VaLiPro: валидатор решений задач линейного программирования для кластерных вычислительных систем*

Л.Б. Соколинский, И.М. Соколинская

Южно-Уральский государственный университет

В статье представлен параллельный алгоритм валидации решений задач линейного программирования. Идея метода состоит в том, чтобы генерировать регулярный набор точек на гиперсфере малого радиуса, центрированной в точке тестируемого решения. Целевая функция вычисляется для каждой точки валидационного множества, принадлежащей допустимой области. Если все полученные значения меньше или равны значению целевой функции в точке, проверяемой как решение, то эта точка считается корректным решением. Параллельная реализация алгоритма VaLiPro выполнена на языке C++ с использованием параллельного BSF-каркаса, инкапсулирующего в проблемно-независимой части своего кода все аспекты, связанные с распараллеливанием программы на базе библиотеки MPI. Приводятся результаты масштабных вычислительных экспериментов на кластерной вычислительной системе, подтверждающие эффективность предложенного подхода.

Ключевые слова: линейное программирование, валидатор решений, VaLiPro, параллельный алгоритм, кластерные вычислительные системы, параллельный BSF-каркас.

1. Введение

Эра больших данных [1,2] привела к появлению задач линейного программирования (ЛП) сверхбольших размерностей [3]. Подобные задачи возникают в экономике, индустрии, логистике, статистике, квантовой физике и других областях. Решение таких сверхбольших задач невозможно без масштабируемых параллельных алгоритмов, ориентированных на кластерные вычислительные системы. В соответствии с этим в последние годы интенсифицировались усилия по разработке новых и модернизации известных параллельных алгоритмов решения задач ЛП. В качестве примеров можно привести работы [4–8]. При разработке новых масштабируемых алгоритмов ЛП возникает необходимость их тестирования на различных задачах. Источниками таких задач могут быть как эталонные репозитории, например Netlib-Lp [9], так и генераторы случайных задач, см. например [10,11], в которых решение заранее известно. При этом на практике часто встречаются классы задач с неизвестными решениями. При тестировании ЛП-решателя на таких классах задач возникает необходимость валидации, сертификации и уточнения полученного решения. Проблеме сертификации и уточнения решения задачи ЛП посвящен ряд работ. В статье [12] был предложен метод, проверяющий вычисленное решение на его принадлежность допустимой области и оптимальность на основе рациональной арифметики. Указанный подход впоследствии был реализован с более высокой эффективностью Кохом (Koch) в [13] и Апплгейтом (Applegate) с соавторами в [14]. Кох разработал программу верификации *perPlex*, базирующуюся на симплекс-методе и использующую разреженную LU-декомпозицию с выбором ведущего элемента по Марковицу (Markowitz pivoting). Апплгейт с соавторами разработали программу *QSopt_ex* на основе известной библиотеки GMP (GNU Multi Precision), позволяющей выполнять вычисления с произвольной точностью. Программа стартует, начиная с точности, поддерживаемой целевой вычислительной системой, затем происходит переход к 128 разрядам, и затем разрядность динамически увеличивается на 50%, пока не будет получено решение, удовлетворяющее всем ограничениям с требуемой точностью. Программа также может использоваться для проверки задачи ЛП на неразрешимость или неограниченность. Основным недостатком программы QSopt_ex является то, что использование

* Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 20-07-00092-а.

дробно-рациональных вычислений в случае больших и сложных задач ЛП требует больших временных затрат. Панюков и Горбик в работе [15] попытались преодолеть этот недостаток путем использования параллельных вычислений на распределенной памяти. Ими предложены два метода распараллеливания симплекс-метода. Первый метод основан на декомпозиции симплекс-таблицы по столбцам. Второй метод базируется на методе обратной матрицы и использует декомпозицию исходной матрицы по столбцам, а обратной – по строкам. Однако результаты проведенных экспериментов недостаточно убедительны, так как отсутствует сравнение с лучшими последовательными решениями, вычисления проводились только с использованием трех разреженных задач ЛП (число ненулевых элементов не превышало 5%), и, кроме этого, максимальная граница масштабирования ограничивалась 16 процессорами. Еще одно оригинальное решение предложено Глейкснером (Gleixner) и соавторами в [16]. В этой работе описывается процедура итеративного уточнения решения задачи ЛП с любой заданной точностью. Точное решение вычисляется путем решения последовательности связанных задач ЛП с использованием арифметики ограниченной точности. Решаемые задачи ЛП имеют ту же матрицу коэффициентов, что и исходная задача, преобразуются только целевая функция, правые части ограничений и границы переменных.

Все рассмотренные выше методы концентрируются на уточнении уже найденного приближенного решения. Если же найденное решение находится слишком далеко от правильного, что означает наличие ошибки в алгоритме, то применение этих методов становится нецелесообразным. Кроме того, все указанные алгоритмы имеют высокую вычислительную сложность и при этом не допускают эффективного распараллеливания на больших кластерных вычислительных системах. Метод, предлагаемый в данной статье, ориентирован на отладку и валидацию новых алгоритмов решения задач ЛП на кластерных вычислительных системах. Он реализован в виде параллельной программы *VaLiPro* (*Validator of Linear Program*), обладающей хорошей масштабируемостью на многопроцессорных вычислительных системах с распределенной памятью. Статья организована следующим образом. В разделе 2 дается формальное описание предлагаемого метода валидации решений задач ЛП и приводится последовательная версия алгоритма VaLiPro. В разделе 3 рассматривается параллельная версия алгоритма VaLiPro. В разделе 4 предлагается ее реализация с использованием параллельного BSF-каркаса и приводятся результаты масштабных вычислительных экспериментов на кластерной вычислительной системе, подтверждающие эффективность предложенного подхода. В разделе 5 суммируются полученные результаты и приводятся планы по использованию валидатора VaLiPro для разработки искусственной нейронной сети, способной решать задачи ЛП большой размерности.

2. Метод валидации решений задач ЛП

Пусть в евклидовом пространстве $\mathbf{R}^n$ задана задача линейного программирования

$$\bar{x} = \arg\max\left\{\langle c, x\rangle \middle| Ax \le b, x \in \mathbf{R}^n\right\}, \tag{1}$$

где c – вектор коэффициентов целевой функции. Здесь и далее $\langle \square, \square \rangle$ обозначает скалярное произведение двух векторов. Обозначим $M = \left\{x \in \square^n \middle| Ax \le b\right\}$ – множество допустимых точек задачи (1). По определению множество M является выпуклым. Везде далее мы будем предполагать, что M является непустым, ограниченным (и, следовательно, замкнутым) множеством, то есть задача (1) имеет, по крайней мере, одно решение. Пусть $\tilde{x} \in \mathbf{R}^n$ – приближенное решение задачи (1), полученное с помощью некоторого ЛП-решателя, и требующее сертификации.

Идея метода валидации VaLiPro, описываемого в данной работе, заключается в построении конечного множества точек V, покрывающих гиперсферу S малого радиуса ρ с центром в точке сертифицируемого решения $\tilde{x}$:

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

Здесь и далее $\|\cdot\|$ обозначает евклидову норму. На множестве точек $V \cap M$ вычисляется максимум целевой функции:

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

Если $|\langle c, \bar{v}\rangle - \langle c, \tilde{x}\rangle| < \varepsilon$, то приближенное решение $\tilde{x}$ считается корректным. В противном случае $\tilde{x}$ считается некорректным решением. Здесь $\varepsilon \in \mathbf{R}_{>0}$ – некоторая малая положительная константа, являющаяся параметром алгоритма валидации.

Опишем метод построения валидационного множества V. Известно [17], что координаты любой точки $v = (v_1, \ldots, v_n)$, лежащей на поверхности гиперсферы S, задаваемой уравнением

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

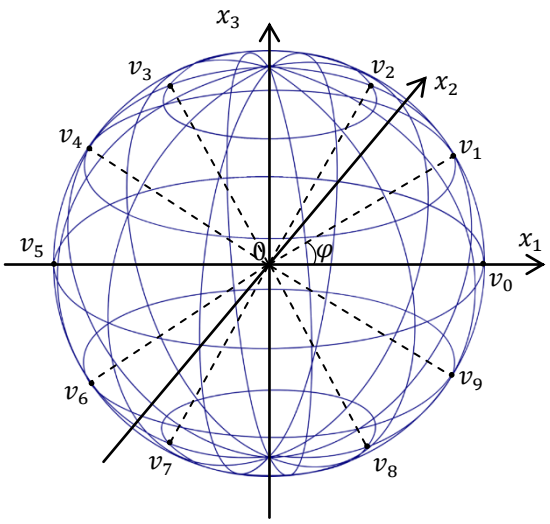

Рис. 1. Построение точек валидационного множества V на трехмерной сфере ($n = 3, d = 5$).

могут быть представлены в виде

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

где $0 \le \phi_j \le \pi$ $(j = 1,\ldots,n-2)$, $0 \le \theta < 2\pi$. Алгоритм генерации валидационного множества V проиллюстрируем на трехмерной сфере (см. рис. 1). Зафиксируем *нечетное число параллелей* $d \ge 3$ (полюса исключаются). Положим

$$\varphi = \pi/d. \tag{3}$$

В плоскости $(x_1, 0, x_3)$ отложим от оси $(0, x_1)$ углы $0, \varphi, \ldots, (d-1)\varphi$. Получившиеся лучи в пересечении со сферой дадут множество точек $\{v_0, \ldots, v_{d-1}\}$, которые однозначно определяют d параллелей. Теперь в плоскости $(x_1, 0, x_2)$ аналогичным образом отложим от оси $(0, x_1)$ углы $0, \varphi, \ldots, (d-1)\varphi$ и определим d *меридианов*. Точки, получающиеся на пересечении параллелей и меридианов, за исключением точек полюсов, образуют валидационное множество точек на трехмерной сфере. В общем виде описанный метод генерации точек валидационного множества для $n \ge 3$ представлен в виде алгоритма 1а. Вложенные циклы с заголовками на шагах 3, 5, 8, 10 фактически генерируют сферические координаты очередной валидационной точки:

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

На шагах 12-19 сферические координаты преобразуются в декартовы по формулам (2). Используя границы переменных в заголовках циклов на шагах 3, 5, 8, 10, легко подсчитать, что алгоритм 1а порождает $2d(d+1)^{n-2}$ точек. Недостатком алгоритма 1а является то, что он генерирует некоторые точки с повторениями. Проведенный вычислительный эксперимент показал, что для размерности $n = 4$ при количестве параллелей $d = 5$ алгоритм 1а порождает 189 дубликатов

при

Алгоритм 1. Генерация точек валидационного множества

а) С дубликатами	б) Без дубликатов
1: **input** d	1: **input** d, ρ
2: $\varphi \coloneqq \pi/d$	2: $\varphi \coloneqq \pi/d$
3: **for** $j_{n-1} = 0 \dots (2d-1)$ **do begin**	3: **for** $j_{n-1} = 0 \dots (2d-1)$ **do begin**
4: $\theta \coloneqq j_{n-1}\varphi$	4: $\theta \coloneqq j_{n-1}\varphi$
5: **for** $j_{n-2} = 0 \dots d$ **do begin**	5: **for** $j_{n-2} = 1 \dots (d-1)$ **do begin**
6: $\phi_{n-2} \coloneqq j_{(n-2)}\varphi$	6: $\phi_{n-2} \coloneqq j_{n-2}\varphi$
7: ...	7: ...
8: **for** $j_2 = 0 \dots d$ **do begin**	8: **for** $j_2 = 1 \dots (d-1)$ **do begin**
9: $\phi_2 \coloneqq j_2\varphi$	9: $\phi_2 \coloneqq j_2\varphi$
10: **for** $j_1 = 0 \dots d$ **do begin**	10: **for** $j_1 = 1 \dots (d-1)$ **do begin**
11: $\phi_1 \coloneqq j_1\varphi$	11: $\phi_1 \coloneqq j_1\varphi$
12: $\Pi \coloneqq 1$	12: $\Pi \coloneqq 1$
13: $v_1 \coloneqq \rho\, cos(\phi_1)$	13: $v_1 \coloneqq \rho\, cos(\phi_1)$
14: **for** $l = 2 \dots n-2$ **do begin**	14: **for** $l = 2 \dots n-2$ **do begin**
15: $\Pi \coloneqq \sin(\phi_{l-1})\Pi$	15: $\Pi \coloneqq \sin(\phi_{l-1})\Pi$
16: $v_l \coloneqq \rho\cos(\phi_l)\Pi$	16: $v_l \coloneqq \rho\cos(\phi_l)\Pi$
17: **end**	17: **end**
18: $v_{n-1} \coloneqq \rho\sin(\theta)\Pi$	18: $v_{n-1} \coloneqq \rho\sin(\theta)\Pi$
19: $v_n \coloneqq \rho\cos(\theta)\Pi$	19: $v_n \coloneqq \rho\cos(\theta)\Pi$
20: **output** v	20: **output** v
21: **end**	21: **end**
22: **end**	22: **end**
23: ...	23: ...
24: **end**	24: **end**
25: **end**	25: **end**
26: **stop**	26: **stop**

общем количестве точек равном 360, что составляет более 50%. Дубликаты порождаются в итерациях, когда $\phi_i = 0$ или $\phi_i = \pi$, что соответствует значениям $j_i = 0$ и $j_i = d$ $(i = 1,\dots,n-2)$. Это вызвано тем, что в этом случае один из сомножителей $\sin\phi_i$ в (2) окажется равным нулю и, следовательно, вариации других сомножителей уже не смогут изменить значение соответствующей координаты. Решение проблемы дубликатов без серьезной переделки алгоритма 1а достигается путем задания границ переменных циклов $j_1,\dots,j_{n-2}$ от 1 до $d-1$, как это сделано в алгоритме 1б (шаги 5, 8 и 10). При этом вместе с дубликатами теряется и некоторое количество значимых точек. При $n = 4$ и $d = 5$ это количество равно 11, что составляет менее 7% от полного набора после удаления дубликатов. Эксперименты показали, что такая потеря не оказывает существенного влияния на качество работы валидационного алгоритма. Анализ алгоритма 1б показывает, что общее количество точек валидационного множества V, генерируемых этим алгоритмом, определяется по формуле

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

Недостатком алгоритма 1б является то, что количество циклов зависит от размерности задачи, что не позволяет использовать размерность как параметр программы. Кроме того, этот подход оказывается неприемлемым для больших размерностей, так как, например, для размерности $n = 1000$ нам пришлось бы написать тысячу циклов for. Для преодоления указанного недостатка мы использовали вектор-функцию $g : \{0,\dots,2d(d-1)^{n-2}-1\} \to \mathbb{R}^n$, вычисляющую координаты точки валидационного множества по ее номеру k (нумерация начинается с нуля) в том поряд-

ке, в котором их генерирует алгоритм 1б. Определение этой функции представлена в виде алгоритма 2.

Алгоритм 2. Функция g (вычисление точки v по ее номеру k)

```
 1: function g(k) begin
 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 begin
 6:         u_j := ⌊k/(d-1)^j⌋ + 1
 7:         k := k mod (d-1)^j
 8:     end
 9:     Π := 1
10:     φ := π/d
11:     v_1 := ρ cos(u_1 φ)
12:     for j = 2 ... (n-2) do begin
13:         Π = Π sin(u_{j-1} φ)
14:     v_j = ρ cos(u_j φ) Π
15:     end
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
20: end
```

Алгоритм 3. Валидация решения $\tilde{x}$ задачи ЛП

```
 1: input n, A, b, c, d, ρ, ε, x̃
 2:     φ := π/d
 3:     for k = 0 ... 2d(d-1)^{n-2} - 1 do begin
 4:         v := g(k)
 5:         if Av ≤ b ∧ ⟨c, v⟩ > ⟨c, x̃⟩ + ε goto 9
 6:     end
 7:     output "Solution" x̃ "is correct."
 8:     goto 10
 9:     output "Solution" x̃ "is incorrect."
10: stop
```

Финальная реализация метода VaLiPro, использующая вектор-функцию g, представлена в виде алгоритма 3. В качестве дополнительного параметра этого алгоритма фигурирует малая положительная константа ε (по умолчанию $\varepsilon = 10^{-6}$), нивелирующая возможные погрешности вычислений при сравнении значений целевой функции на шаге 5. Кратко прокомментируем шаги алгоритма 3. На шаге 1 вводятся исходные данные задачи ЛП (1), параметры алгоритма и сертифицируемое решение $\tilde{x}$. На шаге 2 вычисляется угол φ в соответствии с формулой (3). Шаг 3 открывает цикл по номеру точки k, изменяющемуся от 0 до $2d(d-1)^{n-2}-1$ в соответствии с формулой (5). На шаге 4 с помощью вектор-функции g (см. алгоритм 2) вычисляется очередная валидационная точка v. На шаге 5 проверяется, принадлежит ли v допустимой области задачи (1), и сравниваются значения целевой функции в точке v и в точке сертифицируемого решения $\tilde{x}$. Если целевая функция принимает большее значение в точке v и эта точка является допустимой, то осуществляется переход на шаг 9, где выводится сообщение, что сертифицируемое решение не является корректным. В противном случае осуществляется переход к следующей итерации цикла. Если цикл завершился естественным образом, управление переходит на шаг 7, где выводится сообщение, что решение $\tilde{x}$ является корректным, после чего осуществляется переход на шаг 10, где алгоритм завершает свою работу.

3. Параллельный алгоритм валидации решений задач ЛП

В соответствии с формулой (5) мощность валидационного множества точек, генерируемого алгоритмом 3, экспоненциально зависит от размерности пространства. Поэтому алгоритм 3 будет иметь высокую вычислительную сложность для больших размерностей. Для сокращения временных затрат на вычисления нами была разработана и реализована параллельная версия алгоритма 3, представленная в виде алгоритма 4. Данный алгоритм разработан в соответствии с моделью параллельных вычислений BSF [18], использующей парадигму «мастер-рабочий» [19]. В соответствии с этой моделью узел-мастер является центром управления и коммуникации. Все узлы-рабочие выполняют один и тот же код, но над разными данными. BSF-модель

требует представления алгоритма в виде операций над списками с использованием функций высшего порядка *Map* и *Reduce*, определяемых формализмом Бирда-Миртенса (Bird–Meertens formalism) [20].

Алгоритм 4. Параллельный алгоритм валидации решения задачи ЛП

Мастер	l-й рабочий (l=0,…,L-1)
M1:	W1: **input** n, A, b, c, d, ρ, ε, $\tilde{x}$
M2:	W2: $W_l := [lK/L, \dots, (l+1)K/L - 1]$
M3:	W3: $Z_l := Map(f_{\tilde{x}}, W_l)$
M4:	W4: $s_l := Reduce(\wedge, Z_l)$
M5: RecvFromWorkers($s_1, \dots, s_L$)	W5: SendToMaster(s_l)
M6: $s := Reduce(\wedge, [s_1, \dots, s_L])$	W6:
M7: **if** $s = false$ **goto** M10	W7:
M8: **output** "Solution" $\tilde{x}$ "is correct."	W8:
M9: **goto** M11	W9:
M10: **output** "Solution" $\tilde{x}$ "is incorrect."	W10:
M11: **stop**	W11: **stop**

Функция высшего порядка *Map* преобразует исходный список $W = [w_0, \dots, w_{K-1}]$ в список $Z = [z_0, \dots, z_{K-1}]$, применяя к каждому элементу функцию $f_{\tilde{x}}$:

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

В данном случае элементами списка W являются номера точек валидационного множества:

$$W = [0, \dots, K-1] \quad \left(K = 2d(d-1)^{n-2}\right).$$

Булева функция $f_{\tilde{x}} : \{0, \dots, K-1\} \to \{true, false\}$ задается следующей формулой:

$$f_{\tilde{x}}(w) = \begin{cases} true & \left| A \cdot g(w) \le b \wedge \langle c, g(w) \rangle \le \langle c, \tilde{x} \rangle, \right. \\ false & \left| A \cdot g(w) > b \vee \langle c, g(w) \rangle > \langle c, \tilde{x} \rangle, \right. \end{cases}$$

где вектор-функция g вычисляет координаты валидационной точки по ее номеру w. Функция $f_{\tilde{x}}$ возвращает значение «истина» (*true*), если точка $g(w)$ принадлежит допустимой области задачи (1), и значение целевой функции в этой точке меньше, либо равно значению целевой функции в точке $\tilde{x}$. В противном случае функция $f_{\tilde{x}}$ возвращает значение «ложь» (*false*). Список $Z = [z_0, \dots, z_{K-1}]$, таким образом, содержит булевы индикаторы для всех точек валидационного множества. Если хотя бы один элемент в этом списке имеет значение *false*, то точка $\tilde{x}$ считается некорректным решением задачи (1).

Функция высшего порядка *Reduce* преобразует список $Z = [z_0, \dots, z_{K-1}]$ в атомарное булево значение s, применяя операцию конъюнкции $\wedge$ ко всем элементам списка Z:

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

В параллельном алгоритме 4 на шаге W2 l-тый рабочий определяет свою часть W_l списка W:

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

Здесь L обозначает количество рабочих. Для простоты мы предполагаем, что K кратно L. На шаге W3 рабочий применяет функцию *Map* к своей части списка. Получившийся список булевых значений на шаге W4 редуцируется к одному булеву значению s_l путем применения функции *Reduce*, первым параметром которой является операция логического «и». Вычисленное таким образом логическое значение s_l на шаге W5 пересылается мастеру. Мастер на шаге M5 получает от рабочих все вычисленные значения. На шаге M6 список полученных значений ре-

дуцируется к одному булеву значению s с помощью функции *Reduce*. На шагах M7-M10 анализируется вычисленное булево значение s и выводится соответствующее заключение.

Табл. 1. Характеристики кластера «Торнадо ЮУрГУ».

Количество процессорных узлов	480
Процессоры	Intel Xeon X5680 (6 cores 3.33 GHz)
Количество процессоров в узле	2
Оперативная память узла	24 GB DDR3
Соединительная сеть	InfniBand QDR (40 Gbit/s)
Операционная система	Linux CentOS

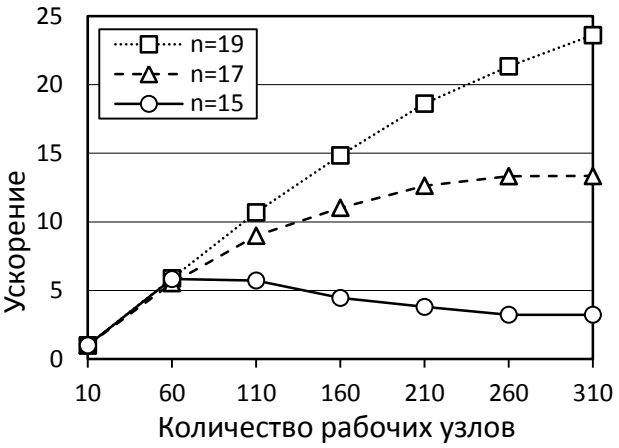

Рис. 2. Ускорение параллельного алгоритма ValiPro для различных размерностей.

4. Программная реализация и вычислительные эксперименты

Параллельный алгоритм 4 был нами реализован на языке C++ с использованием параллельного BSF-каркаса [21,22]. BSF-каркас базируется на модели параллельных вычислений BSF [18] и инкапсулирует в проблемно независимой части своего кода все аспекты, связанные с распараллеливанием программы с помощью библиотеки MPI [23]. Исходные коды параллельной программы ValiPro свободно доступны в сети Интернет по адресу *https://github.com/leonid-sokolinsky/BSF-LPP-Validator*. С использованием указанной программы нами были проведены масштабные вычислительные эксперименты на вычислительном кластере «Торнадо ЮУрГУ» [24], характеристики которого приведены в табл. 1. Для экспериментов использовались случайные задачи ЛП, полученные с помощью программы FRaGenLP [25] при следующих значениях параметров: длина ребра ограничивающего гиперкуба $\alpha = 200$, радиус большой гиперсферы $\theta = 100$, радиус малой гиперсферы $\rho = 50$, верхняя граница «почти параллельности» для гиперплоскостей $L_{max} = 0.35$, минимальная допустимая близость для гиперплоскостей $S_{min} = 100$, максимальное допустимое абсолютное значение при генерации коэффициентов случайных неравенств $a_{max} = 1000$, максимальное допустимое абсолютное значение при генерации правых частей случайных неравенств $b_{max} = 10000$. Эксперименты проводились для размерностей $n = 15$, $n = 17$ и $n = 19$. Количество неравенств, соответственно, составляло 46, 52 и 58. Из них случайных: 15, 17 и 19 соответственно. Решения задач получались с помощью апекс-метода [4]. Во всех случаях при запуске валидатора использовались следующие значения параметров: $d = 5$, $\rho = 1$, $\varepsilon = 10^{-6}$. Результаты экспериментов представлены на рис. 2. Время верификации решения для случайной задачи ЛП размерности $n = 19$ на конфигурации из одного узла-мастера и одного узла-рабочего составило 17 минут. На конфигурации из одного узла-мастера и 310 узлов-рабочих верификация решения такой же задачи заняла 4 секунды. Анализ результатов показывает, что граница масштабируемости предложенного алгоритма существенно зависит от размерности задачи (под границей масштабируемости здесь понимается максимум кривой ускорения). При $n = 19$ параллельная версия алгоритма ValiPro продемонстрировала масштабируемость, близкую к линейной, вплоть до 310 процессорных узлов. Для задачи размерности $n = 17$ граница масштабируемости составила примерно 260 узлов, а на задаче размерности $n = 15$ она уменьшилась до 60 узлов. Это связано с тем, что задача такой размерности

уже не способна загрузить работой большое количество процессорных узлов: время, затрачиваемое на передачу данных по сети, начинает превалировать над временем, затрачиваемым на вычисления, и процессоры начинают простаивать.

5. Заключение

В статье представлен параллельный алгоритм VaLiPro для валидации решений задач линейного программирования на кластерных вычислительных системах. Идея валидационного алгоритма заключается в генерации регулярного множества точек на гиперсфере малого радиуса с центром в проверяемом решении. Решение считается корректным, если все точки валидационного множества, принадлежащие допустимой области задачи линейного программирования, характеризуются меньшим значением целевой функции по сравнению с проверяемой точкой. Реализация параллельного алгоритма VaLiPro была выполнена на языке C++ с использованием параллельного BSF-каркаса, инкапсулирующего в проблемно независимой части своего кода все аспекты, связанные с распараллеливанием программы с помощью библиотеки MPI. Исходные коды разработанной параллельной программы свободно доступны в сети Интернет по адресу *https://github.com/leonid-sokolinsky/BSF-LPP-Validator*. Предложенный метод валидации является универсальным и годится для любых задач линейного программирования. Преимуществом параллельного алгоритма VaLiPro является почти линейное ускорение для достаточно больших размерностей пространства, начиная с размерности 19. Существенным недостатком, ограничивающим применение предложенного метода, является экспоненциальный рост количества точек валидационного множества при увеличении размерности пространства, что приводит к экспоненциальному росту временной сложности алгоритма. Описанный алгоритм был использован вместе с алгоритмом FRaGenLP [25] и апекс-методом [4] для автоматической генерации обучающего набора из 70 000 прецедентов, который планируется использовать для разработки искусственной нейронной сети, способной решать многомерные задачи линейного программирования.

Литература

1. Jagadish H. V. et al. Big data and its technical challenges // Communications of the ACM. 2014. Vol. 57, no. 7. P. 86–94. DOI:10.1145/2611567.
2. Hartung T. Making Big Sense From Big Data // Frontiers in Big Data. 2018. Vol. 1. P. 5. DOI:10.3389/fdata.2018.00005.
3. Соколинская И.М., Соколинский Л.Б. О решении задачи линейного программирования в эпоху больших данных // Параллельные вычислительные технологии (ПаВТ’2017). Короткие статьи и описания плакатов. Челябинск: Издательский центр ЮУрГУ, 2017. С. 471–484.
4. Соколинский Л.Б., Соколинская И.М. Исследование масштабируемости апекс-метода для решения сверхбольших задач линейного программирования на кластерных вычислительных системах // Суперкомпьютерные дни в России: Труды международной конференции. 21-22 сентября 2020 г. Москва: МАКС Пресс, 2020. С. 49–59.
5. Mamalis B., Pantziou G. Advances in the Parallelization of the Simplex Method // Algorithms, Probability, Networks, and Games. Lecture Notes in Computer Science, vol. 9295 / ed. Zaroliagis C., Pantziou G., Kontogiannis S. Cham: Springer, 2015. P. 281–307. DOI:10.1007/978-3-319-24024-4_17.
6. Huangfu Q., Hall J.A.J. Parallelizing the dual revised simplex method // Mathematical Programming Computation. Springer Verlag, 2018. Vol. 10, no. 1. P. 119–142. DOI:10.1007/s12532-017-0130-5.
7. Tar P., Stágel B., Maros I. Parallel search paths for the simplex algorithm // Central European Journal of Operations Research. Springer Verlag, 2017. Vol. 25, no. 4. P. 967–984. DOI:10.1007/s10100-016-0452-9.

8. Yang L., Li T., Li J. Parallel predictor-corrector interior-point algorithm of structured optimization problems // 3rd International Conference on Genetic and Evolutionary Computing, WGEC 2009. 2009. P. 256–259. DOI:10.1109/WGEC.2009.68.
9. Gay D.M. Electronic mail distribution of linear programming test problems // Mathematical Programming Society COAL Bulletin. 1985. no. 13. P. 10–12.
10. Charnes A. et al. On Generation of Test Problems for Linear Programming Codes // Communications of the ACM. 1974. Vol. 17, no. 10. P. 583–586. DOI:10.1145/355620.361173.
11. Arthur J.L., Frendewey J.O. GENGUB: A generator for linear programs with generalized upper bound constraints // Computers and Operations Research. Pergamon, 1993. Vol. 20, no. 6. P. 565–573. DOI:10.1016/0305-0548(93)90112-V.
12. Dhiflaoui M. et al. Certifying and Repairing Solutions to Large LPs How Good are LP-solvers? // SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms. USA: Society for Industrial and Applied Mathematics, 2003. P. 255–256.
13. Koch T. The final NETLIB-LP results // Operations Research Letters. 2004. Vol. 32, no. 2. P. 138–142. DOI:10.1016/S0167-6377(03)00094-4.
14. Applegate D.L. et al. Exact solutions to linear programming problems // Operations Research Letters. North-Holland, 2007. Vol. 35, no. 6. P. 693–699. DOI:10.1016/j.orl.2006.12.010.
15. Панюков А.В., Горбик В.В. Применение массивно-параллельных вычислений для решения задач линейного программирования с абсолютной точностью // Автоматика и телемеханика. 2012. № 2. С. 73–88.
16. Gleixner A.M., Steffy D.E., Wolter K. Iterative refinement for linear programming // INFORMS Journal on Computing. INFORMS Inst.for Operations Res.and the Management Sciences, 2016. Vol. 28, no. 3. P. 449–464. DOI:10.1287/ijoc.2016.0692.
17. Blumenson L.E. A Derivation of n-Dimensional Spherical Coordinates // The American Mathematical Monthly. 1960. Vol. 67, no. 1. P. 63–66. DOI:10.2307/2308932.
18. 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. 2021. Vol. 149. P. 193–206. DOI:10.1016/j.jpdc.2020.12.009.
19. Sahni S., Vairaktarakis G. The master-slave paradigm in parallel computer and industrial settings // Journal of Global Optimization. 1996. Vol. 9, no. 3–4. P. 357–377. DOI:10.1007/BF00121679.
20. Bird R.S. Lectures on Constructive Functional Programming // Constructive Methods in Computing Science. NATO ASI Series F: Computer and Systems Sciences, vol. 55 / ed. Broy M. Berlin, Heidlberg: Springer, 1988. P. 151–216.
21. Соколинский Л.Б. Параллельный программный каркас BSF. Свидетельство о государственной регистрации программы для ЭВМ 2020661344, 22.09.2020.
22. Sokolinsky L.B. BSF-skeleton [Electronic resource]. 2019. URL: https://github.com/leonid-sokolinsky/BSF-skeleton (accessed: 09.04.2021).
23. Gropp W. MPI 3 and Beyond: Why MPI Is Successful and What Challenges It Faces // Recent Advances in the Message Passing Interface. EuroMPI 2012. Lecture Notes in Computer Science, vol. 7490 / ed. Träff J.L., Benkner S., Dongarra J.J. Berlin, Heidelberg: Springer, 2012. P. 1–9. DOI:10.1007/978-3-642-33518-1_1.
24. Kostenetskiy P.S., Safonov A.Y. SUSU Supercomputer Resources // Proceedings of the 10th Annual International Scientific Conference on Parallel Computing Technologies (PCT 2016). CEUR Workshop Proceedings. Vol. 1576. 2016. P. 561–573.
25. Соколинский Л.Б., Соколинская И.М. FRaGenLP: генератор случайных задач линейного программирования для кластерных вычислительных систем // Параллельные вычислительные технологии (ПаВТ'2021). Короткие статьи и описания плакатов. Челябинск: Издательский центр ЮУрГУ, 2021. С. 244–254. DOI:10.14529/pct2021.