SERBIAN JOURNAL OF ELECTRICAL ENGINEERING
Vol. 13, No. 1, February 2016, 31-43

UDC: 004.9:519.8 **DOI: 10.2298/SJEE1601031V**

Performance of Shortest Path Algorithm Based on Parallel Vertex Traversal

Mihailo Vesović[1], Aleksandra Smiljanić[1], Dušan Kostić[1]

Abstract: Shortest path algorithms for different applications, such as Internet routing, VLSI design and so on are used. Dijkstra and Bellman-Ford are commonly used shortest path algorithms which are typically implemented in networks with hundreds of nodes. However, scale of shortest path problems is increasing, and more efficient algorithms are needed. With the development of multicore processors, one natural way to speedup shortest path algorithms is through parallelization. In this paper, we propose a novel shortest path algorithm with parallel vertex transversal, and compare its speed with standard solutions in datacenter topologies.

Keywords: Parallelization, Shortest Path Algorithms, Single Source Shortest Path, Bellman-Ford, Dijkstra

1 Introduction

Shortest path algorithms are particularly important on the Internet, as they determine the most efficient paths along which packets should reach their destinations. OSPF and IS-IS routing protocols employ Dijkstra algorithm, while RIP utilizes Bellman-Ford algorithm [1]. These algorithms find shortest paths from a single source node to all other nodes, and, they are called single source shortest path (SSSP) algorithms for this reason [2]. Complexities of these two algorithms are $O(mn)$ and $O(m+n\log n)$, respectively, where n is the number of vertices, and m the number of edges.

Networks have hierarchical structure, and routers do not get the full information about network topology. Shortest paths are calculated in domains of several hundred routers. However, nowadays large datacenter networks with flat topologies are developing. In such networks, SSSP algorithms have to become faster [3, 4].

Let as denote a digraph representing a given network as a pair $G=(V,E)$, where V represents a set of n vertices, $V=\{v_i \mid i=0,1,\dots,n-1\}$, and E

[1]School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia; E-mails: mikives@gmail.com; aleksandra@etf.rs; kostic.dusan1990@gmail.com

represents a set of m edges, $E = \{e_i \mid i = 0,1,\ldots,m-1\}$. Edge $e_i = (v_j, v_k, w_i)$ is defined as a link between vertices v_j and v_k, and is considered to be directed from predecessor vertex v_j to successor vertex v_k, while w_i is weight associated to this edge.

Bellman-Ford algorithm is algorithm based on edge relaxations [5 − 7], i.e. updating terminal vertex distance with better value if it is obtained in the current iteration as shown by pseudo code given in Fig. 1.

```
BELLMAN-FORD (G, s)
    for each vertex $v \in G.V$
        $v.d$ = INF
        $v.p$ = NULL
    $s.d = 0$
    for $i$ = 1 to $|G.V|$ - 1
        for each edge $(u, v, w) \in G$.E
            if $v.d > u.d + w$
                $v.d = u.d + w$
                $v.p = u$
    for each edge $(u, v, w) \in G.E$
        if $v.d > u.d + w$
            return FALSE
    return TRUE
```

Fig. 1 – *Bellman-Ford algorithm pseudo code [4].*

At first, vertex distances are set to maximal values (INF), and vertex predecessors to NULL, except for the source vertex with the distance which is initially set to 0. Afterwards, edge relaxations are performed for all the edges, $n-1$ times at maximum. In each iteration, distance of terminal vertex is changed to the sum of the predecessor vertex distance and the edge weight, if this sum has smaller value.

Dijkstra algorithm [8] iterates through all vertices in predefined order. In each iteration, a vertex with minimum distance is found among those which have not been previously processed. In order to find such vertices, minimum priority queues are used [9]. The fastest version of Dijkstra utilizes Fibonacci heap [10] as its priority queue. In practice, binary heaps are used instead, because of their simplicity. Pseudo code of Dijkstra algorithm is shown in Fig. 2.

Initialization phase is identical as in Bellman-Ford algorithm. Then, the vertex with the smallest tentative distance is extracted from the minimum priority queue, and its edges are relaxed. At the end of the relaxation procedure, minimum priority queue must be rearranged.

```
DIJKSTRA (G, s)
  for each vertex v ϵ G.V
      v.d = INF
      v.p = NULL
  s.d = 0
  Q = G.V
  while Q ≠ ∅
      u' = min (Q )
      Q = Q \ { u' }
      for each edge (u, v, w) ϵ G.E such that u = u'
          if v.d > u.d + w
              v.d = u.d + w
              v.p = u
              rearrange (Q, v)
```

Fig. 2 – *Dijkstra algorithm pseudo code [4].*

There are several proposed parallel SSSP algorithms designed for graphic processor units (GPU). These solutions offer significant speedups, but at a higher cost. Paper [11] proposes efficient GPU algorithm in which the number of threads should be equal to the number of edges. In this way, all edges could be relaxed in parallel. GPU solutions are not suitable for general purpose processors (GPP) because of the much smaller number of cores on GPPs. Algorithms designed for GPUs need to be significantly modified in order to be executed on GPPs.

In recent years, GPP frequency scaling has reached its limit. Processors are improved by comprising multiple cores on a single chip, and workload is distributed accross these cores. The number of cores used in GPPs is typically below 8, and it is much smaller than in the case of GPUs with hundreds of cores. However, GPPs are much cheaper. For this reason, we are proposing a novel parallelized SSSP algorithm for GPPs. Our proposed algorithm is a combination of Dijkstra and Bellman-Ford algorithms. It traverses the nodes in a similar order as Dijkstra, but using the iterative procedure as in Bellman-Ford which can be parallelized. We will compare the speed of our proposed algorithm with speeds of commonly used shortest path algorithms, Dijkstra and Bellman-Ford, when implemented on GPPs. In addition, we will examine dependence of these algorithms on the network parameters, such as the average node degree, and network diameter.

2 Proposed Parallel SSSP Algorithm

Dijkstra has been proven to be the fastest sequential SSSP algorithm. In Dijkstra, each vertex gets the chance to relax its edges, one after another. Vertex traversing order is strictly defined – next vertex that is going to relax its edges is the one that currently has the minimal distance. Dijkstra is, therefore, hard to parallelize as a global minimum needs to be found in each iteration.

In this paper, we propose SSSP algorithm which could be executed in parallel, while maintaining smart vertex traversal order. Its pseudo code is given in Fig. 3. Different vertices are assigned to each thread and edges adjacent to these vertices are processed in parallel. Only vertices with the distance that was updated in the (k–1)-th iteration will relax their edges in the k-th iteration. In the first iteration, only source vertex will relax its edges.

It is not practical to assign thread to each vertex when GPP is used due to huge overheads even for fairly small topologies. Vertices must share threads. As soon as a thread finishes edge relaxations of one vertex, it will get another one to process.

Some synchronization primitives need to be introduced to avoid race conditions. During the edge relaxation procedure, terminal vertices need to be locked. Secondly, algorithm will differentiate even and odd iterations. As it was already mentioned, in the k-th iteration only vertices with the distance that was updated in the previous, (k–1)-th iteration, need to relax their edges. However, due to parallelization, it may happen that the vertex labeled in the k-th iteration relaxes its edges in the same, k-th, iteration which might corrupt the results. In order to prevent data corruption, additional lists for storing labeled vertices are used to differentiate between vertices updated in odd and even iterations, as shown in the pseudo code in Fig. 3. Here, S1 and S2 are used in order to store information about labeled vertices from even and odd iterations, and vertices labeled in odd iterations can only relax their edges in even iteration, and vice-versa.

General assumption is that each vertex can be reached from the root vertex and that the graph does not have any negative weight loops. If these conditions are satisfied, convergence is guaranteed. Additionally, algorithm can be generalized to detect negative weight loops. If the loops exist, the shortest path tree cannot be found.

Few considerations about total number of threads need to be made. First, number of threads should be equal to the total number of cores in a system. In general, it is possible to spawn arbitrary number of threads across cores. However, these threads would need to share vital hardware resources, such as the arithmetical unit, and execution time might get longer.

```
PARALLEL-SSSP (G, s)
    for each vertex v ϵ G.V
            v.d = INF
            v.p = NULL
    s.d = 0
    S1 = ∅
    S2 = {s}
    odd = FALSE
    while S1 ≠ ∅ or S2 ≠ ∅
            odd = ¬odd
            for each vertex u' ϵ G.V do in parallel
                    if (odd and u' ϵ S2) or (¬odd and u' ϵ S1)
                            for each edge (u, v, w) ϵ G.E such that u = u'
                                    LOCK (v)
                                    if v.d > u.d + w
                                            v.d = u.d + w
                                            v.p = u
                                            if odd
                                                    S1 = S1 ∪ {v}
                                            else
                                                    S2 = S2 ∪ {v}
                                    UNLOCK (v)
                    if odd
                            S2 = S2 \ { u' }
                    else
                            S1 = S1 \ { u' }
```

Fig. 3 – *Parallel SSSP algorithm based on Bellman-Ford.*

3 Implementation Details

We have implemented sequential Bellman-Ford algorithm, Dijkstra algorithm with binary heap priority queue and proposed parallel SSSP algorithm. Algorithms are implemented in programming language C and OpenMP is used for parallelization.

Table 1
Testing Machine Specification.

Processor	Intel® Core™ i7-5820K Processor
Number of Cores	6
Hyperthreading	2 threads per core
Frequency	3.3GHz
Cache size	L1d = 32K, L1i=32K, L2=256K, L3=15M
RAM	16GB, DDR4
OS	Ubuntu 14.04.1

Same data structures were used for all algorithms in order to provide a fair comparison. Linux operating system was used, and GCC was chosen as a compiler. Test cases were generated by scripts written in Python scripting language (version 3.4). Specifications of the PC used for testing are shown in **Table 1**.

4 Results and Discussion

4.1 Execution times for standard datacenter topologies

We have analyzed performance of implemented algorithms on common datacenter topologies: dragonfly [12], flattened butterfly [13], fat-tree [14] and slim fly topology [15].

In dragonfly topology, nodes are divided into groups according to their proximity so that the number of longer and more costly links is reduced. Every pair of nodes inside one group is connected, while each group is connected to all other groups. Dragonfly topology is described with two parameters, number of vertices in a group - *p,* and number of external connection per each vertex – *q*. Total number of groups is calculated as $g = pq + 1$. The average vertex degree for dragonfly topology is $p + q - 1$, and the diameter is 3.

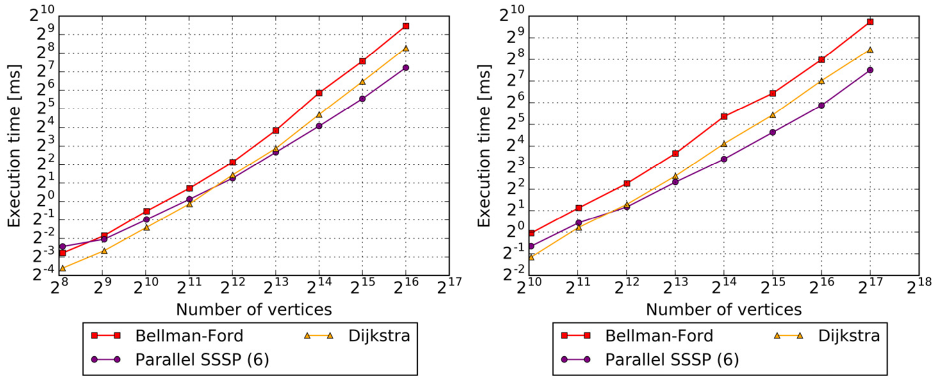

Fig. 4 – *Execution times versus number of vertices in dragonfly topology with:* (a) *16 nodes in group*; (b) *32 nodes in group.*

Algorithms were tested on dragonfly topologies with $p = 16$ and $p = 32$ vertices in a group, and various q parameters, $q \in \{2^0, 2^1, \ldots, 2^8\}$ and $q \in \{2^0, 2^1, \ldots, 2^7\}$ respectively. Obtained results are presented in Fig. 4. For small number of vertices, parallel SSSP algorithm has worse execution time than Dijkstra algorithm. In this case, the time necessary for creation of threads exceeds the total time for finding the SSSP. On the other hand, for larger graphs, parallel SSSP algorithm outperforms both Bellman-Ford and Dijkstra algorithms. As the number of vertices exceeds 2^{15}, parallel SSSP algorithm is two times better than Dijkstra for both topologies. Parallel SSSP could yield the same or even better performance with the increasing number of processor cores.

Flattened butterfly topology has $n = 2^p$ number of nodes. It can be defined as follows: if nodes are labeled with numbers from 0 to $n-1$, only pairs of nodes with the binary representations of labels that differ in only 1 bit are connected. It is easy to see that the number of edges is $n\log_2 n$, while the average vertex degree and the diameter for flattened butterfly topology are both p.

Fig. 5 presents execution time of all algorithms for various numbers of vertices in flattened butterfly topology, $p \in \{10, 11, \ldots, 19\}$. It can be seen that the best algorithm for this topology is Dijkstra algorithm, and it is followed by Bellman-Ford algorithm. As the number of vertices exceeds 2^{17}, parallel SSSP algorithm is better to use than Bellman-Ford. Parallel SSSP is likely to improve its performance as the number of cores increases.

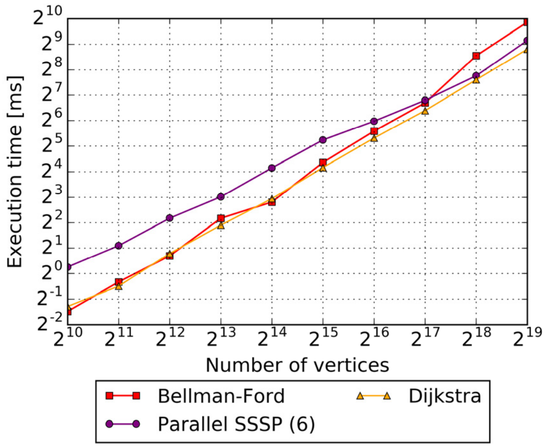

Fig. 5 – *Execution times versus number of vertices in flattened butterfly topology.*

Fat tree topology is a popular datacenter topology. We will consider fat tree topology with vertices that are organized in three levels. This fat tree topology has $n = 5k^2/4$ nodes, where k is the number of groups. The average vertex degree for fat tree topology is $4k/5$, and the diameter is 4.

We tested algorithms in fat tree topologies with three levels and various numbers of nodes, $k \in \{2^4, 2^5, 2^6, 2^7, 2^8\}$. Since results were similar for nodes at different levels of hierarchy, we present only the results for nodes that are at the top level of hierarchy in Fig. 6. It can be seen that speed-up effects of parallel SSSP compared to sequential Bellman-Ford and Dijkstra is notable for number of nodes larger than $2^{11} = 2048$ and $2^{12} = 4096$, respectively. For smaller number of nodes execution times of all three algorithms are below 1ms. For larger number of nodes parallel SSSP has the best performance and sequential Bellman-Ford has the worst performance. Performance of Dijkstra is between parallel SSSP and sequential Bellman-Ford.

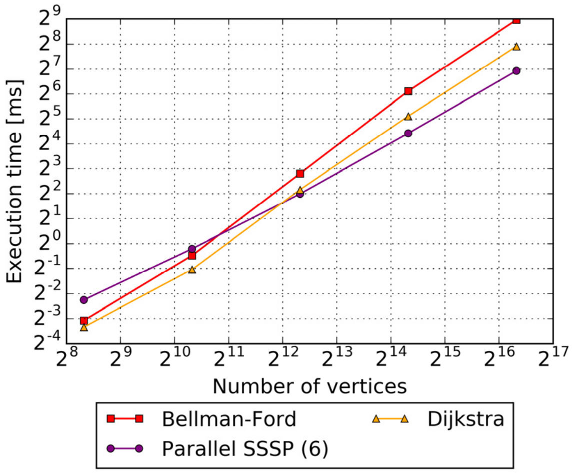

Fig. 6 – *Execution times versus number of vertices in fat tree topology.*

In slim fly topology, two groups of nodes are generated. Inside these groups, nodes are divided into q subgroups with q nodes in each subgroup, where q is a prime number. Based on rules given in paper [15], nodes inside each subgroup are connected and subgroups within different groups are interconnected in a different manner. The total number of nodes is $2q^2$. Parameter q can be represented in the form $q = 4\omega + \delta$, $\omega \in N$, $\delta \in \{-1, 0, 1\}$. The average vertex degree for slim fly topology is then $(3q - \delta)/2$, and the diameter is 2.

Results for different $q \in \{53, 61, 73, 89, 97, 101, 109, 113, 137\}$ in slim fly topology are shown in Fig. 7. Parallel SSSP outperforms both Dijkstra and Bellman-Ford for all numbers of nodes. On a machine with more available cores, the speed-up of parallel SSSP algorithm is expected to be even more pronounced.

It can be concluded from the previous graphs that the speed of parallel SSSP algorithm increases compared to Dijkstra as the number of vertices increases. In each of the graphs, vertex degree increases with the graph size. In graphs from Fig. 4 associated with dragonfly topology, vertex degree is $p+q-1$, where p is constant, and q is increasing along x-axis. Vertex degree is in the range $15 \le d \le 271$ for $p=16$, and $31 \le d \le 159$ for $p=32$. Similarly, vertex degree increases with the increase of the graph size in other figures as well. Vertex degree is in the range $10 \le d \le 19$ for flattened butterfly topology in Fig. 5, $12.8 \le d \le 204.8$ for fat-tree topology in Fig. 6 and $79 \le d \le 205$ for slim fly topology in Fig. 7. In all the graphs, SSSP algorithm gets better compared to Dijkstra as the average vertex degree increases.

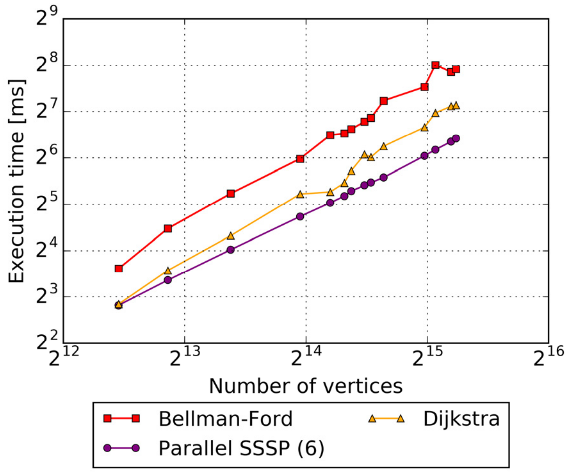

Fig. 7 – *Execution times versus number of vertices in slim fly topology.*

In network topologies with small and constant diameter such as dragonfly, fat tree and slim fly, parallel SSSP is faster algorithm than Dijkstra for almost any number of vertices. However, in flattened butterfly topology from Fig. 5, the diameter was increasing with the graph size in range $10 \le l \le 19$. In this case, Dijkstra algorithm was advantageous. We conjecture that the Dijkstra algorithm is more efficient for the topologies with larger diameter.

4.2 Execution times versus graph diameter and average vertex degree

In this section, we will further explore dependence of the execution time of parallel SSSP algorithm on graph parameters such as the average vertex degree *d* and the network diameter *l*. Based on these findings, we will derive general conclusions about suitability of the SSSP algorithms for particular network topologies.

Diameter of the graph, with the given starting vertex, can be calculated as the depth of its shortest path tree. In our parallel SSSP algorithm, in the i-*th* outer-loop iteration, all vertices that have the depth of *i* in the shortest path tree will have its distance settled, provided that there are no negative weight cycles. Therefore, the total number of outer loop iterations is $l+1$, where *l* is the graph diameter. In each outer loop iteration, only labeled vertices have to relax their edges. Vertex v_i has to relax $d(v_i)$ outer edges, where $d(v_i)$ is the vertex degree. Let assume that all vertices have the same vertex degree, *d*. The amount of work necessary to perform all relaxations in one outer-loop iteration is proportional to the vertex degree, *d*.

Dijkstra algorithm performs only one outer-loop iteration, which is its advantage. However, compared to the parallel SSSP, Dijkstra has to perform much more work in this iteration. The work includes rearrangement of the priority queue for each successful relaxation. Rearrangement of the priority queue for one edge is done in $O(\log(n))$ time, while parallel SSSP algorithm does not have to arrange priority queue, i.e. relaxation operation for one edge can be performed in $O(1)$ time. Additionally, edge relaxations are performed in parallel.

As a consequence, parallel SSSP algorithm is advantageous when the network diameter is small and when the vertex degree is large, which is usually the case in datacenter topologies. In fact, we have already observed such behavior in the previous section. In topologies that have small diameter, like slim fly, dragonfly and fat tree, parallel SSSP algorithm performs much faster. However, in the flattened butterfly topology, diameter is larger, and the parallel SSSP loses its advantage compared to Dijkstra.

In order to confirm this observation, we have created custom topology in which the network parameters can be independently set. Because of its unique shape, we will name this topology *stellar*. Stellar topology can be defined as follows:

- Vertices are organized in $K>1$ groups g_i, $i=1,...,K$. The number of vertices in each group is *D*. Vertices are labeled as x_{ij}, $i=1,...,K$, $j=1,...,D$ where *i* denotes the group in which the vertex is located;
- In each group, vertices are connected in full mesh, i.e. each vertex has a bidirectional connection to each other vertex in a group;
- There is a bidirectional connection between the vertices x_{ij} and $x_{(i+1)j}$, $i=1,...,K-1$, $j=1,...,D$, and between the vertices x_{1j} and x_{Kj}, $j=1,\dots,D$.

One example of the stellar topology, for $K=D=4$, is shown in Fig. 8.

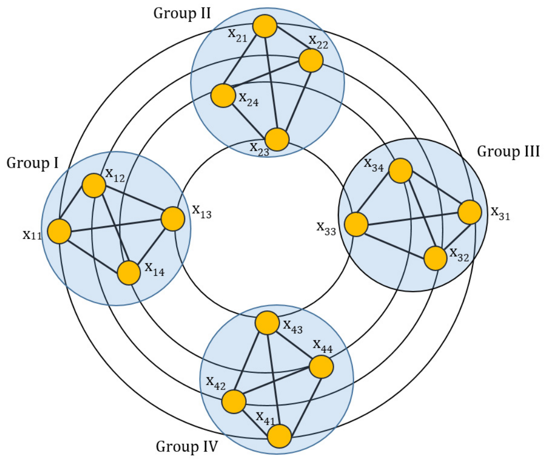

Fig. 8 – *Stellar topology for K=D=*4.

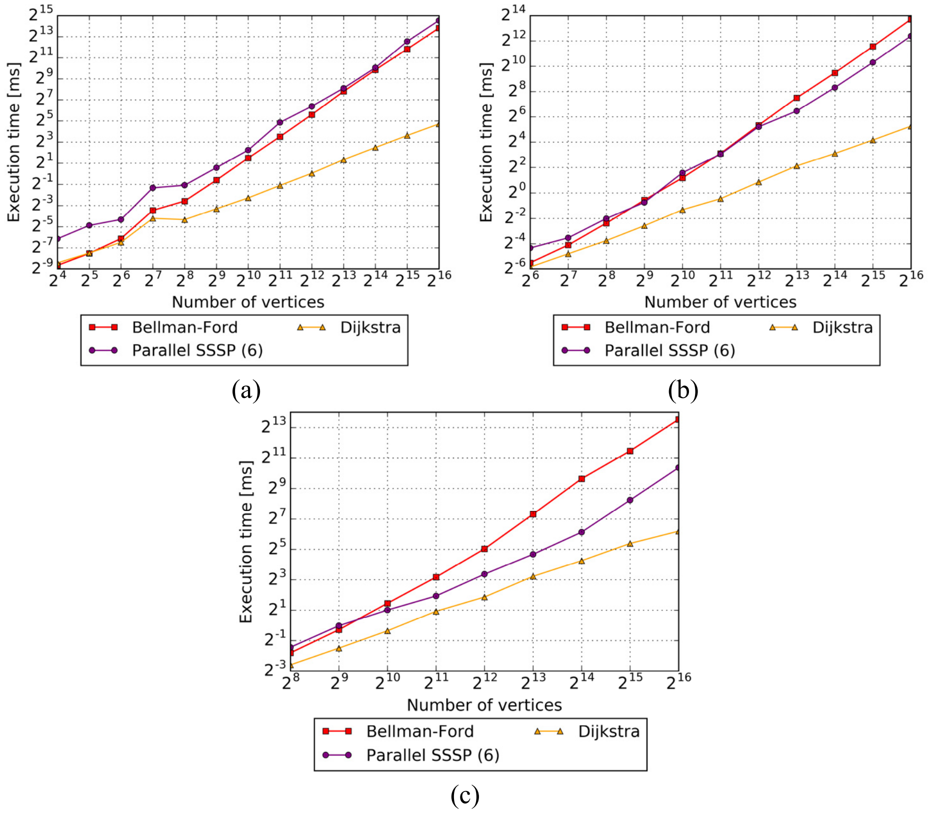

Fig. 9 – *Execution times for varying number of vertices in stellar topology in which:* (a) *D* = 4; (b) *D* = 16; (c) *D* = 64.

In stellar topology, vertex degree is $d = D + 1$ for all vertices as each vertex has $D - 1$ internal and two external links. Two most distant groups are separated by $\lfloor K/2 \rfloor$ hops. Vertices of one group are only one hop away from each other, as they are connected in a full mesh topology. Therefore, the network diameter is $l = 1 + \lfloor K/2 \rfloor$. From the stellar topology definition, it follows that $l \neq l(d)$ and $d \neq d(l)$, which means that these two parameters can be independently set.

Execution speeds of the algorithms under the consideration in stellar topology with fixed vertex degree and varying number of vertices are shown in Fig. 9, for $D \in \{4,16,64\}$. Vertex degree is $d \in \{5,17,65\}$, and the diameter is in the range $3 \leq l \leq 8193$, $3 \leq l \leq 2049$ and $3 \leq l \leq 513$ respectively. If the vertex degree is fixed, the number of vertices is increasing with the increase of the network diameter. It can be observed that Dijkstra outperforms parallel SSSP in stellar networks in which vertex degrees are limited and network diameters are large. In the case of smaller networks, both diameter and vertex degrees are small, and execution times depend primarily on the operating system overhead. In addition, parallel SSSP algorithm approaches the performance of Dijkstra as the vertex degree increases, which can be observed from graphs in Fig. 9. This confirms our conjecture that the performance of parallel SSSP becomes advantageous in networks with large vertex degree and small network diameter.

5 Conclusion

Sequential shortest path algorithms such as Bellman-Ford and Dijkstra dominate on the Internet. As the Internet is growing, faster and more scalable algorithms are desired. Parallel algorithm proposed in this paper outperforms Bellman-Ford for most of the tested topologies. Performance of parallel SSSP algorithm is in all cases comparable to the performance of Dijkstra and it outperforms Dijkstra in large networks. We have shown that our algorithm works well for the common datacenter topologies, especially for the topologies in which the diameter is small and the average vertex degree is large. Another advantage of the proposed algorithm is its potential, as the higher number of cores is likely to give increasing speedups.

6 Acknowledgement

The Serbian Ministry of Science and Education and companies Telekom Srbija and Informatika supported this work.

7 References

[1] A. Smiljanić: Internet Fundamentals and Applications, Academic Mind, Belgrade, Serbia, 2015. (In Serbian).

[2] T.H. Cormen, E.C. Leiserson, R.L. Rivest: Introduction to Algorithms, MIT Press, Cambridge, MA, USA, 1990.

[3] A. Smiljanić, N. Maksić: Improving Utilization of Data Center Networks, IEEE Communication Magazine, Vol. 51, No. 11, Nov. 2013, pp. 32 – 38.

[4] A. Smiljanić, J. Chao, C. Minkenberg, E. Oki, M. Hamdi: Switching and Routing for Scalable and Energy-Efficient Networking, Vol. 32, No. 1, Jan. 2014, pp. 1 – 3.

[5] R. Bellman: On a Routing Problem, Quarterly of Applied Mathematics, Vol. 16, No. 1, 1958, pp. 87 – 90.

[6] L.R. Ford, D.R. Fulkerson: Flows in Networks, Princeton University Press, Princeton, NJ, USA, 1962.

[7] E.F. Moore: The Shortest Path through a Maze, International Symposium on the Theory of Switching, Cambridge, MA, USA, 02-05 April 1957.

[8] E.W. Dijkstra: A Note on Two Problems in Connection with Graphs, Numerische Mathematik, Vol. 1, No. 1, Dec. 1959, pp. 269 – 271.

[9] R.E. Tarjan: Data Structures and Network Algorithms, SIAM, Philadelphia, PA, USA, 1983.

[10] M.L. Fredman R.E. Tarjan: Fibonacci Heaps and their uses in Improved Network Optimization Algorithms, Journal of the ACM, Vol. 34, No. 3, July 1987, pp. 596 – 615.

[11] D. Dundjerski, M. Tomasević: Graphical Processing Unit-based Parallelization of the Open Shortest Path First and Border Gateway Protocol Routing Protocols, Concurrency and Computation: Practice and Experience, Vol. 27, No. 1, Jan. 2015, pp. 237 – 251.

[12] J. Kim, W.J. Dally, S. Scott, D. Abts: Technology-driven, Highly-scalable Dragonfly Topology, International Symposium on Computer Architecture, Beijing, China, 21-25 June 2008, pp. 77 – 88.

[13] J. Kim, W.J. Dally, D. Abts: Flattened Butterfly: A Cost-Efficient Topology for High-Radix Networks, International Symposium on Computer Architecture, San Diego, CA, USA, 09-13 June 2007, pp. 126 – 137.

[14] M. Al-Fares, A. Loukissas, A. Vahdat: A Scalable, Commodity Data Center Network Architecture, Conference SIGCOMM 2008, Seattle, WA, USA, 17-22 Aug. 2008, pp. 63 – 74.

[15] M. Besta, T. Hoefler: Slim Fly: A Cost Effective Low-diameter Network Topology, International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 16-21 Nov. 2014, pp. 348 – 359.