УDC 004
DOI: 10.21122/2309-4923-2024-4-4-12

*PRIHOZHY A.A., KARASIK O.N.*

# COMPETING ALL-PAIRS SHORTEST PATHS ALGORITHMS FOR SPARSE / DENSE GRAPHS: IMPLEMENTATION AND COMPARISON

*Belarussian National Technical University, Minsk, Republic of Belarus*

*In this paper we consider two families of competing algorithms for finding the shortest paths between all pairs of vertices (APSP) in directed weighted large graphs with different edge densities: Dijkstra and Floyd-Warshall. For comparison, we have taken Dijkstra's algorithm with dynamically varying binary heap, which solves the APSP prob-lem purely in parallel by repeatedly executing on all vertices of the graph considered as source vertices, and we have taken blocked Floyd-Warshall algorithm, which is also well-parallelizable. It is known that in terms of computa-tional complexity, the first algorithm is preferable on sparse graphs and the second algorithm is preferable on dense graphs. At the same time, it is not clear what are the ranges of graph densities at which the first algorithm will con-sume less CPU time than the second algorithm. This paper describes multithreaded implementations of parallel al-gorithms on multicore processors that make different usage of synchronization primitives such as mutex, conditional variable, lock-ing, and atomic operation. By conducting computational experiments on an 8-core Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz, we found that each algorithm has a preferred graph density. In the case of multi-threaded parallel imple-mentation, the blocked Floyd-Warshall algorithm has lower running time than Dijkstra's algorithm if the graph densi-ty is greater than 0.5. Otherwise, Dijkstra's algorithm runs faster. In the case of single-threaded implementation, the split point is 0.43.*

***Keywords:*** *Sparse graph, dense graph, APSP problem, Dijkstra-family algorithms, Floyd-Warshall family algorithms, multi-core processor, muti-threaded implementation, comparison*

## Introduction

Models, methods, algorithms, and tools for finding shortest paths between vertices of large, weighted directed and undirected sparse and dense graphs [1] help to solve many problems in many application areas. In this paper, we consider directed simple graphs $G = (V, E)$ where $V$ is a set of vertices and $E$ is a set of edges with positive weights. A graph can have a different number of edges and can range from sparse to dense.

For the single-source shortest paths problem (SSSP), Dijkstra's algorithm [2] with the min-priority queue has the worst-time complexity of $O((|V| + |E|) \cdot \log|V|)$ if the queue is implemented using a binary heap (*BH*). A whole family of algorithms has been developed based on Dijkstra's algorithm [3–5]. Thus, Dijkstra's al-gorithm implemented using a Fibonacci heap (*FH*) has a running time of $O(|V| \cdot \log|V| + |E|)$. The algorithm is most suitable for sparse graphs.

The Floyd-Warshall (*FW*) algorithm [6] for the all-pairs shortest paths problem (APSP) has a time complexity of $O(|V|^3)$ regardless of the number of edges of the graph. The blocked Floyd-Warshall (BFW) algorithm [7–20] is a generalization of the first algorithm with increasing performance. The algorithm is most suitable for dense graphs. APSP can also be solved by performing $N$ runs of Dijkstra's algorithm for vertices $V$ considered as source vertices. For directed simple graphs, the graph density is defined as:

$$Y = |E| / |V| \cdot (|V|-1).$$

Then the Dijkstra running time in the case of the APSP problem depending on the graph density is:

1) for a binary heap

$$O(|V| \cdot (|V| + Y \cdot |V| \cdot (|V|-1)) \cdot \log V);$$

2) for a Fibonacci heap

$$O(|V| \cdot (|V| \cdot \log V + Y \cdot |V| \cdot (|V|-1))).$$

Thus, the speedup of Dijkstra's algorithm compared to the Floyd-Warshall algorithm is:

1) for a binary heap

$$Speedup(BH) = |V| / ((1 + Y \cdot (|V|-1)) \cdot \log |V|);$$

2) for a Fibonacci heap

$$Speedup(FH) = |V| / ((\log |V| + Y \cdot (|V|-1))).$$

Figure 1 shows the dependence of *Speedup(BH)* on the density of graphs consist-ing of 2400 vertices. Dijkstra’s algorithm is faster than the Floyd-Warshall algorithm in the graph density interval [0.0, 0.128118]. The Floyd-Warshall algorithm is faster in interval (0.128118, 1.0]. Dijkstra’s algorithm with the Fibonacci heap is faster than the Floyd-Warshall algorithm in the much wider interval [0.0, 0.99716]. If we move from algorithms to their realization on multicore processors, the division point 0.128118 can be moved in the interval [0, 1].

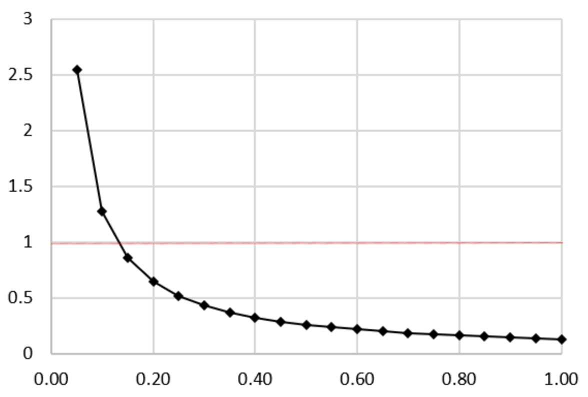

Figure 1. Speedup (in times) of Dijkstra's APSP algorithm compared to the Floyd-Warshall algorithm as a function of graph density Y when graph size $|V| = 2400$

It has been observed in the literature that the binary heap in Dijkstra's algorithm can be implemented more efficiently compared to the Fibonacci heap. Therefore, in the paper, we consider and compare different sequential and parallel implementations of the Floyd-Warshall and Dijkstra (with a dynamically varying binary heap) algorithms on large graphs of different densities.

## Two parallel implementations of Dijkstra's APSP algorithm with dynamic binary heap

Let $N = |V|$ and $W$ be the adjacency matrix for a graph $G$: $w(i, i) = 0$ for $1 \le i \le N$; $w(i, j)$ is the weight of edge $(i, j) \in E$; $w(i, j) = \infty$ for $(i, j) \infty E$ and $i \neq j$. Let $D$ be the distance matrix between all pairs of vertices $i, j \in V$, $i \neq j$ and $d_{ij}$ be the length of the shortest path from vertex $i$ to vertex $j$. Let $P$ be a matrix whose element $p_{ij}$ is the vertex preceding vertex $j$ in the path to be shortest from $i$ to $j$. The task of the APSP algorithm is to compute all elements of matrices $D$ and $P$ given by the graph $G$.

Dijkstra's SSSP algorithm can be easily extended to the APSP algorithm by repeatedly applying it to rows $i$ of matrices $D$ and $P$. All rows can be computed in parallel. Figure 2 shows the architecture of our version Dv.1 of the Dijkstra APSP parallel algorithm implementation. The set of rows of matrix $D$ and the set of rows of matrix $P$ are partitioned into corre-sponding slots 1 … $T$ of rows, which are computed by separate threads. Each thread uses its own *Dist* and *Prev* vectors and its own dynamic , therefore it can operate completely independently of other threads. There is no need to use synchronization facilities. If the running time of Dijkstra's algorithm for one source vertex is close to the running time for another source vertex, the computational load is almost the same for all threads.

Algorithm 1 describes the behavior of the multithreaded parallel Dijkstra APSP algorithm. It creates a thread that executes a function Dijkstra_APSP to compute the shortest paths from each source vertex of the corresponding slot. Algorithm 2 implements this function. Its inputs are the thread number $t$, the number $N$ of vertices, and the adjacency list $AL$, which is the set of graph edges (and their weights) outgoing from each vertex. Its outputs are matrices $D$ and $P$, whose row slots are updated by thread $t$. A *Slot* defines the range from the *first* to the *last* row of matrices $D$ and $P$ that are assigned to a thread. The function *Dijkstra_SSSP* computes the vectors *Dist* and *Prev*, which are assigned to the corresponding rows of $D$ and $P$.

Algorithm 3 describes the Dijkstra SSSP algorithm, which uses the $AL$ graph adjacency list and works with a min-priority queue *QueueB*, arrays *Dist* and *Prev*. We represent the queue as a labeled dynamically changing binary tree heap. Initially, the tree consists of $2N - 1$ nodes ($N$ terminal and $N - 1$ nonterminal) and has $\lceil \log n \rceil$ depth.

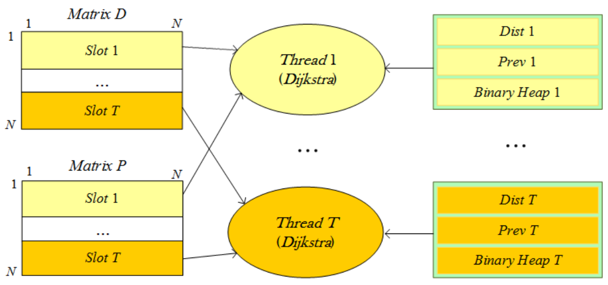

Figure 2. Version Dv.1 of parallel multithreaded implementation of Dijkstra APSP algorithm with dynamic binary heap

**Algorithm 1:** Multithreaded parallel Dijkstra APSP algorithm (version Dv.1)

```
Input:  A number $N$ of graph vertices
Input:  A number $T$ of threads
Input:  A graph adjacent list $AL$
Output: A matrix $D[N\times N]$ of shortest path distances
Output: A matrix $P[N\times N]$ of previous vertices in
        shortest paths
   for $t \leftarrow 1$ to $T$ do
       Create_Thread(t) with function
             Dijkstra_APSP(t, N, AL, D, P)
   for $t \leftarrow 1$ to $T$ do
       Join_Thread(t)
   return B, P
```

**Algorithm 2:** *Dijkstra_APSP*

```
Input:  A number $t$ of thread
Input:  A number $N$ of graph vertices
Input:  An adjacent list $AL$ of graph
Output: A matrix $D$ of shortest path distances
Output: A matrix $P$of previous vertices in shortest
        paths
   QueueB ← Create_Binary_Queue(N, AL)
   Dist ← Create_Initialize_Dist(N)
```

```
    Prev ← Create_Initialize_Prev(N)
    Slot ← Choose_Slot(N, t)
    for row ← Slot.first to Slot.last do
        Dijkstra_SSSP(N, AL, row, QueueB, Dist, Prev)
        D(row) ← Dist
        P(row) ← Prev
    return B, P
```

Each node has two labels: vertex identifier *v* and key *Dist*[*v*]. The root of the tree points to the vertex with the smallest key. *QueueB* supports two operations:

1. *Extract-min* removes the *NearestV* element with the smallest key from *QueueB*; this removes one terminal and one non-terminal vertex from the tree.

2. *Decrease-key* replaces the current key *Dist*[*AdjV.id*] of adjacent vertex *AdjV.id* with the *NewDist* key, and then reorders *QueueB* by changing the labels of the vertices in the tree.

**Algorithm 3:** *Dijkstra_SSSP*

```
Input:  A number N of graph vertices
Input:  An adjacent list AL of graph
Input:  A row of matrix D and matrix P
Output:  A vector Dist of shortest path distances
Output:  A vector Prev of previous vertices in paths
    for i ← 1 to N do
        Dist[v] ← ∞
        Prev[v] ← undefined
    Dist[row] ← 0
    Prev[row] ← row
    QueueB ← Initialize_Queue(N, row)
    for Step ← 1 to N do
        NearestV ← QueueB.root.id
        Distance ← Dist[NearestV]
        if Distance = ∞ ← then  break
        for AdjV ← AL[NearestV].list do
            NewDist ← Distance + AdjV.key
            if  Dist[AdjV.id] > NewDist then
                Dist[AdjV.id] ← NewDist
                Prev[AdjV.id] ← NearestV
                Decrease_Key (QueueB, AdjV.id, NewDist)
                Extract_Min (QueueB, NearestV)
        return Dist, Prev
```

As the binary tree becomes smaller step by step, the average time complexity of *Extract-min* and *Decrease-key* operations is less than log |*V*|. This is the source of speeding up of Algorithm 3.

Figure 3 shows the Dv.2 version of our implementation of the parallel Dijkstra APSP algorithm. Each thread captures matching rows of matrices *D* and *P* concurrently and calls *Dijkstra _SSSP*. After computing the shortest paths for the source, the thread captures the next rows. Since the pool is shared by all threads, our implementation uses atomic operations to select rows from the pool.

## Parallel BFW and its implementation with threads, block pools and atomic operations

*BFW* performs graph partitioning into sub-graphs with *S* vertices and creates blocked matrices *B*[*M*×*M*] of distances of shortest paths and *P*[*M*×*M*] of previous vertices on shortest paths, where $M = N/S$. In the outer loop along *m*, three types of blocks are computed sequentially: diagonal, cross and peripheral. The cross blocks are collected in a *PoolC* of size $2{\cdot}(M - 1)$ and are computed mutually in parallel by the *Perform_Parallel_Pool* function. Peripheral blocks are collected in *PoolP* of size $(M-1)^2$ and are computed in parallel by the same function.

The architecture of the multithreaded implementation of the Dv.1 version of the algorithm is shown in Figure 3. The main thread computes the diagonal blocks using the *FW* or *GEA* algorithm [15]. Threads 1 to *T* compute cross and peripheral blocks from *PoolC* and *PoolP*. The architecture has a drawback as the *Perform_Parallel_Pool* function (Algorithm 5) creates and deletes all pools and threads 2·M times. Its advantage is no need for synchronization between Algorithms 4 and 5.

**Algorithm 4:** Parallel BFW with pools and atomic operations

```
Input:  A number N of input graph vertices
Input:  A matrix W[N×N] of graph edge weights
Input:  A number M of blocks
Input:  A size S of block
Output:  A blocked matrix B[M×M] of path distances
Output:  A blocked matrix P[M×M] of previous vertices
    B[M×M] ← W[N×N]
    Initialize_Prev (P[M×M])
    for m ← 1 to M do
        Calculate_Block (S, B, P, m, m, m)
        Initialize_Pool_C (Pool_C_of_Blocks)
        for v ∈ {1…M} and v ≠ m do
            Add_to_Pool_C (v, m, m)
            Add_to_Pool_C(m, m, v)
        Perform_Parallel_Pool (PoolC_of_Blocks, S, B, P)
        Initialize_Pool_P (Pool_P_of_Blocks)
        for v ∈ {1…M} and v ≠ m do
            for u ∈ {1…M} and u ≠ m do
                Add_to_Pool_P(v, m, u )
        Perform_Parallel_Pool (PooIP_of_Blocks, S, B, P)
    return B, P
```

**Algorithm 5:** *Perform_Parallel_Pool*

```
Input: A Pool_of_Blocks to be computed
Input: A size S of block
Inout: A blocked matrix B of path distances
Inout: A blocked matrix P of previous vertices
    for t ← 1 to T do
        Create_Thread(t) with function
            Compute_Blocks (Pool_of_Blocks, t, S, B, P)
    for t ← 1 to T do
        Join_Thread(t)
```

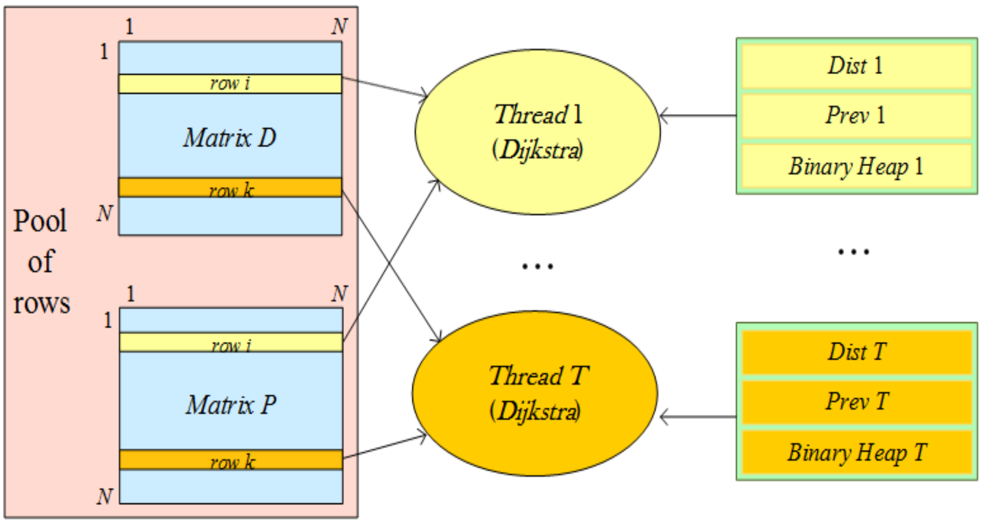

Figure 3. Version Dv.2 of parallel multi-threaded implementation of the Dijkstra APSP algorithm with a pool of rows

The *Compute_Blocks* function (Algorithm 6) is run by threads 1 to *T*. Each function call iteratively grabs a unique record from the same pool and uses it to recalculate one block. All records that are in the pool can be processed in parallel. The capture of records is accomplished using atomic operations. Different threads can process different number of records.

**Algorithm 6:** *Compute_Blocks* implemented by each thread

```
Input:  A Pool_of_Blocks to be computed
Input:  A thread number $t$
Input:  A size $S$ of block
InOut:  A matrix $B$ updated for blocks of pool
InOut:  A matrix $P$ updated for blocks of pool
    while (true) do
        rec ← Atomic_Next_record_Capture (Pool_of_Blocks)
        if $rec \neq empty$ then
            Calculate_Block (S, rec.v, rec.m, rec.u, B, P)
        else   break
```

The *Compute_Blocks* function (Algorithm 7) has six inputs: the block size *S*, the three indices *v, m* and *u* of the vertex subsets, the matrices *B* and *P*. It recalculates the block $B_{v,u}$ through the blocks $B_{v,m}$ and $B_{m,u}$, of which two or three may be the same.

**Algorithm 7:** Block calculation *(Calculate_Block)*

```
Input:  A size $S$ of block
Input:  Indices $v$, $m$ and $u$ of vertex subsets
InpOut:  A blocked matrix $B$ of path distances
InpOut:  A blocked matrix $P$ of previous vertices
    for $k \leftarrow 1$ to $S$ do
        for $i \leftarrow 1$ to $S$ do
            for $j \leftarrow 1$ to $S$ do
                $sum \leftarrow B_{v,m}(i, k) + B_{m,u}(k, j)$
                if $B_{v,u}(i, j) > sum$ then
                    $B_{v,u}(i, j) \leftarrow sum$
                    $P_{v,u}(i, j) \leftarrow P_{m,u}(k, j)$
```

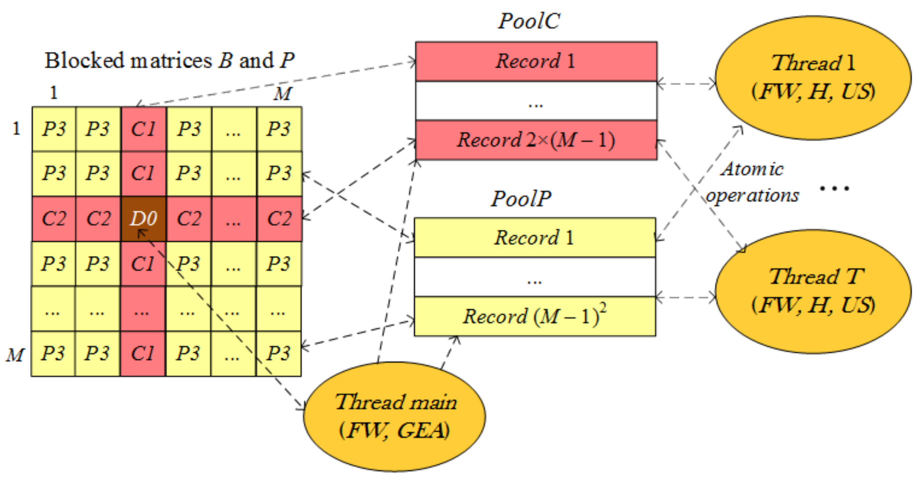

Figure 4. FWv.1 version of parallel multi-threaded implementation of BFW algorithm with pools *PoolC* and *PoolP*; *FW* stands for Floyd-Warshall, *H* stands heterogeneous, *US* – unequal sizes and GEA stands for graph extension algorithm

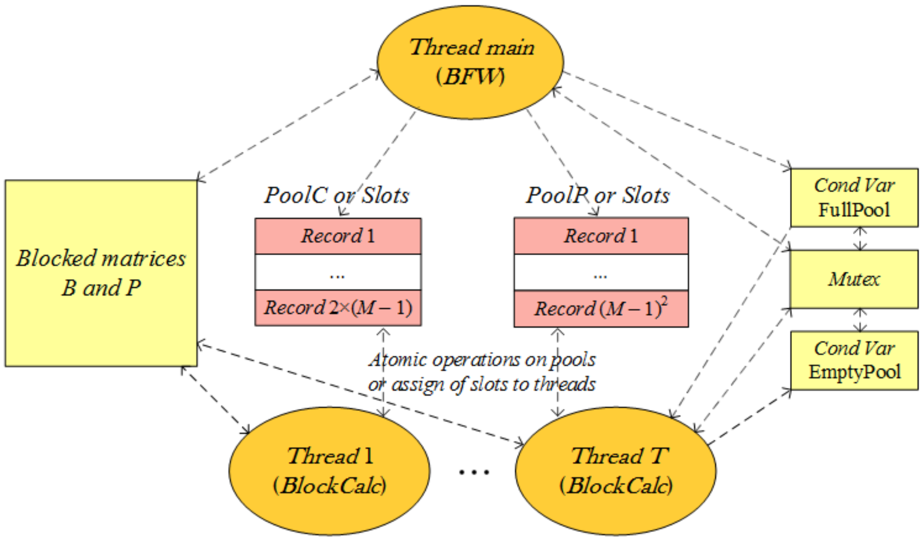

Figure 5. Version FWv.2 and FWv.3 of parallel multi-threaded implementation of *BFW* algorithm with pools, mutexes and conditional variables

## Two implementations of parallel BFW using threads, mutexes and conditional variables

To avoid multiple creation and deletion of worker threads in the loop across *m* (Algorithm 4), we have developed FWv.2 and FWv.3 ver-sions of parallel implementations of *BFW* (Figure 5). They use a mutex and conditional variables *FullPool* and *EmptyPool* to synchronize the main thread executing Algorithm 4 and worker threads 1 to T executing Algorithm 7 to safely recalculate all blocks described in the pool. The mutex protects the pool from destruction. The versions differ from each other in the way the threads access entries in the pool. The FWv.2 version pre-divides the pool record set into slots, one slot for one thread. No conflict oc-curs when two and more threads capture the same record. In the FWv.3 version, each thread has access to all records in the pool and uses atomic operations to dynamically capture the next record for processing. The FWv.2 version is preferred when each block is computed at approximately the same time and the running time of each thread is predictable in advance. The FWv.1 and FWv.3 versions may be faster if blocks of different types are computed by different heterogeneous algorithms requiring different CPU times, or if blocks are of unequal size and require different computation times.

## Results

Experimental results are obtained on an Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz 8-core 16 physical thread processor using C++ language and the Visual Studio 2019 Community Edition compiler (MSVC++ 14.29). Table 1 shows that version Dv.1 of Dijkstra's multi-threaded implementation of APSP is on average 3.2 % faster than version Dv.2, although it loses to Dv.2 on graphs of certain sizes with density 0.2. For graphs of density 0.8, Dv.2 is on average 2.85 % faster than Dv.1.

Table 1. Comparison of running times (%) on different graph sizes of two versions of multi-threaded implementations of Dijkstra's APSP algorithm for graph densities of 0.2 and 0.8

| Graph size | Density 0.2 | Density 0.8 |
|---|---|---|
| 1200 | -2.86 | 1.85 |
| 2400 | 1.25 | -0.61 |
| 3600 | -0.24 | 3.73 |
| 4800 | -0.41 | 5.54 |
| 6000 | 7.84 | 4.63 |
| 7200 | 10.27 | 2.78 |
| 8400 | 6.58 | 2.05 |

Table 2 compares the runtimes of three multi-threaded implementations of the blocked Floyd-Warshall algorithm on graphs of sizes 1200–8400. For graphs of density 0.8, the FWv.1 version wins on average 4.52 % over FWv.2 and 4.64 % over FWv.3. The FWv.2 version wins 0.26 % over FWv.3.

Table 2. Comparison (%) of running times of three versions of BFW implementions for graph density 0.8

| Graph size | v.1 / v.2 | v.1 / v.3 | v.2 / v.3 |
|---|---|---|---|
| 1200 | -1.94 | -1.94 | 0.00 |
| 2400 | 1.90 | 0.32 | 1.59 |
| 3600 | 8.81 | 15.01 | -7.30 |
| 4800 | 14.36 | 10.94 | 3.85 |
| 6000 | -0.53 | -1.12 | 0.58 |

Figure 6 compares single-thread implementations of sequential *FW, BFW* and Dijkstra APSP on graphs of different densities. As can be seen, *BFW* is about 1.83 times faster than *FW* on all graphs. Dijkstra is 6.07 times faster than *FW* at *density* = 0.1 and is 1.1 times slower at *density* = 1. Dijkstra is 3.67 times faster than *BFW* at *density* = 0.1 and twice as slow at *density* = 1. If *density* $\leq$ 0.43, Dijkstra beats *BFW*, otherwise it loses.

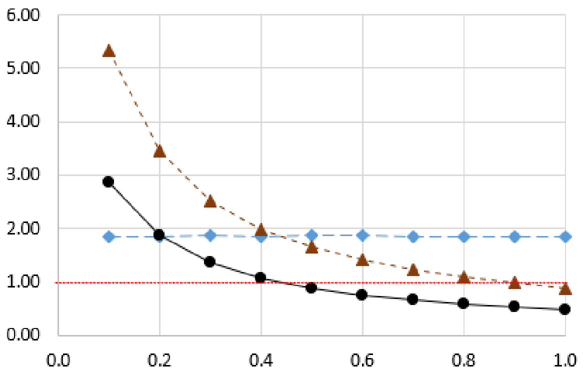

Figure 6. Comparison of three single-threaded APSP algorithms on graphs with 2400 vertices and block-size 120 as a function of graph density: long dashed line – reduction of runtime (in times) of *BFW* compared to *FW*, dashed line – Dijkstra compared to *FW*, and solid line – Dijkstra compared to *BFW*

Figure 7 shows a comparison of two parallel multi-threaded implementations of the APSP algorithms with each other and with their single-thread implementations at graph densities from 0.1 to 1. It is shown that parallel APSP Dijkstra is up to 3.98 times faster than parallel *BFW* when the graph *density* $\leq$ 0.5. Parallel *BFW* is twice as fast as parallel Dijkstra if the *density* > 0.5.

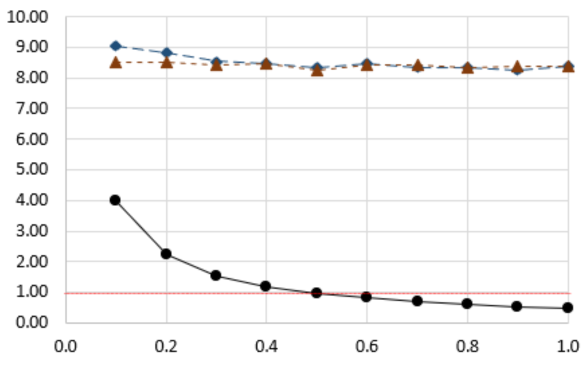

Figure 7. Comparison of two parallel 16-threaded implementations of APSP algorithms with each other and with their sequential counterparts on graphs of 2400 vertices and block-size 120 as a function of graph density: solid line – runtime reduction (in times) of parallel Dijkstra compared to parallel *BFW*, long dashed line – of parallel Dijkstra compared to sequential Dijkstra, and dashed line – of parallel *BFW* compared to sequential *BFW*

Parallel Dijkstra is faster than sequential Dijkstra by an average 8.5 times. Parallel *BFW* is 8.41 times faster than sequential BFW. The division point of the interval [0, 1] has moved to the centre. We attribute this with three main reasons:

1) the average case rather than the worst case is evaluated; 2) the effect of a multi-core processor with hierarchical memory; and 3) the properties of a dynamically modified heap. Table 3 shows that the number the decrease-key calls is bounded and the number of levels of binary heap the key moves over remains between 1.59 and 1.71 when the graph density is in [0.1, 1.0].

Table 3. Parameters of Decrease-Key in dynamic binary heap for graphs of 2400 vertices with different edge densities

| Edges % | Edge count | DK calls | Edges per one call | Levels per call |
|---|---|---|---|---|
| 0.1 | 578122 | 10243 | 56 | 1.71 |
| 0.2 | 1152695 | 10828 | 106 | 1.68 |
| 0.3 | 1728197 | 11071 | 156 | 1.67 |
| 0.4 | 2305101 | 11206 | 205 | 1.66 |
| 0.5 | 2879364 | 11245 | 256 | 1.65 |
| 0.6 | 3455521 | 11366 | 304 | 1.64 |
| 0.7 | 4031390 | 11491 | 350 | 1.62 |
| 0.8 | 4606640 | 11561 | 398 | 1.61 |
| 0.9 | 5182374 | 11561 | 448 | 1.60 |
| 1.0 | 5757600 | 11511 | 500 | 1.59 |

Figure 8 shows that the parallel Dijkstra algorithm is faster than the parallel BFW algorithm by a factor of 2.23 to 2.67 on sparse graphs of 1200–8400 vertices with density 0.2.

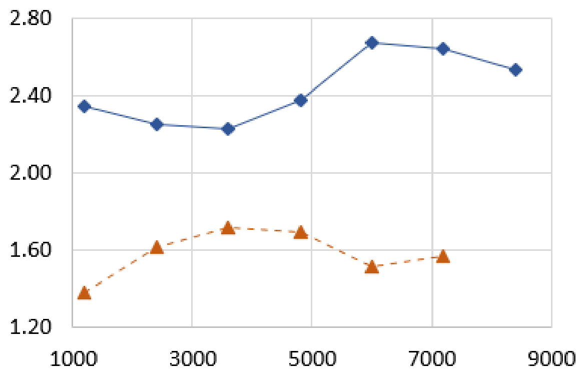

Figure 8. Comparison of parallel APSP Dijkstra and *BFW* on graph of different sizes: solid line – reduction of runtime (in times) of Dijkstra compared to *BFW* at *density* = 0.2; dashed line – reduction of runtime of *BFW* compared to Dijkstra at *density* = 0.8

At the same time, Figure 8 shows that the parallel *BFW* algorithm is faster than Dijkstra's parallel algorithm by a factor of 1.39 to 1.72 on dense graphs of 1200–7200 vertices with density 0.8. Dijkstra's gain over *BFW* decreases as the graph size increases from 1200 to 3600. Then the gain increases up to a graph size of 6000. For larger graphs, the gain decreases.

## Conclusion

The paper has shown that the computational complexity of all-pirs shortest paths algorithms can be evaluated theoretically approximated. When the algorithms are implemented with par-allel multithreaded applications for multi-core processors, the regions of preference of the al-gorithms differs from those theoretically pre-dicted. We have developed two parallel multi-thread implementations of parallel Dijkstra APSP algorithm with dynamic binary heap and three parallel multi-thread implementations of blocked Floyd-Warshall algorithm and experi-mentally have shown on Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz 8-core processor that Dijkstra's APSP is faster than the blocked Floyd-Warshall on sparse graphs with *density* $\leq 0.5$. On graphs with *density* $> 0.5$, the blocked Floyd-Warshall is faster than Dijkstra's APSP.

## REFERENCES

1. **Madkour A., Aref W.G., Rehman F.U., Rahman M.A., Basalamah S.** A Survey of Shortest-Path Algorithms. ArXiv: 1705.02044v1 [cs.DS], 4 May 2017, 26 p.
2. **Dijkstra E.W.** A note on two problems in connexion with graphs. Numerische Mathematik, 1959, vol. 1, no. 1, pp. 269-271.
3. **Bellman R.E.** On a routing problem. Quarterly of Applied Mathematics, 1958, vol. 16, no. 1, pp. 87-90.
4. **Johnson D.B.** Efficient Algorithms for Shortest Paths in Sparse Networks. J. ACM, 1977, vol. 24 no. 1, pp. 1-13.
5. **Harish P., Narayanan P.J.** Accelerating large graph algorithms on the GPU using CUDA. International conference on high-performance computing. Springer, 2007, pp. 197-208.
6. **Floyd R.W.** Algorithm 97: Shortest path. Communications of the ACM, 1962, no. 5 (6), p. 345.
7. **Katz G.J., Kider J.T.** All-pairs shortest paths for large graphs on the GPU. GH'08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware. ACM, 2008, pp. 47-55.
8. **Djidjev H., Thulasidasan S., Chapuis G., Andonov R. and Lavenier D.** Efficient multi-GPU computation of all-pairs shortest paths. IEEE 28th International Parallel and Distributed Processing Symposium. IEEE, 2014, pp. 360-369.
9. **Venkataraman G., Sahni S., Mukhopadhyaya S.** A Blocked All-Pairs Shortest Paths Algorithm. Journal of Experimental Algorithmics (JEA), 2003, vol. 8, pp. 857-874.
10. **Park J.S., Penner M., and Prasanna V.K.** Optimizing graph algorithms for improved cache performance. IEEE Trans. on Parallel and Distributed Systems, 2004, no. 15 (9), pp. 769-782.
11. **Yang S., Liu X., Wang Y., He X., Tan G.** Fast All-Pairs Shortest Paths Algorithm in Large Sparse Graph. ICS '23: Proceedings of the 37th International Conference on Supercomputing, 2023, pp. 277-288.
12. **Prihozhy A.A., Karasik O.N.** Advanced heterogeneous block-parallel all-pairs shortest path algorithm. Proceedings of BSTU, issue 3, Physics and Mathematics. Informatics, 2023, no. 1 (266), pp. 77-83.
13. **Prihozhy A.A., Karasik O.N.** New blocked all-pairs shortest paths algorithms operating on blocks of unequal sizes. System analysis and applied information science, 2023, no. 4, pp. 4-13.
14. **Karasik O.N., Prihozhy A.A.** Blocked algorithm of finding all-pairs shortest paths in graphs divided into weakly connected clusters. System analysis and applied information science, 2024, no. 2, pp. 4-10.
15. **Prihozhy A.A., Karasik O.N.** Inference of shortest path algorithms with spatial and temporal locality for big data processing. Big Data and Advanced Analytics: proceedings of VIII international conference. Minsk, Bestprint Publ., 2022, pp. 56-66.
16. **Karasik O.N., Prihozhy A.A.** Tuning block-parallel all-pairs shortest path algorithm for efficient multi-core implementation. System analysis and applied information science, 2022, no. 3, pp. 57-65.
17. **Prihozhy A.A.** Generation of shortest path search dataflow networks of actors for parallel multicore implementation. Informatics, 2023, vol. 20, no. 2, pp. 65-84.
18. **Prihozhy A.A.** Optimization of data allocation in hierarchical memory for blocked shortest paths algorithms. System analysis and applied information science, 2021, no. 3, pp. 40-50.
19. **Prihozhy A.A.** Simulation of direct mapped, k-way and fully associative cache on all-pairs shortest paths algorithms. System analysis and applied information science, 2019, no. 4, pp. 10-18.
20. **Prihozhy A.A., Karasik O.N.** Influence of shortest path algorithms on energy consumption of multi-core processors. System analysis and applied information science, 2023, no. 2, pp. 4-12.

## ЛИТЕРАТУРА

1. **Madkour A., Aref W.G., Rehman F.U., Rahman M.A., Basalamah S.** A Survey of Shortest-Path Algorithms. ArXiv: 1705.02044v1 [cs.DS], 4 May 2017, 26 p.
2. **Dijkstra E.W.** A note on two problems in connexion with graphs. Numerische Mathematik, 1959, vol. 1, no. 1, pp. 269-271.
3. **Bellman R.E.** On a routing problem. Quarterly of Applied Mathematics, 1958, vol. 16, no. 1, pp. 87-90.
4. **Johnson D.B.** Efficient Algorithms for Shortest Paths in Sparse Networks. J. ACM, 1977, vol. 24, no. 1, pp. 1-13.

5. **Harish P., Narayanan P.J.** Accelerating large graph algorithms on the GPU using CUDA. International conference on high-performance computing. Springer, 2007, pp. 197-208.

6. **Floyd R.W.** Algorithm 97: Shortest path. Communications of the ACM, 1962, no. 5 (6), p. 345.

7. **Katz G.J., Kider J.T.** All-pairs shortest paths for large graphs on the GPU. GH'08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware. ACM, 2008, pp. 47-55.

8. **Djidjev H., Thulasidasan S., Chapuis G., Andonov R. and Lavenier D.** Efficient multi-GPU computation of all-pairs shortest paths. IEEE 28th International Parallel and Distributed Processing Symposium. IEEE, 2014, pp. 360-369.

9. **Venkataraman G., Sahni S., Mukhopadhyaya S.** A Blocked All-Pairs Shortest Paths Algorithm. Journal of Experimental Algorithmics (JEA), 2003, vol. 8, pp. 857-874.

10. **Park J.S., Penner M., and Prasanna V.K.** Optimizing graph algorithms for improved cache performance. IEEE Trans. on Parallel and Distributed Systems, 2004, no. 15 (9), pp. 769-782.

11. **Yang S., Liu X., Wang Y., He X., Tan G.** Fast All-Pairs Shortest Paths Algorithm in Large Sparse Graph. ICS '23: Proceedings of the 37th International Conference on Supercomputing, 2023, pp. 277–288.

12. **Прихожий А.А., Карасик О.Н.** Усовершенствованный разнородный блочно-параллельный алгоритм поиска кратчайших путей на графе. Труды БГТУ. Сер. 3, Физико-математические науки и информатика, 2023, № 1 (266), с. 77-83.

13. **Прихожий А.А., Карасик О.Н.** Новые блочные алгоритмы поиска кратчайших путей между всеми парами вершин графа, работающие на блоках неравных размеров. Системный анализ и прикладная информатика. 2023, № 4, с. 4-13.

14. **Карасик О.Н., Прихожий А.А.** Блочный алгоритм поиска кратчайших путей между всеми парами вершин в графах со слабосвязанными кластерами. Системный анализ и прикладная информатика. 2024, № 2, с. 4-10.

15. **Prihozhy A.A., Karasik O.N.** Inference of shortest path algorithms with spatial and temporal locality for big data processing. Big Data and Advanced Analytics: сборник научный статей VIII Международной научно-практической конференции, Минск, 11-12 мая 2022 года. Минск, БГУИР, 2022, с. 56-66.

16. **Карасик О.Н., Прихожий А.А.** Настройка блочно-параллельного алгоритма поиска кратких путей на эф-фективную многоядерную реализацию. Системный анализ и прикладная информатика. 2022, № 3, с. 57-65.

17. **Прихожий А.А.** Генерация потоковых сетей акторов поиска кратчайших путей для параллельной многоядерной реализации. Информатика, 2023, № 2, с. 65-84.

18. **Прихожий А.А.** Оптимизация размещения данных в иерархической памяти для блочных алгоритмов поиска кратчайших путей. Системный анализ и прикладная информатика, 2021, no. 3, с. 40-50.

19. **Прихожий А.А.** Моделирование кэш прямого отображения и ассоциативных кэш на алгоритмах поиска кратчайших путей на графе. Системный анализ и прикладная информатика, 2019, no. 4, с. 10-18.

20. **Прихожий А.А., Карасик О.Н.** Влияние алгоритмов поиска кратчайших путей на энергопотребление многоядерных процессоров. Системный анализ и прикладная информатика, 2023, № 2, с. 4-12.

*ПРИХОЖИЙ А.А., КАРАСИК О.Н.*

# КОНКУРИРУЮЩИЕ АЛГОРИТМЫ ПОИСКА КРАТЧАЙШИХ ПУТЕЙ МЕЖДУ ВСЕМИ ПАРАМИ ВЕРШИН РАЗРЕЖЕННЫХ / ПЛОТНЫХ ГРАФОВ: РЕАЛИЗАЦИЯ И СРАВНЕНИЕ

*Белорусский национальный технический университет, Минск, Республика Беларусь*

*В статье рассматриваются два семейства конкурирующих алгоритмов поиска кратчайших путей между всеми парами вершин (APSP) в ориентированных взвешенных больших графах с различной плотностью ребер: Дейкстры и Флойда-Уоршелла. Для сравнения мы взяли алгоритм Дейкстры с динамически изменяемой двоичной кучей, который решает задачу APSP чисто параллельно путем многократного выполнения на всех вершинах графа, рассматриваемых в качестве исходных, и взяли блочный алгоритм Флойда-Уоршелла, который также является хорошо распараллеливаемым. Известно, что с точки зрения вычислительной сложности первый алгоритм предпочтительнее на разреженных графах, а второй – на плотных. В то же время неясно, каковы диапазоны плотностей графов, при которых первый алгоритм будет потреблять процессорное время, меньшее, чем второй алгоритм. В статье описаны реализации многопоточных параллельных алгоритмов на многоядерных процессорах, которые по-разному используют такие примитивы синхронизации, как мьютекс, условная переменная, блокировка и атомарная операция. Проведя вычислительные эксперименты на 8-ядерном процессоре Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz, мы обнаружили, что каждый алгоритм имеет предпочтительную плотность графов. В случае многопоточной параллельной реализации блочный алгоритм Флойда-Уоршелла имеет меньшее время работы, чем алгоритм Дейкстры, если плотность графа больше 0,5. В противном случае алгоритм Дейкстры работает быстрее. В случае однопоточной реализации точка разделения – 0,43.*

***Ключевые слова:*** *Разреженный граф, плотный граф, задача APSP, алгоритмы семейства Дейкстры, алгоритмы семейства Флойда-Уоршелла, многоядерный процессор, многопоточная реализация, сравнение*

**Anatoly Prihozhy** is full professor at Computer and system software department of Belarussian national technical university, Doctor of Science (1999) and Full Professor (2001). His research interests include programming and hardware description languages, parallelizing compilers, and computer aided design techniques and tools for software and hardware at logic, high and system levels, and for incompletely specified logical systems. He has over 300 publications in Eastern and Western Europe, USA and Canada. Such worldwide publishers as IEEE, Springer, Kluwer Academic Publishers, World Scientific and others have published his works.

**Прихожий А.А.,** профессор кафедры «Программное обеспечение информационных систем и технологий» Белорусского национального технического университета, д.т.н. (1999), диплом профессора (2001). В сферу его научных интересов входят языки программирования и описания аппаратуры, распараллеливающие компиляторы, методы и средства автоматизированного проектирования программных и аппаратных средств на логическом, высоком и системном уровнях, а также не полностью определенных логических систем. Имеет более 300 публикаций в Восточной и Западной Европе, США и Канаде. Его работы опубликованы в таких мировых издательствах, как IEEE, Springer, Kluwer Academic Publishers, World Scientific и других.

**E-mail:** prihozhy@bntu.by

**Karasik Oleg** is a Technology Lead at ISsoft Solutions (part of Coherent Solutions) in Minsk, Belarus, and PhD in Technical Science. His research interests include parallel multithreaded applications and the parallelization for multicore and multiprocessor systems.

**Карасик О.Н.,** ведущий инженер иностранного производственного унитарного предприятия «ИССОФТ СОЛЮШЕНЗ» (часть Coherent Solutions), г. Минск, Беларусь, к.т.н. (2019). В сферу его научных интересов входят параллельные многопоточные приложения и распараллеливание для многоядерных и многопроцессорных систем.

**E-mail:** karasik.oleg.nikolaevich@gmail.com