Performance Analysis of Single-source Shortest Path Algorithms on Distributed-memory Systems

Thap Panitanarak Kamesh Madduri
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA, USA
{txp214, madduri}@cse.psu.edu

1 Introduction

We consider the problem of single-source shortest path (SSSP) computation in a distributed setting, where a large sparse graph with non-negative edge weights is partitioned across the nodes of a parallel system. SSSP is a key computation arising in large-scale network analysis and a possible candidate for inclusion in the Graph500 benchmark. *Work-efficient* SSSP algorithms are based on Dijkstra's algorithm [5]. However, there is limited concurrency to exploit in Dijkstra's algorithm, as it belongs to the *label-setting* class of shortest path algorithms. *Label-correcting* algorithms relax the constraint that the vertex with minimum weight be picked at each step, and these algorithms do not use a priority queue. The Bellman-Ford algorithm [1] is an example of a label-correcting and is more amenable to parallelization, but it is not work-optimal for graphs with non-negative edge weights. The Delta-stepping [11] algorithm can be considered a hybrid approach combining Bellman-Ford and Dijkstra's algorithm. Prior work has studied shared-memory [10] and distributed-memory [6, 7] implementations of Delta-stepping. There is one key parameter in this algorithm, Δ, whose setting determines the number of parallel phases, the work performed, the load-balance across multiple tasks in a parallel system, and hence the running time.

2 Our Contributions

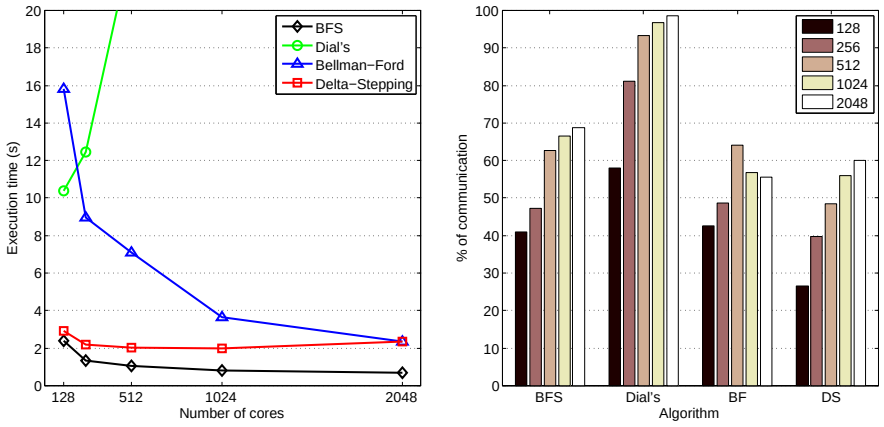
We have developed optimized parallel implementations of three SSSP algorithms – Dial's algorithm [4], Bellman-Ford, and Delta-stepping – for graphs with positive integer edge weights. Dial's algorithm is a special case of Dijkstra's algorithm for graphs with integer weights, and does not require a priority queue. Bellman-Ford also does not use a priority queue because it is a label-correcting algorithm. We use a bucket array data structure, parameterized by Δ, as the priority queue in our implementation of Delta-stepping. All three approaches are bulk-synchronous. We use a distributed compressed sparse row representation for the graph. On a system with p tasks, each task holds n/p vertices and all outgoing edges from these vertices. The distance array is also partitioned and distributed in a similar manner. Delta-stepping has light and heavy edge relaxation phases. We have designed our method so that all communication happens at the end of these phases, and all relaxations are performed locally. Our approach is thus a bulk-synchronous modification of the implementation in [10]. Further, our code has the following optimizations:

- The buckets are implemented as dynamic arrays. Each bucket is allocated only if a vertex being added to it, and it can be resized when needed. At the end of a light edge relaxation phase of bucket i, it can be deallocated and the memory can be reused for other bucket allocations, since there will be no more insertions to that bucket. Moreover, we use two auxiliary arrays of size n/p to provide constant time insertions and deletions of vertices in each bucket [2].
- A semi-sorting routine is used as a preprocessing step of the algorithm. It reorders the edges of each vertex based on edge weights and the value of Δ such that all light edges appear before heavy ones.
- Local lookup arrays are used to track the tentative distance of every vertex, avoiding duplicate requests being sent.

Additional details are discussed in a technical report [13].

We collect parallel performance results on the TACC Stampede cluster. This is a 10 PetaFlop/s Dell Linux cluster with more than 6400 Dell PowerEdge server nodes. Each node has two Intel Xeon E5 Sandy Bridge processors and an Intel Xeon Phi coprocessor. We do not use the coprocessor in the current study. Our implementation uses MPI. The main collective communication routines used in our code are Alltoallv, Alltoall, and Allreduce. We use synthetic graphs generated using the Graph500 reference implementation v1.2 [8]. The generated graphs have skewed degree distributions and a very low graph diameter. We experiment with uniform and normal edge weight distributions, as well as integer (includes Dial) and double-precision (weights in $(0, 1]$) edge weights. We also vary the maximum weight in case of graphs with integer weights.

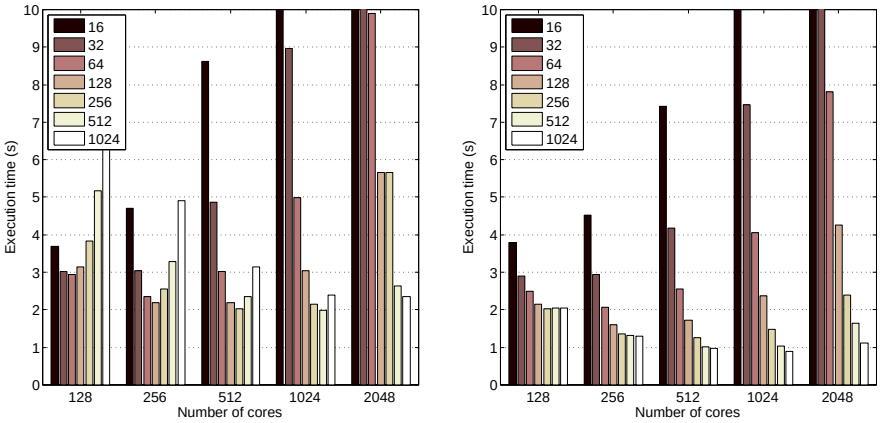
Our single-node Delta-stepping performance (16 MPI tasks) is faster than the shared-memory Chaotic relaxation approach of Galois [12, 9] on a large collection of graphs. It is 1.4× faster with uniform edges and 3.36× faster with normal weight distributions. It is also nearly 10× faster than the Parallel Boost graph library implementation on a single node.



(a) Execution time (b) Communication time

Figure 1: **Strong scaling:** Overall execution time and percentage of time spent in communication for Dial's algorithm, Bellman-Ford, and Delta-stepping, on SCALE 27 Graph500 graphs with uniform weights and max weight set to 1024.

Multinode strong scaling results are shown in Figure 1. We also report performance of a parallel Breadth-First Search (BFS) for comparison. Delta-stepping scales up to 512 cores (32 nodes), but beyond that point, there is computational load imbalance. Bellman-Ford is initially slower than Delta-stepping, but scales better due to fewer number of parallel phases. Dial's algorithm shows the worst scaling, but note that its performance is dependent on the maximum edge weight. Dial performance is comparable to BFS if all weights are set to 1. Delta-stepping would incur a significant overhead due to bucket maintenance in that setting.



(a) Uniform weight distribution (b) Normal weight distribution

Figure 2: The effect of Δ on a SCALE 27 Graph500 network.

Note from Figure 2 that the performance of Delta-stepping is heavily dependent on the value of Δ used in the case of uniformly-distributed edge weights. In general, larger values of Δ are preferable at higher task concurrencies.

To alleviate the load imbalance seen at 512 cores, we are also developing a parallel implementation of Delta-stepping based on a 2D edge-based partitioning and distribution of the graph. Our 2D approach supports arbitrary partitions of the graph (e.g., with 512 MPI tasks, 32×16, 64×8, 128×4, etc. are possible, and the 512×1 case corresponds to 1D row-based partitioning). The MPI collective Allgatherv is additionally used in the 2D implementation. While we verified that the 2D implementation improves load balance, our current 2D implementation is still slower than the best 1D approach (after tuning for Δ) due to bucketing implementation overhead.

In a recent paper [3], Chakaravarthy et al. present a new parallel algorithm for SSSP, and a highly optimized implementation for IBM Blue Gene/Q systems. Assuming a 1D graph distribution, this Delta-stepping based algorithm introduces a new edge pruning strategy to reduce inter-node communication, and a new load-balancing strategy for graphs with skewed degree distributions. We are extending our Delta-stepping implementation to include both these optimization strategies.

In current work, we are developing a hybrid MPI-OpenMP implementation to mitigate the load imbalance issue seen in strong scaling. We are also developing a automated scheme to choose the value of Δ given the task concurrency and the weight distribution. Thirdly, we are exploring hybrid Delta-stepping and Bellman-Ford approaches, where we switch to Bellman-Ford after a certain number of Delta-stepping phases.

Acknowledgments

This work is supported by NSF grant ACI-1253881 and used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by NSF grant OCI-1053575.

References

[1] R. Bellman. On a routing problem. *Quart. Appl. Math.*, 16:87–90, 1958.

[2] P. Briggs and L. Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1-4):59–69, 1993.

[3] V. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal. Scalable single source shortest path algorithms for massively parallel systems. In *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2014.

[4] R. Dial. Algorithm 360: Shortest path forest with topological ordering. *Commun. ACM*, 12:632–633, 1969.

[5] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[6] N. Edmonds, T. Hoefler, and A. Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *Proc. ACM Int'l. Conf. on High Performance Computing (HiPC)*, 2010.

[7] N. Edmonds, J. Willcock, and A. Lumsdaine. Expressing graph algorithms using generalized active messages. In *Proc. ACM Int'l. Conf. on Supercomputing (ICS)*, 2013.

[8] Graph500 benchmark. `http://www.graph500.org/`, last accessed Feb 2014.

[9] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proc. ACM SIGPLAN Conf. on Programming language design and implementation (PLDI)*, 2007.

[10] K. Madduri, D. A. Bader, J. Berry, and J. R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Proc. Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[11] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *J. Algs.*, 49(1):114–152, 2003.

[12] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proc. ACM Symp. on Operating Systems Principles (SOSP)*, 2013.

[13] T. Panitanarak and K. Madduri. Performance analysis of single-source shortest path algorithms on distributed-memory systems. Technical Report CSE #14-003, The Pennsylvania State University, 2014.