Технология программирования MPI

Антонов Александр Сергеевич,

к.ф.-м.н., с.н.с. лаборатории Параллельных информационных технологий НИВЦ МГУ

Координаты для связи:

- E-mail: asa@parallel.ru, parallel@parallel.ru
- Тел: (495) 939-23-47
- Web: http://parallel.ru

Параллелизм:

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:

Пусть **size** – число процессов, а **rank** – номер процесса (**0<=rank<=size-1**).

Крупноблочное распараллеливание:

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

Параллелизм:

Наибольший ресурс параллелизма в программах сосредоточен в циклах!

Распределение итераций циклов:

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

Параллелизм:

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:

Рассмотрим простейший цикл:

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

Параллелизм:

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:

```
// 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:

Циклическое распределение:

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

Параллелизм:

Цели распараллеливания:

- *равномерная загрузка процессоров*
- *минимизация количества и объема необходимых пересылок данных*

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

MPI

- MPI - *Message Passing Interface*, интерфейс передачи сообщений.

- Стандарт MPI 2.2.

- Более 250 процедур.

- SPMD-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.

MPI

Каждый процесс может одновременно входить в разные коммуникаторы.

Два основных атрибута процесса: коммуникатор (группа) и номер процесса в коммуникаторе (группе).

Если коммуникатор 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 - 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

```
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

`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.

Процедуры можно вызвать до **`MPI_Init`** и после **`MPI_Finalize`**.

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

```
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:

```
#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_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Get_processor_name(name, &len);
```

MPI

```
   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

```
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

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

Модификации процедуры **`MPI_Send`**:

- **`MPI_Bsend`** — передача сообщения с буферизацией.
- **`MPI_Ssend`** — передача сообщения с синхронизацией.
- **`MPI_Rsend`** — передача сообщения по готовности.

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(buf, size)
```

Освобождение массива **`buf`** для других целей. Процесс блокируется до того момента, когда все сообщения уйдут из данного буфера.

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 коммуникаторе **comm** с заполнением структуры атрибутов приходящего сообщения **status**.

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 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

```
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

Тупиковые ситуации (deadlock):

процесс 0:	процесс 1:
`Recv(1)`	`Recv(0)`
`Send(1)`	`Send(0)`

процесс 0:	процесс 1:
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

MPI

Разрешение тупиковых ситуаций:

1.

процесс 0:	процесс 1:
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

2. Использование неблокирующих операций

3. Использование функции **`MPI_Sendrecv`**

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

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

Неблокирующая посылка сообщения. Возврат из функции происходит сразу после инициализации передачи. Переменная **`request`** идентифицирует пересылку.

MPI

Модификации функции **`MPI_Isend`**:

- **`MPI_Ibsend`** — передача сообщения с буферизацией;
- **`MPI_Issend`** — передача сообщения с синхронизацией;
- **`MPI_Irsend`** — передача сообщения по готовности.

MPI

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

Неблокирующий приём сообщения. Возврат из функции происходит сразу после инициализации передачи. Переменная **`request`** идентифицирует пересылку.

MPI

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

До завершения неблокирующей операции не следует записывать в используемый массив данных!

MPI

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

Получение информации о структуре ожидаемого сообщения без блокировки. В аргументе **`flag`** возвращается значение **`1`**, если сообщение с подходящими атрибутами уже может быть принято.

MPI

```
int MPI_Wait(MPI_Request
*request, MPI_Status *status)
```

Ожидание завершения асинхронной операции, ассоциированной с идентификатором **request.** Для неблокирующего приёма определяется параметр **status. request** устанавливается в значение **MPI_REQUEST_NULL.**

MPI

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

Ожидание завершения **`count`** асинхронных операций, ассоциированных с идентификаторами **`requests`**. Для неблокирующих приёмов определяются параметры в массиве **`statuses`**.

MPI

Обмен по кольцевой топологии при помощи неблокирующих операций:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
   int rank, size, prev, next;
   int buf[2];
   MPI_Request reqs[4];
   MPI_Status stats[4];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

MPI

```
   prev = rank - 1;
   next = rank + 1;
   if(rank==0) prev = size - 1;
   if(rank==size - 1) next = 0;
   MPI_Irecv(&buf[0], 1, MPI_INT, prev, 5, MPI_COMM_WORLD,
&reqs[0]);
   MPI_Irecv(&buf[1], 1, MPI_INT, next, 6, MPI_COMM_WORLD,
&reqs[1]);
   MPI_Isend(&rank, 1, MPI_INT, prev, 6, MPI_COMM_WORLD,
&reqs[2]);
   MPI_Isend(&rank, 1, MPI_INT, next, 5, MPI_COMM_WORLD,
&reqs[3]);
   MPI_Waitall(4, reqs, stats);
   printf("process %d prev = %d next=%d\n", rank, buf[0],
buf[1]);
   MPI_Finalize();
}
```

MPI

```
int MPI_Waitany(int count,
MPI_Request *requests, int
*index, MPI_Status *status)
```

Ожидание завершения одной из **count** асинхронных операций, ассоциированных с идентификаторами **requests**. Для неблокирующего приёма определяется параметр **status**.

MPI

Если к моменту вызова завершились несколько из ожидаемых операций, то случайным образом будет выбрана одна из них.

Параметр **`index`** содержит номер элемента в массиве **`requests`**, содержащего идентификатор завершённой операции.

MPI

```
int MPI_Waitsome(int incount,
MPI_Request *requests, int
*outcount, int *indexes,
MPI_Status *statuses)
```

Ожидание завершения хотя бы одной из **`incount`** асинхронных операций, ассоциированных с идентификаторами **`requests`**.

MPI

Параметр **`outcount`** содержит число завершённых операций, а первые **`outcount`** элементов массива **`indexes`** содержат номера элементов массива **`requests`** с их идентификаторами.

Первые **`outcount`** элементов массива **`statuses`** содержат параметры завершённых операций (для неблокирующих приёмов).

MPI

```
int MPI_Test(MPI_Request
*request, int *flag, MPI_Status
*status)
```

Проверка завершённости асинхронной операции, ассоциированной с идентификатором **`request`**. В параметре **`flag`** возвращается значение **`1`**, если операция завершена.

MPI

```
int MPI_Testall(int count,
MPI_Request *requests, int
*flag, MPI_Status *statuses)
```

Проверка завершённости **count** асинхронных операций, ассоциированных с идентификаторами **requests.**

MPI

```
int MPI_Testany(int count,
MPI_Request *requests, int
*index, int *flag, MPI_Status
*status)
```

В параметре **`flag`** возвращается значение **`1`**, если хотя бы одна из операций асинхронного обмена завершена.

MPI

```
int MPI_Testsome(int incount,
MPI_Request *requests, int
*outcount, int *indexes,
MPI_Status *statuses)
```

Аналог **MPI_Waitsome,** но возврат происходит немедленно. Если ни одна из операций не завершилась, то значение **outcount** будет равно нулю.

MPI

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

Формирование отложенного запроса на посылку сообщения. Сама операция пересылки не начинается!

MPI

Модификации функции **`MPI_Send_init`**:

- **`MPI_Bsend_init`** — формирование запроса на передачу сообщения с буферизацией;
- **`MPI_Ssend_init`** — формирование запроса на передачу сообщения с синхронизацией;
- **`MPI_Rsend_init`** — формирование запроса на передачу сообщения по готовности.

MPI

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

Формирование отложенного запроса на приём сообщения. Сама операция приёма не начинается!

MPI

```
int MPI_Start(MPI_Request *request)
```

Инициализация отложенного запроса на выполнение операции обмена, соответствующей значению параметра **`request.`** Операция запускается как неблокирующая.

MPI

```
int MPI_Startall(int count,
MPI_Request *requests)
```

Инициализация **count** отложенных запросов на выполнение операций обмена, соответствующих значениям первых **count** элементов массива **requests.** Операции запускаются как неблокирующие.

MPI

Сообщение, отправленное при помощи отложенного запроса, может быть принято любой из процедур **`MPI_Recv`** и **`MPI_Irecv`**, и наоборот.

По завершении отложенного запроса значение параметра **`request (requests)`** сохраняется и может использоваться в дальнейшем!

MPI

```
int MPI_Request_free(MPI_Request *request)
```

Удаляет структуры данных, связанные с параметром **`request. request`** устанавливается в значение **`MPI_REQUEST_NULL`**. Если операция, связанная с этим запросом, уже выполняется, то она завершится.

MPI

Схема итерационного метода с обменом по кольцевой топологии при помощи отложенных запросов:

```
prev = rank - 1;
next = rank + 1;
if (rank == 0) prev = size - 1;
if (rank == (size - 1)) next = 0;
MPI_Recv_init(&rbuf[0], 1, MPI_FLOAT, prev, tag1,
MPI_COMM_WORLD, &reqs[0]);
MPI_Recv_init(&rbuf[1], 1, MPI_FLOAT, next, tag2,
MPI_COMM_WORLD, &reqs[1]);
MPI_Send_init(&sbuf[0], 1, MPI_FLOAT, prev, tag2,
MPI_COMM_WORLD, &reqs[2]);
MPI_Send_init(&sbuf[1], 1, MPI_FLOAT, next, tag1,
MPI_COMM_WORLD, &reqs[3]);
```

MPI

```
for(…){
 sbuf[0]=…;
 sbuf[1]=…;
 MPI_Startall(4, reqs);
 ...
 MPI_Waitall(4, reqs, stats);
 ...
}
MPI_Request_free(&reqs[0]);
MPI_Request_free(&reqs[1]);
MPI_Request_free(&reqs[2]);
MPI_Request_free(&reqs[3]);
```

MPI

```
int MPI_Sendrecv(void *sbuf, int
scount, MPI_Datatype stype, int
dest, int stag, void *rbuf, int
rcount, MPI_Datatype rtype, int
source, int rtag, MPI_Comm comm,
MPI_Status *status)
```

MPI

Совмещённые приём и передача сообщений с блокировкой. Буферы передачи и приёма не должны пересекаться. Тупиковой ситуации не возникает!

Сообщение, отправленное операцией **`MPI_Sendrecv`**, может быть принято обычным образом, и операция **`MPI_Sendrecv`** может принять сообщение, отправленное обычной операцией.

MPI

```
int MPI_Sendrecv_replace(void
*buf, int count, MPI_Datatype
datatype, int dest, int stag,
int source, int rtag, MPI_comm
comm, MPI_Status *status)
```

Совмещённые приём и передача сообщений с блокировкой через общий буфер **`buf`**.

MPI

Обмен по кольцевой топологии при помощи процедуры **MPI_Sendrecv**:

```
prev = rank - 1;
next = rank + 1;
if (rank == 0) prev = size - 1;
if (rank == (size - 1)) next = 0;
MPI_Sendrecv(&sbuf[0], 1, MPI_FLOAT, prev,
tag2, &rbuf[0], 1, MPI_FLOAT, next, tag2,
MPI_COMM_WORLD, &status1);
MPI_Sendrecv(&sbuf[1], 1, MPI_FLOAT, next,
tag1, &rbuf[1], 1, MPI_FLOAT, prev, tag1,
MPI_COMM_WORLD, &status2);
```

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

Задание 1: напишите программу, 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.

MPI

Задание 2: Напишите программу, реализующую скалярное произведение двух распределённых между процессами векторов (без использования коллективных операций). Постройте график зависимости времени выполнения скалярного произведения от длины векторов и количества процессов.