Massive Time Lost:

In 2024, the average U.S. driver lost **43 hours** annually due to traffic congestion—equivalent to a full work week

Economic Impact:

That lost time cost **$771 per driver** and totaled **$74 billion** nationwide in productivity and fuel wasted

Extreme Hotspots:

Some corridors, like Interstate 95 through Stamford, CT, see drivers losing up to **150 hours** per year on that single stretch

Urban Leaders Suffer Most:

Cities such as New York and Chicago top the charts at **102 hours** lost per driver, costing over $1,800 each in time wasted

Safety & Efficiency Risks:

Severe congestion delays emergency response, increases fuel consumption, and degrades air quality

Dynamic Environment:

Road closures, accidents, and new lanes open **minute by minute**, yet most navigation systems still **recompute routes from scratch**, leading to further delays and resource waste .

The Challenge at Scale

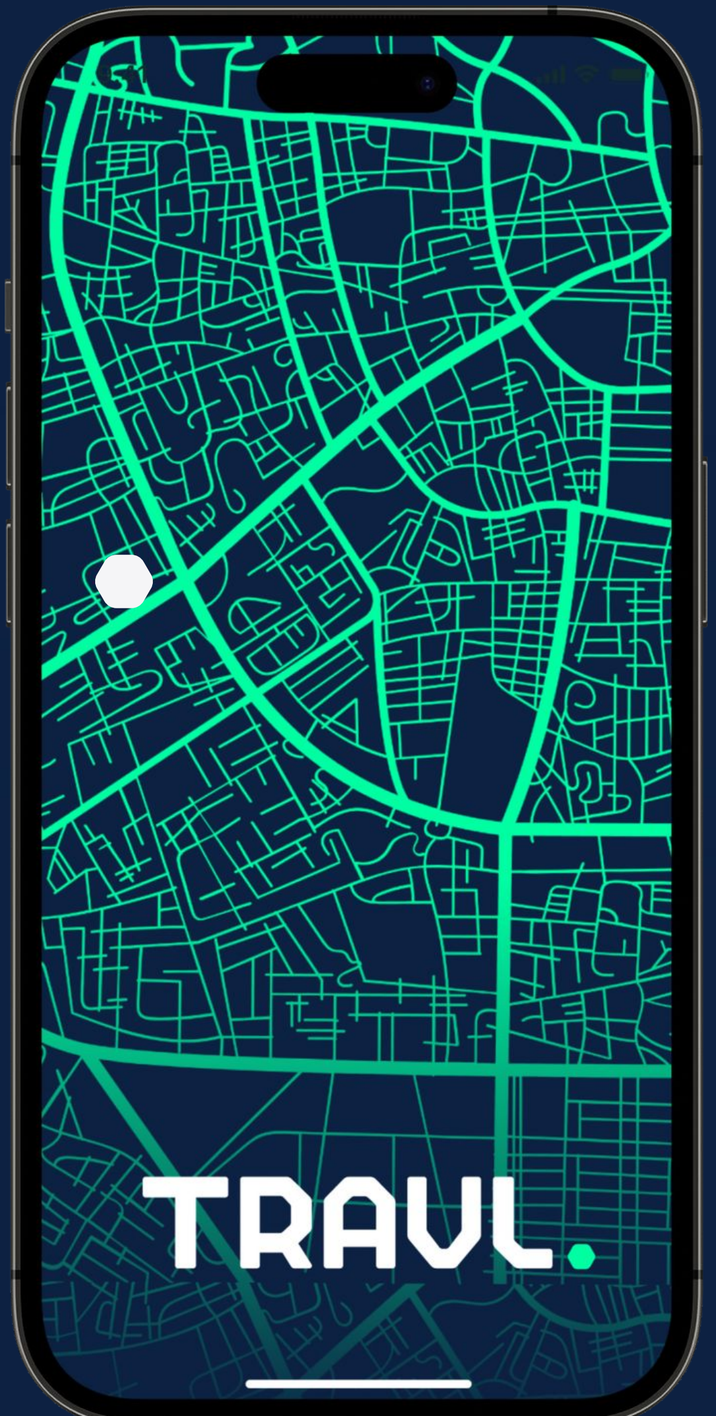
Smart-city routing graphs span **millions of nodes and billions of edges**—full recomputation on every change is infeasible for real-time need.

Call to Action

We need **incremental, parallel** algorithms that update only the **affected regions** of the network to restore optimal routing in **milliseconds**, not minutes.

PARALLEL ALGORITHM TEMPLATE FOR UPDATING SINGLE-SOURCE SHORTEST PATHS IN LARGE-SCALE DYNAMIC NETWORKS

Momin Munir _ 22I-0854
Wajahat Ullah Khan _ 22I-0776
Ahmad Aqeel _ 22I-1134



Dynamic Graphs:

Dynamic graphs are graph data structures whose **topology changes over time** — that is, **vertices (nodes)** and **edges (connections)** can be **added or deleted** as real-world events occur.

“Processing Matters”

Real Time Navigation

Threat Detection

Social Media Analytics

We **cannot afford to recompute** graph algorithms (like shortest path) **from scratch** after every change.

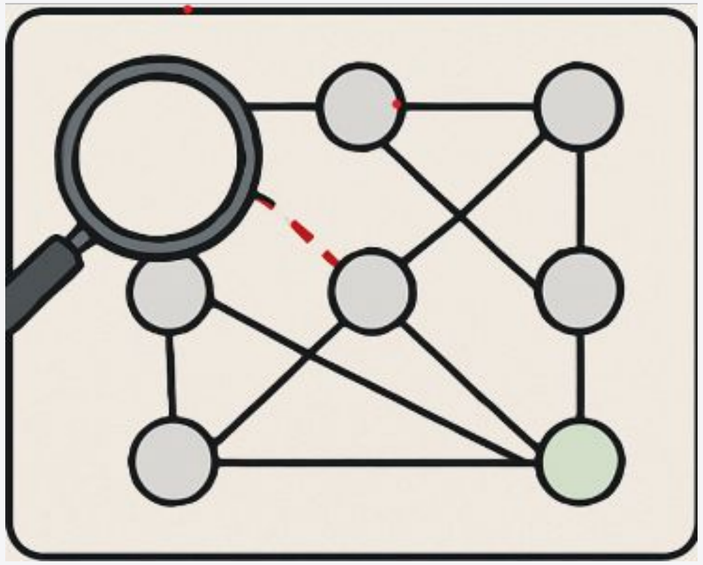
Instead, we need **incremental or update-based algorithms** that only process the **affected parts of the graph**.

Single Source Shortest Path in Dynamic Networks

Problem

(SSSP) algorithms work only on **static graphs**.
real-world networks are dynamic — their structure **changes over time** (edges are added or deleted

Recomputing the SSSP from scratch after every change is **computationally expensive and inefficient**, especially on **large-scale graphs**.



Dijkstra Algorithm
Bellman Ford

Motivation

Real-world applications (e.g., social, transport, and communication networks) need **fast updates** to graph properties as changes occur.
There's a **lack of scalable parallel algorithms** that:
Efficiently update SSSP for **dynamic networks**
Work across **different hardware platforms** (e.g., multicore CPUs and GPUs)
The paper proposes a **novel parallel framework** that focuses on **identifying only the affected subgraph**, enabling faster updates.

Serial Implementation:

On each edge change (`e(u, v)`):

1. Identify affected vertex
2. Add it to a Priority Queue
3. Use Dijkstra's algorithm to update distances
4. Repeat until the queue is empty

Red Area:

Loop dequeues a node, updates its shortest distance using Dijkstra's algorithm, and then enqueues all its neighbors if updates occur. While this approach ensures correctness, it introduces significant inefficiency in dynamic graphs, particularly when updates are frequent or widespread.

1.) **Repeated Relaxation**
2.) **Local Change, Global Update**
3.) **Lack of Parallelism**

Input: Weighted Graph $G(V, E)$, SSSP Tree T
Changed edge $\Delta E = e(u, v)$
Output: Updated the SSSP Tree Tu:
1: Function SingleChangge $(Æ, G, T)$:
// Find the affected vertex, x
3: **if** $\text{Dist}_u[u] > \text{Dist}_u[v]$ then
$x + u,\ y \to v$
6: **else**
$x \leftarrow v,\ u \to u$
// Initialize Priortity Queue PQ and update $\text{Dist}_u[x]$
7: PQ $\leftarrow x$
8: **if** E is ininserted then
$\text{Dist}_u[x] \leftarrow \text{Dist}_u[y] + W(u, v)$
10: if E is deleted then
// Udpdate the subgraph affected by x
12: **while** (PQ not empty) **do**
13: Update+ *F*alse
14: $z \leftarrow$ PQ:top()
16: Updated SSSP(zG, T)
// Calculate a shortest dististence from source vertex to z
17: **if** Updated $=$*True* then
18: **for** n where n is the neighorof z do
PQ:enqueue(n)
19: PQ:enqueue(n)

Algorithm 1: Updating SSSP for a Single Change

Input: Weighted Graph $G(V, E)$, SSSP Tree T, Changed edge DE)

Output: Updated the SSSP Tree T_u

```
1  Function SingleChange (ΔE, G, T):
4  // Find the affected vertex, x
     3  if Distu,|u| > Distu.|v|  then
     4  else  x = v, y =u
     // Initializate Priority quQ and
     5  while (PQ not empty) do
     8      Updated = False
     9      z = PQ:top( – Distu,|y + W(u,v)
    10      if UpdatedSSSP(z, G, T)
    11   if Undated = True
    18   for n where n is theighbor f z do
    19   PQ:enqueue:(n)
```

Data Dependency Ordering

Priority queue (PQ) enforces the correct **data-flow ordering**. By always popping the vertex *z* with the smallest current distance estimate (lines 14–15), the algorithm guarantees that every relaxation on *z* sees the final, minimal distances of all predecessors. In other words, you cannot process *z* (and call `UpdatedSSSP(z,G,T)`) until all vertices with shorter paths have already been settled

Control Dependency for Propagation

After updating *z*'s distance, the boolean `Updated` flag guards whether its neighbors should be enqueued (lines 17–19). Only if `Updated == True`—i.e., *z*'s distance actually decreased—do we propagate further relaxations to its neighbors. This **control dependency** prevents redundant work on unaffected subgraphs

Termination Dependency

The outer loop `while (PQ not empty)` (line 12) encodes the **termination condition**: the algorithm can only stop once **all** vertices whose distances might change have been extracted and processed from the queue. Draining the PQ therefore ensures completeness—no dependent updates remain pending

logo.

DYNAMIC SSSP PROPOSED BY PAPER

Two-step parallel framework:

1.) Algorithm One

identifying only the subgraph affected by edge insertions/deletions

2.) Algorithm Two

updating distances in that subgraph iteratively without full recomputation

Approach and Implementation:

1.) Pre Vertex Flags
2.) No Global Synchronization
3.) Dynamic scheduling to balance workload across processing units.

Purpose:

In a fully parallel pass, determine exactly which vertices in the existi
of recent edge deletions or insertions.

Input:

1.)Current graph G and its SSSP tree T
2.)Changed-edge set DE = Del U Ins
3.)Per-vertex arrays: `Parent[v]`, `Dist[v]`

Output:

1.)Updated copy of the graph G_u (with deletions applied, new edges added)
2.)Two Boolean flag arrays, size |V|:
`Affected_Del[v]`: v's subtree must be reset
`Affected[v]`: v's distance may decrease

Effectiveness:

Fully Parallel:

No locks or barriers—each changed edge is handled independently.

Exact Locality:

Only vertices whose distance *could* change are flagged, drastically reducing work in the subse

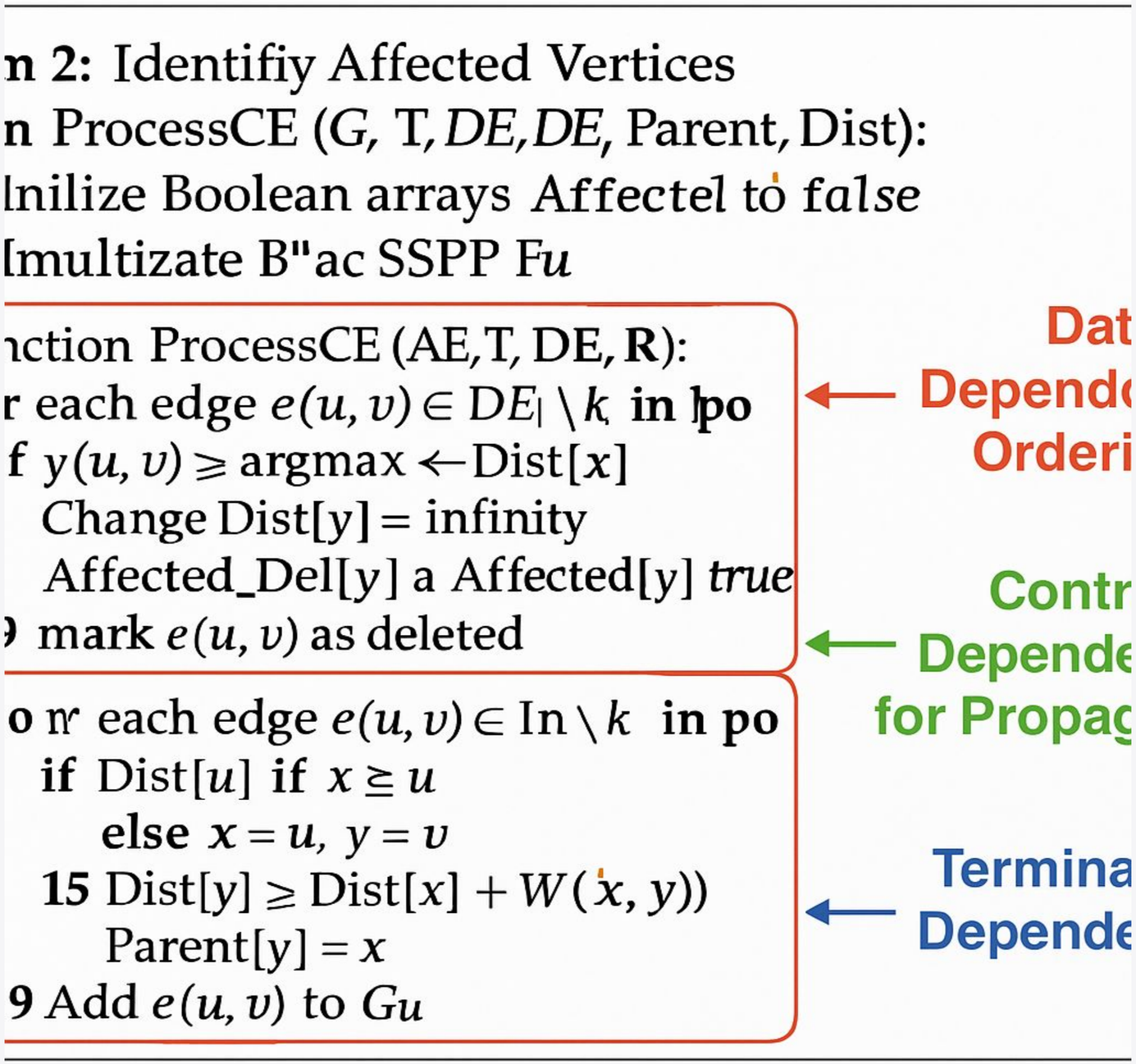
Preparation for Step 2:

A trimmed graph G_u
Arrays `Affected_Del[]` and `Affected[]` pinpointing the precise subgraph to r$_{evisit.}$

Algorithm 2. Identify Affected Vertices

1: **Function** `ProcessCE` $(G, T, \Delta E, Parent, Dist)$:
2: Initialize Boolean arrays $Affected\_Del$ and $Affected$ to $false$
3: $G_u \leftarrow G$
4: **for** *each edge* $e(u,v) \in Del_k$ *in parallel* **do**
5: **if** $e(u,v) \in T$ **then**
6: $y \leftarrow argmax_{x \in \{u,v\}}(Dist[x])$
7: Change $Dist[y]$ to infinity
8: $Affected\_Del[y] \leftarrow True$
$Affected[y] \leftarrow True$
9: Mark $e(u,v)$ as deleted
10: **for** *each edge* $e(u,v) \in Ins_k$ *in parallel* **do**
11: **if** $Dist[u] > Dist[v]$ **then**
12: $x \leftarrow v, y \leftarrow u$
13: **else**
14: $x \leftarrow u, y \leftarrow v$
15: **if** $Dist[y] > Dist[x] + W(x,y)$ **then**
16: $Dist[y] \leftarrow Dist[x] + W(x,y)$
17: $Parent[y] \leftarrow x$
18: $Affected[y] \leftarrow True$
19: Add $e(u,v)$ to G_u

Data Dependency and Hotspot Areas



Purpose:

In a fully parallel pass, propagate distance changes through disconnected subtrees and relax edges until the SSSP tree converges.

Input:

1) Trimmed graph G_n and its SSSP tree T
2) Per-vertex arrays: Dist[v], Parent[v]
3) Boolean flag arrays (|V| entries) from Alg 2:

- **Affected_Del[v]: subtree of v must be reset**
- **Affected[v]: v's distance (or its descendants') may decrease**

Output:

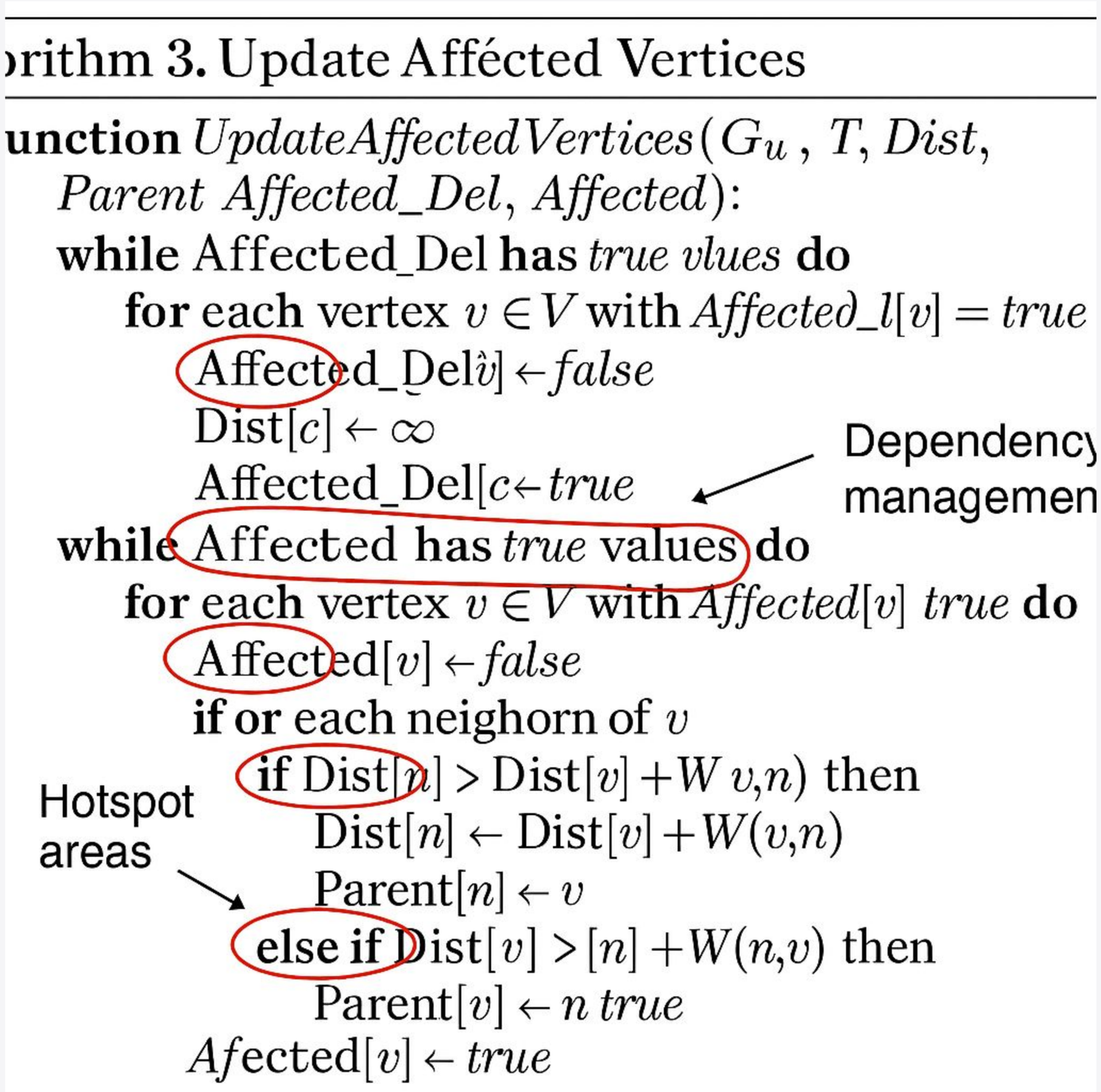
1) Updated SSSP tree T_n
2) Final Dist[] and Parent[] arrays encoding the new shortest paths

Effectiveness:

- Fully Parallel: Two barrier-free loops over flagged vertices
- Lock-Free Convergence: Iteratively refines distances without mutexes
- Exact Locality: Only vertices in Affected_Del or Affected are touched

Algorithm 3. Update Affected Vertices

```
1: Function UpdateAffectedVertices(G_n, T, Dist,
   Parent Affected_Del and Affected):
2:   while Affected_Del has true values do
3:     for each vertex v ∈ V such that Affected_Del[v] = true
       in parallel do
4:       Affected_Del[v] ← false
5:       for all vertex c, where c is child of v in the SSSP tree T do
6:       Set Dist[c] as infinity
7:       Affected_Del[c] ← True
8:       Affected[c] ← True
9:   while Affected has true values do
10:    for each vertex v ∈ V such that Affected[v] = true in paral-
       lel do
11:      Affected[v] ← False
12:      for vertex n, where n ∈ V and n is neighbor of v do
13:        if Dist[n] > Dist[v] + W(v, n) then
14:          Dist[n] ← Dist[v] + W(v, n)
15:          Parent[n] ← v
16:          Affected[n] ← True
17:        else if Dist[v] > Dist[n] + W(n, v) then
18:          Dist[v] ← Dist[n] + W(n, v)
19:          Parent[v] ← n
20:          Affected[v] ← True
```



`while Affected_Del has true values`
Hotspot due to possible deep subtree resets; high workload if many deletions.

`Affected_Del[v] ← false`
Frequent write per flagged vertex; poor locality can degrade performance.

`while Affected has true values`
Key convergence loop; may run many times based on graph structure.

`Affected[v] ← false`
High-frequency flag reset; essential for correctness and efficiency.

`if Dist[n] > Dist[v] + W(v,n)`
Main relaxation check; dominates compute and memory usage.

`else if Dist[v] > Dist[n] + W(n,v)`
Reverse check; doubles edge operations, impacts branches and caches.

logo.

IMPLEMENTATIO N STARTEGY

Our Brand Tagline

Serial Implementation:

```
//------------------------------------------------------------------------
---
// 1. INITIAL “FROM-SCRATCH” SSSP (DIJKSTRA)
//------------------------------------------------------------------------
---
procedure ComputeSSSP(G = (V,E), source s):
  // Initialize distances and parent pointers
  for each v in V do
    Dist[v]   ← ∞
    Parent[v] ← NIL
  Dist[s] ← 0

  // Min-priority queue keyed by Dist
  PQ ← new MinHeap()
  for each v in V do
    PQ.insert(v, Dist[v])

  while PQ is not empty do
    u ← PQ.popMin()        // vertex with smallest Dist
    for each edge (u → v) ∈ E do
      if Dist[v] > Dist[u] + w(u,v) then
        Dist[v]   ← Dist[u] + w(u,v)
        Parent[v] ← u
        PQ.decreaseKey(v, Dist[v])
  return (Dist, Parent)



//------------------------------------------------------------------------
---
// 2. SEQUENTIAL UPDATE FOR A BATCH OF CHANGES
//   (uses SingleChange for each edge insertion/deletion)
//------------------------------------------------------------------------
---
procedure UpdateSSSPSequential(G, changes, Dist, Parent):
  // changes: list of (u,v, type) where type ∈ {INSERT, DELETE}
  for each (u,v,type) in changes do
    SingleChange(u, v, type, G, Dist, Parent)
```

```
//------------------------------------------------------------------------
// 3. HANDLE ONE EDGE CHANGE (Algorithm 1 adapted)
//------------------------------------------------------------------------
procedure SingleChange(u, v, type, G, Dist, Parent):
  // 3.1 Identify which endpoint is “farther” in the current tree
  if Dist[u] > Dist[v] then
    x ← u;  y ← v
  else
    x ← v;  y ← u

  // 3.2 Prepare PQ for affected subgraph
  PQ ← new MinHeap()

  // 3.3 If it’s an insertion, attempt to relax y via x
  if type == INSERT then
    if Dist[y] > Dist[x] + w(x,v) then
      Dist[y]   ← Dist[x] + w(x,v)
      Parent[y] ← x
      PQ.insert(y, Dist[y])

  // 3.4 If it’s a deletion, and (x,y) was in the SSSP tree, disconnect x
  else if type == DELETE then
    if Parent[x] == y or Parent[y] == x then
      // Choose the endpoint that was child in the tree
      if Parent[x] == y then
        victim ← x
      else
        victim ← y
      Dist[victim]   ← ∞
      Parent[victim] ← NIL
      PQ.insert(victim, Dist[victim])

  // 3.5 Propagate updates throughout the affected region
  while PQ is not empty do
    z ← PQ.popMin()
    // Re-relax all neighbors of z
    for each edge (z → n) ∈ G.E do
      if Dist[n] > Dist[z] + w(z,n) then
        Dist[n]   ← Dist[z] + w(z,n)
        Parent[n] ← z
        PQ.insertOrDecrease(n, Dist[n])

  // End of SingleChange
```

Open MPI Based Implementation:

Parallel Target:

Distributed-memory clusters using MPI/C++ for **exascale scalability**.

Key Idea:

Touch only the **affected subgraphs** and coordinate updates via MPI, rather than full recomputation.

Data Decomposition

1. **Vertex-cut Partitioning**: assign each MPI rank a disjoint set of vertices and all incident edges; minimizes redundant storage and localizes most relaxationas.
2. **Graph Partitioning Libraries**: use **ParMETIS** for balanced 1D/2D cuts or **Zoltan** for adaptive repartitioning; both integrate with MPI and scale to millions of vertices.
3. **Ghost/Halo Cells**: each rank also keeps “ghost” copies of one-hop neighbors owned by other ranks, enabling boundary relaxations

Phase 1 – Identify Affected Vertices

- **Broadcast Changes**: collect all edge insertions/deletions on rank 0 and `MPI_Bcast` to all ranks, or have each rank `MPI_Allgather` its local updates Intel.

- **Local Marking**: each rank scans only those changes touching its owned or ghost vertices, marking them “affected” if insertion lowers a known distance or deletion severs a tree edge
- **No Locks Needed**: marking is embarrassingly parallel and incurs zero synchronization within a rank.

Phase 2 – Iterative Relaxation & Convergence

Local Relax: for each affected vertex *v*, relax all outgoing edges *(v,u)*:

```
if dist[u] > dist[v] + w(v,u):
    dist[u] = dist[v] + w(v,u)
   mark u as affected
```

Boundary Exchange:
after local relax, exchange updated `dist` values of boundary vertices via `MPI_Neighbor_alltoallv` over an MPI graph communicator. This sends only to ranks that own those ghosts.

Overlap Comm/Comp:
Post non blocking sends/receives (`MPI_Isend`/`MPI_Irecv`) for boundary data, then compute on purely local vertices; `MPI_Waitall` before integrating incoming data.

Global Convergence:
Maintain a `changed` flag per rank; use `MPI_Allreduce(changed, MPI_LOR)` to detect when no rank had updates, then terminate

MPI Communication Patterns

- **MPI_Graph_Comm**: build with `MPI_Dist_graph_create_adjacent` to encode partition adjacency, enabling concise neighbor collectives open-mpi.org.
- **MPI_Neighbor_alltoallv**: exchanges variable-length messages with each neighbor in one collective, avoiding multiple point-to-point calls docs.open-mpi.org.
- **Nonblocking Collectives**: where available (MPI-3 onward), use `MPI_Ineighbor_alltoallv` + `MPI_Wait` to overlap with computation MPI Forum.
- **Asynchronous Progress**: ensure your MPI library is configured for network progression, or use explicit progression threads if needed

Pseudocode:

```
MPI_Init(...)
// 1. Partition & Ghost Setup
PartitionGraphWithParMETISorZoltan()
BuildMPI_Graph_Comm()

for each batch_of_changes do
  // Phase 1: Identify
  if rank==0: collect_and_broadcast(changes)
  LocalMarkAffected(changes)

  // Phase 2: Relax & Converge
  do
    changed_local = false
    RelaxLocalVertices()       // update dist[], mark new affected
    PostNonblockingBoundarySends()
    ComputeInteriorVertices()   // overlap
    WaitBoundaryReceives()
    IntegrateReceivedDist()     // update ghosts
    MPI_Allreduce(changed_local, changed_global, MPI_LOR)
  while(changed_global)

  // Rebalance?
  assess_load_imbalance()
  if imbalance_exceeds_threshold:
    RepartitionWithZoltanOrParMETIS()
    RebuildGhostLayers()

MPI_Finalize()
```

METIS:

Using METIS with MPI

1. **Partition on Rank 0**
   – Rank 0 loads full graph, calls `METIS_PartGraphKway()` → gets `part[]`.

2. **Scatter to MPI Ranks**
   –Rank 0 sends each process its vertices, edges, and ghost neighbors.

3. **Run MPI Strategy**
   – Each rank builds subgraph, sets up `MPI_Dist_graph_create_adjacent()`, performs local relaxations, exchanges boundary data, and checks convergence.

OpenMP Based Implementation:

Data Structures:

- `Dist[v]` (float or int): current shortest-path estimate from source
- `Parent[v]` (vertex ID or –1): pointer to parent in SSSP tree
- `AffectedDel[v]` (bool): true if **deletion** has just disconnected v
- `Affected[v]` (bool): true if v needs its distance re-relaxed

All four arrays are of length O (|V|) and shared across threads; updates to `Affected*[]` are idempotent (only set from false→true), so no atomics or locks are needed.

Parallel Change Processings:

.1 Mark Deletions

Each thread scans a block of the `changes[]` array; if an edge deletion severs the SSSP tree, its child endpoint is marked:

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < num_changes; i++) {
    int u = changes[i].u, v = changes[i].v;
    if (changes[i].type == DELETE) {
        // If (u,v) was a tree-edge, mark the child for disconnection
        if (Parent[u] == v)       AffectedDel[u] = true;
        else if (Parent[v] == u)  AffectedDel[v] = true;
    }
}
```

2.2 Process Insertions

Similarly, each thread handles insertions by relaxing the “far” endpoint:

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < num_changes; i++) {
    int u = changes[i].u, v = changes[i].v, w = changes[i].w;
    if (changes[i].type == INSERT) {
        int x = (Dist[u] > Dist[v] ? u : v);
        int y = (x == u ? v : u);
        int alt = Dist[x] + w;
        if (alt < Dist[y]) {
            Dist[y]    = alt;
            Parent[y]  = x;
            Affected[y] = true;
        }
    }
}
```

Constant-time work per change; `static` scheduling suffices here

Two-Wave Lock-Free Update

- **Wave 1** – Disconnect Deletions: Repeatedly scan all vertices in parallel; any vertex whose parent-edge was deleted “consumes” its flag and marks its children for disconnection and re-relaxation.
- **Wave 2** – Iterative Relaxation: Repeatedly scan all vertices in parallel; any vertex flagged for re-relaxation explores its neighbors to propagate new shortest-path distances both forward and backward.

Convergence via Flags

- A global `changed` flag (boolean) controls each `while` loop. Within each iteration, any thread that makes a disconnection or relaxation sets `changed=true`, ensuring the waves repeat until no further updates occur.

Irregular Workload Balancing

- Subtrees and wavefronts can be wildly different in size, so we parcel out single-vertex tasks dynamically to threads, avoiding idle cores.

Open MP Commands Used:

```
// Wave 1: Disconnect cascading deletions
bool changed = true;
while (changed) {
    changed = false;
    #pragma omp parallel for schedule(dynamic, 1)
    for (int v = 0; v < V; v++) {
        if (AffectedDel[v]) {
            // consume flag & mark children…
            changed = true;
```

```
// Wave 2: Iterative relaxation & re-attachment
changed = true;
while (changed) {
    changed = false;
    #pragma omp parallel for schedule(dynamic, 1)
    for (int v = 0; v < V; v++) {
        if (Affected[v]) {
            // relax neighbors…
            changed = true;
```

logo.

THANK YOU