ЛАБОРАТОРНАЯ РАБОТА №1

ИСЛЕДОВАНИЕ СРЕДСТВ СОЗДАНИЯ РАСПРЕДЕЛЕННО ВЫПОЛНЯЮЩИХСЯ ПОГРАММ

ЦЕЛЬ РАБОТЫ: исследовать функции библиотеки MPI, необходимые для создания и взаимодействия распределено выполняемых программ.

1. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

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).

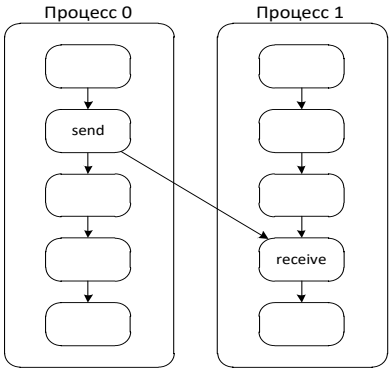

Рисунок 1.1 – Взаимодействие процкссов типа «точка-точка»

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

Типы данных MPI	Типы данных C
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
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.

1.2. Режимы связи

Изложенная выше процедура отправки является блокирующей: выход из функции не происходит, пока сообщение не будет безопасно сохранено на приемной стороне, тогда отправитель свободно может изменять буфер отправки. Сообщение может быть напрямую скопировано в буфер приема или оно может быть скопировано во временную системную память. Отправка с блокировкой завершиться как только сообщение было буферизировано, даже если не была вызвана функция подходящего приема получателем.

С другой стороны буферизация сообщения может быть затратной, так как влечет за собой дополнительное копирование (**memory-to-memory**), что требует выделение памяти для буферизации. MPI предоставляет выбор нескольких коммуникационных режимов, которые позволяют контролировать способ передачи.

Вызов отправки, описанный выше, использует стандартный (**standard**) режим связи. В этом режиме MPI решает, будет ли исходящее сообщение буферизировано. Стандартный режим является **non-local**: успешное завершение операции отправки зависит от событий связанных с соответствующим получением.

Существует три дополнительных режима связи: буферизированный, синхронизированный, по готовности. Синтаксис первого из них следующий:

```
int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

Операция отправки в буферизированном (**buffered**) режиме может быть начата независимо от того была ли создана соответствующая функция приема на стороне получателя. Она может завершиться до того, как на стороне получателя процесс достиг точки выполнения функции приема. В отличие от стандартной отправки эта операция является локальной (**local)**, ее завершение не зависит от событий соответствующего приема. Поэтому если отправка была выполнена, и не была реализована функция приема, MPI должен буферизировать исходящее сообщение для того, чтобы завершить операцию отправки. Ошибка может произойти, только если недостаточно места для буферизации. Количество доступного места для буферизации контролируется пользователем.

Синтаксис функции, обеспечивающей передачу данных во втором режиме, следующий:

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

Отправка, которая использует синхронизированный (**synchronous**) режим может начаться в независимости от того был ли реализован вызов функция приема, но она завершиться успешно только если соответствующая функция приема создана и уже вызвана для получения сообщения синхронной отправки. Поэтому завершение синхронной отправки не только указывает, что буфер отправки может быть снова использован, но также указывает, что приемник достиг перекрестной точки выполнения, другими словами начал выполнять соответствующий прием. Если обе операции отправки и приема являются блокирующими операциями, тогда использование синхронного режима обеспечивает семантику синхронной коммуникации: связь не прекратится, пока оба процесса не «встретятся» во время коммуникации. Отправка, выполняемая в этом режиме, является **non-local** (зависит от взаимодействия процессов). Синтаксис функции, обеспечивающей передачу данных в третьем режиме, следующий:

Int MPI_Rsend(void* buf, intcount, MPI_Datatype datatype, intdest, int tag, MPI_Comm comm);

Отправка, которая использует режим связи по готовности(**ready**), может начаться, только если уже вызвана функция соответствующего приема. В противном случае операция является ошибочной, и ее результат является неопределенным. На некоторых системах это позволяет убрать подтверждение установления связи, что в свою очередь приводит к улучшению производительности. Завершение операции отправки не зависит от статуса соответствующего приема и только указывает, что буфер отправки может быть снова использован. Операция отправки, которая использует режим готовности, имеет туже семантику, что и стандартная отправка или отправка с синхронизацией, только отправитель при этом обеспечивает дополнительной информацией систему о том, что функция соответствующего приема уже создана. Рассмотренная операция приема подходит для любого режима отправки. Эта операция приема является блокирующей, потому что завершение функции происходит только после того как буфер приема будет содержать новое принятое сообщение. Прием может быть завершен до завершения выполнения функции соответствующей отправки. Различные виды буферизации, представлены на Рис.1.2.

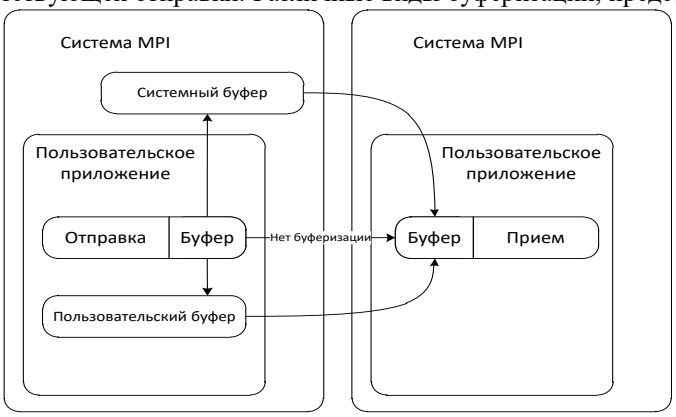

Рисунок 1.2 – Использование буферов при реализации обмена

Рассматриваемая реализация MPI обеспечивает следующие особенности обмена типа «точка-точка»:

1. **Порядок**. Сообщения «не обгоняют» друг друга(**non-overtaking**). Если отправитель отсылает два сообщения последовательно в один и тот же пункт назначения, оба сообщения имеют одинаковые параметры (могут подходить для одной и той же функции приема), то операция приема не может принять второе сообщение, пока не будет принято первое. Так как процесс имеет только один поток выполнения, любые две коммуникации, выполняющиеся этим процессом, упорядочены.
2. **Прогресс.** Если пара соответствующих операций отправки и приема были инициализированы на двух процессах, тогда хотя бы одна из этих двух операций должна завершиться:
- операция отправки будет окончена, если только соответствующая функция приема не получила другое подходящее сообщение и не завершилась;
- операция приема будет завершена, если только отправленное сообщение не было принято другой подходящей функцией приема, которая была создана в том же пункте назначения.
3. **Справедливость.** MPI не гарантирует справедливость в обработке коммуникаций между процессами. Если отправителем была вызвана функция отправки, существует возможность, что в пункте назначения процесс, неоднократно вызывающий функцию приема, подходящую под параметры сообщения, никогда не получит это сообщение, потому что каждый раз будет получать другое сообщение с такими же параметрами, но отправленное другим адресатом.
4. **Ограниченность ресурсов.** Любая незавершенная передача использует ресурсы системы, которые являются ограниченными. Из-за недостатка ресурсов выполнение MPI-функций может привести к ошибке. Программа считается «безопасной», если для завершения программы не требуется ни одна буферизация сообщения.

1.3. Размещение буфера

В буферизированном режиме пользователь может определить буфер, который будет использоваться для отправляемых сообщений. Чтобы обеспечить MPI видимость буфера, используемого для буферизации исходящих сообщений, реализуется вызов функции со следующим синтаксисом:
int MPI_Buffer_attach(void* buffer, int size). **Атрибутами этой функции являются: in** buffer **- указатель на буфер; in** size **- размер буфера в байтах (неотрицательное число);**

Буфер используется только для отправляемых сообщений в буферизированном режиме. Однократно только один буфер может быть выделен процессу. Для того чтобы отсоединить буфер, ассоциированный с MPI-программой для передачи данных в буферизированном режиме, необходимо использовать функцию, синтаксис которой следующий:
int MPI_Buffer_detach(void* buffer_addr, int* size)**. Атрибутами этой функции являются: out** buffer_addr **- указатель на буфер; out** size **- размер буфера в байтах (неотрицательное число).**

Операция будет заблокирована, пока все сообщения из буфера не будут переданы. В зависимости от возвращаемых значений пользователь может снова использовать или освободить место, занимаемое буфером.

1.4. 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.

1.5. 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.

2. ЗАДАНИЕ НА РАБОТУ

Задание выбирается в соответствии с вариантом, назначенным преподавателем. Реализовать при помощи посылки сообщений типа точка-точка следующие схемы коммуникации процессов:

Вариант№1. Программа осуществляет умножение двух матриц. Размеры матриц – 3*3 и 4*4. На каждом процессе, определяет произведение одной строки первой матрицы на все столбцы второй матрицы. Результаты возвращаются в родительскую задачу.

Вариант №2. Программа осуществляет вычисление определителя матрицы 4*4 методом треугольников. Каждый процесс подсчитывает только произведения, определение результата осуществляется в родительской задаче, куда передаются результаты работы процесса. По процессам распределяется вся матрица.

Вариант №3. Вычислительный кластер реализует конвейер по обработке введенных матриц (3*3). Конвейер состоит из трех сегментов (процессов). Первый сегмент конвейера реализует ввод данных. Второй сегмент конвейера определяет миноры матрицы на основе исходной матрицы, третий сегмент, используя матрицу миноров и, вычислив определитель, находит матрицу, обратную данной.

3. КОНТРОЛЬНЫЕ ВОПРОСЫ

3.1. Какими атрибутами обладает в MPI каждое посылаемое сообщение?
3.2. Что гарантирует блокировка при отправке/приеме сообщения?
3.3. Как принимающий процесс может определить длину полученного сообщения?
3.4. Сравнить эффективность реализации различных режимов пересылок данных с блокировкой между двумя выделенными процессами.
3.5. Сравнить эффективность реализации пересылок данных между двумя выделенными процессами с блокировкой и без блокировки.