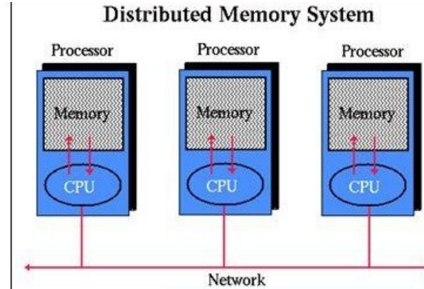
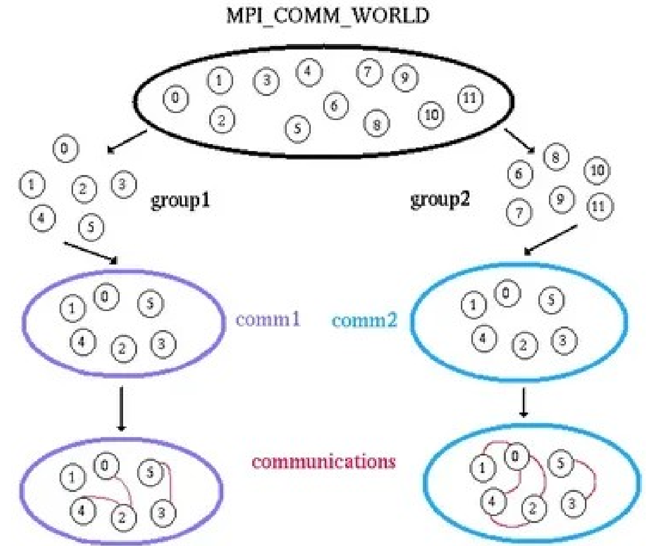
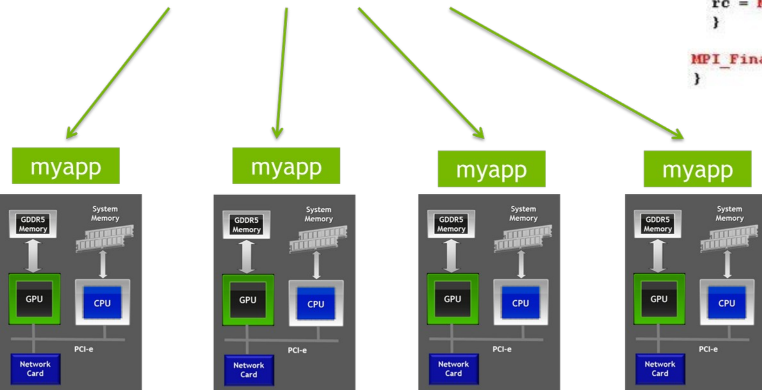
Архитектуры параллельных ВС
Message Passing Interface (MPI)

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

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, dest, source, rc, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

MPI_Finalize();
}
```






```
mpirun -np 4 ./myapp <args>
```



Определения

Message Passing Interface (MPI, интерфейс передачи сообщений) — программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу.

Является наиболее распространённым стандартом интерфейса обмена данными в параллельном программировании.

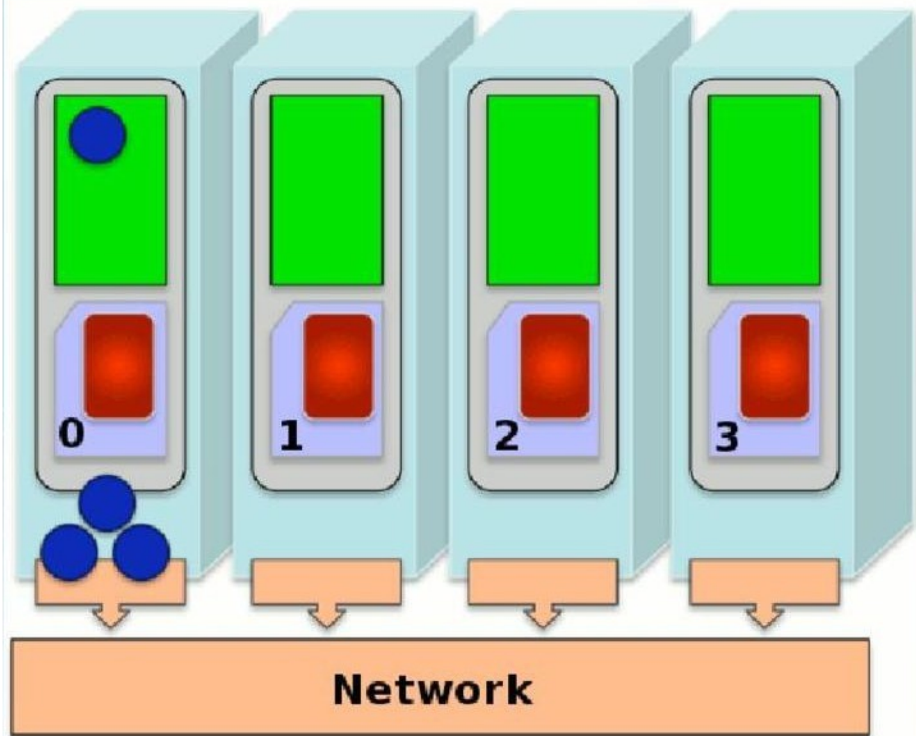
Существуют реализации для большого числа компьютерных платформ.

Используется при разработке программ для кластеров и суперкомпьютеров. Основным средством коммуникации между процессами в MPI является передача сообщений друг другу.

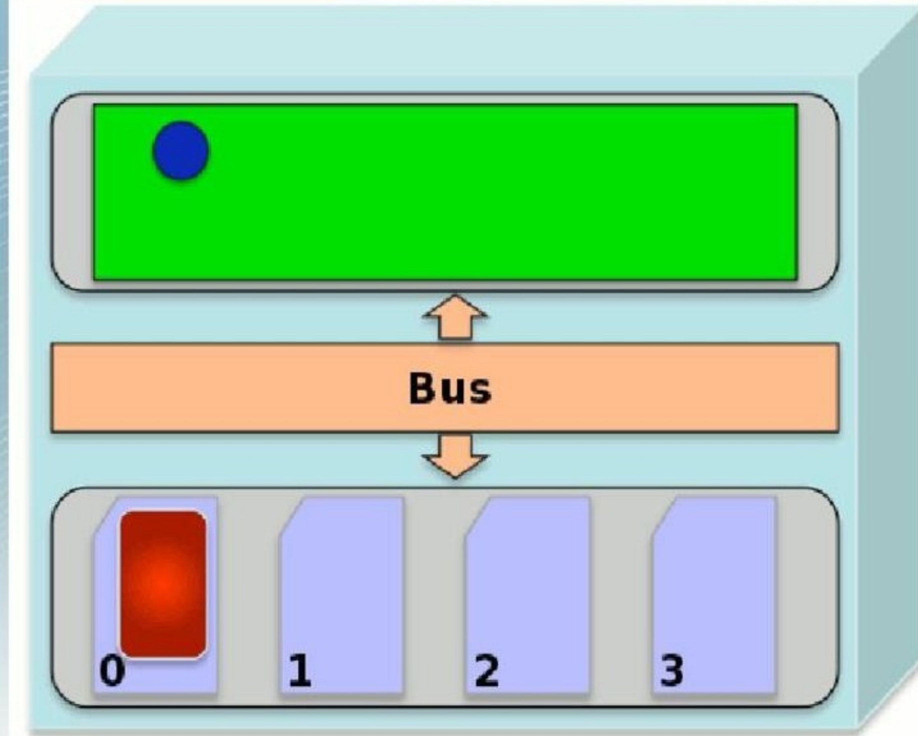
Определения

В первую очередь MPI ориентирован на системы с распределенной памятью, то есть когда затраты на передачу данных велики, в то время как OpenMP ориентирован на системы с общей памятью (многоядерные с общим кэшем). Обе технологии могут использоваться совместно, чтобы оптимально использовать в кластере многоядерные системы.

MPI vs OpenMP Programing



Message-Passing Parallelism



Shared-Memory Parallelism

Стандарты MPI

Первая версия MPI 1 вышла в 1994.

Опубликован 12 июня 1995 года, первая реализация появилась в 2002 году)

Большинство современных реализаций поддерживают версию 1.1.

Поддерживаются следующие функции:

- передача и получение сообщений между отдельными процессами;
- коллективные взаимодействия процессов;
- взаимодействия в группах процессов;
- реализация топологий процессов.

Стандарты MPI

MPI 2.0 опубликован 18 июля 1997 года

Дополнительно поддерживаются следующие функции:

- динамическое порождение процессов и управление процессами;
- односторонние коммуникации (Get/Put);
- параллельный ввод и вывод;
- расширенные коллективные операции (процессы могут выполнять коллективные операции не только внутри одного коммуникатора, но и в рамках нескольких коммуникаторов).

Версия MPI 2.1 вышла в начале сентября 2008 года.

Версия MPI 2.2 вышла 4 сентября 2009 года.

Версия MPI 3.0 вышла 21 сентября 2012 года.

Функционирование MPI

Базовый механизм связи между процессами MPI - передача и приём сообщений.

Сообщение содержит передаваемые данные и информацию, позволяющую принимающей стороне осуществлять их выборочный приём:

- отправитель — ранг (номер в группе) отправителя сообщения;
- получатель — ранг получателя;
- признак — может использоваться для разделения различных видов сообщений;
- коммуникатор — код группы процессов.

Функционирование MPI

Операции приёма и передачи могут быть блокирующимися и неблокирующимися.

Для неблокирующихся операций определены функции проверки готовности и ожидания выполнения операции.

Другим способом связи является удалённый доступ к памяти (RMA). Позволяет читать и изменять область памяти удалённого процесса. Локальный процесс может переносить область памяти удалённого процесса (внутри указанного процессами окна) в свою память и обратно, а также комбинировать данные, передаваемые в удалённый процесс с имеющимися в его памяти данными (например, путём суммирования). Все операции удалённого доступа к памяти неблокирующиеся, однако, до и после их выполнения необходимо вызывать блокирующиеся функции синхронизации.

Основные функции MPI (общие функции)

```
int MPI_Init( int* argc, char** argv)
```

Инициализация параллельной части приложения. Реальная инициализация для каждого приложения выполняется не более одного раза. Если MPI уже был инициализирован, то происходит немедленный возврат из функции.

Все остальные MPI-функции могут быть вызваны только после вызова MPI_Init.

Возвращает: в случае успешного выполнения — **MPI_SUCCESS (=0)**, иначе - код ошибки (как и прочие функции).

Типы аргументов MPI_Init предусмотрены для того, чтобы передавать всем процессам аргументы main.

Основные функции MPI (общие функции)

int MPI_Finalize( )

Завершение параллельной части приложения. Последующие обращения к MPI-процедурам, включая MPI_Init, запрещены.

К моменту вызова MPI_Finalize некоторым процессом все действия, требующие его участия в обмене сообщениями, должны быть завершены.

```
int main(int argc, char** argv)
{
        MPI_Init(&argc, &argv);
            ...
        MPI_Finalize();
}
```

Основные функции MPI (общие функции)

```
int MPI_Comm_size( MPI_Comm comm, int* size)
```

Определение общего числа параллельных процессов в группе comm.

comm - идентификатор группы

OUT: size - размер группы (возвращается)

Основные функции MPI (общие функции)

```
int MPI_Comm_rank( MPI_Comm comm, int* rank)
```

Определение номера процесса в группе comm.

Значение, возвращаемое по адресу &rank, лежит в диапазоне от 0 до size_of_group-1.

- comm - идентификатор группы
- OUT: rank - номер вызывающего процесса в группе comm

```
double MPI_Wtime(void)
```

Возвращает астрономическое время в секундах, прошедшее с некоторого момента в прошлом. Гарантируется, что этот момент не будет изменен за время существования процесса.

Hello, MPI

```
#include <iostream>
#include <mpi.h>
int main (int argc, char* argv[]) {
    int rank, commSize;
    int errCode;
    // Инициализация и определение основных параметров
    if ((errCode = MPI_Init(&argc, &argv)) != MPI_SUCCESS) { // != 0
        return errCode;
    }
    auto time = MPI_Wtime();
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &commSize);
    if (rank == 0) {
        std::cout << "commSize = " << commSize << "\n";
        std::cout << time << ": Hello from main MPI! I'm process " << rank << "\n";
    } else {
        std::cout << time << ": Hello from other MPI! I'm process " << rank << "\n";
    }

    MPI_Finalize();
    return 0;
}
```

Hello, MPI (openmpi.host)

```
mpic++  main.cpp
mpirun -hostfile openmpi.host -np 16 ./a.out

192.168.1.32
Localhost
192.168.1.41 slots=4
127.0.0.1
192.168.1.45
...

localhost slots=20
127.0.0.1
127.0.0.1 slots=5
```

Основные функции MPI (прием/передача сообщений между процессами)

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

Прием/передача сообщений с блокировкой

- buf - адрес начала буфера посылки сообщения
- count - число передаваемых элементов в сообщении
- datatype - тип передаваемых элементов
- dest - номер процесса-получателя
- msgtag - идентификатор сообщения
- comm - идентификатор группы

Основные функции MPI (прием/передача сообщений между процессами)

Блокирующая посылка сообщения msgtag, из count элементов типа datatype, процессу с номером dest. Все элементы сообщения расположены подряд в buf. Значение count м.б. = 0. Тип элементов datatype указывается с помощью **предопределенных констант**. Можно передавать сообщение себе.

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. Выбор способа осуществления гарантии: копирование в промежуточный буфер или непосредственная передача процессу dest, остается за MPI. Возврат из MPI_Send не означает, что сообщение уже передано процессу dest, или покинуло процессор, на котором выполняется процесс, выполнивший MPI_Send.

Основные функции MPI
(прием/передача сообщений между процессами)

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

OUT buf - адрес начала буфера приема сообщения

count - число элементов в принимаемом сообщении

datatype - тип элементов принимаемого сообщения

source - номер процесса-отправителя

msgtag - идентификатор принимаемого сообщения

comm - идентификатор группы

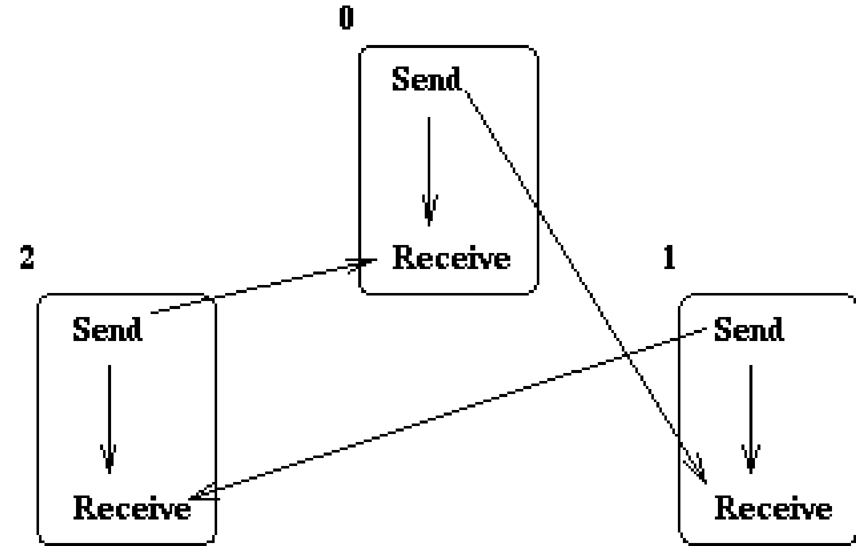
OUT status - параметры принятого сообщения

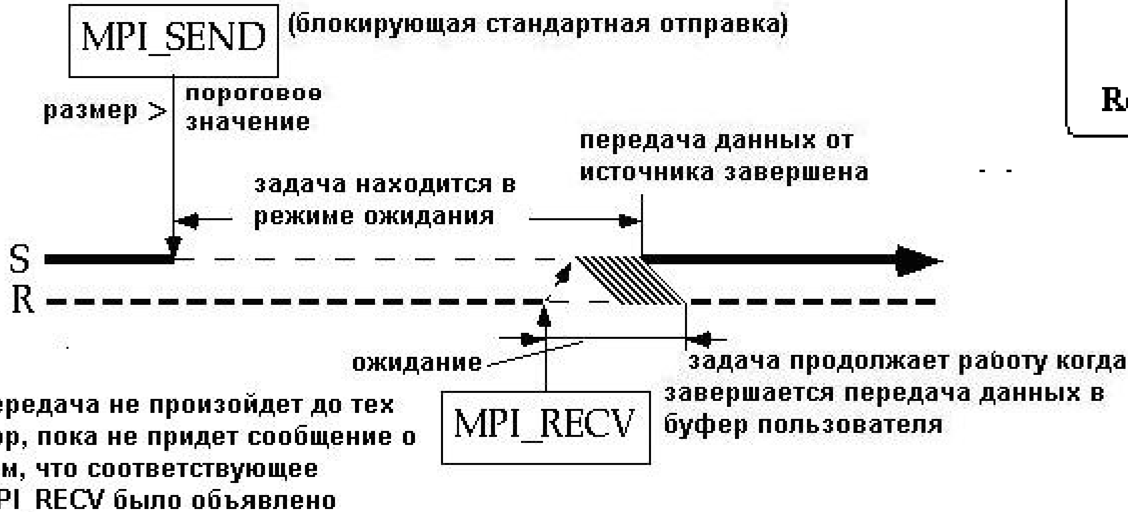
Основные функции MPI (прием/передача сообщений между процессами)

Прием сообщения msgtag от процесса source с блокировкой. Число элементов в принимаемом сообщении не должно превосходить count. Если число принятых элементов меньше count, то гарантируется, что в буфере buf изменятся только элементы, соответствующие элементам принятого сообщения. Если нужно узнать точное число элементов, то можно воспользоваться MPI_Probe. Гарантирует, что после возврата из подпрограммы все элементы сообщения приняты и расположены в буфере buf.

Если процесс посылает два сообщения другому процессу и оба эти сообщения соответствуют одному и тому же вызову MPI_Recv, то первым будет принято то сообщение, которое было отправлено раньше.

Основные функции MPI (прием/передача сообщений между процессами)





Основные функции MPI (прием/передача сообщений между процессами)

```
int MPI_Probe( int source, int msgtag,
                    MPI_Comm comm, MPI_Status *status)
```

source - номер отправителя или MPI_ANY_SOURCE

msgtag – ид.ожидаемого сообщения или MPI_ANY_TAG

comm - идентификатор группы

OUT status - параметры обнаруженного сообщения

Получение информации о структуре ожидаемого сообщения с блокировкой. Возврата из подпрограммы не произойдет до тех пор, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для получения. Атрибуты доступного сообщения можно определить обычным образом с помощью status. *Функция определяет только факт прихода сообщения, но реально его не принимает.*

Основные функции MPI (прием/передача сообщений между процессами)

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

status - параметры принятого сообщения

datatype - тип элементов принятого сообщения

OUT count - число элементов сообщения

По значению параметра status функция определяет число уже принятых (после обращения к MPI_Recv) или принимаемых (после обращения к MPI_Probe или MPI_Iprobe) элементов сообщения типа datatype.

Send-Recive

```
#include <mpi.h>
#include <cstdio>
#include <unistd.h>
const int len = 50;

int main(int argc, char **argv ) {
    int i,rank,size;
    char buffer[len];
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    sprintf(buffer, "Hello from %d", rank); // формирование сообщения
    auto t1 = MPI_Wtime();  // фиксация времени «начала посылки»,
    MPI_Send(buffer, len, MPI_CHAR, size-(rank+1), rank, MPI_COMM_WORLD); // (1)
    MPI_Recv(buffer, len, MPI_CHAR, size-(rank+1), size-(rank+1),
                                                MPI_COMM_WORLD, &status); // (2)
    auto t2 = MPI_Wtime(); // фиксация времени «окончания приема»,
    printf("Process %d ---> Buffer = %s\n", rank, buffer);   // вывод сообщения
    printf("From process %d. Time = %le\n", rank, (t2-t1));  // вывод времени
    MPI_Finalize();
    return 0;
}
```

Использование барьеров

```
#include <mpi.h>
#include <stdio.h>
int main( int argc, char **argv ) {
    int size, rank, i;
    MPI_Init( &argc, &argv ); // инициализация MPI-библиотеки
    MPI_Comm_size( MPI_COMM_WORLD, &size ); // определение количества ветвей
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );// определение своего ранга
    // ветвь с нулевым рангом сообщает количество запущенных процессов
    if( rank == 0 ) { printf("Total processes count = %d\n", size ); }
    // все ветви сообщают свой ранг
    printf("Hello! My rank in MPI_COMM_WORLD = %d\n", rank );
    // Точка синхронизации
    MPI_Barrier( MPI_COMM_WORLD );
    // ветвь с рангом 0 сообщает аргументы командной строки,
    if( rank == 0 ) {
        for( puts("Command line of process 0:"), i=0; i < argc; i++ ) {
            printf( "%d: \"%s\"\n", i, argv[i] );
        }
    }
    MPI_Finalize(); // закрытие MPI-библиотеки
    return 0;
}
```

Основные функции MPI (работа с группами процессов)

```
int MPI_Comm_split( MPI_Comm comm,
                    int color, int key, MPI_Comm *newcomm)
```

comm - идентификатор группы

color - признак разделения на группы

key - параметр, определяющий нумерацию в новых группах

OUT newcomm - идентификатор новой группы

Функция разбивает множество процессов, входящих в группу comm, на непересекающиеся подгруппы - одну подгруппу на каждое значение параметра color (неотрицательное число). Каждая новая подгруппа содержит все процессы одного цвета. Если в качестве color указано значение MPI_UNDEFINED, то в newcomm будет возвращено значение MPI_COMM_NULL.

Основные функции MPI
(работа с группами процессов)

```
int MPI_Comm_free( MPI_Comm comm)
```

OUT comm - идентификатор группы

Уничтожает группу, ассоциированную с идентификатором comm, который после возвращения устанавливается в MPI_COMM_NULL.

Группы и широковещательные рассылки

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    int rank1, size1, rank2, size2; value;
    MPI_Comm comm1;
    MPI_Status status;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank1);
    MPI_Comm_size (MPI_COMM_WORLD, &size1);
    printf ("Мой ранг в  MPI_COMM_WORLD %d. ", rank1);
    printf ("Всего  в  MPI_COMM_WORLD %d процессов\n", size1);
    MPI_Comm_split (MPI_COMM_WORLD, (rank1%2==0) ? 0 : 1, 0, &comm1);
    MPI_Comm_rank (comm1, &rank2);
    MPI_Comm_size (comm1, &size2);
    if ((rank2==0) && (rank1%2)) value=777;
    if ((rank2==0) && !(rank1%2)) value=666;
    MPI_Bcast (&value, 1, MPI_INT, 0, comm1);
    printf ("Мой ранг в  MPI_COMM_WORLD  %d, broadcasted message %d, value = %d\n",
                                                            rank1, rank2, value);
    MPI_Finalize();
    return 0;
}
```

Сумма квадратов элементов массива

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

using namespace std;

const int Tag = 0;
const int root = 0;
double calcTime = 0.0;

// Вычисление суммы квадратов n элементов массива
double sum_array(double *array, int n) {
  double sum = 0;
  for (int i = 0; i < n; ++i) {
    sum += array[i]*array[i];
    //sum += array[i];
  }
  return sum;
}
```

Сумма квадратов элементов массива

```
// Главная функция, поддерживающая работу головного процесса
// и запускающая вспомогательные процессы.
int main() {
  int rank, commSize;

  // Инициализация и определение основных параметров
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &commSize);

  double *arr, sum = 0, buffer;
  int n;
  MPI_Status status;
```

Сумма квадратов элементов массива

```
// С числом элементов разбирается головной процесс
if (root == rank) {
  cout << "commSize = " << commSize << "\n";
  cout << "n? ";
  cin >> n;
  // Он же используется для ввода данных
  arr = new double[n];
  for (int i = 0; i < n; ++i) {
      arr[i] = i + 1;
  }
  // Определение размера каждой из обрабатываемой порций
  int partSize = n/commSize;

  auto time1 = MPI_Wtime();

  // Определение остатка для корневого процесса
  int shift = n % commSize;
  // Рассылка одинаковых порций остальным процессам
  for (int i = root+1; i < commSize; ++i) {
    MPI_Send(arr + shift + partSize*i, partSize, MPI_DOUBLE,
                                            i, Tag, MPI_COMM_WORLD);
  }
```

Сумма квадратов элементов массива

```
  // Основной процесс обрабатывает начальную порцию
  sum = sum_array(arr, shift + partSize);
  // Вывод промежуточной суммы головного процесса
  cout << "Main process is " << rank << " : " << sum << endl;

  // Получение данных от порожденных процессов
  for (int i = root+1; i < commSize; ++i) {
    // Прием данных в буфер по очереди от всех процессов
    MPI_Recv(&buffer, 1, MPI_DOUBLE, i, Tag, MPI_COMM_WORLD, &status);
    // суммирование
    sum += buffer;
  }
  auto time2 = MPI_Wtime();
  calcTime = time2 - time1;
}
```

Сумма квадратов элементов массива

```
else {
  // Работа порожденных процессов
  MPI_Probe(root, Tag, MPI_COMM_WORLD, &status);
  MPI_Get_count(&status, MPI_DOUBLE, &n);

  arr = new double[n];
  MPI_Recv(arr, n, MPI_DOUBLE, root, Tag, MPI_COMM_WORLD, &status);

  sum = sum_array(arr, n);

  MPI_Send(&sum, 1, MPI_DOUBLE, root, Tag, MPI_COMM_WORLD);

}
```

Сумма квадратов элементов массива

```
    // Вывод промежуточных сумм вспомогательных процессов
    cout << "Process " << rank << " : " << sum << endl;

    MPI_Barrier(MPI_COMM_WORLD);     // Точка синхронизации

  if (root == rank) {
      cout << "Resulting Summa = " << sum << "\n";
      cout << "Computational time = " << calcTime << "\n";
  }

  delete[] arr;
  MPI_Finalize();
  return 0;
}
```

Используемые источники

1. Википедия. Message Passing Interface - https://ru.wikipedia.org/wiki/Message_Passing_Interface

2. MPI. Вводный курс - http://ssd.sscc.ru/old/old/kraeva/MPI.html

3. Воеводин Вл.В. Лекция 5. Технологии параллельного программирования. Message Passing Interface (MPI) - http://masters.donntu.org/2013/fknt/kyrylov/library/MPI.htm

3. Блог программиста. Основы технологии MPI на примерах - https://pro-prof.com/archives/4386