Computing single-source shortest paths on graphs with over 8 trillion edges

1PDAMO GmbH
20000-0002-2877-074X
30000-0002-1967-0077
4Kyushu University, Institute of Mathematics for Industry
50000-0002-1967-0077
60000-0001-7848-1172
7RIKEN Center for Computational Science
80000-0002-2902-882X
Abstract
This paper introduces an implementation for solving the single-source shortest path problem on distributed-memory machines. It is tailored to power-law graphs and scales to trillions of edges. The new implementation reached 2nd and 10th place in the latest Graph500 benchmark in June 2022 and handled the largest and second-largest graphs among all participants.

1 Introduction
The single-source shortest path problem (SSSP) is one of the fundamental problems in combinatorial optimization and can be found in many practical applications [Chen, 1996]. Given a weighted graph and a vertex $r$, the SSSP is to find shortest paths from $r$ to all other vertices of the graph. In the age of big data, the size of the graphs to be handled is ever increasing.

Against this backdrop, the Graph500 benchmark was initiated in 2010 to evaluate large-scale graph processing performance [gra]. New listings of the top-performing systems are released every six months (June and November); the benchmark consists of a breadth-first search (BFS) and an SSSP category. For both BFS and SSSP a measure named traversed edges per second (TEPS) is used. Given a a graph with $m$ edges, the TEPS for an SSSP computation on this graph that took $t$ seconds is defined as $\frac{m}{t}$. In the Graph500 benchmark, a scale-free graph called Kronecker graph [Leskovec et al., 2010] is used. The term scale-free describes a graph whose vertex degree distribution follows (at least asymptotically) a power-law distribution. Social network graphs, for example, are known to usually be scale-free.

Contributions This article describes a distributed memory SSSP algorithm for large-scale graphs. It is especially tailored to scale-free graphs. As opposed to previous state-of-the-art implementations, the new algorithm is based on...
a 2D partitioning of the adjacency matrix of the underlying graph. Furthermore, we introduce several improvements to speed up the performance. The new algorithm is shown to scale to graphs with over 8 trillion edges. A previous implementation of the algorithm reached the 2nd and 10th place of the Graph500 benchmark, while processing the largest and second-largest graphs of all Graph500 SSSP participants.

2 Single-source shortest path algorithms

In the following, consider an undirected graph \( G = (V, E) \), edge weights \( c : E \to \mathbb{R}_{\geq 0} \), and a vertex \( r \in V \) called root. For simplicity, we will assume that \( G \) is connected. For any vertex \( v \in V \), the distance \( d^*(v) \) of \( v \) is defined as the length (with respect to \( c \)) of a shortest path between \( r \) and \( v \).

The SSSP algorithms described in the following all maintain a tentative distance \( d(v) \) for each \( v \in V \). Throughout the execution of the algorithms it holds that \( d^*(v) \leq d(v) \) for all \( v \in V \). At termination it holds that \( d^*(v) = d(v) \) for all \( v \in V \). During execution, we say that a vertex \( v \in V \) is settled if the respective algorithm can guarantee that \( d^*(v) = d(v) \) holds. To update the tentative distance \( d(v) \) of a vertex \( v \in V \), a so-called relaxation along an incident edge \( \{u, v\} \) of \( v \) is used. This operation is defined as follows:

\[
d(v) \leftarrow \min\{d(v), d(u) + c(\{u, v\})\}.
\]

Based on this operation, we describe several SSSP algorithms below.

**Dijkstra’s algorithm [Dijkstra, 1959]** The algorithm maintains a partition of \( V \) into settled, queued, and unreached vertices. Queued vertices satisfy \( d(v) < \infty \) but are not settled yet, whereas unreached vertices satisfy \( d(v) = \infty \). Initially, only \( r \) is queued. We set \( d(r) = 0 \), and \( d(v) = \infty \) for all \( v \in V \setminus \{r\} \). In each iteration of Dijkstra’s algorithm, a queued vertex \( u \) with minimum tentative distance among all queued vertices is removed from the queue. All incident edges \( \{u, v\} \) of \( u \) are relaxed. Any vertex \( v \) whose tentative distance is reduced by the relaxation is (re-) inserted into the queue. The algorithm terminates once the queue is empty. It can be shown that a vertex is settled as soon as it has been removed from the queue. Using a Fibonacci heap [Fredman and Tarjan, 1987], one can realize Dijkstra’s algorithm in time \( O(|V| \log |V| + |E|) \).

**Bellman-Ford algorithm [Bellman, 1958]** Unlike Dijkstra’s algorithm, the Bellman-Ford algorithm performs the relaxation operation from several vertices in each iteration. The algorithm keeps a list of active vertices for each iteration. Initially, only the root vertex is active. In each iteration, for each active vertex \( u \) all incident edges \( \{u, v\} \) are relaxed. Each vertex whose tentative distance is decreased is treated as an active vertex in the next iteration. The Bellman-Ford algorithm terminates once there are no more active vertices. The algorithm is guaranteed to require at most \( |V| - 1 \) iterations. The worst-case run time is \( O(|V||E|) \).

**Delta-stepping algorithm [Meyer and Sanders, 2003]** One observes that Dijkstra’s algorithm requires a small amount of work (since each edge is relaxed
Algorithm 1: Basic delta-stepping

Data: SSSP instance $I = (V, E, c, r)$

Result: Shortest path distances $d(v)$ for each $v \in V$

1. foreach $v \in V \setminus \{r\}$ do $d(v) := v$
2. foreach $k = 1, 2, 3, \ldots$ do $B_k := \emptyset$
3. $d(r) := 0$
4. $B_0 := \{r\}$
5. $B_{\infty} := V \setminus \{r\}$
6. $k := 0$
7. while $k < \infty$ do
   8.     $X := B_k$
   9.     while $X \neq \emptyset$ do
      10.        $X' := \emptyset$
      11.        foreach $\{u, v\} \in E$ with $u \in X$ do
      12.            $q := \left\lfloor \frac{d(v)}{\Delta} \right\rfloor$
      13.            $d(v) := \min\{d(v), d(u) + c(\{u, v\})\}$
      14.            $q' := \left\lfloor \frac{d(v)}{\Delta} \right\rfloor$
      15.            if $q' < q$ then
      16.                $B_q := B_q \setminus \{v\}$
      17.                $B_{q'} := B_{q'} \cup \{v\}$
      18.            end
   19.     if $q' = k$ then $X' := X' \cup \{v\}$
   20.     end
   21.     $X := X'$
   22. end
23. $k := \min\{q > k : B_q \neq \emptyset \lor q = \infty\}$
24. end

only once). However, it always requires $|V| - 1$ iterations. On the other hand, the Bellman-Ford algorithm usually requires more work, but far fewer iterations. The delta-stepping algorithm aims for a middleground between these two behaviours. Initially, choose a constant $\Delta \in \mathbb{R}_{>0}$. Throughout the algorithm, the vertex set $V$ is partitioned into buckets, depending on the tentative distances. For each integer $k \geq 0$, define

$$B_k := \{v \in V : d(v) \in [k\Delta, (k+1)\Delta)\}.$$

All vertices with $d(v) = \infty$ are assigned to the bucket $B_{\infty}$. The bucket index $k$ of any vertex $v$ is given by $\left\lfloor \frac{d(v)}{\Delta} \right\rfloor$.

Initially, we set $d(r) := 0$ and $d(v) = \infty$ for all $v \in V \setminus \{r\}$, Thus, $B_0 = \{r\}$ and $B_{\infty} = V \setminus \{r\}$. The algorithm works in so-called epochs. Each epoch $k$ settles all vertices in bucket $B_k$. Initially, we set $k = 0$. Each epoch $k$ consists of several phases. In each of these phases any vertex in $B_k$ is considered active if its tentative distance changed in the previous phase or the current phase is the first one. The algorithm performs the relaxation operation along each edge incident to an active vertex, i.e., along all edges $\{u, v\}$ such that $u$ is active. Note that in each phase vertices can move to different buckets, according to
their (possibly updated) tentative distances. The epoch is finished once there are no more active vertices in $B_k$. In this case, $k$ is incremented until $B_k$ is non-empty. If there is no non-empty bucket $B_s$ with $s > k$, apart from $B_\infty$, then the algorithm terminates. For a pseudo-code description of the above see Algorithm 1. The algorithm obtains an SSSP instance $I = (V, E, c, r)$ where $V$ is the vertex set, $E$ the edge set, $c$ the edge costs, and $r$ the root node. For simplicity, we only compute the shortest distances, but the algorithm can be easily extended to include ancestor information in order to also provide the SSSP tree.

Meyer and Sanders [2003] also suggest an improvement of the algorithm, which partitions the edges into two sets. The light edges consist of all $e \in E$ such that $c(e) \leq \delta$. The heavy edges consist of the remaining ones. Each epoch $k$ is changed as follows. In all phases of the epoch only light edges are relaxed. Once no vertices in $B_k$ are active anymore, the algorithm relaxes along all heavy edges $\{u, v\}$ with $u \in B_k$. In this way, one can avoid redundant updates of the tentative distances.

A small further improvement—which is mostly useful for distributed-memory implementations, which have high communication costs—is suggested by Chakaravarthy et al. [2017]. Consider epoch $k$. In each phase, the relaxation operation is only performed along light edges $\{u, v\}$ such that $d(u) + c(\{u, v\}) < (k+1)\Delta$, i.e., if the updated vertex would be moved into the current bucket $B_k$. Once the epoch is finished, the relaxation operation is performed along all edges $\{u, v\}$ such that $u \in B_k$ and $d(u) + c(\{u, v\}) \geq (k+1)\Delta$.

**Hybridization**  Another hybridization, which is especially efficient for scale-free graphs, can be obtained by switching from Delta-stepping to Bellman-Ford in the course of the algorithm. This approach is for example suggested by Chakaravarthy et al. [2017]. The Delta-stepping algorithm usually requires fewer relaxation operations, but more iterations than Bellman-Ford. However, in scale-free graphs most relaxation operations are performed in the first few epochs. The reason for this behaviour is that vertices with higher degree usually have smaller shortest distances and get settled in early epochs. In contrast, the vertices with lower degree usually have larger shortest distances and get settled only later on. Thus, a natural idea is to switch to Bellman-Ford later on to reduce the number of epochs. The crucial decision to be taken is when to change to Bellman-Ford. Chakaravarthy et al. [2017] suggest a simple heuristic that tracks the number of newly settled vertices in each epoch and changes to Bellman-Ford once a local maxima has been observed.

## 3 Distributed parallelization

The distribution of the graph data plays a pivotal role in the design of any parallel SSSP algorithm for distributed-memory. For large-scale graph processing, the arguably two most prominent distribution patterns are 1D partitioning, see e.g. Checconi and Petrini [2014] and 2D partitioning, see e.g. Ueno et al. [2016].

1D partitioning assigns each vertex to a specific parallel process. Additionally, all edges incident to a vertex are assigned to the same process. One key advantage of 1D partitioning is its simplicity. On the downside, it can lead to significant load imbalances, especially for graphs that have a very different
distribution of vertex degrees (e.g., scale-free graphs). Additionally, processing any edge \( \{u, v\} \) usually involves communication between two different processes, namely the owner process of \( u \) and the owner process of \( v \). Overall, an extensive all-to-all communication is required when the entire graph is processed.

2D partitioning distributes submatrices of the adjacency matrix of graph \( G \) among the processes. Given numbers \( R, C \in 1, 2, ..., |V| \), the adjacency matrix of \( G \) is partitioned into \( R \times C \) submatrices, as illustrated in Figure 1. Each of the submatrices is assigned to a single parallel process. One advantage of the 2D partition is a generally more even distribution of the graph data on the parallel processes. Additionally, one can avoid the expensive all-to-all communication, as will be demonstrated in the following.

### 3.1 2D-distributed SSSP

The predominant partitioning for distributed-parallel SSSP has been the 1D partition, see Chakaravarthy et al. [2017] for the state of the art. In contrast, we describe a 2D SSSP parallelization in the following. Consider again the partition of the adjacency matrix into \( R \times C \) submatrices illustrated in Figure 1. The parallel processes are virtually arranged in a corresponding \( R \times C \) matrix \( P \), where each entry (process) is referred to as \( P(i, j) \) for \( i \in \{1, 2, ..., R\}, j \in \{1, 2, ..., C\} \). For example, the submatrix \( A_{1,2} \) in Figure 1 is assigned to the parallel process \( P(1,2) \). We write \( P( :, j) \) for all the processors in the \( j \)-th column of \( P \), and \( P(i, :) \) for all the processors in the \( i \)-th row of \( P \).

The implementation of the 2D distribution for this article is based on the parallel BFS implementation by Ueno et al. [2016]. For example, we also use their bitmap-based sparse matrix storage. Algorithm 2 provides the pseudo-code of our 2D SSSP algorithm. As before, we do not include any improvements, such as heavy/light edges, for simplicity. We assume that the buckets are distributed among the parallel processes and each process stores only the vertices that it owns. Allgatherv() and alltoallv() are the standard MPI collectives. Importantly, communication is only performed along the processor row and columns. Recall that each process \( P(i, j) \) stores the adjacency sub-matrix \( A_{ij} \).
Algorithm 2: Basic 2D-parallel delta-stepping

Data: SSSP instance $I = (V, E, c, r)$
Result: Shortest path distances $d(v)$ for each $v \in V$

1. $\text{foreach } v \in V \setminus \{r\} \text{ do } d(v) := v$
2. $\text{foreach } k = 1, 2, 3, \ldots \text{ do } B_k := \emptyset$
3. $d(r) := 0$
4. $B_0 := \{r\}$
5. $B_{\infty} := V \setminus \{r\}$
6. $k := 0$
7. $\text{while } k < \infty \text{ do}$
   8. $S := \{(u, d(u)) : u \in B_k\}$
   9. $\text{transpose}(S)$
   10. $\text{allgatherv}(S, P(i,:))$
   11. $\text{while } S \neq \emptyset \text{ do}$
       12. $S' := \emptyset$
       13. $\text{foreach } \{u, v\} \in A_{ij} \text{ with } (u, d_u) \in S \text{ do}$
           14. $S' := S' \cup \{(v, d_u + c(\{u, v\}))\}$
       15. \text{end}
       16. $S := S'$
       17. $\text{alltoallv}(S, P(:j))$
       18. $\text{transpose}(S)$
       19. $\text{foreach } (v, dv) \in S \text{ do}$
           20. $q := \left\lfloor \frac{dv}{\Delta} \right\rfloor$
           21. $d(v) := \min \{d(v), dv\}$
           22. $q' := \left\lfloor \frac{d(v)}{\Delta} \right\rfloor$
           23. $\text{if } q' < q \text{ then}$
               24. $B_q := B_q \setminus \{v\}$
               25. $B_{q'} := B_{q'} \cup \{v\}$
           \text{end}
       \text{end}
   \text{end}
   28. $k := \min\{q > k : B_q \neq \emptyset \lor q = \infty\}$
\text{end}

3.2 Improvements

Several improvements of Algorithm 2 are done in our implementation. First, those mentioned in Section 2, such as using heavy/light edges. Additionally, we can get rid of the \text{transpose} operation, by using an adjacency matrix partition from Yoo et al. [2005], see Figure 2. In this way, each parallel processor owns already the correct vertices, and no \text{transpose} is needed.

Additionally, we use a modification of the direction-optimization suggested in Chakaravarthy et al. [2017] (we cannot use the original version, because it requires point-to-point communication between the vertex owners for each edge). Additionally, we use several smaller improvements such as filtering the set $S$ (by using a hashing approach) for duplicates before doing the all-to-all operation.
4 Experimental results

This section reports on two large-scale runs of our implementation. One was on the Fugaku supercomputer, which is installed at the RIKEN Center for Computational Science in Japan, the other one on the Lise supercomputer, which is installed at Zuse Institute Berlin.

Fugaku consists of 158,976 compute nodes with a total of 7,630,848 cores. The majority of the compute nodes have 32 GB memory and consist of Fujitsu A64FX CPUs with a clock speed of 2.2 GHz each. Lise consists of 1,270 compute nodes with a total of 121,920 compute cores. The majority of the compute nodes have 384 GB memory, and consist of 48 Intel Cascade Lake Platinum 9242 CPUs with a clock speed of 2.3 GHz.

In Table 1 we give the results of our implementation on both Fugaku and Lise. Due to availability restrictions we were only able to use half of the compute nodes of Fugaku. In column four we report the scale value for the Graph500 graph generator. For scale $s$ the generated graph has $2^s$ vertices and $16 \times 2^s$ edges. Thus, the graph generated on Fugaku has roughly $8.8 \times 10^{12}$ edges. Column five reports the obtained giga TEPS (GTEPS). Column 6 reports the official ranking in the Graph500 SSSP benchmark of June 2022.

<table>
<thead>
<tr>
<th>Machine</th>
<th># compute nodes</th>
<th># compute cores</th>
<th>scale</th>
<th>GTEPS</th>
<th>Graph500 ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fugaku</td>
<td>82,944</td>
<td>3,981,312</td>
<td>39</td>
<td>2,126</td>
<td>2</td>
</tr>
<tr>
<td>Lise</td>
<td>1,270</td>
<td>121,920</td>
<td>38</td>
<td>198</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 1: Results of runs for the Graph500 SSSP benchmark (June 2022).

Figure 3 illustrates the TEPS distribution of the TOP 10 participants of
Figure 3: TEPS distribution of the top 10 positions of the June 2022 Graph500 benchmark.

Figure 4: Graph size distribution of the top 10 positions of the June 2022 Graph500 benchmark.

the Graph500 benchmark. Similarly, Figure 4 shows the graph sizes used by the TOP 10 participants. It can be seen that our submissions have the largest (Fugaku) and second-largest (Lise) graph size among the participants. Indeed, also beyond the TOP 10, no graph scale larger than 36 is used. This result demonstrates the scalability of our approach with respect to the graph size.

5 Acknowledgements

We would like to thank the employees of RIKEN for their support for executing the benchmark on the Fugaku supercomputer. Likewise, we could like to thank the employees of the supercomputer division of Zuse Institute Berlin for helping us execute the benchmark on Lise. We would also like to thank Matthias Läuter for generating two of the figures used in this paper.

References


