Optimization of Regression Analysis by Conducting Parallel Calculations

Lesia Mochurad

Artificial intelligence Department, Lviv Polytechnic National University, Lviv, 79013, Ukraine

Abstract

The regression analysis of large data sets by parallelization is investigated in the work. Different approaches to parallel computing are proposed. The parallelization of the gradient descent method and the stochastic gradient descent using OpenMP and MPI technologies, as well as their hybrid, is performed. The efficiency of the proposed parallel algorithms is analyzed. A number of numerical experiments were performed. Acceleration of 5 times was achieved using the six-core architecture of a personal computer. By varying the number of threads and processor cores, the possibility of further optimization of the computational process is established. The problem of erroneous exchange is considered, which often has a negative effect on acceleration during parallelization using OpenMP. The advantages and disadvantages of using each of the technologies are analyzed. Without reducing the generality, the software product is developed in the work, which can be used in the regression analysis of big data processing in a wide range of economic problems.

Keywords 1

Gradient descent algorithm, multicore, OpenMP technology, MPI technology, acceleration

1. Introduction

When processing large amounts of information, a large number of calculations are implemented, which consumes significant resources and time for its processing. Moreover, the construction of systems that perform the necessary operations depends on the type of data received and describes each specific situation. Due to the variety of data that can simultaneously characterize the task, there is a problem of creating techniques that will universalize the processing of large amounts of information, regardless of its type [1].

As is known [2], in static modeling, regression analysis is a set of statistical processes for estimating the relationships between a dependent variable and one or more independent ones. The most common form of regression analysis is linear regression, when the researcher finds the line (or more complex linear combination) that is most appropriate for the data according to some particular mathematical criterion. Regression analysis is mainly used for two conceptually different purposes. First, such analysis is widely used for forecasting and prediction, where its use intersects with the field of machine learning [3]. Second, in some situations, regression analysis can be used to infer causal relationships between independent and dependent variables. Regression methods continue to be an area of active research. In recent decades, new methods of regression analysis have been developed, such as time series [4], increasing curves, nonparametric regression, and so on.

Modern economics uses many methods to assess the impact of factors on the level of development, but a special place is occupied by correlation regression analysis. This type of analysis allows not only to assess the reality of the influence of factors, but also to determine the intensity (momentum) of their impact on the effective indicator of regional development [5, 6].

COLINS-2021: 5th International Conference on Computational Linguistics and Intelligent Systems, April 22–23, 2021, Kharkiv, Ukraine
EMAIL: lesia.i.mochurad@lpnu.ua (L. Mochurad).
ORCID: 0000-0002-4957-1512 (L. Mochurad).

© 2021 Copyright for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-2870/paper74.pdf

In the work [7] the authors expanded the stochastic gradient descent for support vector machines in several ways to develop the new multiclass SVM-SGD for efficiently classifying large image datasets into many classes. ages with very-high-dimensional signatures into thousands of classes.

In the work [8] the authors created optimized implementations of gradient descent on both GPU and multi-core CPU platforms, and perform a detailed analysis of both systems' performance characteristics. The GPU implementation was done using CUDA, whereas the multi-core CPU implementation was done with OpenMP.

In the work [9] the authors presented the results of an empirical study of stochastic projection and stochastic gradient descent methods as means of obtaining approximate inverses and preconditioners for iterative methods. But the results are preliminary due to the code being not yet fully optimized.

Big data is a field that considers methods of analysis, systematic extraction of information, work with data sets that are too large or complex to be analyzed by traditional data processing software [10, 11]. There are problems when regression analysis needs to be performed on big data.

"Big Data" as a concept is becoming increasingly popular in the world. It is used in marketing research, in the analysis of world markets, sociological research and even elections [12]. Thus, large amounts of data can contain important information that requires efficient processing. It is usually performed using machine learning algorithms that study the available data. However, as a rule, such calculations require duration and complexity at the stages of learning or forecasting, which makes it impossible to use implementations of sequential algorithms, because they are unable to cope with the current data. Therefore, the aim of this work is to study the performance of regression analysis on large data sets by parallelization. Parallel approaches to increase processing speed are proposed.

The main contribution of this article can be summarized as follows:

- A simple yet accurate and computationally efficient method is developed, which combines the gradient descent method and the stochastic gradient descent using OpenMP and MPI technologies and their hybrid. It is especially relevant in modern trends in the development of the multi-core architecture of modern computers.
- The main advantage of the proposed technique is that I have received an acceleration that goes to the number of cores of the appropriate computing system; I have investigated the problem of false exchange, which often negatively affects acceleration with parallel with OpenMP.

2. Theoretical Basis

In general, the linear regression model is defined as:

$$y = \theta_0 + \theta_1 x_1 + ... + \theta_n x_n, \tag{1}$$

where y – dependent variable, x – property vector (independent variables), n – number of independent variables, θ – vector of linear regression coefficients.

Hypothesis is a function that best describes the purpose of machine learning (see formula (2)).

$$h(x) = \theta_0 + \theta_1 x_1 + ... + \theta_n x_n, \tag{2}$$

where $h(x)$ – hypothesis.

I will use the least squares method to estimate the unknown parameters. It takes as an estimate of the parameter values that minimize the sum of the squares of the residues for all observations. Thus, the cost function takes the form:

$$J(\theta_0, \theta_1, ... \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta (x^{(i)} - y^{(i)})^2), \tag{3}$$

where m – number of sample items, n – number of properties. The next step is to minimize the cost function (3). There are two known methods: the normal equation [13] or the gradient descent [14].

Normal equation:

$$\theta = (X^T X)^{-1} X^T y, \tag{4}$$

where X – matrix of values of independent variables, y – vector of values of the dependent variable, θ – vector of regression coefficients.

As is known [15], this method has advantages over the method of gradient descent, because it does not require a learning step and does not require iterations. As can be seen from formula (4) in the method is the multiplication of matrices of independent variables, which can be parallelized. However, this method has disadvantages in finding linear regression coefficients when working with large amounts of data. First, with a large size of the dataset, there may be a problem with its storage in memory, and secondly, for a large number of elements, a gradient descent is the best approach due to the high computational cost of a normal equation.

The research of the gradient descent method and the possibility of its parallelization is carried out in the work. This method requires a large number of iterations, but works well even with a significant amount of properties. The difficulty in parallelizing this method is that the algorithm is quite consistent. Each subsequent update of the coefficients depends on their previous values. Gradient descent algorithm:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1), \;\; j = 0 \vee 1, \tag{5}$$

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}, \tag{6}$$

θ_j – j– regression coefficient, α– learning step, m– the number of elements in the data set, h_θ – hypothesis, $x^{(i)}$ – i– string in the data set, $x_j^{(i)}$ – j– element in the i string of the data set, $y^{(i)}$ – i– dependent variable. It is necessary to iterate to a certain stop condition while updating the coefficients θ_j.

The gradient descent has a known optimization – stochastic gradient descent (7) [16]. This optimization method is well suited for solving the problem, because it coincides faster on a large amount of data, but the method is not always accurate enough [17].

$$\theta_j := \theta_j - \alpha * (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}, \; i = \overline{1,m}, \; j = \overline{1,n}. \tag{7}$$

Similarly, as in formula (6) it is necessary to iterate to a certain stop condition while updating the coefficients θ_j.

Therefore, in order to solve this problem, the gradient descent method and the stochastic gradient descent method were parallelized using OpenMP [18, 19] and MPI [20] technologies.

Regarding the choice of OpenMP technology, the advantages are:

- Due to the idea of "incremental parallelization", OpenMP is ideal for developers who want to quickly parallelize their computing programs with large parallel loops. The developer does not create a new parallel program, but simply consistently adds OpenMP directives to the text of the program.
- At the same time, OpenMP is a fairly flexible mechanism that gives the developer great control over the behavior of the parallel program.
- It is assumed that the OpenMP program on a single-processor platform can be used as a serial program, ie there is no need to support serial and parallel versions.

As for MPI technology, in the MPI programming model, the program generates several processes that interact with each other by calling routines for receiving and transmitting messages. Usually, when initializing an MPI program, a fixed set of processes is created, and (which, however, is optional) each of them runs on its own processor. These programs can run different programs, so the MPI model is sometimes called the MIMD model (Multiple Instruction, Multiple Data), in contrast to the SIMD model, where each processor performs only the same tasks. MPI supports two-point and global, synchronous and asynchronous, blocking and non-blocking types of communications. A special mechanism – a communicator – hides from the programmer internal communication structures. The structure of communications can change over the life of the process, but the number of tasks must remain constant (MPI-2 supports a dynamic change in the number of tasks). The MPI

specification provides program portability at the source code level. Work on heterogeneous clusters and symmetric multiprocessor systems is supported. Process startup during MPI program execution is not supported. The specification does not describe parallel I/O and program debugging – these features can be included in a specific MPI implementation in the form of additional packages or utilities. Compatibility of different implementations is not guaranteed.

3. Formulation of the problem

Without reducing the generality, the paper considers the problem of simulating the temperature of the stator and rotor in real time. Due to the complex structure of the electric traction drive, direct measurements with thermal sensors are not possible for the rotor temperature. In addition, accurate thermal modeling is becoming increasingly important with increasing relevance of functional safety. Therefore, it is necessary to create a model that estimates the stator temperature based on the input data.

The main task is to find the regression equation of the form (1). There is one dependent (stator temperature) and several independent variables. The current sample size, which is taken to solve the suppressed problem of finding the regression equation of 500 thousand elements, and the number of properties 7. However, the sample size can be much larger and vary the number of properties from 1000 and more, then the usual sequential algorithm can be long or calculations will be too large to fit in RAM. This problem can be solved by parallelizing the regression equation search algorithm.

Selected sample (dataset contains 500,000 data, 7 properties and the dependent variable [21]):

1. The ambient temperature measured by a thermal sensor located close to the stator;
2. Coolant temperature (water cooling motor);
3. d–component;
4. q–component;
5. Engine speed;
6. The d–current component;
7. The q–current component.

Dependent variable is the stator temperature measured by a thermal sensor.

4. Research results

The paper parallels the gradient descent algorithm and stochastic gradient descent using OpenMP technology.

Stages of parallelization of the gradient descent algorithm (5):

1. In the first stage, only the code fragment of the whole algorithm, which is responsible for multiplying the matrix X by the vector θ in the function of hypothesis h (2), is parallelized using the OpenMP directive.

Code snippet:

```
double* hypothesisP(double **X, double *Q, int m, int n) {
    double* result = new double[m];
    #pragma omp parallel for num_threads(2)
    for (int i = 0; i < m; i++) {
           result[i] = 0;
           for (int j = 0; j < n; j++) {
                  result[i] += X[i][j] * Q[j];
           }
    }
    return result; }
```

2. At this stage, the parallelization of the calculations required to find the result of the cost function (3). That is, the parallelization of the function of hypothesis (2), which is described in the first stage, and the parallelization of the error summation operation for each element (6).

```
double costP(double** X, double* Y, double* Q, int m, int n, int k) {
    double* hQ = new double[m];
    hQ = hypothesisP(X, Q, m, n);
```

```
    double* temp = new double[m];
    double sum = 0;
    #pragma omp parallel for num_threads(2)
    for (int i = 0; i < m; i++) {
            sum += (hQ[i] - Y[i]) * X[i][k];
    }
    return sum / m;
}
double* hypothesisP(double **X, double *Q, int m, int n) {
    double* result = new double[m];
    #pragma omp parallel for num_threads(2)
    for (int i = 0; i < m; i++) {
            result[i] = 0;
            for (int j = 0; j < n; j++) {
                    result[i] += X[i][j] * Q[j];
}}
    return result;}
```

The time spent on the second stage of the parallel gradient descent algorithm with different number of iterations and flows on a 2-core processor is given in Table 1 and Table 2, and the visualization of the results is presented in Figure 1 and Figure 2.

Table 1

The results of parallelization of the second stage using a 2-core processor

Number of iterations	Sequential algorithm execution time, sec	Execution time of parallel algorithm, sec	
		Number of threads	
		(2, 2)	(4, 4)
10	3.55	2.14	2.15
20	14.26	7.4	7.1
30	24.5	7.9	7.5

Table 2

Efficiency and acceleration (second stage, 2-core processor)

Number of iterations	**2 threads**		**4 threads**	
	Acceleration	**Efficiency**	**Acceleration**	**Efficiency**
10	1.65	0.825	1.65	0.41
20	1.92	0.96	2.0	0.5
30	3.10	1.55	3.26	0.81

Consequently, when parallelization the second stage of the proposed parallel algorithm we obtained the acceleration equal 3 and the efficiency – 0.8, when testing on the 2-core processor.

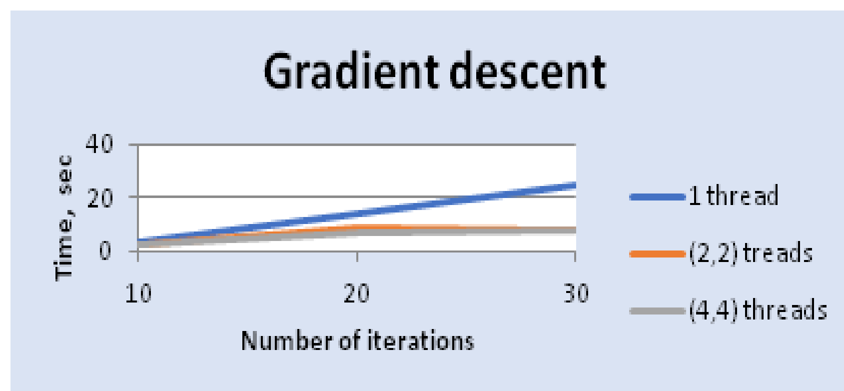

Figure 1: Visualization of execution time at parallelization of the second stage (2-core processor)

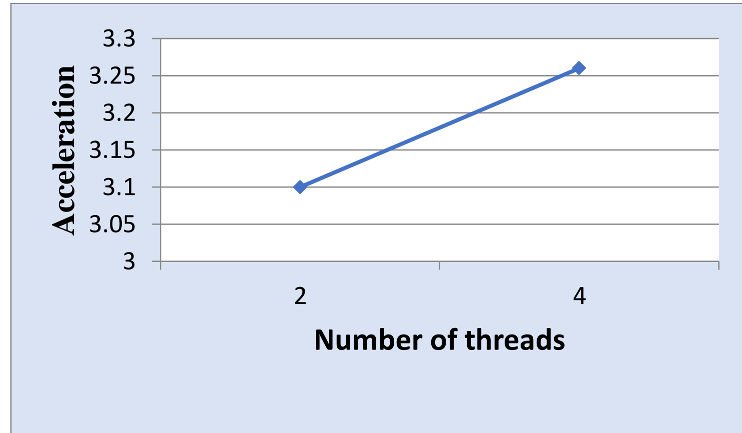

Figure 2: Acceleration obtained by parallelization of the second stage (2-core processor)

Similar numerical experiments were performed when tested on a 6-core processor. The results are given in Table 3 and Table 4 and in Figure 3 and Figure 4.

Table 3
The results of parallelization of the second stage using a 6-core processor

Number of iterations	Sequential algorithm execution time, sec	Execution time of parallel algorithm, sec			
		Number of threads			
		2	3	6	12
10	1.91	1.34	1.21	0.89	1.13
20	3.71	2.46	1.97	1.79	2.05
30	5.79	4.61	2.94	2.87	3.34
40	7.67	5.55	4.31	5.79	4.97
50	9.91	7.40	5.71	5.65	6.33
60	14.61	9.21	8.03	7.34	8.25

Table 4
The results of parallelization of the second stage using a 6-core processor

Number of iterations	3 threads		6 threads		12 threads	
	Acceleration	Efficiency	Acceleration	Efficiency	Acceleration	Efficiency
10	1.58	0.53	2.15	0.36	1.69	0.14
20	1.88	0.63	2.07	0.35	1.81	0.15
30	1.97	0.66	2.02	0.34	1.73	0.14
40	1.78	0.59	1.32	0.22	1.54	0.13
50	1.74	0.58	1.75	0.29	1.57	0.13
60	1.82	0.61	1.99	0.33	1.77	0.15

When using a 6-core processor we obtained acceleration equal to 2 and when choosing the number of streams to more than six, we observe a significant reduction in efficiency, which confirms the reliability of the results.

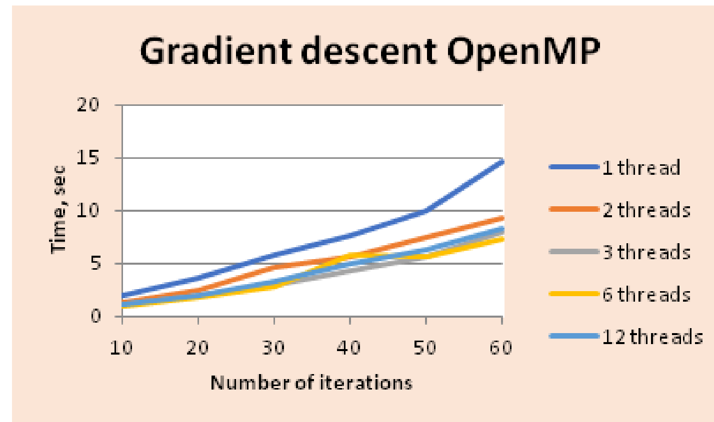

Figure3: Visualization of execution time at parallelization of the second stage (6-core processor)

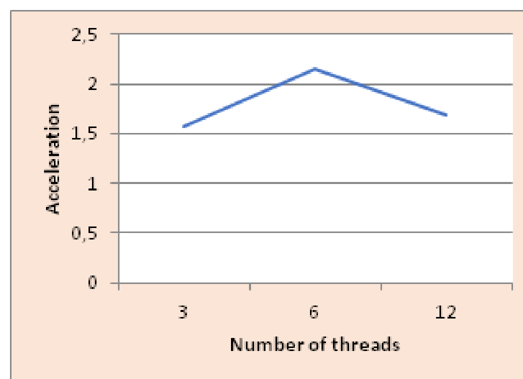

Figure 4: Acceleration obtained by parallelization of the second stage (6-core processor)

Sometimes significant benefits cannot be achieved through false sharing. Simultaneous updates of individual items in the same cache line coming from different processors invalidate entire lines of the cache, although these updates are logically independent of each other. Each update of an individual cache line item marks the line as invalid. Other processors that access another item on the same line see the string marked as invalid. They have to get a newer copy of the string from memory or elsewhere, even if the item they accessed has not been changed. This is because cache coherence is maintained based on the cache line, not for individual elements.

As a result, there will be an increase in the relationship between traffic and overhead costs. Also, while the cache line is being updated, access to items in the line is denied. This situation is called false exchange. If this happens often, the performance and scalability of OpenMP will be significantly affected. Therefore it is necessary to use various ways of distribution of iterations:

3. At this stage, the parallelization described in step 2 is performed, but with different distribution of iterations schedule (type, chunk_size) [19].

```
double costP(double** X, double* Y, double* Q, int m, int n, int k) {
    double* hQ = new double[m];
    hQ = hypothesisP(X, Q, m, n);
    double sum = 0;
    #pragma omp parallel for num_threads(2) schedule(static, 250000)
    for (int i = 0; i < m; i++) {
            sum += (hQ[i] - Y[i]) * X[i][k];
    }
    return sum / m;}
double* hypothesisP(double **X, double *Q, int m, int n) {
    double* result = new double[m];
    #pragma omp parallel for num_threads(2) schedule(static, 250000)
    for (int i = 0; i < m; i++) {
            result[i] = 0;
```

```
            for (int j = 0; j < n; j++) {
                  result[i] += X[i][j] * Q[j];}}
      return result;}
```

The time spent on the third stage of the gradient descent algorithm with different number of iterations and threads on a 2-core processor with different distribution of iterations schedule (type, chunk_size) are given in Table 5.

Table 5
The results of the parallelization of the third stage using a 2-core processor

Number of iterations	Sequential algorithm execution time, sec	Sequential algorithm execution time, sec			
		Number of threads			
		(2, 2) (static, 250000)	(2, 2) (dynamic, 50 000)	(2, 2) (dynamic, 10 000)	(2, 2) (guided, 10 000)
30	20.7	11	12.1	12.2	9.8

We can conclude that on average the parallel algorithm works much faster with different iteration distributions.

Similarly, the execution time when tested on a 6-core processor is given in Table 6, and the visualization of the results in Figure 5.

Table 6
The results of the parallelization of the third stage using a 6-core processor

Number of iterations	Sequential algorithm execution time, sec	**Execution time of parallel algorithm, sec**			
		Number of threads			
		(6, 6) (static, 250000)	(6, 6) (dynamic, 50 000)	(6, 6) (dynamic, 10 000)	(6, 6) (guided, 10 000)
30	5.79	3.65	3.25	2.63	2.7
40	7.67	5.6	4.03	4.10	4.7
50	9.91	8.7	5.3	5.28	5.45

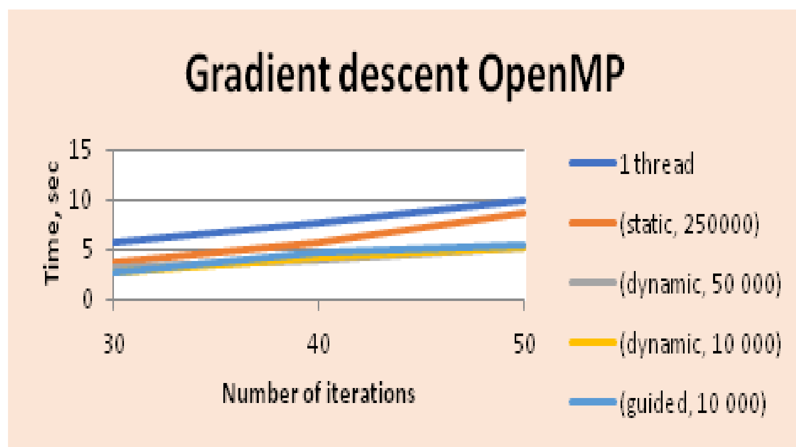

Figure 5: Visualization of execution time at parallelization of the third stage (6-core processor)

The parallelization algorithm of stochastic gradient descent is also performed in the work. The parallelization of this algorithm can occur by blocking the change of coefficients by each process, but

this approach obviously does not allow to obtain significant advantages. However, there is an interesting Hogwild algorithm! [23, 24], which allows you to change the coefficients without synchronization. This method works without any negative impact on the mathematical efficiency of the algorithm. The description of the algorithm is as follows:

Each thread draws a random example i from the training data.

- Thread reads current state of θ.
- Thread updates. $\theta \leftarrow (\theta - \alpha \nabla L(f_\theta(x_i), y_i))$.

Code snippet for two threads:

```
#pragma omp parallel for num_threads(2)
    for (int i = 0; i < m; i++) {
          for (int j = 0; j < n; j++) {
                tempQ[j] = Q[j] - alpha * (h(X[i], Q, n) - Y[i]) * X[i][j];

          }
          for (int k = 0; k < n; k++) {
                Q[k] = tempQ[k];
          }
    }
```

When the sample size is large enough (in this case 500,000) then one iteration is enough. Execution time for testing on 6 cores is given in Table 7 and Table 8 and data visualization in Figure 6.

Table 7
The time spent on performing a stochastic gradient descent at different numbers of threads

Sequential algorithm execution time, sec	**The execution time of the parallel algorithm, sec**			
	Number of threads			
	2	3	6	12
0.11	0.087	0.087	0.082	0.06

Table 8
Efficiency and acceleration when tested on a 6-core processor

3 threads		**6 threads**		**12 threads**	
Acceleration	**Efficiency**	**Acceleration**	**Efficiency**	**Acceleration**	**Efficiency**
1.3	0.45	1.34	0.22	1.83	0.15

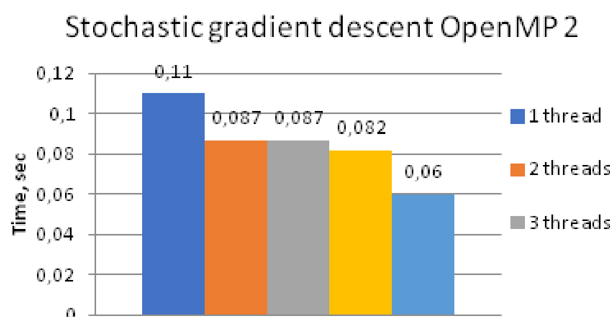

Figure 6: Visualization of execution time at parallelization of stochastic gradient descent (6-core processor)

Next, we present the results of parallelization of the gradient descent algorithm and stochastic gradient descent using MPI technology.

1. Parallelization of the gradient descent algorithm.

The parallelization occurs by the fact that the rows of the matrix $X[i]$ are distributed between the threads. Each thread calculates the result of the cost function on an equal amount of data allocated to it, then sends a calculation from each data selection and the main thread based on this data makes changes to the model. The disadvantage of this method is that it requires the exchange of information between threads, which can slow down the parallelization.

Code snippet:

```
DWORD Start = GetTickCount();
    if (ProcRank == 0) {
            double* tempQ = new double[n];
            for (int iter = 0; iter < 20; iter++) {
                    for (int qiter = 0; qiter < n; qiter++) {
                            for (int p = 1; p < ProcNum; p++) {
                                    MPI_Send(Q, n, MPI_DOUBLE, p, 1, MPI_COMM_WORLD);
                                    MPI_Send(&qiter, 1, MPI_INT, p, 1, MPI_COMM_WORLD);
                            }
                            double procSum = costMpi(X, Y, Q, m, n, qiter, ProcNum, ProcRank);
                            double allSum = procSum;
                            for (int pr = 1; pr < ProcNum; pr++)
                            {
                            double curSum = 0;
                            MPI_Recv(&curSum,  1,  MPI_DOUBLE,  pr,  MPI_ANY_TAG,  MPI_COMM_WORLD,
&Status)
                                                            allSum += curSum;
                            }
                            tempQ[qiter] = Q[qiter] - alpha * (allSum / m);
                    }
                    for (int qiter = 0; qiter < n; qiter++) {
                            Q[qiter] = tempQ[qiter];
                    }
            }
    }
    else {

            for (int t = 0; t < n * 20; t++) {
                    int k = 0;
                    MPI_Recv(Q, n, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, &Status);
                    MPI_Recv(&k, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &Status);
                    double procSum = costMpi(X, Y, Q, m, n, k, ProcNum, ProcRank);
                    MPI_Send(&procSum, 1, MPI_DOUBLE, 0, ProcRank, MPI_COMM_WORLD);
            }}
    MPI_Finalize();
    DWORD dwRunTime = GetTickCount() - Start;
```

Execution time for testing on a 6-core processor is given in Table 9 and Table 10, and visualization of results Figure 7 and Figure 8.

Table 9

Time spent performing the gradient descent algorithm using MPI technology (6-core processor)

Number of iterations	Sequential algorithm execution time, sec	Execution time of parallel algorithm, sec			
		Number of threads			
		2	3	6	12
10	1.59	0.84	0.59	0.37	0.34
20	3.38	1.77	1.19	0.77	0.69
30	5.09	2.58	1.79	1.16	1.02
40	6.83	3.49	2.39	1.54	1.37
50	8.23	4.41	3.0	1.93	1.74
60	10.1	5.15	3.63	2.31	2.08

Table 10
Efficiency and acceleration are based on MPI technology (6-core processor)

Number of iterations	3 threads		6 threads		12 threads	
	Acceleration	Efficiency	Acceleration	Efficiency	Acceleration	Efficiency
10	2.69	0.90	4.30	0.72	4.68	0.39
20	2.84	0.95	4.39	0.73	4.90	0.41
30	2.84	0.95	4.39	0.73	4.99	0.42
40	2.86	0.95	4.44	0.74	4.99	0.42
50	2.74	0.91	4.26	0.71	4.73	0.39
60	2.78	0.93	4.37	0.73	4.86	0.40

As you can see from the results, MPI technology allows you to significantly improve the results of parallelization of gradient descent. Acceleration is equal to 5 on 6-core processor.

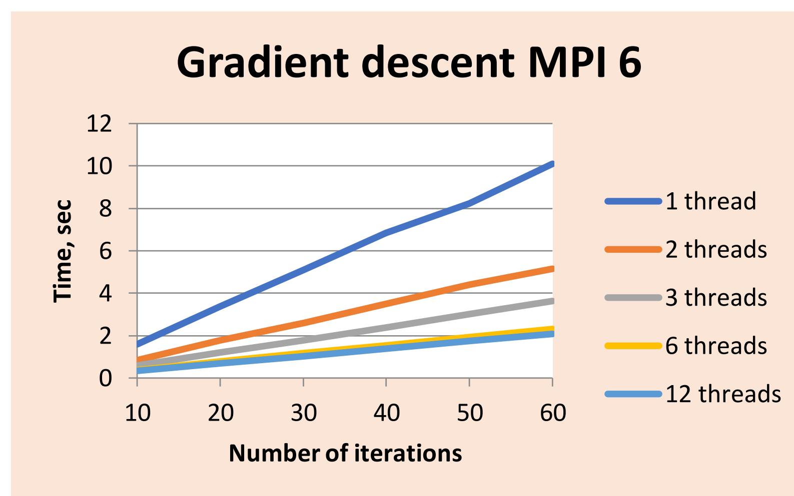

Figure 7: Runtime visualization based on MPI technology (6-core processor)

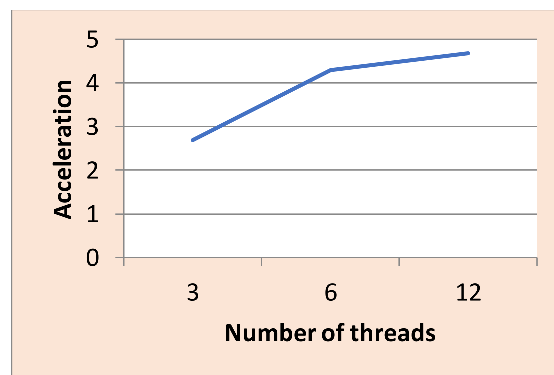

Figure 8: Acceleration is based on MPI technology (6-core processor)

2. Parallelization of the stochastic gradient descent algorithm.

The parallelization occurs by the fact that the rows of the matrix – $X[i]$ are distributed between the threads. Each thread searches for a local model, then sends the found model to the main thread, where all local models are averaged and we get a global model. The disadvantage of this method is that it requires additional calculations to find the model.

```
	DWORD Start = GetTickCount();
	double* tempQ = new double[n];
```

```
	double* localQ = new double[n];
	if (ProcRank == 0) {

		int end = m / ProcNum;
		for (int i = 0; i < end; i++) {
			for (int j = 0; j < n; j++) {
			tempQ[j] = Q[j] - alpha * (h(X[i], Q, n) - Y[i]) * X[i][j];
			}
			for (int k = 0; k < n; k++) {
				Q[k] = tempQ[k];
			}
		}
		for (int p = 1; p < ProcNum; p++) {
			MPI_Recv(localQ, n, MPI_DOUBLE, p, 1, MPI_COMM_WORLD, &Status);
			for (int qiter = 0; qiter < n; qiter++) {
				Q[qiter] += localQ[qiter];
			}
		}
		for (int qiter = 0; qiter < n; qiter++) {
			Q[qiter] /= ProcNum;
		}
	}
	else {
		int start = ProcRank * (m / ProcNum);
		int end = ProcRank * (m / ProcNum) + (m / ProcNum);

		for (int i = start; i < end; i++) {
			for (int j = 0; j < n; j++) {
			tempQ[j] = Q[j] - alpha * (h(X[i], Q, n) - Y[i]) * X[i][j];
			}
			for (int k = 0; k < n; k++) {
				Q[k] = tempQ[k];
			}
		}
		MPI_Send(Q, n, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
	}
```

Execution time for testing on a 6-core processor is given in Table 11 and Table 12, and visualization of results Figure 9.

Table 11

Time spent performing the stochastic gradient descent algorithm using MPI technology (6-core processor)

Sequential algorithm execution time, sec	The execution time of the parallel algorithm, sec			
	Number of threads			
	2	3	6	12
0.10	0.046	0.034	0.022	0.061

Table 12

Efficiency and acceleration are based on MPI technology (6-core processor)

3 threads		6 threads		12 threads	
Acceleration	**Efficiency**	**Acceleration**	**Efficiency**	**Acceleration**	**Efficiency**
2.17	0.72	4.54	0.75	1.63	0.13

To the parallel algorithm of the stochastic gradient descent, acceleration equals 4.5 when using MPI technology.

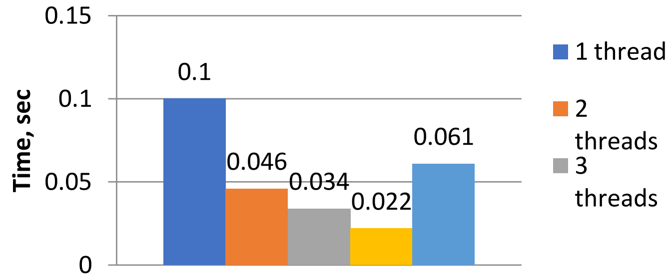

Figure 9: Visualization of the execution time of a parallel stochastic gradient descent algorithm based on MPI technology (6-core processor)

The following are the results of parallelization of the gradient descent algorithm and stochastic gradient descent based on a hybrid of MPI + OpenMP technology.

The algorithm described above is based on parallelization using MPI technology, but also uses fragments of the algorithm described in the last step using OpenMP technology.

Execution time when testing on a 2-core processor using hybrid technology for parallelization of the gradient descent method is given in Table 13.

Table 13

The time spent on performing the parallel algorithm of the gradient descent method

Number of iterations	Sequential algorithm execution time, sec	The execution time of the parallel algorithm, sec	
		Number of threads	
		2 omp parallel for	
20	5.95	4.1	4.5
30	11.026	4.8	4.9

Execution time when tested on a 2-core processor using hybrid technology for parallelization of the method of stochastic gradient descent is given in Table 14.

Table 14

Time spent performing the parallel algorithm of the stochastic gradient descent method

Sequential algorithm execution time, sec	Execution time of parallel algorithm, sec					
	Number of threads					
	2 omp parallel for		4 omp parallel for		8 omp parallel for	
	2	4	2	4	2	4
0.38	0.14	0.18	5.7	8.15	3.0	10.15

As you see the hybrid of OpenMP and MPI technologies allows for both methods when parallel to get accelerated to the number of a core of a multi-core computing system. This is especially relevant in a modern tendency to develop such architectures.

5. Conclusions

In this work, to optimize the regression analysis, the methods of gradient descent and stochastic gradient descent based on OpenMP and MPI technologies are paralleled. Without reducing the generality, a number of numerical experiments were performed on one network date. On the basis of which the efficiency of the proposed approach is analyzed in detail. Acceleration was achieved three times when parallelizing the algorithm and gradient descent using OpenMP for two consecutive cycles with the number of threads 2 and 4 on a six-core processor. The problem of erroneous exchange is considered, which often has a negative effect on acceleration during parallelization using OpenMP. Acceleration can be achieved twice with different distribution of iterations static, dynamic, guided and chunk_size. The Hogwild algorithm! was an interesting discovery and its application to parallelize the task.

The advantage was obtained by using MPI technology, but only with two parallel threads (when tested on a 2-core processor), because with the increase in the number of threads increases the need for more frequent communication, which begins to slow down the program. Significant acceleration of almost 5 times was achieved by parallelization of the algorithm of gradient and stochastic gradient descent on a 6-core processor.

It is also worth noting that each of the technologies has its advantages and disadvantages. With the help of OpenMP it is easy to programmatically implement parallelism, based on MPI technology the program is more difficult to develop and difficult to perform error finding. When using OpenMP technology, communication between threads is implicit, while on MPI it is explicit, so you need to think about how to implement communication between threads. MPI is portable to machines with shared and shared memory, OpenMP only for those with shared memory. In OpenMP there can be a problem with data placement, in MPI – no.

Thus, the problem was completely solved, the algorithm was parallelized by using different technologies.

With minor modifications depending on the requirements, the software implementation can be used in other big data processing tasks and tested on other data sets.

6. References

[1] R. Tkachenko, I. Izonin, Model and Principles for the Implementation of Neural-Like Structures Based on Geometric Data, Advances in Intelligent Systems and Computing 754 (2019). doi:10.1007/978-3-319-91008-6_58.

[2] R. M. Litnarovich, Construction and study of a mathematical model based on sources of experimental data by regression analysis, Manual. Rivne, MEGU, 2011.

[3] G. Bonaccorso, Machine Learning Algorithms: A reference guide to popular algorithms for data science and machine learning, Packt Publishing, 2017.

[4] B. Kedem, K. Fokianos, Regression Models for Time Series Analysis, Wiley, New York, 2002.

[5] B. Gurtner, “The Financial and Economic Crisis and Developing Countries”, International Development Policy 1 (2010):189-213.

[6] K.G. Serdyukov, O.V. Kaidash, Correlation-regression analysis of the impact of macroeconomic indicators on the volume of capital investment, KSU Bulletin Series Economic Sciences, volume 3 (2018): 65-69.

[7] Do. Thanh-Nghi, Parallel multiclass stochastic gradient descent algorithms for classifying million images with very-high-dimensional signatures into thousands classes, Vietnam J Computer Science 1 (2014) 107–115. doi:10.1007/s40595-013-0013-2.

[8] S. Kylee, O. Shashank, Parallelizing Gradient Descent, 2018. URL: https://shashank-ojha.github.io/ParallelGradientDescent/Final%20Report.pdf.

[9] M. E., Sahin, A. Lebedev, V. Alexandrov, Empirical Analysis of Stochastic Methods of Linear Algebra Computational Science, in: Proceedings of the ICCS 2020 20th International Conference, Amsterdam, The Netherlands, June 3–5, Part VII, 25 May 2020, 12143:539-549PMCID: PMC7304785.

[10] I. V. Hunko, Factors and indicators of forming enterprises’ investment attractiveness, J. Financial space 1, 2013, pp. 85-88.

[11] N. Shakhovska, S. Fedushko, N. Melnykova, I. Shvorob, Y. Syerov, Big Data analysis in development of personalized medical system, Procedia Computer Science 160, (2019). doi: 10.1016/j.procs.2019.09.461.
[12] N. Shakhovska, O. Veres, M. Hirnyak, Generalized formal model of Big Data, Econtechmod. an International Quarterly Journal 5, (2016), pp. 33–38.
[13] E.C. Alexopoulos, Introduction to multivariate regression analysis, Hippokratia 14, (2010), pp. 23-28.
[14] N. Polishchuk, S. Hrinyuk, S. Datsyuk, Comparison of optimization methods for neural networks training, Computer-integrated technologies education, science, production 35, 2020. URL: http://cit-journal.com.ua/index.php/cit/article/view/71.
[15] D. Mishchuk, A. Boychenko, Development of the concept of control system work for plaster works based on neural network, Mining, constructional, road and melioration machines 93, (2019), pp. 46-60.
[16] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.
[17] L. Bottou, Stochastic Learning, Advanced Lectures on Machine Learning, LNAI 3176, Springer, (2004) 146–168.
[18] L. Mochurad, K. Shakhovska, S. Montenegro, Parallel Solving of Fredholm Integral Equations of the First Kind by Tikhonov Regularization Method Using OpenMP Technology, in: Shakhovska N., Medykovskyy M. (eds) Advances in Intelligent Systems and Computing IV. CCSIT 2019. Advances in Intelligent Systems and Computing, volume 1080, 2020, pp. 25-35. doi: 10.1007/978-3-030-33695-0_3.
[19] M. J. Voss, OpenMP share memory parallel programming, Toronto, Kanada, 2003, 270 p.
[20] L. Mochurad, A. Solomiia, Optimizing the Computational Modeling of Modern Electronic Optical Systems, in: Lytvynenko V., Babichev S., Wójcik W., Vynokurova O., Vyshemyrskaya S., Radetskaya S. (eds) Lecture Notes in Computational Intelligence and Decision Making. ISDMCI 2019. Advances in Intelligent Systems and Computing, volume 1020, 2020, pp 597-608. doi: 10.1007/978-3-030-26474-1_41.
[21] Data sampling. URL: https://www.kaggle.com/wkirgsn/electric-motor-temperature.
[22] A. S. Antonov, MPI and OpenMP parallel programming technologies, M.: Moscow University Press, Series "Supercomputer education", (2012) 344 p.
[23] F. Niu, B. Recht, Ch. Re, S. J. Wright, HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent, in: Advances in Neural Information Processing Systems 24, 2011, pp. 693–701.
[24] Ch. De Sa, Ce Zhang, K. Olukotun, Ch. Re, Taming the wild: A unified analysis of HOGWILD!-style algorithms, in: NIPS, 2015.