

Отчет по решению задачи Дирихле для уравнения Пуассона

Вариант 7: Область-сапожок

Автор: Пудовиков А.Д.

Группа: 61бмс

Реализация: [git](#)

1. Постановка задачи

1.1 Математическая формулировка

Требуется решить двумерную задачу Дирихле для уравнения Пуассона:

$$-\Delta u = f(x, y), \quad (x, y) \in D,$$

$$u(x, y) = 0, \quad (x, y) \in \gamma$$

где:

- Оператор Лапласа:

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

- Правая часть:

$$f(x, y) = 1 \quad \forall (x, y) \in D$$

- Область D (сапожок):

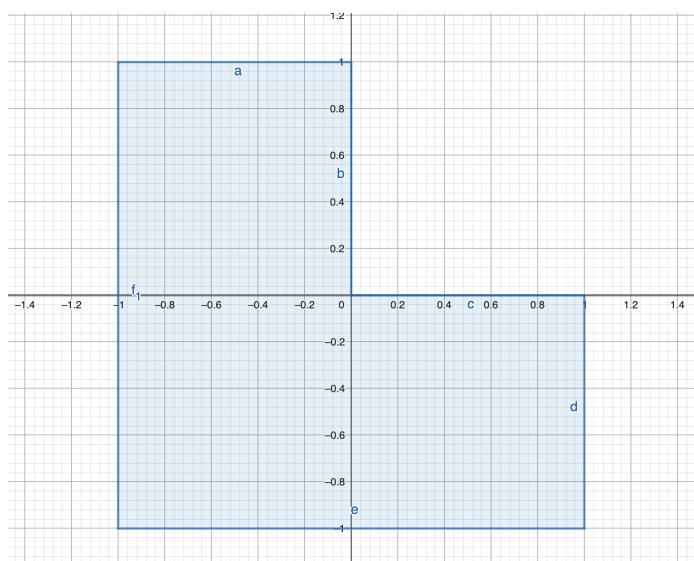
$$D = \{(x, y) : -1 < x, y < 1\} \setminus \{(x, y) : 0 \leq x, y < 1\}$$

Это квадрат $[-1, 1] \times [-1, 1]$ с вырезанным правым верхним квадрантом $[0, 1] \times [0, 1]$.

- Граница γ :

$$\gamma = \partial D$$

1.2 Геометрия области



2. Метод решения

2.1 Метод фиктивных областей

Криволинейная область D заменяется прямоугольником:

$$\Pi = \{(x, y) : -1 < x < 1, -1 < y < 1\}$$

с введением кусочно-постоянного коэффициента:

$$k(x, y) = \begin{cases} 1, & (x, y) \in D \\ 1/\varepsilon, & (x, y) \in \hat{D} \end{cases}$$

где $\hat{D} = \Pi \setminus D$ — фиктивная область, $\varepsilon = h^2$ — параметр метода (h — максимальный шаг сетки).

Решается вспомогательная задача:

$$-\frac{\partial}{\partial x} \left(k(x, y) \frac{\partial v}{\partial x} \right) - \frac{\partial}{\partial y} \left(k(x, y) \frac{\partial v}{\partial y} \right) = F(x, y)$$

$$v(x, y) = 0, \quad (x, y) \in \Gamma$$

где:

$$F(x, y) = \begin{cases} f(x, y) = 1, & (x, y) \in D \\ 0, & (x, y) \in \hat{D} \end{cases}$$

Теорема (приближение):

$$\max_{(x,y) \in D} |v(x, y) - u(x, y)| < C\varepsilon$$

2.2 Разностная схема

На равномерной прямоугольной сетке $\bar{\omega}_h = \bar{\omega}_1 \times \bar{\omega}_2$:

$$\begin{aligned} \bar{\omega}_1 &= \{x_i = -1 + ih_1, i = 0, \overline{M}\}, & h_1 &= \frac{2}{M} \\ \bar{\omega}_2 &= \{y_j = -1 + jh_2, j = 0, \overline{N}\}, & h_2 &= \frac{2}{N} \end{aligned}$$

Дифференциальное уравнение аппроксимируется разностным:

$$-\frac{1}{h_1} \left(a_{i+1,j} \frac{w_{i+1,j} - w_{i,j}}{h_1} - a_{i,j} \frac{w_{i,j} - w_{i-1,j}}{h_1} \right) - \frac{1}{h_2} \left(b_{i,j+1} \frac{w_{i,j+1} - w_{i,j}}{h_2} - b_{i,j} \frac{w_{i,j} - w_{i,j-1}}{h_2} \right) = F_{i,j}$$

для $i = 1, \overline{M-1}, j = 1, \overline{N-1}$.

Коэффициенты:

$$a_{i,j} = \frac{1}{h_2} \int_{y_{j-1/2}}^{y_{j+1/2}} k(x_{i-1/2}, t) dt$$

$$b_{i,j} = \frac{1}{h_1} \int_{x_{i-1/2}}^{x_{i+1/2}} k(t, y_{j-1/2}) dt$$

Правая часть:

$$F_{i,j} = \frac{1}{h_1 h_2} \iint_{\Pi_{i,j}} F(x, y) dx dy$$

где $\Pi_{i,j}$ — ячейка сетки вокруг узла (x_i, y_j) .

Границные условия:

$$w_{i,j} = 0, \quad (x_i, y_j) \in \Gamma$$

2.3 Метод сопряженных градиентов с предобуславливанием

Разностная схема приводится к системе линейных алгебраических уравнений:

$$Aw = B$$

где $A : H \rightarrow H$ — самосопряженный положительно определенный оператор.

Скалярное произведение и норма:

$$(u, v) = \sum_{i=1}^{M-1} \sum_{j=1}^{N-1} h_1 h_2 u_{i,j} v_{i,j}$$

$$\|u\|_E = \sqrt{(u, u)}$$

Диагональный предобуславливатель:

$$(Dw)_{i,j} = \left[\frac{a_{i+1,j} + a_{i,j}}{h_1^2} + \frac{b_{i,j+1} + b_{i,j}}{h_2^2} \right] w_{i,j}$$

Алгоритм:

1. Начальное приближение: $w^{(0)} = 0$
2. Невязка: $r^{(0)} = B - Aw^{(0)} = B$
3. Решение: $Dz^{(0)} = r^{(0)}$
4. Направление спуска: $p^{(1)} = z^{(0)}$

Для $k = 0, 1, 2, \dots$ до сходимости:

5. $Ap^{(k+1)}$ — применение оператора
6. $\alpha_{k+1} = \frac{(z^{(k)}, r^{(k)})}{(Ap^{(k+1)}, p^{(k+1)})}$
7. $w^{(k+1)} = w^{(k)} + \alpha_{k+1} p^{(k+1)}$
8. **Проверка сходимости:** если $\|w^{(k+1)} - w^{(k)}\|_E < \delta$, то STOP
9. $r^{(k+1)} = r^{(k)} - \alpha_{k+1} Ap^{(k+1)}$
10. $Dz^{(k+1)} = r^{(k+1)}$
11. $\beta_{k+1} = \frac{(z^{(k+1)}, r^{(k+1)})}{(z^{(k)}, r^{(k)})}$
12. $p^{(k+1)} = z^{(k+1)} + \beta_{k+1} p^{(k)}$

Контроль монотонности:

Функционал $H(w^{(k)}) = (B + r^{(k)}, w^{(k)})$ должен монотонно возрастать. При нарушении монотонности — перезапуск метода.

3. Структура кодовой базы

```

src/
├── poisson_sequential.cpp      # Последовательная версия
├── poisson_omp.cpp            # OpenMP версия
├── poisson_mpi.cpp           # MPI версия
├── poisson_mpi_omp.cpp       # Гибридная MPI+OpenMP версия
├── poisson_solver_mpi.h      # MPI solver с основной логикой, в т.ч. Grid
├── poisson_solver_mpi_omp.h  # MPI + OpenMP solver с основной логикой
├── domain_decomposition.h    # Функции декомпозиции домена для выбора сетки
└── plot_solution.py          # Визуализация решения
                             # Визуализация производительности

```

4. Результаты запусков последовательной программы

4.1 Численные результаты для последовательного метода

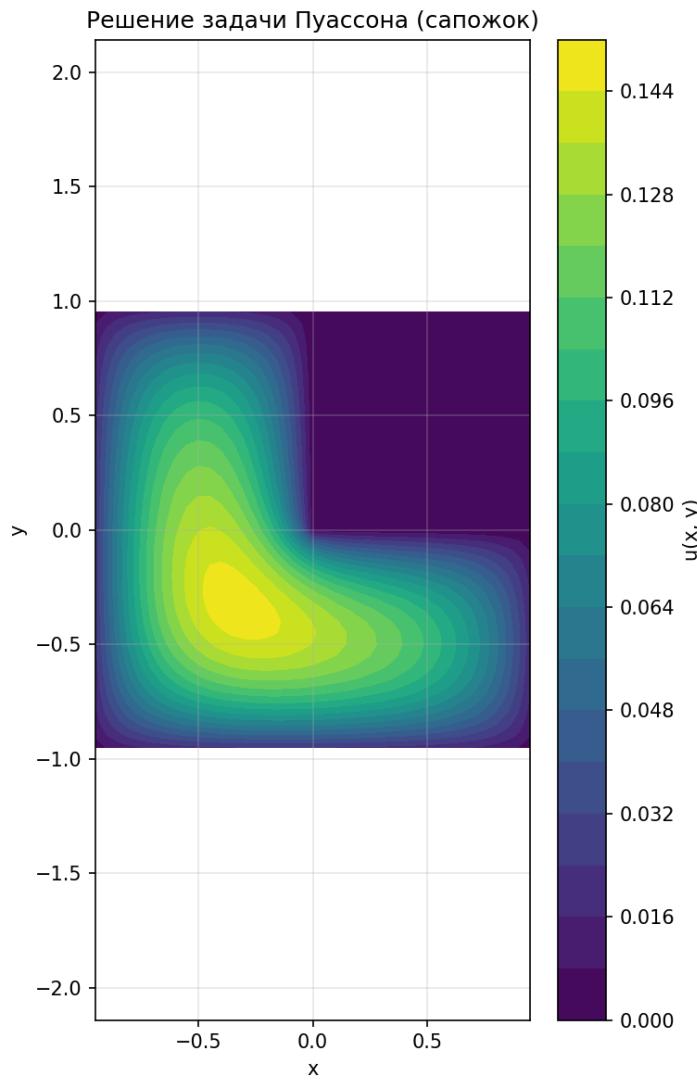
Сетка $M \times N$	Узлов	Итераций	Время (с)	$\ w\ _E$	$\ w\ _C$
10 × 10	81	21	0.000127	1.497e-01	1.512e-01
20 × 20	361	40	0.000935	1.450e-01	1.492e-01
40 × 40	1521	73	0.006733	1.441e-01	1.491e-01

Где $\|w\|_E$ - евклидова норма решения, $\|w\|_C$ - максимум норма решения.

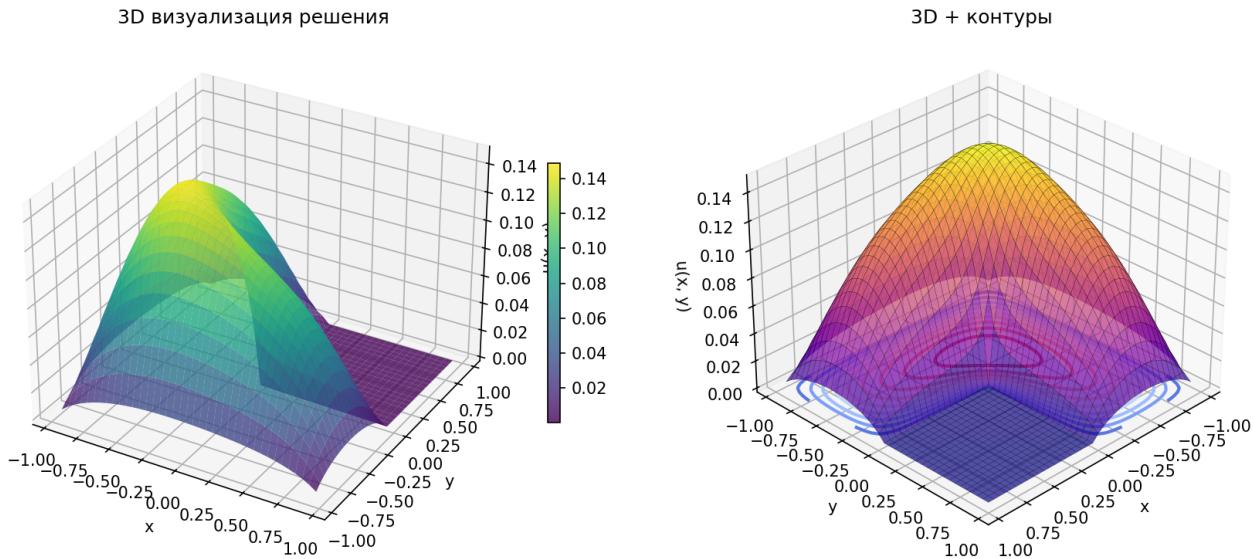
Результаты ожидаемые - узмельчение сетки привело к увеличению количества итераций и стабилизации нормы решения.

Визуализация решения

2D визуализация (цветная карта и изолинии):



3D визуализация (поверхность с контурами):



Из визуализации видно:

- Решение положительно во всей области ($f = 1 > 0$)

- Максимум значения $u \approx 0.149$ достигается внутри области
- Значения плавно убывают к границе (граничное условие $u = 0$)
- Решение симметрично относительно диагонали $y = x$

4.2 Свойства решения

1. **Положительность:** $u(x, y) > 0$ для всех $(x, y) \in D$ (так как $f = 1 > 0$)
2. **Максимум:** достигается внутри области (принцип максимума для эллиптических уравнений)
3. **Симметрия:** решение примерно симметрично относительно диагонали $y = x$ (из-за геометрии области)
4. **Границные значения:** $u \rightarrow 0$ при приближении к границе

5. Результаты запусков OpenMP программы

Основная модификация OpenMP версии от классической последовательной заключалась в использовании директив перед циклами.

Так как в программе в основном использовались двойные циклы для вычисления, приведу пример для них:

```
#pragma omp parallel for reduction(+:var) schedule(static) collapse(2)
```

Директива `for`, клауда `reduction(+:var)` - для оператора + сохраняем значения в `var`,
`schedule(static)` - деление цикла на блоки равной длины, `collapse(2)` - сворачивание двойных циклов.

5.1 Сравнение запусков на сетке 40×40 OpenMP

Тип Запуска	Итераций	Время (с)	$ w _E$	$ w _C$
40 × 40 последовательный	73	0.006733	1.441e-01	1.491e-01
40 × 40 1 нить	73	0.0094	1.440562e-01	1.490863e-01
40 × 40 4 нити	73	0.003975	1.440562e-01	1.490863e-01
40 × 40 16 нитей	73	0.003129	1.440562e-01	1.490863e-01

Видим, что на такой небольшой решётке время запусков OpenMP решения больше для 1 нити и меньше для большего числа нитей.

Это может быть связано с тем, что был добавлен "лишний" код OpenMP, который нам надо было исполнять, но он не использовался для ускорения программы.

При этом нормы решения сопоставимые, а значит нет оснований предполагать, что есть ошибка.

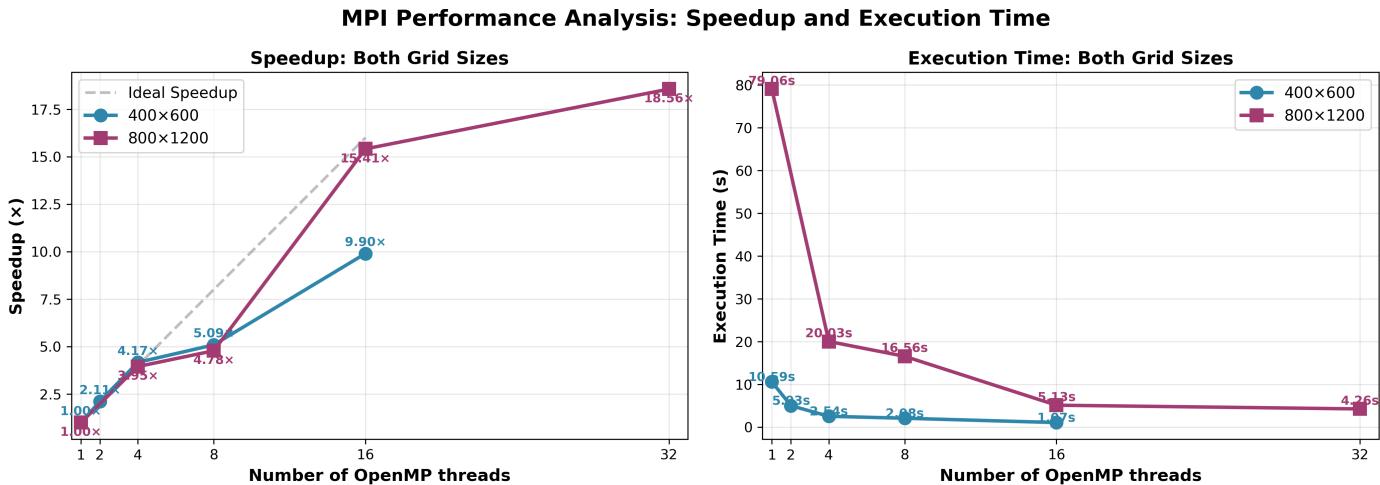
На параллельных запусках - наблюдаем тенденцию уменьшения времени работы программы.

5.2 Запуски на больших сетках и графики ускорения

Таблица 1: OpenMP версии

Кол-во нитей	Число точек сетки (M×N)	Число итераций	Время, с	Ускорение, раз
1	400×600	533	10.59	-
2	400×600	533	5.03	2.11
4	400×600	533	2.54	4.17
8	400×600	533	2.08	5.09
16	400×600	533	1.07	9.90
1	800×1200	1048	79.06	-
4	800×1200	1048	20.03	3.95
8	800×1200	1048	16.56	4.78
16	800×1200	1048	5.13	15.41
32	800×1200	1048	4.26	18.56

Графики ускорения и времени выполнения



Из графиков видно:

- Ускорение растет с увеличением числа нитей, достигая 18.56× на 32 нитях для сетки 800×1200
- Для сетки 800×1200 эффективность выше благодаря большему объему вычислений

6. Результаты запусков MPI программы

Для создания MPI версии программы было необходимо разработать алгоритм двумерного разбиения исходного прямоугольника, так чтобы:

1. Отношение количества узлов по переменным x и y в каждом домене принадлежало диапазону $[1/2, 2]$.
Это требование выполняется с помощью выбора "самого квадратного" разбиения (у которого отношение сторон ближе всего к 1).
Перебор происходит только по тем разбиениям, которые удовлетворяют заявленному условию.
2. Количество узлов по переменным x и y любых двух доменов отличалось не более, чем на единицу. Это требование автоматически выполняется при использовании блочного распределения узлов.
Если всего N_x внутренних узлов по x распределяются на P_x процессов, то каждый процесс получает либо $\lfloor N_x / P_x \rfloor$, либо $\lceil N_x / P_x \rceil$ узлов, то есть разница в один узел.

Далее надо было реализовать обмен граничными значениями на решётке для процессов, для этого используется функция `exchange()` с `MPI_Sendrecv()`.

Для нахождения соседей по решётке использовал `MPI_Cart_shift()`.

Для проверки условий остановки и вычисления норм используются операции `MPI_Allreduce()`, чтобы учесть значения со всех доменов.

6.1 Сравнение запусков на сетке 40×40 MPI

Тип Запуска	Итераций	Время (с)	$ w _E$	$ w _C$
40×40 последовательный	73	0.006733	1.441e-01	1.491e-01
40×40 1 процесс	80	0.005571	1.327e-01	1.412e-01
40×40 2 процесса	80	0.005007	1.327e-01	1.412e-01
40×40 4 процесса	80	0.003197	1.327e-01	1.412e-01

Как и в случае OpenMP не получается наблюдать ускорение для использования MPI программы, так как накладные расходы значительно выше, чем потенциальный выигрыш.

Для версий MPI наблюдаем уменьшение времени работы, что говорит об эффективности нашего решения.

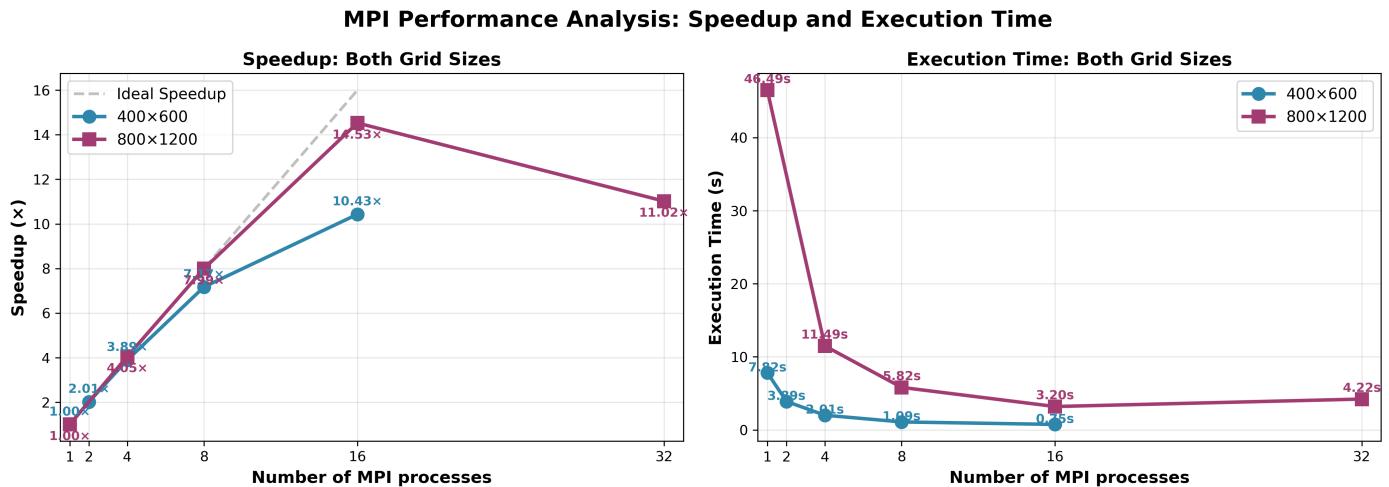
При этом нормы решения сопоставимые, а значит нет оснований предполагать, что есть ошибка.

6.2 Запуски на больших сетках и графики ускорения

Таблица 2: MPI версии

Кол-во процессов MPI	Число точек сетки (M×N)	Число итераций	Время, с	Ускорение, раз
1	400×600	730	7.82	-
2	400×600	730	3.89	2.01
4	400×600	730	2.01	3.89
8	400×600	730	1.09	7.17
16	400×600	730	0.75	10.43
1	800×1200	1048	46.49	-
4	800×1200	1048	11.49	4.05
8	800×1200	1048	5.82	7.99
16	800×1200	1048	3.20	14.53
32	800×1200	1048	4.22	11.02

Графики ускорения и времени выполнения



Из графиков видно:

- Ускорение растет с увеличением числа процессов, достигая 14.53 раз на 16 процессах для сетки 800×1200
- Видно, что для сетки 800×1200 ускорение не монотонно на 32 процессах, это может быть связано с накладными расходами, которые возникают на большем количестве процессов.
- Для сетки 800×1200 эффективность выше благодаря большему объему вычислений
- Ускорение во всех случаях оказывается меньше линейного

7. Результаты запусков MPI+OpenMP программы.

Основная задача - разработать гибридный алгоритм программы для запусков совмещённой версии MPI и OpenMP. Для этого в MPI код были добавлены OpenMP директивы по аналогии с чисто OpenMP версией.

7.1 Сравнение запусков на сетке 40 × 40 MPI

Тип Запуска	Итераций	Время (с)	$ w _E$	$ w _C$
40 × 40 последовательный	73	0.006733	1.441e-01	1.491e-01
40 × 40 1 MPI процесс + 4 нити	80	0.003406	1.327e-01	1.412e-01
40 × 40 2 MPI процесс + 4 нити	80	0.003304	1.327e-01	1.412e-01

Ожидаемо, что нормы решения совпали с MPI версией и последовательной.

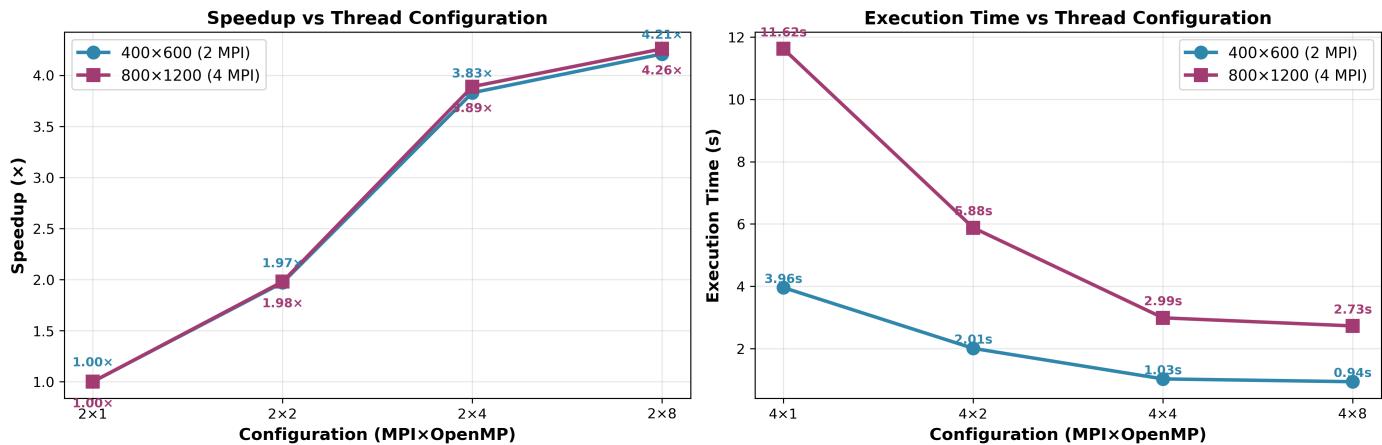
По времени исполнения программы наблюдаем ускорение, использовалась привязка нитей к ядрам.

7.2 Запуски на больших сетках и графики ускорения

Таблица 3: MPI + OpenMP версия

Кол-во процессов MPI	Кол-во нитей OpenMp	Число точек сетки (M×N)	Число итераций	Время, с	Ускорение, раз
2	1	400×600	730	3.96	-
2	2	400×600	730	2.01	1.97
2	4	400×600	730	1.03	3.83
2	8	400×600	730	0.94	4.21
4	1	800×1200	1065	11.62	-
4	2	800×1200	1065	5.88	1.98
4	4	800×1200	1065	2.99	3.89
4	8	800×1200	1065	2.73	4.26

MPI+OpenMP Hybrid Performance Analysis: Speedup and Execution Time



Из графиков видно:

- Ускорение растет с увеличением числа процессов, достигая 4.26 раз на 4 MPI 8 OpenMP для сетки 800×1200
- Видно, что для обеих сеток ускорение монотонно
- Для сетки 800×1200 эффективность незначительно выше благодаря большему объему вычислений
- Ускорение во всех случаях оказывается меньше линейного по числу нитей

7. Результаты запусков MPI+CUDA программы.

Основная сложность заключалась в организации редукции для решения задачи, так как по условию задачи нам запрещено использовать разделяемую память.

Поэтому были использована глобальная память. Для ускорения процесса копирований был использовано обмен только граничными значениями, что позволило опередить остальные решения

Основная идея решения заключается в использовании CUDA ядер и соответствующих функций для их запуска.

На GPU считаем самые большие функции (применение операторов A , D^{-1} , частичный подсчёт для скалярных произведений, норм).

После этого копируем граничные значения на CPU, делаю обмен через MPI.

Обновляем границы с помощью копирования обратно на GPU

Технические вопросы:

- Перешёл на сетку 2000×3200 , чтобы лучше увидеть преимущества параллельных версий, время последовательной программы уже более 16 минут
- Сетка 2000×3200 требует поддержки обязательных массивов для $a, b, F, D_{diag}, w, r, z, p, Ap$ размеров около всей сетки. Т.к. double занимает 8 байт, то всего около $2000 * 3200 * 8 * 9 = 439\text{MB}$
- Расширил поддержку Makefile для CUDA, использовать с компилятором HOST_COMP=mpicxx.
- В результате перебора на 160 потоков для гибридной версии MPI + OpenMP выбор пал на комбинацию 10 MPI \times 16

threads.

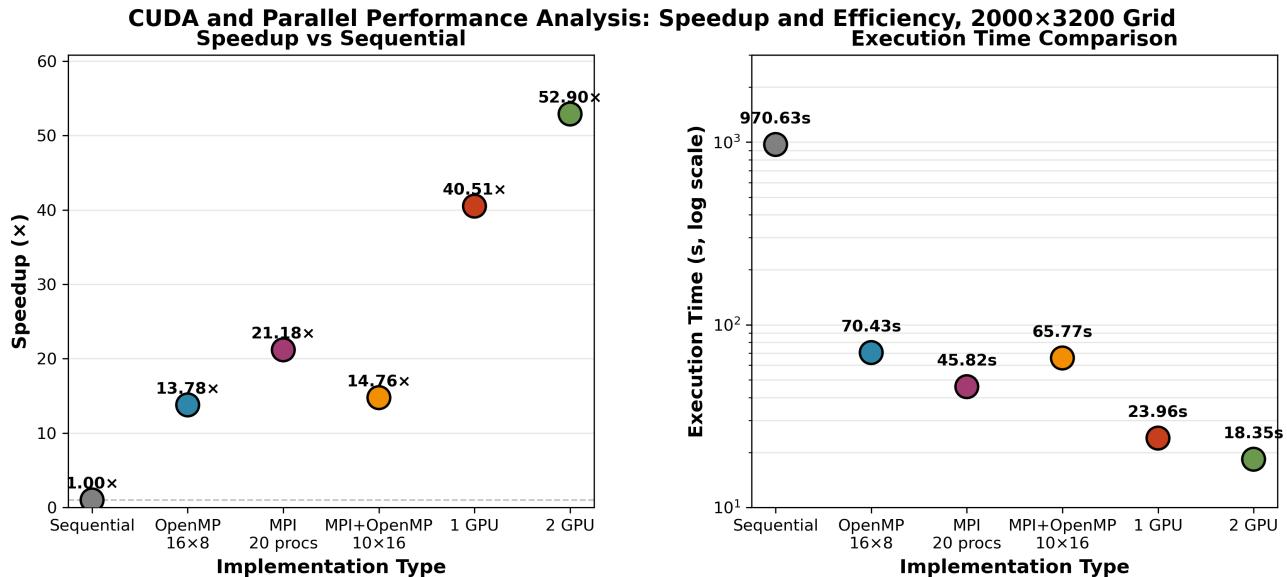
- Так как нельзя пользоваться shared memory, то пользовался глобальной памятью, редукцией на CPU.
- В решение использовал самый простой вариант - с блокирующими операциями (`MPI_Sendrecv()`, `cudaMemcpy()`). В таблицы приведены соответствующие времена.

7.1 Сравнение работы программы на большой сетке.

Для того, что увидеть ускорение в программе с GPU была использована увеличенная до 2000×3200 сетка.

Тип Запуска	Итераций	Время общее (с)	$\ w\ _E$	$\ w\ _C$	GPU init	apply_A	apply_D_inv	Обновление векторов	GPU->CPU копирования	CPU->GPU копирования	MPI обмен	MPI allreduce	CPU reduce	Coeff. Init	OpenMP reduce
2000 × 3200, последовательный	2618	970.629623	1.440e	1.495e-01	-	369.12	64.07	56.85	-	-	-	-	75.34	0.58	-
2000 × 3200, 16 × 8 threads OpenMP	2618	70.43	1.327e-01	1.412e-01	-	45.20	9.91	8.62	-	-	-	-	0.09	6.64	
2000 × 3200, 20 MPI	2641	45.82	1.438e-01	1.438e-01	-	23.25	3.48	9.90	-	-	0.127	4.40	4.65	0.04	-
2000 × 3200, MPI + OpenMP	2641	65.77	1.438e-01	1.494e-01	-	15.56	3.46	9.71	-	-	2.260	25.64	9.12	0.037	-
2000 × 3200, 1 GPU	2641	23.96	1.438e-01	1.494e-01	0.014	2.84	2.21	18.59	0.043	0.012	-	-	1.71	-	
2000 × 3200, 2 GPU	2641	18.35	1.438e-01	1.494e-01	0.009	1.36	0.97	14.64	0.286	0.114	0.26	0.014	-	0.91	-

Так как нормы решений совпадают и алгоритмы сходятся, то есть все основания считать, что предложенные реализации алгоритмов работают.



Запуск на 2ух GPU оказался лучше, чем на 1ом.

Сетка для решения уже достаточно большая (последовательная версия посчиталась за 16 минут на Polus).

Оценка правдоподобности времени последовательных вычислений - для сетки 40×40 имеем 73 итерации за 0.006733 с, тогда

$$0.006733 * 2600 / 73 * 2000 / 40 * 3200 / 40 = 959.22,$$

что сопоставимо с экспериментальным временем.

При этом видим, что ускорение GPU операций всего 1.5 раза при увеличении количества GPU в 2 раза.

Самая долгая операция - это обновление вспомогательных векторов, которое использует много блокирующих операций, что приводит к задержке.

Копирования с GPU --> CPU и GPU --> CPU присутствуют только для случая нескольких GPU (как доп требование к заданию).

Для реализации оптимизированной логики для одного GPU были добавлены соответствующие ядра.

Помимо этого `dot_product_gpu_ptr` с редукцией перешла на GPU, что дало итоговый прирост и снижение количества ненужных копирований.

8. Выводы.

В рамках предоставленного задания мне удалось численно решить двумерную задачу Дирихле для уравнения Пуассона на языке C++.

Для ускорения вычислений на мелких сетках я использовал ускорение через

- OpenMP нити,
- MPI процессы,
- MPI процессы и OpenMP нити
- MPI процесса и поддержка CUDA

Все перечисленные методы показали необходимость в их использовании для решения больших задач.

Для запуска решений использовался удалённый сервер Polus с планировщиком IBM Spectrum LSF.