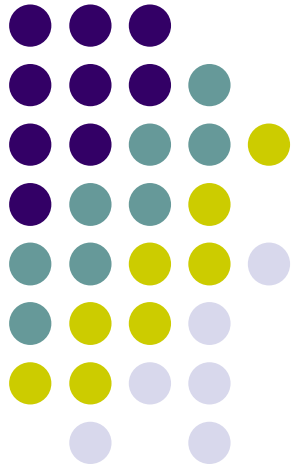
Распределенные системы

Практическая часть 2

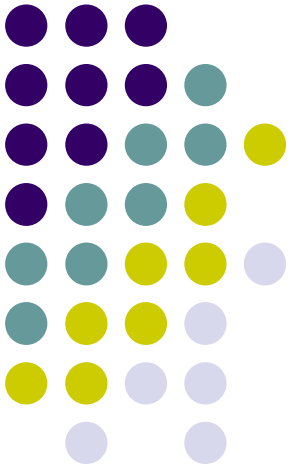
Технологии распределенного программирования

к.т.н. Приходько Т.А.



Библиотека MPI

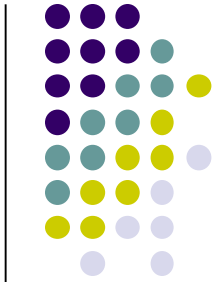
Message Passing Interface



История MPI

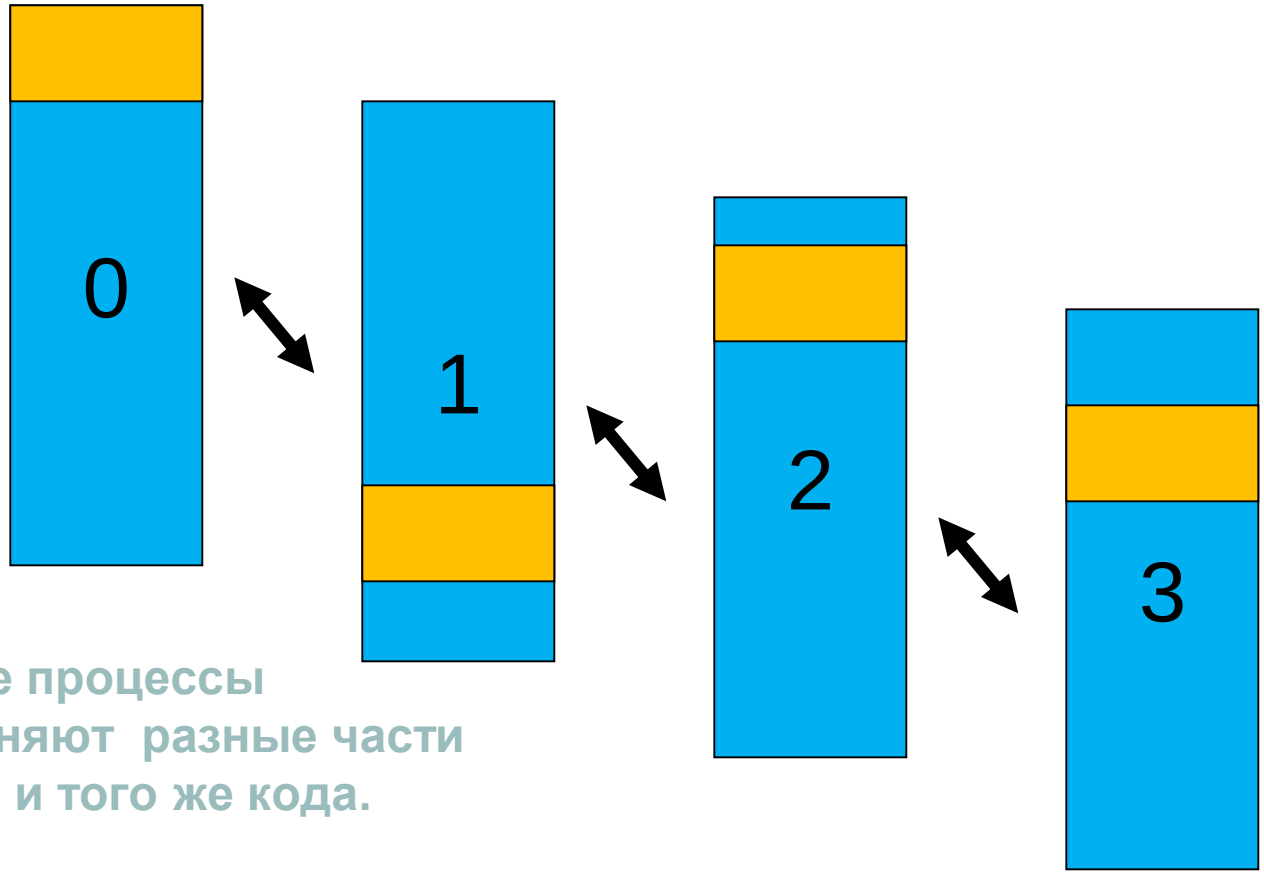
Стандарт MPI 1.0 1995 год, MPI 2.0 1998 год. Определяет API (варианты для Си, C++, Fortran, Java).

«Комплект поставки» MPI



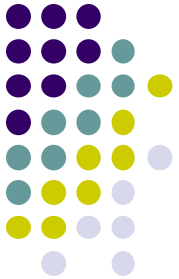
- Библиотека.
- Средства компиляции и запуска приложения.

SIMD (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**.

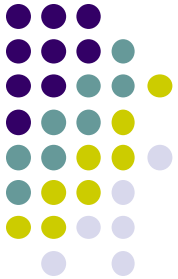
```
mpirun –np 4 mpihello
```

Структура программы на MPI

Структура *параллельной программы*, разработанная с использованием *MPI*, должна иметь следующий вид:

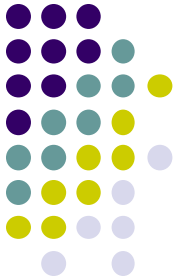
```
#include "mpi.h"
int main(int argc, char *argv[]) {
<программный код без использования функций MPI>
MPI_Init(&argc, &argv);
<программный код с использованием функций MPI>
MPI_Finalize();
<программный код без использования функций MPI>
return 0;
}
```

MPI “Hello, World” на C



```
#include <stdio.h>
#include <mpi.h>
main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);
  printf("Hello, World!\n");
  MPI_Finalize();
}
```

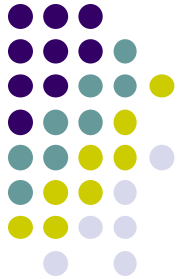
Тоже простая MPI-программа на C и JAVA



```c
#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;
}
```

```java
import mpi.*;
public class MPIAPP {

public static void main(String[] args) throws Exception{

MPI.Init(args);
int rank = MPI.COMM_WORLD.Rank();
int size = MPI.COMM_WORLD.Size();
System.out.println("Hi from <"+rank+">");
MPI.Finalize();
    }}
```

Функции определения ранга и числа процессов



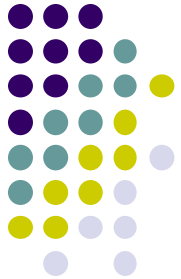
int MPI_Comm_size (MPI_Comm comm, int* size)

comm - коммуникатор
size – число процессов

int MPI_Comm_rank(MPI_Comm comm, int* rank)

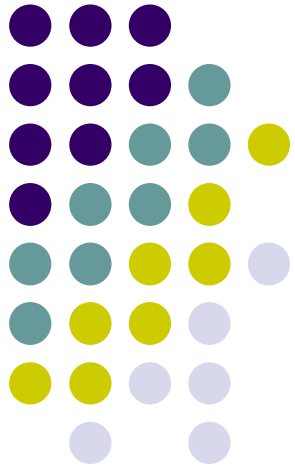
comm – коммуникатор
rank – ранг процесса

Общая схема распараллеливания:

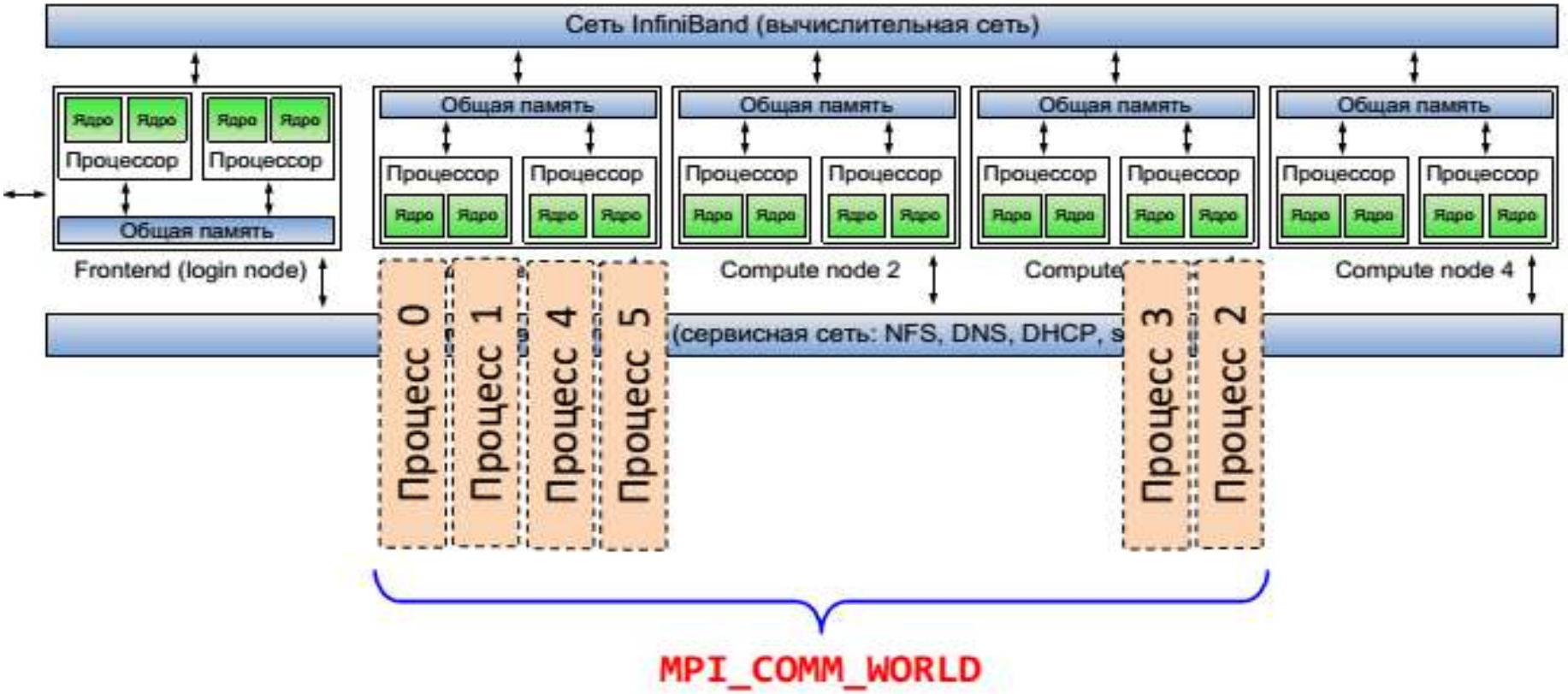


```
if (rank == 0) {/* операции, выполняемые 0-ым процессом */}
if (rank == K) {/* операции, выполняемые K-ым процессом */}

for (i = 0; i < N; i++) {
        if (i == rank) {
/* операции i-й итерации для выполнения процессом rank */
        }
}
```

Двухточечные взаимодействия



Коммуникаторы MPI



- **Коммуникатор (communicator)** – множество процессов, образующих логическую область для выполнения коллективных операций (обменов информацией и др.)
  - В рамках коммуникатора процессы имеют номера: 0, 1, …, n – 1

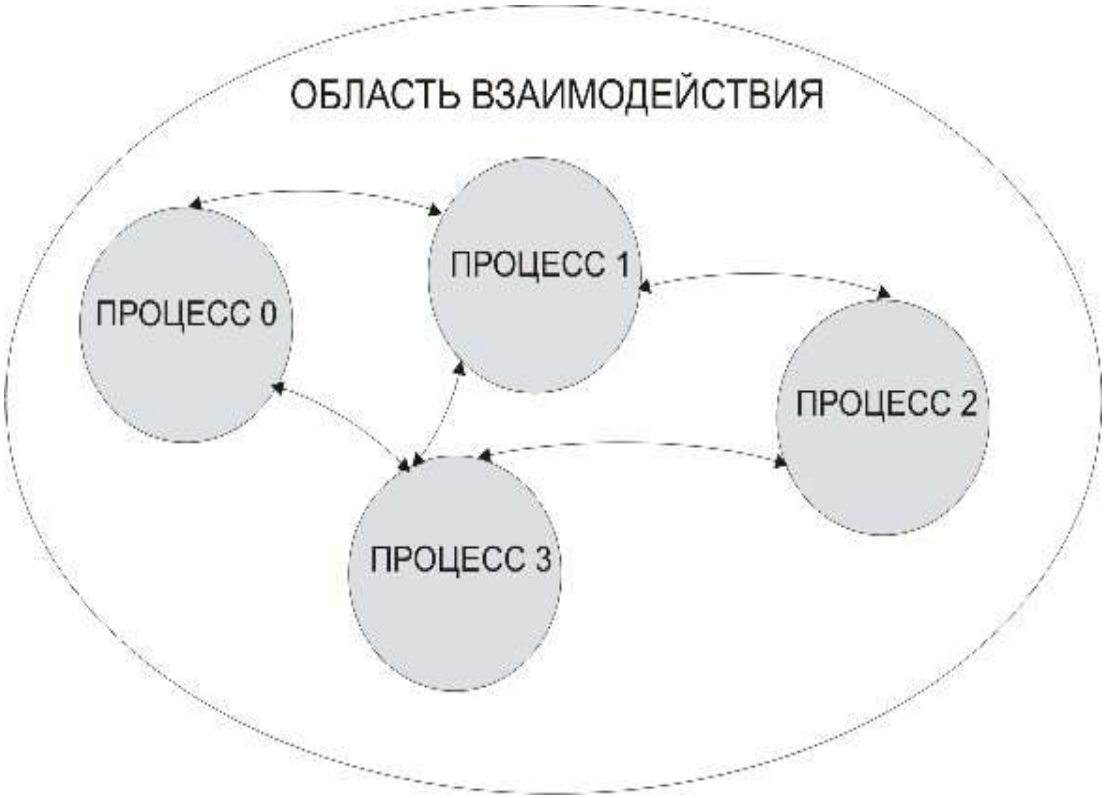
Двухточечные обмены

Двухточечный (point-to-point, p2p) обмен

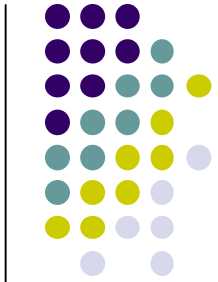
В двухточечном обмене участвуют только два процесса, процесс-отправитель и процесс-получатель (источник сообщения и адресат). Двухточечные обмены используются для организации локальных и неструктурированных коммуникаций.

Двухточечные обмены

Двухточечный обмен возможен только между процессами, принадлежащими одной области взаимодействия (одному коммуникатору).



Двухточечные обмены



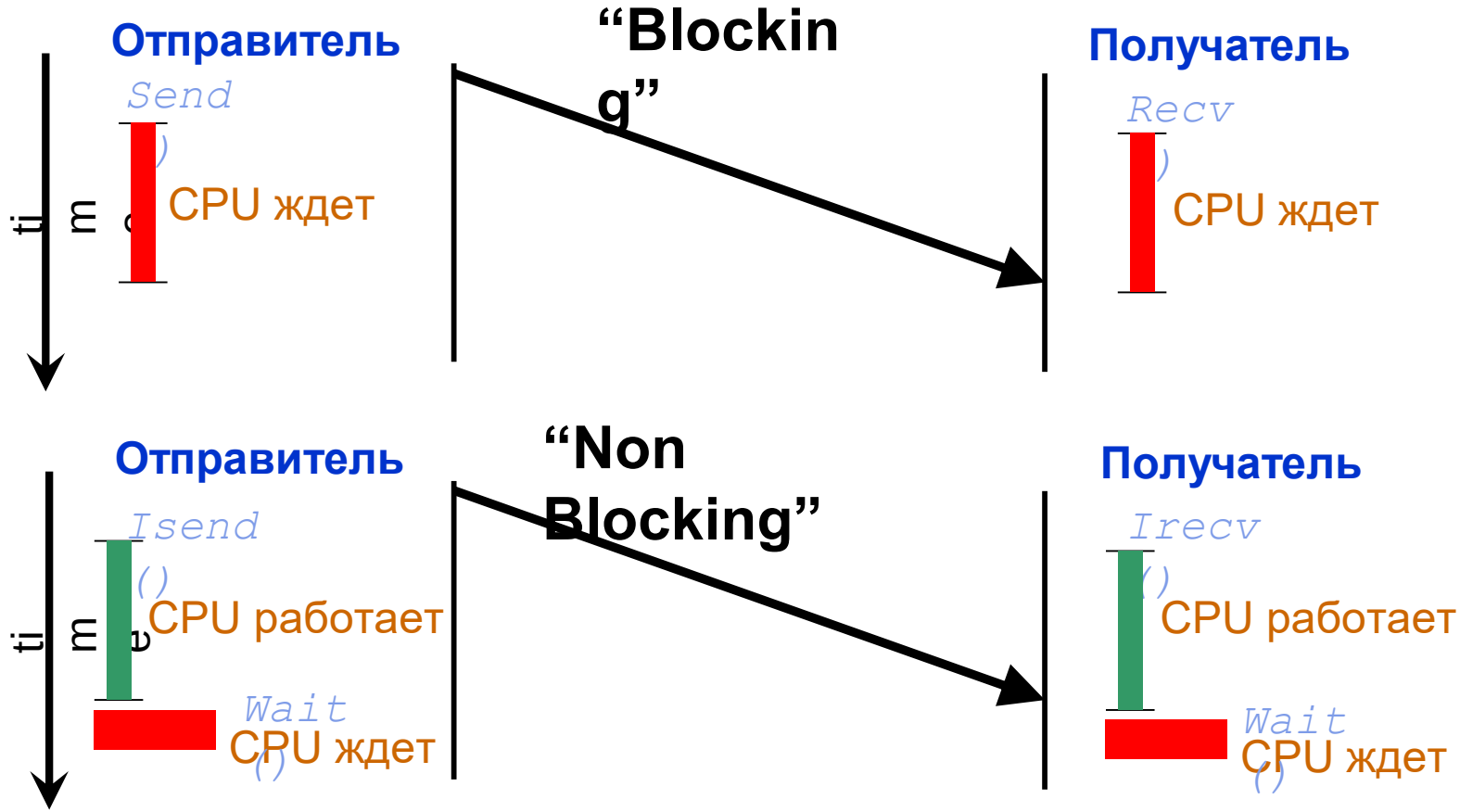
Несколько разновидностей двухточечного обмена

- ***блокирующие*** прием/передача, которые приостанавливают выполнение процесса на время приема или передачи сообщения (4 разновидности);
- ***неблокирующие*** прием/передача, при которых выполнение процесса продолжается в фоновом режиме, а программа в нужный момент может запросить подтверждение завершения приема сообщения (4 разновидности).

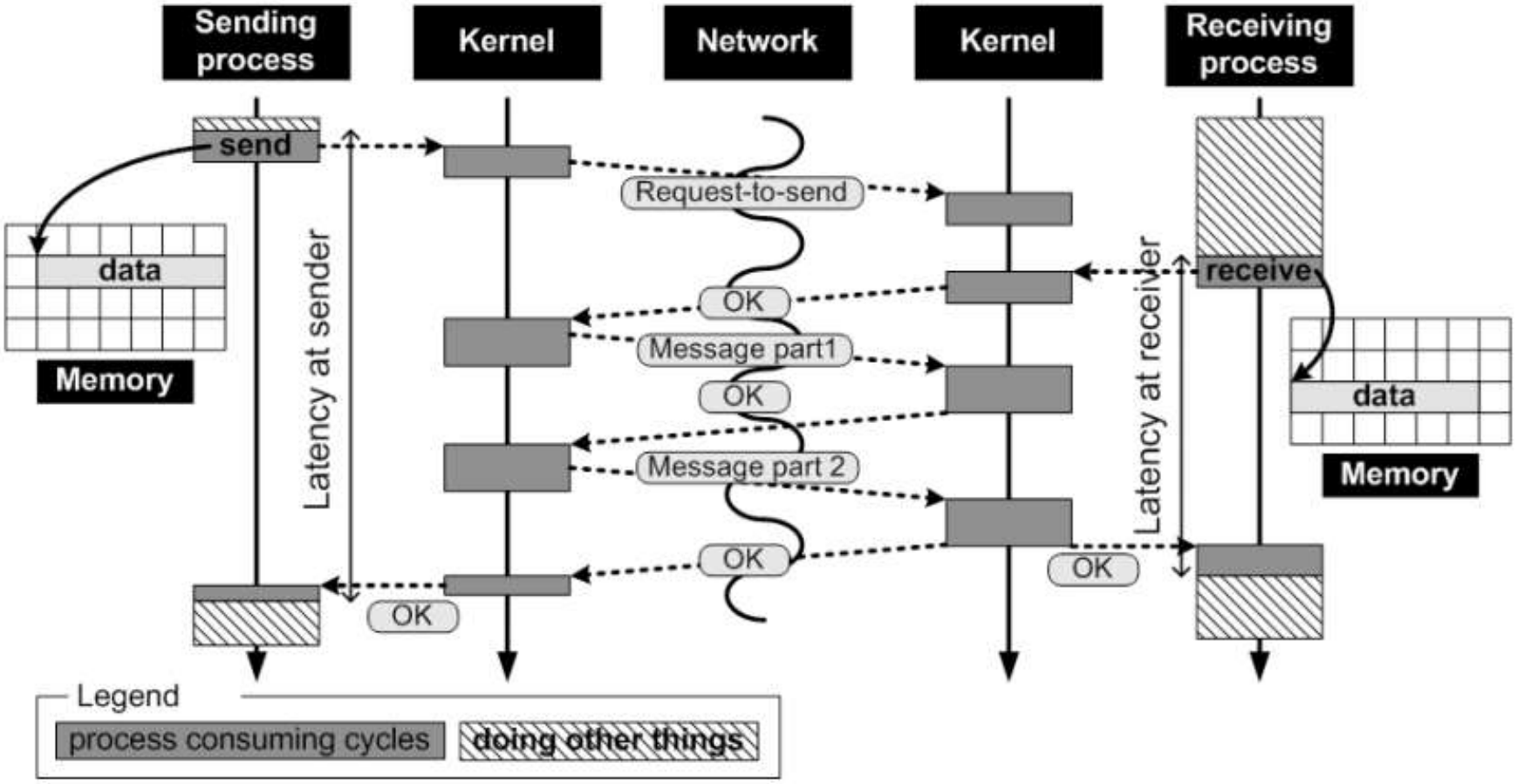
Их синонимами можно считать

- ***синхронный*** обмен, который сопровождается уведомлением об окончании приема сообщения;
- ***асинхронный*** обмен, который таким уведомлением не сопровождается.

Двухточечные обмены

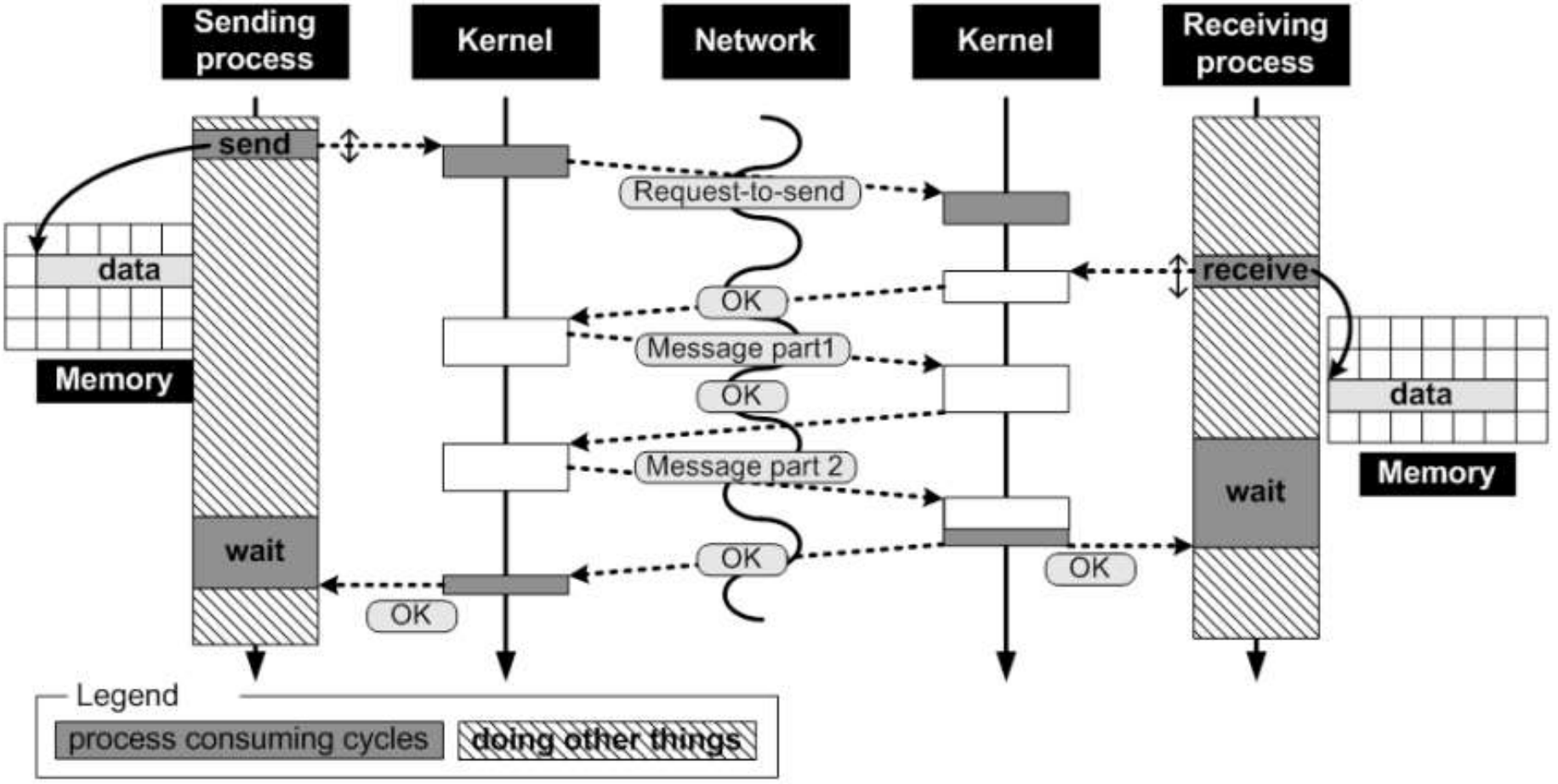


Двухточечные обмены - блокирующий



Blocking	MPI_Send	MPI_Bsend	MPI_Ssend	MPI_Rsend	MPI_Recv

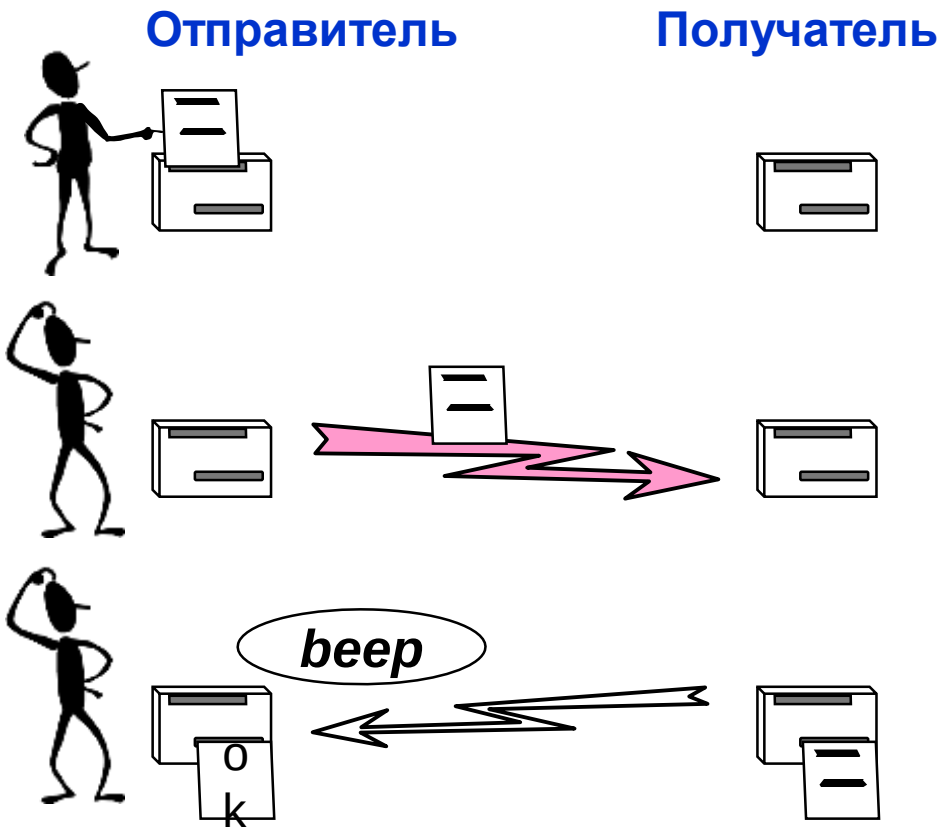
Двухточечные обмены - неблокирующий



Non-blocking	MPI_Isend	MPI_Ibsend	MPI_Issend	MPI_Irsend	MPI_Irecv

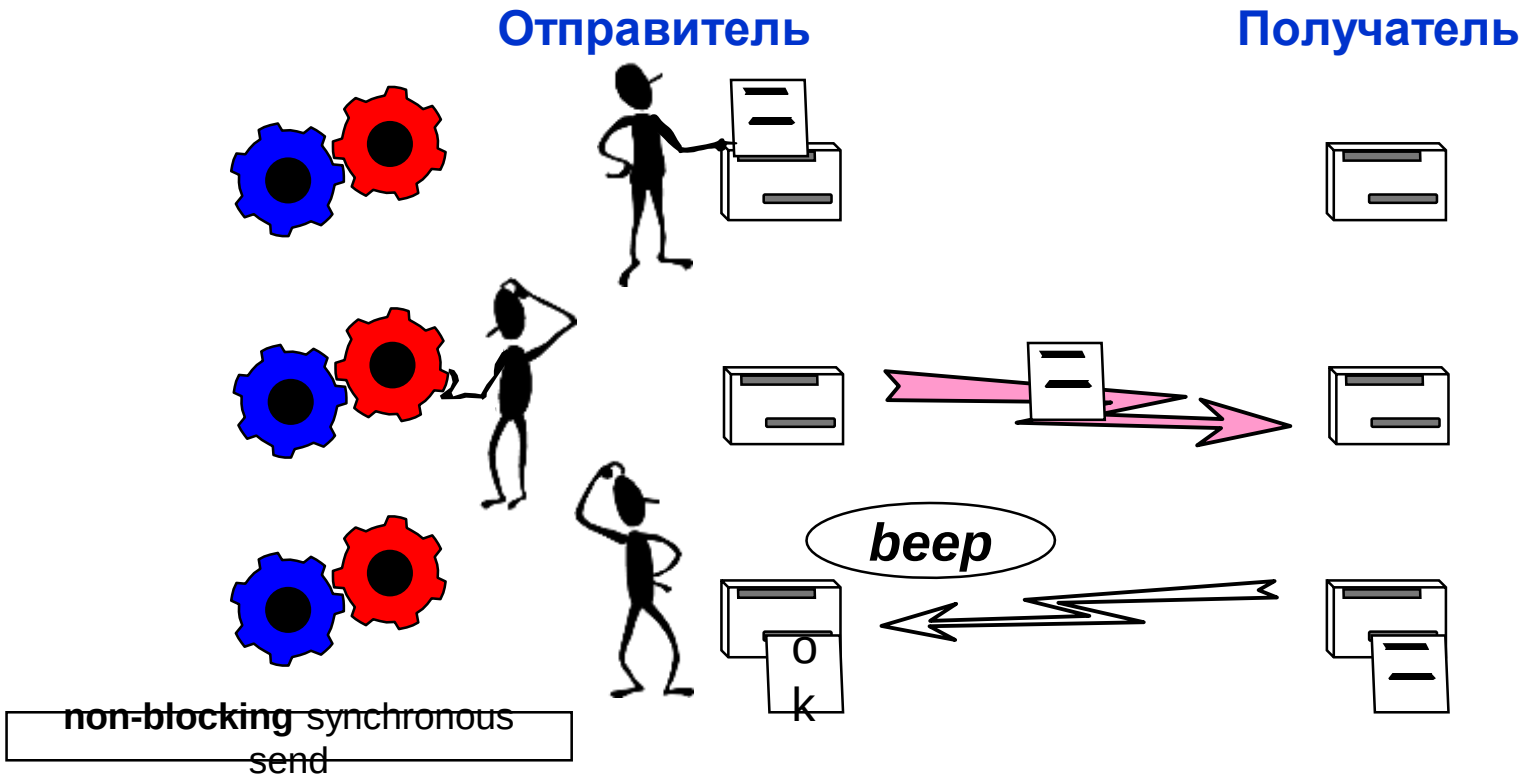
MPI – блокирующая посылка (синхронная)

- Процессор-отправитель ожидает информацию о том, когда получатель примет сообщение.
- Пример, факс получатель присылает тег завершения приема.



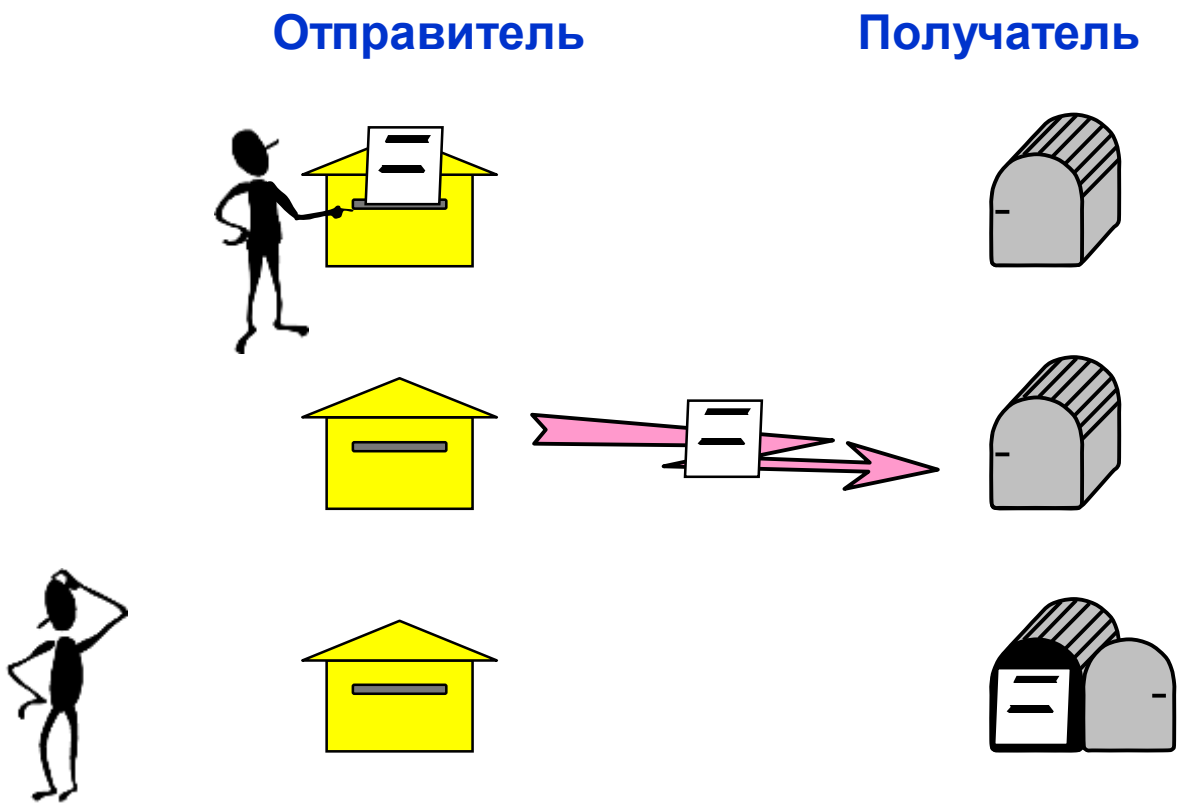
MPI–посылки без блокировки (асинхронные)

- Неблокирующие операции немедленно возвращают управление программе. Программа выполняет следующие действия.
- Для того, что бы спустя некоторое время убедиться, что неблокирующая функция передачи данных выполнена полностью, нужно вызвать функцию MPI_Test или MPI_Wait.

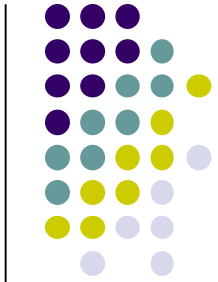


MPI – Буферизированная посылка или Несинхронная посылка

- Процессор-отправитель знает только когда сообщение ушло.



Двухточечные обмены



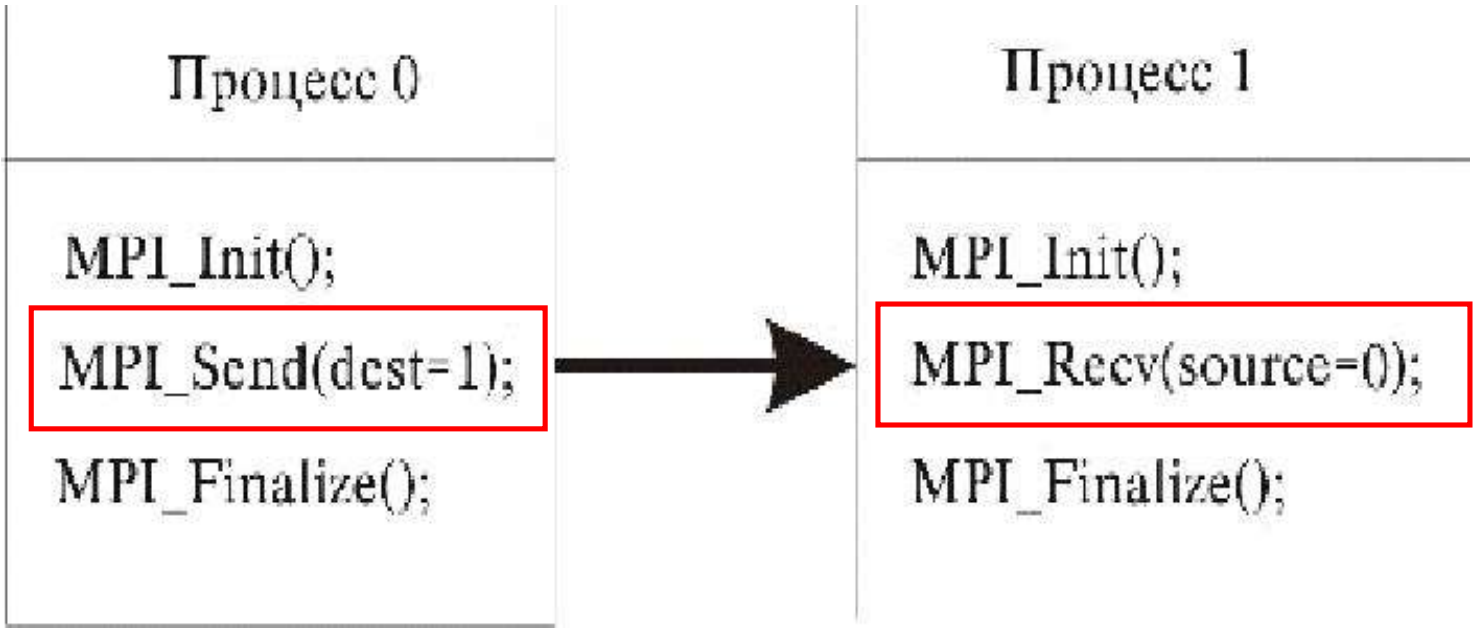
Правильно организованный двухточечный обмен сообщениями должен исключать возможность блокировки или некорректной работы параллельной MPI-программы.

Примеры ошибок в организации двухточечных обменов:

- выполняется передача сообщения, но не выполняется его прием;
- процесс-источник и процесс-получатель одновременно пытаются выполнить блокирующие передачу или прием сообщения.

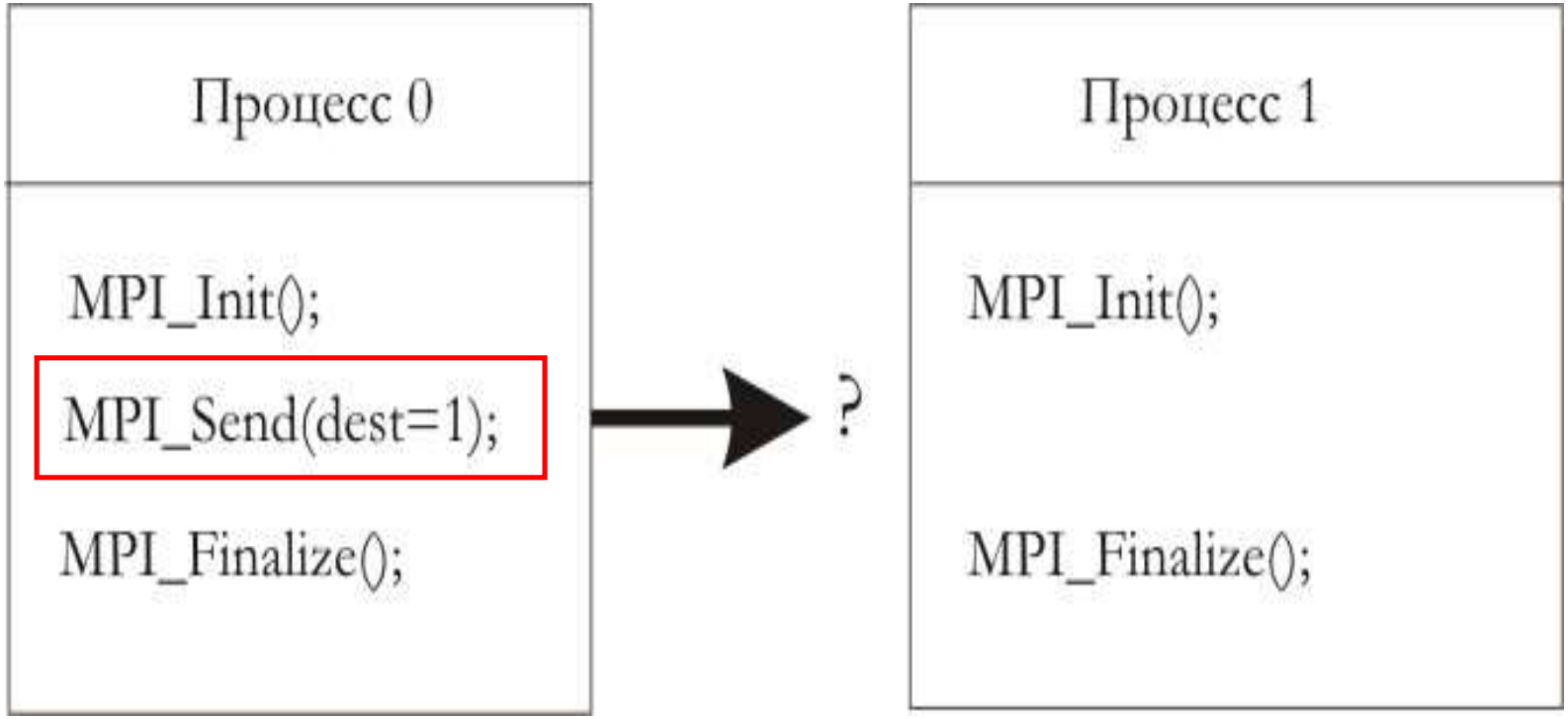
Двухточечные обмены

Правильно



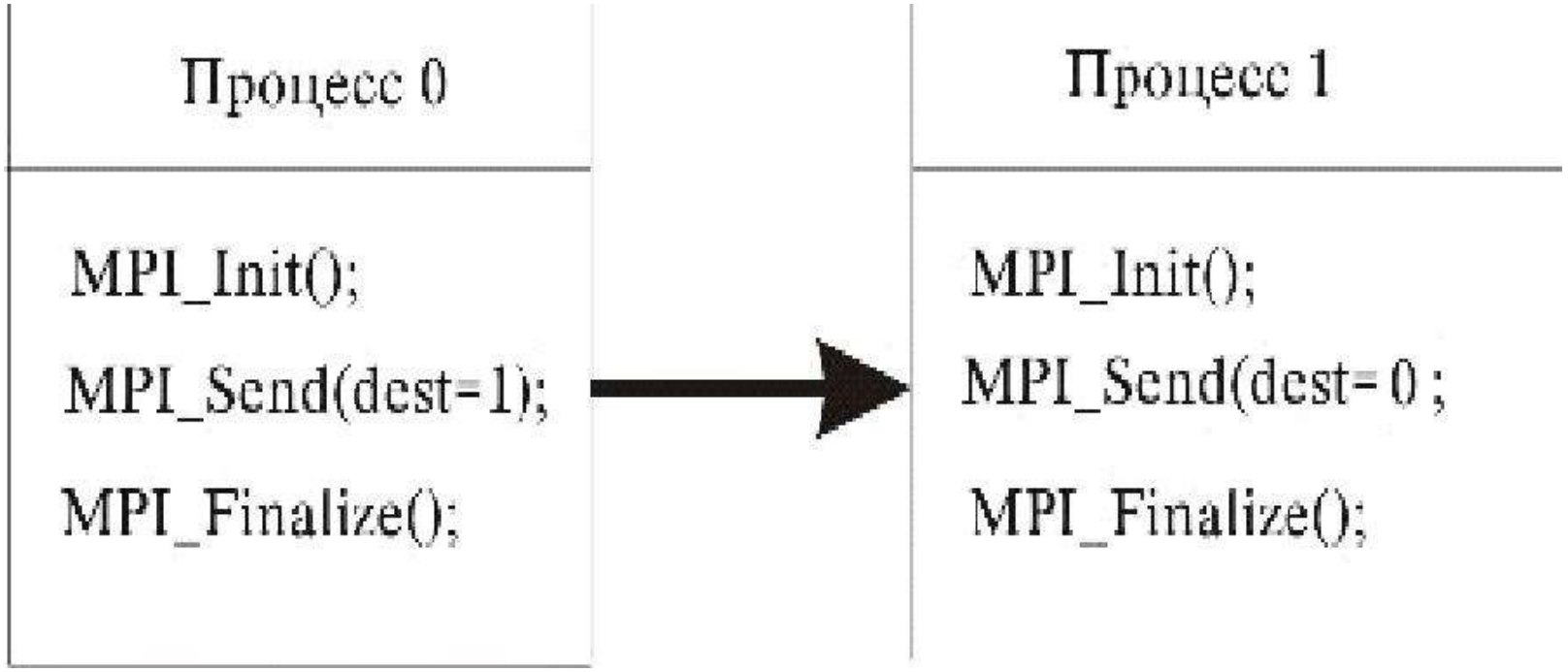
Двухточечные обмены

Неправильно

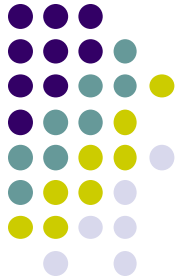


Двухточечные обмены

Неправильно



Двухточечные обмены

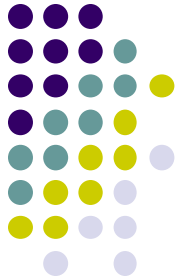


- В MPI приняты следующие соглашения об именах подпрограмм двухточечного обмена:

  MPI_[I][R, S, B]Send

- здесь префикс [I] (Immediate) обозначает неблокирующий режим. Префиксы [R, S, B] обозначают режимы обмена:
  - ***по готовности, синхронный и буферизованный.***
- Отсутствие префикса обозначает подпрограмму стандартного (блокирующего) обмена. Имеется 8 разновидностей операций передачи сообщений.

Двухточечные обмены

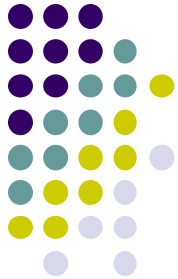


- Для подпрограмм приема:

  MPI_[I]Recv

- то есть всего 2 разновидности приема (блокирующий и неблокирующий).
- Подпрограмма MPI_Irsend, например, выполняет передачу «по готовности» в неблокирующем режиме, MPI_Bsend буферизованную передачу с блокировкой, а MPI_Recv выполняет блокирующий прием сообщений. Подпрограмма приема любого типа может принять сообщения от любой подпрограммы передачи.

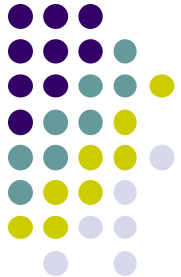
Двухточечные обмены



Список коммуникационных функций типа точка-точка		
Режимы выполнения	**С блокировкой**	**Без блокировки**
Стандартная посылка	MPI_Send	MPI_Isend
Синхронная посылка	MPI_Ssend	MPI_Issend
Буферизованная посылка	MPI_Bsend	MPI_Ibsend
Согласованная посылка (по готовности)	MPI_Rsend	MPI_Irsend
Прием информации	MPI_Recv	MPI_Irecv

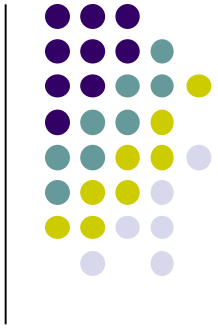
Blocking	MPI_Send	MPI_Bsend	MPI_Ssend	MPI_Rsend	MPI_Recv
Non-blocking	MPI_Isend	MPI_Ibsend	MPI_Issend	MPI_Irsend	MPI_Irecv

Двухточечные обмены



- ***Блокирующие вызовы*** отправки и получения приостанавливают исполнение программы до момента, когда данные будут отправлены (скопированы из буфера отправки), но они не обязаны быть получены получающей задачей. Содержимое буфера отправки теперь может быть безопасно модифицировано без воздействия на отправленное сообщение. Завершение блокирующего обмена подразумевает, что данные в буфере получения правильные.
- ***Неблокирующие вызовы*** возвращаются немедленно после инициации коммуникации. Программист не знает скопированы ли отправляемые данные из буфера отправки, или являются ли передаваемые данные прибывшими к получателю. Таким образом, перед очередным использованием *буфера сообщения* программист должен проверить его статус (команды Wait или Test).

Пример простейшей блокирующей 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();
}
```

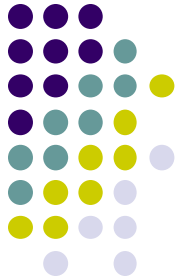
buf - адрес начала буфера посылаемых данных

count - число пересылаемых объектов типа, соответствующего datatype

dest - номер процесса-приемника

tag - уникальный тэг, идентифицир-й сообщение

Функции обменов точка-точка



Стандартная блокирующая передача на С

```
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
datatype - MPI-тип принимаемых данных
dest - номер процесса-приемника. Ранг процесса-получателя сообщения (целое число от 0 до n – 1, где n число процессов в области взаимодействия);
tag – уникальное число от 0 до 32767, идентифицирующий сообщение. Позволяет различать сообщения, приходящие от одного процесса. Теги могут использоваться и для соблюдения определенного порядка приема сообщений.
comm - коммуникатор

Функции обменов точка-точка

Стандартная блокирующая передача

JAVA

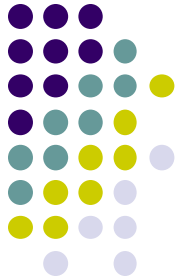
```
C: int MPI_Send(buf, count, datatype, dest, tag, comm )
```

JAVA:

```
public void Send(java.lang.Object buf, int offset,
int count, Datatypepublic void Send(java.lang.Object buf,
int offset, int count, Datatype datatype, int dest, int tag)
throws MPIException
```

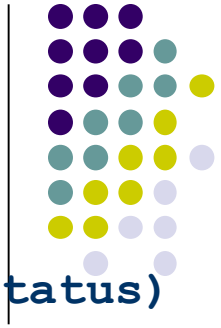
buf - адрес начала буфера посылаемых данных(одномерный массив)
offset – сдвиг в пересылаемом буфере,
count - число пересылаемых данных,
datatype - MPI-тип каждого из значений в отправляемом буфере
dest - номер процесса-приемника. Ранг процесса-получателя сообщения (целое число от 0 до n – 1, где n число процессов в области взаимодействия);
tag - уникальный тэг, идентифицирующий сообщение

Функции обменов точка-точка



- При стандартной **блокирующей** передаче после завершения вызова (после возврата из функции/процедуры передачи) можно использовать любые переменные, использовавшиеся в списке параметров. Такое использование не повлияет на корректность обмена.
- Дальнейшая «судьба» сообщения зависит от реализации MPI. Сообщение может быть сразу передано процессу-получателю или может быть скопировано в буфер передачи.

Функции обменов точка-точка



Стандартный блокирующий прием

```
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 - ранг процесса-отправителя сообщения (целое число от 0 до n – 1, где n – число процессов в области взаимодействия);
tag - уникальный тэг, идентифицирующий сообщение
datatype - MPI-тип принимаемых данных
comm - коммуникатор
status - статус завершения

Функции обменов точка-точка

Стандартный блокирующий прием

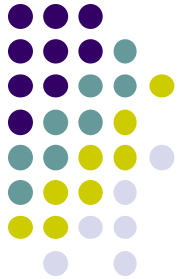
JAVA

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

public Status Recv(java.lang.Object buf, int offset, int count, Datatype datatype, int source, int tag) throws MPIException

buf - адрес начала буфера посылаемых данных(одномерный массив)
offset – сдвиг в получаемом буфере,
count - число получаемых данных,
datatype - MPI-тип каждого из значений в отправляемом буфере
source - номер процесса-отправителя (ранг процесса)
tag - уникальный тэг - сообщение
status –возвращает статус завершения

Функции обменов точка-точка

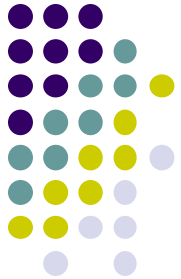


Содержимое статуса завершения (C)

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

count – число полученных элементов
MPI_SOURCE – ранг процесса-передатчика данных
MPI_TAG – тэг сообщения
MPI_ERROR – код ошибки

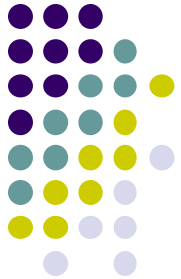
Функции обменов точка-точка



- Значение параметра ***`count`*** может оказаться больше, чем количество элементов в принятом сообщении. В этом случае после выполнения приёма в буфере изменится значение только тех элементов, которые соответствуют элементам фактически принятого сообщения.

- Для функции **`MPI_Recv`** ***гарантируется, что после завершения вызова сообщение принято и размещено в буфере приема.***

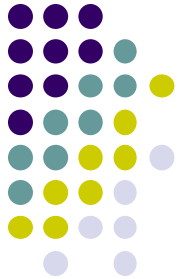
Функции обменов точка-точка



Специфические коды завершения:

- ***MPI_ERR_COMM*** - неправильно указан коммуникатор. Часто возникает при использовании «пустого» коммуникатора;
- ***MPI_ERR_COUNT*** - неправильное значение аргумента ***count*** (количество пересылаемых значений);
- ***MPI_ERR_TYPE*** - неправильное значение аргумента, задающего тип данных;
- ***MPI_ERR_TAG*** - неправильно указан тег сообщения;
- ***MPI_ERR_RANK*** - неправильно указан ранг источника или адресата сообщения;
- ***MPI_ERR_ARG*** - неправильный аргумент, ошибочное задание которого не попадает ни в один класс ошибок;
- ***MPI_ERR_REQUEST*** - неправильный запрос на выполнение операции.

Функции обменов точка-точка



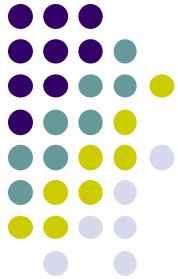
«Джокеры» для приема сообщений

В качестве ранга источника сообщения и в качестве тега сообщения можно использовать «джокеры» :

- ***MPI_ANY_SOURCE*** - любой источник;
- ***MPI_ANY_TAG*** - любой тег.

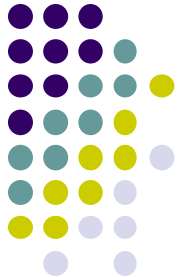
При использовании «джокеров» есть опасность приема сообщения, не предназначенного данному процессу.

Функции обменов точка-точка



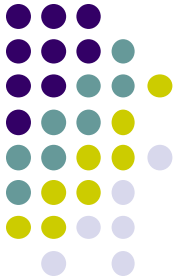
- Подпрограмма **`MPI_Recv` может принимать сообщения, отправленные *в любом режиме*.**
- Прием может выполняться от произвольного процесса, а в операции передачи должен быть указан вполне определенный адрес. Приемник может использовать «джокеры» для источника и для тега. Процесс может отправить сообщение и самому себе, но следует учитывать, что использование в этом случае блокирующих операций может привести к «тупику».

Пример использования операции блокирующего двухточечного обмена на C



```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
        int ProcNum, ProcRank, tmp;
        MPI_Status status;

        MPI_Init ( &argc, &argv );
        MPI_Comm_size (MPI_COMM_WORLD, &ProcNum);
        MPI_Comm_rank (MPI_COMM_WORLD, &ProcRank);
        if (ProcRank == 0){
            printf("Send message from process %i \n", ProcRank);
            for(int i = 1; i < ProcNum; i++)
                    {
MPI_Recv(&tmp,1, MPI_INT, MPI_ANY_SOURCE,0,MPI_COMM_WORLD, &status);
                    printf("Message recived from process %d \n", tmp);
                    }

        }else{ MPI_Send(&ProcRank,1,MPI_INT,0,0,MPI_COMM_WORLD);}
        MPI_Finalize();
        return 0;
}
```

Пример использования операции блокирующего двухточечного обмена на JAVA



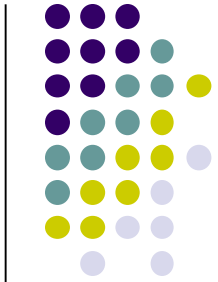
```
package mpiapp;
import java.util.*;
import mpi.*;

public class MPIAPP {

    public static void main(String[] args) throws Exception{

MPI.Init(args);

int myrank = MPI.COMM_WORLD.Rank();
int size=MPI.COMM_WORLD.Size();
       if(myrank == 0) {
          char[] message ="Hello from boss!".toCharArray();
          for (int i=1;i<size;i++)
          MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR, i, 1) ;

          char[] backmessage = new char[15];
          for (int i=1;i<size;i++)
          {MPI.COMM_WORLD.Recv(backmessage, 0, 15, MPI.CHAR, i, 2) ;
          System.out.println("received: from rank "+i+ " "+new String(backmessage)) ;}
       }
```

Пример использования операции блокирующего двухточечного обмена на JAVA (окончание)

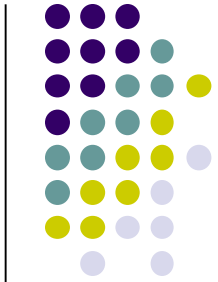
```
else {
        char[] message = new char[20] ;
        MPI.COMM_WORLD.Recv(message, 0, 20, MPI.CHAR, 0, 1) ;
        System.out.println("received: from rank "+myrank+ " "+new String(message)) ;
        char [] backmessage="Hello, master".toCharArray();
      MPI.COMM_WORLD.Send(backmessage, 0, backmessage.length, MPI.CHAR, 0, 2) ;
      }
MPI.Finalize();
   }
}
```

```
run:
MPJ Express (0.44) is started in the multicore configuration
received: from rank 1 Hello from boss!
received: from rank 4 Hello from boss!
received: from rank 3 Hello from boss!
received: from rank 2 Hello from boss!
received: from rank 1 Hello, master
received: from rank 2 Hello, master
received: from rank 3 Hello, master
received: from rank 4 Hello, master
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 2 секунды)
```

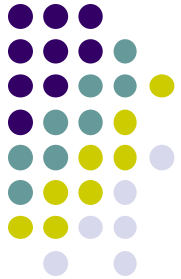
Задание 1



- В исходном тексте программы на языке С (след. слайд) предусмотрена некая схема обмена сообщениями между процессами параллельной программы.
- Определите схему обмена.
- В исходном тексте программы на языке С пропущены вызовы процедур двухточечного обмена. Предполагается, что при запуске четного числа процессов, те из них, которые имеют четный ранг, отправляют сообщение следующим по величине ранга процессам. Добавить эти вызовы, откомпилировать и запустить программу (на С либо на Java).



```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
int myrank, size, message;
int TAG = 0;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
message = myrank;
if((myrank % 2) == 0)
  {
        if((myrank + 1) != size)
        MPI_Send(...);
}else
       {  if(myrank != 0)
        MPI_Recv(...);
        printf("received :%i\n", message);
        }
MPI_Finalize();
return 0;
}
```

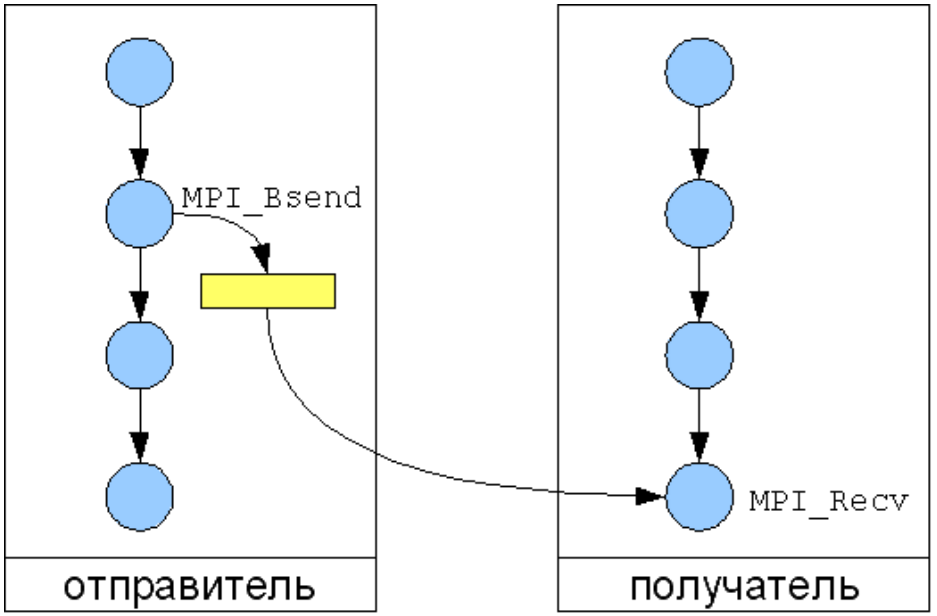


Блокирующий двухточечный обмен с буферизацией

Двухточечные обмены (буферизация)

- Передача сообщения в буферизованном режиме может быть начата независимо от того, зарегистрирован ли соответствующий прием. Источник копирует сообщение в буфер, а затем передает его в неблокирующем режиме. Эта операция локальна, поскольку ее выполнение не зависит от наличия соответствующего приема. Если объем буфера недостаточен, возникает ошибка. Выделение буфера и его размер контролируются программистом.

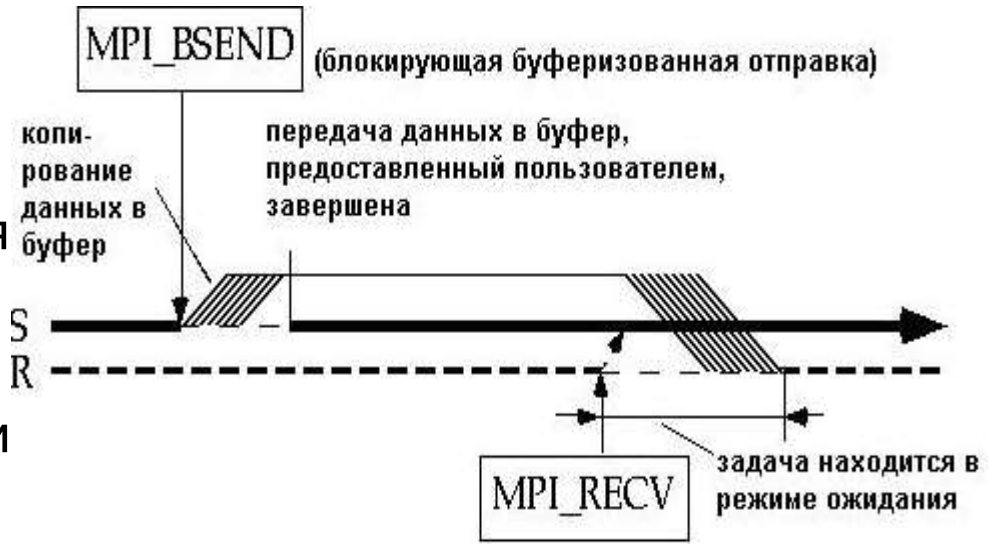
- Процесс-**отправитель** выделяет буфер и регистрирует его в системе.
- Функция `MPI_Bsend` помещает данные в выделенный буфер.



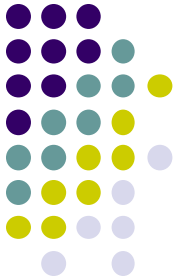
Двухточечные обмены
(буферизация)

Блокирующая буферизованная отправка MPI_Bsend () копирует данные из буфера сообщения в буфер предложенный пользователем. Отправляющая задача может поэтому выполнять вычисления, которые модифицируют оригинальный буфер сообщения зная что эти модификации не будут отражены в данных уже отправленных. Данные будут скопированы из предложенного пользователем буфера в сеть, когда будет получено уведомление "готов получить".

- Буферизованный способ испытывает дополнительную системную накладку (ожидание) из-за дополнительного копирования буфера сообщения в пользовательский буфер.
- Синхронизационная накладка исключается на задаче отправки -- время получения теперь не имеет отношения к отправителю.



Двухточечные обмены (буферизация)

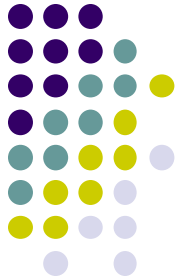


- Размер буфера должен превосходить размер сообщения на величину MPI_BSEND_OVERHEAD. Это дополнительное пространство используется подпрограммой буферизованной передачи для своих целей.

- Если перед выполнением операции буферизованного обмена не выделен буфер, MPI ведет себя так, как если бы с процессом был связан буфер нулевого размера. Работа с таким буфером обычно завершается сбоем программы.

- Буферизованный обмен рекомендуется использовать в тех ситуациях, когда программисту требуется больший контроль над распределением памяти. Этот режим удобен и для отладки, поскольку причину переполнения буфера определить легче, чем причину тупика.

Двухточечные обмены
(буферизация)



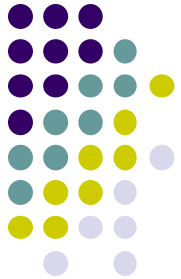
При выполнении буферизованного обмена программист должен заранее создать буфер достаточного размера:

```
int MPI_Buffer_attach(void *buf, size)
```

```
MPI_Buffer_attach(buf, size, ierr)
```

В результате вызова создается буфер `buf` размером `size` байтов. В программах на языке Fortran роль буфера может играть массив. За один раз к процессу может быть подключен только один буфер.

Двухточечные обмены
(буферизация)



Буферизованная передача завершается сразу, поскольку сообщение немедленно копируется в буфер для последующей передачи. В отличие от стандартного обмена, в этом случае работа источника и адресата не синхронизована:

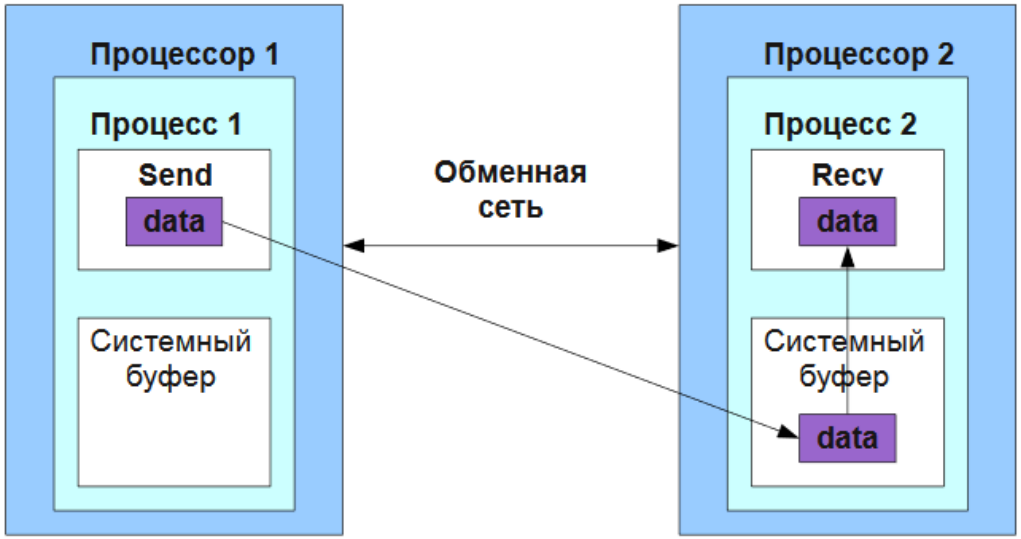
```
int MPI_Bsend(void *buf, int count, MPI_Datatype
datatype, int     dest, int tag, MPI_Comm comm)
```

MPI_Bsend(buf, count, datatype, dest, tag, comm, ierr)

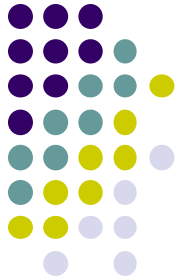
Двухточечные обмены
(буферизация)

Системный буфер MPI

- Полностью скрыт от программиста и вся работа осуществляется средствами библиотеки
- Ограниченный ресурс, за размеры которого легко вылезти
- Может быть на отсылающей, на принимающей или обоих сторонах
- Иногда позволяет увеличить скорость работы программы за счет асинхронных взаимодействий



Двухточечные обмены
(буферизация)

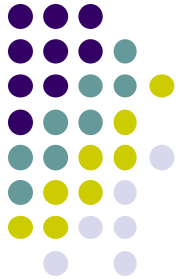


После завершения работы с буфером его необходимо отключить:

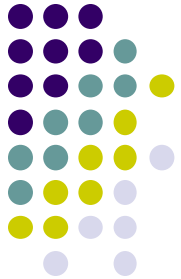
```
int MPI_Buffer_detach(void *buf, int *size)

MPI_Buffer_detach(buf, size, ierr)
```

• Передается адрес (`buf`) и размер отключаемого буфера (`size`). Эта операция блокирует работу процесса до тех пор, пока все сообщения, находящиеся в буфере, не будут обработаны.
• Вызов данной подпрограммы можно использовать для форсированной передачи сообщений. После завершения вызова можно вновь использовать память, которую занимал буфер. В языке С данный вызов не освобождает автоматически память, отведенную для буфера.

Пример программы на C, использующей обмен с буферизацией



```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
 int *buffer;
 int myrank;
 MPI_Status status;
 int buffsize = 1;
 int TAG = 0;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 if (myrank == 0)
{
        buffer = (int *) malloc(buffsize + MPI_BSEND_OVERHEAD);
        MPI_Buffer_attach(buffer, buffsize + MPI_BSEND_OVERHEAD);
        buffer = (int *) 10;
        MPI_Bsend(&buffer, buffsize, MPI_INT, 1, TAG, MPI_COMM_WORLD);
        MPI_Buffer_detach(&buffer, &buffsize);
}     else
{    MPI_Recv(&buffer, buffsize, MPI_INT, 0, TAG, MPI_COMM_WORLD, &status);
 printf("received: %i\n", buffer);
}
 MPI_Finalize();
 return 0;}
```

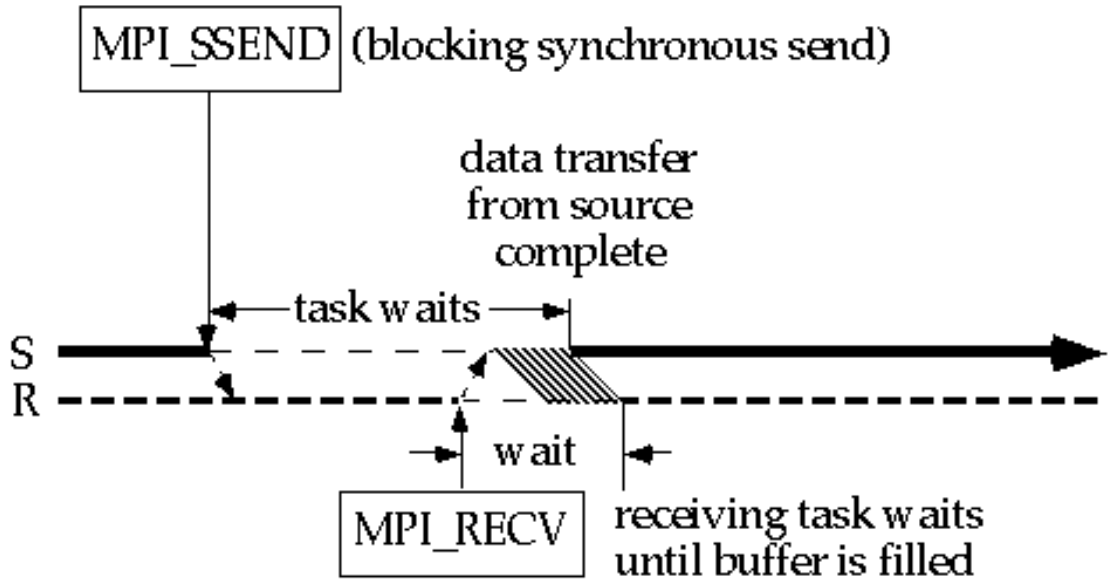


Другие разновидности двухточечного блокирующего обмена

Двухточечные обмены

Синхронный обмен

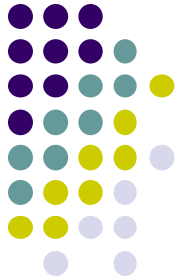
Линия **S** - время исполнения для задачи отправки (на одном ***узле***), линия **R** - время исполнения для задачи получения (на втором узле). Разрывы в этих линиях представляют прерывания, обусловленные событием передачи сообщения.



Задача отправки передает задаче получения сообщение "готова к отправке". Когда задача получателя исполняет вызов получения, и посылает сообщение "готова к получению". Затем данные передаются.

Двухточечные обмены

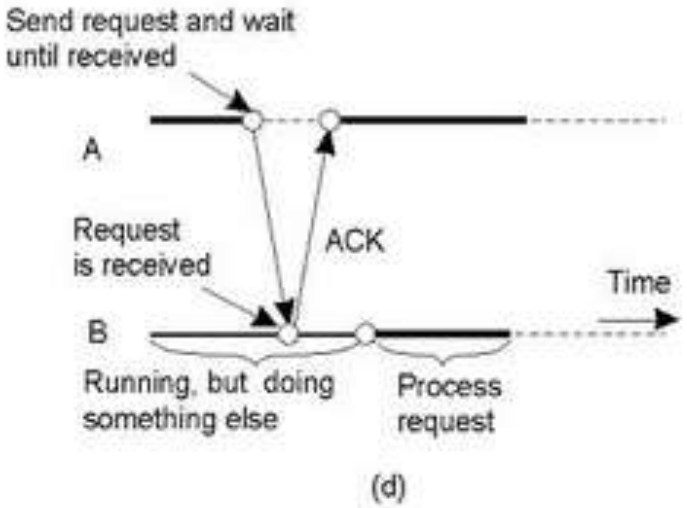
Синхронный обмен (отличие от MPI_SEND)



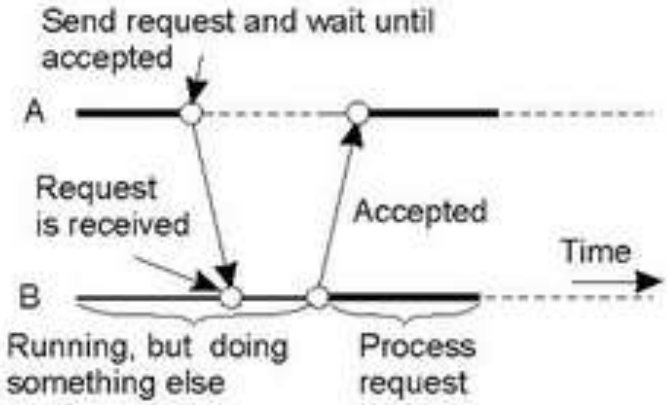
- Есть небольшая, но важная разница между MPI_Send и MPI_Ssend (вы можете найти ее в документе стандарта MPI в разделе 3.4).
- С обычной MPI_SEND, реализация вернется в приложение, когда буфер станет доступен для повторного использования. Это возможно перед тем, как процесс-получатель фактически разместил сообщение о приеме. Например, это может быть, когда небольшое сообщение было скопировано во внутренний буфер и буфер приложения больше не требуется.
- Однако, для больших сообщений, которые не могут быть буферизованы сразу целиком, вызов не может быть возвращен, пока достаточная часть сообщения не была отправлена на удаленный процесс.

Двухточечные обмены

Синхронный обмен (отличие от MPI_SEND)



(d)



(e)

Blocking Send Operation

the caller will be blocked until the message has been copied to MPI runtime system at the sender's side

Blocking Send Operation

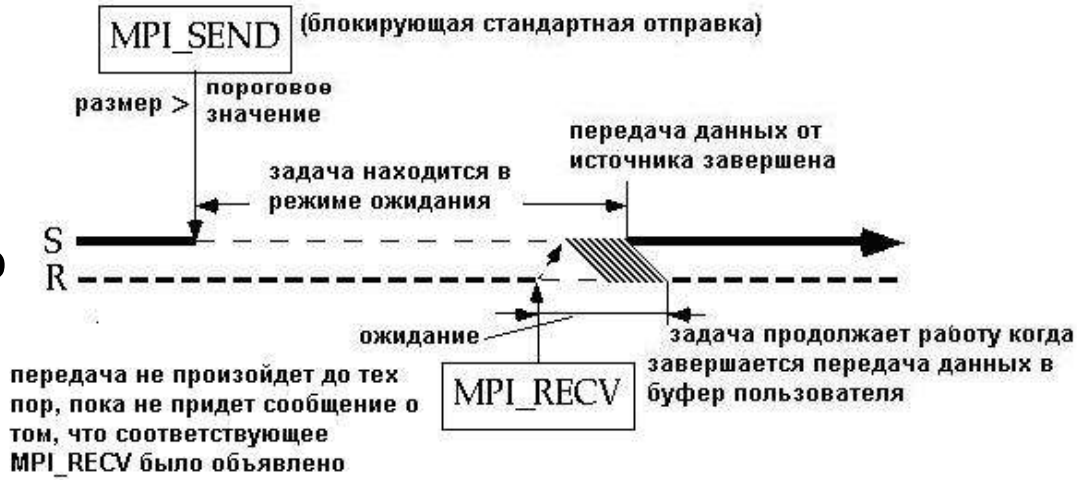
the caller will be blocked until the receiver initiated a receive operation

Двухточечные обмены

Синхронный обмен (отличие от MPI_SEND)

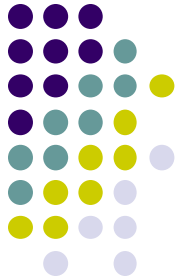
Размер сообщения меньше или равен пороговому значению.

Когда размер сообщения больше порогового значения, поведение блокирующей стандартной отправки MPI_Send, по существу, то же самое, как и в случае синхронного способа.



Двухточечные обмены

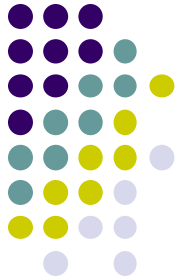
Ограничения на объем сообщений



For standard mode, the library implementer specifies the system behavior that will work best for most users on the target system. For IBM's MPI, there are two scenarios, depending on whether the message size is greater or smaller than a threshold value (called the **eager limit**). The eager limit depends on the number of tasks in the application:

Number of Tasks	Eager Limit (bytes) = threshold
1 - 16	4096
17 - 32	2048
33 - 64	1024
65 - 128	512

Двухточечные обмены



Синхронный обмен

Завершение передачи происходит только после того, как прием сообщения закончен другим процессом. Адресат посылает источнику «квитанцию» - уведомление о завершении приема. После получения этого уведомления обмен считается завершенным и источник "знает", что его сообщение получено:

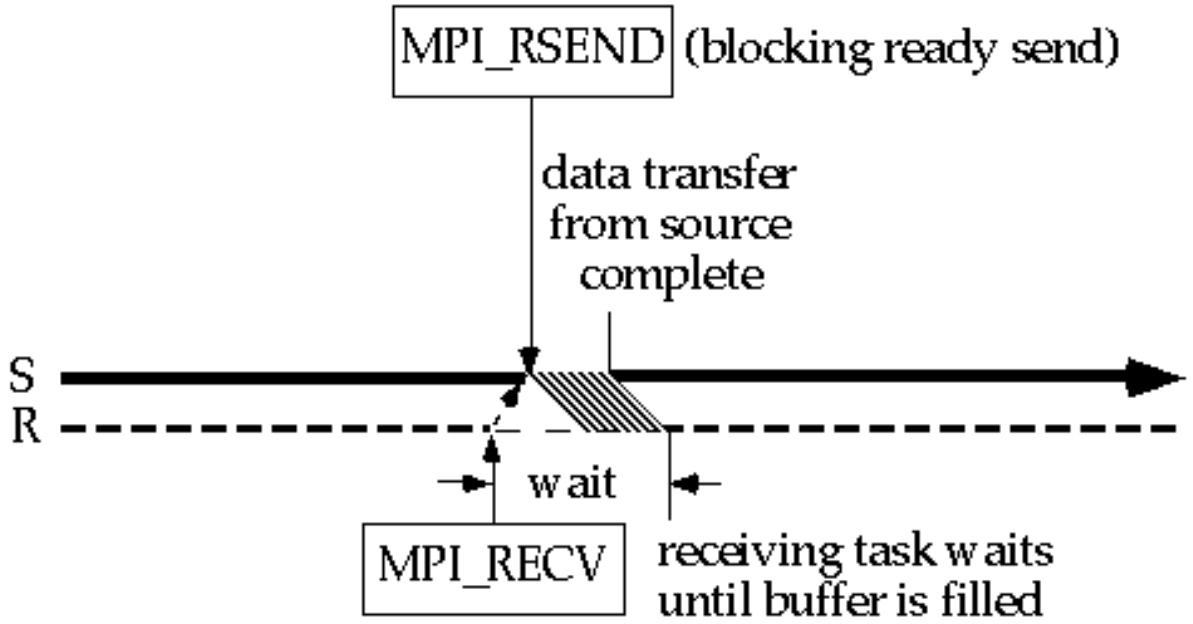
```
int MPI_Ssend(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
```

```
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,
IERR)
```

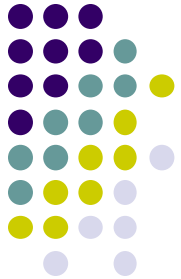
Двухточечные обмены

Обмен «по готовности»

Отправка способом по готовности требует, чтобы прибыло уведомление "готов к получению". Если сообщение не прибыло, то отправка способом по готовности выдаст ошибку.
Способ по готовности имеет целью минимизировать системное ожидание и синхронизационное ожидание, вызванное задачей отправления.



Двухточечные обмены



Обмен «по готовности»

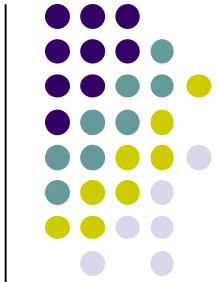
Передача «по готовности» выполняется с помощью подпрограммы `MPI_Rsend`:

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

`MPI_Rsend(buf, count, datatype, dest, tag, comm, ierr)`

Передача «по готовности» должна начинаться, если уже зарегистрирован соответствующий прием. При несоблюдении этого условия результат выполнения операции не определен.

Двухточечные обмены



Обмен «по готовности»

- Завершение передачи не зависит от того, вызвана ли другим процессом подпрограмма приема данного сообщения или нет, оно означает только, что буфер передачи можно использовать вновь.
- Сообщение просто выбрасывается в коммуникационную сеть в надежде, что адресат его получит. Эта надежда может и не сбыться.
- Обмен «по готовности» может увеличить производительность программы, поскольку здесь не используются этапы установки межпроцессных связей, а также буферизация. Все это – операции, требующие времени.
- Таким образом, обмен «по готовности» быстр, но потенциально опасен: он усложняет отладку, поэтому **его рекомендуется использовать только в том случае, когда правильная работа программы гарантируется ее логической структурой, а выигрыша в быстродействии надо добиться любой ценой.**

Двухточечные обмены

Совместные прием и передача

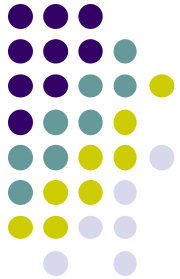
```
MPI_Send(..., anyRank ,...); /* Посылаем данные */
MPI_Recv(..., anyRank ,...); /* Принимаем подтверждение */
```

Ситуация настолько распространенная, что в MPI специально введены две функции, осуществляющие одновременно посылку одних данных и прием других.

Операции приемопередачи объединяют в едином вызове передачу сообщения одному процессу и прием сообщения от другого процесса. Данный вид обменов может оказаться полезным при выполнении сложных схем обмена сообщениями, например, по цепи процессов.

Подпрограммы приемопередачи могут взаимодействовать с обычными подпрограммами обмена и подпрограммами зондирования.

Двухточечные обмены



Совместные прием и передача

Подпрограмма `MPI_Sendrecv` выполняет передачу и прием данных с блокировкой:

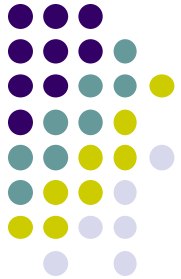
```
int MPI_Sendrecv(void *sendbuf, int sendcount,
MPI_Datatype sendtype, int dest, int sendtag, void
*recvbuf, int recvcount, MPI_Datatype recvtype, int
source, int recvtag, MPI_Comm comm, MPI_Status
*status)
```

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest,
sendtag, recvbuf, recvcount, recvtype, source,
recvtag, comm, status, ierr)
```

Имеются разновидности операции приемопередачи.

Двухточечные обмены

Совместные прием и передача



Подпрограмма **MPI_Sendrecv_replace** выполняет прием и передачу данных, используя общий буфер для передачи и приёма:

```
 int MPI_Sendrecv_replace(void *buf, int count,
 MPI_Datatype datatype, int dest, int sendtag, int
source, int recvtag, MPI_Comm comm, MPI_Status *status)
```

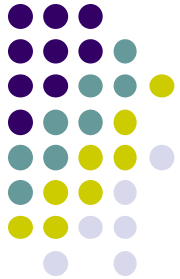
```
 MPI_Sendrecv_replace(BUF, COUNT, DATATYPE, DEST,
SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERR)
```

Показания к применению:

- принимаемые данные должны быть заведомо НЕ ДЛИННЕЕ отправляемых;
- принимаемые и отправляемые данные должны иметь одинаковый тип;
- отправляемые данные затираются принимаемыми.

Двухточечные обмены

Блокировки (deadlock)



Краткая иллюстрация этой ошибки, очень распространенной там, где для пересылок используется разделяемая память.

Вариант 1:

-- Поток A --
Recv(из потока B)
Send(в поток B)

-- Поток B ---
Recv(из потока A)
Send(в поток A)

Вариант 1 вызовет клинч, какой бы инструментарий не использовался: функция приема не вернет управления до тех пор, пока не получит данные; поэтому функция передачи не может приступить к отправке данных; поэтому не работает и функция приема на другой стороне...

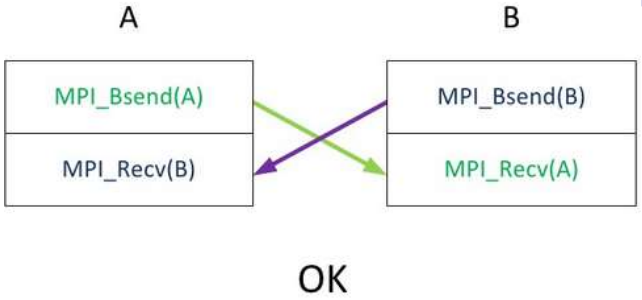
Двухточечные обмены

Блокировки

Вариант 2:

-- Поток A --
Send(в ветвь B)
Recv(из ветви B)

-- Поток B–
Send(в ветвь A)
Recv(из ветви A)



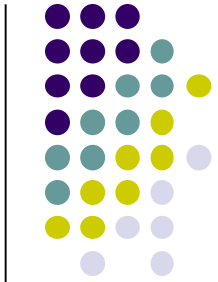
Вариант 2 вызовет клинч, если функция передачи возвращает управление только после того, как данные попали в пользовательский буфер на приемной стороне.

Однако при использовании MPI зависания во втором варианте не произойдет! MPI_Send, если на приемной стороне нет готовности (не вызван MPI_Recv), не станет ее дожидаться, а положит данные во временный буфер и вернет управление программе НЕМЕДЛЕННО.

Когда MPI_Recv будет вызван, данные он получит не из пользовательского буфера напрямую, а из промежуточного системного. Буферизация - дело громоздкое - может быть, и не всегда сильно экономит время, зато повышает надежность: делает программу более устойчивой к ошибкам программиста.

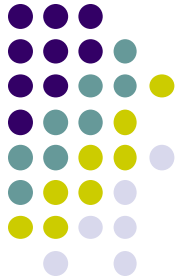
MPI_Sendrecv и MPI_Sendrecv_replace также делают программу более устойчивой: с их использованием программист лишается возможности перепутать варианты 1 и 2.

Двухточечные блокирующие обмены. Выводы.



- Синхронный код безопасен, так как он не зависит от порядка, в котором отправка и получение исполняются (в отличие от способа по готовности) или количества буферного пространства (в отличие от буферизованного способа или стандартного способа). Синхронизационный способ может вызвать существенную синхронизационную накладку.
- Способ по готовности имеет наименьшую суммарную временну*ю* ю накладку. Он действительно не требует встречи (рукопожатия) отправителя и получателя (подобно синхронизированному способу) или дополнительного копирования в буфер (подобно буферизованному или стандартному способу). Этот способ не подходит для всех сообщений.

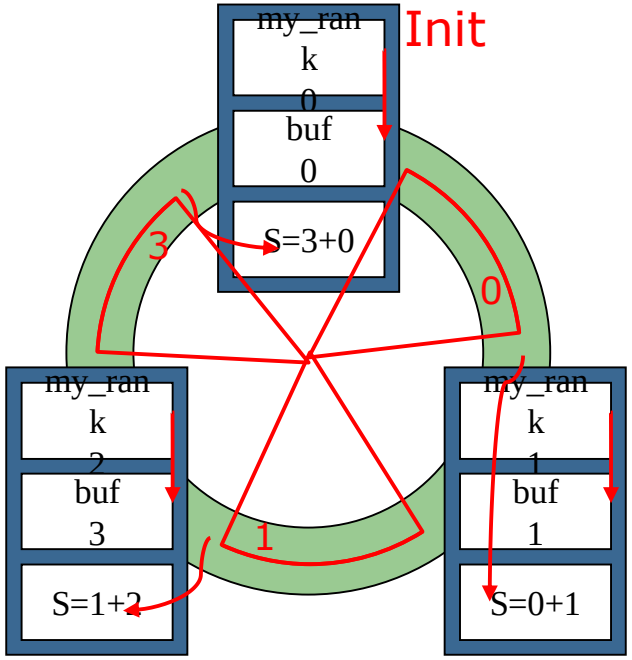
Двухточечные блокирующие обмены. Выводы.



- Буферизованный способ разъединяет отправителя и получателя. Это исключает синхронизационную накладку на отправляющей задаче и гарантирует, что порядок исполнения отправки и получения не имеет значения (в отличие от способа по готовности). Дополнительным преимуществом является то, что программист может контролировать размер сообщений, которые будут буферизованы и полное количество буферного пространства. Существует дополнительная системная накладка по времени, вызванная копированием в буфер.
- Поведение стандартного способа определяется реализацией. Разработчик библиотеки выбирает системное поведение, что обеспечивает хорошую эффективность работы и приемлемую безопасность.

Задание 2 — Пересылка данных по кольцу

- Каждый процессор помещает свой ранг в целочисленную переменную *buf*.
- Каждый процессор пересылает переменную *buf* соседу справа.
- Каждый процессор суммирует принимаемое значение в переменную *s*, а затем передаёт рассчитанное значение соседу справа.
- Пересылки по кольцу прекращаются, когда нулевой процессор просуммирует ранги всех процессоров.

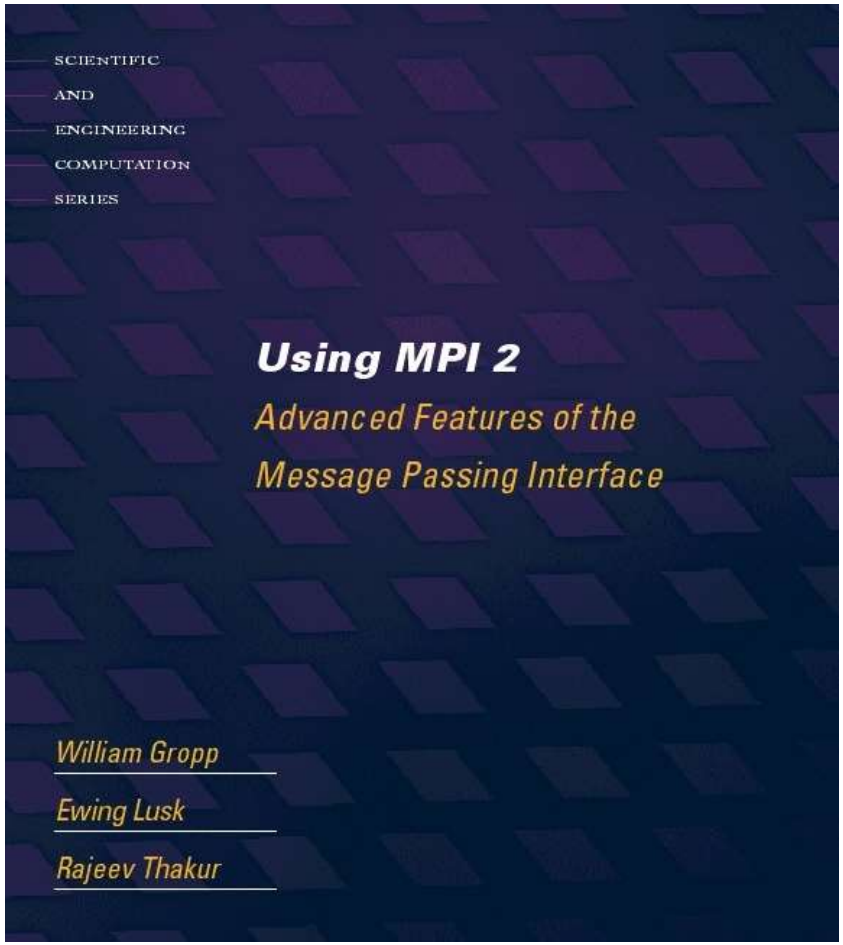


Выполнить 2 варианта: в блокирующем и неблокирующем режиме, там, где это возможно использовать send_recieve.

Литература

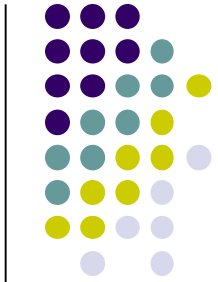
1. Л Е. Карпов. "Архитектура распределенных систем программного обеспечения", М., МАКС Пресс, 2007.
2. Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju. "Web Services. Concepts, Architectures and Applications". Springer-Verlag, 2004.
3. Andrew S. Tanenbaum, Maarten van Steen. "Distributed Systems. Principles and paradigms". Prentice Hall, Inc., 2002 (Э. Таненбаум, М. ван Стеен. "Распределенные системы. Принципы и парадигмы". СПб.: Питер, 2003)
4. *http://www.w3.org/2000/xp/Group/*
5. *http://www.w3.org/TR/*
6. *http://www.w3.org/TR/2008/REC-xml-20081126/*
7. *http://www.w3.org/TRJsoap/*

Tutorial Material



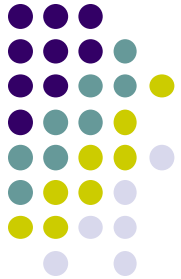
http://www.mcs.anl.gov/mpi/usingmpi2

Обзор технологий параллельного программирования



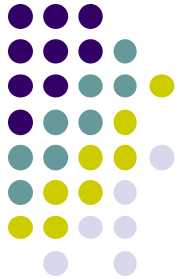
Вопросы к экзамену:

1. Виды двухточечных обменов в MPI. Их особенности, схемы, достоинства и недостатки.
2. Принципы работы с коммуникаторами.



НЕБЛОКИРУЮЩИЕ ДВУХТОЧЕЧНЫЕ ОБМЕНЫ

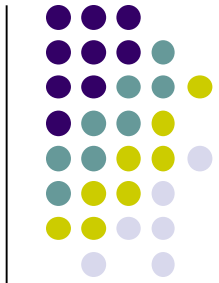
Неблокирующие обмены



- Вызов подпрограммы неблокирующей передачи инициирует, но не ждет ее завершения. Завершится выполнение подпрограммы может еще до того, как сообщение будет скопировано в буфер передачи.

- Применение неблокирующих операций улучшает производительность программы, поскольку в этом случае допускается перекрытие (то есть одновременное выполнение) вычислений и обменов. Передача данных из буфера или их считывание может происходить одновременно с выполнением процессом другой работы.

Неблокирующие обмены



- Неблокирующий обмен выполняется в два этапа:
  1. Инициализация обмена.
  2. Проверка завершения обмена.

- Для завершения неблокирующего обмена требуется вызов дополнительной процедуры, которая проверяет, скопированы ли данные в буфер передачи.

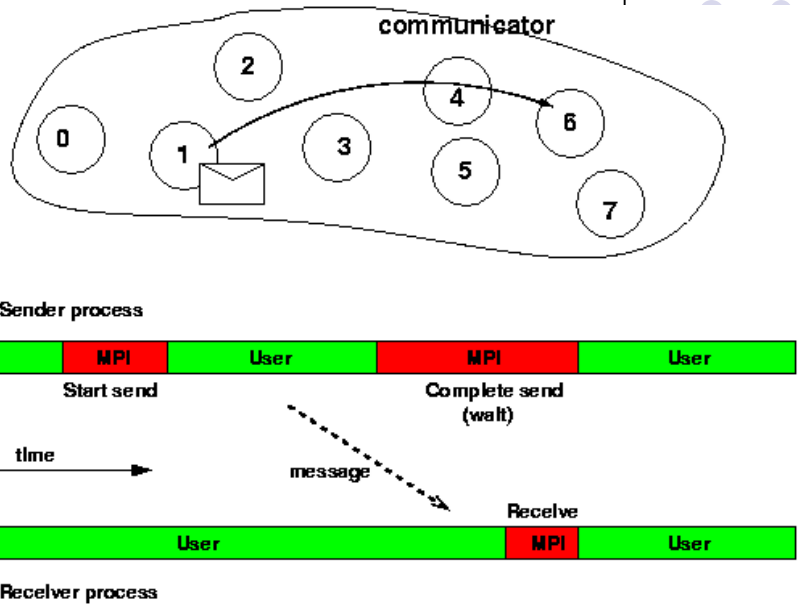
ВНИМАНИЕ!

- При неблокирующем обмене возвращение из подпрограммы обмена происходит сразу, но запись в буфер или считывание из него после этого производить нельзя - сообщение может быть еще не отправлено или не получено и работа с буфером может «испортить» его содержимое.

Неблокирующие обмены

Неблокирующий обмен выполняется в два этапа:

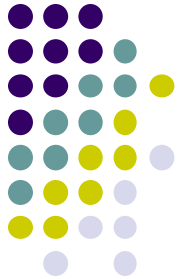
1. **Инициализация** обмена;
2.Выполнение каких-либо вычислений.
3. **Проверка завершения** обмена.



Разделение этих шагов делает необходимым *маркировку* каждой операции обмена, которая позволяет целенаправленно выполнять проверки завершения соответствующих операций.

Для маркировки в неблокирующих операциях используются *идентификаторы операций обмена*.

Неблокирующие обмены



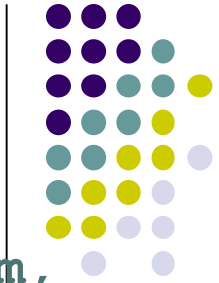
Инициализация неблокирующей стандартной ***передачи*** выполняется подпрограммами `MPI_I[S, B, R]send`. Стандартная неблокирующая передача выполняется подпрограммой:

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

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

Входные параметры этой подпрограммы аналогичны аргументам подпрограммы `MPI_Send`.
Выходной параметр `request` - идентификатор операции.

Неблокирующие обмены (Java)



```
C: MPI_Isend(buf, count, datatype, dest, tag, comm,
request, ierr)
```

```
JAVA: Request Comm.Isend(Object buf, int offset,
int count, Datatype datatype, int dest, int tag)
```

buf send buffer array

offset initial offset in send buffer

count number of items to send

datatype datatype of each item in send buffer

dest rank of destination

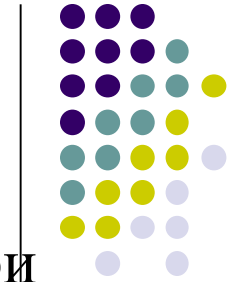
tag message tag

returns: communication ***request***

Non-blocking methods return a `Request` object:

- `Wait()` //waits until communication completes
- `Test()` //test if the communication has finished

Неблокирующие обмены



Инициализация неблокирующего ***приема*** выполняется при вызове подпрограммы:

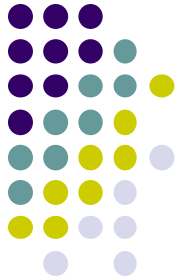
```
int MPI_Irecv(void *buf, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm comm,
MPI_Request *request)
```

C: MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierr)

JAVA: Request Comm.Irecv(Object buf, int offset, int count, Datatype datatype, int source, int tag)

Назначение аргументов здесь такое же, как и в ранее рассмотренных подпрограммах, за исключением того, что указывается ранг не адресата, а источника сообщения (`source`).

Неблокирующие обмены

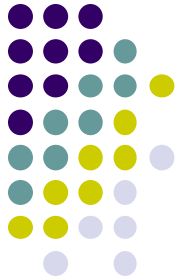


Вызовы подпрограмм неблокирующего обмена формируют ***запрос*** на выполнение операции обмена и связывают его с идентификатором операции **`request`**. Запрос идентифицирует свойства операции обмена:

- ❑ режим;
- ❑ характеристики буфера обмена;
- ❑ контекст;
- ❑ тег и ранг.

Запрос содержит информацию о состоянии ожидающих обработки операций обмена и может быть использован для получения информации о состоянии обмена или для ожидания его завершения.

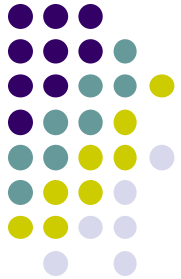
Неблокирующие обмены



Проверка выполнения обмена (Wait и Test)

Проверка фактического выполнения передачи или приема в неблокирующем режиме осуществляется с помощью **вызова *подпрограмм ожидания***, блокирующих работу процесса до завершения операции или неблокирующих подпрограмм проверки, возвращающих логическое значение «истина», если операция выполнена.

Неблокирующие обмены



В том случае, когда одновременно несколько процессов обмениваются сообщениями, можно использовать проверки, которые применяются одновременно к нескольким обменам.

Есть три типа таких проверок:

1. проверка завершения всех обменов;
2. проверка завершения любого обмена из нескольких;
3. проверка завершения заданного обмена из нескольких.

Каждая из этих проверок имеет две разновидности:

1. «ожидание» - блокирующая проверка;
2. «проверка» - неблокирующая проверка.

Неблокирующие обмены

Блокирующие операции проверки

MPI_Wait (S) **вызывается непосредственно перед тем, как задача отправки заполнит буфер сообщения новыми данными.**

Подпрограмма `MPI_Wait` блокирует работу процесса до завершения приема или передачи сообщения:

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

```
C:     MPI_Wait(request, status, ierr)
JAVA:  Status Request.Wait()
```

Входной параметр `request` – идентификатор операции обмена, выходной – статус (`status`).

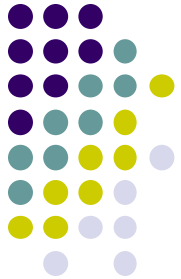
Неблокирующие обмены

Блокирующие операции проверки

MPI_Wait (S)

This is most necessary when you have a loop that repeatedly fills the message buffer and sends the message. You can't write anything new into that buffer until you know for sure that the preceding message has been successfully copied out of the buffer. Even if a send buffer is not re-used, it is advantageous to complete the communication, as this releases system resources.

Неблокирующие обмены

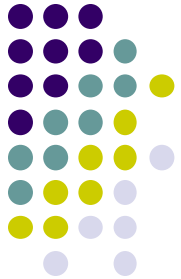


Операции проверки JAVA Status

```
Status Comm.Recv(Object buf, int offset, int count,
Datatype datatype, int source, int tag)
```

buf receive buffer array
offset initial offset in receive buffer
count number of items in receive buffer
datatype datatype of each item in receive buffer
source rank of source tag message tag
returns: status object

Неблокирующие обмены



- Успешное выполнение подпрограммы ***MPI_Wait***() после вызова ***MPI_Ibsend*** подразумевает, что буфер передачи можно использовать вновь, то есть, пересылаемые данные отправлены или скопированы в буфер, выделенный при вызове подпрограммы ***MPI_Buffer_attach***.
- В этот момент уже нельзя отменить передачу. Если не будет зарегистрирован соответствующий прием, буфер нельзя будет освободить. В этом случае можно применить подпрограмму ***MPI_Cancel***, которая освобождает память, выделенную подсистеме коммуникаций.

Неблокирующие обмены

Проверка завершения всех обменов

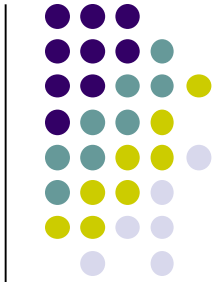
Проверка завершения всех обменов выполняется подпрограммой:

```
int MPI_Waitall(int count, MPI_Request requests[],
MPI_Status statuses[])
```

C: MPI_Waitall(count, requests, statuses, ierr)
JAVA: static Status [] Request.Waitall(Request [] array_of_requests)

При вызове этой подпрограммы выполнение процесса блокируется до тех пор, пока **все** операции обмена, связанные с активными запросами в массиве `requests`, не будут выполнены. Возвращается статус этих операций. Статус обменов содержится в массиве `statuses`. `count` - количество запросов на обмен (размер массивов `requests` и `statuses`).

Неблокирующие обмены

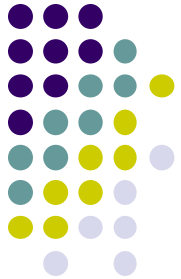


- В результате выполнения подпрограммы MPI_Waitall запросы, сформированные неблокирующими операциями обмена, аннулируются, а соответствующим элементам массива присваивается значение MPI_REQUEST_NULL.

- В случае неуспешного выполнения одной или более операций обмена подпрограмма MPI_Waitall возвращает код ошибки MPI_ERR_IN_STATUS и присваивает полю ошибки статуса значение кода ошибки соответствующей операции.

- Если операция выполнена успешно, полю присваивается значение MPI_SUCCESS, а если не выполнена, но и не было ошибки - значение MPI_ERR_PENDING. Это соответствует наличию запросов на выполнение операции обмена, ожидающих обработки.

Неблокирующие обмены



Проверка завершения любого числа обменов

Проверка завершения любого числа обменов выполняется подпрограммой:

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

C: MPI_Waitany(count, requests, index, status, ierr)

JAVA: static Status Request.Waitany(Request [] array_of_requests)

Выполнение процесса блокируется до тех пор, пока, по крайней мере, один обмен из массива запросов (`requests`) не будет завершен.

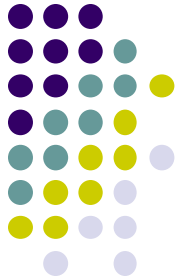
Входные параметры:

- `requests` - запрос;
- `count` - количество элементов в массиве `requests`.

Выходные параметры:

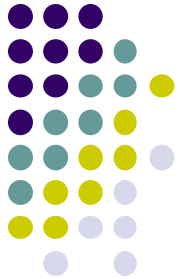
- `index` - индекс запроса (в языке C это целое число от 0 до `count`-1 , а в языке Fortran от 1 до `count`) в массиве `requests`;
- `status` - статус.

Неблокирующие обмены



Если в списке вообще нет активных запросов или он пуст, вызовы завершаются сразу со значением индекса `MPI_UNDEFINED (MPI.UNDEFINED)` и пустым статусом, это подразумевает, в частности, что все неблокирующие передачи обмена завершены.

Неблокирующие обмены



Неблокирующие процедуры проверки

Подпрограмма **`MPI_Test()`** выполняет неблокирующую проверку завершения приема или передачи сообщения: Там где wait задерживает исполнение до тех пор пока операция завершится, test возвращается немедленно с информацией о ее текущем состоянии.

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

C: `MPI_Test(request, flag, status, ierr)`

Входной параметр: идентификатор операции обмена `request`.

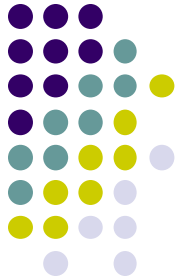
Выходные параметры:

- `flag` – «истина», если операция, заданная идентификатором `request`, выполнена;
- `status` – статус выполненной операции.

JAVA: `Status Request.Test()`

returns: status object or null reference

Неблокирующие обмены



Неблокирующая проверка завершения всех обменов

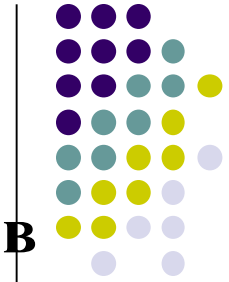
Подпрограмма **`MPI_Testall`** выполняет неблокирующую проверку завершения приема или передачи всех сообщений:

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

```
MPI_Testall(count, requests, flag, statuses, ierr)
```

При вызове возвращается значение флага (`flag`) «истина», если все обмены, связанные с активными запросами в массиве `requests`, выполнены. Если завершены не все обмены, флагу присваивается значение «ложь», а массив `statuses` не определен.
Параметр `count` - количество запросов.
Каждому статусу, соответствующему активному запросу, присваивается значение статуса соответствующего обмена.

Неблокирующие обмены



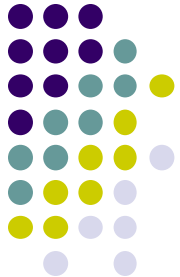
Неблокирующая проверка завершения всех обменов

JAVA:

```
static Status [] Request.Testall(Request []
array_of_requests)
```

array of requests – массив запросов
возвращает : массив статус-объектов, или null

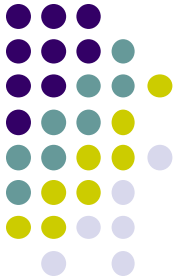
Неблокирующие обмены



Другие операции проверки

Подпрограммы **`MPI_Waitsome`** и **`MPI_Testsome`** действуют аналогично подпрограммам **`MPI_Waitany`** и **`MPI_Testany`**, кроме случая, когда завершается более одного обмена. В подпрограммах **`MPI_Waitany`** и **`MPI_Testany`** обмен из числа завершенных выбирается произвольно, именно для него и возвращается статус, а для **`MPI_Waitsome`** и **`MPI_Testsome`** статус возвращается для всех завершенных обменов. Эти подпрограммы можно использовать для определения того, сколько обменов завершено.

Неблокирующие обмены



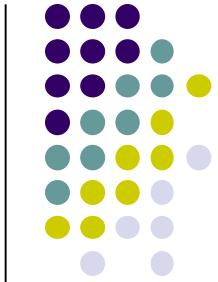
Интерфейс этих подпрограмм:

```
int MPI_Waitsome(int incount, MPI_Request requests[], int
*outcount, int indices[], MPI_Status statuses[])
```

MPI_Waitsome(incount, requests, outcount, indices, statuses, ierr)

Здесь `incount` - количество запросов. В `outcount` возвращается количество выполненных запросов из массива `requests`, а в первых `outcount` элементах массива `indices` возвращаются индексы этих операций. В первых `outcount` элементах массива `statuses` возвращается статус завершенных операций. Если выполненный запрос был сформирован неблокирующей операцией обмена, он аннулируется. Если в списке нет активных запросов, выполнение подпрограммы завершается сразу, а параметру `outcount` присваивается значение `MPI_UNDEFINED`.

Неблокирующие обмены



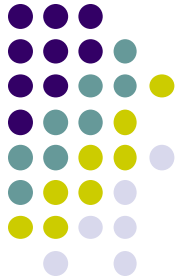
JAVA:

```
static Status [] Request.Testall(Request [] array_of_requests)
```

Возвращает: массив статус-объектов, или null, если нет активных операций обмена.

Tests for completion of all of the operations associated with active requests. Java binding of the MPI operation **`MPI_TESTALL`**. If all operations have completed, the exit values of the argument array and the result array are as for **`Waitall`**. If any operation has not completed, the result value is null and no element of the argument array is modified.

Неблокирующие обмены



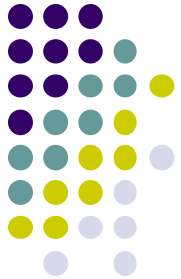
Неблокирующая проверка выполнения обменов:

```
int MPI_Testsome(int incount, MPI_Request
requests[], int *outcount, int indices[],
MPI_Status statuses[])
```

C: MPI_Testsome(incount, requests, outcount, indices, statuses, ierr)

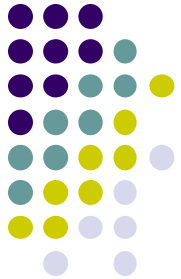
JAVA: static Status [] Request.Testsome(Request [] array_of_requests)

Параметры такие же, как и у подпрограммы `MPI_Waitsome`. Эффективность подпрограммы `MPI_Testsome` выше, чем у `MPI_Testany`, поскольку первая возвращает информацию обо всех операциях, а для второй требуется новый вызов для каждой выполненной операции.



Примеры использования неблокирующих двухточечных обменов

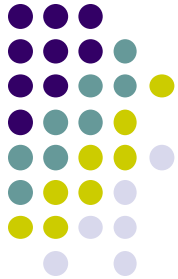
Неблокирующие обмены



Пример 1 Обмен по кольцевой топологии (двунаправленное кольцо) при помощи неблокирующих операций на языке C

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

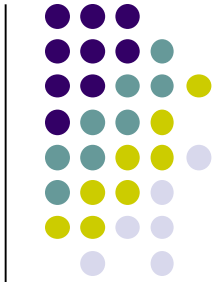
Неблокирующие обмены



Пример 1 (окончание)

```
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);
   printf("process %d prev = %d next=%d\n", rank, buf[0], buf[1]);
   MPI_Finalize();
}
```

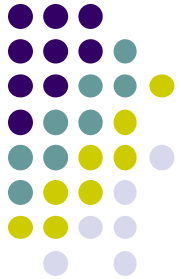
Неблокирующие обмены



Пример 2

```
int main(int nArgC, char *apszArgV[])
{
int nSize, nRank; int N=100;
double adA[N], adB[N];
MPI_Request aRequests[2];
MPI_Status aStatuses[2];
//
MPI_Init(&nArgC, &apszArgV);
MPI_Comm_size(MPI_COMM_WORLD, &nSize);
MPI_Comm_rank(MPI_COMM_WORLD, &nRank);
  if
    (nSize >= 2)
   {
```

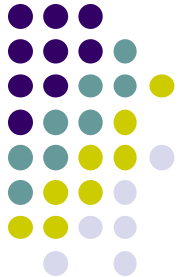
Неблокирующие обмены



Пример 2 (продолжение)

```
switch (nRank)
{
    case 0:
    read_data(adA, adB); /*считывание данных*/
MPI_Isend(adA, N, MPI_DOUBLE, 1, 98, MPI_COMM_WORLD, &aRequests[0]);
MPI_Isend(adB, N, MPI_DOUBLE, 1, 99, MPI_COMM_WORLD, &aRequests[1]);
    MPI_Waitall(2, aRequests, aStatuses);
//
break;
```

Неблокирующие обмены

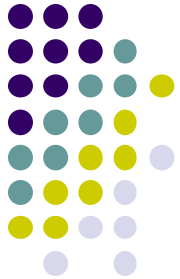


Пример 2 (окончание)

case 1:

```
MPI_Irecv(adA, N, MPI_DOUBLE, 0, 98, MPI_COMM_WORLD, &aRequests[0]);
MPI_Irecv( adB, N, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD, &aRequests[1]);
MPI_Waitall(2, aRequests, aStatuses);
process_data(adA, adB); /*обработка данных*/
    } /* switch (nRank)*/
     } /*if (nSize >= 2)*/
//
MPI_Finalize();
  }  /*main()*/
```

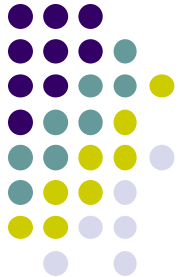
Неблокирующие обмены



Пример 3 (JAVA)

```
MPI.Init(args);
// родительский процесс рассылает приветствие всем остальным процессам
//p2p и получает ответ
int myrank = MPI.COMM_WORLD.Rank();
int size=MPI.COMM_WORLD.Size();
    Request []r=new Request[size-1];
    Status []s =new Status[size-1];
        if(myrank == 0) {
            char[] message ="Hello from boss!".toCharArray();
            for (int i=1;i<size;i++)
                {r[i-1]=MPI.COMM_WORLD.Isend(message, 0, message.length,
                MPI.CHAR, i, 1);
                s[i-1]=r[i-1].Wait();}
            for (int i=0;i<size-1;i++)
            {if (r[i].Is_null())  System.out.println("Isend to "+(i+1)+ "complete");}
```

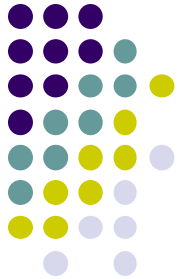
Неблокирующие обмены



Пример 3 (JAVA - окончание)

```
char[] backmessage = new char[15];
          for (int i=1;i<size;i++)
          {MPI.COMM_WORLD.Recv(backmessage, 0, 15, MPI.CHAR, i, 2) ;
           System.out.println("received: from rank "+i+ " "+new
String(backmessage)) ;}
       }
       else {
          char[] message = new char[20] ;
          MPI.COMM_WORLD.Recv(message, 0, 20, MPI.CHAR, 0, 1) ;
          System.out.println("received by: "+myrank+" from rank 0 " +new
String(message)) ;
          char [] backmessage="Hello, master".toCharArray();
          MPI.COMM_WORLD.Send(backmessage, 0, backmessage.length,
MPI.CHAR, 0, 2) ;
       }
MPI.Finalize();
```

Неблокирующие обмены



Результат выполнения ПРИМЕРА № 3:

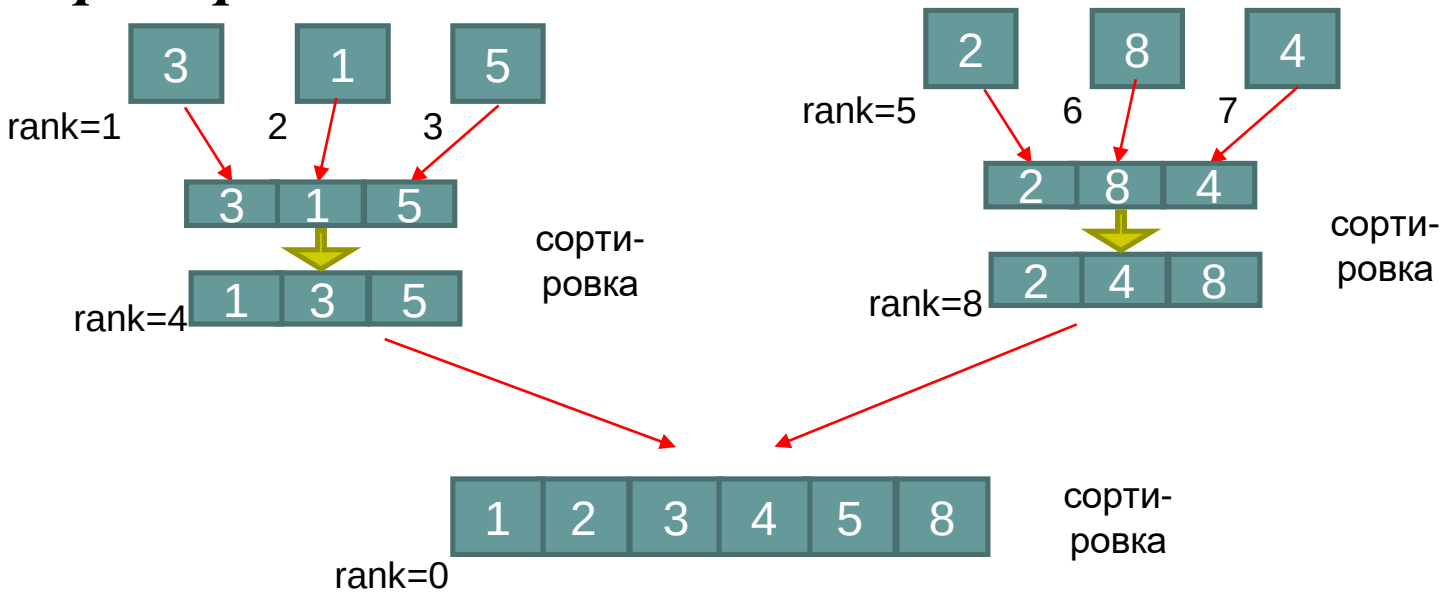
```
run:
MPJ Express (0.44) is started in the multicore configuration
received by: 1 from rank 0 Hello from boss!
Isend to 1complete
received by: 3 from rank 0 Hello from boss!
received by: 2 from rank 0 Hello from boss!
Isend to 2complete
Isend to 3complete
Isend to 4complete
received by: 4 from rank 0 Hello from boss!
received: from rank 1 Hello, master
received: from rank 2 Hello, master
received: from rank 3 Hello, master
received: from rank 4 Hello, master
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 2 секунды)
```

Неблокирующие обмены

Задание 3

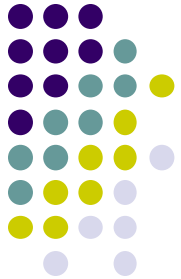
Реализовать так называемую задачу фильтрации, используя неблокирующие обмены+Waitall()+пробник

например:



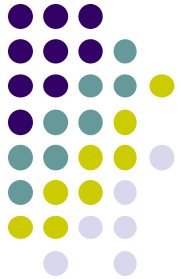
Вариант: к-во потоков 1-го уровня вычисляется так: (№ пп %4)+3

Более высокую оценку получат решения, предоставляющие возможность использовать переменное количество процессов.



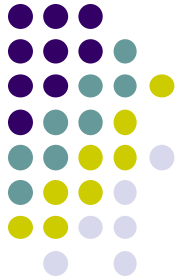
Подпрограммы-пробники

Неблокирующие обмены (пробники)



- Получить информацию о сообщении до его помещения в буфер приема можно с помощью подпрограмм-пробников **`MPI_Probe`** и **`MPI_IProbe`**.
- На основании полученной информации принимается решение о дальнейших действиях.
- С помощью вызова подпрограммы **`MPI_Probe`** фиксируется поступление (но не прием!) сообщения. Затем определяется источник сообщения, его длина, выделяется буфер подходящего размера и выполняется прием сообщения.
- Функция MPI_IPROBE (source, tag, comm, flag, status) возвращает flag = true, если имеется сообщение, которое может быть получено и проанализировано.

Неблокирующие обмены (пробники)



Неблокирующая проверка сообщения C

Неблокирующая проверка сообщения выполняется подпрограммой:

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

`MPI_Iprobe(source, tag, comm, flag, status, ierr)`

Входные параметры этой подпрограммы те же, что и у подпрограммы `MPI_Probe`. Выходные параметры:

- `flag` - флаг;
- `status` - статус.

Если сообщение уже поступило и может быть принято, возвращается значение флага «истина».

Неблокирующие обмены (пробники)

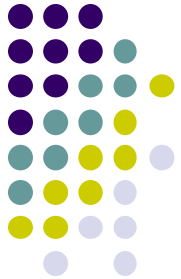
Неблокирующая проверка сообщения JAVA

```
Status Comm.Iprobe(int source, int tag)
source source
rank tag tag value
```

Probe
public Status Probe(int source, int tag) throws MPIException

Wait until there is an incoming message matching the pattern specified.
Java binding of the MPI operation MPI_PROBE.
Returns a status object similar to the return value of a matching Recv operation.
Throws: MPIException

Неблокирующие обмены (пробники)



Размер полученного сообщения (`count`) можно определить с помощью вызова подпрограммы

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype
datatype, int *count)
```

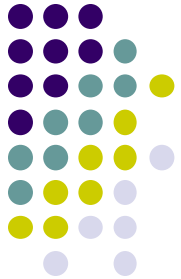
C: MPI_Get_count(status, datatype, count, ierr)
JAVA: int Status.Get_count(Datatype datatype)

Параметры:

- `count` - количество элементов в буфере передачи;
- `datatype` - тип каждого пересылаемого элемента;
- `status` - статус обмена;
- `ierr` - код завершения.

Аргумент `datatype` должен соответствовать типу данных, указанному в операции обмена.

Неблокирующие обмены (пробники)

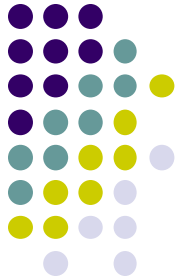


Задание 3.1

Дополнить программу с пробниками, выполнить ее, затем использовать пробники в задании 3.

```
int data[] = new int[1];
int buf[] = {1,3,5};
int count,TAG = 0;
Status st;
data[0] = 2016;
MPI.Init(arg);
int rank = MPI.COMM_WORLD.Rank();
int size=MPI.COMM_WORLD.Size();
if(rank == 0)
{
    MPI.COMM_WORLD.Send(data, 0, 1, MPI.INT, 2, TAG);
 }
 else if(rank == 1){
          MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, 2, TAG);
 }
```

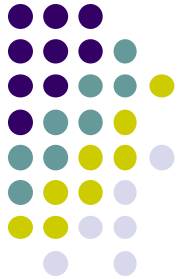
Неблокирующие обмены (пробники)



Задание 3.1

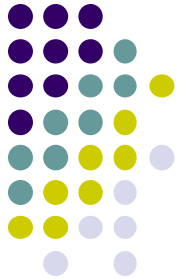
(окончание)

```
else if(rank == 2){
        st = MPI.COMM_WORLD.Probe(…);
        count = st.Get_count(MPI.INT);
        MPI.COMM_WORLD.Recv(back_buf,0,count,MPI.INT,0,TAG);
        System.out.print("Rank = 0 ");
        for(int i = 0 ; i < count ; i ++)
            System.out.print(back_buf[i]+" ");

        st = MPI.COMM_WORLD.Probe(…);
        count = st.Get_count(MPI.INT);
        MPI.COMM_WORLD.Recv(back_buf2,0,count,MPI.INT,1,TAG);
        System.out.print("Rank = 1 ");
        for(int i = 0 ; i < count ; i ++)
            System.out.print(back_buf2[i]+" ");
    }
    MPI.Finalize();
```



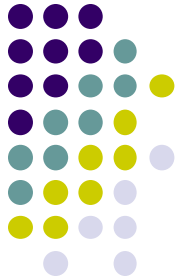
Отложенные обмены

Неблокирующие отложенные обмены



- Процедуры данной группы позволяют снизить накладные расходы, возникающие в рамках одного процессора при обработке приема/передачи и перемещении необходимой информации между процессом и сетевым контроллером.
- Часто в программе приходится многократно выполнять обмены с одинаковыми параметрами (например, в цикле). В этом случае можно один раз инициализировать операцию обмена и потом многократно ее запускать, не тратя на каждой итерации дополнительного времени на инициализацию и заведение соответствующих внутренних структур данных.
- Кроме того, таким образом, несколько запросов на прием и/или передачу могут объединяться вместе для того, чтобы далее их можно было бы запустить одной командой (впрочем, это совсем необязательно хорошо, поскольку может привести к перегрузке коммуникационной сети).

Неблокирующие отложенные обмены



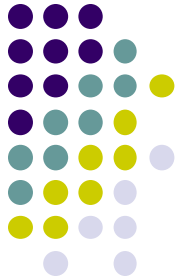
Запрос для стандартной передачи создается при вызове подпрограммы `MPI_Send_init`:

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

`MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierr)`

- Формирование *отложенного запроса* на посылку сообщения. Сама операция пересылки при этом не начинается!
- Способ приема сообщения никак не зависит от способа его посылки: сообщение, отправленное с помощью отложенных запросов либо обычным способом, может быть принято как обычным способом, так и с помощью отложенных запросов.

Неблокирующие отложенные обмены



Отложенный запрос может быть сформирован для всех режимов обмена. Для этого используются подпрограммы **`MPI_Bsend_init`**, **`MPI_Ssend_init`** и **`MPI_Rsend_init`**.

Отложенный обмен инициируется вызовом подпрограммы `MPI_Start`:

```
int MPI_Start(MPI_Request *request)
```

```
MPI_Start(request, ierr)
```

Неблокирующие отложенные обмены

Подпрограмма `MPI_Startall`:

```
int MPI_Startall(int count, MPI_request *requests)
```

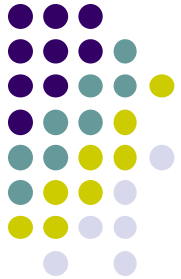
```
MPI_Startall(count, requests, ierr)
```

инициирует все обмены, связанные с запросами на выполнение неблокирующей операции обмена в массиве `requests`.

Завершается обмен при вызове **`MPI_Wait, MPI_Test`** и некоторых других подпрограмм.

- В отличие от неблокирующих операций, по завершении выполнения операции, запущенной при помощи отложенного запроса на взаимодействие, значение параметра REQUEST (REQUESTS) сохраняется и может использоваться в дальнейшем!

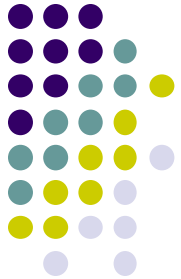
Неблокирующие отложенные обмены



```
Mpi_request_free(request, ierr)
integer request, ierr
```

Данная процедура удаляет структуры данных, связанные с параметром **`Request`**.
После ее выполнения параметр **`Request`** устанавливается в значение **`Mpi_request_null`**.
Если операция, связанная с этим запросом, уже выполняется, то она будет завершена.

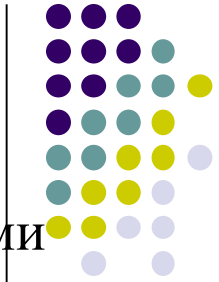
Неблокирующие отложенные обмены



Пример 3. Схема итерационного метода с обменом по кольцевой топологии при помощи отложенных запросов на языке Си

```
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]);
```

Задание 4



Два вектора **a** и **b** размерности N представлены двумя одномерными массивами, содержащими каждый по N элементов. Напишите параллельную MPI-программу вычисления скалярного произведения этих векторов используя блокирующий двухточечный обмен сообщениями. Программа должна быть организована по схеме master-slave, причем master-процесс должен пересылать подчиненным процессам одинаковые по количеству элементов фрагменты векторов.

1.Измерьте и проанализируйте затраченное время во всех случаях.
2.Проведите исследование зависимости ускорения параллельной программы от размера сообщения (графики).
3.Сделайте то же самое для других вариантов блокирующих обменов (буферизированным, синхронизированным, по готовности).
4.Проанализируйте вариант использования неблокирующих функций и реализуйте его.
5.Реализуйте вариант с отложенными обменами.
6.Отчет обязателен! Детали задания см. в Dsys_MPI_1.ppt