International Journal of Parallel Programming (2025) 53:4
https://doi.org/10.1007/s10766-024-00780-1

DyG-DPCD: A Distributed Parallel Community Detection Algorithm for Large-Scale Dynamic Graphs

Naw Safrin Sattar[1] · Khaled Z. Ibrahim[2] · Aydin Buluc[2] · Shaikh Arifuzzaman[3]

Received: 7 April 2024 / Accepted: 1 October 2024 / Published online: 19 November 2024

Abstract
Dynamic (Temporal) graphs capture the valuable evolution of real-world systems, from the continuously evolving patterns of social interactions and genetic pathways to the dynamic fluctuations of economic forces. Detecting communities for such evolving networks poses unique challenges. Detecting and analyzing the evolution of communities within dynamic graphs unlocks valuable insights into the underlying structural and temporal patterns of real-world systems. However, the sheer volume of modern graph data and the inherent complexity of the temporal dimension pose significant challenges to scalable community detection algorithms. Addressing this gap, our work explores the limited landscape of scalable distributed-memory parallel methods specifically designed for dynamic network community detection. We propose a novel parallel algorithm, DyG-DPCD (Dynamic Graph Distributed Parallel Community Detection), to detect communities in dynamic networks using the Message Passing Interface (MPI) framework. We present a vertex-centric approach, allowing us to detect communities through local optimization. Furthermore, we enhance our baseline algorithm by incorporating three heuristics, which improve the algorithm's performance significantly while maintaining the quality of the solutions. We demonstrate the efficiency of our algorithm by experimenting on several real-world large-scale networks with hundreds of millions of edges spanning diverse domains. Notably, DyG-DPCD achieves speedups between 25× and 30× for large networks that we experimented on using NERSC compute nodes. Our algorithm outperforms the STINGER parallel re-agglomeration algorithm by 30×.

Keywords Dynamic graphs · Temporal graphs · Community detection · Parallel algorithm · Distributed-memory · MPI

Extended author information available on the last page of the article

1 Introduction

Graphs, as a powerful representation of real-world complex systems, have been extensively studied in various domains [1–9]. Traditional network mining often focuses on static topological properties to analyze and understand these systems [3, 10–12]. However, when dealing with real- world complex systems such as those found in social systems (e.g., Facebook, Twitter), brain connections, and transportation systems, it becomes essential to account for the dynamic nature of these networks over time [7, 13, 14]. Incorporating temporal dynamics is crucial to accurately represent the evolving nature of these complex systems [6, 15].

The community detection problem, a fundamental task in graph mining, aims to identify groups of nodes that are densely connected within themselves but sparsely connected to other groups. These communities, often referred to as “communities" or “clusters", can represent various real-world phenomena, such as social groups, functional modules in biological networks, or topic-related communities in knowledge graphs. While numerous algorithms have been developed for static community detection [2, 3, 5, 16–18], finding efficient and effective methods for detecting temporal communities in dynamic graphs remains a challenging research problem [19]. A few existing approaches use the modularity [20] metric using different variants of the Louvain method [21, 22]. Maximizing parallel weighted community clustering (PWCC) metric is used with incremental vertices [15]. Permanence is another metric which requires initial network information and community structure to detect communities in the subsequent snapshots [14]. A find-and-merge approach with streaming updates is one of the other methods. A distance-based metric on the inertia of inter-node relationships helps finding temporal communities [23].

Temporal community detection involves identifying communities that evolve over time, reflecting the changing relationships and interactions among nodes in the network. Existing approaches often rely on extending static community detection algorithms to handle temporal dynamics, or by introducing new metrics that capture the temporal aspects of the network. It is computationally expensive to directly employ a well-studied static algorithm repeatedly on the network snapshots of the evolving networks. Further, existing works are mostly done on synthetic networks and few works focus on large-scale networks. Overall, these methods face limitations in terms of scalability [24], computational efficiency, and accuracy, especially when dealing with large-scale dynamic networks. To the best of our knowledge, there exist no distributed-memory parallel algorithms using Message Passing Interface (MPI) to detect communities in large dynamic graphs.

In this paper, we address the challenges of temporal community detection by proposing a novel distributed-memory parallel algorithm, DyG-DPCD. Our algorithm leverages the *permanence* metric, a vertex-centric measure that allows for local optimization and avoids arbitrary tie-breaking issues commonly encountered in other approaches. However, parallelizing a temporal community detection approach in a distributed-memory setting is challenging for a number of reasons including identifying appropriate data structure that supports graph updates, the need for synchronization, and message overhead. We addressed such challenges and employed several heuristics for DyG-DPCD to efficiently handle large-scale dynamic networks and provides a scalable solution for temporal community detection.

Our Contributions. The main contributions of this paper are as follows:

- We present DyG-DPCD, a distributed-memory parallel community detection algorithm for dynamic (temporal) networks. We design and evaluate multiple heuristics that further improve the performance of our baseline parallel algorithm making the algorithm scalable for large graphs.
- We show the performance evaluation and assessment of the correctness of our solutions on several large-scale real-world networks (with up to 234M edges) from six different domains (social, blog, communication, collaboration, citation, biological). Such comprehensive experimentation allows us to evaluate our algorithms on a multitude of different graph topologies.
- We also present a comparative analysis of our algorithm with existing parallel community detection algorithms for dynamic graphs. Our algorithm outperforms the limited prior works significantly.

2 Related Works

Some recent works on dynamic graphs focus on providing a framework [1, 4, 7, 25–29] for storing and managing the networks with high performance and efficiency to support different graph algorithms on GPUs and other various HPC platforms. Some of the parallel algorithms applied on dynamic networks include breadth-first search (BFS) traversal [30], finding subgraphs [8, 31], shortest path [32], k-cores [33], triangle-count [34] and others [35–37].

Community detection has been extensively studied for static networks [38–42]. However, the focus has shifted towards dynamic community detection in recent years. Very few works have been done on the parallel community detection algorithm for dynamic networks. Some approaches extend modularity-based [21, 22] algorithms to handle temporal dynamics, while others use different metrics like partition distance or permanence [14]. Halappanavar et. al [5] have presented a technique to detect communities on dynamic graphs and future work includes implementing the method in their shared-memory-based parallel framework Grappolo [17]. The implementation and experimental analysis of the mentioned technique are not available yet. A shared-memory-based multi-threaded community detection method for streaming graphs is presented in [43]. The authors implemented an incremental re-agglomeration algorithm using the STINGER framework and Open Multi-Processing (OpenMP) that considers a batch of changes to detect communities. The work reports improvement over static re-computation; however, parallel speedup and scaling were not reported.

The only other parallel algorithm for detecting communities in dynamic networks is a Spark-based implementation that shows only 2× parallel speedup with 12 cores

[15]. The algorithm identifies incremental vertices and defines community membership depending on maximizing PWCC metric of the entire network. However, the experimented graphs are mostly synthetic and relatively small with at most 10 M edges. The only real-world network they use have 200K vertices and 500K edges. For the synthetic networks, the degree distribution and the community size is very small which mostly does not represent the real-world dynamic networks like large-scale social networks those are highly skewed.

Addressing the limitations of existing approaches, this paper presents DyG-DPCD, a distributed-memory parallel community detection algorithm for dynamic graphs. DyG-DPCD utilizes the permanence metric, a vertex-centric measure that allows for local optimization. The algorithm addresses challenges related to data structure, synchronization, and message overhead in a distributed-memory setting.

3 Preliminaries

We present the notations, problem definition, and computational model used in our solution below.

3.1 Notation

We denote the input dynamic graph (network) as a collection of multiple graph snapshots over time. We consider that multiple edges appear and disappear in different time frames. Therefore, the full dynamic network can be represented as $G = G_0 \cup G_1 \cup G_2 \ldots \cup G_t \cup G_{t+1}$. A network snapshot at time t is denoted by $G_t(V_t, E_t)$, where V_t and E_t are the sets of vertices and edges, respectively, at time t. Each edge is a tuple of the form (u, v, t) where u and v are source and destination vertices in the temporal graph, and t is a timestamp. We use the words node and vertex interchangeably as well as links and edges. P is the number of processors used in the computation, denoted by $P_0, P_1, \ldots . P_{N-1}$ where $0, 1, 2, \ldots . N-1$ refers

Table 1 Terminology used in the paper

Symbol	Meaning
$G(V, E)$	Graph with V = set of vertices and E = set of edges
G_t	Graph at time t
C_t	Community structure of network at time t
$C(v)$	Community of vertex, v
$d(v)$	Degree of vertex, v
d_{max}	Maximum degree
$\overline{d}$	Average degree
e_{intra}	Intra community edge: $(u, v) \in e, C(u) = C(v)$
e_{inter}	Inter community edge: $(u, v) \in e, C(u) \neq C(v)$
N	Total number of processors (World size)

to the rank of a processor. Notations frequently used throughout the paper are listed in Table 1.

3.2 Problem Definition

Detecting Temporal (Dynamic) Communities: Let $G_t(V_t, E_t)$ be the graph at time step t. Our goal is to compute the communities C_k at time step k in a distributed manner, based on the community structure C_{k-1} at time step $k-1$ and the changed edge set ΔE_k.

3.3 Community Detection through Vertex-centric Optimization

We use *permanence*, a local vertex-based metric, to extract the community structure of large networks based on our prior work [6]. The advantage of this method lies in its ability to be computed locally without necessitating global optimization like modularity [6]. As demonstrated in [14], it surpasses widely-used community detection techniques in terms of runtime complexity. Permanence of a vertex v, $Perm(v)$, is calculated using Eq. 1, where $-1 \leq Perm \leq 1$.

$$Perm(v) = \left[\frac{I(v)}{E_{max}(v)} \times \frac{1}{d(v)}\right] - \left[1 - CC_{in}(v)\right] \tag{1}$$

Here, $I(v)$, $E_{max}(v)$, and $CC_{in}(v)$ denote internal neighbors, the maximum number of connections to a single external community, and the internal clustering coefficient of vertex v, respectively. $CC_{in}(v) = \frac{x}{CC(I(v),2)}$ where, x is existing connections among the internal neighbors of v, and $CC(I(v), 2)$ is the total number of possible connections among the internal neighbors of v. $Perm(v) = 1$ means v is cohesively connected to the assigned community. $Perm(v) = 0$ signifies that v is equally pulled by all the neighbors or forms a singleton community. $Perm(v) = -1$ indicates a wrong community assignment. We ensure the permanence maximization for community assignments during the edge updates (additions and deletions). The permanence value of a community, $Perm(comm)$ is the average of the permanence value of all its member vertices $(v_1, v_2, ...v_k)$ given in Eq. 2.

$$Perm(comm) = \frac{\sum_{i=1}^{k} Perm(v_i)}{k} \tag{2}$$

The selection of the metrics poses a challenging question with multiple feasible metrics considerations. We choose ‘permanence’ to be a good measure based on its definitive characteristic for better output community quality depending on the maximum external pull to an individual community, instead of the total external connection, and also the high connectivity among the internal neighbors to form a tightly knit internal substructure. As we are considering the parallelization in a dynamic setting, we get the advantage of the fact that the total edge weight of the entire graph is not needed for permanence computation, as required in the modularity metric. Again, tracking the total number of edges is difficult when all processors are handling the batch edge additions and deletions locally. Our proposed parallelization could be used with other metrics i.e. modularity, but with some additional communication cost to track these updated edges for modularity maximization.

3.4 Computational Model

We develop the parallel algorithm for Message Passing Interface (MPI) based distributed-memory parallel systems, where each processor has its own local memory. The processors do not have any shared memory, one processor cannot directly access the local memory of another processor, and the processors communicate via exchanging messages using MPI.

4 Methodology

In this section, we describe our algorithm DyG-DPCD, the implementation challenges, the complexity analysis of the algorithm, and the heuristics we use to improve the performance.

4.1 Overview of our Distributed-Memory Algorithm

We develop a distributed-memory algorithm capable of computing communities in a network through a distributed approach. We carefully distribute the entire network among different processors, ensuring minimal communication between them during computation. After handling a mini-batch of edge additions and deletions, we perform collective gather and broadcast operations to keep the community list and network connectivity up to date.

Our approach follows an incremental strategy, where we identify communities in subsequent network snapshots based on the previous snapshot's community structure and dynamic updates, such as changes in edges and vertices. To kickstart the process, we utilize the static Louvain algorithm to determine the community structure in the initial snapshot, which serves as valuable input for our algorithm's subsequent operations. This methodology not only enhances efficiency but also enables us to handle dynamic network changes effectively. We describe the main steps of our algorithm below.

Algorithm 1 DyG-DPCD: MPI-based Parallel Dynamic Community Detection

Data: Initial graph $G_t(V_t, E_t)$, Current graph $G_{t+1}(V_{t+1}, E_{t+1})$, Initial community list C_t
Result: Community at next time-stamp C_{t+1}
Calculate edges added: $edges\_added \leftarrow E_{t+1} - E_t$
Calculate edges deleted: $edges\_deleted \leftarrow E_t - E_{t+1}$
Each processor P_i computes for a set of vertices $V^{(i)}$ based on the partitioning strategy
foreach *processor* P_i *(executing in parallel)* **do**
    foreach *mini-batch of edges* **do**
        Extract a portion of edges added: $E_{add} \leftarrow$ part of *edges_added*
        Extract a portion of edges deleted: $E_{del} \leftarrow$ part of *edges_deleted*
        if P_i==*root* **then**
            Scatter E_{add} and E_{del} using MPI
        Remote_Info_Update()
        `/* Perform Community Updates for Edge Addition */`
        $C'^{(i)}_{t+1} \leftarrow$ Community_Updates_Edge_Addition($V^{(i)}$, $E^{(i)}_{add}$, C_t)
        `/* Perform MPI AllGather to synchronize and exchange community information */`
        MPI_AllGather($C'^{(i)}_{t+1}$, C'_{t+1})
        `/* Perform Community Updates for Edge Deletion */`
        $C''^{(i)}_{t+1} \leftarrow$Community_Updates_Edge_Deletion($V^{(i)}$, $E^{(i)}_{del}$, C'_{t+1})
        `/* Perform MPI AllGather to synchronize and exchange community information */`
        MPI_AllGather($C''^{(i)}_{t+1}$, C''_{t+1})
    if P_i==*root* **then**
        Update C_{t+1} using the received community information: $C_{t+1} \leftarrow C''_{t+1}$
return Updated community C_{t+1}

4.1.1 Initial Graph Partitioning

We employed the widely recognized graph-partitioner METIS [44, 45] to partition our input graph, ensuring an effective distribution among processors. Even though METIS provides both edge-cut and communication volume minimization methods, due to our algorithm's strong reliance on edge-cut minimization between processors, we use this approach for our algorithm.

Algorithm 2 Community_Updates_Edge_Addition

Data: A set of vertices $V^{(i)}$, A set of added edges $E^{(i)}_{add}$, Community list C_t

Result: Updated community list $C'^{(i)}_{t+1}$

```
comm_list ← C_t^(i)
for Each edge e in E_add^(i) do
    e = (u, v)
    /* Intra-Community                                          */
    if C(u) = C(v) then
        G_t.add_edge(u, v)
        Update_Neighbors()
    /* Inter-Community                                          */
    else
        if C(u), C(v) ∈ P_i then
            (perm, comm_list) ← calculate_permanence_local()
        else
            if C(u) ∈ P_i then
                n ← u
                P_j ← C(v)
            else
                n ← v
                P_j ← C(u)
            (perm', comm_list) ← calculate_permanence_remote(n)
            MPI_Send(perm, P_j)
            MPI_Recv(perm', P_j)
            if perm > perm' then
                MPI_Send(comm_list,P_j)
                Update_Neighbors()
            else
                MPI_Recv(comm_list,P_j)
C'_{t+1}^(i) ← comm_list
return C'_{t+1}^(i)
```

4.1.2 Incremental Community Detection and Update

We present the pseudocode of our distributed-memory parallel algorithm for community detection on dynamic networks in Algorithm 1. This algorithm efficiently detects communities in dynamic graphs, where edges are added or removed over time. It leverages Message Passing Interface (MPI) for parallel processing on multiple processors, making it scalable for large graphs. The algorithm takes as input the graph at time-stamp t, the graph at next time-stamp $t + 1$, and the initial community list at time-stamp t. The sets of edges added and removed between these two graph snapshots are identified and shown in lines 1-2. Vertices are divided among processors for parallel processing [line: 3]. Edges are processed in batches to improve efficiency. We process the total edge additions and deletions in mini-batches [line: 5]. Small batches of edges are communicated to all processors by the root processor [line:9]. Next, each processor gathers neighbor information of the vertices that do not belong to the current processor given in line 10. Then the processors in parallel update their local community assignments based on the impact of added edges using the Community_Updates_Edge_Addition function [line: 11]. All processors exchange their updated community information using MPI's AllGather function [line: 12]. Similar to the previous step, processors update communities considering removed edges using the Community_Updates_Edge_Deletion function [line: 13]. Another AllGather synchronizes the updated community information across all processors [line: 14]. Algorithm 2 and 4 describe the updates during the addition and deletion of edges, respectively, in the dynamic network and the corresponding changes in the community structure throughout the updates. After processing all edge batches, the root processor merges the received community information into a final updated community list.

Algorithm 3 Update_Neighbors

$P^u_{list} \leftarrow$ A list of processors where neighbors of u belong to
$P^v_{list} \leftarrow$ A list of processors where neighbors of v belong to
while $iter! = N$ **do**
  if $P_j \; in \; P^u_{list}|P^v_{list}$ **then**
    MPI_Send(1,P_j)
  else
    MPI_Send(0,P_j)
  MPI_Recv (x, P_j)
  $degree(u|v) + = x$

Algorithm 4 Community_Updates_Edge_Deletion

Data: A set of vertices $V^{(i)}$, A set of deleted edges $E_{del}^{(i)}$, Community list C'_{t+1}
Result: Updated community list $C''^{(i)}_{t+1}$

comm_list $\leftarrow C'^{(i)}_{t+1}$
for *Each edge e in* $E_{del}^{(i)}$ **do**
  $e = (u, v)$
  `/* Intra-Community */`
  if $C(u) = C(v)$ **then**
    if $degree(u) = 1 \& degree(v) = 1$ **then**
      $G_t.remove\_edge(u, v)$
      $C(u) \leftarrow C(u) - u - v$
      $comm\_list \leftarrow comm\_list \cup u \cup v$
      if $C(u) \notin P_i$ **then**
        $P_j \leftarrow C(u)$
        MPI_Send(comm_list,P_j)
      MPI_Recv(comm_list,P_j)
    else if $degree(u|v) = 1$ **then**
      $G_t.remove\_edge(u, v)$
      $C(u)|C(v) \leftarrow C(u) - u|C(v) - v$
      $comm\_list \leftarrow comm\_list \cup u|v$
      Update_Neighbors()
      if $C(u)|C(v) \notin P_i$ **then**
        $P_j \leftarrow C(u)|C(v)$
        MPI_Send(comm_list,P_j)
      MPI_Recv(comm_list,P_j)
    else
      non_unit_degree_edge_delete()
  `/* Inter-Community */`
  else
    $G_t.remove\_edge(u, v)$
$C''^{(i)}_{t+1} \leftarrow$ comm_list
return $C''^{(i)}_{t+1}$

Details on community updates during edge addition. There are two cases when edges are added in a dynamic graph, as described below.

- Case 1: *Intra-community edges*: $(u, v) \in E, C(u) = C(v)$. The addition of such edge (u, v) is straightforward. The edge (u, v) is added locally in the network. The internal neighbors of u & v are updated about this addition through MPI communication as shown in Algorithm 3.
- Case 2: *Inter-community edges*: $(u, v) \in E, C(u) \neq C(v)$. The addition of inter-edges requires much communication among the processors as illustrated in

Algorithm 2 to calculate the value of permanence as the neighbors of a vertex are scattered among multiple processors. If both the communities, $C(u)$, $C(v)$ of vertex u and v belong to the current processor P_i, then we have to check whether u moves to the new community depending on the permanence value. $C_{new}(u)$ is then advertised to u's internal neighbors in a Breadth First Search (BFS) manner. If the neighbors of u are in different processors, we do a blocking communication here to ensure accuracy. For inter-processor neighbors of u, the order is changed as all of those neighbors are processed after the intra-processor neighbors are done. For computation of permanence values, both vertex-wise and community-wise, we utilize similar steps as described in our prior work [6]. So, we have not discussed details on line 9 or line 17 of Algorithm 2, and other mentions of these computations.

Details on community updates during edge deletion. Edge deletion is computationally less complicated compared to adding edges in dynamic graphs in terms of permanence calculation. There are two cases with multiple sub-cases to delete edges as shown in Algorithm 4.

- Case 1: *Intra-community edges*: There are two sub-cases for intra-community edge deletions.
  - Case 1A: *Unit degree vertices*: u and v are removed from the current community and u, v form their own singleton community. If the community of both of the vertices u, v or u, or v belong to different processors, we communicate the updates to those processors.
  - Case 1B: *Non-unit degree vertices*: It is similar to the Inter-Community Edge Addition. The computations and the communications follow the same structure as Algorithm 2. The pseudo-code for this non-unit degree vertices is given in Algorithm 5.
- Case II: *Intra-community edges*: No computation of permanence is required in this step. Only communicating the updates of deletion is needed if multiple processors are involved.

Algorithm 5 non_unit_degree_edge_delete

```
if u, v ∈ P_i then
    (perm, comm_list) ← calculate_permanence_local()
else
    if u ∈ P_i then
        n ← u
        P_j ← v
    else
        n ← v
        P_j ← u
    (perm′, comm_list) ← calculate_permanence_remote(n)
    MPI_Send(perm, P_j)
    MPI_Recv(perm′, P_j)
    if perm > perm′ then
        MPI_Send(comm_list,P_j)
        Update_Neighbors()
    else
        MPI_Recv(comm_list,P_j)
```

4.2 Implementation Challenges

While implementing our distributed-memory algorithm, we face a few challenges that we discuss here.

Selection of Efficient Data Structure. During the initial implementation of our algorithm, we have been using the Adjacency List (AL) format to store our input network using the C++ Standard Template Library (STL) containers vector and map. We have avoided the Compressed Sparse Row (CSR) format because it takes $O(d_{max})$ in the worst case to remove an edge from the graph whereas the time complexity is $O(1)$ for AL representation. However, the performance of our DyG-DPCD algorithm gets limited due to the dynamic edge update operations. We experimented with both sorted and unsorted neighbor lists for the vertices. The linear search over the unsorted neighbors and the binary search over the sorted neighbors have similar times as given in Fig. 1. Thus, the major challenge we face is to select a better data structure to minimize the search time while finding edges for calculation and adding/deleting edges for the dynamic updates.

Storing the dynamic networks is a separate and complex problem in itself. Based on the dynamic updates, different types of data structures are considered efficient for storing such temporal information. We consider selecting such a structure that helps us improve the performance of the algorithm by minimizing the update time of the network. After exploring different dynamic graph storing frameworks [4, 28, 46] to find such efficient storage, we find that using Packed-Memory Array (PMA) is a very good alternative [4] to use for skewed graphs like social networks. We also

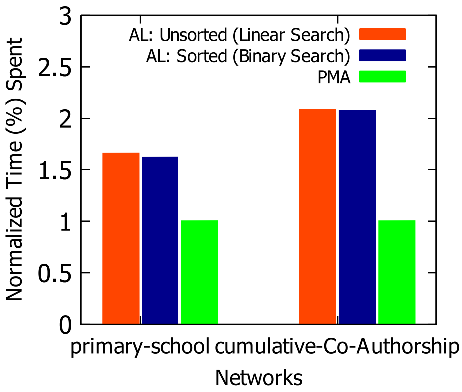


Fig. 1 Time (%) spent in dynamic edge updates from the overall computation time for the sorted and unsorted list using AL and PMA. The time is normalized w.r.t PMA

want to observe how PMA works for networks from other domains as well. So, we have used a sorted array for small-degree vertices and PMA for large-degree vertices to store our networks. From Fig. 1 we see that the update time is reduced to almost half after using PMA.

4.3 Complexity Analysis

We analyze the runtime complexity of our distributed-memory MPI-based Algorithm 1. Suppose, E_a, E_d be the number of edges to be added and deleted for the full network respectively and p be the number of partitions. For Algorithm 2, in the calculation of permanence [Line 9], the computational complexity of calculating permanence based on Equations. 1 and 2, in the worst case scenario would be: $\theta(\sum \tilde{d}^{-\frac{5}{2}})$ and for the best case: $\sum \theta(d^{-\frac{3}{2}})$. Considering the asymptotic growth, growth rate can be bounded by a logarithmic function, and presented the worst case to be: $O(d^2(\log d)^2)$ and the best case: $O(d(\log d)^2)$ where, d denotes degree. The communication complexity is often proportional to the number of cut-edges between partitions denoted by $\vec{E}$. So, we introduce a multiplying factor α that could change depending on the input network properties i.e. edge distribution, sparsity of the network, etc. Thus the communication complexity becomes $O(\alpha\vec{E})$. The line 17 in Algorithm 2 also has the same computational and communication complexity. So, combining these, the average complexity for community updates during edge addition is: $O\left(\frac{E_a}{p}\left[d^2(\log d)^2 + \alpha\vec{E}\right]\right)$. In case of community updates during edge deletion, in Algorithm 4, for the lines [5-21], there is only communication complexity that is the same as the complexity given in edge addition. Again, for Line 23, the complexity is the combination of computational and communication complexity similarly as given in Algorithm 2, Line 1. So, the total run-time complexity of the algorithm becomes:

$$O\left(\frac{E_a}{p}\left[d^2(\log d)^2 + \alpha\vec{E}\right] + \frac{E_d}{p}\left[d^2(\log d)^2 + \alpha\vec{E}\right]\right)$$

Therefore, the time complexity of DyG-DPCD will be higher for both dense and skewed networks.

4.4 Heuristics for Performance Optimization

We employ three different heuristics to further improve the performance of DyG-DPCD. We briefly explain the need for these heuristics below.

Heu-1: Pseudo edge deletion with active/inactive labeled edges We keep one additional parameter to identify edges in the current snapshot. We tag the edge labels as “Active" and “Inactive" to reduce the exact edge deletion operations. When an edge is being deleted, we change the edge label from “Active" to “Inactive" in that particular snapshot. So, we only need to do the search operation instead of removing the element from the data structure. Again, in most networks, a deleted edge may reappear in the subsequent snapshot. Therefore, the inactive edge is changed back to active with the search and update operation only instead of the insertion operation.

Heu-2: Compressing graph size by pruning small degree vertices In the community detection problem, 1-degree or 2-degree vertices do not contribute to the output communities as they mostly act as separate entities in the network. As they are completely isolated in the network, we can reduce the network size to a great extent if we can eliminate such vertices from our calculation. In a sense, we are compressing the network size by keeping them out of the scope of computation. This heuristic can be mostly applicable to networks from the social/blog/communication network domains where some vertices are hub vertices and the network is skewed. So the scope of evolving for most small vertices gets limited.

Heu-3: Computational efficiency by excluding singleton/smaller communities We take into consideration the community distribution of the network at each of the snapshots. While we find singleton communities contributing to a higher percentage, we keep those communities separate in the “Inactive Community List". For the computation of permanence (selecting communities), the computation cost cuts down, as the number of active members in the community list is minimized.

5 Experimental Setup

5.1 Environment

We used Intel CPUs from both US DoE NERSC [49] and LONI [50] supercomputers to perform the experiments.

Table 2 Real-world Dynamic Networks used in the Experiment

Network	Vertices	Edges	$\overline{d}$	d_{max}	Snap-shots	Time Interval
Social [47, 48]						
primary school	242	77.6K	1K	2.6K	6	20 secs
CollegeMsg	1.9K	20.3K	3	1K	7	month
Youtube	3.2M	12.2M	7	0.13M	8	year
Blog/Forum [47, 48]						
sx-stackoverflow	2.6M	63.5M	5	0.1M	9	year
sx-superuser	0.2M	1.4M	7	11.2K	9	year
sx-mathoverflow	24.8K	0.51M	3	7.8K	8	year
fb-forum	899	33.7K	74	1.8K	6	month
Communication [47]						
wiki-talk	1.1M	7.8M	3	22.3K	8	year
email-Eu-core	986	0.3M	4	1K	8	year
email-enron	87K	1.1M	26	38.7K	3	year
Collaboration [47]						
non-cumulative co-Authorship	0.71M	1.2M	2	11.2K	17	year
cumulative co-Authorship	0.71M	1.2M	6	4.3K	17	year
Citation [47]						
HepTh	22.9K	2.7M	233	12K	7	year
HepPh	28.1K	4.6M	327	11.1K	10	year

5.2 Dataset

We use both real-world dynamic and synthetic dynamic updates to large-scale static networks for our experiments. The real-world dynamic networks are depicted in Table 2. We also experiment with some large-scale static networks by doing dynamic updates (insertion and deletion) of edges in batches. Details of these networks are listed in Table 3. We include the synthetic dynamic updates to experiment on even larger networks.

In Fig. 2, we have shown the network statistics, i.e., the number of vertices, edges, communities, and dynamic updates per snapshot for the dynamic real-world networks. We choose one network from each of the domains. From Fig. 3, we observe that except for the 'Youtube' network, the rest of the networks grow in size in subsequent snapshots. The percentage change of addition of edges is higher than that of deletion. Figures 2 and 3 help the readers learn about the network and how the networks evolve over time.

Table 3 Real-world Static Networks with synthetic dynamic updates used in the Experiment

Network	Vertices	Edges	$\bar{d}$	d_{max}	Description
Social [47]					
Slashdot	82.1K	0.95M	21	5K	Slashdot social network from February 2009
LiveJournal	4.9M	68.9M	14	23K	LiveJournal online social network
Orkut	3.1M	0.23B	76	33.3K	Orkut online social network
Biological [48]					
human-gene1	21.9K	12.3M	1.1K	7.9K	Human gene regulatory network, edges represent interaction among genes
human-brain1 (Jung2015-M87104509)	0.7M	50 M	135	5.2K	Edges represent fiber tracts that connect one vertex to another
human-brain2 (Jung2015-M87122310)	0.9M	94.4M	204	8.1K	Edges represent fiber tracts that connect one vertex to another
Collaboration [47, 48]					
com-DBLP	0.3M	1 M	6	343	DBLP collaboration network
coPapersDBLP	0.54M	30.9M	56	3.3K	Edges Connect the papers in the Digital Bibliography and Library Project by co-authorship
Citation [48]					
Patents	3.8M	16.5M	8	793	Citation network among US Patents

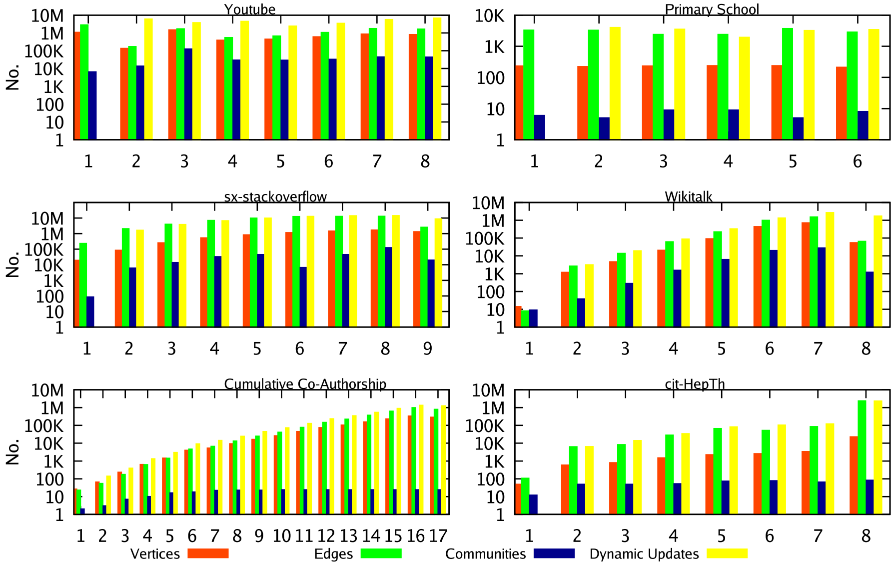


Fig. 2 Network statistics (No. of vertices, edges, community, and dynamic updates) for different real-world dynamic networks in different time slices. The X-axis represents the time steps

6 Experimental Evaluation and Performance

We discuss the performance of our algorithm in terms of scalability and output community quality in this section. We also describe the optimized performance of our algorithm by applying different heuristics. We also compare our algorithm with other related algorithms in the literature.

6.1 Scalability

Speedup is the ratio of the sequential and parallel execution time of the algorithm. The sequential DyPerm algorithm is implemented in Python, while our parallel implementation is in C++. Therefore, we use the execution time of 2 MPI ranks as the sequential time, and calculate the speedup ratio likewise.

Figure 4 represents the speedup for our distributed-memory parallel algorithm for networks from different domains. We get $12-25\times$ speedup for different types of networks with the base version of the algorithm using a maximum of 1024 processors. We get the maximum $25\times$ speedup for 'Orkut', the largest social network in our dataset.

From Fig. 5 we observe that our algorithm gains more scalability for larger networks. For relatively small networks (having edges between ten thousand and one million), the average speedup is $5\times$. Networks having edges in the range of 1 million to 100 million show around $10-15\times$ speedup. For relatively larger networks, up to

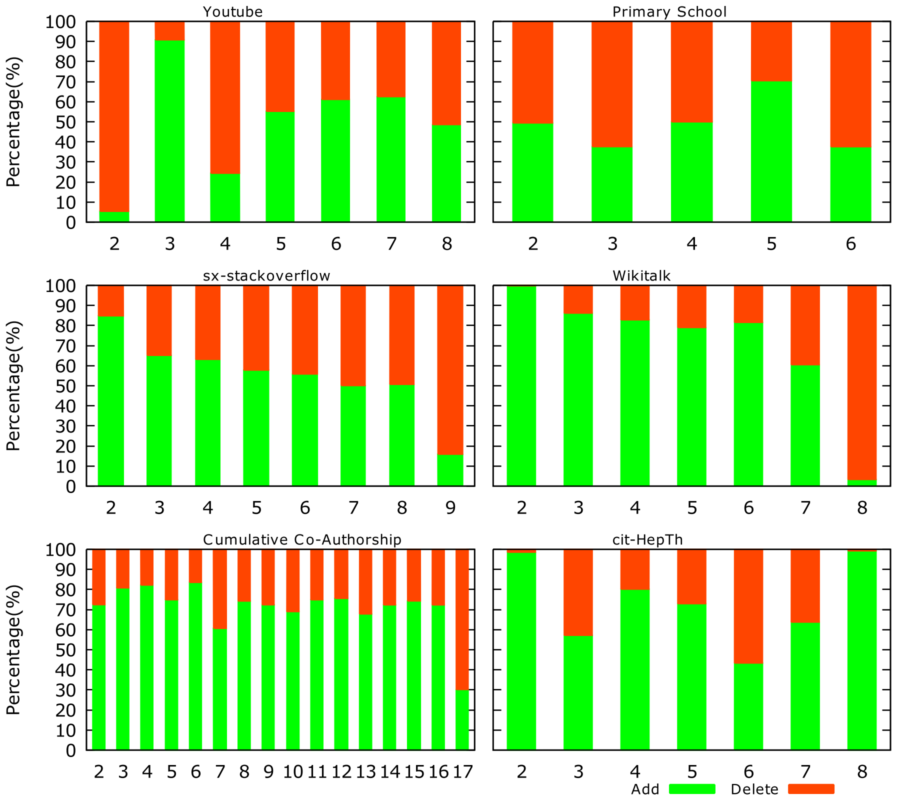


Fig. 3 Percentage change of edge addition and deletion for different real-world dynamic networks in different time-slices. The X-axis denotes different time slices, and the Y-axis denotes the percentage of edge addition and deletion in that particular time slice

1 billion edges, our distributed algorithm shows $20-25\times$ speedup. The fluctuation of the speedup is due to the different structures of the input networks.

6.2 Performance Gain using Different Heuristics

We achieve significant performance gain using all of the heuristics depicted in Figs. 6, 8, and 10.

6.2.1 Heu-1:Pseudo edge deletion with active/inactive labeled edges

Our first heuristic, Heu-1, is applicable to all domains of networks. We have shown the performance gain of our parallel algorithm for the major networks from each domain in Fig. 6. We get a maximum of 20% performance gain in speedup for the 'Patents' dataset. For our largest dataset 'Orkut', we have around 16.5% performance gain, and the speedup increases up to 28.

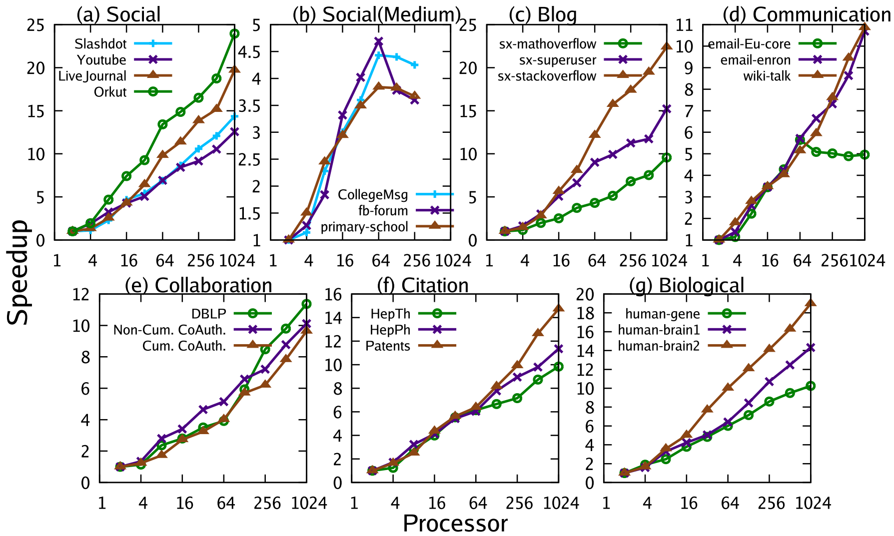
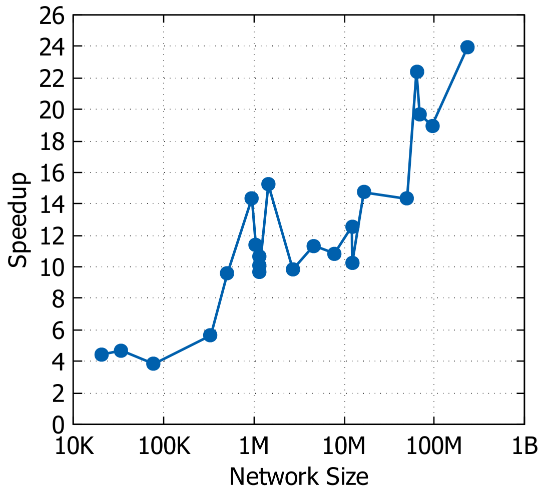


Fig. 4 Speedup for our distributed parallel algorithm for multiple networks from different domains. Larger networks (having edges more than a million) show higher scalability–the speedups keep improving up to 1024 processors (maximum no. of processors used for experiments). Smaller networks understandably scales to a lesser number of processors

Fig. 5 Scalability of our parallel algorithm for increasing network size (in edges) observed for 1024 processors



6.2.2 Heu-2: Compressing graph size by pruning small degree vertices

We apply our second heuristic, Heu-2 to only those networks that have smaller degrees in all of the snapshots. In Fig. 7, we have shown the degree distribution of the ‘sx-stackoverflow’ network. We can observe that for snapshots 3 up to 8, the number of vertices having degree 0/1/2 comprises around 50% – 60% of the total vertices. In snapshots 1 and 2, 30% and 35% of the total vertices have these

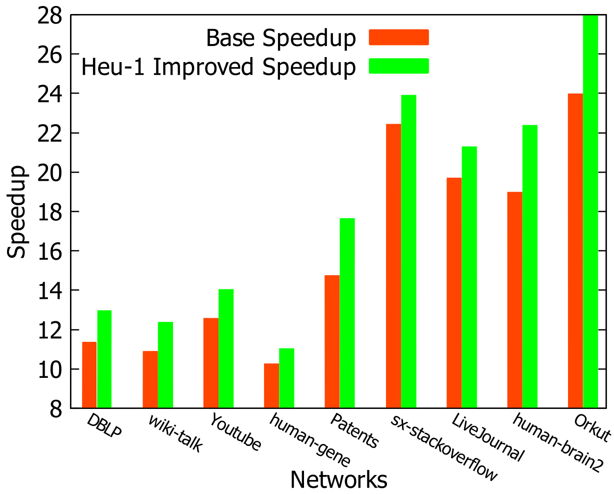


Fig. 6 Performance gain of DyG-DPCD using heuristic, Heu-1 observed for 1024 processors. Heuristic, Heu-1 is based on active/inactive edge labels to decide on the presence or absence of edges at each time slice. The green color denotes the improved speedup gained by running the Heu-1 heuristic-based algorithm, whereas the base speedup shown with the orange color is obtained by running our main algorithm without using any heuristics

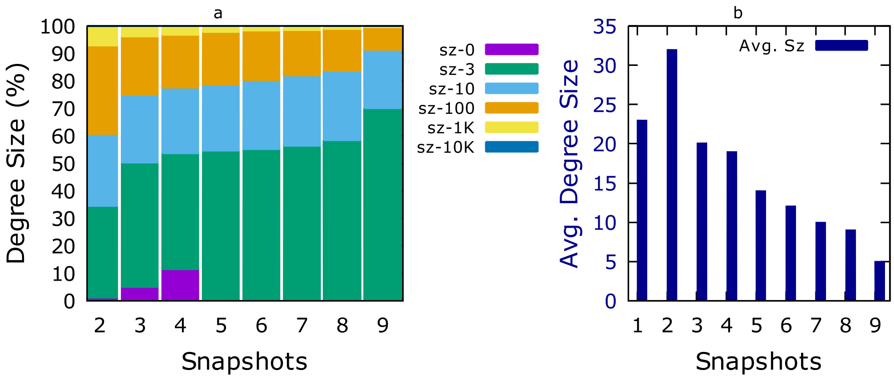


Fig. 7 ‘sx-stackoverflow’ is a representative network with a huge number of smaller degrees (degree: 0-3) in most of the snapshots. The percentage of degree distribution according to the number of degrees is given by the stacked bars in the left figure (a). In the right figure (b), the average degree of the network per snapshot is shown by the blue bars

smaller degrees. In the last snapshot 9, the percentage of smaller degrees is much higher and 70% of the vertices have these small-sized neighbors. The other networks, we choose to apply the Heu-2 heuristic, shows similar kind of degree distribution. Figure 8 shows the performance gain using our Heu-2 heuristic. We get around 26× speedup for the ‘sx-stackoverflow’ dataset. For the ‘Orkut’ network,

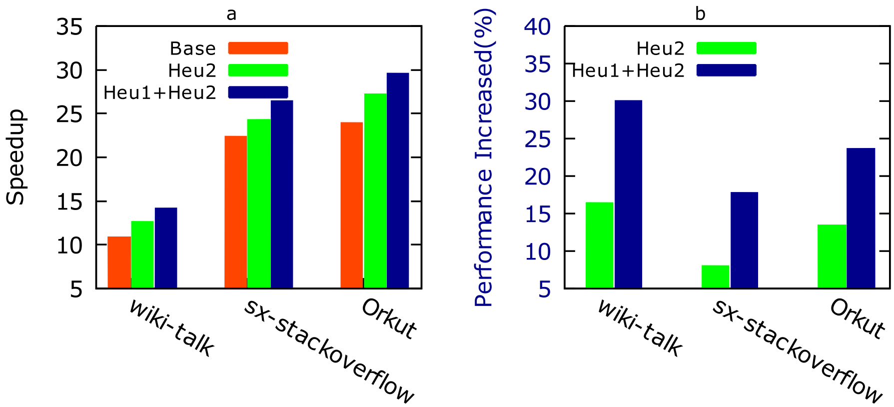


Fig. 8 Performance gain of our parallel algorithm using heuristic, Heu-2 and combining both Heu-1 and Heu-2 heuristics observed for 1024 processors. The left figure (a) shows the change in speedup. The right figure (b) shows the performance gain in percentage

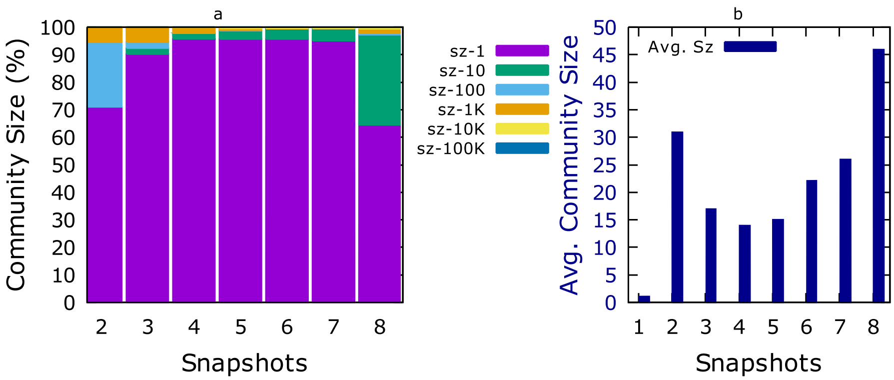


Fig. 9 ‘wiki-talk’ is a representative network with a huge number of singleton communities in most of the snapshots. The percentage of community distribution according to community size is given by the stacked bars in the left figure (a). In the right figure (b), the average number of communities of the network per snapshot is shown by the blue bars

the performance gain is around 20% and we have the maximum speedup of 30 amongst all of our datasets.

6.2.3 Heu-3: Computational efficiency by excluding singleton/smaller communities

We apply our third heuristic, Heu-3 to only those networks which have singleton communities in all of the snapshots. In Fig. 9, we have shown the community distribution of the ‘wiki-talk’ network. We can observe that for all the snapshots around 95% of the total communities are singleton. Figure 10 shows the performance gain using the Heu-3 heuristic. We get around 14× speedup for the ‘wiki-talk’ dataset. If

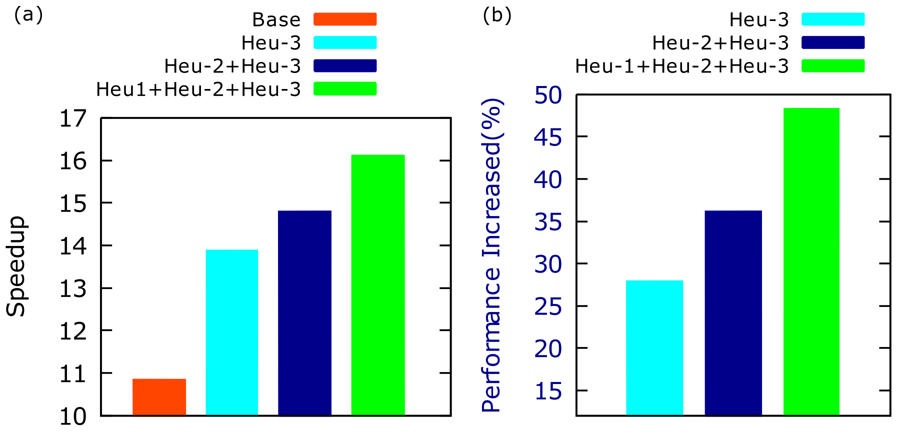


Fig. 10 Performance gain of our parallel algorithm using heuristic Heu-3 and the combination of all Heu-1, Heu-2 and Heu-3 heuristics observed for 1024 processors. The left figure (a) shows the change in speedup. The right figure (b) shows the performance increase in percentage

Table 4 Deviation in permanence value after applying the heuristics, Heu-2 and both (Heu-1+Heu-2) together

Network	Permanence			Change (%)	
	Base	Heu-2	Heu-1+Heu-2	Heu-2	Heu-1+Heu-2
sx-stackoverflow	0.6976	0.6972	0.6967	0.063	0.129
Orkut	0.4300	0.4296	0.4291	0.100	0.200
wiki-talk	0.4676	0.4676	0.4669	0.009	0.154

we apply both Heu-2 and Heu-3 heuristics together, the improvement is around 36% and the speedup becomes 15. Again, if we apply Heu-1, Heu-2, and Heu-3 heuristics altogether, the improvement is around 48% and the overall speedup becomes around 16.

6.3 Qualitative Analysis of the Community Structure

We compare the output communities of both sequential and parallel algorithms comparing the accuracy of the permanence metric based on which we derive the output communities. From Fig. 11 we can observe that the overall quality of the output communities remains the same in the parallel algorithm as in the sequential one. In Table 4 we present the deviation in permanence value after applying the heuristic, Heu-2, and both Heu-1 and Heu-2 heuristics together. We find that

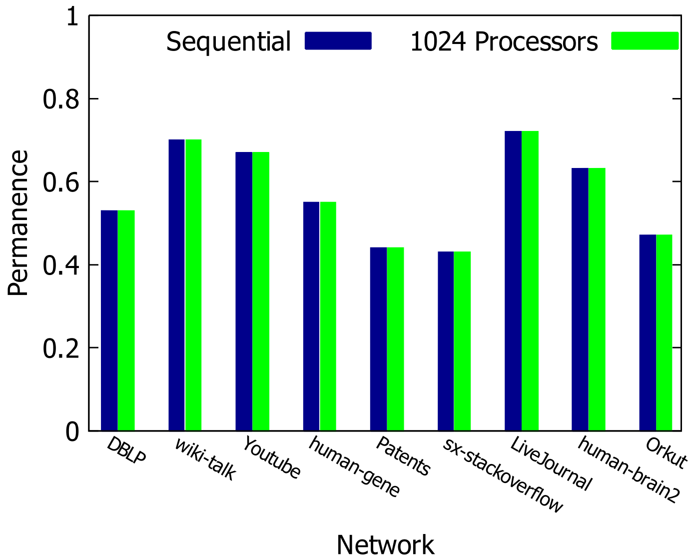


Fig. 11 Comparison of the output community quality in between the sequential and parallel methods based on permanence. Applying heuristic, Heu-1 keeps the permanence value unchanged as well

Table 5 Deviation in permanence value after applying the heuristics, Heu-1, Heu-2 and Heu-3 on wiki-talk network

	Permanence	Change	Change (%)
Base	0.69759	0	0
Heu-3	0.69737	0.00022	0.03083
Heu-2+Heu-3	0.69712	0.00047	0.066668
Heu-1+Heu-2+Heu-3	0.69659	0.00100	0.142644

the deviation is very little as 0.2% and can be considered negligible. In Table 5 we compare the permanence value by applying all three heuristics for the 'wiki-talk' network. We can observe that the deviation is as little as 0.00100 only after applying three of the heuristics together.

6.4 Comparison with the State-of-the-art Algorithms

In this subsection, we have shown the comparison of our parallel algorithm DyG-DPCD with the only 2 existing parallel algorithms in the current literature.

6.4.1 STINGER shared-memory re-agglomeration algorithm

We have compared our distributed-memory-based parallel algorithm, with the only shared-memory parallel algorithm implemented using the STINGER framework. The authors have presented the efficiency of the parallel algorithm in terms of throughput. So we also present the same measure to compare our algorithm with

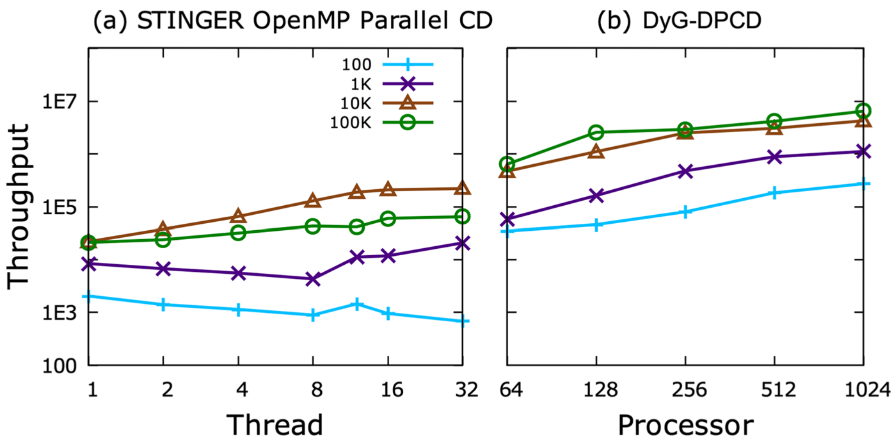


Fig. 12 Comparison of Parallel Performance based on throughput [Updates per second by threads (OpenMP)/processors(MPI)] for coPapersDBLP network. Legends present different batch sizes. Left Figure (a) shows the throughput for STINGER parallel re-agglomeration algorithm. Right Figure (b) shows the throughput for our distributed parallel algorithm DyG-DPCD

theirs. Regarding the throughput calculation, the runtime includes edge updates and community detection computational time. We have used the same network to compare the performance of both algorithms. As the OpenMP implementation is exploiting thread efficiency, we can use a maximum of 32 threads, the number of physical cores available to the system. We report our maximum 1024 processor performance for comparison. From Fig. 12, we can observe that our algorithm can handle more computations using the same number of batch sizes. The best throughput value observed from the STINGER Parallel re-agglomeration algorithm is 0.2 million updates per second (10K batch size). Using 1024 processors, our distributed parallel algorithm can process 6.45 million updates per second for 100K batch size. So our algorithm is able to process the dynamic updates 30× faster compared to the STINGER parallel re-agglomeration algorithm.

6.4.2 Parallel Weighted Community Clustering (PWCC) Maximizing Spark-based Algorithm

We cannot directly compare our work with [15] as the source code is not publicly available. The authors have shown up to 2× parallel speedup whereas for our largest network, we get 30-fold speedup.

6.5 Building on a Strong Foundation: Future Advancements

Designing scalable parallel algorithms in a distributed-memory setting for dynamic graphs is challenging, as evident from the limited prior work. While our DyG-DPCD algorithm demonstrates good performance and scales with increasing network size, we recognize the potential for further advancements. Building upon its strong foundation, we can explore several promising avenues for future optimization. Performance bottlenecks might arise from several sources including an imbalanced load. Our parallel algorithm adopts METIS scheme to build the initial partitions to tackle load balancing issue. One interesting area for future work lies in addressing the unique challenges posed by power-law structured graphs and their hub vertices. Large social networks or other power-law structured graphs make load balancing difficult. In such graphs, a few vertices have very high degrees, and in most partitioning strategy, these high-degree vertices are not identified separately. So, if the edges being deleted or added belong to such vertices, both the communication and computation overhead will increase for a particular processor leading to load imbalance. We aimed to make our current algorithm generalized for all kinds of networks, and hence did not apply any topology-specific strategy to handle these hub vertices. This is left as a future improvement for our algorithm. Another interesting exploration will be considering redistributing vertices after each batch edge addition or deletion. If too many edges are deleted from a particular partition, redistribution could be helpful.

7 Conclusion

We developed a scalable and optimized MPI-based distributed-memory parallel algorithm, DyG-DPCD, for community detection on large dynamic graphs. We experimented on a number of real-world networks from six different domains (e.g., social, communication, biological, etc.), and our algorithm performs well with up to 30× speedup for the largest network in our dataset. We further apply different heuristics that improve the performance of our algorithm by 48% in speedup. Our algorithm shows accurate results with a deviation of as little as $0.009 - 0.2\%$ for graphs with ground truth communities. DyG-DPCD also outperforms the other existing parallel algorithm in literature by 30×. DyG-DPCD represents a significant step forward in efficient parallel community detection for dynamic graphs in a distributed-memory setting. By exploring the promising future directions outlined above, we aim to further refine our algorithm and push the boundaries of performance and scalability in this dynamic and exciting field.

Acknowledgements This manuscript has been authored by employees of UT-Battelle, LLC, under Contract No. DE-AC05-00OR22725, and Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy (DOE). The U.S. Government retains, and the publisher, by accepting the article for publication, acknowledges, that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

Author Contributions N.S., A.B., K.I., and S.A. proposed the methods. N.S. implemented the code and performed the experiments. N.S., K.I., and S.A. contributed to writing and revising the manuscript. All authors read and approved the final manuscript.

Funding This material is based upon work supported by the National Science Foundation under Grant Number 2323533. The work is also partially supported by Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy and the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Dept. Of Energy and National Nuclear Security Administration.

Availability of data and materials All data generated or analyzed during this study are included in this published article

Declarations

Conflict of interest The authors declare that they have no Conflict of interest.

Ethics approval and consent to participate Not applicable.

Consent for publication Not applicable.

References

1. Gabert, K., Sancak, K., Özkaya, M.Y., Pinar, A., Çatalyürek, Ü.V.: Eiga: elastic and scalable dynamic graph analysis. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–15 (2021)
2. Sattar, N.S., Lu, H., Wang, F., Halappanavar, M.: Distributed multi-gpu community detection on exascale computing platforms. In: 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 815–824 (2024). IEEE
3. Sattar, N.S., Arifuzzaman, S.: Scalable distributed louvain algorithm for community detection in large graphs. J. Supercomput. **78**(7), 1–35 (2022)
4. Pandey, P., Wheatman, B., Xu, H., Buluc, A.: Terrace: A hierarchical graph container for skewed dynamic graphs. In: Proceedings of the 2021 International Conference on Management of Data, pp. 1372–1385 (2021)
5. Halappanavar, M., Lu, H., Kalyanaraman, A., Tumeo, A.: Scalable static and dynamic community detection using grappolo. In: 2017 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6 (2017). IEEE
6. Sattar, N.S., Buluc, A., Ibrahim, K.Z., Arifuzzaman, S.: Exploring temporal community evolution: algorithmic approaches and parallel optimization for dynamic community detection. Appl. Netw. Sci. **8**(1), 64 (2023)
7. Awad, M.A., Ashkiani, S., Porumbescu, S.D., Owens, J.D.: Dynamic graphs on the gpu. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 739–748 (2020). IEEE
8. Gabert, K., Pinar, A., Çatalyürek, Ü.V.: A unifying framework to identify dense subgraphs on streams: Graph nuclei to hypergraph cores. In: Proceedings of the 14th ACM International Conference on Web Search and Data Mining, pp. 689–697 (2021)
9. Sattar, N.S., Arifuzzaman, S.: Covid-19 vaccination awareness and aftermath: public sentiment analysis on twitter data and vaccinated population prediction in the usa. Appl. Sci. **11**(13), 6128 (2021)
10. Yang, C., Buluç, A., Owens, J.D.: Graphblast: a high-performance linear algebra-based graph framework on the gpu. ACM Trans. Math. Softw. (TOMS) **48**(1), 1–51 (2022)
11. Faysal, M.A.M., Arifuzzaman, S., Chan, C., Bremer, M., Popovici, D., Shalf, J.: HyPC-Map: A hybrid parallel community detection algorithm using information-theoretic approach. In: 2021 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–8 (2021). IEEE
12. Sattar, N.S., Arifuzzaman, S., Zibran, M.F., Sakib, M.M.: Detecting web spam in webgraphs with predictive model analysis. In: 2019 IEEE International Conference on Big Data (Big Data) (2019). IEEE
13. Concessao, K.J., Cheramangalath, U., Dev, M., Nasre, R.: Meerkat: A framework for dynamic graph algorithms on gpus. arXiv preprint arXiv:2305.17813 (2023)
14. Agarwal, P., Verma, R., Agarwal, A., Chakraborty, T.: Dyperm: Maximizing permanence for dynamic community detection. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining, pp. 437–449 (2018). Springer
15. Zhang, C., Zhang, Y., Wu, B.: A parallel community detection algorithm based on incremental clustering in dynamic network. In: 2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), pp. 946–953 (2018). IEEE

16. Sattar, N.S., Arifuzzaman, S.: Community detection using semi-supervised learning with graph convolutional network on gpus. In: 2020 IEEE International Conference on Big Data (Big Data), pp. 5237–5246 (2020). IEEE
17. Lu, H., Halappanavar, M., Kalyanaraman, A.: Parallel heuristics for scalable community detection. Parallel Comput. **47**, 19–37 (2015)
18. Sattar, N.S.: Scalable graph analytics and hpc operational enhancement: Parallel computing and ml/dl innovations. In: 16th Annual WHPC@SC23 Workshop, 2023 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'23) (2023). https://zenodo.org/doi/10.5281/zenodo.10116236
19. Arifuzzaman, S., Sattar, N.S., Faysal, M.A.M.: Parallel algorithms for mining large-scale time-varying (dynamic) graphs. In: PDSW-DISCS Workshop, 2018 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18) (2018). http://www.pdsw.org/pdsw-discs18/wips/abstracts/arifuzzaman-wip-pdsw-discs18.pdf
20. Newman, M.E., Girvan, M.: Finding and evaluating community structure in networks. Phys. rev. E **69**(2), 026113 (2004)
21. Mucha, P.J., Richardson, T., Macon, K., Porter, M.A., Onnela, J.P.: Community structure in time-dependent multiscale and multiplex networks. Science **328**(5980), 876–878 (2010)
22. Zhuang, D., Chang, M.J., Li, M.: Dynamo: Dynamic community detection by incrementally maximizing modularity. IEEE Trans. Knowl. Data Eng. **33**(5), 1934–1945 (2019)
23. Sarıyüce, A.E., Gedik, B., Jacques-Silva, G., Wu, K.-L., Çatalyürek, Ü.V.: Sonic: streaming overlapping community detection. Data min. knowl. discov. **30**(4), 819–847 (2016)
24. Sattar, N.S., Arifuzzaman, S.: Understanding performance bottleneck to improve parallel efficiency of louvain algorithm. In: PDSW-DISCS Workshop, 2019 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'19) (2019). http://www.pdsw.org/pdsw19/wips/NawSafrinSattar-pdswWIP.pdf
25. Afarin, M., Gao, C., Rahman, S., Abu-Ghazaleh, N., Gupta, R.: Commongraph: Graph analytics on evolving data. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, pp. 133–145 (2023)
26. Wheatman, B., Xu, H.: A parallel packed memory array to store dynamic graphs. In: 2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 31–45 (2021). SIAM
27. Wheatman, B., Burns, R.: Streaming sparse graphs using efficient dynamic sets. In: 2021 IEEE International Conference on Big Data (Big Data), pp. 284–294 (2021). IEEE
28. Dhulipala, L., Blelloch, G.E., Shun, J.: Low-latency graph streaming using compressed purely-functional trees. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 918–934 (2019)
29. Wheatman, B., Xu, H.: Packed compressed sparse row: A dynamic graph representation. In: 2018 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7 (2018). IEEE
30. Green, O.: Inverse-deletion bfs-revisiting static graph bfs traversals with dynamic graph operations. In: 2021 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7 (2021). IEEE
31. Arrighi, E., Grüttemeier, N., Morawietz, N., Sommer, F., Wolf, P.: Multi-parameter analysis of finding minors and subgraphs in edge-periodic temporal graphs. In: International Conference on Current Trends in Theory and Practice of Computer Science, pp. 283–297 (2023). Springer
32. Khanda, A., Srinivasan, S., Bhowmick, S., Norris, B., Das, S.K.: A parallel algorithm template for updating single-source shortest paths in large-scale dynamic networks. IEEE Trans. Parallel Distrib. Syst. **33**(4), 929–940 (2021)
33. Gabert, K., Pinar, A., Çatalyürek, Ü.V.: Shared-memory scalable k-core maintenance on dynamic graphs and hypergraphs. In: 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 998–1007 (2021). IEEE
34. Green, O., Bader, D.A.: custinger: Supporting dynamic graph algorithms for gpus. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6 (2016). IEEE
35. Mertzios, G.B., Molter, H., Niedermeier, R., Zamaraev, V., Zschoche, P.: Computing maximum matchings in temporal graphs. J. Comput. Syst. Sci. **137**, 1–19 (2023)
36. Behnezhad, S.: Dynamic algorithms for maximum matching size. In: Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 129–162 (2023). SIAM
37. Wang, X., Zhang, Q., Guo, D., Zhao, X.: A survey of continuous subgraph matching for dynamic graphs. Knowl. Inform. Syst. **65**(3), 945–989 (2023)

38. Sattar, N.S.: Parallel algorithms for scalable graph mining: Applications on big data and machine learning. PhD thesis, University of New Orleans, Computer Science Department (2022)
39. Sattar, N.S., Arifuzzaman, S.: Overcoming mpi communication overhead for distributed community detection. In: Workshop Softw. Chall. Exascale Computing, pp. 77–90. Springer, Berlin (2018)
40. Sattar, N.S.: Scalable Community Detection using Distributed Louvain Algorithm. https://scholarworks.uno.edu/td/2640/ (2019)
41. Sattar, N.S.: Parallel algorithms for scalable graph mining: Applications on big data and machine learning. In: Doctoral Showcase, 2021 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'21) (2021). https://sc21.supercomputing.org/proceedings/doctoral_showcase/doc_showcase_pages/drs111.html
42. Sattar, N.S., Arifuzzaman, S.: Parallelizing louvain algorithm: distributed memory challenges. In: 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing (DASC 2018), pp. 695–701 (2018). IEEE
43. Riedy, J., Bader, D.A.: Multithreaded community monitoring for massive streaming graph data. In: 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, pp. 1646–1655 (2013). IEEE
44. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. scientific Computing **20**(1), 359–392 (1998)
45. LaSalle, D., Karypis, G.: Multi-threaded graph partitioning. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, pp. 225–236 (2013). IEEE
46. Shun, J., Blelloch, G.E.: Ligra: a lightweight graph processing framework for shared memory. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 135–146 (2013)
47. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data (2014)
48. Rossi, R.A., Ahmed, N.K.: The network data repository with interactive graph analytics and visualization. In: AAAI (2015). https://networkrepository.com
49. Documentation | NERSC | Systems. https://www.nersc.gov/systems/
50. Documentation | User Guides | QB3. http://hpc.loni.org/resources/hpc/system.php?system=QB3

Authors and Affiliations

Naw Safrin Sattar[1] · Khaled Z. Ibrahim[2] · Aydin Buluc[2] · Shaikh Arifuzzaman[3]

✉ Naw Safrin Sattar
sattarn@ornl.gov

✉ Shaikh Arifuzzaman
shaikh.arifuzzaman@unlv.edu

Khaled Z. Ibrahim
kzibrahim@lbl.gov

Aydin Buluc
abuluc@lbl.gov

1 Oak Ridge National Laboratory, Oak Ridge 37831, TN, USA

2 Lawrence Berkeley National Laboratory, Berkeley 94720, CA, USA

3 University of Nevada, Las Vegas 89154, NV, USA