VaLiPro: Linear Programming Validator for Cluster Computing Systems

Leonid B. Sokolinsky[1] (iD), *Irina M. Sokolinskaya*[1] (iD)

The article presents and evaluates a scalable algorithm for validating solutions to linear programming problems on cluster computing systems. The main idea of the method is to generate a regular set of points (validation set) on a small-radius hypersphere centered at the solution point submitted to validation. The objective function is computed at each point of the validation that belongs to the feasible region. If all the values are less than or equal to the value of the objective function at the point that is to be validated, then this point is the correct solution. The parallel implementation of the VaLiPro algorithm is written 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 VaLiPro algorithm.

Keywords: linear programming, solution validator, VaLiPro, parallel algorithm, cluster computing system, BSF-skeleton.

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 problems involves testing them on various benchmarks. One of the most well-known benchmark repositories of linear programming problems is the Netlib-Lp benchmark suite [10]. The solutions to all the problems from this repository are known. At the same time, in practice, it is often necessary to test a new algorithm on certain problems with unknown solutions. When testing an LP solver on such classes of problems, there is a need for validation (certification) and refinement of the obtained solution.

Several works have been devoted to the problem of certification and refinement of LP solutions. The paper [11] presents the LPlex system, which verifies and repairs a given solution to an LP problem for feasibility and optimality using exact arithmetic to guarantee the correctness of the results. The LPlex system can solve medium to large LP problems to optimality. Based on exact arithmetic (integer, rational, or modular), LPlex implements a module to detect block structures in matrices [12] and supports LU-factorizations of sparse matrices, the Bareiss method [13, 14], and the Wiedemann method [15]. The main drawback of the approach is that LPlex fails if the certified solution is not close enough to the optimal one. Koch [16] modified this approach to computing optimal solutions for the full set of Netlib-Lp instances. Rather than attempting to repair a nonoptimal basis with rational pivots, Koch recomputes a floating-point solution using greater precision in the floating-point representations. He employed the long double type that specifies 128-bit values. In [17], Applegate and co-authors extend Koch's methodology with an implementation that dynamically increases the precision of floating-point

---

[1]South Ural State University (National Research University), Chelyabinsk, Russian Federation

computations until a rational solution satisfying the optimality condition is obtained. They modify the conventional simplex algorithm by changing every floating-point type into the rational type provided by the GNU multiple precision arithmetic library (GMP) [18] and replacing every arithmetic operation in the original code with the corresponding GMP operations. The program starts with the best native floating-point precision and then increases it by about 50 % at each iteration (keeping the precision value a multiple of 32 bits to align with the typical word size). The main drawback of this approach is that the use of the multiple-precision arithmetic in the case of large and complex LP problems has high overheads. In [19], Panyukov and Gorbik try to overcome this disadvantage by using parallel computing on distributed memory. In this paper, they utilize rational arithmetic and propose two approaches for parallelizing the simplex method. The first method is based on the decomposition of the simplex tableau by columns. The second method is based on the modified simplex method using the inverse matrix and exploits the decomposition of the original matrix by columns and that of the inverse matrix by rows. However, the results of computational experiments are not sufficiently convincing for the following reasons: there is no comparison with the best sequential solutions; the computations were performed using only three sparse LP problems (the number of nonzero elements did not exceed 5 %); and, in addition, the scalability bound of the proposed parallel algorithm was only 16 processors. Another original approach is suggested by Gleixner and co-authors in [20]. This paper describes an iterative refinement procedure for computing extended-precision or exact solutions to LP problems. Arbitrarily precise solutions can be computed by solving a sequence of closely related LPs with limited-precision arithmetic. These LPs share the same constraint matrix as the original problem instance and are transformed only by modification of the objective function, right-hand sides, and variable bounds. This implementation is publicly available as an extension of the academic LP solver SoPlex.

All the methods discussed above concentrate on refining the approximate solution that has already been found. If the found solution is too far from the correct one, which means that there is an error in the algorithm, then the use of these methods becomes impractical. In addition, all of these algorithms have high computational complexity and do not allow efficient parallelization on large cluster computing systems. The method proposed in this article focuses on debugging and validating new LP algorithms on cluster computing systems. It is implemented as a parallel program, VaLiPro (Validator of Linear Program), which shows good scalability on multiprocessor computing systems with distributed memory. The rest of the paper is organized as follows. Section 1 provides a formal description of the proposed method for validating solutions to LP problems and presents a sequential version of the VaLiPro algorithm. The parallel version of the VaLiPro algorithm is discussed in section 2. Section 3 describes the implementation of the VaLiPro parallel algorithm in C++ using the BSF-skeleton. Here, we present the results of computational experiments on a cluster computing system, which confirm the efficiency of the proposed approach. In conclusion, we summarize the obtained results and expose plans for using the VaLiPro validator in the development of an artificial neural network capable of solving large-scale LP problems.

1. Method for Validating Solutions to LP Problems

Let the following linear programming problem be given in the Euclidean space $\mathbb{R}^n$:

$$\bar{x} = \arg\max\left\{\langle c, x \rangle \mid Ax \leqslant b, x \in \mathbb{R}^n\right\}, \tag{1}$$

where c is the vector of the objective function coefficients. Here and below, $\langle \cdot, \cdot \rangle$ stands for the dot product of vectors. Let us define $M = \{x \in \mathbb{R}^n \mid Ax \leqslant b\}$ as the feasible region of problem (1). By definition, the set M is convex and closed. From now on, we assume that M is a nonempty bounded set, i.e., problem (1) has at least one solution. Let $\tilde{x} \in \mathbb{R}^n$ be an approximate solution of problem (1) obtained using some LP solver that must be certified.

The main idea of the VaLiPro validation method is to construct a finite set of points V covering a hypersphere S of small (compared to the size of the polytope M) radius ρ centered at the certified solution point $\tilde{x}$:

Figure 1. Plotting the points of the validation set V on a three-dimensional sphere with $d = 5$

$$V \subset S = \left\{ x \in \mathbb{R}^n \mid \|x - \tilde{x}\|^2 = \rho^2 \right\}.$$

Here and below, $\|\cdot\|$ denotes the Euclidean norm. Let us compute the maximum of the objective function on the set $V \cap M$:

$$\bar{v} = \arg\max \left\{ \langle c, v \rangle \mid v \in V \cap M \right\}.$$

If $\left| \langle c, \bar{v} \rangle - \langle c, \tilde{x} \rangle \right| < \varepsilon$, then the approximation $\tilde{x}$ is considered correct. Otherwise, $\tilde{x}$ is considered an incorrect solution. Here, $\varepsilon \in \mathbb{R}_{>0}$ is a small positive constant that is a parameter of the validation algorithm.

Let us describe the method for constructing the validation set V. It is known [21] that the coordinates of any point $v = (v_1, \ldots, v_n)$ lying on the surface of the hypersphere S defined by the equation

$$\|x - \tilde{x}\|^2 = \rho^2$$

can be represented as follows:

$$
\begin{aligned}
v_1 &= \rho \cos(\phi_1); \\
v_j &= \rho \cos(\phi_j) \prod_{i=1}^{j-1} \sin(\phi_i) \ (j = 2, \ldots, n-2); \\
v_{n-1} &= \rho \sin(\theta) \prod_{i=1}^{n-2} \sin(\phi_i); \\
v_n &= \rho \cos(\theta) \prod_{i=1}^{n-2} \sin(\phi_i),
\end{aligned}
\tag{2}
$$

where $0 \leqslant \phi_j \leqslant \pi$ ($j = 1, \ldots, n-2$) and $0 \leqslant \theta < 2\pi$. Let us explain the method for generating the validation set V using a three-dimensional sphere (see Fig. 1). Fix an odd number of *parallels* $d \geqslant 3$ (poles are excluded). Set
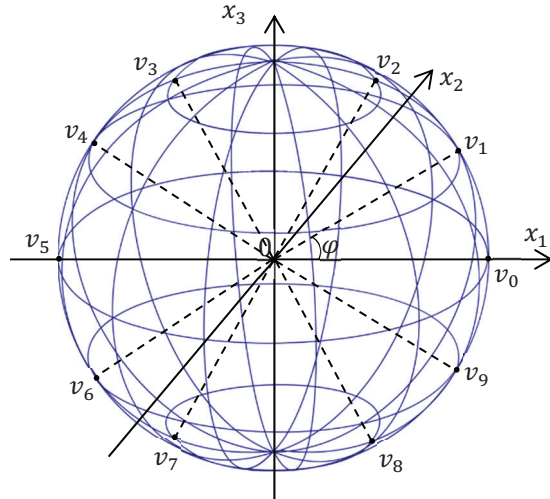
$$\varphi = \pi/d. \tag{3}$$

Algorithm 1 Generating points of the validation set

Parameters: d, ρ

---

a) With duplicates	b) Without duplicates

---

	a) With duplicates		b) Without duplicates
1:	$\varphi := \pi/d$	1:	$\varphi := \pi/d$
2:	**for** $j_{n-1} = 0 \ldots (2d-1)$ **do**	2:	**for** $j_{n-1} = 0 \ldots (2d-1)$ **do**
3:	$\theta := j_{n-1}\varphi$	3:	$\theta := j_{n-1}\varphi$
4:	**for** $j_{n-2} = 0 \ldots d$ **do**	4:	**for** $j_{n-2} = 1 \ldots d-1$ **do**
5:	$\phi_{n-2} := j_{(n-2)}\varphi$	5:	$\phi_{n-2} := j_{(n-2)}\varphi$
6:	$\ldots$	6:	$\ldots$
7:	**for** $j_2 = 0 \ldots d$ **do**	7:	**for** $j_2 = 1 \ldots d-1$ **do**
8:	$\phi_2 := j_2\varphi$	8:	$\phi_2 := j_2\varphi$
9:	**for** $j_1 = 0 \ldots d$ **do**	9:	**for** $j_1 = 1 \ldots d-1$ **do**
10:	$\phi_1 := j_1\varphi$	10:	$\phi_1 := j_1\varphi$
11:	$\varpi := 1$	11:	$\varpi := 1$
12:	$v_1 := \rho\cos(\phi_1)$	12:	$v_1 := \rho\cos(\phi_1)$
13:	**for** $l = 2 \ldots n-2$ **do**	13:	**for** $l = 2 \ldots n-2$ **do**
14:	$\varpi := \sin(\phi_{l-1})\varpi$	14:	$\varpi := \sin(\phi_{l-1})\varpi$
15:	$v_l := \rho\cos(\phi_l)\varpi$	15:	$v_l := \rho\cos(\phi_l)\varpi$
16:	**end for**	16:	**end for**
17:	$v_{n-1} := \rho\sin(\theta)\varpi$	17:	$v_{n-1} := \rho\sin(\theta)\varpi$
18:	$v_n := \rho\cos(\theta)\varpi$	18:	$v_n := \rho\cos(\theta)\varpi$
19:	**output** v	19:	**output** v
20:	**end for**	20:	**end for**
21:	**end for**	21:	**end for**
22:	$\ldots$	22:	$\ldots$
23:	**end for**	23:	**end for**
24:	**end for**	24:	**end for**
25:	**stop**	25:	**stop**

In the plane $(x_1, 0, x_3)$, we set the angles $0, \varphi, \ldots, (2d-1)\varphi$ starting from the axis $(0, x)$. At the intersection with the sphere, the resulting rays give the set of points $\{v_0, \ldots, v_{2d-1}\}$, which uniquely define d parallels. Then, on the plane $(x_1, 0, x_2)$, set the angles $0, \varphi, \ldots, (2d-1)\varphi$ from the axis $(0, x_1)$ and define d *meridians* in the same way. The intersections of parallels and meridians, excluding the poles, give the points that form a validation set on the three-dimensional sphere.

The described method for generating points of the validation set for $n \geqslant 3$ in the general case is given in Algorithm 1a. The nested loops with the headers in steps $2, 4, \ldots, 7$, and 9 generate the following spherical coordinates of a validation point:

$$(\rho, \phi_1, \phi_2, \ldots, \phi_{n-2}, \theta). \tag{4}$$

Algorithm 2 Function g (calculating the point v by its number k)

```
 1: function g(k, d, ρ)
 2:     u_{n-1} := ⌊k/(d-1)^{n-2}⌋
 3:     u_n := u_{n-1}
 4:     k := k  mod (d-1)^{n-2}
 5:     for j = (n-3)...0 do
 6:         u_j := ⌊k/(d-1)^j⌋ + 1
 7:         k := k  mod (d-1)^j
 8:     end for
 9:     ϖ := 1
10:     φ := π/d
11:     v_1 := ρ cos(u_1 φ)
12:     for j = 2...(n-2) do
13:         ϖ := ϖ sin(u_{j-1} φ)
14:         v_j := ρ cos(u_j φ) ϖ
15:     end for
16:     ϖ := ϖ sin(u_{n-2} φ)
17:     v_{n-1} := ρ sin(u_{n-1} φ) ϖ
18:     v_n := ρ  cos(u_n φ) ϖ
19:     return (v_1, ..., v_n)
20: end function
```

In steps 11–18, the spherical coordinates are converted to Cartesian coordinates by Equations (2). Multiplying the quantities of iterations of for-loops with headers 2, 4, ..., 7, and 9, we conclude that Algorithm 1a outputs $2d(d+1)^{n-2}$ validation points. However, there will be duplicates among the output points. The computational experiments showed that if one sets the dimension $n = 4$ and the number of parallels $d = 5$, then Algorithm 1a generates 189 duplicates with a total number of points equal to 360, which is more than 50 %. The duplicates are generated at iterations in which $\phi_i = 0$ or $\phi_i = \pi$, which corresponds to $j_i = 0$ and $j_i = d$ ($i = 1, ..., n-2$). The reason is that one of the factors $\sin(\phi_i)$ in (2) is equal to zero in this case, and therefore the variations of other factors cannot change the value of the corresponding coordinate. This issue can be solved without a major revision of Algorithm 1a, by changing the start values and end values of the control variables in loop headers 4, ..., 7, and 9, as is done in Algorithm 1b. This algorithm generates a validation set without duplicates but, at the same time, it loses a certain number of unique points. With $n = 4$ and $d = 5$, this quantity is 11, which is less than 7 % of the whole set after removing duplicates. Experiments have shown that such a loss does not significantly affect the accuracy of the validation algorithm. The number of points of the validation set V generated by Algorithm 1b is determined by the following equation:

$$|V| = 2d(d-1)^{n-2}. \tag{5}$$

The main drawback of Algorithm 1b is that the number of nested loops depends on the problem dimension, which does not allow using the dimension as a program parameter. To overcome this drawback, we use a vector-valued function that calculates the coordinates of a point by its sequential number k in the point sequence generated by Algorithm 1b (counting starts from zero). The definition of this function is given in Algorithm 2.

The final implementation of the VaLiPro method, using the vector-valued function g, is given in Algorithm 3. An additional parameter of this algorithm is the small positive constant ε (by default, $\varepsilon = 10^{-6}$), which compensates for possible numerical errors when comparing the values of the objective function in Step 5. Let us make several brief comments on the steps of Algorithm 3. Step 1 reads the source data of LP problem (1), the algorithm parameters, and the solution $\tilde{x}$ that is to be certified. Step 2 calculates the angle φ according to Equation (3). Step 3 begins the loop that varies the point number k from 0 to $2d(d-1)^{n-2} - 1$ as per Equation (5). Using the vector-valued function g (see Algorithm 2), Step 4 computes the next validation point v. Step 5 checks whether v belongs to the feasible region of problem (1) and compares the objective-function values at the points v and $\tilde{x}$. If the objective function takes a larger value at the point v, and this point is feasible, then the control is passed to Step 9, which prints a message stating that the certified solution is not correct. Otherwise, the next iteration of the loop

---
Algorithm 3 Validation of the LP solution $\widetilde{x}$

1: **input** $n, A, b, c, d, \rho, \varepsilon, \widetilde{x}$
2: $\varphi := \pi/d$
3: **for** $k = 0 \ldots 2d(d-1)^{n-2} - 1$ **do**
4: $\qquad v := g(k, d, \rho)$
5: $\qquad$ **if** $Av \leqslant b$ & $\langle c, v \rangle > \langle c, \widetilde{x} \rangle + \varepsilon$ **goto** 9
6: **end for**
7: **output** "Solution is correct"
8: **goto** 10
9: **output** "Solution is incorrect"
10: **stop**

---

proceeds. If the loop ends naturally, the control is passed to Step 7, which outputs a message saying that the solution is correct. After that, the control is passed to Step 10, which completes the execution of the algorithm.

2. Parallel Algorithm for Validating LP Solutions

According to Equation (5), the cardinality of the validation set generated by Algorithm 3 depends exponentially on the space dimension. Therefore, Algorithm 3 has high computational complexity for large dimensions. To reduce computational overheads, we developed a parallel version of Algorithm 3, given as Algorithm 4 below. The parallel algorithm is based on the BSF parallel computation model [22, 23], which exploits the master–slave paradigm [24]. According to the BSF model, the master node serves as a control and communication center. All slave nodes execute the same code but on different data. The BSF model assumes the algorithm representation in the form of operations on lists using the higher-order functions *Map* and *Reduce* defined by the Bird–Meertens formalism [25]. The higher-order function *Map* transforms the original list $W = [w_0, \ldots, w_{K-1}]$ into the list $Z = [z_0, \ldots, z_{K-1}]$ by applying the function $f_{\tilde{x}}$ to each element:

$$Z = Map(f_{\tilde{x}}, W) = [f_{\tilde{x}}(w_0), \ldots, f_{\tilde{x}}(w_{K-1})].$$

In the case considered here, the elements of the list W are the sequential numbers of the validation set points, that is,

$$W = [0, \ldots, K-1],$$

where $K = 2d(d-1)^{n-2}$. The Boolean function $f_{\tilde{x}} : \{0, \ldots, K-1\} \to \{true, false\}$ is defined as follows:

$$f_{\tilde{x}}(w) = \begin{cases} true \mid A \cdot g(w) \leqslant b \wedge \langle c, g(w) \rangle \leqslant \langle c, \tilde{x} \rangle; \\ false \mid A \cdot g(w) > b \vee \langle c, g(w) \rangle > \langle c, \tilde{x} \rangle, \end{cases}$$

where the vector-valued function g computes the coordinates of the validation point by its number w. The function $f_{\tilde{x}}$ returns *true* if the point $g(w)$ belongs to the feasible region and if the value of the objective function at this point is less than or equal to the value of the objective function at the point $\tilde{x}$. Otherwise, the function $f_{\tilde{x}}$ returns *false*. Thus, the list $Z = [z_0, \ldots, z_{K-1}]$ contains Boolean indicators for all points of the validation set. If at least one element in this list has the value *false*, then the point $\tilde{x}$ is an incorrect solution of problem (1).

Algorithm 4 Parallel algorithm for validating an LP solution

Master	**Slave ($l=0,\ldots,L\text{-}1$)**
1:	1: **input** $n, A, b, c, d, \rho, \varepsilon, \widetilde{x}$
2:	2: $L :=$ **NumberOfSlaves**
3:	3: $K := 2d(d-1)^{n-2}$
4:	4: $W_l := [lK/L, \ldots, (l+1)K/L - 1]$
5:	5: $Z_l := Map\,(f_{\widetilde{x}}, W_l)$
6:	6: $s_l := Reduce(\wedge, Z_l)$
7: **RecvFromSlaves** $[s_0, \ldots, s_{L-1}]$	7: **SendToMaster** s_l
8: $s := Reduce(\wedge, [s_0, \ldots, s_{L-1}])$	8:
9: **if** $s = true$ **then**	9:
10: **output** "Solution is correct"	10:
11: **else**	11:
12: **output** "Solution is incorrect"	12:
13: **end if**	13:
14: **stop**	14: **stop**

The higher-order function *Reduce* transforms the list $Z = [z_0, \ldots, z_{K-1}]$ into a single Boolean value s by iteratively applying the conjunction operation to all the elements of the list Z:

$$s = Reduce(\wedge, Z) = z_0 \wedge \ldots \wedge z_{K-1}.$$

In Step 4 of Algorithm 4, the l-th slave sets its own part W_l of the list W:

$$W_l = [lK/L, \ldots, (l+1)K/L - 1].$$

Here, L denotes the number of slaves. For simplicity, we assume that K is a multiple of L. In Step 5, the slave applies the *Map* function to its sublist W_l. In Step 6, the resulting sublist of Boolean values is folded into a single Boolean value s_l by applying the *Reduce* function, taking the conjunction operation as the first parameter. In Step 7, the l-th slave sends the value s_l to the master. In the same Step 7, the master receives all the calculated values from the slaves. In Step 8, the master folds the list of received values into a single Boolean value s using the *Reduce* function. In steps 9–12, the master examines the calculated Boolean value s and outputs the corresponding conclusion.

3. Software Implementation and Computational Experiments

We implemented the parallel Algorithm 4 in C++ through the parallel BSF-skeleton [26], which is based on the BSF parallel computation model [22, 23] and encapsulates all aspects related to the parallelization of the program using the MPI library [27]. The source code of the VaLiPro parallel program is freely available at `https://github.com/leonid-sokolinsky/BSF-LPP-Validator`. Using this program, we conducted large-scale computational experiments on the cluster computing system "Tornado SUSU" [28]. The specifications of this system are given in Tab. 1.

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

For experiments, we used random LP problems generated by the program FRaGenLP [29] with the following parameters: $\alpha = 200$ (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 experiments were conducted for the following dimensions: $n = 15$, $n = 17$, and $n = 19$. The numbers of inequalities were 46, 52, and 58, respectively. The solutions to the LP problems were obtained using the apex method [8]. Throughout the experiments, we used the following VaLiPro parameters: $d = 5$, $\rho = 1$, and $\varepsilon = 10^{-6}$. The results of the experiments are shown in Fig. 2. The verification of a solution for a problem of dimension $n = 19$ with a configuration consisting of the master node and one slave node took 17 minutes. The verification of a solution for the same problem with a configuration consisting of the master node and 310 slave nodes took 4 seconds. 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 $n = 19$, the parallel version of the VaLiPro algorithm demonstrated near-linear scalability up to 310 processor nodes. For $n = 17$, the scalability bound was approximately 260 nodes, and for $n = 15$, this bound decreased to 60 processor nodes. This is because a problem of such a small dimension is not able to load such a large number of processor nodes: the time spent on data transfer over the network begins to dominate over the time spent on calculations, and the processors begin to stand idle.

Conclusions

The article presents the parallel algorithm VaLiPro for validating linear programming solutions on cluster computing systems. The main idea of the validation algorithm is to generate a regular set of points on a small-radius hypersphere centered at the solution point that is to be certified. The solution is considered correct if all points of the validation set belonging to the feasible region have lower values of the objective function than does the solution point being certified. The implementation of the parallel algorithm VaLiPro was performed in C++ using the parallel BSF-skeleton, which encapsulates in the problem-independent part of its code all aspects related to the parallelization of a program using the MPI library. The source code of the developed parallel program is freely available at `https://github.com/leonid-sokolinsky/BSF-LPP-Validator`. The proposed validation method is generic and suitable for linear programming problems of any kind. The advantage of the parallel VaLiPro algorithm is the near-linear speedup starting with
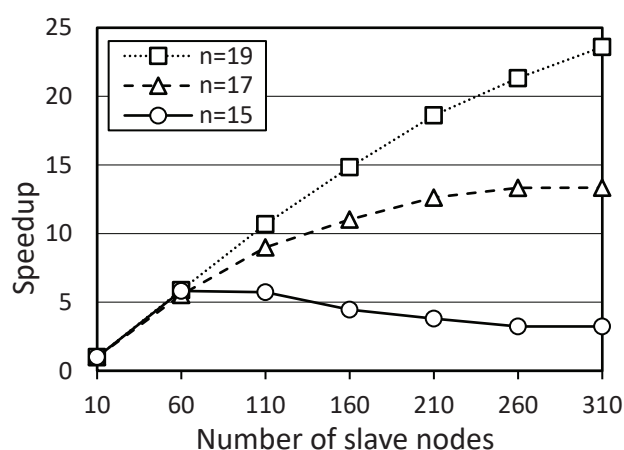
Figure 2. Speedup curves of the VaLiPro parallel algorithm for various dimensions

a problem dimension of 19. The main drawback that limits the practical use of the suggested method is the exponential growth of the number of points in the validation set as the dimension of the space increases; this results in the exponential growth of the computational complexity. In practice, the proposed algorithm can be effectively used for LP problems with a space dimension of no more than 20. The described algorithm was used together with the FRaGenLP generator and the apex method to prepare a training dataset of 70 000 examples, which will be used to develop an artificial neural network capable of solving multidimensional linear programming problems.

Acknowledgements

References

1. Jagadish, H.V., Gehrke, J., Labrinidis, A., et al.: Big data and its technical challenges. Communications of the ACM 57(7), 86–94 (2014). https://doi.org/10.1145/2611567

2. Hartung, T.: Making Big Sense grom Big Data. Frontiers in 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.) Parallel Computational Technologies. PCT 2017. Communications in Computer and Information Science, vol. 753. pp. 86–100. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67035-5_7

4. 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. Lecture Notes in Computer Science, vol. 9295. pp. 281–307. Springer, Cham (2015). `https://doi.org/10.1007/978-3-319-24024-4_17`

5. Huangfu, Q., Hall, J.A.J.: Parallelizing the dual revised simplex method. Mathematical Programming Computation 10(1), 119–142 (2018). `https://doi.org/10.1007/s12532-017-0130-5`

6. Tar, P., Stagel, B., Maros, I.: Parallel search paths for the simplex algorithm. Central European Journal of Operations Research 25(4), 967–984 (2017). `https://doi.org/10.1007/s10100-016-0452-9`

7. 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`

8. 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`

9. 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`

10. Gay, D.M.: Electronic mail distribution of linear programming test problems. Mathematical Programming Society COAL Bulletin (13), 10–12 (1985)

11. Dhiflaoui, M., Funke, S., Kwappik, C., et al.: Certifying and Repairing Solutions to Large LPs How Good are LP-solvers? In: SODA'03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 255–256. Society for Industrial and Applied Mathematics, USA (2003)

12. Rose, D.J., Tarjan, R.E.: Algorithmic Aspects of Vertex Elimination on Directed Graphs. SIAM Journal on Applied Mathematics 34(1), 176–197 (1978). `https://doi.org/10.1137/0134014`

13. Bareiss, E.H.: Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination. Mathematics of Computation 22(103), 565 (1968). `https://doi.org/10.2307/2004533`

14. Middeke, J., Jeffrey, D.J., Koutschan, C.: Common Factors in Fraction-Free Matrix Decompositions. Mathematics in Computer Science 1–20 (2020). `https://doi.org/10.1007/s11786-020-00495-9`

15. Wiedemann, D.H.: Solving Sparse Linear Equations over Finite Fields. IEEE Transactions on Information Theory 32(1), 54–62 (1986). `https://doi.org/10.1109/TIT.1986.1057137`

16. Koch, T.: The final NETLIB-LP results. Operations Research Letters 32(2), 138–142 (2004). `https://doi.org/10.1016/S0167-6377(03)00094-4`

17. Applegate, D.L., Cook, W., Dash, S., Espinoza, D.G.: Exact solutions to linear programming problems. Operations Research Letters 35(6), 693–699 (2007). `https://doi.org/10.1016/j.orl.2006.12.010`

18. The GNU Multiple Precision Arithmetic Library. `https://gmplib.org`

19. Panyukov, A.V., Gorbik, V.V.: Using massively parallel computations for absolutely precise solution of the linear programming problems. Automation and Remote Control 73(2), 276–290 (2012). `https://doi.org/10.1134/S0005117912020063`

20. Gleixner, A.M., Steffy, D.E., Wolter, K.: Iterative refinement for linear programming. INFORMS Journal on Computing 28(3), 449–464 (2016). `https://doi.org/10.1287/ijoc.2016.0692`

21. Blumenson, L.E.: A Derivation of n-Dimensional Spherical Coordinates. The American Mathematical Monthly 67(1), 63–66 (1960). `https://doi.org/10.2307/2308932`

22. Sokolinsky, L.B.: BSF: A parallel computation model for scalability estimation of iterative numerical algorithms on cluster computing systems. Journal of Parallel and Distributed Computing 149, 193–206 (2021). `https://doi.org/10.1016/j.jpdc.2020.12.009`

23. Sokolinsky, L.B.: Analytical Estimation of the Scalability of Iterative Numerical Algorithms on Distributed Memory Multiprocessors. Lobachevskii Journal of Mathematics 39(4), 571–575 (2018). `https://doi.org/10.1134/S1995080218040121`

24. Sahni, S., Vairaktarakis, G.: The master-slave paradigm in parallel computer and industrial settings. Journal of Global Optimization 9(3-4), 357–377 (1996). `https://doi.org/10.1007/BF00121679`

25. 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, Berlin, Heidelberg (1988)

26. Sokolinsky, L.B.: BSF-skeleton: A Template for Parallelization of Iterative Numerical Algorithms on Cluster Computing Systems. MethodsX 8. Art. no. 101437 (2021). `https://doi.org/10.1016/j.mex.2021.101437`

27. Gropp, W.: MPI 3 and Beyond: Why MPI is Successful and What Challenges It Faces. In: Traff, J.L., Benkner, S., Dongarra, J.J. (eds.) Recent Advances in the Message Passing Interface, EuroMPI 2012. Lecture Notes in Computer Science, vol. 7490. pp. 1–9. Springer, Berlin, Heidelberg (2012). `https://doi.org/10.1007/978-3-642-33518-1_1`

28. 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`

29. Sokolinsky, L.B., Sokolinskaya, I.M.: FRaGenLP: A Generator of Random Linear Programming Problems for Cluster Computing Systems. In: Sokolinsky L., Zymbler M. (eds) Parallel Computational Technologies, PCT 2021. Communications in Computer and Information Science, vol 1437. pp. 164–177. Springer, Cham (2021). `https://doi.org/10.1007/978-3-030-81691-9_12`