Investigating Parallel APSP Algorithms for Conditioning Graph Problems

PangLi, Yang
New York University
py2236@nyu.edu

Abstract: This paper mainly discusses solving the All-Pairs Shortest Path (APSP) in parallel, restricting the maximum number of vertices that can be visited in each path. We present parallel implementations of Floyd-Warshall's and Johnson's APSP algorithms with OpenMP and discuss approaches to taking advantage of cache locality for performance. We show that these problems can significantly benefit from parallelism, and parallel implementation of Johnson's algorithm consistently outperforms Floyd-Warshall's in our experiments.

I. Introduction

Graphs and related algorithms are commonly used to abstract problems in data analysis, mapping, or communication applications. Moreover, modifying the graph by adding additional layers can reduce complicated issues with more conditional criteria to fundamental graph problems. In this paper, we mainly discuss solving the All-Pairs Shortest Path (APSP) in parallel, given a limitation on the vertices in each path. An obvious challenge is that adding layers increases the number of vertices in the modified graph linearly, making the computational complexity of the algorithms employed exhibit quadratic or even cubic growth. As a result, we focus on parallelizing the computation with OpenMP, testing and comparing the performance with different APSP algorithms and implementations.

Parallel implementations of graph algorithms have been widely discussed over the past decade, from multithread mechanisms [1] to further accelerating through GPU [2]. Yet, their result can not directly apply to layered graph problems because now sets of vertices would have inner dependence. Besides, this characteristic also makes partitioning graphs techniques [3] that help parallel problems hard to apply. As a result, we investigate two mainstream APSP algorithms, Floyd-Warshall [4] and Johnson [5], and test the performance on the parallel implementation. We implement the fundamental algorithms to address standard APSP problems and the optimized ones for our issues. In sequential, Johnson's algorithm has proven to have lower computational complexity. At the same time, Floyd-Warshall's algorithm tends to have better cache efficiency due to the continuous computation on the adjacency matrix. Therefore, which algorithm will perform better in our problem is intriguing.

Our main finding is that the performance of Johnson's algorithm outperforms Floyd-Warshall's algorithm, especially when we have deep layers, which implies more restrictions on the original graph problems. This result comes from the nature that Johnson's algorithm computes the shortest path from each vertex through independent Dikstra's algorithm [6]. Many created vertices in the modification can be omitted. The result also shows significant improvements in performance in solving the APSP problem in parallel.

Section two will discuss the literature survey on developing parallelizing graph algorithms. Section three will discuss our defined problem and proposed idea. Section four will discuss our experimental setup, including machine, compiler, and variables. Section five will detail how we investigate the performance and analyze our findings. Section six provides our conclusion.

II. Literature Survey

The biggest challenge in solving APSP problems is the computation is considerably large. The asymptotic computational complexity is much higher than the quadratic of the number of vertices, making parallel computation necessary [7]. Fortunately, both Floyd-Warshall's and Johnson's contain parts that can benefit from parallelism. Bellman Ford's Algorithm [8], which helps Johnson's algorithm deal with negative weights, can be done in parallel since the iterations are independent. Besides, each vertex can run Dijkstra's algorithm in parallel. On the other hand, Floyd-Warshall's algorithm is a dynamic programming approach operating on a matrix, giving us the similarity to matrix multiplication [9]. Many efforts are dedicated to optimizing the parallelism on it.

Solomonik et al.'s work [10] rescaled the APSP problems to minimize inter-processor communication and maximize temporal data locality. They also introduced a block-cyclic 2D APSP algorithm. This approach allows us to optimize parallel Floyd-Warshall's algorithm as matrix multiplication by dividing the matrix into smaller blocks that better use cache locality. Regarding our discussion, the overhead of modifying the graph with layers can be significant; we are further investigating the performance of using this blocked approach.

Oguzhan et al.'s work [11] developed an approach to solving the APSP problem that parallelizes the inner operations of Dijsktra's algorithm by identifying the nodes that will be inserted into the frontier set and then relaxing those nodes in parallel. Considering we are focusing on multithread, which has fewer computation units than GPU, the overhead of the inner parallel may dominate.

Yang et al.'s work [12] discussed conditional cases of the shortest path problem to find the shortest path passing through a set of vertices. It proposed a novel approach rather than calculating all permutations for given vertices and then finding the shortest one from these permutations, which is too costly. However, the approach is impractical to program for all cases in general and hard to parallelize.

This paper focuses on fundamental APSP problems but conditions on the maximum number of vertices we are allowed to visit in every path. We provide implementations and show the performance gain benefits from parallelism.

III. Proposed Idea

Some adjustments must be made to solve the APSP problem, which is restricted to a given maximum number of vertices a path can contain. One can overwrite the algorithm with early break criteria or brutally calculate all possible permutations of vertices in the path. A simple way that allows us not to change the algorithm is to modify the graph. We introduce a variable n, which stands for the number of layers, such that adding n layers to the original graph can solve the APSP problem with respect to maximum n - 1 vertices on the path.

The idea is to duplicate all vertices, for example, V vertices, to additional layers, so every layer has its own V vertices. Next, transform the edges' destination from the original vertices to corresponding vertices in the next layers while maintaining the same weight. For example, if we have E(u_0, v_0) = w, where u_0, v_0 are vertices in layer 0 (the original one), we now transform it to E(u_0, v_1) = w, where v_1 is the vertex in layer 1. Finally, if we allow the path to have a number of vertices less than the maximum, we can add all vertices and their corresponding vertices in the next layer of new edges with weight 0. For example, adding E(u_0, u_1) = 0 for all vertices in the graph.

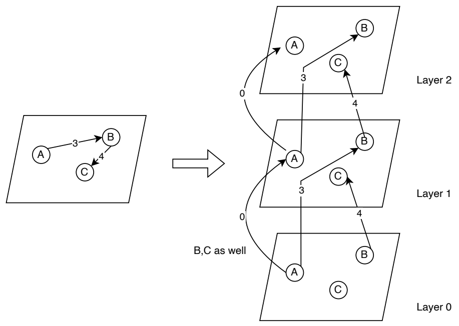


Fig. 1. Adding n = 3 layers

Now, the shortest path we compute from vertex u in layer 0 to vertex v in layer m is the shortest path from u to v under the condition that the path cannot contain over m vertices. For example, we can have a path from A to C if we can have a path with two vertices (C in layer 2), but there is no such path if only one vertex is allowed (C in layer 1).

Through this graph transformation, we can address conditioning graph problems without further modifying algorithms. Yet, it induces some characteristics that need to be considered. First, the graph cannot be easily split and combined to apply partial computation since vertices are dependent because of intermediate vertices. It makes the partitioning approach proposed by Kim et al. [13] inefficient because the extra computation to combine layers grows with the size of the layers we need to add. Therefore, we must handle the dependency when trying to parallelize algorithms for the problem. Another one is that in most cases, we only care about the shortest path from vertices in layer 0, which favors Johnson's algorithm since its operation of vertices is independent.

IV. Experimental Setup

Our experiments are conducted on NYU's crunchy cluster CentOS 7, which has Four AMD Opteron 6272 (2.1 GHz) (64 cores) CPUs and 256 GB RAM. We compile our source code with g++ 9.2 for C++ standard gnu++17 and the latest supported OpenMP.

Our program randomly generates our input graph, which uniformly assigns a weight between two vertices from integer 1 to 30. The connection rate, whether two vertices are connected, is set to be 0.8. All edges are directed. We use an adjacency matrix to represent the graph. Both Johnson's and Floyd-Warshall's algorithms work with negative weights, but here, we only consider positive to omit redundant negative cycle detection.

Our algorithms used for transforming the graph with layers are provided.

```
Algorithm 1: transforming the graph
add layers N
output: new graph NG

for i:0->|G| do
```

```
  for j:0->N*|G| do
    if i=j then
      NG[i][j]=0
    elseIf j>=(i/n+1)*n
        and j<((i/n+2)*n) do
      NG[i][j]=G[i][j-n]
    end
  end
end

for i:0->N*|G| do
  for j:0->N*|G| do
    if i=j then
      NG[i][j]=0
    elseIf j>=(i/n+1)*n
        and j<((i/n+2)*n) do
      NG[i][j]=NG[i-n][j-n]
    end
  end
end
```

V. Experiments and Analysis

A. Floyd-Warshall Experiments

We take sequential Floyd-Warshall's Algorithm as our benchmark because its implementation is concise and works for negative weights. It takes dynamic programming to construct the solution, treating all vertices as intermediate nodes and adding them one by one to test if doing so creates a new shortest path. Obviously, the computational complexity is proportional to the cube of its number of vertices, O(V^3).

```
Algorithm 2: sequential Floyd-Warshall
input: original graph G
output: shortest weight W

for k:0->|G| do
  for i:0->|G| do
    for j:0->|G| do
      W[i][j] = min(W[i][j],
                 W[i][k] + W[k][j])
    end
  end
end
```

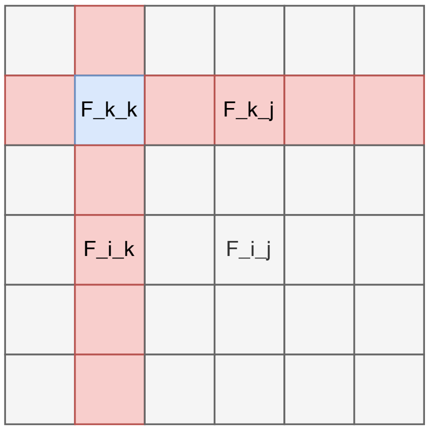
Note that the inner loops, i and j, depend only on k. Therefore, we can parallelize the inner for loop to achieve better performance. We can use the "collapse" clause, but it will not make a notable difference when V is large. We will show that making this naive modification can significantly improve performance.

```
Algorithm 3: parallel Floyd-Warshall
input: original graph G
output: shortest weight W

for k:0->|G| do
  #pragma omp parallel for collapse(2)
  for i:0->|G| do
    for j:0->|G| do
      W[i][j] = min(W[i][j],
                 W[i][k] + W[k][j])
    end
  end
end
```

Similar to matrix multiplication, a large V will induce more cache misses and thus punish performance. It leads us to think about using smaller blocks of the matrix to divide the work and combine them back to the final result, which makes using cache efficient. However, because k makes the inner loops dependent, we need further modification to ensure the correctness of taking advantage of smaller blocks. Venkataraman et al.'s work has proven the correctness of solving blocks in each iteration in a specific order. For each iteration k, we need first to conduct partial Floyd-Warshall computation on the block_k_k, the blue one in Figure 2. Next, compute all horizontal, block_i_k, and vertical blocks, block_k_j, with respect to k, the red ones. Finally, all the rest of the blocks, the grey ones, can be computed independently in parallel.

Fig. 1. Blocked Floyd-Warshall



With this assertion, we can now implement a blocked Floyd-Warshall's algorithm. Note that we might need to add some virtual connect-less vertices to make the graph able to split into blocks evenly. Besides, we use static scheduling with a small chunk size to prevent a thread from being allocated to tasks that do not require computation in the case of i or j = k, which will lower the overall performance by being idle.

```
Algorithm 4: blocked parallel Floyd-Warshall
input: original graph G
partial Floyd-Warshall F
output: shortest weight W

for k:0->|G| do
  F(G,k,k)

  #pragma omp for schedule(static,1)
  for i:0->|G|, i != k do
    F(G,i,k)
    F(G,k,i)

  #pragma omp for collapse(2) \
                  schedule(static,1)
  for i:0->|G|, i != k do
    for j:0->|G|, j != k do
      F(G,i,j)
    end
  end
end
```

B. *Floyd-Warshall Analysis*

First, one straightforward prediction is that the multiple of the number of vertices V and the number of layers L, V*L determines the performance of parallel Floyd-Warshall's algorithm. It is not sensitive to whether the original graph is broader or the layers are deeper. Table 1 demonstrates this prediction.

TABLE 1. ASSESSMENT WITH 64 THREADS

Test Case	**Parameters**		
	V	**L**	**Exec Time (ms)**
Sequential	60	5	3847.39
	30	10	3848.67
	15	20	3848.21
Parallel	60	5	108.94
	30	10	122.79
	15	20	123.95
Blocked Parallel	60	5	294.11
	30	10	272.84
	15	20	279.58

This table contains some critical findings. First, we show significant speedups when using OpenMP to parallelize the algorithms. In this given test setting with 64 threads, we achieve a 31x speedup in the original parallel implementation and a 14x speedup in the blocked one. Second, blocked optimization does not seem to help and even worsens performance. We will further investigate the performance of parallelism based on this finding.

Figure 2 (in log-log scale) plots the execution time (in milliseconds) with respect to the input size V * L. It shows that the sequential implementation grows linearly to the input size, and performance gaps between the parallel and blocked implementation are shrinking. In Figure 3, the blocked implementation outperforms the original parallel one when we test on substantial large input, from 4000 to 5000. This is because the benefit of cache locality now dominates the overhead for applying blocks.

Fig. 2. Floyd-Warshall Exec. Time

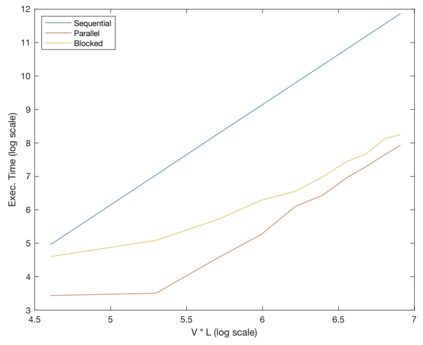


Fig. 3. Large Input Exec. Time

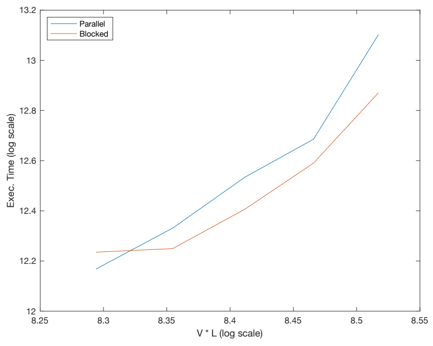
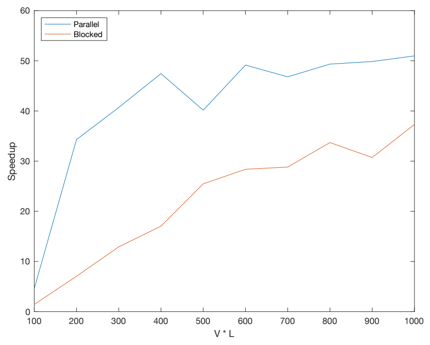


Figure 4 plots the speedup with respect to the input size. It shows rapid growth when we increase the input size from a relatively small number. As the input size grows to a particularly large value, the speedup of the original parallel implementation becomes stable while we still observe the speedup of the blocked one increases. It supports why the blocked implementation favors a larger input size.

Fig. 4. Floyd-Warshall Speedup



C. *Johnson Experiments*

There are some fundamental differences between implementing Johnson's algorithm and Floyd-Warshall's algorithm. First, rather than adjacency matrices, Johnson's algorithm is based on Dijkstra's algorithm, which favors adjacency lists in computation to retrieve a vertex's neighbors in constant time. While transforming adjacency matrices to adjacency lists can benefit from parallelism, it only takes a small proportion of the overall computation cost. Therefore, we only use the map data structure in the C++ Standard Template Library for optimization in our implementation.

Second, Johnson's algorithm relies on Bellman-Ford's algorithm to deal with negative weights through reweighting distances to eliminate negative edges. We can make each vertex in its iteration to generate the shortest distances parallel.

```
Algorithm 5: parallel Bellman-Ford
input: original graph G
source node src
output: shortest distance h

h = int[|G|] <- INT_MAX

h[src] = 0

for k:1->|G| do
  #pragma omp parallel for
  for i:1->|G| do
    for E:G[i] do
      v = E.destination
      w = E.weight+h[i]
      h[v] = min(h[v],w)
    end
  end
end
```

After handling these differences, we can now implement parallel Johnson's algorithm. The fundamental idea is to parallel each Dijkstra computation. Note that it is important to use dynamic scheduling because every vertex depends on its connecting edges and can have varied workloads.

```
Algorithm 6: parallel Johnson
input: original graph G
Bellman-Ford B
Dijkstra D
target size t
output: shortest weight W

G add virtual vertex
h = B(G)

#pragma omp parallel for schedule(dynamic)
for u:0->|G| do
  curr = D(u,G)
  for v:0->|G| do
    idx = |G| - t + v
    W[u][v] = curr[idx] + h[idx] - h[u];
  end
end
```

The optimization comes from the fact that we only care about the shortest distance from vertices on layer 0. Unlike Floyd-Warshall's algorithm, Johnson's algorithm allows us to do Dijkstra in a particular set of vertices without affecting the correctness. This optimization ensures the workload grows linearly with the number of layers in theory, not quadratic. The optimized implementation simply reduces the number of Dijkstra to run. It suggests that using Johnson's algorithm in parallel could be better for the maximum vertices problem we are dealing with. We will further discuss it later.

```
Algorithm 7: optimized Johnson
input: original graph G
Bellman-Ford B
Dijkstra D
target size t
output: shortest weight W

G add virtual vertex
h = B(G)

#pragma omp parallel for schedule(dynamic)
for u:0->t do
  curr = D(u,G)
  for v:0->t do
    idx = |G| - t + v
    W[u][v] = curr[idx] + h[idx] - h[u];
  end
 end
```

D. *Johnson Analysis*

Compared to Floyd-Warshall, we expect that Johnson's algorithm is sensitive to the input size V*L and the number of vertices V. That is, even if we have the same input size, cases with a larger V can cost more in computation. Table 2 provides us with some critical insights.

TABLE 2. ASSESSMENT WITH 64 THREADS

Test Case	Parameters		
	V	L	Exec Time (ms)
Sequential	60	5	758.65
	30	10	620.41
	15	20	409.54
Parallel	60	5	72.89
	30	10	52.284
	15	20	46.72
Optimized	60	5	47.78
	30	10	47.56
	15	20	45.71

It is true that with the same input size, a larger V tends to have more computation cost. The reason why, in optimized implementation, we cannot observe such a result is that some threads are idle. Remember, in the optimized case, we reduce the Dijkstra run only from the vertices in the original graph. While using 64 threads, given vertices V less than 64 will cause additional threads only to help for the Bellman-Ford phrase but not the n-Dijkstra phrase.

Figure 5 (in log-log scale) plots the execution time with respect to the input size V * L. We again observe a significant gain in performance, and the optimized implementation can always perform better as long as the original vertices size is large enough. In Figure 6, we can observe that when we test on a considerably large input, the performance gaps between the original parallel implementation and the optimized one are getting more significant. The result suggests that it is reasonable to keep this optimization in general.

Fig. 5. Johnson Exec. Time

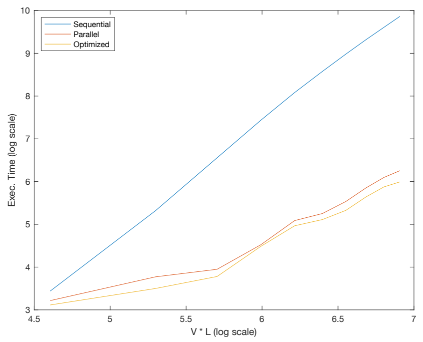


Fig. 6. Large Input Exec. Time

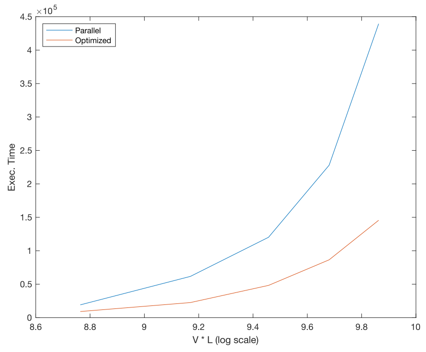
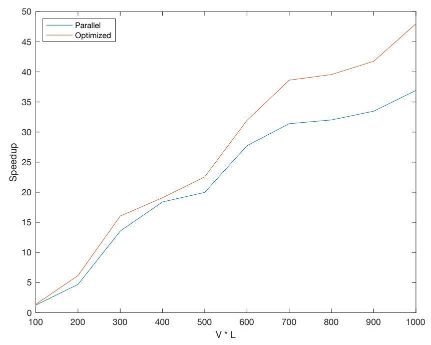


Figure 7 shows the speedup growths with the input size, but at a slower speed than Floyd-Warshall. This is because parallel Johnson's algorithm requires large enough starting vertices to take full advantage of parallel programming.

Fig. 7. Johnson Speedup



E. Comparison

Our initial concern is using Floyd-Warshall's or Johnson's algorithms in parallel to solve APSP problems with conditions on maximum vertices in the path. While Johnson's algorithm has better asymptotic computational complexity, as shown in Figure 8, Floyd-Warshall's algorithm may perform better due to its nature of cache locality because every Dijkstra runs in Johnson's algorithms is independent with little locality between different source vertices.

Fig. 8. Sequential Exec. Time

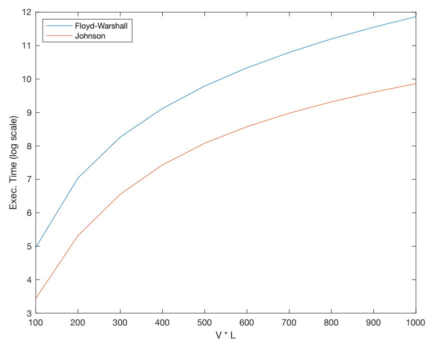
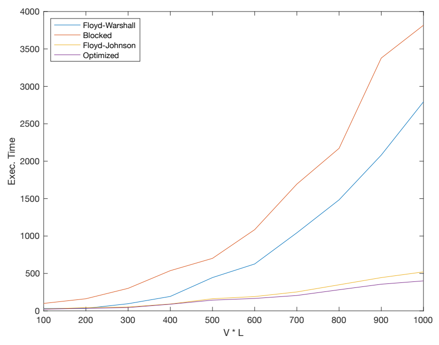


Figure 9 illustrates the comparative performance of Johnson's and Floyd-Warshall's algorithms across various implementations. Notably, both implementations of Johnson's algorithm consistently outperform those of Floyd-Warshall's. Moreover, the performance gaps between the two algorithms tend to widen as the input size increases. As we have shown, both parallel implementations can take advantage of parallelism with a speedup close to 40x under 64 threads. This result supports the theoretical computational complexity that Floyd-Warshall's algorithms are proportioned to the cube of input size while Johnson's algorithm is between cube and quadratic. Because the capable computational units are limited, the underlying algorithm dominates.

Fig. 9. Parallel Exec. Time



This result is not surprising since we discover that taking advantage of cache locality in Floyd-Warshall's algorithm requires additional overhead that parallelism needs a sufficiently large task size to overcome. However, the difference in the underlying algorithms is magnitude by the task size. As a result, Johnson's algorithm presents a better overall performance.

VI. Conclusion

This paper presents several parallel implementations of APSP algorithms with OpenMP. We focus on their performance on conditioning graph problems, particularly on restricting the maximum number of vertices that can be visited in each path. We show that the parallel implementation of Johnson's algorithm consistently outperforms Floyd-Warshall's because taking advantage of cache locality in Floyd-Warshall's algorithm is costly. The optimized Johnson's algorithm implementation can best serve this kind of question, with up to 2x~3x speed up than the basic implementation.

However, the result can change if we introduce GPU for computation and focus on real-world problems. Our parallel implementation of Johnson's algorithm relies on the number of vertices. In some real-world problems, for example, distances between cities, such as vertice entities, can be less than the GPU computational units. Therefore, it fails to take full advantage of parallelism. On the other hand, the blocked Floyd-Warshall implementation can significantly benefit from large amounts of computational units. As we showed, many independent blocks can be computed in parallel. This tradeoff requires further discussion.

References

[1] P. Samer, A. H. Sampaio, A. Milanés and S. Urrutia, "Designing a Multicore Graph Library," 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, Leganes, Spain, 2012, pp. 721-728.

[2] T. Okuyama, F. Ino and K. Hagihara, "A Task Parallel Algorithm for Computing the Costs of All-Pairs Shortest Paths on the CUDA-Compatible GPU," 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications, Sydney, NSW, Australia, 2008, pp. 284-291.

[3] M. H. Alghamdi, L. He, S. Ren and M. Maray, "Efficient Parallel Processing of All-Pairs Shortest Paths on Multicore and GPU Systems," in IEEE Transactions on Consumer Electronics.

[4] Robert W. Floyd. 1962. Algorithm 97: Shortest path. Commun. ACM 5, 6 (June 1962), 345.

[5] Donald B. Johnson. 1977. Efficient Algorithms for Shortest Paths in Sparse Networks. J. ACM 24, 1 (Jan. 1977), 1–13.

[6] Dijkstra, E.W. A note on two problems in connexion with graphs. Numer. Math. 1, 269–271 (1959).

[7] M. Alghamdi, L. He, Y. Zhou and J. Li, "Developing the Parallelization Methods for Finding the All-Pairs Shortest Paths in Distributed Memory Architecture," 2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC), London, UK, 2019, pp. 1-8.

[8] Hajela, Gaurav & Pandey, Manish. (2014). Parallel Implementations for Solving Shortest Path Problem using Bellman-Ford. International Journal of Computer Applications. 95.

[9] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," in IBM Journal of Research and Development, vol. 39, no. 5, pp. 575-582.

[10] E. Solomonik, A. Buluç and J. Demmel, "Minimizing Communication in All-Pairs Shortest Paths," 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, Cambridge, MA, USA, 2013, pp. 548-559.

[11] Taştan, Oğuzhan & Eryüksel, Oğul & Temizel, Alptekin. (2017). Accelerating Johnson's All-Pairs Shortest Paths Algorithm on GPU.

[12] Yajun Yang, Zhongfei Li, Xin Wang, Qinghua Hu, "Finding the Shortest Path with Vertex Constraint over Large Graphs", Complexity, vol. 2019, Article ID 8728245, 13.

[13] Jong Wook Kim, Hyoeun Choi, and Seung-Hee Bae. 2018. Efficient Parallel All-Pairs Shortest Paths Algorithm for Complex Graph Analysis. In Workshop Proceedings of the 47th International Conference on Parallel Processing (ICPP Workshops '18). Association for Computing Machinery, New York, NY, USA, Article 5, 1–10.

[14] Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya. 2004. A blocked all-pairs shortest-paths algorithm. ACM J. Exp. Algorithmics 8 (2003).