A CPU-GPU Hybrid Labelling Algorithm for Massive Shortest Distance Queries on Road Networks

Jiajia Li
Yongzhi Chen
Shenyang Aerospace University
ShenYang, China
lijiajia@sau.edu.cn
chen1490727038@gmail.com

Mengxuan Zhang
Australian National University
Canberra, Australia
mengxuan.zhang@anu.edu.au

Lei Li
DSA Thrust, HKUST (GZ), China
CSE, HKUST, Hong Kong SAR
thorli@ust.hk

ABSTRACT

Shortest distance computation is a fundamental operation in graph-related applications, especially in location-based services. The most efficient method is hop-labeling, which can answer queries in microseconds. However, when the traffic condition changes dynamically, they need a long time to maintain or an even longer time to re-construct, making it hard to catch up with numerous or frequent updates. As a result, real-life applications still rely on slow graph searching algorithms. To improve the hop labeling construction efficiency, we resort to GPU for its high parallelism power and propose the G2H index. Specifically, we first analyze the relation of the graph partitions, index performance, and parallelism to identify the most suitable partition scheme for G2H, with a hybrid scheme and optimized node ordering for faster contraction. Then, we propose a label-pruning method to reduce the label construction workload with several strategies designed to balance and improve the parallel label construction. Finally, experiments on real-life networks show that our G2H can finish construction within seconds for large urban networks and under one minute for large region networks with 6M vertices, which is several times faster than the state-of-the-art methods. Besides, G2H can answer hundreds of millions of queries per second, achieving two orders of magnitude acceleration.

KEYWORDS

Shortest Distance, Tree Decomposition, GPU, Road Network

PVLDB Reference Format:
Jiajia Li, Yongzhi Chen, Mengxuan Zhang, and Lei Li. A CPU-GPU Hybrid Labelling Algorithm for Massive Shortest Distance Queries on Road Networks. PVLDB, 18(3): 770 - 783, 2024.
doi:10.14778/3712221.3712241

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sauccjy/GPU-H2H.

1 INTRODUCTION

Shortest distance query is a fundamental operation in location-based services and graph-related computations. In real-life applications, travel time is often more important than travel distance because it reflects the actual travel cost. While travel time along road segments (*i.e.,* edge weight of graph) varies with traffic conditions, making road networks dynamic. However, between changes, the network can be viewed as a static one at each timestamp, so we continue to use the conventional term "shortest distance" here.

The dynamic network can be modeled as a set of network snapshots, where edge weights and network topology could change between the snapshots. The number of changes between two snapshots is called update number, and the total update number per unit time is called update frequency. Then, depending on the update number and frequency, the dynamicity can be categorized into the following scenarios: 1) *When there is no update between snapshots*, the network is essentially static. Then hop-labeling distance index [2, 4, 5, 10, 17, 32, 44] is the most efficient method that can answer distance queries in microseconds and handle millions of queries per second with only one server. However, it usually takes a long time for index construction; 2) *When there are a few updates, or the frequency is low*, the index maintenance methods [18, 46, 58, 61–64, 68, 69] could take some time for index update to support correct query answering; 3) *When there are a large number of updates or the update frequency is high* such that the index maintenance or re-construction cannot finish before next update, then online search [16, 22] is the only available option [63]. Though some batch processing methods are proposed [23, 31, 52, 59, 60] to further improve the query efficiency, they are still thousands of times slower than the index-based one, so they require thousands of servers to handle millions of queries in real-time.

Motivation. Consequently, to the best of our knowledge, no method can process distance queries at the million level per second in a highly dynamic environment. Hop-labeling is the only method that can address large query demand, but its maintenance cannot handle frequent updates [63]. Therefore, in this work, we resort to accelerating the index construction so that it can handle highly dynamic networks even when the entire graph changes. Specifically, the current state-of-the-art distance index for road networks is H2H [44], which is also the fundamental structure for many advanced distance indexes [13, 28, 29, 36–38, 45, 65]. But it still takes several minutes to construct on networks with millions of vertices, which can hardly catch up with the highly frequent update. As its construction efficiency cannot improve theoretically, efforts are mainly put on providing better vertex orders or tree structure [10, 17, 25, 66],

Proceedings of the VLDB Endowment, Vol. 18, No. 3 ISSN 2150-8097.
doi:10.14778/3712221.3712241

[*] Lei Li is the corresponding author.

but with marginal improvement. So, how to accelerate the H2H construction significantly is still an important open problem.

On the other hand, GPU has become an essential component in modern computation due to its SIMD (Single Instruction Multiple Data) paradigm with huge number of cores and large memory bandwidth. Besides, it has shown success in large graph computation tasks [14, 19, 20, 35, 40, 48, 51, 55]. However, the efficient solution for H2H construction and query answering on GPU has been overlooked. Therefore, we aim to propose a GPU-based H2H index (named G2H) that is fast in index construction and query processing such that a single machine can handle a huge number of queries in highly dynamic road networks.

Challenges. However, it is non-trivial to make the best use of GPU's SIMD paradigm. Firstly, constructing H2H in parallel [33, 36, 64] is based on the existing graph partitioning results [15] with the partitions running in parallel. There are at most a few hundred partitions for one road network, as existing graph partitioning approaches mainly aim at minimizing cut or balancing partitions. Although it works fine on the CPU since its thread number is usually not larger than a few hundred, it cannot fully exploit the GPU's parallel potential with tens of thousands of parallel computing units. Besides, the first vertex contraction phase in index construction has very large data coupling, so it naturally favors larger partition running in serial, which further makes it harder for GPU. Therefore, we first analyze the relations between the graph partition, index performance, and parallelism to identify the most suitable partition scheme for G2H support hundreds of thousands of contraction threads that run conflict-free in parallel. Next, after observing and analyzing the trends of GPU and CPU computation behavior when contracting on the higher-level partitions, we propose a GPU-CPU hybrid contraction method to take advantage of both of them.

Secondly, because vertex order is a crucial factor as it determines the index structure, index size and construction time [17, 25, 34, 66], we analyze the influence of hierarchical partitioning on the vertex order theoretically and then propose a Decomposition Tree Height (DTH) ordering to improve index structure with little influence on contraction efficiency.

Thirdly, the GPU's SIMD nature requires the data space to be allocated and transferred to the GPU's memory beforehand with few branches of operations. However, in the first vertex contraction phase, the space cannot be settled as it changes dynamically. To solve this problem, we propose an upper-bound estimation. During the second label construction phase, there would be an exponential number of data transfers. To avoid the explosion, we propose a shared data allocation. In terms of label construction thread coordinating, we propose two frontier selection strategies to avoid thread divergence and a TD-based parallel granularity to exploit the large thread number further for faster construction.

Finally, label construction is time-consuming since it involves a significant amount of computation. To alleviate this workload, we conduct a theoretical analysis of hop labeling and identify redundancy in the current indexes. Building on this, we propose a label-pruning method that effectively reduces the computation workload by half without compromising the query quality.

Contributions. Our contributions are summarized below:

- We propose G2H, the first GPU-based hop labeling index that can construct an index in seconds and answer queries in *ns*;
- We propose a conflict-free parallel contraction method that has a parallelism degree of hundreds of thousands with hierarchical partitioning, a *DTH* order to refine the index structure, and a *CPU-GPU hybrid contraction scheme* for fast contraction;
- We propose several data allocation strategies and parallel thread coordination strategies to fully take advantage of GPU's SIMD;
- We propose a label pruning method to reduce label computation time by half.
- Our experimental studies on real-life road networks validate the superiority of G2H compared with the state-of-the-arts.

2 PRELIMINARIES

In this section, we first introduce the basic concepts formally. Then, we introduce the H2H index [44] that we aim to parallelize, followed by GPU architecture that our proposed algorithm relies on.

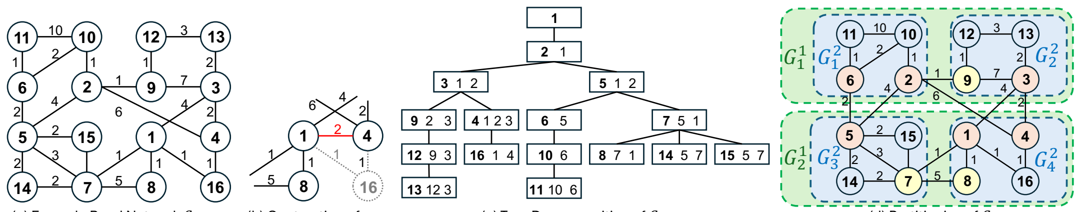
2.1 Basic Concepts

Let $G = (V, E)$ be a weighted road network with vertex set V and edge set $E \subseteq V \times V$. Each vertex $v \in V$ denotes road intersection. Each edge $e(u, v) \in E$ denotes road segment and is associated with a positive weight, also denoted as $e(u, v)$ if unambiguous in the context. For each vertex $v \in V$, its neighbors are denoted as $N(v) = \{u|e(u, v) \in E\}$ with degree $deg(v) = |N(v)|$. A *path* $p(v_1, v_k) = \langle v_1, v_2, v_3, ..., v_k\rangle$ from v_1 to v_k is a sequence of consecutive vertices where $e(v_i, v_{i+1}) \in E, \forall i \in [1, k)$, and its length is $\sum e(v_i, v_{i+1})$. If it has the minimum length among all paths from v_1 to v_k, then we say path $p(v_1, v_k)$ is the shortest path with the shortest distance denoted as $dist(v_1, v_k)$. It should be noted that we use an undirected graph in this work for easier presentation, while it is straightforward to extend to the directed graph. A distance query $q(s, t)$ asks for the shortest distance from s to t. For example in Figure 1-(a), $q(v_2, v_7) = dist(v_2, v_7) = 7$, and $q(v_4, v_6) = dist(v_4, v_6) = 8$.

2.2 H2H Shortest Distance Index

H2H [44] is a kind of 2-hop labeling shortest distance index [12] based on tree decomposition [49], which is suitable for road networks due to their small treewidth [63]. In general, each $v \in V$ is assigned a label set $L(v) = \{(u, dist(u, v))\}$, where the projection of $L(v)$ on the keys is called *hub nodes* $C(v) = \{u|(u, dist(u, v)) \in L(v)\}$. Given a query $q(s, t)$, its shortest distance can be calculated as $dist(s, t) = \min_{x \in C(s) \cap C(t)} dist(s, x) + dist(x, t)$ in $O(|C(s) \cap C(t)|)$ time, where $|C(s) \cap C(t)|$ is normally around hundreds in urban road networks. The labels are correct when the common hubs are the superset of the cuts between any two vertices.

The construction of H2H takes two phases: *Vertex Contraction* and *Label Assignment*. In the first phase, the vertices are contracted in a given order denoted as $r(v)$. When contracting v, we call its neighbors *decomposition neighbors*, denoted as $N_D(v)$, which forms a cut between v and the remaining part of G. v and $N_D(v)$ along with the edges between them form a tree node $T(v)$ represented by v. For all neighbor pairs (u, w) with $u, w \in N_D(v)$, if there is no edge between them, we add a new edge $e(u, w)$ with edge weight $e(u, w) = e(u, v) + e(v, w)$ into G; otherwise, we update its weight as $e(u, v) + e(v, w)$ if $e(u, w) > e(u, v) + e(v, w)$. After that, v along with



(a) Example Road Network G_{Total} (b) Contraction of v_{16} (c) Tree Decomposition of G_{Total} (d) Partitioning of G_{Total}

Figure 1: Road Network G_{total}, its Tree Decomposition, and Hierarchical Partitioning

Table 1: Decomposition Tree $N_D(v_i)$ and H2H Label $L_{3-total}$, $L_{4-total}$. Tradition H2H Label is the same as $L_{4-total}$

v_i	$N_D(v_i)$	$L_{3-total}$	$L_{4-total}$	
		DIS	POS	DIS
v_1	$\emptyset$	0	0	0
v_2	$(v_1, 8)$	8, 0	0, 1	8, 0
v_3	$(v_1, 4), (v_2, 7)$	4, 7, 0	0, 1, 2	4, 7, 0
v_4	$(v_1, 2), (v_2, 6), (v_3, 2)$	2, 6, 2, 0	0, 1, 2, 3	2, 6, 2, 0
v_5	$(v_1, 4), (v_2, 4)$	4, 4, 0	0, 1, 2	4, 4, 0
v_6	$(v_2, 3), (v_5, 2)$	6, 3, 2, 0	2, 3	6, 3, 2, 0
v_7	$(v_1, 1), (v_5, 3)$	1, 7, 3, 0	0, 2, 3	1, 7, 3, 0
v_8	$(v_1, 1), (v_7, 5)$	1, **12**, **8**, **5**, 0	0, 3, 4	1, **9**, **5**, **2**, 0
v_9	$(v_2, 1), (v_3, 6)$	9, 1, 6, 0	1, 2, 3	9, 1, 6, 0
v_{10}	$(v_2, 1), (v_6, 2)$	8, 1, 4, 2, 0	3, 4	8, 1, 4, 2, 0
v_{11}	$(v_6, 1), (v_{10}, 10)$	7, 4, 3, 1, **10**, 0	3, 4, 5	7, 4, 3, 1, **3**, 0
v_{12}	$(v_3, 5), (v_9, 1)$	9, 2, 5, 1, 0	2, 3, 4	9, 2, 5, 1, 0
v_{13}	$(v_3, 2), (v_{12}, 3)$	6, 5, 2, 4, 3, 0	2, 4, 5	6, 5, 2, 4, 3, 0
v_{14}	$(v_5, 2), (v_7, 2)$	3, 6, 2, 2, 0	2, 3, 4	3, 6, 2, 2, 0
v_{15}	$(v_5, 2), (v_7, 2)$	3, 6, 2, 2, 0	2, 3, 4	3, 6, 2, 2, 0
v_{16}	$(v_1, 1), (v_4, 1)$	1, 7, 3, 1, 0	0, 3, 4	1, 7, 3, 1, 0

its edges are removed from the graph. For example in Figure 1-(b), when contracting v_{16}, we add edge $e(v_1, v_4)$ with weight $1 + 1 = 2$ and then delete edges $e(v_1, v_{16})$ and $e(v_4, v_{16})$. When all vertices are contracted, we have a set of tree nodes $\{T(v)\}$. To construct a tree decomposition of G, we set $T(w)$ (with w the last contracted vertex) as the tree root. For each remaining vertex v, we connect $T(v)$ as the child of $T(u)$ where u has the smallest order in $N_D(v)$. For example, in Figure 1-(c), $T(v_{16})$'s parent is $T(v_4)$ because v_4 has smaller order than v_1. The *decomposition neighbors* $N_D(v)$ for each vertex are listed in the second column.

In the second phase, we calculate the label set $L(v)$ for each vertex v in a top-down manner where $C(v) = \{u\}$ with $\{T(u)\}$ the ancestor set of $T(v)$. Specifically, $dist(v, a)$ $(a \in C(v))$ is calculated as $dist(v, a) = min\{e(v, u) + dist(u, a) | \forall u \in N_D(v)\}$. This procedure is correct because $T(u)$ is always an ancestor of $T(v)$ and its label has been constructed previously. The complete label set is shown in the last column of Table 1.

When answering $q(s, t)$, H2H first finds the Lowest Common Ancestor (LCA) [6] of s and t in the tree because the vertices in it are the cuts between s and t. Then the shortest distance is the smallest one among the addition values of these two hop labels. For example, to answer $q(v_8, v_14)$, we find their LCA is $T(v_7)$ with cuts $\{v_7, v_5, v_1\}$. Then $dist(v_8, v_{14}) = min\{dist(v_8, v_7) + dist(v_7, v_{14}),$ $dist(v_8, v_5) + dist(v_5, v_{14}), dist(v_8, v_1) + dist(v_1, v_{14})\} = 4$.

2.3 GPU Architecture

GPU is a data-oriented computing device that achieves high parallelism through *SIMD* (Single Instruction Multiple Data). Its basic physical computing unit is the Streaming Processor (SP), while a batch of SPs forms its smallest execution unit, *i.e.,* the Streaming MultiProcessor (*SM*). The basic logical computing unit is the thread, while a set of threads forms the basic logical execution unit *Warp*, which contains 32 threads in NVidia GPU. A *Block* is a logical grouping of threads (typically 32 to 1024) that execute the same code and share the same memory, while a set of blocks form a larger logical unit *Grid*. Such architecture with thousands of threads favors codes with enormous repeated computations rather than multi-branch codes, otherwise only partial computation can benefit from utilizing GPU. Finally, because GPU and CPU normally have separate memory, it takes extra time for data allocation and transmission between them.

Although GPU and CPU both use threads to support parallelism, they are different in the following ways: 1) CPU's thread number is smaller while GPU's is much larger; 2) It takes longer time for CPU to switch and construct threads but very little time for GPU; 3) Threads in CPU runs separately while threads in GPU has to do exactly the operation at the same time. Therefore, when one thread finishes in the CPU, we can switch to another thread to fill the vacancy. But in GPU, we have to wait for all threads to finish, and it could be long if there are many branches in the program or the thread load is imbalanced. This phenomenon is called *thread divergence*. Next, we formally define our problem as follows:

***Problem Statement*.** Given a road network $G(V, E)$, we aim to use GPU to reduce H2H index construction time and improve its query efficiency.

3 CONTRACTION OPTIMIZATION

In this section, we accelerate the first vertex contraction through parallelization. As introduced in Section 2.2, vertices are contracted in serial in the original H2H. However, if we follow any serial order to parallelize contraction, we may suffer from conflicts. For instance, if we contract v_{12} and v_{13} at the same time, they would conflict in adding edge $e(v_9, v_{13})$ and $e(v_3, v_{12})$, while contracting v_{11}, v_{13}, v_{14} and v_{16} in parallel has no conflict. Therefore, we first resort to partition to identify the conflict-free parallelization opportunities in Section 3.1. Then, we adapt the contraction process to GPU and propose a hybrid scheme in Section 3.2. Finally, we propose a fast tree height order to refine order in Section 3.3.

3.1 Conflict-Free Parallel Contraction

To enable parallel processing on graphs, partitioning [3, 15, 24, 26, 42] is widely used in large graph systems [39, 41, 54] and also pathfindings [11, 30, 36, 56, 57, 67]. However, most of these applications only utilize the partition results with at most two levels [64]: the lower level partitions could run in parallel and an optional upper level (*i.e.,* overlay graph) organizes the connections among partitions. However, such a structure cannot fully utilize the high parallelism of GPU because its power is restricted by the lower-level partition number, in which the inner-partition index construction still runs in serial. In other words, if we have more partitions, then we can have higher parallelism. Therefore, in this section, we discuss the potential of parallelism and further give the parallel contract rank by introducing the Partition Rank Tree.

3.1.1 *Contraction Conflict*.

We first analyze the obstacle of parallel contraction through the following theorem:

Theorem 1 (Contraction Conflict Condition). *The parallel contraction of v_i and v_j have a conflict if and only if they have the same decomposition neighbor v_k.*

Proof. When contracting a node $v_i \in V$, its operations involve v_i and $N_D(v_i)$. Then, only the following data would be changed: 1) The removal of v_i and its connected edges $\{e(v_i, v_k)|v_k \in N_D(v_i)\}$. This operation introduces no conflict because all this information is unique for v_i; 2) The insertion of a non-existing edge $\{e(v_k, v_x)|v_k, v_x \in N_D(v_i)\}$ would require new space in $N_D(v_k)$ (suppose v_k has higher order than v_x), so it could introduce conflict when $N_D(v_k)$ is also accessed by another thread; 3) The weight update of $e(v_k, v_x)$ could also introduce conflict when v_k and v_x also belong to $N_D(v_j)$ and $e(v_k, v_x)$ is accessed or updated when v_j is contracted. In summary, the conflict among vertex contractions could happen only with common decomposition neighbor(s). □

In Figure 1-(b), v_8 and v_{16} have a common neighbor v_1. When we contract v_{16}, a new edge $e(v_1, v_4)$ is added to $N_D(v_1)$, which could also be accessed by v_8 during its contraction (case 2). In terms of Case 3, v_{14} and v_{15} have common ancestors v_5 and v_7, so when they are contracted in parallel, updating $e(v_5, v_7)$ is conflicted. Therefore, avoiding conflict is crucial to parallel contraction, which can be achieved by identifying the non-conflict vertices through the following corollary:

Corollary 2 (Non-Conflict Vertex Pair). *For any v_i and v_j, if $N_D(v_i) \cap N_D(v_j) = \phi$, they can contract in parallel with no conflict and are called non-conflict vertex pairs.*

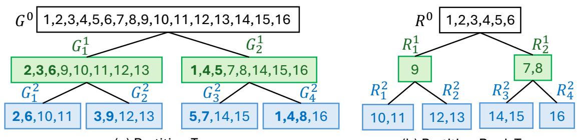
3.1.2 *Hierarchical Graph Partition*.

To identify the non-conflict vertices, we resort to edge-cut graph partitioning. Generally, the road network G is divided into multiple subgraphs $\{G_i|1 \le i \le k\}$ such that $\bigcup_{i\in[1,k]} V(G_i) = V, V(G_i) \cap V(G_j) = \phi(\forall i \ne j, i, j \in [1, k])$. $\forall v \in G_i$, we say v is a *boundary vertex* if there exists a neighbor of v in the another subgraph, that is $\exists u \in N(v), u \in G_j(i \ne j)$. The corresponding edge is called *boundary edge*. Otherwise, v is an *inner vertex*. In $G_i(1 \le i \le k)$, we denote the inner vertex set as I_i and the boundary vertex set as B_i. The opportunity of parallel vertex contraction can be revealed through the following lemma:

Lemma 3. *Given two partitions G_i and G_j, then any $v_i \in I_i$ and $v_j \in I_j$ have no conflict and can contract in parallel.*

Proof. $N(v_i) \cap N(v_j) \subseteq B_i \cap B_j = \phi$ □

Therefore, the contractions among partitions can run in parallel while those within each partition still need to run in serial, and the partition number determines the number of threads or degree of parallelism. Then, a good partition result should satisfy the following properties: 1) It should generate a large number of partitions to fully utilize the GPU's parallelism potential; 2) The vertex number



(a) Partition Tree (b) Partition Rank Tree

Figure 2: Partition Tree and Rank Tree

in different partitions should be similar such that the workload among threads could be balanced; 3) The partition size should be small such that each thread's serial contraction workload is small with fewer branches.

To fulfill the above properties, we propose to partition the graph recursively until each partition has at least 3 vertices (as it takes at least three vertices to contract) and cannot be partitioned anymore. The recursive partition could keep the partition size as small as possible (smaller workload with few branches) while increasing the partition number (larger thread number). Specifically, we use G_i^h to denote the i^{th} partition on the h^{th} layer and the original graph is G^0. In general, for each G_i^h partition, the hierarchical graph partitioning methods [24, 47] partition it into k partition $\{G_1^{h+1}, \cdots, G_k^{h+1}\}$, with k being a tunable parameter and normally at least 4. However, we find that $k = 2$, which is the smallest partition number, is the most suitable to our contraction requirements due to the following reasons: 1) A larger k decreases the overall thread number (each time reduces by k instead of 2), so the parallelism drops faster; 2) A larger k increases the partition sizes faster so the workload and thread size increases. Therefore, we utilize the hierarchical partitioning methods [24, 47] with fanout of 2 for contraction. A partition example is illustrated in Figure 1-(d). We first divide G into G_1^1 and G_2^1 (the green ones) and then continue to divide them into subgraphs G_1^2, G_2^2, G_3^2 and G_4^2.

3.1.3 *Hierarchical Parallel Node Ordering and Contraction*.

Now we discuss the relation between the contraction result on the hierarchical partitioned graph and the original one through node ordering. Firstly, the partitioned results can be represented by the following structure:

Definition 1 (Partition Tree). *Given a road network G and its hierarchical partitions G^h. For any partition G_i^k ($k < h$), it is connected as a parent node of the $k + 1$ layer subgraphs partitioned from it. Then a tree structure is formed.*

Figure 2-(a) shows a 3-layer partition tree of Figure 1-(d). We show each subgraph's boundary vertices in bold. The relation between the neighboring layers' boundary vertex is shown below:

Property 1 (Inter-Layer Boundary Inclusion). *For any partition G_i^k ($k < h$), its boundary vertex set B_i^k is a subset of its children's boundary vertex set.*

It is easy to prove that new boundaries would be introduced by partitioning while the old ones still serve as boundaries. The contractions among non-boundary vertices within one partition are conducted in serial, while those in the same-layer partitions are in parallel. Then, if we only keep the contracted vertex at each layer, we obtain a *Partition Rank Tree* as shown in Figure 2-(b), with the contracted vertices of G_i^j denoted by R_i^j. Such a rank tree provides us with a tool to analyze the index equivalence with the original H2H index. We first define a set of ordering below.

Definition 2 (**Hierarchical Parallel Node Ordering Set**). *Given a partition rank tree, we can form a set of serial node orders according to the following rules:*

(1) *$\forall 0 \le k < h$, R^k's order is higher than R^{k+1};*
(2) *Within each k-layer, for any pair of $v_i \in G_i^k$ and $v_j \in G_j^k$ from different partition, their relative order can be arbitrary;*
(3) *Within each partition, the vertex relative orders are fixed.*

Rule 1 requires R^0's order higher than R^1's, which is higher than R^2's. Rule 2 allows the R_i^2's orders arbitrary. For instance, orders among v_{10}, v_{12}, v_{14} and v_{16} could be arbitrary, and orders among v_9, v_7, v_8 can also be arbitrary. Rule 3 requires the orders of v_{10} lower than v_{11}, v_{12} lower than v_{13}, and v_{14} lower than v_{15} (suppose each partition's inner vertex follows the ID-increasing order). In this way, we can generate a set of node orders like $\langle v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}, v_{16}, v_9, v_7, v_8, v_1, v_2, v_3, v_4, v_5, v_6\rangle$, $\langle v_{10}, v_{16}, v_{13}, v_{12}, v_{11}, v_{14}, v_{15}, v_7, v_9, v_8, v_1, v_2, v_3, v_4, v_5, v_6\rangle$.

The following theorem proves that orders following the above definition are equivalent with the same index constructed.

Theorem 4. *The serial orders generated by the hierarchical parallel node ordering set are equivalent as they create the same tree structure and the same H2H label.*

Proof. Firstly, within each partition on the same layer, its tree structure is only determined by the contraction order of its inner vertices, but not the relative order with inner vertices from other partitions because they are separated by boundary vertices and do not have common neighbors. Secondly, as the parent partition's inner vertex order is always higher than their children's, the tree structure between layers is also fixed. Therefore, the tree structure is fixed as long as each partition's inner vertex order is relatively fixed, so this set of orders would construct the same tree decomposition, which further determines the same label. □

Now we can derive the following parallel contraction procedure that generates the equivalent order and label:

(1) The contraction is conducted in the unit of layers from lower to higher in the partition rank tree;
(2) The partitions of the same layer are contracted in parallel;
(3) The contraction within the same partition is contracted in serial.

Therefore, the degree of parallelism is the number of partitions of each layer, and the round of parallelism is the number of layers.

3.2 GPU-CPU Hybrid Contraction

In this section, we first present how to conduct the parallel contraction on GPU, then we propose a GPU-CPU hybrid contraction scheme to further improve contraction efficiency.

3.2.1 ***GPU Data Allocation.*** Unlike CPU, which can access memory randomly, GPUs' SIMD nature requires threads' data space to be pre-allocated. However, the contraction operations would insert new edges to the contracted vertex's neighbors $N_D(v_i)$ dynamically, which is unknown before all the vertices lower than it has been contracted. So it is impossible to accurately determine and allocate the data spaces. Nevertheless, as there must exist an upper bound of the data allocation for any vertex, in the following, we propose to estimate the contracted *Node size Upper Bound* (*NUB*) rather than the actual size.

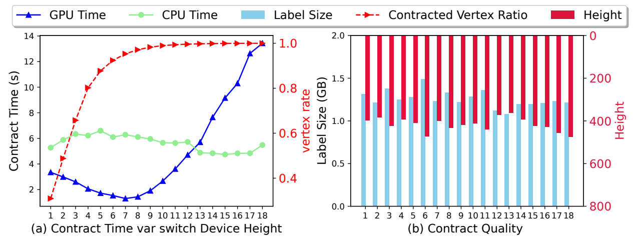


Figure 3: Contract Time and Index Quality Trend for FLA

Specifically, as all the contractions are conducted within partitions, then for any $v_i \in G_j$, we denote its node size upper bound as $NUB_{G_j}(v_i)$. Since the contraction would only increases $N_D(v_i)$ rather than reducing it, lower bound of $NUB_{G_j}(v_i)$ could be initialized as its current $|N_D(v_i)|$. Then, depending on the type of the vertex, we discuss them separately:

Case 1: *Inner vertex from the lowest layer.* Because all the inner vertices in one partition would be contracted, the newly added neighbor number could not exceed $|I_i|$. Besides, since the boundaries could also exist in $N_D(v_i)$, the maximum size is $|V_i|$. Because the initial value is no larger than $|V_i|$ (otherwise, it would contain vertices from other partitions), then the upper bound $NUB_{G_i}(v_i) = |V_i|$. It should be noted that V_i here refers to the contracted subgraphs but not the original partition results.

Case 2: *Boundary Vertex.* Because the boundaries do not contract in this round (layer) of contraction, its $N_D(v_i)$ increases by at most $|V_i|$. Besides, the boundaries also connect to boundaries from other partitions, which is denoted by $N_{out}(v_i)$. Then, the upper bound of $NUB_{G_i}(v_i) = |V_i| + |N_{out}(v_i)|$. It should be noted that $N_{out}(v_i)$ would also change during sub graph merging.

For example, the upper bound $NUB_{G_1^2}(v_{11})$ of the inner vertex v_{11} in sub-graph G_1^2 is $|V_1^2| = 4$; The upper bound $NUB_{G_2^2}(v_9)$ of the boundary vertex v_9 in sub-graph G_2^2 is $|V_2^2| + |N_{out}(v_9)| = 4+1 = 5$. Then the same v_9 would become an inner vertex in G_1^1, and this time its upper bound $NUB_{G_1^1}(v_9)$ is $max(|V_1^1|, NUB_{G_1^2}(v_9)) = max(4, 5) = 5$. It should be noted that NUB is the maximum possible $N_D(v_i)$ size but not the actual size.

3.2.2 ***Hybrid Contraction.*** Although GPU can provide high parallelism, it does not necessarily guarantee faster index construction for all the layers. As shown in Figure 3-(a), the GPU contraction is faster than GPU for the lower levels, but it soars up for the higher levels. This corresponds to the contracted vertex ratio, as the last few vertices form a nearly complete graph that can hardly be parallelized. Figure 3-(b) shows the tree height and label size, which are the smallest when the GPU time and CPU time intersect in (a).

Therefore, to take advantage of GPU's high efficiency and CPU's high quality, we propose the GPU-CPU hybrid contraction scheme. Because the network structure is rather stable, we test all the contracted layer and select the one with the fastest contraction time as the default threshold. After that, the remaining vertices would perform the global contraction on CPU in serial. More importantly, this trade-off ensures that the average degree is not too high during global contraction, avoids imbalance between partitions, and provides opportunity for order optimization.

3.3 Decomposition Tree Height Order

The order of vertices has a fundamental impact on every aspect of the distance labeling (construction time, index size, query time) as it determines the index structure. In the context of H2H, it is the tree height that determines the label size and the tree width that determines the query efficiency. A good order could reduce these two factors and provide better index quality. As discussed in Theorem 4, the partitioned-based parallel contraction generates a set of orders that are equivalent to a serial node order, which is unknown beforehand. Then, a question arises naturally: to what extent could this order be improved during parallel contraction?

We first analyze this order to identify the optimization opportunities: 1) the relative orders between layers cannot be changed as the partitions are contracted layer by layer, and the layers are determined by the balanced hierarchical partition algorithm; 2) the orders among the same-layer-different-partition inner vertices can still be arbitrary as they would not affect the tree structure; 3) it leaves the order of the same-partition inner vertices; 4) In the CPU contraction phase of the hybrid contraction, we have a chance to re-order all the remaining vertices. Therefore, in this section, we aim to improve the order of each inner partition vertices during their contraction. Specifically, the remaining vertices contracted on CPU serial can also served as a sub graph.

The essence of hop labeling is the graph vertex cut: for any two vertices s and t, a set of cut vertices exists such that if we remove them, s and t would belong to two separate components. Therefore, the shortest path must pass through at least one cut vertices. The common set of s and t's labels is their cuts, and it is achieved by using the LCA of s and t as the cuts in H2H. Obviously, the cut size (tree width), or N_D in our case, should be as small as possible. As N_D is formed through contraction, then the earlier contracted vertex would not appear in the latter contracted vertex's N_D. In other words, if a vertex wants to appear in more vertex pairs' cut, it should be contracted later with a higher order. Then the problem is converted to which vertex should have higher order and appear in more vertex pair's cut? Apparently, it should be the vertices that appear on more vertex pair's shortest paths, which can be quantified by the *Betweenness Centrality (BC)*. However, it takes $O(|V| \cdot (|V|log|V| + |E|))$ [8] to compute, which is intolerable during contraction. Fortunately, we only need to order the inner vertices in the subgraph, which determines this sub-tree's structure, so we only need an approximate BC on the tree.

Definition 3 (Decomposition Tree Height (DTH)). *Given a subgraph $G_D(V_D, E_D)$ during contraction, $\forall v_k \in V_D$, there exists a set $P_i = \langle v_1, v_2, v_3, ..., v_k \rangle$, where v_1 is the first decomposed vertex in P_i with $v_2 \in N_D(v_1)$, $v_3 \in N_D(v_2)$,..., and $v_k \in N_D(v_{k-1})$. Because it is a path, the size of P_i represents the lower bound of this partial tree height. If more than one such set exists, the largest of them is called v_k's DTH and denoted as $DTH_(v_k)$.*

Although the exact DTH can only be obtained after the tree is constructed, we can utilize it reversely to optimize the tree. Firstly, we present how to compute DTH accumulately. Algorithm 1 shows the pseudocode of contraction. We first initialize the DTH of all the inner vertices to 0. Then, after contracting a vertex v_t, we update all its neighbors' DTH (Line 12). Specifically, v_j's new DTH is the larger one of its current DTH and $DTH(v_t) + 1$. This is because DTH is always the longest one, and we can always accumulate it from the starting vertex as defined in Definition 3. This procedure is integrated into the contraction, so when a vertex is contracted, its exact DTH is determined. Besides, it does not introduce extra complexity. The remaining problem is selecting the next vertex to contract, and we propose the following order strategy:

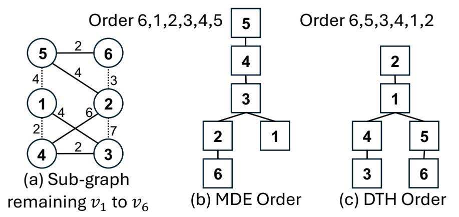


Figure 4: Decomposition Tree Example for Sub-graph G^0

Definition 4 (DTH Order Strategy). *The vertices are sorted first on $DTH(v_i) + N_D(v_i)$, then their current degree $|N_D|$, and finally, their ID to break a tie.*

To explain the intuition, suppose $DTH(v_i) < DTH(v_j)$ and they are neighbor to each other. If we contract v_j earlier than v_i, then $DTH(v_j)$ does not change but $DTH(v_i)$ would increase to larger than $DTH(v_j)$; On the other hand, if we contract v_i earlier, both of their DTH do not change as $DTH_(v_i) + 1 \leq DTH_(v_j)$.

Algorithm 1: DTH Optimized Contraction

```
   Data: Sub-graph $G_i$ and its inner vertex set $I_i$
   Result: Decomposition Tree $DT = \{N_D(v_i)\}$.
1  forall $v_j \in I_i$ do
2      $DTH(v_j) \leftarrow 0$;                                ▷ DTH Initialization
3  $PQ \leftarrow I_i$;                          ▷ Priority Queue with DTH Strategy
4  while $PQ$ not empty do
5      $v_t \leftarrow PQ$'s top vertex; $DT.insert(N_D(v_t))$;
6      forall Neighbor pair $(v_j, v_k)$ in $N_D(v_t)$ do
7          if $e(v_j, v_k)$ not exist or $e(v_j, v_k) > e(v_j, v_t) + e(v_t, v_k)$ then
8              Add or Update edge $e(v_j, v_k)$ in $G_i$;
9              $e(v_j, v_k) \leftarrow e(v_j, v_t) + e(v_t, v_k)$;
10     forall $v_j$ in $N_D(v_t)$ do
11         Delete edge $e(v_i, v_j)$ in $G_i$;
12         $DTH(v_j) = max(DTH(v_j), DTH(v_t) + 1)$
13     Delete $v_t$ from $G_i$ and $PQ$;
14 return $DT$;
```

For example, in Figure 4-(a), it is the last layer of the sub-graph to contract. The MDE order used in the original H2H generates a tree shown in (b). To utilize the DTH order, we first contract v_6 with minimum $DTH(v_6) + N_D(v_6) = 0 + 2 = 2$ and change $DTH(v_2) = max(DTH(v_6)+1, DTH(v_2)) = 1$, $DTH(v_5)=max(0+1, 0) = 1$. Next, v_5 with $DTH(v_5) + N_D(v_5) = 1 + 2 = 3$ and minimum $N_D(v_t) = 2$ is contracted. It change $DTH(v_2)$ to $max(DTH(v_5) + 1, DTH(v_2)) = max(1 + 1, 1) = 2$, $DTH(v_1)=max(1 + 1, 0) = 2$. Thirdly, v_3 with $DTH(v_3) + N_D(v_3) = 0 + 3 = 3$, $N_D(v_3)$=3 and ID=3 is selected and change $DTH(v_2) = max(DTH(v_3)+1, DTH(v_2)) = max(0+1, 2) = 2$, $DTH(v_4) = max(0 + 1, 0) = 1$. Obviously, DTH order generates a shorter tree (c) than MDE.

4 LABEL CONSTRUCTION WITH GPU

In this section, we present how to construct labels in parallel with GPU. This is the most time-consuming phase as it takes more than

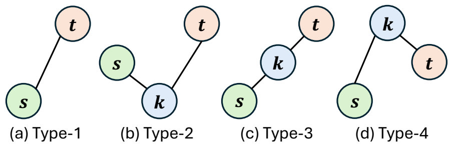


Figure 5: Path Types

half of the total index construction time. Specifically, we first analyze the path types that lay the foundation for the later analysis in Section 4.1. Then, we analyze the label structures to reduce the label construction workload in Section 4.2. After that, we present the GPU version of parallelized label construction with thread organization to unleash to full power in Section 4.3. Finally, we present how to answer distance queries in Section 4.4.

4.1 Path and Label Types

Firstly, depending on the orders of the starting vertex s, ending vertex t, and intermediate vertices k, we classify all the shortest paths into the following four types, as illustrated in Figure 5:

Type-1: A single edge with no intermediate vertex;
Type-2: Intermediate vertices are all lower than s and t;
Type-3: Intermediate vertices are all between s and t;
Type-4: A least one intermediate vertex is higher than s and t.

The above categorization is complete because 1) Type-1 and the other three divide the complete path set into with-intermediate and without-intermediate; 2) Type-2 distinguishes the without-intermediate with all lower than s and t, Type-3 covers the in-between, and Type-4 covers the remaining cases. In addition, these four types have no overlapping.

Because the contraction is conducted from the bottom up with lower vertices contracted first to preserve the shortest distance between two higher neighbors, the contraction result N_D only contains Type-1 and Type-2 paths [58], and the label construction phase further adds Type-3 and part of Type-4 paths into the index, with the remaining Type-4 paths computed on the fly during query with 2-hops. In the following, we analyze the completeness and redundancy of these labels.

Specifically, $\forall v_i \in G$, its ancestor vertices on the tree are denoted as $ANC(v_i)$. Then v_i and $ANC(v_i)$ together with their corresponding N_Ds forms a sub-graph denoted as $G_{v_i}(V_i, E_i)$. This sub-graph has the following properties: 1) $\forall v_k \in ANC(v_i)$, $r(v_k) > r(v_i)$; 2) E_i contains all information about paths passing through lower order vertices. For example in Figure 6, v_8's ancestor $ANC(v_8) = \{v_7, v_5, v_2, v_1\}$ forms subgraph G_{v_8} shaded in yellow, and v_{14}'s ancestor $ANC(v_{14}) = \{v_7, v_5, v_2, v_1\}$ forms $G_{v_{14}}$ shaded in blue.

To find the shortest distance from v_i to v_j, we take the union of their subgraphs to form a larger subgraph $G_{v_i+v_j}$, and this subgraph's information is sufficient [44]. Their vertex intersection $V_{v_i} \cap V_{v_j}$ forms a cut between v_i and v_j, so $dist(v_i, v_j) = min\{dist(v_i, v_k) + dist(v_k, v_j) | v_k \in V_{v_i} \cap V_{v_j}\}$. However, some of these two distances, like $dist(v_{14}, v_1)$ and $dist(v_8, v_2)$, do not exist yet due to the lack Type-3 and Type-4 indexes. The original H2H computes the distance from v_i to all of its $ANC(v_i)$, which covers all Type-3 and Type-4 paths. However, such an indiscriminate scheme introduces redundancy and cannot reveal the parallel opportunity. Therefore, we analyze the label construction and the corresponding data flow by incrementally adding Type-3 paths and Type-4 paths to identify

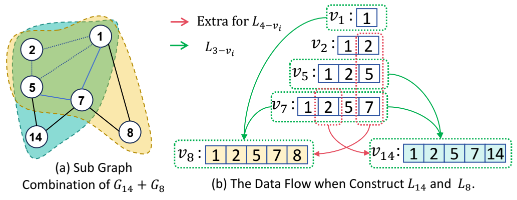


Figure 6: Combination of Two Sub Graphs $G_{v_8} + G_{v_{14}}$, and Data Flow Example for $L_{3-v_i} and L_{4-v_i}$.

the feasibility of parallel label construction. For those labels covered Type-1-2-3 paths, we denote them as $L_{3-total}$, while L_{3-v_i} for only v_i's. Similarly, for those labels covering all four path types, we record them as $L_{4-total}$ and L_{4-v_i} for v_i's. It should be noted that $L_{4-total}$ is the same as the original H2H labels.

4.2 Label Computation Pruning

In this section, we analyze the label construction in detail.

4.2.1 *H2H Label Construction Analysis.*

In H2H, each vertex regards all its ancestors as hubs and computes the distance to them. As the labels are constructed from tree root recursively, the computation can utilize the previously constructed higher-order labels. Therefore, we only need to describe the intermediate procedure of label construction. Specifically, suppose v_i is the current vertex to construct labels, then the labels of v_i would be $L_{v_i} = \{(v_i, v_j) | \forall v_j \in ANC(v_i)\}$. The number of labels to construct is $|ANC(v_i)|$. For each $L_{v_i}(v_j)$, it is computed as $min\{e(v_i, v_k) + L_{v_k}(v_j) | \forall v_k \in N_D(v_i)\}$. This is because $V_D(v_i)$ forms a cut from v_i to its ancestors, and these cut labels have already been computed. Finally, the computation number for each vertex label is $|ANC(v_i)| \times |N_D(v_i)|$.

For instance, in Figure 6-(b), $L_8(v_1) = min\{e(v_8, v_1), e(v_8, v_2) + L_{v_2}(v_1), e(v_8, v_5) + L_{v_5}(v_1), e(v_8, v_7) + L_{v_7}(v_1)\}$. The same amount of computations are required for $L_8(v_2)$, $L_8(v_5)$, and $L_8(v_7)$. The overall computation time is $4 \times 4 = 16$. As we can see, the above procedure does not consider the relative orders of v_k and v_j, so its computation involves all the Type-3 and Type-4 paths.

4.2.2 *Pruning with Type-3 Only Computation.*

In this section, we propose to reduce the label construction workload by only keeping the Type-3 paths in computation. In other words, all the labels are $L_{3-total}$. The following theorem proves its correctness:

THEOREM 5. *$L_{3-total}$ can find shortest distances with 2-hop.*

PROOF. Because $L_{3-total}$ contains all Type-1,2,3 shortest paths, it can find the shortest paths of these three types directly. Therefore, we only need to prove $L_{3-total}$ can find the shortest paths of Type-4. Specifically, during two-hop label concatenation, there could only be three cases: 1) No Type-4 paths, which can be handled with L_3 labels; 2) Requires one Type-4 path in the two labels; and 3) Require both labels to be Type-4 paths. Therefore, we only need to prove the latter two cases:

Case 1: As illustrated in Figure 7-(a), $p(s, u)$ and $p(u, t)$ are linked by hub vertex u, where $p(s, u)$ is a Type-4 path with highest rank hub v. Because $p(s, v)$, $p(u, v)$, and $p(t, u)$ are all Type-1,2,3 paths, there must exist labels of $L_{3-s}(v)$, $L_{3-u}(v)$ and $L_{3-t}(u)$. Further, $L_{3-t}(v)$ also exists because $p(t, v)$ is also Type-3 path.

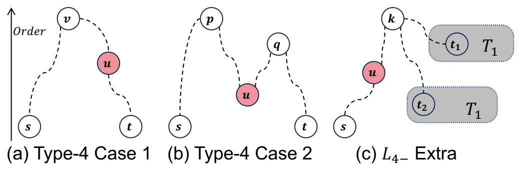


Figure 7: Type-4 Path Examples

Therefore, v exists in both s and t's labels, so such a shortest path $p(s, v, u, t)$ could be computed by $dist(s,t) = L_{3-t}(v) + L_{3-s}(v)$.

Case 2: $p(s,u)$ and $p(u,t)$ in Figure 7-(b) are two Type-4 paths with highest vertex p and q. Suppose $r(p) > r(q)$, p is the highest vertex in $p(p,q)$, then $p(p,q)$ is a Type-3 path and it exists in $L_{3-q}(p)$. Then path $p(s,p,q,t)$ is reduced to a Case 1 path, whose distance can be computed by $dist(s,t) = L_{3-t}(p)+L_{3-s}(p)$. □

For example, in Table 1, the third column lists the $L_{3-total}$ of the running example. The Case 1 shortest path $p(v_{14}, v_7, v_1, v_8)$ with hub v_7, shortest distance $dist(v_{14}, v_8)$ could be calculated by highest vertex v_1: $L_{3-v_{14}}(v_1) + L_{3-v_8}(v_1)$ =1+3=4. The Case 2 shortest path $p(v_{11}, v_6, v_{10}, v_2, v_9, v_{12})$ with hub v_{10}, $dist(v_{11}, v_{12})$ could be calculated by highest rank hub v_2: $L_{3-v_{11}}(v_2) + L_{3-v_{12}}(v_2)$ =4+2=6.

The cost of only utilizing Type-3 labels is during query answering, it can only use $L_s \cap L_t$ to find the common hops but cannot use LCA as the original H2H. Therefore, the query complexity increases from $O(max(|N_D(v_i)|))$ to $O(max(DTH))$. However, this cost is beneficial in real life because to utilize LCA, we need extra space to store the location of LCA for each label which takes $O(\Sigma|N_D(v_i)|)$ space and also a global LCA index. The saved space would allow us to make the best use of the precious GPU memory to answer queries on larger networks.

Algorithm 2: Label L_{3-v_i} Construction

```
Data: Decomposition Tree $DT = \{N_D(v_i)\}$ and $v_i$
Result: Label $L_{3-v_i}$
1 forall $(v_k, e(v_i, v_k)) \in N_D(v_i)$ in $r(k)$ do
2     forall label $L_{3-v_k}(v_j) \in L_{3-v_k}$ do
3         $L_{3-v_i}(v_j) = min(L_{3-v_k}(v_j) + e(v_i, v_k), L_{3-v_i}(v_j))$;
4 return $L_{3-v_i}$;
```

Algorithm 2 describes how to construct the L_3 labels L_{3-v_i} for v_i. Instead of iterating from the ancestor's perspective, we iterate from the N_D's perspective (line 1). This is because all the labels of $v_k \in N_D(v_i)$ all have higher orders than v_k and are Type-3 paths. Therefore, updating $v_j \in ANC(v_i)$ through v_k and v_k's Type-3 labels (line 2-3) does not involve Type-4 path and would not introduce Type-4 Path. The reduced computations are illustrated in the shade in Figure 7. **The overall computation number of L_{v_i} is** $\Sigma_{v_k \in N_D(v_i)}|ANC(v_k)|$. Because $|ANC(v_k)|$ smaller than $ANC(v_i)$ and it is decreasing as the order grows, the overall computation number is much smaller than the original's $|ANC(v_i)| \times |N_D(v_i)|$. Therefore, it is faster to construct labels.

Figure 6-(b) illustrates an example for constructing $L_{v_{14}}$ and L_{v_8}. When constructing L_{3-v_8}, all needed Type-3 paths are $p\langle v_8, v_7, v_5\rangle$, $p\langle v_8, v_7, v_2\rangle$, $p\langle v_8, v_7, v_1\rangle$. They all can be enumerated by equation $L_{3-v_8}(v_i) = L_{3-v_8}(v_7) + L_{3-v_7}(v_i)$, where $v_i \in \{v_5, v_2, v_1\}$. Type-4 paths $p\langle v_8, v_1, v_2\rangle$, $p\langle v_8, v_1, v_5\rangle$, $p\langle v_8, v_1, v_7\rangle$ are ignored but used to construct L_{4-v_8}. Same as v_{14}, five Type-3 paths can be enumerated by equation $L_{3-v_{14}}(v_i) = L_{3-v_{14}}(v_7) + L_{3-v_7}(v_i)$ and $L_{3-v_{14}}(v_j) = L_{3-v_{14}}(v_5) + L_{3-v_5}(v_j)$, where $v_i \in \{v_5, v_2, v_1\}$ and $v_j \in \{v_2, v_1\}$. Ignoring Type-4 path $p\langle v_8, v_5, v_7\rangle$. Compared to Tradition H2H, we reduces 13 over 16 for v_8 and 11 over 16 for v_{14} calculate time. Compared to $L_{4-total}$, we reduces 3 over 6 for v_8 and 1 over 6 for v_{14} calculate time.

4.3 Parallel Label Construction

In this section, we present how to construct labels with GPU through shared data allocation for smaller memory consumption and accurate estimation, BFS node-level ordering for coarse parallelism, and neighbor-level finer parallelism for more threads.

4.3.1 *Shared Data Allocation*.

In Algorithm 2, when computing v_i's labels L_{v_i}, we only need $N_D(v_i)$ and all its corresponding labels $\bigcup_{v_k \in N_D(v_i)} L_{v_k}$. Therefore, these labels share the same base information, and we save lots of memory by not allocating them separately. Furthermore, because $N_D(v_i) \subseteq ANC(v_i)$, we can further merge $N_D(v_i)$ into L_{v_i} as initial values to further reduce memory consumption. In terms of the memory size estimation, because the labels are constructed in a top-down manner, each time we go down one level on the tree, the label size increases by 1. Then, the actual data space size can be accumulated as the labels are constructed from parents to children. For example, in Figure 6, data space for $L_{v_{14}}$ can be calculated as $|L_{v_{14}}| = |L_{v_7}| + 1$. All vertices in $L_{v_{14}}$ can be allocated as $V_{v_7} \cup v_{14}$. All $N_D(v_{14})$ can be merged to $L_{v_{14}}$ before calculate label. Due to the high efficiency of GPU in transmitting large batches of whole data but the low efficiency of transmitting fragmented data, we organize the pre-allocated labels into a hash structure similar to Compressed-Sparse-Row-format (CSR) like Figure 8 and send them to GPU memory before calculation. Specifically, CSR contains a DIS array and a POS array. For each v_i, the DIS array stores the label distances from the tree root to its parent, and POS stores the relative locations of $L_{v_i}(N_D(v_i))$ in DIS.

4.3.2 *CPU and GPU Label Construction Parallelism*.

When constructing any label, due to the dependency on their ancestor labels, we can only parallelize the label construction of each different branch from top to bottom. We use the following frontier to organize this process efficiently and clearly:

Definition 5 (Frontier). *The set of label construction tasks $F_i = \{f_j\}$ that is selected to construct together in parallel in the i^{th} round is called a Frontier, and each $f_j = \{L_{v_k}\} \in F$ is taken care by one thread. $|F_i|$ is the frontier size that indicates its parallel degree, and the f_j with the largest workload determines F_i's running time.*

Then, the next problem is how to determine the frontiers. For example, with the tree in Figure 1-(c), we can have a frontier $F = \{\{L_{v_3}\}, \{L_{v_5}\}\}$ when L_{v_1} and L_{v_2} have finished, with $\{L_{v_3}\}$ and $\{L_{v_5}\}$ run in parallel. Or we can have $F = \{\{L_{v_3}, L_{v_9}, L_{v_{12}}, L_{v_{13}}\}, \{L_{v_{v_5}}, L_{v_{v_6}}, L_{v_{10}}, L_{v_{11}}\}\}$ under the same circumstance. However, the second one 1) has a smaller thread number that is wasteful for GPU, and 2) has an uncontrollable and unbalanced workload that further deteriorates efficiency. Therefore, we force each label construction task to be in the unit of L_{v_i} at this stage. In the following, we propose three possible strategies for CPU and GPU.

Strategy 1 (Preempt Frontier C-PF). *Given a thread pool of size λ, each thread only constructs one label set L_{v_i}. Whenever a L_{v_i} finishes, its children's labels are added to the thread pool.*

This strategy is suitable for CPU but not for GPU because even if f_i contains one L_{v_i}, the threads in frontier still have different workloads. What is worse, the preemption is against the GPU's SIMD synchronization nature, so the divergence is more severe. Furthermore, it requires maintaining a dynamic executable label queue, which can only be transferred from RAM to GPU, so the total amount of transmission explodes.

Strategy 2 (Non-Preempt Frontier G-NPF). *The frontier is organized rigidly level-by-level in the tree, with the labels at the same level forming a frontier and running together.*

This strategy is suitable for GPU because the workloads of threads in a frontier are nearly the same, which equals level number $l_i \times N_D(v_j)$, with the upper bound of $l_i \times max(N_D(v_j))$. Moreover, each F_i is fixed with dynamicity in the thread queue, so data transmission is small. What is better is that because the frontiers can be determined beforehand, we can transfer them to the GPU together before computation, allowing the GPU to focus on computing labels.

4.3.3 *Refine Parallel Granularity.*

Although GNP-F is better than C-PF on GPU, it still cannot outperform C-PF on CPU. This is because the massive threads on the GPU are still limited by the frontier size, which is the largest number of tree nodes at the same level. For instance, the maximum frontier size is 10K in FLA, which is still small for GPU and not enough to outperform CPU, while most of the remaining frontiers are even smaller. Therefore, we propose the following strategy to utilize GPU's threads better.

Strategy 3 (Label-Level Frontier G-LL). *Based on G-NF, each f_i is further decomposed from L_{v_j} to $\{L_{v_j}^{v_k} | \forall v_k \in N_D v_j\}$.*

Specifically, when constructing a label L_{v_i}, the concatenation through each vertex in $N_D(v_i)$ do not affect each other. Moreover, vertices in $N_D(v_i)$ at a higher level have larger degrees but fewer labels, while vertices at a lower level have smaller degrees but more labels. This phenomenon makes the workload at different levels more balanced. In addition, due to the average degree of decomposition of the road network being around 3 to 4, the average parallel granularity of the same layer increases by 3 to 4 times.

However, when using labels as frontiers, the same label may encounter write conflicts when calculating the minimum value. For example, when computing $L_{3-v_{13}}(v_3)$, the distance from $L_{3-v_{12}}(v_3)$ or $L_{4-v_9}(v_3)$ will cause a write conflict. Therefore, we use CUDA's atomic operation *atomicMin* to resolve it. What's more, *atomicMin* can solve the performance degradation caused by the use of branch statements at the same time. Finally, G-LL frontiers also exist in the L_4 labels so it can be transferred to GPU in intrinsically.

4.4 G2H Shortest Distance Query

Query Preparation. To reduce data transmission during query processing, the index can be pre-loaded into GPU. Specifically, when answering $q(s, t)$ we need all the labels to search common hops by $ANC(s) \cap ANC(t)$. Because GPU's memory is small compared to RAM, in order to use the same memory for larger networks, we propose not to store the actual label vertex but only the distance

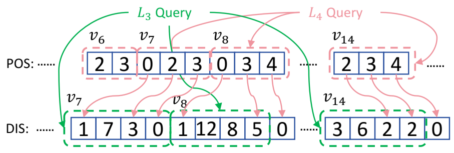


Figure 8: CSR Label Example and Query Addressing Times

array. When L_4 is used, a POS array is also needed. In addition, to find the LCA in constant time, an extra $O(|V| + |V| \times log_2(|V|))$ space is needed to store the *EULA* and *RMQ* index [6]. Finally, if the label is constructed by C-PF on CPU, we need to load it to GPU first; if the label is constructed by G-NPF on GPU, we only need to load the ones that are not in the GPU.

G2H Query Answering. Because labels are only read but not changed during query processing, the shortest distance query can run in parallel naturally. Unlike the CPU, which can answer the queries individually thread by thread, the GPU shines at processing a large number of queries while returning their results together. Besides, there is a limit for the number of threads within per block, and in our case, it is ≤1024. In order to trade-off between reducing the number of empty threads and responding to queries more quickly, we set it to 256 for the 1.5M queries set, because it has fewer empty threads and more threads per block.

In terms of the query procedure and efficiency, both L_3 and L_4 need to find $LCA(s, t)$ first. The difference is that L_3 does not need POS, so it needs the height of $LCA(s, t)$ to identify the length of their common ancestors in the label set. In other words, it traverses s and t's DIS from the first one value to the $h(LCA(s, t))^{th}$ value, and each pair along them is guaranteed to be the same common ancestor, while the latter ones are different. Besides, it takes two random accesses to locate the beginning of the two DIS arrays. If uses L_4, it takes three random accesses to locate LCA's POS and s and t's DIS arrays. Then, instead of linear scans, it takes $|N_D(LCA(s, t))|$ times of random accesses as vertices in $N_D(LCA(s, t))$ forms a smaller hub set than $ANC(s, t)$. Figure 8 shows the example of CSR Label and the procedures for L_3 and L_4 queries. When answering $q(v_8, v_{14})$, L_3 only needs 2 random accesses and two linear scans, while L_4 needs 1 random access for *POS* array in v_7, 2 random access to locate DIS arrays and $2 \times |N_D(v_7)| = 6$ random accesses to the 0^{th}, 2^{nd}, and 3^{rd} elements of them. Finally, when processing queries in batches on fewer threads CPU, the random access time cannot be ignored, which leads to lower efficiency on L_4. On the other hand, when using a large number of threads on GPU, the L_4 computation time saved by LCA can cover the insufficient addressing time, as revealed in Table 4.

5 PERFORMANCE STUDIES

5.1 Experimental Settings

Environment. We run experiments on a server with i5-13600k 3.5GHz 20 threads CPU, a NVIDIA RTX 4090 24GB GPU, 32GB memory, and running Ubuntu 22.04. All algorithms are implemented in CUDA C++ nvcc compiler with flags -O3 and -arch=sm_60.

Datasets. We use 13 public real-life networks shown in Table 2 from DIMACS [1] and OSM [43]. The *Height* represents the maximum Partition Tree Height, and *-P Layers* represents the layers of GPU contraction.

Table 2: Datasets

Data	Network	$\|V\|$	$\|E\|$	Partition Depth	-P Layer
NY	New York	264, 346	733, 846	17	6
BAY	San Francisco Bay Area	321, 270	800, 172	17	6
BJ	Beijing	296, 381	774, 660	17	9
COL	Colorado	435, 666	1, 057, 066	18	6
PAR	Paris	461, 542	1, 272, 524	18	7
FLA	Florida	1, 070, 376	2, 712, 798	19	7
NW	Northwest USA	1, 207, 945	2, 840, 208	19	9
NE	Northeast USA	1, 524, 453	3, 897, 636	19	7
CAL	California and Nevada	1, 890, 815	4, 657, 742	20	9
LKS	Great Lakes	2, 758, 119	6, 885, 658	20	8
EC	East China	3, 008, 173	7, 793, 146	21	12
E	Eastern USA	3, 598, 623	8, 778, 114	21	11
W	Wastern USA	6, 262, 104	15, 248, 146	22	12

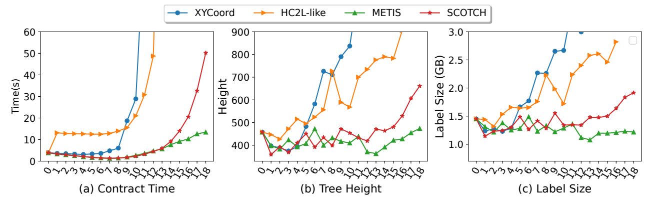


Figure 9: FLA Contract Time(s), Tree Height and Label Size(MB) var -P Layer and Partitioning Methods

Algorithms. 1) **H2H** [44]: Original serial H2H; 2) **HC2L and HC2L^p** [17]: Hierarchical Cut 2-hop Labelling and it's CPU parallel version; 3) **DTH_only** [25]: The top contract order is only sorted by DTH, not by $DTH + degree$; 3) **G2H-C** and **G2H**: Our methods using DTH order and G2H-C is the CPU version.

5.2 Overall Performance Comparison

We compare our algorithm with H2H and HC2L in index construction and quality, as shown in Table 3. More details about hierarchical node ordering version G2H-C are listed in Table 5.

Index Construction Time. Our method is 3.1× to 5.3× faster than H2H and 1.2× to 3.7× faster than HC2L^p. Specifically, it takes less than 2 seconds for urban megacity networks like New York, less than 5.2 seconds for state-level networks, less than half a minute for larger regions, and less than 1 minute for the largest W network. The CPU version that utilizes our orderings is also faster than the SOTA baselines, and its efficiency is comparable to the GPU version when the network is not too big. This is because the data allocation in GPU also takes time, and the detailed breakdown will be discussed later. Nevertheless, G2H is the only one that can construct index in W under 1 minute and its advantage grows larger when the network is bigger because more layers of partitions could utilize more threads.

Index Size and Tree Height. These two numbers are tightly related as the tree height determines the label size. Firstly, HC2L has the smallest tree height and smallest index size at the cost of longer construction and query time. This is because its label is constructed directly on the hierarchical partition results without tree decomposition. Secondly, G2H has a smaller index size and height than H2H on smaller networks, which proves the effectiveness of our DTH ordering. However, the benefit vanishes on the larger networks, since DTH on these graphs is too balanced, which loses its tree height-controlling power.

Graph Partitioning In this section, we compare the performance of different partitioned methods: *kd-Tree* [7] that partition the graph based on the medium value of x and y coordinates interchangeably; HC2L [17]; METIS [24] and SCOTCH [47]. As shown in Figure 9, METIS achieves the fastest contraction time of 1.305s when GPU contracted to layer 12 and the smallest label size of 1.103GB if further contracted to layer 6. SCOTCH is slightly worse. HC2L performs best when only contracting one layer in GPU, but its performance deteriorates dramatically for upper layers. Therefore, we use METIS as the default partition method and use the fastest layer number as the default. We also test the partition fanout of 2, 4, 8, 16, 32, and 64 on W. Their contraction time is generally increasing (11.04s, 11.86s, 12.49s, 15.12s, 18.46s, and 25.32s), while label sizes are nearly the same. Therefore, it validates that fanout=2 is the best setting.

5.3 Contraction Performance

5.3.1 *Effectiveness of DTH Order.*

Contraction Time. In Figure 10, the first three bars are the contraction time. As we can see, our G2H takes only 1/3 to 1/2 of the H2H's contraction time. In addition, the GPU time on the contraction is nearly invisible (detailed time is in Table 5), while the majority of the time is spent on the higher level's CPU global contraction. In fact, it is these fast GPU contractions that contribute most to the reduction in the overall contraction time. Finally, the DTH's reduction in tree height also contributes to reducing the global contraction time.

Label Quality. As shown in Table 5, DTH-enabled G2H tends to have smaller tree height and index but deteriorates to worse than H2H on larger networks. This is because the small networks are irregular at the city or state level then DTH has more chances to optimize order. In the larger networks, the difference between each vertex's DTH becomes insignificant, and its benefit is not sufficient to remedy the bad influence introduced by the partition order.

5.3.2 *Effectiveness of Parallelism.*

GPU vs CPU. As shown in Table 5, G2H-C conducts the parallel contraction on CPU while G2H on GPU. Both of them conduct the global contraction on CPU. When the graph is small, their time is similar. But then the graph becomes larger, the time saved by GPU is much larger. For instance, GPU parallel time on W only takes 3.87s while it takes 51.53s on CPU. In terms of the index size, CPU is better than GPU on large graphs, and this is caused by the sorting behavior in the dynamic sorting. Due to the thread limitation of GPU kernel function, we can only use bubble sort on it, so the order can hardly be the same. This results in most of the datasets requiring GPU to obtain the minimum label, while W requires CPU, although the contract on CPU takes more time.

5.4 Label Construction Performance

5.4.1 ***Influence of Allocation.*** As shown in the last two red bars in Figure 10, the dark red is the allocation time, and it takes the majority of the label construction time. The detailed allocation time is shown in Table 6. Specifically, the allocation time also contains the time of transferring the labels, BFS-frontier and TD-frontier from RAM to GPU. It can be seen that the small batch memory allocation of H2H actually occupies half of the label construction time. The Pre-allocated space allows H2H to focus on spatial addressing and computation, but not small dynamic data allocation.

Table 3: Index Construction Time, Index Size, and Tree Height

Data Set	Index Construction Time (s)						Index Size(MB)					Tree Height				
	H2H	HC2L	HC2L^P	DTH_only	**G2H-C**	**G2H**	H2H	HC2L	DTH_only	**G2H-C**	**G2H**	H2H	HC2L	DTH_only	**G2H-C**	**G2H**
NY	4.1	8.58	1.72	20.9	1.58	1.26	389.21	181	1147.07	341.65	304.77	506	25	1294	423	376
BAY	2.54	7.01	1.29	2.7	1.34	0.95	379.41	133	741.84	293.57	251.74	404	25	637	321	255
BJ	12.09	31.48	6.52	6.21	2.37	2.5	659.4	500	1088.84	649.25	623.31	720	31	1000	638	614
COL	3.71	12.65	2.95	3.37	2.37	1.55	568.57	238	1148.16	680.98	595.41	466	26	726	481	426
PAR	51.03	OOT	OOT	459.46	15.5	8.26	1544.04	/	5446.85	1975.08	1619.77	1054	/	3178	1222	987
FLA	10.47	33.97	7.29	7.89	8.09	2.96	1521.17	571	3034.32	1388.56	1260.67	521	30	804	417	400
NW	11.69	37.11	8.24	5.98	10.35	4.4	1790.14	649	3488.87	1610.84	1493.28	549	31	796	436	408
NE	29.93	98.24	18.67	77.24	21.17	8.38	3135.89	1615	9026.2	3650.67	3389.7	829	31	1714	791	693
CAL	29.4	89.77	20.22	55.92	22.54	9.05	3908.69	1669	11525.3	4103.83	3837.07	714	33	1647	681	650
LKS	102.08	241.72	47.56	198.88	64.55	20.34	9898.98	3791	22973.1	9313.68	8701.6	1326	31	2294	1034	986
EC	130.58	327.39	77.89	44.41	50.21	22.08	10556.5	4938	13699.7	9370.87	9007.28	1431	36	1376	950	947
E	83.23	311.65	66.37	40.09	78.79	21.66	10068	4440	16241.1	10539	10587.3	1023	33	1270	931	914
W	167.34	566.88	111.75	93.92	285.26	36.58	18273	7334	42146.2	17156.3	17921.5	1042	35	1849	919	931

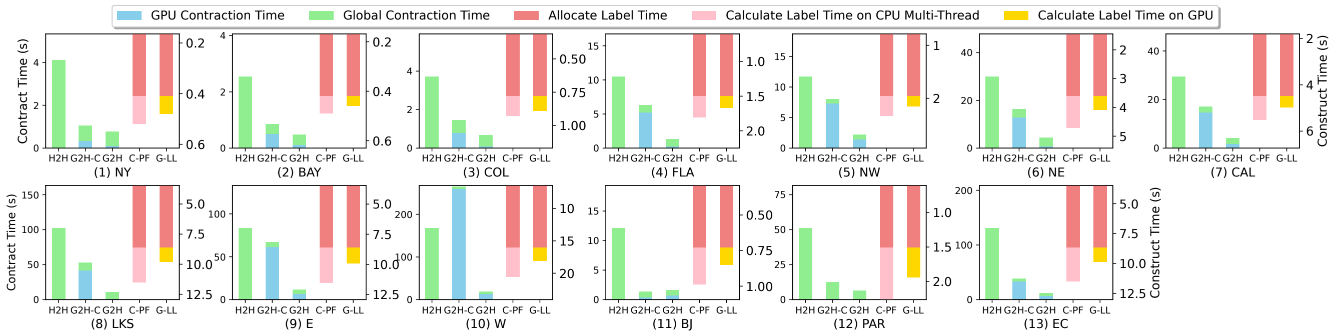


Figure 10: Contract Time Comparison for Contract Step and GPU Acceleration for Label Construction Step

Table 4: Query Time *ns* and Query per Second QpS

Data Set	Answer Query Time (*ns*)							Throughput (million / s)						
	H2H(L_4)	HC2L	G2H(L_3)	CPU-MT-L_3	CPU-MT-L_4	GPU-L_3	GPU-L_4	H2H(L_4)	HC2L	G2H(L_3)	CPU-MT-L_3	CPU-MT-L_4	GPU-L_3	GPU-L_4
NY	371.74	130.15	218.96	29.84	41.01	2.29	2.03	2.69	7.68	4.57	33.51	24.38	436.49	493.58
BAY	313.72	93.75	180.31	24.15	32.38	1.39	1.21	3.19	10.67	5.55	41.41	30.89	718.39	825.76
BJ	619.74	248.48	379.9	60.8	79.89	10.82	8.41	1.61	4.02	2.63	16.45	12.52	92.44	118.96
COL	550.35	182.81	363.32	48.62	61.6	7.3	4.82	1.82	5.47	2.75	20.57	16.23	136.93	207.47
PAR	859.11	/	553.98	100.65	121.66	21.32	16.02	1.16	/	1.81	9.94	8.22	46.9	62.43
FLA	566.14	169.17	334.49	38.98	60.08	4.16	3.64	1.77	5.91	2.99	25.65	16.65	240.5	274.57
NW	597.76	166.52	359.1	41.41	64.04	4.79	4.05	1.67	6.01	2.78	24.15	15.62	208.9	247.16
NE	756.8	229.99	467.92	60.41	84.14	8.6	7.46	1.32	4.35	2.14	16.55	11.89	116.31	134.12
CAL	763.11	284.01	483.02	61.47	85.13	9.43	7.36	1.31	3.52	2.07	16.27	11.75	106.07	135.94
LKS	793.5	327.78	519.19	71.12	92.35	10.79	8.47	1.26	3.05	1.93	14.06	10.83	92.67	118.06
EC	847.4	380.46	554.63	77.81	102.62	13.67	10.31	1.18	2.63	1.8	12.85	9.74	73.17	97.01
E	866.35	353.77	565.42	79.05	106.04	13.87	11.06	1.15	2.83	1.77	12.65	9.43	72.09	90.44
W	906.84	396.1	584.97	83.47	110.27	14.99	11.73	1.1	2.52	1.71	11.98	9.07	66.73	85.22

In the GPU version, extra time is needed to transfer labels and frontiers to the GPU, but as shown in Figure 10, the time for label allocation is almost the same. This is because the time required for a single large-scale transmission only accounts for a small portion of total allocation time. Specifically, if the pre-allocation strategy is not applied, the label construction on CPU will degrade to H2H. Besides, it cannot be implemented on GPU because after the label of each vertex is allocated in the global memory, too many pointers need to be saved, and the GPU pointer stack is unable to withstand such a large load. Finally, G-LL is the best strategy.

5.4.2 *Effectiveness of Parallel Label Construction.*

As shown in Figure 10, the light red is the CPU parallel, and the tiny yellow is the GPU contraction time. Because the original H2H label construction time is much longer than GPU, we list them in Table 6. Although constructing in label calculation with frontier generated by pre-BFS (C-PF) has reduced the label computation time by half, it is not sufficient enough for GPU parallelism (G-NPF) to cover

Table 5: Contraction Time and Quality of G2H and G2H-C

Data	G2H-C					G2H				
	Time (s)			Quality		Time (s)			Quality	
	Total	CPU	Global	Size (MB)	Height	Total	GPU	Global	Size (MB)	Height
NY	1.06	0.34	0.72	341.65	423	0.78	0.11	0.66	304.77	376
BAY	0.85	0.5	0.35	293.57	321	0.48	0.11	0.37	251.74	255
BJ	1.39	0.4	0.98	649.25	638	1.59	0.67	0.92	623.31	614
COL	1.45	0.79	0.66	680.98	481	0.67	0.08	0.59	595.41	426
PAR	12.67	0.73	11.94	1975.08	1222	6.32	0.13	6.19	1619.77	987
FLA	6.28	5.25	1.03	1388.56	417	1.3	0.23	1.08	1260.67	400
NWUSA	8.02	7.34	0.68	1610.84	436	2.24	1.48	0.76	1493.28	408
NEUSA	16.45	12.9	3.55	3650.67	791	4.3	0.69	3.61	3389.7	693
CALT	17.02	14.71	2.31	4103.83	681	4.07	1.74	2.34	3837.07	650
LKS	53.03	42.08	10.95	9313.68	1034	10.54	1.2	9.34	8701.6	986
EC	38.73	33.4	5.33	9370.87	950	12.18	7.17	5.01	9007.28	947
E	67.26	61.65	5.61	10539	931	11.75	6.57	5.17	10587.3	914
W	264.69	259.79	4.89	17156.3	919	18.45	12.95	5.5	17921.5	931

computation. Therefore, the computation branches and low parallelism lead to the longest construction time. Finally, because of the high parallelism and lower branches of the G-LL, it can finish construction in under 1s, which is 10× faster than C-PF.

5.4.3 *Effectiveness of L_3 Pruning.*

As shown in Table 6, when constructed in serial (indicating the actual workload), L_3 is always faster than L_4, so it proves the effectiveness of reducing the computation load. Moreover, some of the L_3 time is smaller than 1/3 of L_4,

Table 6: Label Construction Time Breakdown (s)

Data	H2H	G2H (Allocation + Computation)										
		Allocation	Compute L_4					Compute L_3				
			Serial	C-PF	C-LL	G-NPF	G-LL	Serial	C-PF	C-LL	G-NPF	G-LL
NY	1.24	0.41	0.29	0.11	0.63	1.35	0.07	0.24	0.1	0.63	0.59	**0.06**
BAY	0.84	0.42	0.2	0.07	0.37	0.46	0.04	0.17	0.07	0.24	0.36	**0.04**
BJ	1.91	0.73	0.68	0.26	1.29	5.65	0.18	0.47	0.17	2.16	1.18	**0.14**
COL	1.18	0.78	0.42	0.15	0.81	1.6	0.11	0.34	0.13	0.79	0.8	**0.09**
PAR	5.7	1.51	2.32	1.31	4.02	31.75	0.43	1.2	0.43	9.63	3.66	**0.32**
FLA	3.37	1.5	1.08	0.31	1.67	1.41	0.17	0.91	0.29	0.77	1.66	**0.15**
NW	3.5	1.96	1.18	0.37	1.82	1.59	0.2	1.01	0.35	0.85	1.8	**0.18**
NE	8.25	3.61	3.27	1.11	5.36	9.15	0.47	2.55	1	3.8	5.3	**0.42**
CAL	8.63	4.49	3.36	1.03	5.34	6.76	0.49	2.67	0.94	3.11	5.31	**0.44**
LKS	25.45	8.63	8.22	2.89	13.76	26.8	1.17	6.25	2.26	10.83	13.72	**1.05**
EC	32.83	8.68	9.4	2.8	15.09	20.91	1.21	7.47	2.37	9.3	15.3	**1.1**
E	22.65	8.62	8.64	2.91	14.64	20.44	1.29	6.98	2.56	8.71	14.68	**1.2**
W	42.49	16.06	14.07	4.52	23.81	19.02	2.08	11.72	3.87	9.88	24.14	**1.98**

which is lower than the theoretical acceleration ratio of 1/2. This reveals a phenomenon: the actual proportion of Type-3 paths could be higher than Type-4 paths.

5.5 Query Processing

We test the query time on 1.5M randomly generated queries and report the single query time and Query per Second in Tabel 4. We have tested 1 to 16 threads on CPU and find the query time is almost flat after 16. For GPU, we evaluate that 256 threads/block can achieve optimal performance. The first column set is the single thread CPU query answering, which reflects the quality of node order and index. Specifically, our method is always faster than the original H2H, but slower than the SOTA HC2L for most of the graphs. The second and the third column sets are the CPU parallel version and GPU parallel version. Generally, The CPU parallel can achieve 10× faster than the single thread version, and the GPU parallel is another 10× faster on top of it. Consequently, the GPU version can handle more than 100× more queries than the original H2H, which is hundreds of millions on a single machine with a single GPU. Finally, L_4 is slower than L_3 on CPU but faster on GPU. This is because L_3 does not need to find LCA and only needs to traverse the hop array, while L_4 needs to find LCA, traverse the position array, and then do the hop concatenation. Such addressing time will take up to 1/3 of the time. However, when querying on GPU, L_4 is faster because the parallel addressing time can be ignored rather than less computation, and achieves 1.1× speed up.

5.6 Hardware Influence

We first test the influence of CPU in Table 7. The construction time of 8753C is around 1.3× slower than i5, which corresponds to the CPU frequency difference. As for the query performance, although 8375C has 6× more threads than i5, its average query efficiency is only 2.4× faster. Then we test the influence of GPU. Because 4060 has a smaller memory, we only show the results that it can hold. As shown in Table 8, 4090 is around 2-3× faster in construction and 3-4× faster in query, which corresponds to the CUDA core numbers.

6 RELATED WORK

2-Hop Labeling [12]. It is the most efficient class of index for shortest distance answering. They are also characterized by large index size and slow index construction, which stops them from being applicable in real-life dynamic scenarios. They can be roughly categorized into two types: 1) Cut-based methods like *TEDI* [53], *m-Hop* [9], *H2H* [44], *P2H* [10], and *HC2L* [17]. These methods have a tree structure to capture the cut information such that queries can be answered faster. Among them, *H2H* is the fastest one to construct. Nevertheless, these methods are only suitable for networks with small treewidth like road networks; 2) Pruning-based methods like *PLL* [5], *GLL* [27], *PSL* [32], and *PCL* [68]. These methods use predefined node orders for pruning and do not have extra structure to organize cuts so they can easily extend to parallel and can scale to any network. However, their query performance is generally slower than the cut-based methods.

GPU-based Graph Processing. GPU-based computation has shown success in various graph computation tasks, like BFS [35], path search [14], constraint path search [40], PageRank [20], subgraph enumeration [21], nearest neighbor search [55], betweenness centrality [50], and biclique counting [48]. However, none of them works on the hop labeling.

Table 7: Influence of CPU

Data	Xeon(R) Platinum 8375C 2.9GHz (128 threads)						i5-13600K 3.5GHz (20 threads)					
	Contract (s)	Construct (s)			Query (ns)		Contract (s)	Construct (s)			Query (ns)	
	Global	Serial	C-PF	C-LL	L_3	L_4	Global	Serial	C-PF	C-LL	L_3	L_4
NY	0.89	0.36	0.54	1.19	14.46	19.4	0.59	0.29	0.11	0.63	29.84	41.01
BAY	0.42	0.24	0.43	0.78	11.51	13.94	0.29	0.2	0.07	0.37	24.15	32.38
BJ	1.39	0.86	0.77	2.03	34.21	35.69	0.82	0.68	0.26	1.29	60.8	79.89
COL	0.74	0.53	0.57	1.32	27.51	27.56	0.49	0.42	0.15	0.81	48.62	61.6
PAR	9.1	3.02	1.51	4.97	51.31	55.44	6.13	2.32	1.31	4.02	100.65	121.66
FLA	1.08	1.44	0.76	2.29	18.69	21.62	0.79	1.08	0.31	1.67	38.98	60.08
NW	0.69	1.59	0.79	2.18	20.44	26.77	0.45	1.18	0.37	1.82	41.41	64.04
NE	4.94	4.29	1.49	5.84	30.84	36.6	3.2	3.27	1.11	5.36	60.41	84.14
CAL	2.76	4.63	1.36	6.09	30.69	33.39	1.79	3.36	1.03	5.34	61.47	85.13
LKS	13.39	10.95	2.84	13.19	33.72	38.54	8.48	8.22	2.89	13.76	71.12	92.35
EC	6.46	12.96	3.19	13.7	40.08	42.21	3.99	9.4	2.8	15.09	77.81	102.62
E	6.68	11.99	2.89	13.8	39.25	44.06	4.1	8.64	2.91	14.64	79.05	106.04
W	5.83	19.53	4.67	18.45	43.3	48.11	3.67	14.07	4.52	23.81	83.47	110.27

Table 8: Influence of GPU

Data	NVidia 4090 24GB 16384 CUDA Cores					NVidia 4060ti 8GB 4352 CUDA Cores				
	Contract	Construct		Query		Contract	Construct		Query	
	Partition	G-PF	G-LL	L_3	L_4	Partition	G-PF	G-LL	L_3	L_4
NY	0.11	1.35	0.07	2.29	2.03	0.12	1.41	0.12	9.63	8.38
BAY	0.1	0.46	0.04	1.39	1.21	0.11	0.51	0.09	6.18	5.27
BJ	0.7	5.65	0.18	10.82	8.41	0.7	5.87	0.24	38.5	32.13
COL	0.08	1.6	0.11	7.3	4.82	0.08	1.74	0.18	24.92	17.97
PAR	0.13	31.75	0.43	21.32	16.02	0.14	32.99	0.62	73.54	60.22
FLA	0.24	1.41	0.17	4.16	3.64	0.23	1.58	0.25	14.69	11.96
NW	1.5	1.59	0.2	4.79	4.05	1.53	1.77	0.28	15.32	13.24
NE	0.69	9.15	0.47	8.6	7.46	0.72	9.82	0.72	27.63	25.93
CAL	1.72	6.76	0.49	9.43	7.36	1.79	7.43	0.76	29.51	24.44

7 CONCLUSION

In this paper, we propose the first GPU-enabled shortest distance index G2H through conflict-free hierarchical graph partition and parallel contraction, DTH vertex order optimization, label pruning, and thread organization. As a result, the index construction time is up to 5.3× faster than the original H2H. Consequently, we are able to re-construct a distance index in a few seconds for large urban cities and states, and under one minute for large networks with more than 6 million vertices. Furthermore, it breaks the distance QPS at hundreds of millions (10^8) on a single machine for the first time. This remarkable efficiency outperforms the existing index maintenance in highly dynamic environments and finally makes hop labeling practically in real-life applications.

ACKNOWLEDGMENTS

This work is supported by Key Research and Development Program of Liaoning Province under Grant No.2023JH26/10300022, Natural Science Foundation of China #62202116, Guangzhou-HKUST(GZ) Joint Funding Scheme #2023A03J0135, Guangzhou Basic and Applied Basic Research Scheme #2024A04J4455, Guangdong-Hong Kong Technology Innovation Joint Funding #2024A0505040012, and Guangzhou municipality big data intelligence key lab #2023A03J0012.

REFERENCES

[1] [n.d.]. 9th DIMACS Implementation Challenge - Shortest Paths. http://users.diag.uniroma1.it/challenge9/download.shtml.

[2] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2012. Hierarchical Hub Labelings for Shortest Paths. In *Algorithms – ESA 2012*, Leah Epstein and Paolo Ferragina (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 24–35.

[3] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2017. Engineering a direct k-way hypergraph partitioning algorithm. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 28–42.

[4] Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata. 2014. Fast shortest-path distance queries on road networks by pruned highway labeling. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments* (Portland, Oregon). Society for Industrial and Applied Mathematics, 147–154.

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

[6] Michael A Bender and Martin Farach-Colton. 2000. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics: 4th Latin American Symposium*. Springer, 88–94.

[7] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[8] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001), 163–177.

[9] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Hong Cheng, and Miao Qiao. 2012. The exact distance to destination in undirected world. *The VLDB Journal* 21 (2012), 869–888.

[10] Zitong Chen, Ada Wai-Chee Fu, Minhao Jiang, Eric Lo, and Pengfei Zhang. 2021. P2H: Efficient Distance Querying on Road Networks by Projected Vertex Separators. In *SIGMOD* (Virtual Event, China). ACM, 313–325.

[11] Theodoros Chondrogiannis and Johann Gamper. 2016. ParDiSP: A partition-based framework for distance and shortest path queries on road networks. In *2016 17th IEEE International Conference on Mobile Data Management (MDM)*, Vol. 1. IEEE, 242–251.

[12] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2002. Reachability and distance queries via 2-hop labels. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, California) *(SODA '02)*. Society for Industrial and Applied Mathematics, 937–946.

[13] Tangpeng Dan, Xiao Pan, Bolong Zheng, and Xiaofeng Meng. 2023. Double Hierarchical Labeling Shortest Distance Querying in Time-dependent Road Networks. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2077–2089.

[14] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. 2014. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 349–359.

[15] Daniel Delling, Andrew V Goldberg, Ilya Razenshteyn, and Renato F Werneck. 2011. Graph partitioning with natural cuts. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 1135–1146.

[16] E. W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1, 1 (dec 1959), 269–271.

[17] Muhammad Farhan, Henning Koehler, Robert Ohms, and Qing Wang. 2023. Hierarchical Cut Labelling - Scaling Up Distance Queries on Road Networks. *SIGMOD* 1, 4, Article 244 (dec 2023), 25 pages.

[18] Muhammad Farhan, Qing Wang, and Henning Koehler. 2022. Batchhl: Answering distance queries on batch-dynamic networks at scale. In *Proceedings of the 2022 International Conference on Management of Data*. 2020–2033.

[19] Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin-Yu Chen, Xiao-Fei Liao, and Hai Jin. 2019. A survey on graph processing accelerators: Challenges and opportunities. *Journal of Computer Science and Technology* 34 (2019), 339–371.

[20] Wentian Guo, Yuchen Li, Mo Sha, and Kian-Lee Tan. 2017. Parallel personalized pagerank on dynamic graphs. *Proceedings of the VLDB Endowment* 11, 1 (2017), 93–106.

[21] Wentian Guo, Yuchen Li, and Kian-Lee Tan. 2020. Exploiting reuse for gpu subgraph enumeration. *IEEE Transactions on Knowledge and Data Engineering* 34, 9 (2020), 4231–4244.

[22] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[23] Manuel Hotz, Theodoros Chondrogiannis, Leonard Wörteler, and Michael Grossniklaus. 2021. Online landmark-based batch processing of shortest path queries. In *Proceedings of the 33rd International Conference on Scientific and Statistical Database Management*. 133–144.

[24] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[25] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. 2010. Distributed time-dependent contraction hierarchies. In *Experimental Algorithms: 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings 9*. Springer, 83–93.

[26] Deyu Kong, Xike Xie, and Zhuoxu Zhang. 2022. Clustering-based partitioning for large web graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 593–606.

[27] Kartik Lakhotia, Rajgopal Kannan, Qing Dong, and Viktor Prasanna. 2019. Planting trees for scalable and efficient canonical hub labeling. *Proceedings of the VLDB Endowment* 13, 4 (2019), 492–505.

[28] Jiajia Li, Cancan Ni, Dan He, Lei Li, Xiufeng Xia, and Xiaofang Zhou. 2023. Efficient k NN query for moving objects on time-dependent road networks. *The VLDB Journal* 32, 3 (2023), 575–594.

[29] Jiajia Li, Xing Xiong, Lei Li, Dan He, Chuanyu Zong, and Xiaofang Zhou. 2023. Finding Top-k Optimal Routes with Collective Spatial Keywords on Road Networks. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 368–380.

[30] Lei Li, Sibo Wang, and Xiaofang Zhou. 2019. Time-dependent hop labeling on road network. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 902–913.

[31] Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. 2020. Fast Query Decomposition for Batch Shortest Path Processing in Road Networks. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1189–1200.

[32] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Scaling Distance Labeling on Small-World Networks. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands). Association for Computing Machinery, 1060–1077.

[33] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2020. Scaling up distance labeling on graphs with core-periphery properties. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1367–1381.

[34] Ye Li, Leong Hou U, Man Lung Yiu, and Ngai Meng Kou. 2017. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment* 11, 4 (2017), 445–457.

[35] Hang Liu and H Howie Huang. 2015. Enterprise: breadth-first graph traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[36] Ziyi Liu, Lei Li, Mengxuan Zhang, Wen Hua, Pingfu Chao, and Xiaofang Zhou. 2021. Efficient constrained shortest path query answering with forest hop labeling. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1763–1774.

[37] Ziyi Liu, Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. 2022. FHL-cube: multi-constraint shortest path querying with flexible combination of constraints. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3112–3125.

[38] Ziyi Liu, Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. 2024. Approximate Skyline Index for Constrained Shortest Pathfinding with Theoretical Guarantee. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4222–4235.

[39] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *PVLDB* 5, 8 (apr 2012), 716–727.

[40] Shengliang Lu, Bingsheng He, Yuchen Li, and Hao Fu. 2020. Accelerating exact constrained shortest paths on GPUs. *PVLDB* 14, 4 (dec 2020), 547–559.

[41] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[42] Ruben Mayer and Hans-Arno Jacobsen. 2021. Hybrid edge partitioner: Partitioning large power-law graphs under memory constraints. In *Proceedings of the 2021 International Conference on Management of Data*. 1289–1302.

[43] OpenStreetMap contributors. 2017. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org.

[44] 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 *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA). ACM, 709–724.

[45] Dian Ouyang, Dong Wen, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Progressive top-k nearest neighbors search in large road networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1781–1795.

[46] 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. *PVLDB* 13, 5 (jan 2020), 602–615.

[47] François Pellegrini and Jean Roman. 1996. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking: International Conference and Exhibition HPCN EUROPE 1996 Brussels, Belgium, April 15–19, 1996 Proceedings 4*. Springer, 493–498.

[48] Linshan Qiu, Zhonggen Li, Xiangyu Ke, Lu Chen, and Yunjun Gao. 2024. Accelerating Biclique Counting on GPU. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 3191–3203.

[49] Neil Robertson and Paul D Seymour. 1984. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B* 36, 1 (1984), 49–64.

[50] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V Çatalyürek. 2013. Betweenness centrality on GPUs and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. 76–85.

[51] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph processing on GPUs: A survey. *ACM Computing Surveys (CSUR)* 50, 6 (2018), 1–35.

[52] Jeppe Rishede Thomsen, Man Lung Yiu, and Christian S Jensen. 2012. Effective caching of shortest paths for location-based services. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 313–324.

[53] Fang Wei. 2010. TEDI: efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 99–110.

[54] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M Tamer Özsu, Wei-Shinn Ku, and John CS Lui. 2020. G-thinker: A distributed framework for mining subgraphs in a big graph. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1369–1380.

[55] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2022. GPU-accelerated proximity graph approximate nearest Neighbor search and construction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 552–564.

[56] Ziqiang Yu, Xiaohui Yu, Nick Koudas, Yueting Chen, and Yang Liu. 2024. A Distributed Solution for Efficient K Shortest Paths Computation Over Dynamic Road Networks. *IEEE Transactions on Knowledge and Data Engineering* (2024).

[57] Yuanyuan Zeng, Yixiang Fang, Chenhao Ma, Xu Zhou, and Kenli Li. 2024. Efficient Distributed Hop-Constrained Path Enumeration on Large-Scale Graphs. *SIGMOD* 2, 1 (2024), 1–25.

[58] Mengxuan Zhang, Lei Li, Wen Hua, Rui Mao, Pingfu Chao, and Xiaofang Zhou. 2021. Dynamic hub labeling for road networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 336–347.

[59] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2019. Efficient Batch Processing of Shortest Path Queries in Road Networks. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*. 100–105.

[60] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2020. Stream processing of shortest path query in dynamic road networks. *IEEE Transactions on Knowledge and Data Engineering* 34, 5 (2020), 2458–2471.

[61] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2021. Efficient 2-Hop Labeling Maintenance in Dynamic Small-World Networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 133–144.

[62] Mengxuan Zhang, Lei Li, Goce Trajcevski, Andreas Züfle, and Xiaofang Zhou. 2023. Parallel hub labeling maintenance with high efficiency in dynamic small-world networks. *IEEE Transactions on Knowledge and Data Engineering* 35, 11 (2023), 11751–11768.

[63] Mengxuan Zhang, Lei Li, and Xiaofang Zhou. 2021. An experimental evaluation and guideline for path finding in weighted dynamic network. *PVLDB* 14 (2021), 2127–2140.

[64] Mengxuan Zhang, Xinjie Zhou, Lei Li, Ziyi Liu, Goce Trajcevski, Yan Huang, and Xiaofang Zhou. 2023. A Universal Scheme for Partitioned Dynamic Shortest Path Index. *arXiv preprint arXiv:2310.08213* (2023).

[65] 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* (2021).

[66] Bolong Zheng, Yong Ma, Jingyi Wan, Yongyong Gao, Kai Huang, Xiaofang Zhou, and Christian S. Jensen. 2023. Reinforcement Learning based Tree Decomposition for Distance Querying in Road Networks. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 1678–1690.

[67] Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, Lizhu Zhou, and Zhiguo Gong. 2015. G-Tree: An Efficient and Scalable Index for Spatial Search on Road Networks. *IEEE Transactions on Knowledge and Data Engineering* 27, 8 (2015), 2175–2189.

[68] Xinjie Zhou, Mengxuan Zhang, Lei Li, and Xiaofang Zhou. 2024. Scalable Distance Labeling Maintenance and Construction for Dynamic Small-World Networks. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4573–4585.

[69] Xinjie Zhou, Mengxuan Zhang, Lei Li, and Xiaofang Zhou. 2025. High Throughput Shortest Distance Query Processing on Large Dynamic Road Networks. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE.