COMPUTING THE CLOSENESS CENTRALITY OF EVOLVING NETWORKS ON CLUSTERS

Ahmet Erdem Sarıyüce[1,2], *Erik Saule*[4], *Kamer Kaya*[1], *Ümit V. Çatalyürek*[1,3]
Depts. [1]*Biomedical Informatics,* [2]*Computer Science and Engineering,* [3]*Electrical and Computer Engineering*
The Ohio State University
[4]*Dept. Computer Science, University of North Carolina at Charlotte*
Email:sariyuce.1@osu.edu, esaule@uncc.edu, kaya.20@osu.edu, umit@bmi.osu.edu

SIAM Workshop on Network Science 2014
July 6-7 · Chicago

Abstract

Networks are commonly used to model the traffic patterns, social interactions, or web pages. Closeness centrality (CC) is a global metric that quantifies how important is a given node in the network. When the network is dynamic and keeps changing, the relative importance of the nodes also changes. The best known algorithm to compute the CC scores makes it impractical to recompute them from scratch after each modification. In this paper, we propose Streamer, a distributed memory framework for incrementally maintaining the closeness centrality scores of a network upon changes. It leverages pipelined and replicated parallelism and takes NUMA effects into account. It speeds up the maintenance of the CC of a real graph with 916K vertices and 4.3M edges by a factor of 497 using a 64 nodes cluster.

Introduction

Many of today's networks are dynamic. And for such networks, maintaining the exact centrality scores is a challenging problem which has been studied in the literature [3, 4, 5]. Offline CC computation can be expensive for large-scale networks. Yet, one could hope that the incremental graph modifications can be handled in an inexpensive way. In a previous study, we proposed a sequential incremental closeness centrality algorithm which is orders of magnitude faster than the best offline algorithm [5]. Still, the algorithm was not fast enough to be used in practice. In this work [6], we present Streamer, a framework to efficiently parallelize the incremental CC computation on high-performance clusters.

Streamer (Figure 1) employs *DataCutter* [1], our inhouse data-flow programming framework for distributed memory systems. In DataCutter, the computations are carried by independent computing elements, called *filters*, that have different responsibilities and operate on data passing through them.

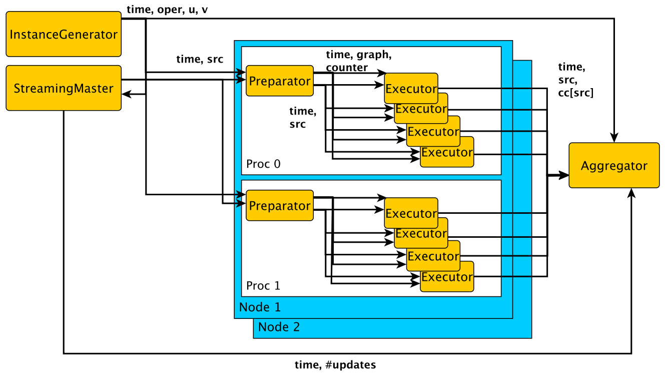
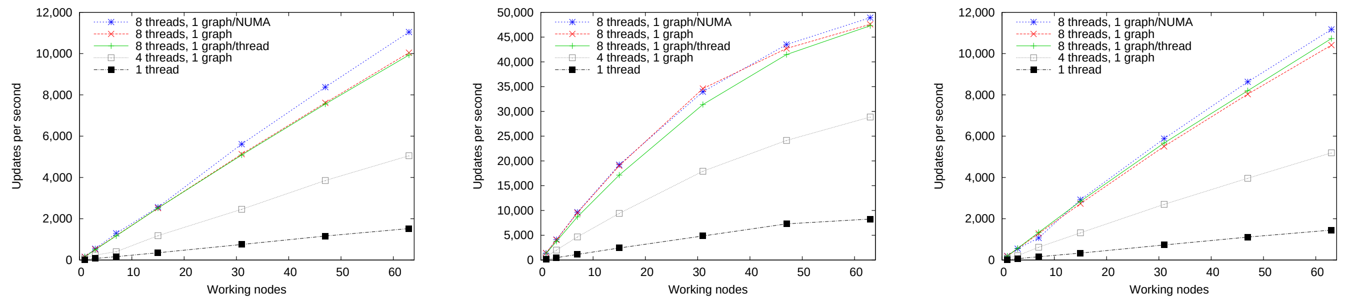


Figure 1: Architecture of Streamer taking into account the hierarchical nature of the nodes.

The best available algorithm for the offline centrality computation is pleasingly parallel (and scalable if enough memory is available) since it involves n independent executions of the single-source shortest path (SSSP) algorithm [2]. In a naive distributed framework for the offline case, one can distribute the SSSPs to the nodes and gather their results. Here the computation is static, i.e., when the graph changes, the previous results are ignored and the same n SSSPs are re-executed. On the other hand, in the online approach, the updates can arrive at any time even while the centrality scores for a previous update are still being computed. Furthermore, the scores which need to be recomputed (the SSSPs that need to be executed) change w.r.t. the update. Finding these SSSPs and distributing them to the nodes is not a straightforward task. To be able to do that, the incremental algorithms maintain complex information such as the biconnected component decomposition of the current graph [5]. Hence, after each edge insertion/deletion, this information needs to be updated. There are several (synchronous and asynchronous) blocks in the online approach. And it is not trivial to obtain an



(a) `amazon0601` (b) `web-NotreDame` (c) `web-Google`

Figure 2: Scalability: the performance is expressed in the number of updates per second. Different worker-node configurations are shown. The number of threads indicated the number of *Executor* per node, while the number of graphs indicates the number of *Preparator* per node.“1 graph/NUMA” means that one per processor.

efficient parallelization of the incremental algorithm.

Streamer

The structure of STREAMER is presented in Figure 1. The *InstanceGenerator* generates the graph at the beginning of the application and send the modifications on the graph to all the filters. *StreamingMaster* is responsible for the deciding which vertices of the graph need their centrality updated and output a list to the third filter and statistic information to the fourth one. The *Preparator* and *Executor* pair performs the real work and computes the new CC scores after each graph modification. The *Aggregator* reconstructs the exact closeness centrality values for all vertices at each modification of the graph.

The computation is organized so that multiple *Executor* are managed by a single *Preparator*. The *Preparator* is repsonsible for managing the graph data structures which is passed to *Executor* using memory references, leverage the fact that they are running on the same computing node. That way, the four *Executor* on the same processor share the same graph data structure and benefit from cache reutilization.

Performance results

Figure 2 shows the performance and scalability of the system in different configurations on a cluster of 64 nodes equipped with two quad-core Intel processors. The performance is expressed in number of updates per second. The framework obtains up to 11, 000 updates/sec on `amazon0601` and `web-Google`, and 49, 000 updates/sec on `web-NotreDame`. It appears to scale linearly on the graphs `amazon0601` and `web-Google`. For the first two graphs, it reaches a speedup of 456 and 497, respectively, with 63 nodes and 8 threads/node compared to the single node-single thread configuration. The last graph, `web-NotreDame`, does not exhibit a linear scaling and obtains a speedup of only 316.

Conclusion

Maintaining the correctness of a graph analysis is important in today’s dynamic networks. Computing the closeness centrality scores from scratch after each graph modification is prohibitive, and even sequential incremental algorithms are too expensive for networks of practical relevance. In this paper, we proposed STREAMER, a distributed memory framework which guarantees the correctness of the CC scores, exploits replicated and pipelined parallelism, and takes the hierarchical architecture of modern clusters into account. Using STREAMER on a 64 nodes, 8 cores/node cluster, we reached a speedup of 497.

References

[1] M. D. Beynon, T. Kurç, Ü. V. Çatalyürek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, Oct. 2001.

[2] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[3] O. Green, R. McColl, and D. A. Bader. A fast algorithm for streaming betweenness centrality. In *Proc. of SocialCom*, 2012.

[4] M.-J. Lee, J. Lee, J. Y. Park, R. H. Choi, and C.-W. Chung. QUBE: a Quick algorithm for Updating BEtweenness centrality. In *Proc. of WWW*, 2012.

[5] A. E. Sarıyüce, K. Kaya, E. Saule, and Ü. V. Çatalyürek. Incremental algorithms for network management and analysis based on closeness centrality. *CoRR*, abs/1303.0422, 2013.

[6] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. STREAMER: a distributed framework for incremental closeness centrality computation. In *Proc. of IEEE Cluster 2013*, Sep 2013.