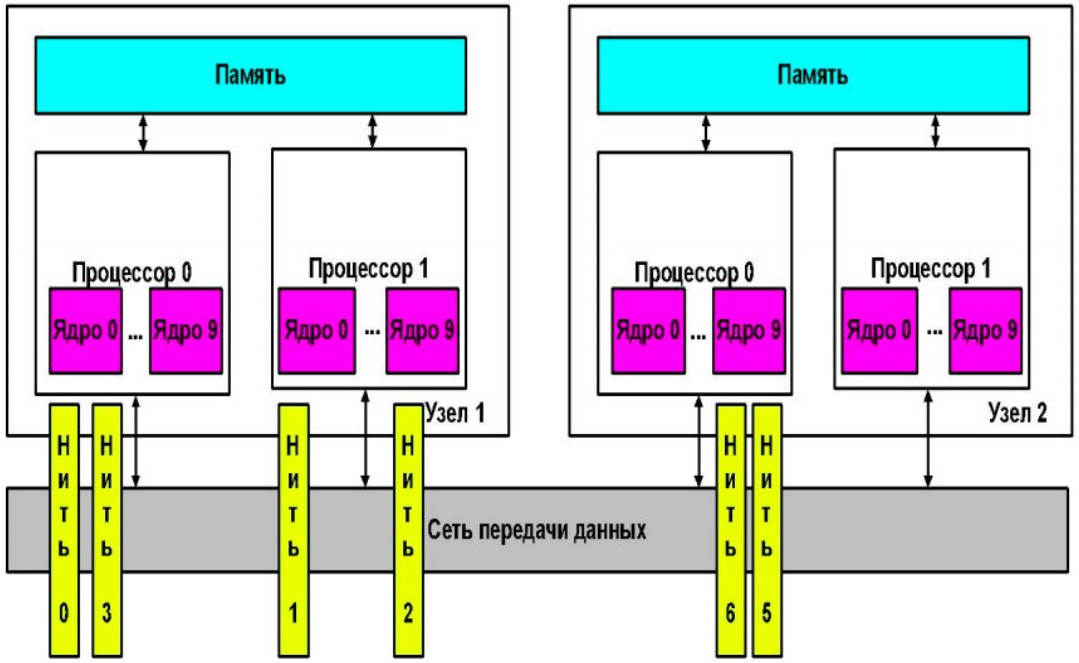
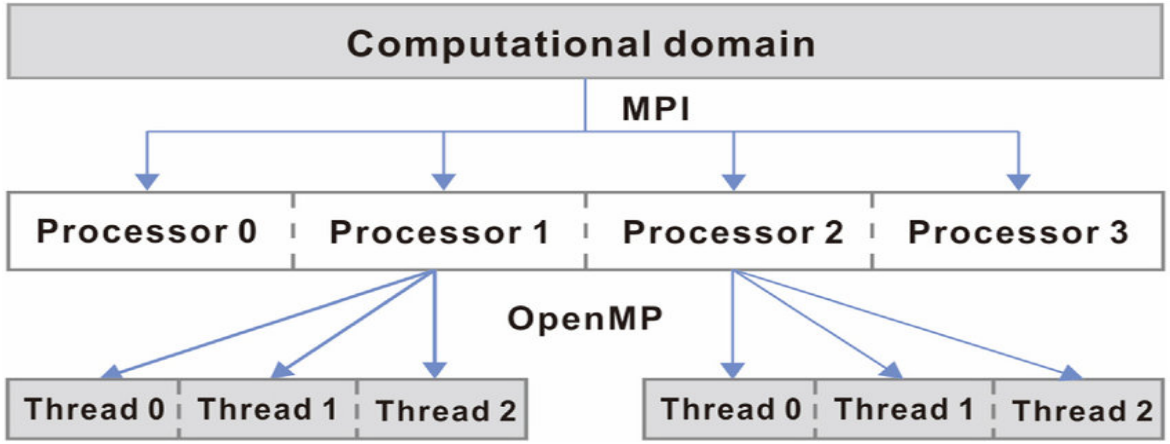
MPI

MPI

- **Message Passing Interface** (MPI, интерфейс передачи сообщений) — программный интерфейс для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу.
- Процессы обладают распределенной памятью.
- Модель SPMP - одна программа для множества процессов.

Структура работы MPI





Установка и запуск

- MPIHC - одна из реализаций интерфейса MPI.
- Компилятор MPIC++
- Запуск: mpirun -np 4 ./program

Инициализация MPI

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

MPI_Send

- MPI_Send(
- void* data, данные
- int count, размер данных
- MPI_Datatype datatype, тип посылаемых данных
- int destination, ранг процесса получателя
- int tag, тэг сообщения
- MPI_Comm communicator, Коммуникатор, в котором отправляется сообщение (MPI_COMM_WORLD))

MPI_Recv

- MPI_Recv(
- void* data, данные
- int count, размер данных
- MPI_Datatype datatype, тип посылаемых данных
- int source, ранг процессса отправителя
- int tag, тэг
- MPI_Comm communicator, коммуникатор
- MPI_Status* status, статус операции)

MPI_Status

- typedef struct _MPI_Status {
- int count;
- int cancelled;
- int MPI_SOURCE; Откуда пришло сообщение
- int MPI_TAG; Тэг
- int MPI_ERROR; Код ошибки }

/* The following two fields (count, cancelled) are internal to the Open MPI implementation and should not be accessed by MPI applications. They are subject to change at any time. These are not the droids you're looking for. */

MPI_Probe

- MPI_Probe(
- int source,
- int tag,
- MPI_Comm communicator,
- MPI_Status* status)

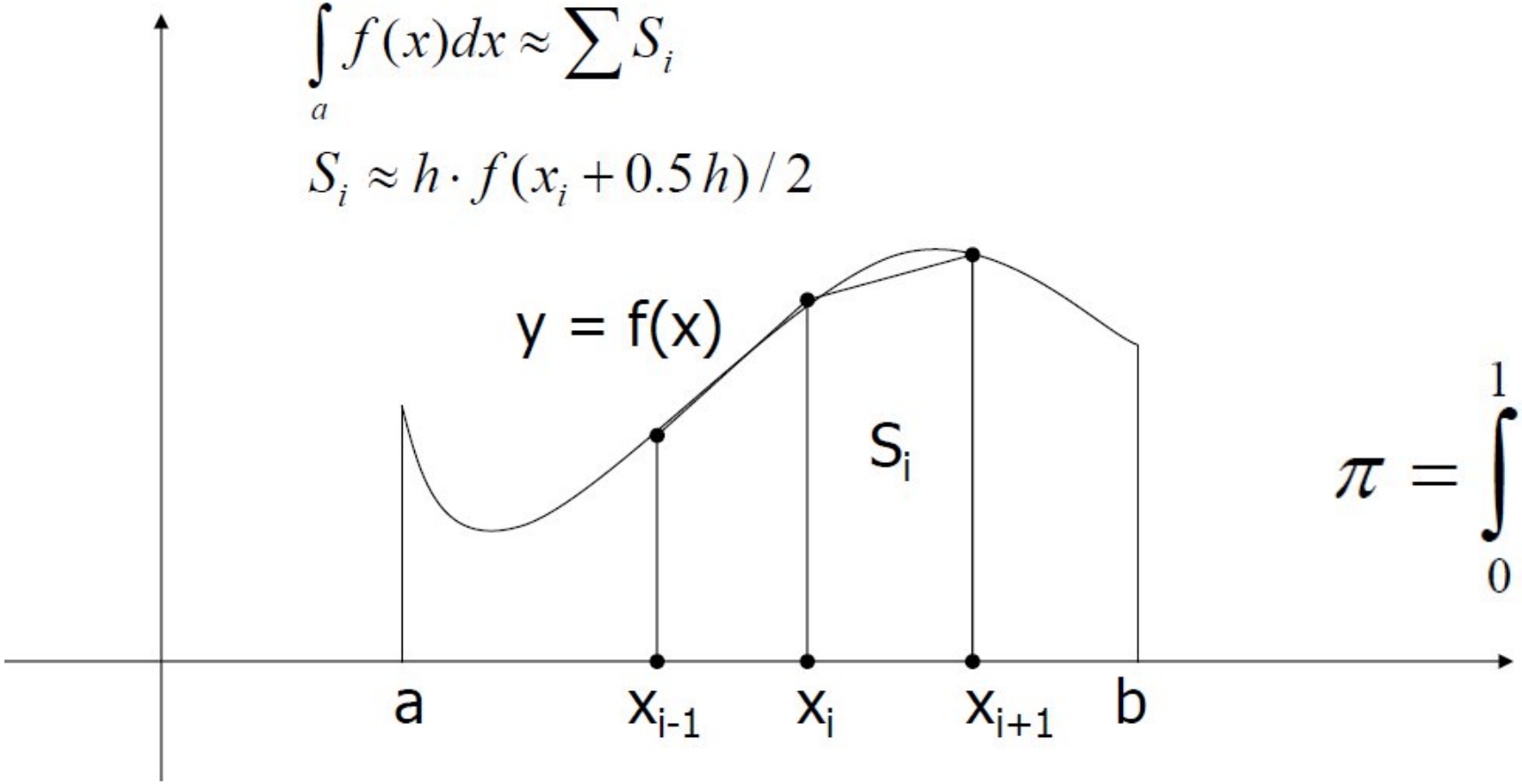
MPI_Get_count

- MPI_Get_count(
- const MPI_Status *status, Возвращает статус операции
- MPI_Datatype datatype, Тип данных ожидаемых эллементов
- int *count)

Задача на интеграл

$$\int_a^b f(x)dx \approx \sum S_i$$

$$S_i \approx h \cdot f(x_i + 0.5h)/2$$



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

Джокеры

- MPI_ANY_SOURCE
- MPI_ANY_TAG

MPI_Sendrecv

- Можно одновременно отправлять и получать сообщения. MPI гарантирует отсутсвие deadlock.
- 12 аргументов, первые 5 как в MPI_Send, остальные 7 как в MPI_Recv
- MPI_Sendrecv_replace - использует общий буфер для отправки и получения. Принимаемые данные должны быть заведомо НЕ ДЛИННЕЕ отправляемых.

Упаковка/Распаковка структур

```
//Передача
//========
int bufPos = 0;
char tempBuf[ sizeof(s) ];
MPI_Pack(&s.i, incount: 1, MPI_INT,   tempBuf, sizeof(tempBuf), &bufPos, MPI_COMM_WORLD );
MPI_Pack( s.f, incount: 4, MPI_FLOAT, tempBuf, sizeof(tempBuf), &bufPos, MPI_COMM_WORLD );
MPI_Pack( s.c, incount: 8, MPI_CHAR,  tempBuf, sizeof(tempBuf), &bufPos, MPI_COMM_WORLD );
MPI_Send( tempBuf, bufPos, MPI_BYTE, targetRank, msgTag, MPI_COMM_WORLD );

//Прием
//=====
int bufPos = 0;
char tempBuf[ sizeof(s) ];
MPI_Recv( tempBuf, sizeof(tempBuf), MPI_BYTE, sourceRank, msgTag,
          MPI_COMM_WORLD, &status );
MPI_Unpack( tempBuf, sizeof(tempBuf), &bufPos,&s.i,  outcount: 1, MPI_INT,  MPI_COMM_WORLD);
MPI_Unpack( tempBuf, sizeof(tempBuf), &bufPos, s.f,  outcount: 4, MPI_FLOAT,MPI_COMM_WORLD);
MPI_Unpack( tempBuf, sizeof(tempBuf), &bufPos, s.c,  outcount: 8, MPI_CHAR, MPI_COMM_WORLD);
```

```
#define msgTag 10
struct {
    int   i;
    float f[4];
    char  c[8];
} s;
```

```
//Чит
Передача:  MPI_Send( &s, sizeof(s), MPI_BYTE, ... );
Прием:     MPI_Recv( &s, sizeof(s), MPI_BYTE, ... );
```

Введение нового типа данных

- MPI_Type_struct - формирование нового типа-структуры
- MPI_Type_commit - сохранение типа в MPI
- MPI_Type_free - очистка данных о новом типе

```
#include <cstddef>  /* подключаем макрос 'offsetof()' */

    typedef struct {
        int    i;
        double d[3];
        long   l[8];
        char   c;
    } AnyStruct;

    AnyStruct st;

    MPI_Datatype anyStructType;

    int          len[5] = { 1, 3, 8, 1, 1 };
    MPI_Aint     pos[5] = { offsetof(AnyStruct,i), offsetof(AnyStruct,d),
                            offsetof(AnyStruct,l), offsetof(AnyStruct,c),
                            sizeof(AnyStruct) };
    MPI_Datatype typ[5] = { MPI_INT,MPI_DOUBLE,MPI_LONG,MPI_CHAR,MPI_UB };

    MPI_Type_struct( count: 5, len, pos, typ, &anyStructType );
    MPI_Type_commit( &anyStructType );
    /* подготовка закончена */

    MPI_Send( st, 1, anyStructType, ... );
```

MPI_Isend

- int MPI_Isend(
- const void *buf,
- int count,
- MPI_Datatype datatype,
- int dest,
- int tag,
- MPI_Comm comm,
- MPI_Request * request - Возвращаемый параметр для отслеживания завершения операции)

MPI_Irecv

- int MPI_Irecv(
- void *buf,
- int count,
- MPI_Datatype datatype,
- int source,
- int tag,
- MPI_Comm comm,
- MPI_Request * request)

MPI_Wait

- int MPI_Wait(MPI_Request *request, MPI_Status *status) - ждет завершения одного реквеста
- int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[]) - ждет завершения всех реквестов в массиве
- int MPI_Waitany(int count, MPI_Request array_of_requests[], int *indx, MPI_Status * status) - ждет завершения одного из реквестов в массиве

MPI_Test

- int MPI_Test(
- MPI_Request *request,
- int *flag, flag == true если реквест завершен
- MPI_Status *status)

- Есть такие же вариации как и для MPI_Wait

Режимы отправки сообщений

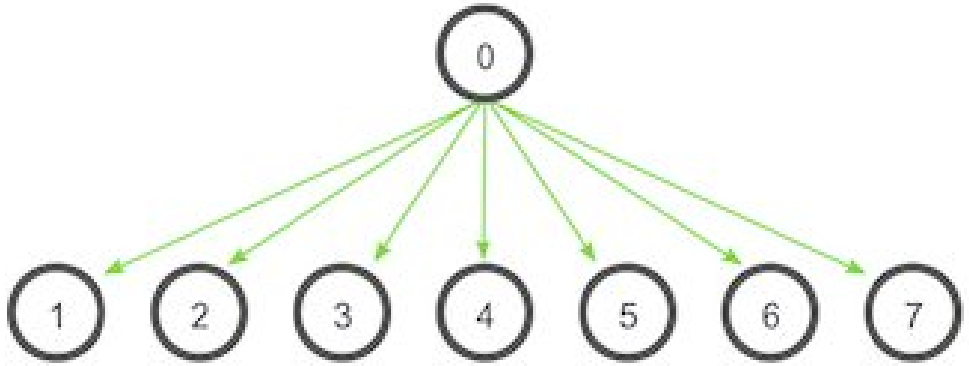
- Bsend (buffered) - Локальная операция, если операция приема не начата - отправляет сообщение в буффер. Контроль буффера отдается пользователю. Осуществяется при помощи MPI_Buffer_attach и MPI_Buffer_detach.
- Ssend (synchronous) - Отправка ждет пока другим процессом будет начат процесс получения сообщения.
- Rsend (ready) - Отправка требует начала операции приема другим процессом.

MPI_Barrier

- int MPI_Barrier(MPI_Comm comm)
- Все процессы в комуникаторе синхронизируются на этой команде

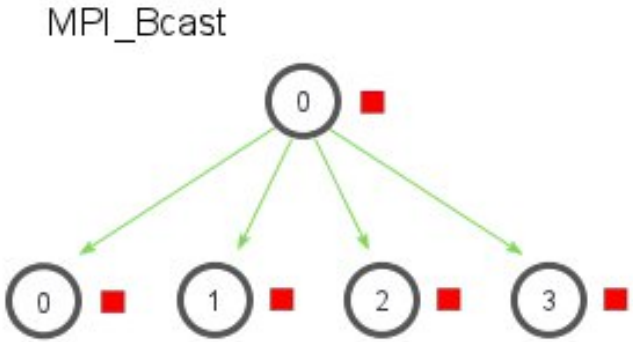
MPI_Bcast

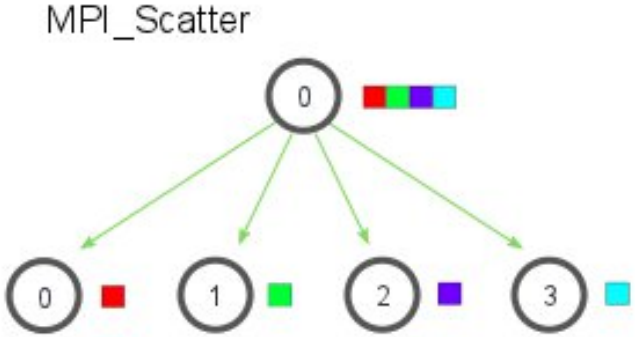
- MPI_Bcast(
- void* data,
- int count,
- MPI_Datatype datatype,
- int root,
- MPI_Comm communicator)

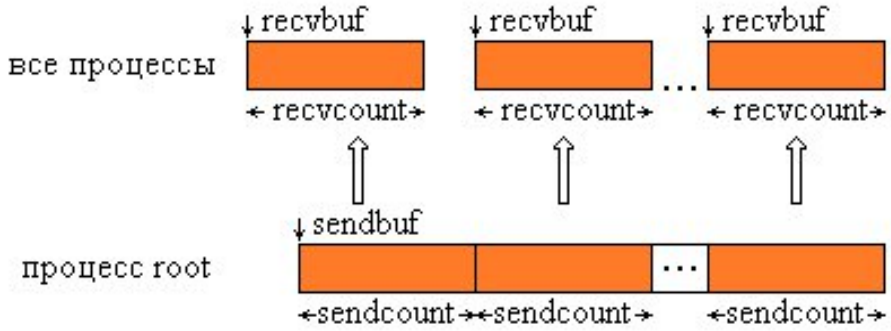


MPI_Scatter

- MPI_Scatter(
- void* send_data,
- int send_count,
- MPI_Datatype send_datatype,
- void* recv_data,
- int recv_count,
- MPI_Datatype recv_datatype,
- int root,
- MPI_Comm communicator)

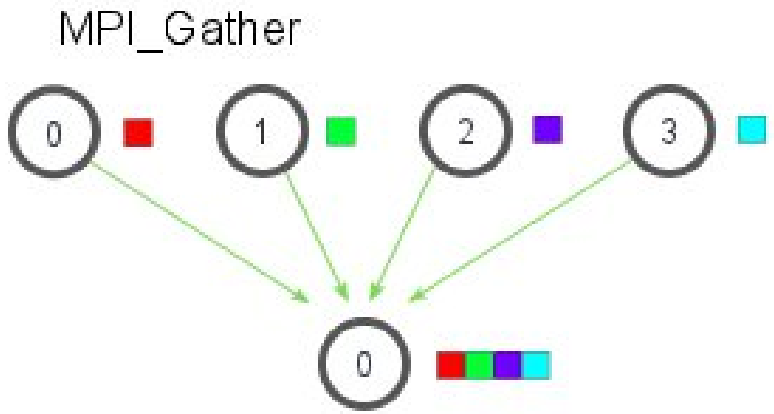


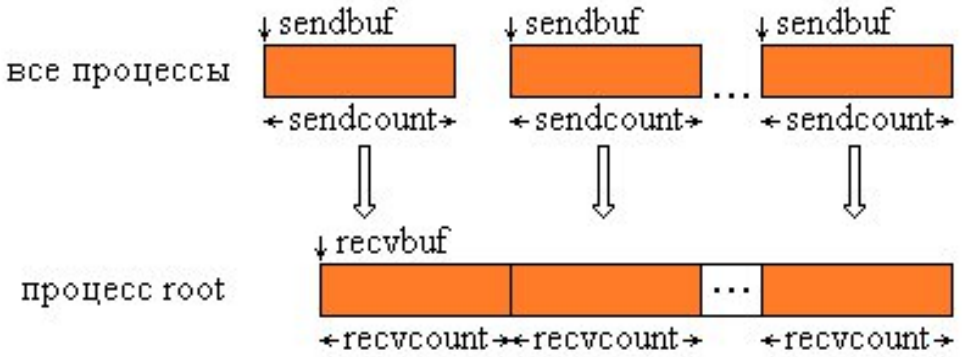




MPI_Gather

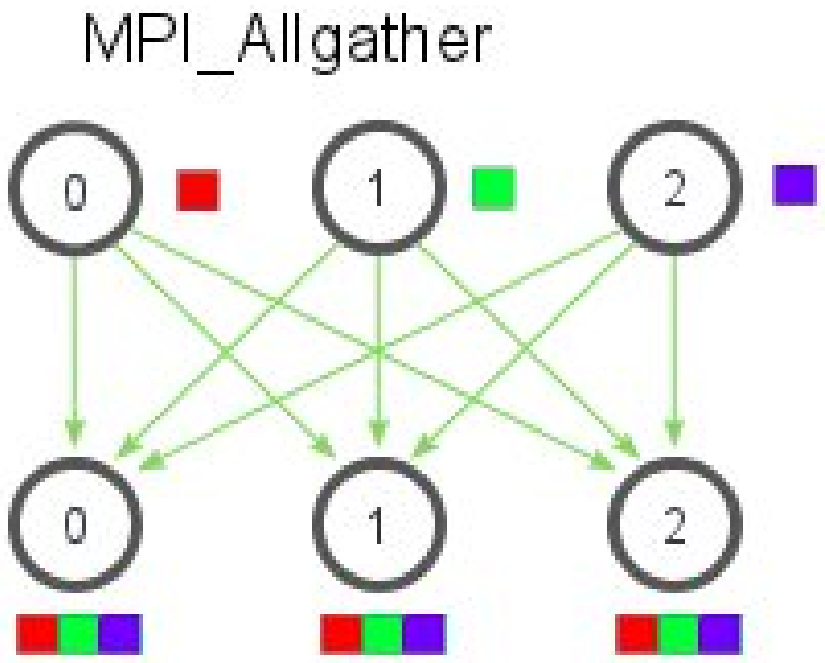
- MPI_Gather(
- void* send_data,
- int send_count,
- MPI_Datatype send_datatype,
- void* recv_data,
- int recv_count,
- MPI_Datatype recv_datatype,
- int root,
- MPI_Comm communicator)





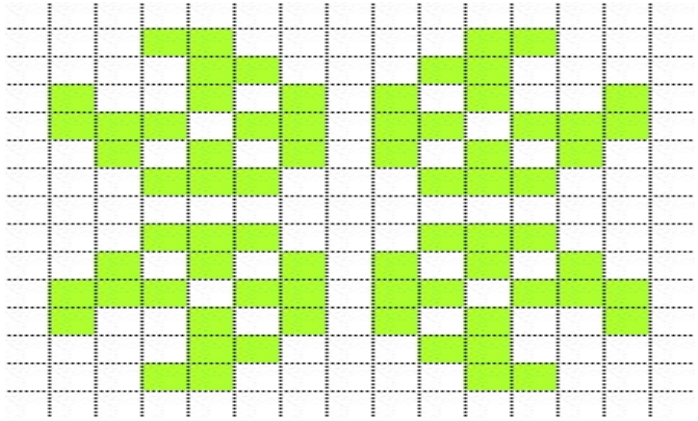
MPI_Allgather

- MPI_Allgather(
- void* send_data,
- int send_count,
- MPI_Datatype send_datatype,
- void* recv_data,
- int recv_count,
- MPI_Datatype recv_datatype,
- MPI_Comm communicator)



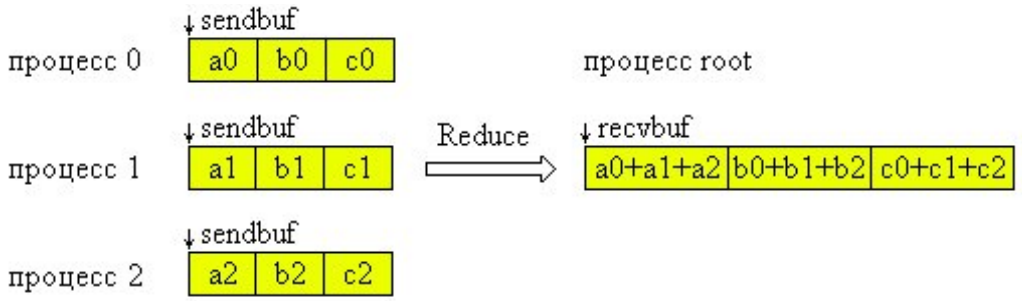
Игра жизнь

- Поле N на N клеток
- Клетка может быть живой или мертвой
- В каждый следующий момент времени:
  - Если у клетки 3 соседа - в ней появляется жизнь
  - Если у клетки один сосед или меньше - она умирает
  - Если у клетки больше трех соседей - она умирает



MPI_Reduce

- int MPI_Reduce(
- void* sendbuf,
- void* recvbuf,
- int count,
- MPI_Datatype datatype,
- MPI_Op op, операция, по которой выполняется редукция
- int root,
- MPI_Comm comm)



MPI_Op

Название	Операция	Разрешенные типы
MPI_MAX MPI_MIN	Максимум Минимум	C integer, FORTRAN integer, Floating point
MPI_SUM MPI_PROD	Сумма Произведение	C integer, FORTRAN integer, Floating point, Complex
MPI_LAND MPI_LOR MPI_LXOR	Логическое AND Логическое OR Логическое исключающее OR	C integer, Logical
MPI_BAND MPI_BOR MPI_BXOR	Поразрядное AND Поразрядное OR Поразрядное исключающее OR	C integer, FORTRAN integer, Byte
MPI_MAXLOC MPI_MINLOC	Максимальное значение и его индекс Минимальное значение и его индекс	Специальные типы для этих функций

Операции **MAXLOC** и **MINLOC** выполняются над специальными парными типами, каждый элемент которых хранит две величины: значения, по которым ищется максимум или минимум, и индекс элемента. В MPI имеется 9 таких предопределенных типов.

В таблице используются следующие обозначения:

C integer:	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG
FORTRAN integer:	MPI_INTEGER
Floating point:	MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_LONG_DOUBLE
Logical:	MPI_LOGICAL
Complex:	MPI_COMPLEX
Byte:	MPI_BYTE

C:

MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int
MPI_LONG_INT	long and int
MPI_2INT	int and int
MPI_SHORT_INT	short and int
MPI_LONG_DOUBLE_INT	long double and int

MINLOC пример

Каждый процесс имеет непустой массив чисел. Требуется найти минимальное глобальное число, номер процесса, хранящего его, и его индекс в этом процессе.

```
#define  LEN   1000

float val[LEN];  /* локальный массив значений */
int count;       /* локальное количество значений */
int myrank, minrank, minindex;
float minval;

struct {
    float value;
    int   index;
} in, out;

    /* локальный minloc */
in.value = val[0];
in.index = 0;
for (i=1; i < count; i++)
    if (in.value > val[i]) {
        in.value = val[i];
        in.index = i;
    }

    /* глобальный minloc */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
in.index = myrank*LEN + in.index;
MPI_Reduce(in, out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm);
    /* в этой точке результат помещается на корневой процесс
     */
if (myrank == root) {
minval = out.value;
    minrank = out.index / LEN;
    minindex = out.index % LEN;
}
```

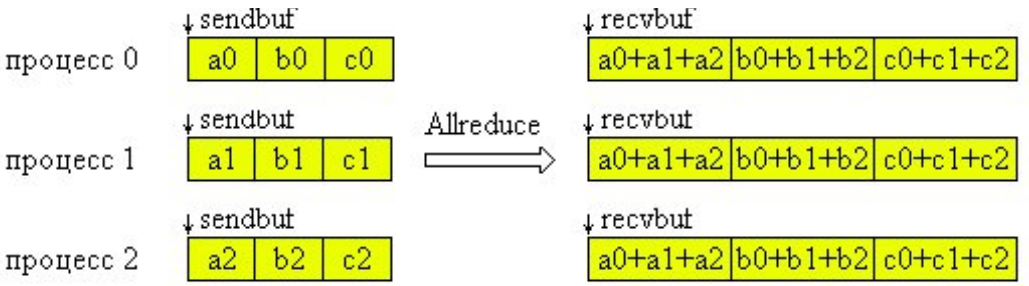
Reduce modifications

- **MPI_Reduce_scatter**
- **MPI_Allreduce**
- **MPI_Scan**

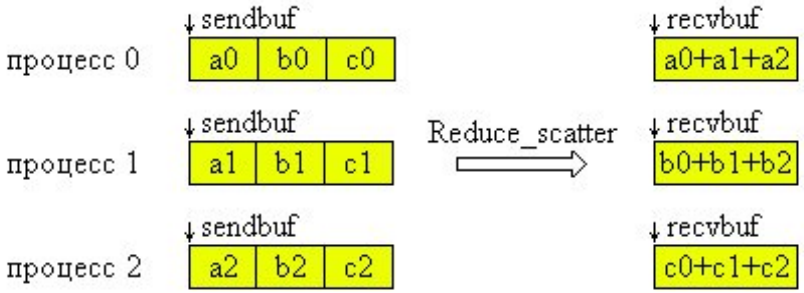
- Параметры:
- **void* sendbuf,**
- **void* recvbuf,**
- i**nt counts,** - для Allreduce и Scan - число элементов во входном буфере, для Reduce_scatter имеет тип **int*** - массив, в котором задаются размеры блоков, посылаемых процессам;
- **MPI_Datatype datatype,**
- **MPI_Op op,**
- **MPI_Comm comm**

MPI_Allreduce vs MPI_Reduce_scatter

Allreduce

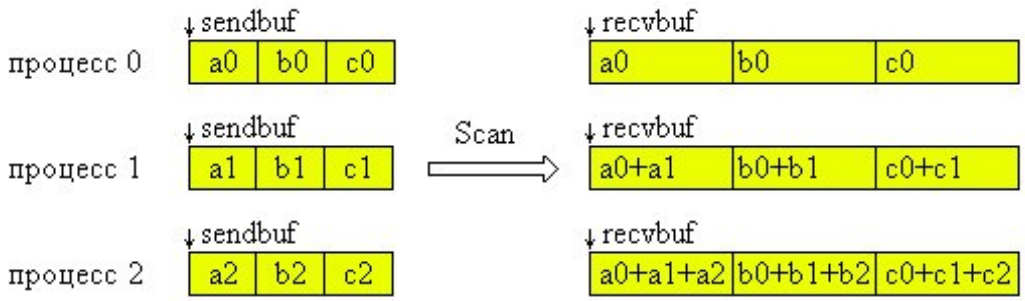


Reduce_scatter



MPI_Scan

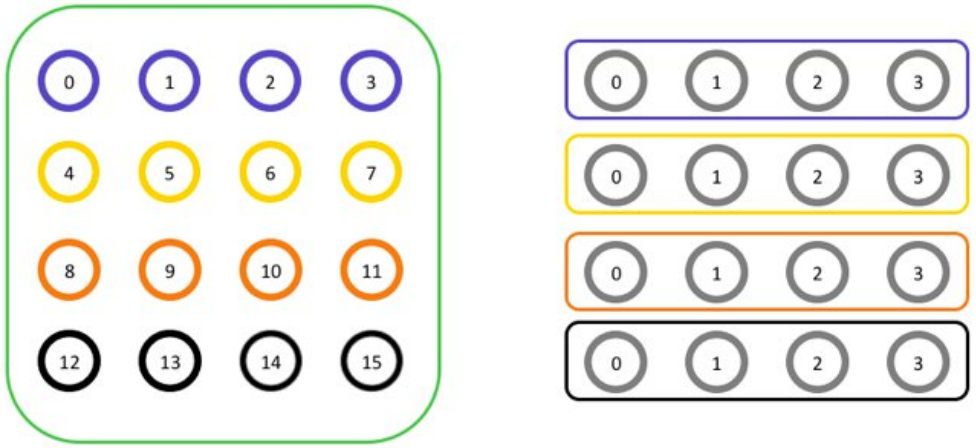
Префиксная редукция.
Каждый процесс получает свои данные.



MPI_Comm

- **MPI_Comm_dup**(MPI_Comm comm, MPI_Comm *newcomm)
- **MPI_Comm_create**(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
- **MPI_Comm_split**(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
- **MPI_Comm_free**(MPI_Comm *comm)

Split a Large Communicator Into Smaller Communicators




```c
// Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",
       world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);
```

MPI_Group

- **MPI_Group_size**(MPI_Group group, int *size)
- **MPI_Group_rank**(MPI_Group group, int *rank)
- **MPI_Comm_group**(MPI_Comm comm, MPI_Group *group)
- **MPI_Group_union/intersection/difference**(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
- **MPI_Group_incl**(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
- **MPI_Group_excl**(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)