Методы и алгоритмы параллельных вычислений

Программный интерфейс для передачи информации (Message Passing Interface, MPI)

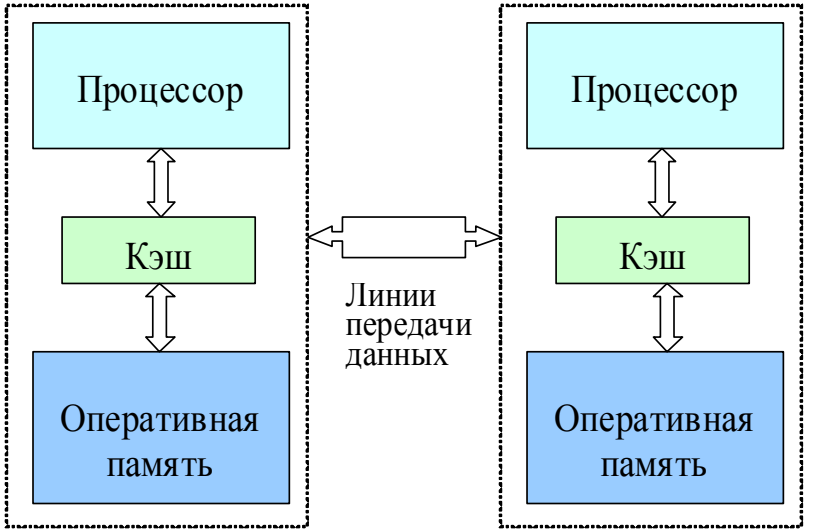
Кулаков Кирилл Александрович

2016
Петрозаводск

Введение

В вычислительных системах с распределенной памятью процессоры работают независимо друг от друга.

Для организации параллельных вычислений необходимо уметь:



– *распределять* вычислительную нагрузку,
– *организовать* информационное взаимодействие (*передачу данных*) между процессорами.

Решение всех перечисленных вопросов обеспечивает MPI -интерфейс передачи данных (message passing interface)

Введение...

- В рамках MPI для решения задачи разрабатывается одна программа, она запускается на выполнение одновременно на всех имеющихся процессорах
- Для организации различных вычислений на разных процессорах:
  - Есть возможность подставлять разные данные для программы на разных процессорах,
  - Имеются средства для идентификации процессора, на котором выполняется программа
- Такой способ организации параллельных вычислений обычно именуется как *модель "одна программа множество процессов"* (*single program multiple processes* or *SPMP*)

Введение…

- В MPI существует множество операций передачи данных:
  - Обеспечиваются разные способы пересылки данных,
  - Реализованы практически все основные коммуникационные операции.

Эти возможности являются наиболее сильной стороной MPI (об этом, в частности, свидетельствует и само название MPI)

Введение…

Что означает MPI?

- MPI - это стандарт, которому должны удовлетворять средства организации передачи сообщений.
- MPI – это программные средства, которые обеспечивают возможность передачи сообщений и при этом соответствуют всем требованиям стандарта MPI:
  - программные средства должны быть организованы в виде библиотек программных модулей (*библиотеки MPI*),
  - должны быть доступны для наиболее широко используемых алгоритмических языков C и Fortran.

Введение…

Достоинства MPI

- MPI позволяет существенно снизить остроту проблемы переносимости параллельных программ между разными компьютерными системами.
- MPI содействует повышению эффективности параллельных вычислений - практически для каждого типа вычислительных систем существуют реализации библиотек MPI.
- MPI уменьшает сложность разработки параллельных программ:
  - большая часть основных операций передачи данных предусматривается стандартом MPI,
  - имеется большое количество библиотек параллельных методов, созданных с использованием MPI.

Введение

История разработки MPI

1992 г. Начало работ над стандартом библиотеки передачи сообщений (Oak Ridge National Laboratory, Rice University).

Ноябрь 1992 г. Объявление рабочего варианта стандарта MPI 1.

Ноябрь 1993 г. Обсуждение стандарта на конференции Supercomputing'93.

5 мая 1994 г. Окончательный вариант стандарта MPI 1.0.

12 Июня 1995 г. Новая версия стандарта - MPI 1.1.

18 Июля 1997 г. Опубликован стандарт MPI-2: Extensions to the Message-Passing Interface.

21 Сентября 2012 г. Опубликован стандарт MPI 3.0

Разработка стандарта MPI производится международным консорциумом ***MPI Forum***

MPI: основные понятия и определения…

Понятие параллельной программы

- Под *параллельной программой* в рамках MPI понимается множество одновременно выполняемых *процессов*:
  - Процессы могут выполняться на разных процессорах; вместе с этим, на одном процессоре могут располагаться несколько процессов,
  - Каждый процесс параллельной программы порождается на основе копии одного и того же программного кода (*модель SPMP*).
- Исходный программный код разрабатывается на алгоритмических языках C или Fortran с использованием библиотеки MPI.
- Количество процессов и число используемых процессоров определяется в момент запуска параллельной программы средствами среды исполнения MPI программ. Все процессы программы последовательно перенумерованы. Номер процесса именуется *рангом* процесса.

MPI: основные понятия и определения…

В основу MPI положены четыре основные концепции:

- Тип операции передачи сообщения
- Тип данных, пересылаемых в сообщении
- Понятие коммуникатора (группы процессов)
- Понятие виртуальной топологии

MPI: основные понятия и определения…

Операции передачи данных

- Основу MPI составляют операции передачи сообщений.
- Среди предусмотренных в составе MPI функций различаются:
  - парные (*point-to-point*) операции между двумя процессами,
  - коллективные (*collective*) коммуникационные действия для одновременного взаимодействия нескольких процессов.

MPI: основные понятия и определения…

Понятие коммуникаторов…

- *Коммуникатор* в MPI - специально создаваемый служебный объект, объединяющий в своем составе группу процессов и ряд дополнительных параметров (*контекст)*:
  - парные операции передачи данных выполняются для процессов, принадлежащих одному и тому же коммуникатору,
  - Коллективные операции применяются одновременно для всех процессов коммуникатора.
- Указание используемого коммуникатора является обязательным для операций передачи данных в MPI.

MPI: основные понятия и определения…

Понятие коммуникаторов

- В ходе вычислений могут создаваться новые и удаляться существующие коммуникаторы.
- Один и тот же процесс может принадлежать разным коммуникаторам.
- Все имеющиеся в параллельной программе процессы входят в состав создаваемого по умолчанию коммуникатора с идентификатором MPI_COMM_WORLD.
- При необходимости передачи данных между процессами из разных групп необходимо создавать глобальный коммуникатор (*intercommunicator*).

MPI: основные понятия и определения…

Типы данных

- При выполнении операций передачи сообщений для указания передаваемых или получаемых данных в функциях MPI необходимо указывать тип пересылаемых данных.
- MPI содержит большой набор *базовых типов данных*, во многом совпадающих с типами данных в алгоритмических языках C и Fortran.
- В MPI имеются возможности для создания новых *производных типов данных* для более точного и краткого описания содержимого пересылаемых сообщений.

MPI: основные понятия и определения

Виртуальные топологии

- Логическая топология линий связи между процессами имеет структуру полного графа (независимо от наличия реальных физических каналов связи между процессорами).
- В MPI имеется возможность представления множества процессов в виде *решетки* произвольной размерности При этом, граничные процессы решеток могут быть объявлены соседними и, тем самым, на основе решеток могут быть определены структуры типа *тор*.
- В MPI имеются средства и для формирования логических (виртуальных) топологий любого требуемого типа.

Структура MPI приложения

- Инициализация и завершение MPI программ

  Первой вызываемой функцией MPI должна быть функция:

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

  (служит для инициализации среды выполнения MPI программы; параметрами функции являются количество аргументов в командной строке и текст самой командной строки.)

  Последней вызываемой функцией MPI обязательно должна являться функция:

```
int MPI_Finalize (void)
```

Структура MPI приложения

- Инициализация и завершение MPI программ

  структура параллельной программы, разработанная с использованием MPI, должна иметь следующий вид:

```
#include "mpi.h"
int main ( int argc, char *argv[] ) {
  <программный код без использования MPI функций>
  MPI_Init ( &agrc, &argv );
    <программный код с использованием MPI функций >
  MPI_Finalize();
  <программный код без использования MPI функций >
  return 0;
}
```

Определение количества и ранга процессов

- Определение количества процессов в выполняемой параллельной программе осуществляется при помощи функции:

```
int MPI_Comm_size ( MPI_Comm comm, int *size )
```

- Для определения ранга процесса используется функция:

```
int MPI_Comm_rank ( MPI_Comm comm, int *rank )
```

Определение количества и ранга процессов

- Как правило, вызов функций MPI_Comm_size и MPI_Comm_rank выполняется сразу после MPI_Init:

```
#include "mpi.h"
int main ( int argc, char *argv[] ) {
  int ProcNum, ProcRank;
  <программный код без использования MPI функций>
  MPI_Init ( &agrc, &argv );
  MPI_Comm_size ( MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank ( MPI_COMM_WORLD, &ProcRank);
    <программный код с использованием MPI функций >
  MPI_Finalize();
  <программный код без использования MPI функций >
  return 0;
}
```

Определение количества и ранга процессов

- Коммуникатор MPI_COMM_WORLD создается по умолчанию и представляет все процессы выполняемой параллельной программы;
- Ранг, получаемый при помощи функции MPI_Comm_rank, является рангом процесса, выполнившего вызов этой функции, и, тем самым, переменная ProcRank будет принимать различные значения в разных процессах.

Передача сообщений

- Для передачи сообщения процесс-отправитель должен выполнить функцию:

```
int MPI_Send(void *buf, int count, MPI_Datatype type,
  int dest, int tag, MPI_Comm comm), где

  - buf   – адрес буфера памяти, в котором располагаются
данные отправляемого сообщения,
  - count – количество элементов данных в сообщении,
  - type  - тип элементов данных пересылаемого сообщения,
  - dest  - ранг процесса, которому отправляется сообщение,
  - tag   - значение-тег, используемое для идентификации
сообщений,
  - comm  - коммуникатор, в рамках которого выполняется
передача данных.
```

Передача сообщений

Базовые типы данных MPI для алгоритмического языка C

MPI_Datatype	C Datatype
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	Double
MPI_FLOAT	Float
MPI_INT	Int
MPI_LONG	Long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Передача сообщений

- Отправляемое сообщение определяется через указание блока памяти (буфера), в котором это сообщение располагается. Используемая для указания буфера триада (buf, count, type) входит в состав параметров практически всех функций передачи данных,
- Процессы, между которыми выполняется передача данных, обязательно должны принадлежать коммуникатору, указываемому в функции MPI_Send,
- Параметр tag используется только при необходимости различения передаваемых сообщений, в противном случае в качестве значения параметра может быть использовано произвольное целое число.

Прием сообщений

- Для приема сообщения процесс-получатель должен выполнить функцию:

```
int MPI_Recv(void *buf, int count, MPI_Datatype type,
  int source,int tag, MPI_Comm comm, MPI_Status *status),
где
  - buf, count, type – буфер памяти для приема сообщения
  - source - ранг процесса, от которого должен быть выполнен
             прием сообщения,
  - tag    - тег сообщения, которое должно быть принято для
             процесса,
  - comm   - коммуникатор, в рамках которого выполняется
             передача данных,
  - status – указатель на структуру данных с информацией о
             результате выполнения операции приема данных.
```

Прием сообщений

- Буфер памяти должен быть достаточным для приема сообщения, а тип элементов передаваемого и принимаемого сообщения должны совпадать; при нехватке памяти часть сообщения будет потеряна и в коде завершения функции будет зафиксирована ошибка переполнения,
- При необходимости приема сообщения от любого процесса-отправителя для параметра source может быть указано значение MPI_ANY_SOURCE,
- При необходимости приема сообщения с любым тегом для параметра tag может быть указано значение MPI_ANY_TAG,

Прием сообщений

- Параметр status позволяет определить ряд характеристик принятого сообщения:

```
-status.MPI_SOURCE – ранг процесса-отправителя принятого сообщения,
-status.MPI_TAG    - тег принятого сообщения.
```

- Функция

```
MPI_Get_count(MPI_Status *status, MPI_Datatype type, int
*count )
```

возвращает в переменной count количество элементов типа type в принятом сообщении.

Прием сообщений

- Функция MPI_Recv является блокирующей для процесса-получателя, т.е. его выполнение приостанавливается до завершения работы функции. Таким образом, если по каким-то причинам ожидаемое для приема сообщение будет отсутствовать, выполнение параллельной программы будет блокировано.

Пример передачи сообщений

- Каждый процесс определяет свой ранг, после чего действия в программе разделяются (разные процессы выполняют различные действия)
- Все процессы, кроме процесса с рангом 0, передают значение своего ранга нулевому процессу
- Процесс с рангом 0 сначала печатает значение своего ранга, а далее последовательно принимает сообщения с рангами процессов и также печатает их значения

Пример передачи сообщений

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
  int world_size, world_rank, hello_number;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  if (world_rank == 0) {
    printf("Hello world from process %d\n", world_rank);
    MPI_Status status;
    for (int i = 1; i < world_size; i++) {
      MPI_Recv(&hello_number, 1, MPI_INT, i, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
      printf("Hello world from process %d\n", hello_number);
    }
 } else {
    MPI_Send(&world_rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
 }
  MPI_Finalize();
}
```

Пример передачи сообщений

```
Hello from process 0
Hello from process 2
Hello from process 1
Hello from process 3
```

- Порядок приема сообщений заранее не определен и зависит от условий выполнения параллельной программы (более того, этот порядок может изменяться от запуска к запуску). Если это не приводит к потере эффективности, следует обеспечивать однозначность расчетов и при использовании параллельных вычислений:

```
MPI_Recv(&hello_number, 1, MPI_INT, i, MPI_ANY_TAG,
         MPI_COMM_WORLD, &status)
```

- Указание ранга процесса-отправителя регламентирует порядок приема сообщений.

Структура MPI приложения

- Можно рекомендовать при увеличении объема разрабатываемых программ выносить программный код разных процессов в отдельные программные модули (функции). Общая схема MPI программы в этом случае будет иметь вид:

```
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if ( ProcRank == 0 ) DoProcess0();
else if ( ProcRank == 1 ) DoProcess1();
else if ( ProcRank == 2 ) DoProcess2();
```

Структура MPI сообщения

- Для контроля правильности выполнения все функции MPI возвращают в качестве своего значения код завершения.
- При успешном выполнении функции возвращаемый код равен MPI_SUCCESS. Другие значения кода завершения свидетельствуют об обнаружении тех или иных ошибочных ситуаций в ходе выполнения функций:

```
- MPI_ERR_BUFFER – неправильный указатель на буфер,
- MPI_ERR_COMM – неправильный коммуникатор,
- MPI_ERR_RANK – неправильный ранг процесса
и др.
```

Определение времени выполнение MPI программы

- Необходимо определять время выполнения вычислений для оценки достигаемого ускорения за счет использования параллелизма,
- Получение времени текущего момента выполнения программы обеспечивается при помощи функции:

```
double MPI_Wtime(void)
```

- Точность измерения времени может зависеть от среды выполнения параллельной программы. Для определения текущего значения точности может быть использована функция:

```
double MPI_Wtick(void)
```

- (время в секундах между двумя последовательными показателями времени аппаратного таймера используемой системы)

Коллективные операции передачи данных

- Будем использовать учебную задачу суммирования элементов вектора x:

$$S = \sum_{i=1}^{n} x_i$$

- Для решения необходимо разделить данные на равные блоки, передать эти блоки процессам, выполнить в процессах суммирование полученных данных, собрать значения вычисленных частных сумм на одном из процессов и сложить значения частичных сумм для получения общего результата решаемой задачи,
- Для более простого изложения примера процессам программы будут передаваться весь суммируемый вектор, а не отдельные блоки этого вектора.

Передача данных от одного процесса всем процессам программы

- Необходимо передать значения вектора x всем процессам параллельной программы,
- Можно воспользоваться рассмотренными ранее функциями парных операций передачи данных:

```
MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
for (i=1; i<ProcNum; i++)
  MPI_Send(&x,n,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
```

- Повторение операций передачи приводит к суммированию затрат (латентностей) на подготовку передаваемых сообщений,
- Данная операция может быть выполнена за меньшее число операций передачи данных.

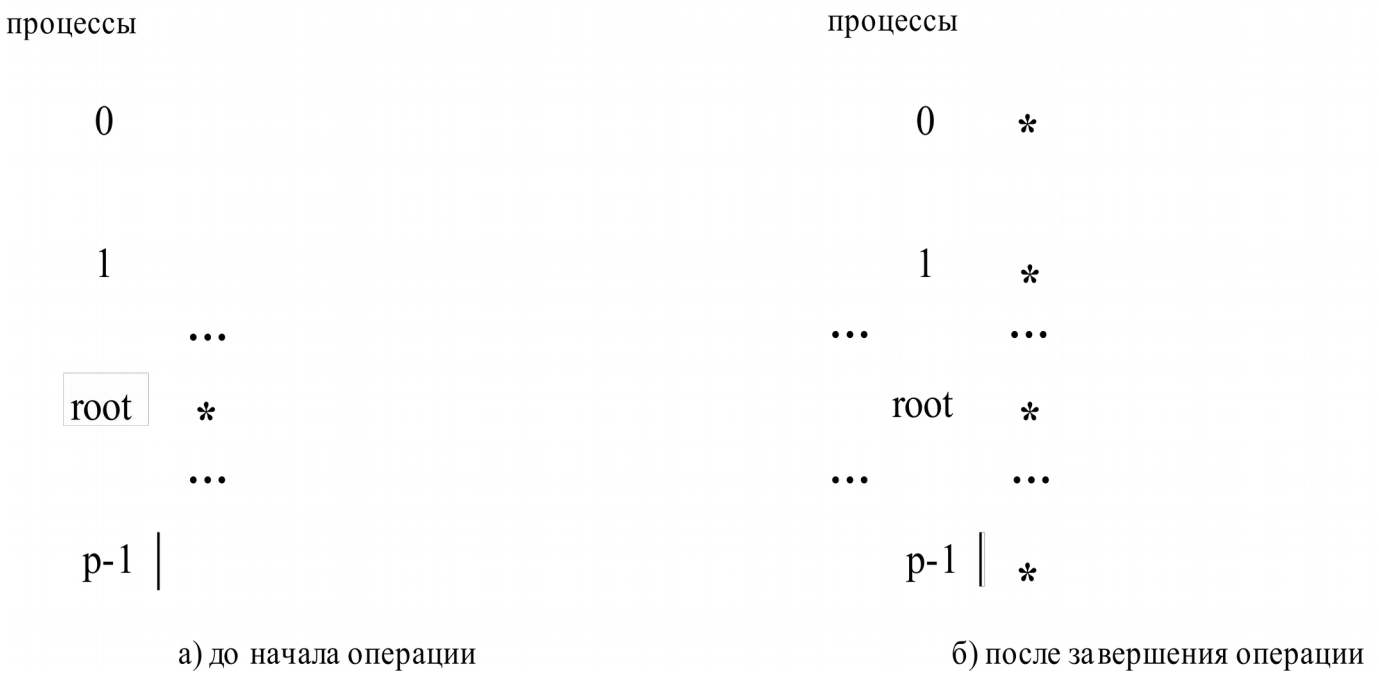
Передача данных от одного процесса всем процессам программы

- Широковещательная рассылка данных может быть обеспечена при помощи функции MPI:

```
int MPI_Bcast(void *buf,int count,MPI_Datatype
  type, int root,MPI_Comm comm),
где
- buf, count, type – буфер памяти с отправляемым
  сообщением (для процесса с рангом 0), и для
  приема сообщений для всех остальных процессов,
- root - ранг процесса, выполняющего рассылку
  данных,
- comm - коммуникатор, в рамках которого
  выполняется передача данных.
```

Передача данных от одного процесса всем процессам программы

- Функция MPI_Bcast осуществляет рассылку данных из буфера buf, содержащего count элементов типа type с процесса, имеющего номер root, всем процессам, входящим в коммуникатор comm



а) до начала операции

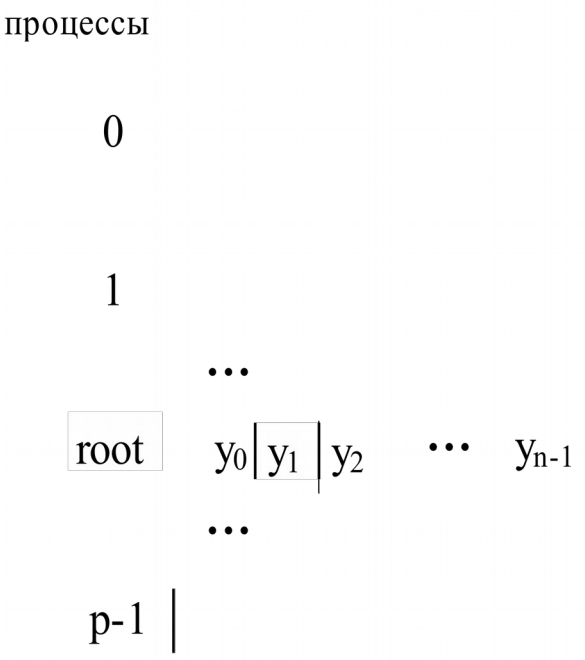
б) после завершения операции

Передача данных от одного процесса всем процессам программы

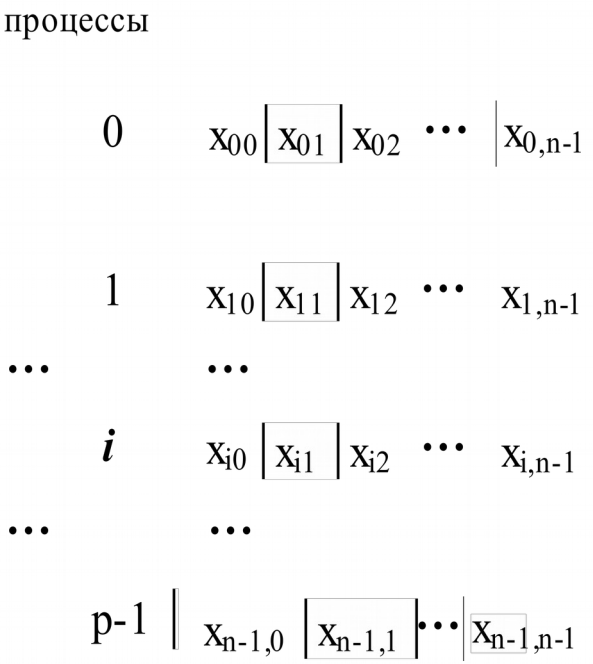
- Функция MPI_Bcast определяет коллективную операцию, вызов функции MPI_Bcast должен быть осуществлен всеми процессами указываемого коммуникатора,
- Указываемый в функции MPI_Bcast буфер памяти имеет различное назначение в разных процессах:
  - Для процесса с рангом root, с которого осуществляется рассылка данных, в этом буфере должно находиться рассылаемое сообщение.
  - Для всех остальных процессов указываемый буфер предназначен для приема передаваемых данных.

Передача данных от всех процессов одному процессу

- Процедура сбора и последующего суммирования данных является примером часто выполняемой коллективной операции передачи данных от всех процессов одному процессу, в которой над собираемыми значениями осуществляется та или иная обработка данных.



а) после завершения операции



б) до начала операции

$$y_j = \bigotimes_{i=0}^{n-1} x_{ij}, \; 0 \le j < n$$

Передача данных от всех процессов одному процессу

```
int MPI_Reduce(void *sendbuf, void *recvbuf,int count,
      MPI_Datatype type, MPI_Op op,int root,MPI_Comm
comm),
```

где

- **sendbuf** - буфер памяти с отправляемым сообщением,
- **recvbuf** – буфер памяти для результирующего сообщения (только для процесса с рангом root),
- **count** - количество элементов в сообщениях,
- **type** – тип элементов сообщений,
- **op** - операция, которая должна быть выполнена над данными,
- **root** - ранг процесса, на котором должен быть получен результат,
- **comm** - коммуникатор, в рамках которого выполняется операция.

Передача данных от всех процессов одному процессу

- Типы операций MPI для функций редукции данных…

Операция	**Описание**
MPI_MAX	Определение максимального значения
MPI_MIN	Определение минимального значения
MPI_SUM	Определение суммы значений
MPI_PROD	Определение произведения значений
MPI_LAND	Выполнение логической операции "И" над значениями сообщений
MPI_BAND	Выполнение битовой операции "И" над значениями сообщений
MPI_LOR	Выполнение логической операции "ИЛИ" над значениями сообщений
MPI_BOR	Выполнение битовой операции "ИЛИ" над значениями сообщений
MPI_LXOR	Выполнение логической операции исключающего "ИЛИ" над значениями сообщений
MPI_BXOR	Выполнение битовой операции исключающего "ИЛИ" над значениями сообщений
MPI_MAXLOC	Определение максимальных значений и их индексов
MPI_MINLOC	Определение минимальных значений и их индексов

Передача данных от всех процессов одному процессу

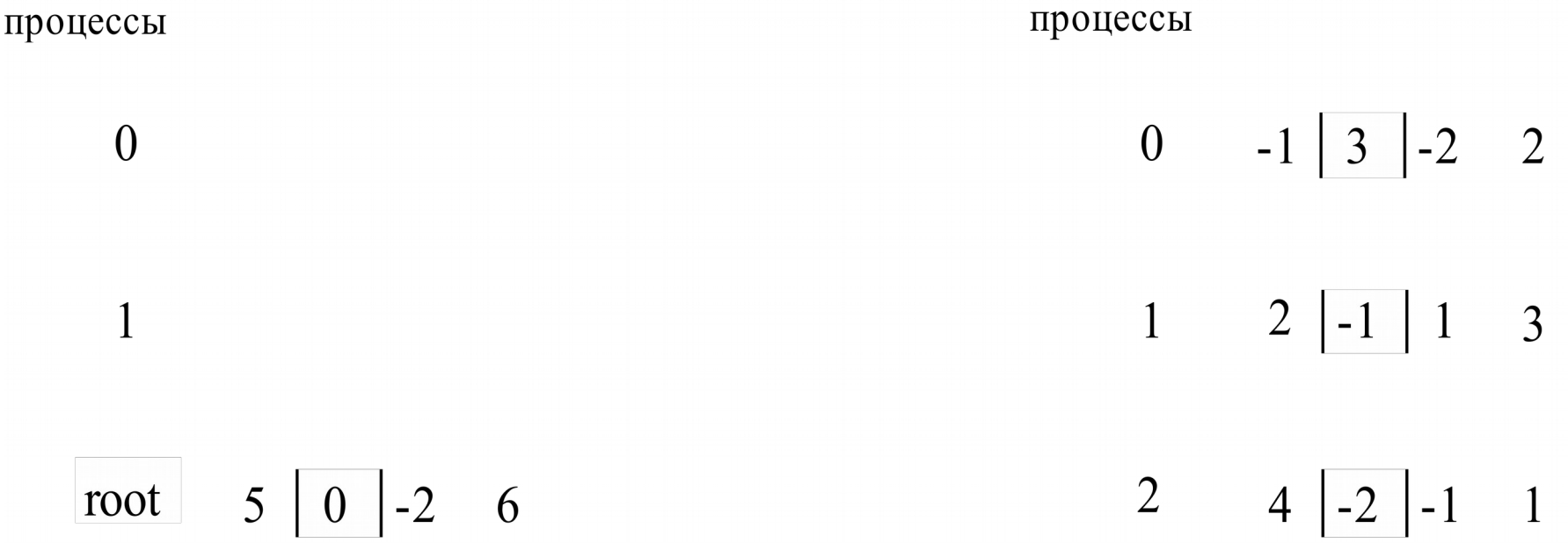
- MPI_MAX и MPI_MIN ищут поэлементные максимум и минимум;
- MPI_SUM вычисляет поэлементную сумму векторов;
- MPI_PROD вычисляет поэлементное произведение векторов;
- MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR - логические и двоичные операции И, ИЛИ, исключающее ИЛИ;
- MPI_MAXLOC, MPI_MINLOC - поиск индексированного минимума/максимума

Передача данных от всех процессов одному процессу

- Функция MPI_Reduce определяет коллективную операцию и, тем самым, вызов функции должен быть выполнен всеми процессами указываемого коммуникатора, все вызовы функции должны содержать одинаковые значения параметров count, type, op, root, comm,
- Передача сообщений должна быть выполнена всеми процессами, результат операции будет получен только процессом с рангом root,
- Выполнение операции редукции осуществляется над отдельными элементами передаваемых сообщений.

Передача данных от всех процессов одному процессу

- Пример для операции суммирования



а) после завершения операции

б) до начала операции

Передача данных от всех процессов одному процессу

- Функция MPI_Reduce определяет коллективную операцию и, тем самым, вызов функции должен быть выполнен всеми процессами указываемого коммуникатора, все вызовы функции должны содержать одинаковые значения параметров count, type, op, root, comm,
- Передача сообщений должна быть выполнена всеми процессами, результат операции будет получен только процессом с рангом root,
- Выполнение операции редукции осуществляется над отдельными элементами передаваемых сообщений.

Синхронизация вычислений

- Синхронизация процессов, т.е. одновременное достижение процессами тех или иных точек процесса вычислений, обеспечивается при помощи функции MPI:

```
int MPI_Barrier(MPI_Comm comm);
```

- Функция MPI_Barrier определяет коллективную операцию, при использовании должна вызываться всеми процессами коммуникатора.
- Продолжение вычислений любого процесса произойдет только после выполнения функции MPI_Barrier всеми процессами коммуникатора.

Группы процессов

- Процессы параллельной программы объединяются в группы. В группу могут входить все процессы параллельной программы или в группе может находиться только часть имеющихся процессов. Один и тот же процесс может принадлежать нескольким группам,
- Управление группами процессов предпринимается для создания на их основе коммуникаторов,
- Группы процессов могут быть созданы только из уже существующих групп. В качестве исходной группы может быть использована группа, связанная с предопределенным коммуникатором MPI_COMM_WORLD:

```
int MPI_Comm_group ( MPI_Comm comm, MPI_Group *group )
```

Группы процессов

- На основе существующих групп, могут быть созданы новые группы
  - создание новой группы newgroup из существующей группы oldgroup, которая будет включать в себя n процессов, ранги которых перечисляются в массиве ranks:

```
int MPI_Group_incl(MPI_Group oldgroup,int n, int *ranks,
                   MPI_Group *newgroup);
```

  - создание новой группы newgroup из группы oldgroup, которая будет включать в себя n процессов, ранги которых не совпадают с рангами, перечисленными в массиве ranks:

```
int MPI_Group_excl(MPI_Group oldgroup,int n, int *ranks,
                   MPI_Group *newgroup);
```

Группы процессов

- На основе существующих групп, могут быть созданы новые группы
  - создание новой группы newgroup как объединения групп group1 и group2:

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
        MPI_Group *newgroup);
```

  - создание новой группы newgroup как пересечения групп group1 и group2:

```
int MPI_Group_intersection ( MPI_Group group1,
      MPI_Group group2, MPI_Group *newgroup );
```

  - создание новой группы newgroup как разности групп group1 и group2:

```
int MPI_Group_difference ( MPI_Group group1,
      MPI_Group group2, MPI_Group *newgroup );
```

Группы процессов

- Получение информации о группе процессов:
  - получение количества процессов в группе:

```
int MPI_Group_size ( MPI_Group group, int *size );
```

  - получение ранга текущего процесса в группе:

```
int MPI_Group_rank ( MPI_Group group, int *rank );
```

- После завершения использования группа должна быть удалена:

```
int MPI_Group_free ( MPI_Group *group );
```

Группы процессов

- Процесс может входить в несколько групп. Подпрограмма MPI_Group_translate_ranks преобразует ранг процесса в одной группе в его ранг в другой группе:

  int MPI_Group_translate_ranks(MPI_Group group1, int n, int *ranks1, MPI_Group group2, int *ranks2)

  MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierr)

- Эта функция используется для определения относительной нумерации одних и тех же процессов в двух разных группах.

Группы процессов

Подпрограмма `MPI_Group_compare` используется для сравнения групп `group1` и `group2`:

```
int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int
*result)

MPI_Group_compare(group1, group2, result, ierr)
```

Результат выполнения этой подпрограммы:

- Если группы полностью совпадают, возвращается значение `MPI_IDENT`.
- Если члены обеих групп одинаковы, но их ранги отличаются, результатом будет значение `MPI_SIMILAR`.
- Если группы различны, результатом будет `MPI_UNEQUAL`.

Группы процессов

Подпрограмма `MPI_Group_compare` используется для сравнения групп `group1` и `group2`:

```
int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int
*result)

MPI_Group_compare(group1, group2, result, ierr)
```

Результат выполнения этой подпрограммы:

- Если группы полностью совпадают, возвращается значение `MPI_IDENT`.
- Если члены обеих групп одинаковы, но их ранги отличаются, результатом будет значение `MPI_SIMILAR`.
- Если группы различны, результатом будет `MPI_UNEQUAL`.

Коммуникаторы

- Под коммуникатором в MPI понимается специально создаваемый служебный объект, объединяющий в своем составе группу процессов и ряд дополнительных параметров (контекст), используемых при выполнении операций передачи данных,
- Будем рассматривать управление интракоммуникаторами, используемыми для операций передачи данных внутри одной группы процессов.

Коммуникаторы

- Создание коммуникатора:
  - дублирование уже существующего коммуникатора:

```
int MPI_Comm_dup ( MPI_Comm oldcom, MPI_comm *newcomm );
```

  - создание нового коммуникатора из подмножества процессов существующего коммуникатора:

```
int MPI_comm_create (MPI_Comm oldcom, MPI_Group group,
      MPI_Comm *newcomm);
```

- Операция создания коммуникаторов является коллективной и, тем самым, должна выполняться всеми процессами исходного коммуникатора,
- После завершения использования коммуникатор должен быть удален:

```
int MPI_Comm_free ( MPI_Comm *comm );
```

Управление коммуникаторами

Пример создания коммуникатора

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
 char message[24];
 MPI_Group MPI_GROUP_WORLD;
 MPI_Group group;
 MPI_Comm fcomm;
 int size, q, proc;
 int* process_ranks;
 int rank, rank_in_group;
 MPI_Status status;

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

Управление коммуникаторами

```
 printf("New group contains processes:");
 q = size – 1;
 process_ranks = (int*) malloc(q*sizeof(int));
 for (proc = 0; proc < q; proc++)
{
 process_ranks[proc] = proc;
 printf("%i ", process_ranks[proc]);
 }
 printf("\n");
 MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
 MPI_Group_incl(MPI_GROUP_WORLD, q, process_ranks, &group);
 MPI_Comm_create(MPI_COMM_WORLD, group, &fcomm);
 if (fcomm != MPI_COMM_NULL) {
 MPI_Comm_group(fcomm, &group);
 MPI_Comm_rank(fcomm, &rank_in_group);
```

Управление коммуникаторами

```
 if (rank_in_group == 0) {
 strcpy(message, "Hi, Parallel Programmer!");
 MPI_Bcast(&message, 25, MPI_BYTE, 0, fcomm);
 printf("0 send: %s\n", message);
  }
else
{
 MPI_Bcast(&message, 25, MPI_BYTE, 0, fcomm);
 printf("%i received: %s\n", rank_in_group, message);
}
 MPI_Comm_free(&fcomm);
 MPI_Group_free(&group);
}
 MPI_Finalize();
 return 0;
}
```

Управление коммуникаторами

Эта программа работает следующим образом. Пусть в коммуникатор `MPI_COMM_WORLD` входят `p` процессов. Сначала создается список процессов, которые будут входить в область взаимодействия нового коммуникатора. Затем создается группа, состоящая из этих процессов. Для этого требуются две операции. Первая определяет группу, связанную с коммуникатором `MPI_COMM_WORLD`. Новая группа создается вызовом подпрограммы `MPI_Group_incl`. Затем создается новый коммуникатор. Для этого используется подпрограмма `MPI_Comm_create`. Новый коммуникатор – `fcomm`. В результате всех этих действий все процессы, входящие в коммуникатор `fcomm`, смогут выполнять операции коллективного обмена, но только между собой.

Результат выполнения программы:

Управление коммуникаторами

```
[nemnugin@pd00 ~]$ mpiexec -n 8 ./a.out
New group contains processes:0 1 2 3 4 5 6
New group contains processes:0 1 2 3 4 5 6
New group contains processes:0 1 2 3 4 5 6
New group contains processes:0 1 2 3 4 5 6
New group contains processes:0 1 2 3 4 5 6
New group contains processes:0 1 2 3 4 5 6
New group contains processes:0 1 2 3 4 5 6
New group contains processes:0 1 2 3 4 5 6
New group contains processes:0 1 2 3 4 5 6
0 send: Hi, Parallel Programmer!
2 received: Hi, Parallel Programmer!
1 received: Hi, Parallel Programmer!
6 received: Hi, Parallel Programmer!
3 received: Hi, Parallel Programmer!
5 received: Hi, Parallel Programmer!
4 received: Hi, Parallel Programmer!
[nemnugin@pd00 ~]$
```

Коммуникаторы

- Одновременное создание нескольких коммуникаторов :

```
int MPI_Comm_split ( MPI_Comm oldcomm, int split, int key,
  MPI_Comm *newcomm ),
где
- oldcomm – исходный коммуникатор,
- split   – номер коммуникатора, которому должен принадлежать
            процесс,
- key     – порядок ранга процесса в создаваемом
коммуникаторе,
- newcomm – создаваемый коммуникатор
```

- Вызов функции MPI_Comm_split должен быть выполнен в каждом процессе коммуникатора oldcomm,
- Процессы разделяются на непересекающиеся группы с одинаковыми значениями параметра split. На основе сформированных групп создается набор коммуникаторов. Порядок нумерации процессов соответствует порядку значений параметров key (процесс с большим значением параметра key будет иметь больший ранг).

Управление коммуникаторами

Если процессы A и B вызывают `MPI_Comm_split` с одинаковым значением `split`, а аргумент `rank`, переданный процессом A, меньше, чем аргумент, переданный процессом B, ранг A в группе, соответствующей новому коммуникатору, будет меньше ранга процесса B. Если же в вызовах используется одинаковое значение `rank`, система присвоит ранги произвольно. Для каждой подгруппы создается собственный коммуникатор `newcomm`.

`MPI_Comm_split` – коллективная подпрограмма, ее должны вызвать все процессы из старого коммуникатора, даже если они не войдут в новый коммуникатор. Для этого в качестве аргумента `split` в подпрограмму передается предопределенная константа `MPI_UNDEFINED`. Соответствующие процессы вернут в качестве нового коммуникатора значение `MPI_COMM_NULL`. Новые коммуникаторы, созданные подпрограммой `MPI_Comm_split`, не пересекаются.

Управление коммуникаторами

В следующем примере создаются три новых коммуникатора (если исходный коммуникатор `comm` содержит не менее трех процессов):

```
MPI_Comm comm, newcomm;
int rank, split;
MPI_Comm_rank(comm, &rank);
split = rank%3;
MPI_Comm_split(comm, split, rank, &newcomm);
```

Если количество процессов 9, каждый новый коммуникатор будет содержать 3 процесса. Это можно использовать, например, для того, чтобы расщепить двумерную решетку 3x3, с каждым узлом которой связан один процесс, на 3 одномерных подрешетки.

Управление коммуникаторами

В следующем примере процессы разбиваются на две группы. Одна содержит процессы с чётными рангами, а другая – с нечётными.

```
#include "stdio.h"
#include "mpi.h"

void main(int argc, char *argv[])
{
      int num, p;
      int Neven, Nodd, members[6], even_rank, odd_rank;
      MPI_Group group_world, even_group, odd_group;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &num);
      MPI_Comm_size(MPI_COMM_WORLD, &p);
```

Управление коммуникаторами

```
      Neven = (p + 1)/2;
      Nodd = p - Neven;
      members[0] = 2;
      members[1] = 0;
      members[2] = 4;
      MPI_Comm_group(MPI_COMM_WORLD, &group_world);
      MPI_Group_incl(group_world, Neven, members,
&even_group);
      MPI_Group_excl(group_world, Neven, members,
&odd_group);
      MPI_Barrier(MPI_COMM_WORLD);
      if(num == 0) {
       printf("Number of processes is %d\n", p);
       printf("Number of odd processes is %d\n", Nodd);
       printf("Number of even processes is %d\n", Neven);
       printf("members[0] is assigned rank %d\n", members[0]);
       printf("members[1] is assigned rank %d\n", members[1]);
       printf("members[2] is assigned rank %d\n", members[2]);
       printf("\n");
       printf("     num     even      odd\n");
      }
```

Управление коммуникаторами

```
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Group_rank(even_group, &even_rank);
    MPI_Group_rank( odd_group,  &odd_rank);
    printf("%8d %8d %8d\n",num, even_rank, odd_rank);
    MPI_Finalize();
}
```

Управление коммуникаторами

Результат выполнения:

```
[nemnugin@pd00 ~]$ mpiexec -n 4 ./a.out
Number of processes is 4
Number of odd processes is 2
Number of even processes is 2
members[0] is assigned rank 2
members[1] is assigned rank 0
members[2] is assigned rank 4

      Iam       even        odd
        0          1   -32766
        2          0   -32766
        1   -32766          0
        3   -32766          1
[nemnugin@pd00 ~]$
```

Управление коммуникаторами

Сравнение двух коммуникаторов (`comm1`) и (`comm2`) выполняется подпрограммой `MPI_Comm_compare`:

```
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int
*result)

MPI_Comm_compare(comm1, comm2, result, ierr)
```

Результат ее выполнения `result` – целое значение, которое равно:

- `MPI_IDENT`, если контексты и группы коммуникаторов совпадают;
- `MPI_CONGRUENT`, если совпадают только группы;
- `MPI_UNEQUAL`, если не совпадают ни группы, ни контексты.

В качестве аргумента нельзя использовать пустой коммуникатор `MPI_COMM_NULL`.

Управление коммуникаторами

К числу операций управления коммуникаторами можно отнести операции `MPI_Comm_size` и `MPI_Comm_rank`. Они позволяют, в частности, распределить роли между процессами в модели «хозяин – работник» (master-slave).

Управление коммуникаторами

С помощью подпрограммы `MPI_Comm_set_name` можно присвоить коммуникатору `comm` строковое имя `name`:

```
int MPI_Comm_set_name(MPI_Comm com, char *name)

MPI_Comm_set_name(com, name, ierr)
```

и наоборот, подпрограмма `MPI_Comm_get_name` возвращает `name` – строковое имя коммуникатора `comm`:

```
int MPI_Comm_get_name(MPI_Comm comm, char *name, int *reslen)

MPI_Comm_get_name(comm, name, reslen, ierr)
```

Имя представляет собой массив символьных значений, длина которого должна быть не менее `MPI_MAX_NAME_STRING`. Длина имени – выходной параметр `reslen`.

Управление коммуникаторами

Две области взаимодействия можно объединить в одну. Подпрограмма `MPI_Intercomm_merge` создает интракоммуникатор `newcomm` из интеркоммуникатора `oldcomm`:

```
int MPI_Intercomm_merge(MPI_Comm oldcomm, int high, MPI_Comm
*newcomm)
```

```
MPI_Intercomm_merge(oldcomm, high, newcomm, ierr)
```

Параметр `high` здесь используется для упорядочения групп обоих интракоммуникаторов в `comm` при создании нового коммуникатора.

Управление коммуникаторами

Доступ к удаленной группе, связанной с интеркоммуникатором `comm`, можно получить, обратившись к подпрограмме:

```
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
```

```
MPI_Comm_remote_group(comm, group, ierr)
```

Ее выходным параметром является удаленная группа `group`.

Управление коммуникаторами

Подпрограмма `MPI_Comm_remote_size` определяет размер удаленной группы, связанной с интеркоммуникатором `comm`:

```
int MPI_Comm_remote_size(MPI_Comm comm, int *size)

MPI_Comm_remote_size(comm, size, ierr)
```

Ее выходной параметр `size` – количество процессов в области взаимодействия, связанной с коммуникатором `comm`.

Интеробмены

При выполнении интеробмена процессу-источнику сообщения указывается ранг адресата относительно удаленной группы, а процессу-получателю — ранг источника (также относительно удаленной по отношению к получателю группы). Обмен выполняется между лидерами обеих групп (MPI-1). Предполагается, что в обеих группах есть, по крайней мере, по одному процессу, который может обмениваться сообщениями со своим партнером.

Интеробмены

Интеробмен возможен, только если создан соответствующий интеркоммуникатор, а это можно сделать с помощью подпрограммы:

```
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm
*new_intercomm)

MPI_Intercomm_create(local_comm, local_leader, peer_comm,
remote_leader, tag, new_intercomm, ierr)
```

Входные параметры этой подпрограммы:

- `local_comm` – локальный интракоммуникатор;
- `local_leader` – ранг лидера в локальном коммуникаторе (обычно 0);
- `peer_comm` – удаленный коммуникатор;
- `remote_leader` – ранг лидера в удаленном коммуникаторе (обычно 0);
- `tag` – тег интеркоммуникатора, используемый лидерами обеих групп для обменов в контексте родительского коммуникатора.

Интеробмены

Выходной параметр – интеркоммуникатор (`new_intercomm`). «Джокеры» в качестве параметров использовать нельзя. Вызов этой подпрограммы должен выполняться в обеих группах процессов, которые должны быть связаны между собой. В каждом из этих вызовов используется локальный интракоммуникатор, соответствующий данной группе процессов.

ВНИМАНИЕ

При работе с `MPI_Intercomm_create` локальная и удаленная группы процессов не должны пересекаться, иначе возможны «тупики».

Интеробмены

Пример создания интеркоммуникаторов

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
 int counter, message, myid, numprocs, server;
 int color, remote_leader_rank, i, ICTAG = 0;
 MPI_Status status;
 MPI_Comm oldcommdup, splitcomm, oldcomm, inter_comm;
 MPI_Init(&argc, &argv);
 oldcomm = MPI_COMM_WORLD;
 MPI_Comm_dup(oldcomm, &oldcommdup);
 MPI_Comm_size(oldcommdup, &numprocs);
 MPI_Comm_rank(oldcommdup, &myid);
 server = numprocs – 1;
 color = (myid == server);
 MPI_Comm_split(oldcomm, color, myid, &splitcomm);
```

Интеробмены

```
if(!color) {
 remote_leader_rank = server;
 }
 else {
 remote_leader_rank = 0;
 }
 MPI_Intercomm_create(splitcomm, 0, oldcommdup,
remote_leader_rank, ICTAG, &inter_comm);
 MPI_Comm_free(&oldcommdup);
 if (myid == server) {
 for(i = 0; i<server; i++){
  MPI_Recv(&message, 1, MPI_INT, i, MPI_ANY_TAG, inter_comm,
&status);
  printf("Process rank %i received %i from %i\n", myid,
message, status.MPI_SOURCE);}
}
```

Интеробмены

```
else{
 counter = myid;
 MPI_Send(&counter, 1, MPI_INT,  0, 0, inter_comm);
 printf("Process rank %i send %i\n", myid, counter);
}
 MPI_Comm_free(&inter_comm );
 MPI_Finalize();
}
```

Интеробмены

В этой программе процессы делятся на две группы: первая состоит из одного процесса (процесс с максимальным рангом в исходном коммуникаторе `MPI_COMM_WORLD`), это – «сервер», а во вторую входят все остальные процессы. Между этими группами и создается интеркоммуникатор `inter_comm`. Процессы-клиенты передают серверу сообщения. Сервер принимает эти сообщения с помощью подпрограммы стандартного блокирующего двухточечного приема и выводит их на экран. «Ненужные» коммуникаторы удаляются. Распечатка вывода этой программы выглядит следующим образом:

Интеробмены

```
[nemnugin@pd00 ~]$ mpiexec -n 10 ./a.out
Process rank 0 send 0
Process rank 1 send 1
Process rank 8 send 8
Process rank 5 send 5
Process rank 3 send 3
Process rank 7 send 7
Process rank 9 received 0 from 0
Process rank 9 received 1 from 1
Process rank 4 send 4
Process rank 2 send 2
Process rank 6 send 6
Process rank 9 received 2 from 2
Process rank 9 received 3 from 3
Process rank 9 received 4 from 4
Process rank 9 received 5 from 5
Process rank 9 received 6 from 6
Process rank 9 received 7 from 7
Process rank 9 received 8 from 8
[nemnugin@pd00 ~]$
```

Виртуальные топологии

- Под топологией вычислительной системы понимают структуру узлов сети и линий связи между этими узла. Топология может быть представлена в виде графа, в котором вершины есть процессоры (процессы) системы, а дуги соответствуют имеющимся линиям (каналам) связи.
- Парные операции передачи данных могут быть выполнены между любыми процессами коммуникатора, в коллективной операции принимают участие все процессы коммуникатора. Следовательно, логическая топология линий связи между процессами в параллельной программе имеет структуру полного графа.
- Возможно организовать логическое представление любой необходимой виртуальной топологии. Для этого достаточно сформировать тот или иной механизм дополнительной адресации процессов.

Декартовы топологии (решетки)

- В декартовых топологиях множество процессов представляется в виде прямоугольной решетки, а для указания процессов используется декартова системы координат,
- Для создания декартовой топологии (решетки) в MPI предназначена функция:

```
int MPI_Cart_create(MPI_Comm oldcomm, int ndims, int *dims,
 int *periods, int reorder, MPI_Comm *cartcomm),
где:
- oldcomm  - исходный коммуникатор,
- ndims    - размерность декартовой решетки,
- dims     - массив длины ndims, задает количество процессов в
             каждом измерении решетки,
- periods  - массив длины ndims, определяет, является ли
             решетка периодической вдоль каждого измерения,
 - reorder  - параметр допустимости изменения нумерации
процессов,
 - cartcomm – создаваемый коммуникатор с декартовой топологией
              процессов.
```

Декартовы топологии (решетки)

- Для определения декартовых координат процесса по его рангу можно воспользоваться функцией:

```
int MPI_Card_coords(MPI_Comm comm,int rank,int ndims,int *coords),
где:
 - comm  – коммуникатор с топологией решетки,
 - rank  - ранг процесса, для которого определяются декартовы
координаты,
 - ndims - размерность решетки,
 - coords- возвращаемые функцией декартовы координаты процесса.
```

- Обратное действие – определение ранга процесса по его декартовым координатам – обеспечивается при помощи функции:

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank),
где
 - comm   – коммуникатор с топологией решетки,
 - coords - декартовы координаты процесса,
 - rank   - возвращаемый функцией ранг процесса.
```

Декартовы топологии (решетки)

- Процедура разбиения решетки на подрешетки меньшей размерности обеспечивается при помощи функции:

```
int MPI_Card_sub(MPI_Comm comm, int *subdims, MPI_Comm *newcomm),
где:
 - comm    - исходный коммуникатор с топологией решетки,
 - subdims - массив для указания, какие измерения должны остаться
             в создаваемой подрешетке,
 - newcomm - создаваемый коммуникатор с подрешеткой.
```

- В ходе своего выполнения функция MPI_Cart_sub определяет коммуникаторы для каждого сочетания координат фиксированных измерений исходной решетки.

Декартовы топологии (решетки)

- Дополнительная функция MPI_Cart_shift обеспечивает поддержку операции сдвига данных:
- Циклический сдвиг на k элементов вдоль измерения решетки – в этой операции данные от процесса i пересылаются процессу (i+k) mod dim, где dim есть размер измерения, вдоль которого производится сдвиг,
- Линейный сдвиг на k позиций вдоль измерения решетки – в этом варианте операции данные от процессора i пересылаются процессору i+k (если таковой существует),
- Функция MPI_Cart_shift только определяет ранги процессов, между которыми должен быть выполнен обмен данными в ходе операции сдвига. Непосредственная передача данных, может быть выполнена, например, при помощи функции MPI_Sendrecv.

Декартовы топологии (решетки)

- Функция MPI_Cart_shift обеспечивает получение рангов процессов, с которыми текущий процесс (процесс, вызвавший функцию MPI_Cart_shift) должен выполнить обмен данными:

```
int MPI_Card_shift(MPI_Comm comm, int dir, int disp, int *source,
  int *dst),
где:
 - comm   – коммуникатор с топологией решетки,
 - dir    – номер измерения, по которому выполняется сдвиг,
 - disp   – величина сдвига (<0 – сдвиг к началу измерения),
 - source – ранг процесса, от которого должны быть получены
данные,
 - dst    – ранг процесса которому должны быть отправлены данные.
```

Топология графа

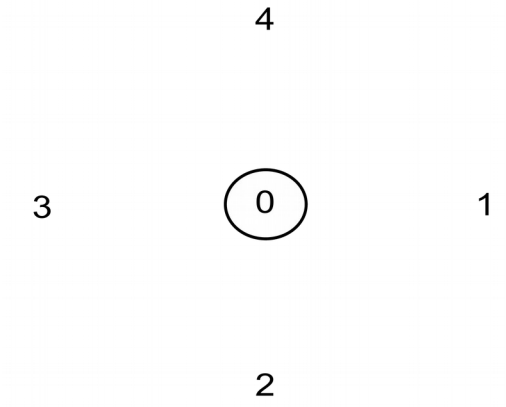
- Создание коммуникатора с топологией типа граф:

```
int MPI_Graph_create(MPI_Comm oldcomm, int nnodes, int *index,
  int *edges, int reorder, MPI_Comm *graphcomm),
где:
 - oldcomm  - исходный коммуникатор,
 - nnodes   - количество вершин графа,
 - index    - количество исходящих дуг для каждой вершины,
 - edges    - последовательный список дуг графа,
 - reorder  - параметр допустимости изменения нумерации
процессов,
 - cartcomm – создаваемый коммуникатор с топологией типа граф.
```

- Операция создания топологии является коллективной и, тем самым, должна выполняться всеми процессами исходного коммуникатора.

Топология графа

- Граф для топологии звезда, количество процессоров равно 5, порядки вершин принимают значения (4,1,1,1,1), а матрица инцидентности имеет вид:



Процессы связи	Линии
0	1,2,3,4
1	0
2	0
3	0
4	0

- Для создания топологии с графом данного вида необходимо выполнить следующий программный код:

```
int index[] = { 4,1,1,1,1 };
int edges[] = { 1,2,3,4,0,0,0,0 };
MPI_Comm StarComm;
MPI_Graph_create(MPI_COMM_WORLD, 5, index, edges, 1, &StarComm);
```

Топология графа

- Топология графа
- Количество соседних процессов, в которых от проверяемого процесса есть выходящие дуги, может быть получено при помощи функции:

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

- Получение рангов соседних вершин обеспечивается функцией:

```
int MPI_Graph_neighbors(MPI_Comm comm,int rank,
      int mneighbors, int *neighbors);
```

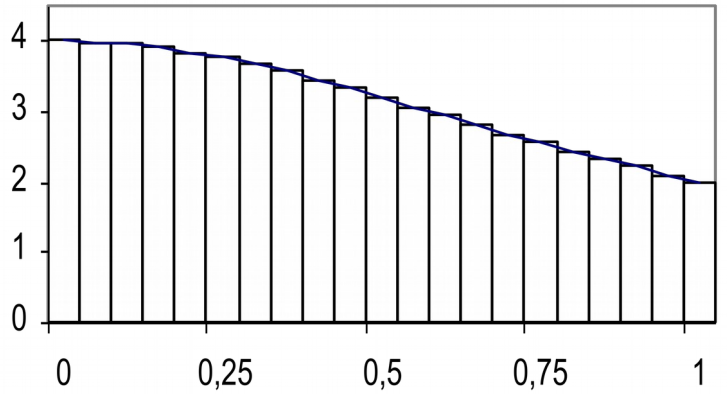
(mneighbors есть размер массива neighbors)

Пример: *Вычисление числа π...*

- Значение числа Пи может быть получено при помощи интеграла

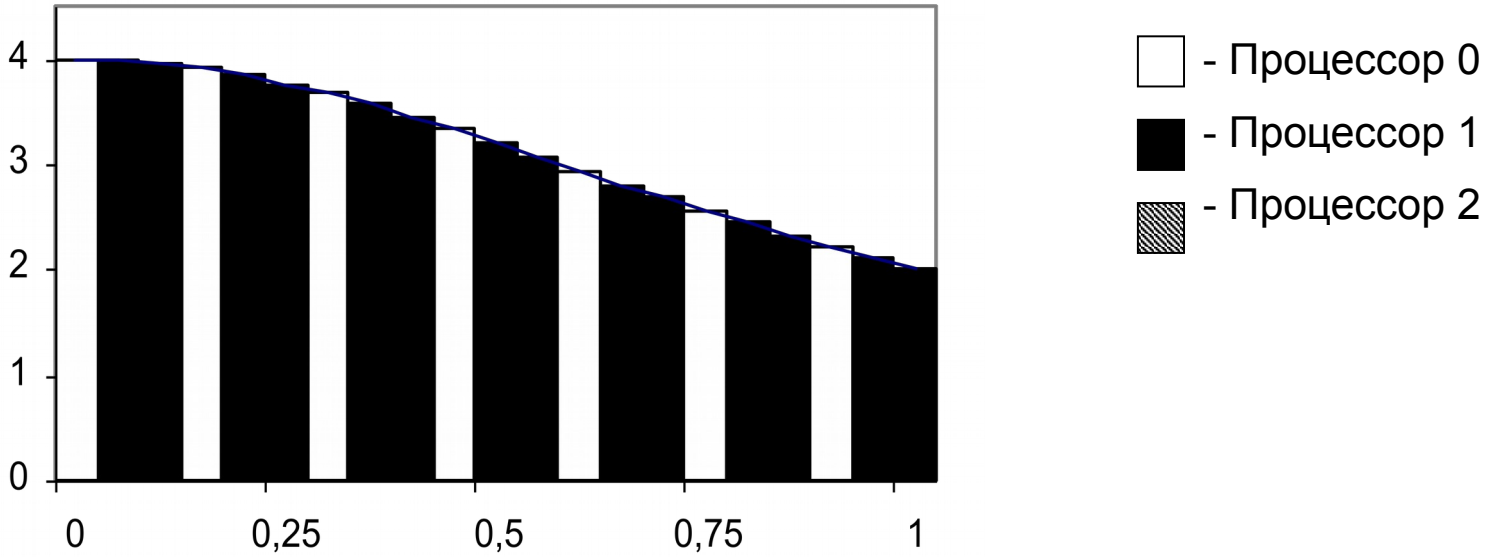
$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

- Для численного интегрирования применим метод прямоугольников



Пример: Вычисление числа π

- Распределим вычисления между p процессорами (циклическая схема)
- Получаемые на отдельных процессорах частные суммы должны быть просуммированы



Пример: Вычисление числа π

```
#include "mpi.h"
#include <math.h>
double f(double a) {
  return (4.0 / (1.0 + a*a));
}
int main(int argc, char *argv) {
  int ProcRank, ProcNum, done = 0, n = 0, i;
  double PI25DT = 3.14159265358979323846264643;
  double mypi, pi, h, sum, x, t1, t2;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD,&ProcRank);
  while (!done ) { // основной цикл вычислений
    if ( ProcRank == 0) {
      printf("Enter the number of intervals: ");
      scanf("%d",&n);
      t1 = MPI_Wtime();
    }
```

Пример: Вычисление числа π

```
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if (n > 0) { // вычисление локальных сумм
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = ProcRank + 1; i <= n; i += ProcNum) {
      x = h * ((double)i  0.5);
      sum += f(x);
    }
    mypi = h * sum;
    // сложение локальных сумм (редукция)
   MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
    if ( ProcRank == 0 ) { // вывод результатов
      t2 = MPI_Wtime();
      printf("pi is approximately %.16f, Error is
                  %.16f\n",pi, fabs(pi  PI25DT));
      printf("wall clock time = %f\n",t2-t1);
    }
  } else done = 1;
 }
 MPI_Finalize();
}
```

Собственные типы данных

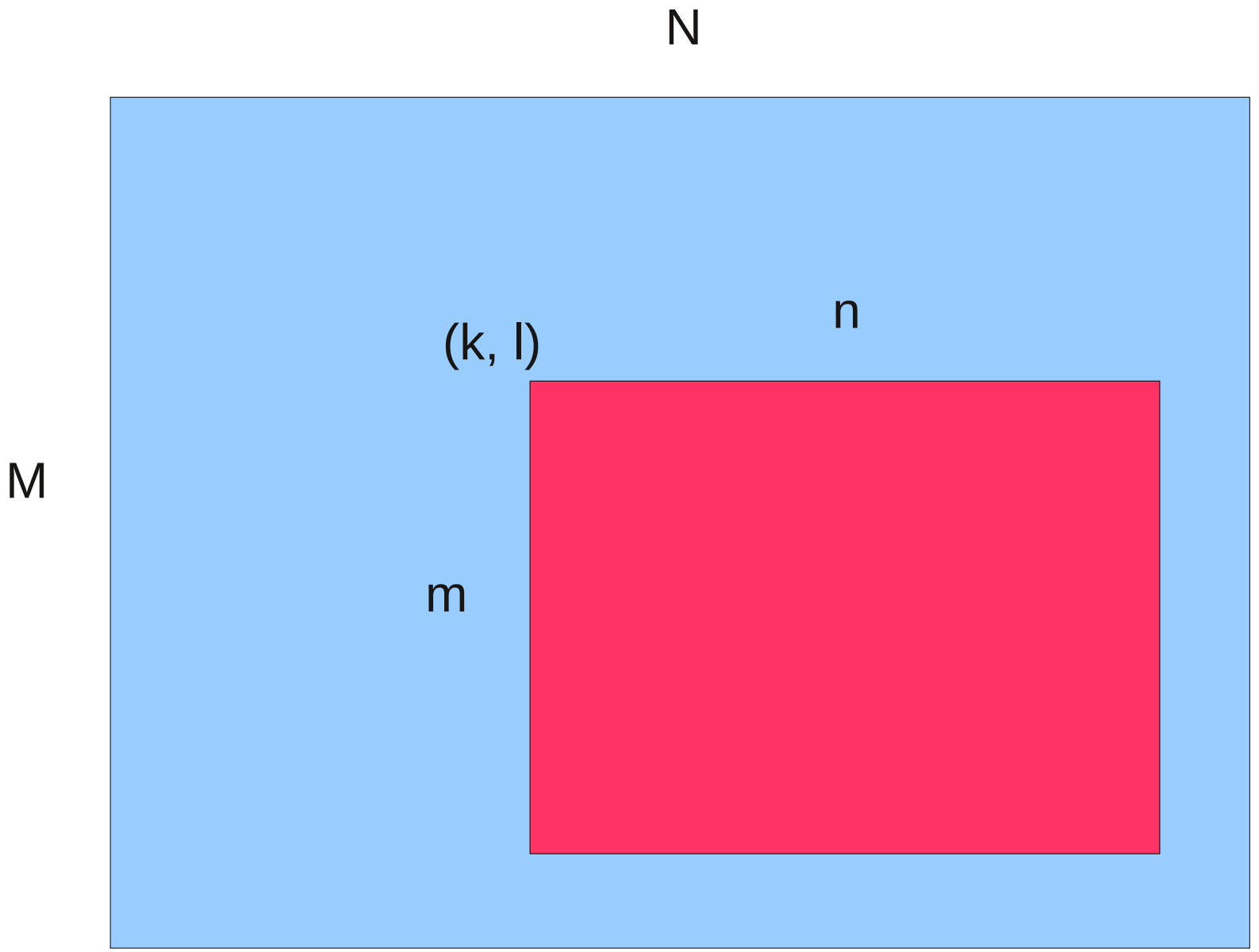
- Традиционно пересылаются числа, массивы, строки
- Иногда требуется пересылка данных не находящихся в памяти последовательно или специальных структур
- Существует несколько способов организации пересылки

Пример: пересылка подматрицы

- Имеется матрица в языке C в виде двумерного массива размером M x N
- Необходимо переслать подматрицу размера m x n
- Подматрица отсчитывается от ячейки с номером (k, l)
- Можно использовать цикл

```
for (i=0; i<n; ++i) {
   MPI_Send(&a[k+i][l], m, MPI_DOUBLE,  dest, tag,
  MPI_COMM_WORLD);
}
```

Пример: пересылка подматрицы



Пример: пересылка подматрицы

- Преимущества такого подхода — простота реализации
- Недостатки — множество сообщений
- Каждое сообщение, независимо от размера имеет оверхед
- Если заменить одно большое сообщение множеством маленьких, то производительность очень сильно уменьшится
- Если не часто встречается в программе, то такое решение может быть приемлемо
- Можно использовать буферизацию данных
- Существуют специальные инструменты в MPI для решения таких задач

Буферизация

- Если данные расположены непоследовательно в памяти можно предварительно скопировать их в буфер
- Для нашего примера:

```
p = &buffer;
for (i=0; i<n; ++i) {
   for(j=0; j<m; ++j) {
     *(p++) = a[i + k][j + l];
   }
}
MPI_Send(p, n*m, MPI_DOUBLE, dest, tag,
   MPI_COMM_WORLD)
```

Недостатки такого подхода

- Накладные расходы на память и на копирование данных
- Возможна пересылка только одного типа данных
- Ухудшает читаемость кода
- При попытках пересылать одни типы данных через другие возможны ошибки

Буферизация в MPI: MPI_Pack

- Используется для пересылок наборов различных данных расположенных в памяти не последовательно
- Заполняет буфер правильным для MPI образом и дает аргументы для MPI_Send
- Копирует данные в буфер и при необходимости транслирует их во внутреннее представление MPI
- После копирования данных в буфер можно пересылать их с типом MPI_PACKED

Пример: упаковка подматицы

```
count = 0
for (i=0; i<n; ++i) {
  MPI_Pack(&a[k+i][l], m, MPI_DOUBLE, buffer, bufsize,
  &count, MPI_COMM_WORLD);
}
MPI_Send(buffer, count, MPI_PACKED, dest, tag,
  MPI_COMM_WORLD);
```

- count изначально выставлен в 0, что говорит о начале заполнения буфера
- Каждый вызов обновляет значение count и конечное значение используется при пересылках

Буферизация в MPI

- MPI_PACKED — специальный тип данных, говорящий о том, что буфер упакован
- MPI_Unpack — используется для распаковки принятых данных
- После упаковки данные могут занимать больше места из-за перевода их в другое представление
- Для определения размера типа в упакованном виде можно использовать MPI_Pack_size

MPI_Pack_size

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

```
int bufsize = 0;
MPI_Pack_size(10, MPI_DOUBLE,
MPI_COMM_WORLD, &bufsize);
buffer = (char *) malloc((unsigned) bufsize);
```

Позволяет узнать размер данных после упаковки для выделения буфера

Производные типы

- Замена MPI_Pack
- Позволяет выполнять упаковку данных на-лету, без выделения дополнительной памяти
- Позволяет сэкономить процессорное время на упаковку данных в память и их трансляцию в понятную для MPI форму
- Позволяет читать и писать непосредственно в рабочие области памяти

Производные типы данных

- Contiguous — несколько копий указанного типа данных
- Vector — несколько копий данных с отступами между блоками
- Indexed — массивом задается мапинг на новый тип
- Struct — мапинг на области памяти, в частности на структуры языка C

Последовательность работы с типами

- Создание типа с помощью функций MPI:
  - MPI_Type_contiguous, MPI_Type_vector,
  - MPI_Type_struct, MPI_Type_indexed,
  - MPI_Type_hvector, MPI_Type_hindexed
- Определить тип (commit) для создания внутренних буфером MPI для данного типа
- Использование своего типа в функциях приема/передачи и т. д.
- Очистить тип данных после использования:
  - MPI_Type_Free(newtype, ierr)

MPI_Contiguous

- Простейший тип, состоящий из наборов элементов заданного типа последовательно идущих в памяти
- C:

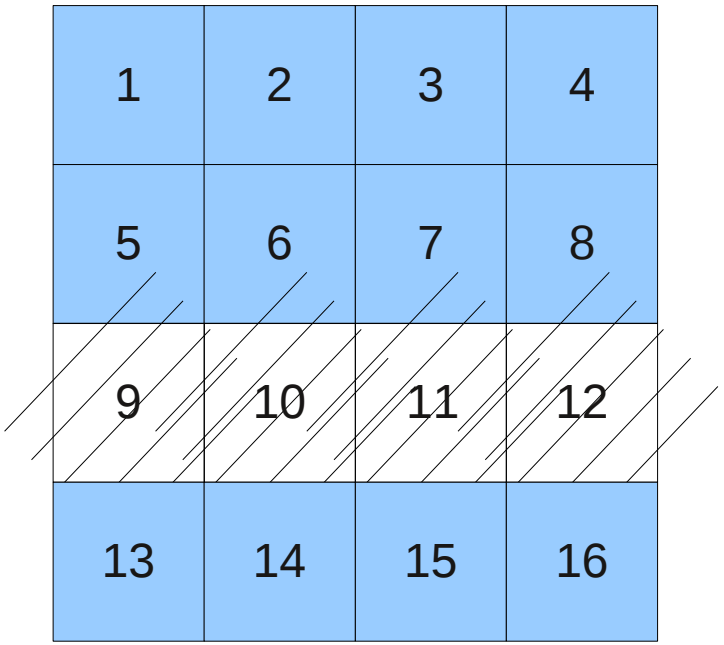
  int MPI_Type_contiguous (int count,MPI_datatype oldtype,MPI_Datatype *newtype)

- Fortran:
  - MPI_TYPE_CONTIGUOUS(COUNT,OLDTYPE,NEWTYPE,IERROR)
  - INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR

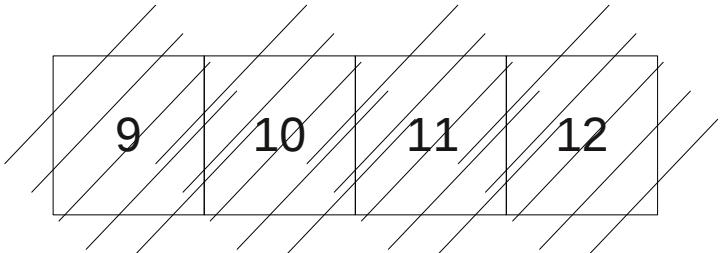
MPI_Type_contiguous

MPI_Type_contiguous (count,oldtype,&newtype)

count = 4;



a[4][4];



1 элемент rowtype &a[2][0]

Пример

```
#include <stdio.h>
#include<mpi.h>
int main(int argc, char *argv[]) {
        int rank;
        struct {
                int x;
                int y;
                int z;
        } point;
        MPI_Datatype ptype;
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Type_contiguous(3,MPI_INT,&ptype);
        MPI_Type_commit(&ptype);
        if(rank==3){
                point.x=15; point.y=23; point.z=6;
                MPI_Send(&point,1,ptype,1,52,MPI_COMM_WORLD);
        } else if(rank==1) {
                MPI_Recv(&point,1,ptype,3,52,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                printf("P:%d received coords are (%d,%d,%d) \n",rank,point.x,point.y,point.z);
        }
        MPI_Finalize();
}
```

```
P:1 received coords are (15,23,6)
```

MPI_Vector

- Пользователь задает расположение элементов старого типа в памяти
- C:

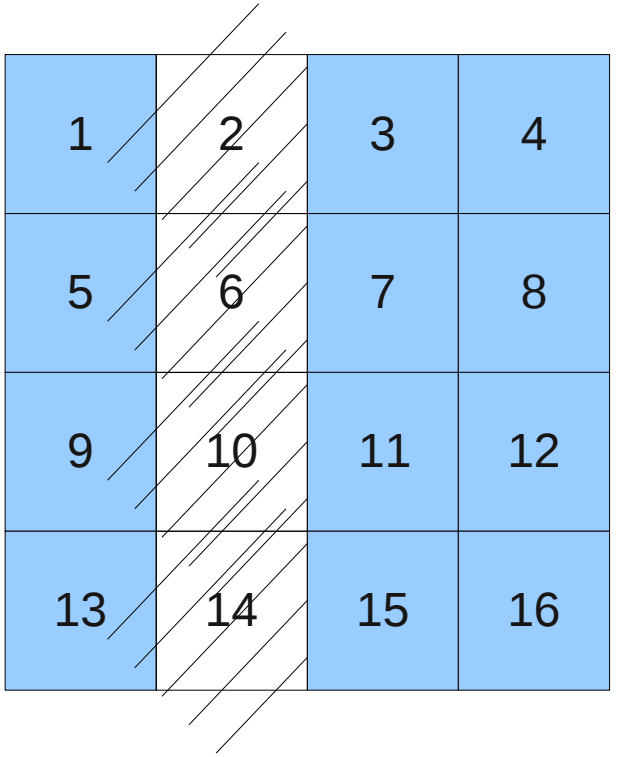
  int MPI_Type_vector(int count, int blocklength, int stride,MPI_Datatype oldtype, MPI_Datatype *newtype)
- Fortran:

  CALL MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE,OLDTYPE, NEWTYPE, IERROR)
- Новый тип состоит из count блоков каждый из которых состоит из blocklength элементов старого типа
- Отступы меду типами задаются stride

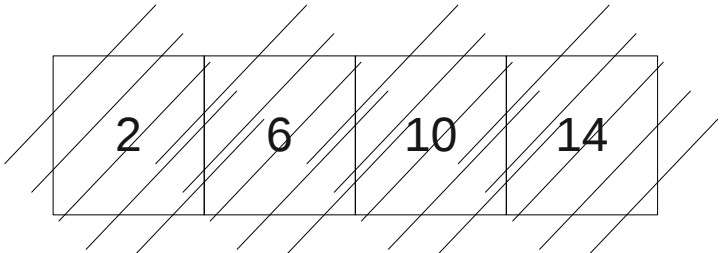
MPI_Type_vector

MPI_Type_vector (count,blocklength,stride,oldtype,&newtype)

count = 4;
blocklength = 1;
stride = 4;



a[4][4];



1 элемент newtype &a[0][1]

Пример

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
      int rank,i,j;
      MPI_Status status;
      double x[4][8];
      MPI_Datatype coltype;
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD,&rank);
      MPI_Type_vector(4,1,8,MPI_DOUBLE,&coltype);
      MPI_Type_commit(&coltype);
      if(rank==3){
            for(i=0;i<4;++i)
            for(j=0;j<8;++j) x[i][j]=pow(10.0,i+1)+j;
            MPI_Send(&x[0][7],1,coltype,1,52,MPI_COMM_WORLD);
      } else if(rank==1) {
            MPI_Recv(&x[0][2],1,coltype,3,52,MPI_COMM_WORLD,&status);
            for(i=0;i<4;++i)printf("P:%d my x[%d][2]=%1f\n",rank,i,x[i][2]);
      }
      MPI_Finalize();
}
```

```
P:1 my x[0][2]=17.000000
P:1 my x[1][2]=107.000000
P:1 my x[2][2]=1007.000000
P:1 my x[3][2]=10007.000000
```

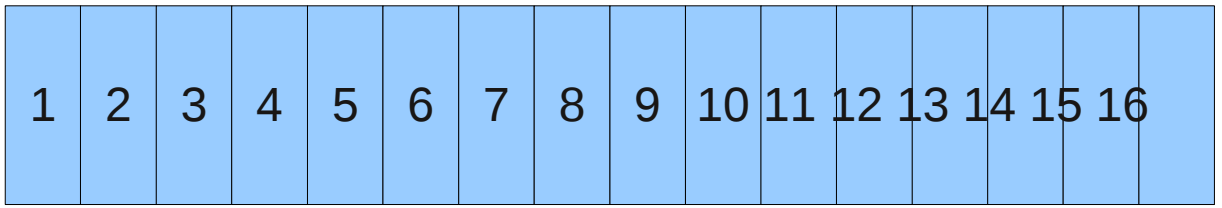
MPI_Indexed

- Позволяют задавать блоки различной длины через различные отступы
- int MPI_Type_indexed(int count,int blocklens[], int indices[],MPI_Datatype old_type, MPI_Datatype *newtype)
- count - число блоков, а также размер входных массивов
- blocklens — число элементов в каждом блоке
- indices — отступ у каждого следующего типа

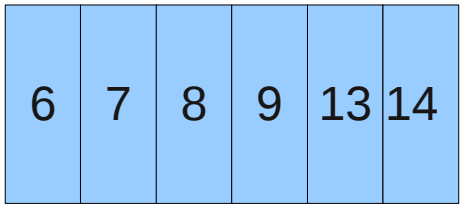
MPI_Type_indexed

MPI_Type_indexed (count,blocklens[],offsets[],old_type,&newtype)

count = 2; blocklengths[0] = 4; blocklengths[1] = 2;
displasements[0] = 5; displaysments[1] = 12;



a[16];



newtype a[0];

Пример

```
#include "mpi.h"
#include <stdio.h>
#define NELEMENTS 6

int main(int argc,char *argv[]) {
        int numtasks, rank, source=0, dest, tag=1, i;
        int blocklengths[2], displacements[2];
        float a[16] =
                {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
                 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
        float b[NELEMENTS];
        MPI_Status stat;
        MPI_Datatype indextype;
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
        blocklengths[0] = 4;
        blocklengths[1] = 2;
        displacements[0] = 5;
        displacements[1] = 12;
        MPI_Type_indexed(2, blocklengths, displacements, MPI_FLOAT, &indextype);
```

Пример

```
      if (rank == 0) {
       for (i=1; i<numtasks; i++)
                MPI_Send(a, 1, indextype, i, tag, MPI_COMM_WORLD);
       } else {
            MPI_Recv(b, NELEMENTS, MPI_FLOAT, source, tag, MPI_COMM_WORLD,
&stat);
      printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f %3.1f %3.1f\n",
      rank,b[0],b[1],b[2],b[3],b[4],b[5]);
  }
MPI_Type_free(&indextype);
MPI_Finalize();
}
```

```
rank= 1  b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 2  b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 3  b= 6.0 7.0 8.0 9.0 13.0 14.0
```

MPI_Extent

- Используется при операциях над типами в MPI
- Показывает на сколько байт тип распределен в памяти
- При этом фактический размер типа может быть меньше
- C:

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

- Fortran:

```
CALL MPI_TYPE_EXTENT (DATATYPE, EXTENT,
IERROR)
```

MPI_Struct

- Используется для задания гетерогенных типов
- Наиболее общий тип
- Используется со структурами языка C

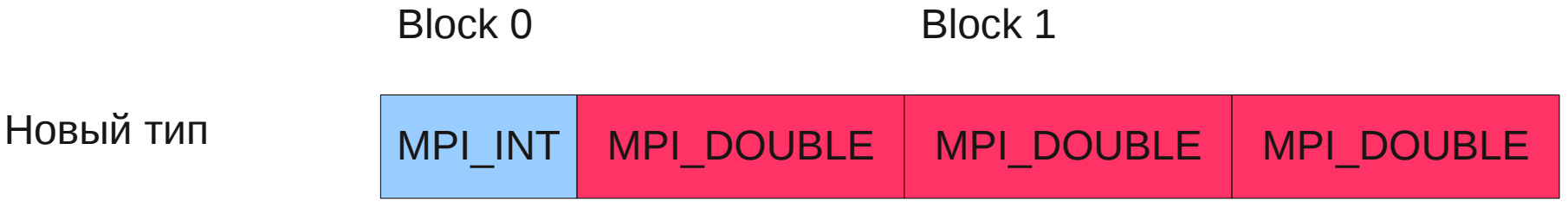
```
int MPI_Type_struct (int count, int
*array_of_blocklengths, MPI_Aint
*array_of_displacements, MPI_Datatype
*array_of_types, MPI_Datatype *newtype)
```

MPI_Struct

MPI_INT

MPI_DOUBLE

```
struct {
        int type;
        double x, y, z;
} point;
```



- count = 2
- array_of_blocklengths = {1,3}
- array_of_types = {MPI_INT, MPI_DOUBLE}
- array_of_displacements = {0, extent(MPI_INT)}

Пример

```
#include <stdio.h>
#include<mpi.h>
int main(int argc, char *argv[]) {
        int rank,i;
        MPI_Status status;
        struct {
                int num;
                float x;
                double data[4];
        } a;
        int blocklengths[3]={1,1,4};
        MPI_Datatype types[3]={MPI_INT, MPI_FLOAT, MPI_DOUBLE};
        MPI_Aint displacements[3];
        MPI_Datatype restype;
        MPI_Aint intex, floatex;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Type_extent(MPI_INT, &intex);MPI_Type_extent(MPI_FLOAT,
&floatex);
        displacements[0] = (MPI_Aint) 0; displacemtnts[1] = intex;
        displacements[2] = intex+floatex;
        MPI_Type_struct(3, blocklengths, displacements, types, &restype);
```

Пример

```
    MPI_Type_commit(&restype);
    If (rank==3){
        a.num=6; a.x=3.14; for(i=0;i 4;++i) a.data[i]=(double) i;
        MPI_Send(&a,1,restype,1,52,MPI_COMM_WORLD);
    } else if(rank==1) {
        MPI_Recv(&a,1,restype,3,52,MPI_COMM_WORLD,&status);
        printf("P:%d my a is %d %f %lf %lf %lf %lf\n",
        rank,a.num,a.x,a.data[0],a.data[1],a.data[2],a.data[3]);
    }
    MPI_Finalize();
}
```

```
P:1 my a is 6 3.140000 0.000000 1.000000 2.000000 3.000002
```