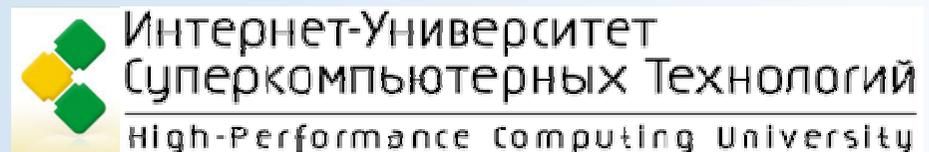


# Основы параллельного программирования с использованием MPI

## Лекция 3

*Немнюгин Сергей Андреевич*  
Санкт-Петербургский государственный университет  
кафедра вычислительной физики

[snemnyugin@mail.ru](mailto:snemnyugin@mail.ru)



# Лекция 3

## **Аннотация**

В этой лекции дается краткий обзор некоторых реализаций МРІ. Объясняется роль демона `mpd`. Вводятся основные понятия и терминология. Приводятся типовые схемы организации параллельных МРІ-программ, их структура. Рассматриваются привязки к языкам программирования С и Fortran, а также способы компиляции и запуска МРІ-программ.

## План лекции

- Основные понятия, терминология.
- Реализации MPI.
- Компиляция и запуск MPI-программ в MPI-1. Файлы конфигурации.
- Компиляция и запуск MPI-программ в MPI-2. Демон mrd. Файлы конфигурации.
- Структура MPI-программы.
- Простейшая MPI-программа.

# **Основные понятия, терминология**

**Сообщение** содержит пересылаемые данные и служебную информацию:

- идентификатор процесса-отправителя сообщения (*ранг* процесса);
- адрес, по которому размещаются пересылаемые данные процесса-отправителя;
- тип пересылаемых данных;
- количество данных (размер буфера сообщения для того, чтобы принять сообщение, процесс должен отвести для него достаточный объем оперативной памяти);
- идентификатор процесса, который должен получить сообщение;
- адрес, по которому должны быть размещены данные процессом-получателем;
- идентификатор коммутатора, описывающего область взаимодействия, внутри которой происходит обмен.

**Ранг** источника дает возможность различать сообщения, приходящие от разных процессов

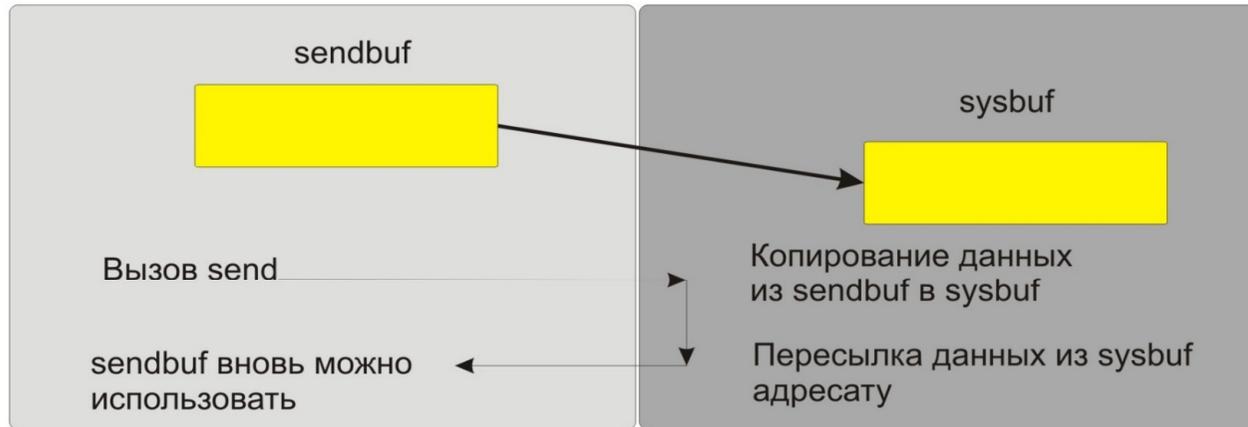
**Тег** - задаваемое пользователем целое число (от 0 до 32767), идентификатор сообщения.

Теги могут использоваться для соблюдения определенного порядка приема сообщений.

# Передача-прием сообщения

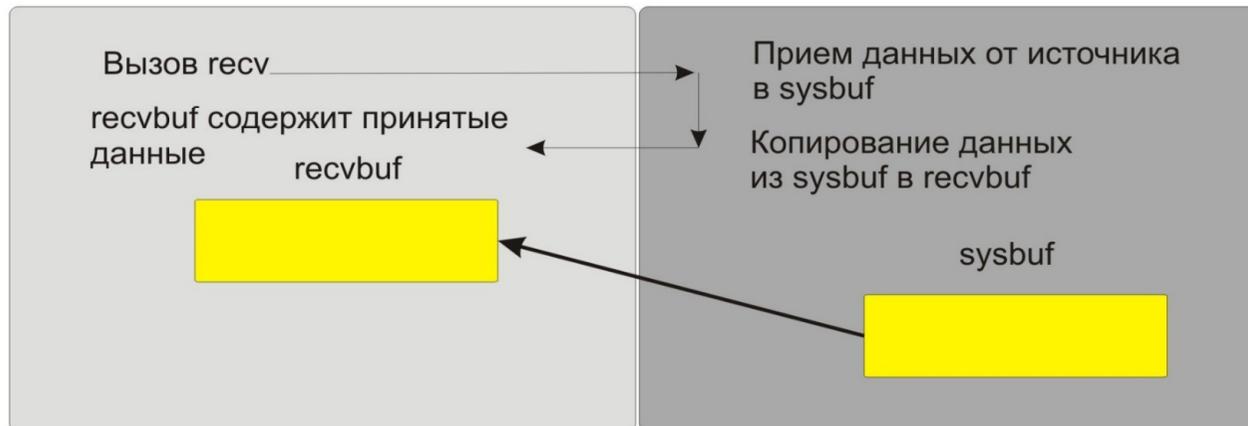
Пользовательский режим

Системный режим



Пользовательский режим

Системный режим



Данные, содержащиеся в сообщении, в общем случае организованы в массив элементов, каждый из которых имеет один и тот же тип. По умолчанию предполагается, что элементы массива располагаются в памяти последовательно, один за другим.

Кроме пересылки данных система передачи сообщений поддерживает пересылку информации о состоянии процессов коммуникации. Это может быть, например, уведомление о том, что прием данных, отправленных другим процессом, завершен.

Перед использованием процедур передачи сообщений программа должна "подключиться" к системе обмена сообщениями.

Подключение выполняется с помощью соответствующего вызова процедуры из библиотеки. В одних реализациях модели допускается только одно подключение, а в других несколько подключений к системе.

Прием сообщения начинается с подготовки буфера достаточного размера. В этот буфер записываются принимаемые данные.

Операция отправки или приема сообщения считается завершенной, если программа может вновь использовать такие ресурсы, как буферы сообщений.

# Реализации MPI

**MPI CHameleon (MPICH)** является свободно распространяемой “opensource” реализацией MPI, распространяемой по лицензии BSD. Этот пакет доступен в исходных кодах, поэтому допускает гибкую настройку.

Поддерживается работа в различных версиях ОС UNIX, Mac OS и Microsoft Windows.

Последние дистрибутивы MPICH соответствуют спецификации MPI-3.1.

Поддерживаются различные коммуникационные среды (в т.ч. InfiniBand, Myrinet, Quadrics).

Имеется версия с поддержкой пакета Globus.

**Официальный сайт:**

<http://www.mpich.org>

**OpenMPI** – “open source” реализация MPI, разрабатываемая консорциумом представителей академических, научных и промышленных кругов.

Полное соответствие спецификации MPI-3.

Поддержка различных ОС.

Поддержка различных коммуникационных сред.

**Официальный сайт:**

<http://www.open-mpi.org>

**Microsoft MPI** входит в состав Compute Cluster Pack SDK.

Ориентирован на работу в среде ОС Microsoft Windows и доступен, в том числе, по академической лицензии.

Основан на MPICH, но включает дополнительные средства управления заданиями.

**Intel® MPI** входит в состав Intel® Cluster Toolkit.

Это коммерческая реализация MPI, оптимизированная для архитектуры Intel.

**Сайт в Интернете:**

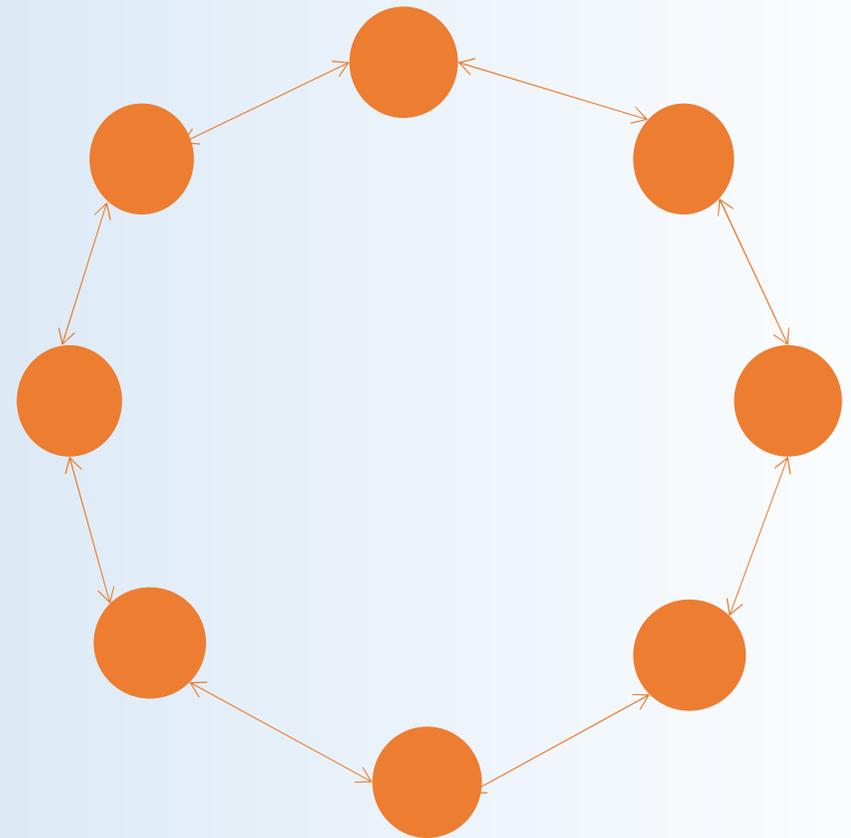
<http://www.intel.com>

# **Технические подробности**

## Демон mpd

В MPI-2 демон mpd играет важную роль. Параллельная программа может выполняться только если предварительно были запущены демоны mpd, образующие «кольцо демонов».

Он управляет выполнением процессов MPI-программы на данном вычислительном узле. Демоны запускаются от имени конкретных пользователей ОС и не влияют друг на друга.



- Кольцо демонов создаётся один раз и может быть использовано многократно, разными программами одного пользователя.
- Кольцо демонов прекращает своё существование в результате выполнения команды `mpdallexit`.
- Кольцо демонов одного пользователя не может взаимодействовать с демонами mpd других пользователей.
- Благодаря демонам mpd запуск MPI-программ выполняется быстрее.
- Исключена возможность «зависания» процессов.

## Конфигурационный файл **.mpd.conf**

### Пример файла **.mpd.conf**

```
MPD_SECRETWORD=valenki  
MPD_PORT_RANGE=1003:10003
```

## Файл **mpd.hosts**

### Пример файла **mpd.hosts**

```
pd00 ifhn=195.168.0.69  
pd01 ifhn=192.168.0.74  
pd02 ifhn=192.168.0.75  
pd03 ifhn=192.168.0.76  
pd04 ifhn=192.168.0.77  
pd05  
pd06  
pd07
```

В программных реализациях MPI имеются средства управления работой демонов mrd.

Запуску параллельной программы предшествует запуск демона mrd на всех узлах вычислительной системы. Демоны взаимодействуют друг с другом.

Запуск демонов (в этом примере 5) выполняется командой:

```
mpdboot -n 5
```

Проверка взаимодействия демонов между собой выполняется командой:

```
mpdtrace
```

Если при выполнении этой команды выводятся сообщения об ошибках, это говорит о неправильной настройке MPI или локальной сети.

Завершение работы демонов выполняется командой:

```
mpdallexit
```

## Трансляция в режиме командной строки

```
# mpicc -compile_info
```

```
cc -DUSE_STDARG -DHAVE_STDLIB_H=1 -DHAVE_STRING_H=1 -DHAVE_UNISTD_H=1 -DHAVE_STDARG_H=1  
-DUSE_STDARG=1 -DMALLOC_RET_VOID=1 -I/usr/local/mpich/include -c
```

```
# mpicc -link_info
```

```
cc -DUSE_STDARG -DHAVE_STDLIB_H=1 -DHAVE_STRING_H=1 -DHAVE_UNISTD_H=1 -DHAVE_STDARG_H=1  
-DUSE_STDARG=1 -DMALLOC_RET_VOID=1 -L/usr/local/mpich/lib -lmpich
```

```
# mpif77 -compile_info
```

```
f77 -I/usr/local/mpich/include -c
```

```
# mpif77 -link_info
```

```
f77 -L/usr/local/mpich/lib -lmpich
```

## Запуск параллельной программы

```
mpirun [ключи] -n число имя_исполняемого файла
```

# **Привязки к языкам программирования**

## Привязка к C/C++

При использовании MPI в программах на языке C в именах функций используется префикс MPI\_, а первая буква имени набирается в верхнем регистре.

Имена подпрограмм имеют вид **Класс\_действие\_подмножество** или **Класс\_действие**.

Аналогичное правило действует и для подпрограмм MPI для языка Fortran.

В C++ подпрограмма является методом для определенного класса, имя имеет в этом случае вид **MPI::Класс::действие\_подмножество**.

Значения кода завершения имеют целый тип и определяются по значению функции.

Имена констант MPI записываются в верхнем регистре. Их описания находятся в заголовочном файле **mpi.h**, который обязательно включается в MPI-программу. Имя этого файла может быть другим в других реализациях MPI.

Входные параметры функций передаются по значению, а выходные (и INOUT) — по ссылке.

В MPI принята своя система обозначения типов данных, которая соответствует типам данных в языках C и Fortran (соответствие неполное).

Тип данных MPI	Тип данных 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	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	Нет соответствия
MPI_PACKED	Нет соответствия

Все имена подпрограмм и констант MPI начинаются с **MPI\_**.

Коды завершения передаются через дополнительный параметр целого типа (находится на последнем месте в списке параметров подпрограммы).

Код успешного завершения — **MPI\_SUCCESS**.

Константы и другие объекты MPI описываются в файле **mpi.h**, который обязательно включается в MPI-программу с помощью оператора **include**. Этот оператор находится в начале программы.

# Привязка к Fortran

Подпрограммы MPI в программах на языке Fortran являются процедурами и вызываются оператором **call**. В именах используется префикс **MPI\_**.

Переменная **status** является массивом стандартного целого типа. Его размер и индексы задаются именованными константами:

```
integer status(MPI_STATUS_SIZE)
...
if(status(MPI_TAG).EQ.tag1) then
...
```

В программах на языке Fortran такие объекты MPI, как **MPI\_Datatype** или **MPI\_Comm** — целого типа (**integer**).

Тип данных MPI	Тип данных FORTRAN
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	Нет соответствия
MPI_PACKED	Нет соответствия
<b>Типы, которые имеются не во всех реализациях MPI</b>	
MPI_INTEGER1	INTEGER*1
MPI_INTEGER2	INTEGER*2
MPI_INTEGER4	INTEGER*4
MPI_REAL4	REAL*4
MPI_REAL8	REAL*8

## Коды завершения

Коды завершения возвращаются в качестве значения функции C или через последний аргумент процедуры Fortran. Исключение составляют подпрограммы **MPI\_Wtime** и **MPI\_Wtick**, в которых возвращение кода ошибки не предусмотрено.

Используются стандартные значения:

- `MPI_SUCCESS` — при успешном завершении вызова;
- `MPI_ERR_OTHER` — обычно при попытке повторного вызова процедуры **MPI\_Init**.

Вместо числовых кодов в программах обычно используют специальные именованные константы. Среди них:

- `MPI_ERR_BUFFER` — неправильный указатель на буфер;
- `MPI_ERR_COMM` — неправильный коммутатор;
- `MPI_ERR_RANK` — неправильный ранг;
- `MPI_ERR_OP` — неправильная операция;
- `MPI_ERR_ARG` — неправильный аргумент;
- `MPI_ERR_UNKNOWN` — неизвестная ошибка;
- `MPI_ERR_INTERN` — внутренняя ошибка. Обычно возникает, если системе не хватает памяти.

# **Структура МРІ-программы**

В программе MPI следует соблюдать определенные правила, без которых она окажется неработоспособной.

В начале программы, сразу после ее заголовка, необходимо подключить соответствующий заголовочный файл. В программе на языке C это **mpi.h**:

```
#include "mpi.h"
```

а в программе на языке Fortran — **mpif.h**:

```
include "mpif.h"
```

В этих файлах содержатся описания констант и переменных библиотеки MPI.

Первым вызовом библиотечной процедуры MPI в программе должен быть вызов подпрограммы инициализации **MPI\_Init**, перед ним может располагаться только вызов **MPI\_Initialized**, с помощью которого определяют, инициализирована ли система MPI. Вызов процедуры инициализации выполняется только один раз.

В языке Fortran у процедуры инициализации единственный аргумент — код ошибки:

```
integer IERR  
  
call mpi_init(ierr)
```

В C параметры функции инициализации получают адреса аргументов главной программы, задаваемых при ее запуске:

```
MPI_Init(&argc, &argv);
```

Загрузчик **mpirun** в конец командной строки запуска MPI-программы добавляет служебные параметры, необходимые **MPI\_Init**. В программах на языке Fortran аргументы командной строки не используются.

Процедура инициализации создает коммунитор со стандартным именем **MPI\_COMM\_WORLD**. Это имя указывается во всех последующих вызовах процедур MPI.

После выполнения всех обменов сообщениями в программе должен располагаться вызов процедуры

```
MPI_Finalize(ierr)
```

В результате этого вызова удаляются структуры данных MPI и выполняются другие необходимые действия. Программист должен позаботиться о том, чтобы к моменту вызова процедуры **MPI\_Finalize** были завершены все пересылки данных. После выполнения данного вызова другие вызовы процедур MPI, включая **MPI\_Init**, недопустимы. Исключение составляет подпрограмма **MPI\_Initialized**, которая возвращает значение "истина", если процесс вызывал **MPI\_Init**. Данный вызов может находиться в любом месте программы.

## Организация программы по схеме master-slave

```
program parallel
...
if (процесс = мастер) then
  master
else
  slave
endif
...
end
```

# **Простейшая МРІ-программа**

```
MPI_Comm_size(comm, size)
```

определение размера области взаимодействия.

Здесь **comm** - входной параметр-коммуникатор, выходным является параметр **size** целого типа - количество процессов в области взаимодействия.

```
MPI_Comm_rank(comm, pid)
```

определение номера процесса.

Здесь **pid** - идентификатор процесса в указанной области взаимодействия.

## simple\_MPI.cpp

```
#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);
    printf("Hello world from process %i \n", ProcRank);
    MPI_Finalize();
    return 0;
}
```

## simple\_MPI.f90

```
program main_mpi
  include 'mpif.h'
  integer :: myid, numprocs, ierr
  call mpi_init(ierr)
  call mpi_comm_rank(mpi_comm_world, myid, ierr)
  call mpi_comm_size(mpi_comm_world, numprocs, ierr)
  print *, "process ", myid, " of ", numprocs
  call mpi_finalize(ierr)
end program
```

```
[nemnugin@pd00 ~]$ mpdboot -n 3  
[nemnugin@pd00 ~]$ mpicxx simple_MPI.cpp  
[nemnugin@pd00 ~]$ mpiexec -n 3 ./a.out  
Hello world from process 0  
Hello world from process 1  
Hello world from process 2  
[nemnugin@pd00 ~]$ █
```

## Задания для самостоятельной работы

В исходном тексте программы на языке C пропущены вызовы процедур подключения к MPI, определения количества процессов и ранга процесса. Добавить эти вызовы, откомпилировать и запустить программу.

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

## Задания для самостоятельной работы

В исходном тексте программы на языке Fortran пропущены вызовы процедур подключения к MPI, определения количества процессов и ранга процесса. Добавить эти вызовы, откомпилировать и запустить программу.

```
program main_mpi
include 'mpif.h'
integer myid, numprocs, ierr
....
print *, "process ", myid, " of ", numprocs
call mpi_finalize(ierr)
stop
end
```