Библиотека MPI

Message Passing Interface

История MPI

Стандарт

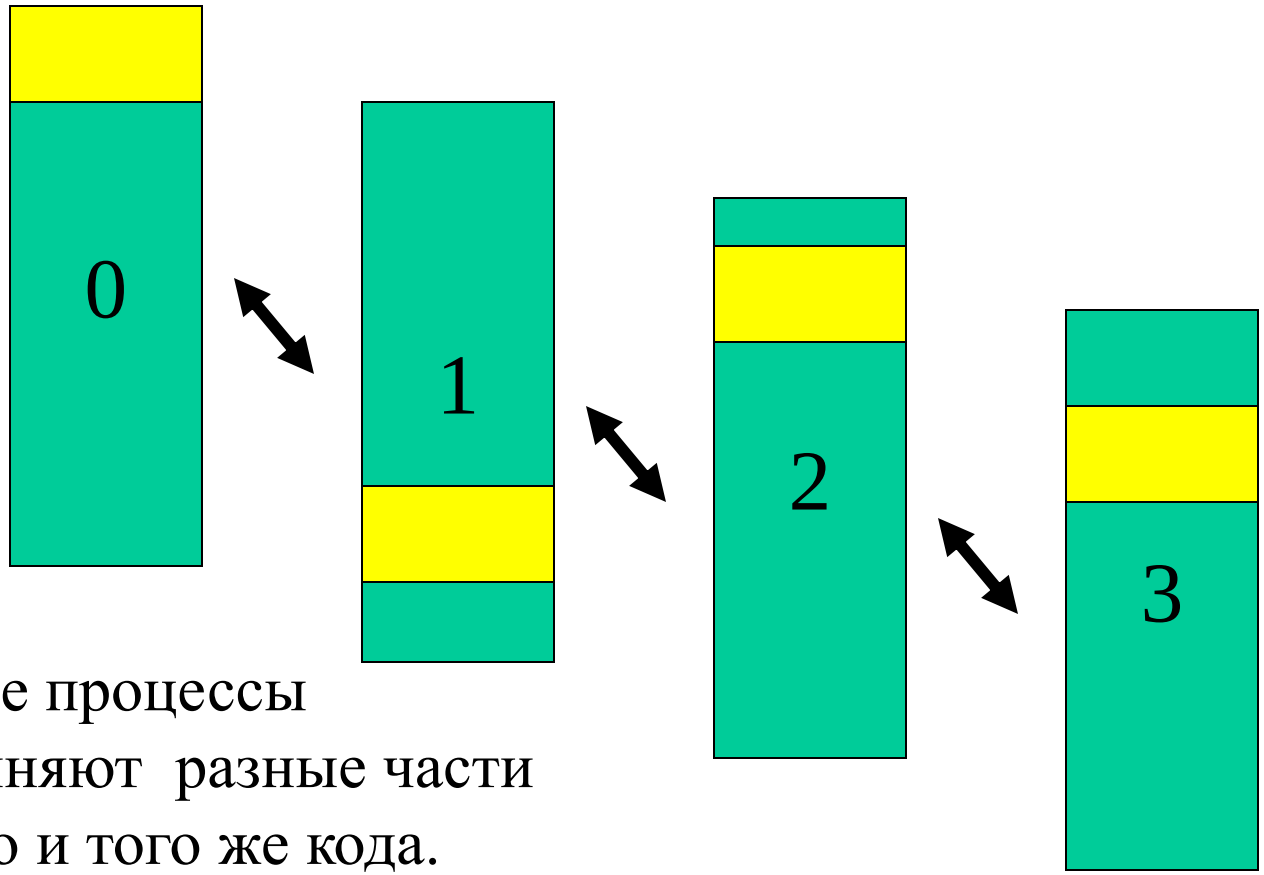
- MPI 1.0 1995 год
- MPI 2.0 1998 год
- MPI 3.0 2012 год.

Определяет API (варианты для Си, C++, Fortran, Java).

«Комплект поставки» MPI

- Библиотека.
- Средства компиляции и запуска приложения.

SPMD-модель.



Разные процессы выполняют разные части одного и того же кода.

Сборка MPI-приложения.

Сборка MPI-приложения осуществляется с помощью специальной утилиты. В случае Си – **mpicc**. Пример:

```
mpicc –o mpihello mpihello.c
```

Запуск MPI-приложения осуществляется с помощью команды **mpirun.**

```
mpirun –np 4 mpihello
```

MPI “Hello, World”

```
#include <stdio.h>
#include <mpi.h>

main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);
  printf("Hello, World!\n");
  MPI_Finalize();
}
```

Функции инициализации и завершения работы.

int MPI_Init(int* argc,char*** argv)

argc – указатель на счетчик аргументов командной строки

argv – указатель на список аргументов

int MPI_Finalize()

Тоже простая MPI-программа

```
#include <mpi.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Функции определения ранга и числа процессов.

int MPI_Comm_size (MPI_Comm comm, int* size)

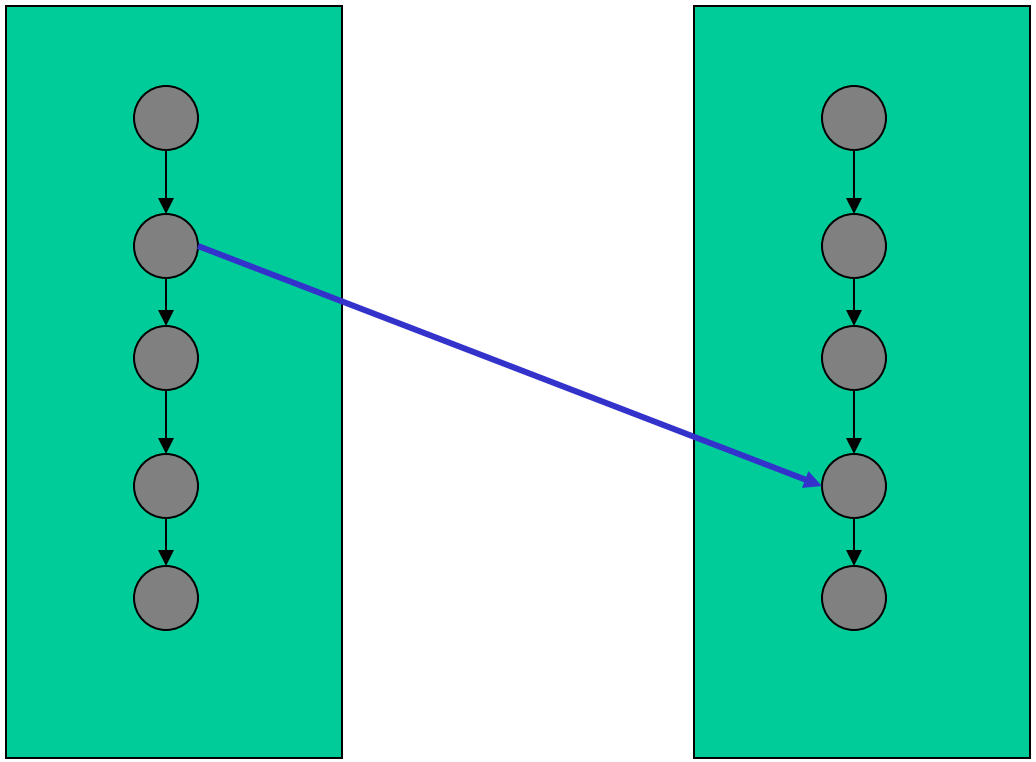
comm - коммуникатор
size – число процессов

int MPI_Comm_rank(MPI_Comm comm, int* rank)

comm – коммуникатор
rank – ранг процесса

Попарные взаимодействия

Назначение попарных взаимодействий



Пример простейшей пересылки

```
#include <stdio.h>
#include <mpi.h>
main(int argc, char* argv[])
{
  int rank;
  MPI_Status st;
  char buf[64];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if(rank == 0) {
    sprintf(buf, "Hello from processor 0");
    MPI_Send(buf, 64, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
  } else {
    MPI_Recv(buf, 64, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &st);
    printf("Process %d received %s \n", rank, buf);
  }
  MPI_Finalize();
}
```

Функции попарных обменов

```
int MPI_Send( buf, count, datatype, dest, tag, comm )

void *buf;                                  /* in */
int  count, dest, tag;                 /* in */
MPI_Datatype datatype;     /* in */
MPI_Comm comm;                /* in */
```

buf - адрес начала буфера посылаемых данных
count - число пересылаемых объектов типа, соответствующего datatype
dest - номер процесса-приемника
tag - уникальный тэг, идентифицирующий сообщение
datatype - MPI-тип принимаемых данных
comm - коммуникатор

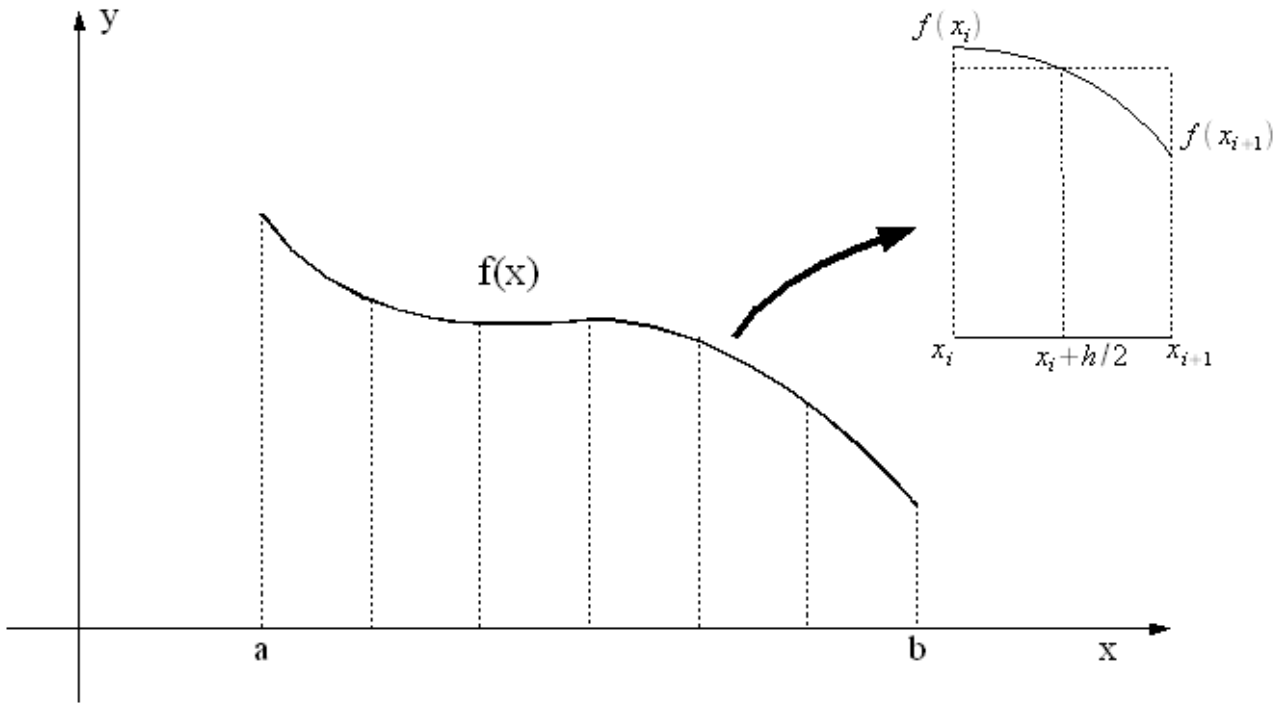
```
int MPI_Recv( buf, count, datatype, source, tag, comm, status )

void *buf;                  /* in */
int count, source, tag;     /* in */
MPI_Datatype datatype;      /* in */
MPI_Comm comm;              /* in */
MPI_Status *status;         /* out */
```

buf - адрес буфера для приема сообщения
count - максимальное число объектов типа datatype, которое может быть записано в буфер
source - номер процесса, от которого ожидается сообщение
tag - уникальный тэг, идентифицирующий сообщение
datatype - MPI-тип принимаемых данных
comm - коммуникатор
status - статус завершения

```
typedef struct
{
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
} MPI_Status;
```

MPI_SOURCE - ранг процесса-передатчика данных
MPI_TAG - тэг сообщения
MPI_ERROR - код ошибки

Численное интегрирование



MPI-программа численного интегрирования

```
#include <mpi.h>
#include <stdio.h>

double f(double x)
{
  return 4./(1 + x * x);
}

main(int argc, char* argv[])
{
  int r;
  int p;
  int i;
  double sum;
  double h;
  MPI_Status st;
  double t;
  int n = 100000000;
  double a = 0.0;
  double b = 1.0;
```

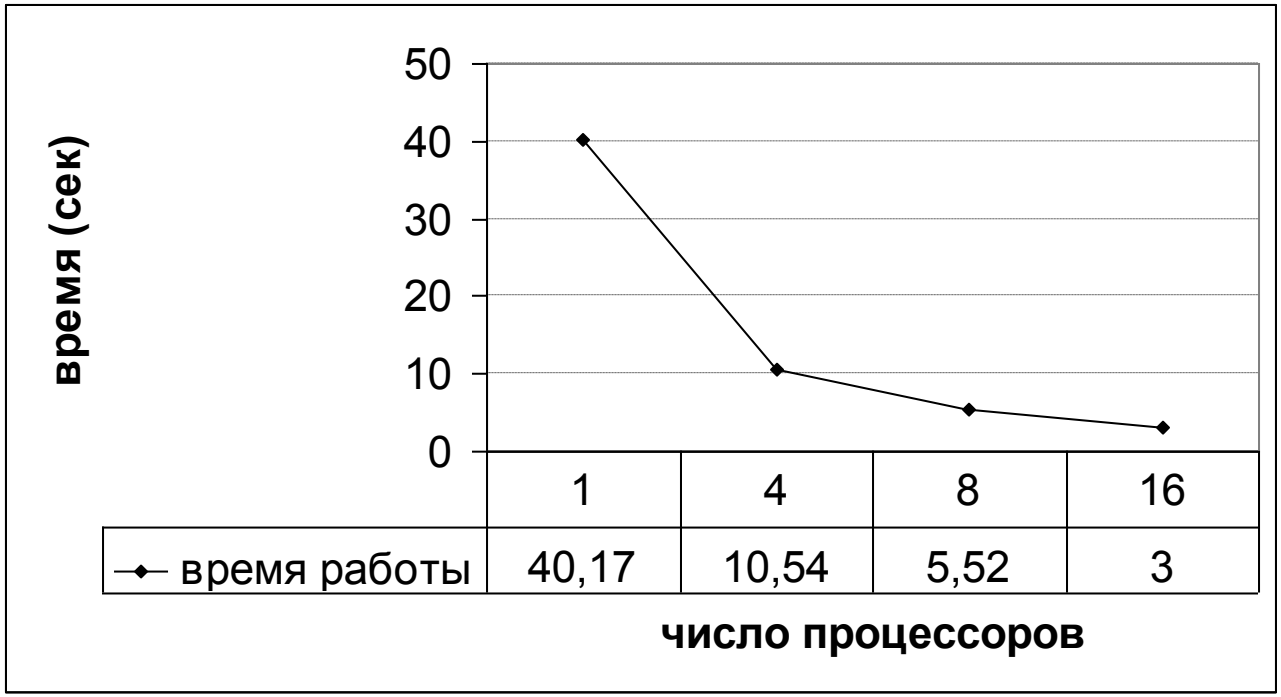
MPI-программа численного интегрирования

```
MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &r);
 MPI_Comm_size(MPI_COMM_WORLD, &p);

 if(r == 0)
   t = MPI_Wtime();


 sum = 0.0;
 h = (b - a) / n;
 for(i = r; i < n; i += p)
   sum += f(a + (i + 0.5) * h);

 sum *= h;
```

MPI-программа численного интегрирования

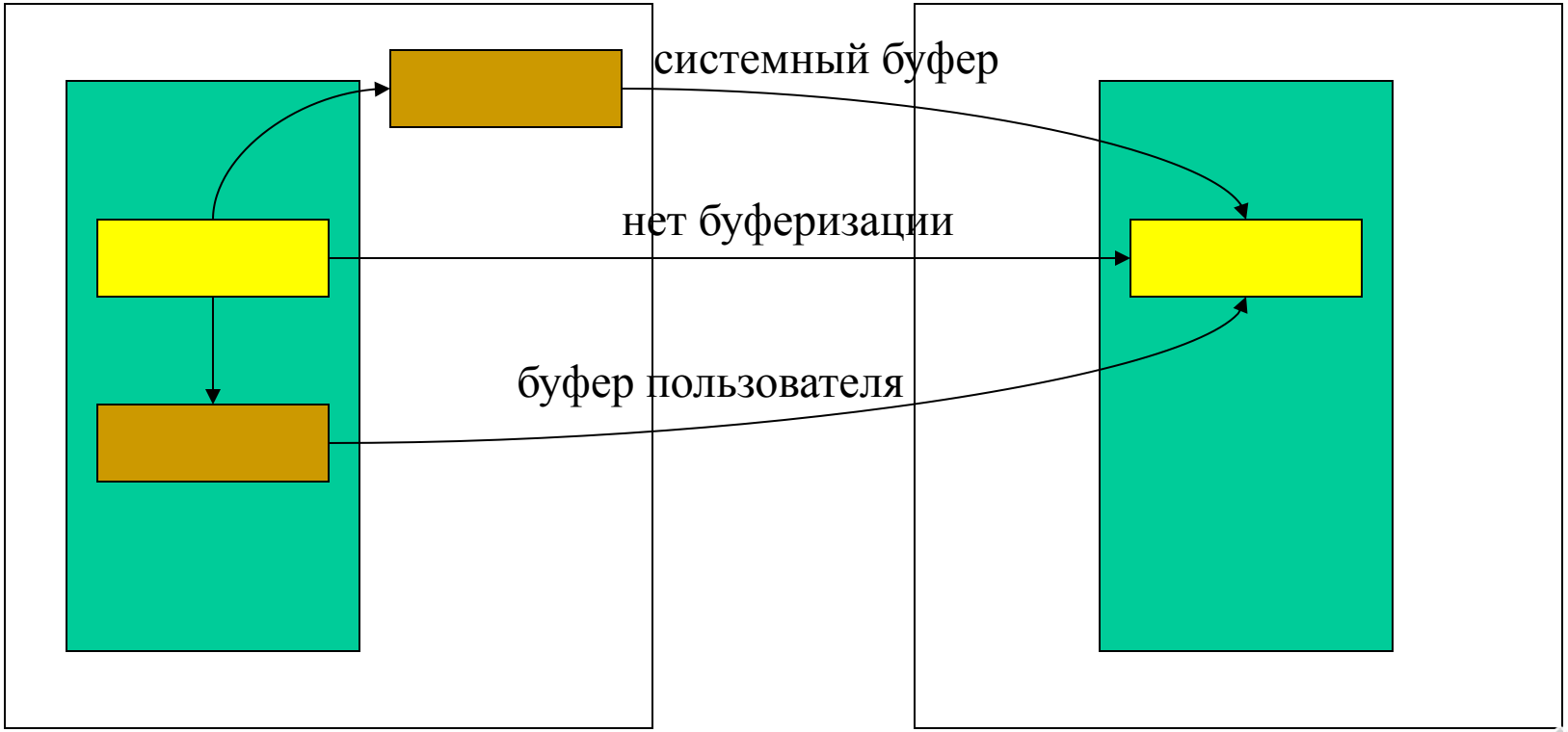
```
  if(r != 0) {
      MPI_Send(&sum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
   } else {
     double s;

     for(i = 1; i < p; i ++) {
       MPI_Recv(&s, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &st);
       sum += s;
     }

     t = MPI_Wtime() - t;
     printf("Integral value = %lf.  Time = %lf sec.\n", sum, t);
   }

  MPI_Finalize();
}
```

Результаты вычислительного эксперимента



Данные получены на MVS-15000BM.

Семантика попарных взаимодействий



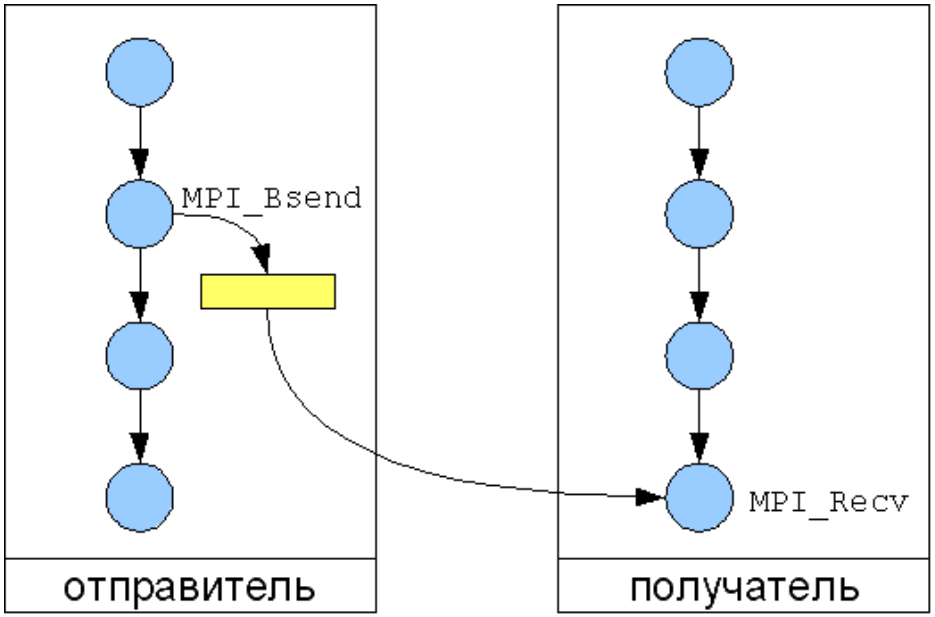
Виды попарных взаимодействий

MPI_Send	**блокирующая пересылка** функция возвращает управление тогда, когда исходный буфер можно освобождать (т.е. данные или скопированы в промежуточный или отправлены)
MPI_Bsend	**буферизованная пересылка** функция возвращает управление тогда, когда данные скопированы в буфер, выделяемый пользователем

MPI_Ssend	**синхронная пересылка** функция возвращает управление тогда, когда процесс-приемник преступил к выполнению соответствующей операции приема
MPI_Rsend	**интерактивная пересылка** поведение функции не определено, если соответствующая операция приема не начала выполнения (для увеличения производительности)

Буферизованная пересылка

- Процесс-**отправитель** выделяет буфер и регистрирует его в системе.
- Функция `MPI_Bsend` помещает данные выделенный буфер.
- Данные посылаются получателю



Буферизованная пересылка

```
int MPI_Bsend(void* buf, int count,
  MPI_Datatype datatype, int dest, int
  tag, MPI_Comm comm)
```

Завершается после копирования данных из буфера `buf` в буфер для отсылаемых сообщений, выделенный программой.

Если места в буфере недостаточно, то возвращается ошибка.

Функции работы с буфером обмена

```
int MPI_Buffer_attach( buffer, size )
void *buffer; /* in */
int  size;    /* in */
```

buffer - адрес начала буфера
size - размер буфера в байтах

```
int MPI_Buffer_detach( bufferptr, size )
void *bufferptr; /* out */
int  *size;      /* out */
```

*bufferptr - адрес высвобожденного буфера
*size - размер высвобожденного пространства

функция **MPI_Buffer_detach** блокирует процесс до тех пор, пока все данные не отправлены из буфера

Вычисление размера буфера

```
int MPI_Pack_size(int incount, MPI_Datatype
datatype, MPI_Comm comm, int *size)
```

Вычисляет размер памяти для хранения одного сообщения.

`MPI_BSEND_OVERHEAD` – дополнительный объем для хранения служебной информации (организация списка сообщений).

Размер буфера для хранения n одинаковых сообщений вычисляется по формуле:

`n x (размер_одного_сообщения + MPI_BSEND_OVERHEAD)`

Порядок организации буферизованных пересылок

- Вычислить необходимый объем буфера (MPI_Pack_size).
- Выделить память под буфер (malloc).
- Зарегистрировать буфер в системе (MPI_Buffer_attach).
- Выполнить пересылки.
- Отменить регистрацию буфера (MPI_Buffer_dettach)
- Освободить память, выделенную под буфер (free).

Особенности работы с буфером

•Буфер всегда один.

• Для изменения размера буфера сначала следует отменить регистрацию, затем увеличить размер буфера и снова его зарегистрировать.

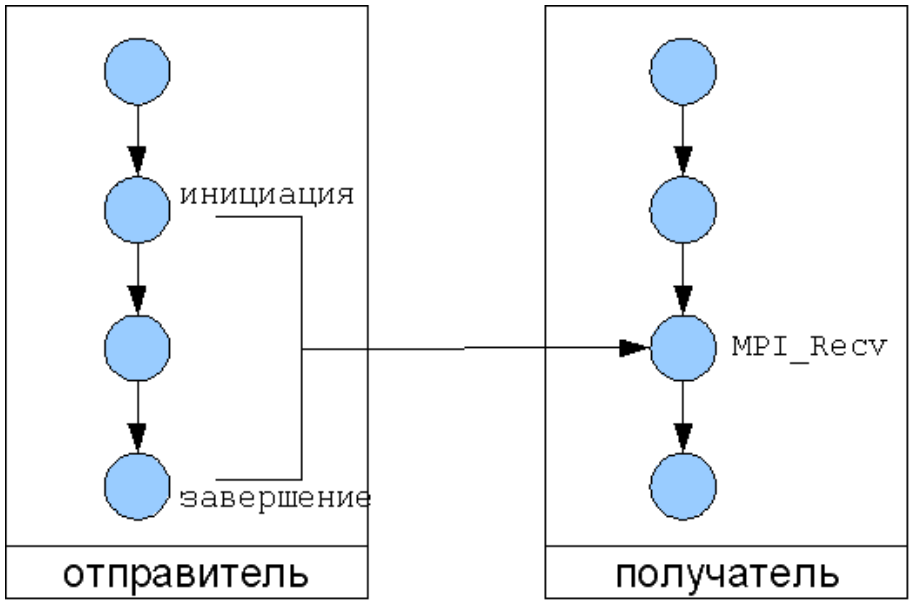
•Освобождать буфер следует только после того, как отменена регистрация.

Пример буферизованной пересылки

```
MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD,&msize)
blen = M * (msize + MPI_BSEND_OVERHEAD);
buf = malloc(blen);
MPI_Buffer_attach(buf, blen);
for(i = 0; i < M; i ++) {
  n = i;
  MPI_Bsend(&n, 1, MPI_INT, 1, i, MPI_COMM_WORLD);
}
MPI_Buffer_detach(&abuf, &ablen);
free(abuf);
```

Неблокирующие пересылки

- Предназначены для перекрытия обменов и вычислений.
- Операция расщепляется на две: инициация и завершение.



Неблокирующая пересылка

Инициация:

```
int MPI_Isend( buf, count, datatype, dest, tag, comm, request)

MPI_Request      *request; /* out */
```

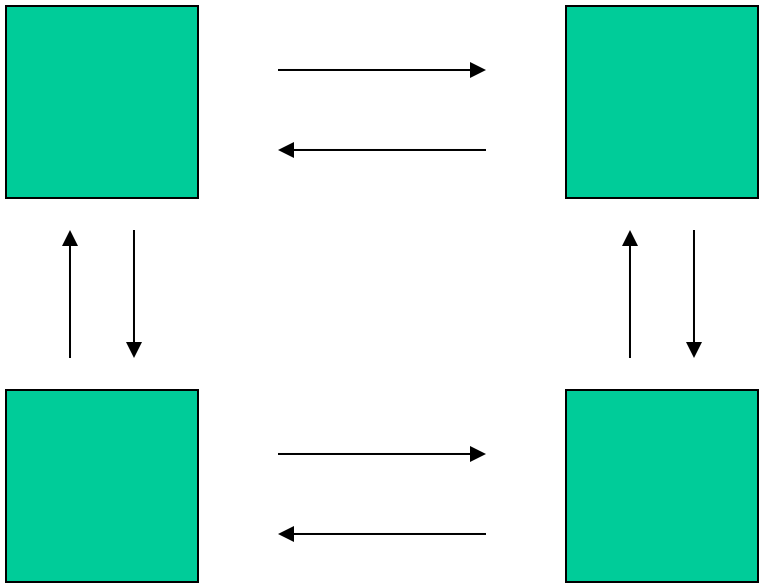
MPI_Ibsend(…), MPI_Issend(…), MPI_Irsend(…)

```
int MPI_Irecv( buf, count, datatype, source, tag, comm, request )

MPI_Request      *request; /* out */
```

Завершение:

```
int MPI_Wait (MPI_Request * request, MPI_Status * status)

int MPI_Test(MPI_Request *request, int *flag,
      MPI_Status *status)

int MPI_Waitall(int count, MPI_Request array_of_requests[],
      MPI_Status array_of_statuses[] )

int MPI_Waitany(int count, MPI_Request array_of_requests[],
      int* index, MPI_Status *status )
```

Пример: кольцевой сдвиг данных



```
#include <mpi.h>
#include <stdio.h>


void main (int argc, char* argv[])
{
  int numtasks, rank, next, prev, buf[2], tag1 = 1, tag2 = 2;
  MPI_Request reqs[4];
  MPI_Status stats[4];

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

  prev = (rank == 0) ? (numtasks - 1) : (rank - 1);
  next = (rank == (numtasks - 1)) ? 0 : (rank + 1);
```

```
  MPI_Irecv (&buf[0], 1, MPI_INT, prev, tag1,
       MPI_COMM_WORLD, &reqs[0]);
  MPI_Irecv (&buf[1], 1, MPI_INT, next, tag2,
       MPI_COMM_WORLD, &reqs[1]);

  MPI_Isend (&rank, 1, MPI_INT, prev, tag2,
       MPI_COMM_WORLD, &reqs[2]);
  MPI_Isend (&rank, 1, MPI_INT, next, tag1,
       MPI_COMM_WORLD, &reqs[3]);

  MPI_Waitall (4, reqs, stats);

  printf("rank: %d, buf[0]: %d, buf[1]: %d\n",
       rank, buf[0], buf[1]);


  MPI_Finalize ();
}
```

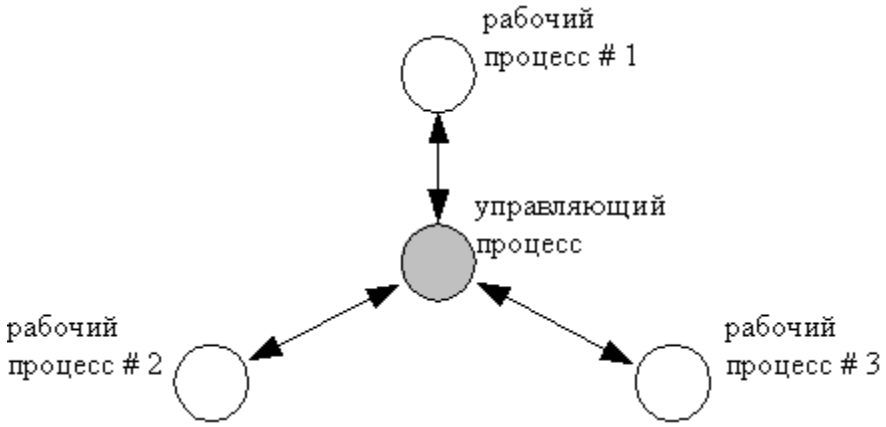
Прием по шаблону

В качестве параметров **source** и **tag** в функции **MPI_Recv** могут быть использованы константы

MPI_ANY_SOURCE и **MPI_ANY_TAG**

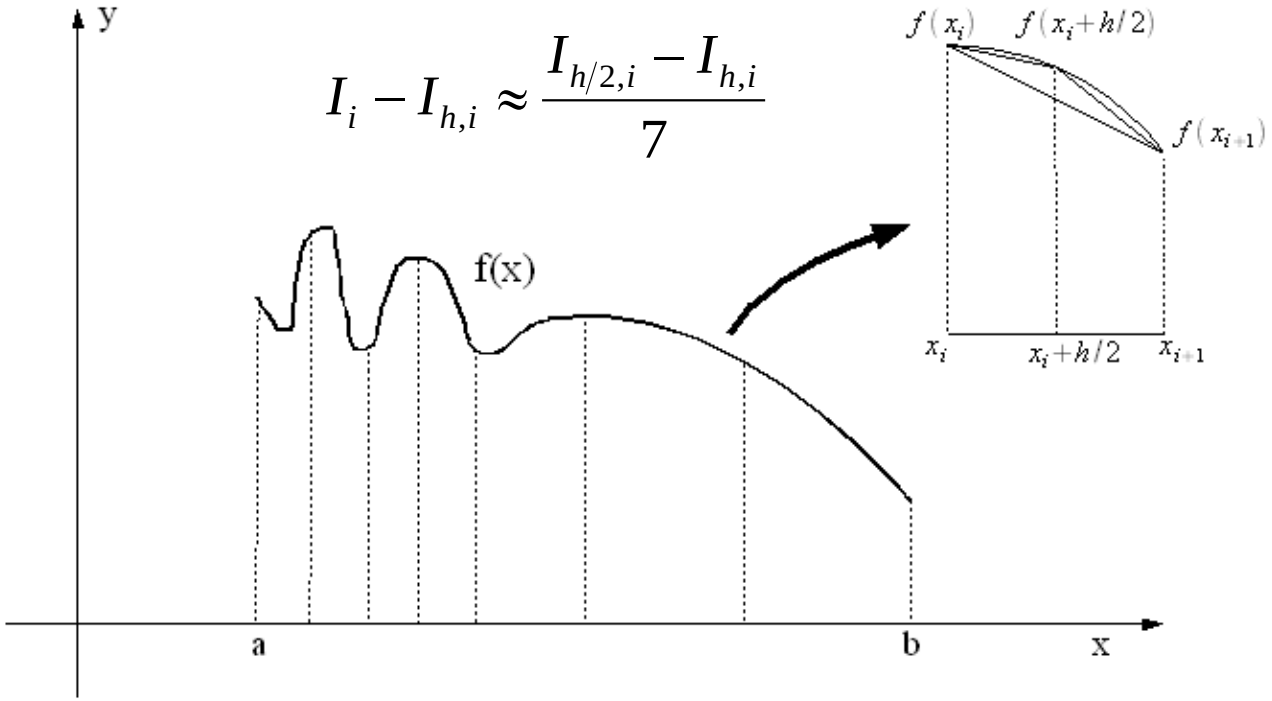
соответственно. Допускается прием от процесса с произвольным номером и/или сообщения с произвольным тэгом.

Стратегия управляющий-рабочие



Адаптивная квадратура

$$I_i - I_{h,i} \approx \frac{I_{h/2,i} - I_{h,i}}{7}$$



Частота разбиения выбирается в соответствии с плавностью изменения функции.

ПРИМЕР РЕАЛИЗАЦИИ

```
#include <math.h>
#include <mpi.h>
#define MYABS(A) (((A) < 0) ? (-(A)) : (A))

double f(double x)
{
  return sin(1. / x);
}
```

```
int adint(double (*f) (double), double left, double right, double eps, double *nint)
{
  double mid;
  double h;
  double Iold;
  double Ileft;
  double Iright;

  h = 0.5 * (right - left);
  mid = 0.5 * (right + left);

  Iold =  h * (f(left) + f(right));
  Ileft = 0.5 * h * (f(left) + f(mid));
  Iright = 0.5 * h * (f(mid) + f(right));

  *nint = Ileft + Iright;

  if(MYABS(Iold - *nint) < eps)
    return 1;
  else
    return 0;
}
```

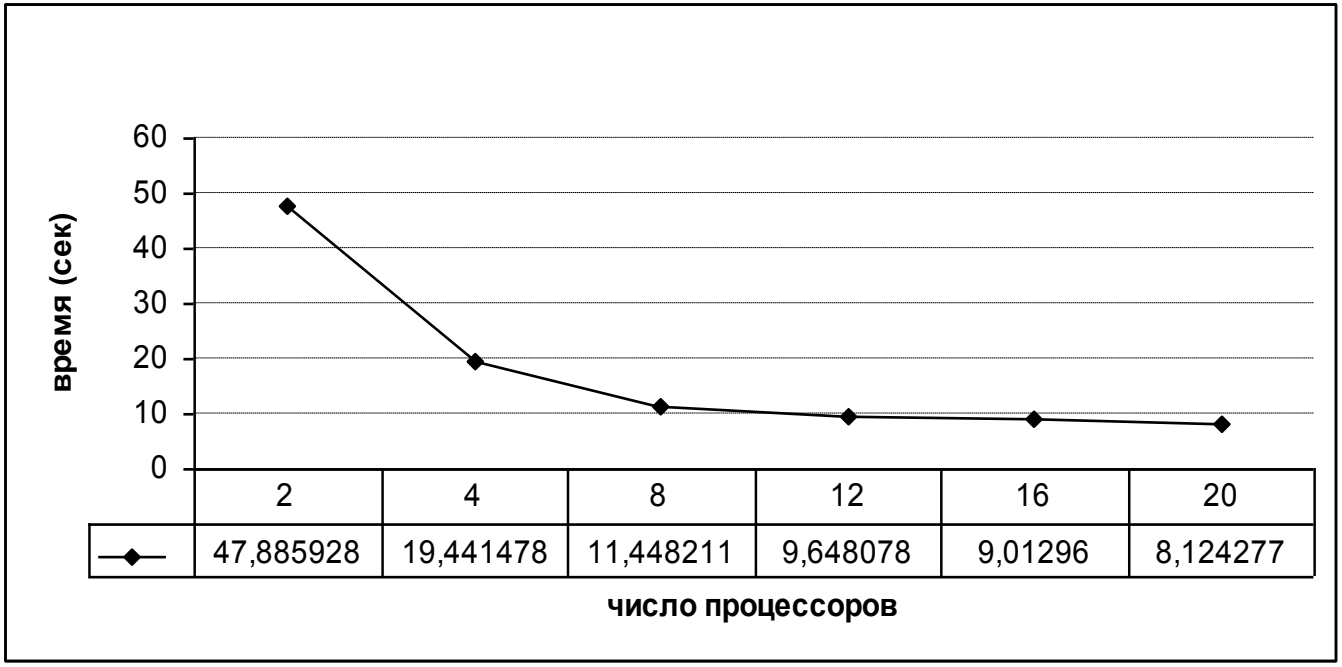
```
double recadint(double (*f)(double), double left, double right, double eps)
{
    double I;
     if(adint(f, left, right, eps, &I)) {
            return I;
     } else {
            double Ileft;
            double Iright;
            double mid;

            mid = 0.5 * (right + left);
            Ileft = recadint(f, left, mid, 0.5 * eps);
            Iright = recadint(f, mid, right, 0.5 * eps);
            return Ileft + Iright;
     }
}
```

```
main(int argc, char* argv[])
{
  int r, p, n = 100000, s = 0;
  double a = 0.0001, b = 1.0, eps = 0.0001, I = 0., h, t;

  MPI_Status st;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &r);
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  h = (b - a) / n;
  n --;
  t = MPI_Wtime();
```

```
if(r == 0) {
  while(s != (p-1)) {
    double Islave;
    MPI_Recv(&Islave, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
                    MPI_COMM_WORLD, &st);
    I += Islave;
    MPI_Send(&n, 1, MPI_INT, st.MPI_SOURCE, 0, MPI_COMM_WORLD);
    if(n >= 0)
      n --;
    else
     s += 1;
  }
} else {
  int m;
  while(1) {
    MPI_Send(&I, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    MPI_Recv(&m, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &st);
    if(m >= 0)
            I = recadint(f, a + h * m, a + h * (m + 1), eps * h / (b - a));
    else
            break;
  }
 }
```

```
 if(r == 0) {
   t = MPI_Wtime() - t;
   printf("Integral value: %lf, time = %lf\n", I, t);
  }
  MPI_Finalize();
}
```

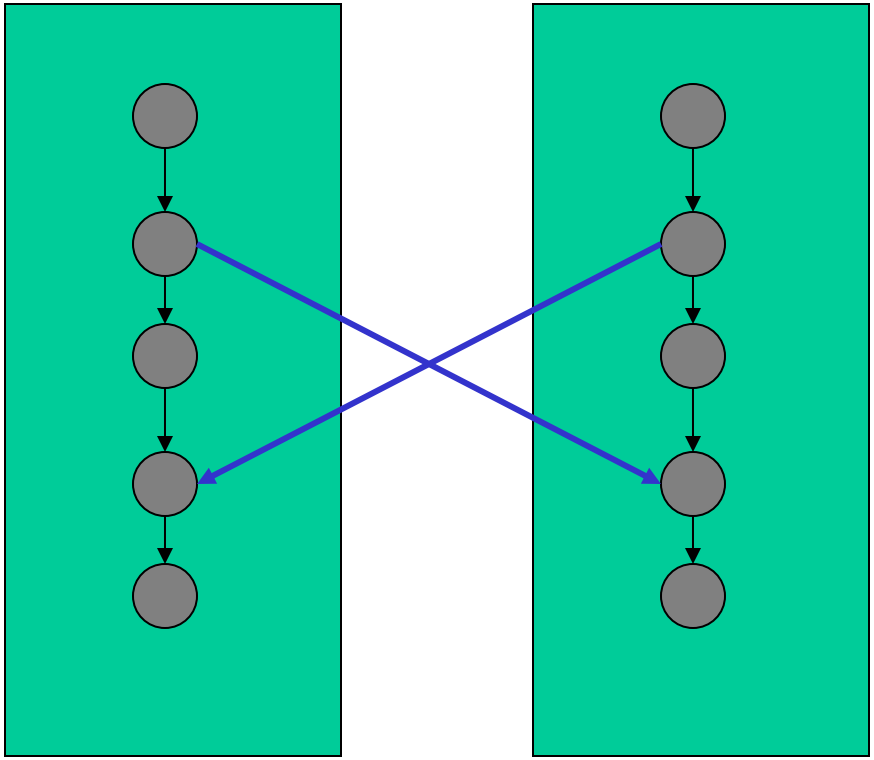
Результаты экспериментов



данные получены на MVS 15000 BM

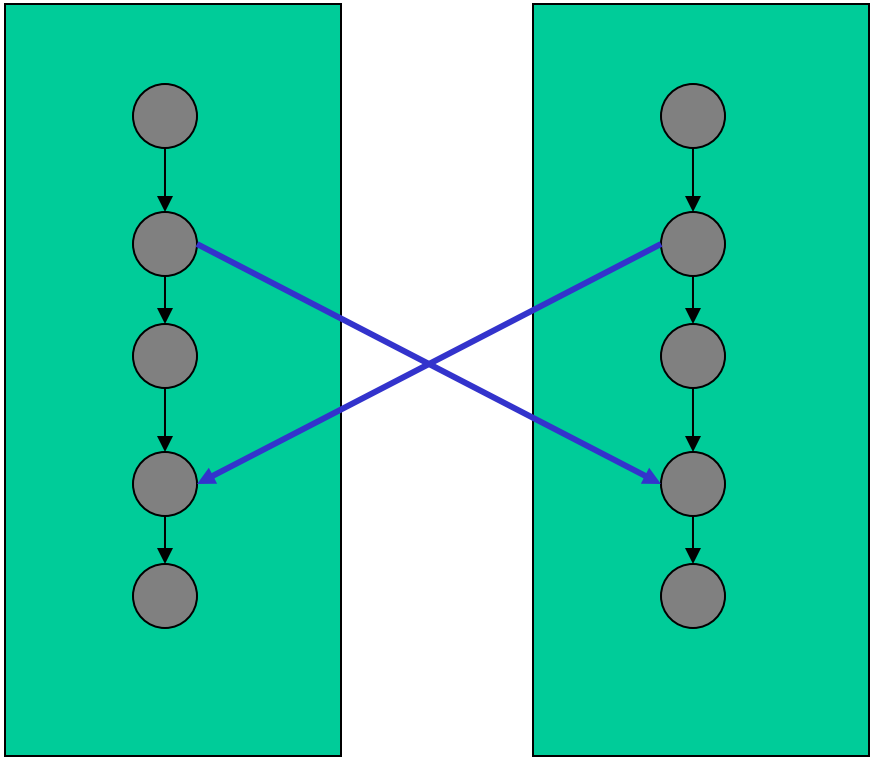
Deadlock

```
if(rank == 0) {
        MPI_Ssend(… 1 …)
        MPI_Recv(…1…)
} else {
        MPI_Ssend(… 0 …)
        MPI_Recv(…0…)
}
```

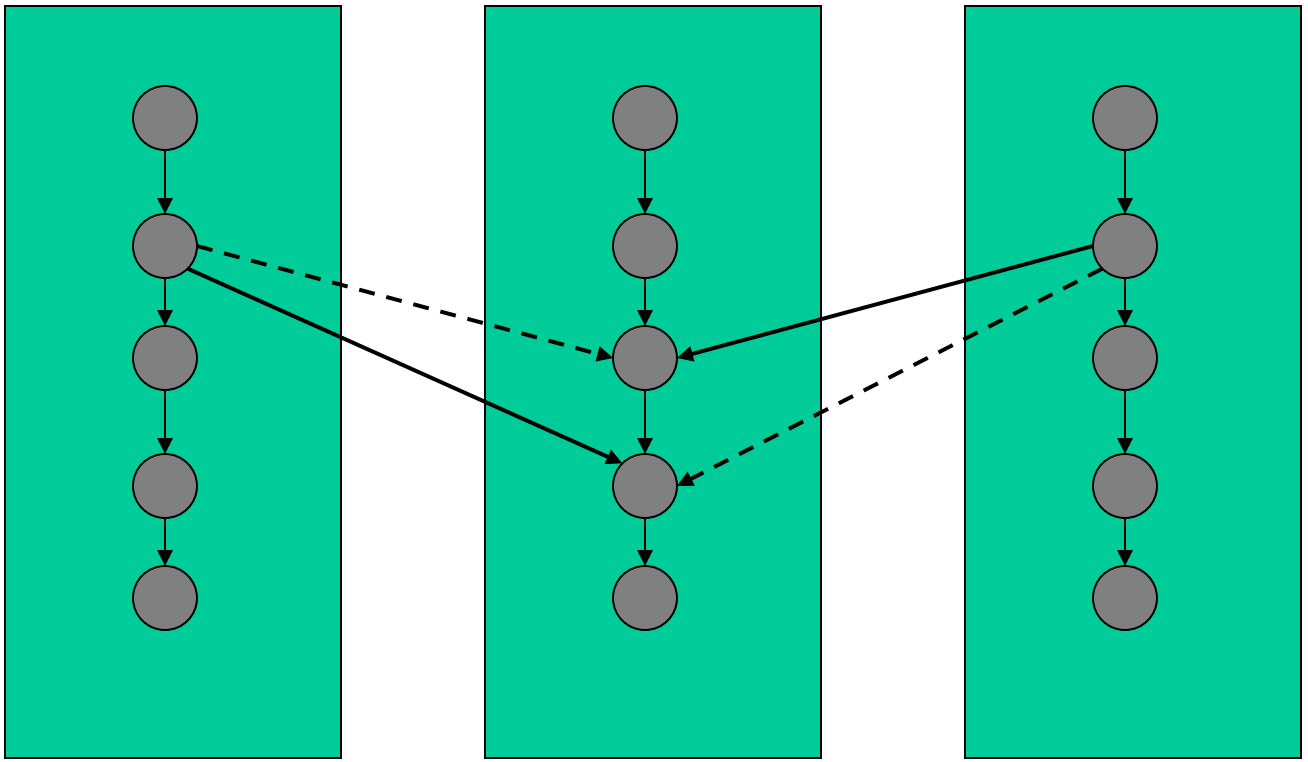


«Недетерминированный» deadlock

```
if(rank == 0) {
        MPI_Send(… 1 …)
        MPI_Recv(…1…)
} else {
        MPI_Send(… 0 …)
        MPI_Recv(…0…)
}
```



Недетерминизм за счет разницы в относительных скоростях процессов (race condition)



Коллективные взаимодействия процессов

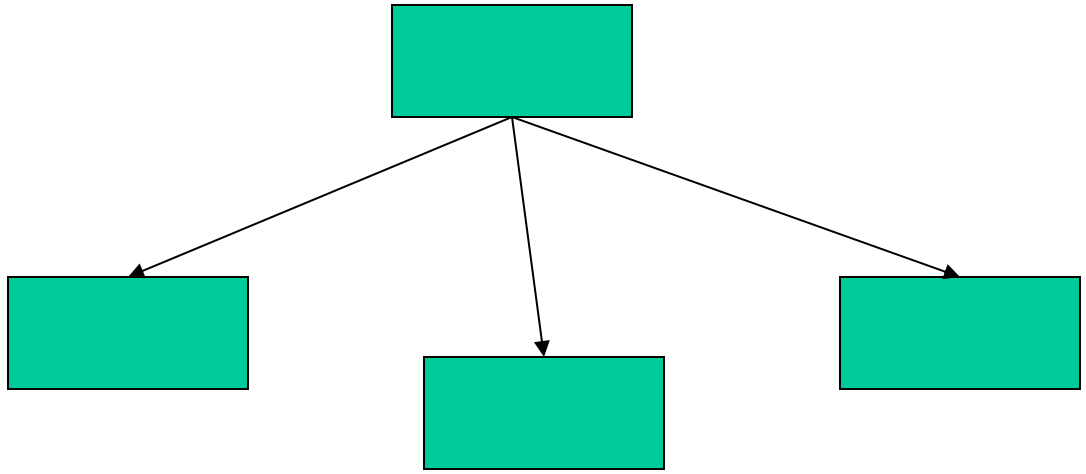
Коллективные взаимодействия процессов

MPI предоставляет ряд функций для коллективного взаимодействия процессов.

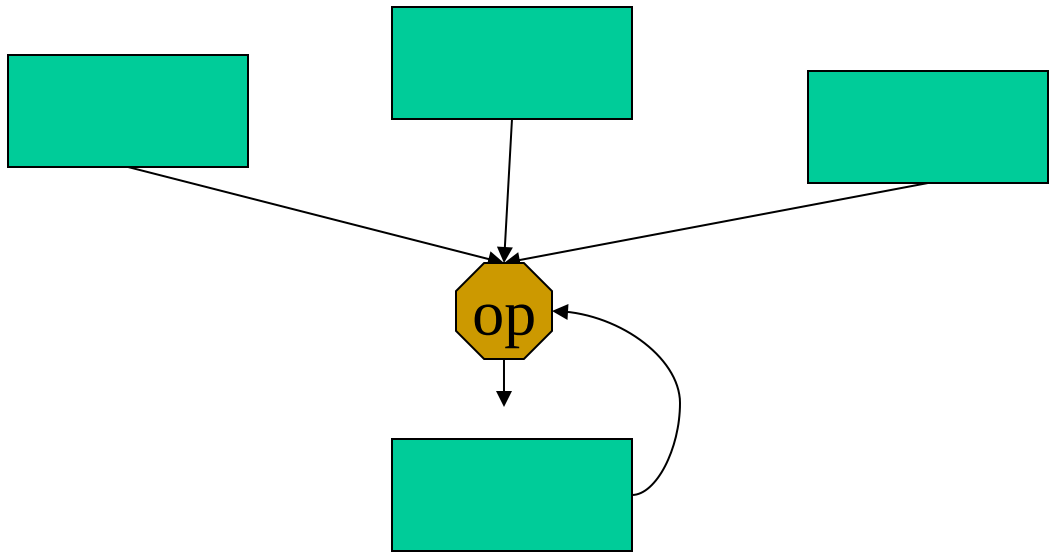
Эти функции называют коллективными, поскольку они должны вызываться на всех процессах, принадлежащих некоторому коммуникатору.

```
int MPI_Bcast ( buffer, count, datatype, root, comm
)
```

void* buffer - начальный адрес буфера для передачи собщений
int count - число передаваемых элементов данных
MPI_Datatype datatype - тип передаваемых данных
int root - ранг процесса, посылающего данные
MPI_Comm comm - коммуникатор



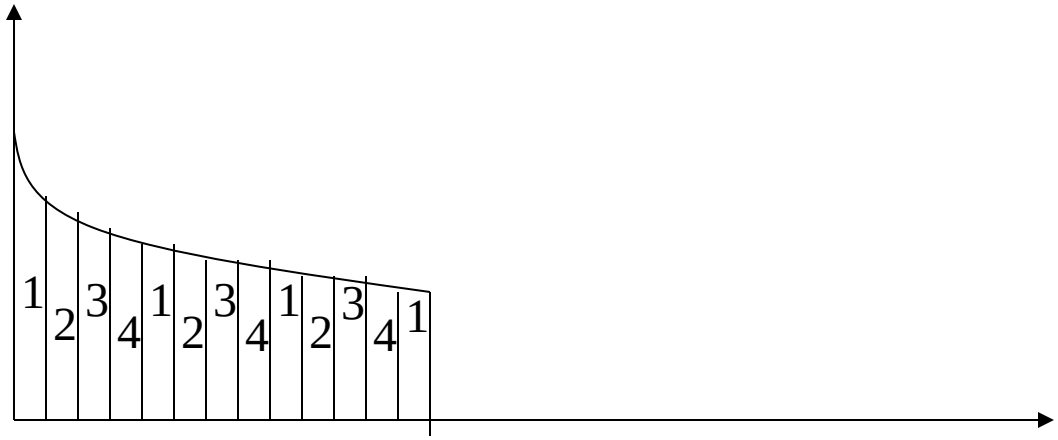
```
int MPI_Reduce ( sendbuf, recvbuf, count,
      datatype, op, root, comm )

void  *sendbuf;     буфер операндов
void *recvbuf;      буфер приема
int  count;         число данных
MPI_Datatype datatype; тип данных
MPI_Op op;          операция
int root;           ранг процесса, содержащего результат
MPI_Comm comm;      коммуникатор
```



MPI_MAX	максимум
MPI_MIN	минимум
MPI_SUM	сумма
MPI_PROD	произведение
MPI_LAND	логическое "и"
MPI_BAND	побитовое "и"
MPI_LOR	логическое "или"
MPI_BOR	побитовое "или"
MPI_LXOR	логическое исключающее "или"
MPI_BXOR	побитовое исключающее "или"

Вычисление числа Пи

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

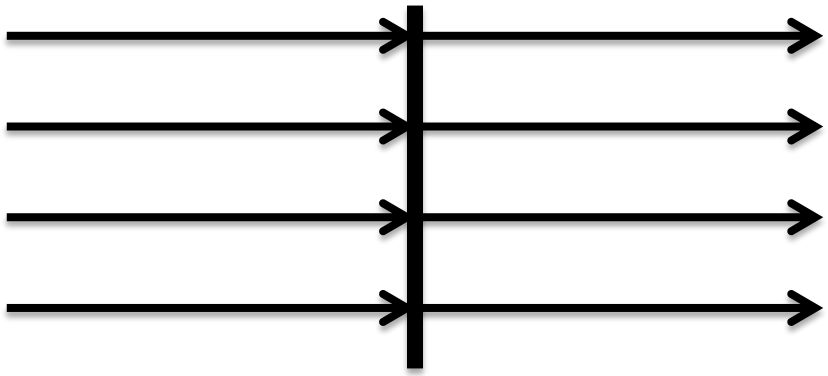


Вычисление числа Pi

```
#include "mpi.h"
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
  int n, myid, numprocs, i;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

```
while (1)
{
  if (myid == 0) {
      printf("Enter the number of intervals: (0 quits) ");
      scanf("%d",&n);
  }
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if (n == 0) break;

  h   = 1.0 / (double) n;
  sum = 0.0;
  for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
  }
  mypi = h * sum;
```

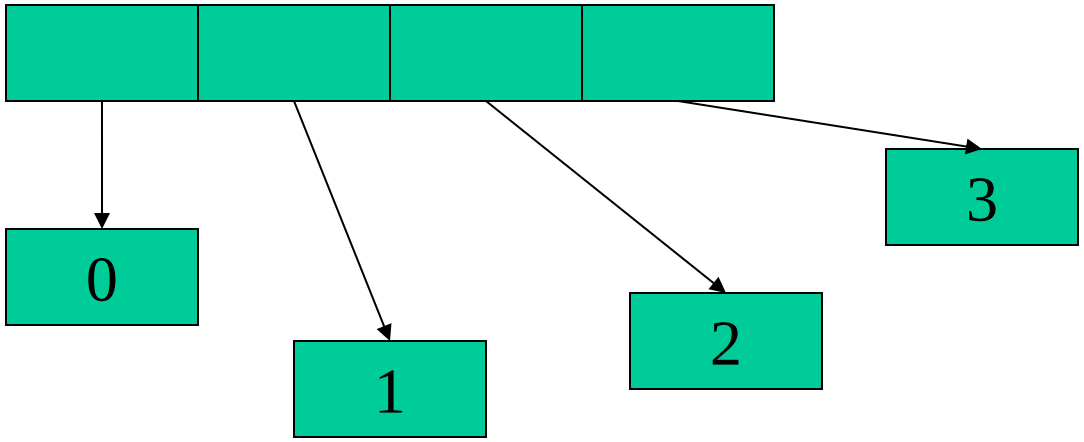
```
 MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);

    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",
               pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
}
```



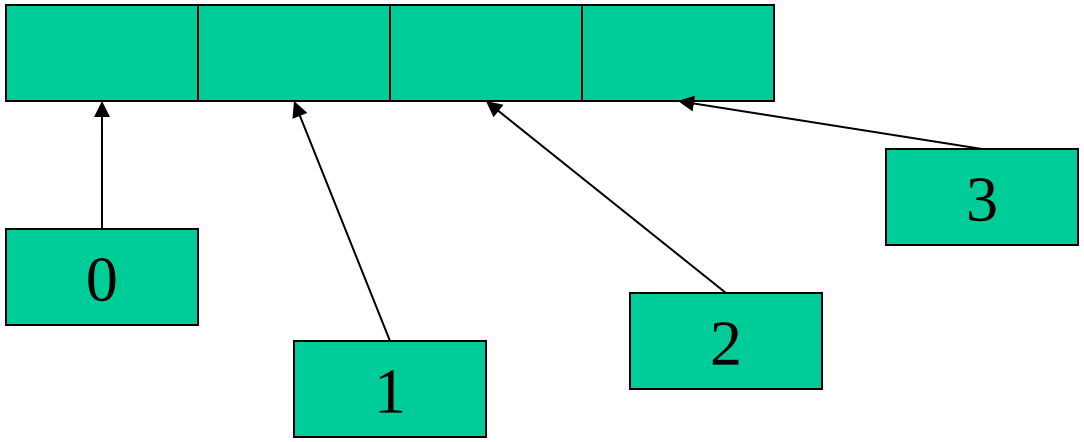
Функция синхронизации процессов:

```
int MPI_Barrier ( comm ) ;

MPI_Comm comm;
```

```
int MPI_Scatter ( sendbuf, sendcnt, sendtype,
recvbuf, recvcnt, recvtype, root, comm )

void *sendbuf;
int  sendcnt;
MPI_Datatype  sendtype;
void *recvbuf;
int recvcnt;
MPI_Datatype recvtype;
int root;
MPI_Comm comm;
```

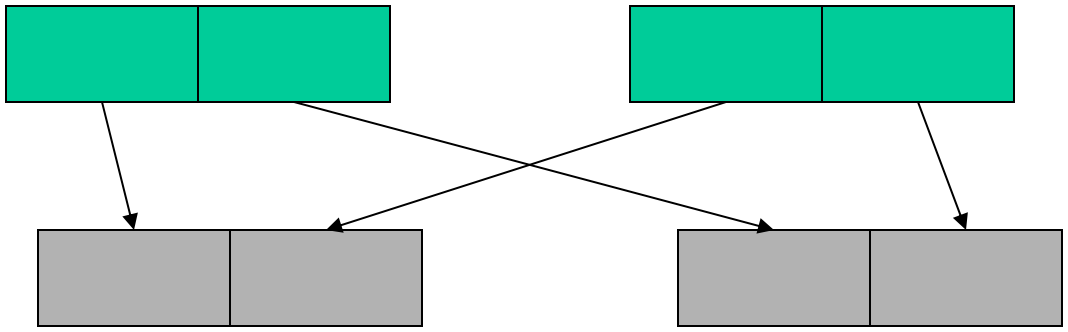


```
int MPI_Gather ( sendbuf, sendcnt, sendtype, recvbuf,
    recvcount, recvtype, root, comm )

    void               *sendbuf;
    int                 sendcnt;
    MPI_Datatype        sendtype;
    void               *recvbuf;
    int                 recvcount;
    MPI_Datatype        recvtype;
    int                 root;
    MPI_Comm            comm;
```



```
int MPI_Allreduce ( sendbuf, recvbuf, count, datatype, op,
comm )

void *sendbuf;
void *recvbuf;
int count;
MPI_Datatype datatype;
MPI_Op op;
MPI_Comm comm;


int MPI_Allgather ( sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, comm )

void *sendbuf;
int  sendcount;
MPI_Datatype sendtype;
void *recvbuf;
int  recvcount;
MPI_Datatype recvtype;
MPI_Comm comm;
```

```
int MPI_Alltoall( sendbuf, sendcount,
sendtype, recvbuf, recvcnt, recvtype, comm )

void *sendbuf;
int  sendcount;
MPI_Datatype sendtype;
void *recvbuf;
int recvcnt;
MPI_Datatype recvtype;
MPI_Comm comm;
```



Метод Якоби решения линейных систем

$$Ax = f$$

, -

$$A = (a_{ij})$$

$$x = (x_1, \ldots, x_m)^T \qquad f = (f_1, \ldots, f_m)^T$$

$$Ax = f$$

$$x_i = -\sum_{j=1}^{i-1} \frac{a_{ij}}{a_{ii}} x_j - \sum_{j=i+1}^{m} \frac{a_{ij}}{a_{ii}} x_j + \frac{f_i}{a_{ii}}, \quad i = 1,2,\ldots,m.$$

$$x_i^{n+1} = -\sum_{j=1}^{i-1} \frac{a_{ij}}{a_{ii}} x_j^n - \sum_{j=i+1}^{m} \frac{a_{ij}}{a_{ii}} x_j^n + \frac{f_i}{a_{ii}}, \quad i = 1,2,\ldots,m.$$

Условие остановки: $\left\| x^{n+1} - x^n \right\| = \sqrt{\sum_{i=1}^{n} (x_i^{n+1} - x_i^n)^2} \le \varepsilon$

Матричная форма записи

$$x_i^{n+1} = -\sum_{j=1}^{i-1} \frac{a_{ij}}{a_{ii}} x_j^n - \sum_{j=i+1}^{m} \frac{a_{ij}}{a_{ii}} x_j^n + \frac{f_i}{a_{ii}}, \quad i = 1,2,\ldots,m.$$

$$x^{n+1} = -Bx^n + D^{-1} f = -Bx^n + g$$

$$D = \begin{pmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{pmatrix}$$

$$B = \begin{pmatrix} 0 & \frac{a_{12}}{a_{11}} & \frac{a_{13}}{a_{11}} \\ \frac{a_{21}}{a_{22}} & 0 & \frac{a_{23}}{a_{22}} \\ \frac{a_{31}}{a_{33}} & \frac{a_{32}}{a_{33}} & 0 \end{pmatrix}$$

Условие сходимости

Диагональное преобладание – достаточное условие сходимости.

$$a_{ii} > \sum_{j \neq i} \left| a_{ij} \right|$$

Пример:

$$a_{ii} = 2m, a_{ij} = 1,\ f_i = \frac{m(m+1)}{2} + i(2m-1)$$

$$\textit{решение}: \quad x_i = i, i = 1, \ldots, m$$

Последовательный алгоритм

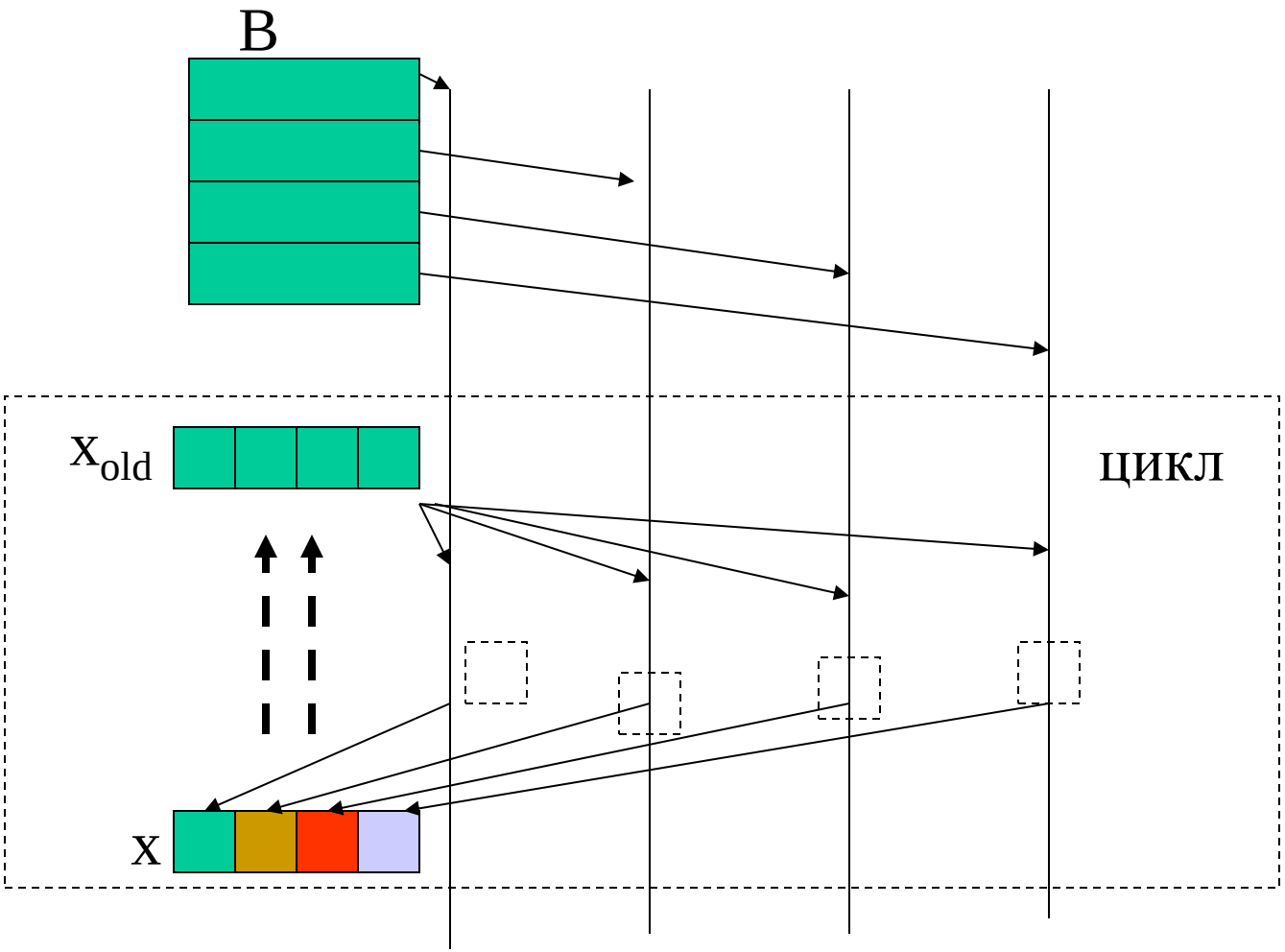
```
#define MAXITERS 1000
void init(int m, double* B, double* g, double* x)
{
  int i, j;
  for(i = 0; i < m; i ++) {
      g[i] = ((double)m+1)/4. + (1.-1./(double)(2 * m))*(i + 1);
      x[i] = 1.;
      B[i * m + i] = 0.0;
  }
  for(i = 0; i < m; i ++) {
    for(j = 0; j < m; j ++)
      B[i * m + j] = 1./(double)(2 * m);
  }
}
```

```
double evalDiff(double* u, double* v, int m)
{
  int i;
  double a = 0.0;
  for(i = 0; i < m; i ++) {
    double b;
    b = v[i] - u[i];
    a += b * b;
  }
  return sqrt(a);
}
```

```
main(int argc, char* argv[])
{
  int m, I = 0;
  double *B, *g, *x, *xold, eps, diff, t;
  m = atoi(argv[1]);
  eps = atof(argv[2]);
  B = (double*)malloc(m * m * sizeof(double));
  g = (double*)malloc(m * sizeof(double));
  xold = (double*)malloc(m * sizeof(double));
  x = (double*)malloc(m * sizeof(double));
  init(m, B, g, xold);
  t = time(NULL);
```

```
do {
    int i;
    for(i = 0; i < m; i ++) {
      int j;
      double a = 0.;
      double* row = B + i * m;
      for(j = 0; j < m; j ++) {
          a += row[j] * xold[j];
      }
      x[i] = -a + g[i];
    }
    diff = evalDiff(xold, x, m);
    I ++;
    printf("diff = %lf, eps = %lf\n", diff, eps);
    memcpy(xold, x, m * sizeof(double));
  } while ((diff >= eps) && (I <= MAXITERS));
  t = time(NULL) - t;
  printf("%d iterations consumed %lf seconds\n", I, t);
}
```

Параллельный алгоритм



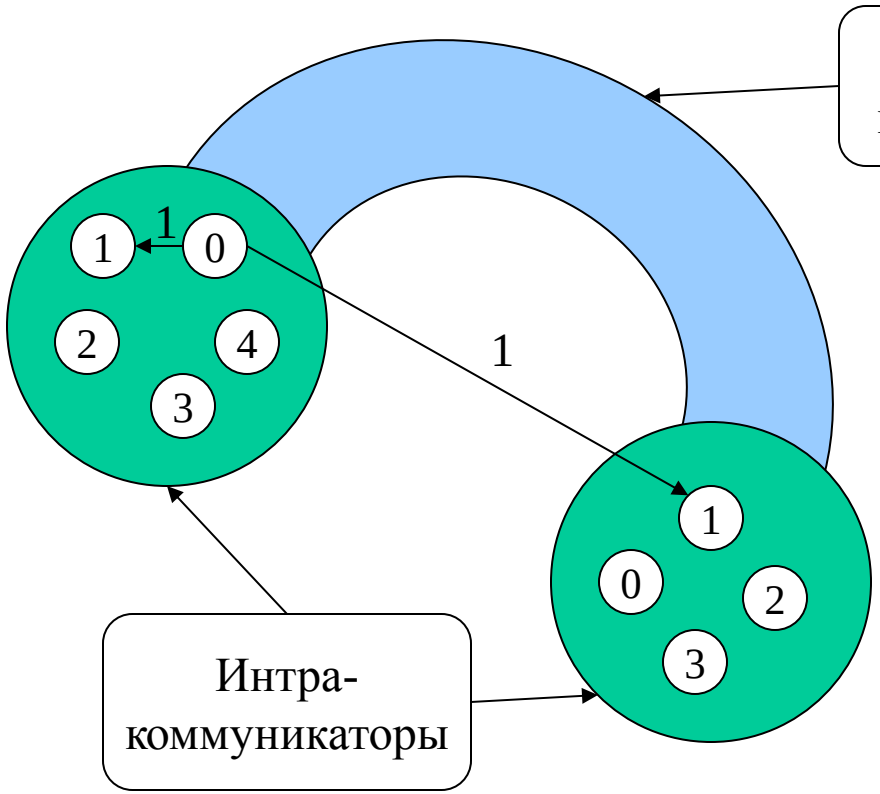
```
main(int argc, char* argv[])
{
  int m, np, rk, chunk, i, I = 0;
  double *B, *Bloc, *g, *x, *xloc, *xold, t, diff, eps;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Comm_rank(MPI_COMM_WORLD, &rk);
  if(rk == 0) {
    m = atoi(argv[1]);
    eps = atof(argv[2]);
    chunk = m / np;
    B = (double*)malloc(m * m * sizeof(double));
  }
  MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&chunk, 1, MPI_INT, 0, MPI_COMM_WORLD);
  g = (double*)malloc(m * sizeof(double));
  x = (double*)malloc(m * sizeof(double));
  xold = (double*)malloc(m * sizeof(double));
  Bloc = (double*)malloc(chunk * m * sizeof(double));
  xloc = (double*)malloc(chunk * sizeof(double));
  if(rk == 0){
    init(m, B, g, xold);
    t = MPI_Wtime();
  }
```

```
MPI_Scatter(B, m * chunk, MPI_DOUBLE,
          Bloc, m * chunk, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(g, m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
do {
  MPI_Bcast(xold, m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  for(i = 0; i < chunk; i ++) {
    int j;
    double b = 0.;
    double* row = Bloc + i * m;
    for(j = 0; j < m; j ++) {
       b += row[j] * xold[j];
    }
    xloc[i] = -b + g[rk * chunk + i];
  }
  MPI_Gather(xloc, chunk, MPI_DOUBLE,
           x, chunk, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  if(rk == 0) {
    diff = evalDiff(xold, x, m);
    printf("diff = %lf, eps = %lf\n", diff, eps);
    memcpy(xold, x, m * sizeof(double));
  }
  MPI_Bcast(&diff, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  I ++;
} while((diff >= eps) && (I <= MAXITERS));
```

```
  if(rk == 0) {
    t = MPI_Wtime() - t;
    printf("%d iterations consumed %lf sec\n", I, t);
  }
  MPI_Finalize();
}
```

Группы и коммуникаторы

Интер- и интра-коммуникаторы



Интра-коммуникаторы объединяют процессы из одной группы.

Интер-коммуникатор позволяет передавать данные между процессами из разных **интра**-коммуникаторов.

Интер-коммуникаторы не могут использоваться в коллективных взаимодействиях.

Назначение коммуникаторов

- Поддержка параллельных библиотек.
- Поддержка коллективных операций на части вычислительного пространства.
- Повышение уровня абстракции параллельных приложений.

```
#include "mpi.h"
#include <math.h>
double g(double x)
{
  return 4.0 / (1.0 + x * x);
}
void quad(int n, double (*f) (double), MPI_Comm comm, double* result)
{
  int myid, numprocs, i;
  double h, sum, x, mypi;

  MPI_Comm_rank(comm,&myid);
  MPI_Comm_size(comm, &numprocs);
  MPI_Bcast(&n, 1, MPI_INT, 0, comm);
  h   = 1.0 / (double) n;
  sum = 0.0;
  for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += f(x);
  }
  mypi = h * sum;
  MPI_Reduce(&mypi, result, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
}
```

```
int main(int argc, char *argv[])
{
  int n, myid;
  double PI25DT = 3.14159265358979323846264 3;
  double pi;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  while (1)
  {
     if (myid == 0) {
       printf("Enter the number of intervals: (0 quits) ");
       scanf("%d",&n);
             if(n == 0)
               MPI_Abort(MPI_COMM_WORLD, -1);
      }
      quad(n, g, MPI_COMM_WORLD, &pi);
      if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",  pi, fabs(pi - PI25DT));
  }

  MPI_Finalize();
}
```

Создание коммуникаторов

Разбиение коммуникатора на несколько:

int **MPI_Comm_split**(MPI_Comm comm, int color, int key, MPI_Comm* newcomm)

comm – «старый коммуникатор»

color – селектор коммуникатора

key – задает порядок на создаваемых коммуникаторах

newcomm – создаваемый коммуникатор

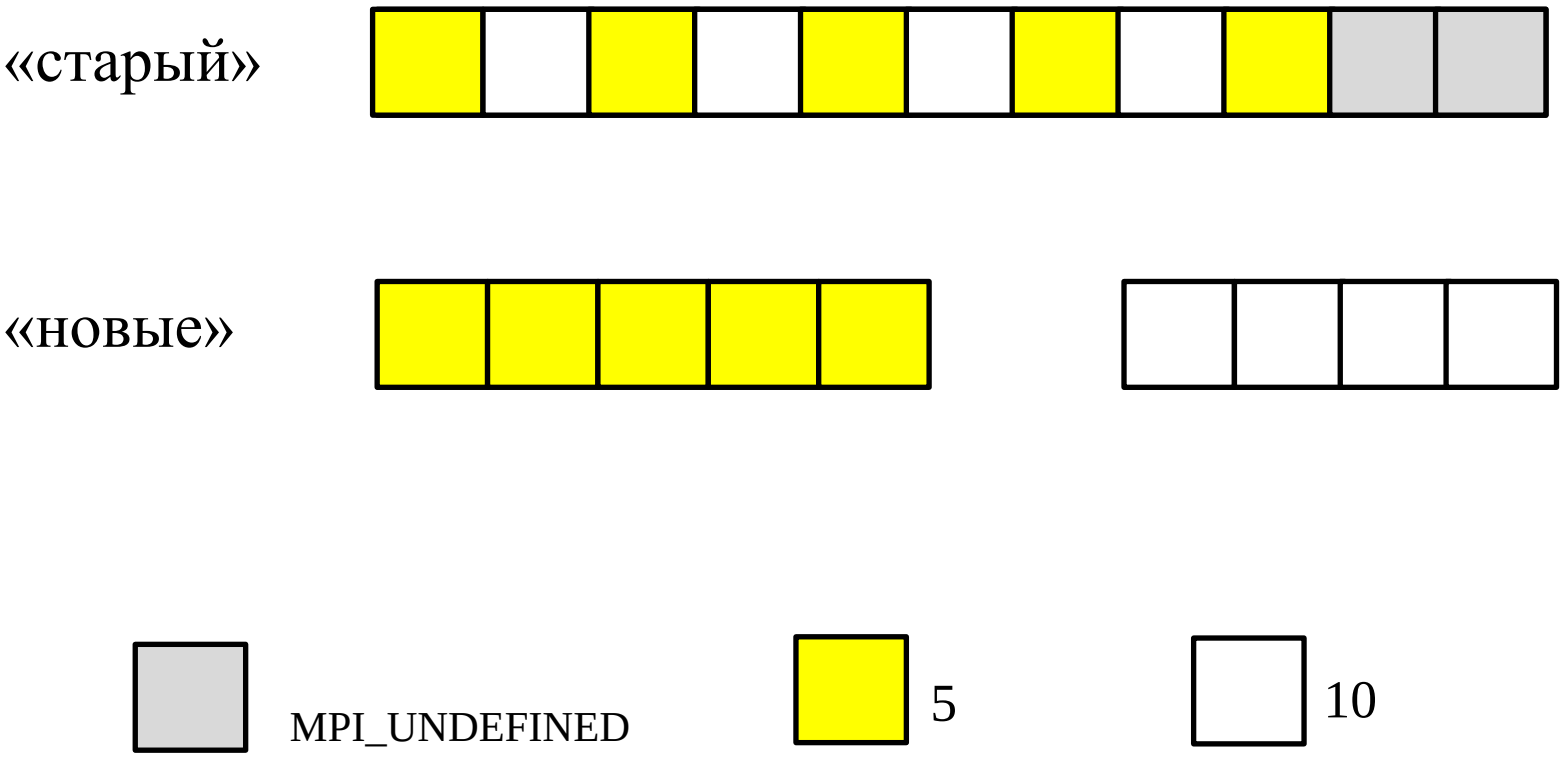
color >= 0

для

color = MPI_UNDEFINED будет создан коммуникатор MPI_COMM_NULL

ранги во вновь создаваемых коммуникаторах присваиваются в соответствии с возрастанием key

MPI_Comm_split



```
int main(int argc, char **argv)
{
  int n, myid;
  double PI25DT = 3.14159265358979323846264338;
  double pi, res;
  MPI_Comm comm;
  MPI_Comm rcomm;


  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  MPI_Comm_split(MPI_COMM_WORLD, myid % 2, 0, &comm);
  MPI_Comm_split(MPI_COMM_WORLD, (myid < 2) ? 1 : MPI_UNDEFINED, 0,
&rcomm);

  while (1)  {
    if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d",&n);
        if(n == 0)
           MPI_Abort(MPI_COMM_WORLD, -1);
    }
    if(myid < 2)
      MPI_Bcast(&n, 1, MPI_INT, 0, rcomm);
```

```
if(myid % 2)
      quad(n, g, comm, &pi);
    else
      quad(n, h, comm, &pi);

    if(myid < 2)
      MPI_Reduce(&pi, &res, 1, MPI_DOUBLE, MPI_SUM, 0, rcomm);
    if (myid == 0)
        printf("result is %f\n", res);
  }
  MPI_Finalize();
}
```

Группы и коммуникаторы

Совокупности MPI-процессов образуют **группы**.

Понятие ранга процесса имеет смысл только по отношению к определенной группе или коммуникатору.

Каждому интра-коммуникатору соответствует группа процессов. По группе процессов можно построить коммуникатор.

Информационные функции для работы с группами

Определение размера группы:

int MPI_Group_size(MPI_Group group, int *size)

group – группа;

size – указатель на область памяти для записи информации о количестве процессов в группе;

Определение номера процесса, выполняющего вызов функции, в группе:

int MPI_Group_rank(MPI_Group group, int *rank)

group – группа;

rank – указатель на область памяти для сохранения номера процесса;

Информационные функции для работы с группами

Установление соответствия между номерами процессов в различных группах:

int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1, MPI_Group group2, int *ranks2)

group1 – первая группа;

n – число элементов массивов **ranks1** и **ranks2**;

ranks1 – массив номеров процессов в первой группе;

group2 – вторая группа;

ranks2 – массив для сохранения номеров процессов во второй группе;

Эта функция заполняет массив **ranks2** номерами процессов в группе **group2**, которые имеют номера, перечисленные в **ranks**1 в группе **group**1.

Информационные функции для работы с группами

Сравнение двух групп процессов:

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

group1 – первая группа;

group2 – вторая группа;

result – указатель на область памяти для сохранения результата;

Если группа **group1** содержит те же процессы, что и группа **group2**, и порядок процессов в этих группах совпадает, группы считаются одинаковыми и по адресу **result** записывается константа **MPI_IDENT**, в противном случае результатом будет **MPI_UNEQUAL**.

Предопределенные группы

Педопределенные группы:
MPI_GROUP_EMPTY – «пустая» группа (не содержит процессов);
MPI_GROUP_NULL – «нулевая группа» (не соответствует никакой группе, аналог NULL.
Освобождение памяти, отведенной для группы:

Конструкторы и деструткоры групп

int MPI_Group_free(MPI_Group* group)

group – идентификатор освобождаемой группы.

Получение коммуникатора по группе:

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

comm – коммуникатор;

group – указатель на область памяти для сохранения полученной группы;

Объединение двух групп:

```
int MPI_Group_union(MPI_Group gr1, MPI_Group g2,
MPI_Group* gr3)
```

gr1 – первая группа;
gr2 – вторая группа;
gr3 – указатель на область для сохранения результата операции;

Набор процессов, входящих в gr3 получается объединением процессов, входящих в gr1 и gr2, причем элементы группы gr2, не вошедшие в gr1, следуют за элементами gr1.

Пересечение двух групп:

```
int MPI_Group_intersection(MPI_Group gr1, MPI_Group
g2, MPI_Group* gr3)
```

gr1 – первая группа;
gr2 – вторая группа;
gr3 – указатель на область для сохранения результата операции;

Группа gr3 составлена из процессов, входящих как в gr1,так и в gr2, расположенных в том же порядке,

Разность двух групп:

```
int MPI_Group_difference(MPI_Group gr1, MPI_Group
g2, MPI_Group* gr3)
      gr1 – первая группа;
      gr2 – вторая группа;
      gr3 – указатель на область для сохранения
результата операции;
Группа gr3 составлена из процессов, входящих в gr1,
но не входящих в gr2,  расположенных в том же
порядке, что и в gr1.
```

Переупорядочивание (с возможным удалением) процессов в существующей группе:

```
int MPI_Group_incl(MPI_Group group, int n, int*
ranks, MPI_Group* newgroup)
      group – исходная группа;
      n – число элементов в массиве ranks;
      ranks – массив номеров процессов, из которых
будет создана новая группа;
      newgroup – указатель на область для сохранения
результата операции;
Созданная группа newgroup содержит элементы группы
group, перечисленные в массиве ranks: i-й процесс
создаваемой группы newgroup совпадает с процессом,
имеющим номер ranks[i] в группе group.
```

Удаление процессов из группы:

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

group – исходная группа;

n – число элементов в массиве ranks;

ranks – массив номеров удаляемых процессов;

newgroup – указатель на область для сохранения результата операции;

В результате выполнения этой операции создается новая группа **newgroup**, получаемая удалением из исходной группы процессов с номерами, перечисленными в массиве **ranks**.

Создание коммуникатора по группе

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

Требования:

1. Конструктор вызывается на всех процессах коммуникатора **comm**;
2. **group** – подгруппа группы коммуникатора comm, одинакова на всех процессах.

Результат:

newcomm – на процессах, вошедших в **group**, новый коммуникатор, на остальных – **MPI_COMM_NULL**

Удаление коммуникатора

Освобождение коммуникатора:

```
int MPI_Comm_free(MPI_Comm *comm)
```

При освобождении коммуникатора все незавершенные операции будут завершены, только после этого коммуникатор будет удален физически.

Вычисление числа Pi методом Монте-Карло

Из книги Gropp, Lusk, Skjellum

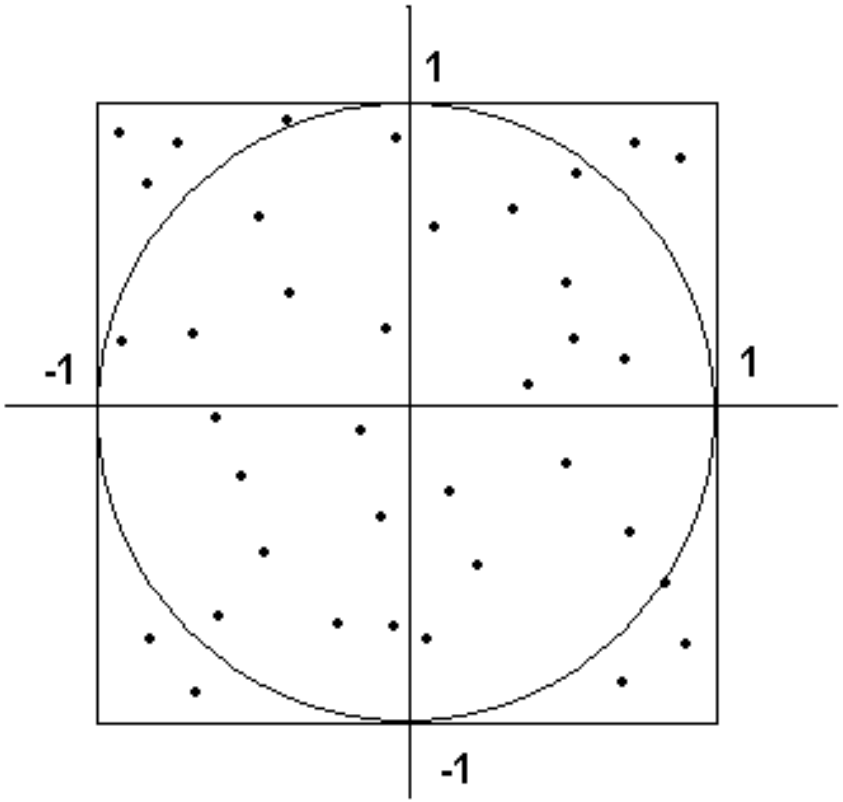
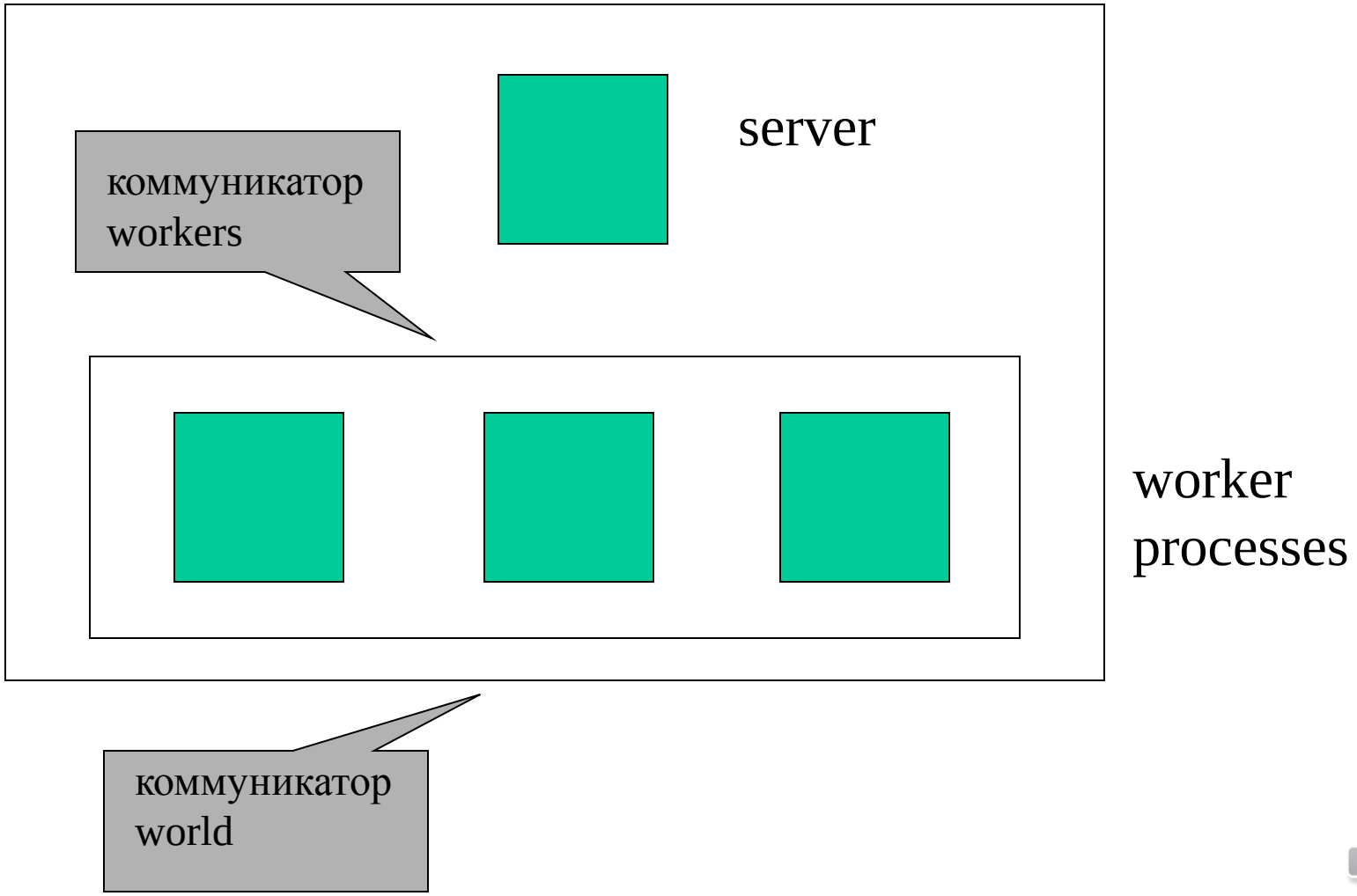


Схема вычислений



```
/* compute pi using Monte Carlo method */
#include <math.h>
#include "mpi.h"

#define CHUNKSIZE      1000
#define INT_MAX 1000000000
/* message tags */
#define REQUEST  1
#define REPLY    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;

    MPI_Init(&argc,&argv);
    world  = MPI_COMM_WORLD;
    MPI_Comm_size(world,&numprocs);
    MPI_Comm_rank(world,&myid);
```

```
server = numprocs-1;
  if (myid == 0)
    sscanf( argv[1], "%lf", &epsilon );
MPI_Bcast( &epsilon, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD );
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);

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; i++)
                                rands[i] = random();
                            MPI_Send(rands, CHUNKSIZE, MPI_INT,
                                  status.MPI_SOURCE, REPLY, world);
                    }
             } while( request>0 );
  }
```

```
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, &status );
          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);
        error = fabs( Pi-3.14159265358979323846264643);
        done = (error < epsilon || (totalin+totalout) > 1000000);
        request = (done) ? 0 : 1;
        if (myid == 0) {
            printf( "\rpi = %f", Pi );
            MPI_Send( &request, 1, MPI_INT, server, REQUEST,
                      world );
        }
        else {
            if (request)
                MPI_Send(&request, 1, MPI_INT, server, REQUEST,
                         world);
        }
    }
}
```

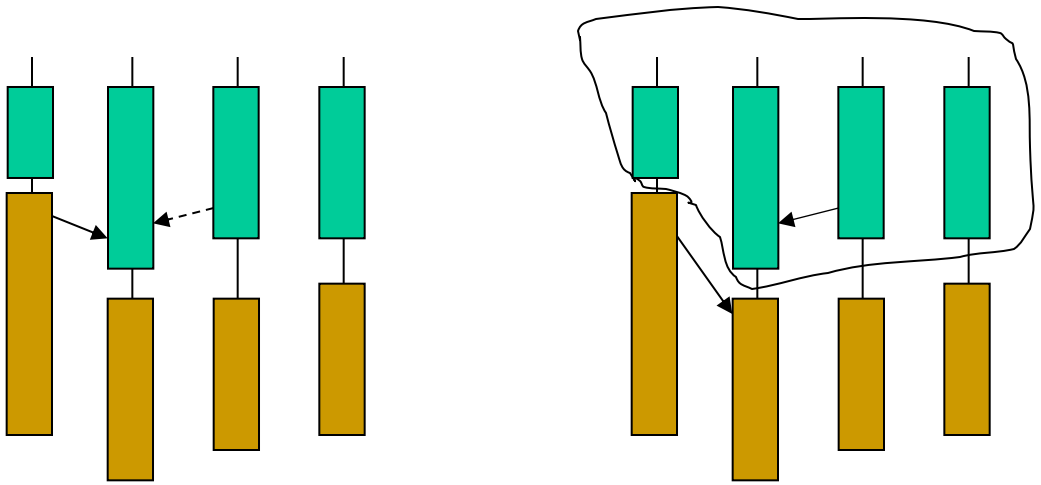
```
 if (myid == 0) {
        printf( "\npoints: %d\nin: %d, out: %d, <ret> to exit\n",
               totalin+totalout, totalin, totalout );
        getchar();
    }
    MPI_Comm_free(&workers);
    MPI_Finalize();
}
```

Дублирование коммуникатора

Получение дубликата коммуникатора:

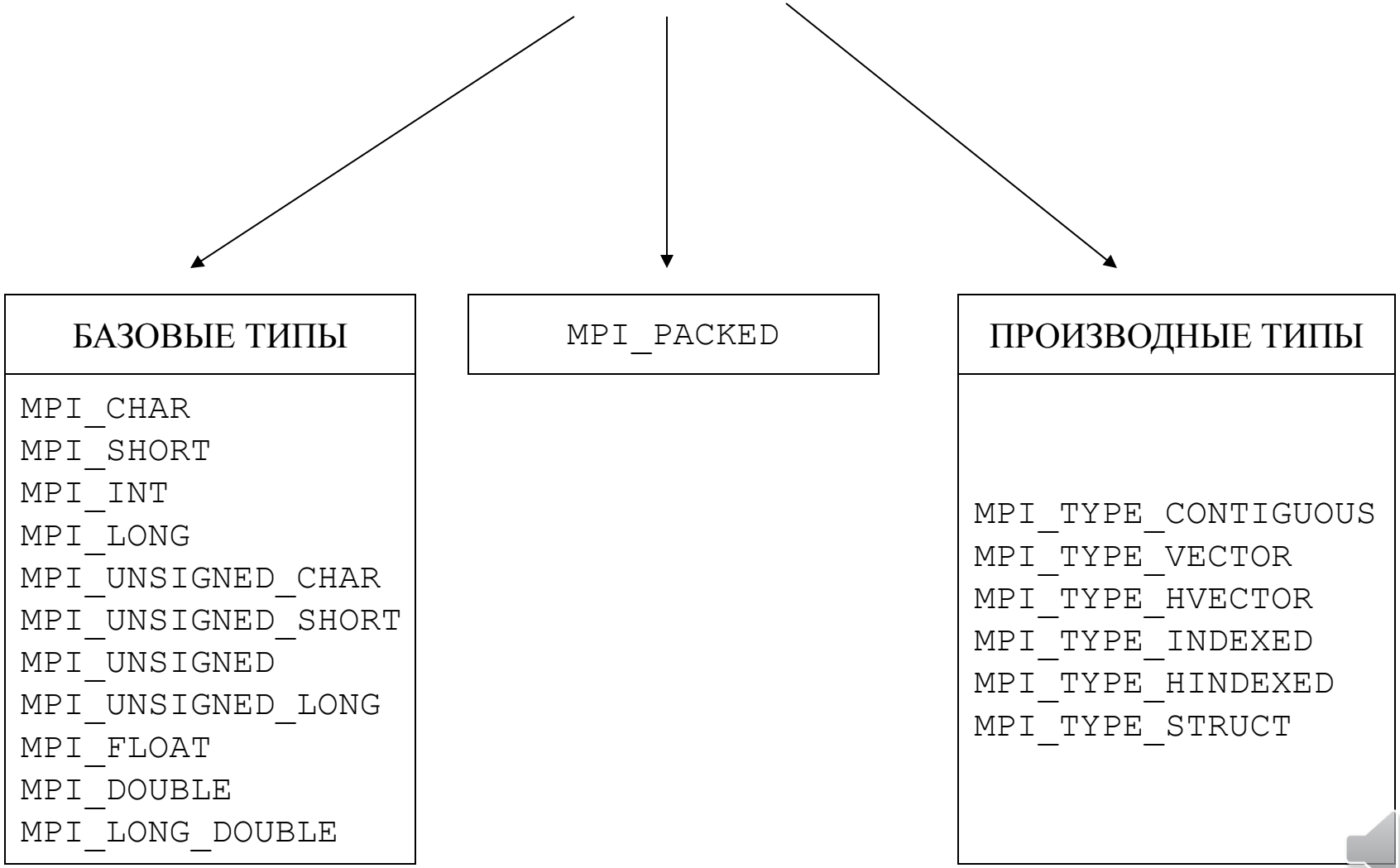
int **MPI_Comm_dup**(MPI_Comm comm, MPI_Comm* newcomm)

Используется для того, чтобы снабдить библиотечную новым коммуникатором, совпадающим по характеристикам со старым, но вместе с тем, создающим новый контекст для коммуникаций. Цель: исключить проблемы, связанные с «перемешиванием коммуникаций».



Система типов сообщений MPI

Типы в MPI


БАЗОВЫЕ ТИПЫ
MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_UNSIGNED_CHAR
MPI_UNSIGNED_SHORT
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
MPI_PACKED
ПРОИЗВОДНЫЕ ТИПЫ
MPI_TYPE_CONTIGUOUS
MPI_TYPE_VECTOR
MPI_TYPE_HVECTOR
MPI_TYPE_INDEXED
MPI_TYPE_HINDEXED
MPI_TYPE_STRUCT

БАЗОВЫЕ ТИПЫ

тип MPI	тип Си
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

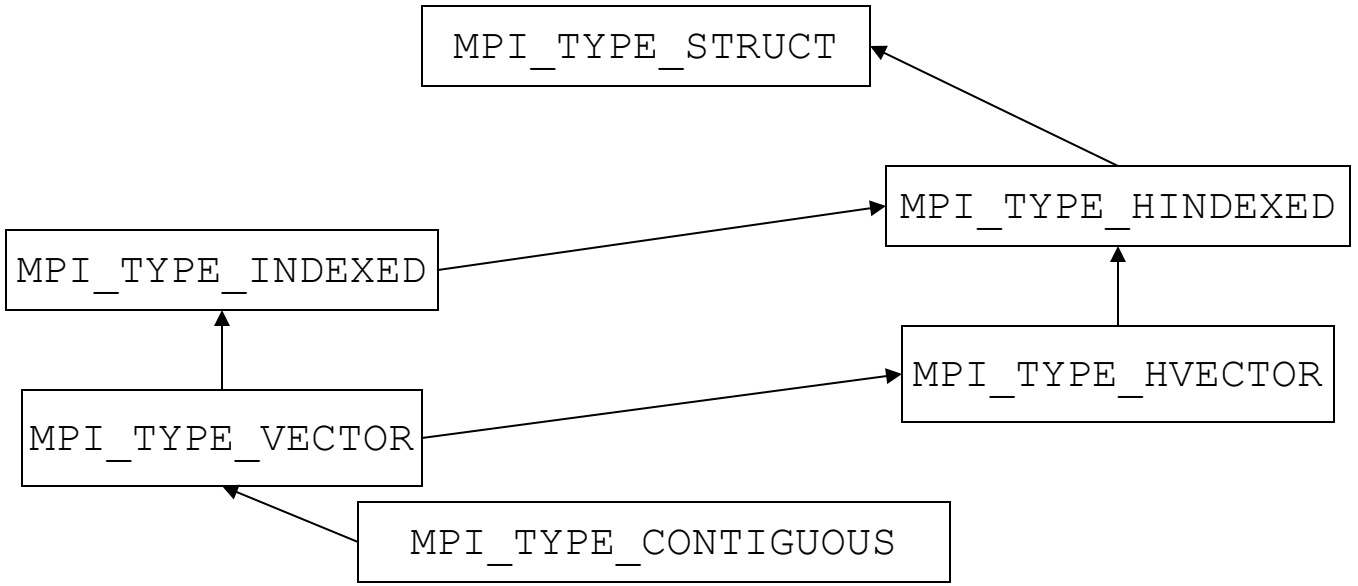
ПРОИЗВОДНЫЕ ТИПЫ

MPI_TYPE_CONTIGUOUS – массив «без дырок»;

MPI_TYPE_VECTOR, MPI_TYPE_HVECTOR – регулярно (с постоянным шагом) расположенные в памяти блоки однотипных элементов;

MPI_TYPE_INDEXED,MPI_TYPE_HINDEXED – произвольно расположенные блоки однотипных элементов;

MPI_TYPE_STRUCT – произвольно расположенные в памяти блоки элементов произвольных типов;

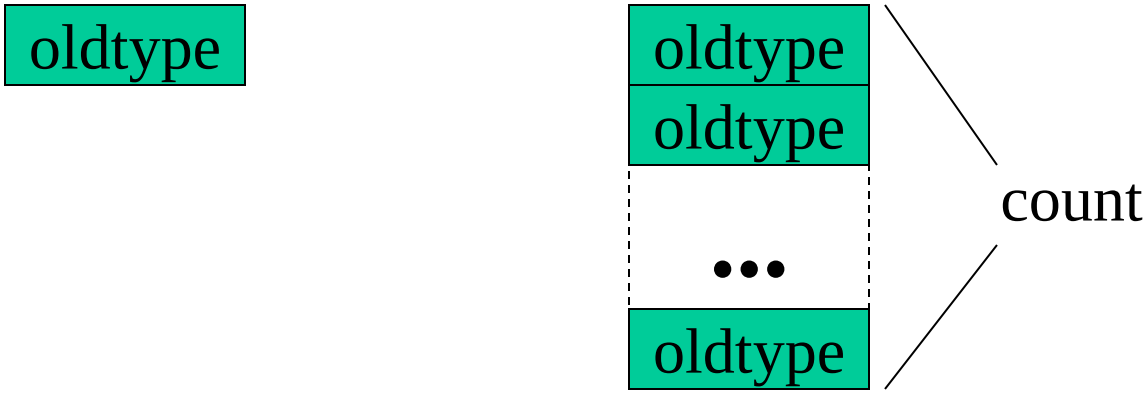


Назначение производных типов

- пересылка данных, расположенных в несмежных областях памяти в одном сообщении;
- пересылка разнотипных данных в одном сообщении;
- облегчение понимания программы;

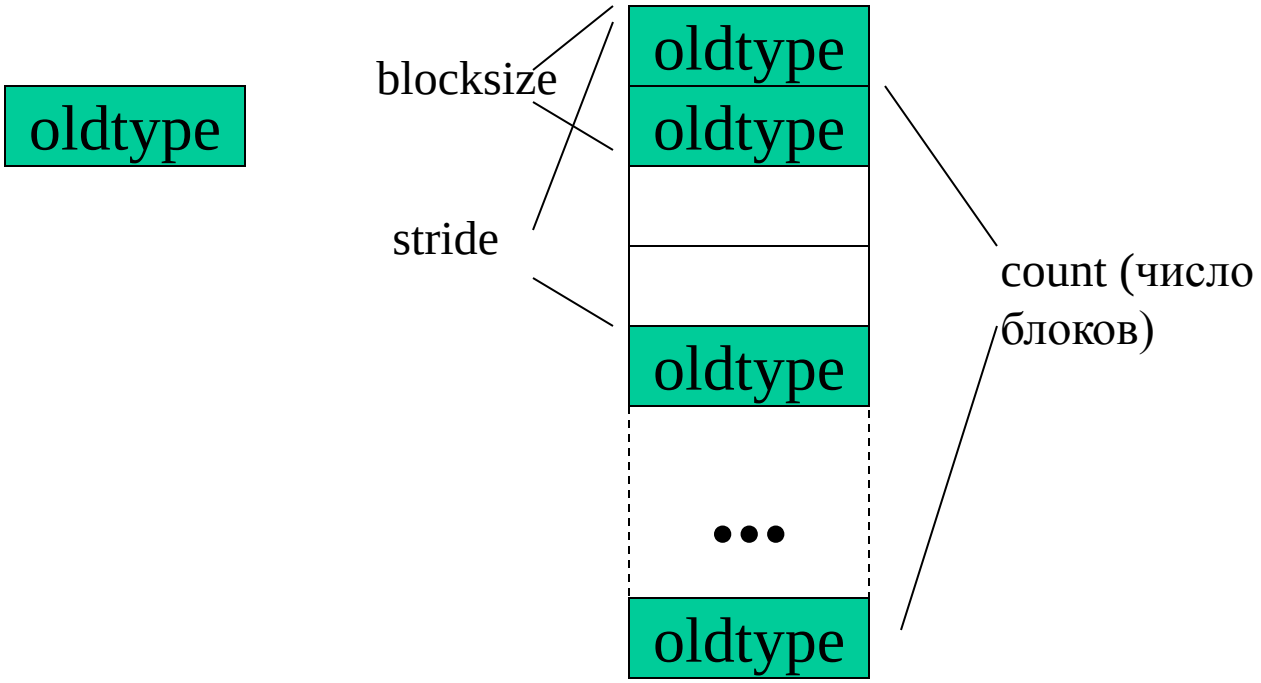
MPI_TYPE_CONTIGUOUS

MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

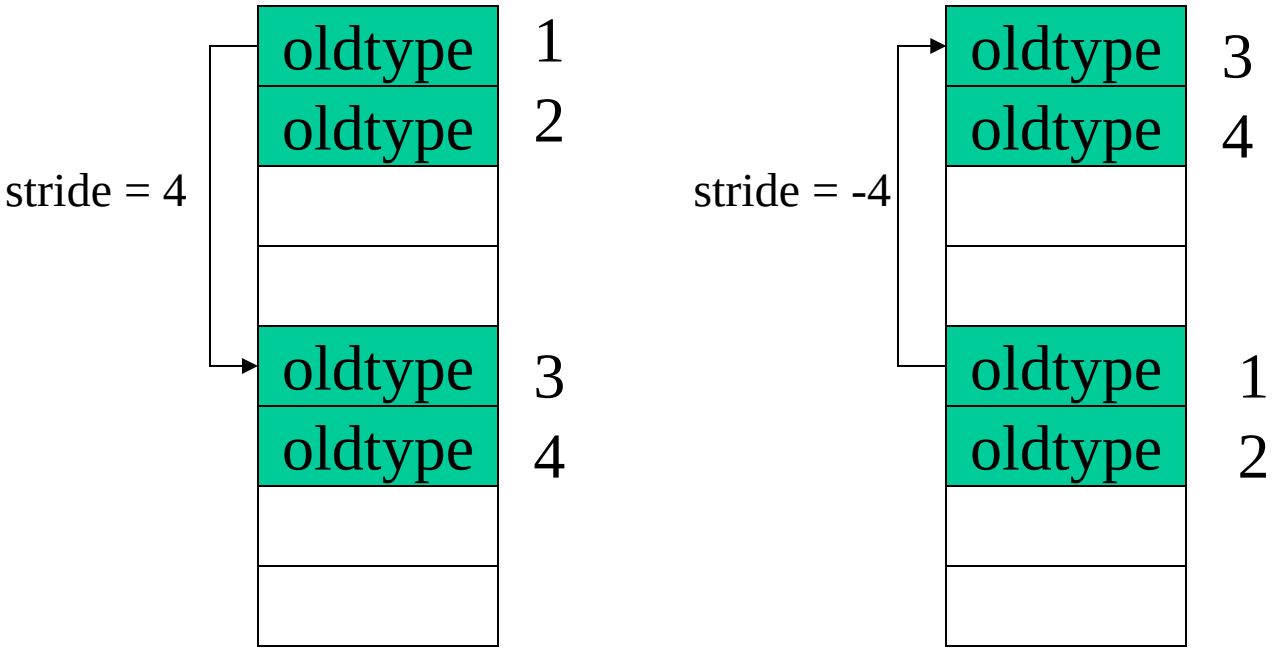


MPI_TYPE_VECTOR

MPI_Type_vector(int count, int blocksize, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

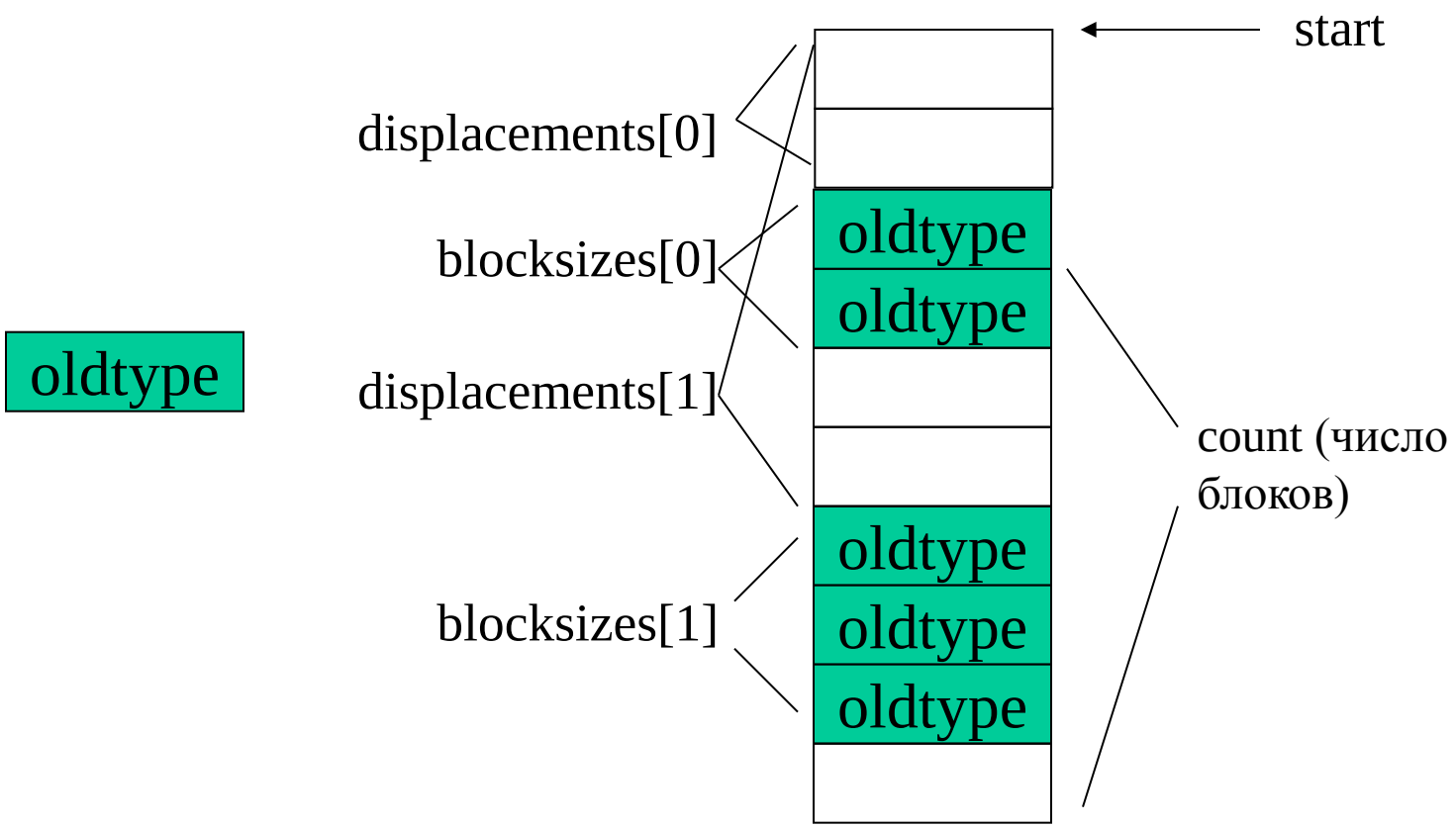


Отрицательный шаг



MPI_TYPE_INDEXED

MPI_Type_indexed(int count, int* blocksizes, int* displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)



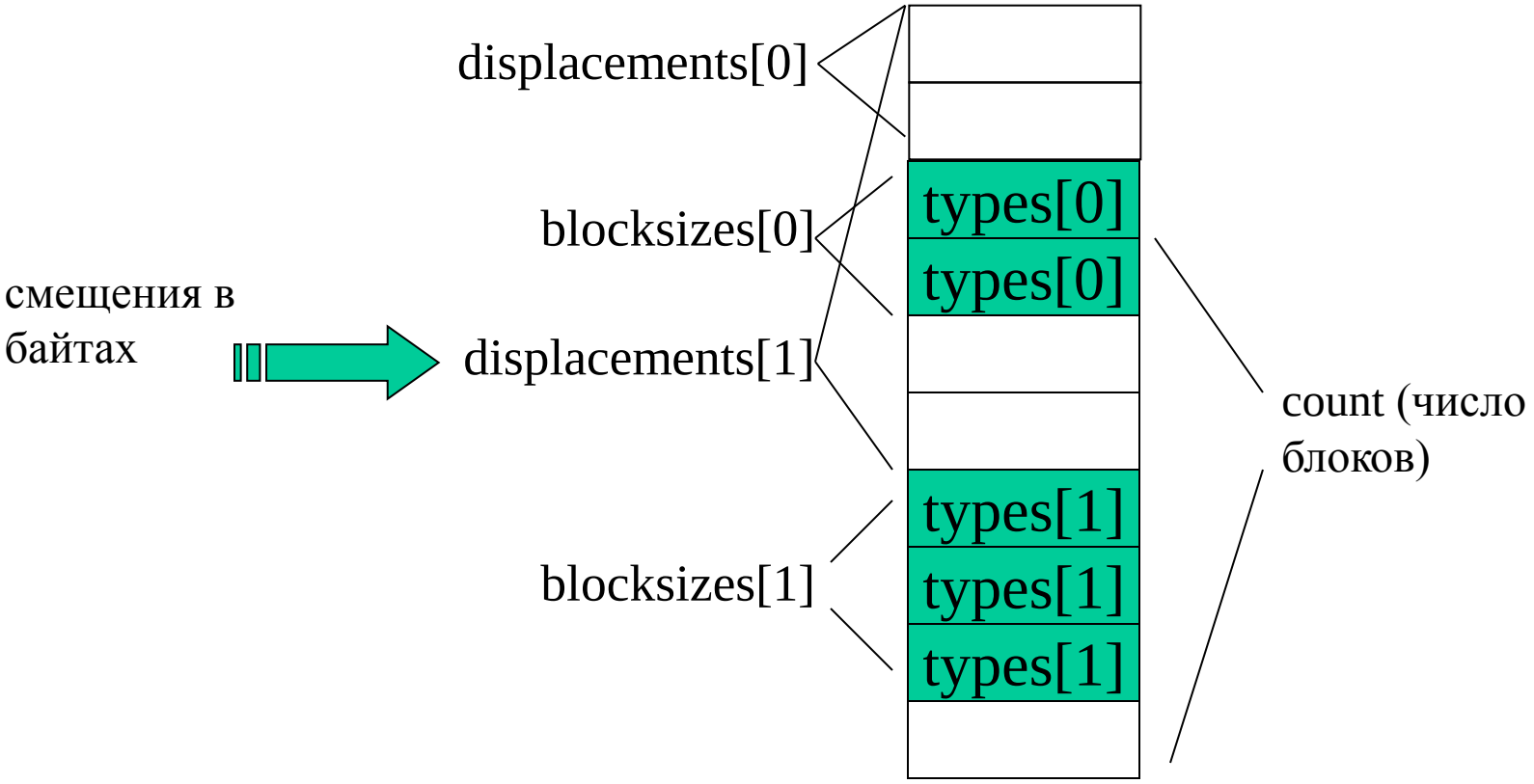
MPI_TYPE_HVECTOR
MPI_TYPE_HINDEXED

основное отличие: смещение задается в байтах –

необходимо знать точные значения размеров и требований выравнивания для типов в конкретной архитектуре;

MPI_TYPE_STRUCT

MPI_Type_struct(int count, int* blocksizes, int* displacements, MPI_Datatype *types, MPI_Datatype *newtype)



РЕГИСТРАЦИЯ ТИПА

Регистрация типа:

int MPI_Type_commit(MPI_Datatype *datatype)

Освобождение памяти:

int MPI_Type_free(MPI_Datatype *datatype)

- типы, которые строились на основе освобождаемого, не подвергаются воздействию;
- все коммуникации с использованием этого типа завершаются корректно.

ПОРЯДОК РАБОТЫ С ПРОИЗВОДНЫМИ ТИПАМИ

- Создание типа с помощью конструктора.
- Регистрация.
- Использование.
- Освобождение памяти.

КАРТА И СИГНАТУРА ТИПА

Карта типа - набор пар (базовый тип, смещение):

$((type_1, disp_1), (type_2, disp_2), ..., (type_n, disp_n))$,

соответствующая сигнатура типа – набор базовых типов:

$(type_1, type_2, ..., type_n)$.

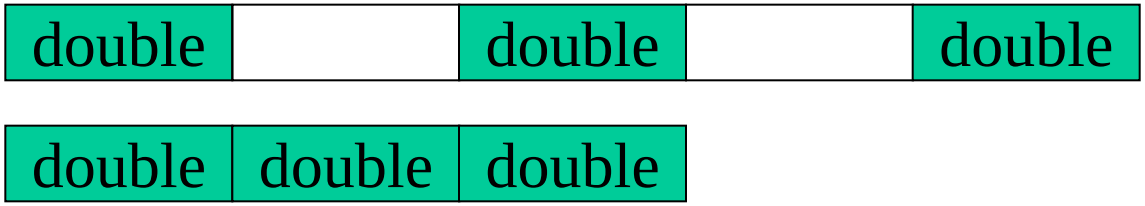
СООТВЕТСТВИЕ ТИПОВ

Соответствие типов отправителя и получателя:

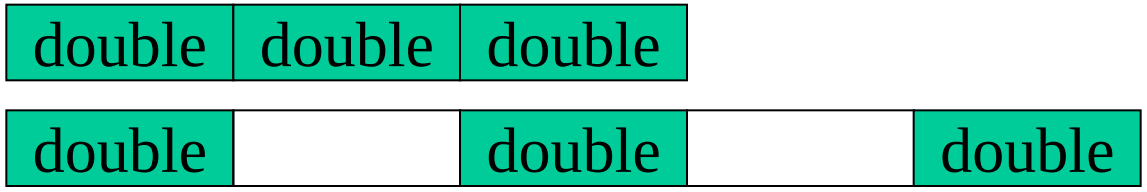
Сигнатура типа пришедшего сообщения является начальной подпоследовательностью сигнатуры типа, указанного в операции приема.

ПРИМЕРЫ

send: MPI_TYPE_VECTOR(3, 1, 2, MPI_DOUBLE)
recv: MPI_TYPE_CONTIGUOUS(3, MPI_DOUBLE)

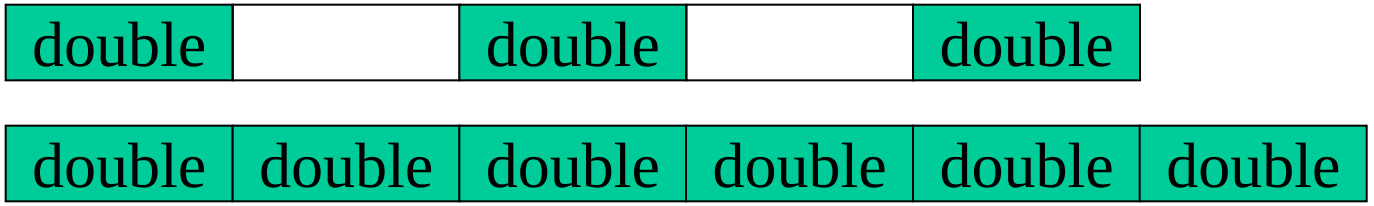


send:MPI_TYPE_CONTIGUOUS(3, MPI_DOUBLE)
recv: MPI_TYPE_VECTOR(3, 1, 2, MPI_DOUBLE)

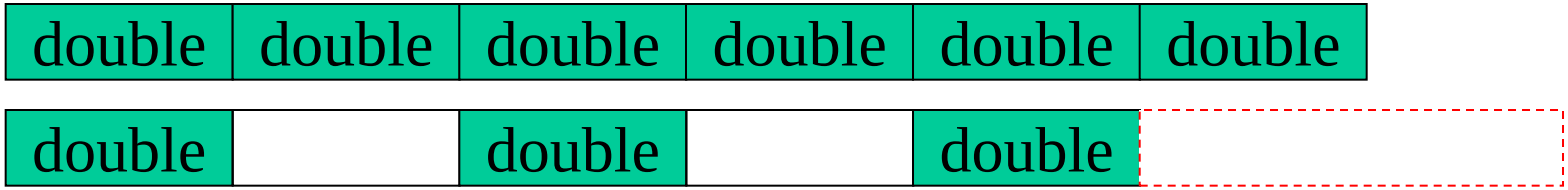


ПРИМЕРЫ

send: MPI_TYPE_VECTOR(3, 1, 2, MPI_DOUBLE)
recv: MPI_TYPE_CONTIGUOUS(6, MPI_DOUBLE)

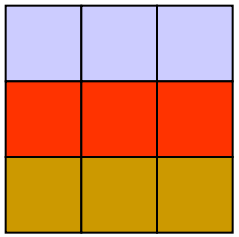


send: MPI_TYPE_CONTIGUOUS(6, MPI_DOUBLE)
recv:MPI_TYPE_VECTOR(3, 1, 2, MPI_DOUBLE)

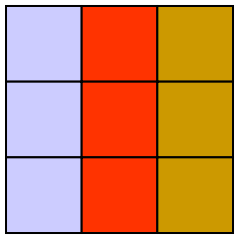


ТРАНСПОНИРОВАНИЕ МАТРИЦЫ

A



A^T



Процесс #1

Процесс #2

```
#include <stdio.h>
#include <mpi.h>

#define N 3
int A[N][N];

void fill_matrix()
{
  int i,j;
  for(i = 0; i < N; i ++)
    for(j = 0; j < N; j ++)
      A[i][j] = i * N + j;
}

void print_matrix()
{
  int i,j;
  for(i = 0; i < N; i ++) {
    for(j = 0; j < N; j ++)
      printf("%d ", A[i][j]);
    printf("\n");
  }
}
```

```
main(int argc, char* argv[])
{
  int r, i;
  MPI_Status st;
  MPI_Datatype typ;


  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &r);

  if(r == 0) {
    fill_matrix();
    printf("\n Source:\n");
    print_matrix();
    MPI_Type_contiguous(N, MPI_INT, &typ);
    MPI_Type_commit(&typ);
    for(i = 0; i < N; i ++)
      MPI_Send(&(A[i][0]), 1, typ, 1, 0, MPI_COMM_WORLD);
    }
```

```
else if(r == 1){
    MPI_Type_vector(N, 1, N, MPI_INT, &typ);
    MPI_Type_commit(&typ);
    for(i = 0; i < N; i ++)
      MPI_Recv(&(A[0][i]), 1, typ, 0, 0, MPI_COMM_WORLD, &st);
    printf("\n Transposed:\n");
    print_matrix();
  }

  MPI_Type_free(&typ);
  MPI_Finalize();
}
```

РЕЗУЛЬТАТ РАБОТЫ

Transposed:
0 3 6
1 4 7
2 5 8

Source:
0 1 2
3 4 5
6 7 8

ОДНА ПЕРЕСЫЛКА

```
if(r == 0) {
   fill_matrix();
   printf("\n Source:\n");
   print_matrix();
   MPI_Type_contiguous(N * N, MPI_INT, &typ);
   MPI_Type_commit(&typ);
   MPI_Send(&(A[0][0]), 1, typ, 1, 0, MPI_COMM_WORLD);
 } else if(r == 1){
   MPI_Type_vector(N, 1, N, MPI_INT, &typ);
   MPI_Type_hvector(N, 1, sizeof(int), typ, &typ1);
   MPI_Type_commit(&typ1);
   MPI_Recv(&(A[0][0]), 1, typ1, 0, 0, MPI_COMM_WORLD, &st);
   printf("\n Transposed:\n");
   print_matrix();
 }
```

УПАКОВКА СООБЩЕНИЙ

- дает возможность пересылать разнородные данные в одном сообщении;
- отделяет операцию формирования сообщения от операции пересылки;
- способствует развитию библиотек на базе MPI;

MPI_PACK

int **MPI_Pack**(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outcount, int *position,, MPI_Comm comm)

inbuf – буфер с данными для запаковки;
incount – число элементов для запаковки;
datatype – тип элементов данных;
outbuf – буфер сообщения;
outcount – размер буфера сообщения;
position – позиция в буфере сообщения, с которой заполнять буфер (изменяется);
comm – коммуникатор, по которому сообщение будет посылаться;

MPI_UNPACK

int **MPI_Unpack**(void* inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)

inbuf – буфер сообщения;
insize – размер inbuf
position – позиция, с которой распаковывать данные (изменяется);
outbuf – буфер для распаковки;
outcount – число элементов для распаковки;
datatype – тип элементов данных;
comm – коммуникатор, по которому сообщение будет посылаться;

ОПРЕДЕЛЕНИЕ РАЗМЕРА СООБЩЕНИЯ

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)

incount – число элементов данных в сообщении;

datatype – тип элементов данных;

comm – коммуникатор;

size – размер сообщения;

ПРИМЕР

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define N 3


main(int argc, char* argv[])
{
  int r;
  int i;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &r);
```

ПРИМЕР

```
if(r == 0){
    int sz;
    int pos = 0;
    int a = 1;
    void* buf;

    MPI_Pack_size(N, MPI_INT, MPI_COMM_WORLD, &sz);
    buf = (void*) malloc(sz);
    for(i = 0; i < N; i ++) {
      MPI_Pack(&a, 1, MPI_INT, buf, sz, &pos, MPI_COMM_WORLD);
      a ++;
    }

    MPI_Send(buf, pos, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
  }
```

ПРИМЕР

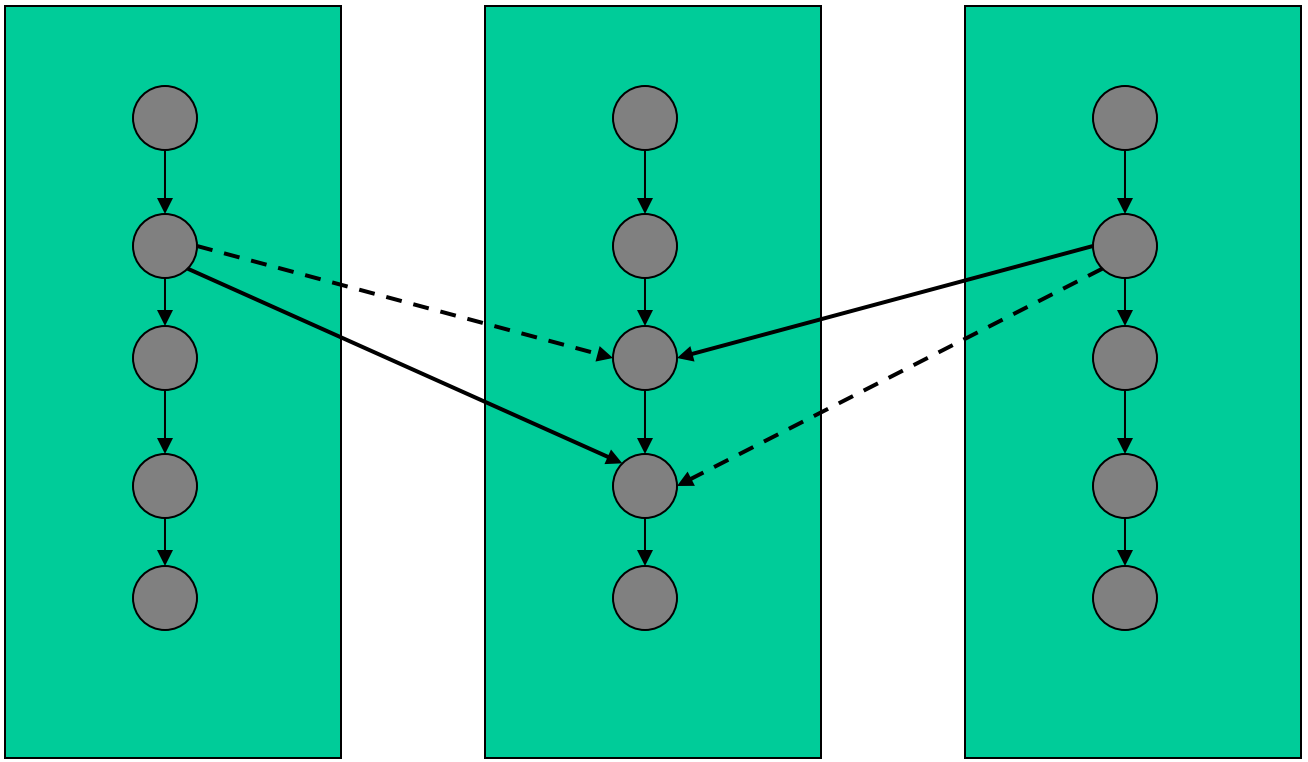
```
else {
    MPI_Status st;
    int A[N];

    MPI_Recv(A, N, MPI_INT, 0, 0, MPI_COMM_WORLD, &st);
    for(i = 0; i < N; i ++)
      printf("%d ", A[i]);

  }

  MPI_Finalize();
}
```

ХАРАКТЕРНЫЕ ОШИБКИ В MPI-ПРОГРАММАХ

ВИДЫ ОШИБОК

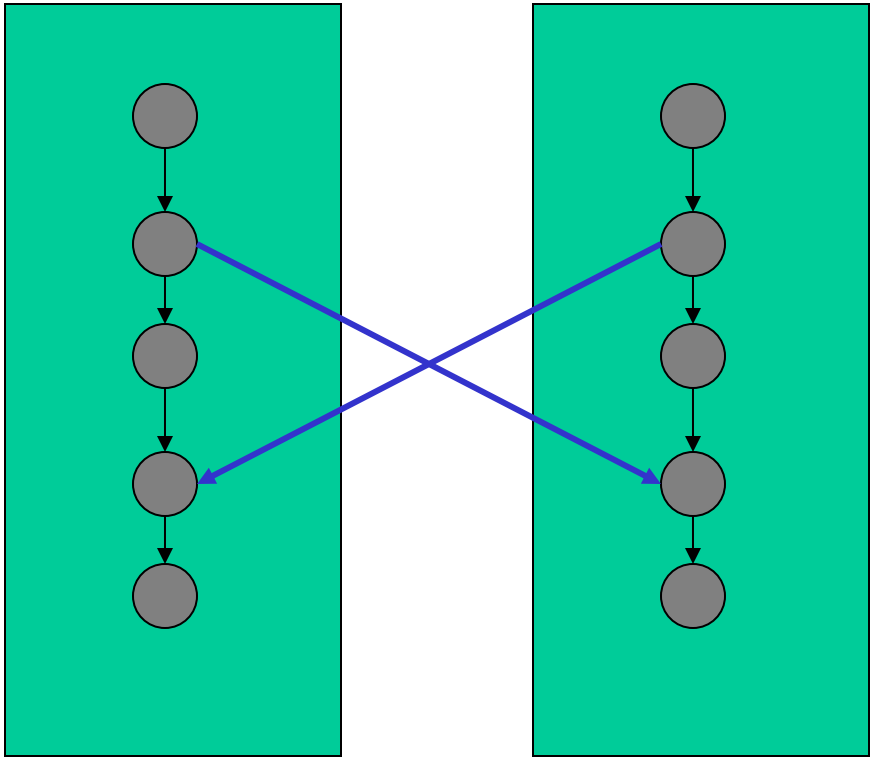
- Ошибки последовательных программ.
- Ошибки несоответствия типов.
- Ошибки работы с MPI-объектами.
- Взаимные блокировки.
- Недетерминизм.

Недетерминизм за счет разницы в относительных скоростях процессов (race condition)



Deadlock

```
if(rank == 0) {
        MPI_Ssend(… 1 …)
        MPI_Recv(…1…)
} else {
        MPI_Ssend(… 0 …)
        MPI_Recv(…0…)
}
```



«Недетерминированный» deadlock

```
if(rank == 0) {
        MPI_Send(… 1 …)
        MPI_Recv(…1…)
} else {
        MPI_Send(… 0 …)
        MPI_Recv(…0…)
}
```