

# Технология программирования MPI (5)

Антонов Александр Сергеевич,  
к.ф.-м.н., вед.н.с. лаборатории Параллельных  
информационных технологий НИВЦ МГУ

Летняя суперкомпьютерная академия  
Москва, 2016

**МРІ**

**Динамическое управление  
процессами**

# MPI

```
MPI_Comm_spawn(char *command,  
char *argv[], int maxprocs,  
MPI_Info info, int root,  
MPI_Comm comm, MPI_Comm  
*intercomm, int  
array_of_errcodes[])
```

Попытка породить **maxprocs** процессов, выполняющих программу **command**, которая будет выполнена командным интерпретатором.

# MPI

Создаётся интеркоммуникатор **intercomm**, объединяющий порождающие и порожденные процессы для возможности дальнейших обменов данными. В **argv** передаются параметры запуска программы **command** (если не требуется, можно указать предопределённую константу **MPI\_ARGV\_NULL**). Первые четыре аргумента значимы только на процессе **root**, на остальных процессах они игнорируются.

# MPI

Вызов является коллективным для процессов коммуникатора **comm** и не завершается до тех пор, пока во всех порождённых процессах не будет вызвана процедура **MPI\_Init**. И наоборот, процедура **MPI\_Init** в порождаемых процессах не выполнится, пока все порождающие процессы не вызовут **MPI\_Comm\_spawn**.

# MPI

Порождённые процессы получают свой собственный коммуникатор **MPI\_COMM\_WORLD**, отличный от того, который был у выполнявшихся ранее процессов. Порядок нумерации процессов в группах интеркоммуникатора **intercomm** соответствует порядку процессов в коммуникаторе **comm** порождающих процессов и в коммуникаторе **MPI\_COMM\_WORLD** порождённых процессов.

# MPI

В порождённых процессах интеркоммуникатор может быть получен при помощи вызова процедуры **MPI\_Comm\_get\_parent**. Если не удаётся породить **maxprocs** процессов, то возвращается ошибка **MPI\_ERR\_SPAWN**. Аргумент **info** задаёт информацию о том, где и как запускать порождаемые процессы. Если не требуется, можно использовать предопределённую константу **MPI\_INFO\_NULL**.

# MPI

```
int MPI_Comm_get_parent (MPI_Comm  
*parent)
```

Процедура возвращает на динамически порождённых процессах интеркоммуникатор **parent** для общения с порождающими процессами. Если вызов стоит не в порождённом процессе или если коммуникатор порождающего процесса больше не существует или недоступен, то возвращается значение **MPI\_COMM\_NULL**.



# MPI

master.c:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
    int size, rank1, rank2;
    MPI_Status status;
    MPI_Comm intercomm;
    char slave[10]="./slave";
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_spawn(slave, MPI_ARGV_NULL, 2, MPI_INFO_NULL,
0, MPI_COMM_SELF, &intercomm, MPI_ERRCODES_IGNORE);
    MPI_Recv(&rank1, 1, MPI_INT, 0, 0, intercomm, &status);
    MPI_Recv(&rank2, 1, MPI_INT, 1, 1, intercomm, &status);
```

# MPI

```
printf("Slaves %d and %d are working\n", rank1, rank2);  
MPI_Finalize();  
return 0;  
}
```

# MPI

slave.c:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
    int rank, size;
    MPI_Comm intercomm;
    MPI_Init(&argc, &argv);
    MPI_Comm_get_parent(&intercomm);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Send(&rank, 1, MPI_INT, 0, rank, intercomm);
    MPI_Finalize();
    return 0;
}
```

# MPI

```
int MPI_Comm_spawn_multiple(int
count, char
*array_of_commands[], char
**array_of_argv[], int
array_of_maxprocs[], MPI_Info
array_of_info[], int root,
MPI_Comm comm, MPI_Comm
*intercomm, int
array_of_errcodes[])
```

Порождение процессов, выполняющих **count** разных программ.

# MPI

Команды перечислены в **array\_of\_commands**. Аргументы задаются в массиве **array\_of\_argv** (если не требуется, можно указать константу **MPI\_ARGVS\_NULL**). Значения максимального количества процессов для соответствующих программ задаются в массиве **array\_of\_maxprocs**, информация о запуске – в массиве **array\_of\_info**.

# MPI

Все порождаемые процессы объединяются одним коммуникатором **MPI\_COMM\_WORLD**.

Порядок нумерации процессов соответствует порядку перечисления запускаемых программ.

# МРІ

Ряд процедур МРІ позволяет организовать общение двух независимо существующих групп процессов, не объединённых каким-либо коммуникатором. Для этого одна из таких групп (*сервер*) должна принимать соединение от другой группы (*клиент*).

# MPI

```
int MPI_Open_port(MPI_Info info,  
char *port_name)
```

Устанавливает на сервере адрес, задаваемый строкой **port\_name**, для связи с клиентами. Тип адреса зависит от реализации, возможно, с использованием информации из аргумента **info**. Если не требуется, можно указать константу **MPI\_INFO\_NULL**. В **port\_name** возвращается строка длиной не более **MPI\_MAX\_PORT\_NAME** символов. Обычно **port\_name** является сетевым адресом.



# MPI

```
int MPI_Close_port(char  
*port_name)
```

Освобождает сетевой адрес, заданный строкой **port\_name**.

# MPI

```
int MPI_Comm_accept(char  
*port_name, MPI_Info info, int  
root, MPI_Comm comm, MPI_Comm  
*newcomm)
```

Возвращает интеркоммуникатор **newcomm**, для установления связи с клиентом по адресу **port\_name**. Процедура является коллективной для **comm**. Значение **port\_name** должно быть получено при помощи вызова процедуры **MPI\_Open\_port**.

# MPI

```
int MPI_Comm_connect(char  
*port_name, MPI_Info info, int  
root, MPI_Comm comm, MPI_Comm  
*newcomm)
```

Возвращает интеркоммуникатор **newcomm**, для установления связи с сервером по адресу **port\_name**. Если адрес не существует (или порт закрыт), то возвращается ошибка **MPI\_ERR\_PORT**. Адрес **port\_name** должен быть тем же, который возвращает процедура **MPI\_Open\_port**.

# MPI

```
int MPI_Publish_name(char  
*service_name, MPI_Info info,  
char *port_name)
```

Сопоставление известному имени **service\_name** системно-зависимого адреса **port\_name**. В дальнейшем для установления связи можно использовать имя **service\_name**.

# MPI

```
int MPI_Unpublish_name(char  
*service_name, MPI_Info info,  
char *port_name)
```

Удаление сопоставления имени **service\_name** с адресом **port\_name**. Если сопоставления не было, вернется ошибка **MPI\_ERR\_SERVICE**. Перед закрытием соответствующего порта все сопоставления имен должны быть удалены.

# MPI

```
int MPI_Lookup_name(char  
*service_name, MPI_Info info,  
char *port_name)
```

По имени сервиса **service\_name** определяется связанный с ним адрес **port\_name**. Если данный адрес с сервисом не был связан, процедура вернет ошибку **MPI\_ERR\_NAME**.

# MPI

server.c:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
    int r;
    char port_name[MPI_MAX_PORT_NAME];
    MPI_Status status;
    MPI_Comm intercomm;
    MPI_Open_port(MPI_INFO_NULL, port_name);
    MPI_Publish_name("example", MPI_INFO_NULL, port_name);
    MPI_Comm_accept(port_name, MPI_INFO_NULL, 0,
MPI_COMM_SELF, &intercomm);
    MPI_Recv(&r, 1, MPI_INT, 0, 0, intercomm, &status);
```

# MPI

```
    MPI_Unpublish_name("example", MPI_INFO_NULL,  
port_name);  
    MPI_Close_port(port_name);  
    printf("Client is sent %d\n", r);  
    MPI_Finalize();  
    return 0;  
}
```



# MPI

client.c:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
    int rank;
    char port_name[MPI_MAX_PORT_NAME];
    MPI_Comm intercomm;
    MPI_Lookup_name("example", MPI_INFO_NULL, port_name);
    MPI_Comm_connect(port_name, MPI_INFO_NULL, 0,
MPI_COMM_SELF, &intercomm);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Send(&rank, 1, MPI_INT, 0, 0, intercomm);
    MPI_Finalize();
    return 0;
}
```

# MPI

```
int MPI_Comm_disconnect(MPI_Comm  
*comm)
```

Процессы, вызвавшие процедуру, ждут завершения всех связанных с коммуникатором **comm** пересылок данных (в отличие от использования процедуры **MPI\_Comm\_free**), удаляют данный коммуникатор и устанавливают **comm** в **MPI\_COMM\_NULL**.

# MPI

## Параллельный ввод/вывод

# MPI

В MPI под *файлом (file)* понимается упорядоченная последовательность типизированных блоков данных.

Поддерживается произвольный или последовательный доступ к любому неделимому набору этих блоков. Файл может быть открыт группой процессов и обрабатываться с помощью коллективных операций. Под *смещением (displacement)* понимается расстояние в байтах относительно начала файла.

# MPI

*Элементарный тип* данных (*etype*) – единица доступа к данным и позиционирования в файле. Это может быть любой *предопределённый* или *производный* тип. Производные элементарные типы создаются при помощи любой из процедур создания типов данных MPI, обеспечивающих, чтобы получающиеся смещения внутри типа были неотрицательными и монотонно не убывали. Доступ к данным производится в единицах элементарных типов.

# MPI

*Файловый тип (filetype)* - основа разделения файла между процессами, определяет шаблон доступа к файлу. Файловый тип – это либо элементарный тип, либо производный тип данных MPI, состоящий из нескольких вхождений одного и того же элементарного типа. Размер любого «пропуска» в файловом типе должен быть кратным размеру этого элементарного типа. Смещения внутри файлового типа неотрицательные и монотонно неубывающие.

# MPI

*Образ (view)* определяет текущий набор данных, доступный в открытом файле в виде упорядоченного набора элементарных типов. Каждый процесс имеет свой образ файла, определяемый тремя параметрами: смещением, элементарным типом и файловым типом. Шаблон, описанный файловым типом, получается таким, как если бы он задавался процедурой **MPI\_Type\_contiguous** из файлового типа и сколь угодно большого числа повторений.

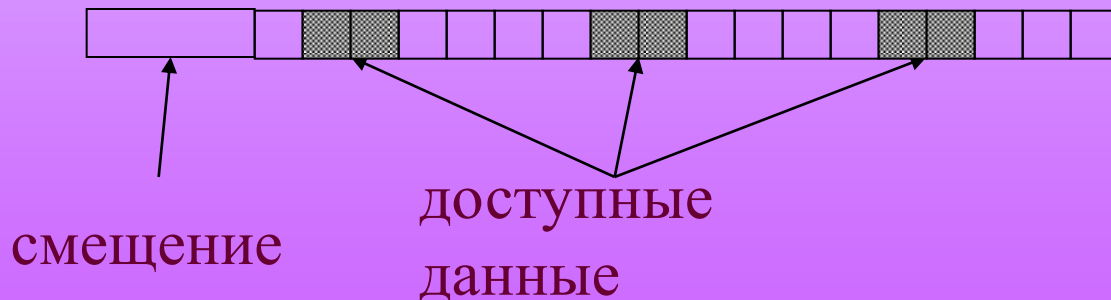
# MPI

Пусть элементарный тип данных изображается так: ■

Пусть пропуск в файле изображается так: □

Тогда файловый тип может быть задан, например, следующим образом: □■□□□

Разбиение файла на элементы файлового типа может иметь следующий вид:





# MPI

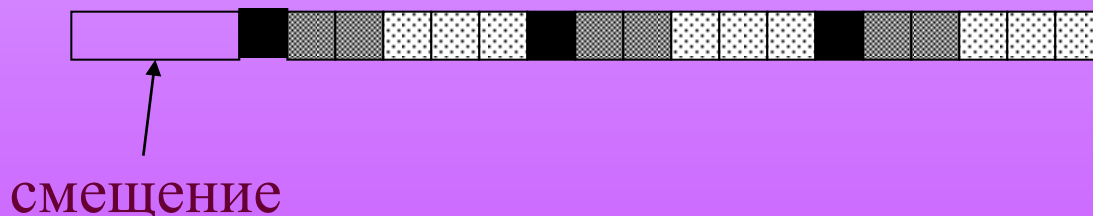
Образы разных процессов могут дополнять друг друга для распределения данных.

Файловый тип процесса 0: 

Файловый тип процесса 1: 

Файловый тип процесса 2: 

Разбиение файла на элементы файловых типов трёх процессов:



# МРІ

*Сдвиг (offset)* – позиция в файле относительно текущего образа, выраженная через количество элементарных типов. При вычислении сдвига «пропуски» в файловом типе образа не учитываются. Нулевой сдвиг – это позиция первого видимого в образе элементарного типа (после пропуска смещения и начальных «пропусков» в образе). Например, сдвиг **2** для процесса **1** в предыдущем примере будет иметь позиция **8**-го элементарного типа после смещения.

# MPI

*Размер файла (file size)* измеряется в байтах от начала файла. Только что созданный файл имеет размер 0 байт. Использование размера файла в качестве абсолютного смещения определяет позицию байта, следующего сразу за последним байтом файла. Для любого образа *конец файла (end of file)* – это сдвиг первого элементарного типа, доступного в данном образе после последнего байта файла.

# MPI

*Указатель на файл (file pointer) – неявный сдвиг, устанавливаемый средствами MPI. Индивидуальные указатели на файл – это файловые указатели, локальные для каждого процесса, открывающего файл. Общие файловые указатели – это указатели, используемые группой процессов.*

*Дескриптор файла (file handle) – скрытый объект, создаваемый процедурой `MPI_File_open` и удаляемый процедурой `MPI_File_close`.*

# MPI

```
int MPI_File_open(MPI_Comm comm,  
char *filename, int amode,  
MPI_Info info, MPI_File *fh)
```

Открытие файла **filename** на всех процессах интракоммуникатора **comm**. Если нужно открыть файл на одном процессе, можно использовать предопределённый коммуникатор **MPI\_COMM\_SELF**.

Возвращаемый дескриптор файла **fh** в дальнейшем используется для доступа к файлу вплоть до его закрытия. Формат имени файла **filename** зависит от реализации.

# MPI

**amode** задаёт способ доступа к файлу комбинацией (при помощи побитовой операции «или») следующих констант:

**MPI\_MODE\_RDONLY** – только чтение;

**MPI\_MODE\_RDWR** – чтение и запись;

**MPI\_MODE\_WRONLY** – только запись;

**MPI\_MODE\_CREATE** – создавать файл, если он не существует;

**MPI\_MODE\_EXCL** – выдавать ошибку, если файл уже существует;

# MPI

**MPI\_MODE\_DELETE\_ON\_CLOSE** – удалять файл при закрытии;

**MPI\_MODE\_UNIQUE\_OPEN** – к файлу не будет осуществляться доступ другими процессами;

**MPI\_MODE\_SEQUENTIAL** – файл будет доступен только в последовательном режиме;

**MPI\_MODE\_APPEND** – установить начальную позицию всех файловых указателей на конец файла.

# MPI

```
int MPI_File_close(MPI_File *fh)
```

Синхронизация состояния (**MPI\_File\_sync**) и закрытие файла, связанного с дескриптором **fh**. Если файл был открыт со способом доступа **MPI\_MODE\_DELETE\_ON\_CLOSE**, то он удаляется. Процедура является коллективной. Перед вызовом должны быть завешены все операции, использующие файловый дескриптор **fh**. После выполнения процедуры **fh** принимает значение **MPI\_FILE\_NULL**.



# MPI

```
int MPI_File_delete(char  
*filename, MPI_Info info)
```

Удаление файла с именем **filename**. Если не существует, возникает ошибка **MPI\_ERR\_NO\_SUCH\_FILE**. Если удаляемый файл открыт данным процессом, то поведение при любой попытке доступа к файлу зависит от реализации. При этом также зависит от реализации, будет ли удален данный файл. Если нет, возникает ошибка **MPI\_ERR\_FILE\_IN\_USE** или **MPI\_ERR\_ACCESS**.

# MPI

```
int MPI_File_set_size(MPI_File  
fh, MPI_Offset size)
```

Изменяет в **size** байт размер файла, заданного дескриптором **fh**. Процедура является коллективной, все процессы группы должны устанавливать одно и то же значение **size**. Если новый размер меньше прежнего, то файл урезается. Если больше, то размер файла увеличивается до **size**. Прежние данные не изменяются, а значения данных в новых участках файла являются неопределёнными.

# MPI

Процедура **MPI\_File\_set\_size** не изменяет файловые указатели. Если при открытии файла был определён способ доступа **MPI\_MODE\_SEQUENTIAL**, то использование данной процедуры является ошибочным.

Перед вызовом процедуры **MPI\_File\_set\_size** должны быть завершены все операции, использующие файловый дескриптор **fh**.

# MPI

**int**

**MPI\_File\_preallocate**(MPI\_File  
**fh**, MPI\_Offset **size**)

Обеспечивает пространство для хранения первых **size** байт файла, заданного дескриптором **fh**. Данная операция является коллективной. При этом записанные ранее данные не изменяются. В новых областях файла данные не определены. Если **size** больше текущего размера файла, то размер увеличивается до **size**. Если меньше либо равно, то размер не изменяется.

# MPI

Процедура **MPI\_File\_preallocate** не изменяет файловые указатели. Если при открытии файла был определён способ доступа **MPI\_MODE\_SEQUENTIAL**, то использование данной процедуры ошибочно.

Перед вызовом процедуры **MPI\_File\_preallocate** должны быть завешены все операции, использующие файловый дескриптор **fh**.

# MPI

```
int MPI_File_get_size(MPI_File  
fh, MPI_Offset *size)
```

Процедура возвращает в аргументе **size** размер в байтах файла, заданного дескриптором **fh**.

# MPI

```
int MPI_File_get_group(MPI_File  
fh, MPI_Group *group)
```

Процедура возвращает в аргументе **group** копию группы коммутатора, использованного при открытии файла с дескриптором **fh**.

```
int MPI_File_get_amode(MPI_File  
fh, int *amode)
```

Процедура возвращает в аргументе **amode** способ доступа, заданный при открытии файла с дескриптором **fh**.

# MPI

```
int MPI_File_set_view(MPI_File  
fh, MPI_Offset disp,  
MPI_Datatype etype, MPI_Datatype  
filetype, char *datarep,  
MPI_Info info)
```

Процедура изменяет образ файла для вызвавшего процесса. В аргументе **disp** задаётся начальное смещение создаваемого образа, в аргументе **etype** – элементарный тип данных, в аргументе **filetype** – файловый тип.



# MPI

Процедура **MPI\_File\_set\_view** обнуляет все индивидуальные и общие файловые указатели. Процедура является коллективной, значения **datatype** и размеры элементарных типов должны совпадать во всех процессах группы; значения **disp**, **filetype** и **info** могут различаться. Типы данных, передаваемые в **etype** и **filetype**, должны быть согласованы.

# MPI

Если файл был открыт как **MPI\_MODE\_SEQUENTIAL**, то в качестве **disp** должно быть задано специальное смещение **MPI\_DISPLACEMENT\_CURRENT**.

Если файл открыт для записи, то ни элементарный тип, ни файловый тип не должны иметь перекрывающихся областей. При этом файловые типы из разных процессов могут перекрывать друг друга.

# MPI

Если в файловом типе есть «пропуски», то тогда данные в них недоступны для вызывающего процесса. Однако аргументы **disp**, **etype** и **filetype** могут быть изменены посредством последующих вызовов процедуры **MPI\_File\_set\_view**, что позволяет получить доступ к различным частям файла.

При конструировании элементарных и файловых типов нельзя использовать абсолютные адреса.

# MPI

Аргумент **datatype** задаёт *представление данных* в файле. В стандарте MPI описывается три вида представления:

**"native"**

**"internal"**

**"external32"**

# MPI

```
int MPI_File_get_view(MPI_File  
fh, MPI_Offset *disp,  
MPI_Datatype *etype,  
MPI_Datatype *filetype, char  
*datarep)
```

В аргументе **disp** возвращается текущее значение смещения, в аргументах **etype** и **filetype** – новые типы, эквивалентные элементарному типу и файловому типу соответственно, в аргументе **datarep** – представление данных. Длина **datarep** не больше **MPI\_MAX\_DATAREP\_STRING**.

# МРІ

Данные перемещаются между файлами и процессами при помощи вызовов операций чтения и записи. Доступ к данным имеет три основных аспекта:

*позиционирование* (указание явного смещения или использование указателя на файл),

*синхронизация* (блокирующие или неблокирующие операции) и

*координация* (локальные или коллективные операции).

# MPI

Процедуры доступа к данным с указанием явного смещения:

Синхронизация	Координация	
	локальные	коллективные
блокирующие	<code>MPI_File_read_at</code> <code>MPI_File_write_at</code>	<code>MPI_File_read_at_all</code> <code>MPI_File_write_at_all</code>
неблокирующие или разделенные коллективные	<code>MPI_File_iread_at</code> <code>MPI_File_iwrite_at</code>	<code>MPI_File_read_at_all_begin</code> <code>MPI_File_read_at_all_end</code> <code>MPI_File_write_at_all_begin</code> <code>MPI_File_write_at_all_end</code>

# MPI

Процедуры доступа к данным с заданием индивидуальных файловых указателей:

Синхронизация	Координация	
	ЛОКАЛЬНЫЕ	КОЛЛЕКТИВНЫЕ
блокирующие	<code>MPI_File_read</code> <code>MPI_File_write</code>	<code>MPI_File_read_all</code> <code>MPI_File_write_all</code>
неблокирующие или разделенные коллективные	<code>MPI_File_iread</code> <code>MPI_File_iwrite</code>	<code>MPI_File_read_all_begin</code> <code>MPI_File_read_all_end</code> <code>MPI_File_write_all_begin</code> <code>MPI_File_write_all_end</code>



# MPI

Процедуры доступа к данным с заданием общих файловых указателей:

Синхронизация	Координация	
	локальные	коллективные
блокирующие	<code>MPI_File_read_shared</code> <code>MPI_File_write_shared</code>	<code>MPI_File_read_ordered</code> <code>MPI_File_write_ordered</code>
неблокирующие или разделенные коллективные	<code>MPI_File_iread_shared</code> <code>MPI_File_iwrite_shared</code>	<code>MPI_File_read_ordered_begin</code> <code>MPI_File_read_ordered_end</code> <code>MPI_File_write_ordered_begin</code> <code>MPI_File_write_ordered_end</code>

# MPI

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
    int rank;
    MPI_File fh;
    char buf[10];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_File_open(MPI_COMM_WORLD, "file1.txt",
MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
    MPI_File_set_view(fh, rank*10, MPI_CHAR, MPI_CHAR,
"native", MPI_INFO_NULL);
    MPI_File_read_all(fh, buf, 10, MPI_CHAR,
MPI_STATUS_IGNORE);
    printf("process %d, buf=%s\n", rank, buf);
    MPI_File_close(&fh);
    MPI_Finalize();
}
```

# MPI

```
int MPI_File_sync(MPI_File fh)
```

Процедура гарантирует, что все предыдущие записи вызывающего процесса в файл с дескриптором **fh** будут завершены.

Процедура является коллективной.

Пользователь должен гарантировать, что все неблокирующие коллективные операции должны быть завершены перед вызовом **MPI\_File\_sync**.

# МРІ

Задание 8: Замерьте скорость обмена данными между процессами, установившими клиент-серверную связь.

# MPI

Задание 9: Напишите программу, в которой один процесс считывает из файла нечётные элементы массива данных некоторого типа, а другой процесс – чётные элементы.