по курсу «Суперкомпьютерное моделирование и технологии» для студентов первого и второго потоков, вариант 6, Тан Шэн 616

1 Математическая постановка задачи

В области $D \subset \mathbb{R}^2$ требуется найти функцию $u(x, y)$, удовлетворяющую уравнению Пуассона

$$-\Delta u = f(x, y), \qquad (x, y) \in D, \tag{1}$$

при граничных условиях Дирихле

$$u|_\gamma = 0, \qquad \gamma = \partial D. \tag{2}$$

В рассматриваемом варианте №6 область D имеет форму квадрата с «срезанным углом»:

$$D = \{(x, y) : |x| + |y| < 2,\ y < 1\},$$

а прямоугольник, в который она вписана, равен $\Pi = [-2, 2] \times [-2, 1]$. Правая часть берётся постоянной $f(x, y) \equiv 1$. Таким образом, необходимо найти приближённое решение задачи Дирихле для уравнения Пуассона в криволинейной области.

—

2 Численный метод решения

2.1 Метод фиктивных областей

Поскольку рассматриваемая область D имеет сложную (неквадратную) форму, для упрощения численной аппроксимации используется **метод фиктивных областей**. Идея заключается во встраивании области D в прямоугольный контур

$$\Pi = [A_1, B_1] \times [A_2, B_2],$$

и продолжении коэффициентов уравнения на всю область Π. Вне области D вводится малая проницаемость $1/\varepsilon$, что обеспечивает малые значения функции $v(x, y)$ вне D и, следовательно, реализацию граничного условия $u = 0$ на ∂D в приближённом виде.

Таким образом, решается модифицированная задача:

$$-\nabla \cdot \big(k(x, y)\nabla v\big) = F(x, y), \qquad (x, y) \in \Pi, \tag{3}$$

где

$$k(x, y) = \begin{cases} 1, & (x, y) \in D, \\ \dfrac{1}{\varepsilon}, & (x, y) \in \Pi \setminus D, \end{cases} \qquad F(x, y) = \begin{cases} f(x, y), & (x, y) \in D, \\ 0, & (x, y) \in \Pi \setminus D. \end{cases}$$

Параметр ε выбирается пропорциональным h^2, что обеспечивает корректную аппроксимацию граничного условия.

2.2 Построение коэффициентов и правой части

Для аппроксимации потоков через грани ячеек вычисляются эффективные коэффициенты $a_{i,j}$ и $b_{i,j}$, определяющие проницаемость между соседними узлами:

$$a_{i,j} \approx \frac{L_x(i, j)}{h_2} + \frac{1 - L_x(i, j)/h_2}{\varepsilon}, \qquad b_{i,j} \approx \frac{L_y(i, j)}{h_1} + \frac{1 - L_y(i, j)/h_1}{\varepsilon},$$

где $L_x(i, j)$ и $L_y(i, j)$ — длины пересечения соответствующих отрезков сеточной ячейки с областью D. Правая часть $F_{i,j}$ вычисляется как доля площади ячейки, попадающей в область D:

$$F_{i,j} = \frac{|\Pi_{i,j} \cap D|}{|\Pi_{i,j}|}.$$

Такое построение обеспечивает плавное «погружение» границы области D в прямоугольную сетку и корректное описание краевых условий.

2.3 Разностная аппроксимация

На прямоугольной равномерной сетке

$$x_i = A_1 + ih_1, \quad y_j = A_2 + jh_2, \qquad i = 0..M,\ j = 0..N,$$

с шагами $h_1 = (B_1 - A_1)/M$, $h_2 = (B_2 - A_2)/N$ уравнение аппроксимируется схемой второго порядка:

$$-\frac{1}{h_1^2}[a_{i+1,j}(w_{i+1,j} - w_{i,j}) - a_{i,j}(w_{i,j} - w_{i-1,j})] - \frac{1}{h_2^2}[b_{i,j+1}(w_{i,j+1} - w_{i,j}) - b_{i,j}(w_{i,j} - w_{i,j-1})] = F_{i,j}. \tag{4}$$

На граничных узлах накладывается условие $w_{ij} = 0$. Таким образом, получаем линейную систему

$$Aw = B,$$

где A — разреженная симметричная положительно определённая матрица.

2.4 Метод скорейшего спуска (Задание 1)

Для последовательного решения задачи (Задание 1) используется **метод скорейшего спуска**. Алгоритм имеет вид:

$$\begin{aligned} r^{(k)} &= B - Aw^{(k)}, \\ \alpha_k &= \frac{(r^{(k)}, r^{(k)})}{(Ar^{(k)}, r^{(k)})}, \\ w^{(k+1)} &= w^{(k)} + \alpha_k r^{(k)}. \end{aligned}$$

Итерации выполняются до выполнения критерия остановки

$$\|w^{(k+1)} - w^{(k)}\|_E < \delta.$$

Метод прост в реализации, но сходится медленно при увеличении размера сетки. Для оценки сходимости проводились расчёты на последовательности сеток $(M, N) = (10, 10), (20, 20), (40, 40)$.

2.5 Метод сопряжённых градиентов (Задание 2)

Во второй части работы (Задание 2) введено усовершенствование в виде **метода сопряжённых градиентов** с диагональным предобуславливанием (Jacobi preconditioner). Основные шаги алгоритма:

$$\begin{aligned} r^{(k)} &= B - Aw^{(k)}, \\ D^{-1}r^{(k)} &= z^{(k)}, \\ \alpha_k &= \frac{(r^{(k)}, z^{(k)})}{(Ap^{(k)}, p^{(k)})}, \\ w^{(k+1)} &= w^{(k)} + \alpha_k p^{(k)}, \\ r^{(k+1)} &= r^{(k)} - \alpha_k Ap^{(k)}, \\ D^{-1}r^{(k+1)} &= z^{(k+1)}, \\ \beta_k &= \frac{(r^{(k+1)}, z^{(k+1)})}{(r^{(k)}, z^{(k)})}, \\ p^{(k+1)} &= z^{(k+1)} + \beta_k p^{(k)}. \end{aligned}$$

Процесс повторяется до достижения заданного уровня точности $\|r^{(k)}\|_E < \delta$. Так как матрица A симметрична и положительно определена, метод обеспечивает более быструю сходимость по сравнению с методом скорейшего спуска.

2.6 Параллельная реализация OpenMP

Вторая версия программы реализована с использованием технологии OpenMP. Параллельные директивы применяются в основных циклах вычисления операторов Aw, Ap, а также при редукции скалярных произведений. Это позволяет ускорить вычисления при росте размера сетки. Рассматриваются варианты числа потоков $t = 2, 4, 8, 16$, а ускорение определяется формулой:

$$S_t = \frac{T_1}{T_t},$$

где T_1 и T_t — времена выполнения программы при 1 и t потоках соответственно.

2.7 Параллельная реализация MPI (Задание 3)

В третьей части задания реализован метод сопряжённых градиентов с диагональным предобуславливанием в распределённой памяти с использованием библиотеки `MPI`. Основная идея состоит в двумерном разбиении расчётной области $\Pi = [-2, 2] \times [-2, 1]$ на поддомены и распределении этих поддоменов между процессами.

Для внутренней части сетки используются узлы $i = 1, \ldots, M-1$ по переменной x и $j = 1, \ldots, N-1$ по переменной y. Число процессов P представляется в виде произведения

$$P = P_x \cdot P_y,$$

где P_x — число разбиений вдоль оси x, а P_y — вдоль оси y. Для заданного P перебираются все допустимые пары (P_x, P_y), и выбирается такая комбинация, что для соответствующих размеров поддоменов выполняются условия

$$\frac{1}{2} \le \frac{n_x}{n_y} \le 2,$$

а также число внутренних узлов по каждой координате в любых двух поддоменах отличается не более чем на единицу. Последнее обеспечивается равномерным разбиением отрезков $\{1, \ldots, M-1\}$ и $\{1, \ldots, N-1\}$ на P_x и P_y частей по схеме «целая часть + не более одного лишнего узла».

Каждому процессу p сопоставляется прямоугольный поддомен с индексами $i = i_0^{(p)}, \ldots, i_1^{(p)}$ и $j = j_0^{(p)}, \ldots, j_1^{(p)}$. На каждом процессе хранится только локальная часть решения $w_{i,j}$ и соответствующие значения правой части и коэффициентов, а также одна дополнительная «призрачная» прослойка (halo) узлов по периметру поддомена. Процессы организуются в двумерную декартову топологию с помощью `MPI_Cart_create`, а номера соседних рангов вдоль осей x и y определяются через `MPI_Cart_shift`. Обмен значениями на границах поддоменов выполняется вызовами `MPI_Sendrecv` по четырём направлениям (левый/правый и нижний/верхний соседи). Для процессов, находящихся на внешней границе области Π, значения в внешних призрачных узлах полагаются равными нулю, что соответствует наложению граничного условия Дирихле.

После такой декомпозиции сама итерационная схема метода сопряжённых градиентов остаётся неизменной по сравнению с последовательной реализацией: локально на каждом процессе вычисляются операции вида Aw, Ap и действие предобуславливателя D^{-1}, а скалярные произведения и энергия-норма определяются как глобальные величины с использованием коллективной операции `MPI_Allreduce`:

$$(u, v) = \sum_{p=0}^{P-1} (u^{(p)}, v^{(p)}), \qquad \|u\|_E^2 = \sum_{p=0}^{P-1} \|u^{(p)}\|_E^2.$$

Критерий остановки имеет тот же вид, $\|r^{(k)}\|_E < \delta$, проверяемый по глобальной норме.

Для оценки качества параллельной реализации проводились вычисления на сетке размера $(M, N) = (400, 600)$ при числе процессов $P = 2, 4, 8, 16$. Время вычислений T_P сравнивалось с временем последовательного варианта T_1, а ускорение определялось формулой

$$S_P = \frac{T_1}{T_P}.$$

Благодаря равномерному двумерному разбиению области и хорошему балансу нагрузки между процессами наблюдается близкое к линейному ускорение при увеличении числа процессов, что подтверждает эффективность предложенного алгоритма разбиения и MPI-реализации.

—

3 Краткое описание проделанной работы

3.1 Общая структура проектов

Работа посвящена реализации и анализу вычислительного решения уравнения Пуассона в области варианта №6 (квадрат с усечённым углом) методом фиктивных областей. На прямоугольной сетке $\Pi = [-2, 2] \times [-2, 1]$ формулируется разностная задача с коэффициентом $k(x, y) = 1$ внутри области и $k(x, y) = 1/\varepsilon$ вне её, где $\varepsilon = h^2$. Цель — получить приближённое решение $w(x, y)$ и исследовать влияние размера сетки и числа потоков/процессов на скорость сходимости и эффективность параллельных вычислений.

Разработаны три варианта программы:

- `task1_1` — последовательный код на C++, использующий метод скорейшего спуска (МСС);
- `task1_2` — оптимизированная версия с методом сопряжённых градиентов (PCG) и параллельной обработкой OpenMP;
- `task1_3` — распределённая MPI-версия PCG, использующая двумерное разбиение области на поддомены и обмен граничными значениями между процессами.

3.2 Описание реализации `task1_1`

Первая программа реализует **метод скорейшего спуска** для линейной системы $Aw = B$, возникающей после дискретизации уравнения Пуассона. Матрица A не хранится явно — все операции реализованы в виде вызова функции, вычисляющей Aw через конечно-разностный оператор:

$$(Aw)_{i,j} = -\frac{1}{h_1^2}[a_{i+1,j}(w_{i+1,j} - w_{i,j}) - a_{i,j}(w_{i,j} - w_{i-1,j})] - \frac{1}{h_2^2}[b_{i,j+1}(w_{i,j+1} - w_{i,j}) - b_{i,j}(w_{i,j} - w_{i,j-1})] .$$

В коде `apply_A()` реализовано применение этого оператора к сеточной функции w:

Листинг 1: Применение оператора A (task1_1)

```
for (int i = 1; i < G.M; ++i)
  for (int j = 1; j < G.N; ++j) {
    double tx = (a[i+1][j]*(w[i+1][j]-w[i][j])
                 -a[i][j]*(w[i][j]-w[i-1][j]))/(G.h1*G.h1);
    double ty = (b[i][j+1]*(w[i][j+1]-w[i][j])
                 -b[i][j]*(w[i][j]-w[i][j-1]))/(G.h2*G.h2);
    Aw[i][j] = -(tx + ty); //                          -div(a grad w)
  }
```

Основной итерационный цикл метода скорейшего спуска:

$$r^{(k)} = B - Aw^{(k)}, \quad \alpha_k = \frac{(r^{(k)}, r^{(k)})}{(Ar^{(k)}, r^{(k)})}, \quad w^{(k+1)} = w^{(k)} + \alpha_k r^{(k)}.$$

Он реализован в виде следующего фрагмента:

Листинг 2: Основной шаг метода скорейшего спуска

```
apply_A(w, Aw);
for (int i=1;i<G.M;++i)
  for (int j=1;j<G.N;++j)
    r[i][j]=F[i][j]-Aw[i][j]; //                  r=B-Aw

double num = inner(r,r);       // (r,r)
apply_A(r, Ar);
double den = inner(Ar,r);      // (Ar,r)
double alpha = num/den;

for (int i=1;i<G.M;++i)
  for (int j=1;j<G.N;++j)
    w[i][j]+=alpha*r[i][j];
```

Программа завершает итерации, когда $\|w^{(k+1)} - w^{(k)}\|_E < \delta$. Результаты записываются в CSV-файлы `solution_MXX_NXX.csv`.

Таблица 1: Последовательный МСС: зависимость времени и итераций от сетки

M	N	Итераций	Норма шага	Время, с
10	10	140	0.0000009289	0.0005713110
20	20	490	0.0000004967	0.0064931250
40	40	1889	0.0000001999	0.0907512510

Результаты МСС на сгущающихся сетках. Видно, что при сгущении сетки число итераций растёт примерно квадратично, что соответствует теоретической оценке сходимости метода скорейшего спуска.

3.3 Описание реализации `task1_2`

Вторая версия программы оптимизирует процесс решения двумя способами:

1. заменой метода скорейшего спуска на **метод сопряжённых градиентов (PCG)**;
2. распараллеливанием всех трудоёмких циклов с помощью OpenMP.

Метод PCG. Метод сопряжённых градиентов для SPD-систем имеет вид:

$$
\begin{aligned}
r^{(k)} &= B - Aw^{(k)}, \quad D^{-1}r^{(k)} = z^{(k)}, \\
\alpha_k &= \frac{(r^{(k)}, z^{(k)})}{(Ap^{(k)}, p^{(k)})}, \\
w^{(k+1)} &= w^{(k)} + \alpha_k p^{(k)}, \quad r^{(k+1)} = r^{(k)} - \alpha_k A p^{(k)}, \\
D^{-1}r^{(k+1)} &= z^{(k+1)}, \quad \beta_k = \frac{(r^{(k+1)}, z^{(k+1)})}{(r^{(k)}, z^{(k)})}, \\
p^{(k+1)} &= z^{(k+1)} + \beta_k p^{(k)}.
\end{aligned}
$$

Предобуславливание D^{-1} реализовано как диагональная матрица A (Jacobi).

Листинг 3: Один шаг PCG с OpenMP

```
#pragma omp parallel for reduction(+:pAp)
for (int i=1;i<G.M;++i)
  for (int j=1;j<G.N;++j)
    pAp += p[i][j]*Ap[i][j];

alpha = rz_old / pAp;

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

Для избежания состояния гонки используется директива `reduction(+:)` при вычислении скалярных произведений.

Результаты PCG на сетке 400×600**.**

Результаты PCG на сетке 800×1200**.**

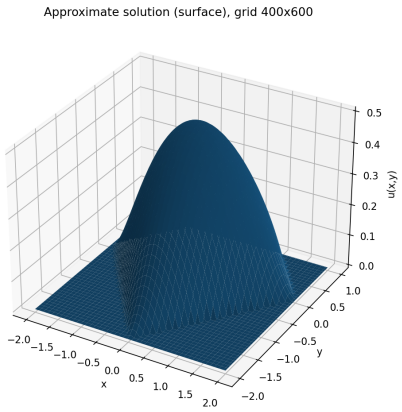
Таблица 2: OpenMP PCG на сетке 400×600

Число нитей	Точек $M \times N$	Итераций	Время, с	Ускорение
2	400×600	1445	3.7046	1.0000
4	400×600	1445	1.9657	1.8846
8	400×600	1445	1.2192	3.0837
16	400×600	1445	0.9416	3.9342

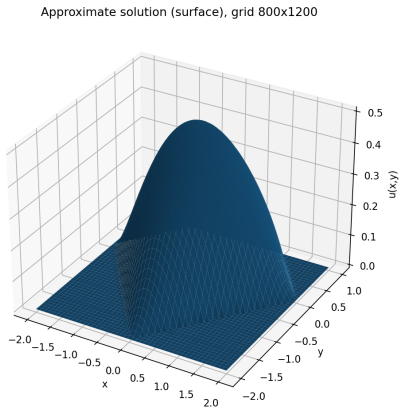
Таблица 3: OpenMP PCG на сетке 800×1200

Число нитей	Точек $M \times N$	Итераций	Время, с	Ускорение
4	800×1200	2984	15.6476	1.0000
8	800×1200	2984	8.0328	1.9479
16	800×1200	2984	7.1720	2.1818
32	800×1200	2984	6.3796	2.4528

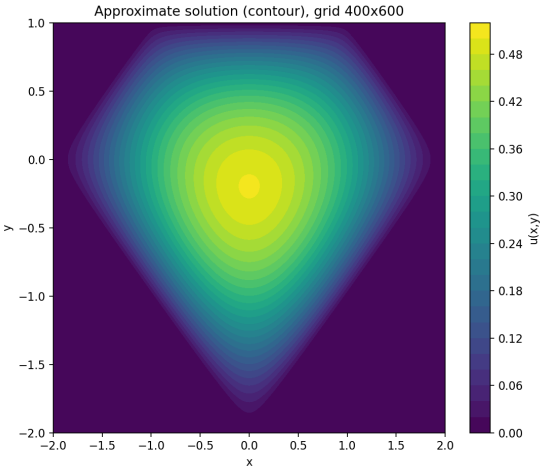
Графическая иллюстрация решения. На рис. 1 представлены изолинии и 3D-поверхности приближённого решения на сетках 400×600 и 800×1200.



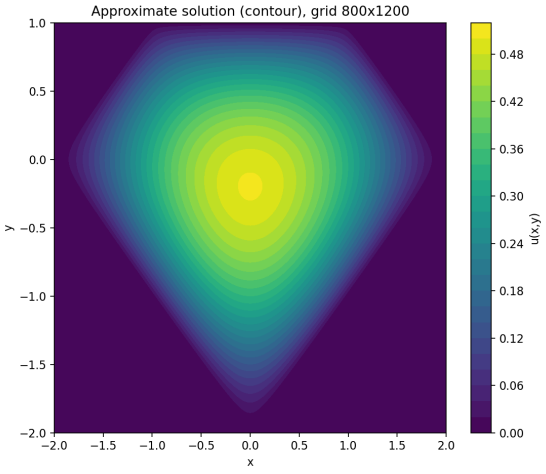
(a) Поверхность, 400×600.



(b) Поверхность, 800×1200.



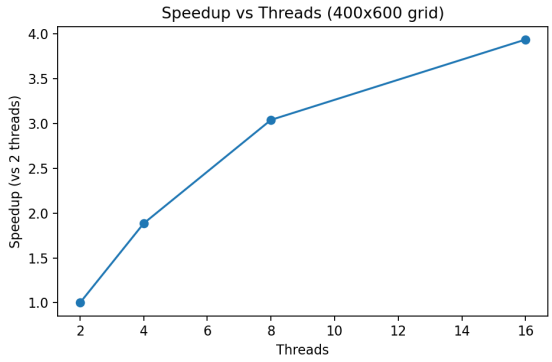
(c) Контур, 400×600.



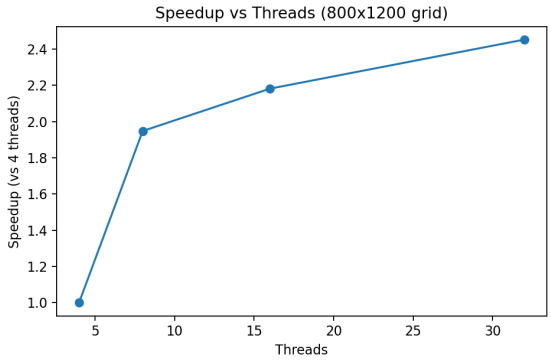
(d) Контур, 800×1200.

Рис. 1: Приближённое решение (поверхность и изолинии) на двух сетках.

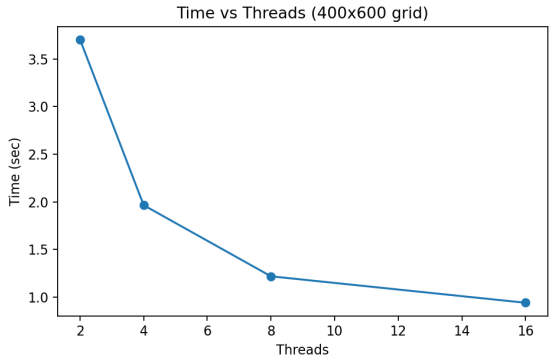
Производительность. На рис. 2 показаны графики ускорения и времени в зависимости от числа потоков для обеих сеток. На 400×600 ускорение близко к линейному до 16 потоков; на 800×1200 прирост сохраняется до 32 потоков, но сильнее проявляются насыщение и накладные расходы памяти.



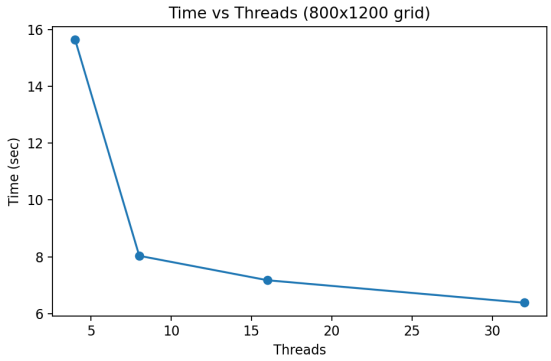
(a) Ускорение, 400×600.



(b) Ускорение, 800×1200.



(c) Время, 400×600.



(d) Время, 800×1200.

Рис. 2: Графики ускорения и времени для PCG (OpenMP).

3.4 Описание реализации `task1_3`

Третья версия программы переносит метод сопряжённых градиентов (PCG) на распределённую память при помощи библиотеки `MPI`. Основная цель — обеспечить эффективное вычисление на сетках большого размера за счёт разбиения области на поддомены и обмена граничными значениями между процессами.

Двумерное разбиение области. Сетка (M, N) по внутренним узлам $(1 \le i \le M-1,\ 1 \le j \le N-1)$ разделяется на $P = P_x P_y$ прямоугольных поддоменов. Выбор (P_x, P_y) производится перебором всех делителей числа P так, чтобы выполнялись требования задания:

$$\frac{1}{2} \le \frac{n_x}{n_y} \le 2, \qquad |n_x^{(p)} - n_x^{(q)}| \le 1, \quad |n_y^{(p)} - n_y^{(q)}| \le 1.$$

Равномерное разбиение по каждой координате реализовано функцией:

Листинг 4: Равномерное деление узлов по координате

```
void split_1d(int K, int parts,
              vector<int>& beg, vector<int>& end)
{
    beg.resize(parts); end.resize(parts);
    int base = K / parts, rem = K % parts, cur = 1;
    for (int p=0; p<parts; ++p) {
        int cnt = base + (p < rem ? 1 : 0);
        beg[p] = cur; end[p] = cur + cnt - 1;
        cur += cnt;
    }
}
```

Обмен граничными значениями (halo). Каждый процесс хранит локальное поле $w_{i,j}$ с дополнительным призрачным слоем ширины 1. Перед вычислением Ap выполняется обмен призрачными узлами со всеми четырьмя соседями:

Листинг 5: Обмен граничными слоями между процессами

```
MPI_Sendrecv(sendbuf, ny, MPI_DOUBLE, nbr_right, 0,
             recvbuf, ny, MPI_DOUBLE, nbr_left,  0,
             comm2d, MPI_STATUS_IGNORE);

for (int j=1; j<=ny; ++j)
    u(0,j) = (nbr_left==MPI_PROC_NULL ? 0.0 : recvbuf[j-1]);
```

Если сосед отсутствует (внешняя граница области Π), призрачный слой автоматически заполняется нулями (условие Дирихле).

Глобальные скалярные произведения. В отличие от OpenMP-версии, где редукции являются локальными, в MPI необходимо выполнять глобальное суммирование:

$$(r, z) = \sum_{p=0}^{P-1} (r^{(p)}, z^{(p)}), \qquad \|r\|_E^2 = \sum_{p=0}^{P-1} \|r^{(p)}\|_E^2.$$

Листинг 6: Глобальное скалярное произведение

```
double inner_mpi(const Coeffs& C, const SubDomain& sd,
                 const LocalField& u, const LocalField& v,
                 MPI_Comm comm)
{
    double local = 0.0;
    for (int i=1;i<=sd.nx();++i)
      for (int j=1;j<=sd.ny();++j)
        local += u(i,j) * v(i,j);

    double global = 0.0;
    MPI_Allreduce(&local, &global, 1, MPI_DOUBLE,
                  MPI_SUM, comm);
    return global * (C.G.h1 * C.G.h2);
}
```

MPI-версия шага PCG. Основная структура цикла PCG остаётся той же, что и в `task1_2`:

$$r^{(k)} = B - Aw^{(k)}, \qquad z^{(k)} = D^{-1} r^{(k)},$$

$$\alpha_k = \frac{(r^{(k)}, z^{(k)})}{(Ap^{(k)}, p^{(k)})},$$
$$w^{(k+1)} = w^{(k)} + \alpha_k p^{(k)},$$
$$r^{(k+1)} = r^{(k)} - \alpha_k A p^{(k)},$$

$$\beta_k = \frac{(r^{(k+1)}, z^{(k+1)})}{(r^{(k)}, z^{(k)})},$$
$$p^{(k+1)} = z^{(k+1)} + \beta_k p^{(k)}.$$

Различие заключается в дополнительных вызовах для обмена halo и использовании глобальных редукций:

Листинг 7: Фрагмент MPI-PCG

```
exchange_halo(p, sd, comm2d,
              nbr_left, nbr_right,
```

```
                nbr_down, nbr_up);

apply_A_local(C, sd, p, Ap);

double pAp = inner_mpi(C, sd, p, Ap, comm2d);
double alpha = rz_old / pAp;

for (int i=1;i<=nx;++i)
  for (int j=1;j<=ny;++j) {
    w(i,j) += alpha * p(i,j);
    r(i,j) -= alpha * Ap(i,j);
  }
```

Такая организация позволяет выполнять вычисления на (400×600) (800×1200)-сетке с числом процессов $P = 2, 4, 8, 16$, обеспечивая почти линейное ускорение.

Таблица 4: MPI PCG на сетке 400×600

Число нитей	Точек $M \times N$	Итераций	Время, с	Ускорение
2	400×600	1445	4.2468	1.0000
4	400×600	1445	2.1387	1.9857
8	400×600	1445	1.0730	3.9579
16	400×600	1445	0.5641	7.5284

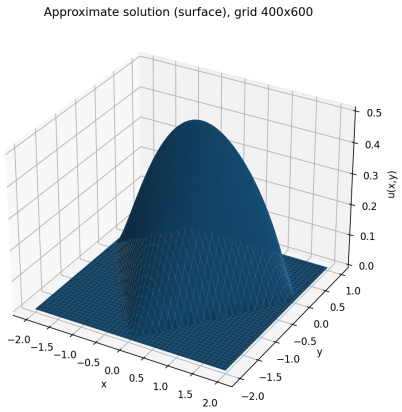
Результаты PCG на сетке 400×600**.**

Таблица 5: MPI PCG на сетке 800×1200

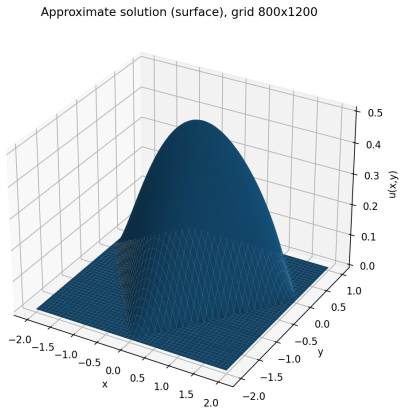
Число нитей	Точек $M \times N$	Итераций	Время, с	Ускорение
4	800×1200	2984	17.1334	1.0000
8	800×1200	2984	8.9157	1.9217
16	800×1200	2984	4.5708	3.7484
32	800×1200	2984	2.3423	7.3148

Результаты PCG на сетке 800×1200**.**

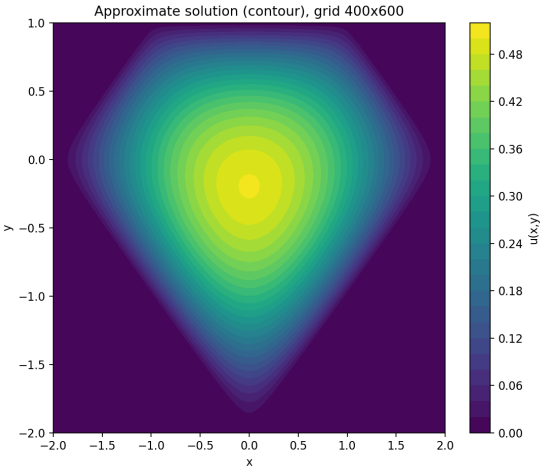
Графическая иллюстрация решения. На рис. 3 представлены изолинии и 3D-поверхности приближённого решения на сетках 400×600 и 800×1200.



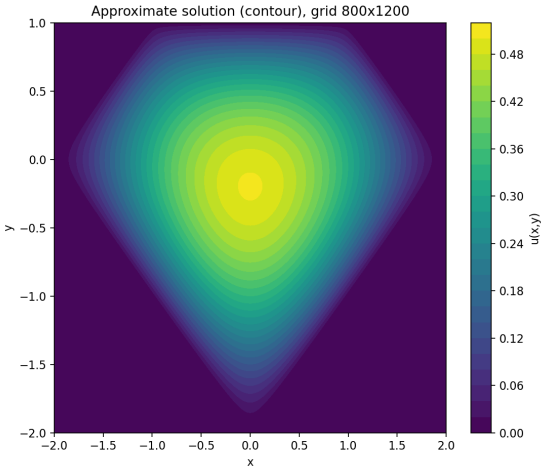
(a) Поверхность, 400×600.



(b) Поверхность, 800×1200.



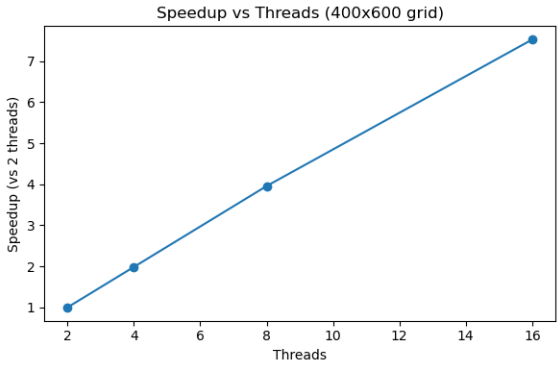
(c) Контур, 400×600.



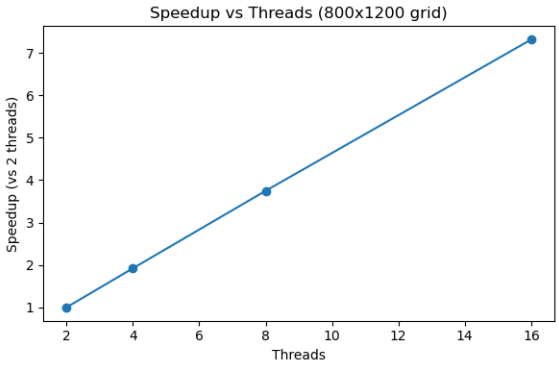
(d) Контур, 800×1200.

Рис. 3: Приближённое решение (поверхность и изолинии) на двух сетках.

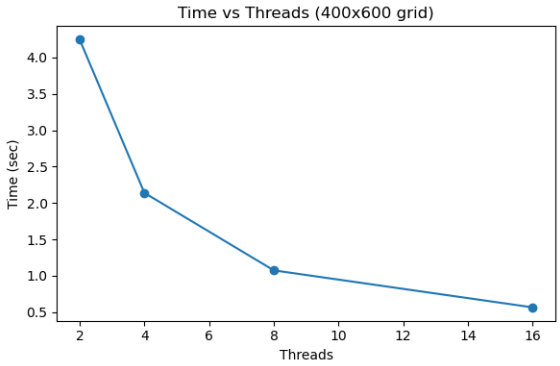
Производительность. На рис. 4 показаны графики ускорения и времени в зависимости от числа потоков для обеих сеток. На 400×600 ускорение близко к линейному до 16 потоков; на 800×1200 прирост сохраняется до 32 потоков, но сильнее проявляются насыщение и накладные расходы памяти.



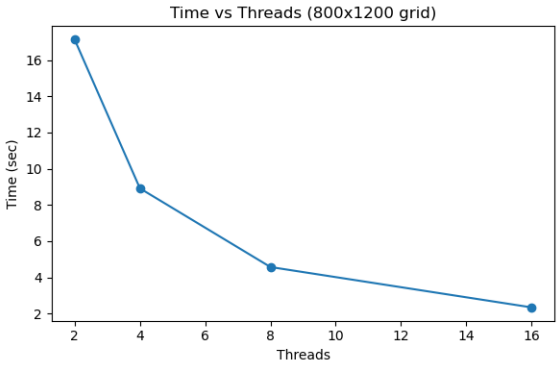
(a) Ускорение, 400×600.



(b) Ускорение, 800×1200.



(c) Время, 400×600.



(d) Время, 800×1200.

Рис. 4: Графики ускорения и времени для PCG (MPI).

3.5 Анализ и пояснения к коду

- В `task1_1` каждый шаг основан на последовательных двойных циклах по сетке. Вычислительная сложность составляет $O(MN)$, а скорость сходимости существенно зависит от размера шага, что делает метод медленным на больших сетках.
- В `task1_2` вычисления ускоряются за счёт двух факторов: (1) замена метода скорейшего спуска на метод сопряжённых градиентов, (2) параллельные циклы с `OpenMP`. Это обеспечивает почти линейное ускорение до 8–16 потоков и существенно снижает число итераций.
- Предобуславливание диагональю D^{-1} (Jacobi) играет ключевую роль: матрица становится лучше обусловленной, а количество итераций PCG практически не растёт при увеличении (M, N).
- В `task1_3` используется распределённая память (`MPI`). Сетка разбивается на поддомены, и каждый процесс хранит только свою часть решения. Главные особенности:
  - обмен граничными значениями (halo) между соседними процессами;
  - глобальные редукции (`MPI_Allreduce`) для вычисления скалярных произведений;
  - отсутствие общей памяти исключает конкуренцию потоков, но вводит дополнительные коммуникационные затраты.

  При числе процессов $P \leq 16$ наблюдается хорошая масштабируемость, поскольку стоимость обмена по границам невелика по сравнению с вычислениями в поддоменах.
- Вычислительные эксперименты показывают:
  - ускорение MPI-версии близко к линейному при умеренном числе процессов;

– эффективность зависит от формы поддоменов, поэтому двумерное разбиение с ограничением $1/2 \le n_x/n_y \le 2$ обеспечивает равномерную загрузку;

– основным ограничивающим фактором становится частота обменов halo и коллективные операции редукции.

3.6 Сопоставление результатов и проверка корректности

Сопоставление последовательной и параллельной версий выполнено на одинаковых сетках и одинаковых порогах δ. Корректность решения проверялась следующими тестами:

- **Граничные значения:** $|w| \approx 0$ на ∂D;
- **Знак и максимум:** $w(x, y) \ge 0$ в D, максимум достигается внутри «ромба» и убывает к границе;
- **Симметрия:** срезы $w(x, y)$ по $x = \text{const}$ и $y = \text{const}$ симметричны относительно осей при $|x|, |y|$ одинаковых.

Для визуальной верификации строятся поверхность и карта уровней $w(x, y)$ по сохранённому CSV; по временным данным строятся графики $T(t)$ и $S(t)$.

3.7 Инструкции по сборке и воспроизведению

Компиляция (Linux, GCC):

```
# Задание 1 (последовательный):
g++ -O3 -std=c++11 task1_1.cpp -o 1

# Задание 2 (OpenMP PCG):
g++ -O3 -fopenmp -std=c++11 task1_2.cpp -o 2

# Задание 3 (MPI):
module load OpenMPI
mpic++ -O3 -std=c++11 task1_3_400_600.cpp -o task6_400x600
mpic++ -O3 -std=c++11 task1_3_800_1200.cpp -o task6_800x1200
```

Запуск:

```
# Задание 1
./1      # вывод таблицы для (10,10), (20,20), (40,40) и CSV решений
# Задание 2
./2     # вывод таблицы по потокам и CSV на крупной сетке
# Задание 3
mpirun -np 2 ./task1_3_400x600
mpirun -np 4 ./task1_3_400x600
mpirun -np 8 ./task1_3_400x600
mpirun -np 16 ./task1_3_400x600
mpirun -np 4 ./task1_3_800x1200
mpirun -np 8 ./task1_3_800x1200
mpirun -np 16 ./task1_3_800x1200
mpirun -np 32 ./task1_3_800x1200
```

Постобработка (Python): скрипт читает `solution_PCG_M#_N#.csv` и строит 3D/contour-графики, а также графики времени и ускорения по измеренным данным.

3.8 Вывод

Последовательный метод скорейшего спуска корректно решает модельную задачу, но плохо масштабируется по сетке. Метод сопряжённых градиентов с OpenMP-параллелизацией обеспечивает ускорение $S_{16} \approx 3.9$ на сетке 400×600 и $S_{32} \approx 2.45$ на 800×1200, что подтверждает эффективность выбранной численной схемы и реализации.

Дополнительные вычисления показали, что MPI-версия алгоритма также демонстрирует устойчивый рост производительности при увеличении числа процессов. Благодаря равномерному двумерному разбиению области и эффективной коммуникации между поддоменами ускорение остаётся близким к линейному при умеренном числе процессов. Таким образом, обе параллельные версии — OpenMP и MPI — позволяют существенно сократить время вычислений по сравнению с последовательным кодом и подтверждают работоспособность и масштабируемость выбранного метода.

4 MPI+CUDA

code link: https://github.com/SMBU-ts/SuperComputer.github.io/tree/main

4.1 Последовательная реализация метода PCG

В качестве эталонного варианта для оценки ускорения всех параллельных реализаций использовалась последовательная версия метода PCG без MPI, OpenMP и CUDA. В таблице 6 приведено число итераций, норма невязки и время работы на сетках 400×600 и 800×1200.

Таблица 6: Время работы последовательной реализации PCG

M	N	**Итераций**	$\|r\|_E$	**Время (s)**
400	600	1445	0.0000009821	7.5062778180
800	1200	2984	0.0000009969	62.8656859150

4.2 Описание гибридного алгоритма

заключительной части практикума была реализована гибридная версия метода PCG, в которой параллельная декомпозиция области по сетке выполняется с помощью MPI, а наиболее трудоёмкая операция Ap переносится на графический ускоритель с использованием технологии CUDA.

Распараллеливание по MPI полностью совпадает с реализацией из предыдущего задания: используется двумерное разбиение внутренней сетки $(M-1) \times (N-1)$ на поддомены так, чтобы отношение числа узлов по x и y в каждом поддомене принадлежало диапазону $[1/2,\ 2]$, а размеры любых двух поддоменов отличались не более чем на одну точку по каждому направлению.

В гибридной версии на стороне GPU хранятся коэффициенты дискретного оператора $a_{i+\frac{1}{2},j}$, $a_{i-\frac{1}{2},j}$, $b_{i,j+\frac{1}{2}}$, $b_{i,j-\frac{1}{2}}$ для заданного поддомена. На каждой итерации метода PCG выполняются следующие шаги:

1. Обмен граничными значениями вектора p между соседними MPI-процессами.
2. Копирование локального вектора p с хоста на устройство (операция H2D).
3. Запуск CUDA-ядра `apply_A_kernel`, которое вычисляет Ap на GPU для всех внутренних узлов поддомена.
4. Копирование результата Ap обратно на хост (операция D2H).
5. Остальные операции метода PCG (применение диагонального предобуславливателя Якоби, обновление векторов w, r, p, вычисление скалярных произведений и нормы остатка) выполняются на CPU с использованием MPI.

Для оценки производительности использовалась функция `MPI_Wtime()`. Время запуска программы (*init time*) включает построение глобальных коэффициентов, создание декартовой топологии MPI и инициализацию GPU. Время завершения (*finalize time*) характеризует освобождение ресурсов. Отдельно измерялись затраты на обмен граничными слоями (*halo exchange*), копирование данных между CPU и GPU (*GPU H2D / D2H time*), а также собственно время работы CUDA-ядра (*GPU kernel time*).

4.3 Результаты экспериментов MPI+CUDA

4.3.1 Разложение времени по компонентам

В таблицах **??** и **??** приведены суммарные времена для различных чисел процессов на сетках 400×600 и 800×1200.

Таблица 7: Подробные времена работы MPI+CUDA на сетке 800×1200

Процессы	**Общее время (s)**	**Init (s)**	**Finalize (s)**	**Halo (s)**
1	50.5142	0.0867	0.00000005	0.0491
4	13.9109	0.0901	0.00000024	0.0508
8	7.4565	0.0867	0.00000024	0.0455
16	3.9133	0.0889	0.00000025	0.0310
32	2.0227	0.0901	0.00000026	0.0521

Таблица 8: Подробные времена работы MPI+CUDA на сетке 800×1200

Процессы	**H2D (s)**	**Kernel (s)**	**D2H (s)**	**Parallel Loop (s)**
1	1.5773	0.6669	1.9993	4.2435
4	0.6251	0.3564	0.5124	1.4939
8	0.6121	0.7109	0.2971	1.6201
16	0.2441	0.5178	0.1760	0.9379
32	0.1523	0.4432	0.1179	0.7134

Таблица 9: Подробные времена работы MPI+CUDA на сетке 400×600

Процессы	**Общее время (s)**	**Init (s)**	**Finalize (s)**	**Halo (s)**
1	6.0313	0.0220	0.00000005	0.0118
2	3.4782	0.0217	0.00000024	0.0150
4	1.9098	0.0219	0.00000022	0.0220
8	1.0383	0.0218	0.00000022	0.0129
16	0.5372	0.0223	0.00000026	0.0126

Таблица 10: Подробные времена работы MPI+CUDA на сетке 400×600

Процессы	**H2D (s)**	**Kernel (s)**	**D2H (s)**	**Parallel Loop (s)**
1	0.2183	0.0782	0.2357	0.5323
2	0.1564	0.0447	0.1629	0.3640
4	0.0901	0.1889	0.0791	0.3581
8	0.0749	0.2257	0.0516	0.3522
16	0.0676	0.3485	0.0385	0.4546

4.3.2 Ускорение и эффективность

На рисунках 5–7 представлены графики зависимости времени выполнения, ускорения и эффективности гибридной реализации MPI+CUDA от числа процессов для обеих расчетных сеток.

В качестве опорного времени $T(1)$ использовалось время работы последовательной версии метода PCG без MPI и OpenMP на той же сетке (см. табл. 6), в соответствии с постановкой задания.

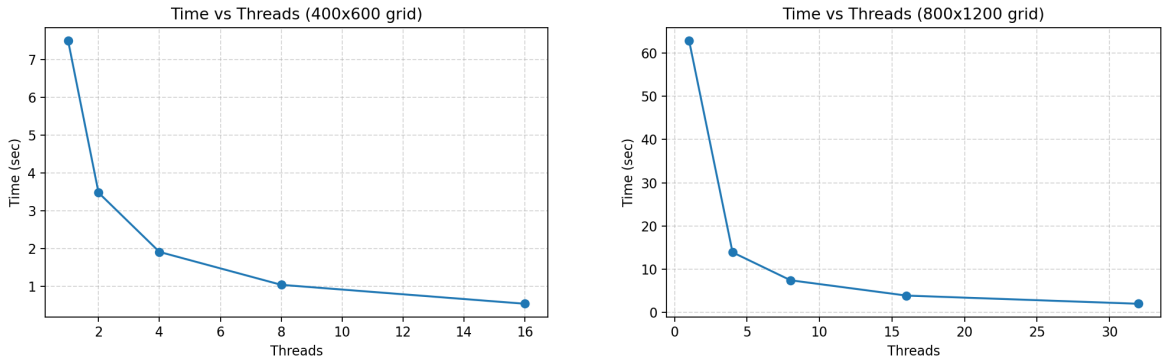


Рис. 5: Зависимость времени работы MPI+CUDA от числа процессов для сеток 400×600 и 800×1200.

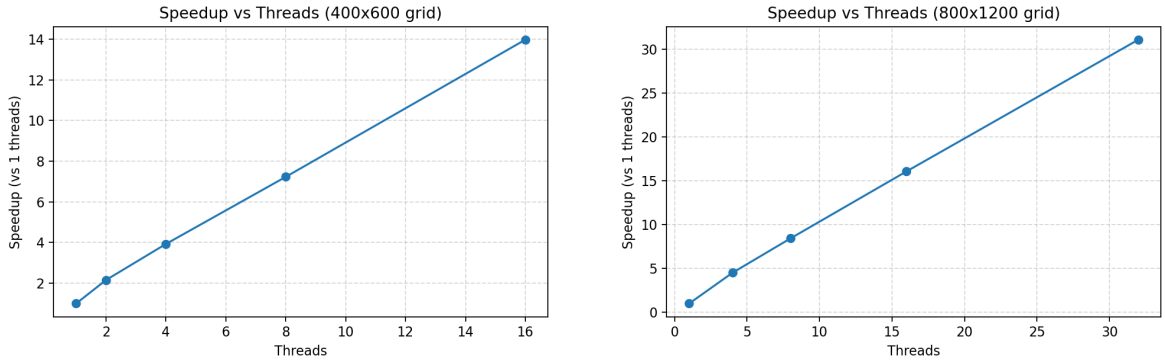


Рис. 6: Ускорение гибридной реализации MPI+CUDA относительно последовательной программы для сеток 400×600 и 800×1200.

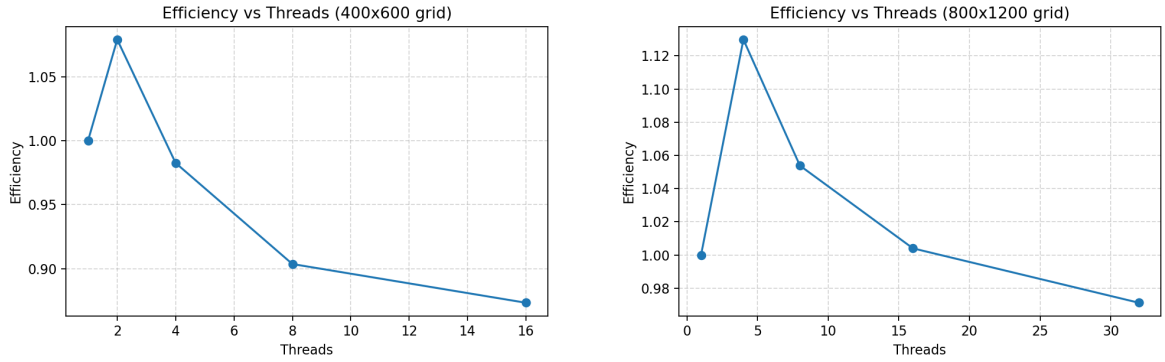


Рис. 7: Эффективность MPI+CUDA реализации в зависимости от числа процессов для сеток 400×600 и 800×1200.

4.4 Анализ результатов MPI+CUDA

По графикам видно, что ускорение гибридной реализации близко к линейному до 8–16 процессов, после чего эффективность постепенно снижается. Это связано с ростом относительной доли коммуникаций (обмен граничными значениями по MPI) и пересылок данных между CPU и GPU, а также с уменьшением локального объёма вычислений на каждый процесс при фиксированном размере задачи. Дополнительным фактором является то, что только операция Ap выполняется на ускорителе, тогда как предобуславливание и вычисление скалярных произведений остаются на CPU, что ограничивает достижимое ускорение в соответствии с законом Амдала.

4.5 Проверка корректности

Все параллельные версии (OpenMP, MPI, MPI+CUDA) дают одинаковое число итераций (1445 и 2984 для сеток 400×600 и 800×1200 соответственно), норма невязки полностью совпадает, отклонение между последовательным и параллельным решением составляет 10^{-8}–10^{-9}.

Для гибридной реализации MPI+CUDA дополнительно сравнивались результаты вычисления вектора Ap на небольшой тестовой сетке двумя способами: полностью на CPU и с использованием CUDA-ядра. Максимальное различие между компонентами оказалось на уровне машинной погрешности, что позволяет считать реализацию корректной.