UDC 004.272.2

O. N. Karasik[1], A. A. Prihozhy[2]
[1]ISsoft Solutions (part of Coherent Solutions)
[2]Belarusian National Technical University

RECONFIGURABLE HETEROGENEOUS BLOCKED SHORTEST PATH ALGORITHM FOR CLUSTERED GRAPHS

Transportation systems, social networks, network configurations, and many other systems in the surrounding world can be represented by clustered graphs consisting of weakly connected dense clusters. Finding shortest paths in such large-size graphs is a challenging but computationally difficult problem. Heterogeneous block algorithms that utilize information about clusters and the connections between them are a promising approach to solving this problem. The paper develops a new reconfigurable heterogeneous blocked all-pairs shortest paths algorithm for the clustered graphs. The algorithm is based on two new sub-algorithms for calculating cross blocks which account for the reduced communication between clusters and the features of multi-core processors. The paper provides results of two-series experiments, carried out on samples of clustered 9600-vertex graphs at the aim of analysis and comparison of the new algorithm with known techniques. The experiments convincingly demonstrate that the new feature of the algorithm to reconfigure its structure depending on the graph properties provides the speed up from 17.05% to 29.62% in single-threaded and from 5.03% to 14.72% in multi-threaded implementations.

Keywords: shortest path, blocked algorithm, heterogeneous algorithm, reconfiguration, multi-core system, throughput.

For citation: Karasik O. N., Prihozhy A. A. Reconfigurable heterogeneous blocked shortest path algorithm for clustered graphs. *Proceedings of BSTU, issue 3, Physics and Mathematics. Informatics*, 2026, no. 1 (302), pp. 118–127.

DOI: 10.52065/2520-6141-2026-302-10.

О. Н. Карасик[1], А. А. Прихожий[2]
[1]Иностранное производственное унитарное предприятие «Иссофт Солюшенз»
[2]Белорусский национальный технический университет

РЕКОНФИГУРИРУЕМЫЙ НЕОДНОРОДНЫЙ БЛОЧНЫЙ АЛГОРИТМ ПОИСКА КРАТЧАЙШИХ ПУТЕЙ В КЛАСТЕРИЗОВАННЫХ ГРАФАХ

Системы транспортного сообщения, социальные сети, сетевые конфигурации а также многие другие системы окружающего мира могут быть представлены кластеризованными графами, состоящими из слабо связанных плотных кластеров (блоков). Поиск кратчайших путей в таких больших графах – актуальная, но вычислительно трудная задача. Гетерогенные блочные алгоритмы, использующие информацию о кластерах и связях между ними, являются перспективным подходом к решению этой задачи. В данной статье мы разрабатываем новый реконфигурируемый гетерогенный блочный алгоритм для поиска кратчайших путей между всеми парами вершин в кластеризованном графе. Алгоритм основан на двух новых подалгоритмах расчета перекрестных блоков, которые учитывают слабую связность кластеров и специфические особенности многоядерных процессоров. В статье представлены результаты двух серий экспериментов, проведенных на выборках кластерных графов с 9600 вершинами в целях анализа и сравнения нового алгоритма с существующими методами. Эксперименты убедительно демонстрируют, что новое свойство алгоритма, позволяющее осуществлять реконфигурацию в зависимости от свойств графа, обеспечивает увеличение скорости с 17,05% до 29,62% в однопоточных реализациях и с 5,03% до 14,72% в многопоточных реализациях.

Ключевые слова: кратчайший путь, блочный алгоритм, разнородный алгоритм, реконфигурация, многоядерная система, производительность.

Для цитирования: Карасик О. Н., Прихожий А. А. Реконфигурируемый неоднородный блочный алгоритм поиска кратчайших путей в кластеризованных графах // Труды БГТУ. Сер. 3, Физико-математические науки и информатика. 2026. № 1 (302). С. 118–127 (На англ.).

DOI: 10.52065/2520-6141-2026-302-10.

Introduction. The problem of finding shortest paths in a graph is a classic computer science problem. It remains an important research topic in terms of performance, security, privacy and operational planning [1–5]. The problem can be formulated in two ways. One is to find shortest paths from one vertex to all other vertices in the graph (Single Source Shortest Path or *SSSP*). Another is to find shortest paths between all pairs of vertices (All-Pairs Shortest Path or *APSP*).

Both, SSSP and APSP have well-known classic solutions – Dijkstra algorithm [6] for SSSP and Floyd-Warshall algorithm [7] for APSP. In scope of this work, we are focusing on the APSP formulation of the problem.

The *Floyd-Warshall* (*FW*) algorithm works with graph G of V vertices and E edges, represented with a weight-adjacency matrix D of $N \times N$ size (where N is the number of vertices in G). The algorithm iterates over all vertices and checks the path from a pair of vertices (from i to j) through an intermediate vertex k. The algorithm has $O(n^3)$ computational complexity and suffers performance penalties when the matrix does not fit into CPU LLC (Last Level Cache).

In [8] authors propose *Blocked Floyd-Warshall (BFW)* algorithm, which divides matrix D into blocks of size S to create a block matrix B of size M. The algorithm calculates these blocks in a particular order to maintain data dependencies. All blocks are calculated using the same sub-algorithm. *BFW* has $O(n^3)$ time complexity the same as *FW*. However, the block-based calculation introduces spatial locality and enables the algorithm to work with matrices that do not fit in the CPU LLC.

In our previous works [9–12], we have introduced multiple algorithms which extend and significantly modify the *BFW* algorithm to use different sub-algorithms for every type of block, to work with blocks of unequal sizes, to use information about graph's clusters, and to calculate blocks through bridge vertices. *BFW* and the algorithms introduced in our earlier works calculate the same number of blocks of four types (*M diagonal*; M^2*–M vertical*, M^2*–M horizontal*, and $M\,(M-1)^2$ *peripheral* blocks). Because of such decomposition, the execution time is dominated by the calculation of *peripheral* blocks, which in large graphs diminishes the impact of diagonal, vertical and horizontal blocks. In our latest work [13], we introduced the *Heterogenous Blocked Shortest Path algorithm for Clustered Graphs (HBSPCG)* (Fig. 1), which uses information about clusters to calculate *horizontal* blocks through output vertices of cluster, the *vertical* blocks through input vertices, and the *peripheral* blocks through the smallest number of input or output vertices. This results in situations when *HBSPCG* might skip some blocks or significantly reduce the number of operations to calculate them. For instance, when the cluster has only input or output bridge vertices, *peripheral* blocks won't be calculated.

HBSPCG changes how the computations are distributed between different types of blocks, making the optimisation of the block calculation sub-algorithms extremely actual.

In this work, we introduce a new *Reconfigurable Heterogeneous Blocked Shortest Path algorithm for Clustered Graphs* (*RHBSPCG*), which extends the *HBSPCG* with a new logic to dynamically select the most effective sub-algorithms to calculate vertical and horizontal blocks depending on the cluster information. We also propose new highly efficient sub-algorithms for calculating the vertical and horizontal blocks.

Main part. *The C1CGE and C2CGE sub-algorithms*. In [14] we proposed new sub-algorithms *C1CGE* and *C2CGE* for the calculation of *vertical* and *horizontal* blocks (these sub-algorithms work with the same input graph as the *C1CG* and *C2CG* sub-algorithms from *HBSPCG*).

C1CGE and *C2CGE* use information about the clusters to avoid unnecessary calculations. They do so by starting the most outer k loop from the first bridge vertex (input for *C1CGE* and output for *C2CGE*) and by checking in the loop body whether the vertex is a bridge.

As was proven in the original work [14], *C1CGE* and *C2CGE* reduce the number of iterations and improve data locality. However, the final performance of sub-algorithms depends on the configuration of the input graph:

1. When the first bridge index is in the beginning of the vertex list, then the sub-algorithms would perform *(cluster size) – (index of first bridge vertex) – (count of bridge vertexes)* unnecessary iterations of the outermost loop k.

2. When checking if a vertex is a bridge inside the innermost *j* loop – the check is on the hot execution path and impacts branch prediction and vectorization.

```
algorithm HBSPCG(B, C)
 for (m = 0; m < SIZE(B); ++m)
  D0CG(B[m,m])
  for (i = 0; i < SIZE(B) && i != m; ++i)
   C1CG(B[i,m], B[m,m], C[m])
   C2CG(B[m,i], B[m,m], C[m])
  for (i = 0; i < SIZE(B) && i != m; ++i)
   for (j = 0; j < SIZE(B) && j != m; ++j)
    P3CG(B[i,j], B[i,m], B[m,j], C[m])

function D0CG(B1)
 for (k = 1; k < SIZE(B1); ++k)
  for (i = 0; i < k; ++i)
   for (j = 0; j < k; ++j)
    B1[i,j] = MIN(B1[i,j], B1[i,k-1] + B1[k-1,j])
    B1[i,k] = MIN(B1[i,k], B1[i,j] + B1[j,k])
    B1[k,j] = MIN(B1[k,j], B1[k,i] + B1[i,j])

 x = SIZE(B1)-1

 for (i = 0; i < x; ++i)
  for (j = 0; j < x; ++j)
   B1[i,j] = MIN(B1[i,j], B1[i,x] + B1[x,j]

function C1CG(B1, B2, CM)
 for (i = 0; i < HEIGHT(B1); ++i)
  for (k in INPUT(CM))
   for (j = 0; j < WIDTH(B1); ++j)
    B1[i,j] = MIN(B1[i,j], B1[i,k] + B2[k,j])

function C2CG(B1, B2, CM)
 for (i = 0; i < HEIGHT(B1); ++i)
  for (k in OUTPUT(CM))
   for (j = 0; j < WIDTH(B1); ++j)
    B1[i,j] = MIN(B1[i,j], B2[i,k] + B1[k,j])

function P3CG (B1, B2, B3, CM)
 for (i = 0; i < HEIGHT(B1); ++i)
  for (k in MIN_ARRAY(INPUT(CM), OUTPUT(CM))
   for (j = 0; j < WIDTH(B1); ++j)
    B1[i,j] = MIN(B1[i,j], B2[i,k] + B3[k,j])
```

Fig. 1. Pseudocode of algorithm *HBSPCG*
and sub-algorithms *D0CG*, *C1CG*, *C2CG*, and *P3CG*

In the original work, we pointed out that these unnecessary computations could be avoided by grouping all bridges at the end of the vertex list. In this case, the sub-algorithms would perform a certain number of iterations over the bridges and would eliminate all checks to see if a vertex is a bridge. The pseudocode of the modifications of *C1CGE* and *C2CGE* that assume that the bridge vertices are at the end of the list is shown in Fig. 2.

Weight-adjacency matrix representation of the graph. To accommodate the requirements of *C1CGE* and *C2CGE* sub-algorithms the vertices in every cluster of the matrix need to be reordered. The sub-algorithms operate on a graph represented as a blocked weight-adjacency matrix where diagonal blocks describe clusters (this configuration is achieved through a matrix transformation that groups the vertices of clusters together [12]).

All vertices in the cluster are divided into four categories: ordinary (V_O), input (V_{IN}), output (V_{OUT}), and input-output ($V_{IN\text{-}OUT}$). Table 1 presents arrangements that provide the best conditions for the new sub-algorithms (when the sub-algorithm performs the minimal number of iterations).

Clearly, if a cluster contains all three types of bridge vertices, the vertex permutation can be performed to ensure the best conditions for one of the sub-algorithms. However, if the cluster has (V_{IN} and $V_{IN\text{-}OUT}$, or V_{OUT} and $V_{IN\text{-}OUT}$, or just $V_{IN\text{-}OUT}$), then the vertex permutation can ensure best conditions for both *C1CGE* and *C2CGE* sub-algorithms.

```
function C1CGE(B1, B2, CM)
 first = FIRST(INPUT(CM))
 w     = WIDTH(B1)
 h     = HEIGHT(B1)

 for (k = first+1; k < w; ++k)
  for (i = 0; i < h; ++i)
   for (j = 0; j < k; ++j)
    B1[i,j] = MIN(B1[i,j], B1[i,k-1] + B2[k-1,j])
    B1[i,k] = MIN(B1[i,k], B1[i,j] + B2[j,k])
 for (i = 0; i < h; ++i)
  for (j = 0; j < w-1; ++j)
   B1[i,j] = MIN(B1[i,j], B1[i,w-1] + B2[w-1,j])

function C2CGE(B1, B2, CM)
 first = FIRST(OUTPUT(CM))
 w     = WIDTH(B1)
 h     = HEIGHT(B1)

 for (k = first+1; k < h; ++k)
  for (i = 0; i < k; ++i)
   for (j = 0; j < w; ++j)
    B1[i,j] = MIN(B1[i,j], B2[i,k-1] + B1[k-1,j])
    B1[k,j] = MIN(B1[k,j], B2[k,i] + B1[i,j])
 for (i = 0; i < h-1; ++i)
  for (j = 0; j < w; ++j)
   B1[i,j] = MIN(B1[i,j], B2[i,h-1] + B1[h-1,j])
```

Fig. 2. Pseudocode of *C1CGE* and *C2CGE* sub-algorithms. *B*1 is a vertical rectangular block in *C1CGE* and is a horizontal block in *C2CGE*, *B*2 is a diagonal square block. The *FIRST* function returns the first bridge in the array

Clusters in which vertices can be rearranged to provide the best conditions for both sub-algorithms simultaneously are referred to as VHC (Vertical Horizontal Cluster).

Fig. 3 shows an example of the input graph and the corresponding blocked matrix after all transformations.

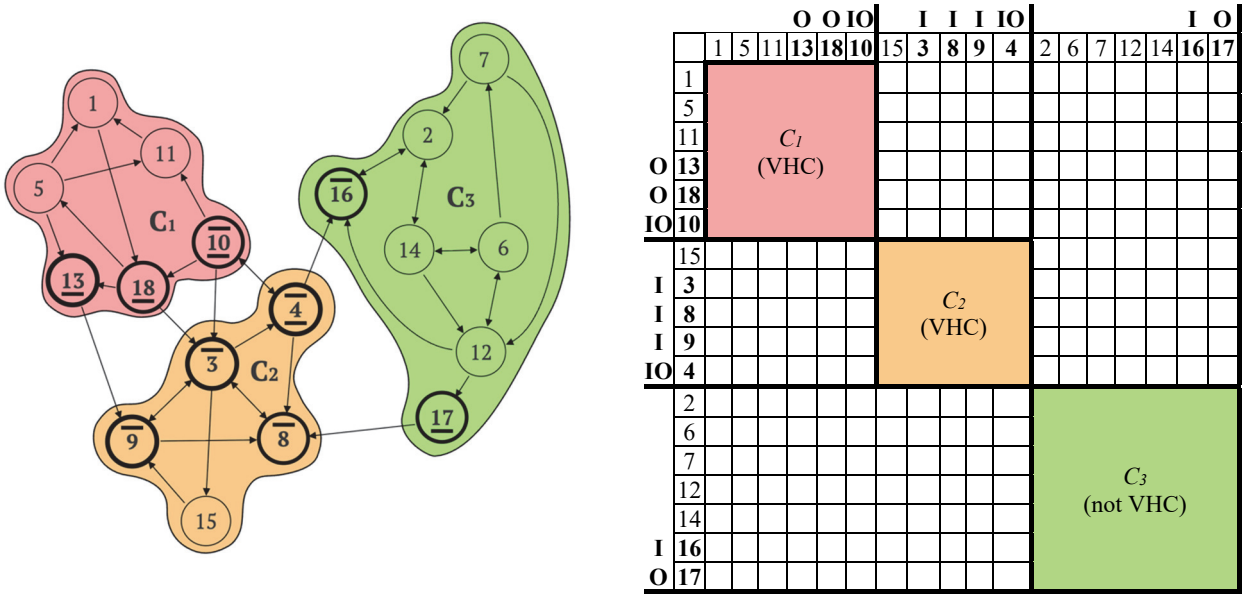

Fig. 3. Illustration of graph *G* divided into three weakly-connected clusters C_1, C_2 and C_3 (on the left) and weight-adjacency matrix representation of the graph with the clusters aligned along the matrix's principal diagonal (on the right); vertices marked **I** are input bridges, marked **O** are output bridges, and marked **IO** are input-output bridges; the vertices are ordered and moved to the end of the cluster's vertex list

Improvement of sub-algorithms C1CGE and C2CGE. In their current form, the *C1CGE* and *C2CGE* sub-algorithms offer more opportunities for improvements.

Modification of *C1CGE* sub-algorithm:

1. Update the loop order from *k-i-j* to *i-k-j*.

2. Split the innermost loop *j* into two loops (from *0* to *FIRST* – hereinafter loop *A*, and from *FIRST* to *k* – hereinafter loop *B*).

3. Use transposed diagonal block *B2T* in loop *B* to compute the *B1*[*i*, *k*] values.

4. Remove computation of *B1*[*i*, *k*] from loop *A* because it is required only for the input vertices.

Together, steps 3 and 4 avoid column access and unnecessary computation of the *B1*[*i*, *k*] values. The *C1CGE* sub-algorithm with above modifications is denoted *C1CGEX*.

Modification of *C2CGE* sub-algorithm:

1. Split the outermost loop *k* into two loop nests (the first nest – hereinafter nest *C* – where outer loop *i* is from 0 to *FIRST* and the second nest – hereinafter nest *D* – where outer loop *i* is from *FIRST* to *k*).

2. Update the order of loops from *k-i-j* to *i-k-j* in nest *C*.

3. Use transposed diagonal block *B2T* in nest *D* to compute *B1*[*i*, *j*] values.

4. Remove the computation of the *B1*[*k*, *j*] values from the nest *C* because it is required only for the output vertices.

Together, steps 3 and 4 avoid column access and unnecessary computation of *B1*[*i*, *k*] values. The *C2CGE* sub-algorithm with above modifications is denoted *C2CGEX*.

Fig. 4 depicts the pseudocode of the *C1CGEX* and *C2CGEX* sub-algorithms.

```
function C1CGEX(B1, B2, B2T, CM)
 first = FIRST(INPUT(CM))
 w = WIDTH(B1)
 h = HEIGHT(B1)

 for (i = 0; i < h; ++i)
  for (k = first+1; k < w; ++k)
   for (j = 0; j < first; ++j) // A
    B1[i,j] = MIN(B1[i,j], B1[i,k-1] + B2[k-1,j])
   for (j = first; j < k; ++j) // B
    B1[i,j] = MIN(B1[i,j], B1[i,k-1] + B2[k-1,j])
    B1[i,k] = MIN(B1[i,k], B1[i,j] + B2T[k,j])

 for (i = 0; i < h; ++i)
  for (j = 0; j < w-1; ++j)
   B1[i,j] = MIN(B1[i,j], B1[i,w-1] + B2[w-1,j])

function C2CGEX(B1, B2, B2T, CM)
 first = FIRST(OUTPUT(CM))
 w = WIDTH(B1)
 h = HEIGHT(B1)

 for (i = 0; i < first; ++i)
  for (k = first+1; k < h; ++k) // C
   for (j = 0; j < w; ++j)
    B1[i,j] = MIN(B1[i,j], B2[i,k-1] + B1[k-1,j])
 for (k = first+1; k < h; ++k)  // D
  for (i = first; i < h; ++i)
   for (j = 0; j < w; ++j)
    B1[i,j] = MIN(B1[i,j], B2T[k-1,i] + B1[k-1,j])
    B1[k,j] = MIN(B1[k,j], B2[k,i] + B1[i,j])

 for (i = 0; i < h-1; ++i)
  for (j = 0; j < w; ++j)
   B1[i,j] = MIN(B1[i,j], B2[i,h-1] + B1[h-1,j])
```

Fig. 4. Pseudocode of sub-algorithms *C1CGEX* and *C2CGEX*. *B*1 is a vertical rectangular block in *C1CGEX* and is a horizontal block in *C2CGEX*. *B*2 is a diagonal square block, and *B*2*T* is transposed *B*2 block

Reconfigurable Heterogeneous Blocked Shortest Paths algorithm for Clustered Graphs (RHBSPCG). The main idea of the *RHBSPCG* algorithm lies in using the *C1CGEX* and *C2CGEX* sub-algorithms to calculate vertical and horizontal blocks in the cases of best conditions presented

in the Table 1. In the rest of the cases, the *RHBSPCG* algorithm yields to the *C1CG* and *C2CG* sub-algorithms from the *HBSPCG* algorithm.

Table 1. **Arrangement of vertices in a cluster that provide the best conditions for the *C1CGE* and *C2CGE* sub-algorithms. A – when the cluster has V_{IN}, V_{OUT} and $V_{IN\text{-}OUT}$ vertices, *B* – when the cluster has V_{IN} and $V_{IN\text{-}OUT}$ or V_{OUT} and $V_{IN\text{-}OUT}$, and *C* – when the cluster has $V_{IN\text{-}OUT}$ vertices. The cells in bold indicate the arrangements that are best conditions for both sub-algorithms. In all arrangements, the V_O vertices are grouped at the start of the cluster**

Sub-algorithm	*A*	*B*	*C*
C1CGE	$V_{OUT} \rightarrow V_{IN} \rightarrow V_{IN\text{-}OUT}$	$\boldsymbol{V_{IN} \rightarrow V_{IN\text{-}OUT}}$	$\boldsymbol{V_{IN\text{-}OUT}}$
	$V_{OUT} \rightarrow V_{IN\text{-}OUT} \rightarrow V_{IN}$	$V_{IN\text{-}OUT} \rightarrow V_{IN}$	
C2CGE	$V_{IN} \rightarrow V_{OUT} \rightarrow V_{IN\text{-}OUT}$	$\boldsymbol{V_{OUT} \rightarrow V_{IN\text{-}OUT}}$	$\boldsymbol{V_{IN\text{-}OUT}}$
	$V_{IN} \rightarrow V_{IN\text{-}OUT} \rightarrow V_{OUT}$	$V_{IN\text{-}OUT} \rightarrow V_{OUT}$	

To implement the idea, the *RHBSPCG* algorithm extends (Fig. 5) the main loop of the *HBSPCG* algorithm:

– It creates a transposed block *BMT* made from the *B*[*m*, *m*] block.

– If the current cluster is VHC, then horizontal and vertical blocks are calculated using *C1CGEX* and *C2CGEX* sub-algorithms.

– If the current cluster is not VHC and $V_{IN} > V_{OUT}$ then vertical and horizontal blocks are calculated using *C1CGEX* and *C2CG* sub-algorithms.

– If the current cluster is not VHC and $V_{IN} < V_{OUT}$ then vertical and horizontal blocks are calculated using *C1CG* and *C2CGEX* sub-algorithms respectively.

```
function RHBSPCG(B, C)
 s = SIZE(B)

 for (m = 0; m < s; ++m)
  DOCG(B[m,m])
  BMT = TRANSPOSE(B[m,m])
  for (i = 0; i < s && i != m; ++i)
   if ISVHS(C[m]) then
    C1CGEX(B[i,m], B[m,m], BMT, C[m])
    C2CGEX(B[m,i], B[m,m], BMT, C[m])
   else
    if INPUT(C[m]) > OUTPUT(C[m]) then
     C1CGEX(B[i,m], B[m,m], BMT, C[m])
     C2CG(B[m,i], B[m,m], C[m])
    else
     C1CG(B[i,m], B[m,m], C[m])
     C2CGEX(B[m,i], B[m,m], BMT, C[m])

 for (i = 0; i < s && i != m; ++i)
  for (j = 0; j < s && j != m; ++j)
   P3CG(B[i,j], B[i,m], B[m,j])
```

Fig. 5. Pseudocode of *RHBSPCG* algorithm. *B* is the matrix of blocks. *C* describes clusters. Function *SIZE* returns the matrix size, function *ISVHC* returns true if the cluster is VHC, functions *INPUT* and *OUTPUT* return lists of input and output bridges respectively, function *TRANSPOSE* creates a transposed block

Experiments. The experiments were conducted on the rack server equipped with 2 Intel Xeon E5-2620 v4 CPUs and 32 GB of RAM. The experimental algorithms were written in the C++ programming language and compiled using GCC v14.2.0 compiler with O3 optimisation level. The parallelization was done using OpenMP v4.5.

We performed two sets of experiments with *HBSPCG* and *RHBSPCG* algorithms.

First set of experiments. These experiments were conducted on three specially crafter random graphs of 9600 vertices – *XG1* graph with all clusters having large number of V_{OUT} and small

number of V_{IN}, *XG2* with all clusters having large number of V_{IN} and small number of V_{OUT} and *XG3* graph with all clusters having large number of $V_{IN\text{-}OUT}$ (Table 2).

Table 2. **Detailed information about clusters and bridge vertices of *XG1*, *XG2* and *XG3* experimental graphs**

Graph		*XG1*		*XG2*		*XG3*	
Cluster	Vertex Count	V_{IN}	V_{OUT}	V_{IN}	V_{OUT}	V_{IN}	V_{OUT}
0	1841	162	1266	1243	152	1226	1226
1	1871	164	1248	1246	164	1226	1226
2	2102	155	1245	1257	160	1226	1226
3	2084	165	1245	1249	167	1226	1226
4	1702	152	1244	1257	159	1226	1226

In these experiments, we measured the execution time of all sub-algorithms, and the total execution time of the *HBSPCG* and *RHBSPCG* algorithms (Table 3).

Table 3. **Results of the "first series" of experiments running *RHBSPCG* and *HBSPCG* on three experimental graphs *XG2*, *XG1*, and *XG3*. Time is in milliseconds. The diagonal block execution time includes the cost of creating a BMT block transposed from a diagonal block**

All time values are in milliseconds (ms)		*XG2*		*XG1*		*XG3*	
		RHBSPCG	*HBSPCG*	*RHBSPCG*	*HBSPCG*	*RHBSPCG*	*HBSPCG*
Algorithm	Total Time	37418	39378	34844	39278	118803	127096
Peripheral Block	Average Time	128	134	128	133	980	1014
Horizontal Block	Average Time	128	133	847	1024	835	1012
Vertical Block	Average Time	974	1026	128	133	927	1008
Diagonal Block	Average Time	1011	1089	1013	1087	1016	1093

On the *XG1* graph, *RHBSPCG* outperformed *HBSPCG* by 12.73% because of the faster processing of the horizontal blocks with the *C2CGEX* sub-algorithm, which outperformed *C2CG* by 20.98%, and the faster processing of the diagonal, vertical and peripheral blocks (by 7.24%, 3.75% and 3.91% respectively).

On the *XG2* graph, *RHBSPCG* outperformed *HBSPCG* by 5.24% because of the faster processing of the vertical blocks with the *C1CGEX* sub-algorithm, which outperformed *C1CG* by 5.36%, and the faster processing of diagonal, horizontal and peripheral blocks (by 7.78%, 3.91% and 4.38% respectively).

On the *XG3* graph *RHBSPCG* outperformed *HBSPCG* by 6.98% because of the faster processing of the horizontal and vertical blocks with he *C1CGEX* and *C2CGEX* sub-algorithms which outperformed *C1CG* and *C2CG* by 8.69% and 21.25%, and because of the faster processing of diagonal and peripheral blocks (by 7.60% and 3.51% respectively).

The reason why on the *XG3* graph the *RHBSPCG* demonstrated smaller speedup compared to *XG1* (6.98% on *XG3* compared to 12.73% on *XG1*) is because of the clusters configuration – in *XG3* all clusters have only $V_{IN\text{-}OUT}$ vertices, which means, all peripheral blocks were calculated from the $V_{IN\text{-}OUT}$ vertices (contrary to *XG1* where all peripheral blocks were recaluclated through smallest vertical or horizontal vertices), which lead to a longer total execution time and therefore to a smaller speedup.

The speedup in the calculation of the diagonal, vertical, horizontal (when calculated by *C1CG* and *C2CG* sub-algorithms), and peripheral blocks in the *RHBSPCG* algorithm is caused by the matrix transformation which aligned all bridge vertices at the end of the clusters. This alignment enables better spatial locality and therefore better utilization of CPU caches.

Second set of experiments. These experiments were conducted on four random graphs of 9600 vertices where each graph has a different number of clusters, from five to twenty. All graphs have V_{IN}, V_{OUT} and $V_{IN\text{-}OUT}$ bridge vertices in random proportions for every cluster (Table 4).

Table 4. **Clusters and bridge vertices of 9600-10, 9600-15 and 9600-20 experimental graphs**

Graph	Cluster Count	V_{BRIDGE}	V_{IN}	V_{OUT}	$V_{IN\text{-}OUT}$
9600-5	5	5940	3038	4071	1169
9600-10	10	5772	3381	3193	802
9600-15	15	5984	3819	3151	986
9600-20	20	5340	2137	3618	415

In these experiments, we measured total execution time of *HBSPCG* and *RHBSPCG* algorithms in single-threaded and multi-threaded variants (Table 5). In the single-threaded variant the *RHBSPCG* outperformed *HBSPCG* by 17.05% on 9600-5, by 23.45% on 9600-10, by 29.37% on 9600-15 and by 29.62% on 9600-20. In the multi-threaded variant, the *RHBSPCG* outperformed *HBSPCG* by 14.72%, 5.40%, 6.93% and 5.03% respectively.

Table 5. **Results of the "second series" of experiments of running *RHBSPCG* and *HBSPCG* algorithms in single-threaded and multi-threaded variants on three experimental graphs 9600-5, 9600-10, 9600-15 and 9600-20. Time is in milliseconds**

Graph	*HBSPCG*	*RHBSPCG*	*HBSPCG* (OpenMP)	*RHBSPCG* (OpenMP)
9600-5	58702	50152	22586	19688
9600-10	59224	47975	6045	5735
9600-15	70762	54700	4612	4313
9600-20	44686	34475	3208	3055

In all experiments. *RHBSPCG* outperformed *HBSPCG* from 17.05% to 29.62% in single threaded variant and from 14.72% to 5.03% in multi-threaded variant.

On average, the achieved speedup was 27.48% for the single-threaded implementations and 5.79% for the multi-threaded implementations.

Conclusion. We introduced new versions *C1CGEX* and *C2CGEX* of sub-algorithms for the fast computation of horizontal and vertical blocks of the shortest path distance matrix when solving the all-pairs shortest paths problem. We proposed a new reconfigurable blocked *RHBSPCG* algorithm for solving the problem. The algorithm uses the newly introduced *C1CGEX* and *C2CGEX* sub-algorithms together with the input matrix transformation (which aligns bridge vertices at the end of the cluster vertex list) and uses the dynamic selection of the sub-algorithms for calculation of vertical and horizontal blocks based on the cluster reconfiguration. These advancements allowed the *RHBSPCG* algorithm to demonstrate an average speed up of 27.48% over the *HBSPCG* algorithm in a single-threaded variant, and an average speed up of 5.79% for the same experimental graphs in multi-threaded variant.

References

1. Luzzi M., Guerriero F., Maratea M., Greco G., Garafalo M. Chatgpt and operations research: evaluation on the shortest path problem. *Soft Computing*, 2025, vol. 1, no. 3, pp. 1407–1418.

2. Ehrmanntraut V., Meyer U. Going Faster: Privacy-Preserving Shortest paths from Start to End. *Cryptology ePrint Archive: 2025/1794*, 2025.

3. Duan R., Mao J., Mao X., Shu X., Yin L. Breaking the Sorting Barrier for Directed Single-Source Shortest Paths. *arXiv:2504.17033*, 2025.

4. D'Emidio, M., Di Stefano G. On Computing Top-k Simple Shortest Paths from a Single Source. *arXiv: 2509.26094*, 2025.

5. Cohen A., Gromov A., Yang K., Tian Y. Spectral Journey: How Transformers Predict the Shortest Path. *arXiv: 2502.08794*, 2025.

6. Dijkstra E. W. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1959, vol. 1, no. 1, pp. 269–271.

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

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

9. Prihozhy A. A., Karasik O. N. Heterogeneous blocked all-pairs shortest paths algorithm. *Sistemnyy analiz i prikladnaya informatika* [System analysis and Applied Information Science], 2017, no. 3, pp. 68–75. Available at: https://doi.org/10.21122/2309-4923-2017-3-6-75 (accessed 29.01.2023) (In Russian).

10. Prihozhy A. A., Karasik O. N. Advanced heterogeneous block-parallel all-pairs shortest path algorithm. *Trudy BGTU* [Proceedings of BSTU], issue 3, Physics and Mathematics. Informatics, 2023, no. 1 (266), pp. 77–83.

11. 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.

12. 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.

13. Prihozhy A. A., Karasik O. N. Heterogeneous blocked all-pairs shortest paths algorithm for clustered weighted graphs. *Journal of the Belarusian State University. Mathematics and Informatics*, 2025, no. 3, pp. 61–74.

14. Prihozhy A. A., Karasik O. N. Localization of data references in blocked heterogeneous shortest paths algorithm for clustered graphs. *Trudy BGTU* [Proceeding of BSTU], issue 3, Physics and Mathematics. Informatics, 2025, no 2 (296), pp. 83–90.

Список литературы

1. Luzzi M., Guerriero F., Maratea M., Greco G., Garafalo M. Chatgpt and operations research: evaluation on the shortest path problem // Soft Computing. 2025. Vol. 29, no 3. P. 1407–1418.

2. Ehrmanntraut V., Meyer U. Going Faster: Privacy-Preserving Shortest paths from Start to End // Cryptology ePrint Archive: 2025/1794. 2025.

3. Duan R., Mao J., Mao X., Shu X., Yin L. Breaking the Sorting Barrier for Directed Single-Source Shortest Paths // arXiv:2504.17033. 2025.

4. D'Emidio M., Di Stefano G. On Computing Top-k Simple Shortest Paths from a Single Source // arXiv:2509.26094. 2025.

5. Cohen A., Gromov A., Yang K., Tian Y. Spectral Journey: How Transformers Predict the Shortest Path // arXiv:2502.08794. 2025.

6. Dijkstra E. W. A note on two problems in connexion with graphs // Numerische Mathematik. 1959. Vol. 1, no. 1. P. 269–271.

7. Floyd R. W. Algorithm 97: Shortest path // Communications of the ACM. 1962. No. 5 (6). P. 345.

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

9. Прихожий А. А., Карасик О. Н. Разнородный блочный алгоритм поиска кратчайших путей между всеми парами вершин графа // Системный анализ и прикладная информатика. 2017. № 3. С. 6–75. URL: https://doi.org/10.21122/2309-4923-2017-3-6-75 (дата обращения: 29.01.2023).

10. Prihozhy A. A., Karasik O. N. Advanced heterogeneous block-parallel all-pairs shortest path algorithm // Труды БГТУ. Сер. 3, Физико-математические науки и информатика. 2023. № 1 (266). С. 77–83.

11. 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. P. 4–13.

12. 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. P. 4–10.

13. Prihozhy A. A., Karasik O. N. Heterogeneous blocked all-pairs shortest paths algorithm for clustered weighted graphs // Journal of the Belarusian State University. Mathematics and Informatics. 2025. No. 3. P. 61–74.

14. Prihozhy A. A., Karasik O. N. Localization of data references in blocked heterogeneous shortest paths algorithm for clustered graphs // Труды БГТУ. Сер. 3, Физико-математические науки и информатика. 2025. № 2 (296). С. 83–90.

Information about the authors

Prihozhy Anatoly Alexievich − DSc (Engineering), Professor, Professor, the Department of Computer and System Software. Belarusian National Technical University (65 Nezavisimosti Ave., 220013, Minsk, Republic of Belarus). E-mail: prihozhy@bntu.by. ORCID: 0000-0002-1941-0806.

Karasik Oleg Nikolaevich − PhD (Engineering), Tech Lead. ISsoft Solutions (5 Chapaeva str., 220034, Minsk, Republic of Belarus). E-mail: karasik.oleg.nikolaevich@gmail.com. ORCID: 0009-0000-9670-4181.

Информация об авторах

Прихожий Анатолий Алексеевич − доктор технических наук, профессор, профессор кафедры программного обеспечения информационных систем и технологий. Белорусский национальный технический университет (пр-т Независимости, 65, 220013, г. Минск, Республика Беларусь). E-mail: prihozhy@bntu.by. ORCID: 0000-0002-1941-0806.

Карасик Олег Николаевич − кандидат технических наук, ведущий инженер. Иностранное производственное унитарное предприятие «Иссофт Солюшенз» (ул. Чапаева, 5, 220034, г. Минск, Республика Беларусь). E-mail: karasik.oleg.nikolaevich@gmail.com. ORCID: 0009-0000-9670-4181.

Received 17.12.2025