Tiako P.F. (ed) *Competitive Tools, Techniques, and Methods*. Chronicle of Computing. OkIP. AHPC24#06
 https://doi.org/10.55432/978-1-6692-0007-9_8

Hybrid Parallelization for Accelerating Visibility-Graph Construction and Community Detection on Temporal Data

Xun Jia[a], Minzhang Zheng[b], Liwei Wang[a], Shuangshuang Jin[a]

[a]*School of Computing, Clemson University, 1240 Supply St, North Charleston, 29406, USA*
[b]*Department of Hematology, St. Jude Children's Research Hospital, 262 Danny Thomas Pl, Memphis, 38105, USA*

ABSTRACT

In the present era, vast amounts of time series data, particularly in biology, require efficient analysis due to high-dimensional datasets from advanced technologies. Transforming time series into networks and applying community detection methods can uncover dynamic patterns as temporal network communities. However, the large size of these datasets often extends analysis time. High Performance Computing (HPC) addresses this by accelerating traditional applications. This article presents an HPC-optimized version of the visibility-graph-based temporal community detection method. Enhancing the original algorithm with parallel processing, including shared memory and message passing models, improves adaptability. Experiments using an artificial sine series and two real-world datasets—biological time series and power consumption patterns—on Clemson's Palmetto Cluster demonstrate significant performance improvements and scalability over the original approach.

Keywords: High Performance Computing, Temporal Series Analysis, Visibility Graph, Community Detection

I. INTRODUCTION

Temporal data holds substantial value for advancing knowledge across various scientific disciplines. In environmental studies, the tracking of temporal changes in environmental factors plays a pivotal role in climate research and disaster prediction. Moreover, temporal data finds indispensable applications in diverse domains such as epidemiology, transportation planning, climate science, and urban development, serving as the foundational framework for modeling, prediction, and response to dynamic processes (Langran, 1989). The introduction of the visibility graph (VG) algorithm (Lacasa et al., 2008) has revolutionized the analysis of temporal data by transforming time series into structured graphs, thereby facilitating the application of graph algorithms to temporal data. Community detection, a fundamental graph analysis tool, emerges as a potent instrument for unveiling evolving patterns, enhancing comprehension of dynamic behaviors, and empowering data-driven decision-making in domains characterized by temporal data evolution. However, conventional methodologies, while effective in fulfilling their intended purposes, were not originally conceived with runtime performance optimization in mind. The relentless proliferation of time series data, characterized by its burgeoning volume, has led to prolonged processing times, impeding research endeavors, particularly in temporal community detection. Scientific computations traditionally hinge on single-core performance, severely limiting their real-time capabilities. Furthermore, community detection algorithms confront significant challenges amidst the surge in large-scale, intricate network data across scientific fields. Massive network data forms a large-scale network, composed of billions of nodes and edges, which generates models with large amount of super-parameters and extensive training sets (Jin et al., 2021). The TCCD model proposed in (Wang, Jin, Musial, & Dang, 2019) and the stochastic model method proposed in (Jin, Wang, Dang, He, & Zhang, 2016) train advanced models by applying approximation and reduction on the sophisticated network structure to keep the training efficiency, leading to the sacrifice of modeling accuracy.

Fig. 1. The process of community detection on time series

High Performance Computing (HPC) accelerates scientific computations by utilizing supercomputers and cloud computing. Researchers across various fields use parallel computing to develop high-performance applications, maximizing run-time performance within available resources. However, developing mature HPC-based applications is complex, requiring deep understanding of parallel program behaviors and significant programming expertise.

In this study, we introduce a parallelized VG-based temporal community detection algorithm for time series analysis. Parallel processing is incorporated into both the conversion of time series to VGs and the community detection process. Figure 1 provides an overview of the procedure. We enhanced the original Python algorithm by optimizing data structures and integrating multiprocessing, shared memory, and message passing for parallel processing. Performance comparisons between the HPC-optimized and serial versions using synthetic and real data demonstrate significant speed improvements.

The paper is organized as follows: Section II reviews HPC and temporal community detection. Section III details the HPC-optimized algorithm. Section IV describes the experimental setup and presents comparative results. Section V concludes with future work.

* Corresponding author E-mail: xunj@g.clemson.edu

Presented at the 2024 OkIP International Conference
on Advances in High-Performance Computing (AHPC)
in Oklahoma City, OK, USA, and Online, on October 3, 2024

II. Background

A. High Performance Computing

HPC aggregates computing resources for high computational capability, solving large problems in science and engineering. Traditional applications, often designed serially, face performance limitations due to single-CPU restrictions. Modern multi-core CPUs and cloud HPC resources address these limitations, advancing research in machine learning (Ramirez-Gargallo, Garcia-Gasulla, & Mantovani, 2019), physics (Jiang, Liu, & Cheng, 2022), power systems (Wang, Jin, Huang, Huang, & Chen, 2022), and biology (Sanbonmatsu, & Tung, 2007).

HPC uses two classical models: data-level parallelism and task-level parallelism. Data-level parallelism, often implemented with threads and shared memory (e.g., OpenMP (Dagum, & Menon, 1998), pthreads (Nichols, Buttlar, & Farrell, 1996)), performs identical operations on separate data segments. GPUs (Luebke, & Harris, 2004) exemplify this model. Task-level parallelism distributes different tasks across nodes, used in MPI (Gabriel et al., 2004), Hadoop MapReduce (Dean, & Ghemawat, 2008) and Apache Spark (Apache sparkTM). Innovative applications like PETSc (Abhyankar et al., 2018), and tools like Python's NumPy (Harris et al., 2020) and Julia (Bezanson, 2017) leverage low-level libraries (e.g., (BLAS), (LAPACK)) for specialized needs.

B. Visibility Graph and Temporal Community Detection

VG transforms time series data into networks, preserving essential properties (Lacasa et al, 2008). Nodes represent time points, connected if they can be joined by an unobstructed line. VGs facilitate diverse applications, such as analyzing human behavior in biology (Masoudi-Sobhanzadeh, Gholaminejad, Gheisari, & Roointan, 2022), (Kutluana, & Türker, 2024) and planning collision-free paths in robotics (Bonin-Font, & Burguera, 2020). Variants like weighted and dynamic VGs cater to novel applications.

Community detection identifies clusters within data, applicable in fields like biology (Rahnavard et al., 2021), computer science (Gasparetti, Sansonetti, & Micarelli, 2021), and social science (Guerrero-Solé, 2017). It enhances understanding of social networks and improves recommendation systems. Advanced methods, such as tensor-based algorithms (Al-Sharoa, Al-Khassaweneh, & Aviyente, 2018), track brain network structures over time.

Despite many VG applications, parallel programming to enhance runtime performance is not a primary focus. The growing demand for processing intensive tasks calls for more efficient programs. HPC empowers researchers and engineers to efficiently handle complex scenarios and large datasets, enabling real-time applications when necessary.

III. Methodology

In our study, we adapted the serial approach by (Zheng, Domanskyi, Piermarocchi, & Mias, 2021) as the foundational framework (Zheng, Domanskyi, Piermarocchi, & Mias, 2021) which effectively identifies communities within temporal data. This method transforms time series data into a VG, then detects temporal communities within these graphs. In biology, these communities represent groups of time points within a signal that likely indicate the same biological state.

To enhance performance, we integrated shared-memory and message-passing models into the algorithm, introducing two-tiered task-level parallelism for concurrent processing of multiple time series. We utilized (Harris et al., 2020) to handle dense-matrix-based large data structures, boosting the computational efficiency.

A. The serial method

The serial method deals with time series data first. It maps time series to a "Weighted Dual-Perspective Visibility Graph (WDPVG)". Algorithm 1 describes the process of VG construction. The VG is constructed by first representing the time series points as N nodes in a network, where nodes i and j represent time t_i and t_j, with intensities $s(t_i)$ and $s(t_j)$. Edges are added between node i and j if an intermediate time point k has an intensity $s(t_k)$ that satisfies the following conditions for natural VG (NVG) and Horizontal VG (HVG) respectively.

$$s(t_k) < s(t_j) + (s(t_i) - s(t_j))\frac{t_j - t_k}{t_j - t_i} \quad \text{(NVG)}$$

$$s(t_i), s(t_j) > s(t_k) \quad \text{(HVG)}$$

When applying weight to the added edge, various choices are to be offered including no weight, Euclidean distance, the tangent of the view angle, or the time difference, as described respectively by (1)-(3) and thus the adjacency matrix is obtained.

$$w_{ij} = \sqrt{(s(t_i) - s(t_j))^2 + (t_i - t_j)^2} \quad (1)$$

$$w_{ij} = \left|\frac{s(t_i) - s(t_j)}{t_i - t_j}\right| + 10^{(-8)} \quad (2)$$

$$w_{ij} = |t_i - t_j| \quad (3)$$

Then, the reflected perspective NVG/HVG is constructed by reflecting the intensities S_t across the time axis $S_t' = -S_t$ and then repeat the steps mentioned above. To finally get the WDPVG, the normal perspective NVG/HVG and reflected perspective NVG/HVG is combined by the following criteria:

$$A_{ij}^{d} = max\{A_{ij}, A_{ij}^{'}\} \quad (4)$$

where A_{ij} and A'_{ij} represent the adjacency matrix of the normal perspective NVG/HVG and the reflected perspective NVG/HVG respectively.

Algorithm **1** Serial Visibility Graph Creation

Input: tp: time stamps, **data:** time series data
Output: G: the adjacency matrix of VG
Initialization: **G** = empty matrix, **dim** = number of time stamps

```
for i = 0 to dim do
    if i < dim - 1 then
        G[i, i + 1] = G [i + 1, i] ← designated value
        for j = i + 2 to dim do
            if max(data[i +1 : j])<min(data[i], data[j]) then
```

```
6:            G[i, jj] = G[j, i] ← designated value
7:          end if
8:        end for
9:      end if
10: end for
```

After constructing the VG, the next steps involve community identification and optional merging. The shortest path between the VG's initial and final nodes is computed to form the community stems, with each node on this path considered a community core. Nodes outside this path are assigned to communities based on proximity to the stem nodes, using the shortest path length. If multiple nodes have identical path lengths, they are assigned to the community of the first node in the predefined direction. The shortest path lengths for all node pairs in the VG are calculated. After community construction, an optional merging process may be applied: communities with shortest path lengths below a cutoff value are merged, based on evaluating path lengths between pairs and selecting the smallest. Algorithm 2 provides a brief explanation of the process.

Algorithm 2 Serial Temporal Community Detection

Input: **G**: the adjacency matrix of VG
Output: **Communities**: the detected communities from the graph

```
    Initialization: dist = empty_matrix, dim = the dimension
          of the matrix, Communities = empty_set
1:  for i = 0 to dim do
2:      for j = 0 to dim do
3:              dist[i, j] = dist[j,i]
    ← Dijkstra(i,j)
4:      end for
5:  end for
6:  shortest_path = DijkstraPath(start_node, end_node)
7:  for i = 0 to length_of(shortest_path) do
8:      Communities.append(i:[shortest_path[i]])
9:  end for
10: for i = 0 to dim do
        if i not in shortest_path then
12:         c_id = min(all pair of Dijkstra(i,v) for v in
            shortest_path)
13:         Communities[c_id].append(i)
14:     end if
15: end for
```

Suppose that V represents the number of the nodes in the graph, and E represents the number of the edges, the process of building VGs has a time complexity of $O(N^3)$ and a space complexity of $O(N^2)$. In the community detection algorithm, finding the shortest path using Dijkstra's algorithm is an $O(NlogN)$ process, calculating all pairs of Dijkstra's path length cost $N * O(NlogN)$ time, assigning the nodes into their communities costs $O(N^2)$ time, and the optional merging process has a run-time of $O(N^2)$, hence, all the sub processes adds up to an $O(Nlog_N + N^{2logN} + N^2)$=$O(N^{2logN})$ overall time complexity. The space complexity is $O(N^2)$ as only the adjacency matrix is needed through the whole process.

It is worth noting that while the accuracy and performance of temporal community detection show promise, as demonstrated by the author, it is hindered by extended execution times. With the ever-increasing volume of accessible data, inefficient algorithms can impede progress in research and engineering endeavors. To address this challenge, we have incorporated HPC techniques into our approach for a faster and more efficient data processing.

B. Parallelization and Implementation for the Serial Version

Since Python is widely used in scientific research and the original algorithm was implemented in Python, we continued using Python for our parallel design. Due to Python's Global Interpreter Lock (GIL) restricting multithreading, we employed multiprocessing as an alternative. The VG computation and node allocation involve numerous identical, independently executable operations, making the shared-memory model suitable. To mitigate slowdowns from atomic operations in accessing critical data fields, we also incorporated a message-passing model. Algorithm 3 includes a load balancer to address performance issues from line 5 of Algorithm 1, where the workload decreases as the value of i increases, potentially causing uneven task distribution.

Algorithm 3 Parallelized Visibility Graph Creation

Input: LoadBalancer():function that distribute work,
worker(chunk):worker function that calculates the assigned chunk of the adjacency matrix, **tp:** time stamps, **data**: time series data
num_proc: number of worker processes
Output: G: the adjacency matrix of VG
Initialization: ***G*** = shared memory block, **dim** = number of time stamps

```
1: chunks = LoadBalancer()
2: for i in () to num_proc do
3:     p = new_process(worker, args=(chunks[i], G))
4:     p.start()
5:     p.join()
6: end for
```

In Algorithm 3, the outer loop in line 1 from Algorithm 1 is evenly divided among all worker processes to ensure equitable task distribution. To further counteract workload imbalances, suppose we have a total of n processes with indices ranging from 1 to n. Each process is allocated an equal share of $\frac{dim}{n}$ numbers, and each i is assigned to process p_{index} using the following equation:

$$WorkList_{index} = [i, if(i \mod n) = index] \quad (5)$$

*A*The adjacency matrix is defined as a NumPy matrix and resides within a Python shared memory block. Individual processes are initiated as distinct Python interpreter processes, each equipped with its own dedicated work list. These processes operate concurrently to compute the adjacency matrix. For an in-depth understanding of the parallelized algorithm, including the load-balancing mechanism, please refer to Algorithm 3.

In Algorithm 4, we also define the distance matrix as a shared NumPy matrix, and the while loop in line 1 of Algorithm 2 is evenly distributed among the processes. Lines 11 to 15 introduce a redesigned assignment procedure encapsulated within a function. Each process is responsible for handling its own chunk of nodes and stores the results in its local memory to prevent potential data race issues arising from simultaneous writes. Every process is assigned a specific set of nodes and concurrently executes the function to allocate nodes to their respective communities. Upon completing the computation, the results are transmitted to the main process, which consolidates them to obtain the outcome. Once all sub-results are received, the main process returns.

Algorithm 4 Parallelized Community Detection

Input: C: the adjacency matrix of the VG,
worker_Com(chunk, local):worker function that assign nodes to their community,
num_proc: number of worker processes,
worker_Dijkstra(chunk, mat): worker function that calculates the Dijkstra's path lengths

Output: communities: the detected communities from the graph
Initialization: **dist** = empty matrix, **dim** = the dimension of the matrix,

```
chunk = dim / num_proc
for i = 0 to num_proc do
    p = new_process(worker_Dijk(i*chunk, (i+1)*chunk), dist)
    p.start()
    p.join()
end for
shortest_path = DijkstraPath(start node, end node)
for i = 0 to length_of(shortest_path) do
    Communities.append(i:[shortest_path[i]])
end for
for i = () to num_proc do
    if process == 'main' then
        p = new_process(worker Com(i*chunk,(i+1)*chunk), local)
        p.start()
        p.send_to_main(local)
        p.join()
    end if
end for
if process == 'main' then
    i = 0
    while i<num_proc do
        local = process.receive()
        Communities.merge(local)
        i + = 1
    end while
end if
```

Our approach introduces more efficient parallel procedures and leverages NumPy matrices to streamline the code and enhance performance compared to the original implementation. Assuming a time series length of N and the utilization of *num_proc* processes, Amdahl's law (Amdahl, 1967) indicates that the theoretical maximum speedup is:

$$\lim_{N\to\infty} \frac{O(N^3)+O(N^2 logN)+O(NlogN)+O(N^2)}{\frac{O(N^3)+O(N^2 logN)+O(N^2)}{num\_proc}+O(NlogN)} = num\_proc \quad (6)$$

C. Two-Tiered Parallelism

While the parallelized algorithm is primarily designed for single long time series, scenarios arise where researchers need to work with datasets comprising thousands of short time series. Algorithms 3 and 4, while capable of significantly accelerating the processing of long time series, may incur substantial overhead due to process scheduling when applied to numerous short time series. If the higher-level task (e.g., executing one VG construction and community detection on individual time series) remains serial, a challenge persists: resources are not fully utilized, and runtime performance remains suboptimal.

To tackle this issue, we have implemented a two-tiered parallelism approach. While the lower-level parallelism utilizes Algorithm 3 and Algorithm 4 to accelerate the processing of one task, the higher-level parallelism utilized message passing model and enables users to concurrently process multiple tasks. A master process initiates multiple sub-processes to execute higher-level tasks concurrently. Once a sub-process completes its task, it sends the partial result back to the master program, which aggregates all the partial results to derive the comprehensive result. Users can determine the number of processes to employ for higher-level tasks based on an analysis of which number of processes yields the best speedup at the inner level and the available core count. For instance, if there are 40 available cores and it's determined that using 8 processes provides the optimal speedup for a single time series processing task (lower-level parallelism), users can opt to employ 5 processes to concurrently process 5 tasks (higher-level parallelism). This approach further enhances and rationalizes resource utilization while enhancing performance gains. Supplementary information can be found in Algorithm 5.

Algorithm 5 Parallelized Task Processing

Input: **data**: the set of time series. **NUM_PROC**: number of processes, **worker(data)**: the two functions above,
Output: **result**: the results

Initialization: **result** = empty set, **chunks** = data divided evenly to **num.proc chunks.**

```
if process == main then
    for i = 0 to NUM_PROC do
        p = Process(worker(chunks[i]))
        p.send_to_main()
        p.start()
    end for
end if
if process == 'main' then
    i = 0
```

```
10:  while i < num_proc do
11:    local=process.receive()
12:    result.merge(local)
13:    i  + = 1
14:  end while
15: end if
```

IV. Experiment and Analysis

In this section, we present the implementation and evaluation of our proposed approaches. The implementation was conducted on the Palmetto cluster at Clemson University, featuring 34,916 CPU cores, high speed interconnection, and large RAM, with over 850 nodes are equipped with NVIDIA Tesla GPUs. One of the nodes we used for testing consists of 40 Intel Xeon Gold 6258R CPUs, each operating at 2.70 GHz, and equipped with 128 GB of global shared memory to test single long time series. The other node, to support testing multiple time series, features 80 Intel Xeon Gold 6138 CPUs running at 2.00 Ghz with 750 GB of global shared memory.

We conducted comprehensive tests by comparing the performance of the parallelized versions against the original serial version. To assess the performance gains effectively, we performed tests on both artificial sine data and real-world datasets. The sine data comprised 10,000 data points generated using Eq.7 and Eq.8, where the "timestamp" ranged from 0 to 1114π with an increment of 0.35. The first real-world dataset was obtained from (Zhou et al., 2019) and contains 10,346 time series that containing 152 data points. The second real-world dataset was sourced from (Hebrail, & Berard, 2012), where we truncate the first 10000 points for efficiently fulfilling the testing purpose.

$$times = range(0, 1114\pi, 0.35) \tag{7}$$

$$data = 5\cos(times) + 2 \times random(len(times)) \tag{8}$$

A. Single Long Time Series

We assessed the speedup achieved by the parallelized VG construction process using the sine data and the power data, encompassing all four types of graphs. Figure 2 and 3 illustrates the runtime, while Table 1 and 2 present speedup metrics for VG construction using the same datasets. Notably, our observations indicate that more complex graphs benefit to a greater extent from parallelization. Moreover, as the dataset size increases, the speedup tends to approach linearity.

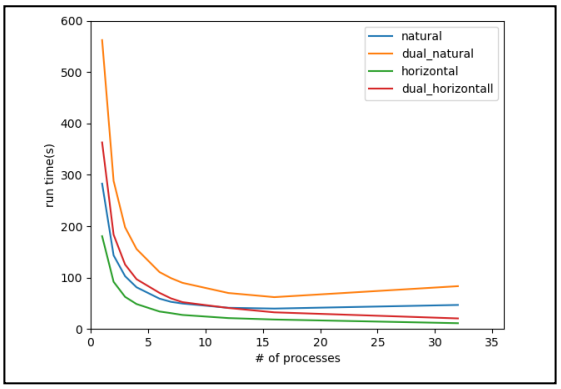

Fig. 2. Runtime of VG construction on Sine data

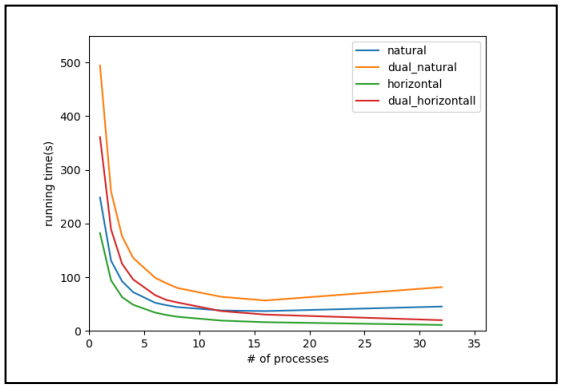

Fig. 3. Runtime of VG construction on Power data

TABLE I. SPEEDUP OF VG CONSTRUCTION ON SINE DATA

# of Processes	1	2	4	8	12	16	32
NVG	1.00	1.97	3.49	5.71	6.87	7.13	6.05
Dual NVG	1.00	1.98	3.61	6.26	8.03	9.07	6.75
HVG	1.00	1.96	3.72	6.63	8.53	9.82	16.07
Dual HVG	1.00	1.98	3.75	6.96	8.89	11.22	17.69

TABLE II. SPEEDUP OF VG CONSTRUCTION ON POWER DATA

# of Processes	1	2	4	8	12	16	32
NVG	1.00	1.90	3.44	5.61	6.54	6.75	5.48
Dual NVG	1.00	1.90	3.63	6.18	7.79	8.74	6.07
HVG	1.00	1.94	3.75	6.92	9.54	11.30	16.57
Dual HVG	1.00	1.91	3.77	6.81	9.81	11.92	18.16

Table 3 provides insights into the size of dual-perspective HVG with distance-based weight, constructed using Algorithm \ref{parallel-VG}. The graph from Sine data contains 10000 nodes and 29961 edges, the graph from (Hebrail, & Berard, 2012) contains 10000 nodes and 18841 edges. The 10346 graphs generated from (Zhou et al., 2019) consist of 152 vertices with edge counts ranging between 82 and 236. We subsequently executed Algorithm 4 on the dual-perspective HVG with weight attributed as "distance" from both the sine series and the power consumption series (Hebrail, & Berard, 2012). The corresponding runtime and speedup data are presented in Table 4, wherein the upper section displays the outcomes pertaining to the sine data, while the lower section exhibits the results associated with the power consumption data. The results indicate that this algorithm demonstrates performance and characteristics similar to that of Algorithm 3.

TABLE III. SUMMARY OF VGS

Name	# of Vertices	# of Edges	# of TSs
Sine Data	10,000	29,961	1

Biological	152	82,236	10,346
Power Consumption	10,000	18,441	1

B. Two-Tiered Parallelism for Multiple Time Series

To evaluate the performance of our two-tiered parallelism approach, we conducted tests using time series data from (Zhou et al., 2019). Since a single time series is relatively short, we chose to use only 1 process to avoid the overhead from process scheduling. Specifically, we ran the VG construction and community detection processes with 1 processes executing a single task, and 1, 2, 4, 8, 16, 32 processes on the higher-level and observed efficient results. Although the overhead hurts the performance, a noteworthy speedup is still seen. We also examined a dataset comprising 4 time series extracted from the head of (Hebrail, & Berard, 2012), with each chunk containing 10080 data points (as the data is recorded every minute, 10080 records contains data from a week). Upon testing, we utilized 16 processes in the lower-level parallelism for a single task to achieve a maximum performance, and varied the number of processes from 1 to 4 for the higher-level processing and achieved promising results (Table 5). These outcomes affirm the efficiency of the second-level parallelism: For the data from from (Zhou et al., 2019), though the scheduling overhead from 32 processes reduced the speedup, we still achieve 10.76x speedup against the serial version. For the data from (Hebrail, & Berard, 2012), While the higher-level parallelism utilizes 3 processes, the result shows a 3.2x speedup comparing to using only 1 processes, and a 34.5x speedup comparing to the serial execution (2431.5s upon testing). However, when using 4 processes, the speedup drops due to the overhead of massive process scheduling.

TABLE IV. PERFORMANCE SUMMARY OF ALGORITHM 4 ON SINGLE LONG TIME SERIES

# of Processes	**1**	**2**	**4**	**8**	**16**	**32**
Runtime(s)	315.9	163.6	84.1	44.0	26.1	16.5
Speedup	1.00x	1.93x	3.76x	7.18x	12.10x	19.15x
# of Processes	**1**	**2**	**4**	**8**	**16**	**32**
Runtime(s)	234.9	123.4	64.2	35.8	21.2	14.7
Speedup	1.00x	1.90x	3.65x	6.56x	11.08x	15.98x

TABLE V. PERFORMANCE SUMMARY OF THE TWO-TIERED PARALLELISM

# of Processes	**1**	**2**	**4**	**8**	**16**	**32**
Runtime(s)	795.6	693.1	411.8	254.2	133.1	73.9
Speedup	1.00x	1.14x	1.93x	3.24x	5.97x	10.76x

# of Processes	**1**	**2**	**3**	**4**
Runtime(s)	225.8	125.0	70.5	80.9
Speedup	1.00x	1.81x	3.20x	2.79x

Our validation process included comparisons between the results obtained from the proposed parallelized methods and those from the original version. Remarkably, the results were found to be consistent. For single-long time series, we achieved a nearly 7x speedup with 8 processes, with greater computational complexity yielding more substantial benefits from parallelization. In cases involving datasets containing numerous short time series, the two-tiered parallelism exhibited remarkable performance improvements. Moreover, for datasets featuring a substantial number of mid-sized to huge time series, the two-tiered parallelism approach yielded optimal performance enhancements. Users can flexibly determine the number of processes based on their specific requirements and problem domains.

V. CONCLUSION

This paper introduces a parallelized approach for VG-based community detection applied to time series data. We have innovatively incorporated two-tiered parallelism to address a broader spectrum of real-world scenarios. Our implementation demonstrates commendable performance in terms of execution time, without incurring additional memory overhead. Furthermore, it exhibits robust scalability as problem sizes increase. This work underscores the adaptability of parallel programming in computationally intensive applications.

While parallel applications have brought significant advantages in both academic and industrial domains, it is worth noting that their successful implementation often demands substantial expertise. The intricacies of parallel programming can introduce challenges related to debugging and verification. Additionally, the task of implementing and comparing the performance of diverse parallel APIs to select the most suitable one can be labor-intensive. To tackle this problem, our future tasks will focus on program verification and automated parallelization, with the aim of streamlining the development cycle for parallel applications. This will enable a broader audience to harness the benefits of parallelization in their respective domains.

REFERENCES

Langran, G. (1989). A review of temporal database research and its use in GIS applications. *International journal of geographical information system*, *3*(3), 215-232.

Lacasa, L., Luque, B., Ballesteros, F., Luque, J., & Nuno, J. C. (2008). From time series to complex networks: The visibility graph. *Proceedings of the National Academy of Sciences*, *105*(13), 4972-4975.

Jin, D., Yu, Z., Jiao, P., Pan, S., He, D., Wu, J., ... & Zhang, W. (2021). A survey of community detection approaches: From statistical modeling to deep learning. *IEEE Transactions on Knowledge and Data Engineering*, *35*(2), 1149-1170.

Wang, Y., Jin, D., Musial, K., & Dang, J. (2019, July). Community detection in social networks considering topic correlations. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 33, No. 01, pp. 321-328).

Jin, D., Wang, H., Dang, J., He, D., & Zhang, W. (2016, February). Detect overlapping communities via ranking node popularities. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 30, No. 1).

Ramirez-Gargallo, G., Garcia-Gasulla, M., & Mantovani, F. (2019, May). TensorFlow on state-of-the-art HPC clusters: a machine learning use case. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (pp. 526-533). IEEE.

Jiang, L., Liu, Y., & Cheng, M. C. (2022). Fast-Accurate Full-Chip Dynamic Thermal Simulation With Fine Resolution Enabled by a Learning Method. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *42*(8), 2675-2688.

Wang, C., Jin, S., Huang, R., Huang, Q., & Chen, Y. (2022). A configurable hierarchical architecture for parallel dynamic contingency analysis on gpus. *IEEE Open Access Journal of Power and Energy*, *10*, 187-194.

Sanbonmatsu, K. Y., & Tung, C. S. (2007). High performance computing in biology: multimillion atom simulations of nanoscale systems. *Journal of structural biology*, *157*(3), 470-480.

Dagum, L., & Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, *5*(1), 46-55.

Nichols, B., Buttlar, D., & Farrell, J. (1996). *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.".

Luebke, D., & Harris, M. (2004, June). General-purpose computation on graphics hardware. In Workshop, SIGGRAPH (Vol. 33, p. 6).

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., ... & Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11* (pp. 97-104). Springer Berlin Heidelberg.

Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107-113.

Apache sparkTM - unified engine for large-scale data analytics. Apache SparkTM - Unified Engine for large-scale data analytics. (n.d.). https://spark.apache.org/

Abhyankar, S., Brown, J., Constantinescu, E. M., Ghosh, D., Smith, B. F., & Zhang, H. (2018). PETSc/TS: A modern scalable ODE/DAE solver library. *arXiv preprint arXiv:1806.01437*.

Harris, C. R., Millman, K. J., Van Der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... & Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357-362.

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. SIAM review, 59(1), 65-98.

Blas (Basic Linear Algebra Subprograms). (n.d.). https://www.netlib.org/blas/

LAPACK - linear algebra package. (n.d.). https://www.netlib.org/lapack/

Masoudi-Sobhanzadeh, Y., Gholaminejad, A., Gheisari, Y., & Roointan, A. (2022). Discovering driver nodes in chronic kidney disease-related networks using Trader as a newly developed algorithm. *Computers in Biology and Medicine*, *148*, 105892.

Kutluana, G., & Türker, İ. (2024). Classification of cardiac disorders using weighted visibility graph features from ECG signals. *Biomedical Signal Processing and Control*, *87*, 105420.

Bonin-Font, F., & Burguera, A. (2020). Towards multi-robot visual graph-SLAM for autonomous marine vehicles. *Journal of Marine Science and Engineering*, *8*(6), 437.

Rahnavard, A., Chatterjee, S., Sayoldin, B., Crandall, K. A., Tekola-Ayele, F., & Mallick, H. (2021). Omics community detection using multi-resolution clustering. *Bioinformatics*, *37*(20), 3588-3594.

Gasparetti, F., Sansonetti, G., & Micarelli, A. (2021). Community detection in social recommender systems: a survey. *Applied Intelligence*, *51*(6), 3975-3995.

Guerrero-Solé, F. (2017). Community detection in political discussions on Twitter: An application of the retweet overlap network method to the Catalan process toward independence. *Social science computer review*, *35*(2), 244-261.

Al-Sharoa, E., Al-Khassaweneh, M., & Aviyente, S. (2018). Tensor based temporal and multilayer community detection for studying brain dynamics during resting state fMRI. *IEEE Transactions on Biomedical Engineering*, *66*(3), 695-709.

Zheng, M., Domanskyi, S., Piermarocchi, C., & Mias, G. I. (2021). Visibility graph based temporal community detection with applications in biological time series. *Scientific reports*, *11*(1), 5623.

Amdahl, G. M. (1967, April). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference* (pp. 483-485).

Zhou, W., Sailani, M. R., Contrepois, K., Zhou, Y., Ahadi, S., Leopold, S. R., ... & Snyder, M. (2019). Longitudinal multi-omics of host–microbe dynamics in prediabetes. *Nature*, *569*(7758), 663-671.

Hebrail, G., & Berard, A. (2012). Individual household electric power consumption data set. *UCI Machine Learning Repository*.