2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

doi 10.26089/NumMet.v27r216 УДК 519.6

Параллельный алгоритм поиска регуляризованного решения плохо обусловленных СЛАУ с адаптивным учетом ошибок машинного округления

В. Д. Шинкарев
Московский государственный университет имени М. В. Ломоносова,
физический факультет, кафедра математики, Москва, Российская Федерация
ORCID: 0000-0001-8365-7224, e-mail: shinkarev.vd17@physics.msu.ru

А. М. Златковский
Московский государственный университет имени М. В. Ломоносова,
физический факультет, кафедра математики, Москва, Российская Федерация
ORCID: 0009-0001-8733-9650, e-mail: zlatkovskii.am20@physics.msu.ru

Д. В. Лукьяненко
Московский государственный университет имени М. В. Ломоносова,
физический факультет, кафедра математики, Москва, Российская Федерация
Московский государственный университет имени М. В. Ломоносова,
Научно-исследовательский вычислительный центр, Москва, Российская Федерация
Московский Центр фундаментальной и прикладной математики, Москва, Российская Федерация
ORCID: 0000-0001-5140-3617, e-mail: lukyanenko@physics.msu.ru

Аннотация: В работе обсуждаются особенности построения параллельных алгоритмов и их программной реализации для решения некорректно поставленных линейных обратных задач, которые сводятся к необходимости решения больших переопределенных систем линейных алгебраических уравнений с плотно заполненной матрицей. Такие обратные задачи решаются с использованием регуляризирующих алгоритмов, одним из этапов реализации которых является выбор параметра регуляризации. В вычислительно сложных многомерных обратных задачах на точность регуляризованного решения могут оказывать критическое влияние ошибки машинного округления, накапливающиеся в процессе счета. Включение оценки накопленных ошибок машинного округления в процедуру выбора параметра регуляризации позволяет повысить точность полученного регуляризованного решения, однако может существенно ограничить масштабируемость соответствующей параллельной программной реализации алгоритма. В работе сравниваются возможности различных стандартов технологии параллельного программирования Message Passing Interface по решению проблемы частичной компенсации дополнительных вычислительных и коммуникационных затрат, связанных с учетом ошибок машинного округления. Приводятся примеры реализации предложенного алгоритма на языке программирования Python с использованием пакета `mpi4py`, структура которых ориентирована на простую адаптацию для языков C/C++/Fortran.

Ключевые слова: некорректно поставленная задача, регуляризирующий алгоритм, метод сопряженных градиентов, ошибки машинного округления, распараллеливание, MPI.

Благодарности: Статья подготовлена при финансовой поддержке Министерства науки и высшего образования в рамках программы Московского центра фундаментальной и прикладной математики по Соглашению № 075–15–2025–345. Для численных экспериментов использовалось оборудование Центра коллективного пользования сверхвысокопроизводительными вычислительными ресурсами МГУ имени М. В. Ломоносова [1].

Для цитирования: Шинкарев В.Д., Златковский А.М., Лукьяненко Д.В. Параллельный алгоритм поиска регуляризованного решения плохо обусловленных СЛАУ с адаптивным учетом ошибок машинного округления // Вычислительные методы и программирование. 2026. **27**, № 2. 227–260. doi 10.26089/NumMet.v27r216.

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

Parallel algorithm for finding a regularized solution to ill-conditioned systems of linear algebraic equations with adaptive accounting for round-off errors

Valentin D. Shinkarev
Lomonosov Moscow State University, Faculty of Physics, Department of Mathematics, Moscow, Russia
ORCID: 0000-0001-8365-7224, e-mail: shinkarev.vd17@physics.msu.ru

Akim M. Zlatkovskii
Lomonosov Moscow State University, Faculty of Physics, Department of Mathematics, Moscow, Russia
ORCID: 0009-0001-8733-9650, e-mail: zlatkovskii.am20@physics.msu.ru

Dmitry V. Lukyanenko
Lomonosov Moscow State University, Faculty of Physics, Department of Mathematics, Moscow, Russia
Lomonosov Moscow State University, Research Computing Center, Moscow, Russia
Moscow Center of Fundamental and Applied Mathematics, Moscow, Russia
ORCID: 0000-0001-5140-3617, e-mail: lukyanenko@physics.msu.ru

Abstract: The paper discusses the construction of parallel algorithms and their software implementation for solving ill-posed linear inverse problems. Such problems often reduce to solving large overdetermined systems of linear algebraic equations (SLAEs) with dense matrices. These inverse problems are addressed using regularizing algorithms, a key step of which involves selecting the regularization parameter. In computationally challenging multidimensional inverse problems, the accuracy of the regularized solution can be critically affected by round-off errors accumulated during the computation. Incorporating an estimate of these accumulated errors into the procedure for selecting the regularization parameter can enhance the accuracy of the resulting solution. However, it may also significantly limit the scalability of the corresponding parallel software implementation. This work compares the capabilities of various standards of the Message Passing Interface parallel programming technology to solve the problem of partial compensation for additional computing and communication costs associated with accounting for round-off errors. Examples of the proposed algorithm's implementation are provided using the Python programming language and the `mpi4py` package, with their structure designed for straightforward adaptation to C/C++/Fortran.

Keywords: ill-posed problem, regularizing algorithm, conjugate gradient method, round-off errors, parallelization, MPI.

Acknowledgements: The work was supported by the Ministry of Education and Science of the Russian Federation as part of the program of the Moscow Center for Fundamental and Applied Mathematics under the Agreement No. 075–15–2025–345. The equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University [1] was used for numerical experiments.

For citation: V. D. Shinkarev, A. M. Zlatkovskii, and D. V. Lukyanenko, "Parallel algorithm for finding a regularized solution to ill-conditioned systems of linear algebraic equations with adaptive accounting for round-off errors," Numerical Methods and Programming. **27** (2), 227–260 (2026). doi 10.26089/NumMet.v27r216.

1. Введение. Во многих случаях прикладные обратные задачи являются линейными и сводятся к необходимости решения больших переопределенных СЛАУ вида

$$A_h\, x = b_\delta. \tag{1}$$

Здесь $A_h \in \mathbb{R}^{M\times N}$ — в общем случае плотно заполненная матрица, $x \in \mathbb{R}^N$, $b_\delta \in \mathbb{R}^M$. При этом обычно: 1) $M \geqslant N$; 2) вместо точной правой части b известно ее приближение b_δ, измеренное в эксперименте с ошибкой δ, т.е. $\|b - b_\delta\| \leqslant \delta$; 3) вместо точной матрицы A, определяющей точный закон $A : x \mapsto b$, известно ее приближение A_h, заданное с некоторой ошибкой h, т.е. $\|A - A_h\| \leqslant h$.

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

При выписанных условиях задача (1) является некорректно поставленной, и для ее решения необходимо строить регуляризирующий алгоритм. В качестве такого алгоритма может быть использован регуляризирующий алгоритм А. Н. Тихонова [2], основанный на минимизации функционала

$$M^\alpha(x) = \|A_h x - b_\delta\|^2 + \alpha\|Rx\|^2 \tag{2}$$

с выбором параметра регуляризации по обобщенному принципу невязки [2] из уравнения

$$\rho(\alpha) \equiv \|A_h x^\alpha - b_\delta\|^2 - \big(\delta + h\|x^\alpha\|\big)^2 - \mu^2 = 0. \tag{3}$$

Здесь x^α — элемент, реализующий минимум функционала (2) при заданном значении параметра регуляризации α, $\mu \equiv \mu(A_h, b_\delta)$ — мера несовместности приближенных данных, определяемая как $\mu = \inf\limits_{x\in\mathbb{R}^N} \|A_h x - b_\delta\|$; матрица R в (2) содержит информацию об априорных ограничениях на искомое решение x и в данной работе для упрощения изложения материала полагается как $R \equiv I$, где I — единичная матрица. Регуляризованное решение определяется как x^{α^*}, где α^* — корень обобщенной невязки (3).

На практике выбор параметра регуляризации α^* по обобщенному принципу невязки (3) осуществляется с помощью какого-либо итерационного алгоритма решения нелинейных уравнений. На каждом шаге такого алгоритма необходимо для очередного приближения $\alpha > 0$ к α^* искать элемент x^α, реализующий минимум функционала (2). Этот элемент может быть найден из регуляризованной системы нормальных уравнений

$$\big(A_h^T A_h + \alpha R^T R\big)x^\alpha = A_h^T b_\delta. \tag{4}$$

Таким образом, самой вычислительно сложной операцией при поиске регуляризованного решения СЛАУ вида (1) является решение системы (4), которую при этом надо решать многократно для различных значений α.

Замечание (о вычислении μ). Мера несовместности приближенных данных μ может быть вычислена как $\mu = \|A_h x^0 - b_\delta\|$, где x^0 — решение системы (4) при $\alpha = 0$.

В случае $R \equiv I$ матрица системы (4) является невырожденной и решение этой системы можно найти с помощью прямых методов решения СЛАУ с квадратной матрицей. Прямые методы решения таких систем, основанные на матричных разложениях, в большинстве своем имеют вычислительную сложность $O(MN^2)$ арифметических операций. Итерационные методы считаются более “дешевой” альтернативой. Один из самых популярных итерационных методов решения систем вида (4) — метод сопряженных градиентов (см. оригинальную работу [3]). Он является весьма экономичным и при этом обладает рядом хороших свойств: точное решение может быть найдено не более чем за N итераций (в точной арифметике); сходимость к точному решению является линейной в общем случае; возможна суперлинейная сходимость [4]. Формально вычислительная сложность этого метода также составляет $O(MN^2)$ арифметических операций, так как вычислительная сложность каждой из N итераций определяется несколькими матрично-векторными умножениями, требующими $O(MN)$ операций.

Однако, учитывая, что при решении практических задач все вычисления выполняются лишь приближенно (вследствие наличия ошибок машинного округления), утверждение о возможности минимизации функционала не более чем за N итераций в общем случае оказывается неверным. На практике возможны следующие ситуации.

1. На каждой итерации находится очередное более точное приближение искомого решения, и, начиная с какой-то итерации, решение перестает улучшаться по некоторой норме из-за наличия ошибок машинного округления. Поэтому во многих случаях возможно досрочное прекращение итерационного процесса (зачастую число итераций $n_{\text{iter}} \ll N$), в результате чего можно считать, что фактическая вычислительная сложность метода сопряженных градиентов составляет $O(n_{\text{iter}}MN)$ операций. При этом также возможна ситуация, при которой продолжение итерационного процесса и выполнение полного набора из N итераций приведет к “разрушению” численного решения исходной некорректно поставленной задачи. Таким образом, возможность досрочно прервать итерационный процесс может быть особенно полезной при решении прикладных задач. Реализация такой возможности на практике не только экономит вычислительные ресурсы, но и позволяет найти адекватное приближенное решение.
2. Из-за влияния ошибок машинного округления на точность определения направлений минимизации и шагов вдоль них после совершения N итераций значение минимизируемого функционала остается

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

достаточно большим. Это означает, что найденное через N итераций приближенное решение может быть уточнено, если итерационный процесс продолжить. В этом случае итерационный процесс необходимо продолжать до тех пор, пока значение функционала не выйдет на фон ошибок машинного округления.

Другими словами, если на практике используется "классический" критерий остановки итерационного процесса в методе сопряженных градиентов (по фиксированному числу итераций, равному N), то в первом случае приближенное решение будет найдено, но для его поиска будут затрачены излишние вычислительные ресурсы (также возможна ситуация, когда приближенное решение не будет найдено вовсе); во втором случае будет найдено очень грубое приближенное решение, которое может быть уточнено.

Таким образом, при решении многих прикладных задач чрезвычайно важным является вопрос о возможности разработки такого критерия прекращения итерационного процесса в методе сопряженных градиентов, который был бы способен учитывать накапливаемые в процессе вычислений ошибки машинного округления. Подобным вопросам посвящено множество работ (см., например, [5-15]). Несмотря на обширные исследования по этой тематике, все указанные работы сосредоточены на решении хорошо обусловленных систем. В частности, в одной из последних работ авторов [16] был сформулирован критерий прекращения итерационного процесса метода сопряженных градиентов, основанный на учете ошибок машинного округления, который на практике позволил успешно решать все указанные выше проблемные ситуации при решении больших переопределенных СЛАУ с плотно заполненной матрицей. Однако условие применимости предложенного алгоритма также заключалось в том, что он предполагал решение хорошо обусловленных СЛАУ. Такое допущение приемлемо при решении многих прикладных некорректно поставленных обратных задач, так как параметр регуляризации зачастую можно выбрать заранее на основе предварительного решения класса характерных модельных задач. В общем же случае число итераций, реализующееся в соответствии с предложенным критерием, зависит от параметра регуляризации α. В частности, для достаточно малых значений α число итераций может существенно превышать N. Как уже было упомянуто ранее, решать систему (4) необходимо для очередного приближения α в процессе реализации какого-либо итерационного метода решения уравнения $\rho(\alpha) = 0$. Очевидно, что может возникнуть необходимость решения такой системы для достаточно малых значений α (в том числе для $\alpha = 0$ при вычислении меры несовместности). Поэтому требуемое для реализации предложенного в работе [16] метода число итераций может оказаться неразумно большим. Такая особенность предложенного метода связана с тем, что оценка ошибок машинного округления для части операций не выполнялась с целью получения наиболее экономичного варианта метода, в том числе допускающего эффективную параллельную программную реализацию (см. работу [17]).

В настоящей работе рассматривается уточнение алгоритма из работы [16] и соответствующее обобщение параллельной программной реализации из работы [17] с целью применимости при любых значениях α для последующего поиска регуляризованного решения x^{α^*} задачи (1) из обобщенного принципа невязки (3).

В связи с вышесказанным структура статьи следующая. Сначала, в разделе 2, для решения самой вычислительно сложной задачи — решения системы (4) — формулируется усовершенствованный метод сопряженных градиентов в виде алгоритма, в котором прекращение итерационного процесса согласуется со значением накопленных ошибок машинного округления. Затем в разделе 3 описываются параллельные алгоритмы реализации некоторых базовых операций линейной алгебры, которые будут использоваться для построения параллельного варианта алгоритма из раздела 2. Наконец, в разделе 4 описываются различные подходы к построению параллельной версии этого алгоритма и его программной реализации с учетом возможностей различных стандартов MPI [18, 19]. В частности, акцент делается на преимуществах использования неблокирующих операций коллективного взаимодействия процессов из стандартов MPI-3 и MPI-4. В разделе 5 формулируется алгоритм поиска значения параметра регуляризации из обобщенного принципа невязки (3). В разделе 6 с целью воспроизводимости результатов приводится описание тестового примера и его программная реализация. В разделе 7 проводится исследование предложенных программных реализаций на наличие сильной масштабируемости, а также даются рекомендации по их использованию при вычислениях на больших вычислительных системах с распределенной памятью. В разделе 8 приводятся некоторые примеры расчетов. Исходные тексты программ доступны на сайте `https://github.com/rafsip/parallel_cg_with_adaptive_accounting_of_round-off_errors`.

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

2. Усовершенствованный метод сопряженных градиентов. В работе [16] был упомянут подход к учету ошибок машинного округления, который позволяет сформулировать усовершенствованный метод сопряженных градиентов, представленный в виде алгоритма 1.

Алгоритм 1. Псевдокод усовершенствованного метода сопряженных градиентов iCG для решения системы нормальных уравнений (4) (для случая $R = I$)

Algorithm 1. Pseudocode of the improved conjugate gradient method iCG for solving a system of normal equations (4) (for the case of $R = I$)

Входные данные: A_h, b_δ, α
Результат: $x \equiv x^\alpha$
$x \leftarrow 0$
$s \leftarrow 1$
$p \leftarrow 0$
while *True* **do**
  if $s = 1$ **then**
    $r \leftarrow A_h^T(A_h x - b_\delta) + \alpha x$
    $D_r \leftarrow \left(A_h^T\right)^{\circ 2}\left(A_h^{\circ 2} x^{\circ 2} + b^{\circ 2}\right) + \alpha^2 x^{\circ 2}$
  else
    $r \leftarrow r - \dfrac{q}{(p,q)}$
    $D_r \leftarrow D_r + \dfrac{1}{(p,q)^2} \cdot \left\{ q^{\circ 2}, \dfrac{(p,q)^2 D_q - 2(p,q)p \circ q \circ D_q + (p^{\circ 2}, D_q)q^{\circ 2}}{(p,q)^2} \right.$
  end
  if $\dfrac{\Delta^2 \sqrt{(D_r, D_r)}}{(r,r)} \geqslant 1$ **then**
    return x, s
  end
  $p \leftarrow p + \dfrac{r}{(r,r)}$
  $q \leftarrow A_h^T(A_h p) + \alpha p$
  $D_q \leftarrow \left(A_h^T\right)^{\circ 2}(A_h^{\circ 2} p^{\circ 2}) + \alpha^2\, p^{\circ 2}$
  $x \leftarrow x - \dfrac{p}{(p,q)}$
  $s \leftarrow s + 1$
end

Здесь $^{\circ 2}$ обозначает “степень Адамара” (Hadamard power) — поэлементное возведение вектора/матрицы во вторую степень, $\circ$ — поэлементную операцию умножения матриц, Δ — относительную ошибку машинного округления ($\Delta = 10^{-16.3}$ для двойной точности, $\Delta = 10^{-34}$ для четверной точности).

Если опустить строки, выделенные синим цветом, то получится менее точный, но более быстрый вариант алгоритма, который был описан в работе авторов [16] и который допускает эффективную параллельную реализацию (см. работу [17]). Однако этот упрощенный вариант применим только для тех значений параметра регуляризации α, при которых система (4) является хорошо обусловленной. Более точный учет ошибок машинного округления (строки, выделенные синим) позволяет решать систему (4) для произвольных значений α, что существенно при реализации итерационных методов поиска корня α^* обобщенной невязки (3).

Замечание. Если в алгоритме 1 полностью исключить действия, выделенные красным и синим цветом, и заменить условие *True* на $s \leqslant N$, то получится классическая версия метода сопряженных градиентов.

Замечание. Множитель, выделенный синим цветом в формуле для обновления D_r в алгоритме 1, может быть представлен в альтернативной форме записи

$$\frac{(p,q)^2 D_q - 2(p,q)p \circ q \circ D_q + (p^{\circ 2}, D_q)q^{\circ 2}}{(p,q)^2} = \left(I - \frac{qp^T}{(p,q)}\right)^{\circ 2} D_q,$$

которая является более компактной, но с вычислительной точки зрения требует выполнения большего числа арифметических операций. Это связано с тем, что умножение матрицы размера $N \times N$ на вектор

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

требует выполнения порядка N^2 операций, тогда как оригинальная форма записи предполагает выполнение лишь порядка N арифметических операций.

Замечание (о способе вычисления меры несовместности приближенных данных μ). При выполнении алгоритма 1 с параметром $\alpha = 0$ найденный вектор x^0 позволяет вычислить меру несовместности приближенных данных:

$$\mu = \|A_h x^0 - b_\delta\|.$$

Python-код для функции, реализующей алгоритм 1, имеет достаточно компактный вид (листинг 1).

Листинг 1. Реализация алгоритма 1 на языке Python
Listing 1. Implementation of the algorithm 1 in Python

```python
import numpy as np

def iCG(A, b, alpha=0):
    M, N = np.shape(A)
    delta = np.finfo(np.float64).eps
    s = 1

    x = np.zeros(N)
    p = np.zeros(N)
    A2 = A**2

    while True:
        if s == 1:
            r = np.dot(A.T, np.dot(A, x) - b) + alpha * x
            Dr = np.dot(A2.T, np.dot(A2, x**2) + b**2) + alpha**2 * x**2
        else:
            r -= q / pq
            Dr += (pq**2 * Dq
                   - (2 * pq * p * q * Dq)
                   + Dpq * q**2) / pq**4

        rr = np.dot(r, r)
        Ds = np.sum(Dr)

        if delta**2 * Ds / rr >= 1:
            return x, s

        p += r / rr

        q = np.dot(A.T, np.dot(A, p)) + alpha * p
        Dq = np.dot(A2.T, np.dot(A2, p**2)) + alpha**2 * p**2

        pq = np.dot(p, q)
        Dpq = np.dot(p**2, Dq)

        x -= p / pq
        s += 1
```

Данная программная реализация содержит следующие особенности.

1. Функция возвращает: 1) массив `x`, который содержит решение системы (4), найденное по усовершенствованному алгоритму 1, 2) число `s` итераций, которые потребовалось сделать алгоритму, чтобы найти приближенное решение.

2. Особенности обозначений: результаты скалярных произведений векторов обозначаются как “склеенные” имена векторов, например, результат (p, q) обозначается как `pq`, результат (r, r) — как `rr`; с помощью `Ds` обозначено выражение $\sqrt{(D_r, D_r)}$, а с помощью `Dpq` — $(p^{\circ 2}, D_q)$.

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

3. Отметим также некоторые особенности синтаксиса языка Python и наименования функций пакета `numpy`, возможно неочевидные для читателей, незнакомых с ними: возведение в степень обозначается как `**`, транспонирование матрицы — `.T`, скалярное произведение векторов и произведение матрицы на вектор — `np.dot` (можно передать дополнительный именованный аргумент `out=`, чтобы не выделять дополнительную память на результат операции).

4. Предполагается, что для вычислений используется стандартный пакет `numpy`. Как следствие, эта вроде бы последовательная программная реализация алгоритма на самом деле содержит параллельные вычисления, что связано с особенностью используемой для основных вычислений функции `dot()` из пакета `numpy`. Эта функция реализована на языке программирования C++/Fortran и при расчетах автоматически использует многопоточность, если программа запущена на многоядерном процессоре — все ядра процессора задействуются за счет использования технологии параллельного программирования OpenMP.

5. В алгоритме присутствует как умножение матрицы на вектор, так и умножение матрицы, возведенной поэлементно в квадрат, т.е. $A^{\circ 2}$, на вектор.

   В `numpy` нет встроенной операции для умножения $A^{\circ 2}$ на вектор, операции `A**2` и `A.T**2` включают в себя аллокацию дополнительной памяти под хранение промежуточных данных. Это очень плохо, потому что подразумевается, что решается задача (1) с огромной матрицей A. Поэтому при решении реальных "больших" задач массив, содержащий матрицу A, может занимать большую часть памяти. Как следствие, на дополнительные массивы места в памяти может не хватить.

   Эту проблему можно обойти несколькими путями.

   Во-первых, можно оформить указанную команду в виде отдельной функции на языке программирования C/C++/Fortran, используя для вычислений только элементы массива, в котором хранится матрица A.

   Во-вторых, можно использовать функцию `np.einsum('ij,ij,j->i', A, A, x**2)` (где не реализовано распараллеливание с помощью технологии OpenMP) или написать собственную функцию

   ```
   Dr = dot_special(A.T, dot_special(A, x**2, M, N) + b**2, N, M)
   ```

   которая использует jit-компиляцию с помощью пакета `numba`:

   ```
   from numba import jit, prange

   @jit(nopython=True, parallel=True)
   def dot_special(A, x, M, N):
       b = np.zeros(M)
       for m in prange(M):
           for n in range(N):
               b[m] += A[m, n]**2 * x[n]
       return b
   ```

   Мы будем использовать третий путь: матрица $A_2 = A^{\circ 2}$ будет вычисляться и сохраняться до начала итерационного процесса для последующего использования. Хотя в этом случае алгоритм требует примерно в два раза больше оперативной памяти (операция `A2.T` дополнительной памяти не выделяет), но при этом на каждой итерации пропадает необходимость в выполнении дополнительных $M \times N$ арифметических операций по возведению в квадрат элементов исходной матрицы, что эквивалентно самым вычислительноемким операциям на каждой итерации алгоритма — умножению матрицы на вектор и умножению транспонированной матрицы на вектор.

6. Для простоты программы у матрицы A_h и вектора b_δ опускаются индексы (в программе используются как `A` и `b` соответственно).

3. Подходы к построению параллельного алгоритма и его программной реализации. В данном разделе будут описаны основные подходы, используемые для построения параллельного варианта рассматриваемого алгоритма решения самой вычислительно сложной задачи — решения системы (4) — и его последующей программной реализации.

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

3.1. Распараллеливаемые операции. В рассматриваемой реализации метода сопряженных градиентов в виде алгоритма 1 для решения системы (4) все вычисления приходятся на следующие четыре операции.

1. Скалярное произведение двух векторов размера N.

   Эта операция требует N умножений и $N-1$ сложение — в сумме $2N-1$ арифметических операций. Принято обозначать $2N-1=O(N^1)$, что говорит о линейной вычислительной сложности операции скалярного произведения.

2. Умножение матрицы размера $M \times N$ на вектор размера N.

   Для получения каждого элемента итогового вектора необходимо скалярно умножить соответствующую строку матрицы A на умножаемый вектор. Такие вычисления надо провести для каждого элемента вектора, являющегося результатом умножения. Следовательно, в сумме надо совершить $M \cdot (2N-1)$ арифметических операций. В случае квадратной матрицы ($M=N$) таких операций будет $O(N^2)$, что говорит о квадратичной вычислительной сложности операции умножения матрицы на вектор.

3. Умножение транспонированной матрицы размера $N \times M$ на вектор размера M.

   Вычислительная сложность этой операции эквивалентна операции умножения матрицы на вектор.

4. Сложение двух векторов размера N (или M) или умножение вектора на число.

   Эта операция требует N (M) арифметических операций, т.е. вычислительная сложность этой операции является линейной.

Далее рассматриваются используемые для распараллеливания этих операций алгоритмы (см., например, [20]), которые будут применены для построения параллельной версии алгоритма 1 и его программной реализации.

3.2. Распределение данных по участвующим в вычислениях MPI-процессам. Предполагается, что все данные задачи распределены по адресному пространству n вычислительных MPI-процессов, участвующих в вычислениях и входящих в коммуникатор `comm` $\equiv$ `MPI.COMM_WORLD`, следующим образом.

Каждый вычислительный процесс имеет свой идентификатор/номер `rank` $\in \{0,\dots,n-1\}$. Матрица A размера $M \times N$ делится между этими процессами двумерным образом (рис. 1) на блоки $A_{\mathrm{part}_{(\mathtt{rank})}}$ размера $M_{\mathrm{part}_{(k)}} \times N_{\mathrm{part}_{(l)}}$. Тем самым предполагается, что процессы образуют двумерную решетку процессов размера $n_1 \times n_2$ (при этом $n_1 \cdot n_2 = n$). Поэтому номер процесса `rank` связан с индексами k и l, которые определяют координаты процесса в двумерной решетке процессов следующим образом:

$$k = \left\lfloor \frac{\mathtt{rank}}{n_2} \right\rfloor, \quad l = \mathtt{rank} - \left\lfloor \frac{\mathtt{rank}}{n_2} \right\rfloor \cdot n_2.$$

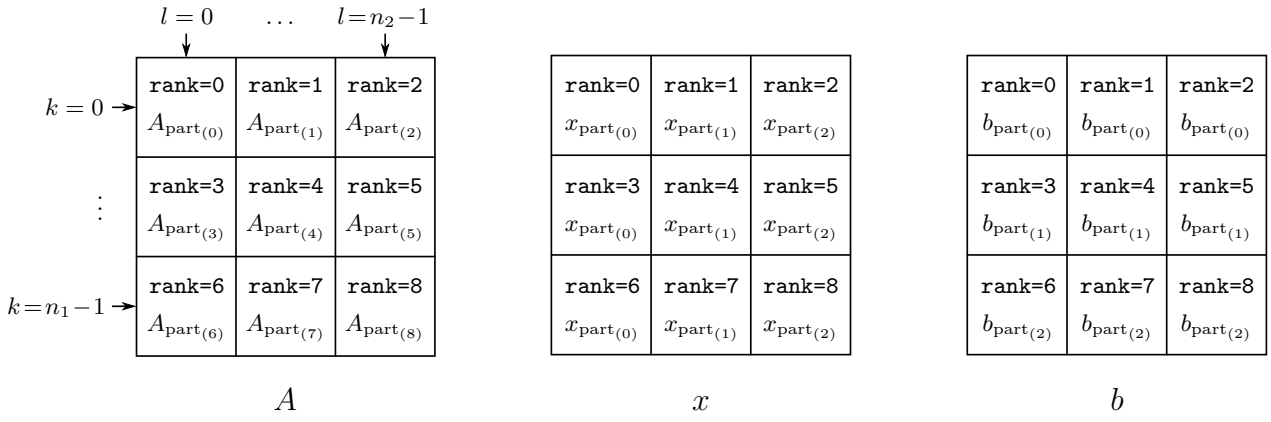

Рис. 1. Пример распределения данных по девяти вычислительным процессам, которые образуют двумерную решетку 3×3

Fig. 1. An example of data distribution across nine computing processes that form a two-dimensional 3×3 lattice

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

Либо, если необходимо пересчитать `rank` по k и l:

$$\texttt{rank} = k \cdot n_2 + l.$$

Отметим также, что

$$\sum_{k=0}^{n_1-1} M_{\mathrm{part}_{(k)}} = M, \quad \sum_{l=0}^{n_2-1} N_{\mathrm{part}_{(l)}} = N.$$

Вектор x делится на n_2 частей $x_{\mathrm{part}_{(l)}}$, $l = \overline{0, n_2 - 1}$, размеров $N_{\mathrm{part}_{(l)}}$. При этом часть $x_{\mathrm{part}_{(l)}}$ для фиксированного индекса l хранится на всех процессах столбца решетки процессов с индексом l (рис. 1), т.е. на процессах с координатой $(\cdot, l)$ в двумерной решетке процессов.

Вектор b делится на n_1 частей $b_{\mathrm{part}_{(k)}}$, $k = \overline{0, n_1 - 1}$, размеров $M_{\mathrm{part}_{(k)}}$. При этом часть $b_{\mathrm{part}_{(k)}}$ для фиксированного индекса k хранится на всех процессах строки решетки процессов с индексом k (рис. 1), т.е. на процессах с координатой $(k, \cdot)$ в двумерной решетке процессов.

Таким образом, предполагается, что на каждом вычислительном процессе содержится один из блоков $A_{\mathrm{part}_{(\cdot)}}$ матрицы A, а также одна из частей $x_{\mathrm{part}_{(\cdot)}}$ и $b_{\mathrm{part}_{(\cdot)}}$ векторов x и b соответственно.

3.3. Параллельный алгоритм умножения матрицы на вектор в случае двумерного деления матрицы на блоки. При сформулированном способе хранения данных по вычислительным процессам результатом умножения матрицы A размера $M \times N$ на вектор x размера N является вектор (Ax), который будет распределен по различным вычислительным процессам по частям $(Ax)_{\mathrm{part}_{(\cdot)}}$. При этом структура распределения этого вектора по различным процессам должна соответствовать структуре распределения вектора b (рис. 1). Часть вектора (Ax) может быть вычислена по следующей формуле:

$$(Ax)_{\mathrm{part}_{(k)}} = \sum_{l=0}^{n_2-1} A_{\mathrm{part}_{(k \cdot n_2 + l)}} x_{\mathrm{part}_{(l)}}, \quad k = \overline{0, n_1 - 1}. \tag{5}$$

На рис. 2 приведен поясняющий эту формулу пример.

Каждый процесс, участвующий в вычислениях, может вычислить слагаемое $A_{\mathrm{part}_{(k \cdot n_2 + l)}} x_{\mathrm{part}_{(l)}}$ для своей пары значений (k, l) независимо от аналогичных вычислений, проводимых другими процессами, поэтому слагаемые в формуле (5) могут быть вычислены параллельно. Затем слагаемые, расположенные на процессах строки решетки процессов с индексом k, должны быть просуммированы, а результат — вектор $(Ax)_{\mathrm{part}_{(k)}}$ — должен быть размещен на всех процессах этой строки решетки процессов. Организация взаимодействия процессов влечет накладные расходы на прием/передачу сообщений, содержащих

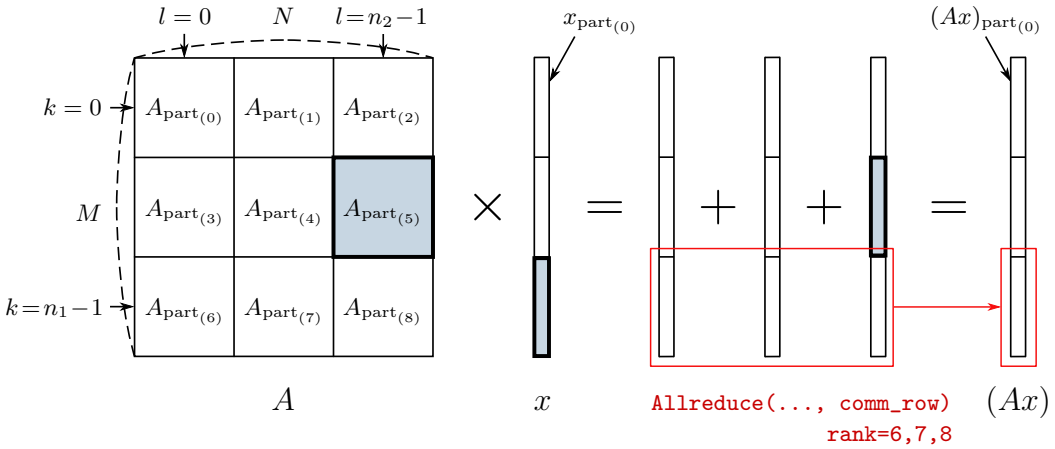

Рис. 2. Параллельный алгоритм умножения матрицы на вектор в случае двумерного разбиения матрицы на части. Показано распределение данных по девяти вычислительным процессам, которые образуют двумерную решетку 3×3

Fig. 2. A parallel algorithm for matrix-vector multiplication in the case of a two-dimensional matrix partition. The case of data distribution across nine computational processes that form a two-dimensional 3×3 lattice is depicted

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

результаты промежуточных вычислений через физическую коммуникационную среду многопроцессорной вычислительной системы.

Таким образом, этот параллельный алгоритм (учитывающий двумерное деление матрицы A на блоки) содержит следующие особенности.

1. На каждом процессе надо хранить не весь вектор x, а лишь часть $x_{\mathrm{part}\,(l)}$ этого вектора. Вектор $x_{\mathrm{part}\,(l)}$ придется рассылать не по всем процессам коммуникатора `comm`, а только по части процессов этого коммуникатора — по процессам столбца с индексом l решетки процессов. При этом передача данных среди процессов каждого столбца сетки процессов может быть организована параллельно с передачей данных среди процессов других столбцов сетки процессов, что даст выигрыш по времени.
2. При вычислении части $(Ax)_{\mathrm{part}\,(k)}$ итогового вектора (Ax) необходимо обмениваться сообщениями не всем процессам коммуникатора `comm`, а лишь части процессов этого коммуникатора — процессам строки с индексом k введенной решетки процессов. При этом передача данных вдоль каждой строки решетки процессов может быть организована параллельно с передачей данных среди процессов других строк решетки процессов, что также даст выигрыш по времени.

Опишем теперь некоторые особенности программной реализации этого алгоритма с учетом возможностей технологии параллельного программирования MPI.

Как известно, обмен сообщениями между процессами, организованный с помощью MPI-функций `Send()` и `Recv()`, возможен, но неэффективен. Гораздо эффективнее использовать MPI-функции коллективного взаимодействия процессов. Но такие функции должны быть вызваны на всех процессах коммуникатора. В текущий момент мы работаем только с коммуникатором `comm`, который кроме тех процессов, которые мы хотим задействовать (процессы из отдельного столбца или строки решетки процессов), содержит и другие процессы, которые между собой взаимодействовать не будут. Поэтому мы приходим к необходимости создания дополнительных коммуникаторов, которые будут содержать только те группы процессов, внутри которых мы хотим организовывать взаимодействия по обмену данными с помощью функций коллективного взаимодействия процессов.

Такие группы коммуникаторов могут быть созданы, например, следующим образом.

```
comm_col = comm.Split(rank % n_2, rank)
comm_row = comm.Split(rank // n_1, rank)
```

Здесь, например, в первой строке порождаются коммуникаторы `comm_col`. Обратим внимание на то, что порождаются именно коммуникаторы (во множественном числе), а не один коммуникатор. Давайте приведем поясняющий пример для случая 9-ти MPI-процессов (т.е. `numprocs = 9`). Первым аргументом функции `Split()` является значение `color`, определенное нами как `rank % n_2`, которое для процессов с `rank = 0, 1, 2, 3, 4, 5, 6, 7, 8` примет значения `0, 1, 2, 0, 1, 2, 0, 1, 2`. Таким образом будет создано три коммуникатора: в первый войдут процессы со значением `color = 0`, во второй — со значением `color = 1`, в третий — со значением `color = 2`. Можно дать наглядную интерпретацию такой группировки процессов — см. рис. 2: если процессы исходного коммуникатора `comm` образуют двумерную решетку, то процессы коммуникаторов `comm_col` являются столбцами этой двумерной решетки процессов.

При этом объект `comm_col` на каждом MPI-процессе будет содержать информацию о разных процессах. Например, на процессах с `rank = 1, 4` исходного коммуникатора `comm` объект `comm_col` будет содержать информацию о процессах с `rank = 1, 4, 7` исходного коммуникатора `comm`. А на процессе с `rank = 2` исходного коммуникатора `comm` объект `comm_col` будет содержать информацию о процессах с `rank = 2, 5, 8` исходного коммуникатора `comm`.

Аналогично, если процессы исходного коммуникатора `comm` образуют двумерную решетку, то процессы коммуникаторов `comm_row` являются строками этой двумерной решетки процессов.

Таким образом, параллельная программная реализация умножения матрицы на вектор может быть оформлена в следующем виде.

```
b_part = np.dot(A_part, x_part)
allreduce(comm_row, b_part, M_part)
```

Здесь мы для удобства и краткости вводим вспомогательную функцию для MPI-операции `Allreduce()`:

```
def allreduce(comm, part, size):
    comm.Allreduce(MPI.IN_PLACE, [part, size, MPI.DOUBLE], op=MPI.SUM)
```

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

Отметим, что результат такого умножения (вектор $b \equiv Ax$) будет храниться на всех процессах по частям: часть вектора $b_{\mathrm{part}_{(k)}}$ для фиксированного индекса k будет храниться во всех ячейках строки решетки процессов с индексом k.

Замечание. В указанной программной реализации переменная `b_part` предварительно используется для хранения промежуточного результата произведения, а затем переиспользуется для хранения окончательного результата произведения на данном MPI-процессе. Это позволяет переиспользовать память, но при этом результат вычисления `b_part = np.dot(A_part, x_part)` формально нельзя назвать b_{part}, так как он содержит лишь одно слагаемое из правой части выражения (5).

Замечание. Отметим, что время обмена сообщениями в такой реализации пропорционально $\log_2 n_2$, но при этом объем передаваемых сообщений пропорционален $\dfrac{1}{n_1}$.

3.4. Параллельный алгоритм умножения транспонированной матрицы на вектор в случае двумерного деления матрицы на блоки. Результатом умножения транспонированной матрицы A^T размера $N \times M$ на вектор b размера M будет вектор $(A^T b)$, который будет распределен по различным вычислительным процессам по частям $(A^T b)_{\mathrm{part}_{(\cdot)}}$. При этом структура распределения этого вектора по различным процессам должна соответствовать структуре распределения вектора x (рис. 1). Часть вектора $(A^T b)$ может быть вычислена по следующей формуле:

$$(A^T b)_{\mathrm{part}_{(l)}} = \sum_{k=0}^{n_1-1} A^T_{\mathrm{part}_{(k\cdot n_2+l)}} b_{\mathrm{part}_{(k)}}, \quad l = \overline{0, n_2-1}. \tag{6}$$

На рис. 3 приведен поясняющий эту формулу пример.

Каждый процесс, участвующий в вычислениях, может вычислить слагаемое $A^T_{\mathrm{part}_{(k\cdot n_2+l)}} b_{\mathrm{part}_{(k)}}$ для своей пары значений (k, l) независимо от аналогичных вычислений, проводимых другими процессами. Поэтому слагаемые в формуле (6) могут быть вычислены параллельно. Затем соответствующие слагаемые, расположенные на процессах столбца решетки процессов с индексом l, должны быть просуммированы, а результат — вектор $(A^T b)_{\mathrm{part}_{(l)}}$ — должен быть размещен на всех процессах этого столбца решетки процессов.

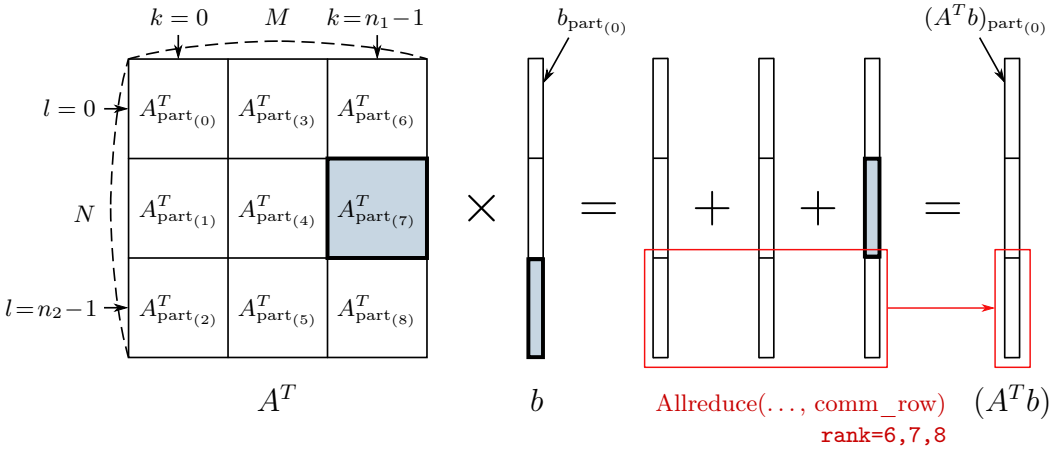

Рис. 3. Параллельный алгоритм умножения транспонированной матрицы на вектор в случае двумерного разбиения матрицы на блоки. Изображен случай распределения данных по девяти вычислительным процессам, которые образуют двумерную решетку 3×3. С учетом транспонирования предполагается, что строки изображенной решетки процессов являются столбцами исходной решетки процессов; аналогично, столбцы представленной решетки процессов являются строками исходной решетки процессов

Fig. 3. A parallel algorithm for multiplying a transposed matrix by a vector in the case of a two-dimensional matrix division into blocks. The case of data distribution across nine computing processes that form a two-dimensional 3×3 lattice is depicted. Taking into account the transposition, it is assumed that the rows of the depicted process lattice are the columns of the original process lattice; similarly, the columns of the presented process lattice are the rows of the original process lattice

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

Таким образом, этот параллельный алгоритм (учитывающий двумерное деление матрицы A на блоки) содержит следующие особенности.

1. На каждом процессе надо хранить не весь вектор b, а лишь часть $b_{\mathrm{part}_{(k)}}$ этого вектора. Вектор $b_{\mathrm{part}_{(k)}}$ придется рассылать не по всем процессам коммуникатора `comm`, а только по части процессов этого коммуникатора — по процессам строки с индексом k решетки процессов. При этом передача данных среди процессов каждой строки решетки процессов может быть организована параллельно с передачей данных среди процессов других строк решетки процессов, что даст выигрыш по времени.
2. При вычислении части $x_{\mathrm{part}_{(l)}}$ итогового вектора x необходимо обмениваться сообщениями не всем процессам коммуникатора `comm`, а лишь части процессов этого коммуникатора — процессам столбца с индексом l введенной решетки процессов. При этом передача данных вдоль каждого столбца решетки процессов может быть организована параллельно с передачей данных среди процессов других столбцов решетки процессов, что также даст выигрыш по времени.

Как прекрасно видно, параллельный алгоритм умножения транспонированной матрицы на вектор в случае двумерного деления матрицы на блоки обладает симметрией относительно параллельного алгоритма умножения матрицы на вектор, рассмотренного в предыдущем подразделе. Этот алгоритм получается из рассмотренного ранее за счет замен $k \leftrightarrow l$ и “строки” $\leftrightarrow$ “столбцы”. При этом все действия совершаются не с частями матрицы $A_{\mathrm{part}_{(\mathtt{rank})}}$, а с их транспонированными частями $A^T_{\mathrm{part}_{(\mathtt{rank})}}$. Следовательно, программная реализация этого алгоритма мало отличается от программной реализации параллельного алгоритма умножения матрицы на вектор (в том числе необходима работа с дополнительными коммуникаторами, которые будут содержать только те группы процессов, среди которых мы хотим организовывать взаимодействия по обмену данными с помощью функций коллективного взаимодействия процессов).

Таким образом, параллельная программная реализация умножения транспонированной матрицы на вектор (с учетом того, что мы оперируем с элементами исходной нетранспонированной матрицы) может быть оформлена в следующем виде.

```
x_part = np.dot(A_part.T, b_part)
allreduce(comm_col, x_part, N_part)
```

Отметим, что результат такого умножения (вектор $x \equiv A^T b$) будет храниться на всех процессах по частям: часть вектора $x_{\mathrm{part}_{(l)}}$ для фиксированного индекса l будет храниться во всех ячейках столбца решетки процессов с индексом l.

3.5. Параллельный алгоритм скалярного умножения векторов. Так как размер векторов в последовательных алгоритмах решения задачи (1) составляет либо M, либо N, то будем использовать характерные векторы этого размера. Поэтому в случае упоминания вектора x будет предполагаться, что его размер N, а в случае упоминания вектора b будет предполагаться, что его размер M. При этом предполагается, что эти векторы распределены по вычислительным процессам по схеме, представленной на рис. 1.

Результатом скалярного произведения векторов $x^{(1)}$ и $x^{(2)}$ размера N будет число $(x^{(1)}, x^{(2)})$. Оно может быть вычислено по следующей формуле:

$$\left(x^{(1)}, x^{(2)}\right) = \sum_{l=0}^{n_2-1} \left(x^{(1)}_{\mathrm{part}_{(l)}}, x^{(2)}_{\mathrm{part}_{(l)}}\right). \tag{7}$$

Каждый процесс, участвующий в вычислениях, может вычислить слагаемое $\left(x^{(1)}_{\mathrm{part}_{(l)}}, x^{(2)}_{\mathrm{part}_{(l)}}\right)$ для своего значения l независимо от аналогичных вычислений, проводимых другими процессами, поэтому слагаемые в формуле (7) могут быть вычислены параллельно. Затем соответствующие слагаемые, расположенные на процессах строки решетки процессов, должны быть просуммированы, а результат — число $\left(x^{(1)}, x^{(2)}\right)$ — должен быть размещен на всех процессах этой строки решетки процессов.

Таким образом, параллельная программная реализация скалярного умножения векторов может быть оформлена в следующем виде.

```
res = np.array(0, np.float64)
np.dot(x1_part, x2_part, out=res)
allreduce(comm_row, res, 1)
```

Отметим, что результат такой программной реализации скалярного умножения будет храниться на всех процессах коммуникатора `comm`.

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

При этом необходимо отметить следующую особенность алгоритма. Процессы каждой строки решетки процессов выполняют те же вычисления, что и процессы других строк решетки процессов. Следовательно, вычисления скалярного произведения распараллеливаются не по всем n вычислительным процессам, а только по n_2 процессам. Такой подход является конструктивным, так как распределение векторов $x^{(1)}$ и $x^{(2)}$ по всем n процессам (а не только по n_2 процессам строки решетки процессов), а затем сбор данных со всех них потребует дополнительных накладных расходов, пропорциональных $\log_2 n$, что при решении "больших" задач практически всегда превосходит выигрыш в уменьшении времени непосредственных вычислений: $\sim (2N-1)/n$ вместо $\sim (2N-1)/n_2$ в рассматриваемом варианте параллельного алгоритма.

Аналогично, результатом скалярного произведения векторов $b^{(1)}$ и $b^{(2)}$ размера M будет число $\left(b^{(1)}, b^{(2)}\right)$. Оно может быть вычислено по следующей формуле:

$$\left(b^{(1)}, b^{(2)}\right) = \sum_{k=0}^{n_1-1} \left(b^{(1)}_{\mathrm{part}\,(k)}, b^{(2)}_{\mathrm{part}\,(k)}\right). \tag{8}$$

Каждый процесс, участвующий в вычислениях, может вычислить слагаемое $\left(b^{(1)}_{\mathrm{part}\,(k)}, b^{(2)}_{\mathrm{part}\,(k)}\right)$ для своего значения k независимо от аналогичных вычислений, проводимых другими процессами, поэтому слагаемые в формуле (8) могут быть вычислены параллельно. Затем соответствующие слагаемые, расположенные на процессах столбца решетки процессов, должны быть просуммированы, а результат — число $\left(b^{(1)}, b^{(2)}\right)$ — должен быть размещен на всех процессах этого столбца решетки процессов. Организация взаимодействия процессов влечет накладные расходы на прием/передачу сообщений, содержащих результаты промежуточных расчетов, через физическую коммуникационную среду многопроцессорной вычислительной системы.

Таким образом, параллельная программная реализация скалярного умножения векторов длины M может быть оформлена в следующем виде.

```
res = np.array(0, np.float64)
np.dot(b1_part,b2_part, out=res)
allreduce(comm_col, res, 1)
```

4. Параллельная версия усовершенствованного алгоритма и его программная реализация. В этом разделе описываются различные подходы к построению параллельной версии алгоритма 1 и его программной реализации. Эти подходы учитывают возможности различных стандартов MPI. Сначала (см. подраздел 4.1) описывается "наивный" подход к распараллеливанию, который основан на использовании стандарта MPI-2. В этом подходе этапы вычислений и этапы пересылки сообщений, содержащих результаты промежуточных вычислений, разделены. Затем (см. подраздел 4.2) описывается подход к распараллеливанию, основанный на использовании стандарта MPI-3. Этот подход за счет использования неблокирующих (асинхронных) операций обмена сообщениями между процессами позволяет скрыть часть дополнительных вычислений, связанных с учетом ошибок машинного округления, на фоне обмена сообщениями между процессами в основном алгоритме, а дополнительные пересылки сообщений (связанные с появлением дополнительных операций по учету ошибок машинного округления) скрыть на фоне вычислений основного алгоритма. За счет этого многие дополнительные операции (вычисления и пересылка сообщений между MPI-процессами) не требуют дополнительных затрат времени. Затем (см. подраздел 4.3) даются комментарии касательно использования стандарта MPI-4, который позволяет за счет использования отложенных запросов на взаимодействие значительно повысить эффективность параллельной программной реализации.

4.1. Учет возможностей стандарта MPI-2. "Наивный" параллельный вариант алгоритма 1, реализующего усовершенствованный метод сопряженных градиентов для решения системы (4), может быть оформлен в виде следующего псевдокода (алгоритм 2). В этом алгоритме выделенные красным цветом строки соответствуют действиям, которые надо совершить, чтобы реализовать усовершенствованный критерий прекращения итерационного процесса. Если эти строки убрать и изменить условие $True$ на $s \leqslant N$, то получится классическая реализация метода сопряженных градиентов для решения системы (4).

Программная реализация функции, которая реализует алгоритм 2 для решения системы (4), показана в листинге 2. В ней предполагается, что входными данными функции являются: 1) части A_{part} и b_{part} исходной матрицы A и вектора b; 2) параметр регуляризации α; 3) вспомогательные коммуникаторы `comm_row` и `comm_col`.

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

Алгоритм 2. Псевдокод (“наивный”) для параллельной версии алгоритма iCG

Algorithm 2. Pseudocode (“naive”) for a parallel version of the iCG algorithm

Data: A_{part}, b_{part}, α, `comm_row`, `comm_col`
Результат: $x_{\text{part}} \equiv x^{\alpha}_{\text{part}}$
$x_{\text{part}} \leftarrow 0$; $s \leftarrow 1$; $p_{\text{part}} \leftarrow 0$
while *True* **do**
  if $s = 1$ **then**
    $t_{\text{part}} \leftarrow A_{\text{part}} x_{\text{part}}$
    $t_{\text{part}} \leftarrow$ `comm_row.Allreduce`(t_{part})
    $D^t_{\text{part}} \leftarrow A^{\circ 2}_{\text{part}} x^{\circ 2}_{\text{part}}$
    $D^t_{\text{part}} \leftarrow$ `comm_row.Allreduce`(D^t_{part})
    $t_{\text{part}} \leftarrow t_{\text{part}} - b_{\text{part}}$
    $r_{\text{part}} \leftarrow A^T_{\text{part}} t_{\text{part}}$
    $r_{\text{part}} \leftarrow$ `comm_col.Allreduce`(r_{part})
    $D^t_{\text{part}} \leftarrow D^t_{\text{part}} + b^{\circ 2}_{\text{part}}$
    $D^r_{\text{part}} \leftarrow (A^T_{\text{part}})^{\circ 2} D^t_{\text{part}}$
    $D^r_{\text{part}} \leftarrow$ `comm_col.Allreduce`(D^r_{part})
    $r_{\text{part}} \leftarrow r_{\text{part}} + \alpha x_{\text{part}}$
    $D^r_{\text{part}} \leftarrow D^r_{\text{part}} + \alpha^2 x^{\circ 2}_{\text{part}}$
  else
    $$r_{\text{part}} \leftarrow r_{\text{part}} - \frac{q_{\text{part}}}{\text{pq}}$$
    $$D^r_{\text{part}} \leftarrow D^r_{\text{part}} + \frac{\text{pq}^2 D^q_{\text{part}} - 2 \cdot \text{pq} \cdot p_{\text{part}} \circ q_{\text{part}} \circ D^q_{\text{part}} + q^{\circ 2}_{\text{part}} D^{\text{pq}}}{\text{pq}^4}$$
  end
  $\text{rr} \leftarrow (r_{\text{part}}, r_{\text{part}})$
  $\text{rr} \leftarrow$ `comm_row.Allreduce`(rr)
  $D^s \leftarrow \sum_n (D^r_{\text{part}})_n$
  $D^s \leftarrow$ `comm_row.Allreduce`(D^s)
  if $\dfrac{\Delta^2 D^s}{\text{rr}} \geqslant 1$ **then**
    return x_{part}, s
  end
  $p_{\text{part}} \leftarrow p_{\text{part}} + \dfrac{r_{\text{part}}}{\text{rr}}$
  $t_{\text{part}} \leftarrow A_{\text{part}} p_{\text{part}}$
  $t_{\text{part}} \leftarrow$ `comm_row.Allreduce`(t_{part})
  $D^t_{\text{part}} \leftarrow A^{\circ 2}_{\text{part}} p^{\circ 2}_{\text{part}}$
  $D^t_{\text{part}} \leftarrow$ `comm_row.Allreduce`(D^t_{part})
  $q_{\text{part}} \leftarrow A^T_{\text{part}} t_{\text{part}}$
  $q_{\text{part}} \leftarrow$ `comm_col.Allreduce`(q_{part})
  $D^q_{\text{part}} \leftarrow (A^T_{\text{part}})^{\circ 2} D^t_{\text{part}}$
  $D^q_{\text{part}} \leftarrow$ `comm_col.Allreduce`(D^q_{part})
  $q_{\text{part}} \leftarrow q_{\text{part}} + \alpha p_{\text{part}}$
  $\text{pq} \leftarrow (p_{\text{part}}, q_{\text{part}})$
  $\text{pq} \leftarrow$ `comm_row.Allreduce`(pq)
  $D^q_{\text{part}} \leftarrow D^q_{\text{part}} + \alpha^2 p^{\circ 2}_{\text{part}}$
  $D^{\text{pq}} \leftarrow (p^{\circ 2}_{\text{part}}, D^q_{\text{part}})$
  $D^{\text{pq}} \leftarrow$ `comm_row.Allreduce`(D^{pq})
  $$x_{\text{part}} \leftarrow x_{\text{part}} - \frac{p_{\text{part}}}{\text{pq}}$$
  $s \leftarrow s + 1$
end

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

Листинг 2. Реализация алгоритма 2 на языке Python

Listing 2. Implementation of the algorithm 2 in Python

```
def iCG(A_part, b_part, alpha, comm_row, comm_col):
    M_part, N_part = np.shape(A_part)
    delta = np.finfo(np.float64).eps
    s = 1

    rr, Ds, pq, Dpq = [np.array(0, np.float64) for _ in range(4)]
    r_part, Dr_part, q_part, Dq_part = \
        [np.empty(N_part, np.float64) for _ in range(4)]
    t_part, Dt_part = [np.empty(M_part, np.float64) for _ in range(2)]

    x_part = np.zeros(N_part, np.float64)
    p_part = np.zeros(N_part, np.float64)
    A2_part = A_part**2

    while True:
        if s == 1:
            np.dot(A_part, x_part, out=t_part)
            np.dot(A2_part, x_part**2, out=Dt_part)

            allreduce(comm_row, t_part, M_part)
            allreduce(comm_row, Dt_part, M_part)

            t_part -= b_part
            Dt_part += b_part**2
            np.dot(A_part.T, t_part, out=r_part)
            np.dot(A2_part.T, Dt_part, out=Dr_part)

            allreduce(comm_col, r_part, N_part)
            allreduce(comm_col, Dr_part, N_part)

            r_part += alpha * x_part
            Dr_part += alpha**2 * x_part**2
        else:
            r_part -= q_part / pq
            Dr_part += (pq**2 * Dq_part
                        - (2 * pq * p_part * q_part * Dq_part)
                        + Dpq * q_part**2) / pq**4

        np.dot(r_part, r_part, out=rr)
        np.sum(Dr_part, out=Ds)

        allreduce(comm_row, rr, 1)
        allreduce(comm_row, Ds, 1)

        if delta**2 * Ds / rr >= 1:
            return x_part, s

        p_part += r_part / rr

        np.dot(A_part, p_part, out=t_part)
        np.dot(A2_part, p_part**2, out=Dt_part)

        allreduce(comm_row, t_part, M_part)
        allreduce(comm_row, Dt_part, M_part)
```

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

```
55          np.dot(A_part.T, t_part, out=q_part)
56          np.dot(A2_part.T, Dt_part, out=Dq_part)
57
58          allreduce(comm_col, q_part, N_part)
59          allreduce(comm_col, Dq_part, N_part)
60
61          q_part += alpha * p_part
62          Dq_part += alpha**2 * p_part**2
63
64          np.dot(p_part, q_part, out=pq)
65          np.dot(p_part**2, Dq_part, out=Dpq)
66
67          allreduce(comm_row, pq, 1)
68          allreduce(comm_row, Dpq, 1)
69
70          x_part -= p_part / pq
71          s += 1
```

Аналогично последовательной версии программы по итогу работы этой функции каждый MPI-процесс возвращает: 1) массив `x_part`, который содержит часть решения системы (4), найденное по усовершенствованному алгоритму, и который при объединении с аналогичными данными с остальных процессов даст массив `x`, содержащий полное решение x^{α} системы (4); 2) число `s` итераций, которые потребовалось сделать алгоритму, чтобы найти приближенное решение.

4.2. Учет возможностей стандарта MPI-3. При использовании функции `Allreduce()` из стандарта MPI-2 этапы вычислений и этапы пересылки данных промежуточных вычислений между различными вычислительными процессами четко разделены во времени. При использовании неблокирующей (асинхронной) функции `Iallreduce()` из стандарта MPI-3 появляется возможность хотя бы частично совместить во времени некоторые этапы вычислений и пересылки данных. Особенности этой неблокирующей функции используются следующим образом. Функционально (в смысле результата) работа функции `Allreduce()` из стандарта MPI-2 эквивалентна последовательности запуска функций `Iallreduce()` "+" `wait()` из стандарта MPI-3. Иногда результат обмена сообщениями между вычислительными узлами нужен для последующих вычислений не сразу же после вызова функции `Iallreduce()`. В таких случаях имеет смысл приступить к вычислениям, которые не используют результат работы функции `Iallreduce()`, и только затем провести вычисления с использованием полученных данных промежуточных вычислений с других вычислительных процессов, исходя из предположения, что обмен данными уже завершен. Перед проведением соответствующих вычислений необходимо вызвать функцию `wait()`, чтобы убедиться в завершении обмена сообщениями.

За счет применения описанного подхода возможно значительное уменьшение накладных расходов по приему и передаче сообщений, содержащих результаты промежуточных расчетов (см., например, [21]). Для рассматриваемого алгоритма на каждой итерации градиентного метода поиск очередного приближения для решения системы (4) и оценка накопленных ошибок машинного округления могут выполняться независимо друг от друга. Это дает возможность эффективно использовать неблокирующую (асинхронную) MPI-функцию `Iallreduce()` в связке с MPI-функцией `wait()`.

Подход к распараллеливанию, основанный на использовании стандарта MPI-3, представлен в виде алгоритма 3. При его программной реализации (листинг 3), по аналогии с функцией `allreduce()`, введенной в конце подраздела 3.3, будет удобно использовать функцию `iallreduce()` вида

```
def iallreduce(comm, part, size):
    return comm.Iallreduce(MPI.IN_PLACE,
                           [part, size, MPI.DOUBLE], op=MPI.SUM)
```

а также функцию `wait()`, которая позволят дожидаться окончания пересылок результатов промежуточных расчетов:

```
def wait(reqs):
    MPI.Request.Waitall(reqs)
```

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

Алгоритм 3. Псевдокод для эффективной параллельной версии алгоритма iCG

Algorithm 3. Pseudocode for an efficient parallel version of the iCG algorithm

Data: A_{part}, b_{part}, α, `comm_row`, `comm_col`
Результат: $x_{\text{part}} \equiv x^{\alpha}_{\text{part}}$
$x_{\text{part}} \leftarrow 0$; $s \leftarrow 1$; $p_{\text{part}} \leftarrow 0$
while *True* **do**
  if $s = 1$ **then**
    $t_{\text{part}} \leftarrow A_{\text{part}} x_{\text{part}}$
    $t_{\text{part}} \leftarrow$ `comm_row.Allreduce`(t_{part})
    $D^t_{\text{part}} \leftarrow A^{\circ 2}_{\text{part}} x^{\circ 2}_{\text{part}}$
    $D^t_{\text{part}} \leftarrow$ `comm_row.Allreduce`(D^t_{part})
    $t_{\text{part}} \leftarrow t_{\text{part}} - b_{\text{part}}$; $r_{\text{part}} \leftarrow A^T_{\text{part}} t_{\text{part}}$
    $r_{\text{part}} \leftarrow$ `comm_col.Allreduce`(r_{part})
    $D^t_{\text{part}} \leftarrow D^t_{\text{part}} + b^{\circ 2}_{\text{part}}$
    $D^r_{\text{part}} \leftarrow (A^T_{\text{part}})^{\circ 2} D^t_{\text{part}}$; $D^r_{\text{part}} \leftarrow$ `comm_col.Allreduce`(D^r_{part})
    $r_{\text{part}} \leftarrow r_{\text{part}} + \alpha x_{\text{part}}$; $D^r_{\text{part}} \leftarrow D^r_{\text{part}} + \alpha^2 x^{\circ 2}_{\text{part}}$
  else
    $r_{\text{part}} \leftarrow r_{\text{part}} - \dfrac{q_{\text{part}}}{\text{pq}}$
    $D^{\text{pq}} \leftarrow$ `Request.Wait`$(\text{req}_{D^{\text{pq}}})$
    $D^r_{\text{part}} \leftarrow D^r_{\text{part}} + \dfrac{\text{pq}^2 D^q_{\text{part}} - 2 \cdot \text{pq} \cdot p_{\text{part}} \circ q_{\text{part}} \circ D^q_{\text{part}} + q^{\circ 2}_{\text{part}} D^{\text{pq}}}{\text{pq}^4}$
  end
  $\text{rr} \leftarrow (r_{\text{part}}, r_{\text{part}})$
  $\text{req}_{\text{rr}} \leftarrow$ `comm_row.Iallreduce`(rr)
  $D^s \leftarrow \sum\limits_n (D^r_{\text{part}})_n$
  $\text{req}_{D^s} \leftarrow$ `comm_row.Iallreduce`(D^s)
  $\text{rr}, D^s \leftarrow$ `Request.Wait`$(\text{req}_{\text{rr}}, \text{req}_{D^s})$
  if $\dfrac{\Delta^2 D^s}{\text{rr}} \geqslant 1$ **then**
    return x_{part}, s
  end
  $p_{\text{part}} \leftarrow p_{\text{part}} + \dfrac{r_{\text{part}}}{\text{rr}}$
  $t_{\text{part}} \leftarrow A_{\text{part}} p_{\text{part}}$
  $\text{req}_t \leftarrow$ `comm_row.Iallreduce`(t_{part})
  $D^t_{\text{part}} \leftarrow A^{\circ 2}_{\text{part}} p^{\circ 2}_{\text{part}}$
  $\text{req}_{D^t} \leftarrow$ `comm_row.Iallreduce`(D^t_{part})
  $t_{\text{part}} \leftarrow$ `Request.Wait`(req_t)
  $q_{\text{part}} \leftarrow A^T_{\text{part}} t_{\text{part}}$
  $\text{req}_q \leftarrow$ `comm_col.Iallreduce`(q_{part})
  $D^t_{\text{part}} \leftarrow$ `Request.Wait`(req_{D^t})
  $D^q_{\text{part}} \leftarrow (A^T_{\text{part}})^{\circ 2} D^t_{\text{part}}$
  $\text{req}_{D^q} \leftarrow$ `comm_col.Iallreduce`(D^q_{part})
  $q_{\text{part}} \leftarrow$ `Request.Wait`(req_q)
  $q_{\text{part}} \leftarrow q_{\text{part}} + \alpha p_{\text{part}}$
  $\text{pq} \leftarrow (p_{\text{part}}, q_{\text{part}})$
  $\text{req}_{\text{pq}} \leftarrow$ `comm_row.Iallreduce`(pq)
  $D^q_{\text{part}} \leftarrow$ `Request.Wait`(req_{D^q})
  $D^q_{\text{part}} \leftarrow D^q_{\text{part}} + \alpha^2 p^{\circ 2}_{\text{part}}$
  $D^{\text{pq}} \leftarrow (p^{\circ 2}_{\text{part}}, D^q_{\text{part}})$
  $\text{req}_{D^{\text{pq}}} \leftarrow$ `comm_row.Iallreduce`(D^{pq})
  $\text{pq} \leftarrow$ `Request.Wait`(req_{pq})
  $x_{\text{part}} \leftarrow x_{\text{part}} - \dfrac{p_{\text{part}}}{\text{pq}}$; $s \leftarrow s + 1$
end

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

Листинг 3. Реализация алгоритма 3 на языке Python

Listing 3. Implementation of the algorithm 3 in Python

```python
def iCG(A_part, b_part, alpha, comm_row, comm_col):
    M_part, N_part = np.shape(A_part)
    delta = np.finfo(np.float64).eps
    s = 1

    rr, Ds, pq, Dpq = [np.array(0, np.float64) for _ in range(4)]
    r_part, Dr_part, q_part, Dq_part = \
        [np.empty(N_part, np.float64) for _ in range(4)]
    t_part, Dt_part = [np.empty(M_part, np.float64) for _ in range(2)]

    x_part = np.zeros(N_part, np.float64)
    p_part = np.zeros(N_part, np.float64)
    A2_part = A_part**2

    while True:
        if s == 1:
            np.dot(A_part, x_part, out=t_part)
            np.dot(A2_part, x_part**2, out=Dt_part)

            allreduce(comm_row, t_part, M_part)
            allreduce(comm_row, Dt_part, M_part)

            t_part -= b_part
            Dt_part += b_part**2
            np.dot(A_part.T, t_part, out=r_part)
            np.dot(A2_part.T, Dt_part, out=Dr_part)

            allreduce(comm_col, r_part, N_part)
            allreduce(comm_col, Dr_part, N_part)

            r_part += alpha * x_part
            Dr_part += alpha**2 * x_part**2
        else:
            r_part -= q_part / pq
            Dr_part += (pq**2 * Dq_part
                        - (2 * pq * p_part * q_part * Dq_part)
                        + Dpq * q_part**2) / pq**4

        np.dot(r_part, r_part, out=rr)
        np.sum(Dr_part, out=Ds)

        req_rr = iallreduce(comm_row, rr, 1)
        req_Ds = iallreduce(comm_row, Ds, 1)
        wait([req_rr, req_Ds])

        if delta**2 * Ds / rr >= 1:
            return x_part, s

        p_part += r_part / rr

        np.dot(A_part, p_part, out=t_part)
        np.dot(A2_part, p_part**2, out=Dt_part)

        req_t = iallreduce(comm_row, t_part, M_part)
```

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

```
55          req_Dt = iallreduce(comm_row, Dt_part, M_part)
56          wait([req_t, req_Dt])
57
58          np.dot(A_part.T, t_part, out=q_part)
59          np.dot(A2_part.T, Dt_part, out=Dq_part)
60
61          req_q = iallreduce(comm_col, q_part, N_part)
62          req_Dq = iallreduce(comm_col, Dq_part, N_part)
63          wait([req_q, req_Dq])
64
65          q_part += alpha * p_part
66          Dq_part += alpha**2 * p_part**2
67
68          np.dot(p_part, q_part, out=pq)
69          np.dot(p_part**2, Dq_part, out=Dpq)
70
71          req_pq = iallreduce(comm_row, pq, 1)
72          req_Dpq = iallreduce(comm_row, Dpq, 1)
73          wait([req_pq, req_Dpq])
74
75          x_part -= p_part / pq
76          s += 1
```

4.3. Учет возможностей стандарта MPI-4. Функционально (в смысле результата) работа функции `Allreduce()` из стандарта MPI-2 эквивалентна последовательности запуска функций `Allreduce_init()` “+” `start()` “+” `wait()` из стандарта MPI-4. С учетом того, что работа функции `Allreduce()` внутри цикла `while` реализуется над одними и теми же областями адресного пространства, одинаковую операцию `Allreduce_init()` можно вынести вне цикла `while`. За счет этого могут значительно уменьшиться накладные расходы по приему и передаче сообщений, содержащих результаты промежуточных расчетов.

Изменения в программной реализации будут достаточно простыми. Необходимо выполнить следующую последовательность изменений.

1. До основного цикла `while` необходимо сформировать отложенные запросы на взаимодействия с помощью MPI-функций вида `req = Allreduce_init()` для всех функций коллективного взаимодействия процессов `Allreduce()` и `Iallreduce()`, которые встречаются внутри цикла.

   Замечание. При инициализации сформированных запросов на взаимодействие с помощью MPI-функции `start()` данные будут считываться/записываться в области памяти, которые фиксируются один раз при формировании отложенного запроса на взаимодействие с помощью `Allreduce_init()`. Поэтому необходимо предварительно выделить место в памяти под все массивы, которые являются аргументами этих функций, а при последующих вычислениях следить за тем, чтобы соответствующие результаты вычислений сохранялись в правильных областях памяти.
2. Внутри цикла `while` заменить каждую MPI-функцию `Allreduce()` на последовательность MPI-функций `Start(req)` “+” `Wait(req)`.
3. Внутри цикла `while` заменить каждую MPI-функцию `Iallreduce()` на MPI-функцию `Start(req)`.

   Для удобства выделим эти операции во вспомогательные функции.

```
def allreduce_init(comm, part, size):
    return comm.Allreduce_init(MPI.IN_PLACE,
                               [part, size, MPI.DOUBLE], op=MPI.SUM)
def start_and_wait(reqs):
    MPI.Prequest.Startall(reqs)
    MPI.Request.Waitall(reqs)
```

Соответствующая параллельная реализация алгоритма 3 для решения системы (4) с учетом описанных изменений показана в листинге 4.

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

Листинг 4. Модифицированная реализация алгоритма 3 на языке Python

Listing 4. Modified implementation of the algorithm 3 in Python

```
def iCG(A_part, b_part, alpha, comm_row, comm_col):
    M_part, N_part = np.shape(A_part)
    delta = np.finfo(np.float64).eps
    s = 1

    rr, Ds, pq, Dpq = [np.array(0, np.float64) for _ in range(4)]
    r_part, Dr_part, q_part, Dq_part = \
        [np.empty(N_part, np.float64) for _ in range(4)]
    t_part, Dt_part = [np.empty(M_part, np.float64) for _ in range(2)]

    x_part = np.zeros(N_part, np.float64)
    p_part = np.zeros(N_part, np.float64)
    A2_part = A_part**2

    req_rr = allreduce_init(comm_row, rr, 1)
    req_Ds = allreduce_init(comm_row, Ds, 1)
    req_pq = allreduce_init(comm_row, pq, 1)
    req_Dpq = allreduce_init(comm_row, Dpq, 1)

    req_t = allreduce_init(comm_row, t_part, M_part)
    req_Dt = allreduce_init(comm_row, Dt_part, M_part)
    req_q = allreduce_init(comm_col, q_part, N_part)
    req_Dq = allreduce_init(comm_col, Dq_part, N_part)

    while True:
        if s == 1:
            np.dot(A_part, x_part, out=t_part)
            np.dot(A2_part, x_part**2, out=Dt_part)

            allreduce(comm_row, t_part, M_part)
            allreduce(comm_row, Dt_part, M_part)

            t_part -= b_part
            Dt_part += b_part**2
            np.dot(A_part.T, t_part, out=r_part)
            np.dot(A2_part.T, Dt_part, out=Dr_part)

            allreduce(comm_col, r_part, N_part)
            allreduce(comm_col, Dr_part, N_part)

            r_part += alpha * x_part
            Dr_part += alpha**2 * x_part**2
        else:
            r_part -= q_part / pq
            Dr_part += (pq**2 * Dq_part
                        - (2 * pq * p_part * q_part * Dq_part)
                        + Dpq * q_part**2) / pq**4

        np.dot(r_part, r_part, out=rr)
        np.sum(Dr_part, out=Ds)

        start_and_wait([req_rr, req_Ds])

        if delta**2 * Ds / rr >= 1:
```

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

```
            return x_part, s

        p_part += r_part / rr

        np.dot(A_part, p_part, out=t_part)
        np.dot(A2_part, p_part**2, out=Dt_part)

        start_and_wait([req_t, req_Dt])

        np.dot(A_part.T, t_part, out=q_part)
        np.dot(A2_part.T, Dt_part, out=Dq_part)

        start_and_wait([req_q, req_Dq])

        q_part += alpha * p_part
        Dq_part += alpha**2 * p_part**2

        np.dot(p_part, q_part, out=pq)
        np.dot(p_part**2, Dq_part, out=Dpq)

        start_and_wait([req_pq, req_Dpq])

        x_part -= p_part / pq
        s += 1
```

5. Поиск параметра регуляризации из обобщенного принципа невязки. Поиск параметра регуляризации α^* из обобщенного принципа невязки (3) заключается в решении уравнения

$$\rho(\alpha) \equiv \|A_h x^\alpha - b_\delta\|^2 - \left(\delta + h\|x^\alpha\|\right)^2 - \mu^2 = 0.$$

Здесь необходимо напомнить, что μ — мера несовместности приближенных данных, которая вычисляется по формуле

$$\mu = \|A_h x^0 - b_\delta\|,$$

где x^0 — результат работы предложенного алгоритма 1 при $\alpha = 0$.

Для решения этого уравнения конструктивно использовать метод хорд, программная реализация которого приведена далее.

```
def secant(f, a, b, tol=1e-17, max_iter=1000):
    fa = f(a)
    fb = f(b)
    if abs(fa) < tol:
        return a
    if abs(fb) < tol:
        return b

    for _ in range(max_iter):
        if fb - fa == 0:
            return b
        c = b - fb * (b - a) / (fb - fa)
        fc = f(c)
        if abs(fc) < tol:
            return c
        a, fa = b, fb
        b, fb = c, fc
    raise RuntimeError('Value was not found')
```

Одним из условий применимости метода хорд является монотонность функции $\rho(\alpha)$ и наличие двух начальных приближений для искомого корня α^*, при которых функция $\rho(\alpha)$ принимает значения раз-

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

личных знаков. Для поиска таких начальных приближений можно воспользоваться следующим приемом, исходя из того, что согласно теории функция $\rho(\alpha)$ является монотонно возрастающей [2].

1. Выбирается произвольное начальное приближение для α^*, например, $\alpha = 1$.
2. Вычисляется значение $\rho(\alpha)$. Если $\rho(\alpha) > 0$, выполняется переопределение значения α: $\alpha := \alpha/2$. В противном случае при $\rho(\alpha) < 0$: $\alpha := 2\alpha$.
3. Если знак $\rho(\alpha)$ не поменялся, осуществляется переход на шаг 2.
4. В качестве начальных приближений выбираются два последних значения α из построенной на предыдущих шагах алгоритма последовательности значений α.

Программная реализация такого алгоритма имеет следующий вид.

```
def init_approx(rho, alpha=1):
    if rho(alpha) > 0:
        alpha /= 2
        while rho(alpha) >= 0:
            alpha /= 2
        return alpha, alpha * 2
    else:
        alpha *= 2
        while rho(alpha) <= 0:
            alpha *= 2
        return alpha / 2, alpha
```

Используя две введенные вспомогательные функции `secant()` и `init_approx()`, можно построить алгоритм поиска приближенного значения α^*, программная реализация которого будет иметь следующий вид.

```
def find_optimal_alpha(A_part, b_part, delta2, h,
                       M_part, comm_row, comm_col):
    x_part, _ = cg(A_part, b_part, 0, comm_row, comm_col)

    t_part = np.dot(A_part, x_part)
    allreduce(comm_row, t_part, M_part)

    mu2 = np.array(np.linalg.norm(t_part - b_part) ** 2)
    allreduce(comm_col, mu2, 1)

    def rho(alpha):
        x_part, _ = cg(A_part, b_part, alpha, comm_row, comm_col)

        norm2 = np.array(np.linalg.norm(x_part)**2)
        allreduce(comm_row, norm2, 1)

        t_part = np.dot(A_part, x_part)
        allreduce(comm_row, t_part, M_part)

        D2 = np.array(np.linalg.norm((t_part - b_part))**2)
        allreduce(comm_col, D2, 1)
        return D2 - (delta2**0.5 + h * norm2**0.5)**2 - mu2

    alpha_left, alpha_right = init_approx(rho, 1)

    return secant(rho, alpha_left, alpha_right)
```

Обратим внимание на некоторые особенности этой программной реализации.

1. Значение `mu2` считается по векторам размера M, а значит, коллективные MPI-операции должны осуществляться среди MPI-процессов, принадлежащих коммуникаторам `comm_col`.
2. Имя переменной `_` обычно используется, чтобы показать, что мы дальше не используем результат этой переменной.
3. Так как вычисления по объему пренебрежимо малы по сравнению с вызовом метода сопряженных градиентов, использование возможностей MPI-3 или MPI-4 не дает существенного преимущества.

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

6. Описание тестового примера. В качестве тестового примера рассмотрим трехмерную обратную задачу электростатики. Пусть $\rho(\boldsymbol{r})$ — объемная плотность электрического заряда в области $V \subset \mathbb{R}^3$, а $\boldsymbol{E}(\boldsymbol{r}_s)$ — напряженность электрического поля, создаваемого этим распределением в точке измерений $\boldsymbol{r}_s \notin V$. Требуется определить функцию $\rho(\boldsymbol{r})$, $\boldsymbol{r} \in V$, по измеренным значениям компонент вектора $\boldsymbol{E}(\boldsymbol{r}_s)$ в некотором наборе точек.

Математическая постановка имеет вид интегрального уравнения Фредгольма первого рода

$$\boldsymbol{E}(\boldsymbol{r}_s) = \iiint\limits_V \frac{(\boldsymbol{r}_s - \boldsymbol{r})\,\rho(\boldsymbol{r})}{|\boldsymbol{r}_s - \boldsymbol{r}|^3}\,dV. \tag{9}$$

Данная задача является некорректно поставленной: ее решение может не существовать, быть не единственным или неустойчивым к погрешностям входных данных. Кроме того, задача физически переопределена: по трем скалярным компонентам вектора $\boldsymbol{E} = (E_x, E_y, E_z)^T$ необходимо восстановить лишь одну скалярную функцию ρ.

Для упрощения численного моделирования будем считать, что все заряды сосредоточены на тонкой прямолинейной струне единичной площади поперечного сечения ($S = 1$), расположенной вдоль оси Ox на отрезке $[a, b]$. Предположим также, что измерения компонент вектора $\boldsymbol{E}$ проводятся на отрезке, параллельном оси Ox, с концами в точках (c, l_y, l_z) и (d, l_y, l_z). При таких ограничениях задача становится одномерной: требуется найти функцию $\rho(x) \equiv \rho(x, 0, 0)$, $x \in [a, b]$, по значениям $\boldsymbol{E}(x_s) \equiv \boldsymbol{E}(x_s, l_y, l_z)$, $x_s \in [c, d]$.

Уравнение (9) после интегрирования по поперечному сечению (которое дает множитель, равный единице) преобразуется к виду

$$\begin{pmatrix} E_x \\ E_y \\ E_z \end{pmatrix}(x_s) = \int\limits_a^b \left[\begin{pmatrix} x_s \\ l_y \\ l_z \end{pmatrix} - \begin{pmatrix} x \\ 0 \\ 0 \end{pmatrix} \right] \frac{\rho(x)}{\big((x_s - x)^2 + l_y^2 + l_z^2\big)^{3/2}}\,dx. \tag{10}$$

Для удобства дальнейших выкладок обозначим координату измерений через $s \equiv x_s$ и перепишем векторное уравнение (10) в виде системы трех одномерных интегральных уравнений:

$$\begin{cases} E_x(s) = \displaystyle\int\limits_a^b \frac{(s - x)\,\rho(x)}{\big((s - x)^2 + l_y^2 + l_z^2\big)^{3/2}}\,dx, \\ E_y(s) = \displaystyle\int\limits_a^b \frac{l_y\,\rho(x)}{\big((s - x)^2 + l_y^2 + l_z^2\big)^{3/2}}\,dx, \\ E_z(s) = \displaystyle\int\limits_a^b \frac{l_z\,\rho(x)}{\big((s - x)^2 + l_y^2 + l_z^2\big)^{3/2}}\,dx. \end{cases} \tag{11}$$

Перейдем к конечно-разностной аппроксимации. Введем на отрезке $[a, b]$ равномерную сетку $X_N = \{x_n :\ x_n = a + nh,\ n = \overline{0, N}\}$, $h = \dfrac{b - a}{N}$. Искомые сеточные значения плотности обозначим $\rho_n \equiv \rho(x_n)$. Аппроксимируя интегралы в (11) по формуле трапеций, получим:

$$\begin{cases} E_x(s) = \dfrac{(s - x_0)h}{2\big((s - x_0)^2 + l_y^2 + l_z^2\big)^{3/2}}\rho_0 + \displaystyle\sum_{n=1}^{N-1} \frac{(s - x_n)h}{\big((s - x_n)^2 + l_y^2 + l_z^2\big)^{3/2}}\rho_n + \frac{(s - x_N)h}{2\big((s - x_N)^2 + l_y^2 + l_z^2\big)^{3/2}}\rho_N, \\ E_y(s) = \dfrac{l_y h}{2\big((s - x_0)^2 + l_y^2 + l_z^2\big)^{3/2}}\rho_0 + \displaystyle\sum_{n=1}^{N-1} \frac{l_y h}{\big((s - x_n)^2 + l_y^2 + l_z^2\big)^{3/2}}\rho_n + \frac{l_y h}{2\big((s - x_N)^2 + l_y^2 + l_z^2\big)^{3/2}}\rho_N, \\ E_z(s) = \dfrac{l_z h}{2\big((s - x_0)^2 + l_y^2 + l_z^2\big)^{3/2}}\rho_0 + \displaystyle\sum_{n=1}^{N-1} \frac{l_z h}{\big((s - x_n)^2 + l_y^2 + l_z^2\big)^{3/2}}\rho_n + \frac{l_z h}{2\big((s - x_N)^2 + l_y^2 + l_z^2\big)^{3/2}}\rho_N. \end{cases} \tag{12}$$

Пусть измерения проводятся в N_s точках s_m, $m = \overline{1, N_s}$, образующих равномерную сетку на отрезке $[c, d]$: $s_m = c + (m - 1)h_s$, $h_s = \dfrac{d - c}{N_s - 1}$. Записывая систему (12) для каждого s_m, $m = \overline{1, N_s}$, получим

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

переопределенную СЛАУ с плотно заполненной матрицей размера $M \times (N-1) \equiv (3 \cdot N_s) \times (N-1)$:

$$AX = B. \tag{13}$$

Здесь $X \equiv (\rho_0\ \rho_1\ \ldots\ \rho_N)^T$ — вектор, состоящий из искомых сеточных значений функции $\rho(x)$ в узлах сетки X_N, а вектор $B \equiv \big(E_x(s_1)\ E_y(s_1)\ E_z(s_1)\ \ldots\ E_x(s_{N_s})\ E_y(s_{N_s})\ E_z(s_{N_s})\big)^T$ составлен из “измеренных” (в вычислительном эксперименте) компонент напряженности поля.

Программная реализация решения модельной задачи для описанного тестового примера примет следующий вид. Вначале импортируем необходимые модули.

```
from contextlib import contextmanager
from mpi4py import MPI
import numpy as np
import matplotlib.pyplot as plt
```

Далее определим вспомогательную функцию для подготовки двумерной декартовой топологии типа двумерного тора в виде коммуникатора `comm_cart`, посредством которого будут происходить взаимодействия между MPI-процессами. Также создадим на основе этого коммуникатора вспомогательные коммуникаторы `comm_col` и `comm_row` (см. подробные пояснения в подразделе 3.3).

```
def prepare_comms():
    comm = MPI.COMM_WORLD
    num_row = num_col = np.int32(np.sqrt(comm.Get_size()))
    comm_cart = comm.Create_cart(dims=(num_row, num_col),
                                 periods=(True, True), reorder=True)

    rank_cart = comm_cart.Get_rank()

    comm_col = comm_cart.Split(rank_cart % num_col, rank_cart)
    comm_row = comm_cart.Split(rank_cart // num_col, rank_cart)
    return comm_cart, comm_col, comm_row, num_row, num_col, rank_cart
```

Чтобы сопоставить индексы всего вектора размера `len`, и его локальных частей на группе процессов длины `num`, напишем вспомогательную функцию. Результатом будут массив локальных длин `counts`, массив абсолютных индексов нулевых элементов `displs` и их значения для конкретного процесса с индексом (идентификатором) `local`.

```
def calculate_sizes(len, num, local):
    ave, res = divmod(len, num)
    counts = np.empty(num, np.int32)
    displs = np.empty(num, np.int32)
    for k in range(0, num):
        counts[k] = ave + 1 if k < res else ave
        displs[k] = 0 if k == 0 else displs[k - 1] + counts[k - 1]
    return counts, displs, counts[local], displs[local]
```

Также напишем вспомогательную функцию для сбора распределенного вектора на нулевом процессе.

```
def gather(part, comm_row,
           rank_cart, num_col, len, len_part,
           counts, displs):
    dst = None
    if rank_cart == 0:
        dst = np.empty(len, np.float64)

    if rank_cart in range(num_col):
        comm_row.Gatherv([part, len_part, MPI.DOUBLE],
                         [dst, counts, displs, MPI.DOUBLE], root=0)
    return dst
```

Для измерения времени будем использовать средства MPI, но для простоты запишем их в виде декоратора. Синтаксис, представленный ниже, предполагает, что `measure_time` будет использовано вместе с ключевым словом `with`, причем вначале будет исполнен код внутри `measure_time` до ключевого слова `yield`, затем тело блока `with`, а затем код внутри `measure_time` после `yield`.

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

```
@contextmanager
def measure_time(comm):
    comm.Barrier()
    time_took = np.array(MPI.Wtime(), np.float64)
    yield time_took
    time_took[None] = MPI.Wtime() - time_took
    comm.Allreduce(MPI.IN_PLACE, [time_took, 1, MPI.DOUBLE], op=MPI.MAX)
```

Следующие функции описывают генерацию матрицы, приведенной в СЛАУ (13), а также модельный вектор значений плотности заряда.

```
def prepare_matrix(N_part, M_part,
                   xc_part, xc_step, xc_min, xc_max,
                   xs_part, l_y, l_z):
    A_part = np.ones((M_part, N_part), np.float64)
    if xc_part[0] == xc_min:
        A_part[:, 0] = 0.5
    if xc_part[-1] == xc_max:
        A_part[:, -1] = 0.5
    for j, s in enumerate(xs_part):
        denom = ((s - xc_part)**2 + l_y**2 + l_z**2) ** 1.5
        A_part[3*j, :] *= (s - xc_part) * xc_step / denom
        A_part[3*j+1, :] *= l_y * xc_step / denom
        A_part[3*j+2, :] *= l_z * xc_step / denom
    return A_part

def density(x):
    return 2*np.exp(-(x - 0.382)**2/0.009) + \
                    1.2*np.exp(-(x - 0.618)**2/0.018)
```

Далее приводится код основной функции, в которой используются все вспомогательные функции, описанные выше: создается MPI-топология; заполняются векторы сеточных значений для зарядов и датчиков; по ним вычисляется модельная функция плотности заряда в виде набора сеточных значений этой функции на введенной сетке; вычисляется матрица оператора задачи; вычисляется модельная правая часть b, к которой добавляется некоторая ошибка, симулирующая ошибки экспериментальных наблюдений. Для решения задачи вначале находится значение параметра регуляризации α^* из обобщенного принципа невязки (3), а затем с его помощью вычисляется регуляризированное решение x^{α^*} задачи (1). В конце работы метода вычисляется погрешность найденного решения относительно известного модельного решения, а также строится график решения.

```
def main(Ns = 100, Nc = 199):
    xc_min, xc_max = 0, 1
    xs_min, xs_max = 0.2, 1
    l_y, l_z = 0.2, 0.8

    comm_cart,comm_col,comm_row,num_row,num_col,rank_cart = prepare_comms()
    my_row, my_col = comm_cart.Get_coords(rank_cart)

    N = Nc + 1
    counts_N, displs_N, N_part, N_displ = calculate_sizes(N,num_col,my_col)
    *_, Ns_part, Ns_displ = calculate_sizes(Ns, num_row, my_row)

    M = 3 * Ns; M_part = 3 * Ns_part; M_displ = 3 * Ns_displ

    xc, xc_step = np.linspace(xc_min, xc_max, N, retstep=True)
    xc_part = xc[N_displ : N_displ + N_part]

    xs = np.linspace(xs_min, xs_max, Ns)
    xs_part = xs[Ns_displ : Ns_displ + Ns_part]

    A_part = prepare_matrix(N_part, M_part,
```

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

```
                            xc_part, xc_step, xc_min, xc_max,
                            xs_part, l_y, l_z)

x_model = density(xc)
b_part = np.dot(A_part, x_model[N_displ : N_displ + N_part])
allreduce(comm_row, b_part, M_part)

rng = np.random.default_rng(0)
delta_b =  1e-8 * rng.uniform(-0.5, 0.5, size=M)

b_part += delta_b[M_displ : M_displ + M_part]
delta2 = np.linalg.norm(delta_b) ** 2

with measure_time(comm_cart) as time:
    alpha = find_optimal_alpha(A_part, b_part, delta2, 0,
                               M_part, comm_row, comm_col)

    x_part, s = iCG(A_part, b_part, alpha, comm_row, comm_col)

x = gather(x_part, comm_row,
           rank_cart, num_col, N, N_part,
           counts_N, displs_N)

if rank_cart == 0:
    print('s = {}, time = {}, alpha = {}, |diff|/|x|*100% = {}'.format(
    s, time, alpha,
    np.linalg.norm(x - x_model) / np.linalg.norm(x_model) * 100))

    plt.xlabel('i')
    plt.ylabel('x[i]')
    plt.plot(np.arange(N), x_model, '--', color='silver', lw=2)
    plt.plot(np.arange(N), x, 'C0', lw=2)
    plt.show()
```

7. Оценка эффективности и масштабируемости программной реализации параллельного алгоритма. В этом разделе обсуждается эффективность и масштабируемость предложенной параллельной программной реализации алгоритма 1. Сначала приводится описание тестового примера и использованной для тестовых вычислений многопроцессорной системы, а затем обсуждаются результаты исследования предложенных программных реализаций.

Для тестирования эффективности программных реализаций параллельного алгоритма использовался вычислительный раздел “compute” суперкомпьютера “Ломоносов-2” [1] Научно-исследовательского вычислительного центра МГУ имени М. В. Ломоносова. Каждый вычислительный узел раздела “compute” содержит 14-ядерный процессор Intel Xeon E5-2697 v3 2.60GHz с 64 GB оперативной памяти (4.5 GB на ядро) и видеокарту Tesla K40s с объемом видеопамяти 11.56 GB.

В качестве тестовой задачи была выбрана задача с матрицей A размера $M \times N$ с $M = 60\,000$, $N = 50\,000$. Выбор таких численных параметров обусловлен тем, что мы хотим провести расчеты для системы с как можно большей матрицей A, но при этом чтобы эта матрица помещалась в оперативную память одного вычислительного узла (в противном случае мы не сможем засечь время работы T_1 последовательной версии программы). В случае выбранных параметров для хранения матрицы системы A необходимо $N \times M \times 8$ байт $= 22.35$ GB. Такой же объем памяти необходим и для хранения транспонированной матрицы A^T (см. пояснения конструктивности такого подхода в конце раздела 2). Для оценки эффективности программных реализаций предложенного метода iCG при тестовых расчетах мы ограничились 100 итерациями метода сопряженных градиентов. В этом случае время работы T_1 последовательной версии функции `iCG()` без встроенного распараллеливания внутри `dot()` средствами `numpy` на одном вычислительном узле составляет порядка 10 мин (напомним, что по умолчанию функция `dot()` из пакета `numpy` использует все доступные ядра процессора на вычислительном узле для распараллеливания операций за счет использования технологии параллельного программирования OpenMP).

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

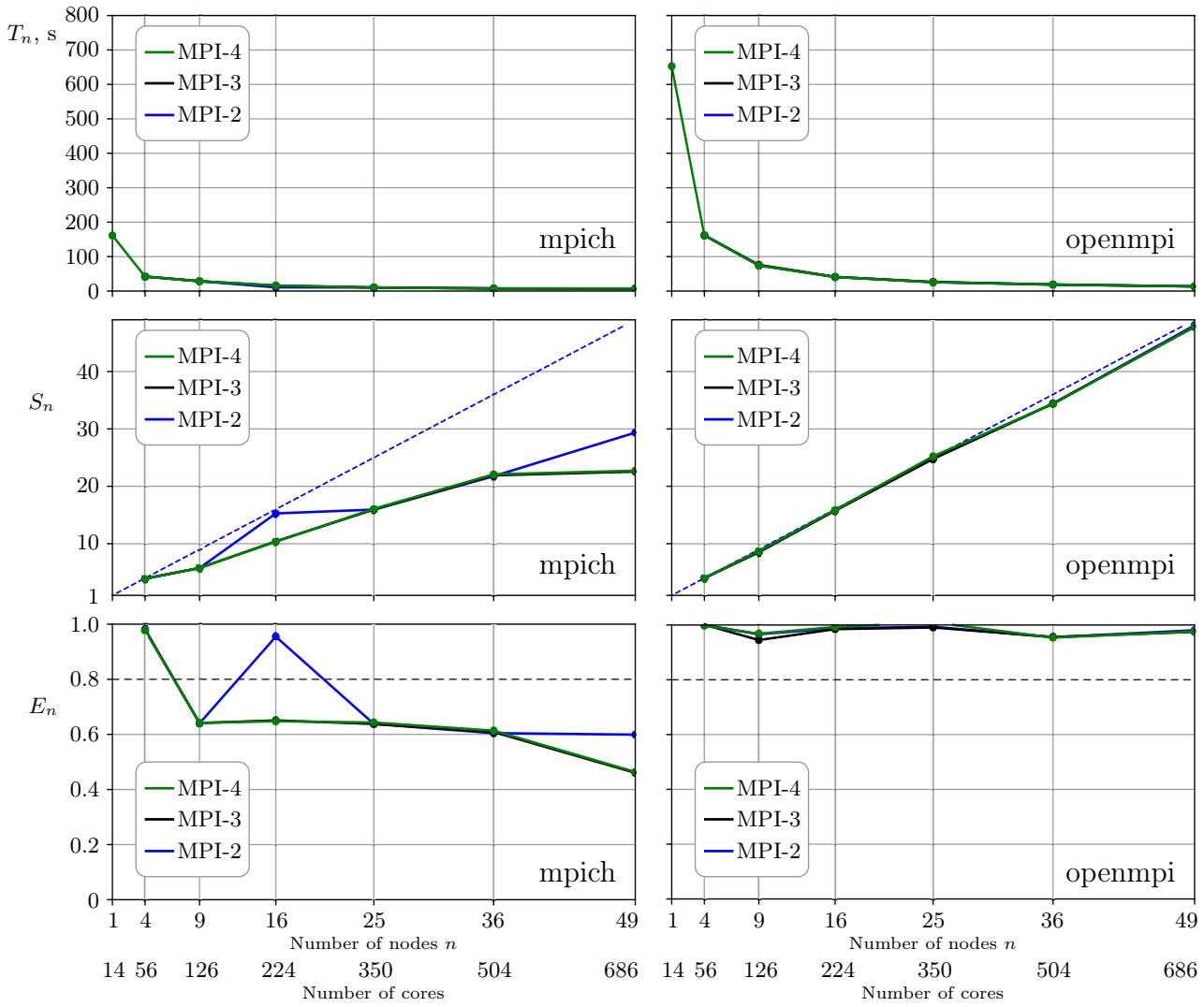

Рис. 4. Графики времени работы, ускорения и эффективности параллельной реализации алгоритма 1 в зависимости от количества узлов для различных реализаций (`mpich` и `openmpi`) стандартов MPI-2, MPI-3 и MPI-4 технологии параллельного программирования MPI

Fig. 4. Graphs of the running time, speedup and efficiency of the parallel implementation of algorithm 1 depending on the number of nodes for various implementations (`mpich` and `openmpi`) of the MPI-2, MPI-3, and MPI-4 standards of the MPI parallel programming technology

Использовались пакеты: 1) `numpy` версии 2.4.2, 2) `mpi4py` версии 4.1.1, 3) `openmpi` версии 5.0.10 или `mpich` версии 4.2.0.

Программа запускалась с привязкой каждого MPI-процесса ровно к одному вычислительному узлу на числе процессов n, которое является квадратом натурального числа (1, 4, 9, 16, 25, 36, 49 и т.д.). Это связано с тем, что в тестовых программах использовалась виртуальная топология процессов с двумерной декартовой сеткой с параметрами $\texttt{num\_row} = \texttt{num\_col} = \sqrt{n}$. Было засечено время работы T_n параллельной версии функции `iCG()` для каждого значения n из указанного числа процессов. Используя оценку времени работы T_1 последовательной версии функции `iCG()`, по формуле $S_n = \dfrac{T_1}{T_n}$ было вычислено ускорение S_n, а затем эффективность $E_n = \dfrac{S_n}{n}$ программной реализации. Вычисления проводились с помощью пакетов `mpich` и `openmpi`. Графики времени работы, ускорения и эффективности параллельной реализации алгоритма 1 в зависимости от количества узлов для различных реализаций (`mpich` и `openmpi`) стандартов MPI-2, MPI-3 и MPI-4 технологии параллельного программирования MPI представлены на рис. 4.

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

Отметим следующие особенности полученных результатов и программных реализаций.

1. Время работы программы с использованием MPI-реализации `openmpi` примерно в 4 раза больше, чем аналогичное время работы с использованием MPI-реализации `mpich`. Были проведены дополнительные эксперименты, которые показывают, что при параметрах запуска по умолчанию программа с использованием MPI-реализации `openmpi` не задействует дополнительные ядра для распараллеливания `dot()` с помощью технологии `OpenMP`. Результат аналогичен выключению распараллеливания внутри функции `dot()` при запуске с использованием MPI-реализации `mpich`.
2. Эффективность параллельных программ при запусках с использованием MPI-реализации `openmpi` в рамках использованных вычислительных ресурсов близка к 1, тогда как при запусках с использованием MPI-реализации `mpich` эффективность быстро падает до 60% и с увеличением числа узлов сохраняется постоянной (опять же, в рамках использованного для тестирования числа вычислительных узлов). В работе [22] авторами уже обсуждалась проблема возможной неодинаковой производительности ядер. Так как, в отличие от `openmpi`, `mpich` использует все ядра на процессоре вычислительного узла, производительность всего вычислительного узла ограничивается производительностью самого “слабого” ядра на процессоре.
3. На основе полученных результатов можно сделать вывод о том, что при наличии ядер с разной производительностью имеет смысл использовать автоматическую перебалансировку объема вычислительных операций между MPI-процессами в зависимости от полученной в процессе вычислений оценочной производительности ядер или первоначально делить матрицу неравномерно между MPI-процессами в зависимости от их производительности, а не равными частями. Такой подход может дать более высокие показатели эффективности, но этот вопрос выходит за рамки данной статьи. Отметим, что возможный подход к решению этого вопроса может быть подобен идеям, описанным в статье [23], в которой рассматривались идеи крупномасштабного автотюнинга под особенности используемой вычислительной системы с распределенной памятью для решения задач, подобных рассматриваемым в данной статье.
4. Эффективности программ с использованием особенностей стандартов MPI-2, MPI-3 и MPI-4 получаются эквивалентными, что не соответствует результатам, полученным в предыдущей статье [17], где было явно заметно преимущество использования MPI-функций из стандарта MPI-4 над остальными программными реализациями рассматриваемого алгоритма. Это связано с тем, что последние версии библиотек `mpich` и `openmpi` могут самостоятельно оптимизировать вызовы коллективных операций взаимодействия типа `Alreduce()` за счет хэширования информации о сформированном при первом вызове функции правиле взаимодействия между MPI-процессами, если при предыдущих вызовах этой функции пересылка данных уже происходила между теми же областями в памяти. Так как в наших реализациях программ массивы аллоцируются один раз до запуска итерационного процесса, а затем обновляются in-place, по сути неявно происходит та же самая оптимизация, что была проделана явно с использованием функций из стандарта MPI-4. Отметим, что в общем случае оптимизации автоматически могут быть применены не всегда.

Также отметим следующие особенности предложенных программных реализаций.

1. Все примеры программ построены таким образом, что их можно легко переписать с использованием языков программирования C/C++/Fortran. Это связано с тем, что все MPI-функции, используемые в программных реализациях Python, используют синтаксис, эквивалентный синтаксису соответствующих MPI-функций в языках программирования C/C++/Fortran.
2. Алгоритм 1 в первую очередь предназначен для решения систем линейных уравнений с плотно заполненной матрицей. Он не тестировался для решения систем линейных уравнений с разреженной матрицей. В силу технических особенностей работы с разреженными матрицами возможно, что результаты по эффективности программной реализации могут существенно отличаться (как в лучшую, так и в худшую сторону). Требуются дополнительные исследования по этому вопросу.
3. В случае наличия на вычислительных узлах видеокарт программы можно легко модифицировать, если заменить основные вычисления с помощью функции `dot()` из пакета `numpy` на вычисления с помощью аналогичной функции из пакета `cupy`, которая позволяет задействовать для вычислений видеокарту. Таким образом легко использовать технологии гибридного параллельного программирования MPI+OpenMP+CUDA. В связи с тем, что изменения в программной реализации будут достаточно простыми, код программы здесь не приводится.

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

8. Пример расчетов. Приведем примеры некоторых расчетов. При этом сразу обратим внимание на то, что для наглядности влияния ошибок машинного округления в численных экспериментах будет полагаться, что $\delta = 0$ и $h = 0$, т.е. входные данные считаются заданными точно, а оператор A_h совпадает с точным A. Тем не менее, даже в этом идеализированном случае при численном решении возникает необходимость в регуляризации из-за ошибок машинного округления, которые приводят к ненулевой мере несовместности приближенных данных $\mu = \inf\limits_x \|A_h x - B_\delta\|$.

В рамках описанного в разделе 6 примера с матрицей A размера $M \times N \equiv 60\,000 \times 50\,000$ получены следующие результаты. Мера несовместности, вычисленная с помощью предложенного подхода, получилось равной по порядку 10^{-8}. Значение параметра регуляризации α^*, выбранное по обобщенному принципу невязки (3), получилось равным по порядку 10^{-10}. Отметим, что при значениях параметра регуляризации α близких к α^* при поиске x^α из уравнения (4) с помощью рассматриваемого метода требовалось совершить всего лишь около 30 итераций (вместо 50 000 при использовании классического алгоритма). При этом относительная ошибка решения составила 25%. Классический метод решения уравнения (4) в рамках этого примера оказался совершенно неконструктивным для применения, так как для каждого значения α при итерационном поиске α^* из уравнения (3) требовалось выполнять все 50 000 итераций.

Поэтому для демонстрации отличия предложенного метода от классического был выбран пример с матрицей A меньшего размера $M \times N \equiv 15\,000 \times 12\,500$, для которого удалось получить наглядные результаты. На рис. 5 представлены результаты вычисления регуляризованного решения x^{α^*} системы (13) — набора сеточных значений функции $\rho(x)$, полученного двумя способами решения системы (4): 1) классическим методом сопряженных градиентов (итерации останавливаются после N шагов); 2) усовершенствованным методом iCG (алгоритм 1) с критерием остановки, учитывающим ошибки округления.

Видно, что учет ошибок машинного округления позволяет существенно повысить качество регуляризованного решения. Классический метод решения (4), не контролирующий накопление погрешностей, после выполнения N итераций дает сильно завышенную оценку меры несовместности μ, которая по порядку равна 10^2, в то время как мера несовместности, вычисленная с помощью предложенного подхода, имеет значение по порядку равное 10^{-9}. Это приводит к выбору завышенного значения α^* (10^{-1} вме-

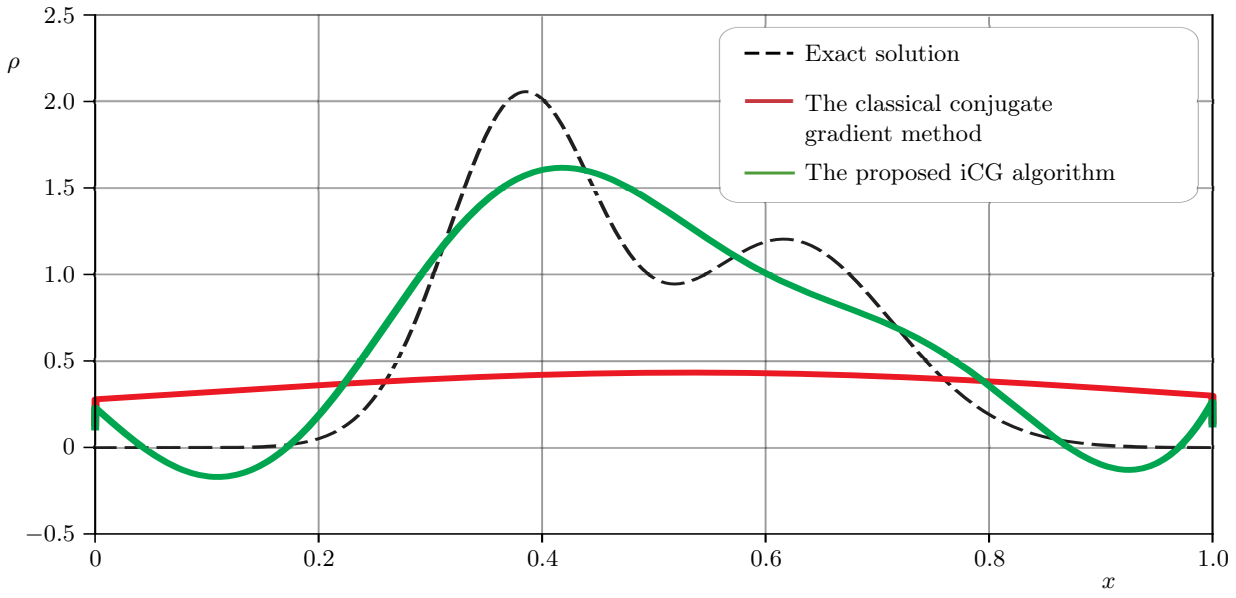

Рис. 5. Сравнение восстановленных решений: классический метод сопряженных градиентов (красная линия) и предложенный алгоритм iCG зеленая линия). Точное решение показано черным пунктиром. Параметры расчета: $M = 15\,000$ ($N_s = 5\,000$), $N = 12\,500$, $\delta = 0$, $h = 0$, $\Delta = 10^{-16.3}$

Fig. 5. Comparison of the reconstructed solutions: the classical conjugate gradient method (red line) and the proposed iCG algorithm (green line). The exact solution is shown as a black dotted line. Calculation parameters: $M = 15\,000$ ($N_s = 5\,000$), $N = 12\,500$, $\delta = 0$, $h = 0$, $\Delta = 10^{-16.3}$

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

сто 10^{-11}) и, как следствие, к излишнему сглаживанию ("перерегуляризации"). При этом предложенный в работе подход останавливает итерации на ранней стадии (35 итерации вместо 12 500 при реализации классического алгоритма), когда невязка становится сравнима с "шумовым фоном", обусловленным ошибками округления. При таком подходе улучшенный алгоритм обеспечивает более точное приближение к истинному решению: относительная ошибка решения составила 24% по сравнению с 73% при применении классического алгоритма.

Замечание. Если сделать все 12 500 итераций классическим методом, то решение станет не определено из-за деления на машинный ноль. В этом случае решение не восстанавливается (что, в частности, делает сложным применение метода секущих для поиска α^*, так как ожидается, что функция будет определена для всех α). Поэтому в данном примере использовалась следующая эвристика: делается максимально возможное число итераций, пока решение не обращается в `NaN` — 45. Данный факт опять же говорит о преимуществах рассматриваемого в статье метода, который не использует такого рода эвристик.

9. Дискуссия. Обсудим несколько важных вопросов.

1. Регуляризованное решение x^{α^*} посредством обобщенного принципа невязки (3) зависит от δ, h и μ, т.е. оно зависит: 1) от погрешности задания входных данных, за оценку которой отвечают экспериментаторы, проводящие соответствующий эксперимент; 2) от погрешности задания матрицы оператора, за оценку которой отвечают теоретики, строящие модель рассматриваемого явления/процесса (что включает как учет/пренебрежение различными законами в модели, так и учет ошибок конечноразностной аппроксимации при переходе от бесконечномерной постановки задачи к конечномерной); 3) от оценки меры несовместности приближенных данных, которая вычисляется с помощью каких-либо численных методов. На первые два значения обычно нет возможности повлиять, и они считаются заданными. Однако, поскольку результат вычисления меры несовместности приближенных данных μ зависит от выбранного численного метода минимизации функционала (2), на аккуратность вычисления соответствующего значения можно повлиять. В частности, авторами было замечено, что хорошо известные из литературы методы минимизации могут приводить к завышению оценки меры несовместности и, как следствие, к излишнему сглаживанию регуляризованного решения. Учет накапливающихся при вычислениях ошибок машинного округления может улучшить оценку меры несовместности и, как следствие, улучшить качество полученного регуляризованного решения.

   При этом надо отметить, что в методах итеративной регуляризации номер итерации s играет роль параметра регуляризации. Как следствие, вполне естественно выбирать этот параметр регуляризации, согласуя его в том числе с учетом ошибок машинного округления. Поэтому критерий прекращения итерационного процесса в рамках рассматриваемого подхода может быть сформулирован как

$$\|A_h x^{(s)} - b_\delta\|^2 - \left(\delta + h\|x^{(s)}\|\right)^2 - \mu^2 < 0.$$

   Таким образом, учет ошибок машинного округления опять же будет содержаться в процедуре аккуратной оценки меры несовместности приближенных данных μ. В настоящей работе, в которой за основу был взят вариационный подход к построению регуляризирующих алгоритмов, эта оценка выполняется достаточно естественным образом. В случае же итеративной регуляризации вопрос о возможности аккуратной оценки μ в процессе вычислений остается открытым и представляет большой интерес для возможных исследований.
2. В рамках теории регуляризации существуют более простые методы, которые предполагают, что характерное значение параметра регуляризации α^* заранее известно для решаемого класса прикладных задач. В настоящей работе рассмотрен метод, который применим для произвольных линейных некорректно поставленных обратных задач и любых используемых вычислительных систем. Зачастую, если есть априорная информация об особенностях решения, возможно использование более простых методов.
3. Из рис. 5 видно, что даже на "точных" данных ($\delta = 0$ и $h = 0$) может быть невозможным восстановление "точного" решения из-за ошибок машинного округления. Подобная проблема может возникнуть и при решении множества других прикладных обратных задач большой численной размерности. Единственный путь к улучшению качества численного решения в таких случая — использование вычислений с большей точностью. Например, использование вычислений с "четверной" точностью ($\Delta = 10^{-34.0}$).

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

10. Заключение. Предложен и программно реализован усовершенствованный параллельный алгоритм решения больших переопределенных систем линейных алгебраических уравнений с плотно заполненной матрицей, возникающих при решении некорректно поставленных линейных обратных задач. Ключевой особенностью алгоритма является адаптивный учет накопленных ошибок машинного округления непосредственно в процессе выбора параметра регуляризации. Это позволяет существенно повысить точность получаемого регуляризованного решения.

На примере трехмерной обратной задачи электростатики, сведенной к одномерной, проведено исследование сильной масштабируемости предложенных реализаций на суперкомпьютере "Ломоносов-2". Показано, что: 1) использование неблокирующих коллективных операций (стандарт MPI-3) и отложенных запросов на взаимодействие (стандарт MPI-4) позволяет частично компенсировать дополнительные вычислительные и коммуникационные затраты, связанные с учетом ошибок округления; 2) эффективность параллельных реализаций существенно зависит от выбора конкретной MPI-реализации стандарта MPI (`mpich` или `openmpi`), в частности, при использовании `openmpi` наблюдалась близкая к идеальной эффективность, но время реальных вычислений было значительно бóльшим по сравнению с использованием `mpich`; 3) современные версии MPI-библиотек способны автоматически оптимизировать повторяющиеся коллективные операции (например, за счет хэширования коммуникационных шаблонов), что в некоторых случаях нивелирует преимущества явного применения средств MPI-4; тем не менее, предложенный подход с отложенными запросами на взаимодействие сохраняет актуальность для систем, где такие автоматические оптимизации не гарантированы.

Программная реализация на языке Python с использованием пакетов `numpy` и `mpi4py` выполнена таким образом, что ее ключевые фрагменты могут быть легко перенесены на языки C/C++/Fortran. Это обеспечивает воспроизводимость результатов и возможность применения алгоритма в высокопроизводительных вычислительных средах.

Полученные результаты подтверждают целесообразность адаптивного учета ошибок машинного округления при решении плохо обусловленных СЛАУ, а также демонстрируют эффективность современных стандартов MPI для построения масштабируемых параллельных реализаций.

Дальнейшие исследования могут быть направлены на: 1) адаптацию предложенного алгоритма для систем с разреженными матрицами; 2) использование гетерогенных вычислительных ресурсов (GPU) за счет интеграции с библиотеками типа `cupy`; 3) разработку методов динамической балансировки нагрузки при неравномерной производительности ядер; 4) применение идей суперкомпьютерного кодизайна для разработки алгоритмов автотюнинга для автоматического подбора оптимальных параметров декомпозиции данных и коммуникационных стратегий в зависимости от архитектуры конкретной вычислительной системы.

Список литературы

1. *Voevodin Vl.V., Antonov A.S., Nikitenko D.A., et al.* Supercomputer Lomonosov-2: large scale, deep monitoring and fine analytics for the user community // Supercomputing Frontiers and Innovations. 2019. **6**, N 2. 4–11. doi 10.14529/jsfi190201.
2. *Tikhonov A.N., Goncharsky A.V., Stepanov V.V., Yagola A.G.* Numerical Methods For The Solution Of Ill-Posed Problems. Dordrecht: Kluwer Academic Publishers, 1995. doi 10.1007/978-94-015-8480-7.
3. *Hestenes M.R., Stiefel E.* Methods of conjugate gradients for solving linear systems // Journal of Research of the National Bureau of Standards. 1952. **49**, N 6. 409–436.
4. *Strakoš Z.* On the real convergence rate of the conjugate gradient method // Linear Algebra and Its Applications. 1991. **154**. 535–549. doi 10.1016/0024-3795(91)90393-B.
5. *Woźniakowski H.* Roundoff-error analysis of a new class of conjugate-gradient algorithms // Linear Algebra and Its Applications. 1980. **29**. 507–529.
6. *Kaasschieter E.F.* A practical termination criterion for the conjugate gradient method // BIT Numerical Mathematics. 1988. **28**, N 2. 308–322. doi 10.1007/BF01934094.
7. *Arioli M., Duff I., Ruiz D.* Stopping criteria for iterative solvers // SIAM Journal on Matrix Analysis and Applications. 1992. **13**, N 1. 138–144. doi 10.1137/0613012.
8. *Notay Y.* On the convergence rate of the conjugate gradients in presence of rounding errors // Numerische Mathematik. 1993. **65**, N 1. 301–317. doi 10.1007/BF01385754.

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

9. *Axelsson O., Kaporin I.* Error norm estimation and stopping criteria in preconditioned conjugate gradient iterations // Numerical Linear Algebra with Applications. 2001. **8**, N 4. 265–286. doi 10.1002/nla.244.
10. *Strakoš Z., Tichý P.* On error estimation in the conjugate gradient method and why it works in finite precision computations // Electronic Transactions on Numerical Analysis. 2002. **13**. 56–80.
11. *Meurant G.* The Lanczos and conjugate gradient algorithms: from theory to finite precision computations. SIAM, 2006.
12. *Chang X.-W., Paige C.C., Titley-Peloquin D.* Stopping criteria for the iterative solution of linear least squares problems // SIAM Journal on Matrix Analysis and Applications. 2009. **31**, N 2. 831–852. doi 10.1137/080724071.
13. *Jiránek P., Strakoš Z., Vohralík M.* A posteriori error estimates including algebraic error and stopping criteria for iterative solvers // SIAM Journal on Scientific Computing. 2010. **32**, N 3. 1567–1590. doi 10.1137/08073706X.
14. *Cools S., Yetkin E.F., Agullo E., et al.* Analyzing the effect of local rounding error propagation on the maximal attainable accuracy of the pipelined conjugate gradient method // SIAM Journal on Matrix Analysis and Applications. 2018. **39**, N 1. 426–450. doi 10.1137/17M1117872.
15. *Greenbaum A., Liu H., Chen T.* On the convergence rate of variants of the conjugate gradient algorithm in finite precision arithmetic // SIAM Journal on Scientific Computing. 2021. **43**, N 5. S496–S515. doi 10.1137/20M1346249.
16. *Lukyanenko D.V., Shinkarev V.D., Yagola A.G.* Accounting for round-off errors when using gradient minimization methods // Algorithms. 2022. **15**, N 9. Article Number 324. doi 10.3390/a15090324.
17. *Lukyanenko D.V.* Parallel algorithm for solving overdetermined systems of linear equations, taking into account round-off errors // Algorithms. 2023. **16**, N 5. Article Number 242. doi 10.3390/a16050242.
18. *Dalcin L., Fang Y.-L.L.* mpi4py: status update after 12 years of development // Computing in Science & Engineering. 2021. **23**, N 4. 47–54. doi 10.1109/MCSE.2021.3083216.
19. *Rogowski M., Aseeri S., Keyes D., Dalcin L.* mpi4py.futures: MPI-based asynchronous task execution for Python // IEEE Transactions on Parallel and Distributed Systems. 2023. **34**, N 2. 611–622. doi 10.1109/TPDS.2022.3225481.
20. *Demmel J.W., Heath M.T., Van Der Vorst H.A.* Parallel numerical linear algebra // Acta Numerica. 1993. **2**. 111–197. doi 10.1017/S096249290000235X.
21. *Eller P.R., Gropp W.* Scalable non-blocking preconditioned conjugate gradient methods // in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16), Salt Lake City, UT, USA, November 13–18, 2016. IEEE Press, 2016. pp. 204-215. doi 10.1109/SC.2016.17.
22. *Lukyanenko D.V., Torbin S.S., Shinkarev V.D.* How to parallelize "non-parallelizable"minimization functions // Lobachevskii Journal of Mathematics. 2025. **46**, N 8. 3725–3740. doi 10.1134/S199508022560997X.
23. *Antonov A.S., Voevodin Vl.V., Lukyanenko D.V.* Supercomputing co-design for solving ill-posed linear inverse problems using iterative algorithms // Supercomputing Frontiers and Innovations. 2025. **12**, N 4. 16–33. doi 10.14529/jsfi250402.

Получена	*Принята*	*Опубликована*
15 марта 2026 г.	*6 мая 2026 г.*	*26 мая 2026 г.*

Информация об авторах

Валентин Дмитриевич Шинкарев — аспирант; Московский государственный университет имени М. В. Ломоносова, физический факультет, кафедра математики, Ленинские горы, 1, стр. 2, 119991, Москва, Российская Федерация.

Аким Михайлович Златковский — студент; Московский государственный университет имени М. В. Ломоносова, физический факультет, кафедра математики, Ленинские горы, 1, стр. 2, 119991, Москва, Российская Федерация.

Дмитрий Витальевич Лукьяненко — д.ф.-м.н., профессор; 1) Московский государственный университет имени М. В. Ломоносова, физический факультет, кафедра математики, Ленинские горы, 1, стр. 2, 119991, Москва, Российская Федерация; 2) Московский государственный университет имени М. В. Ломоносова, Научно-исследовательский вычислительный центр, Ленинские горы, 1, стр. 4, 119991, Москва, Российская Федерация; 3) Московский Центр фундаментальной и прикладной математики, Ленинские горы, 1, 119234, Москва, Российская Федерация.

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

References

1. Vl. V. Voevodin, A. S. Antonov, D. A. Nikitenko, et al., "Supercomputer Lomonosov-2: Large Scale, Deep Monitoring and Fine Analytics for the User Community," Supercomputing Frontiers and Innovations **6** (2), 4–11 (2019). doi 10.14529/jsfi190201.
2. A. N. Tikhonov, A. V. Goncharsky, V. V. Stepanov, and A. G. Yagola, *Numerical Methods for The Solution of Ill-Posed Problems* (Kluwer Academic Publishers, Dordrecht, 1995). doi 10.1007/978-94-015-8480-7.
3. M. R. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," Journal of Research of the National Bureau of Standards **49** (6), 409–436 (1952).
4. Z. Strakoš, "On the Real Convergence Rate of the Conjugate Gradient Method," Linear Algebra and Its Applications **154**, 535–549 (1991). doi 10.1016/0024-3795(91)90393-B.
5. H. Woźniakowski, "Roundoff-Error Analysis of a New Class of Conjugate-Gradient Algorithms," Linear Algebra and Its Applications **29**, 507–529 (1980).
6. E. F. Kaasschieter, "A Practical Termination Criterion for the Conjugate Gradient Method," BIT Numerical Mathematics **28** (2), 308–322 (1988). doi 10.1007/BF01934094.
7. M. Arioli, I. Duff, and D. Ruiz, "Stopping Criteria for Iterative Solvers," SIAM Journal on Matrix Analysis and Applications **13** (1), 138–144 (1992). doi 10.1137/0613012.
8. Y. Notay, "On the Convergence Rate of the Conjugate Gradients in Presence of Rounding Errors," Numerische Mathematik **65** (1), 301–317 (1993). doi 10.1007/BF01385754.
9. O. Axelsson and I. Kaporin, "Error Norm Estimation and Stopping Criteria in Preconditioned Conjugate Gradient Iterations," Numerical Linear Algebra with Applications **8** (4), 265–286 (2001). doi 10.1002/nla.244.
10. Z. Strakoš and P. Tichý, "On Error Estimation in the Conjugate Gradient Method and Why It Works in Finite Precision Computations," Electronic Transactions on Numerical Analysis **13**, 56–80 (2002).
11. G. Meurant, *The Lanczos and Conjugate Gradient Algorithms: From Theory to Finite Precision Computations* (SIAM, 2006).
12. X.-W. Chang, C. C. Paige and D. Titley-Peloquin, "Stopping Criteria for the Iterative Solution of Linear Least Squares Problems," SIAM Journal on Matrix Analysis and Applications **31** (2), 831–852 (2009). doi 10.1137/080724071.
13. P. Jiránek, Z. Strakoš, and M. Vohralík, "A Posteriori Error Estimates Including Algebraic Error and Stopping Criteria for Iterative Solvers," SIAM Journal on Scientific Computing **32** (3), 1567–1590 (2010). doi 10.1137/08073706X.
14. S. Cools, E. F. Yetkin, E. Agullo, et al., "Analyzing the Effect of Local Rounding Error Propagation on the Maximal Attainable Accuracy of the Pipelined Conjugate Gradient Method," SIAM Journal on Matrix Analysis and Applications **39** (1), 426–450 (2018). doi 10.1137/17M1117872.
15. A. Greenbaum, H. Liu, and T. Chen, "On the Convergence Rate of Variants of the Conjugate Gradient Algorithm in Finite Precision Arithmetic," SIAM Journal on Scientific Computing **43** (5), S496–S515 (2021). doi 10.1137/20M1346249.
16. D. V. Lukyanenko, V. D. Shinkarev, and A. G. Yagola, "Accounting for Round-Off Errors When Using Gradient Minimization Methods," Algorithms **15** (9), Article Number 324 (2022). doi 10.3390/a15090324.
17. D. V. Lukyanenko, "Parallel Algorithm for Solving Overdetermined Systems of Linear Equations, Taking into Account Round-Off Errors," Algorithms **16** (5), Article Number 242 (2023). doi 10.3390/a16050242.
18. L. Dalcin and Y.-L. L. Fang, "mpi4py: Status Update After 12 Years of Development," Computing in Science & Engineering **23** (4), 47–54 (2021). doi 10.1109/MCSE.2021.3083216.
19. M. Rogowski, S. Aseeri, D. Keyes, and L. Dalcin, "mpi4py.futures: MPI-Based Asynchronous Task Execution for Python," IEEE Transactions on Parallel and Distributed Systems **34** (2), 611–622 (2023). doi 10.1109/TPDS.2022.3225481.
20. J. W. Demmel, M. T. Heath, and H. A. Van Der Vorst, "Parallel Numerical Linear Algebra," Acta Numerica **2**, 111–197 (1993). doi 10.1017/S096249290000235X.
21. P. R. Eller and W. Gropp, "Scalable Non-Blocking Preconditioned Conjugate Gradient Methods," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16), Salt Lake City, UT, USA, November 13–18, 2016* (IEEE Press, 2016), pp. 204-215. doi 10.1109/SC.2016.17.
22. D. V. Lukyanenko, S. S. Torbin, and V. D. Shinkarev, "How to Parallelize "Non-Parallelizable" Minimization Functions," Lobachevskii Journal of Mathematics **46** (8), 3725–3740 (2025). doi 10.1134/S199508022560997X.

2026, **27** (2), 227–260. doi 10.26089/NumMet.v27r216

23. A. S. Antonov, Vl. V. Voevodin, and D. V. Lukyanenko, "Supercomputing Co-Design for Solving Ill-Posed Linear Inverse Problems Using Iterative Algorithms," Supercomputing Frontiers and Innovations **12** (4), 16–33 (2025). doi 10.14529/jsfi250402.

Received *March 15, 2026*
Accepted *May 6, 2026*
Published *May 26, 2026*

Information about the authors

Valentin D. Shinkarev — Postgraduate Student; Lomonosov Moscow State University, Faculty of Physics, Department of Mathematics, Leninskie Gory, 1, building 2, 119991, Moscow, Russia.

Akim M. Zlatkovskii — Student; Lomonosov Moscow State University, Faculty of Physics, Department of Mathematics, Leninskie Gory, 1, building 2, 119991, Moscow, Russia.

Dmitry V. Lukyanenko — Dr.Habil, Professor; 1) Lomonosov Moscow State University, Faculty of Physics, Department of Mathematics, Leninskie Gory, 1, building 2, 119991, Moscow, Russia; 2) Lomonosov Moscow State University, Research Computing Center, Leninskie Gory, 1, building 4, 119991, Moscow, Russia; 3) Moscow Center of Fundamental and Applied Mathematics, Leninskie Gory, 1, 119234, Moscow, Russia.