Отчет по выполнению задания
Численное решение краевой задачи для уравнения Пуассона
(вариант 10)

Сенин Александр, 617

16 декабря 2022 г.

Математическая постановка задачи

Требуется методом конечных разностей приближенно решить краевую задачу для уравнения Пуассона с потенциалом в прямоугольной области.

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

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

в котором оператор Лапласа

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

Для выделения единственного решения уравнение дополняется граничными условиями. На каждом отрезке границы прямоугольника Π задается условие одним из трех способов — условиями Дирихле, Неймана и третьего типа.

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 и граничным условиям, требуется найти.

Заметим, что нормаль n не определена в угловых точках прямоугольника. Краевое условие второго и третьего типа следует рассматривать лишь в тех точках границы, где нормаль существует.

Численный метод решения

Будем решать задачу Пуассона численно с помощью метода конечных разностей.

Для этого П разобьем на сетку $\overline{\omega}_h = \overline{\omega}_1 \times \overline{\omega}_2$, где $\overline{\omega}_1$ – разбиение по оси Ox с шагом $h_1 = (A_2 - A_1)/M$ и $\overline{\omega}_2$ – разбиение по оси Oy с шагом $h_2 = (B_2 - B_1)/N$:

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

Через ω_h обозначим узлы сетки $\overline{\omega}_h$

Рассмотрим 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 $\rho_{ij} = \rho^{(1)}(x_i)\,\rho^{(2)}(y_j)$, где

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

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

$$Aw = B$$

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

В варианте 10 имеем дело с функциями $u(x, y) = 1 + \cos(\pi xy)$, $k(x, y) = 4 + x + y$, $q(x, y) = 0$, с двумя граничными условиями 2 типа и 2 условиями 3 типа.

Собираем разностную схему из соотношений во внутренних узлах, на граничных и угловых узлах. Систему соотношений можно воспринимать как систему линейных алгебраических уравнений, которая единственным образом определяет искомые значения w_{ij}. Далее в коде программы будем отдельно инициализировать правую часть, используя двойной цикл для внутренних узлов, одинарный цикл для граничных узлов и четыре присваивания для угловых узлов. После этого в коде определим действие разностной схемы на левую часть (аналогичные по структуре двойные и одинарные циклы и угловые присваивания). Приближенное решение полученной системы будем искать итерационно методом наименьших невязок. Такой алгоритм реализован в последовательном виде на языке Си (см. прикрепленный файл или листинг в конце этого отчета).

Описание проделанной работы

MPI программа

Будем разбивать сетку на прямоугольные блоки, каждый MPI-процесс будет независимо обсчитывать применение разностного оператора на своем блоке. Определяем распределение MPI-процессов по блокам с помощью топологии, созданной

MPI_Cart_create. На каждый MPI-процесс выделяем необходимые ему матрицы, после чего запускаем итерации метода наименьших невязок. Для корректного вычисления разностной схемы на границе блоков требуются актуальные значения с границы соседнего блока. Это реализуется обменом между MPI-процессами граничными значениями с помощью MPI_Isend и MPI_Recv. После вычисления скалярного произведения необходимо определить общий для всех MPI-процессов шаг метода, для этого производим редукцию в совокупное скалярное произведение с помощью MPI_Allreduce с редукцией вида MPI_SUM. Аналогичные редукции проводим для определения знаменателя шага метода и для определения величины критерия останова.

MPI+OpenMP программа

Расширим программу с помощью директив OpenMP. В общем случае циклы в программе сопровождались директивой #pragma omp parallel for. Приватные переменные (если есть) задавались внутри private(...), в цикле скалярного произведения задавалась редукция reduction(+:result). Код см. в прикрепленном файле или листинге в конце этого отчета.

MPI+OpenMP+OpenACC программа

Дополнительно была реализована гибридная программа для вычислений на GPU с помощью директив OpenACC. Базовые функции, вычисляющие математические функции $u, F, k, \psi, \phi, \rho$ обрамлены директивой #pragma acc routine. Аналогично примеру на лекциях, прямолинейное использование директив #pragma acc parallel или #pragma acc kernels не привело к ускорению из-за существенных накладных расходов на постоянное копирование матриц между GPU и CPU (которое происходило несколько раз внутри каждой итерации метода). Поэтому матрицы были предварительно скопированы на GPU перед всеми итерациями с помощью директив #pragma acc data copy(...). Матрицы, которые используются внутри цикла в качестве промежуточных (например, матрица w_next, которая нужна для промежуточного расчета матрицы с разницей w_diff) были созданы сразу на GPU с помощью директивы #pragma acc data create(...).

Чтобы помочь компилятору понять, что матрицы уже находятся на GPU, блоки вычислений с матрицами внутри функций применения разностного оператора, скалярного произведения и синхронизации границ были обрамлены директивой #pragma acc data present(...). Непосредственно циклы с вычислениями по матрицам обрамлены #pragma acc parallel. Известно, что в случае, если внутри блока сразу несколько циклов, то такая директива не гарантирует разделения между ними – второй цикл может начаться раньше первого. Во всех таких случаях в программе это допустимо. Сами циклы сопровождались #pragma acc loop в общем случае и #pragma acc loop independent, в случае если итерации цикла независимы по данным. При вычислении скалярного произведения осуществлялась редукция reduction(+:result). Для гарантии

компилятору, что матрицы, лежащие по указателям в аргументах функций, используются только непосредственно этими указателями, аргументы были сопровождены ключевым словом restrict.

Корректность реализации проверялась сравнением итогового решения с программой без OpenACC, а также величин критерия и скалярных произведений на каждой итерации. В ходе проверки корректности было обнаружены вычислительные проблемы (появлялись nan-ы). Причина была обнаружена – функция синхронизации границ не отрабатывала из-за того, что матрицы находились на GPU (границы на GPU не обновлялись). Для решения проблемы буферы для обмена границ были скопированы на GPU перед внешними итерациями метода наименьших невязок. После каждого обновления на GPU буфер копировался обратно на CPU с помощью #pragma acc data copyout(...) для дальнешей отправки другим MPI-процессам.

Матрицы и массивы в программе выделялись с помощью malloc, поэтому для помощи компилятору во всех директивах #pragma acc data указывались размеры оперируемых данных.

OpenMP директивы были сохранены в циклах при инициализации правой части B системы уравнений. Эта инициализация проводится один раз, поэтому ее можно проводить на CPU без больших потерь. Код см. в прикрепленном файле или листинге в конце этого отчета.

Исследование параллельных характеристик

Все программы были протестированы с заданной точностью $eps = 1e^{-6}$ на матрицах размера 160 x 160. Чтобы сэкономить общее время по рекомендации автора 4 задания было решено поставить ограничение на число итераций метода наименьших невязок при оценке времени работы на больших матрицах 500 x 500 и 500 x 1000. Ограничение в 10000 итераций по сути эквивалентно послаблению заданной точности eps ($eps \approx 2e^{-5}$ для 500 x 500 и $eps \approx 4e^{-5}$ для 500 x 1000).

Результаты запусков всех трех программ приведены в Таблицах 1, 2, 3. В Таблице 1 представлены столбцы «Уск. от послед.» — ускорение относительно последовательной программы; «Уск. от 1 MPI» — ускорение относительно MPI программы на 1 процессе. Дополнительно в Таблице 2 представлен столбец «Уск. от 1 MPI» — ускорение относительно аналогичной конфигурации на 1 потоке.

Число MPI-процессов	Размерность	Время (с)	Уск. от послед.	Уск. от 1 MPI
1 (послед.)	500 x 500	124,5	1,00	1,16
1 (MPI)		145,4	0.85	1.00
2		84,3	1,48	1,72
4		38,1	3,27	3,81
8		34,9	3,57	4,16
16		28,7	4,34	5,06
32		26,6	4,68	5,46
1 (послед.)	500 x 1000	238,2	1,00	1,23
1 (MPI)		295,0	0,81	1,00
2		175,4	1,36	1,68
4		88,5	2,69	3,33
8		69,7	3,42	4,23
16		53,0	4,49	5,56
32		44,9	5,31	6,57

Таблица 1: Запуски MPI программы. Ограничение в 10к итераций.

MPI-проц.	Размер	Время (с)	Уск. от послед.	Уск. от 1 MPI	Уск. от 1 потока
1 (послед.)	500 x 500	124,5	1,00	0,57	1,00
1 (MPI)		71,3	1,75	1,00	2,04
2		44,2	2,82	1,61	1,91
4		32,0	3,89	2,23	1,19
8		26,6	4,68	2,68	1,31
1 (послед.)	500 x 1000	238,2	1,00	0,60	1,00
1 (MPI)		142,8	1,67	1,00	2,07
2		87,5	2,72	1,63	2,00
4		65,1	3,66	2,19	1,36
8		46,3	5,14	3,08	1,51

Таблица 2: Запуски MPI+OpenMP программы в 4 потока. Ограничение в 10к итераций.

Конфигурация	Время (с)	Ускорение
Последовательная	124,5	1,00
MPI 20 процессов	28,1	4,43
MPI 40 процессов	24,0	5,18
MPI OMP 20 процессов 4 потока	22,7	5,48
MPI OMP 40 процессов 4 потока	19,9	6,25
MPI OMP ACC 1 GPU	4,9	25,40
MPI OMP ACC 2 GPU	3,8	32,76

Таблица 3: Запуски MPI+OpenMP+OpenACC. Размерность 500 x 500. Ограничение в 10к итераций.

Прокомментируем полученные результаты.

Результаты CPU

Получилось достичь ускорения как за счет использования MPI-процессов, так и за счет директив OpenMP. Стартовое увеличение числа процессов в 2, 4 раза приводит к почти пропорциональному ускорению. Тем не менее, с дальнейшим ростом числа процессов рост ускорения замедляется. Это объясняется как общей загруженностью, так и повышением накладных расходов на содержание параллельных процессов и синхронизацию буферов с граничными значениями блоков.

Результаты GPU

Статистика по запуску на 1 GPU: из 4.9 секунд расчета 0.06 секунд ушло на копирование с хоста (CPU) на девайс (GPU), 0.1 сек на копирование с девайса на хост, 0.3 сек на редукцию и 3.3 сек на полезное исполнение параллельных циклов.

Кроме того, можно получить более красивые результаты ускорения на GPU. Так, на расчет по матрицам 500 x 1000 ушло 6.7 секунд против 4.9 на 500 x 500. В то же время аналогичное увеличение данных приводит в среднем к росту времени выполнения в два раза на CPU. Вероятно, существенное увеличение размера данных приведет к дополнительному увеличению ускорения от использования GPU против CPU.

Визуальное сравнение на графике.

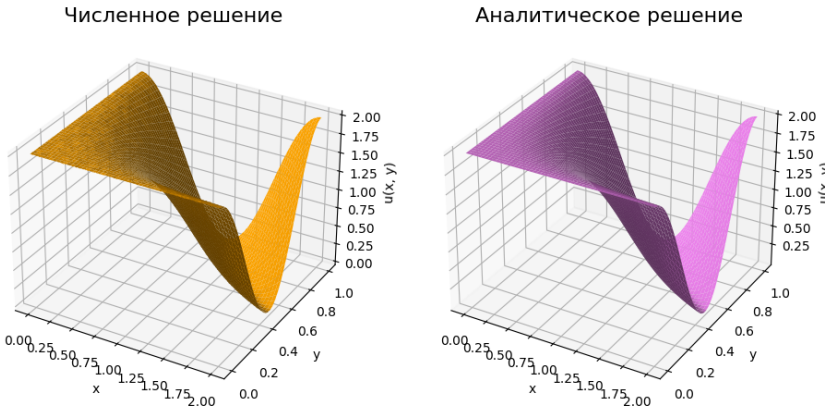

Рис. 1: Рисунок точного и приближенного решения на сетке с наибольшим количеством узлов.

Дополнительные эксперименты

Запуски в Таблице 3 проводились на матрицах размера 500 x 500. В процессе выполнения MPI+OpenMP+OpenACC кода на GPU весь итерационный метод для такой размерности занимает порядка 250 мб видеопамяти.

Конфигурация	Время (с)	Ускорение
1 процесс	1095.2	1
20 процессов 4 потока	89.5	12.3
1 GPU	9.1	120.4
2 GPU	6.3	173.9

Таблица 4: Запуски MPI+OpenMP+OpenACC. Размерность 14500 x 14500. Ограничение в 100 итераций.

Возникло предложение провести замеры на больших матрицах, занимающих большую часть видеопамяти. Был выбран размер матриц 14500 x 14500, для такого размера видеопамять загружена на 11,5 гб. Результаты приведены в Таблице 4.

Листинги

Листинг последовательной Си программы

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

const double A1 = 0;
const double A2 = 2;
const double B1 = 0;
const double B2 = 1;

const double EPSILON = 1e-6;

double dudx(double x, double y) {
    return -sin(M_PI * x * y) * M_PI * y;
}

double dudy(double x, double y) {
    return -sin(M_PI * x * y) * M_PI * x;
}

double u(double x, double y){ return 1 + cos(M_PI * x * y); }

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

double q(double x, double y){ return 0; }

double F(double x, double y){
    return sin(M_PI * x * y) * M_PI * (x + y) +
    k(x, y) * cos(M_PI * x * y) * M_PI * M_PI * (x * x + y * y);
}

double psi_R(double x, double y){
    return u(x, y) + k(x, y) * dudx(x, y);
}

double psi_L(double x, double y){
    return u(x, y) - k(x, y) * dudx(x, y);
}

double psi_T(double x, double y){
    return k(x, y) * dudy(x, y);
}

double psi_B(double x, double y){
    return -k(x, y) * dudy(x, y);
}


double rho(int i, int j, int i_max, int j_max) {
    double result = 1;
    if (i == 0 || i == i_max) {
        result *= 0.5;
    }
    if (j == 0 || j == j_max) {
        result *= 0.5;
    }
    return result;
}

void print_matrix(int M, int N, double (*Matrix)[N + 2], char* title) {
    printf("%s\n", title);
    for (int i = 0; i <= M + 1; i++) {
        for (int j = 0; j <= N + 1; j++) {
            printf("%3.3f ", Matrix[i][j]);
        }
        printf("\n");
    }
}

double dot_product(int M, int N, double (*X)[N + 2], double (*Y)[N + 2], double h1, double h2) {
    double result = 0.;
    int i, j;
    int i_max = M;
    int j_max = N;

    for (i = 1; i <= i_max; i++) {
        for (j = 1; j <= j_max; j++) {
            result += X[i][j] * Y[i][j] * rho(i, j, i_max, j_max) * h1 * h2;
        }
    }
    return result;
```

```
82  }
83
84  double norm(int M, int N, double (*X)[N + 1], double h1, double h2) {
85      return sqrt(dot_product(M, N, X, X, h1, h2));
86  }
87
88  void init_B_border(int M, int N, double (*Matrix)[N + 2], double h1, double h2, double x_min, double y_min) {
89      int i, j;
90      double x, y; double psi_ij;
91      int i_max = M;
92      int j_max = N;
93
94      /* left */
95      i = 0;
96      x = x_min + i * h1;
97      for (j = 0; j < j_max; j++) {
98          y = y_min + j * h2;
99          Matrix[i + 1][j + 1] = F(x, y) + psi_L(x, y) * 2 / h1;
100     }
101
102     /* right */
103     i = i_max - 1;
104     x = x_min + i * h1;
105     for (j = 0; j < j_max; j++) {
106         y = y_min + j * h2;
107         Matrix[i + 1][j + 1] = F(x, y) + psi_R(x, y) * 2 / h1;
108     }
109
110     /* bottom */
111     j = 0;
112     y = y_min + j * h2;
113     for (i = 0; i < i_max; i++) {
114         x = x_min + i * h1;
115         Matrix[i + 1][j + 1] = F(x, y) + psi_B(x, y) * 2 / h2;
116     }
117
118     /* top */
119     j = j_max - 1;
120     y = y_min + j * h2;
121     for (i = 0; i < i_max; i++) {
122         x = x_min + i * h1;
123         Matrix[i + 1][j + 1] = F(x, y) + psi_T(x, y) * 2 / h2;
124     }
125
126     /* left bottom */
127     i = 0; j = 0;
128     x = x_min + i * h1;
129     y = y_min + j * h2;
130     psi_ij = (psi_B(x, y) + psi_L(x, y)) / 2;
131     Matrix[i + 1][j + 1] = F(x, y) + psi_ij * (2 / h1 + 2 / h2);
132
133     /* left top */
134     i = 0; j = j_max - 1;
135     x = x_min + i * h1;
136     y = y_min + j * h2;
137     psi_ij = (psi_T(x, y) + psi_L(x, y)) / 2;
138     Matrix[i + 1][j + 1] = F(x, y) + psi_ij * (2 / h1 + 2 / h2);
139
140     /* right top */
141     i = i_max - 1; j = j_max - 1;
142     x = x_min + i * h1;
143     y = y_min + j * h2;
144     psi_ij = (psi_T(x, y) + psi_R(x, y)) / 2;
145     Matrix[i + 1][j + 1] = F(x, y) + psi_ij * (2 / h1 + 2 / h2);
146
147     /* right bottom */
148     i = i_max - 1; j = 0;
149     x = x_min + i * h1;
150     y = y_min + j * h2;
151     psi_ij = (psi_B(x, y) + psi_R(x, y)) / 2;
152     Matrix[i + 1][j + 1] = F(x, y) + psi_ij * (2 / h1 + 2 / h2);
153 }
154
155 void init_B(int M, int N, double (*B)[N + 2], double h1, double h2, double x_min, double y_min) {
156     int i, j;
157     double x, y;
158     int i_max = M;
159     int j_max = N;
160
161     for (i = 0; i <= i_max + 1; i++) {
162         for (j = 0; j <= j_max + 1; j++) {
163             x = x_min + (i - 1) * h1;
164             y = y_min + (j - 1) * h2;
165             B[i][j] = F(x, y);
166         }
167     }
168
169     init_B_border(M, N, B, h1, h2, x_min, y_min);
```

```
}

void A_mul_vec(int M, int N, double (*w)[N + 2], double (*res)[N + 2], double h1, double h2,
               double x_min, double y_min, int i_max, int j_max) {
    double aw, bw, x, y;
    int i, j;
    for (i = 0; i <= i_max + 1; i++) {
        for (j = 0; j <= j_max + 1; j++) {
            if (i == 0 || j == 0 || i == i_max + 1 || j == j_max + 1)
                res[i][j] = w[i][j];
            else {
                x = x_min + (i - 1) * h1;
                y = y_min + (j - 1) * h2;
                aw = 1 / h1 * (k(x + 0.5 * h1, y) * (w[i + 1][j] - w[i][j]) / h1 - \
                               k(x - 0.5 * h1, y) * (w[i][j] - w[i - 1][j]) / h1);
                bw = 1 / h2 * (k(x, y + 0.5 * h2) * (w[i][j + 1] - w[i][j]) / h2 - \
                               k(x, y - 0.5 * h2) * (w[i][j] - w[i][j - 1]) / h2);

                res[i][j] = -aw - bw;
            }
        }
    }

    /* left */
    for (j = 1; j <= j_max; j++) {
        y = y_min + (j - 1) * h2;

        x = x_min + h1;
        aw =  k(x - 0.5 * h1, y) * (w[2][j] - w[1][j]) / h1;

        x = x_min;
        bw = 1 / h2 * (k(x, y + 0.5 * h2) * (w[1][j + 1] - w[1][j]) / h2 - \
                       k(x, y - 0.5 * h2) * (w[1][j] - w[1][j - 1]) / h2);

        res[1][j] = -2/h1 * aw - bw + (2 / h1) * w[1][j];
    }

    /* right */
    for (j = 1; j <= j_max; j++) {
        y = y_min + (j - 1) * h2;

        x = x_min + (i_max - 1) * h1;
        aw = k(x - 0.5 * h1, y) * (w[i_max][j] - w[i_max - 1][j]) / h1;

        bw = 1 / h2 * (k(x, y + 0.5 * h2) * (w[i_max][j + 1] - w[i_max][j]) / h2 - \
                       k(x, y - 0.5 * h2) * (w[i_max][j] - w[i_max][j - 1]) / h2);

        res[i_max][j] = 2/h1 * aw - bw + (2 / h1) * w[i_max][j];
    }

    /* bottom */
    for (i = 1; i <= i_max; i++) {
        x = x_min + (i - 1) * h1;

        y = y_min;
        aw = 1 / h1 * (k(x + 0.5 * h1, y) * (w[i + 1][1] - w[i][1]) / h1 - \
                       k(x - 0.5 * h1, y) * (w[i][1] - w[i - 1][1]) / h1);

        y = y_min + h2;
        bw = k(x, y - 0.5 * h2) * (w[i][2] - w[i][1]) / h2;

        res[i][1] = -2/h2 * bw - aw;
    }

    /* top */
    for (i = 1; i <= i_max; i++) {
        x = x_min + (i - 1) * h1;

        y = y_min + (j_max - 1) * h2;
        aw = 1 / h1 * (k(x + 0.5 * h1, y) * (w[i + 1][j_max] - w[i][j_max]) / h1 - \
                       k(x - 0.5 * h1, y) * (w[i][j_max] - w[i - 1][j_max]) / h1);
        bw = k(x, y - 0.5 * h2) * (w[i][j_max] - w[i][j_max - 1]) / h2;

        res[i][j_max] = 2/h2 * bw - aw;
    }

    /* left bottom */
    x = x_min + h1; y = y_min;
    aw = k(x - 0.5 * h1, y) * (w[2][1] - w[1][1]) / h1;
    x = x_min; y = y_min + h2;
    bw = k(x, y - 0.5 * h2) * (w[1][2] - w[1][1]) / h2;
    res[1][1] = -2/h1 * aw - 2/h2 * bw + (2 / h1) * w[1][1];

    /* left top */
    x = x_min + h1; y = y_min + (j_max - 1) * h2;
    aw = k(x - 0.5 * h1, y) * (w[2][j_max] - w[1][j_max]) / h1;
    x = x_min; y = y_min + (j_max - 1) * h2;
    bw = k(x, y - 0.5 * h2) * (w[1][j_max] - w[1][j_max - 1]) / h2;
```

```
258         res[1][j_max] = -2/h1 * aw + 2/h2 * bw + (2 / h1) * w[1][j_max];
259
260         /* right top */
261         x = x_min + (i_max - 1) * h1; y = y_min + (j_max - 1) * h2;
262         aw = k(x - 0.5 * h1, y) * (w[i_max][j_max] - w[i_max - 1][j_max]) / h1;
263         bw = k(x, y - 0.5 * h2) * (w[i_max][j_max] - w[i_max][j_max - 1]) / h2;
264         res[i_max][j_max] = 2/h1 * aw + 2/h2 * bw + (2 / h1) * w[i_max][j_max];
265
266         /* right bottom */
267         x = x_min + (i_max - 1) * h1; y = y_min;
268         aw = k(x - 0.5 * h1, y) * (w[i_max][1] - w[i_max - 1][1]) / h1;
269         x = x_min + (i_max - 1) * h1; y = y_min + h2;
270         bw = k(x, y - 0.5 * h2) * (w[i_max][2] - w[i_max][1]) / h2;
271         res[i_max][1] = 2/h1 * aw - 2/h2 * bw + (2 / h1) * w[i_max][1];
272     }
273
274     double max_abs_matr(int M, int N, double (*w)[N + 2]) {
275         double res = 0;
276
277         for (int i = 0; i <= M; i++) {
278             for (int j = 0; j <= N; j++) {
279                 double value = fabs(w[i][j]);
280                 if (value > res)
281                     res = value;
282             }
283         }
284         return res;
285     }
286
287     double mean_abs_matr(int M, int N, double (*w)[N + 2]) {
288         double res = 0;
289
290         for (int i = 0; i <= M; i++) {
291             for (int j = 0; j <= N; j++) {
292                 res += fabs(w[i][j]);
293
294             }
295         }
296         return res / (M * N);
297     }
298
299     int main(int argc, char* argv[]) {
300         int M = atoi(argv[argc - 2]);
301         int N = atoi(argv[argc - 1]);
302         //const int N = 10;
303         //const int M = 10;
304
305         double h1 = (A2 - A1) / M;
306         double h2 = (B2 - B1) / N;
307
308         printf("h1 = %3.8f h2 = %3.8f\n", h1, h2);
309
310         double (*B)[N + 2] = malloc(sizeof(double[M + 2][N + 2]));
311         double (*w)[N + 2] = malloc(sizeof(double[M + 2][N + 2]));
312         double (*w_next)[N + 2] = malloc(sizeof(double[M + 2][N + 2]));
313         double (*Aw)[N + 2] = malloc(sizeof(double[M + 2][N + 2]));
314         double (*r)[N + 2] = malloc(sizeof(double[M + 2][N + 2]));
315         double (*Ar)[N + 2] = malloc(sizeof(double[M + 2][N + 2]));
316         double (*w_diff)[N + 2] = malloc(sizeof(double[M + 2][N + 2]));
317
318         double tau, Ar_norm, criteria;
319         int converge = 0;
320         int i, j, n_iter = 0; int max_iter = 10000;
321         int i_max = M; int j_max = N;
322
323         clock_t start_timestamp = clock();
324         init_B(M, N, B, h1, h2, A1, B1);
325
326         while (!converge) {
327             if (n_iter > max_iter) break;
328             A_mul_vec(M, N, w, Aw, h1, h2, A1, B1, M, N);
329
330             for (i = 0; i <= i_max; i++) {
331                 for (j = 0; j <= j_max; j++) {
332                     if (i == 0 || i == i_max || j == 0 || j == j_max) {
333                         r[i][j] = 0;
334                     }
335                     r[i][j] = Aw[i][j] - B[i][j];
336                 }
337             }
338
339             A_mul_vec(M, N, r, Ar, h1, h2, A1, B1, M, N);
340
341             Ar_norm = norm(M, N, Ar, h1, h2);
342             tau = dot_product(M, N, Ar, r, h1, h2) / (Ar_norm * Ar_norm);
343
344             for (i = 0; i <= i_max; i++) {
345                 for (j = 0; j <= j_max; j++) {
```

```
346                     w_next[i][j] = w[i][j] - tau * r[i][j];
347                     w_diff[i][j] = w_next[i][j] - w[i][j];
348                     w[i][j] = w_next[i][j];
349                 }
350             }
351 
352             criteria = norm(M, M, w_diff, h1, h2);
353             converge = criteria < EPSILON;
354             n_iter++;
355 
356             if (n_iter % 1000 == 0) {
357                 printf("iter %d criteria %3.8f dot %3.8f norm %3.8f tau %3.8f \n",
358                     n_iter, criteria, dot_product(M, N, Ar, r, h1, h2), Ar_norm, tau);
359             }
360         }
361         clock_t fin_timestamp = clock();
362         double time_taken = (double)(fin_timestamp - start_timestamp) / CLOCKS_PER_SEC;
363 
364         printf("Over (%d iters , time %f sec)\n", n_iter, time_taken);
365 
366 
367         free(w_diff); free(Ar); free(r);
368         free(Aw); free(w_next); free(w); free(B);
369 
370         return 0;
371 
372 }
```

Листинг MPI+OpenMP программы

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4 
5  #include "mpi.h"
6 
7  const double A1 = 0;
8  const double A2 = 2;
9  const double B1 = 0;
10 const double B2 = 1;
11 
12 const double EPSILON = 1e-6;
13 
14 
15 
16 double dudx(double x, double y) { return -sin(M_PI * x * y) * M_PI * y; }
17 
18 double dudy(double x, double y) { return -sin(M_PI * x * y) * M_PI * x; }
19 
20 double u(double x, double y){ return 1 + cos(M_PI * x * y); }
21 
22 double k(double x, double y){ return 4 + x + y; }
23 
24 double F(double x, double y){
25     return sin(M_PI * x * y) * M_PI * (x + y) +
26     k(x, y) * cos(M_PI * x * y) * M_PI * M_PI * (x * x + y * y);
27 }
28 
29 double psi_R(double x, double y){
30     return u(x, y) + k(x, y) * dudx(x, y);
31 }
32 
33 double psi_L(double x, double y){
34     return u(x, y) - k(x, y) * dudx(x, y);
35 }
36 
37 double psi_T(double x, double y){
38     return k(x, y) * dudy(x, y);
39 }
40 
41 double psi_B(double x, double y){
42     return -k(x, y) * dudy(x, y);
43 }
44 
45 
46 double rho(int i, int j, int i_max, int j_max,
47            int left_flag, int right_flag,
48            int top_flag, int bottom_flag) {
49     double result = 1;
50     if ((i == 1 && left_flag) || (i == i_max && right_flag)) {
51         result *= 0.5;
52     }
53     if ((j == 1 && bottom_flag) || (j == j_max && top_flag)) {
```

```
54          result *= 0.5;
55      }
56      return result;
57  }
58
59  void print_matrix(int M, int N, double (*Matrix)[N + 2], char* title) {
60      int curr_rank;
61      MPI_Comm_rank(MPI_COMM_WORLD, &curr_rank);
62      printf("%s\n", title);
63      for (int i = 0; i <= M + 1; i++) {
64          for (int j = 0; j <= N + 1; j++) {
65              printf("%3.8f(rank %d) ", Matrix[i][j], curr_rank);
66          }
67          printf("\n");
68      }
69  }
70
71  double dot_product(int M, int N, double (*X)[N + 2], double (*Y)[N + 2], double h1, double h2,
72                     int left_flag, int right_flag,
73                     int top_flag, int bottom_flag) {
74      double result = 0.;
75      int i, j;
76      int i_max = M;
77      int j_max = N;
78
79      #pragma omp parallel for private(i, j) reduction(+:result)
80      for (i = 1; i <= i_max; i++) {
81          for (j = 1; j <= j_max; j++) {
82              result += X[i][j] * Y[i][j] * rho(i, j, i_max, j_max, left_flag, right_flag, top_flag, bottom_flag) * h1 * h2;
83          }
84      }
85      return result;
86  }
87
88  double norm(int M, int N, double (*X)[N + 1], double h1, double h2,
89              int left_flag, int right_flag,
90              int top_flag, int bottom_flag) {
91      return dot_product(M, N, X, X, h1, h2, left_flag, right_flag, top_flag, bottom_flag);
92  }
93
94  void init_B_fill_border(int M, int N, double (*Matrix)[N + 2], double h1, double h2,
95                      double x_min, double y_min, int left_flag, int right_flag,
96                      int top_flag, int bottom_flag) {
97      int i, j;
98      double x, y; double psi_ij;
99      int i_max = M;
100     int j_max = N;
101
102     /* left */
103     if (left_flag) {
104         i = 0;
105         x = x_min + i * h1;
106
107         #pragma omp parallel for private(j, y)
108         for (j = 0; j < j_max; j++) {
109             y = y_min + j * h2;
110             Matrix[i + 1][j + 1] = F(x, y) + psi_L(x, y) * 2 / h1;
111         }
112     }
113
114     /* right */
115     if (right_flag) {
116         i = i_max - 1;
117         x = x_min + i * h1;
118
119         #pragma omp parallel for private(j, y)
120         for (j = 0; j < j_max; j++) {
121             y = y_min + j * h2;
122             Matrix[i + 1][j + 1] = F(x, y) + psi_R(x, y) * 2 / h1;
123         }
124     }
125
126     /* bottom */
127     if (bottom_flag) {
128         j = 0;
129         y = y_min + j * h2;
130
131         #pragma omp parallel for private(i, x)
132         for (i = 0; i < i_max; i++) {
133             x = x_min + i * h1;
134             Matrix[i + 1][j + 1] = F(x, y) + psi_B(x, y) * 2 / h2;
135         }
136     }
137
138     /* top */
139     if (top_flag) {
140         j = j_max - 1;
141         y = y_min + j * h2;
```

```
142
143         #pragma omp parallel for private(i, x)
144         for (i = 0; i < i_max; i++) {
145             x = x_min + i * h1;
146             Matrix[i + 1][j + 1] = F(x, y) + psi_T(x, y) * 2 / h2;
147         }
148     }
149
150     /* left bottom */
151     if (left_flag && bottom_flag) {
152         i = 0; j = 0;
153         x = x_min + i * h1;
154         y = y_min + j * h2;
155         psi_ij = (psi_B(x, y) + psi_L(x, y)) / 2;
156         Matrix[i + 1][j + 1] = F(x, y) + psi_ij * (2 / h1 + 2 / h2);
157     }
158
159     /* left top */
160     if (left_flag && top_flag) {
161         i = 0; j = j_max - 1;
162         x = x_min + i * h1;
163         y = y_min + j * h2;
164         psi_ij = (psi_T(x, y) + psi_L(x, y)) / 2;
165         Matrix[i + 1][j + 1] = F(x, y) + psi_ij * (2 / h1 + 2 / h2);
166     }
167
168     /* right top */
169     if (right_flag && top_flag) {
170         i = i_max - 1; j = j_max - 1;
171         x = x_min + i * h1;
172         y = y_min + j * h2;
173         psi_ij = (psi_T(x, y) + psi_R(x, y)) / 2;
174         Matrix[i + 1][j + 1] = F(x, y) + psi_ij * (2 / h1 + 2 / h2);
175     }
176
177     /* right bottom */
178     if (right_flag && bottom_flag) {
179         i = i_max - 1; j = 0;
180         x = x_min + i * h1;
181         y = y_min + j * h2;
182         psi_ij = (psi_B(x, y) + psi_R(x, y)) / 2;
183         Matrix[i + 1][j + 1] = F(x, y) + psi_ij * (2 / h1 + 2 / h2);
184     }
185 }
186
187 void init_B(int M, int N, double (*B)[N + 2], double h1, double h2, double x_min, double y_min) {
188     int i, j;
189     double x, y;
190     int i_max = M;
191     int j_max = N;
192
193     #pragma omp parallel for private(i, j, x, y)
194     for (i = 0; i <= i_max + 1; i++) {
195         for (j = 0; j <= j_max + 1; j++) {
196             x = x_min + (i - 1) * h1;
197             y = y_min + (j - 1) * h2;
198             B[i][j] = F(x, y);
199         }
200     }
201 }
202
203 void A_mul_vec_fill_border(int M, int N, double (*w)[N + 2], double (*res)[N + 2],
204                            double h1, double h2, double x_min, double y_min,
205                            int i_max, int j_max, int left_flag, int right_flag,
206                            int top_flag, int bottom_flag) {
207     double aw, bw, x, y;
208     int i, j;
209
210
211     /* left */
212     if (left_flag) {
213         #pragma omp parallel for private(j, x, y, aw, bw)
214         for (j = 1; j <= j_max; j++) {
215             y = y_min + (j - 1) * h2;
216
217             x = x_min + h1;
218             aw =  k(x - 0.5 * h1, y) * (w[2][j] - w[1][j]) / h1;
219
220             x = x_min;
221             bw = 1 / h2 * (k(x, y + 0.5 * h2) * (w[1][j + 1] - w[1][j]) / h2 - \
222                            k(x, y - 0.5 * h2) * (w[1][j] - w[1][j - 1]) / h2);
223
224             res[1][j] = -2/h1 * aw - bw + (2 / h1) * w[1][j];
225         }
226     }
227
228     /* right */
229     if (right_flag) {
```

```
230         #pragma omp parallel for private(j, x, y, aw, bw)
231         for (j = 1; j <= j_max; j++) {
232             y = y_min + (j - 1) * h2;
233
234             x = x_min + (i_max - 1) * h1;
235             aw = k(x - 0.5 * h1, y) * (w[i_max][j] - w[i_max - 1][j]) / h1;
236
237             bw = 1 / h2 * (k(x, y + 0.5 * h2) * (w[i_max][j + 1] - w[i_max][j]) / h2 - \
238                         k(x, y - 0.5 * h2) * (w[i_max][j] - w[i_max][j - 1]) / h2);
239
240             res[i_max][j] = 2/h1 * aw - bw + (2 / h1) * w[i_max][j];
241         }
242     }
243
244     /* bottom */
245     if (bottom_flag) {
246         #pragma omp parallel for private(i, x, y, aw, bw)
247         for (i = 1; i <= i_max; i++) {
248             x = x_min + (i - 1) * h1;
249
250             y = y_min;
251             aw = 1 / h1 * (k(x + 0.5 * h1, y) * (w[i + 1][1] - w[i][1]) / h1 - \
252                            k(x - 0.5 * h1, y) * (w[i][1] - w[i - 1][1]) / h1);
253
254             y = y_min + h2;
255             bw = k(x, y - 0.5 * h2) * (w[i][2] - w[i][1]) / h2;
256
257             res[i][1] = -2/h2 * bw - aw;
258         }
259     }
260
261     /* top */
262     if (top_flag) {
263         #pragma omp parallel for private(i, x, y, aw, bw)
264         for (i = 1; i <= i_max; i++) {
265             x = x_min + (i - 1) * h1;
266
267             y = y_min + (j_max - 1) * h2;
268             aw = 1 / h1 * (k(x + 0.5 * h1, y) * (w[i + 1][j_max] - w[i][j_max]) / h1 - \
269                            k(x - 0.5 * h1, y) * (w[i][j_max] - w[i - 1][j_max]) / h1);
270             bw = k(x, y - 0.5 * h2) * (w[i][j_max] - w[i][j_max - 1]) / h2;
271
272             res[i][j_max] = 2/h2 * bw - aw;
273         }
274     }
275
276     /* left bottom */
277     if (left_flag && bottom_flag) {
278         x = x_min + h1; y = y_min;
279         aw = k(x - 0.5 * h1, y) * (w[2][1] - w[1][1]) / h1;
280         x = x_min; y = y_min + h2;
281         bw = k(x, y - 0.5 * h2) * (w[1][2] - w[1][1]) / h2;
282         res[1][1] = -2/h1 * aw - 2/h2 * bw + (2 / h1) * w[1][1];
283     }
284
285     /* left top */
286     if (left_flag && top_flag) {
287         x = x_min + h1; y = y_min + (j_max - 1) * h2;
288         aw = k(x - 0.5 * h1, y) * (w[2][j_max] - w[1][j_max]) / h1;
289         x = x_min; y = y_min + (j_max - 1) * h2;
290         bw = k(x, y - 0.5 * h2) * (w[1][j_max] - w[1][j_max - 1]) / h2;
291         res[1][j_max] = -2/h1 * aw + 2/h2 * bw + (2 / h1) * w[1][j_max];
292     }
293
294     /* right top */
295     if (right_flag && top_flag) {
296         x = x_min + (i_max - 1) * h1; y = y_min + (j_max - 1) * h2;
297         aw = k(x - 0.5 * h1, y) * (w[i_max][j_max] - w[i_max - 1][j_max]) / h1;
298         bw = k(x, y - 0.5 * h2) * (w[i_max][j_max] - w[i_max][j_max - 1]) / h2;
299         res[i_max][j_max] = 2/h1 * aw + 2/h2 * bw + (2 / h1) * w[i_max][j_max];
300     }
301
302     /* right bottom */
303     if (right_flag && bottom_flag) {
304         x = x_min + (i_max - 1) * h1; y = y_min;
305         aw = k(x - 0.5 * h1, y) * (w[i_max][1] - w[i_max - 1][1]) / h1;
306         x = x_min + (i_max - 1) * h1; y = y_min + h2;
307         bw = k(x, y - 0.5 * h2) * (w[i_max][2] - w[i_max][1]) / h2;
308         res[i_max][1] = 2/h1 * aw - 2/h2 * bw + (2 / h1) * w[i_max][1];
309     }
310 }
311
312 void A_mul_vec(int M, int N, double (*w)[N + 2], double (*res)[N + 2], double h1, double h2,
313                double x_min, double y_min, int i_max, int j_max) {
314     double aw, bw, x, y;
315     int i, j;
316
317     #pragma omp parallel for private(i, j, x, y, aw, bw)
```

```
318     for (i = 0; i <= i_max + 1; i++) {
319         for (j = 0; j <= j_max + 1; j++) {
320             if (i == 0 || j == 0 || i == i_max + 1 || j == j_max + 1)
321                 res[i][j] = w[i][j];
322             else {
323 
324                 x = x_min + (i - 1) * h1;
325                 y = y_min + (j - 1) * h2;
326                 aw = 1 / h1 * (k(x + 0.5 * h1, y) * (w[i + 1][j] - w[i][j]) / h1 - \
327                                k(x - 0.5 * h1, y) * (w[i][j] - w[i - 1][j]) / h1);
328                 bw = 1 / h2 * (k(x, y + 0.5 * h2) * (w[i][j + 1] - w[i][j]) / h2 - \
329                         k(x, y - 0.5 * h2) * (w[i][j] - w[i][j - 1]) / h2);
330 
331                 res[i][j] = -aw - bw;
332             }
333         }
334     }
335 }
336 
337 void sync_borders(int x_n, int y_n, MPI_Comm MPI_COMM_CARTESIAN_TOPOLOGY, int left_flag, int right_flag,
338                   int top_flag, int bottom_flag, int process_dims[2], int curr_coords[2],
339                   double left_send_buf[y_n], double right_send_buf[y_n], double top_send_buf[x_n], double bottom_send_buf[x_
340                   double left_recv_buf[y_n], double right_recv_buf[y_n], double top_recv_buf[x_n], double bottom_recv_buf[x_
341                   double (*w)[y_n + 2], int MPI_tag, double x_min, double y_min, double h1, double h2, int i_x, int i_y) {
342     int near_coords[2];
343     int near_rank; int i, j;
344 
345     MPI_Status status; MPI_Request requests[4] = {MPI_REQUEST_NULL, MPI_REQUEST_NULL, MPI_REQUEST_NULL, MPI_REQUEST_NULL};;
346 
347     // left send
348     if ((left_flag == 0) && (process_dims[0] > 1)) {
349         near_coords[0] = curr_coords[0] - 1;
350         near_coords[1] = curr_coords[1];
351 
352         for (j = 0; j < y_n; j++) {
353             left_send_buf[j] = w[1][j + 1];
354         }
355 
356         MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
357         MPI_Isend(left_send_buf, y_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &requests[0]);
358     }
359 
360     // right send
361     if ((right_flag == 0) && (process_dims[0] > 1)) {
362         near_coords[0] = curr_coords[0] + 1;
363         near_coords[1] = curr_coords[1];
364 
365         for (j = 0; j < y_n; j++) {
366             right_send_buf[j] = w[x_n][j + 1];
367         }
368 
369         MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
370         MPI_Isend(right_send_buf, y_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &requests[1]);
371     }
372 
373     // top send
374     if ((top_flag == 0) && (process_dims[1] > 1)) {
375         near_coords[0] = curr_coords[0];
376         near_coords[1] = curr_coords[1] + 1;
377 
378         for (i = 0; i < x_n; i++) {
379             top_send_buf[i] = w[i + 1][y_n];
380         }
381 
382         MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
383         MPI_Isend(top_send_buf, x_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &requests[2]);
384     }
385 
386     // bottom send
387     if ((bottom_flag == 0) && (process_dims[1] > 1)) {
388         near_coords[0] = curr_coords[0];
389         near_coords[1] = curr_coords[1] - 1;
390 
391         for (i = 0; i < x_n; i++) {
392             bottom_send_buf[i] = w[i + 1][1];
393         }
394 
395         MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
396         MPI_Isend(bottom_send_buf, x_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &requests[3]);
397     }
398 
399     // left recieve
400     if ((left_flag && (process_dims[0] > 1)) || (process_dims[0] == 1)) {
401         #pragma omp parallel for private(j)
402         for (j = 0; j < y_n; j++) {
403             w[0][j + 1] = u(x_min + (i_x - 1) * h1, y_min + (i_y + j) * h2);
404         }
405     }
```

```
406         else {
407             near_coords[0] = curr_coords[0] - 1;
408             near_coords[1] = curr_coords[1];
409             MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
410             MPI_Recv(left_recv_buf, y_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &status);
411             #pragma omp parallel for private(j)
412             for (j = 0; j < y_n; j++) {
413                 w[0][j + 1] = left_recv_buf[j];
414             }
415         }
416 
417         // right recieve
418         if ((right_flag && (process_dims[0] > 1)) || (process_dims[0] == 1)) {
419             #pragma omp parallel for private(j)
420             for (j = 0; j < y_n; j++) {
421                 w[x_n + 1][j + 1] = u(x_min + (i_x + x_n) * h1, y_min + (i_y + j) * h2);
422             }
423         }
424         else {
425             near_coords[0] = curr_coords[0] + 1;
426             near_coords[1] = curr_coords[1];
427             MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
428             MPI_Recv(right_recv_buf, y_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &status);
429             #pragma omp parallel for private(j)
430             for (j = 0; j < y_n; j++) {
431                 w[x_n + 1][j + 1] = right_recv_buf[j];
432             }
433         }
434 
435         // top recieve
436         if ((top_flag && (process_dims[1] > 1)) || (process_dims[1] == 1)) {
437             #pragma omp parallel for private(i)
438             for (i = 0; i < x_n; i++) {
439                 w[i + 1][y_n + 1] = u(x_min + (i_x + i) * h1, y_min + (i_y + y_n) * h2);
440             }
441         }
442         else {
443             near_coords[0] = curr_coords[0];
444             near_coords[1] = curr_coords[1] + 1;
445             MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
446             MPI_Recv(top_recv_buf, x_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &status);
447             #pragma omp parallel for private(i)
448             for (i = 0; i < x_n; i++) {
449                 w[i + 1][y_n + 1] = top_recv_buf[i];
450             }
451         }
452 
453         // bottom recieve
454         if ((bottom_flag && (process_dims[1] > 1)) || (process_dims[1] == 1)) {
455             #pragma omp parallel for private(i)
456             for (i = 0; i < x_n; i++) {
457                 w[i + 1][0] = u(x_min + (i_x + i) * h1, y_min + (i_y - 1) * h2);
458             }
459         }
460         else {
461             near_coords[0] = curr_coords[0];
462             near_coords[1] = curr_coords[1] - 1;
463             MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
464             MPI_Recv(bottom_recv_buf, x_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &status);
465             #pragma omp parallel for private(i)
466             for (i = 0; i < x_n; i++) {
467                 w[i + 1][0] = bottom_recv_buf[i];
468             }
469         }
470 
471         // MPI_Waitall(4, requests, MPI_STATUSES_IGNORE);
472 
473         for (i = 0; i <= 3; i++) {
474            MPI_Wait(&requests[i], &status);
475         }
476     }
477 
478     double max_abs_matr(int M, int N, double (*w)[N + 2]) {
479         double res = 0;
480 
481         for (int i = 0; i <= M; i++) {
482             for (int j = 0; j <= N; j++) {
483                 double value = fabs(w[i][j]);
484                 if (value > res)
485                     res = value;
486             }
487         }
488         return res;
489     }
490 
491     double mean_abs_matr(int M, int N, double (*w)[N + 2]) {
492         double res = 0;
493 
```

```
494         for (int i = 0; i <= M; i++) {
495             for (int j = 0; j <= N; j++) {
496                 res += fabs(w[i][j]);
497 
498             }
499         }
500         return res / (M * N);
501     }
502 
503     int main(int argc, char* argv[]) {
504         int M = atoi(argv[argc - 2]);
505         int N = atoi(argv[argc - 1]);
506         // const int N = 10;
507         // const int M = 10;
508 
509         double h1 = (A2 - A1) / M;
510         double h2 = (B2 - B1) / N;
511 
512         printf("h1 = %3.8f h2 = %3.8f\n", h1, h2);
513 
514         MPI_Init(&argc, &argv);
515 
516         // MPI variables
517         int curr_rank, n_processes; int MPI_tag = 0;
518         int process_dims[2] = {0, 0};
519         int grid_is_periodic[2] = {0, 0};
520         int curr_coords[2];
521         int left_flag = 0, right_flag = 0, top_flag = 0, bottom_flag = 0;
522 
523         MPI_Comm_rank(MPI_COMM_WORLD, &curr_rank);
524         MPI_Comm_size(MPI_COMM_WORLD, &n_processes);
525 
526         MPI_Comm MPI_COMM_CARTESIAN_TOPOLOGY;
527         MPI_Dims_create(n_processes, 2, process_dims);
528         MPI_Cart_create(MPI_COMM_WORLD, 2, process_dims, grid_is_periodic, 1, &MPI_COMM_CARTESIAN_TOPOLOGY);
529         MPI_Cart_coords(MPI_COMM_CARTESIAN_TOPOLOGY, curr_rank, 2, curr_coords);
530 
531         if (curr_coords[0] == 0) {
532             left_flag = 1;
533         }
534         if (curr_coords[1] == 0) {
535             bottom_flag = 1;
536         }
537         if ((curr_coords[0] + 1) == process_dims[0]) {
538             right_flag = 1;
539         }
540         if ((curr_coords[1] + 1) == process_dims[1]) {
541             top_flag = 1;
542         }
543 
544         printf("rank %d left %d right %d top %d bottom %d \n", curr_rank, left_flag, right_flag, top_flag, bottom_flag);
545 
546         int x_i, x_n, y_i, y_n;
547         if ((M + 1) % process_dims[0] == 0) {
548             x_n = (M + 1) / process_dims[0];
549             x_i = curr_coords[0] * x_n;
550         }
551         else {
552             if (curr_coords[0] == 0) {
553                 x_n = (M + 1) / process_dims[0] + (M + 1) % process_dims[0];
554                 x_i = 0;
555             }
556             else {
557                 x_n = (M + 1) / process_dims[0];
558                 x_i = curr_coords[0] * x_n + (M + 1) % process_dims[0];
559             }
560         }
561 
562         if ((N + 1) % process_dims[1] == 0) {
563             y_n = (N + 1) / process_dims[1];
564             y_i = curr_coords[1] * y_n;
565         }
566         else {
567             if (curr_coords[1] == 0) {
568                 y_n = (N + 1) / process_dims[1] + (N + 1) % process_dims[1];
569                 y_i = 0;
570             }
571             else {
572                 y_n = (N + 1) / process_dims[1];
573                 y_i = curr_coords[1] * y_n + (N + 1) % process_dims[1];
574             }
575         }
576 
577         printf("rank %d x_i %d y_i %d x_n %d y_n %d \n", curr_rank, x_i, y_i, x_n, y_n);
578 
579         double *left_send_buf = malloc(sizeof(double[y_n])); double *right_send_buf = malloc(sizeof(double[y_n]));
580         double *left_recv_buf = malloc(sizeof(double[y_n])); double *right_recv_buf = malloc(sizeof(double[y_n]));
581         double *top_send_buf = malloc(sizeof(double[x_n])); double *bottom_send_buf = malloc(sizeof(double[x_n]));
```

```
582     double *top_recv_buf = malloc(sizeof(double[x_n])); double *bottom_recv_buf = malloc(sizeof(double[x_n]));
583
584     // algorithm variables
585     int converge = 0;
586     int i, j, n_iter = 0;
587     int i_max = x_n; int j_max = y_n;
588     double CRITERIA = 0; double criteria = 0; int MAX_ITER = 10000;
589     double TAU, TAU_DOT, TAU_NORM; double tau_dot, tau_norm;
590
591     double (*B)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
592     double (*w)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
593     double (*w_next)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
594     double (*Aw)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
595     double (*r)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
596     double (*Ar)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
597     double (*w_diff)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
598     double (*true_u)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
599
600     double start_timestamp = MPI_Wtime();
601
602     init_B(x_n, y_n, B, h1, h2, A1 + h1 * x_i, B1 + h2 * y_i);
603     init_B_fill_border(x_n, y_n, B, h1, h2, A1 + h1 * x_i, B1 + h2 * y_i, left_flag, right_flag, top_flag, bottom_flag);
604
605     #pragma omp parallel for private(i, j)
606     for (i = 0; i <= x_n + 1; i++) {
607         for (j = 0; j <= y_n; j++) {
608             w[i][j] = 0;
609         }
610     }
611
612
613     while (!converge) {
614         if (n_iter > MAX_ITER) break;
615
616         sync_borders(x_n, y_n, MPI_COMM_CARTESIAN_TOPOLOGY, left_flag, right_flag,
617                      top_flag, bottom_flag, process_dims, curr_coords,
618                      left_send_buf, right_send_buf, top_send_buf, bottom_send_buf,
619                      left_recv_buf, right_recv_buf, top_recv_buf, bottom_recv_buf,
620                      w, MPI_tag, A1, B1, h1, h2, x_i, y_i);
621
622         A_mul_vec(x_n, y_n, w, Aw, h1, h2, A1 + h1 * x_i, B1 + h2 * y_i, x_n, y_n);
623
624
625         A_mul_vec_fill_border(x_n, y_n, w, Aw, h1, h2, A1 + h1 * x_i, B1 + h2 * y_i,
626                               x_n, y_n, left_flag, right_flag, top_flag, bottom_flag);
627
628
629         #pragma omp parallel for private(i, j)
630         for (i = 0; i <= x_n + 1; i++) {
631             for (j = 0; j <= y_n + 1; j++) {
632                 if (i == 0 || i == x_n + 1 || j == 0 || j == y_n + 1) {
633                     r[i][j] = 0;
634                 }
635                 else {
636                     r[i][j] = Aw[i][j] - B[i][j];
637                 }
638             }
639         }
640
641         sync_borders(x_n, y_n, MPI_COMM_CARTESIAN_TOPOLOGY, left_flag, right_flag,
642                      top_flag, bottom_flag, process_dims, curr_coords,
643                      left_send_buf, right_send_buf, top_send_buf, bottom_send_buf,
644                      left_recv_buf, right_recv_buf, top_recv_buf, bottom_recv_buf,
645                      r, MPI_tag, A1, B1, h1, h2, x_i, y_i);
646
647         A_mul_vec(x_n, y_n, r, Ar, h1, h2, A1 + h1 * x_i, B1 + h2 * y_i, x_n, y_n);
648         A_mul_vec_fill_border(x_n, y_n, r, Ar, h1, h2, A1 + h1 * x_i, B1 + h2 * y_i,
649                               x_n, y_n, left_flag, right_flag, top_flag, bottom_flag);
650
651
652         tau_norm = norm(x_n, y_n, Ar, h1, h2, left_flag, right_flag, top_flag, bottom_flag);
653         tau_dot = dot_product(x_n, y_n, Ar, r, h1, h2, left_flag, right_flag, top_flag, bottom_flag);
654
655         MPI_Allreduce(&tau_norm, &TAU_NORM, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_CARTESIAN_TOPOLOGY);
656         MPI_Allreduce(&tau_dot, &TAU_DOT, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_CARTESIAN_TOPOLOGY);
657
658         TAU = TAU_DOT / TAU_NORM;
659
660         #pragma omp parallel for private(i, j)
661         for (i = 1; i <= x_n; i++) {
662             for (j = 1; j <= y_n; j++) {
663                 w_next[i][j] = w[i][j] - TAU * r[i][j];
664                 w_diff[i][j] = w_next[i][j] - w[i][j];
665                 w[i][j] = w_next[i][j];
666             }
667         }
668
669         criteria = sqrt(norm(x_n, y_n, w_diff, h1, h2, left_flag, right_flag, top_flag, bottom_flag));
```

```
670             MPI_Allreduce(&criteria , &CRITERIA, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_CARTESIAN_TOPOLOGY);
671
672             converge = CRITERIA < EPSILON;
673             n_iter++;
674             if (curr_rank == 0) {
675                 if (n_iter % 1000 == 0) {
676                     printf("iter %d criteria %3.8f dot %3.8f norm %3.8f tau %3.8f \n",
677                       n_iter, CRITERIA, TAU_DOT, TAU_NORM, TAU);
678                 }
679             }
680         }
681
682         double fin_timestamp = MPI_Wtime();
683         if (curr_rank == 0) {
684             printf("Over (rank %d, %d iters , time %f sec)\n", curr_rank, n_iter, fin_timestamp - start_timestamp);
685         }
686
687         MPI_Barrier(MPI_COMM_WORLD);
688
689         free(left_send_buf); free(left_recv_buf);
690         free(right_send_buf); free(right_recv_buf);
691         free(top_send_buf); free(top_recv_buf);
692         free(bottom_send_buf); free(bottom_recv_buf);
693         free(true_u); free(w_diff); free(Ar); free(r);
694         free(Aw); free(w_next); free(w); free(B);
695
696         MPI_Finalize();
697         return 0;
698
699     }
```

Листинг MPI+OpenMP+OpenACC программы

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4
5   #include <mpi.h>
6
7   const double A1 = 0;
8   const double A2 = 2;
9   const double B1 = 0;
10  const double B2 = 1;
11
12  const double EPSILON = 1e-6;
13
14
15
16
17  # pragma acc routine
18  double dudx(double x, double y) { return -sin(M_PI * x * y) * M_PI * y; }
19
20  # pragma acc routine
21  double dudy(double x, double y) { return -sin(M_PI * x * y) * M_PI * x; }
22
23  # pragma acc routine
24  double u(double x, double y){ return 1 + cos(M_PI * x * y); }
25
26  # pragma acc routine
27  double k(double x, double y){ return 4 + x + y; }
28
29  # pragma acc routine
30  double F(double x, double y){
31      return sin(M_PI * x * y) * M_PI * (x + y) +
32      k(x, y) * cos(M_PI * x * y) * M_PI * M_PI * (x * x + y * y);
33  }
34
35  # pragma acc routine
36  double psi_R(double x, double y){
37      return u(x, y) + k(x, y) * dudx(x, y);
38  }
39
40  # pragma acc routine
41  double psi_L(double x, double y){
42      return u(x, y) - k(x, y) * dudx(x, y);
43  }
44
45  # pragma acc routine
46  double psi_T(double x, double y){
47      return k(x, y) * dudy(x, y);
48  }
49
50  # pragma acc routine
```

```
51  double psi_B(double x, double y){
52      return -k(x, y) * dudy(x, y);
53  }
54
55  # pragma acc routine
56  double rho(int i, int j, int i_max, int j_max,
57             int left_flag, int right_flag,
58             int top_flag, int bottom_flag) {
59      double result = 1;
60      if ((i == 1 && left_flag) || (i == i_max && right_flag)) {
61          result *= 0.5;
62      }
63      if ((j == 1 && bottom_flag) || (j == j_max && top_flag)) {
64          result *= 0.5;
65      }
66      return result;
67  }
68
69  void print_matrix(int M, int N, double (*Matrix)[N + 2], char* title) {
70      int curr_rank;
71      MPI_Comm_rank(MPI_COMM_WORLD, &curr_rank);
72      printf("%s\n", title);
73      for (int i = 0; i <= M + 1; i++) {
74          for (int j = 0; j <= N + 1; j++) {
75              printf("%3.8f(rank␣%d)␣", Matrix[i][j], curr_rank);
76          }
77          printf("\n");
78      }
79  }
80
81  double dot_product(int M, int N, double (*restrict X)[N + 2], double (*restrict Y)[N + 2], double h1, double h2,
82                     int left_flag, int right_flag,
83                     int top_flag, int bottom_flag) {
84      double result = 0.;
85
86      #pragma acc data present(X[:M + 2][:N + 2], Y[:M + 2][:N + 2])
87      {
88      # pragma acc parallel loop independent reduction(+:result)
89      for (int i = 1; i <= M; i++) {
90          for (int j = 1; j <= N; j++) {
91              result += X[i][j] * Y[i][j] * rho(i, j, M, N, left_flag, right_flag, top_flag, bottom_flag) * h1 * h2;
92          }
93      }
94      }
95      return result;
96  }
97
98  double norm(int M, int N, double (*X)[N + 1], double h1, double h2,
99              int left_flag, int right_flag,
100             int top_flag, int bottom_flag) {
101     return dot_product(M, N, X, X, h1, h2, left_flag, right_flag, top_flag, bottom_flag);
102 }
103
104 void init_B_fill_border(int M, int N, double (*Matrix)[N + 2], double h1, double h2,
105                         double x_min, double y_min, int left_flag, int right_flag,
106                         int top_flag, int bottom_flag) {
107     int i, j;
108     double x, y; double psi_ij;
109     int i_max = M;
110     int j_max = N;
111
112     /* left */
113     if (left_flag) {
114         i = 0;
115         x = x_min + i * h1;
116
117         #pragma omp parallel for private(j, y)
118         for (j = 0; j < j_max; j++) {
119             y = y_min + j * h2;
120             Matrix[i + 1][j + 1] = F(x, y) + psi_L(x, y) * 2 / h1;
121         }
122     }
123
124     /* right */
125     if (right_flag) {
126         i = i_max - 1;
127         x = x_min + i * h1;
128
129         #pragma omp parallel for private(j, y)
130         for (j = 0; j < j_max; j++) {
131             y = y_min + j * h2;
132             Matrix[i + 1][j + 1] = F(x, y) + psi_R(x, y) * 2 / h1;
133         }
134     }
135
136     /* bottom */
137     if (bottom_flag) {
138         j = 0;
```

```
139             y = y_min + j * h2;
140
141             #pragma omp parallel for private(i, x)
142             for (i = 0; i < i_max; i++) {
143                 x = x_min + i * h1;
144                 Matrix[i + 1][j + 1] = F(x, y) + psi_B(x, y) * 2 / h2;
145             }
146         }
147
148         /* top */
149         if (top_flag) {
150             j = j_max - 1;
151             y = y_min + j * h2;
152
153             #pragma omp parallel for private(i, x)
154             for (i = 0; i < i_max; i++) {
155                 x = x_min + i * h1;
156                 Matrix[i + 1][j + 1] = F(x, y) + psi_T(x, y) * 2 / h2;
157             }
158         }
159
160         /* left bottom */
161         if (left_flag && bottom_flag) {
162             i = 0; j = 0;
163             x = x_min + i * h1;
164             y = y_min + j * h2;
165             psi_ij = (psi_B(x, y) + psi_L(x, y)) / 2;
166             Matrix[i + 1][j + 1] = F(x, y) + psi_ij * (2 / h1 + 2 / h2);
167         }
168
169         /* left top */
170         if (left_flag && top_flag) {
171             i = 0; j = j_max - 1;
172             x = x_min + i * h1;
173             y = y_min + j * h2;
174             psi_ij = (psi_T(x, y) + psi_L(x, y)) / 2;
175             Matrix[i + 1][j + 1] = F(x, y) + psi_ij * (2 / h1 + 2 / h2);
176         }
177
178         /* right top */
179         if (right_flag && top_flag) {
180             i = i_max - 1; j = j_max - 1;
181             x = x_min + i * h1;
182             y = y_min + j * h2;
183             psi_ij = (psi_T(x, y) + psi_R(x, y)) / 2;
184             Matrix[i + 1][j + 1] = F(x, y) + psi_ij * (2 / h1 + 2 / h2);
185         }
186
187         /* right bottom */
188         if (right_flag && bottom_flag) {
189             i = i_max - 1; j = 0;
190             x = x_min + i * h1;
191             y = y_min + j * h2;
192             psi_ij = (psi_B(x, y) + psi_R(x, y)) / 2;
193             Matrix[i + 1][j + 1] = F(x, y) + psi_ij * (2 / h1 + 2 / h2);
194         }
195     }
196
197     void init_B(int M, int N, double (*B)[N + 2], double h1, double h2, double x_min, double y_min) {
198         int i, j;
199         double x, y;
200         int i_max = M;
201         int j_max = N;
202
203         #pragma omp parallel for private(i, j, x, y)
204         for (i = 0; i <= i_max + 1; i++) {
205             for (j = 0; j <= j_max + 1; j++) {
206                 x = x_min + (i - 1) * h1;
207                 y = y_min + (j - 1) * h2;
208                 B[i][j] = F(x, y);
209             }
210         }
211     }
212
213     void A_mul_vec_fill_border(int M, int N, double (*restrict w)[N + 2], double (*restrict res)[N + 2],
214                                double h1, double h2, double x_min, double y_min,
215                                int i_max, int j_max, int left_flag, int right_flag,
216                                int top_flag, int bottom_flag) {
217         #pragma acc data present(w[:M + 2][:N + 2], res[:M + 2][:N + 2])
218         {
219
220
221         /* left */
222         if (left_flag) {
223             #pragma acc parallel loop
224             for (int j = 1; j <= j_max; j++) {
225                 double aw, bw, x, y;
226                 y = y_min + (j - 1) * h2;
```

```
227
228             x = x_min + h1;
229             aw =  k(x - 0.5 * h1, y) * (w[2][j] - w[1][j]) / h1;
230
231             x = x_min;
232             bw = 1 / h2 * (k(x, y + 0.5 * h2) * (w[1][j + 1] - w[1][j]) / h2 - \
233                            k(x, y - 0.5 * h2) * (w[1][j] - w[1][j - 1]) / h2);
234
235             res[1][j] = -2/h1 * aw - bw + (2 / h1) * w[1][j];
236         }
237     }
238
239     /* right */
240     if (right_flag) {
241         #pragma acc parallel loop
242         for (int j = 1; j <= j_max; j++) {
243             double aw, bw, x, y;
244             y = y_min + (j - 1) * h2;
245
246             x = x_min + (i_max - 1) * h1;
247             aw = k(x - 0.5 * h1, y) * (w[i_max][j] - w[i_max - 1][j]) / h1;
248
249             bw = 1 / h2 * (k(x, y + 0.5 * h2) * (w[i_max][j + 1] - w[i_max][j]) / h2 - \
250                            k(x, y - 0.5 * h2) * (w[i_max][j] - w[i_max][j - 1]) / h2);
251
252             res[i_max][j] = 2/h1 * aw - bw + (2 / h1) * w[i_max][j];
253         }
254     }
255
256     /* bottom */
257     if (bottom_flag) {
258         #pragma acc parallel loop
259         for (int i = 1; i <= i_max; i++) {
260             double aw, bw, x, y;
261             x = x_min + (i - 1) * h1;
262
263             y = y_min;
264             aw = 1 / h1 * (k(x + 0.5 * h1, y) * (w[i + 1][1] - w[i][1]) / h1 - \
265                            k(x - 0.5 * h1, y) * (w[i][1] - w[i - 1][1]) / h1);
266
267             y = y_min + h2;
268             bw = k(x, y - 0.5 * h2) * (w[i][2] - w[i][1]) / h2;
269
270             res[i][1] = -2/h2 * bw - aw;
271         }
272     }
273
274     /* top */
275     if (top_flag) {
276         #pragma acc parallel loop
277         for (int i = 1; i <= i_max; i++) {
278             double aw, bw, x, y;
279             x = x_min + (i - 1) * h1;
280
281             y = y_min + (j_max - 1) * h2;
282             aw = 1 / h1 * (k(x + 0.5 * h1, y) * (w[i + 1][j_max] - w[i][j_max]) / h1 - \
283                            k(x - 0.5 * h1, y) * (w[i][j_max] - w[i - 1][j_max]) / h1);
284             bw = k(x, y - 0.5 * h2) * (w[i][j_max] - w[i][j_max - 1]) / h2;
285
286             res[i][j_max] = 2/h2 * bw - aw;
287         }
288     }
289
290
291
292     double aw, bw, x, y;
293
294     /* left bottom */
295     if (left_flag && bottom_flag) {
296         x = x_min + h1; y = y_min;
297         aw = k(x - 0.5 * h1, y) * (w[2][1] - w[1][1]) / h1;
298         x = x_min; y = y_min + h2;
299         bw = k(x, y - 0.5 * h2) * (w[1][2] - w[1][1]) / h2;
300         res[1][1] = -2/h1 * aw - 2/h2 * bw + (2 / h1) * w[1][1];
301     }
302
303     /* left top */
304     if (left_flag && top_flag) {
305         x = x_min + h1; y = y_min + (j_max - 1) * h2;
306         aw = k(x - 0.5 * h1, y) * (w[2][j_max] - w[1][j_max]) / h1;
307         x = x_min; y = y_min + (j_max - 1) * h2;
308         bw = k(x, y - 0.5 * h2) * (w[1][j_max] - w[1][j_max - 1]) / h2;
309         res[1][j_max] = -2/h1 * aw + 2/h2 * bw + (2 / h1) * w[1][j_max];
310     }
311
312     /* right top */
313     if (right_flag && top_flag) {
314         x = x_min + (i_max - 1) * h1; y = y_min + (j_max - 1) * h2;
```

```
315             aw = k(x - 0.5 * h1, y) * (w[i_max][j_max] - w[i_max - 1][j_max]) / h1;
316             bw = k(x, y - 0.5 * h2) * (w[i_max][j_max] - w[i_max][j_max - 1]) / h2;
317             res[i_max][j_max] = 2/h1 * aw + 2/h2 * bw + (2 / h1) * w[i_max][j_max];
318         }
319
320         /* right bottom */
321         if (right_flag && bottom_flag) {
322             x = x_min + (i_max - 1) * h1; y = y_min;
323             aw = k(x - 0.5 * h1, y) * (w[i_max][1] - w[i_max - 1][1]) / h1;
324             x = x_min + (i_max - 1) * h1; y = y_min + h2;
325             bw = k(x, y - 0.5 * h2) * (w[i_max][2] - w[i_max][1]) / h2;
326             res[i_max][1] = 2/h1 * aw - 2/h2 * bw + (2 / h1) * w[i_max][1];
327         }
328         }
329     }
330
331     void A_mul_vec(int M, int N, double (*restrict w)[N + 2], double (*restrict res)[N + 2], double h1, double h2, double x_min
332
333         #pragma acc data present(w[:M + 2][:N + 2], res[:M + 2][:N + 2])
334         {
335         # pragma acc parallel
336         {
337         # pragma acc loop independent
338         for (int i = 0; i <= M + 1; i++) {
339             res[i][0] = w[i][0];
340             res[i][N + 1] = w[i][N + 1];
341         }
342
343         # pragma acc loop independent
344         for (int j = 0; j <= N + 1; j++) {
345             res[0][j] = w[0][j];
346             res[M + 1][j] = w[M + 1][j];
347         }
348
349         # pragma acc loop independent
350         for (int i = 1; i <= M; i++) {
351             # pragma acc loop independent
352             for (int j = 1; j <= N; j++) {
353                     double x = x_min + (i - 1) * h1;
354                     double y = y_min + (j - 1) * h2;
355                     double aw = 1 / h1 * (k(x + 0.5 * h1, y) * (w[i + 1][j] - w[i][j]) / h1 - \
356                                           k(x - 0.5 * h1, y) * (w[i][j] - w[i - 1][j]) / h1);
357                     double bw = 1 / h2 * (k(x, y + 0.5 * h2) * (w[i][j + 1] - w[i][j]) / h2 - \
358                                           k(x, y - 0.5 * h2) * (w[i][j] - w[i][j - 1]) / h2);
359
360                     res[i][j] = -aw - bw;
361             }
362         }
363         }
364         }
365
366     }
367
368     void sync_borders(int x_n, int y_n, MPI_Comm MPI_COMM_CARTESIAN_TOPOLOGY, int left_flag, int right_flag,
369                       int top_flag, int bottom_flag, int process_dims[2], int curr_coords[2],
370                       double left_send_buf[y_n], double right_send_buf[y_n], double top_send_buf[x_n], double bottom_send_buf[x_
371                       double left_recv_buf[y_n], double right_recv_buf[y_n], double top_recv_buf[x_n], double bottom_recv_buf[x_
372                       double (*w)[y_n + 2], int MPI_tag, double x_min, double y_min, double h1, double h2, int i_x, int i_y) {
373         int near_coords[2];
374         int near_rank; int i, j;
375
376         MPI_Status status; MPI_Request requests[4] = {MPI_REQUEST_NULL, MPI_REQUEST_NULL, MPI_REQUEST_NULL, MPI_REQUEST_NULL};;
377
378         // left send
379         if ((left_flag == 0) && (process_dims[0] > 1)) {
380             near_coords[0] = curr_coords[0] - 1;
381             near_coords[1] = curr_coords[1];
382
383             #pragma acc data present(w[:x_n + 2][y_n + 2], left_send_buf[:y_n])
384             {
385             for (j = 0; j < y_n; j++) {
386                 left_send_buf[j] = w[1][j + 1];
387             }
388             }
389
390             # pragma acc data copyout(left_send_buf[:y_n])
391             {
392             MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
393             MPI_Isend(left_send_buf, y_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &requests[0]);
394             }
395         }
396
397         // right send
398         if ((right_flag == 0) && (process_dims[0] > 1)) {
399             near_coords[0] = curr_coords[0] + 1;
400             near_coords[1] = curr_coords[1];
401
402             #pragma acc data present(w[:x_n + 2][y_n + 2], right_send_buf[:y_n])
```

```
403             {
404             for (j = 0; j < y_n; j++) {
405                 right_send_buf[j] = w[x_n][j + 1];
406             }
407             }
408 
409             # pragma acc data copyout(right_send_buf[:y_n])
410             {
411             MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
412             MPI_Isend(right_send_buf, y_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &requests[1]);
413             }
414         }
415 
416         // top send
417         if ((top_flag == 0) && (process_dims[1] > 1)) {
418             near_coords[0] = curr_coords[0];
419             near_coords[1] = curr_coords[1] + 1;
420 
421             #pragma acc data present(w[:x_n + 2][y_n + 2], top_send_buf[:x_n])
422             {
423             for (i = 0; i < x_n; i++) {
424                 top_send_buf[i] = w[i + 1][y_n];
425             }
426             }
427 
428             # pragma acc data copyout(top_send_buf[:x_n])
429             {
430             MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
431             MPI_Isend(top_send_buf, x_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &requests[2]);
432             }
433         }
434 
435         // bottom send
436         if ((bottom_flag == 0) && (process_dims[1] > 1)) {
437             near_coords[0] = curr_coords[0];
438             near_coords[1] = curr_coords[1] - 1;
439 
440             #pragma acc data present(w[:x_n + 2][y_n + 2], bottom_send_buf[:x_n])
441             {
442             for (i = 0; i < x_n; i++) {
443                 bottom_send_buf[i] = w[i + 1][1];
444             }
445             }
446 
447             # pragma acc data copyout(bottom_send_buf[:x_n])
448             {
449             MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
450             MPI_Isend(bottom_send_buf, x_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &requests[3]);
451             }
452         }
453 
454         // left recieve
455         if ((left_flag && (process_dims[0] > 1)) || (process_dims[0] == 1)) {
456 
457             #pragma acc data present(w[:x_n + 2][y_n + 2])
458             {
459             for (j = 0; j < y_n; j++) {
460                 w[0][j + 1] = u(x_min + (i_x - 1) * h1, y_min + (i_y + j) * h2);
461             }
462             }
463         }
464         else {
465             near_coords[0] = curr_coords[0] - 1;
466             near_coords[1] = curr_coords[1];
467             MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
468             MPI_Recv(left_recv_buf, y_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &status);
469 
470             #pragma acc data present(w[:x_n + 2][y_n + 2], left_send_buf[:y_n])
471             {
472             for (j = 0; j < y_n; j++) {
473                 w[0][j + 1] = left_recv_buf[j];
474             }
475             }
476         }
477 
478         // right recieve
479         if ((right_flag && (process_dims[0] > 1)) || (process_dims[0] == 1)) {
480 
481             #pragma acc data present(w[:x_n + 2][y_n + 2])
482             {
483             for (j = 0; j < y_n; j++) {
484                 w[x_n + 1][j + 1] = u(x_min + (i_x + x_n) * h1, y_min + (i_y + j) * h2);
485             }
486             }
487         }
488         else {
489             near_coords[0] = curr_coords[0] + 1;
490             near_coords[1] = curr_coords[1];
```

```
491             MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
492             MPI_Recv(right_recv_buf, y_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &status);
493
494             #pragma acc data present(w[:x_n + 2][y_n + 2], right_recv_buf[:y_n])
495             {
496             for (j = 0; j < y_n; j++) {
497                 w[x_n + 1][j + 1] = right_recv_buf[j];
498             }
499             }
500         }
501
502         // top recieve
503         if ((top_flag && (process_dims[1] > 1)) || (process_dims[1] == 1)) {
504
505             #pragma acc data present(w[:x_n + 2][y_n + 2])
506             {
507             for (i = 0; i < x_n; i++) {
508                 w[i + 1][y_n + 1] = u(x_min + (i_x + i) * h1, y_min + (i_y + y_n) * h2);
509             }
510             }
511         }
512         else {
513             near_coords[0] = curr_coords[0];
514             near_coords[1] = curr_coords[1] + 1;
515             MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
516             MPI_Recv(top_recv_buf, x_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &status);
517
518             #pragma acc data present(w[:x_n + 2][y_n + 2], top_recv_buf[:x_n])
519             {
520             for (i = 0; i < x_n; i++) {
521                 w[i + 1][y_n + 1] = top_recv_buf[i];
522             }
523             }
524         }
525
526         // bottom recieve
527         if ((bottom_flag && (process_dims[1] > 1)) || (process_dims[1] == 1)) {
528
529             #pragma acc data present(w[:x_n + 2][y_n + 2])
530             {
531             for (i = 0; i < x_n; i++) {
532                 w[i + 1][0] = u(x_min + (i_x + i) * h1, y_min + (i_y - 1) * h2);
533             }
534             }
535         }
536         else {
537             near_coords[0] = curr_coords[0];
538             near_coords[1] = curr_coords[1] - 1;
539             MPI_Cart_rank(MPI_COMM_CARTESIAN_TOPOLOGY, near_coords, &near_rank);
540             MPI_Recv(bottom_recv_buf, x_n, MPI_DOUBLE, near_rank, MPI_tag, MPI_COMM_CARTESIAN_TOPOLOGY, &status);
541
542             #pragma acc data present(w[:x_n + 2][y_n + 2], bottom_recv_buf[:x_n])
543             {
544             for (i = 0; i < x_n; i++) {
545                 w[i + 1][0] = bottom_recv_buf[i];
546             }
547             }
548         }
549
550         // MPI_Waitall(4, requests, MPI_STATUSES_IGNORE);
551
552         for (i = 0; i <= 3; i++) {
553            MPI_Wait(&requests[i], &status);
554         }
555     }
556
557     double max_abs_matr(int M, int N, double (*w)[N + 2]) {
558         double res = 0;
559
560         for (int i = 0; i <= M; i++) {
561             for (int j = 0; j <= N; j++) {
562                 double value = fabs(w[i][j]);
563                 if (value > res)
564                     res = value;
565             }
566         }
567         return res;
568     }
569
570     double mean_abs_matr(int M, int N, double (*w)[N + 2]) {
571         double res = 0;
572
573         for (int i = 0; i <= M; i++) {
574             for (int j = 0; j <= N; j++) {
575                 res += fabs(w[i][j]);
576
577             }
578         }
```

```
579         return res / (M * N);
580     }
581
582     int main(int argc, char* argv[]) {
583         int M = atoi(argv[argc - 2]);
584         int N = atoi(argv[argc - 1]);
585         // const int N = 10;
586         // const int M = 10;
587
588         double h1 = (A2 - A1) / M;
589         double h2 = (B2 - B1) / N;
590
591         printf("h1 = %3.8f h2 = %3.8f\n", h1, h2);
592
593         MPI_Init(&argc, &argv);
594
595         // MPI variables
596         int curr_rank, n_processes; int MPI_tag = 0;
597         int process_dims[2] = {0, 0};
598         int grid_is_periodic[2] = {0, 0};
599         int curr_coords[2];
600         int left_flag = 0, right_flag = 0, top_flag = 0, bottom_flag = 0;
601
602         MPI_Comm_rank(MPI_COMM_WORLD, &curr_rank);
603         MPI_Comm_size(MPI_COMM_WORLD, &n_processes);
604
605         MPI_Comm MPI_COMM_CARTESIAN_TOPOLOGY;
606         MPI_Dims_create(n_processes, 2, process_dims);
607         MPI_Cart_create(MPI_COMM_WORLD, 2, process_dims, grid_is_periodic, 1, &MPI_COMM_CARTESIAN_TOPOLOGY);
608         MPI_Cart_coords(MPI_COMM_CARTESIAN_TOPOLOGY, curr_rank, 2, curr_coords);
609
610         if (curr_coords[0] == 0) {
611             left_flag = 1;
612         }
613         if (curr_coords[1] == 0) {
614             bottom_flag = 1;
615         }
616         if ((curr_coords[0] + 1) == process_dims[0]) {
617             right_flag = 1;
618         }
619         if ((curr_coords[1] + 1) == process_dims[1]) {
620             top_flag = 1;
621         }
622
623         printf("rank %d left %d right %d top %d bottom %d \n", curr_rank, left_flag, right_flag, top_flag, bottom_flag);
624
625         int x_i, x_n, y_i, y_n;
626         if ((M + 1) % process_dims[0] == 0) {
627             x_n = (M + 1) / process_dims[0];
628             x_i = curr_coords[0] * x_n;
629         }
630         else {
631             if (curr_coords[0] == 0) {
632                 x_n = (M + 1) / process_dims[0] + (M + 1) % process_dims[0];
633                 x_i = 0;
634             }
635             else {
636                 x_n = (M + 1) / process_dims[0];
637                 x_i = curr_coords[0] * x_n + (M + 1) % process_dims[0];
638             }
639         }
640
641         if ((N + 1) % process_dims[1] == 0) {
642             y_n = (N + 1) / process_dims[1];
643             y_i = curr_coords[1] * y_n;
644         }
645         else {
646             if (curr_coords[1] == 0) {
647                 y_n = (N + 1) / process_dims[1] + (N + 1) % process_dims[1];
648                 y_i = 0;
649             }
650             else {
651                 y_n = (N + 1) / process_dims[1];
652                 y_i = curr_coords[1] * y_n + (N + 1) % process_dims[1];
653             }
654         }
655
656         printf("rank %d x_i %d y_i %d x_n %d y_n %d \n", curr_rank, x_i, y_i, x_n, y_n);
657
658         double *left_send_buf = malloc(sizeof(double[y_n])); double *right_send_buf = malloc(sizeof(double[y_n]));
659         double *left_recv_buf = malloc(sizeof(double[y_n])); double *right_recv_buf = malloc(sizeof(double[y_n]));
660         double *top_send_buf = malloc(sizeof(double[x_n])); double *bottom_send_buf = malloc(sizeof(double[x_n]));
661         double *top_recv_buf = malloc(sizeof(double[x_n])); double *bottom_recv_buf = malloc(sizeof(double[x_n]));
662
663         // algorithm variables
664         int converge = 0;
665         int n_iter = 0;
666         int i_max = x_n; int j_max = y_n;
```

```
667     double CRITERIA = 0; double criteria = 0; int MAX_ITER = 10000;
668     double TAU, TAU_DOT, TAU_NORM; double tau_dot, tau_norm;
669
670     double (*B)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
671     double (*w)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
672     double (*w_next)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
673     double (*Aw)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
674     double (*r)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
675     double (*Ar)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
676     double (*w_diff)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
677     double (*true_u)[y_n + 2] = malloc(sizeof(double[x_n + 2][y_n + 2]));
678
679     double start_timestamp = MPI_Wtime();
680
681     init_B(x_n, y_n, B, h1, h2, A1 + h1 * x_i, B1 + h2 * y_i);
682     init_B_fill_border(x_n, y_n, B, h1, h2, A1 + h1 * x_i, B1 + h2 * y_i, left_flag, right_flag, top_flag, bottom_flag);
683
684     #pragma omp parallel for
685     for (int i = 0; i <= x_n + 1; i++) {
686         for (int j = 0; j <= y_n; j++) {
687             w[i][j] = 0;
688         }
689     }
690
691     # pragma acc data copy(w[0:x_n + 2][0:y_n + 2], Aw[0:x_n + 2][0:y_n + 2], r[0:x_n + 2][0:y_n + 2], \
692     B[:x_n + 2][0:y_n + 2], Ar[0:x_n + 2][0:y_n + 2], \
693     left_send_buf[:y_n], right_send_buf[:y_n], left_recv_buf[:y_n], right_recv_buf[:y_n],\
694     top_send_buf[:x_n], bottom_send_buf[:x_n], top_recv_buf[:x_n], bottom_recv_buf[:x_n]), \
695     create(w_next[0:x_n + 2][0:y_n + 2], w_diff[0:x_n + 2][0:y_n + 2])
696     {
697     while (!converge && n_iter <= MAX_ITER) {
698
699         sync_borders(x_n, y_n, MPI_COMM_CARTESIAN_TOPOLOGY, left_flag, right_flag,
700                   top_flag, bottom_flag, process_dims, curr_coords,
701                   left_send_buf, right_send_buf, top_send_buf, bottom_send_buf,
702                   left_recv_buf, right_recv_buf, top_recv_buf, bottom_recv_buf,
703                   w, MPI_tag, A1, B1, h1, h2, x_i, y_i);
704
705         double x_min = A1 + h1 * x_i, y_min = B1 + h2 * y_i;
706         A_mul_vec(x_n, y_n, w, Aw, h1, h2, x_min, y_min);
707
708         A_mul_vec_fill_border(x_n, y_n, w, Aw, h1, h2, x_min, y_min,
709                               x_n, y_n, left_flag, right_flag, top_flag, bottom_flag);
710
711
712         # pragma acc parallel
713         {
714         # pragma acc loop independent
715         for (int i = 0; i <= x_n + 1; i++) {
716             r[i][0] = 0;
717             r[i][y_n + 1] = 0;
718         }
719
720         # pragma acc loop independent
721         for (int j = 0; j <= y_n + 1; j++) {
722             r[0][j] = 0;
723             r[x_n + 1][j] = 0;
724         }
725
726
727         # pragma acc loop independent
728         for (int i = 1; i <= x_n; i++) {
729             # pragma acc loop independent
730             for (int j = 1; j <= y_n; j++) {
731                 r[i][j] = Aw[i][j] - B[i][j];
732             }
733         }
734         }
735
736         sync_borders(x_n, y_n, MPI_COMM_CARTESIAN_TOPOLOGY, left_flag, right_flag,
737                   top_flag, bottom_flag, process_dims, curr_coords,
738                   left_send_buf, right_send_buf, top_send_buf, bottom_send_buf,
739                   left_recv_buf, right_recv_buf, top_recv_buf, bottom_recv_buf,
740                   r, MPI_tag, A1, B1, h1, h2, x_i, y_i);
741
742
743         A_mul_vec(x_n, y_n, r, Ar, h1, h2, x_min, y_min);
744         A_mul_vec_fill_border(x_n, y_n, r, Ar, h1, h2, x_min, y_min,
745                               x_n, y_n, left_flag, right_flag, top_flag, bottom_flag);
746
747
748         tau_norm = norm(x_n, y_n, Ar, h1, h2, left_flag, right_flag, top_flag, bottom_flag);
749         tau_dot = dot_product(x_n, y_n, Ar, r, h1, h2, left_flag, right_flag, top_flag, bottom_flag);
750
751         MPI_Allreduce(&tau_norm, &TAU_NORM, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_CARTESIAN_TOPOLOGY);
752         MPI_Allreduce(&tau_dot, &TAU_DOT, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_CARTESIAN_TOPOLOGY);
753         TAU = TAU_DOT / TAU_NORM;
754
```

```
755             # pragma acc parallel
756             {
757             # pragma acc loop independent
758             for (int i = 1; i <= x_n; i++) {
759                 # pragma acc loop independent
760                 for (int j = 1; j <= y_n; j++) {
761                     w_next[i][j] = w[i][j] - TAU * r[i][j];
762                     w_diff[i][j] = w_next[i][j] - w[i][j];
763                     w[i][j] = w_next[i][j];
764                 }
765             }
766             }
767 
768             criteria = sqrt(norm(x_n, y_n, w_diff, h1, h2, left_flag, right_flag, top_flag, bottom_flag));
769             MPI_Allreduce(&criteria, &CRITERIA, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_CARTESIAN_TOPOLOGY);
770 
771             converge = CRITERIA < EPSILON;
772 
773             n_iter++;
774             if (curr_rank == 0) {
775                 if (n_iter % 1000 == 0) {
776                     printf("iter␣%d␣criteria␣%3.8f␣dot␣%3.8f␣norm␣%3.8f␣tau␣%3.8f␣\n",
777                       n_iter, CRITERIA, TAU_DOT, TAU_NORM, TAU);
778                 }
779             }
780 
781         }
782         }
783 
784         double fin_timestamp = MPI_Wtime();
785         if (curr_rank == 0) {
786             printf("Over␣(rank␣%d,␣%d␣iters,␣time␣%f␣sec)\n", curr_rank, n_iter, fin_timestamp - start_timestamp);
787         }
788 
789         MPI_Barrier(MPI_COMM_WORLD);
790 
791         free(left_send_buf); free(left_recv_buf);
792         free(right_send_buf); free(right_recv_buf);
793         free(top_send_buf); free(top_recv_buf);
794         free(bottom_send_buf); free(bottom_recv_buf);
795         free(true_u); free(w_diff); free(Ar); free(r);
796         free(Aw); free(w_next); free(w); free(B);
797 
798         MPI_Finalize();
799         return 0;
800 
801     }
```