TWMS J. Pure Appl. Math., V.8, N.1, 2017, pp.12-21

RESEARCH AND DEVELOPMENT OF JOHNSON'S ALGORITHM PARALLEL SCHEMES IN GPGPU TECHNOLOGY

S.D. POGORILYY[1], M.S. SLYNKO[1], Y.I. RUSTAMOV[2]

Abstract. The use of Johnson's algorithm for finding the shortest path for all pairs of weighted directed graph nodes is proposed. Its formalization in terms of Glushkov's modified systems of algorithmic algebras is realized. The expediency of GPGPU technology using to accelerate the algorithm is proved. A number of schemas of parallel algorithm optimized for using in GPGPU are obtained. Approach to the implementation of the schemes obtained with the use of computing architecture NVIDIA CUDA is proposed. An experimental study of improved performance by using GPU for computations is realized.

Keywords: NVIDIA CUDA, GPGPU, SSSP, APSP, Thrust, SAA Scheme, Johnson's Algorithm.

AMS Subject Classification: 68Q85.

1. Introduction

Many graph theory applications exist in the field of network analysis. Packet routing in computer networks is one of the most important and most relevant application areas of the all-pairs shortest paths (APSP) problem [4]. Graph theory is also used in chemistry and molecular biology for modeling tasks. Graph is a convenient model for solving such problems as the VLSI circuit design, phylogenetics, data mining, bioinformatics, network analysis etc. The article considers the all-pairs shortest paths problem, which requires large computational resources. For example, the Floyd-Warshall algorithm (1962) complexity is $O(|V^3|)$, given $|V|$– a number of graph vertices. Different algorithms offer different approaches to solve the problem: for instance, the Dantzig algorithm is based on iterative calculations of shortest paths submatrices of increasing dimensions, using recurrent procedure. However, the most obvious way is to iteratively apply the single source shortest paths (SSSP) algorithm to each graph vertex. The article considers Johnson's algorithm ([6]), which is based on the iterative use of the Dijkstra algorithm. Formalization schemes are given in terms of the system of algorithmic algebras (SAA).

Modern NVIDIA GPU architectures Graphic adapters were originally used for image processing only. One of the main features of modern graphic processing units is the availability of a set of streaming multiprocessors (SM), which are universal data processing units. This allows using the GPU computational resources for solving a wide range of non-graphic related problems (GPGPU technology – General Purpose GPU).

CUDA (Computed Unified Device Architecture) is a parallel computing platform and programming model invented by NVIDIA, which simplifies GPGPU programming by using high-level API. CUDA programming model defines CPU as "host", and GPU – as "device" [9].

[1]Taras Shevchenko University of Kiev, Faculty Radiphisics, Ukraine
[2]Institute of Control Systems, Baku, Azerbaijan
e-mail: sdp@rpd.univ.kiev.ua, viteldmitry@gmail.com, terlan56@mail.ru

Manuscript received September 2016.

In fact, CUDA device is a multi-core processor used for computing tasks, which can only be engaged from the CPU. According to Flynn's taxonomy, CUDA device is close to the SIMD classification (Single Instruction, Multiple Data). It should however be noted, that each thread may not only execute the same instruction over the dedicated data subset, but also may have a different execution path according to specified conditions. CUDA developers call such approach a SIMT (Single Instruction, Multiple Threads), meaning that a single instruction is executed simultaneously by many threads, but without any limitations on behavior of a particular thread. Thus, the main computing unit from a software perspective is a thread. A set of concurrent threads that can communicate using shared memory and barrier synchronization forms a block. A set of blocks that execute the same kernel function form a grid [3, 9].

CUDA defines a $C/C++$ language extension, which allows writing the code to be run on the GPU, by defining kernel functions. Each thread can be uniquely identified by a number of context variables:

1. gridDim – vector variable that stores the grid dimensions
2. blockIdx – vector variable that stores a 3-dimensional index of a block in a grid
3. blockDim – vector variable that stores the block dimensions
4. threadIdx – vector variable that stores a 3-dimensional index of a thread in a block
5. warpSize – variable that stores size of a "warp" group [7].

For example, the following kernel function adds two numbers

```
    __global__ void add(int* a, int* b, int* c){
     *c = *a + *b;
}
    int main(void){
     ... //initialization of variables, GPU memory
allocation
     add<<<1, 1>>>(a, b, c); // execute 1-threaded, 1-
block kernel function at GPU
     return 0;
    }
```

2. The system of algorithmic algebras

Victor Glushkov developed a mathematical apparatus of algorithmic (microprogram) algebras (SAA) in order to create formalized representations of algorithms which describe the operations of the computer abstract model. Particular SAA is a dibasic algebraic system, which is based on a set of operators and a set of conditions [1]. SAA operations are divided into logical and operator categories. Logical include common Boolean operations and operation of a left multiplication of operator A to condition α (designed for computation process prediction), while operator category includes basic blocks of structured programming (sequential execution, cyclical execution etc.).

Glushkov theorem. For any algorithm a SAA exists (not unique in general case), where the algorithm can be represented by a formalized scheme. Thus, if one defines the SAA basics for the particular algorithm, this algorithm can be represented by the scheme, and further transformations and optimization could be performed on the scheme, not on the algorithm itself. Some basic operations of algorithmic algebra and their counterparts in procedural programming languages are listed in Table 1, 2

Table 1. SAA operation syntax.

SAA operation syntax	**Appropriate Pascal operator**
Composition: * (or $A \times B$)	A;B;
α-disjunction ${}_{\alpha}(A \vee B)$	if α then A else B;
α-iteration ${}_{\alpha}\{A\}$	while α do A;
Inversed α-iteration $\{A\}_{\alpha}$	repeat A until not α

Table 2. SSAA modified (SAA-M) operation syntax.

Parallel SAA-M operation syntax	**Description**
Asynchronous disjunction $A\,\|\|B$ (or $A \overset{\bullet}{\vee} B$)	Processes operators A and B in parallel using one substructure (subset of the information set) for each operator
Filtration $\underline{\underline{u}}$	Unary operation that can be used for filter operator generations
Synchronous disjunction $A \vee B$	Synchronously processes operators A and B

2.1. **Johnson's algorithm.** The idea of the algorithm is to iteratively apply Dijkstra's algorithm to each of the graph vertices, so one can find the shortest paths between any pair of vertices. Although Dijkstra's algorithm disadvantage is the inability of its application for graphs with negative edge weights, problem can be reduced to the case of non-negative weights, if there are negative weighted edges in a graph, but there are no cycles of negative weight. In this case graph can be re-weighted, replacing initial weight function w with a new weight function $\hat{w}$ such that:

1. Shortest paths remain the same: for every pair of vertices $u, v \in V$ the shortest path from u to v in terms of weight function w is also the shortest path in terms of weight function $\hat{w}$ and vice versa.

2. Weights of all edges in a graph are non-negative in terms of a new weight function ($\hat{w}$) [6].

Let's introduce the following notations:
$G(V, E)$ – source graph;
V – initial graph dimension;
$isEdge(i, j)$ – predicate whose value will be true if there exists an edge from to j;
d_{ij} – length of the shortest path found from to j;
w_{ij} – weight of an edge from to j;
$\{h_i\}, i = \overline{0, V+1}$ – set of the shortest paths from the V-th vertex to all vertices of a graph;
$setWay(i, j, val)$ – operator which sets the value (val) of the d_{ij} path;
$setEdge(i, j, val)$ – operator which sets the edge value (val) from i-th vertex to j-th;
$add(Q, i)$ – operator which adds element "i'' to collection Q;
$erase(Q, i)$ – operator which removes element "i'' from collection Q;
$eraseAll(Q, u)$ – operator which removes all elements from collection Q;
$size(Q)$ – operator which returns size of a collection Q;

Thus the first step of the algorithm is creation of a new graph $G' = (V', E')$, such that $V' = V \bigcup \{s\}$, where s is a new node. This includes creation of the edges $\{(s, v) : v \in V \bigcup w_{sv} = 0\}$. Let's introduce the following operators to formalize the considered algorithm:

extendGraph operator extends the given graph with a new node (with serial number V), adding edges with zero weight from a new node to all existing nodes:

$$extendGraph = {}_{i<V}\{setEdge(V, i, 0)\}. \tag{1}$$

init operator performs initialization prior to the Dijkstra and Bellman-Ford algorithms execution, setting the shortest distances from the starting node q to all other nodes except q, as ∞ :

$$init(q) = {}_{i<V}\{{}_{i=q}(setWay(q,i,0) \vee setWay(q,i,\infty)) \times (++i)\}. \tag{2}$$

RelaxEdge operator carries relaxation of the edge from node u to node v. Relaxation condition is the minimum distance from the starting node q to currently processed nodes u and v:

$$RelaxEdge(q,u,v) = {}_{isEdge(u,v)}({}_{d_{uv}>d_{qu}+w_{uv}}(setWay(q,v,d_{qu}+w_{uv}) \vee E) \vee E). \tag{3}$$

RelaxEdges operator carries relaxation of all edges in a graph:

$$RelaxEdges(q) = {}_{u<V}\{{}_{v<V}\{RelaxEdge(q,u,v) \times (++v)\} \times (++u)\}. \tag{4}$$

2.2. **Bellman-Ford algorithm.** For correct execution of the Johnson's algorithm, input graph must not have negative-weight cycles. Bellman-Ford algorithm is executed for the new node "s" in order to check for such cycles. An array of shortest paths from vertex s to all other vertices, obtained as a result of the Bellman-Ford algorithm execution (which will contain only non-negative values), allows transformation to the new weight function $\hat{w}$.

According to the algorithm definition, all d_{uv} are set after $(V-1)$ relaxation cycles. Therefore, satisfaction of the relaxation condition after $(V-1)$ relaxation cycles means having a negative-weight cycle in the input graph [6], which is checked by the *cnc* condition:

$$cnc(q) = \overset{u<V}{\underset{u=1}{\wedge}} \overset{v<V}{\underset{v=1}{\wedge}} (isEdge(u,v) \wedge d_{qv} < d_{qu} + w_{uv}). \tag{5}$$

So we retrieve the following SAA scheme for the Bellman-Ford algorithm with a starting node q:

$$BellmanFord(q) = init(q) \times {}_{i<V-1}\{RelaxEdges(q) \times (++i)\} \times cnc(q). \tag{6}$$

The result of the Bellman-Ford algorithm execution is a Boolean value that indicates the presence of negative weight cycles in a graph. Also the algorithm determines the shortest paths from the additional vertex q to all existing vertices (i.e., sets $\{h_i\}$values).

If the graph has no negative weight cycles, it can be re-weighted using the next operator:

$$UpdateWeights = {}_{i<V+1}\{{}_{j<V+1}\{{}_{isEdge(i,j)}(w_{ij} = (w_{ij} + h_i - h_j) \vee E) \times (++j)\} \times (++i)\}. \tag{7}$$

2.3. **Dijkstra's algorithm.** Dijkstra's algorithm is essentially serial and finds the shortest path from a given vertex to all other vertices in a graph having the $O(V \log V + E)$complexity. Algorithm distinguishes two types of vertices: settled and unsettled. Node is considered as settled if it is reachable from the starting node, and all outgoing edges either are being relaxed during the current iteration or have been relaxed before. At the beginning of the algorithm starting node is added to the list of settled nodes and its outgoing edges are being relaxed. Then the node u with the lowest tentative distance from the starting node is taken as the new frontier node, its outgoing edges are being relaxed and the node itself becomes considered as settled. The algorithm finishes when all nodes are settled [6].

Let's introduce additional notations for retrieving the regular scheme of a sequential Dijkstra's algorithm:

Q – set of the unsettled vertices;

$ExtractMin(Q,q,u)$ – operator which finds a new unsettled frontier node u, depending a minimum tentative distance (d_{qu}) from a starting node:

$$\begin{aligned} &ExtractMin(Q,q,u) = (mv = d_{q,Q_0}) \times (mi = 0) \times {}_{i \in Q}\{(v = Q_i) \times {}_{mv<d_{qv}}(mv = d_{qv} \times \\ &\times (mi = i) \times (++i) \vee E) \times (v = Q_i) \times erase(Q, mi)\}. \end{aligned} \tag{8}$$

As a result, we retrieve the following scheme of Dijkstra's algorithm:

$$SubD1(Q) = ExtractMin(Q, q, u) \times {}_{v<V}\{RelaxEdge(q, u, v) \times (++v)\},$$

$$D(q) = init(q) \times {}_{Q\neq\emptyset}\{SubD1(Q)\}. \tag{9}$$

The result of Dijkstra's algorithm provides an array of shortest paths from one vertex to all others in terms of the new weight function. We should perform reverse transformation to obtain the shortest paths in terms of original weight function:

$$to(D, i) = {}_{j<V}\{setWay(i, j, D_j + h_j - h_i) \times (++j)\}, \tag{10}$$

where $\{D_j\}, j = \overline{0, V}$ is a set of shortest paths from the node i to all other nodes (obtained via Dijkstra's algorithm).

Using the introduced notations, the serial Johnson's algorithm scheme was created:

$$\alpha = BellmanFord(V),$$

$$SubJohns1(V) = {}_{i<V}\{to(D(i), i) \times (++i)\},$$

$$SubJohns2(V) = {}_{\alpha}(UpdateWeights \times SubJohns1(V) \vee E),$$

$$Johnson = extendGraph(V) \times SubJohns2(V). \tag{11}$$

In terms of optimization and acceleration algorithms, the emphasis should be placed on accelerating the Dijkstra's algorithm, because of the majority of calculations, executed in it.

3. Developing SAA scheme for parallel version of algorithm

Classic approach to APSP algorithms parallelization is based on parallel execution of several SSSP algorithm iterations. Such approach is efficient if all the calculations are performed on the CPU. When using GPGPU technology and CUDA model, the level of branch divergence (difference of execution paths of different threads in the same warp) greatly affects performance of the resulting application. Despite the fact that according to SIMT principle, different threads may have different execution paths, the warp serially executes each branch path taken, disabling threads that are not on that path. CPU architecture developers put a lot of effort to create countervailing measures, including speculative execution and branch prediction. However, the majority of such optimizations haven't been applied to the GPU architectures, so the use of large, highly divergent branches may lead to significant performance loss [8].

The paper presents parallel schemes, based on parallel execution of calculations within SSSP iterations. This makes it possible to reduce the level of branch divergence to minimum.

3.1. Dijkstra's algorithm parallel schemes.

Dijkstra's algorithm consists of two cycles-outer selects a new, yet unsettled, node on each iteration, while inner cycle actually performs relaxation of all outgoing edges of the previously selected node. Outer cycle parallelization means that more than one vertex, which can be settled without affecting the validity of the results, are chosen on each iteration. Inner cycle parallelization means that multiple outgoing edges of the same vertex are relaxed in parallel.

3.2. **Outer cycle parallelization.** At each iteration Dijkstra's algorithm finds a node u such that is not settled yet and which has the smallest tentative distance to the initial vertex (d_{qu}). Selecting more nodes at this step means achieving higher level of parallelism.

Thus, if the serial version selected a single node u (with minimal d_{qu}), parallel version selects u as a set of vertices v, such that each of them satisfies condition:

$$d_{qv} \leq d_{qu} + \Delta, \tag{12}$$

where Δ is a minimum weight associated to any edge of the graph [8]. Such option was chosen to save memory and computational resources by a small performance reduction. For further performance increase (due to additional memory allocation), one may replace Δ with Δ_v, where Δ_v is the minimum weight of edges, outgoing from v node. This change will provide an opportunity to adjust the acceptability criteria for each node separately, which will increase the level of parallelism. The next schema is retrieved after transformation applied to the (8) operator:

$$\begin{aligned}&ExtractMin(Q,q,u) = ExtractLimit(Q,q,m) \times {}_{i<size(Q)}\{{}_{d_{ql}<d_{qm}+\Delta}(add(u,l) \vee E)\times \\ &\times(++i)\} \times eraseAll(Q,u).\end{aligned} \tag{13}$$

ExtractLimit – is an operator, which finds the node with a minimum tentative distance from the starting one (similar to (8) for the serial version, but without removing the node found from Q). Therefore we transform the (13) scheme, taking into account that each vertex from Q can be processed in parallel, using the $B_Q(i)$ barrier, which delays further thread execution until the specified number of threads reach the synchronization point (internal barrier formalization is given in [5]):

$$\begin{aligned}&PExtractMin(Q,q,u) = ExtractLimit(Q,q,m)\times \\ &\times \mathop{\overset{\bullet}{\vee}}\limits_{i=1}^{size(Q)} ({}_{d_{qi}<d_{qm}+\Delta}(add(u,l) \vee E) \times B_Q(size(Q))) \times eraseAll(Q,u).\end{aligned} \tag{14}$$

Thus, Dijkstra's algorithm with outer loop parallelization processes multiple vertices at once, relaxing outgoing edges in parallel. After this, threads are synchronized by a barrier, algorithm defines a new set of nodes to be processed and the next iteration begins. As a result, the following scheme is obtained:

$$PSubD1(u) = \mathop{\overset{\bullet}{\vee}}\limits_{l=1}^{size(u)} ({}_{v<V}(RelaxEdge(q,u_l,v) \times (++v)\} \times B_u(size(u))),$$

$$Dijkstra(q) = init(q) \times {}_{Q\neq\emptyset}\{PExtractMin(Q,q,u) \times PSubD1(u)\}. \tag{15}$$

In this scheme the number of threads equals to the number of vertices in the graph. General algorithm flow is given below:

```
thrust::device_vector<int> Q(V);
INIT<<<blocks, threads>>>(Q, M, C, q);
while(Q.size() != 0)
{
 RELAX_WITH_MASK<<<blocks, threads>>>(G, M, C);
 threshold = ExtractMin(Q, C);
 UPDATE_MASK<<<blocks, threads>>>(Q, C, M, threshold);
}
```

At each iteration relaxation is performed with a certain mask, so not all the vertices have their outgoing edges relaxed simultaneously:

```
    RELAX_WITH_MASK(G, M, C)
    {
     tid = threadx.Id;
     if(M[tid] == true)
     {
          <execute for all edges [edgeId], outgoing from
the tid node>                    atomicMin(C[edgeId],
G[edgeId] + C[tid]);
     }
    }
```

As described above, node is considered as settled and processed, if its tentative distance to the starting node is lower than a threshold:

```
    UPDATE_MASK(Q, C, M, threshold)
    {
     tid = thread.Id;
     if(C[Q[tid]] <= threshold)
     {
          M[Q[tid]] = true;
          Q[tid] = -1; // '-1' elements will be removed
after execution
     }
    }
```

GPGPU application developers should solve not only algorithmic problem, but also memory bank conflicts, branch divergence optimizations and so on. For complex tasks optimization steps are usually unobvious and degrade the readability and understandability of the code. Therefore CUDA developers have created a template library in C++, similar to STL. *Thrust* solves a number of problems associated with optimizing applications for specific architecture or use different approaches for devices that support different versions of CUDA. Developers can fully concentrate on solving the high-level tasks, delegating choice of particular computing implementation to the *Thrust*.

```
    struct compare_unsettled_value
    {
     const int* data;
     compare_unsettled_value(int* ptr) { data = ptr; }
     __host__ __device__
          bool operator()(int lhs, int rhs){
               return this->data[lhs] < this->data[rhs];

     }
    };

    ExtractMin(Q, C) //Q - device_vector<int>
{
     ...

     thrust::detail::normal_iterator<device_ptr<int>>
minIndex = thrust::min_element(
    thrust::device, Q.begin(), Q.end(),
compare_unsettled_value(C);
     ...
}
```

In the above code block we use the library method *min_element*, which performs array reduction (array boundaries are set via $Q.begin() - Q.end()$ range) using minimum as a binary operation. We introduce *compare_unsettled_value* structure as a permutation iterator, because vector Q stores indexes, not the actual distances. All the calculations are performed on the GPU due to the explicitly set $thrust :: device$ policy [2].

3.3. **Inner cycle paralellization.** As mentioned above, inner cycle parallelization means parallel relaxation of all outgoing edges of the selected node. Thus, operator (13) is left the same as in serial version (8), but we change the logic of the (3) operator: now each thread represents an edge that can be relaxed independently. Parallel edge relaxation is followed by barrier synchronization, after which current node processing is complete and the algorithm selects the node for the next iteration.

$$PSubD2(u) = \overset{\bullet}{\underset{v=1}{\overset{V}{\vee}}}(RelaxEdge(q,u,v) \times B_v(v)),$$

$$Dijkstra(q) = init(q) \times {}_{Q \neq \emptyset}\{ExtractMin(Q,u) \times PSubD2(u)\}. \tag{16}$$

3.4. **Bellman-Ford algorithm parallel scheme.** Since the Bellman-Ford algorithm is used only once during Johnson's algorithm execution, even an ideal parallel scheme will not provide big performance increase. However, we may transform the (4) operator so it processes each node in parallel, receiving the following parallel scheme:

$$RelaxEdges(q) = \overset{\bullet}{\underset{u=1}{\overset{V}{\vee}}}({}_{u<V}\{RelaxEdge(q,u,v) \times (++v) \times B_v(u)\}). \tag{17}$$

Other operators and conditions of the algorithm remain the same.

4. Experimental results

The performance results were obtained using CPU Intel ® Core ^{TM} $i7 - 47703.4HHts$, $8GBDDR3$ RAM and GeForce GTX 770 (Kepler) GPU. Input graph was stored in a form of adjacency matrix.

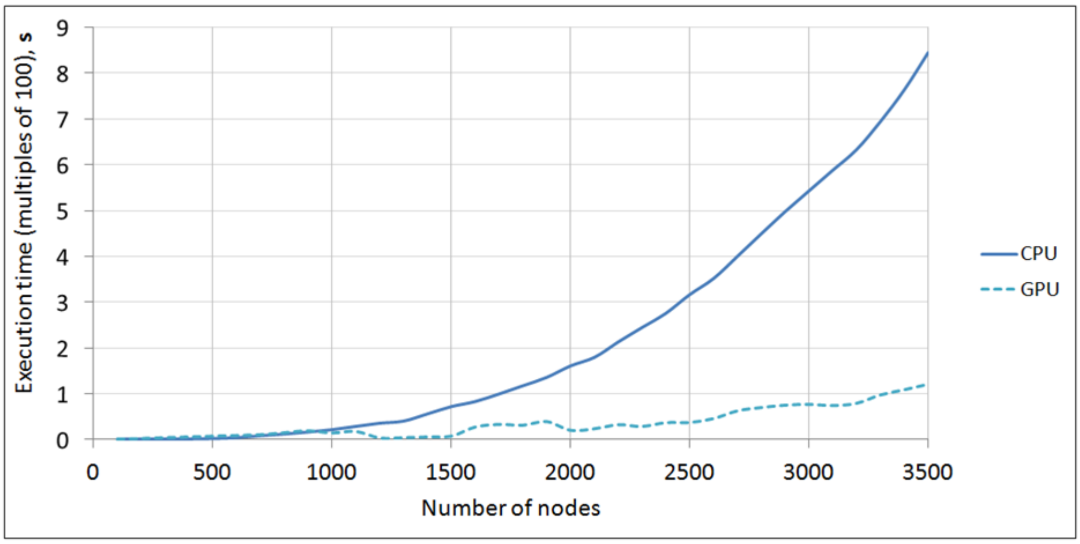


Figure 1. CPU vs. GPU implementation times depending on number of nodes in a graph.

5. Conclusions

Primary optimization technique of Johnson's algorithm is based on improving Dijkstra's algorithm performance while increasing dimension of the input graph by using GPGPU technology. Executing several iterations of Dijkstra's algorithm in parallel is impractical for GPGPU, therefore main principles of its parallelization considered vertex parallelization (each CUDA thread represents a node) and edge parallelization (each CUDA thread represents an edge to be relaxed). The SAA-M schemes usage allows you to abstract from the specific software platform and implementation details, paying more attention to the algorithm scheme itself, and the concepts and methods of its transformation. Advantages of the parallel schemes developed using CUDA model are experimentally confirmed. We have compared our GPU approach with the relevant CPU implementation, obtaining up to 7x speed-up in case of GPU implementation.

References

[1] Anisimov, A.V., Pogorilyy, S.D., Vitel, D.Yu., (2013), About the issue of algorithms formalized design for parallel computer architectures, Appl. and Comput. Mat., 12(2), pp.140-151.

[2] Bell, N., Hoberock, J., (2011), Thrust: A Productivity-Oriented Library for CUDA, [Online]. Available from: info:M6ha1PssSvYJ:scholar.google.com.

[3] Levchenko, R.I., Sudakov, O.O. Pogorilyy, S.D., (2009), DDCI: Simple Dynamic Semiautomatic Parallelizing for Heterogeneous Multicomputer Systems, Proceedings of the 5th IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, Rende Cosenza, Italy.

[4] Pogorilyy, S.D., Boyko, Yu.V., Gusarov, A.D., Lozytskyi, S.I., (2009), An approach to the parallel solution of a high-dimensional basic flow problem, Cybernetics and Systems Analysis, 45(2), pp.291-296. Springer Science and Business Media Inc. ISSN:1060-0396.

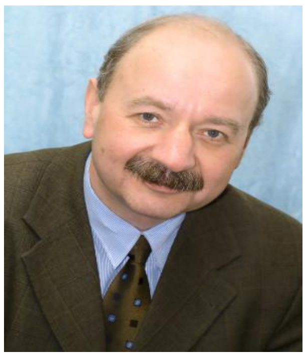
[5] Pogorilyy, S.D., Maryanovsky, V.A., Boyko, Yu.V., Vereshinsky, O.A., (2009), Research of Danzig algorithm parallel schemes for computing systems with shared memory, Mathematical Machines and Systems, V.4.

[6] Pogorilyy, S.D., Shkulipa, I.Yu., (2009), A conception for creating a system of parametric design of parallel algorithms and their software implementations, Cybernetics and System Analysis, 45(6), pp.952-958. Springer Science and Business Media Inc. ISSN:1060-0396.

[7] Pogorilyy, S.D., Vereshchynsky, O.A., (2011), Investigation of the parallel schemes of Danzig algorithm for SMP systems, Journal of Qafqaz University. An International Journal Mathematic and Computer Science, 32, pp.1-11.

[8] Tiaynyi, D.H., Abdelrahman, T.S., (2011), Reducing Branch Divergence in GPU Programs, Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, New York, NY, USA.

[9] Tribrat, M.I., Pogorilyy, S.D., Afanas'ev, D.V., (2011), Approaches to Parallel Implementation Yen's Algorithm Using CUDA, Proceedings of the 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 1, pp.243-246.



Sergiy D. Pogorilyy - Doctor of Sciences, Professor, Head of Computer Engineering Department (Ukraine, Taras Shevchenko National University of Kyiv, Faculty of Radio Physics, Electronics and Computer Systems). He has graduated from Donetsk Technical University (Donetsk, Ukraine) in 1971 and received M.Sc. in Applied Mathematics degree. In 1972-1975 he studied at postgraduate courses at Institute of Cybernetics (Kyiv, Ukraine), Department of Computing Engineering, maintained a thesis and received Ph.D. degree in Computing Sciences. He is an author of 250 articles and 21 books

Maksym S. Slynko - M.Sc. in Computer Engineering (Ukraine, Taras Shevchenko National University of Kyiv, Faculty of Radio Physics, Electronics and Computer Systems) and a senior software engineer. He has graduated from Taras Shevchenko National University of Kyiv in 2014 (bachelor degree, with honors) and received M.Sc. in Computer Engineering in 2016 (with honors). He has been acquiring Ph.D. degree in Computer Sciences since 2016.

Yasin Rustamov - graduated from Azerbaijan State University in 1978. He received his Ph.D. degree from the Institute of Hydrotechnical and melioration (Ukraine) in 1991. Currently, he is a head of the department at the Institute of Control Systems of ANAS. His research interests are in the fields of on probability theory, mathematical statistics, reliability theory, melioration, revegetation and soil protection).