Shared-Memory Parallel Algorithms for Community Detection in Dynamic Graphs (Extended Report)

Anonymous Author(s)

ABSTRACT

Community detection is the problem of recognizing natural divisions in networks. A relevant challenge in this problem is to find communities on rapidly evolving graphs. This calls for designing dynamic algorithms that reuse earlier output, only process vertices likely to change their community, extract parallelism by processing a batch of updates, and have low overhead.

This paper addresses the design of a high-speed community detection algorithm in the batch dynamic setting. First, our optimized parallel implementations of the *Louvain* and *LPA* algorithms is presented. These implementations identify communities in 6.2 seconds and 2.7 seconds, respectively, on a single 64-core CPU, when processing an undirected web graph with 1.9 billion edges. Next, our *Dynamic Frontier* approach is discussed. Given a batch update of edge deletion and insertions, this approach incrementally identifies an approximate set of affected vertices in the graph is with minimal overhead. We apply this approach to both *Louvain*, a high quality, and *Label Propagation Algorithm (LPA)*, a high speed static community detection algorithm. Our approach achieves a mean speedup of 1.5× and 10.0× when applied to *Louvain* and *LPA* respectively, compared to the best of two other dynamic approaches. Finally, we show how to combine *Louvain* and *LPA* with the *Dynamic Frontier* approach to arrive at a novel hybrid algorithm. This algorithm produces high-quality communities while providing a speedup of 7.5× on top of *Dynamic Frontier* based *Louvain*.

CCS CONCEPTS

• **Theory of computation** → **Parallel algorithms**; **Shared memory algorithms**; **Graph algorithms analysis**; **Dynamic graph algorithms**.

KEYWORDS

Dynamic Graphs, Community Detection, Parallel Algorithms, Dynamic Frontier Louvain algorithm, Label Propagation Algorithm (LPA), Hybrid Louvain-LPA

1 INTRODUCTION

Graphs present a powerful abstraction for representing data and their internal relationships. The collection of data and the relationships among them, represented as graphs, have reached unmatched levels in recent years. These graphs are usually immense in scale, stemming from applications such as machine learning and social networks. *Community detection* is an NP-hard problem in graph analytics with numerous applications in domains such as drug discovery, protein annotation, disease prediction, topic discovery, inferring land use, and criminal identification. Finding communities in graphs is a well-studied problem, where the primary objective is to identify groups of vertices that exhibit dense internal connections but sparse connections with the rest of the graph. These communities are intrinsic when identified based on network topology alone, without external attributes, and they are disjoint when each vertex belongs to only one community [22].

One of the difficulties in the community detection problem is the lack of apriori knowledge on the number and size distribution of communities. Two popular heuristic-based approaches to community detection are the *Louvain* method [8] and the *Label Propagation Algorithm (LPA)* [48]. Current work on community detection uses the modularity metric proposed by Newman et al. [43] to measure the quality of the communities identified. Several recent studies show how to implement *Louvain* method and *LPA* on modern architectures such as multi-core CPUs [16], GPUs [10], CPU-GPU hybrid platforms [6], distributed platforms [20], and the like. While the *Louvain* algorithm obtains high-quality communities, we find it to be 2.3 − 14× slower than *LPA* (which obtains communities of lower quality by 3.0 − 30%).

With the data deluge and ever-changing application requirements, newer challenges are emerging. Many real-world graphs evolve with the insertion/deletion of edges/vertices. For efficiency reasons, one needs algorithms that update the results without recomputing from scratch. Such algorithms are known as *dynamic algorithms*. Examples of parallel dynamic algorithms include those for dynamic graph coloring [5, 74], maintaining shortest routes [29, 78], and updating centrality scores [50, 61].

However, parallel algorithms for dynamic graphs are challenging. One popular approach to extract parallelism is the *batch dynamic* setting, where one processes a batch of updates simultaneously. A batch update allows extraction of parallelism and minimizes the work performed compared to the setting where the updates are applied individually. Parallel algorithms have to be designed appropriately to benefit from such latent parallelism.

There is a pressing need for efficient parallel algorithms for community detection on large dynamic graphs. The goal of *dynamic community detection* is to obtain high-quality communities while minimizing computation time. One does this by choosing a suitable algorithm, reusing old community membership of vertices, and processing only a small subset of the graph likely to be affected by changes. Note that if the subset of the graph identified as *affected* is too small, we may end up with inaccurate communities, and if the subset is too large, we incur a significant computation time. Hence, one should look to identify the appropriate set of affected vertices. In addition, determining the vertices to be processed should have low overhead [49]. Existing approaches either process all vertices (such as *Naive-dynamic*), over/underestimate the set of affected vertices, and/or have high overhead (such as Δ-*screening* [75]). The algorithm of Zarayeneh et al. [75] suffers from identifying far too many vertices as affected, thereby requiring a large amount of work to identify the new communities.

One recent approach for dynamic community detection in the parallel setting is the work of Reidy and Bader [52]. While being a parallel batch dynamic algorithm, the work of Reidy and Bader [52] has a few limitations. They do not consider cascading changes to community labels on an update or study the quality of obtained communities. We summarize the state-of-the-art in Table 1. From Table 1, we note that there is a need for efficient parallel algorithms for community detection on dynamic graphs.

1.1 Our Contributions

This paper addresses the design of a high-speed community detection algorithm in the batch dynamic setting (where multiple edge updates are processed simultaneously), and identifies communities of high quality while addressing the concerns mentioned above.

We start by discussing our optimized parallel implementations of the *Louvain* and *LPA* algorithms, determining suitable parameter settings and optimizing the original algorithm through experimentation with a variety of techniques. The combined optimizations significantly enhance the performance of the OpenMP-based *Louvain* and *LPA*, achieving completion times of 6.2 seconds and 2.7 seconds, respectively, on a single 64-core CPU, when processing an undirected web graph with 1.9 billion edges. We have experimented with COPRA [22], SLPA [69], and LabelRank [68], but found LPA [48] to be the most performant, while yielding communities of equivalent quality.

We then discuss our *Dynamic Frontier* approach that incrementally identifies an approximate set of affected vertices in the graph, given a batch of edge deletions and insertions, with low runtime overhead. We show how to combine our *Dynamic Frontier* approach to existing static algorithms for community detection such as *Louvain* and *LPA*. We perform the parallel executions and experimentations on a shared-memory multi-core CPU platform.

We compare *Dynamic Frontier* with two other dynamic approaches, the *Naive-dynamic* approach, and the *Dynamic Δ-screening* approach. On a collection of 12 graphs from four different classes, our experiments indicate that the *Dynamic Frontier* approach coupled with *Louvain* has a mean improved performance of 1.5× compared to *Naive-dynamic Louvain*, while obtaining communities of the same quality. On the same collection of graphs, the *Dynamic Frontier* approach coupled with *LPA* has a mean improved performance of 10.0× compared to *Naive-dynamic LPA*, again while obtaining communities of equivalent quality. The work presented by Zarayeneh et al. [75] demonstrates improved performance of *Δ-screening* compared to *Dynamo* [79] and *Batch* [11]. As the *Dynamic Frontier* approach outperforms *Δ-screening*, we expect similar gains compared to *Dynamo* and *Batch*.

Finally, we discuss our *Dynamic Frontier* based *Hybrid Louvain-LPA* that combines *Static Louvain* and *Dynamic Frontier* based *LPA* into a hybrid dynamic algorithm. This allows it to leverage the strengths of both algorithms while overcoming their individual limitations. It offers a 7.5× speedup over *Dynamic Frontier* based *Louvain*, while yielding communities of high quality. To our knowledge, no prior work has explored a similar hybrid approach. [1]

2 PRELIMINARIES

Let $G(V, E, w)$ be an undirected graph, with V as the set of vertices, E as the set of edges, and $w_{ij} = w_{ji}$ a positive weight associated with each edge in the graph. If the graph is unweighted, we assume each edge to be associated with unit weight ($w_{ij} = 1$). Further, we denote the neighbors of each vertex i as $J_i = \{j \mid (i, j) \in E\}$, the weighted degree of each vertex i as $K_i = \sum_{j \in J_i} w_{ij}$, the total number of vertices in the graph as $N = |V|$, the total number of edges in the graph as $M = |E|$, and the sum of edge weights in the undirected graph as $m = \sum_{i,j \in V} w_{ij}/2$.

	[75]	[79]	[11]	[68]	[52]	**This Paper**
Parallel	×	×	×	×	✓	✓
Fully Dynamic	✓	✓	✓	✓	✓	✓
Batch update	✓	✓	✓	✓	✓	✓

Table 1: Comparison of the features of various algorithms for community detection.

2.1 Community detection

Disjoint community detection is the process of arriving at a community membership mapping, $C : V \rightarrow \Gamma$, which maps each vertex $i \in V$ to a community-id $c \in \Gamma$, where Γ is the set of community-ids. We denote the vertices of a community $c \in \Gamma$ as V_c. We denote the community that a vertex i belongs to as C_i. Further, we denote the neighbors of vertex i belonging to a community c as $J_{i \rightarrow c} = \{j \mid j \in J_i \; and \; C_j = c\}$, the sum of those edge weights as $K_{i \rightarrow c} = \{w_{ij} \mid j \in J_{i \rightarrow c}\}$, the sum of weights of edges within a community c as $\sigma_c = \sum_{(i,j) \in E \; and \; C_i = C_j = c} w_{ij}$, and the total edge weight of a community c as $\Sigma_c = \sum_{(i,j) \in E \text{ and } C_i = c} w_{ij}$ [34, 75].

2.2 Modularity

Modularity is a fitness metric that is used to evaluate the quality of communities obtained by community detection algorithms (as they are heuristic based). It is calculated as the difference between the fraction of edges within communities and the expected fraction of edges if the edges were distributed randomly. It lies in the range $[-0.5, 1]$ (higher is better) [9]. Optimizing this function theoretically leads to the best possible grouping [42, 64].

We can calculate the modularity Q of obtained communities using Equation 1, where δ is the Kronecker delta function ($\delta(x, y) = 1$ if $x = y$, 0 otherwise). The *delta modularity* of moving a vertex i from community d to community c, denoted as $\Delta Q_{i:d \rightarrow c}$, can be calculated using Equation 2.

$$Q = \frac{1}{2m} \sum_{(i,j) \in E} \left[w_{ij} - \frac{K_i K_j}{2m} \right] \delta(C_i, C_j) = \sum_{c \in \Gamma} \left[\frac{\sigma_c}{2m} - \left(\frac{\Sigma_c}{2m} \right)^2 \right] \quad (1)$$

$$\Delta Q_{i:d \rightarrow c} = \frac{1}{m}(K_{i \rightarrow c} - K_{i \rightarrow d}) - \frac{K_i}{2m^2}(K_i + \Sigma_c - \Sigma_d) \quad (2)$$

2.3 Algorithms for Static Graphs

2.3.1 Louvain algorithm [8]. *Louvain* is a greedy, modularity optimization based agglomerative algorithm that can find high quality communities within a graph, with a time complexity of $O(KM)$ (where K is the number of iterations performed across all passes), and a space complexity of $O(N + M)$ [33]. It consists of two phases:

[1] For reproducibility, our source code is at https://bit.ly/hipc23-01-artifact

the *local-moving phase*, where each vertex i greedily decides to move to the community of one of its neighbors $j \in J_i$ that gives the greatest increase in modularity $\Delta Q_{i:C_i \rightarrow C_j}$ (using Equation 2), and the *aggregation phase*, where all the vertices in a community are collapsed into a single super-vertex. These two phases make up one pass, which repeats until there is no further increase in modularity. As a result, we have a hierarchy of community memberships for each vertex as a dendrogram. The top-level hierarchy is the final result of the algorithm [34].

2.3.2 Label Propagation Algorithm (LPA) [48]. *LPA* is a popular diffusion-based method for finding communities, that is simpler, faster, and more scalable (due to its lower memory footprint), compared to *Louvain*. In *LPA*, every vertex i is initialized with a unique label (community id) C_i and at every step, each vertex adopts the label with the most interconnecting weight, as shown in Equation 3. Through this iterative process, densely connected groups of vertices form a consensus on a unique label to form communities. The algorithm converges when at least $1 - \tau$ fraction of vertices do not change their community membership (where τ is the tolerance parameter). It has a time complexity of $O(KM)$ and a space complexity of $O(N + M)$, where K is the number of iterations performed.

$$C_i = \underset{c \,\in\, \Gamma}{\arg\max} \sum_{j \in J_i \,\mid\, C_j = c} w_{ij} \tag{3}$$

2.4 Dynamic approaches

A dynamic graph can be denoted as a sequence of graphs, where $G^t(V^t, E^t, w^t)$ denotes the graph at time step t. The changes between graphs $G^{t-1}(V^{t-1}, E^{t-1}, w^{t-1})$ and $G^t(V^t, E^t, w^t)$ at consecutive time steps $t - 1$ and t can be denoted as a batch update Δ^t at time step t which consists of a set of edge deletions $\Delta^{t-} = \{(i, j) \mid i, j \in V\} = E^{t-1} \setminus E^t$ and a set of edge insertions $\Delta^{t+} = \{(i, j, w_{ij}) \mid i, j \in V; w_{ij} > 0\} = E^t \setminus E^{t-1}$ [75]. We refer to the setting where Δ^t consists of multiple edges being deleted and inserted as a *batch update*.

2.4.1 Naive-dynamic approach. The *Naive-dynamic* approach is a simple approach for identifying communities in dynamic networks. Here, one assigns vertices to communities from the previous snapshot of the graph and processes all the vertices, regardless of the edge deletions and insertions in the batch update (hence the prefix *naive*). This is demonstrated in Figure 6, where all vertices are marked as affected, highlighted in yellow. Since all communities are also marked as affected, they are all shown as hatched. Note that within the figure, edge deletions are shown in the top row (denoted by dashed lines), edge insertions are shown in the middle row (also denoted by dashed lines), and the migration of a vertex during the community detection algorithm is shown in the bottom row. The community membership obtained through this approach is guaranteed to be at least as accurate as the static algorithm. We refer to the parallel version of this approach as P-ND in the rest of the paper.

2.4.2 Dynamic Delta-screening approach [75]. *Dynamic Δ-screening* is a dynamic community detection approach that uses modularity-based scoring to determine an approximate region of the graph in which vertices are likely to change their community membership [75]. Figure 6 presents a high-level overview of the vertices (and communities), linked to a single source vertex i, that are identified as affected using the *Dynamic Δ-screening* approach in response to a batch update involving both edge deletions and insertions. As mentioned above, in this figure, edge deletions are shown in the top row (denoted by dashed lines), edge insertions are shown in the middle row (also denoted by dashed lines), and the migration of a vertex during the community detection algorithm is shown in the bottom row. Further, vertices marked as affected are highlighted in yellow, while entire communities marked as affected are hatched (in addition to its vertices being highlighted in yellow).

In the *Dynamic Δ-screening* approach, Zarayeneh et al. first sort the batch update consisting of edge deletions $(i, j) \in \Delta^{t-}$ and insertions $(i, j, w) \in \Delta^{t+}$ by their source vertex-id (separately). For edge deletions within the same community, they mark i's neighbors and j's community as affected. For edge insertions across communities, they pick the highest modularity changing vertex $j*$ among all the insertions linked to vertex i and mark i's neighbors and $j*$'s community as affected. Edge deletions between different communities and edge insertions between the same community are unlikely to affect the community membership of either of the vertices or any other vertices and hence ignored.

Dynamic Δ-screening proposed by Zarayeneh et al. [75] is not parallel. In this paper, we translate their approach into a multicore parallel algorithm. To this end, we scan sorted edge deletions and insertions in parallel, apply *Dynamic Δ-screening* as mentioned above, and mark vertices, neighbors of a vertex, and the community of a vertex using three separate flag vectors. Finally, we use the neighbors and community flag vectors to mark appropriate vertices. We also use per-thread collision-free hash tables. In the rest of the paper, we refer to this parallel version of *Dynamic Δ-screening* as P-DDS.

We apply P-DDS to the first pass of *Louvain* and initialize the community membership of each vertex at the start of the community detection algorithm to that obtained in the previous snapshot of the graph. We note that *Dynamic Δ-screening*, and in extension P-DDS, is *not* guaranteed to explore all vertices that have the potential to change their membership [75].

3 APPROACH

3.1 Our Parallel Louvain implementation

We use a parallel implementation of the *Louvain* method to determine suitable parameter settings and optimize the original algorithm through experimentation with a variety of techniques. We use *asynchronous* version of *Louvain*, where threads work independently on different parts of the graph. This allows for faster convergence but can also lead to more variability in the final result [8, 25]. Further, we allocate a separate hashtable per thread to keep track of the delta-modularity of moving to each community linked to a vertex in the local-moving phase of the algorithm, and to keep track of the total edge weight from one super-vertex to the other super-vertices in the aggregation phase of the algorithm.

Our optimizations include using OpenMP's `dynamic` loop schedule, limiting the number of iterations per pass to 20, using a tolerance drop rate of 10, setting an initial tolerance of 0.01, using

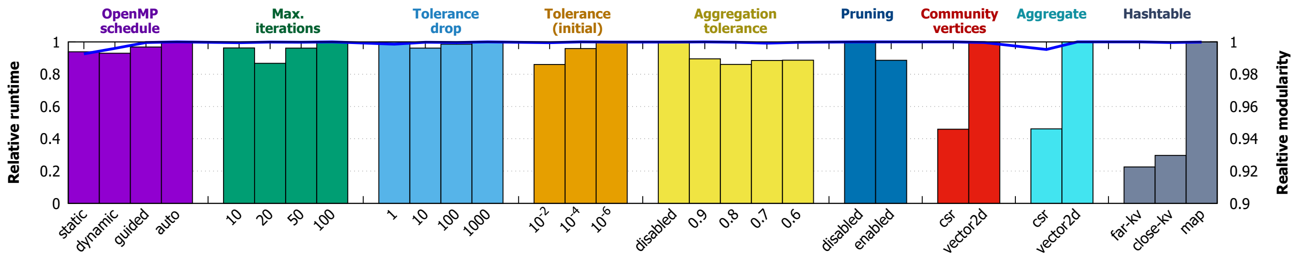


Figure 1: Impact of various parameter controls and optimizations on the runtime and result quality (modularity) of the *Louvain* algorithm. We show the impact of each optimization upon the relative runtime on the left Y-axis, and upon the relative modularity on the right Y-axis.

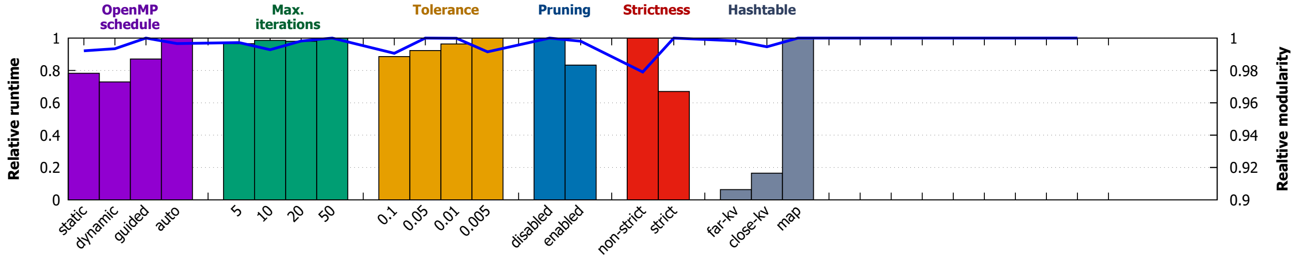


Figure 2: Impact of various parameter controls and optimizations on the runtime and result quality (modularity) of the *LPA*. Again, we show the impact of each optimization upon the relative runtime on the left Y-axis, and upon the relative modularity on the right Y-axis.

an aggregation tolerance of 0.8, employing vertex pruning, making use of parallel prefix sum and preallocated Compressed Sparse Row (CSR) data structures for finding community vertices and for storing the super-vertex graph during the aggregation phase, and using fast collision-free per-thread hashtables which are well separated in their memory addresses (*Far-KV*) for the local-moving and aggregation phases of the algorithm. Details on each of the optimizations is given below.

For each optimization, we test a number of relevant alternatives, and show the relative time and the relative modularity of communities obtained by each alternative in Figure 1. This result is obtained the running the tests on each graph in the dataset (see Table 2), 5 times on each graph to reduce the impact of noise, taking their geometric mean and arithmetic mean for the runtime and modularity respectively, and representing them as a ratio within each optimization category.

3.1.1 *Adjusting OpenMP loop schedule.*

We attempt *static*, *dynamic*, *guided*, and *auto* loop scheduling approaches of OpenMP (each with a chunk size of 2048) to parallelize the local-moving and aggregation phases of the *Louvain* algorithm. Results indicate that the scheduling behavior can have small impact on the quality of obtained communities. We consider OpenMP's `dynamic` loop schedule to be the best choice, as it helps achieve better load balancing when the degree distribution of vertices is non-uniform, and offers a 7% reduction in runtime with respect to OpenMP's *auto* loop schedule, with only a 0.4% reduction in the modularity of communities obtained (which is likely to be just noise).

3.1.2 *Limiting the number of iterations per pass.*

Restricting the number of iterations of the local-moving phase ensures its termination within a reasonable number of iterations, which helps minimize runtime. This can be important since the local-moving phase performed in the first pass is the most expensive step of the algorithm. However, choosing too small a limit may worsen convergence rate. Our results indicate that limiting the maximum number of iterations to 20 allows for 13% faster convergence, when compared to a maximum iterations of 100.

3.1.3 *Adjusting tolerance drop rate (threshold scaling).*

Tolerance is used to detect convergence in the local-moving phase of the *Louvain* algorithm, i.e., when the total delta-modularity in a given iteration is below or equal to the specified tolerance, the local-moving phase is considered to have converged. Instead of using a fixed tolerance across all passes of the *Louvain* algorithm, we can start with an initial high tolerance and then gradually reduce it. This is known as threshold scaling [25, 37, 41], and it helps minimize runtime as the first pass of the algorithm (which is usually the most expensive). Based on our findings, a tolerance drop rate of 10 yields 4% faster convergence, with respect to a tolerance drop rate of 1 (threshold scaling disabled), with no reduction in quality of communities obtained.

3.1.4 *Adjusting initial tolerance.*

Starting with a smaller initial tolerance allows the algorithm to explore broader possibilities for community assignments in the early stage, but comes at the cost of increased runtime. We find an initial tolerance of 0.01 provides

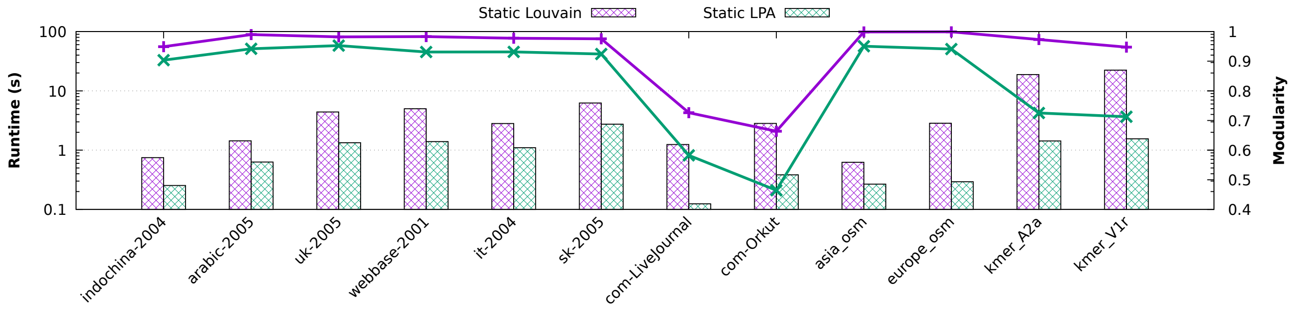
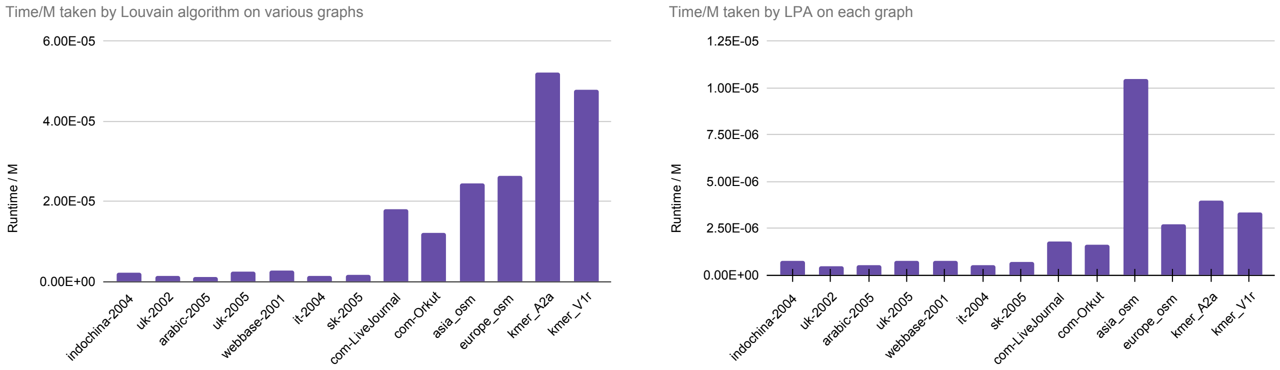


Figure 3: Time taken (boxes), and modularity of communities obtained (lines) with optimized *Static Louvain* and *Static LPA* for each graph in the dataset. Runtime is shown on the left Y-axis (in seconds), and modularity is shown on the right Y-axis.



(a) Louvain algorithm

(b) LPA

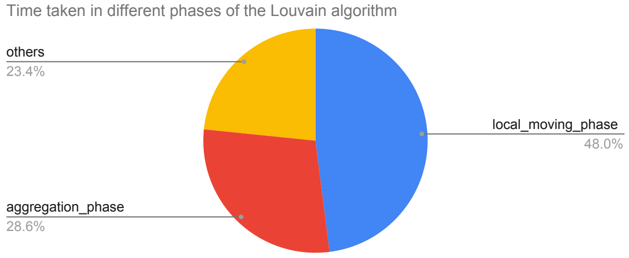
Figure 4: Runtime $/M$ with *Static Louvain* shown on the left, and runtime $/M$ with *Static LPA* shown on the right.

a 14% reduction in runtime of the algorithm with no reduction in the quality of identified communities, when compared to an initial tolerance of 10^{-6}.

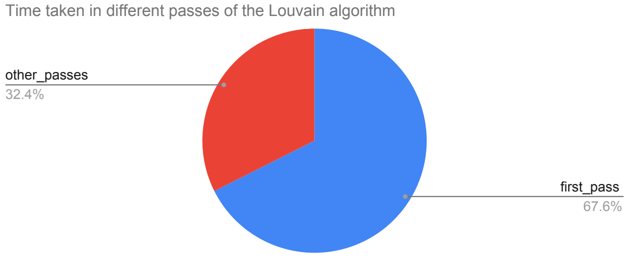
3.1.5 Adjusting aggregation tolerance. The aggregation tolerance determines the point at which communities are considered to have converged based on the number of community merges. In other words, if too few communities merged this pass we should stop here, i.e., if $|V_{aggregated}|/|V| \geq$ aggregation tolerance, we consider the algorithm to have converged. Adjusting aggregation tolerance allows the algorithm to stop earlier when further merges have minimal impact on the final result. According to our observations, an aggregation tolerance of 0.8 appears to be the best choice, as it presents a 14% reduction in runtime, when compared to the aggregation tolerance being disabled (1), while identifying final communities of equivalent quality.

3.1.6 Vertex pruning. Vertex pruning is a technique that is used to minimize unnecessary computation. Here, when a vertex changes its community, its marks its neighbors to be processed. Once a vertex has been processed, it is marked as not to be processed. However, it comes with an added overhead of marking an unmarking of vertices. Based on our results, vertex pruning justifies this overhead, and should be enabled for 11% improvement in performance.

3.1.7 Finding community vertices for aggregation phase. In the aggregation phase of the *Louvain* algorithm, the communities obtained in the previous local-moving phase of the algorithm are combined into super-vertices in the aggregated graph, with the edges between two super-vertices being equal to the total weight of edges between the respective communities. This requires one to obtain the list of vertices belonging to each community, instead of the mapping of community membership of each vertex that we have after the local-moving phase ends. A straight-forward implementation of this would make use of two-dimensional arrays for storing vertices belonging to each community, with the index in the first dimension representing the community id c, and the index in the second dimension pointing to the n^{th} vertex in the given community c. However, this requires memory allocation during the algorithm, which is expensive. Employing a parallel prefix sum technique along with a preallocated Compressed Sparse Row (CSR) data structure eliminates repeated memory allocation and deallocation, enhancing performance. Indeed, out findings indicate that using parallel prefix sum along with a preallocated CSR is 2.2× faster than using 2D arrays.



(a) Phase split



(b) Pass split

Figure 5: Phase split of *Static Louvain* shown on the left, and pass split of the algorithm shown on the right.

3.1.8 Storing aggregated communities (super-vertex graph). After the list of vertices belonging to each community have been obtained, the communities need to be aggregated (or compressed) into super-vertices, such that edges between two super-vertices being equal to the total weight of edges between the respective communities. This is generally called the super-vertex graph, or the compressed graph. It is then used as an input to the local-moving phase of the next pass of the Louvain algorithm. A simple data structure to store the super-vertex graph in the adjacency list format would be a two-dimensional array. Again, this requires memory allocation during the algorithm, which is bad for performance. Utilizing two preallocated CSRs, one for the source graph and the other for the target graph (except the first pass, where the dynamic graph may be stored in any desired format suitable for dynamic batch updates), along with parallel prefix sum can help here. We observe that using parallel prefix sum along with preallocated CSRs for maintaining the super-vertex graph is again 2.2× faster than using 2D arrays.

3.1.9 Hashtable design for local-moving/aggregation phases. One can use C++'s inbuilt maps as per-thread (independent) hashtables for the *Louvain* algorithm. But this has poor performance. So we use a key-list and a full-size values array (collision-free) to dramatically improve performance. However, if the memory addresses of the hashtables are nearby (*Close-KV*), even if each thread uses its own hashtable exclusively, performance is not as high. This is possibly due to false cache-sharing. Alternatively, if we ensure that the memory address of each hashtable are farther away (*Far-KV*), the performance improves. Our results indicate that *Far-KV* has the best performance and is 4.4× faster than *Map*, and 1.3× faster than *Close-KV*.

3.1.10 Results with optimized implementation. The combined optimizations yield impressive performance improvements in the OpenMP-based *Static Louvain*, with a completion time of 6.2 seconds on the undirected `sk-2005` graph containing 1.9 billion edges (refer to Figure 3). We observe that graphs with lower average degree (*road networks* and *protein k-mer graphs*) and graphs with poor community structure (such as `com-LiveJournal` and `com-Orkut`) have a larger time/M factor, as shown in Figure 4(a).

The phase-wise and pass-wise split of the optimized *Static Louvain* is shown in Figure 5. Note how 48% (most) of the runtime of the algorithm is spent in the local-moving phase, while only 29% of the runtime is spent in the aggregation phase of the algorithm. Further, 68% (most) of the runtime is spent in the first pass of the algorithm, which is the most expensive pass due to the size of the original graph (later passes work on super-vertex graphs) [67].

3.2 Our Parallel LPA implementation

Like *Louvain*, we use a parallel implementation of *LPA* and experiment with different optimizations and parameter settings. Again, as with *Louvain*, we use the *asynchronous* version of *LPA*. We observe that parallel *LPA* obtains communities of higher quality than its sequential implementation, possibly due to randomization. Further, we allocate a separate hash table per thread, as with the *Louvain* algorithm. In *LPA*, the hashtable is used to keep track of the total weight of each unique label linked to a vertex.

For *LPA*, our optimizations include using OpenMP's `dynamic` loop schedule, setting an initial tolerance of 0.05, enabling vertex pruning, employing the strict version of *LPA*, and using fast collision-free per-thread hashtables which are well separated in their memory addresses (*Far-KV*). See below for the details on each optimization. We evaluate multiple alternatives for each optimization, and show the relative time and the relative modularity of communities obtained by each alternative in Figure 2. Similar to *Louvain*, we perform these tests on every graph in the dataset (refer to Table 2), conducting them five times on each graph to minimize the influence of noise. We then calculate the geometric mean for the runtime and arithmetic mean for the modularity, and represent them as ratios within each optimization category.

3.2.1 Adjusting OpenMP loop schedule. We attempt *static*, *dynamic*, *guided*, and *auto* loop scheduling approaches of OpenMP (each with a chunk size of 2048) to parallelize *LPA*. Similar to the *Louvain* method, we consider OpenMP's `dynamic` loop schedule to be the best choice, due to its ability of work balancing among threads, and because it yields a runtime reduction of 27% when compared to OpenMP's *auto* loop schedule, while incurring only a 0.7% reduction in the modularity of obtained communities (which again is likely to be just noise).

3.2.2 Limiting the number of iterations. Restricting the number of iterations of *LPA* can ensure its termination within a reasonable number of iterations, but choosing a small limit may worsen the quality of communities obtained. Our results suggest that limiting the maximum number of iterations to 20 strikes a good balance between runtime and modularity.

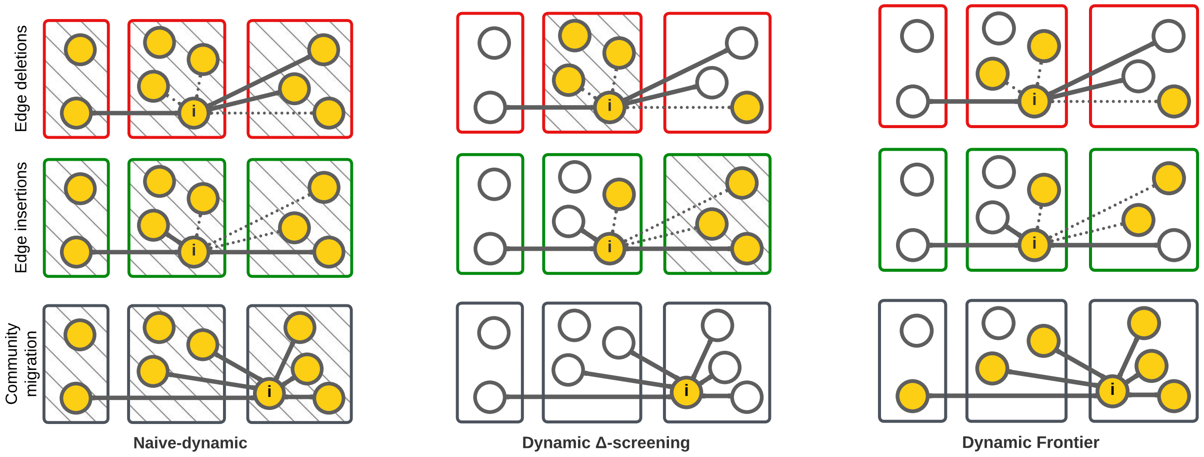


Figure 6: Comparison of dynamic community detection approaches: *Naive-dynamic* (P-ND), *Dynamic* Δ*-screening* (P-DDS), and *Dynamic Frontier* (P-DF). Vertices marked as affected (initially) with each approach are highlighted in yellow, and when entire communities are marked as affected, they are hatched.

3.2.3 Adjusting tolerance. Using a small tolerance allows the algorithm to explore broader possibilities for community assignments, but comes at the cost of increased runtime. We find an initial tolerance of 0.05 to be suitable. A tolerance of 0.1 may also be acceptable, but provides a very small gain in performance when compared to a tolerance of 0.05.

3.2.4 Vertex pruning. Vertex pruning is a method utilized to minimize unnecessary computation. In this approach, when a vertex alters its community, it assigns its neighbors for processing. Once a vertex has been processed, it is labeled as ineligible for further processing. However, this procedure incurs an additional overhead due to the marking and unmarking of vertices. Based on our findings, the employment of vertex pruning justifies this overhead and results in a performance enhancement of 17%.

3.2.5 Picking the best label. When there exist multiple labels connected to a vertex with maximum weight, we may randomly pick one of them (non-strict *LPA*), or pick only the first of them (strict *LPA*). We implement non-strict *LPA* using a simple modulo operator on the label id, as we observe that using *xorshift* based random number generator does not provide any advantage. Results indicate that the strict version of *LPA* is 1.5× faster than the non-strict approach, while also offering a gain in modularity of 2.1%.

3.2.6 Hashtable design. One can utilize C++'s inbuilt map as per-thread (independent) hashtables for the *LPA* algorithm. However, as mentioned before, this exhibits poor performance. Therefore, we employ a key-list and a collision-free full-size values array to dramatically improve performance. However, if the memory addresses of the hashtables are nearby (*Close-KV*), even if each thread uses its own hashtable exclusively, the performance is not as high. This is possibly due to false cache-sharing. Alternatively, if we ensure that the memory address of each hashtable is farther away (*Far-KV*), the performance improves. Our results indicate that *Far-KV* has the best performance and is 15.8× times faster than *Map*, and 2.6× times faster than *Close-KV* with *LPA*.

3.2.7 Results with optimized implementation. The combined optimizations result in high performance of the OpenMP-based *Static LPA* (see Figure 3). It has a runtime of 2.7 seconds on the undirected `sk-2005` graph containing 1.9 billion edges. We observe, as shown in Figure 4(b), that graphs with lower average degree (*road networks* and *protein k-mer graphs*) have a larger time/M factor.

3.3 Our Dynamic Frontier approach (P-DF)

Given a batch update on the original graph, it is likely that only a small subset of vertices in the graph would change their community membership. Selection of the appropriate set of affected vertices to be processed (that are likely to change their community), in addition to the overhead of finding them, plays a significant role in the overall accuracy and efficiency of a dynamic batch parallel algorithm. Too small a subset may result in poor-quality communities, while a too-large subset will increase computation time. However, P-ND processes all the vertices, while P-DDS generally overestimates the set of affected vertices and has a high overhead. Our proposed *Dynamic Frontier* approach (which we from here on refer to as P-DF) addresses these issues.

3.3.1 Explanation of the approach. We now explain P-DF. Consider a batch update consisting of edge deletions $(i, j) \in \Delta^{t-}$ and insertions $(i, j, w) \in \Delta^{t+}$, shown with double-dashed lines and double-solid lines respectively, with respect to a single source vertex i, in Figure 6. At the start of the community detection algorithm, we initialize the community membership of each vertex to that obtained in the previous snapshot of the graph.

Initial marking of affected vertices on edge deletion/insertion. For edge deletions between vertices belonging to the same community

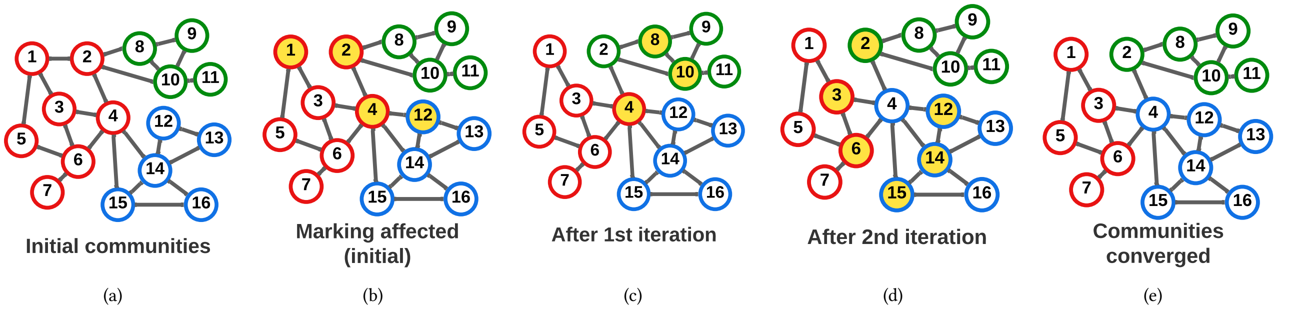


Figure 7: An example explaining the *Dynamic Frontier* approach (P-DF). The community membership of each vertex is shown with border color (red, green, or blue), and the algorithm proceeds from left to right.

and edge insertions between vertices belonging to different communities, we mark the source vertex i as affected, as shown with vertices highlighted in yellow, in Figure 6. Note that batch updates are undirected, so we effectively mark both the endpoints i and j. Edge deletions between vertices lying across communities and edge insertions for vertices lying within the same community are ignored (for reasons stated before).

Incremental marking of affected vertices on vertex migration to another community. When a vertex i changes its community during the community detection algorithm (shown with an arrow, with the direction indicating the migration of source vertex i from its previous community to another new community), we mark all its neighbor vertices $j \in J_i$ as affected, as shown if Figure 6 (highlighted in yellow), and mark i as not affected. To minimize unnecessary computation, we also mark an affected vertex i as not affected even if i does not change its community. We call this as the vertex pruning optimization. The process is akin to a graph traversal and continues until the communities have converged.

Application to the first pass of Louvain algorithm. We apply P-DF only to the first pass of *Louvain* algorithm (see Line 35 in Algorithm 1), as with P-DDS.

3.3.2 *A simple example.* Figure 7 shows an example of P-DF.

Initial communities. The original graph comprises a total of 16 vertices, which are divided into three communities, distinguished by the border colors of *red*, *green*, and *blue* (see Figure 7(a)). This community membership information of each vertex could have been obtained by executing either the static or dynamic version of the *Louvain*/*LPA* algorithm.

Batch update and Marking affected (initial). Subsequently a batch update is applied to the original graph (see Figure 7(b)), involving the deletion of an edge between vertices 1 and 2, and the insertion of an edge between vertices 4 and 12. Following the batch update, we perform the initial step of P-DF, marking endpoints 1, 2, 4, and 12 as affected. At this point, we are ready to execute the first iteration of a community detection algorithm.

After first iteration. During the first iteration (see Figure 7(c)), the community membership of vertex 2 changes from *red* to *green* because it exhibits stronger connections with vertices in the *green* community. In response to this change, the P-DF incrementally marks the neighboring vertices of 2 as affected, specifically vertices 8 and 10. Vertex 2 is no longer marked as affected due to vertex pruning.

After second iteration. Let us now consider the second iteration (see Figure 7(d)). Vertex 4 is now more strongly connected to the *blue* community, resulting in a change of its community membership from *red* to *blue*. As before, we mark the neighbors of vertex 4 as affected, namely vertices 12, 14, and 15. Vertex 4, once again, no longer marked as affected due to vertex pruning.

Communities converged. In the subsequent iteration (see Figure 7(e)), no other vertices have a strong enough reason to change their community membership. At this point, when employing *Louvain*, the aggregation phase commences consolidating communities into super-vertices to prepare for the subsequent pass of the algorithm. However, when employing the *LPA*, this marks the conclusion of the algorithm.

3.4 Our Dynamic Frontier based Louvain (P-DF$_L$, Algorithm 1)

We show how to apply our *Dynamic Frontier* approach (P-DF) to *Louvain* in Algorithm 1 (which we call P-DF$_L$). We take as input the previous snapshot of the graph G^{t-1}, the previous community membership of each vertex C^{t-1}, and the batch update consisting of edge deletions Δ^{t-} and insertions Δ^{t+} in Line 1. First, in Lines 3-4, we check if we should use *Static Louvain* to maintain the quality of communities (explained in Section 3.7). Then, in Lines 5-8, based on P-DF, we mark the initial set of vertices as affected. In Lines 9 and 11-12, we initialize the community membership of each vertex in the graph G^t, and each super-vertex in the aggregated graph G' (in the first pass, this is the same as the input graph G^t).

For each pass (Lines 10-20), we perform the *local-moving* phase of *Louvain* in Line 13. If the community labels converge after one iteration, we terminate the algorithm (Line 14). In Line 15, we renumber the community-ids. This renumbering helps generate the aggregated graph G' in the Compressed Sparse Row (CSR) format and counting the number of communities. In Line 16, we update the community membership of each vertex C based on the community membership of each super-vertex, such that we refer to the top-level hierarchy of the dendrogram as the final result.

Algorithm 1 *Dynamic Frontier* based *Louvain* (P-DF$_\text{L}$).

```
1:  function LOUVAIN($G^{t-1}, C^{t-1}, \Delta^{t-}, \Delta^{t+}$)
2:      $G^t \leftarrow (G^{t-1} \setminus \Delta^{t-}) \cup \Delta^{t+}$ ; $G' \leftarrow G^t$
3:      if $t$ mod RESTART_LOUVAIN = 0 then
4:          return $C^{t-1} \leftarrow staticLouvain(G^t)$
5:      for all $(i, j) \in \Delta^{t-}$ in parallel do
6:          Mark $i$ as affected if $C^{t-1}[i] = C^{t-1}[j]$
7:      for all $(i, j, w) \in \Delta^{t+}$ in parallel do
8:          Mark $i$ as affected if $C^{t-1}[i] \neq C^{t-1}[j]$
9:      Vertex membership: $C \leftarrow [0..|V^t|)$
10:     for all $l_p \in [0..\text{MAX\_PASSES})$ do
11:         Super-vertex membership: $C' \leftarrow [0..|V'|)$
12:         if $l_p = 0$ then $C' \leftarrow C^{t-1}$                ▷ First pass?
13:         $l_i \leftarrow louvainMove(G', C', l_p)$ ; $C_{old} \leftarrow C$
14:         if $l_i \leq 1$ then break                ▷ Globally converged?
15:         $C' \leftarrow$ Renumber communities in $C'$
16:         $C \leftarrow$ Lookup dendrogram using $C$ to $C'$
17:         $|\Gamma|, |\Gamma_{old}| \leftarrow$ Number of communities in $C, C_{old}$
18:         if $|\Gamma|/|\Gamma_{old}| < \tau_{agg}$ then break                ▷ Low shrink?
19:         $G' \leftarrow$ Aggregate communities in $G'$ using $C'$
20:         $\tau \leftarrow \tau/\text{TOLERANCE\_DROP}$                ▷ Threshold scaling
21:     return $C^{t-1} \leftarrow C$

22: function LOUVAINMOVE($G', C', l_p$)
23:     $K' \leftarrow$ Total edge weight of each vertex in $G'$
24:     $\Sigma' \leftarrow$ Total edge weight of each community in $G'$
25:     for all $l_i \in [0..\text{MAX\_ITERATIONS})$ do
26:         Delta modularity: $\Delta Q \leftarrow 0$
27:         for all $i \in V'$ in parallel do
28:             if $i$ is not affected then continue
29:             Mark $i$ as not affected (prune)
30:             $c^* \leftarrow$ Best community linked to $i$ in $G'$
31:             $\delta Q^* \leftarrow$ Delta-modularity of moving $i$ to $c^*$
32:             if $c^* = C'[i]$ then continue
33:             $C'[i] \leftarrow c^*$ ; $\Delta Q \leftarrow \Delta Q + \delta Q^*$
34:             $\Sigma'[d] -= K'[i]$ ; $\Sigma'[c] += K'[i]$ atomic
35:             if $l_p \neq 0$ then continue                ▷ Not first pass?
36:             Mark neighbors of $i$ as affected
37:         if $\Delta Q \leq \tau$ then break                ▷ Locally converged?
38:     return $l_i$
```

Next, at Line 18, we check whether only a small portion of communities have merged together. To do this, we calculate the ratio of the number of communities after the *local-moving* phase $|\Gamma|$ to the original number of communities $|\Gamma_{old}|$. If this ratio $|\Gamma|/|\Gamma_{old}|$ is less than a specific value called the aggregation tolerance (τ_{agg}, whose optimal value is measured and mentioned in Section 3.1), it means that the merging has been minimal. In such a situation, we stop the algorithm to avoid the computationally expensive *aggregation* phase, as it doesn't provide significant benefits. However, if a sufficiently large number of communities have merged together, we proceed with the *aggregation* phase and obtain the aggregated graph G'. This graph is stored in CSR format.

Algorithm 2 *Dynamic Frontier* based *LPA* (P-DF$_\text{LPA}$).

```
function LPA($G^{t-1}, C^{t-1}, \Delta^{t-}, \Delta^{t+}$)
    $G^t \leftarrow (G^{t-1} \setminus \Delta^{t-}) \cup \Delta^{t+}$ ; $G' \leftarrow G^t$
    for all $(i, j) \in \Delta^{t-}$ in parallel do
        Mark $i$ as affected if $C^{t-1}[i] = C^{t-1}[j]$
    for all $(i, j, w) \in \Delta^{t+}$ in parallel do
        Mark $i$ as affected if $C^{t-1}[i] \neq C^{t-1}[j]$
    Vertex membership: $C' \leftarrow C^{t-1}$
    for all $l_i \in [0..\text{MAX\_ITERATIONS})$ do
        $\Delta N \leftarrow lpaMove(G', C')$
        if $\Delta N/N \leq \tau$ then break                ▷ Converged?
    return $C^{t-1} \leftarrow C'$

function LPAMOVE($G', C'$)
    Changed vertices: $\Delta N \leftarrow 0$
    for all $i \in V'$ in parallel do
        if $i$ is not affected then continue
        Mark $i$ as not affected (prune)
        $c^* \leftarrow$ Most weighted label to $i$ in $G'$
        if $c^* = C'[i]$ then continue
        $C'[i] \leftarrow c^*$ ; $\Delta N \leftarrow \Delta N + 1$
        Mark neighbors of $i$ as affected
    return $\Delta N$
```

Algorithm 3 *Dynamic Frontier Hybrid Louvain-LPA* (P-DF$_\text{H}$).

```
function HYBRIDLOUVAINLPA($G^{t-1}, C^{t-1}, \Delta^{t-}, \Delta^{t+}$)
    $G^t \leftarrow (G^{t-1} \setminus \Delta^{t-}) \cup \Delta^{t+}$
    if $t$ mod RESTART_HYBRID = 0 then
        return $C^{t-1} \leftarrow staticLouvain(G^t)$
    return $lpa(G^{t-1}, C^{t-1}, \Delta^{t-}, \Delta^{t+})$
```

Next, we perform the threshold scaling optimization [41], i.e., we reduce the tolerance τ by a tolerance drop factor of TOLERANCE_DROP. This optimization helps minimize the number of iterations performed in the local-moving phase of *Louvain*. It improves performance with little sacrifice in the quality of communities obtained.

We now discuss the *local-moving* phase of *Louvain*. First, in Lines 23-24, we calculate the total edge weights linked to each vertex K' and the total edge weights linked to each community Σ'. For each iteration (Lines 25-36), and for each affected vertex i in the graph G', we use per-thread collision-free hashtables to obtain the best community linked to each vertex c^*, as well as the associated delta-modularity (highest) δQ^* in parallel using Equation 2 (Lines 30-31). If the best community c^* is different from the original community membership $C'[i]$ of vertex i (Line 32), we update the community membership of the vertex and atomically update the total edge weights linked to each community in Lines 33-34. In addition, if this is the first pass of *Louvain* (Line 35), based on P-DF, we mark the neighbors of vertex i as affected in Line 36. To minimize unnecessary computation, we also mark vertex i as not affected (whether or not i changes its community) as part of vertex pruning optimization in Line 29. At the end of each iteration, if the total delta-modularity across all vertices ΔQ is less than the specified tolerance τ, we terminate the local-moving phase (Line 37). We prove the correctness of P-DF$_L$ in Section 4.1.

3.5 Our Dynamic Frontier based LPA (P-DF$_{LPA}$, Algorithm 2)

We show how to apply our *Dynamic Frontier* approach (P-DF) to *LPA* in Algorithm 2 (which we call P-DF$_{LPA}$). As before, we take as input the previous snapshot of the graph G^{t-1}, the previous community membership of each vertex C^{t-1}, and the batch update consisting of edge deletions Δ^{t-} and insertions Δ^{t+} in Line 1. In Lines 3-6, based on P-DF, we mark the initial set of vertices as affected. In Line 7, we initialize the community membership of each vertex in the graph G^t. For each iteration (Lines 8-10), we perform the *label-propagation* step of *LPA* in Line 9. If only a small fraction of vertices changed their community membership, we recognize that the communities have converged and hence end the algorithm (Line 10).

We now discuss the *label-propagation* step of *LPA*. For each affected vertex i in the graph G', we use per-thread collision-free hash tables to obtain the most weighted label to each vertex c^* in parallel using Equation 3 (Line 17). If the best label c^* is different from the original label $C'[i]$ of vertex i (Line 18), we update the label associated with the vertex in Line 19. In addition, based on P-DF, we mark the neighbors of vertex i as affected in Line 20. As earlier, we also mark vertex i as not affected (whether or not i changes its community) as part of vertex pruning optimization in Line 16. We prove the correctness of P-DF$_{LPA}$ in Section 4.2.

3.6 Our Dynamic Frontier based Hybrid Louvain-LPA (P-DF$_H$, Algorithm 3)

Louvain is known for its high-quality community detection but at the cost of being slow. On the other hand, *LPA* is fast, but the communities it detects are of moderate quality. We propose to combine the strengths of both algorithms by creating a Hybrid dynamic algorithm, which we call P-DF$_H$. One could use either *Louvain* or *LPA* as the base (static) algorithm for obtaining the initial communities and use the other (dynamic) algorithm to update the communities. Our initial experimentation shows that using *Louvain* as the base/static method and *LPA* as the dynamic method (i.e., P-DF$_H$), provides us with superior performance. Algorithm 3 shows the details of this approach.

3.7 Maintaining quality across batches

In our experiments with continuous batch updates of edge insertions of size $10^{-3}|E|$, we find that the quality of communities obtained using the P-DDS$_L$ and P-DF$_L$ starts to drop (compared to *Static Louvain*) by 48% (rapid decline) and 5.7% respectively after around 1300 batches of updates. The same happens for P-DF$_H$ by 10% after around 600 updates. Therefore, we conservatively use a `RESTART_LOUVAIN` of 1000, and a `RESTART_HYBRID` of 500 (see Lines 3-4 in Algorithm 1, and Lines 3-4 in Algorithm 3). As these Lines show, we run *Static Louvain* once for 1000/500 batches of updates to maintain the quality of communities across multiple batch updates to correct the error introduced by repeated applications. We adjust our reported timings by amortizing its cost.

3.8 Time and Space complexity

To analyze the time complexity of our algorithms, we use N_B to denote the number of vertices marked as affected (which is dependent on the size and nature of batch update) by the dynamic algorithm on a batch B of edge updates, use M_B to denote the number of edges with one endpoint in N_B, and K to denote the total number of iterations performed. Then the time complexity of Algorithms 1-3 is $O(KM_B)$. In the worst case, the time complexity of our algorithms would be the same as that of the respective static algorithms, i.e., $O(KM)$. The space complexity of our algorithms is the same as that of the static algorithms, i.e., $O(N + M)$.

4 CORRECTNESS

We now provide arguments for the correctness of P-DF$_L$ and P-DF$_{LPA}$. To help with this, we refer the reader to Figure 8. Here, pre-existing edges are represented by solid lines, and i represents a source vertex of edge deletions/insertions in the batch update. Edge deletions in the batch update with i as the source vertex are shown in the top row (denoted by dashed lines), edge insertions are shown in the middle row (also denoted by dashed lines), and community migration of vertex i is is shown in the bottom row. Vertices i_n and j_n represent the destination vertices (of edge deletions or insertions). Vertices i', j', and k' signify neighboring vertices of vertex i. Finally, vertices i'', j'', and k'' represent non-neighbor vertices (to vertex i). Yellow highlighting is used to indicate vertices marked as affected, initially or in the current iteration of the community detection algorithm. We understand this figure is dense, but we tried to capture several details for correctness arguments in Sections 4.1 and 4.2.

4.1 Correctness of Dynamic Frontier based Louvain (P-DF$_L$, Algorithm 1)

Given a batch update consisting of edge deletions Δ^{t-} and insertions Δ^{t+}, we now show that P-DF$_L$ marks the essential vertices, which have an incentive to change their community membership, as affected. For any given vertex i in the original graph (before the batch update), the delta-modularity of moving it from its current community d to a new community c is given by Equation 4. We now consider the direct effect of each individual edge deletion (i, j) or insertion (i, j, w) in the batch update, on the delta-modularity of the a vertex, as well as the indirect cascading effect of migration of a vertex (to another community) on other vertices.

$$\Delta Q_{i:d\to c} = \frac{1}{m}(K_{i\to c} - K_{i\to d}) - \frac{K_i}{2m^2}(K_i + \Sigma_c - \Sigma_d) \qquad (4)$$

4.1.1 On edge deletion.

Lemma 4.1. *Given an edge deletion (i, j) between vertices i and j belonging to the same community d, vertex i (and j) should be marked as affected.*

Consider the case of edge deletion (i, j) of weight w between vertices i and j belonging to the same community $C_i = C_j = d$ (see Figure 8, where $j = i_1$). Let i'' be a vertex belonging i's community $C_{i''} = d$, and let k'' be a vertex belonging to another community $C_{k''} = b$. As shown below in Case **(1)**, the delta-modularity of vertex i moving from its original community d to another community b

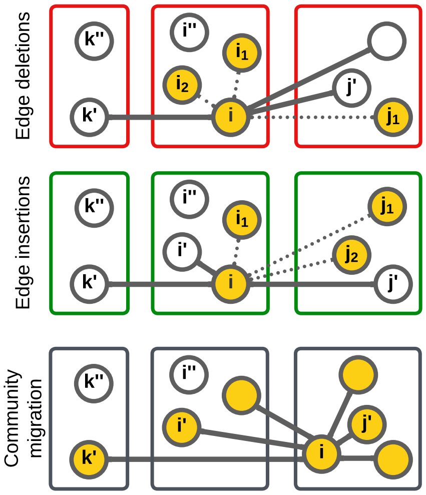


Figure 8: *Dynamic Frontier* approach (P-DF) in detail.

has a significant positive factor w/m. There is thus a chance that vertex i would change its community membership, and we should mark it as affected. The same argument applies for vertex j, as the edge is undirected. On the other hand, for the Cases **(2)-(3)**, there is only a small positive change in delta-modularity for vertex k''. Thus, there is little incentive for vertex k'' to change its community membership, and no incentive for a change in community membership of vertex i''.

Note that it is possible that the community d would split due to the edge deletion. However, this is unlikely, given that one would need a large number of edge deletions between vertices belonging to the same community for the community to split. With P-DF$_L$, such rare events are taken care of by running *Static Louvain* every `RESTART_LOUVAIN` batch updates, which also helps us ensure high-quality communities. The same applies to P-DDS$_L$.

(1) $\Delta Q^{new}_{i:d\to b} = \Delta Q_{i:d\to b} + [\frac{w}{m}] + \frac{w}{2m^2}(\Sigma_c - \Sigma_d + w)$

(2) $\Delta Q^{new}_{i'':d\to b} = \Delta Q_{i'':d\to b} - \frac{wK_{i''}}{m^2}$

(3) $\Delta Q^{new}_{k'':b\to d} = \Delta Q_{k'':b\to d} + \frac{wK_{k''}}{m^2}$

Now, consider the case of edge deletion (i, j) between vertices i and j belonging to different communities, i.e., $C_i = d, C_j = c$ (see Figure 8, where $j = j_2$ or j_3). Let i'' be a vertex belonging to i's community $C_{i''} = d$, j'' be a vertex belonging to j's community $C_{j''} = c$, and k'' be a vertex belonging another community $C_{k''} = b$. As shown in Cases **(4)-(8)**, due to the absence of any significant positive change in delta-modularity, there is little to no incentive for vertices i, j, k'', i'', and j'' to change their community membership.

(4) $\Delta Q^{new}_{i:d\to c} = \Delta Q_{i:d\to c} - \frac{w}{m} + \frac{w}{2m^2}(2K_i + \Sigma_c - \Sigma_d - w)$

(5) $\Delta Q^{new}_{i:d\to b} = \Delta Q_{i:d\to b} + \frac{w}{2m^2}(K_i + \Sigma_b - \Sigma_d)$

(6) $\Delta Q^{new}_{i'':d\to c} = \Delta Q_{i'':d\to c}$

(7) $\Delta Q^{new}_{i'':d\to b} = \Delta Q_{i'':d\to b} - \frac{wK_{i''}}{2m^2}$

(8) $\Delta Q^{new}_{k'':b\to d/c} = \Delta Q_{k'':b\to d/c} + \frac{wK_{k''}}{m^2}$ ◇

4.1.2 On edge insertion.

Lemma 4.2. *Given an edge insertion (i, j, w) between vertices i and j belonging to different communities d and c, vertex i (and j) should be marked as affected.*

Let us consider the case of edge insertion (i, j, w) between vertices i and j belonging to different communities $C_i = d$ and $C_j = c$ respectively (see Figure 8, where $j = j_3$). Let i'' be a vertex belonging i's community $C_{i''} = d$, j'' be a vertex belonging to j's community $C_{j''} = c$, and k'' be a vertex belonging to another community $C_{k''} = b$. As shown below in Case **(9)**, we have a significant positive factor w/m (and a small negative factor) which increases the delta-modularity of vertex i moving to j's community after the insertion of the edge (i, j). There is, therefore, incentive for vertex i to change its community membership. Accordingly, we mark i as affected. Again, the same argument applies for vertex j, as the edge is undirected. Further, we observe from other Cases (**(10)-(13)**) there is only a small change in delta-modularity. Thus, there is hardly any to no incentive for a change in community membership of vertices i'', j'', and k''.

(9) $\Delta Q^{new}_{i:d\to c} = \Delta Q_{i:d\to c} + [\frac{w}{m}] - \frac{w}{2m^2}(2K_i + \Sigma_c - \Sigma_d + w)$

(10) $\Delta Q^{new}_{i:d\to b} = \Delta Q_{i:d\to b} - \frac{w}{2m^2}(K_i + \Sigma_b - \Sigma_d)$

(11) $\Delta Q^{new}_{i'':d\to c} = \Delta Q_{i'':d\to c}$

(12) $\Delta Q^{new}_{i'':d\to b} = \Delta Q_{i'':d\to b} + \frac{wK_{i''}}{2m^2}$

(13) $\Delta Q^{new}_{k'':b\to d/c} = \Delta Q_{k'':b\to d/c} - \frac{wK_{k''}}{2m^2}$

Now, consider the case of edge insertion (i, j, w) between vertices i and j belonging to the same community $C_i = C_j = d$ (see Figure 8, where $j = i_1$ or i_2). From Cases **(14)-(16)**, we note that it is little to no incentive for vertices i'', k'', i, and j to change their community membership. Note that it is possible for the insertion of edges within the same community to cause it to split into two more strongly connected communities, but it is very unlikely.

(14) $\Delta Q^{new}_{i:d\to b} = \Delta Q_{i:d\to b} - \frac{w}{m} - \frac{w}{2m^2}(\Sigma_c - \Sigma_d - w)$

(15) $\Delta Q^{new}_{i'':d\to b} = \Delta Q_{i'':d\to b} + \frac{wK_{i''}}{m^2}$

(16) $\Delta Q^{new}_{k'':b\to d} = \Delta Q_{k'':b\to d} - \frac{wK_{k''}}{m^2}$ ◇

4.1.3 On vertex migration to another community.

Lemma 4.3. *When a vertex i changes its community membership, and vertex j is its neighbor, j should be marked as affected.*

We considered the direct effects of deletion and insertion of edges above. Now we consider its indirect effects by studying the impact of change in community membership of one vertex on the other vertices. Consider the case where a vertex i changes its community membership from its previous community d to a new community c (see Figure 8). Let i' be a neighbor of i and i'' be a non-neighbor of i belonging to i's previous community $C_{i'} = C_{i''} = d$, j' be a neighbor of i and j'' be a non-neighbor of i belonging to i's new community $C_{j'} = C_{j''} = c$, k' be a neighbor of i and k'' be a non-neighbor of i belonging to another community $C_{k'} = C_{k''} = b$. From Cases **(17)-(22)**, we note that neighbors i' and k' have an incentive to change their community membership (as thus necessitate marking), but not j'. However, to keep the algorithm simple, we simply mark all the neighbors of vertex i as affected.

(17) $\Delta Q^{new}_{i':d\to c} = \Delta Q_{i':d\to c} + [\frac{2w_{ii'}}{m}] - \frac{K_i K_{i'}}{m^2}$

(18) $\Delta Q^{new}_{i':d\to b} = \Delta Q_{i':d\to b} + [\frac{w_{ii'}}{m}] - \frac{K_i K_{i'}}{2m^2}$

(19) $\Delta Q^{new}_{j':c\to d} = \Delta Q_{j':c\to d} - \frac{2w_{ij'}}{m} + \frac{K_i K_{j'}}{m^2}$

(20) $\Delta Q^{new}_{j':c\to b} = \Delta Q_{j':c\to b} - \frac{w_{ij'}}{m} + \frac{K_i K_{j'}}{2m^2}$

(21) $\Delta Q^{new}_{k':b\to d} = \Delta Q_{k':b\to d} - \frac{w_{ik'}}{m} + \frac{K_i K_{k'}}{2m^2}$

(22) $\Delta Q^{new}_{k':b\to c} = \Delta Q_{k':b\to c} + [\frac{w_{ik'}}{m}] - \frac{K_i K_{k'}}{2m^2}$

Further, from Cases **(23)-(28)**, we note that there is hardly any incentive for a change in community membership of vertices i'', j'', and k''. This is due to the change in delta-modularity being insignificant. There could still be an indirect cascading impact, where a common neighbor between vertices i and j would change its community, which could eventually cause vertex j to change its community as well [75]. However, this case is automatically taken care of as we perform marking of affected vertices during the community detection process.

(23) $\Delta Q^{new}_{i'':d\to c} = \Delta Q_{i'':d\to c} + \frac{K_i K_{i''}}{m^2}$

(24) $\Delta Q^{new}_{i'':d\to b} = \Delta Q_{i'':d\to b} - \frac{K_i K_{i''}}{2m^2}$

(25) $\Delta Q^{new}_{j'':c\to d} = \Delta Q_{j'':c\to d} + \frac{K_i K_{j''}}{m^2}$

(26) $\Delta Q^{new}_{j'':c\to b} = \Delta Q_{j'':c\to b} + \frac{K_i K_{j''}}{2m^2}$

(27) $\Delta Q^{new}_{k'':b\to d} = \Delta Q_{k'':b\to d} + \frac{K_i K_{k''}}{2m^2}$

(28) $\Delta Q^{new}_{k'':b\to c} = \Delta Q_{k'':b\to c} - \frac{K_i K_{k''}}{2m^2}$ ◇

4.1.4 Overall. Finally, based on Lemmas 4.1, 4.2, and 4.3, we can state the following for P-DF$_{\text{L}}$.

THEOREM 4.4. *Upon a given batch update, P-DF$_{\text{L}}$ marks vertices having an incentive to change their community membership as affected.* □

We note that with P-DF$_{\text{L}}$, outlier vertices may not be marked as affected even if they have the potential to change community without any direct link to vertices in the frontier. Such outliers may be weakly connected to multiple communities, and if the current community becomes weakly (or less strongly) connected, they may leave and join some other community. It may also be noted that P-DDS$_{\text{L}}$ is also an approximate approach and can miss certain outliers. In practice, however, we see little to no impact of this approximation of the affected subset of the graph on the final quality (modularity) of the communities obtained, as shown in Section 5.

4.2 Correctness of Dynamic Frontier based LPA (P-DF$_{\text{LPA}}$, Algorithm 2)

Given a batch update consisting of edge deletions Δ^{t-} and insertions Δ^{t+}, we now show that P-DF$_{\text{LPA}}$ marks all vertices as affected that might change their community membership. With *LPA*, the label C_i of a vertex i is determined as given in Equation 5. We now consider the direct effect of each individual edge deletion (i, j) or insertion (i, j, w) in the batch update, on the label a vertex, along with the indirect cascading effect of the change of label of a vertex on the label associated with other vertices.

$$C_i = \underset{c \,\in\, \Gamma}{\arg\max} \sum_{j \in J_i \,\mid\, C_j = c} w_{ij} \tag{5}$$

4.2.1 On edge deletion.

LEMMA 4.5. *Given an edge deletion (i, j) between vertices i and j having the same label, vertex i (and j) should be marked as affected.*

Consider the case of edge deletion (i, j) of weight w, between vertices i and j having the same label $C_i = C_j = d$. The new label of i would be $C_i^{new} = \arg\max\{[d, K_{i\to d} - w], ...\}$. Here we have a reduced total weight associated with the previous best label d. Thus, i's label can change, and we mark it as affected. The same argument applies to vertex j as the edges are undirected.

Now consider the case of edge deletion (i, j) between vertices i and j having different labels $C_i = d$ and $C_j = c$ respectively. The new label for vertex i would be $C_i^{new} = \arg\max\{(d, K_{i\to d}), ...\}$. As we do not have any reduction in total weight associated with the previous best label d, the label of vertex i cannot change. Again, the same argument applies from vertex j. ◇

4.2.2 On edge insertion.

LEMMA 4.6. *Given an edge insertion (i, j, w) between vertices i and j having different labels, vertex i (and j) should be marked as affected.*

Consider the case of edge insertion (i, j, w) between vertices i and j having different labels $C_i = d$ and $C_j = c$. The new label for vertex i would be $C_i^{new} = \arg\max\{(d, K_{i\to d}), (c, K_{i\to c} + w)\}$. Here, c may be the new maximum label for vertex i. We thus mark vertex i as affected. Again, the same argument applies for j due to the edges being undirected.

Now consider the case of edge insertion (i, j, w) between vertices i and j having the same label $C_i = C_j = d$. The new label for vertex i would be $C_i^{new} = \arg\max\{(d, K_{i\to d}) + w), ...\}$. Now, here we actually have an increase in the total weight associated with the previous best label d. Thus, the label of vertex i cannot change. Again, the same argument applies to j. ◇

4.2.3 On vertex migration to another community.

LEMMA 4.7. *When a vertex i changes its label, and vertex j is its neighbor, the neighbor vertex j should be marked as affected.*

We now consider the indirect effects of deletion and insertion of edges by observing the impact of change in the label of one vertex on the labels of other vertices. Consider the case where a vertex i with label $C_i = d$ changes its label to $C_i^{new} = c$. Let i' be a neighbor of i with i's previous label $C_{i'} = d$, j' be a neighbor of i with i's new label $C_{j'} = d$, and k' be a neighbor of i with another label $C_{k'} = b$.

From Cases **(29)-(31)**, we note that neighbors i' and k' have a possibility to change their community membership (as thus necessitate marking), but not j'. However, to keep the algorithm simple, we simply mark all the neighbors of vertex i as affected. Finally, consider the case where vertices i and i'' are not neighbors, and vertex i changes its label. Note that by the definition of *LPA*, this cannot affect the label of vertex i''. However, there could still be an indirect impact, where a common neighbor between vertices i and i'' would change its label, which could eventually cause vertex i'' to change its label. Note that this case is automatically taken care of as we perform marking of affected vertices during the community detection process.

(29) $C^{new}_{i'} = \arg\max\{(d, K_{i'\to d} - w), (c, K_{i'\to c} + w)\}$

(30) $C^{new}_{j'} = \arg\max\{(c, K_{j'\to c} + w)\}$

(31) $C^{new}_{k'} = \arg\max\{(d, K_{i'\to d} - w), (c, K_{i'\to c} + w)\}$ ◇

4.2.4 Overall. Finally, based on Lemmas 4.5, 4.6, and 4.7, we can state the following for P-DF$_{\text{LPA}}$.

Theorem 4.8. *Upon a given batch update, P-DF$_{LPA}$ marks any vertices that could change their labels as affected.* □

5 EVALUATION

5.1 Experimental setup

5.1.1 System. For our experiments, we use a server that has an x86-based 64-bit AMD EPYC-7742 processor. This processor has a clock frequency of 2.25 GHz and 512 GB of DDR4 system memory. Each core has an L1 cache of 4 MB, an L2 cache of 32 MB, and a shared L3 cache of 256 MB. The machine runs on Ubuntu 20.04. We use GCC 9.4 and OpenMP 5.0 [45].

5.1.2 Configuration. We use 32-bit unsigned integer for vertex ids, 32-bit floating point for edge weights, but use 64-bit floating point for hashtable values, total edge weight, modularity calculation, and all places where performing an aggregation/sum of floating point values. Unless mentioned otherwise, we execute all parallel implementations with a default of 64 threads (to match the number of cores available on the system).

Table 2: List of 12 graphs obtained from [31] (directed graphs are marked with $*$). Here, $|V|$ is the total number of vertices, $|E|$ is the total number of edges (after making the graph undirected), $|\Gamma|$ is the number of communities obtained using *Static Louvain*. In the table, B refers to a billion, M refers to a million and K refers a thousand.

Graph	$\|V\|$	$\|E\|$	$\|\Gamma\|$
Web Graphs (LAW)			
indochina-2004*	7.41M	341M	4.24K
arabic-2005*	22.7M	1.21B	3.66K
uk-2005*	39.5M	1.73B	20.8K
webbase-2001*	118M	1.89B	2.76M
it-2004*	41.3M	2.19B	5.28K
sk-2005*	50.6M	3.80B	3.47K
Social Networks (SNAP)			
com-LiveJournal	4.00M	69.4M	2.54K
com-Orkut	3.07M	234M	29
Road Networks (DIMACS10)			
asia_osm	12.0M	25.4M	2.38K
europe_osm	50.9M	108M	3.05K
Protein k-mer Graphs (GenBank)			
kmer_A2a	171M	361M	21.2K
kmer_V1r	214M	465M	6.17K

5.1.3 Reproducibility. All our results are reproducible. The source code for the experiments reported in this paper along with necessary scripts for obtaining the datasets and compiling the software is available at https://bit.ly/hipc23-01-artifact.

5.1.4 Dataset. Table 2 shows the graphs we use in our experiments. All of them are obtained from the SuiteSparse Matrix Collection [31]. The number of vertices in the graphs varies from 3.07 to 214 million, and the number of edges varies from 25.4 million to 3.80 billion. We ensure that all edges are undirected and weighted with a default weight of 1.

5.1.5 Batch generation. We take a base graph from the dataset and generate a random batch update consisting of an equal mix of edge deletions and insertions (each with an edge weight of 1). All batch updates are undirected, i.e., for every edge insertion (i, j, w) in the batch update, there is a reverse edge (j, i, w) that is also a part of the batch update. For simplicity, these edges are generated such that the selection of each vertex (as endpoint) is equally probable, and we ensure that no new vertices are added to the graph.

Adjusting batch size. For all dynamic graph-based experiments, we modify the batch size as a fraction of the total number of edges in the original (undirected) graph from 10^{-7} to 0.1 (i.e., $10^{-7}|E|$ to $0.1|E|$). For a billion-edge graph, this amounts to a batch size of 100 to 100 million edges. Keep in mind that dynamic graph algorithms are helpful for small batch sizes in interactive applications. For large batches, it is usually more efficient to run the static algorithm.

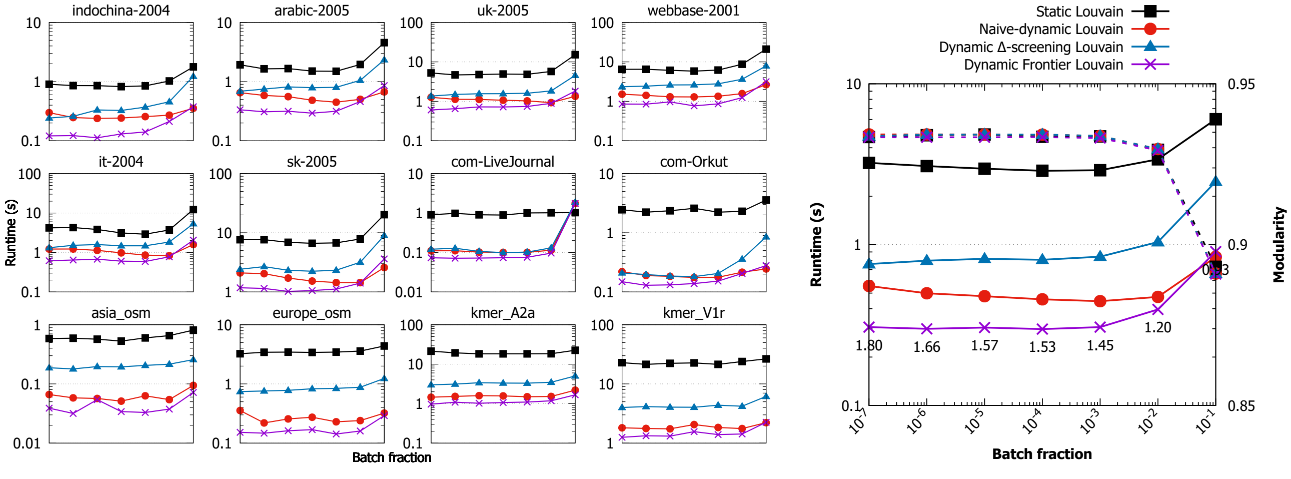
Minimizing measurement noise. We employ 5 distinct random batch updates for each batch size and report average across these runs in our experiments.

5.1.6 Determining optimality of result. Community detection is an NP-hard problem and existing polynomial algorithms are *heuristic*. We study correctness in terms of *modularity score* of communities identified (higher is better), similar to previous works in the area [65, 75]. As Figures 9-13 show, modularity of communities detected by our proposed dynamic algorithms is close to the modularity of communities detected by corresponding static algorithms.

5.2 Performance of Dynamic Frontier based Louvain (P-DF$_{\text{L}}$, Algorithm 1)

5.2.1 Overall Performance. We first study the performance of P-DF$_{\text{L}}$ on batch updates of size $10^{-7}|E|$ to $0.1|E|$, and compare it with *Static Louvain*, P-ND$_{\text{L}}$, and P-DDS$_{\text{L}}$. The work of Zarayeneh et al. [75] demonstrates improved performance of *Δ-screening* compared to *Dynamo* [79] and *Batch* [11]. Thus, we limit our comparison to P-DDS. When executing P-DF$_{\text{L}}$ and P-DDS$_{\text{L}}$, we reinitialize the community memberships every RESTART_LOUVAIN batches with the community labels obtained via *Static Louvain* (see Section 3.7). As mentioned in Section 5.1.5, we generate 5 different random batch updates for each batch size to minimize measurement noise.

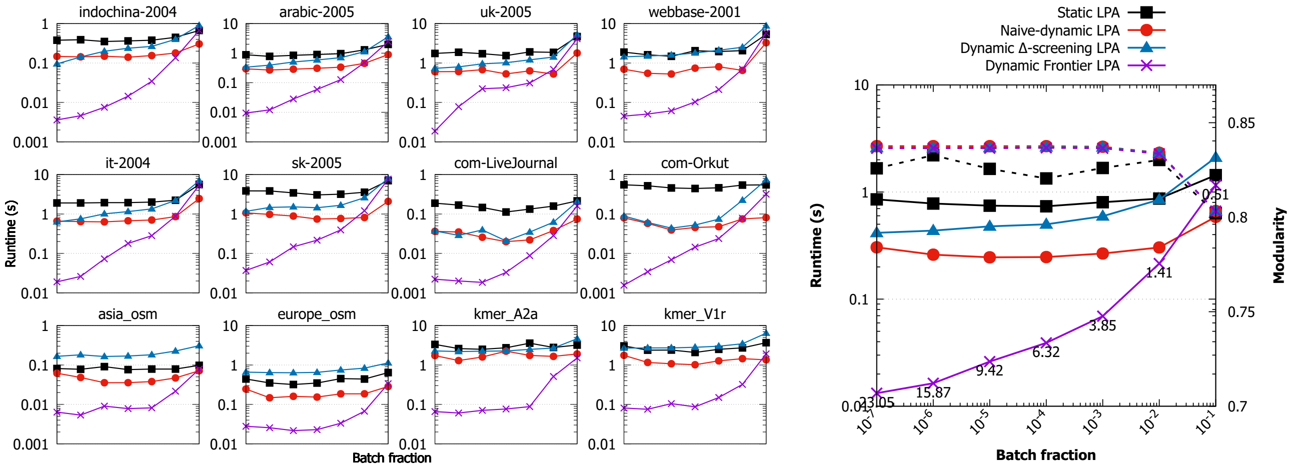
Figure 9(b) shows the results of the experiment. We observe the following from Figure 9(b). The modularity of communities obtained by all the dynamic approaches is nearly identical. P-DF$_{\text{L}}$ converges the fastest with an average speedup of 1.5× over P-ND$_{\text{L}}$. As the batch size increases, the number of vertices marked as affected by P-DF$_{\text{L}}$ increases. This results in an increase in the time taken by P-DF$_{\text{L}}$ as the batch size increases. Further, as Figure 9(a) shows, dynamic approaches significantly outperform *Static Louvain* on *social networks*, *road networks*, and *k-mer protein graphs* (which do not have a dense community structure, or have a low $|E|/|V|$ ratio).



(a) Results of each graph

(b) Overall result

Figure 9: Time taken (solid lines), and modularity of communities obtained (dashed lines) along the right Y-axis, with *Static Louvain*, P-ND$_L$, P-DDS$_L$, and P-DF$_L$(Algorithm 1) on batch updates of increasing size from $10^{-7}|E|$ to $0.1|E|$. Note that both axes are logarithmic. Speedup of P-DF$_L$ with respect to P-ND$_L$ is labeled.



(a) Results of each graph

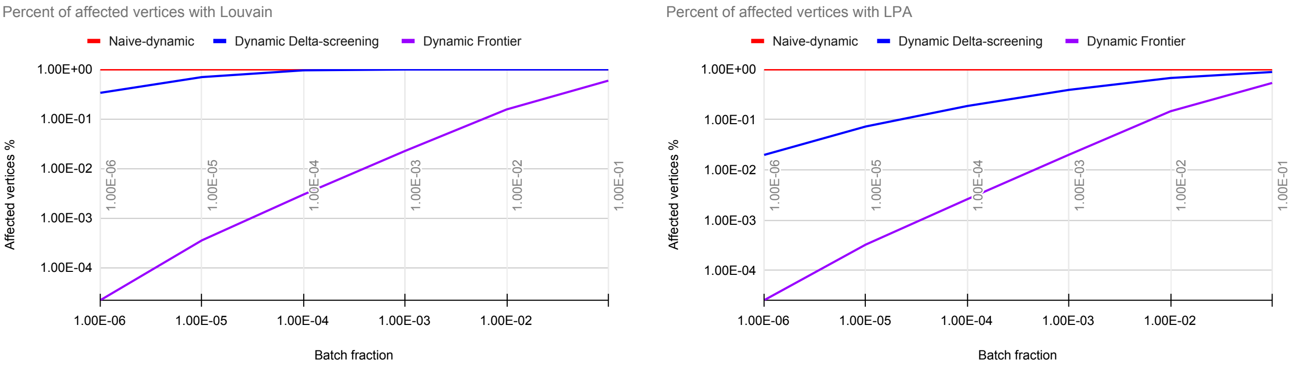
(b) Overall result

Figure 10: Time taken (solid lines), and modularity of communities obtained (dashed lines) along the right Y-axis, with *Static LPA*, P-ND$_{LPA}$, P-DDS$_{LPA}$, and P-DF$_{LPA}$ (Algorithm 2) on batch updates of increasing size from $10^{-7}|E|$ to $0.1|E|$. Note that both axes are logarithmic. Speedup of P-DF$_{LPA}$ with respect to P-ND$_{LPA}$ is labeled.

Performance of Dynamic Δ-screening (P-DDS$_L$). We note that P-DDS$_L$ does not perform better than P-ND$_L$. This is because it tends to mark a large fraction of the vertices as affected, even for small batch updates. Our experiments on the graphs in Table 2 indicate that compared to the P-DF$_L$, the P-DDS$_L$ marks nearly 44× more vertices as affected on a batch size of $10^{-3}|E|$.

Slowdown of static algorithm. Uniform batches of insertions/deletions arbitrarily disrupt the original community structure. This results in *Static Louvain* needing more iterations to converge.

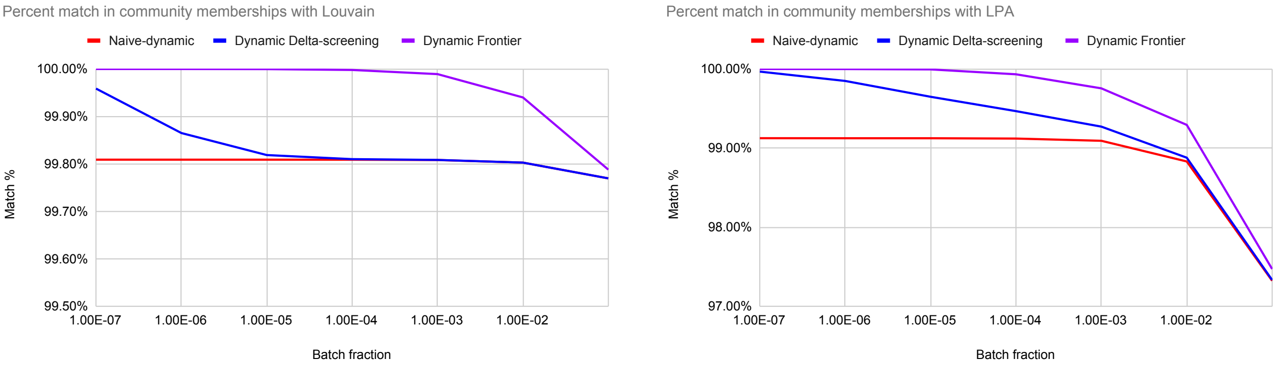
5.2.2 Comparison with respect to Riedy and Bader [52]. Riedy and Bader propose a batch parallel dynamic algorithm for community detection. They compare the run time of their dynamic algorithm to that of a static recomputation. On the graphs `caidaRouterLevel`, `coPapersDBLP`, and `eu-2005`, from [52] and at the batch size of $0.1|E|$, $0.03|E|$, and $0.1|E|$ respectively, they report a speedup of 40×, 1.08×, and 327×, respectively, over their corresponding static algorithm performing a full recomputation. On these three graphs and batch sizes, P-DF$_L$ achieves a speedup of 6.1×, 10.9×, and 4.2×, respectively, compared to a full static recomputation. This might compare unfavorably with the speedups claimed by Riedy and Bader



(a) Dynamic Louvain algorithms

(b) Dynamic LPA algorithms

Figure 11: Percent of vertices marked as affected (mean) with P-ND, P-DDS, and P-DF based *Louvain* and *LPA*, as mentioned in Sections 5.2 and 5.3, on graphs in Table 2.



(a) Dynamic Louvain algorithms

(b) Dynamic LPA algorithms

Figure 12: Percent match in community memberships after reverse and forward batch updates (mean) with P-ND, P-DDS, and P-DF based *Louvain* and *LPA*, as mentioned in Sections 5.2.3 and 5.3.2, on graphs in Table 2.

[52]. However, note from Section 1 that the algorithm of Riedy and Bader [52] does not identify cascading changes to communities. In addition, as their source code is not available, we could not do a more direct comparison.

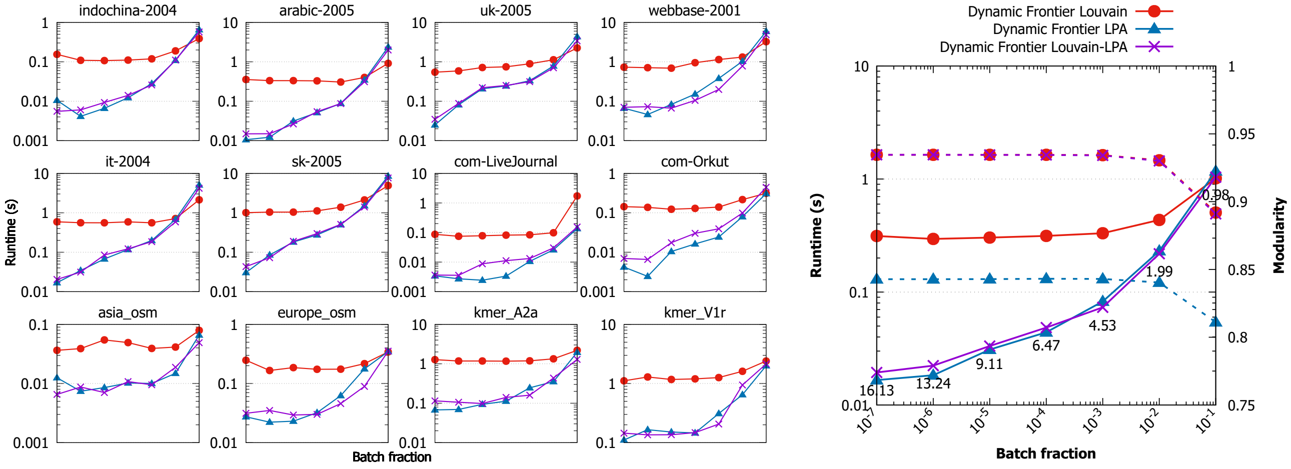
5.2.3 Stability. Intuitively, if the graph at G^t and $G^{t'}$ are identical for some t and t', we expect dynamic community detection algorithms to produce the same communities for G^t and $G^{t'}$. We refer to this property of a dynamic algorithm as its stability and measure it as the percentage of vertices that agree on the community label across two identical graphs.

To measure the stability of P-ND$_\text{L}$, P-DDS$_\text{L}$, and P-DF$_\text{L}$, we proceed as follows. Let G be an initial graph. We generate random batch updates of size $10^{-7}|E|$ to $0.1|E|$ consisting of edge deletions to obtain the graph G^1. We then apply each of the above algorithms on G^1 to identify the new communities. Subsequently, we create another batch of updates that consists of inserting the edges deleted in the prior time step. This graph, G^2, is essentially the original graph G. We obtain the community labels of the vertices in the graph G^2 by appealing to the dynamic algorithms. Finally, we compare the community label of each vertex in the graphs G and G^2. The resulting match in community membership of vertices with P-ND$_\text{L}$, P-DDS$_\text{L}$, and P-DF$_\text{L}$ on batch updates of size $10^{-7}|E|$ to $0.1|E|$ is shown in Figure 12(a).

From Figure 12(a), we observe thatP-ND$_\text{L}$ and P-DDS$_\text{L}$ have minimum of 99.68% match with the original community memberships across all batch sizes, while P-DF$_\text{L}$ has a minimum of 99.70% match. This indicates that all these algorithms are stable.

5.3 Performance of Dynamic Frontier based LPA (P-DF$_\text{LPA}$, Algorithm 2)

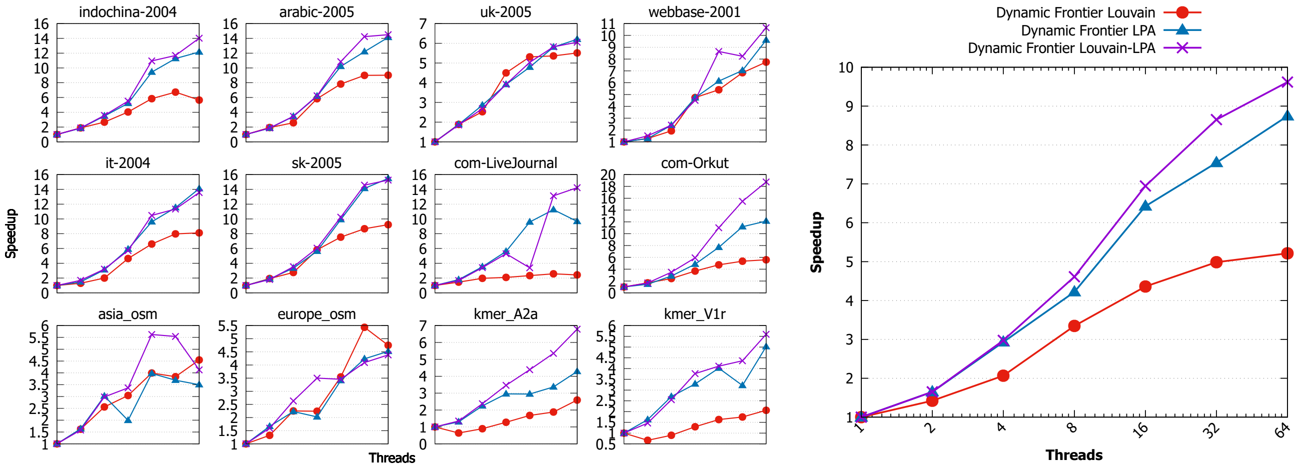
5.3.1 Overall Performance. In this experiment, we study the performance of P-DF$_\text{LPA}$ with batch updates of size ranging from $10^{-7}|E|$ to $0.1|E|$, and compare it to *Static LPA*, P-ND$_\text{LPA}$, and P-DDS$_\text{LPA}$. Unlike *Louvain*, none of the *LPA* based dynamic approaches require



(a) Results of each graph

(b) Overall result

Figure 13: Time taken (solid lines), and modularity of communities obtained (dashed lines) along the right Y-axis, with P-DF$_L$, P-DF$_{LPA}$, and P-DF$_H$ (Algorithms 1, 2, and 3) on batch updates of increasing size from $10^{-7}|E|$ to $0.1|E|$. Note that both axes are logarithmic. Speedup of P-DF$_H$ with respect to P-DF$_L$ is labeled.



(a) Results of each graph

(b) Overall result

Figure 14: Speedup with respect to sequential, of P-DF$_L$, P-DF$_{LPA}$, and P-DF$_H$ (Algorithms 1, 2, and 3) on batch updates of size $10^{-3}|E|$, with increasing number of threads from 1 to 64 in powers of 2. Note that the Y-axis is linear, while the X-axis is logarithmic and captures doubling of threads naturally.

re-initialization of communities every RESTART_LOUVAIN batches. As discussed in Section 5.1.5, we employ 5 distinct random batch updates for every batch size in order to minimize measurement noise.

Figure 10(b) shows the overall result of the experiment. While all approaches obtain communities of equivalent modularity, P-DF$_{LPA}$ converges on average 10.0× faster than P-ND$_{LPA}$ from a batch size of $10^{-7}|E|$ up to $0.01|E|$. As shown in Figure 10(a), it has good performance on *web graphs* and *social networks* (graphs with high $|E|/|V|$ ratio). Note, however, that the quality of communities obtained with *LPA* is not on par with *Louvain*.

Performance of Dynamic Δ-screening. Further, we note that P-DDS$_{LPA}$ generally fails to perform better than P-ND$_{LPA}$. This is because it tends to mark a large fraction of the vertices as affected (see Figure 11(b)), and has a high associated overhead.

Slowdown of static algorithm. As mentioned in Section 5.2, the uniform batches of insertions/deletions arbitrarily disrupt the original community structure, necessitating more iterations for *Static LPA* to converge.

5.3.2 Stability. Just as in Section 5.2.3, we study the stability of P-ND$_{LPA}$, P-DDS$_{LPA}$, and P-DF$_{LPA}$ on random batch updates of size $10^{-7}|E|$ to $0.1|E|$. From the results, shown in Figure 12(b), we observe that P-ND$_{\text{LPA}}$ and P-DDS$_{\text{LPA}}$ have minimum of 95.53% match with the original community memberships across all batch sizes, while P-DF$_{\text{LPA}}$ has a minimum of 95.75% match. This indicates that P-DF$_{\text{LPA}}$ is stable.

5.4 Performance of Dynamic Frontier based Hybrid Louvain-LPA (P-DF$_{\text{H}}$, Algorithm 3)

5.4.1 Overall Performance. We now study the performance of P-DF$_{\text{H}}$, which uses *Static Louvain* every RESTART_HYBRID batches and uses P-DF$_{\text{LPA}}$ for updating communities for the remainder batches (see Sections 3.6 and 3.7). We do this on batch updates of size $10^{-7}|E|$ to $0.1|E|$, and compare it with P-DF$_{\text{L}}$ and P-DF$_{\text{LPA}}$ (Algorithms 1 and 2). As stated in Section 5.1.5, we employ five distinct random batch updates for every batch size to minimize measurement noise.

The modularity of communities obtained by P-DF$_{\text{H}}$ is nearly identical to that obtained by P-DF$_{\text{L}}$, as shown in Figure 13(b), while obtaining a mean speedup of 7.5× across batch sizes of $10^{-7}|E|$ to $0.1|E|$. P-DF$_{\text{H}}$ is thus an efficient and high-quality dynamic community detection approach, especially of *web graphs*, as shown in Figure 13(a), where it significantly outperforms P-DF$_{\text{L}}$ for smaller batch updates. Note that P-DF$_{\text{LPA}}$ has about the same performance, but obtains communities of lower quality.

5.4.2 Scalability. Finally, we study the strong-scaling behavior of P-DF$_{\text{H}}$ (Algorithm 3), and compare it with P-DF$_{\text{L}}$ and P-DF$_{\text{LPA}}$ (Algorithms 1 and 2). To do this, we fix the batch size at $10^{-3}|E|$, vary the number of threads in use from 1 to 64, and measure the speedup of each algorithm to its sequential version. Figure 14 demonstrates scalability of dynamic algorithms with varying thread count, measured as the time taken by the algorithm compared to the same algorithm running on one thread. Again, as indicated in Section 5.1.5, we employ five distinct random batch updates for each batch size to minimize measurement noise.

As shown in Figure 14(b), P-DF$_{\text{L}}$, P-DF$_{\text{LPA}}$, and P-DF$_{\text{H}}$ obtain a speedup of 5.2×, 8.7×, and 9.6× respectively at 64 threads; with their speedup increasing at a mean rate of 1.31×, 1.44×, and 1.46× respectively for every doubling of threads. Also note from Figure 14(a) that P-DF$_{\text{H}}$ offers good speedup on *web graphs* and *social networks*, but does not scale well on *road networks* and *k-mer protein graphs* (which have a low $|E|/|V|$ ratio).

6 RELATED WORK

Identifying hidden communities within networks is a crucial graph analytics problem that arises in various domains such as drug discovery, disease prediction, protein annotation, topic discovery, inferring land use, and criminal identification. The main objective is to identify groups of vertices that exhibit dense internal connections but sparse connections with the rest of the graph [22]. However, this problem is NP-hard, and there is a lack of apriori knowledge on the number and size distribution of communities [8].

To solve this issue, researches have come up with a number of heuristics for finding communities. These include label propagation [22, 48], random walk [55], diffusion [30], spin dynamics [51], fitness metric optimization [17, 43], statistical inference [14, 44], core clustering [57], simulated annealing [23, 51], clique percolation [15, 24], information theory (infomap) [53, 54], and biological evolution (genetics) [21, 38] are studied over the decades for this problem. To evaluate the success of these methods, metrics such as the modularity score [8, 43], Normalized Mutual Information index (NMI) [12, 28], and Jaccard Index [28] are often employed.

The *Louvain* algorithm, based on modularity optimization, employs a greedy strategy to hierarchically merge graph vertices and extract communities [8]. It has a time complexity of $O(KM)$ (where M represents the number of edges in the graph, K represents the total number of iterations performed across all passes), and it efficiently identifies communities with resulting high modularity. As a result, the *Louvain* method is widely favored among researchers [33].

Algorithmic improvements to the original algorithm have been proposed, which include early pruning of non-promising candidates (leaf vertices) [25, 58, 73, 77], attempting local move only on likely vertices [46, 58, 62, 77], ordering of vertices based on node importance [2], moving nodes to a random neighbor community [63], threshold scaling [25, 37, 41], threshold cycling [20], subnetwork refinement [65, 66], multilevel refinement [18, 56, 62], and early termination [20],

To parallelize the algorithm on multicore CPUs, GPUs [10], hybrid CPU-GPUs [7], multi GPUs [6, 10, 19], and distributed systems [6], a number of strategies have been attempted. These include parallelizing the costly first iteration [67], performing iterations asynchronously [47, 62], ordering vertices via graph coloring [25], using vector based hashtables [25], using adaptive parallel thread assignment [16, 40, 41], using sort-reduce instead of hashing [10], using simple partitions based of vertex ids [10, 20], and identifying and moving ghost/doubtful vertices [6, 7, 47, 76].

The *Label Propagation Algorithm (LPA)* is a method used for identifying communities or groups within a network by initializing each vertex with a unique label and diffusing these labels across the graph. It is faster and more scalable than the Louvain algorithm, as it does not require repeated optimization steps and is easy to parallelize [42, 48].

Improvements upon the LPA include using a stable (non-random) mechanism of label choosing in the case of multiple best labels [70], addressing the issue of monster communities [4, 59], identifying central nodes and combining communities for improved modularity [72], and using frontiers with alternating push-pull to reduce the number of edges visited and improve solution quality [36]. A GPU-accelerated parallel implementation of the original LPA is available that is able to deal with large-scale datasets that do not fit into GPU memory [32].

A growing number of research efforts have focused on detecting communities in dynamic networks. The simplest approach is to use the community membership of vertices from the previous snapshot of the graph [3, 11, 60, 79] (which we call *Naive-dynamic*). Alternatively, more advanced techniques have been employed to minimize computation by identifying a smaller subset of the graph that is affected by changes, such as moving only changed vertices [1, 71], recomputing vertices close to an updated edge (below a given threshold distance) [27], disbanding affected communities to lower-level network [13], or using a dynamic modularity metric to compute community membership of vertices from scratch [39].

Delta-Screening (or Δ*-screening*) is a recently proposed technique that finds a subset of vertices impacted by changes in a graph using delta-modularity [75].

Significant research effort has also been dedicated to the development of dynamic label-propagation methods, due to their simplicity, efficiency, and scalability. In addition to the *Naive-dynamic* approach, a number of advanced techniques been proposed. These include using the MapReduce model to efficiently adjust the communities of certain vertices based on previous intervals [35], and using a stabilized label propagation process based on the static LabelRank algorithm [68]. Adaptive Label Propagation Algorithm (ALPA) is another dynamic approach, which first performs a warm-up LPA on a subset of the network determined by edge deletions and insertions, followed by Local Label Propagation (LLP) which expands as a frontier of nodes that change labels and removes nodes that do not change labels [26].

7 CONCLUSION

This study addressed the design of a high-speed community detection algorithm in the batch dynamic setting. First, we presented our optimized parallel implementations of the *Louvain* and *LPA* algorithms. These implementations identified communities in 6.2 seconds and 2.7 seconds, respectively, on a single 64-core CPU when processing an undirected web graph with 1.9 billion edges.

Next, we discussed our *Dynamic Frontier* approach (P-DF). Given a batch update of edge deletions and insertions, this approach addresses the issue of finding and processing only an appropriate set of affected vertices with minimal overhead. We tested it using parallel implementation of the *Louvain* (P-DF$_{\text{L}}$) and *LPA* (P-DF$_{\text{LPA}}$) algorithms, compared it to two other dynamic approaches, *Naive-dynamic* (P-ND) and *Dynamic* Δ*-screening* (P-DDS) [75], and demonstrated an improved performance of up to 1.5× with *Louvain* (P-DF$_{\text{L}}$) and 10.0× with *LPA* (P-DF$_{\text{LPA}}$), compared to the best of the other two dynamic approaches.

Finally, we presented our novel *Dynamic Frontier*-based *Hybrid Louvain-LPA* (P-DF$_{\text{H}}$) that combines *Louvain* and *LPA* into a hybrid method. It leverages the advantages of both algorithms and addresses their respective limitations. We show that this approach produced high-quality results while being 7.5× faster than *Dynamic Frontier*-based *Louvain* (P-DF$_{\text{L}}$).

Our *Dynamic Frontier* approach (P-DF) incrementally identifies fewer affected vertices compared to *Dynamic* Δ*-screening* (P-DDS), and converges in fewer iterations than static algorithms. This translates to decreased workload, leading us to anticipate similar performance benefits across other shared memory programming models. We plan to explore similar algorithms on GPUs and other multiprocessor systems.

REFERENCES

[1] R. Aktunc, I. Toroslu, M. Ozer, and H. Davulcu. 2015. A dynamic modularity based community detection algorithm for large-scale networks: DSLM. In *Proceedings of the IEEE/ACM international conference on advances in social networks analysis and mining.* 1177–1183.

[2] A. Aldabobi, A. Sharieh, and R. Jabri. 2022. An improved Louvain algorithm based on Node importance for Community detection. *Journal of Theoretical and Applied Information Technology* 100, 23 (2022), 1–14.

[3] T. Aynaud and J. Guillaume. 2010. Static community detection algorithms for evolving networks. In *8th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks.* IEEE, IEEE, Avignon, France, 513–519.

[4] K. Berahmand and A. Bouyer. 2018. LP-LPA: A link influence-based label propagation algorithm for discovering community structures in networks. *International Journal of Modern Physics B* 32, 06 (10 mar 2018), 1850062. http://www.worldscientific.com/doi/abs/10.1142/S0217979218500625

[5] S. Bhattacharya, D. Chakrabarty, M. Henzinger, and D. Nanongkai. 2018. Dynamic algorithms for graph coloring. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms.* SIAM, 1–20.

[6] A. Bhowmick, S. Vadhiyar, and V. PV. 2022. Scalable multi-node multi-GPU Louvain community detection algorithm for heterogeneous architectures. *Concurrency and Computation: Practice and Experience* 34, 17 (2022), 1–18.

[7] A. Bhowmik and S. Vadhiyar. 2019. HyDetect: A Hybrid CPU-GPU Algorithm for Community Detection. In *IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC).* IEEE, Goa, India, 2–11.

[8] V. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (Oct 2008), P10008.

[9] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, and D. Wagner. 2007. On modularity clustering. *IEEE transactions on knowledge and data engineering* 20, 2 (2007), 172–188.

[10] C. Cheong, H. Huynh, D. Lo, and R. Goh. 2013. Hierarchical Parallel Algorithm for Modularity-Based Community Detection Using GPUs. In *Proceedings of the 19th International Conference on Parallel Processing* (Aachen, Germany) *(Euro-Par'13).* Springer-Verlag, Berlin, Heidelberg, 775–787.

[11] W. Chong and L. Teow. 2013. An incremental batch technique for community detection. In *Proceedings of the 16th International Conference on Information Fusion.* IEEE, IEEE, Istanbul, Turkey, 750–757.

[12] P. Chopade and J. Zhan. 2017. A Framework for Community Detection in Large Networks Using Game-Theoretic Modeling. *IEEE Transactions on Big Data* 3, 3 (Sep 2017), 276–288.

[13] M. Cordeiro, R. Sarmento, and J. Gama. 2016. Dynamic community detection in evolving networks using locality modularity optimization. *Social Network Analysis and Mining* 6, 1 (2016), 1–20.

[14] E. Côme and P. Latouche. 2015. Model selection and clustering in stochastic block models based on the exact integrated complete data likelihood. *Statistical Modelling* 15 (3 2015), 564–589. Issue 6. https://doi.org/10.1177/1471082X15577017 doi: 10.1177/1471082X15577017.

[15] I. Derényi, G. Palla, and T. Vicsek. 2005. Clique percolation in random networks. *Physical review letters* 94, 16 (2005), 160202.

[16] M. Fazlali, E. Moradi, and H. Malazi. 2017. Adaptive parallel Louvain community detection on a multicore platform. *Microprocessors and microsystems* 54 (Oct 2017), 26–34.

[17] S. Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3-5 (2010), 75–174.

[18] O. Gach and J. Hao. 2014. Improving the Louvain algorithm for community detection with modularity maximization. In *Artificial Evolution: 11th International Conference, Evolution Artificielle, EA , Bordeaux, France, October 21-23, . Revised Selected Papers 11.* Springer, Springer, Bordeaux, France, 145–156.

[19] N. Gawande, S. Ghosh, M. Halappanavar, A. Tumeo, and A. Kalyanaraman. 2022. Towards scaling community detection on distributed-memory heterogeneous systems. *Parallel Comput.* 111 (2022), 102898.

[20] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarria-Miranda, A. Khan, and A. Gebremedhin. 2018. Distributed louvain algorithm for graph community detection. In *IEEE international parallel and distributed processing symposium (IPDPS).* IEEE, IEEE, Vancouver, British Columbia, Canada, 885–895.

[21] A. Ghoshal, N. Das, S. Bhattacharjee, and G. Chakraborty. 2019. A fast parallel genetic algorithm based approach for community detection in large networks. In *11th International Conference on Communication Systems & Networks (COMSNETS).* IEEE, Bangalore, India, 95–101.

[22] S. Gregory. 2010. Finding overlapping communities in networks by label propagation. *New Journal of Physics* 12 (10 2010), 103018. Issue 10.

[23] R. Guimera and L. Amaral. 2005. Functional cartography of complex metabolic networks. *nature* 433, 7028 (2005), 895–900.

[24] S. Gupta, D. Singh, and J. Choudhary. 2022. A review of clique-based overlapping community detection algorithms. *Knowledge and Information Systems* 64, 8 (2022), 2023—-2058.

[25] M. Halappanavar, H. Lu, A. Kalyanaraman, and A. Tumeo. 2017. Scalable static and dynamic community detection using Grappolo. In *IEEE High Performance Extreme Computing Conference (HPEC).* IEEE, Waltham, MA USA, 1–6.

[26] J. Han, W. Li, L. Zhao, Z. Su, Y. Zou, and W. Deng. 2017. Community detection in dynamic networks via adaptive label propagation. *PloS one* 12, 11 (2017), e0188655.

[27] P. Held, B. Krause, and R. Kruse. 2016. Dynamic clustering in social networks using louvain and infomap method. In *Third European Network Intelligence Conference (ENIC).* IEEE, IEEE, Wroclaw, Poland, 61–68.

[28] A. Jain, R. Nasre, and B. Ravindran. 2017. DCEIL: Distributed Community Detection with the CEIL Score. In *IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference*

on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS). IEEE, Bangkok, Thailand, 146–153.

[29] A. Khanda, S. Srinivasan, S. Bhowmick, B. Norris, and S. Das. 2021. A parallel algorithm template for updating single-source shortest paths in large-scale dynamic networks. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 929–940.

[30] K. Kloster and D. Gleich. 2014. Heat kernel based community detection. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, New York, USA, 1386–1395.

[31] S. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. Davis, M. Henderson, Y. Hu, and R. Sandstrom. 2019. The SuiteSparse matrix collection website interface. *The Journal of Open Source Software* 4, 35 (Mar 2019), 1244.

[32] Y. Kozawa, T. Amagasa, and H. Kitagawa. 2017. GPU-Accelerated Graph Clustering via Parallel Label Propagation. In *Proceedings of the ACM on Conference on Information and Knowledge Management - CIKM '17*. ACM Press, New York, New York, USA, 567–576.

[33] A. Lancichinetti and S. Fortunato. 2009. Community detection algorithms: a comparative analysis. *Physical Review. E, Statistical, Nonlinear, and Soft Matter Physics* 80, 5 Pt 2 (Nov 2009), 056117.

[34] J. Leskovec. 2021. CS224W: Machine Learning with Graphs | 2021 | Lecture 13.3 - Louvain Algorithm. https://www.youtube.com/watch?v=0zuiLBOIcsw

[35] G. Li, K. Guo, Y. Chen, L. Wu, and D. Zhu. 2017. A dynamic community detection algorithm based on parallel incremental related vertices. In *IEEE 2nd International Conference on Big Data Analysis (ICBDA)*. IEEE, IEEE, Beijing, China, 779–783.

[36] X. Liu, M. Halappanavar, K. Barker, A. Lumsdaine, and A. Gebremedhin. 2020. Direction-optimizing label propagation and its application to community detection. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*. ACM, New York, NY, USA, 192–201.

[37] H. Lu, M. Halappanavar, and A. Kalyanaraman. 2015. Parallel heuristics for scalable community detection. *Parallel computing* 47 (Aug 2015), 19–37.

[38] Y. Lu and G. Chakraborty. 2020. Improving Efficiency of Graph Clustering by Genetic Algorithm Using Multi-Objective Optimization. *International Journal of Applied Science and Engineering* 17, 2 (Jun 2020), 157–173.

[39] X. Meng, Y. Tong, X. Liu, S. Zhao, X. Yang, and S. Tan. 2016. A novel dynamic community detection algorithm based on modularity optimization. In *7th IEEE international conference on software engineering and service science (ICSESS)*. IEEE, IEEE, Beijing,China, 72–75.

[40] M. Mohammadi, M. Fazlali, and M. Hosseinzadeh. 2020. Accelerating Louvain community detection algorithm on graphic processing unit. *The Journal of supercomputing* (Nov 2020).

[41] M. Naim, F. Manne, M. Halappanavar, and A. Tumeo. 2017. Community detection on the GPU. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Orlando, Florida, USA, 625–634.

[42] M. Newman. 2004. Detecting community structure in networks. *The European Physical Journal B - Condensed Matter* 38, 2 (Mar 2004), 321–330.

[43] M. Newman. 2006. Finding community structure in networks using the eigenvectors of matrices. *Physical review E* 74, 3 (2006), 036104.

[44] M. Newman and G. Reinert. 2016. Estimating the number of communities in a network. *Physical review letters* 117, 7 (2016), 078301.

[45] OpenMP Architecture Review Board. 2018. OpenMP Application Program Interface Version 5.0. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[46] N. Ozaki, H. Tezuka, and M. Inaba. 2016. A simple acceleration method for the Louvain algorithm. *International Journal of Computer and Electrical Engineering* 8, 3 (2016), 207.

[47] X. Que, F. Checconi, F. Petrini, and J. Gunnels. 2015. Scalable community detection with the louvain algorithm. In *IEEE International Parallel and Distributed Processing Symposium*. IEEE, IEEE, Hyderabad, India, 28–37.

[48] U. Raghavan, R. Albert, and S. Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E* 76, 3 (Sep 2007), 036106–1–036106–11.

[49] G. Ramalingam. 1996. Bounded Incremental Computation. *Lecture Notes in Computer Science* 1089 (1996), 101–129.

[50] S. Regunta, S. Tondomker, K. Shukla, and K. Kothapalli. 2021. Efficient parallel algorithms for dynamic closeness-and betweenness centrality. *Concurrency and Computation: Practice and Experience* 0, 0 (2021), 1–22.

[51] J. Reichardt and S. Bornholdt. 2006. Statistical mechanics of community detection. *Physical review E* 74, 1 (2006), 016110.

[52] Jason Riedy and David A Bader. 2013. Multithreaded community monitoring for massive streaming graph data. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 1646–1655.

[53] L. Rita. 2020. Infomap Algorithm. An algorithm for community finding. https://towardsdatascience.com/infomap-algorithm-9b68b7e8b86

[54] M. Rosvall, D. Axelsson, and C. Bergstrom. 2009. The map equation. *The European Physical Journal Special Topics* 178, 1 (Nov 2009), 13–23.

[55] M. Rosvall and C. Bergstrom. 2008. Maps of random walks on complex networks reveal community structure. *Proceedings of the national academy of sciences* 105, 4 (2008), 1118–1123.

[56] R. Rotta and A. Noack. 2011. Multilevel local search algorithms for modularity clustering. *Journal of Experimental Algorithmics (JEA)* 16 (2011), 2–1.

[57] Y. Ruan, D. Fuhry, J. Liang, Y. Wang, and S. Parthasarathy. 2015. *Community discovery: simple and scalable approaches*. Springer International Publishing, Cham, 23–54.

[58] S. Ryu and D. Kim. 2016. Quick community detection of big graph data using modified louvain algorithm. In *IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, Sydney, NSW, 1442–1445.

[59] M. Sattari and K. Zamanifar. 2018. A spreading activation-based label propagation algorithm for overlapping community detection in dynamic social networks. *Data & knowledge engineering* 113 (Jan 2018), 155–170.

[60] J. Shang, L. Liu, F. Xie, Z. Chen, J. Miao, X. Fang, and C. Wu. 2014. A real-time detecting algorithm for tracking community structure of dynamic networks.

[61] Z. Shao, N. Guo, Y. Gu, Z. Wang, F. Li, and G. Yu. 2020. Efficient closeness centrality computation for dynamic graphs. In *Database Systems for Advanced Applications: 25th International Conference, DASFAA , Jeju, South Korea, September 24–27, , Proceedings, Part II*. Springer, 534–550.

[62] J. Shi, L. Dhulipala, D. Eisenstat, J. Łącki, and V. Mirrokni. 2021. Scalable community detection via parallel correlation clustering.

[63] V. Traag. 2015. Faster unfolding of communities: Speeding up the Louvain algorithm. *Physical Review E* 92, 3 (2015), 032801.

[64] V. Traag, P. Dooren, and Y. Nesterov. 2011. Narrow scope for resolution-limit-free community detection. *Physical Review E* 84, 1 (2011), 016114.

[65] V. Traag, L. Waltman, and N. Eck. 2019. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific Reports* 9, 1 (Mar 2019), 5233.

[66] L. Waltman and N. Eck. 2013. A smart local moving algorithm for large-scale modularity-based community detection. *The European physical journal B* 86, 11 (2013), 1–14.

[67] C. Wickramaarachchi, M. Frincu, P. Small, and V. Prasanna. 2014. Fast parallel algorithm for unfolding of communities in large graphs. In *IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, IEEE, Waltham, MA USA, 1–6.

[68] J. Xie, M. Chen, and B. Szymanski. 2013. LabelrankT: Incremental community detection in dynamic networks via label propagation. In *Proceedings of the Workshop on Dynamic Networks Management and Mining*. ACM, New York, USA, 25–32.

[69] J. Xie, B. Szymanski, and X. Liu. 2011. SLPA: Uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process. In *IEEE 11th International Conference on Data Mining Workshops*. IEEE, IEEE, Vancouver, Canada, 344–349.

[70] Y. Xing, F. Meng, Y. Zhou, M. Zhu, M. Shi, and G. Sun. 2014. A node influence based label propagation algorithm for community detection in networks. *The Scientific World Journal* 2014 (2014), 1–14.

[71] S. Yin, S. Chen, Z. Feng, K. Huang, D. He, P. Zhao, and M. Yang. 2016. Node-grained incremental community detection for streaming networks. In *IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 585–592.

[72] X. You, Y. Ma, and Z. Liu. 2020. A three-stage algorithm on community detection in social networks. *Knowledge-Based Systems* 187 (2020), 104822.

[73] Y. You, L. Ren, Z. Zhang, K. Zhang, and J. Huang. 2022. Research on improvement of Louvain community detection algorithm. In *2nd International Conference on Artificial Intelligence, Automation, and High-Performance Computing (AIAHPC)*, Vol. 12348. SPIE, Zhuhai, China, 527–531.

[74] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang. 2017. Effective and efficient dynamic graph coloring. *Proceedings of the VLDB Endowment* 11, 3 (2017), 338–351.

[75] N. Zarayeneh and A. Kalyanaraman. 2021. Delta-Screening: A Fast and Efficient Technique to Update Communities in Dynamic Graphs. *IEEE transactions on network science and engineering* 8, 2 (Apr 2021), 1614–1629.

[76] J. Zeng and H. Yu. 2015. Parallel Modularity-Based Community Detection on Large-Scale Graphs. In *IEEE International Conference on Cluster Computing*. IEEE, 1–10.

[77] J. Zhang, J. Fei, X. Song, and J. Feng. 2021. An improved Louvain algorithm for community detection. *Mathematical Problems in Engineering* 2021 (2021), 1–14.

[78] X. Zhang, F.T. Chan, H. Yang, and Y. Deng. 2017. An adaptive amoeba algorithm for shortest path tree computation in dynamic graphs. *Information Sciences* 405 (2017), 123–140.

[79] D. Zhuang, J. Chang, and M. Li. 2019. DynaMo: Dynamic community detection by incrementally maximizing modularity. *IEEE Transactions on Knowledge and Data Engineering* 33, 5 (2019), 1934–1945.

A REPRODUCIBILITY

We now describe the computational artifact for the paper **Shared-Memory Parallel Algorithms for Community Detection in Dynamic Graphs**. It includes the source code for *three experiments* and the source code for *generating plots* in four separate directories.

1. `louvain-communities-openmp-dynamic/` contains the source code for the experiment which compares the performance of *Static*, *Naive-dynamic*, *Dynamic Delta-screening*, and *Dynamic Frontier* based *Louvain*.
2. `rak-communities-openmp-dynamic/` contains the source code for the experiment which compares the performance of *Static*, *Naive-dynamic*, *Dynamic Delta-screening*, and *Dynamic Frontier* based *LPA* (aka *RAK*).
3. `communities-openmp-dynamic/` contains the source code for the experiment which compares the performance of *Dynamic Frontier* based *Louvain*, *LPA*, and *Hybrid Louvain-LPA*. It also includes the script (and steps) to run the strong scaling experiment.
4. `gnuplot-scripts-communities-cpu/` contains the source code for generating the plots for the experiments.

A.1 Dependencies and requirements

We run all experiments on a server that has an AMD EPYC-7742 64-bit processor. This processor has a clock frequency of 2.25 GHz and 512 GB of DDR4 system memory. The CPU has 64 x86 cores. Each core has L1 cache of 4 MB, L2 cache of 32 MB, and a shared L3 cache of 256 MB. The machine runs on *Ubuntu 20.04*. It is possible to run the experiment on any 64-bit system running a recent version of Linux by configuring the number of threads to use for the experiment. We use *GCC 9.4* and *OpenMP 5.0* to compile with optimization level 3 (`-O3`). Executing the build and run script requires `bash`. Additionally, *Node.js 18 LTS* is needed to process generated output into *CSV*, *Google sheets* is needed to generate charts and summarized CSVs, and *gnuplot 5.4* is needed to generate the plot from summarized CSVs.

We use `13` graphs in *Matrix Market (.mtx)* file format from the *SuiteSparse Matrix Collection* as our input dataset. These must be placed in the `~/Data` directory **before running** the experiments. In addition, a `~/Logs` directory **must be created**, where the output logs of each experiment are written to. Please use `setup.sh` in the current directory to create the necessary directories and download the input dataset. The graphs in the **input dataset** are as follows:

```
indochina-2004.mtx
uk-2002.mtx
arabic-2005.mtx
uk-2005.mtx
webbase-2001.mtx
it-2004.mtx
sk-2005.mtx
com-LiveJournal.mtx
com-Orkut.mtx
asia_osm.mtx
europe_osm.mtx
kmer_A2a.mtx
kmer_V1r.mtx
```

A.2 Installation and deployment process

Each experiment includes a `mains.sh` file which needs to be **executed** in order to run the experiment. To run an experiment, try the following:

```
# Run experiment with a default of 64 threads
$ DOWNLOAD=0 ./mains.sh

# Run experiment with 32 threads
$ DOWNLOAD=0 MAX_THREADS=32 ./mains.sh
```

Please refer to any additional details in the `README.md` of each experiment. Output logs are written to the `~/Logs` directory. These logs can be processed with the `process.js` script to generate a *CSV* file as follows:

```
$ node process.js csv ~/Logs/"experiment".log "experiment".csv
```

The generated *CSV* file can be loaded into the `data` sheet of the linked **sheets** document in the respective experiment. Ensure that there are no newlines at the end of the `data` sheet after loading. All the charts are then automatically updated. See `"graph"` sheet for results on a specific input graph, or the `all` sheet for the average result on all input graphs. You can then use the `csv` sheet to retrieve *summarized CSVs* which can be used to generate plots using the `gnuplot` scripts in the `gnuplot-scripts-communities-cpu` directory.

A.3 Reproducibility of Experiments

The workflow of each experiment is as follows:

1. Setup the necessary directories and download the input dataset with `setup.sh`.
2. Run an experiment of choice with `DOWNLOAD=0 ./mains.sh` in respective subdirectory.
3. Output of the experiment is written to `~/Logs` directory.
4. Process the output logs into CSV with `node process.js csv ~/Logs/"experiment".log "experiment".csv`.
5. Import the CSV into the `data` sheet of the linked **sheets** document of the experiment.
6. All the charts are automatically updated. See `"graph"` sheet for results on a specific input graph, or the `all` sheet for the average result on all input graphs.
7. Use the `csv` sheet to retrieve summarized CSVs.
8. Use the summarized CSVs to generate plots using the `gnuplot` scripts in the `gnuplot-scripts-communities-cpu` subdirectory.
9. Compare the generated plots with that of the paper.