Parallel implementation of minimum spanning tree algorithms using MPI

Vladimir Lončar* and Srdjan Škrbic**
Faculy of Science, Depatment for Mathematics an Informatics, University of Novi Sad, Serbia
* vlada.loncar@gmail.com
** shkrba@uns.ac.rs

***Abstract*—In this paper we study parallel algorithms for finding minimum spanning tree of a graph. We present two algorithms, based on sequential algorithms of Prim and Kruskal, targeting message passing parallel machine with distributed memory. First algorithm runs in $O(n^2/p+n\log p)$ and second algorithm runs in $O(n^2/p+n^2\log p)$.**

***Index Terms*—Minimum spanning tree, parallel algorithms, message passing, distributed memory computer.**

I. Introduction

A minimum spanning tree (MST) of a graph $G = (V, E)$ is a subset of E that forms a spanning tree of G with minimum total weight. MST problem has many applications in computer and communication network design, as well as indirect applications in fields such as computer vision and cluster analysis [1].

In this paper we implement two parallel algorithms for finding MST of a graph, based on classical algorithms of Prim and Kruskal. Algorithms target message passing parallel machine with distributed memory. Primary characteristic of this architecture is that the cost of interprocess communication is high in comparison to cost of computation. Our goal was to develop algorithms which minimize communication, and to measure the impact of communication on the performance of algorithms. Our primary interest were graphs which have significantly larger number of vertices than processors involved in computation. Since graphs of this size cannot fit into a memory of single process, we use simple partitioning scheme to divide the input graph among processes. We considered both sparse and dense graphs.

First algorithm is a parallelization of Prim's sequential algorithm. Each process is assigned a subset of vertices and in each step of computation, every process finds a *candidate* minimum-weight edge connecting one of it's vertices to MST. Leader process collects those candidates and selects one with minimum weight which it adds to MST, and broadcasts result to other processes. This step is repeated until every vertex is in MST.

Second algorithm is based on Kruskal's approach. Processes get a subset of G in the same way as in first algorithm, and then find local minimum spanning tree (or forest). Next, processes merge their MST edges until only one process remains, which holds edges that form MST of G.

Algorithms we present are both easy to understand and implement, and since they use fixed communication patterns, their performance can easily be predicted.

II. Related Work

Algorithms for MST problem have mostly been based on one of three approaches, that of Boruvka [2], Prim [3] and Kruskal [4], however, a number of new algorithms has been developed. Gallager et al. presented an algorithm where processor exists at each node of the graph (thus $n = p$), useful in computer network design [5]. Katriel and Sanders designed an algorithm exploiting cycle property of a graph targeting dense graph, [6], while Ahrabian and Nowzari-Dalini's algorithm relies on depth first search of the graph [7].

Due to it's parallel nature, Boruvka's algorithm (also known as Sollin's algorithm) has seen the most research. Examples of algorithms based on Boruvka's approach include Chung and Condon [8], Wang and Gu [9] and Dehne and Götz [10].

Parallelization of Prim's algorithm has been presented by Deo and Yoo [11]. Their algorithm targets shared-memory computers. Improved version of Prim's algorithm has been presented by Gonina and Kale [12]. Their algorithm adds multiple vertices per iteration, thus achieving significant speedups. Another approach targeting shared-memory computers presented by Setia et al. [13] uses the cut property of a graph to grow multiple trees in parallel. Hybrid approach, combining both Boruvka's and Prim's approaches has been developed by Bader and Cong [14].

Examples of parallel implementation of Kruskal's algorithm can be found in work of Jin and Baker [15], and Osipov et al [16]. Osipov et al. proposes a modification to Kruskal's algorithm to avoid edges which certainly are not in a graph. Their algorithm runs in near linear time if graph is not too sparse.

Bulk of the research into parallel MST algorithms has targeted shared-memory computers like PRAM, i.e. computers where entire graph can fit into memory. Our algorithms target distributed-memory computers and use partitioning scheme to divide the input graph evenly among processors. Because no process contains info about partition of other processes, we designed our algorithms to use predictable communication patterns, and not depend on the properties of input graph.

978-1-4673-5206-2/12/$31.00 ©2012 IEEE

III. The Algorithms

In the remainder of this paper, we will assume that graph $G = (V, E)$ is connected and undirected. Without loss of generality, it can be assumed that each weight is distinct, thus G is guaranteed to have only one MST. This assumption simplifies implementation, otherwise a numbering scheme can be applied to edges with same weight, at the cost of additional implementation complexity.

Let n be the number of vertices, m the number of edges ($|V| = n$, $|E| = m$), and p the number of processes involved in computation of MST. Let $w(v, u)$ denote weight of edge connecting vertices v and u. Input graph G is represented as $n \times n$ adjacency matrix $A = (a_{i,j})$ defined as:

$$a_{i,j} = \begin{cases} w(v_i, v_j) & \text{if } (v_i, v_j) \in E \\ 0 & otherwise \end{cases} \quad (1)$$

A. Prim's Algorithm

Prim's algorithm starts from an arbitrary vertex and then grows the MST by choosing a new vertex and adding it to MST in each iteration. Vertex with an edge with lightest weight incident on the vertices already in MST is added in every iteration. The algorithm continues until all the vertices have been added to the MST. This algorithm requires $O(n^2)$ time. Implementations of Prim's algorithm commonly use auxiliary array d of length n to store distances (weight) from each vertex to MST. In every iteration a lightest weight edge in d is added to MST and d is updated to reflect changes.

Parallelizing the main loop of Prim's algorithm is difficult [17], since after adding a vertex to MST lightest edges incident on MST change. Only two steps can be parallelized: selection of the minimum-weight edge connecting a vertex not in MST to a vertex in MST, and updating array d after a vertex is added to MST. Thus, parallelization can be achieved in the following way:

1) Partition the input set V into p subsets, such that each subset contains n/p consecutive vertices and their edges, and assign each process a different subset. Each process also contains part of array d for vertices in it's partition. Let V_i be the subset assigned to process p_i, and d_i part of array d which p_i maintains. Partitioning of adjacency matrix is illustrated in Fig. 1.
2) Every process p_i finds minimum-weight edge e_i (*candidate*) connecting MST with a vertex in V_i.
3) Every process p_i sends its e_i edge to leader process using all-to-one reduction.
4) From the received edges, leader process selects one with a minimum weight (called global minimum-weight edge e_{min}), adds it to MST and broadcasts it to all other processes.
5) Processes mark vertices connected by e_{min} as belonging to MST and update their part of array d.
6) Repeat steps 2-5 until every vertex is in MST.

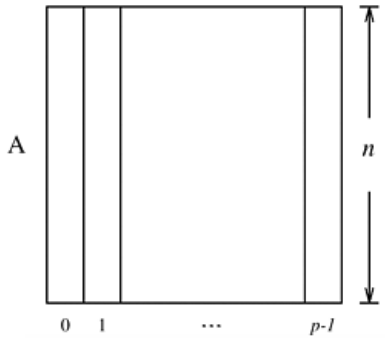

Fig. 1. Partitioning of adjacency matrix among p processes

Finding a minimum-weight edge and updating of d_i during each iteration costs $O(n/p)$. Each step also adds a communication cost of all-to-one reduction and all-to-one broadcast. These operations complete in $O(\log p)$. Combined, cost of one iteration is $O(n/p + \log p)$. Since there are n iterations, total parallel time this algorithm runs in is:

$$T_p = O\left(\frac{n^2}{p}\right) + O\left(n \log p\right) \quad (2)$$

In comparison to sequential algorithm, this algorithm achieves a speedup and efficiency of:

$$S = \frac{O(n^2)}{O(n^2/p + n \log p)} \quad (3)$$

$$E = \frac{1}{1 + O((p \log p)/n)} \quad (4)$$

From equations 3 and 4 we conclude that this formulation of Prim's algorithm is efficient only for $p = O(n/\log n)$ processes.

Prim's algorithm is better suited for dense graphs and works best for complete graphs. This also applies to it's parallel formulation presented here. Ineffectiveness of the algorithm on sparse graphs stems from the fact that Prim's algorithm runs in $O(n^2)$, regardless of the number of edges. A well-known modification [18] of Prim's algorithm is to use binary heap data structure and adjacency list representation of a graph to reduce the run time to $O(m \log n)$. Furthermore, using Fibonacci heap asymptotic running time of Prim's algorithm can be improved to $O(m + n \log n)$. Since we use adjacency matrix representation, investigating alternative approaches for Prim's algorithm was out of the scope of this paper.

B. Kruskal's Algorithm

Unlike Prim's algorithm which grows a single tree, Kruskal's algorithm grows multiple trees in parallel. Algorithm first creates a forest F, where each vertex in the graph is a separate tree. Next step is to sort all edges in E based on their weight. Algorithm then loops the sorted set and chooses minimum-weight edge e_{min} (i.e. first edge in sorted set). If e_{min} connects two different

trees in F, add it to the forest and combine two trees into a single tree, otherwise discard e_{min}. Algorithm loops until either all edges have been selected, or F contains only one tree, which is the MST of G. This algorithm is commonly implemented using Union-Find algorithm [19]. *Find* operation is used to determine which tree a particular vertex is in, while *Union* operation is used to merge two trees. Kruskal's algorithm runs in $O(m \log n)$ time, but can be made even more efficient by using more sophisticated Union-Find data structure, which uses *union by rank* and *path compression* [20]. If the edges are already sorted, using improved Union-Find data structure Kruskal's algorithm runs in $O(m\alpha(n))$, where $\alpha(n)$ is the inverse of an Ackerman function.

Our parallel implementation of Kruskal's algorithm uses the same partitioning scheme of adjacency matrix as in Prim's approach and is thus bounded by $O(n^2)$ time to find all edges in matrix. Every process first sorts edges contained in its partition. From edges in partition V_i, every process p_i finds a local minimum spanning tree (or forest, MSF) T_i using Kruskal's algorithm. At the end of this step, local MSTs are merged. Merging is performed in the following manner. Let a and b denote two processes which are to merge their local trees (or forests), and let A and B denote their respective set of local MST edges. Process a sends set A to b, which forms a new local MST (or MSF) from $A \cup B$. After merging, process a is no longer involved in computation and can terminate. Merging continues until only one process remains, which will contain MST of G.

Example of parallel Kruskal's algorithm is illustrated in Fig. 2. Input graph in *(a)* is divided among processes p_1 and p_2 which compute local MST based on edges incident on vertices assigned to them (*(b)* and *(c)*). Next, processes merge their local MST-s to form a MST of input graph. The dashed lines represent edges which are in local MST of a process, but are removed after merging.

Creating a new local MSF during merge step can be performed in a number of different ways. One approach is to perform Kruskal's algorithm again on $A \cup B$. Alternatively, a modified *depth-first search* (DFS) can be used. For every edge in A, it is first determined if it is already in the same tree of B (using *find* operation). If it is not, it is added in MSF and *union* operation is called. Merging two trees can produce a cycle, so a modified DFS is run to eliminate edge with a heaviest weight.

Computing the local MST takes $O(n^2/p)$. There is a total of $\log p$ merging stages, each costing $O(n^2 \log p)$. During one merge step one process transmits maximum of $O(n)$ edges for a total parallel time of:

$$T_p = O(n^2/p) + O(n^2 \log p) \tag{5}$$

Based on speedup and efficiency metrics, it can be shown that this parallel formulation is efficient for $p = O(n/\log n)$, same as first algorithm.

IV. Implementation

Algorithms were simple to implement using ANSI C and MPI. Simplicity is the result of fixed communication

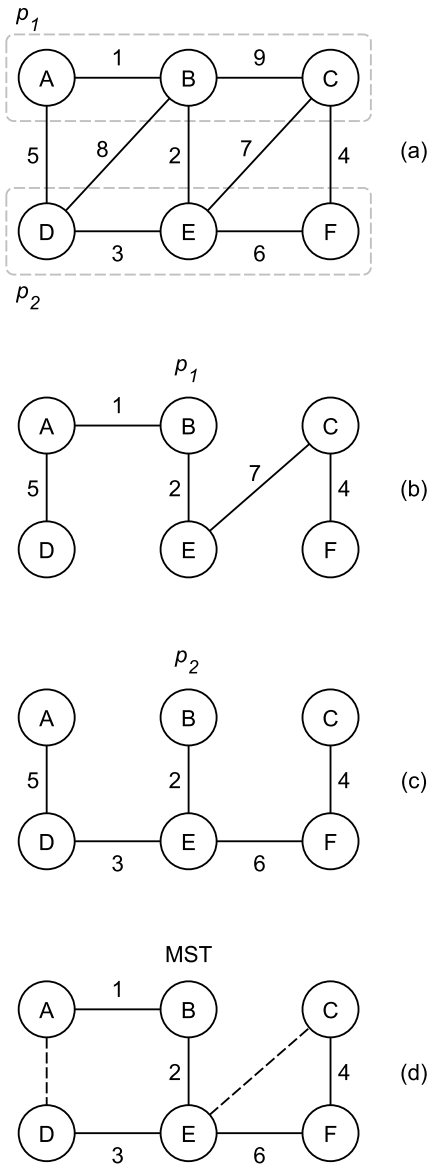

Fig. 2. Example of merge step for two processes

patterns which directly map to MPI operations. During implementation we explored alternative communication patterns in order to grow multiple trees in parallel, similar to approaches of parallelization of Boruvka's algorithm. We have found that using data-dependant communication paths results in imbalanced computation due to arbitrary communication between processes. Also, implementation of arbitrary communication can be difficult with MPI, since number of messages each process sends or receives is not known in advance for every input. Overcoming this obstacle often requires adding additional communication complexity, at the cost of overall performance.

Communication pattern in Prim's algorithm can be improved by using MPI *MPI_Allreduce* operation instead of the standard combination of *MPI_Reduce* and *MPI_Bcast*. This optimization does not necessarily result in better performance, since MPI implementations can implement

MPI_Allreduce operation as a simple all-to-one reduce, followed by a broadcast, without any performance improvements [21].

Main performance bottleneck of Prim's algorithm is communication overhead of all-to-one reduce operation. Reduce operation is costly in comparison to local computation, and all other processes are idle while waiting for reduce to complete. This prevents Prim's algorithm to achieve significant speedups on a larger number of processors. Therefore, Prim's algorithm is best used on a smallest number of processes on which partitioned input graph can fit.

Unlike Prim's algorithm, Kruskal's algorithm doesn't use collective communication operations during which all processes except one are idle. Performance-wise, critical part Kruskal's algorithm is merging of local MST-s. Merge part of Kruskal's algorithm is only fully efficient in case when p is a power of 2. Since merging is pairwise operation, in other cases at least one merge step will have a process without a pair. This process will be idle until a merge partner is available. In our implementation, idling can span multiple merge steps, thus causing a considerable efficiency degradation. For example, if there are 9 processes in computation, one process will be idle until the very last merge step. One approach to solving this issue would be introduction of a special 3-way merge (or in general a d-ary merge, where $d = 2, 3, 4 \ldots$) along with a load balancing logic to minimize or remove the effect on performance of algorithm.

V. Conclusion

We presented two parallel implementations of algorithms for finding minimum spanning tree of a graph. Our algorithms are parallelizations of classical sequential algorithms of Prim and Kruskal. Parallel processes work on a subset of input graph, and communicate using fixed communication pattern. First algorithm takes $O(n^2/p + n \log p)$ time, while second takes $O(n^2/p + n^2 \log p)$ time.

Our analysis has identified several bottlenecks in our implementations, and further work in this area would include minimizing communication cost by reducing the number and size of messages passed, as well as improving merge step of the second algorithm. Also, further experimental work would give us information about practical limitations of our algorithms for wider array of input graphs and uncover new areas for improvement.

Acknowledgment

Authors are partially supported by Ministry of Education and Science of the Republic of Serbia, through project no. III47003: " Infrastructure for technology enhanced learning in Serbia".

References

[1] R. L. Graham and P. Hell, "On the history of the minimum spanning tree problem," *IEEE Ann. Hist. Comput.*, vol. 7, no. 1, pp. 43–57, Jan. 1985. [Online]. Available: http://dx.doi.org/10.1109/MAHC.1985.10011

[2] O. Boruvka, " O Jistém Problému Minimálním (About a Certain Minimal Problem) (in Czech, German summary)," *Práce Mor. Prírodoved. Spol. v Brne III*, vol. 3, 1926.

[3] R. C. Prim, "Shortest connection networks and some generalizations," *Bell System Technology Journal*, vol. 36, pp. 1389–1401, 1957.

[4] J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, Feb. 1956. [Online]. Available: http://www.jstor.org/stable/2033241

[5] R. G. Gallager, P. A. Humblet, and P. M. Spira, "A distributed algorithm for minimum-weight spanning trees," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 1, pp. 66–77, Jan. 1983. [Online]. Available: http://doi.acm.org/10.1145/357195.357200

[6] I. Katriel, P. Sanders, J. L. Träff, and J. L. Tra, "A practical minimum spanning tree algorithm using the cycle property," in *In 11th European Symposium on Algorithms (ESA), number 2832 in LNCS*. Springer, 2003, pp. 679–690.

[7] H. Ahrabian and A. Nowzari-Dalini, "Parallel algorithms for minimum spanning tree problem," *International Journal of Computer Mathematics*, vol. 79, no. 4, pp. 441–448, 2002.

[8] S. Chung and A. Condon, "Parallel implementation of borvka's minimum spanning tree algorithm," in *Proceedings of the 10th International Parallel Processing Symposium*, ser. IPPS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 302–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=645606.661036

[9] W. Guang-rong and G. Nai-jie, "An efficient parallel minimum spanning tree algorithm on message passing parallel machine," *Journal of Software*, vol. 11, no. 7, pp. 889–898, 2000.

[10] F. Dehne and S. Götz, "Practical parallel algorithms for minimum spanning trees," in *In Workshop on Advances in Parallel and Distributed Systems*, 1998, pp. 366–371.

[11] Y. Y. B. Deo, Narsingh, "Parallel algorithms for the minimum spanning tree problem." in *Proceedings of the International Conference on Parallel Processing*, 1981, pp. 188–189.

[12] E. Gonina and L. V. Kale, "Parallel prim's algorithm on dense graphs with a novel extension," *PPL Technical Report*, October 2007.

[13] A. N. R. Setia and S. Balachandran, "A new parallel algorithm for minimum spanning tree problem," in *Proc.International Conference on High Performance Computing (HiPC)*, 2009, pp. 1–5.

[14] D. A. Bader and G. Cong, "Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs," *J. Parallel Distrib. Comput.*, vol. 66, no. 11, pp. 1366–1378, Nov. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2006.06.001

[15] M. Jin and J. W. Baker, "Two graph algorithms on an associative computing model," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2007, Las Vegas, Nevada, USA, June 25-28, 2007, Volume 1*, 2007, pp. 271–277.

[16] V. Osipov, P. Sanders, and J. Singler, "The filter-kruskal minimum spanning tree algorithm," in *ALENEX'09*, 2009, pp. 52–61.

[17] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing (2nd Edition)*, 2nd ed. Addison Wesley, Jan. 2003. [Online]. Available: http://www.worldcat.org/isbn/0201648652

[18] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[19] D.-Z. Pan, Z.-B. Liu, X.-F. Ding, and Q. Zheng, "The application of union-find sets in kruskal algorithm," in *Proceedings of the 2009 International Conference on Artificial Intelligence and Computational Intelligence - Volume 02*, ser. AICI '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 159–162. [Online]. Available: http://dx.doi.org/10.1109/AICI.2009.155

[20] Z. Galil and G. F. Italiano, "Data structures and algorithms for disjoint set union problems," *ACM Comput. Surv.*, vol. 23, no. 3, pp. 319–344, Sep. 1991. [Online]. Available: http://doi.acm.org/10.1145/116873.116878

[21] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *J. Parallel Distrib. Comput.*, vol. 69, no. 2, pp. 117–124, Feb. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2008.09.002