Отчет по третьему заданию курса “Суперкомпьютерное моделирование и технологии” Численное решение задачи математической физики

Сюй Минчуань, группа 617, номер варианта: 8

12 декабря 2022 г.

Содержание

1 Постановка задачи

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

Рассматривается в прямоугольнике $\Pi = \{(x, y) : A_1 \leqslant x \leqslant A_2, B_1 \leqslant y \leqslant B_2\}$ дифференциальное уравнение Пуассона с потециалом

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

в котором оператор Лапласа

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

В частности, для **варианта 8** нужно восстановить функцию $u(x, y) = \sqrt{4 + xy}, \Pi = [0, 4] \times [0, 3]$ с коэффциентом $k(x, y) = 4 + x + y$ и потенциалом $q(x, y) = x + y$.

Для выделения единственного рещения уравнения понадобятся граничные условия. В своём варианте для левой (γ_L) и правой (γ_R) границы задано условие третьего типа:

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

а для верхней (γ_T) и нижней (γ_B) границы задано условие второго типа:

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

где n - единичная внешняя нормаль к границе. Так как в угловых точках области нормаль не определена, то краевое условие рассматривается лишь в тех точках границы, где есть нормаль.

Необходимо определить правую часть $F(x, y)$, сначала вычислить $-\Delta u$:

$$\begin{aligned}
-\Delta u &= -\frac{\partial}{\partial x}\left((4 + x + y)\frac{\partial}{\partial x}\sqrt{4 + xy}\right) - \frac{\partial}{\partial y}\left((4 + x + y)\frac{\partial}{\partial y}\sqrt{4 + xy}\right) \\
&= -\frac{\partial}{\partial x}\left(\frac{y(4 + x + y)}{2\sqrt{4 + xy}}\right) - \frac{\partial}{\partial y}\left(\frac{x(4 + x + y)}{2\sqrt{4 + xy}}\right) \\
&= -\frac{y \cdot 2\sqrt{4 + xy} - y^2(4 + x + y)(\sqrt{4 + xy})^{-1}}{4(4 + xy)} - \frac{x \cdot 2\sqrt{4 + xy} - x^2(4 + x + y)(\sqrt{4 + xy})^{-1}}{4(4 + xy)} \\
&= \frac{(4 + x + y)(x^2 + y^2) - 2(x + y)(4 + xy)}{4(4 + xy)^{3/2}}
\end{aligned}$$

Значит

$$F(x, y) = \frac{(4 + x + y)(x^2 + y^2) - 2(x + y)(4 + xy)}{4(4 + xy)^{3/2}} + (x + y)\sqrt{4 + xy}$$

а ещё граничные условия:

$$\gamma_L : \psi_L(x, y) = (4 + x + y) \cdot \left(\frac{-y}{2\sqrt{4 + xy}}\right) + \sqrt{4 + xy} = \{\text{при } x = 0\} = \frac{1}{4} \cdot (8 - 4y - y^2)$$

$$\gamma_R : \psi_R(x, y) = (4 + x + y) \cdot \left(\frac{y}{2\sqrt{4 + xy}}\right) + \sqrt{4 + xy} = \{\text{при } x = 4\} = \frac{y^2 + 16y + 8}{4\sqrt{1 + y}}$$

$$\gamma_T : \psi_T(x, y) = (4 + x + y) \cdot \left(\frac{x}{2\sqrt{4 + xy}}\right) = \{\text{при } y = 3\} = \frac{x(7 + x)\sqrt{4 + 3x}}{2(4 + 3x)}$$

$$\gamma_B : \psi_B(x, y) = (4 + x + y) \cdot \left(\frac{-x}{2\sqrt{4 + xy}}\right) = \{\text{при } y = 0\} = -\frac{1}{4}(x^2 + 4x)$$

2 Численный метод решения

2.1 Разностная схема решения задачи

Предлагается методом конечных разностей решить данную задачу. В рассматриваемой области П определяется равномерная сетка $\bar{\omega_h} = \bar{\omega_1} \times \bar{\omega_2}$, где

$$\bar{\omega_1} = \{x_i = A_1 + ih_1, i = \overline{0, M}\}, \bar{\omega_2} = \{y_j = B_1 + jh_2, j = \overline{0, N}\}$$

здесь $h_1 = (A_2 - A_1)/M, h_2 = (B_2 - B_1)/N$. Рассмотрим линейное пространство H функций, заданных на сетке $\bar{\omega_h}$. Обозначим через ω_{ij} значение сеточной функции $\omega \in H$ в узле сетки (x_i, y_i). Будем считать, что в пространстве H задано скалярное произведение и норма

$$[u, v] = \sum_{i=0}^{M} h_1 \sum_{j=0}^{N} h_2 \rho_{ij} u_{ij} v_{ij}, \quad \|u\|_E = \sqrt{[u, u]}$$

где весовая функция ρ_{ij} равна 1 когда ω_{ij} - внутренний узел; равна 1/2 когда граничный узел и равна 1/4 когда угловый узел. Уравнение во всех внутренних точках сетки аппроксимируется разностным уравнением

$$-\Delta_h \omega_{ij} + q_{ij}\omega_{ij} = F_{ij}, \quad i = \overline{1, M-1}, j = \overline{1, N-1}$$

в котором $F_{ij} = F(x_i, y_j), q_{ij} = q(x_i, y_j)$, а разностный оператор Лапласа $-\Delta_h \omega_{ij}$ можно записать в виде $\Delta_h \omega_{ij} = (a\omega_{\bar{x}})_{x,ij} + (b\omega_{\bar{y}})_{y,ij}$, если вводя специальные обозначения:

$$a_{ij} = k(x_i - 0.5h_1, y_j), \quad b_{ij} = k(x_i, y_j - 0.5h_2)$$
$$w_{x,ij} = \frac{w_{i+1,j} - w_{ij}}{h_1}, \quad w_{\bar{x},ij} = \frac{w_{i,j} - w_{i-1,j}}{h_1}$$
$$w_{y,ij} = \frac{w_{i,j+1} - w_{ij}}{h_2}, \quad w_{\bar{y},ij} = \frac{w_{i,j} - w_{i,j-1}}{h_2}$$

К аппроксимации граничных условий треьтего типа (и второго типа, который является частным случаем третьего) добавляются члены с точностью аппроксимации второго порядка (аппроксимация исходного дифференциального уравнения) с целью повышения точности аппроксимации.

Возвращаем к уравнению варианта, в котором на участках границы γ_R и γ_L заданы краевые условия третьего типа, на участках γ_B и γ_T заданы краевые условия Неймана. Значит, итоговая система уравнений, которую предостоить решить принимает вид:

$$-\Delta_h \omega_{ij} + q_{ij}\omega_{ij} = F_{ij}, i = \overline{1, M-1}, j = \overline{1, N-1}, \text{ (внут.)}$$
$$-(2/h_1)(a\omega_{\bar{x}})_{1j} + (q_{0j} + 2/h_1)\omega_{0j} - (b\omega_{\bar{y}})_{y,0j} = F_{0j} + (2/h_1)\psi_{0j}, j = \overline{1, N-1}, \text{ (лев.)}$$
$$(2/h_1)(a\omega_{\bar{x}})_{Mj} + (q_{Mj} + 2/h_1)\omega_{Mj} - (b\omega_{\bar{y}})_{y,Mj} = F_{Mj} + (2/h_1)\psi_{Mj}, j = \overline{1, N-1}, \text{ (прав.)}$$
$$-(2/h_2)(b\omega_{\bar{y}})_{i1} + q_{i0}\omega_{i0} - (a\omega_{\bar{x}})_{x,i0} = F_{i0} + (2/h_2)\psi_{i0}, i = \overline{1, M-1}, \text{ (ниж.)}$$
$$(2/h_2)(b\omega_{\bar{y}})_{iN} + q_{iN}\omega_{iN} - (a\omega_{\bar{x}})_{x,iN} = F_{iN} + (2/h_2)\psi_{iN}, i = \overline{1, M-1}, \text{ (верх.)}$$
$$(2/h_1)(a\omega_{\bar{x}})_{MN} + (2/h_2)(b\omega_{\bar{y}})_{MN} + (q_{MN} + 2/h_1)\omega_{MN} = F_{MN} + (2/h_1 + 2/h_2)\psi_{MN}, (\nearrow)$$
$$-(2/h_1)(a\omega_{\bar{x}})_{1N} + (2/h_2)(b\omega_{\bar{y}})_{0N} + (q_{0N} + 2/h_1)\omega_{0N} = F_{0N} + (2/h_1 + 2/h_2)\psi_{0N}, (\nwarrow)$$
$$-(2/h_1)(a\omega_{\bar{x}})_{10} - (2/h_2)(b\omega_{\bar{y}})_{01} + (q_{00} + 2/h_1)\omega_{00} = F_{00} + (2/h_1 + 2/h_2)\psi_{00}, (\swarrow)$$
$$(2/h_1)(a\omega_{\bar{x}})_{M0} - (2/h_2)(b\omega_{\bar{y}})_{M1} + (q_{M0} + 2/h_1)\omega_{M0} = F_{M0} + (2/h_1 + 2/h_2)\psi_{M0}, (\searrow)$$

Полученную СЛАУ можно представить в операторном виде $Aw = B$ (см. выше), где оператор A определяется левой частью линейных уравнений, функция B – правой частью.

2.2 Получение решения СЛАУ методом наименьших невязок

Метод наименьших невязок позволяет получить последовательность сетоных функций $\omega^{(k)}$ сходящуюся по норме пространства H к решению разностной схемы, т.е. $\|\omega - \omega^{(k)}\|_E \to 0, k \to +\infty$, стартуя из любого начального приближения. Одношаговая итерация вычисляется согласно равенству

$$\omega_{ij}^{(k+1)} = \omega_{ij}^{(k)} - \tau_{k+1} r_{ij}^{(k)}, \quad r^{(k)} = A\omega^{(k)} - B, \quad \tau_{k+1} = \frac{[Ar^{(k)}, r^{(k)}]}{\|Ar^{(k)}\|_E^2}$$

Критерий останова является

$$\|\omega^{(k+1)} - \omega_{(k)}\|_E \leqslant \varepsilon$$

Замечание В связи с ограничениям по времени выполнения программы на системе Polus (максимум 30 минут на один запуск) сделал следующие предположения:

1. Константу ε предполагалось взять $7e-6$.
2. Начальное приближение $w^{(0)}$ выбралось равно 2.5 во всех точках сетки.

2.3 Получение решения СЛАУ методом сопряженных градиентов

Метод наименьших невязок, конечно же, сойдется к истинному решению задачи, но он вычислительно трудоемко в плане числа итераций. Поэтому, стоит попробовать более продвинутые итерационные методы решения системы линейных уравнений.

Отметим, что описанные выше разностные схемы обладают самосопряженным и положительным определенным оператором A. Учитывая данную характкристику задачи, будем использовать *метод сопряженных градиентов* для сравнения. Этот метод может сходится за n итераций при любом начальном приближении, где n - размерность задачи. При заданной ε может раньше достичь желаемой точности решения задачи.

Общая схема алгоритма такая: вычисляем невязку $g^{(0)} = B - Aw^{(0)}$. Положим $d^{(0)} = g^{(0)}$. На каждой итерации обновляем значения векторов $w^{(k+1)}, g^{(k+1)}, d^{(k+1)}$.

$$w^{(k+1)} = w^{(k)} + \alpha_k d^{(k)}, \quad r^{(k+1)} = r^{(k)} - \alpha_k A d^{(k)}, \quad \alpha_k = \frac{[g^{(k)}, g^{(k)}]}{[d^{(k)}, Ad^{(k)}]}$$

$$d^{(k+1)} = g^{(k+1)} + \beta_k d^{(k)}, \quad \beta_k = \frac{[g^{(k+1)}, g^{(k+1)}]}{[g^{(k)}, g^{(k)}]}$$

Для консистентности сравнения результатов будем использовать $\varepsilon = 7e-6$ и начальное приближение $w^{(0)} = 2.5$ везде.

3 Описание программной реализации

Согласно постановке задания необходимо написать три кода программы: последовательный код, параллельный код с использованием MPI и гибридный код с использованием MPI и OpenMP. Приведем ниже их описания реализацией.

3.1 Последовательная реализация

Были реализованы следующие методы:

- **vector_diff**: выполняет операцию вычитания одного вектора их другого вектора.

- **_inner_product**: вычисляет скалярное произведение двух заданных векторов.
- **norm**: вычисляет евклидову норму заданного вектора.
- **init_B**: инициализировать вектор B системы уравнений.
- **A_vector_mult**: выполняет операцию матрично-векторного произведения.
- **solve**: реализация метода наименьших невязок.

3.2 Параллельная реализация: MPI

На основе последовательной программы реализована её параллельная версия. Вся логика реализации параллельной версии программы описана в классе **Process**, далее подробно опишем методы этого класса (значения атрибутов понятны из комментарий, некоторые из них также будут подробно объяснены в описании методов):

```
struct Process
{
    // constructor
    Process(int _M, int _N, double _eps);
    // solve the system using minimal discrepancies method
    void solve();
    // number of processors and processor rank
    int size, rank;

private:
    // global communicator
    MPI_Comm cart_comm;
    // MPI status
    MPI_Status status;

    // size of the block
    int size_x, size_y;
    // shifted i and j
    int i_x, j_y;
    // overall sizes
    int M, N;
    // tau global
    double tau_global;
    // steps of approximation
    double h1, h2;
    // solution accuracy
    double eps;

    // the number of processeds in each dimension
    int proc_number[2];
    // the coords of process in the topology
    int coords[2];

    // the send & recv buffers
    double *s_buf_up, *s_buf_down, *s_buf_left, *s_buf_right;
    double *r_buf_up, *r_buf_down, *r_buf_left, *r_buf_right;
    // the intermidate results
    double *Aw, *Ar, *B, *r, *w, *w_pr, *diff_w_and_w_pr;

    void create_communicator();
    void init_processor_config();
    void fill_data();
    void exchange_data(double *_w);
```

```
44      // one iteration for solving the linear system
45      double solve_iteration();
46  };
```

Листинг 1: Структура Process

- **Конструктор Process**
  Конструктор принимает значения размерности задачи M, N и требуемую точность решения ε из командной строки. После присваивания и вычисления шагов $h1, h2$ процесс получает свой **rank**, узнает число процессов **size**, и распределяет процессы по 2 размерности.

```
1  Process::Process(int _M, int _N, double _eps){
2      // get M, N and eps
3      M = _M, N = _N;
4      h1 = (double)(A2 - A1) / M;
5      h2 = (double)(B2 - B1) / N;
6      eps = _eps;
7
8      // get current process rank
9      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10
11     // get processes number
12     MPI_Comm_size(MPI_COMM_WORLD, &size);
13
14     // divide processes to 2 dims and store in ‘proc_number‘ array
15     MPI_Dims_create(size, 2, proc_number);
16 }
17
```

Листинг 2: Process::Process()

- **create_communicator**
  Этот метод генерирует двумерную декартовскую топологию без циклов. И процесс **rank** получает свою координацию в этой топологии.

```
1  void Process::create\_communicator(){
2
3      // boolean array to define periodicy of each dimension
4      int Periods[2] = {0, 0};
5
6      // (world communicator, num of dims, dim_size, periodicy for each
       dimention, no reorder, cart_comm)
7      MPI_Cart_create(MPI_COMM_WORLD, 2, proc_number, Periods, 0, &cart_comm);
8
9      // (cart_comm, the given rank, num of dims, the corresponding coords)
10     MPI_Cart_coords(cart_comm, rank, 2, coords);
11 }
12
```

Листинг 3: Process::create_communicator()

- **init_processor_config**
  Этот метод служит для конфигурации расчетной области процесса. После того как топология создана, процесс необходимо для своей расчетной области Π_{ij} определит размер по x, y, то есть **size_x, size_y**. Чтобы более равномерно распределяет нагрузку процессрам, общие правила, которые нужно соблюдать такие:

1. отношение количества узлов по переменным x и y в каждом домене принадлежало диапазону $[1/2, 2]$ - то есть для одного процесса расчетная область должна быть похожа на квадратную.
2. количество узлов по переменным x и y любых двух доменов отличалось не более, чем на единицу - то есть должно быть почти равномерно распределены размер по любым размерностям.

Поэтому, схема распределения такая:

1. Сначала всем процессам распределяет размер, который является *минимальным*. Например, если 11 точек распределяет 3 процессам по оси x, то сначала все получают $11/3 = 3$ точки.
2. Затем, распределяет оставшиеся точки *по одному процессу* до исчерпывания точек. Например, в предыдущем пункте осталось 2 точки, значит процесс номера 0 и 1 по оси x получают ещё одну точку, то есть в итоге 4 точки.

После распределения точек вычисляются глобальные индексы, с которых начинается расчетная область. Затем выводится на экран информация о распределения и выделяется память для буферов обмена данных боковых границ. Поскольку каждая граница нужна как получать данные из соседного процесса, так и пересылать данные соседнему процессу, необходимы 8 буферов. Также выделяется память для промежуточных массивов.

```
void Process::init_processor_config(){

    // calculate the size_x & size_y
    // need to garantee that each size w.r.t x and y has diff <= 1

    // there are M + 1 and N + 1 points
    size_x = (M + 1) / proc_number[0];
    size_y = (N + 1) / proc_number[1];

    // distribute the extra nodes to the first processes
    if (coords[0] < (M + 1) % proc_number[0]) size_x += 1;
    if (coords[1] < (N + 1) % proc_number[1]) size_y += 1;

    // calculate the i_x & j_y (real start nodes of this block)
    i_x = coords[0] * ((M + 1) / proc_number[0]) + min((M + 1) % proc_number
    [0], coords[0]);
    j_y = coords[1] * ((N + 1) / proc_number[1]) + min((N + 1) % proc_number
    [1], coords[1]);

    if (rank == 0){
        cout << "basic processors and task info:" << endl
        << "proc_dims = " << proc_number[0] << " " << proc_number[1]
        << " | M, N, h1, h2= " << M << " " << N << " " << h1 << " " << h2
        << endl;
    }

    cout << "rank " << rank
    << " | size_x, size_y= " << size_x << " " << size_y
    << " | i_x, i_y= " << i_x << " " << j_y
    << endl;

    // init send & recv buffers for every direction
    r_buf_up = new double [size_x];
    r_buf_down = new double [size_x];
```

```
33      r_buf_left = new double [size_y];
34      r_buf_right = new double [size_y];
35      s_buf_up = new double [size_x];
36      s_buf_down = new double [size_x];
37      s_buf_left = new double [size_y];
38      s_buf_right = new double [size_y];
39
40      // allocate memory
41      // padding = 1 to better perform A_vec_mult
42      Aw = new double [(size_x + 2) * (size_y + 2)];
43      Ar = new double [(size_x + 2) * (size_y + 2)];
44      B = new double [(size_x + 2) * (size_y + 2)];
45      r = new double [(size_x + 2) * (size_y + 2)];
46      w = new double [(size_x + 2) * (size_y + 2)];
47      w_pr = new double [(size_x + 2) * (size_y + 2)];
48      diff_w_and_w_pr = new double [(size_x + 2) * (size_y + 2)];
49  }
50
```

Листинг 4: Process::init_processor_config()

- **fill_data**
  Этот метод служит для заполнения данных начального приближения и правой части системы. В данном случае начальное приближение инициализируется значением 2.5, чтобы алгоритм сходился быстрее.

```
1   void Process::fill_data(){
2       // init w_pr
3       for(int i = 0; i <= size_x + 1; i++)
4           for(int j = 0; j <= size_y + 1; j++)
5               w_pr[i * (size_y + 2) + j] = 2.5;
6
7       // init w_0
8       for(int i = 0; i <= size_x + 1; i++)
9           for(int j = 0; j <= size_y + 1; j++)
10              w[i * (size_y + 2) + j] = 2.5;
11
12      init_B(B, M, N, size_x, size_y, i_x, j_y, h1, h2);
13  }
14
```

Листинг 5: Process::fill_data()

- **exchange_data**
  Этот метод осуществляет обмен данных боковых границ. Ниже приведен только одна часть кода из четырех (обмен данных по оси x на отрицательное(левое) направление). С помощью метода **MPI_Cart_shift** процесс получает **rank** соседних процессов, затем через методы **MPI_Sendrecv, MPI_Send, MPI_Recv** пересылаются и получаются данные. Перед Send необходимо оформить пересылаемый буфер, а после Recv необходимо сохранить полученные значения.

```
1   void Process::exchange_data(double *_w){
2       int rank_recv, rank_send;
3       int c, i, j;
4
5       // along x to the left -> dim = 0, disp = -1
6       // (communicator, direction, disp, source, dest)
7       MPI_Cart_shift(cart_comm, 0, -1, &rank_recv, &rank_send);
8       // the inner processes: send & recv
```

```
    if (coords[0] != 0 && coords[0] != proc_number[0] - 1){
        // generate send_buffer
        c = 0;
        for (j = 1; j <= size_y; j++){
            s_buf_left[c] = _w[1 * (size_y + 2) + j];
            c++;
        }
        /* (sendbuf, sendcount, sendtype, dest, sendtag,
            recvbuf, recvcount, recvtype, source, recvtag,
            comm, status) */
        MPI_Sendrecv(s_buf_left, size_y, MPI_DOUBLE, rank_send, TAG_X,
                     r_buf_right, size_y, MPI_DOUBLE, rank_recv, TAG_X,
                     cart_comm, &status);
        // store recv_buffer
        c = 0;
        for (j = 1; j <= size_y; j++){
            _w[(size_x + 1) * (size_y + 2) + j] = r_buf_right[c];
            c++;
        }
    }
    // the left process: recv
    else if (coords[0] == 0 && coords[0] != proc_number[0] - 1){
        MPI_Recv(r_buf_right, size_y, MPI_DOUBLE,
        rank_recv, TAG_X, cart_comm, &status);
        // store recv_buffer
        c = 0;
        for (j = 1; j <= size_y; j++){
            _w[(size_x + 1) * (size_y + 2) + j] = r_buf_right[c];
            c++;
        }
    }
    // the right process: send
    else if (coords[0] != 0 && coords[0] == proc_number[0] - 1){
        // generate send_buffer
        c = 0;
        for (j = 1; j <= size_y; j++){
            s_buf_left[c] = _w[1 * (size_y + 2) + j];
            c++;
        }
        MPI_Send(s_buf_left, size_y, MPI_DOUBLE,
        rank_send, TAG_X, cart_comm);
    }

    ......
    ......

```

Листинг 6: Process::exchange_data()

- **solve_iteration**
  Этот метод выполняет одну итерацию метода. В каждой итерации каждый процесс выполняет свою часть расчетов, необходимых для вычисления глобального значения τ. Зачем через **MPI_Allreduce** аккумулируются результаты расчетов, потому пересылаются суммированные значения числителя и знаменателя всем процессам и вычисляет итоговый tau каждый. На своей области обновляется w_{ij}. Возвращается локальная разность норм между итерациями. Заметим, что перед операцией матрично-векторного умножения **A_vec_mult** необходим обмен данных, чтобы при умножении на свой области можно использовать значения боковых границ соседних процессов.

```
double Process::solve_iteration(){
```

```
2
3       double diff_local;
4       double tau_numerator_global, tau_denominator_global;
5
6       //sync padding values
7       exchange_data(w);
8       A_vec_mult(Aw, w, M, N, size_x, size_y, i_x, j_y, h1, h2);
9       //sync padding values
10      exchange_data(Aw);
11      vector_diff(r, Aw, B, size_x, size_y);
12      A_vec_mult(Ar, r, M, N, size_x, size_y, i_x, j_y, h1, h2);
13
14      double tau_numerator_local = _inner_product(Ar, r, size_x, size_y, i_x,
        j_y, M, N, h1, h2);
15      double tau_denominator_local = _inner_product(Ar, Ar, size_x, size_y, i_x
        , j_y, M, N, h1, h2);
16
17      // (input data, output data, data size, data type, operation type,
        communicator)
18      MPI_Allreduce(&tau_numerator_local, &tau_numerator_global, 1, MPI_DOUBLE,
         MPI_SUM, MPI_COMM_WORLD);
19      MPI_Allreduce(&tau_denominator_local, &tau_denominator_global, 1,
        MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
20
21      tau_global = tau_numerator_global / tau_denominator_global;
22
23      // update points
24      for (int i = 1; i <= size_x; i++)
25          for (int j = 1; j <= size_y; j++)
26              w[i * (size_y + 2) + j] = w[i * (size_y + 2) + j] - tau_global *
        r[i * (size_y + 2) + j];
27
28      // calculate diff
29      vector_diff(diff_w_and_w_pr, w, w_pr, size_x, size_y);
30      diff_local = norm(diff_w_and_w_pr, size_x, size_y, i_x, j_y, M, N, h1, h2
        );
31
32      // store current w
33      for (int i = 1; i <= size_x; i++)
34          for (int j = 1; j <= size_y; j++)
35              w_pr[i * (size_y + 2) + j] = w[i * (size_y + 2) + j];
36
37      // the sum of squared elements
38      return diff_local * diff_local;
39  }
40
```

Листинг 7: Process::solve_iteration()

- **solve**
  Этот метод представляет собой итоговый solver задачи. Сначала создается коммуникатор, конфигурируется расчетная область, заполняются начальные данные, обмениваются данные при необходимости. В основном цикле вызывается метод **solve_iteration**, затем вычисляется общая разность между итерациями. Если разность меньше заданной требуемой точности, значит выход, сохраняем результаты на диск, освобождаем память и завершаем работу.

```
1   void Process::solve(){
2
3       create_communicator();
```

```
    init_processor_config();
    fill_data();
    //sync padding values for B
    exchange_data(B);

    double diff, diff_local;
    int iter = 0;

    if(rank == 0) cout << "Starting..." << endl;

    do{
        iter++;

        diff_local = solve_iteration();

        // calculate overall difference
        // (input data, output data, data size, data type, operation type, communicator)
        MPI_Allreduce(&diff_local, &diff, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        diff = sqrt(diff);

        if (rank == 0 && iter % PRINT_FREQ == 0) cout << "iter: " << iter << ", tau: " << tau_global << ", err_norm: " << diff << "\n";

    } while (diff > eps);

    // make barrier and wait for sync
    MPI_Barrier(MPI_COMM_WORLD);

    if (rank == 0){
        cout << "Finished !!!" << endl;
        cout << "total_iter: " << iter << ", final_tau: "<< tau_global <<", final_err_norm: " << diff << "\n";
        cout << "Record the results.." << "\n";
    }

    // save results to disk and free memory.
    ......
    ......

```

Листинг 8: Process::solve()

3.3 Параллельная реализация: MPI & OpenMP

На основе MPI-реализации были добавлены OpenMP директивы к циклам для осуществления запуска программы с использованием нитей. Добавлены директивы к циклам следующих функций или частям программы:

- vector_diff() - разность двух веткоров
- _inner_product() - скалярное произведение двух векторов
- init_B() - инициализация значений веткора B
- A_vec_mult() - MatVec умножение
- обновление значений вектора $w^{(k)}$ в итерации.

Есть два типа добавленные OpenMP диективы:

1. #pragma omp parallel for default(shared) private(...) schedule(dynamic)
2. #pragma omp parallel for default(shared) private(...) schedule(dynamic) reduction (+:...)

Первый директив распараллеливает цикл for нитям. Некоторые из переменных, которые живут в области распараллеливания, были объявлены приватными (это индексы по которым осуществляются цикл и индексирование массивов), чтобы сделать нити работают независимо под своими выделенными подзадачами вычисления. Используется динамичекое расписание распределедения по нитям. Второй служит для операции редукции в вычислении скалярного произведения.

3.4 Реализация метода сопряженных градиентов

В общем структура такая же, только переписал цикл в **Process::solve** на схему метода сопряженных градиентов, за то получил большой выигрыш :). Отмечу, что реализованы только последовательный и MPI варианты, и просто хотел понял, насколько CG может получить прирост скорости сходимости по сравнению с предложенным итерационным методом.

4 Анализ результатов расчетов на системе Polus

На основе написанной программы провел разные эксперименты, чтобы исследовать масштабируемость реализованных программ. Программы запустились для различного числа MPI-процессов и различных размерностей задачи. Заполнил таблицу с полученными результатами. Из ходя из полученных результатов, можно сделать такие **выводы:**

- Как видно из таблицы №1, реализованная MPI версия программы хорошо масштабируется: при увеличении числа процессов время выполнения расчета сильно уменьшается, ускорение заметное. Эффективность распараллеливания (отношение ускорения к числу процессов) чуть уменьшается с увеличением числа процессов. Это объясняется уменьшением расчетной области каждого процесса, и увеличением накладных расходов между ними для тех операций, которые требуют коммуникации, например **MPI_Allreduce**.
- Для задачи с большей размерности также заметно ускорение. Но есть интересное наблюдение в том, что время расчетов для задачи размера 500×1000 меньше времени для размера 500×500. Правильность расчетов было проверена сравнением полученного численного решения с точным решением. Как оказалось, большая размерность необязательно требует больше числа итераций, так как структура матрицы большой системы может строиться в пользу итерационного метода. Эффективность распараллеливания меньше чем 500×500 варианта, что естественно поскольку размеры массивов при коммуникации между процессами растут, из-за это расплачивается больше времени.
- Из таблицы №2 видно, что использование директив OpenMP позволяет ускорить выполнение программы в столько раз, сколько ожидалось (то есть, почти прекрасное ускорение, если ограничим рассмотрением только числа процессов $<= 4$). При увеличении числа процессов на 8 этот показатель уменьшается, так как общий объем работы для каждой нити уменьшается и было больше затрачена на накладные расходы. На больших сетках ускорение более заметно по аналогичной причине.

- Также заметим, что если сравниваем таблицу №1 и №2, при одинаковым числе worker'ов (тут под worker'ом понимается либо процесс MPI, либо нить) MPI+OpenMP программа работает быстрее чем MPI программа. Например, время выполнения программы для задачи 500×500 с 8 процессами, каждый с 4 нити, есть 168.822 секунд, а время выполнения программы с 32 процессами составляет 198.376 секунд. Это объясняется тем, что в MPI-OpenMP варианте между нитями нет практически обмена информаци, каждый выполняет свою подзадачу, возможна только операция редукции в некоторых местах (скалярное произведение), поэтому накладные расходы меньше.

- Наконец, для интереса реализовал и запустил программу с методом сопряженных градиентов (только MPI версия). Из таблицы №1 можно увидеть большой выигрыш в плане времени выполнения. В случае CG $500 * 1000$ требует больше времени(больше итераций) за счет увеличения размерности.

Число процессов MPI	Число точек сетки M × N	Время(s) решения исх. метода	Время(s) решения метода CG	Ускорение исх. метода
4	500 × 500	1198.100	16.868	1
8	500 × 500	645.545	9.058	1.856
16	500 × 500	356.502	5.027	3.361
32	500 × 500	198.376	3.928	6.040
4	500 × 1000	877.178	58.074	1
8	500 × 1000	499.297	30.015	1.757
16	500 × 1000	287.097	16.691	3.055
32	500 × 1000	158.011	12.328	5.551

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

Число процессов MPI	Количество OMP-нитей в процессе	Число точек сетки M × N	Время решения (s)	Ускорение
1	4	500 × 500	1073.640	1
2	4	500 × 500	531.197	2.021
4	4	500 × 500	268.924	3.992
8	4	500 × 500	168.822	6.35
1	4	500 × 1000	791.297	1
2	4	500 × 1000	417.500	1.895
4	4	500 × 1000	198.427	3.988
8	4	500 × 1000	104.533	7.570

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

5 Визуализация полученного численного решения

Ниже приведены рисунки графика точного решения (первая картинка) и приближенного решения (вторая картинка), полученного в результате работы программы на сетке 1000×1000.

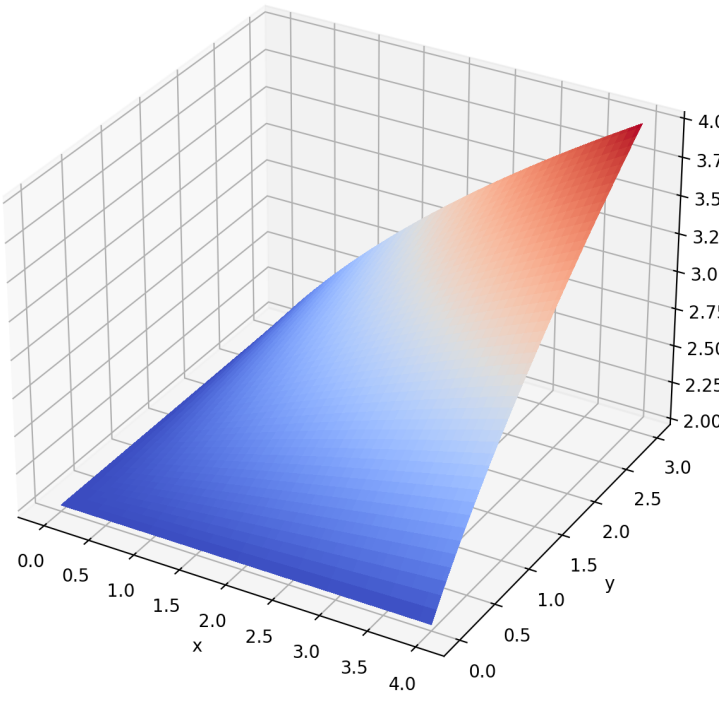

Рис. 1: График точного решения $u(x, y) = \sqrt{4 + xy}$

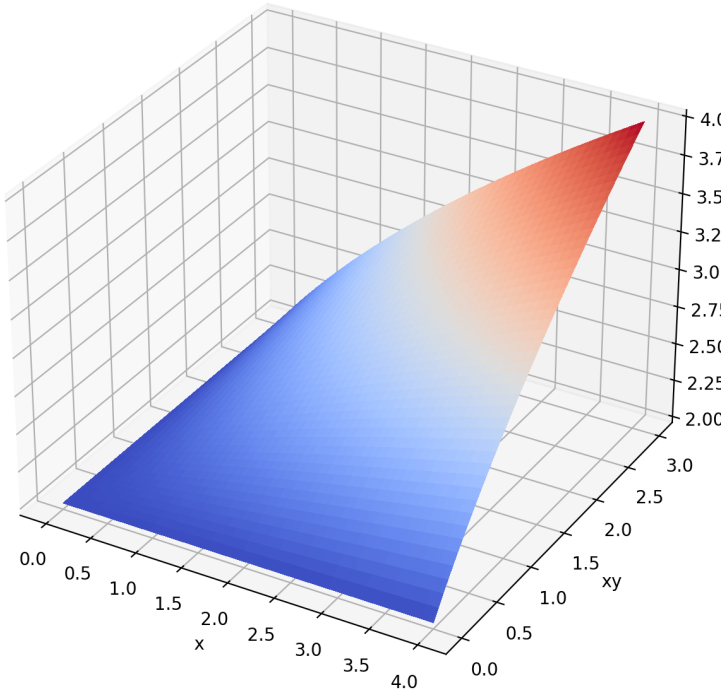

Рис. 2: График приближенного решения, полученного на сетке 1000×1000