Parallel Multi Objective Shortest Path Update Algorithm in Large Dynamic Networks

S. M. Shovan, *Student Member, IEEE*, Arindam Khanda, *Member, IEEE*, and Sajal K. Das, *Fellow, IEEE*

***Abstract*—The multi objective shortest path (MOSP) problem, crucial in various practical domains, seeks paths that optimize multiple objectives. Due to its high computational complexity, numerous parallel heuristics have been developed for static networks. However, real-world networks are often dynamic where the network topology changes with time. Efficiently updating the shortest path in such networks is challenging, and existing algorithms for static graphs are inadequate for these dynamic conditions, necessitating novel approaches. Here, we first develop a parallel algorithm to efficiently update a single objective shortest path (SOSP) in fully dynamic networks, capable of accommodating both edge insertions and deletions. Building on this, we propose *DynaMOSP*, a parallel heuristic for Dynamic Multi Objective Shortest Path searches in large, fully dynamic networks. We provide a theoretical analysis of the conditions to achieve Pareto optimality. Furthermore, we devise a dedicated shared memory CPU implementation along with a version for heterogeneous computing environments. Empirical analysis on eight real-world graphs demonstrates that our method scales effectively. The shared memory CPU implementation achieves an average speedup of 12.74× and a maximum of 57.22×, while on an Nvidia GPU, it attains an average speedup of 69.19×, reaching up to 105.39× when compared to state-of-the-art techniques.**

***Index Terms*—Multi-objective shortest path, dynamic graph, GPU, shared-memory, SYCL programming model.**

I. Introduction

FINDING the shortest path is a classic problem in graph theory with applications ranging from optimal route suggestions in road networks [1], centrality analysis in social networks [2], enhancing communication in diverse network topologies [3], to aiding drone-based deliveries [4], [5], among others. However, determining the shortest path becomes challenging in dynamic scenarios where the network topology changes frequently. The complexity escalates to NP-hard when the search for the shortest path optimizes multiple objective functions [6]. Such paths, known as multi-objective shortest paths (MOSP), are widely applicable in various practical scenarios, as outlined below.

i) In the context of a *road transportation network*, it is commonly desired to identify the most favorable route by considering factors such as distance, travel duration, congestion, road conditions, fuel economy, and safety.
ii) Planning flight paths in *aerospace* considers factors like carbon emission, weather conditions, and flight time [7].
iii) In *wireless sensor networks (WSNs)*, it is a standard practice to send the data to a central data aggregation point, known as the sink. This transmission is facilitated through a tree structure rooted at the sink. Minimizing the distance between sensor nodes and the sink in a route can reduce the latency of data collection. However, this approach may lead to rapid energy depletion in sensor nodes that are located closer to the sink. Consequently, this has the potential to negatively impact the total lifespan of the network. Therefore, it is important to jointly optimize the delay and energy usage in the data aggregation routes in WSNs [8].
iv) The *supply chain network* helps to deliver the product in a timely manner. As the consciousness of the environment is rising, it is expected to balance CO_2 emissions while maximizing responsiveness to demands [8].

In the aforementioned cases, the task of finding the most optimal route can be modeled as an MOSP problem where the objectives are not correlated. The most established method for addressing multi-objective optimization challenges involves employing Pareto optimality. This technique provides a set of solutions and ensures that no solution is inferior to any other feasible solution across all objectives [9]. However, finding all possible optimal solutions, especially in a large network is a computationally hard problem as the number of candidate solutions grows exponentially [10]. The challenge intensifies when the network is dynamic.

Various optimization techniques and parallel approaches have been explored in the literature to address the intense computational demands of MOSP problems in large *static networks*. Strategies include mapping multi-objective to single-objective problems [11], approximating the optimal solution [12], ranking the Pareto optimal candidates to limit the exponential growth of candidate paths using survival analysis [13], etc. Trivially, static graph algorithms can be applied to recompute MOSPs after each set of topological changes in dynamic networks. However, this approach is inefficient as it requires repeated and redundant calculations of NP-hard MOSP problem. As real-world problems are complex and large in volume, they make this trivial approach nonviable.

Recent research shows that the recalculation of graph properties is avoidable by exploiting the previous knowledge about the graph property, proper identification of the affected subgraph, and strategic recomputation in a bounded network region [14],

Received 2 May 2024; revised 7 January 2025; accepted 20 January 2025. Date of publication 30 January 2025; date of current version 7 April 2025. This work was supported by NSF under Grant ECCS-2319995, Grant OAC-2104078, and Grant CNS-2030624. Recommended for acceptance by M. Li. *(S. M. Shovan and Arindam Khanda contributed equally to this work.) (Corresponding author: S. M. Shovan.)*

The authors are with the Department of Computer Science, Missouri University of Science and Technology, Rolla, MO 65409 USA (e-mail: sskg8@mst.edu; akkcm@mst.edu; sdas@mst.edu).

This article has supplementary downloadable material available at https://doi.org/10.1109/TPDS.2025.3536357, provided by the authors.

Digital Object Identifier 10.1109/TPDS.2025.3536357

[15], [16]. These methods maintain the result's quality while reducing the execution time, which is the main motivation for our research efforts.

In this study, we extend our previous work on MOSP search in incremental networks [16], which supported only edge insertion, to fully dynamic networks that handle simultaneous edge insertions and deletions, addressing new challenges in our algorithm design. We start by developing a parallel algorithm to update SOSP in large dynamic networks, then introduce a novel heuristic, DynaMOSP, for MOSP search. DynaMOSP is adaptable to any number of objectives and incorporates user preferences to quickly identify a single shortest path among all possible Pareto optimal shortest path candidates in large dynamic networks. It utilizes a novel grouping technique to avoid critical sections in parallel threads, enhancing scalability. Our key contributions are as follows.

- We propose a parallel algorithm to update the Single Objective Shortest Paths (SOSP) in large, fully dynamic networks. Our algorithm utilizes a grouping technique to minimize the total computational effort while guaranteeing correctness.
- Leveraging the efficiency of our SOSP-update algorithm, we develop a novel heuristic algorithm, *DynaMOSP*, capable of swiftly updating a single Multi-Objective Shortest Path (MOSP) in large networks experiencing time-varying dynamics.
- For all proposed algorithms, we offer implementations based on SYCL for heterogeneous computing architectures, alongside specialized implementations using OpenMP for shared memory CPUs.
- Through extensive experimental analysis, including scalability tests and speedup comparisons on large real networks, we have demonstrated the effectiveness of our proposed methods. Our shared memory implementation of *DynaMOSP* outperforms state-of-the-art techniques by an average of $12.74\times$. On average, our GPU implementation (SYCL-based) achieves a speedup of over $69.19\times$.

The paper is structured as follows: Section II covers essential preliminaries including SOSP, MOSP, Pareto optimal shortest paths, and dynamic networks. Section III details DynaMOSP for updating SOSP and MOSP in fully dynamic networks. Section IV discusses the implementation specifics for shared-memory and GPU environments. Performance evaluation and comparisons with existing methods are provided in Section V. Section VI reviews prior work on SOSP and MOSP. The paper concludes in Section VII, outlining future research directions.

II. Preliminaries

Consider a directed network denoted as $G(V, E)$, where V is the set of vertices and E is the set of edges. An edge $e(u, v) \in E$ originates from vertex u and terminates at vertex v and is assigned a non-negative weight $W(e)$. In a complex system, the edge weight can represent a composite of values tied to various objectives. A path is a sequence of vertices $(u_1, u_2, \ldots, u_x)$ where each consecutive vertex pair u_i, u_j is linked by a directed edge $e = (u_i, u_j) \in E$. A directed path between two vertices is the *shortest* if the sum of its edge weights is minimal.

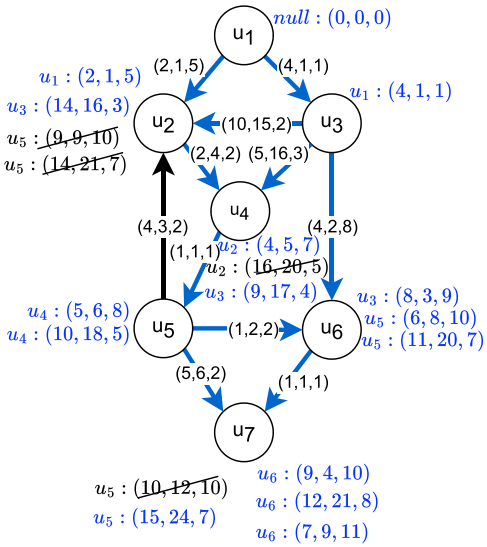


Fig. 1. MOSP computation.

A. Single Versus Multi-Objective Shortest Paths

Based on the number of objectives k, the shortest path problem can be categorized into: a) *Single Objective Shortest Path (SOSP)*, which addresses just one objective ($k = 1$); and b) *Multi-objective Shortest Path (MOSP)*, applicable when $k \geq 2$. The solution to the SOSP problem yields a tree structure known as *SOSP tree*, (Let T) containing the edges that constitute the shortest paths from the source vertex.

In a multi-objective scenario, where $k \geq 2$, the weight of an edge $e = (u, v)$ can be denoted as a weight vector $W(e) = (w_1, \ldots, w_k)$. The $i^{th}(2 \leq i \leq k)$ term of W represents the edge weight between u and v measured using i^{th} objective function only. Consequently, the problem of finding the shortest path is tasked with optimizing multiple objective functions evolving into a MOSP search. Pareto optimization is a recognized method in dealing with multi-objective optimization problems, delivering solutions that are not inferior to any other Pareto optimal solutions [17]. Let the minimal distance, denoted by Pareto optimal labels, of a vertex u from the source vertex be $(u, \overrightarrow{l}) = \{\{d_1^{p1}, \ldots, d_{|p1|}^{p1}\}, \ldots, \{d_1^{pz}, \ldots, d_{|pz|}^{pz}\}\}$ where d_j^{pi} represents the j^{th} distance label along the Pareto optimal path through parent vertex p_i. Every label d_j^{pi} corresponds to individual distance component. Therefore, $d = (\delta_1, \ldots, \delta_k)$ wherein δ_k is the distance component solely calculated for the k^{th} objective along the Pareto optimal path.

Consider each edge in Fig. 1 has three objectives, O_1, O_2 and O_3, with no direct correlation among them. In such a scenario, the Pareto optimal label for vertex u_6 is expressed as $(u_6, \overrightarrow{l}) = \{u_3 : \{(8, 3, 9)\}, u_5 : \{(6, 8, 10), (11, 20, 7)\}$. This indicates that there is one shortest path passing through vertex u_3 and two shortest paths passing through vertex u_5. Each tuple in the label indicates the values corresponding to O_1, O_2 and O_3 respectively along the path.

During the computation of Pareto optimal shortest paths for a vertex u, a candidate path distance $d_i = (\delta_1^i, \ldots, \delta_k^i)$ is considered *dominated* if and only if at least another distance

TABLE I
LIST OF NOTATIONS

Notations	Expression
$G(V, E)$	A directed weighted graph
$W(e)$	Weight vector (corresponding to edge e)
$Inc(u)$	Set of in-neighbors of u
k	Objectives count
ΔE	Batch of changed edges
Del	Subset of ΔE containing only deleted edges
Ins	Subset of ΔE containing only inserted edges
T_i	SOSP tree for the objective i
L	A list of Pareto optimal labels
$(u, \overrightarrow{l})$	Tuple containing vertex u and a vector of all Pareto optimal labels of u
d_j^u	j^{th} Pareto optimal distance through parent u
δ_i	Distance component related to objective i

$d_j = (\delta_1^j, \ldots, \delta_k^j)$ exists fulfilling the following conditions:

$$\delta_x^j < \delta_x^i, \text{ for at least one value of } x \text{ where } 1 \leq x \leq k \quad (1)$$

$$\delta_y^j \leq \delta_y^i, \text{ for all } y \neq x \text{ and } 1 \leq y \leq k \quad (2)$$

If d_j dominates d_i, it is denoted as $d_j \prec d_i$. Any distance that is dominated is subsequently excluded from the set of Pareto optimal distances.

Fig. 1 illustrates the computation of Pareto optimal paths and distance labels in an example network, where u_1 is the source vertex and each edge is associated with three weights corresponding to different objectives, O_1, O_2 and O_3, respectively. The direct edges from the source vertex to u_2 and u_3 yield distance labels $\{u_1 : (2, 1, 5)\}$ and $\{u_1 : (4, 1, 1)\}$, respectively. u_2 can also be reached via both u_3 and u_5. However, u_5 currently has $\{null : (\infty, \infty, \infty)\}$ due to initialization, not shown in Fig. 1. Therefore, the resulting distance labels of u_2 are $\{u_1 : (2, 1, 5)\}$, $\{u_3 : (14, 16, 3)\}$ and $\{u_5 : (\infty, \infty, \infty)\}$. Only distance label $\{u_5 : (\infty, \infty, \infty)\}$ of u_2 is dominated by the other labels of u_2 and therefore discarded. $\{u_2 : (4, 5, 7)\}$, $\{u_2 : (16, 20, 5)\}$ and $\{u_3 : (9, 17, 4)\}$ distance labels are obtained for u_4 where $\{u_2 : (16, 20, 5)\}$ is dominated by $\{u_3 : (9, 17, 4)\}$, again discarded. Similarly, $\{u_4 : (5, 6, 8)\}$ and $\{u_4 : (10, 18, 5)\}$ distance labels are obtained for u_5, however, we keep both of the labels this time due to the absence of any dominating vertex. Although u_5 attempts to update the distance of u_2 with new labels $\{u_5 : (9, 9, 10)\}$ and $\{u_2 : (14, 21, 7)\}$, as we assume all the edge weights are non-negative, both of the labels must be dominated by at least one of the existing labels of u_2, that is $\{u_2 : (2, 1, 5)\}$. We treat rest of the vertices, v_6 and v_7, similarly. There exist four optimal paths to reach destination u_7 from source u_1 and edges participating in optimal paths are colored as blue in Fig. 1. Table I lists the notations used in this article.

B. Dynamic Networks

The topology of a dynamic network changes over time and these changes can be in the form of addition or deletion of nodes, edges, or changes in attributes associated with nodes or edges. Interestingly, all variations of the change can be mapped into either edge insertion or deletion. For example, vertex deletion (or insertion) can be mapped into edge deletion (or, respectively, edge insertion) by removing (or, respectively, adding) all the incident and outgoing edges to/from that specific vertex. Thus, edge insertion and deletion can be considered as the generalization of all variations of changes. Depending on the time-varying characteristics, a dynamic network can be of different types. An *incremental network* pertains to a dynamic network where the change is restricted to the addition of edges. Conversely, a *decremental network* refers to a network where changes are exclusively deletions of edges. In a *fully-dynamic* network, both additions and deletions of edges are allowed. In this paper, we consider shortest-path problems in fully dynamic networks and present solutions capable of handling any mixture of edge addition and deletion.

III. PROPOSED APPROACH

In this Section, we first devise a parallel algorithm for efficiently updating SOSP in a fully dynamic network. Subsequently, we utilize this SOSP-update algorithm as a foundation to create a novel heuristic for updating MOSP.

Let $G_t(V_t, E_t)$ be the state of a fully-dynamic directed network at a discrete time step t, where $(u, \overrightarrow{l})_t \in L_t$ signifies the Pareto optimal distance labels (equivalently, the shortest distances in the context of SOSP) for a vertex u. Let Ins_t and Del_t represent the sets of inserted and deleted edges, respectively, so that $E_{t+1} = (E_t \setminus Del_t) \cup Ins_t$. We assume G_t is connected and, after deleting Del_t edges, the updated graph G_{t+1} stays connected. Our goal is to efficiently calculate distance labels, $(u, \overrightarrow{l})_{t+1} \in L_{t+1}$, for all vertices $u \in V_{t+1}$, avoiding a complete recomputation from scratch. For the sake of simplicity, we opt to ignore the subscript t in our algorithm.

A. Single-Objective Shortest Path (SOSP) Update

Our SOSP-update algorithm uses the pre-calculated SOSP tree $T = \{((u, \delta), Parent[u]) : u \in V\}$, and the set of changed edges Ins and Del to update the distance labels. Here, (u, δ) and Parent$[u]$ respectively store the distance and the parent of vertex u in the SOSP tree.

If a set of changed edges $\{(u_1, v), \ldots, (u_x, v)\}$ with the same destination endpoint v are processed asynchronously, all of them may affect the distance of vertex v leading to a race condition. To avoid such erroneous updates, we employ a grouping technique in Step 0 for independent operations among asynchronous threads.

Preprocessing (Step 0): Here, all inserted edges (similarly deleted edges) (u, v) are grouped by the destination endpoint v and stored in $I_{Ins}[v]$ (respectively $I_{Del}[v]$ for deletion). This step mitigates race conditions by grouping the dependent operations together and splitting the independent operations among the asynchronous threads.

Process Changed Edges (Step 1): In this step, each group of changed edges is evaluated by a single thread. For an *insertion* of a new edge (u, v) that reduces the distance of vertex v, the distance is updated to $(v, \delta) \leftarrow (u, \delta) + W((u, v))$, and v is flagged as affected (refer to Algorithm 1, Lines 10–13). The *deletion* of an edge (u, v) not in the current SOSP tree has no impact on distance labels. However, removing an edge $(u, v) \in T$, disconnects vertex v from the source vertex in T. Therefore the algorithm identifies vertex v as affected and

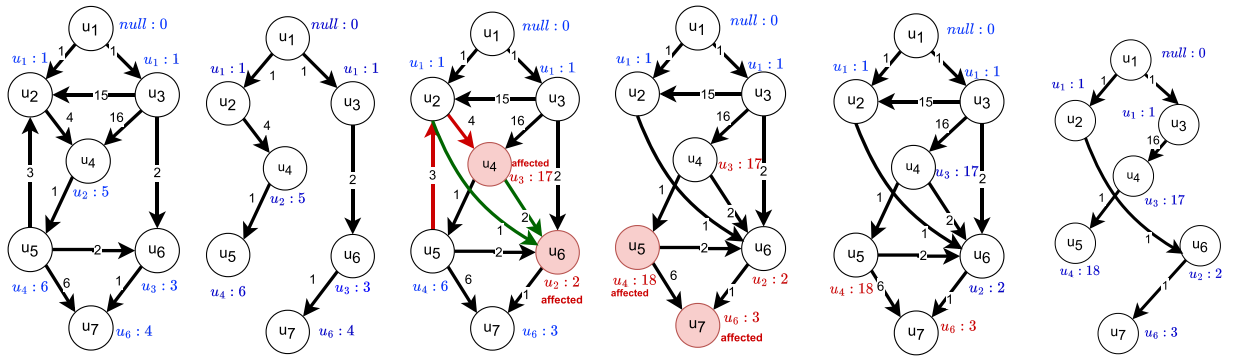


Fig. 2. SOSP update.

seeks an alternate route to reconnect it with the SOSP tree. Since the SOSP tree is represented through a parent-child relationship between consecutive vertices on the shortest path, searching an alternative path for v involves finding a new parent vertex in T for v. A potential parent, p_c, is any vertex within v's in-neighbor set, $Inc(v)$, excluding vertex u. The algorithm appoints a vertex P from $\{Inc(v) \setminus u\}$ as v's new parent if the path via P yields the least distance (v, δ), updating the distance to $(P, \delta) + W((P, v))$. If no new parent is found, a $Null$ value is assigned to the parent and the distance becomes ∞.

Propagate the Update (Step 2): In Step 1, vertices affected by changes can alter the distances of their neighboring vertices. Therefore, Step 2 identifies the *candidate* neighbor vertices requiring distance recalculations and proceeds to update these distances. Assigning a single thread to each affected vertex $v \in Aff$ for both identifying and updating distances of neighbors can lead to inefficiencies when a neighboring vertex u_n, shared by multiple affected vertices, is processed by several threads. This approach not only increases redundant computations but also elevates the risk of race conditions as multiple threads attempt to update the same vertex concurrently. To avoid such situations, we divide the process into below two sub-steps:

i) Identifying the candidates: The algorithm identifies the neighboring vertices of each affected vertex $v \in Aff$ and adds them into a set $\mathcal{N}$ as candidates. Concurrently, it removes v from Aff (Algorithm 1 Line 23 to 25).

ii) Updating distances: The distance for each candidate vertex in $\mathcal{N}$ is evaluated and updated as necessary, with a dedicated thread assigned to each vertex. For every vertex v in $\mathcal{N}$, the algorithm examines each possible parent P within $Inc(v)$, selecting the one that minimizes the new distance $Dist_{new} \leftarrow (P + \delta) + W((P, v))$. If $Dist_{new}$ differs from the current distance label (v, δ), both the distance and the parent of v are updated accordingly. If the distance is updated, v is subsequently added to the list of affected vertices (Algorithm 1 Line 26 to 32).

Given that vertices with newly updated distances in Step 2 might influence another group of neighboring vertices, Step 2 is designed as an iterative procedure that converges when no further vertices require distance updates.

Fig. 2 illustrates the working principle of Algorithm 1. Fig. 2(a) depicts a network similar to Fig. 1, considering only the second objective, O_2. The initial SOSP tree of the graph of Fig. 2(a) has shown in Fig. 2(b). Fig. 2(c) shows the affected vertices due to changed edges $Del = \{(u_2, u_4), (u_5, u_2)\}, Ins = \{(u_2, u_6) : W = (10, 3, 12), (u_1, u_6) : W = (12, 2, 14)\}$ for second objective only. Here, the deletion shown in color red, otherwise insertion, shown in color green in Fig. 2(c). The vertex has been marked affected according to the Algorithm 1.step 1 as shown in Fig. 2(c). The Algorithm 1.step 2 has been demonstrated in Fig. 2(d)–(e) for propagating the effects due to the affected nodes marked in Algorithm 1.step 1. Fig. 2(f) is the updated SOSP tree for the second objective, O_2, after applying the changes.

A Special Case: Fig. 3 illustrates a special case where a descendant of an affected vertex from the previous shortest path becomes its new parent, creating a loop. Suppose the edge between u_1 and u_2 is deleted, as shown in Fig. 3(a). Due to this deletion, Algorithm 1 marks u_2 as affected and identifies u_5, a descendant of u_2, as the new parent, changing the distance of u_2 to 9. During the first iteration of Step 2in Algorithm 1, u_4 is identified as a neighboring candidate requiring a distance update and is added to $\mathcal{N}$. Subsequently, the distance of u_4 is updated with u_2 remaining its parent in the shortest path (Fig. 3(b)). In the second iteration, u_5, the only out-neighbor of u_4, is added to $\mathcal{N}$ and its distance is updated (Fig. 3(c)). In the next iteration, the out-neighbors u_2, u_6, u_7 of vertex u_5 are identified as candidates and the distance of u_2 is updated. Notably, the distance of u_5 was initially affected by the update propagation from u_2. By the third iteration, this propagation reaches u_2 again, creating a loop. The update propagation in the loop continues until a vertex in the loop finds a new parent vertex outside the loop. In our example, the loop breaks when a non-descendent vertex is selected as the parent of u_3 in iteration 4 as shown in Fig. 3(e). The absence of any such non-descendent parent can create an infinite loop that occurs only when there is a disconnected component of the graph due to edge deletion. If the loop iterates more than $\mathcal{D}/\omega$ times, the infinite loop is detected. Here, $\omega = \sum W(e) : e \in E_{loop}$ where E_{loop} is the edge set participating in the loop and $\mathcal{D}$ is the diameter of the graph after the topological change.

In practice, determining ω in advance is challenging. However, by tracking the distance increase for each vertex at each iteration, we can detect an infinite loop if a consistent distance increase (denoted ω') persists for a set of vertices for at most $\mathcal{D}/\omega'$ iterations. Upon detection, a conditional statement can halt updates for vertices in the disconnected subgraph, while allowing updates to continue for the subgraph connected to

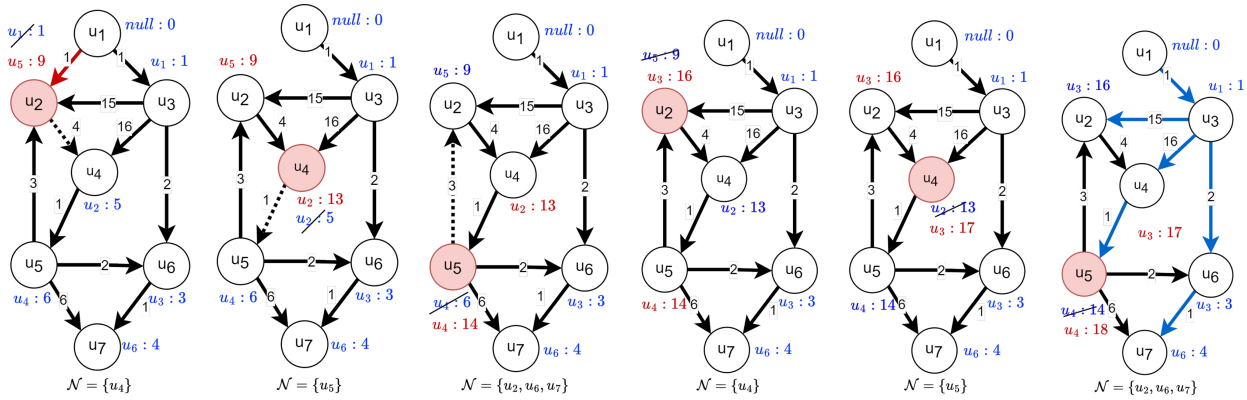


Fig. 3. Special case: Loop formation.

the source vertex. Finding a more efficient way to handle the disconnected graphs requires further research. In this paper, we assume the graph remains connected despite topological changes.

B. Multi-Objective Shortest Path Update

The Pareto optimal solution for MOSP search typically yields *all* optimal shortest paths where no path is inferior to the others. However, in practical applications, finding a *single* and quick solution for MOSP problem is more viable rather than exhaustively determining all possible paths. Moreover, finding a single MOSP enhances execution efficiency and reduces the demand for computational resources. Under these considerations, we introduce a novel heuristic algorithm to determine a single MOSP instead of all possible optimal solutions in a fully dynamic network. We divide the MOSP search problem (k-objectives) into k SOSP-update problems in dynamic networks and strategically combine them to compute a single MOSP efficiently.

Our MOSP-update algorithm takes the original network $G(V, E)$, initial SOSP trees T_i for each objective $1 \leq i \leq k$, sets of inserted (Ins) and deleted (Del) edges, along with preference vector $Pref$ where each item correspond to each objective function as input. $1 \leq Pref_i \leq k$, for ith item, is considered to avoid negative edge weight according to Lemma 1. Lower the value, higher the preference is considered and all the values of $Pref$ vector is set 1 as default value when no preference is given. It outputs a single optimal or suboptimal MOSP solution for the updated network.

Step 1. Update SOSP trees: Step 1 of Algorithm 2 updates the SOSP trees T_i for each objective i, using Algorithm 1. For updating T_i, the algorithm only assesses the i^{th} term in the edge weight vector $W(e)$ for all edges in E and all the changed edges in Ins and Del. At this stage, the SOSP trees are updated one after another, with each update employing parallel processing within the SOSP-update procedure.

Step 2. Create a combined graph: Upon completing Step 1, each updated SOSP tree T_i offers paths with the shortest distance δ_i for all vertices, focusing on a single objective. However, real-world applications frequently necessitate MOSP solutions that integrate or balance multiple objectives, accommodating various user-defined priorities for these objective functions. Therefore, our algorithm aggregates the partial solutions, i.e., the updated SOSP trees T_i, into an ensemble graph $\mathbb{G}(V, \mathbb{E})$. This graph encompasses all edges from each T_i, such that $\mathbb{E} = \bigcup_{i=1}^{k} e \in T_i$. Our algorithm then searches for the shortest path within $\mathbb{G}$ to determine the final MOSP solution. Since k is low in our case, the workload for each thread remains similar. The overhead related to Step 2 can increase if the number of objectives is large.

The user-defined priorities for different objectives are incorporated into the ensemble graph $\mathbb{G}$ through newly assigned non-negative edge weights. Let the preference level for the i^{th} objective be denoted by an integer $1 \leq Pref_i \leq k$ with a lower $Pref_i$ indicating higher priority. Given the goal of identifying the shortest path, weights assigned to edges $e \in \mathbb{E}$ should be proportional to $Pref_i$ (or inversely proportional to $-1/Pref_i$), reflecting their desirability for selection in the final MOSP solution. Specifically, for edges $e \in \mathbb{E}$ present in multiple SOSP trees, their weights $\mathbb{W}(e)$ must aggregate the preferences from all relevant objectives. These can be calculated as $\mathbb{W}(e) = k + 1 + \sum_{i=1}^{k} -\frac{1}{Pref_i} \cdot \chi(e \in T_i)$, where $\chi(e \in T_i)$ is an indicator function that equals 1 if e belongs to the specific SOSP tree, and 0 otherwise. This weight assignment ensures that each edge $e \in \mathbb{E}$ receives a non-negative weight, accurately reflecting the combined preferences. The procedure for determining these weights is outlined from line 5 to line 9 in Algorithm 2.

Lemma 1: The weight of any edge $e \in \mathbb{E}$ should be positive.

Proof: Given that $\mathbb{W}(e) = k + 1 - \sum_{i=1}^{k} \frac{1}{Pref_i} \cdot \chi(e \in T_i)$, where $Pref_i$ are non-negative, the weight of an edge e decreases as the sum $\sum_{i=1}^{k} \frac{1}{Pref_i} \cdot \chi(e \in T_i)$ increases, reaching a minimum edge weight when this sum is maximized. For every i satisfying $1 \leq i \leq k$, the sum attains its maximum when $Pref_i = 1$, which is the lowest feasible value for $Pref_i$, and concurrently, when the edge e is present in all updated SOSP trees. Consequently, the maximum sum value is $\sum_{i=1}^{k} 1 \cdot 1 = k$, resulting in the minimum possible edge weight of $\mathbb{W}(e) = k + 1 - k = 1$, which ensures that the edge weight remains positive. □

Theorem 2: Let a vertex v be reachable in h hops using two disjoint paths $\mathcal{P}_1$ and $\mathcal{P}_2$ in the combined graph $\mathbb{G}$, where the edges of $\mathcal{P}_1$ appear in $l_1 < k$ SOSP trees and the edges of $\mathcal{P}_2$ appear in the remaining $l_2 = k - l_1$ SOSP trees. Let $\mathcal{P}_2$ contain the edges from the SOSP tree related to a specific objective i', and we want to prioritize this objective to select $\mathcal{P}_2$ in the

Algorithm 1: SOSP_Update($G(V, E), T, Ins, Del, Dist$).

```
/* Step 0: Preprocessing */
Initialize two arrays I_ins and I_del of size |V| where each element is an empty list.
for each directed edge (u, v) ∈ Ins do
    Add (u, v) to I_Ins[v]
for each directed edge (u, v) ∈ Del do
    Add (u, v) to I_Del[v]
/* Step 1: Process Changed Edges */
Initialize an empty list Aff
for each vertex v ∈ V in parallel do
    for each edge (u, v) ∈ I_Ins[v] do
        E ← E ∪ (u, v)
        if (v, δ) > (u, δ) + W((u, v)) then
            Parent[v] ← u
            (v, δ) ← (u, δ) + W((u, v))
            Aff ← Aff ∪ v
    for each edge (u, v) ∈ I_Del[v] do
        E ← E \ (u, v)
        if (u, v) ∈ T then
            P ← arg min_{p_c ∈ Inc(v) & p_c ≠ u}((p_c, δ) + W(p_c, v))
            Parent[v] ← P
            (v, δ) ← (P = Null)?∞ : (P, δ) + W(P, v)
            Aff ← Aff ∪ v
/* Step 2: Propagate the update */
while Aff is not empty do
    Initialize empty vectors N
    for each v ∈ Aff in parallel do
        Aff ← Aff \ v
        Add the neighbors of v in N
    for each v ∈ N in parallel do
        P ← arg min_{p_c ∈ Inc(v)}((p_c, δ) + W(p_c, v))
        Dist_new ← (P = Null)?∞ : (P, δ) + W(P, v)
        if Dist_new ≠ (v, δ) then
            (v, δ) ← Dist_new
            Parent[v] ← P
            Aff ← Aff ∪ v
```

Algorithm 2: MOSP_Update($G(V, E), \{T_1, \ldots, T_k\}, Ins, Del, Pref.$)

```
/* Step 1: Find updated SOSP tree T_i */
for i = 1 to k do
    SOSP_Update(G(V, E), T_i, Ins, Del)
/* Step 2: Create a combined graph */
𝔼 = ∪_{i=1}^{k}(e ∈ T_i)
Create a graph 𝔾 with vertex set V and edge set 𝔼
for each e ∈ 𝔼 in parallel do
    𝕎(e) = k + 1
    for i = 1 to k do
        if e ∈ T_i then
            𝕎(e) = 𝕎(e) − 1/Pref_i
/* Step 3: Find SOSP in combined graph */
Find SOSP in 𝔾
Assign actual edge weights from updated G on the output SOSP tree to find the MOSP
```

final MOSP solution. An ideal choice to achieve this would be $Pref_{i'} = x < \frac{y}{l_1 - l_2 + 1}$, where $Pref_j = y$ for all j, $1 \leq j \leq k$, $i' \neq j$, $l_1 \geq l_2$, and $y > l_1 - l_2 + 1$.

Proof: According to our proposed formula, the weight of each edge on the path $\mathcal{P}_1$ is given by $\mathbb{W}(e \in \mathcal{P}1) = k + 1 - \sum i = 1^{l_1} \frac{1}{y} = k + 1 - \frac{l_1}{y}$. Similarly, the weight of each edge on $\mathcal{P}_2$ can be computed as $\mathbb{W}(e \in \mathcal{P}2) = k + 1 - \sum i = 1^{l_2 - 1} \frac{1}{y} - \frac{1}{x} = k + 1 - \frac{l_2 - 1}{y} - \frac{1}{x}$.

Since both paths require h hops to reach vertex v, the total distance to v along $\mathcal{P}_1$ is $h(k + 1 - \frac{l_1}{y})$, and along $\mathcal{P}_2$ is $h(k + 1 - \frac{l_2 - 1}{y} - \frac{1}{x})$. For $\mathcal{P}_2$ to be selected in the final MOSP solution, the distance along this path should be the shortest to reach v. Therefore the following condition must be satisfied in the combined graph:

$$h\left(k + 1 - \frac{l_2 - 1}{y} - \frac{1}{x}\right) < h\left(k + 1 - \frac{l_1}{y}\right)$$

$$Or, -\frac{l_2 - 1}{y} - \frac{1}{x} < -\frac{l_1}{y}$$

$$Or, x < \frac{y}{l_1 - l_2 + 1}$$

□

In Theorem 2, the condition $l_1 \geq l_2$ is necessary to ensure that the preference value x remains positive. Additionally, the condition $y > l_1 - l_2 + 1$ is required to prevent x from taking fractional values.

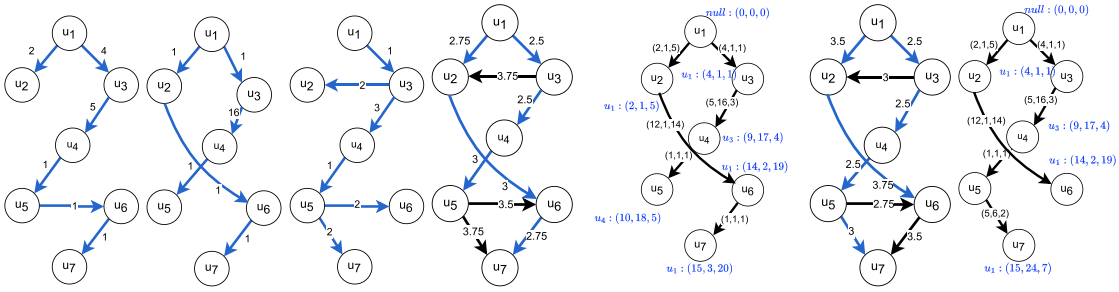
Corollary 1: When both paths contain edges that appear in the same number of SOSP trees, i.e., $l_1 = l_2 = \frac{k}{2}$, the preference selection should satisfy the condition $x < y$.

Developing a generic approach that includes cases with different hop counts per path for selecting a specific Pareto optimal path requires further research.

Step 3. Find SOSP in the combined graph: The combined graph $\mathbb{G}(V, \mathbb{E})$ includes edges $e \in \mathbb{E}$ present in at least one SOSP tree, with new weights $\mathbb{W}(e)$ allocated according to the priorities of the objectives. In Step 3, a parallel single-source shortest path algorithm determines an SOSP in $\mathbb{G}$. Reassigning the original weights from $G(V, E)$ to the edges of the identified SOSP allows the algorithm to produce an optimal or sub-optimal MOSP solution.

Lemma 3: In the graph $G(V, E)$, where edge weights correspond to multiple objectives, if T_j is the unique SOSP tree associated with objective $j (1 \leq j \leq k)$, then any path within this tree constitutes a subpath of the Pareto optimal MOSP solutions.

Proof: Given a vertex v with distances $d(v) = (\delta_1, \ldots, \delta_j, \ldots, \delta_k)$ along the SOSP tree T_j and $d_x(v) = (\delta_1^x, \ldots, \delta_j^x, \ldots, \delta_k^x)$ along an alternate path. Assuming $d(v)$ is not Pareto-optimal and $d_x(v)$ dominates $d(v)$, that is, $d_x(v) \prec d(v)$, it follows that $\delta_j^x \leq \delta_j$. However, it is impossible as δ_j is the shortest distance component of v along the only SOSP related to objective j. Therefore, the assumption that $d(v)$ is not Pareto-optimal is false, establishing $d(v)$ as the Pareto optimal distance. Consequently, any path within T_j is part of the Pareto optimal MOSP solutions. □



(a) SSSP (Obj. 1) (b) SSSP (Obj. 2) (c) SSSP (Obj. 3) (d) $Pref = \{4, 1, 4\}$ (e) MOSP for 4d (f) $Pref = \{4, 4, 1\}$ (g) MOSP for 4f

Fig. 4. Finding a single MOSP for $Del = \{(u_2, u_4), (u_5, u_2)\}, Ins = \{(u_4, u_6) : W = (10, 2, 12), (u_2, u_6) : W = (12, 1, 14)\}$.

Theorem 4: Let T_i be the *only possible* SOSP tree for objective i in the graph $G(V, E)$. A combined graph $\mathbb{G}(V, \mathbb{E})$ is constructed by merging all edges from the SOSP trees T_i for all $i = 1, \ldots, k$, with edge weights modified to reflect user preferences (as outlined in Algorithm 2). If T' is identified as an SOSP within $\mathbb{G}$, then restoring the original edge weights from E to the edges of T' results in a Pareto optimal shortest path solution for G.

Proof: For a single objective, the SOSP inherently is the Pareto optimal shortest path. With multiple objectives, assuming a Pareto optimal path from the source to u via T' (where u may be the source), and given an edge $(u, v) \in T'$, we aim to show the path from u to v is also part of Pareto optimal shortest path solution.

Given that T' is an SOSP within $\mathbb{G}(V, \mathbb{E})$, any edge $(u, v) \in T'$ implies $(u, v) \in \mathbb{E}$. Since every edge $e \in \mathbb{E}$ is part of at least one SOSP tree, it follows that the edge (u, v) is included in an SOSP tree T_i associated with objective i, where $1 \leq i \leq k$. As there exists only one SOSP tree related to each objective, invoking Lemma 3 allows us to deduce that the direct path to v via u is Pareto optimal. □

Fig. 4 is the depiction of the working principle of the MOSP algorithm for the changed edges of $Del = \{(u_2, u_4), (u_5, u_2)\}$, $Ins = \{(u_4, u_6) : W = (10, 2, 12), (u_2, u_6) : W = (12, 1, 14)\}$. Fig. 4(a), Fig. 4(b) and (c) are the SOSP trees for objective 1, 2 and 3 respectively upon applying the update of Algorithm 1 on each of the objective individually. Fig. 4(d) is the combined graph generated using the Algorithm 2 for the preference vector $Pref$ of $\{4, 1, 4\}$. Here we bias $Pref$ vector towards objective 2 setting the lower value of 1 compared to 4 for objectives 1 and 3. The SSSP tree of the combined graph of Fig. 4(d) is shown in Fig. 4(e) which illustrates the shortest path and distance from source u_1 to destination u_7. Similarly, the Fig. 4(f) and (g) are the combined graph and corresponding SSSP tree, respectively, for the $Pref$ of $\{4, 4, 1\}$ to bias toward objective 3. There exist 4 Pareto optimal solutions after the changed edges to reach u_7 from u_1, $u_5 : (15, 24, 7), u_6 : (12, 21, 8)$, $u_6 : (9, 4, 10)$ and $u_6 : (15, 3, 20)$. With the $Pref$ vector of $\{4, 1, 4\}$, we give more priority to objective 2 to select the solution $u_6 : (15, 3, 20)$ which minimizes the cost for objective 2. Similarly, the $Pref$ vector of $\{4, 4, 1\}$ prioritize objective 3 to select solution of $u_5 : (15, 24, 7)$ which also minimizes cost of the objective 3. The other Pareto optimal solutions, i.e., $u_6 : (12, 21, 8)$ and $u_6 : (9, 4, 10)$ can also be selected for different preference vectors to get an alternative optimal shortest path from source u_1 to destination u_7.

Complexity Analysis: In Step 1 of Algorithm 1, processing each changed edge in parallel yields a time complexity of $O(\frac{|Ins|+|Del|}{p})$, with p denoting the processor count. Step 2 involves checking neighbors of affected vertices, linking workload to affected vertices' degrees. With x vertices affected in a Step 2 iteration and deg_{avg} as the average vertex degree, each iteration's time complexity is $O(\frac{x \cdot deg_{avg}}{p})$. The maximum iterations needed are $O(\mathcal{D})$, with $\mathcal{D}$ representing the diameter of the graph. Therefore, the total time complexity of Step 2 is $O(\frac{\mathcal{D} \cdot x \cdot deg_{avg}}{p} + \mathcal{D})$, where the last term, $\mathcal{D}$, accounts for the minimum constant time required for each iteration. Consequently, the overall time for updating a SOSP becomes $\Lambda = O(\frac{|Ins|+|Del|}{p} + \frac{\mathcal{D} \cdot x \cdot deg_{avg}}{p} + \mathcal{D})$.

Our algorithm, DynaMOSP, updates SOSP for k objectives in $k \cdot \Lambda$ time. In Step 2 of Algorithm 2, it merges k SOSP trees, each with $|V| - 1$ edges, within $O(\frac{k \cdot (|V|-1)}{p})$ time. The final step, employing a parallel shortest path on the combined graph $\mathbb{G}(V, \mathbb{E})$, with $|\mathbb{E}| = O(k \cdot |V|)$, completes in $k \cdot \frac{|V|^2}{p}$ time. Consequently, DynaMOSP's overall time complexity is $O(k \cdot (\frac{|Ins|+|Del|}{p} + \frac{\mathcal{D} \cdot x \cdot deg_{avg}}{p} + \mathcal{D} + \frac{|V|-1}{p} + \frac{|V|^2}{p})) = O(k \cdot (\frac{|Ins|+|Del|}{p} + \frac{\mathcal{D} \cdot x \cdot deg_{avg}}{p} + \mathcal{D} + \frac{|V|^2}{p}))$.

Discussion: In real-world applications, one objective may depend on another, and our MOSP-update algorithm performs effectively regardless of these dependencies. However, when objectives are positively correlated and an equality can express their dependencies, they can be combined into a single objective. This approach can be incorporated into the preprocessing step to reduce the execution time by lowering the effective number of objectives.

In Algorithm 2 Step 3, finding MOSP from the combined graph can be treated as a parallel single-objective shortest path problem. Moreover, this step could be further optimized by treating it as a SOSP *update* step, where the shortest path on the combined graph from the previous time instance t can be updated based on new edges in the combined tree at the current time instance $(t + 1)$.

IV. Implementation Details

We have developed an implementation using SYCL to enable support across heterogeneous computing platforms, including CPUs and GPUs from various manufacturers. Additionally,

we've created a separate implementation utilizing OpenMP, specifically targeting shared-memory CPUs.

In both implementations, to store the input graph G, we utilize two adjacency lists in the Compressed Sparse Row (CSR) format for in-edges and out-edges. This approach quickly identifies the parent candidates and affected neighbor vertices in the later stages. The changed edges are grouped according to their incident vertices and stored in the CSR format for easy retrieval. SOSP trees are represented through parent-child relation, employing two vectors each of size $|V|$: one tracks each vertex's parent, and the other records the shortest distance for each vertex.

Dynamic graph data structures like LLAMA [18], GraphOne [19], and TeGraph+[20] outperform CSR for maintaining dynamic graph structures by reducing the overhead of readjusting adjacency lists and indices. However, these CPU-based structures rely on dynamic memory allocation, making them inefficient for GPU-based implementations. While direct computation on compressed graphs [21] improves space efficiency, further research is needed for dynamic operations on such data. As our primary focus is efficient MOSP updates, we use the simpler CSR and leave advanced data structures for dynaMOSP as future work.

For OpenMP based shared memory implementation, we employ the *'#pragma omp parallel for'* directive across all parallel constructs. In SYCL-based implementation, kernels are submitted as tasks to the SYCL queue for parallel execution. For dependencies, the SYCL constructs a directed acyclic graph (DAG) to manage task execution. In each kernel, we rely on SYCL *'parallel_for'* to distribute the total work among work groups. For each parallel construct in our algorithm, we define the global range for work items, divide them into workgroups by SYCL, and assign them to compute units.

In Algorithm 1, Step 0, inserted and deleted edges are processed as independent tasks without synchronization, as they operate on separate arrays and group changed edges by incident vertices. In Step 1, affected vertices are marked, and distances are updated using parallel threads. A bit array of size $|V|$ marks affected vertices, with threads setting bits independently. This approach can lead to load imbalance when edge changes are unevenly distributed. Solving this issue requires a sizewise sorting of the changed edge groups and workgroup distribution according to the size. Later a parallel filter kernel compacts vertex IDs into an array Aff.

In Algorithm 1 Step 2, the update propagates by selecting the neighbors of the currently affected vertices in Aff and updating their distances as needed. We implement it using a three-stage process like below.

1. *Forward traversal:* Parallel threads mark out-neighbors of affected vertices in Aff using a bit array.
2. *Filter:* Identifies unique neighbor vertices $\mathcal{N}$ using the bit array. It reduces redundant work in a later stage.
3. *Backward traversal:* Threads independently find potential parents of each $u \in \mathcal{N}$ using the in-edge list and update distances by selecting the best parent vertex.

This process involves using out-edges for forward traversal and in-edges for distance updates, as shown in Fig. 5. For example, if Step 1 marks u_1 and u_3 as affected, two parallel threads mark neighbors in $\mathcal{N}_{unfiltered}$. After filtering, $\mathcal{N} = \{u_4, u_5, u_6\}$ are updated concurrently by three threads assessing their parent vertices.

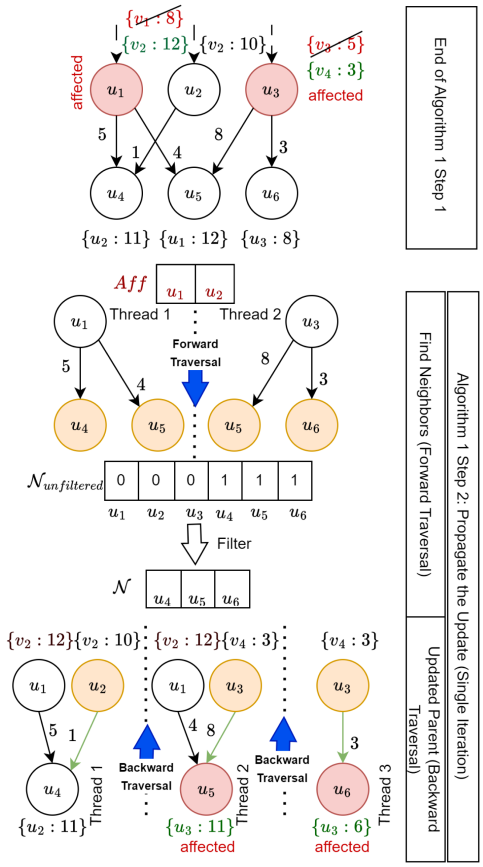


Fig. 5. Update propagation: An iterative method consisting of both forward and backward traversal.

The MOSP update algorithm updates SOSP trees related to each objective, merges them into a combined graph $\mathbb{G}$, and assigns edge weights based on user preferences, using atomic *'fetch_sub'* to prevent race conditions.

Ultimately, it computes SOSP on $\mathbb{G}$ using any parallel SOSP implementation and updates MOSP by reassigning actual edge weights from updated network G. For computing SOSP on $\mathbb{G}$ we use a parallel Bellman-Ford algorithm proposed in [22] to enable concurrent vertex visits.

V. Performance Evaluation

Experimental Setup: The shared memory experiments were performed using dual 32-core AMD EPYC Rome 7452 CPUs with 64 GB of DDR4 RAM, while GPU experiments utilized an NVIDIA A100 GPU with 80 GB memory. The large graphs used in the experiments are detailed in Table II. Originally, the datasets did not contain multiobjective values. Therefore, for the purpose of testing, multiple edge weights were randomly generated for each edge.

A. Experiment on OpenMP Implementation

To evaluate scalability, in our first experiment, we adjusted the OpenMP thread count from 1 to 64 while maintaining a constant total of 50 K changed edges, including both insertions and deletions. We altered the deletion percentage for each batch of changed edges. For instance, a 25% deletion rate in 50 K total

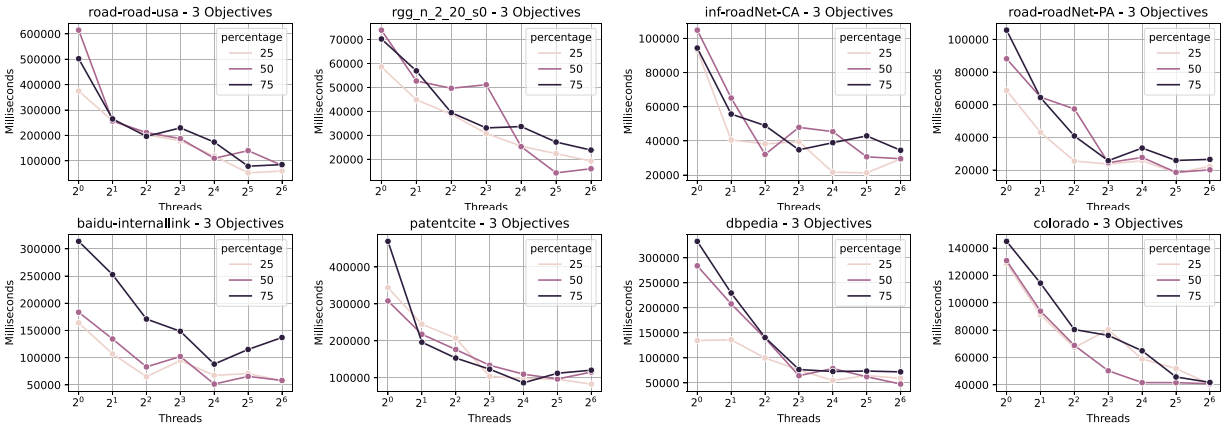


Fig. 6. Scalability analysis using OpenMP implementation (Total changed edges $|Ins| + |Del| = 50K$).

TABLE II
NETWORKS IN OUR EXPERIMENT [23], [24]

Name	Num. of Vertices	Num. of Edges
road-usa	23,947,347	28,854,312
rgg-n-2-20-s0	1,048,576	6,891,620
Baidu-Internal	2,141,300	17,794,839
US patent	3,774,768	16,518,947
roadNet-CA	1,971,281	5,533,214
Colorado	435,666	1,042,400
roadNet-PA	1,090,920	3,083,796
dbpedia	3,966,924	13,820,853
soc-twitter	56M	265M

changes means 12.5 K edge deletions and 37.5 K insertions. Instead of fully random edge changes, we generate inserted edges by picking two random vertices and setting the edge weight below the graph's average. This approach raises the likelihood of inserted edges impacting the existing SOSP tree. For deletions, an edge is chosen only if it is part of the SOSP tree, ensuring its removal affects the updated SOSP tree. Such targeted changes result in a higher workload compared to completely random alterations. DynaMOSP algorithm was tested with 3 objectives across 8 different networks. The scalability results of this experiment are shown in Fig. 6, where the X-axis represents the thread count on a log_2 scale, and the Y-axis indicates the execution time in milliseconds. Generally, across all graphs, the execution time decreases as the number of threads increases, demonstrating the good scalability of DynaMOSP. Since changed edges for each experiment are generated independently, with endpoints chosen randomly, the same quantity of changed edges can result in different total workloads depending on their location within the graph. Consequently, we observe that the curves are not smooth and include some peaks, indicating instances of higher effective workload.

In the next experiment, we employed a similar setup but fixed the deletion percentage at 50% and varied the total number of changed edges among $25K$, $50K$, and $100K$. Fig. 7 illustrates the findings of this experiment. It demonstrates that as the number of changed edges increases, the execution time also increases. However, the difference in execution times across batches of different sizes of changed edges diminishes as the number of threads increases. We observe that the execution times for batches with $25K$ and $50K$ changed edges converge in some cases when the thread count reaches 32.

B. Experiment on SYCL Implementation

Our SYCL implementation is compatible with various computing architectures, including both CPUs and GPUs. We primarily tested it on the Nvidia GPU platform using Intel LLVM SYCL and the *Clang++* compiler. For a comparative analysis, we utilized the same implementation on CPUs with Intel OneAPI SYCL and the *icpx* compiler, with detailed findings presented in Section V-C.

Fig. 8 illustrates execution times for networks with 50 K changed edges, varying deletions at $25\%, 50\%$, and 75%. It covers bi-objective and tri-objective scenarios. Tri-objective updates, needing an extra SOSP operation compared to bi-objective cases, result in longer execution times. Although the total execution time depends on multiple factors, including the location of changes in the network, a higher percentage of edge deletion generally results in a greater workload. Because, according to DynaMOSP design, processing a deletion is more complex than processing an insertion. As a result, we see an increase in execution time in when the percentage of deletion is increased.

Fig. 9 displays the execution time distribution across different steps of the experiment. Step 1, which updates the SOSP trees in parallel and performs the most computationally intensive tasks, accounts for the majority of the execution time. Step 2, involving the creation of the ensemble graph and edge weight computation using SYCL atomic operations, is the least time-consuming. Step 3 calculates SOSP on the ensemble graph, which consists of only up to $k * (|V| - 1)$ edges, with $k = 2$ for this experiment. The parallel SOSP implementation requires approximately 20% of the total execution time to complete.

In Figs. 6 and 7, we observe that execution time does not always decrease as the thread increases. The execution time in our approach depends on the location of the change in the graph. A change near the root of an SOSP tree impacts a larger subgraph compared to a change near a leaf node. To experiment we have taken the unweighted version of the graph and found the diameter of it.We then divide the vertices into two sets: *low*,

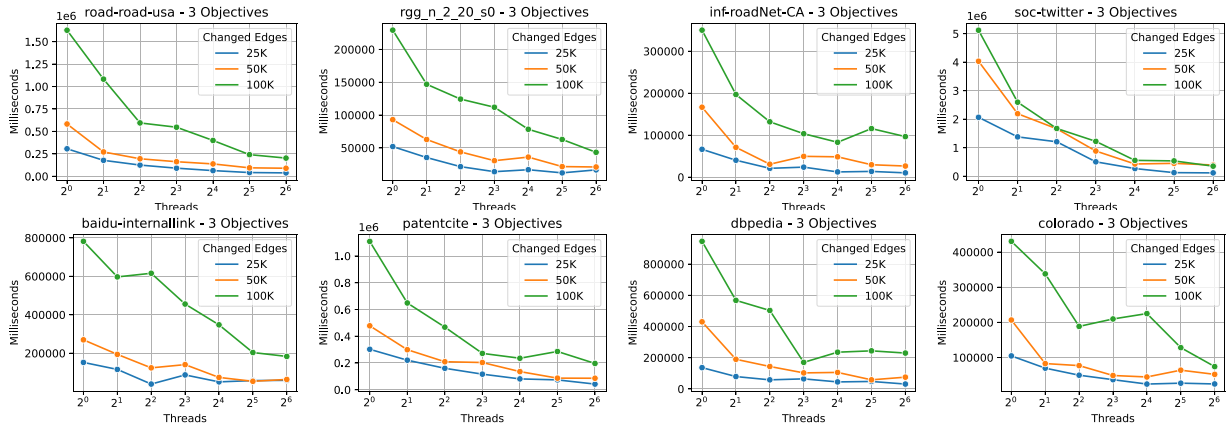
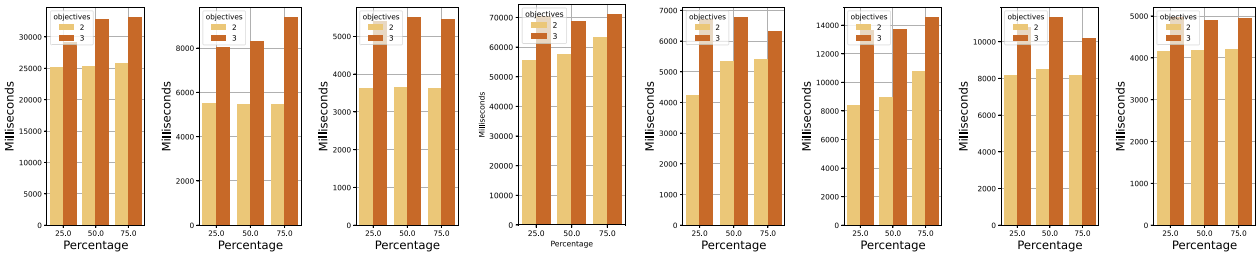


Fig. 7. Scalability analysis using OpenMP implementation (Varying changed edge batch size).



(a) road-road-usa (b) rgg_n_2_20_s0 (c) roadNet-CA (d) soc-twitter (e) baidu (f) patentcite (g) dbpedia (h) colorado

Fig. 8. Execution time for SYCL (GPU) implementation.

Fig. 9. Time distribution across Algorithm 2 steps.

containing vertices with distances greater than or equal to the diameter, and *high*, containing the remaining vertices. Then we generate two separate random changed edges of size $50K$ for considering vertices in set *low* and *high*. Experiment on these sets illustrates that the time requirement is higher when nodes are affected closer to the source (Fig. 10).

C. Comparison Study

To the best of our knowledge, there is no existing work in the literature that considers parallel MOSP search in a dynamic network. In this section, we compare our implementations with paPaSearch [6], the state-of-the-art algorithm for finding MOSP in large *static* networks using shared memory parallelism. Given that paPaSearch cannot process modifications to edges directly, we update the network by incorporating the Ins edges and removing the Del edges from the original edge list. This updated network then serves as the input for paPaSearch, allowing us to accurately measure its execution time. Unlike our approach, which finds the shortest path from a single source to all other vertices, paPaSearch identifies all MOSPs from a source to a specific destination vertex.

To evaluate the performance of our OpenMP-based implementation relative to paPaSearch, we conducted experiments on four large networks: rgg-n-2-20-s0, US patent, dbpedia, and roadNet-CA. Our experimental design involved varying the thread count from 1 to 64 while measuring execution times for finding MOSP in a bi-objective scenario with total changes set at $25K$, $50K$, and $75K$. In each scenario, we maintained a consistent deletion rate of 50%. The results, depicted in Fig. 11, illustrate the performance advantage of our approach. The Y-axis represents the speedup, defined as the execution time ratio of paPaSearch to our DynaMOSP implementation using OpenMP, while the X-axis illustrates the range of thread counts. Across all networks and thread counts, our shared memory DynaMOSP outperformed paPaSearch, achieving a maximum speedup of $90\times$ and an average speedup of $18\times$.

In the subsequent set of experiments, we set the total number of changed edges at $50K$ and the deletion percentage at 50%. Observing that in most cases the optimal shared memory

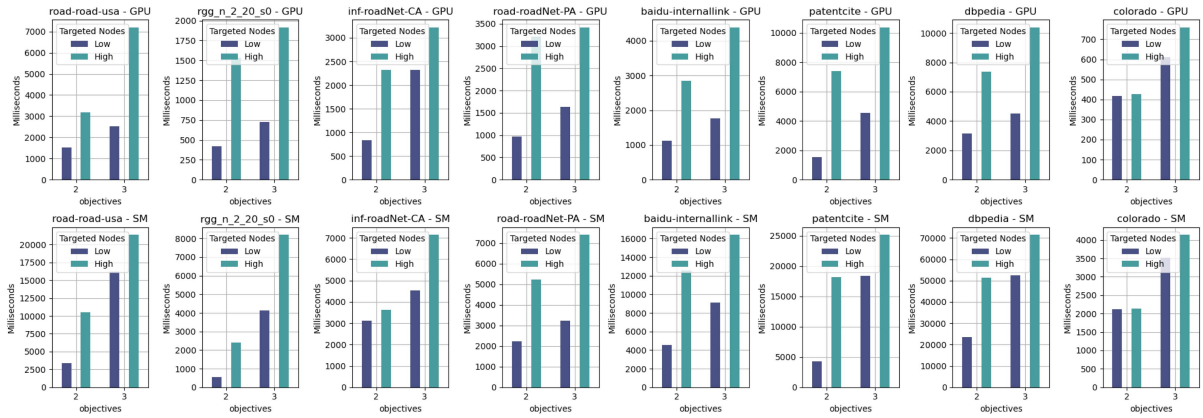
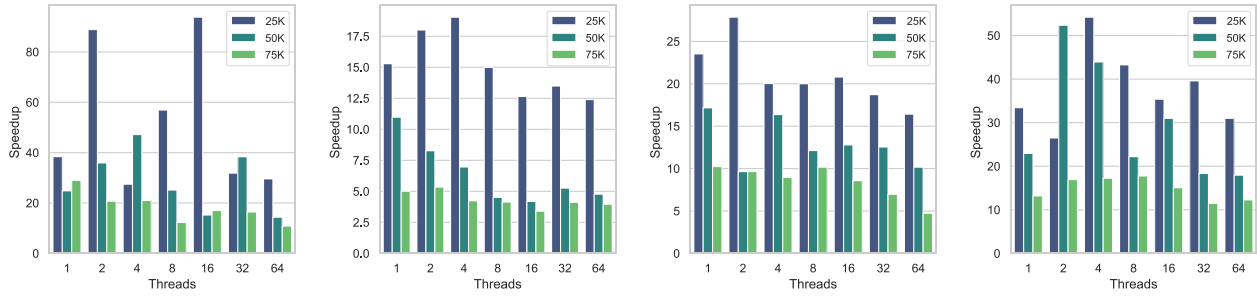


Fig. 10. Time required targeting the node closer (low) and farther (high) from the source node.



(a) Speedup for rgg-n-2-20-s0 (b) Speedup for US patent (c) Speedup for dbpedia (d) Speedup for roadNet-CA

Fig. 11. Speedup comparison between paPaSearch and the proposed method varying the threads for different datasets. The X-axis denotes the thread count and the Y-axis is the speedup achieved for 50% insertion percentage.

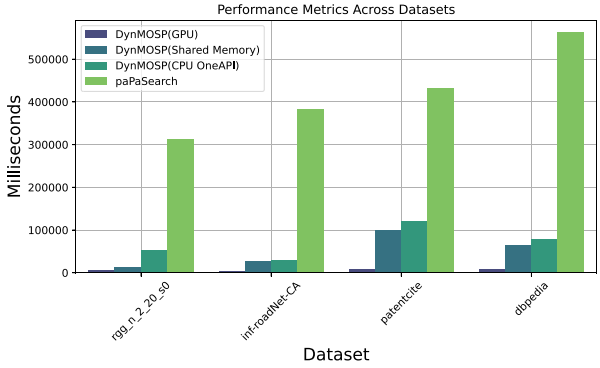


Fig. 12. Execution time comparison among paPaSearch and different DynaMOSP implementations.

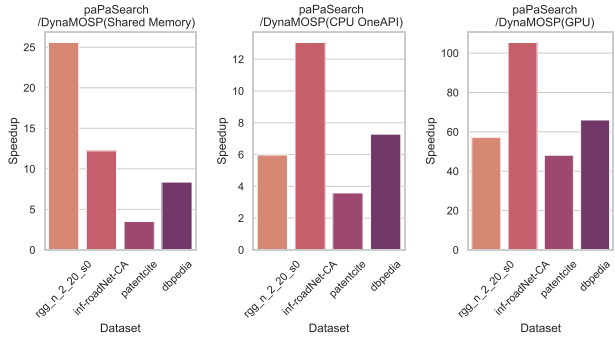


Fig. 13. Speedup comparison of various implementations relative to paPaSearch.

performance for both paPaSearch and shared memory DynaMOSP was achieved around 32 threads, we decided to maintain this configuration. We employed the same four networks as in the previous experiment and recorded the execution times for paPaSearch, the OpenMP-based DynaMOSP, and the SYCL-based DynaMOSP implementations (both on CPU and GPU), as depicted in Fig. 12. The results indicate that our SYCL implementation on the GPU exhibited the shortest execution time. The OpenMP-based implementation, utilizing shared memory, demonstrated the second shortest execution time, outperforming the OneAPI SYCL-based CPU implementation. In every scenario, paPaSearch showed the longest execution time. Fig. 13 illustrates the speedup of various DynaMOSP implementations relative to paPaSearch. The results indicate that the OpenMP-based DynaMOSP implementation can achieve a speedup of up to $25\times$. In comparison, the OneAPI SYCL implementation on the CPU attains up to $12\times$ speedup, while the SYCL implementation on the GPU exceeds $50\times$, reaching up to $100\times$ speedup.

For quality comparison, we observe the percentage of destinations obtaining optimal distances. Due to memory limits, paPaSearch cannot store all distance labels for large graphs. On small random graphs, over 94% of vertices achieve Pareto optimal distances, with no correlation to graph size. Additional results are in the appendix, available online.

VI. Related Works

This section reviews parallel solutions for the SOSP problem and explores both sequential and parallel approaches to finding MOSP in large networks.

A. Single Objective Shortest Path

Authors in [25] pioneered the parallelization of Dijkstra's shortest path algorithm. Thereafter, they introduced an array of nodes, termed as 'bucket', containing a tentative distance and concurrently relaxed all the outgoing edges with a weight of at most the tentative distance value [26]. Since then, a multitude of parallel algorithms for SOSP has been proposed in the literature [27], [28], [29].

Gunrock [30], a graph library for Nvidia GPUs, offers parallel SOSP implementation through a high-performance, three-step process: advance, filter, and compute, applied to the vertices' frontier. Here frontier refers to the set of vertices actively being processed at a particular step within an algorithm. A GPU-based Bellman-Ford SOSP algorithm, proposed in [22], uses a mechanism to remove duplicates from the frontier to reduce memory access. A framework for updating SOSP in dynamic networks is introduced in [14]. This framework includes an algorithm to identify vertices affected by changes and places them in a frontier for parallel processing to recompute distances. However, this approach struggles with race conditions and depends on multiple iterations of processing the same affected vertices to ensure accuracy. Our SOSP update algorithm adopts the framework suggested in [14] but uniquely mitigates race conditions by assigning each vertex's computations to a single thread. This strategy not only prevents race conditions but also minimizes redundant computations by reducing the likelihood of processing vertices multiple times.

B. Multi Objective Shortest Path

In static networks, various sequential MOSP algorithms have been proposed, including bi-objective [31], [32] and tri-objective [33], [34] shortest paths, along with pruning techniques [35]. A sequential MOSP algorithm for dynamic networks, combining optimistic linear support and label correcting methods for exact solutions, has been recently presented in [36], but it has scalability issues.

The authors in [6] proposed a parallel label-setting algorithm where the frontier list of labels is represented as the leaf nodes ordered lexicographically to find the Pareto optimal nodes in parallel while pruning the search space. The authors used the red-black tree as a data structure to ensure the balanced binary to limit the logarithmic search space in the worst case. Later, the authors in [37] replaced the red-black tree with the B-tree to eliminate the overhead of the red-black tree reconstruction after each change during the bulk update. On the other hand, the authors in [38] used a dominance check where objective cost is precalculated in a balanced tree to check if the new label is dominated by the existing labels based on the cost. Although Pareto optimal labels increase exponentially with graph size, limiting their number to the nodes allows for an effective, sub-optimal solution, as proposed in [39]. The authors in [40] primarily focused on road networks, ranking Pareto optimal labels using genetic algorithms and pruning the search space by utilizing meta-heuristics. They concentrated only on the most promising labels to obtain a near-optimal shortest path in GPU architecture. Prior to [16], no work addressed finding MOSP in parallel in large dynamic networks. We introduced a shared memory parallel algorithm for MOSP in large incremental networks in [16] and now extend this to fully dynamic networks, offering a SYCL-based implementation for heterogeneous computing architectures.

VII. Conclusion

In this study, we introduce DynaMOSP, a heuristic designed to efficiently find multi-objective shortest paths in large-scale, fully dynamic networks. Initially, we design a single-objective shortest path update algorithm that incorporates a simple yet effective grouping technique to enhance scalability. Then we leverage this algorithm to develop DynaMOSP that finds a single multi-objective shortest path solution efficiently. DynaMOSP prioritizes specific objectives based on user-defined preferences or, in the absence of preferences, balances all objectives to find the optimal path. A theoretical analysis of the proposed algorithm provides the criteria to achieve optimality. We implemented DynaMOSP using OpenMP and SYCL for shared memory CPUs and heterogeneous computing architectures, respectively. Empirical results demonstrate that the algorithm is scalable and applicable for any arbitrary number of objectives. We achieve speedup up to $57.22\times$ and $105.39\times$ over the state-of-the-art technique for CPU and Nvidia GPU, respectively.

To prevent infinite loops during the update propagation stage of our algorithm, we assume the graph stays connected despite topological changes. Our future work will focus on developing a more generic algorithm to manage potentially disconnected dynamic graphs. Additionally, we would like to explore the combination of user preferences to choose each Pareto optimal shortest path solution.

Acknowledgment

The authors are grateful to anonymous reviewers and associate editor for constructive suggestions that helped significantly improve the quality of the manuscript.

References

[1] H. Gao, W. Huang, and X. Yang, "Applying probabilistic model checking to path planning in an intelligent transportation system using mobility trajectories and their statistical data," *Intell. Automat. Soft Comput.*, vol. 25, no. 3, pp. 547–559, 2019.

[2] J. Li, T. Cai, K. Deng, X. Wang, T. Sellis, and F. Xia, "Community-diversified influence maximization in social networks," *Inf. Syst.*, vol. 92, 2020, Art. no. 101522.

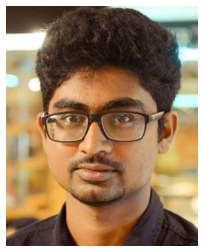
[3] X. Wu, G. Chen, and S. K. Das, "Avoiding energy holes in wireless sensor networks with nonuniform node distribution," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 5, pp. 710–720, May 2008.

[4] A. Khanda, F. Corò, F. B. Sorbelli, C. M. Pinotti, and S. K. Das, "Efficient route selection for drone-based delivery under time-varying dynamics," in *Proc. IEEE 18th Int. Conf. Mobile Ad Hoc Smart Syst.*, 2021, pp. 437–445.

[5] A. Khanda, F. Corò, and S. K. Das, "Drone-truck cooperated delivery under time varying dynamics," in *Proc. 2022 Workshop Adv. Tools Program. Lang. PLatforms Implementing Evaluating Algorithms Distrib. Syst.*, 2022, pp. 24–29.

[6] P. Sanders and L. Mandow, "Parallel label-setting multi-objective shortest path search," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 215–224.
[7] C. J. Weit, J. Wen, T. A. Zaidi, and D. N. Mavris, "Estimating supersonic commercial aircraft market and resulting CO_2 emissions using public movement data," *CEAS Aeronautical J.*, vol. 12, no. 1, pp. 191–203, 2021.
[8] S. K. A. Imon, A. Khan, M. Di Francesco, and S. K. Das, "Rasmalai: A randomized switching algorithm for maximizing lifetime in tree-based wireless sensor networks," in *Proc. 2013 IEEE Conf. Comput. Commun.*, 2013, pp. 2913–2921.
[9] K. Deb and H. Gupta, "Searching for robust pareto-optimal solutions in multi-objective optimization," in *Proc. 3rd Int. Conf. Evol. Multi-Criterion Optim.*, Springer, 2005, pp. 150–164.
[10] J. M. A. Pangilinan and G. K. Janssens, "Evolutionary algorithms for the multiobjective shortest path problem," *World Acad. Sci., Eng. Technol.*, vol. 25, pp. 205–210, 2007.
[11] F. Helff, L. Gruenwald, and L. d'Orazio, "Weighted sum model for multi-objective query optimization for mobile-cloud database environments," in *Proc. EDBT/ICDT Workshops*, 2016, pp. 1–6.
[12] A. Zhou, Q. Zhang, and Y. Jin, "Approximating the set of pareto-optimal solutions in both the decision and objective spaces by an estimation of distribution algorithm," *IEEE Trans. Evol. Comput.*, vol. 13, no. 5, pp. 1167–1189, Oct. 2009.
[13] X. Zan, Z. Wu, C. Guo, and Z. Yu, "A pareto-based genetic algorithm for multi-objective scheduling of automated manufacturing systems," *Adv. Mech. Eng.*, vol. 12, no. 1, 2020, Art. no. 1687814019885294.
[14] A. Khanda, S. Srinivasan, S. Bhowmick, B. Norris, and S. K. Das, "A parallel algorithm template for updating single-source shortest paths in large-scale dynamic networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 4, pp. 929–940, Apr. 2022.
[15] A. Khanda, S. Bhowmick, X. Liang, and S. K. Das, "Parallel vertex color update on large dynamic networks," in *Proc. IEEE 29th Int. Conf. High Perform. Comput. Data Analytics*, 2022, pp. 115–124.
[16] A. Khanda, S. Shovan, and S. K. Das, "A parallel algorithm for updating a multi-objective shortest path in large dynamic networks," in *Proc. Workshops Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2023, pp. 739–746.
[17] Y. Censor, "Pareto optimality in multiobjective problems," *Appl. Math. Optim.*, vol. 4, no. 1, pp. 41–59, 1977.
[18] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 363–374.
[19] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," *ACM Trans. Storage*, vol. 15, no. 4, pp. 1–40, 2020.
[20] C. Huan et al., "TeGraph+: Scalable temporal graph processing enabling flexible edge modifications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 8, pp. 1469–1487, Aug. 2024.
[21] Z. Chen et al., "CompressGraph: Efficient parallel graph analytics with rule-based compression," in *Proc. ACM Manage. Data*, vol. 1, no. 1, pp. 1–31, 2023.
[22] F. Busato and N. Bombieri, "An efficient implementation of the bellman-ford algorithm for kepler GPU architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 8, pp. 2222–2233, Aug. 2016.
[23] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proc. Conf. Assoc. Advance. Artif. Intell.*, 2015, pp. 4292–4293.
[24] J. Kunegis, "Konect: The koblenz network collection," in *Proc. 22nd Int. Conf. World Wide Web*, 2013, pp. 1343–1350.
[25] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders, "A parallelization of dijkstra's shortest path algorithm," in *Proc. 23rd Int. Symp. Math. Found. Comput. Sci.*, Springer, 1998, pp. 722–731.
[26] U. Meyer and P. Sanders, "δ-stepping: A parallelizable shortest path algorithm," *J. Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.
[27] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak, "Parallel shortest path algorithms for solving large-scale instances," in *Proc. Int. Conf. Shortest Path Problem*, 2006, pp. 249–290.
[28] G. E. Blelloch, Y. Gu, Y. Sun, and K. Tangwongsan, "Parallel shortest paths using radius stepping," in *Proc. 28th ACM Symp. Parallelism Algorithms Archit.*, 2016, pp. 443–454.
[29] J. Li, "Faster parallel algorithm for approximate shortest path," in *Proc. 52nd Annu. ACM SIGACT Symp. Theory Comput.*, 2020, pp. 308–321.
[30] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *Proc. 21st ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2016, pp. 1–12.
[31] A. Sedeno-Noda and A. Raith, "A dijkstra-like method computing all extreme supported non-dominated solutions of the biobjective shortest path problem," *Comput. Operations Res.*, vol. 57, pp. 83–94, 2015.
[32] F. He, H. Qi, and Q. Fan, "An evolutionary algorithm for the multi-objective shortest path problem," in *Proc. Int. Conf. Intell. Syst. Knowl. Eng.*, Atlantis Press, 2007, pp. 1276–1280.
[33] A. Hidalgo-Paniagua, M. A. Vega-Rodríguez, J. Ferruz, and N. Pavón, "Solving the multi-objective path planning problem in mobile robotics with a firefly-based approach," *Soft Comput.*, vol. 21, pp. 949–964, 2017.
[34] F. Ahmed and K. Deb, "Multi-objective optimal path planning using elitist non-dominated sorting genetic algorithms," *Soft Comput.*, vol. 17, pp. 1283–1299, 2013.
[35] D. Duque, L. Lozano, and A. L. Medaglia, "An exact method for the biobjective shortest path problem for large-scale road networks," *Eur. J. Oper. Res.*, vol. 242, no. 3, pp. 788–797, 2015.
[36] J. M. da Silva, G. D. O. Ramos, and J. L. Barbosa, "The multi-objective dynamic shortest path problem," in *Proc. 2022 IEEE Congr. Evol. Comput.*, 2022, pp. 1–8.
[37] S. Erb, M. Kobitzsch, and P. Sanders, "Parallel bi-objective shortest paths using weight-balanced b-trees with bulk updates," in *Proc. Int. Symp. Exp. Algorithms*, Springer, 2014, pp. 111–122.
[38] F.-J. Pulido, L. Mandow, and J.-L. Pérez-de-la Cruz, "Dimensionality reduction in multiobjective shortest path search," *Comput. Operations Res.*, vol. 64, pp. 60–70, 2015.
[39] P. M. de las Casas, A. Sedeno-Noda, and R. Borndörfer, "An improved multiobjective shortest path algorithm," *Comput. Operations Res.*, vol. 135, 2021, Art. no. 105424.
[40] Y. Yao, Z. Peng, and B. Xiao, "Parallel hyper-heuristic algorithm for multi-objective route planning in a smart city," *IEEE Trans. Veh. Technol*, vol. 67, no. 11, pp. 10307–10318, Nov. 2018.

S. M. Shovan (Student Member, IEEE) received the bachelor's of science and master's of science degrees from the Department of Computer Science and Engineering from Rajshahi University of Engineering and Technology. He is enrolled in Computer Science Department in the PhD program of Missouri University of Science and Technology. His research interests includes large-scale dynamic graph analysis, high performance computing and algorithm analysis.



Arindam Khanda (Member, IEEE) received the BTech degree in ECE from the Institute of Engineering and Management in 2015 and the MTech degree in software systems from BITS Pilani in 2019. He is currently working toward the PhD degree with the Missouri University of Science and Technology. His research interests include parallel programming models, dynamic graphs, and high-performance computing (HPC).

Sajal K. Das (Fellow, IEEE) is a Curators' distinguished professor of computer science, and Daniel St. Clair Endowed chair with Missouri University of Science and Technology. His research interests include parallel computing, cloud and edge computing, graph algorithms, sensor and IoT networks, mobile and pervasive computing, cyber-physical systems, smart environments, cyber-security, and biological and social networks. He is the editor-in-chief of *Pervasive and Mobile Computing journal*, and associate editor of IEEE TRANSACTIONS ON SUSTAINABLE COMPUTING, IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, IEEE/ACM TRANSACTION ON NETWORKING, and *ACM Transactions on Sensor Networks*.