

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/examples/ (примеры)

Примеры из учебника "Технологии параллельного программирования MPI и OpenMP"

- 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:

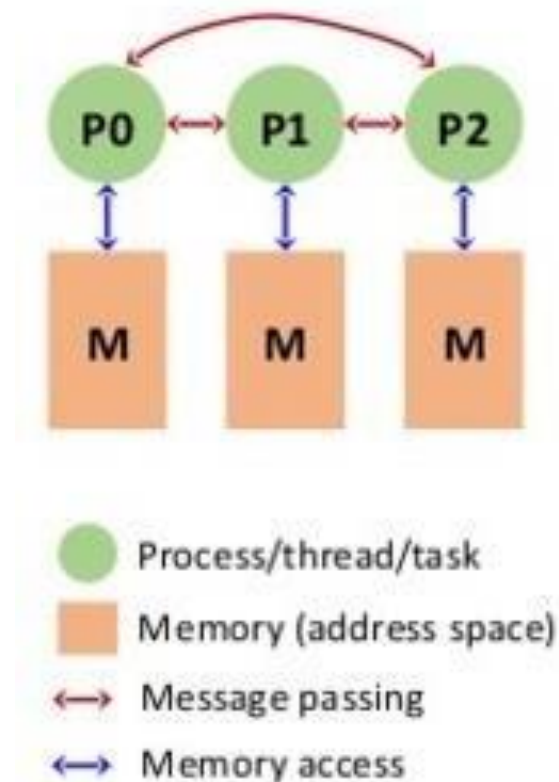
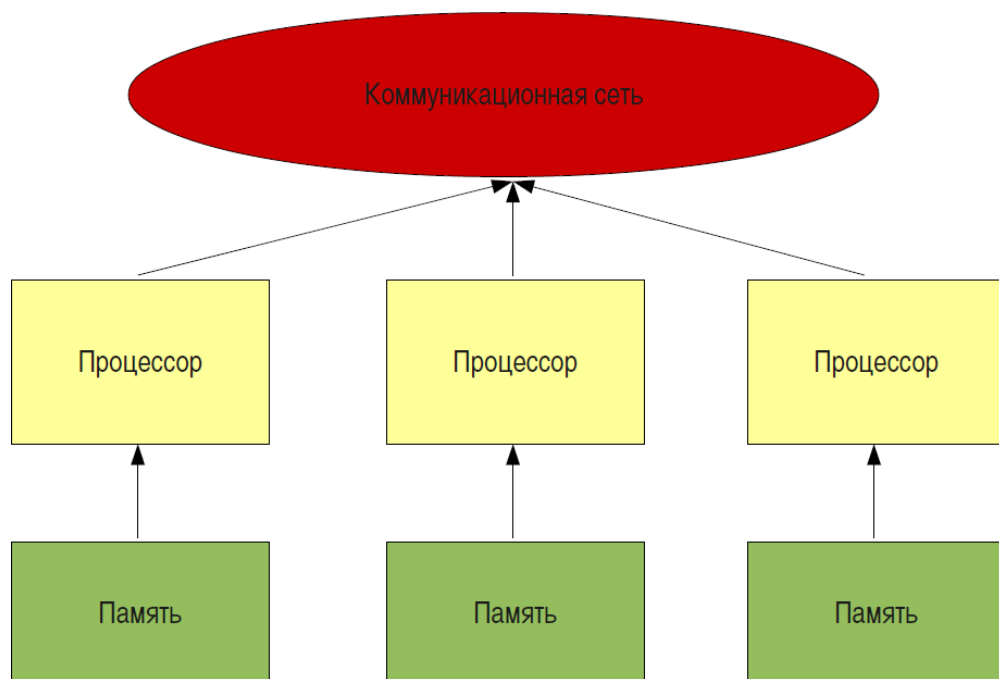
- отправитель – ранг (номер в группе) отправителя сообщения;
- получатель – ранг получателя;
- Признак (attribute) – может использоваться для разделения различных видов сообщений;
- коммуникатор – код группы процессов.

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



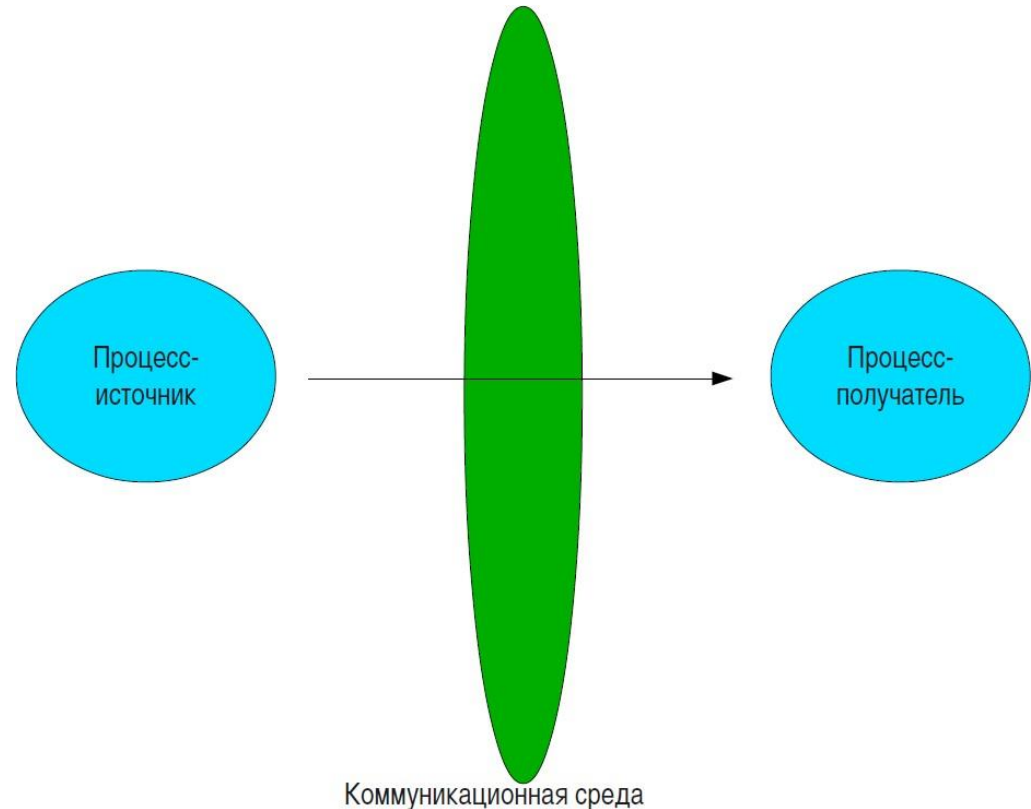
Программа состоит из N параллельных процессов, которые порождаются при запуске (MPI-1) или м.б. динамически созданы во время выполнения (MPI-2).

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.

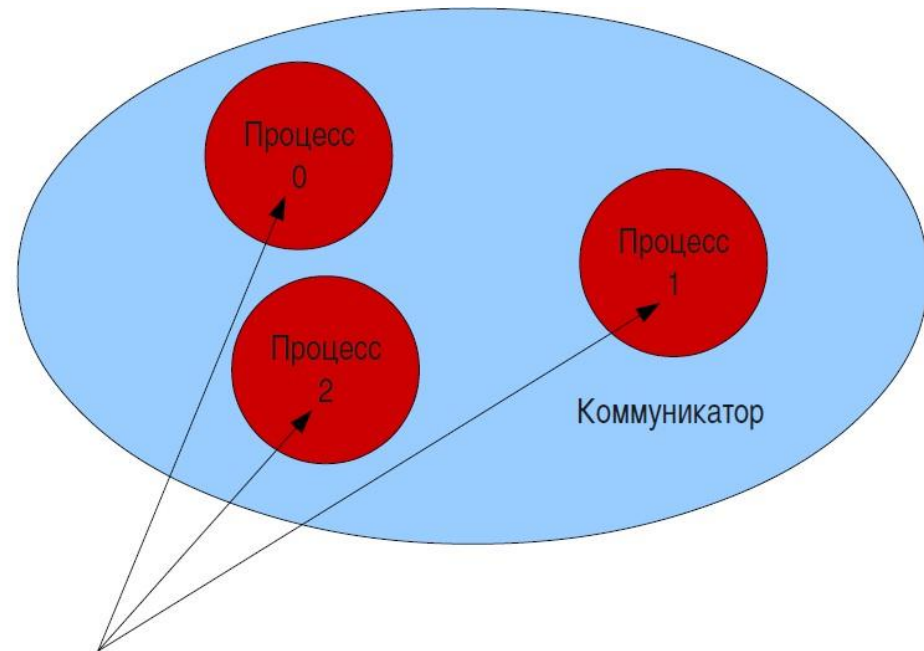
С каждой группой ассоциирован свой *коммуникатор*.

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 предопределенный 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 предопределенным 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();
}
```

`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.

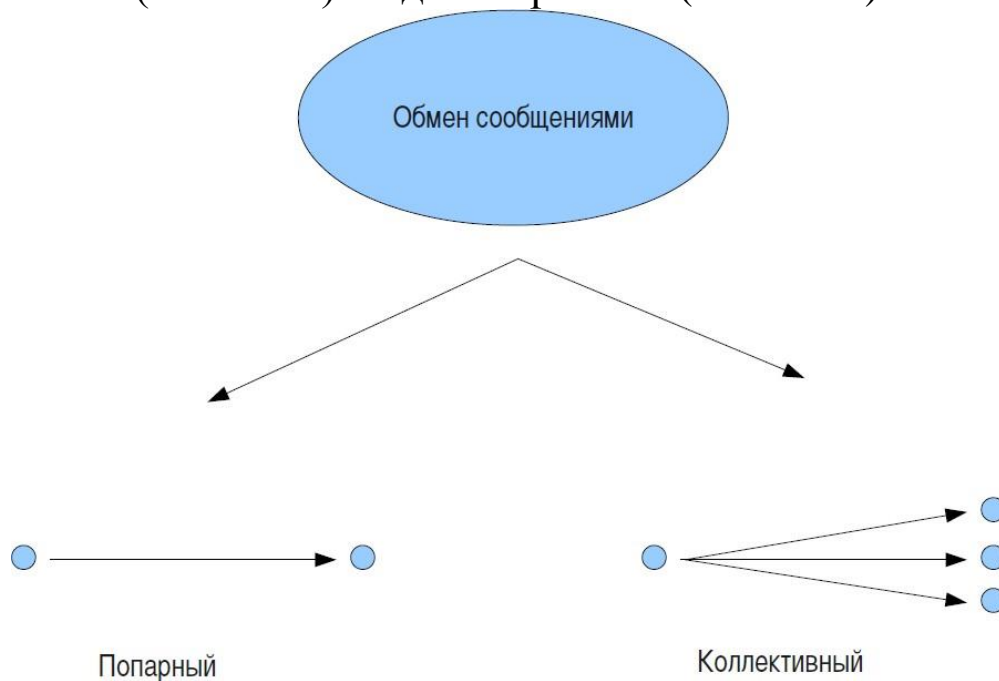
В MPI реализованы практически все основные коммуникационные шаблоны: двухточечные (point-to-point), коллективные (collective) и односторонние (one-sided).

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

Процедуры передачи:

Индивидуальные типа точка-точка.

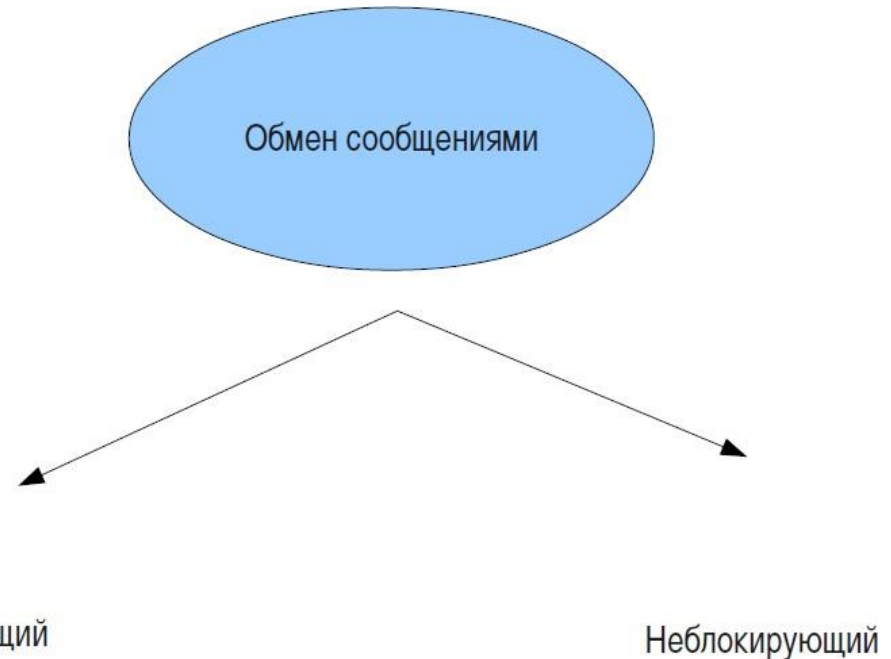
Коллективные – в операцию вовлечены все процессоры коммутатора.



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.



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); |
```

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;

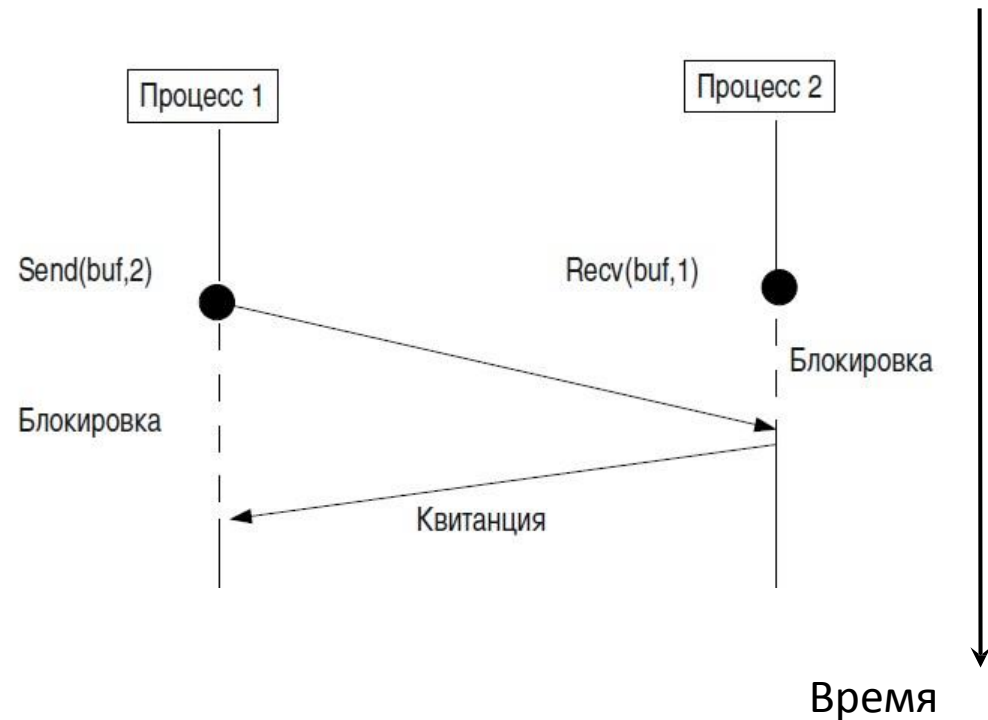
comm – идентификатор группы.

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*.

Возврат из подпрограммы *MPI_Send* не означает ни того, что сообщение уже передано процессу *dest*, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший *MPI_Send*.



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 – подпрограмма *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();
}
```

Пример MPI_Send/MPI_Recv

Каждый процесс с четным номером посылает сообщение соседу с номером на 1 большим.

Поставлена проверка для процесса с максимальным номером он не посылает сообщение несуществующему процессу.

Значение b изменяется только на процессах с нечетными номерами.

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

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv) {
    int size, rank, a, b;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    a = rank;      b = -1;
    if ((rank%2) == 0) {
        if (rank < size - 1)
            MPI_Send(&a, 1, MPI_INT, rank+1, 5, MPI_COMM_WORLD);
    }
    else
        MPI_Recv(&b, 1, MPI_INT, rank-1, 5, MPI_COMM_WORLD, &status);
    cout << " process " << rank << " a = " << a << " b = " << b << endl;
    MPI_Finalize();
}
```

Выборочный прием сообщения

При приеме сообщений вместо аргументов SOURCE и MSGTAG можно использовать predefined константы:

- MPI_ANY_SOURCE – признак, что подходит сообщение от любого процесса;
- MPI_ANY_TAG – признак, что подходит сообщение с любым идентификатором.

При одновременном использовании будет принято сообщение с любым идентификатором от любого процесса.

О статусе сообщения

Атрибуты принятого сообщения можно определить по элементам массива status.

Параметр status – структура predefined типа MPI_Status с полями MPI_SOURCE (реальный ранг сообщения), MPI_TAG (реальный тег) и MPI_ERROR (код ошибки).

Номер принимающего процесса требуется указать явно.

Если один процесс посылает два сообщения, соответствующие одному MPI_Recv, другому процессу, то первым принимается первое сообщение.

Если сообщение отправлено разными процессами, то порядок получения не определен.

Прием/передача сообщений без блокировки

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

OUT *request* – идентификатор асинхронной передачи

Передача аналогична *MPI_Send*, но возврат из подпрограммы сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере *buf*. Нельзя повторно использовать буфер без получения дополнительной информации о завершении данной посылки.

Сообщение, отправленное любой из процедур *MPI_Send* и *MPI_Isend*, может быть принято любой из процедур *MPI_Recv* и *MPI_Irecv*.

Предусмотрены три дополнительных варианта, подобные модификациям процедуры *MPI_Send*.

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

OUT *request* – идентификатор асинхронного приема сообщения

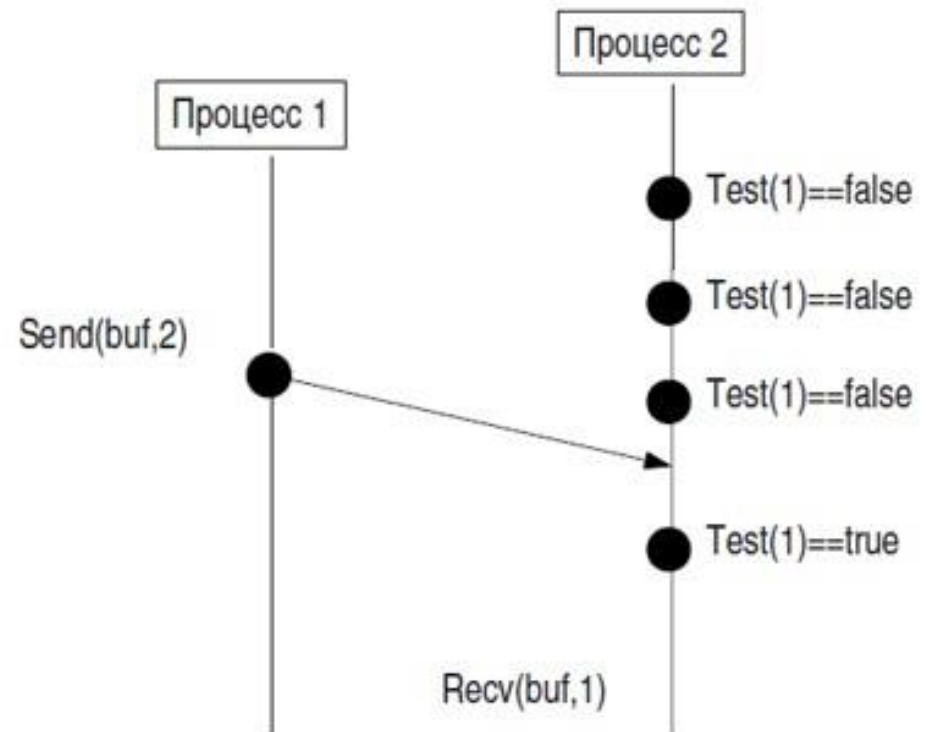
Прием сообщения аналогичный *MPI_Recv*, однако возврат из подпрограммы происходит сразу после инициализации процесса приема без ожидания получения сообщения в буфере *buf*.

Окончание процесса приема можно определить с помощью параметра *request* и процедур *MPI_Wait* и *MPI_Test*.

Неблокирующий обмен сообщениями

Асинхронная передача данных.
Возврат сразу после вызова без
остановки работы процессов.

Для завершения асинхронного
обмена требуются
дополнительные процедуры, с
целью использования буфера.
До завершения неблокирующей
операции нельзя записывать в
используемый массив данных.



Завершение асинхронного обмена

int MPI_Wait(MPI_Request *request, MPI_Status *status)

request – идентификатор асинхронного приема или передачи;

OUT *status* – параметры сообщения.

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

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

requests – массив идентификаторов асинхронного приема или передачи;

OUT *statuses* – параметры сообщений.

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

Неблокирующий обмен - пример

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

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

Top50

N	Место	Кол-во CPU/ядер	Архитектура (тип процессора / сеть)	Производительность (Tflop/s)		Разработчик
				Linpack	Пиковая	
1	Москва Московский государственный университет имени М.В.Ломоносова 2016 г.	1472/42688	узлов: 1472 (Xeon E5-2697v3 [Acc: Tesla K40M] 2.6 GHz 64 GB RAM) сеть: Infiniband FDR/Infiniband FDR/Gigabit Ethernet	2,102.00	2,962.30	Т-Платформы
2	Москва Московский государственный университет имени М.В.Ломоносова 2012 г.	12422/82468	узлов: 4160 (2xXeon 5570 2.93 GHz 12 GB RAM) узлов: 777 (2xXeon E5630 [Acc: 2xTesla X2070] 2.53 GHz 12 GB RAM) узлов: 640 (2xXeon 5670 2.93 GHz 24 GB RAM) узлов: 288 (2xXeon E5630 [Acc: 2xTesla X2070] 2.53 GHz 24 GB RAM) узлов: 260 (2xXeon 5570 2.93 GHz 24 GB RAM) узлов: 40 (2xXeon 5670 2.93 GHz 48 GB RAM) узлов: 30 (2xPowerXCell 8i 3.2 GHz 16 GB RAM) узлов: 4 (4xXeon E7650 2.26 GHz 512 GB RAM) сеть: Infiniband QDR/Gigabit Ethernet/Gigabit Ethernet	901.90	1,700.21	Т-Платформы
3	Санкт-Петербург Суперкомпьютерный центр Санкт- Петербургский политехнический университет 2017 г.	1468/20552	узлов: 623 (2xXeon E5-2697v3 2.6 GHz 64 GB RAM) узлов: 56 (2xXeon E5-2697v3 [Acc: 2x NVIDIA K40] 2.6 GHz 64 GB RAM) узлов: 36 (2xXeon E5-2697v3 2.6 GHz 128 GB RAM) узлов: 8 (2xXeon E5-2697v3 [Acc: NVIDIA K1] 2.6 GHz 128 GB RAM) узлов: 8 (2xXeon E5-2697v3 [Acc: NVIDIA K2] 2.6 GHz 128 GB RAM) узлов: 3 (2xXeon E5-2697v3 2.6 GHz 256 GB RAM) сеть: Infiniband FDR/Gigabit Ethernet/Gigabit Ethernet	715.94	1,015.10	Группа компаний РСК
4	Москва МСЦ РАН 2016 г.	416/28704	узлов: 208 (2xXeon E5-2690 [Acc: 2x Xeon Phi 7110X] 2.9 GHz 80 GB RAM) сеть: Infiniband FDR/Gigabit Ethernet/Fast Ethernet	383.21	523.83	Группа компаний РСК
5	Москва Суперкомпьютерный вычислительный комплекс НИЦ "Курчатовский институт" 2015 г.	296/10064	узлов: 148 (2xXeon E5-2650v2 [Acc: 2x Tesla K80] 2.6 GHz 128 GB RAM) сеть: Infiniband FDR/Gigabit Ethernet/Gigabit Ethernet	381.40	601.00	Т-Платформы
6	Москва Центр обработки данных НИЦ "Курчатовский институт" 2015 г.	774/11082	узлов: 364 (2xXeon E5-2680v3 2.5 GHz 128 GB RAM) узлов: 23 (2xXeon E5-2680v3 [Acc: 3x Nvidia K80] 2.5 GHz 128 GB RAM) сеть: Infiniband FDR/Gigabit Ethernet/Gigabit Ethernet	374.13	500.55	SuperMicro, Борлас