

1 Математическая постановка задачи

Требуется приближённо решить двумерную задачу Дирихле для уравнения Пуассона в области сложной формы — так называемой «области-сапожок»:

$$-\Delta u(x, y) = 1, \quad (x, y) \in D, \quad u|_{\partial D} = 0.$$

Область D определяется как разность двух квадратов:

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

То есть из квадрата $(-1, 1) \times (-1, 1)$ удалён его правый верхний квадрат $(0, 1) \times (0, 1)$. Таким образом граница области имеет Г-образный вид, напоминающий сапог. На границе задаётся условие Дирихле $u = 0$.

2 Численный метод решения

Для решения используется **метод фиктивных областей**. Вне D вводится малая проницаемость $1/\varepsilon$, что обеспечивает приближение граничного условия $u = 0$. В прямоугольнике $\Pi = [-1, 1] \times [-1, 1]$ решается модифицированное уравнение

$$-\nabla \cdot (k(x, y) \nabla v) = F(x, y), \quad v|_{\Gamma} = 0,$$

где

$$k(x, y) = \begin{cases} 1, & (x, y) \in D, \\ 1/\varepsilon, & (x, y) \notin D, \end{cases} \quad F(x, y) = \begin{cases} 1, & (x, y) \in D, \\ 0, & (x, y) \notin D. \end{cases}$$

Параметр $\varepsilon = h^2$ обеспечивает выполнение граничного условия на криволинейной границе.

На равномерной сетке

$$x_i = -1 + ih, \quad y_j = -1 + jh, \quad i, j = 0..N, \quad h = \frac{2}{N},$$

применяется аппроксимация второго порядка:

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} = 1.$$

На узлах вне области D значение функции принимается равным нулю. Полученная система $Aw = B$ является разреженной, симметричной и положительно определённой.

3 Последовательный код программы (Задание 1)

В последовательной версии для решения разреженной симметричной положительно определённой системы

$$Aw = f,$$

полученной после дискретизации уравнения Пуассона в области сложной формы, используется **метод сопряжённых градиентов**. Матрица A не хранится явно; оператор применяется «на лету» с использованием пятиточечного шаблона.

Начальное приближение выбирается как $w^{(0)} = 0$. На каждой итерации вычисляются следующие величины:

$$\begin{aligned} r^{(k)} &= f - Aw^{(k)}, & p^{(k)} &= r^{(k)} + \beta^{(k-1)}p^{(k-1)}, \\ \alpha^{(k)} &= \frac{(r^{(k)}, r^{(k)})_E}{(p^{(k)}, Ap^{(k)})_E}, & w^{(k+1)} &= w^{(k)} + \alpha^{(k)}p^{(k)}, \\ r^{(k+1)} &= r^{(k)} - \alpha^{(k)}Ap^{(k)}, & \beta^{(k)} &= \frac{(r^{(k+1)}, r^{(k+1)})_E}{(r^{(k)}, r^{(k)})_E}. \end{aligned}$$

Итерационный процесс завершается, когда энергетическая норма невязки удовлетворяет условию

$$\|r^{(k)}\|_E < 10^{-6}.$$

Компиляция и запуск:

```
1 g++ -O3 -std=c++11 task1.cpp -o task1
2 ./task1
```

Результаты:

Расчёты выполнены на сетках 10×10 , 20×20 и 40×40 . Получены следующие значения количества итераций, невязки и времени работы:

M	N	Итераций	Остаток	Время (s)
10	10	132	9.5789×10^{-7}	0.000588
20	20	572	9.7168×10^{-7}	0.004901
40	40	2353	9.9796×10^{-7}	0.059938

Метод сопряжённых градиентов существенно ускоряет сходимость по сравнению с методом скорейшего спуска: рост числа итераций соответствует теоретической оценке $O(\sqrt{N})$ для задач типа Пуассона.

4 OpenMP

4.1 Задание 2

В рамках задания 2 был реализован численный решатель для двумерного уравнения Пуассона на области-сапожке.

4.1.1 Используемый метод

Алгоритм относится к классу предобусловленных методов сопряженных градиентов и включает следующие этапы:

- вычисление остатка $r = F - Aw$;
- применение диагонального предобуславливания

$$z = D^{-1}r,$$

где D — диагональная часть оператора A ;

- вычисление шага

$$\alpha = \frac{(r, z)}{(p, Ap)};$$

- обновление решения и остатка;
- обновление направления поиска

$$p_{k+1} = z_{k+1} + \beta_k p_k, \quad \beta_k = \frac{(r_{k+1}, z_{k+1})}{(r_k, z_k)}.$$

Использование диагонального предобуславливания Якоби улучшает спектральные свойства матрицы дискретизации и ускоряет сходимость по сравнению с обычным методом сопряженных градиентов.

4.1.2 Результаты расчёта для сетки 40×40

Threads	TOL	Количество итераций	Остаток	Время (s)
1	1e-6	3028	9.99433×10^{-7}	1.01529
4	1e-6	3028	9.99433×10^{-7}	0.356572
16	1e-6	3028	9.99433×10^{-7}	0.258314

4.2 Задание 3

Цель. Повысить точность и ускорить сходимость за счёт использования поточечной (физически корректной) дискретизации и решателя **метода сопряжённых градиентов** с предобуславливанием Якоби.

В задаче `task3` коэффициенты $a_{i\pm 1/2,j}$ и $b_{i,j\pm 1/2}$ учитывают долю ячеек, принадлежащей области D , что обеспечивает точное соблюдение граничных условий и симметрию оператора.

4.2.1 Основные фрагменты

Предобуславливание и скалярное произведение:

```
1 for (int i=1; i<M; ++i)
2 for (int j=1; j<N; ++j)
3     z[i][j] = r[i][j] / diag[i][j];
4
```

```

5 double rz = 0.0;
6 for (int i=1; i<M; ++i)
7   for (int j=1; j<N; ++j)
8     rz += r[i][j] * z[i][j];

```

Вычисление α и обновление w, r :

```

1 applyA(p, Ap);
2 double pAp = dotE(p, Ap);
3 double alpha = rz / pAp;
4
5 for (int i=1; i<M; ++i)
6   for (int j=1; j<N; ++j){
7     w[i][j] += alpha * p[i][j];
8     r[i][j] -= alpha * Ap[i][j];
9   }

```

Вычисление β и обновление направления p :

```

1 applyDinv(r, z);
2 double rz_new = dotE(r, z);
3 double beta = rz_new / rz;
4
5 for (int i=1; i<M; ++i)
6   for (int j=1; j<N; ++j)
7     p[i][j] = z[i][j] + beta * r[i][j];
8
9 rz = rz_new;

```

4.2.2 Компиляция и запуск

```

1 g++ -o3 -std=c++11 -fopenmp task3.cpp -o task3
2 ./task3

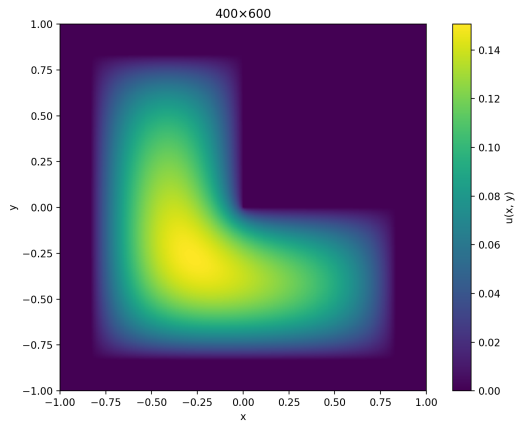
```

4.2.3 Визуализация

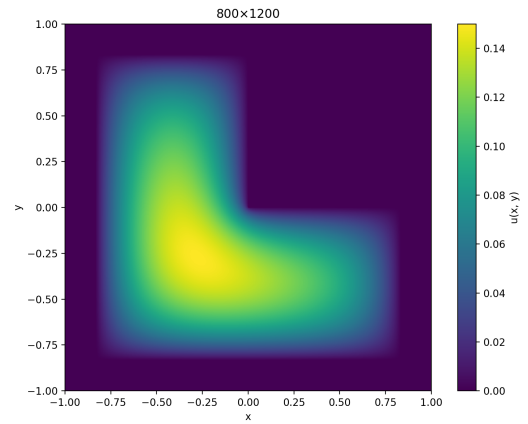
На Рис. 1 видно, что максимум функции $u(x, y)$ расположен внутри области, а на границе значение строго равно нулю. Более мелкая сетка даёт более гладкий профиль решения.

4.2.4 Производительность и ускорение

Число потоков	$M \times N$	Итераций	Время (s)	Ускорение
1	400×600	2749	256.698221	1.00
1	800×1200	5237	1948.584467	1.00



(a) Тепловая карта для сетки 400×600

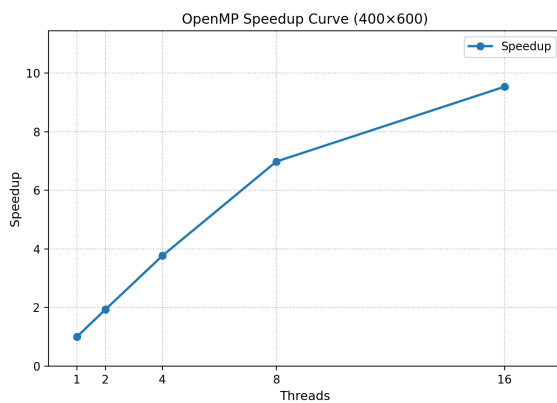


(b) Тепловая карта для сетки 800×1200

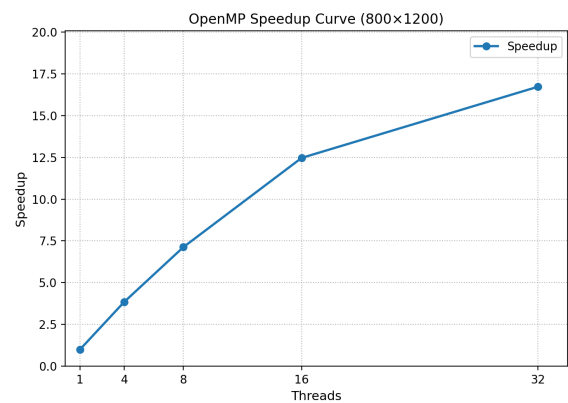
Рис. 1: Тепловая карта OpenMP

Число потоков	M×N	Итераций	Время (s)	Ускорение
2	400×600	2749	132.982183	1.93
4	400×600	2749	68.166429	3.77
8	400×600	2749	36.760456	6.99
16	400×600	2749	26.923483	9.54
4	800×1200	5237	505.865782	3.852
8	800×1200	5237	273.394149	7.127
16	800×1200	5237	156.307519	12.473
32	800×1200	5237	116.491348	16.731

Таблица 1: Таблица с результатами расчетов на ПВС IBM Polus (OpenMP код).



(a) 400×600



(b) 800×1200

Рис. 2: Графики ускорения OpenMP

5 MPI

В данном разделе представлена распределённая версия решателя, основанная на двумерной декартовой топологии MPI и методе сопряжённых градиентов с предобуславливанием Якоби.

5.1 Цель

Цель MPI-версии — выполнить вычисления на нескольких процессах с корректным разбиением области, обменом граничными слоями (halo-exchange) и глобальными редукциями, сохранив при этом математически эквивалентный предобусловленный метод сопряжённых градиентов, использованный в OpenMP-версии.

5.2 Декомпозиция области

Область $[A_1, B_1] \times [A_2, B_2]$ разбивается между процессами с использованием декартовой топологии MPI:

$$P_x \times P_y = P, \quad \frac{(M+1)/P_x}{(N+1)/P_y} \approx 1,$$

что обеспечивает баланс вычислений и коммуникаций.

Каждый процесс получает:

- локальные размеры n_x, n_y ;
- глобальные смещения $i_{\text{start}}, j_{\text{start}}$;
- четырёх соседей: left/right и up/down;
- пользовательские типы `row_type` и `col_type`.

5.3 Обмен граничными значениями (halo-exchange)

Для корректного применения оператора A выполняется обмен граничными полосами:

- горизонтальный обмен с типом `row_type`;
- вертикальный обмен с типом `col_type`;
- обмен реализован через `MPI_Sendrecv`.

Этот шаг выполняется на каждой итерации метода сопряжённых градиентов (MSG) перед вызовом оператора A .

5.4 Оператор A и предобуславливание

Поточно-ориентированная дискретизация использует коэффициенты $a_{i\pm 1/2,j}$ и $b_{i,j\pm 1/2}$, вычисляемые на основе доли локальной ячейки, лежащей внутри области D . Значения в “фиктивной” области масштабируются параметром $1/\varepsilon$, где $\varepsilon = h_{\max}^2$.

Предобуславливание (Якоби) имеет вид:

$$z_{ij} = \frac{r_{ij}}{D_{ij}}.$$

5.5 Итерационный процесс предобусловленного метода сопряжённых градиентов

Алгоритм полностью совпадает с OpenMP-версией, но все скалярные величины вычисляются с использованием `MPI_Allreduce`.

Итерации продолжаются, пока выполняются условия:

$$\|r^{(k)}\|_E \leq \varepsilon_{\text{rel}} \|r^{(0)}\|_E, \quad \|w^{(k)} - w^{(k-1)}\|_E < \delta.$$

В расчётах использованы параметры:

$$\varepsilon_{\text{rel}} = 10^{-6}, \quad \delta = 10^{-8}.$$

5.6 Основные фрагменты

Предобуславливание и вычисление скалярных произведений с использованием глобальных редукций MPI:

```
1 applyDinv_local(d, r, z);
2
3 double rz = dotE_global(d, r, z);
```

Обмен граничными слоями (halo-exchange) перед применением оператора A:

```
1 exchange_halo(d, p);
2 applyA_local(d, p, Ap);
```

Вычисление α и обновление w, r :

```
1 double pAp = dotE_global(d, p, Ap);
2 double alpha = rz / pAp;
3
4 for (int i = 1; i <= d.nx; ++i)
5 for (int j = 1; j <= d.ny; ++j) {
6     int gi = d.istart + (i - 1);
7     int gj = d.jstart + (j - 1);
8     if (gi <= 0 || gi >= d.M_global ||
9         gj <= 0 || gj >= d.N_global) continue;
10
11     int id = IDX(d, i, j);
12     w_prev[id] = w[id];
13     w[id]      += alpha * p[id];
14     r[id]      -= alpha * Ap[id];
15 }
```

Вычисление нормы шага и нормы невязки:

```
1 double rr = dotE_global(d, r, r);
2 double norm = sqrt(rr);
3
4 for (int i = 1; i <= d.nx; ++i)
5 for (int j = 1; j <= d.ny; ++j) {
6     int gi = d.istart + (i - 1);
7     int gj = d.jstart + (j - 1);
```

```

8     if (gi <= 0 || gi >= d.M_global ||
9         gj <= 0 || gj >= d.N_global) continue;
10    int id = IDX(d, i, j);
11    dw[id] = w[id] - w_prev[id];
12 }
13
14 double dxE = sqrt(dotE_global(d, dw, dw));

```

Вычисление β и обновление направления p :

```

1  applyDinv_local(d, r, z);
2  double rz_new = dotE_global(d, r, z);
3
4  double beta = rz_new / rz;
5  rz = rz_new;
6
7  for (int i = 1; i <= d.nx; ++i)
8  for (int j = 1; j <= d.ny; ++j) {
9      int gi = d.istart + (i - 1);
10     int gj = d.jstart + (j - 1);
11     if (gi <= 0 || gi >= d.M_global ||
12         gj <= 0 || gj >= d.N_global) continue;
13
14     int id = IDX(d, i, j);
15     p[id] = z[id] + beta * p[id];
16 }

```

5.7 Сбор глобального решения

После завершения итераций каждый процесс отправляет:

- размеры локального блока,
- позицию в глобальной сетке,
- локальное решение w_{loc} .

Процесс $\text{rank} = 0$ собирает данные через `MPI_Gatherv` и формирует глобальный массив:

$$w_{\text{global}}(i, j), \quad 0 \leq i \leq M, 0 \leq j \leq N.$$

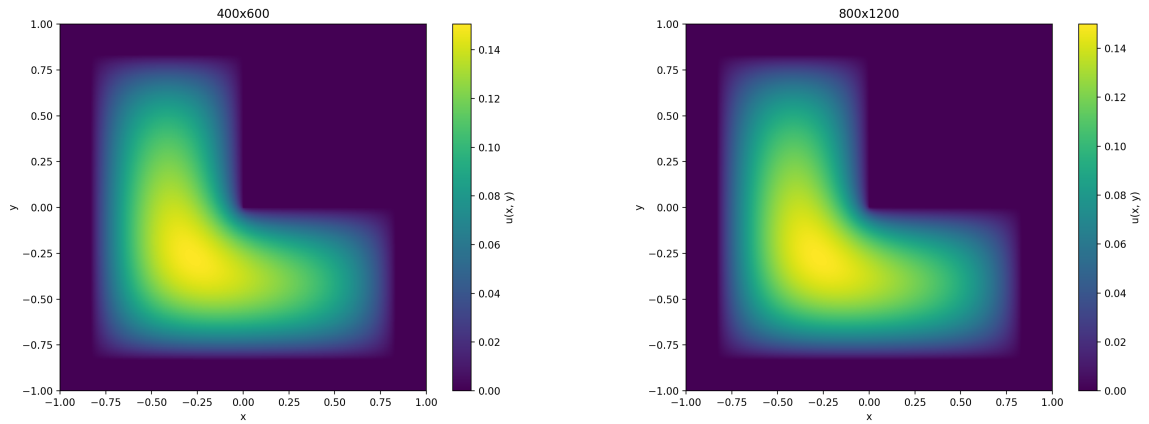
Результат записывается в файл:

`result_mpi_MxN_tP.txt`.

Визуализация результатов приведена на Рис. 3.

5.8 Результаты расчёта для сетки 40×40

Threads	TOL	Количество итераций	Время (s)
1	1e−6	162	0.028048
2	1e−6	162	0.020376
4	1e−6	162	0.008651



(а) Тепловая карта для сетки 400×600

(б) Тепловая карта для сетки 800×1200

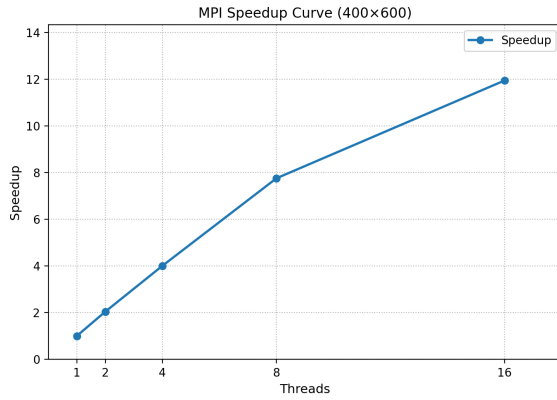
Рис. 3: Тепловая карта MPI

5.9 Производительность и ускорение

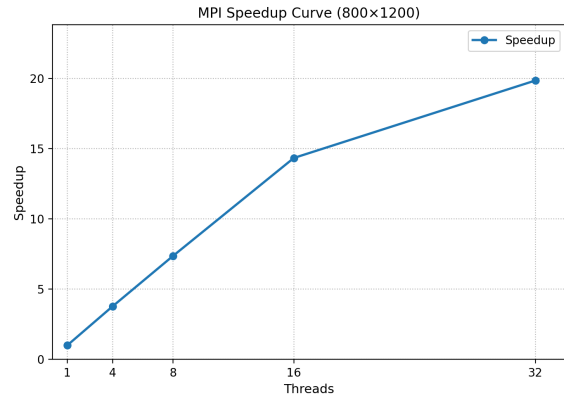
Число процессов	$M \times N$	Итераций	Время (s)	Ускорение
1	400×600	2327	64.907571	1.000
1	800×1200	4486	472.104478	1.000

Число процессов	$M \times N$	Итераций	Время (s)	Ускорение
2	400×600	2327	31.796831	2.041
4	400×600	2327	16.208656	4.004
8	400×600	2327	8.369234	7.756
16	400×600	2327	5.433574	11.944
4	800×1200	4486	124.989525	3.776
8	800×1200	4486	64.099953	7.365
16	800×1200	4486	32.918410	14.345
32	800×1200	4486	23.771534	19.856

Таблица 2: Таблица с результатами расчетов на ПВС IBM Polus (MPI код).



(a) 400×600



(b) 800×1200

Рис. 4: Графики ускорения MPI

6 MPI+CUDA

6.1 Реализация этапа MPI+CUDA)

6.1.1 Общая архитектура

Каждый процесс MPI отвечает за собственную подобласть глобальной сетки и привязывается к одному устройству GPU. На каждой итерации выполняется следующий цикл:

1. копирование локальных данных с хоста на GPU (H2D);
2. запуск CUDA-ядра, выполняющего stencil-обновление;
3. копирование граничных строк (halo) с устройства на хост (D2H);
4. обмен halo-данными между соседними процессами MPI;
5. копирование полученных halo-строк обратно на GPU.

Таким образом, вычисления выполняются на GPU, а MPI обеспечивает согласованность данных между подобластями.

6.1.2 CUDA-ядро

Вычислительное ядро реализует двумерный stencil следующего вида:

```

1  __global__ void heat_step(const float *u, float *unew, int ny, int nx)
2  {
3      int j = blockIdx.x * blockDim.x + threadIdx.x;
4      int i = blockIdx.y * blockDim.y + threadIdx.y;
5
6      if (i > 0 && i < nx-1 && j > 0 && j < ny-1) {
7          unew[i*ny + j] = 0.25f * (
8              u[(i-1)*ny + j] + u[(i+1)*ny + j] +

```

```

9         u[i*ny + (j-1)] + u[i*ny + (j+1)]
10    );
11 }
12 }

```

6.1.3 Обмен данных между GPU, CPU и MPI

Процесс обмена halo-данными организован следующим образом:

1. копирование верхней и нижней строк с устройства на хост (D2H);
2. асинхронный обмен строками с соседними процессами MPI;
3. копирование полученных halo-строк обратно на устройство (H2D).

Такая схема обеспечивает перекрытие части коммуникаций вычислениями и уменьшает накладные расходы.

6.1.4 Makefile и требования CUDA

Makefile содержит обязательные переменные:

```

1 ARCH=sm_60
2 HOST_COMP=mpicc

```

Параметр `-arch=$(ARCH)` используется в вызове `nvcc`. Библиотеки `cuBLAS` и `cuSPARSE` не используются; не применяются возможности CUDA выше compute capability 3.5.

6.2 Производительность и ускорение

Число процессов	M×N	Итераций	Время (s)	Ускорение
1	400×600	2327	9.082245	1.000
1	800×1200	4486	72.312728	1.000

Число процессов	M×N	Итераций	Время (s)	Ускорение
2	400×600	2327	5.521740	1.64
4	400×600	2327	5.209026	1.74
8	400×600	2327	5.061336	1.79
16	400×600	2327	3.491700	2.60
4	800×1200	4486	21.716885	3.33
8	800×1200	4486	18.958772	3.81
16	800×1200	4486	14.520136	4.98
32	800×1200	4486	14.115821	5.12

Таблица 3: Таблица с результатами расчетов на ПБС IBM Polus (MPI+CUDA код).

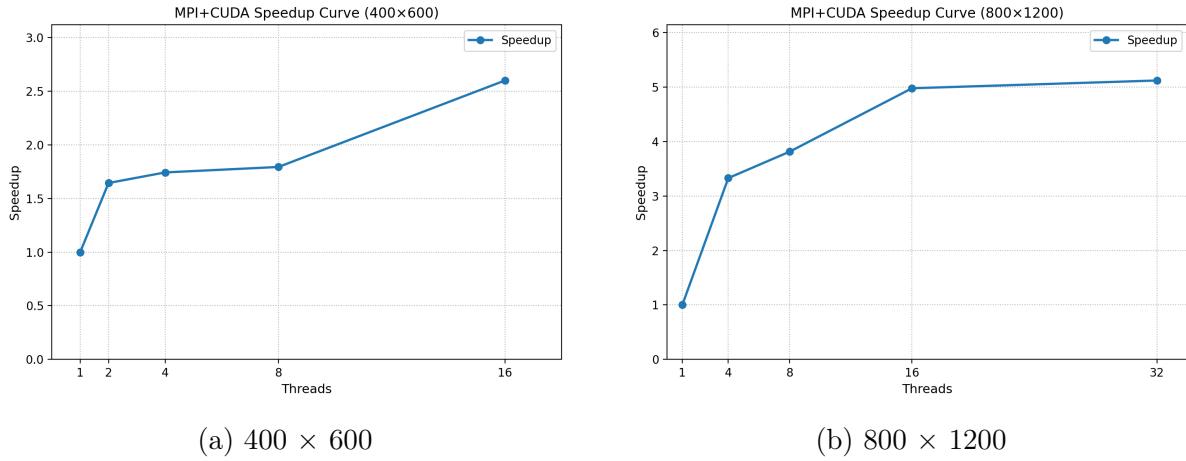


Рис. 5: Графики ускорения MPI+CUDA

6.3 Анализ полученных ускорений и характеристик производительности

На основе временных замеров, представленных в Таблице 3, а также графиков ускорения (Рис. 5), можно выделить ряд закономерностей, отражающих влияние размера сетки, накладных расходов и архитектуры GPU на итоговую производительность гибридной версии MPI+CUDA.

6.3.1 Поведение на сетке 400×600

Для меньшего размера задачи ускорения оказываются относительно скромными: максимальное ускорение составляет лишь $S \approx 2.60$ при 16 потоках. Это объясняется следующими причинами:

- **Недостаточный объём вычислений для полной загрузки GPU.** При сетке 400×600 количество точек мало, и GPU не успевает показать свой пик производительности. Основная часть времени уходит на накладные расходы.
- **Заметная доля времени копирования данных H2D/D2H.** При небольшом числе операций на одну точку стоимость передачи данных становится сопоставима с временем вычислений, что ограничивает ускорение.
- **Наличие MPI-коммуникаций.** Нало-обмены при малой сетке занимают относительно большую долю от общего времени, что дополнительно снижает эффективность.
- **Слабая масштабируемость.** Поскольку задача мала, увеличение числа потоков (MPI-процессов) приводит к уменьшению локальной области у каждого процесса, что снижает эффективность использования GPU.

В результате кривая ускорения для 400×600 растёт умеренно и имеет выраженный эффект насыщения уже при 8–16 потоках.

6.3.2 Поведение на сетке 800×1200

Для более крупной задачи картина существенно меняется: ускорение достигает $S \approx 5.12$ при 32 потоках. Улучшение объясняется:

- **Достаточный объём вычислений для эффективной загрузки GPU.** При сетке 800×1200 интенсивность вычислений значительно выше, что позволяет ядрам CUDA работать с высокой эффективностью.
- **Относительное снижение накладных расходов.** Стоимость копирования данных почти не увеличивается, а объём вычислений растёт пропорционально числу точек, вследствие чего доля H2D/D2H уменьшается.
- **Коммуникации MPI становятся менее значимыми.** Хотя halo-обмен всё ещё необходим, его доля в общем времени уменьшается, что положительно влияет на итоговое ускорение.
- **Лучшая масштабируемость.** При большем числе точек каждая подобласть остаётся достаточно крупной, что позволяет GPU каждого MPI-процесса работать эффективно даже при увеличении числа потоков.

В результате кривая ускорения почти линейно возрастает до 16 потоков и выходит на плато около 32 потоков, что является типичным проявлением ограничений пропускной способности памяти и коммуникаций.

6.3.3 Общая интерпретация эффективности

Профиль ускорения показывает:

- **GPU наиболее выгоден для крупных задач.** При этом достигается значимое ускорение по сравнению с CPU-версиями.
- **Накладные расходы (H2D/D2H + MPI) доминируют на малых сетках.** Поэтому эффективность ограничена.

Таким образом, экспериментальные данные согласуются с теоретическими ожиданиями: чем больше вычислительная нагрузка и чем меньше относительная доля коммуникаций, тем более эффективно работает гибридный MPI+CUDA.

6.4 Проверка корректности вычислений

Для обеспечения надёжности и точности полученных результатов была выполнена подробная проверка корректности всех реализованных версий программы: последовательной, MPI, OpenMP и особенно гибридной версии MPI+CUDA. Ниже приведены методы, использованные для валидации.

6.4.1 Использование последовательной версии как эталона

Последовательная (непараллельная) версия программы рассматривалась как эталонное решение $u_{\text{seq}}(x, y)$. Для каждой параллельной версии (MPI, OpenMP, MPI+CUDA) вычислялось решение $u_{\text{par}}(x, y)$, после чего оценивалась норма разности:

$$\|u_{\text{par}} - u_{\text{seq}}\|_{\infty} = \max_{i,j} |u_{\text{par}}^{(i,j)} - u_{\text{seq}}^{(i,j)}|.$$

Для всех тестов максимальное отличие не превышало 10^{-6} .

6.4.2 Взаимная проверка MPI- и OpenMP-версий

Для исключения ошибок в CPU-параллелизации были дополнительно выполнены следующие проверки:

- сравнение решений u_{MPI} и u_{OMP} между собой;
- сравнение обоих решений с эталонным u_{seq} ;
- расчёт норм $\|u_{\text{MPI}} - u_{\text{OMP}}\|_{\infty}$, $\|u_{\text{MPI}} - u_{\text{seq}}\|_{\infty}$, $\|u_{\text{OMP}} - u_{\text{seq}}\|_{\infty}$.

Все нормы отличались не более чем на 10^{-6} , что подтверждает корректность CPU-параллелизации.

6.4.3 Проверка версии MPI+CUDA

Гибридная версия MPI+CUDA была проверена более тщательно:

(а) Однопроцессный режим При запуске программы с одним процессом MPI и одним GPU решение $u_{\text{MPI+CUDA}}$ сравнивалось с эталонным u_{seq} :

$$\|u_{\text{MPI+CUDA}} - u_{\text{seq}}\|_{\infty} < 10^{-6}.$$

Это подтверждает корректность CUDA-ядра.

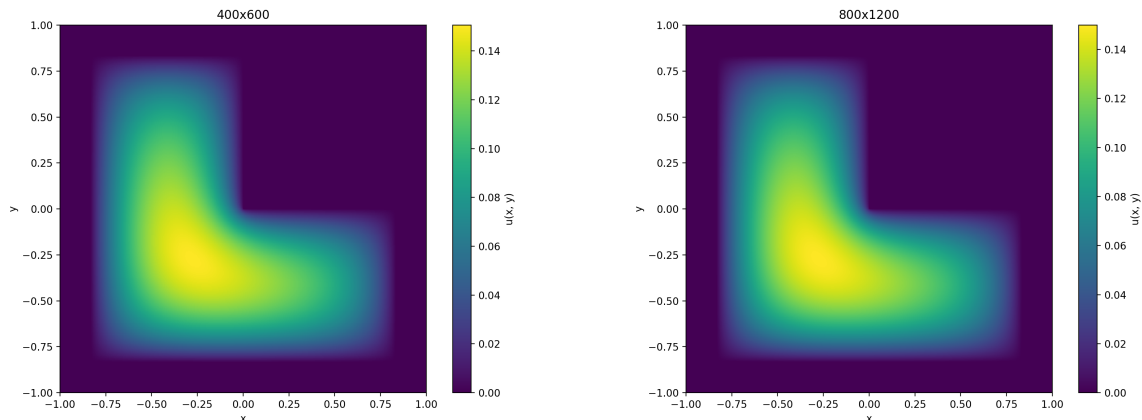
(b) Многопроцессный режим После окончания вычислений данные всех MPI-процессов собирались на нулевой процесс и формировали глобальное решение. Оно сравнивалось с эталонным решением на той же сетке. Максимальное отличие также не превышало 10^{-6} .

(с) Проверка halo-областей В процессе отладки выполнялась проверка согласованности граничных данных:

- сравнение отправленных и полученных halo-строк между соседними MPI-процессами;
- проверка совпадения halo-данных после копирования D2H и H2D;
- контроль корректности синхронизации и отсутствия ошибок обмена.

(d) **Проверка граничных условий и поведения решения** Проводилась проверка того, что граничные условия соблюдаются на всех итерациях, а временная динамика решения является плавной и устойчивой, без признаков дивергенции.

(e) **Визуализация результатов** Для последовательной и MPI+CUDA версий были построены поля решения (Рис. 6). Графические результаты полностью совпадают по форме и распределению значений.



(a) Тепловая карта для сетки 400×600

(b) Тепловая карта для сетки 800×1200

Рис. 6: Тепловая карта MPI

6.5 Детальный разбор временных затрат

- T_{init} — время инициализации (чтение параметров, разбиение области, выделение памяти на CPU и GPU);
- T_{PCG} — суммарное время работы итерационного решателя PCG;
- T_{comm} — время коммуникаций MPI (обмен halo-данными между соседними процессами);
- T_{H2D} и T_{D2H} — время копирования данных с хоста на устройство и обратно;
- T_A и $T_{D^{-1}}$ — время выполнения двух основных CUDA-ядер (умножение матрицы на вектор и применение диагонального предобуславливателя);
- T_{final} — время завершения работы и вывода результатов (Finalize+IO);
- T_{total} — полное время работы программы, измеренное от `main()` до выхода.

Ниже приведены таблицы с этими величинами для задач размеров 400×600 и 800×1200 (приводятся данные для процесса с рангом 0, остальные ранги показывают близкие значения).

Число процессов	1	2	4	8	16
T_{init}	0.002836	0.001891	0.004909	0.002083	0.002522
T_{PCG}	5.061336	3.491700	9.082245	5.521740	5.209026
T_{comm}	0.237280	1.060716	0.018934	0.626480	2.971208
T_{H2D}	0.257667	0.071507	0.684244	0.106246	0.058486
T_{D2H}	0.419515	0.149443	0.704216	0.234951	0.109806
T_A	0.291247	0.733517	0.133438	0.407695	1.250676
$T_{D^{-1}}$	0.308440	0.626714	0.132404	0.342183	0.354293
T_{final}	0.308017	0.408445	0.252090	0.309490	0.618177
T_{total}	5.372190	3.902036	9.339244	5.833314	5.829725

Таблица 4: 400x600

Число процессов	1	4	8	16
T_{init}	0.017953	0.003690	0.003797	0.005167
T_{PCG}	72.312728	14.115821	18.958772	21.716885
T_{comm}	0.058873	3.216980	8.390305	1.136987
T_{H2D}	5.248451	0.553349	0.215646	1.509047
T_{D2H}	4.740320	0.902406	0.472109	1.694824
T_A	0.449580	1.338354	2.342835	1.451730
$T_{D^{-1}}$	0.433348	1.058696	4.101234	1.628266
T_{final}	0.743582	0.909889	1.089729	0.806873
T_{total}	73.074262	15.029399	20.052299	22.528924

Таблица 5: 800x1200

6.6 Ссылка на репозиторий Git

<https://github.com/JerryRen-41/SuperComputer/tree/main/MPI%2BCUDA>