Программирование в MPI

Коммуникационная библиотека MPI является общепризнанным стандартом в параллельном программировании с использованием механизма передачи сообщений. MPI-программа представляет собой набор независимых процессов, каждый из которых выполняет одинаковую программу написанную на языке C или FORTRAN. Процессы MPI-программы взаимодействуют друг с другом посредством вызова коммуникационных процедур. Каждый процесс выполняется в своем собственном адресном пространстве, однако это может быть как последовательный процесс, так и многопотоковый (гибридное программирование MPI + OpenMP). MPI не предоставляет никаких средств для распределения процессов по вычислительным узлам и для запуска их на исполнение. Эти функции возлагаются на операционную систему. В частности, на кластерах это специальный командный файл (скрипт) mpirun, который предполагает, что исполнимые модули уже каким-то образом распределены по компьютерам кластера.

Терминология и обозначения

Номер процесса - целое неотрицательное число, являющееся уникальным атрибутом каждого процесса.

Атрибуты сообщения - номер процесса-отправителя, номер процесса-получателя и идентификатор сообщения. Для них заведена структура MPI_Status, содержащая три поля:
MPI_Source (номер процесса отправителя),
MPI_Tag (идентификатор сообщения, тэг), MPI_Error (код ошибки); могут быть и добавочные поля.

Идентификатор сообщения (тэг) - атрибут сообщения, являющийся целым неотрицательным числом, лежащим в диапазоне от 0 до 32767.
Для идентификации наборов процессов вводится понятие группы, объединяющей все или какую-то часть процессов. Каждая группа образует область связи, с которой связывается специальный объект - коммуникатор области связи. Процессы внутри группы нумеруются целым числом в диапазоне 0..groupsize-1. Все коммуникационные операции с некоторым коммуникатором будут выполняться только внутри области связи, описываемой этим коммуникатором. При инициализации MPI создается предопределенная область связи, содержащая все процессы MPI-программы, с которой связывается предопределенный коммуникатор MPI_COMM_WORLD, объединяющий все процессоры в одну группу. Поэтому термины процесс и процессор становятся синонимами, а величина groupsize становится равной NPROCS - числу процессоров, выделенных задаче.

Итак, если сформулировать коротко, MPI - это библиотека функций, обеспечивающая взаимодействие параллельных процессов с помощью механизма передачи со-

общений. Это достаточно объемная и сложная библиотека, состоящая примерно из 130 функций, в число которых входят:

функции инициализации и закрытия MPI процессов;
функции, реализующие коммуникационные операции типа точка-точка;
функции, реализующие коллективные операции;
функции для работы с группами процессов и коммуникаторами;
функции для работы со структурами данных;
функции формирования топологии процессов.

В принципе, любая параллельная программа может быть написана с использованием всего 6 MPI функций, а достаточно полную и удобную среду программирования составляет набор из 24 функций.

Каждая из MPI функций характеризуется способом выполнения:

Локальная функция - выполняется внутри вызывающего процесса. Ее завершение не требует коммуникаций.

Нелокальная функция - для ее завершения требуется выполнение MPI процедуры другим процессом.

Глобальная функция - процедуру должны выполнять все процессы группы. Несоблюдение этого условия может приводить к зависанию задачи.

Блокирующая функция - возврат управления из процедуры гарантирует возможность повторного использования параметров, участвующих в вызове. Никаких изменений в состоянии процесса, вызвавшего блокирующий запрос, до выхода из процедуры не может происходить.

Неблокирующая функция - возврат из процедуры происходит немедленно, без ожидания окончания операции и до того, как будет разрешено повторное использование параметров, участвующих в запросе. Завершение неблокирующих операций осуществляется специальными функциями.

В языке С все процедуры являются функциями, и большинство из них возвращает код ошибки. При использовании имен подпрограмм и именованных констант необходимо строго соблюдать регистр символов. Массивы индексируются с 0. Логические переменные представляются типом int (true соответствует 1, a false - 0). Определение всех именованных констант, прототипов функций и определение типов выполняется подключением файла mpi.h. Введение собственных типов в MPI было продиктовано тем обстоятельством, что стандартные типы языков на разных платформах имеют различное представление. MPI допускает возможность запуска процессов параллельной программы на компьютерах различных платформ, обеспечивая при этом автоматическое преобразование данных при пересылках.

В таблице приведено соответствие предопределенных в MPI типов стандартным типам языка C.

тип MPI	тип языка C
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_UNSIGNED_CHAR	unsigned char
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_BYTE	
MPI_PACKED	

БАЗОВЫЕ ФУНКЦИИ MPI

Любая прикладная MPI-программа (приложение) должна начинаться с вызова функции инициализации MPI (функция MPI_Init). В результате выполнения этой функции создается группа процессов, в которую помещаются все процессы приложения, и создается область связи, описываемая предопределенным коммуникатором MPI_COMM_WORLD. Эта область связи объединяет все процессы приложения. Процессы в группе упорядочены и пронумерованы от 0 до groupsize-1, где groupsize равно числу процессов в группе.

Локальные функции

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

Каждому процессу при инициализации передаются аргументы функции main, полученные из командной строки.

Функция завершения MPI программ MPI_Finalize.

```
int MPI_Finalize(void)
```

Функция закрывает все MPI-процессы и ликвидирует все области связи.

Функция определения числа процессов в области связи MPI_Comm_size.

int MPI_Comm_size(MPI_COMM_WORLD, int *size)

size - число процессов.

Функция возвращает количество процессов в области связи коммуникатора MPI_COMM_WORLD.

Функция определения номера процесса MPI_Comm_rank.

int MPI_Comm_rank(MPI_COMM_WORLD, int *rank)

rank - номер процесса, вызвавщего функцию.

Функция возвращает номер процесса, вызвавшего эту функцию. Номера процессов лежат в диапазоне 0..size-1 (значение size может быть определено с помощью предыдущей функции).

Функция отсчета времени (таймер) MPI_Wtime.

double MPI_Wtime(void)

Функция возвращает астрономическое время в секундах, прошедшее с некоторого момента в прошлом (точки отсчета). Гарантируется, что эта точка отсчета не будет изменена в течение жизни процесса. Для хронометрирования участка программы вызов функции делается в начале и конце участка и определяется разница между показаниями таймера.

```
{
 double starttime, endtime;
 starttime = MPI_Wtime();
 ... хронометрируемый участок ...
 endtime = MPI_Wtime();
printf("Выполнение заняло %f секунд\n", endtime-starttime);
}
```

ОБЗОР КОММУНИКАЦИОННЫХ ОПЕРАЦИЙ ТИПА ТОЧКА-ТОЧКА

В коммуникационных операциях типа точка-точка всегда участвуют не более двух процессов: передающий и принимающий. В MPI имеется множество функций, реализующих такой тип обменов. Многообразие объясняется возможностью организации таких обменов множеством способов.

Блокирующие функции подразумевают полное окончание операции после выхода из процедуры, т.е. вызывающий процесс блокируется, пока операция не будет завершена. Для функции посылки сообщения это означает, что все пересылаемые данные помещены в буфер (для разных реализаций MPI это может быть либо какой-то промежуточный системный буфер, либо непосредственно буфер получателя). Для функции приема сообщения блокируется выполнение других операций, пока все данные из буфера не будут помещены в адресное пространство принимающего процесса.

Неблокирующие функции подразумевают совмещение операций обмена с другими операциями, поэтому неблокирующие функции передачи и приема по сути дела являются функциями инициализации соответствующих операций. Для опроса завершенности операции (и завершения) вводятся дополнительные функции.

Блокирующие коммуникационные операции

Функция передачи сообщения MPI_Send.

int **MPI_Send**(void* buf, int count, MPI_Datatype datatype, int dest, int tag,

MPI_COMM_WORLD)

buf - адрес начала расположения пересылаемых данных;
count - число пересылаемых элементов;
datatype - тип посылаемых элементов;
dest - номер процесса-получателя в группе
tag - идентификатор сообщения;

Функция выполняет посылку count элементов типа datatype сообщения с идентификатором tag процессу dest. Переменная buf - это, как правило, массив или скалярная переменная. В последнем случае значение count = 1.

Функция приема сообщения MPI_Recv.

int **MPI_Recv**(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_COMM_WORLD, MPI_Status *status)

buf - адрес начала расположения принимаемого сообщения;
count - максимальное число принимаемых элементов;
datatype - тип элементов принимаемого сообщения;
source - номер процесса-отправителя;
tag - идентификатор сообщения;
status - атрибуты принятого сообщения.

В стандартном режиме выполнение операции обмена включает три этапа:

Передающая сторона формирует пакет сообщения, в который помимо передаваемой информации упаковываются адрес отправителя (source), адрес получателя (dest), идентификатор сообщения (tag). Этот пакет передается отправителем в системный буфер, и на этом функция посылки сообщения заканчивается.
Сообщение системными средствами передается адресату.

Принимающий процессор извлекает сообщение из системного буфера, когда у него появится потребность в этих данных. Содержательная часть сообщения помещается в адресное пространство принимающего процесса (параметр buf), а служебная в параметр status.

Поскольку операция выполняется в асинхронном режиме, адресная часть принятого сообщения состоит из трех полей:
коммуникатора (comm), поскольку каждый процесс может одновременно входить в несколько областей связи;
номера отправителя в этой области связи (source);
идентификатора сообщения (tag), который используется для взаимной привязки конкретной пары операций посылки и приема сообщений.

Число элементов в принимаемом сообщении не должно превосходить значения count. Если число принятых элементов меньше значения count, то гарантируется, что в буфере buf изменятся только элементы, соответствующие элементам принятого сообщения. Блокировка гарантирует, что после возврата из подпрограммы все элементы сообщения приняты и расположены в буфере buf.

Таким образом, после чтения сообщения некоторые параметры могут оказаться неизвестными, а именно: число считанных элементов, идентификатор сообщения и адрес отправителя. Эту информацию можно получить с помощью параметра status. Переменные status должны быть явно объявлены в MPI программе.

В языке C status - это структура типа MPI_Status с тремя полями MPI_SOURCE, MPI_TAG, MPI_ERROR. Назначение полей переменной status представлено в таблице.

Поля status	
Процесс-отправитель	status.MPI_SOURCE
Идентификатора сообщения	status.MPI_TAG
Код ошибки	status.MPI_ERROR

Как видно из таблицы, количество считанных элементов в переменную status не заносится. Для определения числа фактически полученных элементов сообщения необходимо использовать специальную функцию MPI_Get_count:

int **MPI_Get_count** (MPI_Status *status, MPI_Datatype datatype, int *count)

status - атрибуты принятого сообщения;
datatype - тип элементов принятого сообщения;
count - число полученных элементов.

Подпрограмма MPI_Get_count может быть вызвана после чтения сообщения (функциями MPI_Recv, MPI_Irecv).

Блокированная передача могла бы завершаться, как только сообщение буферизовалось в системном буфере, даже если нет соответствующего получателя. Буферизация сообщения "развязывает" посылающее и получающее действия. С другой стороны, буферизация сообщения может быть дорога, поскольку это влечет за собой дополнительное копирование "память-память" и требуется распределение памяти для буферов. MPI предлагает выбор разных версий блокированной связи (буферизированный, синхронный, по готовности), которые допускают, чтобы пользователи управляли выбором протокола связи.

В хорошо построенных программах применение блокированной передачи приводит к полезному эффекту регулятора. Рассмотрим случай, когда производитель неоднократно производит новые значения и посылает их потребителю. Допустим, что производитель производит новые значения быстрее, чем потребитель может потреблять их. Если используются блокированная посылка, то производитель будет автоматически регулироваться, когда пространство системного буфера недоступно. В плохо построенных программах блокированная передача может приводить к ситуации deadlock, где все процессы заблокированы. Такие программы могут нормально закончить, когда достаточное пространство буфера доступно, но будут терпеть неудачу на системах, которые делают меньшее количество буферизации или когда наборы данных (и размеры сообщения) увеличены. Так как любая система исчерпает буферные ресурсы, если размеры сообщения увеличены, и некоторые выполнения могут испытывать недостаток в буферизации, MPI рекомендует безопасным программам не полагаться на буферизацию системы.

MPI не предписывает безопасный тип программирования. Пользователи свободны в выборе разных типов обменных функций.
Следующие примеры иллюстрируют безопасные фрагменты программ.
Пример 1. Обмен сообщениями.

```
MPI_Comm_rank(comm,  ftrank); if(rank == 0)
{ MPI_Send(sendbuf,  count,  MPI_FLOAT,   1,  tag,   comm);
MPI_Recv(recvbuf,  count,  MPI_FLOAT,   1,  tag,   comm,  &status);
else  if(rank ==1)
{ MPI_Recv(recvbuf,  count,  MPI_FLOAT,  0,  tag,   comm, &status);
MPI_Send(sendbuf,  count,  MPI.FLOAT,  0,  tag,comm);
```

Эта программа безопасна и будет всегда завершаться правильно, даже если никакое пространство буфера не доступно для данных.

Пример 2. Попытка обмена сообщениями.

```
MPI_Comm_rank(comm, &rank); if(rank == 0)
{ MPI_Recv(recvbuf, count, MPI.FLOAT, 1, tag, comm, &status); MPI_Send(sendbuf,
count, MPI_FL0AT, 1, tag, comm);
else if(rank = = 1)
{ MPI_Recv(recvbuf, count, MPI.FLOAT, 0, tag, comm, &status);
MPI_Send(sendbuf, count, MPI.FLOAT, 0, tag, comm);
```

Получающая функция нулевого процесса должна вначале принять данные от первого процесса, затем уже послать ему данные и может завершиться, только если первый процесс выполнит посылку данных нулевому. Аналогично, первый процесс вначале ждет данных от нулевого и только после этого пошлет ему данные. Эта программа неправильная и будет находиться в состоянии deadlock.

Пример 3. Обмен, который полагается на буферизацию.

```
MPI_Comm_rank(comm, &rank);
if(rank == 0)
MPI_Send(sendbuf, count, MPI_FLOAT, 1, tag, comm);
MPI_Recv(recvbuf, count, MPI_FLOAT, 1, tag, comm, &status);
else if(rank = = 1)
{ MPI_Send(sendbuf, count, MPI_FLOAT, 0, tag, comm);
MPI_Recv(recvbuf, count, MPI_FLOAT, 0, tag, comm, &status);
```

Сообщение, посланное каждым процессом, должно копироваться где-нибудь, чтобы посылающие функции завершились и запустились получающие функции. Для завершения программы необходимо что бы, по крайней мере, одно из двух сообщений буферизовалось. Таким образом, эта программа преуспеет, только если система связи будет буферизировать в своем внутреннем буфере, иначе зайдет в состояние deadlock. Успех ее будет зависеть от количества пространства буфера, доступного в частном выполнении. С небольшим массивом данных эта программа выполнится правильно, но она ненадежна.

Неблокирующие коммуникационные операции

Использование неблокирующих коммуникационных операций более безопасно с точки зрения возникновения тупиковых ситуаций, а также может увеличить скорость работы программы за счет совмещения выполнения вычислительных и коммуникационных операций. Эти задачи решаются разделением коммуникационных операций на две стадии: инициирование операции и проверку завершения операции.

Неблокирующие операции используют специальный скрытый объект "запрос обмена" (request) для связи между функциями обмена и функциями опроса их завершения. Для прикладных программ доступ к этому объекту возможен только через вызовы MPI функций. Если операция обмена завершена, подпрограмма проверки снимает "запрос обмена", устанавливая его в значение MPI_REQUEST_NULL. Снять запрос без ожидания завершения операции можно подпрограммой MPI_Request_free.

Функция передачи сообщения без блокировки MPI_Isend.

int **MPI_Isend**(void* buf, int count, MPI_Datatype datatype, int dest, int tag,

MPI_COMM_WORLD, MPI_Request *request)

buf	- адрес начала расположения передаваемых данных;
count	- число посылаемых элементов;
datatype	- тип посылаемых элементов;
dest	- номер процесса-получателя;
tag	- идентификатор сообщения;
request	- "запрос обмена".

Возврат из подпрограммы происходит немедленно, без ожидания окончания передачи данных. Этим объясняется префикс I в именах функций. Поэтому переменную buf повторно использовать нельзя до тех пор, пока не будет погашен "запрос обмена". Это можно сделать с помощью подпрограмм MPI_Wait или MPI_Test, передав им параметр request.

Функция приема сообщения без блокировки MPI_Irecv.

int **MPI_Irecv**(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_COMM_WORLD, MPI_Request *request)

buf	- адрес для принимаемых данных;
count	- максимальное число принимаемых элементов;
datatype	- тип элементов принимаемого сообщения;
source	- номер процесса-отправителя;
tag	- идентификатор сообщения;
request	- "запрос обмена".

Возврат из подпрограммы происходит немедленно, без ожидания окончания приема данных. Определить момент окончания приема можно с помощью подпрограмм MPI_Wait или MPI_Test с соответствующим параметром request.

Воспользоваться результатом неблокирующей коммуникационной операции или повторно использовать ее параметры можно только после ее полного завершения. Имеется два типа функций завершения неблокирующих операций:

Операции ожидания завершения семейства WAIT блокируют работу процесса до полного завершения операции.

Операции проверки завершения семейства TEST возвращают значения TRUE или FALSE в зависимости от того, завершилась операция или нет. Они не блокируют работу процесса и полезны для предварительного определения факта завершения операции.

Функция ожидания завершения неблокирующей операции MPI_Wait.

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

request - запрос связи;
status - атрибуты сообщения.

Это нелокальная блокирующая операция. Возврат происходит после завершения операции, связанной с запросом request. В параметре status возвращается информация о законченной операции.

Функция проверки завершения неблокирующей операции MPI_Test.

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

request - запрос связи;
flag - признак завершенности проверяемой операции;
status - атрибуты сообщения, если операция завершилась.

Это локальная неблокирующая операция. Если связанная с запросом request операция завершена, возвращается flag = true, а status содержит информацию о завершенной операции. Если проверяемая операция не завершена, возвращается flag = false, а значение status в этом случае не определено.

Функция снятия запроса без ожидания завершения неблокирующей операции MPI_Request_free.

int **MPI_Request_free**(MPI_Request *request)

request - запрос связи.

Параметр request устанавливается в значение MPI_PROC_NULL. Связанная с этим запросом операция не прерывается, однако проверить ее завершение с помощью

MPI_Wait или MPI_Test уже нельзя. Для прерывания коммуникационной операции следует использовать функцию

MPI_Cancel(MPI_Request *request).

Объединенные функции передачи-приема данных

В MPI введены две специальной функции, объединяющие в одном запросе посылающее и получающее действия. Такие функции полезны для моделей связи, где каждый процесс и посылает, и получает сообщения.

int **MPI_Sendrecv**(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, MPI_Datatype recvtag, MPI_Comm comm, MPI_Status *status)

sendbuf	адрес посылаемого буфера
sendcount	количество посылаемых элементов
sendtype	тип элементов в посылаемом буфере
dest	ранг (номер) процесса, которому осуществляется передача
sendtag	тег посылаемого сообщения
recvbuf	буфер для приема данных
recvcount	максимальное количество принимаемых элементов
recvtype	тип принимаемых элементов
source	ранг (номер) передающего процесса
recvtag	тег принимаемых данных
comm	имя переключателя каналов (communicator)
status	статус полученного сообщения

Посылающее и получающее действия используют тот же самый переключатель каналов, но имеют различные аргументы тегов. Посылающийся буфер и буфер для приема не должны пересекаться и могут иметь различные длины и типы данных. Следующая функция аналогична предыдущей, но у нее посылаемый буфер совпадает с буфером для получения данных.

int **MPI_Sendrecv_replace**(void*buf, int count, MPI_Datatype datatype,

int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)

buf	адрес посылаемого буфера и буфера для приема данных
count	количество элементов в посылаемом буфере и буфере для приема
datatype	тип элементов в посылаемом буфере и буфере для приема
dest	ранг (номер) процесса, которому осуществляется передача

sendtag тег посылаемого сообщения
source ранг (номер) передающего процесса
recvtag тег принимаемых данных
comm имя переключателя каналов (communicator)
status статус полученного сообщения

MPI_Sendrecv_replace выполняется с блокированием посылки и получения. Используется тот же самый буфер, как для посылающего, так и для получающего оператора. Посланное сообщение затем заменяется полученным сообщением.

Пример применения блокированных операций «точка-точка»

```
// Простая передача-прием в MPI:  MPI_Send, MPI_Recv для двух процессов
  #include <mpi.h>
  #include <stdio.h>
int main(int argc, char **argv)
   { int size, rank, i, count;  double Data[10];  MPI_Status status;
     MPI_Init(&argc, &argv);  // Инициализация библиотеки MPI
  // Каждая ветвь узнает количество задач в стартовавшем приложении
// и свой собственный номер: от 0 до (size-1)
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// Пользователь должен запустить ровно две задачи, иначе ошибка
     if(size != 2) {
if(rank == 0) printf("Error: two processes required instead of %d, abort\n",  size);
       // Все ветви в области связи MPI_COMM_WORLD будут стоять,
      //пока ветвь 0 не выведет сообщение.
        MPI_Barrier(MPI_COMM_WORLD);
      /* Без точки синхронизации может оказаться, что одна из ветвей
      вызовет MPI_Abort раньше, чем успеет отработать printf()  в ветви 0,
      MPI_Abort немедленно принудительно завершит  все ветви и сообщение
      выведено не будет. Все задачи аварийно завершают работу */
        MPI_Abort( MPI_COMM_WORLD,  MPI_ERR_OTHER);
        return -1;
       }  if(rank == 0) { for(i=0;i<10;i++)Data[i]=i;       // Передача из ветви 0 в ветвь 1
        MPI_Send( Data,  5, MPI_DOUBLE, 1, 1,MPI_COMM_WORLD);
       } else {
      // Ветвь 1 принимает  данные от ветви 0,
      // дожидается сообщения и помещает пришедшие данные в буфер
     MPI_Recv( Data,10, MP _DOUBLE, 0, 1,  MPI_COMM_WORLD, &status);
       // Вычисляем фактически принятое количество данных
        MPI_Get_count( &status,  MPI_DOUBLE, &count);
       for(i=0;i<count;i++)if(Data[i]!=i){printf("Error 1:i=%3d\n",i);
}
```

```
 // Выводим фактическую длину принятого на экран
    printf("rank = %d Received %d elems\n", rank, count);
  // Обе ветви завершают выполнение */
MPI_Finalize();  return 0; }
```

Обзор коллективных операций

Набор операций типа точка-точка является достаточным для программирования любых алгоритмов, однако MPI вряд ли бы завоевал такую популярность, если бы ограничивался только этим набором коммуникационных операций. Одной из наиболее привлекательных сторон MPI является наличие широкого набора коллективных операций, которые берут на себя выполнение наиболее часто встречающихся при программировании действий. Например, часто возникает потребность разослать некоторую переменную или массив из одного процессора всем остальным. Каждый программист может написать такую процедуру с использованием операций Send/Recv, однако гораздо удобнее воспользоваться коллективной операцией MPI_Bcast. Причем гарантировано, что эта операция будет выполняться гораздо эффективнее, поскольку MPI-функция реализована с использованием внутренних возможностей коммуникационной среды. Главное отличие коллективных операций от операций типа точка-точка состоит в том, что в них всегда участвуют все процессы, связанные с коммуникатором. Несоблюдение этого правила приводит либо к аварийному завершению задачи, либо к еще более неприятному зависанию задачи.

Набор коллективных операций включает:

Синхронизацию всех процессов с помощью барьеров (MPI_Barrier);
Коллективные коммуникационные операции, в число которых входят:

- рассылка информации от одного процесса всем остальным членам некоторой области связи (MPI_Bcast);
- сборка (gather) распределенного по процессам массива в один массив с сохранением его в адресном пространстве выделенного (root) процесса (MPI_Gather, MPI_Gatherv)
- сборка (gather) распределенного массива в один массив с рассылкой его всем процессам некоторой области связи (MPI_Allgather, MPI_Allgatherv)
- разбиение массива и рассылка его фрагментов (scatter) всем процессам области связи (MPI_Scatter, MPI_Scatterv);
- совмещенная операция Scatter/Gather (All-to-All), каждый процесс делит данные из своего буфера передачи и разбрасывает фрагменты всем остальным процессам, одновременно собирая фрагменты, посланные другими процессами в свой буфер приема (MPI_Alltoall, MPI_Alltoallv).

Глобальные вычислительные операции (sum, min, max и др.) над данными, расположенными в адресных пространствах различных процессов.

Все коммуникационные подпрограммы, за исключением MPI_Bcast, представлены в двух вариантах:

- простой вариант, когда все части передаваемого сообщения имеют одинаковую длину и занимают смежные области в адресном пространстве процессов;
- "векторный" вариант предоставляет более широкие возможности по организации коллективных коммуникаций, снимая ограничения, присущие простому варианту, как в части длин блоков, так и в части размещения данных в адресном пространстве процессов. Векторные варианты отличаются дополнительным символом "v" в конце имени функции.

Отличительные особенности коллективных операций:

Коллективные коммуникации не взаимодействуют с коммуникациями типа точка-точка.
Коллективные коммуникации выполняются в режиме с блокировкой. Возврат из подпрограммы в каждом процессе происходит тогда, когда его участие в коллективной операции завершилось, однако это не означает, что другие процессы завершили операцию.
Количество получаемых данных должно быть равно количеству посланных данных.
Типы элементов посылаемых и получаемых сообщений должны совпадать.
Сообщения не имеют идентификаторов.

Изучение коллективных операций начнем с рассмотрения двух функций, стоящих особняком: MPI_Barrier и MPI_Bcast.

Функция синхронизации процессов MPI_Barrier блокирует работу вызвавшего ее процесса до тех пор, пока все другие процессы группы также не вызовут эту функцию. Завершение работы этой функции возможно только всеми процессами одновременно (все процессы "преодолевают барьер" одновременно).

int **MPI_Barrier**(MPI_COMM_WORLD)

Синхронизация с помощью барьеров используется, например, для завершения всеми процессами некоторого этапа решения задачи, результаты которого будут использоваться на следующем этапе. Использование барьера гарантирует, что ни один из процессов не приступит раньше времени к выполнению следующего этапа, пока результат работы предыдущего не будет окончательно сформирован. Неявную синхронизацию процессов выполняет любая коллективная функция.

Широковещательная рассылка данных выполняется с помощью функции MPI_Bcast. Процесс с номером root рассылает сообщение из своего буфера передачи всем процессам области связи коммуникатора comm.

int **MPI_Bcast**(void* buffer, int count, MPI_Datatype datatype, int root, MPI_COMM_WORLD)

buffer	- адрес начала расположения в памяти рассылаемых данных;
count	- число посылаемых элементов;
datatype	- тип посылаемых элементов;
root	- номер процесса-отправителя;

После завершения подпрограммы каждый процесс, включая и самого отправителя, получит копию сообщения от процесса-отправителя root.

Функции сбора блоков данных от всех процессов группы

Семейство функций сбора блоков данных от всех процессов группы состоит из четырех подпрограмм: MPI_Gather, MPI_Allgather, MPI_Gatherv, MPI_Allgatherv. Каждая из указанных подпрограмм расширяет функциональные возможности предыдущих.

Функция MP_Gather производит сборку блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером root. Длина блоков предполагается одинаковой. Объединение происходит в порядке увеличения номеров процессов-отправителей. То есть данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf процесса root. Длина массива, в который собираются данные, должна быть достаточной для их размещения.

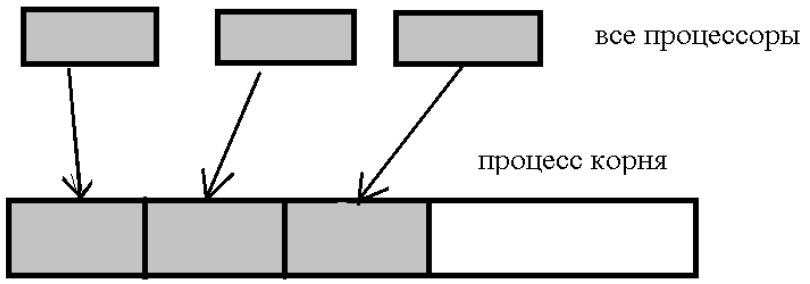

Схема работы функции MPI_Gather

int **MPI_Gather**(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype,
int root, MPI_COMM_WORLD)

sendbuf - адрес начала размещения посылаемых данных;
sendcount - число посылаемых элементов;
sendtype - тип посылаемых элементов;
recvbuf - адрес начала буфера приема (используется только в процессе-получателе root);
recvcount - число элементов, получаемых от каждого процесса (используется только в процессе-получателе root);
recvtype - тип получаемых элементов;
root - номер процесса-получателя;

Тип посылаемых элементов sendtype должен совпадать с типом recvtype получаемых элементов, а число sendcount должно равняться числу recvcount. То есть, recvcount в вызове из процесса root - это число собираемых от каждого процесса элементов, а не общее количество собранных элементов.

Функция MPI_Allgather выполняется так же, как MPI_Gather, но получателями являются все процессы группы. Данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf каждого процесса. После завершения операции содержимое буферов приема recvbuf у всех процессов одинаково.
int **MPI_Allgather**(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_COMM_WORLD)

sendbuf - адрес начала буфера посылки;
sendcount - число посылаемых элементов;
sendtype - тип посылаемых элементов;
recvbuf - адрес начала буфера приема;
recvcount - число элементов, получаемых от каждого процесса;
recvtype - тип получаемых элементов;

Функции распределения блоков данных по всем процессам группы

Семейство функций распределения блоков данных по всем процессам группы состоит из двух подпрограмм: MPI_Scatter и MPI_Scaterv.

Функция MPI_Scatter разбивает сообщение из буфера посылки процесса root на равные части размером sendcount и посылает i-ю часть в буфер приема процесса с номером i (в том числе и самому себе). Процесс root использует оба буфера (посылки и приема), поэтому в вызываемой им подпрограмме все параметры являются существенными. Остальные процессы группы являются только получателями, поэтому для них параметры, специфицирующие буфер посылки, не существенны.

int **MPI_Scatter**(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)

sendbuf - адрес начала размещения блоков распределяемых данных (используется только в процессе-отправителе root);
sendcount - число элементов, посылаемых каждому процессу;
sendtype - тип посылаемых элементов;
recvbuf - адрес начала буфера приема;
recvcount - число получаемых элементов;
recvtype - тип получаемых элементов;
root - номер процесса-отправителя;

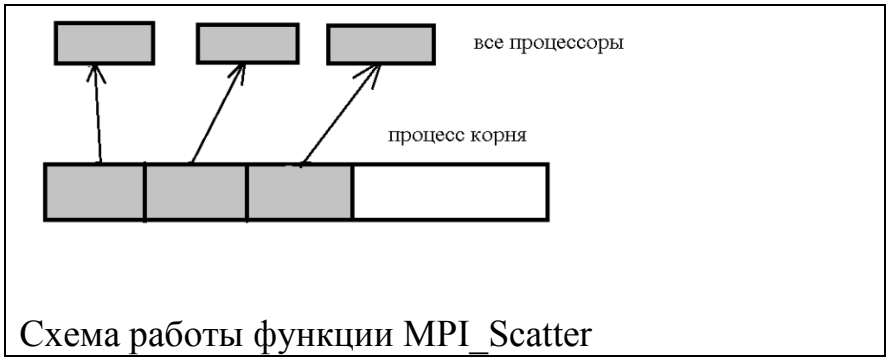

Схема работы функции MPI_Scatter

Тип посылаемых элементов sendtype должен совпадать с типом recvtype получаемых элементов, а число посылаемых элементов sendcount должно равняться числу принимаемых recvcount. Следует обратить внимание, что значение sendcount в вызове из процесса root - это число посылаемых каждому процессу элементов, а не общее их количество. Операция Scatter является обратной по отношению к Gather.

Совмещенные коллективные операции

Функция MPI_Alltoall совмещает в себе операции Scatter и Gather и является по сути дела расширением операции Allgather, когда каждый процесс посылает различные данные разным получателям. Процесс i посылает j-ый блок своего буфера sendbuf процессу j, который помещает его в i-ый блок своего буфера recvbuf. Количество посланных данных должно быть равно количеству полученных данных для каждой пары процессов.

int **MPI_Alltoall**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_COMM_WORLD)

sendbuf - адрес начала буфера посылки;
sendcount - число посылаемых элементов;
sendtype - тип посылаемых элементов;
recvbuf - адрес начала буфера приема;
recvcount - число элементов, получаемых от каждого процесса;
recvtype - тип получаемых элементов;

int **MPI_Reduce**(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op oper, int root, MPI_Comm comm)

-- sendbuf - адрес передаваемого буфера;
-- recvbuf - адрес буфера приема;
-- count - количество передаваемых элементов;
-- datatype - тип посылаемых элементов;
-- oper - операция редукции;
-- root - ранг корневого процесса;
-- comm - коммуникатор.

Функция выполняет операцию редукции oper над элементами массива sendbuf и помещает результат в resbuf корневого процесса root.

Имена некоторых операций:

MPI_MAX – определение максимального значения;
MPI_MIN – определение минимального значения;
MPI_SUM – суммирование;
MPI_PROD – произведение;

Эта функция объединяет элементы, находящиеся в передаваемом буфере каждого процесса, используя операцию oper и возвращает объединенное значение в буфер приема в процессе с рангом root. Аргументы oper и root должны быть идентичны во всех процессах. Каждый процесс может обеспечивать один элемент или последовательность элементов, когда объединяющая операция выполнена поэлементно на каждом элементе последовательности. Например, если выполняется операция MPI_MAX и посылающий буфер содержит два элемента, то resbuf(0) = глобальному max(sendbuf(0)) и resbuf(1) = глобальному max(sendbuf(1)).

Виртуальные топологии

Под виртуальной топологией в MPI понимается программно реализуемая топология в виде конкретного графа, например: кольцо, решетка, тор, звезда, дерево и вообще произвольно задаваемый граф на существующей физической топологии. Виртуальная топология обеспечивает очень удобный механизм наименования процессов, связанных коммуникатором, и является мощным средством отображения процессов на оборудование системы. Виртуальная топология в MPI может задаваться только в группе процессов, объединенных intracommunicator (внутри-групповым коммуникатором), и не может задаваться в группе процессов, объединенных intercommunicator'ом (межгрупповым коммуникатором). Группа процесса в MPI - это набор из n процессов. Каждому процессу в группе назначен ранг между 0 и n-1. Во многих параллельных приложениях линейная нумерация процессов адекватно не отражает логическую модель связи процессов (которая обычно определяется основной геометрией задачи и определенным используемым алгоритмом). Часто параллельные алгоритмы представляются в топологических моделях типа двумерных или объемных сетках. В более общем случае логическое расположение процессов описывается некоторым произвольным графом.

Нужно различать виртуальную топологию процессов и топологию основного физического оборудования. Механизм виртуальных топологий значительно упрощает и облегчает написание параллельных программ, делает программы легко читаемыми и понятными. Пользователю при этом не нужно программировать схему физических связей процессоров, а только схему виртуальных связей между процессами. Отображение виртуальных связей на физические осуществляет система, что делает параллельные программы машинно-независимыми и легко переносимыми.

Таким образом, механизм виртуальных топологий позволяет оптимально отображать структуру параллельной задачи на архитектуру вычислительной системы. Имеются общеизвестные методы отображения структур сетка/тор на топологии оборудования типа гиперкуб или сетка.

Модель связи множества процессов друг с другом может быть представлена графом. Узлы представляют процессы, а грани соединяют процессы, которые связаны друг с другом. Так как связь наиболее часто симметрична, графы связи приняты симметричными: если ребро uv соединяет узел u с узлом v, то ребро vu соединяет узел v с узлом u.

MPI обеспечивает передачу сообщений между любой парой процессов в группе, которая может быть представлена как направленный граф, где вершина - это процессы и грани - сообщения. Программист сообщает системе MPI типичные связи, например, топологию его программы. Это шаг к компромиссам, когда определенная топология создается под определенную связность процессов, которая используется в любое время в программе. В целом, однако, механизм топологии был выбран как по-

лезный компромисс между функциональными возможностями и ухудшением использования.

Определение виртуальной топологии в терминах графа достаточно для всех приложений. Однако во многих приложениях структура графа регулярна, и детальная установка графа была бы неудобна для пользователя и могла бы быть менее эффективна во время выполнения. Большая часть всех параллельных приложений использует топологию связей процессов, подобную кольцам, двухмерным (или выше) сеткам или торам. Эти структуры полностью определены числом размерности и числом процессов в каждом направлении координаты. Также отображение сеток и торов - обычно более простая задача, чем общие графы. Таким образом, желательно задавать эти топологии явно.

Так, параллельный алгоритм умножения двух матриц удобно реализовать в топологии процессоров типа «кольцо». Рассмотрим работу такого алгоритма для случая четырех процессоров. Пусть, например, матрицы являются квадратными, левая из которых разрезана на четыре полосы по строкам, правая также на четыре полосы, но по столбцам. В начале вычислений в каждом i-том процессоре располагаются i-я строка матрицы A и i-й столбец матрицы B (рис. 1). В результате их перемножения вычисляется соответствующая часть строки результирующей матрицы C. Далее процессоры осуществляют обмен столбцами, в ходе которого каждая из них передает свой столбец матрицы B следующему процессору в соответствии с кольцевой структурой информационных взаимодействий. Повторение описанных действий за 4 такта завершает выполнение параллельного алгоритма.

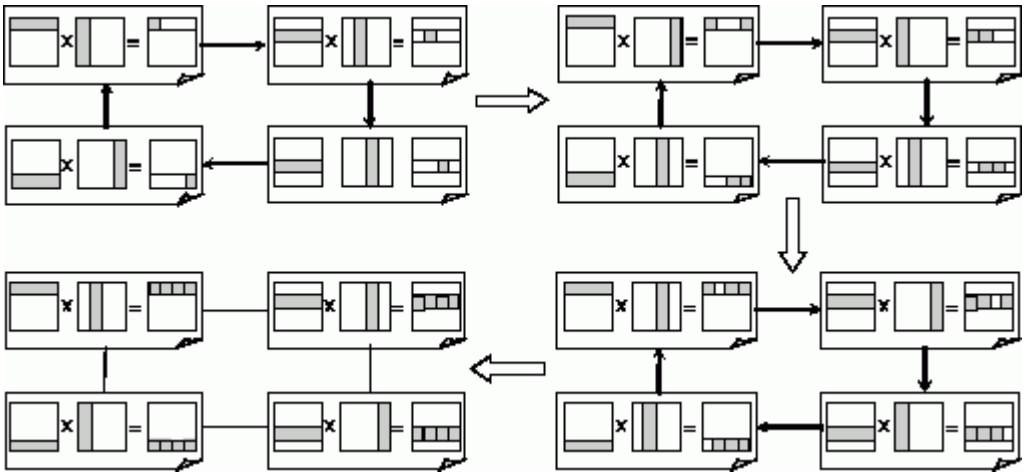

Рис. 1. Общая схема передачи данных для параллельного алгоритма матричного умножении.

Координаты процессов в декартовой структуре начинают их маркирование с 0. Линейная нумерация всегда используется для процессов в декартовой структуре. Это означает, например, что соотношение между рангом группы и координатами для двенадцати процессов в 2D топологии на сетке 3x4 такое, как показано на рис. 2. Верхний номер в каждой клетке - ранг процесса, а нижнее значение (строка, столбец) - координат.

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 2 (2,2)	11 (2,3)

Рис. 2. Связь между рангами и декартовыми координатами для 3x4 2D топологии

функции декартовых топологий

Рассмотрим несколько наиболее употребительных функций MPI для создания декартовых топологий.

Функция, конструирующая декартову топологию

MPI_Cart_create используется для описания декартовой структуры произвольного измерения. Для каждого направления координаты определяется, является ли структура процесса периодической или нет. Для 1D топологии - это линейка, если она не периодическая, и кольцо, если она периодическая. Для 2D топологии - это прямоугольник, цилиндр или тор.

int MPI_**Cart_create**(MPI_Comm comm_old, int ndims, int *dims,
int *periods, int reorder, MPI Comm *comm_cart)

comm_old	входной (старый) коммуникатор
ndims	количество измерений в декартовой топологии
dims	целочисленный массив размером ndims, определяющий количество процессов в каждом измерении
periods	массив размером ndims логических значений, определяющих периодичность (true) или нет (false) в каждом измерении
reorder	ранги могут быть перенумерованы (true) или нет (false)
comm_cart	коммуникатор новой (созданной) декартовой топологии

MPI_Cart_create возвращает управление новому коммуникатору, к которому присоединена информация декартовой топологии. Эта функция коллективная. Как в случае с другими коллективными функциями, вызов ее должен быть синхронизован во всех процессах.

Если reorder = false, тогда ранг каждого процесса в новой группе идентичен ее рангу в старой группе. Иначе функция может переупорядочивать процессы (возможно, чтобы выбрать хорошее отображение виртуальной топологии на физическую топологию). Если полный размер декартовой сетки меньше, чем размер группы comm_old, то некоторые процессы возвращают MPI_COMM_NULL.

Декартова функция задания сетки

Для декартовой топологии функция MPI_Dims_create помогает пользователю выбрать сбалансированное распределение процессов по направлению координат, в за-

висимости от числа процессов в группе и необязательных ограничений, которые могут быть определены пользователем. Одно возможное использование этой функции - это разбиение всех процессов (размер группы MPI_COMM_WORLD) в N-мерную топологию.

int **MPI_Dims_create**(int nnodes, int ndims, int *dims)

nnodes	количество узлов в решетке
ndims	мерность декартовой топологии
dims	целочисленный массив, определяющий количество узлов в каждой размерности

Элементы в массиве dims представляют описание декартовой сетки с размерностями ndims и общим количеством узлов nnodes. Пользователь может ограничивать действие этой функции, определяя элементы массива dims. Если в dims[i] уже записано число, то функция не будет изменять количество узлов в измерении i; функция модифицирует только нулевые элементы в массиве, т.е. где dims[i] = 0. Для dims[i] установленных функцией, dims[i] будут упорядочены в монотонно уменьшающемся порядке. Массив dims подходит для использования, как вход к функции MPI_Cart_create. Функция MPI_Dims_create локальная. Отдельные типовые запросы показаны на следующем примере

Пример

dims перед вызо-	Функции	dims после воз-
(0,0)	MPI_Dims_create (6, 2, dims)	(3,2)
(0,0)	MPI_Dims_create (7, 2, dims)	(7,1)
(0,3,0)	MPI_Dims_create (6, 3, dims)	(2,3,1)

Декартовы функции транслирования

Функции, приведенные в этом пункте здесь переводят на/из ранга в декартовы координаты топологии. Эти запросы локальные.

int **MPI_Cart_rank**(MPI_Comm comm, int *coords, int *rank)

comm	коммуникатор с декартовой топологией
coords	целочисленный массив, определяющий координаты нужного процесса в декартовой топологии
rank	ранг нужного процесса

Для группы процессов с декартовой структурой функция MPI_Cart_rank переводит логические координаты процессов в ранги, поскольку ранги используются в point-to-

point операциях, coords - массив размером ndims. Для примера на рис.1 процесс с coords= (1,2) возвратил бы rank = 6.
Для измерения i с periods(i) = true, если координата, coords (i), находится вне диапазона, т. е. coords (i < 0 или coords(i >= dims(i), она перемещается назад к интервалу 0 <= coords(i) < dims(i) автоматически. Если топология на рис. 1 периодическая в обеих размерностях, то процесс с coords = (4,6) также возвратил бы rank = 6. Для непериодических размерностей диапазон вне координат ошибочен.

int **MPI_Cart_coords**(MPI_Comm comm, int rank, int maxdims,
int *coords)

comm	коммуникатор с декартовой топологией
rank	ранг процесса в топологии comm
maxdims	максимальный размер массивов dims, periods, и coords в вызывающей программе
coords	целочисленный массив, определяющий координаты нужного процесса в декартовой топологии

MPI_Cart_coords переводит ранг процесса в координаты процесса в топологии. Это обратное отображение MPI_Cart_rank.

Декартова функция смещения

Если в декартовой топологии используется функция MPI_Sendrecv для смещения данных вдоль направления какой либо координаты, то входным аргументом MPI_Sendrecv берется ранг процесса source (процесса источника) для приема данных и ранг процесса dest (процесса назначения) для передачи данных. Операция смещения в декартовой топологии определяется координатой смещения и размером шага смещения (положительным или отрицательным). Функция MPI_Cart_shift возвращает информацию для входных спецификаций, требуемых в вызове MPI_Sendrecv . Функция MPI_Cart_shift локальная.

int **MPI_Cart_shift**(MPI_Comm comm, int direction, int disp,
int *rank_source, int *rank_dest)

comm	коммуникатор с декартовой топологией
direction	номер измерения (в топологии), где делается смешение
disp	направление смешения (> 0 смешение в сторону увеличения номеров координаты direction, < 0 смешение в сторону уменьшения номеров координаты direction)
rank_source	ранг процесса источника
rank_dest	ранг процесса назначения

Аргумент direction указывает измерение, в котором осуществляется смещение данных. Координаты маркируются от 0 до ndims-1, где ndims - число размерностей.

В зависимости от периодичности декартовой топологии в указанном направлении координат, MPI_Cart_shift обеспечивает идентификаторы rank_source и rank_dest для кругового или end-о смещения. В случае end-о смещения в rank_source и/или rank_dest может быть возвращено значение MPI_PROC_NULL, указывая, что процесс источника и/или процесс назначения для смещения находится вне диапазона. Это имеющий силу входной аргумент к функции MPI_Sendrecv .

Обмен данными на системе компьютеров с топологией связи "кольцо"

В этом примере демонстрируется операция сдвига данных вдоль кольца компьютеров. Эта операция очень часто встречается при решении задач на кластерах. Этот пример может послужить неким шаблоном, который можно применять для решения сложных задач. В нем выполняется сдвиг данных соседним ветвям вдоль кольца компьютеров на один шаг, т.е. все ветви параллельной программы передают данные соседним ветвям, например, в сторону увеличения рангов (номеров) компьютеров.

```
/* Сдвиг данных на кольце компьютеров */

#include <mpi.h>
#include <stdio.h>
#define NUM_DlMS 1
int main(   int argc,   char** argv )
{
int     rank,   size,   l,   A,   B,  dims[NUM_DIMS];
int     periods [NUM_DIMS] ,  source,  dest;
int     reorder = 0;
MPI_Comm      comm_cart;
MPI_Status      status;
MPI_Init(&argc, &argv);

 /*  Каждая ветвь узнает обшее количество ветвей */
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
 /* и свой номер    от 0 до  (size-1)  */
A=rank;B=-1;
/* Обнуляем массив dims и заполняей насеяв periods для топологии "кольцо" */
for( i= 0; i < NUM_DIMS; i++) { dims[i] = 0; periods[i] = 1; }
/* Заполняем массив dims, где указываются размеры (одномерной) решетки */
MPI_Dims_create(size, NUM_DIMS, dims);
/* Создаем топологию "кольцо" с communicator(om) comm_cart */
MPI_Cart_create(MPI_COMM_WORLD, NUM_DIMS, dims, periods, reorder,
&comm_cart);
```

```
/* Каждая ветвь находит своих соседей вдоль кольца, в направлении
*       больших значений рангов */
MPI_Cart_shift(comm_cart, 0, 1, &source, &dest);
/* 0-ветвь инициирует передачу данных (значение своего ранга) вдоль
*       кольца, и принимает это же значение от ветви size-1 */
if(rank == 0)
MPI_Send(&A, 1, MPI_lNT, dest, 12, comm_cart);
MPI_Recv(&B, 1, MPI_INT, source, 12, comm_cart, &status);
printf("rank= %d A=%d B=%d \n", rank, A, B); }
/* Работа всех остальных ветвей */
else
{ MPI_Recv(&B, 1, MPI.INT, source, 12, comm_cart, &status);
MPI_Send(&B, 1, MPI.INT, dest, 12, comm_cart); }
/* Все ветви завершают системные процессы, связанные с топологией
comm_cart и завершаю выполнение программы */
MPI_Comm_free(&comm_cart);
MPI_Finalze(); return 0;}
```

Обмен данными на системе компьютеров с топологией связи "линейка"

Этот пример демонстрирует операцию сдвига данных на линейке компьютеров, которая очень часто встречается при решении задач на кластере. Он, как и предыдущий, может послужить неким шаблоном, который используется для решения сложных задач. Все ветви линейки одновременно сдвигают свои данные соседним ветвям вдоль линейки на один шаг в сторону увеличения и затем в сторону уменьшения ранга ветвей. У граничных ветвей соседи имеются только с одной стороны. Поэтому здесь удобно использовать понятие MPI_PROC_NULL процессов.

```
/*
Сдвиг данных на линейке процессов с использованием MPI_PROC_NULL процессов
*/
#inlude <mpi.h>
#include <stdio.h>
int main(int argc,   char **argv)
{
int     rank,  size,  i,  A,  B,  dims[1] ;
int     periods[1],  new_coords[1];
int     souirceb,   destb,   sourcem,  destm;
int     reorder = 0;
MPI_Comm   comm_cart;
MPI_Status    status;
MPI_Init(&argc, &argv);
/* Каждая ветвь узнает количество ветвей */
```

```
MPl_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 /* и свой номер от 0 но (size-1) */
/* Обнуляем массив dims и заполняем массив periods для топологии "линейка" */
for(i = 0; i < 1; i++) { dims[i] = 0; periods[i] = 0; }
/* Заполняем массив dims, где указываются размеры (одномерной) решетки */
MPI_Dims_create(size, 1, dims);
 /* Создаем топологию "линейка" с communicator(ом) comm_cart */
MPI_Cart_create(MPI_COMM_WORLD, 1, dims, periods, reorder,
&comm_cart);
 /* Отображаем ранги в координаты и выводим их */
MPI_Cart_coords(comm_cart, rank, 1, new_coords);
A = new_coords[0];B = -1;
 /* Каждая ветвь находит своих соседей вдоль линейки, в направлении больших значений номеров компьютеров и в направлении меньших значений номеров. Ветви с номером new_coords[0] == 0 не имеют соседей с меньшим номером, поэтому с этого направления эти ветви принимают данные от несуществующих ветвей, т.е. от ветвей sourcem = MPI_PROC_NULL, и, соответственно, передают данные в этом направлении ветвям destm = MPI_PROC_NULL Аналогично определяется соседство для ветвей с номером new_coords[0] == dims[0] - 1
*/
if(new_coords[0] == 0){sourcem = destm = MPI_PROC_NULL;}
else{ sourcem = destm = new_coords[0]-l;}
if(new_coords[0]  == dims[0]-l){destb = sourceb =NPI_PROC_NULL;}
else{ destb = sourceb = new_coords[0]+l;}
 /* Каждая ветвь передает своя данные (значение переменной A) своей
соседней ветви с большим номером и принимает данные в B от
соседней ветви с меньшим номером. Свой номер и номер,
принятый в B выводятся на печать
*/
MPI_Sendrecv(&A, 1, MPI_INT, destb, 12, &B, 1, MPI_INT, sourcem, 12,
comm_cart, &status);
printf ("new_coords[0] = %d B = %d\n", new_coords[0] , B);

/* Сдвиг данных в противоположную сторону и вывод соответствующих данных */
MPI_Sendrecv(&A, 1, MPI_INT, destm, 12, &B, 1, MPI_INT, sourceb, 12,
comm_cart, &status);
printf("new_coords[0] = %d B = %d\n", new_coords[0], B);
 /* Все ветви завершают системные процессы, связанные с топологией
comm_cart и завершают выполнение программы */
MPI_Comm_free(&comm_cart);
MPI_Finalize();
return 0;
```

Применение MPI_PROC_NULL процессов значительно упрощает программирование подобных операций, т. к. не нужно отдельно программировать граничные и внутренние ветви решетки.

ЗАКЛЮЧЕНИЕ

Приведенные описания даже небольшого числа функций MPI показывают, что при написании параллельных программ с использованием механизма передачи сообщений алгоритмы решения даже простейших задач, таких как, например, перемножения матриц, перестают быть тривиальными. И совсем уж не тривиальной становится задача написания эффективных программ для решения более сложных задач линейной алгебры. Сложность программирования с использованием механизма передачи сообщений долгое время оставалась основным сдерживающим фактором на пути широкого использования многопроцессорных систем с распределенной памятью.

В последние годы ситуация значительно изменилась благодаря появлению достаточно эффективных библиотек подпрограмм для решения широкого круга задач. Такие библиотеки избавляют программистов от рутинной работы по написанию подпрограмм для решения стандартных задач численных методов и позволяют сконцентрироваться на предметной области. Однако использование этих библиотек не избавляет от необходимости ясного понимания принципов параллельного программирования и требует выполнения достаточно объемной подготовительной работы.

ЛИТЕРАТУРА

1. Воеводин Вл.В. Курс лекций "Параллельная обработка данных". http://parallel.ru/parallel/vvv

2. Корнеев В.Д. Параллельное программирование в MPI. – Москва-Ижевск: Ин-т компьютерных исследований, 2003. – 304 с.

3. MPI: The Message Passing Interface. http://parallel.ru/tech/tech_dev/mpi.html

ПРОГРАММИРОВАНИЕ НА OpenMP

Интерфейс OpenMP задуман как стандарт для программирования на масштабируемых SMP-системах (SSMP,ccNUMA, etc.) в модели общей памяти (shared memory model). В стандарт OpenMP входят спецификации набора директив компилятора, процедур и переменных среды.

Основной источник информации - сервер www.openmp.org. На сервере доступны спецификации, статьи, учебные материалы, ссылки.

До появления OpenMP не было подходящего стандарта для эффективного программирования на SMP-системах.

Следует отметить, что наиболее гибким, переносимым и общепринятым интерфейсом параллельного программирования является MPI (интерфейс передачи сообщений). Однако модель передачи сообщений 1) недостаточно эффективна на SMP-системах; 2) относительно сложна в освоении, так как требует мышления в "невычислительных" терминах.

Преимущества OpenMP

За счет идеи "инкрементального распараллеливания" OpenMP идеально подходит для разработчиков, желающих быстро распараллелить свои вычислительные программы с большими параллельными циклами. Разработчик не создает новую параллельную программу, а просто последовательно добавляет в текст последовательной программы OpenMP-директивы.

OpenMP - достаточно гибкий механизм, предоставляющий разработчику большие возможности контроля над поведением параллельного приложения.

Предполагается, что OpenMP-программа на однопроцессорной платформе может быть использована в качестве последовательной программы, т.е. нет необходимости поддерживать последовательную и параллельную версии. Директивы OpenMP просто игнорируются последовательным компилятором.

В OpenMP используется терминология и модель программирования, близкая к Pthreads (динамически порождаемые нити, общие и разделяемые данные, механизм "замков" для синхронизации). Разделяемые для параллельных процессов данные располагаются в общей памяти и для организации взаимодействия не требуется операций передачи сообщений.

Простой пример: вычисление приближенного значения числа "Пи" за конечное число шагов N. $$\pi \approx \frac{1}{N}\sum_{i=1}^{N}\frac{4}{1+[(i-0.5)/N]^2}$$.

В последовательную программу вставлена одна строка, и она распараллелена!

```
#include <omp.h>
main( )
{    int i, N=1000;
   double x, pi, step=1.0/N, sum = 0.0;
#pragma omp parallel for reduction(+:sum)  private(x)
   for (i = 1; i <= N; i++)
   {
   x = (i - 0.5)*step;  sum = sum + 4.0 / (1.0 + x * x);
   }
   pi = step * sum;
  printf("pi= %13.4e",pi);
}
```

Операция редукции

Параметр reduction определяет список переменных, для которых выполняется операция редукции. Перед выполнением параллельной области для каждого потока создаются копии этих переменных, потоки формируют значения в своих локальных переменных и при завершении параллельной области над всеми локальными значениями выполняются необходимые операции редукции, результаты которых запоминаются в исходных (глобальных) переменных reduction (operator: list).

Классы переменных

В OpenMP переменные в параллельных областях программы разделяются на два основных класса:

shared (общие; под именем А все нити видят одну переменную) и

private (приватные; под именем А каждая нить видит свою переменную).

Отдельные правила определяют поведение переменных при входе и выходе из параллельной области или параллельного цикла: reduction, firstprivate, lastprivate, copyin.

По умолчанию, все переменные, порожденные вне параллельной области, при входе в эту область остаются общими (shared). Исключение составляют переменные - счетчики итераций в цикле, по очевидным причинам. Переменные, порожденные внутри параллельной области, являются приватными (private).

Директивы

Директивы OpenMP на языке C начинаются с комбинации символов "#pragma. Директивы можно разделить на 3 категории: определение параллельной секции, разделение работы, синхронизация. Каждая директива может иметь несколько дополнительных атрибутов - клауз. Отдельно специфицируются клаузы для назначения классов переменных, которые могут быть атрибутами различных директив.

Порождение нитей

Директива parallel (основная директива OpenMP). Когда основной поток выполнения достигает директиву parallel, создается набор (team) из N потоков (нитей); входной поток является основным потоком этого набора (master thread) и имеет номер 0. Код области дублируется или разделяется между потоками для параллельного выполнения. В конце области обеспечивается синхронизация потоков – выполняется ожидание завершения вычислений всех потоков; далее все потоки завершаются – дальнейшие вычисления продолжает выполнять только основной поток. Предполагается, что в SMP-системе нити будут распределены по различным процессорам (однако это, как правило, находится в ведении операционной системы). Каким образом между порожденными нитями распределяется работа - определяется директивами do, sections и single.

Формат директивы parallel

#pragma omp parallel [clause ...] newline
structured_block

Возможные параметры (clause)

if (scalar_expression) (если условие в if не выполняется, то процессы не создаются)
private (list)
firstprivate (list)
default (shared | none)
shared (list)
copyin (list)
reduction (operator: list)
num_threads(integer-expression)

Пример использования директивы parallel

```
#include <omp.h>
main ( ) {
int nthreads, tid;
// Создание параллельной области
#pragma omp parallel private(nthreads, tid)
```

```
{
// печать номера потока
tid = omp_get_thread_num();printf("Hello World from thread = %d\n", tid);
// Печать количества потоков – только master
if (tid == 0) { nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
} // Завершение параллельной области
}
```

С помощью функций omp_get_thread_num( ) и omp_get_num_threads( ); нить может узнать свой номер и общее число нитей, а затем выполнять свою часть работы в зависимости от своего номера.

Управление областью видимости данных

Управление областью видимости обеспечивается при помощи параметров (clause) директив private, firstprivate, lastprivate, shared, default, reduction, copyin которые определяют, какие соотношения существуют между переменными последовательных и параллельных фрагментов выполняемой программы

Параметр shared (list) определяет список переменных, которые будут общими для всех потоков параллельной области; правильность использования таких переменных должна обеспечиваться программистом

Параметр private (list) определяет список переменных, которые будут локальными для каждого потока; переменные создаются в момент формирования потоков параллельной области; начальное значение переменных является неопределенным.

Параметр firstprivate (list) позволяет создать локальные переменные потоков, которые перед использованием инициализируются значениями исходных переменных

Параметр lastprivate (list) позволяет создать локальные переменные потоков, значения которых запоминаются в исходных переменных после завершения параллельной области (используются значения потока, выполнившего последнюю итерацию цикла или последнюю секцию).

Распределение вычислений между потоками

Существует 3 директивы для распределения вычислений в параллельной области

for – распараллеливание циклов
sections – распараллеливание раздельных фрагментов кода (функциональное распараллеливание)
single – директива для указания последовательного выполнения кода

Начало выполнения директив по умолчанию не синхронизируется

Завершение директив по умолчанию является синхронным

Формат директивы for

#pragma omp for [clause ...] newline
for loop
Возможные параметры (clause)
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
ordered
schedule(kind[, chunk_size])
nowait

Клауза schedule определяет способ распределения итераций по нитям:

static,m – статически, блоками по m итераций
dynamic,m – динамически, блоками по m (каждая нить берет на выполнение первый еще невзятый блок итераций)
guided,m – размер блока итераций уменьшается экспоненциально до величины m

По умолчанию, в конце цикла происходит неявная синхронизация; эту синхронизацию можно запретить с помощью nowait

Пример использования директивы for

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000
main () { int i, n, chunk;
float a[NMAX], b[NMAX], c[NMAX];
for (i=0; i < NMAX; i++) a[i] = b[i] = i * 1.0;
n = NMAX; chunk = CHUNK;
#pragma omp parallel shared(a,b,c,n,chunk) private(i)
{
#pragma omp for schedule(dynamic,chunk) nowait
for (i=0; i < n; i++) c[i] = a[i] + b[i];
} // end of parallel
}
```

Директива sections – распределение вычислений для раздельных фрагментов кода. Фрагменты выделяются при помощи директивы section. Каждый фрагмент выполняется однократно, разные фрагменты выполняются разными потоками. Завершение директивы по умолчанию синхронизируется. Директивы section должны использоваться только в статическом контексте

Формат директивы sections

```
#pragma omp sections [clause ...] newline
{
#pragma omp section newline
structured_block
#pragma omp section newline
structured_block
}
```

Возможные параметры (clause)

```
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
nowait
```

Пример использования директивы sections

```
#include <omp.h>
#define NMAX 1000
main () { int i, n;
float a[NMAX], b[NMAX], c[NMAX];
for (i=0; i < NMAX; i++) a[i] = b[i] = i * 1.0;
n = NMAX;
#pragma omp parallel shared(a,b,c,n) private(i)
{
#pragma omp sections nowait
{
#pragma omp section
for (i=0; i < n/2; i++) c[i] = a[i] + b[i];
#pragma omp section
for (i=n/2; i < n; i++) c[i] = a[i] + b[i];
} // end of sections
} // end of parallel section
}
```

Формат директивы single

```
#pragma omp single [clause ...] newline
structured_block
```

Определяет блок, который будет исполнен только одной нитью (первой, которая дойдет до этого блока).

Возможные параметры (clause)

```
private(list)
firstprivate(list)
copyprivate(list)
nowait
```

Объединение директив parallel и for/sections

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000
main () {
int i, n, chunk; float a[NMAX], b[NMAX], c[NMAX];
for (i=0; i < NMAX; i++) a[i] = b[i] = i * 1.0;
n = NMAX; chunk = CHUNK;
#pragma omp parallel for shared(a,b,c,n) private(i) \
schedule(static,chunk)
for (i=0; i < n; i++) c[i] = a[i] + b[i];
}
```

Директивы синхронизации

Директива master определяет фрагмент кода, который должен быть выполнен только основным потоком; все остальные потоки пропускают данный фрагмент кода (завершение директивы по умолчанию не синхронизируется)

```
#pragma omp master newline
   structured_block
```

Директива critical определяет блок кода, который должен выполняться только одним потоком в каждый текущий момент времени, то есть блок кода, который не должен выполняться одновременно двумя или более нитями

```
#pragma omp critical [name] newline
   structured_block
```

Пример использования директивы critical

```
#include <omp.h>
 main() {     int x;  x = 0;
   #pragma omp parallel shared(x)
   {
    #pragma omp critical
      x = x + 1;
   } // end of parallel section
 }
```

Директива barrier – определяет точку барьерной синхронизации, в которой каждая нить дожидается всех остальных.

```
#pragma omp barrier newline
```

Пример программы на OpenMP

```
/*    Умножение двух матриц  A - M x N и B - N x L, целиком располагающихся
   в памяти каждого процессора. OpenMP программа
*/
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define M 2400
#define N 2000
#define L 100
double A[M][N],B[N][L],C[M][L],C_t[M][L];

int main( )
{ int i, j, k; double sum;
        for(i = 0; i < M; i++)for(j = 0; j < N; j++)A[i][j] = i + j;
        for(i = 0; i < N; i++)for( j = 0; j < L; j++)B[i][j] = i * j;
#pragma omp parallel for private(i, j, k)
 for(i = 0; i < M; i++)
 {
        for(j = 0; j < L; j++)
      {
        C[i][j] = 0.0;
        for(k = 0; k < N; k++) C[i][j]  += A[i][k] * B[k][j];
       }
  }
         return 0;
}
```

Информационные ресурсы

www.openmp.org
Что такое OpenMP – http://parallel.ru/tech/tech_dev/openmp.html

Introduction to OpenMP -
www.llnl.gov/computing/tutorials/workshops/workshop/openMP/MAIN.html

Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J. Parallel Programming in OpenMP. – Morgan Kaufmann Publishers, 2000

Quinn, M. J. Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill, 2004.