Efficient Parallel All-Pairs Shortest Paths Algorithm for Complex Graph Analysis

Jong Wook Kim*
Handong Global University
Pohang, Republic of Korea
po00076@gmail.com

Hyoeun Choi†
Handong Global University
Pohang, Republic of Korea
hyoeun.choi@wmich.edu

Seung-Hee Bae‡
Western Michigan University
Kalamazoo, Michigan
seung-hee.bae@wmich.edu

ABSTRACT

The all-pairs shortest path problem is a classic problem to study characteristics of the given graphs. Though many efficient all-pairs shortest path algorithms have been published, it is still a very expensive computing task, especially with large graph datasets. In this paper, we propose an efficient parallel all-pairs shortest path algorithm based on Peng *et al.*'s fast sequential algorithm on shared-memory parallel environments to achieve faster and more efficient calculation for large-scale real-world networks. Peng *et al.*'s algorithm needs to sort vertices with respect to their degrees. However, it turns out the original algorithm uses less efficient sorting method, which is a significant portion of parallel overhead. Therefore, we also propose an efficient parallel method to sort data within a fixed range, in order to minimize the parallel overhead in our parallel algorithm. The optimized efficient sorting method can be used for general sorting purposes. Our experimental analysis shows that our proposed parallel algorithm achieves very high parallel speedup, even hyper-linear speedup, with real-world test datasets on two different shared-memory multi-core systems.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**; **Shared memory algorithms**; • **Theory of computation** → **Shortest paths**;

KEYWORDS

Parallel Algorithms, Shared-memory parallelism, All-pairs shortest paths

ACM Reference Format:
Jong Wook Kim, Hyoeun Choi, and Seung-Hee Bae. 2018. Efficient Parallel All-Pairs Shortest Paths Algorithm for Complex Graph Analysis. In *ICPP '18 Comp: 47th International Conference on Parallel Processing Companion, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3229710.3229730

*Jong Wook's work for this manuscript was done at Western Michigan University.
†Hyoeun's work for this manuscript was done at Western Michigan University.
‡The corresponding author.

ICPP '18 Comp, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6523-9/18/08…$15.00
https://doi.org/10.1145/3229710.3229730

1 INTRODUCTION

Finding the shortest paths is one of the most widely studied classic graph problems, due to its broad applicability to our life. We can easily find applications related to shortest-path problems in many domains, such as computer science, transportation research, game development, and network analysis, to name a few.

Recently, complex network analysis has been studied in many research area, since many real-world systems can be represented as complex networks. Furthermore, those graphs are emerging and are becoming much larger these days. Web-graphs, social network graphs, scholarly collaboration networks, and biological sequence interaction networks are some typical examples of the large-scale complex networks. Finding shortest-paths among the vertices in these large-scale networks is critical to study the characteristics of those networks.

Many algorithms have been proposed for solving all-pairs shortest-path (APSP) problem efficiently since Floyd-Warshall algorithm published [10]. Among them, the fast APSP algorithm, which was proposed by Peng *et al.* [14], is one of the *state-of-the-art* algorithms. Its empirical time-complexity is $O(n^{2.4})$, and they also proposed optimization methods from their basic algorithm based on the *scale-free* feature of the large-scale complex graphs [14]. Though the Peng *et al.*'s algorithm is faster than others, their algorithms are still implemented in sequential and it is an expensive computing task, especially with larger graphs.

In this paper, we would like to implement an efficient parallel APSP algorithm based on Peng *et al.*'s state-of-the-art APSP algorithm under shared-memory parallel environments. We find some parallel overhead from our initial parallel algorithm, and we minimize the overhead to make our proposed parallel algorithm more efficient. Overall, our proposed parallel APSP algorithm, called **ParAPSP**, performs linear speedup with most of the tested datasets, even *hyper-linear* speedup in some cases. In addition to high parallel efficiency, our parallel algorithm provides the exact same outputs of the Peng *et al.*'s algorithm, which are the precise APSP solutions.

We summarize the main contributions of this paper as follows:

- We propose an efficient parallel APSP algorithm on shared memory environments.
- We optimize the ordering process in several orders of magnitudes.
- We propose an efficient parallel ordering procedure, which can be used for general purpose.
- We evaluate the performance of the proposed parallel APSP solution to show its high efficiency.

Algorithm 1 Pseudo code for the modified Dijkstra's algorithm [14]

input: Graph $G = (V, E)$, source s, weight matrix L, distance matrix D, vector *flag*
$D[s, s] = 0$
$Q = s$
while Q is not empty **do**
  $t = DeQueue(Q)$
  if $flag[t] = 1$ **then**
    for each vertex $v \in V$ **do**
      if $D[s, t] + D[t, v] < D[s, v]$ **then**
        $D[s, v] = D[s, t] + D[t, v]$
      end if
    end for
  else
    for each edge (t, v) outgoing from t **do**
      if $D[s, t] + L[t, v] < D[s, v]$ **then**
        $D[s, v] = D[s, t] + L[t, v]$
        Enqueue(Q,v)
      end if
    end for
  end if
end while
$flag[s] = 1$
output: updated distance matrix D, updated vector *flag*

The rest of the paper is organized as follows: Section 2 briefly explains the sequential state-of-the-art algorithm [14]. Section 3 describes processes of parallelizing the basic and optimized algorithms. We discuss the optimization of the descending ordering steps in Section 4. In Section 5, we present the experimental evaluation of the proposed ParAPSP with real-world datasets on two tested environments. Section 6 discusses the related work of this paper followed by the conclusion and the future work in Section 7.

2 BACKGROUND

A classic algorithm for solving APSP problem is the Floyd-Warshall algorithm [10], whose time complexity is $O(n^3)$ where n is the number of vertices. Alternatively, we can find an APSP solution by applying single-source shortest-path (SSSP) algorithm from each of the vertices in the graph. There are two well-known algorithms for solving SSSP: the Bellman-Ford [4] and the Dijkstra's [8] algorithms. As the Bellman-Ford algorithm can handle graphs with some negative weighted edges, the Bellman-Ford algorithm is more versatile than the Dijkstra's algorithm. However, the Dijkstra's algorithm is faster than the Bellman-Ford algorithm for the same problem, since the Dijkstra's algorithm takes $O(n^2)$ but Bellman-Ford takes $O(nm)$, where n is the number of vertices, m is the number of edges, and $n \ll m$ most of the real-world graphs.

Many algorithms have been proposed to solve APSP problem more efficiently [15, 16]. In 2005, Chan [5] published an algorithm, which can solve APSP problem in time complexity of $O(n^3/\log n)$. Peng *et al.* [14] proposed a fast APSP algorithm by applying dynamic programming method to Dijkstra's algorithm on the APSP problem. The Peng *et al.*'s algorithm runs the modified Dijkstra's algorithm iteratively with respect to each vertex as a source. The modified Dijkstra's algorithm utilizes previously obtained information to expedite the search process. Peng *et al.* showed that their proposed algorithm works much better for both of the Erdös and Rényi (ER) *random* graph model [9] and the *scale-free* Albert-Barabási (AB) network model [2]. Though Peng *et al.* [14] didn't provide a theoretical analysis of the time complexity of their proposed algorithm, they empirically showed that their basic proposed algorithm performed in $O(n^{2.4})$ for complex scale-free networks. Furthermore, they proposed two optimized algorithms, which are faster than the basic proposed algorithm. In our experiments, the optimized algorithm shows two to four times faster performance than the basic algorithm. In this paper, we propose an efficient parallel APSP algorithm, which is based on Peng *et al.*'s algorithm, on shared-memory parallel environments by using OpenMP [7]. In this section, we briefly introduce the fast algorithm from Peng *et al.*'s paper [14].

Algorithm 2 Basic Algorithm for APSP Problem [14]

input: Graph $G = (V, E)$ and Weight matrix L
for each vertex pair (u,v) **do**
  $D[u, v] = \infty$
end for
for i = 1 to n **do**
  flag[i] = 0
end for
for i = 1 to n **do**
  call the modified Dijkstra's procedure (Algorithm 1)
  using the source of index i
end for
output: distance matrix D

2.1 Sequential Algorithm 1 - Basic Algorithm

Dijkstra's algorithm is a simple and efficient algorithm based on a breadth-first search approach to find all shortest paths from a single source to all other vertices in a given graph. A naïve approach to finding an APSP solution is to apply Dijkstra's algorithm on each vertex iteratively, and it would take $O(n^3)$ as its time complexity. Though the naïve approach is simple and intuitive, we could have a better algorithm if we utilize the shortest paths information obtained in previous iterations.

Peng *et al.* [14] proposed a modified Dijkstra's algorithm, which utilizes the information obtained in the previous iterations in the middle of the procedure as in Algorithm 1 so that they can directly use the already known shortest paths information without recalculating them. They also proved the correctness of the modified Dijkstra's algorithm in their paper. For details of the modified Dijkstra's algorithm, please refer to Peng *et al.*'s paper [14]. Algorithm 2 illustrates the pseudo-code of the basic algorithm for APSP problem, which proposed by Peng *et al.* [14].

Algorithm 3 Optimized Algorithm for APSP Problem [14]

```
1: input: Graph G = (V, E), Weight matrix L, and ratio r where 0.0 < r ≤ 1.0
2: initialize D matrix and flag vector as in Algorithm 2
3: for i = 1 to n do
4:    order[i] = i
5: end for
6: for i = 1 to r × n do
7:    for j = i + 1 to n do
8:       if degree[order[j]] > degree[order[i]] then
9:          swap(order[j], order[i])
10:      end if
11:   end for
12: end for
13: for i = 1 to n do
14:   call the modified Dijkstra's procedure (Algorithm 1)
15:   using the source of index order[i]
16: end for
17: output: distance matrix D
```

2.2 Sequential Algorithm 2 - Optimized Algorithm

Many real-world networks, such as web-graphs, social networks, and citation networks, are complex networks. Two well-known properties of the complex networks are so-called "small-world" [18] and "scale-free" [3]. The *scale-free* feature means that the degree distribution of a complex network follows a scale-free power-law distribution. In other words, very small number of vertices are high-degree vertices and most of the vertices are low-degree. Therefore, the connectivity of the network is dominated by those high-degree vertices.

Since few vertices in a complex graph have a large number of neighbors, those high-degree vertices could be intermediate vertices of shortest paths of other vertices in high probability. Thus, if we compute the shortest paths from these high-degree vertices as early as possible, those computed shortest paths could be used maximally in the modified Dijkstra's algorithm as shown in Algorithm 1. Peng *et al.* proposed the optimized algorithm based on the *scale-free* property of the complex networks as mentioned above. They added a new step, which sorts vertices in descending order of the degree of each vertex, into the optimized algorithm, and the optimized algorithm calls the modified Dijkstra's algorithm with respect to the sorted order of source vertices. Algorithm 3 illustrates the optimized APSP algorithm. Refer to the details of the optimized algorithm in the original paper [14].

They also proposed an adaptive optimization algorithm, which adapts the *order* array as iteration goes by giving more priority to the vertices, which were actually in the middle of shortest paths of two other vertices. In this paper, however, we have parallelized the optimized algorithm shown in Algorithm 3 instead of the adaptive optimized algorithm for two reasons: 1) the performance gain of the adaptive optimized algorithm compared to the optimized algorithm is relatively small in their experimental analysis, and 2) the adaptive optimized ordering process is inherently dependent to the previous iteration so that it results in more complex and inefficient parallel implementation. You can find the details of the adaptive optimization algorithm in Peng *et al.*'s paper [14].

3 PARALLEL APSP ALGORITHMS

In Section 2, we briefly summarize a fast sequential algorithm for solving APSP problem by Peng *et al.* [14], and they empirically showed that the algorithm's time complexity is $O(n^{2.4})$ based on their linear regression result. To the best of our knowledge, it is one of the fastest solutions for the APSP problem.

Though it is a fast algorithm for APSP problem, it still takes very long as input graph size increases. For instance, it takes several hours to run the algorithm in sequential for "Flickr" dataset in Table 2, as shown in Section 5, so an efficient parallel implementation of the algorithm is essential to get an APSP solution for bigger data fastly.

In this paper, we would like to propose an efficient parallel implementation of Peng *et al.*'s APSP algorithm so that we can fully utilize all of the cores in multi-core systems, which are ubiquitous in these days, in order to solve the APSP problem in much shorter time than sequential implementation. In this section, we explain our design of parallel implementation. We use *OpenMP* [7] API for our parallel APSP algorithm under shared-memory parallel environments.

3.1 Parallel Algorithm 1 - ParAlg1

When we implement a parallel APSP solution based on Peng *et al.*'s algorithm, we could consider two parallel design approaches: 1) implement a parallel version of the modified Dijkstra's algorithm in Algorithm 1, and run it iteratively with each vertex in a given graph as a source; 2) run the modified Dijkstra's algorithm with multiple sources at the same time, until all of the vertices are used as sources of the SSSP solution. For the former approach, we can parallelize Algorithm 1 by running in parallel both of the *for-statements* at line 7 and line 13 in Algorithm 1. However, the possible parallel runs of the second for-statement at line 13 is limited by the degree of vertex t, which could be very small for the most of the vertices in complex networks. Furthermore, there could be a race condition during the enqueuing operation at line 16, so we should handle it by an appropriate synchronization method, which results in additional parallel overheads. Thus, we selected the latter approach for our parallel implementation, since each SSSP run is independent so we can call multiple SSSP runs in parallel without complex coordination, and we can achieve high efficiency if we are able to accomplish good *load-balance*.

We can parallelize Peng *et al.*'s basic algorithm in Algorithm 2 by using OpenMP's **parallel-for** directives on each for-loops in Algorithm 2. The parallel basic algorithm shows good performance in terms of parallel speedup/efficiency as shown in Section 5. Hereafter, we named it **ParAlg1**.

3.2 Parallel Optimized Algorithm - ParAlg2

Though **ParAlg1** shows very good parallel speedup, it is based on basic APSP algorithm of Peng *et al.*. They also proposed the optimized APSP algorithm which shows better performance than the basic algorithm, especially on complex networks [14]. In this

Algorithm 4 Parallelization of the Peng *et al.*'s Optimized Algorithm for APSP Problem - **ParAlg2**

1: **input:** Graph $G = (V, E)$, Weight matrix L
2: initialize D matrix and *flag* vector as in Algorithm 2
3: generate order[] array with vertexIDs w.r.t. the descending order of degree of vertices.
4: **#pragma omp parallel for *schedule*(dynamic, 1)**
5: **for** i = 1 to n **do**
6: call the modified Dijkstra's procedure (Algorithm 1)
7: using the source of index order[i]
8: **end for**
9: **output:** distance matrix D

section, we describe our parallel implementation of Peng *et al.*'s optimized APSP algorithm.

The main difference between the basic algorithm and the optimized algorithm is the ordering procedure for determining the order of source vertices, as shown in Algorithm 2 and Algorithm 3. We need to parallelize for the ordering procedure, which is between line 6 and line 12 in Algorithm 3. However, this procedure is hard to parallelize efficiently. For the ordering procedure, they utilize a selection sort, which makes a *loop-carried dependency* inherently between each iteration. For instance, an i-th iteration is depended on the result of the $(i-1)$-th iteration in the ordering procedure between line 6 and line 12 in Algorithm 3. Therefore, we cannot run in parallel the ordering procedure without changing the procedure. Here, we leave the algorithm as used in Peng *et al.*'s original paper, so the ordering procedure is run in sequential. Note that the time complexity of the ordering procedure is $O(n^2)$, which could be a significant portion of the parallel overhead. We explore how to improve the ordering procedure in Section 4.

We can parallelize the iterative Dijkstra's procedure in the last for-loop in Algorithm 3 as similar as in ParAlg1 by using a *parallel-for* directive in OpenMP. However, the scheduling scheme is much more critical in parallel optimized APSP solution than in parallel basic APSP solution, since the order of source vertices of calling iterative Dijkstra's algorithm is the key of the optimization.

We experimented the effect of scheduling scheme with a small graph, **ca-HepPh** dataset[1] in SNAP network data repository [13], which has 12008 vertices and 118521 edges. We tested three different scheduling schemes: default (*block-partitioning*) scheme, *static-cyclic* scheme, and *dynamic-cyclic* scheme. By using *schedule* clause of the parallel-for directive, we can specify an appropriate scheduling scheme: '*schedule*(static, 1)' for a static-cyclic scheme, and '*schedule*(dynamic, 1)' for a dynamic-cyclic scheduling scheme.

Figure 1 shows the parallel performance of the parallel optimized algorithm on the **ca-HepPh** dataset with respect to the scheduling schemes. As shown in Figure 1, static and dynamic cyclic scheduling schemes outperform the default block partitioning scheme. In the optimized APSP algorithm, the execution order of the iterative Dijkstra's algorithm affects the overall performance significantly, so it is predictable that both the static and dynamic cyclic scheduling schemes work very well. Between dynamic-cyclic and

[1] http://snap.stanford.edu/data/ca-HepPh.html

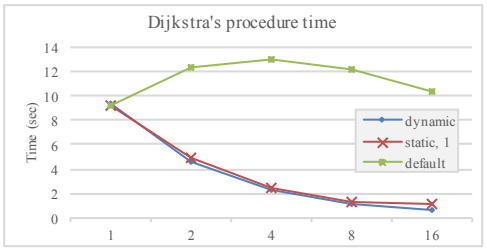


Figure 1: The effect of the scheduling scheme in ParAlg2. Note that x-axis represents the number of threads.

static-cyclic scheduling schemes, dynamic-cyclic schemes slightly outperforms static-cyclic scheme. The main difference between those two scheduling schemes is whether the execution order of the iterative Dijkstra's algorithm is exactly same as the order found by the previous ordering procedure in Algorithm 3 or not. The dynamic-cyclic scheduling scheme guarantees that the execution order is same as the order found by the ordering procedure, but the static-cyclic scheme doesn't guarantee it though its execution order would be close to the result of the ordering procedure. Based on the scheduling execution mentioned above, we decided to use *dynamic-cyclic* scheduling scheme in our parallel algorithm. Algorithm 4 describes the parallelized version of the Peng *et al.*'s optimized APSP algorithm, and we call the algorithm **ParAlg2**, hereafter.

As shown in Section 5, ParAlg2 much better than ParAlg1. In our experiment, ParAlg2 is two to four times faster than ParAlg1. However, ParAlg2 shows less parallel efficiency due to the sequential ordering procedure for finding the optimized order of source vertices. In Section 4, we optimize the ordering procedure.

4 OPTIMIZING THE ORDERING PROCEDURE

Despite the significant improvement of the elapsed time compared to the sequential optimized algorithm, ParAlg2 has an obvious parallel overhead; the ordering procedure, which is $O(n^2)$ time-complexity, remains in sequential. In this section, we explore various methods to find a much more efficient ordering procedure.

By utilizing the simple but critical fact that the maximum degree of any given graph ($G = (V, E)$) is limited by a factor of the number of vertices, where $0 \leq degree[v] \leq n, \forall v \in V$, we can use a bucket-sort based method for the ordering procedure instead of the bubble sort so that we can reduce the time complexity of the ordering procedure from $O(n^2)$ to $O(n)$.

4.1 Parallel Bucket Ordering Procedure - ParBuckets

Initially, we assumed that an approximately sorted order would be good enough to get some benefit of the optimized algorithm. Thus, we used a bucketing method instead of exact bucket sort by using a fixed number of buckets. We tested with 100 widths between the minimum (**min**) and maximum (**max**) degrees of the given graph. Since we used 100 ranges with inclusive minimum and maximum degree, the exact number of buckets will be 101. For a vertex v, we can find an index of the appropriate bucket of the vertex v with respect to the degree of the vertex v ($deg(v)$), **min**, and **max** degree

Algorithm 5 Parallel Bucket Ordering Procedure - **ParBuckets**

```
1: Find max/min degree of the given graph.
2: Initialize the list of buckets and corresponding locks.
3: #pragma omp parallel for
4: for i = 1 to n do
5:    binNo = findBin(degree[i], max, min) as in (1).
6:    omp_set_lock(&lock[binNo])
7:    add vertex i to bucketList[binNo]
8:    omp_unset_lock(&lock[binNo])
9: end for
10: count = 1
11: for j = 100 to 0 do
12:    for each vertex v ∈ bucketList[j] do
13:       order[count] = v
14:       count++
15:    end for
16: end for
```

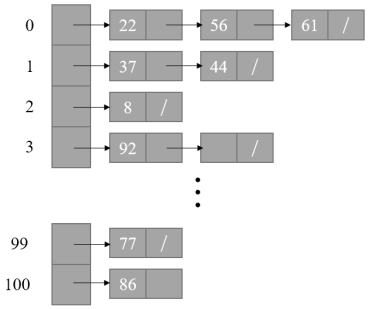


Figure 2: An illustration of the list of buckets

by using a simple equation as in (1):

$$\left\lceil 100 \times \frac{deg(v) - min}{max - min} \right\rceil. \quad (1)$$

This equation returns an integer number that is in the range from 0 to 100 based on a given degree of a vertex.

Figure 2 demonstrates our design of the list of buckets to be used in parallel bucket ordering procedure. In Algorithm 5, *bucketList* represents the list of buckets to store the vertices depending on their degrees. The *bucketList*[n] contains vertices that are assigned to the n-th bucket.

In addition to reducing the time complexity from $O(n^2)$ to $O(n)$, bucket sort eliminates the loop-carried dependency problem, since each vertex can find its corresponding bucket without regard to other vertices' bucket assignment. Therefore, we can try to parallelize the ordering procedure as well.

Since all of the threads can access the shared list of buckets, *race-condition* would occur when multiple threads try to add vertices to the same bucket. Therefore, we used a lock for each bucket to avoid a possible race condition. When a thread needs to add a vertex to a bucket, the thread should acquire the lock for the bucket before adding the vertex, and after the addition of the vertex is done, it should release the lock for the bucket so other threads can access it.

Table 1: Comparison of the parallel elapsed time of the ordering scheme of ParAlg2 and ParBuckets algorithms with WordNet dataset in Table 2 in milli-seconds.

# of Threads	1	2	4	8	16
ParAlg2	46,847	46,858	46,882	46,851	46,830
parBuckets	10	46	102	132	166

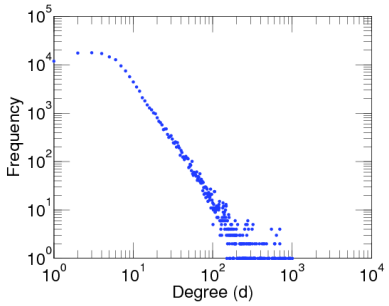


Figure 3: The degree distribution of the WordNet graph in Table 2.

The ordering procedure in Algorithm 4 is replaced with Algorithm 5 in Parallel Bucket Ordering Procedure, hereafter called **ParBuckets**. As shown in Table 1, the ParBuckets outperforms the original sequential ordering procedure used in Algorithm 4 by the several orders of the magnitude.

4.2 Parallel Maximum Buckets Ordering - ParMax

Although **ParBuckets** provides several orders of magnitudes of the speedup compared to **ParAlg2** in the ordering procedure time as shown in Table 1, it still has some issues we should take care of. First, the approximated ordering results affect the elapsed time of the iterative Dijkstra's calls as in Figure 5. Interestingly, the parallel performance of ParBuckets ordering runtimes gets worse as the number of threads increases.

Figure 5 proves that it is critical to find the precise descending order for getting the maximum benefit of the optimized APSP algorithm, and an approximated descending order by coarse-grained bucketing is not enough. We also tested with 1000 buckets instead of 100, and it still shows the performance gap compared to the precise order result although the gap is reduced. Therefore, we decided to increase the number of buckets up to the (maximum degree + 1) of a given graph, which is still limited by the number of vertices. By setting the number of buckets to the (**max + 1**), the proposed bucket sort guarantees to find the exact descending order with respect to the degrees of vertices in $O(n)$ so we could eliminate the performance gap due to the approximated ordering output, and we can find the index of the appropriate bucket without computation of (1).

The second problem is that the ParBuckets ordering procedure performs worse as the number of threads increases. We found that the reason for the problem is the power-law degree distribution of

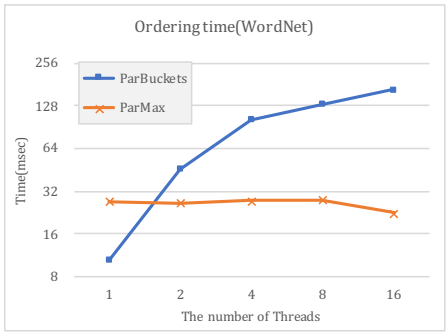


Figure 4: The ordering time comparison between ParBuckets and ParMax.

the tested real-world datasets and corresponding significant lock-contentions in lower buckets, which would actually increase as the number of threads increases.

Figure 3 describes the degree distribution of **WordNet** graph in Table 2, which is available at KONECT[2] [12] network collection. As shown in Figure 3, most of the vertices have very low degrees, so they are assigned to the few lowest buckets. Most of the real-world graphs, including our tested graphs in Table 2, are scale-free graphs which follows the power-law degree distribution [2]. Thus, most of the vertices in real-world graphs are the low-degree vertices so those vertices are assigned to the same few buckets. This will cause lots of the lock-contentions, which result in the worse performance with higher parallelism.

To solve the heavy lock-contention problem, we designed our parallel bucket sort with two separate for-loops: In the first for-loop, all of the threads simultaneously assign each scheduled vertex to a corresponding bucket only if the degree of the vertex is higher than a threshold. In our test, we set the threshold to 1% of the maximum degree so if vertices whose degrees are in the higher 99% then those vertices will be assigned to the corresponding buckets in parallel. The second for-loop is used to assign the remaining vertices with the degrees less than the threshold in sequential for the purpose of avoiding heavy lock-contentions in those lower degree buckets. By using an additional boolean array, named *added*, we are able to run the second for-loop without unnecessary checking of the degrees of vertices which were already added during the first for-loop.

In the above paragraphs, we described our solution, hereafter called **ParMax**, for the problems of ParBuckets. Algorithm 6 illustrates the **ParMax** parallel ordering procedure. As shown in Figure 4, ParMax performs faster ordering runtime as the number of threads gets bigger. The ParMax algorithm guarantees to generate a precise descending order with respect to the degrees of the vertices, so we expected that the iterative SSSP procedure time would be similar to the ParAlg2 algorithm, which is shown in Figure 5.

4.3 Parallel Ordering Procedure with Multiple Lists of Buckets - MultiLists

Though ParMax reduces the overhead of locks significantly compared to the ParBuckets algorithm, it still has only a marginal

[2]KONECT (the Koblenz Network Collection) at http://konect.uni-koblenz.de/networks/

Algorithm 6 Parallel Maximum Bucket Ordering Procedure - **ParMax**

```
1:  Find max/min degree of the given graph.
2:  Initialize the list of buckets and corresponding locks.
3:  #pragma omp parallel for
4:  for i = 1 to n do
5:      if degree[i] >= max ×0.01 then
6:          omp_set_lock(&lock[degree[i]])
7:          add vertex i to bucketList[degree[i]]
8:          omp_unset_lock(&lock[degree[i]])
9:          added[i] = true
10:     end if
11: end for
12: for i = 1 to n do
13:     if added[i] = false then
14:         add vertex i to bucketList[degree[i]]
15:     end if
16: end for
17: count = 1
18: for j = max to 0 do
19:     for each vertex v ∈ bucketList[j] do
20:         order[count] = v
21:         count++
22:     end for
23: end for
```

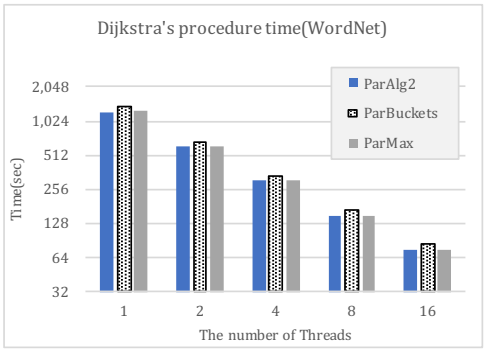


Figure 5: The parallel elapsed running time of Dijkstra algorithm part comparison between ParAlg2, ParBuckets, and ParMax ordering procedures.

speedup in terms of the ordering runtime, as in Figure 4. Most of the vertices, whose degrees are below the threshold, are assigned to the corresponding buckets in sequential. In addition, the ParMax algorithm requires additional for-loop which adds another $O(n)$ time-complexity, and the ParMax algorithm might still put some vertices into the same buckets at the same time in the first parallel for-loop, so it was still inevitable to use locks for avoiding race-condition in the first for-loop.

In order to remove those inherent parallel overheads in the ParMax algorithm, we propose another ordering algorithm which utilizes multiple lists of buckets, called **MultiLists**; each thread has its own list of buckets (*bucketList*) so that each thread can put vertices

Algorithm 7 Parallel MultiLists Ordering Procedure - **MultiLists**

1: Find **max**/**min** degree of the given graph.
2: Initialize the multiple lists of buckets for all of the threads.
3: #pragma omp parallel
4: threadID = omp_get_num_thread()
5: #pragma omp for
6: **for** i = 1 to n **do**
7: add vertex i to *bucketLists*[threadID][degree[i]]
8: **end for**
9: find corresponding starting position of *order*[] array for each bucket and store those positions in *orderPos*[][] two-dimensional array.
10: parRatio = 0.1
11: **for** deg = 0 to **max** $\times parRatio$ **do**
12: #pragma omp parallel for
13: **for** tID = 0 to numThreads **do**
14: index = *orderPos*[tID][deg]
15: **for** $v \in$ *bucketLists*[tID][deg] **do**
16: order[index++] = v
17: **end for**
18: **end for**
19: **end for**
20: add higher degree vertices, where $degree[j] >= \textbf{max} \times parRatio$ for each vertex j, into *order*[] array in sequential.

into the appropriate buckets in its own bucketList without using locks. After the local ordering procedure is finished in parallel, we combine those locally sorted buckets and generate a final global ordering result in the *order* array. The proposed **MultiLists** ordering algorithm, we can produce the exact descending degree-order of the vertices without using locks.

The proposed **MultiLists** algorithm consists of two phases: the first phase is a partial ordering part, and the second phase is the procedure of generating a global *order* array with descending order of the degree of vertices from all of the sorted bucketLists. The first phase can be easily parallelized, since each thread has its own list of buckets so it can update its local buckets independently. We can also implement the second phase in parallel with a simple additional step, which is to calculate the corresponding location of the global *order* array for each bucket of all of the bucketLists. The pseudocode of the MultiLists algorithm is described in Algorithm 7. The first phase is shown between line 3 and line 9 in Algorithm 7, and the second phase is depicted between line 10 and line 20.

Note that, in the second phase, we generate the global *order* array in parallel only the vertices in the small portion of degrees (here we used $\textbf{max} \times 0.1$) as shown in Algorithm 7 from line 11 to line 19, and the higher-degree vertices are added into the *order* array in sequential. Two main reasons of our parallel design of the second phase are as follows: 1) almost 99% of the vertices belong to those small-degree ranges ($0.1 \times \textbf{max}$), while the remaining 1% of the vertices are spread within the 90% of overall degrees, 2) so, if we had parallelized all of the degree ranges, it would have triggered a lot of *false sharing* within the broad higher-degree ranges for writing in the global *order* array.

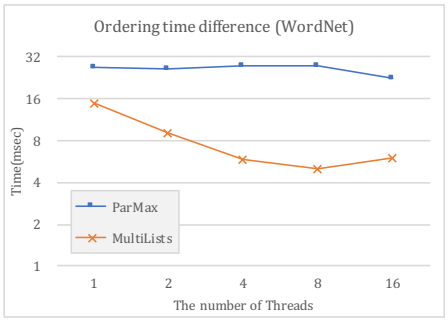


Figure 6: The ordering time comparison between ParMax and MultiLists methods.

Algorithm 8 Proposed Parallel APSP Solution - **ParAPSP**

1: **input:** Graph $G = (V, E)$, Weight matrix L, and ratio r where $0.0 < r \leq 1.0$
2: initialize D matrix and *flag* vector as in Algorithm 2
3: find the descending sorted order of vertices w.r.t. the degrees by the MultiLists ordering procedure in Algorithm 7
4: #pragma omp parallel for *schedule*(dynamic, 1)
5: **for** i = 1 to n **do**
6: call the modified Dijkstra's procedure (Algorithm 1)
7: using the source of index order[i]
8: **end for**
9: **output:** distance matrix D

Figure 6 describes that the proposed **MultiLists** ordering algorithm outperforms the **ParMax** algorithm. The MultiLists algorithm performs better as the number of threads increases, though the parallel performance gets slightly worse when the number of threads increases from 8-thread to 16-thread. Since we suspected that the small number of vertices would be the main reason of the poor parallel performance of the MultiLists algorithm with higher parallel units in Figure 6, we tested only the ordering algorithm part with much larger real-world graph datasets, *soc-Pokec*[3] and *soc-LiveJournal1*[4], from SNAP graph data repository [13]. *soc-Pokec* and *soc-LiveJournal1* have about 1.6 million and 4.8 million vertices, correspondingly. In the ordering test with a larger number of vertices, MultiLists shows the better performance with the more number of threads. Note that the proposed parallel MultiLists ordering algorithm can be used in general parallel sorting problem when keys are in limited ranges.

Our proposed parallel APSP solution, which is named **ParAPSP**, is shown in Algorithm 8. Our proposed parallel algorithm achieves high efficiency and even *hyper-linear* efficiency in some cases, as shown in Section 5.

5 EXPERIMENTAL ANALYSIS

In this section, we evaluate the parallel performance of our proposed parallel algorithm, ParAPSP. Since Peng *et al.* haven't provided

[3] http://snap.stanford.edu/data/soc-pokec.html
[4] http://snap.stanford.edu/data/soc-LiveJournal1.html

Table 2: Salient information of tested real-world graph datasets.

Name	Type	Vertex	Edge
ego-Twitter	Directed	81,306	1,768,149
Livemocha	Undirected	104,103	2,193,083
Flickr	Undirected	105,938	2,316,948
WordNet	Undirected	146,005	656,999
sx-superuser	Directed	194,085	1,443,339

the source code of their state-of-the-art fast APSP algorithm, we have implemented our own version of their algorithms based on the paper [14]. We found that the APSP solution of our proposed **ParAPSP** algorithm is exactly same as the output of sequential runs, so we focus on how efficiently we can find the same solution by our proposed parallel algorithm in our tested shared-memory parallel environments. Our results in this section show that the proposed algorithm achieves almost *linear* speedup, with some datasets even *hyper-linear* speedup.

5.1 Experimental Environment

Datasets. In order to evaluate the parallel performance of our proposed parallel algorithm, we used various real-world graphs. Table 2 shows the salient information of the tested graph datasets in our paper, you can find the details of those graph datasets from SNAP [13] large network dataset collection[5] and KONECT [12] network collection website[6]. Note that we are working on shared-memory parallel environments and the APSP algorithm requires $O(n^2)$ memory space to store the all-pairs shortest-path result, so we are limited to choose experimental datasets with respect to the available main memory size in our tested environments.

Test Machines. We have used two shared-memory multicore systems to perform our parallel experiments with the proposed algorithms. **Machine-I** has dual Intel Xeon E5-2670 eight-core processors (total 16 cores) with 2.6 GHz and 20MB of cache memory and 128 GB of main memory. **Machine-II** has quad Intel Xeon E5-4640 eight-core processors (total 32 cores) with 2.4 GHz and 20MB of cache memory and 256 GB of main memory. The operating system for both **Machine-I** and **Machine-II** is CentOS version 6. Although the CPUs in both test machines support *hyper-threading* technology, we disabled the hyper-threading option in our experimental analysis. Note that the experimental results in this section are based on the average of 10 runs of each test, unless we specially mention about the test specification.

5.2 Basic APSP algorithm vs. Optimized APSP Algorithm

First, we compare our two parallel versions of the two original APSP algorithms proposed by Peng *et al.* Figure 7 shows the elapsed parallel runtimes of the **ParAlg1** and the **ParAlg2** algorithms with **Flickr** dataset in Table 2 with respect to the number of threads on the **Machine-I**. Note that the y-axis of Figure 7 is log-scale.

[5]http://snap.stanford.edu/data/
[6]http://konect.uni-koblenz.de/networks/

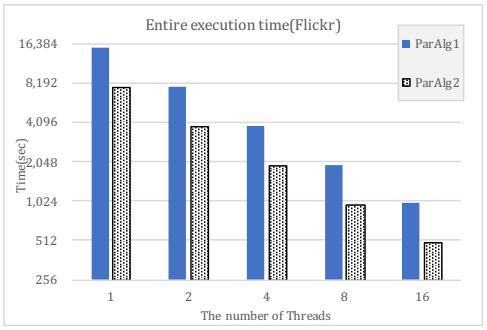


Figure 7: The comparison of the parallel elapsed running time of the ParAlg1 and ParAlg2 algorithms.

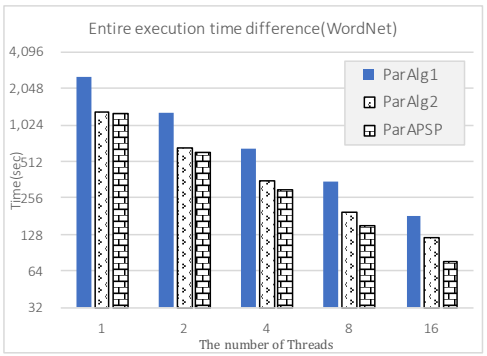


Figure 8: The overall elapsed running time of ParAlg1, ParAlg2, and the proposed ParAPSP algorithms.

Although we present the experimental results with Flickr dataset due to the duplicity of the graphs, all of the experimental results with other tested datasets show the same pattern of Figure 7. Both algorithms show linear speedup, such that the runtime is reduced as a half as the number of threads doubles in Figure 7. In the comparison of ParAlg1 and ParAlg2, ParAlg2 is almost twice faster than ParAlg1 with all the number of threads as shown in Figure 7. For over all of the test datasets, ParAlg2 always outperforms ParAlg1 and it shows two to four times more efficient than ParAlg1.

5.3 Optimized Parallel Ordering Procedure

Though **ParAlg2** outperforms **ParAlg1** by two to four factors of the overall elapsed runtime, the inherent sequential portion of the ordering procedure, which takes $O(n^2)$ time-complexity, in the ParAlg2 algorithm was a significant parallel overhead. For example, the original sequential ordering procedure took about 45 seconds when we ran ParAlg2 with *WordNet* dataset on **Machine-I**. While 45 seconds out of 1300 seconds, which is the elapsed runtime of 1-thread runs with ParAlg2, sounds insignificant, 45 seconds out of 122 seconds by 16-thread runs is more than one-third of the overall elapsed runtime, which drops down the parallel efficiency as shown in Figure 9. In order to improve the parallel efficiency of ParAlg2, we had investigated how to optimize the ordering procedure in Section 4, and we proposed the **MultiLists** ordering procedure,

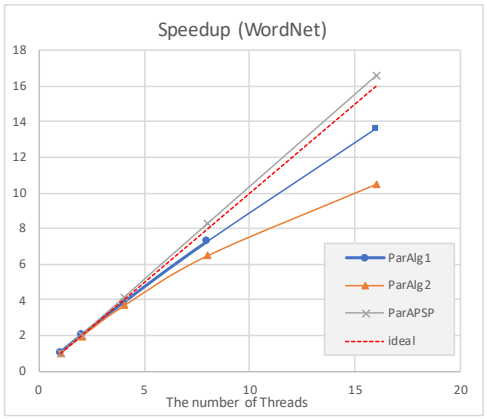


Figure 9: The parallel speedup comparison among ParAlg1, ParAlg2, and the proposed ParAPSP algorithms.

which makes a significant improvement of the overall ordering performance in parallel. Our proposed parallel APSP solution, which includes the MultiLists algorithm, is **ParAPSP**.

The overall elapsed runtimes of ParAlg1, ParAlg2, and our proposed ParAPSP, which are tested with *WordNet* dataset on **Machine-I**, are represented in Figure 8. Figure 8 clearly illustrates the benefit of the optimized ordering procedure. As same as in Section 5.2, ParAlg2 and ParAPSP outperforms ParAlg1 due to the optimization of the descending degree order. When we compare **ParAPSP** with **ParAlg2** in Figure 8, it shows an interesting feature: ParAPSP shows slightly better performance than ParAlg2 in 1-thread experiments, and the performance gap between ParAPSP and ParAlg2 gets bigger as the number of threads increases. Though we discussed the experimental results with *WordNet* dataset as an exemplary case in this section, our experimental results with other datasets also showed the similar performance patterns as in Figure 8.

Figure 9 shows the parallel *speedup* of ParAlg1, ParAlg2, and ParAPSP tested with *WordNet* dataset, which is based on the elapsed runtimes in Figure 8. Though the ParAlg2 algorithm performs better than the ParAlg1 algorithm in terms of the actual runtimes, ParAlg2 shows less speedup than ParAlg1 due to its sequential order procedure. Our proposed ParAPSP algorithm removes the parallel overhead in ParAlg2 and achieves even *hyper-linear* speedup as shown in Figure 9.

5.4 Overall Parallel Performance of ParAPSP

In Section 5.2 and Section 5.3, we discussed parallel performance with respect to the ParAlg1, ParAlg2, and the proposed ParAPSP algorithms, and our experimental analysis shows ParAPSP outperforms ParAlg1 and ParAlg2. In this section, we would like to show the overall parallel performance of our proposed **ParAPSP** algorithm with test datasets in Table 2 on two different shared-memory parallel environments, Machine-I and Machine-II.

Figure 10 consists of (a) the parallel elapsed time of ParAPSP with tested datasets and (b) the corresponding parallel speedup. Note that *sx-superuser* dataset is experimented on **Machine-II** up to 32 threads, since the required memory size for the dataset is at least 160 GB. Other datasets are experimented on **Machine-I** up to 16 threads. As shown in Figure 10, the ParAPSP algorithm shows almost linear speedup or even *hyper-linear* speedup, with tested datasets, interestingly.

We conjecture that the dynamic programming approach of the ParAPSP algorithm would be the reason for the hyper-linear speedup of our proposed algorithm. As shown in Algorithm 1, the modified Dijkstra's algorithm, which was proposed by Peng *et al.* [14], uses the shortest path results that are computed in previous iterations as in dynamic programming methods. The reuse of known shortest path information in the middle of Dijkstra's procedure is the key idea to reduce the unnecessary recalculating shortest paths from intermediate vertices for the APSP problem. The benefit of the modified Dijkstra's algorithm can be expedited by parallelism, since parallel runs of modified Dijkstra's algorithm produce much more available SSSP outputs in the same amount of time, which results in the faster runtime of the modified Dijkstra's algorithm.

Overall, the ParAPSP algorithm achieves exceptional parallel performance as shown in Figure 10 with respect to all of the tested real-world datasets in Table 2 on both of our tested environments.

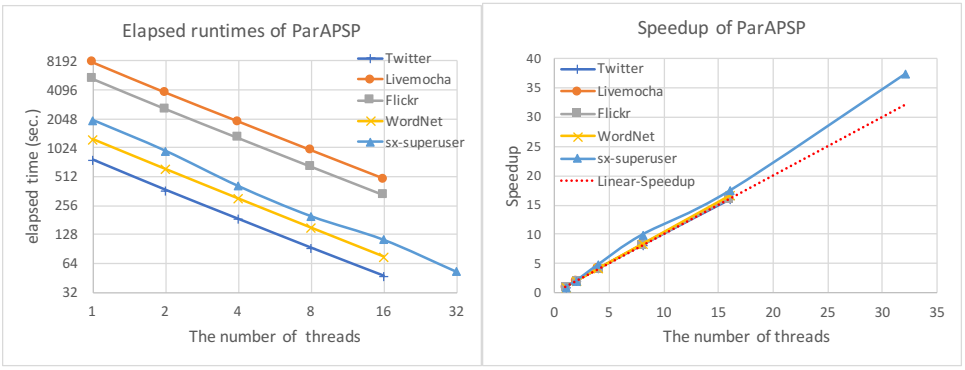
6 RELATED WORK

Since classical shortest path algorithms proposed in late 1950's and early 1960's [4, 8, 10], there have been many attempts to improve the classic shortest path algorithms for both SSSP or APSP problems. Some of the APSP algorithms actually improved the time complexity from $O(n^3)$ to $O(n^3/\log n)$ [5] or to $O(n^{2.4})$ [14]. Though these were significant improvements, it still computationally expensive when we run those algorithms with large real-world datasets. Therefore, many parallel algorithms have been proposed to reduce further the overall runtime.

In 2008, Tang *et al.* [17] proposed a parallel SSSP algorithm is based on Dijkstra's algorithm. Their algorithm consists of two phases: graph partitioning phase and iterative correcting phase. After partitioning an input graph, each sub-graph is assigned to one of the parallel units, and, in the iterative correcting phase, computation step and communication step are processed interchangeably until there is no communication necessary. Their algorithm achieved more than a 15-fold speedup with 16 processors.

Similarly, Abdelghany *et al.* [1] also presented a parallel APSP algorithm, which is based on network decomposition and correcting mechanism. Although it has some gain of a parallel implementation, the overall parallel efficiency is not significant.

Katz and Kider Jr [11] proposed a shared-memory cache-efficient GPU based parallel APSP algorithm based on the Floyd-Warshall algorithm. This algorithm is implemented in CUDA API [6] for utilizing GPU. This approach also partitions the matrix that holds the graph into sub-matrices. It loads some necessary blocks into GPU memory and processes it using the Floyd-Warshall algorithm and returns the result to global memory. Though their proposed algorithm achieves good parallel performance, its time complexity is still $O(n^3)$. In addition, the limited size of memory on GPU could affect its parallel performance when it runs with very large graphs, although their algorithm could handle large graphs, which requires larger memory than GPU's on-board memory as shown in their paper.



(a) Elapsed Time **(b) Parallel Speedup**

Figure 10: Parallel elapsed time and speedup of the proposed ParAPSP algorithm with the tested datasets in Table 2 on Machine-I and Machine-II. *sx-superuser* dataset was run on Machine-II and other datasets were run on Machine-I.

In contrast, our proposed parallel algorithm does not require extra partitioning steps. **ParAPSP** is based on the modified Dijkstra's algorithm, proposed in Peng *et al.*'s paper [14], which uses a simple and efficient idea by saving and loading the previously calculated shortest paths. Its time complexity is shown as $O(n^{2.4})$ empirically [14]. Furthermore, we optimize the ordering scheme, so the proposed parallel algorithm achieves even *hyper-linear* speedup in our experiments.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose an efficient shared-memory parallel all-pairs shortest path (APSP) algorithm based on the state-of-the-art sequential APSP algorithm [14], which is practically in $O(n^{2.4})$ time-complexity and further optimized algorithms were also proposed.

Preliminarily, we parallelized Peng *et al.*'s basic and optimized APSP algorithms, as in ParAlg1 and ParAlg2. Although ParAlg2 showed good parallel performance, it still had an intrinsic sequential overhead of the ordering procedure. Therefore, in this paper, we implement an optimized parallel ordering algorithm based on bucket-sort like approach, called MultiLists. The MultiLists ordering procedure significantly reduces the parallel overhead by several orders of magnitudes, and we propose **ParAPSP** algorithm by applying the MultiLists ordering procedure to the ParAlg2 algorithm.

The experimental analysis shows that our proposed algorithm is very efficient in parallel with various real-world graphs on two shared-memory parallel environments. The ParAPSP algorithm achieves even hyper-linear speedup in our experimental analysis due to the removal of the significant parallel overhead from the ordering procedure and the expedited benefits of the dynamic programming approach in the modified Dijkstra's algorithm by the parallel execution of it.

In future work, we would like to extend the ParAPSP algorithm on distributed-memory parallel environments so that we could find APSP solutions for much larger graphs, which cannot be handled on a commodity single machine.

ACKNOWLEDGMENTS

The research work of Seung-Hee Bae is supported by the startup fund from College of Engineering and Applied Science at Western Michigan University. The research work of Jong Wook Kim and Hyoeun Choi is supported by Handong Global University.

REFERENCES

[1] Khaled Abdelghany, Hossein Hashemi, and Ala Alnawaiseh. 2016. Parallel All-Pairs Shortest Path Algorithm: Network Decomposition Approach. *Transportation Research Record: Journal of the Transportation Research Board* 2567 (2016), 95–104.
[2] Réka Albert and Albert-László Barabási. 2000. Topology of evolving networks: local events and universality. *Physical review letters* 85, 24 (2000), 5234.
[3] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *Science* 286, 5439 (1999), 509–512.
[4] Richard Bellman. 1958. On a routing problem. *Quarterly of applied mathematics* 16, 1 (1958), 87–90.
[5] Timothy M Chan. 2005. All-pairs shortest paths with real weights in O (n3/log n) time. In *Workshop on Algorithms and Data Structures*. Springer, 318–324.
[6] NVIDIA Corporation. 2018. CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/. (2018).
[7] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55. https://doi.org/10.1109/99.660313
[8] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.
[9] Paul Erdös and Alfréd Rényi. 1960. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci* 5 (1960), 17–61.
[10] Robert W. Floyd. 1962. Algorithm 97: Shortest Path. *Commun. ACM* 5, 6 (June 1962), 345–. https://doi.org/10.1145/367766.368168
[11] Gary J Katz and Joseph T Kider Jr. 2008. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. Eurographics Association, 47–55.
[12] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*. ACM, 1343–1350.
[13] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).
[14] Wei Peng, Xiaofeng Hu, Feng Zhao, and Jinshu Su. 2012. A Fast algorithm to find all-pairs shortest paths in complex networks. *Procedia Computer Science* 9 (2012), 557–566.
[15] Seth Pettie. 2004. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science* 312, 1 (2004), 47–74.
[16] Liam Roditty and Uri Zwick. 2004. On dynamic shortest paths problems. In *European Symposium on Algorithms*. Springer, 580–591.
[17] Yuxin Tang, Yunquan Zhang, and Hu Chen. 2008. A parallel shortest path algorithm based on graph-partitioning and iterative correcting. In *High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on*. IEEE, 155–161.
[18] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393, 6684 (1998), 440.