Scalable Single-Source Shortest Path Algorithms on Distributed Memory Systems

Thap Panitanarak(✉)

Department of Mathematics and Computer Science, Chulalongkorn University, Patumwan 10330, Bangkok, Thailand
Thap.p@chula.ac.th

Abstract. Single-source shortest path (SSSP) is a well-known graph computation that has been studied for more than half a century. It is one of the most common graph analytical analysis in many research areas such as networks, communication, transportation, electronics and so on. In this paper, we propose scalable SSSP algorithms for distributed memory systems. Our algorithms are based on a Δ-stepping algorithm with the use of a two dimensional (2D) graph layout as an underlying graph data structure to reduce communication overhead and improve load balancing. The detailed evaluation of the algorithms on various large-scale real-world graphs is also included. Our experiments show that the algorithm with the 2D graph layout delivers up to three times the performance (in TEPS), and uses only one-fifth of the communication time of the algorithm with a one dimensional layout.

Keywords: SSSP · Parallel SSSP · Parallel algorithm · Graph algorithm

1 Introduction

With the advance of online social networks, World Wide Web, e-commerce and electronic communication in the last several years, data relating to these areas has become exponentially larger day by day. This data is usually analyzed in a form of graphs modeling relations among data entities. However, processing these graphs is challenging not only from a tremendous size of the graphs that is usually in terms of billions of edges, but also from graph characteristics such as sparsity, irregularity and scale-free degree distributions that are difficult to manage.

Large-scale graphs are commonly stored and processed across multiple machines or in distributed environments due to a limited capability of a single machine. However, current graph analyzing tools which have been optimized and used on sequential systems cannot directly be used on these distributed systems without scalability issues. Thus, novel graph processing and analysis are required, and parallel graph computations are mandatory to be able to handle these large-scale graphs efficiently.

Single-source shortest path (SSSP) is a well-known graph computation that has been studied for more than half a century. It is one of the most common graph analytical analysis for many graph applications such as networks, communication, transportation, electronics and so on. There are many SSSP algorithms that have been proposed such as a well-known Dijkstra's algorithm [9] and a Bellman-Ford algorithm

B. W. Yap et al. (Eds.): SCDS 2018, CCIS 937, pp. 19–33, 2019.
https://doi.org/10.1007/978-981-13-3441-2_2

[3, 10]. However, these algorithms are designed for serial machines, and do not efficiently work on parallel environments. As a result, many researchers have studied and proposed parallel SSSP algorithms or implemented SSSP as parts of their parallel graph frameworks. Some well-known graph frameworks include the Parallel Boost Graph Libray [14], GraphLab [16], PowerGraph [12], Galois [11] and ScaleGraph [8]. More recent frameworks have been proposed based on Hadoop sytems [26] such as Cyclops [6], GraphX [27] and Mizan [15]. For standalone implementations of SSSP, most recent implementations usually are for GPU parallel systems such as [7, 25, 28]. However, high performance GPU architectures are still not widely available and they also require fast CPUs to speed up the overall performance. Some SSSP implementations on shared memory systems include [17, 20, 21].

In this paper, we focus on designing and implementing efficient SSSP algorithms for distributed memory systems. While the architectures are not relatively new, there are few efficient SSSP implementations for this type of architectures. We aware of the recent SSSP study of Chakaravarthy et al. [5] that is proposed for massively parallel systems, IBM Blue Gene/Q (Mira). Their SSSP implementations have applied various optimizations and techniques to achieve very good performance such as direction optimization (or a push-pull approach), pruning, vertex cut and hybridization. However, most techniques are specifically for SSSP algorithms and can only be applied to a limited variety of graph algorithms. In our case of SSSP implementations, most of our techniques are more flexible and can be extended to many graph algorithms, while still achieving good performance. Our main contributions include:

- Novel SSSP algorithms that combine advantages of various well-known SSSP algorithms.
- Utilization of a two dimensional graph layout to reduce communication overhead and improve load balancing of SSSP algorithms.
- Distributed cache-like optimization that filters out unnecessary SSSP updates and communication to further increase the overall performance of the algorithms.
- Detailed evaluation of the SSSP algorithms on various large-scale graphs.

2 Single-Source Shortest Path Algorithms

Let $G = (V, E, w)$ be a weighted, undirected graph with $n = |V|$ vertices, $m = |E|$ edges, and integer weights $w(e) > 0$ for all $e \in E$. Define $s \in V$ called a source vertex, and $d(v)$ to be a tentative distance from s to $v \in V$ (initially set to ∞). The single source shortest path (SSSP) problem is to find $\delta(v) \leq d(v)$ for all $v \in V$. Define $d(s) = 0$, and $d(v) = \infty$ for all v that are not reachable from s.

Relaxation is an operation to update $d(v)$ using in many well-known SSSP algorithms such as Dijkstra's algorithm and Bellman-Ford. The operation updates $d(v)$ using a previously updated $d(u)$ for each $(u, v) \in E$. An edge relaxation of (u, v) is defined as $d(v) = \min\{d(v), d(u) + w(u, v)\}$. A vertex relaxation of u is a set of edge relaxations of all edges of u. Thus, a variation of SSSP algorithms is generally based on the way the relaxation taken place.

The classical Dijkstra's algorithm relaxes vertices in an order starting from a vertex with the lowest tentative distance first (starting with s). After all edges of that vertex are relaxed, the vertex is marked as settled that is the distance to such vertex is the shortest possible. To keep track of a relaxing order of all active vertices v (or vertices that have been updated and wait to be relaxed), the algorithm uses a priority queue that orders active vertices based on their $d(v)$. A vertex is added to the queue only if it is visited for the first time. The algorithm terminates when the queue is empty. Another variant of Dijkstra's algorithm for integer weight graphs that is suited for parallel implementation is called Dial's algorithm. It uses a bucket data structure instead of a priority queue to avoid the overhead from maintaining the queue while still giving the same work performance as Dijkstra's algorithm. Each bucket has a unit size, and holds all active vertices that have the same tentative distance as a bucket number. The algorithm works on buckets in order starting from the lowest to the highest bucket numbers. Any vertex in each bucket has an equal priority, and can be processed simultaneously. Thus, the algorithm concurrency is from the present of these buckets.

Another well-known SSSP algorithm, Bellman-Ford, allows vertices to be relaxed in any order. Thus, there is no guarantee if a vertex is settled after it has been once relaxed. Generally, the algorithm uses a first-in-first-out (FIFO) queue to maintain the vertex relaxation order since there is no actual priority of vertices. A vertex is added to the queue when its tentative distance is updated, and is removed from the queue after it is relaxed. Thus, any vertex can be added to the queue multiple times whenever its tentative distance is updated. The algorithm terminates when the queue is empty. Since the order of relaxation does not affect the correctness of the Bellman-Ford algorithm, it allows the algorithm to provide high concurrency from simultaneous relaxation.

While Dijkstra's algorithm yields the best work efficiency since each vertex is relaxed only once, it has very low algorithm concurrency. Only vertices that have the smallest distance can be relaxed at a time to preserve the algorithm correctness. In contrast, Bellman-Ford requires more works from (possibly) multiple relaxations of each vertex. However, it provides the best algorithm concurrency since any vertex in the queue can be relaxed at the same time. Thus, the algorithm allows simultaneously relaxations while the algorithm's correctness is still preserved.

The Δ-stepping algorithm [18] compromises between these two extremes by introducing an integer parameter $\Delta \geq 1$ to control the trade-off between work efficiency and concurrency. At any iteration $k \geq 0$, the Δ-stepping algorithm relaxes the active vertices that have tentative distances in $[k\Delta, (k+1)\Delta - 1]$. With $1 < \Delta < \infty$, the algorithm yields better concurrency than the Dijkstra's algorithm and lower work redundancy than the Bellman-Ford algorithm. To keep track of active vertices to be relaxed in each iteration, the algorithm uses a bucket data structure that puts vertices with the same distant ranges in the same bucket. The bucket k contains all vertices that have the tentative distance in the range $[k\Delta, (k+1)\Delta - 1]$. To make the algorithm more efficient, two processing phases are introduced in each iteration. When an edge is relaxed, it is possible that the updated distance of an adjacency vertex may fall into the current bucket, and it can cause cascading re-updates as in Bellman-Ford. To minimize these re-updates, edges of vertices in the current bucket with weights less than Δ (also called light edges) are relaxed first. This forces any re-insertion to the current bucket to happen earlier, and, thus, decreasing the number of re-updates. This phase is called a

light phase, and it can iterate multiple times until there is no more re-insertion, or the current bucket is empty. After that, all edges of vertices which are previously relaxed in the light phases with weights greater than Δ (also called heavy edges) are then relaxed. This phase is called a heavy phase. It only occurs once at the end of each iteration since, with edge weights greater than Δ, the adjacency vertices from updating tentative distances are guaranteed not to fall into the current bucket. The Δ-stepping algorithm can be viewed as a general case of SSSP algorithms with the relaxation approach. The algorithm with $\Delta = 1$ is equivalent to Dijkstra's algorithm, while the algorithm with $\Delta = \infty$ yields Bellman-Ford.

3 Novel Parallel SSSP Implementations

3.1 General Parallel SSSP for Distributed Memory Systems

We consider SSSP implementations in [19] which are based on a bulk-synchronous Δ-stepping algorithm for distributed memory sysyem. The algorithm composes of three main steps, a local discovery, an all-to-all exchange and a local update for both light and heavy phases. In the local discovery step, each processor looks up to all adjacencies v of its local vertices u in the current bucket, and generates corresponding tentative distances $dtv = d(u) + w(u, v)$ of those adjacencies. Note that, in the light phase, only adjacencies with light edges are considered, while, in the heavy phase, only adjacencies with heavy edges are processed. For each (u, v), a pair (v, dtv) is generated, and stored in a queue called QRequest. The all-to-all exchange step distributes these pairs in QRequest to make them local to processors so that each processor can use these information to update a local tentative distance list in the local update step. An edge relaxation is part of the local update step that invokes updating vertex tentative distances, and adding/removing vertices to/from buckets based on their current distances.

3.2 Parallel SSSP with 2D Graph Layout

A two dimensional (2D) graph layout had been previously studied in [4] for breadth-first search. This approach partitions an adjacency matrix of graph vertices into grid blocks instead of a traditional row partition or as one dimensional (1D) graph layout. The 2D layout reduces communication space and also provides better edge distributions of a distributed graph than the 1D layout as any dense row of the high degree vertices can now be distributed across multiple processors instead of only one processor as in the 1D layout.

To apply the 2D graph layout for the Δ-stepping algorithm, each of the three steps needs to be modified according to the changes in the vertex and edge distributions. While the vertices are distributed in similar manner as in the 1D graph layout, edges are now distributed differently. Previously in the 1D layout, all edges of local vertices are assigned to one processor. However, with the 2D layout, these edges are now

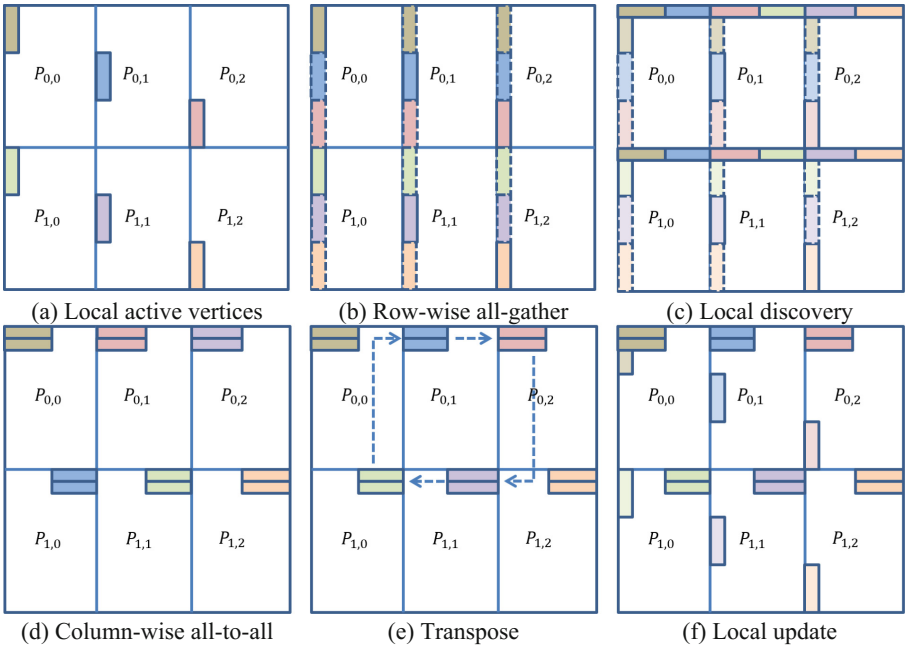


Fig. 1. The main SSSP operations with the 2D layout. (a) Each color bar shows the vertex information for active vertices owned to each processor P_{ij}. (b) The row-wise all-gather communication gathers all information of actives vertices among the same processor rows to all processors in the same row. (c) Each processor uses the information to update the vertex adjacencies. (d,e) The column-wise all-to-all and transpose communications group the information of the updated vertices owned by the same processors and send this information to the owner processors. (f) Each processor uses the received information to update its local vertex information (Color figure online).

distributed among row processors that have the same row number. Figure 1(a) illustrates the partitioning of vertices and edges for the 2D layout.

In the local discovery step, there is no need to modify the actual routine. The only work that needs to be done is merging all current buckets along the processor rows by using a row-wise all-gather communication. The reason is that the edge information (such as edge weights and adjacencies) of local vertices owned by each processor is now distributed among the processor rows. Thus, each processor with the same row number is required to know all the active vertices in the current bucket of their neighbor processor rows before the discovery step can take place. After the current buckets are merged (see Fig. 1(b)), each processor can now simultaneously work on generating pairs (v, dtv) of its local active vertices (see Fig. 1(c)).

In the all-to-all exchange step, the purpose of this step is to distribute the generated pairs (v, dtv) to the processors that are responsible to maintain the information relating to vertices v. In our implementation, we use two sub-communications, a column-wise all-to-all exchange and a send-receive transposition. The column-wise all-to-all communication puts all information pairs of vertices owned by the same owner onto one processor. Figure 1(d) shows a result of this all-to-all exchange. After that, each processor sends and receives these pair lists to the actual owner processors. The latter communication can be viewed as a matrix transposition as shown in Fig. 1(e).

In the local update step, there is no change within the step itself, but only in the data structure of the buckets. Instead of only storing vertices in buckets, the algorithm needs to store both vertices and their current tentative distances so that each processor knows the distance information without initiating any other communication. Figure 1(f) illustrates the local update step. Since all pairs (d, dtv) are local, each processor can update the tentative distances of their local vertices simultaneously.

The complete SSSP algorithm with the 2D graph layout is shown in Algorithm 1. The algorithm initialization shows in the first 10 lines. The algorithm checks for the termination in line 11. The light and heavy phases are shown in lines 12–25 and lines 26-35, respectively. The termination checking for the light phases of a current bucket is in line 12. The local discovery, all-to-all exchange and local update steps of each light phase are shown in lines 13–19, 20 and 22, respectively. Similarly for each heavy phase, its local discovery, all-to-all exchange and local update steps are shown in lines 26–31, 32 and 34, respectively. Algorithm 2 shows the relaxation procedure used in Algorithm 1.

3.3 Other Optimizations

To further improve the algorithm performance, we apply other three optimizations, a cache-like optimization, a heuristic Δ increment and a direction optimization. The detailed explanation is as follows.

Cache-like optimization: We maintain a tentative distance list of every unique adjacency of the local vertices as a local cache. This list holds the recent values of tentative distances of all adjacencies of local vertices. Every time a new tentative distance is generated (during the discovery step), this newly generated distance is compared to the local copy in the list. If the new distance is shorter, it will be processed in the regular manner by adding the generated pair to the QRequest, and the local copy in the list is updated to this value. However, if the new distance is longer, it will be discarded since the remote processors will eventually discard this request during the relaxation anyway. Thus, with a small trade-off of additional data structures and computations, this approach can significantly avoid unnecessary work that involves both communication and computation in the later steps.

Algorithm 1 Distributed SSSP with 2D Graph Layout

```
1:  for each u do
2:    d[u] ← ∞
3:  end for
4:    current ← 0
5:  if onwer(s) = rank then
6:    d[s] ← 0
7:  end if
8:  if onwerRow(s) = rankRow then
9:    Bucket[current] ← Bucket[current] ∪ (s, 0)
10: end if
11: while Bucket ≠ ∅ do            //Globally check
12:   while Bucket[current] ≠ ∅ do //Globally check
13:     for each (u, du) ∈ Bucket[current] do
14:       for each (u, v) ∈ LightEdge do
15:         dtv ← du + w(u, v)
16:         QRequest ← QRequest ∪ (v, dtv)
17:       end for
18:       QHeavy ← QHeavy ∪ (u, du)
19:     end for
20:     Alltoallv(QRequest, row); Transpose(QRequest)
21:     for each (v, dtv) ∈ QRequest do
22:       Relax(v, dtv)
23:     end for
24:     Allgatherv(Bucket[current], col)
25:   end while
26:   for each (u, du) ∈ QHeavy do
27:     for each (u, v) ∈ HeavyEdge do
28:       dtv ← du + w(u, v)
29:       QRequest ← QRequest ∪ (v, dtv)
30:     end for
31:   end for
32:   Alltoallv(QRequest, row); Transpose(QRequest)
33:   for each (v, dtv) ∈ QRequest do
34:     Relax(v, dtv)
35:   end for
36:   current ← current + 1        //Move to next bucket
37:   Allgatherv(Bucket[current], col)
38: end while
```

Algorithm 2 $Relax(v, dtv)$

```
1: if d[v] > dtv then
2:     old ← d[v]/Δ; new ← dtv/Δ
3:     Bucket[old] ← Bucket[old] − (v, d[v])
4:     Bucket[new] ← Bucket[new] ∪ (v, dtv)
5:     d[v] ← dtv
6: end if
```

Heuristic Δ increment: The idea of this optimization is from the observation of the Δ-stepping algorithm that the algorithm provides a good performance in early iterations when Δ is small since it can avoid most of the redundant work in the light phases. Meanwhile, with a large Δ, the algorithm provides a good performance in later iterations since most of vertices are settled so that the portion of the redundant work is low. Thus, the benefit of the algorithm concurrency outweighs the redundancy. The algorithm with Δ that can be adjusted when needed can provide better performance. From this observation, instead of using a fix Δ value, we implement algorithms that starts with a small Δ until some thresholds are met, then, the Δ is increased (usually to ∞) to speed up the later iterations.

Direction-optimization: This optimization is a heuristic approach first introduced in [2] for breadth-first search (BFS). Conventional BFS usually proceeds in an top-down approach such that, in every iteration, the algorithm checks all adjacencies of each vertex in a frontier whether they are not yet visited, adds them to the frontier, and then marks them as visited. The algorithm terminates whenever there is no vertex in the frontier. We can see that the algorithm performance is highly based on processing vertices in this frontier. The more vertices in the frontier, the more work that needs to be done. From this observation, the bottom-up approach can come to play for efficiently processing of the frontier. The idea is that instead of proceeding BFS only using the top-down approach, it can be done in a reverse direction if the current frontier has more work than the work using the bottom-up approach. With a heuristic determination, the algorithm can alternately switch between top-down and bottom-up approaches to achieve an optimal performance. Since the discovery step in SSSP is done in similar manner as BFS, Chakaravarthy et al. [5] adapts a similar technique called a push-pull heuristic to their SSSP algorithms. The algorithms proceed with a push (similar to the top-down approach) by default during heavy phases. If a forward communication volume of the current bucket is greater than a request communication volume of aggregating of later buckets, the algorithms switch to a pull. This push-pull heuristic considerably improves an overall performance of the algorithm. The main reason of the improvement is because of the lower of the communication volume, thus, the consequent computation also decreases.

3.4 Summary of Implementations

In summary, we implement four SSSP algorithms:

1. **SP1a**: The SSSP algorithm based on Δ-stepping with the cache-like optimization
2. **SP1b**: The SP1a algorithm with the direction optimization
3. **SP2a**: The SP1a algorithm with the 2D graph layout
4. **SP2b**: The SP2a algorithm with the Δ increment heuristic

The main differences of each algorithm are the level of optimizations that additionally increases from SP#a to SP#b that is the SP#b algorithms are the SP#a algorithms with more optimizations, and from SP1x to SP2x that is the SP1x algorithms use the 1D layout while the SP2x algorithms use the 2D layout.

4 Performance Results and Analysis

4.1 Experimental Setup

Our experiments are run on a virtual cluster using StarCluster [24] with the MPICH2 complier version 1.4.1 on top of Amazon Web Service (AWS) Elastic Compute Cloud (EC2) [1]. We use 32 instances of AWS EC2 m3.2xlarge. Each instance consists of 8 cores of high frequency Intel Xeon E5-2670 v2 (Ivy Bridge) processors with 30 GB of memory. The graphs that we use in our experiments are listed in Table 1. The graph500 is a synthetic graph generated from the Graph500 reference implementation [13]. The graph generator is based on the RMAT random graph model with the parameters similar to those use in the default Graph500 benchmark. In this experiment, we use the graph scale of 27 with edge factor of 16 that is the graphs are generated with 2^{27} vertices with an average of 16 degrees for each vertex. The other six graphs are real-world graphs that are obtained from Stanford Large Network Dataset Collection (SNAP) [22], and the University of Florida Sparse Matrix Collection [23]. The edge weights of all graphs are randomly, uniformly generated between 1 and 512.

Table 1. The list of graphs used in the experiments

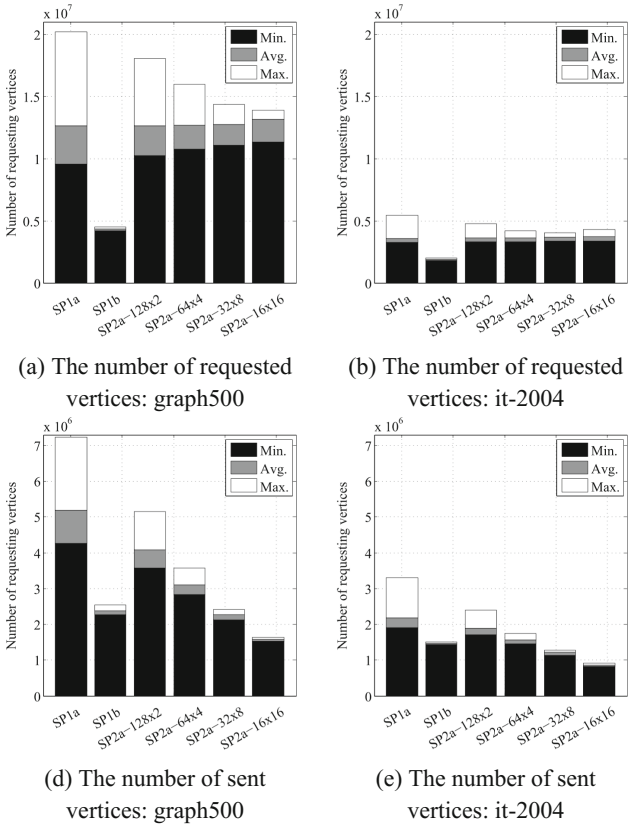
Graph	Number of vertices (millions)	Number of edges (billions)	Reference
graph500	134	2.1	[13]
it-2004	41	1.1	[23]
sk-2005	50	1.9	[23]
friendster	65	1.8	[22]
orkut	3	0.12	[22]
livejournal	4	0.07	[22]

We fix the value of Δ to 32 for all algorithms. Please note that this value might not be the optimal value in all test cases, but, in our initial experiments on the systems, it gives good performance in most cases. To get the optimal performance in all cases is

not practical since Δ needs to be changed accordingly to the systems such as CPU, network bandwidth and latency, and numbers of graph partitions. For more discussion about the Δ value, please see [19].

4.2 Algorithm and Communication Cost Analysis

For SSSP algorithms with the 2D layout, when the number of columns increases, the all-to-all communication overhead also decreases, and the edge distribution is more balanced. Consider processing a graph with n vertices and m edges on $p = r \times c$ processors. The all-to-all and all-gather communication spaces are usually proportional to r and c, respectively. In other words, the maximum number of messages for each all-to-all communication is proportional to m/c while the maximum number of messages for each all-gather communication is proportional to n/r. In each communication phase, processor $P_{i,j}$ requires to interact with processors $P_{k,j}$ for the all-to-all communication where $0 \leq k < r$, and with processors $P_{i,l}$ for the all-gather communication



(a) The number of requested vertices: graph500

(b) The number of requested vertices: it-2004

(d) The number of sent vertices: graph500

(e) The number of sent vertices: it-2004

Fig. 2. The numbers of (a,b) requested and (c,d) sent vertices during the highest relaxation phase of the SP2a algorithm on graph500 and it-2004 using different combinations of processor rows and columns on 256 MPI tasks.

where $0 \leq l < c$. For instance, by setting $r = 1$ and $c = p$, the algorithms do not need any all-to-all communication, but the all-gather communication now requires all processors to participate.

During the SSSP process on scale-free graphs, there are usually a few phases of the algorithms that consume most of the computation and communication times due to the present of few vertices with high degrees. The Fig. 2(a,b) and (c,d) show the average, minimum and maximum vertices to be requested and sent, respectively, for relaxations during the phase that consumes the most time of the algorithms SP1a, SP1b and SP2a on graph500 and it-2004 with 256 MPI tasks. Note that we use the abbreviation SP2a-R $\times$ C for the SP2a algorithm with R and C processor rows and columns, respectively. For example, SP2a-64 $\times$ 4 is the SP2a algorithm with 64 row and 4 column processors (which are 256 processors in total). The improvement of load balancing of the requested vertices for relaxations can easily be seen in Fig. 2(a,b) as the minimum and maximum numbers of the vertices decrease on both graphs from SP1a to SP1b and SP1a to SP2a. The improvement from SP1a to SP1b is significant as the optimization is specifically implemented for reducing the computation and communication overheads during the high-requested phases. On the other hand, SP2a still processes on the same number of vertices, but with lower communication space and better load balancing. Not only the load balancing of the communication improves, but the numbers of (average) messages among inter-processors also reduce as we can see in Fig. 2(c,d). However, there are some limitations of both SP1b and SP2a. For SP1b, the push-pull heuristic

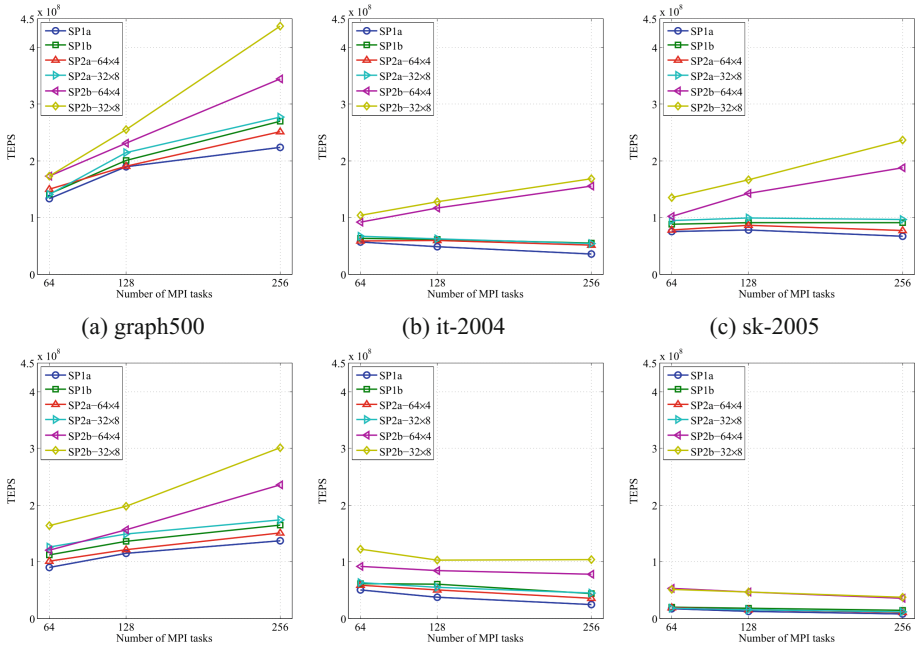


Fig. 3. The performance (in TEPS) of SSSP algorithms up to 256 MPI tasks

may not trigger in some phases that the costs of push and pull approaches are slightly different. In contrast, for SP2a, although increasing numbers of columns improves load balancing and decreases the all-to-all communication in every phase, it also increases the all-gather communication proportionally. There is no specific number of columns that gives the best performance of the algorithms since it depends on various factors such as the number of processors, the size of the graph and other system specifications.

4.3 Benefits of 2D SSSP Algorithms

Figure 3 shows the algorithm performance in terms of traversed edges per second (TEPS) on Amazon EC2 up to 256 MPI tasks. Although SP1b can significantly reduce computation and communication during the high-requested phases, its overall performance is similar to SP2a. The SP2b algorithm gives the best performance in all cases, and it also gives the best scaling when the number of processors increases. The peak performance of SP2b-32 $\times$ 8 is approximately 0.45 GTEPS that can be observed on graph500 with 256 MPI tasks, which is approximately 2x faster than the performance of SP1a on the same setup. The SP2b algorithm also shows good scaling on large graphs such as graph500, it-2004, sk-2005 and friendster.

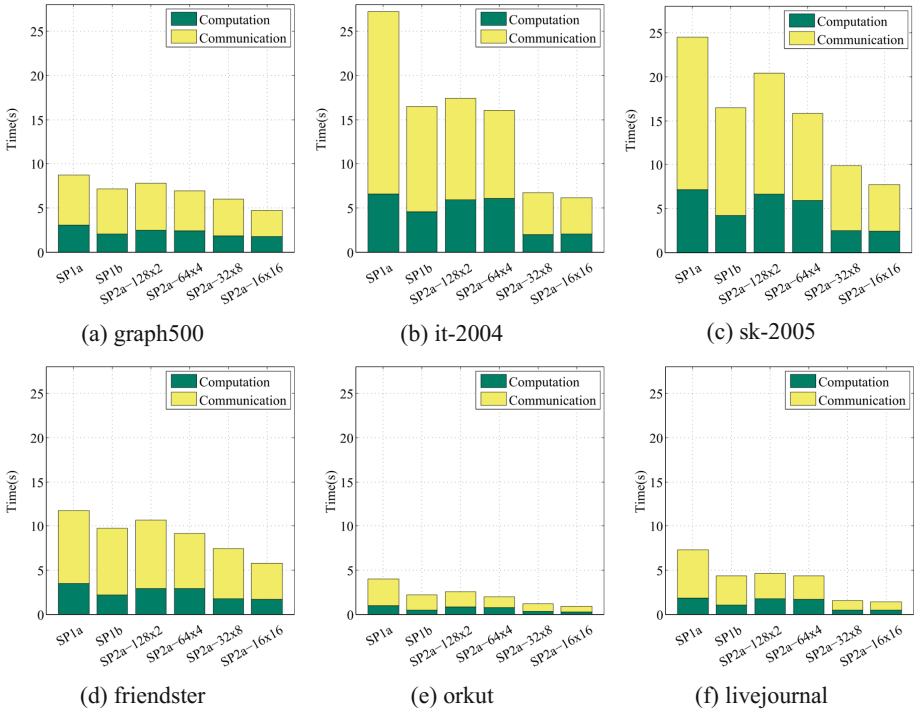


Fig. 4. The communication and computation times of SSSP algorithms on 256 MPI tasks

4.4 Communication Cost Analysis

Figure 4 shows the breakdown execution time of total computation and communication of each algorithm. More than half of the time for all algorithms is spent on communication as the networks of Amazon EC2 is not optimized for high performance computation. The improvement of SP1b over SP1a is from the reduction of computation overhead as the number of processing vertices in some phases are reduced. On the other hand, SP2a provides lower communication overhead over SP1a as the communication space is decreased from the use of the 2D layout. The SP2b algorithm further improves the overall performance by introducing more concurrency in the later phases resulting in lower both communication and communication overhead during the SSSP runs. Figure 5 shows the breakdown communication time of all algorithms. We can see that when the number of processor rows increases, it decreases the all-to-all communication, and slightly increases the all-gather and transpose communications. In all cases, SP2b shows the least communication overhead with up to 10x faster for the all-to-all communication and up to 5x faster for the total communication.

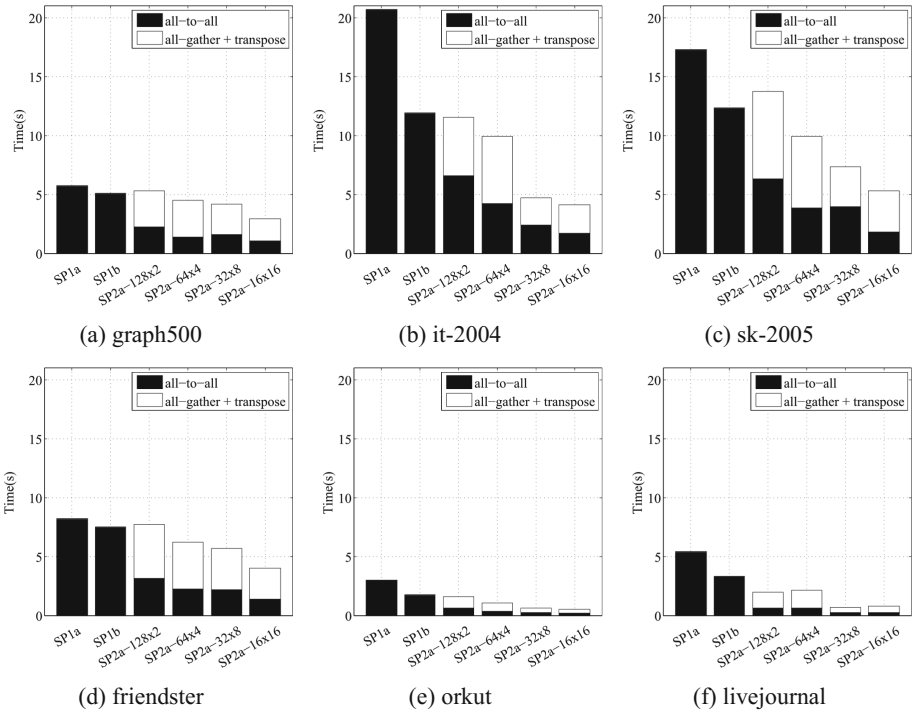


Fig. 5. Communication breakdown of SSSP algorithms on 256 MPI tasks

5 Conclusion and Future Work

We propose scalable SSSP algorithms based on the Δ-stepping algorithm. Our algorithms reduce both communication and computation overhead from the utilization of the 2D graph layout, the cache-like optimization and the Δ increment heuristic. The 2D layout improves the algorithm performance by decreasing the communication space, thus, reducing overall communication overhead. Furthermore, the layout also improves the distributed graph load balancing, especially, on scale-free graphs. The cached-like optimization avoid unnecessary workloads for both communication and communication by filtering out all update requests that are known to be discarded. Finally, by increasing the Δ values during the algorithms progress, we can improve the concurrency of the algorithms in the later iterations.

Currently, our algorithm is based on the bulk-synchronous processing for distributed memory systems. We plan to extend our algorithms to also utilize the shared memory parallel processing that can further reduce the inter-processing communication of the algorithms.

Acknowledgement. The author would like to thank Dr. Kamesh Madduri, an associate professor at Pennsylvania State University, USA, for the inspiration and kind support.

References

1. Amazon Web Services: Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/. Accessed 15 July 2018
2. Beamer, S., Asanovi´c, K., Patterson, D.: Direction-optimizing breadth-first search. Sci. Prog. **21**(3–4), 137–148 (2013)
3. Bellman, R.: On a routing problem. Q. Appl. Math. **16**, 87–90 (1958)
4. Buluc, A., Madduri, K.: Parallel breadth-first search on distributed memory systems. In: Proceedings of High Performance Computing, Networking, Storage and Analysis (SC) (2011)
5. Chakaravarthy, V.T., Checconi, F., Petrini, F., Sabharwal, Y.: Scalable single source shortest path algorithms for massively parallel systems. In: Proceedings of IEEE 28th International Parallel and Distributed Processing Symposium, pp. 889–901 May 2014
6. Chen, R., Ding, X., Wang, P., Chen, H., Zang, B., Guan, H.: Computation and communication efficient graph processing with distributed immutable view. In: Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, pp. 215–226. ACM (2014)
7. Davidson, A.A., Baxter, S., Garland, M., Owens, J.D.: Work-efficient parallel GPU methods for single-source shortest paths. In: International Parallel and Distributed Processing Symposium, vol. 28 (2014)
8. Dayarathna, M., Houngkaew, C., Suzumura, T.: Introducing ScaleGraph: an X10 library for billion scale graph analytics. In: Proceedings of the 2012 ACM SIGPLAN X10 Workshop, p. 6. ACM (2012)
9. Dijkstra, E.W.: A note on two problems in connection with graphs. Numer. Math. **1**(1), 269–271 (1959)
10. Ford, L.A.: Network flow theory. Technical. report P-923, The Rand Corporation (1956)
11. Galois. http://iss.ices.utexas.edu/?p=projects/galois. Accessed 15 July 2018

12. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: distributed graph-parallel computation on natural graphs. In: OSDI, vol. 12, p. 2 (2012)
13. The Graph 500. http://www.graph500.org. Accessed 15 July 2018
14. Gregor, D., Lumsdaine, A.: The Parallel BGL: a generic library for distributed graph computations. Parallel Object-Oriented Sci. Comput. (POOSC) **2**, 1–18 (2005)
15. Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D., Kalnis, P.: Mizan: a system for dynamic load balancing in large-scale graph processing. In: Proceedings of the 8th ACM European Conference on Computer Systems, pp. 169–182. ACM (2013)
16. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (2012)
17. Madduri, K., Bader, D.A., Berry, J.W., Crobak, J.R.: An experimental study of a parallel shortest path algorithm for solving large-scale graph instances, Chap. 2, pp. 23–35 (2007)
18. Meyer, U., Sanders, P.: Δ-stepping: a parallelizable shortest path algorithm. J. Algorithms **49** (1), 114–152 (2003)
19. Panitanarak, T., Madduri, K.: Performance analysis of single-source shortest path algorithms on distributed-memory systems. In: SIAM Workshop on Combinatorial Scientific Computing (CSC), p. 60. Citeseer (2014)
20. Prabhakaran, V., Wu, M., Weng, X., McSherry, F., Zhou, L., Haridasan, M.: Managing large graphs on multi-cores with graph awareness. In: Proceedings of USENIX Annual Technical Conference (ATC) (2012)
21. Shun, J., Blelloch, G.E.: Ligra: a lightweight graph processing framework for shared memory. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2013 pp. 135–146 (2013)
22. SNAP: Stanford Network Analysis Project. https://snap.stanford.edu/data/. Accessed 15 July 2018
23. The University of Florida Sparse Matrix Collection. https://www.cise.ufl.edu/research/sparse/matrices/. Accessed 15 July 2018
24. StarCluster. http://star.mit.edu/cluster/. Accessed 15 July 2018
25. Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A. Owens, J.D.: Gunrock: a high-performance graph processing library on the GPU. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 265–266. PPoPP 2015 (2015)
26. White, T.: Hadoop: The Definitive Guide. O'Reilly Media Inc, Sebastopol (2012)
27. Xin, R.S., Gonzalez, J.E., Franklin, M.J. Stoica, I.: Graphx: A resilient distributed graph system on spark. In: First International Workshop on Graph Data Management Experiences and Systems, p. 2. ACM (2013)
28. Zhong, J., He, B.: Medusa: simplified graph processing on GPUs. Parallel Distrib. Syst. IEEE Trans. **25**(6), 1543–1552 (2014)