UDC [611.018.51+615.47]:612.086.2

PARALLEL BLOCKED ALL-PAIR SHORTEST PATH ALGORITHM: BLOCK SIZE EFFECT ON CACHE OPERATION IN MULTI-CORE SYSTEM

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

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

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

O.N. Karasik

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

A.A. Prihozhy

Full professor at the "Computer and system software" department of Belarusian national technical university, Doctor of Science (1999) and full professor (2001). His research interests include programming and hardware description languages, parallelizing compilers, and computer aided design techniques and tools for software and hardware at logic, high and system levels, and for incompletely specified logical systems. He has over 300 publications in Eastern and Western Europe, USA, and Canada. Such worldwide publishers as IEEE, Springer, Kluwer Academic Publishers, World Scientific, and others have published his works.

Abstract. The problem of finding all-pairs of shortest path in a graph is a classical computer-science problem which has numerous practical applications in multiple domains. This paper analyzes a parallel version of the blocked all-pair shortest path algorithm at the aim of evaluating the influence of the hierarchical cache memory on the parameters of algorithm implementations on multi-processor/multi-core systems. Computational experiments carried out by means of a profiling tool on various graph sizes have convincingly shown that the behavior and parameters of the cache memory operation don't depend on the graph size and are determined only by the distance matrix block size. Obtained results show, that for every target machine the optimal block size can be found once in the case the graph size isn't high, it is divisible by the block size, and it is larger than the size of processor's last level shared cache. After that the optimal block size can be reused for efficient solving of the shortest paths problem on all graphs of larger size.

Keywords: shortest path, Floyd-Warshall algorithm, blocked algorithm, multithreaded application, multiprocessor system, hierarchical cache memory, parallelism, throughput.

Problem formulation

The problem of finding a shortest path exists for ages. It has a long history of being deeply investigated by different researchers to solve different problems, starting from solving mazes and ending by optimization of networks [1,2]. The shortest path problem has two classes: finding a shortest path between two vertices in a graph and finding shortest paths between all pairs of vertices in a graph. The former is called **S**ingle **S**ource **S**hortest **P**ath (SSSP), or **P**oint-to-**P**oint (P2P), and the latter is called **A**ll **P**airs **S**hortest **P**ath (APSP). Both problems are computationally expensive. The go-to algorithm for solving SSSP problems is Dijkstra's algorithm. It has a $O(n^2)$ computational complexity. For APSP problems the go-to algorithm is Floyd-Warshall's algorithm which has $O(n^3)$ computational complexity. On large graphs (over 10000 vertices) these algorithms will require impractical amount of time, even on modern hardware. That is why, effective parallelization of such algorithms is an important computational problem.

Parallelization of any algorithm is a complex and time-consuming work. It requires a highly qualified professionals [3] to adapt algorithm's mechanics and then implement them in a way to run effectively on target machines, which presents a separate challenge due number of cores and theirs architecture differences [4].

However, effective parallelization of any algorithm depends on multiple factors including (but not limited to): 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 an existing parallel algorithm for solving APSP problem to understand usage of hierarchical cache memory and how it is affected by major algorithm's parameters – block size and graph size.

Algorithm description

Floyd-Warshall's algorithm [8], whose pseudocode is presented in

Figure 1, operates on a cost adjacency matrix D of size N, where N is a size of a graph, initialized with weights of the graph edges in such a way, that element $D[i,j]$ contains a weight of the edge between vertices i and j (upon completion, element $D[i,j]$ will contain a length of the shortest path between vertices i and j). The algorithm scans the matrix and checks existence of a path from vertex i to vertex j through existence of paths from i to k and from k to j.

```
int M = ...
...
function algorithm(matrix D)
  for k = 0 to N do
    for i = 0 to N do
      for j = 0 to N do
        D[i,j] = min(D[i,j], D[i,k] + D[k,j])
      end
    end
  end
end function
```

Figure 1 – Pseudocode of original Floyd-Warshall algorithm

Considering that matrix D is represented linear in memory, and assuming that matrix size is larger than LLC (**L**ast **L**evel **C**ache) size, such implementation leads to a significant memory traffic because on every iteration k, the algorithm reads (and in the worst case, writes) every element of D. This might cause a complete reload of the matrix from the main memory, which in turn leads to pure performance. To 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 [9] authors proposed a blocked (also known as "tiled") version of Floyd-Warshall's algorithm (see

).

```
int M = ...
int L = ...
...
function algorithm_srl(block_matrix B)
  for m = 0 to M do
    proc(B[m,m], B[m,m], B[m,m]);
    for i = 0 to m - 1 do
      proc(B[i,m], B[i,m], B[m,m]);
      proc(B[m,i], B[m,m], B[m,i]);
    end
    for i = m + 1 to M - 1 do
      proc(B[i,m], B[i,m], B[m,m]);
      proc(B[m,i], B[m,m], B[m,i]);
    end
    for i = 0 to m – 1 do
      for j = 0 to m–1 do
        proc(B[i,j], B[i,m], B[m,j]);
      end
      for j = m + 1 to M – 1 do
        proc(B[i,j], B[i,m], B[m,j]);
      end
    end
    for i = m + 1 to M - 1 do
      for j = 0 to m – 1 do
        proc(B[i,j], B[i,m], B[m,j]);
      end
      for j = m + 1 to M – 1 do
        proc(B[i,j], B[i,m], B[m,j]);
      end
    end
  end
end function
```

```
int M = ...
int L = ...
...
function algorithm_prl(block_matrix B)
  #pragma omp parallel
  #pragma omp single
  for m = 0 to M do
    proc(B[m,m], B[m,m], B[m,m]);
    for i = 0 to m - 1 do
      #pragma omp task untied
      proc(B[i,m], B[i,m], B[m,m]);
      #pragma omp task untied
      proc(B[m,i], B[m,m], B[m,i]);
    end
    for i = m + 1 to M - 1 do
      #pragma omp task untied
      proc(B[i,m], B[i,m], B[m,m]);
      #pragma omp task untied
      proc(B[m,i], B[m,m], B[m,i]);
    end
    #pragma omp taskwait
    for i = 0 to m – 1 do
      for j = 0 to m–1 do
        #pragma omp task untied
        proc(B[i,j], B[i,m], B[m,j]);
      end
      for j = m + 1 to M – 1 do
        #pragma omp task untied
        proc(B[i,j], B[i,m], B[m,j]);
      end
    end
    for i = m + 1 to M - 1 do
      for j = 0 to m – 1 do
        #pragma omp task untied
        proc(B[i,j], B[i,m], B[m,j]);
      end
      for j = m + 1 to M – 1 do
        #pragma omp task untied
        proc(B[i,j], B[i,m], B[m,j]);
      end
    end
    #pragma omp taskwait
  end
end function
```

```
function proc(B1, B2, B3)
  for k = 0 to L do
    for i = 0 to L do
      for j = 0 to L do
        B1[i,j] = min(B1[i,j], B2[i,k] + B3[k,j])
      end
    end
  end
end function
```

Figure 2 – Pseudocodes of serial (*algorithm_srl*) and parallel (*algorithm_prl*) version of blocked Floyd-Warshall algorithm. The parallelization is done using OpenMP

This version splits matrix D into blocks of size L, effectively creating a matrix B of blocks of size M, where $M * L = N$. Algorithm performs M iterations, each consisting of three phases: calculation of “diagonal” block, calculation of “cross” blocks and calculation of the “peripheral” blocks (see **Ошибка! Источник ссылки не найден.**).

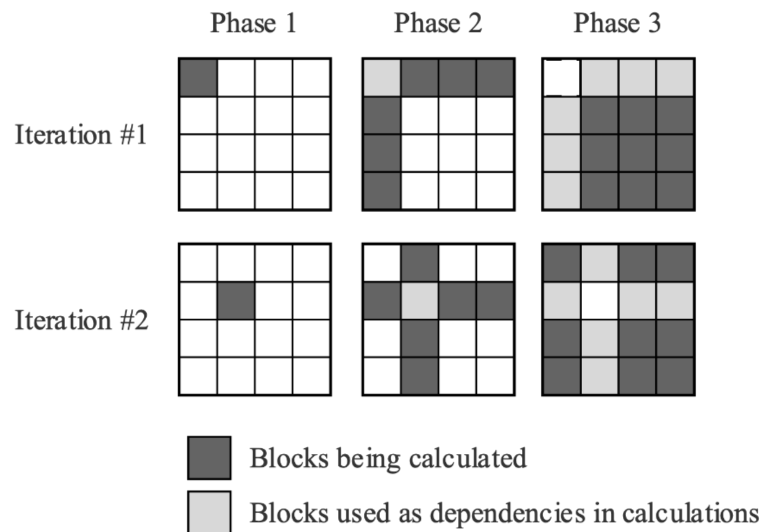

Figure 3 – Illustration of calculation phases of blocked version of Floyd-Warshall's algorithm on example of first two iterations.

The order of block calculation within an iteration doesn't have to match the above-described phases. Instead, it can be purely driven by data dependencies between blocks[1]:

–"diagonal" block depends on itself.

–"cross" block depends on the "diagonal" block and self.

–"peripheral" block depends on the corresponding "cross" blocks (both vertical and horizonal).

The computation is now done in blocks, with at most three active blocks (when computing "peripheral" blocks) at a time. This reduces the process-memory traffic by a factor of L [7]. The blocked version can be parallelized using OpenMP directives. In parallel version, on each phase, all blocks are calculated in parallel[2]. The parallel version presented on
is used in all presented experiments.

In [9], the optimal block size (to minimize L1 cache misses) is determined using the following equation:

$$3L^2 * E \leq C \tag{1}$$

where L – is the size of the block
E – is the size of matrix element in bytes
C – is the size of L1 cache

In addition, L should be a multiple of S/E (where S is a size of processor cache line, which is the smallest unit of data brought into L1 cache). This equation works for serial version of the algorithm. However, as stated in [7] and also demonstrated in our experiments for parallel version

[1] Despite differences in data dependencies all blocks are calculated using the same procedure (see
). The exploitation of the facts that "diagonal" and "cross" and "peripheral" blocks depend on itself was covered in multiple researches [10–12]. However, in this research we aren't focusing on these improvements for simplicity reasons.

[2] It is obvious that such implementation while being simple has a drawback in form of non-optimal utilization of data dependencies. The possibilities to optimize computations by the excessive use of data dependencies is covered in multiple researches [13–16].

(where the optimal block size is turned to be 120x120 instead of 48x48 defined by the equation) the best size of the block must be found experimentally. Looking for an optimal block size for large graphs can be time consuming and in general results in a significant time lose.

In this work we analyze the behavior of hierarchical cache memory and the parallel algorithm execution time to understand the relationship between experimental graph size, optimal block size and cache utilization.

Experimental setup and results

All computational experiments on the parallel version of blocked all-pairs shortest path algorithm (hereinafter algorithm if not mentioned otherwise) were conducted on a rack server equipped with 2xIntel Xeon E5-2620 v4 processors containing 8 cores (16 hardware threads) each. Every core is equipped with a private L1 (32KB), L2 (256KB) caches and all processor cores share inclusive L3 (20MB) cache. The algorithm was implemented in C++ language using GNU GCC compiler v10.2.0 and OpenMP 4.5. We used Intel VTune Profiler 2021.8 to measure cache behavior.

We conducted a series of experiments on multiple randomly generated directed acyclic graphs (4800, 9600 and 19200 vertex) with edge probability of 80%. Every experiment was repeated multiple times and computation results were verified against the results of original Floyd-Warshall algorithm obtained on the same graph. To ensure, attached profiler (VTune) doesn't introduce significant noise, every experiment was executed 10 times with profiler attached and 10 times without it. The difference in execution time was calculated and will accompany every set of experimental results.

All experiments were conducted on multiple block sizes: 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. However, depending on the size of the graph the resulting number of blocks might or might not be divisible by total number of hardware threads – 32.

To understand the usage of hierarchical cache memory by the algorithm we measured the following PMU (**P**erformance **M**onitor **U**nit) events and total execution time:

– MEM_LOAD_UOPS_RETIRED.L1_HIT_PS – indicates L1 hit.

– MEM_LOAD_UOPS_RETIRED.L2_HIT_PS – indicates L2 hit (also means L1 miss)

– MEM_LOAD_UOPS_RETIRED.L3_HIT_PS – indicates L3 hit (also means L2 miss)

– MEM_LOAD_UOPS_RETIRED.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 [17] on every run of the algorithm. Each event value (in tables), ex. L1_HIT_PS, is a sum of all such events recorded on all cores (on all processors) during sample interval [18]. Tables 1–3 report an average of 10 runs with the profiler attached.

The data presented in Tables 1–3 might seem overwhelming, so let's extract important bits of information from them. The most interesting information in the above data are "break points" i.e., block sizes where the algorithm no longer efficiently uses current level of cache and starts to rely on the next one, ex. three blocks don't fit in L1 cache, so algorithm starts to more extensively use L2 cache. To relate these "break points", it is required to see when the cache level can no longer include three blocks. This information is presented in

Table 5.

Table 1. Experimental results of parallel version of blocked all-pair shortest path algorithm on a graph of 4800 vertexes; profiler contribution is up to 1.70%; arrows (→) emphasise "break point"; bold emphasis maximum value of the event; colored cell indicates minimal execution time.

Table 2 – Experimental results of parallel version of blocked all-pair shortest path algorithm on a graph of 4800 vertexes; profiler contribution is up to 1.70%; arrows (→) emphasise "break point"; bold emphasis maximum value of the event; colored cell indicates minimal execution time.

E vent / Bloc k	0	8	50	75	00	120	150	60	92	00	40	00
L 1 hit (10^5)	**68 73**	72 83	40439 → 38530		48 45	34828	35994	50 43	46 42	47 61	48 19	51 70
L 2 hit (10^3)	08 56	72 9	6057 → 30399		92 77	60426 → 37956		99 47	46 43	34 77	44 25	**03 16**
L 3 hit (10^2)	86 65	92 70	16655 → 8565		33 5	6455 → 13350		31 65	**08 65**	88 35	28 25	26 75
L 3 miss (10^2)	**84 45**	24 40	11115 → 4240		35 5	1820 → 1150		95	07 5	20	80	95
T ime (ms)	**37 27**	36 4	5062	4140	54 8	3529	3833	76 7	83 6	87 9	05 8	10 9

Table 3 – Experimental results of parallel version of blocked all-pair shortest path algorithm on graph of 9600 vertexes; profiler contribution is up to 1.54%

E vent / Bloc k	0	8	50	75	00	120	150	60	92	00	40	00
L 1 hit (10^5)	**77 48 1**	91 46 1	321318 → 307641		77 97 2	276581	285402	76 28 1	73 63 7	72 37 3	72 94 1	73 14 2
L 2 hit (10^3)	63 67 6	72 76	43748 → 224790		86 06 6	418948 → 256369		84 97 7	63 20 8	10 60 1	23 46 3	**62 83 9**
L 3 hit (10^2)	44 89 5	05 41 0	95930 → 44870		58 50	41505 → 106605		59 27 5	**17 80 0**	91 81 5	18 05 5	19 41 5
L 3 miss (10^2)	**24 25 5**	36 65	84025 → 29610		56 65	13345 → 9180		33 0	99 0	80 5	85 0	51 0
T ime (ms)	**07 81 3**	82 69	38839	31679	73 73	26943	29080	83 31	84 36	84 08	83 01	83 42

Table 4 – Experimental results of parallel version of blocked all-pair shortest path algorithm on graph of 19200 vertexes; profiler contribution is up to 1.74%.

E vent / Bloc k	0	8	50	75	00	120	150	60	92	00	40	00
L 1 hit (10^5)	**999 389**	329 939	2575667 → 2457022		219 581	2208397	2275142	201 419	169 963	160 880	154 133	151 285
L 2 hit (10^3)	307 591	690 97	328966 → 1674118		641 162	2860500 → 1831359		696 093	492 397	866 370	118 256	**692 303**
L 3 hit (10^2)	442 295	965 40	601210 → 246135		574 20	278130 → 755925		765 010	**691 230**	562 450	360 810	048 290
L 3 miss (10^2)	**763 520**	870 10	645975 → 214050		133 65	100005 → 71330		130 5	674 5	599 5	541 0	699 5
T ime (ms)	**581 30**	053 00	305305	248988	162 79	212737	229031	223 65	224 92	214 61	187 44	179 58

Table 5 – Number of blocks fit in each level of cache depending on the block size.

Block Size	30	48	50	75	10 0	12 0	15 0	16 0	19 2	20 0	2 40	3 00
L1 (blocks)	9,1 0	3,5 6	3,2 8	1,4 6	0,8 2	0,5 7	0,3 6	0,3 2	0,2 2	0,2 0	0, 14	0, 09
L2 (blocks)	72, 82	28, 44	26, 21	11, 65	6,5 5	4,5 5	2,9 1	2,5 6	1,7 8	1,6 4	1, 14	0, 73
L3 (blocks)	582 5,42	227 5,56	209 7,15	93 2,07	52 4,29	36 4,09	23 3,02	20 4,80	14 2,22	13 1,07	9 1,02	5 8,25

According to
Table 5, the first "break point" (between L1 and L2 caches) should reveal itself on a block size of 75x75. Indeed, in Tables 1–3 you can see that when block size reaches 75x75 the number of L1 cache hits lowers a bit and the number of L2 cache hits increases significantly (around 5 times). We also see a reduction in L3 hits (around 2 times) and L3 miss (around 3 times), which means less L2 misses. At the same time, we start to see a stabilization of the number of L1 cache hits on larger block sizes (100 – 300)[3]. Now according to

Table 5, the second "break point" (between L2 and L3 caches) should reveal itself on a block size of 150x150. As we can see in Tables 1–3, when block size reaches 150x150 the number L2 cache hits is reduced (around 1.5 times) and the number of L3 hits is significantly increased (around 2.5 times). Then we can see the continues growth of L2 hits, temporary growth of L3 hits (which changes to gradual reduction after block size reaches 192x192) and continues reduction of L3 miss. The increase of L2 hits is caused by the fact that L1 cache can't hold even a fraction of block (ex. when block size is 160x160 the L1 cache includes only $1/3$ of a block), so the algorithm makes extensive use of L2 cache. The behavior of L3 cache is caused by the adaptation of L2 cache to the fact it can't include three blocks, then two blocks and in the end a single block. The continues reduction of L3 misses is explained by the fact that almost all requests are satisfied by L2 or L3 cache.

You can find the shares of the cache hits and misses for all the experimental graphs relative to the block size in table 6

[3] The standard deviation from average for L1 hit on block sizes 100 – 300 are 1.23% for a graph of 4800, 1.55% for 9600 and 1.93% for 19200.

Table 6 – Shares of the all cache hits and misses from the total number of all cache related events (L1 + L2 + L3 hits + L3 miss), share of L2 hits from number of L2 cache related events (L2 + L3 hits + L3 miss) and share of L3 hits from number of L3 cache related events (L3 hits + L3 miss) for all experimental graphs relative to the block size.

Event / Block Size	30	48	50	75	100	120	150	160	192	200	240	300
4800 vertexes												
L1 hit / Total	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 / Total	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 / Total	0,12	0,05	0,04	0,02	0,02	0,02	0,04	0,09	0,26	0,22	0,20	0,17
L3 miss / Total	0,06	0,03	0,03	0,01	0,01	0,01	<0,01	<0,01	<0,01	<0,01	<0,01	<0,01
L2 hit / L2 Total	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 Total	67,35	60,77	59,97	66,89	72,90	78,01	92,07	97,37	98,83	99,22	99,48	99,53
9600 vertexes												
L1 hit / Total	99,39	99,81	99,81	99,25	98,61	98,49	99,07	98,53	98,08	97,91	97,90	97,80
L2 hit / Total	0,43	0,13	0,14	0,73	1,37	1,49	0,89	1,37	1,66	1,84	1,88	2,02
L3 hit / Total	0,12	0,04	0,03	0,01	0,01	0,01	0,04	0,09	0,26	0,25	0,22	0,19
L3 miss / Total	0,06	0,03	0,03	0,01	0,01	<0,01	<0,01	<0,01	<0,01	<0,01	<0,01	<0,01
L2 hit / L2 Total	70,98	67,55	70,85	96,79	98,68	98,71	95,68	93,52	86,49	87,98	89,38	91,51
L3 hit / L3 Total	66,49	58,86	53,31	60,24	69,59	75,67	92,07	97,25	99,17	99,17	99,38	99,52
19200 vertexes												
L1 hit / Total	99,39	99,83	99,82	99,30	98,81	98,70	99,17	98,71	98,16	97,99	97,88	97,64
L2 hit / Total	0,43	0,12	0,13	0,68	1,18	1,28	0,80	1,21	1,58	1,75	1,87	2,13
L3 hit / Total	0,11	0,03	0,02	0,01	0,01	0,01	0,03	0,08	0,26	0,25	0,24	0,23
L3 miss / Total	0,06	0,03	0,03	0,01	0,01	<0,01	<0,01	<0,01	<0,01	<0,01	<0,01	<0,01
L2 hit / L2 Total	71,52	67,71	72,51	97,32	98,62	98,70	95,68	93,69	85,89	87,33	88,41	90,24
L3 hit / L3 Total	66,12	54,27	48,21	53,49	69,43	73,55	91,38	97,18	99,19	99,18	99,34	99,47

As you can see in Table 5, we didn't experiment on graphs of enormous sizes to exhaust L3 cache and see the stabilization of L2 hits and then increase of L3 miss when less and less blocks fit in it. However, the presented data clearly demonstrates that the cache usage by the algorithm doesn't dependent on the graph size[4] but on the block size (see Figure 2).

[4] You can also notice that maximum values for L1, L2, L3 hits and L3 miss (in **bold**) in Tables 1–3 are registered on the same block factor for all experimental graphs.

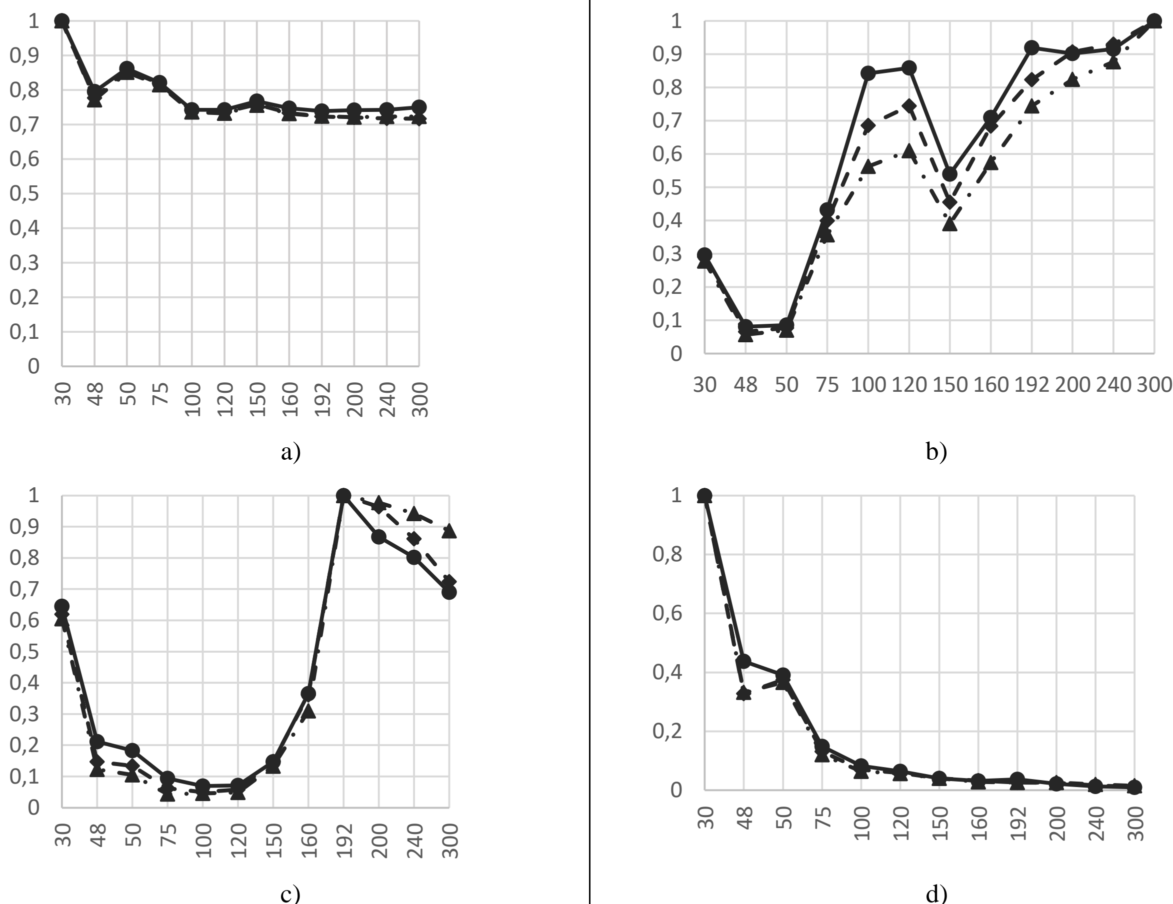

Figure 2 – Normalized (around maximum values – see **bold** values in Tables 1–3) charts of a) L1 hits b) L2 hits c) L3 hits d) L3 misses across all three experimental graphs – 4800 (solid), 9600 (dashed) and 19200 (dashed dotted)

The change in execution time follows the similar pattern as cache usage does (see
Figure 3) – it gets increased or reduced by the same fraction depending on the block size. The important conclusion from this observation is that optimal block size can be found experimentally on smaller graphs and then used to carry out calculations of larger graphs.

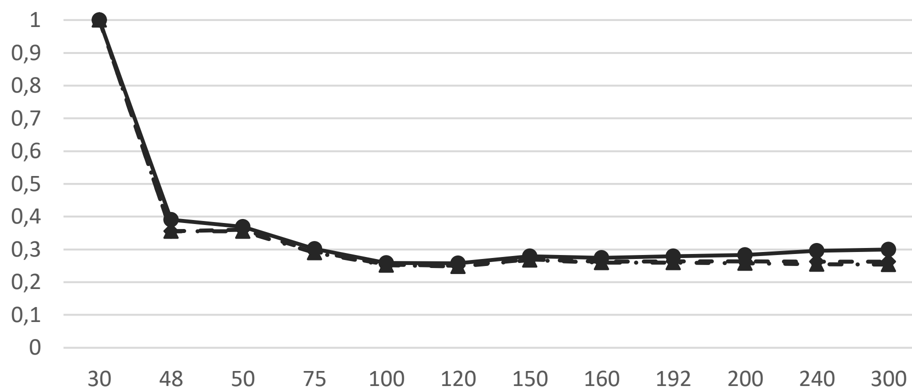

Figure 3 – Normalized (around maimum values – see **bold** values in Tables 1–3) execution time across all three experimental graphs – 4800 (solid), 9600 (dashed) and 19200 (dashed dotted)

Next, it is important to understand why after reaching the block size of 120x120 the execution time continues to increase while at the same time we can see reduction in L3 miss and increase in L2 and L3 hits. The answer lays in the difference between latencies of different cache levels (see

Table 7).

Table 7 – Approximate latencies of L1, L2 and L3 caches for Intel Xeon E5-2620 v4 processor.

Name	Latency
L1 (data)	4 cycles
L2 (unified)	12 cycles
L3 (unified, inclusive)	30 cycles

The latency changes significantly between L1 and L2 (3 times) and then again between L2 and L3 (2.5 times). This is still much faster than accessing DRAM, but the best execution time is dictated by the optimal number of all cache hits on all levels and not one. Hence, it must be possible to analytically calculate optimal block size knowing enough analytical data. However, such analysis is out of scope of this paper.

Conclusion

In this paper we have analyzed hierarchical cache memory usage by the parallel version of blocked all-pair shortest path algorithm and have demonstrated that it doesn't depend on the graph size but instead depends on the selected block size.

Next, we have showed that the optimal block size doesn't depend on the graph size and therefore can be precalculated for large graphs in advance using small graphs (which still must be larger than LLC).

We have also experimentally demonstrated that in case of parallel version of the algorithm, executed on modern hardware (equipped with multiple levels of cache memory and with simultaneous multi-threading support), the optimal block size is no longer can be found in the same way as it was for the sequential version, therefore, leaving discovery of optimal block size to experimental lookup and future research.

References

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

[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, № 3. P. 263–273.

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

[4] Zheng Y., Davis B.T., Jordan M. Performance evaluation of exclusive cache hierarchies // IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software, 2004. 2004. P. 89–96.

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

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

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

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

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

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

[11] Прихожий А.А., Карасик О.Н. Матричный многопоточный параллельный алгоритм поиска кратчайших путей на графе // Материалы международной научно-практической конференции. Минск: РИВШ, 2017. P. 15–18.

[12] Прихожий А.А., Карасик О.Н. Разнородный блочно-параллельный алгоритм учитывает особенности многоядерной архитектуры // Материалы международной научно-технической конференции. Минск: БНТУ, 2018. P. 6–7.

[13] Карасик О.Н. Кооперативный многопоточный планировщик и блочно-параллельные алгоритмы решения задач на многоядерных системах. Белорусский государственный университет информатики и радиоэлектроники, 2019.

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

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

[16] Лиходед Н.А., Сипейко Д.С. Обобщенный блочный алгоритм Флойда-Уоршелла // Журнал Белорусского государственного университета. Математика. Информатика. Белорусский государственный университет, 2019. № 3. P. 84–92.

[17] Intel Corporation. Allow Multiple Runs or Multiplex Events [Electronic resource]. 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 (accessed: 07.03.2022).

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

БЛОЧНО-ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ ПОИСКА КРАТЧАЙШИХ ПУТЕЙ: ВЛИЯНИЕ РАЗМЕРА БЛОКА НА ФУНКЦИОНИРОВАНИЕ КЭШ-ПАМЯТИ МНОГОЯДЕРНОЙ СИСТЕМЫ

О.Н. Карасик

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

А.А. Прихожий

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

Аннотация. Задача нахождения всех кратчайших путей в графе является классической задачей информатики, имеющей многочисленные практические применения в самых различных областях. В статье выполнены профилирование и анализ блочно-параллельной версия алгоритма поиска кратчайшего пути между вершинами графа с целью оценки влияния параметров алгоритма на использование иерархической кэш-памяти на многоядерных системах. Вычислительные эксперименты, проведенные с использованием профилировщика, на различных размерах графов, убедительно показали, что использование алгоритмом кэш-памяти не зависят от размера графа, а определяется только выбранным размером блока. Полученные результаты показывают, что для каждой многоядерной системы оптимальный размер блока может быть найден единожды (если количество вершин в графе делится на выбранный размер блока и размер графа в памяти превышает объем кэш последнего уровня). После этого, найденный оптимальный размер блока может быть повторно использован для эффективного решения задачи кратчайших путей на графах большего размера.

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