

Введение в параллельные вычисления

Лекция 8. MPI (часть 2)

КС-40, КС-44
РХТУ

Преподаватель
Митричев Иван Игоревич, к.т.н.,
ассистент кафедры ИКТ

2017

Пример MPI_Send/MPI_Recv

Каждый процесс с четным номером посылает сообщение соседу с номером на 1 большим.

Поставлена проверка для процесса с максимальным номером он не посылает сообщение несуществующему процессу.

Значение b изменяется только на процессах с нечетными номерами.

```
process 1 a = 1 b = 0
process 0 a = 0 b = -1
process 2 a = 2 b = -1
```

```
#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;
    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);
    cout << " process " << rank << " a = " << a << "
    b = " << b << endl;
    MPI_Finalize();
}
```

108_01.cpp

Выборочный прием сообщения

При приеме сообщений вместо аргументов SOURCE и MSGTAG можно использовать предопределенные константы:

- MPI_ANY_SOURCE – признак, что подходит сообщение от любого процесса;
- MPI_ANY_TAG – признак, что подходит сообщение с любым идентификатором.

При одновременном использовании будет принято сообщение с любым идентификатором от любого процесса.

О статусе сообщения

Атрибуты принятого сообщения можно определить по элементам массива status.

Параметр status – **структура** предопределенного типа MPI_Status с полями MPI_SOURCE (реальный ранг сообщения), MPI_TAG (реальный тег) и MPI_ERROR (код ошибки).

Номер принимающего процесса требуется указать явно.

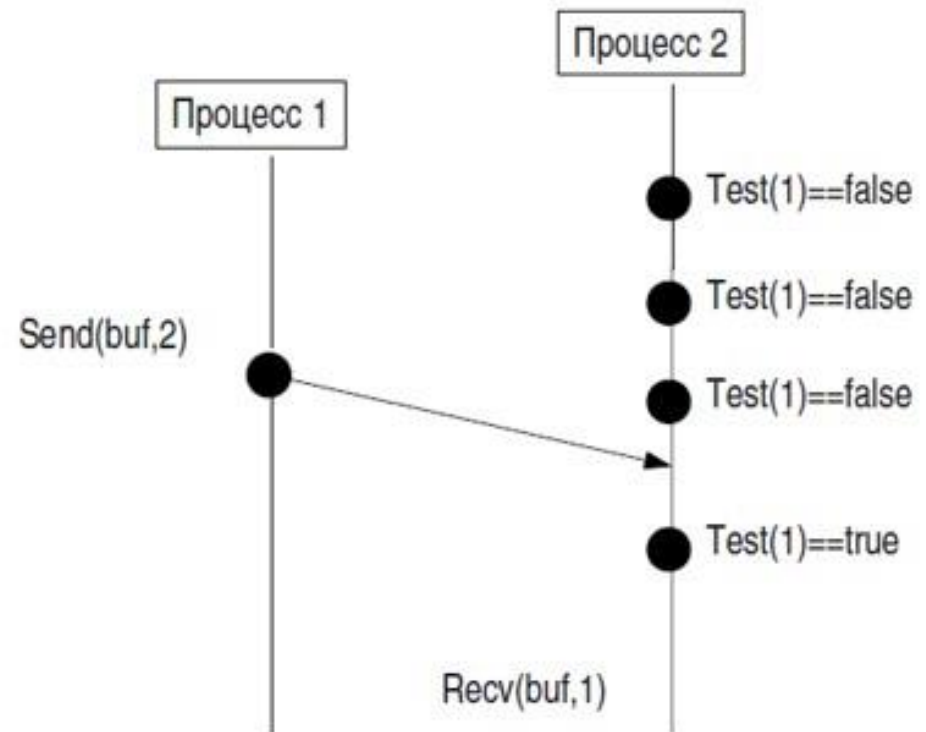
Если один процесс посылает два сообщения, соответствующие одному MPI_Recv, другому процессу, то первым принимается первое сообщение.

Если сообщение отправлено разными процессами, то порядок получения не определен.

Неблокирующий обмен сообщениями

Асинхронная передача данных.
Возврат сразу после вызова без
остановки работы процессов.

Для завершения асинхронного
обмена требуются
дополнительные процедуры, с
целью использования буфера.
До завершения неблокирующей
операции нельзя записывать в
используемый массив данных.



Прием/передача сообщений без блокировки

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

OUT *request* – идентификатор асинхронной передачи

Передача аналогична *MPI_Send*, но возврат из подпрограммы сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере *buf*. Нельзя повторно использовать буфер без получения дополнительной информации о завершении данной посылки.

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

Предусмотрены три дополнительных варианта, подобные модификациям процедуры *MPI_Send*.

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

OUT *request* – идентификатор асинхронного приема сообщения

Прием сообщения аналогичный *MPI_Recv*, однако возврат из подпрограммы происходит сразу после инициализации процесса приема без ожидания получения сообщения в буфере *buf*.

Окончание процесса приема можно определить с помощью параметра *request* и процедур *MPI_Wait* и *MPI_Test*.

Отличие в параметрах от блокирующего обмена: Request вместо Status.

Завершение асинхронного обмена

int MPI_Wait(MPI_Request *request, MPI_Status *status)

request – идентификатор асинхронного приема или передачи;

OUT *status* – параметры сообщения.

Ожидание завершения асинхронных процедур *MPI_Isend* или *MPI_Irecv*, ассоциированных с идентификатором *request*. В случае приема, атрибуты и длину полученного сообщения можно определить с помощью параметра *status*.

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

requests – массив идентификаторов асинхронного приема или передачи;

OUT *statuses* – параметры сообщений.

Выполнение процесса блокируется до тех пор, пока все операции обмена, ассоциированные с указанными идентификаторами, не будут завершены. Если во время одной или нескольких операций обмена возникли ошибки, то поле ошибки в элементах массива *statuses* будет установлено в соответствующее значение.

Неблокирующий обмен - пример

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

int main(int argc, char **argv)
{
    int rank, size, prev, next;
    int buf[1];
    MPI_Request reqs[2];
    MPI_Status stats[2];
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Irecv(&buf[0], 1, MPI_INT, prev, 5, MPI_COMM_WORLD, &reqs[0]);
    MPI_Isend(&rank, 1, MPI_INT, next, 5, MPI_COMM_WORLD, &reqs[1]);
    MPI_Waitall(2, reqs, stats);
    cout << " process " << rank << buf[0] << endl;
    MPI_Finalize();
}
```

Top50

N	Место	Кол-во CPU/ядер	Архитектура (тип процессора / сеть)	Производительность (Tflop/s)		Разработчик
				Linpack	Пиковая	
1	Москва Московский государственный университет имени М.В.Ломоносова 2016 г.	1472/42688	узлов: 1472 (Xeon E5-2697v3 [Acc: Tesla K40M] 2.6 GHz 64 GB RAM) сеть: Infiniband FDR/Infiniband FDR/Gigabit Ethernet	2,102.00	2,962.30	Т-Платформы
2	Москва Московский государственный университет имени М.В.Ломоносова 2012 г.	12422/82468	узлов: 4160 (2xXeon 5570 2.93 GHz 12 GB RAM) узлов: 777 (2xXeon E5630 [Acc: 2xTesla X2070] 2.53 GHz 12 GB RAM) узлов: 640 (2xXeon 5670 2.93 GHz 24 GB RAM) узлов: 288 (2xXeon E5630 [Acc: 2xTesla X2070] 2.53 GHz 24 GB RAM) узлов: 260 (2xXeon 5570 2.93 GHz 24 GB RAM) узлов: 40 (2xXeon 5670 2.93 GHz 48 GB RAM) узлов: 30 (2xPowerXCell 8i 3.2 GHz 16 GB RAM) узлов: 4 (4xXeon E7650 2.26 GHz 512 GB RAM) сеть: Infiniband QDR/Gigabit Ethernet/Gigabit Ethernet	901.90	1,700.21	Т-Платформы
3	Санкт-Петербург Суперкомпьютерный центр Санкт- Петербургский политехнический университет 2017 г.	1468/20552	узлов: 623 (2xXeon E5-2697v3 2.6 GHz 64 GB RAM) узлов: 56 (2xXeon E5-2697v3 [Acc: 2x NVIDIA K40] 2.6 GHz 64 GB RAM) узлов: 36 (2xXeon E5-2697v3 2.6 GHz 128 GB RAM) узлов: 8 (2xXeon E5-2697v3 [Acc: NVIDIA K1] 2.6 GHz 128 GB RAM) узлов: 8 (2xXeon E5-2697v3 [Acc: NVIDIA K2] 2.6 GHz 128 GB RAM) узлов: 3 (2xXeon E5-2697v3 2.6 GHz 256 GB RAM) сеть: Infiniband FDR/Gigabit Ethernet/Gigabit Ethernet	715.94	1,015.10	Группа компаний РСК
4	Москва МЦ РАН 2016 г.	416/28704	узлов: 208 (2xXeon E5-2690 [Acc: 2x Xeon Phi 7110X] 2.9 GHz 80 GB RAM) сеть: Infiniband FDR/Gigabit Ethernet/Fast Ethernet	383.21	523.83	Группа компаний РСК
5	Москва Суперкомпьютерный вычислительный комплекс НИЦ "Курчатовский институт" 2015 г.	296/10064	узлов: 148 (2xXeon E5-2650v2 [Acc: 2x Tesla K80] 2.6 GHz 128 GB RAM) сеть: Infiniband FDR/Gigabit Ethernet/Gigabit Ethernet	381.40	601.00	Т-Платформы
6	Москва Центр обработки данных НИЦ "Курчатовский институт" 2015 г.	774/11082	узлов: 364 (2xXeon E5-2680v3 2.5 GHz 128 GB RAM) узлов: 23 (2xXeon E5-2680v3 [Acc: 3x Nvidia K80] 2.5 GHz 128 GB RAM) сеть: Infiniband FDR/Gigabit Ethernet/Gigabit Ethernet	374.13	500.55	SuperMicro, Борлас

Режимы коммуникации (обмена сообщениями)

Определяют, когда завершается (а также, и когда начинается) прием и отправка сообщений.

Отправка сообщения

1. Стандартный (standard) – сообщения могут буферизироваться или нет в зависимости от наличия свободного места в буфере, могут не буферизироваться с целью оптимизации производительности. Является нелокальным, то есть использует информацию о наличии ожидающих запросов на получение сообщения (receive). Без префикса
2. Буферизованный (buffered) – сообщения обязательно буферизуются. Является локальным, то есть вызов завершается независимо от того, есть ли ожидающие запросы на получение сообщения (receive). Префикс R у функций (MPI_RSEND)
3. Синхронный (synchronous) – отправка завершается только тогда, когда есть соответствующий запрос на получение. Префикс S (MPI_SSEND)
4. По готовности (ready) – отправка сообщения может быть начата только, когда есть соответствующий запрос на получение. Префикс R (MPI_RSEND)

Прием сообщения – один режим. Когда сообщение поступает в буфер, прием завершается

Модификации MPI_Send

В MPI_Send нет гарантии, что сообщение получено процессом dest.

MPI_Bsend – передача сообщения *с буферизацией* (сообщение записывается в буфер, не зависит от приема сообщения, код ошибки, если места в буфере недостаточно).

MPI_Ssend – передача сообщения *с синхронизацией* (выход из процедуры, если прием посылаемого сообщения инициализирован получателем; может замедлить выполнение кода).

MPI_Rsend – передача сообщения *по готовности* (используется если получатель уже инициализировал прием, например, использовав MPI_Barrier, может сократить протокол взаимодействия между отправителем и получателем).

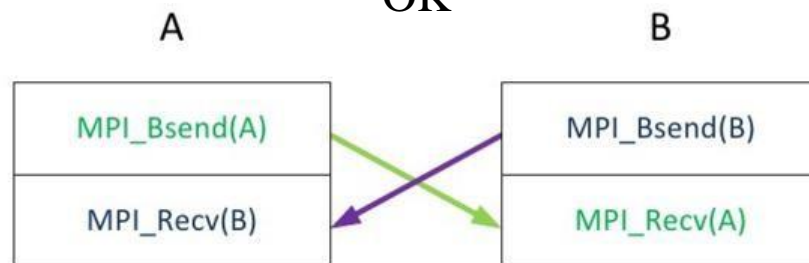
Обычно в MPI выделяется некоторый объём памяти для буферизации сообщений, рекомендуется явно выделять в программе *достаточный* буфер для всех пересылок с буферизацией.

Различные ситуации при обмене сообщениями

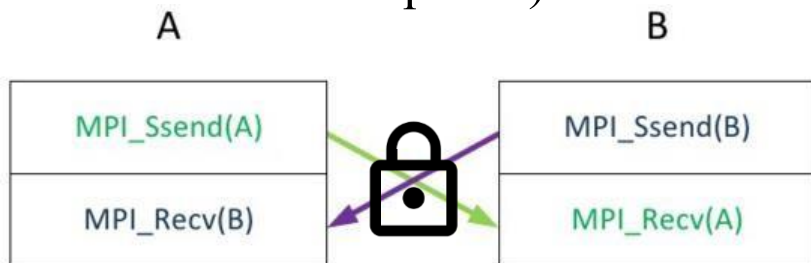
? (стандарт не
определяет, дает
гибкость)



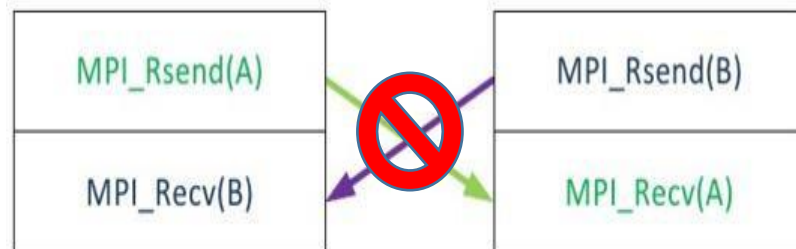
OK



Deadlock (взаимная
блокировка)




fail (передача будет
A проигнорирована) B



Определение размера сообщения

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



получить размер сообщения

- status – информация о полученном сообщении;
- datatype – тип данных, в единицах которого нужно получить размер сообщения;

Выходной параметр:

count – количество полученных элементов в единицах datatype или константа MPI_UNDEFINED, если длина данных сообщения не делится нацело на размер типа datatype.

При коллективных обменах получатель информацию о сообщении не получает.

Коллективный сбор от всех в одну переменную: MPI_Reduce

Коллективный обмен сообщениями можно считать избыточным (Богачев К.Ю.), любая программа м.б. написана без их использования.

Коллективный обмен м.б. заменен на цикл парных обменов, но при этом скорость ниже и не используется специфика вычислительной установки.

int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

sendbuf – адрес буфера с данными;

count – число элементов типа *datatype* в буфере;

op – идентификатор операции (типа *MPI_Op*), которую нужно осуществить над пересланными

данными для получения результата в буфере *recvbuf*;

root – ранг получатель в коммуникатора *comm*;

выходной параметр:

recvbuf – указатель на буфер, где требуется получить результат.

Функция должна быть вызвана во всех процессах группы *comm* с одинаковыми значениями аргументов *root*, *comm*, *count*, *datatype*, *op*.

Результат – в процессе с номером *root* в группе *comm*.

Операции над данными в MPI

Операция MPI	Значение
MPI_MAX	максимум
MPI_MIN	минимум
MPI_SUM	сумма
MPI_PROD	произведение
MPI_LAND	логическое «и»
MPI_BAND	побитовое «и»
MPI_LOR	логическое «или»
MPI_BOR	побитовое «или»
MPI_LXOR	логическое «исключающее или»
MPI_BXOR	побитовое «исключающее или»
MPI_MAXLOC	максимум и его позиция
MPI_MINLOC	минимум и его позиция

Пример: вычисление интеграла

```
#include "mpi.h"
using namespace std;
#define N 10000000
#define NTIMES 100
static double a = 0.;
static double b = 1.;
```

а и b – границы интервала;
INTEGRAL – результат;
p – число процессов;
proc_integral – функция, работающая в процессе с номером myrank;
a_p, b_p, n_p, h_p – параметры подинтервалов для текущего процесса;

```
static double INTEGRAL = 0.;
void proc_integral(double (*fun)(double), const int myrank, int p)
{
```

```
    double h_p = (b - a) / p;          int n_p = N / p; int i;
    double a_p = a + myrank * h_p;     double b_p = a_p + h_p;
    double integ = 0.;                  double x_p = a_p;
    while (x_p < b_p) {
        integ += fun(x_p) * h_p / n_p;
        x_p += h_p / n_p;
```

integ – значение интеграла в
текущем процессе;
MPI_Reduce – сложить все
ответы и передать процессу 0.

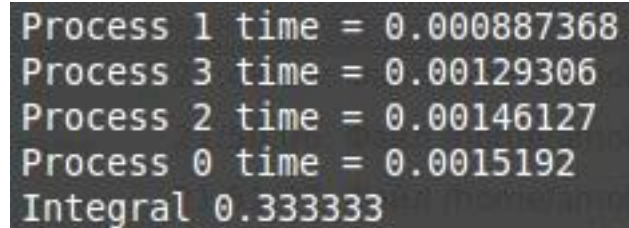
```
    }
    MPI_Reduce(&integ, &INTEGRAL, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
}
```

Пример: вычисление интеграла (продолжение)

```
double f(double x) { return x*x; }
int main(int argc, char **argv)
{
    double time_start, time_finish;    int myrank; int p;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    time_start = MPI_Wtime();
    proc_integral(f, myrank, p);
    time_finish = MPI_Wtime();
    cout << " Process " << myrank << " time = " << (time_finish-time_start)/NTIMES << endl;
    if (myrank == 0)
        cout << " Integral " << INTEGRAL << endl;
    MPI_Finalize();
    return 0;
}
```

proc_integral(f, myrank, p); –
вычисление интеграла в каждом из
процессов;

печать результата в процессе 0.



```
Process 1 time = 0.000887368
Process 3 time = 0.00129306
Process 2 time = 0.00146127
Process 0 time = 0.0015192
Integral 0.333333
```