UDC 004.272.2
DOI: 10.21122/2309-4923-2022-3-57-65

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

TUNING BLOCK-PARALLEL ALL-PAIRS SHORTEST PATH ALGORITHM FOR EFFICIENT MULTI-CORE IMPLEMENTATION

Belarusian National Technical University

Finding shortest paths in a weighted graph is one of the key problems in computer-science, which has numerous practical applications in multiple domains. This paper analyzes the parallel blocked all-pairs shortest path algorithm at the aim of evaluating the influence of the multi-core system and its hierarchical cache memory on the parameters of algorithm implementation depending on the size of the graph and the size of distance matrix's block. It proposes a technique of tuning the block-size to the given multi-core system. The technique involves profiling tools in the tuning process and allows the increase of the parallel algorithm throughput. Computational experiments carried out on a rack server equipped with two Intel Xeon E5-2620 v4 processors of 8 cores and 16 hardware threads each have convincingly shown for various graph sizes that the behavior and parameters of the hierarchical cache memory operation don't depend on the graph size and are determined only by the distance matrix's block size. To tune the algorithm to the target multi-core system, the preferable block size can be found once for the graph size whose in-memory matrix representation is larger than the size of cache shared among all processor's cores. Then this block-size can be reused on graphs of bigger size for efficient solving the all-pairs shortest path problem.

Keywords: *shortest path; Floyd-Warshall algorithm; blocked algorithm; multithreaded application; multi-core system; hierarchical cache memory; parallelism; throughput.*

Introduction

The problem of finding a shortest path exists for ages. It has a long history of being deeply investigated by different researchers to solve various practical problems, starting from solving mazes and ending by optimization of networks [1,2]. The shortest path problem has two formulations: finding a shortest path between a source and each other vertex in a weighted graph (Single Source Shortest Path – SSSP) and finding shortest paths between all pairs of vertices (All Pairs Shortest Path – APSP). Dijkstra's algorithm solves SSSP and has a $O(n^2)$ computational complexity. Floyd-Warshall's algorithm solves APSP and has $O(n^3)$ computational complexity. Both problems are computationally expensive for large graphs. On graph of over 10000 vertices the algorithms require impractical amount of time, even on modern hardware. That is why, effective parallelization of the algorithms on multi-core systems is an important computational problem.

The algorithm parallelization requires highly qualified professionals [3] to adapt algorithm's mechanics and implement it in a way to meet features of the target machine, which is a separate challenge due to the increasing number of cores and their architectural differences [4]. The effective algorithm parallelization depends on multiple factors including (but not limited to) the distribution of worker threads between processor's cores [5,6] and optimization of hierarchical cache memory usage [7].

In this paper we are focusing on analyzing the block-parallel APSP algorithm and tuning it with respect to the graph and block sizes to account for the multi-core system architecture and its hierarchical cache memory with the objective of increasing the algorithm implementation throughput.

Block-parallel shortest paths algorithm

The Floyd-Warshall algorithm [8] operates on a cost adjacency matrix $D[N \times N]$, where N is a number of vertices in a graph. The matrix is initialized with weights of the edges in such a way that element D_{ij} contains a weight of the edge between vertices i and j (upon completion, element D_{ij} will contain a length of the shortest path between the vertices). When an edge is absent the value of D_{ij} is ∞. The algorithm recalculates all elements of D within each of N iterations of a loop along the graph vertices.

The authors of [9] proposed a blocked (also known as "tiled") version of the Floyd-Warshall algorithm. This version splits matrix D into blocks of size $S \times S$, effectively creating a matrix $B[M \times M]$ of blocks, where equality $M \cdot S=N$ holds. It performs M iterations, each consisting of three phases (see Figure 1) of calculating the "diagonal" block (depends on itself), $2 \cdot (M-1)$ "cross" blocks (each depends on itself and the corresponding "diagonal" block), and $(M-1)^2$ "peripheral" blocks (each depends on the corresponding vertical and horizontal "cross" blocks.

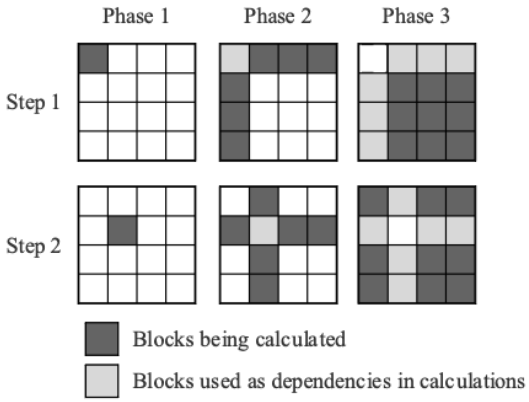


Figure 1. Illustration of calculation phases of block-parallel Floyd-Warshall algorithm on first two iterations (steps)

In [10–13], the authors shown that the blocked Floyd-Warshall algorithm can be parallelized due to all "cross" blocks are calculated mutually in parallel and all "peripheral" blocks are calculated mutually in parallel too. The "diagonal", "cross" and "peripheral" blocks are calculated sequentially. Algorithm 1 describes the resulting block-parallel Floyd–Warshall algorithm by means of OpenMP facilities.

It should be noted that the order of block calculation within a loop iteration along m doesn't have to match the above-described calculation phases. Instead, it can be purely driven by the data dependencies among blocks [14–17]. Des-pite differences in data dependencies, which potentially can be exploited [18], all blocks are calculated using the same procedure (see Algorithm 2).

Algorithm 1: Block-parallel Floyd–Warshall

```
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 ← N / S    B[M×M] ← W[N×N]
  #pragma omp parallel
  #pragma omp single
  for m ← 1 to M do
      BCA (B_m,m, B_m,m, B_m,m)                    // Diagonal
      for i ← 1 to M do
          if i ≠ m then
              #pragma omp task untied
              BCA (B_i,m, B_i,m, B_m,m)            // Cross
              #pragma omp task untied
              BCA (B_m,i, B_m,m, B_m,i)            // Cross
      #pragma omp task wait
      for i ← 1 to M do
          if i ≠ m then
              for j ← 1 to M do
                  if j ≠ m then
                      BCA (B_i,j, B_i,m, B_m,j)    // Peripheral
                      #pragma omp task untied
      #pragma omp task wait
  return B
```

Problem formulation

Although the computational complexity of both the basic and blocked Floyd-Warshall algorithms is the same, the blocked algorithm contrary to the basic one obtains a property of spatial locality, which is very important regarding the reduction of multiple data transfers between fast and slow memory levels in a multi-core system.

In the basic Floyd-Warshall algorithm, matrix D is allocated in row-major style in main memory. For large graphs, the matrix size exceeds the last level cache (LLC) size, which leads to a significant memory traffic because the algorithm reads (and

Algorithm 2: Block calculation *BCA*

```
Input: A size S of block
Input: Input blocks B^1, B^2 and B^3
Output: Recalculated block B^1
  for k ← 1 to S do
      for i ← 1 to S do
          for j ← 1 to S do
              sum ← b^2_i,k + b^3_k,j
              if b^1_i,j > sum then b^1_i,j ← sum
  return B^1
```

probably writes) every element of *D* within every iteration. This may cause a complete or partial reload of the matrix from the main memory in each iteration and may lead to pore performance.

The blocked Floyd-Warshall algorithm performs computations over blocks, with at most three active blocks at a time. Due to spatial locality, it can improve the performance on both small (when *D* doesn't' fit in L1 processor cache) and large (when *D* doesn't fit in LLC) graphs.

In [7], the authors shown that on state-of-the-art processors of that time the algorithm can reduce the process-memory traffic by a factor of *S*. In [9], to minimize L1 cache misses the authors proposed to choose the size *S* of block which meets the following inequality:

$$3 \cdot E \cdot S^2 \leq C \quad (1)$$

where *E* is the size of matrix element and *C* is the size of L1 cache. The proposed approach works for serial implementations of the algorithm.

Nowadays, the situation has changed with respect to finding the matrix block size as the size of processed graphs has grown, modern computing systems have many levels of hierarchical memory, each level's volume can vary from one processor to another, and the number of cores and their parallelization potential has increased. New techniques of searching for an optimal (or preferable) size of block of large-size matrices have to be developed.

In this work we develop a multicore-system profiler-based technique [17,19,20] to analyze the parallel operation of cores within one processor and the behavior of many-level cache memory to tune the size of block in the block-parallel all-pairs shortest path algorithm with the aim of increasing the implementation throughput.

Algorithm profiling on multi-core system

To understand the dependence of the algorithm execution time on the multi-core system architecture and the organization of its hierarchical cache memory we used the Intel VTune Profiler 2021.8. The profiler has facilities to measure the following PMU (Performance Monitor Unit) MEM_LOAD_UOPS_RETIRED events:

- L1_HIT_PS – indicates L1 hit.
- cL2_HIT_PS – indicates L2 hit (also means L1 miss)
- L3_HIT_PS – indicates L3 hit (also means L2 miss)
- L3_MISS_PS – indicates L3 miss and access to RAM.

These events as well as the execution time were collected all at once without multiplexing [19] on every run of the algorithm. In all experimental results, the value of each event is a sum of all such events recorded on all cores and processors during a sample interval [20].

In the paper, we report results obtained on a rack server equipped with two Intel Xeon E5-2620 v4 processors containing 8 cores and 16 hardware threads each. Every core is equipped with a private L1 (32 KB) and L2 (256 KB) caches, and all processor cores share inclusive L3 (20 MB) cache. Table 1 reports the cache latencies. Similar profiling results were obtained on other processors. The algorithm was implemented in C++ language using GNU GCC compiler v10.2.0 and parallelized by means of OpenMP 4.5.

Table 1. Sizes and approximate latencies of L1, L2 and L3 caches for Intel Xeon E5-2620 v4 processor

Name	Size	Latency
L1 (data)	32 KB	4 cycles
L2 (unified)	256 KB	12 cycles
L3 (unified, inclusive)	20 MB	30 cycles

We conducted a series of experiments on randomly generated directed graphs. Here we report results for graphs of 4800, 9600 and 19200 vertices. It should be noted that the matrix D size representing a graph of 4800 vertices is already larger than the L3 cache size. Every experiment was repeated multiple times and the results of computation were verified. To ensure that the VTune profiler doesn't introduce significant noise, the execution time was measured with and without the profiler attached.

All experiments were conducted on block sizes as follows: 30x30, 48x48, 50x50, 75x75, 100x100, 120x120, 150x150, 160x160, 192x192, 200x200, 240x240 and 300x300. All block-sizes divide the matrix into blocks of equal size without remainders.

Table 2. Number of blocks fit in each level of cache vs. block size

Block Size	30	48	50	75	100	120	150	160	192	200	240	300
L1 (blocks)	9,10	3,56	3,28	1,46	0,82	0,57	0,36	0,32	0,22	0,20	0,14	0,09
L2 (blocks)	72,82	28,44	26,21	11,65	6,55	4,55	2,91	2,56	1,78	1,64	1,14	0,73
L3 (blocks)	5825,42	2275,56	2097,15	932,07	524,29	364,09	233,02	204,80	142,22	131,07	91,02	58,25

Results of algorithm profiling

Tables 3, 4, and 5 report the block-parallel algorithm execution time and the number of L1, L2, L3 and main memory hits obtained as an average of 10 runs with the VTune profiler attached. “Break points” (block sizes where the algorithm no longer efficiently uses current level of cache and starts to rely on the next level) represented by → are the most interesting information in the tables. For instance, the block-size of 75x75 is a “break-point” because three blocks of the size don’t fit in L1 cache (see Table 2), therefore the algorithm starts to more extensively use L2 cache. The number of L1 cache hits is getting lower and the number of L2 cache hits increases significantly (around 5 times). The number of L3 hits reduces around 2 times and L3 miss around 3 times, which means less L2 misses. We can also see a stabilization of the number of L1 cache hits on larger block sizes (from 100 to 300). The standard deviation over 10 runs of the algorithm is on average 1.23% for L1 hits for block sizes 100 – 300 on 4800 vertices of graph, is 1.55% on 9600 vertices and 1.93% on 19200 vertices.

The block size of 150x150 is the second “break point” (between L2 and L3 caches) where the number of L2 cache hits is reduced (around 1.5 times) and the number of L3 hits is significantly increased (around 2.5 times). Then the number of L2 hits continuously grows, the number of L3 hits temporary grows (with changing to gradual reduction after the block size of 192x192) and the number of L3 miss continuously reduces. The increase of L2 hits is caused by the fact that L1 cache can’t hold even a fraction of block (for instance, the L1 cache contains only 1/3 of a block of 160x160 size). The behavior of L3 cache is associated with the adaptation of L2 cache to the

Table 3. Event count vs. block size for block-parallel algorithm on a graph of 4800 vertices; profiler contribution is up to 1.70%; arrow (→) represents “break point”; bold indicates maximum number of events; filled cell indicates minimal execution time.

Event / Block	30	48	50	75	100	120	150	160	192	200	240	300
L1 hit (10^5)	**46873**	37283	40439 → 38530		34845	34828	35994	35043	34642	34761	34819	35170
L2 hit (10^3)	20856	5729	6057 → 30399		59277	60426 → 37956		49947	64643	63477	64425	**70316**
L3 hit (10^2)	58665	19270	16655 → 8565		6335	6455 → 13350		33165	**90865**	78835	72825	62675
L3 miss (10^2)	**28445**	12440	11115 → 4240		2355	1820 → 1150		895	1075	620	380	295
Time (ms)	**13727**	5364	5062	4140	3548	3529	3833	3767	3836	3879	4058	4109

Table 4. Event count vs. block size for block-parallel algorithm on graph of 9600 vertices; profiler contribution is up to 1.54%.

Event / Block	30	48	50	75	100	120	150	160	192	200	240	300
L1 hit (10^5)	**377481**	291461	321318 → 307641		277972	276581	285402	276281	273637	272373	272941	273142
L2 hit (10^3)	163676	37276	43748 → 224790		386066	418948 → 256369		384977	463208	510601	523463	**562839**
L3 hit (10^2)	444895	105410	95930 → 44870		35850	41505 → 106605		259275	**717800**	691815	618055	519415
L3 miss (10^2)	**224255**	73665	84025 → 29610		15665	13345 → 9180		7330	5990	5805	3850	2510
Time (ms)	**107813**	38269	38839	31679	27373	26943	29080	28331	28436	28408	28301	28342

Table 5. Event count vs. block size for block-parallel algorithm on graph of 19200 vertices; profiler contribution is up to 1.74%.

Event / Block	30	48	50	75	100	120	150	160	192	200	240	300
L1 hit (10^5)	**377481**	291461	321318 → 307641		277972	276581	285402	276281	273637	272373	272941	273142
L2 hit (10^3)	163676	37276	43748 → 224790		386066	418948 → 256369		384977	463208	510601	523463	**562839**
L3 hit (10^2)	444895	105410	95930 → 44870		35850	41505 → 106605		259275	**717800**	691815	618055	519415
L3 miss (10^2)	**224255**	73665	84025 → 29610		15665	13345 → 9180		7330	5990	5805	3850	2510
Time (ms)	**107813**	38269	38839	31679	27373	26943	29080	28331	28436	28408	28301	28342

Table 6. Caches and main memory hits in percent over the total number of events (L1 + L2 + L3 hits + L3 miss) for graph of 4800 vertices vs. block size

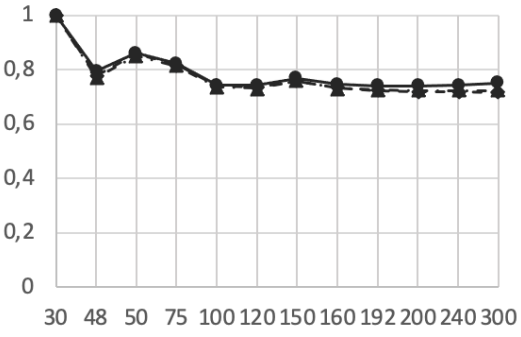
Block Size	30	48	50	75	100	120	150	160	192	200	240	300
L1 hit	99,37	99,76	99,78	99,18	98,30	98,27	98,92	98,50	97,91	97,99	97,98	97,87
L2 hit	0,44	0,15	0,15	0,78	1,67	1,70	1,04	1,40	1,83	1,79	1,81	1,96
L3 hit	0,13	0,06	0,04	0,03	0,02	0,02	0,04	0,09	0,26	0,22	0,20	0,17
Main memory hit	0,06	0,03	0,03	0,01	0,01	0,01	<0,01	<0,01	<0,01	<0,01	<0,01	<0,01

Table 7. Hits of L1, L2 and L3 cahes in percent over the cache related events for graph of 4800 vertices vs. block size

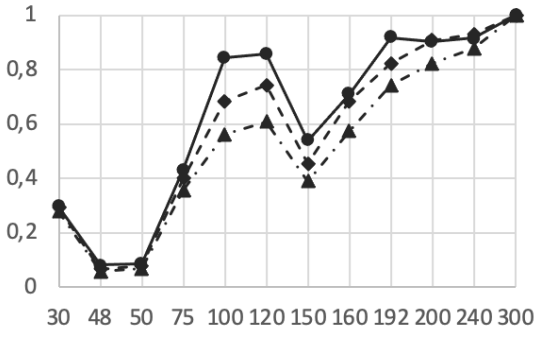
Block Size	30	48	50	75	100	120	150	160	192	200	240	300
L1 hit / L1	99,37	99,76	99,78	99,18	98,30	98,27	98,92	98,50	97,91	97,99	97,98	97,87
L2 hit / L2	70,54	64,37	68,56	95,96	98,56	98,65	96,32	93,62	87,55	88,88	89,80	91,78
L3 hit / L3	67,35	60,77	59,97	66,89	72,90	78,01	92,07	97,37	98,83	99,22	99,48	99,53

situation when it can't fit three blocks, then two blocks, and in the end a single block. The continuously reduction of L3 misses is explained by the fact that with increasing the block-size almost all requests are served by L2 or L3 caches (see Table 2).

Table depicts shares in percent of the cache hits and misses vs. block size for the graph-size of 4800 vertices. L1 cache processes from 99.37 % down to 97.87 % of data requests depending on the block-size. The L2 and L3 caches process much less requests. Table 7 shows that the less share of requests to L2 cache is processed at the L2 level against L1 cache. The same concerns L3 cache. The number of requests to the main memory falls with the block-size growth. Shares for graphs of 9600 and 19200 vertices are close to those of graph of 4800 vertices.



a)



b)

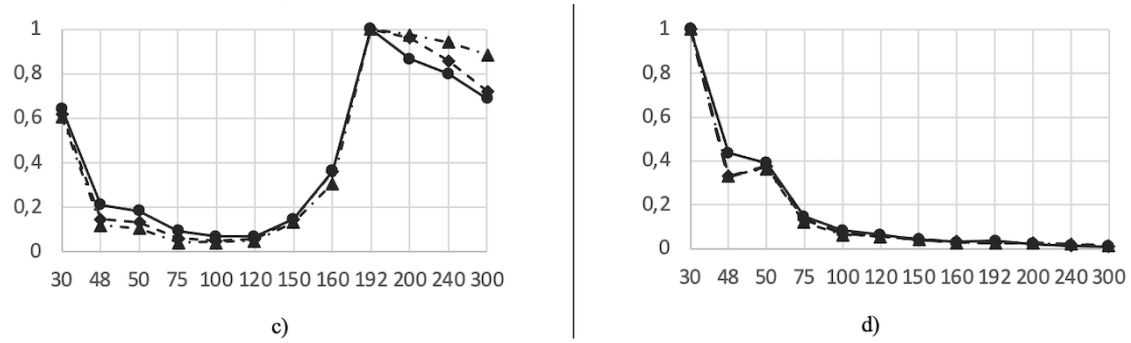


Figure 2. Normalized maximum count (bold figures in Tables 3-5) of a) L1 hits b) L2 hits c) L3 hits and d) L3 misses across graphs of 4800 (solid), 9600 (dashed) and 19200 (dashed dotted) vertices vs. block-size

The results obtained on two Intel Xeon E5-2620 v4 processors (see Figure 2) and one Intel Core i5-6200U clearly demonstrate that the cache usage by the algorithm depends mainly on the block size and almost doesn't depend on the graph size. Moreover, the maximum values for L1, L2, L3 hits and L3 miss shown in bold in Tables 3–5 correspond to the same block sizes for all of the graphs. Figure 3 shows that the changes in execution time follow the pattern that is similar to one of the cache usages: they get increased or reduced in the same manner. It is important to understand why after reaching the block size of 120x120 the execution time continues to increase while the L3 misses decrease and L2 and L3 hits increase. An explanation can be seen in the latencies of the L1, L2 and L3 cache levels depicted in Table 1.

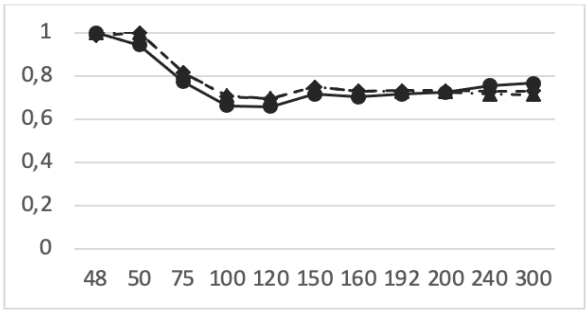


Figure 3. Normalized minimum execution time from Tables 3–5 for graphs of 4800 (solid), 9600 (dashed) and 19200 (dashed dotted) vertices

Block-parallel algorithm tuning technique

Now we can formulate the block-parallel algorithm tuning technique targeting the shortest paths problem on large graph and increasing the throughput of the multi-core systems. The technique consists of the steps as follows:

1. Studying the features of structure and parameters of the multi-core system.
2. Attaching a multi-core system profiler to the algorithm program code.
3. Selecting or generating a weighted graph which in-memory matrix representation is larger than the last level cache size.
4. Profiling the algorithm on the graph for various block size.
5. Finding the block-size giving a minimum of algorithm execution time.
6. Solving the shortest paths problem on larger graphs with high throughput using the block-size determined on step 5.

Conclusion

The Floyd-Warshall block-parallel all-pairs shortest path algorithm requires a tuning of the block-size to modern multi-core system architectures at the aim of increasing throughput. The main result of the paper is the proposed technique of finding an optimal (preferable) block size experimentally on a smaller graph by means of a multi-core system profiler and then using the found block-size for multiple solving with higher throughput the shortest paths problem on large graphs. To obtain the result we have analysed the hierarchical cache memory usage by the algorithm and have demonstrated that it doesn't depend on the graph size but instead depends on the selected block size. We have also experimentally demonstrated that the optimal block size is no longer can be found in the same way as it was done for the sequential algorithm, therefore, leaving discovery of optimal block size to experimental lookup and future research.

ЛИТЕРАТУРА

1. **Schrijver, A.** On the history of the shortest path problem / A. Schrijver // Documenta Mathematica. – 2012. – Vol. 17, №. 1. – P. 155–167.

2. **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.

3. **Atachiants, R.** Parallel Performance Problems on Shared-Memory Multicore Systems: Taxonomy and Observation / R. Atachiants, G. Doherty, D. Gregg // IEEE Transactions on Software Engineering. – 2016. – Vol. 42, №. 8. – P. 764–785.

4. **Zheng, Y.** Performance evaluation of exclusive cache hierarchies / Y. Zheng, B. T. Davis, M. Jordan. – 2004. – P. 89–96.

5. **Прихожий А.А., Карасик О.Н.** Исследование методов реализации многопоточных приложений на многоядерных системах // Информатизация образования, 2014, № 1, с. 43–62.

6. **Прихожий А.А., Карасик О.Н.** Кооперативная модель оптимизации выполнения потоков на многоядерной системе // Системный анализ и прикладная информатика, 2014, № 4, с. 13–20.

7. **Park, J.** Optimizing graph algorithms for improved cache performance / J. Park, M. Penner, V. K. Prasanna // IEEE Transactions on Parallel and Distributed Systems. – 2004. – Vol. 15, №. 9. – P. 769–782.

8. **Floyd, R. W.** Algorithm 97: Shortest Path / R. W. Floyd // Communications of the ACM. – 1962. – Vol. 5, №. 6. – P. 345-.

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

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

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

12. **Singh, A.** Performance Analysis of Floyd Warshall Algorithm vs Rectangular Algorithm / A. Singh, P. K. Mishra // International Journal of Computer Applications. – 2014. – Vol. 107, №. 16. – P. 23–27.

13. An Experimental Study of a Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances / K. Madduri [et al.] // 2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX). – Society for Industrial and Applied Mathematics, 2007. – P. 23–35.

14. **Карасик, О. Н.** Кооперативный многопоточный планировщик и блочно-параллельные алгоритмы решения задач на многоядерных системах / О. Н. Карасик. – Белорусский государственный университет информатики и радиоэлектроники, 2019.

15. **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.

16. **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.

17. **Прихожий, А.А.** Моделирование кэш прямого отображениея и ассоциативных кэш на алгоритмах поиска кратчайших путей на графе / А.А. Прихожий // Системный анализ и прикладная информатика. – 2019. – No. 4. – С. 10–18.

18. **Prihozhy, A.** Inference of shortest path algorithms with spatial and temporal locality for Big Data processing / A. Prihozhy, O. Karasik // Big Data and Advanced Analytics: сб. материалов VIII Междунар. науч.-практ. конф., Минск, 11-12 мая 2022. – Минск: Бесринт, 2022. – P. 56–66.

19. Intel Corporation. Allow Multiple Runs or Multiplex Events [Electronic resource]. – Mode of access: URL: https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top/analyze-performance/hw-event-based-sampling-collection/allow-multiple-runs-or-multiplex-events.html. – Date of access: 07.03.2022.

20. Intel Corporation. Hardware Event-based Sampling Collection [Electronic resource]. – Mode of access: URL: https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top/analyze-performance/hw-event-based-sampling-collection.html – Date of access: 07.03.2022).

REFERENCES

1. **Schrijver, A.** On the history of the shortest path problem / A. Schrijver // Documenta Mathematica. – 2012. – Vol. 17, №. 1. – P. 155–167.

2. **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.

3. **Atachiants, R.** Parallel Performance Problems on Shared-Memory Multicore Systems: Taxonomy and Observation / R. Atachiants, G. Doherty, D. Gregg // IEEE Transactions on Software Engineering. – 2016. – Vol. 42, №. 8. – P. 764–785.

4. **Zheng, Y.** Performance evaluation of exclusive cache hierarchies / Y. Zheng, B. T. Davis, M. Jordan. – 2004. – P. 89–96.

5. **Prihozhy, A.A.** Investigation of methods for implementing multithreaded applications on multicore systems / A.A. Prihozhy, O.N. Karasik // Informatization of education. – 2014. – No. 1. – P. 43–62.

6. **Prihozhy, A.A.** Cooperative model for optimization of execution of threads on multi-core system / A.A. Prihozhy, O.N. Karasik // System analysis and applied information science. – 2014. – No. 4. – P. 13–20.

7. **Park, J.** Optimizing graph algorithms for improved cache performance / J. Park, M. Penner, V. K. Prasanna // IEEE Transactions on Parallel and Distributed Systems. – 2004. – Vol. 15, №. 9. – P. 769–782.

8. **Floyd, R. W.** Algorithm 97: Shortest Path / R. W. Floyd // Communications of the ACM. – 1962. – Vol. 5, №. 6. – P. 345-.

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

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

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

12. **Singh, A.** Performance Analysis of Floyd Warshall Algorithm vs Rectangular Algorithm / A. Singh, P. K. Mishra // International Journal of Computer Applications. – 2014. – Vol. 107, №. 16. – P. 23–27.

13. An Experimental Study of a Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances / K. Madduri [et al.] // 2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX). – Society for Industrial and Applied Mathematics, 2007. – P. 23–35.

14. **Karasik, O.N.** Cooperative multi-threaded scheduler and block-parallel algorithms of solving tasks on multi-core systems / O.N. Karasik. – Belarusian state university of informatics and radio-electronics, 2019.

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

16. **Prihozhy, A.A.** Cooperative block-parallel algorithms for task execution on multi-core system / A.A. Prihozhy, O.N. Karasik // System analysis and applied information science. – 2015. – No. 2. – P. 10–18.

17. **Prihozhy, A.A.** Simulation of direct mapped, k-way and fully associative cache on all pairs shortest paths algorithms / A.A. Prihozhy // System analysis and applied information science. – 2019. – No. 4. – P. 10–18.

18. **Prihozhy, A.A.** Inference of shortest path algorithms with spatial and temporal locality for Big Data processing / A.A. Prihozhy, O.N. Karasik // Big Data and Advanced Analytics: Proc. VIII Intern. Conf., Minsk, May 11-12, 2022. – Minsk: Bestprint, 2022. – P. 56–66.

19. Intel Corporation. Allow Multiple Runs or Multiplex Events [Electronic resource]. – Mode of access: URL: https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top/analyze-performance/hw-event-based-sampling-collection/allow-multiple-runs-or-multiplex-events.html. – Date of access: 07.03.2022.

20. Intel Corporation. Hardware Event-based Sampling Collection [Electronic resource]. – Mode of access: URL: https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top/analyze-performance/hw-event-based-sampling-collection.html – Date of access: 07.03.2022).

О. Н. КАРАСИК, А. А. ПРИХОЖИЙ

НАСТРОЙКА БЛОЧНО-ПАРАЛЛЕЛЬНОГО АЛГОРИТМА ПОИСКА КРАТКИХ ПУТЕЙ НА ЭФФЕКТИВНУЮ МНОГОЯДЕРНУЮ РЕАЛИЗАЦИЮ

Белорусский национальный технический университет

Поиск кратчайших путей во взвешенном графе — одна из ключевых задач компьютерных наук, которая имеет множество практических приложений в различных областях. В данной работе анализируется блочно-параллельный алгоритм поиска кратчайших путей с целью оценки влияния многоядерной системы и ее иерархической кэш-памяти на параметры реализации алгоритма в зависимости от размера графа и размера блока матрицы расстояний. В ней предлагается метод настройки размера блока на особенности многоядерной системы. Метод предполагает использование инструментов профилирования в процессе настройки и позволяет увеличить производительность параллельного алгоритма. Вычислительные эксперименты, проведенные на стоечном сервере, оснащенном двумя процессорами Intel Xeon E5-2620 v4, состоящих из 8 ядер и 16 аппаратных потоков каждый, убедительно показали для различных размеров графов, что поведение и параметры работы иерархической кэш-памяти слабо зависят от размера графа и определяются размером блока матрицы расстояний. Чтобы настроить алгоритм на целевую многоядерную систему, предпочтительный размер блока может быть найден один раз для графа, размер представления которого превышает размер кэша, совместно используемого ядрами процессора. После этого найденный размер блока можно многократно использовать для эффективного решения задачи о кратчайших путях на графах большего размера.

Ключевые слова: *кратчайший путь; алгоритм Флойда-Уоршелла; блочный алгоритм; многопоточный алгоритм; многопроцессорная система; иерархическая кэш память, параллелизм.*

Анатолий Прихожий, профессор кафедры Программное обеспечение информационных систем и технологий Белорусского национального технического университета, доктор технических наук (1999 г.), профессор (2001 г.). Исследовательские интересы: языки программирования и описания оборудования, распараллеливающие компиляторы, а также методы и инструменты автоматизированного проектирования программного и аппаратного обеспечения на логическом, высоком и системном уровнях, а также для неполностью определенных логических систем. Имеет более 300 публикаций в Восточной и Западной Европе, США и Канаде. Его работы публиковали такие мировые издательства, как IEEE, Springer, Kluwer Academic Publishers, World Scientific и другие.

Anatoly Prihozhy, full professor at the Computer and system software department of Belarus national technical university, doctor of science (1999) and full professor (2001). His research interests include programming and hard- ware 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.

prihozhy@bntu.by

Карасик Олег, технический директор компании ISsoft Solutions (часть Coherent Solutions) в Минске, Беларусь, кандидат технических наук. Его исследовательские интересы включают параллельные многопоточные приложения и распараллеливание для многоядерных и многопроцессорных систем.

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