Технология MPI реализации параллельных распределенных вычислений.

ОБЩИЕ СВЕДЕНИЯ О БИБЛИОТЕКЕ MPI И ЕЕ ЦЕЛИ

Библиотека MessagePassingInterface (MPI) является средством, позволяющим выполнять создание параллельных распределенных приложений, и является стандартом при реализации распределено выполняющихся программ. Под MPI подразумевается модель реализации задач, взаимодействующих посредством параллельной передачи сообщений, в которой через совместные операции над каждым из процессов данные перемещаются из адресного пространства одного процесса в адресное пространство другого. MPI не является языком программирования, все операции MPI выражаются в виде функций, подпрограмм, или методов с соответствующими привязками к языку (C/C++, Fortran-77, 95).

Рисунок. 1.1 –Высокоуровневое представление MPI

Главное преимущество модели передачи сообщений, реализуемой в MPI, является его переносимость, легкость использования, а также возможность реализации как в вычислительных системах с общей памятью, так и в вычислительных системах с распределенной памятью. Необходимость использования библиотеки MPI и модели взаимодействия процессов посредством передачи сообщений является очевидной в вычислительной системе с распределенной памятью, в которой взаимодействие хостов (рабочих станций) реализуется посредством коммуникационной среды.

Реализация параллелизма в MPI

MPI-программа состоит из независимых процессов, выполняющих свой собственный код, т.е. реализует MIMD (**Multiple Instruction Multiple Data**) подход к организации параллельных программ. При реализации MPI-программ операционная система для каждого процесса размещает копию выполняемой программы в оперативной памяти отдельной компоненты (хоста, рабочей станции в сети) вычислительной системы. Таким образом, независимо от конфигурации системы (Рис. 1.2–1. 4) код программы располагается на всех процессах. Для запуска требуемого количества копий программы необходимо в исполняемой среде указать необходимое количество процессов (запускаемых копий программы) .

Каждый процесс начинает выполнять свою собственную копию кода. Различные процессы могут выполнять различные участки кода посредством ветвления внутри программы, т.е. коды, выполняемые процессом, не обязательно должны быть идентичными (как правило, они являются различными). Для определения участков кода в программе, которые должны выполнять процессы (участка кода, который должен выполнять каждый процесс) используются ранги процессов, но могут быть созданы и более сложные структуры распределения заданий для каждого процесса.

Ветвление для процессов на основе рангов рассмотрено на Рис. 1.5.

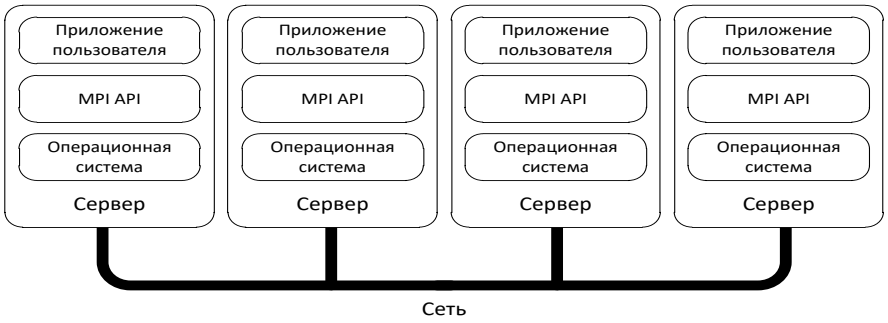

Рисунок 1.2 – Вычислительная система с четырьмя рабочими станциями

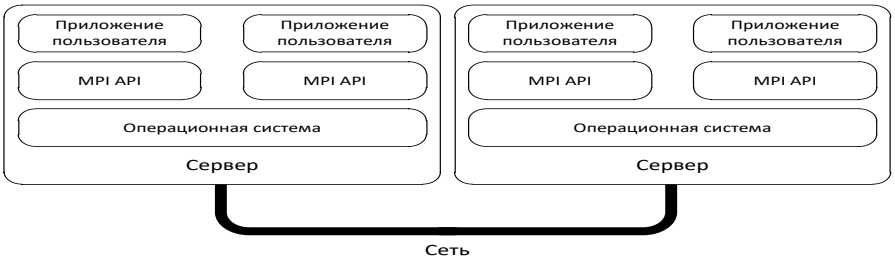

Рисунок 1.3 – Вычислительная система с двумя рабочими станциями

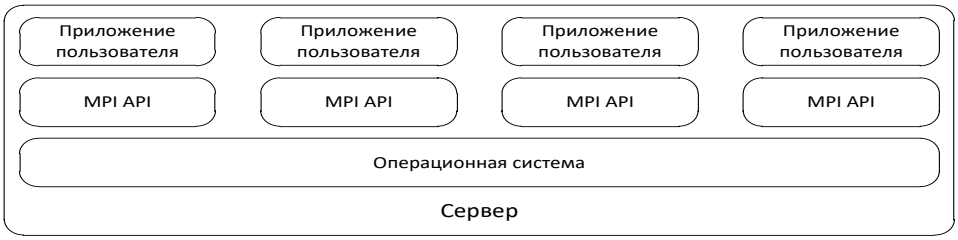

Рисунок 1.4 – Вычислительная система с одной рабочей станцией

Таким образом, каждый процесс исполняется в собственном адресном пространстве. Процессы взаимодействуют посредством вызовов функций

библиотеки MPI, обеспечивающих передачу сообщений. Одно из преимуществ MPI –это достижение переносимости исходного кода. Это значит, что программа, использующая MPI и подчиняющаяся существующим стандартам языка, при написании уже является переносимой, т.е. не требуется вносить какие-либо изменения в код при переносе программы с одной системы на другую.

Любая реализация MPI-программ требует некоторые начальные установки, перед тем как какие-либо другие MPI-подпрограммы могли бы быть вызванными. Для этого в программе на C++ должен быть выполнен вызов следующей MPI-функции (функции инициализации):

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

in **argc** - указатель на счетчик аргументов командной строки; in **argv** - указатель на список аргументов;

Любая MPI программа должна содержать только один вызов инициализирующей подпрограммы: **MPI_Init** или **MPI_Init_thread** (для многопотоковых процессов).

Также каждый процесс должен вызывать функцию завершения после любого использования MPI-функций:

intMPI_Finalize(void)

Эта подпрограмма очищает всю MPI-систему.

Таким образом, структура любой MPI-программы следующая:

```
int main(int argc, char **argv)
{
   MPI_Init(&argc, &argv);
   /* анализ аргументов */
   /* основная часть */
   MPI_Finalize();
}
```

Для того чтобы получить количество выполняемых копий процесса, используется следующая функция:

int MPI_Comm_size(MPI_Comm comm, int *size)
Атрибутами функции являются:
in **comm** –коммуникатор; out **size** - количество процессов в группе **comm**

Если для аргумента **comm** указать константу **MPI_COMM_WORLD**, функция вернет количество всех доступных процессов программы. Аналогичным образом, при указании данной константы в качестве значения аргумента **comm** в рассматриваемых функциях действия выполняются со всеми процессами программы.

Ранг (идентификатор) копии программы (ранг процесса) в отдельной группе коммуникатора можно определить с помощью соответствующей функции: **int MPI_Comm_rank(MPI_Comm comm, int *rank)ее атрибутами функции** являются**:**

in **comm** –коммуникатор; out **rank** - ранг вызывающего процесса в группе **comm**

При получении от того же коммуникатора значение **size**, значение ранга будет лежать в диапазоне от **0** до **size-1**.

На рисунке 1.5 рассмотрен пример MPI-программы для трех процессов. В соответствии с рангом, каждый процесс выполняет определенный только для него код (серым цветом выделены участки кода, которые процесс не выполняет). Перед завершением каждый процесс совершает вызов функции **MPI_Barrier()**, которая приостанавливает выполнение процесса до тех пор, пока все процессы не совершат вызов данной функции (выполняется взаимная синхронизация выполняющихся процессов).

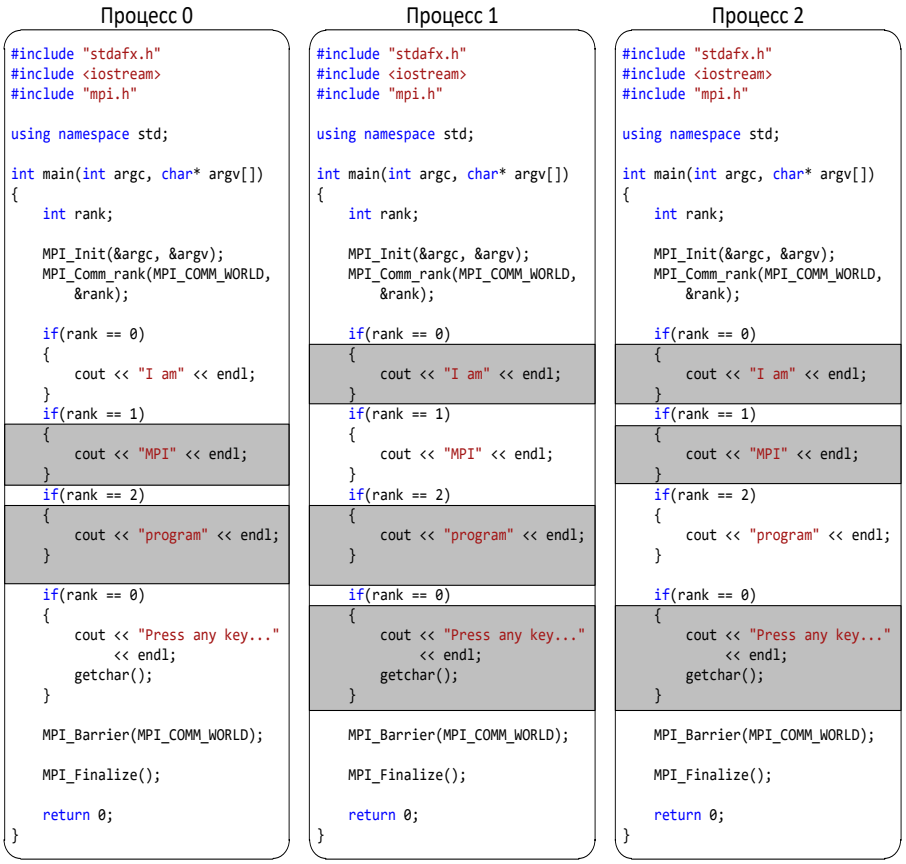

Рисунок 1.5 – Пример ветвления при организации выполнения процессов

Тип передачи данных «точка-точка»

Базовым механизмом связи между процессами в MPI является передача типа «точка-точка», в которой реализуются операции отправки (**send)** и приема (**receive**) сообщений (рис. 1.1).

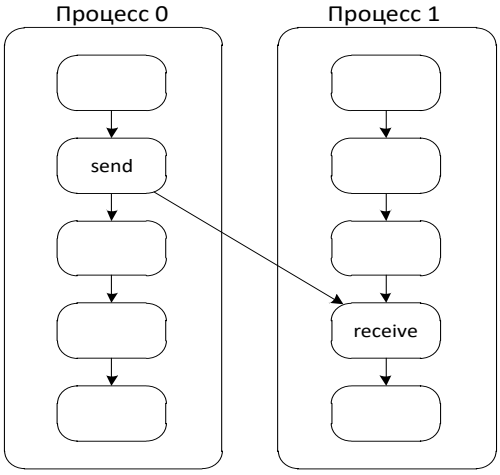

Рисунок 1.1 – Взаимодействие процкссов типа «точка-точка»

Одной из функций, реализующей отправку сообщений, является **MPI_Send()**. Она представляет собой операцию с блокировкой, ее завершение происходит после совершения передачи, синтаксис функции следующий:

int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,int tag, MPI_Comm comm);

Атрибутами функции MPI_Send() **являются: in** buf**– указатель на буфер; in** Count**– количество элементов в буфере; in** Datatype **- тип данных буфера; in** Dest **- ранг пункта назначения (процесса приемника); in** Tag **- метка сообщения; in** Comm **– коммуникатор;**

Буфер отправки, определяемый операцией **MPI_Send**, состоит из некоторой последовательности элементов, количество которых указывается в аргументе **count**. Тип данных элементов передается в аргумент **datatype**. Указатель на последовательность данных содержится в переменной **buf**. Необходимо заметить, что длина сообщения указывается количеством элементов, а не количеством байтов, так как реализуемый код не зависит от машины, на которой он будет исполняться. Аргумент **count** может быть равен нулю, в таком случае информационная часть сообщения является пустой. Основные типы данных, которые могут быть переданы в сообщении, соответствуют основным типам данных языка С (табл. 1.1).

Таблица 1.1. Связь типов данных в MPI

Типы данных MPI	Типы данных C
MPI_CHAR	char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG (синоним)	signed long long int
MPI_SIGNED_CHAR	signedchar
MPI_UNSIGNED_CHAR	unsignedchar
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX	float _Complex

Типы данных MPI	Типы данных C
MPI_C_DOUBLE_COMPLEX	double _Complex

Аргумент **comm** является коммуникатором, определяющим среду, в которой выполняется связь между процессами. Каждая такая среда обеспечивает некую отделенную «коммуникационную сферу» (**communication universe**), в которой осуществляется прием сообщений. Коммуникатор также определяет ряд процессов, которые делят между собой данную сферу. Эта **группа процессов** упорядочена, и процессы в ней идентифицируются по их рангу. Поэтому диапазон значений для аргумента **dest** может принимать значения в диапазоне от **0** до **n-1**, где **n** это число процессов в группе.

Для того чтобы принять отправленное сообщение необходимо использовать функцию приема MPI_Recv(). Синтаксис функции блокирующего приема следующий:

Int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,int tag, MPI_Comm comm, MPI_Status *status);

Атрибутами функции являются: out **Buf** - указатель на буфер; in **Count** - Количество элементов в буфере (неотрицательное значение); in **Datatype** - Тип данных буфера; in **Source** - Ранг пункта отправки; in **Tag** - Метка сообщения; in **Comm** – Коммуникатор; out **Status** - Статус передачи;

Длина принятого сообщения должна быть меньше или равна длине буфера приема (чтобы избежать связанных с этим ошибок, необходимо использовать функцию **MPI_Probe**, которая позволяет принимать сообщения с неизвестной длинной, ее синтаксис будет описан далее). В аргумент **source** можно передать так называемое групповое значение **MPI_ANY_SOURCE**, в этом случае функция примет сообщение от любого отправителя. Также можно в аргумент **tag** передать групповое значение **MPI_ANY_TAG**, указывающее, что функция примет сообщение с любой меткой. Для аргумента **comm** нет такого значения. Необходимо заметить асимметрию между операциями отправки и получения: операция получения может принимать сообщения от произвольного отправителя, с другой стороны, отправитель должен указать однозначного получателя. Это согласовано с механизмом “push” коммуникации, где передача данных выполняется отправителем (в отличие от механизма “pull”, где передача выполняется получателем).

Если в операции приема использовать групповые значения, то источник и ярлык (Tag) принятого сообщения в таком случае неизвестны. Также, если выполнять многократные запросы одной MPI-функцией, может понадобиться индивидуальный код ошибки для каждого запроса. Данная информация возвращается в аргументе **status**. Переменная статуса (**MPI_Status**) должна быть создана явно пользователем, так как это не системный объект. Данная переменная

является структурой, которая содержит три поля (также могут быть и дополнительные поля). Таким образом, источник, ярлык и код ошибки соответствующие принятому сообщению содержаться в следующих полях: **status.MPI_SOURCE**, **status.MPI_TAG**, **status.MPI_ERROR**.

Аргумент **status** также возвращает информацию о длине принятого сообщения, но эта информация не является доступной напрямую как компонента структуры. Для «декодирования» данной информации необходим вызов следующей функции:

Int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
Атрибутами этой функции являются:
in **status** - статус операции приема; in **Datatype** - тип данных в буфере; out **count** - количество элементов в буфере;

С помощью данной функции можно получать сообщения, количество элементов в которых первоначально неизвестно. Функцию необходимо использовать после вызова **MPI_Probe**, получив информацией о типе данных, можно использовать вызов **MPI_Recv** с таким же типом для получения требуемого сообщения.

Во многих случаях приложения конструируются так, что для них нет необходимости проверять аргумент **status**. Тогда для пользователя не необходимости создавать переменную статуса, а для MPI заполнять поля этого объекта. Для этого существует константа **MPI_STATUS_IGNORE** (**MPI_STATUSES_IGNORE** для функций с массивами), которая при передаче аргумента **status** информирует систему о том, что данный аргумент не следует заполнять.

Все операции отправки и приема изложенные далее, используют аргументы **buf**, **count**, **datatype**, **source**, **dest**, **tag**, **comm** и **status** таким же образом, как и блокирующие операции **MPI_Send** и **MPI_Recv**. Использование описанных функций проиллюстрировано в следующем примере:

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include "mpi.h"
using namespace std;
int main(int argc, char* argv[])
{
   char message[20];
   int rank;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank == 0)    /* code for process zero */
   {   strcpy(message, "Hello, there");
      MPI_Send(message, strlen(message)+1, MPI_CHAR, 1,99,
```

```
            MPI_COMM_WORLD);
    }
    else
    {  if (rank == 1) /* code for process one */
       {   MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
          cout << "Received message: "'<< message << "'" << endl;
          cin.get();
       }
    }
    MPI_Finalize();
    return 0;
}
```

В этом примере нулевой процесс (**rank=0**) посылает сообщение первому процессу, используя операцию **MPI_Send**. Операция определяет буфер отправки в памяти процесса-отправителя, из которой берутся данные сообщения. Из примера видно, что буфер отправки – переменная **message** памяти нулевого процесса. Местоположение, размер и тип буфера отправки указаны первыми тремя параметрами **send** операции. Операция отправки ассоциирует «конверт» (**envelope**) с сообщением (формирует некоторую адресную информацию для сообщения). Этот конверт указывает пункт назначения сообщения и содержит характерную информацию, которая может быть использована операцией получения, чтобы выбрать именно это сообщение. Последние три параметра операции отправки, вместе с рангом отправителя определяют конверт для отправляемого сообщения. Первый процесс (**rank=1**) получает это сообщение с помощью **receive**-операции **MPI_Recv**. Получаемое сообщение выбирается согласно параметрам его «конверта», данные сообщения сохраняются в буфере приема. В примере буфер состоит из строковой переменной в памяти первого процесса. Первые три параметра операции приема указывают местоположение, размер и тип буфера приема. Следующие три параметра используются для выбора входящего сообщения. Последний параметр используется для того, чтобы получить недостающую информацию о только что принятом сообщении.

Связь без блокировки

При реализации программ можно повысить производительность путем использования **неблокирующей коммуникации**. Вызов неблокирующего старта отправки (**sendstart**) инициирует операцию отправки, но не завершает ее. Возврат из данного вызова может осуществиться до того, как сообщение было скопировано из буфера отправки. Для завершения коммуникации необходим отдельный вызов операции завершения отправки (**sendcomplete**), т.е. подтверждение того, что данные были скопированы из буфера отправки. Таким образом, перенос данных из памяти отправки может продолжаться одновременно с совершаемыми вычислениями в отправителе, после того как была инициализирована отправка и до того как она окончилась.

Подобным образом возврат из вызова не блокирующего старта приема (**receivestartcall**) может осуществиться до того, как сообщение было сохранено в буфере приема. Для того чтобы завершить данную операцию и подтвердить, что данные были сохранены в буфере приема, необходим отдельный вызов завершения приема (**receivecomplete**). Перенос данных в память приемника может продолжаться одновременно с совершаемыми вычислениями, после того как прием был инициирован, и перед тем как он завершился. Использование неблокирующих приемов может также позволить избежать системной буферизации и копирования из памяти в память. Не блокирующий вызов старта отправки использует те же четыре режима, как и отправка с блокировкой: **standard**, **buffered**, **synchronous** и **ready**. Не блокирующая коммуникация использует скрытый объект запроса (**request**) для идентификации коммуникационных операций и сопоставления операций, которые инициирует передачу с операциями, завершающими ее. Доступ к этому объекту можно получить через дескриптор. Объект запроса определяет различные свойства коммуникационных операций, такие как режим отправки, связанный с ним коммуникационный буфер, его контекст, ярлык и аргументы пункта назначения, используемые для отправки, или ярлык и аргументы пункта отправки, используемые для приема. Также этот объект хранит информацию о статусе незавершённых коммуникационных операций. Соглашения по именованию используются такие же, как и для блокирующих операций, но с добавлением префикса **I** (**immediate**), который говорит о том, что операция является не блокирующей. Для того чтобы начать не блокирующую отправку в стандартном режиме используется функция, синтаксис которой следующий:

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

Атрибутами этой функции являются: in buf **- указатель на буфер; in** count **- количество элементов в буфере; in** datatype **- тип данных буфера; in** dest **-**

ранг пункта назначения; in tag **- метка сообщения; in** comm **– коммуникатор; out** request **- коммуникационный запрос.**

Название функций, использующие другие режимы коммуникации, следующие: **MPI_Ibsend**; **MPI_Issend**; **MPI_Irsend**.

Для того чтобы начать неблокирующий прием, используется функция, синтаксис которой приведен ниже:

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

Атрибутами этой функции являются: out buf **- указатель на буфер; in** count **- количество элементов в буфере; in** datatype **- тип данных буфера; in** source **– ранг пункта отправки;in** tag **- метка сообщения; in** comm **– коммуникатор; out** request **- коммуникационный запрос.**

Вызовы данных функций создают коммуникационный объект запроса и ассоциируют его с дескриптором запроса (аргумент **request**). Этот аргумент может быть использован для получения статуса коммуникации или ожидания ее завершения. Вызов не блокирующей отправки говорит о том, что система может начать копировать данные из буфера отправки. Отправитель при этом не может модифицировать буфер отправки после того, как операция вызвана и до того, как она завершится. Вызов не блокирующего приема говорит о том, что система может начать сохранять данные в буфере приема. Получатель при этом не может получить доступ к буферу приема после того как операция вызвана и до того как она завершиться. Для определения завершения неблокирующих коммуникаций используются функции **MPI_Wait** и **MPI_Test**. Завершение операции отправки говорит о том, что отравитель может свободно обновлять значения в буфере отправки (операция отправки оставляет содержимое буфера отправки неизмененным).

Завершение операции приема говорит о том, что буфер приема содержит принятое сообщение, приемник теперь свободно может получать доступ к нему и о том, что установлен статус объекта. Это не значит, что соответствующая операция отправки завершилась (но указывает на то, что отправка была инициирована). Для анализа атрибута **request** используются рассматриваемые ниже функции. Для ожидания завершения не блокирующих операций используется следующая функция:

int MPI_Wait(MPI_Request *request, MPI_Status *status);
Атрибутами этой функции являются: inout request **– запрос; out** status **- статус передачи.**

Возврат из функции **MPI_Wait** происходит, когда операция определенная аргументом **request** завершилась. Если коммуникационный объект, ассоциированный с этим запросом, был создан не блокирующим вызовом

отправки или приема, тогда объект освобождается вызовом **MPI_Wait** и дескриптор запроса устанавливается в значение **MPI_REQUEST_NULL**.

Вызов возвращает в переменную status **информацию о завершённой операции. Также разрешен вызов** MPI_Wait **с нулевым или неактивным аргументом** request**. В таком случае операция завершается незамедлительно с пустым аргументом** status**. Также существует другая функция проверки завершения не блокирующей операции, синтаксис которой приведен ниже:**
Int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
Атрибутами этой функции являются:

Inout request – Запрос; Out flag - Флаг завершения операции; Out status - Статус передачи;

Вызов **MPI_Test** возвращает в переменную **flag** значение **true**, если операция, указанная в аргументе **request** завершена. В этом случае объект статуса установлен таким образом, что содержит информацию о завершенной операции; если объект коммуникации был создан неблокируемой отправкой или приемом, тогда он освобождается, и дескриптор запроса устанавливается в **MPI_REQUEST_NULL**. Вызов возвращает в переменную **flag** значение **false**, в том случае если значение объекта статуса неопределенно. Функция **MPI_Test** является локальной операцией. Коммуникационный объект запроса может быть освобожден без ожидания завершения ассоциированной с ним коммуникации, используя следующую операцию:

int MPI_Request_free(MPI_Request *request);

inout **Request** – запрос;

Данная функция освобождает память для коммуникационного объекта запроса, и присваивает аргументу **request** значение **MPI_REQUEST_NULL**. Продолжающаяся коммуникация, которая ассоциирована с запросом, будет завершена, но запрос будет освобожден только после ее завершения.

Зондирование и отмена

Операции **MPI_Probe** и **MPI_Iprobe** позволяют проверить входящие сообщения без непосредственного их приема. Пользователь может после решить, как принять их, основываясь на информации возвращенной зондированием (в основном информация возвращается в аргументе **status**). В частном случае пользователь может выделить память для буфера приема в соответствии с длиной сообщения зондирования.

Операция **MPI_Cancel** позволяет отменить незавершенные передачи. Эта функция необходима для освобождения ресурсов (памяти), используемых для

обмена данными. Другими словами, реализация отправки или получения требует от пользователя выделения ресурсов системы (например, создать буфер), а отмена при этом необходима, чтобы освободить эти ресурсы в нужный момент. Синтаксис данных функций следующий:

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);

Атрибутами этой функции являются:
in **source** - ранг пункта отправки; in **tag** - метка сообщения; in **comm** – коммуникатор; out **flag** - флаг успешности приема требуемого сообщения; out **status** - статус передачи;

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);

Атрибутами этой функции являются:
in **source** - ранг пункта отправки; in **tag** - метка сообщения; in **comm** – коммуникатор; out **status** - статус передачи;

int MPI_Cancel(MPI_Request *request);

Атрибутом этой функции является:
in **request** - коммуникационный запрос.

Если операция была отменена, тогда информация об этом будет возвращена в аргументе статуса операции, которая должна завершить коммуникацию. Для того что бы ее получить используется следующая функция:

Int MPI_Test_Cancelled (MPI_Status *status, int *flag);

in **status** - статус передачи; out **flag** - флаг состояния;

Функция вернет в переменную **flag** значение **true,**если коммуникация, ассоциированная с объектом статуса, была отменена успешно. В таком случае все другие поля аргумента **status** (**count** или **tag**) являются неопределенными. Для отмены операции приема сначала следует вызвать функцию **MPI_Test_cancelled**, чтобы проверить была ли операция отменена, для того, чтобы использовать остальные поля возвращенного объекта статуса.