

Московский государственный университет имени М.В.Ломоносова

Факультет вычислительной математики и кибернетики

Кафедра математических методов прогнозирования

**Задание по курсу «Суперкомпьютерное
моделирование и технологии»**

**Решение краевой задачи для уравнения Пуассона
методом конечных разностей.**

Выполнил:

студент 617 группы

Г.В. Кормаков

Содержание

1	Постановка задачи	2
2	Численный метод решения	3
2.1	Общая схема метода конечных разностей	3
2.2	Конкретный вид разностной схемы для варианта 8	3
2.3	Метод решения СЛАУ	6
2.4	Вид функций	7
3	Нахождение $F(x, y), \psi(x, y)$	7
4	Описание программной реализации	8
4.1	Формализация требований на домены	8
4.2	Алгоритм разбиения на блоки	9
4.3	Реализация с использованием MPI и OpenMP	11
5	Результаты на системах Blue Gene/P и Polus	12
6	Заключение	16
7	Литература	17
8	Приложение 1. Код MPI программы	18
9	Приложение 2. Код MPI программы с оптимизацией сопряжёнными градиентами	31

1 Постановка задачи

В прямоугольнике $\Pi = \{(x, y) : A_1 \leq x \leq A_2, B_1 \leq y \leq B_2\}$, граница Γ которого состоит из отрезков

$$\gamma_R = \{(A_2, y), B_1 \leq y \leq B_2\}, \quad \gamma_L = \{(A_1, y), B_1 \leq y \leq B_2\}$$

$$\gamma_T = \{(x, B_2), A_1 \leq x \leq A_2\}, \quad \gamma_B = \{(x, B_1), A_1 \leq x \leq A_2\}$$

рассматривается дифференциальное уравнение Пуассона с потенциалом

$$-\Delta u + q(x, y)u = F(x, y) \quad (1)$$

в котором оператор Лапласа

$$\Delta u = \frac{\partial}{\partial x} \left(k(x, y) \frac{\partial u}{\partial x} \right) + \frac{\partial}{\partial y} \left(k(x, y) \frac{\partial u}{\partial y} \right) \quad (2)$$

Для выделения единственного решения уравнение дополняется граничными условиями. На каждом отрезке границы прямоугольника Π задается условие одним из двух способов - условиями Дирихле (1-ого типа) или условиями второго (Неймана) и третьего типа.

Для выданного варианта 8 задания краевые условия задаются следующими условиями: третьего типа на правой и левой границе и второго на верхней и нижней на сетке (см. раздел 2). Общая формула условий третьего типа выглядит следующим образом:

$$\left(k \frac{\partial u}{\partial \mathbf{n}} \right) (x, y) + \alpha u(x, y) = \psi(x, y), \quad (3)$$

где \mathbf{n} - единичная внешняя нормаль к границе прямоугольника.

Краевое условие второго типа (условие Неймана) содержится в краевом условии третьего типа (случай $\alpha = 0$ в 3).

Функции $F(x, y)$, $\varphi(x, y)$, $\psi(x, y)$, коэффициент $k(x, y)$, потенциал $q(x, y)$ и параметр $\alpha \geq 0$ считаются известными, функцию $u(x, y)$, удовлетворяющую уравнению 1 и граничным условиям, определенным вариантом задания 8, требуется найти.

Важно отметить, что нормаль \mathbf{n} не определена в угловых точках прямоугольника. Краевое условие третьего типа будет рассматриваться лишь в тех точках границы, где нормаль существует.

2 Численный метод решения

2.1 Общая схема метода конечных разностей

В качестве метода решения задачи Пуассона 1 с потенциалом предлагается использовать метод конечных разностей.

Для этого область Π дискретизуем сеткой $\bar{\omega}_h = \bar{\omega}_1 \times \bar{\omega}_2$, где $\bar{\omega}_1$ – разбиение сетки по оси Ox с шагом $h_1 = \frac{A_2 - A_1}{M}$ и $\bar{\omega}_2$ – разбиение сетки по оси Oy с шагом $h_2 = \frac{B_2 - B_1}{N}$:

$$\bar{\omega}_1 = \{x_i = A_1 + ih_1, i = \overline{0, M}\}, \bar{\omega}_2 = \{y_j = B_1 + jh_2, j = \overline{0, N}\}$$

Также примем обозначение ω_h для внутренних узлов сетки $\bar{\omega}_h$

Рассматривается линейное пространство H функций, заданных на сетке $\bar{\omega}_h$. Обозначим через w_{ij} значение сеточной функции $w \in H$ в узле сетки $(x_i, y_j) \in \bar{\omega}_h$. Будем считать, что в пространстве H задано скалярное произведение и евклидова норма

$$[u, v] = \sum_{i=0}^M h_1 \sum_{j=0}^N h_2 \rho_{ij} u_{ij} v_{ij}, \quad \|u\|_E = \sqrt{[u, u]},$$

где ρ_{ij} – весовая функция $\rho_{ij} = \rho^{(1)}(x_i) \rho^{(2)}(y_j)$ с

$$\rho^{(1)}(x_i) = \begin{cases} 1, & 1 \leq i \leq M-1 \\ 1/2, & i = 0, i = M \end{cases} \quad \rho^{(2)}(y_j) = \begin{cases} 1, & 1 \leq j \leq N-1 \\ 1/2, & j = 0, j = N \end{cases}$$

В методе конечных разностей дифференциальная задача математической физики заменяется конечно-разностной операторной задачей вида

$$Aw = B \tag{4}$$

где $A : H \rightarrow H$ – оператор, действующий в пространстве сеточных функций, $B \in H$ – известная правая часть. Задача 4 называется разностной схемой. Решение этой задачи считается численным решением исходной дифференциальной задачи.

2.2 Конкретный вид разностной схемы для варианта 8

Уравнение 1 приближается для всех внутренних точек ω_h сетки разностной схемой следующего вида:

$$-\Delta_h w_{ij} + q_{ij} w_{ij} = F_{ij}, \quad i = \overline{1, M-1}, j = \overline{1, N-1},$$

в котором $F_{ij} = F(x_i, y_j)$, $q_{ij} = q(x_i, y_j)$ и разностный оператор Лапласа

$$\Delta_h w_{ij} = \frac{1}{h_1} \left(k(x_i + 0.5h_1, y_j) \frac{w_{(i+1)j} - w_{ij}}{h_1} - k(x_i - 0.5h_1, y_j) \frac{w_{ij} - w_{(i-1)j}}{h_1} \right) + \\ + \frac{1}{h_2} \left(k(x_i, y_j + 0.5h_2) \frac{w_{i(j+1)} - w_{ij}}{h_2} - k(x_i, y_j - 0.5h_2) \frac{w_{ij} - w_{i(j-1)}}{h_2} \right)$$

В разностном операторе Лапласа введём обозначения для правых и левых разностных производных по координатам:

$$w_{x,ij} \equiv \frac{w_{(i+1)j} - w_{ij}}{h_1}, \quad w_{\bar{x},ij} \equiv w_{x,(i-1)j} = \frac{w_{ij} - w_{(i-1)j}}{h_1} \\ w_{y,ij} \equiv \frac{w_{i(j+1)} - w_{ij}}{h_2}, \quad w_{\bar{y},ij} \equiv w_{y,i(j-1)} \equiv \frac{w_{ij} - w_{i(j-1)}}{h_2}$$

и зададим сеточные коэффициенты

$$a_{ij} \equiv k(x_i - 0.5h_1, y_j), \quad b_{ij} \equiv k(x_i, y_j - 0.5h_2).$$

Тогда разностный оператор Лапласа равен

$$\Delta_h w_{ij} = \frac{1}{h_1} (k(x_i + 0.5h_1, y_j) w_{x,ij} - k(x_i - 0.5h_1, y_j) w_{\bar{x},ij}) + \\ + \frac{1}{h_2} (k(x_i, y_j + 0.5h_2) w_{y,ij} - k(x_i, y_j - 0.5h_2) w_{\bar{y},ij}) = \\ = \frac{a_{(i+1)j} w_{x,ij} - a_{ij} w_{\bar{x},ij}}{h_1} + \frac{b_{i(j+1)} w_{y,ij} - b_{ij} w_{\bar{y},ij}}{h_2} = \\ = \frac{a_{(i+1)j} w_{\bar{x},(i+1)j} - a_{ij} w_{\bar{x},ij}}{h_1} + \frac{b_{i(j+1)} w_{\bar{y},i(j+1)} - b_{ij} w_{\bar{y},ij}}{h_2} \equiv (aw_{\bar{x}})_{x,ij} + (bw_{\bar{y}})_{y,ij} \quad (5)$$

Итого, получаем $(M-1) \cdot (N-1)$ уравнений во внутренних точках:

$$-\Delta_h w_{ij} + q_{ij} w_{ij} = F_{ij}, \quad i = \overline{1, M-1}, j = \overline{1, N-1} \quad (6)$$

Рассмотрим конкретные граничные условия, заданные в варианте 8 (см. таблицу 1).

Для правой (схема 7) и левой (схема 8) границы (γ_R, γ_L соответственно) задаются условия третьего типа. Разностный вариант (с учётом членов для соответствия порядка погрешности аппроксимации с основным уравнением 5) для них выглядит следующим образом:

$$\frac{2}{h_1} (aw_{\bar{x}})_{Mj} + \left(q_{Mj} + \frac{2\alpha_R}{h_1} \right) w_{Mj} - (bw_{\bar{y}})_{y,Mj} = F_{Mj} + \frac{2}{h_1} \psi_{Mj}^{(R)}, \quad j = \overline{1, N-1} \quad (7)$$

$$-\frac{2}{h_1} (aw_{\bar{x}})_{1j} + \left(q_{0j} + \frac{2\alpha_L}{h_1} \right) w_{0j} - (bw_{\bar{y}})_{y,0j} = F_{0j} + \frac{2}{h_1} \psi_{0j}^{(L)}, \quad j = \overline{1, N-1} \quad (8)$$

Для верхней (схема 9) и нижней (схема 10) границы (γ_T, γ_B соответственно) задаются условия второго типа (Неймана).

$$\frac{2}{h_2} (bw_{\bar{y}})_{iN} + q_{iN} w_{iN} - (aw_{\bar{x}})_{x,iN} = F_{iN} + \frac{2}{h_2} \psi_{iN}^{(T)}, \quad i = \overline{1, M-1} \quad (9)$$

$$-\frac{2}{h_2} (bw_{\bar{y}})_{i1} + q_{i0} w_{i0} - (aw_{\bar{x}})_{x,i0} = F_{i0} + \frac{2}{h_2} \psi_{i0}^{(B)}, \quad i = \overline{1, M-1} \quad (10)$$

Также в угловых точках не определена нормаль, поэтому необходимо их учесть отдельными уравнениями.

Для точки $P(A_1, B_1)$ прямоугольника Π

$$-\frac{2}{h_1}(aw_{\bar{x}})_{10} - \frac{2}{h_2}(bw_{\bar{y}})_{01} + \left(q_{00} + \frac{2\alpha_L}{h_1}\right)w_{00} = F_{00} + \left(\frac{2}{h_1} + \frac{2}{h_2}\right)\psi_{00} \quad (11)$$

Для точки $P(A_2, B_1)$ прямоугольника Π

$$\frac{2}{h_1}(aw_{\bar{x}})_{M0} - \frac{2}{h_2}(bw_{\bar{y}})_{M1} + \left(q_{M0} + \frac{2\alpha_R}{h_1}\right)w_{M0} = F_{M0} + \left(\frac{2}{h_1} + \frac{2}{h_2}\right)\psi_{M0} \quad (12)$$

Для точки $P(A_2, B_2)$ прямоугольника Π

$$\frac{2}{h_1}(aw_{\bar{x}})_{MN} + \frac{2}{h_2}(bw_{\bar{y}})_{MN} + \left(q_{MN} + \frac{2\alpha_R}{h_1}\right)w_{MN} = F_{MN} + \left(\frac{2}{h_1} + \frac{2}{h_2}\right)\psi_{MN} \quad (13)$$

Для точки $P(A_1, B_2)$ прямоугольника Π

$$-\frac{2}{h_1}(aw_{\bar{x}})_{1N} + \frac{2}{h_2}(bw_{\bar{y}})_{0N} + \left(q_{0N} + \frac{2\alpha_L}{h_1}\right)w_{0N} = F_{0N} + \left(\frac{2}{h_1} + \frac{2}{h_2}\right)\psi_{0N} \quad (14)$$

Уточним, что обозначили за $\psi^{(T)}, \psi^{(B)}$ функции краевых условий второго типа (для данного варианта), а за $\psi^{(R)}, \psi^{(L)}$ – функции краевых условий третьего типа. В угловых точках вектор нормали не определён, поэтому краевые условия равны значению функции $u(x, y)$.

Вариант задания	Граничные условия				Решение $u(x, y)$	Коэфф. $k(x, y)$	Потенциал $q(x, y)$
	γ_R	γ_L	γ_T	γ_B			
8	3 тип	3 тип	2 тип	2 тип	$u_2(x, y)$	$k_3(x, y)$	$q_2(x, y)$

Таблица 1: Условия задания

Запишем **финальную систему**, убедившись, что она определена.

$$\begin{aligned}
& - (aw_{\bar{x}})_{x,ij} - (bw_{\bar{y}})_{y,ij} + q_{ij}w_{ij} = F_{ij}, i = \overline{1, M-1}, j = \overline{1, N-1} \\
& \frac{2}{h_1} (aw_{\bar{x}})_{Mj} + \left(q_{Mj} + \frac{2}{h_1} \right) w_{Mj} - (bw_{\bar{y}})_{y,Mj} = F_{Mj} + \frac{2}{h_1} \psi_{Mj}^{(R)}, j = \overline{1, N-1} \\
& - \frac{2}{h_1} (aw_{\bar{x}})_{1j} + \left(q_{0j} + \frac{2}{h_1} \right) w_{0j} - (bw_{\bar{y}})_{y,0j} = F_{0j} + \frac{2}{h_1} \psi_{0j}^{(L)}, j = \overline{1, N-1} \\
& \frac{2}{h_2} (bw_{\bar{y}})_{iN} + q_{iN}w_{iN} - (aw_{\bar{x}})_{x,iN} = F_{iN} + \frac{2}{h_2} \psi_{iN}^{(T)}, i = \overline{1, M-1} \\
& - \frac{2}{h_2} (bw_{\bar{y}})_{i1} + q_{i0}w_{i0} - (aw_{\bar{x}})_{x,i0} = F_{i0} + \frac{2}{h_2} \psi_{i0}^{(B)}, i = \overline{1, M-1} \\
& - \frac{2}{h_1} (aw_{\bar{x}})_{10} - \frac{2}{h_2} (bw_{\bar{y}})_{01} + \left(q_{00} + \frac{2}{h_1} \right) w_{00} = F_{00} + \left(\frac{2}{h_1} + \frac{2}{h_2} \right) \frac{\psi_{00}^{(L)} + \psi_{00}^{(B)}}{2} \\
& \frac{2}{h_1} (aw_{\bar{x}})_{M0} - \frac{2}{h_2} (bw_{\bar{y}})_{M1} + \left(q_{M0} + \frac{2}{h_1} \right) w_{M0} = F_{M0} + \left(\frac{2}{h_1} + \frac{2}{h_2} \right) \frac{\psi_{M0}^{(R)} + \psi_{M0}^{(B)}}{2} \\
& \frac{2}{h_1} (aw_{\bar{x}})_{MN} + \frac{2}{h_2} (bw_{\bar{y}})_{MN} + \left(q_{MN} + \frac{2}{h_1} \right) w_{MN} = F_{MN} + \left(\frac{2}{h_1} + \frac{2}{h_2} \right) \frac{\psi_{MN}^{(R)} + \psi_{MN}^{(T)}}{2} \\
& - \frac{2}{h_1} (aw_{\bar{x}})_{1N} + \frac{2}{h_2} (bw_{\bar{y}})_{0N} + \left(q_{0N} + \frac{2}{h_1} \right) w_{0N} = F_{0N} + \left(\frac{2}{h_1} + \frac{2}{h_2} \right) \frac{\psi_{0N}^{(L)} + \psi_{0N}^{(T)}}{2}
\end{aligned}$$

Получили $(N-1)(M-1) + 2(M-1) + 2(N-1) + 4 = MN - M - N + 1 + 2M + 2N - 4 + 4 = MN + N + M + 1$ уравнений. Число неизвестных w_{ij} равно $(M+1)(N+1) = MN + N + M + 1$. И система гарантирует единственность решения для данной разностной схемы.

Таким образом, матрица оператора A определяется коэффициентами перед неизвестными в левой части, а матрицы B – в правой.

2.3 Метод решения СЛАУ

Приближенное решение системы уравнений для сформулированных выше краевых задач может быть получено итерационным методом наименьших невязок. Этот метод позволяет получить последовательность сеточных функций $w^{(k)} \in H, k = 1, 2, \dots$, сходящуюся по норме пространства H к решению разностной схемы, т.е.

$$\|w - w^{(k)}\|_E \xrightarrow{k \rightarrow +\infty} 0$$

Метод является одношаговым. Итерация обновления $w^{(k+1)}$ записывается в виде

$$w_{ij}^{(k+1)} = w_{ij}^{(k)} - \tau_{k+1} r_{ij}^{(k)},$$

где невязка $r^{(k)} = Aw^{(k)} - B$, итерационный параметр $\tau_{k+1} = \frac{[Ar^{(k)}, r^{(k)}]}{\|Ar^{(k)}\|_E^2}$

В качестве условия останковки используем неравенство $\|w^{(k+1)} - w^{(k)}\|_E < \varepsilon$, где ε - положительное число, определяющее точность итерационного метода. Константу ε для данной задачи предлагается взять равной 10^{-6} .

2.4 Вид функций

В таблице 1 приведены конкретные функции для данного варианта. Они задаются следующим образом

$$u_2(x, y) = \sqrt{4 + xy}, \Pi = [0, 4] \times [0, 3] \quad (15)$$

$$k_3(x, y) = 4 + x + y \quad (16)$$

$$q_2(x, y) = \begin{cases} x + y, & x + y \geq 0 \\ 0, & x + y < 0 \end{cases} \quad (17)$$

3 Нахождение $F(x, y), \psi(x, y)$

Поскольку для численной реализации нам предлагается сравнить приближенное решение с истинным, то необходимо найти аналитический вид функции $F(x, y)$ и краевых условий¹ для известного решения $u(x, y) = u_2(x, y) = \sqrt{4 + xy}$, $\Pi = [0, 4] \times [0, 3]$, $k(x, y) = k_3(x, y) = 4 + x + y$ и потенциалом $q(x, y) = q_2(x, y) = \max(0, x + y)$. Для определённости, возьмём вектор нормали \mathbf{n} , направленным извне.

$$\begin{aligned} \frac{\partial u}{\partial x} &= \frac{y}{2\sqrt{4 + xy}}; & \frac{\partial u}{\partial y} &= \frac{x}{2\sqrt{4 + xy}}; \\ \frac{\partial}{\partial x} \left(k(x, y) \frac{y}{2\sqrt{4 + xy}} \right) &= \frac{y}{2\sqrt{4 + xy}} - k(x, y) \frac{y^2}{4(4 + xy)^{3/2}} \\ \text{Аналогично для } \frac{\partial}{\partial y}. \end{aligned}$$

¹В случае варианта 8 необходимо найти вид только функции $\psi(x, y)$

Тогда

$$\begin{aligned}
F(x, y) &= -\Delta u + q(x, y)u = \\
&= -\frac{\partial}{\partial x} \left(k(x, y) \frac{y}{2\sqrt{4+xy}} \right) - \frac{\partial}{\partial y} \left(k(x, y) \frac{x}{2\sqrt{4+xy}} \right) + q(x, y)u = \\
&= -\frac{y}{2u(x, y)} + k(x, y) \frac{y^2}{2u^3(x, y)} - \frac{x}{2u(x, y)} + k(x, y) \frac{x^2}{4u^3(x, y)} + q(x, y)u = \\
&= \frac{-2u^2y + ky^2 - 2u^2x + kx^2 + 4qu^4}{4u^3} = \\
&= \frac{-2(4+xy)(x+y) + (4+x+y)y^2 - 2(4+xy)x + (4+x+y)x^2 + 4qu^4}{4(4+xy)^{3/2}} = \\
&= \frac{x^3 - x^2(2y-4-y) - x(8+2y^2-y^2) + y(-8+4y+y^2) + 4q(4+xy)^2}{4(4+xy)^{3/2}} = \\
&= \frac{x^3 - x^2(y-4) - x(y^2+8) + y(y^2+4y-8) + 4\max(0, x+y)(4+xy)^2}{4(4+xy)^{3/2}} \quad (18)
\end{aligned}$$

Приведём пример вычислений функции $\psi(x, y)$ для в «общем виде» (знак \pm не значит обязательное наличие члена, в зависимости от направления, одна из компонент нормали может равняться 0).

$$\begin{aligned}
\psi(x, y) &= \left(k \frac{\partial u}{\partial \mathbf{n}} \right) (x, y) + \alpha u(x, y) = k \left(\pm \frac{y}{2u} \pm \frac{x}{2u} \right) + \alpha u = \\
&= \frac{(4+x+y)(\pm x \pm y) + 2\alpha(4+xy)}{2\sqrt{4+xy}}
\end{aligned}$$

Для γ_R, γ_L $\alpha = 1$, для γ_T, γ_B $\alpha = 0$. Также учтём знаки нормали к поверхности в соответствующих направлениях. Таким образом, в обозначениях уравнений 11 - 13

$$\psi^{(R)}(x, y) = \frac{y(4+x+y) + 2(4+xy)}{2\sqrt{4+xy}}; \quad \psi^{(L)}(x, y) = \frac{-y(4+x+y) + 2(4+xy)}{2\sqrt{4+xy}} \quad (19)$$

$$\psi^{(T)}(x, y) = \frac{x(4+x+y)}{2\sqrt{4+xy}}; \quad \psi^{(B)}(x, y) = \frac{-x(4+x+y)}{2\sqrt{4+xy}} \quad (20)$$

4 Описание программной реализации

4.1 Формализация требований на домены

Перед непосредственной реализацией важно понять, как осуществлять разбиение на блоки-домены Π_{ij} , соблюдая следующие условия:

1. отношение количества узлов по переменным x и y в каждом домене принадлежало диапазону $[1/2, 2]$

2. количество узлов по переменным x и y любых двух доменов отличалось не более, чем на единицу.

Условия в данной формулировке приведены в постановке задания. Уточним более корректно смысл каждого пункта². Обозначим количество узлов сетки в одном домене $n_x(i)$ и $n_y(j)$ соответственно по x и y . Количество узлов, вообще говоря, зависит от числа процессоров p , выделенных на задачу, в приведённых обозначениях считаем, что размер зависит от числа процессов, выделенных линейно на каждую ось (далее будет объяснён алгоритм разбиения).

Первое условие утверждает, что для любого домена его форма должна быть похожа на прямоугольную или квадратную. Формальнее,

$$\frac{n_x(i)}{n_y(j)} \in [0.5, 2] \quad \forall i, j \quad (21)$$

Данная конфигурация позволяет минимизировать потери во времени на пересылки.

Второе условие даёт равномерность разбиения на домены. Т.е., вообще говоря, размеры доменов по оси x не должны отличаться более, чем на 1, и по y аналогично. Таким образом, $n_x(i), n_y(j)$, фактически должны быть константными, но из-за того, что размеры сетки могут быть не кратны числу процессоров, на граничных блоках мы получаем иные размеры. Именно на то, чтобы отличия в размерах были в этом случае минимальны и наложено ограничение 2.

Формальнее,

$$|n_x(i_1) - n_x(i_2)| \leq 1, |n_y(j_1) - n_y(j_2)| \leq 1, \forall (i_1, j_1), (i_2, j_2) \quad (22)$$

4.2 Алгоритм разбиения на блоки

Приведём алгоритм, удовлетворяющий условиям 21,22. Обозначим за p_x число процессоров, работающих в ячейках по оси x , и p_y — по y .

Для гарантированного выполнения условия 22 возьмём последовательность размеров блоков по каждой оси, отличающихся только на 1 точку. Т.е. будем чередовать по оси x домены с размером a_x и $a_x + 1$ (по y соответственно обозначим за a_y и $a_y + 1$).

²Корректность гарантирована объяснениями преподавателей на лекции, посвящённой постановке задачи

Таким образом, получаем следующие разложения

$$\sum_{i=1}^{p_x} n_x(i) = k_1 a_x + k_2 (a_x + 1) = M, \quad k_1 + k_2 = p_x \Rightarrow a_x p_x + k_2 = M \quad (23)$$

$$\sum_{j=1}^{p_y} n_y(j) = s_1 a_y + s_2 (a_y + 1) = N, \quad s_1 + s_2 = p_y \Rightarrow a_y p_y + s_2 = N \quad (24)$$

Условие [21](#) в новых обозначениях формулируется как $0.5 \leq \frac{a_x}{a_y} \leq 2$, т.к. в приведённой формулировке алгоритма чередоваться бóльшие блоки будут одновременно по осям.

Из формул [23](#), [24](#) видно, что необходимо поделить с остатком M и N на количество процессоров по осям. При этом не стоит забывать, что должно соблюдаться неравенство $p_x p_y \leq p$.

Предлагается следующая схема: берём $p_x = 2^k$ процессоров на ось x и $p_y = \lfloor \frac{p}{2^k} \rfloor$ процессоров на ось y .³ Для выполнения условия [21](#) логичным кажется выбирать эти числа, исходя из пропорции [25](#).

$$\frac{2^k}{\frac{p}{2^k}} = \frac{M}{N} \Rightarrow \frac{2^{2k}}{p} = \frac{M}{N} \Rightarrow 2^{2k} = \frac{pM}{N} \Rightarrow k = \left\lfloor \frac{\log_2 \left(\frac{pM}{N} \right)}{2} \right\rfloor \quad (25)$$

В случае $p = 2^n$ выражение [25](#) переходит в $k = \left\lfloor \frac{n + \log_2 \left(\frac{M}{N} \right)}{2} \right\rfloor$.

Таким образом, итоговая схема получения блоков, удовлетворяющих условиям [21](#) (из-за выбора k согласно [25](#)) и [22](#) (из-за выбора определённой последовательности), выглядит следующим образом:

1. Задаём $p_x = 2^{\left\lfloor \frac{\log_2 \left(\frac{pM}{N} \right)}{2} \right\rfloor}$, $p_y = \left\lfloor \frac{p}{p_x} \right\rfloor$.
2. Получаем $a_x = \lfloor \frac{M}{p_x} \rfloor$, $a_y = \lfloor \frac{N}{p_y} \rfloor$ и фиксируем $k_2 = M \bmod p_x$, $s_2 = N \bmod p_y$
3. Далее генерируем $k_1 = p_x - k_2$ доменов по оси x с размером a_x и $s_1 = p_y - s_2$ доменов по оси y с размером a_y .
4. Затем начинается генерация k_2 доменов по оси x с размером $a_x + 1$ и s_2 доменов по оси y с размером $a_y + 1$

³Данная формула носит общий характер, в рамках текущего задания $p = 2^n$ и можно сразу сказать, что $p_y = 2^{n-k}$

4.3 Реализация с использованием MPI и OpenMP

В рамках экспериментов была осуществлена реализация описанной разностной схемы с использованием MPI и OpenMP. Реализация написана на языке C. Классическая MPI реализация приведена в [1](#), с методом сопряжённых градиентов в [9](#).

Также (из-за неэффективности вычисления на системе Blue Gene/P) реализована схема оптимизации с помощью сопряжённых градиентов (реализация с MPI в приложении [2](#)). Метод сопряжённых градиентов (везде далее CG) может показывать нестабильность в случае несимметричной матрицы СЛАУ, результаты сравнения продемонстрированы в сводных таблицах. Также для сравнения приводится время вычисления решения на ноутбуке с 4 ядрами (8-ю логическими процессорами)⁴

Реализация с использованием OpenMP отличается добавлением в циклах директив `pragma omp parallel for`.

Коротко опишем ключевые этапы программы с MPI. С помощью `MPI_Cart_create` создаётся топология на прямоугольной сетке, совпадающей по реализации с разбиением, предлагаемом в разделе [4.2](#).

Далее проверяется, какая ошибка каждого блока по разностной схеме от истинного решения, известного заранее. После проверки начинается основной цикл оптимизации. Критерием останова является получение нормы относительной ошибки меньше заданного значения. В случае сопряжённых градиентов - получение отношения нормы остатков к норме матрицы правой части меньше заданного числа.

В цикле заполняются блоки, расширенные по одной точке по каждому направлению (для получения от соседних процессов информации об их граничных точках). Обмен происходит с помощью асинхронной отправки и синхронного получения.

Стоит отметить, что на локальном запуске не требовалось наличия функции `MPI_Wait()`, однако Blue Gene/P потребовал написания данной реализации из-за специфики архитектуры.

После этого вычисляется остаток на текущем шаге и перед получением произведения Ar также проводится необходимая синхронизация.

⁴Asus ZenBook BX433F, Processor Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99 GHz. Installed RAM 16.0 GB (15.8 GB usable). System type 64-bit operating system, x64-based processor.

Затем вычисляется коэффициент (коэффициенты, в случае сопряжённых градиентов) для оптимизации и делается шаг.

В конце программы происходит барьерная синхронизация и вывод результатов по времени на стандартный поток вывода и по данным в необходимые для визуализации файлы.

5 Результаты на системах Blue Gene/P и Polus

Для данной задачи выполнены подсчёты ускорения программы на системах Blue Gene/P и Polus.

Под ускорением программы, запущенной на p MPI-процессах, понимается величина:

$$S_p = \frac{T_m}{T_p}$$

где T_m — время работы на минимальном числе MPI-процессов, T_p — время работы программы на p MPI-процессах.

Результаты запусков стандартной реализации на системе Blue Gene/P с использованием MPI и OpenMP приведены в 2 и 3.

Результаты использования только MPI 2 показывают лишь замедление времени исполнения при увеличении числа узлов. Возможные варианты объяснения видятся следующие:

- Топология, создаваемая MPI_Cart, становится ресурсоёмкой при нескольких обменах в ходе исполнения программы
- Медленный метод сходимости
- Загрузка суперкомпьютера
- Компиляция без оптимизации под архитектуру

Также видна закономерность в том, что на бóльшей сетке метод оптимизации сходится быстрее⁵.

⁵Отметим, что чтобы уместиться в разрешённые временные рамки, была изменена начальная инициализация.

Число процессоров p	Сетка $M \times N$	Время T (мин)	Ускорение S
128	500×1000	5.031	1
256	500×1000	6.023	0.83
512	500×1000	9.889	0.51
128	1000×1000	1.975	1
256	1000×1000	2.341	0.84
512	1000×1000	3.488	0.57
(локально, CG) 4	500×1000	4.491	1.12
(локально, CG) 4	1000×1000	20.674	0.10

Таблица 2: Результаты расчетов MPI версии на ПВС IBM Blue Gene/P

Рассмотрим результаты для запусков программы, скомпилированной с OpenMP директивами.

Число процессоров p	Сетка $M \times N$	Время T (мин)	Ускорение S
128	500×1000	5.391	1
256	500×1000	3.214	1.68
512	500×1000	3.213	1.68
128	1000×1000	9.424	1
256	1000×1000	5.573	1.69
512	1000×1000	3.554	2.65
(локально, CG) 4	500×1000	8.905	0.61
(локально, CG) 4	1000×1000	11.094	0.85

Таблица 3: Результаты расчетов MPI+OpenMP версии на ПВС IBM Blue Gene/P

Для сравнения также приводится время, потраченное на локальном компьютере с 4 ядрами на тех же размерах сеток. Для корректности сравнения приведём результаты запуска метода сопряжённых градиентов на системе Blue Gene/P (см. 4, 5)

Число процессоров p	Сетка $M \times N$	Время T (мин)	Ускорение S
128	500×1000	0.824	1
256	500×1000	0.470	1.75
512	500×1000	0.470	1.75
128	1000×1000	1.856	1
256	1000×1000	1.053	1.76
512	1000×1000	0.608	3.05
(локально, CG) 4	500×1000	4.491	0.183
(локально, CG) 4	1000×1000	20.674	0.090

Таблица 4: Результаты расчетов MPI версии с CG на ПВС IBM Blue Gene/P

Число процессоров p	Сетка $M \times N$	Время T (мин)	Ускорение S
128	500×1000	5.742	1
256	500×1000	4.081	1.41
512	500×1000	3.089	1.86
128	1000×1000	0.422	1
256	1000×1000	0.310	1.36
512	1000×1000	3.495	0.121
(локально, CG) 4	500×1000	8.905	0.64
(локально, CG) 4	1000×1000	11.094	0.04

Таблица 5: Результаты расчетов MPI+OpenMP версии с CG на ПВС IBM Blue Gene/P

На системе ПВС IBM Polus с заданной точностью $1e - 6$ за 15 минут не удалось получить результаты по некоторым конфигурациям, поэтому проведено сравнение только метода с CG оптимизацией.

Число процессоров p	Сетка $M \times N$	Время T (мин)	Ускорение S
4	500×500	1.391	1
8	500×500	0.669	2.08
16	500×500	0.363	3.83
32	500×500	0.257	5.41
4	500×1000	4.569	1
8	500×1000	2.647	1.73
16	500×1000	1.465	3.12
32	500×1000	0.791	5.78
(локально, CG) 4	500×500	2.616	0.53
(локально, CG) 4	500×1000	4.491	1.02

Таблица 6: Таблица с результатами расчетов MPI версии (только CG) на ПВС IBM Polus

На основании таблиц 2 - 6 можно сделать следующие выводы.

На Blue Gene/P

- OpenMP версия показывает ускорение почти в два раза. Без указания директив *pragma* у параллельных циклов (результаты таблицы 2) нет должного ресурса параллелизма и при увеличении числа процессоров наблюдается увеличение времени.
- Реализация метода оптимизации сопряжённых направлений ускоряет вычисление в 5 раз (из сравнения 2 с 4)
- При добавлении директив OpenMP наблюдается увеличение времени работы. Особенно выделяется случай с 512 процессорами и сеткой размера 1000×1000 .

В случае прямоугольной сетки увеличение времени объясняется нарушением симметричности матрицы СЛАУ, что не позволяет воспользоваться преимуществами метода сопряжённых градиентов.

В случае с 512 процессорами увеличение времени работы можно объяснить тем, что размеры блоков (исходя из вывода 25 и дальнейшего алгоритма) различаются почти в 2 раза ($1000/32$ и $1000/16$), что является граничным случаем условий на домены и также нарушает симметрию.

На Polus

- Метод конечных разностей сходится явно дольше, чем на Blue Gene/P (наблюдалось достижение точности $6e-5$ только к 15 минуте подсчёта).
- Идеально демонстрируется ускорение за счёт взятия бóльшего числа процессоров (получение ускорения в 5-6 раз при взятии 32 процессоров).
- Вычислительная мощность на прямоугольной сетке совпадает (или даже уступает производительности ноутбука с приведённым выше описанием).

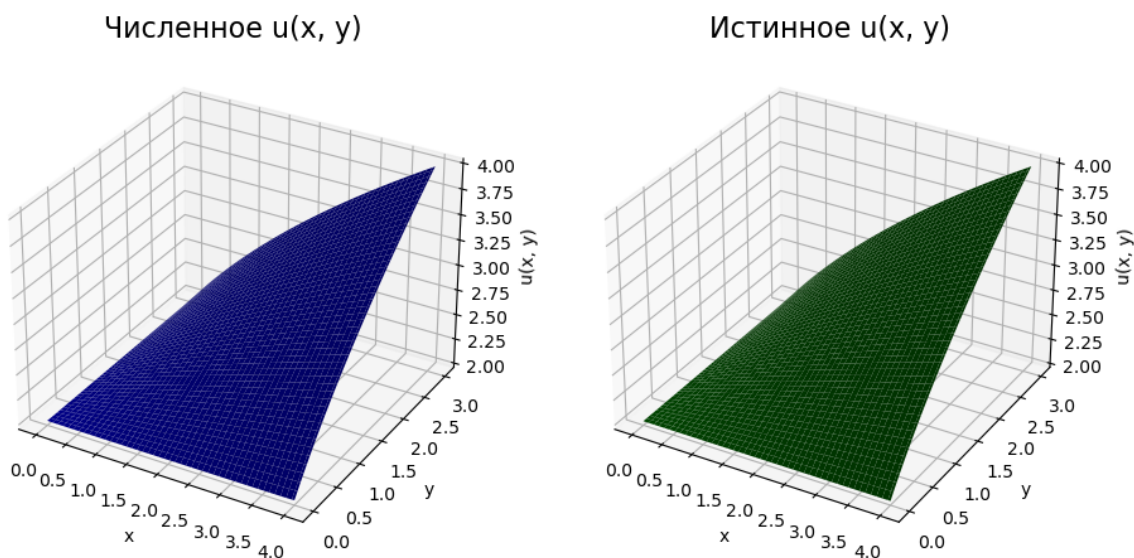


Рис. 1: Результат на сетке 500×1000

Итоговые результаты функций для сеток с наибольшим числом узлов приведены на рисунках 1, 2.

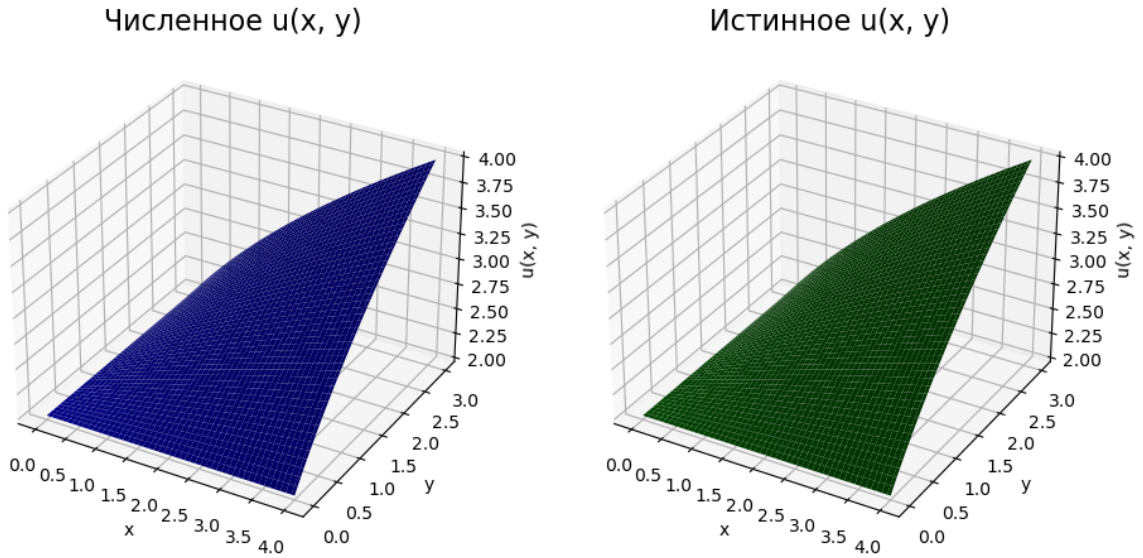


Рис. 2: Результат на сетке 1000×1000

6 Заключение

В ходе проведённых экспериментов была реализована программа для численного решения СЛАУ методом конечных разностей. Вычислительный эксперимент, с целью получения информативных результатов, дополнен сравнением с результатами, полученными методом сопряжённых градиентов и запуском на локальном компьютере.

Также описан аналитически метод, лежащий в основе разбиения на блоки `MPI_Cart_create`, получены аналитические выражения для правых частей и описана разностная схема.

Для представления результатов также использовались стандартные методы визуализации python, получающие на вход файлы, записанные каждым доменом (процессором) отдельно.

7 Литература

Источники

- [1] Бахвалов Н.С., Жидков Н.П., Кобельков Г.М. ЧИСЛЕННЫЕ МЕТОДЫ. - М.: Наука, 1987.

8 Приложение 1. Код MPI программы

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "mpi.h"
4 #include <time.h>
5 #include <math.h>
6 #include <unistd.h>
7
8
9 void print_matrix(int M, int N, double (*A)[N+2]){
10     printf("Matrix:\n");
11     for (int i=N+1; i>=0; i--){
12         for (int j=0; j<=M+1; j++){
13             printf("%3.2f ", A[i][j]);
14         }
15         printf("\n");
16     }
17 }
18
19 void print_matrix_to_file(FILE *file,
20                           int M, int N,
21                           double (*A)[N+2]){
22     for (int i=N+1; i>=0; i--){
23         for (int j=0; j<=M+1; j++){
24             fprintf(file, "%g ", A[i][j]);
25         }
26         fprintf(file, "\n");
27     }
28 }
29
30 double u_2(double x, double y){
31     return sqrt(4 + x * y);
32 }
33
34 double k_3(double x, double y){
35     return 4 + x + y;
36 }
37
38 double q_2(double x, double y){
39     double sum = x + y;
40     if (sum < 0) {
41         return 0;
42     } else {
43         return sum;
44     }
45 }
46
47 double F(double x, double y){
48     return ((pow(x, 3) - x*x*(y - 4) - x*(y*y + 8) +
49             y*(y*y + 4*y - 8) + 4*q_2(x, y)*pow((4 + x*y), 2)) /
50             (4 * pow((4 + x*y), 1.5)));
51 }
52
53 double psi_R(double x, double y){
54     return (y*(4 + x + y) + 2*(4 + x*y)) / (2*sqrt(4 + x*y));
55 }
56
57
58 double psi_L(double x, double y){
59     return (-y*(4 + x + y) + 2*(4 + x*y)) / (2*sqrt(4 + x*y));
60 }
```

```

61
62
63 double psi_T(double x, double y){
64     return (x*(4 + x + y)) / (2*sqrt(4 + x*y));
65 }
66
67
68 double psi_B(double x, double y){
69     return -psi_T(x, y);
70 }
71
72 double rho_1(int i,
73             int M,
74             int left_border,
75             int right_border){
76     if ((left_border && i == 1) || (right_border && i == M))
77         return 0.5;
78     return 1;
79 }
80
81 double rho_2(int j,
82             int N,
83             int bottom_border,
84             int top_border){
85     if ((bottom_border && j == 1) || (top_border && j == N))
86         return 0.5;
87     return 1;
88 }
89
90 double dot_product(int M, int N,
91                   double (*U)[N + 2], double (*V)[N + 2],
92                   double h1, double h2,
93                   int left_border, int right_border,
94                   int top_border, int bottom_border
95                   ){
96     double answer = 0.;
97     double rho, r1, r2;
98
99     for (int i=1; i <= M; i++){
100         for (int j=1; j <= N; j++){
101             r1 = rho_1(i, M, left_border, right_border);
102             r2 = rho_2(j, N, bottom_border, top_border);
103             rho = r1 * r2;
104             answer += (rho * U[i][j] * V[i][j] * h1 * h2);
105         }
106     }
107     return answer;
108 }
109
110
111 double norm(int M, int N, double (*U)[N + 2],
112            double h1, double h2,
113            int left_border, int right_border,
114            int top_border, int bottom_border){
115     return sqrt(dot_product(M, N, U, U, h1, h2,
116                             left_border, right_border,
117                             top_border, bottom_border));
118 }
119
120
121 void B_right(int M, int N, double (*B)[N+2],
122             double h1, double h2,

```

```

123     double x_start, double y_start,
124     int left_border, int right_border,
125     int top_border, int bottom_border){
126 int i, j;
127
128 for(i = 0; i <= M + 1; i++)
129     for (j = 0; j <= N + 1; j++)
130         B[i][j] = F(x_start + (i - 1) * h1, y_start + (j - 1) * h2);
131
132 if (left_border){
133     for (j = 1; j <= N; j++) {
134         B[1][j] = (F(x_start, y_start + (j - 1) * h2) +
135                 psi_L(x_start, y_start + (j - 1) * h2) * 2/h1);
136     }
137 }
138 if (right_border){
139     for (j = 1; j <= N; j++) {
140         B[M][j] = (F(x_start + (M - 1)*h1, y_start + (j - 1) * h2) +
141                 psi_R(x_start + (M - 1)*h1, y_start + (j - 1) * h2) * 2/
142                     h1);
143     }
144 }
145 if (top_border){
146     for (i = 1; i <= M; i++) {
147         B[i][N] = (F(x_start + (i - 1)*h1, y_start + (N - 1)*h2) +
148                 psi_T(x_start + (i - 1)*h1, y_start + (N - 1)*h2) * 2/h2
149                     );
150     }
151 }
152 if (bottom_border){
153     for (i = 1; i <= M; i++) {
154         B[i][1] = (F(x_start + (i - 1)*h1, y_start) +
155                 psi_B(x_start + (i - 1)*h1, y_start) * 2/h2);
156     }
157 }
158 if (left_border && top_border){
159     B[1][N] = (F(x_start, y_start + (N - 1)*h2) +
160             (2/h1 + 2/h2) * (psi_L(x_start, y_start + (N - 1)*h2) +
161                 psi_T(x_start, y_start + (N - 1)*h2)) / 2);
162 }
163 if (left_border && bottom_border){
164     B[1][1] = (F(x_start, y_start)
165             + (2/h1 + 2/h2) * (psi_L(x_start, y_start) + psi_B(x_start,
166                 y_start)) / 2);
167 }
168 if (right_border && top_border){
169     B[M][N] = (F(x_start + (M - 1)*h1, y_start + (N - 1)*h2) +
170             (2/h1 + 2/h2) * (psi_R(x_start + (M - 1)*h1, y_start + (N - 1)
171                 *h2) +
172                 psi_T(x_start + (M - 1)*h1, y_start + (N - 1)*
173                     h2)) / 2);
174 }
175 if (right_border && bottom_border){
176     B[M][1] = (F(x_start + (M - 1)*h1, y_start) +
177             (2/h1 + 2/h2) * (psi_R(x_start + (M - 1)*h1, y_start) +
178                 psi_B(x_start + (M - 1)*h1, y_start)) / 2);
179 }
180 }
181
182 double aw_x_ij(int N,

```

```

180         double (*w)[N+2],
181         double x_start, double y_start,
182         int i, int j,
183         double h1, double h2
184     ){
185         return (1/h1) * (k_3(x_start + (i + 0.5 - 1) * h1, y_start + (j - 1) * h2)
186             * (w[i + 1][j] - w[i][j]) / h1
187             - k_3(x_start + (i - 0.5 - 1) * h1, y_start + (j - 1) * h2) *
188                 (w[i][j] - w[i - 1][j]) / h1);
189     }
190
191     double aw_ij(int N,
192         double (*w)[N+2],
193         double x_start, double y_start,
194         int i, int j,
195         double h1, double h2
196     ){
197         return (k_3(x_start + (i - 0.5 - 1) * h1, y_start + (j - 1) * h2) * (w[i][
198             j] - w[i - 1][j]) / h1);
199     }
200
201     double bw_y_ij(int N,
202         double (*w)[N+2],
203         double x_start, double y_start,
204         int i, int j,
205         double h1, double h2
206     ){
207         return (1/h2) * (k_3(x_start + (i - 1) * h1, y_start + (j + 0.5 - 1) * h2)
208             * (w[i][j + 1] - w[i][j]) / h2
209             - k_3(x_start + (i - 1) * h1, y_start + (j - 0.5 - 1) * h2) *
210                 (w[i][j] - w[i][j - 1]) / h2);
211     }
212
213     double bw_ij(int N,
214         double (*w)[N+2],
215         double x_start, double y_start,
216         int i, int j,
217         double h1, double h2
218     ){
219         return (k_3(x_start + (i - 1) * h1, y_start + (j - 0.5 - 1) * h2) * (w[i][
220             j] - w[i][j-1]) / h2);
221     }
222
223     void Aw_mult(int M, int N,
224         double (*A)[N+2], double (*w)[N+2],
225         double h1, double h2,
226         double x_start, double y_start,
227         int left_border, int right_border,
228         int top_border, int bottom_border
229     )
230     {
231         double aw_x, bw_y;
232         int i, j;
233         for (i = 0; i <= M+1; i++){
234             for (j = 0; j <= N+1; j++){
235                 if ((i == 0) || i == M+1 || j == 0 || j == N+1){
236                     A[i][j] = w[i][j];
237                 } else {
238                     aw_x = aw_x_ij(N, w, x_start, y_start, i, j, h1, h2);
239                     bw_y = bw_y_ij(N, w, x_start, y_start, i, j, h1, h2);
240                     A[i][j] = -aw_x - bw_y + q_2(x_start + (i - 1) * h1,
241                         y_start + (j - 1) * h2) * w[i][j];
242                 }
243             }
244         }
245     }

```

```

236     }
237 }
238 }
239
240 // Left interior border filling
241 if (left_border){
242     for (j = 1; j <= N; j++) {
243         aw_x = aw_ij(N, w, x_start, y_start, 2, j, h1, h2);
244         bw_y = bw_ij(N, w, x_start, y_start, 1, j, h1, h2);
245         A[1][j] = -2*aw_x / h1 - bw_y + (q_2(x_start,
246                                     y_start + (j - 1) * h2) + 2/h1) *
                                     w[1][j];
247     }
248 }
249
250 // Right interior border
251 if (right_border){
252     for (j = 1; j <= N; j++) {
253         aw_x = aw_ij(N, w, x_start, y_start, M, j, h1, h2);
254         bw_y = bw_ij(N, w, x_start, y_start, M, j, h1, h2);
255         A[M][j] = 2*aw_x / h1 - bw_y + (q_2(x_start + (M - 1) * h1,
256                                     y_start + (j - 1) * h2) + 2/h1) *
                                     w[M][j];
257     }
258 }
259
260 // Top border
261 if (top_border){
262     for (i = 1; i <= M; i++) {
263         aw_x = aw_x_ij(N, w, x_start, y_start, i, N, h1, h2);
264         bw_y = bw_ij(N, w, x_start, y_start, i, N, h1, h2);
265         A[i][N] = -aw_x + 2*bw_y / h2 + q_2(x_start + (i - 1) * h1,
266                                     y_start + (N - 1) * h2) * w[i][N];
267     }
268 }
269
270 // Bottom border
271 if (bottom_border){
272     for (i = 1; i <= M; i++) {
273         aw_x = aw_x_ij(N, w, x_start, y_start, i, 1, h1, h2);
274         bw_y = bw_ij(N, w, x_start, y_start, i, 2, h1, h2);
275         A[i][1] = -aw_x - 2*bw_y / h2 + q_2(x_start + (i - 1)* h1, y_start
276                                     ) * w[i][1];
277     }
278 }
279 if (left_border && bottom_border){
280     aw_x = aw_ij(N, w, x_start, y_start, 2, 1, h1, h2);
281     bw_y = bw_ij(N, w, x_start, y_start, 1, 2, h1, h2);
282     A[1][1] = -2*aw_x / h1 - 2*bw_y / h2 + (q_2(x_start, y_start) + 2/h1)
283             * w[1][1];
284 }
285 if (left_border && top_border){
286     aw_x = aw_ij(N, w, x_start, y_start, 2, N, h1, h2);
287     bw_y = bw_ij(N, w, x_start, y_start, 1, N, h1, h2);
288     A[1][N] = -2*aw_x / h1 + 2*bw_y / h2 + (q_2(x_start, y_start + (N - 1)
289             * h2) + 2/h1)* w[1][N];
290 }
291 if (right_border && bottom_border){
292     aw_x = aw_ij(N, w, x_start, y_start, M, 1, h1, h2);
293     bw_y = bw_ij(N, w, x_start, y_start, M, 2, h1, h2);

```

```

292     A[M][1] = 2*aw_x / h1 - 2 * bw_y / h2 + (q_2(x_start + (M - 1) * h1,
293         y_start) + 2/h1) * w[M][1];
294 }
295 if (right_border && top_border) {
296     aw_x = aw_ij(N, w, x_start, y_start, M, N, h1, h2);
297     bw_y = bw_ij(N, w, x_start, y_start, M, N, h1, h2);
298     A[M][N] = 2*aw_x / h1 + 2 * bw_y / h2 + (q_2(x_start + (M - 1) * h1,
        y_start + (N - 1) * h2) + 2/
        h1) * w[M][N];
299 }
300 }
301
302
303 void calculate_r(int M, int N,
304     double (*r)[N+2],
305     double (*Aw)[N+2],
306     double (*B)[N+2]
307 ){
308     int i, j;
309
310     for(i = 0; i <= M + 1; i++) {
311         for (j = 0; j <= N + 1; j++) {
312             if(i == 0 || i == M+1 || j == 0 || j == N+1)
313                 r[i][j] = 0;
314             else
315                 r[i][j] = Aw[i][j] - B[i][j];
316         }
317     }
318 }
319
320
321 void get_idx_n_idx(int *idx,
322     int *n_idx,
323     int process_amnt,
324     int grid_size,
325     int coordinate){
326     if (grid_size % process_amnt == 0) {
327         *n_idx = grid_size / process_amnt;
328         *idx = coordinate * (grid_size / process_amnt);
329     }
330     else
331     {
332         if (coordinate == 0){
333             *n_idx = grid_size % process_amnt + grid_size / process_amnt;
334             *idx = 0;
335         } else
336         {
337             *n_idx = grid_size / process_amnt;
338             *idx = grid_size % process_amnt + coordinate * (grid_size /
                process_amnt);
339         }
340     }
341 }
342
343
344 #define A1 0.0
345 #define A2 4.0
346 #define B1 0.0
347 #define B2 3.0
348 #define EPS_REL 1e-6
349 #define DOWN_TAG 1000
350 #define MAX_ITER 100000

```



```

351
352
353 void send_rcv_borders(int n_x, int n_y,
354                       const int process_amounts[2],
355                       double x_idx,
356                       double y_idx,
357                       const int my_coords[2],
358                       int tag,
359                       double (*w)[n_y+2],
360                       double b_send[n_x],
361                       double l_send[n_y],
362                       double t_send[n_x],
363                       double r_send[n_y],
364                       double b_rec[n_x],
365                       double l_rec[n_y],
366                       double t_rec[n_x],
367                       double r_rec[n_y],
368                       int left_border, int right_border,
369                       int top_border, int bottom_border,
370                       double h1, double h2,
371                       MPI_Comm MPI_COMM_CART
372                       ){
373
374     int i, j;
375     int neighbour_coords[2];
376     int neighbour_rank;
377     MPI_Request request[4] = {MPI_REQUEST_NULL, MPI_REQUEST_NULL,
378                               MPI_REQUEST_NULL, MPI_REQUEST_NULL};
379     MPI_Status status;
380
381     // Bottom border send
382     if ((process_amounts[1] > 1) && !bottom_border) {
383       for (i = 0; i < n_x; i++)
384         b_send[i] = w[i+1][1];
385
386       neighbour_coords[0] = my_coords[0];
387       neighbour_coords[1] = my_coords[1] - 1;
388
389       MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
390       MPI_Isend(b_send, n_x, MPI_DOUBLE,
391                neighbour_rank, tag + DOWN_TAG,
392                MPI_COMM_CART, &request[0]);
393     }
394
395     // Left border send
396     if ((process_amounts[0] > 1) && !left_border) {
397       for (j = 0; j < n_y; j++)
398         l_send[j] = w[1][j+1];
399
400       neighbour_coords[0] = my_coords[0] - 1;
401       neighbour_coords[1] = my_coords[1];
402
403       MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
404       MPI_Isend(l_send, n_y, MPI_DOUBLE,
405                neighbour_rank, tag,
406                MPI_COMM_CART, &request[1]);
407     }
408
409     // Top border
410     if ((process_amounts[1] > 1) && !top_border) {
411       for (i = 0; i < n_x; i++)
412         t_send[i] = w[i+1][n_y];

```

```

412     neighbour_coords[0] = my_coords[0];
413     neighbour_coords[1] = my_coords[1] + 1;
414
415     MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
416     MPI_Isend(t_send, n_x, MPI_DOUBLE,
417              neighbour_rank, tag,
418              MPI_COMM_CART, &request[2]);
419 }
420
421 // Right border
422 if ((process_amounts[0] > 1) && !right_border) {
423     for (j = 0; j < n_y; j++)
424         r_send[j] = w[n_x][j+1];
425
426     neighbour_coords[0] = my_coords[0] + 1;
427     neighbour_coords[1] = my_coords[1];
428
429     MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
430     MPI_Isend(r_send, n_y, MPI_DOUBLE,
431              neighbour_rank, tag,
432              MPI_COMM_CART, &request[3]);
433 }
434
435 // Receive borders
436 // Bottom border
437 if ((bottom_border && (process_amounts[1] > 1)) || (process_amounts[1] ==
438     1)) {
439     for (i = 1; i <= n_x; i++)
440         w[i][0] = u_2(A1 + (x_idx + i - 1) * h1, B1 + (y_idx - 1) * h2);
441 } else {
442     neighbour_coords[0] = my_coords[0];
443     neighbour_coords[1] = my_coords[1] - 1;
444     MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
445     MPI_Recv(b_rec, n_x, MPI_DOUBLE,
446             neighbour_rank, tag, MPI_COMM_CART, &status);
447
448     for (i = 1; i <= n_x; i++)
449         w[i][0] = b_rec[i - 1];
450 }
451
452 // Left border
453 if ((left_border && (process_amounts[0] > 1)) || (process_amounts[0] ==
454     1)) {
455     for (j = 1; j <= n_y; j++){
456         w[0][j] = u_2(A1 + (x_idx - 1) * h1, B1 + (y_idx + j - 1) * h2);
457     }
458 } else {
459     neighbour_coords[0] = my_coords[0] - 1;
460     neighbour_coords[1] = my_coords[1];
461
462     MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
463     MPI_Recv(l_rec, n_y, MPI_DOUBLE,
464             neighbour_rank, tag, MPI_COMM_CART, &status);
465
466     for (j = 1; j <= n_y; j++)
467         w[0][j] = l_rec[j - 1];
468 }
469
470 // Top border

```

```

471     if ((top_border && (process_amounts[1] > 1)) || (process_amounts[1] == 1)
472         ) {
473         for (i = 1; i <= n_x; i++)
474             w[i][n_y + 1] = u_2(A1 + (x_idx + i - 1) * h1, B1 + (y_idx + n_y)
475                 * h2);
476     } else {
477         neighbour_coords[0] = my_coords[0];
478         neighbour_coords[1] = my_coords[1] + 1;
479         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
480         MPI_Recv(t_rec, n_x, MPI_DOUBLE,
481             neighbour_rank, tag + DOWN_TAG,
482             MPI_COMM_CART, &status);
483         for (i = 1; i <= n_x; i++)
484             w[i][n_y + 1] = t_rec[i - 1];
485     }
486     // Right border
487     if ((right_border && (process_amounts[0] > 1)) || (process_amounts[0] ==
488         1)) {
489         for (j = 1; j <= n_y; j++)
490             w[n_x + 1][j] = u_2(A1 + (x_idx + n_x)*h1, B1 + (y_idx + j - 1) *
491                 h2);
492     } else {
493         neighbour_coords[0] = my_coords[0] + 1;
494         neighbour_coords[1] = my_coords[1];
495         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
496         MPI_Recv(r_rec, n_y, MPI_DOUBLE,
497             neighbour_rank, tag, MPI_COMM_CART, &status);
498         for (j = 1; j <= n_y; j++)
499             w[n_x + 1][j] = r_rec[j - 1];
500     }
501     for (int i = 0; i < 4; i++) {
502         MPI_Wait(&request[i], &status);
503     }
504 }
505
506 int main(int argc, char *argv[]) {
507     if (argc != 3) {
508         printf("Program receive %d numbers. Should be 2: M, N\n", argc);
509         return -1;
510     }
511
512     int M = atoi(argv[argc - 2]);
513     int N = atoi(argv[argc - 1]);
514     if ((M <= 0) || (N <= 0)) {
515         printf("M and N should be integer and > 0!!!\n");
516         return -1;
517     }
518     printf("M = %d, N = %d\n", M, N);
519
520     int my_rank;
521     int n_processes;
522     int process_amounts[2] = {0, 0};
523     int write[1] = {0};
524
525     double h1 = (A2 - A1) / M;
526     double h2 = (B2 - B1) / N;
527     double cur_eps = 1.0;
528

```

```

529 MPI_Init(&argc, &argv);
530 MPI_Status status;
531 MPI_Request request;
532
533 // For the cartesian topology
534 MPI_Comm MPI_COMM_CART;
535 MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
536 MPI_Comm_size(MPI_COMM_WORLD, &n_processes);
537
538 // Creating rectangular supports
539 MPI_Dims_create(n_processes, 2, process_amounts);
540
541 printf("p_x = %d, p_y = %d\n", process_amounts[0], process_amounts[1]);
542 int periods[2] = {0, 0};
543
544 // Create cartesian topology in communicator
545 MPI_Cart_create(MPI_COMM_WORLD, 2,
546                process_amounts, periods,
547                1, &MPI_COMM_CART);
548
549 int my_coords[2];
550 // Receive corresponding to rank process coordinates
551 MPI_Cart_coords(MPI_COMM_CART, my_rank, 2, my_coords);
552
553 int x_idx, n_x;
554 get_idx_n_idx(&x_idx, &n_x, process_amounts[0], M+1, my_coords[0]);
555
556 int y_idx, n_y;
557 get_idx_n_idx(&y_idx, &n_y, process_amounts[1], N+1, my_coords[1]);
558
559 double start_time = MPI_Wtime();
560
561 // Create each block of size n_x and n_y with borders
562 double *t_send = malloc(sizeof(double[n_x]));
563 double *t_rec = malloc(sizeof(double[n_x]));
564 double *b_send = malloc(sizeof(double[n_x]));
565 double *b_rec = malloc(sizeof(double[n_x]));
566
567 double *l_send = malloc(sizeof(double[n_y]));
568 double *l_rec = malloc(sizeof(double[n_y]));
569 double *r_send = malloc(sizeof(double[n_y]));
570 double *r_rec = malloc(sizeof(double[n_y]));
571
572 int i, j;
573 int n_iters = 0;
574 double block_eps;
575
576 double (*w)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
577 double (*w_pr)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
578 double (*B)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
579 double tau = 0;
580 double global_tau = 0;
581 // double alpha_k, beta_k;
582 double denominator;
583 double whole_denum;
584 double global_alpha, global_beta;
585 double eps_local, eps_r;
586
587 double (*Aw)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
588 double (*r_k)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
589 double (*Ar)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
590 double (*w_w_pr)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));

```

```

591
592 int left_border = 0;
593 int top_border = 0;
594 int right_border = 0;
595 int bottom_border = 0;
596
597
598 if (my_coords[0] == 0)
599     left_border = 1;
600
601 if (my_coords[0] == (process_amounts[0] - 1))
602     right_border = 1;
603
604 if (my_coords[1] == 0)
605     bottom_border = 1;
606
607 if (my_coords[1] == (process_amounts[1] - 1))
608     top_border = 1;
609
610 printf("L%d, R%d, T%d, B%d, 'x'%d, 'y'%d\n",
611        left_border, right_border, top_border, bottom_border, my_coords[0],
612        my_coords[1]);
613
614 printf("%d %d\n", x_idx, y_idx);
615
616 double (*Au)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
617 double (*U)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
618 for (i = 0; i <= n_x + 1; i++)
619     for (j = 0; j <= n_y + 1; j++)
620         U[i][j] = u_2(A1 + (x_idx + i - 1) * h1, B1 + (y_idx + j - 1) * h2
621            );
622
623 Aw_mult(n_x, n_y, Au, U, h1, h2,
624         A1 + x_idx * h1, B1 + y_idx * h2,
625         left_border, right_border,
626         top_border, bottom_border);
627
628 B_right(n_x, n_y, B,
629         h1, h2,
630         A1 + x_idx * h1,
631         B1 + y_idx * h2,
632         left_border, right_border,
633         top_border, bottom_border);
634
635 double error_mean = 0;
636 int amnt = 0;
637 for (i = 1; i <= n_x; i++)
638     for (j = 1; j <= n_y; j++){
639         error_mean += fabs(Au[i][j] - B[i][j]);
640         amnt += 1;
641     }
642 printf("ERROR FROM B = %3.2f\n", error_mean / amnt);
643
644 for (i = 0; i <= n_x + 1; i++)
645     for (j = 0; j <= n_y + 1; j++)
646         w[i][j] = 0; // u_2(A1 + (x_idx + i - 1) * h1, B1 + (y_idx + j -
647            1) * h2);
648
649 int tag = 0;
650 while ((cur_eps > EPS_REL) && (n_iters < MAX_ITER)) {
651     if (my_rank == 0) {
652         if (n_iters % 1000 == 0)
653             printf("%g \n", cur_eps);

```

```

650     }
651     n_iters++;
652
653     for (i = 0; i <= n_x + 1; i++) {
654         for (j = 0; j <= n_y + 1; j++) {
655             if (i == 0 || j == 0 || i == n_x + 1 || j == n_y + 1) {
656                 w_pr[i][j] = 0;
657             } else {
658                 w_pr[i][j] = w[i][j];
659             }
660         }
661     }
662
663     send_recv_borders(n_x, n_y, process_amounts,
664                     x_idx, y_idx, my_coords, tag,
665                     w,
666                     b_send, l_send, t_send, r_send,
667                     b_rec, l_rec, t_rec, r_rec,
668                     left_border, right_border,
669                     top_border, bottom_border,
670                     h1, h2, MPI_COMM_CART);
671
672     Aw_mult(n_x, n_y,
673            Aw, w,
674            h1, h2,
675            A1 + x_idx * h1, B1 + y_idx * h2,
676            left_border, right_border,
677            top_border, bottom_border);
678
679     calculate_r(n_x, n_y, r_k, Aw, B);
680
681     send_recv_borders(n_x, n_y, process_amounts,
682                     x_idx, y_idx, my_coords, tag,
683                     r_k,
684                     b_send, l_send, t_send, r_send,
685                     b_rec, l_rec, t_rec, r_rec,
686                     left_border, right_border,
687                     top_border, bottom_border,
688                     h1, h2, MPI_COMM_CART);
689
690     Aw_mult(n_x, n_y,
691            Ar, r_k,
692            h1, h2,
693            A1 + x_idx * h1, B1 + y_idx * h2,
694            left_border, right_border,
695            top_border, bottom_border);
696
697     tau = dot_product(n_x, n_y, Ar, r_k, h1, h2,
698                     left_border, right_border,
699                     top_border, bottom_border
700                     );
701     denominator = dot_product(n_x, n_y, Ar, Ar, h1, h2,
702                             left_border, right_border,
703                             top_border, bottom_border);
704     MPI_Allreduce(&tau, &global_tau, 1,
705                 MPI_DOUBLE, MPI_SUM, MPI_COMM_CART);
706     MPI_Allreduce(&denominator, &whole_denum, 1,
707                 MPI_DOUBLE, MPI_SUM, MPI_COMM_CART);
708     global_tau = global_tau / whole_denum;
709
710
711     for (i = 1; i <= n_x; i++)

```

```

712         for (j = 1; j <= n_y; j++) {
713             w[i][j] = w[i][j] - global_tau * r_k[i][j];
714         }
715
716         calculate_r(n_x, n_y, w_w_pr, w, w_pr);
717         block_eps = norm(n_x, n_y, w_w_pr, h1, h2,
718                         left_border, right_border,
719                         top_border, bottom_border);
720
721         MPI_Allreduce(&block_eps, &cur_eps, 1,
722                     MPI_DOUBLE, MPI_SUM, MPI_COMM_CART);
723     }
724
725     // Waiting for all processes
726     MPI_Barrier(MPI_COMM_WORLD);
727     double end_time = MPI_Wtime();
728
729     if (my_rank != 0) {
730         MPI_Recv(write, 1, MPI_INT, my_rank - 1, 0, MPI_COMM_WORLD, &status);
731     } else {
732         printf("TIME = %f\n", end_time - start_time);
733         printf("Number of iterations = %d\n", n_iters);
734         printf("Tau = %f\n", tau);
735         printf("Eps = %f\n", EPS_REL);
736     }
737
738     // usleep(500);
739     if (my_rank != n_processes - 1)
740         MPI_Send(write, 1, MPI_INT, my_rank + 1, 0, MPI_COMM_WORLD);
741
742     FILE *dim0, *dim1, *grid, *u_file, *true_u_file;
743     char u_file_name[FILENAME_MAX];
744     sprintf(u_file_name, "u_%d_%d.csv", my_coords[0], my_coords[1]);
745     char true_u_file_name[FILENAME_MAX];
746     sprintf(true_u_file_name, "true.u_%d_%d.csv", my_coords[0], my_coords[1])
747         ;
748
749     dim0 = fopen("dim0.csv", "w");
750     dim1 = fopen("dim1.csv", "w");
751     grid = fopen("grid.csv", "w");
752     u_file = fopen(u_file_name, "w");
753     true_u_file = fopen(true_u_file_name, "w");
754
755     for (int j = y_idx; j < y_idx + n_y; j++) {
756         for (int i = x_idx; i < x_idx + n_x; i++) {
757             fprintf(u_file, "%g ", w[i - x_idx + 1][j - y_idx + 1]);
758             fprintf(true_u_file, "%g ", u_2(A1 + i*h1, B1 + j*h2));
759         }
760         fprintf(u_file, "\n");
761         fprintf(true_u_file, "\n");
762     }
763
764     if (my_rank == 0) {
765         for (int j = 0; j <= N; j++) {
766             fprintf(dim0, "%g ", B1 + j*h2);
767         }
768
769         for (int i = 0; i <= M; i++) {
770             fprintf(dim1, "%g ", A1 + i*h1);
771         }
772         fprintf(grid, "%d %d", process_amounts[1], process_amounts[0]);

```

```

773     }
774     fclose(dim0);
775     fclose(dim1);
776     fclose(grid);
777     fclose(u_file);
778     fclose(true_u_file);
779
780     free(Au);
781     free(U);
782     free(w);
783     free(w_pr);
784     free(B);
785     free(Ar);
786     free(r_k);
787     free(Aw);
788     free(w_w_pr);
789
790     free(t_send);
791     free(t_rec);
792     free(b_send);
793     free(b_rec);
794     free(r_send);
795     free(r_rec);
796     free(l_send);
797     free(l_rec);
798     MPI_Finalize();
799     return 0;
800 }

```

Листинг 1: neyman_pde_mpi.c

9 Приложение 2. Код MPI программы с оптимизацией сопряжёнными градиентами

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "mpi.h"
4  #include <time.h>
5  #include <math.h>
6  #include <unistd.h>
7
8
9  void print_matrix(int M, int N, double (*A)[N+2]){
10     printf("Matrix:\n");
11     for (int i=N+1; i>=0; i--){
12         for (int j=0; j<=M+1; j++){
13             printf("%3.2f ", A[i][j]);
14         }
15         printf("\n");
16     }
17 }
18
19 void print_matrix_to_file(FILE *file,
20                           int M, int N,
21                           double (*A)[N+2]){
22     for (int i=N+1; i>=0; i--){
23         for (int j=0; j<=M+1; j++){

```



```

24         fprintf(file, "%g ", A[i][j]);
25     }
26     fprintf(file, "\n");
27 }
28 }
29
30 double u_2(double x, double y){
31     return sqrt(4 + x * y);
32 }
33
34 double k_3(double x, double y){
35     return 4 + x + y;
36 }
37
38 double q_2(double x, double y){
39     double sum = x + y;
40     if (sum < 0) {
41         return 0;
42     } else {
43         return sum;
44     }
45 }
46
47 double F(double x, double y){
48     return ((pow(x, 3) - x*x*(y - 4) - x*(y*y + 8) +
49     y*(y*y + 4*y - 8) + 4*q_2(x, y)*pow((4 + x*y), 2)) /
50     (4 * pow((4 + x*y), 1.5)));
51 }
52
53 double psi_R(double x, double y){
54     return (y*(4 + x + y) + 2*(4 + x*y)) / (2*sqrt(4 + x*y));
55 }
56
57
58 double psi_L(double x, double y){
59     return (-y*(4 + x + y) + 2*(4 + x*y)) / (2*sqrt(4 + x*y));
60 }
61
62
63 double psi_T(double x, double y){
64     return (x*(4 + x + y)) / (2*sqrt(4 + x*y));
65 }
66
67
68 double psi_B(double x, double y){
69     return -psi_T(x, y);
70 }
71
72 double rho_1(int i,
73             int M,
74             int left_border,
75             int right_border){
76     if ((left_border && i == 1) || (right_border && i == M))
77         return 0.5;
78     return 1;
79 }
80
81 double rho_2(int j,
82             int N,
83             int bottom_border,
84             int top_border){
85     if ((bottom_border && j == 1) || (top_border && j == N))

```

```

86         return 0.5;
87     return 1;
88 }
89
90 double dot_product(int M, int N,
91                   double (*U)[N + 2], double (*V)[N + 2],
92                   double h1, double h2,
93                   int left_border, int right_border,
94                   int top_border, int bottom_border
95                   ){
96     double answer = 0.;
97     double rho, r1, r2;
98
99     for (int i=1; i <= M; i++){
100         for (int j=1; j <= N; j++){
101             r1 = rho_1(i, M, left_border, right_border);
102             r2 = rho_2(j, N, bottom_border, top_border);
103             rho = r1 * r2;
104             answer += (rho * U[i][j] * V[i][j] * h1 * h2);
105         }
106     }
107     return answer;
108 }
109
110
111 double norm(int M, int N, double (*U)[N + 2],
112            double h1, double h2,
113            int left_border, int right_border,
114            int top_border, int bottom_border){
115     return sqrt(dot_product(M, N, U, U, h1, h2,
116                            left_border, right_border,
117                            top_border, bottom_border));
118 }
119
120
121 void B_right(int M, int N, double (*B)[N+2],
122             double h1, double h2,
123             double x_start, double y_start,
124             int left_border, int right_border,
125             int top_border, int bottom_border){
126     int i, j;
127
128     for(i = 0; i <= M + 1; i++)
129         for (j = 0; j <= N + 1; j++)
130             B[i][j] = F(x_start + (i - 1) * h1, y_start + (j - 1) * h2);
131
132     if (left_border){
133         for (j = 1; j <= N; j++) {
134             B[1][j] = (F(x_start, y_start + (j - 1) * h2) +
135                      psi_L(x_start, y_start + (j - 1) * h2) * 2/h1);
136         }
137     }
138     if (right_border){
139         for (j = 1; j <= N; j++) {
140             B[M][j] = (F(x_start + (M - 1)*h1, y_start + (j - 1) * h2) +
141                      psi_R(x_start + (M - 1)*h1, y_start + (j - 1) * h2) * 2/
142                          h1);
143         }
144     }
145     if (top_border){
146         for (i = 1; i <= M; i++) {
147             B[i][N] = (F(x_start + (i - 1)*h1, y_start + (N - 1)*h2) +

```

```

147         psi_T(x_start + (i - 1)*h1, y_start + (N - 1)*h2) * 2/h2
148     );
149 }
150
151 if (bottom_border){
152     for (i = 1; i <= M; i++) {
153         B[i][1] = (F(x_start + (i - 1)*h1, y_start) +
154                 psi_B(x_start + (i - 1)*h1, y_start) * 2/h2);
155     }
156 }
157 if (left_border && top_border){
158     B[1][N] = (F(x_start, y_start + (N - 1)*h2) +
159             (2/h1 + 2/h2) * (psi_L(x_start, y_start + (N - 1)*h2) +
160                 psi_T(x_start, y_start + (N - 1)*h2)) / 2);
161 }
162 if (left_border && bottom_border){
163     B[1][1] = (F(x_start, y_start)
164             + (2/h1 + 2/h2) * (psi_L(x_start, y_start) + psi_B(x_start,
165                 y_start)) / 2);
166 }
167 if (right_border && top_border){
168     B[M][N] = (F(x_start + (M - 1)*h1, y_start + (N - 1)*h2) +
169             (2/h1 + 2/h2) * (psi_R(x_start + (M - 1)*h1, y_start + (N - 1)
170                 *h2) +
171                 psi_T(x_start + (M - 1)*h1, y_start + (N - 1)*
172                     h2)) / 2);
173 }
174 if (right_border && bottom_border){
175     B[M][1] = (F(x_start + (M - 1)*h1, y_start) +
176             (2/h1 + 2/h2) * (psi_R(x_start + (M - 1)*h1, y_start) +
177                 psi_B(x_start + (M - 1)*h1, y_start)) / 2);
178 }
179 }
180
181 double aw_x_ij(int N,
182               double (*w)[N+2],
183               double x_start, double y_start,
184               int i, int j,
185               double h1, double h2
186           ){
187     return (1/h1) * (k_3(x_start + (i + 0.5 - 1) * h1, y_start + (j - 1) * h2)
188         * (w[i + 1][j] - w[i][j]) / h1
189         - k_3(x_start + (i - 0.5 - 1) * h1, y_start + (j - 1) * h2) *
190             (w[i][j] - w[i - 1][j]) / h1);
191 }
192
193 double aw_ij(int N,
194             double (*w)[N+2],
195             double x_start, double y_start,
196             int i, int j,
197             double h1, double h2
198         ){
199     return (k_3(x_start + (i - 0.5 - 1) * h1, y_start + (j - 1) * h2) * (w[i][
200         j] - w[i - 1][j]) / h1);
201 }
202
203 double bw_y_ij(int N,
204               double (*w)[N+2],
205               double x_start, double y_start,
206               int i, int j,

```

```

202         double h1, double h2
203     ){
204         return (1/h2) * (k_3(x_start + (i - 1) * h1, y_start + (j + 0.5 - 1) * h2)
205             * (w[i][j + 1] - w[i][j]) / h2
206             - k_3(x_start + (i - 1) * h1, y_start + (j - 0.5 - 1) * h2) *
207                 (w[i][j] - w[i][j - 1]) / h2);
208     }
209
210     double bw_ij(int N,
211         double (*w)[N+2],
212         double x_start, double y_start,
213         int i, int j,
214         double h1, double h2
215     ){
216         return (k_3(x_start + (i - 1) * h1, y_start + (j - 0.5 - 1) * h2) * (w[i][
217             j] - w[i][j-1]) / h2);
218     }
219
220     void Aw_mult(int M, int N,
221         double (*A)[N+2], double (*w)[N+2],
222         double h1, double h2,
223         double x_start, double y_start,
224         int left_border, int right_border,
225         int top_border, int bottom_border
226     )
227     {
228         double aw_x, bw_y;
229         int i, j;
230         for (i = 0; i <= M+1; i++){
231             for (j = 0; j <= N+1; j++) {
232                 if ((i == 0) || i == M+1 || j == 0 || j == N+1){
233                     A[i][j] = w[i][j];
234                 } else {
235                     aw_x = aw_x_ij(N, w, x_start, y_start, i, j, h1, h2);
236                     bw_y = bw_y_ij(N, w, x_start, y_start, i, j, h1, h2);
237                     A[i][j] = -aw_x - bw_y + q_2(x_start + (i - 1) * h1,
238                         y_start + (j - 1) * h2) * w[i][j];
239                 }
240             }
241         }
242
243         // Left interior border filling
244         if (left_border){
245             for (j = 1; j <= N; j++) {
246                 aw_x = aw_ij(N, w, x_start, y_start, 2, j, h1, h2);
247                 bw_y = bw_y_ij(N, w, x_start, y_start, 1, j, h1, h2);
248                 A[1][j] = -2*aw_x / h1 - bw_y + (q_2(x_start,
249                     y_start + (j - 1) * h2) + 2/h1) *
250                     w[1][j];
251             }
252         }
253
254         // Right interior border
255         if (right_border){
256             for (j = 1; j <= N; j++) {
257                 aw_x = aw_ij(N, w, x_start, y_start, M, j, h1, h2);
258                 bw_y = bw_y_ij(N, w, x_start, y_start, M, j, h1, h2);
259                 A[M][j] = 2*aw_x / h1 - bw_y + (q_2(x_start + (M - 1) * h1,
260                     y_start + (j - 1) * h2) + 2/h1) *
261                     w[M][j];
262             }
263         }
264     }

```

```

259
260 // Top border
261 if (top_border){
262     for (i = 1; i <= M; i++) {
263         aw_x = aw_ij(N, w, x_start, y_start, i, N, h1, h2);
264         bw_y = bw_ij(N, w, x_start, y_start, i, N, h1, h2);
265         A[i][N] = -aw_x + 2*bw_y / h2 + q_2(x_start + (i - 1) * h1,
266                                             y_start + (N - 1) * h2) * w[i][N];
267     }
268 }
269
270 // Bottom border
271 if (bottom_border){
272     for (i = 1; i <= M; i++) {
273         aw_x = aw_ij(N, w, x_start, y_start, i, 1, h1, h2);
274         bw_y = bw_ij(N, w, x_start, y_start, i, 2, h1, h2);
275         A[i][1] = -aw_x - 2*bw_y / h2 + q_2(x_start + (i - 1)* h1, y_start
276                                             ) * w[i][1];
277     }
278 }
279 if (left_border && bottom_border){
280     aw_x = aw_ij(N, w, x_start, y_start, 2, 1, h1, h2);
281     bw_y = bw_ij(N, w, x_start, y_start, 1, 2, h1, h2);
282     A[1][1] = -2*aw_x / h1 - 2*bw_y / h2 + (q_2(x_start, y_start) + 2/h1)
283         * w[1][1];
284 }
285 if (left_border && top_border){
286     aw_x = aw_ij(N, w, x_start, y_start, 2, N, h1, h2);
287     bw_y = bw_ij(N, w, x_start, y_start, 1, N, h1, h2);
288     A[1][N] = -2*aw_x / h1 + 2*bw_y / h2 + (q_2(x_start, y_start + (N - 1)
289         * h2) + 2/h1)* w[1][N];
290 }
291 if (right_border && bottom_border){
292     aw_x = aw_ij(N, w, x_start, y_start, M, 1, h1, h2);
293     bw_y = bw_ij(N, w, x_start, y_start, M, 2, h1, h2);
294     A[M][1] = 2*aw_x / h1 - 2 * bw_y / h2 + (q_2(x_start + (M - 1) * h1,
295         y_start) + 2/h1) * w[M][1];
296 }
297 if (right_border && top_border) {
298     aw_x = aw_ij(N, w, x_start, y_start, M, N, h1, h2);
299     bw_y = bw_ij(N, w, x_start, y_start, M, N, h1, h2);
300     A[M][N] = 2*aw_x / h1 + 2 * bw_y / h2 + (q_2(x_start + (M - 1) * h1,
301         y_start + (N - 1) * h2) + 2/
302         h1) * w[M][N];
303 }
304 }
305
306 void calculate_r(int M, int N,
307                 double (*r)[N+2],
308                 double (*Aw)[N+2],
309                 double (*B)[N+2]
310 ){
311     int i, j;
312     for(i = 0; i <= M + 1; i++) {
313         for (j = 0; j <= N + 1; j++) {
314             if(i == 0 || i == M+1 || j == 0 || j == N+1)
315                 r[i][j] = 0;
316             else
317                 r[i][j] = Aw[i][j] - B[i][j];
318         }
319     }
320 }

```

```

316     }
317 }
318 }
319
320
321 void get_idx_n_idx(int *idx,
322                   int *n_idx,
323                   int process_amnt,
324                   int grid_size,
325                   int coordinate){
326     if (grid_size % process_amnt == 0) {
327         *n_idx = grid_size / process_amnt;
328         *idx = coordinate * (grid_size / process_amnt);
329     }
330     else
331     {
332         if (coordinate == 0){
333             *n_idx = grid_size % process_amnt + grid_size / process_amnt;
334             *idx = 0;
335         } else
336         {
337             *n_idx = grid_size / process_amnt;
338             *idx = grid_size % process_amnt + coordinate * (grid_size /
339                 process_amnt);
340         }
341     }
342 }
343
344 #define A1 0.0
345 #define A2 4.0
346 #define B1 0.0
347 #define B2 3.0
348 #define EPS_REL 1e-6
349 #define DOWN_TAG 1000
350 #define MAX_ITER 100000
351
352
353 void send_rcv_borders(int n_x, int n_y,
354                      const int process_amounts[2],
355                      double x_idx,
356                      double y_idx,
357                      const int my_coords[2],
358                      int tag,
359                      double (*w)[n_y+2],
360                      double b_send[n_x],
361                      double l_send[n_y],
362                      double t_send[n_x],
363                      double r_send[n_y],
364                      double b_rec[n_x],
365                      double l_rec[n_y],
366                      double t_rec[n_x],
367                      double r_rec[n_y],
368                      int left_border, int right_border,
369                      int top_border, int bottom_border,
370                      double h1, double h2,
371                      MPI_Comm MPI_COMM_CART
372 ){
373
374     int i, j;
375     int neighbour_coords[2];
376     int neighbour_rank;

```

```

377 MPI_Request request[4] = {MPI_REQUEST_NULL, MPI_REQUEST_NULL,
378 MPI_REQUEST_NULL, MPI_REQUEST_NULL};
379 MPI_Status status;
380
381 // Bottom border send
382 if ((process_amounts[1] > 1) && !bottom_border) {
383     for (i = 0; i < n_x; i++)
384         b_send[i] = w[i+1][1];
385
386     neighbour_coords[0] = my_coords[0];
387     neighbour_coords[1] = my_coords[1] - 1;
388
389     MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
390     MPI_Isend(b_send, n_x, MPI_DOUBLE,
391             neighbour_rank, tag + DOWN_TAG,
392             MPI_COMM_CART, &request[0]);
393 }
394
395 // Left border send
396 if ((process_amounts[0] > 1) && !left_border) {
397     for (j = 0; j < n_y; j++)
398         l_send[j] = w[1][j+1];
399
400     neighbour_coords[0] = my_coords[0] - 1;
401     neighbour_coords[1] = my_coords[1];
402
403     MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
404     MPI_Isend(l_send, n_y, MPI_DOUBLE,
405             neighbour_rank, tag,
406             MPI_COMM_CART, &request[1]);
407 }
408
409 // Top border
410 if ((process_amounts[1] > 1) && !top_border) {
411     for (i = 0; i < n_x; i++)
412         t_send[i] = w[i+1][n_y];
413
414     neighbour_coords[0] = my_coords[0];
415     neighbour_coords[1] = my_coords[1] + 1;
416
417     MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
418     MPI_Isend(t_send, n_x, MPI_DOUBLE,
419             neighbour_rank, tag,
420             MPI_COMM_CART, &request[2]);
421 }
422
423 // Right border
424 if ((process_amounts[0] > 1) && !right_border) {
425     for (j = 0; j < n_y; j++)
426         r_send[j] = w[n_x][j+1];
427
428     neighbour_coords[0] = my_coords[0] + 1;
429     neighbour_coords[1] = my_coords[1];
430
431     MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
432     MPI_Isend(r_send, n_y, MPI_DOUBLE,
433             neighbour_rank, tag,
434             MPI_COMM_CART, &request[3]);
435 }
436
437 // Receive borders
438 // Bottom border

```

```

438     if ((bottom_border && (process_amounts[1] > 1)) || (process_amounts[1] ==
439         1)) {
440         for (i = 1; i <= n_x; i++)
441             w[i][0] = u_2(A1 + (x_idx + i - 1) * h1, B1 + (y_idx - 1) * h2);
442     } else {
443         neighbour_coords[0] = my_coords[0];
444         neighbour_coords[1] = my_coords[1] - 1;
445         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
446         MPI_Recv(b_rec, n_x, MPI_DOUBLE,
447             neighbour_rank, tag, MPI_COMM_CART, &status);
448
449         for (i = 1; i <= n_x; i++)
450             w[i][0] = b_rec[i - 1];
451     }
452
453     // Left border
454     if ((left_border && (process_amounts[0] > 1)) || (process_amounts[0] ==
455         1)) {
456         for (j = 1; j <= n_y; j++){
457             w[0][j] = u_2(A1 + (x_idx - 1) * h1, B1 + (y_idx + j - 1) * h2);
458         }
459     } else {
460         neighbour_coords[0] = my_coords[0] - 1;
461         neighbour_coords[1] = my_coords[1];
462
463         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
464         MPI_Recv(l_rec, n_y, MPI_DOUBLE,
465             neighbour_rank, tag, MPI_COMM_CART, &status);
466
467         for (j = 1; j <= n_y; j++)
468             w[0][j] = l_rec[j - 1];
469     }
470
471     // Top border
472     if ((top_border && (process_amounts[1] > 1)) || (process_amounts[1] == 1)
473         ) {
474         for (i = 1; i <= n_x; i++)
475             w[i][n_y + 1] = u_2(A1 + (x_idx + i - 1) * h1, B1 + (y_idx + n_y)
476                 * h2);
477     } else {
478         neighbour_coords[0] = my_coords[0];
479         neighbour_coords[1] = my_coords[1] + 1;
480         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
481         MPI_Recv(t_rec, n_x, MPI_DOUBLE,
482             neighbour_rank, tag + DOWN_TAG,
483             MPI_COMM_CART, &status);
484
485         for (i = 1; i <= n_x; i++)
486             w[i][n_y + 1] = t_rec[i - 1];
487     }
488
489     // Right border
490     if ((right_border && (process_amounts[0] > 1)) || (process_amounts[0] ==
491         1)) {
492         for (j = 1; j <= n_y; j++)
493             w[n_x + 1][j] = u_2(A1 + (x_idx + n_x)*h1, B1 + (y_idx + j - 1) *
494                 h2);
495     } else {
496         neighbour_coords[0] = my_coords[0] + 1;
497         neighbour_coords[1] = my_coords[1];
498         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);

```



```

494     MPI_Recv(r_rec, n_y, MPI_DOUBLE,
495             neighbour_rank, tag, MPI_COMM_CART, &status);
496
497     for (j = 1; j <= n_y; j++)
498         w[n_x + 1][j] = r_rec[j - 1];
499 }
500
501 for (int i = 0; i < 4; i++) {
502     MPI_Wait(&request[i], &status);
503 }
504 }
505
506 int main(int argc, char *argv[]) {
507     if (argc != 3) {
508         printf("Program receive %d numbers. Should be 2: M, N\n", argc);
509         return -1;
510     }
511
512     int M = atoi(argv[argc - 2]);
513     int N = atoi(argv[argc - 1]);
514     if ((M <= 0) || (N <= 0)) {
515         printf("M and N should be integer and > 0!!!\n");
516         return -1;
517     }
518     printf("M = %d, N = %d\n", M, N);
519
520     int my_rank;
521     int n_processes;
522     int process_amounts[2] = {0, 0};
523     int write[1] = {0};
524
525     double h1 = (A2 - A1) / M;
526     double h2 = (B2 - B1) / N;
527     double cur_eps = 1.0;
528
529     MPI_Init(&argc, &argv);
530     MPI_Status status;
531     MPI_Request request;
532
533     // For the cartesian topology
534     MPI_Comm MPI_COMM_CART;
535     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
536     MPI_Comm_size(MPI_COMM_WORLD, &n_processes);
537
538     // Creating rectangular supports
539     MPI_Dims_create(n_processes, 2, process_amounts);
540
541     printf("p_x = %d, p_y = %d\n", process_amounts[0], process_amounts[1]);
542     int periods[2] = {0, 0};
543
544     // Create cartesian topology in communicator
545     MPI_Cart_create(MPI_COMM_WORLD, 2,
546                   process_amounts, periods,
547                   1, &MPI_COMM_CART);
548
549     int my_coords[2];
550     // Receive corresponding to rank process coordinates
551     MPI_Cart_coords(MPI_COMM_CART, my_rank, 2, my_coords);
552
553     int x_idx, n_x;
554     get_idx_n_idx(&x_idx, &n_x, process_amounts[0], M+1, my_coords[0]);
555

```

```

556     int y_idx, n_y;
557     get_idx_n_idx(&y_idx, &n_y, process_amounts[1], N+1, my_coords[1]);
558
559     double start_time = MPI_Wtime();
560
561     // Create each block of size n_x and n_y with borders
562     double *t_send = malloc(sizeof(double[n_x]));
563     double *t_rec = malloc(sizeof(double[n_x]));
564     double *b_send = malloc(sizeof(double[n_x]));
565     double *b_rec = malloc(sizeof(double[n_x]));
566
567     double *l_send = malloc(sizeof(double[n_y]));
568     double *l_rec = malloc(sizeof(double[n_y]));
569     double *r_send = malloc(sizeof(double[n_y]));
570     double *r_rec = malloc(sizeof(double[n_y]));
571
572     int i, j;
573     int n_iters = 0;
574     double block_eps;
575
576     double (*w)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
577     double (*w_pr)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
578     double (*B)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
579     double tau = 0;
580     double global_tau = 0;
581     double alpha_k, beta_k;
582     double denominator;
583     double whole_denum;
584     double global_alpha, global_beta;
585     double eps_local, eps_r;
586
587     double (*Aw)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
588     double (*r_k_1)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
589     double (*r_k)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
590     double (*Ar)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
591     double (*z)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
592     double (*Az)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
593     double (*w_w_pr)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
594
595     int left_border = 0;
596     int top_border = 0;
597     int right_border = 0;
598     int bottom_border = 0;
599
600     if (my_coords[0] == 0)
601         left_border = 1;
602
603     if (my_coords[0] == (process_amounts[0] - 1))
604         right_border = 1;
605
606     if (my_coords[1] == 0)
607         bottom_border = 1;
608
609     if (my_coords[1] == (process_amounts[1] - 1))
610         top_border = 1;
611
612     printf("L%d, R%d, T%d, B%d, 'x'%d, 'y'%d\n",
613           left_border, right_border, top_border, bottom_border, my_coords[0],
614           my_coords[1]);
615     printf("%d %d\n", x_idx, y_idx);
616

```

```

617 double (*Au)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
618 double (*U)[n_y + 2] = malloc(sizeof(double[n_x + 2][n_y + 2]));
619 for (i = 0; i <= n_x + 1; i++)
620     for (j = 0; j <= n_y + 1; j++)
621         U[i][j] = u_2(A1 + (x_idx + i - 1) * h1, B1 + (y_idx + j - 1) * h2
        );
622
623 Aw_mult(n_x, n_y, Au, U, h1, h2,
624         A1 + x_idx * h1, B1 + y_idx * h2,
625         left_border, right_border,
626         top_border, bottom_border);
627
628 B_right(n_x, n_y, B,
629         h1, h2,
630         A1 + x_idx * h1,
631         B1 + y_idx * h2,
632         left_border, right_border,
633         top_border, bottom_border);
634 double norm_b, all_norm_b;
635 norm_b = dot_product(n_x, n_y, B, B, h1, h2,
636                     left_border, right_border,
637                     top_border, bottom_border);
638 MPI_Allreduce(&norm_b, &all_norm_b, 1,
639             MPI_DOUBLE, MPI_SUM, MPI_COMM_CART);
640 all_norm_b = sqrt(all_norm_b);
641
642 double error_mean = 0;
643 int amnt = 0;
644 for (i = 1; i <= n_x; i++)
645     for (j = 1; j <= n_y; j++){
646         error_mean += fabs(Au[i][j] - B[i][j]);
647         amnt += 1;
648     }
649 printf("ERROR FROM B = %3.2f\n", error_mean / amnt);
650
651 for (i = 0; i <= n_x + 1; i++)
652     for (j = 0; j <= n_y + 1; j++)
653         w[i][j] = 0; // u_2(A1 + (x_idx + i - 1) * h1, B1 + (y_idx + j -
        1) * h2);
654
655 int tag = 0;
656 while ((cur_eps > EPS_REL) && (n_iters < MAX_ITER)) {
657     if (my_rank == 0) {
658         if (n_iters % 1000 == 0)
659             printf("%g \n", cur_eps);
660     }
661     n_iters++;
662
663     for (i = 0; i <= n_x + 1; i++) {
664         for (j = 0; j <= n_y + 1; j++) {
665             if (i == 0 || j == 0 || i == n_x + 1 || j == n_y + 1) {
666                 w_pr[i][j] = 0;
667             } else {
668                 w_pr[i][j] = w[i][j];
669             }
670         }
671     }
672
673     send_rcv_borders(n_x, n_y, process_amounts,
674                     x_idx, y_idx, my_coords, tag,
675                     w,
676                     b_send, l_send, t_send, r_send,

```

```

677         b_rec, l_rec, t_rec, r_rec,
678         left_border, right_border,
679         top_border, bottom_border,
680         h1, h2, MPI_COMM_CART);
681
682     Aw_mult(n_x, n_y,
683            Aw, w,
684            h1, h2,
685            A1 + x_idx * h1, B1 + y_idx * h2,
686            left_border, right_border,
687            top_border, bottom_border);
688
689     // Make initialization for CG
690     if (n_iters == 1) {
691         calculate_r(n_x, n_y, r_k_1, B, Aw);
692
693         send_recv_borders(n_x, n_y, process_amounts, x_idx, y_idx,
694                        my_coords, tag,
695                        r_k_1,
696                        b_send, l_send, t_send, r_send,
697                        b_rec, l_rec, t_rec, r_rec,
698                        left_border, right_border, top_border,
699                        bottom_border,
700                        h1, h2, MPI_COMM_CART);
701         for (i = 0; i <= n_x + 1; i++)
702             for (j = 0; j <= n_y + 1; j++)
703                 z[i][j] = r_k_1[i][j];
704     }
705
706     Aw_mult(n_x, n_y,
707            Ar, r_k_1,
708            h1, h2,
709            A1 + x_idx * h1, B1 + y_idx * h2,
710            left_border, right_border,
711            top_border, bottom_border);
712
713     Aw_mult(n_x, n_y,
714            Az, z,
715            h1, h2,
716            A1 + x_idx * h1, B1 + y_idx * h2,
717            left_border, right_border,
718            top_border, bottom_border);
719
720     alpha_k = dot_product(n_x, n_y, r_k_1, r_k_1, h1, h2,
721                        left_border, right_border,
722                        top_border, bottom_border);
723     denominator = dot_product(n_x, n_y, Az, z, h1, h2,
724                        left_border, right_border,
725                        top_border, bottom_border);
726
727     MPI_Allreduce(&alpha_k, &global_alpha, 1,
728                  MPI_DOUBLE, MPI_SUM, MPI_COMM_CART);
729     MPI_Allreduce(&denominator, &whole_denum, 1,
730                  MPI_DOUBLE, MPI_SUM, MPI_COMM_CART);
731     global_alpha = global_alpha / whole_denum;
732
733     for (i = 1; i <= n_x; i++)
734         for (j = 1; j <= n_y; j++) {
735             w[i][j] = w[i][j] + global_alpha * z[i][j];
736             r_k[i][j] = r_k_1[i][j] - global_alpha * Az[i][j];
737         }

```

```

737     send_recv_borders(n_x, n_y, process_amounts, x_idx, y_idx, my_coords,
738                       tag,
739                       r_k,
740                       b_send, l_send, t_send, r_send,
741                       b_rec, l_rec, t_rec, r_rec,
742                       left_border, right_border, top_border, bottom_border,
743                       h1, h2, MPI_COMM_CART);
744     beta_k = dot_product(n_x, n_y, r_k, r_k, h1, h2,
745                          left_border, right_border,
746                          top_border, bottom_border
747                          );
748     denominator = dot_product(n_x, n_y, r_k_1, r_k_1, h1, h2,
749                               left_border, right_border,
750                               top_border, bottom_border);
751     MPI_Allreduce(&beta_k, &global_beta, 1,
752                  MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
753     MPI_Allreduce(&denominator, &whole_denum, 1,
754                  MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
755     global_beta = global_beta / whole_denum;
756
757     for (i = 1; i <= n_x; i++)
758         for (j = 1; j <= n_y; j++) {
759             z[i][j] = r_k[i][j] + global_beta * z[i][j];
760             r_k_1[i][j] = r_k[i][j];
761         }
762     send_recv_borders(n_x, n_y, process_amounts, x_idx, y_idx, my_coords,
763                       tag,
764                       z,
765                       b_send, l_send, t_send, r_send,
766                       b_rec, l_rec, t_rec, r_rec,
767                       left_border, right_border, top_border, bottom_border,
768                       h1, h2, MPI_COMM_CART);
769     block_eps = dot_product(n_x, n_y, r_k_1, r_k_1, h1, h2,
770                             left_border, right_border,
771                             top_border, bottom_border);
772     MPI_Allreduce(&block_eps, &cur_eps, 1,
773                  MPI_DOUBLE, MPI_SUM, MPI_COMM_CART);
774     cur_eps = sqrt(cur_eps) / all_norm_b;
775 }
776
777 // Waiting for all processes
778 MPI_Barrier(MPI_COMM_WORLD);
779 double end_time = MPI_Wtime();
780
781 if (my_rank != 0) {
782     MPI_Recv(write, 1, MPI_INT, my_rank - 1, 0, MPI_COMM_WORLD, &status);
783 } else {
784     printf("TIME = %f\n", end_time - start_time);
785     printf("Number of iterations = %d\n", n_iters);
786     printf("Tau = %f\n", tau);
787     printf("Eps = %f\n", EPS_REL);
788 }
789
790 // usleep(500);
791 if (my_rank != n_processes - 1)
792     MPI_Send(write, 1, MPI_INT, my_rank + 1, 0, MPI_COMM_WORLD);
793
794 FILE *dim0, *dim1, *grid, *u_file, *true_u_file;
795 char u_file_name[FILENAME_MAX];
796 sprintf(u_file_name, "u_%d_%d.csv", my_coords[0], my_coords[1]);

```

```

797 char true_u_file_name[FILENAME_MAX];
798 sprintf(true_u_file_name, "true.u_%d_%d.csv", my_coords[0], my_coords[1])
    ;
799
800 dim0 = fopen("dim0.csv", "w");
801 dim1 = fopen("dim1.csv", "w");
802 grid = fopen("grid.csv", "w");
803 u_file = fopen(u_file_name, "w");
804 true_u_file = fopen(true_u_file_name, "w");
805
806
807 for (int j = y_idx; j < y_idx + n_y; j++) {
808     for (int i = x_idx; i < x_idx + n_x; i++) {
809         fprintf(u_file, "%g ", w[i - x_idx + 1][j - y_idx + 1]);
810         fprintf(true_u_file, "%g ", u_2(A1 + i*h1, B1 + j*h2));
811     }
812     fprintf(u_file, "\n");
813     fprintf(true_u_file, "\n");
814 }
815
816 if (my_rank == 0) {
817     for (int j = 0; j <= N; j++) {
818         fprintf(dim0, "%g ", B1 + j*h2);
819     }
820
821     for (int i = 0; i <= M; i++) {
822         fprintf(dim1, "%g ", A1 + i*h1);
823     }
824     fprintf(grid, "%d %d", process_amounts[1], process_amounts[0]);
825 }
826 fclose(dim0);
827 fclose(dim1);
828 fclose(grid);
829 fclose(u_file);
830 fclose(true_u_file);
831
832 free(Au);
833 free(U);
834 free(w);
835 free(w_pr);
836 free(B);
837 free(Az);
838 free(z);
839 free(Ar);
840 free(r_k);
841 free(r_k_1);
842 free(Aw);
843 free(w_w_pr);
844
845 free(t_send);
846 free(t_rec);
847 free(b_send);
848 free(b_rec);
849 free(r_send);
850 free(r_rec);
851 free(l_send);
852 free(l_rec);
853 MPI_Finalize();
854 return 0;
855 }

```

Листинг 2: neyman_pde_mpi_cg.c