ISSN 0361-7688, Programming and Computer Software, 2023, Vol. 49, No. 4, pp. 310–324. © Pleiades Publishing, Ltd., 2023.
Russian Text © The Author(s), 2022, published in Programmnaya Ingeneria, 2022, Vol. 13, No. 1.

Parallel Programming

V. V. Korneev

Research and Development Institute 'Kvant',
4-i Likhachevskii per. 15, Moscow, 125438 Russia
e-mail: korv@rdi-kvant.ru

Received January 9, 2023; revised February 16, 2023; accepted March 21, 2023

Abstract—The genesis of parallel programming models is considered. It is shown that parallelism and hardware support of synchronization inherent in an architecture determine a parallel programming model. Modern VLSI technology require shift to programming models based on shared memory.

Keywords: VLSI, architecture, parallel programming model, dataflow processing, and interconnection graph

DOI: 10.1134/S0361768823040084

1. INTRODUCTION

Even though parallel computing and parallel programming models are now being used in all areas of human activity, the adoption of a particular model is not realized as a technological shift that changes the architecture of computer systems and, as a result, affects the parallel programming model. The architecture of a system determines the possibility of parallel use of computer resources, while the programming model determines the possibility of effective mapping of a program to hardware resources.

This paper considers the genesis of parallel programming models and substantiates the need for a transition to a programming model with shared memory and synchronization based on FE-bits of shared memory words.

2. UNIVERSAL HOMOGENEOUS HIGH-PERFORMANCE COMPUTER SYSTEMS

Parallel data processing, as a means of performance boost unattainable for a single computer, arose from solving challenging computationally hard problems on specialized computer systems. In the early 1960s, at the Sobolev Institute for Mathematics of the Siberian Branch of the Russian Academy of Sciences (former IM SB AS USSR), a research has begun on the architecture of computer systems and computational environments, parallel algorithms and parallel programming tools, as well as computer graphics and pattern recognition, which is now part of artificial intelligence. The initial stage of the research into the design and use of high-performance computer systems resulted in the publication (in 1966) of the monograph [1]. Let us consider the results that were summarized in the monograph into a unified concept based on a model of a collective of computers; these results are of fundamental importance for the emergence and development of parallel programming.

For the first time, based on the analysis of physical constraints imposed on the development of microelectronics, some approaches to improve the performance of computer systems were formulated, which included

- higher clock frequency;
- increased number of processing devices that operate simultaneously (in parallel);
- programmability of structure as a software-based setting of connections among processing devices.

In this case, programmability of structure was considered as a means of attaining universality, which makes it possible to achieve high performance in executing different algorithms owing to the software implementation of specialized computer systems for their execution.

For a computer to operate at a clock frequency ν, the maximum length of conductors between devices should not exceed $d = c/2\xi\nu$, where c is the speed of light, ν is the clock frequency of a unified clock tree of the computer, c/ν is the wavelength corresponding to ν, and ξ is the coefficient that determines the excess of the wavelength over the conductor's length (the constant ranging from 10 to 100). Suppose that ρ is the maximum permissible number of devices per unit volume, which is determined by a hardware manufacturing technology and limits set by heat and energy exchange conditions. Then, the number of devices that can be placed in a ball with a diameter of $2d$ does not exceed $n = 2\pi\rho c^3/3\xi^3\nu^3$. The performance of a computer is directly proportional to the product of the number of devices by their clock frequency. Thus, the

maximum performance of computing devices with a unified clock tree has a fundamental limit.

The performance model considered above implies that, to achieve arbitrarily high performance, it is required to abandon the unified clock tree and redesign the system as a collective of synchronous computers interconnected by asynchronous channels. In that case, the constraint on the maximum number of devices holds only for each individual computer, whereas the number of computers in the system is not limited. This makes it possible to achieve desired performance, which happened in the 21st century. Currently, computer systems are designed based on architectures of the GALS (globally asynchronous, locally synchronous) class, which implement global asynchrony among synchronous computers.

However, the proven possibility of building computer systems with an arbitrarily large number of computers (elementary computers (ECs) in terms of [1]) did not mean the possibility of executing programs with a performance proportional to the number of ECs. To investigate the limit on the performance increase achievable by using parallel computing, the following reasoning (later known as Amdahl's law) was used: if a program has a non-parallelizable part σ and an ideally parallelizable part $(1 - \sigma)$, then the speedup due to parallelization when running this program on m processors is $S = 1/(\sigma + (1 - \sigma)/m)$. In the limiting case, as m tends to infinity, the speedup is $S = 1/\sigma$.

Thus, on the one hand, there are no algorithmic approaches to the design of parallel computer systems and to the development of parallel programs with proven execution efficiency. On the other hand, there are precedents of building specialized parallel systems based on parallel algorithms to execute computationally hard problems. Hence, it was decided to create an experimental parallel computer system Minsk-222 and use it as a basis for investigating parallel programs for some important computational tasks and their most common parts that implement computational methods, e.g., for solving systems of linear equations. In the framework of that project, it was supposed to create efficient algorithms for program parallelization, including automatic parallelization.

The Minsk-222 computer system was built on the basis of a Minsk-22 shelf serial computer by equipping each computer with a system engine (SE). The SE had two control links and two information links for connection to the corresponding links of two neighbor elementary computers. Minsk-222 could consist of up to 16 ECs, each having its own number i, $0 \le i \le N - 1$, where N is the number of ECs in the system. EC with number i was connected via bidirectional control and information links to ECs with numbers k and j, $k = (i - 1) \bmod N$ and $j = (i + 1) \bmod N$. Thus, the structure of computer-to-computer connections with the ring topology was formed.

Each SE had a configuration register, which consisted of triggers *TR*, *TQ*, *TΩ*, as well as a block of system operations, which extended the instruction set of the base computer with the so-called system instructions. These system instructions were divided into four categories:

• configuration instructions, which set configuration registers and registers of EC features;

• generalized unconditional jump (GUJ) instructions, which were used for the initial load of programs and data to the ECs with the TQ trigger set to "1" and for the forced control of the computational process in these computers (the state "1" or "0" of the TQ trigger determines, respectively, the execution or skipping of an instruction that comes from the EC executing the GUJ instruction);

• generalized conditional jump (GCJ) instructions, which execute jumps to specified addresses if

$$\Omega_k = \underset{i \in E}{\&} \omega_{ki}, \quad k = 1, 2, 3,$$

where E is a subset of index numbers for the machines with the TΩ triggers preset to "1" and selected feature ω_k, $k \in \{1, 2, 3\}$ which is used to generate a generalized feature Ω_k, while ω_{ki} is a feature (the result is less than zero, overflow, the result is zero) generated by the i-th EC;

• instruction T(z, A) used to transmit *z* memory words, beginning with the address A specified in the instruction, and instruction R(w, B) used to receive *w* memory words with their allocation in internal memory, beginning with the address B specified in the instruction. The R(w,B) process continues until all *w* words are received; only then the instruction following the R instruction is executed.

Setting to "1" the TR triggers in the ECs with numbers i and $i + n \bmod N$, as well as setting to "0" the TR triggers in the ECs with numbers $i + 1 \bmod N$, ..., $i + n - 1 \bmod N$, is used to form a subsystem of n ECs with numbers i, $i + 1 \bmod N$, ..., $i + n - 1 \bmod N$. Thus, the system could be divided into a set of subsystems operating as independent systems.

The architecture of Minsk-222 defined a parallel programming model based on message passing and distributed memory. Through the efforts of specialists in algorithms and programmers from the IM SB AS USSR, parallel programs tunable to a given number of ECs (as to a certain parameter) were developed. It became possible to create such programs for practically important computational tasks, as well as to achieve their speedup that was directly proportional to the number of computers with a certain multiplying factor. The doubled memory capacity of the Minsk-222 system, as compared to the Minsk-22 individual computer with slow external magnetic-tape data storage, as well as the speed of its communication links (comparable to that of memory), ensured its performance that significantly exceeded the "mechanical" sum of the performances of the constituting computers.

A parallel program was represented as a set of branches with data and control operators for their interaction. A fundamental result from the creation of these parallel programs was the development of typical schemes for data exchange between program branches. These schemes are reduced to exchanges of five types: broadcast, broadcast-cyclic, pipeline-parallel, gathering, and differentiated exchanges. In the case of the broadcast exchange, the same data block is transmitted from one (any) branch simultaneously to all other branches of a program. The broadcast-cyclic exchange transmits a data block from each branch to all others. The pipeline-parallel exchange passes a data block from each branch i, executed on the i-th EC, to a neighbor branch k, $k = (i + 1) \bmod B$, where B is the number of branches in a parallel program. The gathering exchange implements collection of data blocks from some branches of a program into one branch. Finally, the differentiated exchange passes data blocks between specified pairs of branches or from one branch to several branches.

In the mid-1990s, the programming model based on message passing became generally accepted. It was represented as a message passing interface (MPI), a library of functions for creating and executing parallel programs. There is a following correspondence between the Minsk-222 parallel programming system and the MPI:

- broadcast exchange (MPI_Bcast);
- broadcast-cyclic exchange (MPI_Alltoall or MPI_Allscatter);
- gathering exchange (MPI_Gather);
- generalized conditional jump (MPI_Barrier);
- differentiated exchange (MPI_Send, MPI_Recv).

Thus, it can be stated that the monograph [1] proposed the promising (eventually fundamental) architectural concept for parallel computer systems with distributed memory and the parallel programming model based on message passing.

Since the 1970s, attempts were made to create parallel systems based on this concept. For instance, there was a project to interconnect high performance computers of Unified System, manufactured in the USSR; however, even in a large organization, there was only one computer, and interconnection of computers within the country was not just a technical problem. In addition, available electronic components made it possible to create more efficient computers with a unified synchronization tree and increased clock frequency, since the number of elements in them was far from the limit set by the packing density of elements and clock frequency.

The situation changed with the advent of microprocessors. A computer system MVS-1000M [2] was created, with a performance at a level of 10^{12} flops. It was the first domestically developed system included in the top 50 of the TOP500 list of the world's highest-performance computer systems. The architecture of MVS-1000M provides only system operations for transmitting and receiving data blocks. As an application programming interface, users can call programs from the MPI library. To some extent, this is due to the impossibility to add system instructions, as it is done in Minsk-222, to the microprocessor. However, the software implementation of barrier synchronization in the MPI, in contrast to the GCJ system instruction in Minsk-222, is rather time-consuming and is one of the main causes of performance loss in parallel computing. That is why a special communication network was introduced, e.g., in IBM Blue Gene\L to implement an analogue of the GCJ in Minsk-222.

3. COMPUTER SYSTEMS WITH PROGRAMMABLE STRUCTURE

In the early 1970s, some theoretical questions arose concerning the fundamental contradiction between the implementation of the Minsk-222 architecture and the model of parallel systems from [1]. One of the questions was that the implementation implied "long" conductors, e.g., when executing the GCJ instruction, which, with increasing number of computers in the system, should have lead to a decrease in clock frequency. Moreover, the T and R instructions implied the existence of connections between any ECs in the system, no matter how large-scale it was, i.e., the conductors should be long. However, the architectures of the GALS class should not have long conductors. Each EC must have connections only with a limited number of neighbor ECs; to pass data to a non-neighbor EC, a sequence of data passes between neighbor ECs must be carried out. Hence, it was necessary to understand how to build computer systems that only have short local conductors between ECs, as well as understand how to program and efficiently run parallel programs with the performance directly proportional to the number of ECs used. The results of investigating these and related questions were considered in a monograph published in 1985 [3]. Below, we discuss the proposed architecture and approach to parallel programming with local connections.

A potentially unlimited number of ECs interconnected into a system can only be achieved by using a spatially distributed switch of computer-to-computer connections with decentralized distributed control. In any other case, the structure of the system can be represented as a set of computers connected by conductors to one switch, which contradicts the potentially unlimited number of computers at a constant clock frequency.

A spatial NxN switch (N inputs and N outputs) with decentralized control is built of N non-ordinary switches with v + 1 inputs $\mathbf{in}_i$ and outputs $\mathbf{out}_i$ (i = 0, 1, ..., v). An example of such a switch with eight inputs and outputs, which is build on switches with v = 3, is

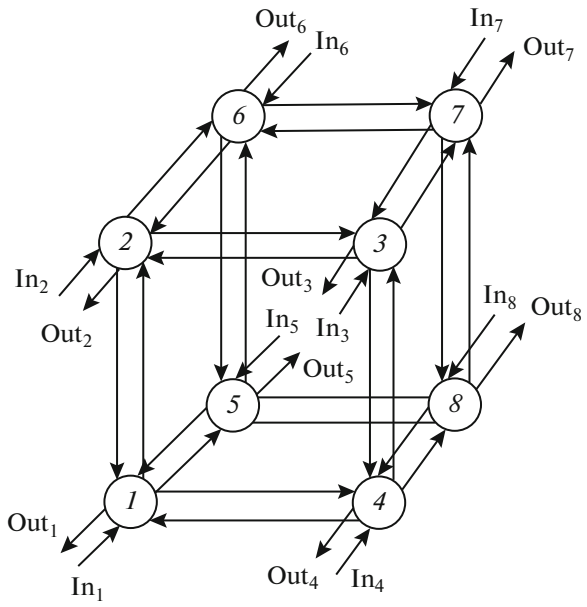

Fig. 1. Spatial 8x8 switch.

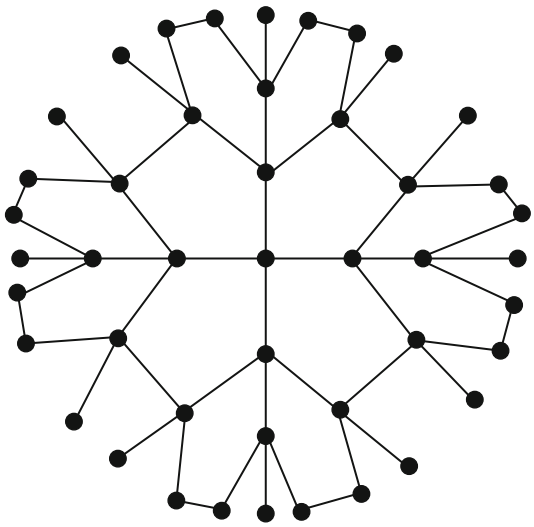

Fig. 2. First three layers of infinite planar graph L(4, 5).

shown in Fig. 1. In this case, $\mathbf{in}_0$ and $\mathbf{out}_0$ of each of the N non-ordinary switches j (j = 1, ..., N) serve as In_j and Out_j of the spatial switch.

When creating a computer system of N computers and switches with v + 1 inputs $\mathbf{in}_i$ and outputs $\mathbf{out}_i$ (i = 0, 1, ..., v), it is required to choose a graph of computer-to-computer connections. Since there are various non-isomorphic homogeneous graphs with the number of nodes N and degree of nodes v, it is necessary to choose an optimal graph in terms of computer-to-computer connections. A hypothesis was put forward and confirmed by statistical experiments [3], that, among the graphs with the number of nodes N and degree of nodes v, the optimal graph has the minimum diameter and minimum average distance between nodes.

As computer interconnection graphs, a parametric family of homogeneous graphs L(N, v, g) was proposed, where N is the number of nodes, v is the degree of nodes, g is the girth, and each node is included in v cycles of length g.

Graph L(N, v, g) is constructed based on an N-node subgraph of infinite planar graph L(v, g) by selecting the corresponding subgraph from all existing ones and by drawing edges to construct L(N, v, g). In this case, the N-node subgraph includes

• all nodes of d layers of L(v, g) if $\sum_{i=0}^{d} Ni = N$, where Ni is the number of nodes at layer i (i = 0, ..., d), $N_0 = 1$, $N_1 = v$, and d is the diameter of L(N, v, g);

• all nodes of d – 1 layers of L(v, g) if $\sum_{i=0}^{d-1} Ni < N$ and $N - \sum_{i=0}^{d-1} Ni$ nodes of layer d.

The algorithm [3] searches through the possible variants and either finds a suitable graph or reports the impossibility of its construction. Figure 2 shows three layers of infinite planar graph L(4, 5).

Graphs L(N, v, g) exist not for all value combinations of parameters N, v, and g. For instance, the exhaustive search algorithm made it possible to construct graphs L(N, 4, 5) for N = 19, ..., 45.

Figure 3 shows graphs L(10, 3, 5) and L(24, 4, 5).

Graph L(10, 3, 5) has diameter 2 and includes all nodes of layers 0, 1, and 2 of graph L(3, 5). Graph L(24, 4, 5) has diameter 3 and includes all nodes of layers 0, 1, and 2 of graph L(4, 5), as well as sever nodes of layer 3 of this graph.

Graphs L(N, 4, 4) are group-graphs of finite abelian groups of order N with two generators. They exist for all values of N.

Graph L(N, v, g) has the minimum diameter d and minimum average distance between nodes d_{avg} if g = 2d + 1 or g = 2d among all graphs with N nodes and node degrees v.

A possible statement of the problem of finding the optimal interconnection graph is to impose a constraint on the diameter (the number of hops, intermediate transmissions of packets). In this case, switches with v ≥ 64 are used. In [4], some algorithms for synthesizing graphs similar to L(N, v, 5) and L(N, v, 7) with diameters 2 and 3, respectively, were considered.

In [5], results of statistical modeling were presented that proved the optimality of L(N, v, g) with minimum d and d_{avg}, including the efficiency of implementing MPI library functions on computer systems with interconnection graphs Dragonfly and L(N, v, 7) with the same number of nodes and node degrees. Using an original exhaustive search algorithm [5], graphs L(20,

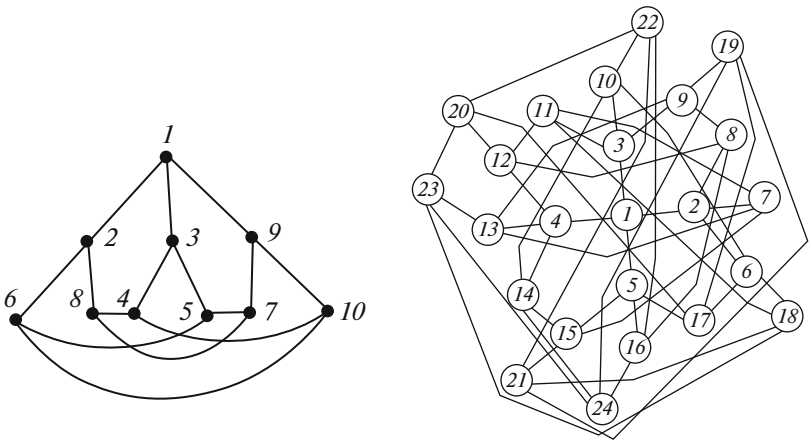

Fig. 3. Graphs L(10, 3, 5) and L(24, 4, 5).

4, 7), L(30, 5, 7), L(36, 5, 7), L(252, 11, 7), and L(264, 11, 7) with diameter 3 were synthesized.

Below are some important practical aspects of using L(N, v, g) with minimum d and d_{avg} as an interconnection graph [3]:

• the maximization of the probability that there is a connected subgraph of all non-failed computers, for given probabilities of computer and communication link failures, among all graphs with the number of nodes N and degree of nodes v (structural robustness);

• the maximization of the probability of simultaneously established connections between individual computers and/or groups of computers (structural switchability).

The architecture with local connections makes it possible to directly pass a data block only between neighbor computers connected by a communication link. The EC structure of a computer system with graph L(N, 4, 4) is shown in Fig. 4.

In the transmitter computer, the data block is placed in the output queue of the communication link. If there is a place in the input queue of this communication link to receive a data block, then it is added to the input queue. If there is no place in the input queue, then the transmission is delayed until the input queue is freed.

The transmission of a data block between computers not connected by a communication link and the execution of the GCJ, as well as the GUJ, is carried out via neighbor computers. The creation of subsystems (programming the structure of the system) is carried out using system software. From a programmer's perspective, the structure for which a parallel program is created is selected among structures presented in Fig. 5, or some other structure is selected. It is important that this structure have a parametric description and can be selected with any required number of computers. It should be noted that a root tree can be created on a connected subgraph for any interconnection graph L(N, v, g). In this case, the line and ring structures have certain constraints on their construction, while the lattice is constructed without the use of transit computers only on L(N, v, 4).

To ensure programmability, the computers must be enumerated. We assume that, when creating the subsystem, a root computer is selected and assigned number 0. The enumeration of computers for the line and ring subsystems is obvious; for the lattice subsystem, there are options for rows and columns. We assume that the computers of the root tree subsystem are enumerated by the depth-first search algorithm taking into account the same order of pole numbers for all switches

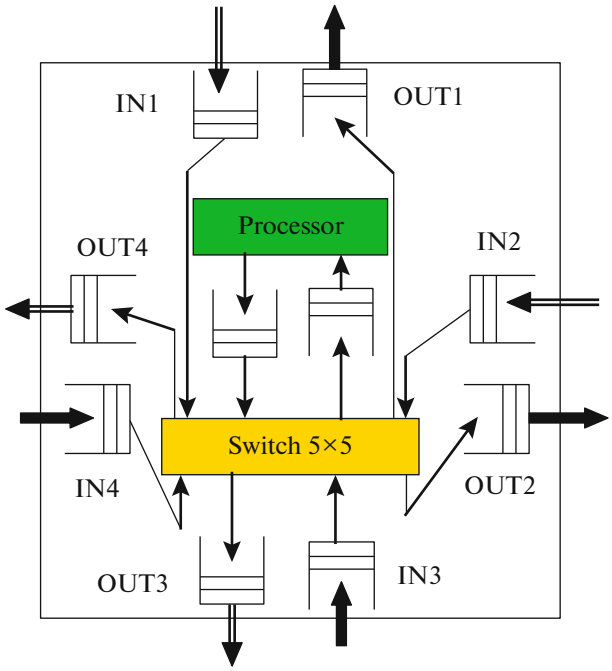

Fig. 4. EC structure of a computer system with graph L(N, 4, 4).

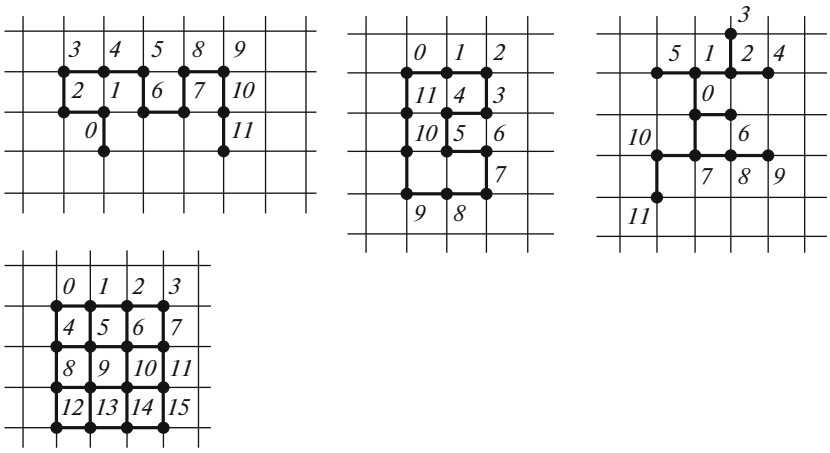

Fig. 5. Structures of the line, ring, root tree, and lattice subsystems.

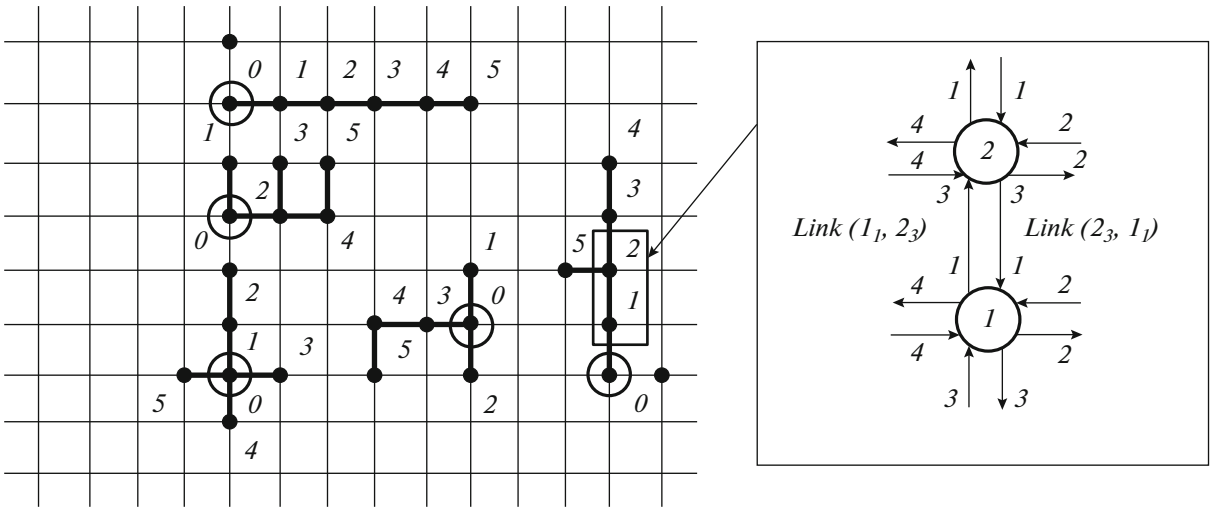

Fig. 6. Subsystems of six computers with specified numbers (tree roots are encircled).

that constitute the spatial switch of the system. Examples of this enumeration are shown in Fig. 6.

A parallel program must define a data flow between locally interacting computers, which locally execute their parallel branches. Each data exchange between branches of a parallel program is carried out according to one of the schemes considered above (broadcast, broadcast-cyclic, pipeline-parallel, gathering, or differentiated). That is why a parallel program must consist of two interacting components: calculation procedure and path procedure.

Let us consider these procedures in detail. Suppose that it is required to use the broadcast-cyclic exchange scheme. Figure 7 shows a subsystem of six computers with the root tree structure, including the bottom-up and top-down networks of the subsystems that are formed to implement this exchange scheme. For the subsystem in Fig. 7, by τ_i we denote the weight of node i, which is equal to the number of nodes in the subtree the root of which is this node.

The path procedure that implements the broadcast-cyclic exchange determines the operation of the bottom-up and top-down networks of the subsystem on the n + 1 computers of which the parallel program is executed.

In the bottom-up network, each EC_i (i = 1, ..., n) loads (in accordance with the cyclic discipline described below) the queue $Out_{e(j)}$ of link (i, j), which transmits a data unit to the EC_j that is its successor in the bottom-up network; here, e(j) is the number of a switch pole of link (i, j), e(j) ∈ {1,..., v}. The first data unit to be transmitted is the one extracted from the beginning of queue Out_0, which is generated in EC_i itself (i = 1, ..., n). Then, τ_{i1} data units are added to $Out_{e(j)}$, each of which being extracted from the beginning of the queue In_{i1} containing the data units passed

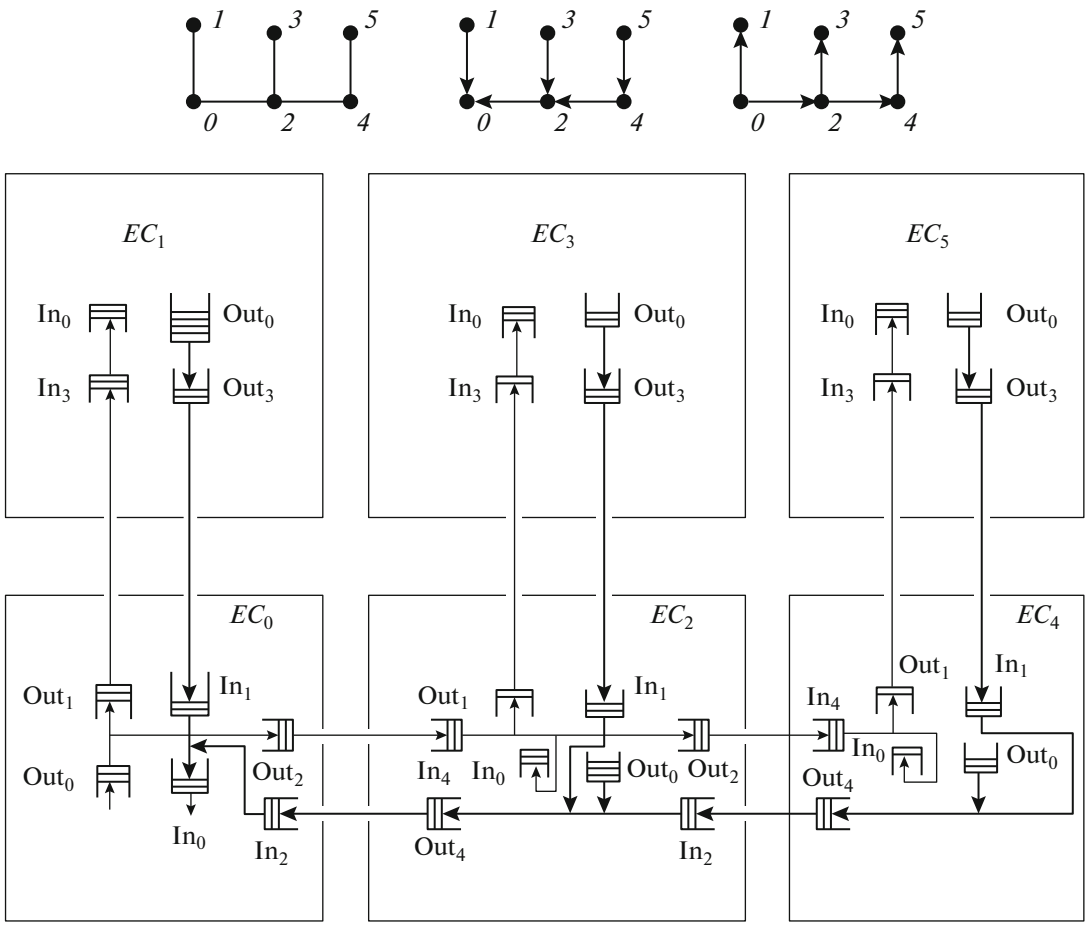

Fig. 7. Subsystem of six computers with the root tree structure.

to EC_i from EC_{i1}. EC_{i1} is a predecessor of EC_i and has the lowest number i1 among all computers EC_{i1},..., EC_{ip}, which are predecessors of EC_i in the bottom-up network ($i_1<...<i_p$). Upon adding τ_{i1} data units from queue In_{i1}, τ_{i2} data units from queue In_{i2} are added to $Out_{e(j)}$ and so on until τ_{ip} from queue In_{ip} are added. Then, this sequence of actions is repeated, starting with the transmission of a data unit extracted from the beginning of Out_0.

Root computer EC_0 loads queue In_0 by cyclically extracting (based on the scheme described above) data units from queues In_{01}, In_{02}, ..., In_{0s}, which are passed to EC_0 by its predecessors EC_{01},..., EC_{0s}.

The operation of the top-down network is initiated by EC_0. From the beginning of queue Out_0, a data unit generated by a calculation procedure in EC_0 is extracted and loaded to the end of Out_{01}, Out_{02}, ..., Out_{0s} for transmission via the top-down network from EC_0 to its successors EC_{01},..., EC_{0s}. Each EC_i in the top-down network adds the data unit extracted from the beginning of $In_{e(j)}$, which arrived from predecessor EC_j, to its queue In_0 and output queues, which pass the data unit to successor computers.

The calculation procedure executed by each computer takes data units from In_0 and, after their processing by executing the calculation algorithm, puts the result in Out_0. If some EC has not completed the calculation procedure by the time it can load the data unit, then the transmission is delayed; it is also delayed if there is no place in some queue to receive the data unit.

The joint operation of the bottom-up and top-down networks results in loading sequence x_1^0, x_2^0, ..., x_n^0, x_1^1, ..., x_n^1, x_1^2, ... to In_0 in each of n + 1 computers of the subsystem.

Using the scheme of broadcast-cyclic exchange described above, it is possible, by slightly modifying it and implementing the corresponding calculation procedure, to build parallel programs that use other exchange schemes. Differentiated exchanges can be implemented using modified bottom-up and top-down networks, as in the broadcast-cyclic scheme. The modification of the bottom-up network consists

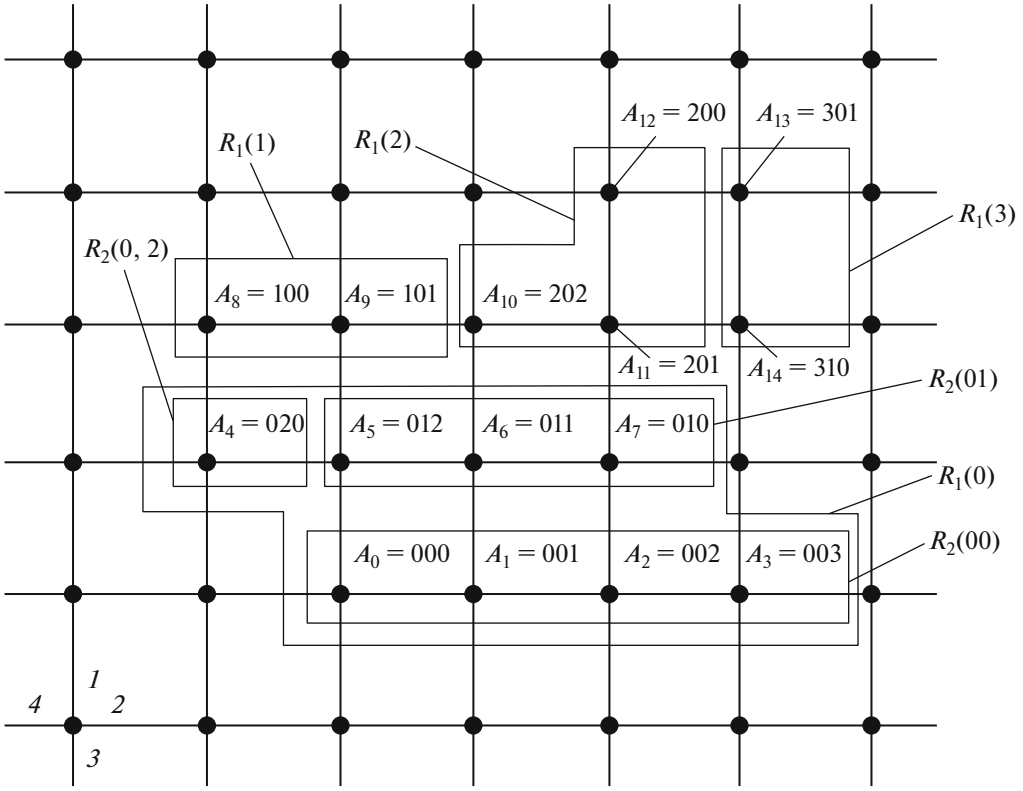

Fig. 8. Subsystem of 15 computers with D(2, 3)-addressing.

in cyclical polling of all queues that supply data units (while skipping empty ones). The data units are provided with tags, which consist of the number of an EC that generated them and the numbers of receiver ECs.

Another implementation of inter-computer exchanges is based on addressing the computers of a subsystem that executes branches of a parallel program.

The implementation of differentiated exchanges can use the assignment of computer addresses and the transmission of data units to receivers along the shortest paths. There is coordinate addressing, which is used in computer systems with L(N, v, 4) or multidimensional cubic structures as interconnection graphs (in which an EC address is given by coordinates in each direction). When passing a data unit, it is required to compare the coordinates of a destination address and the coordinates of an EC in which this comparison is performed. If the coordinates match, then the destination is reached. Among the transmission directions with mismatched coordinates, the direction with the minimum mismatch is selected. This is efficient addressing. It is used in some supercomputers, e.g., those built by Cray Inc.

For computer systems with interconnection graph L(N, v, g), D(z, m)-addressing is proposed. In this case, the address of EC_i, $i \in \{0, ..., n\text{-}1\}$, is also given by vector $A_i = A_{i\,0}, ..., A_{i\,m\text{-}1}$. Each A_{ij} $(j = 0, ..., m-1)$ takes a value from set $\{0, 1, ..., 2^z\text{-}1\}$, $z \in \{1, 2, ...\}$, n is the number of computers in a subsystem, $mz \geq \lceil \log_2 n \rceil$, and $\lceil x \rceil$ is the minimum integer such that $\lceil x \rceil \geq x$. EC_i and EC_k, i and $k \in \{0, 1, ..., n\text{-}1\}$, belong to subsystem $R_r(A_{i\,0}, ..., A_{i\,r-1})$ of layer r if $A_{i\,r} \neq A_{k\,r}$ and $A_{i\,j} = A_{k\,j}$ for all $j < r$ $(r = 1, 2, ..., m)$. $R_0()$ is a zero-layer subsystem (the entire subsystem of n computers) and $R_m(A_{i\,0}, ..., A_{i\,m-1})$ is an m-layer subsystem (which consists of one EC_i). Computer addressing must satisfy the following properties:

• computers of each subsystem $R_r(A_{i\,0}, ..., A_{i\,r\text{-}1})$ of layer r $(r = 1, 2, ..., m)$, together with communication links between them, must form a connected subgraph of the interconnection graph;

• $R_0 \supseteq R_1(A_{i\,0}) \supseteq R_2(A_{i\,0}, A_{i\,1}) \supseteq \supseteq R_m(A_{i\,0}, ..., A_{i\,m\text{-}1})$;

• $R_r(A_{i,\,0}, A_{i,\,1}, ..., A_{i,\,r-1}) = \bigcup_{j=0}^{d} R_{r+1}(A_{i,\,0}, A_{i,\,1}, ..., A_{i,\,r-1}, j)$, $d = 2^z - 1$;

• $R_{r+1}(A_{i\,0}, ..., A_{i\,r-1}, j) \cap R_{r+1}(A_{i\,0}, ..., A_{i\,r-1}, k) = \varnothing$, $j \neq k$.

A subsystem of 15 computers with D(2, 3)-addressing is shown in Fig. 8.

In the case of D(z, m)-addressing, each computer EC_i, $i \in \{0, 1, ..., n-1\}$, must store only $m2^z$ numbers $p_1(0), p_1(1), ..., p_1(2^z-1), p_2(0), p_2(1), ..., p_2(2^z-1), ..., p_m(0), p_m(1), ..., p_m(2^z-1)$ of output poles of EC_i that

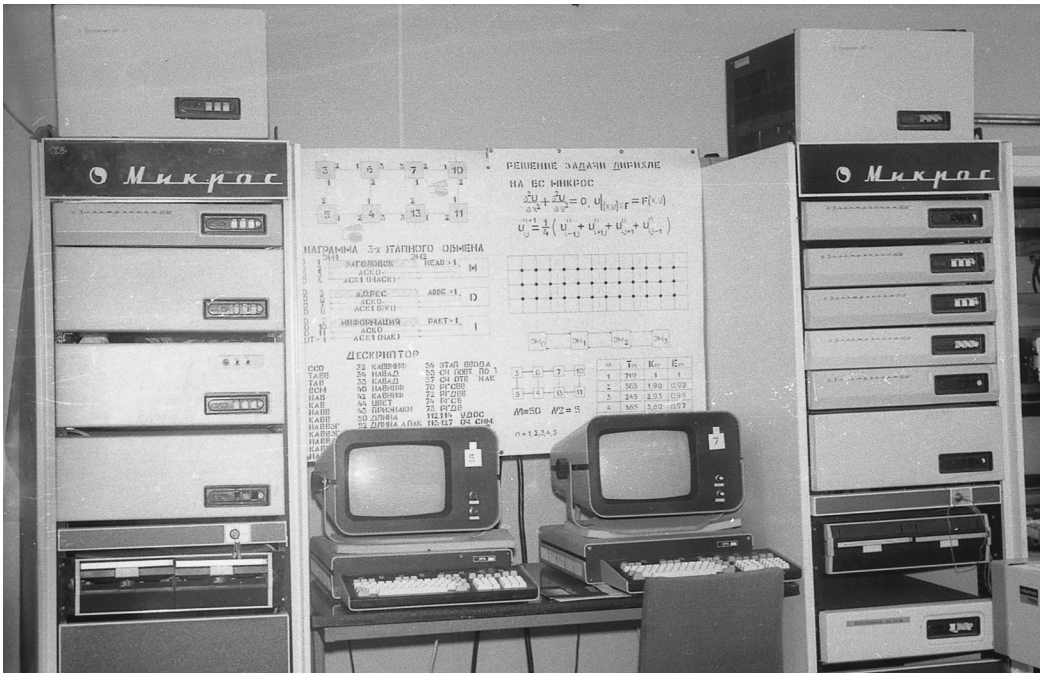

Fig. 9. Computer system MICROS with programmable structure (1986).

belong to the shortest paths from EC_i to (respectively) subsystems $R_1(0)$, $R_1(1)$, ..., $R_1(2^z-1)$, $R_2(A_{i\,0}, 0)$, ..., $R_2(A_{i\,0}, 2^z-1)$,..., $R_m(A_{i\,0}, ..., A_{i\,m-2}, 2^z-1)$. The shortest path to a subsystem is a path to the nearest computer of this subsystem.

With D(z, m)-addressing, the determination of a receiver reach and the selection of a direction for further transmission are performed by the same algorithm as with coordinate addressing, except that the selection of the transmission direction takes into account the mismatch of the coordinate with the minimum number. Transmissions are performed along paths close to the shortest ones, since directions to subsystems rather than to a specified receiver are stored.

Based on D(z, m)-addressing, path procedures for the efficient implementation of the broadcast, broadcast-cyclic, pipeline-parallel, and gathering schemes can be constructed.

Using the path and calculation procedures considered above, parallel programs for a number of computational methods were created with the acceleration of computations being proportional to the number of computers used in the computer system with local connections. This result was achieved by programming the structure required to implement the exchange schemes of an executable parallel algorithm. Hence, it is reasonable to refer to computer systems that have architecture with local connections as computer systems with programmable structure.

To test the whole complex of ideas on creating software and hardware for combining computers into systems, decentralized distributed operating system, structure programming tools, parallel programs, and a computer system with programmable structure MICROS was implemented. The system was built on the basis of Electronics-60 shelf serial microcomputers. The system is shown in Fig. 9. Experimental studies carried out on the MICROS system demonstrated the implementability of the basic aspects of the concept of computer systems with programmable structure, which was confirmed by an act of the commission of the SB AS USSR approved by the chairman of the Presidium of the SB AS USSR.

Presently, parallel computations with local connections between computers, which are also known as systolic and wave computations, are implemented mostly in specialized computers with fixed structure. Modern supercomputers built on VLSIs with a small number (within hundreds) of processing elements are generally programmed using the MPI. However, it appears that, when using VLSIs with 10^3 or more processing elements, programmability of structure becomes practically useful as a means of achieving the universality of parallel programming. It is possible that programmability of structure will be implemented in the MPI, and the application programmer will use a common interface.

4. DATAFLOW COMPUTER SYSTEMS WITH SHARED MEMORY

According to the modern trends in the development of electronic components and architectures, a supercomputer with exascale performance must execute approximately 10^9 and more processes with single-process performance being approximately 10^9 flops. However, the practice of using modern computer systems from the TOP500 list shows that, already for computer systems with millions of processes, there are obstacles to their efficient use and further increase in their performance because of processor downtime due to waiting for the completion of data read/write operations (the so-called memory wall).

Conceptually, the gap between the time of instruction execution in the processor and the memory access time can be alleviated when creating programs that use all parallelism inherent in the algorithm up to the smallest granules. In this case, while waiting for one process to complete its memory access, it becomes possible to execute other processes parallel to it [6], i.e., to work with memory at the temp of servicing the request flow. To reduce performance losses when the processor switches from the execution of one process to the selection and initiation of execution of another, lightweight threads (q-threads) [6], which have minimal context, were proposed. With this approach, a program for an exascale supercomputer must be a description of generation of billions of threads, as well as interthread communication and synchronization. The compiler (statically) and libraries (dynamically at runtime) transform this program into a set of threads, while determining the resources on which these threads are executed or the mapping of threads to resources in the runtime system.

Mappings can be used for asynchronous threads that run in universal processors and have their own individual instruction counter, as well as for synchronous threads executed on one instruction counter in different ALUs. For instance, NVIDIA Fermi GPGPU can synchronously run up to 512 threads in each clock cycle. Synchronous threads are a hardware-efficient alternative to asynchronous threads; however, they are more difficult to program [7]. Thus, the program is executed on heterogeneous resources. It is necessary to ensure the optimal allocation of the heterogeneous resources available to a task to the task's threads. The ultimate goal is automatic allocation; however, one should begin with creating an API of a library for generating a required collection of resources and allocating these resources to threads. It is necessary to be able to identify heterogeneous resources, e.g., devices for executing bundles of synchronous threads, as well as to select threads in a program that should be executed in synchronous mode on these devices. When mapping threads to resources, interthread communication must be organized using the most suitable communication resource and resource utilization modes (generation of message flows, synthesis of messages of specified lengths by combining several messages, etc.).

The existing programming model based on message passing does not allow one to efficiently use potential parallelism of calculation algorithms. The creation of other exascale programming models is necessary to

— efficiently use the significantly increased parallelism of processing (10^9 and more processors);

— facilitate the development of parallel programs while ensuring high degree of parallelism, i.e., avoid reducing the productivity of their development.

Moreover, new models should multiply this productivity as compared to its current level. This should be promoted by the use of globally addressable memory, as well as by a raise in the level and non-procedurality of parallel programming tools, which are characteristic of new models.

As a basis for an exascale programming model that satisfies the above requirements, the parallel random access model (PRAM) [8] can be used. This model is based on a multiprocessor architecture with shared memory. Processors P_i ($i = 1, ..., n$) have access to distributed shared memory (DSM) or partitioned global address space (PGAS), which is physically distributed over computer modules (processor + memory block) but has a common address space and is logically accessible to all processors. When creating a program, the user assumes that flow instructions in each processor are executed synchronously (the execution time of all instructions is the same), while interprocess communication and synchronization are carried out through shared memory cells.

There are various approaches to resolving conflicts between different processors when accessing the same shared memory cell [8]. In the context of this discussion, we consider the exclusive-read exclusive-write (EREW) approach (whereby only one processor can read from any one memory location at one time, and only one processor can write to any one memory location at one time). In this case, simultaneous read accesses precede write accesses.

In this parallel programming model, the user needs to specify which computations can be carried out in parallel taking into account only the chosen algorithm. This helps to identify all parallelism inherent in the algorithm and improves programming productivity [8].

The mechanism of resolving conflicts in accessing one EREW cell is implemented by memory access control hardware. Historically, it was the lack of an effective scalable hardware solution for resolving memory access conflicts that led to the abandonment of PRAM and the adoption of the message passing model. However, it seems that this solution presently exists.

Let us separately consider the linguistic means of generating and terminating threads and the means of interthread communication and synchronization in the framework of the proposed PRAM extension [8].

As a parallel programming language to implement the PRAM model, the extension [8] of the C language can be used. It has three additional functions for generating and terminating asynchronous threads: spawn (a, n), join, and fetch_and_add (e, x). These functions make it possible (respectively) to

• generate threads (the number of which is specified by parameter n) sequentially enumerated beginning with number a;

• terminate a thread that executes *join* and jump to the next statement if all threads generated by the corresponding *spawn* are terminated;

• execute an atomic operation of assigning the value of parameter x to a variable (e.g., thread number) and increment the value of x by adding value e to x.

An example of parallel programming for the problem of generating an array B that consists of non-zero elements of another array A is presented below [8]. Symbol $ denotes the number of the current thread.

```
int x;
x=0;
spawn(1, n) {
int e;
e=1;
if (A[$]!=0) {
a= fetch_and_add(e, x)
B[a]=A[$];
}
}
```

The implementation of PRAM implies that the EC should be a "standard" multithreaded processor running under an operating system (OS) of the Unix family.

The EC under Minix3 OS can natively implement algorithms for program parallelization into multiple threads, thread mapping, core load control, etc. With native parallelization, code fragments of a program are assigned with very low overhead to many execution cores [9]. In this case, the cores receive a pointer to a code fragment passed to them for execution. The results are stored in shared memory. In relatively conventional terms, this is called an approach based on lightweight threads. This approach is implemented, e.g., in the Qthreads library [6] created by Sandia National Laboraroties. Its main idea is to provide a mechanism for generating parallel executable threads at the program level rather than at the OS level. Ideally, the overhead of calling such a thread is comparable to the overhead of calling a regular function. In the framework of the Qthread formalism, the generation of a thread is described by a call of the form

```
void start_thread(int(*)(int), int, …)
```

The call can be made from any core. The initial function needs to be passed a pointer to the executable function, the number of its arguments, and arguments themselves (if any). Together with atomic non-blocking increment operations and some other standard operations, this call makes it possible to spawn and join groups of parallel threads, thus creating a serial-parallel program.

Synchronization based on OS primitives, as in Unix processes and p-threads [10], causes significant latency. In the proposed model, the computations carried out by each processor are represented by lightweight threads. To synchronize threads, a mechanism of interthread synchronization and communication based on shared memory is proposed. Its idea is to add a bit with full/empty value (FE-bit) to each memory word and use, together with traditional instructions, synchronization instructions for memory access [11]. Memory access instructions writeef, readfe, readff, and writeff can only be executed if the value of the FE-bit of a memory word matches with the first component of their suffix; upon executing the memory access, they set the value of this bit defined by the programmer in the second component of the suffix. Instruction execution is delayed if the FE-bit does not have the required value. For instance, writeff requires that, before its execution, the FE-bit of the memory word to be written have value full; after the execution, it preserves the same value. The value of the FE bit usually determines whether the memory cell is full or empty.

Synchronization based on FE-bits requires neither special shared variables, such as locks, semaphores, etc., nor synchronization mechanisms, which are very expensive (in terms of time) to define and execute with millions of threads [6, 11]. In addition, the use of shared variables and atomic sequences of instructions that determine the value of the branch condition and execute the jump in accordance with a received value is much more difficult and time-consuming than the execution of memory access instructions when synchronizing dynamically generated lightweight threads. Thus, synchronization based on FE-bits makes it possible to use the maximum possible (in an executable program) number of memory requests generated by lightweight threads, as well as to execute these requests in parallel in interleaved memory.

It should be noted that, in CrayXMT [11], the C extension that enables synchronization based on FE-bits implies the introduction of synchronization variables x$ and generic functions for read and write operations. Among them, the following ones should be mentioned:

• purge x$ is the assignment of an empty value to FE-bit x$;

• writeqr x$, g means the write of value g to x$ if the value of FE-bit x$ is q or delaying the write until

the value of the FE-bit is set to q; after writing, the value of the FE bit becomes r;

• readqr x$, means the read of x$ if the value of FE-bit x$ is q or delaying the read until the value of the FE-bit is set to q; after reading, the value of the FE bit becomes r.

Synchronization variables can be used as semaphores to create barriers and critical intervals that grand one thread from a set of threads an exclusive access to the variables shared by this set of threads. In addition, synchronization variables can be used to define dataflow computations. For instance, $F(i, j) = 0.25*(F(i-1, j) + F(i-1, j-1) + F(i, j-1) + F(i, j))$ can be evaluated for region $0 < i < n + 1$ and $0 < j < n + 1$, given values of F(0, 0), F(0, j), F(i, 0), and values of FE-bits set to full for F(0, 0), F(0, j), F(i, 0) as follows (symbol $ denotes the number of the current process [8]):

```
  spawn (1, n) {
  int i;
  i=$;
  spawn (1, n) {
  int j;
  j=$;
  purge (&(F[i, j]);//set FE-bits to
empty in region 0<i<n+1, 0<j<n+1
  }
  }
  spawn (1, n) {
  int i;
  i=$;
  spawn (1, n) {
  int j;
  j=$;
  double Left=readff(&(F[i, j-1]);
  double BottomLeft=readff(&(F[i-1, j-1]);
  double Bottom=readff(&(F[i-1, j]);
  double t= readee(&(F[i, j]);
  t=0.25*(t+Left+BottomLeft+Bottom);
  writeef(&(F[i, j]), t);
  }
  }
```

The program generates n threads, each of which spawns n threads. At the first step, the memory cells that store F(i, j) in region $0 < i < n + 1$, $0 < j < n + 1$ have the FE-bit set to empty. Then, as at the previous step, n x n threads are generated. In each thread, the corresponding function is evaluated. First, F [1, 1] is computed because F(0, 0), F(0, 1), and F(1, 0) initially have FE-bits set to full. Next, F [2, 1] and F [1, 2] can be computed in parallel because value F [1, 1], which is required for their computation and has the full FE-bit, becomes available. Then, F [3, 1], F [2, 2], and F [1, 3] are computed in parallel, and so on.

It seems that the PRAM-based parallel programming model considered above, which is implemented by extending C [8] with the functions for spawning and joining asynchronous lightweight threads, as well as synchronization based on atomic operations and memory access using FE-bits, satisfies the requirement that the user only needs to specify which computations can be carried out in parallel taking into account only a chosen algorithm.

Let us consider the implementation of the memory request flow mode supported by the considered programming model under the constraints imposed by modern electronic components.

According to the architectural solution shown in Fig. 10, an exascale computer system is built of computer modules (CMs), each having one or more processor chips with connected memory blocks and interfaces of a communication fabric that integrates all computer modules [7, 9, 12, 13].

The processor chip has an on-chip interconnect fabric for processing elements (PEs), which consist of computing cores with connected local memory blocks (including cache memory), special computing engines, and one or more memory management units (MMUs). The implementation of memory blocks, which create an effect of a large percentage of transistors that do not switch in each cycle in the local chip area occupied by a PE, creates some additional effects. Short instruction and data conductors among cores and local memory blocks ensure high power efficiency and clock speed. The chip can accommodate many (thousands) of PEs, the operation of which requires loading and unloading their memory blocks to change programs and data, as well as to initialize program execution in them. These architectural solutions are used in chips with various specializations, e.g., PEZY-SC2 [14], which is focused on numerical methods for solving differential equations, and Graphcore Colossus IPU [15], designed for solving artificial intelligence problems. The specialization of a chip architecture determines whether a chip serves only as a field of homogeneous PEs loaded and controlled from a host computer external to the chip (like Graphcore Colossus) or a hierarchical heterogeneous cluster architecture is implemented in which computing devices have local memory blocks at each hierarchical level (as in PEZY-SC2). In this architecture, the communication between the host computer and the special-purpose chip is partially shifted to control processors of clusters, which reduces the requirements for the performance of the host computer and for the bandwidth of the communication channel between the host computer and the special-purpose chip or a set of special-purpose chips.

The computer module also has a number of smart controllers for memory blocks, memory blocks themselves and one or more network interface controllers

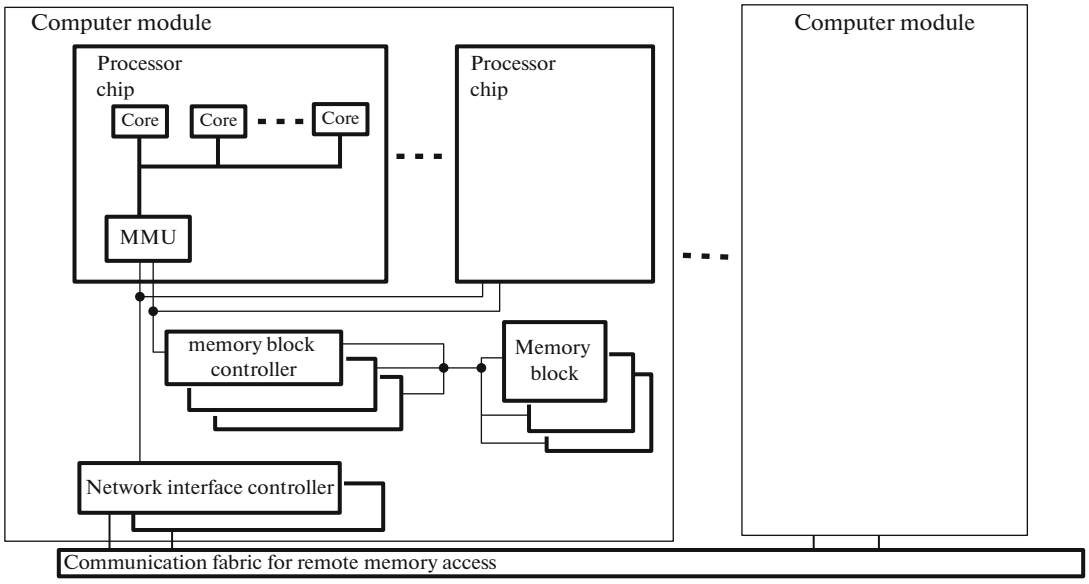

Fig. 10. Architecture of an exascale supercomputer.

necessary to provide the required throughput of the communication fabric of a supercomputer system.

Supercomputer distributed shared memory (DSM) consists of a set of memory blocks located in each CM. The number of memory blocks in each CM, on the one hand, should be as large as possible, which makes it potentially possible to serve more memory requests. On the other hand, it should be taken into account that their number is limited by the complexity of memory management and implementation.

When the supercomputer is initialized, the global address space is configured by distributing the address space over memory blocks. This distribution is fixed in MMUs.

When the current thread executes a read or write instruction to access shared memory upon generating a memory access request, each on-chip core suspends this thread until it receives a response from memory. The request from the thread is added to the queue of memory requests of the MMU selected by the address in the read or write instruction.

Based on the distribution of the global address space over memory blocks in the process of initialization, the MMU determines the destination of the request: local block or remote block of another CM. In the latter case, the request is sent to the network interface controller, which must forward it to the corresponding remote memory block of another CM while putting this request in its queue of requests. After receiving a response from the remote memory block, the network interface controller removes the corresponding request from the queue and passes the received response to the corresponding MMU.

After receiving the response from the remote or local memory block, the MMU removes the corresponding request from its queue of memory requests and sends the memory access result to the core that generated the request, thus making it possible to resume the execution of the suspended thread.

Thus, the execution of the memory access instruction in the core is delayed until the response on memory access completion from the MMU is received. Having received a request from the MMU directly or via the network interface controller, the smart controller of the memory block can operate in standard mode (read and write requests are executed in the order in which they are received and do not use additional features) or in extended mode, taking into account the mechanism of the FE-bit of a specified memory cell.

The smart controller of the memory block contains the FE bits for the memory words of the memory blocks controlled by this controller.

If the FE bit does not have the required value, then the memory request is placed in a wait table of the smart controller. The rows of this table contain memory access information (read/write instruction, memory cell address, FE-bit values before and after the access, service information for extended mode, and pointer to the corresponding MMU).

If the FE bit has the required value, then the memory block controller performs the required read or write and forms the specified FE-bit value. Without going into detail of memory controller implementation, it should be noted that, once the access to a memory cell is completed, the wait table must be searched through for the rows corresponding to this cell and a check must be performed on whether the corresponding memory request can be executed. A change in the state of the FE-bit initiates the processing of the

wait table with the application of the accepted approach to resolving EREW access conflicts.

If it can, then the request is passed for execution, and the corresponding row is removed from the table. Obviously, the actions described above require associative search through the waiting table.

The distribution of memory requests serves as a guarantee of high performance proportional to the number of smart controllers for memory blocks of computer modules.

The difference between the considered approach and traditional dataflow architectures [16] should be noted. The traditional architectures are based on associative memory, which is used to store instructions that wait for available operands. When all operands are ready, the instruction is extracted from associative memory and executed. Obviously, the number of instructions simultaneously extracted from associative memory determines the performance of a dataflow computer system. Therefore, the associative memory must be large. However, the creation of this memory is hindered by energy costs and drop in performance with increasing memory capacity.

Attempts made at software implementation of associative memory based on addressable memory using hash functions, partitioning shared associative memory into local blocks, and determining the readiness of only initial instructions from instruction sequences did not fundamentally change the situation [17, 18].

However, the approach at the level of memory block controllers that work with FE-bits seems quite feasible. First, there can be many memory blocks; second, only necessary synchronization is carried out without any preliminary partitioning of programs into instruction fragments.

The implementation of out-of-order execution of instructions in the processor through a reservation station and the use of memory block controllers that work with FE-bits cover the functionality of traditional dataflow architectures.

5. CONCLUSIONS

The research and development of parallel programming languages and tools requires analysis of existing architectural solutions aimed at improving performance, creation of experimental supercomputers, and programming of currently challenging tasks for them the execution of which on modern machines is unsatisfactory in terms of performance and scalability. The solution should be sought for from different perspectives while changing the representation of parallel programs and methods for mapping software components to resources, including the implementation of synchronous and asynchronous threads and software-hardware support of thread synchronization and interthread communication to determine satisfactory versions of architectures, OSs, runtime systems, program generation tools, and their compilations. In other words, it requires joint development of hardware and software.

The development of parallel programming is possible only in the framework of a particular model that takes into account the intrinsic features of an architecture that led to the emergence of this model. For instance, in the case of the model with message passing, this feature is the atomicity of message passes with the possibility of continuing execution only after receiving the entire message. In the case of the model with shared memory, the feature is in the implementation of access control for shared memory cells. That is why parallel programming for these models is based on different approaches.

The time has come to change the paradigm of supercomputing in general and the programming paradigm in particular. In this regard, there is a similarity with the situation in the early 1990s, which led to the replacement of vector pipeline computing based on the sequential programming paradigm extended with vector operations by the massively parallel paradigm based on message passing.

The new paradigm consists in representing all possible processing parallelism. The user only needs to specify which computations can be carried out by parallel threads over distributed shared memory taking into account only the chosen algorithm. When the same value needs to be read by many threads and then a new value needs to be written instead of it into the corresponding shared memory cell, the user assumes that the conflict resolution mechanism is implemented by hardware means of memory access control.

REFERENCES

1. Evreinov, E.V. and Kosarev, Yu.G., *Odnorodnye universal'nye vychislitel'nye sistemy vysokoi proizvoditel'nosti* (Homogeneous Universal High-Performance Computing Systems), Novosibirsk: Nauka, 1966.
2. Fortov, V.E., Levin, V.K., Savin, G.I., Zabrodin, A.V., Karatanov, V.V., Elizarov, G.S., Korneev, V.V., and Shabanov, B.M., The MVS-1000M supercomputer and its application prospects, *Nauka Prom. Ross.,* 2001, vol. 55, no. 11, p. 49.
3. Korneev, V.V., *Arkhitektura vychislitel'nykh sistem s programmiruemoi strukturoi* (Architecture of Computer Systems with Programmable Structure), Novosibirsk: Nauka, 1985.
4. Besta, M. and Hoefler, T., Slim Fly: A cost effective low-diameter network topology, *Proc. Int. Conf. High Performance Computing, Networking, Storage, and Analysis,* New Orleans, 2014, pp. 348–359.
5. Deng, Y., Guo, M., Ramos, A.F., Huang, X., Xu, Zh., and Liu, W., Optimal low-latency network topologies for cluster performance enhancement.
6. Wheeler, K., et al., Qthreads: An API for programming with millions of lightweight threads, *Proc. Workshop*

Multithreaded Architectures and Applications at IEEE IPDPS, 2008.

7. Korneev, V.V., An approach to programming supercomputers based on multicore multithreaded VLSI chips, *Vychisl. Metody Program.,* 2009, vol. 10, pp. 123–128.
8. Wen, X. and Vishkin, U., FPGA-based prototype of a PRAM-on-chip processor, *Proc. 5th Conf. Computing Frontiers Pages,* New York, 2008, pp. 55–66.
9. Elizarov, S.G., Luk'yanchenko, G.A., and Korneev, V.V., Technology of parallel programming of exaflops computers, *Program. Inzh.,* 2015, no. 7, pp. 3–10.
10. Institute of Electrical and Electronics Engineers, Portable Operating Systems Interface (POSIX.1), 1990. https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/fipspub151-2.pdf.
11. Feo, J., Dataflow on Cray XMT. https://www.youtube.com/watch?v=5nj1Ql0Eo5k&t=1266s.
12. Korneev, V.V., Programming model: Paradigm shift, *Otkrytye Sist.,* 2010, no. 3, pp. 29–31.
13. Korneev, V.V., Programming model and architecture of an exaflop supercomputer, *Otkrytye Sist.,* 2014, no. 10, pp. 20–22.
14. Torii, S. and Ishikawa, H., ZettaScaler: Liquid immersion cooling manycore based supercomputer, 2017.
15. Jia, Zh., Tillman, B., Maggioni, M., and Scarpazza, D., Dissecting the graphcore IPU architecture via microbenchmarking, 2019.
16. Gurd, J., Bohm, W., and Teo, Y., Performance issues in dataflow machines, *Future Gener. Comput. Syst.,* 1987, pp. 285–297.
17. Burtsev, V.S., The choice of a new system for organizing the execution of highly parallel computing processes, examples of possible architectural solutions for building supercomputers, *Parallelizm vychislitel'nykh protsessov i razvitie arkhitektury SuperEVM* (Parallelism of Computing Processes and Development of Supercomputer Architecture), 1997, pp. 41–78.
18. Klimov, A.V., Levchenko, N.N., and Okunev, A.S., Advantages of a data flow computing model in heterogeneous networks, *Inf. Tekhnol. Vychisl. Sist.,* 2012, no. 2, pp. 36–45.

Translated by Yu. Kornienko