Лабораторная работа №6. Основы MPI. Парные обмены

6.1 Цель лабораторной работы

Получить навыки разработки параллельных программ с использованием попарного взаимодействия процессов в технологии MPI.

6.2 Теоретический материал

6.2.1 Установка MS MPI и настройка проекта

По ссылке https://www.microsoft.com/en-us/download/details.aspx?id=57467 скачать и установить файл msmpisdk.msi.

Настройка MS MPI в Visual Studio

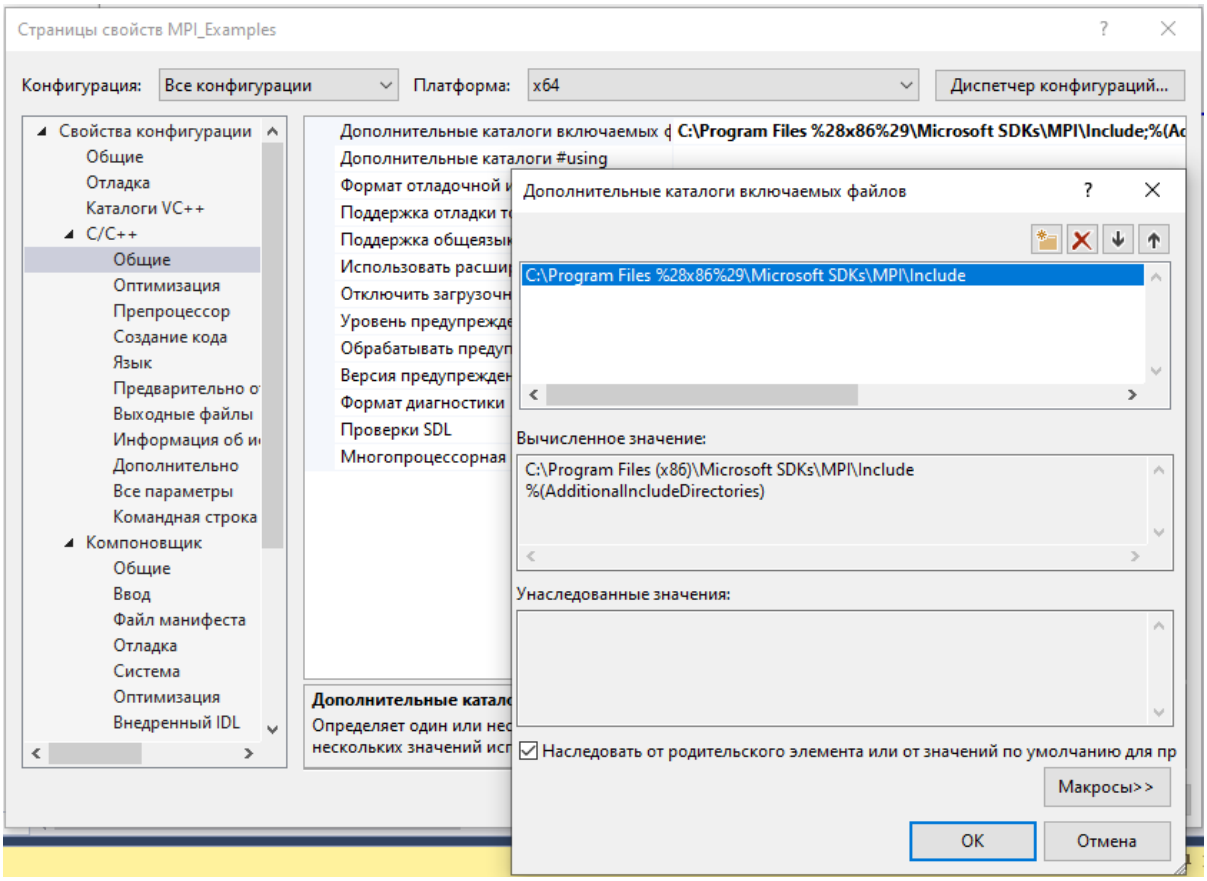
1. Создать проект. При выборе папки для хранения проекта следует учитывать, что в пути до проекта не должно быть русских букв.

2. Скопировать в программу следующий код

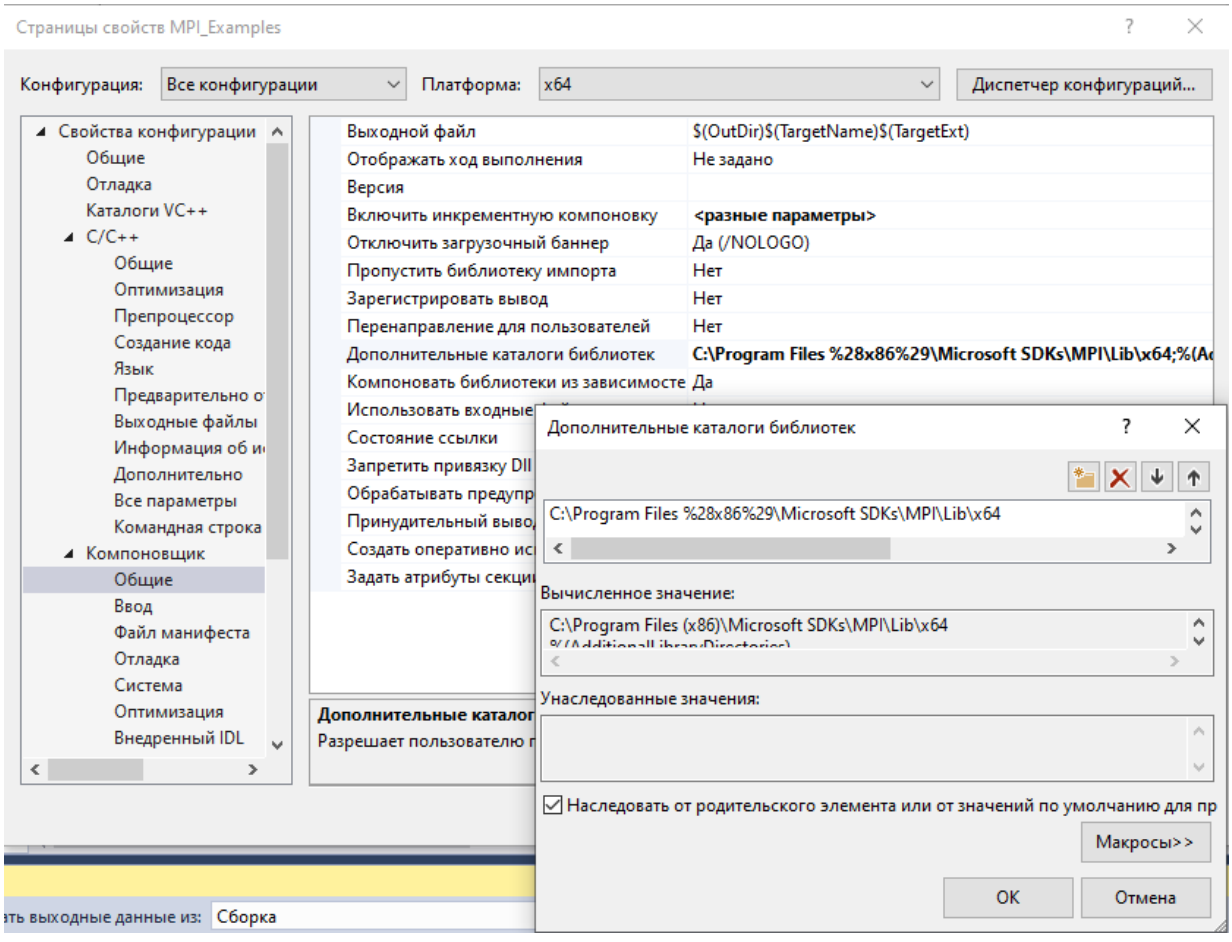
```
#include <iostream>
#include <mpi.h>

int main(int argc, char**argv)
{
   MPI_Init(&argc, &argv);
   std::cout << "Hello World!\n";
   MPI_Finalize();
}
```

3. В свойствах проекта (Проект->Свойства < имя проекта >) добавить следующие настройки:

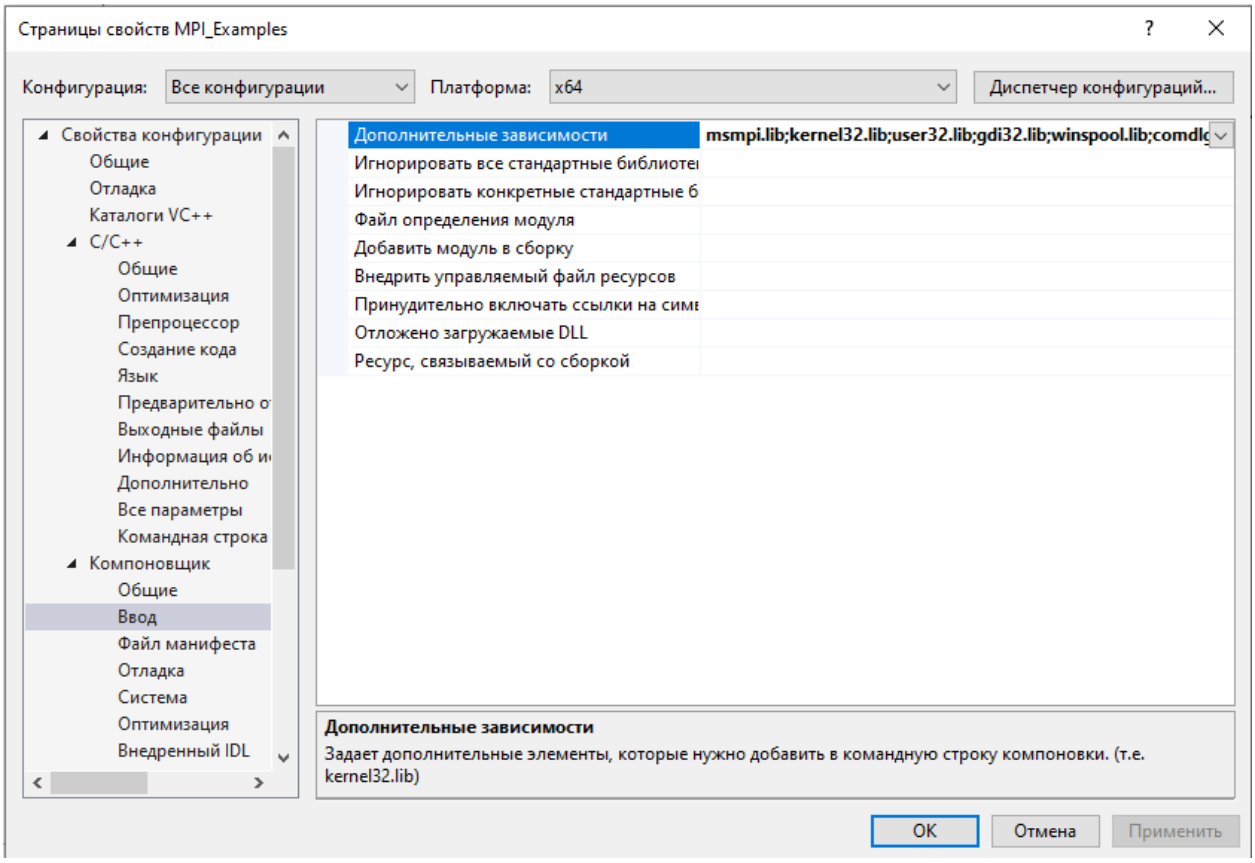
– C/C++ -> Общие -> Дополнительные каталоги включаемых файлов (...\Program Files(x86)\Microsoft SDKs\MPI\Include);



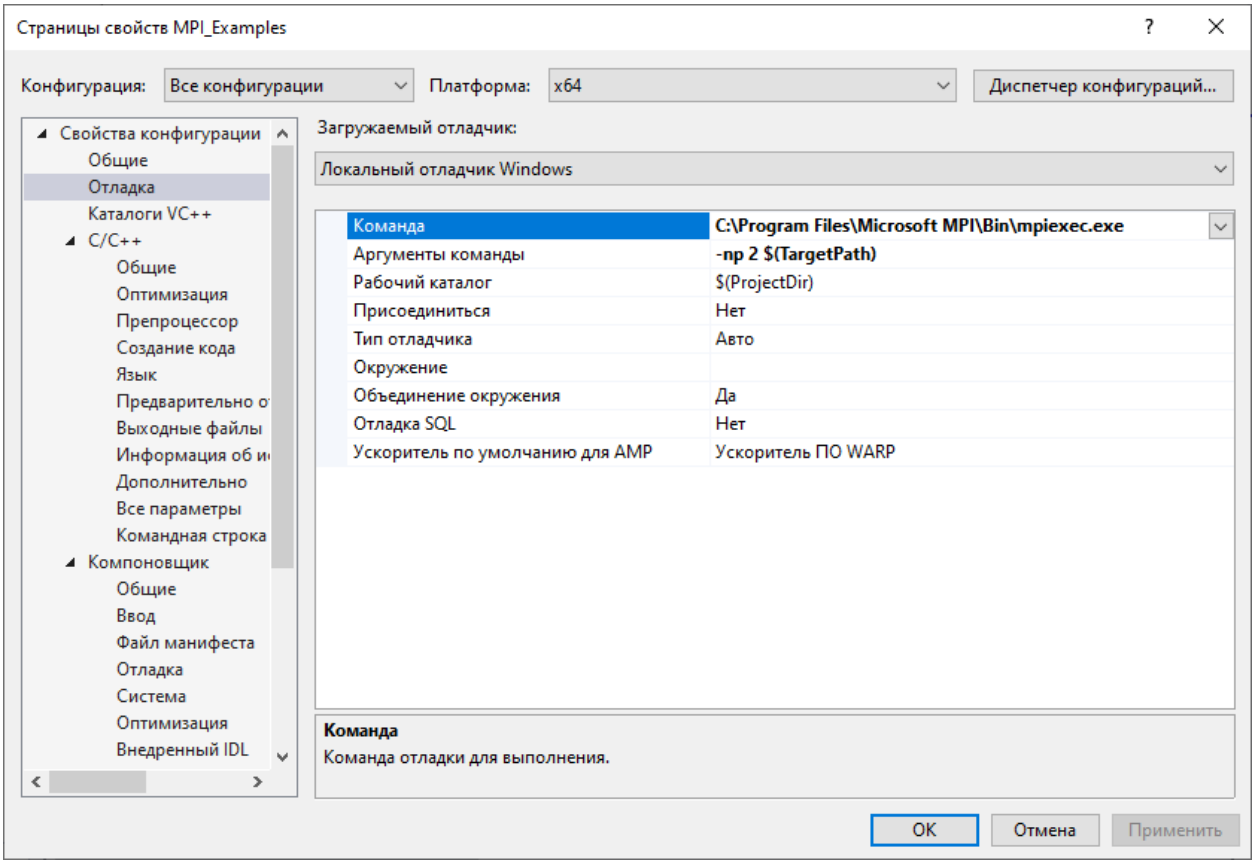
– Компоновщик -> Общие -> Дополнительные каталоги библиотек (...\Microsoft SDKs\MPI\Lib\x64);



– Компоновщик -> Ввод -> Дополнительные зависимости (добавить в начало списка msmpi.lib);



– Отладка -> Команда (...\Microsoft MPI\Bin\mpiexec.exe).



4. В свойствах проекта установить необходимое число запускаемых процессов (см. рисунок выше)

– Отладка -> аргументы команды (-np 2 $(TargetPath));

5. Запустить программу на выполнение.

6.2.2 Замер времени работы MPI-программы

Стандарт MPI включает определение специальных функций для измерения времени. Получение текущего момента времени выполнения программы обеспечивается при помощи функции `MPI_Wtime()`, результат вызова которой есть количество секунд, прошедших от некоторого определенного момента времени в прошлом.

Синтаксис указанных функций приведен ниже.

`double MPI_Wtime(void)` – возвращает количество секунд, прошедшее с некоторого определенного момента в прошлом.

Возможная схема применения функции **`MPI_Wtime`** может состоять в следующем:

```
double start, end, diff;
start = MPI_Wtime();
...
end   = MPI_Wtime();
diff  = end – start;
cout << diff << endl;
```

Точность измерения времени также может зависеть от среды выполнения параллельной программы. Для определения текущего значения точности может быть использована функция `MPI_Wtick(void)`, позволяющая определить время в секундах между двумя последовательными показателями времени аппаратного таймера используемой компьютерной системы.

6.2.3 Структура параллельной программы

Параллельная MPI-программа – множество одновременно выполняемых процессов. Процессы могут выполняться на разных процессорах, но на одном процессоре может располагаться несколько процессов. Количество процессов и число используемых процессоров задается в момент запуска параллельной

программы средствами среды исполнения MPI. Все процессы в программе пронумерованы. Номер процесса называется рангом процесса.

Перед использованием функций MPI нужно подключить библиотеку mpi.h

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

Начало параллельной области задаётся функцией

```
MPI_Init( &argc, &argv);
```

В конце должна быть

```
MPI_Finalize();
```

Нужно помнить, что у процессов нет доступа к памяти друг друга и всё общение происходит через передачу сообщений.

Для взаимодействия процессов используются коммуникаторы. Для всех процессов программы автоматически создается коммуникатор MPI_COMM_WORLD.

```
#include <mpi.h>
int main( int argc, char *argv[] )
{
   int proc_num, proc_rank;
   <программный код без использования MPI функций>
   MPI_Init( &argc, &argv);
   MPI_Comm_size( MPI_COMM_WORLD, &proc_num);
   MPI_Comm_rank( MPI_COMM_WORLD, &proc_rank);

   <программный код с использованием MPI функций>
   MPI_Finalize();
   <программный код без использования MPI функций>
}
```

Все функции MPI возвращают код ошибки. Все возвращаемые значения функций MPI приведены в стандарте. Вот основные из них:

- ✓ MPI_SUCCESS – успешно выполнена;
- ✓ MPI_ERR_BUFFER – неправильный указатель на буфер;
- ✓ MPI_ERR_COMM – неправильный коммуникатор;
- ✓ MPI_ERR_RANK – неправильный ранг процесса.

6.2.4 Парные коммуникации

Все функции передачи сообщений MPI можно разделить на две группы: парные и коллективные. Парные функции используются для передачи сообщений между двумя процессами, а коллективные выполняются одновременно всеми процессами заданного коммуникатора.

В этой лабораторной работе рассмотрим парные коммуникации.

Для отправки сообщения используется функция MPI_Send:

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

где

- ✓ `buf` – адрес буфера памяти, в котором располагаются отправляемые данные;
- ✓ `count` – количество элементов данных в сообщении;
- ✓ `type` – тип элементов данных в сообщении;
- ✓ `dest` – ранг процесса, которому отправляется сообщение;
- ✓ `tag` – значение, используемое для идентификации сообщений;
- ✓ `comm` – коммуникатор, в рамках которого выполняется передача данных.

В MPI используются специальные типы данных, такие как

- ✓ `MPI_CHAR (char)`
- ✓ `MPI_INT (int)`
- ✓ `MPI_FLOAT (float)`
- ✓ `MPI_DOUBLE (double)`
- ✓ `MPI_CXX_BOOL (bool)`
- ✓ `MPI_CXX_FLOAT_COMPLEX (std::complex<float>)`

Все доступные типы данных можно посмотреть в нужном стандарте.

Получить отправленное сообщение можно при помощи функции MPI_Recv:

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

где

- ✓ `buf` – адрес буфера памяти, в который принимаются данные;
- ✓ `count` – размер буфера;

- ✓ `type` – тип принимаемых элементов;
- ✓ `source` – ранг процесса, от которого принимается сообщение;
- ✓ `tag` – значение, используемое для идентификации сообщений;
- ✓ `comm` – коммуникатор, в рамках которого выполняется передача данных;
- ✓ `status` – указатель на структуру данных с информацией о результате выполнения операции приема данных.

В случае, если процесс должен получить сообщения от нескольких других, можно указать вместо ранга процесса-отправителя константу `MPI_ANY_SOURCE`. Константа `MPI_ANY_TAG` может заменить тег принимаемого сообщения в случае, если процессу получателю неважно, с каким тегом отправлено сообщение.

Узнать неизвестные параметры можно через переменную status

- ✓ `status.MPI_SOURCE` – ранг процесса-отправителя;
- ✓ `status.MPI_TAG` – тег принятого сообщения.

Количество переданных даных можно узнать при помощи функции MPI_Get_count:

```
int MPI_Get_count(MPI_Status * status,
                  MPI_Datatype type, int * count)
```

```
int proc_rank, proc_num;
    const int buf_size = 20;
    char buf[buf_size];

    MPI_Status st;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &proc_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &proc_num);
    if (proc_rank == 0) {
      for (int i = 1; i < proc_num; i++) {
        MPI_Recv(buf, buf_size, MPI_CHAR, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &st);
        printf("%s\n", buf);
```

```
      }
    }
    else {
      sprintf(buf, "Hello from %d", proc_rank);
      MPI_Send(buf,    buf_size,    MPI_CHAR,    0,    0,
MPI_COMM_WORLD);
    }
    MPI_Finalize();
```

6.2.5 Одновременное выполнение передачи и приема сообщений

Функции MPI_Send и MPI_Recv являются блокирующими, то есть вызов не возвращает программе управление до тех пор, пока данные не будут скопированы в указанное место (полностью переданы)

```
if(rank == 0) {
   MPI_Send(… 1 …);
   MPI_Recv(… 1 …);
}
else {
   MPI_Send(… 0 …);
   MPI_Recv(… 0 …);
}
```

Для того, чтобы избежать взаимоблокировки процессов при множественных пересылках сообщений между процессами, можно использовать функции для совмещения передачи и приема сообщений:

```
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);
int MPI_Sendrecv_replace(
                 void  *buf,  int  count,  MPI_Datatype  type,
```

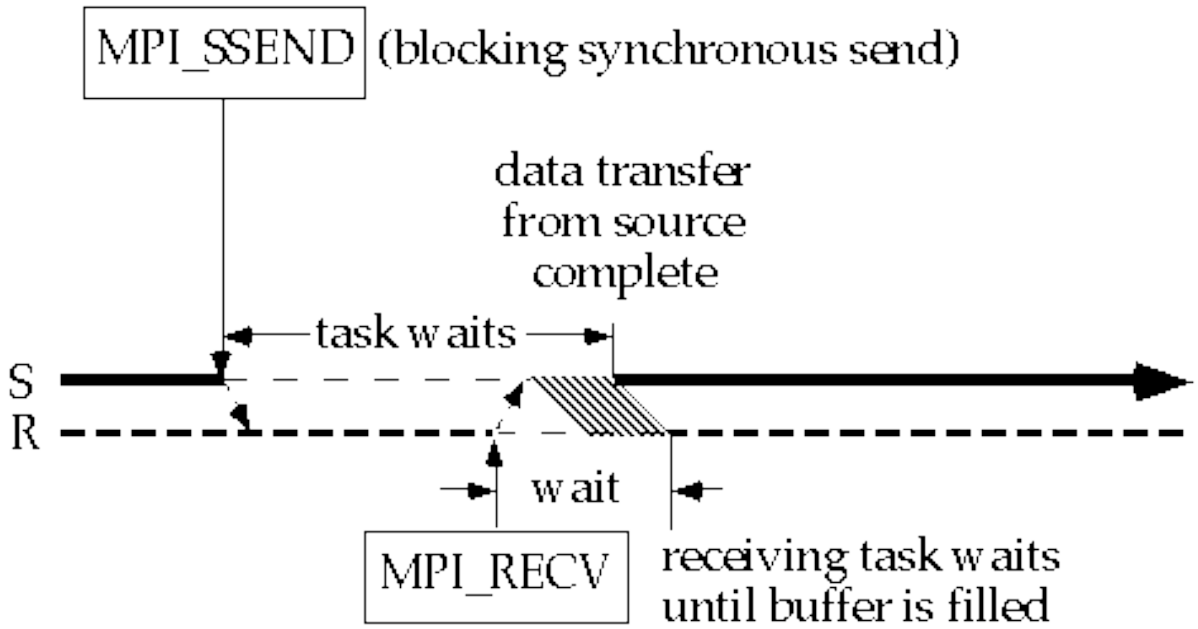
```
                    int dest, int stag,
                    int source, int rtag,
                    MPI_Comm comm, MPI_Status *status);
```

6.2.6 Способы коммуникации

Способ коммуникации – метод, по которому система обрабатывает сообщения. Способ коммуникации определяется при отправке сообщения. Существуют 4 способа коммуникации:

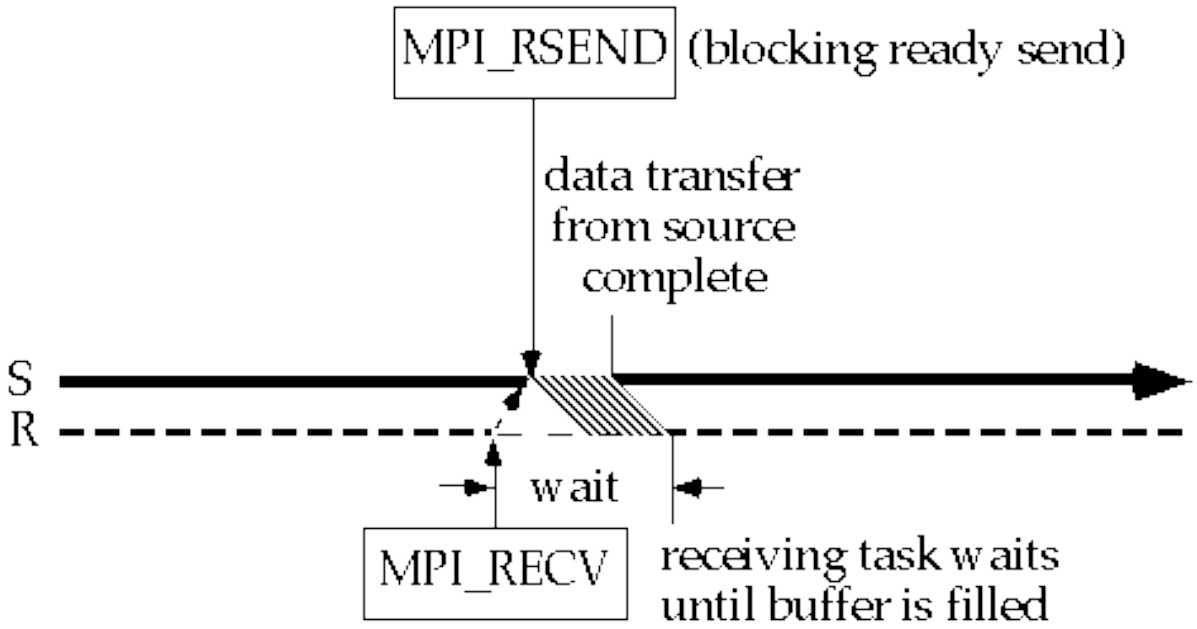
- ✓ *стандартный* (*standard*) – `MPI_Send`
- ✓ *синхронный* (*synchronous*) – `MPI_Ssend`
- ✓ *буферизованный* (*buffered*) – `MPI_Bsend`
- ✓ *по готовности* (*ready*) – `MPI_Rsend`

Функция получения не определяет способ коммуникации (всегда `MPI_Recv`).

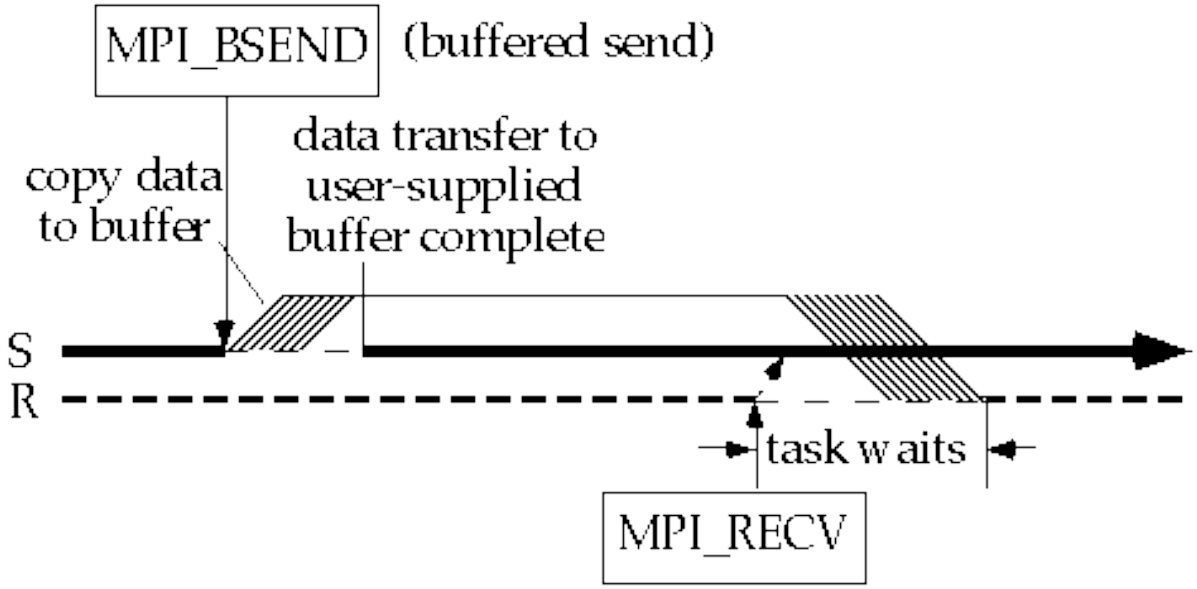


Существуют два источника задержки при передаче сообщения

- ✓ ***системная накладка*** – вызывается копированием данных сообщения из буфера сообщения отправителя и копированием данных из сети в буфер сообщения получателя;
- ✓ ***синхронизационная накладка*** – время, потраченное на ожидание другой задачи.



Если операция приёма не начала выполнение, то поведение не определено (выдаст ошибку).



На стороне отправителя выделяется дополнительный буфер, в котором отправляемые данные хранятся до момента их получения

Для использования буферизованного режима передачи должен быть создан и передан MPI буфер памяти для буферизации сообщений

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

где

- ✓ `buffer` – буфер памяти для буферизации сообщений;
- ✓ `size` – размер буфера в байтах.

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

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

Пример.

```
int BUFSIZE = sizeof(int) + MPI_BSEND_OVERHEAD;
char *buf = new char[BUFSIZE] {0};
int buf_size; //размер буфера
int proc_rank; //ранг процесса
int num; //число
MPI_Status st;
MPI_Init(&argc, &argv); //начало области MPI
MPI_Comm_rank(MPI_COMM_WORLD, &proc_rank); //ранг
if (proc_rank == 0) { //0 процесс
    cin >> num;
    MPI_Buffer_attach(buf, BUFSIZE); //присоединяем буфер
    MPI_Bsend(&num, 1, MPI_INT, 1, 100, MPI_COMM_WORLD);
    MPI_Buffer_detach(buf, &buf_size); //отсоединяем
}
if (proc_rank == 1) { //1 процесс
    MPI_Recv(&num, 1, MPI_INT, 0, 100, MPI_COMM_WORLD, &st);
    cout << "Process 1 received number " << num << " from
process " << st.MPI_SOURCE << endl;
}
MPI_Finalize();
```

Решение о том, будет ли исходящее сообщение буферизовано, принимает MPI. Посылка может зависеть от условий приёма и размера передаваемого сообщения

6.2.7 Неблокирующие обмены

У всех парных операций пересылки есть неблокирующие аналоги, имеющие префикс I (Immediate). Неблокирующие (асинхронные) операции лишь инициируют процесс передачи или приёма сообщения, управление возвращается

сразу. При выполнении данных операций отсутствует ожидание завершения копирования данных в промежуточный буфер или из него. Это позволяет экономить время на передачу данных, но возлагает ответственность на программиста за корректную работу с буфером.

Коммуникационные операции разделяются на две стадии

- ✓ инициирование операции;
- ✓ проверка завершения операции.

Использование неблокирующих операций повышает безопасность программы с точки зрения возникновения тупиковых ситуаций, а также может увеличить скорость работы программы за счёт совмещения выполнения вычислительных и коммуникационных операций.

Замечание. Многое зависит от реализации, не всегда асинхронные операции поддерживаются аппаратурой и системным окружением.

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

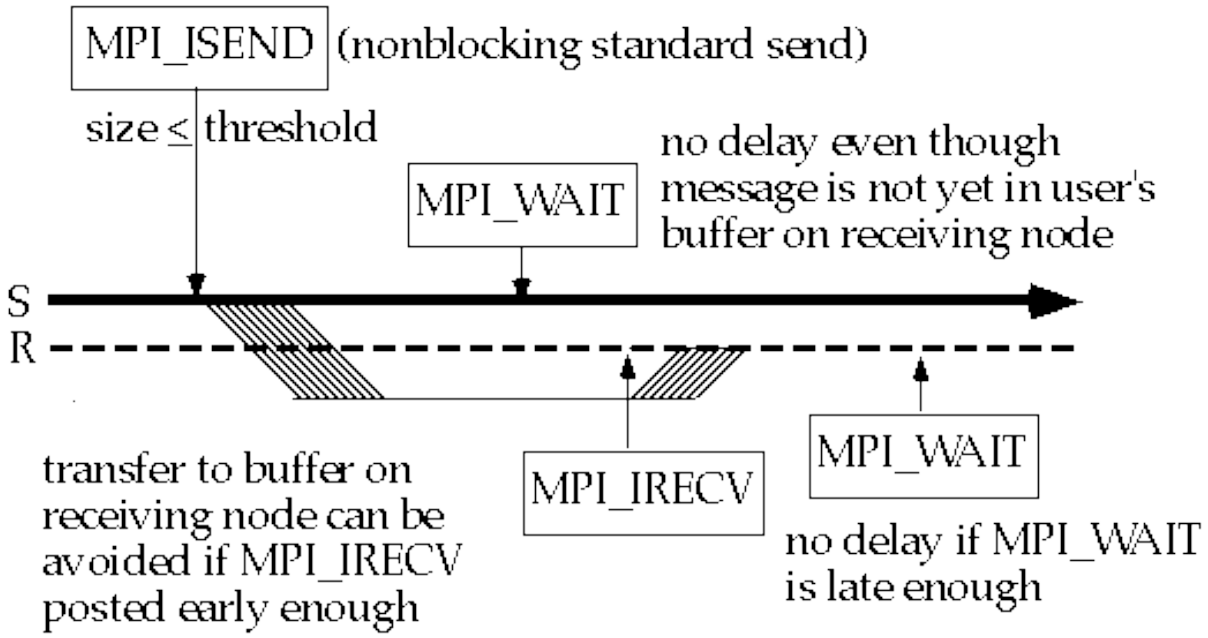
Дополнительный параметр `request` типа `MPI_Request` используется для идентификации конкретной неблокирующей операции. Возврат из функции происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере.

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

Возврат из процедуры происходит сразу после инициализации процесса приёма без ожидания получения всего сообщения и его записи в буфер.

Операция неблокирующей посылки имеет три дополнительных варианта

- ✓ `MPI_Issend`;
- ✓ `MPI_Ibsend`;
- ✓ `MPI_Irsend`.



Определить момент времени, когда можно повторно использовать буфер, можно при помощи функций MPI_Wait и MPI_Test.

`int MPI_Wait(MPI_Request *request, MPI_Status *status)` – блокирует процесс до тех пор, пока асинхронная операция с параметром `request` не будет завершена

- ✓ `request` – дескриптор операции;
- ✓ `status` – результат выполнения операции обмена.

После выполнения идентификатор неблокирующей операции request устанавливается в значение `MPI_REQUEST_NULL`.

Другие варианты операции ожидания:

- ✓ `int MPI_Waitany(int count, MPI_Request requests[], int *index, MPI_Status *status)`
- ✓ `int MPI_Waitall(int count, MPI_Request requests[], MPI_Status statuses[])`
- ✓ `int MPI_Waitsome(int incount, MPI_Request requests[], int *outcount, int indices[], MPI_Status statuses[])`

`int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)` – проверяет, завершена ли асинхронная операция, ассоциированная с индентификатором request. Не является блокирующей. Инициализирует переменную `flag` (true/false). Если flag = true, то операция завершена, иначе –

продолжает выполняться. Если операция завершена, то будет проинициализирована переменная status. После выполнения идентификатор неблокирующей операции request устанавливается в значение MPI_REQUEST_NULL.

Другие варианты операции:

- ✓ `MPI_Testany;`
- ✓ `MPI_Testall;`
- ✓ `MPI_Testsome.`

6.2.8 Примеры

Задача 1. Написать MPI программу, в которой процессы с положительными рангами должны отправлять сообщения "Hello from process k" (где *k* – ранг отправившего процесса) процессу с рангом 0, а процесс с рангом 0 должен получать и выводить полученные сообщения.

Решение.

```
#include <iostream>
#include <math.h>
#include <mpi.h>
int main(int argc, char * argv[]) {
   int proc_rank, proc_num;
   const int buf_size = 20;
   char buf[buf_size];
   MPI_Status st;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &proc_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &proc_num);
   if (proc_rank == 0)    {
      for (int i = 1; i < world_size; i++) {
         MPI_Recv(buf, buf_size, MPI_CHAR, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &st);
         printf("%s\n", buf);
      }
```

```
    }
    else {
       sprintf(buf, "Hello from %d", proc_rank);
       MPI_Send(buf, buf_size, MPI_CHAR,0,0,MPI_COMM_WORLD);
    }
    MPI_Finalize();
 }
```

Задача 2. Вычислить сумму первых N натуральных чисел ($N \leq 2^{31} - 1$).

Решение.

```
 int main(int argc, char * argv[])
 {
    int proc_rank, proc_num;
    long long N, sum = 0, tmpsum = 0;
    MPI_Status st;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &proc_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &proc_num);
    if (proc_rank == 0)
    {
       std::cout << "input N: ";
       std::cin >> N;
       for (int i = 1; i < proc_num; ++i)
          MPI_Send(&N,    1,    MPI_LONG_LONG,    i,    0,
 MPI_COMM_WORLD);
       for (int i = 1; i < proc_num; ++i) {
          MPI_Recv(&tmpsum,        1,        MPI_LONG_LONG,
 MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &st);
          sum = sum + tmpsum;
       }
       std::cout << "Result: " << sum;
    }
```

```
   else
   {
      MPI_Recv(&N,  1,  MPI_LONG_LONG,  0,  MPI_ANY_TAG,
MPI_COMM_WORLD, &st);
      int num = ceil((double)N / (proc_num - 1));
      int start = num * (proc_rank - 1) + 1;
      int end = num * proc_rank + 1;
      if (proc_rank == proc_num - 1)
         end = N + 1;
      for (int i = start; i < end; ++i)
         sum = sum + i;
      MPI_Send(&sum,    1,    MPI_LONG_LONG,    0,    0,
MPI_COMM_WORLD);
   }
   MPI_Finalize();
}
```

6.3. Задание на лабораторную работу

Создайте новый проект, настройте поддержку MPI.

1. Напишите программу, в которой каждый процесс выводит на экран свой номер и общее количество процессов в формате

   `I am < Номер процесса > process from < Количество процессов > processes!`

2. Напишите программу, в которой каждый процесс с чётным номером выводит на экран строку

   `<Номер процесса>: FIRST!`

   а каждый процесс с нечётным номером строку

   `<Номер процесса>: SECOND!`

   Нулевой процесс должен вывести на экран информацию о количестве работающих процессов в формате

   `<Количество процессов> processes.`

3. Скомпилируйте и запустите на выполнение приведённый ниже код. Поясните, почему возникла тупиковая ситуация? Исправьте программу, заменив блокирующие вызовы на неблокирующие.

```
#include <iostream>
#include <mpi.h>
#include <vector>

constexpr auto MSGLEN = 32768; //размер сообщения
constexpr auto TAG_A = 100;
constexpr auto TAG_B = 200;

using namespace std;

int main(int argc, char * argv[])
{
    vector<float> message1(MSGLEN), message2(MSGLEN); //пересылаемые
сообщения
    int rank, dest, source, send_tag, recv_tag;
    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    for (int i = 0; i < MSGLEN; i++) {
         message1[i] = 1 - 2 * rank;
    }
```

```
    if (rank == 0) {
        dest = 1;
        source = 1;
        send_tag = TAG_A;
        recv_tag = TAG_B;
    }
    else if (rank == 1) {
        dest = 0;
        source = 0;
        send_tag = TAG_B;
        recv_tag = TAG_A;
    }

    cout << "Task " << rank << " has sent the message" << endl;
    MPI_Send(message1.data(), MSGLEN, MPI_FLOAT, dest, send_tag,
MPI_COMM_WORLD);
    MPI_Recv(message2.data(), MSGLEN, MPI_FLOAT, source, recv_tag,
MPI_COMM_WORLD, &status);
    cout << " Task " << rank << " has received the message" << endl;

    MPI_Finalize();
    return 0;
}
```

4. ***Эстафетная палочка***. Организовать передачу данных по кругу. Нулевой процесс генерирует случайное целое число и передаёт его первому процессу. Далее каждый процесс получает некоторое целое число, прибавляет к нему 1 и передаёт следующему процессу. Последний процесс передаёт число нулевому. Нулевой процесс выводит «Correct!», если новое число на $p - 1$ больше исходного или «Error!» в противном случае. Остальные процессы выводят сообщение «<Номер процесса> receive number <число>».

5. ***Концепция master-slave***. Нулевой процесс генерирует N массивов целых чисел из M элементов. Далее он распределяет по остальным процессам по одному массиву. Каждый процесс, получив массив, считает сумму его элементов и отсылает обратно нулевому. Нулевой процесс добавляет полученный результат к глобальной сумме и отправляет освободившемуся процессу новую работу. Так происходит до тех пор, пока не вычислена сумма всех элементов всех массивов (сумма элементов матрицы).

   Реализовать алгоритм двумя способами: с использованием блокирующих и неблокирующих операций. Протестировать для следующих пар N и M: 100 и 1000000, 10000 и 10000, 1000000 и 100. Сравнить быстродействие. Сделать выводы.

6. ***Каждый каждому***. Целочисленная матрица размера $p \times p$ хранится следующим образом: на нулевом процессе – нулевая строка, на первом – первая и т.д. Организовать попарные обмены таким образом, чтобы транспонировать матрицу (на нулевом процессе будет лежать нулевой столбец исходной матрицы, на первом – первый столбец и т.д.)

   Пример. Пусть одновременно запущены 3 процесса, при этом на нулевом процессе лежат числа 1, 2, 3, на первом – 4, 5, 6, а на втором – 7, 8, 9. После обменов на нулевом процессе должны оказаться числа 1, 4, 7, на первом – 2, 5, 8, а на втором – 3, 6 и 9.

6.4. Результаты лабораторной работы

Результаты лабораторной работы представляются в виде отчета по лабораторной работе. В отчет включается титульный лист, цель работы, задание на лабораторную работу, листинг с комментариями и выводы по лабораторной работе.

Отчет оформляется в электронном виде и высылается на e-mail vbyzov.vyatsu@gmail.com

Лабораторная работа считается зачтенной после её защиты.