Отчёт по курсу «Суперкомпьютерное моделирование и технологии» для студентов первого и второго потоков, вариант 7, Жэнь Цзежуй 618-1

1 Математическая постановка задачи

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

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

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

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

где

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

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

На равномерной сетке

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

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 $A\,w = B$ 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:

$$r^{(k)} = f - Aw^{(k)}, \qquad p^{(k)} = r^{(k)} + \beta^{(k-1)} p^{(k-1)},$$

$$\alpha^{(k)} = \frac{(r^{(k)}, r^{(k)})_E}{(p^{(k)}, Ap^{(k)})_E}, \qquad w^{(k+1)} = w^{(k)} + \alpha^{(k)} p^{(k)},$$

$$r^{(k+1)} = r^{(k)} - \alpha^{(k)} Ap^{(k)}, \qquad \beta^{(k)} = \frac{(r^{(k+1)}, r^{(k+1)})_E}{(r^{(k)}, r^{(k)})_E}.$$

Итерационный процесс завершается, когда энергетическая норма невязки удовлетворяет условию

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

Компиляция и запуск:

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

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

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 потоковой (физически корректной) дискретизации и решателя **метода сопряжённых градиентов** с предобуславливанием Якоби.

В задаче `task3` коэффициенты $a_{i\pm1/2,j}$ и $b_{i,j\pm1/2}$ учитывают долю ячеек, принадлежащей области D, что обеспечивает точное соблюдение граничных условий и симметрию оператора.

4.2.1 Основные фрагменты

Предобуславливание и скалярное произведение:

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

```

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

Вычисление α и обновление w, r:

```
applyA(p, Ap);
double pAp = dotE(p, Ap);
double alpha = rz / pAp;

for (int i=1; i<M; ++i)
for (int j=1; j<N; ++j){
    w[i][j] += alpha * p[i][j];
    r[i][j] -= alpha * Ap[i][j];
}
```

Вычисление β и обновление направления p:

```
applyDinv(r, z);
double rz_new = dotE(r, z);
double beta = rz_new / rz;

for (int i=1; i<M; ++i)
for (int j=1; j<N; ++j)
    p[i][j] = z[i][j] + beta * p[i][j];

rz = rz_new;
```

4.2.2 Компиляция и запуск

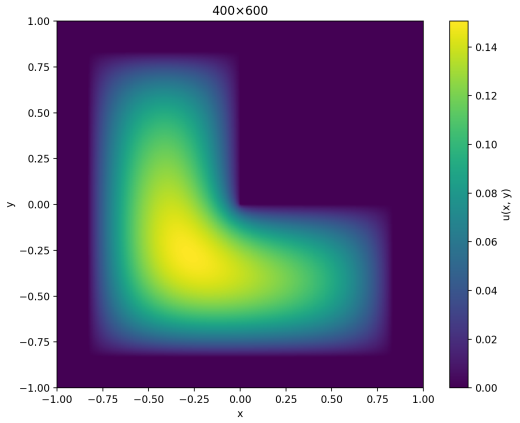
```
g++ -o3 -std=c++11 -fopenmp task3.cpp -o task3
./task3
```

4.2.3 Визуализация

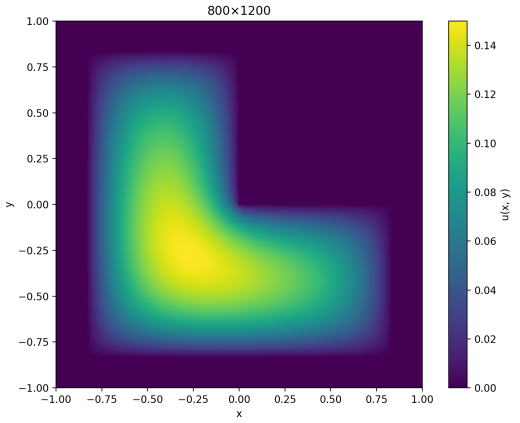
На Рис. 1 видно, что максимум функции $u(x, y)$ расположен внутри области, а на границе значение строго равно нулю. Более мелкая сетка даёт более гладкий профиль решения.

4.2.4 Производительность и ускорение

Число потоков	M×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

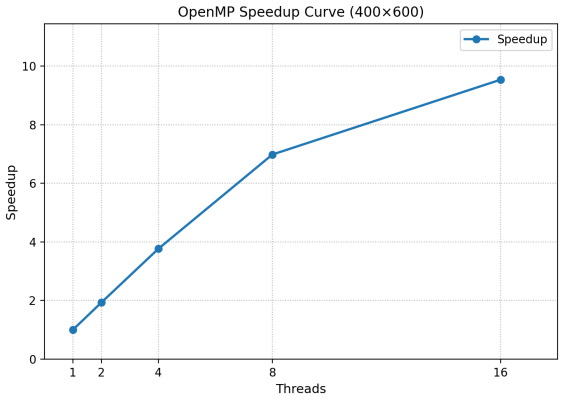


(b) Тепловая карта для сетки 800×1200

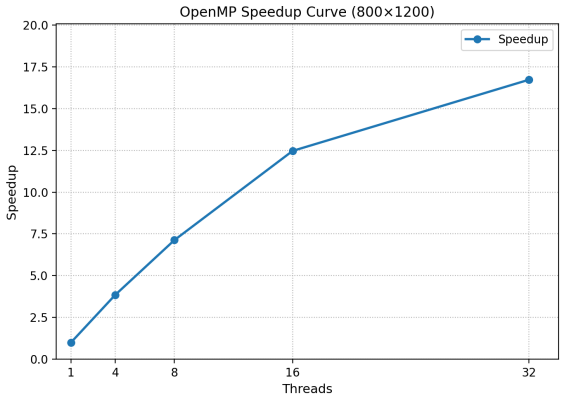
Рис. 1: Тепловая карта OpenMP

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:

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

что обеспечивает баланс вычислений и коммуникаций.

Каждый процесс получает:

- локальные размеры n_x, n_y;
- глобальные смещения i_{start}, j_{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 (МСГ) перед вызовом оператора A.

5.4 Оператор A и предобуславливание

Поточно-ориентированная дискретизация использует коэффициенты $a_{i\pm1/2,j}$ и $b_{i,j\pm1/2}$, вычисляемые на основе доли локальной ячейки, лежащей внутри области D. Значения в “фиктивной” области масштабируются параметром $1/\varepsilon$, где $\varepsilon = h^2_{\max}$.

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:

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

В расчётах использованы параметры:

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

5.6 Основные фрагменты

Предобуславливание и вычисление скалярных произведений с использованием глобальных редукций MPI:

```
applyDinv_local(d, r, z);

double rz = dotE_global(d, r, z);
```

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

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

Вычисление α и обновление w, r:

```
double pAp = dotE_global(d, p, Ap);
double alpha = rz / pAp;

for (int i = 1; i <= d.nx; ++i)
for (int j = 1; j <= d.ny; ++j) {
    int gi = d.istart + (i - 1);
    int gj = d.jstart + (j - 1);
    if (gi <= 0 || gi >= d.M_global ||
        gj <= 0 || gj >= d.N_global) continue;

    int id = IDX(d, i, j);
    w_prev[id] = w[id];
    w[id]      += alpha * p[id];
    r[id]      -= alpha * Ap[id];
}
```

Вычисление нормы шага и нормы невязки:

```
double rr  = dotE_global(d, r, r);
double norm = sqrt(rr);

for (int i = 1; i <= d.nx; ++i)
for (int j = 1; j <= d.ny; ++j) {
    int gi = d.istart + (i - 1);
    int gj = d.jstart + (j - 1);
```

```
    if (gi <= 0 || gi >= d.M_global ||
        gj <= 0 || gj >= d.N_global) continue;
    int id = IDX(d, i, j);
    dw[id] = w[id] - w_prev[id];
}

double dxE = sqrt(dotE_global(d, dw, dw));
```

Вычисление β и обновление направления p:

```
applyDinv_local(d, r, z);
double rz_new = dotE_global(d, r, z);

double beta = rz_new / rz;
rz = rz_new;

for (int i = 1; i <= d.nx; ++i)
for (int j = 1; j <= d.ny; ++j) {
    int gi = d.istart + (i - 1);
    int gj = d.jstart + (j - 1);
    if (gi <= 0 || gi >= d.M_global ||
        gj <= 0 || gj >= d.N_global) continue;

    int id = IDX(d, i, j);
    p[id] = z[id] + beta * p[id];
}
```

5.7 Сбор глобального решения

После завершения итераций каждый процесс отправляет:

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

Процесс `rank = 0` собирает данные через `MPI_Gatherv` и формирует глобальный массив:

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

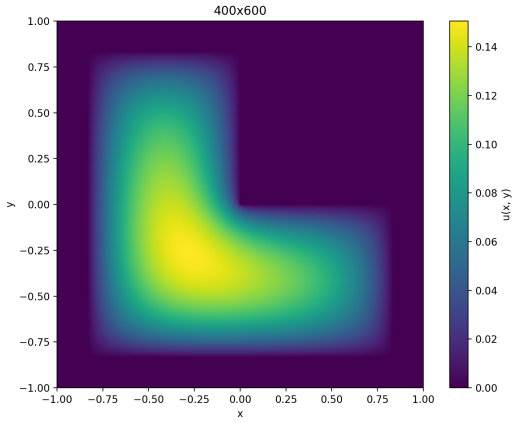
Результат записывается в файл:

`result_mpi_MxN_tP.txt`.

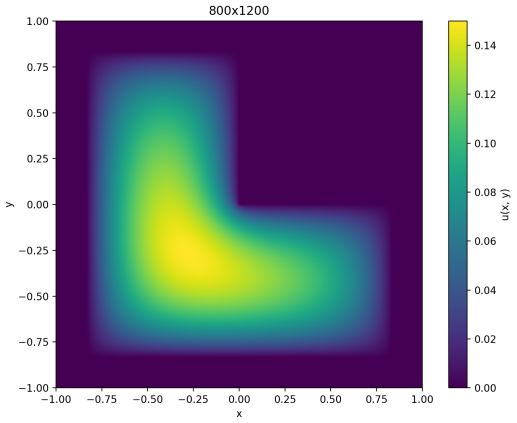
Визуализация результатов приведена на Рис. 3.

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



(a) Тепловая карта для сетки 400×600



(b) Тепловая карта для сетки 800×1200

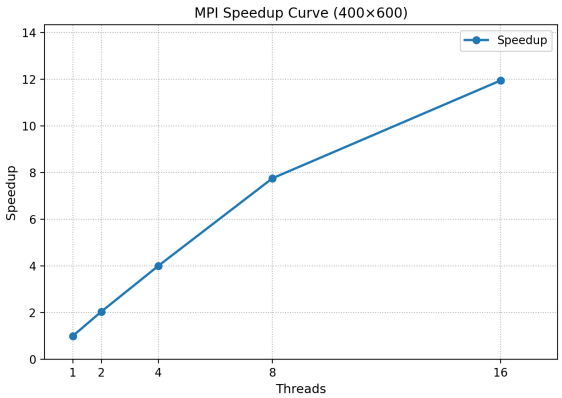
Рис. 3: Тепловая карта MPI

5.9 Производительность и ускорение

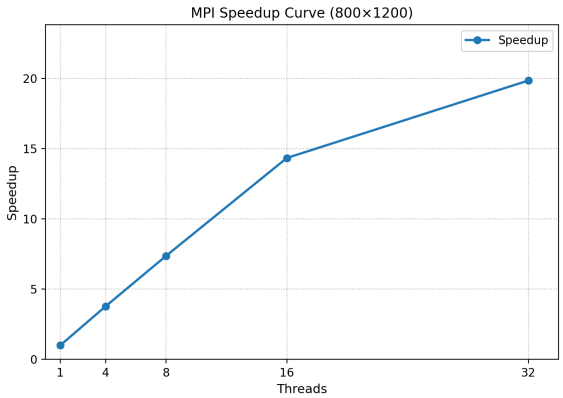
Число процессов	M×N	Итераций	Время (s)	Ускорение
1	400×600	2327	64.907571	1.000
1	800×1200	4486	472.104478	1.000

Число процессов	M×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:

```
__global__ void heat_step(const float *u, float *unew, int ny, int nx)
{
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;

    if (i > 0 && i < nx-1 && j > 0 && j < ny-1) {
        unew[i*ny + j] = 0.25f * (
            u[(i-1)*ny + j] + u[(i+1)*ny + j] +
```

```
            u[i*ny + (j-1)] + u[i*ny + (j+1)]
        );
    }
}
```

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:

```
ARCH=sm_60
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

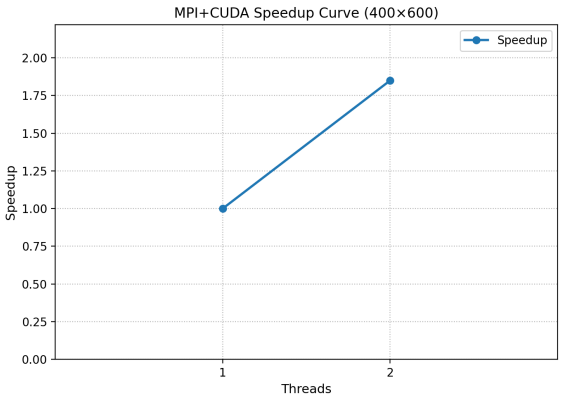
Результаты последовательного кода:

Число процессов	M×N	Итераций	Время (s)
1	400×600	1164	73.6218
1	800×1200	2344	592.356

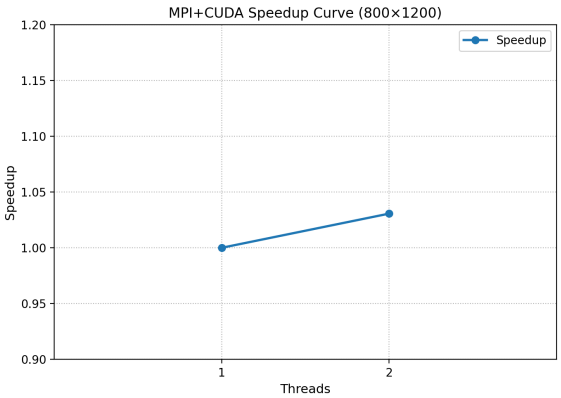
Результаты кода MPI+CUDA:

Число процессов	M×N	Итераций	Время (s)	Ускорение
1	400×600	2327	9.505309	1.00
2	400×600	2327	5.138451	1.85
1	800×1200	4486	74.488343	1.00
2	800×1200	4486	70.162509	1.06

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



(a) 400 × 600



(b) 800 × 1200

Рис. 5: Графики ускорения MPI+CUDA

6.3 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.

6.3.1 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:

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

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

6.3.2 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.

6.3.3 Проверка версии MPI+CUDA

Гибридная версия MPI+CUDA была проверена более тщательно:

(a) 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.

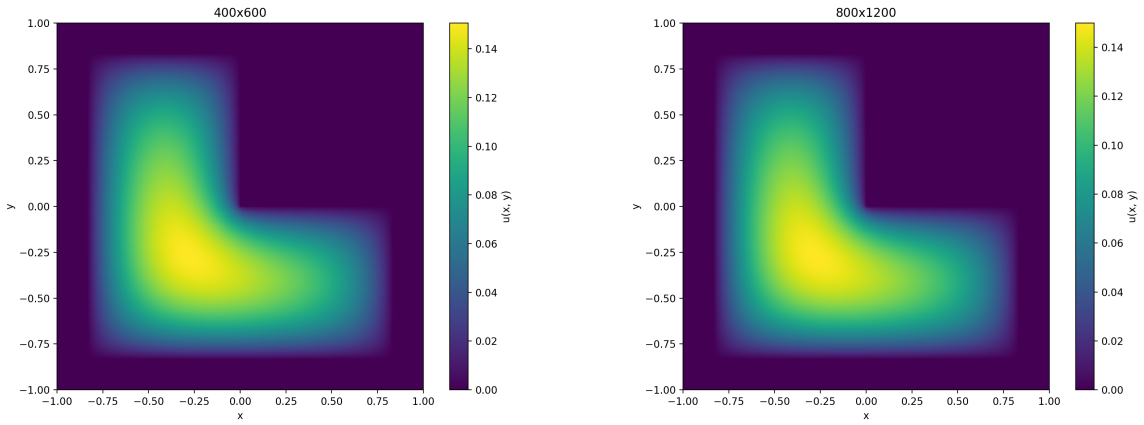
(b) Многопроцессный режим После окончания вычислений данные всех MPI-процессов собирались на нулевой процесс и формировали глобальное решение. Оно сравнивалось с эталонным решением на той же сетке. Максимальное отличие также не превышало 10^{-6}.

(c) 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

6.4 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
T_{init}	0.007408	0.002936
T_{PCG}	9.505309	5.138451
T_{comm}	0.023716	0.166008
T_{H2D}	0.729567	0.305095
T_{D2H}	0.743440	0.481848
T_A	0.137834	0.296752
$T_{D^{-1}}$	0.139519	0.321395
T_{final}	0.266149	0.254848
T_{total}	9.778865	5.396234

Таблица 4: 400x600

Число процессов	1	2
T_{init}	0.029628	0.009562
T_{PCG}	74.488343	70.162509
T_{comm}	0.078059	0.894409
T_{H2D}	5.867452	3.275765
T_{D2H}	6.091555	3.191888
T_A	0.457079	1.075008
$T_{D^{-1}}$	0.427959	1.003918
T_{final}	0.743910	0.738399
T_{total}	75.261881	0.738399

Таблица 5: 800x1200

6.5 Ссылка на репозиторий Git

https://github.com/JerryRen-41/SuperComputer