# Введение в MPI

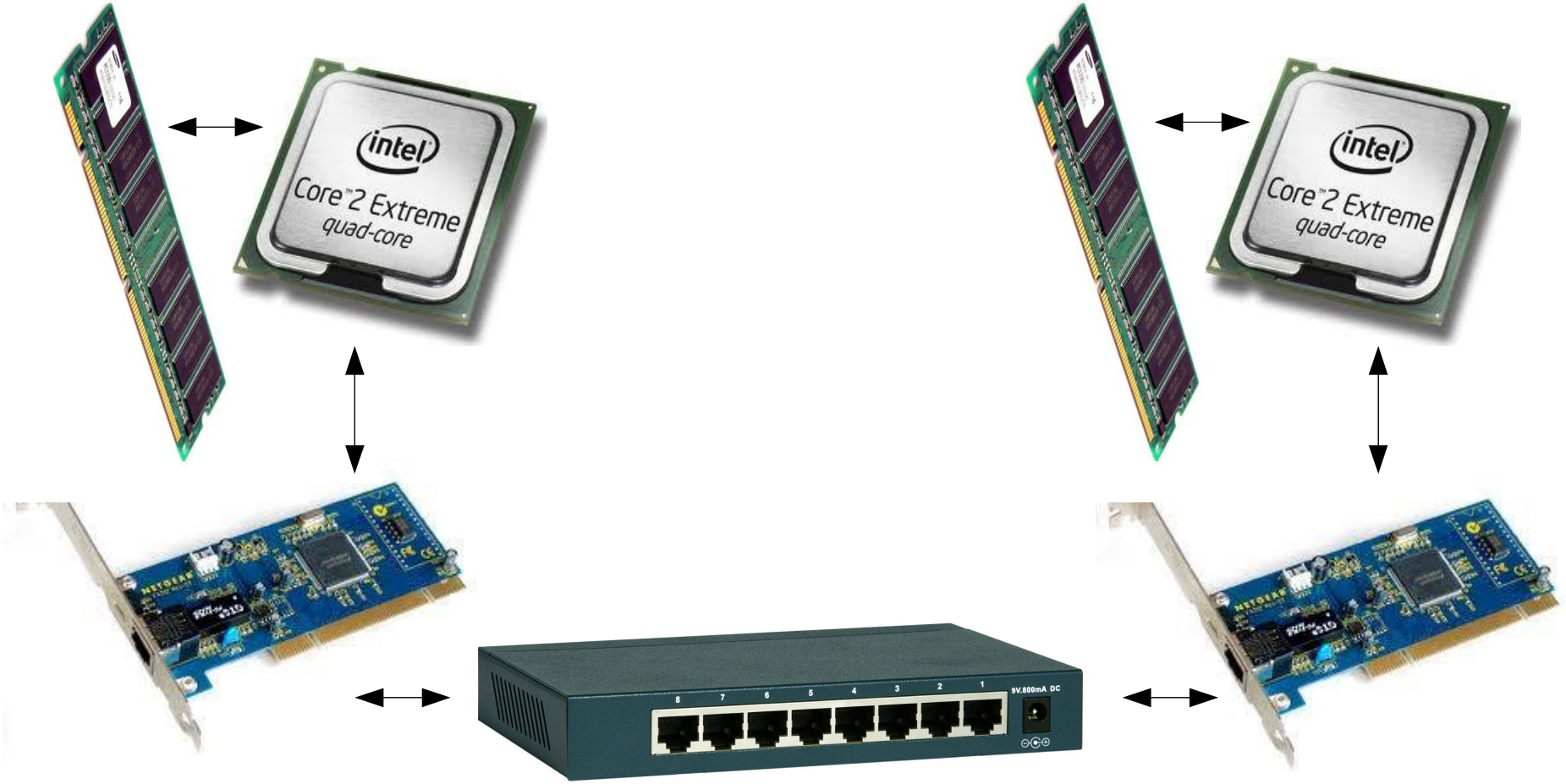
Алексей А. Романенко

*arom@ccfit.nsu.ru*

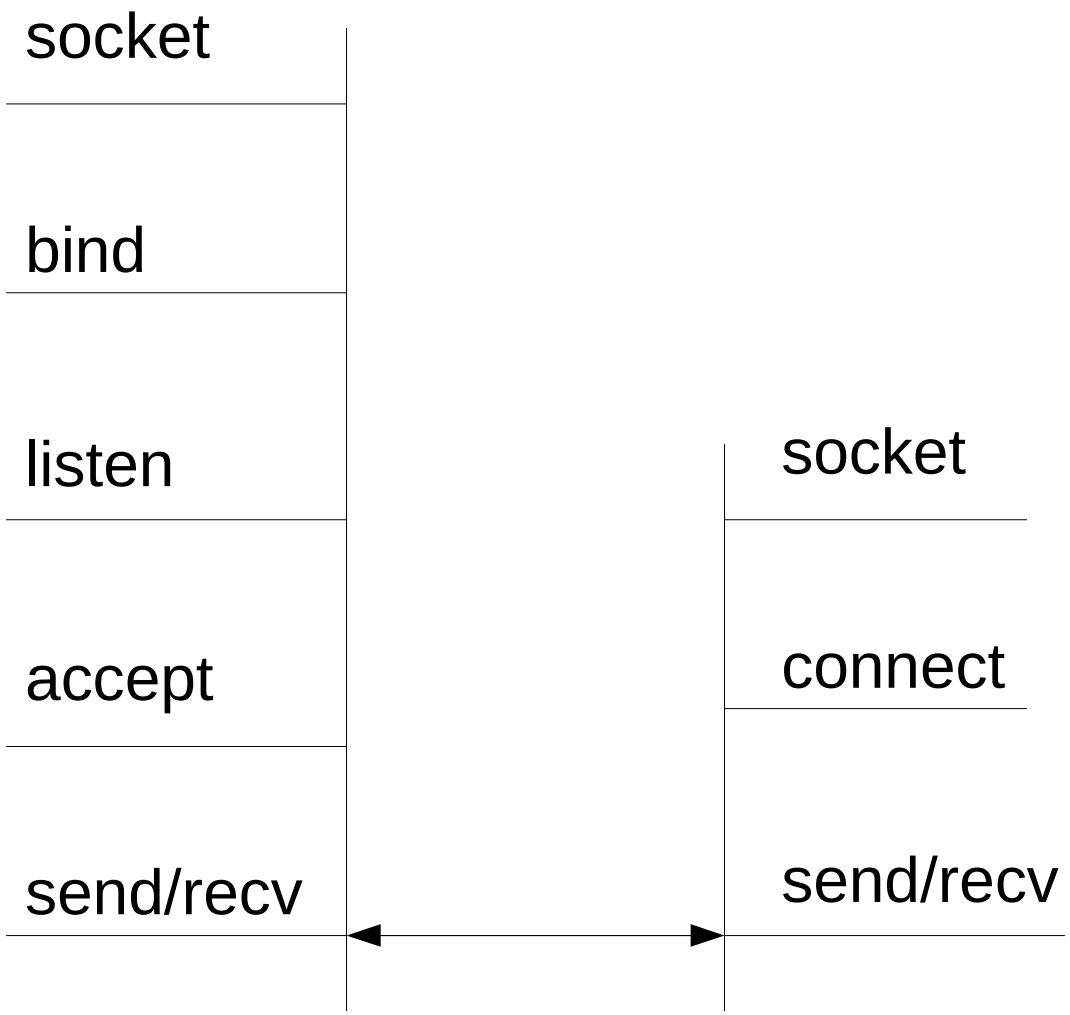
# Содержание раздела

- Системы с распределенной памятью
- MPI
  - История
  - Содержание
  - Типы данных
  - Передача сообщений
  - Топологии

# Системы с распределенной памятью



# Передача данных



# PVM, MPI

- PVM: Parallel Virtual Machine
  - Разработан 80’s
  - Первая высокоуровневая спецификация функций передачи сообщений
  - Процесс PVM демон заботится о передаче сообщений между узлами
- MPI: Message Passing Interface
  - Разработан 90’s
  - Высокоуровневая спецификация функций передачи сообщений
  - Доступно несколько реализаций: mpich, mpi-lam
  - Библиотека функций подключается в программе (демоны отсутствуют)

# Рождение MPI - 1

- MPI-1 Forum
  - Первый стандарт MPI
  - 60 представителей из 40+ организаций США и Европы
  - Два года согласований, обсуждений
  - Разработан и утвержден документ ***Message Passing Interface***
  - MPI 1.0 – июнь, 1994
  - MPI 1.1 – июнь 12, 1995

# MPI-2

- MPI-2 Forum
  - Та же процедура
  - MPI-2: Документ ***Extensions to Message Passing Interface*** (Июль 18, 1997)
  - MPI 1.2 – В основном разъяснения
  - MPI 2.0 – расширение к MPI 1.2

http://www.hlrs.de/mpi/

# MPI - Message Passing Interface

- MPI (MPI-1) является спецификацией на библиотечные функции
- MPI-2:  добавлен параллельный В/В динамическое управление процессами, удаленные операции в памяти и пр.

- MPI Standard:
  *www-unix.mcs.anl.gov/mpi/standard.html*
- MPI Standard 1.1 Index:
  *www.mpi-forum.org/docs/mpi-11-html/node182.html*
- MPI-2 Standard Index:
  *www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-2.0/node306.htm*
- MPI Forum Home Page:
  *www.mpi-forum.org/index.html*

# Цели и границы MPI

- Основная цель MPI
  - Обеспечить механизм передачи сообщений.
  - Обеспечить переносимость кода.
  - Позволить эффективную реализацию.
- Предлагает:
  - Большой объем функциональности.
  - Поддержку гетерогенных архитектур.
- MPI-2:
  - Дополнительную функциональность.
  - Совместимость с MPI-1.

# Сборка и запуск

Заголовочные файлы

C - #include <mpi.h>

Fortran - include ´mpif.h´

сборка:

gcc -o *prog prog.c* -lmpi

mpicc -o *prog prog.c*

*mpiCC -o prog prog.cpp*

g77 -o *prog prog.f* -lmpi

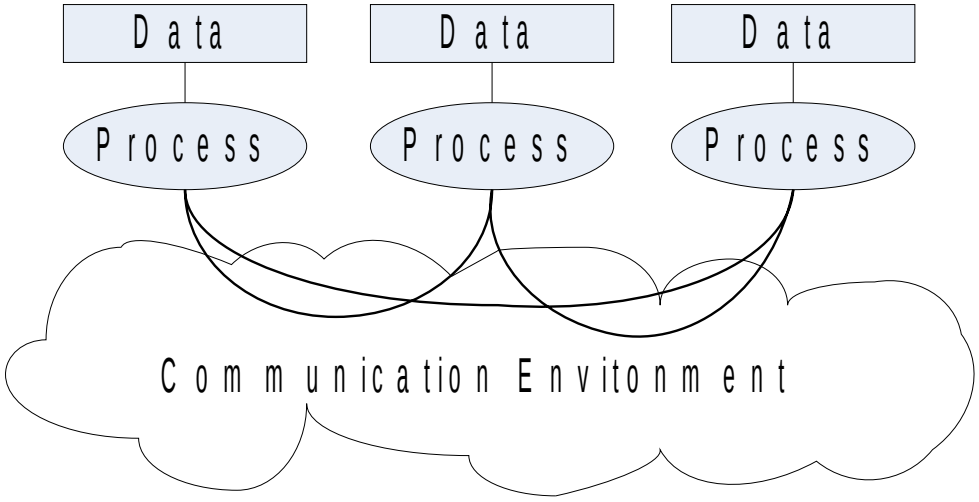
mpif77 -o *prog prog.f*

Запуск

mpirun -np <#> ./*prog* <parameters>

# Парадигма MPI

- Каждый процессор выполняет подпрограмму:
  - написанную на последовательном языке, например, C или Fortran,
  - обычно одну и ту же для каждого процессора (SPMD),
  - переменные в подпрограммах имеют одинаковое имя, но разное расположение различные значения! Т.е. Все переменные приватные.
  - Взаимодействующую через специальные вызова передачи и приема сообщений



# Распределение данных и задач

- Каждый процесс идентифицируется по **rank**
- **Rank -** число  от 0 до **np** – количество параллельных процессов
- Значение **rank** можно узнать
- Количество процессов определяется при запуске программы (mpirun)
- Все распределение работы основываетс на значение **rank**

# Что такое SPMD?

- Single Program, Multiple Data
- Одна и та же  (под-)программа выполнется на разных процессорах
- MPI допускает MPMD, т.е., Multiple Program, но некоторые реализации ограничены только SPMD
- MPMD можно реализовать через SPMD

# Эмуляция MPMD

```
int main(int argc, char *argv[]){
    if (rank < .... /* process should run the ocean model */){
        ocean( /* arguments */ );
    }else{
        weather( /* arguments */ );
    }
}
```

# Сообщения

- Сообщение — порция данных передаваемая между (под)программами
- Необходимые атрибуты:
  - sending process – receiving process (ranks)
  - source location – destination location
  - source data type – destination data type
  - source data size – destination buffer size
- Все посланные сообщения должны быть получены!

# Взаимодействие точка-точка

- Простейший способ взаимодействия.
- Один процесс посылает сообщение другому.
- Типы:
  - Синхронная передача
  - Буферизованая = асинхронная

# Синхронная передача

- Подтверждение того, что сообщение получено.

# Буферизованая передача

- Информация о том, что сообщение послано.
- MPI гарантирует, что сообщение будет доставлено.

# Блокирующие операции

- Операции локального действия
  - Посылка (сообщения)
  - передача (сообщения)
- Блокировка одного процесса возможна пока другой процесс исполняется:
  - Синхронная передача блокируется до тех пор пока сообщение не будет принято;
  - Прием блокируется до тех пор пока не будет послано и получено сообщение.
- Выход из функции блокируется до тех пор пока не завершится операция.

# Неблокирующие операции

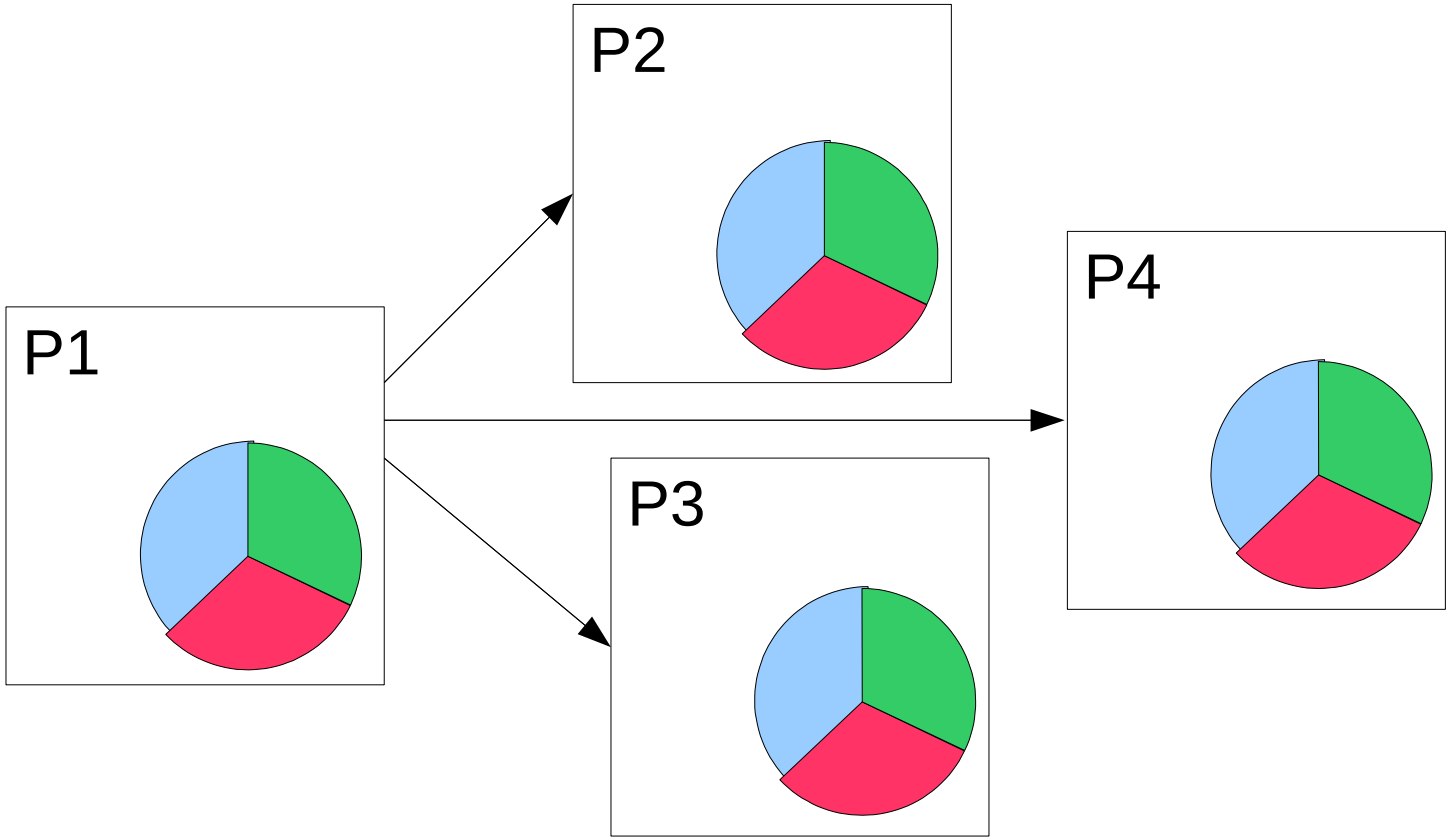
- Немедленный выход из функции.
- Для всех неблокируемых вызовов должна быть выполнена проверка завершенности.
- Некоторые ресурсы могут быть переиспользованы или освобождены после завершения неблокируемых вызовов.
- Неблокируемый вызов с проверкой завершения (wait) - блокируемый вызов.

# Колективное взаимодействие

- Функции высокого уровня.
- Вовлечено несколько процессов.
- Могут применяться оптимизированные алгоритмы для выполнения
- Могут быть построены на базе вызовов точка-точка.

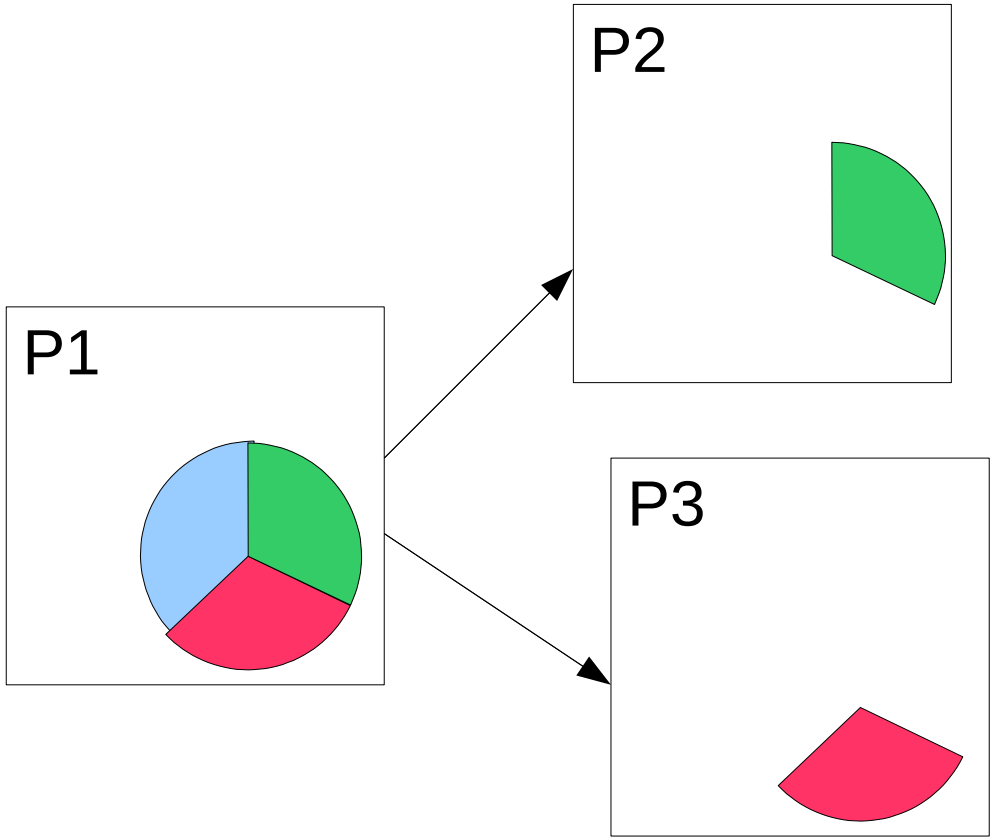
# Broadcast

- Один ко многим.



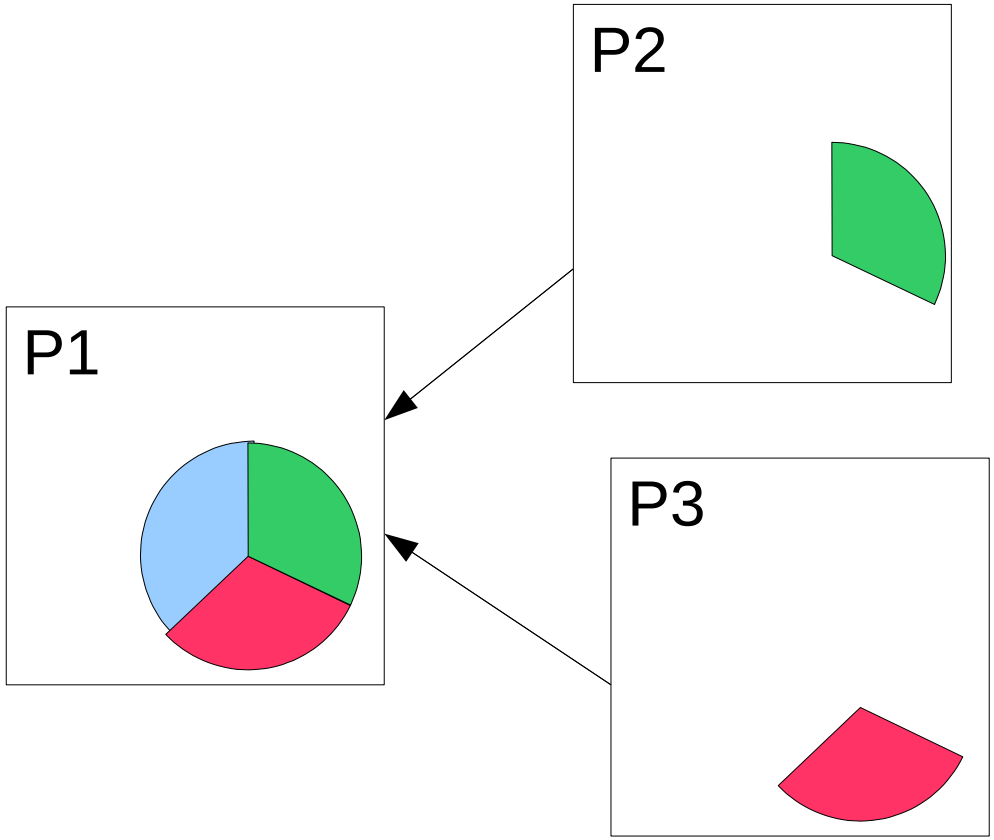
# Scatter

- Распределение данных по процессам



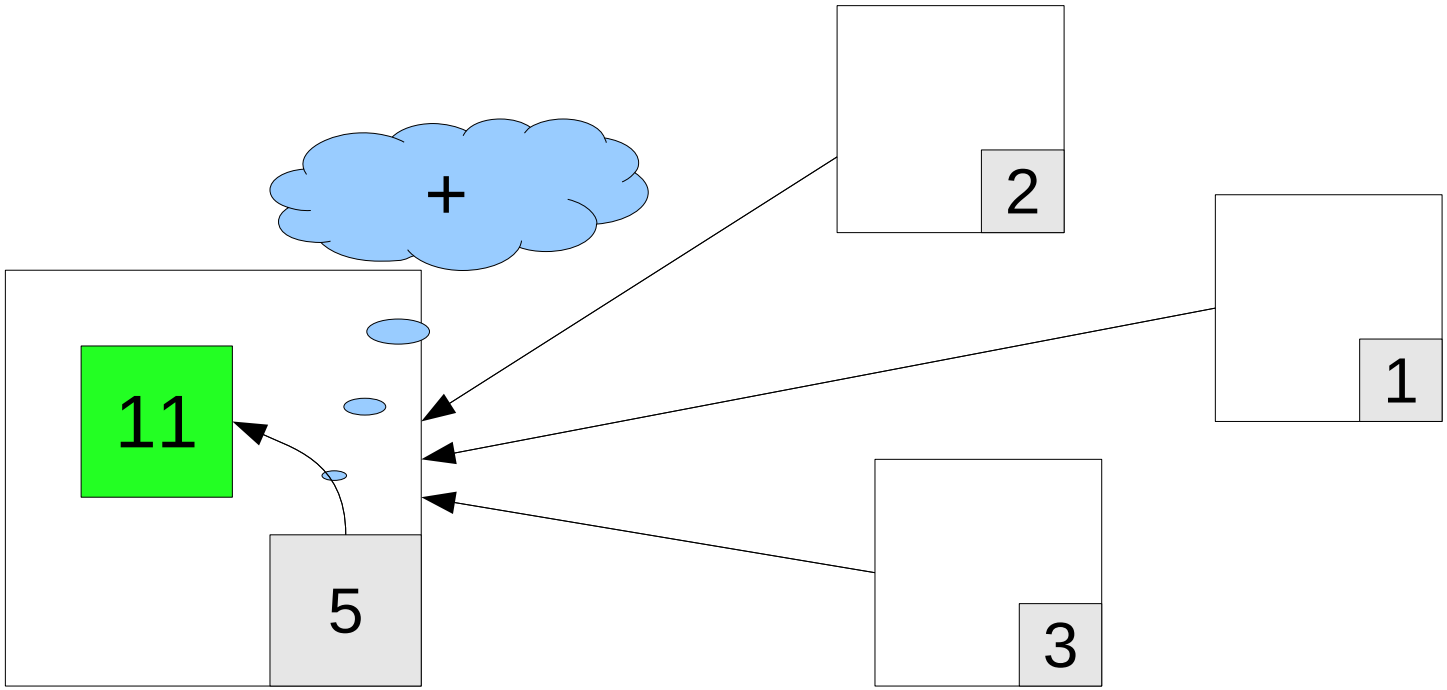
# Gather

- Сбор  данных от процессов



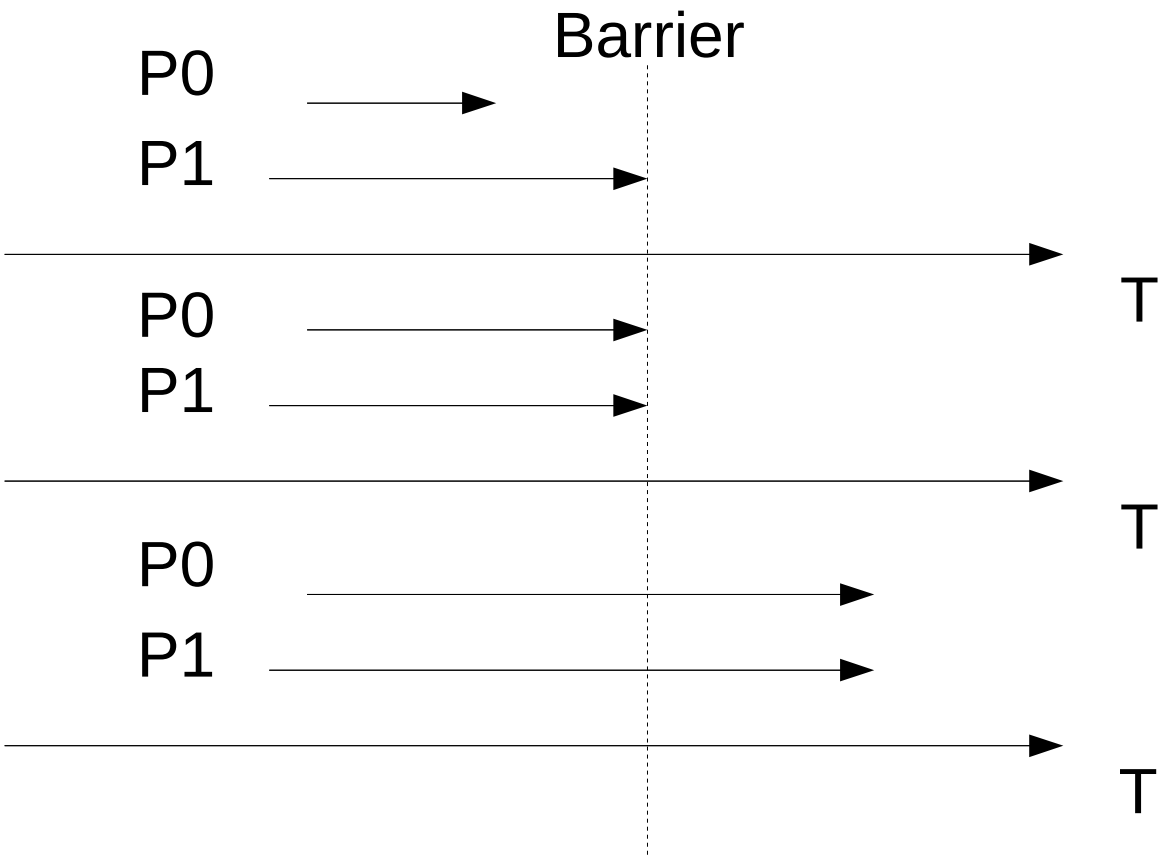
# Операция редукции

- Сбор данных от о всех и передача результата выделенному процессу.



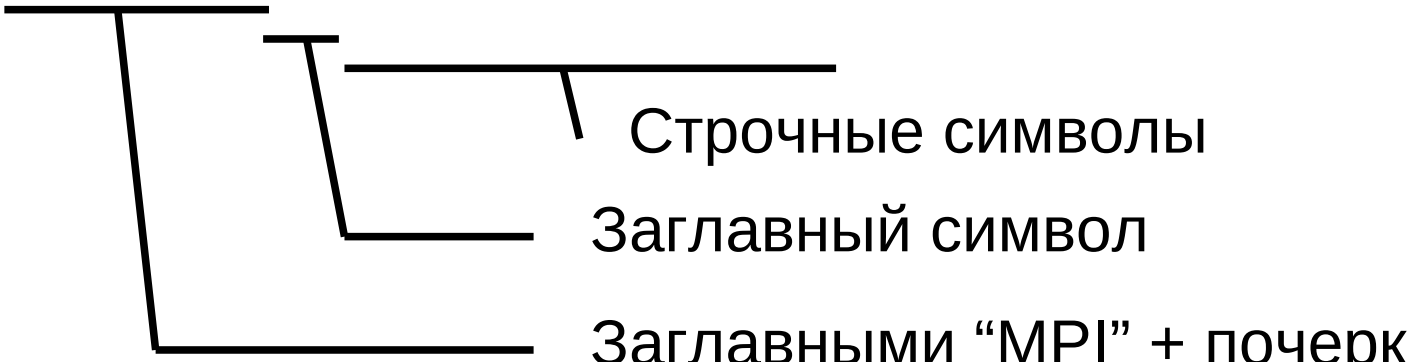
# Barriers

- Синхронизация процессов.



# Правило именования в MPI

- MPI_Xxxx_xxxx



# Инициализация MPI

- C:
  int MPI_Init( int *argc, char ***argv)
- Fortran:
  MPI_INIT( IERROR )
  INTEGER IERROR
- Должен быть первым вызовом MPI!

# Выход из MPI программы

- C:
  int MPI_Finalize()
- Fortran:
  MPI_FINALIZE( IERROR )
- MPI вызовы MPI_Finanize() не допустимы.
- Все процессы должны выполнить этот вызов!

# Коммуникатор MPI_COMM_WORLD

- Все процессы (= подпрограммы) одной MPI программы объединены коммуникатором MPI_COMM_WORLD.
- MPI_COMM_WORLD — константа определенная в mpi.h и mpif.h.
- В коммуникаторе каждый процесс имеет свой rank:
  - Начиная с «0»
  - Заканчивая (size-1)

# Rank

- Идентифицирует процусс в MPI программе.
- Служит для распределения работы.
- C:
  int MPI_Comm_rank( MPI_Comm comm, int *rank)
- Fortran:
  MPI_COMM_RANK( comm, rank, ierror); INTEGER comm, rank, ierror

# Size

- Определяет количество процессов в коммуникаторе
- C:
  int MPI_Comm_size( MPI_Comm comm, int *size)
- Fortran:
  MPI_COMM_SIZE( comm, size, ierror);
  INTEGER comm, size, ierror

# “Hello world” example

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

int main(int argc, char* argv[]){
        int rank, size;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        printf(“I am %d from %d\n”, rank, size);
        MPI_Finalize();
        return 0;
}
```

# “Hello world” example (II)

```
# gcc -o hello hello.c -lmpi
# mpirun -np 4 ./hello
I am 0 from 4
I am 1 from 4
I am 3 from 4
I am 2 from 4
# mpirun -np 4 ./hello
I am 2 from 4
I am 1 from 4
I am 3 from 4
I am 0 from 4
# mpirun -np 4 ./hello
I am 0 from 4
I am 2 from 4
I am 1 from 4
I am 3 from 4
```

# Типы данных

- Сообщение содержит некоторое количество элементов определенного типа данных.
- Типы даннх MPI:
  - Базовые (встроенные).
  - Пользовательские.
- Пользовательские типы данных могут строиться из базовых.
- Типы данных Си отличны от типов данных Fortran.

# Базовые типы данных (C)

| MPI datatype | C datatype |
|---|---|
| MPI_PACKED | |
| MPI_BYTE | |
| MPI_LONG_DOUBLE | long double |
| MPI_DOUBLE | double |
| MPI_FLOAT | float |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_LONG | signed long int |
| MPI_INT | signed int |
| MPI_SHORT | signed short int |
| MPI_CHAR | signed char |

# Базовые типы данных (Fortran)

| MPI Datatype | Fortran datatype |
|---|---|
| MPI_PACKED | |
| MPI_BYTE | |
| MPI_CHARACTER | CHARACTER(1) |
| MPI_LOGICAL | LOGICAL |
| MPI_COMPLEX | COMPLEX |
| MPI_DOUBLE_PRECISION | DOUBLE PRECISION |
| MPI_REAL | REAL |
| MPI_INTEGER | INTEGER |

# Посылка сообщения

- C: int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- Fortran: MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
- <type> BUF(*), INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
- **buf** — указатель на начало сообщения длинной в count элементов, каждый из которых имеет тип **datatype**.
- **dest** - ранг получателя в коммуникаторе **comm**.
- **tag** — положительное целое число идентифицирующее передачу.
- **tag** может использоваться чтобы различать сообщения.

# Получение сообщения

- C: int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
- Fortran: MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
- **buf**/**count**/**datatype** описание буфера для приема сообщения.
- Получать сообщение с **rank** = **source** в коммуникаторе **comm**.
- Информация о сообщение возвращается в **status**.
- Получаются только сообщения с совпадающим **tag**.

# Требования к коммуникациям точка-точка

Для успешного завершения операции:

- При посылке указан правильный rank получателя сообщения.
- При приеме указан правильный rank отправителя сообщения.
- Указан один и тот же коммуникатор.
- tag, типы данных должны совпадать.
- Буфер в приемнике >= длине сообщения.

# Send-Recv example

```
#include <mpi.h>

int main(int argc, char* argv[]){
    int rank, buf[256];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank == 0){
        // init buffer
        MPI_Send(buf, 256, MPI_INT, 1, 10, MPI_COMM_WORLD);
    }
    if(rank == 1){
        MPI_Recv(buf, 256, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
        // process buffer
    }
    MPI_Finalize();
    return 0;
}
```

# Wildcard

- Получение по wildcard.
- Получить сообщение от любого источника — **source** = MPI_ANY_SOURCE
- Получить сообщение с любым tag — **tag** = MPI_ANY_TAG
- Реальные значения source и tag возвращаются в структуре **status**.

# Информация о сообщение

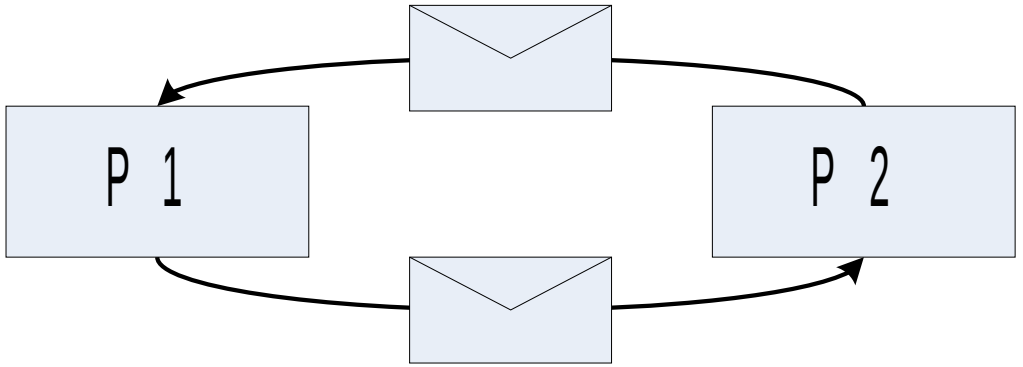
- Возвращается в переменную status.
- C: status.MPI_SOURCE
  status.MPI_TAG
  count через MPI_Get_count()
- Fortran: status(MPI_SOURCE)
  status(MPI_TAG)
  count через MPI_GET_COUNT()

# Значение count при получении

- C:
  int MPI_Get_count(MPI_Status *status,
  MPI_Datatype datatype, int *count)
- Fortran:
  MPI_GET_COUNT(STATUS, DATATYPE,
  COUNT, IERROR)

# Неблокирующие операции

# Deadlock




- Код в каждом MPI  процессе:
  MPI_Send(…, right_rank, …)
  MPI_Recv( …, left_rank, …)

# Неблокирующие операции

- Выполнение в три этапа:
- Инициализация
  - немедленный выход
  - функции начинаются с MPI_I…
- Выполняем какую-то работу
- Ожидание завершения

# Non-blocking Send

- C:
  MPI_Isend(buf, count, datatype, dest, tag, comm, &request_handle);
  MPI_Wait(&request_handle, &status);
- Fortran:
  CALL MPI_ISEND( buf, count, datatype, dest, tag, comm, request_handle, ierror);
  CALL MPI_WAIT(request_handle, status, ierror)
- buf не должен использоваться между  Issend и Wait
- “Isend + Wait сразу после Issend” эквивалентно (Ssend)
- status в Isend не используется, только в Wait

# Non-blocking Receive

- C:
  MPI_Irecv ( buf, count, datatype, source, tag, comm, &request_handle);
  MPI_Wait(&request_handle, &status);
- Fortran:
  CALL MPI_IRECV (buf, count, datatype, source, tag, comm, request_handle, ierror);
  CALL MPI_WAIT(request_handle, status, ierror)
- Buf не должен использоваться между Irecv и Wait

# Завершение

- C:
  MPI_Wait( &request_handle, &status);
  MPI_Test( &request_handle, &flag, &status);
- Fortran:
  CALL MPI_WAIT( request_handle, status, ierror)
  CALL MPI_TEST( request_handle, flag, status, ierror)
- Должно быть или
  - WAIT или
  - Цикл с  TEST пока  flag не станет равным  1 или “TRUE”.

# Завершение (2)

Обработка нескольких запросов:

- Wait или test завершения нескольких операций
  - MPI_Waitany / MPI_Testany
- Wait или test завершения всех операций
  - MPI_Waitall / MPI_Testall
- Wait или test завершения как можно большего количества операций
  - MPI_Waitsome / MPI_Testsome

# Blocking and Non-Blocking

- Send и receive могут быть как блокируемые, так и неблокируемые
- Блокирующий send может использоваться с неблокирующим receive, и наоборот.

# Типы данных MPI

- Описывают расположение данных в памяти
  - При отправке
  - При приеме
- Базовые типы
- Производные типы
  - векторы
  - структуры
  - прочие

# Производные типы данных

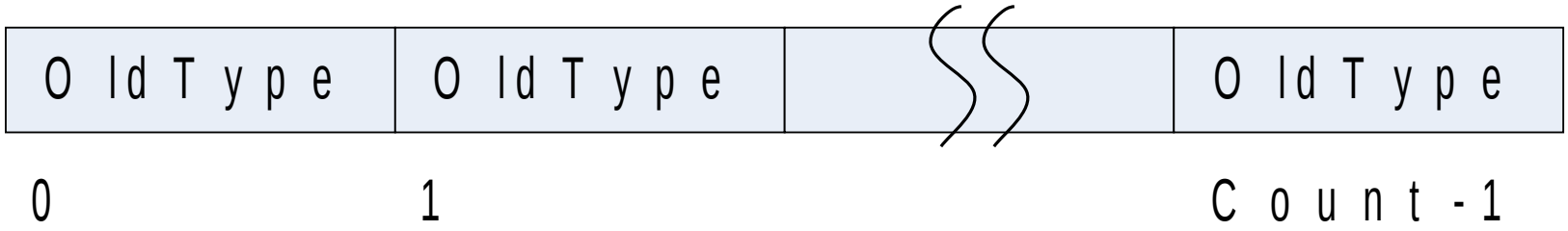
- Указатель на список базовых типов и их смещения

| | |
|---|---|
| basic datatype 0 | displacement of datatype 0 |
| basic datatype 1 | displacement of datatype 1 |
| ... | |
| basic datatype n-1 | displacement of datatype n-1 |

# Производный тип данных (пример)

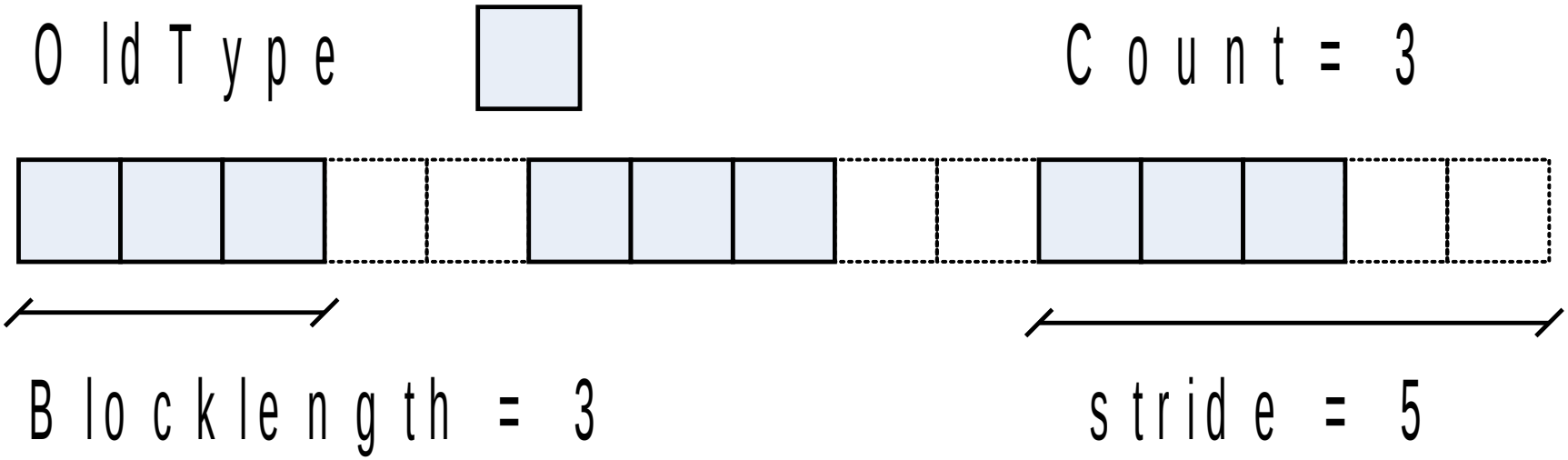
- struct new_type{
  char a;
  int b,c;
  float e;
  }
- Базовые типы {MPI_CHAR, MPI_INT, MPI_INT, MPI_FLOAT}
- Базовое смещение {0,4,8,12}

# Contiguous Data




- Простейший производный тип данных
- C:
  int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
- Fortran:
  MPI_TYPE_CONTIGUOUS( COUNT, OLDTYPE, NEWTYPE, IERROR)

# Vector Datatype




- C:
  int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
- Fortran:
  MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)

# Struct Datatype

- C:
  int MPI_Type_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)
- Fortran: MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)

# Structure datatype example

- struct { int a; char b; } foo;
- sizeof(foo) > sizeof(int) + sizeof(char)

- blen[0] = 1; indices[0] = 0; oldtypes[0] = MPI_INT;
  blen[1] = 1; indices[1] = &foo.b - &foo; oldtypes[1] = MPI_CHAR;
  blen[2] = 1; indices[2] = sizeof(foo); oldtypes[2] = MPI_UB;
  MPI_Type_struct(3, blen, indices, oldtypes, &newtype);

# Вычисление смещения

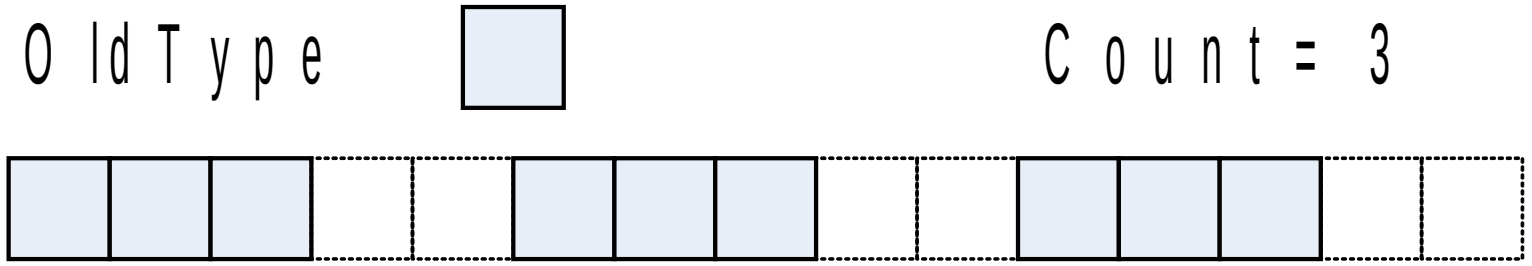
- array_of_displacements[i] := address(block_i) – address(block_0)
- MPI-1
  - C:
    int MPI_Address(void* location, MPI_Aint *address)
  - Fortran:
    MPI_ADDRESS(LOCATION, ADDRESS, IERROR)

# Создание типа

- Перед тем, как новый тип данных можно будет использовать нужно завершить его создание MPI_TYPE_COMMIT.
- Достаточно выполнить только один раз.
- C:
  int MPI_Type_commit(MPI_Datatype *datatype);
- Fortran:
  MPI_TYPE_COMMIT(DATATYPE, IERROR)

# Size и Extent  типа данных (1)

- Size := количество байт, которое будет передано
- Extent := количество байт в памяти
- Для мазовых типов Size = Extent
- Производные типы:




size = 9*sizeof(OldType); extend = 15*sizeof(OldType)

# Size и Extent типа данных (2)

- C:
  int MPI_Type_size(MPI_Datatype datatype, int *size)
  int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
- Fortran:
  MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
  MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)

# Виртуальные топологии

- Удобное наименование процессов
- Адаптация расположения процессов к архитектуре кластера
- Упрощает код
- Передача данных может быть оптимизирована

# Использование вируальных топологий

- Создание новой топологии и соответствующего коммуникатора.
- MPI функции для создания топологий и наименования процессов в них.

# Типы топологий

- Декартова
  - Каждый процесс связан со своим соседом в виртаальной сети,
  - Возможно связывание границ,
  - Идентификация процессов координатами в декартовой системе координат,
  - Коммуникации между всеми процессами возможны
- Граф

# Создание декартовой виртуальной топологии

- C:
  int MPI_Cart_create(MPI_Comm comm_old,
  int ndims, int *dims, int *periods, int reorder,
  MPI_Comm *comm_cart)
- Fortran:
  MPI_CART_CREATE( COMM_OLD, NDIMS,
  DIMS, PERIODS, REORDER,
  COMM_CART, IERROR)

Изменение порядка может привести к оптимизации передач.

# Получение координат

- C:
  int MPI_Cart_coords(MPI_Comm comm_cart, int rank,
  int maxdims, int *coords);
  int MPI_Cart_rank(MPI_Comm comm_cart,
  int *coords, int *rank)

- Fortran:
  MPI_CART_COORDS(COMM_CART, RANK,
  MAXDIMS, COORDS, IERROR);
  MPI_CART_RANK(COMM_CART, COORDS,
  RANK, IERROR)

- Получение своих координат
  MPI_Comm_rank(comm_cart, my_rank);
  MPI_Cart_coords(comm_cart, my_rank,
  maxdims, my_coords)

# Соседи в декартовой топологии

- Вычисление раков соседей
- C:

```
int MPI_Cart_shift(MPI_Comm comm_cart,
        int direction, int disp, int *rank_source,
        int *rank_dest)
```

- Fortran:

```
MPI_CART_SHIFT( COMM_CART, DIRECTION,
        DISP, RANK_SRC, RANK_DEST, IERROR)
```

- Если соседей нет, то MPI_PROC_NULL.
- MPI_PROC_NULL может использоваться как rank
  → передачи не будет!

# Создание топологии типа граф

- C

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

- Fortran

```
MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX,
        EDGES, REORDER, COMM_GRAPH, IERROR)
```

# Коллективные операции

- Взаимодействует группа процессов.
- Вовлечены все процессы в коммуникаторе
- Примеры:
  - Синхронизация (барьер).
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.

# Характеристики коллективных операций

- Коллективная над коммуникатором
- ВСЕ процессы в рамках коммуникатора должны вызвать коллективную функцию
- Коллективные операции - блокирующие.
- Нет tag-а.
- Буфер на отправку данных должен быть такого же размера как и буфер на прием данных.

# Barrier Synchronization

- C: int MPI_Barrier(MPI_Comm comm)
- Fortran: MPI_BARRIER(COMM, IERROR)
- MPI_Barrier обычно не используется, но:
  - Используется для отладки:
    - Не забывайте удалять после отладки.
  - Используется для профилирования
    - Load imbalance of computation [ MPI_Wtime(); MPI_Barrier(); .... MPI_Wtime() ]
    - communication epochs [ MPI_Wtime(); MPI_Allreduce(); …; MPI_Wtime() ]
  - Используется для синхронизации внешних операций (например, I/O):
    - Посылка сообщений может быть более эффективна

# Broadcast

- C:
  int MPI_Bcast(void *buf, int count,
  MPI_Datatype datatype,int root, MPI_Comm comm)
- Fortran:
  MPI_Bcast(BUF, COUNT, DATATYPE, ROOT,
  COMM, IERROR)

# Scatter

- C:

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

- Fortran:

```
MPI_SCATTER(SENDBUF, SENDCOUNT,
        SENDTYPE, RECVBUF, RECVCOUNT,
        RECVTYPE, ROOT, COMM, IERROR)
```

# Gather

- C:

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

- Fortran:

```
    MPI_GATHER(SENDBUF, SENDCOUNT,
        SENDTYPE, RECVBUF, RECVCOUNT,
        RECVTYPE, ROOT, COMM, IERROR)
```

# Глобальные операции редукции

- To perform a global reduce operation across all members of a group.
  $d_0$ o $d_1$ o $d_2$ o $d_3$ o … o $d_{s-2}$ o $d_{s-1}$
  - $d_i$ = data in process rank i
    - single variable, or vector
  - o = associative operation
- Example:
  - global sum or product
  - global maximum or minimum
  - global user-defined operation
- floating point rounding may depend on usage of associative law.

# Предопределенные операторы редукции

| Операторы | Функции |
|---|---|
| MPI_BAND | Bitwise AND |
| MPI_BOR | Bitwise OR |
| MPI_BXOR | Bitwise exclusive OR |
| MPI_MAX | Maximum |
| MPI_MAXLOC | Maximum and location of the maximum |
| MPI_MIN | Minimum |
| MPI_MINLOC | Minimum and location of the minimum |
| MPI_LAND | Logical AND |
| MPI_LOR | Logical OR |
| MPI_LXOR | Logical exclusive OR |
| MPI_PROD | Product |
| MPI_SUM | Sum |

# Глобална редукция

- C:
  MPI_Reduce(void *sendbuf, void *recvbuf, int count,
  MPI_Datatype datatype, MPI_Op op,
  int root, MPI_Comm comm)
- Fortran:
  MPI_REDUCE(SENDBUF, RECVBUF, COUNT,
  DATATYPE, OP, ROOT, COMM, IERROR)

# Заключение

Модель процессов MPI

- Передача сообщений
  - Блокируемая → several modes (standard, buffered, synchronous, ready)
  - неблокируемая
    - Позволяет параллельную передачу сообщений
    - Позволяет исключить мертвые блокировки
  - Собственные типы данных
    - Передача различных данных одной посылкой
- Виртуальные топологии → схема нумерации процессов
- Коллективные операции → доп. Возможность оптимизации

# Пример 1. Контрольная сумма

```
int res, gres;
char * buf, *rbuf;
MPI_Init(&argc, &argv);
// init bufs
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

while(/* have data */){
    if(rank == 0) read_data(buf, 100*size);
    MPI_Scatter(buf, 100,  MPI_CHAR, rbuf, 100,
                                MPI_CHAR, 0, MPI_COMM_WORLD);
    res = process_data(rbuf, 100);
    MPI_Reduce(&res, &gres, 1, MPI_INT, MPI_XOR, 0,
                                MPI_COMM_WORLD);
    save_res(gres);
}
```

# Пример 2. Master-slave

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

for(i=0; i<size; i++){
    read_data(buf, d_size);
    MPI_Send(buf, d_size, MPI_INT, i, 0, MPI_COMM_WORLD);
}

while(/* have data */){
    MPI_Recv(buf, d_size, MPI_INT, MPI_ANY_SOURCE, 0,
            MPI_COMM_WORLD, &status);
    save_res(buf, d_size);
    read_data(buf, d_size);
    i = status.MPI_SOURCE;
    MPI_Send(buf, d_size, MPI_INT, i, 0, MPI_COMM_WORLD);
}
```

# Пример 2. Master-slave (прод.)

```
for(i=0; i<size; i++){
    MPI_Recv(buf, d_size, MPI_INT, MPI_ANY_SOURCE, 0,
            MPI_COMM_WORLD, &status);
    save_res(buf, d_size);
    // send completion
    MPI_Send(buf, d_size, MPI_INT, i, 1, MPI_COMM_WORLD);
}

MPI_Finalize();
```