Revised Pursuit Algorithm for Solving Non-stationary Linear Programming Problems on Modern Computing Clusters with Manycore Accelerators

Irina Sokolinskaya and Leonid Sokolinsky(✉)

South Ural State University,
76, Lenin prospekt, Chelyabinsk, Russia 454080
{Irina.Sokolinskaya,Leonid.Sokolinsky}@susu.ru

Abstract. This paper is devoted to the new edition of the parallel *Pursuit* algorithm proposed the authors in previous works. The *Pursuit* algorithm uses Fejer's mappings for building pseudo-projection on polyhedron. The algorithm tracks changes in input data and corrects the calculation process. The previous edition of the algorithm assumed using a cube-shaped pursuit region with the number of K cells in one dimension. The total number of cells is K^n, where n is the problem dimension. This resulted in high computational complexity of the algorithm. The new edition uses a cross-shaped pursuit region with one cross-bar per dimension. Such a region consists of only $n(K-1)+1$ cells. The new algorithm is intended for cluster computing system with Xeon Phi processors.

Keywords: Non-stationary linear programming problem · Fejer's mappings · Pursuit algorithm · Massive parallelism · Cluster computing systems · MIC architecture · Intel Xeon Phi · Native mode · OpenMP

1 Introduction

In the papers [7,8], the authors proposed the new *Pursuit* algorithm for solving high-dimension, non-stationary, linear programming problem. This algorithm is focused on cluster computing systems. High-dimensional, non-stationary, linear programming problems with quickly-changing input data are often seen in modern economic-mathematical simulations. The non-stationary problem is characterized by the fact that the input data is changing during the process of its solving. One example of such problem is the problem of investment portfolio management by using algorithmic trading methods (see [1,2]). In such problems, the number of variables and inequalities in the constraint system can be in

I. Sokolinskaya—The reported study was partially funded by RFBR according to the research project No. 17-07-00352-a, and by Act 211 Government of the Russian Federation according to the contract No. 02.A03.21.0011.

© Springer International Publishing AG 2016
V. Voevodin and S. Sobolev (Eds.): RuSCDays 2016, CCIS 687, pp. 212–223, 2016.
DOI: 10.1007/978-3-319-55669-7_17

the tens and even hundreds of thousands, and the period of input data change is within the range of hundredths of a second. The first version of the algorithm designed by the authors used a cubic-shaped pursuit region with the quantity of K cells in one dimension. In this case, the total number of cells is equal to K^n, where n is the dimension of the problem. This results in the high computational complexity of the algorithm. In this paper, we describe a new edition of the *Pursuit* algorithm, which uses a cross-shaped pursuit region with one cross-bar per dimension and containing only $n(K-1)+1$ cells. The main part of the *Pursuit* algorithm is a subroutine of calculating the pseudoprojection on the polyhedron. Pseudoprojection uses Fejer's mappings to substitute the projection operation on a convex set [4]. The authors implemented this algorithm in C++ language parallel programming technology OpenMP 4.0 [6] and the vector instruction set of Intel C++ Compiler for Xeon Phi [9]. The efficiency of the algorithm implementation for coprocessor Xeon Phi with KNC architecture [10] was investigated using a scalable synthetic linear programming problem. The results of these experiments are presented in this paper. The rest of this paper is organized as follows. In Sect. 2, we give a formal statement of a linear programming problem and define Fejer's process and the projection operation on a polyhedron. Section 3 describes the new version of the algorithm with a cross-shaped pursuit region. Section 4 provides a description of the main subroutine and subroutine for calculating the pseudoprojection of the revised algorithm by using UML activity diagrams. Section 5 is devoted to investigation of the efficiency of Intel Xeon Phi coprocessor usage for computing pseudoprojection. In conclusion, we summarize the results obtained and propose the directions for future research.

2 Problem Statement

Given a linear programming problem

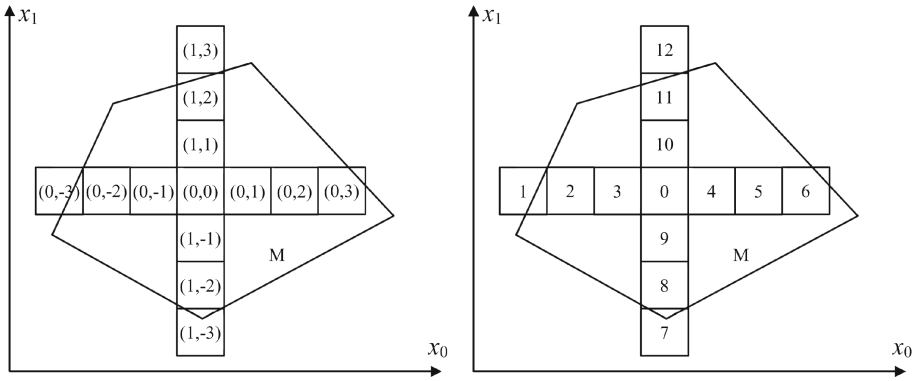
$$\max\left\{\langle c,x\rangle \,|Ax\le b,\ x\ge 0\right\}. \tag{1}$$

Let us define the Fejer's mapping $\varphi:\mathbb{R}^n\to\mathbb{R}^n$ as follows:

$$\varphi(x)=x-\sum_{i=1}^{m}\alpha_i\lambda_i\frac{\max\left\{\langle a_i,x\rangle-b_i,0\right\}}{\|a_i\|^2}a_i. \tag{2}$$

Let M be a polyhedron defined by the constraints of the linear programming problem (1). This polyhedron is always convex. It's known [3] that φ will be a single-valued continuous M-fejerian mapping for any $\alpha_i>0$ $(i=1,\ldots,m)$, $\sum\limits_{i=1}^{m}\alpha_i=1$, and $0<\lambda_i<2$. Putting in formula (2) $\lambda_i=\lambda$ and $\alpha_i=1/m$ $(i=1,\ldots,m)$, we get the formula

$$\varphi(x)=x-\frac{\lambda}{m}\sum_{i=1}^{m}\frac{\max\left\{\langle a_i,x\rangle-b_i,0\right\}}{\|a_i\|^2}a_i, \tag{3}$$



a) double index numbering (χ, η) b) linear numbering

Fig. 1. Cross-shaped pursuit region ($n = 2$, $K = 7$).

which is used in the *Pursuit* algorithm.

Let us set

$$\varphi^s(x) = \underbrace{\varphi \dots \varphi}_{s}(x). \tag{4}$$

Let the Fejerian process generated by mapping φ from an arbitrary initial approximation $x_0 \in \mathbb{R}^n$ to be a sequence $\{\varphi^s(x_0)\}_{s=0}^{+\infty}$. It is known that this Fejerian process converges to a point belonging to the set M:

$$\{\varphi^s(x_0)\}_{s=0}^{+\infty} \to \bar{x} \in M. \tag{5}$$

Let us denote this concisely as follows: $\lim\limits_{s\to\infty} \varphi^s(x_0) = \bar{x}$.

Let φ-*projection* (*pseudoprojection*) of point $x \in \mathbb{R}^n$ on polyhedron M be understood as the mapping $\pi_M^{\varphi}(x) = \lim\limits_{s\to\infty} \varphi^s(x)$.

3 Description of the Revised Algorithm

Without losing of generality, we may suppose all the processes are carried out in the region of positive coordinates.

Let n be the dimension of solution space. The new edition of the algorithm uses a cross-shaped pursuit region. This region consists of $n(K-1)+1$ hypercubical cells of equal size. The edges of all cells are codirectional with the coordinate axis. One of these cells designates the center. We will call this cell *central*. The remaining cells form an axisymmetrical cross-shaped figure around the central cell. An example of a cross-shaped pursuit region in a two-dimensional space is presented in Fig. 1. The total number of cells in the cross-shaped pursuit region can be calculated by the following formula:

$$P = n(K-1) + 1. \tag{6}$$

Each cell in the cross-shaped pursuit region is uniquely identified by a label being a pair of integer numbers (χ, η) such that $0 \le \chi < n$, $|\eta| \le (K-1)/2$. From an informal point of view, χ specifies the cell column codirectional to the coordinate axis indexed by χ, and η specifies the cell sequence number in the column in relation to the center cell. The corresponding double index numbering is shown in Fig. 1(a).

We will call the vertex closest to the origin a *zero vertex*. Let $(g_0, \ldots, g_{n-1})$ be the Cartesian coordinates of the central cell zero vertex. Let us denote by s the cell edge length. Then the Cartesian coordinates $(y_0, \ldots, y_{n-1})$ of the zero vertex of the cell (χ, η) are defined by the following formula:

$$y_j = \begin{cases} g_\chi + \eta s, & \textbf{if } j = \chi \\ g_j, & \textbf{if } j \ne \chi \end{cases} \tag{7}$$

for all $j = 0, \ldots, n-1$.

Informally, the algorithm with cross-shaped pursuit region can be described by the following sequence of steps.

1. Initially, we choose a cross-shaped pursuit region which has K cells in one dimension, with the cell edge length equal to s, in such a way, that the central cell has nonempty intersection with the polyhedron M.
2. The point $z = g$ is chosen as an initial approximation.
3. Given dynamically changing input data (A, b, c), for all cells of cross-shaped pursuit region, the pseudoprojection from the point z on the intersection of the cell and polyhedron M is calculated. If intersection is empty, then the corresponding cells are discarded.
4. If the obtained set of pseudoprojections is empty then we increase the cell size w times and go to the step 3.
5. If we receive a nonempty set of pseudoprojections then, for each cross bar, we choose the cell for which the cost function takes the maximal value at the point of pseudoprojection if it exist. For the set of cells obtained in such a way, we calculate the centroid and move point z at the position of the centroid.
6. If the distance between centroid and central cell is less than $\frac{1}{4}s$ then we decrease the cell length s 2 times.
7. If the distance between centroid and central cell is greater than $\frac{3}{4}s$ then we increase the cell length s 1.5 times.
8. We translate the cross-shaped pursuit region in such a way that its central point be situated at the centroid point found at the step 5.
9. Go to the step 3.

In the step 3, the pseudoprojections for the different cells can be calculated in parallel without data exchange between MPI-processes. This involves P MPI-processes, where P is determined by the formula (6). We use the linear cell numbering for the distributing the cells on the MPI-processes. Each cell of the cross-shaped pursuit region is assigned an unique number $\alpha \in \{0, \ldots, P-1\}$.

The sequential number α can be uniquely converted to the label (χ, η) by the following formulas[1]:

$$\chi = (\alpha - 1) \div (K - 1) \tag{8}$$

$$\eta = \begin{cases} 0, \ \textbf{if } \alpha = 0 \\ (\alpha - 1) \bmod \frac{K-1}{2} - \frac{K-1}{2}, \ \textbf{if } 0 \le (\alpha - 1) \bmod (K - 1) < \frac{K-1}{2} \\ (\alpha - 1) \bmod \frac{K-1}{2} + 1, \ \textbf{if } (\alpha - 1) \bmod (K - 1) \ge \frac{K-1}{2} \end{cases} \tag{9}$$

The reverse conversion of (χ, η) in α can be performed by the formula

$$\alpha = \begin{cases} 0, \ \textbf{if } \eta = 0 \\ \eta + \frac{K-1}{2} + \chi(K - 1) + 1, \ \textbf{if } \eta < 0 \\ \eta + \frac{K-1}{2} + \chi(K - 1), \ \textbf{if } \eta > 0 \end{cases} \tag{10}$$

Figure 1(b) shows the linear numbering corresponding to the double index numbering shown in Fig. 1(a).

4 Implementation of Revised Algorithm

This section describes the changes in the implementation of the new version of the *Pursuit* algorithm with reference to the description given in the paper [8].

4.1 Diagram of Main Subroutine

The activity diagram of the main subroutine of the *Pursuit* algorithm is shown in Fig. 2. In the loop *until* with label 1, the approximate solution $z = (z_0, \ldots, z_{n-1})$ of the linear programming problem (1) is permanently recalculated according to the algorithm outline presented in the Sect. 3. As an initial approximation, z may be chosen as an arbitrary point.

The main subroutine of the *Pursuit* algorithm is implemented as an independent process, which is performed until the variable *stop* takes the value of *true*.

The initial setting of the variable *stop* to the value *false* is performed by the root process corresponding to the main program. The same root process sets the variable *stop* to the value *true*, when the computations must be stopped.

In the body of the loop *until*, the following steps are performed. In the step 2, the K parallel threads are created. Each of them independently calculates the pseudoprojection from the point z on the intersection of the i-th cell and polyhedron M $(i = 0, \ldots, P - 1)$. Recall that P is equal to the number of MPI-processes that in turn is equal to the total number of cells in the cross-shaped pursuit region calculated by the formula (6). The activity diagram of subroutine π for calculating the pseudoprojection is described in Sect. 4.2.

In the loop *for* with label 3, for each cross bar $\chi = 0, \ldots, n - 1$, we calculate the sequential number α'_χ of the cell in this cross bar, in which the cost function C takes the maximum. It is calculated in the loop with label 5. In order to

[1] We use symbol $\div$ to denote the integer division.

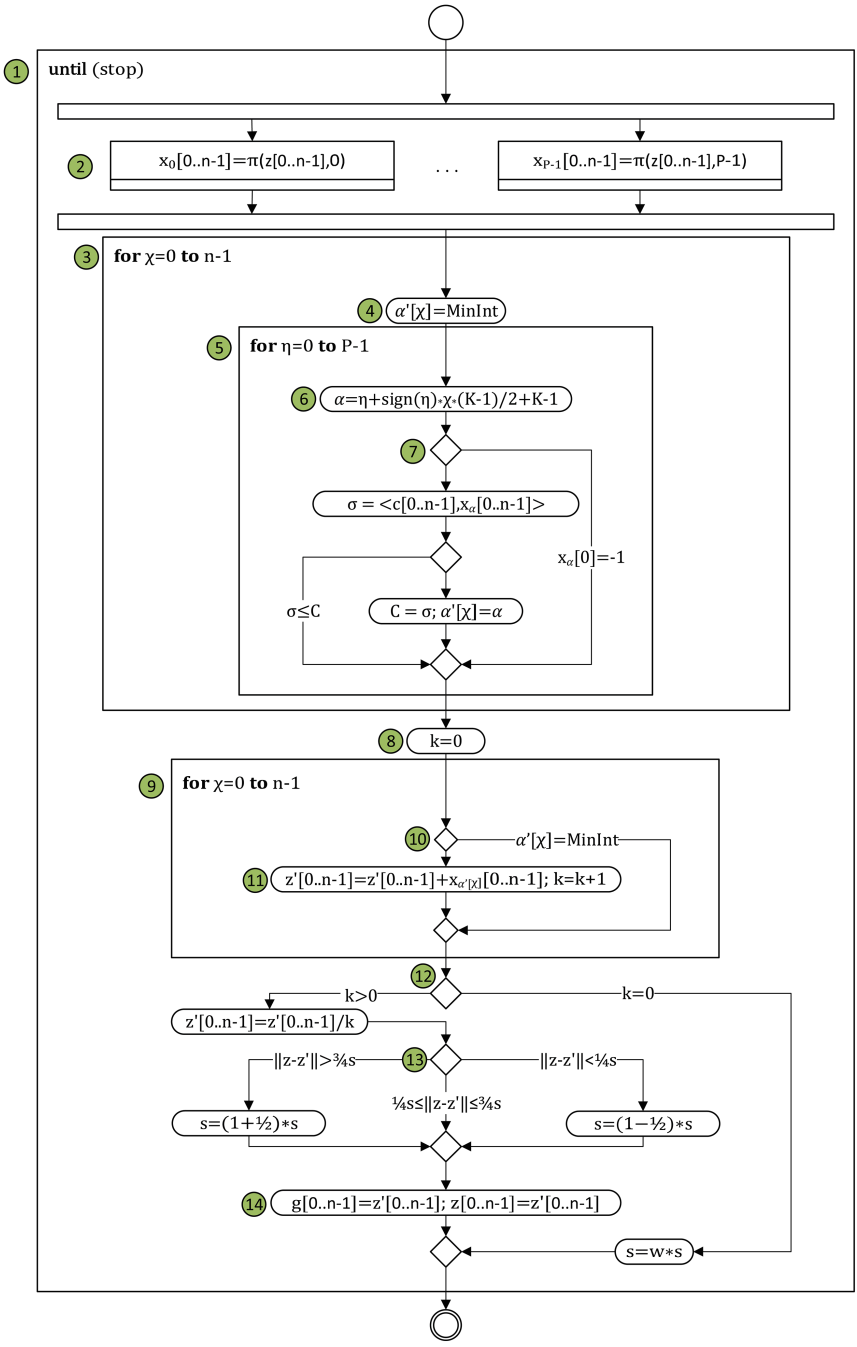


Fig. 2. Main subroutine of *Pursuit* algorithm.

guarantee the correct execution of the cycle 5, we initially assign the value *MinInt* to variable α'_χ. This value corresponds to the minimal value of the integer type. In the step 6, we calculate the sequential number α for the cell with label (χ, η) by using formula (10).

The subroutine π calculating the point $x_\alpha = (x_0, \ldots, x_{n-1})$ of pseudoprojection from the point z on the intersection of polyhedron M with the cell with number α assigns value -1 to x_0 when the pseudoprojection point x_α does not belong to the polyhedron M. This situation occurs when the intersection of the polyhedron M with the cell with number α is empty. If x_α belongs to the polyhedron then value of x_0 can't be negative because of our assumption that all the processes are carried out in the region of positive coordinates (see Sect. 3). This condition is checked in the step 7. Cases with $x_0 = -1$ are excluded from consideration. If all the cells in the current cross bar of pursuit region have empty intersection with polyhedron M then the variable α'_χ saves the value *MinInt*. This case is fixed in the step 10.

Then, in the loop 9, a new approximate solution z' of the problem (1) is calculated. Variable k takes the value which is equal to the number of cross bars having the nonempty intersections with polyhedron M. For this purpose, in the step 8, it is assigned the zero value. In the step 10, the cross bars having the empty intersections with polyhedron M are excluded from consideration. In the step 11, we calculate the sum of all pseudoprojection points, in which the cost function takes maximum, and assign this value to z'.

If in the step 12 we have $k = 0$, it means that the pursuit region has empty intersection with polyhedron M. In this case, the length s of cell edge is increased w times, and we go back to the step 1. The constant w is a parameter of the algorithm. If in the step 12 we have $k > 0$ then the new approximation z' is assigned the value which is equal to the centroid of all the cell selected in the loop labeled 9.

In the step 13, we investigate how far the new approximation z' is distant from the previous approximation z. If the distance between z' and z is greater than $\frac{3}{4}s$ then the length s of cell edge is increased 1.5 times. If the distance between z' and z is less than $\frac{3}{4}s$ then the length s of cell edge is decreased 2 times. If the distance between z' and z is greater than or equal $\frac{1}{4}s$ and less than or equal $\frac{3}{4}s$ then the length s of cell edge is unchanged. The values 1/4 and 3/4 are the parameters of the algorithm.

In the step 14, the pursuit region is translated by vector $(z' - z)$, z is assigned z', and computation is continued.

4.2 Diagram of Subroutine Calculating Pseudoprojection

In Fig. 3, the activity diagram of the subroutine calculating the pseudoprojection $x = \pi(z, \alpha)$ from the point z on the intersection of the polyhedron M and the cell with number α calculated by the formula (10) is presented. The pseudoprojection is calculated by organizing a Fejerian process (5) (see Sect. 2). In the step 1, the initialization of the variables used in iterative process is performed. The initial value of x is assigned to point z; the zero vertex y of the cell with number α is

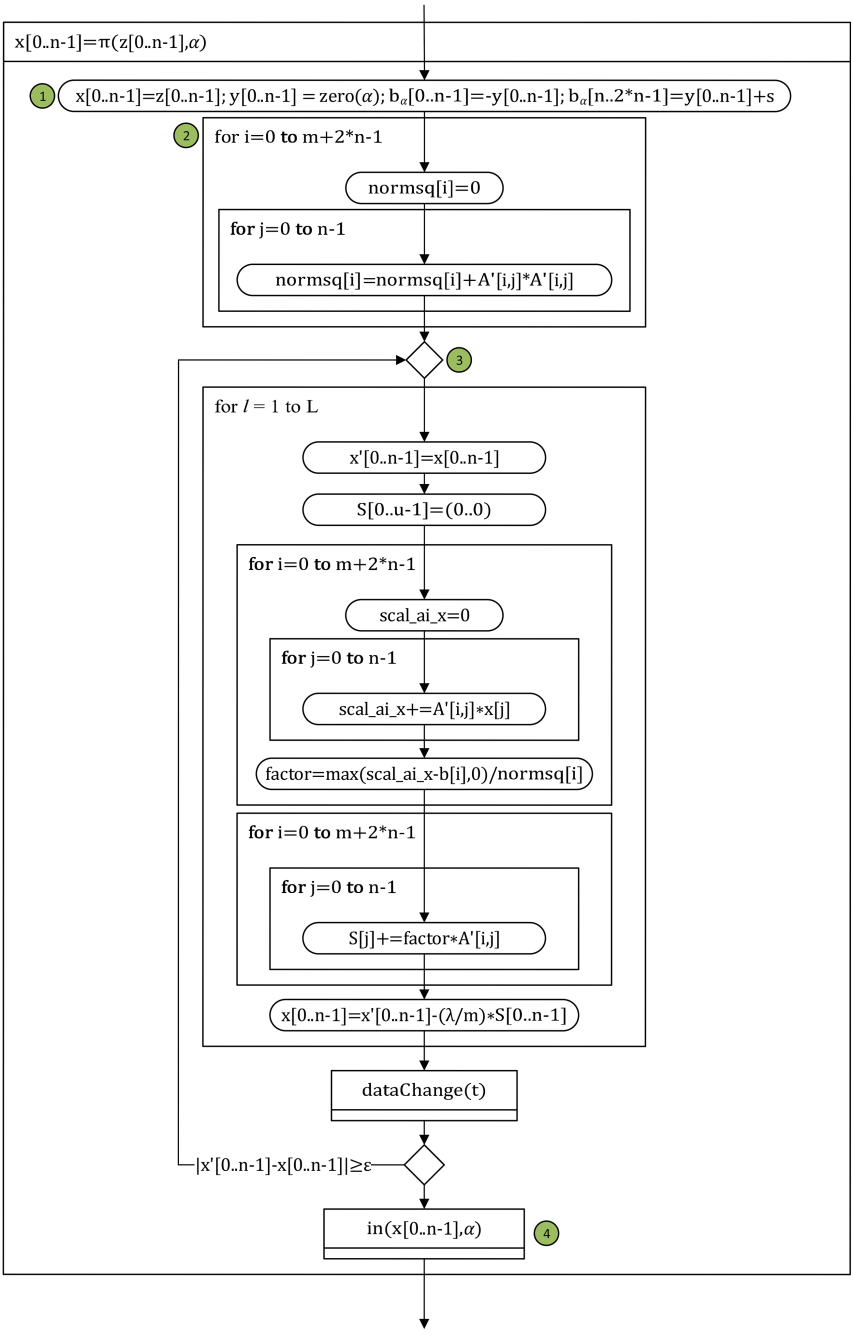


Fig. 3. Subroutine π calculating pseudoprojection.

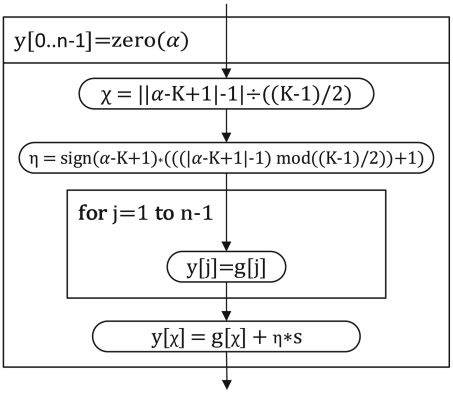


Fig. 4. Subroutine of zero vertex calculation for cell with number α.

$$
\begin{cases}
x_0 & & & & & \leq & 200 \\
 & x_1 & & & & \leq & 200 \\
 & & \ddots & & & \cdots & \cdots \\
 & & & & x_{n-1} & \leq & 200 \\
x_0 & + & x_1 & \cdots & + \; x_{n-1} & \leq & 200(n-1)+100 \\
x_0 & + & x_1 & \cdots & + \; x_{n-1} & \geq & 100 \\
x_0 & & & & & \geq & 0 \\
 & x_1 & & & & \geq & 0 \\
 & & \ddots & & & \cdots & \cdots \\
 & & & & x_{n-1} & \geq & 0
\end{cases}
$$

$$Q_{\max}(x) = 2x_0 + 2x_1 + \ldots + 2x_{n-2} + x_{n-1}$$

Fig. 5. *Model-n* synthetic problem.

calculated by using subroutine *zero* (see Fig. 4); the variable part of extended column b' of the constraint system is obtained by intersecting the polyhedron M and the cell with number α is defined (see [8]). In the loop 2, we calculate *normsq* being a vector of squares of norms of rows of the extended matrix A': $normsq_i = \|a'_i\|^2$ (see [8]).

In step 1, we organize an iterative process which calculates a pseudoprojection based on the formula (3). The subroutine *dataChange* changes the input data every t seconds (where t is a positive number, which can take a value less than 1).

The iterative process is terminated when the distance between the last two approximations x and x' is less than ε. In the step 1, the subroutine *in* (see [8]) checks belonging of the obtained pseudoprojection point x to the cell with the number α. If x does not belong to the cell with the number α, then $x[0]$ is assigned the value (-1). The constant ε defines a small positive number, which allows to correctly handle approximate values.

The activity diagram of the subroutine which calculates the zero vertex of the cell numbered α is presented in Fig. 4. Calculations are performed by using the formulas (8), (9) and (10).

5 Computing Pseudoprojection on Intel Xeon Phi

The most CPU intensive operation of the *Pursuit* algorithm is the operation computing the projections, which is implemented in the subroutine described in the Sect. 4.2. In order to achieve a high performance we investigated the possibility of effective use of coprocessors Intel Xeon Phi to calculate the pseudoprojection.

In our experiments, we exploited a self-made synthetic linear programming problem *Model-n* presented in the Fig. 5. Such the problems allow us to easily calculate the precise solution analytically. Therefore, they are well suited for algorithm validation and scalability evaluation.

We implemented the algorithm in C++ language using OpenMP. The task run was performed on Xeon Phi in native mode [10]. For computational experiments, we used the computer system "Tornado-SUSU" [5] with a cluster architecture. It includes 384 processor units connected by the InfiniBand QDR and Gigabit Ethernet. One processor unit includes two six-core CPU Intel Xeon X5680, 24 GB RAM and coprocessor Intel Xeon Phi SE10X (61 cores, 1.1 GHz) connected by PCI Express bus.

In the first series of experiments, we investigated the efficiency of parallelization of calculating pseudoprojection for different numbers of threads. The results are presented in Fig. 6. The pseudoprojections were calculated on the intersection of the polyhedron defined by constrains of linear programming problem *Model-n* and the cell with edge length $s = 20$ having coordinates of zero vertex equaling to $(100, \ldots, 100)$. The calculations were conducted for the dimensions $n = 1200, 9600, 12000$. The graphs show that the parallelization efficiency is strongly depends on the dimension of the problem. So, for the dimension $n = 1200$, the speedup curve actually stops growing after 15 threads. This means that a problem of such dimension cannot fully load all cores of Xeon Phi. The situation is changed when $n = 9600$ and more. Speedup becomes near-linear up to 60 threads, which is equal to the number of cores in Xeon Phi. Then parallelization efficiency is decreased, and for the dimension $n = 9600$, we even observe a performance degradation. The degradation is most evident at the point corresponding to the use of 180 threads. This dip is due to the fact that is not divisible by the dimension of 9600 divisible by 180, hence the compiler cannot uniformly distribute the iterations of the *parallel for* cycle between threads. The same situation takes place at the point "45 threads" for the dimensions 1200 and 9600. However, if the dimension is increased up to 12000, this effect is weakened.

In the second series of experiments, we compared the performance of two CPUs Intel Xeon and coprocessor Intel Xeon Phi. The results are presented in Fig. 7. The calculations were made for the dimensions $n = 9600, 12000, 19200$. For the Intel Xeon Phi, we made two builds: without the vectorization (MIC) and with the vectorization including data alignment (MIC+VECTOR). In all the cases, for the 2×CPU runs we used 12 threads, and for Xeon Phi runs we used 240 threads. The experiments show that for the dimension $n = 9600$ the 2×CPU outperform the coprocessor Xeon Phi, for the dimension $n = 12000$ the 2×CPU demonstrate the same performance as the coprocessor Xeon Phi does, and for the dimension $n = 19200$ the coprocessor Xeon Phi noticeably outperforms 2×CPU. At the same time, for the dimensions 9600, 12000 and 19200, the vectorization and data alignment provides a performance boost of 12%, 12.3% 20% correspondingly. Thus, we can conclude that the efficiency of the Xeon Phi coprocessor usage increases with the growth of the problem dimension. Simultaneously the significance of the vectorization and data alignment is increased.

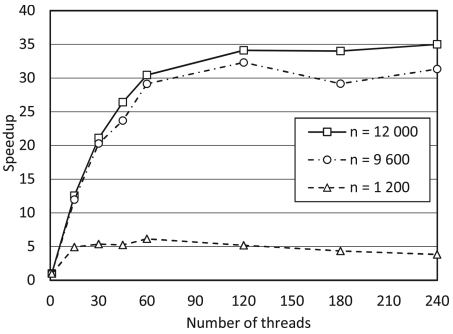


Fig. 6. Speedup of computing pseudoprojection on Xeon Phi.

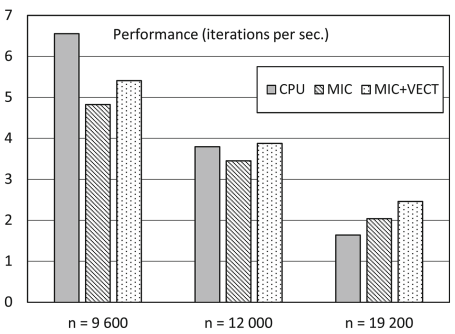


Fig. 7. Performance comparison of CPU and Xeon Phi (MIC).

6 Conclusion

The paper describes a new version of the *Pursuit* algorithm for solving high-dimension, non-stationary linear programming problem on the modern cluster computing systems. The distinctive feature of the new version is that it uses a cross-shaped pursuit region consisting of $n(K-1)+1$ cells, where n – dimension of the problem, K – number of cells in one cross-bar. The previous version of the algorithm uses a cube-shaped pursuit region consisting of K^n cells that results in high computational complexity of the algorithm. The results of the computational experiments investigating the efficiency of the coprocessor Xeon Phi use for pseudoprojection computation were presented. In these experiments, a synthetic linear programming problem was used. Studies have shown that the use of Intel Xeon Phi coprocessors is effective for high-dimension problems (over 10000). Our future goal is to investigate the efficiency of the proposed algorithm on the cluster computing systems using MPI technology.

References

1. Ananchenko, I.V., Musaev, A.A.: Torgovye roboty i upravlenie v khaoticheskikh sredakh: obzor i kriticheskiy analiz [Trading robots and management in chaotic environments: an overview and critical analysis]. Trudy SPIIRAN [SPIIRAS Proc.] **34**(3), 178–203 (2014)
2. Dyshaev, M.M., Sokolinskaya, I.M.: Predstavlenie torgovykh signalov na osnove adaptivnoy skol'zyashchey sredney Kaufmana v vide sistemy lineynykh neravenstv [Representation of trading signals based on Kaufman's adaptive moving average as a system of linear inequalities]. Vestnik Yuzhno-Ural'skogo gosudarstvennogo universiteta **2**(4), 103–108 (2013). Seriya: Vychislitel'naya matematika i informatika [Bulletin of South Ural State University. Series: Computational Mathematics and Software Engineering]
3. Eremin, I.I.: Fejerovskie metody dlya zadach linejnoj i vypukloj optimizatsii [Fejer's Methods for Problems of Convex and Linear Optimization], 200 p. Publishing of the South Ural State University, Chelyabinsk (2009)

4. Ershova, A.V., Sokolinskaya, I.M.: O skhodimosti masshtabiruemogo algoritma postroeniya psevdoproektsii na vypukloe zamknutoe mnozhestvo [About convergence of scalable algorithm of constructing pseudo-projection on convex closed set]. Vestnik YuUrGU. Seriya "Matematicheskoe modelirovanie i programmirovanie" [Bulletin of South Ural State University. Series: Mathematical simulation and programming], vol. 10, no. 37(254), pp. 12–21 (2011)
5. Kostenetskiy, P.S., Safonov, A.Y.: SUSU supercomputer resources. In: Proceedings of the 10th Annual International Scientific Conference on Parallel Computing Technologies (PCT 2016), CEUR Workshop Proceedings, vol. 1576, pp. 561–573. CEURWS (2016)
6. OpenMP Application Program Interface. Version 4.0, July 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
7. Sokolinskaya, I., Sokolinsky, L.: Solving unstable linear programming problems of high dimension on cluster computing systems. In: Proceedings of the 1st Russian Conference on Supercomputing – Supercomputing Days (RuSCDays 2015), Moscow, Russian Federation, 28–29 September 2015, CEUR Workshop Proceedings, vol. 1482, pp. 420–427. CEURWS (2015)
8. Sokolinskaya, I., Sokolinsky, L.: Implementation of parallel pursuit algorithm for solving unstable linear programming problems. In: Proceedings of the 10th Annual International Scientific Conference on Parallel Computing Technologies (PCT 2016), Arkhangelsk, Russia, 29–31 March 2016, CEUR Workshop Proceedings, vol. 1576, pp. 685–698. CEURWS (2016)
9. Supalov, A., Semin, A., Klemm, M., Dahnken, C.: Optimizing HPC Applications with Intel Cluster Tools. 269 p. Apress (2014). doi:10.1007/978-1-4302-6497-2
10. Thiagarajan, S.U., Congdon, C., Naik, S., Nguyen, L.Q.: Intel Xeon Phi coprocessor developer's quick start guide. White Paper, Intel (2013) https://software.intel.com/sites/default/files/managed/ee/4e/intel-xeon-phi-coprocessor-quick-start-developers-guide.pdf