М. Э. Абрамян

Библиотека MPI 1.1
и электронный задачник
Programming Taskbook for MPI

Общее описание

© М. Э. Абрамян, 2011

1. Основные типы, константы и функции библиотеки MPI

Приведенное в данном разделе описание библиотеки MPI для языка C соответствует стандарту MPI-1.1 [1]. Стандарта MPI для языка Паскаль не существует; в настоящем пособии описывается реализация библиотеки MPI для Паскаля в виде модуля MPI.pas, выполненная автором и включенная в электронный задачник Programming Taskbook for MPI. Все функции, описанные в модуле MPI.pas, импортируются из динамической библиотеки mpich.dll, входящей в комплекс MPICH для Windows.

В данном разделе описываются 102 функции MPI (из примерно 130 функций, входящих в стандарт MPI-1.1) и связанные с ними типы и константы. Не описаны функции, относящиеся, в основном, к следующим разделам стандарта MPI-1.1:

- работа с интеркоммуникаторами (см. [1, п. 5.6]);
- настройка атрибутов для коммуникаторов (см. [1, п. 5.7]);
- обработка ошибок (см. [1, п. 7.2–7.3]).

Для каждой функции вначале приводятся ее заголовки на языках C и Паскаль, затем описываются ее параметры, после чего дается краткое описание самой функции. Возвращаемые значения функций не описываются, так как любая функция MPI (кроме MPI_Wtime и MPI_Wtick) возвращает информацию об успешности своего выполнения (в частности, при успешном завершении функция возвращает значение MPI_SUCCESS).

При описании параметров функций используются следующие обозначения: *IN* — параметр является входным, *OUT* — параметр является выходным, *INOUT* — параметр является входным и выходным. В функциях, связанных с коллективным взаимодействием процессов, обозначение *INOUT* может означать, что для некоторых процессов данный параметр используется как входной, а для других — как выходной.

Быстро найти описание требуемой функции можно по алфавитному указателю функций MPI, приведенному в конце пособия.

1.1. Типы и константы библиотеки MPI

```
MPI_Datatype (тип)
```

Определяет тип пересылаемых данных. Переменные этого типа являются дескрипторами, связанными либо со стандартными типами данных, включенными в библиотеку MPI, либо с пользовательскими типами, определенными с помощью соответствующих функций MPI (см. п. 1.7). Ниже перечислены константы, связанные со стандартными типами данных MPI (в скобках указан соответствующий тип языка C):

MPI_SHORT (signed short int);
MPI_INT (signed int);
MPI_LONG (signed long int);
MPI_UNSIGNED_SHORT (unsigned short int);
MPI_UNSIGNED (unsigned int);
MPI_UNSIGNED_LONG (unsigned long int);
MPI_FLOAT (float);
MPI_DOUBLE (double);
MPI_LONG_DOUBLE (long double);
MPI_CHAR (signed char);
MPI_UNSIGNED_CHAR (unsigned char);

MPI_DATATYPE_NULL — данная константа записывается в дескриптор пользовательского типа при уничтожении этого типа функцией MPI_Type_free;

MPI_PACK — тип, используемый при посылки и приеме упакованных данных (см. п. 1.7).

MPI_Request (тип)

Определяет идентификаторы приема или передачи («запросы обмена»), которые используются при неблокирующих взаимодействиях (см. п. 1.4–1.5). Переменные этого типа являются дескрипторами. С данным типом связана единственная константа MPI_REQUEST_NULL, обозначающая пустой запрос. Переменная типа MPI_Request принимает значение MPI_REQUEST_NULL в результате обработки связанного с ней запроса или его удаления функцией MPI_Request_free.

MPI_Comm (тип)

Определяет коммуникаторы, используемые при пересылке данных. Переменные этого типа являются дескрипторами, связанными либо со стандартными коммуникаторами, либо с пользовательскими коммуникаторами, определенными с помощью соответствующих функций MPI (см. п. 1.8). Ниже перечислены константы, связанные со стандартными коммуникаторами:

MPI_COMM_WORLD — коммуникатор, включающий все процессы параллельного приложения;

MPI_COMM_SELF — коммуникатор, включающий только текущий процесс;

MPI_COMM_NULL — значение, используемое для указания ошибочного коммуникатора.

При *определении типа виртуальной топологии* для данного коммуникатора (с помощью функции MPI_Topo_test) результат возвращается в виде одной из следующих целочисленных констант:

MPI_CART — с данным коммуникатором связана декартова топология;

MPI_GRAPH — с данным коммуникатором связана топология графа;

MPI_UNDEFINED — с данным коммуникатором не связана никакая виртуальная топология.

При *сравнении коммуникаторов* функцией MPI_Comm_compare результат возвращается в виде одной из следующих целочисленных констант:

MPI_IDENT — два дескриптора указывают на один и тот же коммуникатор;

MPI_CONGRUENT — два различных коммуникатора содержат одинаковые наборы одинаково упорядоченных процессов;

MPI_SIMILAR — два коммуникатора содержат одинаковые наборы процессов, однако порядок расположения процессов в них различается;

MPI_UNEQUAL — два коммуникатора содержат различные наборы процессов.

MPI_Group (тип)

Определяет группы процессов, используемые при пересылке данных. Переменные этого типа являются дескрипторами, связанными либо со стандартными группами, либо с пользовательскими группами, определенными с помощью соответствующих функций MPI (см. п. 1.8). Ниже перечислены константы, связанные со стандартными группами:

MPI_GROUP_EMPTY — группа, не содержащая элементов;

MPI_GROUP_NULL — значение, используемое для указания ошибочной группы.

При *сравнении групп* функцией MPI_Group_compare результат возвращается в виде одной из следующих целочисленных констант:

MPI_IDENT — две группы содержат одинаковые наборы процессов, причем эти наборы одинаково упорядочены;

MPI_SIMILAR — две группы содержат одинаковые наборы процессов, однако порядок расположения процессов в них различается;

MPI_UNEQUAL — две группы содержат различные наборы процессов.

MPI_Op (тип)

Определяет операцию («операцию редукции»), используемую при выполнении коллективной пересылки данных. Переменные этого типа являются дескрипторами, связанными либо со стандартными операциями, либо с пользовательскими операциями, определенными с помощью соответствующих функций MPI (см. п. 1.6). Ниже перечислены константы, связанные со стандартными операциями редукции:

MPI_MAX — операция нахождения максимального значения;

MPI_MIN — операция нахождения минимального значения;

MPI_SUM — операция вычисления суммы;

MPI_PROD — операция вычисления произведения;

MPI_LAND — операция «логическое И»;

MPI_BAND — операция «побитовое И»;

MPI_LOR — операция «логическое ИЛИ»;
MPI_BOR — операция «побитовое ИЛИ»;
MPI_LXOR — операция «логическое исключающее ИЛИ»;
MPI_BXOR — операция «побитовое исключающее ИЛИ»;
MPI_MAXLOC — операция нахождения максимума и его индекса;
MPI_MINLOC — операция нахождения минимума и его индекса.

MPI_Aint (тип)

Тип, предназначенный для хранения адресов, а также смещений между различными адресами в памяти. Реализован как знаковый целочисленный тип, размер которого является достаточным для хранения любого адреса.

MPI_Status (тип)

Структурный тип, предназначенный для хранения дополнительной информации, связанной с уже принятым или ожидающим принятия сообщением. Все поля типа MPI_Status являются целочисленными.

С помощью непосредственного обращения к полям данного типа можно определить ранг процесса, пославшего сообщение (поле MPI_SOURCE), идентификатор («метку») сообщения (поле MPI_TAG) и код ошибки, связанной с данным сообщением (поле MPI_ERROR). Кроме того, тип MPI_Status содержит дополнительное поле (в реализации MPICH имеющее имя count), позволяющее определить количество элементов в сообщении. Вместо непосредственного обращения к данному полю следует использовать функцию MPI_Get_count.

С посылкой и приемом сообщений связаны также следующие константы целого типа:

MPI_PROC_NULL — используется для посылки сообщения несуществующему процессу;

MPI_ANY_SOURCE — используется для приема сообщения от любого процесса;

MPI_ANY_TAG — используется для приема сообщения с любым идентификатором;

MPI_TAG_UB — позволяет определить максимально допустимое значение для идентификатора сообщения;

MPI_BSEND_OVERHEAD — позволяет определить размер служебной части буфера, используемого при передаче буферизованных сообщений.

MPI_User_function (тип)

Прототип функции в C/C++ (процедурный тип в Паскале), используемый при определении новой операции с помощью функции MPI_Op_create. Определен следующим образом:

```
typedef void MPI_User_function(void* invec, void* inoutvec,
  int* len, MPI_Datatype* datatype);
type MPI_User_function = procedure(invec: pointer; inoutvec:
```

```
    pointer; var len: longint; var datatype: MPI_Datatype); cdecl;
```

Параметры invec и inoutvec являются указателями на массивы, содержащие len элементов типа datatype. Элементы массивов invec[i] и inoutvec[i], i = 0, …, len–1, считаются соответственно левым и правым операндом определяемой пользовательской операции; результат применения этой операции к элементам invec[i] и inoutvec[i] должен сохраняться в элементе inoutvec[i]. Таким образом, массив invec является входным, а массив inoutvec — одновременно входным и выходным (что и объясняет выбор их имен).

Если пользовательскую операцию предполагается применять только к данным фиксированного типа, то при определении пользовательской операции можно считать, что массивы invec и inoutvec имеют требуемый тип, и не анализировать параметр datatype.

1.2. Функции общего назначения

Функции MPI_Comm_size и MPI_Comm_rank, описанные в данном пункте, относятся, строго говоря, к категории функций, связанных с группами процессов и коммуникаторами (см. п. 1.8). Тем не менее их можно считать и функциями общего назначения, поскольку вызов данных функций (для коммуникатора MPI_COMM_WORLD) обычно выполняется в любом процессе любой содержательной параллельной программы.

```
int MPI_Init(int* argc, char*** argv);
function MPI_Init(var argc: longint; var argv: PPChar): longint;
```

INOUT argc – количество параметров командной строки;

INOUT argv – строковый массив параметров командной строки.

Инициализирует параллельную часть приложения. Параметры передаются по ссылке, поскольку стандарт MPI предусматривает возможность такой реализации данной функции, при которой передача параметров осуществляется не из параллельной программы в среду MPI, а наоборот: из среды MPI в параллельную программу.

```
int MPI_Finalize();
function MPI_Finalize: longint;
```

Завершает параллельную часть приложения.

```
int MPI_Comm_size(MPI_Comm comm, int* size);
function MPI_Comm_size(comm: MPI_Comm; var size: longint):
    longint;
```

IN comm – коммуникатор;

OUT size – число процессов в коммуникаторе.

Определяет общее число параллельных процессов, связанных с указанным коммуникатором.

```
int MPI_Comm_rank(MPI_Comm comm, int* rank);
function MPI_Comm_rank(comm: MPI_Comm; var rank: longint):
     longint;
```

IN comm – коммуникатор;

OUT rank – ранг текущего процесса в коммуникаторе.

Определяет ранг текущего (т. е. вызвавшего данную функцию) процесса в указанном коммуникаторе. Если текущий процесс не входит в указанный коммуникатор, то в параметре rank возвращается значение MPI_UNDEFINED.

```
int MPI_Initialized(int* flag);
function MPI_Initialized(var flag: longint): longint;
```

OUT flag – признак нахождения процесса в параллельном режиме (не равен 0, если процесс находится в параллельном режиме; равен 0 в противном случае).

Проверяет, находится ли процесс в параллельном режиме; точнее, была ли вызвана для данного процесса функция MPI_Init. Следует заметить, что после вызова функции MPI_Init вызов функции MPI_Initialized *всегда* возвращает ненулевое значение параметра flag, даже если в процессе уже была вызвана функция MPI_Finalize.

```
double MPI_Wtime();
function MPI_Wtime: double;
```

Возвращает время в секундах (вещественное число), прошедшее с некоторого момента в прошлом. Предполагается, что в программе будет использоваться *разность* между возвращаемыми значениями этой функции.

```
double MPI_Wtick();
function MPI_Wtick: double;
```

Возвращает длительность в секундах (вещественное число) между последовательными тактами срабатывания таймера.

```
int MPI_Get_processor_name(char* name, int* len);
function MPI_Get_processor_name(name: PChar; var len: longint):
     longint;
```

OUT name – имя узла сети;

OUT len – количество символов в имени узла.

Возвращает имя узла сети, на котором запущен текущий процесс. Для выходного строкового параметра name следует зарезервировать не менее MPI_MAX_PROCESSOR_NAME символов (в реализации MPICH для Windows константа MPI_MAX_PROCESSOR_NAME равна 256).

1.3. Блокирующая пересылка сообщений

При блокирующей пересылке выход из любой функции, связанной с операцией посылки или приема сообщения, происходит только после завершения этой операции. Имеются четыре варианта блокирующей переда-

чи сообщений с идентичным набором параметров и один вариант блокирующего приема сообщений.

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int
     dest, int msgtag, MPI_Comm comm);
function MPI_Send(buf: pointer; count: longint; datatype:
     MPI_Datatype; dest: longint; msgtag: longint; comm:
     MPI_Comm): longint;
```

IN `buf` – адрес начала буфера посылки сообщения;
IN `count` – число передаваемых элементов в сообщении;
IN `datatype` – тип передаваемых элементов;
IN `dest` – ранг процесса-получателя;
IN `msgtag` – идентификатор сообщения;
IN `comm` – коммуникатор.

Выполняет блокирующую посылку сообщения в *стандартном режиме* (standard mode). При этом сама среда MPI определяет, будет ли использован режим буферизации (с применением системного буфера). Если используется системный буфер, то операция посылки сообщения завершается после пересылки данных в этот буфер независимо от того, начался или нет процесс приема этого сообщения в процессе-получателе. Если системный буфер не используется, то операция посылки сообщения завершается только после того, как процесс-получатель начал действия по приему этого сообщения. Данная операция является нелокальной, т. е. ее завершение может зависеть от действий другого процесса.

```
int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int
     dest, int msgtag, MPI_Comm comm);
function MPI_Bsend(buf: pointer; count: longint; datatype:
     MPI_Datatype; dest: longint; msgtag: longint; comm:
     MPI_Comm): longint;
```

IN `buf` – адрес начала буфера посылки сообщения;
IN `count` – число передаваемых элементов в сообщении;
IN `datatype` – тип передаваемых элементов;
IN `dest` – ранг процесса-получателя;
IN `msgtag` – идентификатор сообщения;
IN `comm` – коммуникатор.

Выполняет блокирующую посылку сообщения в *режиме с буферизацией* (buffered mode). Предварительно в процессе-отправителе с помощью функции MPI_Buffer_attach должен быть выделен пользовательский буфер достаточного размера. Операция посылки сообщения завершается после пересылки данных в этот буфер независимо от того, начался или нет прием этого сообщения в процессе-получателе, поэтому операция буферизованной посылки данных является локальной.

```
int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int
     dest, int msgtag, MPI_Comm comm);
function MPI_Ssend(buf: pointer; count: longint; datatype:
     MPI_Datatype; dest: longint; msgtag: longint; comm:
     MPI_Comm): longint;
```

IN buf – адрес начала буфера посылки сообщения;
IN count – число передаваемых элементов в сообщении;
IN datatype – тип передаваемых элементов;
IN dest – ранг процесса-получателя;
IN msgtag – идентификатор сообщения;
IN comm – коммуникатор.

Выполняет блокирующую посылку сообщения в *синхронном режиме* (synchronous mode). Операция посылки в синхронном режиме может начаться независимо от того, инициирован ли прием этого сообщения в принимающем процессе, однако завершится эта операция только после начала приема этого сообщения процессом-получателем. Данная операция является нелокальной.

```
int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int
     dest, int msgtag, MPI_Comm comm);
function MPI_Rsend(buf: pointer; count: longint; datatype:
     MPI_Datatype; dest: longint; msgtag: longint; comm:
     MPI_Comm): longint;
```

IN buf – адрес начала буфера посылки сообщения;
IN count – число передаваемых элементов в сообщении;
IN datatype – тип передаваемых элементов;
IN dest – ранг процесса-получателя;
IN msgtag – идентификатор сообщения;
IN comm – коммуникатор.

Выполняет блокирующую посылку сообщения в *режиме «по готовности»* (ready mode). В этом режиме операция посылки может начаться только при условии, что процесс-получатель уже инициировал прием данного сообщения (в противном случае операция посылки считается ошибочной, и ее результат не определен). Данная операция является нелокальной.

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int
     source, int msgtag, MPI_Comm comm, MPI_Status* status);
function MPI_Recv(buf: pointer; count: longint; datatype:
     MPI_Datatype; source: longint; msgtag: longint; comm:
     MPI_Comm; var status: MPI_Status): longint;
```

OUT buf – адрес начала буфера приема сообщения;
IN count – максимальное число элементов в принимаемом сообщении;
IN datatype – тип элементов принимаемого сообщения;

IN source – ранг процесса-отправителя;

IN msgtag – идентификатор принимаемого сообщения;

IN comm – коммуникатор;

OUT status – параметры принятого сообщения.

Выполняет прием сообщения с блокировкой. Операция завершается только после заполнения буфера приема сообщения.

```
int MPI_Probe(int source, int msgtag, MPI_Comm comm, MPI_Status*
     status);
function MPI_Probe(source: longint; msgtag: longint; comm:
     MPI_Comm; var status: MPI_Status): longint;
```

IN source – ранг процесса-отправителя;

IN msgtag – идентификатор ожидаемого сообщения;

IN comm – коммуникатор;

OUT status – параметры обнаруженного сообщения.

Возвращает информацию о структуре ожидаемого сообщения (блокирующий вариант). Операция завершается только после завершения получения данных от процесса-отправителя. Данная функция обычно используется в ситуации, когда число элементов в переданном сообщении заранее неизвестно.

```
int MPI_Get_count(MPI_Status* status, MPI_Datatype datatype, int*
     count);
function MPI_Get_count(var status: MPI_Status; datatype:
     MPI_Datatype; var count: longint): longint;
```

IN status – параметры принятого сообщения;

IN datatype – тип элементов принятого сообщения;

OUT count – число элементов сообщения.

Определяет число элементов для уже принятого или ожидающего принятия сообщения.

```
int MPI_Buffer_attach(void* buf, int size);
function MPI_Buffer_attach(buf: pointer; size: longint): longint;
```

IN buf – адрес начала буфера;

IN size – размер буфера в байтах.

Задает буфер, используемый в дальнейшем при посылке сообщений в режиме с буферизацией. Размер буфера должен быть достаточным для хранения как пересылаемых сообщений, так и служебной информации. Размер памяти (в байтах), необходимый для размещения служебной информации, определяется константой MPI_BSEND_OVERHEAD; в реализации MPICH он равен 512. В каждый момент времени процесс может использовать только один буфер.

```
int MPI_Buffer_detach(void* buf, int* size);
function MPI_Buffer_detach(buf: pointer; var size: longint):
     longint;
```

OUT `buf` – адрес начала буфера;

OUT `size` – размер буфера в байтах.

Освобождает буфер, ранее выделенный для использования в режиме посылки сообщений с буферизацией, и возвращает его характеристики.

1.4. Неблокирующая пересылка сообщений

При неблокирующей пересылке сообщений операции посылки/приема сообщения лишь инициируют соответствующие действия, после чего немедленно завершаются, возвращая особый объект MPI — «запрос обмена» (request), с помощью которого в дальнейшем можно проверить состояние данной операции, используя либо функции группы Wait, блокирующие выполнение программы до полного завершения операции, либо неблокирующие функции группы Test. При завершении неблокирующей операции связанный с ней запрос обмена «сбрасывается», принимая значение MPI_REQUEST_NULL (это происходит либо при вызове функции группы Wait, либо при таком вызове функции группы Test, при котором была возвращена информация о завершении операции). Как и для блокирующей пересылки, имеются четыре варианта неблокирующей передачи сообщений с идентичным набором параметров и один вариант неблокирующего приема сообщений.

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int
     dest, int msgtag, MPI_Comm comm, MPI_Request* request);
function MPI_Isend(buf: pointer; count: longint; datatype:
     MPI_Datatype; dest: longint; msgtag: longint; comm:
     MPI_Comm; var request: MPI_Request): longint;
```

IN `buf` – адрес начала буфера посылки сообщения;

IN `count` – число передаваемых элементов в сообщении;

IN `datatype` – тип передаваемых элементов;

IN `dest` – ранг процесса-получателя;

IN `msgtag` – идентификатор сообщения;

IN `comm` – коммуникатор;

OUT `request` – идентификатор передачи («запрос обмена»).

Инициирует неблокирующую передачу сообщения в стандартном режиме (см. описание функции MPI_Send), возвращая связанный с ней запрос обмена. До сброса запроса обмена буфер buf нельзя повторно использовать.

```
int MPI_Ibsend(void* buf, int count, MPI_Datatype datatype, int
     dest, int msgtag, MPI_Comm comm, MPI_Request* request);
function MPI_Ibsend(buf: pointer; count: longint; datatype:
     MPI_Datatype; dest: longint; msgtag: longint; comm:
     MPI_Comm; var request: MPI_Request): longint;
```

IN buf – адрес начала буфера посылки сообщения;

IN count – число передаваемых элементов в сообщении;

IN datatype – тип передаваемых элементов;

IN dest – ранг процесса-получателя;

IN msgtag – идентификатор сообщения;

IN comm – коммуникатор;

OUT request – идентификатор передачи («запрос обмена»).

Инициирует неблокирующую передачу сообщения в режиме с буферизацией (см. описание функции MPI_Bsend), возвращая связанный с ней запрос обмена. До сброса запроса обмена буфер buf нельзя повторно использовать.

```
int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int
     dest, int msgtag, MPI_Comm comm, MPI_Request* request);
function MPI_Issend(buf: pointer; count: longint; datatype:
     MPI_Datatype; dest: longint; msgtag: longint; comm:
     MPI_Comm; var request: MPI_Request): longint;
```

IN buf – адрес начала буфера посылки сообщения;

IN count – число передаваемых элементов в сообщении;

IN datatype – тип передаваемых элементов;

IN dest – ранг процесса-получателя;

IN msgtag – идентификатор сообщения;

IN comm – коммуникатор;

OUT request – идентификатор передачи («запрос обмена»).

Инициирует неблокирующую передачу сообщения в синхронном режиме (см. описание функции MPI_Ssend), возвращая связанный с ней запрос обмена. До сброса запроса обмена буфер buf нельзя повторно использовать.

```
int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int
     dest, int msgtag, MPI_Comm comm, MPI_Request* request);
function MPI_Irsend(buf: pointer; count: longint; datatype:
     MPI_Datatype; dest: longint; msgtag: longint; comm:
     MPI_Comm; var request: MPI_Request): longint;
```

IN buf – адрес начала буфера посылки сообщения;

IN count – число передаваемых элементов в сообщении;

IN datatype – тип передаваемых элементов;

IN dest – ранг процесса-получателя;

IN msgtag – идентификатор сообщения;

IN comm – коммуникатор;

OUT request – идентификатор передачи («запрос обмена»).

Инициирует неблокирующую передачу сообщения в режиме «по готовности» (см. описание функции MPI_Rsend), возвращая связанный с ней запрос обмена. До сброса запроса обмена буфер buf нельзя повторно использовать.

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int
     source, int msgtag, MPI_Comm comm, MPI_Request* request);
function MPI_Irecv(buf: pointer; count: longint; datatype:
     MPI_Datatype; source: longint; msgtag: longint; comm:
     MPI_Comm; var request: MPI_Request): longint;
```

OUT buf – адрес начала буфера приема сообщения;

IN count – максимальное число элементов в принимаемом сообщении;

IN datatype – тип элементов принимаемого сообщения;

IN source – ранг процесса-отправителя;

IN msgtag – идентификатор принимаемого сообщения;

IN comm – коммуникатор;

OUT request – идентификатор приема («запрос обмена»).

Инициирует операцию приема сообщения без блокировки, возвращая связанный с ней запрос обмена. До сброса запроса обмена буфер buf нельзя повторно использовать.

```
int MPI_Wait(MPI_Request* request, MPI_Status* status);
function MPI_Wait(var request: MPI_Request; var status:
     MPI_Status): longint;
```

INOUT request – идентификатор приема или передачи («запрос обмена»);

OUT status – параметры завершившейся операции.

Ожидает завершения неблокирующей операции передачи или приема сообщения.

```
int MPI_Waitall(int count, MPI_Request* requests, MPI_Status*
     statuses);
function MPI_Waitall(count: longint; var requests: array of
     MPI_Request; var statuses: array of MPI_Status): longint;
```

IN count – число идентификаторов;

INOUT requests – массив идентификаторов приема или передачи («запросов обмена»);

OUT statuses – массив параметров завершившихся операций.

Блокирует выполнение процесса, пока все операции обмена, ассоциированные с указанными идентификаторами, не будут завершены.

```
int MPI_Waitany(int count, MPI_Request* requests, int* index,
     MPI_Status* status);
function MPI_Waitany(count: longint; var requests: array of
     MPI_Request; var index: longint; var status: MPI_Status):
     longint;
```

IN count – число идентификаторов;

INOUT requests – массив идентификаторов приема или передачи («запросов обмена»);

OUT index – номер завершенной операции обмена;

OUT status – параметры завершившейся операции.

Блокирует выполнение процесса, пока какая-либо операция обмена, ассоциированная с указанными идентификаторами, не будет завершена.

```
int MPI_Waitsome(int incount, MPI_Request* requests, int*
     outcount, int* indexes, MPI_Status* statuses);
function MPI_Waitsome(incount: longint; var requests: array of
     MPI_Request; var outcount: longint; var indexes: array of
     longint; var statuses: array of MPI_Status): longint;
```

IN incount – число идентификаторов;

INOUT requests – массив идентификаторов приема или передачи («запросов обмена»);

OUT outcount – число идентификаторов завершившихся операций обмена;

OUT indexes – массив номеров завершившихся операции обмена;

OUT statuses – массив параметров завершившихся операций.

Блокирует выполнение процесса, пока хотя бы одна из операций обмена, ассоциированных с указанными идентификаторами, не будет завершена. В отличие от функции MPI_Waitany может возвращать информацию о нескольких завершенных операциях.

```
int MPI_Test(MPI_Request* request, int* flag, MPI_Status*
     status);
function MPI_Test(var request: MPI_Request; var flag: longint;
     var status: MPI_Status): longint;
```

INOUT request – идентификатор приема или передачи («запрос обмена»);

OUT flag – признак завершенности операции обмена;

OUT status – параметры завершившейся операции.

Проверяет завершенность неблокирующей операции передачи или приема, ассоциированной с указанным идентификатором. Если операция завершена, то параметр flag возвращает ненулевое значение (при этом значение запроса обмена сбрасывается в MPI_REQUEST_NULL), в противном случае в параметре flag возвращается 0 (в этом случае состояние запроса обмена не изменяется).

```
int MPI_Testall(int count, MPI_Request* requests, int* flag,
     MPI_Status* statuses);
function MPI_Testall(count: longint; var requests: array of
     MPI_Request; var flag: longint; var statuses: array of
     MPI_Status): longint;
```

IN count – число идентификаторов;

INOUT requests – массив идентификаторов приема или передачи («запросов обмена»);

OUT flag – признак завершенности операций обмена;

OUT statuses – массив параметров завершившихся операций.

Проверяет завершенность всех неблокирующих операций передачи или приема, ассоциированных с массивом идентификаторов.

```
int MPI_Testany(int count, MPI_Request* requests, int* index,
     int* flag, MPI_Status* status);
function MPI_Testany(count: longint; var requests: array of
     MPI_Request; var index: longint; var flag: longint; var
     status: MPI_Status): longint;
```

IN count – число идентификаторов;

INOUT requests – массив идентификаторов приема или передачи («запросов обмена»);

OUT index – номер завершенной операции обмена;

OUT flag – признак завершенности операции обмена;

OUT status – параметры завершившейся операции.

Проверяет завершенность одной из неблокирующих операций передачи или приема, ассоциированных с массивом идентификаторов.

```
int MPI_Testsome(int incount, MPI_Request* requests, int*
     outcount, int* indexes, MPI_Status* statuses);
function MPI_Testsome(incount: longint; var requests: array of
     MPI_Request; var outcount: longint; var indexes: array of
     longint; var statuses: array of MPI_Status): longint;
```

IN incount – число идентификаторов;

INOUT requests – массив идентификаторов приема или передачи («запросов обмена»);

OUT outcount – число идентификаторов завершившихся операций обмена;

OUT indexes – массив номеров завершившихся операции обмена;

OUT statuses – массив параметров завершившихся операций.

Проверяет завершенность хотя бы одной из неблокирующих операций передачи или приема, ассоциированных с массивом идентификаторов. В отличие от функции MPI_Testany может возвращать информацию о нескольких завершенных операциях.

```
int MPI_Iprobe(int source, int msgtag, MPI_Comm comm, int* flag,
     MPI_Status* status);
function MPI_Iprobe(source: longint; msgtag: longint; comm:
     MPI_Comm; var flag: longint; var status: MPI_Status):
     longint;
```

IN source – ранг процесса-отправителя;

IN msgtag – идентификатор ожидаемого сообщения;

IN comm – коммуникатор;

OUT flag – признак завершенности операции обмена;

OUT status – параметры обнаруженного сообщения.

Возвращает информацию о поступлении ожидаемого сообщения и его структуре (неблокирующий вариант). Функция не ожидает завершения операции приема сообщения; если прием не завершен, то в параметре flag возвращается нулевое значение (в этом случае параметр status использовать не следует). Данная функция, как и ее блокирующий вариант MPI_Probe, обычно используется в ситуации, когда число элементов в переданном сообщении заранее неизвестно.

```
int MPI_Request_free(MPI_Request* request);
function MPI_Request_free(var request: MPI_Request): longint;
```

INOUT request – идентификатор приема или передачи («запрос обмена»).

Уничтожает указанный идентификатор приема или передачи, возвращая в аргументе значение MPI_REQUEST_NULL. При этом сама операция, связанная с данным идентификатором, не отменяется (если ранее она не была помечена для отмены), хотя получить информацию о ее завершении в дальнейшем будет невозможно.

```
int MPI_Cancel(MPI_Request* request);
function MPI_Cancel(var request: MPI_Request): longint;
```

INOUT request – идентификатор приема или передачи («запрос обмена»).

Помечает для отмены неблокирующую операцию приема или передачи, связанную с указанным запросом обмена. Для завершения отмены операции необходимо вызвать либо функцию MPI_Request_free, либо одну из функций группы Wait или Test.

```
int MPI_Test_cancelled(MPI_Status* status, int* flag);
function MPI_Test_cancelled(var status: MPI_Status; var flag:
     longint): longint;
```

IN status – параметры анализируемой операции;

OUT flag – признак отмены операции.

Позволяет проверить, была ли отменена неблокирующая операция, информация о которой ранее была возвращена функцией группы Wait или Test в параметре status. Если операция была успешно отменена, то пара-

метр flag возвращает ненулевое значение (в этом случае все основные поля параметра status являются неопределенными).

1.5. Отложенные и совмещенные запросы на взаимодействие

```
int MPI_Send_init(void* buf, int count, MPI_Datatype datatype,
     int dest, int msgtag, MPI_Comm comm, MPI_Request* request);
function MPI_Send_init(buf: pointer; count: longint; datatype:
     MPI_Datatype; dest: longint; msgtag: longint; comm:
     MPI_Comm; var request: MPI_Request): longint;
```

IN buf – адрес начала буфера посылки сообщения;
IN count – число передаваемых элементов в сообщении;
IN datatype – тип передаваемых элементов;
IN dest – ранг процесса-получателя;
IN msgtag – идентификатор сообщения;
IN comm – коммуникатор;
OUT request – идентификатор передачи («запрос обмена»).

Формирует отложенный запрос на выполнение пересылки данных в стандартном режиме.

```
int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype,
     int dest, int msgtag, MPI_Comm comm, MPI_Request* request);
function MPI_Bsend_init(buf: pointer; count: longint; datatype:
     MPI_Datatype; dest: longint; msgtag: longint; comm:
     MPI_Comm; var request: MPI_Request): longint;
```

IN buf – адрес начала буфера посылки сообщения;
IN count – число передаваемых элементов в сообщении;
IN datatype – тип передаваемых элементов;
IN dest – ранг процесса-получателя;
IN msgtag – идентификатор сообщения;
IN comm – коммуникатор;
OUT request – идентификатор передачи («запрос обмена»).

Формирует отложенный запрос на выполнение пересылки данных в режиме с буферизацией.

```
int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype,
     int dest, int msgtag, MPI_Comm comm, MPI_Request* request);
function MPI_Ssend_init(buf: pointer; count: longint; datatype:
     MPI_Datatype; dest: longint; msgtag: longint; comm:
     MPI_Comm; var request: MPI_Request): longint;
```

IN buf – адрес начала буфера посылки сообщения;
IN count – число передаваемых элементов в сообщении;
IN datatype – тип передаваемых элементов;
IN dest – ранг процесса-получателя;

IN msgtag – идентификатор сообщения;

IN comm – коммуникатор;

OUT request – идентификатор передачи («запрос обмена»).

Формирует отложенный запрос на выполнение пересылки данных в синхронном режиме.

```
int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype,
     int dest, int msgtag, MPI_Comm comm, MPI_Request* request);
function MPI_Rsend_init(buf: pointer; count: longint; datatype:
     MPI_Datatype; dest: longint; msgtag: longint; comm:
     MPI_Comm; var request: MPI_Request): longint;
```

IN buf – адрес начала буфера посылки сообщения;

IN count – число передаваемых элементов в сообщении;

IN datatype – тип передаваемых элементов;

IN dest – ранг процесса-получателя;

IN msgtag – идентификатор сообщения;

IN comm – коммуникатор;

OUT request – идентификатор передачи («запрос обмена»).

Формирует отложенный запрос на выполнение пересылки данных в режиме «по готовности».

```
int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype,
     int source, int msgtag, MPI_Comm comm, MPI_Request*
     request);
function MPI_Recv_init(buf: pointer; count: longint; datatype:
     MPI_Datatype; source: longint; msgtag: longint; comm:
     MPI_Comm; var request: MPI_Request): longint;
```

OUT buf – адрес начала буфера приема сообщения;

IN count – число принимаемых элементов в сообщении;

IN datatype – тип принимаемых элементов;

IN source – ранг процесса-отправителя;

IN msgtag – идентификатор сообщения;

IN comm – коммуникатор;

OUT request – идентификатор приема («запрос обмена»).

Формирует отложенный запрос на выполнение приема данных.

```
int MPI_Start(MPI_Request* request);
function MPI_Start(var request: MPI_Request): longint;
```

INOUT request – идентификатор приема или передачи («запрос обмена»).

Запускает в неблокирующем режиме отложенную операцию обмена, связанную с указанным идентификатором.

```
int MPI_Startall(int count, MPI_Request* requests);
function MPI_Startall(count: longint; var requests: array of
     MPI_Request): longint;
```

IN count – число запросов на взаимодействие;

INOUT requests – массив идентификаторов приема или передачи («запросов обмена»).

Запускает в неблокирующем режиме все отложенные операции обмена, связанные с указанными идентификаторами.

```
int MPI_Sendrecv(void* sbuf, int scount, MPI_Datatype stype, int
     dest, int stag, void* rbuf, int rcount, MPI_Datatype rtype,
     int source, int rtag, MPI_Comm comm, MPI_Status* status);
function MPI_Sendrecv(sbuf: pointer; scount: longint; stype:
     MPI_Datatype; dest: longint; stag: longint; rbuf: pointer;
     rcount: longint; rtype: MPI_Datatype; source: longint; rtag:
     longint; comm: MPI_Comm; var status: MPI_Status): longint;
```

IN sbuf – адрес начала буфера посылки сообщения;

IN scount – число передаваемых элементов в сообщении;

IN stype – тип передаваемых элементов;

IN dest – ранг процесса-получателя;

IN stag – идентификатор посылаемого сообщения;

OUT rbuf – адрес начала буфера приема сообщения;

IN rcount – число принимаемых элементов сообщения;

IN rtype – тип принимаемых элементов;

IN source – ранг процесса-отправителя;

IN rtag – идентификатор принимаемого сообщения;

IN comm – коммуникатор;

OUT status – параметры принятого сообщения.

Объединяет в одном запросе посылку и прием сообщений в стандартном режиме (с блокировкой).

```
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype
     datatype, int dest, int stag, int source, int rtag, MPI_Comm
     comm, MPI_Status* status);
function MPI_Sendrecv_replace(buf: pointer; count: longint;
     datatype: MPI_Datatype; dest: longint; stag: longint;
     source: longint; rtag: longint; comm: MPI_Comm; var status:
     MPI_Status): longint;
```

INOUT buf – адрес начала общего буфера посылки и приема сообщения;

IN count – число элементов в передаваемом и принимаемом сообщении;

IN datatype – тип передаваемых и принимаемых элементов;

IN dest – ранг процесса-получателя;

IN `stag` – идентификатор посылаемого сообщения;

IN `source` – ранг процесса-отправителя;

IN `rtag` – идентификатор принимаемого сообщения;

IN `comm` – коммуникатор;

OUT `status` – параметры принятого сообщения.

Объединяет в одном запросе посылку и прием сообщений в стандартном режиме (с блокировкой), используя общий буфер для посылки и приема данных.

1.6. Коллективное взаимодействие процессов

Все операции, связанные с коллективным взаимодействием процессов, выполняются в стандартном режиме с блокировкой.

```
int MPI_Barrier(MPI_Comm comm);
function MPI_Barrier(comm: MPI_Comm): longint;
```

IN `comm` – коммуникатор.

Блокирует работу процессов, вызвавших данную процедуру, пока все оставшиеся процессы коммуникатора также не выполнят эту процедуру.

```
int MPI_Bcast(void* buf, int count, MPI_Datatype datatype, int
     root, MPI_Comm comm);
function MPI_Bcast(buf: pointer; count: longint; datatype:
     MPI_Datatype; root: longint; comm: MPI_Comm): longint;
```

INOUT `buf` – адрес начала буфера рассылки/приема сообщения;

IN `count` – число передаваемых элементов в сообщении;

IN `datatype` – тип передаваемых элементов;

IN `root` – ранг процесса, выполняющего рассылку данных;

IN `comm` – коммуникатор.

Рассылает данные от процесса-источника всем процессам.

Пример:

```
                buf                    buf
Процесс 0:                             ( b0 b1 b2 b3 )
Процесс 1:  ( b0 b1 b2 b3 )  ==>  ( b0 b1 b2 b3 )
Процесс 2:                             ( b0 b1 b2 b3 )
```

```
int MPI_Gather(void* sbuf, int scount, MPI_Datatype stype, void*
     rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm
     comm);
function MPI_Gather(sbuf: pointer; scount: longint; stype:
     MPI_Datatype; rbuf: pointer; rcount: longint; rtype:
     MPI_Datatype; root: longint; comm: MPI_Comm): longint;
```

IN `sbuf` – адрес начала буфера рассылки;

IN `scount` – число элементов в посылаемом сообщении;

IN `stype` – тип элементов отсылаемого сообщения;

OUT rbuf – адрес начала буфера сборки данных;

IN rcount – число элементов, принимаемых от каждого процесса;

IN rtype – тип элементов принимаемого сообщения;

IN root – ранг процесса, выполняющего прием данных;

IN comm – коммуникатор.

Собирает данные со всех процессов в буфере процесса-приемника (от каждого процесса принимается одинаковое число элементов данных).

Пример:

```
            sbuf            rbuf
Процесс 0:  ( a0 a1 )
Процесс 1:  ( b0 b1 )  ==>  ( a0 a1 | b0 b1 | c0 c1 )
Процесс 2:  ( c0 c1 )
```

```
int MPI_Gatherv(void* sbuf, int scount, MPI_Datatype stype, void*
     rbuf, int* rcounts, int* displs, MPI_Datatype rtype, int
     root, MPI_Comm comm);
function MPI_Gatherv(sbuf: pointer; scount: longint; stype:
     MPI_Datatype; rbuf: pointer; const rcounts: array of
     longint; const displs: array of longint; rtype:
     MPI_Datatype; root: longint; comm: MPI_Comm): longint;
```

IN sbuf – адрес начала буфера рассылки;

IN scount – число элементов в посылаемом сообщении;

IN stype – тип элементов отсылаемого сообщения;

OUT rbuf – адрес начала буфера сборки данных;

IN rcounts – массив, в котором указывается число элементов, принимаемых от каждого процесса;

IN displs – массив смещений (в элементах) от начала буфера приема данных;

IN rtype – тип элементов принимаемого сообщения;

IN root – ранг процесса, выполняющего прием данных;

IN comm – коммуникатор.

Собирает данные со всех процессов в буфере процесса-приемника (от каждого процесса может приниматься различное число элементов данных). Данные, полученные от каждого процесса, записываются в буфер процесса-приемника со смещением, определяемым соответствующим элементом массива displs.

Пример:

```
            sbuf               rbuf
Процесс 0:  ( a0 a1 )
Процесс 1:  ( b0 )        ==>  ( a0 a1 | b0 | c0 c1 c2 )
Процесс 2:  ( c0 c1 c2 )
```

```
int MPI_Scatter(void* sbuf, int scount, MPI_Datatype stype, void*
     rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm
     comm);
function MPI_Scatter(sbuf: pointer; scount: longint; stype:
     MPI_Datatype; rbuf: pointer; rcount: longint; rtype:
     MPI_Datatype; root: longint; comm: MPI_Comm): longint;
```

IN sbuf – адрес начала буфера рассылки;
IN scount – число элементов, посылаемых каждому процессу;
IN stype – тип элементов отсылаемого сообщения;
OUT rbuf – адрес начала буфера приема данных;
IN rcount – число элементов в буфере приема данных;
IN rtype – тип элементов принимаемого сообщения;
IN root – ранг процесса, выполняющего рассылку данных;
IN comm – коммуникатор.

Рассылает данные от процесса-источника во все процессы группы (каждый процесс принимает одинаковое число элементов данных).

Пример:

```
            sbuf                             rbuf
Процесс 0:                                   ( b0 b1 )
Процесс 1:  ( b0 b1 | b2 b3 | b4 b5 )  ==>  ( b2 b3 )
Процесс 2:                                   ( b4 b5 )
```

```
int MPI_Scatterv(void* sbuf, int* scounts, int* displs,
     MPI_Datatype stype, void* rbuf, int rcount, MPI_Datatype
     rtype, int root, MPI_Comm comm);
function MPI_Scatterv(sbuf: pointer; const scounts: array of
     longint; const displs: array of longint; stype:
     MPI_Datatype; rbuf: pointer; rcount: longint; rtype:
     MPI_Datatype; root: longint; comm: MPI_Comm): longint;
```

IN sbuf – адрес начала буфера рассылки;
IN scounts – массив, в котором указывается число элементов, посылаемых каждому процессу;
IN displs – массив смещений (в элементах) от начала буфера рассылки;
IN stype – тип элементов отсылаемого сообщения;
OUT rbuf – адрес начала буфера приема данных;
IN rcount – число элементов в буфере приема данных;
IN rtype – тип элементов принимаемого сообщения;
IN root – ранг процесса, выполняющего рассылку данных;
IN comm – коммуникатор.

Рассылает данные от процесса-источника во все процессы группы (каждый процесс может принимать различное число элементов данных).

Данные, посылаемые каждому процессу, должны размещаться в буфере процесса-источника со смещением, определяемым соответствующим элементом массива displs.

Пример:

```
             sbuf                                rbuf
Процесс 0:                                       ( b0 b1 )
Процесс 1:  ( b0 b1 | b2 | b3 b4 b5 )  ==>  ( b2 )
Процесс 2:                                       ( b3 b4 b5 )
```

```
int MPI_Allgather(void* sbuf, int scount, MPI_Datatype stype,
     void* rbuf, int rcount, MPI_Datatype rtype, MPI_Comm comm);
function MPI_Allgather(sbuf: pointer; scount: longint; stype:
     MPI_Datatype; rbuf: pointer; rcount: longint; rtype:
     MPI_Datatype; comm: MPI_Comm): longint;
```

IN sbuf – адрес начала буфера рассылки;
IN scount – число элементов в посылаемом сообщении;
IN stype – тип элементов отсылаемого сообщения;
OUT rbuf – адрес начала буфера приема данных;
IN rcount – число элементов, принимаемых от каждого процесса;
IN rtype – тип элементов принимаемого сообщения;
IN comm – коммуникатор.

Собирает данные со всех процессов и размещает их в буфере приема каждого процесса (каждый процесс посылает одинаковое число элементов данных).

Пример:

```
             sbuf            rbuf
Процесс 0:  ( a0 a1 )       ( a0 a1 | b0 b1 | c0 c1 )
Процесс 1:  ( b0 b1 )  ==>  ( a0 a1 | b0 b1 | c0 c1 )
Процесс 2:  ( c0 c1 )       ( a0 a1 | b0 b1 | c0 c1 )
```

```
int MPI_Allgatherv(void* sbuf, int scount, MPI_Datatype stype,
     void* rbuf, int* rcounts, int* displs, MPI_Datatype rtype,
     MPI_Comm comm);
function MPI_Allgatherv(sbuf: pointer; scount: longint; stype:
     MPI_Datatype; rbuf: pointer; const rcounts: array of
     longint; const displs: array of longint; rtype:
     MPI_Datatype; comm: MPI_Comm): longint;
```

IN sbuf – адрес начала буфера рассылки;
IN scount – число элементов в посылаемом сообщении;
IN stype – тип элементов отсылаемого сообщения;
OUT rbuf – адрес начала буфера приема данных;
IN rcounts – массив, в котором указывается число элементов, принимаемых от каждого процесса;

IN displs – массив смещений (в элементах) от начала буфера приема данных;

IN rtype – тип элементов принимаемого сообщения;

IN comm – коммуникатор.

Собирает данные со всех процессов и размещает их в буфере приема каждого процесса (каждый процесс может посылать различное число элементов данных). Данные, полученные от каждого процесса, записываются в буфер процессов-приемников со смещением, определяемым соответствующим элементом массива displs.

Пример:

```
            sbuf                 rbuf
Процесс 0:  ( a0 a1 )            ( a0 a1 | b0 | c0 c1 c2 )
Процесс 1:  ( b0 )          ==>  ( a0 a1 | b0 | c0 c1 c2 )
Процесс 2:  ( c0 c1 c2 )         ( a0 a1 | b0 | c0 c1 c2 )
```

```
int MPI_Alltoall(void* sbuf, int scount, MPI_Datatype stype,
     void* rbuf, int rcount, MPI_Datatype rtype, MPI_Comm comm);
function MPI_Alltoall(sbuf: pointer; scount: longint; stype:
     MPI_Datatype; rbuf: pointer; rcount: longint; rtype:
     MPI_Datatype; comm: MPI_Comm): longint;
```

IN sbuf – адрес начала буфера рассылки;

IN scount – число элементов в посылаемом сообщении;

IN stype – тип элементов отсылаемого сообщения;

OUT rbuf – адрес начала буфера приема данных;

IN rcount – число элементов, принимаемых от каждого процесса;

IN rtype – тип элементов принимаемого сообщения;

IN comm – коммуникатор.

Обеспечивает рассылку каждым процессом различных данных всем другим процессам (каждый процесс принимает одинаковое число элементов данных от разных процессов).

Пример:

```
           sbuf                             rbuf
Процесс 0: ( a0 a1 | a2 a3 | a4 a5 )        ( a0 a1 | b0 b1 | c0 c1 )
Процесс 1: ( b0 b1 | b2 b3 | b4 b5 )   ==>  ( a2 a3 | b2 b3 | c2 c3 )
Процесс 2: ( c0 c1 | c2 c3 | c4 c5 )        ( a4 a5 | b4 b5 | c4 c5 )
```

```
int MPI_Alltoallv(void* sbuf, int* scounts, int* sdispls,
     MPI_Datatype stype, void* rbuf, int* rcounts, int* rdispls,
     MPI_Datatype rtype, MPI_Comm comm);
function MPI_Alltoallv(sbuf: pointer; const scounts: array of
     longint; const sdispls: array of longint; stype:
     MPI_Datatype; rbuf: pointer; const rcounts: array of
```

```
     longint; const rdispls: array of longint; rtype:
     MPI_Datatype; comm: MPI_Comm): longint;
```

IN sbuf – адрес начала буфера рассылки;

IN scounts – массив, в котором указывается число элементов, посылаемых каждому процессу;

IN sdispls – массив смещений (в элементах) от начала буфера рассылки;

IN stype – тип элементов отсылаемого сообщения;

OUT rbuf – адрес начала буфера приема данных;

IN rcounts – массив, в котором указывается число элементов, принимаемых от каждого процесса;

IN rdispls – массив смещений (в элементах) от начала буфера приема данных;

IN rtype – тип элементов принимаемого сообщения;

IN comm – коммуникатор.

Обеспечивает рассылку каждым процессом различных данных всем другим процессам (каждый процесс может принимать различное число элементов данных от разных процессов). Данные, посылаемые каждому процессу, должны размещаться в буфере процессов-источников со смещением, определяемым соответствующим элементом массива sdispls. Данные, полученные от каждого процесса, записываются в буфер процессов-приемников со смещением, определяемым соответствующим элементом массива rdispls.

Пример:

```
            sbuf                              rbuf
Процесс 0:  ( a0 a1 | a2 | a3 a4 a5 )         ( a0 a1 | b0 | c0 c1 c2 )
Процесс 1:  ( b0 | b1 b2 b3 | b4 b5 )   ==>   ( a2 | b1 b2 b3 | c3 c4 )
Процесс 2:  ( c0 c1 c2 | c3 c4 | c5 )         ( a3 a4 a5 | b4 b5 | c5 )
```

```
int MPI_Reduce(void* sbuf, void* rbuf, int count, MPI_Datatype
     datatype, MPI_Op op, int root, MPI_Comm comm);
function MPI_Reduce(sbuf: pointer; rbuf: pointer; count: longint;
     datatype: MPI_Datatype; op: MPI_Op; root: longint; comm:
     MPI_Comm): longint;
```

IN sbuf – адрес начала буфера для аргументов;

OUT rbuf – адрес начала буфера для результата;

IN count – число аргументов у каждого процесса;

IN datatype – тип аргументов;

IN op – идентификатор операции;

IN root – ранг процесса, выполняющего прием данных;

IN comm – коммуникатор.

Выполняет глобальную операцию с возвратом результатов в указанный процесс-приемник.

Пример:

```
              sbuf                 rbuf
Процесс 0:  ( a0 a1 a2 )
Процесс 1:  ( b0 b1 b2 )  ==>  ( a0+b0+c0 a1+b1+c1 a2+b2+c2 )
Процесс 2:  ( c0 c1 c2 )
```

```
int MPI_Allreduce(void* sbuf, void* rbuf, int count, MPI_Datatype
     datatype, MPI_Op op, MPI_Comm comm);
function MPI_Allreduce(sbuf: pointer; rbuf: pointer; count:
     longint; datatype: MPI_Datatype; op: MPI_Op; comm:
     MPI_Comm): longint;
```

IN sbuf – адрес начала буфера для аргументов;
OUT rbuf – адрес начала буфера для результата;
IN count – число аргументов у каждого процесса;
IN datatype – тип аргументов;
IN op – идентификатор операции;
IN comm – коммуникатор.

Выполняет глобальную операцию с возвратом результатов во все процессы.

Пример:

```
              sbuf                 rbuf
Процесс 0:  ( a0 a1 a2 )       ( a0+b0+c0 a1+b1+c1 a2+b2+c2 )
Процесс 1:  ( b0 b1 b2 )  ==>  ( a0+b0+c0 a1+b1+c1 a2+b2+c2 )
Процесс 2:  ( c0 c1 c2 )       ( a0+b0+c0 a1+b1+c1 a2+b2+c2 )
```

```
int MPI_Reduce_scatter(void* sbuf, void* rbuf, int* rcounts,
     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
function MPI_Reduce_scatter(sbuf: pointer; rbuf: pointer; const
     rcounts: array of longint; datatype: MPI_Datatype; op:
     MPI_Op; comm: MPI_Comm): longint;
```

IN sbuf – адрес начала буфера для аргументов;
OUT rbuf – адрес начала буфера для результатов;
IN rcounts – массив, в котором указывается число элементов, посылаемых каждому процессу;
IN datatype – тип аргументов;
IN op – идентификатор операции;
IN comm – коммуникатор.

Вначале выполняет глобальную операцию для всех элементов исходного буфера, а затем рассылает указанное количество результирующих элементов каждому процессу (каждый процесс может принимать различное число результирующих элементов).

Пример:

```
               sbuf                            rbuf
Процесс 0:  ( a0 a1 a2 a3 a4 a5)        ( a0+b0+c0 )
Процесс 1:  ( b0 b1 b2 b3 b4 b5)  ==>  ( a1+b1+c1 a2+b2+c2 a3+b3+c3 )
Процесс 2:  ( c0 c1 c2 c3 c4 c5)        ( a4+b4+c4 a5+b5+c5 )
```

```
int MPI_Scan(void* sbuf, void* rbuf, int count, MPI_Datatype
     datatype, MPI_Op op, MPI_Comm comm);
function MPI_Scan(sbuf: pointer; rbuf: pointer; count: longint;
     datatype: MPI_Datatype; op: MPI_Op; comm: MPI_Comm):
     longint;
```

IN sbuf – адрес начала буфера для аргументов;

OUT rbuf – адрес начала буфера для результатов;

IN count – число аргументов у каждого процесса;

IN datatype – тип аргументов;

IN op – идентификатор операции;

IN comm – коммуникатор.

Выполняет последовательность частичных глобальных операций (в i-й процесс посылается результат выполнения глобальной операции для процессов от нулевого до i-го включительно).

Пример:

```
             sbuf                 rbuf
Процесс 0:  ( a0 a1 a2 )         ( a0       a1       a2       )
Процесс 1:  ( b0 b1 b2 )   ==>   ( a0+b0    a1+b1    a2+b2    )
Процесс 2:  ( c0 c1 c2 )         ( a0+b0+c0 a1+b1+c1 a2+b2+c2 )
```

```
int MPI_Op_create(MPI_User_function* func, int commute,
     MPI_Op* op);
function MPI_Op_create(var func: MPI_User_function; commute:
     longint; var op: MPI_Op): longint;
```

IN func – адрес функции с определением новой пользовательской операции;

IN commute – признак коммутативности определяемой операции;

OUT op – идентификатор операции.

Определяет новую пользовательскую операцию.

```
int MPI_Op_free(MPI_Op* op);
function MPI_Op_free(var op: MPI_Op): longint;
```

INOUT op – идентификатор операции.

Уничтожает указанную пользовательскую операцию, возвращая в аргументе значение MPI_OP_NULL.

1.7. Определение пользовательских типов и упаковка данных

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
     MPI_Datatype* newtype);
function MPI_Type_contiguous(count: longint; oldtype:
     MPI_Datatype; var newtype: MPI_Datatype): longint;
```

IN `count` – количество элементов базового типа;

IN `oldtype` – базовый тип;

OUT `newtype` – производный тип.

Создает производный тип, состоящий из указанного числа последовательно расположенных элементов базового типа.

Пример:

```
Исходный тип: [T1]
Производный тип: [T1][T1][T1][T1][T1]
```

```
int MPI_Type_vector(int count, int blocklen, int stride,
     MPI_Datatype oldtype, MPI_Datatype* newtype);
function MPI_Type_vector(count: longint; blocklen: longint;
     stride: longint; oldtype: MPI_Datatype; var newtype:
     MPI_Datatype): longint;
```

IN `count` – количество блоков;

IN `blocklen` – количество элементов базового типа в каждом блоке;

IN `stride` – расстояние (в элементах базового типа) между начальными позициями соседних блоков;

IN `oldtype` – базовый тип;

OUT `newtype` – производный тип.

Создает производный тип, состоящий из указанного числа блоков, каждый из которых содержит одинаковое число элементов базового типа и находится на одном и том же расстоянии от начала предыдущего блока (расстояние указывается в количестве элементов базового типа).

Пример:

```
Исходный тип: [T1]
Участок памяти, равный протяженности исходного типа: [..]
Производный тип:
[T1][T1][T1][..][..][T1][T1][T1][..][..][T1][T1][T1]
```

```
int MPI_Type_hvector(int count, int blocklen, MPI_Aint stride,
     MPI_Datatype oldtype, MPI_Datatype* newtype);
function MPI_Type_hvector(count: longint; blocklen: longint;
     stride: MPI_Aint; oldtype: MPI_Datatype; var newtype:
     MPI_Datatype): longint;
```

IN `count` – количество блоков;

IN blocklen – количество элементов базового типа в каждом блоке;

IN stride – расстояние (в байтах) между начальными позициями соседних блоков;

IN oldtype – базовый тип;

OUT newtype – производный тип.

Создает производный тип, состоящий из указанного числа блоков, каждый из которых содержит одинаковое число элементов базового типа и находится на одном и том же расстоянии от начала предыдущего блока (расстояние указывается в байтах).

Пример:

```
Исходный тип: [T1]
Участок памяти, равный 1 байту (обозначается точкой): .
Производный тип: [T1][T1][T1]....[T1][T1][T1]....[T1][T1][T1]
```

```
int MPI_Type_indexed(int count, int* blocklens, int* displs,
     MPI_Datatype oldtype, MPI_Datatype* newtype);
function MPI_Type_indexed(count: longint; const blocklens: array
     of longint; const displs: array of longint; oldtype:
     MPI_Datatype; var newtype: MPI_Datatype): longint;
```

IN count – количество блоков;

IN blocklens – массив длин каждого блока (в элементах базового типа);

IN displs – массив смещений каждого блока от начала первого блока (в элементах базового типа);

IN oldtype – базовый тип;

OUT newtype – производный тип.

Создает производный тип, состоящий из указанного числа блоков, каждый из которых может содержать различное число элементов базового типа и находится на указанном расстоянии от начала первого блока (расстояние указывается в количестве элементов базового типа).

Пример:

```
Исходный тип: [T1]
Участок памяти, равный протяженности исходного типа: [..]
Производный тип:
[T1][T1][..][T1][T1][T1][..][..][T1][..][..][..][T1][T1]
```

```
int MPI_Type_hindexed(int count, int* blocklens, MPI_Aint*
     displs, MPI_Datatype oldtype, MPI_Datatype* newtype);
function MPI_Type_hindexed(count: longint; const blocklens: array
     of longint; const displs: array of MPI_Aint; oldtype:
     MPI_Datatype; var newtype: MPI_Datatype): longint;
```

IN count – количество блоков;

IN blocklens – массив длин каждого блока (в элементах базового типа);

IN displs – массив смещений каждого блока от начала первого блока (в байтах);

IN oldtype – базовый тип;

OUT newtype – производный тип.

Создает производный тип, состоящий из указанного числа блоков, каждый из которых может содержать различное число элементов базового типа и находится на указанном расстоянии от начала первого блока (расстояние указывается в байтах).

Пример:

```
Исходный тип: [T1]
Участок памяти, равный 1 байту (обозначается точкой): .
Производный тип: [T1][T1].[T1][T1][T1]...[T1].....[T1][T1]
```

```
int MPI_Type_struct(int count, int* blocklens, MPI_Aint* displs,
     MPI_Datatype* oldtypes, MPI_Datatype* newtype);
function MPI_Type_struct(count: longint; const blocklens: array
     of longint; const displs: array of MPI_Aint; const oldtypes:
     array of MPI_Datatype; var newtype: MPI_Datatype): longint;
```

IN count – количество блоков;

IN blocklens – массив длин каждого блока (в элементах соответствующего базового типа);

IN displs – массив смещений каждого блока от начала первого блока (в байтах);

IN oldtypes – массив базовых типов;

OUT newtype – производный тип.

Создает производный тип, состоящий из указанного числа блоков, каждый из которых может содержать различное число элементов базового типа и находится на указанном расстоянии от начала первого блока (расстояние указывается в байтах), причем для разных блоков можно указывать различные базовые типы.

Пример:

```
Исходные типы: [T1], [T2], [T3], [T4]
Участок памяти, равный 1 байту (обозначается точкой): .
Производный тип: [T1][T1].[T2][T2][T2]...[T3].....[T4][T4]
```

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint* extent);
function MPI_Type_extent(datatype: MPI_Datatype; var extent:
     MPI_Aint): longint;
```

IN datatype – тип данных;

OUT extent – протяженность элемента указанного типа.

Возвращает протяженность элемента указанного типа, т. е. количество байт, которое он занимает в памяти.

```
int MPI_Type_size(MPI_Datatype datatype, int* size);
function MPI_Type_size(datatype: MPI_Datatype; var size:
     longint): longint;
```

IN `datatype` – тип данных;

OUT `size` – размер элемента указанного типа.

Возвращает размер элемента указанного типа, равный суммарному размеру всех базовых элементов, входящих в этот тип (промежутки между элементами не учитываются). Размер, как и протяженность, измеряется в байтах.

```
int MPI_Type_commit(MPI_Datatype* datatype);
function MPI_Type_commit(var datatype: MPI_Datatype): longint;
```

INOUT `datatype` – регистрируемый тип данных.

Регистрирует производный тип для возможности его использования в операциях пересылки сообщений (незарегистрированные типы можно использовать при определении новых типов, но при пересылке данных их применять нельзя).

```
int MPI_Type_free(MPI_Datatype* datatype);
function MPI_Type_free(var datatype: MPI_Datatype): longint;
```

INOUT `datatype` – пользовательский тип данных.

Уничтожает указанный пользовательский тип данных, возвращая в аргументе значение MPI_DATATYPE_NULL. Производные типы, определенные с помощью данного пользовательского типа, сохраняются.

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype,
     void* outbuf, int outsize, int* position, MPI_Comm comm);
function MPI_Pack(inbuf: pointer; incount: longint; datatype:
     MPI_Datatype; outbuf: pointer; outsize: longint; var
     position: longint; comm: MPI_Comm): longint;
```

IN `inbuf` – адрес начала входного буфера;

IN `incount` – количество элементов во входном буфере;

IN `datatype` – тип элементов во входном буфере;

OUT `outbuf` – адрес начала выходного буфера (с упакованными данными);

IN `outsize` – размер выходного буфера (в байтах);

INOUT `position` – текущая позиция в выходном буфере (в байтах);

IN `comm` – коммуникатор, для которого упаковываются данные.

Упаковывает указанное количество элементов требуемого типа в выходном буфере, начиная с указанной позиции. После выполнения данной операции значение параметра position увеличивается, определяя новую те-

кущую позицию в выходном буфере. При первом вызове функции для данного выходного буфера параметр position следует положить равным 0.

```
int MPI_Unpack(void* inbuf, int insize, int* position, void*
     outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm);
function MPI_Unpack(inbuf: pointer; insize: longint; var
     position: longint; outbuf: pointer; outcount: longint;
     datatype: MPI_Datatype; comm: MPI_Comm): longint;
```

IN `inbuf` – адрес начала входного буфера (с упакованными данными);

IN `insize` – размер входного буфера (в байтах);

INOUT `position` – текущая позиция во входном буфере (в байтах);

OUT `outbuf` – адрес начала выходного буфера;

IN `outcount` – количество элементов, извлекаемых из входного буфера;

IN `datatype` – тип элементов, извлекаемых из входного буфера;

IN `comm` – коммуникатор, от которого получены распаковываемые данные.

Распаковывает указанное количество элементов требуемого типа в выходном буфере, начиная с указанной позиции входного буфера. После выполнения данной операции значение параметра position увеличивается, определяя новую текущую позицию во входном буфере. При первом вызове функции для данного входного буфера параметр position следует положить равным 0.

```
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm
     comm, int* size);
function MPI_Pack_size(incount: longint; datatype: MPI_Datatype;
     comm: MPI_Comm; var size: longint): longint;
```

IN `incount` – количество упаковываемых элементов;

IN `datatype` – тип упаковываемых элементов;

IN `comm` – коммуникатор, для которого упаковываются данные;

OUT `size` – размер (в байтах), необходимый для упаковки.

Определяет объем памяти (в байтах), *достаточный* для хранения incount упакованных данных типа datatype. Возвращенное значение size может оказаться *больше* того, которое реально потребуется для хранения указанного числа упакованных данных.

1.8. Работа с группами процессов и коммуникаторами

```
int MPI_Group_size(MPI_Group group, int* size);
function MPI_Group_size(group: MPI_Group; var size: longint):
     longint;
```

IN `group` – группа процессов;

OUT `size` – количество процессов в группе.

Определяет количество процессов в указанной группе.

```
int MPI_Group_rank(MPI_Group group, int* rank);
function MPI_Group_rank(group: MPI_Group; var rank: longint):
     longint;
```

IN group – группа процессов;

OUT rank – ранг текущего процесса в группе.

Определяет ранг текущего (т. е. вызвавшего данную функцию) процесса в указанной группе. Если текущий процесс не входит в указанную группу, то в параметре rank возвращается значение MPI_UNDEFINED.

```
int MPI_Group_translate_ranks(MPI_Group group1, int n, int*
     ranks1, MPI_Group group2, int* ranks2);
function MPI_Group_translate_ranks(group1: MPI_Group; n: longint;
     const ranks1: array of longint; group2: MPI_Group; var
     ranks2: array of longint): longint;
```

IN group1 – первая группа;

IN n – число анализируемых процессов;

IN ranks1 – массив рангов процессов для первой группы;

IN group2 – вторая группа;

OUT ranks2 – массив рангов тех же процессов для второй группы.

Позволяет определить ранги процессов во второй группе, если известны их ранги в первой группе. Если какой-либо из процессов первой группы не входит во вторую группу, то соответствующему элементу массива ranks2 присваивается значение MPI_UNDEFINED.

```
int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int*
     result);
function MPI_Group_compare(group1: MPI_Group; group2: MPI_Group;
     var result: longint): longint;
```

IN group1 – первая группа процессов;

IN group2 – вторая группа процессов;

OUT result – результат сравнения групп.

Сравнивает две группы процессов (возможные результаты сравнения, возвращаемые данной функцией, перечислены в п. 7.1 при описании типа MPI_Group).

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group* group);
function MPI_Comm_group(comm: MPI_Comm; var group: MPI_Group):
     longint;
```

IN comm – коммуникатор;

OUT group – группа процессов, соответствующая коммуникатору.

Возвращает группу процессов, соответствующую указанному коммуникатору.

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
     MPI_Group* newgroup);
function MPI_Group_union(group1: MPI_Group; group2: MPI_Group;
     var newgroup: MPI_Group): longint;
```

IN group1 – первая группа процессов;

IN group2 – вторая группа процессов;

OUT newgroup – новая группа процессов.

Возвращает новую группу процессов, являющуюся объединением двух указанных групп. Объединение состоит из всех процессов первой группы (взятых в том же порядке), дополненных теми процессами второй группы (в том же порядке), которые отсутствуют в первой группе. Операция объединения групп не является коммутативной, так как при перемене местами исходных групп может измениться порядок расположения процессов в результирующем объединении.

```
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
     MPI_Group* newgroup);
function MPI_Group_intersection(group1: MPI_Group; group2:
     MPI_Group; var newgroup: MPI_Group): longint;
```

IN group1 – первая группа процессов;

IN group2 – вторая группа процессов;

OUT newgroup – новая группа процессов.

Возвращает новую группу процессов, являющуюся пересечением двух указанных групп. Пересечение состоит из тех процессов первой группы (взятых в том же порядке), которые входят во вторую группу. Полученная группа может быть пустой; в этом случае в параметре newgroup возвращается значение MPI_GROUP_EMPTY. Операция пересечения групп не является коммутативной, так как при перемене местами исходных групп может измениться порядок расположения процессов в результирующем пересечении.

```
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
     MPI_Group* newgroup);
function MPI_Group_difference(group1: MPI_Group; group2:
     MPI_Group; var newgroup: MPI_Group): longint;
```

IN group1 – первая группа процессов;

IN group2 – вторая группа процессов;

OUT newgroup – новая группа процессов.

Возвращает новую группу процессов, являющуюся разностью двух указанных групп. Разность состоит из тех процессов первой группы (взятых в том же порядке), которые не входят во вторую группу. Полученная группа может быть пустой; в этом случае в параметре newgroup возвращается значение MPI_GROUP_EMPTY.

```
int MPI_Group_incl(MPI_Group group, int n, int* ranks, MPI_Group*
     newgroup);
function MPI_Group_incl(group: MPI_Group; n: longint; const
     ranks: array of longint; var newgroup: MPI_Group): longint;
```

IN group – исходная группа процессов;

IN n – размер новой группы процессов;

IN ranks – массив рангов процессов из исходной группы;

OUT newgroup – новая группа.

Создает новую группу процессов, содержащую указанные процессы исходной группы в указанном порядке.

```
int MPI_Group_excl(MPI_Group group, int n, int* ranks, MPI_Group*
     newgroup);
function MPI_Group_excl(group: MPI_Group; n: longint; const
     ranks: array of longint; var newgroup: MPI_Group): longint;
```

IN group – исходная группа процессов;

IN n – размер удаляемой части процессов;

IN ranks – массив рангов процессов, удаляемых из исходной группы;

OUT newgroup – новая группа процессов.

Создает новую группу процессов, содержащую все процессы исходной группы, за исключением указанных.

```
int MPI_Group_free(MPI_Group* group);
function MPI_Group_free(var group: MPI_Group): longint;
```

INOUT group – группа процессов.

Уничтожает указанную группу процессов, возвращая в аргументе значение MPI_GROUP_NULL.

```
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int*
     result);
function MPI_Comm_compare(comm1: MPI_Comm; comm2: MPI_Comm; var
     result: longint): longint;
```

IN comm1 – первый коммуникатор;

IN comm2 – второй коммуникатор;

OUT result – результат сравнения коммуникаторов.

Сравнивает два коммуникатора (возможные результаты сравнения, возвращаемые данной функцией, перечислены в п. 7.1 при описании типа MPI_Comm).

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm* newcomm);
function MPI_Comm_dup(comm: MPI_Comm; var newcomm: MPI_Comm):
     longint;
```

IN comm – исходный коммуникатор;

OUT newcomm – новый коммуникатор.

Создает новый коммуникатор, содержащий ту же группу процессов и те же атрибуты, что и указанный коммуникатор.

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm*
     newcomm);
function MPI_Comm_create(comm: MPI_Comm; group: MPI_Group; var
     newcomm: MPI_Comm): longint;
```

IN comm – исходный коммуникатор;

IN group – группа процессов;

OUT newcomm – новый коммуникатор.

Создает новый коммуникатор, содержащий указанную группу процессов, входящих в исходный коммуникатор. Данная функция должна быть вызвана во всех процессах, входящих в коммуникатор comm; при этом для тех процессов, которые не входят в указанную группу group, в параметре newcomm будет возвращено значение MPI_COMM_NULL.

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm*
     newcomm);
function MPI_Comm_split(comm: MPI_Comm; color: longint; key:
     longint; var newcomm: MPI_Comm): longint;
```

IN comm – исходный коммуникатор;

IN color – признак разделения на группы (неотрицательное число);

IN key – ключ, определяющий нумерацию в новом коммуникаторе;

OUT newcomm – новый коммуникатор.

Разбивает множество процессов, входящих в указанный коммуникатор, на отдельные коммуникаторы (по числу различных значений аргумента color); возвращает тот из полученных коммуникаторов, в который входит текущий процесс. Если текущий процесс не следует включать ни в один из создаваемых коммуникаторов, то в качестве параметра color надо указать константу MPI_UNDEFINED; в этом случае в параметре newcomm будет возвращено значение MPI_COMM_NULL. Данная функция должна быть вызвана во всех процессах, входящих в коммуникатор comm.

```
int MPI_Comm_free(MPI_Comm comm);
function MPI_Comm_free(comm: MPI_Comm): longint;
```

INOUT comm – коммуникатор.

Уничтожает указанный коммуникатор, возвращая в аргументе значение MPI_COMM_NULL.

1.9. Виртуальные топологии

```
int MPI_Topo_test(MPI_Comm comm, int* status);
function MPI_Topo_test(comm: MPI_Comm; var status: longint):
     longint;
```

IN comm – коммуникатор;

OUT status – обнаруженный тип топологии.

Позволяет определить тип топологии, связанной с указанным коммуникатором (возможные значения, возвращаемые данной функцией, перечислены в п. 7.1 при описании типа MPI_Comm).

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int* dims, int*
      periods, int reorder, MPI_Comm* comm_cart);
function MPI_Cart_create(comm_old: MPI_Comm; ndims: longint;
      const dims: array of longint; const periods: array of
      longint; reorder: longint; var comm_cart: MPI_Comm):
      longint;
```

IN comm_old – исходный коммуникатор;

IN ndims – размерность декартовой решетки;

IN dims – массив с числом процессов по каждому измерению декартовой решетки;

IN periods – массив флагов, определяющих периодичность каждого измерения (измерение является периодическим, если соответствующий флаг не равен 0);

IN reorder – флаг, разрешающий (при ненулевом значении) или запрещающий (при нулевом значении) среде MPI изменять порядок нумерации процессов;

OUT comm_cart – коммуникатор с декартовой топологией.

Создает на основе исходного коммуникатора новый коммуникатор с декартовой топологией и таким же или меньшим числом процессов. Если текущий процесс не входит в созданную декартову решетку, то в параметре comm_cart возвращается значение MPI_COMM_NULL.

```
int MPI_Dims_create(int nnodes, int ndims, int* dims);
function MPI_Dims_create(nnodes: longint; ndims: longint; var
      dims: array of longint): longint;
```

IN nnodes – число узлов декартовой решетки;

IN ndims – размерность декартовой решетки;

INOUT dims – массив с числом узлов по каждому измерению декартовой решетки.

Позволяет определить оптимальное число узлов по каждому измерению декартовой решетки указанной размерности. Элементы массива dims, требующие определения, должны иметь нулевые исходные значения; исходные положительные значения элементов массива dims считаются фиксированными и не изменяются. В случае отрицательных исходных значений или невозможности выбора хотя бы одного варианта требуемой декартовой решетки функция возвращает значение, отличное от MPI_SUCCESS.

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int* index,
      int* edges, int reorder, MPI_Comm* comm_graph);
function MPI_Graph_create(comm_old: MPI_Comm; nnodes: longint;
      const index: array of longint; const edges: array of
```

```
     longint; reorder: longint; var comm_graph: MPI_Comm):
     longint;
```

IN comm_old – исходный коммуникатор;

IN nnodes – число вершин графа;

IN index – массив степеней вершин, i-й элемент которого равен суммарному количеству соседей для первых i вершин (например, если начальная вершина имеет двух соседей, а следующая за ней вершина — трех, то два первых элемента массива index будут равны 2 и 5);

IN edges – массив ребер, содержащий упорядоченный список соседей для всех вершин (вначале указываются все соседи начальной вершины в порядке возрастания их номеров, затем — все соседи вершины, следующей за начальной, также в порядке возрастания их номеров, и т. д.); вершины нумеруются от 0;

IN reorder – флаг, разрешающий (при ненулевом значении) или запрещающий (при нулевом значении) среде MPI изменять порядок нумерации процессов;

OUT comm_graph – коммуникатор с топологией графа.

Создает на основе исходного коммуникатора новый коммуникатор с топологией графа и таким же или меньшим числом процессов. Если текущий процесс не входит в созданный коммуникатор, то в параметре comm_graph возвращается значение MPI_COMM_NULL.

```
int MPI_Graphdims_get(MPI_Comm comm, int* nnodes, int* nedges);
function MPI_Graphdims_get(comm: MPI_Comm; var nnodes: longint;
     var nedges: longint): longint;
```

IN comm – коммуникатор с топологией графа;

OUT nnodes – число вершин графа;

OUT nedges – число ребер графа.

Позволяет определить число вершин и число ребер графа для коммуникатора с топологией графа.

```
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int*
     index, int* edges);
function MPI_Graph_get(comm: MPI_Comm; maxindex: longint;
     maxedges: longint; var index: array of longint; var edges:
     array of longint): longint;
```

IN comm – коммуникатор с топологией графа;

IN maxindex – число элементов массива степеней вершин index;

IN maxedges – число элементов массива ребер edges;

OUT index – массив степеней вершин;

OUT edges – массив ребер.

Позволяет для коммуникатора с топологией графа восстановить характеристики, указанные при его создании.

```
int MPI_Cartdim_get(MPI_Comm comm, int* ndims);
function MPI_Cartdim_get(comm: MPI_Comm; var ndims: longint):
     longint;
```

IN comm – коммуникатор с декартовой топологией;

OUT ndims – размерность декартовой решетки.

Позволяет определить размерность декартовой решетки для коммуникатора с декартовой топологией.

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int* dims, int*
     periods, int* coords);
function MPI_Cart_get(comm: MPI_Comm; maxdims: longint; var dims:
     array of longint; var periods: array of longint; var coords:
     array of longint): longint;
```

IN comm – коммуникатор с декартовой топологией;

IN maxdims – число элементов каждого из выходных массивов;

OUT dims – массив с числом процессов по каждому измерению декартовой решетки;

OUT periods – массив флагов, определяющих периодичность каждого измерения (измерение является периодическим, если соответствующий флаг не равен 0);

OUT coords – массив декартовых координат текущего процесса.

Позволяет для коммуникатора с декартовой топологией восстановить характеристики, указанные при его создании, а также узнать декартовы координаты текущего процесса.

```
int MPI_Cart_rank(MPI_Comm comm, int* coords, int* rank);
function MPI_Cart_rank(comm: MPI_Comm; const coords: array of
     longint; var rank: longint): longint;
```

IN comm – коммуникатор с декартовой топологией;

IN coords – массив декартовых координат;

OUT rank – ранг процесса.

Позволяет определить ранг процесса, входящего в коммуникатор с декартовой топологией, по декартовым координатам этого процесса (начальное значение каждой координаты считается равным 0).

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int*
     coords);
function MPI_Cart_coords(comm: MPI_Comm; rank: longint; maxdims:
     longint; var coords: array of longint): longint;
```

IN comm – коммуникатор с декартовой топологией;

IN rank – ранг процесса;

IN maxdims – число элементов массива декартовых координат;

OUT coords – массив декартовых координат.

Позволяет определить по рангу процесса, входящего в коммуникатор с декартовой топологией, декартовы координаты этого процесса (начальное значение каждой координаты считается равным 0).

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int*
     nneighbors);
function MPI_Graph_neighbors_count(comm: MPI_Comm; rank: longint;
     var nneighbors: longint): longint;
```

IN comm – коммуникатор с топологией графа;

IN rank – ранг процесса;

OUT nneighbors – количество соседей процесса в графе.

Позволяет определить по рангу процесса, входящего в коммуникатор с топологией графа, число его непосредственных соседей в графе.

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int
     maxneighbors, int* neighbors);
function MPI_Graph_neighbors(comm: MPI_Comm; rank: longint;
     maxneighbors: longint; var neighbors: array of longint):
     longint;
```

IN comm – коммуникатор с топологией графа;

IN rank – ранг процесса;

IN maxneighbors – число элементов массива соседей;

OUT neighbors – массив рангов процессов-соседей.

Позволяет определить по рангу процесса, входящего в коммуникатор с топологией графа, массив рангов процессов, являющихся его непосредственными соседями.

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int*
     rank_source, int* rank_dest);
function MPI_Cart_shift(comm: MPI_Comm; direction: longint; disp:
     longint; var rank_source: longint; var rank_dest: longint):
     longint;
```

IN comm – коммуникатор с декартовой топологией;

IN direction – номер декартовой координаты (нумерация начинается от 0);

IN disp – смещение (может быть отрицательным);

OUT rank_source – ранг посылающего процесса;

OUT rank_dest – ранг принимающего процесса.

Позволяет определить для текущего процесса ранги посылающего и принимающего процесса при пересылке данных по указанной координате (т. е. при обычном или циклическом сдвиге данных вдоль указанного направления).

```
int MPI_Cart_sub(MPI_Comm comm, int* remain_dims, MPI_Comm*
     newcomm);
function MPI_Cart_sub(comm: MPI_Comm; const remain_dims: array of
     longint; var newcomm: MPI_Comm): longint;
```

IN `comm` – коммуникатор с декартовой топологией;

IN `remain_dims` – массив флагов, определяющий номера остающихся измерений;

OUT `newcomm` – новый коммуникатор.

Возвращает один из коммуникаторов, полученных в результате расщепления исходной декартовой решетки на подрешетки меньшей размерности (возвращается коммуникатор, в который входит текущий процесс).

2. Электронный задачник Programming Taskbook for MPI

2.1. Общее описание задачника PT for MPI

Комплекс «Электронный задачник по параллельному MPI-программированию Programming Taskbook for MPI» (PT for MPI) содержит дополнительные компоненты электронного задачника Programming Taskbook, которые позволяют выполнять задания на разработку параллельных программ с применением технологии MPI. Имеется вариант данного комплекса, предназначенный для использования совместно с вариантом задачника Programming Taskbook, интегрированным в среду PascalABC.NET.

Для возможности использования комплекса PT for MPI его следует установить в системный каталог базового варианта электронного задачника Programming Taskbook версии не ниже 4.9 (обычно системным каталогом задачника является каталог C:\Program Files\PT4). Вариант комплекса, предназначенный для использования в среде PascalABC.NET, должен устанавливаться в подкаталог PT4 системного каталога среды PascalABC.NET (обычно это каталог C:\Program Files\PascalABC.NET). На компьютере надо также установить систему MPICH для Windows версии 1.2.5, которая требуется для запуска программ учащихся в параллельном режиме (дистрибутив данной версии системы MPICH доступен по адресу **ftp://ftp.mcs.anl.gov/pub/mpi/nt/mpich.nt.1.2.5.exe**).

Комплекс PT for MPI является свободно распространяемым программным продуктом (freeware); он может использоваться как с полным вариантом задачника PT4Complete–1100, так и со свободно распространяемым мини-вариантом PT4Mini–270.

Задания по параллельному MPI-программированию могут выполняться на языках Паскаль и C++ в следующих программных средах:

- Borland Delphi 7.0 и Turbo Delphi 2006;
- Free Pascal Lazarus 0.9;
- PascalABC.NET;
- Microsoft Visual C++ 6.0;
- Microsoft Visual Studio .NET 2003, 2005 и 2008.

К программам учащегося должна подключаться динамическая библиотека mpich.dll, входящая в состав комплекса MPICH. Для программ на C++ доступ к библиотеке обеспечивается с помощью файла mpich.lib и набора заголовочных файлов (также входящих в комплекс MPICH), а для программ на Паскале — с помощью модуля MPI.pas, который разработан автором задачника PT for MPI и содержит описания констант, типов и более 120 функций MPI стандарта 1.1.

Задачник PT for MPI предоставляет при выполнении заданий те же возможности, что и базовый задачник Programming Taskbook; в частности, он передает программе учащегося исходные данные, проверяет правильность результатов, полученных программой, и сохраняет сведения о каждом тестовом испытании программы в специальном файле. Кроме того, в задачнике PT for MPI предусмотрены дополнительные возможности, связанные со спецификой выполнения заданий по параллельному программированию:

- демонстрационный просмотр заданий, не требующий использования параллельного режима;
- создание для выбранного задания проекта-заготовки с подключенными к нему модулями библиотеки MPI;
- особый механизм, обеспечивающий выполнение программы учащегося в параллельном режиме при ее обычном запуске из среды разработки: запущенная программа выполняет запуск приложения MPIRun.exe из комплекса MPICH, которое, в свою очередь, запускает программу в параллельном режиме (все процессы выполняются на локальном компьютере);
- передача каждому процессу параллельной программы его собственного набора исходных данных;
- получение от каждого процесса требуемых результатов и их автоматическая пересылка в главный процесс для проверки и отображения в окне задачника;
- вывод информации об ошибках времени выполнения (в том числе ошибках, возникших при выполнении функций MPI) и ошибках ввода-вывода с указанием рангов процессов, в которых эти ошибки произошли;
- возможность вывода отладочной информации для каждого процесса в специальном разделе окна задачника;
- автоматическая выгрузка из памяти всех запущенных процессов даже в случае зависания параллельной программы.

Перечисленные выше возможности избавляют учащегося от необходимости выполнения дополнительных действий, связанных с запуском его программы в параллельном режиме, и упрощают выявление и исправление стандартных ошибок, возникающих в параллельных программах.

Использование исходных данных, подготовленных задачником для каждого процесса параллельной программы, наглядный вывод в одном окне всех результатов, полученных каждым процессом, и их автоматическая проверка, а также дополнительные средства отладки позволяют учащемуся сосредоточиться на реализации алгоритма решения задачи и обеспечивают надежное тестирование разработанного алгоритма.

В состав задачника PT for MPI включена группа учебных заданий MPIBegin, которая содержит 100 заданий, предназначенных для освоения библиотеки MPI.

Реализованные в задачнике PT for MPI средства автоматического запуска и отладки параллельных приложений позволяют использовать его для разработки и тестирования параллельных программ, не связанных с конкретными учебными заданиями. С этой целью в задачник PT for MPI включена вспомогательная группа MPIDebug из 36 заданий, каждое из которых обеспечивает автоматический запуск параллельной программы, причем количество процессов определяется порядковым номером задания. Таким образом, задания группы MPIDebug дают возможность запускать любые параллельные программы с требуемым количеством процессов непосредственно из интегрированной среды и предоставляют для их отладки средства, имеющиеся в задачнике.

С помощью конструктора учебных заданий PT4TaskMaker преподаватель может разрабатывать дополнительные группы заданий, связанные с параллельным MPI-программированием.

Подробная информация о возможностях задачника PT for MPI содержится на сайте **ptaskbook.com**. С этого сайта можно загрузить последние версии задачника Programming Taskbook и его расширения PT for MPI.

2.2. Средства задачника для инициализации заданий и ввода-вывода данных

В данном разделе содержится описание основных подпрограмм задачника Programming Taskbook for MPI, обеспечивающих инициализацию задания, ввод исходных данных и вывод результатов. На языке Паскаль эти подпрограммы реализованы в виде процедур, а на языке C++ — в виде функций типа void; для единообразия во всех последующих описаниях данные подпрограммы называются *процедурами*.

Поскольку в заданиях по параллельному программированию, приведенных в настоящем пособии, используются только числовые типы данных, в описание не включены процедуры базового варианта задачника Programming Taskbook, предназначенные для ввода-вывода данных логического типа, символов, строк и указателей.

Все процедуры задачника описаны в файлах pt4.pas (язык Паскаль) и pt4.h и pt4.cpp (язык C++). При создании проекта-заготовки с использованием программного модуля PT4Load данные файлы подключаются к этому проекту автоматически.

```
void Task(char* name);                                          C++
procedure Task(name: string);                                   Pascal
```

Инициализирует задание с именем name. Процедура Task должна вызываться в начале программы, выполняющей это задание (до вызова процедур ввода-вывода Get–Put). Если в программе, выполняющей задание, не указана процедура Task, то при запуске программы будет выведено окно с сообщением «*Не вызвана процедура Task с именем задания*».

Если процедура Task вызывается в программе несколько раз, то все последующие ее вызовы игнорируются.

Имя задания должно включать имя группы заданий и порядковый номер в пределах группы (например, «MPIBegin1»). Регистр букв в имени группы может быть произвольным. Если указана неверная группа, то программа выведет сообщение об ошибке, в котором будут перечислены названия всех имеющихся групп. Если указан недопустимый номер задания, то программа выведет сообщение, в котором будет указан диапазон допустимых номеров для данной группы. Если после имени задания в параметре name указан символ «?» (например, «MPIBegin1?»), то программа будет работать в демонстрационном режиме, имеющем следующие особенности:

- даже если программа содержит решение задания, это решение не анализируется и информация в файл результатов не заносится;
- после отображения на экране окна задачника в разделе результатов сразу становится активной вкладка «Пример верного решения»;
- при одном запуске программы можно просмотреть несколько вариантов исходных и контрольных данных; для смены набора данных требуется нажать кнопку «Новые данные» или клавишу пробела;
- при одном запуске программы можно последовательно просмотреть все задания данной группы; для перехода к заданию с большим номером требуется нажать кнопку «Следующее задание» или клавишу [Enter], а для перехода к заданию с меньшим номером требуется нажать кнопку «Предыдущее задание» или клавишу [Backspace] (задания перебираются циклически).

Демонстрационный запуск задания по параллельному программированию выполняется в непараллельном режиме, без обращения к программным средствам комплекса MPICH.

Процедура Task может также использоваться для генерации и вывода на экран html-страницы с текстом задания или группы заданий. Для этого необходимо указать в качестве параметра name имя конкретного задания или группы заданий и символ «#», например, «MPIBegin1#» или «MPIBegin#». Для включения в html-страницу нескольких заданий или

групп заданий достаточно для каждого задания или группы вызвать процедуру Task с параметром, оканчивающимся символом #.

```
void GetN(int& a);                                          C++
void GetD(double& a);                                       C++
procedure GetN(var a: integer);                          Pascal
procedure GetR(var a: real);                             Pascal
```

Процедуры обеспечивают ввод исходных числовых данных в программу, выполняющую учебное задание. Они должны вызываться после вызова процедуры Task; в случае их вызова до вызова процедуры Task при запуске программы будет выведено сообщение об ошибке «*В начале программы не вызвана процедура Task с именем задания*».

Используемая процедура ввода должна соответствовать типу очередного элемента исходных данных; в противном случае выводится сообщение об ошибке «*Неверно указан тип при вводе исходных данных*» (такое сообщение будет выведено, например, если очередной элемент данных является вещественным числом, а для его ввода используется процедура GetN).

При попытке ввести больше исходных данных, чем это предусмотрено в задании, выводится сообщение об ошибке «*Попытка ввести лишние исходные данные*». Если исходные данные, необходимые для решения задания, введены не полностью, то выводится сообщение «*Введены не все требуемые исходные данные*».

При выполнении заданий по параллельному программированию процедуры группы Get должны вызываться во всех процессах параллельной программы, в которых условием задания предусмотрен ввод исходных данных.

В варианте задачника для среды PascalABC.NET ввод данных можно выполнять с помощью стандартной процедуры Read, при этом один вызов процедуры можно использовать для ввода *нескольких* элементов исходных данных, например, `Read(a, b, c)`.

```
void PutN(int a);                                           C++
void PutD(double a);                                        C++
procedure PutN(a: integer);                              Pascal
procedure PutR(a: real);                                 Pascal
```

Процедуры обеспечивают вывод на экран результирующих данных, найденных программой, и их сравнение с контрольными данными (т. е. с правильным решением). Как и процедуры группы Get, эти процедуры должны вызываться после вызова процедуры Task; в противном случае при запуске программы будет выведено сообщение об ошибке «*В начале программы не вызвана процедура Task с именем задания*».

В отличие от процедур группы Get, в качестве параметра процедур группы Put можно указывать не только переменные, но и выражения (в частности, константы соответствующего типа). Используемая процедура должна соответствовать типу очередного элемента результирующих дан-

ных, в противном случае выводится сообщение об ошибке «*Неверно указан тип при выводе результатов*». Как и в случае процедур группы Get, при вызовах процедур группы Put программа осуществляет контроль за соответствием количества требуемых и выведенных результирующих данных. Если программа выведет недостаточное или избыточное количество результирующих данных, то после проверки этих данных появится сообщение «*Выведены не все результирующие данные*» или, соответственно, «*Попытка вывести лишние результирующие данные*».

При выполнении заданий по параллельному программированию процедуры группы Put должны вызываться во всех процессах параллельной программы, в которых условием задания предусмотрен вывод результатов.

В варианте задачника для среды PascalABC.NET вывод результатов можно выполнять с помощью стандартной процедуры Write, при этом один вызов процедуры можно использовать для вывода *нескольких* элементов результирующих данных, например, `Write(a, b, c)`.

`pt (поток ввода-вывода)` C++

Поток ввода-вывода pt может применяться в программах на языке C++ вместо процедур групп Get–Put. С его использованием операторы ввода-вывода могут быть оформлены более компактно. Например, вместо последовательности вызовов процедур `GetN(a); GetD(b); GetD(c);` достаточно указать один оператор чтения из потока: `pt >> a >> b >> c;`

2.3. Отладочные средства задачника

В версии 4.9 задачника Programming Taskbook появились средства, позволяющие выводить отладочную информацию непосредственно в окно задачника (в специальный *раздел отладки*). Необходимость в подобных дополнительных средствах возникает, прежде всего, при отладке параллельных программ, поскольку для них нельзя использовать такие стандартные средства отладки, как точки останова, пошаговое выполнение программы и окна просмотра значений переменных. Следует также отметить, что возможность вывода информации в раздел отладки позволяет использовать задачник для написания и отладки параллельных программ, не связанных с выполнением конкретных учебных заданий.

Отладочные средства задачника могут оказаться полезными и для обычных, непараллельных программ. В этом случае их можно применять в качестве дополнения к средствам встроенного отладчика.

Раздел отладки и его элементы

Раздел отладки представляет собой одну или несколько многострочных текстовых областей вывода. Он располагается под основными разделами задачника и выводится на экран только в случае, если в нем содержится какой-либо текст (рис. 1).

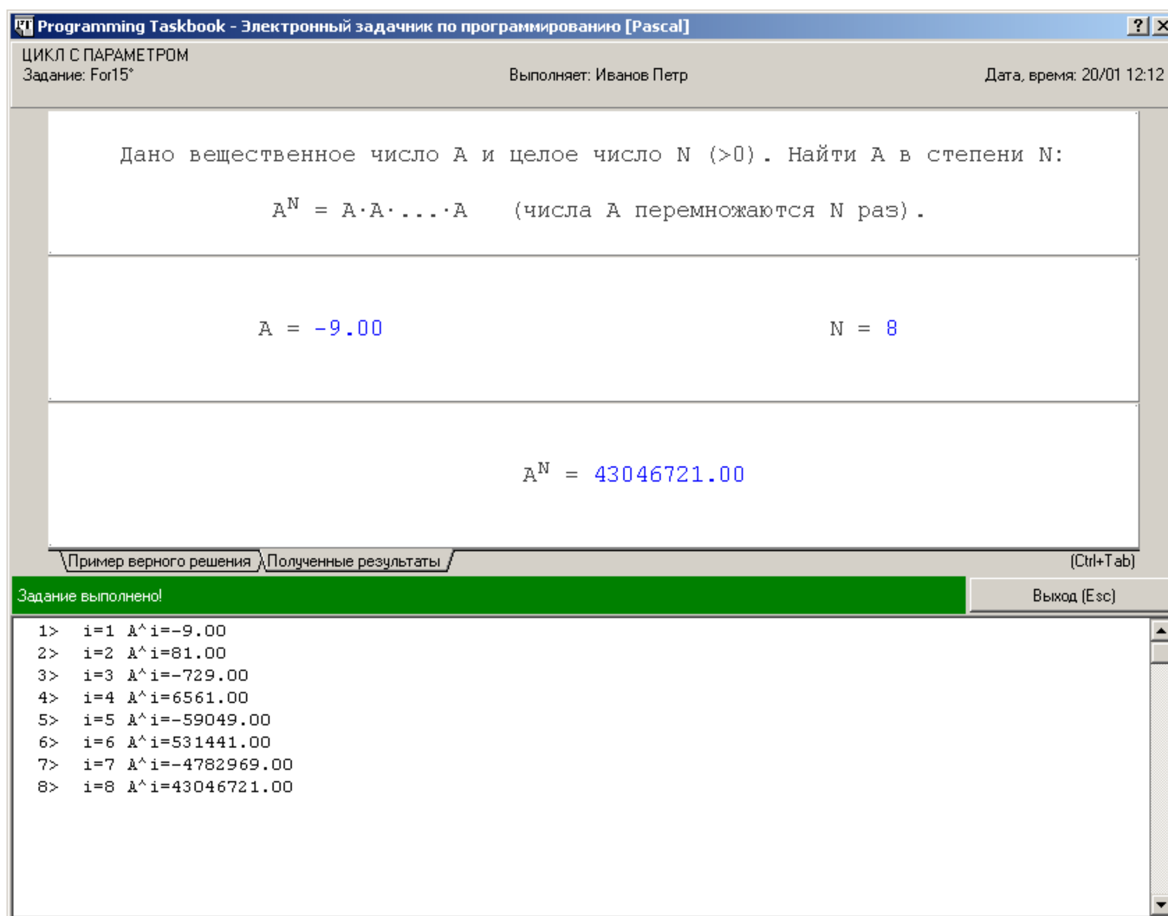

Рис. 1. Окно задачника с разделом отладки

При демонстрационном запуске программы все процедуры, связанные с разделом отладки, игнорируются, поэтому раздел отладки на экране не отображается.

Имеется возможность скрыть в окне задачника все его разделы, кроме раздела отладки; для этого достаточно нажать клавишу пробела. Повторное нажатие пробела восстанавливает в окне задачника ранее скрытые разделы. Для скрытия/отображения основных разделов окна задачника можно также использовать соответствующую команду контекстного меню, связанного с разделом отладки. Скрыть все разделы окна задачника, кроме раздела отладки, можно и программным способом, вызвав процедуру HideTask (см. ее описание в последнем подпункте данного пункта).

Прокрутка содержимого раздела отладки может осуществляться с помощью вертикальной полосы прокрутки, расположенной у его правой границы. Можно также использовать колесико мыши и клавиатурные комбинации [Alt]+[↑], [Alt]+[↓] (прокрутка на одну экранную строку), [Alt]+[PgUp], [Alt]+[PgDn] (прокрутка на 10 экранных строк), [Alt]+[Home], [Alt]+[End] (прокрутка к первой или последней строке области вывода).

Для «непараллельных» заданий раздел отладки содержит единственную область вывода. Для заданий по параллельному программированию число областей вывода равно числу параллельных процессов плюс 1; при

этом в каждый момент времени в разделе отладки отображается одна из областей. Если областей вывода больше одной, то в нижней части раздела отладки выводится набор ярлычков, позволяющих переключиться на любую из имеющихся областей вывода (рис. 2).

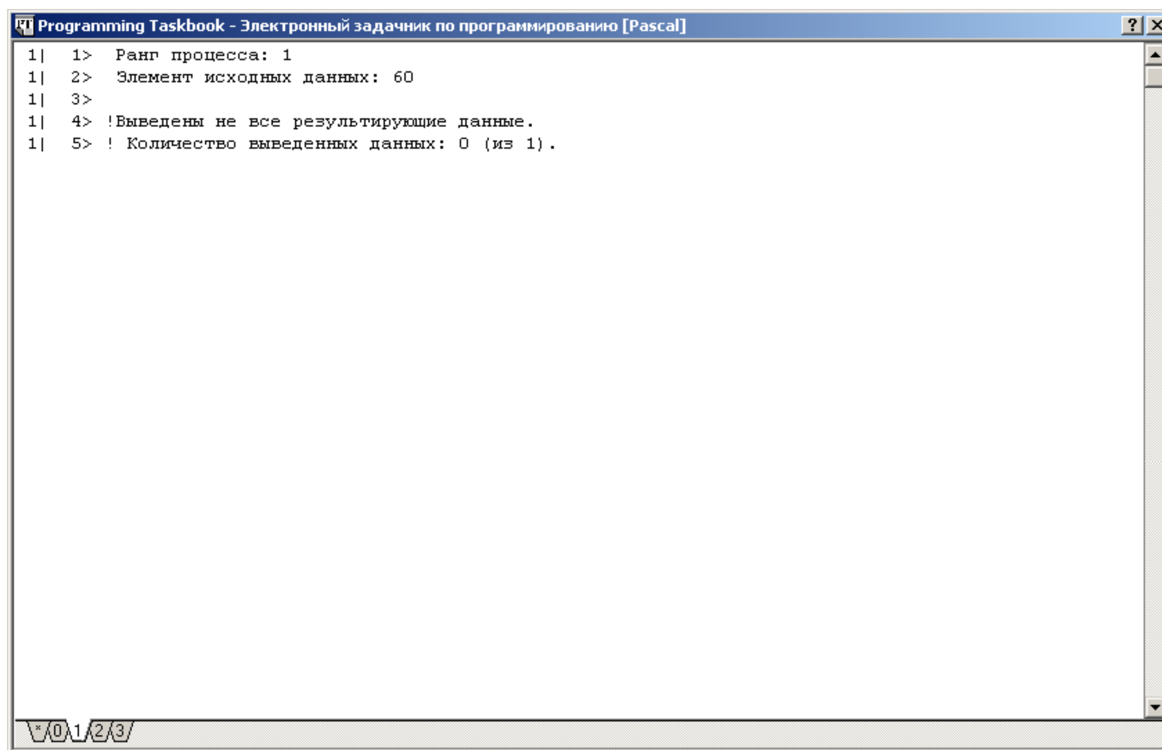

Рис. 2. Вывод отладочной информации для процесса ранга 1

Ярлычки с номерами (от 0 до N–1, где N — количество процессов) позволяют просмотреть содержимое области вывода, связанной с процессом соответствующего ранга; ярлычок с символом «*» позволяет просмотреть область вывода, содержащую объединенный текст всех других областей (рис. 3).

Следует заметить, что отладочная информация, получаемая из подчиненных процессов, предварительно сохраняется в специальных временных файлах в каталоге учащегося, поэтому она будет доступна для просмотра даже если на каком-либо этапе выполнения параллельной программы произойдет зависание некоторых подчиненных процессов. Отладочная информация, получаемая из главного процесса, выводится непосредственно в раздел отладки.

Количество отладочных строк для каждого процесса не должно превышать 999; если некоторый процесс пытается вывести данные в строку с номером, превышающим 999, то в связанной с этим процессом области отладки выводится сообщение об ошибке, и последующий вывод отладочных данных для этого процесса блокируется. Указанное ограничение позволяет, в частности, избежать проблем, возникающих при «бесконечном» выводе отладочной информации из какого-либо зациклившегося подчиненного процесса во временный файл.

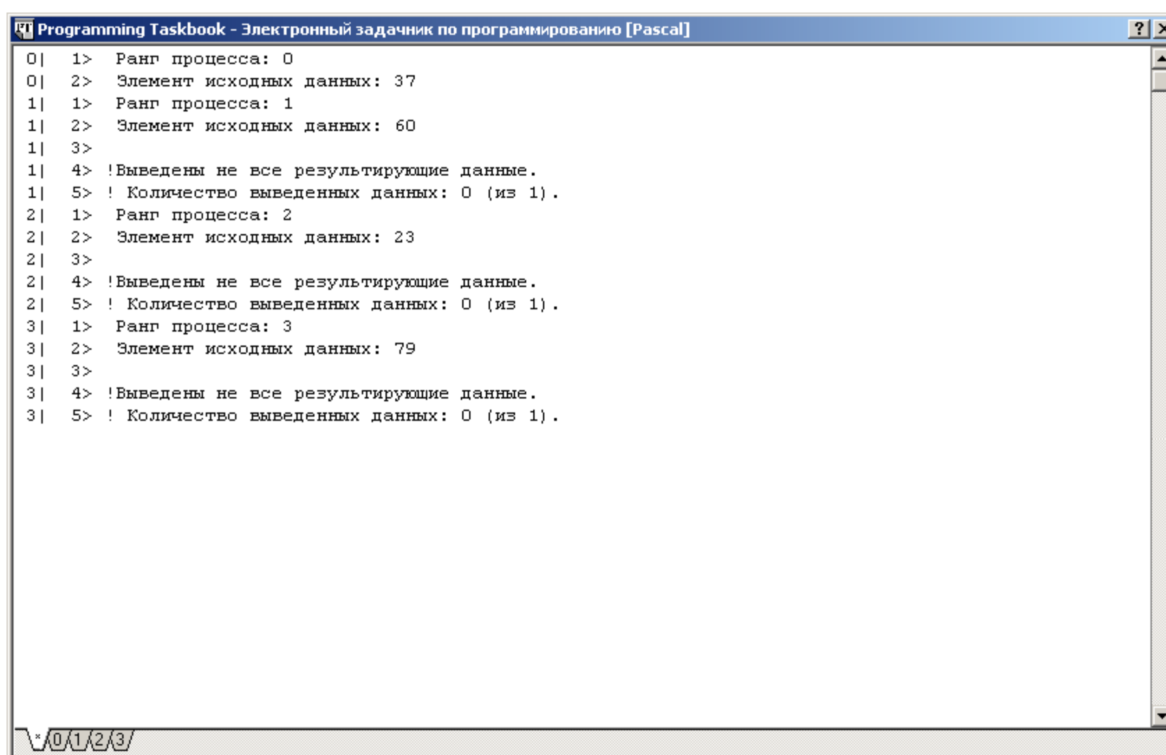

Рис. 3. Вывод отладочной информации для всех процессов параллельного приложения

Для переключения на нужную область вывода достаточно щелкнуть мышью на соответствующем ярлычке. Кроме того, для последовательного перебора ярлычков слева направо или справа налево можно использовать комбинации [Alt]+[→] и [Alt]+[←] соответственно (перебор осуществляется циклически). Можно также сразу перейти к нужной области вывода, нажав соответствующую клавишу: для области «*» — клавишу [*], для областей «0»–«9» — цифровые клавиши [0]–[9], а для областей «10»–«35» — буквенные клавиши от [A] до [Z] (напомним, что при выполнении заданий по параллельному программированию максимально возможное число процессов равно 36).

Предусмотрена возможность изменения размера шрифта, используемого в разделе отладки. Шрифт может изменяться от 7 до 14 пунктов с шагом 1. Для увеличения шрифта предназначена клавиатурная комбинация [Alt]+[+], для уменьшения — комбинация [Alt]+[–]. Соответствующие команды имеются также в контекстном меню раздела отладки. Информация о текущем размере шрифта сохраняется в файле результатов и учитывается при последующих запусках программы.

Если в основных разделах окна задачника отсутствуют прокручиваемые элементы или основные разделы являются скрытыми, то дополнительную клавишу [Alt] в перечисленных выше клавиатурных [Alt]-комбинациях можно не использовать.

Содержимое области вывода, отображаемой в разделе отладки, можно копировать в буфер Windows; для этого предназначена стандартная клавиатурная комбинация [Ctrl]+[C] и соответствующая команда контекстного меню раздела отладки.

Структура области вывода и простейший вариант процедуры Show

Каждая экранная строка, отображаемая в разделе отладки, состоит из служебной области и области данных. Ширина служебной области равна 6 экранным позициям для «непараллельных» заданий и 9 позициям для заданий по параллельному программированию. Ширина области данных равна 80 позициям.

Служебная область состоит из следующих частей (см. рисунки, приведенные в предыдущем подпункте):

- *область нумерации процесса* (только для заданий по параллельному программированию): 2 экранные позиции, отводимые для ранга процесса, и символ «|»;
- *область нумерации строк*: 3 экранные позиции, отводимые для номера строки данных, после которых следует символ «>» и символ пробела;
- *область признака сообщения об ошибке*: одна экранная позиция, в которой может содержаться либо пробел (признак обычного отладочного текста), либо символ «!» (признак сообщения об ошибке).

Если в разделе отладки выводится текст, связанный со всеми процессами параллельного приложения (этот текст связан с ярлычком «*»), то нумерация строк для каждого процесса производится независимо.

При переключении между областями вывода, связанными с различными процессами параллельного приложения, сохраняется номер первой отображаемой строки (за исключением ситуации, когда в новой области вывода отсутствует строка с требуемым номером; в этом случае в новой области вывод осуществляется, начиная с первой строки данных). Отмеченная особенность позволяет быстро просмотреть (и сравнить) один и тот же фрагмент отладочных данных для различных процессов.

Для вывода данных в раздел отладки предназначена процедура Show. Простейший ее вариант содержит единственный строковый параметр s (типа string для языка Паскаль и C++; кроме того, в языке C++ можно использовать параметр типа char*):

```
void Show(char* s);                                        C++
void Show(string s);                                       C++
procedure Show(s: string);                                 Pascal
```

Эта процедура выводит строку s в область данных раздела отладки.

Если текущая экранная строка уже содержит некоторый текст, то строка s снабжается начальным пробелом и приписывается к этому тексту, за исключением случая, когда при таком приписывании размер полученного текста превысит ширину области данных (равную 80 символам). В последнем случае вывод строки s осуществляется с начала следующей экранной строки; если же и в этой ситуации строка s превысит ширину области данных, то строка s будет выведена на нескольких экранных строках, причем разрывы текста будут выполняться по пробельным символам стро-

ки s, а при отсутствии пробелов — при достижении очередного фрагмента строки длины, равной 80.

Строка s может содержать явные команды перехода на новую экранную строку. В качестве таких команд можно использовать или символ с кодом 13 («возврат каретки» — символ `'\r'` на языке C++), или символ с кодом 10 («переход на новую строку» — символ `'\n'` на языке C++), или их комбинацию в указанном порядке (`#13#10` на языке Паскаль, `"\r\n"` на языке C++).

Имеется модификация процедуры Show — процедура ShowLine, которая после вывода строки s осуществляет автоматический переход на следующую экранную строку:

```
void ShowLine(char* s);                                    C++
void ShowLine(string s);                                   C++
procedure ShowLine(s: string);                             Pascal
```

Процедуру ShowLine можно вызывать без строкового параметра; в этом случае в разделе отладки будет просто выполнен переход на новую экранную строку:

```
void ShowLine();                                           C++
procedure ShowLine;                                        Pascal
```

Вывод числовой отладочной информации

В языках Паскаль и C++, помимо базовой процедуры Show с единственным строковым параметром, предусмотрены ее перегруженные варианты, предназначенные для вывода числовых отладочных данных. Использование этих вариантов позволяет максимально упростить действия учащегося, связанные с выводом числовых данных, поскольку избавляет его от необходимости применять стандартные средства языков Паскаль и C++, предназначенные для преобразования чисел в их строковые представления.

Имеется несколько перегруженных вариантов процедуры Show, предназначенных для вывода числовых данных. Ниже приведены полные варианты, содержащие наибольшее количество параметров (остальные варианты получаются из полных вариантов удалением параметра s, параметра w или обоих этих параметров):

```
void Show(char* s, int a, int w);                          C++
void Show(char* s, double a, int w);                       C++
void Show(string s, int a, int w);                         C++
void Show(string s, double a, int w);                      C++
procedure Show(s: string; a: integer; w: integer);         Pascal
procedure Show(s: string; a: real; w: integer);            Pascal
```

Строковый параметр s определяет необязательный комментарий, который указывается перед выводимым числом; если параметр s отсутствует, то комментарий полагается равным пустой строке.

Числовой параметр a определяет выводимое число.

Необязательный целочисленный параметр w определяет *ширину поля вывода* (т. е. количество экранных позиций, отводимое для вывода числа). Если указанной ширины w поля вывода недостаточно, то значение параметра w игнорируется; в этом случае (а также в случае, если параметр w отсутствует) используется ширина поля вывода, минимально необходимая для отображения данного числа. Если число не занимает всего поля вывода, то оно дополняется слева пробелами (т. е. выравнивается по *правой* границе поля вывода). В качестве десятичного разделителя для чисел с дробной частью используется точка. Вещественные числа по умолчанию выводятся в формате с фиксированной точкой и двумя дробными знаками. Изменить формат вывода вещественных чисел можно с помощью вспомогательной процедуры SetPrecision, описываемой в следующем подпункте.

Аналогичные перегруженные варианты предусмотрены и для процедуры ShowLine. В этих вариантах после вывода отладочных данных автоматически выполняется переход на следующую экранную строку.

Вспомогательные процедуры, связанные с выводом отладочной информации

```
void HideTask();                                              C++
procedure HideTask;                                           Pascal
```

Вызов данной процедуры обеспечивает скрытие всех разделов окна задачника, кроме раздела отладки. Если раздел отладки в окне задачника не отображается (в частности, если программа запущена в демонстрационном режиме), то вызов процедуры HideTask игнорируется.

Напомним, что скрыть/восстановить основные разделы окна задачника после его отображения на экране можно с помощью клавиши пробела или соответствующей команды контекстного меню раздела отладки.

```
void SetPrecision(int n);                                     C++
procedure SetPrecision(n: integer);                           Pascal
```

Процедура предназначена для настройки формата вывода вещественных отладочных данных. Если параметр n положителен, то он определяет количество выводимых дробных разрядов; при этом число выводится в формате с фиксированной точкой. Если параметр n равен нулю или является отрицательным, то число выводится в формате с плавающей точкой (экспоненциальном формате). В реализации для языка Паскаль число дробных знаков для экспоненциального формата определяется шириной поля вывода (т. е. параметром w процедуры Show — см. предыдущий подпункт). В реализации для языка C++ число дробных знаков полагается равным *модулю* параметра n; если же параметр n равен нулю, то по умолчанию устанавливается количество дробных знаков, равное 6.

Действие текущей настройки числового формата, определенной процедурой SetPrecision, продолжается до очередного вызова этой процедуры. До первого вызова процедуры SetPrecision вещественные числа выводятся в формате с фиксированной точкой и двумя дробными знаками.

Алфавитный указатель функций MPI

В указателе приведены все функции MPI, описанные в пособии. После имени функции указывается номер пункта, в котором содержится описание данной функции.

MPI_Allgather, 1.6
MPI_Allgatherv, 1.6
MPI_Allreduce, 1.6
MPI_Alltoall, 1.6
MPI_Alltoallv, 1.6
MPI_Barrier, 1.6
MPI_Bcast, 1.6
MPI_Bsend, 1.3
MPI_Bsend_init, 1.5
MPI_Buffer_attach, 1.3
MPI_Buffer_detach, 1.3
MPI_Cancel, 1.4
MPI_Cart_coords, 1.9
MPI_Cart_create, 1.9
MPI_Cart_get, 1.9
MPI_Cart_rank, 1.9
MPI_Cart_shift, 1.9
MPI_Cart_sub, 3.4, 1.9
MPI_Cartdim_get, 1.9
MPI_Comm_compare, 1.8
MPI_Comm_create, 1.8
MPI_Comm_dup, 1.8
MPI_Comm_free, 1.8
MPI_Comm_group, 1.8
MPI_Comm_rank, 1.2
MPI_Comm_size, 1.2
MPI_Comm_split, 1.8
MPI_Dims_create, 1.9
MPI_Finalize, 1.2
MPI_Gather, 1.6
MPI_Gatherv, 1.6
MPI_Get_count, 1.3
MPI_Get_processor_name, 1.2
MPI_Graph_create, 1.9
MPI_Graph_get, 1.9
MPI_Graph_neighbors, 1.9
MPI_Graph_neighbors_count, 1.9
MPI_Graphdims_get, 1.9
MPI_Group_compare, 1.8
MPI_Group_difference, 1.8
MPI_Group_excl, 1.8
MPI_Group_free, 1.8
MPI_Group_incl, 1.8
MPI_Group_intersection, 1.8
MPI_Group_rank, 1.8
MPI_Group_size, 1.8
MPI_Group_translate_ranks, 1.8
MPI_Group_union, 1.8
MPI_Ibsend, 1.4
MPI_Init, 1.2
MPI_Initialized, 1.2
MPI_Iprobe, 1.4
MPI_Irecv, 1.4
MPI_Irsend, 1.4
MPI_Isend, 1.4
MPI_Issend, 1.4
MPI_Op_create, 1.6
MPI_Op_free, 1.6
MPI_Pack, 1.7
MPI_Pack_size, 1.7
MPI_Probe, 1.3
MPI_Recv, 1.3
MPI_Recv_init, 1.5
MPI_Reduce, 1.6
MPI_Reduce_scatter, 1.6
MPI_Request_free, 1.4

MPI_Rsend, 1.3
MPI_Rsend_init, 1.5
MPI_Scan, 1.6
MPI_Scatter, 1.6
MPI_Scatterv, 1.6
MPI_Send, 1.3
MPI_Send_init, 1.5
MPI_Sendrecv, 1.5
MPI_Sendrecv_replace, 1.5
MPI_Ssend, 1.3
MPI_Ssend_init, 1.5
MPI_Start, 1.5
MPI_Startall, 1.5
MPI_Test, 1.4
MPI_Test_cancelled, 1.4
MPI_Testall, 1.4
MPI_Testany, 1.4
MPI_Testsome, 1.4
MPI_Topo_test, 1.9
MPI_Type_commit, 1.7
MPI_Type_contiguous, 1.7
MPI_Type_extent, 1.7
MPI_Type_free, 1.7
MPI_Type_hindexed, 1.7
MPI_Type_hvector, 1.7
MPI_Type_indexed, 1.7
MPI_Type_size, 1.7
MPI_Type_struct, 1.7
MPI_Type_vector, 1.7
MPI_Unpack, 1.7
MPI_Wait, 1.4
MPI_Waitall, 1.4
MPI_Waitany, 1.4
MPI_Waitsome, 1.4
MPI_Wtick, 1.2
MPI_Wtime, 1.2

Литература

1. Message Passing Interface Forum. MPI: A message-passing interface standard. International Journal of Supercomputer Applications, 8 (3/4), 1994. Special issue on MPI.

Содержание

1. Основные типы, константы и функции библиотеки MPI 2
 1.1. Типы и константы библиотеки MPI 2
 1.2. Функции общего назначения 6
 1.3. Блокирующая пересылка сообщений.. 7
 1.4. Неблокирующая пересылка сообщений 11
 1.5. Отложенные и совмещенные запросы на взаимодействие............. 17
 1.6. Коллективное взаимодействие процессов 20
 1.7. Определение пользовательских типов и упаковка данных 28
 1.8. Работа с группами процессов и коммуникаторами 32
 1.9. Виртуальные топологии 36
2. Электронный задачник Programming Taskbook for MPI 41
 2.1. Общее описание задачника PT for MPI... 41
 2.2. Средства задачника для инициализации заданий
 и ввода-вывода данных 43
 2.3. Отладочные средства задачника 46
Алфавитный указатель функций MPI 53
Литература 55