Лекция 7
Параллельные сеточные вычисления

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com
WWW: www.mkurnosov.net

Курс «Параллельные вычислительные технологии»
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)
Осенний семестр, 2016

Расчет стационарного распределения тепла

- С течением времени в теле устанавливается некоторое не зависящее от времени распределение температуры (тепловое состояние выходит на стационарный режим)
- Распределение температуры в таком случае описывается уравнением теплопроводности
- Стационарное двумерное уравнение Лапласа (Laplace equation)

$$\Delta \boldsymbol{U} = \frac{\boldsymbol{d^2 U}}{\boldsymbol{dx^2}} + \frac{\boldsymbol{d^2 U}}{\boldsymbol{dy^2}} = \mathbf{0} \qquad (1)$$

- Функция $U(x, y)$ – неизвестный потенциал (теплота)
- **Задано** уравнение (1), значения функции $U(x, y)$ на границах расчетной 2D-области
- **Требуется** найти значение функции $U(x, y)$ во внутренних точках расчетной 2D-области

Расчет стационарного распределения тепла

- **Найти** решение стационарного двумерного уравнения Лапласа (Laplace equation)

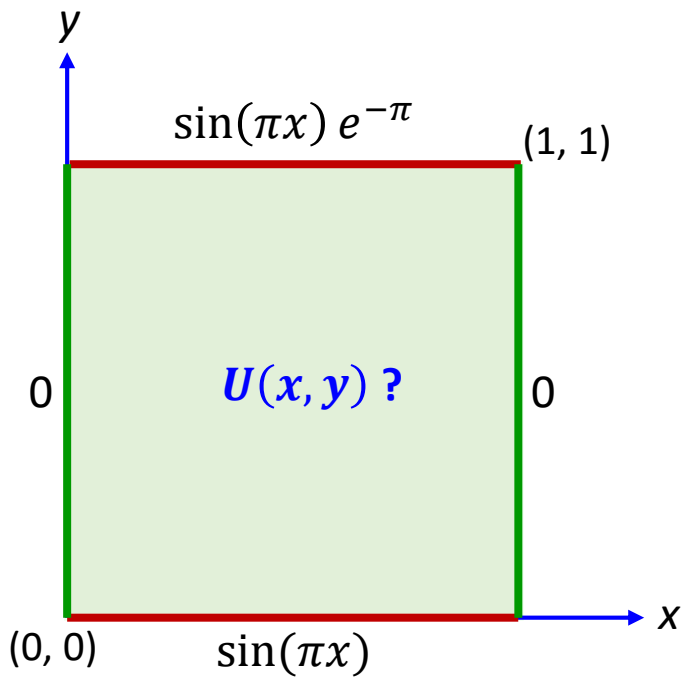
$$\frac{d^2U}{dx^2} + \frac{d^2U}{dy^2} = 0$$

- Расчётная область (domain) – квадрат [0, 1] x [0, 1]

- Граничные условия (boundary conditions):
  - $U(x, 0) = \sin(\pi x)$
  - $U(x, 1) = \sin(\pi x)\, e^{-\pi}$
  - $U(0, y) = U(1, y) = 0$

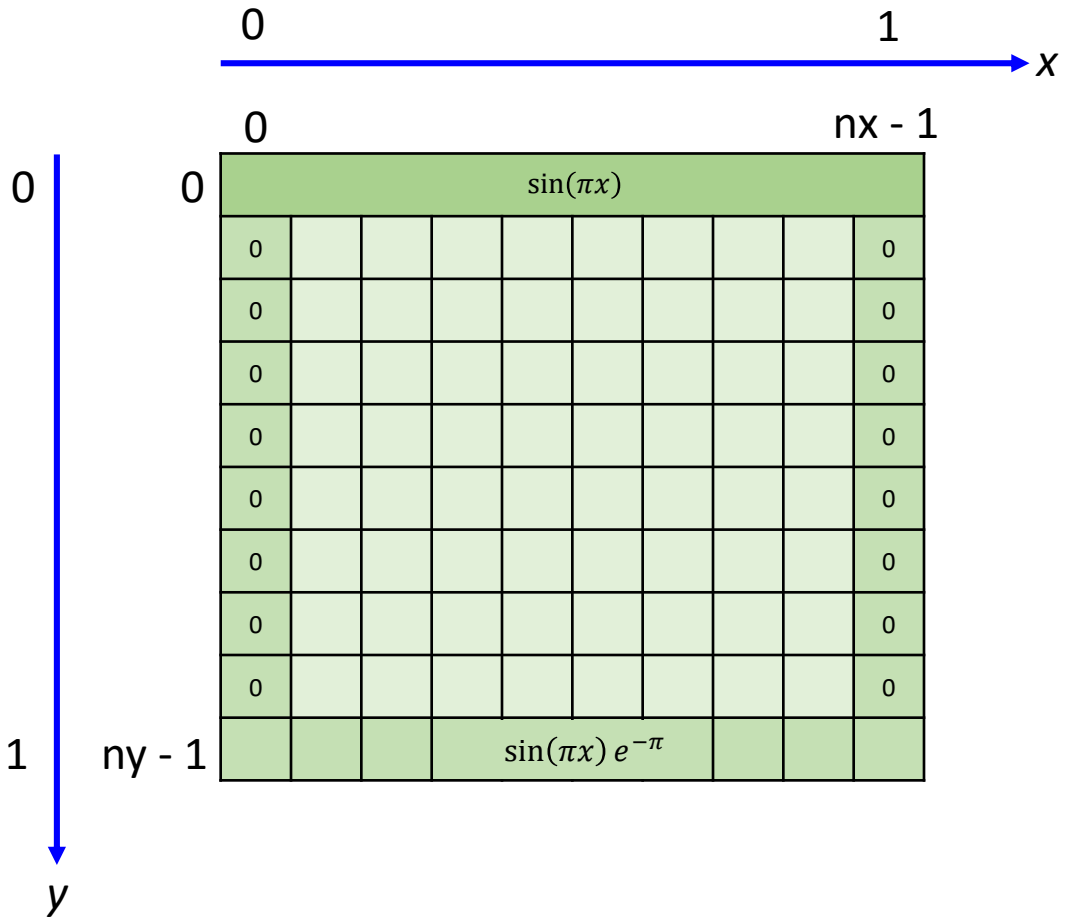
- Для данной задачи известно аналитическое решение

$$U(x, y) = \sin(\pi x)\, e^{-\pi y}$$



Дискретизация расчетной области

- Расчетная область [0, 1] x [0, 1] покрывается прямоугольной сеткой с постоянным шагом: nx точек по оси OX и ny точек по оси OY



- Расчетная сетка – массив [ny, nx] чисел (температура)
- Переход от индекса ячейки [i, j] к координатам в области [0, 1] x [0, 1]:

```
x = j * 1.0 / (nx – 1.0)
y = i * 1.0 / (ny – 1.0)
```

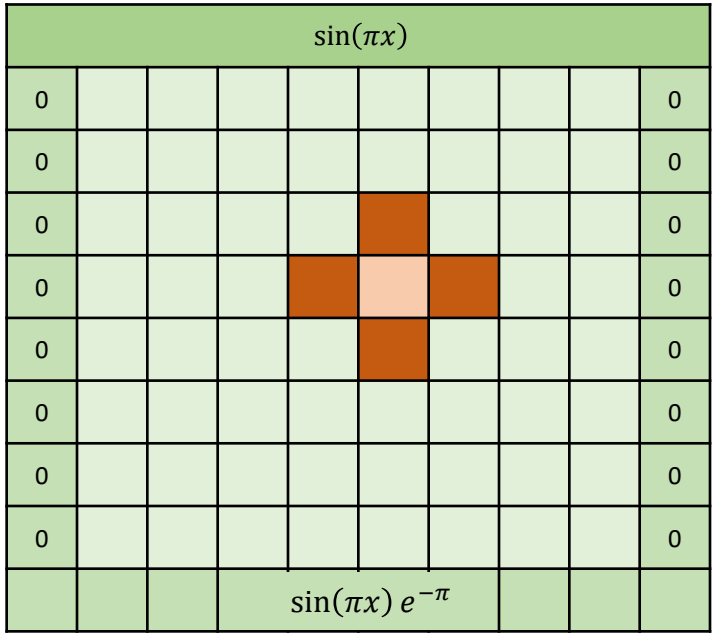
Разностная аппроксимация оператора Лапласа

- Вторые производные аппроксимируются на расчетной сетке разностным уравнением с применением четырехточечного шаблона

$$\Delta U = \frac{d^2U}{dx^2} + \frac{d^2U}{dy^2} = 0$$

- Новое значение в каждой точке сетки равно среднему из предыдущих значений четырех ее соседних точек (схема «крест»)

```
grid_new[i ,j] = (grid[i - 1, j] + grid[i, j + 1] +
                  grid[i + 1, j] + grid[i, j - 1]) / 4
```



Метод последовательных итераций Якоби (Jacobi)

- Вычисляем новое значение в каждой точке [i, j] сетки – среднее из предыдущих значений четырех ее соседних точек (схема «крест»), результат записываем в новую сетку (массив)
- На следующей итерации текущей делаем новую сетку предыдущей итерации
- Заканчиваем итерационный процесс, если разность между каждым текущим и предыдущим значениями по модулю не больше EPSILON

Метод последовательных итераций Якоби (Jacobi)

```
#define EPS 0.001
#define PI 3.14159265358979323846
#define IND(i, j) ((i) * nx + (j))

int main(int argc, char *argv[])
{
    int rows = 100;
    int cols = 100;
    int ny = rows;
    int nx = cols;
    double *local_grid = xcalloc(ny * nx, sizeof(*local_grid));
    double *local_newgrid = xcalloc(ny * nx, sizeof(*local_newgrid));

    double dx = 1.0 / (nx - 1.0);
    // Initialize top border: u(x, 0) = sin(pi * x)
    for (int j = 0; j < nx; j++) {
        int ind = IND(0, j);
        local_newgrid[ind] = local_grid[ind] = sin(PI * dx * j);
    }
    // Initialize bottom border: u(x, 1) = sin(pi * x) * exp(-pi)
    for (int j = 0; j < nx; j++) {
        int ind = IND(ny - 1, j);
        local_newgrid[ind] = local_grid[ind] = sin(PI * dx * j) * exp(-PI);
    }
```

Метод последовательных итераций Якоби (Jacobi)

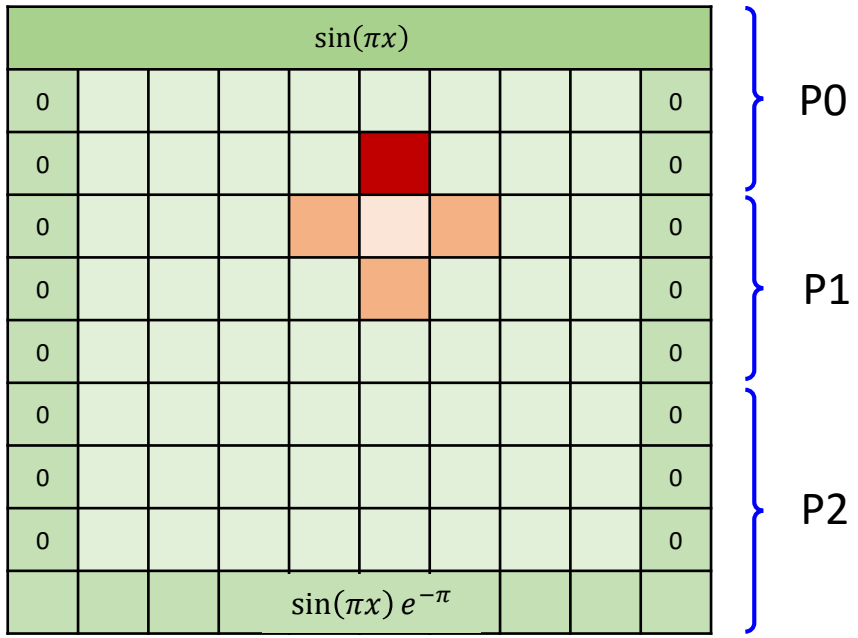
```
int niters = 0;
for (;;) {
    niters++;
    for (int i = 1; i < ny - 1; i++) {        // Update interior points
        for (int j = 1; j < nx - 1; j++) {
            local_newgrid[IND(i, j)] =
                (local_grid[IND(i - 1, j)] + local_grid[IND(i + 1, j)] +
                 local_grid[IND(i, j - 1)] + local_grid[IND(i, j + 1)]) * 0.25;
        }
    }
    // Check termination condition
    double maxdiff = 0;
    for (int i = 1; i < ny - 1; i++) {
        for (int j = 1; j < nx - 1; j++) {
            int ind = IND(i, j);
            maxdiff = fmax(maxdiff, fabs(local_grid[ind] - local_newgrid[ind]));
        }
    }
    // Swap grids (after termination local_grid will contain result)
    double *p = local_grid;
    local_grid = local_newgrid;
    local_newgrid = p;

    if (maxdiff < EPS)
        break;
}
ttotal += wtime();
```

Метод последовательных итераций Якоби (Jacobi)

```
    printf("# Heat 2D (serial): grid: rows %d, cols %d\n", rows, cols);
    printf("# niters %d, total time %.6f\n", niters, ttotal);

    // Save grid
    if (filename) {
        FILE *fout = fopen(filename, "w");
        if (!fout) {
            perror("Can't open file");
            exit(EXIT_FAILURE);
        }
        for (int i = 0; i < ny; i++) {
            for (int j = 0; j < nx; j++)
                fprintf(fout, "%.4f ", local_grid[IND(i, j)]);
            fprintf(fout, "\n");
        }
        fclose(fout);
    }

    return 0;
}
```

Параллельная версия метода Якоби (1D)

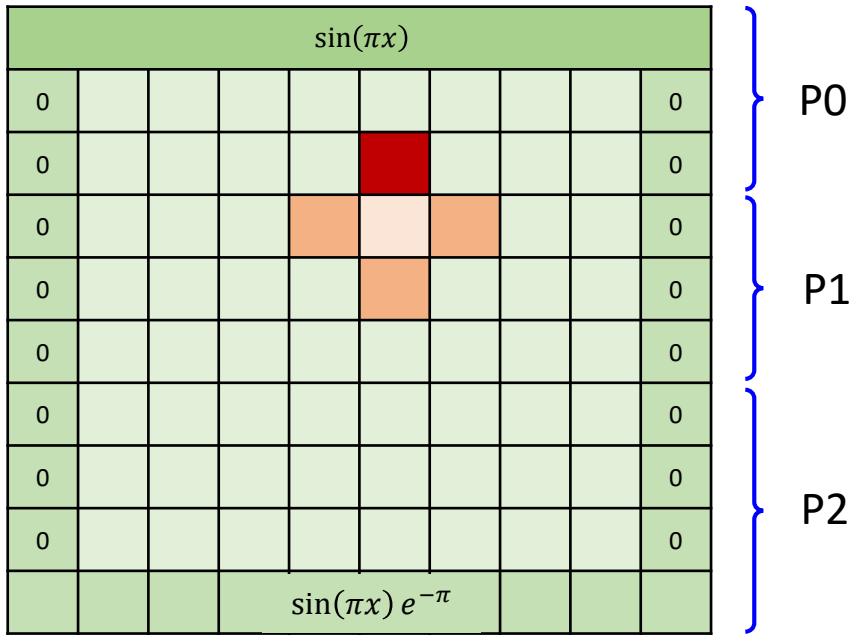
- **Одномерная (1D) декомпозиция расчётной области на горизонтальные полосы**
- Каждому процессу назначается ny / p строк расчетной сетки
- Сетка хранится в памяти в распределенном виде
- **Проблема** – для расчета значений некоторых точек требуются данные соседних полос, которые находятся в памяти других процессов

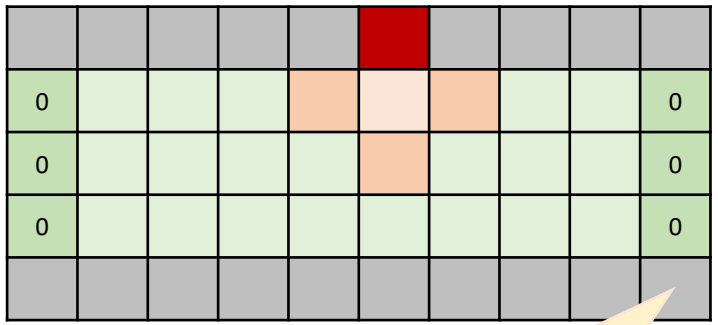


- На каждой итерации перед вычислением значений в точках обмениваемся пограничными строками с соседними процессами
- Как хранить эти строки?
- Как выполнять обмены?

Параллельная версия метода Якоби (1D)

- **Одномерная (1D) декомпозиция расчётной области на горизонтальные полосы**
- Каждому процессу назначается ny / p строк расчетной сетки
- Сетка хранится в памяти в распределенном виде
- **Проблема** – для расчета значений некоторых точек требуются данные соседних полос, которые находятся в памяти других процессов





Теневые ячейки (halo, ghost cells) для хранения строк соседних процессов

Параллельная версия метода Якоби (1D)

```
#define NELEMS(x) (sizeof((x)) / sizeof((x)[0]))
#define IND(i, j) ((i) * cols + (j))

int get_block_size(int n, int rank, int nprocs)
{
    int s = n / nprocs;
    if (n % nprocs > rank) s++;
    return s;
}

int main(int argc, char *argv[])
{
    int commsize, rank;
    MPI_Init(&argc, &argv);
    double ttotal = -MPI_Wtime();
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int rows, cols; // Broadcast command line arguments
    if (rank == 0) {
        rows = (argc > 1) ? atoi(argv[1]) : commsize * 100;
        cols = (argc > 2) ? atoi(argv[2]) : 100;
        if (rows < commsize) {
            fprintf(stderr, "Number of rows %d less then number of processes %d\n", rows, commsize);
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }
        int args[2] = {rows, cols};
        MPI_Bcast(&args, NELEMS(args), MPI_INT, 0, MPI_COMM_WORLD);
    } else {
        int args[2];
        MPI_Bcast(&args, NELEMS(args), MPI_INT, 0, MPI_COMM_WORLD);
        rows = args[0];
        cols = args[1];
    }
```

Параллельная версия метода Якоби (1D)

```
// Allocate memory for local 1D subgrids with 2 halo rows [0..ny + 1][0..cols - 1]
int ny = get_block_size(rows, rank, commsize);
double *local_grid = xcalloc((ny + 2) * cols, sizeof(*local_grid));
double *local_newgrid = xcalloc((ny + 2) * cols, sizeof(*local_newgrid));

// Fill boundary points:
//   - left and right borders are zero filled
//   - top border: u(x, 0) = sin(pi * x)
//   - bottom border: u(x, 1) = sin(pi * x) * exp(-pi)
double dx = 1.0 / (cols - 1.0);
if (rank == 0) {
    // Initialize top border: u(x, 0) = sin(pi * x)
    for (int j = 0; j < cols; j++) {
        int ind = IND(0, j);
        local_newgrid[ind] = local_grid[ind] = sin(PI * dx * j);
    }
}
if (rank == commsize - 1) {
    // Initialize bottom border: u(x, 1) = sin(pi * x) * exp(-pi)
    for (int j = 0; j < cols; j++) {
        int ind = IND(ny + 1, j);
        local_newgrid[ind] = local_grid[ind] = sin(PI * dx * j) * exp(-PI);
    }
}
```

Параллельная версия метода Якоби (1D)

```
// Neighbours
int top = (rank > 0) ? rank - 1 : MPI_PROC_NULL;
int bottom = (rank < commsize - 1) ? rank + 1 : MPI_PROC_NULL;

// Top and bottom borders type
MPI_Datatype row;
MPI_Type_contiguous(cols, MPI_DOUBLE, &row);
MPI_Type_commit(&row);

MPI_Request reqs[4];
double thalo = 0;
double treduce = 0;

int niters = 0;
for (;;) {
    niters++;

    // Update interior points
    for (int i = 1; i <= ny; i++) {
        for (int j = 1; j < cols - 1; j++) {
            local_newgrid[IND(i, j)] =
                (local_grid[IND(i - 1, j)] + local_grid[IND(i + 1, j)] +
                 local_grid[IND(i, j - 1)] + local_grid[IND(i, j + 1)]) * 0.25;
        }
    }
```

Параллельная версия метода Якоби (1D)

```
    // Check termination condition
    double maxdiff = 0;
    for (int i = 1; i <= ny; i++) {
        for (int j = 1; j < cols - 1; j++) {
            int ind = IND(i, j);
            maxdiff = fmax(maxdiff, fabs(local_grid[ind] - local_newgrid[ind]));
        }
    }
    // Swap grids (after termination local_grid will contain result)
    double *p = local_grid;
    local_grid = local_newgrid;
    local_newgrid = p;

    treduce -= MPI_Wtime();
    MPI_Allreduce(MPI_IN_PLACE, &maxdiff, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
    treduce += MPI_Wtime();
    if (maxdiff < EPS)
        break;

    // Halo exchange: T = 4 * (a + b * cols)
    thalo -= MPI_Wtime();
    MPI_Irecv(&local_grid[IND(0, 0)], 1, row, top, 0, MPI_COMM_WORLD, &reqs[0]);          // top
    MPI_Irecv(&local_grid[IND(ny + 1, 0)], 1, row, bottom, 0, MPI_COMM_WORLD, &reqs[1]); // bottom
    MPI_Isend(&local_grid[IND(1, 0)], 1, row, top, 0, MPI_COMM_WORLD, &reqs[2]);          // top
    MPI_Isend(&local_grid[IND(ny, 0)], 1, row, bottom, 0, MPI_COMM_WORLD, &reqs[3]);     // bottom
    MPI_Waitall(4, reqs, MPI_STATUS_IGNORE);
    thalo += MPI_Wtime();
}
```

Параллельная версия метода Якоби (1D)

```
    MPI_Type_free(&row);

    free(local_newgrid);
    free(local_grid);

    ttotal += MPI_Wtime();

    if (rank == 0)
        printf("# Heat 1D (mpi): grid: rows %d, cols %d, procs %d\n", rows, cols, commsize);

    int namelen;
    char procname[MPI_MAX_PROCESSOR_NAME];
    MPI_Get_processor_name(procname, &namelen);
    printf("# P %4d on %s: grid ny %d nx %d, total %.6f, mpi %.6f (%.2f) = allred %.6f (%.2f) + halo %.6f (%.2f)\n",
           rank, procname, ny, cols, ttotal, treduce + thalo, (treduce + thalo) / ttotal,
           treduce, treduce / (treduce + thalo), thalo, thalo / (treduce + thalo));

    double prof[3] = {ttotal, treduce, thalo};
    if (rank == 0) {
        MPI_Reduce(MPI_IN_PLACE, prof, NELEMS(prof), MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
        printf("# procs %d : grid %d %d : niters %d : total time %.6f : mpi time %.6f : allred %.6f : halo %.6f\n",
               commsize, rows, cols, niters, prof[0], prof[1] + prof[2], prof[1], prof[2]);
    } else {
        MPI_Reduce(prof, NULL, NELEMS(prof), MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

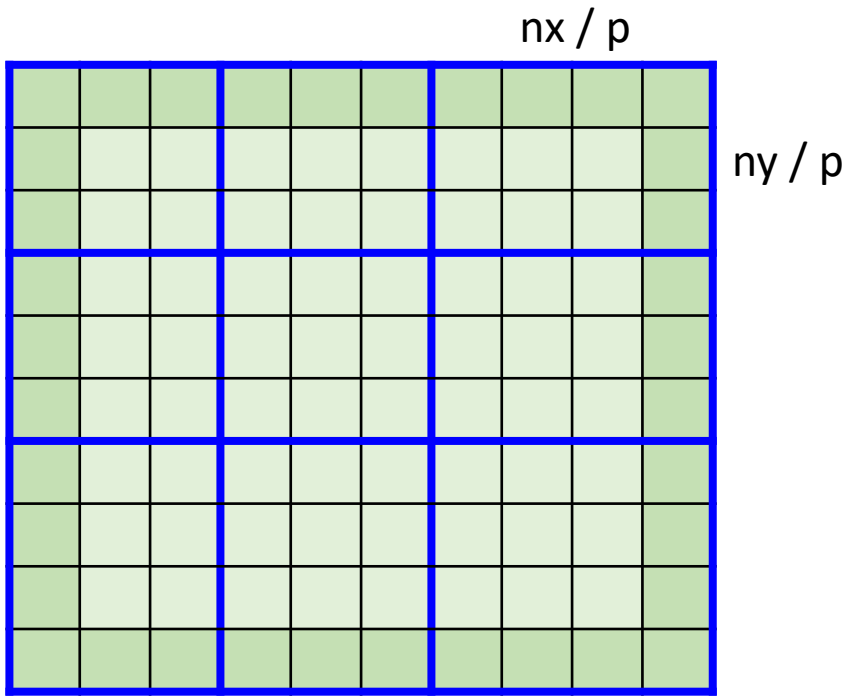
Параллельная версия метода Якоби (1D)

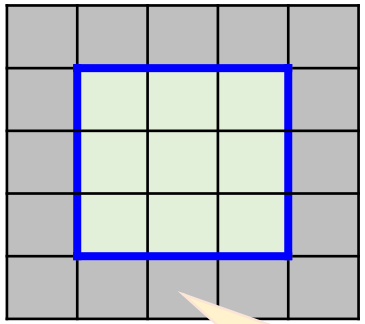
- Анализ времени выполнения информационных обменов при одномерной декомпозиции
- Время передачи сообщения размером *m* байт: $t(m) = \alpha + \beta m$
- Пусть сетка квадратная и содержит *n* строк и столбцов
- На каждой итерации выполняется два send и два recv:

$$t_{iter} = 4\alpha + 4n\beta$$

Параллельная версия метода Якоби (2D)

- **Двумерная (2D) декомпозиция расчётной области на прямоугольные области**
- Каждому процессу назначается подмассив [ny / p, nx / p] строк расчетной сетки
- Сетка хранится в памяти в распределенном виде
- **Проблема** – для расчета значений некоторых точек требуются данные соседних полос, которые находятся в памяти других процессов





Теневые ячейки (halo, ghost cells) для хранения значений из соседних процессов

Параллельная версия метода Якоби (2D)

```
#define EPS 0.001
#define PI 3.14159265358979323846
#define NELEMS(x) (sizeof((x)) / sizeof((x)[0]))
#define IND(i, j) ((i) * (nx + 2) + (j))

int main(int argc, char *argv[])
{
    int commsize, rank;
    MPI_Init(&argc, &argv);
    double ttotal = -MPI_Wtime();
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Create 2D grid of processes: commsize = px * py
    MPI_Comm cartcomm;
    int dims[2] = {0, 0}, periodic[2] = {0, 0};
    MPI_Dims_create(commsize, 2, dims);
    int px = dims[0];
    int py = dims[1];

    if (px < 2 || py < 2) {
        fprintf(stderr, "Invalid number of processes %d: px %d and py %d must be greater than 1\n",
                commsize, px, py);
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periodic, 0, &cartcomm);
    int coords[2];
    MPI_Cart_coords(cartcomm, rank, 2, coords);
    int rankx = coords[0];
    int ranky = coords[1];
```

Параллельная версия метода Якоби (2D)

```
int rows, cols;

// Broadcast command line arguments
if (rank == 0) {
    rows = (argc > 1) ? atoi(argv[1]) : py * 100;
    cols = (argc > 2) ? atoi(argv[2]) : px * 100;

    if (rows < py) {
        fprintf(stderr, "Number of rows %d less then number of py processes %d\n", rows, py);
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }
    if (cols < px) {
        fprintf(stderr, "Number of cols %d less then number of px processes %d\n", cols, px);
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    int args[2] = {rows, cols};
    MPI_Bcast(&args, NELEMS(args), MPI_INT, 0, MPI_COMM_WORLD);
} else {
    int args[2];
    MPI_Bcast(&args, NELEMS(args), MPI_INT, 0, MPI_COMM_WORLD);
    rows = args[0];
    cols = args[1];
}
```

Параллельная версия метода Якоби (2D)

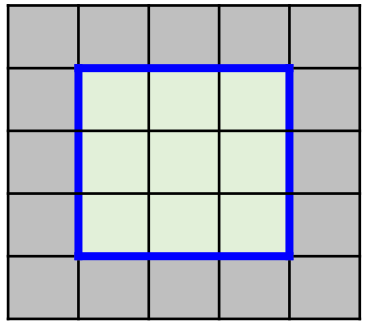
```
// Allocate memory for local 2D subgrids with halo cells [0..ny + 1][0..nx + 1]
int ny = get_block_size(rows, ranky, py);
int nx = get_block_size(cols, rankx, px);
double *local_grid = xcalloc((ny + 2) * (nx + 2), sizeof(*local_grid));
double *local_newgrid = xcalloc((ny + 2) * (nx + 2), sizeof(*local_newgrid));

// Fill boundary points:
//   - left and right borders are zero filled
//   - top border: u(x, 0) = sin(pi * x)
//   - bottom border: u(x, 1) = sin(pi * x) * exp(-pi)
double dx = 1.0 / (cols - 1.0);
int sj = get_sum_of_prev_blocks(cols, rankx, px);
if (ranky == 0) {
    // Initialize top border: u(x, 0) = sin(pi * x)
    for (int j = 1; j <= nx; j++) {
        // Translate col index to x coord in [0, 1]
        double x = dx * (sj + j - 1);
        int ind = IND(0, j);
        local_newgrid[ind] = local_grid[ind] = sin(PI * x);
    }
}
if (ranky == py - 1) {
    // Initialize bottom border: u(x, 1) = sin(pi * x) * exp(-pi)
    for (int j = 1; j <= nx; j++) {
        // Translate col index to x coord in [0, 1]
        double x = dx * (sj + j - 1);
        int ind = IND(ny + 1, j);
        local_newgrid[ind] = local_grid[ind] = sin(PI * x) * exp(-PI);
    }
}
```

Параллельная версия метода Якоби (2D)

```
int get_block_size(int n, int rank, int nprocs)
{
    int s = n / nprocs;
    if (n % nprocs > rank)
        s++;
    return s;
}

int get_sum_of_prev_blocks(int n, int rank, int nprocs)
{
    int rem = n % nprocs;
    return n / nprocs * rank + ((rank >= rem) ? rem : rank);
}
```

Параллельная версия метода Якоби (2D)

```
// Neighbours
int left, right, top, bottom;
MPI_Cart_shift(cartcomm, 0, 1, &left, &right);
MPI_Cart_shift(cartcomm, 1, 1, &top, &bottom);

// Left and right borders type
MPI_Datatype col;
MPI_Type_vector(ny, 1, nx + 2, MPI_DOUBLE, &col);
MPI_Type_commit(&col);

// Top and bottom borders type
MPI_Datatype row;
MPI_Type_contiguous(nx, MPI_DOUBLE, &row);
MPI_Type_commit(&row);

MPI_Request reqs[8];
double thalo = 0;
double treduce = 0;
```

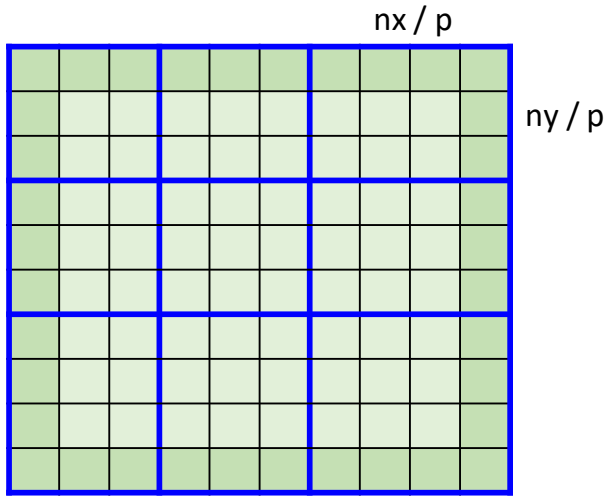


Параллельная версия метода Якоби (2D)

```
int niters = 0;
for (;;) {
    niters++;

    // Update interior points
    for (int i = 1; i <= ny; i++) {
        for (int j = 1; j <= nx; j++) {
            local_newgrid[IND(i, j)] =
                (local_grid[IND(i - 1, j)] + local_grid[IND(i + 1, j)] +
                 local_grid[IND(i, j - 1)] + local_grid[IND(i, j + 1)]) * 0.25;
        }
    }

    // Check termination condition
    double maxdiff = 0;
    for (int i = 1; i <= ny; i++) {
        for (int j = 1; j <= nx; j++) {
            int ind = IND(i, j);
            maxdiff = fmax(maxdiff, fabs(local_grid[ind] - local_newgrid[ind]));
        }
    }
    // Swap grids (after termination local_grid will contain result)
    double *p = local_grid;
    local_grid = local_newgrid;
    local_newgrid = p;

    treduce -= MPI_Wtime();
    MPI_Allreduce(MPI_IN_PLACE, &maxdiff, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
    treduce += MPI_Wtime();
    if (maxdiff < EPS)
        break;
```

Параллельная версия метода Якоби (2D)

```
    // Halo exchange: T = 4 * (a + b * (rows / py)) + 4 * (a + b * (cols / px))
    thalo -= MPI_Wtime();
    MPI_Irecv(&local_grid[IND(0, 1)], 1, row, top, 0, cartcomm, &reqs[0]);          // top
    MPI_Irecv(&local_grid[IND(ny + 1, 1)], 1, row, bottom, 0, cartcomm, &reqs[1]); // bottom
    MPI_Irecv(&local_grid[IND(1, 0)], 1, col, left, 0, cartcomm, &reqs[2]);         // left
    MPI_Irecv(&local_grid[IND(1, nx + 1)], 1, col, right, 0, cartcomm, &reqs[3]);  // right
    MPI_Isend(&local_grid[IND(1, 1)], 1, row, top, 0, cartcomm, &reqs[4]);          // top
    MPI_Isend(&local_grid[IND(ny, 1)], 1, row, bottom, 0, cartcomm, &reqs[5]);     // bottom
    MPI_Isend(&local_grid[IND(1, 1)], 1, col, left, 0, cartcomm, &reqs[6]);         // left
    MPI_Isend(&local_grid[IND(1, nx)], 1, col, right, 0, cartcomm, &reqs[7]);      // right
    MPI_Waitall(8, reqs, MPI_STATUS_IGNORE);
    thalo += MPI_Wtime();
}  // iterations
```



Параллельная версия метода Якоби (2D)

```
    MPI_Type_free(&row);
    MPI_Type_free(&col);

    free(local_newgrid);
    free(local_grid);

    ttotal += MPI_Wtime();

    if (rank == 0)
        printf("# Heat 2D (mpi): grid: rows %d, cols %d, procs %d (px %d, py %d)\n", rows, cols, commsize, px, py);

    int namelen;
    char procname[MPI_MAX_PROCESSOR_NAME];
    MPI_Get_processor_name(procname, &namelen);
    printf("# P %4d (%2d, %2d) on %s: grid ny %d nx %d, total %.6f, mpi %.6f (%.2f) = allred %.6f (%.2f) + halo %.6f (%.2f)\n",
           rank, rankx, ranky, procname, ny, nx, ttotal, treduce + thalo, (treduce + thalo) / ttotal,
           treduce, treduce / (treduce + thalo), thalo, thalo / (treduce + thalo));

    double prof[3] = {ttotal, treduce, thalo};
    if (rank == 0) {
        MPI_Reduce(MPI_IN_PLACE, prof, NELEMS(prof), MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
        printf("# procs %d : grid %d %d : niters %d : total time %.6f : mpi time %.6f : allred %.6f : halo %.6f\n",
               commsize, rows, cols, niters, prof[0], prof[1] + prof[2], prof[1], prof[2]);
    } else {
        MPI_Reduce(prof, NULL, NELEMS(prof), MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

Параллельная версия метода Якоби (2D)

- Анализ времени выполнения информационных обменов при двумерной декомпозиции
- Время передачи сообщения размером *m* байт: $t(m) = \alpha + \beta m$
- Пусть сетка квадратная и содержит *n* строк и столбцов
- На каждой итерации выполняется четыре send и четыре recv:

$$t_{iter} = 8\alpha + \frac{8n}{p}\beta$$

Задание

- Сравнить в модели Хокни время дифференцированных обменов при 1D-декомпозиции и 2D-декомпозиции