VM BASED EVALUATION OF THE SCALABLE PARALLEL MINIMUM SPANNING TREE ALGORITHM FOR PGAS MODEL

V. Bejanyan[a], H. Astsatryan

Institute for Informatics and Automation Problems of the National Academy of Sciences of the Republic of Armenia

E-mail: [a] bejanyan.vahag@protonmail.com

The minimum spanning tree problem has influential importance in computer science, network analysis, and engineering. However, the sequential algorithms become unable to process the given problem as the volume of the data representing graph instances overgrowing. Instead, the high-performance computational resources pursue to simulate large-scale graph instances in a distributed manner. Generally, the standard shared or distributed memory models like OpenMP and Message Passing Interface are applied to address the parallelization. Nevertheless, as an emerging alternative, the Partitioned Global Address Space model communicates in the form of asynchronous remote procedure calls to access distributed-shared memory, positively affecting the performance using overlapping communications and locality-aware structures. The paper presents a modification of the Kruskal algorithm for MST problems based on performance and energy-efficiency evaluation relying on emerging technologies. The algorithm evaluation shows great scalability within the server up to 16 vCPU and between the physical servers coupled with a connected weighted graph using different vertices, edges, and densities.

Keywords: Minimum spanning tree, PGAS model, parallel algorithms, large-scale graphs, Kruskal, VM

1. Introduction

Large scale graph analysis has tremendous importance in network science, social network analysis, and many other fields. However, due to limitations of memory and computational performance of a single physical server, large scale graph problems often tend to be solved on High-Performance Computational (HPC) clusters. The parallel programming models may address this challenge, such as Message Passing Interface (MPI) or OpenMP for distributed memory and shared memory systems [1]. However, traditional models are limited to relying on emerging technologies such as portable, open-source, high-performance communication GASNet-EX library to address networking requirements of runtime systems [2]. As an alternative to traditional parallelization approaches, the Partitioned Global Address Space (PGAS) is a distributed memory programming model providing asynchronous peer-to-peer communication and shared distributed memory capabilities and bases on GASNet-EX [3, 4]. In such a framework, the PGAS model may export a portion of the process address space into a global heap. Therefore, the model simplified access to the distributed memory, leverage better locality via locale affinity, and perform remote procedure calls (RPC) on the data structures stored in a distributed shared memory, for instance, large scale graphs transmission in distributed memory infrastructures.

Several parallel minimum spanning tree (MST) algorithms are available using traditional or emerging parallel programming models [5, 6, 7] with the limited performance and energy efficiency evaluations. The paper aims to present a modification of the Kruskal [8] algorithm for MST problems based on performance evaluation relying on emerging technologies. In the first step of Kruskal's algorithm, a pair of rates with the nearest distance and a line proportional to the distance is selected. Then a pair with the second closest distance connects the nearest pair that is not connected by the same tree. The suggested algorithm focuses on a high-performance C++ UPCXX framework [2] for enabling high-performance simulations through the asynchronous communication framework. The algorithm has been implemented in the scope of the open-source PGAS based graph algorithms library [3].

In Section 2, the methodology of the experiments and algorithms are presented. Then, the experimental results are discussed in Section 3, while the conclusion is presented in Section 4.

2. Methodology

The graph analysis, HPC communication middleware, PGAS, and HPC over cloud framework (see figure 1) are the layers of the suggested methodology for parallel implementation of MST algorithm for large graphs.

The HPC over cloud layer depends on the IaaS (Infrastructure as a Service) cloud service of the Armenian hybrid research computing platform [9]. The PGAS programming model constructs a global memory address space combining the local memories of VMs of IaaS cloud infrastructure without any memory allocation or pinning. As a single program multiple data technique, UPCXX binary code predetermines several single program copies for distributing among processes located on different physical and virtual machines. Each process is identified by a unique *rank* to organize the computations and communicates between the physical and virtual machines. The asynchronous RPCs provided by UPCXX are critical to executing arbitrary functions inside RPC given destination rank. It is possible to achieve better simulation time considering the communication overlapping and waiting on the future returned by RPC only when there is no current work to finish. An example of such communication is a transfer of the candidate MST tree portion between ranks.

The distributed objects provided by UPCXX have been implemented for graph algorithms to store structures necessary for inter-rank communication, such as edge lists. The graph data structures are stored in processes private memory in consecutive memory locations to avoid the overhead caused by RPC. It is assumed that $G=(V, E)$ is a connected graph with distinct edge weights and a unique MST. Besides, it is required unique *id* assigned to each graph vertex. A slightly modified adjacency list stores the graph data structure internally. Instead of keeping pointers to the neighbors, each vertex stores list of unique *id*s of the neighbors. Such *id* is then used to retrieve the pointer to that neighbor from the vertex store. Vertices are distributed among all the computation nodes in a vector to store vertex *id* as key and private pointer as a value. Such distribution provides more significant locality and better performance for memory operations because computation with each vertex is done on the vertex's node to which the vertex has an affinity.

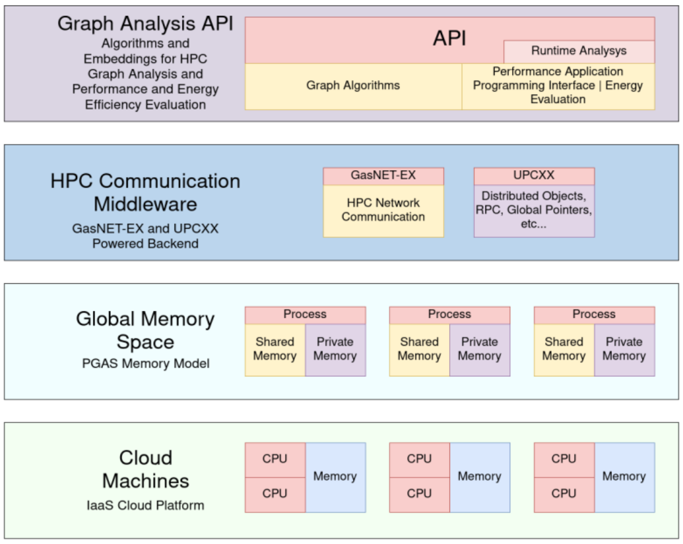
Graph Analysis API
Algorithms and Embeddings for HPC Graph Analysis and Performance and Energy Efficiency Evaluation
API
Runtime Analysys
Graph Algorithms
Performance Application Programming Interface | Energy Evaluation
HPC Communication Middleware
GasNET-EX and UPCXX Powered Backend
GasNET-EX
HPC Network Communication
UPCXX
Distributed Objects, RPC, Global Pointers, etc...
Global Memory Space
PGAS Memory Model
Process
Shared Memory
Private Memory
Cloud Machines
IaaS Cloud Platform
CPU
Memory

Figure 29 Architecture overview diagram

At the beginning of the algorithm, each rank finds its portion of the MST. Then a distributed merging process starts to merge each rank of the MST with its neighbors MST. This procedure continues until only one rank remains: the master rank and contains all the MST edges.

3. Experiments

The nano, micro, small, medium, and large type instances of the Armenian IaaS cloud have been used for the experiments with the following hardware and software configurations:

- Hardware - Intel Xeon 1.99 GHz processor with 1-16 cores, 16-32 GB RAM;
- Software - Ubuntu 20.04.1 OS with Linux kernel version 5.4, UPCXX of version 2021.3.0 with a shared heap of size 1G, and GCC 9.3.0 compiler with C++11/14/17 standard features.

Table 1 summarizes parameters of the cloud VM instances used in the evaluation.

Table 1. Overview of the computational architecture.

Instance size	**Memory (GiB)**	**vCPU**
am16.nano	16	1
am32.medium	32	1
am16.2xnano	16	2
am32.2xmedium	32	2
am16.micro	16	4
am32.xlarge	32	4
am16.2xmicro	16	8
am32.2xlarge	32	8
am16.small	16	16
am32.4xlarge	32	16

The graph generation and MST simulations are evaluated for the experiments. The complexity of the graph generation algorithm depends on the sizes of the vertices and the percentage of graph connectivity input parameters. The suggested graph generation algorithm is divided into three phases.

First, the core of the graph is generated to ensure that the graph will remain connected at later stages. Secondly, the algorithm starts to generate edges inside the current connected component by uniformly choosing vertices. And at the final third phase, various connected components assigned to different processes or machines are connected by randomly adding edges between vertices inside different connected components and adding an edge between them.

The experiments have been carried out with a fixed number of vertices and increasing densities, enabling to benchmark both graph generation and MST algorithm for sparse and dense graphs incrementally. Both algorithms have been evaluated using VMs with different configurations (see table 1).

Figure 2 presents the behavior of the suggested graph generation algorithm delivering great runtime for the one and two vCPUs cases. However, in later cases, communication and synchronization costs are getting higher during the third phase. The high cost incurred by the third phase is caused by repeatedly accessing the memory of other processes.

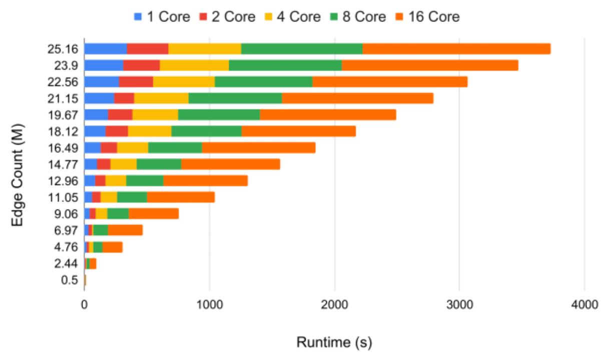

Figure 2. Incremental evaluation of the MST algorithm for VMs over two physical servers

Figure 3 shows the behavior of the presented MST algorithm. The algorithm delivers high scalability over 16 share memory vCPUs, which is mainly achieved by localizing computations and reducing communication only to transfer the small portion of the MST between processes of ranks and overlapping the communication. For example, running on the experimental setup described in table 1, the shared memory algorithm achieves runtime equal to *2.46* seconds for *0.5* million edges with *1* vCPU while for *16* vCPUs runtimes are equal to *0.73* seconds, which means that even for small graph instances the algorithm has scalability over multiple processes. Therefore, the speed up for *16* vCPUs is *3.36* times. At the same time, on the graph with *27.5* million edges, the runtime of the algorithm is *78.73* seconds for one vCPU, while for *16* vCPUs runtimes are nearly equal to *9.18* seconds for each process and speed up is equal to 8.6 times.

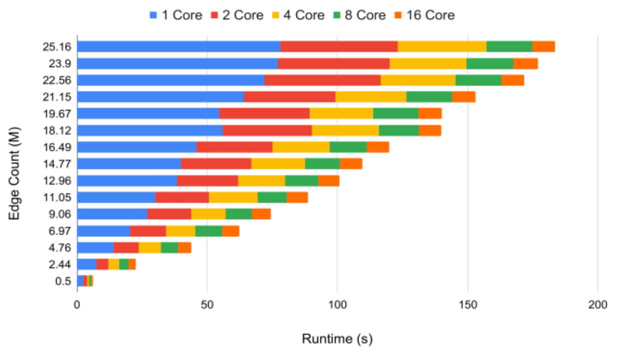

Figure 3. Incremental benchmark of the MST algorithm for a shared memory

After each step of the merging process, half of the computational nodes are done with their work, and hence memory related to the MST algorithm execution is freed to save space in a global heap. The overview of the utilization of the memory and vCPU are presented in Table 2.

Table 2. Overview memory a vCPU utilization

Feature	**Minimum**	**Average**	**Maximum**
Memory (GiB)	0.97	1.60	2.80
CPU (%)	0.0	49.2	100

Figure 4 presents the behavior of the suggested graph generation algorithm for VMs over two physical servers. The algorithm delivers great runtime for the one node benchmark with nearly identical two the shared memory case with execution time. However, in a two-node case, runtime increases because of communication and synchronization costs.

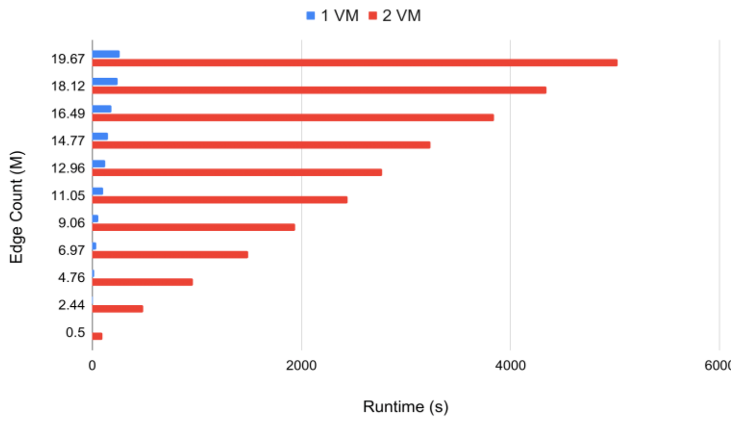

Figure 4. Incremental evaluation of the MST algorithm for VMs over two physical servers

Figure 5 shows the behavior of the presented MST algorithm for a distributed benchmark.

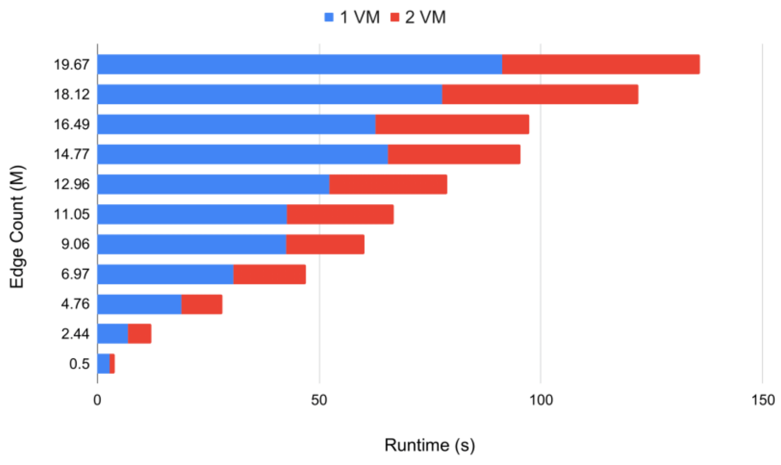

Figure 5. Incremental evaluation of the MST algorithm for a distributed run

The algorithm achieves a great scalability over two VMs even though communication and synchronization costs in a distributed case are much higher than shared memory vCPUs. For example, running on the experimental setup described in table 1, the distributed memory algorithm achieves runtime equal to *2.72* seconds for *0.5* million edges with *1* VM while for *2* VMs runtimes are equal to *1.17* seconds, which is higher than for the shared memory case because of additional costs incurred by inter-node communication. The speed up is equal to 2.42 times. Simultaneously, on the graph with *20* million edges, the runtime of the algorithm is *91.28* seconds for 1 VM, while for *2* VMs runtimes are nearly equal to *44.53* seconds for VM and speed is equal to 2 times.

Memory, vCPU and network utilization are shown in Table 3.

Table 3. Overview memory a vCPU utilization

VM 1	**Minimum**	**Average**	**Maximum**
Memory (GiB)	0.3	1.1	3.5
CPU (%)	0.0	14.8	99.5
Net In	34.5	311.3k	1.2M
Net Out	428.2	314.6k	1.2M
VM 2	**Minimum**	**Average**	**Maximum**
Memory (GiB)	0.3	0.4	1.0
CPU (%)	0.0	12.7	95.5
Net In	27.6	299.5k	1.2M
Net Out	424.4	297.9k	1.2M

4. Conclusion

The article presents a distributed algorithm for finding MST in a PGAS model. The suggested algorithm is a modification of Kruskal's algorithm. An in-depth evaluation of the proposed MST performance has been performed in the cloud on a connected weighted graph with different vertices, edges, and densities modeling sparse and dense graphs. The experimental results show that the algorithm has high scalability over 16 threads for MST problem in a shared memory setup. However, random graph generation time tends to increase due to communication and synchronization costs incurred by multiprocessing. During the distributed run, MST has again shown high scalability over two VMs where communication and synchronization costs are much higher compared to the shared memory case, even for small graphs instances. Still, graph generation's run-time and scalability have suffered due to irregular access patterns at the third phase of the graph generation algorithm.

It is planned to study and develop algorithms for distributed large graphs in the PGAS model considering chunk-sizes, communication costs and network optimizations [10], as well as to develop graph algorithms for centrality, shortest paths, and link analysis using emerging distributed programming languages and HPC technologies like Chapel, InfiniBand or Remote Direct Memory Access [11].

5. Acknowledgement

The paper is supported by the European Union's Horizon 2020 research infrastructures programme under grant agreement No 857645, project NI4OS Europe (National Initiatives for Open Science in Europe).

References

[1] Diaz, J., Muñoz-Caro, C., & Niño, A. (2012). A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. IEEE Transactions on Parallel and Distributed Systems, 23, 1369-1386. doi:10.1109/TPDS.2011.308

[2] Bonachea, D., & Hargrove, P. H. (2019). GASNet-EX: A High-Performance, Portable Communication Library for Exascale. In M. Hall, & H. Sundar (Ed.), Languages and Compilers for Parallel Computing (pp. 138–158). Cham: Springer International Publishing.

[3] Bejanyan, V., & Astsatryan, H. (2021). MST PGAS algorithm. MST PGAS algorithm. Retrieved from https://github.com/lnikon/pgas-graph

[4] Yelick, K., Bonachea, D., Chen, W.-Y., Colella, P., Datta, K., Duell, J., . . . Wen, T. (2007). Productivity and Performance Using Partitioned Global Address Space Languages. (pp. 24–32). New York, NY, USA: Association for Computing Machinery. doi:10.1145/1278177.1278183

[5] Bader, D. A., & Cong, G. (2006). Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. Journal of Parallel and Distributed Computing, 66, 1366-1378. doi:https://doi.org/10.1016/j.jpdc.2006.06.001

[6] Cong, G., Almasi, G., & Saraswat, V. (2010). Fast PGAS Implementation of Distributed Graph Algorithms. SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, (pp. 1-11). doi:10.1109/SC.2010.26

[7] Gallager, R. G., Humblet, P. A., & Spira, P. M. (1983, January). A Distributed Algorithm for Minimum-Weight Spanning Trees. ACM Trans. Program. Lang. Syst., 5, 66–77. doi:10.1145/357195.357200

[8] West, D. B. (2000, September). Introduction to Graph Theory (2 ed.). Prentice Hall.

[9] Shoukourian, Y. H., Sahakyan, V. G., & Astsatryan, H. V. (2013). E-Infrastructures in Armenia: Virtual research environments. Ninth International Conference on Computer Science and Information Technologies Revised Selected Papers, (pp. 1–7). doi:10.1109/CSITechnol.2013.6710360

[10] Astsatryan, H., Narsisian, W., Kocharyan, A., Da Costa, G., Hankel, A., & Oleksiak, A. (2017). Energy optimization methodology for e-infrastructure providers. Concurrency and Computation: Practice and Experience, 29, e4073.

[11] Jenkins, L., Firoz, J. S., Zalewski, M., Joslyn, C., & Raugas, M. (2019, September). Graph Algorithms in PGAS: Chapel and UPC++. In 2019 IEEE High Performance Extreme Computing Conference (HPEC) (pp. 1-6). IEEE.