UDC [611.018.51+615.47]:612.086.2

ALL PAIRS SHORTEST PATHS SEARCH IN LARGE GRAPHS

A.A. Prihozhy
Professor at the Computer and System Software Department, Doctor of Technical Sciences, Full Professor

O.N. Karasik
Tech Lead at ISsoft Solutions (part of Coherent Solutions) in Minsk, Belarus, PhD in Technical Science

Belarusian National Technical University, Belarus
ISsoft Solutions (part of Coherent Solutions), Belarus
E-mail: prihozhy@yahoo.com, karasik.oleg.nikolaevich@gmail.com

A.A. Prihozhy
Full professor at the Computer and system software department of Belarusian 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.

O.N. Karasik
Tech Lead at ISsoft Solutions (part of Coherent Solutions) in Minsk, Belarus; PhD in Technical Science (2019). Interested in parallel computing on multi-core and multi-processor systems.

Abstract. The all pairs shortest paths search in a graph has many different application domains. This paper analyzes the known algorithms for solving the problem and considers its parallelization and scaling to the graph size and multiprocessor architecture. It considers a class of shortest paths block-parallel algorithms and studies their advantages and disadvantages. Modeling and simulation have shown that the block algorithms give a manifold reduction in the exchange of data between the hierarchical memory levels for certain ratios of the graph size to the cache memory size. However, if the graph size is too large its characteristics are close to the Floyd-Warshall algorithm characteristics. To improve the performance, the homogeneous block algorithm has been transformed into a heterogeneous one, which reduces the block computation time. Further improvement has been achieved through the development of a cooperative multi-thread scheduler and a block-parallel algorithm that target large graphs and change the order of block calculation, localize data processing, reduce the critical path, decrease the operation time on a multi-core system, and improve the hierarchical memory operation. Studies have shown that the parallel cooperative multi-thread algorithm works well together with the heterogeneous algorithm, since the problem solving time reduces. The carried out analysis and the conducted computational experiments have made it possible to outline the directions for further development of models and algorithms for scaling the all pairs shortest paths problem.

Keywords: shortest path, Floyd-Warshall algorithm, scaling, block algorithm, heterogeneous algorithm, multithreaded algorithm, cooperative execution, multiprocessor system, hierarchical memory.

Problem formulation.

The problem of finding the shortest paths in a weighted graph [1 – 5] is formulated in different settings: for an directed or undirected, sparse or dense graph, with weighted edges and/or weighted vertices, positive or possibly negative weights, between a pair of vertices (SSSP – Single Source Shortest Path) or all pairs of vertices (APSP – All Pairs Shortest Path), with the obligatory passage of all vertices (the traveling salesman problem) or an optional passage, etc. The computational complexity of different problem statements is different: the search for the shortest path between a pair of vertices is solved in

polynomial second degree time by Dijkstra's algorithm; the search for the shortest paths between all pairs of vertices is solved in polynomial time of the third degree by the Floyd–Warshall algorithm; finding the shortest path with the mandatory passage of all vertices of the graph is an NP-hard problem in combinatorial optimization.

The APSP task plays an important role in many applications [6 – 8]: reducing city traffic, optimizing network infrastructure in data centers, planning tasks, implementing augmented reality, network analysis, microelectronics, programming, computer networks, computer games etc. At the same time, even the Floyd–Warshall algorithm has a search time proportional to the value of 1.25×10^{11} for a graph of 5000 vertices, which is practically unacceptable even on modern computer architectures. In many application domains, real graphs reach much larger sizes, for which the Floyd - Warshall algorithm consumes unrealistically large amount of time.

Works [3 – 5] propose APSP algorithms, which have lower computational complexity over Floyd–Warshall algorithm in special cases. The undirected all-pairs shortest paths algorithm presented in [3, 4] runs on a pointer machine in time $O(m\times n\times\alpha(m, n))$ where m and n are the number of edges and vertices, respectively, and $\alpha(m,n)$ is Tarjan's inverse-Ackermann function. It improves upon all previous APSP algorithms when the graph is sparse, i.e., when $m = o(n \log n)$. Seidel's algorithm [5] solves the APSP problem for undirected, unweighted, connected graphs in $O(n^{\omega}\times\log(n))$ expected time, where $\omega < 2.373$. Despite the lower computational complexity, these algorithms cannot be applied to general-case APSP problem. Moreover, they do not solve the graph scaling problem by means of algorithm parallelization on multi-processor systems.

Another approach for solving the APSP scaling problem is the development of shortest paths algorithm versions, which decompose the large graph and adjacency and weight matrices into blocks of smaller size. Such blocks fit in the size of local cache memory and accelerate the computational process. The blocked Floyd–Warshall algorithm that is proposed and investigated in [9 – 16] helped to solve two major problems: 1) to localize the data processing within blocks and thereby reduce the number of exchange operations in hierarchical memory; 2) to organize the parallel computation of blocks on a multi-processor system. The further development of the blocked algorithm was carried out in [17 – 22]. First, the authors proposed the cooperative threaded parallel APSP algorithm. It runs by means of a cooperative threads scheduler. Second, the homogeneous blocked APSP algorithm was extended to a more powerful heterogeneous blocked APSP algorithm.

Solving large-scale problems in reasonable time is impossible without intensive utilizing the parallelism provided by modern multi-processor systems. The paper aims at analyzing the strength and weakness of known APSP algorithms, at identifying ways of developing new block-parallel algorithms that solve the scaling problem at both the level of graph size and at the level of the processor count in a multi-processor system or the core count in the multi-core system. Improving parameters of the algorithms and their implementations can be only achieved by increasing the efficiency of utilizing all resources of the basic computing system (processors and processor cores, all parts of the hierarchical memory, operating system tools etc.)

Analysis of all pairs shortest paths basic algorithms.

Let a directed weighted graph $G = (V, E)$ with a set V of N vertices and a set E of edges be represented by a matrix W of positive edge weights, in which $w_{i,i}=0$ for $i = 0\ldots N\text{-}1$ and $w_{i,j}=\infty$ for $(i, j) \notin E$. The lengths of the shortest paths between pairs of vertices are described by a matrix D. The Floyd-Warshall (FW) algorithm [1, 2] recalculates the D matrix at steps $0\ldots k\ldots N$ corresponding to the vertices of the graph, thus forming a sequence of matrices $D(0)\ldots D(k)\ldots D(N)$, in which $D(0)=W$ and $D(k)$ is the matrix of the lengths of the shortest paths passing through the vertices 0, 1, …, k-1. Matrix $D(N)$ is the resulting matrix of the shortest distances. Figure 1 illustrates the transition from step k-1 to step k, where the length $D_{ij}(k)$ of the shortest path from vertex i to vertex j is calculated over the elements of the $D(k\text{-}1)$ matrix:

$$D_{ij}(k) = \min\left\{ D_{ij}(k-1),\ D_{ik}(k-1) + D_{kj}(k-1) \right\}$$

As the *k* index increases, the row and column are shifted along the *D*(*k*) matrix from top to bottom and from left to right. The pseudo-code of the Floyd-Warshall algorithm (Figure 2) consists of three loops along indices *k*, *i* and *j*. It uses the single *D* matrix of dimension *N*×*N* instead of a sequence *D*(0)…*D*(*N*) of matrices. Since all elements are calculated in the similar way, the algorithm is highly homogeneous, and its computational complexity is O(N^3). Note that due to the accumulation of all intermediate data in one matrix *D*, the permutation of the cycle along *k* with cycles along *i* and *j* leads to incorrect results.

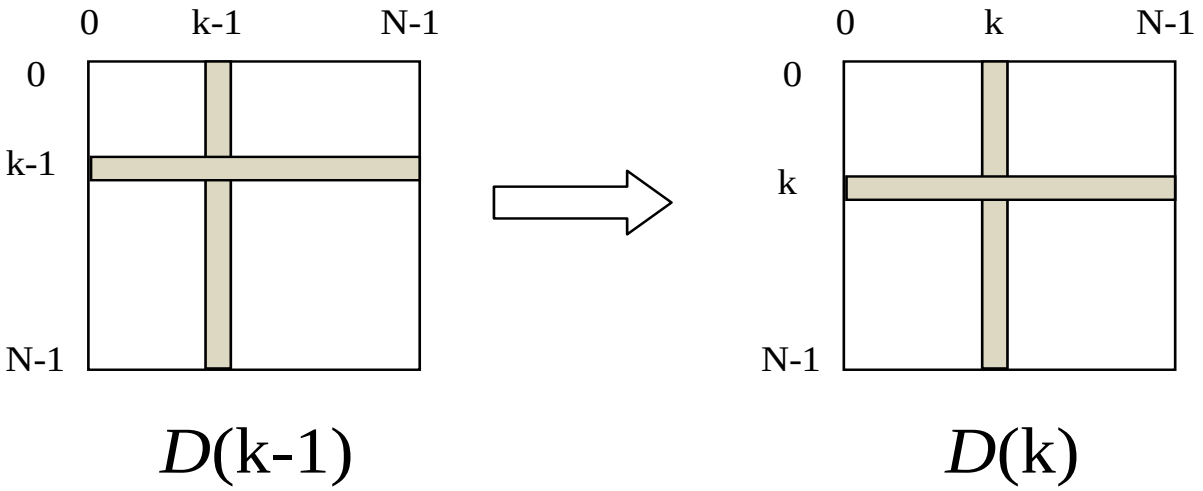


Figure 1. Recalculating pairs shortest path lengths over vertex *k*

```
Algorithm Floyd_Warshall (W) {
    D = W;
    for k=0…N-1  {
        for i=0…N-1  {
            for j=0…N-1  {
                s = Dik + Dkj;
                if(Dij > s) Dij = s;
            }
        }
}}
```

Figure 2. Floyd-Warshall algorithm

The basic Floyd-Warshall algorithm is not capable of processing huge graphs represented by huge matrices, therefore it does not solve the APSP scaling problem. A single thread implements the algorithm. As a result, the thread runs on one core of a multi-core system, and all data pass through a local cache of small size. This leads to a huge number of line misses in the hierarchical memory, therefore cache flaking can be observed during the algorithm execution.

The FW algorithm has no parallelism in its control flow. But the algorithm data flow can be parallelized automatically by an OpenMP compiler, or can be parallelized manually to a multi-thread (multi-process) architecture using thread-based programming facilities or MPI libraries. The drawback of such a solution is that the algorithm as is possesses a hidden parallelism that cannot be extracted automatically by a parallelization compiler. Therefore, parallel modifications of the Floyd-Warshall sequential algorithm have to be developed before creating a program realization. These will be able to

solve the scaling problem and to find the all pairs shortest paths in large graphs on many processors of the basic computing system. The efficient solution of this problem is impossible without developing methods of reducing the consumption of computational resources.

Homogeneous blocked APSP algorithms.

Works [9, 10] generalized the Floyd-Warshall (FW) algorithm to the blocked APSP algorithm (BFW) for finding the shortest paths in a graph whose D matrix of the $N \times N$ dimension is divided into blocks of the $B \times B$ dimension, thus forming a matrix of blocks of the $M \times M$ dimension, where $M = N / B$. Figure 3 illustrates the block algorithm flow and the order of calculating blocks. The calculation of blocks is carried out in a loop along $m = 0 \ldots M\text{-}1$. At first iteration of the loop, the diagonal D0 block with (0, 0) coordinates and the blocks of the cross are calculated, including the blocks C1 of column 0 and blocks C2 of row 0. The remaining peripheral blocks P3 are calculated over blocks C1 and C2. At second iteration of the loop, the diagonal block D0 with coordinates (1, 1) and the blocks of the shifted cross, including the C1 and C2 blocks of column 1 and row 1, are calculated. The peripheral P3 blocks are calculated over blocks C1 and C2. At subsequent iterations, the cross is shifted step by step from the upper left to the lower right corner of the matrix, and the block recalculation procedure remains the same. At one iteration of the loop, one diagonal block, $2 \times (M-1)$ cross blocks and $(M\text{-}1)^2$ peripheral blocks are calculated. At all iterations, BFW recalculates M^3 blocks.

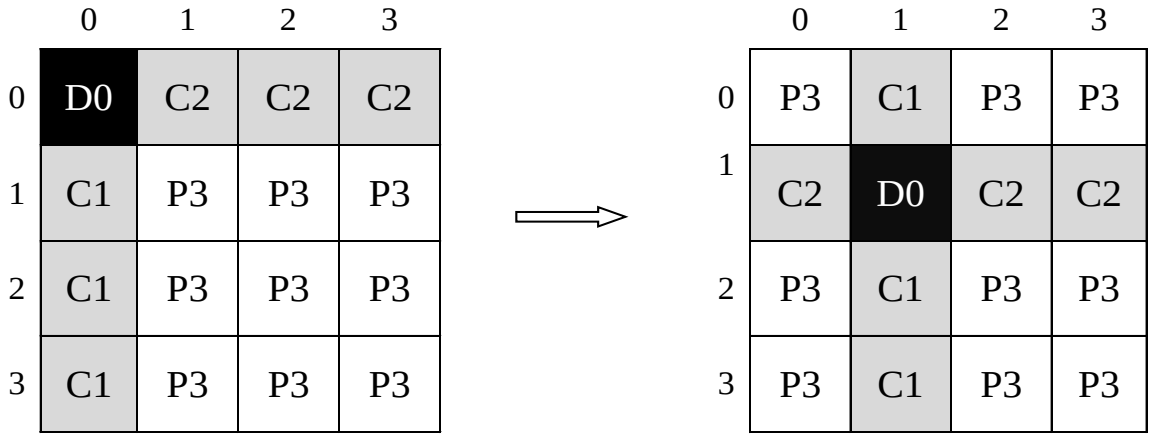


Figure 3. Block calculation flow in the blocked APSP algorithm

Figure 4 depicts the BFW algorithm. Its operation is represented by a loop along index m, at each iteration of which the universal BCA algorithm (Figure 5) recalculate all blocks. This allows the BFW algorithm to be called homogeneous. In Figure 5, argument B^1 is a calculated block, and arguments B^2 and B^3 are blocks, through which the calculation is carried out. M iterations of the loop, recalculate each block M times. The BCA algorithm has the computational complexity of $O(B^3)$. Note that it is difficult to transform and modify the BCA because of its versatility.

```
Algorithm BFW {
    for  m=0…M-1 {
        BCA (Bm,m,Bm,m,Bm,m);        // type D0
        for i=0…m-1 {
            BCA (Bi,m,Bi,m,Bm,m);    // type C1
            BCA (Bm,i,Bm,m,Bm,i);    // type C2
        };
        for i=m+1…M-1 {
            BCA (Bi,m,Bi,m,Bm,m);    // type C1
            BCA (Bm,i,Bm,m,Bm,i);    // type C2
        };
        for i=0…m-1 {
            for j=0…m–1
                BCA (Bi,j,Bi,m,Bm,j);  // type P3
            for j=m+1…M–1
                BCA (Bi,j,Bi,m,Bm,j);  // type P3
        };
        for i=m+1…M-1 {
            for j=0…m–1
                BCA (Bi,j,Bi,m,Bm,j);  // type P3
            for j=m+1…M-1
                BCA (Bi,j,Bi,m,Bm,j);  // type P3
        };
}}
```

Figure 4. Blocked Floyd-Warshall algorithm

```
Algorithm BCA (B1, B2, B3) {
    for k=0…B-1 {
        b3rk = row(B3,k);
        for i=0…B-1  {
            b1ri = row(B1,i);
            b2 = B2i,k;
            for j=0…B-1 {
                bij= b2 + b3rkj;
                if (b1rij > bij) { b1rij = bij;}
            }
}}}
```

Figure 5. Block calculation algorithm

BFW helped to localize the data processing in multi-level memory and to reduce the number of exchange operations between levels, as well as to parallelize the process of calculating blocks. The possibilities of BFW parallelization are shown in [11 – 16]. The block of D0 type works in series with other blocks. All C1 and C2 blocks can work mutually in parallel, and work seriously with D0 and P3 blocks. All P3 blocks can work mutually in parallel.

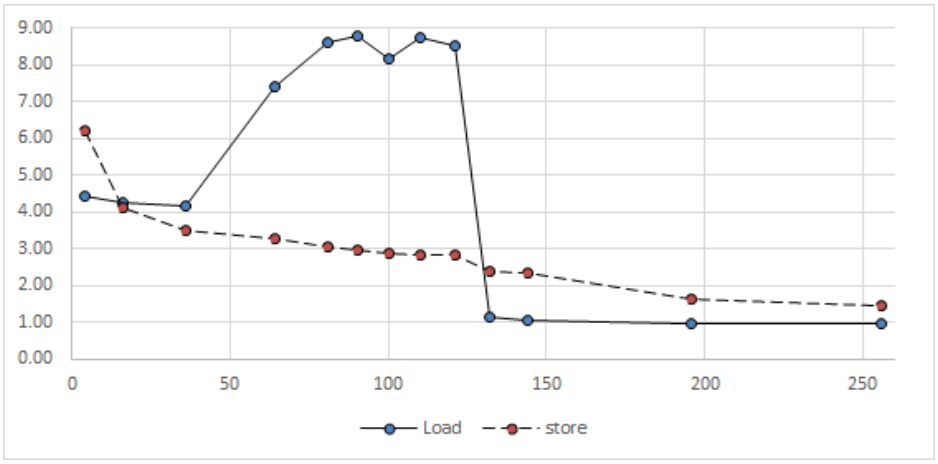


Figure 6. Reduction in number of line read (solid) and write (dash) operations given by BFW against FW (times) vs. matrix size in times to cache size

Comparison of FW and BFW while scaling the APSP problem. We have developed a tool [23] for the simulation of the fully associative cache to find out how the increase in the size of matrix *D* influences the features of FW and BFW. For the matrix size from 4 to 36 times larger to the cache size, the reduction in the number of line reads produced by BFW slightly exceeds 4 times against FW (Figure 6). When the matrix size grows from 64 to 121 times, the reduction reaches 8.79 times. This is a big advantage of BFW

against FW. For a larger matrix where the row size is equal to the three blocks size, the reduction rapidly falls down to 1.0, which means the BFW has no advantage to the FW regarding the usage of cache for solving very large size problems. We can explain this as BFW localizes accesses to lines within one block, but it does not localize data dependencies among blocks.

Heterogeneous blocked APSP algorithms.

Analysis of all calls to the BCA algorithm from the BFW algorithm for calculating a block shows that the calls distinguish four profiles of arguments B^1, B^2 and B^3, which are closely related to the specifics of BCA operation in the BFW algorithm. The profile of type 0, where $B^1=B^2=B^3$, occurs only for the diagonal block D0 (Figure 3). The profile of type 1, where $B^1=B^2\neq B^3$, occurs for C1 blocks of the cross column. The profile of type 2, where $B^1=B^3\neq B^2$, occurs for C2 blocks of the cross row. The profile of type 3, where $B^1\neq B^2\neq B^3$, occurs for peripheral P3 blocks. Differences in the arguments profile and the block behavior itself can be used to find methods that reduce the computational resources consumption during the execution of the BCA algorithm. In particular, the goal is to reduce the operational activity of the cache, as well as to reduce the block computation time. Thus, in [22] it was proposed to use four new more efficient algorithms D0BCA, C1BCA, C2BCA and P3BCA instead of one BCA algorithm. New algorithms are obtained by revising and formal transformation of the FW algorithm, taking into account the peculiarities of calculating blocks of each type. The BFW algorithm that uses the new algorithms of calculating blocks is further called HBFW.

Algorithm D0BCA for calculating diagonal block. Let's change the main principle of the Floyd-Warshall algorithm. The sequence of vertices k of the range from 0 to B-1 will be associated with the process of stepwise adding the vertices to the graph G_m. As a result, a sequence $G_m(0)$, $G_m(1)\ldots$ $G_m(k)\ldots$ $G_m(B\text{-}1)$ of graphs is generated. The matrix of distances between pairs of vertices in the $G_m(k)$ graph is denoted by $B^1(k)$. We represent the block calculation algorithm as a recurrent procedure that calculates the $B^1(k)$ matrix from the $B^1(k\text{-}1)$ matrix and the weights w_{ik} and w_{kj} of the edges connecting the added vertex k with vertices $i, j \in \{0,\ldots,k\text{-}1\}$ (Figure 7).

First, the procedure carries out the A operation of adding the k row and k column to the $B^1(k)$ matrix by means of calculating $B^1{}_{ik}(k)$ as

$$B^1{}_{ik}(k) = \min_{j=1\ldots k-1}\left(B^1{}_{ij}(k-1) + w_{jk}\right)$$

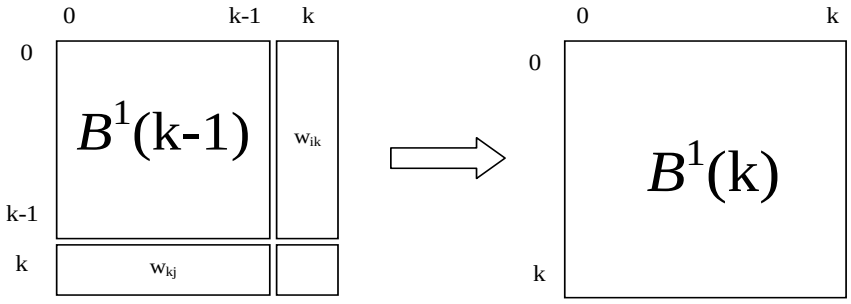


Figure 7. Recurrent procedure of computing the diagonal block and calculating $B^1{}_{kj}(k)$ using a similar formula.

Then the procedure carries out the U operation of updating the $B^1(k\text{-}1)$ matrix to the $B^1(k)$ matrix using the formula

$$B^1{}_{ij}(k) = \min\left\{ B^1{}_{ij}(k-1),\ B^1{}_{ik}(k) + B^1{}_{kj}(k) \right\}$$

The recurrent procedure iteratively executes on all vertices of graph G_m. Multiple execution of the procedure generates a sequence of pairs of the operations: A_0U_0–A_1U_1–…–$A_{N-1}U_{N-1}$. Since the operations

of A_kU_k pair are incompatible in sense of merging the loops along *i* and *j*, and the operations of U_kA_{k+1} pair are compatible, it is preferable to use the resynchronized sequence $U_0A_1–U_1A_2–…–U_{N-2}A_{N-1}–U_{N-1}$ taking into account that A_0 is replaced by a zero initialization of the $B^1(0)$ matrix. In the U_kA_{k+1} pair, operation U_k is a delayed update of the $B^1(k)$ matrix, which is carried out simultaneously with the addition of the *k*+1 vertex. Figure 8 shows the pseudo-code of the D0BCA algorithm, in which d^r and d^c are row *k* and column *k* respectively, and the *row*(B^1,*k*) operation addresses row *k* of $B^1(k)$ matrix. In D0BCA, the number of overall iterations of the most nested loop body is $B^3/3$, which is three times less than in BCA.

After removing the calculation of the diagonal block from the BCA algorithm, it still calculates the C1, C2 and P3 blocks. Its new feature is that it becomes possible to arbitrarily permute the loops along indices *k*, *i* and *j* in six permutation options for each of the C1BCA, C2BCA and P3BCA algorithms. For instance, Figure 9 depicts the P3BCA algorithm that calculates the peripheral block using *i-j-k* permutation.

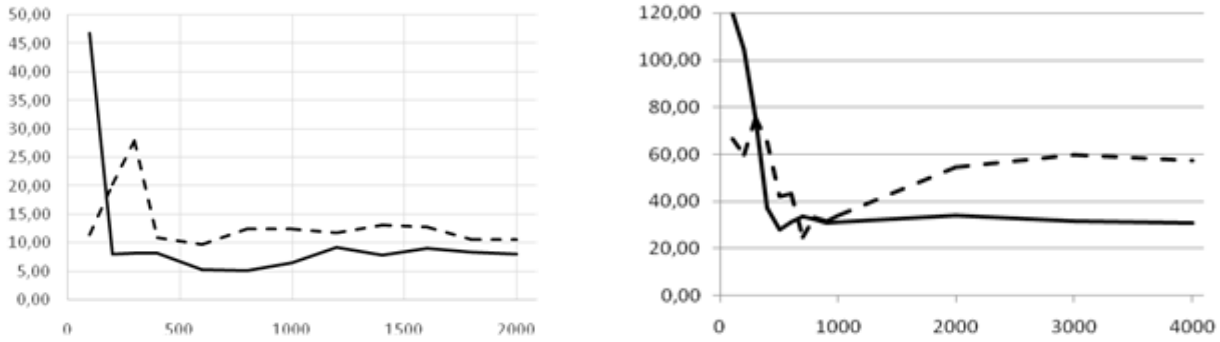
To carrying out computational experiments, we implemented the homogeneous BFW algorithm and heterogeneous HBFW algorithm using OpenMP 3.0. Figure 10a shows that HBFW gives on average an acceleration of 10.88% on Intel® Core ™ i3 CPU 550 @ 3.20 GHz 3.19 GHz (processor cpu1) and of 13.67% on Intel (R) Core (TM) i5-6200UCPU @ 2.20 GHz (processor cpu2) compared to BFW on a 2×2 block matrix. The D0BCA algorithm of calculating the diagonal block (Figure 10b) has mostly contributed in the speedup. It won over 30% on the cpu1 processor against BCA, and won up to 60% on the cpu2 processor.

```
Algorithm D0BCA(B1) {
  dr0 = ∞;  wr0 = B101;
  for k=1…B-1 {
    dc = row(B1,k-1);  wc = row(B1,k);
    for i=0…k-1  { bci = ∞; }
    for i=0…k-1 {
      brmin = ∞;
      for j=0… k-1 {
        z = dri + dcj;  if(B1ij > z) B1ij = z;
        s0 = B1ij + wrj;  if(brmin > s0) brmin = s0;
        s1 = B1ij + wci;  if(bcj > s1) bcj = s1;
      }
      brj = brmin;
    }
    for i=0…k-1  {
      B1ki = bci;  B1ik = dri = bri;  wrj = B1i,k+1;
    }  wrk = B1k,k+1;
  }
  for i=0…B-2 {
    for j=0… B-2 {
      z = dri + dcj;  if(B1ij > z) B1ij = z;
}}}
```

Figure 8. New algorithm D0BCA of recalculating diagonal block

```
Algorithm P3BCA(B1,B2,B3) {
  for i=0…B-1 {
    b2r = row(B2,i);
    d1 r = row(B1,i);
    for j=0…B-1 {
      dmin = ∞;
      for k=0… B-1 {
        s = b2rk + B3kj;
        if(dmin > s) dmin = s;
      }
      if(d1 rj > dmin) d1 rj = dmin;
}}
```

Figure 9. Peripheral block recalculation algorithm P3BCA (index order *i-j-k*)



a) b)

Figure 10. Acceleration (%) of a) HBFW over BFW and b) D0BCA over BCA vs. block size (*B*=100…2000) at *M*=2 and *N*=2**B* on *cpu*1 (solid) и *cpu*2 (dash)

Cooperative multi-thread APSP algorithms.

Nowadays, multi-core processors are widely used in many application domains. The number of cores on a chip goes up to dozens, and for specialized chips it reaches dozens of thousands. The performance of computing system based on a multi-core processor depends very much on the multi-thread software algorithms used, and depends very much on how the algorithms are developed and implemented [24, 25]. From one side, the algorithms can speed up the computing process, but from other side, the algorithms might give no acceleration effect, moreover they may slowdown the computations. In the case of thorough development and optimization, the multi-core-based parallelization may realize speedup factors near the number of cores, or even more if the problem is decomposed to fit in each core's cache, avoiding accesses to slower main memory. To carry out a multi-thread application, the operating system can move threads among different processors. Assigning processors to specific threads can improve performance by eliminating thread migration across processors; such an association between a thread and a processor is called processor affinity.

In [20], a fast cooperative threaded block-parallel algorithm CTBFW is proposed for solving the APSP problem. It is based on the BFW algorithm and on a cooperative model [18, 19] of optimizing the threads execution. The CTBFW algorithm is implemented by a special cooperative scheduler [21].

Let *D* be a matrix M×M of blocks. Let the equalities $N \bmod B = 0$, $M = N / B$ and $M \bmod P = 0$ hold for the block size B, where P is the number of processors. The calculation of the (*i*, *j*) block at the *l* level is denoted by $D^{l}_{i,j}$. Matrix *L* describes the levels of all blocks at each step of the CTBFW operation. To calculate the blocks, *M* treads $t = 0 \ldots M\text{-}1$ are introduced. To localize data, blocks of row *t* of matrix *D* are assigned to the *t* thread. Thread *t* satisfying the condition $t \bmod P = g$ is included and localized on the processor (group) $g \in P$.

CTBFW changes the order of calculating blocks as compared to BFW in order to increase the processor load, reduce data exchange between levels of hierarchical memory, and reduce the time it takes to find the shortest paths. It distinguishes four types of blocks depending on the values of *i*, *j* and *l*: central, horizontal, vertical and peripheral.

The block $D^{l}_{i,j}$ is considered as central if i=j=l–1. Its input data are the $D^{l-1}_{i,j}$ block. The $D^{l}_{i,j}$ block is considered as horizontal if i=l–1 and i≠j. Its input data are $D^{l-1}_{i,j}$ and $D^{l}_{l-1,l-1}$ blocks. The $D^{l}_{i,j}$ and $D^{l}_{l-1,l-1}$ blocks are located in one thread i. The $D^{l}_{i,j}$ block is considered as vertical if j=l–1 and i≠j. Its input data are $D^{l-1}_{i,j}$ and $D^{l}_{l-1,l-1}$ blocks. Dependence of the $D^{l}_{i,j}$ block on the $D^{l}_{l-1,l-1}$ block is resolved through the *switch thread* operation, if these are blocks of the same group, otherwise, through the *wait for* and *notify set* operations, if these are blocks of different groups.

The $D^{l}_{i,j}$ block is considered as peripheral for i≠l–1 and j≠l–1. Its input data are $D^{l-1}_{i,j}$, $D^{l}_{i,l-1}$ and $D^{l}_{l-1,j}$ blocks. Despite the presence of the dependence of the $D^{l}_{i,j}$ block on the $D^{l}_{i,l-1}$ block, the dependence resolution is not required, since both blocks are in the same row of *D* matrix. The resolution of the $D^{l}_{i,j}$ block dependence on the $D^{l}_{l-1,j}$ block is performed using the *switch thread* operation, if these

are blocks of the same group, otherwise by means of *wait for* and *notify set* operations, if these are blocks of different groups.

In [9], when proving the correctness of the proposed recursive block algorithm, it was shown that the $D^{l}_{i,j}$ block can be correctly calculated through $D^{l-1}_{i,j}$, $D^{v}_{i,l-1}$ and $D^{u}_{l-1,j}$ blocks, in case when $v \geq l-1$ and $u \geq l-1$. The CTBFW algorithm uses this relaxation of requirements to reorder block computations in order to more efficiently parallelize threads, increase processor utilization, and improve the localization of data access to the cache. Each thread implements the algorithm that switches from one operating mode to another. CTBFW uses six modes: master, slave, complement, passive_A, passive_B and passive_C. Figure 11 depicts the step-by-step parallel operation of the CTBFW algorithm on a matrix *D* [4×4] and on two processors. The algorithm is implemented in two versions: homogeneous and heterogeneous. Table 1 reports results that show the average gain of 6.16% the heterogeneous HCTBFW algorithm has against the homogeneous CTBFW, and show the average gain of 10.99% (without affinity of threads to processors) the HBFW that is realized using OpenMP has against BFW, and the gain of 11.37% (with affinity) the HBFW has against BFW.

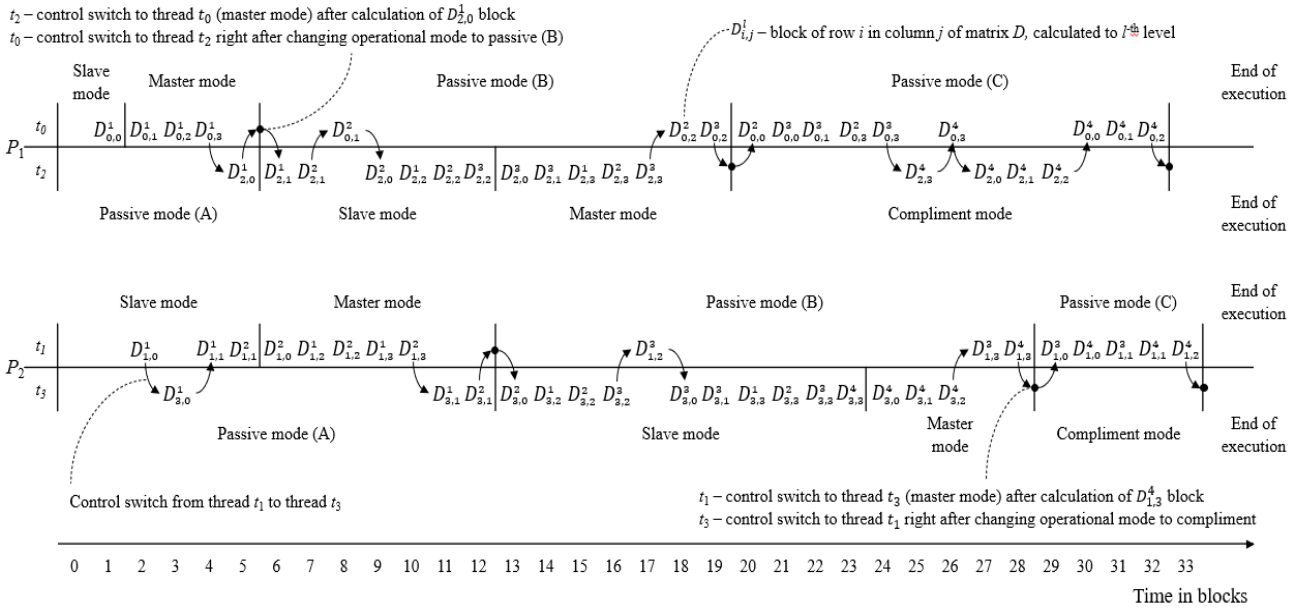


Figure 11. Timing diagram of CTBFW operation on a matrix of 4×4 blocks on two processors (the time unit is the time of calculating one block at one level)

Table 1. Comparison of parallel heterogeneous HBFW to parallel homogeneous blocked APSP algorithms

Graph size in node count	HCTBFW over CTBFW	HBFW over BFW	
		without affinity	with affinity
2400	6.62%	9.67%	9.74%
4800	6.47%	11.68%	12.90%
9600	5.39%	11.63%	11.46%

Conclusion. The basic Floyd-Warshall algorithm does not solve the APSP scaling problem since it is not capable of processing huge graphs. The direct parallelization of the algorithm leads to multi-thread applications, which cannot manage the computational resources of a multi-core system efficiently. The blocked Floyd-Warshall algorithm solves two main tasks: it localizes the data processing within caches and reduces the number of exchange operations between the hierarchical memory levels. It also allows the parallel execution of the blocks. The next step in solving the APSP scaling problem is the heterogeneous blocked APSP algorithm. Its advantage to the homogeneous blocked algorithm is the usage

of a separate specific block calculation algorithm for each of four block types, which takes into account the specific features of computing architecture and accelerates computations. The contribution of the CTBFW algorithm in the APSP scaling is the reduction of processor time spent on the thread execution control, and the increase of core load. Moreover, the algorithm efficiently works together with the heterogeneous algorithm. Our further research will be devoted to the development of memory management techniques embedded in the APSP algorithms. For instance, the assumption that all blocks of a block matrix row are assigned to one processor can be applied to a distributed memory multiprocessor system, but it is not justified for a shared memory multi-core system where the blocks do not fit in the core local cache.

References

[1] Floyd, R.W. Algorithm 97: Shortest path / R.W. Floyd // Communications of the ACM, 1962, 5(6), p.345.

[2] Hofner, P. Dijkstra, Floyd and Warshall Meet Kleene / P. Hofner and B. Moller // Formal Aspect of Computing, Vol.24, No.4, 2012, № 2, pp. 459-476.

[3] Pettie, S. Computing shortest paths with comparisons and additions / S. Pettie, V. Ramachandran // Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 2002, pp. 267–276.

[4] Pettie, S. A new approach to all-pairs shortest paths on real-weighted graphs / S. Pettie // Theoretical Computer Science. 312 (1), 2004: 47–74.

[5] Seidel, R. On the All Pairs Shortest paths Problem in Unweighted Undirected Graphs / R. Seidel // Journal of Computer and System Sciences. 51 (3), 1995, pp. 400-403.

[6] Anu, P. Finding All-Pairs Shortest Path for a Large-Scale Transportation Network Using Parallel Floyd-Warshall and Parallel Dijkstra Algorithms / P. Anu, M. G. Kumar // Journal of Computing in Civil Engineering. – 2013. – Vol. 27, №. 3. – P. 263–273.

[7] Floyd-Warshall all-pair shortest path for accurate multi-marker calibration / L. Wang [et al.] // 2010 IEEE International Symposium on Mixed and Augmented Reality. – Seoul, South Korea: IEEE, 2010. – P. 277–278.

[8] Ridi, L. Developing a Scheduler with Difference-Bound Matrices and the Floyd-Warshall Algorithm / L. Ridi, J. Torrini, E. Vicario // IEEE Software. – 2012. – Vol. 29, №. 1. – P. 76–83.

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

[10] Park, J.S. Optimizing graph algorithms for improved cache performance / J.S. Park, M. Penner, and V.K. Prasanna // IEEE Trans. on Parallel and Distributed Systems, 2004, 15(9), pp.769-782.

[11] Albalawi, E. Task Level Parallelization of All Pair Shortest Path Algorithm in OpenMP 3.0 / E. Albalawi, P. Thulasiraman, R. Thulasiram // 2nd International Conference on Advances in Computer Science and Engineering (CSE 2013), 2013, Los Angeles, CA, July 1-2, 2013, pp. 109-112.

[12] Tang, P. Rapid Development of Parallel Blocked All-Pairs Shortest Paths Code for Multi-Core Computers / P. Tang // IEEE SOUTHEASTCON 2014, pp. 1-7.

[13] Solomonik, E. Minimizing Communication in All Pairs Shortest Paths / E. Solomonik, A. Buluc, and J. Demmel // IEEE 27th International Symposium on Parallel & Distributed Processing, 2013, pp.548-559.

[14] Singh, A. Performance Analysis of Floyd Warshall Algorithm vs Rectangular Algorithm / A. Singh, P.K. Mishra // International Journal of Computer Applications, Vol.107, No.16, 2014, pp. 23-27.

[15] Madduri, K. An Experimental Study of a Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances / K Madduri,.D. Bader, J.W. Berry, J.R. Crobak // Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX), 2007, pp.23-35.

[16] Лиходед Н.А, Сипейко Д.С. Обобщенный блочный алгоритм Флойда – Уоршелла. Журнал Белорусского государственного университета. Математика. Информатика. 2019;3: 84 – 92.

[17] Прихожий, А.А. Исследование методов реализации многопоточных приложений на многоядерных системах / А.А. Прихожий, О.Н. Карасик // Информатизация образования, 2014, № 1, с. 43-62.

[18] Прихожий, А.А. Кооперативная модель оптимизации выполнения потоков на многоядерной системе / А.А. Прихожий, О.Н. Карасик // Системный анализ и прикладная информатика, 2014, № 4..

[19] Прыхожы, А. А. Кааператыўныя блочна-паралельныя алгарытмы рашэння задач на шмат'ядравых сістэмах / А. А. Прыхожы, А. М. Карасік // Сістэмны аналіз і прыкладная інфарматыка. – 2015. – № 2. – С. 10–18.

[20] Карасик, О. Н. Потоковый блочно-параллельный алгоритм поиска кратчайших путей на графе / О. Н. Карасик, А. А. Прихожий // Доклады БГУИР. – 2018. – № 2. – С. 77–84.

[21] Карасик, О. Н. Усовершенствованный планировщик кооперативного выполнения потоков на многоядерной системе / О. Н. Карасик, А. А. Прихожий // Системный анализ и прикладная математика. – 2017. – № 1. – С. 4–11.

[22] Прихожий, А. А. Разнородный блочный алгоритм поиска кратчайших путей между всеми парами вершин графа / А. А. Прихожий, О. Н. Карасик // Системный анализ и прикладная информатика. – № 3. – 2017. – С. 68–75.

[23] 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; (4):10 --18.

[24] Prihozhy, A.A. Asynchronous scheduling and allocation / A.A. Prihozhy / Proceedings Design, Automation and Test in Europe. Paris, France. – IEEE, 1998, pp. 963-964.

[25] Prihozhy, A.A. Analysis, transformation and optimization for high performance parallel computing / A.A. Prihozhy / Minsk: BNTU, 2019. – 229 p.

ПОИСК КРАТЧАЙШИХ ПУТЕЙ МЕЖДУ ВСЕМИ ПАРАМИ ВЕРШИН В ГРАФАХ БОЛЬШОГО РАЗМЕРА

А.А. ПРИХОЖИЙ
Профессор кафедры «Программное обеспечение информационных систем и технологий» Белорусского национального технического университета, д.т.н., профессор

О.Н. КАРАСИК
Ведущий инженер иностранного производственного унитарного предприятия «ИССОФТ СОЛЮШЕНЗ» (ПВТ, г. Минск), к.т.н.

Беларуский национальный технический университет, Беларусь
ИСсофт Солюшенс (часть Кохерент Солюшенс), Беларусь
E-mail: prihozhy@yahoo.com, karasik.oleg.nikolaevich@gmail.com

Аннотация. Проблема поиска кратчайших путей между всеми парами вершин графа имеет много разнообразных областей практического применения. В статье дан анализ известных алгоритмов ее решения и рассмотрена проблема распараллеливания и масштабирования к размеру графа и архитектуре многопроцессорной системы. Исследован класс блочно-параллельных алгоритмов поиска кратчайших путей, изучены их достоинства и недостатки. Путем имитационного моделирования показано, что при определенных соотношениях размера графа и размера кэш памяти, блочный алгоритм дает многократное сокращение обмена данными между уровнями иерархической памяти, однако при слишком большом увеличении размера графа его характеристики приближаются к характеристикам алгоритма Флойда-Уоршелла. С целью повышения производительности, однородный блочный алгоритм трансформирован в разнородный, сокращающий время расчета одного блока. Дальнейшее улучшение характеристик достигнуто за счет разработки кооперативного потокового планировщика и блочно-параллельного алгоритма, ориентированного на графы большого размера и отличающегося изменением порядка расчета блоков, локализацией обработки данных, сокращением критического пути, уменьшением времени работы на многоядерной системе, улучшением работы иерархической памяти. Исследования показали, что параллельный потоковый алгоритм хорошо сочетается с неоднородным алгоритмом, при этом время решения задач сокращается. Выполненный анализ и проведенные вычислительные эксперименты позволили наметить направления дальнейшего развития моделей и алгоритмов решения проблемы поиска кратчайших путей и масштабирования этой проблемы.

Ключевые слова: кратчайший путь, алгоритм Флойда-Уоршелла, масштабирование, блочный алгоритм, разнородный блочный алгоритм, многопоточный алгоритм, кооперативное выполнение, многопроцессорная система, иерархическая память.