Baltic J. Modern Computing, Vol. 11 (2023), No. 2, pp. 241–256
https://doi.org/10.22364/bjmc.2023.11.2.02

Exploring PGAS-based Gossiping Algorithms for Knödel Graphs*

Vahag BEJANYAN, Hrachya ASTSATRYAN

Institute for Informatics and Automation Problems of National Academy of Sciences of the Republic of Armenia, Yerevan, Paruyr Sevak 1, Armenia

bejanyan.vahag@pm.me, hrach@sci.am

Abstract. Knödel graphs of even order n and degree $1 \leq \delta \leq \log_2(n)$, $W_{\delta,n}$, are regular graphs that have an underlying topology that is time optimal for algorithms gossiping among n nodes. Because of their distinctive properties, Knödel graphs act as a time-optimal topology for broadcasting and gossiping, thus arising in many settings, including social and communication networks or agent-based modeling simulations. Experimentation, often based on the extensive generation and analysis of complex networks, relies on high-performance computational resources to efficiently simulate the flow of information. The efficacy of such processing commonly depends on parallel processing and proper provisioning of distributed resources. This article aims to introduce a runtime in the partitioned global address space model that is optimized for performance and designed to improve the processing of Knödel graphs. The sequential and parallel generation of synthetic datasets, and simulation of push-based, and broadcast-based gossiping algorithms with detailed analysis of resource usage and runtime have been studied.

Keywords: Knödel, Uniform, Generation, Simulation, PGAS, HPC, Broadcast, Gossip

1 Introduction

Network science is a rapidly growing field that analyzes and models complex networks, such as social networks, biological networks, and communication networks. To perform these analyses, efficient algorithms and network topologies are essential. Gossiping and broadcasting algorithms are crucial in many applications of network science (Hedetniemi et al., 1988), like simulating epidemic spread, constructing failure-tolerant communication networks, and achieving effective communication in swarms of unmanned

* The work was partially supported by the Republic of Armenia Science Committee in the frames of the No. 2 1 A G - 1 B 0 5 2 “Self-organized Swarm of UAVs Smart Cloud Platform Equipped with Multi-agent Algorithms and Systems” and No. 20TTAT-RBe016 “Software System for Implementing Fault Tolerant Surveillance and Targeted Tasks Performance of a Collective Artificial Intelligent and Self-Organized Swarm of Drones” projects.

aerial vehicles. These applications require the exchange of information among nodes in the network, and efficient procedures ensure that information is disseminated quickly and effectively.

One promising network topology for achieving efficient gossiping and broadcasting algorithms is Knödel graphs (Fertin and Raspaud, 2004) which are regular graphs of even order n and degree $1 <= \delta <= \log_2(n)$. These regular graphs with unique properties are optimal for broadcasting and gossiping. However, processing Knödel graphs can be computationally intensive, and their analysis requires high-performance computational (HPC) resources.

Researchers commonly use parallel algorithms and HPC systems to overcome these challenges (Li et al., 2015; Liang and McKay, 1995; Chen et al., 1994). The Message Passing Interface (MPI) is a popular tool for distributed memory and hybrid parallel programming simulations (Gropp et al., 1996). However, distributed programming can be complex and challenging due to the diversity of HPC systems. One solution is Partitioned Global Address Space (PGAS), a new paradigm that offers high development productivity and runtime performance (De Wael et al., 2015). PGAS-based frameworks, such as UPCXX (Bachan et al., 2019, 2017), hide communication details and provide a global view of memory, which is beneficial for working with graphs with irregular access patterns and provides a straightforward abstraction for one-sided memory operations.

This article aims to introduce a runtime in the PGAS model that is optimized for performance and designed to improve the processing of Knödel graphs (Sala et al., 2021; Hargrove and Bonachea, 2022). Our approach, which builds upon our previous studies (Bejanyan and Astsatryan, 2021, 2022), aims to enhance computational efficiency and enable faster and more effective graph processing (Bejanyan, n.d.). The study includes sequential and parallel generation of synthetic datasets, push-based and broadcast-based gossiping (Manitara et al., 2022; Ghosh and Ghosh, 2023) algorithms simulation, and a detailed resource usage and runtime analysis. The study provides new insights into the efficient computation of Knödel graphs by evaluating memory usage patterns based on a global memory address space abstraction.

2 Runtime

The proposed runtime architecture consists of three layers: a service layer, a process manager layer, and a resource manager layer. These layers are composed of both software and infrastructure components, as illustrated in Figure 1.

2.1 Service

The service layer is a critical component in the design of microservice-based runtimes. Its primary role is to provide a consistent and reliable interface for the communication between microservices (Richardson, 2018), ensuring that they can work together efficiently, securely, and at scale. This layer establishes a standardized way for microservices to interact with each other, enabling seamless communication and collaboration

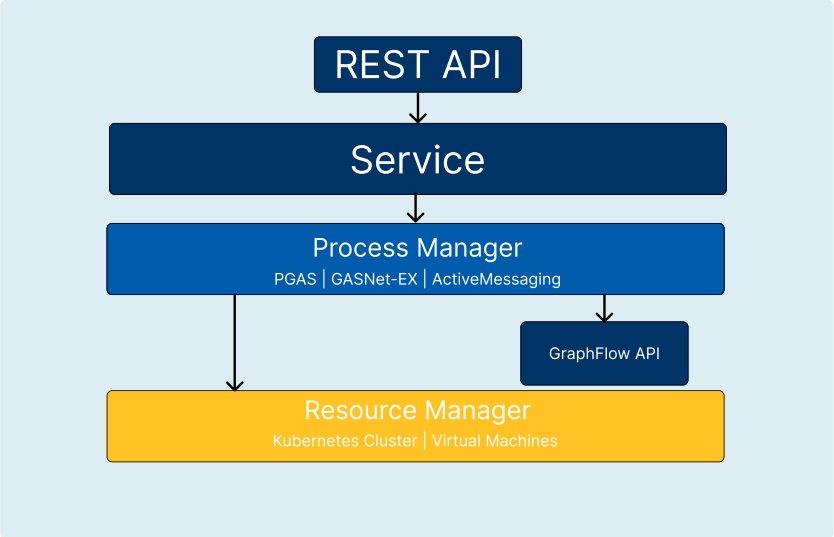


Fig. 1: Runtime architecture layers.

within the cluster. The service layer of a runtime system is accessed through the Representational State Transfer (REST) Application Programming Interface (API) due to its simplicity, scalability, and flexibility. The users rely on standard HTTP GET, POST, PUT, and DELETE methods to initiate computations, access a list of ongoing computations, interrupt or delete them, among other functions. The service layer acts as a gateway to the ”Process Manager” layer, serving a critical function in coordinating the workflow of the cluster.

The requests transmitted to the Service layer are internally redirected to activate the API provided by GraphFlow. When submitting a request to the Service layer, users provide a JSON payload that includes a range of parameters detailing the characteristics of the graph, the selected algorithm, and the necessary computational resources. Section 3 provides a comprehensive overview of the available algorithms, including details on their specific functionalities and requirements. The runtime system can be customized to meet each user's or application's particular needs by providing a flexible and extensible set of algorithms. This enables users to select the most appropriate algorithm for their use case, ensuring the computation is performed accurately and efficiently.

2.2 Process manager

The ”Process manager” layer of the proposed runtime system manages the system's runtime operations (see fig. 2). Its primary function is to launch, monitor, and stop processes as needed to ensure that the cluster operates reliably and efficiently. The layer is composed of both hardware and software components that work together to manage the lifecycle of processes running within the cluster. The software component utilizes PGAS model primitives to manage the execution of processes within the cluster. To im-

plement the PGAS model primitives, our library leverages UPCXX, a C++ library that provides an efficient and scalable implementation of the PGAS programming model.

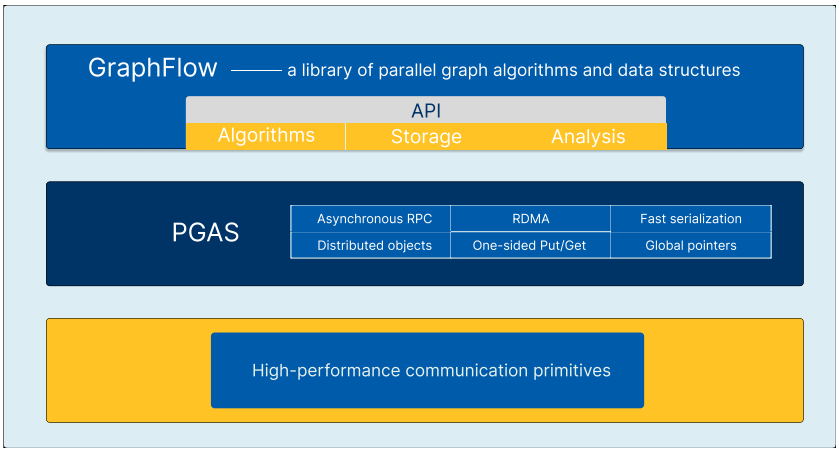


Fig. 2: Architectural diagram of the Process Manager.

The PGAS programming model uses local memories of Virtual Machines (VMs) in an Infrastructure-as-a-Service (IaaS) cloud infrastructure. The Single Program Multiple Data execution model launches a fixed number of program instances at each API call, assigning a unique rank to each instance. The instances communicate asynchronously via Remote Procedure Calls (RPCs), which return futures for explicit blocking or retrieving a return value.

UPCXX integrates PGAS programming primitives, including distributed objects with fast serialization via different network protocols, enabling parallel communication. High-performance communication GASNet-EX (Bonachea and Hargrove, 2018) middleware is integrated into UPCXX to achieve asynchronous communication, such as gossip propagation within the neighborhood of active vertices.

A core component of the proposed runtime, GraphFlow offers parallel graph algorithms and data structures using PGAS programming model and UPCXX library as a foundation. Because the underlying library supports static graphs, fixed-size arrays can be pre-allocated on participating computation nodes, avoiding the high cost incurred by other pointer access or distribution of the hash table. Although this solution is efficient, mapping graph nodes to memory locations on computation nodes requires further optimization.

GraphFlow introduces the concept of Vertex policy, inspired by policy-based design (Alexandrescu, 2001), to enable this mapping. Vertex policy imposes several requirements on a data structure to qualify as a Vertex in GraphFlow. One of these requirements is the presence of a $universalId$, a globally unique identifier that encapsulates sufficient information to support conversion to a node-aware local identifier, enabling $O(1)$

access to a graph partition. Both network models studied in this work provide their implementation to satisfy this requirement.

2.3 Resource manager

The resource manager layer is responsible for managing the allocation and utilization of the available computational resources within the cluster, such as the allocation of VMs, memory reservation, and monitoring. The nano, micro, small, medium, and large type instances of the Armenian hybrid research computing platform have been used for the experiments (Astsatryan et al., 2015).

3 Models and algorithms

In a GraphFlow, a graph $G = (V, E)$ with a vertex set V and an edge set E is generated based on several parameters. The graph is generated in the uniform model, which means that all possible graphs with the same number of vertices and edges are equally likely to be generated. The parameters to generate the graph include the number of vertices $|V|$, the partition density d, and the probability p of two vertices having an edge.Based on this parametrization, GraphFlow estimates an approximate number of edges $|E_n|$ for each partition and populates each using probability p.

Algorithm 1 furnishes a comprehensive account of the generation algorithm employed. The algorithm establishes a connected component on each computational node to construct a graph from a uniform model, laying the foundation for subsequent operations. This initial step is fundamental in achieving the objective of generating a coherent graph in a uniform model and represents a critical aspect of the generation process.

In the next stage, the algorithm augments the density of each connected component by adding edges per the specified requirements (see algorithm 2). This critical stage represents a fundamental aspect of the algorithmic process, as it increases the graph's structural complexity and facilitates the emergence of a more comprehensive network.

In the final stage (see algorithm 3), the algorithm entails the connection of various ranks to the connected components they own. This process further enhances the graph's connectivity and consolidates its constituent nodes' interdependence. Such a step represents a crucial aspect of the algorithmic process, as it promotes the emergence of a more cohesive network capable of facilitating efficient data transfer and analysis.

Figure 3 presents the visualization of one of the possible results of the proposed algorithm.

Contrary to the uniform model, Knödel graphs expose a highly regular structure defined on even $n \geq 2$ nodes with $1 \leq \delta \leq \log_2(n)$ edges with vertices labeled (i, j) where $i \in \{1, 2\}$ and $0 \leq j \leq n/2 - 1$ and an edge between $(1, j)$ and $(2, k)$ where $\forall k \equiv j + 2^p - 1 \pmod{n/2}, p \in \{0, \ldots, \delta - 1\}$.

The Knödel graph is subjected to virtual partitioning among N computational nodes, as demonstrated in algorithm 4. The graph's inherent strictly regular structure permits

Algorithm 1: Core generation algorithm for a uniform graph

Data: $vertexCount \geq 0, percentage \geq 0$
Result: $G_{Uniform} = (V, E)$

```
1  G_Uniform = (V, E) ;                    /* Init distributed, empty graph */
2  rankMe ← upcxx :: rankme();
3  rankN ← upcxx :: rankn();
4  minId ← rankMe * vertexCount;
5  maxId ← (rankMe + 1) * vertexCount − 1;
6  edges ← 0;
7  unconnected ← [minId];
8  connected ← [minId + 1, . . . , maxId];
9  rndGen ← std :: mt19937(std :: randomdevice());
10 weight ← 1 ;                                  /* Distributed variable */
11 while unconnected.length() ≠ 0 do
12     connectedVertexIdx ← connected.peek(rndGen());
13     unconnectedVertexIdx ← unconnected.peek(rndGen());
14     if unconnectedVertexIdx ≠ connectedVertexIdx then
15         if G_Uniform.addEdge(unconnectedVertexIdx, connectedVertexIdx)
             then
16             edges ← edges + 1;
17     unconnected.erase(unconnectedVertexIdx);
18     connected.append(connectedVertexIdx);
```

Algorithm 2: Augmentation of a connected component

```
1  extraEdges ← (vertexCount^2/2 * percentage)/100;
2  copyExtraEdges ← extraEdges;
3  edgesInsideCurrentComponent ← 0;
4  while copyExtraEdges ≠ 0 do
5      startVertexId ← uniformInteger(minId, maxId);
6      endVertexId ← uniformInteger(minId, maxId);
7      if startVertexId ≠ endVertexId then
8          if G_Uniform.addEdge(startVertexId, endVertexId) then
9              edgesInsideCurrentComponent ←
                 edgesInsideCurrentComponent + 1;
10             edges ← edges + 1;
11         copyExtraEdges ← copyExtraEdges − 1;
```

Algorithm 3: Connection of different components

```
1 edgesWithOtherComponents ← 0;
2 for firstRank ← 0 to rankN by 1 do
3     for secondRank ← firstRank + 1 to rankN by 1 do
4         firstId ← generateIdForRank(firstRank, vertexCount);
5         secondId ← generateIdForRank(secondRank, vertexCount);
6         if firstId ≠ secondId then
7             if G_Uniform.addEdge(firstId, secondId) then
8                 edgesWithOtherComponents ←
                    edgesWithOtherComponents + 1;
9                 edges ← edges + 1;
```

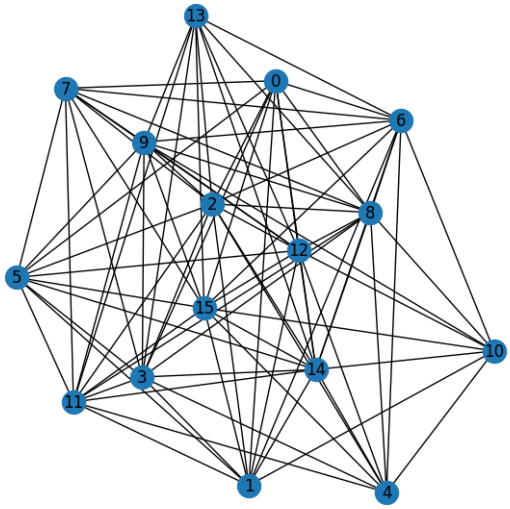


Fig. 3: Demo graph generated from uniform model.

a seamless mapping onto computational nodes, thereby enabling parallel processing of subsets of vertices and edges. The parallelization significantly reduces the algorithm runtime.

Figure 4 presents the visualization of the proposed algorithm.

GraphFlow is a generic framework for gossiping algorithms that frees users from concerns about the underlying data structure. Gossiping algorithms disseminate information across a network by having nodes randomly select and communicate with other nodes. The algorithm is iterative, and at each iteration, nodes randomly select other

Algorithm 4: Generation of Knödel graph

Data: $vertexCount \geq 0, percentage \geq 0, delta \geq 0$
Result: $G_{Knodel} = (V, E)$

$G_{Knodel} = (V, E)$; `/* Init distributed, empty graph */`
$vertexCountPerRank \leftarrow (vertexCount/2 + rankn() - 1)/rankn()$;
$deltaPerRank \leftarrow \log_2(vertexCountPerRank)$;
for $j \leftarrow minId$ **to** $maxId \pmod{vertexCount}$ **by** 1 **do**
    $from \leftarrow (0, j, vertexCount/2)$;
    for $d \leftarrow 0$ **to** $delta$ **by** 1 **do**
        $to \leftarrow (1, j + 2^d - 1 \pmod{vertexCount/2}, vertexCount/2)$;
        if $G_{Knodel}.addEdge(from, to)$ **then**
            $edges \leftarrow edges + 1$;

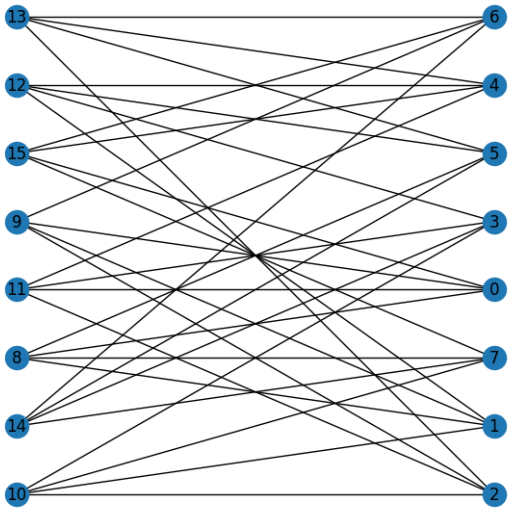


Fig. 4: A Knödel graph with 16 vertices and a degree equal to 4 generated by the suggested algorithm.

nodes to communicate with. GraphFlow's generic nature enables developers to implement gossiping algorithms using any data structure that can act as a graph vertex. GraphFlow imposes minimal restrictions on the data structure, requiring only the implementation of methods for sending and receiving messages, updating the state, and computing the next set of neighbors to communicate with. The gossiping algorithms discussed in this work are generic and follow the same steps regardless of the underlying data structure used to represent the graph vertices. This flexible approach enhances the algorithms' reusability and ease of implementation, enabling testing and application in diverse contexts.

The push-randomized gossip algorithm operates in parallel across multiple computation nodes, as described in algorithm 5. Initially, each rank selects a random vertex from the partition it owns, initiating the gossiping process. The rank of the identified vertex is then computed, and an asynchronous RPC is launched to transmit the gossip to the node where the vertex resides, considering the vertex's affinity to the computational node. After updating the vertex with the new information, a random neighbor is selected and returned. The algorithm converges when the number of visited vertices equals the partition size of the graph owned by the current rank.

Algorithm 5: Push-randomized gossiping algorithm

Data: $G = (V, E), data$
Result: $G = (V, E)$
`/* Distributed container of visited vertex ids */`
1 $visited \leftarrow []$
`/* Gossiping starts from a random vertex */`
2 $randomVertex \leftarrow getRandomVertex(G)$
`/* Waiting for all ranks to agree on initial random vertices */`
3 $upcxx :: barrier()$
4 $nextVeretxId \leftarrow randomVertex.id$
5 **while** $visited.size() \neq G.localStorage().size()$ **do**
6 $vertexIdRank \leftarrow G.getVertexRank(nextVertexId)$
`/* Executing gossiping algorithm on a rank owning the vertex */`
7 $nextVertexId \leftarrow upcxx ::$
$rpc(vertexIdRank, pushRandomizedGossip, nextVertexId, data, visited).wait()$

The lambda function (see algorithm 6) executes whenever a vertex is visited to update the current state of the gossip and select a random neighbor.

The broadcast-based approach (see algorithm 7) is similar in structure to the previous method but operates on frontlines, a set of neighbors, instead of a single neighbor on each computation node. This technique enhances data transmission and processing efficiency on parallel computing platforms and mitigates load imbalances. It also improves performance for algorithms requiring message-passing among neighboring nodes, such as network analysis and graph partitioning. However, its effectiveness may vary depending on factors like the graph structure and number of neighbors per node.

Algorithm 6: A lambda asynchronously executed when visiting next node

```
   Data: G = (V, E), data
   Result: nextVertexId
1  vertex ← getVertex(vertexId)
2  vertex.data ← data
3  randomNeighbourId ← uniformInteger(0, vertex.neighbourhood.size())
4  return randomNeighbourId
```

Using frontlines in the broadcast-based approach offers an efficient, load-balanced, and improved-performance alternative to the single-neighbor process.

Algorithm 7: Broadcast-based gossiping algorithm

```
   Data: G = (V, E), data
   Result: G = (V, E)
   /* Distributed container of visited vertex ids                    */
1  visited ← []
   /* Gossiping starts from a random vertex                          */
2  randomVertex ← getRandomVertex(G)
   /* Waiting for all ranks to agree on initial random vertices      */
3  upcxx :: barrier()
4  nextVeretxId ← randomVertex.id
5  while visited.size() ≠ G.localStorage().size() do
6      vertexIdRank ← G.getVertexRank(nextVertexId)
       /* Executing gossiping algorithm on a rank owning the vertex  */
7      nextVertexIds ← upcxx ::
         rpc(vertexIdRank, broadcastGossip, nextVertexId, data, visited).wait()
       /* Wait for each rank to transfer gossip to its neighbours and
          calculate ids for the frontline                            */
8      upcxx::barrier(); frontline ← []
9      for j ← 0 to nextVertexIds.size() by 1 do
10         frontline.append(transferGossip
11             (G.GetGraphStorage(), nextVertexIds[j], data))
12     for j ← 0 to frontline.size() by 1 do
13         frontline[j].wait()
14     upcxx :: barrier()
```

The lambda function in algorithm 8 is executed asynchronously during broadcasting. It has a structure similar to that of algorithm 6, but with one primary difference. Instead of selecting the next node randomly, the algorithm constructs a frontline for concurrent traversal in the next gossip iteration. This approach improves processing and communication efficiency among computational nodes during graph-based computations.

Algorithm 8: A lambda asynchronously executed to construct frontline

Data: $G = (V, E), data$
Result: $nextVertexId$
1 $vertex \leftarrow getVertex(vertexId)$
2 $vertex.data \leftarrow data$
3 $neighbourhood \leftarrow collectNeighbourhood(vertex)$
4 **return** $neighbourhood$

Algorithm 9: A lambda concurrently executed to update frontline

Data: $G = (V, E), data$
Result: $nextVertexId$
1 $vertex \leftarrow getVertex(vertexId)$
2 $vertex.data \leftarrow data$
3 **return** $neighbourhood$

Algorithm 9 describes a parallel procedure executed after collecting IDs for the current step. The separation of ID collection and the actual update process provides greater control over data flow and facilitates finer-grained synchronization. This is particularly valuable in distributed memory systems, where synchronization can be costly and time-consuming.

4 Evaluation

This section presents the experimental findings of the algorithms discussed in Section 2.

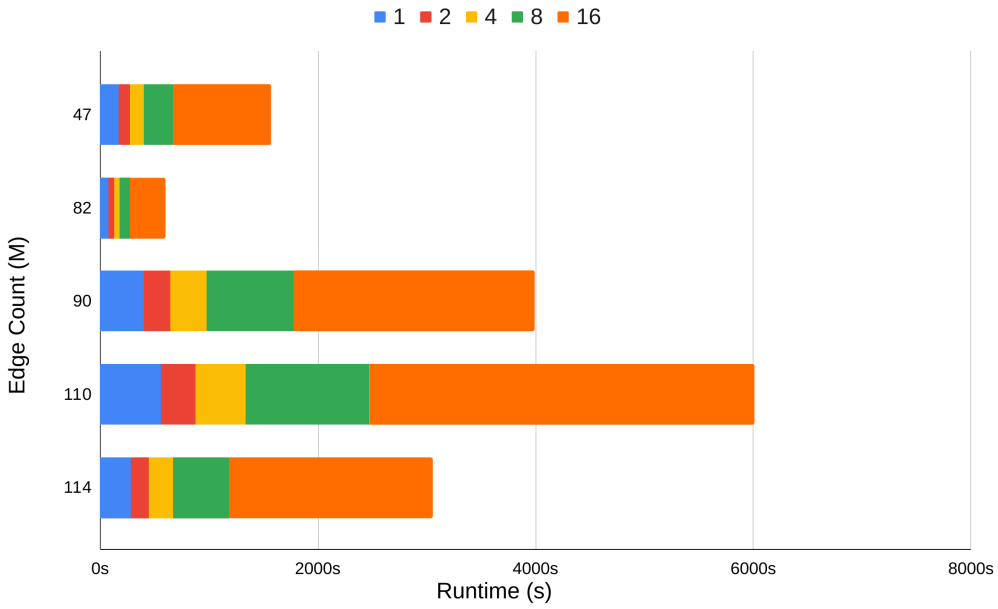
Figure 5 shows the benchmark results for uniform and Knödel graphs generation.

The plot suggests a weak correlation between the number of vCPUs and the number of edges (see fig. 5a), resulting in a negative impact on the runtime of uniform graph generation. Unlike the uniform model, the benchmark results of parallel Knödel graph generation (see fig. 5b) exhibit near-linear scalability, which can be attributed to the inherent properties of the Knödel graph and the algorithm presented in algorithm 4.

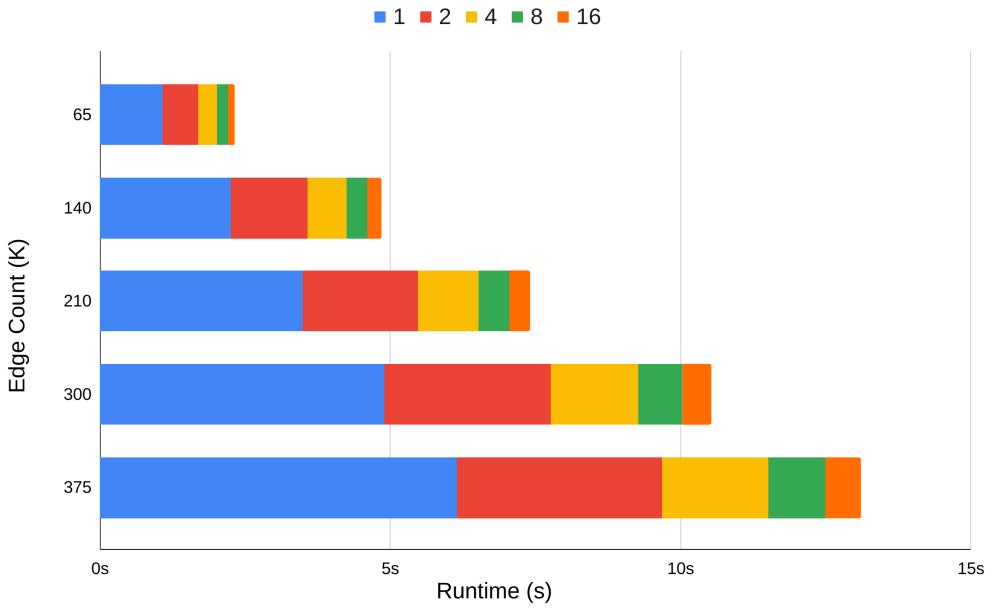
Uniform graphs typically have irregular and random structures. Therefore, Figure 6a, which shows the benchmark results of executing push-randomized gossip algorithms, does not display any significant correlation.

In Figure 6b, a strong negative correlation between the number of vCPUs and graph edges is evident, especially when the number of vCPUs is two. This trend is due to the synchronization required for reliable communication between graph partitions. However, for vCPU numbers of $4, 8$, and 16, the runtime for execution decreases by almost 50% due to the parallelism.

In Figure 7a, the number of vCPUs has a significant positive impact on the execution time within each simulation batch.

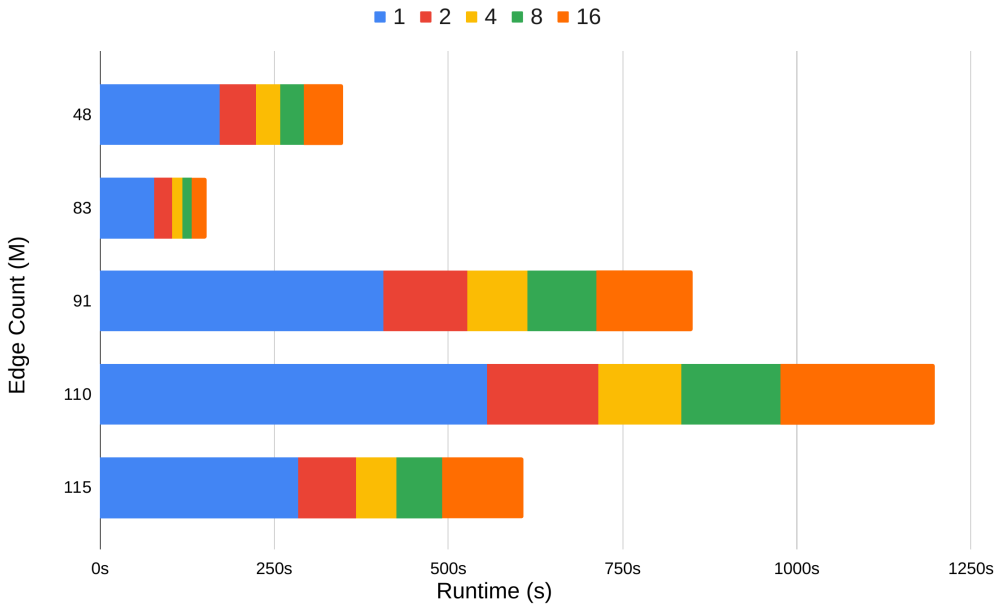


(a) Uniform graph generation.

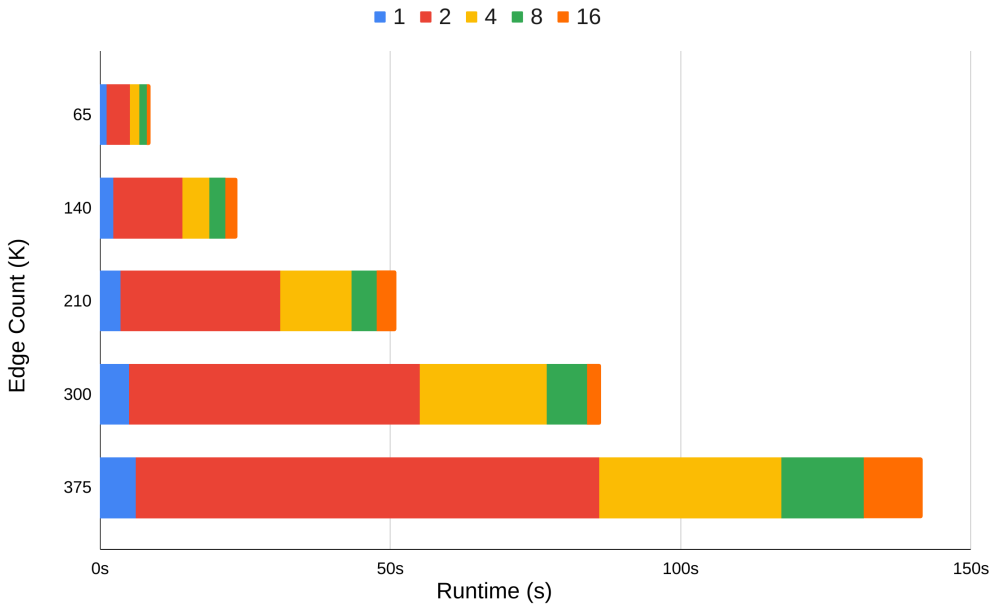


(b) Knödel graph generation.

Fig. 5: Benchmark results for uniform and Knödel graphs generations.

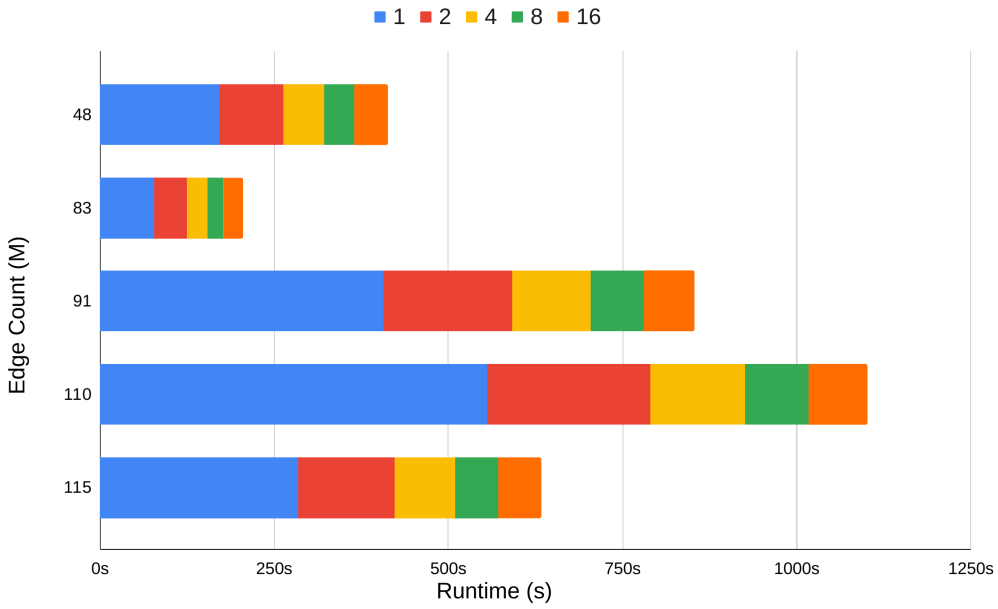


(a) Uniform graphs.

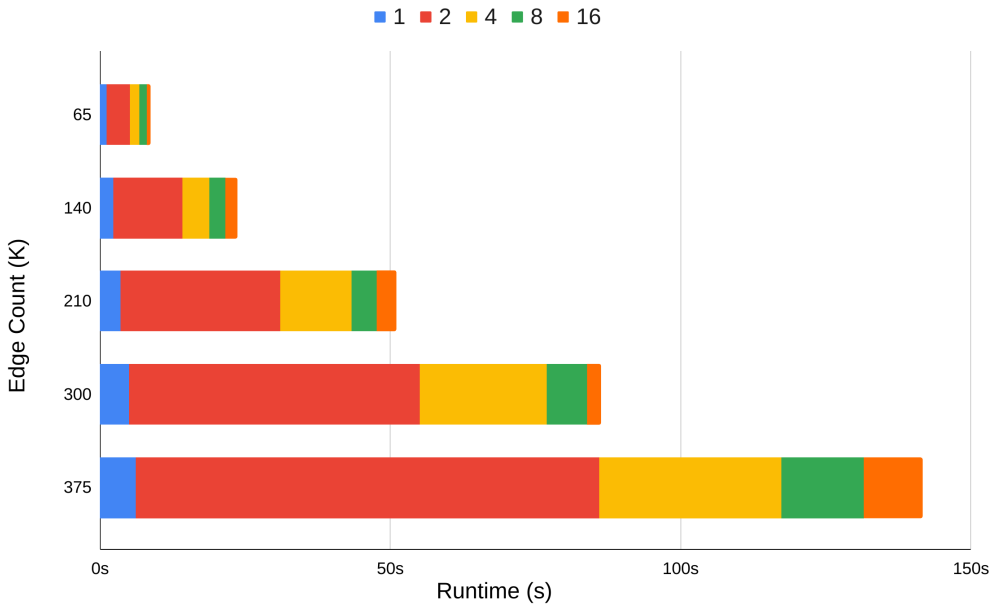


(b) Knödel graphs.

Fig. 6: Benchmark results for push-randomized gossiping in uniform and Knödel graphs



(a) Uniform graphs.



(b) Knödel graphs.

Fig. 7: Benchmark results for broadcast-based gossiping in uniform and Knödel graphs.

Figure 7b shows the benchmark results for broadcast-based gossiping in the Knödel graph. Similar to the push-randomized gossip algorithm on Knödel graphs, there is a communication overhead for the benchmarks with a vCPU number of two. However, for four or more vCPUs, the execution time decreases by over 50% compared to the two vCPU cases and over 90% compared to the push-randomized scenario, with a runtime of approximately 10ms instead of 100ms.

5 Conclusion

In conclusion, this study has provided valuable insights into the performance of randomized push-first and broadcast-based algorithms on uniform and Knödel graphs. Our results demonstrate the significant impact of graph structure on broadcasting scalability, with Knödel graphs exhibiting high scalability in a parallel setting, while broadcasting over uniform graphs may have unpredictable runtime due to the lack of structure.

The study also highlights the potential for nearly linear scalability in parallel Knödel graph generation due to the regular structure of the graph. Based on these findings, future research will focus on developing effective partitioning strategies for graph generation and exploring caching mechanisms to minimize communication between processes and network nodes.

The study also aims to extend existing and new algorithms for accelerated computing, which could improve performance. Finally, exploring generally distributed algorithms such as clocks, consensus, consistency, and replication in the PGAS model may provide insights into designing more efficient and scalable parallel algorithms.

Overall, the findings of this study provide valuable insights for researchers and practitioners in the field of parallel and distributed computing, highlighting the importance of graph structure in achieving scalability and suggesting new avenues for future research.

References

Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley Longman Publishing Co., Inc., USA.

Astsatryan, H., Sahakyan, V., Shoukourian, Y., Dongarra, J., Cros, P.-H., Dayde, M., Oster, P. (2015). Strengthening compute and data intensive capacities of armenia, *2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER)*, pp. 28–33.

Bachan, J., Baden, S. B., Hofmeyr, S., Jacquelin, M., Kamil, A., Bonachea, D., Hargrove, P. H., Ahmed, H. (2019). UPC++: A High-Performance Communication Framework for Asynchronous Computation, *Proceedings of the 33rd IEEE International Parallel & Distributed Processing Symposium*, IPDPS, IEEE.

Bachan, J., Bonachea, D., Hargrove, P. H., Hofmeyr, S., Jacquelin, M., Kamil, A., van Straalen, B., Baden, S. B. (2017). The UPC++ PGAS library for exascale computing, *Proceedings of the Second Annual PGAS Applications Workshop*, PAW17, ACM, New York, NY, USA, pp. 7:1–7:4.

Bejanyan, V. (n.d.). GraphFlow.
https://github.com/lnikon/graphflow

Bejanyan, V., Astsatryan, H. (2021). Vm based evaluation of the scalable parallel spanning tree algorithm for pgas model, *Proceedings of the 9th International Conference "Distributed Computing and Grid Technologies in Science and Education" (GRID'2021), Dubna, Russia, July 5-9, 2021*, pp. 541–547.

Bejanyan, V., Astsatryan, H. (2022). A pgas-based implementation for the parallel minimum spanning tree algorithm, *Large-Scale Scientific Computing: 13th International Conference, LSSC 2021, Sozopol, Bulgaria, June 7–11, 2021, Revised Selected Papers*, Springer, pp. 431–438.

Bonachea, D., Hargrove, P. H. (2018). GASNet-EX: A High-Performance, Portable Communication Library for Exascale, *Proceedings of Languages and Compilers for Parallel Computing (LCPC'18)*, Vol. 11882 of *Lecture Notes in Computer Science*, Springer International Publishing. Lawrence Berkeley National Laboratory Technical Report (LBNL-2001174).

Chen, Y.-W., Horng, S.-J., Kao, T.-W., Tsai, H.-R., Tsai, S.-S. (1994). Parallel connectivity algorithms on permutation graphs, *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN)*, pp. 97–104.

De Wael, M., Marr, S., De Fraine, B., Van Cutsem, T., De Meuter, W. (2015). Partitioned global address space languages, *ACM Comput. Surv.* **47**(4).

Fertin, G., Raspaud, A. (2004). A survey on knödel graphs, *Discrete Applied Mathematics* **137**(2), 173–195.

Ghosh, R. K., Ghosh, H. (2023). *Gossip Protocols*, Vol. Distributed Systems: Theory and Applications, pp. 253–282.

Gropp, W., Lusk, E., Doss, N., Skjellum, A. (1996). A high-performance, portable implementation of the mpi message passing interface standard, *Parallel Computing* **22**(6), 789–828.

Hargrove, P. H., Bonachea, D. (2022). Gasnet-ex rma communication performance on recent supercomputing systems, *The 5th Annual Parallel Applications Workshop, Alternatives To MPI+X (PAW-ATM'22)*, pp. 1–7.

Hedetniemi, S. M., Hedetniemi, S. T., Liestman, A. L. (1988). A survey of gossiping and broadcasting in communication networks, *Networks* **18**(4), 319–349.

Li, B., Gao, Z., Niu, J., Lv, Y., Zhang, H. (2015). Vertex-centric parallel algorithms for identifying key vertices in large-scale graphs, *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pp. 225–231.

Liang, W., McKay, B. (1995). Fast parallel algorithms for testing k-connectivity of directed and undirected graphs, *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*, Vol. 1, pp. 437–441 vol.1.

Manitara, N. E., Rikos, A. I., Hadjicostis, C. N. (2022). Privacy-preserving distributed average consensus in finite time using random gossip, *2022 European Control Conference (ECC)*, pp. 1282–1287.

Richardson, C. (2018). *Microservices Patterns: With examples in Java*, Manning.

Sala, K., Macià, S., Beltran, V. (2021). Combining one-sided communications with task-based programming models, *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 528–541.

Received February 28 , revised March 16, 2023, accepted March 22, 2023