FRaGenLP: A Generator of Random Linear Programming Problems for Cluster Computing Systems

Leonid B. Sokolinsky(✉) and Irina M. Sokolinskaya

South Ural State University (National Research University), 76, Lenin prospekt, Chelyabinsk 454080, Russia
{leonid.sokolinsky,irina.sokolinskaya}@susu.ru

Abstract. The article presents and evaluates a scalable FRaGenLP algorithm for generating random linear programming problems of large dimension n on cluster computing systems. To ensure the consistency of the problem and the boundedness of the feasible region, the constraint system includes $2n + 1$ standard inequalities, called support inequalities. New random inequalities are generated and added to the system in a manner that ensures the consistency of the constraints. Furthermore, the algorithm uses two likeness metrics to prevent the addition of a new random inequality that is similar to one already present in the constraint system. The algorithm also rejects random inequalities that cannot affect the solution of the linear programming problem bounded by the support inequalities. The parallel implementation of the FRaGenLP algorithm is performed in C++ through the parallel BSF-skeleton, which encapsulates all aspects related to the MPI-based parallelization of the program. We provide the results of large-scale computational experiments on a cluster computing system to study the scalability of the FRaGenLP algorithm.

Keywords: Random linear programming problem · Problem generator · FRaGenLP · Cluster computing system · BSF-skeleton

1 Introduction

The era of big data [1,2] has generated large-scale linear programming (LP) problems [3]. Such problems arise in economics, industry, logistics, statistics, quantum physics, and other fields. To solve them, high-performance computing systems and parallel algorithms are required. Thus, the development of new parallel algorithms for solving LP problems and the revision of current algorithms have become imperative. As examples, we can cite the works [4–9]. The development of new parallel algorithms for solving large-scale linear programming

I. M. Sokolinskaya—The reported study was partially funded by the Russian Foundation for Basic Research (project No. 20-07-00092-a) and the Ministry of Science and Higher Education of the Russian Federation (government order FENU-2020-0022).

© Springer Nature Switzerland AG 2021
L. Sokolinsky and M. Zymbler (Eds.): PCT 2021, CCIS 1437, pp. 164–177, 2021.
https://doi.org/10.1007/978-3-030-81691-9_12

problems involves testing them on benchmark and random problems. One of the most well-known benchmark repositories of linear programming problems is Netlib-Lp [10]. However, when debugging LP solvers, it is often necessary to generate random LP problems with certain characteristics, with the dimension of the space and the number of constraints being the main ones. The paper [11] suggested one of the first methods for generating random LP problems with known solutions. The method allows generating test problems of arbitrary size with a wide range of numerical characteristics. The main idea of the method is as follows. Take as a basis an LP problem with a known solution and then randomly modify it so that the solution does not change. The main drawback of the method is that fixing the optimal solution in advance significantly restricts the random nature of the resulting LP problem.

The article [12] describes the GENGUB generator, which constructs random LP problems with a known solution and given characteristics, such as the problem size, the density of the coefficient matrix, the degeneracy, the number of binding inequalities, and others. A distinctive feature of GENGUB is the ability to introduce generalized upper bound constraints, defined to be a (sub)set of constraints in which each variable appears at most once (i.e., has at most one nonzero coefficient). This method has the same drawback as the previous one: by preliminarily fixing the optimal solution, one significantly restricts the random nature of the resulting LP problem.

The article [13] suggests a method for generating random LP problems with a preselected solution type: bounded or unbounded, unique or multiple. Each of these structures is generated using random vectors with integer components whose range can be given. Next, an objective function that satisfies the required conditions, i.e., leads to a solution of the desired type, is obtained. The LP problem generator described in [13] is mainly used for educational purposes and is not suitable for testing new linear programming algorithms due to the limited variety of generated problems.

In the present paper, we suggest an alternative method for generating random LP problems. The method has the peculiarity of generating feasible problems of a given dimension with an unknown solution. The generated problem is fed to the input of the tested LP solver, and the latter outputs a solution that must be validated. The validator program (see, for example, [14]) validates the obtained solution. The method we suggest for generating random LP problems is named FRaGenLP (Feasible Random Generator of LP) and is implemented as a parallel program for cluster computing systems. The rest of the article is as follows. Section 2 provides a formal description of the method for generating random LP problems and gives a sequential version of the FRaGenLP algorithm. In Sect. 3, we discuss the parallel version of the FRaGenLP algorithm. In Sect. 4, we describe the implementation of FRaGenLP using a parallel BSF-skeleton and give the results of large-scale computational experiments on a cluster computing system. The results confirm the efficiency of our approach. Section 5 summarizes the obtained results and discusses plans to use the FRaGenLP generator in the development of an artificial neural network capable of solving large LP problems.

2 Method for Generating Random LP Problems

The method suggested in this paper generates random feasible bounded LP problems of arbitrary dimension n with an unknown solution. To guarantee the correctness of the LP problem, the constraint system includes the following *support inequalities*:

$$\begin{cases} x_1 & & & & \leqslant & \alpha \\ & x_2 & & & \leqslant & \alpha \\ & & \ddots & & & \cdots\cdots \\ & & & x_n & \leqslant & \alpha \\ -x_1 & & & & \leqslant & 0 \\ & -x_2 & & & \leqslant & 0 \\ & & \ddots & & & \cdots\cdots \\ & & & -x_n & \leqslant & 0 \\ x_1 & +x_2 & \cdots & +x_n & \leqslant & (n-1)\alpha + \alpha/2 \end{cases} \tag{1}$$

Here, the positive constant $\alpha \in \mathbb{R}_{>0}$ is a parameter of the FRaGenLP generator. The number of support inequalities is $2n+1$. The number of random inequalities is determined by a parameter $d \in \mathbb{Z}_{\geqslant 0}$. The total number m of inequalities is defined by the following equation:

$$m = 2n + 1 + d. \tag{2}$$

The coefficients of the objective function are specified by the vector

$$c = \theta\,(n, n-1, n-2, \ldots, 1)\,, \tag{3}$$

where the positive constant $\theta \in \mathbb{R}_{>0}$ is a parameter of the FRaGenLP generator that satisfies the following condition:

$$\theta \leqslant \frac{\alpha}{2}. \tag{4}$$

From now on, we assume that the LP problem requires finding a feasible point at which the maximum of the objective function is attained. If the number d of random inequalities is zero, then FRaGenLP generates an LP problem that includes only the support inequalities given in (1). In this case, the LP problem has the following unique solution:

$$\bar{x} = (\alpha, \ldots, \alpha, \alpha/2)\,. \tag{5}$$

If the number d of random inequalities is greater than zero, the FRaGenLP generator adds the corresponding number of inequalities to system (1). The coefficients $a_i = (a_{i1}, \ldots, a_{in})$ of the random inequality and the constant term b_i on the right side are calculated through the function $\mathrm{rand}(l, r)$, which generates

a random real number in the interval $[l, r]$ ($l, r \in \mathbb{R}; l < r$), and the function rsgn(), which randomly selects a number from the set $\{1, -1\}$:

$$\begin{aligned} a_{ij} &:= \mathrm{rsgn}() \cdot \mathrm{rand}(0, a_{\max}), \\ b_i &:= \mathrm{rsgn}() \cdot \mathrm{rand}(0, b_{\max}). \end{aligned} \tag{6}$$

Here, $a_{\max}, b_{\max} \in \mathbb{R}_{>0}$ are parameters of the FRaGenLP generator. The inequality sign is always "$\leqslant$". Let us introduce the following notations:

$$\mathrm{f}(x) = \langle c, x \rangle; \tag{7}$$

$$h = (\alpha/2, \ldots, \alpha/2)\,; \tag{8}$$

$$\mathrm{dist}_h(a_i, b_i) = \frac{|\langle a_i, h \rangle - b_i|}{\|a_i\|}; \tag{9}$$

$$\pi(h, a_i, b_i) = h - \frac{\langle a_i, h \rangle - b_i}{|a_i|^2} a_i. \tag{10}$$

Equation (7) defines the objective function of the LP problem. Here and further on, $\langle \cdot, \cdot \rangle$ stands for the dot product of vectors. Equation (8) defines the central point of the *bounding hypercube* specified by the first $2n$ inequalities of system (1). Furthermore, Eq. (9) defines a function $\mathrm{dist}_h(a_i, b_i)$ that gives the distance from the hyperplane $\langle a_i, x \rangle = b_i$ to the center h of the bounding hypercube. Here and below, $\|\cdot\|$ denotes the Euclidean norm. Equation (10) defines a vector-valued function that expresses the orthogonal projection of the point h onto the hyperplane $\langle a_i, x \rangle = b_i$.

To obtain a random inequality $\langle a_i, x \rangle \leqslant b_i$, we calculate the coordinates of the coefficient vector a_i and the constant term b_i using a pseudorandom rational number generator. The generated random inequality is added to the constraint system if and only if the following conditions hold:

$$\langle a_i, h \rangle \leqslant b_i; \tag{11}$$

$$\rho < \mathrm{dist}_h(a_i, b_i) \leqslant \theta; \tag{12}$$

$$\mathrm{f}\left(\pi\left(h, a_i, b_i\right)\right) > \mathrm{f}\left(h\right); \tag{13}$$

$$\forall l \in \{1, \ldots, i-1\} : \neg\mathrm{like}(a_i, b_i, a_l, b_l). \tag{14}$$

Condition (11) requires that the center of the bounding hypercube be a feasible point for the considered random inequality. If the condition does not hold, then the inequality $-\langle a_i, x \rangle \leqslant -b_i$ is added instead of $\langle a_i, x \rangle \leqslant b_i$. Condition (12) requires that the distance from the hyperplane $\langle a_i, x \rangle = b_i$ to the center h of the bounding hypercube be greater than ρ but not greater than θ. The constant $\rho \in \mathbb{R}_{>0}$ is a parameter of the FRaGenLP generator and must satisfy the condition $\rho < \theta$, where θ, in turn, satisfies condition (4). Condition (13) requires that the objective function value at the projection of the point h onto the hyperplane $\langle a_i, x \rangle = b_i$ be greater than the objective function value at the point h. This condition combined with (11) and (12) cuts off constraints that cannot affect

the solution of the LP problem. Finally, condition (14) requires that the new inequality be *dissimilar* from all previously added ones, including the support ones. This condition uses the Boolean function "like", which determines the *likeness* of the inequalities $\langle a_i, x\rangle \leqslant b_i$ and $\langle a_l, x\rangle \leqslant b_l$ through the following equation:

$$\text{like}(a_i, b_i, a_l, b_l) = \left\| \frac{a_i}{\|a_i\|} - \frac{a_l}{\|a_l\|} \right\| < L_{\max} \wedge \left| \frac{b_i}{\|a_i\|} - \frac{b_l}{\|a_l\|} \right| < S_{\min}. \tag{15}$$

The constants $L_{\max}, S_{\min} \in \mathbb{R}_{>0}$ are parameters of the FRaGenLP generator. In this case, the parameter $L_{\max}$ must satisfy the condition

$$L_{\max} \leqslant 0.7 \tag{16}$$

(we will explain the meaning of this constraint below). According to (15), inequalities $\langle a_i, x\rangle \leqslant b_i$ and $\langle a_l, x\rangle \leqslant b_l$ are *similar* if the following two conditions hold:

$$\left\| \frac{a_i}{\|a_i\|} - \frac{a_l}{\|a_l\|} \right\| < L_{\max}; \tag{17}$$

$$\left| \frac{b_i}{\|a_i\|} - \frac{b_l}{\|a_l\|} \right| < S_{\min}. \tag{18}$$

Condition (17) evaluates the measure of parallelism of the hyperplanes $\langle a_i, x\rangle = b_i$ and $\langle a_l, x\rangle = b_l$, which bound the feasible regions of the corresponding inequalities. Let us explain this. The unit vectors $e_i = a_i/\|a_i\|$ and $e_l = a_l/\|a_l\|$ are normal to the hyperplanes $\langle a_i, x\rangle = b_i$ and $\langle a_l, x\rangle = b_l$, respectively. Let us introduce the notation $\delta = \|e_i - e_l\|$. If $\delta = 0$, then the hyperplanes are parallel. If $0 \leqslant \delta < L_{\max}$, then the hyperplanes are considered to be *nearly parallel*.

Condition (18) evaluates the *closeness* of the parallel hyperplanes $\langle a_i, x\rangle = b_i$ and $\langle a_l, x\rangle = b_l$. Indeed, the scalar values $\beta_i = b_i/\|a_i\|$ and $\beta_l = b_l/\|a_l\|$ are the normalized constant terms. Let us introduce the notation $\sigma = |\beta_i - \beta_l|$. If $\sigma = 0$, then the parallel hyperplanes coincide. If the hyperplanes are nearly parallel and $0 \leqslant \sigma < S_{\min}$, then they are considered to be *nearly concurrent.*

Two linear inequalities in $\mathbb{R}^n$ are considered *similar* if the corresponding hyperplanes are nearly parallel and nearly concurrent.

The constraint (16) for the parameter $L_{\max}$ is based on the following proposition.

Proposition 1. *Let the two unit vectors $e, e' \in \mathbb{R}^n$ and the angle $\varphi < \pi$ between them be given. Then,*

$$\|e - e'\| = \sqrt{2(1 - \cos\varphi)}. \tag{19}$$

Proof. By the definition of the norm in Euclidean space, we have

$$\|e - e'\| = \sqrt{\sum_j (e_j - e'_j)^2} = \sqrt{\sum_j \left(e_j{}^2 - 2e_j e'_j + e'_j{}^2\right)}$$
$$= \sqrt{\sum_j e_j{}^2 - 2\sum_j e_j e'_j + \sum_j e'_j{}^2} = \sqrt{1 - 2\langle e_j, e'_j\rangle + 1}.$$

Thus,

$$\|e - e'\| = \sqrt{2\left(1 - \langle e_j, e'_j\rangle\right)}. \tag{20}$$

By the definition of the angle in Euclidean space, we have, for unit vectors,

$$\langle e_j, e'_j\rangle = \cos\varphi.$$

Substituting in (20) the expression obtained, we have

$$\|e - e'\| = \sqrt{2\left(1 - \cos\varphi\right)}.$$

The proposition is proven.

It is reasonable to consider that two unit vectors e, e' are nearly parallel if the angle between them is less than $\pi/4$. In this case, according to (19), we have

$$\|e - e'\| < \sqrt{2\left(1 - \cos\frac{\pi}{4}\right)}.$$

Taking into account that $\cos(\pi/4) \approx 0.707$, we obtain the required estimate:

$$\|e - e'\| < 0.7.$$

An example of a two-dimensional LP problem generated by FRaGenLP is shown in Fig. 1. The purple color indicates the line defined by the coefficients of the objective function; the black lines correspond to the support inequalities, and the red lines correspond to the random inequalities. For the sake of clarity, we use green dashed lines to plot the large and the small circles defined by the equations $(x_1 - 100)^2 + (x_2 - 100)^2 = 100^2$ and $(x_1 - 100)^2 + (x_2 - 100)^2 = 50^2$. According to condition (12), any random line must intersect the large green circle but not the small green circle. The semitransparent red color indicates the feasible region of the generated LP problem.

Algorithm 1 represents a sequential implementation of the described method. Step 1 assigns zero value to the counter k of random inequalities. Step 2 creates an empty list A to store the coefficients of the inequalities. Step 3 creates an empty list B to store the constant terms. Step 4 adds the coefficients and constant terms of the support inequalities (1) to the lists A and B, respectively. Step 5 generates the coefficients of the objective function according to (3). If the parameter d, which specifies the number of random inequalities, is equal to zero, then Step 6 passes the control to Step 19. Steps 7 and 8 generate the coefficients

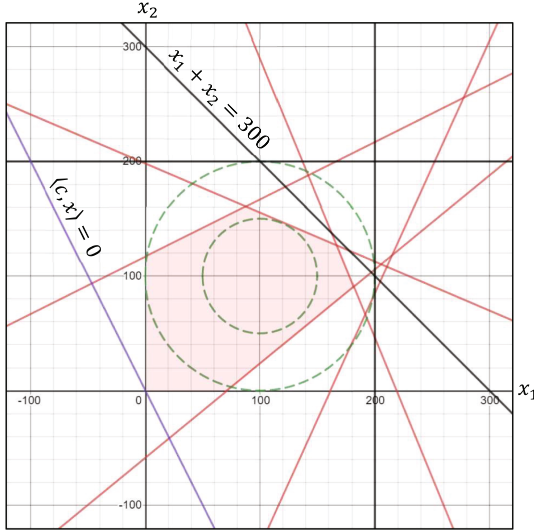

Fig. 1. Random LP problem with $n = 2$, $d = 5$, $\alpha = 200$, $\theta = 100$, $\rho = 50$, $S_{\min} = 100$, $L_{\max} = 0.35$, $a_{\max} = 1000$, and $b_{\max} = 10\,000$

and the constant term of the new random inequality. Step 9 checks condition (11). If the condition does not hold, then the signs of the coefficients and the constant term are reversed (Steps 10, 11). Step 12 checks condition (12). Step 13 checks condition (13). Step 14 checks condition (14). Step 15 appends the coefficients of the new random inequality to the list A ($+\!\!+$ denotes the concatenation of lists). Step 16 appends the constant term of the new random inequality to the list B. Step 17 increments the counter of added random inequalities by one. If the number of added random inequalities has not reached the given quantity d, then Step 18 passes the control to Step 7 to generate the next inequality. Step 19 outputs the results. Step 20 stops computations.

3 Parallel Algorithm for Generating Random LP Problems

Implicit loops generated by passing the control from Steps 12–14 to Step 7 of Algorithm 1 can result in high overheads. For example, during the generation of the LP problem represented in Fig. 1, there were 112 581 returns from Step 12 to label 7, 32 771 from Step 13, and 726 from Step 14. Therefore, generating a large random LP problem on a commodity personal computer can take many hours. To overcome this obstacle, we developed a parallel version of the FRaGenLP generator for cluster computing systems. This version is presented as Algorithm 2. It is based on the BSF parallel computation model [15,16], which assumes the master–slave paradigm [17]. According to the BSF model, the master node serves

Algorithm 1. Sequential algorithm for generating a random LP problem

Parameters: $n, d, \alpha, \theta, \rho, S_{\min}, L_{\max}, a_{\max}, b_{\max}$

1: $k := 0$
2: $A := [\,]$
3: $B := [\,]$
4: $\text{AddSupport}(A, B)$
5: **for** $j = n \dots 1$ **do** $c_j := \theta \cdot j$
6: **if** $d = 0$ **goto** 19
7: **for** $j = 1 \dots n$ **do** $a_j := \text{rsign}() \cdot \text{rand}(0, a_{\max})$
8: $b := \text{rsign}() \cdot \text{rand}(0, b_{\max})$
9: **if** $\langle a, h \rangle \leqslant b$ **goto** 12
10: **for** $j = 1 \dots n$ **do** $a_j := -a_j$
11: $b := -b$
12: **if** $\text{dist}_h(a, b) < \rho$ **or** $\text{dist}_h(a, b) > \theta$ **goto** 7
13: **if** $\text{f}(\pi(h, a, b)) \leqslant \text{f}(h)$ **goto** 7
14: **for all** $(\bar{a}, \bar{b}) \in (A, B)$ **do if** $\text{like}(a, b, \bar{a}, \bar{b})$ **goto** 7
15: $A := A +\!\!+ [a]$
16: $B := B +\!\!+ [b]$
17: $k := k + 1$
18: **if** $k < d$ **goto** 7
19: **output** A, B, c
20: **stop**

as a control and communication center. All slave nodes execute the same code but on different data.

Let us discuss Algorithm 2 in more detail. First, we look at the steps performed by the master node. Step 1 assigns zero value to the counter of random inequalities. Step 2 creates an empty list A_S to store the coefficients of the support inequalities. Step 3 creates an empty list B_S to store the constant terms of the support inequalities. Step 4 adds the coefficients and constant terms of the support inequalities (1) to the lists A_S and B_S, respectively. Step 5 generates the coefficients of the objective function according to (3). Step 6 outputs the coefficients and constant term of the support inequalities. If the parameter d, which specifies the number of random inequalities, is equal to zero, then Step 7 passes the control to Step 36, which terminates the computational process in the master node. Step 8 creates an empty list A_R to store the coefficients of the random inequalities. Step 9 creates an empty list B_R to store the constant terms of the random inequalities. In Step 18, the master node receives one random inequality from each slave node. Each of these inequalities satisfies conditions (11)–(13) and is not similar to any of the support inequalities. These conditions are ensured by the slave nodes. In the loop consisting of Steps 19–32, the master node checks all received random inequalities for similarity with the random inequalities previously included in the lists A_R and B_R. The similar new inequalities are rejected, and the dissimilar ones are added to the lists A_R and B_R. In this case, the inequality counter is increased by one each time some inequality

Algorithm 2. Parallel algorithm for generating a random LP problem

Parameters: $n, d, \alpha, \theta, \rho, S_{\min}, L_{\max}, a_{\max}, b_{\max}$

	Master		**Slave (l=1,...,L)**
1:	$k := 0$	1:	**if** $d = 0$ **goto** 36
2:	$A_S := [\,]$	2:	$A_S := [\,]$
3:	$B_S := [\,]$	3:	$B_S := [\,]$
4:	AddSupport(A_S, B_S)	4:	AddSupport(A_S, B_S)
5:	**for** $j = n \ldots 1$ **do** $c_j := \theta \cdot j$	5:	**for** $j = 1 \ldots n$ **do**
6:	**output** A_S, B_S, c	6:	$\quad a_j^{(l)} := \mathrm{rsign}() \cdot \mathrm{rand}(0, a_{\max})$
7:	**if** $d = 0$ **goto** 36	7:	**end for**
8:	$A_R := [\,]$	8:	$b^{(l)} := \mathrm{rsign}() \cdot \mathrm{rand}(0, b_{\max})$
9:	$B_R := [\,]$	9:	**if** $\langle a^{(l)}, h \rangle \leq b^{(l)}$ **goto** 12
10:		10:	**for** $j = 1 \ldots n$ **do** $a_j^{(l)} := -a_j^{(l)}$
11:		11:	$b^{(l)} := -b^{(l)}$
12:		12:	**if** $\mathrm{dist}_h(a^{(l)}, b^{(l)}) < \rho$ **goto** 5
13:		13:	**if** $\mathrm{dist}_h(a^{(l)}, b^{(l)}) > \theta$ **goto** 5
14:		14:	**if** $\mathrm{f}(\pi(h, a^{(l)}, b^{(l)})) \leqslant \mathrm{f}(h)$ **goto** 5
15:		15:	**for all** $(\bar{a}, \bar{b}) \in (A_S, B_S)$ **do**
16:		16:	$\quad$ **if** like$(a^{(l)}, b^{(l)}, \bar{a}, \bar{b})$ **goto** 5
17:		17:	**end for**
18:	**RecvFromSlaves** $a^{(1)}, b^{(1)}, ..., a^{(L)}, b^{(L)}$	18:	**SendToMaster** $a^{(l)}, b^{(l)}$
19:	**for** $l = 1 \ldots L$ **do**	19:	
20:	$\quad isLike := false$	20:	
21:	$\quad$ **for all** $(\bar{a}, \bar{b}) \in (A_R, B_R)$ **do**	21:	
22:	$\qquad$ **if** like$(a^{(l)}, b^{(l)}, \bar{a}, \bar{b})$ **then**	22:	
23:	$\qquad\quad isLike := true$	23:	
24:	$\qquad\quad$ **goto** 27	24:	
25:	$\qquad$ **end if**	25:	
26:	$\quad$ **end for**	26:	
27:	$\quad$ **if** $isLike$ **continue**	27:	
28:	$\quad A_R := A_R +\!\!+ [a^{(l)}]$	28:	
29:	$\quad B_R := B_R +\!\!+ [b^{(l)}]$	29:	
30:	$\quad k := k + 1$	30:	
31:	$\quad$ **if** $k = d$ **goto** 33	31:	
32:	**end for**	32:	
33:	**SendToSlaves** k	33:	**RecvFromMaster** k
34:	**if** $k < d$ **goto** 18	34:	**if** $k < d$ **goto** 5
35:	**output** A_R, B_R	35:	
36:	**stop**	36:	**stop**

is added to the lists. If the required number of random inequalities has already been reached, then Step 31 performs an early exit from the loop. Step 33 sends the current number of added random inequalities to the slave nodes. If this quantity is less than d, then Step 34 passes the control to Step 18, which requests a new portion of random inequalities from the slave nodes. Otherwise, Step 35 outputs the results, and Step 36 terminates the computational process in the master node.

Let us consider now the steps performed by the l-th slave node. If the parameter d, which specifies the number of random inequalities, is equal to zero, then Step 1 passes the control to Step 36, which terminates the computational process in the slave node. Otherwise, Steps 2 and 3 create the empty lists A_S and B_S to store the support inequalities. Step 4 adds the coefficients and constant terms of the support inequalities (1) to the lists A_S and B_S, respectively. Steps 5–8 generate a new random inequality. Step 9 checks condition (11). If this condition does not hold, then the signs of the coefficients and the constant term are reversed (Steps 10 and 11). Steps 12–14 check conditions (12) and (13). Steps 15–17 check the similarity of the generated inequality to the support inequalities. If any one of these conditions does not hold, then the control is passed to Step 5 to generate a new random inequality. If all conditions hold, then Step 18 sends the constructed random inequality to the master node. In Step 33, the slave receives from the master the current number of obtained random inequalities. If this quantity is less than the required number, then Step 34 passes the control to Step 5 to generate a new random inequality. Otherwise, Step 36 terminates the computational process in the slave node.

4 Software Implementation and the Computational Experiments

We implemented the parallel Algorithm 2 in C++ through the parallel BSF-skeleton [18], which is based on the BSF parallel computation model [15] and encapsulates all aspects related to the parallelization of the program using the MPI library [19].

The BSF-skeleton requires the representation of the algorithm in the form of operations on lists using the higher-order functions *Map* and *Reduce*, defined by the Bird–Meertens formalism [20]. The required representation can be constructed as follows. Set the length of the *Map* and *Reduce* lists equal to the number of slave MPI processes. Define the *Map* list items as empty structures:

$$\text{struct PT_bsf_mapElem_T}\{\ \}\ .$$

Each element of the *Reduce* list stores the coefficients and the constant term of one random inequality $\langle a, x\rangle \leqslant b$:

$$\text{struct PT_bsf_reduceElem_T}\{\ \text{float a[n]; float b}\}\ .$$

Each slave MPI process generates one random inequality using the *PC_bsf_MapF* function, which executes Steps 5–17 of Algorithm 2. The slave

Table 1. Specifications of the "Tornado SUSU" computing cluster

Parameter	Value
Number of processor nodes	480
Processor	Intel Xeon X5680 (6 cores, 3.33 GHz)
Processors per node	2
Memory per node	24 GB DDR3
Interconnect	InfiniBand QDR (40 Gbit/s)
Operating system	Linux CentOS

MPI process stores the inequality that satisfies all conditions to its local *Reduce* list consisting of a single item. The master MPI process receives the generated elements from the slave MPI processes and places them in its *Reduce* list (this code is implemented in the problem-independent part of the BSF-skeleton). After that, the master MPI process checks each obtained inequality for similarity with the previously added ones. If no matches are found, the master MPI process adds the inequality just checked to its local *Reduce* list. These actions, corresponding to Steps 19–32 of Algorithm 2, are implemented as the standard function *PC_bsf_ProcessResults* of the BSF-skeleton. The source code of the FRaGenLP parallel program is freely available on the Internet at https://github.com/leonid-sokolinsky/BSF-LPP-Generator.

Using the program, we conducted large-scale computational experiments on the cluster computing system "Tornado SUSU" [21]. The specifications of the system are given in Table 1. The computations were performed for several dimensions, namely $n = 3000$, $n = 5500$, and $n = 15\,000$. The total numbers of inequalities were, respectively, 6301, 10 001, and 31 501. The corresponding numbers of random inequalities were 300, 500, and 1500, respectively. Throughout the experiments, we used the following parameter values: $\alpha = 200$ (the length of the bounding hypercube edge), $\theta = 100$ (the radius of the large hypersphere), $\rho = 50$ (the radius of the small hypersphere), $L_{\max} = 0.35$ (the upper bound of *near parallelism* for hyperplanes), $S_{\min} = 100$ (the minimum acceptable closeness for hyperplanes), $a_{\max} = 1000$ (the upper absolute bound for the coefficients), and $b_{\max} = 10\,000$ (the upper absolute bound for the constant terms).

The results of the experiments are shown in Fig. 2. Generating a random LP problem with 31 501 constraints with a configuration consisting of a master node and a slave node took 12 min. Generating the same problem with a configuration consisting of a master node and 170 slave nodes took 22 s. The analysis of the results showed that the scalability bound (the maximum of the speedup curve) of the algorithm significantly depends on the dimension of the problem. For $n = 3000$, the scalability bound was 50 processor nodes approximately. This bound increased up to 110 nodes for $n = 5000$, and to 200 nodes for $n = 15\,000$. A further increase in problem size causes the processor nodes to run

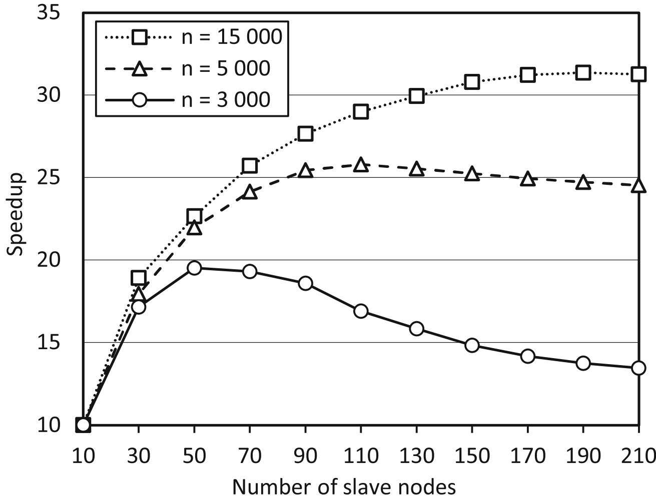

Fig. 2. Speedup curves of the FRaGenLP parallel algorithm for various dimensions

out of memory. It should be noted that the scalability bound of the algorithm significantly depends on the number of random inequalities too. Increasing this number by a factor of 10 resulted in a twofold reduction of the scalability bound. This is because an increase in the number of slave nodes results in a significant increase in the portion of sequential computations performed by the master node in Steps 19–32, during which the slave nodes are idle.

5 Conclusions

In this paper, we described the parallel FRaGenLP algorithm for generating random feasible bounded LP problems on cluster computing systems. In addition to random inequalities, the generated constraint systems include a standard set of inequalities called *support inequalities*. They ensure the boundedness of the feasible region of the LP problem. In geometric terms, the feasible region of the support inequalities is a hypercube with edges adjacent to the coordinate axes, and the vertex that is farthest from the origin is cut off. The objective function is defined in such a manner that its coefficients decrease monotonically. The coefficients and constant terms of the random inequalities are obtained using a random number generator. If the feasible region of a randomly generated inequality does not include the center of the bounding hypercube, then the sign of the inequality is reversed. Furthermore, not every random inequality is included in the constraint system. The random inequalities that cannot affect the solution of the LP problem for a given objective function are rejected. The inequalities, for which the bounding hyperplane intersects a small hypersphere located at the center of the bounding hypercube are also rejected. This ensures the feasibility of the constraint system. Moreover, any random inequality that is "similar" to

at least one of the inequalities already added to the system (including the support ones) is also rejected. To define the "similarity" of inequalities, two formal metrics are introduced for bounding hyperplanes: the measure of parallelism and the measure of closeness.

The parallel algorithm is based on the BSF parallel computation model, which relies on the master–slave paradigm. According to this paradigm, the master node serves as a control and communication center. All slave nodes execute the same code but on different data. The parallel implementation was performed in C++ through the parallel BSF-skeleton, which encapsulates all aspects related to the MPI-based parallelization of the program. The source code of the FRaGenLP generator is freely available on the Internet at https://github.com/leonid-sokolinsky/BSF-LPP-Generator.

Using this implementation, we conducted large-scale computational experiments on a cluster computing system. As the experiments showed, the parallel FRaGenLP algorithm demonstrates good scalability, up to 200 processor nodes for $n = 15\,000$. Generating a random LP problem with 31 501 constraints takes 22 s with a configuration consisting of 171 processor nodes. Generating the same problem with a configuration consisting of a processor node takes 12 min. The program was used to generate a dataset of 70 000 samples for training an artificial neural network capable of quickly solving large LP problems.

References

1. Jagadish, H.V., et al.: Big data and its technical challenges. Commun. ACM **57**(7), 86–94 (2014). https://doi.org/10.1145/2611567
2. Hartung, T.: Making big sense from big data. Front. Big Data. **1**, 5 (2018). https://doi.org/10.3389/fdata.2018.00005
3. Sokolinskaya, I., Sokolinsky, L.B.: On the solution of linear programming problems in the age of big data. In: Sokolinsky, L., Zymbler, M. (eds.) PCT 2017. CCIS, vol. 753, pp. 86–100. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67035-5_7
4. Sokolinsky, L.B., Sokolinskaya, I.M.: Scalable method for linear optimization of industrial processes. In: Proceedings – 2020 Global Smart Industry Conference, GloSIC 2020, pp. 20–26. Article number 9267854. IEEE (2020). https://doi.org/10.1109/GloSIC50886.2020.9267854
5. Sokolinskaya, I., Sokolinsky, L.B.: Scalability evaluation of NSLP algorithm for solving non-stationary linear programming problems on cluster computing systems. In: Voevodin, V., Sobolev, S. (eds.) Supercomputing, RuSCDays 2017. Communications in Computer and Information Science, vol. 793. pp. 40–53. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71255-0_4
6. Mamalis, B., Pantziou, G.: Advances in the parallelization of the simplex method. In: Zaroliagis, C., Pantziou, G., Kontogiannis, S. (eds.) Algorithms, Probability, Networks, and Games. LNCS, vol. 9295, pp. 281–307. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24024-4_17
7. Huangfu, Q., Hall, J.A.J.: Parallelizing the dual revised simplex method. Math. Program. Comput. **10**(1), 119–142 (2018). https://doi.org/10.1007/s12532-017-0130-5

8. Tar, P., Stagel, B., Maros, I.: Parallel search paths for the simplex algorithm. Central Eur. J. Oper. Res. **25**(4), 967–984 (2017). https://doi.org/10.1007/s10100-016-0452-9
9. Yang, L., Li, T., Li, J.: Parallel predictor-corrector interior-point algorithm of structured optimization problems. In: 3rd International Conference on Genetic and Evolutionary Computing, WGEC 2009, pp. 256–259 (2009). https://doi.org/10.1109/WGEC.2009.68
10. Gay, D.M.: Electronic mail distribution of linear programming test problems. Math. Program. Soc. COAL Bull. **13**, 10–12 (1985)
11. Charnes, A., Raike, W.M., Stutz, J.D., Walters, A.S.: On generation of test problems for linear programming codes. Commun. ACM **17**(10), 583–586 (1974). https://doi.org/10.1145/355620.361173
12. Arthur, J.L., Frendewey, J.O.: GENGUB: a generator for linear programs with generalized upper bound constraints. Comput. Oper. Res. **20**(6), 565–573 (1993). https://doi.org/10.1016/0305-0548(93)90112-V
13. Castillo, E., Pruneda, R.E., Esquivel, Mo.: Automatic generation of linear programming problems for computer aided instruction. Int. J. Math. Educ. Sci. Technol. **32**(2), 209–232 (2001). https://doi.org/10.1080/00207390010010845
14. Dhiflaoui, M., et al.: Certifying and repairing solutions to large LPs how good are LP-solvers? In: SODA03: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 255–256. Society for Industrial and Applied Mathematics, USA (2003)
15. Sokolinsky, L.B.: BSF: a parallel computation model for scalability estimation of iterative numerical algorithms on cluster computing systems. J. Parallel Distrib. Comput. **149**, 193–206 (2021). https://doi.org/10.1016/j.jpdc.2020.12.009
16. Sokolinsky, L.B.: Analytical estimation of the scalability of iterative numerical algorithms on distributed memory multiprocessors. Lobachevskii J. Math. **39**(4), 571–575 (2018). http://dx.doi.org/10.1134/S1995080218040121
17. Sahni, S., Vairaktarakis, G.: The master-slave paradigm in parallel computer and industrial settings. J. Glob. Optim. **9**(3–4), 357–377 (1996). https://doi.org/10.1007/BF00121679
18. Sokolinsky, L.B.: BSF-skeleton. User manual. arXiv:2008.12256 [cs.DC] (2020)
19. Gropp, W.: MPI 3 and beyond: why MPI is successful and what challenges it faces. In: Träff, J.L., Benkner, S., Dongarra, J.J. (eds.) EuroMPI 2012. LNCS, vol. 7490, pp. 1–9. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33518-1_1
20. Bird, R.S.: Lectures on constructive functional programming. In: Broy, M. (ed.) Constructive Methods in Computing Science. NATO ASI Series F: Computer and Systems Sciences, vol. 55, pp. 151–216. Springer, Heidlberg (1988). https://doi.org/10.1007/978-3-642-74884-4_5
21. Kostenetskiy, P., Semenikhina, P.: SUSU supercomputer resources for industry and fundamental science. In: Proceedings – 2018 Global Smart Industry Conference, GloSIC 2018, art. no. 8570068, p. 7. IEEE (2018). https://doi.org/10.1109/GloSIC.2018.8570068