

*А.Ю. Сыщиков, А.И. Улудинцева, Ю.Е. Шейнин*

Параллельное программирование.

# **MPI: Message Passing Interface.**

Методические указания к выполнению лабораторных работ.

# Оглавление

Введение.....	3
1    Функции MPI.....	4
1.1    Области связи и коммутаторы .....	4
1.2    Основные типы функций MPI .....	4
1.3    Общие процедуры MPI.....	5
1.4    Прием/передача сообщений между отдельными процессами (связь "точка-точка"). .....	6
1.5    Групповые (коллективные) взаимодействия .....	10
1.6    Функции поддержки распределенных операций.....	15
1.7    Синхронизация процессов .....	17
2    Пример задачи, использующий функции MPI.....	18
3    Установка, компиляция и запуск программ MPI.....	20
3.1    Установка Microsoft MPI.....	20
3.2    Компиляция в Microsoft Visual Studio (на примере VS 2013).....	20
3.3    Запуск скомпилированной программы в командной строке.....	24
4    Задания к лабораторным работам. ....	25
4.1    Замечания к выполнению лабораторных работ.....	25
4.2    Лабораторная работа №1. Пересылка данных. ....	26
4.3    Лабораторная работа №2. Распределенные вычисления. ....	27
4.4    Лабораторная работа №3. Сортировка элементов массива. ....	28
Библиографический список.....	29
Русскоязычные источники: .....	29
Англоязычные источники: .....	29

## Введение

MPI - Message passing interface, интерфейс взаимодействия с помощью передачи сообщений.

MPI - стандарт на программный инструментарий для обеспечения связи между ветвями и процессами параллельного приложения. MPI предоставляет программисту единый механизм взаимодействия ветвей внутри параллельного приложения

В то же время, реализованный в MPI механизм взаимодействия процессов параллельного приложения, ориентированный на идеологию систем с обменом сообщениями, имеет реализации для различных архитектур ВС (однопроцессорные / многопроцессорные, с общей/раздельной памятью). и API операционных систем (например, Linux, Windows, и др.). MPI позволяет программировать системы параллельных процессов при разном взаимном расположении ветвей (на одном процессоре / на разных), создавать приложения, которые могут работать как в подлинно параллельном режиме на мультипроцессорной ВС (например, на Кластерной ВС), так и в мультипрограммном режиме выполнения на однопроцессорной ЭВМ.

Для MPI принято писать программу, содержащую код всех ветвей сразу. MPI-загрузчиком запускается указываемое количество экземпляров программы. Каждый экземпляр определяет свой порядковый номер в запущенном коллективе, и в зависимости от этого номера и размера коллектива выполняет ту или иную ветку алгоритма. Такая модель параллелизма называется Single program/Multiple data (SPMD), и является частным случаем модели Multiple instruction/Multiple data (MIMD). Каждая ветвь имеет пространство данных, полностью изолированное от других ветвей. Обмениваются данными ветви только в виде сообщений MPI.

Все ветви запускаются загрузчиком одновременно. Количество ветвей фиксировано - в ходе работы порождение новых ветвей невозможно. Если MPI-приложение запускается в сети, запускаемый файл приложения должен находиться на каждой машине.

# 1 Функции MPI

## 1.1 Области связи и коммутаторы

В организации пакета MPI используется понятие *области связи* (communication domain). Для организации взаимодействия, процессы параллельного приложения должны быть помещены в некоторую область связи пакета MPI. Все области связи имеют независимую друг от друга нумерацию процессов, которая используется в функциях MPI при указании взаимодействующих процессов.

При запуске приложения все процессы помещаются в создаваемую для приложения общую область связи. При необходимости они могут создавать новые области связи на базе существующих.

В распоряжение программы пользователя предоставляется *коммуникатор* - описатель области связи. Многие функции MPI имеют среди входных аргументов коммуникатор, который ограничивает сферу их действия той областью связи, к которой он прикреплен. Для одной области связи может существовать несколько коммуникаторов таким образом, что приложение будет работать с ней как с несколькими разными областями.

В исходных текстах примеров для MPI часто используется идентификатор MPI\_COMM\_WORLD. Это название коммуникатора, создаваемого библиотекой автоматически. Он описывает стартовую область связи, объединяющую все процессы приложения.

## 1.2 Основные типы функций MPI

**Блокирующие:** останавливают (блокируют) выполнение процесса до тех пор, пока производимая ими операция не будет выполнена. Неблокирующие функции возвращают управление немедленно, а выполнение операции продолжается в фоновом режиме; за завершением операции надо проследить особо. Непрокирующие функции возвращают квитанции ("requests"), которые погашаются при завершении. До погашения квитанции с переменными и массивами, которые были аргументами неблокирующей функции, запрещены любые действия.

**Локальные:** не инициируют пересылку данных между ветвями. Большинство информационных функций является локальными, так как копии системных данных уже хранятся в каждой ветви. Функция передачи MPI\_Send и функция синхронизации MPI\_Barrier не являются локальными, поскольку производят пересылку. Следует заметить, что, к примеру, функция приема MPI\_Recv (парная для MPI\_Send) является локальной: она всего лишь пассивно ждет поступления данных, ничего не пытаясь сообщить другим ветвям.

**Коллективные:** должны быть вызваны всеми ветвями-абонентами того коммуникатора, который передается им в качестве аргумента. Несоблюдение для них

этого правила приводит к ошибкам на стадии выполнения программы (как правило, к зависанию).

Все функции MPI возвращают MPI\_SUCCESS в случае успешного выполнения, иначе - код ошибки.

## 1.3 Общие процедуры MPI

### 1.3.1 Инициализация

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

**MPI\_Init.** Инициализация параллельной части приложения. Реальная инициализация для каждого приложения выполняется не более одного раза, а если MPI уже был инициализирован, то никакие действия не выполняются, и происходит немедленный возврат из подпрограммы. Все оставшиеся MPI-процедуры могут быть вызваны только после вызова MPI\_Init.

MPI\_Init получает адреса аргументов, стандартно получаемых самой main от операционной системы и хранящих параметры командной строки. В конец командной строки программы MPI-загрузчик mpirun добавляет ряд информационных параметров, которые требуются MPI\_Init.

### 1.3.2 Завершение

```
int MPI_Finalize ( )
```

**MPI\_Finalize.** Завершение параллельной части приложения (нормальное закрытие библиотеки). Все последующие обращения к любым MPI-процедурам, в том числе к MPI\_Init, запрещены. К моменту вызова MPI\_Finalize некоторым процессом все действия, требующие его участия в обмене сообщениями, должны быть завершены.

Настоятельно рекомендуется не забывать вызывать эту функцию перед возвращением из программы. Для программы на языках C/C++ это:

- перед вызовом функции exit;
- перед каждым оператором return в функции main(), находящимся после MPI\_Init;
- если функции main() назначен тип void, и она не заканчивается оператором return, то MPI\_Finalize() следует поставить в конец main().

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

### 1.3.3 Аварийное завершение

**MPI\_Abort ( MPI\_comm comm, int errorcode )**

**MPI\_Abort.** Аварийное закрытие библиотеки. Вызывается, если пользовательская программа завершается по причине ошибок времени выполнения, связанных с MPI.

Вызов MPI\_Abort из любой задачи принудительно завершает работу ВСЕХ задач, подсоединенных к заданной области связи. Если указан описатель MPI\_COMM\_WORLD, будет завершено все приложение (все его задачи) целиком, что, по-видимому, и является наиболее правильным решением. Используйте код ошибки MPI\_ERR\_OTHER, если не знаете, как охарактеризовать ошибку в классификации MPI.

### 1.3.4 Определение общего числа параллельных процессов в группе

**int MPI\_Comm\_size ( MPI\_Comm comm, int\* size )**

Аргументы функции:

*comm* -- идентификатор группы.

OUT *size* -- размер группы.

### 1.3.5 Определение номера процесса в группе

**int MPI\_Comm\_rank ( MPI\_comm comm, int\* rank )**

Аргументы функции:

*comm* -- идентификатор группы.

OUT *rank* -- номер вызывающего процесса в группе comm.

### 1.3.6 Определение времени

**double MPI\_Wtime ( )**

Функция возвращает астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом. Гарантируется, что этот момент не будет изменен за время существования процесса.

## 1.4 Прием/передача сообщений между отдельными процессами (связь "точка-точка").

Это самый простой тип связи между задачами: одна ветвь вызывает функцию передачи данных, а другая - функцию приема.

### 1.4.1 Посылка сообщения

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

Аргументы функции:

*buf* -- адрес начала буфера передачи, из которого берутся данные для отсылки. Помните, что наборы данных у каждой задачи свои, поэтому, например, используя одно и то же имя массива в нескольких задачах, Вы указываете не одну и ту же область памяти, а разные, никак друг с другом не связанные.

*count* -- число передаваемых в сообщении элементов. Задается не в байтах, а в количестве ячеек. Указывает, сколько ячеек требуется передать.

*datatype* -- тип ячейки буфера. Для описания базовых типов Си в MPI определены константы `MPI_INT`, `MPI_CHAR`, `MPI_DOUBLE` и так далее, имеющие тип `MPI_Datatype`. Их названия образуются префиксом "MPI\_" и именем соответствующего типа (`int`, `char`, `double`, ...), записанным заглавными буквами. Пользователь может "регистрировать" в MPI свои собственные типы данных, например, структуры, после чего MPI сможет обрабатывать их наравне с базовыми. Процесс регистрации описывается в главе "Типы данных".

*dest* -- номер процесса-получателя, с которым происходит обмен данными.

*msgtag* -- идентификатор сообщения. Это целое число от 0 до 32767, которое пользователь выбирает сам. Оно служит той же цели, что и, например, расширение файла - задача-приемник:

- по идентификатору определяет смысл принятой информации;
- сообщения, пришедшие в неизвестном порядке, может извлекать из общего входного потока в нужном алгоритму порядке. Хорошим тоном является обозначение идентификаторов символьными именами посредством операторов `#define` или `const`.

*comm* -- описатель области связи (коммуникатор).

Блокирующая посылка сообщения с идентификатором *msgtag*, состоящего из *count* элементов типа *datatype*, процессу с номером *dest*. Все элементы сообщения расположены подряд в буфере *buf*. Значение *count* может быть нулем. Тип передаваемых элементов *datatype* должен указываться с помощью предопределенных констант типа. Разрешается передавать сообщение самому себе.

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу *dest*, остается за MPI. Следует специально отметить, что возврат из подпрограммы `MPI_Send` не означает ни того, что сообщение уже передано процессу *dest*, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший `MPI_Send`.

## 1.4.2 Прием сообщения

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

Аргументы функции:

OUT *buf* -- адрес начала буфера приема, в который помещаются принятые данные.

*count* – число принимаемых в сообщении элементов.

*datatype* -- тип ячейки буфера.

*source* -- номер процесса-отправителя, с которым происходит обмен данными. В качестве номера процесса-отправителя можно указать предопределенную константу `MPI_ANY_SOURCE` - признак того, что подходит сообщение от любого процесса. В качестве идентификатора принимаемого сообщения можно указать константу `MPI_ANY_TAG` - признак того, что подходит сообщение с любым идентификатором.

*msgtag* -- идентификатор сообщения. Это целое число от 0 до 32767, которое пользователь выбирает сам. Оно служит той же цели, что и, например, расширение файла - задача-приемник:

- по идентификатору определяет смысл принятой информации;
- сообщения, пришедшие в неизвестном порядке, может извлекать из общего входного потока в нужном алгоритму порядке. Хорошим тоном является обозначение идентификаторов символьными именами посредством операторов `"#define"` или `"const int"`.

*comm* -- описатель области связи (коммуникатор).

OUT *status* – параметры принятого сообщения.

Прием сообщения с идентификатором *msgtag* от процесса *source* с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения *count*. Если число принятых элементов меньше значения *count*, то гарантируется, что в буфере *buf* изменятся только элементы, соответствующие элементам принятого сообщения. Если нужно узнать точное число элементов в сообщении, то можно воспользоваться подпрограммой `MPI_Probe`.

Блокировка 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.



### 1.4.3 Посылка и прием сообщения:

В MPI введены две функции, осуществляющие одновременно посылку одних данных и прием других.

```
int MPI_Sendrecv ( void *sbuf, int scount, MPI_Datatype
stype, int dest, int stag, void *rbuf, int rcount, MPI_Datatype
rtype, int source, MPI_Datatype rtag, MPI_Comm comm, MPI_Status
*status )
```

Аргументы функции:

Прототип содержит 12 параметров: первые 5 параметров такие же, как у MPI\_Send, остальные 7 параметров такие же, как у MPI\_Recv.

*sbuf* -- адрес начала буфера передачи, из которого берутся данные для отсылки.

*scount* -- число передаваемых в сообщении элементов.

*stype* -- тип ячейки буфера отсылки.

*dest* -- номер процесса-получателя, с которым происходит обмен данными.

*stag* -- идентификатор сообщения.

OUT *rbuf* -- адрес начала буфера приема, в который помещаются принятые данные.

*rcount* -- число принимаемых в сообщении элементов.

*rtype* -- тип ячейки буфера приема.

*source* -- номер процесса-отправителя, с которым происходит обмен данными.

*rtag* -- идентификатор сообщения.

*comm* -- описатель области связи (коммуникатор).

OUT *status* -- параметры принятого сообщения.

Примечания:

- и прием, и передача используют один и тот же коммуникатор;
- порядок приема и передачи данных MPI\_Sendrecv выбирает автоматически; гарантируется, что автоматический выбор не приведет к "клинчу";
- MPI\_Sendrecv совместима с MPI\_Send и MPI\_Recv, т.е. может "общаться" с ними.

Данная операция объединяет в едином запросе посылку и прием сообщений. Принимающий и отправляющий процессы могут являться одним и тем же процессом. Сообщение, отправленное операцией MPI\_Sendrecv, может быть принято обычным образом, и точно также операция MPI\_Sendrecv может принять сообщение, отправленное обычной операцией MPI\_Send. Буфера приема и посылки обязательно должны быть различными.

**MPI\_Sendrecv\_replace** помимо общего коммуникатора использует еще и общий для приема-передачи буфер. Не очень удобно, что параметр *count* получает двойное толкование: это и количество отправляемых данных, и предельная емкость входного буфера.

Примечание:

- принимаемые данные должны быть заведомо не длиннее отправляемых;
- принимаемые и отправляемые данные должны иметь одинаковый тип;
- отправляемые данные затираются принимаемыми;
- MPI\_Sendrecv\_replace так же гарантированно не вызывает клинча.

## 1.5 Групповые (коллективные) взаимодействия

Под термином "коллективные" в MPI подразумеваются три группы функций:

- функции коллективного обмена данными;
- точки синхронизации, или барьеры;
- функции поддержки распределенных операций.

Коллективная функция одним из аргументов получает описатель области связи (коммуникатор). Вызов коллективной функции является корректным, только если произведен из всех процессов-абонентов соответствующей области связи, и именно с этим коммуникатором в качестве аргумента (хотя для одной области связи может иметься несколько коммуникаторов, подставлять их вместо друг друга нельзя). В этом и заключается коллективность: либо функция вызывается всем коллективом процессов, либо никем.

Возврат из процедуры коллективного взаимодействия может произойти в тот момент, когда участие процесса в данной операции уже закончено. Как и для блокирующих процедур, возврат означает то, что разрешен свободный доступ к буферу приема или отправки, но не означает ни того, что операция завершена другими процессами, ни даже того, что она ими начата (если это возможно по смыслу операции).

Если требуется ограничить область действия для коллективной функции только частью присоединенных к коммуникатору задач (или расширить область действия), следует создать временную группу/область связи/коммуникатор на базе существующих.

Основные особенности и отличия от коммуникаций типа "точка-точка":

- на прием и/или передачу работают одновременно ВСЕ задачи-абоненты указываемого коммуникатора;
- коллективная функция выполняет одновременно и прием, и передачу; она имеет большое количество параметров, часть которых нужна для приема, а часть для передачи; в разных задачах та или иная часть игнорируется;

- как правило, значения ВСЕХ параметров (за исключением адресов буферов) должны быть идентичными во всех задачах;
- MPI назначает идентификатор для сообщений автоматически;
- кроме того, сообщения передаются не по указываемому коммуникатору, а по временному коммуникатору-дубликату; тем самым потоки данных коллективных функций надежно изолируются друг от друга и от потоков, созданных функциями "точка-точка".

### 1.5.1 Рассылка целого сообщения процессам

```
int MPI_Bcast ( void *buf, int count, MPI_Datatype datatype,
int source, MPI_Comm comm )
```

Аргументы функции:

*OUT buf* -- адрес начала буфера передачи сообщения

*count* -- число передаваемых элементов в сообщении

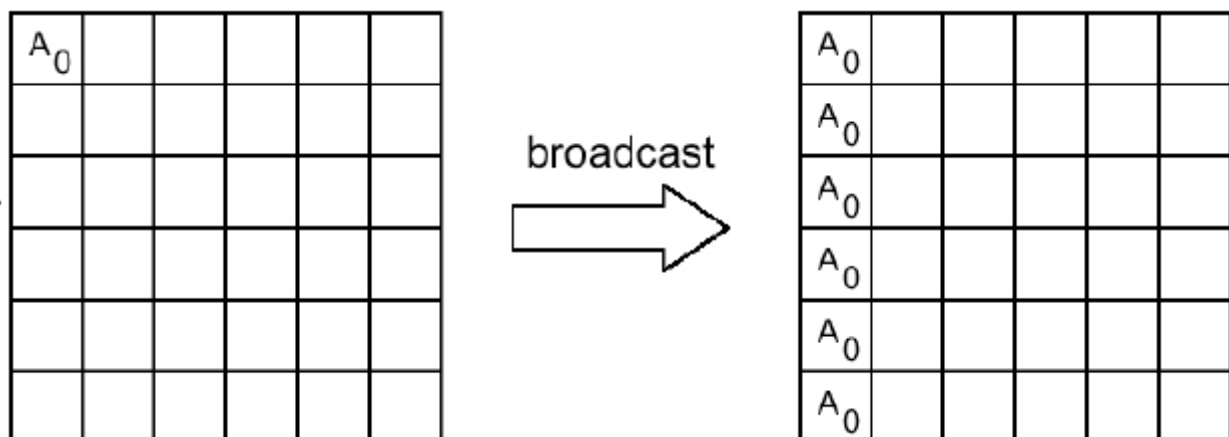
*datatype* -- тип передаваемых элементов

*source* -- номер рассылающего процесса

*comm* -- идентификатор группы

Рассылка сообщения от процесса *source* всем процессам, включая рассылающий процесс. При возврате из процедуры содержимое буфера *buf* процесса *source* будет скопировано в локальный буфер процесса. Значения параметров *count*, *datatype* и *source* должны быть одинаковыми у всех процессов.

Пример:



### 1.5.2 Сборка данных от процессов

```
int MPI_Gather ( void *sbuf, int scount, MPI_Datatype stype,
void *rbuf, int rcount, MPI_Datatype rtype, int dest, MPI_Comm
comm )
```

Аргументы функции:

*sbuf* -- адрес начала буфера передачи

*scount* -- число элементов в посылаемом сообщении

*stype* -- тип элементов отсылаемого сообщения

OUT *rbuf* -- адрес начала буфера сборки данных

*rcount* -- число элементов в принимаемом сообщении

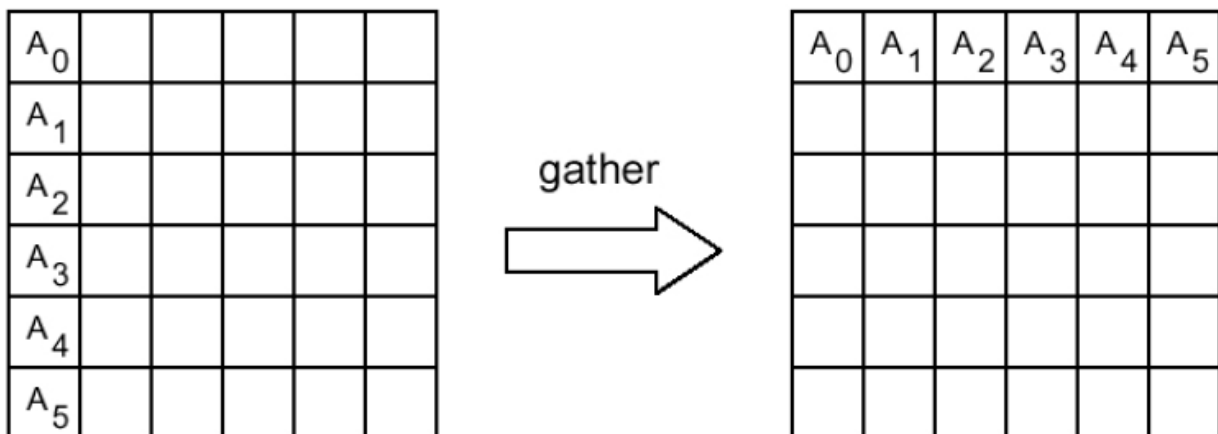
*rtype* -- тип элементов принимаемого сообщения

*dest* -- номер процесса, на котором происходит сборка данных

*comm* -- идентификатор группы

Сборка данных со всех процессов в буфере *rbuf* процесса *dest*. Каждый процесс, включая *dest*, посылает содержимое своего буфера *sbuf* процессу *dest*. Собирающий процесс сохраняет данные в буфере *rbuf*, располагая их в порядке возрастания номеров процессов. Параметр *rbuf* имеет значение только на собирающем процессе и на остальных игнорируется, значения параметров *count*, *datatype* и *dest* должны быть одинаковыми у всех процессов.

Пример:



### 1.5.3 Рассылка частей сообщения процессам

```
int MPI_Scatter ( void *sbuf, int scount, MPI_Datatype  
stype, void *rbuf, int rcount, MPI_Datatype rtype, int root,  
MPI_Comm comm )
```

Аргументы функции:

*sbuf* -- адрес начала буфера передачи

*scount* -- число элементов в посылаемом сообщении

*stype* -- тип элементов отсылаемого сообщения

OUT *rbuf* -- адрес начала буфера сборки данных

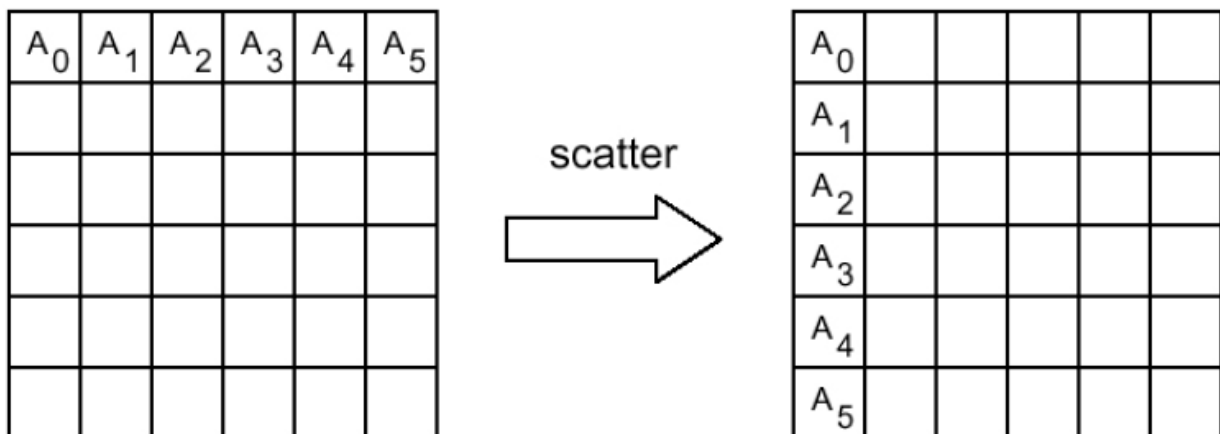
*rcount* -- число элементов в принимаемом сообщении

*rtype* -- тип элементов принимаемого сообщения

*root* -- номер процесса, который рассылает данные  
*comm* -- идентификатор группы

Процедура `MPI_Scatter` по своему действию является обратной к `MPI_Gather`. Она осуществляет рассылку по *scount* элементов данных типа *stype* из буфера *sbuf* процесса *root* в буферы *rbuf* всех процессов коммуникатора *comm*, включая сам процесс *root*. Можно считать, что массив *sbuf* делится на равные части по числу процессов, каждая из которых состоит из *scount* элементов типа *stype*, после чего I-я часть посылается (I-1)-му процессу (процессы нумеруются с 0).

Пример:



```
int MPI_Allgather ( void *sbuf, int scount, MPI_Datatype
stype, void *rbuf, int rcount, MPI_Datatype rtype, MPI_Comm
comm )
```

Аргументы функции:

*sbuf* -- адрес начала буфера передачи

*scount* -- число элементов в посылаемом сообщении

*stype* -- тип элементов отсылаемого сообщения

OUT *rbuf* -- адрес начала буфера сборки данных

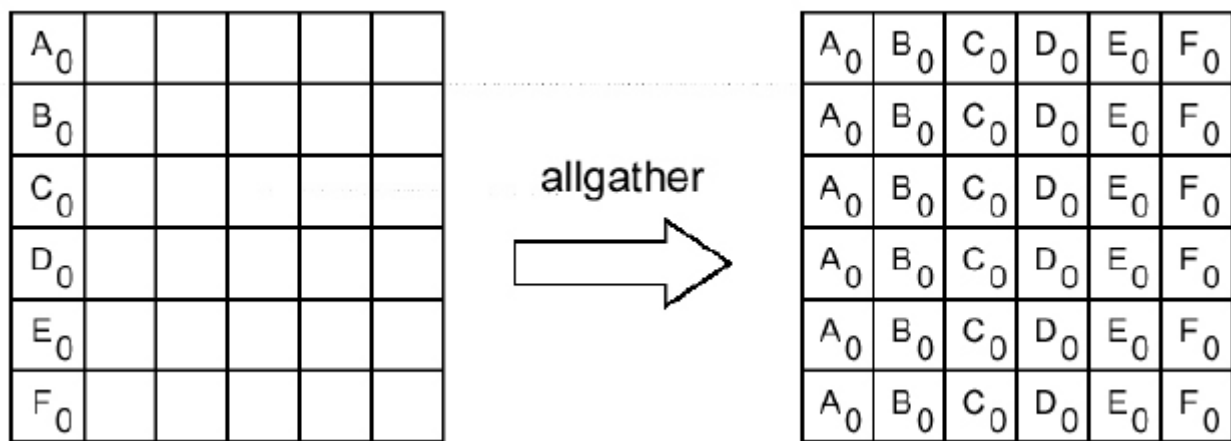
*rcount* -- число элементов в принимаемом сообщении

*rtype* -- тип элементов принимаемого сообщения

*comm* -- идентификатор группы

Сборка данных из массивов *sbuf* со всех процессов коммуникатора *comm* в буфере *rbuf* каждого процесса. Данные сохраняются в порядке возрастания номеров процессов. Блок данных, посланный процессом I-1, размещается в I-ом блоке буфера *rbuf* принимающего процесса. Операцию можно рассматривать как `MPI_Gather`, при которой результат получается на всех процессах коммуникатора *comm*.

Пример:



### 1.5.4 Обмен различными порциями данных между всеми процессами

```
int MPI_Alltoall ( void *sbuf, int scount, MPI_Datatype
stype, void *rbuf, int rcount, MPI_Datatype rtype, MPI_Comm
comm )
```

Аргументы функции:

*sbuf* -- адрес начала буфера передачи

*scount* -- число элементов в посылаемом сообщении

*stype* -- тип элементов отсылаемого сообщения

OUT *rbuf* -- адрес начала буфера сборки данных

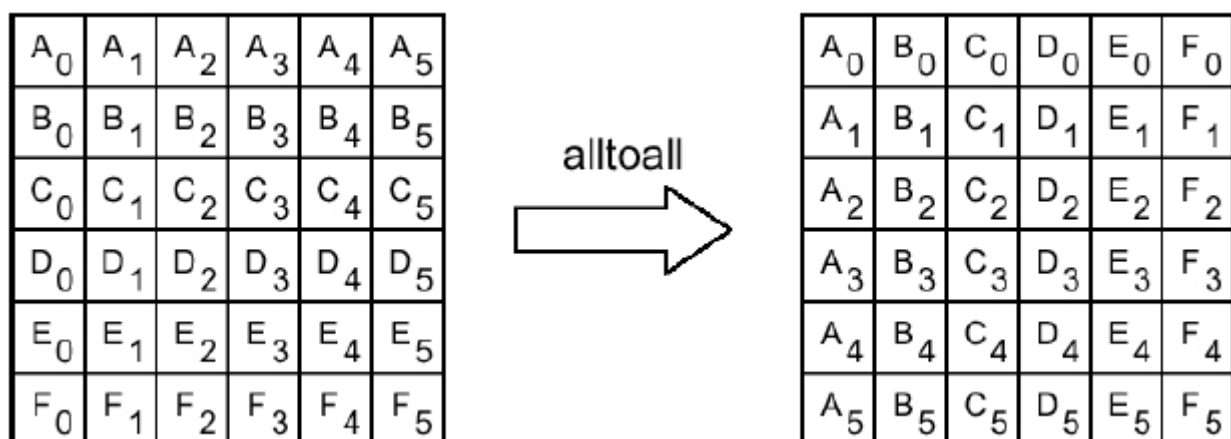
*rcount* -- число элементов в принимаемом сообщении

*rtype* -- тип элементов принимаемого сообщения

*comm* -- идентификатор группы

Рассылка каждым процессом коммуникатора *comm* различных порций данных всем другим процессам. J-ый блок буфера *sbuf* (I-1)- го процесса попадает в I-ый блок данных буфера *rbuf* (J-1)-го процесса.

Пример:



## 1.6 Функции поддержки распределенных операций

### 1.6.1 Выполнение глобальных операций с возвратом результатов в главный процесс

```
int MPI_Reduce ( void *sbuf, void *rbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm )
```

Аргументы функции:

*sbuf* -- адрес начала буфера для аргументов

OUT *rbuf* -- адрес начала буфера для результата

*count* -- число аргументов у каждого процесса

*datatype* -- тип аргументов

*op* -- идентификатор глобальной операции

*root* -- процесс-получатель результата

*comm* -- идентификатор группы

Выполнение *count* глобальных операций *op* с возвратом *count* результатов в буфер *rbuf* главного процесса *root*. Операция выполняется независимо над соответствующими аргументами всех процессов. Значения параметров *count* и *datatype* у всех процессов должны быть одинаковыми. Из соображений эффективности реализации предполагается, что операция *op* обладает свойствами ассоциативности и коммутативности.

Идентификаторы глобальных операций:

MPI\_MAX -- поэлементный максимум;

MPI\_MIN -- поэлементный минимум;

MPI\_SUM -- поэлементная сумма;

MPI\_PROD -- поэлементное произведение;

MPI\_LAND, MPI\_BAND -- логическая и двоичная операции И;

MPI\_LOR, MPI BOR -- логическая и двоичная операции ИЛИ;

MPI\_LXOR, MPI\_BXOR -- логическая и двоичная операции исключающее ИЛИ;

MPI\_MAXLOC, MPI\_MINLOC - поиск индексированного минимума/максимума.

### 1.6.2 Выполнение глобальных операций с возвратом результатов во все процессы

```
int MPI_Allreduce ( void *sbuf, void *rbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

Аргументы функции:

*sbuf* -- адрес начала буфера для аргументов

OUT *rbuf* -- адрес начала буфера для результата

*count* -- число аргументов у каждого процесса  
*datatype* -- тип аргументов  
*op* -- идентификатор глобальной операции  
*comm* -- идентификатор группы

Функция аналогична предыдущей, но результат будет записан в буферы *rbuf* всех процессов.

### 1.6.3 Выполнение глобальных операций с возвратом части результатов во все процессы

```
int MPI_Reduce_scatter ( void *sbuf, void *rbuf, int  
*rcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

Аргументы функции:

*sbuf* -- адрес начала буфера для аргументов

OUT *rbuf* -- адрес начала буфера для результата

*rcounts* -- массив, в котором хранятся количества элементов буфера, передающиеся каждому процессу

*datatype* -- тип аргументов

*op* -- идентификатор глобальной операции

*comm* -- идентификатор группы

Функция аналогична *MPI\_Reduce*, но каждая задача получает не весь массив-результат, а его часть. Длины этих частей находятся в массиве-третьем параметре функции. Размер исходных массивов во всех задачах одинаков и равен сумме длин результирующих массивов.

### 1.6.4 Выполнение независимых частичных операций над элементами массивов

```
int MPI_Scan ( void *sbuf, void *rbuf, int count,  
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
```

Аргументы функции:

*sbuf* -- адрес начала буфера для аргументов

OUT *rbuf* -- адрес начала буфера для результата

*count* -- массив, в котором хранятся количества элементов буфера, передающиеся каждому процессу

*datatype* -- тип аргументов

*op* -- идентификатор глобальной операции



*comm* -- идентификатор группы

Выполнение *count* независимых частичных операций *op* над соответствующими элементами массивов *sbuf*. *I*-ый процесс выполняется *count* глобальных операций над соответствующими элементами массива *sbuf* процессов с номерами от 0 до *I* включительно и помещает полученный результат в массив *rbuf*. Полный результат глобальной операции получается в массиве *rbuf* последнего процесса.

## 1.7 Синхронизация процессов

Блокировка работы процессов:

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

Аргументы функции:

*comm* -- идентификатор группы

Блокирует работу процессов, вызвавших данную процедуру, до тех пор, пока все оставшиеся процессы группы *comm* также не выполнят эту процедуру. Гарантирует, что к выполнению следующей за *MPI\_Barrier* инструкции каждая задача приступит одновременно с остальными.

Это единственная в *MPI* функция, вызовами которой гарантированно синхронизируется во времени выполнение различных ветвей! Некоторые другие коллективные функции в зависимости от реализации могут обладать, а могут и не обладать свойством одновременно возвращать управление всем ветвям; но для них это свойство является побочным и необязательным.

## 2 Пример задачи, использующий функции MPI.

Описание задачи: реализация алгоритма вычисления приближенного значения  $\pi$  в виде интеграла.

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
double f( double a );

double f( double a )      /* Функция интеграла */
{
    return (4.0 / (1.0 + a*a));
}

int main( int argc, char *argv[])
{
    /* Объявление переменных:
    - флаг окончания вычислений
    - параметр погрешности вычисления
    - порядковый номер задачи
    - общее количество задач
    - параметр цикла
    */
    int done = 0, n, myid, numprocs, i;

    /* Эталонное значение  $\pi$  */
    double PI25DT = 3.141592653589793238462643;

    /*
    - текущее значение  $\pi$ 
    - эталонное значение  $\pi$ 
    - шаг интеграла
    - сумма при вычислении интеграла
    - текущий аргумент подынтегральной функции
    */
    double mypi, pi, h, sum, x;

    /*
    - начальное и конечное время вычисления интеграла
    */
    double startwtime = 0.0, endwtime;

    /* Инициализация библиотеки */
    MPI_Init(&argc, &argv);
    /* Получаем общее количество запущенных процессов */
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    /* Получаем порядковый номер текущего процесса */
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    n = 0;
    while (!done)
    {
```

```

        if (myid == 0)      {
            if (n==0) n=100; else n=0;
/* Засекаем время начала вычисления */
            startwtime = MPI_Wtime();
        }
/* Рассылаем содержимое буфера задачи во все остальные */
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) done = 1;
        else {
            h = 1.0 / (double) n;
            sum = 0.0;
            for (i = myid + 1; i <= n; i += numprocs)
            {
                x = h * ((double)i - 0.5);      sum += f(x);
            }
            mypi = h * sum;
/* Массив с результатами размещается в данной задаче */
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
            if (myid == 0)      {
                printf("pi is approximately %.16f, Error is %.16f\n",
                    pi, fabs(pi - PI25DT));
                endwtime = MPI_Wtime(); /*Засекаем время окончания вычисления */
/* Вывод общего времени вычисления */
                printf("wall clock time = %f\n", endwtime-startwtime);
            }
        }
        MPI_Finalize(); /* Нормальное закрытие библиотеки */
        return 0;
    }
}

```

### 3 Установка, компиляция и запуск программ MPI

В состав программного инструментария MPI входят, как правило, два обязательных компонента:

- библиотека программирования для языков Си, Си++;
- среда исполнения и загрузчик исполняемых файлов.

#### 3.1 Установка Microsoft MPI

В качестве базовой реализации MPI предлагается для использования пакет Microsoft MPI, доступный по адресу: <https://goo.gl/Ra2clT> . Он включает в себя библиотеку программирования (пакет msmtpsdk.msi) и загрузчика (пакет MSMpiSetup.exe).

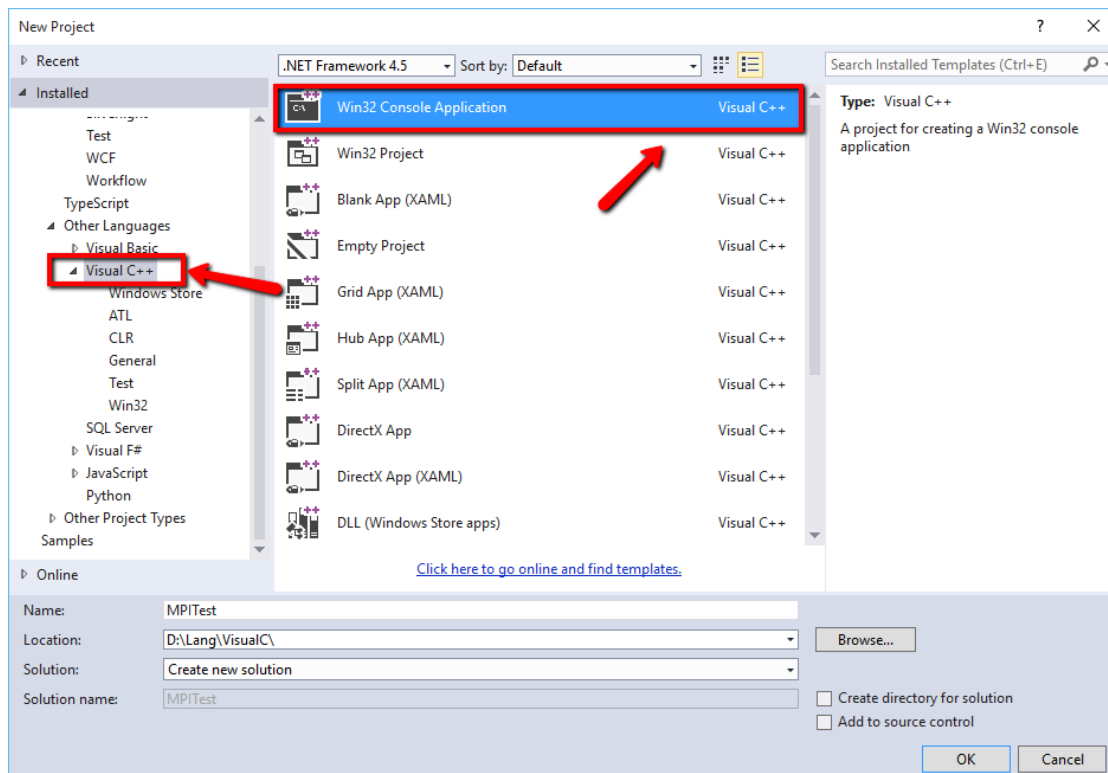
Для работы с MPI необходимо установить и тот и другой пакет в любом порядке. По умолчанию пакеты устанавливаются в следующие каталоги:

- C:\Program Files (x86)\Microsoft SDKs\MPI – библиотека (далее SDK)
- "C:\Program Files\Microsoft MPI\" – загрузчик (далее MPI).

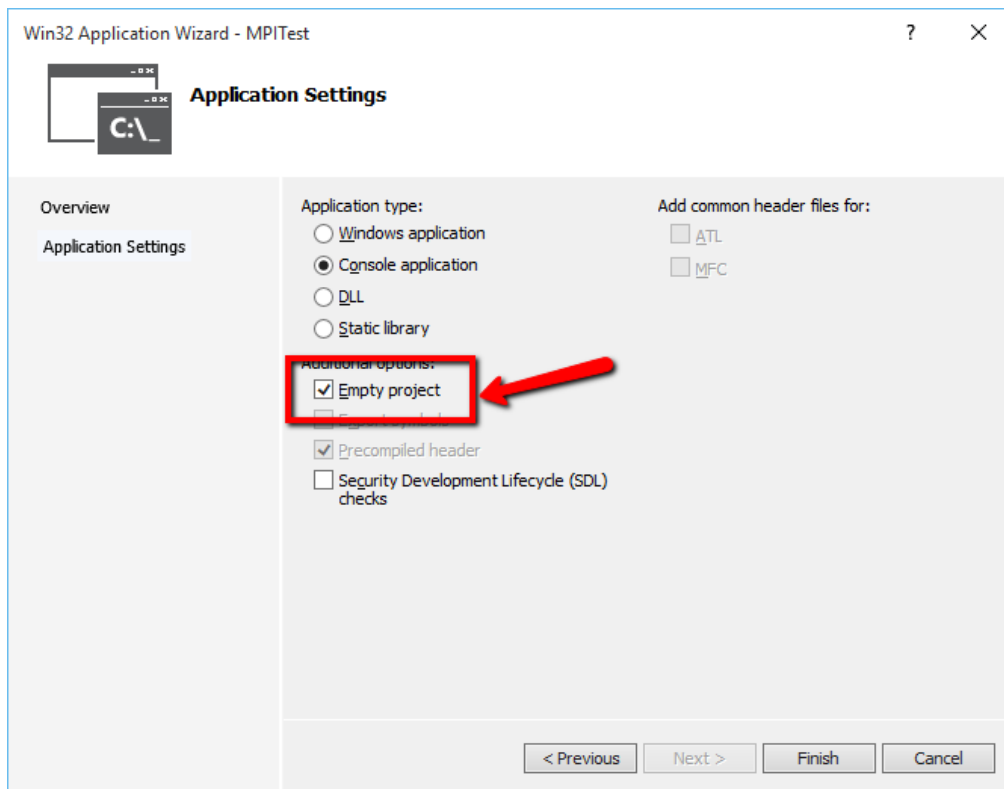
Дальнейшее описание приведено исходя из этих путей. Если вы установили в другие каталоги, учитывайте это в дальнейшей работе.

#### 3.2 Компиляция в Microsoft Visual Studio (на примере VS 2013)

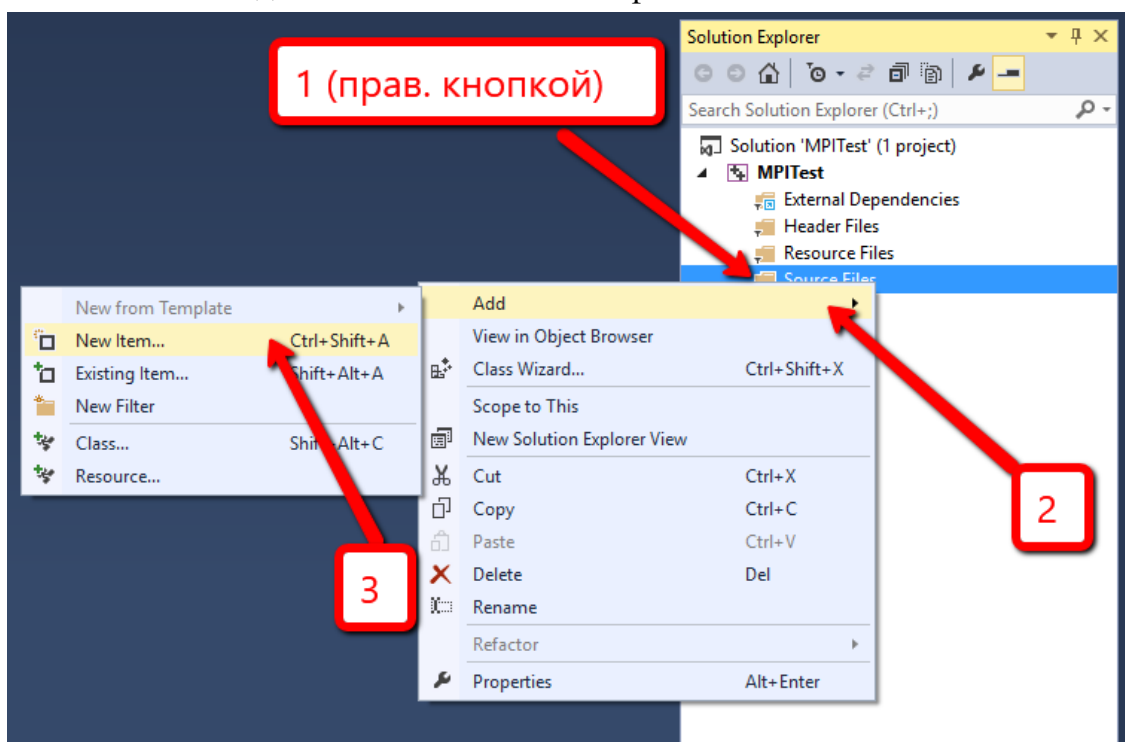
##### 1. Создайте новый проект (C++, Win32, Консольное приложение)



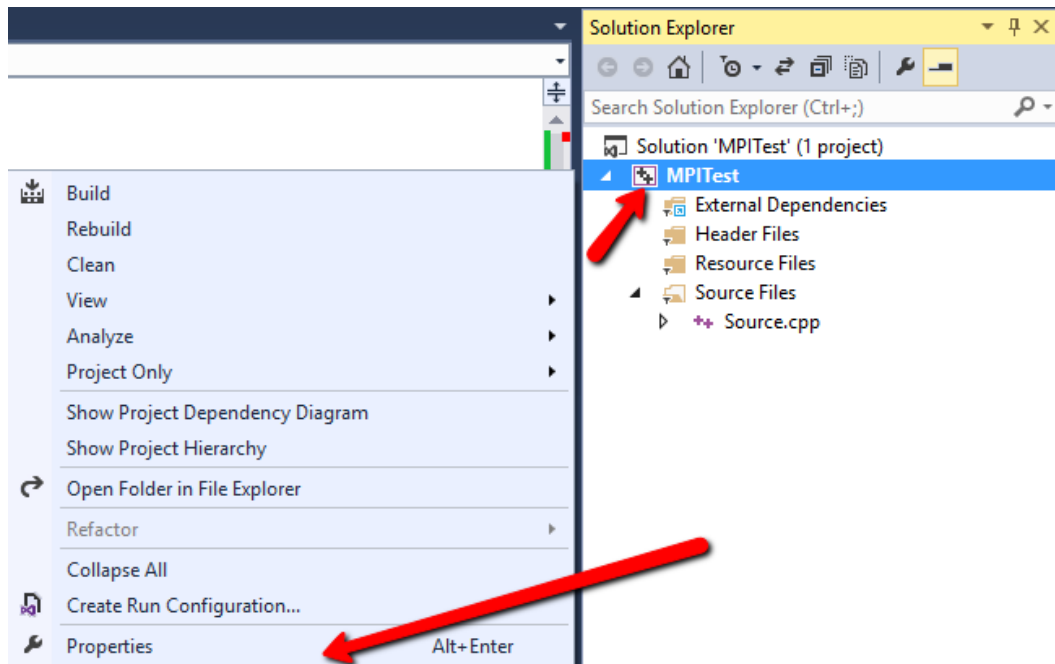
Очень рекомендуется установить галку «Пустой проект»:



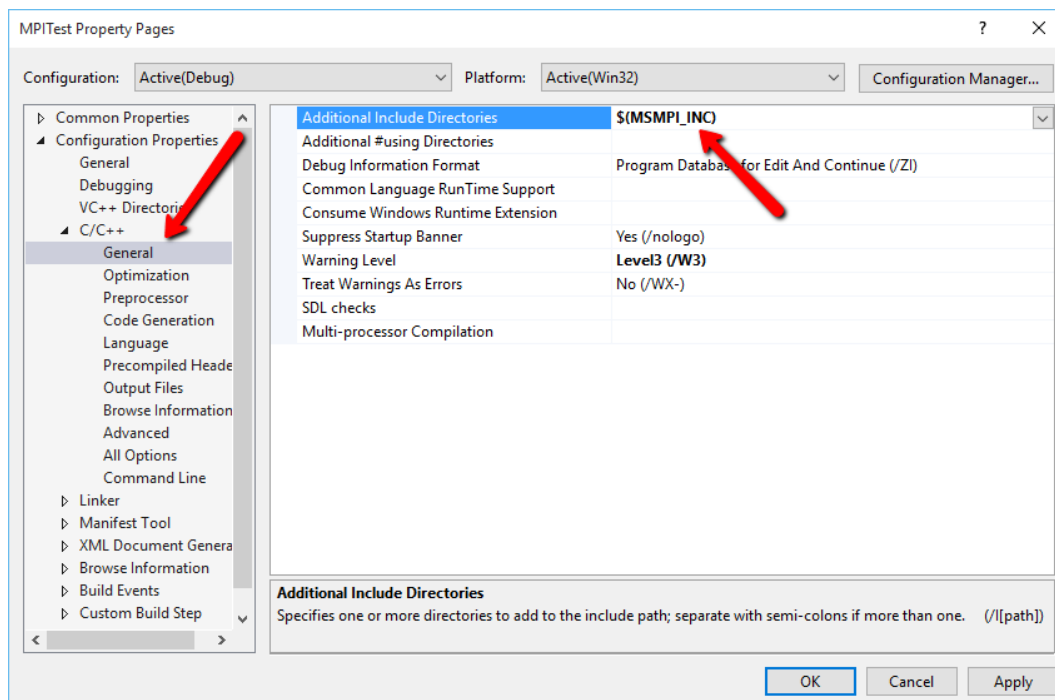
и после этого создать в нем новый C/C++ файл:



2. В настройках проекта:

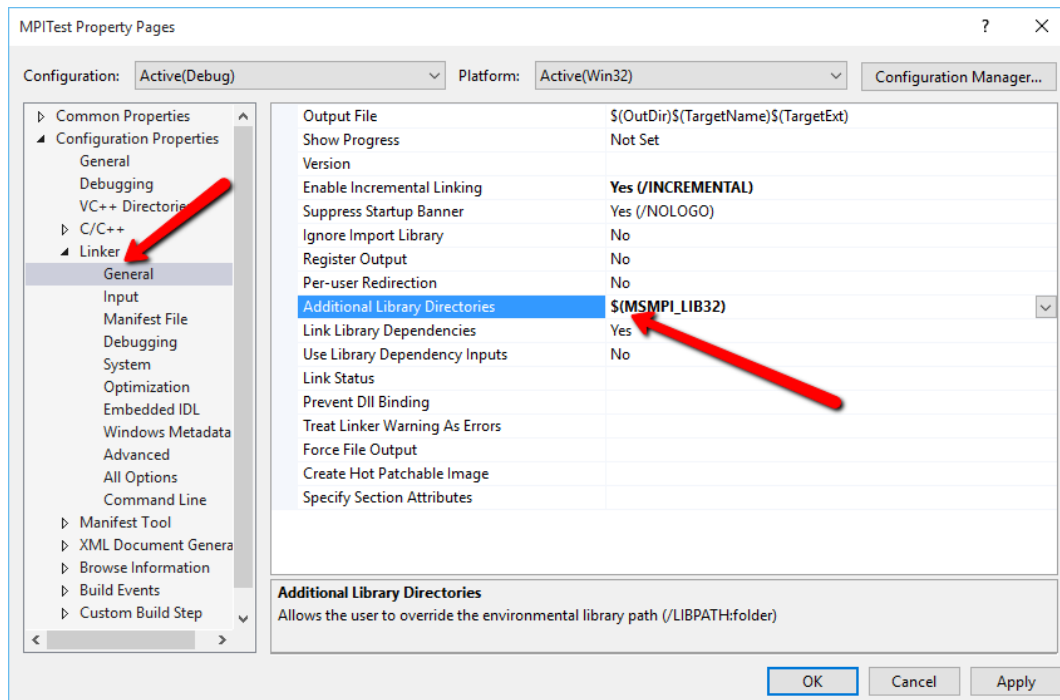


в Additional Include Directories:



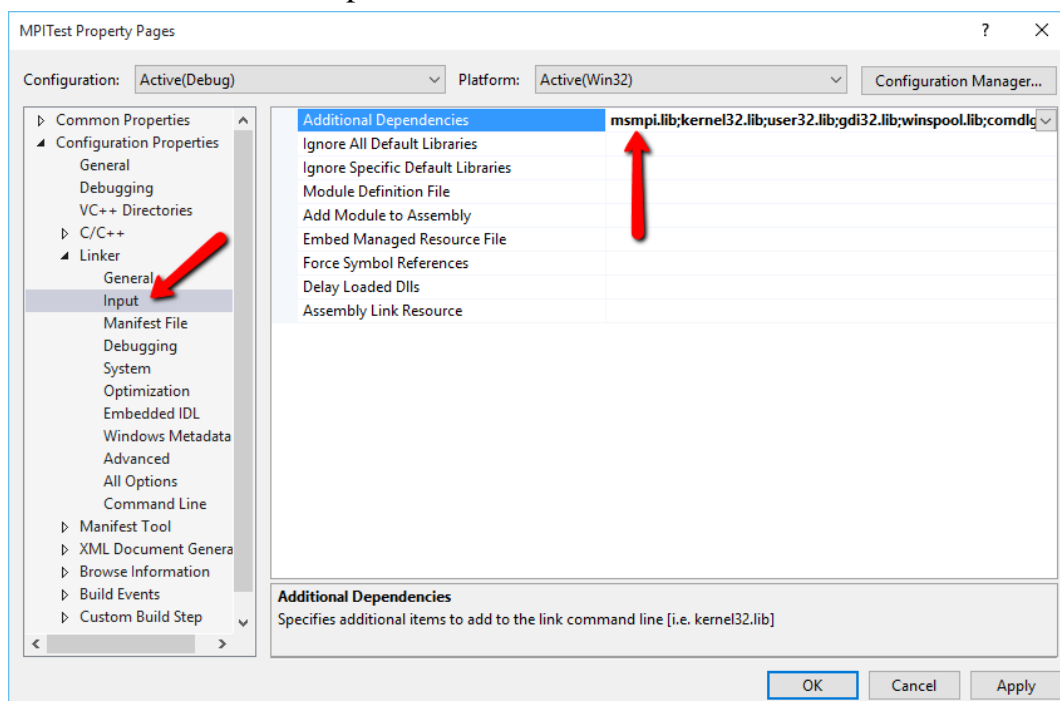
добавить магическую комбинацию: \$(MSMPI\_INC)

3. Там же в настройках проекта в Additional Library Directories:



добавить магическую комбинацию: **\$(MSMPI\_LIB32)**

#### 4. Там же в Additional Dependencies



## Добавьте библиотеку **msmpi.lib**

5. Скомпилируйте приложение.

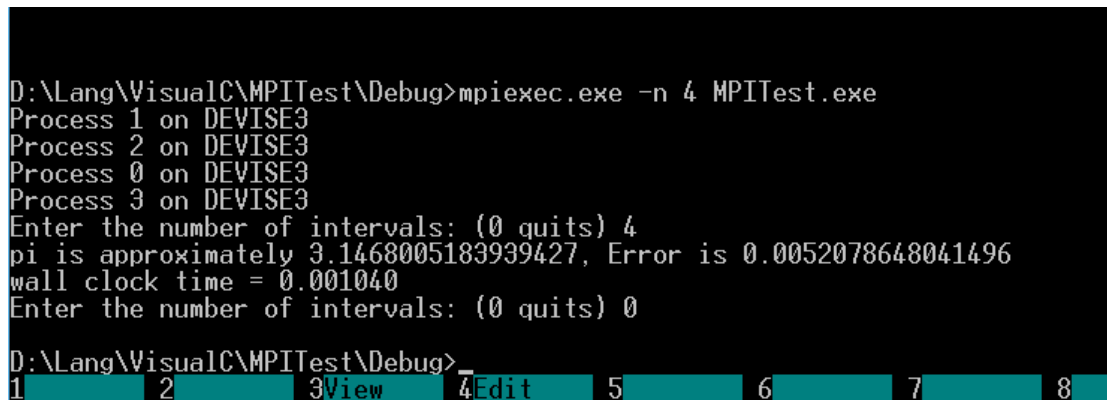
### 3.3 Запуск скомпилированной программы в командной строке

Для запуска MPI программы, используется программа **mpiexec.exe**, которая расположена в директории "C:\Program Files\Microsoft MPI\Bin\". Чтобы запустить программу на, например, 4 процессорах (4 потоках) нужно использовать команду:

```
mpiexec.exe -n 4 mpitest.exe
```

Здесь значение 4 – количество процессов, на которых производится вычисление задачи, а mpitest.exe – имя файла запускаемой задачи.

Результат будет похож на такой:



```
D:\Lang\VisualC\MPITest\Debug>mpiexec.exe -n 4 MPITest.exe
Process 1 on DEVISE3
Process 2 on DEVISE3
Process 0 on DEVISE3
Process 3 on DEVISE3
Enter the number of intervals: (0 quits) 4
pi is approximately 3.1468005183939427, Error is 0.0052078648041496
wall clock time = 0.001040
Enter the number of intervals: (0 quits) 0
D:\Lang\VisualC\MPITest\Debug>
```



## 4 Задания к лабораторным работам.

### 4.1 Замечания к выполнению лабораторных работ

**Процессы:** вне зависимости от количества процессов в задании программа должна работать корректно на любом числе процессов. Под термином «корректно» подразумевается:

- Выдавать одинаковый результат в задачах, где от числа процессов не зависит математический результат вычислений (первая, часть заданий второй и третья лабораторные работы).
- Выдавать различные, но правильные результаты в задачах, где математический результат зависит от числа процессов (например – часть задач 2 лабораторной работы).

**Вычисления:** использование распределенных вычислений должно распределять нагрузку и сокращать время выполнения задачи, а не увеличивать время и количество вычислений относительно однозадачного выполнения. Например:

- Если вектор должен быть передан от одного процесса другим процессам, то этот вектор нужно создавать только в том процессе, который будет рассылать данные, а не во всех.

**Коммуникации:** так как вычислители (процессы, процессоры или вычислительные машины, которые реализуют вычисления) заведомо быстрее, чем коммуникации, следовательно, передача данных между процессорами должна быть минимально необходимой. Например:

- Если при сортировке матрицы отдельные процессы должны сортировать отдельные столбцы (строки), то не следует передавать процессам всю матрицу. Передавать следует непосредственно строки, которые будут отсортированы.
- Особенно не стоит использовать метод Alltoall для транспонирования матриц.

## 4.2 Лабораторная работа №1. Пересылка данных.

### 4.2.1 Цель работы

Переслать вектор, размерности M, N процессам, используя различные виды связи между процессами. Элементы вектора задаются произвольно. Элементы вектора пересылаемого и принятого вектора, а также время выполнения должны быть выведены на экран.

### 4.2.2 Содержание отчета

Задание.

Текст программы.

Результат работы программы.

### 4.2.3 Варианты заданий

№ варианта	M	N	Функция
1	3	3	MPI_Send
2	5	4	MPI_Bcast
3	7	5	MPI_Gather
4	10	3	MPI_Scatter
5	13	4	MPI_Allgather
6	15	5	MPI_Alltoall
7	3	3	MPI_Alltoall
8	5	4	MPI_Allgather
9	7	5	MPI_Scatter
10	10	3	MPI_Gather
11	13	4	MPI_Bcast
12	15	5	MPI_Send
13	3	3	MPI_Gather
14	5	4	MPI_Allgather
15	7	5	MPI_Send
16	10	3	MPI_Alltoall
17	13	4	MPI_Scatter
18	15	5	MPI_Bcast

## 4.3 Лабораторная работа №2. Распределенные вычисления.

### 4.3.1 Цель работы

Сгенерировать в каждом из N процессов вектор чисел. Размерность вектора – M. Произвести поэлементную обработку всех векторов и поместить результирующий вектор в каком-либо процессе. В работе использовать средства MPI для организации распределенных вычислений (см. раздел 1.6).

### 4.3.2 Содержание отчета

Задание.

Текст программы.

Результат работы программы.

### 4.3.3 Варианты заданий

№ вар-та	N	M	Тип элемента вектора	Тип поэлементной обработки
1	3	10	Целый байтовый	Поэлементный максимум
2	4	20	Знаковый короткий целый	Поэлементное умножение
3	5	30	Без знаковый целый	Поэлементная сумма
4	6	40	Длинное целое	Двоичное И
5	7	50	Целое	Двоичное искл. ИЛИ
6	3	10	Целое	Поэлементный максимум
7	4	20	Длинное целое	Поэлементное умножение
8	5	30	Знаковый короткий целый	Поэлементная сумма
9	6	40	Без знаковый целый	Двоичное И
10	7	50	Целый байтовый	Двоичное искл. ИЛИ
11	3	10	Целый байтовый	Двоичное искл. ИЛИ
12	4	20	Знаковый короткий целый	Двоичное И
13	5	30	Без знаковый целый	Поэлементное умножение
14	6	40	Длинное целое	Поэлементная сумма
15	7	50	Целое	Поэлементный максимум
16	3	30	Длинное целое	Двоичное И
17	4	40	Знаковый короткий целый	Поэлементный максимум
18	5	50	Целый байтовый	Поэлементное умножение

## 4.4 Лабораторная работа №3. Сортировка элементов массива.

### 4.4.1 Цель работы

Произвести сортировку элементов в столбцах (или строках) матрицы размерности  $N \times M$ , с использованием распределения вычислений между процессами средствами MPI.

**Замечание:** Программа должна работать корректно на **ЛЮБОМ** количестве процессов. Матрица при этом **НЕ ДОЛЖНА** менять свои размеры.

### 4.4.2 Содержание отчета

Задание.

Текст программы.

Результат работы программы.

### 4.4.3 Варианты заданий

№ варианта	N	M	Тип элемента матрицы	Тип сортировки
1	3	10	Целый байтовый	По убыванию
2	4	9	Знаковый короткий целый	По возрастанию
3	5	8	Без знаковый целый	По возрастанию
4	6	7	Длинное целое	По убыванию
5	7	6	Целое	По возрастанию
6	8	5	Целое	По убыванию
7	9	4	Длинное целое	По возрастанию
8	10	3	Знаковый короткий целый	По возрастанию
9	3	10	Без знаковый целый	По убыванию
10	4	9	Целый байтовый	По возрастанию
11	5	8	Целый байтовый	По убыванию
12	6	7	Знаковый короткий целый	По возрастанию
13	7	6	Без знаковый целый	По возрастанию
14	8	5	Длинное целое	По убыванию
15	9	4	Целое	По возрастанию
16	10	3	Длинное целое	По убыванию
17	3	4	Знаковый короткий целый	По убыванию
18	4	5	Целый байтовый	По возрастанию

## **Библиографический список**

### **Русскоязычные источники:**

Корнеев В.Д. "Параллельное программирование в MPI", изд-во СО РАН, Новосибирск, 2000, 213 стр.

Воеводин В.В. Курс лекций "Параллельная обработка данных".  
<http://parallel.ru/vvv/index.html>.

Шпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI: Пособие - Мн.: БГУ, 2002. -323 с.

MPI: The Message Passing Interface. [http://parallel.ru/tech/tech\\_dev/mpi.html](http://parallel.ru/tech/tech_dev/mpi.html).

### **Англоязычные источники:**

MPI: A Message-Passing Interface Standard.

<http://parallel.ru/docs/Parallel/mpi1.1/mpi-report.html>.

MPICH-A Portable Implementation of MPI.

<http://www-unix.mcs.anl.gov/mpi/mpich>.