

Суперкомпьютерное моделирование и технологии

Лекция Технология параллельного программирования MPI. Часть 3.

.

Попова Нина Николаевна

доцент кафедры СКИ

popova@cs.msu.su

11 ноября 2022 г.

Тема

- Виртуальные топологии MPI
- Параллельный алгоритм решения задачи Дирихле

Тема

- Виртуальные топологии MPI

Понятие коммутатора MPI

- Коммутатор - управляющий объект, представляющий группу процессов, которые могут взаимодействовать друг с другом
- Специальный MPI-тип `MPI_Comm`

Предопределенные коммутаторы

- 3 предопределенных коммутатора:
 - MPI_COMM_WORLD
 - MPI_COMM_NULL
 - MPI_COMM_SELF
- Только MPI_COMM_WORLD (из перечисленных выше) используется для передачи сообщений

Использование MPI_COMM_WORLD

- Содержит все доступные на момент старта программы процессы
- Обеспечивает начальное коммуникационное пространство
- Простые программы часто используют только MPI_COMM_WORLD
- Сложные программы дублируют и производят действия с копиями MPI_COMM_WORLD

Использование MPI_COMM_NULL

- Нет реализации такого коммуникатора
- Не может быть использован как параметр ни в одной из функций
- Может использоваться как начальное значение коммуникатора
- Возвращается в качестве результата в некоторых функциях
- Возвращается как значение после операции освобождения коммуникатора

Использование MPI_COMM_SELF

- Содержит только локальный процесс
- Обычно не используется для передачи сообщений
- Содержит информацию:
 - кэшированные атрибуты, соответствующие процессу
 - предоставление единственного входа для определенных ВЫЗОВОВ

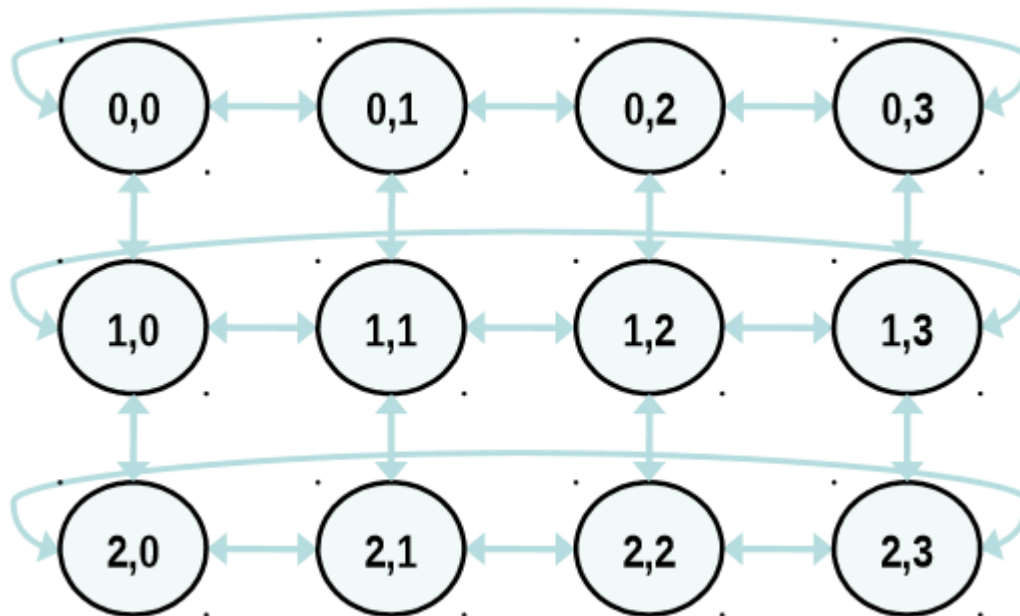
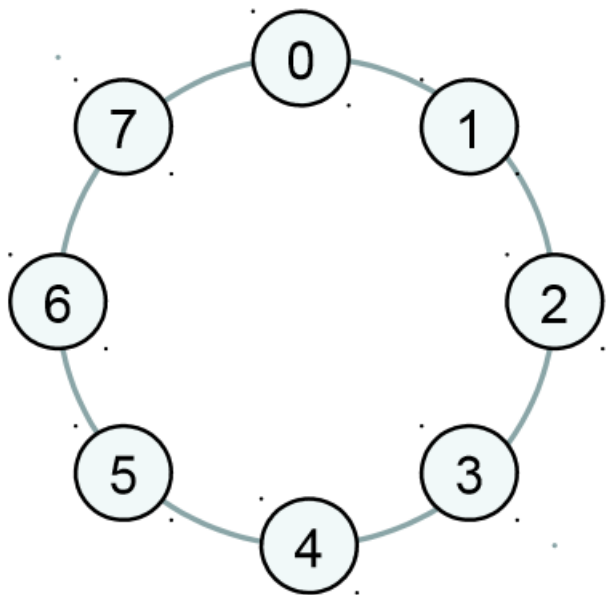
Виртуальная топология MPI

- Во многих параллельных приложениях линейная нумерация процессов не соответствует логической структуре задачи. Часто в задачах процессы упорядочиваются в логические структуры 2D или 3D-решетки. Виртуальные топологии MPI предоставляют удобный механизм именования процессов в группе внутри коммутатора.
- Необходимо четко различать виртуальную топологию MPI и физическую топологию системы. Виртуальная топология MPI является машинно-независимой характеристикой приложения. Mapping (отображение виртуальной топологии на физическую топологию системы) является машинно-зависимым процессом и выполняется пользователем с помощью изменения настроек на целевой системе. находится вне компетенции MPI (<http://www.mpi-forum.org>). Удачный выбор виртуальной топологии для заданной физической топологии системы может существенно улучшить выполнение коммуникаций на целевой системе.

Виртуальные топологии

- Удобный способ именования процессов
- Упрощение написания параллельных программ
- Оптимизация передач
- Возможность выбора топологии, соответствующей логической структуре задачи

Пример виртуальных топологий



Типы виртуальных топологий

- Декартовская (многомерная решетка)
- Графовая

Функции виртуальной топологии «многомерные решетки»

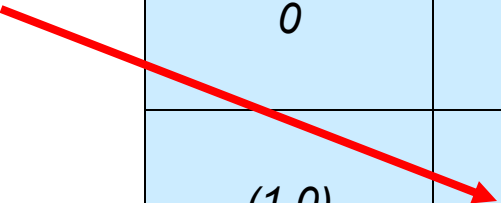
- Основные функции:
 - MPI_CART_CREATE
 - MPI_DIMS_CREATE
 - MPI_CART_COORDS
 - MPI_CART_RANK
 - MPI_CART_SUB
 - MPI_CARTDIM_GET
 - MPI_CART_GET
 - MPI_CART_SHIFT

Как использовать виртуальные ТОПОЛОГИИ

- Создание топологии – новый коммуникатор
- MPI обеспечивает “mapping functions”
- Mapping функции вычисляют ранг процессов, базируясь на топологии

2D решетка

- Отображает линейно упорядоченный массив в 2-мерную решетку (2D Cartesian topology),
- Пример: номер 3 адресуется координатами (1,1).
- Каждая клетка представляет элемент 3x2 матрицы.
- Нумерация начинается с 0.
- Нумерация построчная.



$(0,0)$ 0	$(0,1)$ 1
$(1,0)$ 2	$(1,1)$ 3
$(2,0)$ 4	$(2,1)$ 5

Создание виртуальной топологии решетки

```
int MPI_Cart_create (MPI_Comm comm_old, int ndims, int *dims,  
int *periods,int reorder, MPI_Comm *comm_cart)
```

comm_old старый коммуникатор

ndims размерность

Periods логический массив, указывающий на
 циклическое замыкание:

TRUE/FALSE => циклическое замыкание на границе

reorder возможная перенумерация

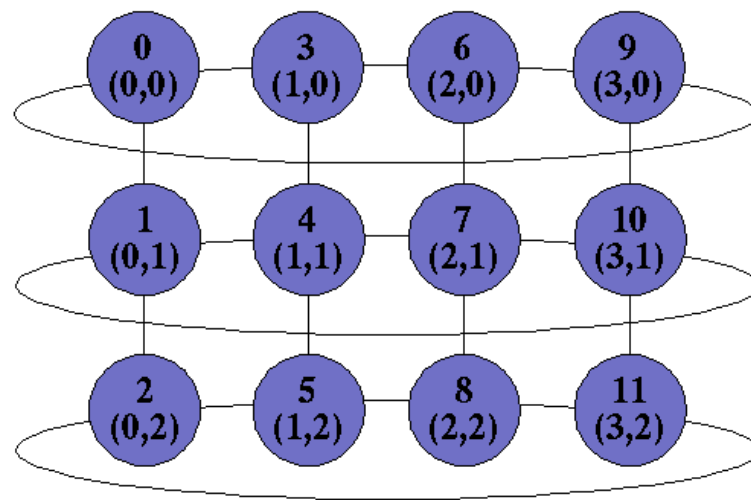
comm_cart новый коммуникатор

Пример виртуальной топологии решетки

```
MPI_Comm vu;  
int dim[2], period[2], reorder;
```

```
dim[0]=4; dim[1]=3;  
period[0]=TRUE; period[1]=FALSE;  
reorder=TRUE;
```

```
MPI_Cart_create(MPI_COMM_WORLD,2,  
               dim,period,reorder,&vu);
```



Координаты процесса в виртуальной решетке

```
int MPI_Cart_coords (  
    MPI_Comm comm,           /* Коммуникатор */  
    int rank,                /* Ранг процесса */  
    int numb_of_dims,        /* Размер решетки */  
    int coords[]             /* координаты процесса в решетке */  
)
```

MPI_CART_RANK

*int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)*

Перевод логических координат процесса в решетке в ранг процесса.

- Если *i*-ое направление размерности периодическое и *i*-ая координата выходит за пределы, значение автоматически сдвигается $0 < \text{coords}(i) < \text{dims}(i)$.
- В противном случае - ошибка

Определение сбалансированного распределения процессов по решетке

int **MPI_Dims_create** (int **nnodes**, int **ndims**, int ***dims**)

nnodes - число процессов

ndims - размер решетки

dims - число элементов по измерениям решетки

- Помогает определить сбалансированное распределение процессов по измерениям решетки.
- Если `dims[i]` – положительное целое, это измерение не будет модифицироваться

dims before call	Function call	dims on return
(0, 0)	MPI_DIMS_CREATE(6, 2, dims)	(3, 2)
(0, 0)	MPI_DIMS_CREATE(7, 2, dims)	(7, 1)
(0, 3, 0)	MPI_DIMS_CREATE(6, 3, dims)	(2, 3, 1)
(0, 3, 0)	MPI_DIMS_CREATE(7, 2, dims)	erroneous call

Пример использования MPI_Dims_create

```
- MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

int dim[3];
dim[0] = 0; // let MPI arrange
dim[1] = 0; // let MPI arrange
dim[2] = 3; // I want exactly 3 planes

MPI_Dims_create(nprocs, 3, dim);

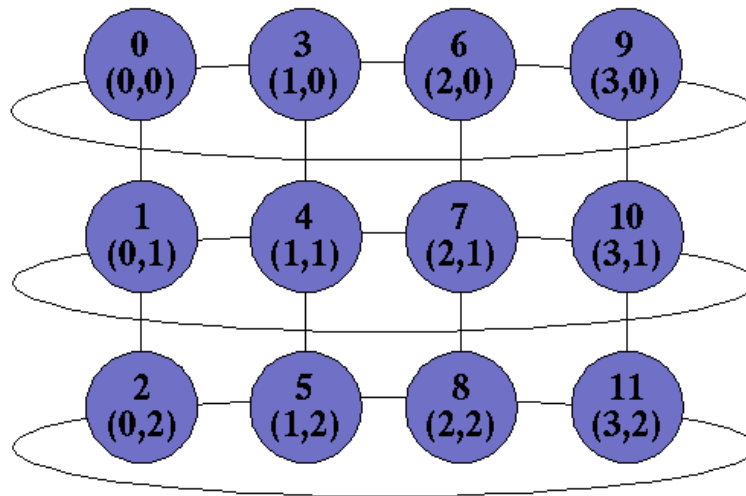
if (dim[0]*dim[1]*dim[2] < nprocs) {
    fprintf(stderr, "WARNING: some processes are not in use!\n")
}

int period[] = {1, 1, 0};
int reorder = 0;

MPI_Cart_create(MPI_COMM_WORLD, 3, dim, period, reorder,
&cube_comm);
```

MPI_CART_SHIFT

- Получение номеров посылающего (**source**) и принимающего (**dest**) процессов в декартовой топологии коммутатора **comm** для осуществления сдвига вдоль измерения **direction** на величину **disp**.



MPI_CART_SHIFT

```
int MPI_Cart_shift( MPI_Comm comm, int direction, int displ, int  
    *source, int *dest )
```

comm - коммуникатор с декартовой топологией;

direction - измерение, вдоль которого выполняется сдвиг;

disp - величина сдвига (может быть как положительной, так и отрицательной; >0 – сдвиг влево/вверх, <0 – сдвиг вправо/вниз)

source - номер процесса, от которого должны быть получены данные;

dest - номер процесса, которому должны быть посланы данные.

MPI_CART_SHIFT

Для периодических измерений осуществляется циклический сдвиг, для непериодических – линейный сдвиг.

Для n -мерной декартовой решетки значение **direction** должно быть в пределах от 0 до $n-1$.

Значения **source** и **dest** можно использовать, например, для обмена функцией **MPI_Sendrecv**.

В случае линейного сдвига в качестве **source** или **dest** можно использовать **MPI_PROC_NULL**.

Пример: Sendrecv в 1D решетке

```
...  
int dim[1], period[1];  
dim[0] = nprocs;  
period[0] = 1;  
MPI_Comm ring_comm;
```

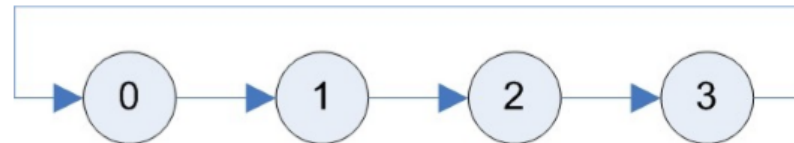
```
MPI_Cart_create(MPI_COMM_WORLD, 1, dim, period, 0, &ring_comm);
```

```
int source, dest;
```

```
MPI_Cart_shift(ring_comm, 0, 1, &source, &dest);
```

```
MPI_Sendrecv(right_boundary, n, MPI_INT, dest, rtag,  
             left_boundary, n, MPI_INT, source, ltag,  
             ring_comm, &status);
```

```
...
```

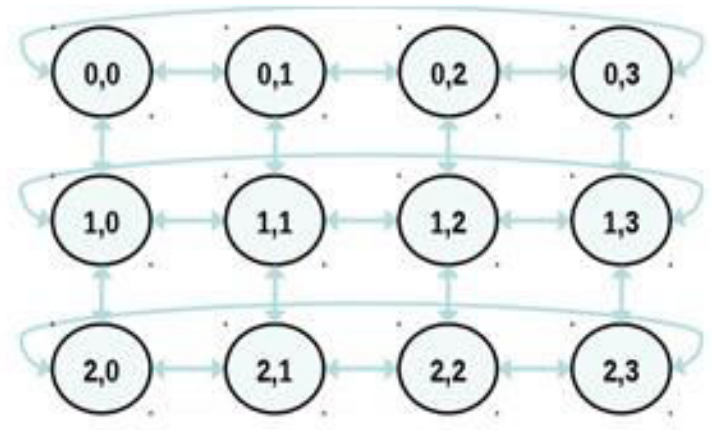


Пример: Sendrecv в 2D решетке

```
...
int dim[] = {4,3};
int period [] = {1,0};
int source, dest;

MPI_Comm grid_comm;

MPI_Cart_create (MPI_COMM_WORLD,2,
    dim, period, 0 , &grid_comm);
for (int dimension = 0; dimension <2; dimension++) {
    for ( int versus = -1; versus < 2; versus+=2) {
        MPI_Cart_shift (grid_comm, dimension, versus, &source, &dest);
        MPI_Sendrecv( buffer, n, MPI_INT, source, stag,
            buffer, n, MPI_INT, dest, dtag,
            grid_comm, &status);
    }
}
```



Создание подрешетки

```
int MPI_Cart_sub (MPI_Comm comm_old,  
                  int remain_dims[], MPI_Comm *new_comm)
```

comm_old – старый коммуникатор

i-ый элемент в **remain_dims** показывает, содержится ли i-ая размерность в подрешетке (true) или нет (false) (вектор логических элементов)

new_comm – новый коммуникатор

```
int dim[] = {2,3,4};
```

```
int remain_dims[]={1,0,1}; // 3 comm with 2x4 processes 2D grid
```

```
...
```

```
int remain_dims[]={0,0,1}; // 6 comm with 4 processes 1D topology
```

MPI_CARTDIM_GET

- Определение числа измерений в решетке.

```
int MPI_Cartdim_get( MPI_Comm comm, int* ndims )
```

- comm коммуникатор (решетка)
- ndims число измерений. Возвращаемый параметр.

Исследование эффективности решения задачи Дирихле для уравнения Лапласа

- Задача Дирихле для уравнения Лапласа (1).

$$\begin{cases} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0, (x, y, z) \in D, \\ u(x, y, z) = g(x, y, z), (x, y, z) \in D^0, \end{cases} \quad (1)$$

где $u(x, y, z)$ - функция, удовлетворяющая в области D уравнению Лапласа и принимающая на границе D^0 области D значения $g(x, y, z)$.

Подобная модель может применяться для описания установившегося течения жидкости, стационарных тепловых полей, процессов теплопередачи с внутренними источниками тепла и деформации упругих пластин.

Постановка задачи. Разностная схема.

- Численный подход к решению задачи (1) основан на замене производных соответствующими конечными разностями (2).

$$\frac{u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}}{h^2} + \frac{u_{i,j+1,k} - 2u_{i,j,k} + u_{i,j-1,k}}{h^2} + \frac{u_{i,j,k+1} - 2u_{i,j,k} + u_{i,j,k-1}}{h^2} = 0, \quad (2)$$

где $h \geq 0$ - шаг сетки, $u_{i,j,k}$ - значение функции $u(x, y, z)$ в точке $x = x_i = ih, i = \overline{0, M+1}, y = y_j = jh, j = \overline{0, N+1}, z = z_k = kh, k = \overline{0, L+1}$, где M, N, L - количество внутренних узлов по каждой координате в области D .

Метод Якоби.

- Одним из простейших методов решения полученной системы (2) является итерационный метод Якоби (3).

$$u_{i,j,k}^n = (u_{i-1,j,k}^{n-1} + u_{i+1,j,k}^{n-1} + u_{i,j+1,k}^{n-1} + u_{i,j-1,k}^{n-1} + u_{i,j,k+1}^{n-1} + u_{i,j,k-1}^{n-1})/6$$
$$u_{i,j,k}^n = g_{i,j,k}, (x, y, z) \in D^0, n = 1, 2, \dots \quad (3)$$

где n - номер итерации.

Метод Якоби – тестовая задача

- Метод Якоби решения разностной задачи для уравнения Лапласа – это типичная **тестовая задача** в параллельном программировании. Он включает в себя общие шаблоны для большинства параллельных задач.
- Метод гарантирует однозначность результатов независимо от способа распараллеливания, но требует использования дополнительного объема памяти. Из-за медленной сходимости метода на практике ему обычно предпочитают более быстрые.

Уравнение Лапласа. 2D вариант.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad x, y \in [0,1] \quad (1)$$

Краевые условия:

$$\begin{aligned} u(x,0) &= \sin(\pi x) & 0 \leq x \leq 1 \\ u(x,1) &= \sin(\pi x)e^{-x} & 0 \leq x \leq 1 \\ u(0,y) &= u(1,y) = 0 & 0 \leq y \leq 1 \end{aligned} \quad (2)$$

Аналитическое решение:

$$u(x,y) = \sin(\pi x)e^{-xy} \quad x, y \in [0,1] \quad (3)$$

Дискретизация уравнения Лапласа

$$u_{i,j}^{n+1} \cong \frac{u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n}{4} \quad i = 1, 2, \dots, m; \quad j = 1, 2, \dots, m \quad (4)$$

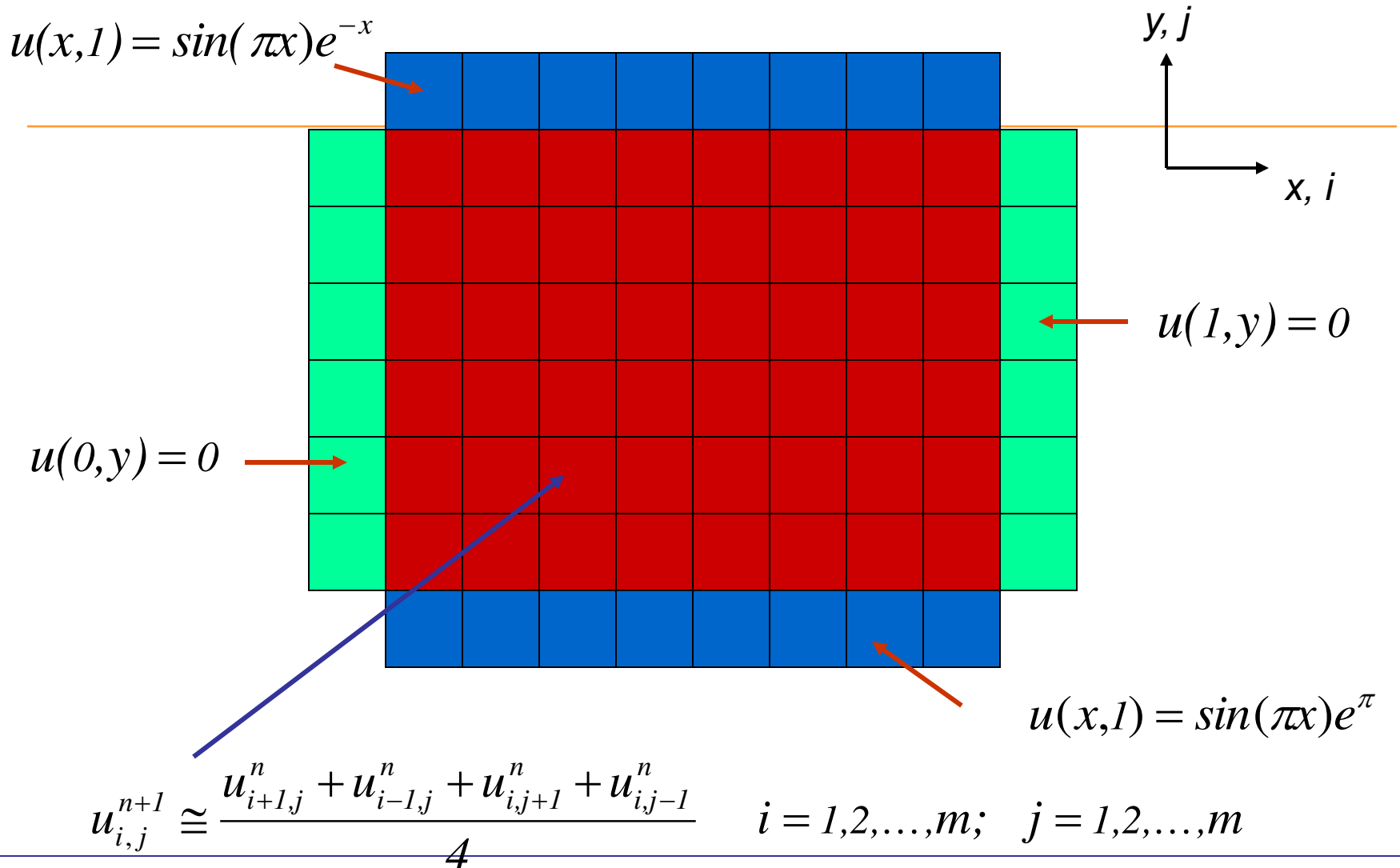
где n и $n+1$ текущий и следующий шаг,

$$\begin{aligned} u_{i,j}^n &= u^n(x_i, y_j) \quad i = 0, 1, 2, \dots, m+1; \quad j = 0, 1, 2, \dots, m+1 \\ &= u^n(i\Delta x, j\Delta y) \end{aligned} \quad (5)$$

Для простоты


$$\Delta x = \Delta y = \frac{1}{m+1}$$

Вычислительная область



5-точечный шаблон

$$u_{i,j}^{n+1} \cong \frac{u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n}{4}$$

 внутренняя область, на которой ищется решение уравнения

  граничная область.

Голубые клетки - неоднородные граничные условия,

Зеленые - однородные

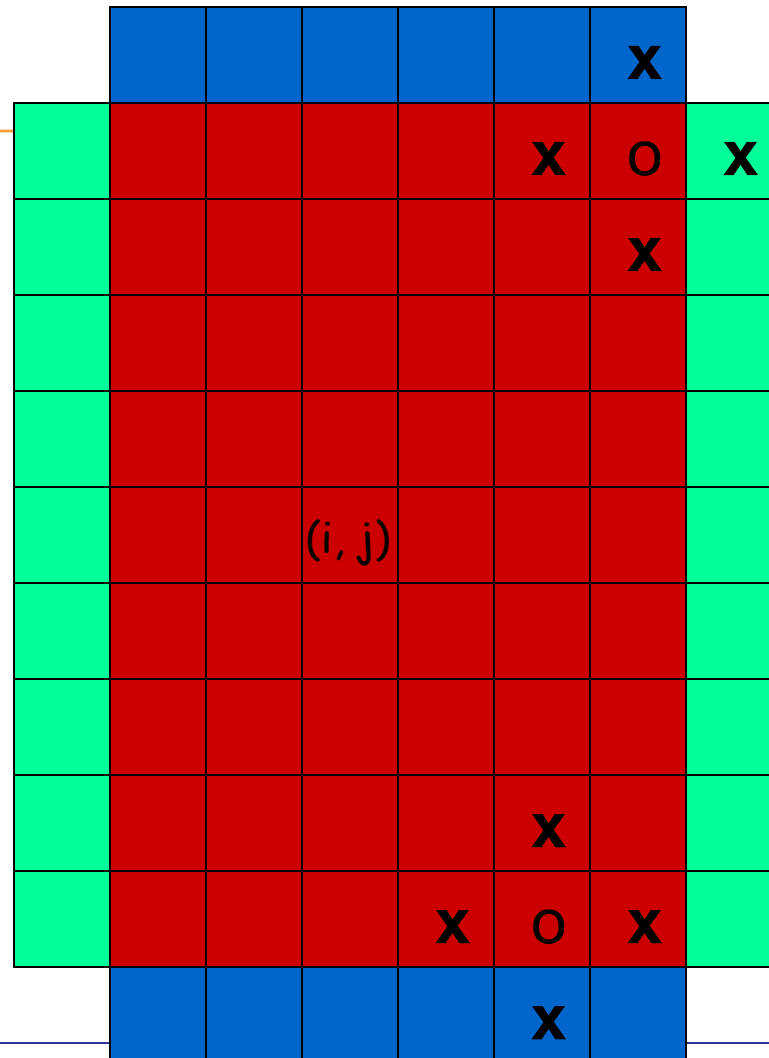


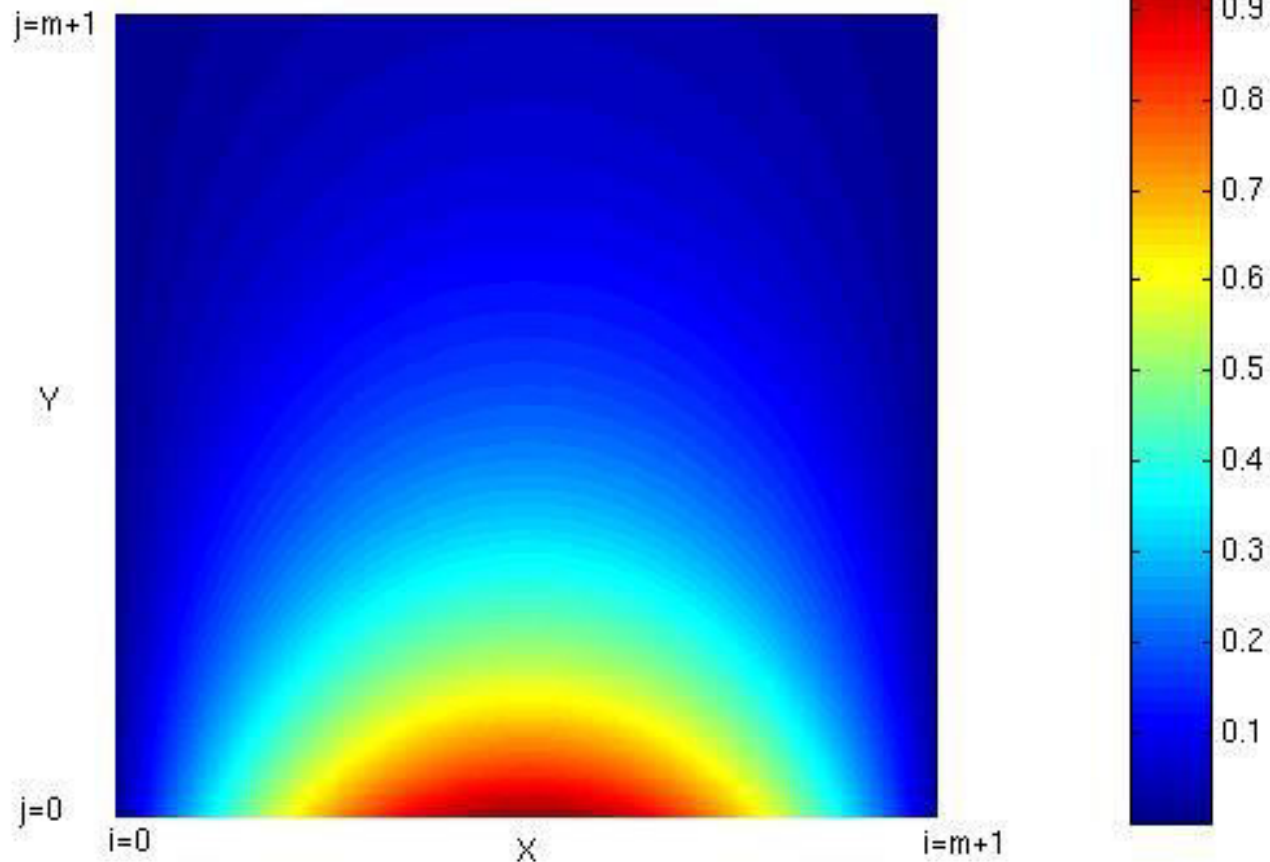
Схема метода Якоби

1. Задание начальных значений u во всех внутренних точках (i,j) в момент времени $n=0$.
2. Используя 5-точечный шаблон вычислить значения во внутренних точках $u_{i,j}^{n+1}$ (i,j) .
3. Завершить процесс, если заданная точность достигнута.
4. Иначе: $u_{i,j}^n = u_{i,j}^{n+1}$ для всех внутренних точек.
5. Перейти на шаг 2.

Это очень простая схема. Медленно сходится, поэтому не используется для решения реальных задач.

Решение (линии уровня)

$\nabla^2 u = 0$ with $u(x,0) = \sin(\pi x)$; $u(x,1) = \sin(\pi x)e^{-\pi}$;
and $u(0,y) = u(1,y) = 0$ yields $u(x,y) = \sin(\pi x)e^{-\pi y}$

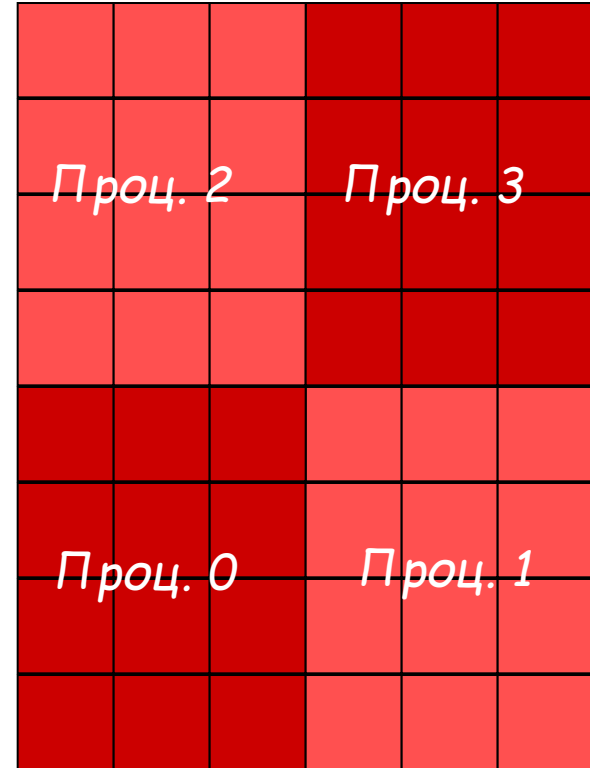


Параллельная реализация

1D Domain Decomposition

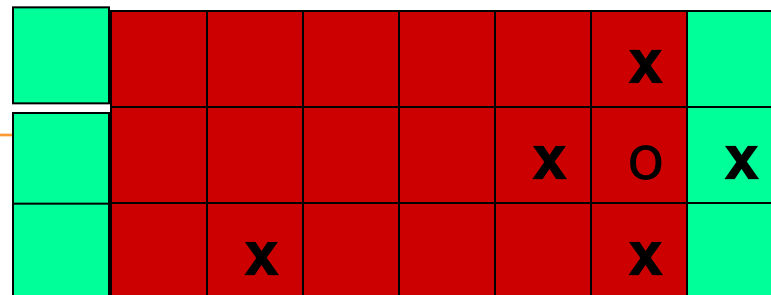


2D Domain Decomposition



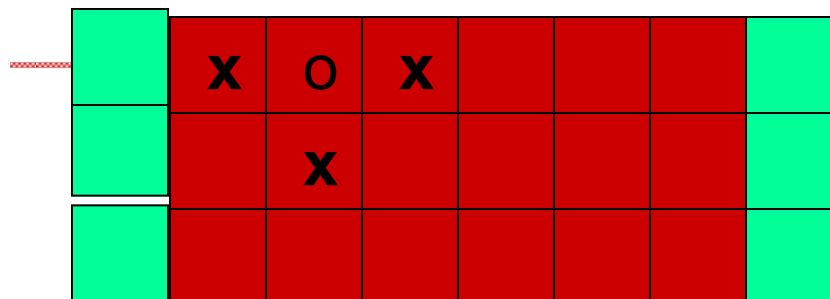
Распределение данных – 1D

5-точечная схема требует значений от соседних потоков

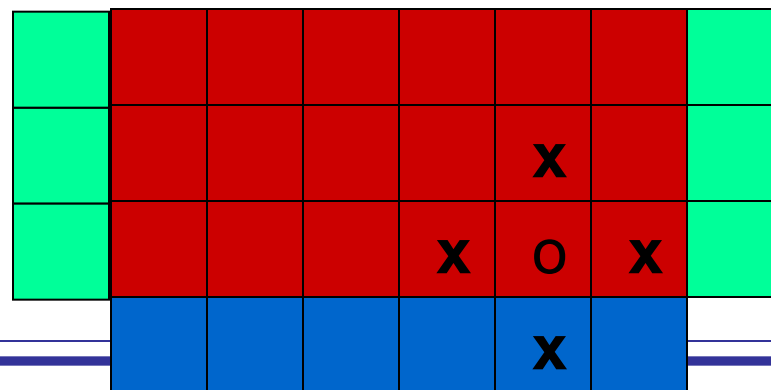


Процесс 2

Необходим обмен данными на границах области

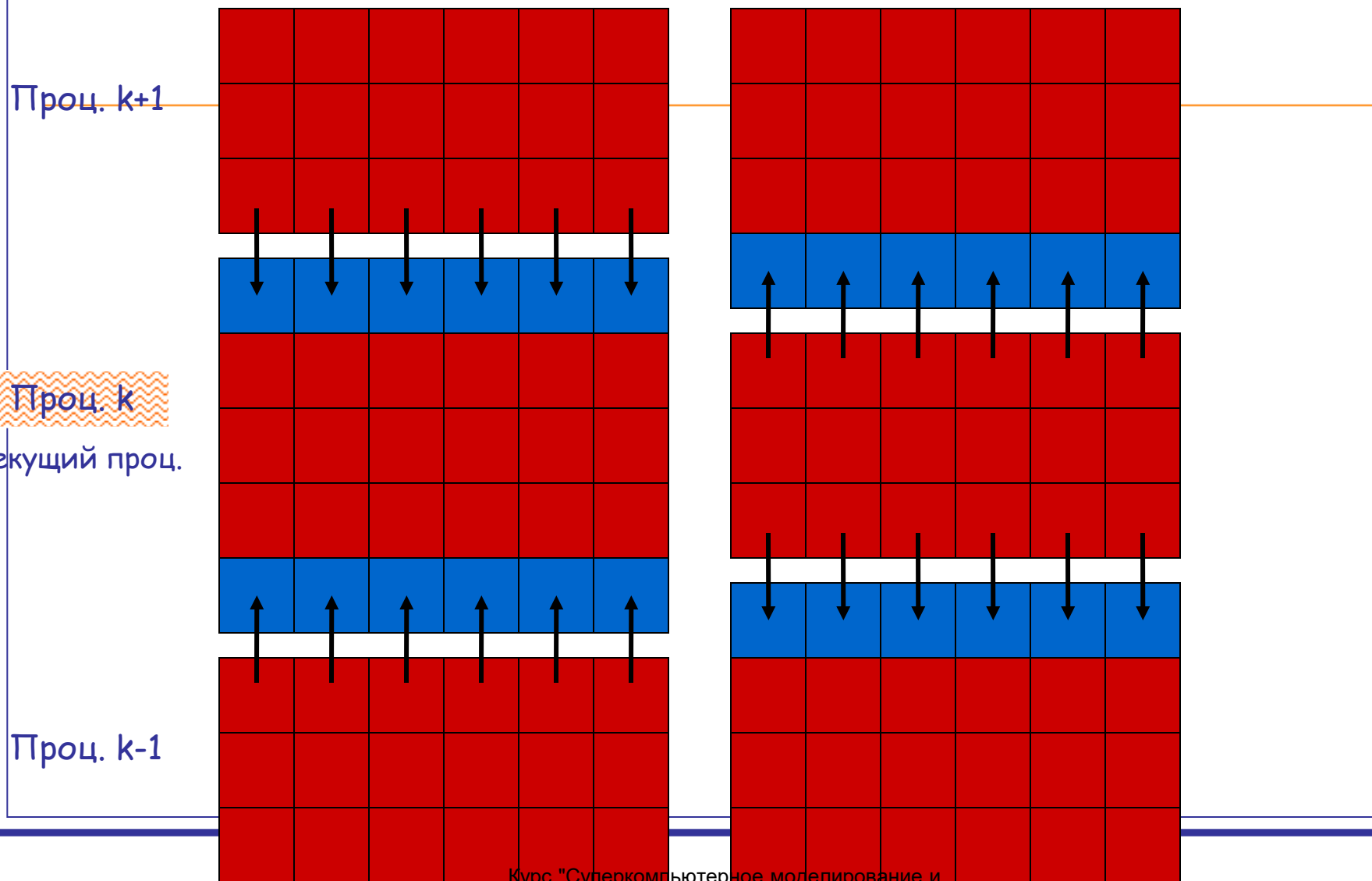


Процесс 1



Процесс 0

Схема передачи данных



3D реализация алгоритма.

- В алгоритме используются два буфера с данными. Один - для значений, полученных на предыдущей итерации, другой - для значений вычисляемых на их основе на новой итерации.
- Перед выполнением алгоритма нужно записать в один из буферов начальное приближение и установить порядок буферов. На каждой итерации алгоритма вычисляется погрешность Δ . Алгоритм останавливается, когда Δ становится меньше некоторого числа или после выполнения заданного числа итераций.
- На процессор, выполняющий обработку какого-либо блока данных, должны быть продублированы боковые грани соседних блоков. Дублирование осуществляется перед началом выполнения каждой очередной итерации метода.

Структура алгоритма.

```
// действия, выполняемые на
каждом процессоре
do {
    // обмен границами с соседями
    // обработка блока
    // вычисление общей
    // погрешности вычислений
while ( $\Delta > \epsilon$ );
```

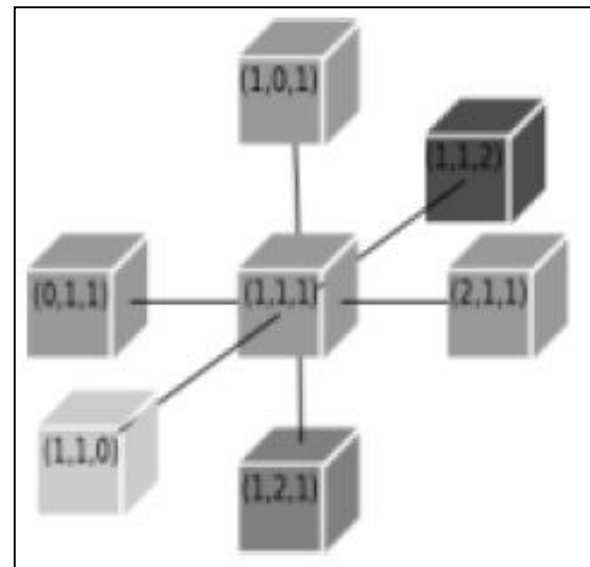
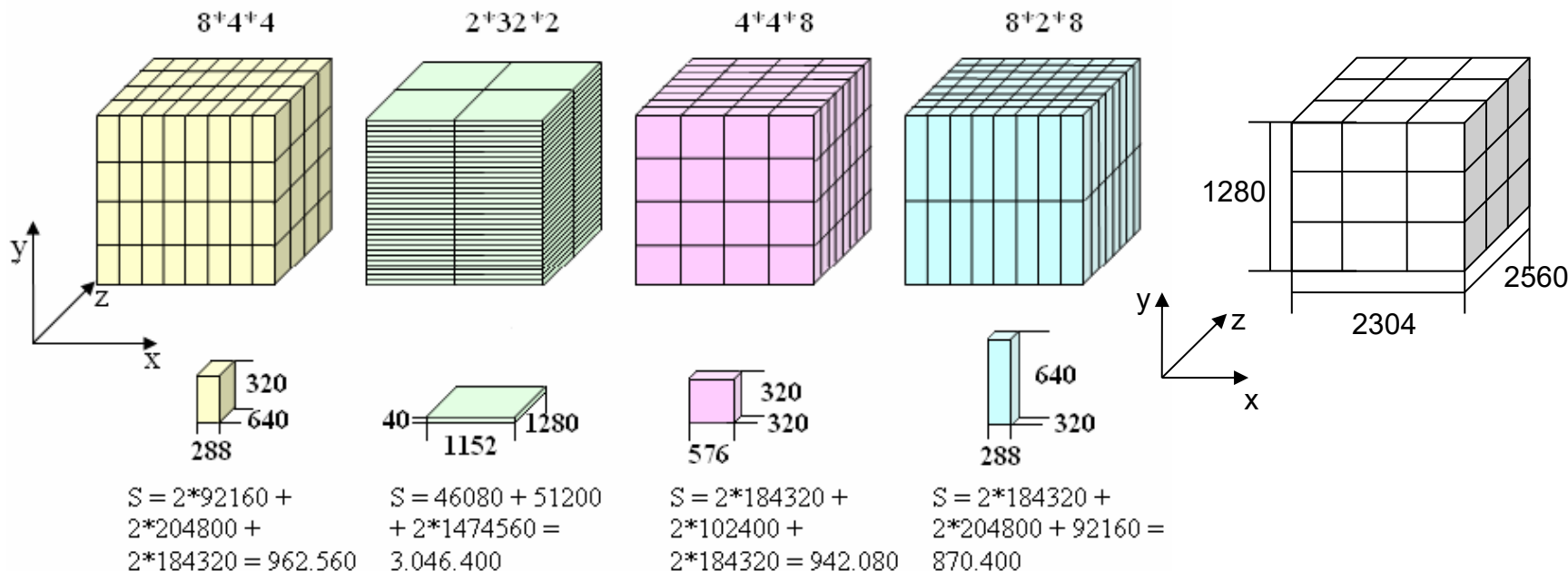


Схема обмена данными:
направления
взаимодействия между
соседними процессами.

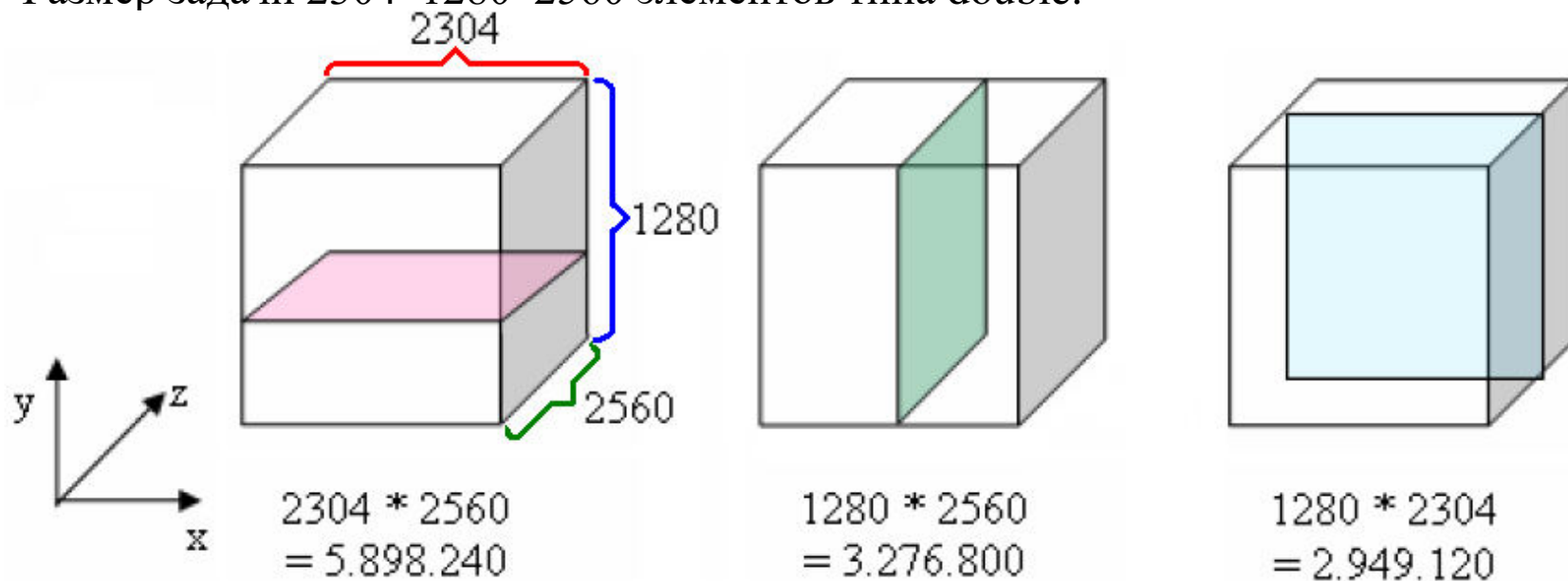
Объем коммуникаций при различных виртуальных топологиях MPI.

- В задаче виртуальная топология влияет, в первую очередь, на объем пересылаемых процессами сообщений, который определяется площадью боковых граней. Размер задачи (2304*1280*2560).



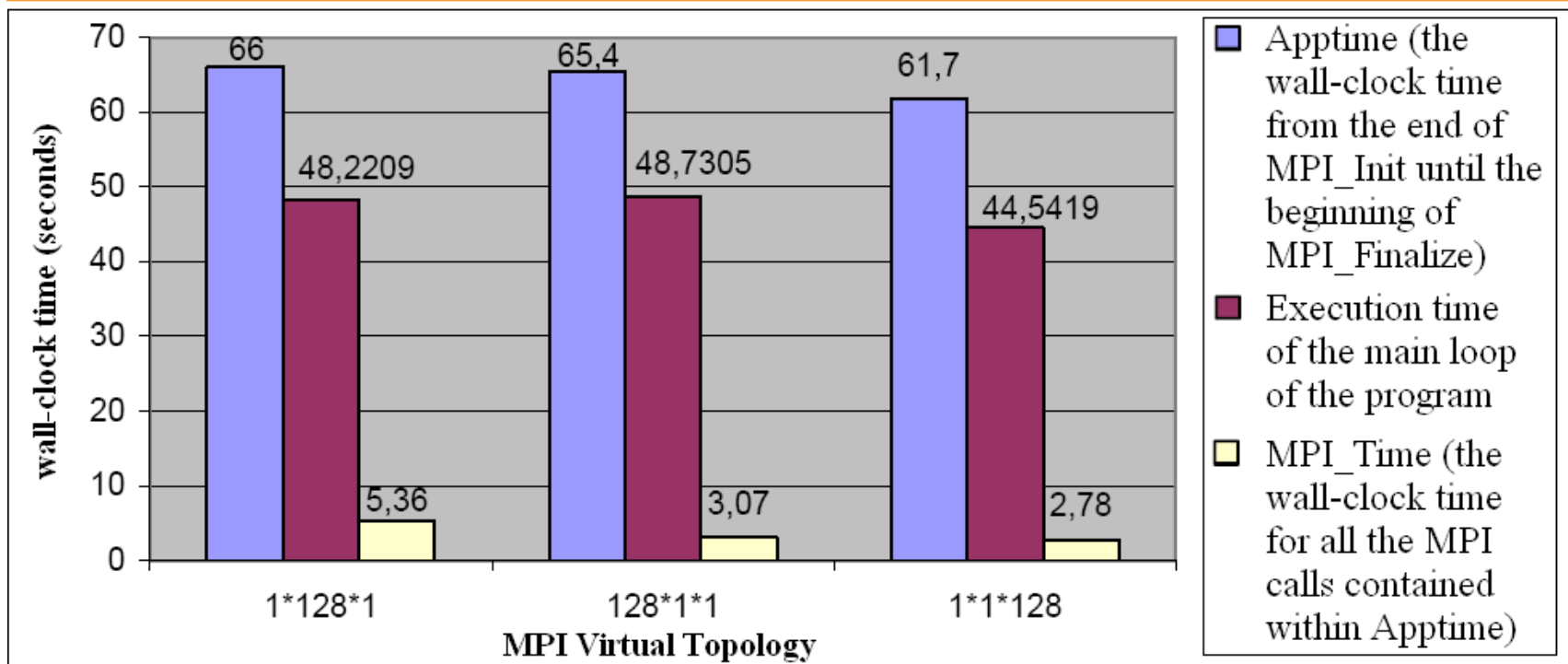
Результаты экспериментов. Распределение данных полосами.

Результаты тестов на партиции из **128** вычислительных узлов (**512 ядер**) BGP.
Размер задачи $2304 \times 1280 \times 2560$ элементов типа double.



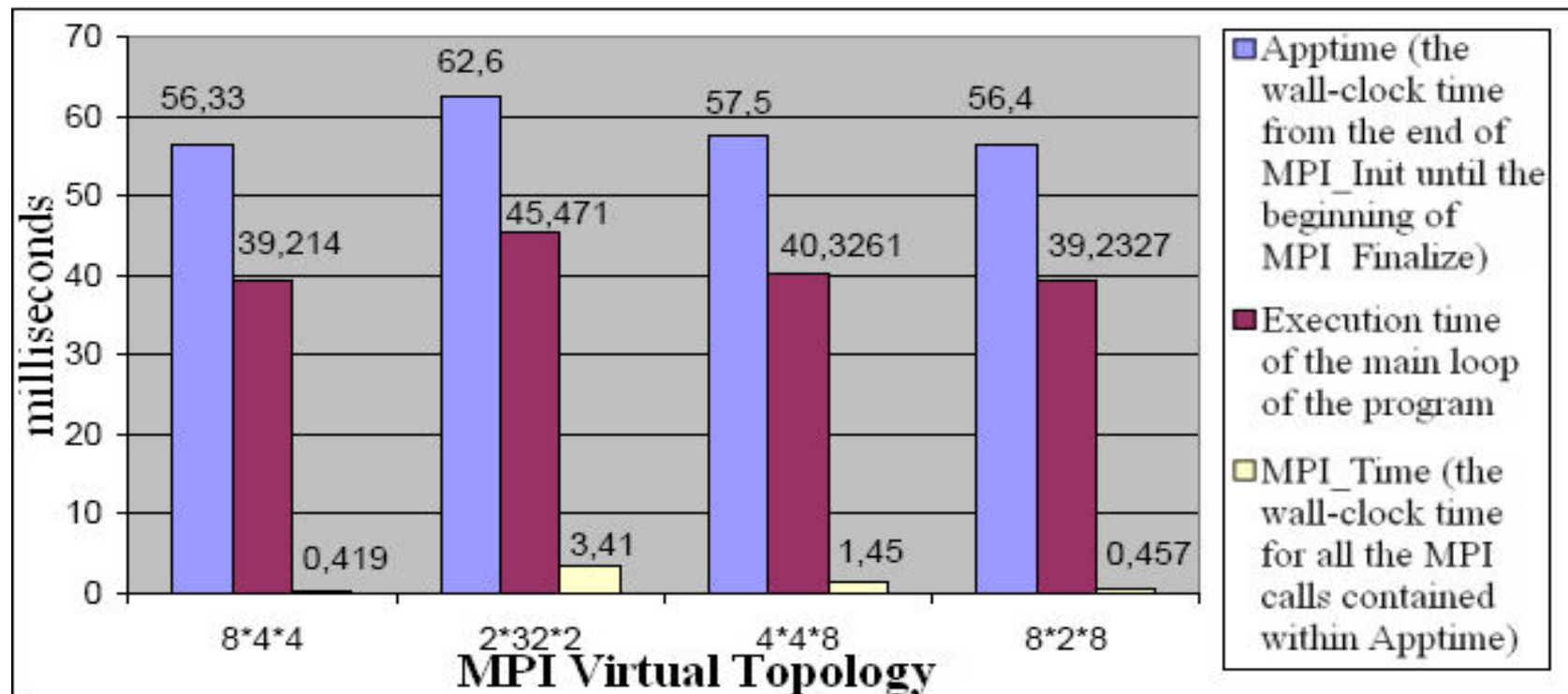
Размер одного сообщения - теневой грани (количество чисел типа double) при различных используемых виртуальных топологиях одномерной решетки.

Результаты экспериментов. Распределение данных полосами.



Среднее время выполнения для **различных виртуальных топологий** в режиме **SMP** на **128 вычислительных узлах** при распределении данных полосами.

Результаты экспериментов. Распределение данных блоками.



Среднее время выполнения для различных виртуальных топологий в режиме SMP на 128 вычислительных узлах при распределении данных блоками.

Фрагменты программы. struct Matrix3.

```
struct Matrix3
{
    int size_x, size_y, size_z;
    double * data;

    Matrix3 (int m, int n, int k);
    /*индекс в одномерном массиве*/
    int ind (int i, int j, int k);
    double & el(int i, int j, int k);
    void print();
    void fill();
    ~Matrix3();
};
```


Фрагменты программы. struct Process.

```
struct Process
{
    Process(int size_x, int size_y, int size_z, int proc_number[], double eps);
    int jacobi_method();//метод Якоби
    void PrintAllData();
    int size, rank;

private:
    MPI_Comm cart_comm;
    int proc_number[3]; //количество процессов в каждом измерении
    int coords[3];
    double eps; //точность
    Matrix3 block1, block2;//текущее значение блока и буфер с предыдущим
    int current_block;//куда записывается значение новой итерации
```

Фрагменты программы. struct Process

```
Matrix2 up, down, left, right, front, back; //границы или теневые грани
//буфера для формирования сообщений
Matrix2 buf_up, buf_down, buf_left, buf_right;
omp_lock_t dmax_lock;
double jacobi_iteration();//итерация метода Якоби
void CreateCommunicator();
void FillData();
void ExchangeData();
};

void Process :: CreateCommunicator()
{
    int Periods[3] = {0, 0, 0}; //коммуникатор - трехмерная решетка
    MPI_Cart_create(MPI_COMM_WORLD, 3, proc_number, Periods, 0,
                    &cart_comm);
    MPI_Cart_coords(cart_comm, rank, 3, coords);
}
```

Фрагменты программы. main.

```
int main(int argc, char * argv[]) {  
    int data_size[3];  
    int block_number[3];  
    double eps;  
    AnalyseCommandString(argc, argv, data_size, block_number, eps);  
    MPI_Init(&argc, &argv);  
  
    Process P(data_size[0]/block_number[0], data_size[1]/block_number[1],  
              data_size[2]/block_number[2], block_number, eps);  
    P.PrintAllData();  
    double time1, time2;  
  
    time1 = MPI_Wtime();  
    P.jacobi_method();  
    time2 = MPI_Wtime();  
  
    P.PrintAllData();  
    if (P.rank == 0) cout << time2 - time1 << endl;  
    MPI_Finalize(); }
```

Фрагменты программы. `jacobi_method()`

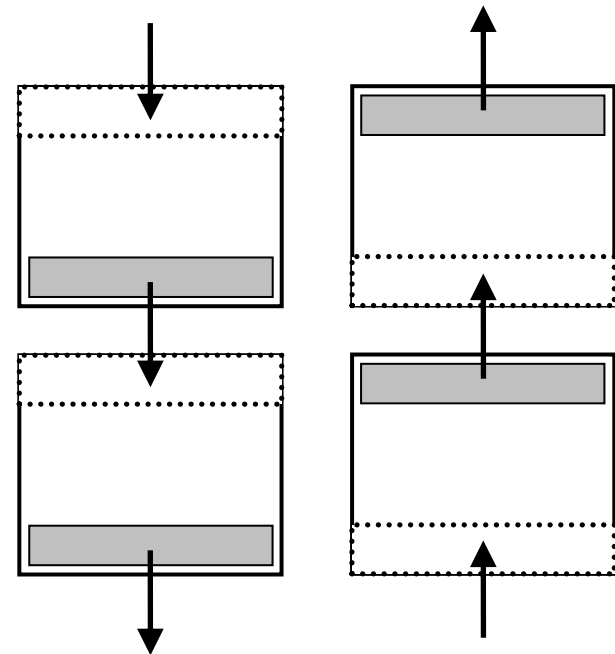
```
int Process :: jacobi_method() {  
    double delta, delta_local;  
    int i = 0;  
    do {  
        i++;  
        current_block = i%2 + 1;  
        ExchangeData(); // обмен теньвыми гранями  
        delta_local = jacobi_iteration();  
        // выполнение редукционной операции  
        MPI_Allreduce(&delta_local, &delta, 1, MPI_DOUBLE,  
        MPI_SUM, MPI_COMM_WORLD);  
        delta = sqrt(delta);  
        current_block = (current_block == 1) ? BLOCK2 : BLOCK1;  
    }  
    while(delta > eps);  
    if (rank == 0) cout << " delta = " << delta << endl;  
    return i; }  
}
```

Обмен данными в программе

- Процессу необходимо обмениваться данными с 6 соседями.
- Из-за того что данные хранятся в памяти не непрерывно, для выполнения межпроцессорных обменов предварительно формируются передаваемые сообщения.

Потом при помощи функций **MPI_Cart_shift** происходит определение ранга процесса, которому надо послать данные, и процесса, от которого данные необходимо получить. После чего при помощи **MPI_Sendrecv** - обмен данными с соседними процессами.

- Всего происходит 6 обменов, по два на каждое измерение.



**Обмен данными вдоль
одного измерения**

Фрагменты программы.

Обмен данными - влево вдоль оси X.

```
MPI_Cart_shift(cart_comm, 0, -1, &rank_recv, &rank_send);
if (coords[0] != 0 && coords[0] != proc_number[0] - 1) {
    MPI_Sendrecv(buf_left.data, size_y * size_z, MPI_DOUBLE,
rank_send, TAG_X, right.data, size_y * size_z, MPI_DOUBLE,
rank_recv, TAG_X, cart_comm, &status);
}
else if (coords[0] == 0 && coords[0] != proc_number[0] - 1){
    MPI_Recv(right.data, size_y * size_z, MPI_DOUBLE, rank_recv,
TAG_X, cart_comm, &status);
}
else if (coords[0] == proc_number[0] - 1 && coords[0] != 0){
    MPI_Send(buf_left.data, size_y * size_z, MPI_DOUBLE,
rank_send, TAG_X, cart_comm);
}
```

Профилирование. Библиотека mpiP. (<http://mpip.sourceforge.net>)

Для профилирования использована библиотека **mpiP**. Она позволяет узнать:

- *mapping*
- *процент от времени выполнения программы, который заняли MPI вызовы*
- *время выполнения отдельных MPI-вызовов, его разброс для различных процессоров и нескольких выполнений одного вызова*
- *статистику для размеров сообщений, передаваемых MPI-вызовами.*
- Использование библиотеки mpiP занимает очень небольшой процент от времени работы приложения, вся информация об MPI-вызовах собирается локально в рамках отдельных MPI-процессов и собирается только на последнем этапе работы приложения для формирования общего отчета.

Файл вывода библиотеки MpiP

```
@ mpiP
@ Command : /gpfs/data/kaelin/jacobi/jac_Hdebug 2304 1280 2560 1 1 128 5
@ Version      : 3.1.2
@ MPIP Build date      : Mar  9 2009, 22:17:50
@ Start time      : 1970 01 01 00:00:40
@ Stop time       : 1970 01 01 00:01:42
@ Timer Used      : gettimeofday
@ MPIP env var     : [null]
@ Collector Rank   : 0
@ Collector PID    : 100
@ Final Output Dir : .
@ Report generation : Single collector task
@ MPI Task Assignment : 0 Rank 0 of 128 <0,0,0,0> R01-M1-N08-J23
@ MPI Task Assignment : 1 Rank 1 of 128 <1,0,0,0> R01-M1-N08-J04
@ MPI Task Assignment : 2 Rank 2 of 128 <2,0,0,0> R01-M1-N08-J12
@ MPI Task Assignment : 3 Rank 3 of 128 <3,0,0,0> R01-M1-N08-J31
```


Файл вывода библиотеки MpiR

```
-----  
@--- MPI Time (seconds) -----  
-----
```

Task	AppTime	MPITime	MPI%
0	61.7	2.78	4.50
1	61.7	2.78	4.50
2	61.7	2.78	4.50
3	61.7	2.78	4.50

AppTime – общее время
работы приложения
MPITime – время,
которое заняли MPI-
вызовы

Вызовы MPI-функций:

```
-----  
@--- Callsites: 34 -----  
-----
```

ID	Lev	File/Address	Line	Parent_Funct	MPI_Call
1	0	0x01002517		[unknown]	Recv
2	0	0x01002477		[unknown]	Send
3	0	0x010029db		[unknown]	Allreduce

Файл вывода библиотеки MpiP

@- Aggregate Time (top twenty, descending, milliseconds)--

Call	Site	Time	App%	MPI%	COV
Sendrecv	31	1.6e+05	2.02	44.92	0.02
Sendrecv	32	1.59e+05	2.02	44.83	0.01
Allreduce	11	1.48e+04	0.19	4.16	0.21

20 MPI-вызовов, занявших наибольшее суммарное (сумма - по всем вызовам и процессам время).

@- Aggregate Sent Message Size (top twenty, descending, bytes)

Call	Site	Count	Total	Avrg	Sent%
Sendrecv	32	2520	5.95e+10	2.36e+07	49.61
Sendrecv	31	2520	5.95e+10	2.36e+07	49.61

20 MPI-вызовов, передавших наибольший суммарный объем сообщений (сумма - по всем вызовам и процессам).

Файл вывода библиотеки MpiP

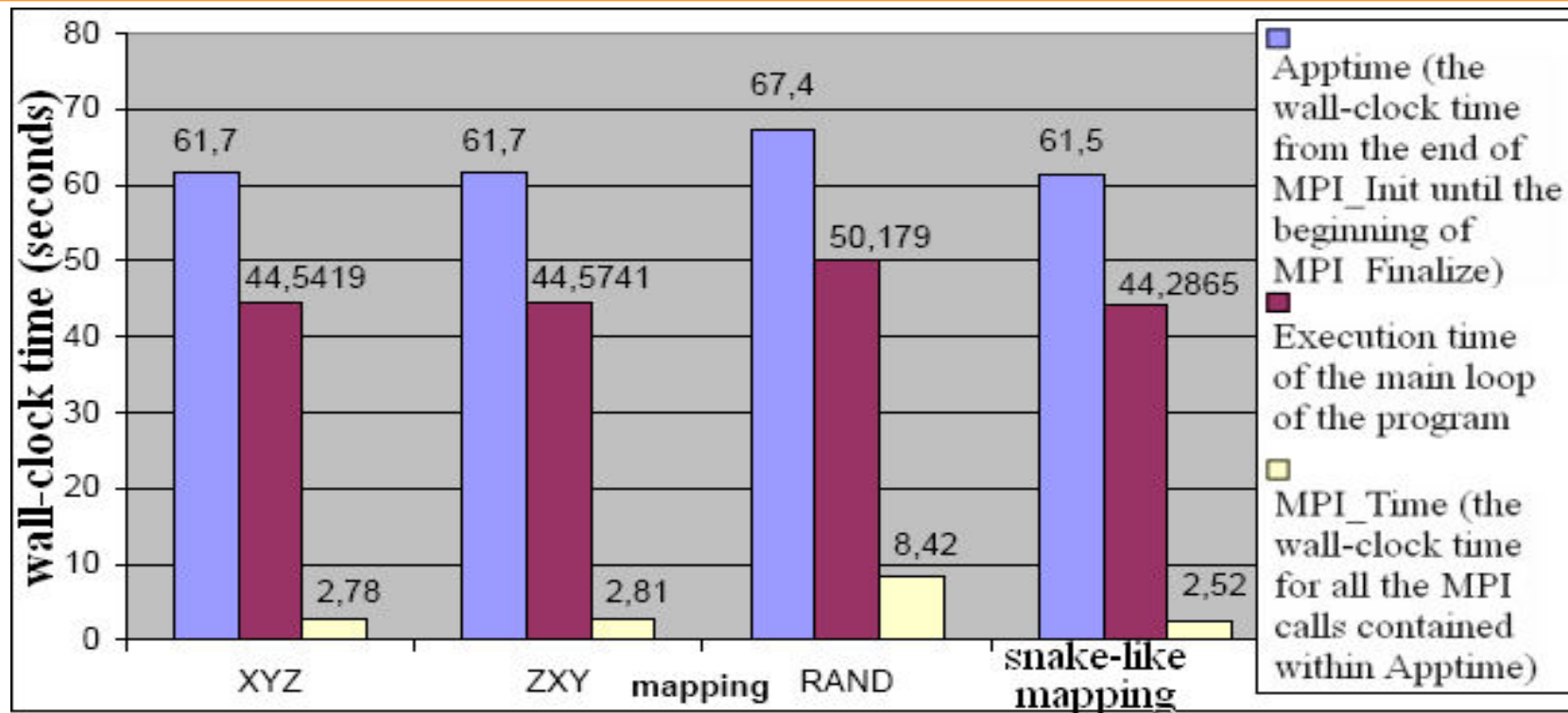
Статистика для времени отдельных MPI-вызовов (аналогичная таблица есть для размеров сообщений).

@--- Callsite Time statistics (all, milliseconds): 3840 ---

Name	Site	Rank	Count	Max	Mean	Min	App%	MPI%
Sendrecv	32	1	20	63.1	63	63	2.04	45.35
Sendrecv	32	2	20	63.7	63	63	2.04	45.35
Sendrecv	32	3	20	63	63	63	2.04	45.33
Sendrecv	32	4	20	63.1	63	63	2.04	45.34
.....
Sendrecv	32	*	2520	187	63.3	62.9	2.02	44.83

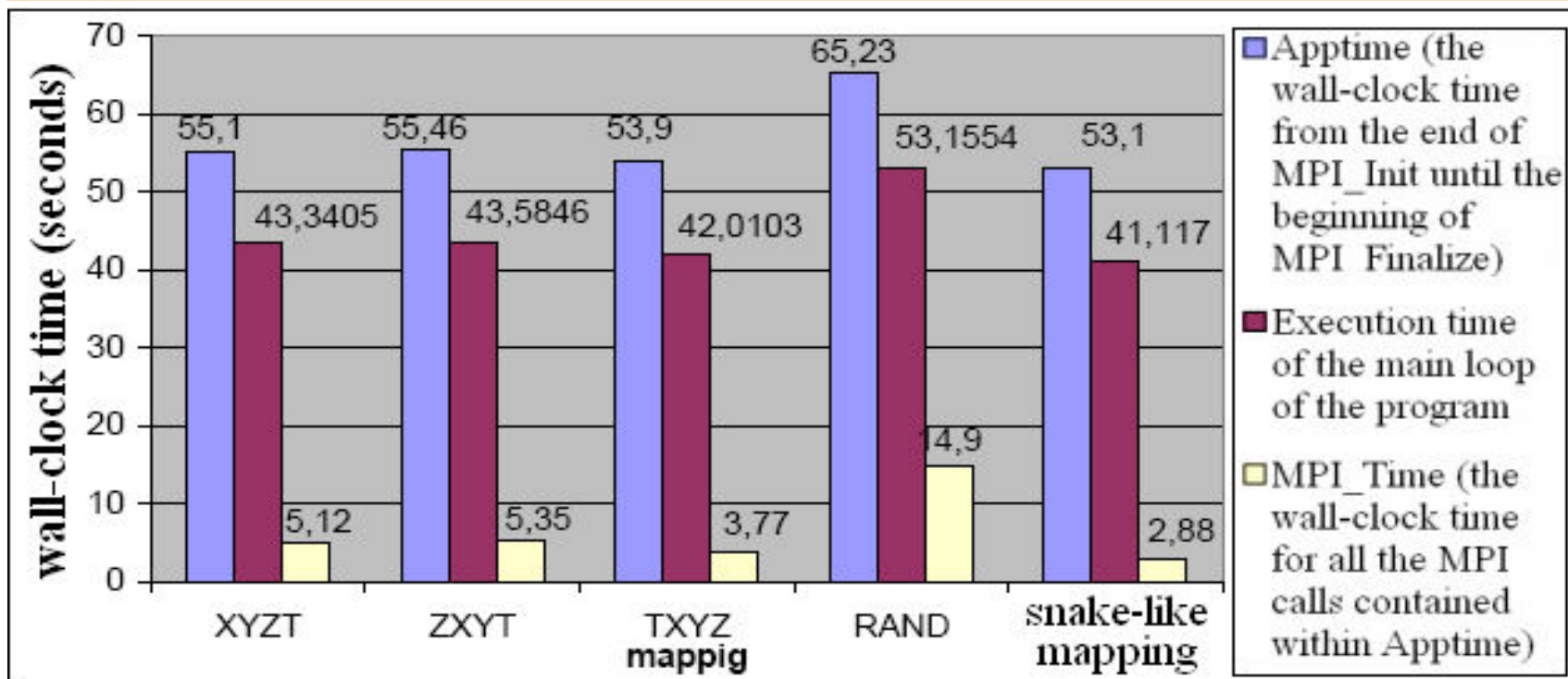
Общая статистика для этого вызова по всем MPI-процессам

Распределение данных полосами. Mapping.



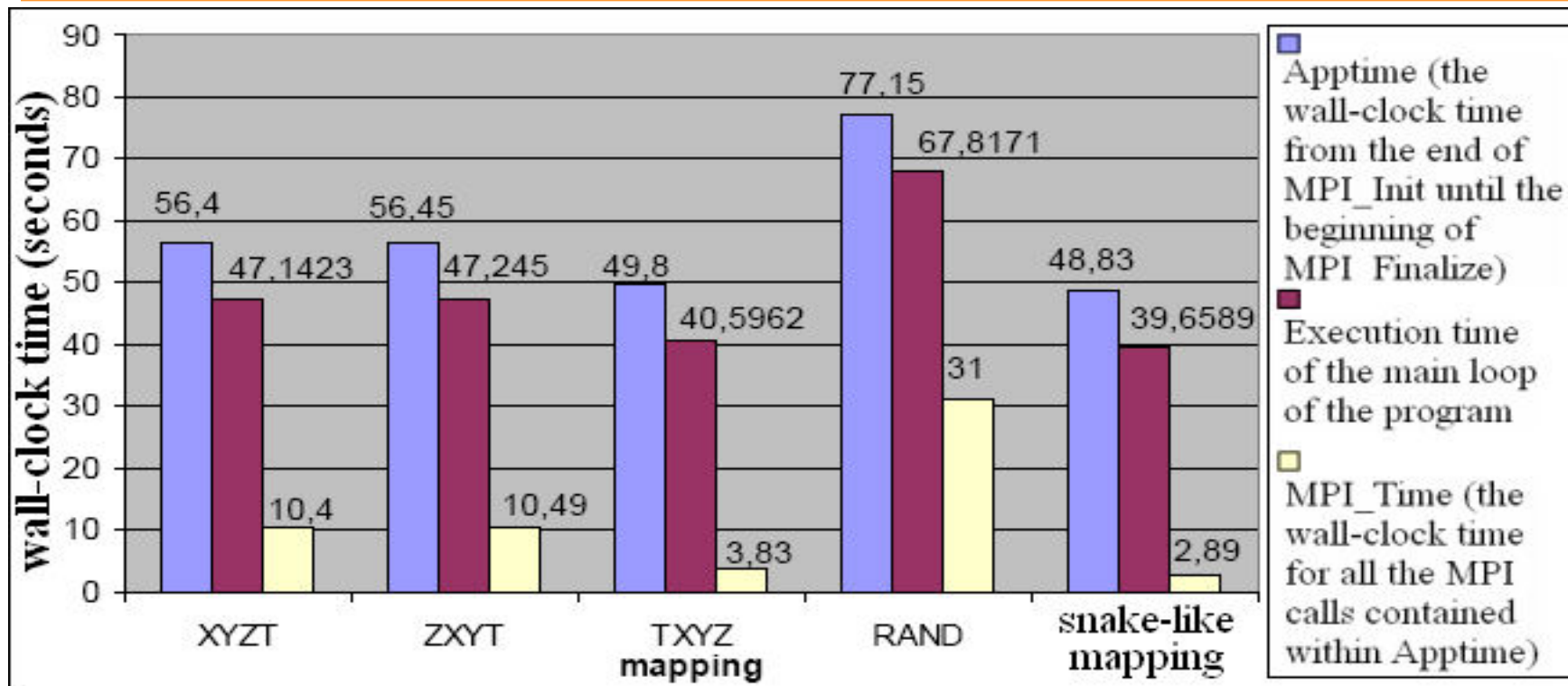
Среднее время выполнения для различного **mappinga** в режиме **SMP** на **128** вычислительных узлах при виртуальной топологии **1*1*128**.

Распределение данных полосами. Mapping.



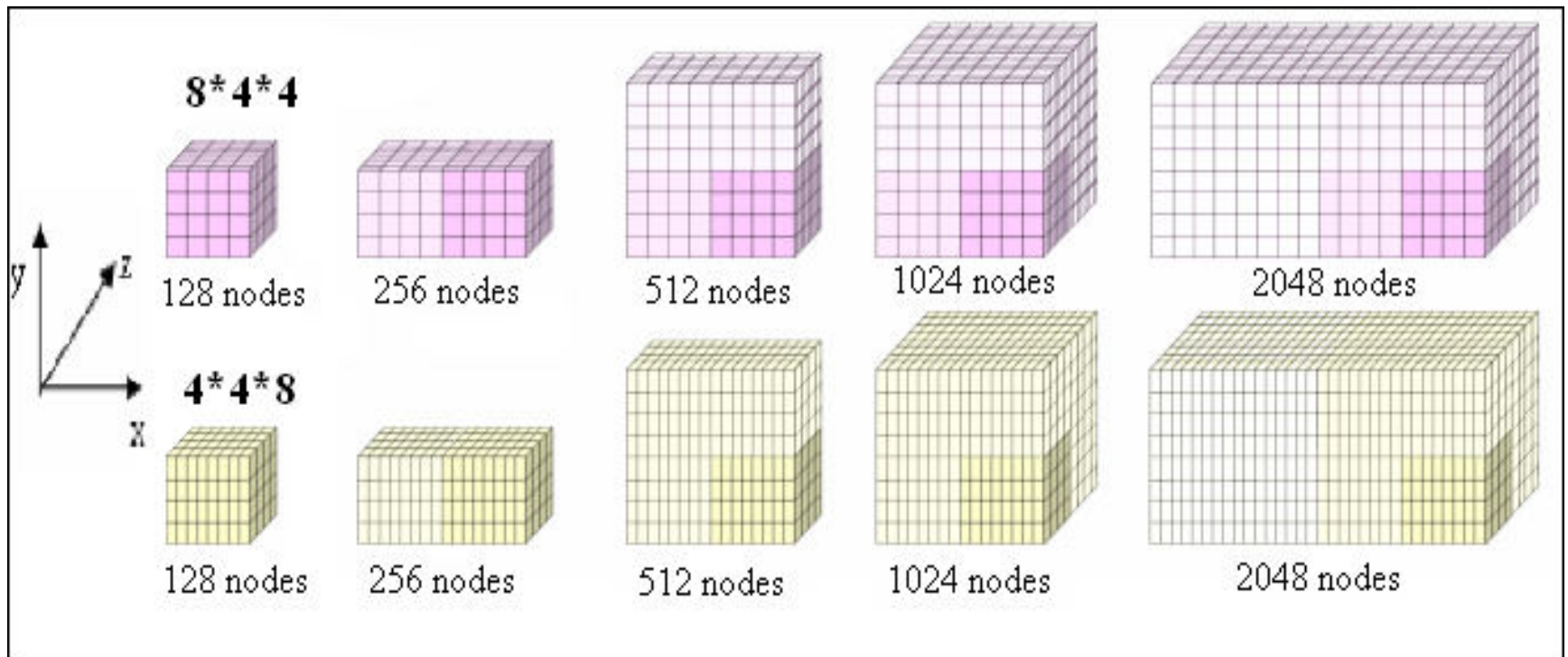
Среднее время выполнения для различного **mappinga** в режиме **DUAL** на **128** вычислительных узлах при виртуальной топологии **1*1*256**.

Распределение данных полосами. Mapping.



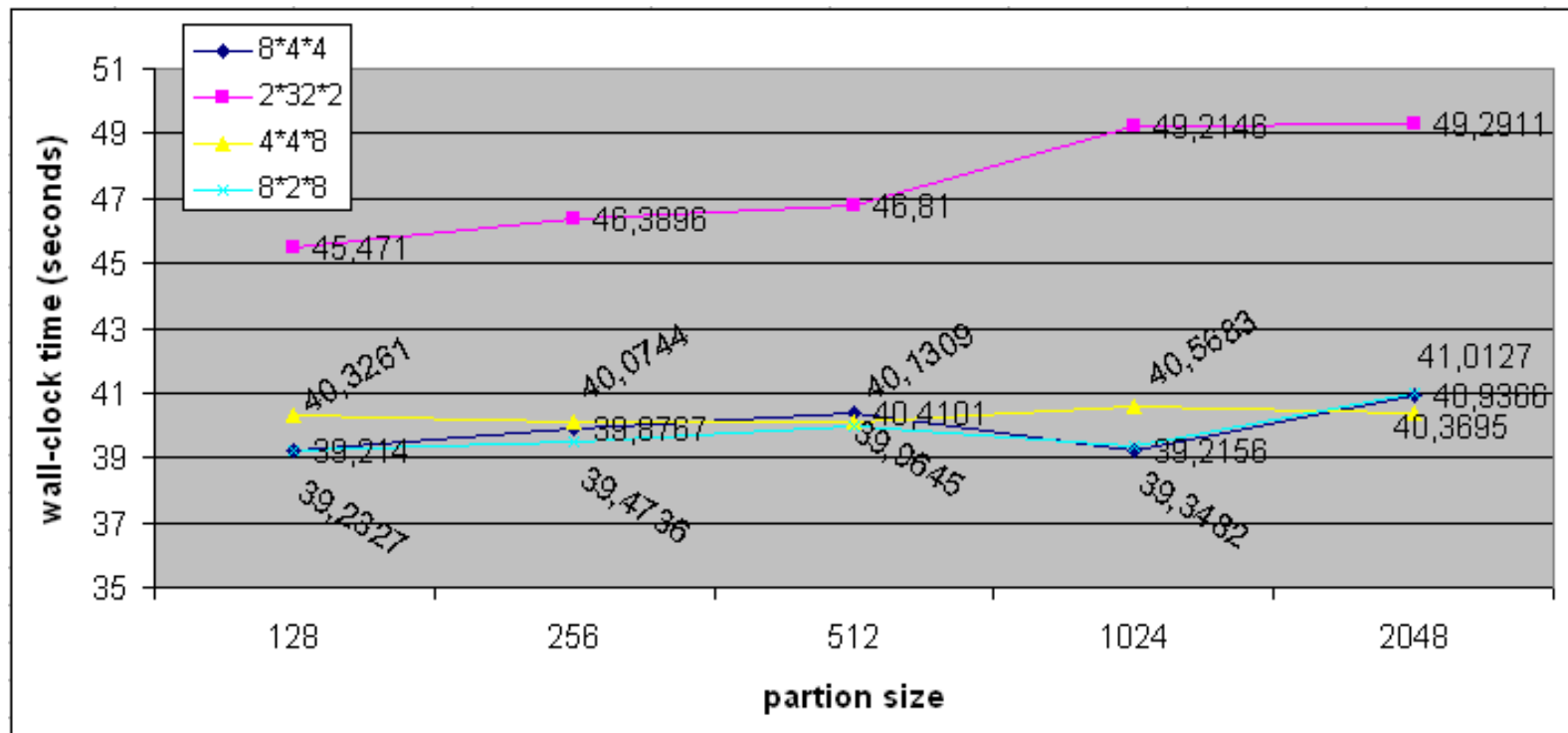
Среднее время выполнения для различного mappinga в режиме VN на 128 вычислительных узлах при виртуальной топологии 1*1*512.

Результаты экспериментов. Блочное распределение. Масштабируемость.



Организация эксперимента по исследованию масштабируемости. Направления увеличения размеров задачи и партиции.

Результаты экспериментов. Блочное распределение. Масштабируемость.



Время выполнения главного цикла программы для различных виртуальных топологий и размеров партии (режим SMP).