Performance analysis of Δ-stepping algorithm on CPU and GPU

Dmitry Lesnikov[1]
dslesnikov@gmail.com

Mikhail Chernoskutov[1,2]
mach@imm.uran.ru

1 – Ural Federal University (Yekaterinburg, Russia)
2 – Krasovskii Institute of Mathematics and Mechanics (Yekaterinburg, Russia)

Abstract

Δ-stepping is an algorithm for solving single source shortest path problem. It is very efficient on a large class of graphs, providing nearly linear time complexity in sequential implementation, and can be parallelized. However, this algorithm requires some tuning and has unpredictable performance behaviour on systems with different parallel architectures. In this paper we describe details of implementation of this algorithm on CPU and GPU. Performing numerous tests we studied scalability of algorithms on CPU and GPU and compared performance for different Δ parameter. We show that performance results for CPU and GPU varies for different delta. Also, we noticed that GPU performance are stable for some different delta values. Our result shows how to choose Δ for achieving best performance for tested graphs.

1 Introduction

The single source shortest path problem (SSSP) is well-studied typical graph problem that has a lot of applications, both theoretical and practical. A huge amount of algorithms and their optimisations was provided for solving this problem efficiently. However, an importance of providing fast scalable algorithm for SSSP has been only growing for the past decade, since quantity of data being represented in terms of graphs hit the point where we have to process that data in parallel using high-performance computational resources.

Δ-stepping algorithm [1], the algorithm for solving SSSP problem that is quite efficient and providing suitable level of parallelism was introduced and widely researched. It is quite convenient and fast, yet has an issue of unstudied performance corresponding to systems with different amount of available parallelism (CPUs and GPUs). In this paper results of Δ-stepping algorithm performance evaluation on CPU and GPU architectures are presented.

To formalize SSSP, consider directed graph $G = G(V, E)$ with set of vertices V and set of edges E, where $|V| = n$ and $|E| = m$. Let $\omega : E \to [0, +\infty)$ be a function assigning nonnegative weights to edges. Consider $s \in V$ as some distinguished vertex that is a source vertex in SSSP problem. The algorithm's purpose is as follows: for each $v \in V$, reachable from s, find a weight of a minimal (i.e. shortest) path, connecting s and v; let us denote this weight as $dist(v)$. So, considering $P(s, v)$ as a path from s to v, the goal is to find $dist(v) = \min_{P(s,v)} \omega(P(s, v))$. Weight of any path is clearly assumed as a sum of weights of all edges in that path. It is also safe to set $dist(v) = +\infty$ for any vertex v that is unreachable from s.

In: A.A. Makhnev, S.F. Pravdin (eds.): Proceedings of the 47th International Youth School-conference "Modern Problems in Mathematics and its Applications", Yekaterinburg, Russia, 02-Feb-2016, published at http://ceur-ws.org

2 Background

SSSP has a lot of efficient solutions. Most of the algorithms are derived either from Dijkstra's algorithm or from Bellman-Ford algorithm. General Dijkstra's algorithm implementation can achieve $O(n \cdot \log n + m)$ time complexity and Bellman-Ford algorithm runs in $O(nm)$ time. Some PRAM algorithms process vertices subsequently in Dijkstra's order, preforming edge relaxation in parallel [3]. This approach may be faster than simple Dijkstra's algorithm, yet it runs in $\Omega(n)$ time. Bellman-Ford algorithm is fairly simple to parallelize since it does all edge relaxations independently, but it is quite work inefficient. Known parallel BFS-based algorithms can perform in $O(\sqrt{n} \cdot \log n \cdot \log^* n)$ time and $O(\sqrt{n} \cdot m \cdot \log n)$ work [4].

Δ-stepping algorithm exposes both Dijkstra's and Bellman-Ford algorithms ideas. Let d be a maximum vertex degree in graph and $L = \max_{v \in V} dist(v)$. It is provable [2] that purely sequential Δ-stepping algorithm solves SSSP in $O(n + m + d \cdot L)$ time. An CRCW PRAM version implementation on arbitrary graphs with random positive weights and $\Delta = \Theta(\frac{1}{d})$ runs in $O(d \cdot L \cdot \log n + \log^2 n)$ time and $O(n + m + d \cdot L \cdot \log n)$ work on average [2]. Thus, for quite large class of graphs, e.g. graphs with constant maximum vertex degree and weights distributed in $[0, 1]$, this yields to $O(\log^2 n)$ time complexity and $O(n + m)$ work complexity. Even further optimisations like shortcuts insertion (assuming that distance $dist(v)$ for some vertex v is already calculated, we insert in graph edge (s, v), connecting source vertex with v and assign this edge weight $dist(v)$) may provide same results on wider class of graphs.

2.1 Sequential algorithm

The basic algorithm pseudocode is represented in Algorithm 1 [2].

Algorithm 1: Basic Δ-stepping algorithm

```
1  begin
2      foreach v ∈ V do
3          dist[v] ← +∞
4      dist[s] ← 0
5      while B ≠ ∅ do
6          i ← min{j : B[j] ≠ ∅}
7          R ← ∅
8          while B[i] ≠ ∅ do
9              Req ← FindRequests(B[i], light)
10             R ← R ∪ B[i]
11             B[i] ← ∅
12             RelaxRequests(Req)
13         Req ← FindRequests(B[i], heavy)
14         RelaxRequests(Req)
15 Function FindRequests(V', kind : {light|heavy})
16     return (w, dist[v] + ω(v, w)) : v ∈ V' & (v, w) ∈ E_kind
17 Function RelaxRequests(Req)
18     foreach (w, x) ∈ Req do
19         if x < dist[w] then
20             B[dist[w]/Δ] ← B[dist[w]/Δ] \ {w}
21             B[x/Δ] ← B[x/Δ] ∪ {w}
22             dist[w] ← x
```

Initially distance to the source vertex is 0 and distance to any other vertex is $+\infty$. This algorithm requires maintaining structure of vertices called *bucket*. Considering B as array of buckets, each bucket $B[i]$ stores set $\{v \in V : v \text{ is queued and } dist(v) \in [i \cdot \Delta, (i + 1) \cdot \Delta)\}$ as one-dimensional array. With cyclic reusing of buckets $|B|$ can be set to $\max_{e \in E} \lceil \omega(e)/\Delta \rceil + 1$.

On each iteration algorithm finds first nonempty bucket $B[i]$ and processes it in inner loop. It is looking for all *light* edges ($e \in E : \omega(e) \leq \Delta$), that are outgoing from vertices in bucket $B[i]$. Then all vertices are removed

from that bucket, but collected in set R for further processing. After that, relaxation of previously found light edges is performed. It is important, that some vertices may be reinserted in this bucket after relaxation due to improvement of their current distance. When current bucket $B[i]$ remains empty after inner loop, all weights for vertices $\{v : dist(v) \in [i \cdot \Delta, (i+1) \cdot \Delta)\}$ have been finally computed. Then all *heavy* edges $(e \in E : \omega(e) > \Delta)$, outgoing from every vertex that was removed from processed bucket and gathered in R, are relaxed. All process above is repeated until all buckets are empty.

3 Implementations

The main interest of this study is a parallel implementation of algorithm. There are a few efficient ways of providing parallel version of Δ-stepping algorithm. Although CPU and GPU implementations will differ in some way, they expose same ideas.

It has to be mentioned that in this paper was considered not an optimal parallel version of algorithm, yet very simple one, that is easy to analyze and implement.

Parallelization strategy is clearly straightforward: some set of vertices assigned for processing by one thread. All threads run in parallel. Hence, procedures like searching for all light or heavy edges outgoing from set of vertices (lines 9, 13) also performs in parallel.

Maintaining buckets structure, algorithm has to be able to concurrently determine whether each vertex belongs to any bucket, and return the index of bucket, that vertex belongs to, if it does. It can be implemented using two arrays. First one stores an index of bucket for each vertex in it, or -1 if vertex does not belong to any bucket. Second array stores size of each bucket.

Also, described strategy allows to relax found edges (lines 12, 14) in parallel.

3.1 CPU implementation

CPU implementation has restriction in number of threads running in parallel. So for that case all vertices are evenly distributed into chunks so that each thread has to process its own chunk. For instance, if p threads are available, all vertices are put into p chunks of size n/p. In this study this division is done automatically using OpenMP pragmas for lines 9–11, 12, 13, 14.

On every iteration of inner loop (lines 8–12) each thread for each vertex in its chunk subsequently checks if vertex belongs to current bucket and moves it to set R if it does, adding relaxation requests for light edges of this vertex. Then it moves to the next vertex. In next parallel OpenMP block each thread relaxes light edges requests for each vertex in its chunk that occurs in request.

When bucket remains empty, two OpenMP blocks of code process set R, creating relaxation requests for heavy edges, outgoing from vertices in that set, and performing that relaxation.

3.2 GPU implementation

For GPU implementation CUDA technology was used. Parallelization strategy is to assign to each CUDA thread a vertex from graph.

This specific implementation uses four different CUDA kernels for lines 9-11, 12, 13, 14. Each of them is being called with number of threads equal to number of vertices in graph, performing parallel processing, described above, for each vertex. However, edge relaxation is performed subsequently for each vertex.

Thus, on every iteration of inner loop each thread in first CUDA kernel checks if its vertex belongs to current bucket. If vertex belongs, thread creates corresponding relaxation-request for light edges and moves vertex to set R. In next CUDA kernel each thread performs subsequent relaxation of previously created requests.

Finally, third kernel creates requests for heavy edges for all vertex in set R, and last kernel relaxes that requests.

4 Benchmarking

4.1 Graphs

For performing benchmarks R-MAT graphs [5] were chosen. It is quite convenient graph model providing good approximation of real graph such as the Internet or social networks. They are configurable, match power-law, can have relatively small diameter, have opportunity to create “communities within communities”; also that

graph model may fit Erdös-Rény model as a special case. R-MAT graphs are well-studied [6] and widely used nowadays.

Let us denote that graph has *scale* n if it has 2^n vertices. In performed benchmarks R-MAT graphs with scale 17 to 23 and average vertex degree 32 were used. Weights of edges are double precision floating point numbers uniformly distributed in range $[0, 1]$.

Graphs were packed in CSR (compressed sparse row) data structure for achieving better memory efficiency while keeping constant time access to any vertex's adjacency list.

4.2 Hardware and software

All test were performed on single node of cluster of Institute of Mathematics and Computer Science in Ural Federal University with following configuration:

- CPU: Intel Xeon E5-2620 v2 @ 2.10 GHz, 12 cores, 32 GB of RAM,
- GPU: Nvidia Tesla K20Xm, 2688 CUDA Cores @ 732 MHz, 6 GB of memory.

All benchmarks code was written in C99. CPU versions were parallelized via OpenMP standard, GPU versions were written in CUDA 7.0. Compilers used: gcc 4.8.5, nvcc 7.0.27.

4.3 Bellman-Ford and Dijkstra algorithms for performance comparison

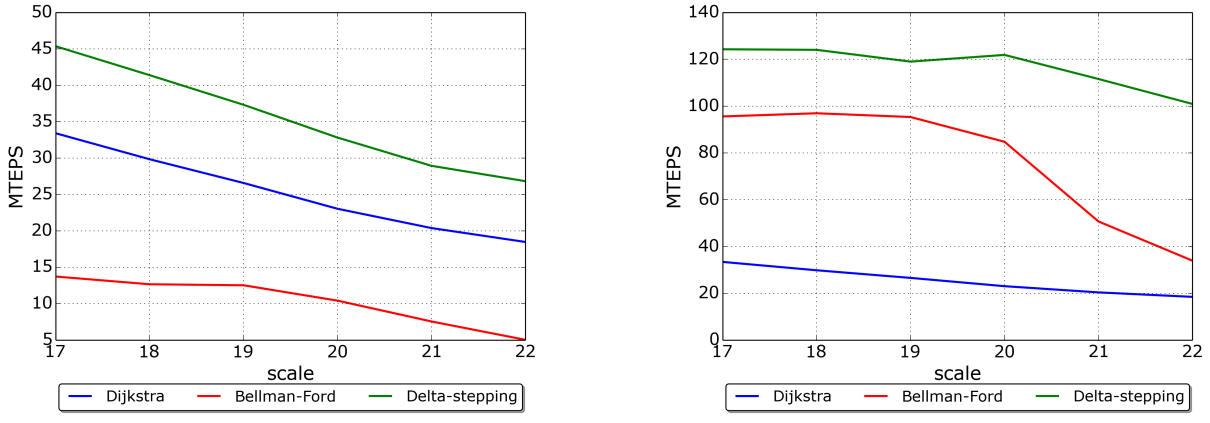
It is reasonable to firstly compare performance of most popular algorithms: Dijkstra's and Bellman-Ford with performance of Δ-stepping algorithm.

These tests were performed on CPU. We separately compared sequential implementations of all three algorithms and parallel implementations of Bellman-Ford and Δ-stepping algorithms along with sequential Dijkstra's algorithm. Figure 1 shows resulting performance of algorithms in millions of *traversed edges per second* (MTEPS). For that comparison Δ parameter was 0.04125.

Δ-stepping outperforms other algorithms in both sequential version and parallel one. This result corresponds asymptotic of these algorithms. Moreover, Δ-stepping does not degrade on larger graphs as Bellman-Ford algorithm does due to some cache efficiency. On each iteration Bellman-Ford algorithm has to relax all edges, so it has to do a lot of memory operations. Δ-stepping algorithm allows processor to keep some buckets in cache so it can access vertices in them and their outgoing edges very fast, keeping high performance.

4.4 Δ-stepping performance on CPU and GPU

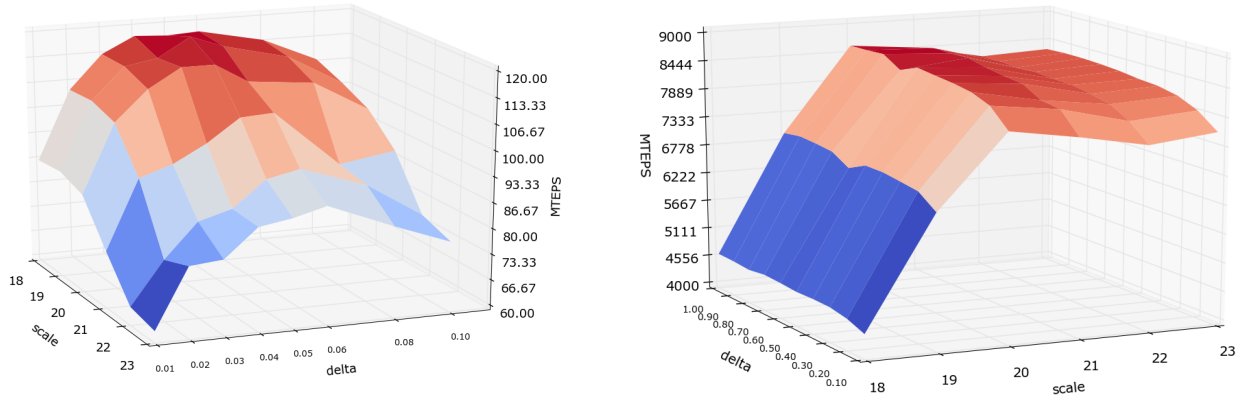
Performance results are based on 128 launches of algorithm for every graph and then taking a mean MTEPS value. Source vertex was randomly chosen each time. Also, for the sake of clarity of dependency on Δ parameter time of moving data to and from GPU was not included in total time of finding problem solution. That means that results in real-life case might be significantly lower. However, trend will be the same.



(a) Sequential implementations

(b) Parallel implementations

Figure 1: Performance comparison of different SSSP algorithms



(a) CPU performance

(b) GPU performance

Figure 2: Δ-stepping performance

Figure 2 shows final results. "Delta" axis represents range of Δ parameter, "scale" axis represents scales of tested R-MAT graph. Performance results are represented in millions of TEPS.

It is quite obvious to see that in tested setup CPU version is much more sensitive to Δ varying. It has to be mentioned that presented surfaces are on global maximum by Δ parameter. Making Δ lower or higher in both CPU and GPU version will provide results worse or equal.

5 Discussion

CPU results seem to fit theoretical research in [2] quite well. Δ that corresponds to highest performance level is a little bit higher than $1/\bar{d}$, where $\bar{d}$ is average vertex degree, 32 case of this study.

GPU performance results need more analysis to find out its cause. Firstly, it is clear that on relatively small graphs (scale 18), device can not achieve enough occupancy performing parallel processing of small number of vertices and edges, and therefore providing bad performance. On larger graph there are enough vertices to fully utilize device, which yields best performance, comparing to even bigger graphs, on which algorithm degrades due to graph scaling.

Quite topical question is analysis of dependency on Δ parameter. Seems like any $\Delta \geq 0.5$ provides good performance for no reason, because algorithm behaves differently on $\Delta = 1$ and $\Delta = 0.5$.

The point is that with relatively low Δ (0.5 in that case) algorithm behaves in optimal way, but average vertex degree in graph is quite small, so all parallel work is done for not large enough set of vertices simultaneously to achieve good occupancy on the device. Small vertex degree provides poor scaling. With higher Δ parameter algorithm treats more edges as light (and with $\Delta \geq 1$ all of them). That means that Δ-stepping algorithm behaves more like Bellman-Ford algorithm, providing good parallelization level, but doing a lot of work. However, with small mean vertex degree and relatively high Δ it is possible to get higher level of device occupancy while doing parallel work, providing better performance. By researching graphs with higher vertex degree Δ parameter can be chosen closer to theoretical best one [7].

It is reasonable to choose $\Delta = \frac{n}{m}$ [8]. For R-MAT, however, equation $1/\bar{d} = \frac{n}{m}$ takes place. Taking $\Delta = 1/32$ in our case results to worse performance than $\Delta = 0.1$ and especially $\Delta = 0.5$.

Δ-stepping algorithm performance depends on Δ parameter because configuring this parameter may solve trade off between a lot of phase number and too much work on each phase [2] [7]. However, in our study graph size ceiling for tested hardware does not allow to consider graphs large enough to show algorithm performance degradation with low number of phases and a lot of work on each phase, but does allow to see that if Δ is too small, and thus there are too many phases without high workload, algorithm performance decreases.

6 Conclusion

This study presents results of one approach of Δ-stepping parallelization and comparison of performance with different Δ parameters. We show that optimal Δ parameter for R-MAT graphs with mean vertex degree 32 is between 0.04 and 0.05 for CPU. Optimal Δ parameter for GPU implementation on same graphs is 0.5 and may become smaller using graphs with higher vertex degree.

This result is quite case-specific due to some restrictions, yet is interesting to analyse and extrapolate on other similar cases.

Further research may be focused on determining best performance algorithm setup on different systems.

References

[1] U. Meyer and P. Sanders. Δ-stepping: A parallel single source shortest path algorithm. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1461 LNCS:393–404, 1998.

[2] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algortihm. *Journal of Algorithms*, 49:114–152, 2003.

[3] G.S. Brodal, J.L. Träff, C.D. Zaroliagis. A parallel priority queue with constant time operations. *Journal of parallel and distributed computing*, 49:4–21, 1998.

[4] P.N. Klein and S. Subramanian. A Randomized Parallel Algorithm for Single-Source Shortest Paths. *Journal of Algorithms*, 25:205–220, 1997.

[5] D. Chakrabarti, Y. Zhan, C. Falaotsos. R-MAT: A recursive model for graph mining. *SIAM Proceedings Series*. Proceedings of the Fourth SIAM International Conference on Data Mining, 442–446, 2004.

[6] C. Groër, B.D. Sullivan, S. Poole. A mathematical analysis of the R-MAT random graph generator. *Networks*, 58(3):159–170, 2011.

[7] V.T. Chakaravarthy, F. Checconi, F. Petrini, Y. Sabharwal. Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems. *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, 889–901, 2014.

[8] K. Madduri, D.A. Bader, W.B. Jonathan, J.R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments and the 4th Workshop on Analytic Algorithms and Combinatorics*, 23–25, 2007.