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

Thap Panitanaraka Kamesh Madduri

madduri@cse.psu.edu

Computer Science and Engineering
The Pennsylvania State University

SIAM CSC14
July 23, 2014
Slides, code, paper at graphanalysis.info
Talk Outline

• Introduction and current state-of-the-art
• Applications of SSSP
• Recommendations on parallel graph algorithm design and implementation
• Our implementations of SSSP algorithms, performance analysis
SSSP (with non-negative edge weights)

- **Input**: Graph with non-negative edge weights, source vertex \( s \).
- **Output**: For every vertex \( v \), shortest path length from \( s \) to \( v \).

![Graph with non-negative edge weights]

| \( d \) | 0 | 5 | 3 | 2 | 3 | 1 | 4 | 1 | 3 | 6 |
SSSP (with non-negative edge weights)

- **Input**: Graph with non-negative edge weights, source vertex s.
- **Output**: For every vertex v, shortest path length from s to v.

- Iteratively update vertex *label* (tentative distance value)
- **Priority queue** operations: \( m \) decrease key ops, \( n \) extract min
  ops (\( m = |E|, n = |V| \))
Sequential SSSP Algorithms (theory)

• Label-setting
  – “Work-efficient”, $n$ extract min ops
  – e.g., Dijkstra’s algorithm with Fibonacci heap

• Label-correcting
  – Relax Dijkstra assumption about $n$ extract min ops
  – e.g., Bellman-Ford algorithm ($O(mn)$ worst-case, but works for graphs with negative edge weights)

• Special cases
  – Undirected graphs, Integer weights, Bounded weights, Weight distributions, Planar graphs
Sequential SSSP Algorithms (practice)

- Optimizations to improve cache efficiency of priority queue operations
- Preprocessing to simplify query time: reach, transit nodes, highway hierarchies

- BFS performance is a good lower bound
  - SSSP performance within 2X of BFS time is great
- A fast general-purpose SSSP solver: MLB priority queue-based (Andrew Goldberg’s network opt package; DIMACS Challenge 9 reference)
- Recent evaluation of priority queues [Larkin et al., arXiv:1403.0252, 2014]
Parallel SSSP Algorithms (theory)

• PRAM algorithms: there’s no known logarithmic time, work-efficient algorithm
• Dijkstra-based
  – Parallelize priority queue operations
• Bellman Ford-based
  – Asymptotically not work-efficient, easy to implement
• Hybrid
  – “For random directed graphs with edge probability $d/n$ and uniformly distributed edge weights a PRAM version works in expected time $O(\log^3 n / \log \log n)$ using linear work”
• Special cases
SSSP Parallel Implementations in practice: 1/2

• No general-purpose parallel solver yet, almost all prior work is for special cases.

• DIMACS Shortest Paths Challenge 9, 2006-07
  – Madduri et al., Parallel Delta-stepping for the Cray MTA-2
  – Crobak et al., Parallel Thorup’s alg. for the Cray MTA-2
  – Edmonds et al., Parallel label-setting approaches for distributed-memory systems using Parallel Boost Graph Library.
  – Efficient only for low diameter graphs

• Malewicz et al. (Pregel paper), Proc. SIGMOD 2010 (Bellman Ford-like).
SSSP Parallel Implementations in practice: 2/2

- PHAST, Delling et al., Proc. IPDPS 2011 (shared mem, preprocessing-based, works for low highway dimension graphs)
- Galois and Ligra SSSP implementations (x86 multicore)
- Chakaravarthy et al., Proc. IPDPS 2014 (label correcting alg. based on Delta-stepping, highly tuned for Graph500 graphs and IBM Blue Gene/Q supercomputers)
- Waterman et al., Proc. IPDPS 2014 (tuned CUDA implementation of Delta stepping-like alg.)
Prior SSSP performance results

- Chakaravarthy et al. IPDPS ’14, 4.39 Trillion-edge Graph500 graph, 32,768 BG/Q nodes, 1.47 seconds.
- Waterman et al. IPDPS ’14, 18M vertex, 90M edge R-MAT graph, NVIDIA GTX 680, 3.9 seconds.
- Pregel SIGMOD ’10, 1B vertex binary tree, 800 worker CPUs, 20 seconds.
- PHAST, IPDPS ’11, USA road network, SSSP in few ms.

- Apples-to-oranges comparison?
- What algorithm to use in my fancy graph analytics engine?
- What optimizations are good to try, what aren’t?
Talk Outline

• Introduction and current state-of-the-art
• Applications of SSSP
• Recommendations on parallel graph algorithm design and implementation
• Our implementations of SSSP algorithms, performance analysis
Applications: Social network analysis

• Edge weights used in community identification and centrality measures
• Digression: Live network analysis demo, using SIAM CSC14 participant list!
• How to automatically determine strength of ties? My solution is to rely on web search.

Do you mean "kamesh madhuri"+"bora ucar"?
Constructing a SIAM CSC14 network
CSC14 participant network
Centrality-sorted radial layout
Network statistics

- ~ 70% (40/57) in giant component
- Weight distribution?
- Degree and clustering coefficient distributions?
Other analytics and applications that use SSSP

• Graph algorithms for problems in Systems Biology
• Parallel discrete event simulations
• Computations that use SSSP as subroutine
  – Path-limited searches
  – Finding k shortest simple paths
  – All-pairs shortest paths
• Remember that SSSP may only be an isolated routine in a large data processing pipeline

• SSSP is proposed Graph500 benchmark #2
Talk Outline

• Introduction and current state-of-the-art
• Applications of SSSP
• Recommendations on parallel graph algorithm design and implementation
• Our implementations of SSSP algorithms, performance analysis
Q. Is my parallel implementation efficient, fast, and scalable?

- **Efficiency**: Augment conventional notion of work efficiency with hw/sw-based performance analysis
  - RAM and PRAM models are too coarse-grained, problem-specific measures are better.
  - Too much emphasis on speedup plots and scaling with # of cores. Instead, look at percentage of machine peak achieved.

- **Speed**: Nearly all massive graph computations are
  - Linear time or sublinear time
  - Dominated by data movement costs
  Keep data as close as possible to cores. Avoid extraneous communication.
  A fast algorithm achieves a high fraction of peak bandwidth.
TACC Stampede system, observed aggregate data bandwidths

Benchmark

- AllGather
- AlltoAll
- RandomAccess
- StreamRead

Total Bandwidth (GB/s)

Number of Nodes

Tmem

Tcomm
Q. Is my graph algorithm efficient, fast, and scalable?

- **Efficiency, Speed:** minimize data movement, compare to system peaks and lower bounds
- **Scalability**
  - Try sufficiently-large test instances for graph problem and hardware configuration. For SSSP, at least
    - 100 M edges for single GPU/Xeon Phi
    - 1 B edges for an Intel dual-socket server (say, with 32 GB memory)
    - 32 B edge graph for 32 nodes of an Intel cluster
  - Results on smaller instances may not hold true for larger graphs
Talk Outline

• Introduction and current state-of-the-art
• Applications of SSSP
• Recommendations on parallel graph algorithm design and implementation
• Our implementations of SSSP algorithms, performance analysis
Our new SSSP implementations

- Delta-stepping
- Dial’s algorithm
  - Special case of Dijkstra’s algorithm, given integer weights
- Bellman Ford
  - Modified version, “Chaotic relaxation” variant

- Common graph representation, optimizations. Implementations differ in priority queue details.
Graph layout and distribution

• 1D-partitioned graph: each MPI task owns \( \frac{n}{p} \) vertices
• Distributed CSR-like representation, 32 bits per edge
• Undirected graphs: edges replicated
• Edge weights stored in separate (partitioned, distributed) array
• Owner updates and determines vertex label
Parallel Delta-Stepping pseudocode

\[ d(u) \leftarrow \infty; B \leftarrow \emptyset; S \leftarrow \emptyset; \]
\[ curr \leftarrow 0; Buffer \leftarrow \emptyset; \]
\[ \text{if} \ \text{owner}(s) = \text{rank} \ \text{then} \]
\[ d(s) = 0; \]
\[ \text{add} \ s \ \text{to} \ B[curr]; \]
\[ \text{while} \ B \neq \emptyset \ (\text{globally}) \ \text{do} \]
\[ \text{while} \ B[curr] \neq \emptyset \ (\text{globally}) \ \text{do} \]
\[ \text{for each} \ u \in B[curr] \ \text{do} \]
\[ \text{remove} \ u \ \text{from} \ B[curr]; \]
\[ \text{if} \ u \notin S \ \text{then} \]
\[ \text{add} \ u \ \text{to} \ S; \]
\[ \text{for each} \ (u, v) \in E_L \ \text{do} \]
\[ p_v = \text{owner}(v); \]
\[ dtv \leftarrow d(u) + w(u, v); \]
\[ \text{add} \ (v, dtv) \ \text{to} \ Buffer_{p_v}; \]
\[ \text{endfor} \]
\[ \text{endfor} \]
\[ \text{endwhile} \]
\[ \text{for each} \ u \in S \ \text{do} \]
\[ \text{for each} \ (u, v) \in E_H \ \text{do} \]
\[ p_v = \text{owner}(v); \]
\[ dtv \leftarrow d(u) + w(u, v); \]
\[ \text{add} \ (v, dtv) \ \text{to} \ Buffer_{p_v}; \]
\[ \text{endfor} \]
\[ \text{endfor} \]
\[ S \leftarrow \emptyset; \]
\[ \text{// Exchange requests} \]
\[ \text{// Relax} \]
\[ curr \leftarrow curr + 1; \]
\[ \text{endwhile} \]
\[ \text{// Create a list of requests} \]
\[ \text{Alltoallv(Buffer);} \]
\[ \text{// Exchange requests for local processing} \]
\[ E_L = \{(u,v) \in E | w(u,v) \leq \Delta \} \ -- \text{Light edges} \]
\[ E_H = \{(u,v) \in E | w(u,v) > \Delta \} \ -- \text{Heavy edges} \]
Modeling parallel execution time

- Time dominated by local memory references and inter-node communication
- Assuming perfectly balanced computation and communication, we have

\[ \beta_L \frac{m}{p} + \alpha_{L,n/p} \frac{n + m}{p} \]

Local memory references:

- Inverse local RAM bandwidth
- Local latency on working set \(|n/p|\)

\[ \beta_{N,a2a}(p) \frac{\text{edgecut}}{p} + \alpha_N p \]

Inter-node communication:

- All-to-all remote bandwidth with \(p\) participating processors
BFS with a 2D-partitioned graph

- Avoid expensive $p$-way All-to-all communication step
- Each process collectively ‘owns’ $n/p_r$ vertices
- Additional ‘Allgatherv’ communication step for MPI tasks in a row

Local memory references:

\[
\beta_L \frac{m}{p} + \alpha_{L,n/p_c} \frac{n}{p} + \alpha_{L,n/p_r} \frac{m}{p}
\]

Inter-node communication:

\[
\beta_{N,a2a}(p_r) \frac{\text{edgecut}}{p} + \alpha_N p_r + \\
\beta_{N,gather}(p_c) \left(1 - \frac{1}{p_r}\right) \frac{n}{p_c} + \alpha_N p_c
\]
Delta-stepping optimizations

• Local bucketing structure on each task
• Extremely lightweight bucket implementation
  – Don’t do exact bucketing
  – Bucket insertion, deletion, resizing overheads can be significant
  – Allows for adaptive Delta-stepping
• Fix number of delta-stepping phases beforehand
  – Switch to Bellman Ford-like approach after a while
• Lightweight preprocessing: heavy and light edge classification
Chakaravarthy et al. IPDPS ’14 algorithm

• Extremely efficient for Graph500 graphs
• With direction-optimizing search [Beamer et al., Proc. SC12], one can perform fewer edge relaxations than Dijkstra’s algorithm!
  – Effective speedup heuristic for low diameter graphs with high average degree
• Load balancing heuristic at high process concurrencies, with high-degree vertex splitting
• Multithreading within compute node
• Custom inter-node communication libraries
• Blue Gene/Q-specific tuning
Improving our implementations

- Optimize bucket data structure
- Direction-optimizing and load-balancing heuristics from Chakaravarthy et al. alg.
- Picking the right value of Delta
- Multithreading within node

- Additional tuning for real-world graphs
- Performance impact of partitioning and reordering
Experimental study

• Performance results and alg. comparisons on
  – NSF Stampede supercomputer (up to 32 nodes, didn’t use Xeon Phi coprocessor)
  – Penn State Cyberstar system (Intel Nehalem cluster)
• Graph500 and real-world graphs
• Integer weights
  – Vary max weight
• Double-precision weights
  – Delta stepping-only
• Uniform and normal weight distributions
Key Performance Observations

• Graph500 graph with $2^{27}$ vertices and $2^{31}$ edges: SSSP performance rate of $\sim 1$ GTEP/s on 16 nodes (256 cores and MPI tasks) of Stampede
  – With best value of Delta, integer weights, unif dist.
  – (Non-direction-optimizing MPI-only) BFS 2X faster

• Real-world graphs: SSSP performance varies between 0.5-3 GTEP/s on 16 nodes
  – Roughly comparable to a best-performing BFS approach on a single shared-memory node

• Communication time is 5-70% of overall running time.
Conclusions

• SSSP on large sparse low-diameter graphs
• Delta stepping-\textit{lite} seems to work best in practice
• Limit # of parallel phases by switching to Bellman Ford
• Lightweight implementation of vertex bucketing is important
• Future avenues for improvement: graph representation, load balancing on real-world graphs, optimal value of Delta ...
Acknowledgment of Support

- NSF ACI award #1253881
- NSF XSEDE allocation TG-CCR13006
Thank you!

- Questions?

  - madduri@cse.psu.edu
  - http://www.cse.psu.edu/~madduri