

# **Введение в параллельные вычисления**

## **Лекция 7. MPI (часть 1)**

КС-40, КС-44  
РХТУ

Преподаватель  
Митричев Иван Игоревич, к.т.н.,  
ассистент кафедры ИКТ

2017

# MPI

Message passing interface -

с англ., дословно - интерфейс передачи сообщений.

# Выбор модели программирования

Системы с общей памятью

- C++11 threads (pthreads)
- OpenMP

Системы с распределенной памятью

- MPI = message passing interface

Гетерогенные системы (графические процессоры)

- CUDA

# Некоторые ссылки

Стандарты MPI <http://www.mpi-forum.org> спецификация MPI-1.1

Обмен данными с использованием MPI.

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

Лекции и семинары

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

Технологии

- [http://parallel.ru/tech/tech\\_dev/mpi.html](http://parallel.ru/tech/tech_dev/mpi.html)

- [http://parallel.ru/tech/tech\\_dev/MPI/examples/](http://parallel.ru/tech/tech_dev/MPI/examples/) (примеры)

Примеры из учебника "Технологии параллельного программирования MPI и OpenMP"

- [http://parallel.ru/tech/tech\\_dev/MPI%20OpenMP/examples/](http://parallel.ru/tech/tech_dev/MPI%20OpenMP/examples/)

# Литература

- Богачёв К.Ю. Основы параллельного программирования. М.: БИНОМ. Лаборатория знаний, 2003.
- Немнюгин С.А. Средства программирования для многопроцессорных вычислительных систем. Спб., 2007.
- Антонов А.С. Параллельное программирование с использованием технологии MPI. М.: Изд-во МГУ, 2004.
- Шпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI. Мн.: БГУ, 2002.

## Учебные материалы по MPI, доступные в Internet

- [Лекция об MPI](#) в курсе "Параллельная обработка данных" (Вл.В.Воеводин).
- [Вычислительный практикум по технологии MPI](#) (А.С.Антонов).
- [Глава об MPI\(link is external\)](#) в книге Яна Фостера "Designing and Building Parallel Programs".
- [Учебные материалы по MPI\(link is external\)](#) на сервере МНРСС.
- [MPI: The Complete Reference\(link is external\)](#). Авторы: Marc Snir, Steve Otto, Steve Huss-Lederman, David Walker, [Jack Dongarra\(link is external\)](#).
- [Tutorial on MPI: The Message Passing Interface\(link is external\)](#). Автор: [Bill Gropp\(link is external\)](#).
- [Writing Message-Passing Parallel Programs with MPI\(link is external\)](#) – учебный курс по MPI, созданный в EPCC (The University of Edinburgh). Авторы: Neil MacDonald, Elspeth Minty, Mario Antonioletti, Joel Malard, Tim Harding, Simon Brown.

# Message Passing Interface



**Message Passing Interface** (MPI, интерфейс передачи сообщений) – программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу.

MPI – наиболее распространённый стандарт интерфейса обмена данными в параллельном программировании, существуют его реализации для большого числа компьютерных платформ. В настоящее время существует большое количество бесплатных и коммерческих реализаций MPI. Существуют реализации для языков Фортран 77/90, Java, Си и Си++.

MPI ориентирован на системы с распределенной памятью, при этом часто затраты на передачу данных велики (бутылочное горлышко).

**Технологии OpenMP и MPI могут использоваться совместно, чтобы оптимально использовать многоядерные системы.**

# Стандарты MPI

Первая версия MPI разрабатывалась в 1993 – 1994 году, и MPI 1 вышла в 1994.

Большинство современных реализаций MPI поддерживают версию 1.1. Стандарт MPI версии 2.0 поддерживается большинством современных реализаций, однако некоторые функции могут быть реализованы не до конца. MPI 3.1 (2015) – последняя версия стандарта.



## Примеры реализаций MPI

- MPICH – самая распространённая свободная реализация, работает на UNIX-системах и Windows NT
- Intel MPI – коммерческая реализация для Windows / Linux
- Oracle HPC ClusterTools – бесплатная реализация для Solaris SPARC/x86 и Linux на основе Open MPI
- MPJ – MPI for Java
- MPJ Express – MPI на Java и др.

# MPI-2

Сегодня и на следующей лекции мы изучаем MPI-1.

MPI-2 содержит 4 коренных отличия от MPI-1:

- 1) Функции для удаленного изменения памяти, а не обмена (MPI\_Put, MPI\_Get, MPI\_Accumulate)
- 2) Динамическое управление процессами (MPI\_Comm\_spawn, MPI\_Comm\_accept / MPI\_Comm\_connect, MPI\_Comm\_join)
- 3) Параллельный ввод-вывод
- 4) Расширение коллективных функций на несколько коммуникаторов



# Функционирование интерфейса

Базовым механизмом связи между MPI процессами является передача и приём сообщений.

Сообщение несёт передаваемые данные и информацию, позволяющую принимающей стороне осуществлять их выборочный приём:

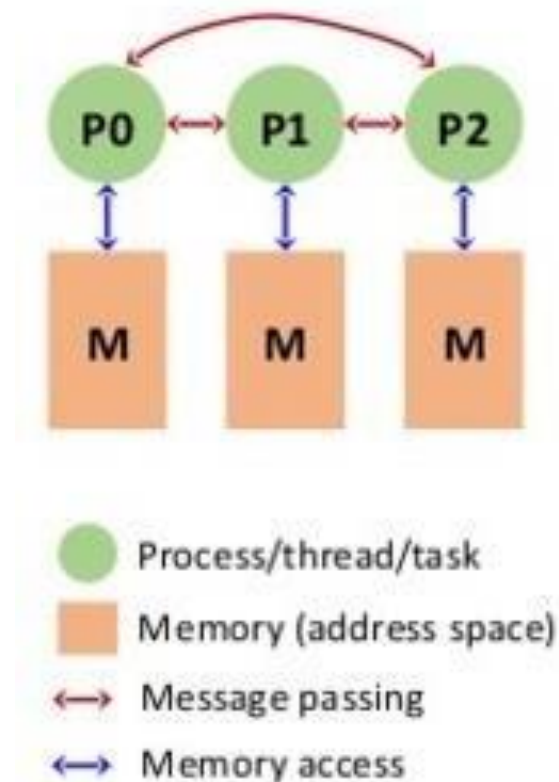
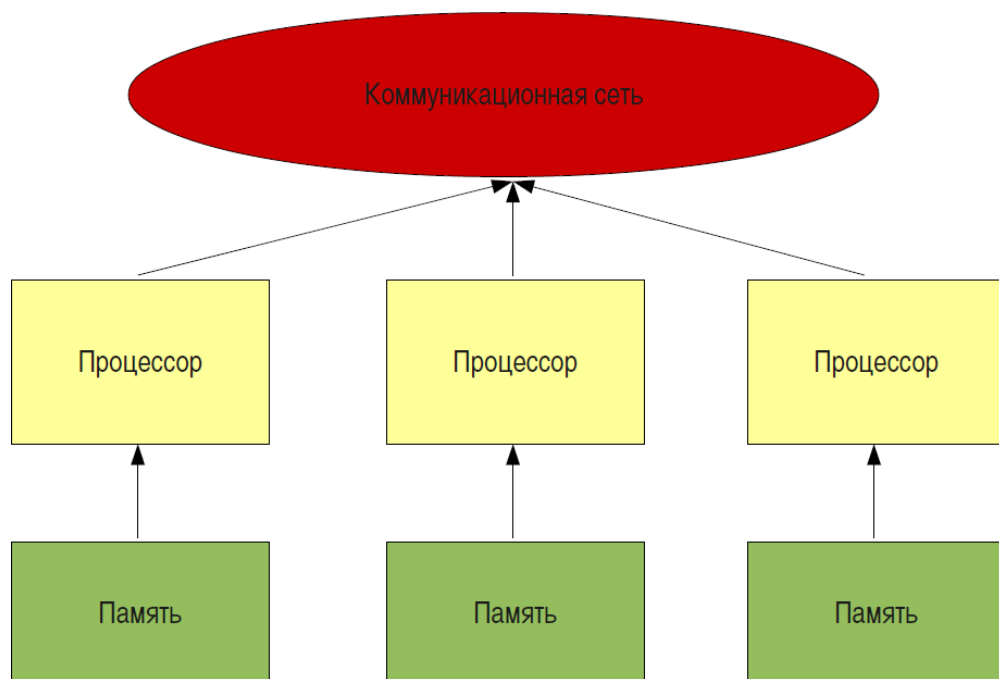
- отправитель – ранг (номер в группе) отправителя сообщения;
- получатель – ранг получателя;
- Признак (attribute) – может использоваться для разделения различных видов сообщений;
- коммуникатор – код группы процессов.

Операции приёма и передачи могут быть блокирующимися и неблокирующимися. Для неблокирующихся операций определены функции проверки готовности и ожидания выполнения.

Другим способом связи является удалённый доступ к памяти (RMA), позволяющий читать и изменять область памяти удалённого процесса. Локальный процесс может переносить область памяти удалённого процесса (внутри указанного процессами окна) в свою память и обратно, а также комбинировать данные, передаваемые в удалённый процесс с имеющимися в его памяти данными (например, путём суммирования). Все операции удалённого доступа к памяти не блокирующие, однако, до и после их выполнения необходимо вызывать блокирующие функции синхронизации.

Полная версия интерфейса содержит описание более 125 процедур и функций.

# Модель передачи сообщений



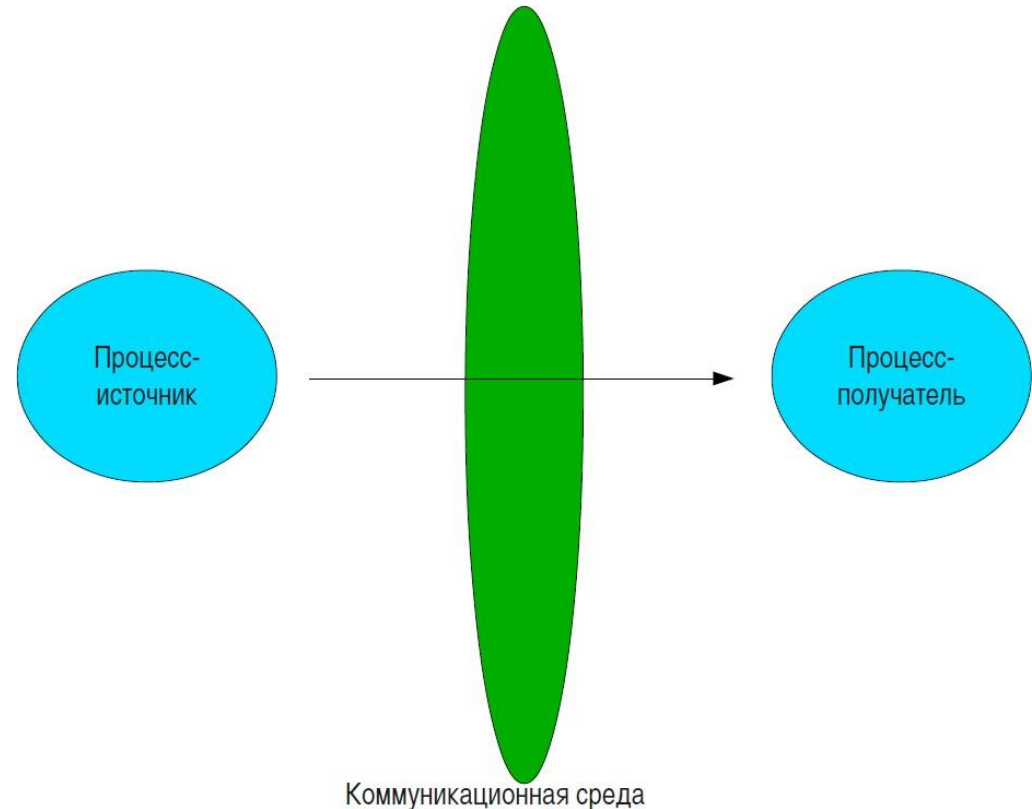
Программа состоит из  $N$  параллельных процессов, которые порождаются при запуске (MPI-1) или м.б. динамически созданы во время выполнения (MPI-2).

# Передача сообщения – способ взаимодействия

**Каждый процесс имеет уникальный идентификатор и изолированное адресное пространство.**

**Общих переменных или данных в MPI нет.**

Процессы могут образовывать группы для реализации коллективных операций обмена информацией.



# Терминология и обозначения

*MPI* – библиотека функций, предназначенная для поддержки работы параллельных процессов в терминах передачи сообщений.

*Номер процесса* – целое неотрицательное число, являющееся уникальным атрибутом каждого процесса.

*Атрибуты сообщения* – номер процесса-отправителя, номер процесса-получателя и идентификатор сообщения.

Заведена структура *MPI\_Status*, содержащая поля:

- *MPI\_Source* (номер процесса отправителя),
- *MPI\_Tag* (идентификатор сообщения),
- *MPI\_Error* (код ошибки); могут быть и добавочные поля.

*Идентификатор сообщения (msgtag)* – атрибут сообщения, являющийся целым неотрицательным числом, лежащим в диапазоне от 0 до 32767.

Процессы объединяются в *группы*, внутри группы все процессы перенумерованы.

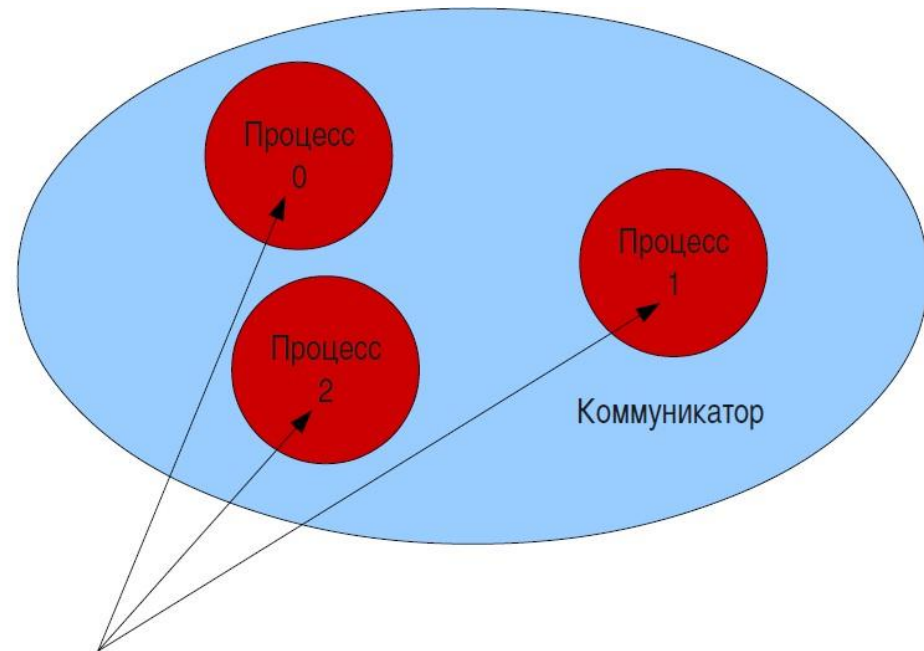
С каждой группой ассоциирован свой *коммуникатор*.

Процесс имеет два основных атрибута – *коммуникатор* и *номер в коммуникаторе*.

# Коммуникатор и ранги

Коммуникатор (communicator) – множество процессов, образующих логическую область для выполнения коллективных операций (обменов информацией и др.)

Состав групп произволен. Группы могут совпадать, входить одна в другую, не пересекаться или пересекаться частично. Процессы могут взаимодействовать только внутри некоторого коммуникатора, сообщения в разных коммуникаторах не пересекаются.



Ранги процессов

Каждый процесс имеет специальный идентификатор – ранг (rank).

Каждый процесс в рамках одного коммуникатора имеет уникальный ранг.

# Локализация и коммуникаторы

Коммуникатор – среда общения.

Коммуникаторы имеют predetermined тип – `MPI_Comm`.

При старте программы все порожденные процессы работают в рамках коммуникатора `MPI_COMM_WORLD`.

Он существует всегда и служит для взаимодействия всех запущенных процессов MPI-программы.

При старте программы имеется коммуникатор `MPI_COMM_SELF`, содержащий только один текущий процесс, и коммуникатор `MPI_COMM_NULL`, не содержащий ни одного процесса.

При пересылке необходимо указать идентификатор группы, внутри которой производится эта пересылка. Все процессы содержатся в группе с predetermined идентификатором `MPI_COMM_WORLD`.

# Установка MPI

Одна из самых распространённых реализаций MPI – MPICH (MPI Chameleon).

Установка в Ubuntu: `sudo apt-get install mpich2`

Компиляция: `mpic++ -o test ./test.cpp`

Запуск: `mpirun -np 4 test`

Официальный сайт: <http://www.mpich.org/>

Скачайте библиотеку MPICH2:

`wget http://www.mpich.org/static/downloads/3.0.4/mpich-3.0.4.tar.gz`

Распаковать tar-архив: `tar xzvf mpich-3.0.4.tar.gz`

# Общий вид MPI программы

Параллельная программа с точки зрения MPI – набор процессов, запущенных на разных вычислительных узлах. Каждый процесс порождается на основе одного и того же программного кода.

```
mpicc -o hello program0.c  
mpiexec -n 5 hello
```

```
#include "mpi.h"  
#include <stdio.h>  
  
int main(int argc, char *argv[])  
{  
    int myid, numprocs;  
    MPI_Init(&argc, &argv);  
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);  
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);  
    fprintf(stdout, "Process %d of %d\n", myid, numprocs);  
    MPI_Finalize();  
    return 0;  
}
```



# Общие процедуры MPI

Общие процедуры необходимы в каждой параллельной программе

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

`MPI_Init` – инициализация параллельной части (реальная инициализация для каждого приложения происходит не более одного раза, если повторно, то действия не выполняются и происходит возврат из подпрограммы).

Все MPI-процедуры могут быть вызваны только после вызова `MPI_Init`.

Процедура возвращает: в случае успешного выполнения – `MPI_SUCCESS`, иначе – код ошибки.

```
int MPI_Finalize( void )
```

`MPI_Finalize` – завершение параллельной части приложения.

К моменту вызова `MPI_Finalize` некоторым процессом все действия, требующие его участия в обмене сообщениями, должны быть завершены. Сложный тип аргументов *MPI\_Init* предусмотрен, чтобы передавать всем процессам аргументы *main*:

```
int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);
    ...
    MPI_Finalize();
}
```

# Общие процедуры MPI

**int MPI\_Comm\_size( MPI\_Comm comm, int\* size)**

Определение общего числа параллельных процессов в группе *comm*.

*comm* – идентификатор группы;    OUT *size* – размер группы.

**int MPI\_Comm\_rank( MPI\_Comm comm, int\* rank)**

Определение номера процесса в группе *comm*.

Значение, возвращаемое по адресу *&rank*, лежит в диапазоне от 0 до *size\_of\_group-1*.

*comm* – идентификатор группы;

OUT *rank* – номер вызывающего процесса в группе *comm*.

**double MPI\_Wtime(void)**

Функция возвращает астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом.

Гарантируется, что этот момент не будет изменен за время существования процесса.

# Пример

Каждый запущенный процесс печатает свой уникальный номер в коммуникаторе `MPI_COMM_WORLD` и число процессов в данном коммуникаторе.

Строка вывода выведена столько раз, сколько процессов порождено при запуске.

Порядок строк не определен.  
(см. код Си)

```
#include "mpi.h"
#include <stdio.h>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int myid, numprocs;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    cout << "    Process " << myid << " of " << numprocs << endl;
    MPI_Finalize();
    return 0;
}
```

```
$ mpirun -np 5 ./hello
```

```
Process 3 of 5
Process 2 of 5
Process 4 of 5
Process 0 of 5
Process 1 of 5
```

# Пример PROSSESSOR\_NAME

```
#define NTIMES 100
int main(int argc, char **argv)
{
    double time_start, time_finish, tick; int rank, i; int len;
    char *name;
    name = (char*)malloc(MPI_MAX_PROCESSOR_NAME*sizeof(char));
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(name, &len);
    tick = MPI_Wtick(); time_start =
    MPI_Wtime(); for (i = 0; i<NTIMES; i++)
        time_finish = MPI_Wtime();
    cout << " processor " << name << " process " << rank << " tick = "; cout << tick << " time
    = " << (time_finish-time_start)/NTIMES << endl; MPI_Finalize();
}
```

**107\_01.cpp**

`MPI_Wtick()` – разрешение таймера на вызвавшем процессоре в сек.

`MPI_Get_processor_name (name, &len)` – name – имя узла, на котором запущен вызвавший процесс; len – количество символов в имени.

# Передача/прием сообщений

Основная операция в MPI – передача сообщений.

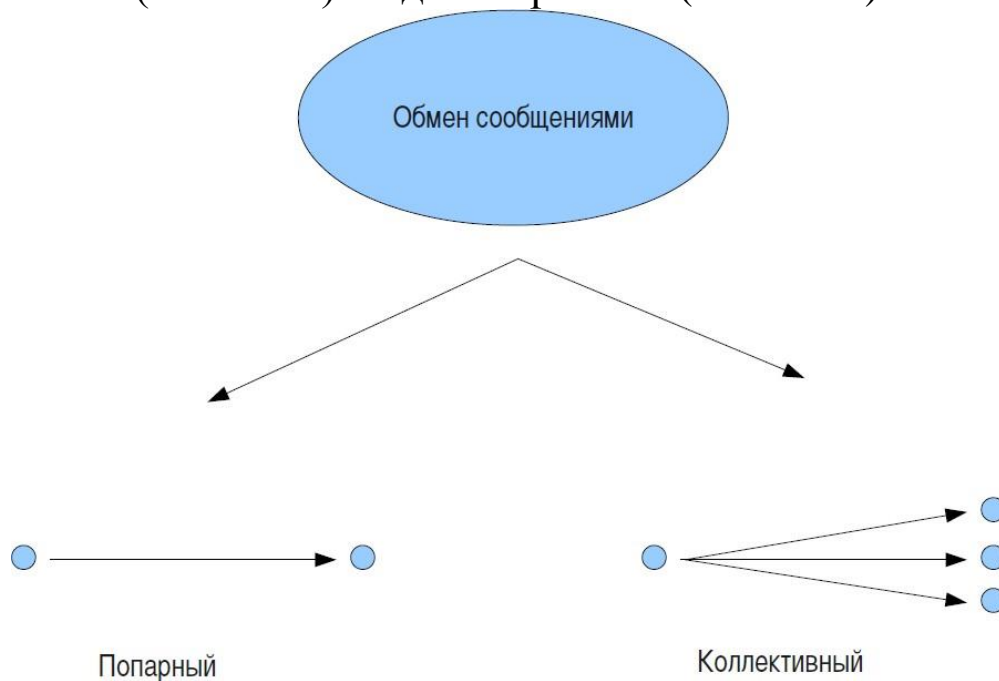
В MPI реализованы практически все основные коммуникационные шаблоны: двухточечные (point-to-point), коллективные (collective) и односторонние (one-sided).

Программы содержат средства порождения и завершения параллельных процессов и средства взаимодействия запущенных процессов между собой.

Процедуры передачи:

Индивидуальные типа точка-точка.

Коллективные – в операцию вовлечены все процессоры коммутатора.



# Обмен сообщениями

Все процедуры делятся на классы: процедуры с блокировкой (с синхронизацией) и без блокировки (асинхронные).

Процедуры обмена с блокировкой приостанавливают работу до выполнения условия. Возврат из асинхронных происходит немедленно после инициализации коммуникационной операции.

Блокирующий

Неблокирующий



Неаккуратность с блокировкой приводит к тупиковым ситуациям.

Использование асинхронных операций требует аккуратного использования массивов данных.

# Составляющие сообщения

1. Блок данных сообщения – void \*
2. Информация о данных сообщения.
  - (a) Тип данных – MPI\_Datatype;
  - (b) Количество данных.
3. Информация о получателе и отправителе сообщения.
  - (a) Коммуникатор – идентификатор группы процессов типа MPI\_Comm, коммуникатор верхнего уровня – MPI\_COMM\_WORLD;
  - (b) Ранг получателя – номер процесса получателя в указанном коммуникаторе;
  - (c) Ранг отправителя – номер процесса отправителя в указанном коммуникатореМожно принимать сообщения от всех отправителей в данном коммуникаторе MPI\_ANY\_SOURCE.
4. Тег сообщения.

Произвольное число типа int. Можно принимать сообщения с определенным тегом, можно с любым – MPI\_ANY\_TAG.

# Соответствие типов данных MPI и C

Тип MPI	Тип C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	unsigned char
MPI_PACKED	

Богачёв К. Ю.

Основы параллельного программирования / К. Ю. Богачёв. — М.: БИНОМ. Лаборатория знаний, 2003. — 342 с., илл.



# Общие процедуры и коммутаторы

```
| int MPI_Comm_size (MPI_Comm comm, int *size); |
```

коммуникатор

указатель на результат

← получить количество  
процессов в коммуникаторе

```
| int p;  
| MPI_Comm_size(MPI_COMM_WORLD, &p); |
```

← получить количество  
всех процессов

```
| int MPI_Comm_rank(MPI_Comm comm, int *rank); |
```

← получить ранг (номер)  
процесса в группе  
(коммуникаторе)

```
| int my_rank;  
| int MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); |
```

← в глобальном  
коммуникаторе

# Режимы коммуникации (обмена сообщениями)

Определяют, когда завершается (а также, и когда начинается) прием и отправка сообщений.

## Отправка сообщения

1. Стандартный (standard) – сообщения могут буферизироваться или нет в зависимости от наличия свободного места в буфере, могут не буферизироваться с целью оптимизации производительности. Является нелокальным, то есть использует информацию о наличии ожидающих запросов на получение сообщения (receive). Без префикса
2. Буферизованный (buffered) – сообщения обязательно буферизуются. Является локальным, то есть вызов завершается независимо от того, есть ли ожидающие запросы на получение сообщения (receive). Префикс R у функций (MPI\_RSEND)
3. Синхронный (synchronous) – отправка завершается только тогда, когда есть соответствующий запрос на получение. Префикс S (MPI\_SSEND)
4. По готовности (ready) – отправка сообщения может быть начата только, когда есть соответствующий запрос на получение. Префикс R (MPI\_RSEND)

Прием сообщения – один режим. Когда сообщение поступает в буфер, прием завершается

# Блокирующая посылка сообщения

Операции точка-точка с синхронизацией (один – отправитель, другой – получатель)

**int MPI\_Send(void\* buf, int count, MPI\_Datatype datatype, int dest, int msgtag, MPI\_Comm comm)**

*buf* – адрес начала буфера посылки сообщения;

*count* – число передаваемых элементов в сообщении;

*datatype* – тип передаваемых элементов;

*dest* – номер процесса-получателя;

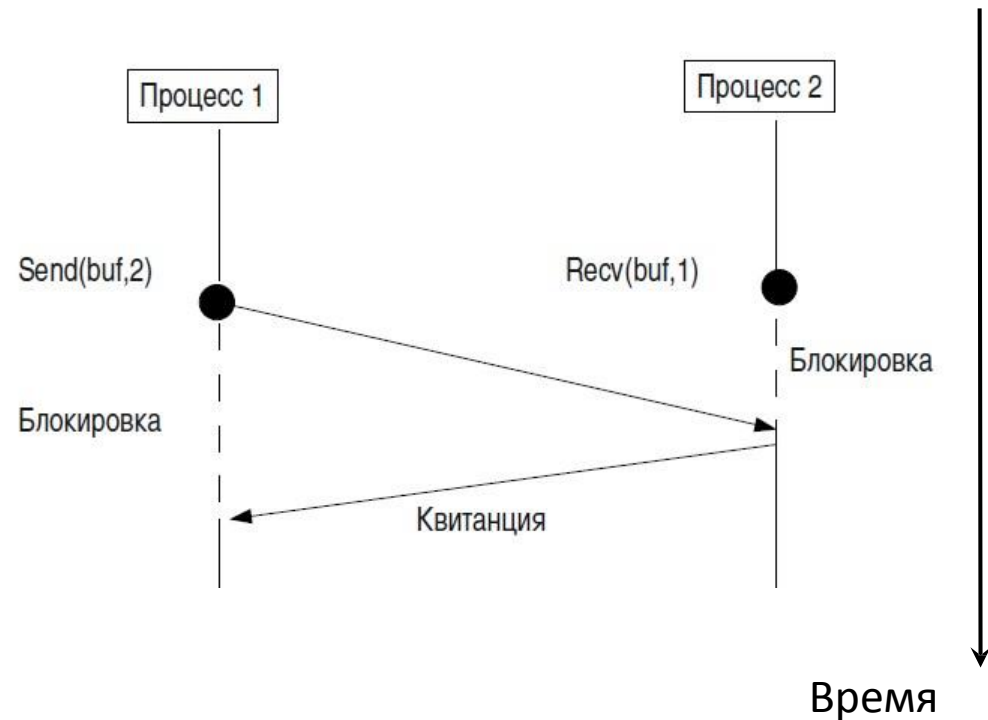
*msgtag* – идентификатор сообщения;

*comm* – идентификатор группы.

Блокирующая посылка сообщения с идентификатором *msgtag*, состоящего из *count* элементов типа *datatype*, процессу с номером *dest*. Все элементы сообщения расположены подряд в буфере *buf*. Значение *count* может быть нулем. Тип передаваемых элементов *datatype* должен указываться с помощью predefined констант типа **MPI\_Datatype**. Разрешается передавать сообщение самому себе, что может привести к тупиковым ситуациям.

# Блокирующая посылка сообщения

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы посредством копирования в промежуточный буфер или непосредственной передачи процессу *dest*.



# Прием сообщения

**int MPI\_Recv(void\* buf, int count, MPI\_Datatype datatype, int source, int msgtag, MPI\_Comm comm, MPI\_Status \*status)**

OUT *buf* – адрес начала буфера приема сообщения;

*count* – максимальное число элементов в принимаемом сообщении;

*datatype* – тип элементов принимаемого сообщения;

*source* – номер процесса-отправителя;

*msgtag* – идентификатор принимаемого сообщения;

OUT *status* – параметры принятого сообщения.

Прием сообщения с идентификатором *msgtag* от процесса *source* с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения *count*. Если число принятых элементов меньше значения *count*, то гарантируется, что в буфере *buf* изменятся только элементы, соответствующие элементам принятого сообщения. Чтобы узнать точное число элементов в сообщении – функция *MPI\_Probe*.

Блокировка гарантирует, что после возврата из подпрограммы все элементы сообщения приняты и расположены в буфере *buf*.

Если процесс посылает два сообщения другому процессу и оба эти сообщения соответствуют одному и тому же вызову *MPI\_Recv*, то первым будет принято то сообщение, которое было отправлено раньше.

# Обмен сообщениями для двух процессов

Нулевой процесс  
посылает  
сообщение  
процессу с номером  
1 и ждет от него  
ответа.

Программа  
запущена на 3-х  
процессах,  
выполняют  
пересылки 0 и 1.

```
process 2 a = 0 b = 0
process 0 a = 2 b = 1
process 1 a = 2 b = 1
```

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

int main(int argc, char **argv) {
    int rank; float a, b;      MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    a = 0.0;      b = 0.0;
    if ( rank == 0 ) {
        b = 1.0;
        MPI_Send(&b, 1, MPI_INT, 1, 5, MPI_COMM_WORLD);
        MPI_Recv(&a, 1, MPI_INT, 1, 5, MPI_COMM_WORLD, &status);
    }
    if ( rank == 1 ) {
        a = 2.0;
        MPI_Recv(&b, 1, MPI_FLOAT, 0, 5, MPI_COMM_WORLD, &status);
        MPI_Send(&a, 1, MPI_FLOAT, 0, 5, MPI_COMM_WORLD);
    }
    cout << " process " << rank << " a = " << a << " b = " << b << endl;
    MPI_Finalize();
}
```

107\_02.cpp