UDC 004.4

REDUCTIONOFPROCESSOR TIME AND ENERGY CONSUMPTION BY PROFILING ALGORITHMS OF FINDING SHORTEST PATHS IN A GRAPH

[1]Prihozhy A. A., [2]Subbota Y. M.
[1]Belarusian National Technical University,
Minsk, Belarus, prihozhy@yahoo.com,
[2]Belarusian National Technical University,
Minsk, Belarus, subbota@mail.com

The paper solves the problem of improving parameters of software implementations on multicore systems and uses the problem of finding shortest (critical) paths between all pairs of vertices of a weighted graph [1–6] as a benchmark. For this purpose, we selected a set of competing algorithms of shortest paths search, and our objective was to choose the best algorithm with respect to time and power parameters by means of profiling programs during their execution on a multicore system. Among the competing algorithms we considered: the classical Floyd-Warshall algorithm [7, 8]; the blocked Floyd-Warshall algorithm [9–16]; the algorithm based on stepwise addition of vertices to the graph [17]. Likwid [18, 19] was chosen as a tool for profiling and measuring program parameters.

Algorithm 1 is the classical Floyd-Warshall (*FW*) algorithm proposed in [7] and built on the matrix *D* of shortest paths distances by means of three nested loops that perform the same kind of calculations on all elements of the matrix. In the body of the most nested loop, the length *sum* of path between vertices *i* and *j* of the graph which passes through vertex *k*, which can update element $d_{i,j}$ of the matrix, is calculated. Fig. 1 illustrates the *FW* operation.

Algorithm 1: Floyd-Warshall (*FW*)

```
Input: A matrix $W$ of graph edge weights
Input: A size $N$ of matrix
Output: A matrix $D$ of path distances
  $D \leftarrow W$
  for $k \leftarrow 1$ to $N$ do
      for $i \leftarrow 1$ to $N$ do
          for $j \leftarrow 1$ to $N$ do
              $sum \leftarrow d_{i,k} + d_{k,j}$
              if $d_{i,j} > sum$ then $d_{i,j} \leftarrow sum$
  return $D$
```

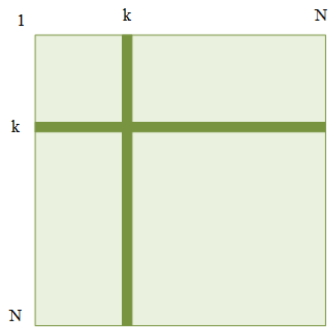


Figure 1 – Illustration of classic Floyd-Warshall algorithm operation

Algorithm 2 is the blocked Floyd-Warshall algorithm (*BFW*) proposed in [9, 10] and constructed by decomposing matrix *D* into matrix *B* of blocks. Algorithm *BCA*($B_{i,j}$,$B_{i,m}$, $B_{m,j}$) recalculates block $B_{i,j}$ through blocks $B_{i,m}$ and $B_{m,j}$. Fig. 2 illustrates the operation of the *BFW*. Using BCA, *BFW* computes the diagonal block *D0*, then the blocks *C1* and *C2* of the cross, and finally the peripheral blocks *P3*. At each iteration, the cross shifts from the upper left to the lower right corner. Partitioning of the *D* matrix into blocks creates spatial locality of references to the elements of the blocks, which increases the efficiency of the hierarchical cache memory operation.

Algorithm 3 [13, 17] is based on stepwise addition of vertices to the graph (GEA). At each iteration it adds row k and column k to matrix *D*, calculates the lengths of shortest paths entering and leaving from vertex *k*, and recalculates path lengths between pairs of vertices from set {1, ..., *k*– 1}. Fig. 3 illustrates the GEA operation. The main advantage of GEA is the increased temporal locality of data processing since the algorithm works with submatrices of monotonically increasing size in the range from 1 × 1 to N × N.

Algorithm 2: Blocked Floyd–Warshall (*BFW*)

```
Input: A number $N$ of graph vertices
Input: A matrix $W$ of graph edge weights
Input: A size $S$ of block
Output: A blocked matrix $B$ of path distances
  $M \leftarrow N / S$     $B[M\times M] \leftarrow W[N\times N]$
  for $m \leftarrow 1$ to $M$ do
      $BCA$ ($B_{m,m}$, $B_{m,m}$, $B_{m,m}$)                // D0
      for $i \leftarrow 1$ to $M$ do
          if $i \neq m$ then
              $BCA$ ($B_{i,m}$, $B_{i,m}$, $B_{m,m}$)        // C1
              $BCA$ ($B_{m,i}$, $B_{m,m}$, $B_{m,i}$)        // C2
      for $i \leftarrow 1$ to $M$ do
          if $i \neq m$ then
              for $j \leftarrow 1$ to $M$ do
                  if $j \neq m$ then
                      $BCA$ ($B_{i,j}$, $B_{i,m}$, $B_{m,j}$)   // P3
  return $B$
```

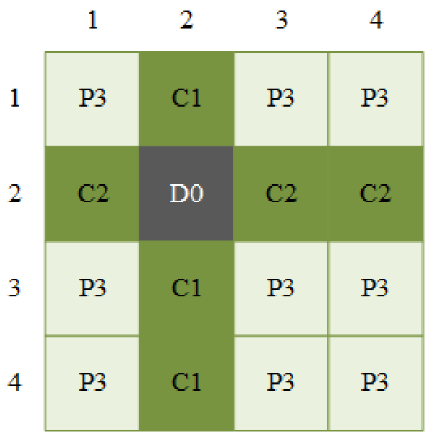


Figure 2 – Illustration of blocked Floyd-Warshall algorithm operation

LIKWID [14] is part of the OpenHPC (High Performance Computing) suite, a reference collection of open-source HPC software components and best practices. Moreover, LIKWID and its team is part of a virtual institute of about 15 academic and industry partners to develop state-of-the-art tools for high-performance computing. There are also maintained packages for ArchLinux and Gentoo. Spack, a package manager for supercomputers, contains LIKWID as a mainline package.

The Likwid software tool allows evaluating the characteristics of software implementations of algorithms, comparing them and giving recommendations on the choice of the preferable algorithm according to a selected criterion. The command utility likwid-perfctr [14] is designed for profiling program code with low overhead and without interfering with the process and results of code execution. Figure 4 shows the relationship between counters, processor events, and Likwid characteristics. Markers are used to highlight the profiled areas of the program code [24]. Nesting or partial overlapping of marked areas is not allowed. At first Likwid is integrated into software implementations of algorithms, then computational experiments are performed on a multicore system, after that the results of parameter measurements are systematized and conclusions are made about algorithm preferences.

Algorithm 3: *GEA* after improving spatial locality

```
Input: A matrix $W$ of graph edge weights
Input: A size $N$ of matrix
Output: A matrix $D$ of shortest path distances
  $D \leftarrow W$  $c1_1 \leftarrow \infty$  $w_1 \leftarrow d_{1,2}$
  for $k \leftarrow 2$ to $N$ do
      $k1 \leftarrow k - 1$    $r \leftarrow getRow(D, k)$    $r1 \leftarrow getRow(D, k1)$
      for $i \leftarrow 1$ to $k1$ do
          $min \leftarrow \infty$    $ri \leftarrow getRow(D, i)$
          for $j \leftarrow 1$ to $k1$ do
              $s_2 \leftarrow c1_i + r1_j$ if $ri_j > s_2$ then $ri_j \leftarrow s_2$
              $s_0 \leftarrow ri_j + w_j$ if $min > s_0$ then $min \leftarrow s_0$
              $s_1 \leftarrow r_i + ri_j$ if $d_{k,j} > s_1$ then $d_{k,j} \leftarrow s_1$
          $c_i \leftarrow min$
      for $i \leftarrow 1$ to $k1$ do
          $c1_i \leftarrow d_{i,k}$  $\leftarrow c_i$  $w_i \leftarrow d_{i,k+1}$
      if $k < N$ then $w_k \leftarrow d_{k,k+1}$
  $k1 \leftarrow N$  $r1 \leftarrow GetRow(D, k1)$
  for $i \leftarrow 1$ to $k1 - 1$ do
      $ri \leftarrow getRow(D, i)$
      for $j \leftarrow 1$ to $k1 - 1$ do
          $s_2 \leftarrow c1_i + r1_j$ if $ri_j > s_2$ then $ri_j \leftarrow s_2$
  return $D$
```

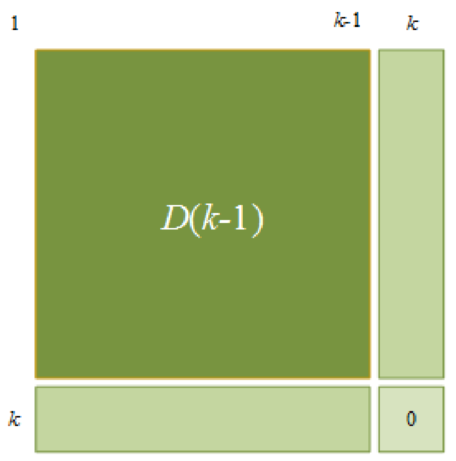


Figure 3 – Illustration of graph extension-based algorithm operation

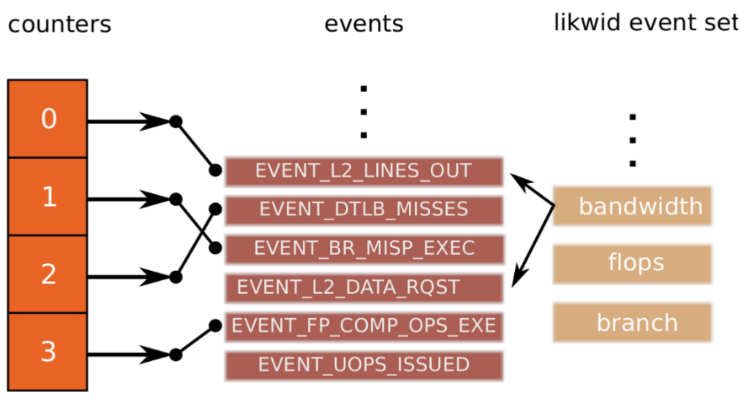


Figure 4 – Counters, events and characteristics of *likwid-perfctr*

Many of Intel processors, including the one we use, contain a RAPL interface that provides an MSR to estimate power consumption for the four power planes of the machine (fig. 5). The power consumption is measured for the CPU cores, GPU, and DRAM separately.

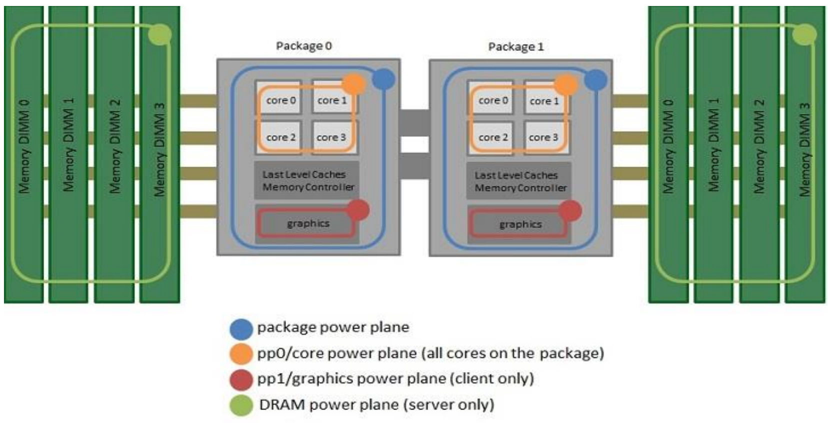


Figure 5 – Architecture of four power planes

Computational experiments that allowed us to evaluate the execution time and energy consumption of the *FW*, *BFW*, and *GEA* algorithms were performed on a computer with an Intel i5-8265U processor. It is an energy efficient processor with 4 cores and 8 hardware threads, built on the Intel Whiskey Lake-U architecture with a power consumption of 15 W and a base clock frequency of 1.60 GHz. The processor has an integrated Intel UHD Graphics 620 graphics card, and the memory controller supports DDR4-2400 and LPDDR3-2133 standards. The computer has 8GB of RAM. The Intel Smart Cache has a capacity of 6 MB. The computer is equipped with a 256 GB SSD drive. The processor cache has three levels: L1, L2 and L3. Each core is equipped with L1 and L2 level caches of 32 Kb and 256 Kb respectively. The L3 level is 6 MB in size and is shared by all cores. The L3 level has the largest capacity but is the slowest, 30 cycles on average. The computer has a Unix-like operating system (Ubuntu 20.04.3 LTS distribution) based on the Linux kernel, the code of which is written in C with some assembly language extensions.

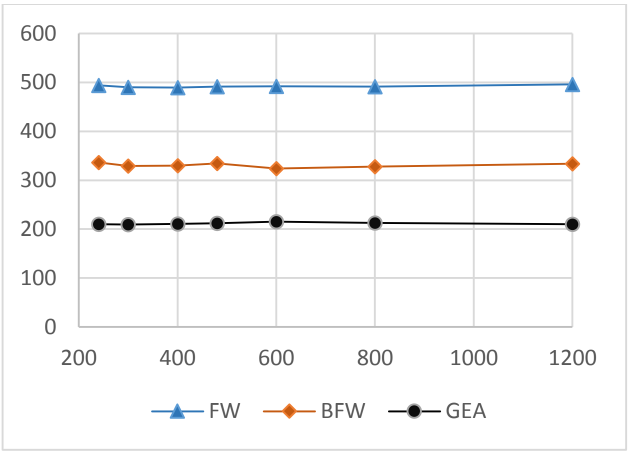
Experiments to measure time and energy consumption were performed on randomly generated complete weighted directed cyclic graphs of various sizes. In the paper, the results of profiling on graphs consisting of 1200 and 2400 vertices are given.

Fig. 6 shows the run-time in *sec* of the *FW*, *BFW*, and *GEA* algorithms on graphs of 1200 and 2400 vertices for different block sizes of matrix *B*. The parameters of *FW* and *GEA* do not depend on the block size, therefore, fig. 6 shows variations in their run-time. In terms of execution time, *GEA* is significantly superior to *FW* and *BFW*.

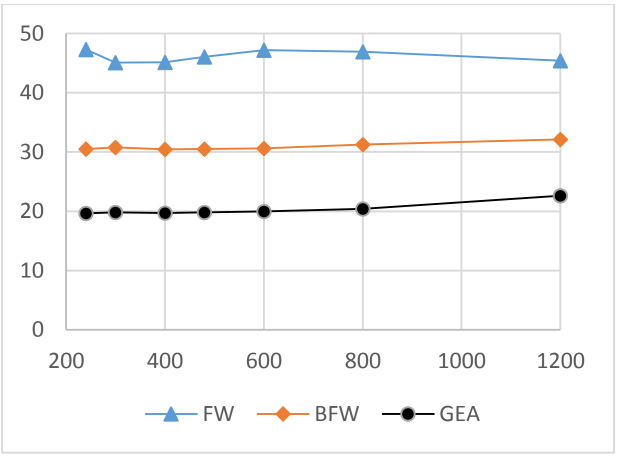
Fig. 7 shows the energy (J) consumed by the DRAM memory when executing the *FW*, *BFW*, and *GEA* algorithms. The *GEA* algorithm significantly outperforms the *FW* and to a lesser extent, outperforms the *BFW* in terms of DRAM energy consumption.

Fig. 8 shows that the processor cores consume significantly more energy (J) when executing the *FW* algorithm in contrast to executing the *GEA* algorithm. The *GEA* algorithm also outperforms the *BFW* in this respect.

Tab.1 gives a comparison of the three algorithms on three parameters, expressed in %. *GEA* has the largest advantage over *FW* in in terms of memory energy. *GEA* has the largest advantage over *BFW* in terms of cores energy and execution time.

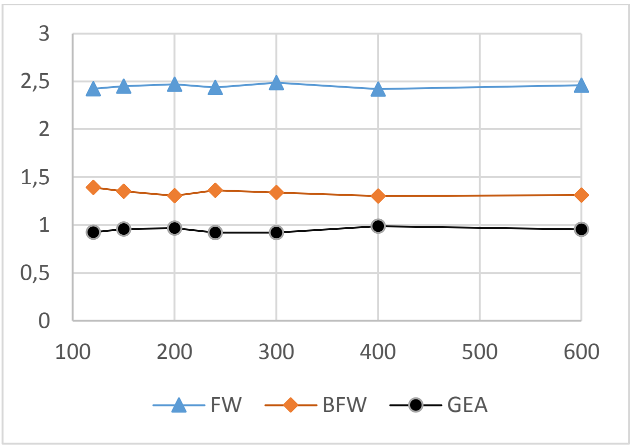


a)

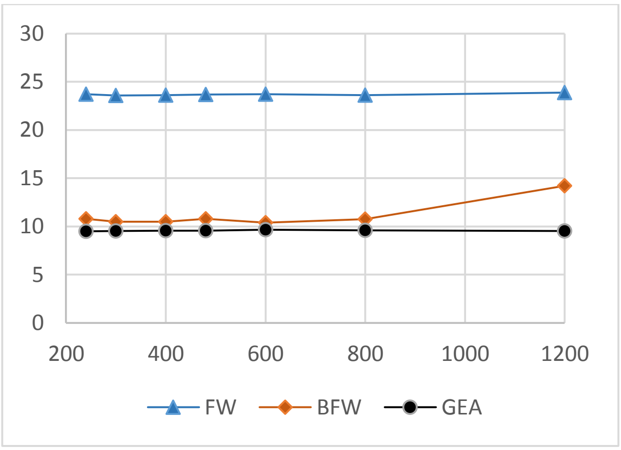


b)

Figure 6 – Run-time (sec) of algorithms *FW*, *BFW* and *GEA* on graphs of a) 1200 and b) 2400 vertices

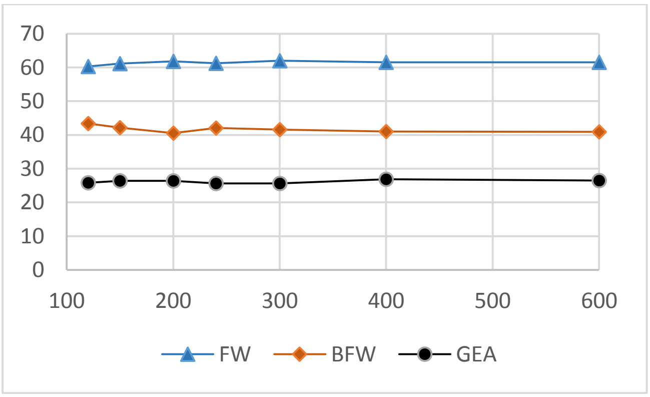


a)

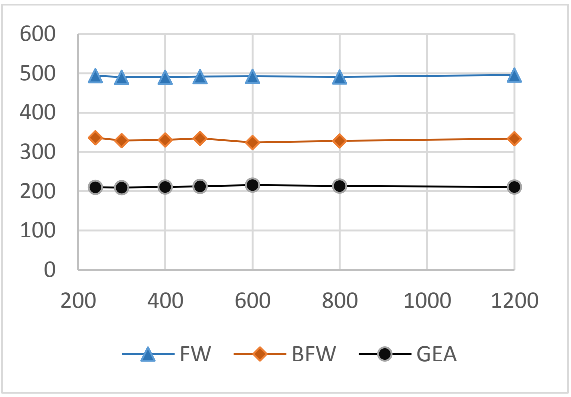


b)

Figure 7 – Energy (J) consumed by DRAM algorithms *FW*, *BFW* and *GEA* on graphs of a) 1200 and b) 2400 vertices



a)



b)

Figure 8 – Energy (J) consumed by processor cores for *FW*, *BFW*, and *GEA* algorithms on graphs consisting of a) 1200 and b) 2400 vertices

Table 1 – Average reduction (%) in time and energy resources by the *GEA* algorithm compared to the *FW* and *BFW* algorithms on two graph sizes

Algorithms	Graphs on 1200 vertices			Graphs on 2400 vertices		
	Time, %	Energy of memory, %	Energy of cores, %	Time, %	Energy of memory, %	Energy of cores, %
GEA / *FW*	130.8	158.7	134.6	128.2	147.9	132.6
GEA / *BFW*	57.1	41.4	59.3	52.6	16.5	56.3

Conclusion. Profiling of program code executed on a multi-core system is an effective means of measuring and estimating parameters of competing algorithms and detecting areas of their preferable use. It can let us reduce the consumption of computational and energy resources significantly when solving typical tasks such as searching for the shortest paths between all pairs of vertices in a graph. In particular, the research results obtained in this paper show that the recently proposed algorithm based on sequential addition of vertices to the graph has convincing time and energy advantages over the well-known classical Floyd-Warshall algorithm and its blocked version, focused on increasing cache efficiency and organization of parallel computations on multicore systems.

References

1. 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.

2. 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.

3. Prihozhy A.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.

4. Prihozhy A.A., Casale-Brunet S., Bezati E., Mattavelli M. Pipeline Synthesis and Optimization from Branched Feedback Dataflow Programs. Journal of Signal Processing Systems, 2020, vol. 92, pp. 1091–1099.

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

6. 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.

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

8. 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.

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

10. 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.

11. 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.

12. 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.

13. 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.

14. 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.

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

16. 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.

17. 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.

18. Treibig J., Hager G., Wellein G. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. 39th International Conference on Parallel Processing Workshops, 2010, pp. 1–10.