

# Технология программирования MPI (1)

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

Летняя суперкомпьютерная академия  
Москва, 2017

# Параллелизм в МРІ

Необходимо выделить группы операций, которые могут вычисляться одновременно и независимо. Возможность этого определяется наличием или отсутствием в программе истинных информационных зависимостей.

Две операции программы называются *информационно зависимыми*, если результат выполнения одной операции используется в качестве аргумента в другой.

# Параллелизм в MPI

Пусть **size** – число процессов, а **rank** – номер процесса ( $0 \leq \text{rank} \leq \text{size} - 1$ ).

*Крупноблочное распараллеливание:*

```
if (rank == 0) { /* операции,  
выполняемые 0-ым процессом */ }
```

...

```
if (rank == K) { /* операции,  
выполняемые K-ым процессом */ }
```

# Параллелизм в MPI

Наибольший ресурс параллелизма в программах сосредоточен в циклах!

*Распределение итераций циклов:*

```
for (i = 0; i < N; i++) {  
    if (i ~ rank) {  
        /* операции i-ой итерации цикла для  
        выполнения процессом rank */  
    }  
}
```

# Параллелизм в МРІ

Примеры способов распределения итераций циклов:

- *Блочное распределение* – по  $\lceil N/P \rceil$  итераций.
- *Блочно-циклическое распределение* – размер блока меньше, распределение продолжается циклически.
- *Циклическое распределение* – циклически по одной итерации.

# Параллелизм в MPI

Рассмотрим простейший цикл:

```
for (i=0; i<N; i++)  
    a[i] = a[i] + b[i];
```

Пусть его итерации требуется распределить между  $P$  процессами. Каждый процесс имеет уникальный идентификатор **rank**.

# Параллелизм в MPI

Блочное распределение:

```
// размер блока итераций  
k = (N-1) / P + 1;  
// начало блока итераций  
// процесса rank  
ibeg = rank * k;  
// конец блока итераций  
// процесса rank  
iend = (rank+1) * k - 1;
```

# Параллелизм в MPI

```
// если не досталось итераций
if (ibeg > N-1)
    iend = ibeg - 1;
else
// если досталось меньше итераций
    if (iend > N-1) iend = N-1;
for (i=ibeg; i<=iend; i++)
    a[i] = a[i] + b[i];
```



# Параллелизм в MPI

Циклическое распределение:

```
for (i=rank; i<N; i+=size)
    a[i] = a[i] + b[i];
```

# Параллелизм в МРІ

Цели распараллеливания:

- *равномерная загрузка процессов;*
- *минимизация количества и объёма необходимых пересылок данных.*

Пересылка данных требуется, если есть информационная зависимость между операциями, которые при выбранной схеме распределения попадают на разные процессы.

# MPI

- MPI - *Message Passing Interface*, интерфейс передачи сообщений.
- Стандарт MPI 3.1 (4 июня 2015 года).
- Более 450 процедур.
- SPMD (Single Program Multiple Data)-основная модель параллельного программирования.

# MPI

- Префикс `MPI_`.

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

(`mpif.h` для языка Фортран)

- *Процессы*, посылка сообщений.

- *Сообщение* – массив однотипных данных, расположенных в последовательных ячейках памяти.

- *Группа* – упорядоченное множество процессов.

# MPI

*Коммуникатор* – контекст обмена группы.

В операциях обмена используются только коммуникаторы!

Коммуникаторы имеют в языке Си предопределённый тип **MPI\_Comm** (в языке Фортран – тип **INTEGER**).

# MPI

**MPI\_COMM\_WORLD** – коммуникатор для всех процессов приложения.

**MPI\_COMM\_SELF** – коммуникатор, включающий только текущий процесс.

**MPI\_COMM\_NULL** – коммуникатор, не содержащий ни одного процесса.

# МРІ

Каждый процесс может одновременно входить в разные коммутаторы.

*Два основных атрибута процесса:  
коммутатор (группа) и номер процесса в коммутаторе (группе).*

Если коммутатор содержит **n** процессов, то номера процессов в нём лежат в пределах от **0** до **n-1**.

# MPI

- *Сообщение* — набор данных некоторого типа.
- Атрибуты сообщения: номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения, коммуниктор.
- Идентификатор сообщения (тег) - целое неотрицательное число в диапазоне от 0 до **MPI\_TAG\_UB** (не меньше **32767**).
- Для работы с атрибутами сообщений введена структура **MPI\_Status**.



# MPI

Возвращаемым значением (в Фортране – последним аргументом) для большинства процедур MPI является информация об успешности завершения.

В случае успешного выполнения процедура вернёт значение **MPI\_SUCCESS**, иначе - код ошибки.

Предопределённые значения, соответствующие различным ошибочным ситуациям, перечислены в файле **mpi.h**

# **MPI**

## **Общие процедуры MPI**

# MPI

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

Инициализация параллельной части программы. Почти все другие процедуры MPI могут быть вызваны только после вызова **MPI\_Init**. Инициализация параллельной части для каждого приложения должна выполняться только один раз.

# MPI

В языке Си функции **MPI\_Init** передаются указатели на аргументы командной строки **argc** и **argv**, из которых системой могут извлекаться и передаваться в параллельные процессы параметры запуска программы. Это позволяет обеспечить их в среде, где аргументы командной строки не предусмотрены. Если не требуется, то могут передаваться значения **NULL**.

# MPI

```
int MPI_Finalize(void)
```

Завершение параллельной части приложения. Все последующие обращения к большинству процедур MPI, в том числе к **MPI\_Init**, запрещены. К моменту вызова **MPI\_Finalize** каждым процессом программы все действия, требующие его участия в обмене сообщениями, должны быть завершены.

# MPI

Простейшая параллельная программа:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
    printf("Before MPI_INIT\n");
    MPI_Init(&argc, &argv);
    printf("Parallel section\n");
    MPI_Finalize();
    printf("After MPI_FINALIZE\n");
}
```

# MPI

```
int MPI_Initialized(int *flag)
```

В аргументе **flag** возвращает **1**, если вызвана после процедуры **MPI\_Init**, и **0** - в противном случае.

```
int MPI_Finalized(int *flag)
```

В аргументе **flag** возвращает **1**, если вызвана после процедуры **MPI\_Finalize**, и **0** - в противном случае.

Эти процедуры можно вызвать до **MPI\_Init** и после **MPI\_Finalize**.

# MPI

```
int MPI_Abort(MPI_Comm comm, int  
errorcode)
```

Процедура делает попытку максимально корректного завершения всех процессов коммуникатора **comm**. Процессы возвращают код завершения **errorcode**.



# MPI

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

В аргументе **size** возвращает число параллельных процессов в коммуникаторе **comm**.

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

В аргументе **rank** возвращает номер процесса в коммуникаторе **comm** в диапазоне от 0 до **size-1**.

# MPI

```
#include <stdio.h>
#include "mpi.h"
#define MAX 100
int main(int argc, char **argv)
{
    int rank, size, n, i, ibeg, iend;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    n=(MAX-1)/size+1;
    ibeg=rank*n+1; iend=(rank+1)*n;
    for(i=ibeg; i<=((iend>MAX)?MAX:iend); i++)
        printf ("process %d, %d^2=%d\n", rank, i, i*i);
    MPI_Finalize();
}
```

# MPI

**double MPI\_Wtime(void)**

Возвращает для каждого вызвавшего процесса астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом. Момент времени, используемый в качестве точки отсчёта, не будет изменён за время существования процесса.

**double MPI\_Wtick(void)**

Возвращает разрешение таймера в секундах.

# MPI

Таймеры разных процессов могут быть не синхронизированы и выдавать существенно различающиеся значения, это можно определить по значению предопределенной константы **MPI\_WTIME\_IS\_GLOBAL**: **1** – синхронизированы, **0** – нет.

# MPI

```
int MPI_Get_processor_name(char  
*name, int *len)
```

Возвращает в строке **name** имя узла, на котором запущен вызвавший процесс. В переменной **len** возвращается количество символов в имени, не превышающее константы **MPI\_MAX\_PROCESSOR\_NAME**.

# MPI

Определение характеристик системного таймера и использование процедуры

**MPI\_Get\_processor\_name:**

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

```
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();
printf ("processor %s, process %d: tick= %lf, time=
%lf\n", name, rank, tick, (time_finish-
time_start)/NTIMES);
MPI_Finalize();
}
```

# MPI

На некоторых системах ряд процедур MPI может работать быстрее, если данные для них располагаются в специально выделенных сегментах памяти.

```
int MPI_Alloc_mem(MPI_Aint size,  
MPI_Info info, void *baseptr)
```

Выделяет **size** байт динамической памяти, **baseptr** указывает на начало выделенного сегмента.



# MPI

Если требуемый объем памяти не может быть выделен, возвращается ошибка **MPI\_ERR\_NO\_MEM**. Аргумент **info** может задавать указания на желаемое размещение выделяемого сегмента (зависит от реализации). Если это не требуется, вместо аргумента **info** можно указать предопределенную константу **MPI\_INFO\_NULL**.

# MPI

```
int MPI_Free_mem(void *base)
```

Процедура реализует освобождение выделенного сегмента динамической памяти, начинающегося с адреса **base**. В случае неверного адреса процедура возвращает ошибку **MPI\_ERR\_BASE**.

# **МРІ**

**Передача и приём сообщений  
типа точка-точка**

# МРІ

В операциях типа точка-точка участвуют два процесса, один является отправителем сообщения, другой – получателем. Процесс-отправитель должен вызвать одну из процедур передачи данных и явно указать номер процесса-получателя в некотором коммутаторе, а процесс-получатель должен вызвать одну из процедур приема с указанием того же коммутатора; он может не знать точный номер процесса-отправителя в данном коммутаторе.

# МРІ

Все процедуры данной группы, в свою очередь, также делятся на два класса: процедуры *с блокировкой* и процедуры *без блокировки (асинхронные)*. Процедуры обмена с блокировкой приостанавливают работу процесса до выполнения некоторого условия, а возврат из асинхронных процедур происходит немедленно после инициализации соответствующей коммуникационной операции.

# **МРІ**

## **Передача и приём сообщений с блокировкой**

# MPI

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

Блокирующая посылка массива **buf** с идентификатором **msgtag**, состоящего из **count** элементов типа **datatype**, процессу с номером **dest** в коммуникаторе **comm**.

# MPI

Операция начинается независимо от того, была ли инициализирована соответствующая процедура приема. Сообщение может быть скопировано как в буфер приема, так и помещено в системный буфер (если это предусмотрено в **MPI**). **count** может быть 0, в этом случае будет пересылаться только системная информация. Процесс может передавать сообщение самому себе, однако это небезопасно и может привести к возникновению тупиковой ситуации.



# MPI

Типы данных:

- `MPI_INT` - `int`
- `MPI_SHORT` - `short`
- `MPI_LONG` - `long`
- `MPI_FLOAT` - `float`
- `MPI_DOUBLE` - `double`
- `MPI_CHAR` - `char`
- `MPI_BYTE` - 8 бит
- `MPI_PACKED` - тип для упакованных данных.

Все типы данных перечислены в файле `mpi.h`.

# MPI

*Блокировка* при пересылке данных гарантирует корректность повторного использования всех параметров после возврата из процедуры. После возврата из **MPI\_Send** можно использовать любые переменные без опасения испортить передаваемое сообщение. Выбор способа этой гарантии – копирование в промежуточный буфер или непосредственная передача процессу **dest** – остается за разработчиками конкретной реализации MPI.

# MPI

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

Блокирующий приём в буфер **buf** не более **count** элементов сообщения типа **datatype** с идентификатором **msgtag** от процесса с номером **source** в коммутаторе **comm** с заполнением структуры атрибутов приходящего сообщения **status**.

# MPI

Если число реально принятых элементов сообщения меньше **count**, то в буфере **buf** изменятся только элементы, соответствующие элементам принятого сообщения. Если количество элементов в принимаемом сообщении больше **count**, то возникает ошибка переполнения.

Блокировка при приеме данных гарантирует, что после возврата из процедуры **MPI\_Recv** все элементы сообщения уже будут приняты и расположены в буфере **buf**.

# MPI

## Обмен сообщениями чётных и нечётных процессов:

```
#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;
```

# MPI

```
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);
printf("process %d a = %d, b = %d\n", rank, a, b);
MPI_Finalize();
}
```

# MPI

Вместо аргументов **source** и **msgtag**  
МОЖНО ИСПОЛЬЗОВАТЬ КОНСТАНТЫ:

- **MPI\_ANY\_SOURCE** — признак того, что подходит сообщение от любого процесса;
- **MPI\_ANY\_TAG** — признак того, что подходит сообщение с любым идентификатором.

# MPI

Параметры принятого сообщения всегда можно определить по соответствующим элементам структуры **status**:

- **status.MPI\_SOURCE** — номер процесса-отправителя;
- **status.MPI\_TAG** — идентификатор сообщения;
- **status.MPI\_ERROR** — код ошибки.



# MPI

Если при приеме сообщения пользователя не интересует заполнение структуры (массива) **status**, то вместо соответствующего аргумента можно указать predetermined константу **MPI\_STATUS\_IGNORE**. Это также позволит сэкономить немного времени, требуемого на запись соответствующих полей.

# MPI

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

Если два сообщения были одновременно отправлены разными процессами, то порядок их получения принимающим процессом заранее не определён.

# MPI

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

По значению параметра **status** определяет число **count** уже принятых (после обращения к **MPI\_Recv**) или принимаемых (после обращения к **MPI\_Probe** или **MPI\_Iprobe**) элементов сообщения типа **datatype**.

# MPI

Специальное значение **MPI\_PROC\_NULL** для несуществующего процесса. Операции с таким процессом завершаются немедленно с кодом завершения **MPI\_SUCCESS**.

```
next = rank + 1;  
if(rank == (size - 1)) next=MPI_PROC_NULL;  
MPI_Send (&buf, 1, MPI_FLOAT, next, tag,  
MPI_COMM_WORLD);
```

# MPI

## Модификации процедуры **MPI\_Send**:

- **MPI\_Bsend** — передача сообщения с буферизацией.
- **MPI\_Ssend** — передача сообщения с синхронизацией.
- **MPI\_Rsend** — передача сообщения по ГОТОВНОСТИ.

# MPI

- **MPI\_Bsend** — передача сообщения с буферизацией.

Если прием сообщения еще не был инициализирован, то сообщение будет записано в специальный буфер, и произойдет немедленный возврат. Процедура может вернуть код ошибки, если места под буфер недостаточно. О выделении массива для буферизации должен заботиться пользователь.

# MPI

```
int MPI_Buffer_attach(void* buf,  
int size)
```

Назначение массива **buf** размера **size** для использования при посылке сообщений с буферизацией. В каждом процессе может быть только один такой буфер.

Размер массива, выделяемого для буферизации, должен превосходить общий размер сообщения как минимум на величину, определяемую константой **MPI\_BSEND\_OVERHEAD**.

# MPI

```
int MPI_Buffer_detach(void*  
buf, int* size)
```

Освобождение массива **buf** для других целей. Возвращает в аргументах **buf** и **size** адрес и размер освобождаемого массива. Процесс блокируется до того момента, когда все сообщения уйдут из данного буфера.



# MPI

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

int main(int argc, char **argv)
{
    int BUFSIZE = sizeof(int) + MPI_BSEND_OVERHEAD;
    char *buf;
    int rank, ibufsize, rbuf, size;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

# MPI

```
if(rank == 0){
    MPI_Buffer_attach(malloc(BUFSIZE), BUFSIZE);
    MPI_Bsend(&rank, 1, MPI_INT, 1, 5, MPI_COMM_WORLD);
    MPI_Buffer_detach(&buf, &ibufsize);
    free(buf);
}
if(rank == 1){
    MPI_Recv(&rbuf, 1, MPI_INT, 0, 5, MPI_COMM_WORLD,
&status);
    printf("Process 1 received %d from process %d\n",
rbuf, status.MPI_SOURCE);
}
MPI_Finalize();
}
```

# MPI

- **MPI\_Ssend** — передача сообщения с синхронизацией.

Выход из процедуры произойдет только тогда, когда прием сообщения будет инициализирован процессом-получателем. Использование передачи с синхронизацией может замедлить выполнение программы, но позволяет избежать наличия в системе большого количества непринятых буферизованных сообщений.

# MPI

- **MPI\_Rsend** — передача сообщения по ГОТОВНОСТИ.

Данной процедурой можно пользоваться только в том случае, если процесс-получатель уже инициировал прием сообщения. В противном случае вызов процедуры, вообще говоря, является ошибочным и результат ее выполнения не определен.

# MPI

Гарантировать инициализацию приема сообщения перед вызовом процедуры **MPI\_Rsend** можно с помощью операций, осуществляющих явную или неявную синхронизацию процессов (например, **MPI\_Barrier** или **MPI\_Ssend**). Во многих реализациях процедура **MPI\_Rsend** сокращает протокол взаимодействия между отправителем и получателем, уменьшая накладные расходы на передачу данных.

# MPI

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

Получение в параметре **status** информации о структуре ожидаемого сообщения с блокировкой. Возврата не произойдёт, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для получения.

# MPI

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
    int rank, size, ibuf;
    MPI_Status status;
    float rbuf;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ibuf = rank; rbuf = 1.0 * rank;
    if(rank==1) MPI_Send(&ibuf, 1, MPI_INT, 0, 5,
MPI_COMM_WORLD);
    if(rank==2) MPI_Send(&rbuf, 1, MPI_FLOAT, 0, 5,
MPI_COMM_WORLD);
```

# MPI

```
if (rank==0) {  
    MPI_Probe(MPI_ANY_SOURCE, 5, MPI_COMM_WORLD, &status);  
    if(status.MPI_SOURCE == 1) {  
        MPI_Recv(&ibuf, 1, MPI_INT, 1, 5, MPI_COMM_WORLD,  
&status);  
        MPI_Recv(&rbuf, 1, MPI_FLOAT, 2, 5, MPI_COMM_WORLD,  
&status);  
    }  
    else if(status.MPI_SOURCE == 2) {  
        MPI_Recv(&rbuf, 1, MPI_FLOAT, 2, 5, MPI_COMM_WORLD,  
&status);  
        MPI_Recv(&ibuf, 1, MPI_INT, 1, 5, MPI_COMM_WORLD,  
&status);  
    }  
    MPI_Finalize();  
}
```



# MPI

Тупиковые ситуации (deadlock):

процесс 0:	процесс 1:
<b>Recv (1)</b>	<b>Recv (0)</b>
<b>Send (1)</b>	<b>Send (0)</b>

процесс 0:	процесс 1:
<b>Send (1)</b>	<b>Send (0)</b>
<b>Recv (1)</b>	<b>Recv (0)</b>

# MPI

Разрешение тупиковых ситуаций:

1.

процесс 0:    процесс 1:

**Send (1)    Recv (0)**

**Recv (1)    Send (0)**

2. Использование функции **MPI\_Sendrecv**

3. Использование неблокирующих операций

# MPI

Задание 3: Напишите программу на MPI, реализующую скалярное произведение двух распределённых между процессами векторов (без использования коллективных операций). Постройте графики зависимости времени выполнения скалярного произведения от количества процессов для разных длин векторов.

# MPI

Задание 4: Напишите программу на MPI, в которой два процесса обмениваются сообщениями, замеряется время на одну итерацию обмена, определяется зависимость времени обмена от длины сообщения.

Определите *латентность* и *максимально достижимую пропускную способность* коммуникационной сети. Постройте график зависимости времени обмена от длины сообщения.