Введение в параллельные вычисления

д.т.н. Мокрова Наталия Владиславовна

асп. Морунов Егор, Сырко Денис

пятница	ауд. 118, 119	
12:50 – 14:20	Лекция	
	1 неделя	2 неделя
14:35 - 17:55	Лаб – КС44	Лаб - КС40

Выбор модели программирования

Системы с общей памятью

- C++11 threads (pthreads)
- OpenMP

Системы с распределенной памятью

- MPI = message passing interface

Гетерогенные системы (графические процессоры)

- CUDA

Лекция 13. MPI. Обмен сообщениями. Взаимодействие процессов

Стандарты MPI http://www.mpi-forum.org

Обмен данными с использованием MPI.

- http://habrahabr.ru/company/intel/blog/251357/

Лекции и семинары

- http://www.slideshare.net/Aleximos/mpi-9793227

Технологии

- http://parallel.ru/tech/tech_dev/mpi.html
- http://parallel.ru/tech/tech_dev/MPI/examples/ (примеры)

Примеры из учебника "Технологии параллельного программирования MPI и OpenMP"

- http://parallel.ru/tech/tech_dev/MPI%26OpenMP/examples/

Двусторонние обмены (Point-to-point)

Блокирующие (Blocking)

- MPI_Bsend
- MPI_Recv
- MPI_Rsend
- MPI_Send
- MPI_Sendrecv
- MPI_Sendrecv_replace
- MPI_Ssend
- ...

Неблокирующие (Non-blocking)

- MPI_Ibsend
- MPI_Irecv
- MPI_Irsend
- MPI_Isend
- MPI_Issend
- ...

Проверки состояния запросов (Completion/Testing)

- MPI_Iprobe
- MPI_Probe
- MPI_Test{, all, any, some}
- MPI_Wait{, all, any, some}
- ...

Постоянные (Persistent)

- MPI_Bsend_init
- MPI_Recv_init
- MPI_Send_init
- ...
- MPI_Start
- MPI_Startall

Прием/передача сообщений без блокировки

int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int msgtag,
MPI_Comm comm, MPI_Request *request)

OUT *request* – идентификатор асинхронной передачи

Передача аналогична *MPI_Send*, но возврат из подпрограммы сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере *buf*.
Нельзя повторно использовать буфер без получения дополнительной информации о завершении данной посылки.
Сообщение, отправленное любой из процедур *MPI_Send* и *MPI_Isend*, может быть принято любой из процедур *MPI_Recv* и *MPI_Irecv*.
Предусмотрены три дополнительных варианта, подобные модификациям процедуры *MPI_Send*.

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int msgtag,
MPI_Comm comm, MPI_Request *request)

OUT *request* – идентификатор асинхронного приема сообщения

Прием сообщения аналогичный *MPI_Recv*, однако возврат из подпрограммы происходит сразу после инициализации процесса приема без ожидания получения сообщения в буфере *buf*.
Окончание процесса приема можно определить с помощью параметра *request* и процедур *MPI_Wait* и *MPI_Test*.

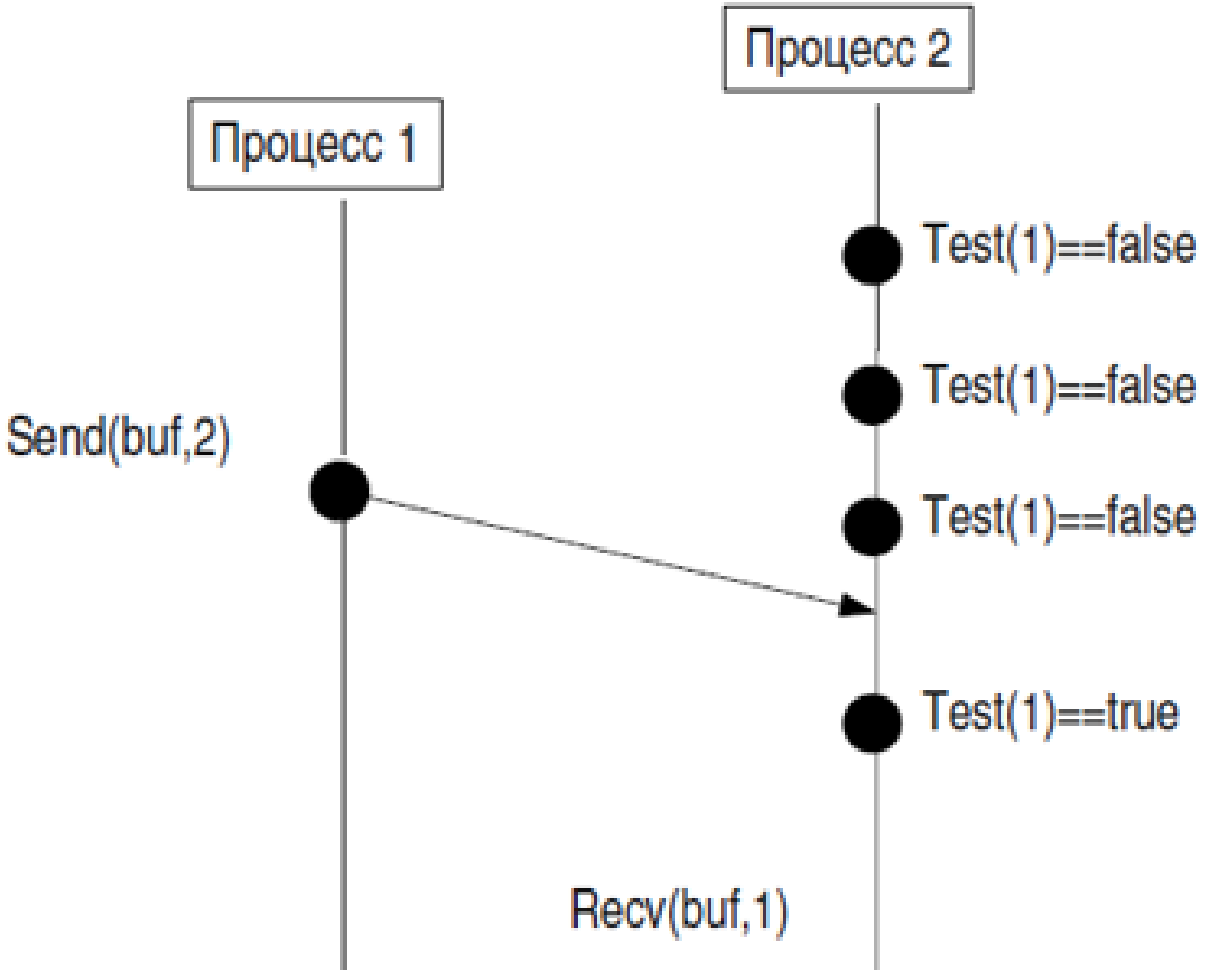
Неблокирующий обмен сообщениями

Асинхронная передача данных.

Возврат сразу после вызова без остановки работы процессов.

Для завершения асинхронного обмена требуются дополнительные процедуры, с целью использования буфера.

До завершения неблокирующей операции нельзя записывать в используемый массив данных.



Проверка состояния

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

request – идентификатор асинхронного приема или передачи;
OUT *flag* – признак завершенности операции обмена;
OUT *status* – параметры сообщения.

Проверка завершенности асинхронных процедур *MPI_Isend* или *MPI_Irecv*, ассоциированных с идентификатором *request*. В параметре *flag* – 1, если соответствующая операция завершена, и значение 0 в противном случае. Если завершена процедура приема, то атрибуты и длину полученного сообщения можно определить с помощью параметра *status*.

int MPI_Iprobe(int source, int msgtag, MPI_Comm comm, int *flag, MPI_Status *status)

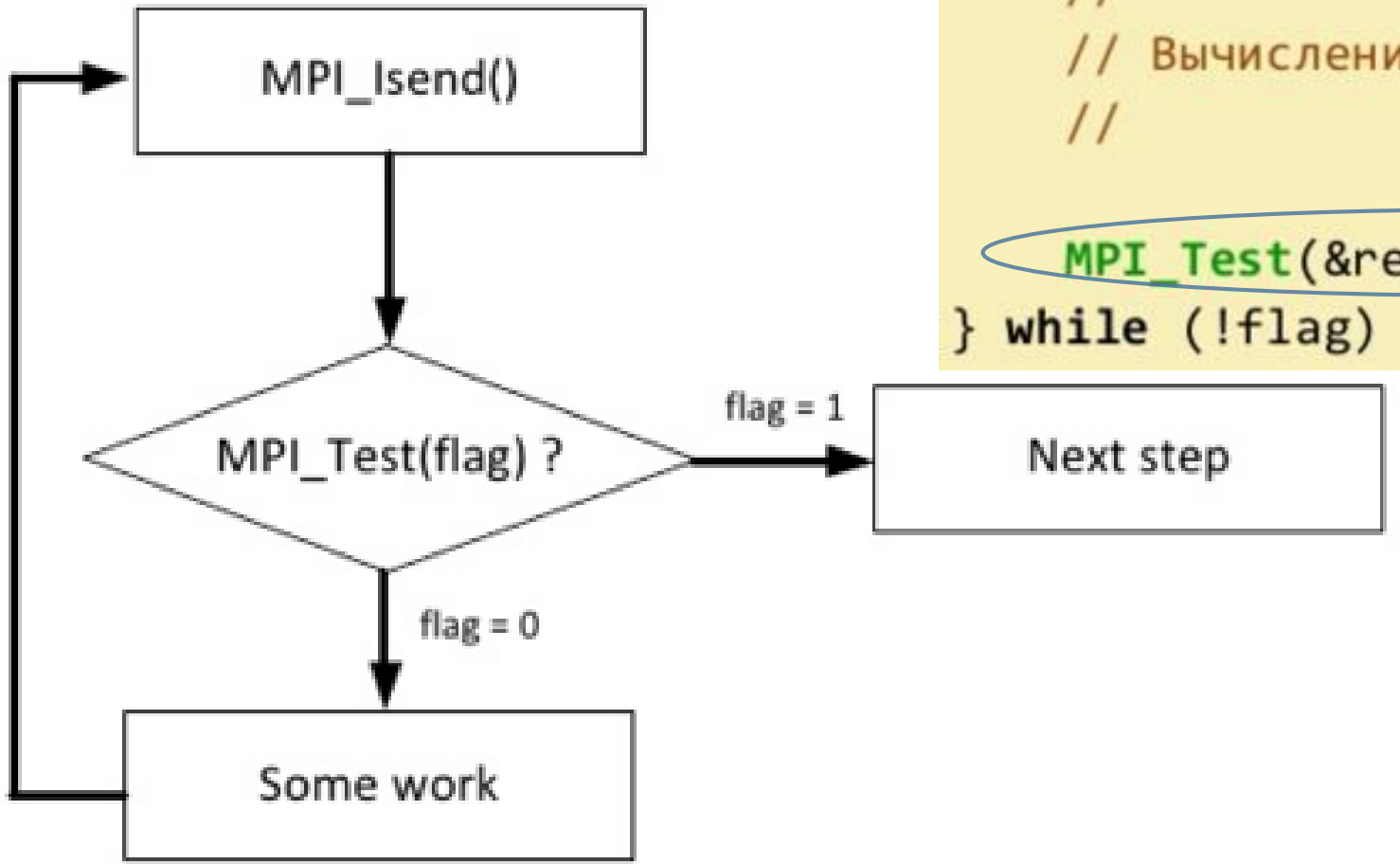
source – номер процесса-отправителя или *MPI_ANY_SOURCE;*
msgtag – идентификатор ожидаемого сообщения или *MPI_ANY_TAG;*
OUT *status* – параметры обнаруженного сообщения.

Получение информации в массиве status о поступлении и структуре ожидаемого сообщения без блокировки. В параметре *flag* значение 1, если сообщение с подходящими атрибутами уже может быть принято (в этом случае ее действие полностью аналогично *MPI_Probe*), и значение 0, если сообщения с указанными атрибутами еще нет.

Совмещение обменов и вычислений

```
MPI_Isend(buf, count, MPI_INT, 1, 0,
          MPI_COMM_WORLD, &req);
do {
    //
    // Вычисления ... (не использовать buf)
    //

    MPI_Test(&req, &flag, &status);
} while (!flag)
```



Пример MPI_Probe

Процедура MPI_Probe использована для определения структуры приходящего сообщения.

Процесс 0 ждет сообщения от любого из процессов 1 и 2 с одинаковым тегом.

Посылаемые данные имеют разный тип.

Чтобы определить в какую переменную помещать приходящее сообщение, процесс определяет от кого оно поступило.

Следующий после MPI_Probe вызов MPI_Recv гарантировано примет нужное сообщение, далее принимается сообщение от другого процесса.

```
Process 0 recv 1 from process 1, 2.000000 from process 2
```

```
int main(int argc, char **argv)
{ int rank, size, ibuf; float rbuf;
   MPI_Status status;     MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
ibuf = rank;                rbuf = 1.0 * rank;
  if(rank == 1) MPI_Send(&ibuf, 1, MPI_INT, 0, 5, MPI_COMM_WORLD);
  if(rank == 2) MPI_Send(&rbuf, 1, MPI_FLOAT, 0, 5, MPI_COMM_WORLD);
  if(rank == 0){
    MPI_Probe(MPI_ANY_SOURCE, 5, MPI_COMM_WORLD, &status);
    if (status.MPI_SOURCE == 1) {
      MPI_Recv(&ibuf, 1, MPI_INT, 1, 5, MPI_COMM_WORLD, &status);
      MPI_Recv(&rbuf, 1, MPI_FLOAT, 2, 5, MPI_COMM_WORLD, &status);
    }
    else if (status.MPI_SOURCE == 2) {
      MPI_Recv(&rbuf, 1, MPI_FLOAT, 2, 5, MPI_COMM_WORLD, &status);
      MPI_Recv(&ibuf, 1, MPI_INT, 1, 5, MPI_COMM_WORLD, &status);
    } cout.setf(ios::fixed);
    cout << " Process 0 recv " << ibuf << " from process 1, " << rbuf << " from process 2" << endl;
  }
  MPI_Finalize();
}
```

Завершение асинхронного обмена

int MPI_Wait(MPI_Request *request, MPI_Status *status)

request – идентификатор асинхронного приема или передачи;

OUT *status* – параметры сообщения.

Ожидание завершения асинхронных процедур *MPI_Isend* или *MPI_Irecv*, ассоциированных с идентификатором *request*. В случае приема, атрибуты и длину полученного сообщения можно определить с помощью параметра *status*.

int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses)

requests – массив идентификаторов асинхронного приема или передачи;

OUT *statuses* – параметры сообщений.

Выполнение процесса блокируется до тех пор, пока все операции обмена, ассоциированные с указанными идентификаторами, не будут завершены. Если во время одной или нескольких операций обмена возникли ошибки, то поле ошибки в элементах массива *statuses* будет установлено в соответствующее значение.

Неблокирующий кольцевой обмен

```
int main(int argc, char **argv)                    {
   int rank, size, prev, next;              int buf[2];
   MPI_Request reqs[4];    MPI_Status stats[4];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   prev = rank – 1;                           next = rank + 1;
   if(rank==0) prev = size – 1;
   if(rank==size – 1) next = 0;
   MPI_Irecv(&buf[0], 1, MPI_INT, prev, 5, MPI_COMM_WORLD, &reqs[0]);
   MPI_Irecv(&buf[1], 1, MPI_INT, next, 6, MPI_COMM_WORLD, &reqs[1]);
   MPI_Isend(&rank, 1, MPI_INT, prev, 6, MPI_COMM_WORLD, &reqs[2]);
   MPI_Isend(&rank, 1, MPI_INT, next, 5, MPI_COMM_WORLD, &reqs[3]);
   MPI_Waitall(4, reqs, stats);
   cout << " process " << rank << " prev = " << buf[0] << " next = " << buf[1] << endl;
   MPI_Finalize();                                 }
```

Все процессы обмениваются сообщениями с соседями в соответствии с топологией кольца при помощи неблокирующих операций.

Если операции блокирующие – deadlock.

```
process 2 prev = 1 next=3
process 0 prev = 3 next=1
process 1 prev = 0 next=2
process 3 prev = 2 next=0
```

Завершение асинхронного обмена

int MPI_Waitany(int count, MPI_Request *requests, int *index, MPI_Status *status)

OUT *index* – номер завершенной операции обмена;

OUT *status* – параметры сообщений.

Выполнение процесса блокируется, пока какая-либо операция обмена, ассоциированная с указанными идентификаторами, не будет завершена. Если операций несколько, то случайно выбирается одна из них. Параметр *index* содержит номер элемента в массиве *requests*, содержащего идентификатор завершенной операции.

int MPI_Waitsome(int incount, MPI_Request *requests, int *outcount, int *indexes,
MPI_Status *statuses)

incount – число идентификаторов;

OUT *outcount* – число идентификаторов завершившихся операций обмена;

OUT *indexes* – массив номеров завершившихся операции обмена;

OUT *statuses* – параметры завершившихся сообщений.

Выполнение процесса блокируется до тех пор, пока по крайней мере одна из операций обмена, ассоциированных с указанными идентификаторами, не будет завершена. Параметр *outcount* содержит число завершенных операций, а первые *outcount* элементов массива *indexes* содержат номера элементов массива *requests* с их идентификаторами. Первые *outcount* элементов массива *statuses* содержат параметры завершенных операций.

Пример master-slave

Все процессы общаются с одним master.

Все процессы кроме 0-го на каждой итерации работают при помощи slave с локальной частью массива *a*.

Процесс 0 инициализирует неблокирующие приемы, после прихода сообщения инициализирует обработку master, далее выставляет неблокирующие приемы.

Процесс 0 обрабатывает готовые порции данных.

```
void slave(double *a, int n)          { /* обработка локальной части массива a */          }
void master(double *a, int n)         { /* обработка массива a */                          }
int main(int argc, char **argv)       {
   int rank, size, num, k, i, indices[MAXPROC], source;
   MPI_Request req[MAXPROC];   MPI_Status statuses[MAXPROC]; double a[N][MAXPROC];
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Пример master-slave

```
if (rank != 0)
      while(1) {    slave((double*)a, N);
        MPI_Send(a, N, MPI_DOUBLE, 0, 5, MPI_COMM_WORLD); }
      else{
        for (i = 0; i<size-1; i++)
          MPI_Irecv(&a[0][i], N, MPI_DOUBLE, i, 5, MPI_COMM_WORLD, &req[i]);
        while(1){
          MPI_Waitsome(size-1, req, &num, indices, statuses);
          for (i = 0; i< num; i++){
            source = statuses[i].MPI_SOURCE;      master(&a[0][source], N);
            MPI_Irecv(&a[0][source], N, MPI_DOUBLE, source, 5, MPI_COMM_WORLD,
                                                                  &req[source]);
          } } }
   MPI_Finalize();
}
```

Если slave работает дольше master – распараллеливание эффективно.

Еще завершение обмена

int MPI_Testall(int count, MPI_Request *requests, int *flag, MPI_Status *statuses)

requests – массив идентификаторов асинхронного приема или передачи
OUT *flag* – признак завершенности операций обмена
OUT *statuses* – параметры сообщений

В параметре *flag* возвращает значение 1, если все операции, ассоциированные с указанными идентификаторами, завершены (с указанием параметров сообщений в массиве *statuses*). В противном случае возвращается 0, а элементы массива *statuses* неопределены.

int MPI_Testany(int count, MPI_Request *requests, int *index, int *flag, MPI_Status *status)

OUT *index* – номер завершенной операции обмена
OUT *status* – параметры сообщения

Если к моменту вызова подпрограммы хотя бы одна из операций обмена завершилась, то в параметре *flag* возвращается значение *1*, *index* содержит номер соответствующего элемента в массиве *requests*, а *status* – параметры сообщения.

int MPI_Testsome(int incount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses)

incount – число идентификаторов
OUT *outcount* – число идентификаторов завершившихся операций обмена
OUT *indexes* – массив номеров завершившихся операции обмена
OUT *statuses* – параметры завершившихся операций

Подпрограмма работает, как и *MPI_Waitsome*, но возврат происходит немедленно. Если ни одна из указанных операций не завершилась, то значение *outcount* равно нулю.

Транспонирование матриц

```
int main(int argc, char **argv)
{
  int rank, size, nl, i, j, ii, jj, ir, irr;   double c;
  MPI_Status status;
  MPI_Request req[MAXPROC*2]; // MAXPROC задается вручную (количество процессов)
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  nl = (N – 1)/size + 1;
  double *a = (double *) calloc(size*nl*nl, sizeof(double));
  double *b = (double *) calloc(size*nl*nl, sizeof(double));
  for (i = 0; i < nl; i++)      {
    ii = i + rank*nl;
    if (ii<N)
      for (j = 0; j < N; j ++)
        *(a + j*nl + i) = 100*ii + j;
                                }
```

Пример применения неблокирующих операций при реализации транспонирования матриц.

Каждый процесс определяет *n*1 строк массива *a*.
MPI_Isnd и MPI_Irecv инициализируют обмены.
Каждый процесс транспонирует свою часть a.
MPI_Waitany ждет приема сообщений, пока не будут получены сообщения от всех процессов.

Транспонирование матриц

```
for(ir = 0; ir < size; ir++)                                                    {
    MPI_Irecv(b+ir*nl*nl, nl*nl, MPI_DOUBLE, ir, 1, MPI_COMM_WORLD, &req[ir]);
    MPI_Isend(a+ir*nl*nl, nl*nl, MPI_DOUBLE, ir, 1, MPI_COMM_WORLD, &req[ir+size]);   }
  for(irr = 0; irr < size; irr++)                                               {
    MPI_Waitany(size, req, &ir, &status);
    ir = status.MPI_SOURCE;
    for(i = 0; i < nl; i++)                                                 {
      for(j = i + 1; j < nl; j++)                                   {
        c = *(b + i*nl + j + ir*nl*nl);
        *(b + i*nl + j + ir*nl*nl) = *(b + j*nl + i + ir*nl*nl);
        *(b + j*nl + i + ir*nl*nl) = c;                                 }       }       }
  for(i = 0; i < nl; i++)               {       ii = i + rank*nl;
    if(ii < N)
      for(j = 0; j < N; j++)
        printf("process %d : a[%d][%d] = %lf b[%d][%d] = %lf\n", rank, ii, j, *(a+j*nl+i), ii, j, *(b+j*nl+i));
                                        }
  free(a);          free(b);          MPI_Finalize();
}
```

```
process 2 : a[2][0] = 200.000000 b[2][0] = 2.000000
process 0 : a[0][0] = 0.000000 b[0][0] = 0.000000
process 2 : a[2][1] = 201.000000 b[2][1] = 102.000000
process 2 : a[2][2] = 202.000000 b[2][2] = 202.000000
process 1 : a[1][0] = 100.000000 b[1][0] = 1.000000
process 1 : a[1][1] = 101.000000 b[1][1] = 101.000000
process 1 : a[1][2] = 102.000000 b[1][2] = 201.000000
process 0 : a[0][1] = 1.000000 b[0][1] = 100.000000
process 0 : a[0][2] = 2.000000 b[0][2] = 200.000000
```

Отложенные запросы на взаимодействие

Процедуры позволяют снизить накладные расходы одного процессора при обработке приема/передачи и перемещении информации между процессом и сетевым контроллером. Несколько запросов могут объединяться вместе, чтобы их можно было бы запустить одной командой. Способ приема сообщения не зависит от способа его посылки: сообщение, отправленное с помощью объединения запросов, может быть принято как обычным способом, так и с помощью объединения запросов.

int MPI_Send_init(void *buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm, MPI_Request *request)

Формирование запроса на выполнение пересылки данных. Все параметры такие, как и у *MPI_Isend*, однако в отличие от нее пересылка не начинается до вызова подпрограммы *MPI_Startall*.

int MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_Comm comm, MPI_Request *request)

Формирование запроса на выполнение приема данных. Все параметры такие же, как у подпрограммы *MPI_Irecv*, в отличие от нее реальный прием не начинается до вызова подпрограммы *MPI_Startall*.

MPI_Startall(int count, MPI_Request *requests)

count – число запросов на взаимодействие
OUT *requests* – массив идентификаторов приема/передачи

Запуск отложенных взаимодействий, ассоциированных вызовами подпрограмм *MPI_Send_init* и *MPI_Recv_init* с элементами массива запросов *requests*. Все взаимодействия запускаются в режиме без блокировки, а их завершение можно определить с помощью процедур *MPI_Wait* и *MPI_Test*.

Отложенные запросы на взаимодействие

Схема итерационного обмена по кольцевой топологии при помощи отложенных запросов.
Операции запускаются на каждой итерации последующего цикла.
По завершении цикла отложенные запросы удаляются.

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
prev = rank – 1; next = rank + 1;
if(rank == 0) prev = size – 1;
if(rank == size – 1) next = 0;
MPI_Recv_init(rbuf[0], 1, MPI_FLOAT, prev, 5, MPI_COMM_WORLD, reqs[0]);
MPI_Recv_init(rbuf[1], 1, MPI_FLOAT, next, 6, MPI_COMM_WORLD, reqs[1]);
MPI_Send_init(sbuf[0], 1, MPI_FLOAT, prev, 6, MPI_COMM_WORLD, reqs[2]);
MPI_Send_init(sbuf[1], 1, MPI_FLOAT, next, 5, MPI_COMM_WORLD, reqs[3]);
for (i = ...) {
        sbuf[0] =...;        sbuf[1] =...;
        MPI_Startall(4, reqs);
        ...
        MPI_Waitall(4, reqs, stats);
        ... }
MPI_Request_free(reqs[0]);        MPI_Request_free(reqs[1]);
MPI_Request_free(reqs[2]);        MPI_Request_free(reqs[3]);
```

Позволяют снизить накладные расходы.
При обменах с одинаковыми параметрами (циклы).
Обмен один раз инициализируется и много раз запускается.

Еще раз тупиковые ситуации

Посылающий процесс ждет приема или наоборот?

1. Возникает тупик
2. Может возникнуть тупик

процесс 0:	процесс 1:
MPI_RECV от процесса 1 MPI_SEND процессу 1	MPI_RECV от процесса 0 MPI_SEND процессу 0

процесс 0:	процесс 1:
MPI_SEND процессу 1 MPI_RECV от процесса 1	MPI_SEND процессу 0 MPI_RECV от процесса 0

Разрешение тупиковых ситуаций

1. Порядок прием-передача
2. Неблокирующая передача

процесс 0:	процесс 1:
MPI_SEND процессу 1 MPI_RECV от процесса 1	MPI_RECV от процесса 0 MPI_SEND процессу 0

MPI_Irecv рекомендуется выставлять как можно раньше, чтобы использовать преимущество асинхронности.

Процесс 0:	процесс 1:
MPI_SEND процессу 1 MPI_RECV от процесса 1	MPI_IRECV от процесса 0 MPI_SEND процессу 0 MPI_WAIT

Совмещенные прием/передача сообщений

int MPI_Sendrecv(void *sbuf, int scount, MPI_Datatype stype, int dest,
int stag, void *rbuf, int rcount, MPI_Datatype rtype, int source,
MPI_Datatype rtag, MPI_Comm comm, MPI_Status *status)

sbuf – адрес начала буфера посылки сообщения;
scount – число передаваемых элементов в сообщении;
stype – тип передаваемых элементов;
dest – номер процесса-получателя;
stag – идентификатор посылаемого сообщения;
OUT *rbuf* – адрес начала буфера приема сообщения;
rcount – число принимаемых элементов сообщения;
rtype – тип принимаемых элементов;
source – номер процесса-отправителя;
rtag – идентификатор принимаемого сообщения;
comm – идентификатор группы;
OUT *status* – параметры принятого сообщения.

Буфера приема и посылки обязательно должны быть различными.

MPI_Sendrecv

Данная операция объединяет в едином запросе посылку и прием сообщений. Принимающий и отправляющий процессы могут являться одним и тем же процессом.

Сообщение, отправленное операцией *MPI_Sendrecv*, может быть принято, и также операция *MPI_Sendrecv* может принять сообщение, отправленное обычной операцией *MPI_Send*.

MPI_Sendrecv содержит 12 параметров: первые 5 параметров такие же, как у MPI_Send, остальные 7 параметров такие же как у MPI_Recv.

Один вызов – те же действия, что и блок IF-ELSE с четырьмя вызовами.

Следует учесть:

- и прием, и передача используют один и тот же коммуникатор;
- порядок приема и передачи данных MPI_Sendrecv выбирает автоматически;
- гарантируется, что автоматический выбор не приведет к deadlock;
- MPI_Sendrecv совместима с MPI_Send и MPI_Recv, т.е может "общаться" с ними.

Гарантировано нет тупиковой ситуации

Операции двунаправленного обмена с соседними процессами в кольцевой топологии

```
int main(int argc, char **argv)
{
  int rank, size, prev, next;  int buf[2];
  MPI_Status status1;        MPI_Status status2;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  prev = rank – 1;next = rank + 1;
  if (rank == 0) prev = size – 1;
  if (rank == size – 1) next = 0;
  MPI_Sendrecv(&rank, 1, MPI_INT, prev, 6, &buf[1], 1, MPI_INT, next, 6,
            MPI_COMM_WORLD, &status2);
  MPI_Sendrecv(&rank, 1, MPI_INT, next, 5, &buf[0], 1, MPI_INT, prev, 5,
            MPI_COMM_WORLD, &status1);
  cout << "process " << rank << " prev = " << buf[0] << " next = " << buf[1] << endl;
  MPI_Finalize();
}
```

```
process 3 prev = 2 next = 0
process 2 prev = 1 next = 3
process 0 prev = 3 next = 1
process 1 prev = 0 next = 2
```

Коллективные взаимодействия процессов

В коллективных операциях участвуют все процессы коммуникатора.

Соответствующая процедура д.б. вызвана каждым процессом, м.б. со своим набором параметров.

Возврат из процедуры коллективного взаимодействия может произойти в тот момент, когда участие процесса в данной операции уже закончено.

Возврат означает, что разрешен свободный доступ к буферу приема или посылки.

Асинхронных коллективных операций нет.

В коллективных операциях можно использовать те же коммуникаторы, что и для операций точка-точка. Коллективные операции не пересекутся с сообщениями индивидуального взаимодействия процессов.

Коллективные операции не дают синхронизации (кроме MPI_Barrier).

В коллективных операциях не используются теги, т.о. они упорядочены согласно их появлению в тексте программы.

MPI_Bcast и MPI_Reduce см. лекция 12.

Пример сумма

```
#define N 5
using namespace std;
int main(int argc, char **argv)
{
  int rank, i, j, s, size, sum;
  int A [ N ];
  MPI_Status status; s = 0;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for(i = 0; i < N; i++)  {
    A [ i ] = i + 1 ; s += A[i];
  }
  MPI_Reduce(&s, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
   if ( rank == 0) cout << endl << sum << endl;
MPI_Finalize();
}
```

```
mpirun -np 5 ./a
```

1 + 2 + 3 + 4 + 5

на 5 процессах

sum = 75

О времени OpenMP

Измеряем время выполнения внутреннего цикла for в OpenMP.
omp_get_wtime – возвращает затраченное фактическое время (в секундах), прошедшее с начала заданной точки отсчета, т.е. фактическое время выполнения цикла for.
Системный вызов clock() используется для оценки времени, затраченного центральным процессором на выполнение всей программы, т. е. прежде чем получить итоговый результат суммируются все временные интервалы с учетом потоков.

```
threads: 1 time on clock(): 0.015229 time on wall: 0.0152249
threads: 2 time on clock(): 0.014221 time on wall: 0.00618792
threads: 3 time on clock(): 0.014541 time on wall: 0.00444412
threads: 4 time on clock(): 0.014666 time on wall: 0.00440478
threads: 5 time on clock(): 0.01594 time on wall: 0.00359988
threads: 6 time on clock(): 0.015069 time on wall: 0.00303698
threads: 7 time on clock(): 0.016365 time on wall: 0.00258303
threads: 8 time on clock(): 0.01678 time on wall: 0.00237703
```

time on clock – время центрального процессора примерно одинаково (не считаем время, затраченное на создание потоков и контекстного переключателя);
time on wall – фактическое время постоянно уменьшается при увеличении количества потоков.

https://www.ibm.com/developerworks/ru/library/au-aix-openmp-framework/

О времени MPI

time
OpenMP

```
real    0m0.038s
user    0m0.020s
sys     0m0.020s
```

```
#define NTIMES 100
int main(int argc, char *argv[])
{
int myid, numprocs, i;
double time_start, time_finish ;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);
   time_start = MPI_Wtime();
   for (i = 0; i<NTIMES; i++)
   time_finish = MPI_Wtime();
        cout << "    Time " <<  time_finish - time_start << endl;
        MPI_Finalize();
return 0;
}
```

```
~ $ time mpirun -np 4 ./aa
    Time 4.29153e-06
    Time 4.76837e-06
    Time 4.29153e-06
    Time 4.05312e-06

real    0m0.221s
user    0m0.220s
sys     0m0.152s
```