Системы и средства параллельного программирования

сентябрь – декабрь 2016 г.

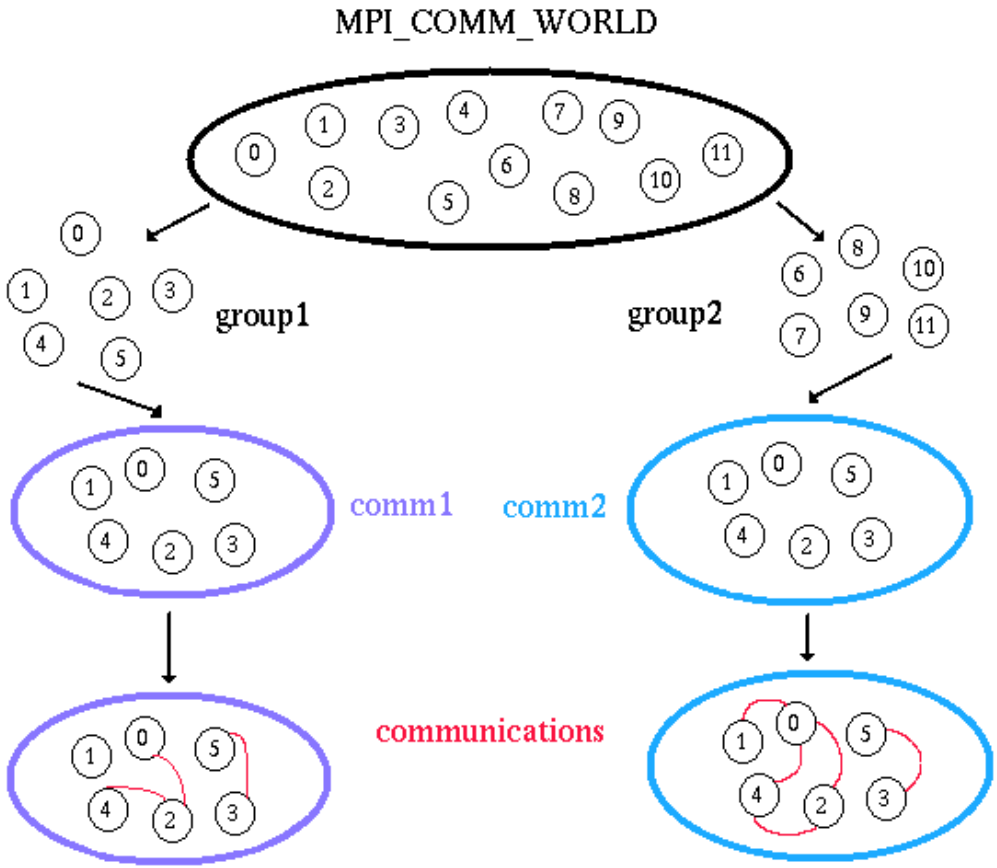
Лектор доцент Н.Н.Попова

Лекция 7
31 октября 2016 г.

Тема

- Группы процессов
- Виртуальные топологии MPI

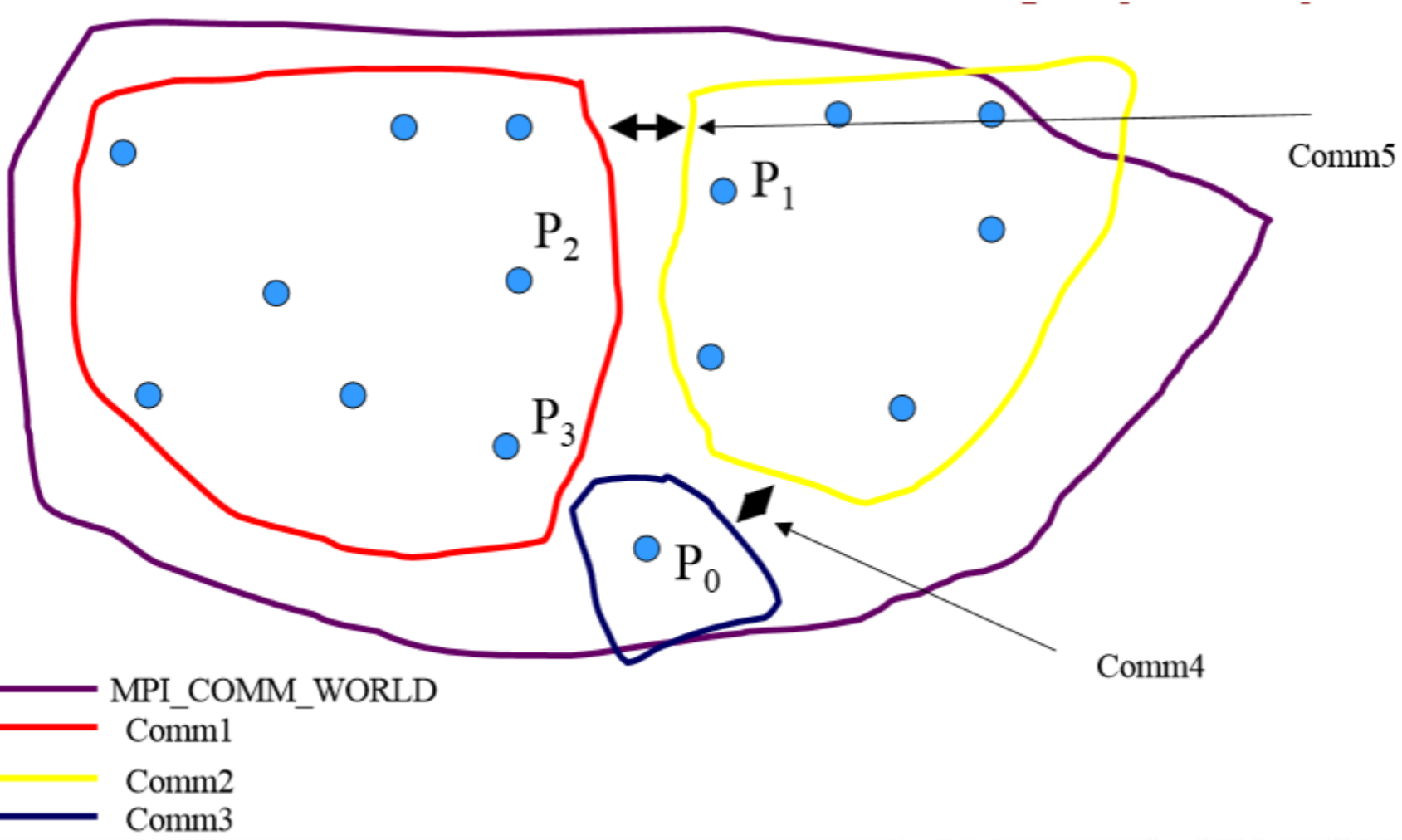
Группы и коммуникаторы



Группы и коммуникаторы

- Группа:
  - Упорядоченное множество процессов
  - Каждый процесс в группе имеет уникальный номер
  - Процесс может принадлежать нескольким группам
    - rank всегда относителен группы
- Коммуникаторы:
  - Все обмены сообщений всегда проходят в рамках коммуникатора
  - С точки зрения программирования группы и коммуникаторы эквивалентны
- Группы и коммуникаторы – динамические объекты, должны создаваться и уничтожаться в процессе работы программы

Типы коммуникаторов



Типы коммуникаторы

- Intercommunicator
  - Обмены (только 2-ухточеченные) между процессами из разных коммуникаторов
- Intracommunicator:
  - Все обмены сообщений всегда проходят в рамках одного коммуникатора

Коммуникатор может быть только одного типа: либо inter, либо intra !

Создание новых коммуникаторов

2 способа создания новых коммуникаторов:

- Использовать функции для работы с группами и коммуникаторами (создать новую группу процессов и по новой группе создать коммуникатор, разделить коммуникатор и т.п.)

- Использовать встроенные в MPI виртуальные топологии

Типичный шаблон работы

1. Извлечение глобальной группы из коммуникатора MPI_COMM_WORLD, используя функцию MPI_Comm_group
2. Формирование новой группы как подмножества глобальной группы, используя MPI_Group_incl или MPI_Group_excl
3. Создание новый коммуникатор для новой группы, используя MPI_Comm_create
4. Определение номера процесса в новом коммуникаторе, используя MPI_Comm_rank
5. Обмен сообщениями, используя функции MPI
6. По окончании освобождение созданных коммуникатора и группы, используя MPI_Comm_free и MPI_Group_free

```
main(int argc, char **argv) {
     int me, count, count2;
     void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
    MPI_Group MPI_GROUP_WORLD, grprem;
    MPI_Comm commslave;
    static int ranks[] = {0};
    MPI_Init(&argc, &argv);
    MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, &grprem);
    MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave);
    if(me != 0){ /* compute on slave */
        MPI_Reduce(send_buf,recv_buff,count, MPI_INT, MPI_SUM, 1,
                                        commslave);
                    }
     /* zero falls through immediately to this reduce, others do later... */
     MPI_Reduce(send_buf2, recv_buff2, count2, MPI_INT, MPI_SUM, 0,
                MPI_COMM_WORLD);
     MPI_Comm_free(&commslave);
     MPI_Group_free(&MPI_GROUP_WORLD);
     MPI_Group_free(&grprem);
     MPI_Finalize();
}
```

Специальные типы MPI

- MPI_Comm

  MPI_COMM_WORLD – коммуникатор для всех процессов приложения.

  MPI_COMM_NULL – значение, используемое для ошибочного коммуникатора.

  MPI_COMM_SELF – коммуникатор, включающий только вызвавший процесс.

- MPI_group

  MPI_GROUP_EMPTY – пустая группа.

  MPI_GROUP_NULL – значение, используемое для ошибочной группы

Количество процессов в группе

- **Размер группы (число процессов в группе)**

```
int MPI_Group_size(MPI_Group comm, int *size)
```

Результат – число процессов

Если указать MPI_GROUP_EMPTY, то size=0

Номер процесса в группе

- **Номер процесса в группе**

```
int MPI_Group_rank(MPI_Group comm, int *rank)
```

Результат – номер процессов или MPI_UNDEFINED

Определение группы по коммуникатору

- **Группа по коммуникатору**

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

Пример:

```
MPI_Group commGroup;
MPI_Comm_group (MPI_COMM_WORLD, &commGroup);
```

Включение процессов в группу

- **Включение процессов в группу**

```
int MPI_Group_incl(MPI_Group comm, ,int n, int *ranks, MPI_Group *newgroup)
```

n – число процессов в новой группе

ranks – номера процессов в группе group, которые будут составлять группу newgroup (выходной параметр);

newgroup – новая группа, составленная из процессов из ranks, в порядке, определенном ranks (выходной параметр).

В случае n=0 ***MPI_Group_incl*** *вернет* ***MPI_GROUP_EMPTY****. Функция может применяться для перенумерации процессов в группе.*

Исключение процессов из группы

- Номер процесса в группе

  *int MPI_Group_excl(MPI_Group oldgroup, , int n, int *ranks, MPI_Group *newgroup)*

  n - число процессов в массиве ranks

  ranks – номера процессов в группе oldgroup, которые будут исключаться из группы oldgroup;

  newgroup – новая группа , не содержащая процессов с номерами из ranks, порядок процессов такой же, как в группе group (выходной параметр).

Каждый из n процессов с номерами из массива ranks должен существовать, иначе функция вернет ошибку. В случае n=0 MPI_Group_excl вернет группу group.

Сравнение групп процессов

```
int MPI_Group_compare(MPI_Group group1,
MPI_Group group2, int *result)
```

MPI_Group_compare возвращает результат сравнения двух групп:

MPI_IDENT – состав и порядок одинаковые в обеих группах;

MPI_SIMILAR – обе группы содержат одинаковые процессы, но их порядок в группах разный;

MPI_UNEQUAL – различные состав и порядок групп.

Трансляция номеров процессов между группами

```
int MPI_Group_translate_ranks ( MPI_Group group_a,
  int n, int *ranks_a, MPI_Group group_b, int
  *ranks_b )
```

Функция возвращает список номеров процессов из группы group_a в их номера в группе group_b

MPI_UNDEFINED возвращается для процессов, которых нет в *group_b*

Создание коммуникатора по группе

int MPI_Comm_create (MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

comm – коммуникатор***;***

group – группа, представляющая собой подмножество процессов, ассоциированное с коммуникатором comm;

newcomm – новый коммуникатор (выходной параметр).

Функия MPI_Comm_create создает новый коммуникатор, с которым ассоциирована группа group. Функция возвращает MPI_COMM_NULL процессам, не входящим в group.

MPI_Comm_create завершится с ошибкой, если не все аргументы group будут одинаковыми в различных вызывающих функцию процессах, или если group не является подмножеством группы, ассоциированной с коммуникатором comm. Вызвать функцию должны все процессы, входящие в comm, даже если они не принадлежат новой группе.

Создание нескольких коммуникаторов

int MPI_Comm_split(MPI_Comm comm, int color, int key,

MPI_Comm *newcomm)

comm - коммуникатор;

color - признак разделения на группы;

key - параметр, определяющий нумерацию в новых коммуникаторах;

newcomm – новый коммуникатор (выходной параметр).

MPI_Comm_split

Функция разбивает все множество процессов, входящих в коммуникатор comm, на непересекающиеся подгруппы - одну подгруппу на каждое значение параметра ***color*** (**неотрицательное** число).

Каждая новая подгруппа содержит все процессы одного цвета. Если в качестве ***color*** указано значение MPI_UNDEFINED, то в newcomm будет возвращено значение MPI_COMM_NULL.

Это **коллективная функция**, но каждый процесс может указывать свои значения для параметров color и key..

MPI_Comm_split

Значение **color** определяет порядок нумерации процессов в новом коммуникаторе:
- процессы с меньшим значением **colo**r получат меньший rank в новом коммуникаторе;

-если значение **color** одинаково, то нумерация процессов в новом коммуникаторе будет определяться порядком следования в исходном коммуникаторе.

Пример

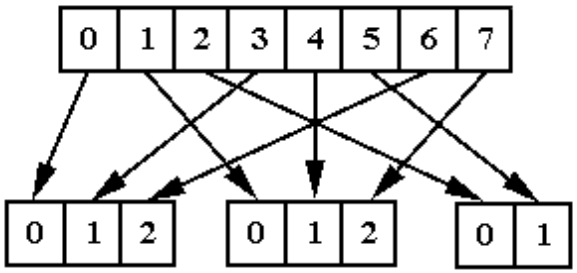
Rank	0	1	2	3	4	5	6	7	8	9	10
Process	a	b	c	d	e	f	g	h	i	j	k
Color	U	3	1	1	3	7	3	3	1	U	3
Key	0	1	2	3	1	9	3	8	1	0	0

Будет создано 3 группы процессов:
{l,c,d}, {k,b,e,g,h}, {f}

Процессы a и j получат значение MPI_COMM_NULL

Пример MPI_Comm_split

```
MPI_comm comm, newcomm;
int myid, color; . . . . . .
MPI_Comm_rank(comm, &myid);
color = myid%3;
MPI_Comm_split(comm, color, myid, &newcomm);
```



Пример: вычисление числа Pi

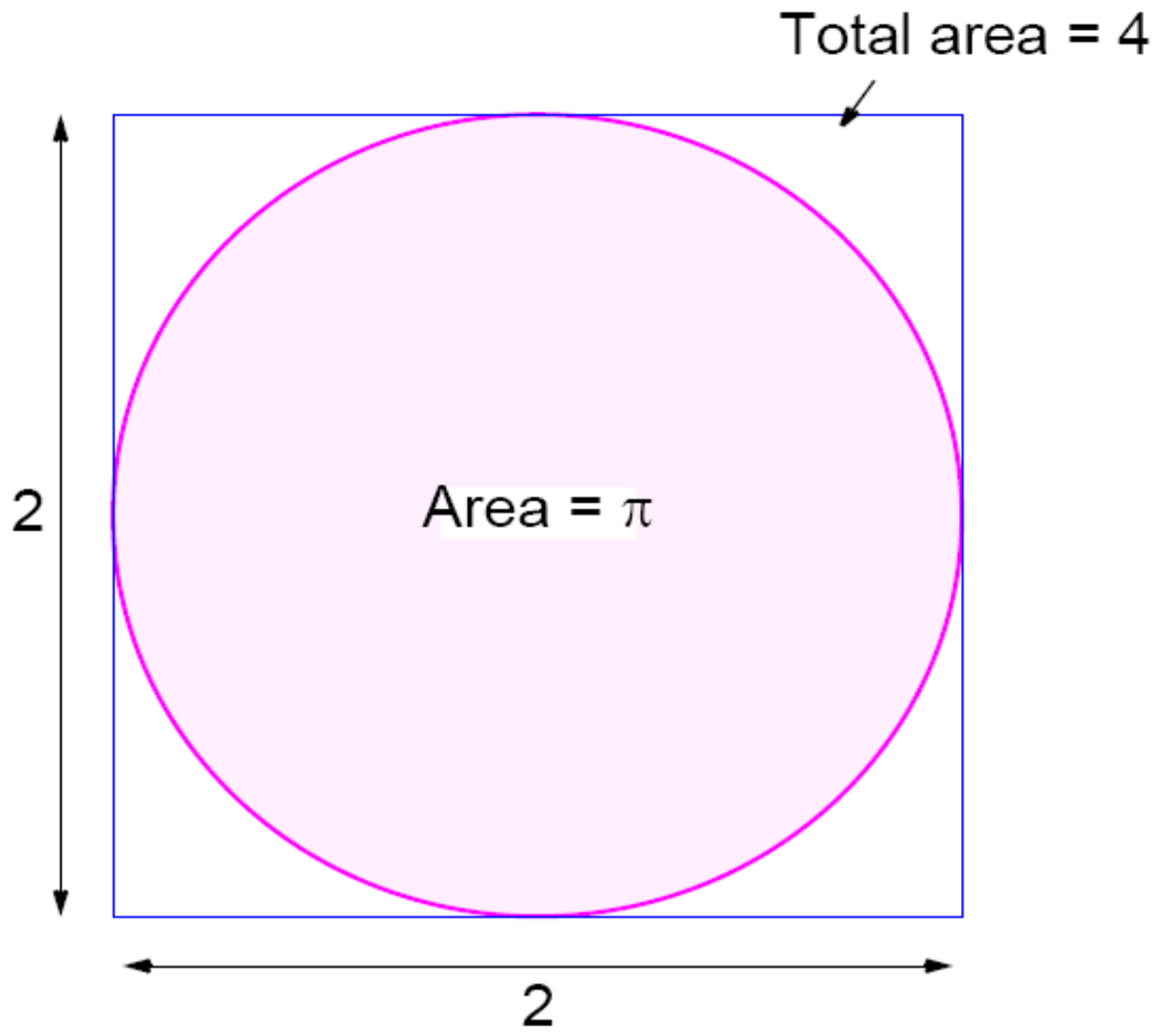
(программа в приложении к лекции)

Окружность вписывается в квадрат 2 x 2. Отношение **площадей:**

$$\frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi(1)^2}{2 \times 2} = \frac{\pi}{4}$$

Точки внутри квадрата выбираются случайно

Отношение числа случайно выбранных точек, попавших в квадрат к числу точек, попавших в круг равно Pi.


Total area = 4
Area = π
2
2

Монте Карло метод для интегрирования произвольных функций

Вычисление случайного значения *x для вычисления* *f*(*x*) и суммы *f*(*x*):

$$\text{Area} = \int_{x_1}^{x_2} f(x)\,dx = \lim_{N \to \infty} \frac{1}{N} \sum_{i=1}^{N} f(x_r)(x_2 - x_1)$$

Xr – случайно сгенерированные значения x, принадлежащие отрезку $[x_1, x_2]$.

Метод Монте Карло полезен для вычисления функций, которые не могут быть проинтегрированы численно (многомерных функций)

Вычисление числа Pi (1)

```
/* compute pi using Monte Carlo method */
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"
#define CHUNKSIZE      1000
/* message tags */
#define REQUEST  1
#define REPLY    2
```

Вычисление числа Pi (2)

```
int main(int argc, char *argv[])
{
    int iter;
    int in, out, i, iters, max, ix, iy, ranks[1], done, temp;
    double x, y, Pi, error, epsilon;
    int numprocs, myid, server, totalin, totalout, workerid;
    int rands[CHUNKSIZE], request;
    MPI_Comm world, workers;
    MPI_Group world_group, worker_group;
    MPI_Status status;
```

Вычисление числа Pi (2)

```
MPI_Init(&argc, &argv);
    world = MPI_COMM_WORLD;
    MPI_Comm_size(world, &numprocs);
    MPI_Comm_rank(world, &myid);
    server = numprocs-1;      /* last proc is server */
    if (myid == 0) {
     if (argc < 2) {
          fprintf(stderr, "Usage: %s epsilon\n", argv[0] );
          MPI_Abort(MPI_COMM_WORLD, 1);
     }
        sscanf( argv[1], "%lf", &epsilon );
    }
    MPI_Bcast(&epsilon, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Вычисление числа Pi (3)

```
MPI_Comm_group(world, &world_group);
  ranks[0] = server;
   MPI_Group_excl(world_group, 1, ranks, &worker_group);
   MPI_Comm_create(world, worker_group, &workers);
   MPI_Group_free(&worker_group);
```

Вычисление числа Pi (4)

```
if (myid == server) {          /* I am the rand server */
    do {
        MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE, REQUEST,
                 world, &status);
        if (request) {
            for (i = 0; i < CHUNKSIZE; ) {
                rands[i] = random();
                if (rands[i] <= INT_MAX) i++;
            }
            MPI_Send(rands, CHUNKSIZE, MPI_INT,
                        status.MPI_SOURCE, REPLY, world);
        }
    }       while(request > 0);
```

Вычисление числа Pi (5)

```
}
    else {                          /* I am a worker process */
     request = 1;
     done = in = out = 0;
     max  = INT_MAX;          /* max int, for normalization */
     MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
     MPI_Comm_rank(workers, &workerid);
     iter = 0;
     while (!done) {
         iter++;
         request = 1;
         MPI_Recv(rands, CHUNKSIZE, MPI_INT, server, REPLY,
                  world, MPI_STATUS_IGNORE);
```

Вычисление числа Pi (6)

```
for (i=0; i<CHUNKSIZE; ) {
        x = (((double) rands[i++])/max) * 2 - 1;
        y = (((double) rands[i++])/max) * 2 - 1;
        if (x*x + y*y < 1.0)
            in++;
        else
            out++;
    }
    MPI_Allreduce(&in, &totalin, 1, MPI_INT, MPI_SUM,
                  workers);
    MPI_Allreduce(&out, &totalout, 1, MPI_INT, MPI_SUM,
                  workers);
 Pi = (4.0*totalin)/(totalin + totalout);
```

Вычисление числа Pi (7)

```
    error = fabs( Pi-3.14159265358979323846264 3);
    done = (error < epsilon || (totalin+totalout) > 100000000);
    request = (done) ? 0 : 1;
    if (myid == 0) {
        printf( "\rpi = %23.20f", Pi );
        MPI_Send(&request, 1, MPI_INT, server, REQUEST,
                 world);
    }   else {
        if (request)
            MPI_Send(&request, 1, MPI_INT, server, REQUEST,
                     world);
    }   }
MPI_Comm_free(&workers);    }
```

Вычисление числа Pi (8)

```
if (myid == 0) {
        printf( "\npoints: %d\nin: %d, out: %d, <ret> to exit\n",
             totalin+totalout, totalin, totalout );
     getchar();
    }
    MPI_Finalize();
    return 0;
}
```

Виртуальные топологии

- ***Топология*** – механизм сопоставления процессам альтернативной схемы адресации. В MPI топологии виртуальны, не связаны с физической топологией сети.

- Два типа топологий:

  - ***декартова*** (прямоугольная решетка произвольной размерности)

  - *топология* ***граф****а*.

Декартова топология

- Логическая топология, определяемая многомерной решеткой.
- Обобщение линейной и матричной топологий на произвольное число измерений.
- Для создания коммуникатора с декартовой топологией используется функция MPI_Cart_create. С помощью этой функции можно создавать топологии с произвольным числом измерений, причем по каждому измерению в отдельности можно накладывать периодические граничные условия.

Виртуальные топологии

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

MPI_CART_CREATE

Создает структуру «прямоугольная решетка» произвольной размерности.

```
int MPI_Cart_create(MPI_Comm old_comm, int ndims, int
  *dim_size, int *periods, int reorder, MPI_Comm *new_comm)
```

.

MPI_CART_CREATE

Параметры			
old_comm	MPI_Comm	Input	Исходный коммуникатор
ndims	int	Input	Число измерений
dim_size	int *	Input	Массив размера ndims для задания числа элементов по каждой из размерностей
periods	int *	Input	Массив размера ndims для задания «периодичности» по каждой из размерностей
reorder	int	Input	Флаг для задания переупорядочивания элементов
new_comm	MPI_Comm *	Output	Новый коммуникатор

MPI_Cart_create

Функция является коллективной, т.е. должна запускаться на всех процессах, входящих в группу коммуникатора comm_old.

Если какие-то процессы не попадают в новую группу, то для них возвращается результат MPI_COMM_NULL. В случае, когда размеры заказываемой сетки больше имеющегося в группе числа процессов, функция завершается аварийно.

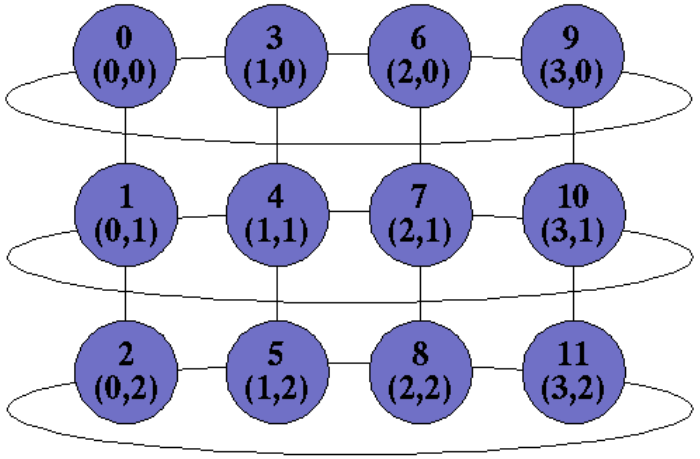
Значение параметра reorder=false означает, что идентификаторы всех процессов в новой группе будут такими же, как в старой группе. Если reorder=true, то MPI будет пытаться перенумеровать их с целью оптимизации коммуникаций.

Пример виртуальной топологии решетка

```
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);
```



Определение оптимальной конфигурации декартовой решетки

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

nnodes - общее число узлов в сетке;

ndims - число измерений;

dims - массив целого типа размерности ndims, в который помещается рекомендуемое число процессов вдоль каждого измерения.

В массив dims должны быть занесены целые неотрицательные числа. Если элементу массива dims[i] присвоено положительное число, то для этой размерности вычисление не производится (число процессов вдоль этого направления считается заданным). Вычисляются только те компоненты dims[i], для которых перед обращением к процедуре были присвоены значения 0. Функция стремится создать максимально равномерное распределение процессов вдоль направлений, выстраивая их по убыванию, т.е. для 12-ти процессов она построит трехмерную сетку 4 x 3 x 1.

MPI_CARTDIM_GET

- Определение числа измерений в решетке.

  int MPI_Cartdim_get(MPI_Comm comm, int* ndims)

- *comm* коммуникатор (решетка)
- *ndims* число измерений

MPI_CARTDIM_GET

Пример

```
/* create column subgrids */
belongs[0] = 1;
belongs[1] = 0;
MPI_Cart_sub(grid_comm, belongs, &col_comm);
/* queries number of dimensions of cartesan grid */
MPI_Cartdim_get(col_comm, &ndims);
```

MPI_CART_COORDS

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,
*int *coords)*

Перевод номера процесса в его координаты в решетке

MPI_CART_COORDS

Параметры			
comm	MPI_Comm	Input	Коммуникатор
rank	int	Input	Ранг процесса
maxdims	int	Input	Число измерений решетки
coords	int *	Output	Координаты в решетке

MPI_CART_RANK

- Используется для перевода логических координат поцесса в решетке в ранг процесса.

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

MPI_CART_RANK

Параметры			
comm	MPI_Comm	Input	Параметры
coords	int *	Input	Массив размера ndims определяющий координаты процесса в решетке
rank	int	Output	Ранг процесса

MPI_CART_SUB

Используется для разделения коммуникатора на подгруппы .

- MPI_CART_SUB создает новый коммуникатор меньшей размерности

*int MPI_Cart_sub(MPI_Comm old_comm, int *belongs, MPI_Comm *new_comm)*

MPI_CART_SUB

			Параметры
old_comm	MPI_Comm	Input	Параметры
belongs	int *	Input	Массив размера ndims, определяющий принадлежность новому коммуникатору new_comm
new_comm	MPI_Comm	Output	Новый коммуникатор решетка

MPI_CART_SUB пример

Предполагаем, что число процессов =6.

Формируем 2D (3x2) решетку. Заполняем матрицу A(i,j)

A(i,j) = (i+1)*10 + j + 1; i=0,1,2; j=0,1

Значения элементов:

A(0,0) = 11, A(2,1) = 32 и т.д..

Создаем подрешетку, используя MPI_CART_SUB.

MPI_CART_SUB пример

```
#include "stdio.h"
#include "mpi.h"
void main(int argc, char *argv[])
{
    int nrow, mcol, i, lastrow, p, root;
    int Iam, id2D, colID, ndim;
    int coords1D[2], coords2D[2], dims[2], aij[1], alocal[3];
    int belongs[2], periods[2], reorder;
    MPI_Comm comm2D, commcol;
    /* Starts MPI processes ... */
    MPI_Init(&argc, &argv);/* starts MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &Iam);
     /* get current process id */
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    /* get number of processes */
```

MPI_CART_SUB пример

```
nrow = 3; mcol = 2; ndim = 2;
root = 0; periods[0] = 1; periods[1] = 0; reorder = 1;
/* create cartesian topology for processes */
dims[0] = nrow;                    /* number of rows */
dims[1] = mcol;                    /* number of columns */
MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, periods, reorder,
&comm2D);
MPI_Comm_rank(comm2D, &id2D);
MPI_Cart_coords(comm2D, id2D, ndim, coords2D);

/* Create 1D column subgrids */
belongs[0] = 1;                    /* this dimension belongs to subgrid */
belongs[1] = 0;
MPI_Cart_sub(comm2D, belongs, &commcol);
MPI_Comm_rank(commcol, &colID);
MPI_Cart_coords(commcol, colID, 1, coords1D);
```

MPI_CART_SUB пример

```
MPI_Barrier(MPI_COMM_WORLD);

/* aij = (i+1)*10 + j + 1; 1 matrix element to each proc */
aij[0] = (coords2D[0]+1)*10 + coords2D[1]+1;

if(Iam == root) {
        printf("\n     MPI_Cart_sub example:");
        printf("\n 3x2 cartesian grid ==> 2 (3x1) column subgrids\n");
        printf("\n   Iam     2D       2D          1D       1D      aij");
        printf("\n  Rank   Rank     coords.     Rank  coords.\n");
}

/* Last element of each column gathers elements of its own column */
for ( i=0; i<=nrow-1; i++) {
        alocal[i] = -1;
}
```

MPI_CART_SUB пример

```
    lastrow = nrow - 1;
    MPI_Gather(aij, 1, MPI_INT, alocal, 1, MPI_INT, lastrow, commcol);

    MPI_Barrier(MPI_COMM_WORLD);

    printf("%6d|%6d|%6d %6d|%6d|%8d|",
    lam,id2D,coords2D[0],coords2D[1],colID,coords1D[0]);
    for (i=0; i<=lastrow; i++) {
            printf("%6d ",alocal[i]);
    }
    printf("\n");

    MPI_Finalize();             /* let MPI finish up ...  */
}
```

MPI_CART_GET

Параметры			
subgrid_comm	MPI_Comm	Input	Коммуникатор
ndims	int	Input	Число измерений
dims	int *	Output	Массив длин по каждому из измерений
periods	int *	Output	Периодичность по направлениям
coords	int *	Output	Координаты вызывающего процесса в решетке (массив размера ndims)

MPI_CART_GET

- Используется для получения информации о параметрах декартовой топологии для заданного коммуникатора

int MPI_Cart_sub(MPI_Comm comm, **int** *maxdims*, **int** **dims*, **int** **periods*, **int** **coords)*

MPI_CART_GET пример

```
/* create Cartesian topology for processes */
dims[0] = nrow;
dims[1] = mcol;
MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, period,
    reorder, &grid_comm);
MPI_Comm_rank(grid_comm, &me);
MPI_Cart_coords(grid_comm, me, ndim, coords);
/* create row subgrids */
belongs[0] = 1;
belongs[1] = 0;
MPI_Cart_sub(grid_comm, belongs, &row_comm);
/* Retrieve subgrid dimensions and other info */
MPI_Cartdim_get(row_comm, &mdims);
MPI_Cart_get(row_comm, mdims, dims, period, row_coords);
```

MPI_CART_SHIFT

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

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

MPI_CART_SHIFT

Параметры

comm	MPI_Comm	Input	Коммуникатор
direction	int	Input	Размерность, по которой будет производиться сдвиг
displ	int	Input	Величина и направление сдвига (<0; >0; or 0)
source	int *	Output	Процесс- источник
dest	int *	Output	Процесс- получатель

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**.

Не является коллективной операцией!

Пример декартовой решетки (send&recv, mesh)

```
MPI_Request reqs[8];
 MPI_Status stats[8];
 MPI_Comm cartcomm;

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
 if (numtasks == SIZE) {
   MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods,
reorder, &cartcomm);
   MPI_Comm_rank(cartcomm, &rank);
   MPI_Cart_coords(cartcomm, rank, 2, coords);
   MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
   MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);
   outbuf = rank;
```

Пример декартовой решетки (send&recv, mesh)

```
for (i=0; i<4; i++) {
      dest = nbrs[i];
      source = nbrs[i];
      MPI_Isend(&outbuf, 1, MPI_INT, dest, tag, MPI_COMM_WORLD, &reqs[i]);
      MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag, MPI_COMM_WORLD,
&reqs[i+4]);
      }
   MPI_Waitall(8, reqs, stats);
   printf("rank= %d coords= %d %d  neighbors(u,d,l,r)= %d %d %d %d
inbuf(u,d,l,r)= %d %d %d %d\n",
rank,coords[0],coords[1],nbrs[UP],nbrs[DOWN],nbrs[LEFT],inbuf[UP],inbuf[DOWN],
inbuf[LEFT],inbuf[RIGHT]);
   }
 else
   printf("Must specify %d tasks. Terminating.\n",SIZE);
 MPI_Finalize(); }
```