Курс «Архитектурно-ориентированная оптимизация кода»

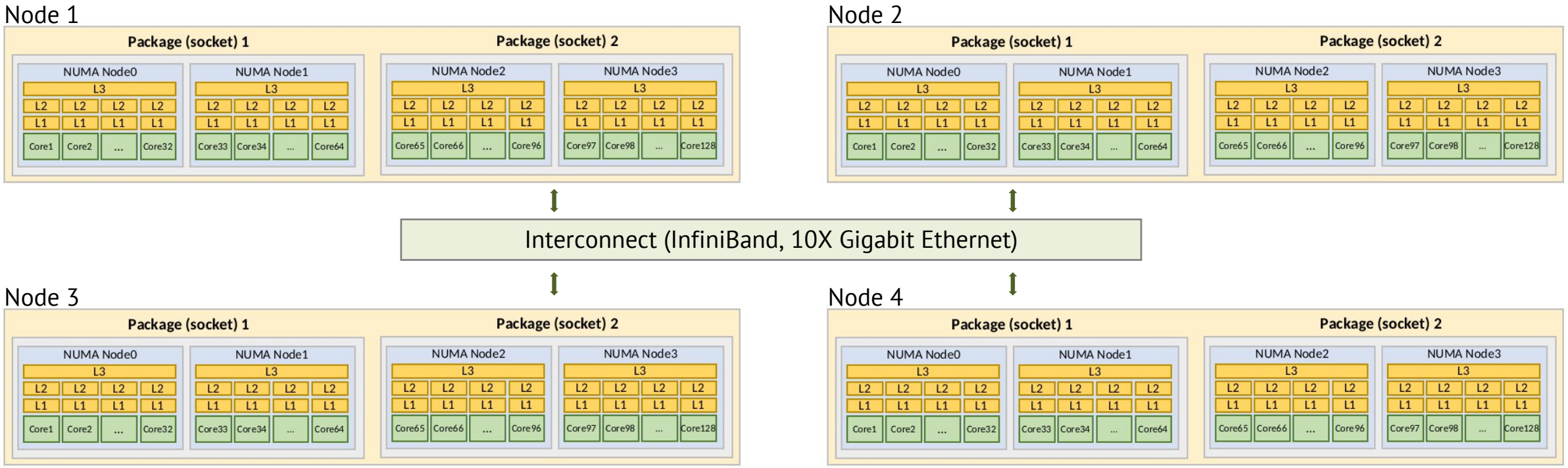
Гибридные MPI-программы

Михаил Курносов

Содержание

- MPI + OpenMP
- MPI + MPI 3.0 RMA Shared Memory
- MPI 3.0 RMA Put/Get
- MPI Neighborhood Collective

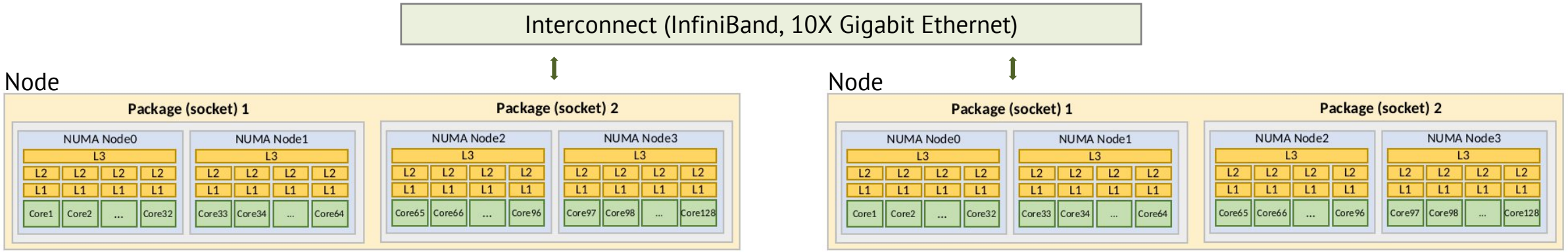
Иерархическая организация вычислительных систем



- Вычислительные узлы с NUMA-архитектурой
- Многоуровневая коммуникационная сеть (fat tree, spine leaf)

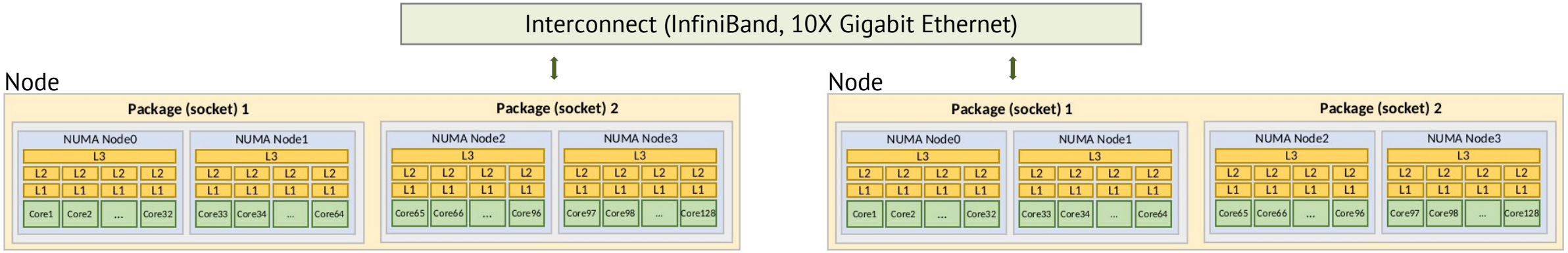
Гибридные программы MPI

- MPI для обменов между узлами через коммуникационную сеть (InfiniBand, 10X Gigabit Ethernet)
- Разделяемая память (shared memory) для взаимодействий в пределах узла (intra-node)
  - OpenMP, TBB, C11/C++11 Threads
  - MPI 3.0 Shared Memory API



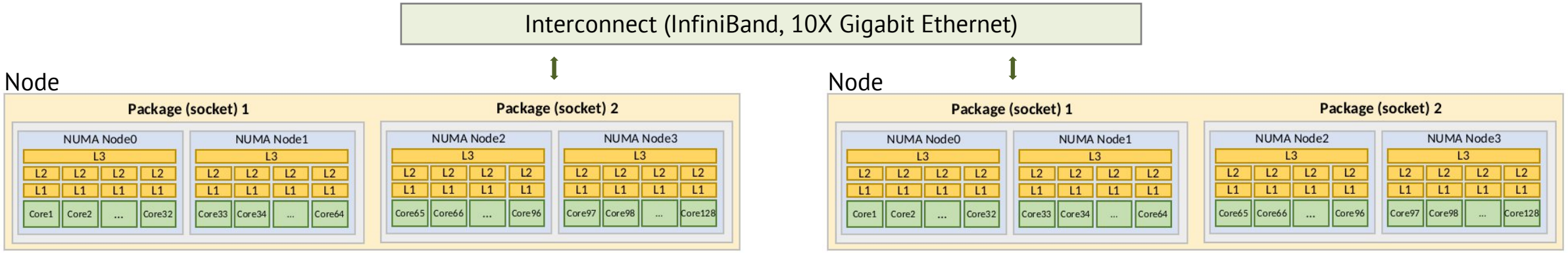
Гибридные программы MPI + OpenMP

- **MPI-процесс на каждом ядре** (Pure MPI)
- **Гибридная программа MPI + OpenMP**
  - Один MPI-процесс на узел + OpenMP-поток на каждое ядро
  - Один MPI-процесс на процессор + OpenMP-поток на каждое ядро процессора
  - Один MPI-процесс на NUMA-узел + OpenMP-поток на каждое ядро NUMA-узла
- **Гибридная программа MPI + MPI 3.0 RMA для обменов в пределах узла**
- **Альтернативы:** UPC, Coarray Fortran, Global Arrays



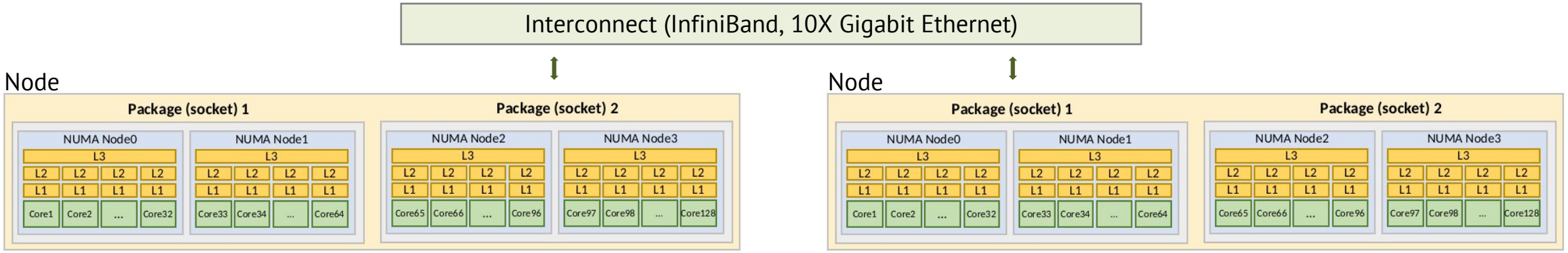
Традиционные MPI-программы (pure MPI)

- **Приемущества использования только MPI (pure MPI)**
  - Устоявшиеся подходы к архитектуре приложений, один стандарт
  - Библиотеке MPI не требуется обеспечивать поддержку многопточности (потокобезопасности)
- **Открытые вопросы**
  - Эффективность реализации MPI-обменов через разделяемую память: протоколы двустронних обменов и коллективных операций (CopyIn-CopyOut, Linux Cross Memory Attach, KNEM, XPMEM), NUMA-архитекутра



Гибридные MPI + OpenMP программы

- **Приемущества MPI + OpenMP**
  - Отсутсвуют обмены в пределах узла
  - Процессу доступа вся память вычислительного узла
  - Совемещение обменов и вычислений
- **Открытые вопросы**
  - Поддержка многопоточности MPI-библиотекой (специальные структуры данных, блокировки)



Поддержка многопоточности в MPI

```
int MPI_Init_thread(int *argc, char ** argv[], int required, int *provided);

int MPI_Query_thread(int *provided);
// provided – текущий уровень поддержки многопоточности (MPI_THREAD_FUNNELED, ...)

int MPI_Is_main_thread(int *flag);
// flag = 1 если поток является главным
// главным считается поток, вызвавший MPI_INIT или MPI_INIT_THREAD
```

- **MPI_THREAD_SINGLE**
  Only one thread will execute
- **MPI_THREAD_FUNNELED**
  The process may be multithreaded, but the application must ensure that only the *main thread* makes MPI calls
- **MPI_THREAD_SERIALIZED**
  The process may be multithreaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are “serialized”)
- **MPI_THREAD_MULTIPLE**
  Multiple threads may call MPI, with no restrictions

MPI_THREAD_FUNNELED (master-only)

Только главный поток может вызывать функции MPI

```
MPI_Init_thread(argc, argv, MPI_THREAD_FUNNELED, &provided);
if (provided < MPI_THREAD_FUNNELED) {
    MPI_Abort(comm, EXIT_FAILURE);
}

#pragma omp parallel
{
    // Вычисления
}

// Обмены в главном потоке
MPI_Send(sbuf, ...)
MPI_Recv(rbuf, ...)

#pragma omp parallel
{
    // Вычисления
}

MPI_Finalize();
```

- Пользователь обеспечивает вызов функций MPI только из главного потока

MPI_THREAD_FUNNELED (master-only)

Только главный поток может вызывать функции MPI

```
MPI_Init_thread(argc, argv, MPI_THREAD_FUNNELED, &provided);
if (provided < MPI_THREAD_FUNNELED) {
    MPI_Abort(comm, EXIT_FAILURE);
}

for (iterations) {
    #pragma omp parallel
    {
        // Вычисления: чтение и запись rbuf, sbuf
    }

    // Обмены в главном потоке
    if (is_master_thread) {
        MPI_Send(sbuf, ...)
        MPI_Recv(rbuf, ...)
    }
}

MPI_Finalize();
```

- Пользователь обеспечивает вызов функций MPI только из главного потока

MPI_THREAD_FUNNELED (master-only)

```
MPI_Init_thread(argc, argv, MPI_THREAD_FUNNELED, &provided);
if (provided < MPI_THREAD_FUNNELED) {
    MPI_Abort(comm, EXIT_FAILURE);
}

for (iterations) {
    #pragma omp parallel
    {
        // Вычисления (пул потоков): чтение и запись rbuf, sbuf

        // Обмены в главном потоке
        #pragma omp barrier
        #pragma omp master
        {
            MPI_Send(sbuf)
            MPI_Recv(rbuf)
        }
        #pragma omp barrier
    }
}

MPI_Finalize();
```

- **Барьерная синхронизация**
  чтение и запись sbuf, rbuf закончены, можно передавать, принимать в них данные

- **Барьерная синхронизация**
  обмены закончены, можно использовать sbuf, rbuf на следующей итерации

MPI_THREAD_FUNNELED (master-only)

```
MPI_Init_thread(argc, argv, MPI_THREAD_FUNNELED, &provided);
if (provided < MPI_THREAD_FUNNELED) {
    MPI_Abort(comm, EXIT_FAILURE);
}

for (iterations) {
    #pragma omp parallel
    {
        // Вычисления (пул потоков): чтение и запись rbuf, sbuf

        // Обмены в одном потоке
        #pragma omp single
        {
            MPI_Send(sbuf)
            MPI_Recv(rbuf)
        }
    }
}

MPI_Finalize();
```

- **Несоответствие стандарту!**
  Секцию single может выполнять любой поток, не только главный

MPI_THREAD_SERIALIZED

Любой поток может вызывать функции MPI, но в любой момент времени только один

```
MPI_Init_thread(argc, argv, MPI_THREAD_SERIALIZED, &provided);
if (provided < MPI_THREAD_SERIALIZED) {
    MPI_Abort(comm, EXIT_FAILURE);
}

for (iterations) {
    #pragma omp parallel
    {
        // Вычисления: чтение и запись rbuf, sbuf

        // Обмены в одном потоке (любом)
        #pragma omp barrier
        #pragma omp single
        {
            MPI_Send(sbuf)
            MPI_Recv(rbuf)
        }
        #pragma omp barrier
    }
}

MPI_Finalize();
```

- Пользователь обеспечивает потокобезопасный вызов функций MPI (из одного потока, из критической секции)

- **Только один поток вызывает Send, Recv** (первым достигший single)

MPI_THREAD_SERIALIZED

Любой поток может вызывать функции MPI, но в любой момент времени только один

```
MPI_Init_thread(argc, argv, MPI_THREAD_SERIALIZED, &provided);
if (provided < MPI_THREAD_SERIALIZED) {
    MPI_Abort(comm, EXIT_FAILURE);
}

for (iterations) {
    #pragma omp parallel
    {
        // Вычисления: чтение и запись rbuf, sbuf

        // Обмены в одном потоке (любом)
        #pragma omp barrier
        #pragma omp critical
        {
            MPI_Send(sbuf)
            MPI_Recv(rbuf)
        }
        #pragma omp barrier
    }
}

MPI_Finalize();
```

- Пользователь обеспечивает потокобезопасный вызов функций MPI (из одного потока, из критической секции)

- **Все потоки вызовут Send, Recv**, но только один поток в любой момент времени

MPI_THREAD_MULTIPLE

Любой поток может вызывать функции MPI, в любой момент времени

```
MPI_Init_thread(argc, argv, MPI_THREAD_MULTIPLE, &provided);
if (provided < MPI_THREAD_MULTIPLE) {
    MPI_Abort(comm, EXIT_FAILURE);
}

for (iterations) {
    #pragma omp parallel
    {
        // Вычисления: чтение и запись rbuf, sbuf

        #pragma omp barrier
        MPI_Send(sbuf, ...)
        MPI_Recv(rbuf, ...)
        #pragma omp barrier
    }
}

MPI_Finalize();
```

- Библиотека MPI обеспечивает потокобезопасное функционирование

- **Все потоки вызывают Send, Recv**

MPI_THREAD_MULTIPLE

Любой поток может вызывать функции MPI, в любой момент времени

```
MPI_Init_thread(argc, argv, MPI_THREAD_MULTIPLE, &provided);

#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task depend (out:sbuf)
        { // Вычисления, запись в sbuf }
        #pragma omp task
        { ... }
        #pragma omp task
        { ... }

        #pragma omp task depend (in:sbuf) depend (out:rbuf)
        { MPI_Sendrecv(sbuf, ..., rbuf, ...); }

        #pragma omp task
        { ... }
        #pragma omp task
        { ... }

        #pragma omp task depend (in:rbuf)
        { // Вычисиления, чтение rbuf }
    }
}
```

Численное интегрирование: MPI

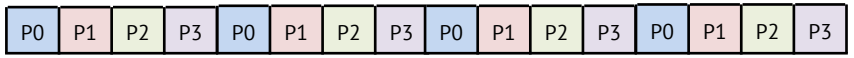
```
double func(double x) { return exp(-x * x); }

int main(int argc, char **argv)
{
    const double a = -4.0; const double b = 4.0;
    const int n = 1000000000;
    int commsize, rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    double h = (b - a) / n, s = 0.0;
    for (int i = rank; i < n; i += commsize)
        s += func(a + h * (i + 0.5));

    double gsum = 0;
    MPI_Reduce(&s, &gsum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        gsum *= h;
        printf("# Result Pi: %.12ff\n", gsum * gsum);
    }
    MPI_Finalize();
    return 0;
}
```

- Pure MPI
- Циклическое распределение итераций между процессами

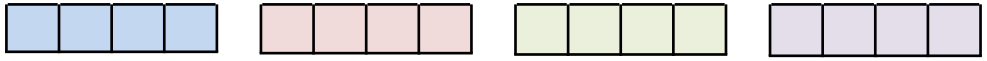
Численное интегрирование: MPI + OpenMP (master-only)

```
double func(double x) { return exp(-x * x); }

int main(int argc, char **argv)
{
    const double a = -4.0;  const double b = 4.0;
    const int n = 1000000000; int commsize, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    double h = (b - a) / n, s = 0.0;
    #pragma omp parallel
    {
        #pragma omp for reduction(+:s)
        for (int i = rank; i < n; i += commsize) {
            s += func(a + h * (i + 0.5));
        }
    }
    double gsum = 0;
    MPI_Reduce(&s, &gsum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        gsum *= h;
        printf("# Result Pi: %.12ff\n", gsum * gsum);
    }
    MPI_Finalize();
    return 0;
}
```

1) Циклическое распределение итераций между MPI-процессами



2) Многопоточная обработка итераций процесса



```
$ mpicc -fopenmp -o int ./int.c -lm

$ cat task.job
#!/bin/sh
#SBATCH --nodes=4 --ntasks-per-node=8
mpiexec -N 1 -x OMP_NUM_THREADS=8 ./int
```

Решение двумерного уравнения Лапласа Pure MPI

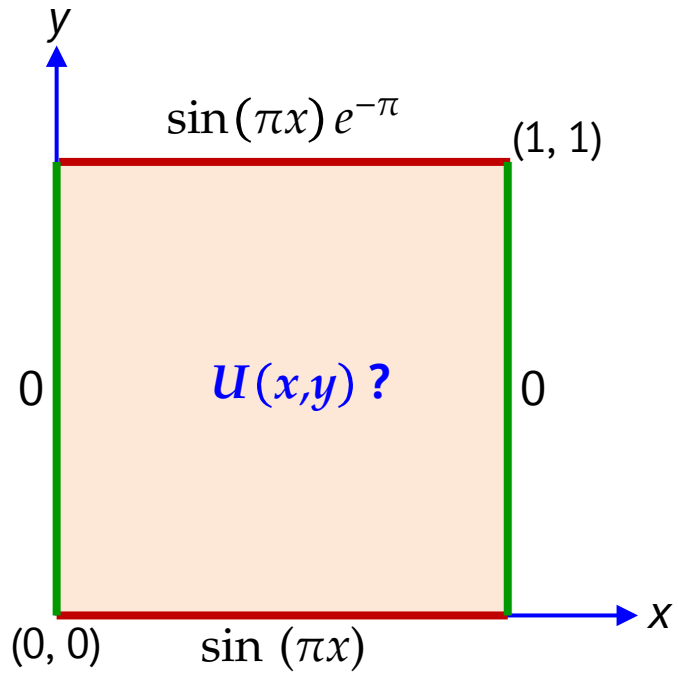
Решение стационарного двумерного уравнения Лапласа

- **Найти** решение стационарного двумерного уравнения Лапласа

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

- 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 заданной задачи известно аналитическое решение

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



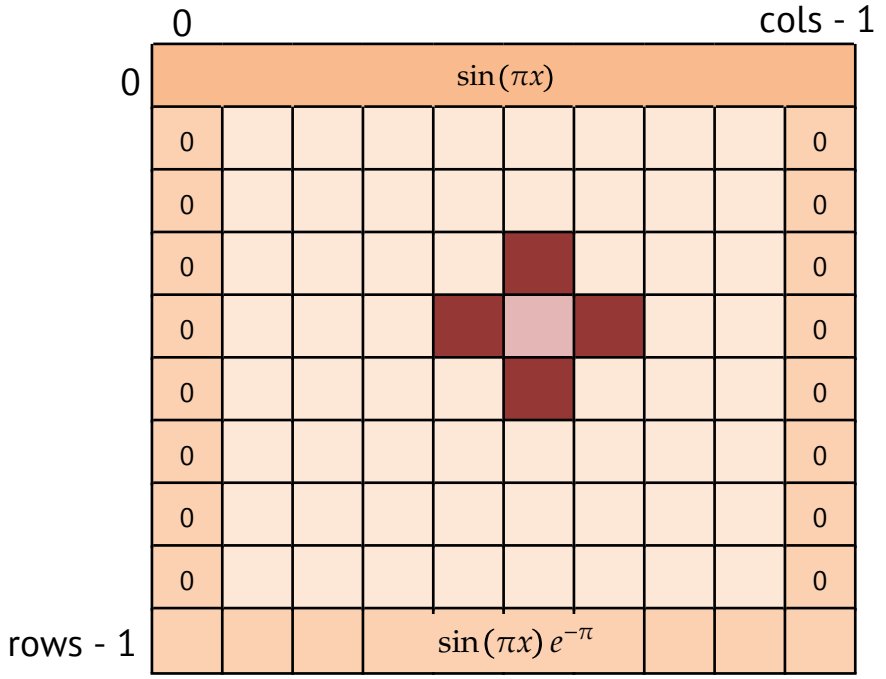
Разностная аппроксимация оператора Лапласа

- 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
```



[*] Эндрюс Г. Основы многопоточного, параллельного и распределенного программирования. - М.: Вильямс, 2003.

Решение двумерного уравнения Лапласа (Pure MPI)

2D декомпозиция расчетной области

```
int main(int argc, char *argv[])
{
    int commsize, rank;
    MPI_Init(&argc, &argv);
    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];

    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];

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

Процессы MPI_COMM_WORLD

0	1	2	3	4	5

Процессы выстроены в 2D решетку

commsize = px * py = 6

	px = 3		
py = 2	0	2	4
	1	3	5

```
# 2D-нумерация процессов
rank = 0, rankx = 0, ranky = 0
rank = 1, rankx = 0, ranky = 1
rank = 2, rankx = 1, ranky = 0

# Соседние процессы
rank = 2:
  left = 0, right = 4
  top = MPI_PROC_NULL,
  bottom = 3
```

Решение двумерного уравнения Лапласа (Pure MPI)

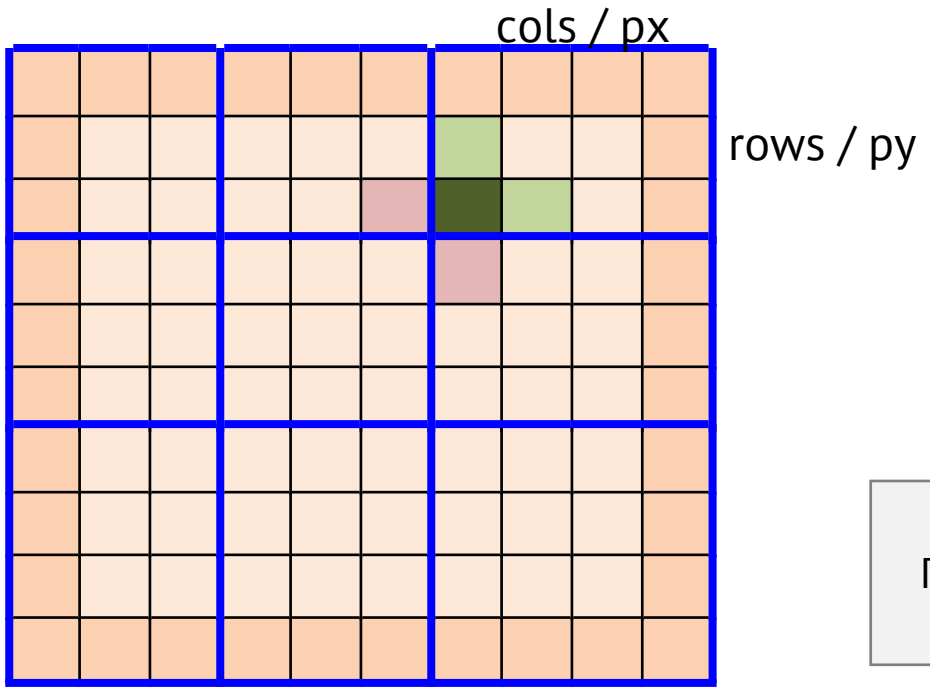
2D декомпозиция расчетной области

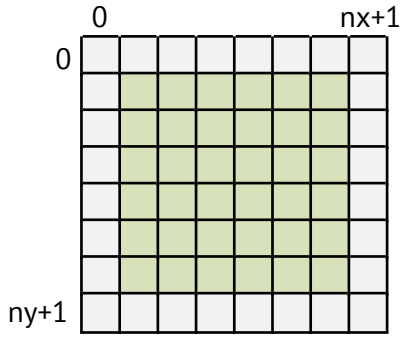
```
/* Broadcast command line arguments */
int rows, cols;
if (rank == 0) {
    rows = (argc > 1) ? atoi(argv[1]) : py * 100;
    cols = (argc > 2) ? atoi(argv[2]) : px * 100;

    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];
}
```

Решение двумерного уравнения Лапласа (Pure MPI)

2D декомпозиция расчетной области

- Сетка хранится в памяти в распределенном виде
- Каждому процессу назначается подмассив [rows / py][cols / px] строк расчетной сетки
- Для расчета значений некоторых точек требуются данные соседних полос, которые находятся в памяти других процессов



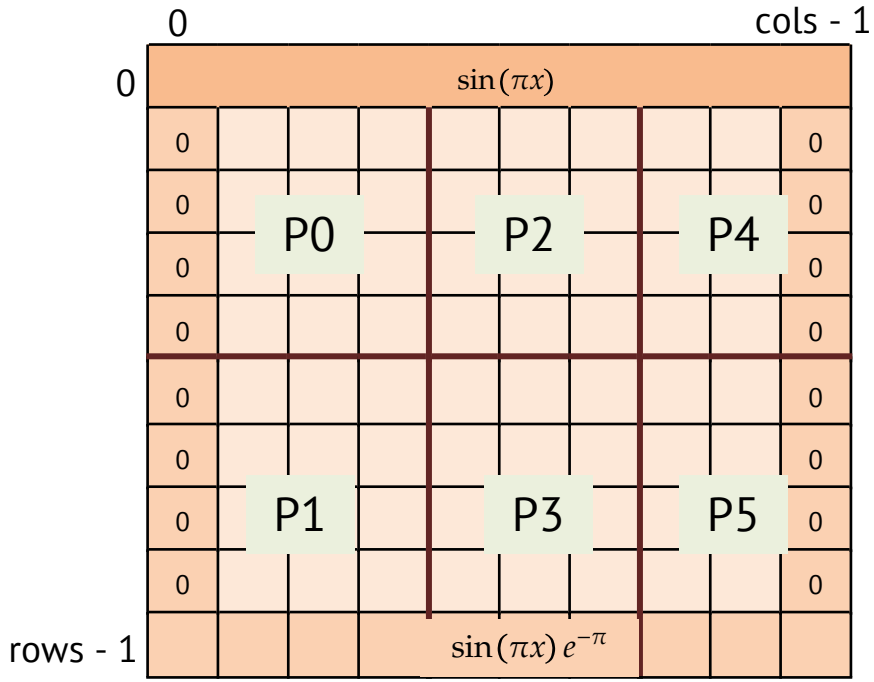


Теневые ячейки (halo, ghost cells)
Процесс получает от соседей значения граничных элементов и сохряняет их в теневых ячейках

Решение двумерного уравнения Лапласа (Pure MPI)

2D декомпозиция расчетной области

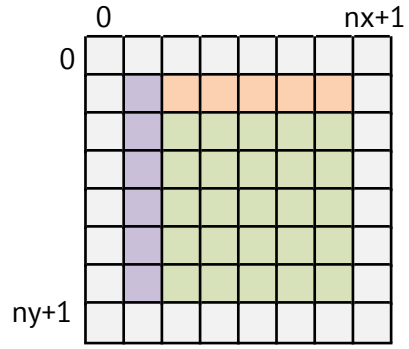
```
/* Allocate memory for local 2D subgrids with halo cells [0..ny + 1][0..nx + 1] */
int ny = get_block_size(rows, ranky, py);
int nx = get_block_size(cols, rankx, px);
double *local_grid = xcalloc((ny + 2) * (nx + 2), sizeof(*local_grid));
double *local_newgrid = xcalloc((ny + 2) * (nx + 2), sizeof(*local_newgrid));

/* Fill boundary points: left and right borders are zero filled */
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);
    }
}
```



Решение двумерного уравнения Лапласа (Pure MPI)

2D декомпозиция расчетной области

```
// Left and right borders type (row-major format)
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);
```



Решение двумерного уравнения Лапласа (Pure MPI)

2D декомпозиция расчетной области

```
MPI_Request reqs[8];
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;

    MPI_Allreduce(MPI_IN_PLACE, &maxdiff, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
    if (maxdiff < EPS)
        break;
```

Решение двумерного уравнения Лапласа (Pure MPI)

2D декомпозиция расчетной области

```
        // Продолжение цикла

        // Halo exchange
        int nreqs = 0;
        MPI_Irecv(&local_grid[IND(0, 1)], 1, row, top, 0, cartcomm, &reqs[nreqs++]);          // top
        MPI_Irecv(&local_grid[IND(ny + 1, 1)], 1, row, bottom, 0, cartcomm, &reqs[nreqs++]); // bottom
        MPI_Irecv(&local_grid[IND(1, 0)], 1, col, left, 0, cartcomm, &reqs[nreqs++]);         // left
        MPI_Irecv(&local_grid[IND(1, nx + 1)], 1, col, right, 0, cartcomm, &reqs[nreqs++]);   // right
        MPI_Isend(&local_grid[IND(1, 1)], 1, row, top, 0, cartcomm, &reqs[nreqs++]);          // top
        MPI_Isend(&local_grid[IND(ny, 1)], 1, row, bottom, 0, cartcomm, &reqs[nreqs++]);      // bottom
        MPI_Isend(&local_grid[IND(1, 1)], 1, col, left, 0, cartcomm, &reqs[nreqs++]);         // left
        MPI_Isend(&local_grid[IND(1, nx)], 1, col, right, 0, cartcomm, &reqs[nreqs++]);       // right
        MPI_Waitall(nreqs, reqs, MPI_STATUSES_IGNORE);
    }
    MPI_Type_free(&row);
    MPI_Type_free(&col);

    free(local_newgrid);
    free(local_grid);
    MPI_Finalize();
}
```

Решение двумерного уравнения Лапласа (Pure MPI)

2D декомпозиция расчетной области

```
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);
}
```

Решение двумерного уравнения Лапласа MPI + OpenMP

Решение двумерного уравнения Лапласа: MPI + OpenMP (v1)

- **MPI**
  - хранение подматриц
  - рассылка начальных значений
  - обмен граничными значениями на каждой итерации

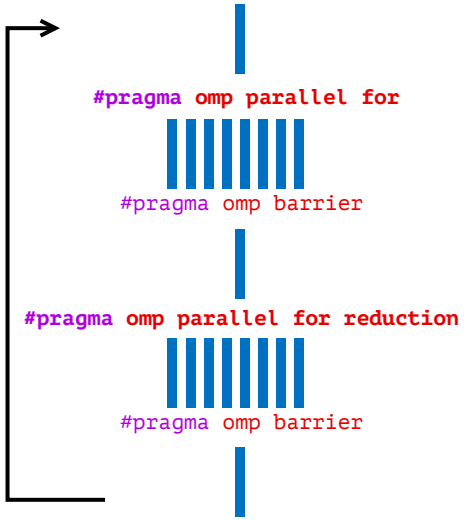
- **OpenMP**
  - многопоточное вычисление значений во внутренних узлах подсетки процесса
  - многопточное вычисление maxdiff для подсеток процесса (grid, newgrid)

Решение двумерного уравнения Лапласа: MPI + OpenMP (v1)

```
int main(int argc, char *argv[])
{
    int commsize, rank, provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
    if (provided < MPI_THREAD_FUNNELED) {
        fprintf(stderr, "MPI: MPI_THREAD_FUNNELED not supported\n");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }
```

Решение двумерного уравнения Лапласа: MPI + OpenMP (v1)

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

    // Update interior points
    #pragma omp parallel for
    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;
    #pragma omp parallel for reduction(max:maxdiff)
    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]));
        }
    }
```



- На каждой итерации параллельный регион активируется дважды (накладные расходы)

Решение двумерного уравнения Лапласа: MPI + OpenMP (v2)

- **MPI**
  - хранение подматриц
  - рассылка начальных значений
  - обмен граничными значениями на каждой итерации

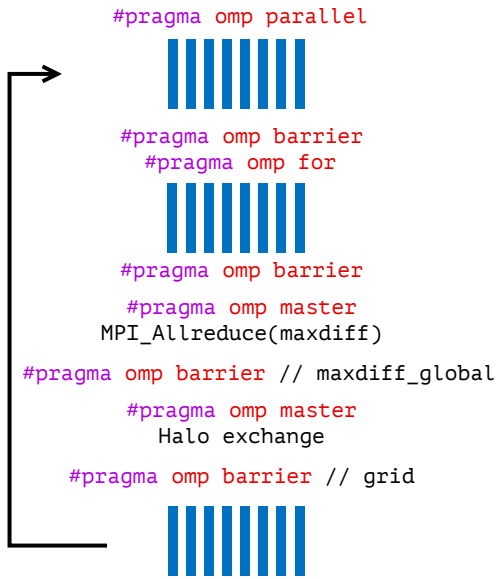
- **OpenMP**
  - **параллельный регион активируется один раз, до основного цикла**
  - многопоточное вычисление значений во внутренних узлах подсетки процесса
  - многопточное вычисление maxdiff для подсеток процесса (grid, newgrid)

Решение двумерного уравнения Лапласа: MPI + OpenMP (v2)

```
int niters_global;
double maxdiff_global = 0, double maxdiff;
#pragma omp parallel
{
    double *grid = local_grid;
    double *newgrid = local_newgrid;
    int niters = 0;

    for (;;) {
        niters++;

        maxdiff = 0;
        #pragma omp barrier

        // Update interior points
        #pragma omp for reduction(max:maxdiff)
        for (int i = 1; i <= ny; i++) {
            for (int j = 1; j <= nx; j++) {
                newgrid[IND(i, j)] =
                    (grid[IND(i - 1, j)] + grid[IND(i + 1, j)] +
                     grid[IND(i, j - 1)] + grid[IND(i, j + 1)]) * 0.25;
                // Check termination condition
                int ind = IND(i, j);
                maxdiff = fmax(maxdiff, fabs(grid[ind] - newgrid[ind]));
            }
        }
```

- Один параллельный регион
- Локальные копии переменных для предотвращения гонки данных (grid, newgrid, niters)

- Барьерная синхронизация гарантирует, что все потоки переустановили maxdiff



Решение двумерного уравнения Лапласа: MPI + OpenMP (v2)

```
// Swap grids
double *p = grid;
grid = newgrid;
newgrid = p;

#pragma omp master
{
MPI_Allreduce(&maxdiff, &maxdiff_global, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
}

#pragma omp barrier
// Wait for maxdiff_global

if (maxdiff_global < EPS) {
    #pragma omp single
    niters_global = niters;
    break;
}
```

Решение двумерного уравнения Лапласа: MPI + OpenMP (v2)

```
            // Halo exchange
            #pragma omp master
            {
            MPI_Irecv(&grid[IND(0, 1)], 1, row, top, 0, cartcomm, &reqs[0]);          // top
            MPI_Irecv(&grid[IND(ny + 1, 1)], 1, row, bottom, 0, cartcomm, &reqs[1]); // bottom
            MPI_Irecv(&grid[IND(1, 0)], 1, col, left, 0, cartcomm, &reqs[2]);         // left
            MPI_Irecv(&grid[IND(1, nx + 1)], 1, col, right, 0, cartcomm, &reqs[3]);  // right
            MPI_Isend(&grid[IND(1, 1)], 1, row, top, 0, cartcomm, &reqs[4]);          // top
            MPI_Isend(&grid[IND(ny, 1)], 1, row, bottom, 0, cartcomm, &reqs[5]);      // bottom
            MPI_Isend(&grid[IND(1, 1)], 1, col, left, 0, cartcomm, &reqs[6]);         // left
            MPI_Isend(&grid[IND(1, nx)], 1, col, right, 0, cartcomm, &reqs[7]);       // right
            MPI_Waitall(8, reqs, MPI_STATUSES_IGNORE);
            }

            #pragma omp barrier
            // Wait for halo cells (grid[][])
        }
    }
```

Решение двумерного уравнения Лапласа MPI + MPI 3.0 Shared Memory

MPI 3 Remote Memory Access (RMA) API

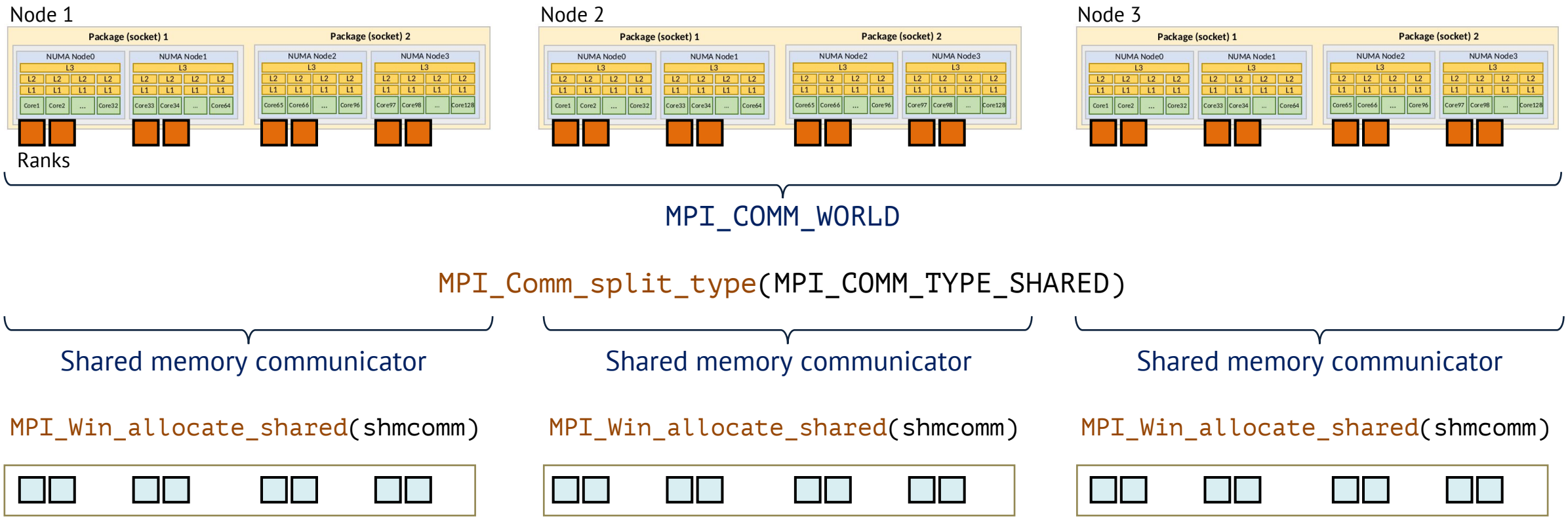
- **MPI 3 RMA** — набор функций и типов данных для создания и прямого доступа (load, store) к областям памяти удаленных процессов

- **Коллективные операции создания окон (областей памяти)**
  - `int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)`
  - `int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win)`
  - `int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win)`
  - `int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)`

- **Функции доступа к областям памяти ()**
  - `int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size, int *disp_unit, void *baseptr)`
  - `int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)`
  - `MPI_Put(), MPI_Get(), MPI_Accumulate(), MPI_Get_accumulate(), MPI_Fetch_and_op(), MPI_Compare_and_swap(), MPI_Rput(), MPI_Rget(), MPI_Raccumulate(), MPI_Rget_accumulate()`
  - `MPI_Win_fence(win), MPI_Win_lock_all(), ...`

- Hoefler T. et al. ***Remote Memory Access Programming in MPI-3*** // ACM TOPC, 2015, http://htor.inf.ethz.ch/publications/img/mpi3-rma-overview-and-model.pdf

Создание сегментов разделяемой памяти MPI 3



- Создан сегмент разделямой памяти (окно) для каждого процесса
- Процессы одного узла (shmcomm) могут получить адрес любого окна — MPI_Win_shared_query()

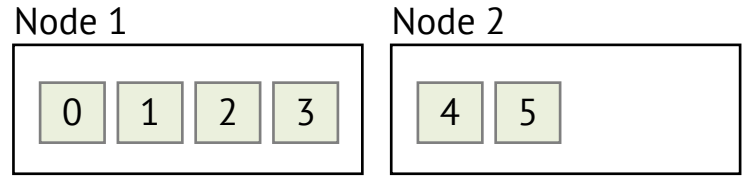
Создание сегментов разделяемой памяти MPI 3

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    MPI_Comm_split_type(MPI_COMM_WORLD, ..., MPI_COMM_TYPE_SHARED, .., &shmcomm);

    MPI_Win_allocate_shared(shmcomm, ..., &win);

    // Starts an RMA access epoch to all processes in win
    MPI_Win_lock_all(win);

    // Copy data to local part of shared memory

    // Synchronize the private and public window copies of win
    MPI_Win_sync(win);

    // Use shared memory

    // Complete a shared RMA access epoch started
    MPI_Win_unlock_all(win);

    MPI_Win_free(&win);
    MPI_Finalize();
    return 0;
}
```

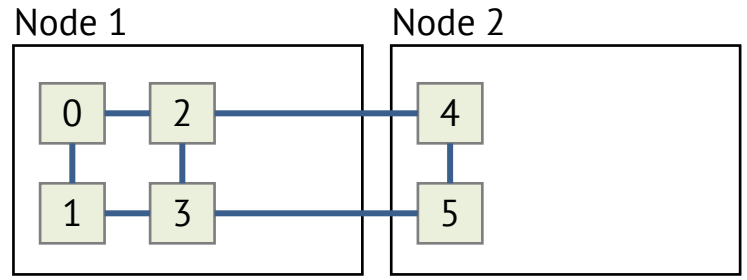
Решение двумерного уравнения Лапласа: MPI + MPI RMA

```
int main(int argc, char *argv[])
{
    int rank, commsize, namelen;
    MPI_Init(&argc, &argv);

    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];

    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];

    /* Compute ranks of neighbours in 2D grid */
    int neigh_left, neigh_right, neigh_top, neigh_bottom;
    MPI_Cart_shift(cartcomm, 0, 1, &neigh_left, &neigh_right);
    MPI_Cart_shift(cartcomm, 1, 1, &neigh_top, &neigh_bottom);
```

MPI_COMM_WORLD

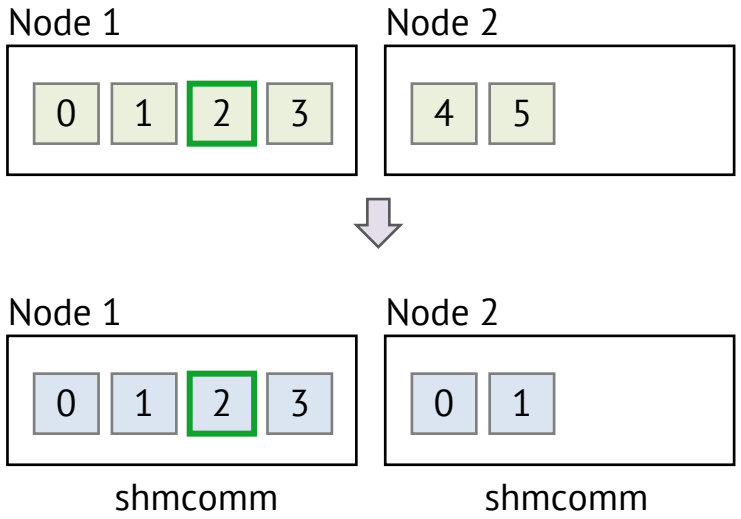


cartcomm
px=3, py=2



Решение двумерного уравнения Лапласа: MPI + MPI RMA

```
// Create communicator for intra-node communication via shared memory (MPI 3 API)
MPI_Comm shmcomm;
MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &shmcomm);
int shmcommsize;
MPI_Comm_size(shmcomm, &shmcommsize);

// Compute ranks of neighbours in the shared memory communicator
const int nneigh = 4;
int neigh[nneigh];
neigh[0] = neigh_left;
neigh[1] = neigh_right;
neigh[2] = neigh_top;
neigh[3] = neigh_bottom;
int neigh_shmcomm[nneigh];

// Translate ranks from MPI_COMM_WORLD to shmcomm
translate_ranks(shmcomm, neigh, neigh_shmcomm, nneigh);

int neigh_left_shmcomm = neigh_shmcomm[0];
int neigh_right_shmcomm = neigh_shmcomm[1];
int neigh_top_shmcomm = neigh_shmcomm[2];
int neigh_bottom_shmcomm = neigh_shmcomm[3];
```

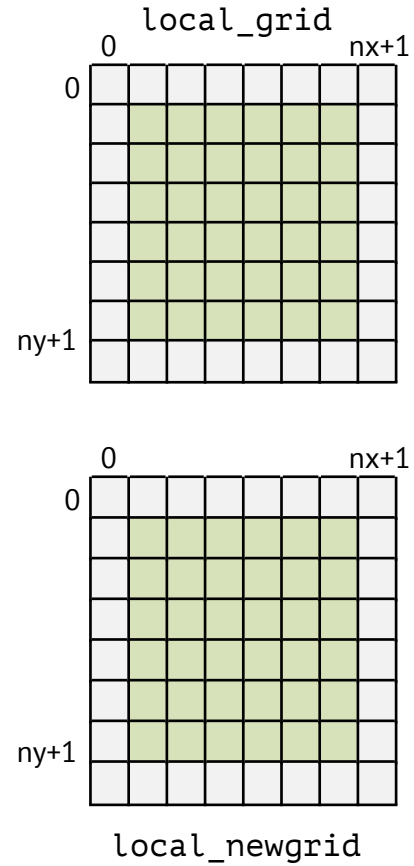


rank 2

```
neigh_left = 0
neigh_right = 4
neigh_top = MPI_PROC_NULL
neight_bottom = 3
neigh_left_shmcomm = 0
neigh_right_shmcomm = MPI_UNDEFINED
neigh_top_shmcomm = MPI_PROC_NULL
neigh_top_bottom = 3
```

Решение двумерного уравнения Лапласа: MPI + MPI RMA

```
// Local process has two 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);

// Create a shared memory window for grid and newgrid
size_t grid_size = (ny + 2) * (nx + 2);
MPI_Win win;
void *shmptr;
MPI_Win_allocate_shared(2 * grid_size * sizeof(double), 1,
                        MPI_INFO_NULL, shmcomm, &shmptr, &win);

double *local_grid = shmptr;
double *local_newgrid = local_grid + grid_size;
```



Window (shmptr):

local_grid: 2 * grid_size * sizeof(double)	local_newgrid: 2 * grid_size * sizeof(double)

- Каждый процесс создает сегмент разделяемой памяти с двумя сетками: grid, newgrid
- Процесс-владелец заполняет внутренние элементы
- В теневые ячейки процесс-владелец считывает данные из памяти соседних процессов или получает их через двустронние обмены

Решение двумерного уравнения Лапласа: MPI + MPI RMA

```
// Each process query pointers to shared memory segments of neighbours (located on the same node)
int neigh_left_gridsize = 0, neigh_right_gridsize = 0, neigh_top_gridsize = 0, neigh_bottom_gridsize = 0;
int neigh_left_nx = 0, neigh_right_nx = 0, neigh_top_ny = 0;
double *neigh_left_shmptr = NULL;  double *neigh_right_shmptr = NULL;
double *neigh_top_shmptr = NULL;   double *neigh_bottom_shmptr = NULL;

if (neigh_left != MPI_PROC_NULL && neigh_left_shmcomm != MPI_UNDEFINED) {
    MPI_Win_shared_query(win, neigh_left_shmcomm, &segsize, &disp_unit, &neigh_left_shmptr);
    neigh_left_gridsize = segsize / sizeof(double) / 2;
    neigh_left_nx = get_block_size(cols, rankx - 1, px);
}
if (neigh_right != MPI_PROC_NULL && neigh_right_shmcomm != MPI_UNDEFINED) {
    MPI_Win_shared_query(win, neigh_right_shmcomm, &segsize, &disp_unit, &neigh_right_shmptr);
    neigh_right_gridsize = segsize / sizeof(double) / 2;
    neigh_right_nx = get_block_size(cols, rankx + 1, px);
}
if (neigh_top != MPI_PROC_NULL && neigh_top_shmcomm != MPI_UNDEFINED) {
    MPI_Win_shared_query(win, neigh_top_shmcomm, &segsize, &disp_unit, &neigh_top_shmptr);
    neigh_top_gridsize = segsize / sizeof(double) / 2;
    neigh_top_ny = get_block_size(rows, ranky - 1, py);
}
if (neigh_bottom != MPI_PROC_NULL && neigh_bottom_shmcomm != MPI_UNDEFINED) {
    MPI_Win_shared_query(win, neigh_bottom_shmcomm, &segsize, &disp_unit, &neigh_bottom_shmptr);
    neigh_bottom_gridsize = segsize / sizeof(double) / 2;
}
```

- Процесс определяет адреса сегментов соседей и размеры их сеток для корректного вычисления индексов для загрузки элементов граничных областей

Решение двумерного уравнения Лапласа: MPI + MPI RMA

```
/*
 * 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)
 */
memset(shmptr, 0, 2 * grid_size * sizeof(double));

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)
    ...
}

if (ranky == py - 1) {
    // Initialize bottom border: u(x, 1) = sin(pi * x) * exp(-pi)
    ...
}
```

Решение двумерного уравнения Лапласа: MPI + MPI RMA

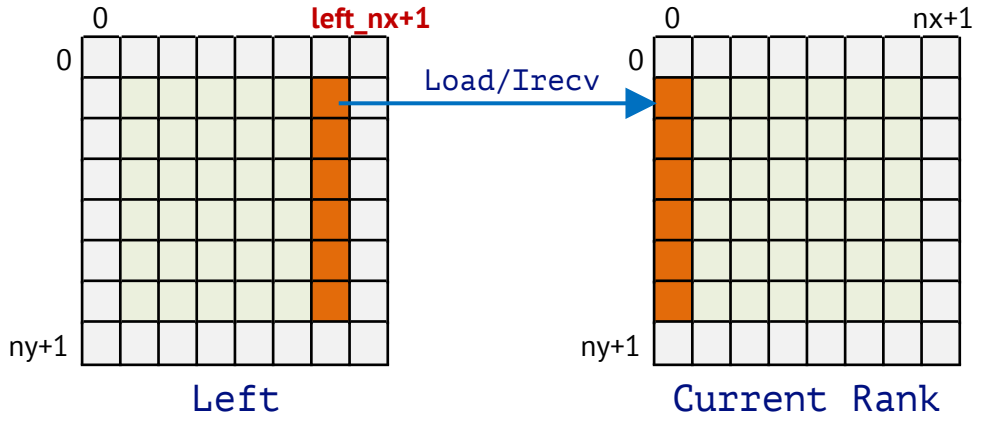
```
// На каждой итерации рабочая сетка меняется
int shm_grid_idx = 0;
int niters = 0;
MPI_Win_lock_all(MPI_MODE_NOCHECK, win);

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]));
        }
    }
```

Решение двумерного уравнения Лапласа: MPI + MPI RMA

```
// Swap grids (after termination local_grid will contain result)
double *p = local_grid;
local_grid = local_newgrid;
local_newgrid = p;
shm_grid_idx ^= 1;

MPI_Allreduce(MPI_IN_PLACE, &maxdiff, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
if (maxdiff < EPS) {
    break;
}

MPI_Win_sync(win);
MPI_Barrier(shmcomm);
// Ожидание завершения работы с областью всех соседей на узле
```

Решение двумерного уравнения Лапласа: MPI + MPI RMA

```
/* Halo exchange */
int nreqs = 0;
if (neigh_left != MPI_PROC_NULL) {
    if (neigh_left_shmptr) {
        /* Left neighbour on the same node -- use shared memory */
        /* Use first or second subgrid according to iteration number */
        double *shm = neigh_left_shmptr + (shm_grid_idx ? neigh_left_gridsize : 0);
        /* Load column from neighbour's memory */
        for (int i = 1; i <= ny; i++) {
            local_grid[IND(i, 0)] = shm[i * (neigh_left_nx + 2) + neigh_left_nx];
        }
    } else {
        /* Left neighbour on a remote node -- use pt2pt */
        MPI_Irecv(&local_grid[IND(1, 0)], 1, col, neigh_left, 0, cartcomm, &reqs[nreqs++]);
        MPI_Isend(&local_grid[IND(1, 1)], 1, col, neigh_left, 0, cartcomm, &reqs[nreqs++]);
    }
}
```



- **Левый сосед на локальном узле** — загружаем из памяти его граничный столбец в столбец своих теневых ячеек
- **Левый сосед на другом узле** — получаем столбец операцией Irecv

Решение двумерного уравнения Лапласа: MPI + MPI RMA

```
if (neigh_right != MPI_PROC_NULL) {
    if (neigh_right_shmptr) {
        double *shm = neigh_right_shmptr + (shm_grid_idx ? neigh_right_gridsize : 0);
        for (int i = 1; i <= ny; i++) {
            local_grid[IND(i, nx + 1)] = shm[i * (neigh_right_nx + 2) + 1];
        }
    } else {
        MPI_Irecv(&local_grid[IND(1, nx + 1)], 1, col, neigh_right, 0, cartcomm, &reqs[nreqs++]);
        MPI_Isend(&local_grid[IND(1, nx)], 1, col, neigh_right, 0, cartcomm, &reqs[nreqs++]);
    }
}

if (neigh_top != MPI_PROC_NULL) {
    if (neigh_top_shmptr) {
        double *shm = neigh_top_shmptr + (shm_grid_idx ? neigh_top_gridsize : 0);
        for (int j = 1; j <= nx; j++) {
            local_grid[IND(0, j)] = shm[neigh_top_ny * (nx + 2) + j];
        }
    } else {
        MPI_Irecv(&local_grid[IND(0, 1)], nx, MPI_DOUBLE, neigh_top, 0, cartcomm, &reqs[nreqs++]);
        MPI_Isend(&local_grid[IND(1, 1)], nx, MPI_DOUBLE, neigh_top, 0, cartcomm, &reqs[nreqs++]);
    }
}
```

Решение двумерного уравнения Лапласа: MPI + MPI RMA

```
        if (neigh_bottom != MPI_PROC_NULL) {
            if (neigh_bottom_shmptr) {
                double *shm = neigh_bottom_shmptr + (shm_grid_idx ? neigh_bottom_gridsize : 0);
                for (int j = 1; j <= nx; j++) {
                    local_grid[IND(ny + 1, j)] = shm[1 * (nx + 2) + j];
                }
            } else {
                MPI_Irecv(&local_grid[IND(ny + 1, 1)], nx, MPI_DOUBLE, neigh_bottom, 0, cartcomm,
                          &reqs[nreqs++]);
                MPI_Isend(&local_grid[IND(ny, 1)], nx, MPI_DOUBLE, neigh_bottom, 0, cartcomm,
                          &reqs[nreqs++]);
            }
        }
        if (nreqs > 0) {
            MPI_Waitall(nreqs, reqs, MPI_STATUSES_IGNORE);
        }
        thalo += MPI_Wtime();
    }
    MPI_Win_unlock_all(win);

    MPI_Type_free(&col);
    MPI_Win_free(&win);
    MPI_Comm_free(&shmcomm);
    MPI_Finalize();
    return 0;
}
```

Решение двумерного уравнения Лапласа
MPI RMA One-Sided Operations

Решение двумерного уравнения Лапласа (MPI One-Sided RMA)

- Процессы выделяют память под две локальные сетки (grid, newgrid, MPI_Alloc_mem)
- Процессы создают окно для удалянного доступа к областям памяти других процессов (MPI_Win_create)
- В основном цикле каждый процесс записывает свои граничные строки и столбцы в теневые ячейки соответствующих соседних процессов (MPI_Put)
- Синхронизация и ожидания завершения RMA-операций:
  - `MPI_Win_lock`
  - `MPI_Win_fence`

Решение двумерного уравнения Лапласа (MPI One-Sided RMA)

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

MPI_Comm cartcomm;
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];

/*
 * Compute ranks of neighbours in 2D grid
 * MPI_Cart_shift returns MPI_PROC_NULL if process have no neighbour for given dimension
 */
int neigh_left, neigh_right, neigh_top, neigh_bottom;
MPI_Cart_shift(cartcomm, 0, 1, &neigh_left, &neigh_right);
MPI_Cart_shift(cartcomm, 1, 1, &neigh_top, &neigh_bottom);
```

Решение двумерного уравнения Лапласа (MPI One-Sided RMA)

```
int ny = get_block_size(rows, ranky, py);
int nx = get_block_size(cols, rankx, px);

/* Create a shared memory window for grid and newgrid */
size_t grid_size = (ny + 2) * (nx + 2);
void *shmptr;
MPI_Alloc_mem(2 * grid_size * sizeof(double), MPI_INFO_NULL, &shmptr);

MPI_Win win;
MPI_Win_create(shmptr, 2 * grid_size * sizeof(double), sizeof(double), MPI_INFO_NULL,
               MPI_COMM_WORLD, &win);

double *local_grid = shmptr;
double *local_newgrid = local_grid + grid_size;
```

- Каждый процесс выделяет память для хранения двух сеток (включая теневые ячейки)
- Коллективной операцией MPI_Win_create() процессы создают окно, которое позволяет им обращаться к указанным областям памяти (shmptr) с использованием односторонних операций

Решение двумерного уравнения Лапласа (MPI One-Sided RMA)

```
int neigh_left_gridsize = 0, neigh_right_gridsize = 0, neigh_top_gridsize = 0, neigh_bottom_gridsize = 0;
int neigh_left_nx = 0, neigh_right_nx = 0, neigh_top_ny = 0, neigh_bottom_ny = 0;

if (neigh_left != MPI_PROC_NULL) {
    neigh_left_nx = get_block_size(cols, rankx - 1, px);
    neigh_left_gridsize = (ny + 2) * (neigh_left_nx + 2);
}
if (neigh_right != MPI_PROC_NULL) {
    neigh_right_nx = get_block_size(cols, rankx + 1, px);
    neigh_right_gridsize = (ny + 2) * (neigh_right_nx + 2);
}
if (neigh_top != MPI_PROC_NULL) {
    neigh_top_ny = get_block_size(rows, ranky - 1, py);
    neigh_top_gridsize = (neigh_top_ny + 2) * (nx + 2);
}
if (neigh_bottom != MPI_PROC_NULL) {
    neigh_bottom_ny = get_block_size(rows, ranky + 1, py);
    neigh_bottom_gridsize = (neigh_bottom_ny + 2) * (nx + 2);
}
```

- В основном цикле каждый процесс записывает свои граничные строки и столбцы в теневые ячейки соответствующих соседних процессов
- Для правильного вычисления смещений в сегменте памяти соседа текущему процессу необходимо знать размер сетки соседа

Решение двумерного уравнения Лапласа (MPI One-Sided RMA)

```
for (;;) {
    niters++;

    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, 0, win);  /* Lock memory for local load/store operations */
    /* 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]));
        }
    }
    MPI_Win_unlock(rank, win);
```

- MPI_Win_lock(MPI_LOCK_EXCLUSIVE) — ограничивает RMA-доступ к области памяти текущего процесса

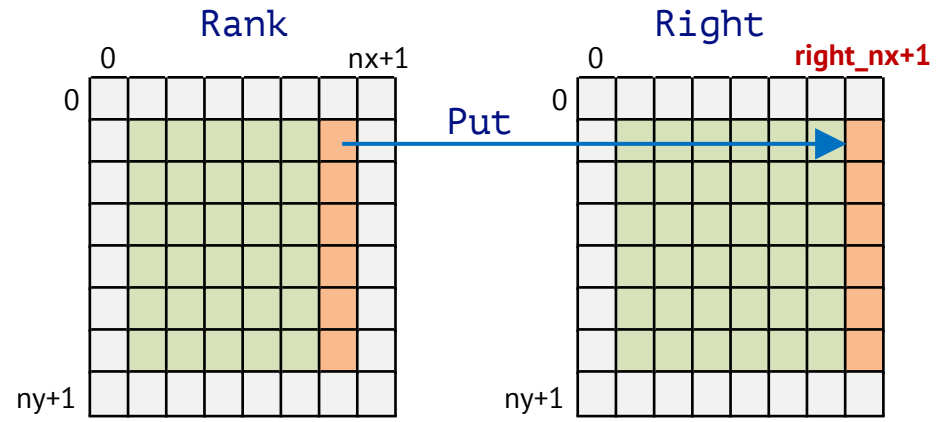
Решение двумерного уравнения Лапласа (MPI One-Sided RMA)

```
/* Swap grids */
double *p = local_grid;
local_grid = local_newgrid;
local_newgrid = p;
shm_grid_idx ^= 1;

MPI_Allreduce(MPI_IN_PLACE, &maxdiff, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
if (maxdiff < EPS) {
    break;
}
```

Решение двумерного уравнения Лапласа (MPI One-Sided RMA)

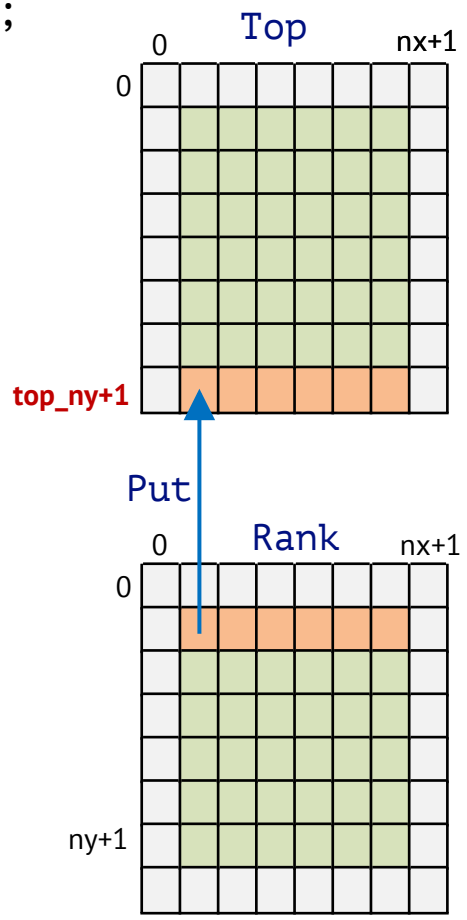
```
/* Halo exchange */
MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
if (neigh_left != MPI_PROC_NULL) {
    /* Put column [1,1] to halo column [1,left_nx+1] of the left neighbour */
    /* Use first or second subgrid according to iteration number */
    MPI_Aint disp = (shm_grid_idx ? neigh_left_gridsize : 0);
    MPI_Put(&local_grid[IND(1, 1)], 1, col, neigh_left,
            disp + (neigh_left_nx + 2) + neigh_left_nx + 1, 1, col, win);
}

if (neigh_right != MPI_PROC_NULL) {
    /* Put column [1,nx] to halo column [1,0] of the right neighbour */
    MPI_Aint disp = (shm_grid_idx ? neigh_right_gridsize : 0);
    MPI_Put(&local_grid[IND(1, nx)], 1, col, neigh_right,
            disp + (neigh_right_nx + 2), 1, col, win);
}
```

- MPI_Win_fence — ожидает завершения RMA-операций
- Процесс записывает свой левый граничный столбец в столбец теневых ячеек соседа справа
- Ширина сетки соседа справа может отличаться от ширины сетеки текущего процесса



Решение двумерного уравнения Лапласа (MPI One-Sided RMA)

```
if (neigh_top != MPI_PROC_NULL) {
    /* Put row [1,1] to halo row [ny+1,1] the top neighbour */
    MPI_Aint disp = (shm_grid_idx ? neigh_top_gridsize : 0);
    MPI_Put(&local_grid[IND(1, 1)], nx, MPI_DOUBLE, neigh_top,
            disp + (neigh_top_ny + 1) * (nx + 2) + 1, nx, MPI_DOUBLE, win);
}

if (neigh_bottom != MPI_PROC_NULL) {
    /* Put row [ny,1] to halo row [0,1] the bottom neighbour */
    MPI_Aint disp = (shm_grid_idx ? neigh_bottom_gridsize : 0);
    MPI_Put(&local_grid[IND(ny, 1)], nx, MPI_DOUBLE, neigh_bottom,
            disp + 1, nx, MPI_DOUBLE, win);
}

MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
```



Решение двумерного уравнения Лапласа
MPI Neighborhood
Collective Communication

Решение двумерного уравнения Лапласа
Neighborhood Collective Operations

- MPI Process Topology задает структуру коммуникационного графа, многомерную нумерацию процессов, но не предоставляет коммункационные операции, учитывающие структуру графа
- **Коллективные операции со смежными процессами** (Neighborhood Collective Communication) учитывают структуру коммуникационного графа коммуникатора (blocking, nonblocking persistent)

```
int MPI_Neighbor_alltoallw(const void *sendbuf, const int sendcounts[],
                           const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
                           void *recvbuf, const int recvcounts[],
                           const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
                           MPI_Comm comm)
```

- `sendcounts[out_degree] = {}`
- `recvcounts[in_degree] = {}`

- `MPI_Neighbor_alltoallw, MPI_Ineighbor_allgather, MPI_Neighbor_alltoallw_init`

Решение двумерного уравнения Лапласа (Neighborhood Colls)

```
#define DISP(i, j) (((i) * (nx + 2) + (j)) * sizeof(double))

for (;;) {
    niters++;
    // Update interior points
    for (int i = 1; i <= ny; i++)
    ...

    // Check termination condition
    double maxdiff = 0;
    ...

    MPI_Allreduce(MPI_IN_PLACE, &maxdiff, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
    if (maxdiff < EPS)
        break;

    // Halo exchange
    int counts[4] = {1, 1, 1, 1};
    MPI_Datatype types[4] = {col, col, row, row}; /* Left, Right, Top, Bottom */
    /* Displacement in bytes */
    MPI_Aint sdispls[4] = {DISP(1, 1), DISP(1, nx), DISP(1, 1), DISP(ny, 1)};
    MPI_Aint rdispls[4] = {DISP(1, 0), DISP(1, nx + 1), DISP(0, 1), DISP(ny + 1, 1)};
    MPI_Neighbor_alltoallw(local_grid, counts, sdispls, types,
                           local_grid, counts, rdispls, types, cartcomm);
}
```

cartcomm (6 ranks)
dim[0]=3, dim[1]=2