

## Лабораторная работа №2

### Коллективные операции передачи данных

**Цель:** изучить основные принципы коллективных операций передачи данных в технологии MPI на примере использования в рамках языка C++.

Для получения **теоретических сведений** настоятельно рекомендуется при подготовке изучить материалы, представленные в списке литературы в конце разработки, а также прочие материалы по тематике лабораторной работы, представленные в открытых источниках.

Далее следует краткий конспект материала, приведенного в данных источниках, в конце включающий короткие примеры фрагментов программ.

#### 1. Передача данных от одного процесса всем процессам программы

Функции **MPI\_Send** и **MPI\_Recv**, обеспечивают возможность выполнения **парных операций** передачи данных между двумя процессами параллельной программы. Для выполнения коммуникационных **коллективных операций**, в которых принимают участие все процессы коммутатора, в MPI предусмотрен специальный набор функций.

Достижение эффективного выполнения операции передачи данных от одного процесса всем процессам программы (**широковещательная рассылка** данных) может быть обеспечено при помощи функции MPI:

```
int MPI_Bcast(void *buf,int count,MPI_Datatype type,int root,MPI_Comm comm);
```

где **buf**, **count**, **type** – буфер памяти с отправляемым сообщением (для процесса с рангом 0), и для приема сообщений для всех остальных процессов,  
**root** - ранг процесса, выполняющего рассылку данных,  
**comm** - коммутатор, в рамках которого выполняется передача данных.

Функция **MPI\_Bcast** осуществляет рассылку данных из буфера **buf**, содержащего **count** элементов типа **type** с процесса, имеющего номер **root**, всем процессам, входящим в коммутатор **comm**.

- функция **MPI\_Bcast** определяет коллективную операцию и, тем самым, при выполнении необходимых рассылок данных вызов функции **MPI\_Bcast** должен быть осуществлен всеми процессами указываемого коммутатора;
- указываемый в функции **MPI\_Bcast** буфер памяти имеет различное назначение в разных процессах. Для процесса с рангом **root**, с которого осуществляется рассылка данных, в этом буфере должно находиться рассылаемое сообщение. Для всех остальных процессов указываемый буфер предназначен для приема передаваемых данных.

#### Пример 2. Программа суммирования элементов вектора.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "mpi.h"

int main(int argc, char* argv[])
{
    double x[100], TotalSum, ProcSum = 0.0;
    int ProcRank, ProcNum, N=100;
    MPI_Status Status;
    // инициализация
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD,&ProcRank);
```

```

// подготовка данных
if ( ProcRank == 0 )
    DataInitialization(x,N);
// рассылка данных на все процессы
MPI_Bcast(x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
// вычисление частичной суммы на каждом из процессов
// на каждом процессе суммируются элементы вектора x от i1 до i2
int k = N / ProcNum;
int i1 = k * ProcRank;
int i2 = k * (ProcRank + 1);
if ( ProcRank == ProcNum-1 )
    i2 = N;
for ( int i = i1; i < i2; i++ )
    ProcSum = ProcSum + x[i];
// сборка частичных сумм на процессе с рангом 0
if ( ProcRank == 0 )
{
    TotalSum = ProcSum;
    for ( int i=1; i < ProcNum; i++ )
    {
        MPI_Recv(&ProcSum, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
            &Status);
        TotalSum = TotalSum + ProcSum;
    }
} else // все процессы отсылают свои частичные суммы
    MPI_Send(&ProcSum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
// вывод результата
if ( ProcRank == 0 )
    printf("\nTotal Sum = %10.2f",TotalSum);
MPI_Finalize();
}

```

## 2. Передача данных от всех процессов одному процессу. Операции редукции

В рассмотренной программе суммирования числовых значений имеющаяся процедура сбора и последующего суммирования данных является примером часто выполняемой коллективной **операции передачи данных от всех процессов одному процессу**. В этой операции над собираемыми значениями осуществляется та или иная обработка данных (данная операция еще именуется **операцией редукции данных**). Реализация операции редукции при помощи обычных парных операций передачи данных является неэффективной и достаточно трудоемкой. Для наилучшего выполнения действий, связанных с редукцией данных, в MPI предусмотрена функция:

```

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

```

где

- sendbuf - буфер памяти с отправляемым сообщением,
- recvbuf – буфер памяти для результирующего сообщения (только для процесса с рангом root),
- count - количество элементов в сообщениях,
- type – тип элементов сообщений,
- op - операция, которая должна быть выполнена над данными,
- root - ранг процесса, на котором должен быть получен результат,
- comm - коммунитор, в рамках которого выполняется операция.

**Таблица 2. Предопределенные виды операций MPI для функций редукции данных**

Операция	Описание
MPI_MAX	Определение максимального значения
MPI_MIN	Определение минимального значения
MPI_SUM	Определение суммы значений

MPI_PROD	Определение произведения значений
MPI_LAND	Выполнение логической операции "И" над значениями сообщений
MPI_BAND	Выполнение битовой операции "И" над значениями сообщений
MPI_LOR	Выполнение логической операции "ИЛИ" над значениями сообщений
MPI BOR	Выполнение битовой операции "ИЛИ" над значениями сообщений
MPI_LXOR	Выполнение логической операции исключающего "ИЛИ" над значениями сообщений
MPI_BXOR	Выполнение битовой операции исключающего "ИЛИ" над значениями сообщений
MPI_MAXLOC	Определение максимальных значений и их индексов
MPI_MINLOC	Определение минимальных значений и их индексов

Помимо данного стандартного набора операций могут быть определены и новые дополнительные операции непосредственно самим пользователем библиотеки MPI.

- функция **MPI\_Reduce** определяет коллективную операцию и, тем самым, вызов функции должен быть выполнен всеми процессами указываемого коммуникатора, все вызовы функции должны содержать одинаковые значения параметров **count, type, op, root, comm**;
- передача сообщений должна быть выполнена всеми процессами, результат операции будет получен только процессом с рангом **root**;
- выполнение операции редукции осуществляется над отдельными элементами передаваемых сообщений. Так, например, если сообщения содержат по два элемента данных и выполняется операция суммирования **MPI\_SUM**, то результат также будет состоять из двух значений, первое из которых будет содержать сумму первых элементов всех отправленных сообщений, а второе значение будет равно сумме вторых элементов сообщений соответственно.

После применения функции редукции к ранее рассмотренной программе суммирования, весь программный код, суммирующий результаты, полученные от отдельных процессов, может быть заменен на вызов функции **MPI\_Reduce**:

```
// сборка частичных сумм на процессе с рангом 0
MPI_Reduce(&ProcSum,&TotalSum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

### 3 Синхронизация вычислений

**Синхронизация** процессов, т.е. одновременное достижение процессами тех или иных точек процесса вычислений, обеспечивается при помощи функции MPI:

```
int MPI_Barrier(MPI_Comm comm);
```

Функция **MPI\_Barrier** должна вызываться всеми процессами используемого коммуникатора. При вызове функции **MPI\_Barrier** выполнение процесса блокируется, продолжение вычислений процесса происходит только после вызова функции **MPI\_Barrier** всеми процессами коммуникатора.

### 4. Режимы передачи данных

Функция **MPI\_Send** обеспечивает так называемый **стандартный** режим отправки сообщений, при котором:

- на время выполнения функции процесс-отправитель сообщения блокируется;
- после завершения функции буфер может быть использован повторно;
- состояние отправленного сообщения может быть различным:
  - сообщение может располагаться в процессе-отправителе;
  - может находиться в процессе передачи;
  - может храниться в процессе-получателе;
  - может быть принято процессом-получателем при помощи функции **MPI\_Recv**.

Кроме стандартного режима в MPI предусматриваются следующие дополнительные режимы передачи сообщений:

- **Синхронный** 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** режим предполагает использование дополнительных системных буферов для копирования в них отправляемых сообщений; как результат, функция отправки сообщения завершается сразу же после копирования сообщения в системный буфер;

- **Режим передачи по готовности** 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 используется название функции **MPI\_Send**, к которому как префикс добавляется начальный символ названия соответствующего режима работы:

- **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 использования буферизованного режима передачи должен быть создан и передан MPI буфер памяти для буферизации сообщений – используемая для этого функция имеет вид:

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

где

- **buf** - буфер памяти для буферизации сообщений;
- **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)
```

По практическому использованию режимов можно привести следующие рекомендации:

1. Режим передачи по готовности формально является наиболее быстрым, но используется достаточно редко, т.к. обычно сложно гарантировать готовность операции приема.
2. Стандартный и буферизованный режимы также выполняются достаточно быстро, но могут приводить к большим расходам ресурсов (памяти) – в целом может быть рекомендован для передачи коротких сообщений.
3. Синхронный режим является наиболее медленным, т.к. требует подтверждения приема. В тоже время, этот режим наиболее надежен – можно рекомендовать его для передачи длинных сообщений.

Для функции приема **MPI\_Recv** не существует различных режимов работы.

## 5. Организация неблокирующих обменов данными между процессорами

Все рассмотренные ранее функции отправки и приема сообщений являются **блокирующими**, т.е. приостанавливающими выполнение процессов до момента завершения работы вызванных функций. В то же время при выполнении параллельных вычислений часть сообщений может быть отправлена и принята заранее до момента реальной потребности в пересылаемых данных. В таких ситуациях желательно иметь возможность выполнения функций обмена данными без блокировки процессов для совмещения процессов передачи сообщений и вычислений. Такой **неблокирующий способ** выполнения обменов помогает уменьшать потери эффективности параллельных вычислений из-за коммуникационных операций.

MPI обеспечивает возможность неблокированного выполнения операций передачи данных между двумя процессами. Наименование неблокирующих аналогов образуется из названий

соответствующих функций путем добавления префикса **I (Immediate)**. Список параметров неблокирующих функций содержит весь набор параметров исходных функций и один дополнительный параметр **request** с типом **MPI\_Request** (в функции **MPI\_Irecv** отсутствует также параметр **status**):

```
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 выполняется 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)
```

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.

## 6. Одновременное выполнение передачи и приема

Достижение эффективного и гарантированного одновременного выполнения операций передачи и приема данных может быть обеспечено при помощи функции 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);
```

где

sbuf, scount, stype, dest, stag - параметры передаваемого сообщения;

rbuf, rcount, rtype, source, rtag - параметры принимаемого сообщения;

comm - коммунитор, в рамках которого выполняется передача данных;

status – структура данных с информацией о результате выполнения операции.

Функция MPI\_Sendrecv передает сообщение, описываемое параметрами (**sbuf, scount, stype, dest, stag**), процессу с рангом **dest** и принимает сообщение в буфер, определяемый параметрами (**rbuf, rcount, rtype, source, rtag**), от процесса с рангом **source**.

В функции **MPI\_Sendrecv** для передачи и приема сообщений применяются разные буфера. В случае же, когда сообщения имеют одинаковый тип, в MPI имеется возможность использования единого буфера:

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

### Лабораторные задания

**Задание.** Модифицировать программу, написанную на Л.Р. №1, так чтобы она работала на основе коллективной передачи сообщений. **Результаты работы сравнить и занести в отчет.**

### Контрольные вопросы

1. Как происходит передача данных от одного процесса всем?
2. Как происходит передача данных от всем процессов одному?
3. Какие используются в MPI для синхронизации вычислений?
4. Как организуется неблокирующий обмен данными между процессами?
5. Как организуется одновременное выполнение прием и передачи данных?

### Требования к сдаче работы

1. При подготовке изучить теоретический материал по тематике лабораторной работы, представленный в списке литературы ниже, выполнить представленные примеры, занести в отчет результаты выполнения.
2. Продемонстрировать программный код для лабораторного задания.
3. Продемонстрировать выполнение лабораторных заданий (можно в виде скриншотов).
4. Ответить на контрольные вопросы.
5. Показать преподавателю отчет.

### Литература

1. Спецификации стандарта Open MPI (версия 1.6, на английском языке):  
<http://www.open-mpi.org/doc/v1.6/>
2. Материалы, представленные на сайте intuit.ru в рамках курса «Intel Parallel Programming Professional (Introduction)»:  
<http://old.intuit.ru/departmentsupercomputing/ppintel5/>
3. С.А. Лупин, М.А. Посыпкин Технологии параллельного программирования. – М.: ИД «ФОРУМ»: ИНФРА-М, 2011. – С. 12-96. (Глава, посвященная MPI)
4. Отладка приложений MPI в кластере HPC  
[http://msdn.microsoft.com/ru-ru/library/dd560808\(v=vs.100\).aspx](http://msdn.microsoft.com/ru-ru/library/dd560808(v=vs.100).aspx)

5. [http://www.parallel.ru/tech/tech\\_dev/mpi.html](http://www.parallel.ru/tech/tech_dev/mpi.html)
6. [http://msdn.microsoft.com/ru-ru/library/ee441265\(v=vs.100\).aspx#BKMK\\_debugMany](http://msdn.microsoft.com/ru-ru/library/ee441265(v=vs.100).aspx#BKMK_debugMany)