

Средства и системы параллельного программирования

сентябрь – декабрь 2021 г.

Лектор доцент Н.Н.Попова

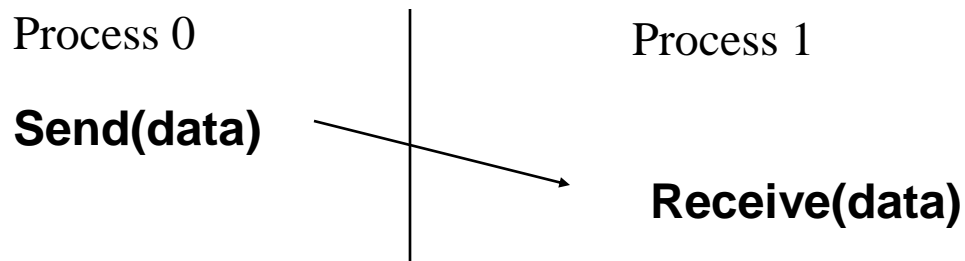
Лекция 7
18 октября 2021 г.

Тема

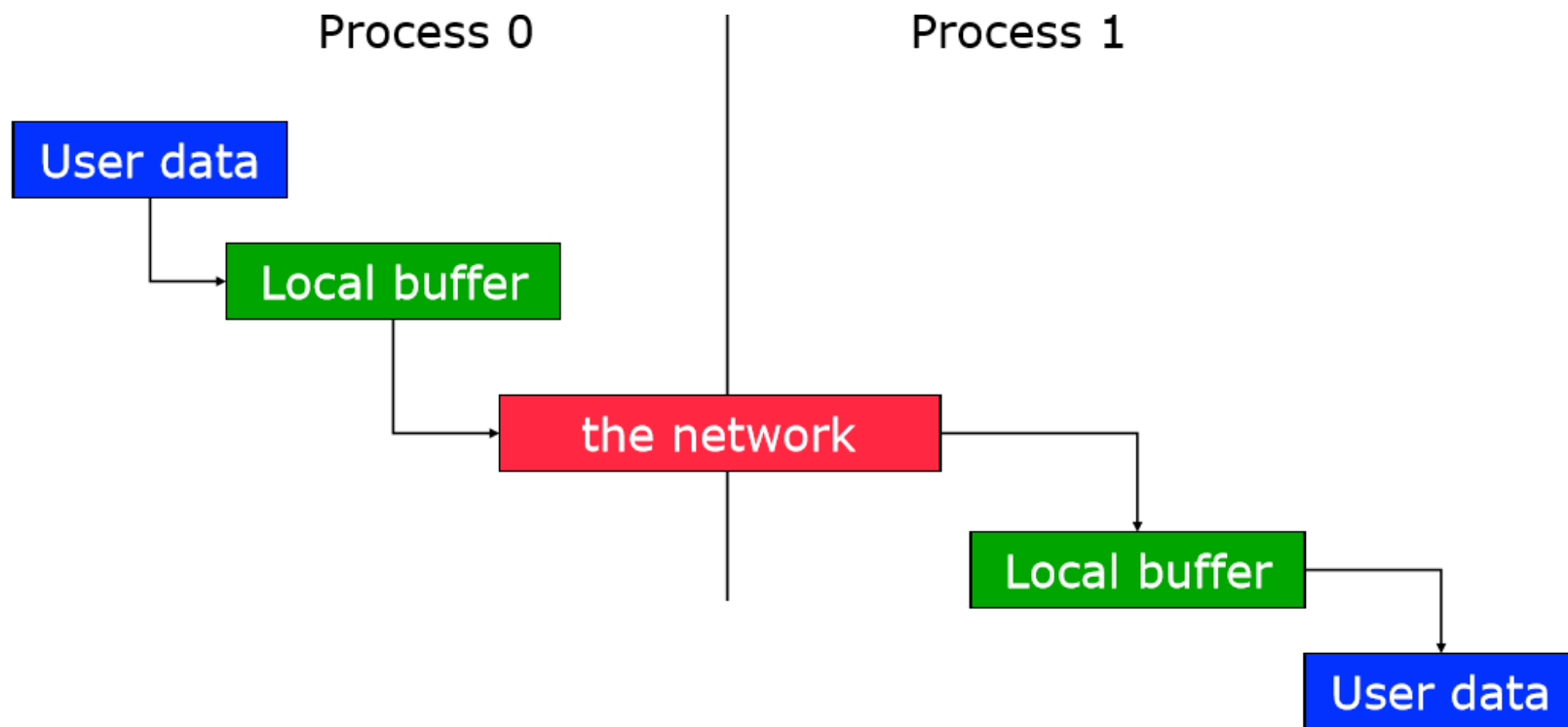
- Режимы выполнения 2-ух точечных обменов в MPI.
- Асинхронные передачи.
- Предотвращение тупиков (deadlocks).
- Профилировочный интерфейс MPI.

Основы передачи данных в MPI

- Данные посылаются одним процессом и принимаются другим.
- Передача и синхронизация совмещены.



Возможная схема выполнения операций передачи сообщений



Основа 2-точечных обменов

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

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

Константа ***MPI_STATUS_IGNORE*** может использоваться, если обработка статуса не требуется

Режимы (моды) операций передачи сообщений

- Режимы MPI-коммуникаций определяют, при каких условиях операции передачи завершаются
- Режимы могут быть блокирующими или неблокирующими
 - Блокирующие: возврат из функций передачи сообщений только по завершению коммуникаций
 - Неблокирующие (асинхронные): немедленный возврат из функций, пользователь должен контролировать завершение передач

Режимы передачи

Режим	Условие завершения
Synchronous send	Завершается только при условии инициации приема
Buffered send	Всегда завершается (за исключением ошибочных передач), независимо от приема
Standard send	Сообщение отослано (состояние приема неизвестно)
Ready send	Всегда завершается (за исключением ошибочных передач), независимо от приема
Receive	Завершается по приему сообщения

Standard send (MPI_Send)

- Критерий завершения: Не предопределен
- Завершается, когда сообщение отослано
- Можно предполагать, что сообщение достигло адресата
- Зависит от реализации

Ready send (MPI_Rsend)

- Критерий завершения: завершается немедленно, но успешно только в том случае, если процесс-получатель выставил receive
- Преимущество: немедленное завершение
- Недостатки: необходимость синхронизации
- Потенциально хорошая производительность

Buffered send (MPI_Bsend)

- Критерий завершения: завершение передачи, когда сообщение скопируется в буфер
- Преимущество: гарантировано немедленное завершение передачи (предсказуемость)
- Недостатки: надо явно выделять буфер под сообщения
- Функции MPI для контроля буферного пространства
 - `MPI_Buffer_attach`
 - `MPI_Buffer_detach`

MPI_Buffer_attach

- `int MPI_Buffer_attach(void *buffer, int size)`
- `int MPI_Buffer_detach(void *buffer)`

`MPI_Buffer_attach` устанавливает буфер для выполнения последующих вызовов буферизованных пересылок (`MPI_Bsend` или `MPI_Ibsend`). Повторный вызов этой функции возможен только после вызова `MPI_Buffer_detach`.

`MPI_Buffer_detach` должна быть вызвана после того, как необходимость в буферизации отпадет. При этом выполнение программы блокируется до завершения всех передач.

Максимальный размер возможного буфера определяется посредством `MPI_BSEND_OVERHEAD` в `mpi.h`

MPI_Buffer_attach

```
if (myrank == 0) {  
    size=bufsize+ MPI_BSEND_OVERHEAD;  
    /* Attach buffer, do buffered send, and then detach buffer */  
    buffer = (void*)malloc(size);  
    rc = MPI_Buffer_attach(buffer, size);  
    if (rc != MPI_SUCCESS)  
        { printf("Buffer attach failed. Return code= %d\n", rc);  
          MPI_Finalize(); }  
    rc = MPI_Bsend(data, NELEM, MPI_DOUBLE, dest, tag,MPI_COMM_WORLD);  
    printf("Sent message. Return code= %d\n",rc);  
    MPI_Buffer_detach(&buffer, &size);  
    free (buffer); }  
else{  
    MPI_Recv(data, NELEM, MPI_DOUBLE, source, tag, MPI_COMM_WORLD,&status);  
    printf("Received message. Return code= %d\n",rc);  
}
```

Возможные проблемы

```
MPI_Buffer_attach( buf,  bufsize+MPI_BSEND_OVERHEAD);  
for (i=1,i<n,i++) {  
    ... MPI_Bsend( bufsize bytes ... )    ...  
    /* в программе должно быть достаточное  
       количество MPI_Recv( ) */  
}  
MPI_Buffer_detach( buf, bufsize);
```

Решение проблемы

```
for (i=1,i<n,i++) {  
    MPI_Buffer_attach( buf,  bufsize+MPI_BSEND_OVERHEAD);  
    ... MPI_Bsend( bufsize bytes ... ) ...  
    /* достаточное число MPI_Recv( ); */  
    MPI_Buffer_detach( buf, bufsize);  
}
```

Buffer detach будет ждать пока сообщение не будет доставлено

Синхронный send (MPI_Ssend)

- Критерий завершения: принимающий процесс посылает подтверждение, которое должно быть получено отправителем прежде, чем send может считаться завершенным
- Используется в случаях, когда надо точно знать, что сообщение получено
- Посылающий и принимающий процессы синхронизируются независимо от того, кто работает быстрее
- Возможен простой процесса
- Самый безопасный режим работы

Сравнительная таблица

Mode	Преимущества	Недостатки
Synchronous	<ul style="list-style-type: none"> - Наиболее безопасный, поэтому наиболее переносимый - Не требуется доп. буфер 	<ul style="list-style-type: none"> - Может повлечь существенные накладные расходы на синхронизацию
Ready	<ul style="list-style-type: none"> - Наименьшие общие накладные расходы - Нет необходимости иметь дополнительный буфер - согласование SEND/RECV (handshake) не требуется 	<ul style="list-style-type: none"> - RECV <i>должен</i> предшествовать SEND
Buffered	<ul style="list-style-type: none"> - отвязывает SEND от RECV - Нет синхронизационных расходов на SEND - Программист может контролировать размер буфера 	<ul style="list-style-type: none"> - Копирование требует доп. системные расходы
Standard	<ul style="list-style-type: none"> - Подходит для многих случаев - Компромиссный вариант 	<ul style="list-style-type: none"> - Протокол определяется реализацией MPI

Проблема тупиков (deadlocks)

- Процесс 0 посылает большое сообщение процессу 1
 - Если в принимающем процессе недостаточно места в системном буфере, процесс 0 должен ждать пока процесс 1 не предоставит необходимый буфер.
 - Что произойдет:

Process 0

Process 1

Send (1)

Send (0)

Recv (1)

Recv (0)

- Называется “unsafe” потому, что зависит от системного буфера.

Deadlock

```
/* simple deadlock */
#include <stdio.h>
#include <mpi.h>
void main (int argc, char **argv) {
    int myrank;
    MPI_Status status;
    double a[100], b[100];
    MPI_Init(&argc, &argv); /* Initialize MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
    if( myrank == 0 ) {
        /* Receive, then send a message */
        MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
        MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
    }else if( myrank == 1 ) {
        /* Receive, then send a message */
        MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
        MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
    } MPI_Finalize(); /* Terminate MPI */ }
```

Без Deadlock

```
/* safe exchange */
#include <stdio.h>
#include <mpi.h>
void main (int argc, char **argv) {
    int myrank;
    MPI_Status status;
    double a[100], b[100];
    MPI_Init(&argc, &argv); /* Initialize MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
    if( myrank == 0 ) {
        MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
        MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
    } else
    if( myrank == 1 ) { /* Send a message, then receive one */
        MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
        MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
    } MPI_Finalize(); /* Terminate MPI */ }
```

Без Deadlock, но зависит от реализации

```
/* depends on buffering */
#include <stdio.h>
#include <mpi.h>
void main (int argc, char **argv) {
    int myrank;
    MPI_Status status;
    double a[100], b[100];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
    if( myrank == 0 ) {
        /* Send a message, then receive one */
        MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
        MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );}
    else if( myrank == 1 ) {
        /* Send a message, then receive one */
        MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
        MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
    }
    MPI_Finalize();
}
```

Deadlocks

Пусть каждый i -ый процесс посылает сообщение $i+1$ процессу (по модулю = число процессов) и получает сообщение от $i-1$ процесса:

```
int a[10], b[10], size, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%size, 1,
         MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+size)%size, 1,
        MPI_COMM_WORLD);
...
```

Deadlock

Deadlocks

Решение:

```
int a[10], b[10], size, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
    MPI_Send(a, 10, MPI_INT, (myrank+1)%size, 1,
             MPI_COMM_WORLD);
    MPI_Recv(b, 10, MPI_INT, (myrank-1+size)%size, 1,
             MPI_COMM_WORLD);
}
else {
    MPI_Recv(b, 10, MPI_INT, (myrank-1+size)%size, 1,
             MPI_COMM_WORLD);
    MPI_Send(a, 10, MPI_INT, (myrank+1)%size, 1,
             MPI_COMM_WORLD);
}
...
```

Совмещение посылки и приема сообщений

Для обмена сообщениями MPI обеспечивает функции:

```
int MPI_Sendrecv(void *sendbuf, int sendcount,  
MPI_Datatype senddatatype, int dest, int sendtag, void *recvbuf,  
int recvcount, MPI_Datatype recvdatatype, int source, int recvtag,  
MPI_Comm comm, MPI_Status *status)
```

Если использовать один буфер:

```
int MPI_Sendrecv_replace(void *buf, int count,  
MPI_Datatype datatype, int dest, int sendtag,  
int source, int recvtag, MPI_Comm comm,  
MPI_Status *status)
```

Пути решения «unsafe» передач

- Упорядочить передачи
- Использовать неблокирующие передачи
- Использовать совмещенные передачи (MPI_Sendrecv)
- Использовать буферизацию

Блокирующие и неблокирующие передачи

- **Блокирующие:** возврат из функций передачи сообщений только по завершению коммуникаций
- **Неблокирующие (асинхронные):** немедленный возврат из функций, пользователь должен контролировать завершение передач

Неблокирующие коммуникации

Цель – уменьшение времени работы параллельной программы за счет совмещения вычислений и обменов.

Неблокирующие операции завершаются, не дожидаясь окончания передачи данных. В отличие от аналогичных блокирующих функций изменен критерий завершения операций – немедленное завершение.

Проверка состояния передач и ожидание завершения передач выполняются специальными функциями.

Неблокирующие функции


Non-Blocking Operation	MPI функции
Standard send	MPI_Isend
Synchronous send	MPI_Issend
Buffered send	MPI_Ibsend
Ready send	MPI_Irsend
Receive	MPI_Irecv

Параметры неблокирующих операций

Datatype	Тип MPI_Datatype
Communicator	Аналогично блокирующим (тип MPI_Comm)
Request	Тип MPI_Request

- Параметр request задается при инициации неблокирующей операции
- Используется для проверки завершения операции

Совмещение блокирующих и неблокирующих операций

- Send и receive могут блокирующими и неблокирующими
- Блокирующий send может соответствовать неблокирующему receive, и наоборот, например,
MPI_Isend  ***MPI_Recv***
- Неблокирующий send может быть любого типа – synchronous, buffered, standard, ready

Форматы неблокирующих функций

MPI_Isend(buf, count, datatype, dest, tag, comm, request)

MPI_Irecv(buf, count, datatype, source, tag, comm, request)

request – “квитанция» о завершении передачи.

Тип: MPI_Request

MPI_REQUEST_NULL – нулевое значение request

MPI_Wait() ожидание завершения.

MPI_Test() проверка завершения. Возвращается флаг, указывающий на результат завершения.

Множественные проверки

- Test или wait для завершения одной (и только одной) передачи:
 - int `MPI_Waitany` (...)
 - int `MPI_Testany` (...)
- Test или wait завершения всех передач:
 - int `MPI_Waitall` (...)
 - int `MPI_Testall` (...)
- Test или wait завершения всех возможных к данному моменту:
 - int `MPI_Waitsome`(...)
 - int `MPI_Testsome`(...)

Асинхронные передачи

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

*int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status, MPI_Request request)*

*int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);*

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

*int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)*
flag : true, если передача завершена

После выполнения MPI_Wait или MPI_Test значение status устанавливается в MPI_REQUEST_NULL


```
MPI_Request request;
MPI_Status status;
int request_complete = 0; // Rank 0 sends, rank 1 receives
if (rank == 0)
{ MPI_Isend(buffer, buffer_count, MPI_INT, 1, 0, MPI_COMM_WORLD,
  &request);
  // Here we do some work while waiting for process 1 to be ready
  while (has_work) {
    do_work();
    // We only test if the request is not already fulfilled
    if (!request_complete)
      MPI_Test(&request, &request_complete, &status); }
  // No more work, we wait for the request to be complete if it's not the case
  if (!request_complete) MPI_Wait(&request, &status);
}
else
{
  MPI_Irecv(buffer, buffer_count, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
  // Here we just wait for the message to come
  MPI_Wait(&request, &status); }
```

Асинхронные передачи

```
int MPI_Testany( int count, MPI_Request array_of_requests[],  
    int *index, int *flag, MPI_Status *status)
```

```
int MPI_Testall (int count, MPI_Request *array_of_requests, int *flag,  
    MPI_Status *array_of_statuses);
```

```
int MPI_Testsome (int incount, MPI_Request *array_of_reqs, int  
    *outcount, int *array_of_indices, MPI_Status *array_of_statuses);
```

В качестве значения параметра status может использоваться константа *MPI_STATUSES_IGNORE*

После выполнения *MPI_Wait* или *MPI_Test* с флагом true значение *request* устанавливается в *MPI_REQUEST_NULL*

Асинхронные передачи

int ***MPI_Waitany***(*int* count, *MPI_Request* array_of_requests[],
int *index, *int *flag*, *MPI_Status *status*)

int ***MPI_Waitall*** (*int* count, *MPI_Request *array_of_requests*, *int *flag*,
*MPI_Status *array_of_statuses*);

int ***MPI_Waitsome*** (*int* incount, *MPI_Request *array_of_reqs*, *int*
**outcount*, *int *array_of_indices*, *MPI_Status *array_of_statuses*);

Свойства

- Сохраняется упорядоченность передач, задаваемая порядком вызовов асинхронных функций
- Гарантируется завершение соответствующей асинхронной передачи



Асинхронные передачи. Действия с *request*.

int MPI_Request_free (*MPI_Request *request*)

Установка значения параметра *request* в значение ***MPI_REQUEST_NULL***. Если операция, связанная с запросом, выполняется, то она будет завершена.

int MPI_Request_get_status (*MPI_Request *request*, *int *flag*, *MPI_Status *status*)

Получение информации об асинхронной операции, ассоциированной с *request*, БЕЗ ОСВОБОЖДЕНИЯ соответствующих структур данных. Значение параметра *request* не меняется.

Асинхронные передачи. Отмена асинхронной операции.

*int **MPI_Cancel** (MPI_Request *request)*

Инициация отмены операции, ассоциированной с *request*.

Дождаться завершения следует с использованием соответствующих функций `MPI_Request_free`, `MPI_Wait`, `MPI_Test` или их производных.

*int **MPI_Test_cancelled** (MPI_Status *status, int *flag)*

flag устанавливается в 1, если операция была успешно отменена, 0 – в противном случае.

Пример. Пересылка по кольцу с использованием асинхронных передач (1)

```
/******  
#include "mpi.h"  
#include <stdio.h>  
#include <stdlib.h>  
  
int main (int argc, char *argv[])  
{  
    int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;  
    MPI_Request reqs[4];  
    MPI_Status stats[4];  
  
    MPI_Init(&argc,&argv);  
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Пример. Пересылка по кольцу с использованием асинхронных передач (2)

```
*****/  
  
prev = rank-1;  
next = rank+1;  
if (rank == 0) prev = numtasks - 1;  
if (rank == (numtasks - 1)) next = 0;  
  
MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);  
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);  
  
MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);  
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);  
  
MPI_Waitall(4, reqs, stats);  
printf("Task %d communicated with tasks %d & %d\n", rank, prev, next);  
MPI_Finalize();  
}
```