DOI: 10.14529/jsfi250402

Supercomputing Co-Design for Solving Ill-Posed Linear Inverse Problems Using Iterative Algorithms

Alexander S. Antonov[1], ***Vladimir V. Voevodin***[1], ***Dmitry V. Lukyanenko***[2]

The paper considers an approach to applying the ideas of supercomputing co-design for the effective use of arbitrary multiprocessor computing systems with distributed memory when using iterative regularization algorithms to solve ill-posed linear inverse problems, which are reduced to solving large overdetermined systems of linear algebraic equations with a dense matrix. The proposed methodology allows for a large number of algorithms to select the best virtual topology of processes (in terms of parallelization efficiency) for solving problems of the class under consideration within the allocated resources of the supercomputer system being used.

Keywords: supercomputing co-design, parallelization efficiency, parallelism, autotuning, AlgoWiki, inverse problem, iterative regularization, conjugate gradient method.

Introduction

In many cases, applied inverse problems are linear and come down to the need to solve large overdetermined systems of linear algebraic equations of the form

$$A\,x = b_\delta \tag{1}$$

with a dense matrix. Here $A \in \mathbb{R}^{M\times N}$, $x \in \mathbb{R}^N$, $b_\delta \in \mathbb{R}^M$. Usually $M \geqslant N$ and instead of the exact right-hand side b its approximation b_δ is known, measured in an experiment with an error δ, i.e., $\|b-b_\delta\| \leqslant \delta$. The matrix can also be specified with an error, i.e., instead of the exact matrix A, its approximation A_h is known, where $\|A - A_h\| \leqslant h$ (hereafter, all vector norms are assumed to be Euclidean, and matrix norms – Frobenius, unless otherwise stated).

Often, such problems are ill-posed, and to solve them, it is necessary to use regularizing algorithms (see, for example, the classical work [10]). One of the most common classes of regularizing algorithms are iterative regularization algorithms. Most iterative regularization algorithms for solving problems of the form (1) contain only the following computational operations that allow for parallel implementation:

1. multiplication of a matrix of size $M \times N$ by a vector of size N (which requires performing $M \cdot N$ multiplications and $M \cdot (N-1)$ additions – in the sum of $M \cdot (2N-1)$ arithmetic operations);
2. multiplication of a transposed matrix of size $N \times M$ by a vector of size M (requires performing $N \cdot (2M-1)$ arithmetic operations);
3. scalar product of vectors of size N (N multiplications and $N-1$ additions – in the sum of $2N-1$ arithmetic operations) or vectors of size M (requires performing $2M-1$ arithmetic operations);
4. adding vectors of size N or M or multiplying them by a number (requires performing N and M arithmetic operations, respectively).

[1]Research Computing Center, Lomonosov Moscow State University, Moscow, Russia
[2]Department of Mathematics, Faculty of Physics, Lomonosov Moscow State University, Moscow, Russia

Two classical iterative regularization algorithms are given below as examples of such algorithms: the conjugate gradient method and the Nesterov accelerated method (see the description of the features of these algorithms in [3, 4] and [5, 7], respectively):

Algorithm 1: Conjugate gradient method.

Data: A, b_δ, δ
Result: x
$s \leftarrow 1$
$x \leftarrow 0$
$p \leftarrow 0$
while $\|Ax - b_\delta\|^2 > \delta^2$ **do**
  if $s = 1$ **then**
    $r \leftarrow A^T(Ax - b_\delta)$
  else
    $r \leftarrow r - \dfrac{q}{(p,q)}$
  end
  $p \leftarrow p + \dfrac{r}{(r,r)}$
  $q \leftarrow A^T(Ap)$
  $x \leftarrow x - \dfrac{p}{(p,q)}$
  $s \leftarrow s + 1$
end

Algorithm 2: Nesterov accelerated method.

Data: A, b_δ, δ, $\beta > -1$
Result: x
$s \leftarrow 1$
$x \leftarrow 0$
$p \leftarrow 0$
while $\|Ax - b_\delta\|^2 > \delta^2$ **do**
  if $s = 1$ **then**
    $x \leftarrow A^T b_\delta$
  else
    $x_{previous} \leftarrow x$
    $p \leftarrow x + \dfrac{s-2}{s-1+\beta}(x - x_{previous})$
    $x \leftarrow p - A^T(Ap - b_\delta)$
  end
  $s \leftarrow s + 1$
end

Note. Sometimes in the literature, these algorithms are presented in a form that assumes that the matrix A of the system (1) is symmetric and positive definite. In this case, the corresponding algorithm implementations do not include the operation of multiplying the transposed matrix by a vector. This type can also be used in the specified implementations of the algorithms if someone replaces $A := A^T A$ and $b_\delta := A^T b_\delta$. But this replacement is not constructive when using algorithms in practice to solve large problems. This is due to the following fact. The computational complexity of the written algorithms at each iteration is $O(MN)$ arithmetic operations. At the same time, the number of iterations s_{iter} that need to be performed is often significantly less than the number of unknowns N – when developing new algorithms for solving problems of the class under consideration, the main motivation is to increase the convergence rate, i.e., to reduce the number of iterations needed to achieve convergence. Thus, the computational complexity of the presented algorithms can often be estimated as $O(MN)$ provided $s_{iter} \ll N$. This is one of the main advantages of using iterative algorithms for solving problems like (1). These substitutions reduce the computational complexity of each iteration by about 2 times, but they require the preliminary execution of $O(MN^2)$ arithmetic operations, which negates the practical advantages of using iterative solution algorithms.

The approaches to the construction of parallel algorithms and their software implementation used in linear algebra operations in such iterative algorithms are well known. They are usually based on a two-dimensional partition of the matrix A into blocks. In this case, a Cartesian virtual topology based on a two-dimensional rectangular process grid is usually used, – in the address space of each computing process, its own block A_{part} of the original matrix A is stored (see Fig. 1). The size of such a process grid is usually chosen intuitively – most often, the number

of rows and columns of this process grid is made as uniform as possible. However, such a choice may not be optimal (in terms of parallelization efficiency) in many special cases, as it depends on various factors, the key ones of which are the following.

First, the optimal choice of the ratio of the sizes of the process grid depends on the problem being solved, namely, on the sizes M and N corresponding to the matrix. If, in the case of a square matrix, it is optimal to choose the same sizes of a two-dimensional process grid, then in the case of significantly different values of M and N, which is quite common in solving applied problems, such a choice will be far from optimal.

Secondly, the optimal size ratio of the process grid depends on the architecture of the multiprocessor computing system used. In particular, there may be a significant dependence on the resources allocated for running the application, which may change with each subsequent launch. At the same time, even if the application is running on the same number of computing nodes, each new launch may result in a set of computing nodes with a significantly different communication profile from the previous set (dedicated processes are located differently in the physical topology of a multiprocessor system).

Therefore, the purpose of this work is to create a methodology that will allow, within the framework of solving the problem of supercomputing co-design, to automatically match the size of the problem being solved and the topology of the computing resources allocated at application launch with the optimal size of a two-dimensional process grid defining the virtual topology of processes in a parallel software implementation of the algorithm.

Solving ill-posed linear inverse problems using iterative algorithms is extremely important for many applied problems in science and technology, while at the same time allowing us to demonstrate the basic ideas of supercomputing co-design [2]. Therefore, it has been chosen as a significant reference problem requiring high-performance computing facilities that test the approaches, methods, tools, techniques and recommendations implemented by the authors of the project to develop the fundamentals of supercomputing co-design based on the descriptions of algorithms in the AlgoWiki Open Encyclopedia of Parallel Algorithmic Features [11].

In connection with the above, the structure of this work will be as follows. Section 1 describes an approach to evaluating the parallelization efficiency of an algorithm for solving a problem of type (1) with fixed dimensions using the allocated resources of a supercomputer system. Section 2 presents a technique for matching the size of the Cartesian virtual topology of processes with the size of the problem being solved, the selected algorithm, and the allocated computing resources in the form of a practice-oriented algorithm. Section 3 demonstrates some of the results of numerical experiments. Section 4 discusses some of the features of the proposed methodology.

1. An Approach to Evaluating the Parallelization Efficiency of an Algorithm within the Framework of the Used Supercomputer System

First, subsection 1.1 will describe the proposed structure for storing problem data in the distributed address space of computing processes. Taking into account the specifics of the problem being solved, the data distribution will be used in accordance with the Cartesian virtual topology of two-dimensional grid type processes. Then, in subsections 1.2–1.5, estimates of the total running time of computational operations that allow for parallel implementation will be presented. Finally, subsection 1.6 will provide a formula for calculating the parallelization effi-

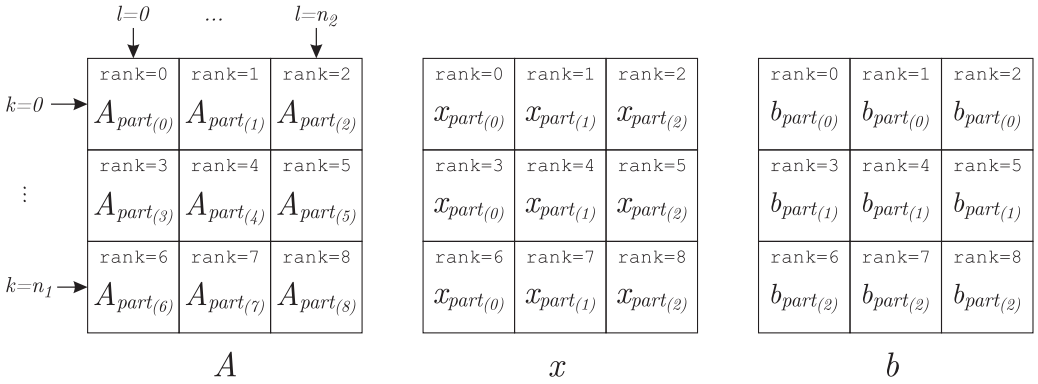

Figure 1. An example of the distribution of data across nine computational processes that form a two-dimensional grid 3×3

ciency of the proposed software implementation of the algorithm under consideration for solving the problem (1), depending on the size of the selected virtual process topology. MPI [1] technology is assumed to be a parallel programming technology for software implementations of the studied algorithms on a computer with distributed memory.

1.1. Distribution of Data by Processes Involved in Calculations

It is assumed that all task data is distributed over the address space of n computing processes involved in the calculations as follows.

Each computing process has its own identifier/number $\texttt{rank} \in \overline{0, \ldots, n-1}$. The matrix A of dimensions $M \times N$ is divided among these processes in two dimensions (see Fig. 1) for blocks $A_{part_{(\texttt{rank})}}$ of sizes $M_{part_{(k)}} \times N_{part_{(l)}}$.

It is assumed that the processes form a two-dimensional process grid of size $n_1 \times n_2$ (with $n_1 \cdot n_2 = n$). Therefore, the process number $\texttt{rank}$ is associated with the indexes k and l, which determine the coordinates of the process in the two-dimensional process grid as follows:

$$k = \left\lfloor \frac{\texttt{rank}}{n_2} \right\rfloor, \quad l = \texttt{rank} - \left\lfloor \frac{\texttt{rank}}{n_2} \right\rfloor \cdot n_2.$$

Or, if it is necessary to recalculate $\texttt{rank}$ by k and l:

$$\texttt{rank} = k \cdot n_2 + l.$$

We also note that

$$\sum_{k=0}^{n_1-1} M_{part_{(k)}} = M, \quad \sum_{l=0}^{n_2-1} N_{part_{(l)}} = N.$$

The vector x is divided into n_2 parts $x_{part_{(l)}}$, $l = \overline{0, n_2 - 1}$, of sizes $N_{part_{(l)}}$. In this case, the part $x_{part_{(l)}}$ for a fixed index l is stored on all processes in the column of the process grid with index l (see Fig. 1), that is, on processes with the coordinate $(\cdot, l)$ in a two-dimensional process grid.

The vector b is divided into n_1 parts $b_{part_{(k)}}$, $k = \overline{0, n_1 - 1}$, of sizes $M_{part_{(k)}}$. In this case, the part $b_{part_{(k)}}$ for a fixed index k is stored on all processes in the row of the process grid with index k (see Fig. 1), that is, on processes with the coordinate $(k, \cdot)$ in a two-dimensional process grid.

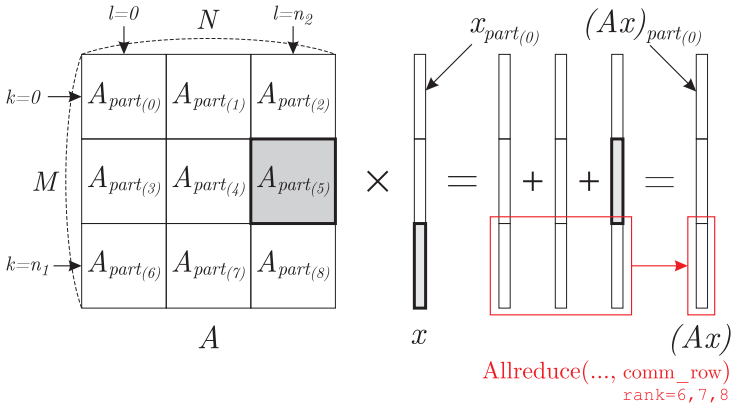

Figure 2. A parallel algorithm for matrix-vector multiplication in the case of two-dimensional matrix partition into blocks. The figure shows the case of data distribution over nine computing processes that form a two-dimensional grid 3×3

Thus, it is assumed that each computing process contains one of the blocks $A_{part_{(\cdot)}}$ of the matrix A, as well as one of the parts $x_{part_{(\cdot)}}$ and $b_{part_{(\cdot)}}$ of the vectors x and b, respectively.

1.2. Estimation of the Implementation Time of a Parallel Matrix-Vector Multiplication Algorithm in the Case of Two-Dimensional Matrix Partition Into Blocks

With the formulated method of storing data on computational processes, the result of multiplying the matrix A of size $M \times N$ by the vector x of size N will be a vector (Ax), which will be distributed over various computational processes in parts $(Ax)_{part_{(\cdot)}}$. At the same time, the distribution structure of this vector for various processes should correspond to the distribution structure of the vector b (see Fig. 1). Part of the vector (Ax) can be calculated using the following formula:

$$(Ax)_{part_{(k)}} = \sum_{l=0}^{n_2-1} A_{part_{(k \cdot n_2 + l)}} x_{part_{(l)}}, \quad k = \overline{0, \ldots, n_1 - 1}. \tag{2}$$

An example explaining this formula is provided in Fig. 2.

Each process involved in the calculations can compute the term $A_{part_{(k \cdot n_2 + l)}} x_{part_{(l)}}$ for its pair of values (k, l), independently of similar calculations performed by other processes. That is, the terms in the formula (2) can be calculated in parallel. Then the terms located on the processes of the row of the process grid with index k should be summed up, and the result – vector $(Ax)_{part_{(k)}}$ – should be placed on all processes of this row of the process grid. The organization of the interaction of processes entails overhead costs for receiving/transmitting messages containing the results of intermediate calculations through the physical communication environment of a multiprocessor computing system.

Let us estimate the total running time of this parallel algorithm, taking into account the overhead costs of organizing the interaction of computing processes.

First, we note that the running time of the sequential implementation of the operation of multiplying the matrix A of size $M \times N$ by the vector x of size N (totaling $M \cdot (2N - 1)$ arithmetic operations) can be evaluated using the formula

$$T_1^{Ax} = C_1 \cdot M(2N - 1). \tag{3}$$

Hereafter, the superscript T denotes the operation being evaluated (in this case, the operation of multiplying a matrix by a vector – Ax), and the subscript denotes the number of processes used for calculation (in this sequential case – 1); C_1 is the average speed of performing one arithmetic operation (determined by the architecture of the processor and the computing node on which the computing process is performed, and is considered known or can be calculated). At the same time, it is further assumed that the number of processor cycles required for adding numbers and multiplying them is equal.

If n computing nodes are used for parallel computing within a processor grid of size $n_1 \times n_2$, the time spent by each computing node on computation can be estimated as

$$C_1(n_1, n_2) \cdot \frac{M}{n_1}\Big(2\frac{N}{n_2} - 1\Big) \equiv C_1(n_1, n_2) \cdot \frac{M(2N - n_2)}{n}.$$

Note. Hereafter, it is assumed that the difference in the sizes of $M_{part_{(k)}}$ and $N_{part_{(l)}}$ data blocks on various processes can be ignored and considered equal to $M_{part_{(k)}} \equiv M/n_1$ and $N_{part_{(l)}} \equiv N/n_2$ respectively. This is due to the fact that usually the maximum difference in the size of data blocks distributed across different processes is no more than 1, which means that in the case of solving "large" problems, this difference can be ignored.

At the same time, it is specifically noted that the coefficient C_1 depends on the sizes n_1 and n_2 of the process grid, since these sizes determine the sizes of the blocks $(Ax)_{part_{(\cdot)}}$ of matrix A that are involved in calculations. Given that in popular programming languages used for parallel programming (for example, C/C++ and Python), matrices (i.e., two-dimensional arrays) are stored line-by-line in memory, the sizes of these matrices will determine the access time to their elements during the software implementation of matrix-vector multiplication.

The overhead time for organizing the interaction of each group of n_2 processes included in each of the rows of the process grid can be estimated as follows. First, the amount of data transmitted by each process is proportional to $\dfrac{M}{n_1}$ (the proportionality coefficient is determined by the type of data transmitted). Secondly, the number of data transfer operations with optimal organization of transfers is proportional to $\log_2 n_2$ (the proportionality coefficient is determined by the implementation of the corresponding function of interaction of computational processes within the framework of the parallel programming technology used — for example, the function of collective interaction of processes `Allreduce()` from MPI). Thus, the overhead time will be proportional to $\dfrac{M}{n_1} \cdot \log_2 n_2$, where the proportionality coefficient will take into account both the already mentioned proportionality coefficients and the technical features of the communication network, as well as the implemented option for distributing processes across computing nodes.

Thus, the total running time of the corresponding parallel algorithm can be estimated using the formula

$$T_n^{Ax} = C_1(n_1, n_2) \cdot \frac{M(2N - n_2)}{n} + C_2(n_1, n_2) \cdot \frac{M}{n_1} \cdot \log_2 n_2. \tag{4}$$

Here, the coefficient C_2 is considered known and 1) depends on the speed of transmission of a unit of information over a communication network, which, in turn, depends on the relative location of computing nodes, in particular, as determined by the sizes n_1 and n_2 of the process grid, as well as on the implemented option for distributing processes across computing nodes; 2) depends on the implementation of the function of collective interaction of processes.

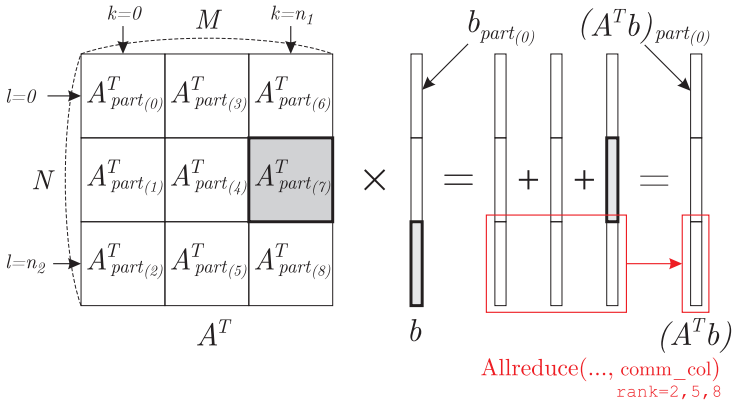

Figure 3. A parallel algorithm for multiplying a transposed matrix by a vector in the case of a two-dimensional matrix partition into blocks. The figure shows the case of data distribution across nine computing processes that form a two-dimensional 3×3 grid. Taking into account the transposition, it is assumed that the rows of the presented process grid are the columns of the original process grid; similarly, the columns of the presented process grid are the rows of the original process grid

The estimates (3) and (4) will be used in subsection 1.6 to construct a formula for evaluating the effectiveness of the parallelization of the software implementation of the algorithm for solving the problem (1).

1.3. Estimation of the Implementation Time of a Parallel Algorithm for Multiplying a Transposed Matrix by a Vector in the Case of a Two-Dimensional Matrix Partition Into Blocks

The result of multiplying the transposed matrix A^T of size $N \times M$ by the vector b of size M will be a vector $(A^T b)$, which will be distributed over various computational processes in parts $(A^T x)_{part_{(.)}}$. At the same time, the distribution structure of this vector for various processes should correspond to the distribution structure of the vector x (see Fig. 1). Part of the vector $(A^T b)$ can be calculated using the following formula:

$$(A^T b)_{part_{(l)}} = \sum_{k=0}^{n_1-1} A^T_{part_{(k \cdot n_2 + l)}} b_{part_{(k)}}, \quad l = \overline{0, \ldots, n_2 - 1}. \tag{5}$$

An example explaining this formula is provided in Fig. 3.

Each process involved in the calculations can compute the term $A^T_{part_{(k \cdot n_2 + l)}} b_{part_{(k)}}$ for its pair of values (k, l), independently of the similar calculations performed by other processes. That is, the terms in the formula (5) can be calculated in parallel. Then the corresponding terms located on the processes of the column of the process grid with index l should be summed up, and the result – vector $(A^T b)_{part_{(l)}}$ – should be placed on all processes of this column of the process grid. The organization of the interaction of processes entails overhead costs for receiving and transmitting messages containing the results of intermediate calculations through the physical communication environment of a multiprocessor computing system.

Similarly to the method described in the previous subsection, it is possible to obtain an estimate of the time required for the sequential implementation of this operation.

$$T_1^{A^T x} = \tilde{C}_1 \cdot N(2M - 1)$$

and parallel:

$$T_n^{A^T x} = \tilde{C}_1(n_1, n_2) \cdot \frac{N \cdot (2M - n_1)}{n} + C_2(n_1, n_2) \cdot \frac{N}{n_2} \cdot \log_2 n_1.$$

The fundamental difference between these formulas and the formulas presented in the previous subsection is that the coefficient C_1 has been replaced by $\tilde{C}_1$. This is due to the fact that the implementation of the multiplication of a transposed matrix by a vector involves sequential access to the elements of the rows of this matrix; however, the transposed matrix is stored in memory by columns. In order to save memory when solving "large" problems, the preliminary transposition of the matrix is not constructive, as it requires additional memory space that may not be available. A different order of access to the elements of the matrix leads to the fact that the coefficient $\tilde{C}_1$ may differ significantly from the coefficient C_1.

1.4. Estimation of the Implementation Time of the Parallel Scalar Product Algorithm of Vectors

Since the size of the vectors in sequential algorithms for solving the problem (1) is either M or N, we will use characteristic vectors of these sizes. Therefore, if the vector x is mentioned, its size will be assumed to be N, and if the vector b is mentioned, its size will be assumed to be M. It is assumed that these vectors are distributed over computational processes according to the scheme shown in Fig. 1.

The result of the scalar product of vectors $x^{(1)}$ and $x^{(2)}$ of size N will be the value $(x^{(1)}, x^{(2)})$. It can be calculated using the following formula:

$$(x^{(1)}, x^{(2)}) = \sum_{l=0}^{n_2-1} \left(x^{(1)}_{part_{(l)}}, x^{(2)}_{part_{(l)}}\right). \tag{6}$$

Each process involved in the calculations can compute the term $\left(x^{(1)}_{part_{(l)}}, x^{(2)}_{part_{(l)}}\right)$ for its value l, which is independent of similar calculations performed by other processes. That is, the terms in the formula (6) can be calculated in parallel. Then the corresponding terms located on the processes of the row of the process grid should be summed up, and the result – the number $(x^{(1)}, x^{(2)})$ – must be placed on all processes in this row of the process grid. The organization of the interaction of processes entails overhead costs for receiving/transmitting messages containing the results of intermediate calculations through the physical communication environment of a multiprocessor computing system.

The running time of a sequential scalar product operation of vectors can be estimated using the formula

$$T_1^{(x^{(1)}, x^{(2)})} = C_1 \cdot (2N - 1),$$

and the running time of the corresponding parallel algorithm (taking into account overhead costs) can be estimated using the formula

$$T_n^{(x^{(1)}, x^{(2)})} = C_1(n_1, n_2) \cdot \frac{2N - n_2}{n_2} + C_2(n_1, n_2) \cdot \log_2 n_2.$$

It is necessary to note the following two features of the algorithm. First, the processes of each row of the process grid perform the same calculations as the processes of the other rows of the process grid. Thus, scalar product calculations are not parallelized over all n computational

processes, but only over n_2 processes. This approach is constructive, since distributing the vectors $x^{(1)}$ and $x^{(2)}$ across all n processes (and not just the n_2 processes of the row of the process grid), and then collecting data from all of them, will require additional overhead proportional to $\log_2 n$, which almost always exceeds the gain in reducing the time of direct calculations when solving "large" problems ($C_1(2N-1)/n$ instead of $C_1(2N-1)/n_2$ in the considered version of the parallel algorithm). Secondly, it assumes that the parts of the vector are well localized in the address space of each process. In this regard, the same estimate can be used for the average execution speed of one arithmetic operation C_1 as in subsection 1.2.

Similarly, the result of the scalar product of vectors $b^{(1)}$ and $b^{(2)}$ of size M will be the value $(b^{(1)}, b^{(2)})$. It can be calculated using the following formula:

$$(b^{(1)}, b^{(2)}) = \sum_{k=0}^{n_1-1} \left(b^{(1)}_{part_{(k)}}, b^{(2)}_{part_{(k)}}\right). \tag{7}$$

Each process involved in the calculations can calculate the term $\left(b^{(1)}_{part_{(k)}}, b^{(2)}_{part_{(k)}}\right)$ for its value k is independent of similar calculations performed by other processes. That is, the terms in the formula (7) can be calculated in parallel. Then the corresponding terms located on the processes of the column of the process grid should be summed up, and the result – the number $(b^{(1)}, b^{(2)})$ – must be placed on all processes in this column of the process grid. The organization of the interaction of processes entails overhead costs for receiving/transmitting messages containing the results of intermediate calculations through the physical communication environment of a multiprocessor computing system.

The running time of a sequential scalar product operation of vectors can be estimated using the formula

$$T_1^{(b^{(1)}, b^{(2)})} = C_1 \cdot (2M-1),$$

and the running time of the implementation of the corresponding parallel algorithm (taking into account overhead costs) can be estimated using the formula

$$T_n^{(b^{(1)}, b^{(2)})} = C_1(n_1, n_2) \cdot \frac{2M-n_1}{n_1} + C_2(n_1, n_2) \cdot \log_2 n_1.$$

1.5. Estimation of the Implementation Time of a Parallel Algorithm for Adding/subtracting Vectors or Multiplying/dividing a Vector by a Number

When distributing vectors across computational processes according to the scheme shown in Fig. 1, in the case of adding two vectors $x^{(1)}$ and $x^{(2)}$ of size N, the following calculations will be performed on each process:

$$(x^{(1)} + x^{(2)})_{part_{(l)}} = x^{(1)}_{part_{(l)}} + x^{(2)}_{part_{(l)}}.$$

There will be no need to exchange messages between computational processes, since the storage structure of the resulting vector corresponds to that used in the parallel algorithms of linear algebra operations described earlier.

Therefore, the time of successive operations of adding vectors of the appropriate size can be estimated using the formulas

$$T_1^{x^{(1)}+x^{(2)}} = C_1 \cdot N,$$
$$T_1^{(b^{(1)}+b^{(2)})} = C_1 \cdot M,$$

and the operating time of the implementation of the corresponding parallel algorithms can be estimated using the formulas

$$T_n^{x^{(1)}+x^{(2)}} = C_1 \cdot \frac{N}{n_2},$$
$$T_n^{(b^{(1)}+b^{(2)})} = C_1 \cdot \frac{M}{n_1}.$$

Exactly the same estimates are true for operations of multiplying the corresponding vectors by a number. Therefore, the formulas for estimating the values of $T^{c \cdot x}$ and $T^{c \cdot b}$ are not written out separately and are considered equivalent to those described above.

At the same time, similarly to the previous subsection (subsection 1.4), it should be noted that the processes of each row (column) of the process grid perform the same calculations as the processes of the other rows (columns) of the process grid. Thus, the corresponding calculations are not parallelized over all n computing processes, but only over n_2 (n_1) processes. The motivation for this approach is similar to that given in subsection 1.4.

1.6. Evaluation of the Parallelization Efficiency of a Parallel Algorithm

The parallelization efficiency of the iterative algorithm for solving the problem (1) can be estimated by the parallelization efficiency of one iteration using the following formula:

$$E_n\big(n_1, n_2, \vec{k}, C_1, \tilde{C}_1, C_2, M, N\big) = \frac{\sum\limits_{op \in \left\{ Ax;\ A^T b;\ (x^{(1)}, x^{(2)});\ (b^{(1)}, b^{(2)});\ x^{(1)}+x^{(2)};\ b^{(1)}+b^{(2)} \right\}} k^{op} T_1^{op}}{\sum\limits_{op \in \left\{ Ax;\ A^T b;\ (x^{(1)}, x^{(2)});\ (b^{(1)}, b^{(2)});\ x^{(1)}+x^{(2)};\ b^{(1)}+b^{(2)} \right\}} k^{op} T_n^{op} n}. \quad (8)$$

Here:

- n_1, n_2 are determined by the selected Cartesian topology of parallel processes of the two-dimensional grid type;
- M, N are determined by the size of the problem being solved;
- $\vec{k} = \left\{ k^{Ax};\ k^{A^T b};\ k^{(x^{(1)}, x^{(2)})};\ k^{(b^{(1)}, b^{(2)})};\ k^{x^{(1)}+x^{(2)}};\ k^{b^{(1)}+b^{(2)}} \right\}$ is determined by the iterative algorithm for solving the problem (1) and contains the values of the number of corresponding operations (while calculating the values of $k^{x^{(1)}+x^{(2)}}$ and $k^{b^{(1)}+b^{(2)}}$, the number of multiplications/divisions of vectors of the appropriate size by a number is also taken into account);
- C_1, $\tilde{C}_1$ are determined by the architecture of the processor and computing node;
- C_2 is determined by the communication network, the implemented option for distributing processes across computing nodes, and the selected implementation of the operation for collective interaction of processes of the `Allreduce()` type (at the same time, as mentioned earlier, this parameter also depends on the size (M and N) of the problem and the selected size (n_1 and n_2) of virtual process topologies – they determine the volume and number of messages transmitted over the communication network);
- T_1^{op} and T_n^{op} are defined by formulas written out in subsections 1.2–1.5.

Example 1. For the algorithm 1, the expression (8) for the main iterations ($s \geqslant 2$) is written as

$$E_n = \frac{2 \cdot T_1^{Ax} + 1 \cdot T_1^{A^T b} + 2 \cdot T_1^{(x^{(1)}, x^{(2)})} + 1 \cdot T_1^{(b^{(1)}, b^{(2)})} + 6 \cdot T_1^{x^{(1)}+x^{(2)}} + 1 \cdot T_1^{b^{(1)}+b^{(2)}}}{\big(2 \cdot T_n^{Ax} + 1 \cdot T_n^{A^T b} + 2 \cdot T_n^{(x^{(1)}, x^{(2)})} + 1 \cdot T_n^{(b^{(1)}, b^{(2)})} + 6 \cdot T_n^{x^{(1)}+x^{(2)}} + 1 \cdot T_n^{b^{(1)}+b^{(2)}}\big)n},$$

Example 2. For the algorithm 2, the expression (8) for the main iterations ($s \geqslant 2$) is written as

$$E_n = \frac{2 \cdot T_1^{Ax} + 1 \cdot T_1^{A^T b} + 3 \cdot T_1^{x^{(1)}+x^{(2)}} + 2 \cdot T_1^{b^{(1)}+b^{(2)}}}{\big(2 \cdot T_n^{Ax} + 1 \cdot T_n^{A^T b} + 3 \cdot T_n^{x^{(1)}+x^{(2)}} + 2 \cdot T_n^{b^{(1)}+b^{(2)}}\big)n}.$$

2. Choosing the Optimal Size of the Virtual Process Topology

The problem of choosing the optimal sizes n_1 and n_2 of the virtual process topology can be set as follows: it is necessary to determine the values of n_1 and n_2 at which the parallelization efficiency of the selected iterative algorithm for solving the problem (1) on the allocated resources of the supercomputer system will be maximum. Such a statement of the problem can be formalized in the following form:

$$(n_1, n_2) = \underset{\substack{n_1, n_2 \\ n_1 \cdot n_2 = n}}{\operatorname{argmax}} E_n(n_1, n_2; \vec{k}, C_1, \tilde{C}_1, C_2, M, N). \tag{9}$$

Here, the first two arguments of the function E_n are specially separated from the remaining arguments by a semicolon to emphasize that these arguments are used for maximization, while the remaining arguments are known parameters.

Therefore, for the practical application of this formula, two questions remain to be solved:

1. How can someone take into account the condition $n_1 \cdot n_2 = n$? Obviously, on the one hand, the formal mathematical solution to the problem (9) in many cases will be non-integers, and on the other hand, the prime number n cannot be decomposed into factors, each of which is different from 1.
2. How does someone define the coefficients C_1, $\tilde{C}_1$, C_2? Taking into account the comments made earlier regarding these constants, it is obvious that the formal evaluation of them can be extremely difficult. For example, we repeat that the coefficient C_2 depends 1) on the technical features of the communication network, 2) on the sizes n_1 and n_2 of the selected virtual process topology, 3) on how well this virtual topology was mapped to the computing resources allocated at the start of the program, 4) on the implementation option of the function of collective interaction of processes in the package used for parallel programming.

The first issue can be solved as follows. It is necessary to impose a convenient limit on the number of computing processes n. Such a natural limitation for many multiprocessor systems is $n = 2^k$, where k is an integer. Thus, in the algorithm formulated below for determining the optimal values of n_1 and n_2, we will assume that n is a power of two, and the values of n_1 and n_2 must be found as integers.

The second issue can be solved as follows. On the allocated computing resources, before starting the main calculations, it is possible to run a preliminary test using matrices and vectors of characteristic sizes M, N, which will determine C_1, $\tilde{C}_1$ and C_2. That is, the corresponding coefficients will be determined automatically for the features of the allocated resources and the selected compiler options that determine the algorithms for implementing the function of collective interaction of processes. Moreover, the corresponding test runs should be made for

different values of n_1 and n_2 in order to determine the dependencies of $C_1(n_1, n_2)$, $\tilde{C}_1(n_1, n_2)$ and $C_2(n_1, n_2)$.

Thus, an algorithm is proposed for determining the optimal values n_1 and n_2 of the virtual process topology, designed as Algorithm 3.

3. Numerical Experiments

Computational experiments were conducted on the "Lomonosov-2" supercomputer [12] and were constructed as follows. The sizes M and N, which determine the computational complexity of the problem (1), were chosen in such a way that, on the one hand, the matrix A of the system (1) had significant size, and on the other hand, a part $A_{part_{(\cdot)}}$ of this matrix, which each individual computing process is responsible for storing, fit into the RAM of the computing node. For example, for pairs $(M, N) \in \{(10^5, 10^5), (10^6, 10^4), (10^7, 10^3)\}$, when using the data type `float64` ("double precision") the matrix A requires 298 GB for its storage in memory (while $A_{part_{(\cdot)}}$ requires 1.2 GB when using 64 computing nodes), and for pairs of $(M, N) \in \{(5 \cdot 10^5, 5 \cdot 10^5), (5 \cdot 10^6, 5 \cdot 10^4), (5 \cdot 10^7, 5 \cdot 10^3)\}$ – 1 863 GB (1.8 Tb) ($A_{part_{(\cdot)}}$ – 29 GB when using the same number of computing nodes).

At the same time, computational experiments were conducted for a series of these pairs of (M, N) in order to demonstrate that problems of the same computational complexity can correspond to completely different values of n_1 and n_2, which determine the optimal virtual topology of processes for parallel computing.

The MPI parallel programming technology was used for the software implementation of the Algorithm 3. The program code implementing the pseudocode from Algorithm 3 is not provided in this article, since the pseudocode is designed in such a way that the corresponding software implementation can be recovered from it in an unambiguous way (in the sense of choosing software solutions that may affect the parallelization efficiency). It should be noted that for the software implementation of the function of collective interaction of processes `Allreduce()`, its blocking version was used.

The computation results are shown in Fig. 4. It is perfectly clear that the values of n_1 and n_2, which determine the optimal virtual topology for parallel computing, depend both on the computational complexity of the problem and on the relative sizes of M and N. In particular, the intuitive facts are experimentally confirmed that, for example, for "elongated matrices" the optimal "elongated process topology" is obtained, and the closer the matrix is to the "square" one, the more optimal it is to use the "square process topology" for calculations.

However, there are some non-obvious results. For example, there may be situations where, if the virtual topology is chosen incorrectly, the parallelization efficiency may be close to zero. Although such a result is obtained for extreme cases (for example, $M = 5 \cdot 10^7$, $N = 5 \cdot 10^3$, i.e., the number of equations exceeds the number of unknowns by four orders of magnitude), which are extremely rare in solving applied problems, it is necessary to keep in mind the possibility of such effects in practice. It is also necessary to note the effect of a sharp increase in parallelization efficiency for a limiting process grid of size $n_1 \times n_2 \equiv 1 \times 64$ (noted in Fig. 4 by dotted line).

Remark 1. If someone uses a variant of the function `Allreduce()` implemented using persistent interaction requests from the MPI-4 standard, the results will obviously change. This is due to the following fact. Functionally (in the sense of the result), the operation of the function `Allreduce()` is equivalent to the sequence of launching functions `Allreduce_init()` "+" `start()` "+" `wait()` from the standard MPI-4. Given that the function `Allreduce()` can be im-

Algorithm 3: The pseudocode of the algorithm for determining the optimal size of the virtual topology of processes using the MPI parallel programming technology.

Data: N, M, $\vec{k}$, n – the power of two
Result: n_1, n_2
comm $\leftarrow$ MPI.COMM_WORLD
$n_1 \leftarrow n$; $n_2 \leftarrow 1$
while $n_1 >= 1$ *and* $n_1 <= n$ **do**
  comm_cart $\leftarrow$ comm.Create_cart(dims=(n_1, n_2), periods=True, reorder=True)
  rank_cart $\leftarrow$ comm_cart.Get_rank()
  comm_col $\leftarrow$ comm_cart.Split(rank_cart % n_2, rank_cart)
  comm_row $\leftarrow$ comm_cart.Split(rank_cart // n_2, rank_cart)
  // formation of arbitrary matrices of characteristic dimensions
  $A_{part} \leftarrow \text{random}(M/n_1, N/n_2)$; $x_{part} \leftarrow \text{random}(N/n_2)$; $b_{part} \leftarrow \text{random}(M/n_1)$
  // estimation of the value of the coefficient C_1
  time_start $\leftarrow$ MPI.Wtime()
  $(Ax)_{part} \leftarrow A_{part} \cdot x_{part}$
  time_elapsed $\leftarrow$ MPI.Wtime() $-$ time_start
  $C_1 \leftarrow \dfrac{\text{time\_elapsed} \cdot n}{M \cdot (2N - n_2)}$
  comm_cart.Allreduce($C_1/n \rightarrow C_1$, op=MPI.SUM) // averaging C_1
  // estimation of the value of the coefficient $\tilde{C}_1$
  time_start $\leftarrow$ MPI.Wtime()
  $(A^T b)_{part} \leftarrow A^T_{part} \cdot b_{part}$
  time_elapsed $\leftarrow$ MPI.Wtime() $-$ time_start
  $\tilde{C}_1 \leftarrow \dfrac{\text{time\_elapsed} \cdot n}{N \cdot (2M - n_1)}$
  comm_cart.Allreduce($\tilde{C}_1/n \rightarrow \tilde{C}_1$, op=MPI.SUM) // averaging $\tilde{C}_1$
  // estimation of the value of the coefficient C_2
  comm_cart.Barrier()
  time_start $\leftarrow$ MPI.Wtime()
  comm_col.Allreduce($(A^T b)_{part} \rightarrow x_{part}$, op=MPI.SUM)
  time_elapsed $\leftarrow$ MPI.Wtime() $-$ time_start
  if $n_1 = 1$ **then** $C_{2temp_1} \leftarrow 0$ **else** $C_{2temp_1} \leftarrow \dfrac{\text{time\_elapsed}}{N/n_2 \cdot \log_2 n_1}$
  comm_cart.Barrier()
  time_start $\leftarrow$ MPI.Wtime()
  comm_row.Allreduce($(Ax)_{part} \rightarrow b_{part}$, op=MPI.SUM)
  time_elapsed $\leftarrow$ MPI.Wtime() $-$ time_start
  if $n_2 = 1$ **then** $C_{2temp_2} \leftarrow 0$ **else** $C_{2temp_2} \leftarrow \dfrac{\text{time\_elapsed}}{M/n_1 \cdot \log_2 n_2}$
  if $n_1 = 1$ **then** $C_{2temp_2} \leftarrow 0$; **if** $n_2 = 1$ **then** $C_{2temp_1} \leftarrow 0$
  if $n_1 \neq 1$ and $n_2 \neq 1$ **then** $C_2 \leftarrow (C_{2temp_1} + C_{2temp_2})/2$
  comm_cart.Allreduce($C_2/n \rightarrow C_2$, op=MPI.SUM)
  // parallelization efficiency estimation
  $E(n_1, n_2) \leftarrow E_n(n_1, n_2; \vec{k}, C_1, \tilde{C}_1, C_2, M, N)$
  $n_1 \leftarrow n_1/2$; $n_2 \leftarrow n_2 \cdot 2$
end
$(n_1, n_2) \leftarrow \text{argmax}\, E(n_1, n_2)$

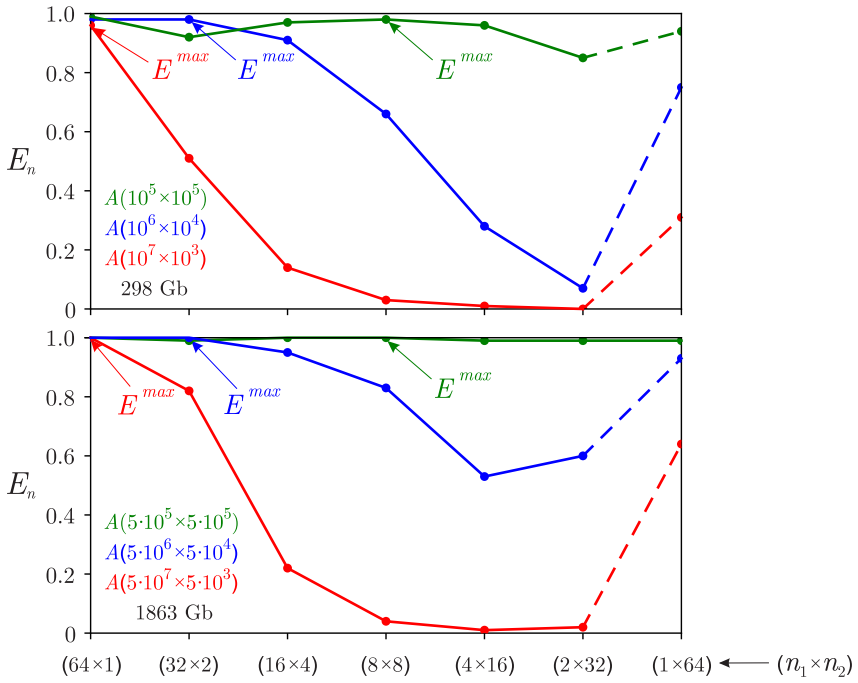

Figure 4. Graphs of the dependence of the parallelization efficiency of Algorithm 1 for matrices of different dimensions, but with the same number of elements, depending on the selected virtual topology when computing on a fixed number of nodes of the "Lomonosov-2" supercomputer [12]

plemented over the same areas of the address space, the same `Allreduce_init()` operation can be performed outside the `while` loop. This can significantly reduce the overhead of receiving and transmitting messages containing the results of intermediate calculations (for more information, see [6]). As a result, the estimate of C_2 may decrease significantly.

Remark 2. It is obvious that the method proposed in the paper for calculating the parallelization efficiency in determining the optimal virtual topology is an estimated one. Therefore, the question arises: how much does the actual parallelization efficiency differ from the estimated one in typical tasks? To answer this question, the following experiment was conducted. An example was chosen for $(M, N) = (8 \cdot 10^4, 8 \cdot 10^4)$. In this case, the matrix A will require 47.8 GB for its storage, as a result of which it will completely fit into the RAM of one computing node (64 GB) of the "Lomonosov-2" supercomputer, which is required to determine the operating time T_1 of the sequential version of the Algorithm 1. For this matrix A and some model right-hand side b_δ of the system (1), a parallel implementation of the Algorithm 1 was launched in two versions – using the functions of collective communication of MPI-processes within the MPI-3 standard and with using the functions of collective communication of MPI-processes within the MPI-4 standard (see previous remark). At the same time, the number of iterations in the Algorithm 1 was forcibly limited to 300 iterations in order to obtain a reasonable counting time for the sequential implementation of the Algorithm $T_1 = 864$ seconds, which is a fairly representative counting time, but at the same time slightly less than the upper limit of 15 minutes, which limits the counting time on the test queue of the "Lomonosov-2" supercomputer. For this example, as for similar examples with a square matrix of higher dimensions, the optimal topology is a "square" one. Therefore, parallel implementations of the algorithm were launched on $n \in \{4, 9, 16, 25, 36, 49, 64\}$ MPI-processes so that $n_1 \equiv n_2 \equiv \sqrt{n}$. The running time T_n for each run was detected, then the speedup of calculations was calculated using the formula $S_n = T_1/T_n$

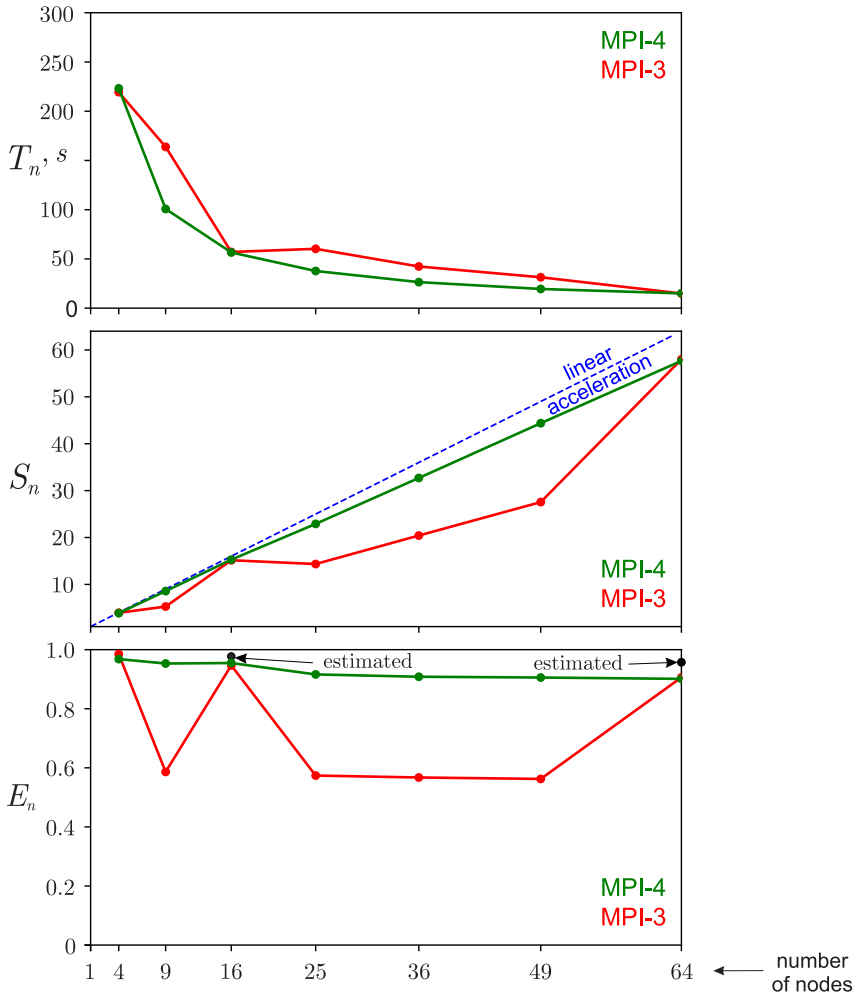

Figure 5. Graphs of the dependence of the counting time, speedup, and parallelization efficiency of the Algorithm 1 depending on the selected number of computing nodes within the framework of the "quadratic process topology". The graphs are typical for a typical launch, rather than the average values for a series of experiments. On the efficiency graph, the points "estimated" are marked separately, the values of which are calculated using the Algorithm 3

and the parallelization efficiency using the formula $E_n = S_n/n$. Based on these data, work schedules, speedup, and efficiency were constructed, as shown in Fig. 5. The key graph here is the graph of the real parallelization efficiency, which demonstrates certain differences between the real parallelization efficiency and the estimated one. However, these differences look insignificant from a practical point of view, which confirms the applicability of the proposed Algorithm 3.

4. Discussion

1. **Large-scale autotuning.** The ideas for adjusting the algorithm parameters to the characteristics of the target computing system, used in the development of the proposed methodology, are consonant with the ideas of autotuning (see, for example, the works [8, 9, 13–15]). However, in these works, the ideas considered mainly relate to "small-scale autotuning" – features of adjusting algorithms to the technical capabilities of "small" computing systems (a multicore processor or a graphics processing unit). The ideas proposed in our work relate

to adjusting the algorithm parameters to the characteristics of a distributed memory supercomputer and have not been previously considered in detail in publicly available sources.

2. **The invariance of the proposed algorithm with respect to the properties of the computing system used.** Such important characteristics of a communication network as latency, data transfer rate, topology of the communication network, and distribution of processes across computing nodes are important in the *a priori* estimation of the coefficient C_2. Also, many technical features of the computing node must be taken into account when *a priori* estimating the coefficients C_1 and $\tilde{C}_1$. In fact, the paper proposes *a posteriori* method for determining these constants based on the results of some preliminary tests, which automatically sets in them the corresponding characteristics of the used communication network and computing nodes. Thus, the proposed algorithm for determining the optimal values of the virtual topology of processes is relevant for any hardware, any parallel programming technology, any compilation options, and any features of the system and application software.
3. **Possible improvements to the proposed algorithm.** In the derivation of all formulas, the assumption was made that the computational complexity of all arithmetic operations is equivalent. If desired, the reader can clarify the output of the corresponding formulas, taking into account that, for example, addition of two numbers requires 1–3 CPU cycles, multiplication – 1–7 cycles, and division – 10–40 cycles. The values of the coefficients C_1 and $\tilde{C}_1$ on a processor-homogeneous computing system should not change much from launch to launch; therefore, they could potentially be calculated once before the start of all experiments, saved and used in the future. However, the distribution of processes across the computing nodes of a supercomputer can vary from launch to launch, even on the same set of computing nodes. Therefore, the coefficient C_2 must be calculated before each run of the computational algorithm implementation. The values of the coefficients C_1 and $\tilde{C}_1$ are averaged over all application processes. Considering that it is supposed to use a computer system that is homogeneous in terms of processors, this should not be a serious limitation and simplification. The coefficient C_2 is calculated as the average value for `Allreduce()` operations performed across rows and columns of the process grid. For greater accuracy of the algorithm, one can do without averaging and use two separate coefficients. Next, the value of the coefficient C_2 is averaged over all application processes, and the impact of this averaging may be significant. Further experiments should demonstrate how significant this factor is in practice and needs to be taken into account in order to obtain more accurate results.
4. **Theoretical assumptions.** Using only powers of two for the values of n_1 and n_2 does not represent a serious limitation of generality. This assumption is often used in practice when evaluating various dynamic characteristics of software implementations.
5. **Applicability of the algorithm.** The algorithm described in this article is designed to solve linear algebra problems using a two-dimensional Cartesian topology of processes, but it can be easily adapted to solve other problems using other topologies.
6. **Features of software implementation of algorithms.** Some statements made in subsections 1.1–1.5 when estimating the implementation time of parallel algorithms using basic linear algebra operations may be incomprehensible to readers who have not previously implemented parallel versions of iterative algorithms for solving systems of linear algebraic equations. For example, the statement may not be obvious (see subsection 1.5), that with

the chosen data storage structure, when adding vectors, there will be no need to exchange messages between different computing processes. Therefore, we provide a link to the work [6], which contains an example of a software implementation of an iterative algorithm from the class of algorithms under consideration (including using the functions of collective interaction of processes from various MPI standards: MPI-3 and MPI-4).

7. **The case of a heterogeneous computing system.** The approach discussed in this article is applicable without additional modifications for computing systems that are homogeneous in terms of core computing – whether they are central processing units (CPUs) or graphics accelerators (GPUs), while the system may be heterogeneous over a communication network. If a supercomputer co-design is required for a more complex heterogeneous system, then a more complex approach beyond the scope of this article may be required to distribute basic computing operations across heterogeneous computers.

Conclusion

The paper demonstrates the fundamental possibility of using the ideas of supercomputing co-design to automatically match the optimal topology of calculations with the features of the problem being solved, the features of the supercomputer system used and parallel programming technology. The proposed methodology for algorithms that are in demand in solving applied problems can increase the efficiency of using supercomputer systems by users.

Acknowledgements

The paper was published with the financial support of the Russian Science Foundation (project 25-11-00181, https://rscf.ru/en/project/25-11-00181/). The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University [12].

References

1. Antonov, A.S.: MPI and OpenMP Parallel Programming Technologies: Textbook. Moscow University Press (2012)

2. Antonov, A.S., Maier, R.V., Nikitenko, D.A., Voevodin, V.V.: An approach to solving the problem of supercomputer co-design. Lobachevskii J Math. 45(7), 2965–2973 (2024). https://doi.org/10.1134/S1995080224603680

3. Kalitkin, N.N., Kuzmina, L.V.: Improved form of the conjugate gradient method. Mathematical Models and Computer Simulations 4(1), 68–81 (2012). https://doi.org/10.1134/S2070048212010061

4. Kalitkin, N.N., Kuzmina, L.V.: Improved forms of iterative methods for systems of linear algebraic equations. Doklady Mathematics 88(1), 489–494 (2013). https://doi.org/10.1134/S1064562413040133

5. Kindermann, S.: Optimal-order convergence of Nesterov acceleration for linear ill-posed problems*. Inverse Problems 37(6), 065002 (2021). https://doi.org/10.1088/1361-6420/abf5bc

6. Lukyanenko, D.: Parallel algorithm for solving overdetermined systems of linear equations, taking into account round-off errors. Algorithms 16(5), 242 (2023). https://doi.org/10.3390/a16050242

7. Neubauer, A.: On Nesterov acceleration for Landweber iteration of linear ill-posed problems. Journal of Inverse and Ill-posed Problems 25(3), 381–390 (2017). https://doi.org/10.1515/jiip-2016-0060

8. Park, J., Shin, Y., Lee, J., *et al.*: HYPERF: End-to-End Autotuning Framework for High-Performance Computing. Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing (20), 1–14 (2025). https://doi.org/10.1145/3731545.3731588

9. Petrovič, F., Střelák, D., Hozzová, J., *et al.*: A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit. Future Generation Computer Systems 108, 161–177 (2020). https://doi.org/10.1016/j.future.2020.02.069

10. Tikhonov, A.N., Goncharsky, A.V., Stepanov, V.V., Yagola, A.G.: Numerical methods for the solution of ill-posed problems. Dordrecht: Kluwer Academic Publishers (1995)

11. Voevodin, V., Antonov, A., Dongarra, J.: AlgoWiki: an Open Encyclopedia of Parallel Algorithmic Features. Supercomputing Frontiers and Innovations 2(1), 4–18 (2015). https://doi.org/10.14529/jsfi150101

12. Voevodin, V., Antonov, A., Nikitenko, D., *et al.*: Supercomputer Lomonosov-2: Large scale, deep monitoring and fine analytics for the user community. Supercomputing Frontiers and Innovations 6(2), 4–11 (2019). https://doi.org/10.14529/jsfi190201

13. Vuduc, R., Demmel, J.W., Yelick, K.A.: OSKI: A library of automatically tuned sparse matrix kernels. Journal of Physics: Conference Series 16(1), 521 (2015). https://doi.org/10.1088/1742-6596/16/1/071

14. van Werkhoven B.: Kernel Tuner: A search-optimizing GPU code auto-tuner. Future Generation Computer Systems 90, 347–358 (2019). https://doi.org/10.1016/j.future.2018.08.004

15. Wu, X., Balaprakash, P., Kruse, M., *et al.*: ytopt: Autotuning scientific applications for energy efficiency at large scales. Concurrency and Computation: Practice and Experience 37(1), e8322 (2023). https://doi.org/10.1002/cpe.8322