UDC 004.272.2
https://doi.org/10.37661/1816-0301-2023-20-2-65-84

Original Paper
Оригинальная статья

Generation of shortest path search dataflow networks of actors for parallel multi-core implementation

Anatoly A. Prihozhy

Belarusian National Technical University,
av. Nezavisimosty, 65, Minsk, 220013, Belarus
E-mail: prihozhy@yahoo.com

Abstract

Objectives. The problem of parallelizing computations on multicore systems is considered. On the Floyd – Warshall blocked algorithm of shortest paths search in dense graphs of large size, two types of parallelism are compared: fork-join and network dataflow. Using the CAL programming language, a method of developing actors and an algorithm of generating parallel dataflow networks are proposed. The objective is to improve performance of parallel implementations of algorithms which have the property of partial order of computations on multicore processors.

Methods. Methods of graph theory, algorithm theory, parallelization theory and formal language theory are used.

Results. Claims about the possibility of reordering calculations in the blocked Floyd – Warshall algorithm are proved, which make it possible to achieve a greater load of cores during algorithm execution. Based on the claims, a method of constructing actors in the CAL language is developed and an algorithm for automatic generation of dataflow CAL networks for various configurations of block matrices describing the lengths of the shortest paths is proposed. It is proved that the networks have the properties of rate consistency, boundedness, and liveness. In actors running in parallel, the order of execution of actions with asynchronous behavior can change dynamically, resulting in efficient use of caches and increased core load. To implement the new features of actors, networks and the method of their generation, a tunable multi-threaded CAL engine has been developed that implements a static dataflow model of computation with bounded sizes of buffers. From the experimental results obtained on four types of multi-core processors it follows that there is an optimal size of the network matrix of actors for which the performance is maximum, and the size depends on the number of cores and the size of graph.

Conclusion. It has been shown that dataflow networks of actors are an effective means to parallelize computationally intensive algorithms that describe a partial order of computations over decomposed data. The results obtained on the blocked algorithm of shortest paths search prove that the parallelism of dataflow networks gives higher performance of software implementations on multicore processors in comparison with the fork-join parallelism of OpenMP.

Keywords: dataflow, network of actors, CAL language, shortest paths, blocked algorithm, multi-core system, speedup

For citation. Prihozhy A. A. *Generation of shortest path search dataflow networks of actors for parallel multi-core implementation*. Informatika [*Informatics*], 2023, vol. 20, no. 2, pp. 65−84.
https://doi.org/10.37661/1816-0301-2023-20-2-65-84

Conflict of interest. The author declares of no conflict of interest.

Received | Поступила в редакцию 20.02.2023
Accepted | Подписана в печать 21.03.2023
Published | Опубликована 29.06.2023

Генерация потоковых сетей акторов поиска кратчайших путей для параллельной многоядерной реализации

А. А. Прихожий

Белорусский национальный технический университет,
пр. Независимости, 65, Минск, 220013, Беларусь
E-mail: prihozhy@yahoo.com

Аннотация
Цели. Рассматривается задача распараллеливания вычислений на многоядерных системах. Посредством блочного алгоритма Флойда – Уоршалла поиска кратчайших путей на плотных графах большого размера сравниваются два вида параллелизма: разветвление/слияние и сетевой потоковый. С использованием языка программирования CAL разрабатываются метод построения акторов потока данных и алгоритм генерации параллельных сетей акторов. Целью работы является повышение производительности параллельных сетевых реализаций алгоритмов, обладающих свойством частичного порядка вычислений, на многоядерных процессорах.
Методы. Используются методы теории графов, теории алгоритмов, теории распараллеливания, теории формальных языков.
Результаты. Доказаны утверждения о возможности переупорядочивания вычислений в блочном алгоритме Флойда – Уоршалла, способствующие повышению загрузки ядер при реализации алгоритма. На основе утверждений разработан метод построения акторов на языке CAL и предложен алгоритм автоматической генерации CAL-сетей потока данных для различных конфигураций матриц блоков, описывающих длины кратчайших путей. Доказано, что сети обладают свойствами согласованности, ограниченности и живучести. В акторах, работающих параллельно, порядок выполнения действий с асинхронным поведением может динамически меняться, что приводит к эффективному использованию кэшей и увеличению загрузки ядер. Для реализации новых возможностей акторов, сетей и метода их генерации разработан настраиваемый многопоточный CAL-движок, реализующий статическую модель потоковых вычислений с ограниченными размерами буферов. Из экспериментальных результатов, полученных на четырех типах многоядерных процессоров, следует, что существует оптимальный размер сетевой матрицы акторов, для которого производительность максимальна, и этот размер зависит от размера графа и количества ядер.
Заключение. Показано, что сети акторов потока данных являются эффективным средством распараллеливания алгоритмов с высокой вычислительной нагрузкой, описывающих частичный порядок вычислений над данными, декомпозированными на части. Результаты, полученные на блочном алгоритме поиска кратчайших путей, показали, что параллелизм сетей потока данных дает более высокую производительность программных реализаций на многоядерных процессорах по сравнению с параллелизмом разветвления/слияния стандарта OpenMP.

Ключевые слова: поток данных, сеть акторов, язык CAL, кратчайшие пути, блочный алгоритм, многоядерная система, ускорение

Для цитирования. Прихожий, А. А. Генерация потоковых сетей акторов поиска кратчайших путей для параллельной многоядерной реализации / А. А. Прихожий // Информатика. – 2023. − Т. 20, № 2. – С. 65–84. https://doi.org/10.37661/1816-0301-2023-20-2-65-84

Конфликт интересов. Автор заявляет об отсутствии конфликта интересов.

Introduction. The problem of finding the shortest and longest paths in weighted graphs [1–5] has many practical applications: computer games, signal processing, city and network traffic, video compression, microelectronics, optimization of computer systems and networks, task scheduling, bioinformatics, and many others. It is formulated in different settings and, therefore, is solved by algorithms of different computational complexity, from polynomial to exponential. In this paper we consider the all-pairs shortest path problem and the blocked Floyd – Warshall algorithm (*BFW*) [6–10] which decomposes the dense graph into subgraphs, has cubic computational complexity and is a basic one for the problem. *BFW* helps to 1) localize the data accesses within blocks and thereby reduce the

data miss count in the processor hierarchical memory; 2) organize the parallel computation of blocks on a multi-processor system. At the same time, *BFW* has drawbacks of recalculating all blocks in every iteration of the loop along graph vertices and of parallelizing the block calculations in the fork-join style, thus providing insufficient load of processors. Usually, *BFW* is implemented with OpenMP. Although the *BFW*'s complexity is polynomial, to handle large graphs *BFW* requires huge computational resources and much runtime even on multiprocessor systems. Scientific research was done, and works were published which improve the properties of *BFW*. Thus, [11] extended the homogeneous blocked Floyd – Warshall algorithm to a heterogeneous one recognizing four types of blocks and speeding up their computation. In [12], a threaded block-parallel algorithm is proposed which uses a cooperative scheduler of threads incorporated in the operating system. Work [13] aims for selecting the optimal size of block. Methods of efficient utilization of hierarchical caches are proposed in [14, 15]. A generalization of blocked Floyd – Warshall algorithm is proposed in [16] aiming at reducing the usage of slow global memory in implementations on GPU.

Parallel dataflow networks [17–20] have not been used yet for the realization of *BFW*; this topic is the subject of the paper. The dataflow actor concept aims at modelling of distributed knowledge-based algorithms. Actors match to the heterogeneous and concurrent dataflow nature of various kinds of embedded systems. The CAL dataflow language is suitable to model applications from cryptography, multimedia processing, network processing, control systems, reconfigurable systems, power optimization, monitoring of hardware and software, and others. Both hardware- and software-oriented CAL-compilers were developed. The concept of actors and principles of concurrency and asynchrony lie in the basis of CAL. Although CAL is a general-purpose actor-programming language, it was most successfully used in the MPEG standard known as reconfigurable video coding. The property of reconfigurability was introduced in CAL due to works [21–24]. In [21], the authors developed the multidimensional synchronous dataflow. In [22], the authors proposed the parameterized dataflow and used it for the reconfiguration of digital signal processing systems. Work [23] defined OpenDF as a dataflow toolset for reconfigurable hardware and multicore systems. The authors of [24] proposed the Boolean parametric data flow as a means for run-time reconfiguration of CAL programs. Since CAL aims for the creation of streaming applications, the authors of works [25–27] developed methods and tools for the synthesis and optimization of dataflow pipelines.

Although this paper develops and implements CAL-networks for parallel solving the all-pairs shortest path problem on multi-core systems, it shows the way of how actors and dataflow networks of actors can be created, generated, and implemented targeting other computationally heavy problems of large sizes with partial order of computations. The main contributions of the paper are:

1. It proves that in parallel *BFW* the block calculations can be moved across iterations of the loop along graph vertices and reordered, thus balancing the computational load among processors.

2. By means of simulation it is shown that the reordering of block calculations can increase the *BFW* speedup up to 25 %.

3. The approach has been developed which extends the reconfigurability principal and yields a method of automatic generation of CAL-actors and dataflow CAL-networks for various block-matrix configurations of shortest paths lengths.

4. Based on the C/C++ language, the generation tool and tunable multi-threaded CAL-engine are developed which create and implement the dataflow networks of actors on multi-core systems.

5. The computational experiments have shown that the optimal size of the block-matrix and CAL-network can be found which depends on the core count and graph size; the CAL-networks give the speedup that is up to 28 % higher than the number of cores and is higher than the speedup OpenMP gives.

CAL dataflow actor language. Many modern forms of computation are very well suited for data flow description and implementation. CAL is the high-level dataflow actor programming language [20–24] in which a program is defined as a network of actors that interact and communicate by sending and receiving data (tokens) along data lossless and order preserving communication channels. An actor is a computational entity that consists of input and output ports, state variables, actions, and a scheduler. Actors run in parallel. When an actor is fired, it consumes tokens from input ports, changes the internal state and produces tokens on output ports. The action is a piece of computation that an

actor performs during firing. An actor may contain any number of actions. When an actor is fired, it selects one of them based on the availability of input tokens and optionally based on conditions relating to the values of tokens and state variables. An action guard enables conditional action firing. A finite state machine (FSM) allows actions to be scheduled according to the current state of the actor and considering action priorities. CAL enables the description of different, but still actor-like, contexts, which have different kinds of objects (and types), different libraries, different primitive data objects and operators. The CAL model has the properties of strong encapsulation, explicit concurrency, and asynchrony (untimedness).

CAL as a domain-specific language provides useful abstractions for parallelizing computations and dataflow programming. It has been shown that the CAL dataflow networks offer a representation that can effectively support the tasks of parallelization and vectorization – thus providing a practical means of supporting multiprocessor systems and utilizing vector instructions. CAL has been used in a wide variety of applications and has been compiled to hardware and software implementations. It has been chosen by the ISO/IEC standardization organization in the MPEG standard called Reconfigurable Video Coding (RVC) (ISO/IEC 23001-4 and 23002-4).

The model of computation [17, 18] defines the semantics of the communication between the actors. It also defines which scheduling policies can be used to fire actors. There exists a variety of models of computation for CAL, which make trade-offs between expressiveness and analysability. The set of recognized dataflow models which are scheduled statically by compiler include Kahn process networks [17], synchronous dataflow networks (SDF) [18], parameterized synchronous dataflow (PSDF) [23], Boolean parametric dataflow (BPDF) [24], multidimensional synchronous dataflow (MDSDF) [22]. Other dataflow networks require dynamical scheduling, which induces a run-time overhead. The Kahn network is a group of deterministic sequential processes that communicate through unbounded FIFO channels. In SDF, the number of tokens read and written by each process is known ahead of time, and the channels have bounded FIFOs. SDF is divided into synchronous sub-networks connected by asynchronous links. PSDF supports dynamic reconfigurability and design reuse, but it does not allow the topology of the dataflow graph to change at runtime. BPDF allows restricted dynamic changes of the graph topology by disabling edges annotated with Boolean expressions.

To be scheduled statically, the dataflow network must have a basic iteration and have the properties of rate consistency, boundedness, and liveness [17–24]. The number of tokens consumed or produced at a given port at each firing is called the rate. The rate consistency of a dataflow network is checked by generating a system of balance equations, which must have a non-null solution for all possible values of parameters. The boundedness is guaranteed if the network returns to its initial state after each iteration. The network liveness is checked by finding a schedule for a basic iteration.

The CAL was first used on the Ptolemy II platform [20]. The complete OpenDF framework has been developed for simulating CAL networks and for generating hardware and software code [21]. The portable CAL interpreter used in the Moses project aimed for simulating a hierarchical network of actors. OpenDF is a compilation framework using a source-to-source compiler. Backends that generate VHDL/Verilog and C for integration with SystemC were developed.

A problem of automatic generation of dataflow networks. Nowadays, dynamic reconfigurable embedded systems [21, 22] are widely used, since they have the capability to modify their functionalities by adding or removing components, and by modifying interconnections among them. The basic idea behind these systems is to autonomously modify its functionalities according to the application’s changes. Dynamic reconfiguration is the process of adding, deleting, or moving resources within the network configuration without deactivating the affected node. The models, architectures, and design methodologies of the reconfigurable systems have been developed. The PSDF approach [23] can dynamically reconfigure the behaviour of dataflow actors, edges, graphs, and subsystems by run-time modification of parameter values. It permits the parameter reconfiguration that does not change the subsystem interface behavior. BPDF [24] combines both the token-rate and topology reconfigurations, although, it does not reconfigure the topology significantly. The dataflow programming models are well-suited to program many-core streaming applications.

There are a variety of application problems where it is difficult or impossible to create a reconfigurable dataflow network; therefore, different dataflow networks must be generated depending on the problem parameters, problem size, and problem formulation. The networks can differ by the actors, input and output ports, actions, etc, and their quantity. In the paper, we consider such a problem, i.e., the all-pairs shortest path search in large graphs to be solved on a multi-processor system. By modifying the blocked Floyd – Warshall algorithm [6, 7], we create and generate dataflow parallel CAL-networks automatically and implement them efficiently on multi-core systems by means of creating a CAL-language-based multithreaded engine.

Block-parallel all-pairs shortest path algorithm. Let $G = (V, E)$ be a simple directed graph with real edge-weights consisting of a set V, $|V| = N$, of vertices and a set E of edges. Let W be the cost adjacency matrix for G. So $w_{i,i} = 0$, $1 \le i \le N$; $w_{i,j}$ is the cost (weight) of edge (i, j) if $(i, j) \in E$ and $w_{i,j} = \infty$ if $i \ne j$ and $(i, j) \notin E$. When G has no cycle with negative sum of weights, the dynamic programming Floyd – Warshall (*FW*) Algorithm 1 [1, 2] computes a series of distance matrices $D^0 \ldots D^k \ldots D^N$ such that $D^0 = W$ and each element $d^k_{i,j}$ of matrix D^k, $k = 1 \ldots N$, is the length of the shortest path from i to j composed of the subset of vertices labelled 1 to k.

The authors of [6, 7, 11, 13] proposed a blocked version *BFW* of the Floyd – Warshall Algorithm 2. *BFW* divides set V of vertices into subsets $V_0 \ldots V_{M-1}$ of size S and splits matrix D into blocks of size $S \times S$ each, creating a block-matrix $B[M \times M]$, where equality $M \cdot S = N$ holds. Algorithm 2 performs M iterations, each consisting of three phases: calculation of diagonal $D0$ block $B_{m,m}$ (accounts for paths inside the subgraph on subset V_m of vertices); calculation of $(M - 1)$ cross blocks $B_{v,m}$ of type $C1$ through block $B_{m,m}$ (accounts for paths from vertices of V_v to vertices of V_m); calculation of $(M - 1)$ cross blocks $B_{m,v}$ of type $C2$ through block $B_{m,m}$ (accounts for paths from vertices of V_m to vertices of V_v); calculation of $(M - 1)^2$ peripheral $P3$ blocks $B_{v,u}$ through blocks $B_{v,m}$ and $B_{m,u}$ (accounts for paths from vertices of V_v to vertices of V_u passing through vertices of V_m). In $B^m_{v,u}$, index m describes the block calculation level. Algorithm 3 (*BCA*) calculates all three types of blocks. In [7], the authors shown that *BFW* can be parallelized to *PBFW* due to all cross blocks can be calculated mutually in parallel as well as all peripheral blocks. Algorithm 2 describes the parallelism by means of OpenMP directives. In *BFW*, the blocks can be also calculated recursively [7].

Algorithm1: Floyd – Warshall *FW*

Input: A number N of graph vertices
Input: An edge cost matrix $W[N \times N]$
Output: Matrix D^N of distances

```
D^0 ← W
for k ← 0 to N – 1 do
    for i ← 0 to N – 1 do
        for j ← 0 to N – 1 do
            d^{k+1}_{i,j} ← min(d^k_{i,j}, d^k_{i,k} + d^k_{k,j})
return D^N
```

Algorithm 3: Block calculation *BCA*

Input: A size S of block
Input: Blocks E, F and H
Output: Block B

```
For k ← 0 to S – 1 do
    For i ← 0 to S – 1 do
        for j ← 0 to S – 1 do
            b_{i,j} ← min(e_{i,j}, f_{i,k} + h_{k,j})
return B
```

Algorithm 2: Block-parallel Floyd – Warshall *PBFW*

Input: A number N of graph vertices
Input: A matrix $W[N \times N]$ of graph edge weights
Input: A size S of block
Output: A blocked matrix $B^M[M \times M]$ of path distances

```
M ← N / S    B^0 ← W
#pragma omp parallel
for m ← 0 to M – 1 do
    #pragma omp single
    B^{m+1}_{m,m} ← BCA (B^m_{m,m}, B^m_{m,m}, B^m_{m,m})          // Diagonal D0
    for v ← 0 to M – 1 do
        if v≠m then
            #pragma omp task united
            B^{m+1}_{v,m} ← BCA (B^m_{v,m}, B^m_{v,m}, B^{m+1}_{m,m})   // Cross C1
            #pragma omp task united
            B^{m+1}_{m,v} ← BCA (B^m_{m,v}, B^{m+1}_{m,m}, B^m_{m,v})   // Cross C2
    #pragma omp task wait
    for v ← 0 to M – 1 do
        if v ≠ m then
            for u ← 0 to M – 1 do
                if u ≠ m then
                    #pragma omp task united
                    B^{m+1}_{v,u} ← BCA (B^m_{v,u}, B^{m+1}_{v,m}, B^{m+1}_{m,u})  // Peripheral P3
    #pragma omp task wait
return B^M
```

Fork-join parallelization potential. OpenMP parallelizes *PBFW* in the fork-join style. Since diagonal block *D*0 is computed in series to all parallel cross blocks *C*1 and *C*2, and all cross blocks are computed in series to all parallel blocks *P*3, (1) estimates the speedup the *PBFW* provides over *BFW* on *P* processors

$$speedup = M^2 \Big/ 1 + \lceil 2(M-1)/P \rceil + \lceil (M-1)^2 / P \rceil \quad (1)$$

The parallelization potential is quite non-uniform when considering the diagonal, cross and peripheral blocks. Let $P = 8$ and $M = 4$. Then, one step is needed for executing the diagonal block, where 1 processor is loaded, and 7 processors stand idle. One step is needed for executing 6 cross blocks, where 6 processors are loaded, and 2 processors stand idle. Two steps are needed for executing 9 peripheral blocks: in first step all 8 processors are loaded; in second step only 1 processor is loaded, and 7 others stand idle. As a result, according to (1) the speedup of *PBFW* is 4 instead of expected ideal 8.

The promising alternative to the fork-join is dataflow parallelism. In the paper we develop dataflow networks which have such a property that the calculations of peripheral blocks can be moved over iterations along *m* in Algorithm 2.

Reordering of block calculations. The authors of [7] proved the following assertion related to the *FW* algorithm:

Claim 1. Suppose $d_{i,j}^{k+1}$, $k = 0 \ldots N-1$, is computed as

$$d_{i,j}^{k+1} = min(\, d_{i,j}^{k},\ d_{i,k}^{k'} + d_{k,j}^{k''} \,) \quad (2)$$

for $k \le k'$, $k'' \le N$, then upon termination, the Floyd – Warshall algorithm correctly computes the all-pairs shortest paths.

In *BFW* and *PBFW*, the diagonal block $B_{m,m}^{m+1}$ of type *D*0 is calculated through $B_{m,m}^{m}$ for which all $B_{m,m}^{1} \ldots B_{m,m}^{m-1}$ have been already calculated. The cross block $B_{v,m}^{m+1}$ of type *C*1 is calculated through $B_{m,m}^{m+1}$ and $B_{v,m}^{m}$ for which all $B_{v,m}^{1} \ldots B_{v,m}^{m-1}$ have been already calculated. The same holds for the cross block $B_{m,v}^{m+1}$ of type *C*2. The peripheral block $B_{v,u}^{m+1}$ of type *P*3 is calculated through $B_{v,u}^{m}$, $B_{v,m}^{m+1}$ and $B_{m,u}^{m+1}$. We formulate and prove Claim 2 which relaxes the requirement to blocks $B_{v,m}^{m+1}$ and $B_{m,u}^{m+1}$.

Claim 2. Suppose *P*3-type block $C_{v,u}^{m+1}$ (that is block $B_{v,u}^{m+1}$ calculated by a different algorithm *BFW'*), $m = 0 \ldots M-1$, is computed as

$$C_{v,u}^{m+1} = BCA\,(C_{v,u}^{m},\ C_{v,m}^{m'},\ C_{m,u}^{m''}) \quad (3)$$

for $m+1 \le m'$, $m'' \le M$, then upon termination, *BFW'* correctly computes the all-pairs shortest paths.

Proof. In *BFW* and *BFW*', blocks *D*0, *C*1 and *C*2 are calculated in the same manner. Let prove by induction that for *P*3-type blocks $C_{v,u}^{m+1}$ and $B_{v,u}^{m+1}$, $0 \le m \le M$-1 the following inequality holds:

$$C_{v,u}^{m+1} \le B_{v,u}^{m+1}, \quad (4)$$

which means inequality $c_{i,j} \le b_{i,j}$ for all pairs of matching elements of the corresponding blocks.

Base case. By definition we have $C_{v,u}^{0} = B_{v,u}^{0} = W_{v,u}$, therefore $C_{v,u}^{0} \le B_{v,u}^{0}$, $v, u = 0 \ldots M-1$.

Induction step. Suppose the inequality as follows holds:

$$C_{v,u}^{m} \le B_{v,u}^{m}, \qquad v, u = 0 \ldots M-1. \quad (5)$$

Then applying Algorithm 3 to the *BCA* call (3) we can conclude:

$$C_{v,u}^{m+1} \le BCA\,(B_{v,u}^{m},\ C_{v,m}^{m'},\ C_{m,u}^{m''}) \tag{6}$$

$$\le BCA\,(B_{v,u}^{m},\ C_{v,m}^{m+1},\ C_{m,u}^{m+1}) \tag{7}$$

$$\le BCA\,(B_{v,u}^{m},\ B_{v,m}^{m+1},\ B_{m,u}^{m+1}) \le B_{v,u}^{m+1}. \tag{8}$$

Inequality (6) is inferred from (3) and (5). Inequality (7) is inferred from Claim 1 applied S times to each element of block B calculated over blocks E, F and H in Algorithm 3 executed through the *BCA* call of (6):

$$b_{i,j}^{S\cdot(m+1)} = min\,(e_{i,j}^{S\cdot m},\ f_{i,k}^{S\cdot m'} + h_{k,j}^{S\cdot m''}) \le min\,(e_{i,j}^{S\cdot m},\ f_{i,k}^{S\cdot(m+1)} + h_{k,j}^{S\cdot(m+1)}), \tag{9}$$

where $S{\cdot}m$ is the calculation level of elements of the block that is at level m of calculation. Inequality (8) that proves (4) is inferred from (7) considering inequality (5) that is used to prove the inequality $C_{v,m}^{m+1} \le B_{v,m}^{m+1}$ of the type $C1$ blocks and to prove the inequality $C_{m,u}^{m+1} \le B_{m,u}^{m+1}$ of the type $C2$ blocks of the *BFW'* and *BFW* algorithms. All these blocks are calculated over the diagonal blocks which meet (5).

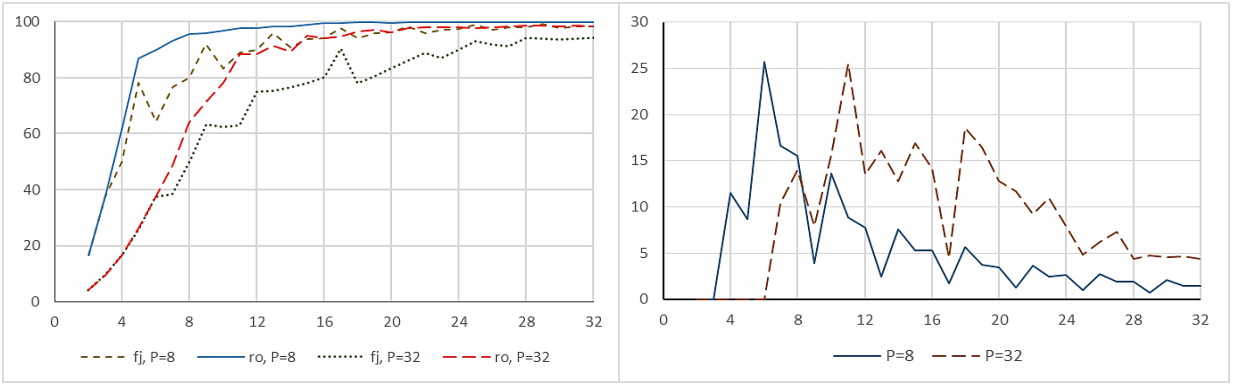
On the other hand, since the traditional blocked Floyd – Warshall algorithm computes the shortest paths at termination and (3) computes the length of some paths, we have:

$$B_{v,u}^{m+1} \le C_{v,u}^{m+1}. \tag{10}$$

It is derived from (4) and (10) that $B_{v,u}^{M} = C_{v,u}^{M}$, which completes the proof.

Claim 2 allows the delaying and reordering of peripheral blocks calculations. Such a reordering was used in the multi-threaded all-pairs shortest path algorithm [12] realized using a cooperative scheduler of threads.

Fig. 1, *a* depicts the estimated parallelization potential of the fork-join *PBFW* algorithm and the parallelization potential of an algorithm *PBFW"* obtained from the former one by reordering of block calculations and balancing the load of processors. To estimate the parallelization potential, a program in C++ was developed which implements *PBFW* and its modification *PBFW"*. Fig. 1, *b* shows that the reordering of block calculations can speed up the shortest paths parallel search up to 25 %. Moreover, the speedup can increase in case when the block execution time is variable. It should be noted that the gain of *PBFW"* against *PBFW* is being reduced with the growth of block count.



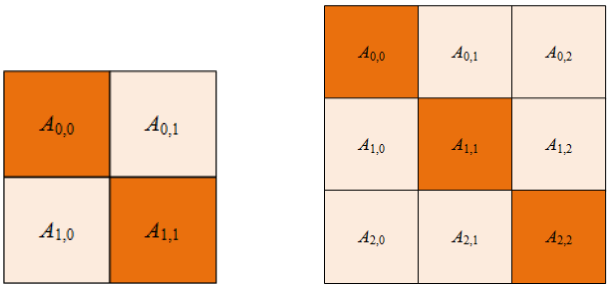
a) *b)*

Fig. 1. Processors load *a)* in percent (vertical axis) provided by algorithms *PBFW* (fork-join parallelism *fj*) and *PBFW"* (reordering *ro* of calculations) for block-counts $M = 2$ to $M = 32$ (horizontal axis) on 8 and 32 processors P; and lower bound of speedup *b)* in percent (vertical axis) *PBFW"* has against *PBFW* for block-counts $M = 2$ to $M = 32$ (horizontal axis) on 8 and 32 processors P

The *BFW* and *PBFW* are homogeneous in sense of calculating all blocks with the single *BCA* function. The authors of work [11] extended the algorithms to heterogeneous ones that calculate the blocks of types *D0*, *C1*, *C2* and *P3* using separate functions which operate faster than *BCA*.

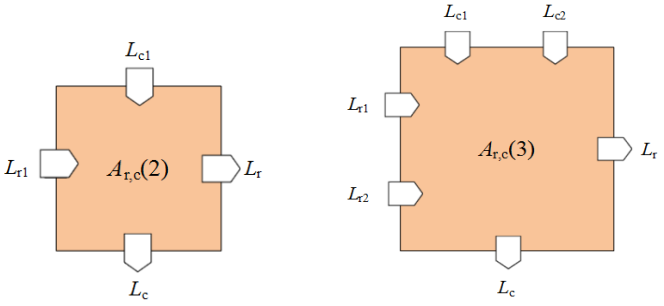
Modelling block calculations by actors. A separate actor $A_{r,c}$ is put into accordance with each block $B_{r,c}$ of block-matrix *B*, which introduces a matrix $A[M \times M]$ of actors. The structure and behavior of actor $A_{r,c}$ depends on two factors: 1) the size $M \times M$ of matrix *B*; 2) the location of $B_{r,c}$ in *B* (diagonal and non-diagonal blocks). Fig. 2 shows diagonal and non-diagonal actors in matrices $A[2 \times 2]$ and $A[3 \times 3]$. The size *M* influences the number of input ports and the total number of actions in the actor. The block location influences the structure and behavior of the actions incorporated in the actor. In the paper, we assume that the actors have access to two global variables *M* and *B* and assume that the actors' ports represent the block calculation levels but the blocks themself. Actor $A_{r,c}$ may update block $B_{r,c}$ and may not other blocks in *B*.

In each actor $A_{r,c}$, the number of input ports equals $2 \cdot (M - 1)$ and the number of output ports equals two. The input ports describe calculation levels of other blocks located in row *r* and column *c*. Both output ports describe the calculation level of block $B_{r,c}$. The overall number of ports and the actor interface are the same for all actors of *A*, no matter the actor is diagonal or non-diagonal. Fig. 3 depicts the input and output ports of the actor in matrices $A[2 \times 2]$ and $A[3 \times 3]$.



a) *b)*

Fig. 2. Matrices *a)* $A[2 \times 2]$ and *b)* $A[3 \times 3]$ of diagonal and non-diagonal CAL actors



a) *b)*

Fig. 3. Interface of actor $A_{r,c}(M)$ that models calculation of block $B_{r,c}$ in matrix $A[M \times M]$: *a)* actor $A_{r,c}(2)$ and *b)* actor $A_{r,c}(3)$

In actors of $A[2 \times 2]$, input port L_{r1} receives the token produced by another actor located in row *r*, and L_{c1} receives the token produced by another actor in column *c*. Two output ports L_r and L_c of $A_{r,c}$ send tokens describing the calculation level of $B_{r,c}$ to other actors in row *r* and column *c* respectively. In matrix $A[3 \times 3]$, the actor interface has four input ports L_{r1}, L_{r2}, L_{c1} and L_{c2} since the number of blocks in row *r* and column *c* is increased to three.

The diagonal and non-diagonal actors have different internal structure and different actions. Algorithm 4 depicts the behavior of a diagonal actor *Block_D* that models the calculation of block $B_{0,0}$ in matrix $A[2 \times 2]$. Input ports *L*_0_1 and *L*_1_0 describe the calculation level of blocks $B_{0,1}$ and $B_{1,0}$

respectively. Output ports *Lrow* and *Lcol* describe the calculation level of block $B_{0,0}$. Variables *Lev*, *Row* and *Col* that describe the calculation level, row, and column of $B_{0,0}$ represent the actor internal state.

The actor contains three actions: diagonal, peripheral, and reset. Action *Dig0* has no input, but two output tokens (both equal *Lev*) directed to ports *Lrow* and *Lcol*. The guard condition requires *Lev* = *Row* = *Col*. The action body increments *Lev* and calls the *BCA* function to recalculate block $B_{0,0}$ over itself. According to the guard, the action is fired once. Action *Per1* has three input and no output tokens. Tokens *L*01 and *L*10 arrive from ports *L*_0_1 and *L*_1_0, and the third token is a constant *k* = 1. The guard condition requires *Lev* < *L*01 and *Lev* < *L*10. The action body increments *Lev* and calls the *BCA* function to recalculate block $B_{0,0}$ over blocks $B_{0,1}$ and $B_{1,0}$. The peripheral action is fired when the input tokens have arrived, and its guard is satisfied. The *Reset* action sets *Lev* to 0. It is fired when the block has been recalculated *M* times.

Algorithm 4: Diagonal actor *Block_D* for block $B_{0,0}$ in $A[2 \times 2]$

```
actor Block_D (m) int L_0_1, int L_1_0 ==> int Lrow, int Lcol:
  int Lev := 0; int Row := m; int Col := m;
  Dig0: action    ==> Lrow: [Lev], Lcol: [Lev]                    // D0
    guard Lev = Row
    do
      Lev := Lev + 1;
      BCA (B[Row, Col], B[Row, Col], B[Row, Col]);
    end
  Per1: action L_0_1: [L01], L_1_0: [L10], 1:[k] ==>              // P3
    guard L01 > Lev and L10 > Lev
    do
      Lev := Lev + 1;
      BCA (B[Row, Col], B[Row, k], B[k, Col]);                    // P3
    end
  Reset: action    ==>
    guard Lev = M do Lev := 0; end
end
```

Algorithm 5: Non-diagonal actor *Block_N* for block $B_{0,1}$ in $A[2 \times 2]$

```
actor Block_N (v, u) int L_0_0, int L_1_1 ==> int Lrow, int Lcol:
  int Lev := 0; int Row := r; int Col := c;
  Crs0: action L_0_0: [L00]  ==> Lcol: [Lev]                      // C2
    guard Lev = L00 – 1
    do
      Lev := Lev + 1;
      BCA (B[Row, Col], B[Row, Row], B[Row, Col]);
    end
  Crs1: action L_1_1: [L11]  ==> Lrow: [Lev]                      // C1
    guard Lev = L11 – 1
    do
      Lev := Lev + 1;
      BCA (B[Row, Col], B[Row, Col], B[Col, Col]);                // P3
    end
  Reset: action    ==>
    guard Lev = M do Lev := 0; end
end
```

Algorithm 5 describes a non-diagonal actor *Block_N* that models the calculation of block $B_{0,1}$ in matrix $A[2 \times 2]$. Two input ports are *L*_0_0 and *L*_1_1. The output ports and state variables are the same as in actor *Block_D*. The actor contains two actions. Action *Crs0* has input token *L*00 arriving from port *L*_0_0 and has output token *Lev* sended to port *Lrow*. Its guard condition requires *Lev* = *L*00 – 1. The action body increments *Lev* and calls the *BCA* function to recalculate block $B_{0,1}$ over diagonal block $B_{0,0}$. *Crs0* is fired when a token arrives at its input port and its guard condition evaluates to true. The behaviour of *Crs1* is like those of *Crs0* except $B_{0,1}$ is recalculated over $B_{1,1}$. Each of actions *Crs0* and *Crs1* is fired once.

Algorithms 6 and 7 describe the behavior of diagonal actor $A_{0,0}$ (*Block_D*) and non-diagonal actor $A_{0,1}$ (*Block_N*) that models the calculation of blocks $B_{0,0}$ and $B_{0,1}$ in matrix $A[3 \times 3]$. In $A_{0,0}$, one action is *Dig* and all others are *Per*. In $A_{0,1}$, two actions are *Crs* and all others are *Per*. Compared to actors of matrix $A[2 \times 2]$, the actors of $A[3 \times 3]$ have four input ports instead of two and have an additional peripheral action each. The output ports and state variables are the same. Unlike the actor of $A[2 \times 2]$, the *Block_D* of $A[3 \times 3]$ of Algorithm 6 contains two peripheral actions that are competitive in firing and can be fired in arbitrary order. The actions' guards are redundant and removed since their firing is correctly managed by input tokens. Thus, block $B_{0,0}$ can have two firing sequences: 1) $B^0_{0,0}$, $B^1_{0,0}$ and $B^2_{0,0}$; 2) $B^0_{0,0}$, $B^2_{0,0}$ and $B^1_{0,0}$. CAL and its implementations require to resolve such competitions in advance by adding schedule and priorities. We do not follow this way since the order of firings of the actions does not influence the computation result. We perform a relaxation of CAL, omit the schedule and priorities in the actors and create our own multi-threaded implementation of CAL which resolves the competitions by means of an appropriate mechanism of synchronizing concurrent actions.

Moreover, we remove guards of actions *Crs0* and *Crs1* because the conditions they describe are fully satisfied by the conditions of arriving tokens on input ports.

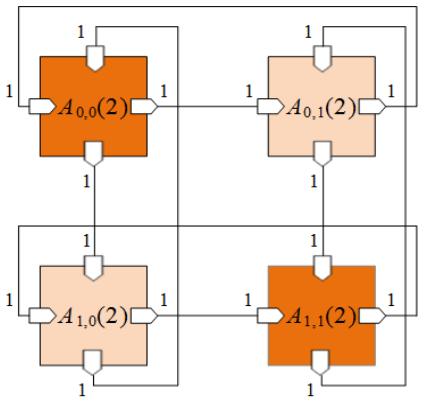
Algorithm 6: Diagonal actor *Block_D* for block $B_{0,0}$ in $A[3 \times 3]$

```
actor Block_D (m) int L_0_1, int L_0_2, int L_1_0, int L_2_0 ==>
                  int Lrow, int Lcol:
  int Lev := 0; int Row := r; int Col := c;
  Dig0: action    ==> Lrow: [Lev], Lcol: [Lev]          // D0
    guard Lev = Row
    do
      Lev := Lev + 1;
      BCA (D[Row, Col], D[Row, Col], D[Row, Col]);
    end
  Per1: action L_0_1: [L01], L_1_0: [L10], 1:[k] ==> // P3
    do
      Lev := Lev + 1;
      BCA (D[Row, Col], D[Row, k], D[k, Col]);
    end
  Per2: action L_0_2: [L02], L_2_0: [L20], 2:[k] ==> // P3
    do
      Lev := Lev + 1;
      BCA (D[Row, Col], D[Row, k], D[k, Col]);
    end
  Reset: action    ==>
    guard Lev = M do Lev := 0; end
end
```

Algorithm 7: Non-diagonal actor *Block_N* for block $B_{0,1}$ in $A[3 \times 3]$

```
actor Block_N (v, u) int L_0_0, int L_0_2, int L_1_1, int L_2_1
                  ==>int Lrow, int Lcol:
  int Lev := 0; int Row := r; int Col := c;
  Crs0: action L_0_0: [L00]  ==> Lcol: [Lev]          // C2
    do
      Lev := Lev + 1;
      BCA (D[Row, Col], D[Row, Row], D[Row, Col]);
    end
  Crs1: action L_1_1: [L11]  ==> Lrow: [Lev]          // C1
    do
      Lev := Lev + 1;
      BCA (D[Row, Col], D[Row, Col], D[Col, Col]);
    end
  Per2: action L_0_2: [L02], L_2_1: [L21], 2:[k]  ==> // P3
    do
      Lev := Lev + 1;
      BCA (D[Row, Col], D[Row, k], D[k, Col]);
    end
  Reset: action    ==>
    guard Lev = M do Lev := 0; end
end
```

Parallel dataflow networks of actors for shortest paths search. Composing actors into a network, setting connections among their input and output ports, and allocating buffers to the connections establish a dataflow network. The shortest paths search network structure depends on *M*. Fig. 4 shows a graphical view and a CAL-code of the NW_{22} network constructed on matrix $A[2 \times 2]$.



a)

```
network NW22 () ⇒ :
entities
        A00 = Block_D (0,0);
        A01 = Block_N (0,1);
        A10 = Block_N (1,0);
        A11 = Block_D (1,1);
structure
        A00. Lrow --> A01.L_0_0;
        A00. Lcol --> A10.L_1_1;
        A01. Lrow --> A00.L_0_1;
        A01. Lcol --> A11.L_1_0;
        A10. Lrow --> A11.L_0_1;
        A10. Lcol --> A00.L_1_0;
        A11. Lrow --> A10.L_0_0;
        A11. Lcol --> A01.L_1_1;
end
```

b)

Fig. 4. Dataflow network NW_{22} constructed on matrix $A[2 \times 2]$ has 4 actors and 8 channels with buffers on them: *a)* graphical view; *b)* CAL-code

NW_{22} consists of two diagonal and two non-diagonal actors, twelve actions and eight channels annotated with produced and consumed token rates. All rates are 1. Every action of every actor is fired once during the network operation. The diagonal Dig_m action of actor $A_{m,m}$ that is guarded with $Lev = m$ is fired once. It produces tokens, which are transferred to actions of cross non-diagonal actors $A_{m,v}$ and $A_{u,m}$ on row m and column m. Since the tokens are produced once, the cross actions Crs_v and Crs_u of the actors are fired once. The cross actions produce once and transfer tokens to actions of all actors outside the cross, therefore, all the peripheral actions can be fired once.

To prove the rate consistency of NW_{22}, we construct a combined balance equation for each channel $(A_v.p_i, A_u.p_j)$ connecting output port p_i of actor A_v with token rate $R(A_v.p_i)$ to input port p_j of actor A_u with token rate $R(A_u.p_j)$:

$$F(A_v.p_i) \cdot R(A_v.p_i) = F(A_u.p_j) \cdot R(A_u.p_j), \quad (11)$$

where $F(A_v.p_i)$ is the number of firings of A_v that produce tokens at p_i, and $F(A_u.p_j)$ is the number of firings of A_u that consume tokens at p_j. For NW_{22}, the system of balance equations (11) is described by (12)

$$\begin{aligned}
&1.\ F(A_{0,0}.Lrow) \cdot 1 = F(A_{0,1}.L_0_0) \cdot 1\\
&2.\ F(A_{0,0}.Lcol) \cdot 1 = F(A_{1,0}.L_1_1) \cdot 1\\
&3.\ F(A_{0,1}.Lrow) \cdot 1 = F(A_{0,0}.L_0_1) \cdot 1\\
&4.\ F(A_{0,1}.Lcol) \cdot 1 = F(A_{1,1}.L_1_0) \cdot 1\\
&5.\ F(A_{1,0}.Lrow) \cdot 1 = F(A_{1,1}.L_0_1) \cdot 1\\
&6.\ F(A_{1,0}Lcol) \cdot 1 = F(A_{0,0}.L_1_0) \cdot 1\\
&7.\ F(A_{1,1}.Lrow) \cdot 1 = F(A_{1,0}.L_0_0) \cdot 1\\
&8.\ F(A_{1,1}.Lcol) \cdot 1 = F(A_{0,1}.L_1_1) \cdot 1.
\end{aligned} \quad (12)$$

Equations 1 and 2 of (12) are satisfied because the single firing of action Dig_0 of actor $A_{0,0}$ (we denote $A_{0,0}.Dig_0$) produces at ports *Lrow* and *Lcol* the tokens consumed by single firing of $A_{0,1}.Crs_0$ and single firing $A_{1,0}.Crs_1$ respectively. Equation 4 is satisfied as $A_{0,1}.Crs_0$ is fired producing at *Lcol* a token that is consumed by $A_{1,1}.Per_1$. Equation 5 is satisfied as $A_{1,0}.Crs_1$ is fired producing at *Lrow* a token that is consumed by $A_{1,1}.Per_1$. Equations 7 and 8 are satisfied because the single firing of action Dig_0 of actor $A_{1,1}$ produces at ports *Lrow* and *Lcol* tokens consumed by firings $A_{1,0}.Crs_0$ and $A_{0,1}.Crs_1$ respectively. Equation 6 is satisfied by the firing of $A_{1,0}.Crs_0$ producing at *Lcol* the token that is consumed by $A_{0,0}.Per_1$. Equation 3 is satisfied by the firing of $A_{0,1}.Crs_0$ producing at *Lcol* the token that is consumed by $A_{0,0}.Per_1$.

The *Reset* action of all diagonal and non-diagonal actors sets *Lev* to initial state 0 after firing of all other actions. This guarantees the boundedness of FIFO buffers in NW_{22}. The following schedule proves the liveness of network NW_{22}

$$A_{0,0}.Dig_0,\ A_{0,1}.Crs_0,\ A_{1,0}.Crs_1,\ A_{1,1}.Per_1,\ A_{1,1}.Dig_0,\ A_{0,1}.Crs_1,\ A_{1,0}.\mathrm{Crs}_0 \text{ and } A_{0,0}.Per_1.$$

The firing of actions of different actors can proceed in series and in parallel. Fig. 5 depicts a rate-consistent dataflow network constructed on matrix $A[3 \times 3]$, which obtains the properties of boundedness and liveness. Therefore, the network supports the synchronous dataflow model (SDF) of computation.

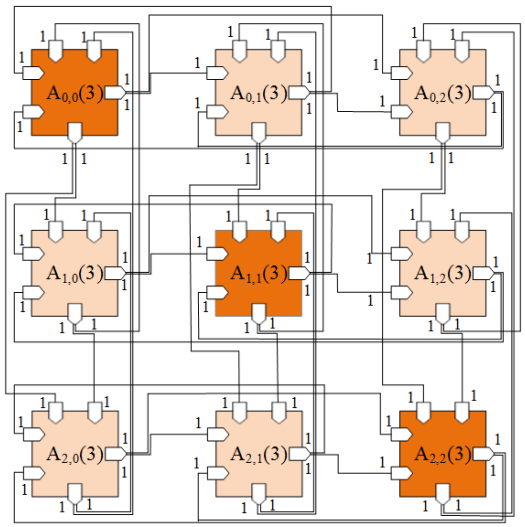


Fig. 5. Graphical view of dataflow network NW_{33} constructed of 9 actors and 27 communication channels from matrix $A[3 \times 3]$

Generation of parallel dataflow networks of actors. Given a blocked matrix $B[M \times M]$, our goal is to generate a matrix $A[M \times M]$ of actors and to establish connections between their input and output ports, thus generating a dataflow network NW_{MM} for shortest paths search. The key new result of the section is a procedure of generating actors of two types and generating a network NW_{MM} for various M. In NW_{MM}, the number of actors is M^2. Each actor has $2 \cdot (M-1)$ input ports, 2 output ports, and M actions.

Algorithm 8 generates the diagonal and non-diagonal actors for block $B_{r,c}$ of matrix $B[M \times M]$. It adds the input and output ports and the state variables to each actor. Then, it creates actions. For the diagonal actor *Block_D* modelling $A_{r,r}$ ($r = c$), it creates one diagonal action Dig_r and $M-1$ peripheral actions Per_k, $k = 0 \ldots M-1$, $k \neq r$. Action Dig_r produces tokens at output ports *Lrow and Lcol* that are transferred to $2 \cdot (M-1)$ input ports of other actors from row r and column c of matrix A. The peripheral actions of the diagonal actor are divided into two groups: $G_D^1 = \{Per_0, \ldots, Per_{r-1}\}$ and $G_D^2 = \{Per_{r+1}, \ldots, Per_{M-1}\}$.

Algorithm 8 adds one cross action of type $C1$, one cross action of type $C2$ and M-2 peripheral actions to the non-diagonal actor *Block_N* representing $A_{r,c}$, $r \neq c$. Cross action Crs_r calculates block $B_{r,c}$ of type $C2$ through diagonal block $B_{r,r}$ and produces a token at output port *Lcol* which is transferred to $M-1$ input ports of actors on column c. Cross action Crs_c calculates block $B_{r,c}$ of type $C1$ over diagonal block $B_{c,c}$ and produces a token at output port *Lrow* which is transferred to $M-1$ input ports of actors on row r. Let $r < c$. Peripheral actions Per_k, $k = 0 \ldots M-1$, $k \neq r$ and $k \neq c$ of actor $A_{r,c}$ are divided into three groups: $G_N^1 = \{Per_0 \ldots Per_{r-1}\}$, $G_N^2 = \{Per_{r+1} \ldots Per_{c-1}\}$ and $G_N^3 = \{Per_{c+1} \ldots Per_{M-1}\}$.

Algorithm 8: Generation of diagonal and non-diagonal actors (*Generate_Actor*)

Input: a number M of blocks (actors) in row (column)
Input: a row number r
Input: a column number c
Output: a generated actor *Actor*

```
if r = c then
    Actor←CreateActor ("Block_D", r, c);
else
    Actor←CreateActor ("Block_N", r, c);
for k ← 0 to M − 1 do
    if k ≠ c then Actor.AddInPort (r, k);
for k ← 0 to M − 1 do
    if k ≠ r then Actor.AddInPort (k, c);
Actor.AddOutPort ("Lrow");  Actor.AddOutPort ("Lcol");  Actor.AddStateVar ("Lev", 0);
Actor.AddStateVar ("Row", r);  Actor.AddStateVar ("Col", c);
if r = c then
    ActionD0 ← CreateAction ("Dig", r);  Actor.AddAction (ActionD0);
    for k ← 0 to M − 1 do
        if k ≠ r then  ActionP3 ← CreateAction ("Per", k);  Actor.AddAction (ActionP3);
else
    ActionC1 ←CreateAction ("Crs", c);  Actor.AddAction (ActionC1);
    ActionC2 ←CreateAction ("Crs", r);  Actor.AddAction (ActionC2);
    for k ← 0 to M − 1 do
        if k ≠ r and k ≠ c then  ActionP3 ←CreateAction("Per", k); Actor.AddAction(ActionP3);
return Actor;
```

Once the actors are created, Algorithm 9 generates connections among them. It traverses all actors identified by *ID_Dest* and *Actor_Dest* and considers them as destinations. For each destination actor, the algorithm takes every input port *Port_Dest* and selects a single source that is an actor

(*ID_Source*, *Actor_Source*) and its output port (*Port_Source*). The first loop along variable *p* goes over destination ports and their sources of column *c*, and the second loop along variable *p* goes over destination ports and sources of row *r*.

The generated dataflow network NW_{MM} has the properties of rate consistency, boundedness, and liveness. The behavior of actors and actions in the network is correctly synchronized. Each actor calculates its own block; therefore, no conflicts occur between the actors. In the diagonal and non-diagonal actors, all actions are connected to distinct input ports, which guarantees that the same token cannot be consumed by different actions; it leads to the independent firing of actions.

Algorithm 9: Generation of connections between actors (*Connect_Actors*)

Input: a number *M* of blocks in row (column)
Output: a CAL network *NW*

```
for r ← 0 to M − 1 do
    for c ← 0 toM − 1 do
        ID_Dest ← r × M + c;
        Actor_Dest←AcatorName (r, c);
        for p ← 0 to M − 1 do
            if p ≠ r then
                Port_Dest←PortName (p, c);   ID_Source ← p × M + c;
                Actor_Source ← AcatorName (p, c);   Port_Source ← "Lcol";
                NW.AddConnect (ID_Dest, Actor_Dest, Port_Dest, ID_Source, Actor_Source, Port_Source);
        for p ← 0 to M − 1 do
            if p ≠ c then
                Port_Dest ← PortName (r, p);   ID_Source ← r × M + p;
                Actor_Source←AcatorName (r, p);   Port_Source ← "Lrow";
                NW.AddConnect (ID_Dest, Actor_Dest, Port_Dest, ID_Source, Actor_Source, Port_Source);
return NW;
```

If block B_1^k of level *k* is used directly (or over other blocks) for calculating block B_2^l to level *l*, we denote it with precedence $B_1^k \Rightarrow B_2^l$. It can be observed from Algorithm 2 that in *BFW* the precedence $B_{v,u}^m \Rightarrow B_{v,u}^{m+1}$ holds for all *v*, *u*, *m* = 0 … *M* – 1. In NW_{MM}, a precedence relation between block calculations exists, which determines a partial order of firing actions.

Claim 3. In the diagonal CAL-actor *Block_D* processing block $B_{m,m}$, actions of G_D^1 are fired before action Dig_{m} and actions of G_D^2 are fired after Dig_{m}. Actions of G_D^1 as well as actions of G_D^2 can be fired in any order with respect to each other. Then NW_{MM} correctly computes the shortest paths between all pairs of vertices.

Proof. In *BFW*, $B_{m,m}^k \Rightarrow B_{m,m}^{k+1}$, *k* = 0 … *m* hold. Actions of G_D^1 calculate block $B_{m,m}$ from level $B_{m,m}^0$ to levels $B_{m,m}^1$ … $B_{m,m}^m$. Action Dig_{m} calculates the block to level $B_{m,m}^{m+1}$ and actions of G_D^2 calculate it to levels $B_{m,m}^{m+2}$ … $B_{m,m}^M$ respectively. The diagonal block calculations hold the following:

1. For block type *D*0, block $B_{m,m}$ must be calculated to level *m* before calculating $B_{m,m}^{m+1}$. It can be only done by calculating $B_{m,m}^1$ … $B_{m,m}^m$ through $B_{m,0}^1$, $B_{0,m}^1$ … $B_{m,m-1}^m$, $B_{m-1,m}^m$ while considering the block as of type *P*3. Then in NW_{MM}, $B_{m,m}^1 \Rightarrow B_{m,m}^m$ … $B_{m,m}^{m-1} \Rightarrow B_{m,m}^m$ hold. The actions of G_D^1 are fired before action Dig_{m}.

2. The precedencies $B_{m,m}^{m+1} \Rightarrow B_{m,m}^{m+2} \ \ldots \ B_{m,m}^{m+1} \Rightarrow B_{m,m}^{M}$ are derived from the fact that $B_{m,m}^{m+2} \ \ldots \ B_{m,m}^{M}$ are directly or indirectly calculated through $B_{m,m}^{m+1}$.

3. The CAL network can reorder $B_{m,m}^{k}$ and $B_{m,m}^{k+1}$, $k = 1 \ \ldots \ m-1$, for three reasons: a) the block calculations are independent and, therefore, do not precede each other since $B_{m,m}$ is calculated at levels k and k+1 by accounting for paths between vertices of V_m passing through vertices of non-intersected subsets V_k and V_{k+1}; b) Claim 2 allows the calculation of $B_{m,m}^{k}$ through $B_{m,k}^{k'}$ and $B_{k,m}^{k''}$ of higher levels of $k' > k$ and $k'' > k$; c) the operation of choosing a minimum of two numbers is commutative and associative.

4. Assertions like those of point 3 are proved for the case when $k = m + 1 \ \ldots \ M - 1$.

Points 1 and 2 prove that in *Block_D* actions of G_D^1 are fired before action Dig_{m} and actions G_D^2 are fired after Dig_{m}. Points 3 and 4 prove that actions of G_D^1 (as well as actions of G_D^2) can be fired in any order with respect to each other in *Block_D*. The claim is proved.

Claim 4. In the non-diagonal CAL-actor *Block_N*, which processes block $B_{v,u}$, $v \neq u$, actions of G_N^1 are fired before action Crs_{v}, actions of G_N^2 are fired after Crs_{v} and before Crs_{u}, and actions of G_N^3 are fired after Crs_{u}. In each of three subsets G_N^1, G_N^2 and G_N^3 the actions can be fired in any order with respect to each other. Then NW_{MM} correctly computes the shortest paths between all pairs of vertices.

Proof. Let $v < u$. Actions of G_N^1 calculate block $B_{v,u}$ from level $B_{v,u}^0$ to levels $B_{v,u}^1 \ \ldots \ B_{v,u}^v$. Action Crs_{v} calculates the block to level $B_{v,u}^{v+1}$ and actions of G_N^2 calculate it to levels $B_{v,u}^{v+2} \ \ldots \ B_{v,u}^u$. Action Crs_{u} calculates the block to level $B_{v,u}^{u+1}$ and actions of G_N^3 calculate it to levels $B_{v,u}^{u+2} \ \ldots \ B_{v,u}^M$. The following precedencies hold for the non-diagonal block calculations:

1. Since the block type *C*2 establishes precedence $B_{v,u}^v \Rightarrow B_{v,u}^{v+1}$, block $B_{v,u}$ must be calculated to level v before calculating $B_{v,u}^{v+1}$. The only way is to perform calculations $B_{v,u}^1 \ \ldots \ B_{v,u}^v$ by considering the block as of type *P*3. In this case, $B_{v,u}^1 \Rightarrow B_{v,u}^v \ldots B_{v,u}^{v-1} \Rightarrow B_{v,u}^v$ hold.

2. Since the block type *C*1 establishes precedence $B_{v,u}^u \Rightarrow B_{v,u}^{u+1}$, block $B_{v,u}$ must be calculated to level u before calculating $B_{v,u}^{u+1}$. The only way is to perform calculations of $B_{v,u}^{v+2} \ \ldots \ B_{v,u}^u$ after calculating $B_{v,u}^{v+1}$ by considering the block as of type *P*3. Therefore, precedencies $B_{v,u}^{v+2} \Rightarrow B_{v,u}^u \ \ldots \ B_{v,u}^{u-1} \Rightarrow B_{v,u}^u$ hold.

3. The precedencies $B_{v,u}^{u+1} \Rightarrow B_{v,u}^{u+2} \ \ldots \ B_{v,u}^{u+1} \Rightarrow B_{v,u}^M$ are derived from the fact that $B_{v,u}^{u+2} \ \ldots \ B_{v,u}^M$ can only be calculated through $B_{v,u}^{u+1}$.

4. NW_{MM} can refuse the precedence $B_{v,u}^m \Rightarrow B_{v,u}^{m+1}$, $m = 1 \ \ldots \ v-1$ and can reorder $B_{v,u}^m$ and $B_{v,u}^{m+1}$ for three reasons: a) the calculations are independent and, therefore, do not precede each other since $B_{v,u}$ is calculated at levels m and m+1 by accounting for paths between vertices of subsets V_v and V_u passing through vertices of non-intersected subsets V_m and V_{m+1}; b) Claim 2 allows the calculation of $B_{v,u}^m$ through $B_{v,u}^{m'}$ and $B_{v,u}^{m''}$ at higher levels of $m' > m$ and $m'' > m$; c) the operation of choosing a minimum of two numbers is commutative and associative.

5. Similar assertions are proved for the case when $m = v + 1 \ \ldots \ u - 1$ and when $m = u + 1 \ \ldots \ M - 1$.

Points 1, 2 and 3 prove that in *Block_N* the actions of G_N^1 are fired before action Crs_v, the actions of G_N^2 are fired after Crs_v and before Crs_u, and the actions of G_N^3 are fired after action Crs_u. Points 4 and 5 prove that the actions of G_N^1 (as well as of G_N^2 and G_N^3) can be fired in any order with respect to each other in *Block_N*. The claim is proved.

The number of possible firing sequences of actions of diagonal actors is $|G_D^1|!\cdot|G_D^2|!$ and is $|G_N^1|!\cdot|G_N^2|!\cdot|G_N^3|!$ of non-diagonal actors, where $|G|!$ is factorial of G's cardinality. The number rapidly increases with the growth of M.

CAL-network development tool and tunable dataflow CAL-engine in C/C++. Based on the C/C++ language we have developed a tool for creating parallel networks of dataflow actors and have developed a tunable multithreaded CAL-based engine (fig. 6) for multicore systems. The tool and engine were used for the realization of the proposed dataflow CAL-actors and parallel networks, which solve the all-pairs shortest path problem. In the current version of engine, any action of any actor is implemented by a separate thread, although we consider other solutions of mapping actors to threads. Since many concurrently and asynchronously operating actions (threads) can simultaneously update shared resources, synchronization primitives protect the resources. The concurrent asynchronous behavior is a source of increasing the throughput of the networks and speeding up the shortest paths search against OpenMP.

Fig. 6 depicts a flexible architecture of the tool and engine. It provides methods of specifying and generating actors, connections and whole network for the problem under solving. The generators process the specifications and elaborate a parallel dataflow network in an internal format. A multithreaded CAL-based engine is firstly tuned to the network and then implements it on a multi-core system.

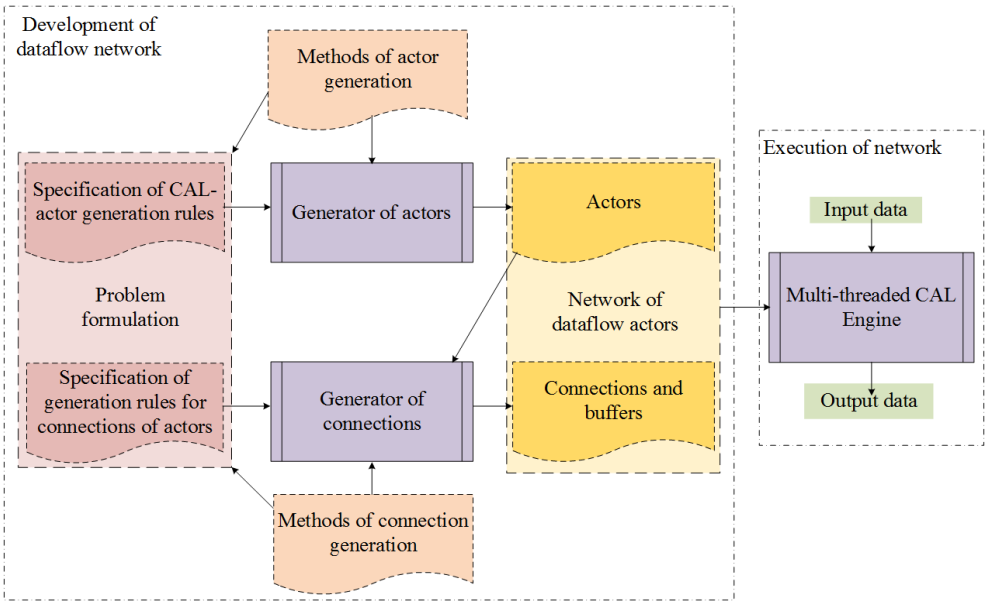


Fig. 6. Architecture of CAL-based development tool and tunable CAL-engine implemented in C/C++

Although the actions may be fired (executed) within one actor only in series, the engine parallelizes the actions' implementations regarding their readiness and selection before firing. It can execute the action in parallel with checking if another action is ready to be fired next (the FSM state, the availability of action's input tokens, the guard condition, and the availability of output ports to receive the produced tokens are considered). To have this property, the CAL-engine implements each action by a separate thread.

The advantage of our CAL-based implementation is the integration of CAL in C/C++ in such a way that all facilities of C/C++ become available for the implementation. C/C++ classes of objects implement all components of the CAL-actor and network. As a result, the network and each of its actors are instantiated over complex data structures and sets of methods written in C/C++. The body of each action is represented as a function in C/C++. To ensure that the implementation is consistent with the CAL model of computation, we have developed a tool for checking and validating the structure.

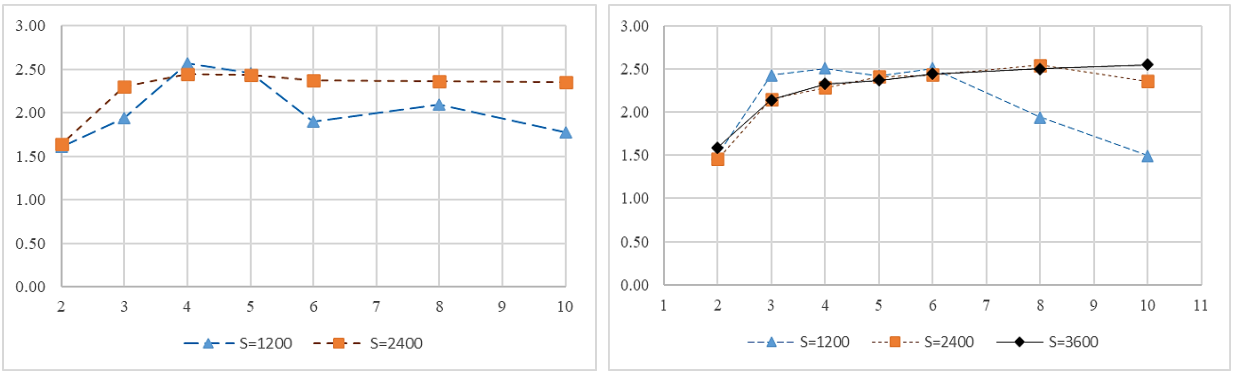
Experimental results. In the paper, we report results given by the implementations of the dataflow parallel networks and OpenMP based implementations of the *PBFW* algorithm on multi-core systems. The networks of dataflow actors were generated from various block-matrix configurations and block sizes and implemented in C/C++ with the threaded CAL engine. The results are obtained on randomly generated simple complete weighted directed graphs of 1200, 2400, 3600 and 4800 vertices on four Intel(R) Core(TM) processors i3-550, i5-5200U, i7-9750h and i7-10700. The graphs provide high computational load which gives a correct comparison of the implementations. Table describes processors' parameters.

Parameters of four multi-core processors

Processor	Cache L1, KB	Cache L2, KB	Cache L3, MB	Frequency, GHz	Cores	Logical processors
i7-10700	8× 64	8× 256	16.0	2.90	8	16
i7-9750h	6 × 64	6 × 256	12.0	2.60	6	12
i5-5200U	2 × 64	2 × 256	3.0	2.20	2	4
i3-550	32 + 64	2×256	4.0	3.20	2	

Fig. 7 and 8 show the speedup the dataflow CAL-networks and their multi-threaded implementations have given against matching single-thread implementations of the Floyd–Warshall Algorithm 1. The block count M in matrix B varied in the range 2 to 10. The number of actors varied in the range 4 to 100, the number of actor input ports varied in the range 2 to 18, and the number of output ports was 2 for all actors. The number of actions within actor varied in the range 2 to 10, therefore, the number of threads in the network implementations reached up to 1000. The optimal number M_{opt} of blocks has given the highest speedup of the CAL-networks.

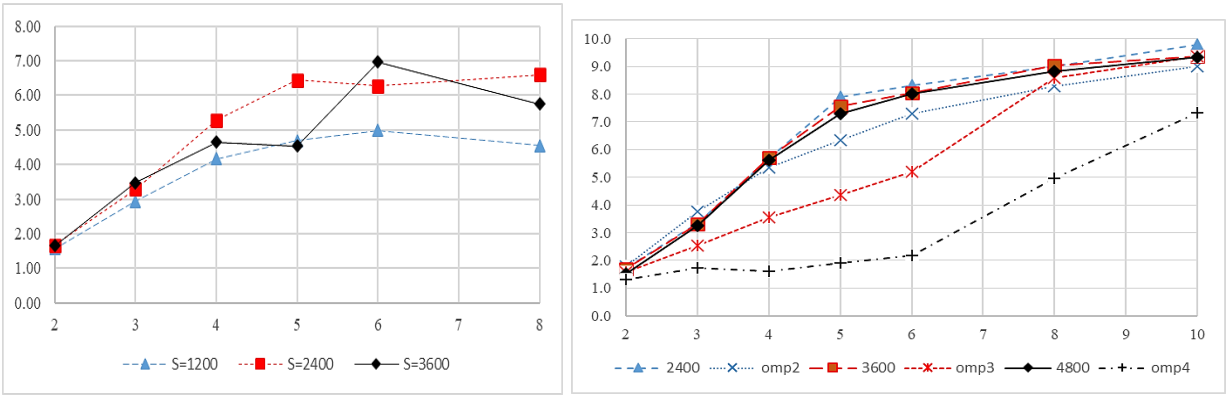
On the 2-core i3-550 processor and the graph of 1200 vertices (fig. 7, *a*), the CAL-network has given the speedup of 2.57 at $M_{opt} = 4$. For larger M the speedup has reduced to 1.61. On the graph of 2400 vertices, the highest speedup of 2.45 is also obtained at $M_{opt} = 4$. For both graphs, the speedup is larger than the number of cores. For the 2-core i5-5200U processor and the graph of 1200 vertices (fig. 7, *b*) we can observe the similar pattern, where the highest speedup of 2.51 is obtained at $M_{opt} = 4$. The increase in the graph size to 2400 and 3600 shifts the value of M_{opt} from 4 to 8 (speedup is 2.54) and then to 10 (speedup is 2.55). We can mainly explain this as the CAL-networks exploit the processor hierarchical memory and caches more efficiently for lower size of block.



a) *b)*

Fig. 7. Speedup (vertical axis) of multi-threaded CAL-networks against single-thread *FW* vs. block count *M* (horizontal axis) on *a)* i3-550 and *b)* i5-5200U processors for three sizes of graphs: 1200, 2400 and 3600 vertices

On the 6-core i7-9750hprocessor and the graph of 1200 vertices (fig. 8, *a*), the CAL-networks have given the maximum speedup of 4.98 at $M_{opt} = 6$, which is smaller than the number of cores due to the insufficient potential parallelism and low useful load (see fig. 1, *a*). On the graphs of 2400 and 3600 vertices, the highest speedups of 6.59 and 6.96 are obtained at $M_{opt} = 8$ and $M_{opt} = 6$ respectively. In both cases, the speedup exceeds the number of cores. On the 8-core i7-10700 processor, the speedup patterns by the CAL-networks are very close for graphs of 2400, 3600 and 4800 vertices (fig. 8, *b*). For all graphs the maximum speedup of 9.78, 9.37 and 9.34 is obtained at $M_{opt} = 10$. Fig. 8, *b* also shows the speedup given by OpenMP on the three graph-sizes, which is significantly less against the networks. The speedup is being decreased with the growth of the graph-size and is being increased with the growth of the number of blocks. It can be observed that the CAL-networks convincingly gain the *BFW* OpenMP implementations with respect to runtime.



a) *b)*

Fig. 8. Speedup (vertical axis) of multi-threaded CAL-networks and OpenMP-*BFW* implementations against single-thread *FW* vs. block count *M* (horizontal axis) on *a)* i7-9750h and *b)* i7-10700 processors for four sizes of graphs (omp2 – 2400, omp3 – 3600 and omp4 – 4800 vertices)

According to (1), the estimated speedup of the block-parallel Floyd – Warshall algorithm implemented in the fork-join style (OpenMP) is 1.33, 3.00, 4.00, 6.25, 5.14, 6.40 and 6.66 for $M = 2$, 3, 4, 5, 6, 8, 10 respectively on 8 cores. For comparison, the networks have given on i7-10700 and on the graph of 2400 vertices much higher speedup of 1.72, 3.35, 5.68, 7.90, 8.31, 9.01 and 9.78 respectively for the same values of *M*. We can mainly explain this by efficient exploitation of caches and advantages of the networks and their threaded parallel implementations due to highly asynchronous behaviour.

The graphics depicted in fig. 7 and 8 have found out the patterns as follows:

1. There is an optimal number M_{opt} of blocks for which the speedup by the multithreaded CAL-networks is the highest compared to the single-thread *FW*.
2. The highest speedup given by the dataflow networks exceeds the number of cores, which is a very good result for the blocked algorithm with strong data dependences between blocks.
3. M_{opt} depends on the number *P* of cores, the block-matrix size *M*, the graph size *N*, and the scheduler of threads of the operating system.
4. The increase in the size *M* of matrices *B* and *A* increases the amount of parallelism in the CAL-networks, which leads to the growth of computation speedup.
5. The larger number *P* of cores requires more parallelism and therefore larger M_{opt}.
6. The growth of the graph size *N* usually leads to the increase of M_{opt} as the processor caches operate more efficiently at smaller block sizes [13].
7. The CAL-networks give the speedup which is higher than that OpenMP gives.

It should be noted that the scheduler of threads of the operating system influences the order of executions of threads, and the increase in M increases the number of threads in the CAL-network implementations which increases the workload of the operating system.

Conclusion. Nowadays, the blocked Floyd – Warshall algorithm is typically parallelized in the fork-join style with OpenMP where each block is calculated in a loop level-by-level. The paper has proven that the block calculations can be reordered, thus increasing the load of cores in the multi-core system. The simulation tool has shown that the reordering can speed up the shortest paths search up to 25 %. The paper has proposed a novel method of generating dataflow networks of CAL-actors, where the management of actor and action firing is carried out over the block calculation levels. The new feature of the networks is that in each actor the executions of actions are ordered partially. The multi-threaded tuneable CAL-engine accounts for the feature and implements the networks in C/C++. The experiments on large complete directed graphs and four multi-core processors have shown that at optimal block count the networks speed up computations against the single-threaded implementations by the following figures: i3-550 (2 cores) – 2.57 (28.5 % higher than core count); i5-5200U (2 cores) – 2.55 (27.5 % higher than core count); i7-9750h (6 cores) – 6.96 (16.0 % higher than core count); i7-10700 (8 cores) – 9.78 (22.3 % higher than core count).

References

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

2. Madkour A, Aref W. G., Rehman F. U., Rahman M. A., Basalamah S. A. *Survey of Shortest-Path Algorithms*, 2017, 26 p. Available at: https://arxiv.org/abs/1705.02044 (accessed 23.11.2022).

3. Anu P., Kumar M. G. Finding all-pairs shortest path for a large-scale transportation network using parallel Floyd-Warshall and parallel Dijkstra algorithms. *Journal of Computing in Civil Engineering*, 2013, vol. 27, no. 3, pp. 263–273.

4. Prihozhy A. A., Mattavelli M., Mlynek D. Evaluation of parallelization potential for efficient multimedia implementations: dynamic evaluation of algorithm critical path. *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 5, 2005, pp. 593–608.

5. Singh A., Mishra P. K. Performance analysis of Floyd Warshall algorithm vs rectangular algorithm. *International Journal of Computer Applications*, 2014, vol. 107, no. 16, pp. 23–27.

6. Venkataraman G. A., Sahni S., Mukhopadhyaya S. Blocked all-pairs shortest paths algorithm. *Journal of Experimental Algorithmics (JEA)*, 2003, vol. 8, pp. 857–874.

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, no. 9, pp. 769–782.

8. Madduri K., Bader D. A., Berry J. W., Crobak J. R. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, 6 January 2007*. New Orleans, 2007, pp. 23–35.

9. Albalwi E., Thulasiraman P., Thulasiram R. Task level parallelization of all pair shortest path algorithm in OpenMP 3.0. *Advances in Computer Science and Engineering (CSE 2013)*. Los Angeles, Atlantis Press, 2013, pp. 109–112.

10. Tang P. Rapid development of parallel blocked all-pairs shortest paths code for multi-core computers. *IEEE SOUTHEASTCON 2014, Lexington, KY, USA, 13–16 March 2014*. Lexington, 2014, pp. 1–7.

11. Prihozhy A. A., Karasik O. N. *Heterogeneous blocked all-pairs shortest paths algorithm*. Sistemnyj analiz i prikladnaja informatika [*System Analysis and Applied Information Science*], 2017, no. 3, pp. 68–75 (In Russ.). https://doi.org/10.21122/2309-4923-2017-3-68-75

12. Karasik O. N., Prihozhy A. A. Threaded block-parallel algorithm for finding the shortest paths on graph. Doklady Belorusskogo gosudarstvennogo universiteta informatiki i radioèlektroniki [*Reports of the Belarusian State University of Informatics and Radioelectronics*], 2018, no. 2, pp. 77–84 (In Russ.).

13. Karasik O. N., Prihozhy A. A. *Tuning block-parallel all-pairs shortest path algorithm for efficient multi-core implementation*. *System Analysis and Applied Information Science*, 2022, no. 3, pp. 68–75. https://doi.org/10.21122/2309-4923-2022-3-57-65

14. 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, pp. 10–18. https://doi.org/10.21122/2309-4923-2019-4-10-18

15. Prihozhy A. A. *Optimization of data allocation in hierarchical memory for blocked shortest paths algorithms*. *System Analysis and Applied Information Science*, 2021, no. 3, pp. 40–50. https://doi.org/10.21122/2309-4923-2021-3-40-50

16. Likhoded N. A., Sipeyko D. S. Generalized blocked Floyd – Warshall algorithm. *Journal of the Belarusian State University. Mathematics and Informatics*, 2019, no. 3, pp. 84–92 (In Russ).

17. Kahn G. The semantics of a simple language for parallel programming. *Information Processing 74: Proceedings of the IFIP Congress 74, Stockholm, Sweden, 5–10 August 1974*. Stockholm, 1974, pp. 471–475.

18. Lee E. A., Messerschmitt D. G. Synchronous dataflow. *Proceedings of the IEEE*, September 1987, vol. 75, no. 9, pp. 1235–1245.

19. Prihozhy A., Mlynek D., Solomennik M., Mattavelli M. Techniques for optimization of net algorithms. *2002 International Conference on Parallel Computing in Electrical Engineering (PARELEC 2002), Warsaw, Poland, 22–25 September 2002*. Warsaw, 2002, pp. 211–216.

20. Eker J., Janneck J. W. *Cal Language Report : Technical Report UCB/ERL M03/48*. University of California at Berkeley, December 2003, 107 p.

21. Bhattacharyya S. S., Brebner G., Janneck J. W., Eker J., Platen C., …, Raulet M. OpenDF – a dataflow toolset for reconfigurable hardware and multicore systems. *First Swedish Workshop on Multi-Core Computing, MCC, Ronneby, Sweden, 27–28 November 2008*. Ronneby, 2008, pp. 43–49.

22. Murthy P. K., Lee E. A. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 2002, vol. 50, no. 8, pp. 2064–2079.

23. Bhattacharya B., Bhattacharyya S. S. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 2001, vol. 49, no. 10, pp. 2408–2421.

24. Bebelis V., Fradet P., Girault A., Lavigueur B. *BPDF: Boolean Parametric Data Flow : Research Report RR-8333*. INRIA, 2013, 21 p.

25. Rahman A.-H. Ab, Prihozhy A., Mattavelli M. Pipeline synthesis and optimization of FPGA-based video processing applications with CAL. *EURASIP Journal on Image and Video Processing*, vol. 2011:19, pp. 1–28. https://doi.org/10.1186/16875281-2011-19

26. Prihozhy A., Casale-Brunet S., Bezati E., Mattavelli M. Efficient dynamic optimization heuristics for dataflow pipelines. *2018 IEEE International Workshop on Signal Processing Systems, SiPS 2018, Cape Town, South Africa, 21–24 October 2018*. Cape Town, 2018, pp. 337–342.

27. Prihozhy A. A., Casale-Brunet S., Bezati E., Mattavelli M. Pipeline synthesis and optimization from branched feedback dataflow programs. *Journal of Signal Processing Systems*, Springer Nature, 2020, vol. 92, pp. 1091–1099. https://doi.org/10.1007/s11265-020-01568-5

Список использованных источников

1. Floyd, R. W. Algorithm 97: Shortest path / R. W. Floyd // Communications of the ACM. – 1962. – Vol. 5, no. 6. – P. 345.

2. Survey of Shortest-Path Algorithms / A. Madkour [et al.]. – 2017. – 26 p. – Mode of access: https://arxiv.org/abs/1705.02044. – Date of access: 23.11.2022.

3. 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 // J. of Computing in Civil Engineering. – 2013. – Vol. 27, no. 3. – P. 263–273.

4. Prihozhy, A. A. Evaluation of parallelization potential for efficient multimedia implementations: dynamic evaluation of algorithm critical path / A. A. Prihozhy, M. Mattavelli, D. Mlynek // IEEE Transactions on Circuits and Systems for Video Technology. – 2005. – Vol. 15, no. 5. – P. 593–608.

5. Singh, A. Performance analysis of Floyd Warshall algorithm vs rectangular algorithm / A. Singh, P. K. Mishra // Intern. J. of Computer Applications. – 2014. – Vol. 107, no. 16. – P. 23–27.

6. Venkataraman, G. A. Blocked all-pairs shortest paths algorithm / G. A. Venkataraman, S. Sahni, S. Mukhopadhyaya // J. of Experimental Algorithmics (JEA). – 2003. – Vol. 8. – P. 857–874.

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, no. 9. – P. 769–782.

8. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances / K. Madduri [et al.] // Proc. of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, 6 Jan. 2007. – New Orleans, 2007. – P. 23–35.

9. 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 : Atlantis Press, 2013. – P. 109–112.

10. Tang, P. Rapid development of parallel blocked all-pairs shortest paths code for multi-core computers / P. Tang // IEEE SOUTHEASTCON 2014, Lexington, KY, USA, 13–16 Mar. 2014. – Lexington, 2014. – P. 1–7.

11. 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. – 2017. – № 3. – С. 68–75. https://doi.org/10.21122/2309-4923-2017-3-68-75

12. 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.

13. Karasik, O. N. Tuning block-parallel all-pairs shortest path algorithm for efficient multi-core implementation / O. N. Karasik, A. A. Prihozhy // System Analysis and Applied Information Science. – 2022. – No. 3. – P. 68–75. https://doi.org/10.21122/2309-4923-2022-3-57-65

14. 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. https://doi.org/10.21122/2309-4923-2019-4-10-18

15. Prihozhy, A. A. Optimization of data allocation in hierarchical memory for blocked shortest paths algorithms / A. A. Prihozhy // System Analysis and Applied Information Science. – 2021. – No. 3. – P. 40–50. https://doi.org/10.21122/2309-4923-2021-3-40-50

16. Лиходед, Н. А. Обобщенный блочный алгоритм Флойда – Уоршелла / Н. А. Лиходед, Д. С. Сипейко // Журнал Бел. гос. ун-та. Математика. Информатика. – 2019. – № 3. – С. 84–92.

17. Kahn, G. The semantics of a simple language for parallel programming / G. Kahn // Information Processing 74: Proc. of the IFIP Congress 74, Stockholm, Sweden, 5–10 Aug. 1974. – Stockholm, 1974. – P. 471–475.

18. Lee, E. A. Synchronous dataflow / E. A. Lee, D. G. Messerschmitt // Proc. of the IEEE. – Sept. 1987. – Vol. 75, no. 9. – P. 1235–1245.

19. Techniques for optimization of net algorithms / A. Prihozhy [et al.] // 2002 Intern. Conf. on Parallel Computing in Electrical Engineering (PARELEC 2002), Warsaw, Poland, 22–25 Sept. 2002. – Warsaw, 2002. – P. 211–216.

20. Eker, J. Cal Language Report : Technical Report UCB/ERL M03/48 / J. Eker, J. W. Janneck. – University of California at Berkeley, Dec. 2003. – 107 p.

21. OpenDF – a dataflow toolset for reconfigurable hardware and multicore systems / S. S. Bhattacharyya [et al.] // First Swedish Workshop on Multi-Core Computing, MCC, Ronneby, Sweden, 27–28 Nov. 2008. – Ronneby, 2008. – P. 43–49.

22. Murthy, P. K. Multidimensional synchronous dataflow / P. K. Murthy, E. A. Lee // IEEE Transactions on Signal Processing. – 2002. – Vol. 50, no. 8. – P. 2064–2079.

23. Bhattacharya, B. Parameterized dataflow modeling for DSP systems / B. Bhattacharya, S. S. Bhattacharyya // IEEE Transactions on Signal Processing. – 2001. – Vol. 49, no. 10. – P. 2408–2421.

24. BPDF: Boolean Parametric Data Flow: Research Report RR-8333/ V. Bebelis [et al.]. – INRIA, 2013. – 21 p.

25. Rahman, A.-H. Ab. Pipeline synthesis and optimization of FPGA-based video processing applications with CAL / A.-H. Ab Rahman, A. Prihozhy, M. Mattavelli // EURASIP J. on Image and Video Processing. – 2011. – Vol. 2011:19. – P. 1–28. https://doi.org/10.1186/16875281-2011-19

26. Efficient dynamic optimization heuristics for dataflow pipelines / A. Prihozhy [et al.] // 2018 IEEE Intern. Workshop on Signal Processing Systems, SiPS 2018, Cape Town, South Africa, 21–24 Oct. 2018. – Cape Town, 2018. – P. 337–342.

27. Pipeline synthesis and optimization from branched feedback dataflow programs / A. A. Prihozhy [et al.] // J. of Signal Processing Systems, Springer Nature. – 2020. – Vol. 92. – P. 1091–1099. https://doi.org/10.1007/s11265-020-01568-5

Information about the author

Anatoly A. Prihozhy, D. Sc. (Eng.), Professor, Belarusian National Technical University.
E-mail: prihozhy@yahoo.com

Информация об авторе

Анатолий Алексеевич Прихожий, доктор технических наук, профессор, Белорусский национальный технический университет.
E-mail: prihozhy@yahoo.com