueting Chen, and Dingyu Yang. 2020. Distributed processing of k shortest path queries over dynamic road networks. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 665–679.

[67] Junhua Zhang, Wentao Li, Long Yuan, Lu Qin, Ying Zhang, and Lijun Chang. 2022. Shortest-path queries on complex networks: experiments, analyses, and improvement. *Proceedings of the VLDB Endowment (PVLDB)* 15, 11 (2022), 2640–2652.

[68] U Zhang, Long Yuan, Wentao Li, Lu Qin, and Ying Zhang. 2021. Efficient label-constrained shortest path queries on road networks: A tree decomposition approach. *Proceedings of the VLDB Endowment (PVLDB)* (2021).

[69] Yikai Zhang and Jeffrey Xu Yu. 2020. Hub labeling for shortest path counting. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1813–1828.

[70] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 2008. Learning transportation mode from raw gps data for geographic applications on the web. In *International World Wide Web Conference (WWW)*. 247–256.