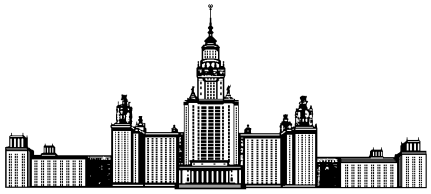

Московский государственный университет имени М.В.Ломоносова

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 с использованием технологий MPI+OpenMP+CUDA.

Выполнил:

студент 617 группы

Г.В. Кормаков

Москва, 2021

Содержание

1 Постановка задачи

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

$$\gamma_R = \{(A_2, y), B_1 \leqslant y \leqslant B_2\}, \quad \gamma_L = \{(A_1, y), B_1 \leqslant y \leqslant B_2\}$$
$$\gamma_T = \{(x, B_2), A_1 \leqslant x \leqslant A_2\}, \quad \gamma_B = \{(x, B_1), A_1 \leqslant x \leqslant A_2\}$$

рассматривается дифференциальное уравнение Пуассона с потенциалом

$$-\Delta u + q(x, y)u = F(x, y) \tag{1}$$

в котором оператор Лапласа

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

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:

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

где $\boldsymbol{n}$ - 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 $\alpha \geqslant 0$ считаются известными, функцию $u(x, y)$, удовлетворяющую уравнению 1 и граничным условиям, определенным вариантом задания 8, требуется найти.

Важно отметить, что нормаль $\boldsymbol{n}$ 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 П дискретизуем сеткой $\overline{\omega}_h = \overline{\omega}_1 \times \overline{\omega}_2$, где $\overline{\omega}_1$ – разбиение сетки по оси Ox с шагом $h_1 = \frac{A_2 - A_1}{M}$ и $\overline{\omega}_2$ – разбиение сетки по оси Oy с шагом $h_2 = \frac{B_2 - B_1}{N}$:

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

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

Рассматривается линейное пространство H функций, заданных на сетке $\overline{\omega}_h$. Обозначим через w_{ij} значение сеточной функции $w \in H$ в узле сетки $(x_i, y_j) \in \overline{\omega}_h$. 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

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

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

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

где $A : H \to H$ – 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:

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

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

$$\Delta_h w_{ij} = \frac{1}{h_1}\left(k(x_i + 0.5h_1, y_j)\, w_{x,ij} - k(x_i - 0.5h_1, y_j)\, w_{x,(i-1)j}\right) +$$
$$+\frac{1}{h_2}\left(k(x_i, y_j + 0.5h_2)\, w_{y,ij} - k(x_i, y_j - 0.5h_2)\, w_{y,i(j-1)}\right) =$$
$$= \frac{a_{(i+1)j} w_{x,ij} - a_{ij} w_{x,(i-1)j}}{h_1} + \frac{b_{i(j+1)} w_{y,ij} - b_{ij} w_{y,i(j-1)}}{h_2} =$$
$$= \frac{a_{(i+1)j} w_{\overline{x},(i+1)j} - a_{ij} w_{\overline{x},ij}}{h_1} + \frac{b_{i(j+1)} w_{\overline{y},i(j+1)} - b_{ij} w_{\overline{y},ij}}{h_2} \equiv (a w_{\overline{x}})_{x,ij} + (b w_{\overline{y}})_{y,ij} \qquad (5)$$

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

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

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:

$$\frac{2}{h_1}(a w_{\overline{x}})_{Mj} + \left(q_{Mj} + \frac{2\alpha_R}{h_1}\right) w_{Mj} - (b w_{\overline{y}})_{y,Mj} = F_{Mj} + \frac{2}{h_1}\psi^{(R)}_{Mj}, \quad j = \overline{1, N-1} \qquad (7)$$
$$-\frac{2}{h_1}(a w_{\overline{x}})_{1j} + \left(q_{0j} + \frac{2\alpha_L}{h_1}\right) w_{0j} - (b w_{\overline{y}})_{y,0j} = F_{0j} + \frac{2}{h_1}\psi^{(L)}_{0j}, \quad j = \overline{1, N-1} \qquad (8)$$

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

$$\frac{2}{h_2}(b w_{\overline{y}})_{iN} + q_{iN} w_{iN} - (a w_{\overline{x}})_{x,iN} = F_{iN} + \frac{2}{h_2}\psi^{(T)}_{iN}, \quad i = \overline{1, M-1} \qquad (9)$$
$$-\frac{2}{h_2}(b w_{\overline{y}})_{i1} + q_{i0} w_{i0} - (a w_{\overline{x}})_{x,i0} = F_{i0} + \frac{2}{h_2}\psi^{(B)}_{i0}, \quad i = \overline{1, M-1} \qquad (10)$$

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

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

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

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

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

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

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

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

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

Уточним, что обозначили за $\psi^{(T)}, \psi^{(B)}$ фунции 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.

$$-\left(aw_{\overline{x}}\right)_{x,ij} - \left(bw_{\overline{y}}\right)_{y,ij} + q_{ij}w_{ij} = F_{ij}, i = \overline{1, M-1}, j = \overline{1, N-1}$$

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

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

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

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

$$-\frac{2}{h_1}\left(aw_{\overline{x}}\right)_{10} - \frac{2}{h_2}\left(bw_{\overline{y}}\right)_{01} + \left(q_{00} + \frac{2}{h_1}\right)w_{00} = F_{00} + \left(\frac{2}{h_1} + \frac{2}{h_2}\right)\frac{\psi^{(L)}_{00} + \psi^{(B)}_{00}}{2}$$

$$\frac{2}{h_1}\left(aw_{\overline{x}}\right)_{M0} - \frac{2}{h_2}\left(bw_{\overline{y}}\right)_{M1} + \left(q_{M0} + \frac{2}{h_1}\right)w_{M0} = F_{M0} + \left(\frac{2}{h_1} + \frac{2}{h_2}\right)\frac{\psi^{(R)}_{M0} + \psi^{(B)}_{M0}}{2}$$

$$\frac{2}{h_1}\left(aw_{\overline{x}}\right)_{MN} + \frac{2}{h_2}\left(bw_{\overline{y}}\right)_{MN} + \left(q_{MN} + \frac{2}{h_1}\right)w_{MN} = F_{MN} + \left(\frac{2}{h_1} + \frac{2}{h_2}\right)\frac{\psi^{(R)}_{MN} + \psi^{(T)}_{MN}}{2}$$

$$-\frac{2}{h_1}\left(aw_{\overline{x}}\right)_{1N} + \frac{2}{h_2}\left(bw_{\overline{y}}\right)_{0N} + \left(q_{0N} + \frac{2}{h_1}\right)w_{0N} = F_{0N} + \left(\frac{2}{h_1} + \frac{2}{h_2}\right)\frac{\psi^{(L)}_{0N} + \psi^{(T)}_{0N}}{2}$$

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 $w^{(k)} \in H, k = 1, 2, \ldots$, сходящуюся по норме пространства H к решению разностной схемы, т.е.

$$\left\|w - w^{(k)}\right\|_E \xrightarrow[k \to +\infty]{} 0$$

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

$$w^{(k+1)}_{ij} = w^{(k)}_{ij} - \tau_{k+1} r^{(k)}_{ij},$$

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

В качестве условия остановки используем неравенство $\left\|w^{(k+1)} - w^{(k)}\right\|_E < \varepsilon$, 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

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

$$k_3(x,y) = 4 + x + y \tag{16}$$

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

3 Нахождение $F(x,y), \psi(x,y)$

Поскольку для численной реализации нам предлагается сравнить приближеннное решение с истинным, то необходимо найти аналитический вид функции $F(x,y)$ и краевых условий[1] 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 $\boldsymbol{n}$, направленным извне.

$$\frac{\partial u}{\partial x} = \frac{y}{2\sqrt{4+xy}}; \quad \frac{\partial u}{\partial y} = \frac{x}{2\sqrt{4+xy}};$$

$$\frac{\partial}{\partial x}\left(k(x,y)\frac{y}{2\sqrt{4+xy}}\right) = \frac{y}{2\sqrt{4+xy}} - k(x,y)\frac{y^2}{4(4+xy)^{3/2}}$$

Аналогично для $\dfrac{\partial}{\partial y}$.

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

Тогда

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

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).

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

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

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

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

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 переменым x и y в каждом домене принадлежало диапазону $[1/2, 2]$

2. количество узлов по переменым 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 пункта[2]. 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,

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

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 направлено ограничение 2.

Формальнее,

$$|n_x(i_1) - n_x(i_2)| \leqslant 1,\ |n_y(j_1) - n_y(j_2)| \leqslant 1,\ \forall (i_1, j_1), (i_2, j_2) \tag{22}$$

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).

[2]Корректность гарантирована объяснениями преподавателей на лекции, посвящённой постановке задачи

Таким образом, получаем следующие разложения

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

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

Условие 21 в новых обозначениях формулируется как $0.5 \leqslant \frac{a_x}{a_y} \leqslant 2$, 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 $p_x p_y \leqslant p$.

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.

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

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 $k_2 = M \mod p_x$, $s_2 = N \mod p_y$

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 $a_y + 1$

[3] Данная 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 нессиметричной 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. Критериев 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.

[4]Asus 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.

4.4 Реализация с использованием CUDA

В дополнение к приведённой выше реализации на MPI+OpenMP был реализован код на CUDA (см. приложение 3). Для этого каждая функция, вычисляющая значение в точке, и циклы над массивами по индексам были сделаны $\_\_device\_\_$ кодом.

Функции умножения на матрицу A и получение матрицы B перенесены на память *device* также.

Основной цикл вычислений теперь максимально упростился – осуществляется уже стандартный вызов инициализации MPI, создание топологии процессоров. Затем происходит инициализация cuda-профилировщика и начинается самое важное – необходимо распределить для каждого MPI-процессора его решётки и блоки внутри неё. Для этого задаётся в виде констант число нитей на один блок по каждой из двумерных осей.

В результате, получается разбиение, в котором общее число нитей равно числу точек в гриде этого процессора.

В основном цикле оптимизации происходит рассчёт, аналогичный предыдущему пункту. Однако важно отметить ключевые изменения:

1. Фукнция пересылки границ теперь использует динамическую память устройства, поэтому внутри есть вызовы копирований памяти по необходимости на хост.

2. Подсчёт скалярного произведения взят из общего доступа и синхронизирует все нити для получения единого значения. Как показалось, это самая сложная часть в реализации при переходе к cuda.

По завершению используемые ресурсы освобождаются и выводится затраченное время.

4.4.1 Сборка cuda-программы

При реализации сначала сделаны собственные библиотеки для удобства поиска ошибок, однако после компиляции на Polus вылетало сообщение о провале запуска MPI_Init.

Поэтому был сделан общий файл и слинкован с MPI библиотекой (см. makefile в 11. Однако результат оказался аналогичным.

4.4.2 Запуск скомпилированной программы

Для запуска был сделан файл постановки на задачу (см. 12). Он добавлял параметры запуска: число процессоров по стойкам, число стоек, число gpu на стойках и количество нитей для OpenMP.

В приложении указано число процессоров 4 и минимальные требования задания.

Запуск приводит к ошибке в MPI_Init, которую не смог исправить (недублирующаяся часть приведена ниже):

```
Rank 0 of MPI_COMM_WORLD could not getenv OPAL_PREFIX. Goodbye.
--------------------------------------------------------------------------
Primary job terminated normally, but 1 process returned
a non-zero exit code. Per user-direction, the job has been aborted.
--------------------------------------------------------------------------
--------------------------------------------------------------------------
Sorry! You were supposed to get help about:
    opal_init:startup:internal-failure
But I couldn't open the help file:
    /__unresolved_path_____________________________/exports/optimized/share/
        spectrum_mpi/help-opal-runtime.txt: No such file or directory. Sorry!
--------------------------------------------------------------------------
........
*** An error occurred in MPI_Init
*** on a NULL communicator
*** MPI_ERRORS_ARE_FATAL (processes in this communicator will now abort,
*** and potentially your MPI job)
```

```
17 [polus-c4-ib.bmc.hpc.cs.msu.ru:6428] Local abort before MPI_INIT completed completed
    successfully, but am not able to aggregate error messages, and not able to
    guarantee that all other processes were killed!
18 -------------------------------------------------------------------------
```

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 быстрее[5].

[5]Отметим, 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	$\mathbf{1000 \times 1000}$	**1.975**	**1**
256	1000×1000	2.341	0.84
512	1000×1000	3.488	0.57
(локально, CG) 4	$\mathbf{500 \times 1000}$	**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	$\mathbf{500 \times 1000}$	**3.213**	**1.68**
128	1000×1000	9.424	1
256	1000×1000	5.573	1.69
512	$\mathbf{1000 \times 1000}$	**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	$\mathbf{500 \times 1000}$	**0.470**	**1.75**
128	1000×1000	1.856	1
256	1000×1000	1.053	1.76
512	$\mathbf{1000 \times 1000}$	**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	$\mathbf{500 \times 1000}$	**3.089**	**1.86**
128	1000×1000	0.422	1
256	$\mathbf{1000 \times 1000}$	**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	$\mathbf{500 \times 500}$	**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	$\mathbf{500 \times 1000}$	**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 наружением 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).

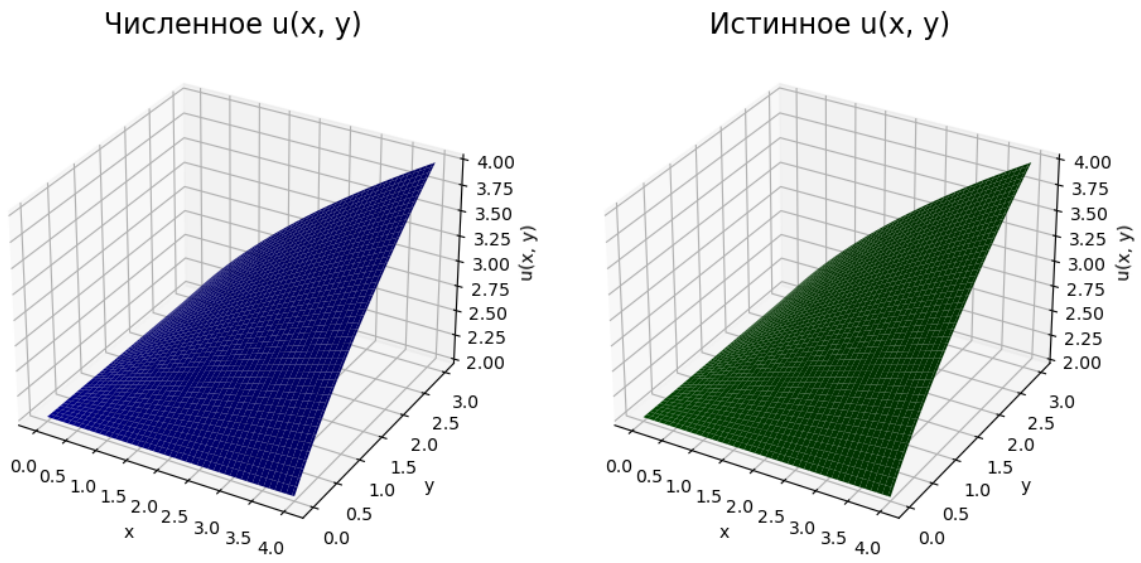

Рис. 1: Результат на сетке 500×1000

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

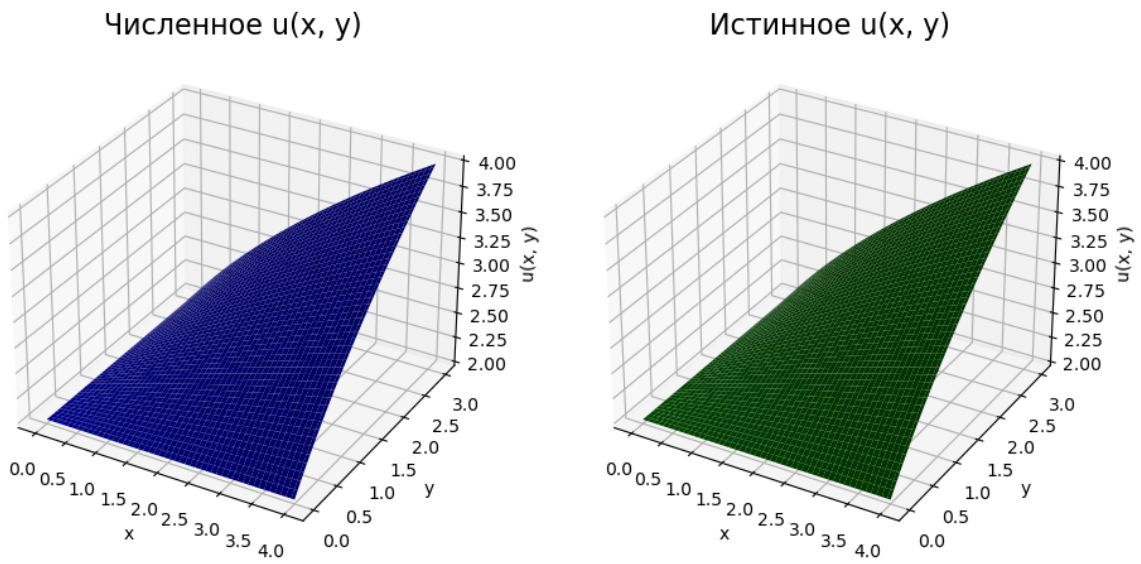

Рис. 2: Результат на сетке 1000×1000

6 Заключение

В ходе проведённых экспериментов была реализована программа для численного решшения 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.

Осуществлена реализация CUDA, работающая на локальном компьютере, однако для данной реализации не было смысла приводить результаты локальной системы, а на суперкомпьютере с системой Polus не удалось получить работающего запуска.

7 Литература

Источники

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

8 Приложение 1. Код MPI программы

```c
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include <time.h>
#include <math.h>
#include <unistd.h>


void print_matrix(int M, int N, double (*A)[N+2]){
    printf("Matrix:\n");
    for (int i=N+1; i>=0; i--){
        for (int j=0; j<=M+1; j++){
            printf("%3.2f ", A[i][j]);
        }
        printf("\n");
    }
}

void print_matrix_to_file(FILE *file,
                          int M, int N,
                          double (*A)[N+2]){
    for (int i=N+1; i>=0; i--){
        for (int j=0; j<=M+1; j++){
            fprintf(file,"%g ", A[i][j]);
        }
        fprintf(file, "\n");
    }
}

double u_2(double x, double y){
    return sqrt(4 + x * y);
}

double k_3(double x, double y){
    return 4 + x + y;
}

double q_2(double x, double y){
    double sum = x + y;
    if (sum < 0) {
        return 0;
    } else {
        return sum;
    }
}

double F(double x, double y){
    return ((pow(x, 3) - x*x*(y - 4) - x*(y*y + 8) +
    y*(y*y + 4*y - 8) + 4*q_2(x, y)*pow((4 + x*y), 2)) /
    (4 * pow((4 + x*y), 1.5)));
}

double psi_R(double x, double y){
    return (y*(4 + x + y) + 2*(4 + x*y)) / (2*sqrt(4 + x*y));
}


double psi_L(double x, double y){
    return (-y*(4 + x + y) + 2*(4 + x*y)) / (2*sqrt(4 + x*y));
}
```

```
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/
                        h1);
142         }
143     }
144     if (top_border){
145         for (i = 1; i <= M; i++) {
146             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,
                        y_start)) / 2);
165     }
166     if (right_border && top_border){
167         B[M][N] = (F(x_start + (M - 1)*h1, y_start + (N - 1)*h2) +
168                 (2/h1 + 2/h2) * (psi_R(x_start + (M - 1)*h1, y_start + (N - 1)
                        *h2) +
169                                 psi_T(x_start + (M - 1)*h1, y_start + (N - 1)*
                                    h2)) / 2);
170     }
171     if (right_border && bottom_border){
172         B[M][1] = (F(x_start + (M - 1)*h1, y_start) +
173                 (2/h1 + 2/h2) * (psi_R(x_start + (M - 1)*h1, y_start) +
174                                 psi_B(x_start + (M - 1)*h1, y_start)) / 2);
175     }
176 }
177 
178 
179 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)
            * (w[i + 1][j] - w[i][j]) / h1
186                     - k_3(x_start + (i - 0.5 - 1) * h1,y_start + (j - 1) * h2) *
                            (w[i][j] - w[i - 1][j]) / h1);
187 }
188
189 double aw_ij(int N,
190              double (*w)[N+2],
191              double x_start, double y_start,
192              int i, int j,
193              double h1, double h2
194 ){
195     return (k_3(x_start + (i - 0.5 - 1) * h1,y_start + (j - 1) * h2) * (w[i][
            j] - w[i - 1][j]) / h1);
196 }
197
198 double bw_y_ij(int N,
199             double (*w)[N+2],
200             double x_start, double y_start,
201             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)
            * (w[i][j + 1] - w[i][j]) / h2
205                     - k_3(x_start + (i - 1) * h1,y_start + (j - 0.5 - 1) * h2) *
                            (w[i][j] - w[i][j - 1]) / h2);
206 }
207
208 double bw_ij(int N,
209              double (*w)[N+2],
210              double x_start, double y_start,
211              int i, int j,
212              double h1, double h2
213 ){
214     return (k_3(x_start + (i - 1) * h1,y_start + (j - 0.5 - 1) * h2) * (w[i][
            j] - w[i][j-1]) / h2);
215 }
216
217 void Aw_mult(int M, int N,
218             double (*A)[N+2], double (*w)[N+2],
219             double h1, double h2,
220             double x_start, double y_start,
221             int left_border, int right_border,
222             int top_border, int bottom_border
223             )
224 {
225     double aw_x, bw_y;
226     int i, j;
227     for (i = 0; i <= M+1; i++){
228         for (j = 0; j <= N+1; j++) {
229             if (( i == 0) || i == M+1 || j == 0 || j == N+1){
230                 A[i][j] = w[i][j];
231             } else {
232                 aw_x = aw_x_ij(N, w, x_start, y_start, i, j, h1, h2);
233                 bw_y = bw_y_ij(N, w, x_start, y_start, i, j, h1, h2);
234                 A[i][j] = -aw_x - bw_y + q_2(x_start + (i - 1) * h1,
235                                              y_start + (j - 1) * h2) * w[i][j];
```

```
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_y_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_y_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
                    ) * w[i][1];
276         }
277     }
278     if (left_border && bottom_border){
279         aw_x = aw_ij(N, w, x_start, y_start, 2, 1, h1, h2);
280         bw_y = bw_ij(N, w, x_start, y_start, 1, 2, h1, h2);
281         A[1][1] = -2*aw_x / h1 - 2*bw_y / h2 + (q_2(x_start, y_start) + 2/h1)
                * w[1][1];
282     }
283     if (left_border && top_border){
284         aw_x = aw_ij(N, w, x_start, y_start, 2, N, h1, h2);
285         bw_y = bw_ij(N, w, x_start, y_start, 1, N, h1, h2);
286         A[1][N] = -2*aw_x / h1 + 2*bw_y / h2 + (q_2(x_start, y_start + (N - 1)
                 * h2) + 2/h1)* w[1][N];
287     }
288
289     if (right_border && bottom_border){
290         aw_x = aw_ij(N, w, x_start, y_start, M, 1, h1, h2);
291         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,
                y_start) + 2/h1) * w[M][1];
293     }
294     if (right_border && top_border) {
295         aw_x = aw_ij(N, w, x_start, y_start, M, N, h1, h2);
296         bw_y = bw_ij(N, w, x_start, y_start, M, N, h1, h2);
297         A[M][N] = 2*aw_x / h1 + 2 * bw_y / h2 + (q_2(x_start + (M - 1) * h1,
298                                                  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_recv_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,
        MPI_REQUEST_NULL, MPI_REQUEST_NULL};
378     MPI_Status status;
379
380     // Bottom border send
381     if ((process_amounts[1] > 1) && !bottom_border) {
382         for (i = 0; i < n_x; i++)
383             b_send[i] = w[i+1][1];
384
385         neighbour_coords[0] = my_coords[0];
386         neighbour_coords[1] = my_coords[1] - 1;
387
388         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
389         MPI_Isend(b_send, n_x, MPI_DOUBLE,
390                   neighbour_rank, tag + DOWN_TAG,
391                   MPI_COMM_CART, &request[0]);
392     }
393
394     // Left border send
395     if ((process_amounts[0] > 1) && !left_border) {
396         for (j = 0; j < n_y; j++)
397             l_send[j] = w[1][j+1];
398
399         neighbour_coords[0] = my_coords[0] - 1;
400         neighbour_coords[1] = my_coords[1];
401
402         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
403         MPI_Isend(l_send, n_y, MPI_DOUBLE,
404                   neighbour_rank, tag,
405                   MPI_COMM_CART, &request[1]);
406     }
407
408     // Top border
409     if ((process_amounts[1] > 1) && !top_border) {
410         for (i = 0; i < n_x; i++)
411             t_send[i] = w[i+1][n_y];
```

```
412
413         neighbour_coords[0] = my_coords[0];
414         neighbour_coords[1] = my_coords[1] + 1;
415
416         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
417         MPI_Isend(t_send, n_x, MPI_DOUBLE,
418                 neighbour_rank, tag,
419                 MPI_COMM_CART, &request[2]);
420     }
421
422     // Right border
423     if ((process_amounts[0] > 1) && !right_border) {
424         for (j = 0; j < n_y; j++)
425             r_send[j] = w[n_x][j+1];
426
427         neighbour_coords[0] = my_coords[0] + 1;
428         neighbour_coords[1] = my_coords[1];
429
430         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
431         MPI_Isend(r_send, n_y, MPI_DOUBLE,
432                 neighbour_rank, tag,
433                 MPI_COMM_CART, &request[3]);
434     }
435
436     // Receive borders
437     // Bottom border
438     if ((bottom_border && (process_amounts[1] > 1)) || (process_amounts[1] ==
          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] ==
          1)) {
454         for (j = 1; j <= n_y; j++){
455             w[0][j] = u_2(A1 + (x_idx - 1) * h1, B1 + (y_idx + j - 1) * h2);
456         }
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         for (i = 1; i <= n_x; i++)
473             w[i][n_y + 1] = u_2(A1 + (x_idx + i - 1) * h1,
474                                 B1 + (y_idx + n_y) * h2);
475     } else {
476         neighbour_coords[0] = my_coords[0];
477         neighbour_coords[1] = my_coords[1] + 1;
478         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
479         MPI_Recv(t_rec, n_x, MPI_DOUBLE,
480                 neighbour_rank, tag + DOWN_TAG,
481                 MPI_COMM_CART, &status);
482 
483         for (i = 1; i <= n_x; i++)
484             w[i][n_y + 1] = t_rec[i - 1];
485     }
486 
487     // Right border
488     if ((right_border && (process_amounts[0] > 1)) || (process_amounts[0] ==
            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) *
                    h2);
491     } else {
492         neighbour_coords[0] = my_coords[0] + 1;
493         neighbour_coords[1] = my_coords[1];
494         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
495         MPI_Recv(r_rec, n_y, MPI_DOUBLE,
496                 neighbour_rank, tag, MPI_COMM_CART, &status);
497 
498         for (j = 1; j <= n_y; j++)
499             w[n_x + 1][j] = r_rec[j - 1];
500     }
501 
502     for (int i = 0; i < 4; i++) {
503         MPI_Wait(&request[i], &status);
504     }
505 }
506 
507 int main(int argc, char *argv[]) {
508     if (argc != 3) {
509         printf("Program receive %d numbers. Should be 2: M, N\n", argc);
510         return -1;
511     }
512 
513     int M = atoi(argv[argc - 2]);
514     int N = atoi(argv[argc - 1]);
515     if ((M <= 0) || (N <= 0)) {
516         printf("M and N should be integer and > 0!!!\n");
517         return -1;
518     }
519     printf("M = %d, N = %d\n", M, N);
520 
521     int my_rank;
522     int n_processes;
523     int process_amounts[2] = {0, 0};
524     int write[1] = {0};
525 
526     double h1 = (A2 - A1) / M;
527     double h2 = (B2 - B1) / N;
528     double cur_eps = 1.0;
529 
```

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

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

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

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

```
774     }
775     fclose(dim0);
776     fclose(dim1);
777     fclose(grid);
778     fclose(u_file);
779     fclose(true_u_file);
780 
781     free(Au);
782     free(U);
783     free(w);
784     free(w_pr);
785     free(B);
786     free(Ar);
787     free(r_k);
788     free(Aw);
789     free(w_w_pr);
790 
791     free(t_send);
792     free(t_rec);
793     free(b_send);
794     free(b_rec);
795     free(r_send);
796     free(r_rec);
797     free(l_send);
798     free(l_rec);
799     MPI_Finalize();
800     return 0;
801 }
```

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++){
```

```
            fprintf(file,"%g ", A[i][j]);
        }
        fprintf(file, "\n");
    }
}

double u_2(double x, double y){
    return sqrt(4 + x * y);
}

double k_3(double x, double y){
    return 4 + x + y;
}

double q_2(double x, double y){
    double sum = x + y;
    if (sum < 0) {
        return 0;
    } else {
        return sum;
    }
}

double F(double x, double y){
    return ((pow(x, 3) - x*x*(y - 4) - x*(y*y + 8) +
    y*(y*y + 4*y - 8) + 4*q_2(x, y)*pow((4 + x*y), 2)) /
    (4 * pow((4 + x*y), 1.5)));
}

double psi_R(double x, double y){
    return (y*(4 + x + y) + 2*(4 + x*y)) / (2*sqrt(4 + x*y));
}


double psi_L(double x, double y){
    return (-y*(4 + x + y) + 2*(4 + x*y)) / (2*sqrt(4 + x*y));
}


double psi_T(double x, double y){
    return (x*(4 + x + y)) / (2*sqrt(4 + x*y));
}


double psi_B(double x, double y){
    return -psi_T(x, y);
}

double rho_1(int i,
             int M,
             int left_border,
             int right_border){
    if ((left_border && i == 1) || (right_border && i == M))
        return 0.5;
    return 1;
}

double rho_2(int j,
             int N,
             int bottom_border,
             int top_border){
    if ((bottom_border && j == 1) || (top_border && j == N))
```

```
        return 0.5;
    return 1;
}

double dot_product(int M, int N,
                   double (*U)[N + 2], double (*V)[N + 2],
                   double h1, double h2,
                   int left_border, int right_border,
                   int top_border, int bottom_border
                   ){
    double answer = 0.;
    double rho, r1, r2;

    for (int i=1; i <= M; i++){
        for (int j=1; j <= N; j++){
            r1 = rho_1(i, M, left_border, right_border);
            r2 = rho_2(j, N, bottom_border, top_border);
            rho = r1 * r2;
            answer += (rho * U[i][j] * V[i][j] * h1 * h2);
        }
    }
    return answer;
}


double norm(int M, int N, double (*U)[N + 2],
            double h1, double h2,
            int left_border, int right_border,
            int top_border, int bottom_border){
    return sqrt(dot_product(M, N, U, U, h1, h2,
                            left_border, right_border,
                            top_border, bottom_border));
}


void B_right(int M, int N, double (*B)[N+2],
       double h1, double h2,
       double x_start, double y_start,
       int left_border, int right_border,
       int top_border, int bottom_border){
    int i, j;

    for(i = 0; i <= M + 1; i++)
        for (j = 0; j <= N + 1; j++)
            B[i][j] = F(x_start + (i - 1) * h1, y_start + (j - 1) * h2);

    if (left_border){
        for (j = 1; j <= N; j++) {
            B[1][j] = (F(x_start, y_start + (j - 1) * h2) +
                   psi_L(x_start, y_start + (j - 1) * h2) * 2/h1);
        }
    }
    if (right_border){
        for (j = 1; j <= N; j++) {
            B[M][j] = (F(x_start + (M - 1)*h1, y_start + (j - 1) * h2) +
                      psi_R(x_start + (M - 1)*h1, y_start + (j - 1) * h2) * 2/
                          h1);
        }
    }
    if (top_border){
        for (i = 1; i <= M; i++) {
            B[i][N] = (F(x_start + (i - 1)*h1, y_start + (N - 1)*h2) +
```

```
                    psi_T(x_start + (i - 1)*h1, y_start + (N - 1)*h2) * 2/h2
                        );
        }
    }

    if (bottom_border){
        for (i = 1; i <= M; i++) {
            B[i][1] = (F(x_start + (i - 1)*h1, y_start) +
                    psi_B(x_start + (i - 1)*h1, y_start) * 2/h2);
        }
    }
    if (left_border && top_border){
        B[1][N] = (F(x_start, y_start + (N - 1)*h2) +
                (2/h1 + 2/h2) * (psi_L(x_start, y_start + (N - 1)*h2) +
                                psi_T(x_start, y_start + (N - 1)*h2)) / 2);
    }
    if (left_border && bottom_border){
        B[1][1] = (F(x_start, y_start)
                + (2/h1 + 2/h2) * (psi_L(x_start, y_start) + psi_B(x_start,
                    y_start)) / 2);
    }
    if (right_border && top_border){
        B[M][N] = (F(x_start + (M - 1)*h1, y_start + (N - 1)*h2) +
                (2/h1 + 2/h2) * (psi_R(x_start + (M - 1)*h1, y_start + (N - 1)
                    *h2) +
                                psi_T(x_start + (M - 1)*h1, y_start + (N - 1)*
                                    h2)) / 2);
    }
    if (right_border && bottom_border){
        B[M][1] = (F(x_start + (M - 1)*h1, y_start) +
                (2/h1 + 2/h2) * (psi_R(x_start + (M - 1)*h1, y_start) +
                                psi_B(x_start + (M - 1)*h1, y_start)) / 2);
    }
}


double aw_x_ij(int N,
            double (*w)[N+2],
            double x_start, double y_start,
            int i, int j,
            double h1, double h2
            ){
    return (1/h1) * (k_3(x_start + (i + 0.5 - 1) * h1,y_start + (j - 1) * h2)
         * (w[i + 1][j] - w[i][j]) / h1
                - k_3(x_start + (i - 0.5 - 1) * h1,y_start + (j - 1) * h2) *
                     (w[i][j] - w[i - 1][j]) / h1);
}

double aw_ij(int N,
              double (*w)[N+2],
              double x_start, double y_start,
              int i, int j,
              double h1, double h2
){
    return (k_3(x_start + (i - 0.5 - 1) * h1,y_start + (j - 1) * h2) * (w[i][
        j] - w[i - 1][j]) / h1);
}

double bw_y_ij(int N,
            double (*w)[N+2],
            double x_start, double y_start,
            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)
            * (w[i][j + 1] - w[i][j]) / h2
205                     - k_3(x_start + (i - 1) * h1,y_start + (j - 0.5 - 1) * h2) *
                            (w[i][j] - w[i][j - 1]) / h2);
206 }
207
208 double bw_ij(int N,
209                 double (*w)[N+2],
210                 double x_start, double y_start,
211                 int i, int j,
212                 double h1, double h2
213 ){
214     return (k_3(x_start + (i - 1) * h1,y_start + (j - 0.5 - 1) * h2) * (w[i][
            j] - w[i][j-1]) / h2);
215 }
216
217 void Aw_mult(int M, int N,
218              double (*A)[N+2], double (*w)[N+2],
219              double h1, double h2,
220              double x_start, double y_start,
221              int left_border, int right_border,
222              int top_border, int bottom_border
223              )
224 {
225     double aw_x, bw_y;
226     int i, j;
227     for (i = 0; i <= M+1; i++){
228         for (j = 0; j <= N+1; j++) {
229             if (( i == 0) || i == M+1 || j == 0 || j == N+1){
230                 A[i][j] = w[i][j];
231             } else {
232                 aw_x = aw_x_ij(N, w, x_start, y_start, i, j, h1, h2);
233                 bw_y = bw_y_ij(N, w, x_start, y_start, i, j, h1, h2);
234                 A[i][j] = -aw_x - bw_y + q_2(x_start + (i - 1) * h1,
235                                              y_start + (j - 1) * h2) * w[i][j];
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_y_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_y_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
                    ) * w[i][1];
276         }
277     }
278     if (left_border && bottom_border){
279         aw_x = aw_ij(N, w, x_start, y_start, 2, 1, h1, h2);
280         bw_y = bw_ij(N, w, x_start, y_start, 1, 2, h1, h2);
281         A[1][1] = -2*aw_x / h1 - 2*bw_y / h2 + (q_2(x_start, y_start) + 2/h1)
                * w[1][1];
282     }
283     if (left_border && top_border){
284         aw_x = aw_ij(N, w, x_start, y_start, 2, N, h1, h2);
285         bw_y = bw_ij(N, w, x_start, y_start, 1, N, h1, h2);
286         A[1][N] = -2*aw_x / h1 + 2*bw_y / h2 + (q_2(x_start, y_start + (N - 1)
                 * h2) + 2/h1)* w[1][N];
287     }
288
289     if (right_border && bottom_border){
290         aw_x = aw_ij(N, w, x_start, y_start, M, 1, h1, h2);
291         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,
                y_start) + 2/h1) * w[M][1];
293     }
294     if (right_border && top_border) {
295         aw_x = aw_ij(N, w, x_start, y_start, M, N, h1, h2);
296         bw_y = bw_ij(N, w, x_start, y_start, M, N, h1, h2);
297         A[M][N] = 2*aw_x / h1 + 2 * bw_y / h2 + (q_2(x_start + (M - 1) * h1,
298                                                 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_recv_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,
            MPI_REQUEST_NULL, MPI_REQUEST_NULL};
378     MPI_Status status;
379
380     // Bottom border send
381     if ((process_amounts[1] > 1) && !bottom_border) {
382         for (i = 0; i < n_x; i++)
383             b_send[i] = w[i+1][1];
384
385         neighbour_coords[0] = my_coords[0];
386         neighbour_coords[1] = my_coords[1] - 1;
387
388         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
389         MPI_Isend(b_send, n_x, MPI_DOUBLE,
390                   neighbour_rank, tag + DOWN_TAG,
391                   MPI_COMM_CART, &request[0]);
392     }
393
394     // Left border send
395     if ((process_amounts[0] > 1) && !left_border) {
396         for (j = 0; j < n_y; j++)
397             l_send[j] = w[1][j+1];
398
399         neighbour_coords[0] = my_coords[0] - 1;
400         neighbour_coords[1] = my_coords[1];
401
402         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
403         MPI_Isend(l_send, n_y, MPI_DOUBLE,
404                   neighbour_rank, tag,
405                   MPI_COMM_CART, &request[1]);
406     }
407
408     // Top border
409     if ((process_amounts[1] > 1) && !top_border) {
410         for (i = 0; i < n_x; i++)
411             t_send[i] = w[i+1][n_y];
412
413         neighbour_coords[0] = my_coords[0];
414         neighbour_coords[1] = my_coords[1] + 1;
415
416         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
417         MPI_Isend(t_send, n_x, MPI_DOUBLE,
418                   neighbour_rank, tag,
419                   MPI_COMM_CART, &request[2]);
420     }
421
422     // Right border
423     if ((process_amounts[0] > 1) && !right_border) {
424         for (j = 0; j < n_y; j++)
425             r_send[j] = w[n_x][j+1];
426
427         neighbour_coords[0] = my_coords[0] + 1;
428         neighbour_coords[1] = my_coords[1];
429
430         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
431         MPI_Isend(r_send, n_y, MPI_DOUBLE,
432                   neighbour_rank, tag,
433                   MPI_COMM_CART, &request[3]);
434     }
435
436     // Receive borders
437     // Bottom border
```

```
438     if ((bottom_border && (process_amounts[1] > 1)) || (process_amounts[1] ==
            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] ==
            1)) {
454         for (j = 1; j <= n_y; j++){
455             w[0][j] = u_2(A1 + (x_idx - 1) * h1, B1 + (y_idx + j - 1) * h2);
456         }
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         for (i = 1; i <= n_x; i++)
473             w[i][n_y + 1] = u_2(A1 + (x_idx + i - 1) * h1, B1 + (y_idx + n_y)
                    * h2);
474     } else {
475         neighbour_coords[0] = my_coords[0];
476         neighbour_coords[1] = my_coords[1] + 1;
477         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
478         MPI_Recv(t_rec, n_x, MPI_DOUBLE,
479                 neighbour_rank, tag + DOWN_TAG,
480                 MPI_COMM_CART, &status);
481 
482         for (i = 1; i <= n_x; i++)
483             w[i][n_y + 1] = t_rec[i - 1];
484     }
485 
486     // Right border
487     if ((right_border && (process_amounts[0] > 1)) || (process_amounts[0] ==
            1)) {
488         for (j = 1; j <= n_y; j++)
489             w[n_x + 1][j] = u_2(A1 + (x_idx + n_x)*h1, B1 + (y_idx + j - 1) *
                    h2);
490     } else {
491         neighbour_coords[0] = my_coords[0] + 1;
492         neighbour_coords[1] = my_coords[1];
493         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 denumenator;
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
601     if (my_coords[0] == 0)
602         left_border = 1;
603
604     if (my_coords[0] == (process_amounts[0] - 1))
605         right_border = 1;
606
607     if (my_coords[1] == 0)
608         bottom_border = 1;
609
610     if (my_coords[1] == (process_amounts[1] - 1))
611         top_border = 1;
612
613     printf("L%d, R%d, T%d, B%d, 'x'%d, 'y'%d\n",
614           left_border, right_border, top_border, bottom_border, my_coords[0],
                  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_recv_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,
```

```
                        b_rec, l_rec, t_rec, r_rec,
                        left_border, right_border,
                        top_border, bottom_border,
                        h1, h2, MPI_COMM_CART);

        Aw_mult(n_x, n_y,
                Aw, w,
                h1, h2,
                A1 + x_idx * h1, B1 + y_idx * h2,
                left_border, right_border,
                top_border, bottom_border);

// Make initialization for CG
        if (n_iters == 1) {
            calculate_r(n_x, n_y, r_k_1, B, Aw);

            send_recv_borders(n_x, n_y, process_amounts, x_idx, y_idx,
                my_coords, tag,
                            r_k_1,
                            b_send, l_send, t_send, r_send,
                            b_rec, l_rec, t_rec, r_rec,
                            left_border, right_border, top_border,
                                bottom_border,
                            h1, h2, MPI_COMM_CART);
            for (i = 0; i <= n_x + 1; i++)
                for (j = 0; j <= n_y + 1; j++)
                    z[i][j] = r_k_1[i][j];
        }

        Aw_mult(n_x, n_y,
                Ar, r_k_1,
                h1, h2,
                A1 + x_idx * h1, B1 + y_idx * h2,
                left_border, right_border,
                top_border, bottom_border);

        Aw_mult(n_x, n_y,
                Az, z,
                h1, h2,
                A1 + x_idx * h1, B1 + y_idx * h2,
                left_border, right_border,
                top_border, bottom_border);

        alpha_k = dot_product(n_x, n_y, r_k_1, r_k_1, h1, h2,
                              left_border, right_border,
                              top_border, bottom_border);
        denumenator = dot_product(n_x, n_y, Az, z, h1, h2,
                                  left_border, right_border,
                                  top_border, bottom_border);

        MPI_Allreduce(&alpha_k, &global_alpha, 1,
                      MPI_DOUBLE, MPI_SUM, MPI_COMM_CART);
        MPI_Allreduce(&denumenator, &whole_denum, 1,
                      MPI_DOUBLE, MPI_SUM, MPI_COMM_CART);
        global_alpha = global_alpha / whole_denum;

        for (i = 1; i <= n_x; i++)
            for (j = 1; j <= n_y; j++) {
                w[i][j] = w[i][j] + global_alpha * z[i][j];
                r_k[i][j] = r_k_1[i][j] - global_alpha * Az[i][j];
            }

```

```
737         send_recv_borders(n_x, n_y, process_amounts, x_idx, y_idx, my_coords,
                tag,
738                           r_k,
739                           b_send, l_send, t_send, r_send,
740                           b_rec, l_rec, t_rec, r_rec,
741                           left_border, right_border, top_border, bottom_border,
742                           h1, h2, MPI_COMM_CART);
743         beta_k = dot_product(n_x, n_y, r_k, r_k, h1, h2,
744                              left_border, right_border,
745                              top_border, bottom_border
746                              );
747         denumenator = dot_product(n_x, n_y, r_k_1, r_k_1, h1, h2,
748                                   left_border, right_border,
749                                   top_border, bottom_border);
750 
751         MPI_Allreduce(&beta_k, &global_beta, 1,
752                       MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
753         MPI_Allreduce(&denumenator, &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,
                tag,
763                           z,
764                           b_send, l_send, t_send, r_send,
765                           b_rec, l_rec, t_rec, r_rec,
766                           left_border, right_border, top_border, bottom_border,
767                           h1, h2, MPI_COMM_CART);
768 
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

10 Приложение 3. Код MPI+CUDA программы

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"
#include <string.h>

#include "cuda_profiler_api.h"
#define A1 0.0
#define A2 4.0
#define B1 0.0
#define B2 3.0

#define epsilon 1e-5
#define threadsPerBlock 4
#define numThreadsX 2
#define numThreadsY 2
#define EPS_REL 1e-6
#define DOWN_TAG 1000
#define MAX_ITER 100000

__device__ double dev_u_2(double x, double y){
    return sqrt(4 + x * y);
}

__device__ double dev_k_3(double x, double y){
    return 4 + x + y;
}

__device__ double dev_q_2(double x, double y){
    double sum = x + y;
    if (sum < 0) {
        return 0;
    } else {
        return sum;
    }
}

__device__ double dev_F(double x, double y){
    return ((pow(x, 3) - x*x*(y - 4) - x*(y*y + 8) +
             y*(y*y + 4*y - 8) + 4*dev_q_2(x, y)*pow((4 + x*y), 2)) /
            (4 * pow((4 + x*y), 1.5)));
}

__device__ double dev_psi_R(double x, double y){
    return (y*(4 + x + y) + 2*(4 + x*y)) / (2*sqrt(4 + x*y));
}

__device__ double dev_psi_L(double x, double y){
    return (-y*(4 + x + y) + 2*(4 + x*y)) / (2*sqrt(4 + x*y));
}


__device__ double dev_psi_T(double x, double y){
    return (x*(4 + x + y)) / (2*sqrt(4 + x*y));
}


__device__ double dev_psi_B(double x, double y){
    return -dev_psi_T(x, y);
}
```

```
61
62 __global__ void init_w(int n_y, double **w){
63     int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
64     int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
65     w[tid_x][tid_y] = 0.0;
66     return;
67 }
68
69 __global__ void copy_interior_w(int M, int N,
70                                 double **w,
71                                 double **w_pr){
72     int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
73     int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
74     if (tid_x == 0 || tid_y == 0 || tid_x == M + 1 || tid_y == N + 1) {
75         w_pr[tid_x][tid_y] = 0.0;
76     } else {
77         w_pr[tid_x][tid_y] = w[tid_x][tid_y];
78     }
79     return;
80 }
81
82 __global__ void get_top(int n_x, int n_y,
83                         int x_idx, int y_idx,
84                         double **w,
85                         double *dev_t_send){
86     int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
87     int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
88     int i = tid_x - x_idx;
89     if (tid_y == (y_idx + n_y - 1))
90         dev_t_send[i] = w[i+1][n_y];
91 }
92
93 __global__ void get_bottom(int n_x, int n_y,
94                            int x_idx, int y_idx,
95                            double **w,
96                            double *dev_b_send){
97     int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
98     int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
99     int i = tid_x - x_idx;
100    if (tid_y == y_idx)
101        dev_b_send[i] = w[i+1][1];
102 }
103 __global__ void get_left(int n_x, int n_y,
104                          int x_idx, int y_idx,
105                          double **w,
106                          double *dev_l_send){
107    int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
108    int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
109    int j = tid_y - y_idx;
110    if (tid_x == x_idx)
111        dev_l_send[j] = w[1][j+1];
112 }
113 __global__ void get_right(int n_x, int n_y,
114                           int x_idx, int y_idx,
115                           double **w,
116                           double *dev_r_send){
117    int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
118    int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
119    int j = tid_y - y_idx;
120    if (tid_x == (x_idx + n_x - 1))
121        dev_r_send[j] = w[n_x][j+1];
122 }
```

```
123
124 __global__ void set_top(int n_x, int n_y,
125                         int x_idx, int y_idx,
126                         double **w,
127                         double *dev_t_recv)
128 {
129     int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
130     int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
131     int i = tid_x - x_idx;
132     if (tid_y == (y_idx + n_y - 1))
133         w[i][n_y + 1] = dev_t_recv[i - 1];
134 }
135
136 __global__ void set_bottom(int n_x, int n_y,
137                            int x_idx, int y_idx,
138                            double **w,
139                            double *dev_b_recv
140 ){
141     int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
142     int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
143     int i = tid_x - x_idx;
144     if (tid_y == y_idx)
145         w[i][0] = dev_b_recv[i-1];
146 }
147 __global__ void set_left(int n_x, int n_y,
148                          int x_idx, int y_idx,
149                          double **w,
150                          double *dev_l_recv
151 ){
152     int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
153     int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
154     int j = tid_y - y_idx;
155     if (tid_x == x_idx)
156         w[0][j] = dev_l_recv[j-1];
157 }
158 __global__ void set_right(int n_x, int n_y,
159                           int x_idx, int y_idx,
160                           double **w,
161                           double *dev_r_recv
162 ){
163     int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
164     int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
165     int j = tid_y - y_idx;
166     if (tid_x == (x_idx + n_x - 1))
167         w[n_x+1][j] = dev_r_recv[j - 1];
168 }
169
170
171 __global__ void preset_top(int n_x, int n_y,
172                            int x_idx, int y_idx,
173                            double **w,
174                            double h1, double h2
175 ){
176     int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
177     int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
178     int i = tid_x - x_idx;
179     if (tid_y == (y_idx + n_y - 1))
180         w[i][n_y + 1] = dev_u_2(A1 + (x_idx + i - 1)*h1,
181                                 B1 + (y_idx + n_y) * h2);
182 }
183
184 __global__ void preset_bottom(int n_x, int n_y,
```

```
                            int x_idx, int y_idx,
                            double **w,
                            double h1, double h2
){
    int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
    int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
    int i = tid_x - x_idx;
    if (tid_y == y_idx)
        w[i][0] = dev_u_2(A1 + (x_idx + i - 1)*h1,
                          B1 + (y_idx - 1) * h2);
}
__global__ void preset_left(int n_x, int n_y,
                            int x_idx, int y_idx,
                            double **w,
                            double h1, double h2
){
    int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
    int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
    int j = tid_y - y_idx;
    if (tid_x == x_idx)
        w[0][j] = dev_u_2(A1 + (x_idx - 1)*h1,
                          B1 + (y_idx + j - 1) * h2);
}
__global__ void preset_right(int n_x, int n_y,
                             int x_idx, int y_idx,
                             double **w,
                             double h1, double h2
){
    int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
    int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
    int j = tid_y - y_idx;
    if (tid_x == (x_idx + n_x - 1))
        w[n_x+1][j] = dev_u_2(A1 + (x_idx + n_x)*h1,
                              B1 + (y_idx + j - 1) * h2);
}


__global__ void cudaB_right(int M, int N, double **B,
                            int x_idx, int y_idx,
                            double h1, double h2,
                            double x_start, double y_start,
                            int left_border, int right_border,
                            int top_border, int bottom_border){
    int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
    int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
    int i = tid_x - x_idx;
    int j = tid_y - y_idx;

    B[i][j] = dev_F(x_start + (i - 1) * h1, y_start + (j - 1) * h2);

    if (left_border){
        B[1][j] = (dev_F(x_start, y_start + (j - 1) * h2) +
                   dev_psi_L(x_start, y_start + (j - 1) * h2) * 2/h1);
    } else if (right_border){
        B[M][j] = (dev_F(x_start + (M - 1)*h1, y_start + (j - 1) * h2) +
                   dev_psi_R(x_start + (M - 1)*h1, y_start + (j - 1) * h2) * 2/
                       h1);
    }
    if (top_border){
        B[i][N] = (dev_F(x_start + (i - 1)*h1, y_start + (N - 1)*h2) +
                   dev_psi_T(x_start + (i - 1)*h1, y_start + (N - 1)*h2) * 2/h2
                       );
```

```
245     } else if (bottom_border){
246         B[i][1] = (dev_F(x_start + (i - 1)*h1, y_start) +
247                    dev_psi_B(x_start + (i - 1)*h1, y_start) * 2/h2);
248     }
249     if (left_border && top_border){
250         B[1][N] = (dev_F(x_start, y_start + (N - 1)*h2) +
251                    (2/h1 + 2/h2) * (dev_psi_L(x_start, y_start + (N - 1)*h2) +
252                                     dev_psi_T(x_start, y_start + (N - 1)*h2)) /
                                        2);
253     } else if (left_border && bottom_border){
254         B[1][1] = (dev_F(x_start, y_start)
255                     + (2/h1 + 2/h2) * (dev_psi_L(x_start, y_start) + dev_psi_B(
                           x_start, y_start)) / 2);
256     } else if (right_border && top_border){
257         B[M][N] = (dev_F(x_start + (M - 1)*h1, y_start + (N - 1)*h2) +
258                    (2/h1 + 2/h2) * (dev_psi_R(x_start + (M - 1)*h1, y_start + (
                       N - 1)*h2) +
259                                     dev_psi_T(x_start + (M - 1)*h1, y_start + (N
                                         - 1)*h2)) / 2);
260     } else if (right_border && bottom_border){
261         B[M][1] = (dev_F(x_start + (M - 1)*h1, y_start) +
262                    (2/h1 + 2/h2) * (dev_psi_R(x_start + (M - 1)*h1, y_start) +
263                                     dev_psi_B(x_start + (M - 1)*h1, y_start)) /
                                        2);
264     }
265 }
266
267 __device__ double dev_aw_x_ij(int N,
268                               double **w,
269                               double x_start, double y_start,
270                               int i, int j,
271                               double h1, double h2
272 ){
273     return (1/h1) * (dev_k_3(x_start + (i + 0.5 - 1) * h1,y_start + (j - 1) *
            h2) * (w[i + 1][j] - w[i][j]) / h1
274                      - dev_k_3(x_start + (i - 0.5 - 1) * h1,y_start + (j - 1) *
                             h2) * (w[i][j] - w[i - 1][j]) / h1);
275 }
276
277 __device__ double dev_aw_ij(int N,
278                             double **w,
279                             double x_start, double y_start,
280                             int i, int j,
281                             double h1, double h2
282 ){
283     return (dev_k_3(x_start + (i - 0.5 - 1) * h1,y_start + (j - 1) * h2) * (w
            [i][j] - w[i - 1][j]) / h1);
284 }
285
286 __device__ double dev_bw_y_ij(int N,
287                               double **w,
288                               double x_start, double y_start,
289                               int i, int j,
290                               double h1, double h2
291 ){
292     return (1/h2) * (dev_k_3(x_start + (i - 1) * h1,y_start + (j + 0.5 - 1) *
            h2) * (w[i][j + 1] - w[i][j]) / h2
293                      - dev_k_3(x_start + (i - 1) * h1,y_start + (j - 0.5 - 1) *
                             h2) * (w[i][j] - w[i][j - 1]) / h2);
294 }
295
296 __device__ double dev_bw_ij(int N,
```

```
297                             double **w,
298                             double x_start, double y_start,
299                             int i, int j,
300                             double h1, double h2
301 ){
302     return (dev_k_3(x_start + (i - 1) * h1,y_start + (j - 0.5 - 1) * h2) * (w
            [i][j] - w[i][j-1]) / h2);
303 }
304
305 __global__ void cuda_Aw_mult(int M, int N,
306                              int x_idx, int y_idx,
307                              double **A, double **w,
308                              double h1, double h2,
309                              double x_start, double y_start,
310                              int left_border, int right_border,
311                              int top_border, int bottom_border
312 ) {
313     double aw_x, bw_y;
314     int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
315     int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
316     int i = tid_x - x_idx;
317     int j = tid_y - y_idx;
318
319     if (( i == 0) || i == M+1 || j == 0 || j == N+1){
320         A[i][j] = w[i][j];
321     } else {
322         aw_x = dev_aw_x_ij(N, w, x_start, y_start, i, j, h1, h2);
323         bw_y = dev_bw_y_ij(N, w, x_start, y_start, i, j, h1, h2);
324         A[i][j] = -aw_x - bw_y + dev_q_2(x_start + (i - 1) * h1,
325                                          y_start + (j - 1) * h2) * w[i][j];
326     }
327
328     // Left interior border filling
329     if (left_border){
330         aw_x = dev_aw_ij(N, w, x_start, y_start, 2, j , h1, h2);
331         bw_y = dev_bw_y_ij(N, w, x_start, y_start, 1, j, h1, h2);
332         A[1][j] = -2*aw_x / h1 - bw_y + (dev_q_2(x_start, y_start + (j - 1) *
                h2)
333                                          + 2/h1) * w[1][j];
334     } else if (right_border){
335         // Right interior border
336         aw_x = dev_aw_ij(N, w, x_start, y_start, M, j, h1, h2);
337         bw_y = dev_bw_y_ij(N, w, x_start, y_start, M, j, h1, h2);
338         A[M][j] = 2*aw_x / h1 - bw_y + (dev_q_2(x_start + (M - 1) * h1,
339                                                 y_start + (j - 1) * h2) + 2/h1) *
                                                     w[M][j];
340     }
341
342     // Top border
343     if (top_border){
344         aw_x = dev_aw_x_ij(N, w, x_start, y_start, i, N, h1, h2);
345         bw_y = dev_bw_ij(N, w, x_start, y_start, i, N, h1, h2);
346         A[i][N] = -aw_x + 2*bw_y / h2 + dev_q_2(x_start + (i - 1) * h1,
347                                                 y_start + (N - 1) * h2) * w[i][N];
348     } else if (bottom_border){
349         // Bottom border
350         aw_x = dev_aw_x_ij(N, w, x_start, y_start, i, 1, h1, h2);
351         bw_y = dev_bw_ij(N, w, x_start, y_start, i, 2, h1, h2);
352         A[i][1] = -aw_x - 2*bw_y / h2 + dev_q_2(x_start + (i - 1)* h1, y_start
                ) * w[i][1];
353     }
354     if (left_border && bottom_border){
```

```
355         aw_x = dev_aw_ij(N, w, x_start, y_start, 2, 1, h1, h2);
356         bw_y = dev_bw_ij(N, w, x_start, y_start, 1, 2, h1, h2);
357         A[1][1] = -2*aw_x / h1 - 2*bw_y / h2 + (dev_q_2(x_start, y_start) + 2/
                h1) * w[1][1];
358     } else if (left_border && top_border){
359         aw_x = dev_aw_ij(N, w, x_start, y_start, 2, N, h1, h2);
360         bw_y = dev_bw_ij(N, w, x_start, y_start, 1, N, h1, h2);
361         A[1][N] = -2*aw_x / h1 + 2*bw_y / h2 + (dev_q_2(x_start, y_start + (N
                - 1) * h2) + 2/h1)* w[1][N];
362     }
363     if (right_border && bottom_border){
364         aw_x = dev_aw_ij(N, w, x_start, y_start, M, 1, h1, h2);
365         bw_y = dev_bw_ij(N, w, x_start, y_start, M, 2, h1, h2);
366         A[M][1] = 2*aw_x / h1 - 2 * bw_y / h2 + (dev_q_2(x_start + (M - 1) *
                h1, y_start) + 2/h1) * w[M][1];
367     } else if (right_border && top_border) {
368         aw_x = dev_aw_ij(N, w, x_start, y_start, M, N, h1, h2);
369         bw_y = dev_bw_ij(N, w, x_start, y_start, M, N, h1, h2);
370         A[M][N] = 2*aw_x / h1 + 2 * bw_y / h2 + (dev_q_2(x_start + (M - 1) *
                h1,
371                                                          y_start + (N - 1) * h2) +
                                                             2/h1) * w[M][N];
372     }
373 }
374
375 __global__ void calculate_r(int M, int N,
376                             int x_idx, int y_idx,
377                             double **r,
378                             double **Aw,
379                             double **B){
380     int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
381     int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
382     int i = tid_x - x_idx;
383     int j = tid_y - y_idx;
384
385     if(i == 0 || i == M+1 || j == 0 || j == N+1)
386         r[i][j] = 0;
387     else
388         r[i][j] = Aw[i][j] - B[i][j];
389 }
390
391
392 __device__ double dev_rho_1(int i,
393                             int M,
394                             int left_border,
395                             int right_border){
396     if ((left_border && i == 1) || (right_border && i == M))
397         return 0.5;
398     return 1;
399 }
400
401 __device__ double dev_rho_2(int j,
402                             int N,
403                             int bottom_border,
404                             int top_border){
405     if ((bottom_border && j == 1) || (top_border && j == N))
406         return 0.5;
407     return 1;
408 }
409
410 __global__ void cuda_dot_product(int n_x, int n_y,
411                                  int x_idx, int y_idx,
```

```
                                double **U, double **V,
                                double h1, double h2,
                                int left_border, int right_border,
                                int top_border, int bottom_border,
                                double *partial_product
){
// int num_threads_x = (int) sqrt(threadsPerBlock);
// int num_threads_y = threadsPerBlock / numThreadsX;
    __shared__ double cache[numThreadsX];
// __shared__ double cache_y[numThreadsY];


    int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
    int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
    int i = tid_x - x_idx;
    int j = tid_y - y_idx;

    int cacheIndex_x = threadIdx.x;
// int cacheIndex_y = threadIdx.y;
    double temp = 0;
    double rho, r1, r2;

    while (i < n_x) {
        j = tid_y - y_idx;
        while (j < n_y) {
            r1 = dev_rho_1(i, n_x, left_border, right_border);
            r2 = dev_rho_2(j, n_y, bottom_border, top_border);
            rho = r1 * r2;
            double part_dot = (rho * U[i][j] * V[i][j] * h1 * h2);
            temp += part_dot;
            j += blockDim.y * gridDim.y;
        }
        i += blockDim.x * gridDim.x;
    }
    cache[cacheIndex_x] = temp;
    __syncthreads();
    int k = blockDim.x / 2;
    while (k > 0) {
        if (cacheIndex_x < k) {
            cache[cacheIndex_x] += cache[cacheIndex_x + k];
        }
        __syncthreads();
        k = k / 2;
    }
    if (cacheIndex_x == 0) {
        partial_product[blockIdx.x] = cache[0];
    }
    return;
}

__global__ void cuda_w_step(int n_y,
                            int x_idx, int y_idx,
                            double **w,
                            double **r_k,
                            double tau
// double *w_next
) {
    int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
    int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
    int i = tid_x - x_idx;
    int j = tid_y - y_idx;
    double r_k_scaled = r_k[i][j] * tau;
```

```
474     w[i][j] = w[i][j] - r_k_scaled;
475     return;
476 }
477 
478 
479 void get_idx_n_idx(int *idx,
480                    int *n_idx,
481                    int process_amnt,
482                    int grid_size,
483                    int coordinate){
484     if (grid_size % process_amnt == 0) {
485         *n_idx = grid_size / process_amnt;
486         *idx = coordinate * (grid_size / process_amnt);
487     }
488     else
489     {
490         if (coordinate == 0){
491             *n_idx = grid_size % process_amnt + grid_size / process_amnt;
492             *idx = 0;
493         } else
494         {
495             *n_idx = grid_size / process_amnt;
496             *idx = grid_size % process_amnt + coordinate * (grid_size /
                        process_amnt);
497         }
498     }
499 }
500 
501 
502 void send_recv_borders(int n_x, int n_y,
503                        const int process_amounts[2],
504                        double x_idx,
505                        double y_idx,
506                        const int my_coords[2],
507                        int tag,
508                        double **w,
509                        double *b_send,
510                        double *l_send,
511                        double *t_send,
512                        double *r_send,
513                        double *b_rec,
514                        double *l_rec,
515                        double *t_rec,
516                        double *r_rec,
517                        int left_border, int right_border,
518                        int top_border, int bottom_border,
519                        double h1, double h2,
520                        MPI_Comm MPI_COMM_CART
521 ){
522     int neighbour_coords[2];
523     int neighbour_rank;
524 // int num_threads_x = (int) sqrt(threadsPerBlock);
525 // int num_threads_y = threadsPerBlock / numThreadsX;
526     int blocksPerGrid_x = n_x / numThreadsX + 1;
527     int blocksPerGrid_y = n_y / numThreadsY + 1;
528     dim3 gridShape = dim3(blocksPerGrid_x, blocksPerGrid_y);
529     dim3 blockShape = dim3(numThreadsX, numThreadsY);
530 
531     MPI_Request request[4] = {MPI_REQUEST_NULL, MPI_REQUEST_NULL,
532                               MPI_REQUEST_NULL, MPI_REQUEST_NULL};
533     MPI_Status status;
534 
```

```
535     double *dev_b_send, *dev_l_send, *dev_t_send, *dev_r_send;
536     double *dev_b_rec, *dev_l_rec, *dev_t_rec, *dev_r_rec;
537     cudaMalloc((void**)&dev_b_send, sizeof(double[n_x]));
538     cudaMalloc((void**)&dev_t_send, sizeof(double[n_x]));
539     cudaMalloc((void**)&dev_b_rec, sizeof(double[n_x]));
540     cudaMalloc((void**)&dev_t_rec, sizeof(double[n_x]));
541
542     cudaMalloc((void**)&dev_l_send, sizeof(double[n_y]));
543     cudaMalloc((void**)&dev_r_send, sizeof(double[n_y]));
544     cudaMalloc((void**)&dev_l_rec, sizeof(double[n_y]));
545     cudaMalloc((void**)&dev_r_rec, sizeof(double[n_y]));
546     ////////////////
547     // Bottom border send
548     if ((process_amounts[1] > 1) && !bottom_border) {
549         get_bottom<<<gridShape, blockShape>>>(n_x, n_y,
550                                               x_idx, y_idx,
551                                               w, dev_b_send);
552         cudaMemcpy(b_send, dev_b_send,
553                    sizeof(double[n_x]), cudaMemcpyDeviceToHost);
554
555         neighbour_coords[0] = my_coords[0];
556         neighbour_coords[1] = my_coords[1] - 1;
557
558         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
559         MPI_Isend(b_send, n_x, MPI_DOUBLE,
560                   neighbour_rank, tag + DOWN_TAG,
561                   MPI_COMM_CART, &request[0]);
562     }
563
564     // Left border send
565     if ((process_amounts[0] > 1) && !left_border) {
566         get_left<<<gridShape, blockShape>>>(n_x, n_y,
567                                             x_idx, y_idx,
568                                             w, dev_l_send);
569         cudaMemcpy(l_send, dev_l_send,
570                    sizeof(double[n_y]), cudaMemcpyDeviceToHost);
571
572         neighbour_coords[0] = my_coords[0] - 1;
573         neighbour_coords[1] = my_coords[1];
574
575         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
576         MPI_Isend(l_send, n_y, MPI_DOUBLE,
577                   neighbour_rank, tag,
578                   MPI_COMM_CART, &request[1]);
579     }
580
581     // Top border
582     if ((process_amounts[1] > 1) && !top_border) {
583         get_top<<<gridShape, blockShape>>>(n_x, n_y,
584                                            x_idx, y_idx,
585                                            w, dev_t_send);
586         cudaMemcpy(t_send, dev_t_send,
587                    sizeof(double[n_x]), cudaMemcpyDeviceToHost);
588
589         neighbour_coords[0] = my_coords[0];
590         neighbour_coords[1] = my_coords[1] + 1;
591
592         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
593         MPI_Isend(t_send, n_x, MPI_DOUBLE,
594                   neighbour_rank, tag,
595                   MPI_COMM_CART, &request[2]);
596     }
```

```
597
598     // Right border
599     if ((process_amounts[0] > 1) && !right_border) {
600         get_right<<<gridShape, blockShape>>>(n_x, n_y,
601                                              x_idx, y_idx,
602                                              w, dev_r_send);
603         cudaMemcpy(r_send, dev_r_send,
604                    sizeof(double[n_y]), cudaMemcpyDeviceToHost);
605
606         neighbour_coords[0] = my_coords[0] + 1;
607         neighbour_coords[1] = my_coords[1];
608
609         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
610         MPI_Isend(r_send, n_y, MPI_DOUBLE,
611                   neighbour_rank, tag,
612                   MPI_COMM_CART, &request[3]);
613     }
614
615     // Receive borders
616     // Bottom border
617     if ((bottom_border && (process_amounts[1] > 1)) || (process_amounts[1] ==
          1)) {
618         preset_bottom<<<gridShape, blockShape>>>(n_x, n_y, x_idx, y_idx, w, h1
              , h2);
619     } else {
620         neighbour_coords[0] = my_coords[0];
621         neighbour_coords[1] = my_coords[1] - 1;
622         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
623         MPI_Recv(b_rec, n_x, MPI_DOUBLE,
624                  neighbour_rank, tag, MPI_COMM_CART, &status);
625
626         cudaMemcpy(dev_b_rec, b_rec,
627                    sizeof(double[n_x]), cudaMemcpyHostToDevice);
628         set_bottom<<<gridShape, blockShape>>>(n_x, n_y, x_idx, y_idx, w,
              dev_b_rec);
629     }
630
631     // Left border
632     if ((left_border && (process_amounts[0] > 1)) || (process_amounts[0] ==
          1)) {
633         preset_left<<<gridShape, blockShape>>>(n_x, n_y, x_idx, y_idx, w, h1,
              h2);
634
635     } else {
636         neighbour_coords[0] = my_coords[0] - 1;
637         neighbour_coords[1] = my_coords[1];
638
639         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
640         MPI_Recv(l_rec, n_y, MPI_DOUBLE,
641                  neighbour_rank, tag, MPI_COMM_CART, &status);
642
643         cudaMemcpy(dev_l_rec, l_rec,
644                    sizeof(double[n_y]), cudaMemcpyHostToDevice);
645         set_left<<<gridShape, blockShape>>>(n_x, n_y, x_idx, y_idx, w,
              dev_l_rec);
646     }
647
648     // Top border
649     if ((top_border && (process_amounts[1] > 1)) || (process_amounts[1] == 1)
          ) {
650         preset_top<<<gridShape, blockShape>>>(n_x, n_y, x_idx, y_idx, w, h1,
              h2);
```

```
651     } else {
652         neighbour_coords[0] = my_coords[0];
653         neighbour_coords[1] = my_coords[1] + 1;
654         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
655         MPI_Recv(t_rec, n_x, MPI_DOUBLE,
656                 neighbour_rank, tag + DOWN_TAG,
657                 MPI_COMM_CART, &status);
658 
659         cudaMemcpy(dev_t_rec, t_rec,
660                   sizeof(double[n_x]), cudaMemcpyHostToDevice);
661         set_top<<<gridShape, blockShape>>>(n_x, n_y, x_idx, y_idx, w,
            dev_t_rec);
662     }
663 
664     // Right border
665     if ((right_border && (process_amounts[0] > 1)) || (process_amounts[0] ==
        1)) {
666         preset_right<<<gridShape, blockShape>>>(n_x, n_y, x_idx, y_idx, w, h1,
             h2);
667     } else {
668         neighbour_coords[0] = my_coords[0] + 1;
669         neighbour_coords[1] = my_coords[1];
670         MPI_Cart_rank(MPI_COMM_CART, neighbour_coords, &neighbour_rank);
671         MPI_Recv(r_rec, n_y, MPI_DOUBLE,
672                 neighbour_rank, tag, MPI_COMM_CART, &status);
673 
674         cudaMemcpy(dev_r_rec, r_rec,
675                   sizeof(double[n_y]), cudaMemcpyHostToDevice);
676         set_right<<<gridShape, blockShape>>>(n_x, n_y, x_idx, y_idx, w,
            dev_r_rec);
677     }
678 
679     for (int i = 0; i < 4; i++) {
680         MPI_Wait(&request[i], &status);
681     }
682 }
683 
684 
685 void cudaDotProduct(int n_x, int n_y,
686                     int x_idx, int y_idx,
687                     double **U, double **V,
688                     double h1, double h2,
689                     int left_border, int right_border,
690                     int top_border, int bottom_border,
691                     double *curr_sum)
692 {
693 // int num_threads_x = (int) sqrt(threadsPerBlock);
694 // int num_threads_y = threadsPerBlock / numThreadsX;
695     int blocksPerGrid_x = n_x / numThreadsX + 1;
696     int blocksPerGrid_y = n_y / numThreadsY + 1;
697     dim3 gridShape = dim3(blocksPerGrid_x, blocksPerGrid_y);
698     dim3 blockShape = dim3(numThreadsX, numThreadsY);
699     /////////
700     double c, *partial_c;
701     double *dev_partial_c;
702     partial_c = (double*) calloc(blocksPerGrid_x, sizeof(double));
703     // Allocate device memory
704     cudaMalloc((void**)&dev_partial_c, blocksPerGrid_x * sizeof(double));
705     /////////
706     cuda_dot_product<<<gridShape, blockShape>>>(n_x, n_y, x_idx, y_idx,
707                                                 U, V, h1, h2,
708                                                 left_border, right_border,
```

```
                                                top_border, bottom_border,
                                                dev_partial_c);
    cudaMemcpy(partial_c, dev_partial_c,
               blocksPerGrid_x * sizeof(double),
               cudaMemcpyDeviceToHost);
    c = 0;
    for (int i = 0; i < blocksPerGrid_x; ++i) {
        c = c + partial_c[i];
    }
    (*curr_sum) = c;
    /////////
    cudaFree(dev_partial_c);
    free(partial_c);
    return;
}


int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Program receive %d numbers. Should be 2: M, N\n", argc);
        return -1;
    }

    int M = atoi(argv[argc - 2]);
    int N = atoi(argv[argc - 1]);
    if ((M <= 0) || (N <= 0)) {
        printf("M and N should be integer and > 0!!!\n");
        return -1;
    }
    printf("M = %d, N = %d\n", M, N);
    int my_rank, n_processes;
    int process_amounts[2] = {0, 0};
    int write[1] = {0};
    double h1 = (A2 - A1) / M;
    double h2 = (B2 - B1) / N;
    double cur_eps = 1.0;

    MPI_Init(&argc, &argv);
    MPI_Status status;

    // For the cartesian topology
    MPI_Comm MPI_COMM_CART;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n_processes);

    // Creating rectangular supports
    MPI_Dims_create(n_processes, 2, process_amounts);
    int periods[2] = {0, 0};

    // Create cartesian topology in communicator
    MPI_Cart_create(MPI_COMM_WORLD, 2,
                    process_amounts, periods,
                    1, &MPI_COMM_CART);

    int my_coords[2];
    // Receive corresponding to rank process coordinates
    MPI_Cart_coords(MPI_COMM_CART, my_rank, 2, my_coords);

    int x_idx, n_x;
    get_idx_n_idx(&x_idx, &n_x, process_amounts[0], M+1, my_coords[0]);

    int y_idx, n_y;
```

```
771     get_idx_n_idx(&y_idx, &n_y, process_amounts[1], N+1, my_coords[1]);
772 
773     double start_time = MPI_Wtime();
774     ////////////////////////
775     cudaProfilerStart();
776 // int num_threads_x = (int) sqrt(threadsPerBlock);
777 // int num_threads_y = threadsPerBlock / numThreadsX;
778     int blocksPerGrid_x = n_x / numThreadsX + 1;
779     int blocksPerGrid_y = n_y / numThreadsY + 1;
780     dim3 gridShape = dim3(blocksPerGrid_x, blocksPerGrid_y);
781     dim3 blockShape = dim3(numThreadsX, numThreadsY);
782     ////////////////////////
783     double *t_send = (double *) malloc(sizeof(double[n_x]));
784     double *t_rec = (double *) malloc(sizeof(double[n_x]));
785     double *b_send = (double *) malloc(sizeof(double[n_x]));
786     double *b_rec = (double *) malloc(sizeof(double[n_x]));
787 
788     double *l_send = (double *) malloc(sizeof(double[n_y]));
789     double *l_rec = (double *) malloc(sizeof(double[n_y]));
790     double *r_send = (double *) malloc(sizeof(double[n_y]));
791     double *r_rec = (double *) malloc(sizeof(double[n_y]));
792     int n_iters = 0;
793     double block_eps;
794 
795     double **w, **w_pr, **B;
796     double **Aw, **r_k, **Ar, **w_w_pr;
797 
798     cudaMalloc((void**)&w, sizeof(double[n_x + 2][n_y + 2]));
799     cudaMalloc((void**)&w_pr, sizeof(double[n_x + 2][n_y + 2]));
800     cudaMalloc((void**)&B, sizeof(double[n_x + 2][n_y + 2]));
801     cudaMalloc((void**)&Aw, sizeof(double[n_x + 2][n_y + 2]));
802     cudaMalloc((void**)&r_k, sizeof(double[n_x + 2][n_y + 2]));
803     cudaMalloc((void**)&Ar, sizeof(double[n_x + 2][n_y + 2]));
804     cudaMalloc((void**)&w_w_pr, sizeof(double[n_x + 2][n_y + 2]));
805     ////////////////////////
806     double tau = 0;
807     double global_tau = 0;
808     double denumenator;
809     double whole_denum;
810 // double global_alpha, global_beta;
811 // double eps_local, eps_r;
812     int left_border = 0;
813     int top_border = 0;
814     int right_border = 0;
815     int bottom_border = 0;
816     if (my_coords[0] == 0)
817         left_border = 1;
818 
819     if (my_coords[0] == (process_amounts[0] - 1))
820         right_border = 1;
821 
822     if (my_coords[1] == 0)
823         bottom_border = 1;
824 
825     if (my_coords[1] == (process_amounts[1] - 1))
826         top_border = 1;
827     ////////////////////////
828     cudaB_right<<<gridShape, blockShape>>>(n_x, n_y, B,
829                                            x_idx, y_idx,
830                                            h1, h2,
831                                            A1 + x_idx * h1,
832                                            B1 + y_idx * h2,
```

```
                                        left_border, right_border,
                                        top_border, bottom_border);
    init_w<<<gridShape, blockShape>>>(n_x, w);

    int tag = 0;
    while ((cur_eps > EPS_REL) && (n_iters < MAX_ITER)) {
        if (my_rank == 0) {
            if (n_iters % 1000 == 0)
                printf("%g \n", cur_eps);
        }
        n_iters++;

        copy_interior_w<<<gridShape, blockShape>>>(n_x, n_y,
                                                   w, w_pr);

        send_recv_borders(n_x, n_y, process_amounts,
                          x_idx, y_idx, my_coords, tag,
                          w,
                          b_send, l_send, t_send, r_send,
                          b_rec, l_rec, t_rec, r_rec,
                          left_border, right_border,
                          top_border, bottom_border,
                          h1, h2, MPI_COMM_CART);
        cuda_Aw_mult<<<gridShape, blockShape>>>(n_x, n_y,
                    x_idx, y_idx,
                    Aw, w,
                    h1, h2,
                    A1 + x_idx * h1, B1 + y_idx * h2,
                    left_border, right_border,
                    top_border, bottom_border);

        calculate_r<<<gridShape, blockShape>>>(n_x, n_y,
                                               x_idx, y_idx,
                                               r_k, Aw, B);
        send_recv_borders(n_x, n_y, process_amounts,
                          x_idx, y_idx, my_coords, tag,
                          r_k,
                          b_send, l_send, t_send, r_send,
                          b_rec, l_rec, t_rec, r_rec,
                          left_border, right_border,
                          top_border, bottom_border,
                          h1, h2, MPI_COMM_CART);
        cuda_Aw_mult<<<gridShape, blockShape>>>(n_x, n_y,
                     x_idx, y_idx,
                     Ar, r_k,
                     h1, h2,
                     A1 + x_idx * h1, B1 + y_idx * h2,
                     left_border, right_border,
                     top_border, bottom_border);
        cudaDotProduct(n_x, n_y,
                     x_idx, y_idx,
                     Ar, r_k, h1, h2,
                     left_border, right_border,
                     top_border, bottom_border,
                     &tau);

        cudaDotProduct(n_x, n_y,
                       x_idx, y_idx,
                       Ar, Ar, h1, h2,
                       left_border, right_border,
                       top_border, bottom_border,
                       &denumenator);
```

```
895         MPI_Allreduce(&tau, &global_tau, 1,
896                       MPI_DOUBLE, MPI_SUM, MPI_COMM_CART);
897         MPI_Allreduce(&denumenator, &whole_denum, 1,
898                       MPI_DOUBLE, MPI_SUM, MPI_COMM_CART);
899         global_tau = global_tau / whole_denum;
900         cuda_w_step<<<gridShape, blockShape>>>(n_y,
901                     x_idx, y_idx,
902                     w, r_k,
903                     tau
904 // w_next
905                     );
906         calculate_r<<<gridShape, blockShape>>>(n_x, n_y,
907                                                x_idx, y_idx,
908                                                w_w_pr, w, w_pr);
909 
910         cudaDotProduct(n_x, n_y,
911                        x_idx, y_idx,
912                        w_w_pr, w_w_pr, h1, h2,
913                        left_border, right_border,
914                        top_border, bottom_border,
915                        &block_eps);
916         block_eps = sqrt(block_eps);
917 
918         MPI_Allreduce(&block_eps, &cur_eps, 1,
919                       MPI_DOUBLE, MPI_SUM, MPI_COMM_CART);
920     }
921 
922     // Waiting for all processes
923     MPI_Barrier(MPI_COMM_WORLD);
924     double end_time = MPI_Wtime();
925 
926     if (my_rank != 0) {
927         MPI_Recv(write, 1, MPI_INT, my_rank - 1, 0, MPI_COMM_WORLD, &status);
928     } else {
929         printf("TIME = %f\n", end_time - start_time);
930         printf("Number of iterations = %d\n", n_iters);
931         printf("Tau = %f\n", tau);
932         printf("Eps = %f\n", EPS_REL);
933     }
934 
935     if (my_rank != n_processes - 1)
936         MPI_Send(write, 1, MPI_INT, my_rank + 1, 0, MPI_COMM_WORLD);
937 
938 
939     cudaFree(w);
940     cudaFree(w_pr);
941     cudaFree(B);
942     cudaFree(Aw);
943     cudaFree(r_k);
944     cudaFree(Ar);
945     cudaFree(w_w_pr);
946 
947     free(t_send);
948     free(t_rec);
949     free(b_send);
950     free(b_rec);
951     free(r_send);
952     free(r_rec);
953     free(l_send);
954     free(l_rec);
955     cudaProfilerStop();
956     MPI_Finalize();
```

```
957     return 0;
958 }
```

Листинг 3: main_huge.cu

11 Приложение 4. Сборка CUDA программы

```
1 cuda_neuman_pde: main_huge.cu
2   nvcc -gencode arch=compute_52,code=sm_52 -gencode arch=compute_60,code=sm_60 -I./ -I
      . -I/opt/ibm/spectrum_mpi/include -L/tmp/lepera/yeti_10010004_RTM2/ibm-smpi/
      ompibase/dependencies/lib -L/opt/ibm/spectrum_mpi/lib -lmpiprofilesupport -
      lmpi_ibm -lm main_huge.cu -o cuda_neuman_pde
```

../code/makefile

12 Приложение 5. Запуск CUDA

```
1 #BSUB -n 4 -q short
2 #BSUB -W 00:15
3 #BSUB -R "span[ptile=2]"
4 #BSUB -gpu "num=1:mode=exclusive_process"
5 #BSUB -o cuda_500_1000_out
6 #BSUB -e cuda_500_1000_err
7 OMP_NUM_THREADS=1 mpirun ./cuda_neuman_pde 500 1000
```

../code/task.lsf