MST in Incremental Graphs Through Tree Contractions

Akanksha Dwivedi, Sameer Sharma, Dip Sankar Banerjee
Department of Computer Science and Engineering, Indian Institute of Technology Jodhpur, India.
{P23CS0002, sharma.131, dipsankarb}@iitj.ac.in

***Abstract*—Dynamic graphs, characterized by rapid changes in their topological structure through adding or deleting edges or vertices, pose significant challenges for algorithm design. This paper presents a parallel algorithm for dynamic graphs in a batched setting. We employ a popular tree contraction mechanism to create a hierarchical representation of the input graph that allows the identification of localized areas that further enable maintaining a critical graph primitive such as the minimum spanning tree (MST) without requiring re-computation from scratch. We perform experiments to demonstrate the application of our algorithm in real-world graphs where batch-dynamic algorithms on large trees are essential for incremental updates. We show experimental validations on GPUs where our proposed technique can provide up to 3.43x speedup over equivalent parallel implementations on shared memory CPUs. Additionally, our methods can provide up to 4.23x speedup over conventional parallel computation from scratch.**

I. Introduction

Minimum Spanning Trees (MSTs) are fundamental graph properties with extensive applications across various domains. In the context of big data, MSTs play a crucial role in optimizing networks by identifying the most influential vertices, which is particularly beneficial for applications involving machine learning techniques. Dynamic MSTs, which adapt to real-time updates, are essential for applications requiring instantaneous processing and analysis, such as real-time traffic management and dynamic social network monitoring [1]. Traditional paradigms require computations that do not scale well with very high input sizes for computing and maintaining MSTs in real-world graphs. Modern big data-centric applications are increasingly dependent on "streaming" data or infinite data, which is difficult to manage in a limited main memory size. Conventional methods in such settings require re-computation from scratch or the use of distributed computers, which are difficult to implement and do not fully exploit the computing power available in modern high-performance platforms, specifically ones equipped with graphics processing units (GPU). It is imperative to investigate *"dynamic"* methods that can yield equivalent results on such settings without requiring a fresh re-computation on single compute vertices that exploits available processing capabilities to its fullest.

Towards the development of dynamic approaches, we specifically focus on the "batch dynamic" setting [1]–[3] where a set of updates can be assumed to be the maximum information that can be accommodated on a shared-memory buffer. Once the buffer is filled, the updates can be applied concurrently using a parallel algorithm, hence providing a higher scope of data parallelism. However, concurrent updates of a batch present their own unique challenges. Challenges such as identification of localized areas of effect, cascading impact of a single update, and consolidation of the results in the affected regions. For example, while maintaining an MST, when adding a new set of edges, it is necessary to maintain the minimum weight, ensure no cycles are formed, and reduce the amount of work involved in every parallel thread.

In this work, we solve the above mentioned challenges using graph contraction, which helps localize the affected areas during an update. This is followed by parallel path queries that can help identify the formed cycles and ensure a minimum weight tree. In graph contraction one of the major computational steps is in identifying sub-trees within the graph that can be collapsed into super-vertices. An existing solution in the sequential setting proposed by Acar et al. [4], shows the use of a novel *rake and compress* (RC) tree that supports the parallel implementation of the tree contraction [5]. Specifically, the rake operation identifies and compresses the vertices that are either leaf or have only one neighbor. This is followed by the compress step, where all the eligible 2-degree vertices are compressed. These two operations, when performed in an iterative manner, lead to the generation of more 1 and 2-degree vertices. The algorithm terminates when the tree is left with only one single super-vertex. Both the rake and the compress operations can be done in a data-parallel manner, given that raking essentially is a marking operation without the requirement of any writes. Compressing, while dependent, can be done in parallel with some optimizations that reduce redundant computations (discussed in Section IV-A). In this paper, we propose a parallel technique to compute an RC tree, which is used as a sub-routine in the batch update algorithm for quick incorporation of new updates into an existing MST.

Once an existing MST is contracted into the RC tree, it is now easy to identify the areas of the MST that will get affected due to the application of the batch updates. On concurrent application of all the edges, we can quickly identify the sub-trees that need updating, perform the update, and quickly re-compute the RC-tree, which can be done in a very fast manner on a massively multi-threaded implementation.

A. Motivation and Contribution

As discussed earlier, a major challenge in the dynamic maintenance of MSTs is in the identification of affected regions followed by ensuring the maintenance of the fundamental properties of an MST. While simple, designing a parallel massively parallel implementation for RC tree has specific set of challenges that can finally yield a scalable algorithm. In this work we propose techniques to handle only insertions in the graph and not deletions of batches that contains both insertions and deletions. So, in that sense our contribution can be classified as a "semi-dynamic" rather than "fully-dynamic".

Our concrete contributions in this work can be listed as follows:

1) We introduce novel methods for parallel computation of an RC tree. The RC tree further provides increased scope for extracting data parallelism.
2) We propose a batch-dynamic parallel algorithm to handle incremental updates of MST using the RC tree.
3) We conduct experiments on large graphs ranging up to 122 million edges using both a shared memory GPU and a CPU. From extensive experimentation, we can observe that our method can provide up to 3.43x speedups over a parallel CPU implementation. In effect, our method can insert a batch of size 122,000 edges in 0.05 milliseconds to maintain the MST.

II. Related Work

Parallel tree contraction on PRAM machines was proposed by Miller et al. [6], where the authors showed an O(log(n)) stable algorithm for insertions and deletions using a bottom-up approach. The bottom-up approach is essentially meant to deal with modifications locally and leads to the use of data structures that are easy to implement and also have polylogarithmic parallel solutions. This idea was expanded upon by Acar et al. in [4], which proposed techniques for the automatic generation of dynamic algorithms for trees from static programs. The core idea was to make the idea of [6] more generic through non-dependence on historical information or requirement of any rooted computations. This was expanded upon for a more generic solution proposed by the same team in [7] where the authors showed how change propagation can be exploited for dynamic tree updates specifically in problems related to path queries, sub-tree queries, search, and MSTs. The work in [4] first proposed the idea of RC trees which was investigated in detail for several applications in the thesis of Anderson [8].

Batch dynamic algorithms for several applications in the areas of spanning forests and clustering was proposed in [9] where the authors showed specialized data structures for maintaining maximal spanning forests in the decremental setting and also use the same for hierarchical agglomerative clustering. For maintaining dynamic trees, the work in [4] was extended in [1] where the authors showed algorithms for batch dynamic updates for several tree problems such as flow, MST, and minimum-cuts. A similar work on batch dynamic algorithm for graph connectivity was proposed in [2]. Maintaining information in fully dynamic trees with the help of top-tree data structures was shown by Alstrup.et.al [10]. This work addressed challenges in efficiently handling updates and queries in dynamically changing tree data, enhancing the scalability and versatility of data structures crucial for various computational tasks.

Luo et al. [11] address the efficient maintenance of minimum spanning trees (MSTs) in dynamic weighted undirected graphs. Their algorithm focuses on updating the MST efficiently when the graph structure undergoes changes, minimizing the need for full recalculations. This approach preserves parts of the original tree that remain valid after updates, thereby reducing computational overhead. The authors provide a formal proof of correctness and analyze the algorithm's time complexity, demonstrating that it compares favorably with Kruskal's algorithm in typical scenarios. Prokopenko et.al [12] proposed GPU solutions for the Euclidean minimum spanning trees. They propose a novel single-tree Borůvka-based algorithm optimized for GPU architectures, leveraging an efficient nearest neighbor approach and minimizing distance calculations by optimizing subtree transversal.

In all the existing works, we have not observed any specific work showing the parallel implementation of tree contraction-based maintenance of MSTs on dynamic graphs. Our lasting takeaway from this work is an algorithm that leverages locality-aware updates for maintaining MSTs on massively parallel platforms.

III. Preliminaries

The Minimum Spanning Tree (MST) problem is a fundamental problem in graph theory, where the goal is to find a subset of edges in a weighted, connected graph that connects all the vertices without any cycles and with the minimum possible total edge weight. Classical algorithms for solving the MST problem include Prim's, Kruskal's, and Borůvka's algorithms.

A. Borůvka's Algorithm

Boruvka's Algorithm [13], also known as Sollin's Algorithm, is an algorithm for finding a minimum spanning tree in a graph for which all edge weights are distinct or a minimum spanning forest in the case of a graph that is not connected. It proceeds by successively adding the smallest weight edge that connects any two trees in the forest until the forest forms a single tree, which is a minimum spanning tree. The algorithm begins by treating each vertex as a separate component. It then iterates through all the edges, and for each component, the edge with the smallest weight is selected to connect the component to a different one. These selected edges are added to the minimum spanning tree and the connected components are merged. This process is repeated until only one component,

the minimum spanning tree, remains. The time complexity of Boruvka's Algorithm is $O(E \log V)$, assuming the use of a union-find data structure with union by rank and path compression. This is because the algorithm performs $O(E)$ operations in each phase to find the minimum weight edge for each component and $O(V)$ operations to perform the unions. There are $O(\log V)$ phases, so the total time complexity is $O(E \log V)$. The space complexity of Boruvka's Algorithm is $O(V + E)$, as it needs to maintain a list of edges and a disjoint-set data structure to keep track of which vertices are in which components.

In our implementations, we create the initial MST using Boruvka's algorithm and again in the subsequent steps where a re-computation is required. Since Boruvka's algorithm treats each of the vertices as a separate component to begin with, it is inherently parallel. We implement a parallel version of Boruvka's [14] for the scratch computations on CPU and GPU. We use a wait-free union-find data structure [15] that maintains the parent and rank information for each vertex and subsequently helps in identifying the valid component for each edge contraction.

B. Rake and Compress Trees

A rake and compress (RC) tree [4] builds upon an earlier technique of tree contraction on directed graphs proposed by by Miller and Reif [6]. The idea for EC trees is to handle dynamic updates in a graph via change propagation. To use RC trees, the authors use annotations specific to an application in a manner so that traversals on an RC tree would suffice to answer dynamic queries. For handling dynamic changes, only specific re-computations are required on the part of the tree that is affected by the change. The RC tree is an O(n) space data-structure which is an improved implementation over equivalent history-dependent implementations which consumed O(nlogn) space [7]. The combination of Boruvka's phased approach and the dynamic, parallel execution of RC trees results in an MST algorithm that is highly effective for large-scale graph processing, offering substantial speedups over traditional methods [16].

IV. Methodology

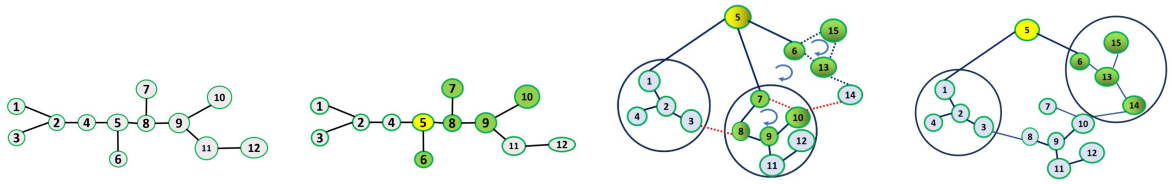
Algorithm 1 Dynamic MST using RC

```
1: procedure DynamicMST(Input G, Batches B)
2:     Input graph G
3:     M=Boruvka(G)
4:     R=Parallel_RC(M)
5:     for b_i ∈ B do
6:         BatchInsert(R, M, B)
7:         Parallel_RC()
```

The broad set of steps for managing the dynamic updates is shown in Algorithm 1 and Figure 1. The initial pre-processing steps involve computing the MST on an existing graph using Boruvka's algorithm. This is followed by creating the contracted tree using the rake and compressed (RC) operations. This is done using one of two mechanisms (iterative and level-wise). Then, we iteratively include the batches into the existing MST by first identifying the vertex of affection from the RC tree, inserting the edge, and finally updating the RC tree with the new edge. Figure 1(a) illustrates the initial graph, representing the starting point before any transformations. In Figure 1(b), the graph is shown after the initial computation of the RC tree, where the two shades of green denote the two clusters in the RC tree. In Figure 1(c), we can observe the batch insertion where the broken lines denote the new edges. After removing the cycles, the RC tree is again re-computed, shown in Figure 1(d). Here, The remaining vertices 7, 8, 9,

Algorithm 2 Parallel RC operations in the CSR augmented RC Tree

```
1: procedure Parallel_RC(Input G)
2:     Input: MST tree M (input graph)
3:     Output: RC tree R
4:     arr[0] ← G,round = 1,update =true.
5:     while update do
6:         if |E| = |V| then
7:             marked=mark_vertices_iterative(CSR_{t-1})
8:         else
9:             marked=mark_vertices_level(CSR_{t-1})
10:        CSR_t=rake(CSR_{t-1}, marked)
11:        c=mark_compressible_vertices(CSR_t)
12:        compress(c, CSR_t)
13:        if |c| = 1 then
14:            update=false
15:        CSR_t = CSR_{t-1}
16: procedure MARK_VERTICES(CSR_{t-1})
17:     for each vertex v in parallel do
18:         if v is not raked and has 1 neighbor then
19:             mark v as rakeable
20: procedure Rake(CSR_{t-1}, marked)
21:     for each vertex v ∈ CSR_{t-1} in parallel do
22:         if u is the only neighbor and is unmarked then
23:             v is raked
24: procedure MARK_COMPRESSIBLE_ITERATIVE(CSR_t)
    ▷ Find an independent set
25:     for all v ∈ CSR_t in parallel do
26:         if v has only two alive neighbors then
27:             mark v
28: procedure MARK_COMPRESSIBLE_LEVEL(CSR_t)  ▷
    Run BFS on CSR_t
29:     for l ≤ H(CSR_t) do
30:         for all v at height l in parallel do
31:             if v has only two alive neighbors then
32:                 mark v
33: procedure COMPRESS(c, CSR_t)
34:     for each vertex v ∈ CSR_t in parallel do
35:         if v is marked then
36:             Compress v
```



(a) Initial tree (b) Initial RC tree. The green ones denote the collapsed vertices. (c) Batch insertions (d) Re-computed RC Tree after the batch iteration

Fig. 1: The overall steps where we start with an initial MST followed by the initial computation of the RC tree. Following that the batch insertions are started where the edges (3,8), (7,10), (10,14), (6,13), (13,15), and (6,15) are inserted introducing the new vertices 13, 14 and 15. Post the batch update, the cycles indicated by blue curved lines are removed and the RC tree is re-computed

Algorithm 3 Batch update in an existing MST

procedure BATCHINSERT(R, M, B_i)
  for $(u, v) \in B_i$ **do in parallel**
    Find $n_u \in R$ where $u \in n_u$
    Find $n_v \in R$ where $v \in n_v$
    if $n_v = n_u$ **then** ▷ Insert (u, v) in $n_u \in R$
      Add (u, v) to R
      needed = DetectCycle(n_u)
      if needed **then**
        BreakCycle(n_u)
    if $n_u \neq n_v$ **then**
      l = LCA(n_u, n_v)
      Insert (u, v) in l
      needed = DetectCycle(n_u)
      if needed **then**
        BreakCycle(n_u)
    if $u \notin R$ or $v \notin R$ **then**
      Add u or v to M
  Return Updated MST and RC Tree rcT
procedure DETECTCYCLE(n_u)
  Mark root(n_u)
  for vertices $k_i \in adj(n_u)$ **do**
    if k_i not marked **then**
      DFS(adj(k_i))
      if not marked return TRUE
  Return FALSE
procedure BREAKCYCLE(n_u)
  Find e = EulerTour(n_u)
  for $w_i \in e_i$ **do**
    Find edge t with the maximum weight
  n_u.delete(t)

10, 11, and 12 are excluded from the RC tree since the other clusters store their properties.

A. Parallel tree contraction

The parallel graph contraction using RC trees is shown in Algorithm 2. The rake operation identifies and compresses the vertices that are either leaves or have only one neighbor. This is followed by the compress step where all the eligible 2-degree vertices are compressed. These two operations, when performed in an iterative manner, lead to the generation of more 1 and 2-degree vertices. The algorithm terminates when the tree is left with only one single super-vertex. While raking is comparatively simpler to perform, compressing involves a significant amount of memory accesses for checking the eligibility of the vertices and then compressing. While theoretically efficient, a sequential implementation of a RC tree shows that it takes close to 90 seconds for a graph of 100 million vertices. This is not beneficial for use as a primitive in stream applications. In this work we propose a parallel implementation of RC trees which does not exist to the best of our knowledge. We propose a new data structure that can handle both rake and compress operations in a parallel setting along with fine-grained data parallelism.

For a parallel GPU implementation, we observe the key challenges in both rake and compress operations. As we observe, the parallel rake operation will require compressible vertices that are at the leaf level or have a single neighbor. This, when done in a concurrent fashion, may lead to two neighboring vertices getting marked and merged into each other, which will lead to an incorrect solution. Additionally, in the compress operation, we must not identify two eligible vertices which are neighbors of each other. However, when executing in parallel, this can be a possibility. Towards this, we propose a new technique via augmenting the base RC tree with compressed sparse row (CSR) representations and double buffering that allows parallel rake and compress operations. In Figure 1, we observe a recursive clustering of the tree resulting from tree contraction. Vertices that are raked or compressed are represented in darker shades. Figure 1(a) illustrates the initial tree, which is represented using CSR_{t-1} at time t-1. In the second step we find the clusters formed by one rake and compress operation. Now, these clusters will function as individual vertices. We will now again apply a rake and compressed operation, treating the clusters formed from the previous step as individual vertices. This is continued in a recursive manner to form the final clusters, as shown in Figure 1(b). We can observe that there is only one un-shaded

vertex remaining in the tree.

As we can see from Algorithm 2, in the initial phase, we consolidate rakeable and compressible vertices into their neighboring vertices, creating super vertices. In the subsequent phase, we iteratively apply this process to the newly formed super vertices and the remaining un-contracted vertices until only a solitary vertex remains. In each operation, we can observe that the marking of the eligible vertices for raking and the raking operation itself are data parallel. This step, in essence, creates an independent set. However, when handling dense graphs, we observe that the neighborhood size of any vertex can be potentially very high, leading to significant serialization when operated in a vertex parallel manner. To alleviate this, we design a *level wise* scheme where a simple BFS on M can provide a count of the number of adjacent vertices. This further allows us to create blocks of threads that are in the same number of the level-wise order and can now be scheduled by a GPU in an efficient manner, thus providing higher performance. However, the iterative method suffices for sparse graphs since the local neighborhoods are not very large. A level wise ordering is hence not significantly beneficial. We perform this check in Lines 5-9 of Algorithm 2 where we set a threshold to ensure that the *iterative* marking scheme is used for sparse graphs and the level wise for dense. The benefits obtained due to the level wise marking is evident from experimental details shown in Section V.

In the compression phase, when we mark the vertices that can be compressed, the challenge is to resolve the data dependencies via computation of a minimal independent set of the raked vertices (Lines 20-22 of Alg 2). This is followed by the compression step, which is again data parallel. In each of the steps, potential thread divergences are handled via the use of bit-masks that reduce divergence but not the work.

B. Batch Insertions

The next step is to perform the batch insertions, which are done in an edge parallel manner. We create a thread for every edge of the batch which then proceeds to update the MST concurrently. The primary goal of constructing the RC tree is to ensure that the amount of computations required for re-constructing the MST with the new updates are minimized. We achieve this via first detecting if the edge undergoing insertion belongs to some vertex n_x of the RC tree R. The ideal scenario where both the vertices are found in the same vertex of R provides the opportunity to simply add the incoming edge to R, and then proceed to checking for the MST properties. However, this may not be the case where we can also potentially have edges where the two vertices lies in different vertices of R. In that case, we insert the edge in the lowest common ancestor (LCA) vertex of the R, which is again processed to ensure that the properties are maintained. The reason for inserting the edge at the LCA vertices is to ensure that we can get the highest weighted edge. Because of the structure of the RC tree, the LCA vertex will hold the weight of the highest weighted edges. This helps efficiently maintain the MST properties during the insertion process.The other case that might arise is when the edge brings new vertices to the graph that did not earlier. This is a trivial case where we can be sure that the new vertices will not affect any of the existing MST, and the new edge can be directly added to it.

To ensure the MST properties, we need to validate that the new edges that are getting added to R are not further creating cycles and that the minimum property of the tree is maintained. Toward that, we use two auxiliary functions, DetectCylce() and BreakCycle(). In DetectCycle() we iterate over all the adjacent vertices from the root vertex of the sub-graph using a DFS. The function returns TRUE to report the presence of a cycle. In case a cycle is detected, the BreakCycle() function is called, which now removes the heaviest edge from the detected cycle. For this, an Euler tour [17] is computed on the cycle that allows us to traverse all the edges and maintain the edge with the heaviest weight. Once the tour yields the heaviest edge, the same is removed from the cycle. The use of Eulerian tour to order the non-tree edges of a spanning tree for dynamic operations has also been demonstrated in earlier works [18], [19].

V. Evaluation

TABLE I: Datasets

Sr. No.	Graph Name	Vertices $\|V\|$	Edges $\|E\|$	Type
G1	road_central [20]	14,081,816	33,866,826	Sparse
G2	road_usa [20]	23,947,347	57,708,624	Sparse
G3	delaunay_n21 [20]	2,097,152	12,582,816	Sparse
G4	asia_osm [20]	11,950,757	25,423,206	Sparse
G5	com-youtube [21]	1,157,827	2,987,624	Sparse
G6	delaunay_n22 [20]	4,194,304	25,165,738	Dense
G7	com-orkut [21]	3,072,441	117,185,083	Dense
G8	as-Skitter [21]	1,696,415	122,190,596	Dense
G9	soc-LiveJournal1 [21]	4,847,571	68,993,773	Dense
G10	hollywood-2009 [20]	1,139,905	113,891,327	Dense

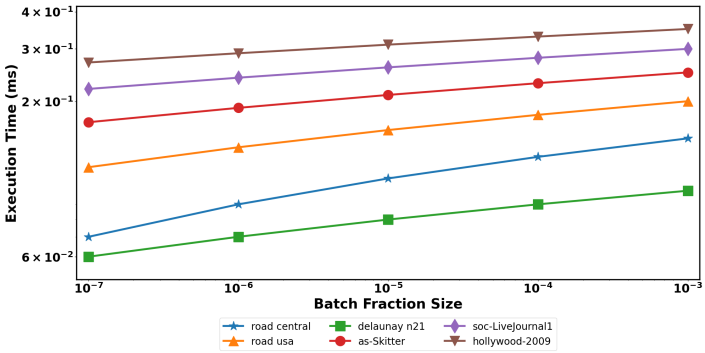
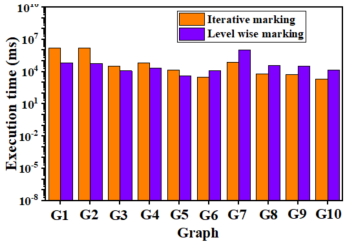
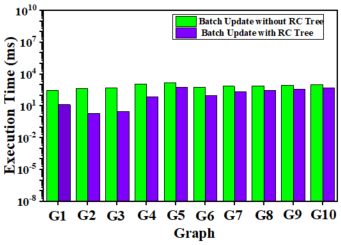


Fig. 2: Performance across different batch sizes

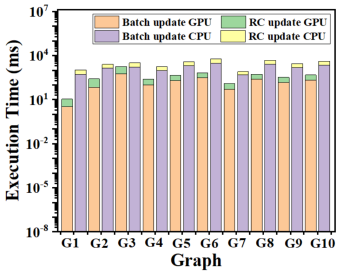
We conducted all experiments on a server with an NVIDIA A5000 GPU with 24 GB of global memory. It is hosted on a CPU chassis with two 32-core AMD EPYC processors. We compile all the programs with CUDA version 11.7 and GCC version 9.4 with OpenMP version 5.0. We use large graphs from the SNAP repository [21] and SuitSparse Collection [20], which are detailed in Table I. We classified the



(a) Runtime for RC tree creation for iterative and level wise methods.



(b) Runtime of batch updates time with or without RC tree



(c) Comparison of parallel CPU versus GPU with fixed batch fraction

Fig. 3: (a) Performance analysis of RC Tree in real world graphs, (b) Performance analysis of Batch updates time with or without RC tree, (c) Performance analysis of real graphs in parallel CPU versus GPU in small size of batch fraction

graphs as sparse or dense based on the vertex ($|V|$) and edge ($|E|$) counts. Traditionally, sparse graphs are characterized by $\mathcal{O}(n)$ edges, while dense graphs approach $\mathcal{O}(n^2)$ edges, where n is the number of vertices.All the programs are written in C++ and CUDA with the CSR representation done using STL vectors. For comparisons, we also perform an equivalent implementation on a parallel CPU using OpenMP directives. The parallel steps highlighted in our algorithms have been parallelized with equivalent OpenMP implicit directives, and a dynamic schedule has been maintained throughout. For experimentation with the batches, we create the batches by choosing the edges uniformly at random from the input edge list.

We first study the overall benefit of our batch update mechanism, which is shown in Figure 2, where we can observe the update times as a function of the batch sizes for the six largest graphs. We can observe a linear growth in the time to update the graph with varying batch sizes. For the largest batch of approximately 122K edges in the as-Skitter graph, we can observe that the minimum time taken is 0.090 msec and the maximum time of 0.370 msec. This is expected behavior since dense graphs generally require more computation time due to their higher edge density and connectivity. Overall, when compared to a parallel static method (fresh re-computation on a same-sized input), we can observe that, on average, our technique can provide 4.82x speedup for the largest batch and 3.40x speedup for the smallest batch.

The overall results from our experiments are shown in Figure 3. In Figure 3(a) we show the the efficacy of iterative marking versus level-wise methods for RC Tree updates across different types of graphs. For sparse graphs, iterative marking consistently shows better performance. On average, for sparse graphs, we can observe a speedup of 3.29x for the iterative marking technique over the dense graphs. Conversely, for dense graphs, we can observe a speedup of 3.20x for the level wise technique over the sparse graphs. Overall, the level-wise technique performs 3.25x better for the graphs we chose than the iterative method. The iterative technique capitalizes on the sparsity of the graph, efficiently handling fewer connections and minimizing unnecessary computations.

For the implementation of the level-wise technique, as we discussed earlier, we create blocks with only the number of adjacent vertices that are present for a vertex at a particular level. This configuration allows a lesser number of threads to be idle, which in turn allows the blocks to complete SIMD execution in a more efficient manner, thus allowing a more seamless swapping of the blocks in and out of the SMs.

In Figure 3(b), we can observe the overall execution times for the batch updates on a fixed batch size ($10^{-3}.E$) for all the graphs. We compare the runtimes achieved by the RC tree with the runtimes of dynamic MST updates that do not use an RC tree. This is analogous to the case that for every edge that is inserted, there are no sub-graphs that can be identified for the MST update, and hence, a re-computation is required on the entire graph. As we can observe, the RC Tree update mechanism outperforms the one without by 4.33x. This is clearly due to the lessened computational work that the RC tree enables by identifying the optimal sub-graphs affected by the update.

In Figure 3(c), we compare the performance of equivalent implementations of MST on parallel CPU and with that of the GPU. As we mentioned earlier, to the best of our knowledge, no parallel batch dynamic implementation of MSTs exists. As we can observe, the GPU implementation provides a speedup of 3.43x over the parallel CPU implementation and, in turn, a net 4.23x speedup over a parallel MST implementation using Boruvka from scratch.

VI. Conclusion and Future Work

In this work, we present a technique for online maintenance of MST in incremental graphs. We show that with the ability to localize the affected areas due to a batch update through tree contraction, good speedups of up to 3.5x can be obtained over equivalent parallel implementations and is clearly beneficial over static re-computations. In the near future, we wish to extend the work to the fully-dynamic case where we shall explore techniques for handling both insertions and deletions in a concurrent fashion.

VII. Acknowledgement

The work is partially supported by a grant from the Science & Engineering Research Board (SERB) of the Department of Science and Technology (DST), India, vide Project No: CRG/2023/005225.

References

[1] U. A. Acar, D. Anderson, G. E. Blelloch, L. Dhulipala, and S. Westrick, "Parallel batch-dynamic trees via change propagation," in *European Symposium on Algorithms (ESA)*, 2020.

[2] U. A. Acar, D. Anderson, G. E. Blelloch, and L. Dhulipala, "Parallel batch-dynamic graph connectivity," in *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019.

[3] C. A. Haryan, G. Ramakrishna, K. Kothapalli, and D. S. Banerjee, "Shared-memory parallel algorithms for fully dynamic maintenance of 2-connected components," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 1195–1205.

[4] U. A. Acar, G. E. Blelloch, and J. L. Vittes, "An experimental analysis of change propagation in dynamic trees," in *Algorithm Engineering and Experiments (ALENEX)*, 2005.

[5] P. Gawrychowski, S. Mozes, and O. Weimann, "Minimum cut in o(m $\log^2$ n) time," in *Intl. Colloq. on Automata, Languages and Programming (ICALP)*, 2020.

[6] G. L. Miller and J. H. Reif, "Parallel tree contraction and its application," in *26th Symposium on Foundations of Computer Science*. Portland, Oregon: IEEE, October 1985, pp. 478–489.

[7] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and S. L. M. Woo, "Dynamizing static algorithms, with applications to dynamic trees and history independence," in *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*. SIAM, 2004, pp. 531–540.

[8] D. Anderson and G. E. Blelloch, "Deterministic and low-span work-efficient parallel batch-dynamic trees," *ACM Transactions on Algorithms (TALG)*, vol. 16, no. 2, pp. 1–32, 2020.

[9] T. Tseng, L. Dhulipala, and J. Shun, "Parallel batch-dynamic minimum spanning forest and the efficiency of dynamic agglomerative graph clustering," *Journal of Graph Algorithms and Applications*, 2023.

[10] S. Alstrup, J. Holm, K. D. Lichtenberg, and M. Thorup, "Maintaining information in fully dynamic trees with top trees," *ACM Trans. Algorithms (TALG)*, vol. 1, no. 2, pp. 243–264, 2005.

[11] M. Luo, H. Qin, X. Wu, C. Xiong, D. Xia, and Y. Ke, "Efficient maintenance of minimum spanning trees in dynamic weighted undirected graphs," *Mathematics*, vol. 12, no. 7, p. 1021, 2024.

[12] A. Prokopenko, P. Sao, and D. Lebrun-Grandié, "A single-tree algorithm to compute the euclidean minimum spanning tree on gpus," in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–10.

[13] O. Borůvka, *O jistém problému minimálním (About a certain minimal problem)*, ser. Práce Moravské přírodovědecké společnosti. Mor. přírodovědecká společnost, 1926.

[14] S. Chung and A. Condon, "Parallel implementation of bouvka's minimum spanning tree algorithm," in *Proceedings of International Conference on Parallel Processing*, 1996, pp. 302–308.

[15] R. J. Anderson and H. Woll, "Wait-free parallel algorithms for the union-find problem," in *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, ser. STOC '91. New York, NY, USA: Association for Computing Machinery, 1991.

[16] D. Anderson and G. E. Blelloch, "Deterministic and low-span work-efficient parallel batch-dynamic trees," in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, 2024, pp. 247–258.

[17] R. E. Tarjan, "Dynamic trees as search trees via euler tours, applied to the network simplex algorithm," *Math. Program.*, vol. 78, no. 2, p. 169–177, aug 1997.

[18] M. R. Henzinger and V. King, "Randomized fully dynamic graph algorithms with polylogarithmic time per operation," *J. ACM*, vol. 46, no. 4, p. 502–516, jul 1999.

[19] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia, "Complexity models for incremental computation," *Theoretical Computer Science*, vol. 130, no. 1, pp. 203–236, 1994.

[20] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011.

[21] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.