Лабораторная работа №3

Блокирующий и неблокирующий обмен. Дедлоки.

Цель работы: получить навыки работы в MPI с блокирующими и неблокирующими обменами данных.

Теоретические положения

Попарная коммуникация (от точки к точке), в отличие от коллективной коммуникации (содержащей группу задач), включает передачу сообщения между одной парой процессов.

В MPI основной контроль за тем, как система управляет сообщением, отдан программисту, которые выбирает **способ коммуникации** в момент выбора функции отправки:

- стандартный режим, при котором на время выполнения функции процесс – отправитель сообщения блокируется, а после завершения функции буфер может быть использован повторно.Состояние отправленного сообщения может быть различным – сообщение может располагаться на процессе-отправителе, может находиться в состоянии передачи, может храниться на процессе-получателе или же может быть принято процессом-получателем при помощи функции MPI_Recv.
- синхронный (synchronous) режим состоит в том, что завершение функции отправки сообщения происходит только при получении от процесса -получателя подтверждения о начале приема отправленного сообщения. Отправленное сообщение или полностью принято процессом-получателем, или находится в состоянии приема;
- буферизованный (buffered) режим предполагает использование дополнительных системных или задаваемых пользователем буферов для копирования в них отправляемых сообщений. Функция отправки сообщения завершается сразу же после копирования сообщения в системный буфер;
- режим передачи по готовности (ready) может быть использован только, если операция приема сообщения уже инициирована. Буфер сообщения после завершения функции отправки сообщения может быть повторно использован.

Для именования функций отправки сообщения для разных режимов выполнения в MPI применяется название функции MPI_Send, к которому как префикс добавляется начальный символ названия соответствующего режима работы, т.е.:

- MPI_Ssend – функция отправки сообщения в синхронном режиме;
- MPI_Bsend – функция отправки сообщения в буферизованном режиме;
- MPI_Rsend – функция отправки сообщения в режиме по готовности.

Список параметров всех перечисленных функций совпадает с составом параметров функции MPI_Send.

Для применения буферизованного режима передачи может быть создан и передан MPI буфер памяти, используемая для этого функция имеет вид:

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

- buf — адрес буфера памяти;
- size — размер буфера.

После завершения работы с буфером он должен быть отключен от MPI при помощи функции:

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

- buf — адрес буфера памяти;
- size — возвращаемый размер буфера.

По практическому использованию режимов можно привести следующие рекомендации:

- стандартный режим обычно реализуется как буферизированный или синхронный, в зависимости от размера передаваемого сообщения, и зачастую является наиболее оптимизированным по производительности;
- режим передачи по готовности формально является наиболее быстрым, но используется достаточно редко, т. к. обычно сложно гарантировать готовность операции приема;
- буферизованный режим также выполняется достаточно быстро, но может приводить к большим расходам ресурсов (памяти), – в целом может быть рекомендован для передачи коротких сообщений;
- синхронный режим является наиболее медленным, т.к. требует подтверждения приема, однако не нуждается в дополнительной памяти для хранения сообщения. Этот режим может быть рекомендован для передачи длинных сообщений.

В дополнение к выбору способа коммуникации программист должен решить, будут ли вызовы отправки и получения блокирующими или неблокирующими. Блокирующая

или неблокирующая отправка может быть спарена с блокирующим или неблокирующим получением. Для функции приема MPI_Recv не существует различных режимов работы.

Функции MPI_Send и MPI_Recv являются **блокирующими**. Возвращение из MPI_Send не происходит до тех пор, пока данные сообщения не будут скопированы во внутренний буфер MPI. MPI_Recv возвращает управление только после того, как сообщение было принято. Поэтому, следующая программа содержит **дедлок** - состояние, при котором ни один процесс не может продолжить свою работу.

Пример

```
#include <mpi.h>
#include <stdio.h >
#include <unistd.h>
int main(int argc, char** argv)
{
        int rank,size;
        char hostname[100];
        MPI_Status st;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        if (size!=2)
        {
                printf("This example should be run on 2
processors, now exiting\n");
                MPI_Finalize();
                return 0;
        }
        else
        {
                MPI_Comm_rank(MPI_COMM_WORLD,&rank);
                gethostname(hostname, 100);
                MPI_Send(hostname, 100, MPI_CHAR, !rank, 1,
MPI_COMM_WORLD);
                MPI_Recv(hostname, 100, MPI_CHAR, !rank, 1,
MPI_COMM_WORLD,&st);
```

```
                printf("I am %d of %d. My workmate is %s\n",
rank,size,hostname);
                MPI_Finalize();
                return 0;
        }
}
```

Оба процесса вызовут функцию `MPI_Send`.При больших размерах буфера возврат из этой функции произойдет только после того, как другой процесс инициирует прием сообщения. Но другой процесс сам вызвал `MPI_Send`! Поэтому необходимо проявлять осторожность при работе с блокирующими функциями.

Приведенный пример демонстрирует одно из необходимых условий возникновения мертвой блокировки. Всего выделяют 4 необходимых условия:

1. Взаимное исключение;
2. Циклическое ожидание;
3. Дозахват ресурса;
4. Временное освобождение ресурса.

Приведенные условия являются необходимыми и нарушение одного из условий не приведет к возникновению дедлока. Четвертое условие может приводить к одной из разновидностей тупиков – живая блокировка (livelock). В этой ситуации помимо того, что работа «стоит», происходит потребление вычислительных ресурсов.

Совмещенные прием и передача сообщений

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

где

`sendbuf` - адрес отправляемых данных

`sendcount` - число отправляемых переменных

`sendtype` - тип отправляемых данных

`dest` - ранг назначения

`sendtag` - тэг отправляемого сообщения

`recvcount`- число принимаемых данных.

recvtype- тип принимаемых данных

source - от кого принимается сообщение

recvtag- тэг принимаемого сообщения

comm - коммуникатор

Выходные параметры

recvbuf- адрес принимаемых данных

status - статус према сообщения

Буфера sendbuf и recvbuf не должны пересекаться. Если необходимо обменяться данными, то следует использовать функцию MPI_Sendrecv_replace

```
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)
```

Входные параметры

count - Число передаваемых данных

datatype - Тип передаваемых данных

dest - ранг получателя

sendtag - тэг отправляемого сообщения

source - ранг отправителя

recvtag- тэг принимаемого сообщения

comm - коммуникатор, в котором происходит передача сообщения.

Выходные параметры

buf - initial address of send and receive buffer (choice)

status - status object (Status)

Пример

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv)
{
        int rank,size;
        int data;
        MPI_Status st;
```

```
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        if (size!=2)
        {
                printf("This example should be run on 2
processors, now exiting\n");
                MPI_Finalize();
                return 0;
        }
        else
        {
                MPI_Comm_rank(MPI_COMM_WORLD,&rank);
                data=100*rank;
                printf("I am %d of %d. Before %d\n",
rank,size,data);
                MPI_Sendrecv_replace(&data, 1, MPI_INT, !rank,
0,!rank,0, MPI_COMM_WORLD,&st);
                printf("I am %d of %d. After %d\n",
rank,size,data);
                MPI_Finalize();
                return 0;
        }
}
```

Неблокирующие операции передачи сообщений

В отличие от блокирующих операций, возврат управления из неблокирующих операций производится немедленно. Поэтому необходимо следить за тем, чтобы не изменить данные до того, как они отправились, либо за тем, чтобы не начать обработку еще не пришедших данных. Для того, чтобы узнать, пришло (отправилось) ли полностью сообщение, используется специальная функция. Ниже следует описание необходимых функций.

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

int MPI_Issend(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *request),

int MPI_Ibsend(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *request),

int MPI_Irsend(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *request),

int MPI_Irecv(void *buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Request *request).

Вызов неблокирующей функции приводит к инициации запрошенной операции передачи, после чего выполнение функции завершается и процесс может продолжить свои действия. Перед своим завершением неблокирующая функция определяет переменную request, которая далее может использоваться для проверки завершения инициированной операции обмена.

Проверка состояния выполняемой неблокирующей операции передачи данных производится при помощи функции:

int MPI_Test(MPI_Request *request, int *flag, MPI_status *status), где

- request — дескриптор операции, определенный при вызове неблокирующей функции;
- flag — результат проверки (true, если операция завершена);
- status — результат выполнения операции обмена (только для завершенной операции).

Операция проверки является неблокирующей, т.е. процесс может проверить состояние неблокирующей операции обмена и продолжить далее свои вычисления, если по результатам проверки окажется, что операция все еще не завершена. Возможная схема совмещения вычислений и выполнения неблокирующей операции обмена может состоять в следующем:

```
MPI_Isend(buf, count, type, dest, tag, comm, &request);
.....
do {
        ....
        MPI_Test(&request, &flag, &status);
} while (!flag);
```

Если при выполнении неблокирующей операции окажется, что продолжение вычислений невозможно без получения передаваемых данных, то может быть использована блокирующая операция ожидания завершения операции:

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

где

- request — дескриптор операции, определенный при вызове неблокирующей функции;
- status — результат выполнения операции обмена (только для завершенной операции).

Кроме рассмотренных, MPI содержит ряд дополнительных функций проверки и ожидания неблокирующих операций обмена:

- MPI_Testall — проверка завершения всех перечисленных операций обмена;
- MPI_Waitall — ожидание завершения всех операций обмена;
- MPI_Testany — проверка завершения хотя бы одной из перечисленных операций обмена;
- MPI_Waitany — ожидание завершения любой из перечисленных операций обмена;
- MPI_Testsome — проверка завершения каждой из перечисленных операций обмена;
- MPI_Waitsome — ожидание завершения хотя бы одной из перечисленных операций обмена и оценка состояния по всем операциям.

Ход работы:

Составить программу с использованием блокирующих и неблокирующих операций согласно варианту.

Вариант	**Операции с векторами**
1	A = SORT(B) + SORT(C)
2	C = A + B
3	C = A – B*x
4	C = A + SORT(B)
5	C = SORT(A) - SORT(B)
6	a = (B*C)
7	b = (A*SORT(C))
8	a = MAX(B)
9	b = MIN(A+C)
10	A = B*MIN(C)
11	c = MAX(A)*(A*B)
12	A = B + C – D*e

Вариант	Операции с векторами
13	C = A - B + D
14	SORT(A + B) - C
15	d = MAX(A + B + C)
16	d = ((A + B)* C)
17	d = (A * (B + C))
18	d = (A*B) + (C*B)
19	d = MAX(B - C) + MIN(A + B)
20	D = MIN(A + B) * (B – C + X)
21	D = SORT(A) + SORT(B) - SORT(C)
22	d = (B*C) - (A*B) +(C*B)
23	E = A + B + C – D*e
24	E = A + C – B*e + D
25	e = ((A +B)*(C + D))

Обеспечить выполнение операций в нескольких процессах. Раздача исходных данных должна выполняться с использованием неблокирующих операций, а сбор результатов в «главный» процесс – с помощью блокирующих. Оцените ускорение.

Объясните, что такое «мёртва блокировка» (дэдлок), при каких условиях она может появиться в вашей программе и как таких ситуаций избегать.

Условные обозначения

a - скалярная величина

A - вектор размерности N

MA - матрица размерности NxN

a*B - произведение вектора на скаляр

a*MB - произведение матрицы на скаляр

(A*B) - скалярное произведение векторов A и B

(MA*MB) - произведение матриц MA и MB

(MA*B) - произведение матрицы на вектор

SORT(A) - сортировка вектора A по возрастанию

MIN(A) - поиск минимального элемента вектора

MAX(A) - поиск максимального элемента вектора

TRANS(MA) - транспонирование матрицы MA

MAX(MA) - поиск максимального элемента матрицы

MIN(MA) - поиск минимального элемента матрицы

SORT(MA) - сортировка строк матрицы по убыванию