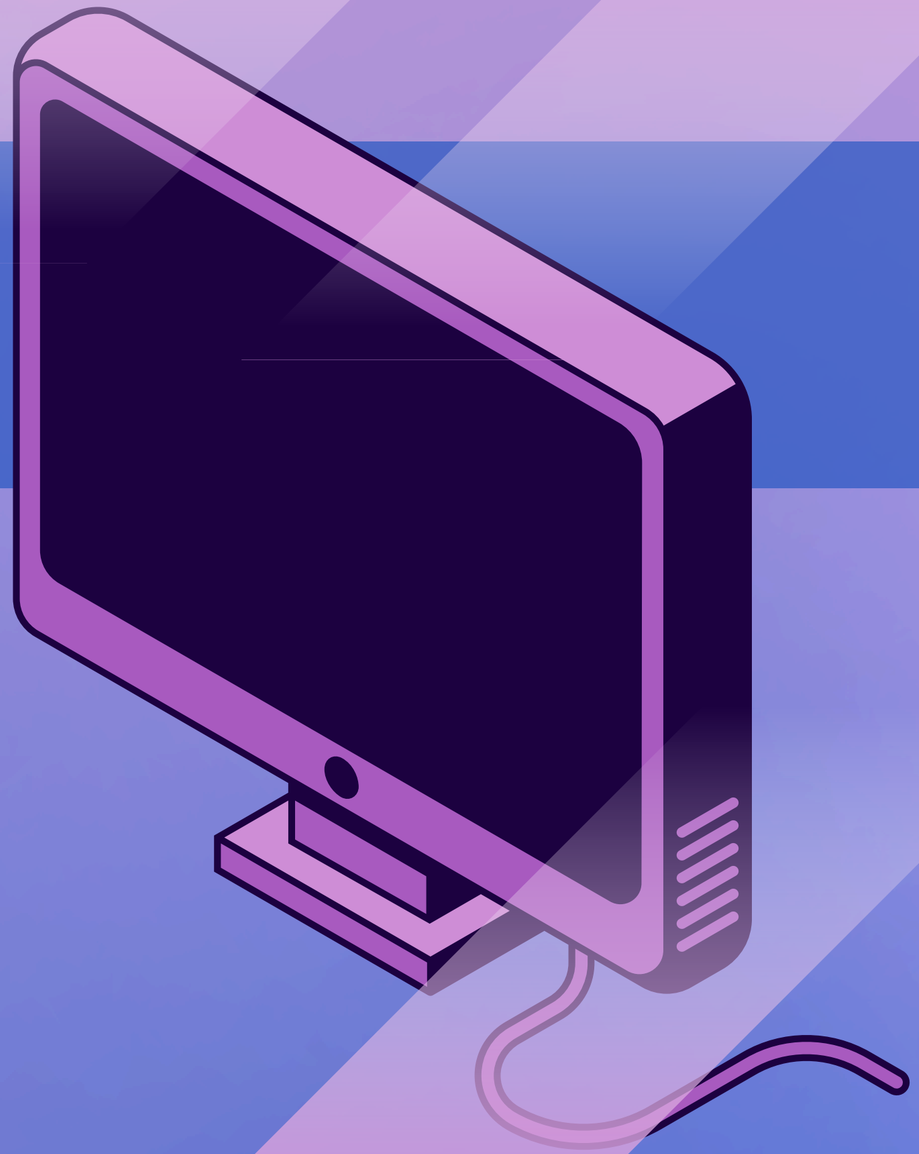
Research Article No. 3

SSSP in Large-Scale Dynamic Networks

Parallel and Dristributed computing



Emaan Ali i222325 | Wania Naeem i222369 |
Nabeeha Shafiq i222336

Introduction

- World Problems consists of grap-like structures
- SSSP (single source shortest path) is a building block for many networks

Why do we require parallelized Techniques?

- For dynamic graphs , recomputing SSSP tree from scratch is inefficient
- The article proposes a parallel algorithm template to update and compute the SSSP tree dynamically

Shared Memory Parallelization

CPU- Based (OpenMP)

- Udates are handled in batches
- Dynamic schedueling is used to balance load
- Uses load buffer per thread

- In parallel loop mark verticies affected by deletion or insertion
- Iterate until no affacted vertices remains

Deletion: disconnect affected vertices and mark subtree

- Insertion : if a better path is found update distance and mark children
- Each update round is done in parallel avoiding locks where possible

Shared Memory Template

Parallel SSSP Update Framework

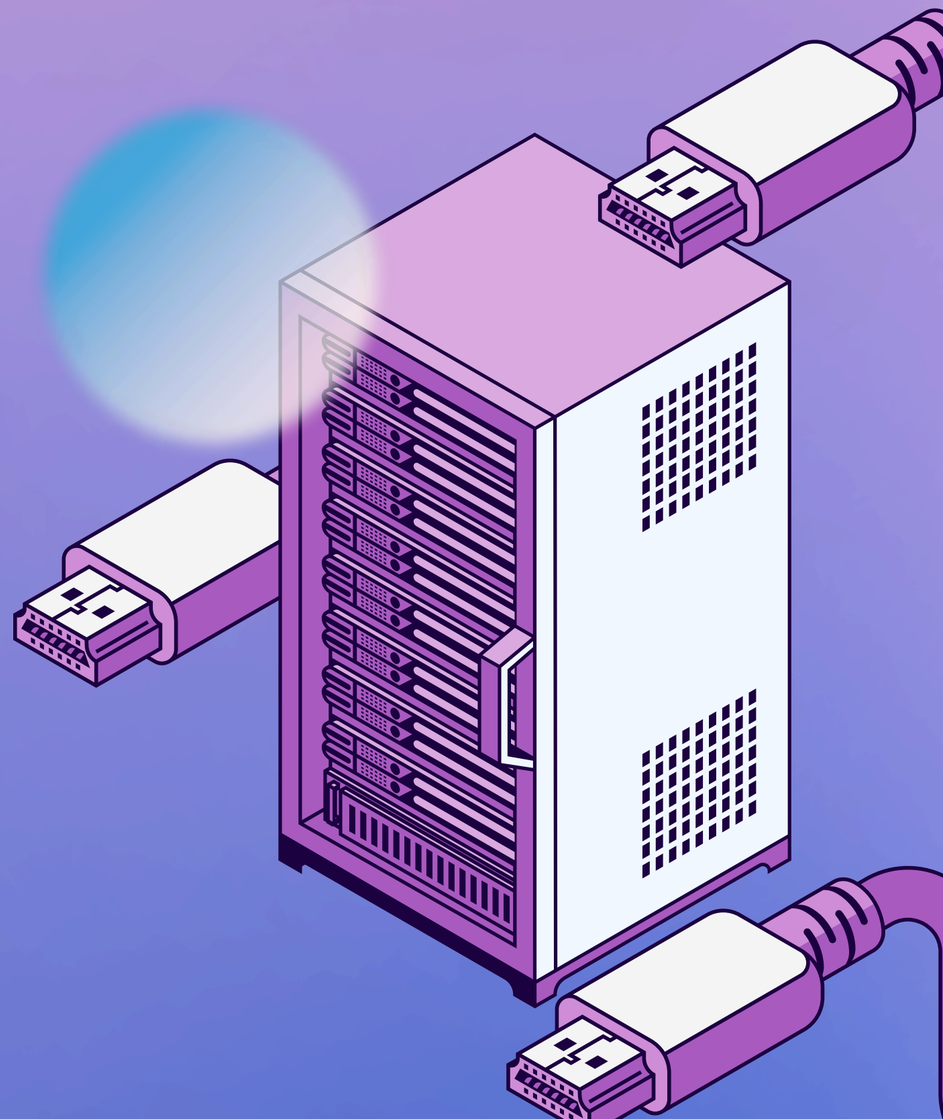
Step#01 Identifying affected vertices

- Run each edge in parallel and find the affected vertices
- Edges are inserted or deleted
- Can run in parallel without synchronization

EX

A → B → X → C

If B-->C is detected, then C's parent is removed, so C vertex is affected.

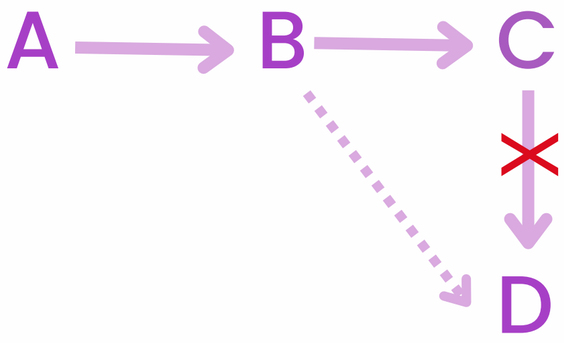


Shared Memory Template

Step#02 Update affected subgraph

- only affected part are recalculated
- Deletion : removes affected branches
- Insertion: Relax path through new edges
- if shorter path is created, parents and children are updated

EX



Algorithm 4. Asynchronous Update of SSSP

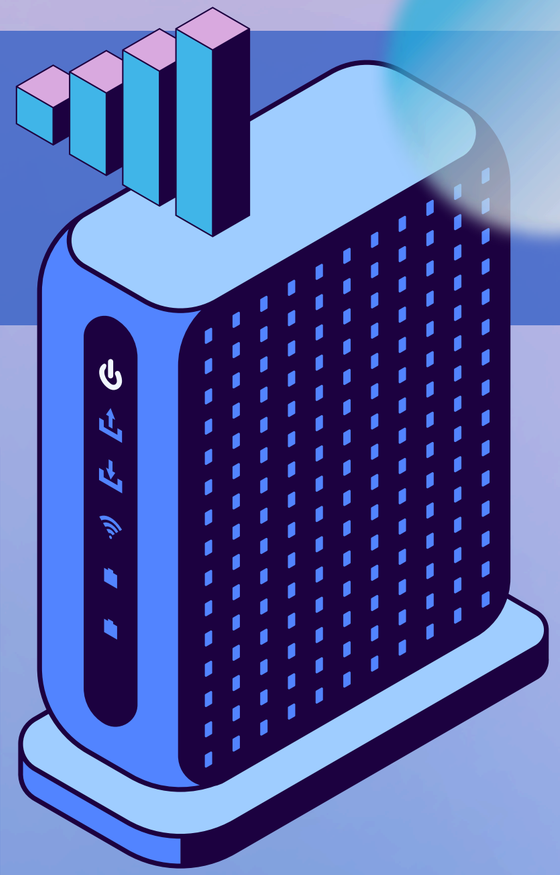
Input: $G(V, E)$, T, $Dist$, $Parent$, source vertex s, $\Delta E(Del_k, Ins_k)$
Output: Updated SSSP Tree T_k

1: **Function** `AsynchronousUpdating` (ΔE, G, T):
2: Set Level of Asynchrony to A.
3: $Change \leftarrow True$
4: **while** $Change$ **do**
5: $Change \leftarrow False$
6: **pragma omp parallel schedule (dynamic)**
7: **for** $v \in V$ **do**
8: **if** $Affected\_Del[v] = True$ **then**
9: Initialize a queue Q
10: Push v to Q
11: $Level \leftarrow 0$
12: **while** Q *is not empty* **do**
13: Pop x from top of Q
14: **for** c *where* c *is child of* x *in* T **do**
15: Mark c as affected and change distance to infinite
16: $Change \leftarrow True$
17: $Level \leftarrow Level + 1$
18: **if** $Level \leq A$ **then**
19: Push c to Q

CPU BASED PERFORMANCE EVALUATION

- 5X Speed vs basline method (galois)
- A syncronous function is faster than synchronous
- Speed improves with:
  1. Fewer aqffected nodes
  2. Larger batch size
  3. More threads

GPU- Based Parallelization Strategy



Step A – Detect & mark (inside each computer, OpenMP)

- Every CPU core scans its own chunk.
-

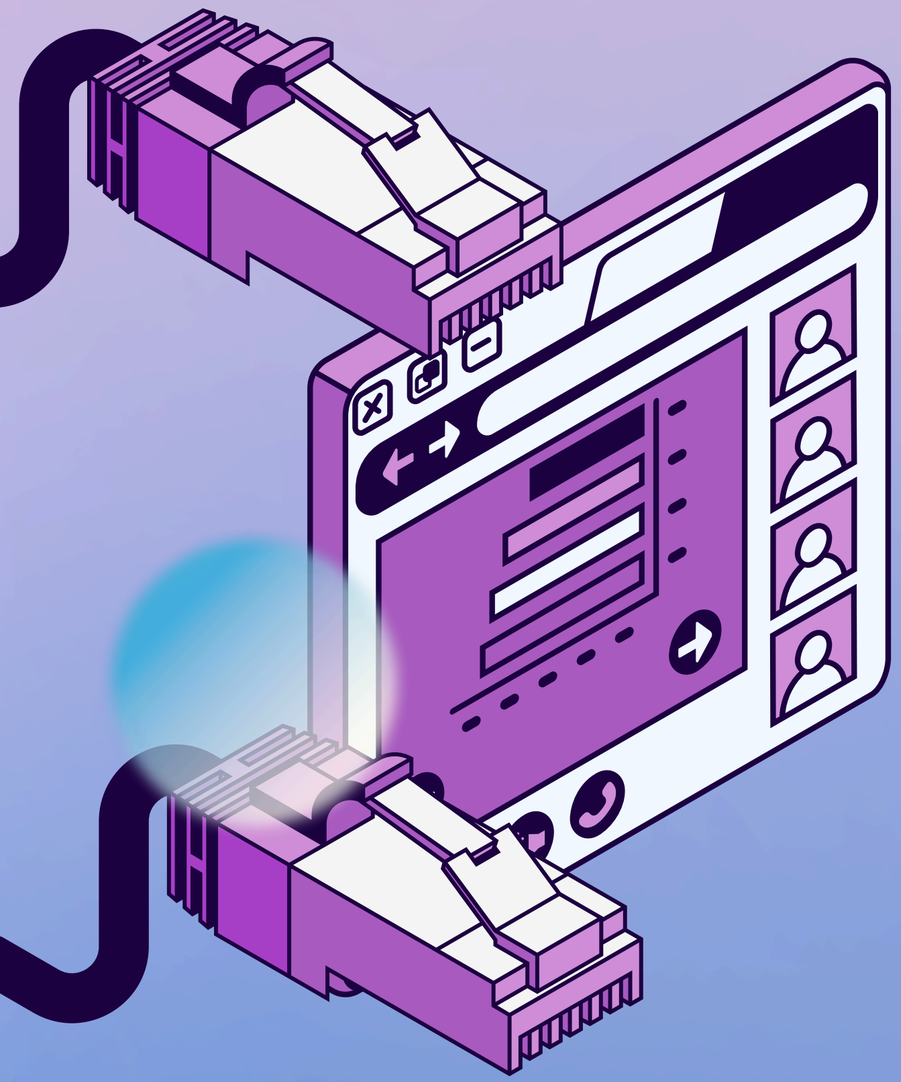
If a vertex's shortest path might change ⇒ set Affected = true.

Step B – Tell other computers (MPI)

- For any affected vertex that actually lives on another computer, send a tiny "vertex X is affected" message.

Step C – Fix distances locally (OpenMP loop)

- While there's an affected list on this computer:
-
- Many cores in parallel relax edges of those vertices.
-
- Better path to local neighbour? → put that neighbour on the next local list.
-
- Better path to remote neighbour? → queue a message for its owner (MPI Send Recv Used here).Step D



Step D – Swap those improvements (MPI)

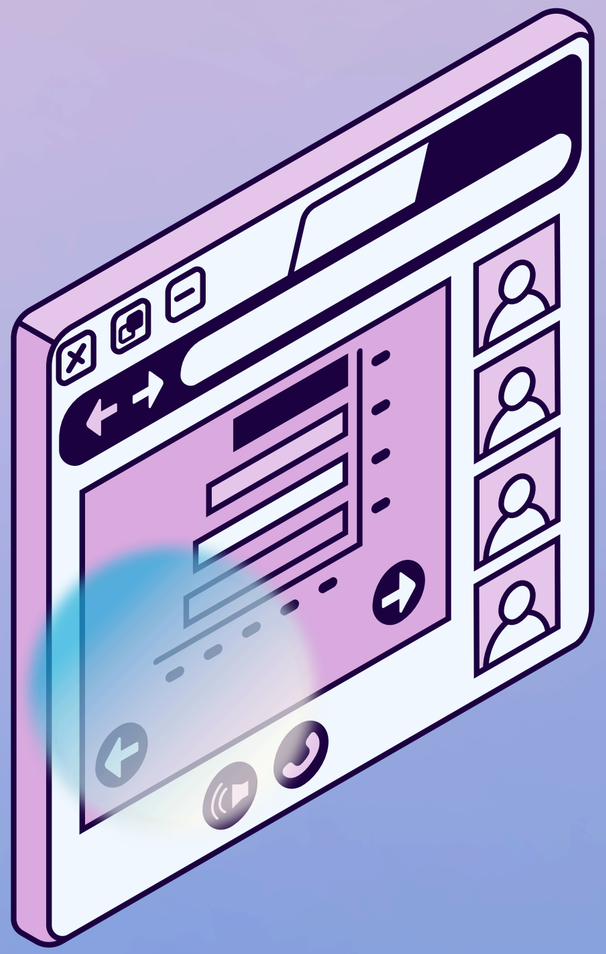
- Ship out queued updates; receive others.

Incoming updates adjust ghost-vertex distances and may add new locals to the list.

Step E – Are we done? (MPI All-reduce)

- All computers say whether their list is now empty.
- If everyone is empty ⇒ finished.
- Otherwise, go back to Step C and keep relaxing ie updating shorter paths

METIS splits once, OpenMP drives the work inside each node, and MPI only passes the “what changed” nuggets across nodes until no more improvements are found.



Algorithm 4. Asynchronous Update of SSSP

Input: $G(V, E)$, T, $Dist$, $Parent$, source vertex s, $\Delta E(Del_k, Ins_k)$
Output: Updated SSSP Tree T_k

```
1:  Function AsynchronousUpdating (ΔE, G, T):
2:    Set Level of Asynchrony to A.
3:    Change ← True
4:    while Change do
5:      Change ← False
6:      pragma omp parallel schedule (dynamic)
7:      for v ∈ V do
8:        if Affected_Del[v] = True then
9:          Initialize a queue Q
10:         Push v to Q
11:         Level ← 0
12:         while Q is not empty do
13:           Pop x from top of Q
14:           for c where c is child of x in T do
15:             Mark c as affected and change distance to
                infinite
16:             Change ← True
17:             Level ← Level + 1
18:             if Level ≤ A then
19:               Push c to Q
20:   Change ← True
21:   while Change do
22:     Change ← False
23:     pragma omp parallel schedule (dynamic)
24:     for v ∈ V do
25:       if Affected[v] == True then
26:         Affected[v] ← false
27:         Initialize a queue Q
28:         Push v to Q
29:         Level ← 0
30:         while Q is not empty do
31:           Pop x from top of Q
32:           for n where n is neighbor of x in G do
33:             Level ← Level + 1
34:             if Dist[x] > Dist[n]+W(x, n) then
35:             Change ← True
36:             Dist[x] = Dist[n]+W(x, n)
37:             Parent[x] = n
38:             Affected[x] ← True
39:             if Level ≤ A then
40:               Push x to Q
41:           if Dist[n] > Dist[x]+W(n, x) then
42:             Change ← True
43:             Dist[n]=Dist[n]+W(n, x)
44:             Parent[n] = x
45:             Affected[n] ← True
46:             if Level ≤ A then
47:               Push n to Q
```

- Udates are handled in batches
- Dynamic schedueling is used to balance load
- Uses load buffer per thread

- In parallel loop mark verticies affected by deletion or insertion
- Iterate until no affacted vertices remains

Deletion: disconnect affected vertices and mark subtree

- Insertion : if a better path is found update distance and mark children
- Each update round is done in parallel avoiding locks where possible

Conclusion

- The framework is scalable and adaptable across hardware:

- OpenMP for multi-core CPUs.
- CUDA-based GPU implementation (VMFB) for highly parallel execution.
- MPI-based distributed version for massive graphs across multiple machines.
- The asynchronous approach reduces synchronization overhead and enhances performance significantly.

- Parallelizing SSSP updates enables efficient handling of real-time changes in massive graphs — making it ideal for dynamic networks like social graphs, traffic systems, and communication networks.

Thank You!