Основы параллельного программирования с использованием MPI

Лекция 7

Немнюгин Сергей Андреевич

Санкт-Петербургский государственный университет

физический факультет

кафедра вычислительной физики
snemnyugin@mail.ru

Интернет-Университет
Суперкомпьютерных Технологий

High-Performance Computing University

Лекция 7

Аннотация

В этой лекции рассматриваются методы работы с пользовательскими типами данных. Пользовательские типы позволяют обойти некоторые ограничения MPI. Разъясняются такие вопросы, как построение карты типа, его регистрация и аннулирование. Обсуждаются примеры использования пользовательских типов в MPI-программах. Рассматриваются также виртуальные топологии, которыми могут быть наделены группы процессов, относящихся к MPI-программе. Основное внимание уделяется декартовым топологиям. Приводятся примеры.

План лекции

- Проблемы организации данных при обмене сообщениями.
- Пользовательские типы. Карта типа, правила использования типа.
- Операции упаковки данных.
- Виртуальные топологии. Декартова топология и топология графа.

Пользовательские типы

Пользовательские типы

Сообщение в MPI представляет собой массив однотипных данных, элементы которого расположены в последовательных ячейках памяти. Такая структура не всегда удобна.

Иногда возникает необходимость в пересылке разнотипных данных или фрагментов массивов, содержащих элементы, расположенные не в последовательных ячейках. Так бывает, например, при программировании вычислений, связанных с операциями с матрицами и векторами, а также в других ситуациях.

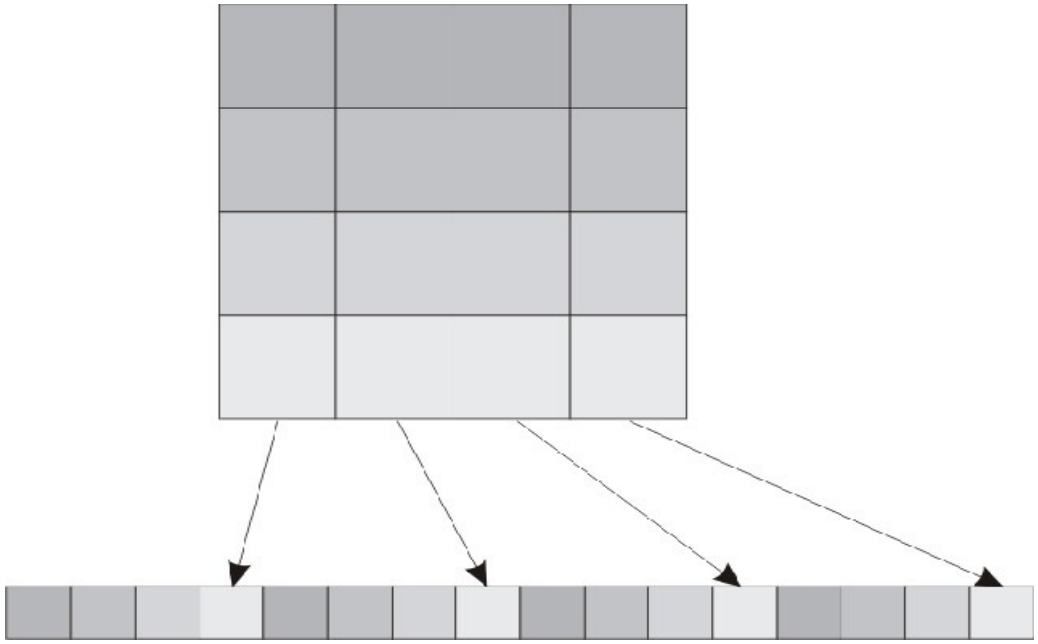
Эта проблема легко решается при программировании с PVM. Концепция «бабушкиной посылки».

В MPI-программе придется создать производный тип данных.

Пользовательские типы

Пример

В численных расчетах часто приходится иметь дело с матрицами. В языках Fortran и C используется линейная модель памяти, в которой матрица хранится в виде последовательно расположенных столбцов (строк). Один из алгоритмов параллельного умножения матриц требует пересылки строк матрицы, а в линейной модели Фортрана элементы строк расположены в оперативной памяти не непрерывно, а с промежутками.



Пользовательские типы

Решать проблему пересылки разнотипных данных или данных, расположенных не в последовательных ячейках памяти, можно разными средствами.

Можно «уложить» элементы исходного массива во вспомогательный массив так, чтобы данные располагались непрерывно. Это неудобно и требует дополнительных затрат памяти и процессорного времени.

Можно различные элементы данных пересылать по отдельности. Это медленный и неудобный способ обмена.

Более эффективным решением является использование *производных типов данных*.

Пользовательские типы

Производные типы данных создаются во время выполнения программы (а не во время ее трансляции), как правило, перед их использованием.

Создание типа — двухступенчатый процесс, который состоит из двух шагов:

1. конструирование типа.
2. регистрация типа.

После завершения работы с производным типом, он аннулируется. При этом все производные от него типы остаются и могут использоваться дальше, пока и они не будут уничтожены.

Коммуникаторы, группы, типы данных, распределенные операции, запросы, атрибуты коммуникаторов, обработчики ошибок — все это объекты, которые (если они были созданы в процессе работы программы) должны удаляться.

Пользовательские типы

Производные типы данных создаются из базовых типов с помощью подпрограмм–конструкторов.

Производный тип данных в MPI характеризуется последовательностью базовых типов и набором целочисленных значений смещения.

Смещения отсчитываются относительно начала буфера обмена и определяют те элементы данных, которые будут участвовать в обмене. Не требуется, чтобы они были упорядочены (например, по возрастанию или по убыванию).

Отсюда следует, что порядок элементов данных в производном типе может отличаться от исходного и, кроме того, один элемент данных может появляться в новом типе многократно.

Пользовательские типы

Последовательность пар (тип, смещение) называется *картой типа*:

1-й элемент пары	2-й элемент пары
Базовый тип 0	Смещение базового типа 0
Базовый тип 1	Смещение базового типа 1
Базовый тип 2	Смещение базового типа 2
…	…
Базовый тип n – 1	Смещение базового типа n – 1

Расстояние в количестве ячеек задается между началом элементов, поэтому элементы могут располагаться с разрывами и перекрываться между собой.

Пользовательские типы

Конструкторы производных типов

Подпрограмма `MPI_Type_struct` является наиболее общим конструктором типа в MPI - программист может использовать полное описание каждого элемента типа.

Если пересылаемые данные содержат подмножество элементов массива, такая детальная информация не нужна, поскольку у всех элементов один и тот же базовый тип. MPI предлагает три конструктора, которые можно использовать в такой ситуации: `MPI_Type_contiguous`, `MPI_Type_vector` и `MPI_Type_indexed`.
Первый из них создает производный тип, элементы которого являются непрерывно расположенными элементами массива.
Второй создает тип, элементы которого расположены на одинаковых расстояниях друг от друга.
Третий создает тип, содержащий произвольные элементы.

Пользовательские типы

Векторный тип создается конструктором `MPI_Type_vector`:

```
int MPI_Type_vector(int count, int blocklen, int stride,
   MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_Type_vector(count, blocklen, stride, oldtype, newtype,
   ierr)
```

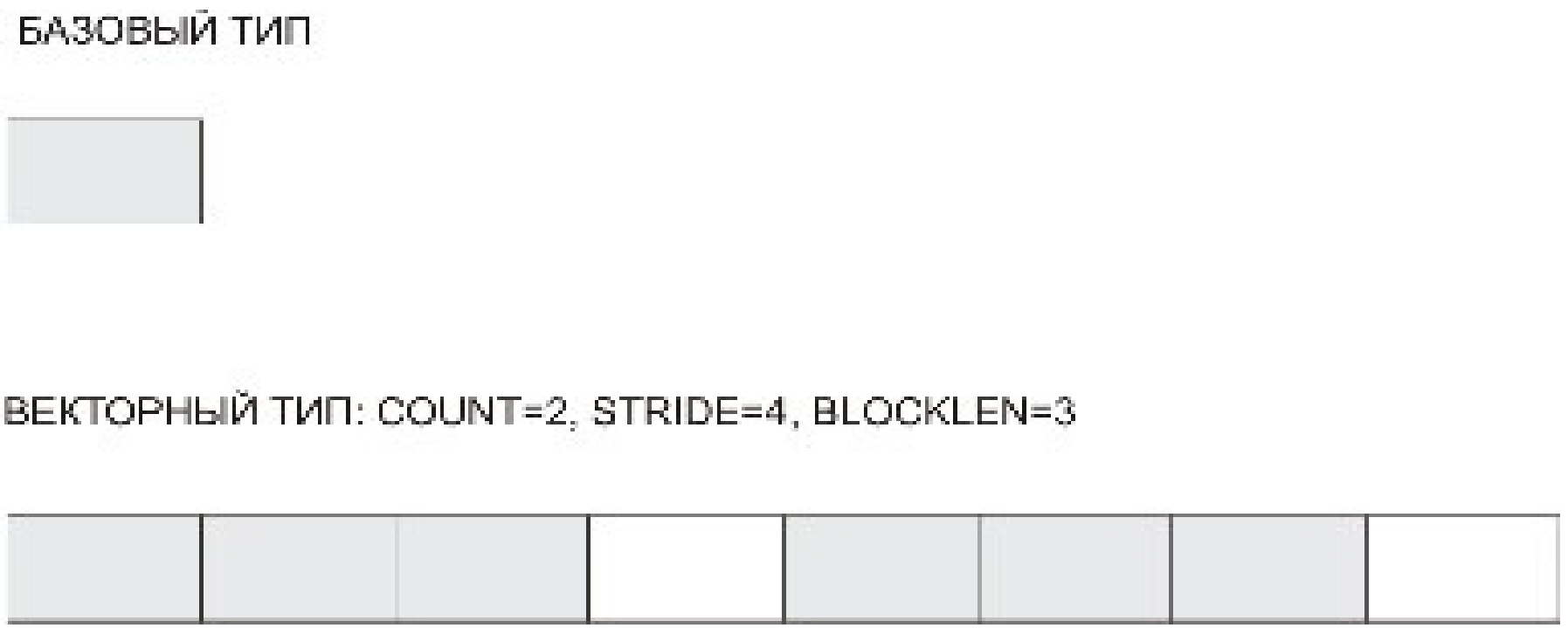
Входные параметры:

- `count` — количество блоков (неотрицательное целое значение);
- `blocklen` — длина каждого блока (количество элементов, неотрицательное целое);
- `stride` — количество элементов, расположенных между началом предыдущего и началом следующего блока («гребенка»);
- `oldtype` — базовый тип.

Выходным параметром является идентификатор нового типа `newtype`. Этот идентификатор назначается программистом. Исходные данные здесь однотипные.

Пользовательские типы

Схема расположения данных в новом типе



Пользовательские типы

Пример

```
count = 2;
stride = 4;
blocklen = 3;
oldtype = double;
```

карта нового типа:

```
{(double, 0), (double, 1), (double, 2), (double, 4),
   (double, 5), (double, 6)}
```

Пользовательские типы

При вызове подпрограммы `MPI_Type_hvector` смещения задаются в байтах:

```
int MPI_Type_hvector(int count, int blocklen, MPI_Aint
stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_Type_hvector(count, blocklen, stride, oldtype, newtype,
ierr)
```

Смысл и назначение параметров этой подпрограммы совпадают с подпрограммой `MPI_Type_vector`, только значение параметра `stride` задается в байтах.

Пользовательские типы

Конструктором структурного типа является подпрограмма `MPI_Type_struct`. Она позволяет создать тип, содержащий элементы различных базовых типов:

```
int MPI_Type_struct(int count, int blocklengths[], MPI_Aint
indices[], MPI_Datatype oldtypes[], MPI_Datatype *newtype)

MPI_Type_struct(count, blocklengths, indices, oldtypes,
newtype, ierr)
```

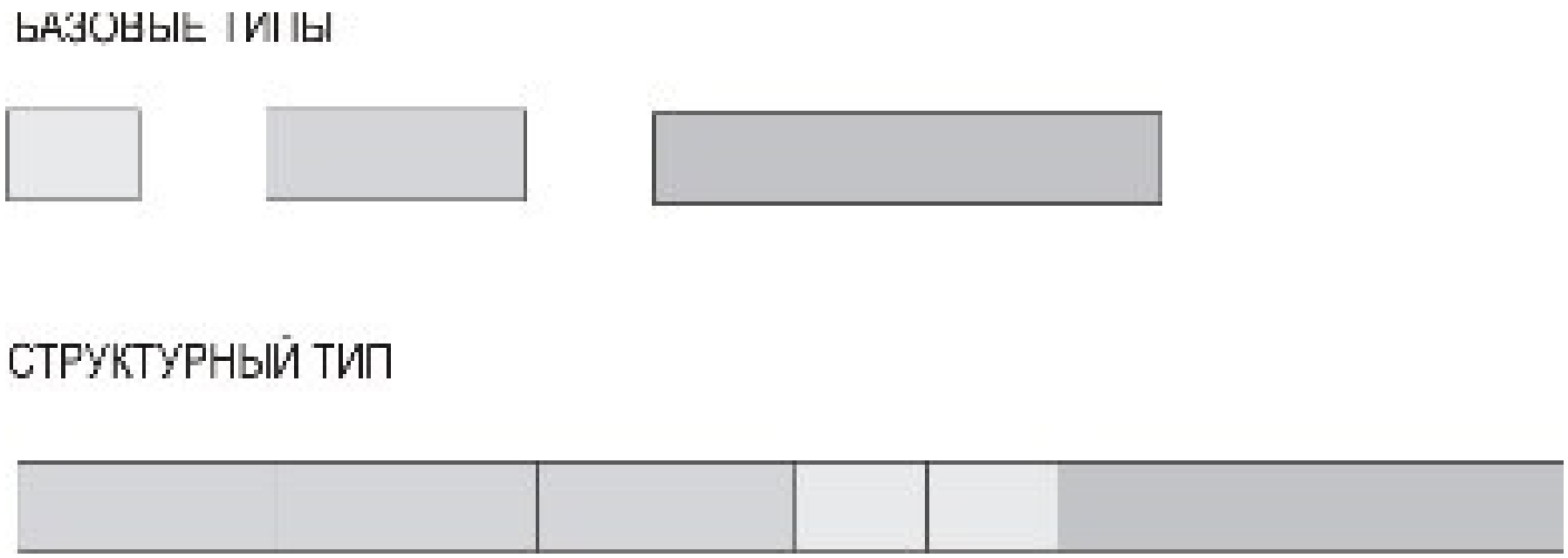
Ее входные параметры:

- `count` — задает количество элементов в производном типе, а также длину массивов `oldtypes`, `indices` и `blocklengths`;
- `blocklengths` — количество элементов в каждом блоке (массив);
- `indices` — смещение каждого блока в байтах (массив);
- `oldtypes` — тип элементов в каждом блоке (массив).

Выходной параметр — идентификатор производного типа `newtype`.

Пользовательские типы

Схема расположения данных в структурном типе



Пользовательские типы

Пример создания структурного производного типа:

```
blen[0] = 1;
indices[0] = 0;
oldtypes[0] = MPI_INT;
blen[1] = 1;
indices[1] = &data.b – &data;
oldtypes[1] = MPI_CHAR;
blen[2] = 1;
indices[2] = sizeof(data);
oldtypes[2] = MPI_FLOAT;
MPI_Type_struct(3, blen, indices, oldtypes, &newtype);
```

В качестве упражнения попробуйте построить карту нового типа `newtype` для этого примера. Достаточно ли для этого информации, содержащейся в приведенном фрагменте программы?

Пользовательские типы

При создании индексированного типа блоки располагаются по адресам с разным смещением и его можно считать обобщением векторного типа. Конструктором индексированного типа является подпрограмма:

```
int MPI_Type_indexed(int count, int blocklens[], int
indices[], MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_Type_indexed(count, blocklens, indices, oldtype, newtype,
ierr)
```

Ее входные параметры:

- `count` — количество блоков, одновременно длина массивов `indices` и `blocklens`;
- `blocklens` — количество элементов в каждом блоке;
- `indices` — смещение каждого блока, которое задается в количестве ячеек базового типа (целочисленный массив);
- `oldtype` — базовый тип.

Выходным параметром является идентификатор производного типа `newtype`.

Пользовательские типы

В языке Fortran роль смещений могут играть значения индексов массива. Смещение в этом случае отсчитывается относительно первого элемента массива. Численные значения смещений в картах типа отсчитываются от нуля, а значения индексов в Fortran по умолчанию отсчитываются от единицы. Для согласования обеих систем отсчета иногда удобно описывать массивы следующим образом:

```
real vector_a(0:99)
```

В этом случае значение индекса массива равно значению смещения.

Пользовательские типы

Подпрограмма `MPI_Type_hindexed` также является конструктором индексированного типа, однако смещения `indices` задаются в байтах:

```
int MPI_Type_hindexed(int count, int blocklens[], MPI_Aint
indices[], MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_Type_hindexed(count, blocklens, indices, oldtype, newtype,
ierr)
```

Пользовательские типы

Подпрограмма `MPI_Type_contiguous` используется для создания типа данных с непрерывным расположением элементов:

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
MPI_Datatype *newtype)

MPI_Type_contiguous(count, oldtype, newtype, ierr)
```

Ее входные параметры:

- `count` — счетчик повторений;
- `oldtype` — базовый тип.

Выходной параметр — `newtype` идентификатор нового типа. Эта подпрограмма фактически создает описание массива.

Пользовательские типы

Пример

```
MPI_Datatype a;
float b[10];
…
MPI_Type_contiguous(10, MPI_REAL, &a);
MPI_Type_commit(&a);
MPI_Send(b, 1, a,…);
MPI_Type_free(&a);
```

Пользовательские типы

Тип данных, соответствующий подмассиву многомерного массива, можно создать с помощью подпрограммы:

```
int MPI_Type_create_subarray(int ndims, int *sizes, int
*subsizes, int *starts, int order, MPI_Datatype oldtype,
MPI_Datatype *newtype)

MPI_Type_create_subarray(ndims, sizes, subsizes, starts,
order, oldtype, newtype, ierr)
```

Входные параметры:

- `ndims` — размерность массива;
- `sizes` — количество элементов типа `oldtype` в каждом измерении полного массива;
- `subsizes` — количество элементов типа `oldtype` в каждом измерении подмассива;
- `starts` — стартовые координаты подмассива в каждом измерении;
- `order` — флаг, задающий переупорядочение;
- `oldtype` — базовый тип.

Пользовательские типы

Новый тип `newtype` является выходным параметром этой подпрограммы.

Существуют и другие конструкторы производных типов, но они используются значительно реже.

Пользовательские типы

Регистрация и удаление производных типов

С помощью вызова подпрограммы `MPI_Type_commit` производный тип `datatype`, сконструированный программистом, регистрируется. После этого он может использоваться в операциях обмена:

```
int MPI_Type_commit(MPI_Datatype *datatype)

MPI_Type_commit(datatype, ierr)
```

Аннулировать производный тип `datatype` можно с помощью вызова подпрограммы `MPI_Type_free`:

```
int MPI_Type_free(MPI_Datatype *datatype)

MPI_Type_free(datatype, ierr)
```

Предопределенные (базовые) типы данных не могут быть аннулированы.

Пользовательские типы

Вспомогательные подпрограммы

Размер типа `datatype` в байтах (объем памяти, занимаемый одним элементом данного типа) можно определить с помощью вызова подпрограммы `MPI_Type_size`:

```
int MPI_Type_size(MPI_Datatype datatype, int *size)

MPI_Type_size(datatype, size, ierr)
```

Выходным параметром является размер `size`.

Пользовательские типы

Количество элементов данных в одном объекте типа `datatype` (его *экстент*) можно определить с помощью вызова подпрограммы `MPI_Type_extent`:

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)

MPI_Type_extent(datatype, extent, ierr)
```

Выходной параметр — `extent`. Смещения могут даваться относительно базового адреса, значение которого содержится в константе `MPI_BOTTOM`.

Пользовательские типы

Адрес (`address`) по заданному положению (`location`) можно определить с помощью подпрограммы `MPI_Address`:

```
int MPI_Address(void *location, MPI_Aint *address)

MPI_Address(location, address, ierr)
```

Эта подпрограмма может понадобиться в программе на языке Fortran, а в C есть собственные средства для определения адреса.

Пользовательские типы

С помощью подпрограммы `MPI_Type_get_contents` можно определить фактические параметры, использованные при создании производного типа:

```
int MPI_Type_get_contents(MPI_Datatype datatype, int
max_integers, int max_addresses, int max_datatypes, int
*integers, MPI_Aint *addresses, MPI_Datatype *datatypes)

MPI_Type_get_contents(datatype, max_integers, max_addresses,
max_datatypes, integers, addresses, datatypes, ierr)
```

Входные параметры:

- `datatype` — идентификатор типа;
- `max_integers` — количество элементов в массиве `integers`;
- `max_addresses` — количество элементов в массиве `addresses`;
- `max_datatypes` — количество элементов в массиве `datatypes`.

Пользовательские типы

Выходные параметры:

- `integers` — содержит целочисленные аргументы, использованные при конструировании указанного типа;
- `addresses` — содержит аргументы `address`, использованные при конструировании указанного типа;
- `datatypes` — содержит аргументы `datatype`, использованные при конструировании указанного типа.

Пользовательские типы

Пример 1 (ex01e.f90)

```
program main_mpi
include 'mpif.h'
parameter (n = 50)
real arr(n, n, n), b(9, 9, 9)
integer slice1, slice2, slice3, sizeofreal
integer rank, ierr, status(MPI_STATUS_SIZE)
integer tag, cnt, vcount, blocklen, stride, count
cnt = 1
tag = 0
vcount = 9
blocklen = 1
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
```

Пользовательские типы

```
if (rank.eq.0) then
call MPI_Type_extent(MPI_REAL, sizeofreal, ierr)
stride = 2
call MPI_Type_vector(vcount, blocklen, stride, MPI_REAL, &
   slice1, ierr)
stride = n * sizeofreal
call MPI_Type_hvector(vcount, blocklen, stride, slice1, &
slice2, ierr)
stride = n * n * sizeofreal
call MPI_Type_hvector(vcount, blocklen, stride, slice2, &
slice3, ierr)
call MPI_Type_commit(slice3, ierr)
call MPI_Send(arr(1, 3, 2), cnt, slice3, 1, tag, &
   MPI_COMM_WORLD, ierr)
else if (rank.eq.1) then
count = vcount * vcount * vcount
call MPI_Recv(b, count, MPI_REAL, 0, tag, MPI_COMM_WORLD,&
   status, ierr)
print *, b
end if
call MPI_Finalize(ierr)
end
```

Пользовательские типы

В этой программе строятся производные типы `slice1`, `slice2` и `slice3`, соответствующие сечениям исходного трехмерного массива `A`.

Каким?
Попробуйте самостоятельно определить производные структуры данных.

Между процессами в сечении выполняется блокирующий двухточечный обмен сообщениями.

Пользовательские типы

Пример 2 (ex02e.cpp) – создание структурного типа

```
#include "mpi.h"
#include <stdio.h>
 struct newtype {
 float a;
 float b;
 int n;
 };
int main(int argc,char *argv[])
{
 int myrank;
 MPI_Datatype NEW_MESSAGE_TYPE;
 int block_lengths[3];
 MPI_Aint displacements[3];
 MPI_Aint addresses[4];
 MPI_Datatype typelist[3];
 int blocks_number;
 struct newtype indata;
 int tag = 0;
 MPI_Status status;
```

Пользовательские типы

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
typelist[0] = MPI_FLOAT;
typelist[1] = MPI_FLOAT;
typelist[2] = MPI_INT;
block_lengths[0] = block_lengths[1] = block_lengths[2] = 1;
MPI_Address(&indata, &addresses[0]);
MPI_Address(&(indata.a), &addresses[1]);
MPI_Address(&(indata.b), &addresses[2]);
MPI_Address(&(indata.n), &addresses[3]);
displacements[0] = addresses[1] – addresses[0];
displacements[1] = addresses[2] – addresses[0];
displacements[2] = addresses[3] – addresses[0];
blocks_number = 3;
MPI_Type_struct(blocks_number, block_lengths, displacements,
  typelist, &NEW_MESSAGE_TYPE);
MPI_Type_commit(&NEW_MESSAGE_TYPE);
```

Пользовательские типы

```
 if (myrank == 0)
{
 indata.a = 3.14159;
 indata.b = 2.71828;
 indata.n = 2010;
 MPI_Send(&indata, 1,NEW_MESSAGE_TYPE, 1, tag, MPI_COMM_WORLD);
 printf("Process %i send: %f %f %i\n", myrank, indata.a,
   indata.b, indata.n);
}
else
{
 MPI_Recv(&indata, 1, NEW_MESSAGE_TYPE, 0, tag, MPI_COMM_WORLD,
   &status);
 printf("Process %i received: %f %f %i, status %s\n", myrank,
   indata.a, indata.b, indata.n, status.MPI_ERROR);
}
 MPI_Type_free(&NEW_MESSAGE_TYPE);
 MPI_Finalize();
 return 0;
}
```

Пользовательские типы

Как работает эта программа

Задаются типы членов производного типа.

Задаётся количество элементов каждого типа.

Вычисляются адреса элементов типа `indata` и определяются смещения трех членов производного типа относительно адреса первого, для которого смещение равно 0. Располагая этой информацией, можно определить производный тип, что и делается с помощью подпрограмм `MPI_Type_struct` и `MPI_Type_commit`.

Созданный таким образом производный тип можно использовать в любых операциях обмена.

Пользовательские типы

Пример 3 (ex03e.f90) – создание структурного типа в программе на Fortran'e

```
program main_mpi
include 'mpif.h'
integer rank, tag, cnt, ierr, status(MPI_STATUS_SIZE)
parameter (bufsize=100)
character rcvbuf1(bufsize)
integer newtype
integer blocks, position
integer disp(2), blen(2), type(2)
integer address(2)
integer data1, data3
complex data2, data4

cnt = 1
tag = 0
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
```

Пользовательские типы

```
if(rank == 0) then
blen(1) = 1
blen(2) = 1
type(1) = MPI_INTEGER
type(2) = MPI_COMPLEX
blocks = 2
call MPI_Address(data1, address(1), ierr)
disp(1) = address(1)
call MPI_Address(data2, address(2), ierr)
disp(2) = address(2)
call MPI_Type_struct(blocks, blen, disp, type, newtype, ierr)
call MPI_Type_commit(newtype, ierr)

data1 = 3
data2 = (1., 3.)
call MPI_Send(MPI_BOTTOM, cnt, newtype, 1, tag, &
MPI_COMM_WORLD, ierr)
print *, "process ", rank, " send ", data1, " and ", data2
call MPI_TYPE_Free(newtype, ierr)
```

Пользовательские типы

```
else
position = 0
call MPI_Recv(rcvbuf1, bufsize, MPI_PACKED, 0, tag, &
   MPI_COMM_WORLD, status, ierr)
call MPI_Unpack(rcvbuf1, bufsize, position, data3, 1,&
   MPI_INTEGER, MPI_COMM_WORLD, ierr)
call mpi_unpack(rcvbuf1, bufsize, position, data4, 1, &
   MPI_COMPLEX, MPI_COMM_WORLD, ierr)
print *, "process ", rank, " received ", data3, " and ", data4
end if
call MPI_Finalize(ierr)
stop
end
```

Пользовательские типы

Результат выполнения этой программы выглядит так:

```
# mpirun -np 2 a.out

process 0 send 3 and (1.,3.)
process 1 received 3 and (1.,3.)
```

В программе используется одна из подпрограмм упаковки и распаковки (`MPI_Unpack`).

Операции упаковки данных

Операции упаковки данных

Использование подпрограмм упаковки и распаковки `MPI_Pack` и `MPI_Unpack` является альтернативным методом группировки данных.

Подпрограмма `MPI_Pack` позволяет явным образом хранить произвольные (в том числе и расположенные не в последовательных ячейках) данные в непрерывной области памяти (буфере передачи).

Подпрограмму `MPI_Unpack` используют для копирования данных из буфера приёма в произвольные (в том числе и не расположенные непрерывно) ячейки памяти.

Операции упаковки данных

Подпрограммы упаковки используются для решения следующих задач:

- обеспечения совместимости с другими библиотеками обмена сообщениями;
- для приема сообщений по частям;
- для того, чтобы буферизовать исходящие сообщения в пользовательское пространство памяти, что дает независимость от системной политики буферизации.

Сообщения передаются по коммуникационной сети, связывающей узлы вычислительной системы. Сеть работает медленно, поэтому, чем меньше в параллельной программе обменов, тем меньше потери на пересылку данных. С учетом этого полезен механизм, который позволял бы вместо отправки трех разных значений тремя сообщениями, отправлять их все вместе. Такие механизмы есть: это параметр `count` в подпрограммах обмена, производные типы данных и подпрограммы `MPI_Pack` и `MPI_Unpack`.

Операции упаковки данных

С помощью аргумента `count` в подпрограммах `MPI_Send`, `MPI_Recv`, `MPI_Bcast` и `MPI_Reduce` можно отправить в одном сообщении несколько однотипных элементов данных. Для этого элементы данных должны находиться в непрерывно расположенных ячейках памяти.

Если элементы данных — это простые переменные, тогда они могут не находиться в последовательных ячейках памяти. В этом случае можно использовать производные типы данных или упаковку.

Подпрограмма упаковки может вызываться несколько раз перед передачей сообщения, содержащего упакованные данные, а подпрограмма распаковки в этом случае также будет вызываться несколько раз после выполнения приема. Для извлечения каждой порции данных применяется новый вызов. При распаковке данных текущее положение указателя в буфере сохраняется.

Операции упаковки данных

Интерфейс подпрограммы `MPI_Pack`:

```
int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype,
void *outbuf, int outcount, int *position, MPI_Comm comm)

MPI_Pack(inbuf, incount, datatype, outbuf, outcount,
position, comm, ierr)
```

При вызове `incount` элементов указанного типа выбираются из входного буфера и упаковываются в выходном буфере, начиная с положения `position`. Входные параметры:

- `inbuf` — начальный адрес входного буфера;
- `incount` — количество входных данных;
- `datatype` — тип каждого входного элемента данных;
- `outcount` — размер выходного буфера в байтах;
- `position` — текущее положение в буфере в байтах;
- `comm` — коммуникатор для упакованного сообщения.

Выходным параметром является стартовый адрес выходного буфера `outbuf`.

Операции упаковки данных

Подпрограмма распаковки данных:

```
int MPI_Unpack(void *inbuf, int insize, int *position, void
*outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)

MPI_Unpack(inbuf, insize, position, outbuf, outcount,
datatype, comm, ierr)
```

Входные параметры:

- `inbuf` — стартовый адрес входного буфера;
- `insize` — размер входного буфера в байтах;
- `position` — текущее положение в байтах;
- `outcount` — количество данных, которые должны быть распакованы;
- `datatype` — тип каждого выходного элемента данных;
- `comm` — коммуникатор для упаковываемого сообщения.

Выходной параметр `outbuf` — стартовый адрес выходного буфера.

Операции упаковки данных

Подпрограмма `MPI_Pack_size` позволяет определить объем памяти `size` (в байтах), необходимый для распаковки сообщения:

```
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm
comm, int *size)

MPI_Pack_size(incount, datatype, comm, size, ierr)
```

Входные параметры:

- `incount` — аргумент `count`, использованный при упаковке;
- `datatype` — тип упакованных данных;
- `comm` — коммуникатор.

Операции упаковки данных

Пример использования подпрограмм упаковки (ex04e.cpp)

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
 int myrank;
 float a, b;
 int n;
 int root = 0;
 char buffer[100];
 int position;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

Операции упаковки данных

```
if (myrank == root){
 printf("Enter a, b, and n\n");
 scanf("%f %f %i", &a, &b, &n);
 position = 0;
 MPI_Pack(&a, 1, MPI_FLOAT, &buffer, 100, &position,
MPI_COMM_WORLD);
 MPI_Pack(&b, 1, MPI_FLOAT, &buffer, 100, &position,
MPI_COMM_WORLD);
 MPI_Pack(&n, 1, MPI_INT, &buffer, 100, &position,
MPI_COMM_WORLD);
 MPI_Bcast(&buffer, 100, MPI_PACKED, root, MPI_COMM_WORLD);
 }
```

Операции упаковки данных

```
else {
 MPI_Bcast(&buffer, 100, MPI_PACKED, root, MPI_COMM_WORLD);
 position = 0;
 MPI_Unpack(&buffer, 100, &position, &a, 1, MPI_FLOAT,
MPI_COMM_WORLD);
 MPI_Unpack(&buffer, 100, &position, &b, 1, MPI_FLOAT,
MPI_COMM_WORLD);
 MPI_Unpack(&buffer, 100, &position, &n, 1, MPI_INT,
MPI_COMM_WORLD);
 printf("Process %i received a=%f, b=%f, n=%i\n", myrank,
a, b, n);
}
 MPI_Finalize();
 return 0;
}
```

Операции упаковки данных

Как работает эта программа

Процесс 0 копирует в буфер значение `a`, затем дописывает туда `b` и `n`. После выполнения широковещательной рассылки главным процессом, остальные используют подпрограмму `MPI_Unpack` для извлечения `a`, `b`, и `n` из буфера. Тип данных, который надо использовать при вызове `MPI_Bcast` — `MPI_PACKED`.

Тип данных `MPI_BYTE` не имеет аналога в языках C или Fortran. Он позволяет хранить данные в «сыром» формате, имеющем одинаковое двоичное представление на стороне источника сообщения и на стороне адресата. Представление символа может различаться на разных машинах, а байт везде одинаков. Тип `MPI_BYTE` может использоваться, если необходимо выполнить преобразование между разными представлениями в гетерогенной вычислительной системе.

Виртуальные топологии

Виртуальные топологии

Организация обмена основана на свойствах коммуникатора — описателя области взаимодействия. Кроме списка процессов и контекста обмена с коммуникатором может быть связана дополнительная информация.

Важнейшей разновидностью такой информации является *топология* обменов. В MPI топология представляет собой механизм сопоставления процессам, принадлежащим группе, альтернативных по отношению к обычной схем нумерации.

Топологии обменов сообщениями в MPI являются *виртуальными*, они не связаны с физической топологией коммуникационной сети параллельной вычислительной системы. Использование коммуникаторов и топологий отличает MPI от большинства других систем передачи сообщений.

Виртуальные топологии

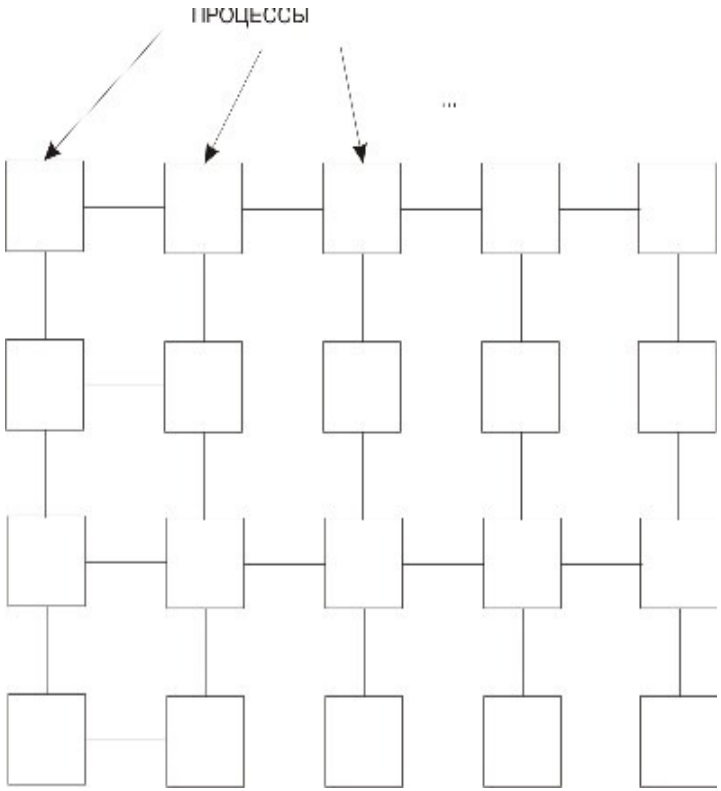
Топологией в данном случае называют структуру соединений - линий и узлов сети без учета характеристик самих этих узлов. Узлами здесь являются процессы, соединениями — каналы обмена сообщениями, а сетью мы, фактически, называем все процессы, входящие в состав параллельной программы.

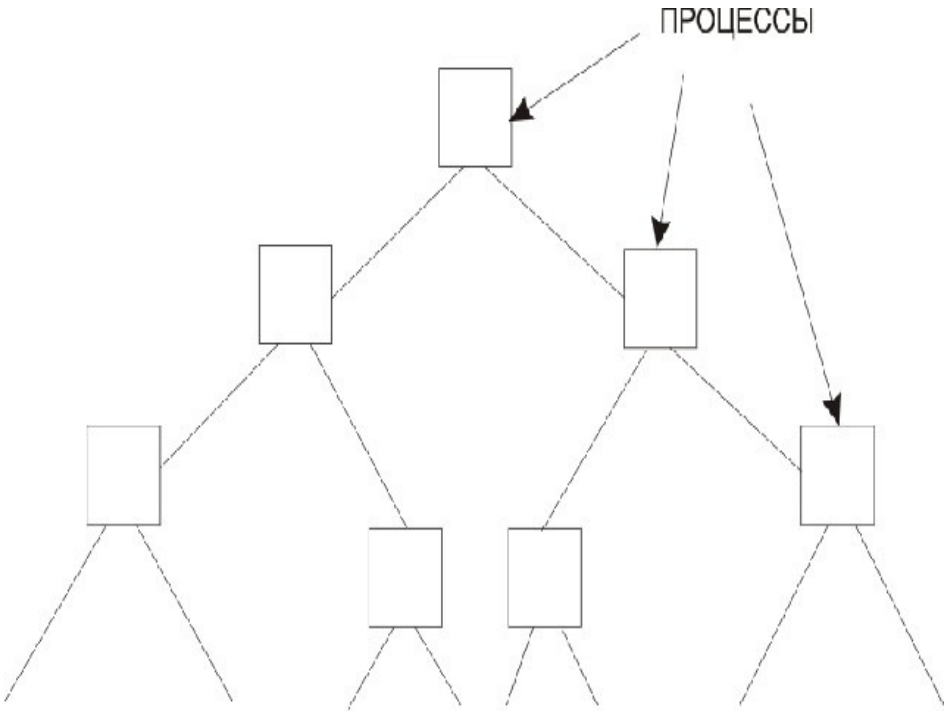
Часто в прикладных программах процессы упорядочены в соответствии с определенной топологией. Такая ситуация возникает, например, если выполняются расчеты, в которых используются решетки (сетки). Это может быть при программировании сеточных методов решения дифференциальных уравнений, а также в других случаях.

Знание топологии задачи можно использовать для того, чтобы эффективно распределить процессы между процессорами параллельной вычислительной системы.

Виртуальные топологии

В MPI существуют два типа топологии: *декартова* топология — прямоугольная решетка произвольной размерности и топология *графа* (в этом случае процессы соединены между собой ребрами, показывающими направление обмена).





Виртуальные топологии

Над топологиями можно выполнять различные операции.
Декартовы решетки можно расщеплять на гиперплоскости, удаляя некоторые измерения.
Данные можно сдвигать вдоль выбранного измерения декартовой решетки. *Сдвигом* в этом случае называют пересылку данных между процессами вдоль определенного измерения.
Вдоль избранного измерения могут быть организованы коллективные обмены.

Виртуальные топологии

Для того, чтобы связать структуру декартовой решетки с коммуникатором `MPI_COMM_WORLD`, необходимо задать следующие параметры:

- *размерность* решетки (значение 2, например, соответствует плоской решетке);
- *размер* решетки вдоль каждого измерения (размеры {10, 5}, например, соответствуют прямоугольной плоской решетке, протяженность которой вдоль оси *x* составляет 10 узлов-процессов, а вдоль оси *y* — 5 узлов);
- *граничные условия* вдоль каждого измерения (решетка может быть *периодической*, если процессы, находящиеся на противоположных концах ряда, взаимодействуют между собой).

MPI дает возможность системе оптимизировать отображение виртуальной топологии процессов на физическую с помощью изменения порядка нумерации процессов в группе.

Виртуальные топологии

Декартовы топологии

Познакомимся с операциями создания и преобразования *декартовых топологий* обмена. Подпрограмма `MPI_Cart_create` создает новый коммуникатор `comm_cart`, наделяя декартовой топологией исходный коммуникатор `comm_old`:

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
int *periods, int reorder, MPI_Comm *comm_cart)

MPI_Cart_create(comm_old, ndims, dims, periods, reorder,
comm_cart, ierr)
```

Входные параметры:

- `comm_old` — исходный коммуникатор;
- `ndims` — размерность декартовой решетки;
- `dims` —целочисленный массив, состоящий из `ndims` элементов, задающий количество процессов в каждом измерении;

Виртуальные топологии

- `periods` — логический массив из `ndims` элементов, который определяет, является ли решетка периодической (значение `true`) вдоль каждого измерения;
- `reorder` — при значении этого параметра «истина», системе разрешено менять порядок нумерации процессов.

Информация о структуре декартовой топологии содержится в параметрах `ndims`, `dims` и `periods`. `MPI_Cart_create` является коллективной операцией (эту подпрограмму должны вызывать все процессы из коммуникатора, наделяемого декартовой топологией).

Виртуальные топологии

После создания виртуальной топологии можно использовать соответствующую схему адресации процессов, но для этого требуется пересчет ранга процесса в его декартовы координаты и наоборот. Определить декартовы координаты процесса по его рангу в группе можно с помощью подпрограммы `MPI_Cart_coords`:

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

MPI_Cart_coords(comm, rank, maxdims, coords, ierr)
```

Входные параметры:

- `comm` — коммуникатор, наделенный декартовой топологией;
- `rank` — ранг процесса в `comm`;
- `maxdims` — длина вектора `coords` в вызывающей программе.

Выходным параметром этой подпрограммы является одномерный целочисленный массив `coords` (его размер равен `ndims`), содержащий декартовы координаты процесса.

Виртуальные топологии

Обратным действием обладает подпрограмма `MPI_Cart_rank`. С ее помощью можно определить ранг процесса (`rank`) по его декартовым координатам в коммуникаторе `comm`:

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

MPI_Cart_rank(comm, coords, rank, ierr)
```

Входной параметр `coords` — целочисленный массив размера `ndims`, задающий декартовы координаты процесса.

`MPI_Cart_rank` и `MPI_Cart_coords` локальны.

Виртуальные топологии

Пример создания декартовой топологии (ex05e.cpp)

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
 MPI_Comm grid_comm;
 int dims[2];
 int periodic[2];
 int reorder = 1, q = 5, ndims = 2, maxdims = 2;
 int coordinates[2];
 int my_grid_rank;
 int coords[2];
 MPI_Comm row_comm;
```

Виртуальные топологии

```
 dims[0] = dims[1] = q;
 periodic[0] = periodic[1] = 1;
 coords[0] = 0; coords[1] = 1;
 MPI_Init(&argc, &argv);
 MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periodic,
reorder, &grid_comm);
 MPI_Comm_rank(grid_comm, &my_grid_rank);
 MPI_Cart_coords(grid_comm, my_grid_rank, maxdims,
coordinates);
 printf("Process rank %i has coordinates %i %i\n",
my_grid_rank, coordinates[0], coordinates[1]);
 MPI_Finalize();
 return 0;
}
```

Виртуальные топологии

В этой программе коммуникатор `grid_comm` наделяется топологией двумерной решетки с периодическими граничными условиями, причем системе разрешено изменить порядок нумерации процессов. С учетом последнего, каждый процесс из коммуникатора `grid_comm` может определить свой ранг с помощью подпрограммы `MPI_Comm_rank`. Декартовы координаты определяются при вызове подпрограммы `MPI_Cart_coords`.

В программе создается топология двумерной квадратной решетки 5×5, поэтому количество процессов при запуске должно быть 25.

Результат выполнения этой программы выглядит так:

```
[nemnugin@pd00 ~]$ mpiexec -n 25 ./a.out
Process rank 0 has coordinates 0 0
Process rank 14 has coordinates 2 4
Process rank 1 has coordinates 0 1
Process rank 15 has coordinates 3 0
Process rank 16 has coordinates 3 1
Process rank 17 has coordinates 3 2
Process rank 24 has coordinates 4 4
Process rank 13 has coordinates 2 3
Process rank 22 has coordinates 4 2
Process rank 12 has coordinates 2 2
Process rank 2 has coordinates 0 2
Process rank 11 has coordinates 2 1
Process rank 3 has coordinates 0 3
Process rank 8 has coordinates 1 3
Process rank 23 has coordinates 4 3
Process rank 9 has coordinates 1 4
Process rank 21 has coordinates 4 1
Process rank 10 has coordinates 2 0
Process rank 6 has coordinates 1 1
Process rank 18 has coordinates 3 3
Process rank 5 has coordinates 1 0
Process rank 7 has coordinates 1 2
Process rank 4 has coordinates 0 4
Process rank 19 has coordinates 3 4
Process rank 20 has coordinates 4 0
```

Виртуальные топологии

Подпрограмма `MPI_Cart_sub` расщепляет коммуникатор `comm` на подгруппы, соответствующие декартовым подрешеткам меньшей размерности:

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm
*comm_new)

MPI_Cart_sub(comm, remain_dims, comm_new, ierr)
```

Здесь `i`-й элемент массива `remain_dims` определяет, содержится ли `i`-е измерение в подрешетке («истина»). Выходным параметром является коммуникатор `newcomm`, содержащий подрешетку, которой принадлежит данный процесс.

Виртуальные топологии

Пример разбиения двумерной решетки на одномерные подрешетки, соответствующие ее рядам, приводится ниже:

```
int coords[2];
MPI_Comm row_comm;
coords[0] = 0; coords[1] = 1;
MPI_Cart_sub(grid_comm, coords, &row_comm);
```

В этом примере вызов подпрограммы `MPI_Cart_sub` создает несколько новых коммуникаторов. Аргумент `coords` определяет, принадлежит ли соответствующее измерение новому коммуникатору.

Несмотря на то, что подпрограмма `MPI_Cart_sub` выполняет функции, аналогичные `MPI_Comm_split`, то есть расщепляет коммуникатор на набор новых коммуникаторов, между ними есть существенное различие — подпрограмма `MPI_Cart_sub` используется только с коммуникатором, наделенным декартовой топологией.

Виртуальные топологии

Информацию о декартовой топологии, связанной с коммуникатором `comm`, можно получить с помощью подпрограммы `MPI_Cart_get`:

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int
*periods, int *coords)

MPI_Cart_get(comm, maxdims, dims, periods, coords, ierr)
```

Входной параметр `maxdims` задает длину массивов `dims`, `periods` и `vectors` в вызывающей программе, а выходные параметры:

- `dims` — целочисленный массив, задающий количество процессов для каждого измерения;
- `periods` — массив логических значений, задающих периодичность (`true`, если решетка периодическая) для каждого измерения;
- `coords` — целочисленный массив, задающий декартовы координаты вызывающего подпрограмму процесса.

Виртуальные топологии

С помощью подпрограммы `MPI_Cart_map` можно определить ранг процесса (`newrank`) в декартовой топологии после переупорядочения процессов:

```
int MPI_Cart_map(MPI_Comm comm_old, int ndims, int *dims, int
*periods, int *newrank)

MPI_Cart_map(comm_old, ndims, dims, periods, newrank, ierr)
```

Входные параметры:

- `comm` — коммуникатор;
- `ndims` — размерность декартовой решетки;
- `dims` — целочисленный массив, состоящий из `ndims` элементов, который определяет количество процессов вдоль каждого измерения;
- `periods` — логический массив размера `ndims`, определяющий периодичность решетки вдоль каждого измерения.

Если процесс не принадлежит решетке, подпрограмма возвращает значение `MPI_UNDEFINED`.

Виртуальные топологии

Между процессами, организованными в декартову решетку, могут выполняться обмены особого вида. Это сдвиги, о которых мы уже упоминали. Имеются два типа сдвигов данных по группе из N процессов:

- *циклический сдвиг* на J позиций вдоль ребра решетки. Данные от процесса K пересылаются процессу $(J + K) \bmod N$;
- *линейный сдвиг* на J позиций вдоль ребра решетки, когда данные в процессе K пересылаются процессу $J + K$, если ранг адресата находится в пределах между 0 и N.

Виртуальные топологии

Ранги источника (`source`) сообщения, которое должно быть принято, и адресата (`dest`), который должен получить сообщение для заданного направления сдвига (`direction`) и его величины (`disp`), можно определить с помощью подпрограммы `MPI_Cart_shift`:

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int displ, int
*source, int *dest)

MPI_Cart_shift(comm, direction, displ, source, dest, ierr)
```

Для n-мерной декартовой решетки значение аргумента `direction` должно находиться в пределах от 0 до $n - 1$.

Подпрограмма `MPI_Cartdim_get` позволяет определить размерность (`ndims`) декартовой топологии, связанной с коммуникатором `comm`:

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)

MPI_Cartdim_get(comm, ndims, ierr)
```

Виртуальные топологии

Топологии графа

Подпрограмма MPI_Graph_create создает новый коммуникатор comm_graph, наделенный топологией графа:

```
int MPI_Graph_create(MPI_Comm comm, int nnodes, int *index,
int *edges, int reorder, MPI_Comm *comm_graph)

MPI_Graph_create(comm, nnodes, index, edges, reorder,
comm_graph, ierr)
```

Входные параметры:

- `comm` — исходный коммуникатор, не наделенный топологией;
- `nnodes` — количество вершин графа;
- `index` — целочисленный одномерный массив, содержащий порядок каждого узла (количество связанных с ним ребер);
- `edges` — целочисленный одномерный массив, описывающий ребра графа;
- `reorder` — значение «истина» разрешает изменение порядка нумерации процессов.

Виртуальные топологии

Подпрограмма `MPI_Graph_neighbors` возвращает соседние с данной вершины графа:

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int
maxneighbors, int *neighbors)

MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierr)
```

Ее входные параметры:

- `comm` — коммуникатор с топологией графа;
- `rank` — ранг процесса в группе коммуникатора `comm`;
- `maxneighbors` — размер массива `neighbors`.

Выходным параметром является массив `neighbors`, содержащий ранги процессов, соседних с данным.

Виртуальные топологии

Количество соседей (`nneighbors`) узла, связанного с топологией графа, можно определить с помощью подпрограммы:

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int
*nneighbors)

MPI_Graph_neighbors_count(comm, rank, nneighbors, ierr)
```

Входные параметры:

- `comm` — коммуникатор;
- `rank` — ранг процесса-узла.

Виртуальные топологии

Получить информацию о топологии графа, связанной с коммуникатором comm, можно с помощью подпрограммы `MPI_Graph_get`:

```
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges,
int *index, int *edges)

MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierr)
```

Входными параметрами являются:

- `comm` — коммуникатор;
- `maxindex` — длина массива `index` в вызывающей программе;
- `maxedges` — длина массива `edges` в вызывающей программе.

Выходные параметры:

- `index` — целочисленный массив, содержащий структуру графа;
- `edges` — целочисленный массив, содержащий сведения о ребрах графа.

Виртуальные топологии

С помощью подпрограммы `MPI_Graph_map` можно определить ранг процесса в топологии графа после переупорядочения (`newrank`):

```
int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int
*edges, int *newrank)

MPI_Graph_map(comm, nnodes, index, edges, newrank, ierr)
```

Ее входные параметры:

- `comm` — коммуникатор;
- `nnodes` — количество вершин графа;
- `index` — целочисленный массив, задающий структуру графа;
- `edges` — целочисленный массив, задающий ребра графа.

Если процесс не принадлежит графу, подпрограмма возвращает значение `MPI_UNDEFINED`.

Виртуальные топологии

Подпрограмма `MPI_Graphdims_get` позволяет получить информацию о топологии графа, связанной с коммуникатором `comm`:

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)

MPI_Graphdims_get(comm, nnodes, nedges, ierr)
```

Выходными параметрами этой подпрограммы являются:

- `nnodes` — количество вершин графа;
- `nedges` — количество ребер графа.

Определить тип топологии (`toptype`), связанной с коммуникатором `comm`, можно с помощью подпрограммы:

```
int MPI_Topo_test(MPI_Comm comm, int *toptype)

MPI_TOPO_TEST(COMM, TOPTYPE, IERR)
```

Выходным параметром является тип топологии `toptype` (значения `MPI_CART` для декартовой топологии и `MPI_GRAPH` для топологии графа).

Заключение

В этой лекции мы рассмотрели:

- способы создания пользовательских типов данных;
- операции упаковки/распаковки данных;
- виртуальные топологии.

Задания для самостоятельной работы

Решения следует высылать по электронной почте:

parallel-g112@yandex.ru

Задания для самостоятельной работы

Задания для самостоятельной работы представлены в тексте презентации