Fast All-Pairs Shortest Paths Algorithm in Large Sparse Graph

Shaofeng Yang[1, 2], Xiandong Liu[1, 2], Yunting Wang[1], Xin He[1], Guangming tan[1, 2],
{yangshaofeng,liuxiandong}@ncic.ac.cn{wangyunting,hexin2016,tgm}@ict.ac.cn
[1] State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2]University of Chinese Academy of Sciences, Beijing, China

ABSTRACT

Finding the All-Pairs Shortest Paths (APSP) in a graph is the key for various domains. Motivated by the graphs are sparse in most real-world applications, we store the whole graph as a compressed storage format in each process of the distributed computing clusters and combine the Floyd algorithm with the Dijkstra algorithm to solve the APSP problem in this work, which leads to the novel Fast APSP algorithm. In contrast to the state-of-the-art Part APSP algorithm, our algorithm adds some memory overhead to store the original sparse graph and uses local Floyd and global Dijkstra algorithms simultaneously. The payoff is the circumvention of expensive global communication, reducing one local FW operation, simplifying the Minplus function, and making its data access continuous. Furthermore, we propose a parallel framework to solve the problem of mismatch between the number of GPUs and the number of divisible blocks of a graph. The Fast APSP algorithm exhibits an average speedup of 16.97x compared to the CPU Dijkstra algorithm, 7.09x compared to the GPU Dijkstra algorithm, 7.09x compared to the Part APSP algorithm, and 4.6x compared to the decentralized Part APSP algorithm. It also shows good scalability in our experiments. It takes about 12.45 minutes to solve the APSP problem for the graph with 11,548,845 vertices by engaging 2048 GPUs.

CCS CONCEPTS

• **Computing methodologies → Massively parallel algorithms**.

KEYWORDS

All-pairs shortest paths, High-Performance Computing, Distributed graph Algorithm

ACM Reference Format:
Shaofeng Yang[1, 2], Xiandong Liu[1, 2], Yunting Wang[1], Xin He[1], Guangming tan[1, 2], . 2023. Fast All-Pairs Shortest Paths Algorithm in Large Sparse Graph. In *2023 International Conference on Supercomputing (ICS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3577193.3593728

1 INTRODUCTION

The motivation of the All-Pairs Shortest Path (APSP) problem is to find the shortest path between arbitrary two vertices in a graph. The APSP problem plays a fundamental role in a tremendously wide range of applications, e.g. traffic road networks, biological information mining, and social networks [1, 21, 28]. To our knowledge, there exist two main families of algorithms to solve the APSP problem, one is to use Floyd-Warshall (FW) algorithm [26], and the other is to call the Single-Source Shortest Path (SSSP) algorithm multiple times [8].

The basic idea of the FW algorithm is the dynamic programming strategy. For a given graph of n vertices, the shortest distance between each pair of vertices is iteratively calculated. The advantage of the FW algorithm is that it can be represented by the matrix operations and thus the highly optimized implementations of the relevant subroutines on the modern hardware are applicable. On the other hand, the high time and space complexities, $O(n^3)$ and $O(n^2)$ respectively limit the applicability border of the FW algorithm. As far as we know, the largest graph which is handled by the FW algorithm consists of 6 million vertices [11] and it takes about 79.8 minutes by engaging 4096 computing nodes of 24,576 GPUs in the Summit supercomputer system.

The SSSP algorithm utilizes the Dijkstra [5] or the Bellman-Ford method [26] to calculate the shortest distance between the source vertex and the others in a given graph. The time complexity of the SSSP algorithm is $O(nlogn + m)$, where m denotes the number of edges. The time complexity of using the SSSP algorithm to solve the APSP problem is $O(n^2logn + nm)$, which is superior to the FW algorithm. But the SSSP algorithm suffers from poor parallelism due to the complex data structure and also cannot effectively use the features of modern computer architecture such as long SIMD vector units or cache. Therefore, it underutilizes modern high-performing computing systems. To improve the parallelism several attempts are made which include the delta-stepping algorithm proposed in [17] and the task level parallelization dealing with multiple SSSP tasks [23]. As far as we know, the largest graph conducted by the SSSP algorithm consists of about 1024K vertices and it takes around 10 hours by using 2 GPUs [19].

Based on the FW algorithm, Hristo Djidjev et al. proposed the Partitioned All-Pairs Shortest Path (Part APSP) algorithm [7]. The algorithm employs a divide-and-conquer approach (see Algorithm 6), which divides the input graph into multiple components and distributes them to different processes. Each component solves the APSP problem between its own vertices and then gets all the boundary vertices of other components through communication to solve the boundary vertices' APSP problem. Finally, all processes communicate their component's APSP results to each other and compute the shortest paths between all pairs of vertices in the input graph. The time complexity of the Part APSP algorithm on the planar graph is as low as $O(n^{2.25})$, and also can be efficiently parallelized on GPU. The largest graph instance described in the paper contains 1,957,027 vertices, and it cost 31 minutes to compute

ICS '23, June 21–23, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0056-9/23/06…$15.00
https://doi.org/10.1145/3577193.3593728

the shortest distances between all vertices using a 64-node cluster with 2 GPUs on each node.

Despite the time complexity of the part APSP algorithm is only $O(n^{2.25})$ and the algorithm is able to compute all the shortest distances of a planar graph with nearly 2 million vertices in 31 minutes. It is limited by their global communication, which cost $log_2(P)$ where P denotes the number of engaged processes. Due to the data dependency problem and centralized communication scheme, the Part APSP algorithm almost gained no benefit from using more than about 63 GPUs in its scalability experiment. Although the Decentralized Part APSP [6] uses a decentralized communication scheme to achieve better performance. The data dependency problem still has not been solved.

In this article, we propose a new algorithm, called the Fast APSP algorithm. In the algorithm, we also divide a graph into multiple components. Then, we combine both algorithms global SSSP and local FW to compute the APSP problem. To meet the needs of the global SSSP algorithm, our work stored the original sparse graph on every process. Since most graphs are very sparse, especially planar graphs, the memory space required to store them in a compressed format is moderate and affordable for modern computers. In our experiment, the largest graph germany-osm has 11,548,845 vertices and 24,738,362 edges. It's only 327MB using CSR format. The Fast APSP algorithm also maintains $O(n^{2.25})$ time complexity for the planar graph and has no communication except in the graph partitioning stage. In addition, in order to handle cases where the graph is not easily partitioned, we propose a flexible parallel framework to handle the mismatch between the number of processes and the number of graph partitions.

Specifically, we make the following contributions:

- We propose an algorithm named Fast APSP, which is based on graph partitioning and combines the FW and SSSP algorithms to solve the APSP problem of large graphs. Thus, the algorithm reduces one local FW calculation, simplifying the MIN-PLUS function, and making its data access continuous. And it has a low time complexity of $O(n^{2.25})$ for the planar graph.
- To eliminate global communication, in the parallel computing framework, we store the original graph in a compressed format in each process to compute the global SSSP algorithm. This approach trades acceptable memory overhead for the elimination of global communication, which makes the Fast APSP algorithm attractive and feasible.
- Finally, we adopt a flexible parallel framework to solve the problem of the optimal number of graph partitioning that cannot completely match the number of GPUs.

2 BACKGROUND

2.1 MIN-PLUS Matrix Multiplication

MIN-PLUS matrix multiplication plays an important role in the solution of APSP problems. Sao et al.[23] discussed in detail the connection between MIN-PLUS (semiring GEMM) and APSP problem. Sections 2.4 and 2.5 will also introduce the role of MIN-PLUS in different APSP algorithms. We introduce MIN-PLUS from the perspective of matrix operations. First, we define ⊗ and ⊕ as the following two scalar operations:

$$x \oplus y := min(x, y) \tag{1}$$

$$x \otimes y := x + y \tag{2}$$

Among them, x and y represent scalars. For matrix $A(m, k)$ and matrix $B(k, n)$. The MIN-PLUS product of A and B is defined as:

$$C_{ij} \leftarrow \sum_{\oplus}^{k} A_{ik} \otimes B_{kj} = \underset{k}{min}(A_{ik} + B_{kj}) \tag{3}$$

It can be seen that the MIN-PLUS is similar to the General matrix multiply (GEMM). Change $(*, +)$ in GEMM to $(+, min)$ will get MIN-PLUS. In order to show the connection between MIN-PLUS and the shortest path in Graph, consider a graph consisting of three types of vertices which are: m source vertices, n destination vertices, and p bridge vertices in Figure 1a. Each bridge vertex is connected to every source vertex and destination vertex.

Figure 1b is the matrix form of MIN-PLUS which has three matrices, A, B, and C. A_{ik} represent the weight of the edge (s_i, b_k), and B_{kj} represent the weight of the edge (b_k, d_j). And C_{ij} represents the shortest distance from source vertex s_i to destination vertex d_j. Then $A_{ik} \otimes B_{kj} = A_{ik} + B_{kj}$ represents the distance from s_i to d_j through b_k. Therefore, we can find the minimum value of C_{ij} via every bridge, which is the shortest distance from s_i to d_j. Due to the connection between MIN-PLUS and GEMM, the optimization techniques of GEMM [2, 15, 27, 30] can be used to optimize MIN-PLUS.

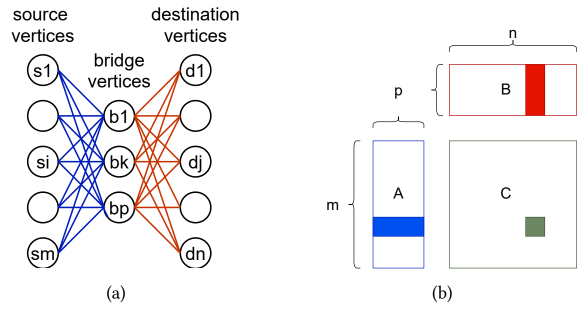

Figure 1: An illustration of MIN-PLUS. (a) shows the shortest path from the source vertices to the destination vertices through the bridge vertices. (b) shows the MIN-PLUS operator in matrix form, which is similar to GEMM.

2.2 The Floyd-Warshall Algorithm

Firstly, we introduce the serial Floyd-Warshall (FW) algorithm, which adopts the strategy of dynamic programming (see Algorithm 1). In the whole algorithm, it is necessary to maintain a matrix of size n^2, which represents the shortest distance of all pairs in the graph, denoted as *dist*. First, initializes the matrix *dist*. For vertex u and vertex v, if there is an edge from vertex u to vertex v in graph G, the weight is w(u,v). Then dist[u,v] will be set to w(u,v). Otherwise, it will be set to positive infinity. After that, n iterations will be performed. In the k_{th} iteration, if there is a shorter path through vertex k, i.e., $dist[i, j] > dist[i, k] + dist[k, j]$, then update

Algorithm 1 Floyd-Warshall Algorithm for APSP

```
1: Let $n = |V|$
2: init $dist$
3: for $k = \{1, 2 \cdots, n\}$ do
4:     for $i = \{1, 2 \cdots, n\}$ do
5:         for $j = \{1, 2 \cdots, n\}$ do
6:             $dist[i, j] = min(dist[i, j], dist[i, k] + dist[k, j])$
7:         end for
8:     end for
9: end for
10: return $dist$
```

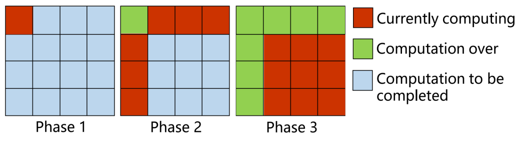

Figure 2: Overview of Katz's algorithm executing where 0,0 is the primary block[13].

$dist[i, j]$ by $dist[i, k] + dist[k, j]$. After n iterations, all pairs can find the shortest path.

Then, we describe the parallel FW algorithm proposed by Katz and Kider[13], which we call the blocked FW algorithm. The algorithm partitions the distance matrix into blocks of equal size. Thus, a distance block can easily be mapped to a GPU thread block. The algorithm proceeds in rounds, with each round consisting of 3 phases. Each of the 3 phases is an individual GPU kernel. Each round gives the algorithm a new primary block. This primary block is set along the diagonal axis of the graph, with the first primary block starting at location $(0, 0)$ and the final block at $(n - blockSize, n - blockSize)$. The $blockSize$ denotes the size of the blocks.

Figure 2 helps to provide a higher-level view of the execution of all 3 phases where the primary block is in location (0,0). As shown in the figure, phase 1 computes the standard FW algorithm for the primary block. Phase 2 uses the MIN-PLUS method as the previous section introduced to compute the blocks on the same row and column as the primary block. Obviously, these block values are dependent on the primary block. In phase 3, all remaining blocks that haven't been computed will be computed. In this phase, the blocks on the same row and column as the current block are used to compute the values of the current block also using the MIN-PLUS operations. After $n/blockSize$ rounds, all pairs' shortest paths will be calculated. The corresponding algorithm is shown in Algorithm 2.

2.3 The SSSP Algorithms

The goal of the SSSP problem is to find the shortest path from the source vertex to all other vertices. The solution to the APSP problem can be obtained by solving the SSSP problem for all vertices in the graph.

For the solution to the SSSP problem, there are two classic algorithms: the Dijkstra algorithm and the Bellman-Ford algorithm. The Dijkstra algorithm utilizes a priority queue so that it has a low time complexity$O(vlogv + e)$, in which v represents the number

Algorithm 2 Blocked Floyd-Warshall Algorithm for APSP

```
1: Let $N = n/blockSize + 1$
2: init $dist$
3: for $k = \{1, 2 \cdots, N\}$ do
       //Phase 1: computing the primary block
4:     $dist_{block_{k,k}} \leftarrow BlockedFW(dist_{block_{k,k}})$;
       //Phase 2: computint the same row and column blocks as primary block
5:     for $i = \{1, 2 \cdots, N\}(i \neq k)$ do
6:         $dist_{block_{k,i}} \leftarrow BlockedFW(dist_{block_{k,k}})$;
7:     end for
8:     for $j = \{1, 2 \cdots, N\}(j \neq k)$ do
9:         $dist_{block_{j,k}} \leftarrow BlockedFW(dist_{block_{k,k}})$;
10:    end for
       //Phase 3: computing the remaining blocks
11:    for $i = \{1, 2 \cdots, N\}(i \neq k)$ do
12:        for $j = \{1, 2 \cdots, N\}(j \neq k)$ do
13:            $dist_{block_{i,j}} \leftarrow BlockedFW(dist_{block_{k,i}}, dist_{block_{k,j}})$;
14:        end for
15:    end for
16: end for
17: return $dist$
```

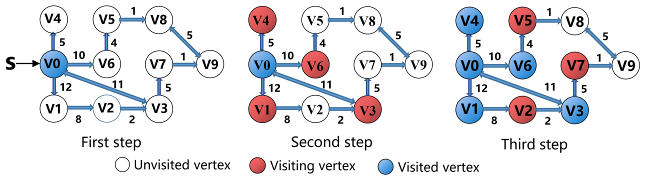

Figure 3: Overview of Harish's algorithm executing the first three steps.

of vertices and e represents the number of edges. But since the algorithm can only process one intermediate vertex at a time, it is poorly suited for a parallel architecture such as a GPU that requires thousands or more parallel threads to fully occupy the machine. On the contrary, the Bellman-Ford algorithm has a higher time complexity$O(ev)$. But it is well-suited to parallel execution across vertices.

Combining the characteristics of these two algorithms, Harish and Narayanan[10] proposed an excellent fine-grained parallel method to target massively parallel machines such as GPUs. Rather than considering one intermediate vertex at a time from a priority queue like Dijkstra's method, this algorithm processes all vertices that are the children of the vertices visited in the last step in parallel at a time. Figure 3 shows the first three steps of the algorithm, and the later calculation process is calculated according to this principle. In the first step, only the source vertex is computed and its neighbors, or rather its children, are activated, which will be parallel computed in the next step. Obviously, the vertices processed in the second step are the vertices activated in the first step. Similarly, the children of the vertices processed in the second step are also activated for the calculation of the third step.

Instead of using the adjacency matrix to store the graph as in the FW method, Harish's algorithm uses the adjacency list to store the original graph, which can easily find the neighbors of all vertices. In the algorithm's implementation, A boolean mask array, M_a, of size n, is constructed, as well as a weight array, W_a, of size |E|. |E| is the number of edges in the graph. Additionally, a cost array, a temporary cost array, a path array, and a temporary path array are created, C_a, TC_a, P_a, TP_a respectively, each of size n. Before the algorithm starts, all the elements of the mask array are set to false except $M_a[0]$, which corresponded to the source vertex. Then, every vertex is assigned to a GPU thread.

Each iteration includes three kernels. During the first kernel's execution, each GPU thread checks if its M_a is true. If yes, it fetches its current cost from the cost array C_a and its neighbor's weights from the weight array W_a. If the neighbor's temporary cost is greater than the cost of the current vertex plus the edge weight of the neighbor, the neighbor's temporary cost is updated by the sum. This process is shown in Algorithm 3. In this kernel, when the cost array TC_a was updated, atomic operations should be used to ensure correctness as described in the work of Kemp et al [14].

Algorithm 3 SSSP-KERNEL1(V_a, E_a, W_a, M_a, C_a, TC_a)

```
1:  $tid \leftarrow threadID$
2:  if $M_a[tid]$ then
3:      for all neighbors nid of tid do
4:          $atomic\{$
5:          if $TC_a[nid] > C_a[tid] + W_a[nid]$ then
6:              $TC_a[nid] \leftarrow C_a[tid] + W_a[nid]$
7:          end if
8:          $\}$
9:      end for
10: end if
```

Kernel 2 updates the paths based on the value of costs is shown in Algorithm4.

Algorithm 4 SSSP-KERNEL2(V_a, E_a, W_a, M_a, C_a, TC_a, TP_a)

```
1: $tid \leftarrow threadID$
2: if $M_a[tid]$ then
3:     $M_a[tid] \leftarrow false$
4:     for all neighbors nid of tid do
5:         if $TC_a[nid] == C_a[tid] + W_a[nid]$ then
6:             $TP_a[nid] \leftarrow tid$
7:         end if
8:     end for
9: end if
```

In the Third kernel, as described in algorithm 5, the cost C_a and the temporary cost TC_a will be compared. If C_a is greater than TC_a, C_a will updated by TC_a, P_a will updated by TP_a and the corresponding M_a will be set true.

In addition, there are also a number of improvements based on Harish's GPU algorithm. Okuyama et al[18] exploit the coarse-grained parallelism by using a task parallelization scheme that associates a task with an SSSP problem. The delta-stepping algorithm proposed [20] is a compromise and shows good performance

Algorithm 5 SSSP-KERNEL3(V_a, E_a, W_a, M_a, C_a, TC_a, P_a, TP_a)

```
1: $tid \leftarrow threadID$
2: if $C_a[tid] > TC_a[tid]$ then
3:     $C_a[tid] \leftarrow TC_a[tid]$
4:     $P_a[tid] \leftarrow TP_a[tid]$
5:     $M_a[tid] \leftarrow true$
6: end if
7: $TC_a[tid] \leftarrow C_a[tid]$
8: $TP_a[tid] \leftarrow P_a[tid]$
```

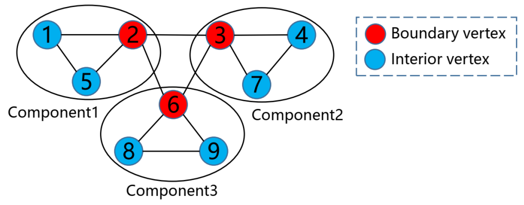

Figure 4: An illustration of the input graph G after graph partition.

on GPU. But the work also points out that for planar graphs, such as road networks, which have high diameters and low degrees, the delta-stepping algorithm is inefficient. Besides, there are many works [3, 16, 29] for the parallel solution of the SSSP problem.

2.4 The Part APSP and decentralized Part APSP Algorithm

The Part APSP algorithm[7] is proposed to compute the shortest distance for all pairs of vertices based on the parallel FW algorithm. Contrary to Katz and Kider's FW algorithm, the domain decomposition strategy is utilized here to handle the large-size graphs. The algorithm applies the idea of divide and conquer, including four steps (see Algorithm 6). Additionally, the matrix representation is shown in Figure 5(1).

- **Step 1: preprocessing.** The input graph is nearly evenly divided into multiple components by the master and broadcast the result to all workers. The vertices owned by each process are categorized into the interior and boundary subsets, as illustrated in Figure 4. The interior vertices are those only connected with vertices belonging to this own process. The boundary vertices correspond to those connected with vertices belonging to the other processes. In the representation by matrix operations, the boundary vertices are ordered in front of the interior vertices.
- **Step 2: solving the local APSP problem.** Each process individually applies the FW algorithm to solve the APSP problem for both the interior and boundary vertices. This step corresponds to the update of the diagonal matrix as shown in Figure 5(1)b. Then each worker sends the shortest distances and paths, which are only relaxed by its local vertices, to master.
- **Step 3: solving the boundary APSP problem.** The boundary vertices from all processes are gathered to form the boundary sub-graph (BG). The master applies still the FW algorithm on the so-arising boundary sub-graph. Then, the results are broadcast to workers for the update by repeating

the calculations in Step 2. After this, each component obtains the all-pairs shortest paths of its local vertices. Finally, each worker broadcasts the shortest distances and paths between their boundary vertices to all other vertices in its own component to other workers.

- **Step 4: solving the global APSP problem.** The shortest distance for all pairs of vertices in the whole graph can be obtained using the MIN-PLUS operations based on the intermediate results from Step 2 and Step 3. We denote the MIN-PLUS operation as $A \otimes B \otimes C$.

Decentralized Part APSP is an optimization of Part APSP, and it also has four steps, as shown in algorithm 7 and Figure 5(2).

- **Step 1: preprocessing.** In the decentralized Part APSP algorithm, each process reads the original graph and partitions the graph.
- **Step 2: solving the local APSP problem.** All the processes calculate the FW algorithm for its local vertices, with is the same as Part APSP. After that, each process broadcasts the distance and path matrices of boundary vertices to other processes.
- **Step 3: solving the boundary SSSP problem.** Using the information of all boundary vertices after being relaxed through their local vertices, each process calculates the shortest path from its own boundary vertices to other boundary vertices using the SSSP algorithm. Then each process updates the all-pairs shortest path of its own component. Finally, each process broadcasts the shortest distances and paths between their boundary vertices to all other vertices in its own component to other processes.
- **Step 4: solving the global APSP problem.** This step is the same as step 4 of Part APSP.

On account of the graph partition, the Part APSP and decentralized Part APSP algorithms obtain a better time complexity $O(n^{2.25})$, which is lower than using the FW algorithm on the whole graph. Additionally, its matrix-based structure is regular enough to allow for effificient parallel implementation on the GPUs. While it's able to take advantage of multi-node GPU clusters by calculating the partitions in parallel. But due to global communication, when the parallel scale is large, their scalability will be limited, even though the decentralized Part APSP has reduced Part APSP's four communications to two.

3 THE FAST APSP ALGORITHM

3.1 Overview

The motivation of the Fast APSP algorithm is to combine the global SSSP algorithm and the local FW algorithm to solve the APSP problem of super-size graphs. Our algorithm also has a low time complexity $O(n^{2.25})$ for the planar graph. Due to the global SSSP algorithm, we store the original graph in each process, which leads to no communication in the computing stages. In addition, compared with the Part APSP and decentralized Part APSP algorithms, our algorithm reduces one local FW operation, simplifies the MIN-PLUS operation $A \otimes B \otimes C$ to $A \otimes D$, and makes data access continuous in the MIN-PLUS calculation.

Algorithm 6 Partitioned All-Pairs Shortest Path Algorithm

```
Input: A graph G( V, E ), where the weights of E are non-negative
output: The distances between the vertices in G
 1: // Step1
 2: read and decompose the graph // master
 3: Scatter the partitions C(i) to all worker processors // master
 4: // Step2
 5: for all worker processor p in parallel do
 6:     Floyd-Warshall(C(p)) %compute_APSP(C(p))
 7:     Send computed boundary distances to the master processor
 8: end for
 9: // Step3
10: Graph_BG = extract_boundary_graph(G) // master
11: compute_APSP(BG) // master
12: send BG to the all worker processor // master
13: for all worker processor p in parallel do
14:     Floyd-Warshall(C(p)) %update_APSP(C(p))
15:     send C(p) to all other worker processors
16: end for
17: // Step4
18: for all worker processor p in parallel do
19:     for all component C2 in G do
20:         // 2 MIN-PLUS operators
21:         compute_APSP_between_components(C(p),C2)
22:     end for
23: end for
```

Table 1: Related Mathematical Symbols

Type	Symbol	Description
process	p	number of MPI process
	p_i	the i^{th} process
	$group_i$	the i^{th} process group
matrix	$dist$	Adjacency matrix of a graph
	$dist_G(s,d)$	shortest path from s to d in G
	$dist_G(s,:)$	shortest path from s to all vertices in G
	$dist_i(s,d)$	shortest path from s to d computed in step i
graph	G	Input graph
	V	Vertice Set
	E	Edge Set
	n	number of vertices
	m	number of edges
	k	number of components
	C_i	the i^{th}components
	B_i	the boundry vertices set in C_i
	In_i	the internal vertice set in C_i
	α_i	percentage of boundary vertices in C_i
	t	the number of parallel tasks in the SSSP algorithm

The details of the proposed algorithm are presented in the following sections. In Table 1, the definition of the mathematical symbols which are utilized in the later sections is summarized.

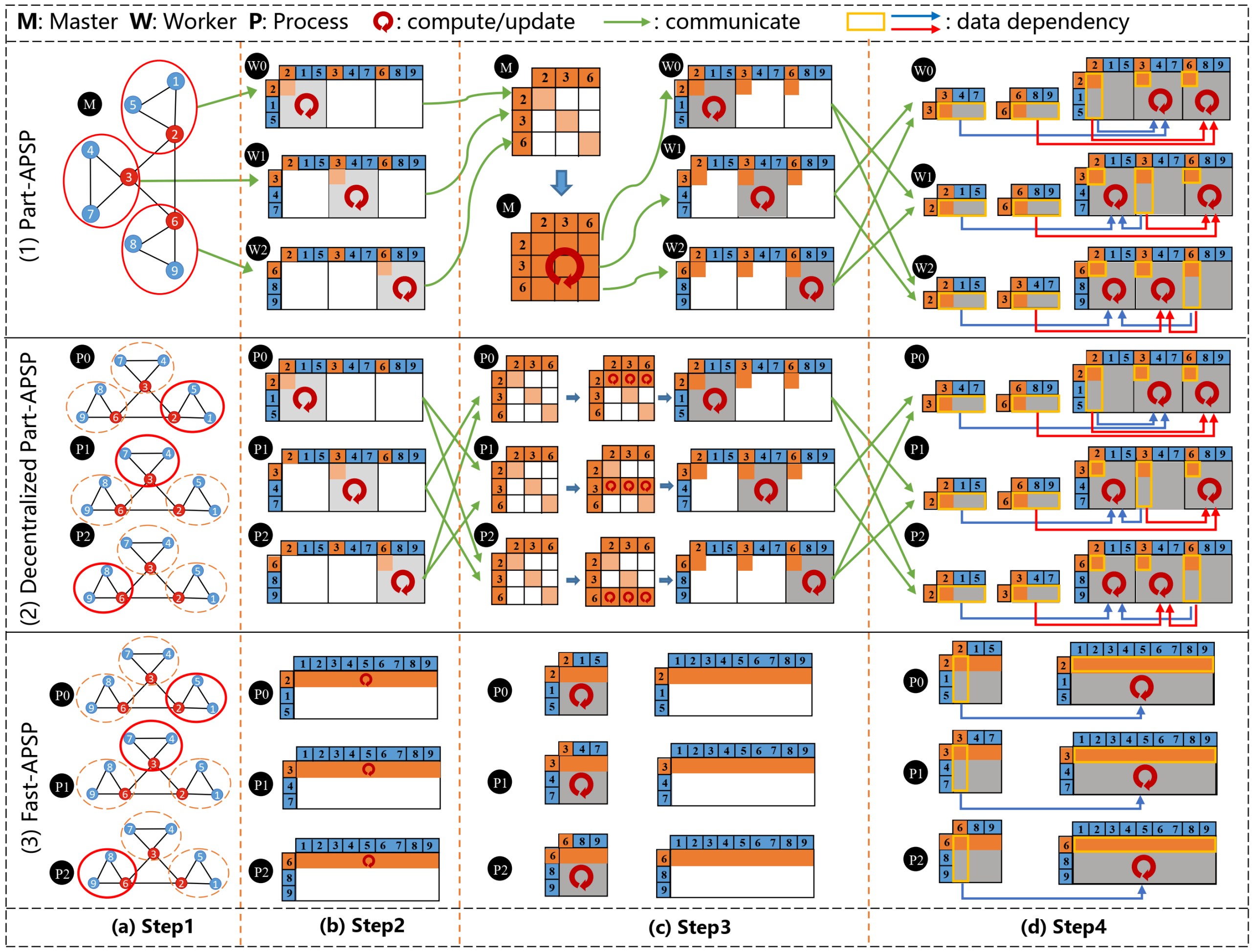

Figure 5: An illustration of three algorithms and (1): Part APSP. (2): Decentralized Part APSP. (3): Fast APSP.

The Fast APSP algorithm is presented as Algorithm 8 and the execution of the matrix representation is shown in Figure 5(3). The Fast APSP algorithm also has four steps.

3.2 Step 1: Preprocessing

Similar to the decentralized Part APSP algorithm, the Fast APSP algorithm also uses the graph partition tool METIS[12] to divide the input graph G into k component, i.e. sub-graph, of nearly the same size and each component is assigned to a process. We denote the boundary vertices by B_i and the interior vertices by In_i. Although there are two matrices, distance and path. For simplicity, we only use the distance matrix $dist$ to describe the calculation. Furthermore, we store the whole graph in the CSR compressed matrix format in each process, which is a suitable format for the SSSP calculation.

3.3 Step 2: Solving SSSP problem of boundary vertices

For each boundary vertex b_i of the set $Bi\{b_1, b_2, b_3, ...\}$, the Fast APSP algorithm adopts the GPU SSSP algorithm, which carried out Kemp's[14] atomic operation optimization and Okuyama's[18] coarse-grained parallel optimization based on Harish's[10] algorithm, to compute the shortest paths to all vertices of the input graph G. The set of the shortest distances for all boundary vertices is denoted by $dist_G\{b_1, b_2, b_3, ...\}$. As Figure 5(3)(b) described, the boundary matrix in the bar matrix corresponding to the shortest distances from the boundary vertices to the other vertices are calculated in this step. Because of we store the origin graph in each process, there is no communication in the global SSSP calculation.

3.4 Step 3: Computing all pairs shortest distances in each component

In this step, We compute the all-pairs shortest paths of vertices in each component using the block FW algorithm. As shown in Figure 5(3)(c), The square matrix is calculated in the step, which stores the shortest distances between pairs of local vertices are calculated. Besides, the computation carried out results in the global shortest paths, which is proved by the following lemma.

Algorithm 7 Decentralized partitioned All-Pairs Shortest Path Algorithm[6]

```
Input: A graph G( V, E ), where the weights of E are non-negative
output: The distances between the vertices in G
 1: // Step1
 2: for all processor p in parallel do
 3:     read and decompose the graph
 4:     get own component C(p)
 5: end for
 6: // Step2
 7: for all processor p in parallel do
 8:     Floyd-Warshall(C(p)) %compute_APSP(C(p))
 9:     send computed boundary distances to all other processoes
10: end for
11: // Step3
12: for all processor p in parallel do
13:     Graph_BG = extract_boundary_graph(G)
14:     for all vertice v in $C(p) \cap BG$ do
15:         solve_SSSP(v,BG)
16:     end for
17: end for
18: for all processor p in parallel do
19:     Floyd-Warshall(C(p)) %compute_APSP(C(p))
20:     send computed distances to all other processors
21: end for
22: // Step4
23: for all processor p in parallel do
24:     for all component C2 in G do
25:         // 2 MIN-PLUS operators
26:         compute_APSP_between_components(C(p),C2)
27:     end for
28: end for
```

Lemma 1. *For any two vertices s and d in the component C_i. The shortest distance computed in step 3, denoted as $dist_3(s,d)$ is equal to the shortest distance between the two vertices in G, denoted as $dist_G(s,d)$, that is $dist_3(s,d) = dist_G(s,d)$.*

Proof: In graph G, There are two possible shortest paths from vertex s to vertex d.

The first case is reaching vertice d without passing through vertices in any other components, as shown in path1 in Figure 6(a). In this case, only need to solve the APSP problem in this component to get an accurate result. Therefore, $dist_3(s,d) = dist_G(s,d)$ obviously holds.

The second case is that the source vertex s needs to pass through multiple vertices outside the component and then reach the destination vertex d. Under these conditions, two boundary vertices have to be passed at least, one for the outgoing vertex and the other one for the incoming vertex as the path2 shown in Figure 6(a). The path2 starts from s, passes through boundary vertices $b1$ and $b2$, and reaches the destination vertex d. We define the shortest distance between s and $b1$ as $dist_3(s,b1)$, the shortest distance between $b1$ and $b2$ as $dist_3(b1,b2)$, and the shortest distance between $b2$ and d as $dist_3(b2,d)$. In step 2, we obtained the shortest distance between $b1$ and all other vertices of the graph, so that we can acquire $dist_3(b1,b2) = dist_G(b1,b2)$. The $dist_3(s,b1)$ and $dist_3(b2,d)$ are the shortest distance between the internal vertices and the boundary vertices of the current component. Their paths don't contain edges outside the component. So the calculations of their global shortest distance only depend on the vertices' relationship inside the component. Consequently, the shortest distance between the internal vertices and the boundary vertices computed by the FW algorithm in step 3 is their final result, i.e. $dist_3(s,b1) = dist_G(s,b1)$ and $dist_3(b2,d) = dist_G(b2,d)$. Therefore, we now have $dist_G(s,d) = dist_3(s,b1) + dist_3(b1,b2)+dist_3(b2,d)$. In addition, because the local shortest distance $dist_3(s,d) \leq dist_3(s,b1) + dist_3(b1,b2)+dist_3(b2,d)$. That is, $dist_3(s,d) \leq dist_G(s,d)$. And because the global shortest distance must not be bigger than the local shortest distance, i.e. $dist_G(s,d) \leq dist_3(s,d)$. So there must be: $dist_3(s,d) = dist_G(s,d)$. Similarly, $dist_3(s,d) = dist_G(s,d)$ is also satisfied if the shortest path between two vertices contains multiple external edges. ■

Algorithm 8 Fast All-Pairs Shortest Path Algorithm

```
Input: A graph G( V, E ), where the weights of E are non-negative
output: The distances between the vertices in G
 1: // step1
 2: for all processor p in parallel do
 3:     read and decompose graph G
 4:     get own component C(p)
 5: end for
 6: // step2
 7: for all processor p in parallel do
 8:     for all boundary vertex v in C(p) do
 9:         solve_SSSP( v, G );
10:     end for
11: end for
12: // step3
13: for all processor p in parallel do
14:     Floyd-Warshall(C(p))%compute_APSP(C(p))
15: end for
16: // step4
17: for all processor p in parallel do
18:     // 1 MIN-PLUS operator
19:     compute_APSP_components( C(p), G )
20: end for
```

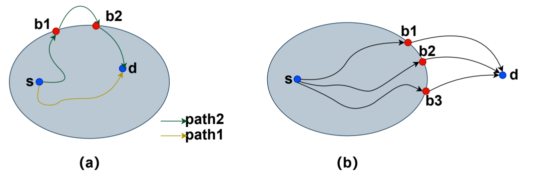

Figure 6: An illustration of proof. (a) lemma1. (b) lemma2.

3.5 Step 4: Computing the shortest path from internal vertices to vertices in other components

We acquired the shortest paths between the boundary vertices and all other vertices of the input graph and the shortest paths between the internal vertices and all the local vertices of their component. In the following phase, we should calculate the shortest paths between the internal vertices and the other vertices of the input graph except the vertices of the local component. After the calculation of step 4, the all-pairs shortest paths of the input graph can be obtained.

In this step, we apply the MIN-PLUS method to get the remaining shortest paths. Let the internal vertices set of the component C_i is: $In\{v_1, v_2, ...\}$, and the set of vertices in other components is $\{d_1, d_2, d_3, ...\}$. In this step, we need to compute the shortest distance from v_i to d_i, denoted as $dist_4(v_i, d_i)$, where $v_i \in In_i$ and $d_i \in (V - C_i)$. V is the set of all the vertices of the input graph. The MIN-PLUS method calculates the the minimum of $dist_3(v_i, b_i) + dist_2(b_i, d_i)$ as $dist_4(v_i, d_i)$, i.e., it finds a boundary vertex b_i that minimizes $dist_4(v_i, d_i)$. According to the previous introduction, we can easily know that the $dist_3(v_i, b_i)$ and $dist_2(b_i, d_i)$ are the results computed from step 3 and step 2 respectively. Since the calculation only depends on the results of the local component, we also need to prove the shortest distances obtained in this step are the global shortest distances, i.e., $dist_4(v_i, d_i)$=$dist_G(v_i, d_i)$.

Lemma 2. *Let v_i belong to the internal vertice in the component C_i, and d belong to any vertice in G but do not belong to the component C_i. Let $dist_4(v_i, d) = min(dist_3(v_i, b_i) + dist_2(b_i, d))$. Then, $dist_4(v_i, d)$ is equal to the shortest distance from v_i to d in G, that is, $dist_4(v_i, d) = dist_G(v_i, d)$.*

Proof: As shown in Figure 6(b), it can be observed that to reach the vertex d_i outside the component, v_i must pass through the boundary vertex b_i on the component C_i.So we calculate the shortest distance between the vertex v_i and d_i is the smallest value of $dist_G(v_i, b_i)$ plus $dist_G(b_i, d_i)$, i.e. $dist_G(v_i, d_i) = min(dist_G(v_i, b_i) + dist_G(b_i, d_i))$. Through the introduction above, we can get $dist_3(v_i, b_i) = dist_G(v_i, b_i)$ and $dist_2(b_i, d_i) = dist_G(b_i, d_i)$. Therefore, there are: $dist_G(v_i, d_i) = min(dist_3(v_i, b_i) + dist_2(b_i, d_i))$, that is, $dist_4(v_i, d_i) = dist_G(v_i, d_i)$. ■

Each component is calculated independently in this step. There are $|In_i||V - C_i|$ kinds of vertices pair combinations needed to be computed. Since it is not known which boundary vertice passed through can get the minimized value in Lemma2. Therefore, all possibilities must be traversed. And for each component C_i, there are $|In_i||V - C_i||B_i|$ possibilities need to be computed.

We denote the MIN-PLUS operation as $A \otimes D$. As shown in figure 5(3)(d), its data access is contiguous in the MIN-PLUS operation.

3.6 Distribute parallel framework

Since the appropriate number of graph partitions k and the number of GPUs p may not completely match, especially in non-planar graphs. So we adopt a flexible parallel framework shown in Figure 7. In the framework, the number of processes is also p, which is determined according to the number of GPUs. Therefore, the relationship between k and p has three situations.

- When $k = p$, there are k tasks corresponding to k components. Each process binds a task which means binds the task of the component C_i to the process P_i. There is no data dependency between the k tasks and no data communication between the P processes. The process P_i only needs to execute the computation in steps 2-4 in sequence to get the accurate solution.
- When $k > p$, there are more tasks than processes. Each process needs to be responsible for multiple tasks. These tasks need to be shuffled and then randomly assigned to different processes. Each process is responsible for k/p tasks.
- When $k < p$, the number of tasks is less than the number of processes. Therefore, the related matrix needs to be divided finely. A process group is responsible for a task. The process group $group_i$ is responsible for the i^{th} task corresponding to the component C_i. Each process group contains p/k processes. Because there is no data dependency among k tasks, there is no data communication between the process groups, and only the processes in the same group have data communication.

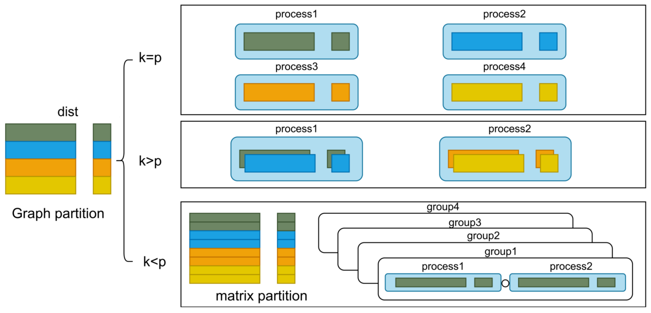

Figure 7: The distributed parallel framework of the Fast APSP algorithm

For the case of $k >= p$, since there is no data dependency and no communication between processes, extending it to a distributed system is easy. For the case of $k < p$, the SSSP algorithm, the Floyd-Warshall algorithm, and the MIN-PLUS algorithm in steps 2-4 need to be extended to a distributed system. It is easier to develop the SSSP problem in a distributed system. Only need to run the SSSP algorithm in different processes. Regarding the parallel implementation of the Floyd-Warshall algorithm, we refer to Kannan's work [11] to divide Floyd-Warshall in one dimension and extend it to distributed clusters. As for the parallel implementation of the MIN-PLUS operator, we implemented a distributed GEMM algorithm by dividing the matrix into rows. In addition, we also implement extended memory in this framework when the GPU memory is insufficient.

3.7 Work Analysis

The time complexity of the Fast APSP algorithm has a large relationship with the characteristics of the graph and the quality of graph partitioning. These effects are mainly reflected in the parameter $\overline{\alpha}$, which represents the proportion of the boundary vertices. In the component, the boundary vertices denoted by B_i refer to those which connect to the other components by the edges. The interior vertices denoted by In_i correspond to those without the connection to the other components. The proportion of the boundary vertices α_i is defined by $\alpha_i = |B_i|/(|B_i| + |In_i|)$. For a graph with n vertices and m edges, when it is divided into k parts, $\overline{\alpha}$ is also determined accordingly.

In step 1, we need to perform graph partitioning. Although graph partitioning is an NP-hard problem, good results can be achieved in a short time using METIS. The time required for this part of the work is $T_{GraphPartition} = O(nlogn)$.

In step 2, we have k tasks which refer to solving the single source shortest path of boundary vertices in k components. Each component has $O(n/k)$ vertices and $O(\overline{\alpha} * n/k)$ boundary vertices. For each boundary vertice, we need to solve its SSSP problem. It takes $O(nlogn + m)$ to solve the single SSSP problem, where m denotes the number of edges. So in step 2, the time complexity is $T_{SSSP} = O(\overline{\alpha}n(nlogn + m))$

In step 3, there are k tasks, and each task needs to use the Floyd-Warshall algorithm on a graph with $O(n/k)$ vertices to solve its APSP problem. So in this step, the required time complexity is $T_{FW} = O(k * (n/k)^3)$.

In step 4, there are k tasks, and each task needs to perform a MIN-PLUS operation. Each MIN-PLUS operation involves two matrices, the dimensions are $(1 - \overline{\alpha})n/k \times \overline{\alpha}n/k, \overline{\alpha}n/k \times n$ respectively. So the time complexity in step 4 is: $T_{MIN-PLUS} = O(\overline{\alpha}(1 - \overline{\alpha})\frac{n^3}{k})$.

The time complexity of the entire algorithm is the sum of the time complexity of the four steps. Since the time complexity of step 1 is relatively small, it is ignored here. Therefore, the time complexity of the algorithm is:

$$O(\overline{\alpha}n^2logn + \overline{\alpha}nm + \frac{n^3}{k^2} + \overline{\alpha}(1 - \overline{\alpha})\frac{n^3}{k}) \ .$$

It can be seen from the formula that the time complexity has four parameters: $\overline{\alpha}$, n, m, and k. There are two cases:

(1) For a planar graph, the time complexity of the algorithm can be further simplified. In the planar graph, the number of boundary vertices of each component is $O(\sqrt{n/k})$ [9], so $\overline{\alpha}$ is $\frac{\sqrt{n/k}}{n/k}$. For k, generally taking $\sqrt{n}$ will have the optimal solution in a planar graph [7]. Because it is a sparse graph, we take $m = n$ here. Also, when n is large, $\overline{\alpha}$ is small, so $(1 - \overline{\alpha})$ is approximated to 1. So the time complexity of the algorithm is:

$$O(n^{7/4}logn + n^2 + n^{9/4}) \ .$$

This time complexity is $O(n^{7/4}logn + n^{9/4})$ more than the time complexity of Part APSP algorithm, which is $O(n^{9/4})$. However, when n is greater than 16, $n^{1/2}$ is greater than $logn$. Therefore, the time complexity of the algorithm is close to $O(n^{9/4})$.

(2) For a non-planar graph, especially graphs that are not easily partitioned, all vertices are boundary vertices in the extreme case, $\overline{\alpha}$ is 1. We also take $\sqrt{n}$ as the partition number. The Fast APSP algorithm degenerates to the Dijkstra algorithm. Its time complexity is the same as the Dijkstra algorithm, $O(n^2logn + nm)$. Similarly, the Part APSP and decentralized Part APSP will degenerate into no partition APSP algorithm, its time complexity is $O(n^3)$.

4 EXPERIMENT AND PERFORMANCE EVALUATION

4.1 Platform and data

In order to comprehensively evaluate the proposed Fast APSP algorithm, several classes of experiments are carried out as follows:

- To find out the influence of the graph partitions on the performance, numerical evaluations of the Fast APSP algorithm with a varying number of graph partitions are performed.
- Evaluations on the distributed clusters are performed to demonstrate the superiority of the proposed Fast APSP algorithm in terms of the significant speedup.
- The significantly improved scalability of the Fast APSP algorithm is illustrated.
- Relying on the Fast APSP algorithm, the attempts to handle the large graphs which have been reported in the relevant references are made and the promising results are presented.

Table 2: Test data

Name	Label	n	nnz	nnz/n
Graph in graph partition experiment				
wing	WIG	62,032	243,088	3.9
delaunay_n16	DLY	65,536	393,150	6.0
fe_tooth	FET	78,136	905,182	11.6
onera_dual	ODL	85,567	419,201	4.9
Graph in performance experiment				
598a	598a	110,971	1.4M	13.4
luxembourg_osm	LUX	114,599	239,332	2.1
web-sk-2005	WSK	121,422	668,838	5.5
usroads-48	USD	126,146	323,900	2.6
fe-ocean	FEO	143,437	819,186	5.7
Graph in scalability experiment				
roadNet-PA	RNT	1M	3M	2.8
web-it-2004	WEB	0.5M	14.4M	28.2
sc-pwtk	SCP	0.2M	11.3M	51.9
soc-youtube	SOC	0.5M	3.9M	7.8
Graph in large scale experiment				
germany-osm	GER	11.5M	24.7M	2.14

(1) Test Bed: The computations associated with the multi-node experiments were run on a supercomputing cluster. Each computation node is equipped with 4 AMD GPUs, a 32 cores AMD CPU, and 128 GB of RAM. The software we used is METIS 5.1.0, G++ 7.3.1, and MPI 4.0.2a1.

(2) APSP baseline: We compared the GPU Dijkstra algorithm [18], CPU Dijkstra as baseline algorithms, Part APSP algorithm[7] and the decentralized Part APSP algorithm[6]. For the CPU Dijkstra algorithm, We extended the Dijkstra algorithm with the binary heap to distributed clusters, which shows better performance than the Dijkstra algorithm from BGL[24].

(3) Test data: Our test graph data is derived from SuiteSparse Matrix Collection [4] and Network Repository [22], as shown in Table 2. To verify that the method performs better on well-partitioned graphs, we selected different types of graphs for experiments. In addition, in order to conduct experiments quickly and achieve different experimental purposes, we adopt different sizes of graphs in different experiments.

4.2 Performance Results and Observations

4.2.1 Performance of Fast APSP algorithm under the different number of graph partitions.

The Fast APSP algorithm consists of three main routines, i.e. the FW, MIN-PLUS, and SSSP algorithms. The performance of the Fast APSP algorithm with a different number of graph partitions is shown in Figure 8. Here four different graphs are utilized and some common

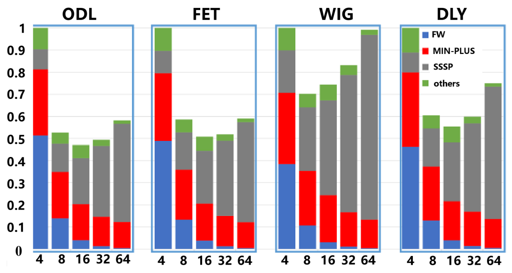

Figure 8: The time after Normalization of Fast APSP algorithm under different graph partitions.

Table 3: $\bar{\alpha}$ of different graphs in different partitions

graph	4	8	16	32	64
ODL	0.039	0.058	0.094	0.142	0.199
FET	0.061	0.103	0.147	0.208	0.273
WIG	0.059	0.088	0.130	0.187	0.253
DLY	0.015	0.030	0.045	0.069	0.103

Table 4: The average speedups of different graphs

Graph	598a	WSK	USD	LUX	FED	AVG
To Dijkstra-CPU	16.19	22.42	17.51	16.74	11.99	16.97
To Dijkstra-GPU	1.37	4.78	8.69	18.02	2.59	7.09
To Part-APSP	12.86	3.84	4.00	3.79	10.94	7.09
To Dec-Part-APSP	6.31	3.97	4.16	3.85	4.72	4.60

observations are made. All experiments are run on one process with one GPU. Overall, there is the optimal number of partitions which results in the smallest computation time. In particular, the computational time of the FW and MIN-PLUS algorithms decreases with more partitions. By contrast, the time of the SSSP algorithm increases and turns to be dominant by enlarging the number of partitions. This can be explained by the results given in Table 3, which demonstrates that the value of $\bar{\alpha}$, to which the time of the SSSP algorithm is proportional continuously increases with more partitions. In other words, the more partitions, the more boundary vertices, and the longer the computation time of the SSSP algorithm.

4.2.2 *Distributed APSP Performance Evaluation.*

We compared the performance of the Fast APSP algorithm with Dijkstra-CPU, Dijkstra-GPU, Part APSP, and decentralized Part APSP algorithms in distributed heterogeneous clusters. The experiments used 2, 4, 8, and 16 nodes, respectively, to run the APSP algorithm by MPI standard[25]. For a fair comparison, we used all the CPUs on each node for the CPU algorithm. In the performance experiment, the number of graph partitions is as same as the number of GPUs used in each experiment. The performance results are shown in Figure 9. We take the running time of Dijkstra-CPU as the benchmark, and the ordinate is the speedup of other algorithms relative to the Dijkstra-CPU algorithm. The average speedups of Fast APSP relative to other algorithms are shown in table 4.

In Figure 9, the Fast APSP algorithm shows the best performance on different computer nodes. When running on the same computer nodes, the speedups of the Fast APSP algorithm on the different

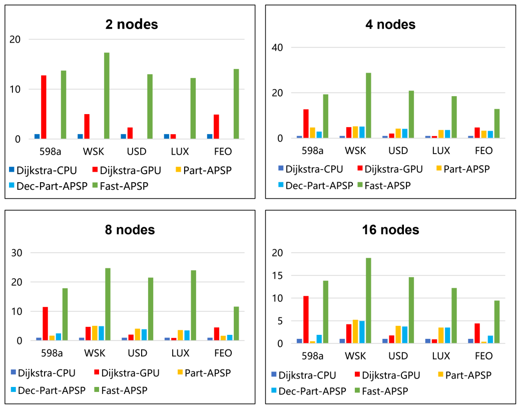

Figure 9: Speedup of Fast APSP, Dijkstra-GPU, Part-APSP, and Decentralized Part-APSP algorithms over Dijkstra-CPU algorithm on different nodes.

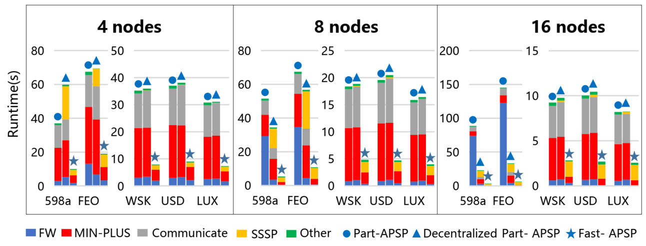

Figure 10: Runtime of each function in each algorithm.

graphs are different. This is mainly due to $\bar{\alpha}$, which is not the same depending on the partition of the graph. Generally, a small $\bar{\alpha}$ means that the ratio of boundary vertices is small, the global SSSP computation is relatively small, such as *road-usroads*, *web-sk-2005*, and *luxembourge_osm*. $\bar{\alpha}$ is large, indicating that the graph is difficult to partition, so the performance is a little bit worse. Moreover, When running on different computer nodes, the speedups of the Fast APSP algorithm on the same graph are also different. This is because graph partitioning is related to the structure of the graph, $\bar{\alpha}$ is not strictly a function of the number of partitions in the graph. In summary, compared to the Dijkstra-CPU, Dijkstra-GPU, Part APSP, and decentralized Part APSP algorithms, it has average speedups of 16.97x, 7.09x, 7.09x, and 4.60x, respectively, which can be seen from Table 4. Noting that in order not to affect the performance of the comparative algorithms, we did not reproduce the memory-extended function of Part APSP and decentralized Part APSP algorithms. Therefore, in the experiment "2 nodes", the Part APSP, and decentralized Part APSP algorithms can't run for the memory limitation.

We measured the running time of the different functions in Part APSP, decentralized Part APSP, and Fast APSP algorithms to explore their performance differences, as shown in Figure10. There are four functions in the algorithm, including FW, SSSP, MIN-PLUS, and Communication. The Part APSP algorithm doesn't have SSSP and the Fast APSP doesn't have communication.

For the FW function, it appears third in the Part APSP algorithm, twice in the decentralized Part APSP algorithm, and only

Table 5: The number of different vertices of the graphs

Graph	598a	WSK	USD	LUX	FED
4 nodes					
avgVerPerComp	6935	7588	7884	7162	8964
allBoundaryVer	15603	556	1230	367	15267
8 nodes					
avginternalPerComp	3467	3794	3942	3581	4482
allBoundary	22667	1164	1899	593	23858
16 nodes					
avginternalPerComp	1733	1897	1971	1790	2241
allBoundary	30978	1976	2954	887	36942

once in the Fast APSP algorithm. So it cost longer time in the Part APSP and decentralized Part APSP algorithms than in the Fast APSP. On the other hand, we find it's especially long for graphs 598a and FEO in the Part APSP algorithm, that is because the total boundary vertices of 598a and FEO is more than other graphs', namely 598a and FEO are not easy to partition and their $\bar{\alpha}$ are bigger. As shown in Table 5, *avgVerPerComp* denotes the average number of vertices in each component, and *allBoundaryVer* denotes the total number of boundary vertice for all components. Due to there being so many boundary vertices, the first FW in "Step 3" cost a long time.

For the SSSP function, it cost more time for graphs 598a and FEO in the decentralized Part APSP algorithm than in the Fast APSP algorithm, which is also caused by the large number of boundary vertices. In the decentralized Part APSP algorithm, before calculating the SSSP function, it needs to convert the many small matrices in COO format received from other processes into a large matrix in CSR format. When the total number of boundary vertices is large, this will consume a lot of time, even more than the time of solving the global SSSP in the Fast algorithm.

For the MIN-PLUS function, the Part APSP and decentralized Part APSP are the same, which calculates the minimum value of $A \otimes B \otimes C$ for all internal vertices. On the contrary, the Fast APSP only calculates the minimum value of $A \otimes D$ for all internal vertices. In addition, since the data C is received from other processes and stored separately in the Part APSP and decentralized Part APSP algorithms, multiple kernels need to be called to complete the calculation in MIN-PLUS, even though the GPU memory can store all the data. But in the Fast APSP, the data is stored contiguously, only one kernel needs to be called if the GPU memory space is sufficient.

For the communication function, except for MPI communication time, most of the time is spent on copying the data before communication and integrating the data after communication, which accounts for about 80% of the total communication time.

4.2.3 *Strong Scaling of the Fast APSP algorithm.*

To explore the strong scalability of the Fast APSP algorithm, we tested different types of graph data, as shown in Table 2. To reduce the influence of other times, such as graph partitioning time and data copy time between different devices, when running on large-scale nodes, the graph adopted in this experiment is relatively larger than the experiment above. The speedup of the Fast APSP algorithm running on different computing nodes compared to running on 1 node is shown in Figure 11. The number of partitions k of all graphs are shown in Table 6.

Table 6: The k of graphs in the strong scaling experiments

graph	roadNet-PA	web-it-2004	sc-pwtk	soc-youtube
k	512	64	32	128

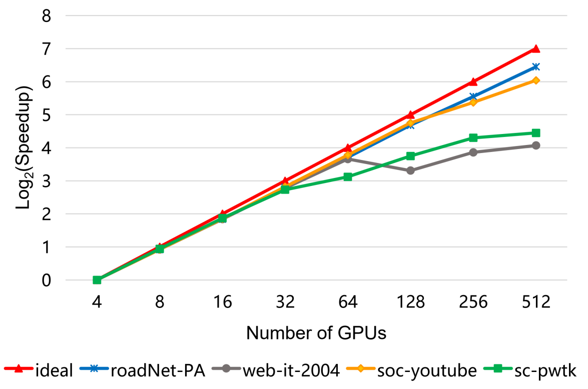

Figure 11: The speedup of the Fast APSP algorithm.

When $k >= p$, the scalability of all graphs is close to linear. The parallel efficiency of *roadNet-PA*, *web-it-2004*, *sc-pwtk* and *soc-youtube* is 68%, 79%, 83% and 84% for $k = p$. When $k < p$, the speedup decreases for all graphs. This is mainly because communication in process groups takes some time when the number of processes is larger than the number of graph partitions. If the graph is easy to partition, such as *road-roadNet-PA*, each process handles one partition, and the Fast-APSP algorithm will obtain a good speedup. Otherwise, the speedup is slightly lower, but still up.

4.2.4 *Large-scale Graph Experiment.*

The largest graph we ran is *germany-osm* from SuiteSparse Matrix Collection [4]. It has 11,548,845 vertices and 24,738,362 edges. Running the APSP algorithm will produce the shortest paths of $1.33 * 10^{14}$ vertice pairs. To solve the shortest path problem on the graph, we used 512 nodes (2048 GPUs) on a distributed system based on AMD GPU, and the computation was finished within 12.45 minutes.

5 CONCLUSION

In our work, we adopt a hybrid method of SSSP and APSP to solve the all-pairs shortest paths problem in parallel, and this method eliminates the data dependence problem in the parallel computation of the part-APSP algorithm and simplified the MIN-PLUS function. Therefore, the Fast-APSP algorithm has achieved good performance and scalability in our experiments. In addition, we used it to solve an APSP problem in a graph with 11,548,845 vertices in 12.45 minutes.

ACKNOWLEDGMENTS

The authors would like to thank all anonymous reviewers for their valuable comments and helpful suggestions. The work is supported National Natural Science Foundation of China, under Grant 62032023, T2125013, 61972377, 62172389.

REFERENCES

[1] Karsten M Borgwardt and Hans-Peter Kriegel. 2005. Shortest-path kernels on graphs. In *Fifth IEEE international conference on data mining (ICDM'05)*. IEEE, 8–pp.

[2] Lung-Sheng Chien. 2010. Hand tuned SGEMM on GT200 gpu. *Technical Report, Tsing Hua University* (2010).

[3] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. 2014. Work-efficient parallel GPU methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 349–359.

[4] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[5] Edsger W Dijkstra et al. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[6] Hristo Djidjev, Guillaume Chapuis, Rumen Andonov, Sunil Thulasidasan, and Dominique Lavenier. 2015. All-Pairs Shortest Path algorithms for planar graph for GPU-accelerated clusters. *J. Parallel and Distrib. Comput.* 85 (2015), 91–103.

[7] Hristo Djidjev, Sunil Thulasidasan, Guillaume Chapuis, Rumen Andonov, and Dominique Lavenier. 2014. Efficient multi-GPU computation of all-pairs shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 360–369.

[8] Donald, B., and Johnson. 1977. Efficient Algorithms for Shortest Paths in Sparse Networks. *J. ACM* 24 (1977).

[9] Greg N Federickson. 1987. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.* 16, 6 (1987), 1004–1022.

[10] Pawan Harish and Petter J Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *International conference on high-performance computing*. Springer, 197–208.

[11] Ramakrishnan Kannan, Piyush Sao, Hao Lu, Drahomira Herrmannova, Vijay Thakkar, Robert Patton, Richard Vuduc, and Thomas Potok. 2020. Scalable knowledge graph analytics at 136 petaflop/s. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 61–73.

[12] George Karypis and Vipin Kumar. 1998. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing* 48, 1 (1998), 96–129.

[13] Gary J Katz and Joseph T Kider. 2008. All-pairs shortest-paths for large graphs on the GPU. (2008).

[14] JEREMY KEMP. 2012. *All-Pairs Shortest Path Algorithms Using CUDA*. Ph. D. Dissertation. Durham University.

[15] Junjie Lai and André Seznec. 2013. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 1–10.

[16] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A pattern based algorithmic autotuner for graph processing on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 201–213.

[17] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152.

[18] Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara. 2012. A task parallel algorithm for finding all-pairs shortest paths using the GPU. *International Journal of High Performance Computing and Networking* 7, 2 (2012), 87–98.

[19] Hector Ortega-Arranz, Yuri Torres, Diego R Llanos, and Arturo Gonzalez-Escribano. 2013. The all-pair shortest-path problem in shared-memory heterogeneous systems. *High-Performance Computing on Complex Environments* (2013), 283–299.

[20] H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano. 2013. A new GPU-based approach to the Shortest Path problem. In *International Conference on High Performance Computing Simulation*.

[21] Arnaud Quirin, Oscar Cordón, Jose Santamaría, Benjamin Vargas-Quesada, and F Moya-Anegón. 2008. A new variant of the pathfinder algorithm to generate large visual science maps in cubic time. *Information processing & management* 44, 4 (2008), 1611–1623.

[22] Ryan A Rossi and Nesreen K Ahmed. 2016. An interactive data repository with visual analytics. *ACM SIGKDD Explorations Newsletter* 17, 2 (2016), 37–41.

[23] Piyush Sao, Ramakrishnan Kannan, Prasun Gera, and Richard Vuduc. 2020. A supernodal all-pairs shortest path algorithm. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 250–261.

[24] J. G. Siek, L. Q. Lee, and A. Lumsdaine. 2002. *The Boost Graph Library: User Guide and Reference Manual.* The Boost Graph Library: User Guide and Reference Manual.

[25] Marc Snir, St Otto, St Huss-Lederman, D Walker, and J Dongarra. 1996. MPI: The Complete Reference The MIT Press. *Cambridge, Massachusetts* (1996).

[26] Clifford Stein, T Cormen, R Rivest, and C Leiserson. 2001. Introduction to algorithms. *The MIT Press* 31, 77 (2001), 13.

[27] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. 2011. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[28] Joshua B Tenenbaum, Vin De Silva, and John C Langford. 2000. A global geometric framework for nonlinear dimensionality reduction. *science* 290, 5500 (2000), 2319–2323.

[29] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–12.

[30] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the gpu microarchitecture to achieve bare-metal performance tuning. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 31–43.