UDC 004.4

SPEEDUP OF BLOCKS CALCULATION IN BLOCKED FLOYD-WARSHALL ALGORITHM

Prihozhy A. A.
Belarusian National Technical University,
Minsk, Belarus, prihozhy@yahoo.com

Finding shortest and longest paths in graphs [1–8] solves optimization problems in many application domains. Based on the classical Floyd-Warshall algorithm (*FW*) [9], the blocked Floyd-Warshall Algorithm 1 (*BFW*) was developed in works [10–19] by means of decomposing the matrix $D[N \times N]$ of shortest path distances in a weighted graph into the matrix $B[M \times M]$ of blocks $B_{v,u}[S \times S]$, where N is the number of graph vertices, S is the block size, $M = N / S$, and $v, u = 0 \dots M{-}1$. In *BFW*, all blocks are calculated by the universal Algorithm 2 (*BCA*), which is in fact *FW* with three input blocks B^1, B^2 and $B^3$and one output block B^1. In *BFW*, there are four calls of *BCA* with different arguments. In the *D0* call, all three arguments are copies of the same block $B_{m,m}$. In the *C1* call, two arguments are copies of block $B_{v,m}$, and one argument is block $B_{m,m}$. In the *C2* call, two arguments are copies of block $B_{m,v}$, and one argument is block $B_{m,m}$. In the *P3* call, all arguments are unique blocks $B_{v,u}$, $B_{v,m}$ and $B_{m,u}$. In the calls, *BCA* consumes different input data of different overall size. Therefore. we develop a unique algorithm for each call, which replaces *BCA* during the *BFW* execution. We refer to such a modification of *BFW* as a heterogeneous blocked shortest paths algorithm, briefly *HET*.

Tab. 1 describes storage consumption per iteration of the loop along k, and the overall storage consumption over all loop iterations. Here we assume that all elements of block B^1 are processed within each iteration, block B^2 is accessed within each iteration row by row, and block B^3 is accessed within each iteration column by column. *BCA* consumes the amount S^3 of storage while calculating the *D0* and consumes the amount $S^3 + 2S^2$ of storage overalls while calculating the *P3* block.

Algorithm 1: Blocked Floyd–Warshall (*BFW*)

```
Input: A number $N$ of graph vertices
Input: A matrix $W[N \times N]$ of graph edge weights
Input: A size $S$ of block
Output: A blocked matrix $B[M \times M]$ of path distances
  $B \leftarrow W M \leftarrow N / S$
  for $m \leftarrow 0$ to $M-1$ do
    $B_{m,m}^{m+1} \leftarrow BCA(B_{m,m}, B_{m,m}, B_{m,m})$          // D0
    for $v \leftarrow 0$ to $M-1$ do
      if $v \neq m$ then
        $B_{v,m}^{m+1} \leftarrow BCA(B_{v,m}, B_{v,m}, B_{m,m})$      // C1
        $B_{m,v}^{m+1} \leftarrow BCA(B_{m,v}, B_{m,m}, B_{m,v})$      // C2
    for $v \leftarrow 0$ to $M-1$ do
```

```
        if v ≠ m then
          for u ← 0 to M–1 do
            if u ≠ m then
              $B_{v,u} \leftarrow BCA(B_{v,u}, B_{v,m}, B_{m,u})$        // P3
    return B
```

Algorithm 2: Block calculation (*BCA*)

```
Input: S is size of block
Input: $B^1$, $B^2$, $B^3$ are input blocks
Output: $B^1$ is recalculated block
  for k ← 0 to S - 1 do
      for i ← 0 to S - 1 do
          for j ← 0 to S - 1 do
              $sum \leftarrow b^2_{i,k} + b^3_{k,j}$
              if $b^1_{i,j} > sum$ then $b^1_{i,j} \leftarrow sum$;
  return $B^1$
```

Table 1 – Storage consumption by four types of blocks

Block type	Per iteration of loop along *k*			Over all iterations
	Input B^1	Input B^2	Input B^3	
D0	S^2	–	–	S^3
C1	S^2	–	S	$S^3 + S^2$
C2	S^2	S	–	$S^3 + S^2$
P3	S^2	S	S	$S^3 + 2S^2$

Calculating diagonal block. Our new algorithm *D0_A* calculates block B^1 in stepwise manner while adding vertices to a graph and adding row k and column k to matrix B^1. Fig. 1 illustrates the transition from matrix $B^1(k – 1)$ to matrix $B^1(k)$ in a loop along k. First, element b^1_{ik} is calculated over b^1_{ij} and ${}^1b_{jk}$ and element b^1_{kj} is calculated over b^1_{ki} and b^1_{ij} for $i, j = 0…k– 1$. Second, element b^1_{ij} is recalculated over b^1_{ik} and b^1_{kj} for $i, j = 0…k – 1$. Algorithm 3 completely describes *D0_A*.

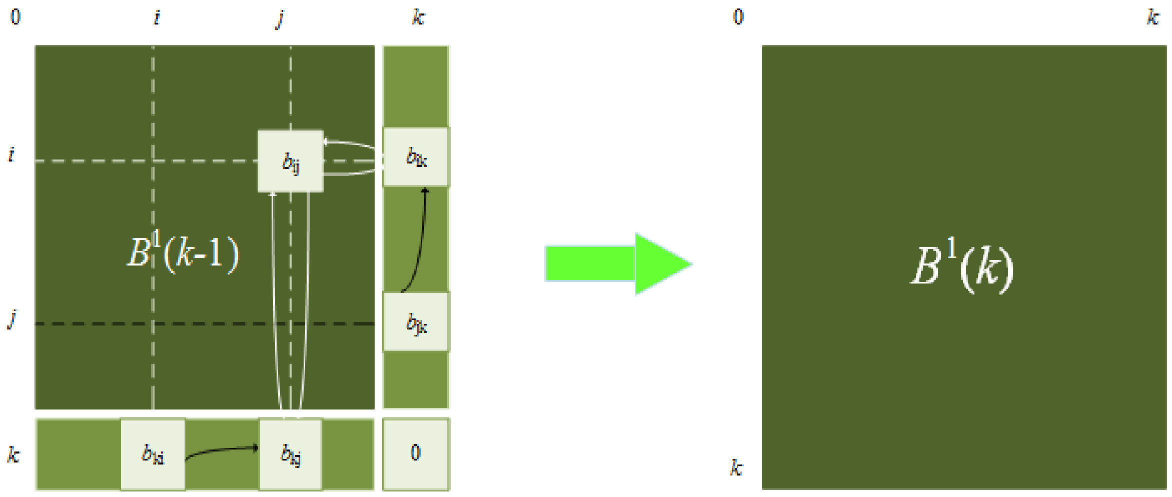

Figure 1 – Calculating diagonal block $B^1(k)$ over $B^1(k – 1)$ – algorithm *D0_A*

Algorithm 3: Diagonal block calculation algorithm (*D0_A*)

```
Input: A block $B^1$
Input: A size $S$ of block
Output: A recalculated block $B^1$
 for $k \leftarrow 0$ to $S - 1$ do
     for $i \leftarrow 0$ to $k - 1$ do
         for $j \leftarrow 0$ to $k - 1$ do
             $s_0 \leftarrow b^1_{ij} + b^1_{jk}$ if $b^1_{ik} > s_0$ then $b^1_{ik} \leftarrow s_0$
             $s_1 \leftarrow b^1_{ki} + b^1_{ij}$ if $b^1_{kj} > s_1$ then $b^1_{kj} \leftarrow s_1$
     for $i \leftarrow 0$ to $k - 1$ do
         for $j \leftarrow 0$ to $k - 1$ do
             $s_2 \leftarrow b^1_{ik} + b^1_{kj}$ if $b^1_{ij} > s_2$ then $b^1_{ij} \leftarrow s_2$
 return $B^1$
```

Calculating vertical block of cross. Our new algorithm *C1_A* calculates block B^1 over block B^3 in stepwise manner while adding a vertex to a graph and adding a column k to matrix B^1. Fig. 2 illustrates the transition from matrix $B^1(k - 1)$ to matrix $B^1(k)$ in a loop along k. First, element b^1_{ik} of B^1 is calculated over b^1_{ij} of B^1 and b^3_{jk} of B^3 for $i = 0 \ldots S - 1$ and $j = 0 \ldots k - 1$. Second, element b^1_{ij} of B^1 is recalculated over b^1_{ik} of B^1 and b^3_{kj} of B^3 for the same ranges of indices. Algorithm 4 completely describes *C1_A*.

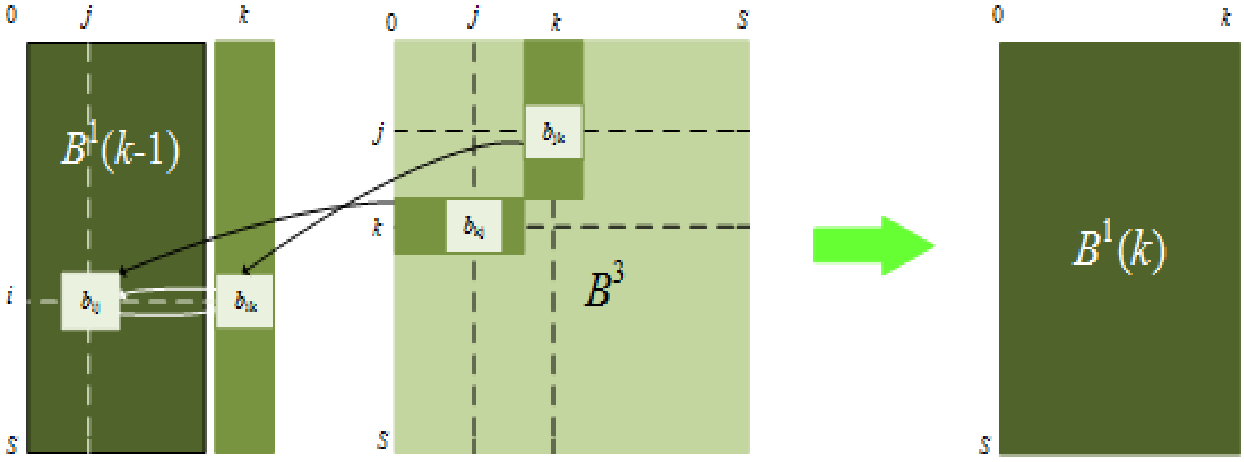

Figure 2 – Calculating vertical block $B^1(k)$ of cross over $B^1(k\text{-}1)$ – algorithm *C1_A*

Algorithm 4: Calculating vertical block of cross (*C1_A*)

```
Input: Blocks $B^1$ and $B^2$
Input: A size $S$ of block
Output: A recalculated block $B^1$
  for $k \leftarrow 1$ to $S - 1$ do
      for $i \leftarrow 0$ to $S - 1$ do
          for $j \leftarrow 0$ to $k - 1$ do
              $s_0 \leftarrow b^1_{ij} + b^3_{jk}$ if $b^1_{ik} > s_0$ then $b^1_{ik} \leftarrow s_0$
      for $i \leftarrow 0$ to $S - 1$ do
          for $j \leftarrow 0$ to $k - 1$ do
              $s_2 \leftarrow b^1_{ik} + b^3_{kj}$ if $b^1_{ij} > s_2$ then $b^1_{ij} \leftarrow s_2$
  return $B^1$
```

Calculating horizontal block of cross. The new algorithm *C2_A* calculates block B^1 over block B^2 in stepwise manner while adding a vertex to a graph and adding a row k to matrix B^1. Fig. 3 illustrates the transition from matrix $B^1(k – 1)$ to matrix $B^1(k)$ in a loop along k. First, element b^1_{kj} of B^1 is calculated over b^2_{ki} of B^2 and b^1_{ij} of B^1 for $i = 0…k – 1$ and $j = 0…S – 1$. Second, element b^1_{ij} of B^1 is recalculated over b^2_{ik} of B^2 and b^1_{kj} of B^1 for the same ranges of indices. Algorithm 5 completely describes *C2_A*.

The algorithms *D0_A*, *C1_A* and *C2_A* are further improved by means of resynchronizing the loops along *i* and *j*, merging the loops, and introducing the sequential reference locality for blocked data due to collecting column elements in one-dimensional arrays. Algorithms *D0_A, C1_A* and *C2_A* have advantages against the BCA algorithm. They reduce the number of loop iterations in nested loops and exploit the hierarchical caches efficiently.

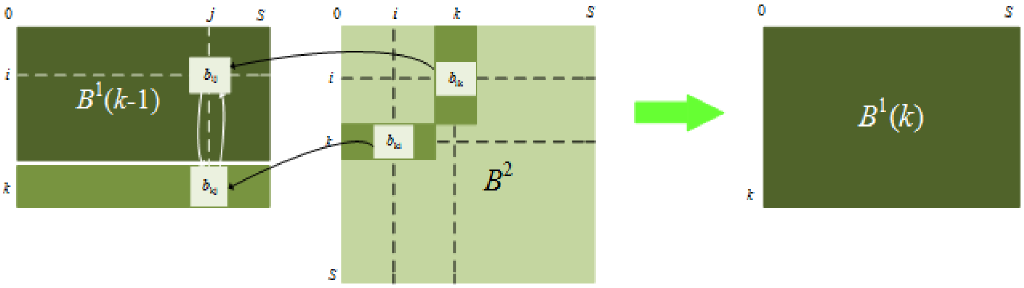

Figure 3 – Calculating horizontal block $B^1(k)$ of cross over $B^1(k – 1)$ – algorithm *C2_A*

*Results.*The experiments were carried out on a multi-core processor Intel(R) Core(TM) i5-6200UCPU @ 2.20 GHz. They aimed for identifying the dependence of the run-time of the *FW, BFW* and *HET* algorithms and the algorithms for calculating four types of blocks depending on the graph size, block size and number of blocks. They make it possible to compare the new *HET* algorithm with the known homogeneous blocked *BFW* algorithm. We used complete graphs with random weights on the edges, for which the problem of shortest paths is the hardest. Tab. 2 reports the run-time of the uniform *BCA* algorithm that is used by *BFW* for all types of blocks on matrix *B*[2×2] of various graph-sizes and various block-sizes. The run-time is close for all blocks of *D0*, *C1*, *C2* and *P3* types.

Algorithm 5: Calculating horizontal block of cross (*C2_A*)

```
Input: Blocks $B^1$ and $B^2$
Input: A size $S$ of block
Output: A recalculated block $B^1$
  for $k \leftarrow 1$ to $S - 1$ do
      for $i \leftarrow 0$ to $k - 1$ do
          for $j \leftarrow 0$ to $S - 1$ do
              $s_0 \leftarrow b^2_{ki} + b^1_{ij}$ if $b^1_{kj} > s_0$ then $b^1_{kj} \leftarrow s_0$
      for $i \leftarrow 0$ to $k - 1$ do
          for $j \leftarrow 0$ to $S - 1$ do
              $s_2 \leftarrow b^2_{ik} + b^1_{kj}$ if $b^1_{ij} > s_2$ then $b^1_{ij} \leftarrow s_2$
  return $B^1$
```

When we apply the *D0_A*, *C1_A*, *C2_A* and *P3_A* algorithms to the same graphs and blocks, their run-time is different (tab. 3). The fastest algorithm is *D0_A*, which yields the speedup of 33.94 % on average over *BCA* (fig. 4). Algorithms *C1_A* and *C2_A* shows the speedup of 24.59 % and 25.26 % respectively. The slowest *P3_A* algorithm has shown the speedup of only 2.72 %. We can explain this fact as the graph extension-based technique has failed to be applied to the blocks of type *P3*. Fig.e 5 gives a pair-wise comparison of the *FW*, *BFW* and *HET* algorithms on graphs of 2400 vertices depending on the block-size. Algorithm *BFW* is faster than *FW* by 5.06 % on average. The gain of the *Het* algorithm is from 9.28 % to 25.64 % over *FW* and is from 3.57 % to 23.40 % over *BFW*. Thus, the new *D0_A*, *C1_A*, *C2_A* and *P3_A* algorithms of block calculation have significantly contributed to the acceleration of the shortest paths search.

Table 2 – Run-time (*ms*) of uniform algorithm *BCA* on all block types of blocked matrix *B*[2×2] vs. vertex count *N* and block size *S*

N	*S*	Mean	Min	Max	Min %	Max %
480	240	45.8	43.5	48.5	4.92	6.01
720	360	142.6	140.5	144.5	1.49	1.31
960	480	335.1	332.0	339.5	0.93	1.31
1200	600	657.1	653.5	661.0	0.55	0.59
1440	720	1123.0	1115.0	1130.5	0.71	0.67
1680	840	1804.8	1782.0	1829.0	1.26	1.34
1920	960	2705.9	2684.0	2721.5	0.81	0.58
2160	1080	3865.4	3851.5	3893.5	0.36	0.73
2400	1200	5287.9	5236.0	5334.0	0.98	0.87

Table 3 – Run-time (*ms*) of algorithms *D0_A*, *C1_A*, *C2_A* and *P3_A* on blocked matrix *B*[2×2] vs. vertex count *N* and block size *S*

N	*S*	*D0_A*	*C1_A*	*C2_A*	*P3_A*
480	240	29.0	33.0	32.5	43.0
720	360	96.0	115.0	108.5	141.0
960	480	227.0	253.0	251.0	330.0
1200	600	438.5	492.0	492.5	647.5
1440	720	751.0	846.0	848.5	1127.5
1680	840	1195.5	1345.0	1329.0	1764.5
1920	960	1838.0	2020.5	1993.0	2644.0
2160	1080	2527.5	2885.5	2837.5	3748.5
2400	1200	3472.5	3958.0	3898.5	5164.0

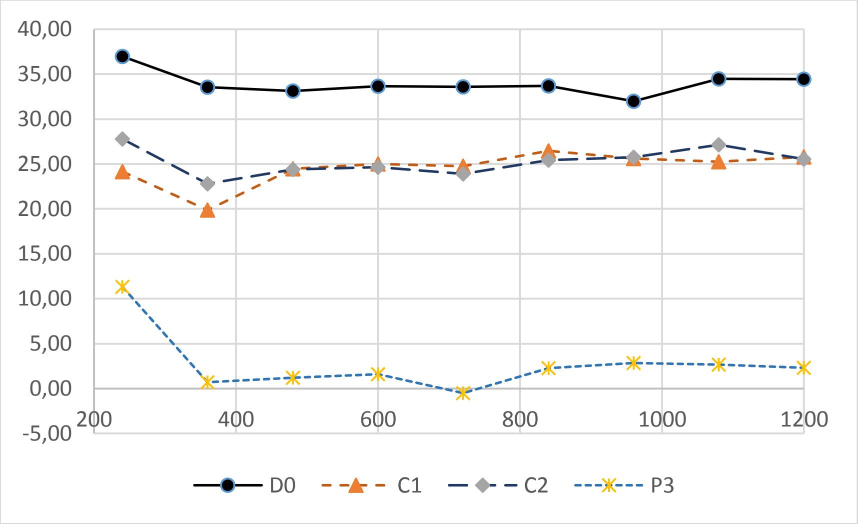

Figure 4 – Speedup % of algorithms *D0_A*, *C1_A*, *C2_A* and *P3_A* over *BCA* for *B*[2×2] vs. block-size 240…1200 in graphs of 480…2400 vertices

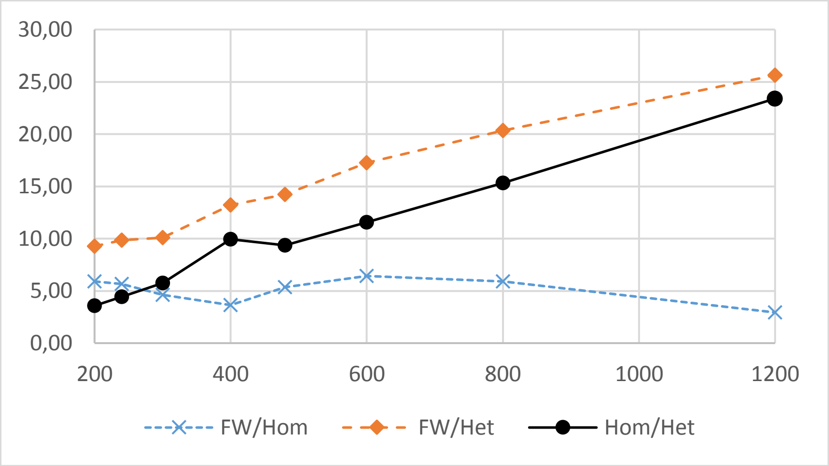

Figure 5 – Speedup (%) of *BFW* over *FW*, *HET* over *FW* and *HET* over *BFW* on graphs of 2400 vertices vs. block-size

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

3. PrihozhyA.,Bezati E., Ab RahmanA.-H.,Mattavelli M. Synthesis and Optimization of Pipelines for HW Implementations of Dataflow Programs, IEEE Transactions on CAD, 2015, vol. 34, no. 10, pp. 1613–1626.

4. Прихожий А. А. Распределенная и параллельная обработка данных. – Минск: БНТУ, 2016. – 90 с.

5. Prihozhy A. A., Mattavelli M., Mlynek D. Data dependences critical path evaluation at C/C++ system level description. International Workshop PATMOS'2003, Springer, 2003, pp. 569–579.

6. Прихожий А. А., Ждановский А. М., Карасик О. Н., Маттавелли М. Эвристический генетический алгоритм оптимизации вычислительных конвейеров. Доклады БГУИР, 2017, № 1, с. 34–41.

7. Prihozhy A. ., Casale-Brunet S., Bezati E., Mattavelli M. Efficient Dynamic Optimisation Heuristics for Dataflow Pipelines. 2018 IEEE International Workshop on Signal Processing Systems (SiPS), 2018, pp. 1–6.

8. Prihozhy A. A., Casale-Brunet S., Bezati E., Mattavelli M. Pipeline Synthesis and Optimization from Branched Feedback Dataflow Programs. Journal of Signal Processing Systems [Electronic resource], 2020, vol. 92, pp. 1091–1099. – Mode of access: https://doi.org/10.1007/s11265-020-01568-5. – Date of access: 20.09.2022.

9. Floyd R. W. Algorithm 97: Shortest Path. Communications of the ACM, 1962, vol. 5, no. 6, p. 345.

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

11. Park J., Penner M., Prasanna V. K. Optimizing graph algorithms for improved cache performance. IEEE Transactions on Parallel and Distributed Systems, 2004, vol. 15, no. 9. pp. 769–782.

12. Albalwi E., Thulasiraman P., Thulasiram R. Task Level Parallelization of All Pair Shortest Path Algorithm in OpenMP 3.0. Advances in Computer Science and Engineering (CSE 2013), Los Angeles: Atlantis Press, 2013, pp. 109–112.

13. Tang P. Rapid development of parallel blocked all-pairs shortest paths code for multi-core computers. IEEE SOUTHEASTCON 2014, Lexington, KY, USA, IEEE, 2014, pp. 1–7.

14. 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. 68–75.

15. 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.

16. Karasik O. N.,Prihozhy A. A. Threaded block-parallel algorithm for finding the shortest pats on graph. Doklady BGUIR, 2018, No. 2, pp. 77–84.

17. Prihozhy A. A.,Karasik O. N. Cooperative block-parallel algorithms for task execution on multi-core system. System analysis and applied information science, 2015, no. 2, pp. 10–18.

18. Prihozhy A. A. Simulation of direct mapped, k-way and fully associative cacheonall pairs shortest paths algorithms. System analysis and applied information science, 2019, no. 4, pp. 10–18.

19. 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 InternationalConference.Minsk: Bestprint, 2022, pp. 56–66.