ЛАБОРАТОРНАЯ РАБОТА №3

Основы использования MPICH, программной реализации интерфейса MPI. Двухточечный неблокирующий обмен сообщениями.

1 ЦЕЛЬ РАБОТЫ

1.1 Освоить использование типовых функций MPICH для неблокирующего обмена сообщениями.

1.2 Освоить использование функций завершения операций передачи и приема MPI-сообщений для одного, нескольких либо всех обменов. Получить представление о механизме «отложенных обменов».

1.3 Попрактиковаться в написании MPICH-приложений.

2 КЛЮЧЕВЫЕ ПОЛОЖЕНИЯ

В данной работе мы рассмотрим неблокирующие операции обмена сообщениями. Их целесообразно использовать прежде всего для избежания простоев во время выполнения обмена. Вызов функции неблокирующей передачи инициирует, но не завершает ее. Завершиться выполнение функции может еще до того, как сообщение будет скопировано в буфер передачи.

Применение неблокирующих операций улучшает производительность программы, поскольку в этом случае допускается перекрытие (т. е. одновременное выполнение) вычислений и обменов. Передача данных из буфера или их считывание может происходить одновременно с выполнением процессом другой работы. Для завершения обмена требуется вызов дополнительной функции, которая проверяет, скопированы ли данные в буфер передачи.

Несмотря на то, что при неблокирующем обмене возвращение из функции обмена происходит сразу, запись в буфер или считывание из него после этого производить нельзя, ведь сообщение может быть еще не отправлено или не получено, и работа с буфером может повредить его содержимое. Таким образом, неблокирующий обмен выполняется в два этапа:

- инициализация обмена;
- проверка завершения обмена.

Разделение этих шагов делает необходимой маркировку каждой операции обмена, что позволяет целенаправленно выполнять проверки завершения соответствующих операций. Для маркировки в неблокирующих операциях используются идентификаторы операций обмена (requests).

Неблокирующая передача может выполняться в тех же четырех режимах, что и блокирующая: стандартном, буферизованном, синхронном и «по готовности». Передача в каждом из них может быть начата независимо от того, был ли зарегистрирован соответствующий прием. Во всех случаях, операция начала передачи локальна, она завершается сразу же и независимо от состояния других процессов.

Инициализация неблокирующей стандартной передачи выполняется с помощью функции MPI_Isend( ):

```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, \
int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

Ее входные параметры аналогичны аргументам подпрограммы MPI_Send( ), а выходным является параметр request — идентификатор операции.

Функция MPI_Issend( ) инициализирует неблокирующую синхронную передачу данных:

```
int MPI_Issend(const void *buf, int count, MPI_Datatype datatype, \
int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

Параметры функции MPI_Issend( ) совпадают с параметрами функции MPI_Isend( ).

Неблокирующую буферизованную передачу сообщения выполняет функция MPI_Ibsend( ):

```
int MPI_Ibsend(const void *buf, int count, MPI_Datatype datatype, \
int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

Неблокирующая передача "по готовности" выполняется функцией MPI_Irsend( ):

```
int MPI_Irsend(const void *buf, int count, MPI_Datatype datatype, \
int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

Параметры всех функций неблокирующей передачи совпадают.

Функция MPI_Irecv( ) начинает неблокирующий прием:

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

Назначение аргументов здесь такое же, как и в предыдущих функциях, за исключением того, что указывается ранг не адресата, а источника сообщения (source).

Вызовы функций неблокирующего обмена формируют запрос на выполнение операции обмена, и связывают его с идентификатором операции request. Запрос устанавливает различные свойства операции обмена, такие как режим, характеристики буфера обмена, контекст, тег и ранг, используемые при обмене. Кроме того, запрос содержит информацию о состоянии ожидающих обработки операций обмена, и может быть использован для получения информации о состоянии обмена или для ожидания его завершения, о чем мы поговорим далее.

Вызов функции неблокирующей передачи означает, что система может начинать копирование данных из буфера. Источник не должен обращаться к буферу передачи во время выполнения неблокирующей передачи. Вызов неблокирующего приема означает, что система может начинать запись данных в буфер приема. Источник во время передачи и адресат во время приема не должны обращаться к буферу. Неблокирующая передача может быть принята функцией блокирующего приема и наоборот.

Неблокирующий обмен можно использовать не только для увеличения скорости выполнения программы, но и в тех ситуациях, где блокирующий обмен может привести к взаимоблокировке. Избежать в этом случае «тупика» можно, используя и функцию блокирующего обмена MPI_Sendrecv( ), но блокирующий обмен замедляет выполнение программы, поскольку его скорость определяется скоростью работы коммуникационной сети, а эта скорость, как правило, не очень высока.

И немного о терминах. Пустым называют запрос с идентификатором MPI_REQUEST_NULL. Отложенный запрос на выполнение операции обмена называют неактивным, если он не связан ни с каким исходящим сообщением. Пустым называют статус, поле тега которого принимает значение MPI_ANY_TAG, поле источника сообщения - MPI_ANY_SOURCE, а вызовы функций MPI_Get_count( ) и MPI_Get_elements( ) возвращают нулевое значение аргумента count.

Проверка фактического выполнения передачи или приема в неблокирующем режиме осуществляется с помощью вызова одной из функций ожидания, блокирующих работу процесса до завершения операции, или одной из неблокирующих функций проверки, возвращающих логическое значение «истина», если операция выполнена.

Функция MPI_Wait( ) блокирует работу процесса до завершения приема или передачи сообщения:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Входной параметр request — идентификатор операции обмена, а выходной — статус (status). В аргументе status возвращается информация о выполненной операции. Значение статуса для операции передачи сообщения можно получить вызовом функции MPI_Test_cancelled( ). Можно вызвать MPI_Wait( ) с пустым или неактивным аргументом request. В этом случае операция завершается сразу же с пустым статусом.

Успешное выполнение подпрограммы MPI_Wait( ) после вызова MPI_Ibsend( ) подразумевает, что буфер передачи можно использовать вновь, т. е. пересылаемые данные

отправлены или скопированы в буфер, выделенный при вызове функции MPI_Buffer_attach( ). В этот момент уже нельзя отменить передачу. Если не будет зарегистрирован соответствующий прием, буфер нельзя будет освободить. В этом случае можно применить функцию MPI_Cancel( ), которая освобождает память, выделенную подсистеме коммуникаций.

Функция MPI_Test( ) выполняет неблокирующую проверку завершения приема или передачи сообщения:

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

Ее входным параметром является идентификатор операции обмена request. Выходные параметры: flag - «истина», если операция, заданная идентификатором request, выполнена, и status — статус выполненной операции. Если при вызове MPI_Test( ) используется пустой или неактивный аргумент request, операция возвращает значение флага "истина" и пустой статус. Функции MPI_Wait( ) и MPI_Test( ) можно использовать для завершения операций приема и передачи.

Механизм работы двухточечного неблокирующего обмена демонстрируется на примере использования функций MPI_Irsend( ) и MPI_Wait( ). Надо заметить, что это лишь «каркас» приложения, без инициализации массива buffer2, вывода на печать результатов обмена и пр. Вы можете самостоятельно дописать недостающие фрагменты кода, и, запустив его на выполнение, проверить работоспособность.

Листинг 1 — Использование функций MPI_Irsend ( ) и MPI_Wait( ).

(пример кода из http://mpi.deino.net)

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
int myid, numprocs, left, right;
int buffer[10], buffer2[10];
MPI_Request request, request2;
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
right = (myid + 1) % numprocs;
left = myid - 1;
if (left < 0)
left = numprocs - 1;
MPI_Irecv(buffer, 10, MPI_INT, left, 123, MPI_COMM_WORLD, &request);
MPI_Barrier(MPI_COMM_WORLD); /* Ready sends require that the receive
buffer be ready before the send call initiates so use a barrier to
ensure this is true */
MPI_Irsend(buffer2, 10, MPI_INT, right, 123, MPI_COMM_WORLD,\
&request2);
MPI_Wait(&request, &status);
MPI_Wait(&request2, &status);
MPI_Finalize();
return 0;
}
```

Функции MPI_Wait( ) и MPI_Test( ) предназначены для контроля выполнения одного действия. В том случае, когда сразу несколько процессов обмениваются сообщениями, можно использовать проверки, которые применяются одновременно к нескольким обменам. Есть три типа таких проверок:

- проверка завершения всех обменов;

- проверка завершения любого обмена из нескольких;

- проверка завершения заданного обмена из нескольких.

Каждая из этих проверок, в свою очередь, имеет две разновидности — «ожидание» и собственно «проверка».

Проверка завершения всех обменов выполняется функцией MPI_Waitall( ), которая блокирует выполнение процесса до тех пор, пока все операции обмена, связанные с активными запросами в массиве requests, не будут выполнены, и возвращает статус этих операций. Статус обменов содержится в массиве statuses:

```
int MPI_Waitall(int count, MPI_Request array_of_requests[],\
MPI_Status array_of_statuses[])
```

Здесь count — количество запросов на обмен (размер массивов requests и statuses).

В результате выполнения подпрограммы MPI_Waitall( ) запросы, сформированные неблокирующими операциями обмена, аннулируются, а соответствующим элементам массива присваивается значение MPI_REQUEST_NULL. Список может содержать пустые или неактивные запросы. Для каждого из них устанавливается пустое значение статуса.

В случае неудачного выполнения одной или более операций обмена функция MPI_Waitall( ) возвращает код ошибки MPI_ERR_IN_STATUS, и присваивает полю ошибки статуса значение кода ошибки соответствующей операции. Если операция выполнена успешно, полю присваивается значение MPI_SUCCESS, а если не выполнена, но и не было ошибки — значение MPI_ERR_PENDING. Последний случай соответствует наличию запросов на выполнение операций обмена, ожидающих обработки.

Неблокирующая проверка выполняется с помощью вызова функции MPI_Testall( ). Она возвращает значение флага (flag) «истина», если все обмены, связанные с активными запросами в массиве requests, выполнены. Если завершены не все обмены, флагу присваивается значение «ложь», а массив statuses не определен:

```
int MPI_Testall(int count, MPI_Request array_of_requests[], \
int *flag, MPI_Status array_of_statuses[])
```

Здесь count — количество запросов.

Каждому статусу, соответствующему активному запросу, присваивается значение статуса соответствующего обмена. Если запрос был сформирован операцией неблокирующего обмена, он аннулируется, а его элементу массива присваивается значение MPI_REQUEST_NULL. Каждому статусу, соответствующему пустому или неактивному запросу, присваивается пустое значение.

Проверки завершения любого числа обменов выполняются с помощью функций MPI_Waitany( ) и MPI_Testany( ). Первый из этих вариантов — блокирующий. Выполнение процесса блокируется до тех пор, пока, по крайней мере, один обмен из массива запросов (requests) не будет завершен. Индекс соответствующего элемента в массиве requests возвращается в аргументе index, а статус — в аргументе status:

```
int MPI_Waitany(int count, MPI_Request array_of_requests[], \
int *index, MPI_Status *status)
```

Входные параметры: requests и count — количество элементов в массиве requests, а выходные: status и index — индекс запроса (целое число от 0 до count — 1).

Если запрос на выполнение операции был сформирован неблокирующей операцией обмена, он аннулируется и ему присваивается значение MPI_REQUEST_NULL. Массив запросов может содержать пустые или неактивные запросы. Если в списке вообще нет активных запросов или он пуст, вызовы завершаются сразу со значением индекса MPI_UNDEFINED и пустым статусом.

Функция MPI_Testany( ) проверяет выполнение любого ранее инициализированного обмена:

```
int MPI_Testany(int count, MPI_Request array_of_requests[], \
int *index, int *flag, MPI_Status *status)
```

Смысл и назначение параметров этой функции те же, что и для функции MPI_Waitany( ), но есть дополнительный аргумент flag, который принимает значение «истина», если одна из операций

завершена. Блокирующая функция MPI_Waitany( ) и неблокирующая MPI_Testany( ) взаимозаменяемы, впрочем, как и другие аналогичные пары.

Функции MPI_Waitsome( ) и MPI_Testsome( ) действуют аналогично функциям MPI_Waitany( ) и MPI_Testany( ), кроме случая, когда завершается более одного обмена. В функциях MPI_Waitany( ) и MPI_Testany( ) обмен из числа завершенных выбирается произвольно, именно для него и возвращается статус, а для MPI_Waitsome( ) и MPI_Testsome( ) статус возвращается для всех завершенных обменов. Эти функции можно использовать для определения, сколько обменов завершено:

```
int MPI_Waitsome(int incount, MPI_Request array_of_requests[],\
int *outcount, int array_of_indices[],MPI_Status array_of_statuses[])
```

Здесь incount — количество запросов. В outcount возвращается количество выполненных запросов из массива requests, а в первых outcount элементах массива indices возвращаются индексы этих операций. В первых outcount элементах массива statuses возвращается статус завершенных операций. Если выполненный запрос был сформирован неблокирующей операцией обмена, он аннулируется. Если в списке нет активных запросов, выполнение подпрограммы завершается сразу, а параметру outcount присваивается значение MPI_UNDEFINED.

Функция MPI_Testsome( ) — это неблокирующая проверка:

```
int MPI_Testsome(int incount, MPI_Request array_of_requests[], \
int *outcount, int array_of_indices[],MPI_Status array_of_statuses[])
```

У нее такие же параметры, как и у функции MPI_Waitsome( ). Эффективность функции MPI_Testsome( ) выше, чем у MPI_Testany( ), поскольку первая возвращает информацию обо всех операциях, а для второй требуется новый вызов для каждой выполненной операции.

Использование функции MPI_Testsome( ) демонстрируется в листинге 2. Как и в предыдущем примере, приводится только каркас приложения. Студентам предлагается дописать недостающие участки кода, и, запустив его на выполнение, проверить работоспособность. Пример предполагает запуск четырех процессов.

Листинг 2 — Пример использования функции MPI_Testsome ()
(пример кода из http://mpi.deino.net)

```
#include "mpi.h"
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
int rank, size, flag, i, count;
int buffer[100];
MPI_Request r[4];
MPI_Status status[4];
int index[4];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (size != 4)
{
printf("Please run with 4 processes.\n");
fflush(stdout);
MPI_Finalize();
return 1;
}
if (rank == 0)
{
int remaining = 3;
for (i=1; i<size; i++)
```

```
{
MPI_Irecv(&buffer[i], 1, MPI_INT, i, 123, MPI_COMM_WORLD, &r[i-1]);
}
while (remaining > 0)
{
MPI_Testsome(size-1, r, &count, index, status);
if (count > 0)
{
printf("%d finished\n", count);
remaining = count;
}
else
{
sleep(1);
}}}
else
{
MPI_Send(buffer, 1, MPI_INT, 0, 123, MPI_COMM_WORLD);
}
MPI_Finalize();
return 0;
}
```

Достаточно часто приходится сталкиваться с ситуацией, когда обмены с одинаковыми параметрами выполняются повторно, например, в цикле. В этом случае можно объединить аргументы функций обмена в один отложенный запрос, который затем повторно используется для инициализации и выполнения обмена сообщениями. Отложенный запрос на выполнение неблокирующей операции обмена позволяет минимизировать накладные расходы на организацию связи между процессором и контроллером связи. Отложенные запросы на обмен объединяют такие сведения об операциях обмена, как адрес буфера, количество пересылаемых элементов данных, их тип, ранг адресата, тег сообщения и коммуникатор.

Для создания отложенного запроса на обмен используются четыре функции. Сам обмен они не выполняют. Запрос для стандартной передачи создается при вызове функции MPI_Send_init( ):

```
int MPI_Send_init(const void *buf, int count, MPI_Datatype datatype,\
int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

Здесь buf — адрес буфера передачи; count — количество элементов; datatype — тип элементов; dest — ранг адресата; tag — тег сообщения; comm — коммуникатор. Выходным параметром является запрос на выполнение операции обмена (request).

Отложенный запрос может быть сформирован для всех режимов обмена. Для этого используются функции MPI_Bsend_init( ), MPI_Ssend_init( ) и MPI_Rsend_init( ). Список аргументов у них совпадает со списком аргументов функции MPI_Send_init( ).

Отложенный обмен инициализируется последующим вызовом функции MPI_Start( ):

```
int MPI_Start(MPI_Request *request)
```

Входным параметром этой функции является запрос на выполнение операции обмена (request). Вызов MPI_Start( ) с запросом на обмен, созданным MPI_Send_init( ), инициирует обмен с теми же свойствами, что и вызов функции MPI_Isend( ), а вызов MPI_Start( ) с запросом, созданным MPI_Bsend_init( ), инициирует обмен аналогично вызову MPI_Ibsend( ). Сообщение, которое передано операцией, инициированной с помощью MPI_Start( ), может быть принято любой функцией приема.

Функция MPI_Startall( ):

```
int MPI_Startall(int count, MPI_Request array_of_requests[])
```

инициирует все обмены, связанные с запросами на выполнение неблокирующей операции обмена в массиве requests. Завершается обмен при вызове MPI_Wait( ), MPI_Test( ) и некоторых других

функций.

Операция MPI_Cancel( ) позволяет аннулировать неблокирующие «ждущие» (т.е. ожидающие обработки) обмены. Ее можно использовать для освобождения ресурсов, прежде всего памяти, отведенной для размещения буферов:

```
int MPI_Cancel(MPI_Request *request)
```

Вызов этой функции завершается сразу, возможно, еще до реального аннулирования обмена. Аннулируемый обмен следует завершить. Это делается с помощью подпрограмм MPI_Request_free( ), MPI_Wait( ), MPI_Test( ) и некоторых других.

Функцию MPI_Cancel( ) можно использовать для аннулирования обменов, использующих как отложенный, так и обычный запрос. После вызова MPI_Cancel( ) и следующего за ним вызова MPI_Wait( ) или MPI_Test( ) запрос на выполнение операции обмена становится неактивным и может быть активизирован для нового обмена. Информация об аннулированной операции содержится в аргументе status.

Функция MPI_Test_cancelled( ) возвращает значение флага (flag) «истина», если обмен, связанный с указанным статусом, успешно аннулирован:

```
int MPI_Test_cancelled(const MPI_Status *status, int *flag)
```

Аннулирование — трудоемкая операция, не стоит ей злоупотреблять. Запрос на выполнение операции (request) можно аннулировать с помощью подпрограммы MPI_Request_free( ):

```
int MPI_Request_free(MPI_Request *request)
```

При вызове эта подпрограмма помечает запрос на обмен для удаления и присваивает ему значение MPI_REQUEST_NULL. Операции обмена, связанной с этим запросом, дается возможность завершиться, а сам запрос удаляется только после завершения обмена. Следует также учитывать, что после того как запрос аннулирован, нельзя проверить, был ли успешно завершен соответствующий обмен, поэтому активный запрос нельзя аннулировать.

Использование отложенных запросов продемонстрируем в листинге 3. Этот код интересен и сам по себе, поскольку реализует один из вариантов измерения длительности операций обмена. Изучите его во всех подробностях, и кратко сформулируйте для себя смысл выполняемой последовательности действий.

Листинг 3 — Пример использования отложенных запросов
(пример кода из https://computing.llnl.gov)

```
/*****************************************************************
* AUTHOR:  01/09/99 Blaise Barney
*****************************************************************/
#include "mpi.h"
#include <stdio.h>

/* Modify these to change timing scenario */
#define TRIALS          10
#define STEPS           20
#define MAX_MSGSIZE     1048576    /* 2^STEPS */
#define REPS            1000
#define MAXPOINTS       10000

int    numtasks, rank, tag=999, n, i, j, k, this, msgsizes[MAXPOINTS];
double  mbytes, tbytes, results[MAXPOINTS], ttime, t1, t2;
char   sbuff[MAX_MSGSIZE], rbuff[MAX_MSGSIZE];
MPI_Status stats[2];
MPI_Request reqs[2];

int main(int argc, char *argv[])
{
```

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

/************************** task 0 *********************************/
if (rank == 0) {

/* Initializations */
n=1;
for (i=0; i<=STEPS; i++) {
msgsizes[i] = n;
results[i] = 0.0;
n=n*2;
}
for (i=0; i<MAX_MSGSIZE; i++)
sbuff[i] = 'x';

/* Greetings */
printf("\n****** Persistent Communications *****\n");
printf("Trials=      %8d\n",TRIALS);
printf("Reps/trial=  %8d\n",REPS);
printf("Message Size   Bandwidth (bytes/sec)\n");

/* Begin timings */
for (k=0; k<TRIALS; k++) {

n=1;
for (j=0; j<=STEPS; j++) {

/* Setup persistent requests for both the send and receive */
MPI_Recv_init (&rbuff, n, MPI_CHAR, 1, tag, MPI_COMM_WORLD, &reqs[0]);
MPI_Send_init (&sbuff, n, MPI_CHAR, 1, tag, MPI_COMM_WORLD, &reqs[1]);

t1 = MPI_Wtime();
for (i=1; i<=REPS; i++){
MPI_Startall (2, reqs);
MPI_Waitall (2, reqs, stats);
}
t2 = MPI_Wtime();

/* Compute bandwidth and save best result over all TRIALS */
ttime = t2 - t1;
tbytes = sizeof(char) * n * 2.0 * (float)REPS;
mbytes = tbytes/ttime;
if (results[j] < mbytes)
results[j] = mbytes;

/* Free persistent requests */
MPI_Request_free (&reqs[0]);
MPI_Request_free (&reqs[1]);
n=n*2;
}   /* end j loop */
}     /* end k loop */

/* Print results */
for (j=0; j<=STEPS; j++) {
```

```
printf("%9d %16d\n", msgsizes[j], (int)results[j]);
}
}       /* end of task 0 */

/**************************  task 1 ********************************/
if (rank == 1) {

/* Begin timing tests */
for (k=0; k<TRIALS; k++) {

n=1;
for (j=0; j<=STEPS; j++) {

/* Setup persistent requests for both the send and receive */
MPI_Recv_init (&rbuff, n, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &reqs[0]);
MPI_Send_init (&sbuff, n, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &reqs[1]);

for (i=1; i<=REPS; i++){
MPI_Startall (2, reqs);
MPI_Waitall (2, reqs, stats);
}

/* Free persistent requests */
MPI_Request_free (&reqs[0]);
MPI_Request_free (&reqs[1]);
n=n*2;

}   /* end j loop */
}     /* end k loop */
}       /* end task 1 */

MPI_Finalize();
}  /* end of main */
```

3 КОНТРОЛЬНЫЕ ВОПРОСЫ

3.1 Поясните особенности выполнения функций двухточечного неблокирующего обмена, и преимущества от их использования.
3.2 Назовите действия (этапы), которые должны быть реализованы в программном коде при использовании MPI-функций двухточечного неблокирующего обмена сообщениями.
3.3 Опишите способ работы и особенности использования MPI-функции неблокирующей стандартной передачи.
3.4 Опишите способ работы и особенности использования MPI-функции неблокирующей синхронной передачи.
3.5 Опишите способ работы и особенности использования MPI-функции неблокирующей буферизованной передачи.
3.6 Опишите способ работы и особенности использования MPI-функции неблокирующей передачи «по готовности».
3.7 Опишите способ работы и особенности использования MPI-функции неблокирующего приема.
3.8 Перечислите функции MPI, используемые для блокирующей проверки завершения операции неблокирующего обмена. Поясните особенности каждой функции, и отличия между ними.
3.9 Перечислите функции MPI, используемые для неблокирующей проверки завершения операции неблокирующего обмена. Поясните особенности каждой функции, и отличия между ними.

3.10 Поясните сущность механизма отложенных запросов на выполнение операций неблокирующего обмена, и преимущества от его использования. Какие MPI-функции реализуют отложенный неблокирующий обмен?

4 ДОМАШНЕЕ ЗАДАНИЕ

4.1 Изучить ключевые положения.
4.2 Письменно ответить на контрольные вопросы.
4.3 Подготовиться к выполнению лабораторного задания.

5 ЛАБОРАТОРНОЕ ЗАВДАНИЕ

5.1 Напишите распределенную программу на C, используя функции неблокирующего двухточечного обмена библиотеки MPICH, описываемые в ключевых положениях. Предполагается, что в результате запуска такого приложения создается произвольное количество процессов (ключ -n), но не меньше трех. Создаваемые процессы моделируют объединение в кольцевую топологию, с последующим двусторонним обменом данными по «кольцу». Тип и количество элементов данных выбрать самостоятельно. На терминал выводить значение ранга текущего процесса, а также номера соседних процессов, с которыми он обменивается данными.
5.2 Перепишите программу предыдущего пункта задания, использовав функции библиотеки MPICH, реализующие отложенную обработку запросов неблокирующего обмена.

6. ТРЕБОВАНИЯ К СОДЕРЖАНИЮ ПРОТОКОЛА

6.1 Название лабораторной работы.
6.2 Цель работы.
6.3 Результаты выполнения домашнего задания.
6.4 Краткое описание проделанной работы.
6.5 Выводы о проделанной работе.
6.6 Дата, подпись студента, виза преподавателя.

Составлено по материалам:

1. *Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. - СПб.:БХВ-Петербург, 2002.*
2. *Мирошников А.С., Гречаный С.В. Методические указания к лабораторным работам по дисциплине «Параллельная обработка данных». - Владикавказ, 2013.*
3. *http://mpi.deino.net*