UDC [611.018.51+615.47]:612.086.2

INFERENCE OF SHORTEST PATH ALGORITHMS WITH SPATIAL AND TEMPORAL LOCALITY FOR BIG DATA PROCESSING

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

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

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

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.

O.N. Karasik

Tech 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.

Abstract. *The all-pair shortest paths problem on large-size graphs has many crucial application domains in science, engineering and economics. Such computer architectures as multi-core systems explore hierarchical memory consisting of local and shared levels, which differ on memory capacity and data transfer time delays. The cores read and write data through the fast local caches, therefore running algorithms which support locality in big data processing are most efficient. The paper develops an inference technique at the aim of creating all-pair shortest paths algorithms that improve the spatial and temporal reference locality and reduce the cache pressure. It proposes and transforms a graph-extension-based shortest paths search algorithm that obtains the reference locality properties and recalculates the lengths of shortest paths at each step of adding a vertex to the graph. Every step of algorithm transformation introduces additional temporal or spatial locality. Computational experiments carried out on two types of multi-core processor and on graphs of thousands of vertices have shown about 40% speedup of the proposed algorithm against the classic Floyd-Warshall one. The proposed algorithm has also shown a gain of 25 – 35% over the blocked Floyd-Warshall algorithm, which has the property of spatial data locality.*

Keywords: *multi-core processor; hierarchical memory; cache; shortest paths algorithm; big data; spatial locality; temporal locality; algorithm transformation; inferring technique.*

Problem formulation

Modern multi-core processors have hierarchical memory architecture shown in Figure 1. Every core has local caches L1 and L2, and all cores have a shared cache L3, which communicates to main memory. Data may transfer in both directions from core to main memory and vice versa.

Additionally, the cores may transfer data to each other through the shared cache. The memory capacity increases from L1 to L2, from L2 to L3, and from L3 to main memory. The data transfer latency increases in the same direction. Larger memory capacities cause higher latency. The data transfer time in hierarchical memory contributes significantly to the overall execution time of program.

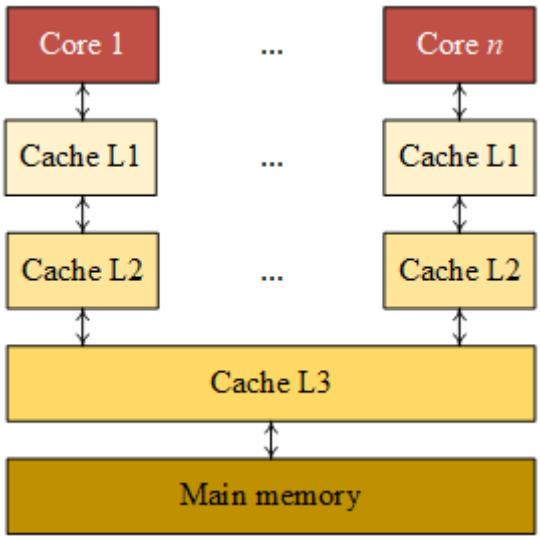

Figure 1. Hierarchical memory of multi-core processor

Locality is a predictable behavior occurring in computer systems. Locality of reference [1-3] is one of the key principles of constructing computer architectures and developing high-performance software. It means the access of processor to the same set of memory locations repetitively over a short period of time. The reference locality can be of two basic types: temporal and spatial. Temporal locality refers to the reuse of specific data within a relatively small-time duration. Spatial locality (also known as data locality) refers to the use of data elements within relatively close storage locations. Spatial locality has a special case termed sequential locality, which occurs when data elements are arranged and accessed linearly. Traversing elements of a one- or many-dimensional array allocated in row-major memory layout is an example of sequential locality. Such techniques as caching, prefetching and branch predicting increase locality of reference.

A working set of information $W(t, \tau)$ of a process at time t is the collection of data referenced by the process during the time interval $(t-\tau, t)$. The working set window dynamically measures the size of working set and estimates reference locality. If the size matches the cache size, the data transfer between the cache and the lower-level memory is not high. The data traffic increases when the active data accede the size of cache. If the working set size is larger than the cache size, intensive data transfer between the memory levels occurs.

The problem of finding shortest and longest paths in a weighted directed cyclic graph [4 – 7] has many important practical applications: reducing city traffic, optimizing network infrastructure in data centers, planning tasks, implementing augmented reality, network analysis, microelectronics, programming, computer networks, computer games etc. Many algorithms developed and published in the literature [8-22] solves the problem. Floyd–Warshall's Algorithm 1 (*FW*) is a basic one among them. Its computational complexity is $O(n^3)$ where n is the number of graph vertices. Although other algorithms have been developed of lower complexity, the Floyd–Warshall algorithm has several advantages: it is applicable to a graph containing positive and negative weights of edges; its space complexity is $O(n^2)$ due to the use of a single displacement array; it has a compact and simple description as a nest of three loops.

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 ← W
for k ← 1 to N do
    for i ← 1 to N do
        for j ← 1 to N do
            sum ← d_i,k + d_k,j
            if d_i,j > sum then d_i,j ← sum
return D
```

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 ← N / S    B[M×M] ← W[N×N]
for m ← 1 to M do
    BCA (B_m,m, B_m,m, B_m,m)                    // D0
    for i ← 1 to M do
        if i ≠ 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 ← 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)    // P3
return B
```

Algorithm 3: Block calculation (*BCA*)

Input: A size S of block
Input: Blocks B^1, B^2 and B^3
Output: B^1 – recalculated block

```
for k ← 1 to S do
    for i ← 1 to S do
        for j ← 1 to S do
            sum ← b²_i,k + b³_k,j
            if b¹_i,j > sum then b¹_i,j ← sum
return B¹
```

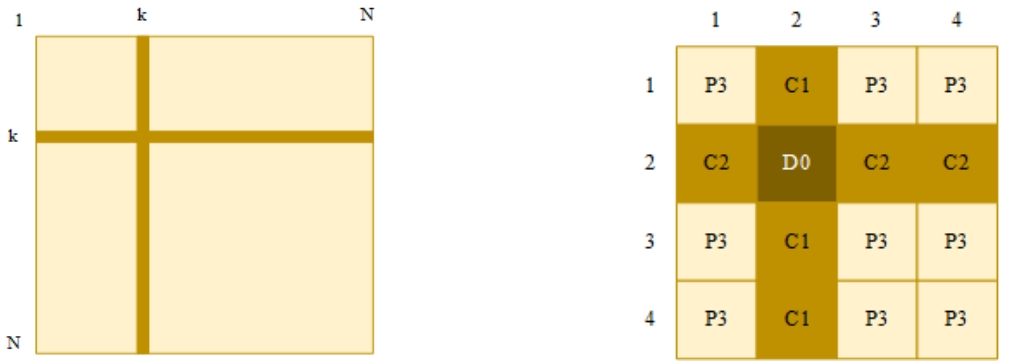

Figure 2. Illustration of algorithm operation a) Floyd-Warshall; b) Blocked Floyd–Warshall

Observing the Floyd–Warshall algorithm we conclude that each iteration along k accesses all elements of matrix D. Totally, every element has N attempts to update. The algorithm operation time crucially depends on which level of memory the matrix D fits completely in. If level L1 or L2 do, the algorithm runs quickly. If level L3 does, the data transfer time delay is higher, and the algorithm runs slower. The slowest case takes place when the size of matrix D is larger than the size of cache L3; elements of D are read from and written to main memory k times, which produces big cache pressure and consumes too much time.

The blocked Floyd–Warshall (*BFW*) Algorithm 2 proposed in [13, 14] has changed the situation. The algorithm introduces spatial locality by decomposing matrix D into blocks of size S and forming blocked matrix $B[M \times M]$, where $M = N / S$ is the number of blocks per row. It provides sequential data locality for each block. Its main loop has M iterations, S times less compared to

FW. Therefore, every iteration updates each element of the matrix as many as *S* times, performing update locally by using one to three blocks simultaneously. Algorithm 3, *BCA* implements the *FW* algorithm, recalculates block B^1 and consumes two additional blocks B^2 and B^3. It is possible to choose the block size in such a way as the processed blocks can be deployed in fast caches simultaneously, which reduces the data traffic between memory levels.

Graph-extension-based shortest paths algorithm

The *FW* algorithm assumes that a graph $G = (V, E)$ is constructed of vertex set V of cardinality N and is represented by matrix W of weights. In our graph-extension-based algorithm, we consider a sequence $G(1)\ldots G(k)\ldots G(N)$ of graphs constructed of 1 to N vertices. We associate the construction process with stepwise adding vertices to graph G. Matrix $D(k)$ describes distances between pairs of vertices in graph $G(k)$. We represent our algorithm as a recurrent procedure (Figure 3) that calculates matrix $D(k)$ from matrix $D(k\text{-}1)$ and weights w_{ik} and w_{kj} of the edges connecting the added vertex k to vertices $i, j \in \{1,\ldots,k\text{-}1\}$. The procedure first adds row k and column k (operation A_k) to $D(k\text{-}1)$ obtaining $D(k)$ and then updates (operation U_k) all elements of $D(k\text{-}1)$.

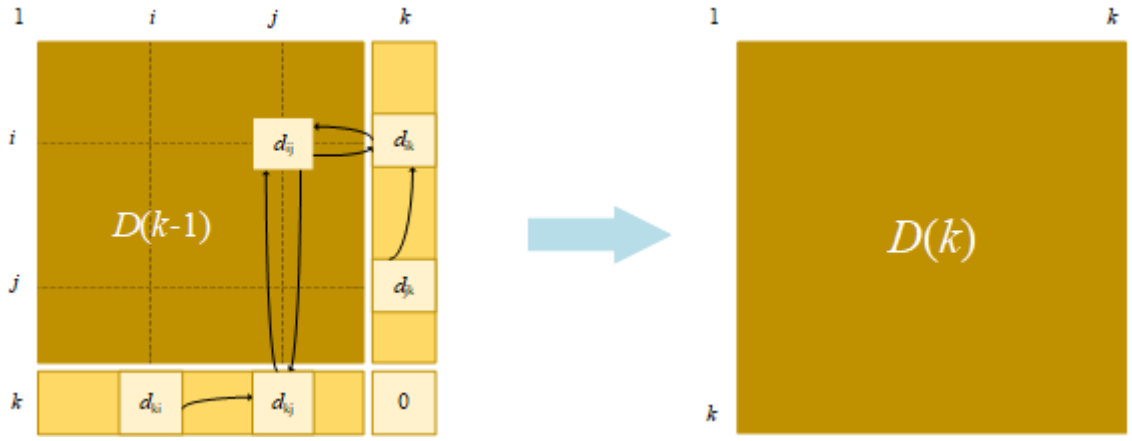

Figure 3. Recurrent procedure of computing distance matrix $D(k)$ from $D(k\text{-}1)$

Equation (1) allows calculating elements $d_{ik}(k)$ of column k.

$$d_{ik}(k) = \min_{j=1\ldots k-1}\left(d_{ij}(k-1) + w_{jk}\right) \quad (1)$$

Equation (2) allows calculating elements $d_{kj}(k)$ of row k.

$$d_{kj}(k) = \min_{i=1\ldots k-1}\left(w_{ki} + d_{ij}(k-1)\right) \quad (2)$$

Note that element $d_{ij}(k\text{-}1)$ is common for (1) and (2). Then the procedure carries out operation U of updating elements of matrix $D(k\text{-}1)$ to elements of matrix $D(k)$ using (3):

$$d_{ij}(k) = \min\left\{d_{ij}(k-1),\, d_{ik}(k) + d_{kj}(k)\right\} \quad (3)$$

where $i, j \in \{1,\ldots,k\text{-}1\}$. We describe the calculations represented by (1), (2) and (3) with a graph-extension-based shortest paths Algorithm 4.

Algorithm 4: Graph-extension-based shortest paths algorithm (*GEA*)

Input: A matrix W of graph edge weights
Input: A size N of matrix
Output: A matrix D of shortest path distances

$D \leftarrow W$
for $k \leftarrow 2$ **to** N **do**
  for $i \leftarrow 1$ **to** $k-1$ **do** // Add A_k
    for $j \leftarrow 1$ **to** $k-1$ **do**
      $s_0 \leftarrow d_{i,j} + d_{j,k}$ **if** $d_{i,k} > s_0$ **then** $d_{ik} \leftarrow s_0$
      $s_1 \leftarrow d_{k,i} + d_{i,j}$ **if** $d_{k,j} > s_1$ **then** $d_{k,j} \leftarrow s_1$
  for $i \leftarrow 1$ **to** $k-1$ **do** // Update U_k
    for $j \leftarrow 1$ **to** $k-1$ **do**
      $s_2 \leftarrow d_{i,k} + d_{k,j}$ **if** $d_{i,j} > s_2$ **then** $d_{i,j} \leftarrow s_2$
return D

Algorithm 4, *GEA* obtains new properties compared to *FW*. The iteration scheme of two loops along *i* and *j* has changed. The loops perform *k* iterations instead of *N* ones in *FW*. They process submatrices of size [1×1], [2×2], …, [N×N] in *N* iterations along *k*. We can observe that in contrast to *FW*, which processes full matrix *D*[*N*×*N*] in each iteration, *GEA* has the property of temporal locality. In *GEA*, the number of body iterations of the most nested loops and the overall amount of processed data are three times less than those in *FW*.

Resynchronization of computations in the algorithm

The recurrent procedure is iteratively executed over all vertices of graph $G(k)$ producing a sequence of pairs of the operations: $A_1U_1–A_2U_2–…–A_NU_N$. It is easy to see that the add and update operations of pair A_kU_k are incompatible in sense of merging two nests of loops along *i* and *j*. Instead, the operations of pair U_kA_{k+1} are compatible, therefore, it is preferable to use the resynchronized sequence $U_1A_2–U_2A_3–…–U_{N-1}A_N–U_N$ considering that A_1 is replaced by a zero initialization of matrix $D(1)$. In the $U_{k-1}A_k$ pair, operation U_{k-1} is a delayed update of matrix $D(k-1)$, which is carried out simultaneously with the addition A_k of row *k* and column *k*.

Algorithm 5 represents *GEA* after resynchronization of the add and update operations. The pseudocode consists of two nests of loops. The first nest has the depth of three loops, and the second nest has the depth of two loops. The first nest consists of a loop along *k* of *N* iterations whose body includes two sequential nests each of two loops along *i* and *j*. In the first nest, which performs operation U_{k-1} by using one addition +, one comparison > and two assignments, the loops along *i* and *j* have *k*-1 iterations each, and in the second nest, which performs operation A_k by using two additions +, two comparisons > and four assignments, the loops have *k* iterations each. Algorithm 5 finalizes computation of matrix *D* by performing the update operation U_N, which is realized by the nest of two loops along *i* and *j* having *N*-1 iterations each. A drawback of the algorithm is the use of two nests of loops to perform operations U_{k-1} and A_k sequentially.

Algorithm 5: *GEA* after resynchronization of computations

Input: A matrix W of graph edge weights
Input: A size N of matrix
Output: A matrix D of shortest path distances

```
D ← W
for k ← 2 to N do
    k1 ← k – 1
    for i ← 1 to k1 – 1 do                          // Update U_{k-1}
        for j ← 1 to k1 – 1 do
            s2 ← d_{i,k1} + d_{k1,j}    if d_{i,j} > s2 then d_{i,j} ← s2
    for i ← 1 to k1 do                              // Add A_k
        for j ← 1 to k1 do
            s0 ← d_{i,j} + d_{j,k}      if d_{i,k} > s0 then d_{i,k} ← s0
            s1 ← d_{k,i} + d_{i,j}      if d_{k,j} > s1 then d_{k,j} ← s1
k1 ← N
for i ← 1 to k1 – 1 do                              // Update U_N
    for j ← 1 to k1 – 1 do
        s2 ← d_{i,k1} + d_{k1,j}    if d_{i,j} > s2 then d_{i,j} ← s2
return D
```

Merging loops in the algorithm

In Algorithm 5, two nests of loops along i and j are different for operations U_{k-1} and A_k upon the right bound of iteration scheme. For the first nest, the bound is $k1 - 1$, and for the second nest it is $k1$. Nevertheless, the transformed Algorithm 6 has merged the nests using the right bound of $k1$ in such a way as to keep the correctness of calculations. Algorithm 6 differs to Algorithm 5 by additional iterations of two loops and by additional execution of two statements

$$s_2 \leftarrow d_{i,k1} + d_{k1,j};\ \textbf{if}\ d_{i,j} > s_2\ \textbf{then}\ d_{i,j} \leftarrow s_2;$$

which implement operation U_{k-1}. We describe the additional iterations with two cases.

Case 1: $i = k1$ and $j = 1,\ldots,k1$. The two statements are reduced to

$$s_2 \leftarrow d_{k1,k1} + d_{k1,j};\ \textbf{if}\ d_{k1,j} > s_2\ \textbf{then}\ d_{k1,j} \leftarrow s_2;$$

Since equality $d_{k1,k1} = 0$ holds and $s_2 = d_{k1,j}$, the value of $d_{i,j} = d_{k1,j}$ keeps unchanged.

Case 2: $j = k1$ and $i = 1,\ldots,k1$. The two statements are reduced to

$$s_2 \leftarrow d_{i,k1} + d_{k1,k1};\ \textbf{if}\ d_{i,k1} > s_2\ \textbf{then}\ d_{i,k1} \leftarrow s_2;$$

Since equality $d_{k1,k1} = 0$ holds and $s_2 = d_{i,k1}$, the value of $d_{i,j} = d_{i,k1}$ also keeps unchanged.

Our conclusion is the additional loop iterations in Algorithm 6 does not affect the D matrix state. Therefore, Algorithms 5 and 6 are functionally equivalent.

The advantages of Algorithm 6 are as follows. First, it eliminates two nested loops and reduces CPU time on implementing the iteration schemes. Second, it increases temporal reference locality due to three references to the same variable $d_{i,j}$ in three subsequent statements which calculate and consume values of variables s_2, s_0 and s_1.

Figure 4 illustrates the operation of Algorithm 6, which is different to the operation of Algorithm 4. Algorithm 4 references the row and column indexed by k. Algorithm 6 simultaneously references two rows and two columns indexed by $k - 1$ and k. Algorithm 4

calculates the row k and column k upon elements of matrix $D(k–1)$ and then updates elements of $D(k–1)$ upon row k and column k to obtain matrix $D(k)$. Algorithm 6 updates elements of matrix $D^*(k–2)$ upon row $k–1$ and column $k–1$ and calculates the row k and column k of matrix $D^*(k)$ upon elements of matrix $D^*(k\text{-}1)$ for which $D^*(k–2)$ is a part. Matrix $D(k)$ produced by operations U_k and A_k in Algorithm 4 differs from matrix $D^*(k)$ produced by operations U_kA_{k+1} in Algorithm 6.

Improving spatial reference locality in cache

In Algorithm 6, the nest of three loops along k, i and j carries out operations U_{k-1} and A_k upon matrix D. The algorithm calculates variables $s2$, $s0$ and $s1$, and matrix elements $d_{i,j}$, $d_{i,k}$ and $d_{k,j}$ upon elements $d_{i,k1}$, $d_{k1,j}$, $d_{i,j}$, $d_{j,k}$ and $d_{k,i}$. To do this, it traverses columns k-1 and k, and rows k-1, k and i multiple times. The rows provide sequential locality and low data transfer in the hierarchical memory. The column elements deployed to different lines and referred many times in nested loops increase the data traffic in hierarchical memory. To avoid the increase, our solution is to preliminary collect the elements in a one-dimensional array, which provide sequential reference locality, and then to access the elements many times.

Algorithm 7 inferred from Algorithm 6 implements our solution. It explores three additional one-dimensional arrays for collecting elements of two columns of matrix D: array $c1$ corresponds to updated column k-1, array w corresponds to initial column k, and array c corresponds to column k that is to be updated. Moreover, it uses references to three rows of matrix D: $r1$ refers to row k-1; r refers to row k; ri refers to row i. Function *getRow*(D, k) returns the address of row k of matrix D. To further speed up the computations, we have implemented Algorithm 7 by means of pointers of the C programming language.

Algorithm 6: *GEA* after merging loops

Input: A matrix W of graph edge weights
Input: A size N of matrix
Output: A matrix D of shortest path distances

```
D ← W
for k ← 2 to N do
    k1 ← k – 1
    for i ← 1 to k1 do
        for j ← 1 to k1 do
            s2 ← d_i,k1 + d_k1,j    if d_i,j > s2 then d_i,j ← s2    // Update U_k-1
            s0 ← d_i,j + d_j,k      if d_i,k > s0 then d_i,k ← s0    // Add A_k
            s1 ← d_k,i + d_i,j      if d_k,j > s1 then d_k,j ← s1    // Add A_k
k1 ← N
for i ← 1 to k1 – 1 do
    for j ← 1 to k1 – 1 do
        s2 ← d_i,k1 + d_k1,j    if d_i,j > s2 then d_i,j ← s2        // Update U_N
return D
```

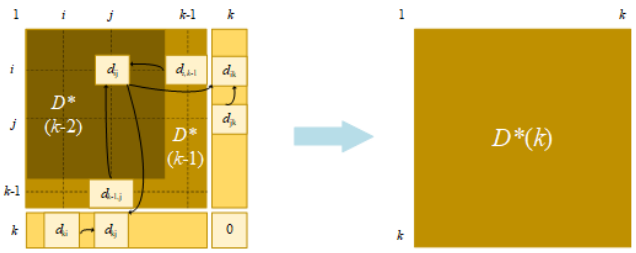

Figure 4. Resynchronized recurrent procedure of computing $D(k)$ from $D(k\text{-}1)$

Algorithm 7: *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 ← W    c1_1 ← ∞    w_1 ← d_{1,2}
for k ← 2 to N do
    k1 ← k – 1    r ← getRow(D, k)    r1 ← getRow(D, k1)
    for i ← 1 to k1 do
        min ← ∞    ri ← getRow(D, i)
        for j ← 1 to k1 do
            s_2 ← c1_i + r1_j    if ri_j > s_2 then ri_j ← s_2      // Update U_{k-1}
            s_0 ← ri_j + w_j    if min > s_0 then min ← s_0          // Add A_k
            s_1 ← r_i + ri_j    if d_{k,j} > s_1 then d_{k,j} ← s_1    // Add A_k
        c_i ← min
    for i ← 1 to k1 do
        c1_i ← d_{i,k} ← c_i    w_i ← d_{i,k+1}
    if k < N then w_k ← d_{k,k+1}
k1 ← N    r1 ← GetRow(D, k1)
for i ← 1 to k1 – 1 do
    ri ← getRow(D, i)
    for j ← 1 to k1 – 1 do
        s_2 ← c1_i + r1_j    if ri_j > s_2 then ri_j ← s_2          // Update U_N
return D
```

Comparison of proposed and Floyd-Warshall algorithms

The iteration scheme of loops along i and j of *GEA* differs from those of *FW*. In *FW*, the overall number of iterations of the most nested loop is N^3. In *GEA*, the overall number of iterations of the loop is $\eta = 1^2 + 2^2 + 3^2 + \ldots + N^2$. Equation (4) evaluates the ratio $\rho = N^3 / \eta$.

$$\rho = B^3 \Big/ \left(B(B+1)(2B+1)/6\right) = 6 \Big/ \left(2 + 3/B + 1/B^2\right) \tag{4}$$

When $N \to \infty$, the ratio $\rho \to 3$. In this case, *GEA* has the overall number of iterations in the most nested loop of $N^3 / 3$, which is three times less than in *FW*.

FW updates N^2 matrix elements in each iteration of the loop along k. It recalculates totally N^3 values. *GEA* successively updates matrices $D[1\times1]$, $D[2\times2]$, …, $D[N\times N]$ of the increasing size. The overall number of updated matrix elements is N^3 / ρ. It is equal to $N^3 / 3$ when $N \to \infty$. Comparing *GEA* against *FW*, we can conclude that the first algorithm is better than the second one regarding both the number of loop iterations executed, and the amount of data processed.

Experimental results

We have implemented the *FW*, *BFW* and *GEA* algorithms [23-28] in the C language, compiled them into single-thread applications using Visual Studio 2019 Community Edition (MSVC++ 14.29) with «Release» configuration and O2 optimization level, and carried out computational experiments on two processors: P1 – Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz and P2 – Intel(R) Core(TM) i5-5200U @ 2.70GHz. The single-thread applications allow evaluating the influence of spatial and temporal locality in cache hierarchy (not the effect of parallel behavior) on the algorithm throughput. Table 1 reports the memory capacity of caches L1, L2 and L3, the processor frequency and the number of cores. We have used randomly generated complete graphs whose size N varies from 400 to 3600 vertices. Algorithms *FW* and *GEA* operated

on non-blocked matrix $D[N \times N]$. Algorithm *BFW* operated on blocked matrix $B[8 \times 8]$ at various graph size. Table 2 and Table 3 report CPU-time the algorithms have consumed while running on processors P1 and P2 respectively. We can observe that the algorithms have consumed about twice larger time on P2 against P1. Since the processors have almost the same frequency, the gain of P1 over P2 is due to the times larger size of caches. We can also observe that *BFW* is faster than *FW*, and *GEA* is faster than both *FW* and *BFW*. Figure 5 shows the speedup of *BFW* and *GEA* against *FW* on each of the P1 and P2 processors. *GEA* is faster to *FW* from 38.92 % to 40.10 % on P1, and from 27.23 % to 42.22 % on P2. As for *BFW*, it is faster to *FW* only from 5.93 % to 13.04 % on P1, and from 3.40 % to 14.39 % on P2.

Table 1. Processor parameters

Processor	L1	L2	L3	Frequency	Cores
Intel(R) Core(TM) i7-10700 CPU	0.5 MB	2.0 MB	16.0 MB	2.9 GHz	8
Intel(R) Core(TM) i5-5200U	0.2 MB	0.5 MB	3.0 MB	2.7 GHz	2

Table 2. CPU time (millisecond) of algoritms *FW*, *BFW* and *GEA* on P1 - Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz vs. graph size

Algorithm	Graph size *N*								
	400	800	1200	1600	2000	2400	2800	3200	3600
FW	118	925	3162	7148	13745	24080	38196	57141	81752
BFW	111	839	2805	6459	12318	21605	34248	50156	71092
GEA	72	565	1894	4358	8365	14518	23283	34383	49040

Table 3. CPU time (millisecond) of algoritms *FW*, *BFW* and *GEA* on P2 - Intel(R) Core(TM) i5-5200U @ 2.70GHz vs. graph size

Algorithm	Graph size *N*								
	400	800	1200	1600	2000	2400	2800	3200	3600
FW	235	1962	6402	15086	29262	49163	79580	117717	157617
BFW	227	1856	6101	14185	25372	42086	70041	105143	142021
GEA	171	1205	3992	9435	18574	28407	47812	69812	94309

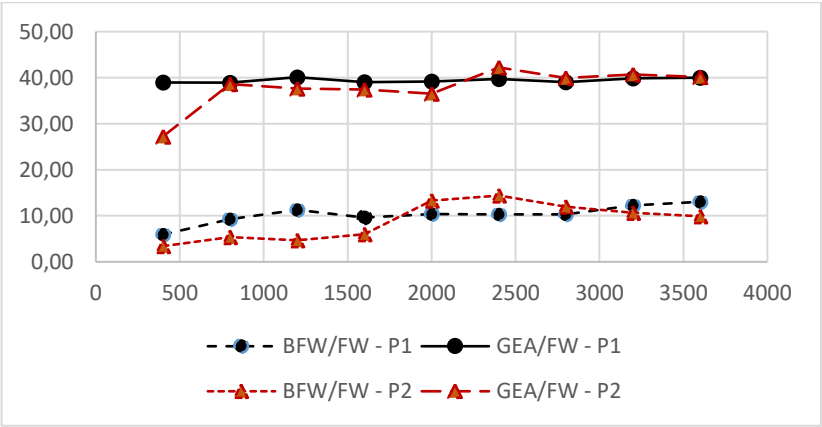

Figure 5. Speedup (%) of GEA (solid) and BFW (long dash) over FW on Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz, and speedup of GEA (dash) and BFW (square dot) over FW on Intel(R) Core(TM) i5-5200U @ 2.70GHz

Conclusion

The Floyd–Warshall algorithm does not use the spatial and temporal reference locality. The blocked Floyd–Warshall algorithm was created to introduce the spatial locality while calculating the matrix of shortest path distances. Our experiments have shown that modern multicore processors and their hierarchical caches do not explore this locality effectively. In the paper we have proposed the technique of inferring and transforming shortest paths algorithms. It extracts in stepwise manner both spatial and temporal reference locality, increases the efficiency of processing big data on modern multi-core systems, and reduces the algorithm run-time dramatically. The technique is capable of inferring algorithms for solving other large-scale problems on state-of-the-art multi-processor systems.

References

[1] Denning, P.J. The Locality Principle. *Communications of the ACM*, Volume 48, Issue 7, (2005), Pages 19–24.

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

[3] Prihozhy A.A. Optimization of data allocation in hierarchical memory for blocked shortest paths algorithms. System analysis and applied information science. – 2021, No. 3, pages 40–50.

[4] 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.

[5] Wang, L. [et al.]. Floyd-Warshall all-pair shortest path for accurate multi-marker calibration. 2010 IEEE International Symposium on Mixed and Augmented Reality. – Seoul, South Korea: IEEE, 2010, pp. 277–278.

[6] Ridi, L., Torrini, J., Vicario, E. Developing a Scheduler with Difference-Bound Matrices and the Floyd-Warshall Algorithm. IEEE Software. – 2012. – Vol. 29, №. 1. – P. 76–83.

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

[8] Floyd, R.W. Algorithm 97: Shortest path. Communications of the ACM, 1962, 5(6), p.345.

[9] 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.

[10] Pettie, S., , Ramachandran, V. Computing shortest paths with comparisons and additions. Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 2002, pp. 267–276.

[11] Pettie, S. A new approach to all-pairs shortest paths on real-weighted graphs. Theoretical Computer Science. 312 (1), 2004: 47–74.

[12] Seidel, R. On the All Pairs Shortest paths Problem in Unweighted Undirected Graphs. Journal of Computer and System Sciences. 51 (3), 1995, pp. 400-403.

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

[14]Park, J.S., Penner, M., and Prasanna, V.K. Optimizing graph algorithms for improved cache performance. IEEE Trans. on Parallel and Distributed Systems, 2004, 15(9), pp.769-782.

[15] Albalawi, E., Thulasiraman, P., Thulasiram, R. Task Level Parallelization of All Pair Shortest Path Algorithm in OpenMP 3.0. 2nd International Conference on Advances in Computer Science and Engineering (CSE 2013), 2013, Los Angeles, CA, July 1-2, 2013, pp. 109-112.

[16] Tang, P. Rapid Development of Parallel Blocked All-Pairs Shortest Paths Code for Multi-Core Computers. IEEE SOUTHEASTCON 2014, pp. 1-7.

[17] Solomonik, E., Buluc, A., and Demmel, J. Minimizing Communication in All Pairs Shortest Paths IEEE 27th International Symposium on Parallel & Distributed Processing, 2013, pp.548-559.

[18] Singh, A., Mishra, P.K. Performance Analysis of Floyd Warshall Algorithm vs Rectangular Algorithm. International Journal of Computer Applications, Vol.107, No.16, 2014, pp. 23-27.

[19] Madduri, K., Bader, D., Berry, J.W., Crobak, J.R. An Experimental Study of a Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances / K Madduri, // Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX), 2007, pp.23-35.

[20] Прыхожы, А. А., Карасік, А. М. Кааператыўныя блочна-паралельныя алгарытмы рашэння задач на шмат'ядравых сістэмах. Системный анализ и прикладная информатика. – 2015. – № 2. – С. 10–18.

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

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

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

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

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

[26] Карасик, О. Н., Прихожий, А. А. Усовершенствованный планировщик кооперативного выполнения потоков на многоядерной системе. Системный анализ и прикладная математика. – 2017. – № 1. – С. 4–11.

[27] Prihozhy, A.A. Asynchronous scheduling and allocation. Proceedings Design, Automation and Test in Europe. Paris, France. – IEEE, 1998, pp. 963-964.

[28] Prihozhy, A.A. Analysis, transformation and optimization for high performance parallel computing. Minsk: BNTU, 2019. – 229 p.

ВЫВОД АЛГОРИТМОВ ПОИСКА КРАТЧАЙШИХ ПУТЕЙ С ВРЕМЕННОЙ И ПРОСТРАНСТВЕННОЙ ЛОКАЛЬНОСТЬЮ ДЛЯ ОБРАБОТКИ БОЛЬШИХ ДАННЫХ

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

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

Беларуский национальный технический университет, Беларусь
ИССофт Солюшенс (часть Кохерент Солюшенс), Беларусь
E-mail: prihozhy@yahoo.com, karasik.oleg.nikolaevich@gmail.com

Аннотация. *Задача о поиске кратчайших путей между всеми парами вершин графа большого размера имеет множество важных прикладных областей в науке, технике и экономике. В таких компьютерных архитектурах, как многоядерные системы, используется иерархическая память, состоящая из локальных и разделяемых уровней, которые различаются объемом и временными задержками передачи данных. Ядра читают и записывают данные через быстрые локальные кэши, поэтому алгоритмы, поддерживающие локальность при обработке больших данных, наиболее эффективны. В статье разрабатывается метод формального вывода, направленный на создание алгоритмов поиска кратчайших путей, которые улучшают пространственную и временную локальность ссылок и снижают нагрузку на кэш. Предлагается и трансформируется алгоритм поиска кратчайших путей, построенный на основе расширения графа, который обладает свойствами локальности ссылок и пересчитывает длины кратчайших путей при каждом добавлении вершины к графу. Каждый шаг преобразования алгоритма вносит дополнительную временную или пространственную локальность. Вычислительные эксперименты, проведенные на двух типах многоядерных процессоров и на графах из тысяч вершин, показали примерно 40 % повышение производительности предложенного алгоритма по сравнению с классическим алгоритмом Флойда-Уоршалла. Предложенный алгоритм также показал выигрыш в 25–35 % по сравнению с блочным алгоритмом Флойда-Уоршалла, обладающим свойством пространственной локальности данных.*

Ключевые слова: *многоядерный процессор; иерархическая память; кэш; алгоритм поиска кратчайших путей; большие данные; пространственная локальность; временная локальность; преобразование алгоритма; формальный вывод.*