Лекция 2
Стандарт MPI
Двусторонние обмены

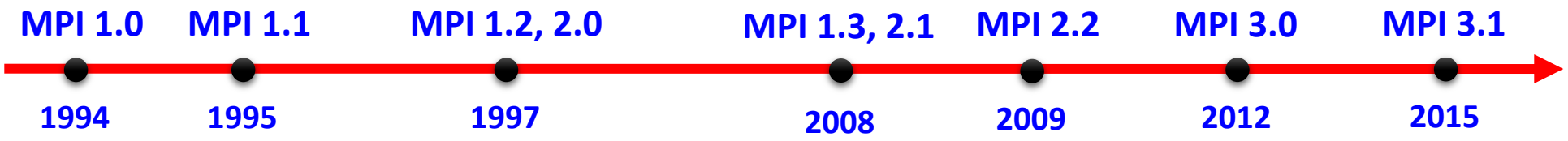
Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com
WWW: www.mkurnosov.net

Курс «Параллельные вычислительные технологии»
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)
Осенний семестр, 2019

Стандарт MPI

- **Message Passing Interface (MPI)** – это стандарт на программный интерфейс коммуникационных библиотек для создания параллельных программ в модели передачи сообщений (message passing)
- Стандарт определяет интерфейс для языков программирования C и Fortran
- Стандарт де-факто для систем с распределенной памятью



http://www.mpi-forum.org

http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

Стандарт MPI

- **Переносимость** программ на уровне исходного кода между разными вычислительными системами (Cray, IBM, NEC, Fujitsu, …)
- **Высокая производительность**
  (MPI – это “ассемблер” в области параллельных вычислений)
- **Масштабируемость** (миллионы процессорных ядер)

Стандарт MPI

3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

MPI_SEND(buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

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

MPI_Send(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
```

Реализации MPI

- **MPICH** (Open source, Argone NL, http://www.mcs.anl.gov/research/projects/mpich2)
- Производные от MPICH:
  MVAPICH (MPICH for InfiniBand), IBM MPI, Cray MPI, Intel MPI, HP MPI, Microsoft MPI

- **Open MPI** (Open source, BSD License, http://www.open-mpi.org)
- Производные от Open MPI: Oracle MPI

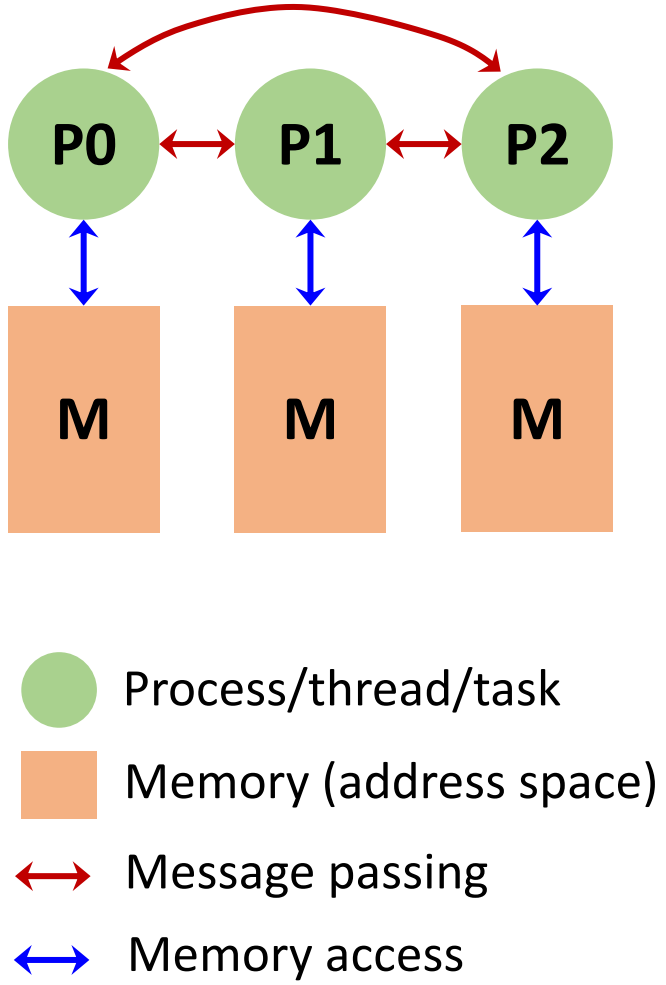
- Высокоуровневые интерфейсы
  - C++: Boost.MPI
  - Java: Open MPI Java Interface, MPI Java, MPJ Express, ParJava
  - C#: MPI.NET, MS-MPI
  - Python: mpi4py, pyMPI

Отличия в реализациях MPI

- **Спектр поддерживаемых архитектур процессоров:**
  Intel, IBM, ARM, Fujitsu, NVIDIA, AMD
- **Типы поддерживаемых коммуникационных технологий/сетей:** InfiniBand, 10 Gigabit Ethernet, Cray Gemini, IBM PERCS/5D torus, Fujitsu Tofu, Myrinet, SCI, …
- **Протоколы дифференцированных обменов двусторонних обменов (Point-to-point):**
  хранение списка процессов, подтверждение передачи (ACK), буферизация сообщений, …
- **Коллективные операции обменов информацией:** коммуникационная сложность алгоритмов, учет структуры вычислительной системы (torus, fat tree, …), неблокирующие коллективные обмены (MPI 3.0, методы хранение collective schedule)
- **Алгоритмы вложения графов программ в структуры вычислительных систем**
  (MPI topology mapping)
- **Возможность выполнения MPI-функций в многопоточной среде и поддержка ускорителей**
  (GPU NVIDIA/AMD, Intel Xeon Phi)

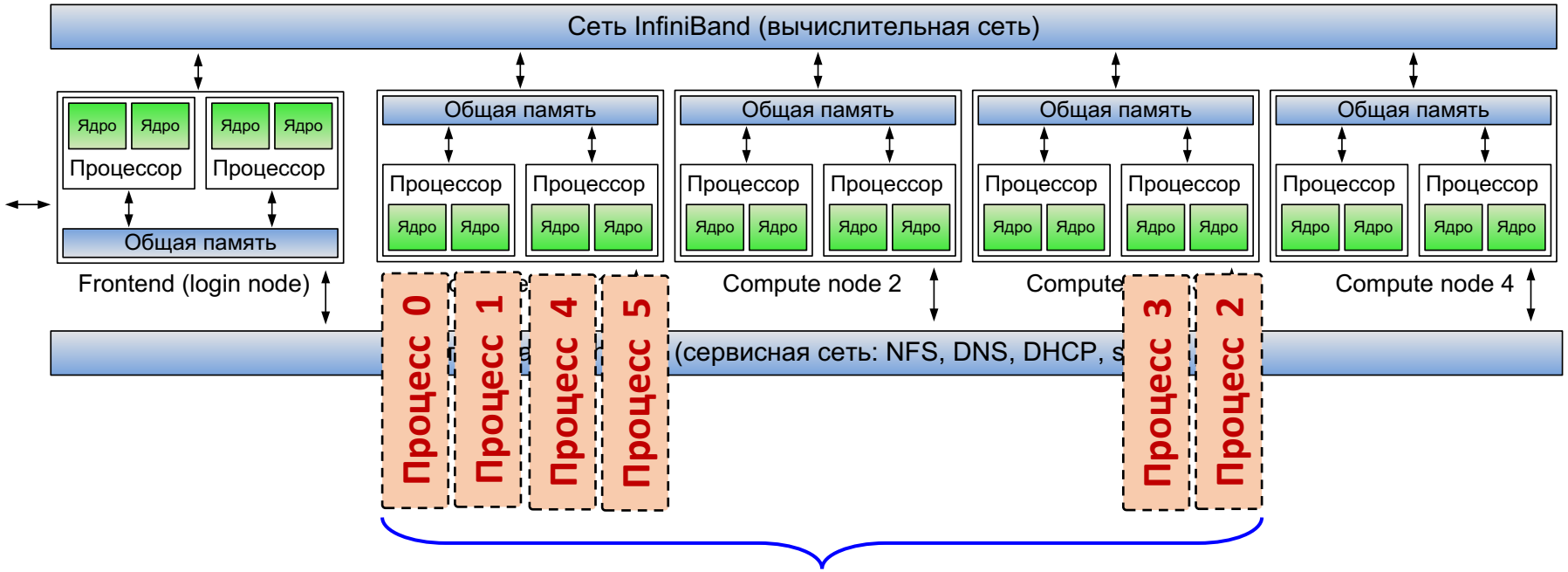
Модель программирования

- Программа состоит из *P* параллельных процессов, которые порождаются при запуске программы (MPI 1) или могут быть динамически созданы во время выполнения (MPI 2)
- Каждый процесс имеет уникальный идентификатор [0, *P* – 1] и изолированное адресное пространство (SPMD)
- Процессы взаимодействуют путем передачи сообщений (message passing)
- Процессы могут образовывать группы для реализации коллективных операций



Понятие коммуникатора (Communicator)

- **Коммуникатор (communicator)** – группа процессов, образующая логическую область для выполнения коллективных операций между процессами
- В рамках коммуникатора процессы имеют номера: 0, 1, …, P – 1
- Все MPI-сообщения должны быть связаны с определенным коммуникатором



Коммуникатор **MPI_COMM_WORLD** включает все процессы

Функции MPI

- Заголовочный файл mpi.h
  #include <mpi.h>
- Функции, типы данных и именованные константы имеют префикс MPI_
- Функции возвращают MPI_SUCCESS или код ошибки
- Результаты возвращаются через аргументы функций

Hello, MPI World!

```
#include <mpi.h>

int main(int argc, char *argv[])
{
    int commsize, rank, len;
    char procname[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(procname, &len);

    printf("Hello, MPI World! Process %d of %d on node %s.\n",
           rank, commsize, procname);

    MPI_Finalize();
    return 0;
}
```

Процессы выполняют один и тот же код (одна программ – Single Program Multiple Data)

Компиляция MPI-программ

```
# Программа на C
$ mpicc -Wall -o hello ./hello.c

# Программа на C++
$ mpicxx -Wall -o hello ./hello.cpp

# Программа на Fortran
$ mpif90 -o hello ./hello.f90
```

Запуск MPI-программ на кластере (TORQUE, кластер Jet)

```
# Формируем паспорт задачи (job-файл)
$ cat task.job
#PBS -N MyTask
#PBS -l nodes=1:ppn=8
#PBS -j oe

cd $PBS_O_WORKDIR

mpiexec ./hello
```

```
# Ставим задачу в очередь
$ qsub ./task.job
882
```

Запуск MPI-программ на кластере (TORQUE)

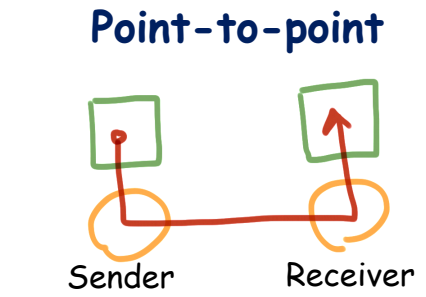
```
# Проверяем состояние задачи в очереди
$ qstat
Job ID          Name             User             Time Use S Queue
--------------- ---------------- ---------------- -------- - -----
876             stream-ep        foobar                 0 Q release
877             xdtask           pdcuser99              0 R debug
882             MyTask           mkurnosov              0 C debug

# Проверяем результат
$ cat ./MyTask.o882
Hello, MPI World! Process 0 of 8 on node cn15.
Hello, MPI World! Process 4 of 8 on node cn15.
Hello, MPI World! Process 1 of 8 on node cn15.
Hello, MPI World! Process 2 of 8 on node cn15.
Hello, MPI World! Process 3 of 8 on node cn15.
Hello, MPI World! Process 5 of 8 on node cn15.
Hello, MPI World! Process 6 of 8 on node cn15.
Hello, MPI World! Process 7 of 8 on node cn15.
```

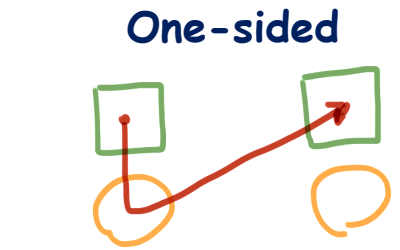
Модель передачи сообщений MPI

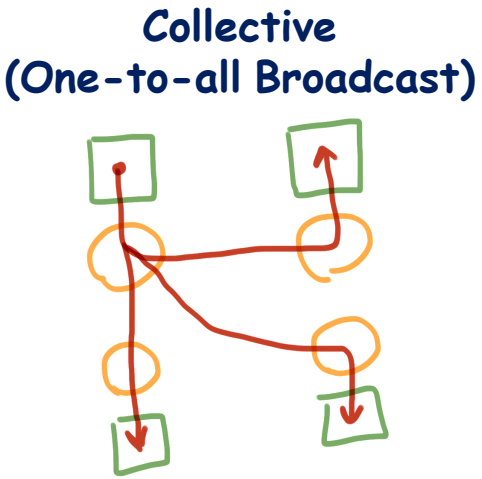
- **Двусторонние обмены (Point-to-point communication)**
  - Один процесс инициирует передачу сообщения (Send), другой его принимает (Receive)
  - Изменение памяти принимающего процесса происходит при его явном участии
  - Обмен совмещен с синхронизацией процессов
- **Односторонние обмены (One-sided communication, Remote memory access)**
  - Только один процесс явно инициирует передачу/прием сообщения из памяти удаленного процесса
  - Синхронизация процессов отсутствует

Виды обменов сообщениями в MPI

- **Двусторонние обмены (Point-to-point communication)** – участвуют два процесса коммуникатора (send, recv)
- **Односторонние обмены (One-sided communication, Remote memory access) –** участвуют два процесса коммуникатора (без синхронизации процессов, put, get)
- **Коллективные обмены (Collective communication)** – участвуют все процессы коммуникатора (one-to-all broadcast, all-to-one gather, all-to-all broadcast)







Структура сообщения (Point-to-point)

- **Данные**
  - Адрес буфера (непрерывный участок памяти)
  - Число элементов в буфере
  - Тип данных элементов в буфере
- **Заголовок (envelope)**
  - Идентификаторы отправителя и получателя
  - Тег сообщения (Tag)
  - Коммуникатор (Communicator)

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

Блокирующие функции Send/Recv

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

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
             MPI_Comm comm, MPI_Status *status)
```

- buf — адрес буфера
- count — число элементов в сообщении
- datatype — тип данных элементов в буфере
- dest — номер процесса-получателя
- source — номер процесса-отправителя или MPI_ANY_SOURCE
- tag — тег сообщения или MPI_ANY_TAG
- comm — идентификатор коммуникатора или MPI_COMM_WORLD
- status — параметры принятого сообщения (содержит поля source, tag)

Соответствие типов данных MPI типам языка C

MPI datatype	C datatype
MPI_CHAR	char (treated as printable character)
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG (as a synonym)	signed long long int
MPI_SIGNED_CHAR	signed char (treated as integral value)
MPI_UNSIGNED_CHAR	unsigned char (treated as integral value)
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t (defined in <stddef.h>) (treated as printable character)

MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	
MPI_PACKED	

Hello, MPI World (2)!

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

int main(int argc, char **argv)
{
    int rank, commsize;

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

    double *buf = malloc(sizeof(*buf) * 100);
    // Work with buffer...

    if (rank == 0) {
        MPI_Send(buf, 100, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(buf, 100, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    free(buf);
    MPI_Finalize();
    return 0;
}
```

Hello, MPI World (3)!

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

int main(int argc, char **argv)
{
    int rank, commsize;

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

    double *buf = malloc(sizeof(*buf) * 100);
    // Work with buffer...

    if (rank == 0) {
        MPI_Send(buf, 100, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
    } else {
        MPI_Recv(buf, 100, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    free(buf);
    MPI_Finalize();
    return 0;
}
```

Как будет вести себя программа если запустить больше двух процессов?

Hello, MPI World (4)!

```
#define NELEMS(x) (sizeof(x) / sizeof((x)[0]))

int main(int argc, char **argv) {
    int rank, commsize, len, tag = 1;
    char host[MPI_MAX_PROCESSOR_NAME], msg[128];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Get_processor_name(host, &len);

    if (rank > 0) {
        snprintf(msg, NELEMS(msg), "Hello, master. I am %d of %d on %s", rank, commsize, host);
        MPI_Send(msg, NELEMS(msg), MPI_CHAR, 0, tag, MPI_COMM_WORLD);
    } else {
        MPI_Status status;
        printf("Hello, World. I am master (%d of %d) on %s\n", rank, commsize, host);
        for (int i = 1; i < commsize; i++) {
            MPI_Recv(msg, NELEMS(msg), MPI_CHAR, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);
            printf("Message from %d: '%s'\n", status.MPI_SOURCE, msg);
        }
    }
    MPI_Finalize(); return 0;
}
```

P – 1 процессов передают свое сообщение процессу 0

Hello, MPI World (2)!

```
Hello, World. I am master (0 of 16) on cn15
Message from 1: 'Hello, master. I am 1 of 16 on cn15'
Message from 2: 'Hello, master. I am 2 of 16 on cn15'
Message from 3: 'Hello, master. I am 3 of 16 on cn15'
Message from 4: 'Hello, master. I am 4 of 16 on cn15'
Message from 5: 'Hello, master. I am 5 of 16 on cn15'
Message from 6: 'Hello, master. I am 6 of 16 on cn15'
Message from 7: 'Hello, master. I am 7 of 16 on cn15'
Message from 14: 'Hello, master. I am 14 of 16 on cn16'
Message from 15: 'Hello, master. I am 15 of 16 on cn16'
Message from 8: 'Hello, master. I am 8 of 16 on cn16'
Message from 9: 'Hello, master. I am 9 of 16 on cn16'
Message from 12: 'Hello, master. I am 12 of 16 on cn16'
Message from 11: 'Hello, master. I am 11 of 16 on cn16'
Message from 10: 'Hello, master. I am 10 of 16 on cn16'
Message from 13: 'Hello, master. I am 13 of 16 on cn16'
```

Семантика двусторонних обменов (Point-to-point)

- Гарантируется сохранение порядка сообщений от каждого процесса-отправителя
- Не гарантируется “справедливость” доставки сообщений от нескольких отправителей

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

Сообщение, отправленное первым send, должно быть получено первым recv

Пример Send/Recv (хотим получить меньше, чем нам отправили)

```
int main(int argc, char **argv) {
    float buf[100];
    // ...

    if (rank == 0) {
        for (int i = 0; i < NELEMS(buf); i++)
            buf[i] = (float)i;
        MPI_Send(buf, NELEMS(buf), MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        // Пытаемся получить меньше, чем нам отправили
        MPI_Status status;
        MPI_Recv(buf, 10, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
    }

    MPI_Finalize();
    return 0;
}
```

Fatal error in MPI_Recv:
Message truncated, error stack:
MPIDI_CH3U_Receive_data_found(284): Message from rank 0 and tag 0 truncated;
400 bytes received but buffer size is 40

Пример Send/Recv (хотим получить больше, чем нам отправили)

```
int main(int argc, char **argv) {
    float buf[100];
    // ...
    if (rank == 0) {
        for (int i = 0; i < NELEMS(buf); i++)
            buf[i] = (float)i;
        MPI_Send(buf, 10, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Status status; // Пытаемся получить больше, чем нам отправили
        MPI_Recv(buf, 100, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
        printf("Master received: ");
        int count;
        MPI_Get_count(&status, MPI_FLOAT, &count);  // count = 10
        for (int i = 0; i < count; i++)
            printf("%f ", buf[i]);
        printf("\n");
    }
    MPI_Finalize();
    return 0;
}
```

Master received: 0.00 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00

Информация о принятом сообщении

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

- Записывает в count число принятых (MPI_Recv) элементов типа datatype

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

- Блокирует выполнение процесса, пока не поступит сообщение (source, tag, comm)
- Информация о сообщении возвращается через параметр status

- Далее, пользователь может создать буфер нужного размера и извлечь сообщение функцией MPI_Recv

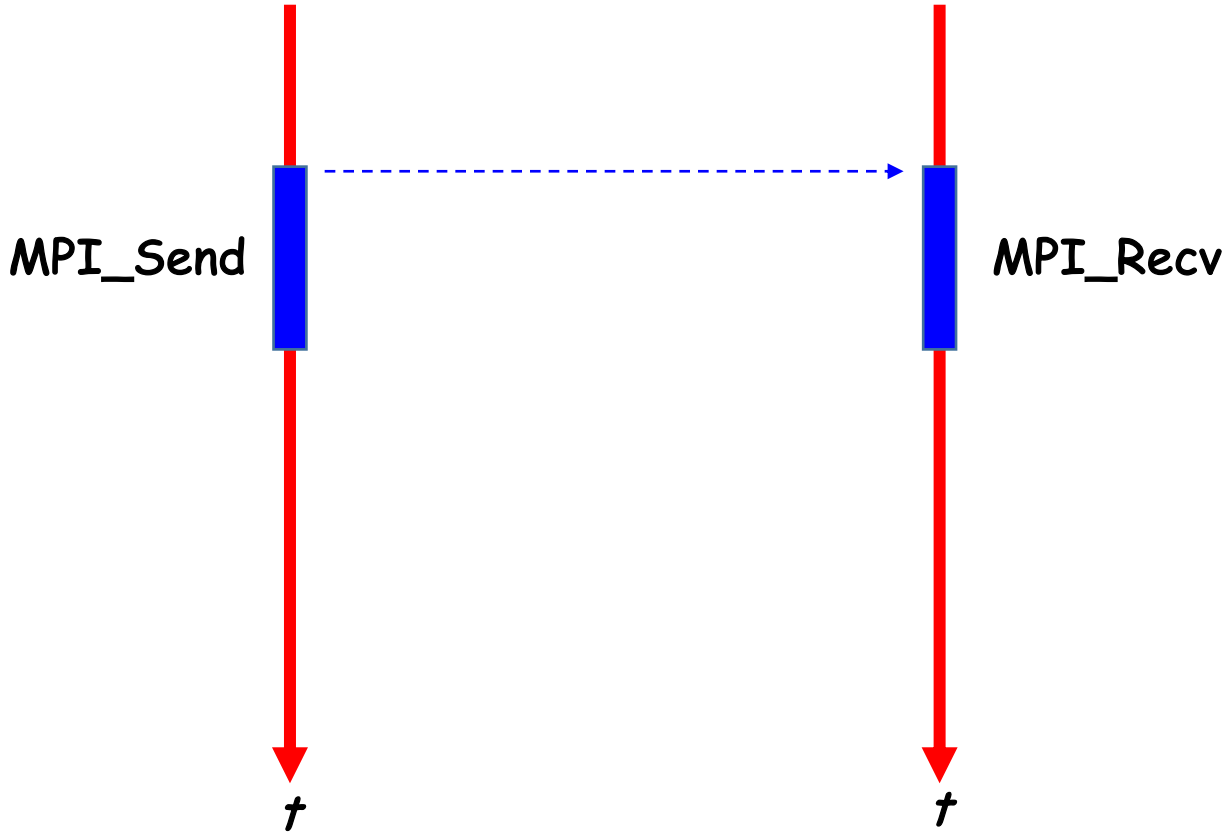
Пример MPI_Probe

```
int main(int argc, char **argv) {

    if (rank == 0) {
        float buf[100];
        MPI_Send(buf, 10, MPI_FLOAT, 2, 0, comm);    // Отправили массив float[10]
    } else if (rank == 1) {
        int buf[32];
        MPI_Send(buf, 6, MPI_INT, 2, 1, comm);       // Отправили массив int[6]
    } else if (rank == 2) {
        MPI_Status status;
        for (int m = 0; m < 2; m++) {
            MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);  // Ждем любого сообщения
            if (status.MPI_TAG == 0) {                              // Определяем тип сообщения
                MPI_Get_count(&status, MPI_FLOAT, &count); // Сколько пришло MPI_FLOAT?
                float *buf = malloc(sizeof(*buf) * count);
                MPI_Recv(buf, count, MPI_FLOAT, status.MPI_SOURCE, status.MPI_TAG, comm, &status);
            } else if (status.MPI_TAG == 1) {
                MPI_Get_count(&status, MPI_INT, &count);   // Сколько пришло MPI_INT?
                int *buf = malloc(sizeof(*buf) * count);
                MPI_Recv(buf, count, MPI_INT, status.MPI_SOURCE, status.MPI_TAG, comm, &status);
            }
        }
    }
```

Состояние процесса после завершения MPI_Send

- Буфер можно повторно использовать, не опасаясь испортить передаваемое сообщение?
- Сообщение покинуло узел процесса-отправителя?
- Сообщение принято процессом-получателем?

Реализация Send/Recv – идеальная ситуация

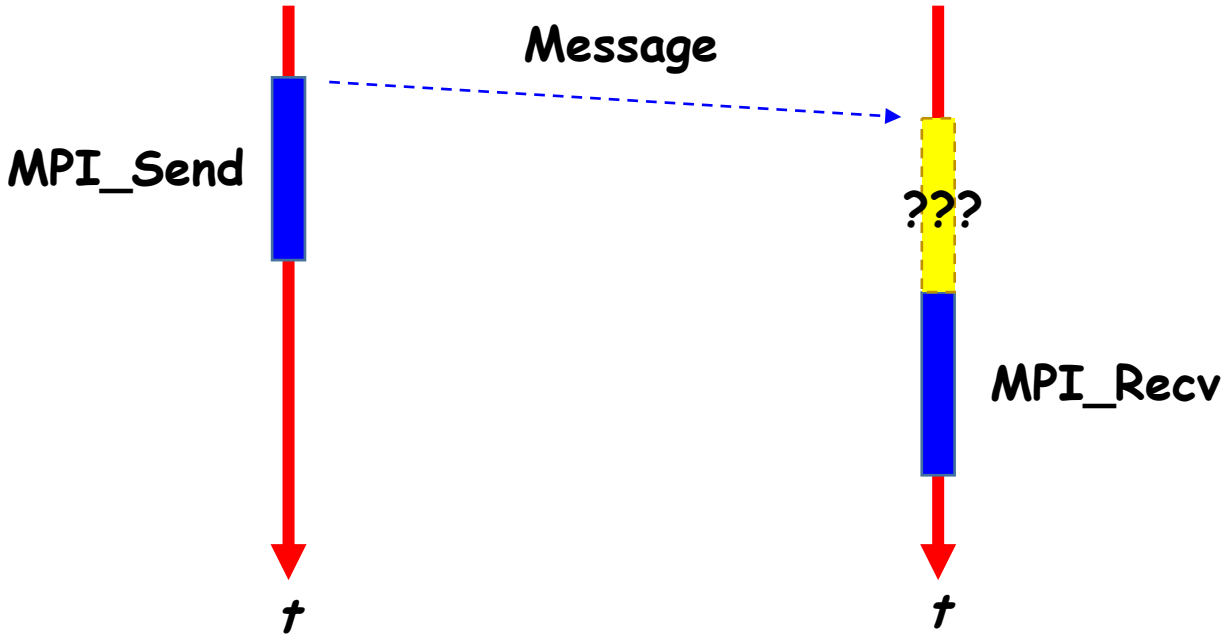


В идеальной ситуации вызов Send должен быть синхронизирован с вызовом Recv (функции должны вызываться в один момент времени)

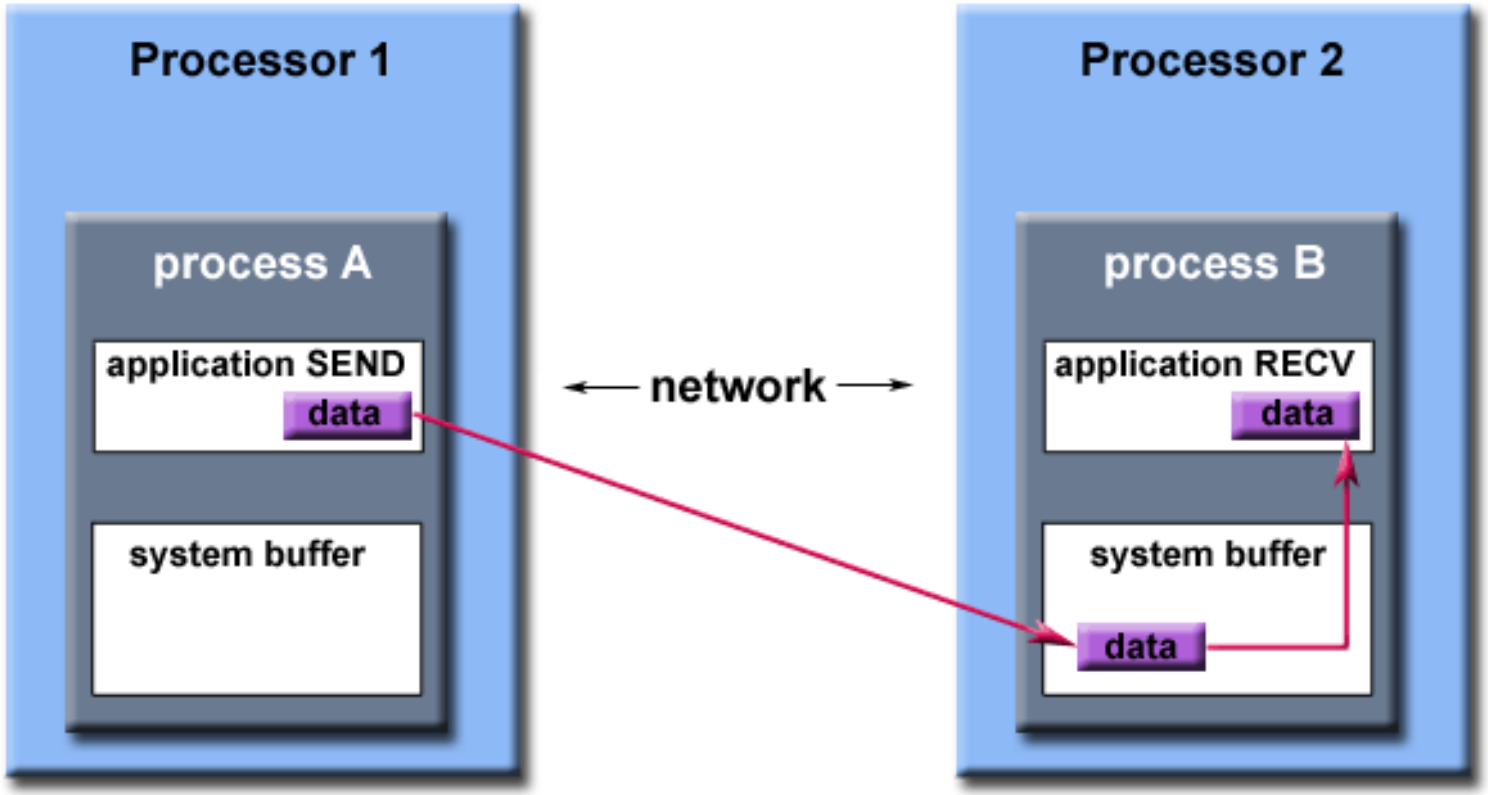
Реализация Send/Recv – реальная ситуация

- Что будет, если Send вызван за 5 секунд до вызова Recv?
  Где будет храниться исходящее сообщение?

- Что будет если несколько процессов одновременно отправляют сообщения процессу-получателю? Как он должен их хранить?



Реализация Send/Recv



Path of a message buffered at the receiving process

https://computing.llnl.gov/tutorials/mpi/

Коммуникационные режимы блокирующих обменов

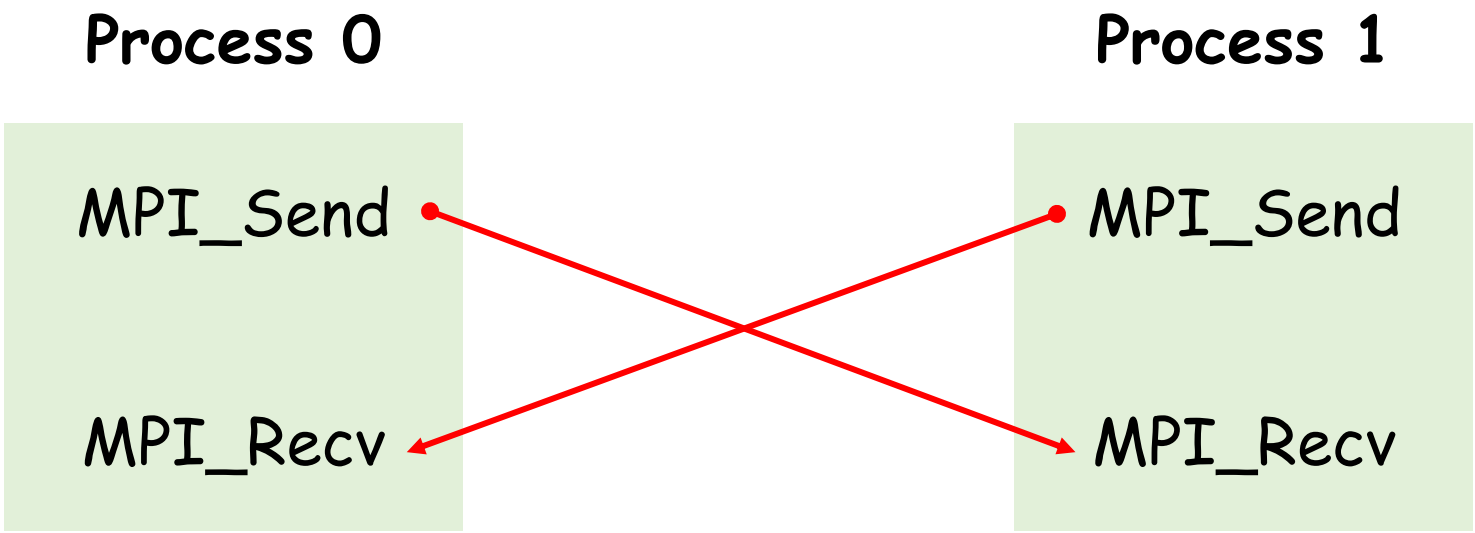
- **Стандартный режим** (Standard communication mode, local/non-local) – реализация определяет будет ли исходящее сообщение буферизовано:
  a) сообщение помещается в буфер, вызов MPI_Send завершается до вызова советующего MPI_Recv
  b) буфер недоступен, вызов MPI_Send не завершится пока не будет вызван соответствующий MPI_Recv (non-local)

- **Режим с буферизацией** (Buffered mode, local) – завершение MPI_Bsend не зависит от того, вызван ли соответствующий MPI_Recv; исходящее сообщение помещается в буфер, вызов MPI_Bsend завершается

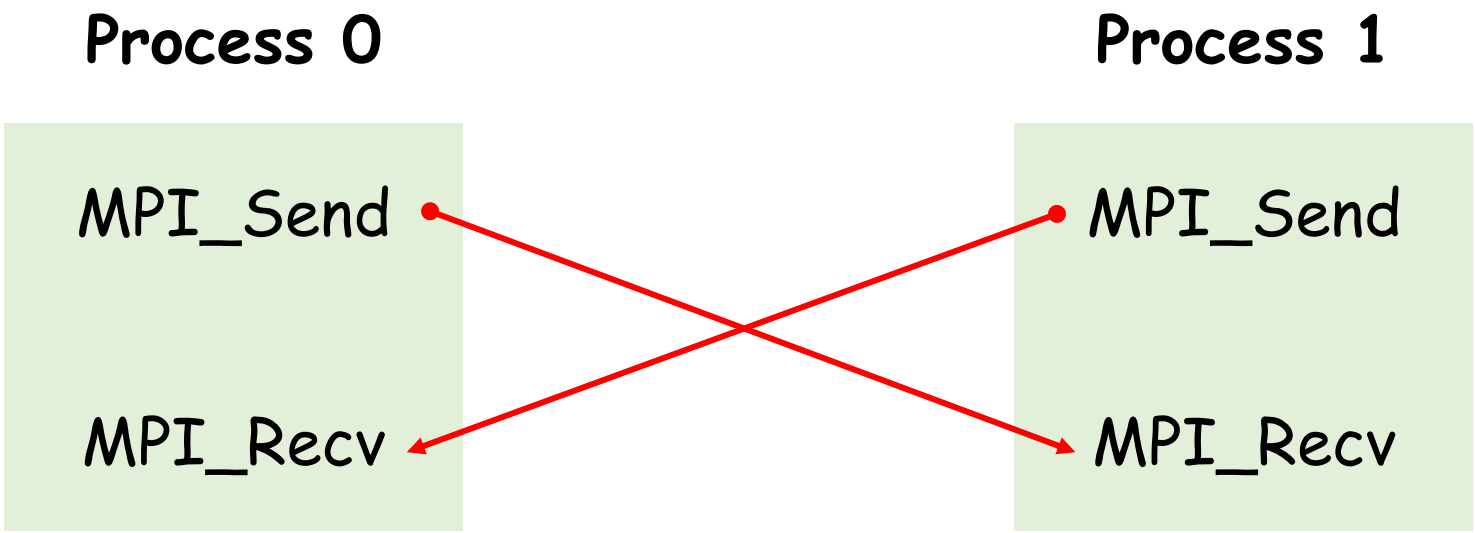
- **Синхронный режим** (Synchronous mode, non-local + synchronization) – вызов MPI_Ssend завершается если соответствующий вызова MPI_Recv начал прием сообщения

- **Режим с передачей по готовности** (Ready communication mode) – вызов MPI_Rsend может начать передачу сообщения если соответствующий MPI_Recv уже вызван (позволяет избежать процедуры “рукопожатия” для сокращения времени обмена)

Взаимная блокировка процессов



Взаимная блокировка процессов



- Поменять порядок операций Send/Recv
- Использовать неблокирующие операции
- Использовать функцию совмещенного обмена MPI_Sendrecv

Совмещение передачи и приема

```
int MPI_Sendrecv(const 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)
```

- Предотвращает возникновение взаимной блокировки при вызове Send/Recv
- Не гарантирует защиту от любых взаимных блокировок!

Неблокирующие функции Send/Recv (Non-blocking)

- Возврат из функции происходит сразу после инициализации процесса передачи/приема
  - Буфер использовать нельзя до завершения операции
- Передача
  - `MPI_Isend(..., MPI_Request *request)`
  - `MPI_Ibsend(..., MPI_Request *request)`
  - `MPI_Issend(..., MPI_Request *request)`
  - `MPI_Irsend(..., MPI_Request *request)`
- Прием
  - `MPI_Irecv(..., MPI_Request *request)`

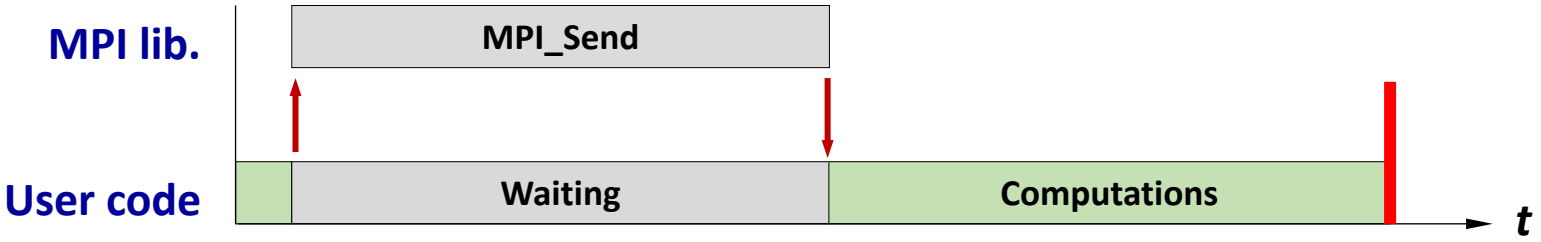
Ожидание завершения неблокирующей операции

- **Блокирующее ожидание завершения операции**
- `int MPI_Wait(MPI_Request *request, MPI_Status *status)`
- `int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, MPI_Status *status)`
- `int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[])`

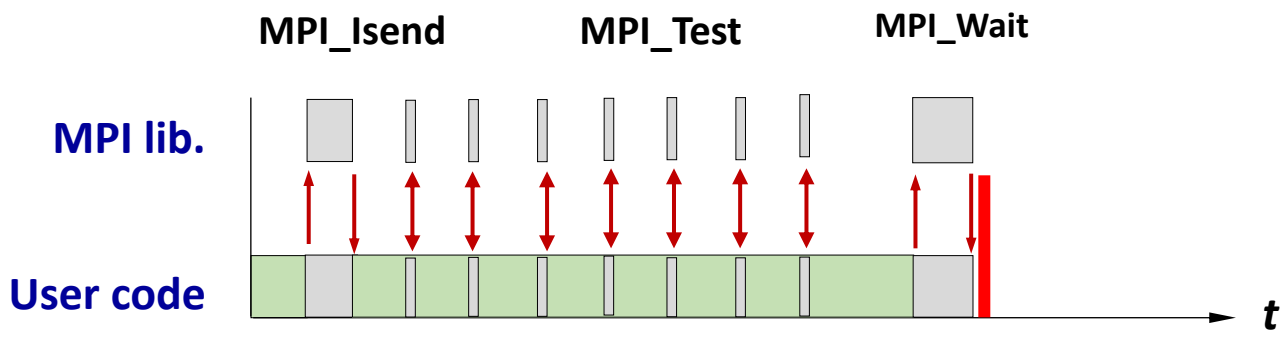
- **Блокирующая проверка состояния операции**
- `int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`
- `int MPI_Testany(int count, MPI_Request array_of_requests[], int *index, int *flag, MPI_Status *status)`
- `int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag, MPI_Status array_of_statuses[])`

Совмещение обменов и вычислений (Overlaping)

Использование блокирующих функций



Использование неблокирующих функций




```
MPI_Isend(buf, count, MPI_INT, 1, 0,
          MPI_COMM_WORLD, &req);

do {
    //
    // Вычисления (не использовать buf)

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

Hello, MPI World (3)!

```
int main(int argc, char **argv) {
    // ...
    char inbuf_prev[50], inbuf_next[50], outbuf_prev[50], outbuf_next[50];
    MPI_Request reqs[4];
    MPI_Status stats[4];

    prev = (rank + commsize - 1) % commsize;
    next = (rank + 1) % commsize;

    snprintf(outbuf_prev, NELEMS(outbuf_prev), "Hello, prev. I am %d of %d on %s",
             rank, commsize, host);
    snprintf(outbuf_next, NELEMS(outbuf_next), "Hello, next. I am %d of %d on %s",
             rank, commsize, host);
    MPI_Isend(outbuf_prev, NELEMS(outbuf_prev), MPI_CHAR, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
    MPI_Isend(outbuf_next, NELEMS(outbuf_next), MPI_CHAR, next, tag2, MPI_COMM_WORLD, &reqs[1]);
    MPI_Irecv(inbuf_prev, NELEMS(inbuf_prev), MPI_CHAR, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
    MPI_Irecv(inbuf_next, NELEMS(inbuf_next), MPI_CHAR, next, tag1, MPI_COMM_WORLD, &reqs[3]);

    MPI_Waitall(4, reqs, stats);
    printf("[%d] Msg from %d (prev): '%s'\n", rank, stats[2].MPI_SOURCE, inbuf_prev);
    printf("[%d] Msg from %d (next): '%s'\n", rank, stats[3].MPI_SOURCE, inbuf_next);
    // ...
}
```

Неблокирующая проверка сообщений

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

- В параметре flag возвращает значение 1, если сообщение с подходящими атрибутами уже может быть принято и 0 в противном случае
- В параметре status возвращает информацию об обнаруженном сообщении (если flag == 1)

Постоянные запросы (persistent)

- Постоянные функции **привязывают** аргументы к дескриптору запроса (persistent request), дальнейшие вызовы операции осуществляется по дескриптору запроса
- Позволяет сократить время выполнения запроса

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

- **Запуск операции (например, в цикле)**
- `int MPI_Start(MPI_Request *request)`
- `int MPI_Startall(int count, MPI_Request array_of_requests[])`

PingPong

```
int main(int argc, char **argv)
{
    int rank, commsize;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int size = 1024 * 1024;
    uint8_t *sbuf = malloc(sizeof(*sbuf) * size);
    uint8_t *rbuf = malloc(sizeof(*rbuf) * size);
    int nruns = 100;

    double t = MPI_Wtime();
    for (int i = 0; i < nruns; i++) {
        if (rank == 0) {
            MPI_Send(sbuf, size, MPI_UINT8_T, 1, 0, MPI_COMM_WORLD);
            MPI_Recv(rbuf, size, MPI_UINT8_T, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        } else if (rank == 1) {
            MPI_Recv(rbuf, size, MPI_UINT8_T, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            MPI_Send(sbuf, size, MPI_UINT8_T, 0, 0, MPI_COMM_WORLD);
        }
    }
    t = (MPI_Wtime() - t) / nruns;
    double mb = 1 << 20;
    printf("Process %d pingpong (%d runs, message %d bytes): time %.6f sec., bandwidth %.2f MiB/sec. (%.2f Mbps)\n",
            rank, nruns, size, t, size / mb / t, size * 8.0 / mb / t);

    free(sbuf);
    free(rbuf);
    MPI_Finalize();
    return 0;
}
```

```
# Кластер Jet – два процесса на разных узла (node=2:ppn=1)
Process 1 pingpong (100 runs, message 1048576 bytes): time 0.018842 sec., bandwidth 53.07 MiB/sec. (424.59 Mbps)
Process 0 pingpong (100 runs, message 1048576 bytes): time 0.018869 sec., bandwidth 53.00 MiB/sec. (423.98 Mbps)
```

```
# Кластер Jet – два процесса на одном узле (node=1:ppn=2)
Process 1 pingpong (100 runs, message 1048576 bytes): time 0.000317 sec., bandwidth 3157.32 MiB/sec. (25258.52 Mbps)
Process 0 pingpong (100 runs, message 1048576 bytes): time 0.000318 sec., bandwidth 3148.14 MiB/sec. (25185.15 Mbps)
```

Домашнее чтение

- Pavan Balaji, Torsten Hoefler. **Advanced Parallel Programming with MPI-1, MPI-2, and MPI-3** // ACM Symposium on Principles and Practice of Parallel Programming, 2013
  http://htor.inf.ethz.ch/teaching/mpi_tutorials/ppopp13/2013-02-24-ppopp-mpi-advanced.pdf

- Rolf Rabenseifner, Georg Hager, Gabriele Jost. **Hybrid MPI and OpenMP Parallel Programming** // Day-long tutorial on Hybrid MPI and OpenMP Parallel Programming from SC13, 2013
  http://openmp.org/wp/sc13-tutorial-hybrid-mpi-and-openmp-parallel-programming/