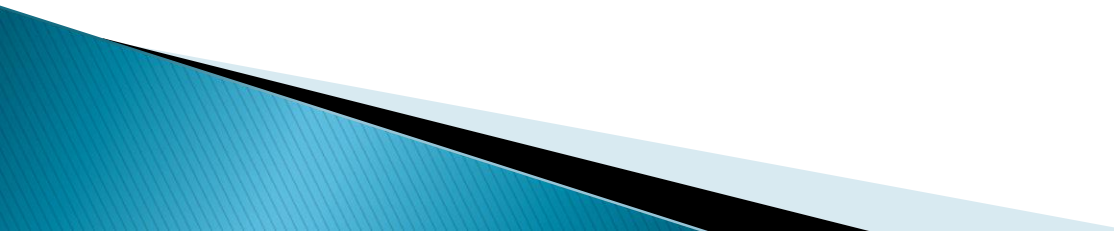


Обзор технологии параллельного программирования MPI

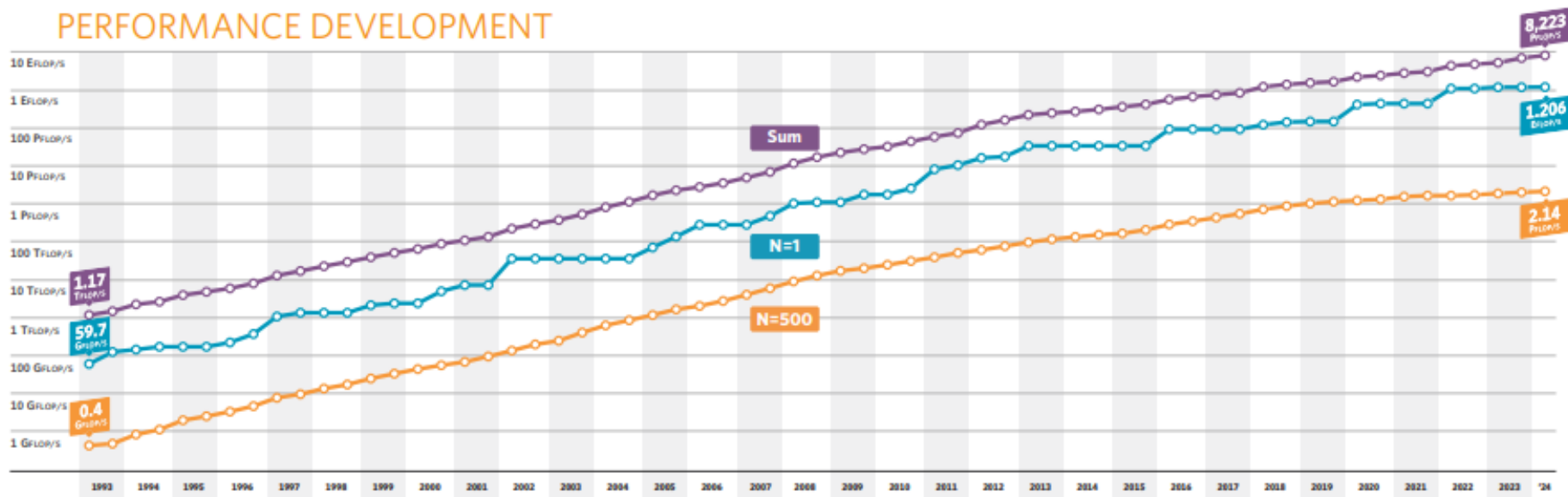
План лекции

- ▶ Стандарт MPI
 - ▶ Основные понятия
 - ▶ Блокирующие двухточечные обмены
 - ▶ Двухточечные обмены с буферизацией, другие типы двухточечных обменов
 - ▶ Коллективные операции
- 

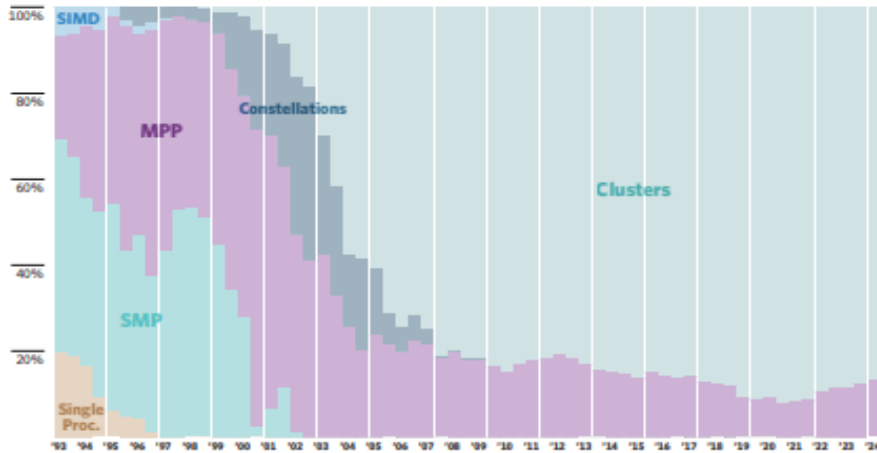
MAY 2024

			SITE	COUNTRY	CORES	RMAX PFL0P/S	POWER MW
1	Frontier	HPE Cray EX235a, AMD Opt 3rd Gen EPYC (64C 2GHz), AMD Instinct MI250X, Slingshot-11	DOE/SC/ORNL	USA	8,699,904	1,206.0	22.7
2	Aurora	HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 (52C 2.4GHz), Intel Data Center GPU Max, Slingshot-11	DOE/SC/ANL	USA	9,264,128	1,012.0	38.7
3	Eagle	Microsoft NDv5, Xeon Platinum 8480C (48C 2GHz), NVIDIA H100, NVIDIA Infiniband NDR	Microsoft Azure	USA	1,123,200	561.2	
4	Fugaku	Fujitsu A64FX (48C, 2.2GHz), Tofu Interconnect D	RIKEN R-CCS	Japan	7,630,848	442.0	29.9
5	LUMI	HPE Cray EX235a, AMD Opt 3rd Gen EPYC (64C 2GHz), AMD Instinct MI250X, Slingshot-11	EuroHPC/CSC	Finland	2,220,288	379.7	6.01

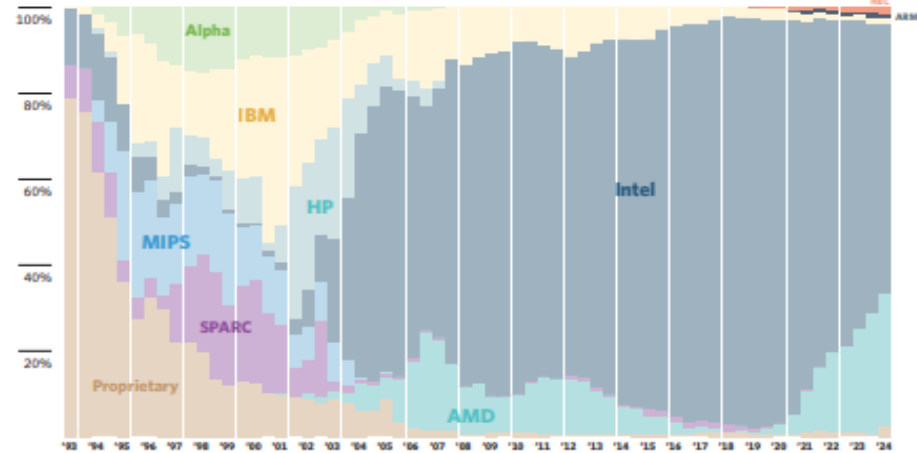
PERFORMANCE DEVELOPMENT



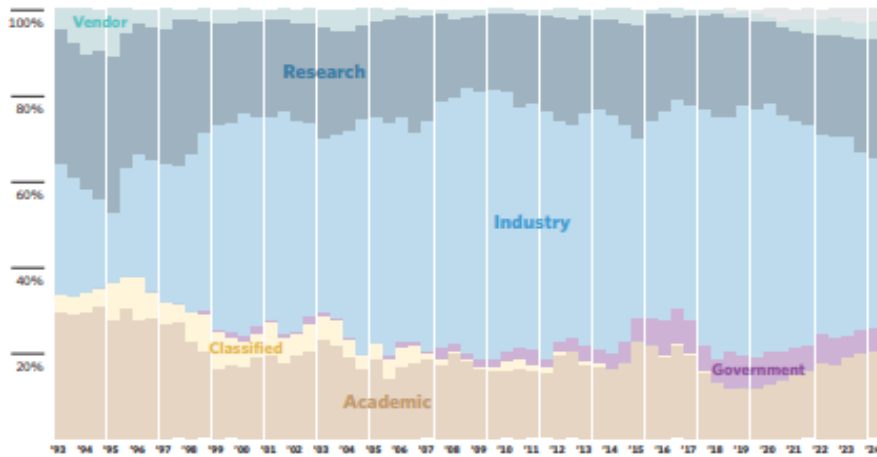
ARCHITECTURES



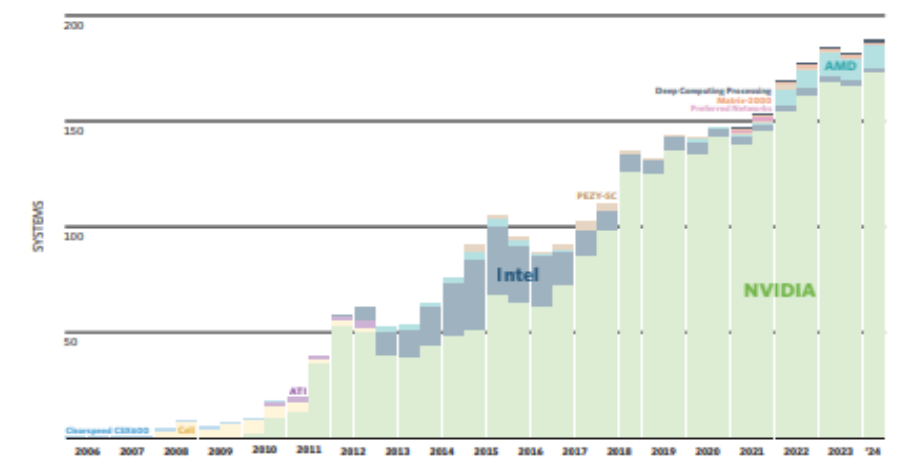
CHIP TECHNOLOGY



INSTALLATION TYPE



ACCELERATORS/CO-PROCESSORS



HPLINPACK

A Portable Implementation of the High Performance Linpack Benchmark for Distributed Memory Computers [FIND OUT MORE AT https://icl.utk.edu/hpl/](https://icl.utk.edu/hpl/)

Рейтинг TOP-500

13 мая 2024 года была опубликована 63-я редакция списка 500 наиболее мощных компьютеров мира Top500.

На первом месте списка остался суперкомпьютер **Frontier** производства HPE Cray на базе процессоров AMD EPYC 64C и графических ускорителей AMD Instinct 250X, установленный в Oak Ridge National Laboratory (ORNL), с пиковой производительностью 1.715 квинтиллионов ($*10^{18}$) операций с плавающей точкой в секунду (EFlop/s) и производительностью на тесте Linpack 1.206 EFlop/s.

На втором месте списка остался суперкомпьютер **Aurora** HPE Cray EX на базе процессоров Intel Xeon CPU Max и графических ускорителей Intel Data Center GPU Max, установленный в Argonne Leadership Computing Facility, чья производительность на тесте Linpack выросла до 1.012 EFlop/s. Aurora официально стала вторым экзафлопсным суперкомпьютером в мире.

На третьем месте списка остался суперкомпьютер **Eagle** на базе процессоров Intel Xeon Platinum 8480C и графических ускорителей NVIDIA H100, установленный в Microsoft Azure Cloud, с производительностью на тесте Linpack 561.2 PFlop/s.

На четвертом месте списка остался японский суперкомпьютер **Fugaku** производства Fujitsu на базе процессоров ARM A64FX, установленный в RIKEN Center for Computational Science (R-CCS), с производительностью на тесте Linpack 442 PFlop/s.

На пятом месте списка остался суперкомпьютер **LUMI** производства HPE Cray на базе процессоров AMD EPYC 64C и графических ускорителей AMD Instinct 250X, установленный в EuroHPC/CSC (Финляндия), чья производительность на тесте Linpack выросла до 379.70 PFlop/s.

Рейтинг TOP-500

Суммарная производительность систем в списке составляет 8.21 EFlop/s (7.03 EFlop/s полгода назад). Последняя, 500-ая система в новой редакции списка была бы полгода назад на 457-ом месте. Для того, чтобы попасть в текущий список, потребовалась производительность на тесте Linpack 2.13 PFlop/s (в ноябре 2.02 PFlop/s).

На первом месте по количеству установленных систем, вошедших в список, остаётся компания Lenovo - 163 системы (169), далее HPE - 112 систем (103), EVIDEN - 49 систем (48), DELL EMC - 34 системы, Inspur - 22 системы (34).

По доле суммарной производительности систем на первом месте HPE - 36.2% (34.9%), далее EVIDEN - 9.6% (9.8%), Lenovo - 7.4% (8.6%), Fujitsu - 7% (8.1%).

Среди стран по количеству установленных систем на первом месте США - 171 система (161 полгода назад), далее Китай - 80 (104), Германия - 40 (36), Япония - 29 (32).

Доля производительности американских систем составляет 53.7% от всех систем списка (53%), доля производительности японских систем - 8.2% (9.5%), а китайских систем - 4.3% (5.8%).

Наиболее популярными коммуникационными технологиями остаются InfiniBand - 239 систем (в прошлом списке 219) и Gigabit Ethernet - 195 систем (209). Коммуникационная технология Omni-Path теперь используется в 32 суперкомпьютерах (33).

Количество систем в списке, построенных на процессорах Intel, уменьшилось с 338 до 314, процессоры AMD используются в 157 системах (140). В список входят 9 систем на базе процессоров Arm. 195 систем использует ускорители или сопроцессоры (в ноябре - 186).

Рейтинг TOP-500. Россия

На 42-ое место списка с 36-го опустился суперкомпьютер **Chervonenkis** производства IPE, Nvidia и Tyan, установленный в Yandex, чья пиковая производительность составляет 29.4 PFlop/s, а производительность на тесте Linpack - 21.5 PFlop/s.

На 69-ое место списка с 58-го опустился суперкомпьютер **Galushkin** производства IPE, Nvidia и Tyan, установленный в Yandex, чья пиковая производительность составляет 20.6 PFlop/s, а производительность на тесте Linpack - 16 PFlop/s.

На 79-ое место списка с 64-го опустился суперкомпьютер **Lyapunov** производства NUDT и Inspur, установленный в Yandex, чья пиковая производительность составляет 20 PFlop/s, а производительность на тесте Linpack - 12.8 PFlop/s.

На 83-е место списка с 67-го опустился суперкомпьютер "**Кристофари Нео**" производства NVIDIA, установленный в Сбербанке, чья пиковая производительность составляет 14.9 PFlop/s, а производительность на тесте Linpack - 12 PFlop/s.

На 142-ое место списка с 119-го опустился суперкомпьютер "**Кристофари**" производства NVIDIA, установленный в Сбербанке, чья пиковая производительность составляет 8.79 PFlop/s, а производительность на тесте Linpack - 6.67 PFlop/s.

На 406-ое место списка со 370-го опустился суперкомпьютер "**Ломоносов-2**" производства компании "Т-Платформы", установленный в Научно-исследовательском вычислительном центре МГУ имени М.В.Ломоносова, чья пиковая производительность составляет 4.95 PFlop/s, а производительность на тесте Linpack - 2.48 PFlop/s.

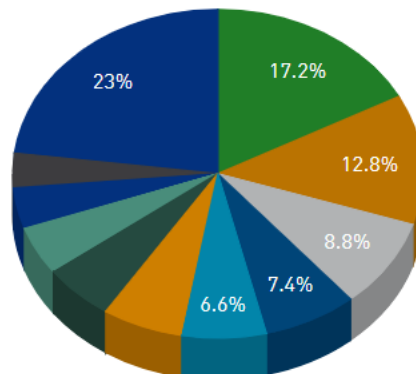
На 472-ое место списка с 433-го опустился суперкомпьютер **MTS GROM** производства NVIDIA, установленный в #CloudMTS, чья пиковая производительность составляет 3.01 PFlop/s, а производительность на тесте Linpack - 2.26 PFlop/s.

Рейтинг TOP-500

	Interconnect	Count	System Share (%)	Rmax (GFlops)	Rpeak (GFlops)	Cores
1	100G Ethernet	86	17.2	245,316,590	493,457,802	7,184,896
2	25G Ethernet	64	12.8	157,549,000	357,813,281	3,737,100
3	10G Ethernet	44	8.8	102,928,230	280,322,604	3,797,280
4	Infiniband HDR	37	7.4	309,119,940	470,352,244	3,685,652
5	Intel Omni-Path	33	6.6	149,385,278	230,799,725	3,437,740
6	Infiniband EDR	32	6.4	135,118,910	275,827,676	4,512,260
7	Mellanox HDR Infiniband	28	5.6	270,614,050	369,870,227	4,114,036
8	Aries interconnect	24	4.8	137,281,734	207,623,044	4,687,596
9	InfiniBand HDR100	20	4	87,450,460	128,260,052	2,572,024
10	Slingshot-10	17	3.4	198,490,180	261,285,460	4,338,192
11	Slingshot-11	15	3	1,673,373,330	2,326,848,381	12,502,848
12	InfiniBand HDR 100	15	3	67,270,730	124,365,914	1,553,176
13	Infiniband FDR	12	2.4	40,510,390	59,228,543	1,466,600
14	Mellanox InfiniBand EDR	10	2	65,522,240	102,087,939	1,082,728
15	Infiniband	8	1.6	79,054,000	112,723,310	716,192
16	Dual-rail Mellanox EDR Infiniband	6	1.2	309,279,000	414,965,922	5,044,160
17	Tofu interconnect D	6	1.2	514,074,800	621,556,837	8,828,928
18	Infiniband HDR200	5	1	38,938,580	49,689,850	356,464
19	Infiniband NDR	4	0.8	9,681,340	15,949,800	79,328
20	Infiniband HDR200	3	0.6	18,687,800	27,683,990	190,304

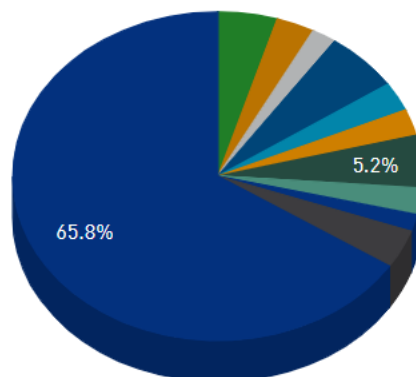
Рейтинг TOP-500

Interconnect System Share



- 100G Ethernet
- 25G Ethernet
- 10G Ethernet
- InfiniBand HDR
- Intel Omni-Path
- InfiniBand EDR
- Mellanox HDR Infiniband
- Aries interconnect
- InfiniBand HDR100
- Slingshot-10
- Others

Interconnect Performance Share



- 100G Ethernet
- 25G Ethernet
- 10G Ethernet
- InfiniBand HDR
- Intel Omni-Path
- InfiniBand EDR
- Mellanox HDR Infiniband
- Aries interconnect
- InfiniBand HDR100
- Slingshot-10
- Others

MPI

- ▶ MPI 1.1 Standard разрабатывался 92–94
- ▶ MPI 2.0 – 95–97
- ▶ MPI 2.1 – сентябрь 2008 г.
- ▶ MPI 3.0 – сентябрь 2012 г.
- ▶ MPI 3.1 – июнь 2015 г.
- ▶ MPI 4.0 – июнь 2021 г.
- ▶ MPI 4.1 – ноябрь 2023 г.

Более 450 процедур

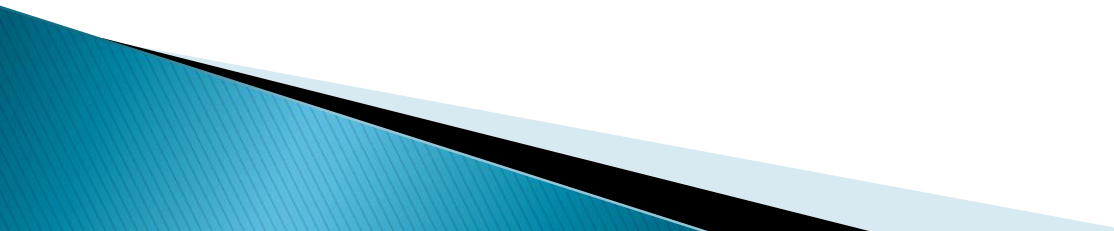
- ▶ Стандарты
<http://www.mpi-forum.org/>
<https://computing.llnl.gov/tutorials/mpi/>
- ▶ Описание функций
<http://www-unix.mcs.anl.gov/mpi/www/>

Цель MPI

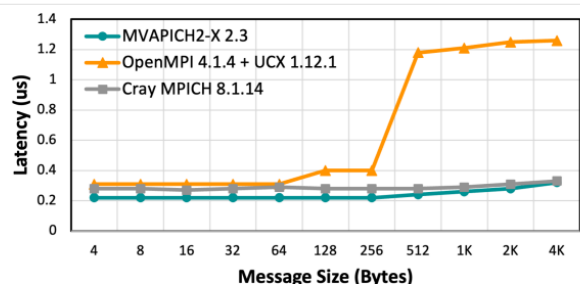
- ▶ Основная цель:
 - Обеспечение переносимости исходных кодов
 - Эффективная реализация

- ▶ Кроме того:
 - Большая функциональность
 - Поддержка неоднородных параллельных архитектур

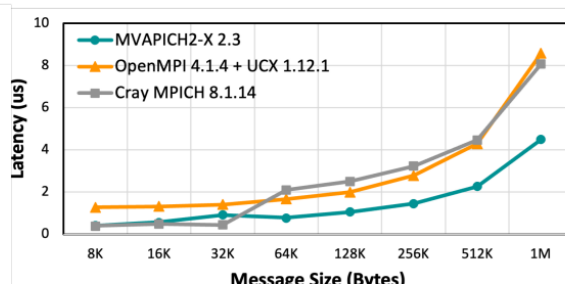
Реализации MPI

- ▶ MPICH
 - ▶ LAM/MPI
 - ▶ Mvarich
 - ▶ OpenMPI
 - ▶ Коммерческие реализации Intel, IBM, Cray и др.
- 

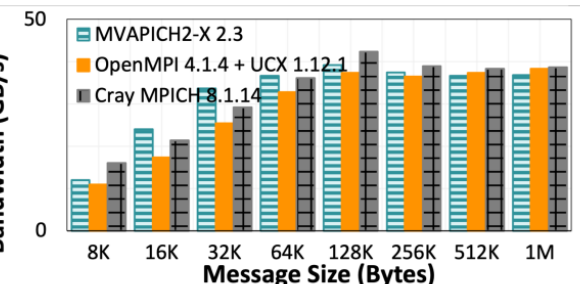
Point-to-Point Performance - Intra-Node CPU



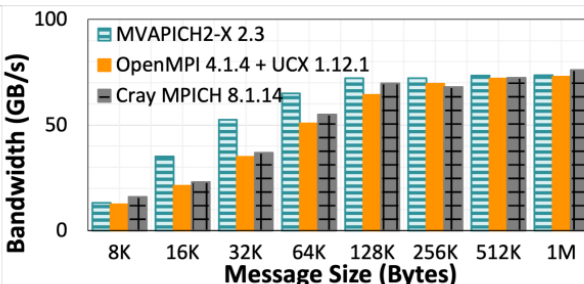
Latency (small messages)



Latency (large messages)



Bandwidth



Bi-Directional Bandwidth



Peak Bandwidth:

- MVAPICH2-X 39.2 GB/s
- OpenMPI+UCX 38.2GB/s
- CrayMPICH 42 GB/s

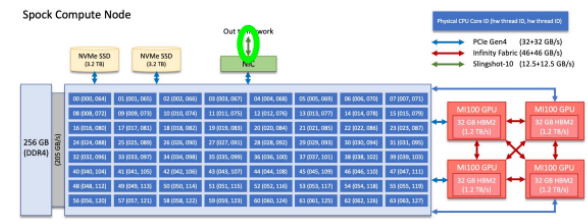
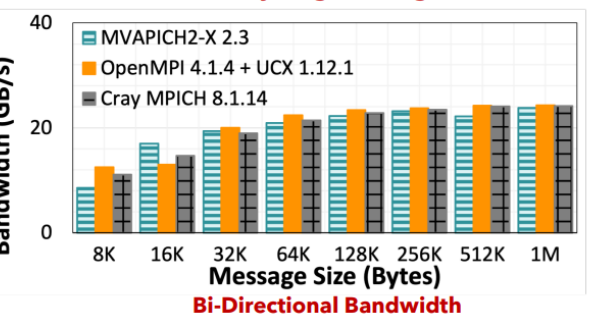
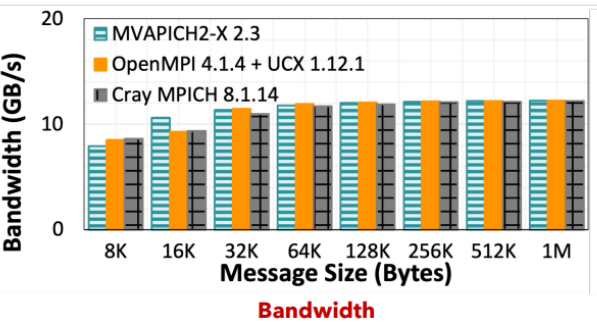
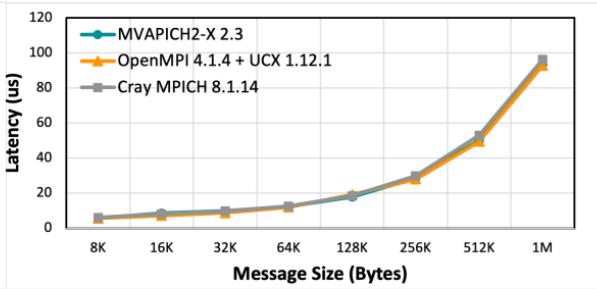
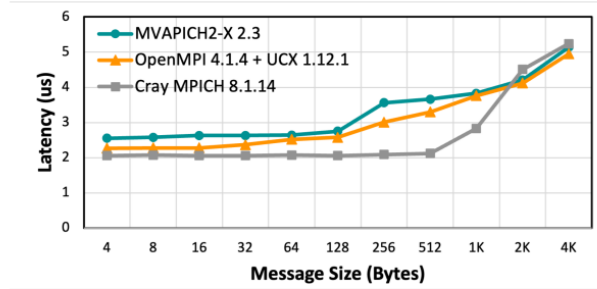
Latency at 4 Bytes:

- MVAPICH2-X 0.22 us
- OpenMPI+UCX 0.31 us
- CrayMPICH 0.27 us

AMD Epyc Rome CPUs on Spock System

Reference: High Performance MPI over the Slingshot Interconnect: Early Experiences K. Khorassani, C. Chen, B. Ramesh, A. Shafi, H. Subramoni, D. Panda Practice and Experience in Advanced Research Computing, Jul 2022.

Point-to-Point Performance - Inter-Node CPU



Slingshot-10 Interconnect for over network communication (12.5+12.5 GB/s)

- Peak Bandwidth:
- MVAPICH2-X 122.4 MB/s
 - OpenMPI+UCX 122.4 MB/s
 - CrayMPICH 122.4 MB/s

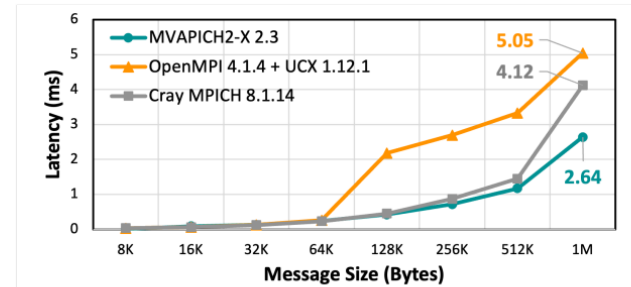
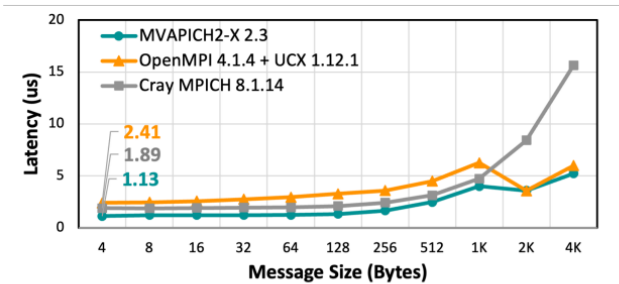
- Latency at 4 Bytes:
- MVAPICH2-X 2.55 us
 - OpenMPI+UCX 2.27 us
 - CrayMPICH 2.07 us

AMD Epyc Rome CPUs on Spock System

Reference: High Performance MPI over the Slingshot Interconnect: Early Experiences K. Khorassani, C. Chen, B. Ramesh, A. Shafi, H. Subramoni, D. Panda Practice and Experience in Advanced Research Computing, Jul 2022.

Collectives Performance - CPU

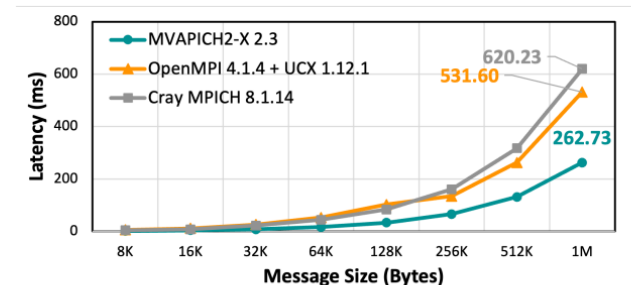
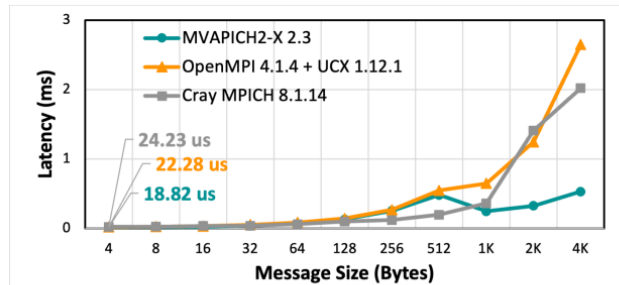
GATHER



ALLGATHER

Gather (small messages)

Gather (large messages)



Allgather (small messages)

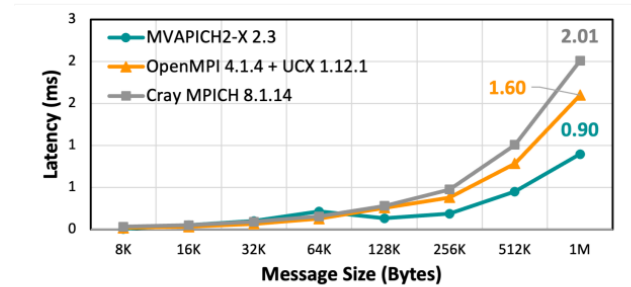
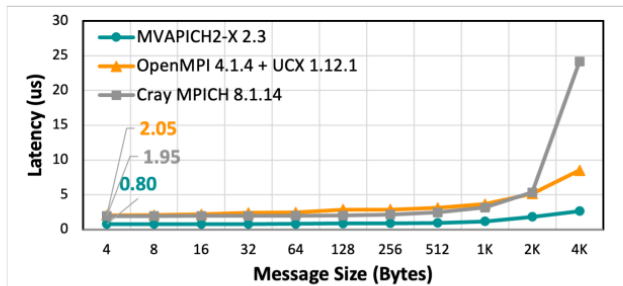
Allgather (large messages)

256 CPUs - 4 Nodes & 64 PPN on Spock System

Reference: High Performance MPI over the Slingshot Interconnect: Early Experiences K. Khorassani, C. Chen, B. Ramesh, A. Shafi, H. Subramoni, D. Panda Practice and Experience in Advanced Research Computing, Jul 2022.

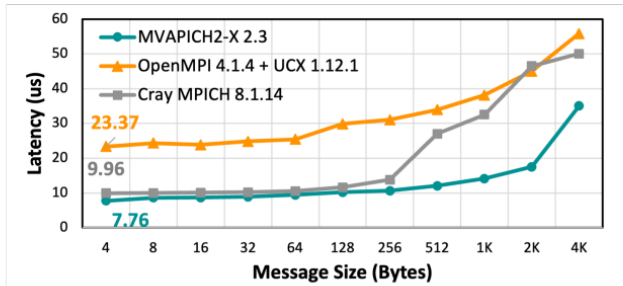
Collectives Performance - CPU

REDUCE

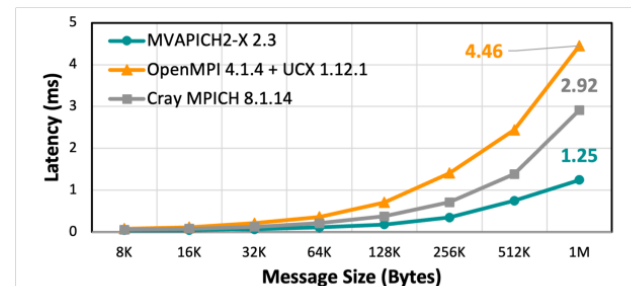


ALLREDUCE

Reduce (small messages)



Reduce (large messages)



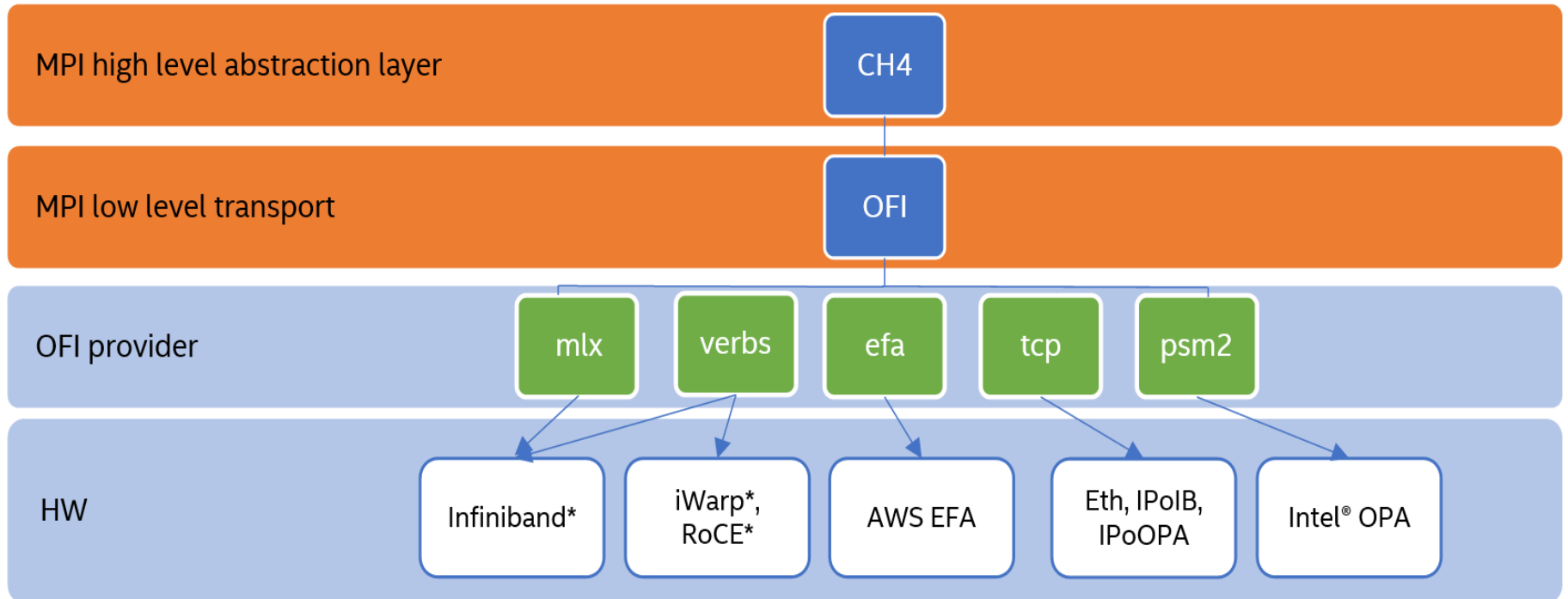
Allreduce (small messages)

Allreduce (large messages)

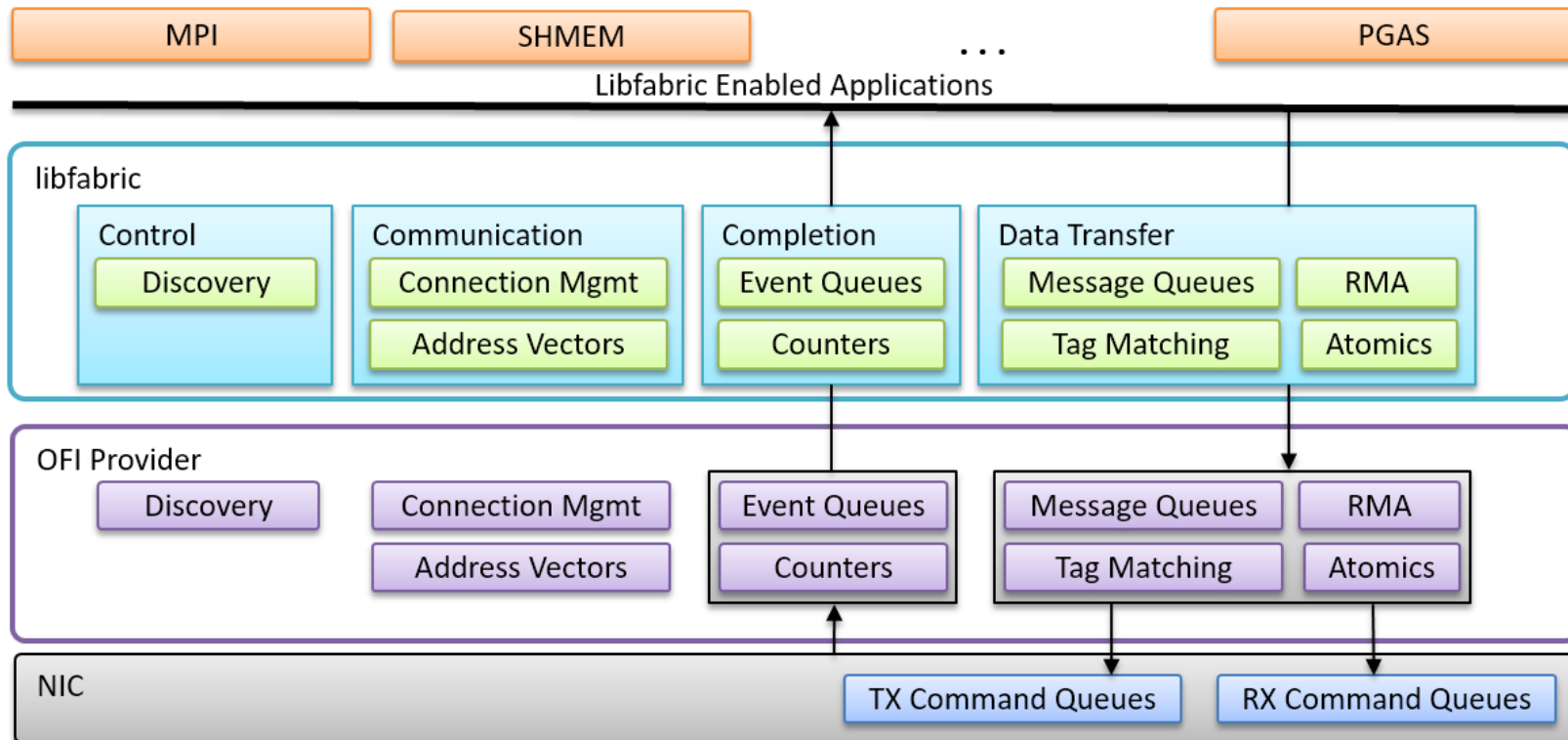
256 CPUs - 4 Nodes & 64 PPN on Spock System

Reference: High Performance MPI over the Slingshot Interconnect: Early Experiences K. Khorassani, C. Chen, B. Ramesh, A. Shafi, H. Subramoni, D. Panda Practice and Experience in Advanced Research Computing, Jul 2022.


Реализация MPI



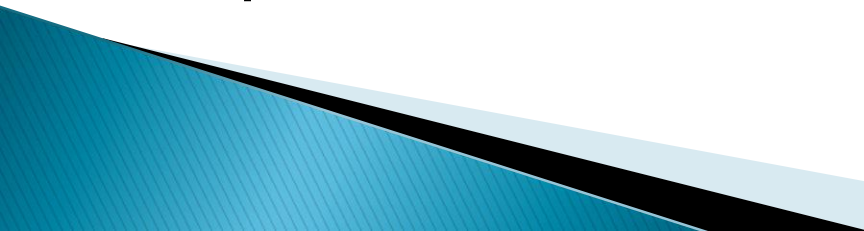
Реализация MPI. Libfabric OpenFabrics



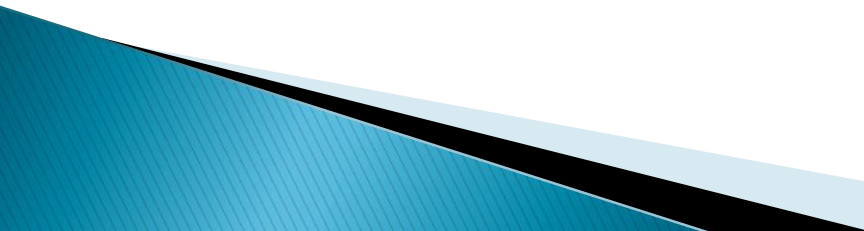
Модель MPI

- ▶ Параллельная программа состоит из процессов, процессы могут быть многопоточными.
 - ▶ MPI реализует передачу сообщений между процессами.
 - ▶ Межпроцессное взаимодействие предполагает:
 - синхронизацию
 - перемещение данных из адресного пространства одного процесса в адресное пространство другого процесса.
- 

Основные понятия

- ▶ Процессы объединяются в группы.
 - ▶ Каждое сообщение посылается в рамках некоторого контекста и должно быть получено в том же контексте.
 - ▶ Группа и контекст вместе определяют коммуникатор.
 - ▶ Процесс идентифицируется своим номером в группе, ассоциированной с коммуникатором.
 - ▶ Коммуникатор, содержащий все начальные процессы, называется `MPI_COMM_WORLD`.
- 

Понятие коммутатора MPI

- ▶ Управляющий объект, представляющий группу процессов, которые могут взаимодействовать друг с другом
 - ▶ Все обращения к MPI функциям содержат коммутатор, как параметр
 - ▶ Наиболее часто используемый коммутатор `MPI_COMM_WORLD`
 - ▶ Определяется при вызове `MPI_Init`
 - ▶ Содержит ВСЕ процессы программы
- 

Типы данных MPI

- ▶ Данные в сообщении описываются тройкой: (address, count, datatype), где datatype определяется рекурсивно как:
 - Предопределенный базовый тип, соответствующий типу данных в базовом языке (например, MPI_INT, MPI_DOUBLE_PRECISION)
 - Непрерывный массив MPI типов
 - Векторный тип
 - Индексированный тип
 - Произвольные структуры
- ▶ MPI включает функции для построения пользовательских типов данных, например, типа данных, описывающих пары (int, float).

Базовые типы данных

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

Понятие тега

- ▶ Сообщение сопровождается определяемым пользователем признаком для идентификации принимаемого сообщения.
- ▶ Теги сообщений у отправителя и получателя должны быть согласованы.
- ▶ Можно указать в качестве значения тэга константу `MPI_ANY_TAG`.
- ▶ Некоторые не-MPI системы передачи сообщений называют тэг типом сообщения.
- ▶ MPI вводит понятие тэга, чтобы не путать это понятие с типом данных MPI.

Формат MPI-функций

```
error = MPI_Xxxxx(parameter,...);  
MPI_Xxxxx(parameter,...);
```

- ▶ Возвращаемое значение – код ошибки. Определяется константой MPI_SUCCESS

```
int error;  
.....  
error = MPI_Init(&argc, &argv);  
if (error != MPI_SUCCESS)  
{  
    fprintf (stderr, " MPI_Init error \n");  
    return 1;  
}
```

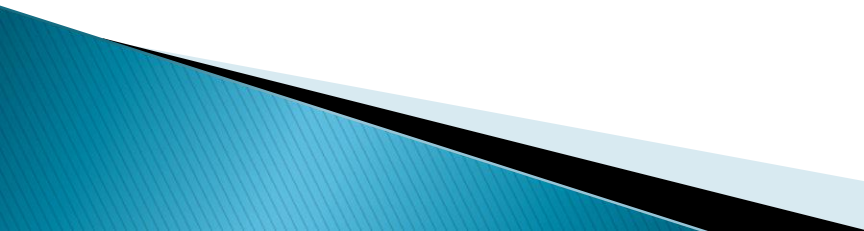
Выполнение MPI-программы

- ▶ При запуске указываем число требуемых процессоров `np` и название программы
`mpirun -np 3 prog`
- ▶ На выделенных для расчета узлах запускается `np` копий указанной программы
- ▶ Каждая копия программы получает два значения:
 - `np`
 - `rank` из диапазона $[0 \dots np-1]$
- ▶ Любые две копии программы могут непосредственно обмениваться данными с помощью функций передачи сообщений

C: MPI helloworld.c

```
#include <mpi.h>
main(int argc, char **argv)
{
    int numtasks, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &
numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Hello World from process %d of %d\n",
rank, numtasks);
    MPI_Finalize();
}
```

Функции определения среды

- ▶ **int MPI_Init(int *argc, char ***argv)**
должна первым вызовом, вызывается только один раз
 - ▶ **int MPI_Comm_size(MPI_Comm comm, int *size)**
число процессов в коммуникаторе
 - ▶ **int MPI_Comm_rank(MPI_Comm comm, int *rank)**
номер процесса в коммуникаторе (нумерация с 0)
 - ▶ **int MPI_Finalize()**
завершает работу процесса
 - ▶ **int MPI_Abort (MPI_Comm comm, int*errorcode)**
завершает работу программы
- 

Функции определения среды

- ▶ **int MPI_Initialized(int *flag)**

В аргументе **flag** возвращает 1, если вызвана после процедуры **MPI_Init**, и 0 в противном случае.

- ▶ **int MPI_Finalized(int *flag)**

В аргументе **flag** возвращает 1, если вызвана после процедуры **MPI_Finalize**, и 0 в противном случае.

Эти процедуры можно вызвать до **MPI_Init** и после **MPI_Finalize**.

- ▶ **int MPI_Get_processor_name(char *name, int *len)**

Возвращает в строке **name** имя узла, на котором запущен вызвавший процесс. В переменной **len** возвращается количество символов в имени, не превышающее константы **MPI_MAX_PROCESSOR_NAME**.

Функции работы с таймером

- ▶ **double MPI_Wtime(void)**

Возвращает для каждого вызвавшего процесса астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом. Момент времени, используемый в качестве точки отсчёта, не будет изменён за время существования процесса.

- ▶ **double MPI_Wtick(void)**

Возвращает разрешение таймера в секундах.

Использование таймера

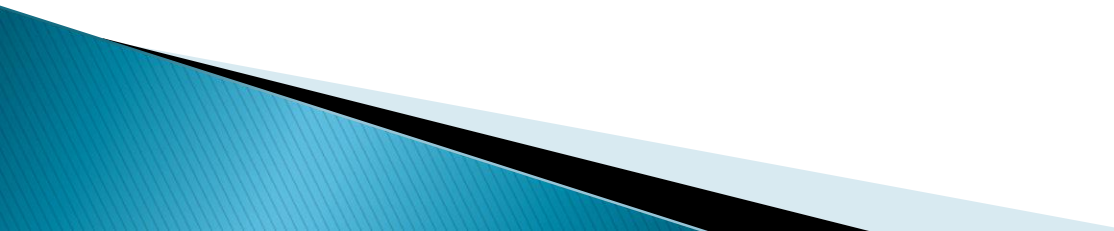
```
#include <stdio.h>
#include "mpi.h"
#define NTIMES 1000
int main(int argc, char **argv)
{
    double time_start, time_finish, tick;
    int rank, i;
    int len;
    char *name;
    name = (char*)malloc(MPI_MAX_PROCESSOR_NAME*sizeof(char));
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(name, &len);
    tick = MPI_Wtick();
    time_start = MPI_Wtime();
    for (i = 0; i < NTIMES; i++) time_finish = MPI_Wtime();
    printf ("node %s, process %d: tick= %lf, time= %lf\n",
           name, rank, tick, (time_finish-time_start)/NTIMES);
    MPI_Finalize();
}
```

Информация о статусе сообщения

Содержит:

- ▶ Source: `status.MPI_SOURCE`
- ▶ Tag: `status.MPI_TAG`
- ▶ Код ошибки: `status.MPI_ERROR`
- ▶ Count: `MPI_Get_count`

Двухточечный (point-to-point, p2p) обмен

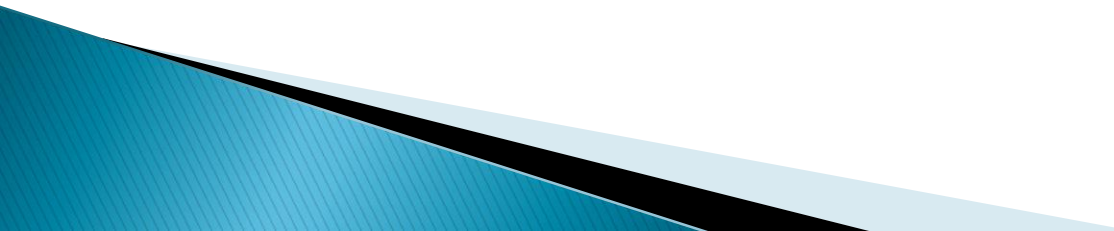
- ▶ В двухточечном обмене участвуют только два процесса, процесс-отправитель и процесс-получатель (источник сообщения и адресат).
 - ▶ Двухточечные обмены используются для организации локальных и неструктурированных коммуникаций.
- 

Двухточечный (point-to-point, p2p) обмен

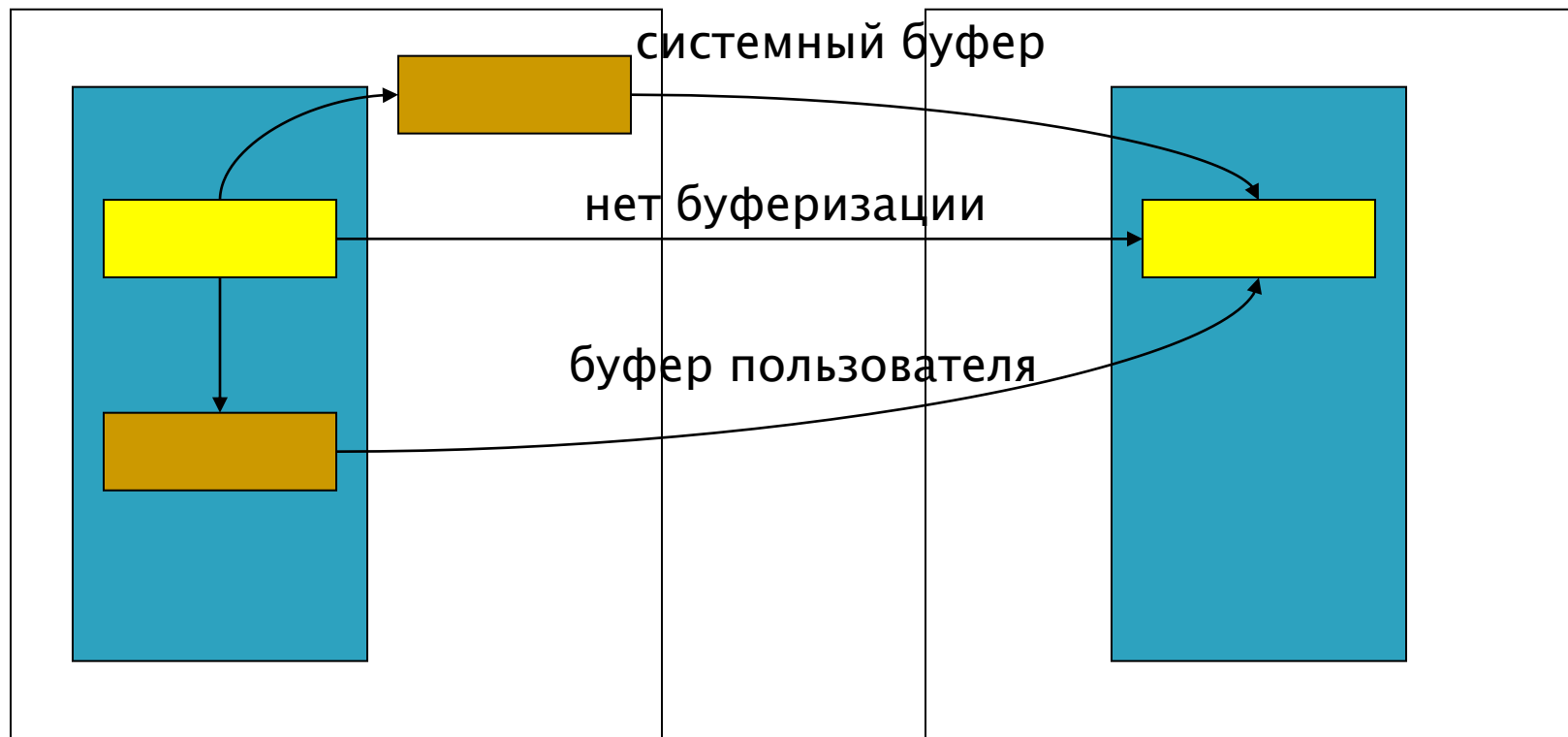


- ▶ Двухточечный обмен возможен только между процессами, принадлежащими одной области взаимодействия (одному коммутатору).

Условия успешного взаимодействия точка–точка

- ▶ Отправитель должен указать правильный rank получателя
 - ▶ Получатель должен указать верный rank отправителя
 - ▶ Одинаковый коммутатор
 - ▶ Тэги должны соответствовать друг другу
 - ▶ Буфер у процесса–получателя должен быть достаточного объема
- 

Выполнение двухточечных обменов



Разновидности двухточечного обмена

- ▶ *блокирующие* прием/передача, которые приостанавливают выполнение процесса на время приема или передачи сообщения;
- ▶ *неблокирующие* прием/передача, при которых выполнение процесса продолжается в фоновом режиме, а программа в нужный момент может запросить подтверждение завершения приема сообщения.

Двухточечный (point-to-point, p2p) обмен

- ▶ Правильно организованный двухточечный обмен сообщениями должен исключать возможность блокировки или некорректной работы параллельной MPI-программы.
- ▶ Примеры ошибок в организации двухточечных обменов:
 - ❑ выполняется передача сообщения, но не выполняется его прием;
 - ❑ процесс-источник и процесс-получатель одновременно пытаются выполнить блокирующие передачу или прием сообщения.

Двухточечный (point-to-point, p2p) обмен

```
#include <mpi.h>
int main (int argc, char **argv)
{
    int rank, size;
    int recv, send, left, right;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    left = (rank > 0)? rank - 1 : size-1;
    right = (rank + 1)%size ;
    MPI_Recv( &recv, 1, MPI_INT, left, 100, MPI_COMM_WORLD, &status );
    send = rank;
    MPI_Send( &send, 1, MPI_INT, right, 100, MPI_COMM_WORLD );
    MPI_Finalize( );
    return 0;
}
```

Двухточечный (point-to-point, p2p) обмен

В MPI приняты следующие соглашения об именах подпрограмм двухточечного обмена:

MPI_[I][R, S, B]Send

здесь префикс [I] (Immediate) обозначает неблокирующий режим.

Один из префиксов [R, S, B] обозначает режим обмена: по готовности, синхронный и буферизованный.

Отсутствие префикса обозначает подпрограмму стандартного обмена.

Имеется 8 разновидностей операции передачи сообщений.

Для подпрограмм приема:

MPI_[I]Recv

то есть всего 2 разновидности приема.

Подпрограмма `MPI_Irsend`, например, выполняет передачу «по готовности» в неблокирующем режиме, `MPI_Bsend` буферизованную передачу с блокировкой, а `MPI_Recv` выполняет блокирующий прием сообщений.

Подпрограмма приема любого типа может принять сообщения от любой подпрограммы передачи.

Стандартная блокирующая передача

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

MPI_Send(buf, count, datatype, dest, tag, comm, ierr)

- ▶ buf – адрес первого элемента в буфере передачи;
- ▶ count – количество элементов в буфере передачи (допускается count = 0);
- ▶ datatype – тип MPI каждого пересылаемого элемента;
- ▶ dest – ранг процесса-получателя сообщения (целое число от 0 до n - 1, где n – число процессов в области взаимодействия);
- ▶ tag – тег сообщения;
- ▶ comm – коммуникатор;
- ▶ ierr – код завершения.

Стандартная блокирующая передача

- ▶ При стандартной блокирующей передаче после завершения вызова (после возврата из функции/процедуры передачи) можно использовать любые переменные, использовавшиеся в списке параметров. Такое использование не повлияет на корректность обмена.
- ▶ Дальнейшая «судьба» сообщения зависит от реализации MPI. Сообщение может быть сразу передано процессу-получателю или может быть скопировано в буфер передачи.
- ▶ Завершение вызова не гарантирует доставки сообщения по назначению.

Стандартный блокирующий прием

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

```
MPI_Recv(buf, count, datatype, dest, tag, comm, status, ierr)
```

- ▶ buf – адрес первого элемента в буфере приёма;
- ▶ count – количество элементов в буфере приёма;
- ▶ datatype – тип MPI каждого пересылаемого элемента;
- ▶ source – ранг процесса-отправителя сообщения (целое число от 0 до $n - 1$, где n – число процессов в области взаимодействия);
- ▶ tag – тег сообщения;
- ▶ comm – коммуникатор;
- ▶ status – статус обмена;
- ▶ ierr – код завершения.

Стандартный блокирующий прием

- ▶ Значение параметра `count` может оказаться больше, чем количество элементов в принятом сообщении. В этом случае после выполнения приёма в буфере изменится значение только тех элементов, которые соответствуют элементам фактически принятого сообщения.
- ▶ Для функции `MPI_Recv` гарантируется, что после завершения вызова сообщение принято и размещено в буфере приема.

Прием сообщения

Если один процесс последовательно посылает два сообщения, соответствующие одному и тому же вызову `MPI_Recv`, другому процессу, то первым будет принято сообщение, которое было отправлено раньше.

Если два сообщения были одновременно отправлены разными процессами, то порядок их получения принимающим процессом заранее не определён.

Коды завершения

- ▶ `MPI_ERR_COMM` – неправильно указан коммуникатор. Часто возникает при использовании «пустого» коммуникатора;
- ▶ `MPI_ERR_COUNT` – неправильное значение аргумента `count` (количество пересылаемых значений);
- ▶ `MPI_ERR_TYPE` – неправильное значение аргумента, задающего тип данных;
- ▶ `MPI_ERR_TAG` – неправильно указан тег сообщения;
- ▶ `MPI_ERR_RANK` – неправильно указан ранг источника или адресата сообщения;
- ▶ `MPI_ERR_ARG` – неправильный аргумент, ошибочное задание которого не попадает ни в один класс ошибок;
- ▶ `MPI_ERR_REQUEST` – неправильный запрос на выполнение операции.

Джокеры

- ▶ В качестве ранга источника сообщения и в качестве тега сообщения можно использовать «джокеры» :
 - `MPI_ANY_SOURCE` – любой источник;
 - `MPI_ANY_TAG` – любой тег.
- ▶ При использовании «джокеров» есть опасность приема сообщения, не предназначенного данному процессу

Двухточечные обмены

Подпрограмма `MPI_Recv` может принимать сообщения, отправленные в любом режиме.

Прием может выполняться от произвольного процесса, а в операции передачи должен быть указан вполне определенный адрес.

Приемник может использовать «джокеры» для источника и для тега. Процесс может отправить сообщение и самому себе, но следует учитывать, что использование в этом случае блокирующих операций может привести к «тупику».

Двухточечные обмены

Размер полученного сообщения (count) можно определить с помощью вызова подпрограммы

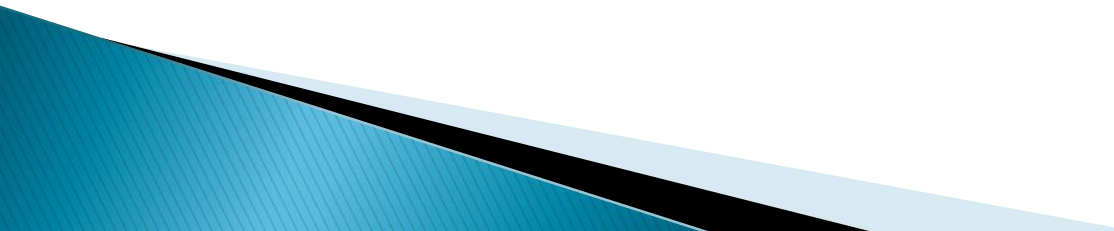
```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```

```
MPI_Get_count(status, datatype, count, ierr)
```

- ▶ count – количество элементов в буфере передачи;
- ▶ datatype – тип MPI каждого пересылаемого элемента;
- ▶ status – статус обмена;
- ▶ ierr – код завершения.

Аргумент datatype должен соответствовать типу данных, указанному в операции обмена

Двухточечный обмен с буферизацией

- ▶ Передача сообщения в буферизованном режиме может быть начата независимо от того, зарегистрирован ли соответствующий прием. Источник копирует сообщение в буфер, а затем передает его в неблокирующем режиме, так же как в стандартном режиме.
 - ▶ Эта операция локальна, поскольку ее выполнение не зависит от наличия соответствующего приема.
 - ▶ Если объем буфера недостаточен, возникает ошибка. Выделение буфера и его размер контролируются программистом.
- 

Двухточечный обмен с буферизацией

- ▶ Размер буфера должен превосходить размер сообщения на величину `MPI_BSEND_OVERHEAD`. Это дополнительное пространство используется подпрограммой буферизованной передачи для своих целей.
- ▶ Если перед выполнением операции буферизованного обмена не выделен буфер, MPI ведет себя так, как если бы с процессом был связан буфер нулевого размера. Работа с таким буфером обычно завершается сбоем программы.
- ▶ Буферизованный обмен рекомендуется использовать в тех ситуациях, когда программисту требуется большой контроль над распределением памяти. Этот режим удобен и для отладки, поскольку причину переполнения буфера определить легче, чем причину тупика.

Двухточечный обмен с буферизацией

- ▶ При выполнении буферизованного обмена программист должен заранее создать буфер достаточного размера:

```
int MPI_Buffer_attach(void *buf, size)
```

```
MPI_Buffer_attach(buf, size, ierr)
```

- ▶ В результате вызова создается буфер `buf` размером `size` байтов. В программах на языке Fortran роль буфера может играть массив. За один раз к процессу может быть подключен только один буфер.

Двухточечный обмен с буферизацией

- ▶ Буферизованная передача завершается сразу, поскольку сообщение немедленно копируется в буфер для последующей передачи:

```
int MPI_Bsend(void *buf, int count,  
             MPI_Datatype datatype, int dest, int tag,  
             MPI_Comm comm)
```

```
MPI_Bsend(buf, count, datatype, dest, tag,  
          comm, ierr)
```

Двухточечный обмен с буферизацией

- ▶ После завершения работы с буфером его необходимо отключить:

```
int MPI_Buffer_detach(void *buf, int *size)
```

```
MPI_Buffer_detach(buf, size, ierr)
```

- ▶ Возвращается адрес (`buf`) и размер отключаемого буфера (`size`). Эта операция блокирует работу процесса до тех пор, пока все сообщения, находящиеся в буфере, не будут обработаны. Вызов данной подпрограммы можно использовать для форсированной передачи сообщений. После завершения вызова можно вновь использовать память, которую занимал буфер. В языке С данный вызов не освобождает автоматически память, отведенную для буфера.

Двухточечный обмен с буферизацией

```
#include <mpi.h>
#include <stdio.h>
#define M 10
int main( int argc, char **argv )
{
    int n;
    int rank, size;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    if ( rank == 0 ) {
        int blen = M * (sizeof(int) + MPI_BSEND_OVERHEAD);
        int *buf = (int*) malloc(blen);
        MPI_Buffer_attach (buf, blen);
        for(int i = 0; i < M; i ++){
            n = i;
            MPI_Bsend (&n, 1, MPI_INT, 1, i, MPI_COMM_WORLD );
        }
        MPI_Buffer_detach(&buf, &blen);
        free(buf);
    }
}
```

```
else if ( rank == 1 ) {
    for(int i = 0; i < M; i ++){
        MPI_Recv (&n, 1, MPI_INT, 0, i,
            MPI_COMM_WORLD,&status );
    }
}
MPI_Finalize();
return 0;
}
```

Синхронный режим

- ▶ Завершение передачи происходит только после того, как прием сообщения инициализирован другим процессом.
- ▶ Посылающая сторона запрашивает у принимающей стороны подтверждение выдачи операции receive – «квитанцию».

```
int MPI_Ssend(void *buf, int count,  
MPI_Datatype datatype, int dest, int tag,  
MPI_Comm comm)
```

Режим «ПО ГОТОВНОСТИ»

Передача «по готовности» выполняется с помощью подпрограммы

```
int MPI_Rsend(void *buf, int count, MPI_Datatype  
datatype, int dest, int tag, MPI_Comm comm)
```

```
MPI_Rsend(buf, count, datatype, dest, tag, comm,  
ierr)
```

Передача «по готовности» должна начинаться, если уже зарегистрирован соответствующий прием. При несоблюдении этого условия результат выполнения операции не определен.

Режим «ПО ГОТОВНОСТИ»

Завершается она сразу же. Если прием не зарегистрирован, результат выполнения операции не определен.

Завершение передачи не зависит от того, вызвана ли другим процессом подпрограмма приема данного сообщения или нет, оно означает только, что буфер передачи можно использовать вновь.

Сообщение просто выбрасывается в коммуникационную сеть в надежде, что адресат его получит. Эта надежда может и не сбыться.

Режим «ПО ГОТОВНОСТИ»

Обмен «по готовности» может увеличить производительность программы, поскольку здесь не используются этапы установки межпроцессных связей, а также буферизация.

Все это — операции, требующие времени. С другой стороны, обмен «по готовности» потенциально опасен, кроме того, он усложняет отладку, поэтому его рекомендуется использовать только в том случае, когда правильная работа программы гарантируется ее логической структурой, а выигрыша в быстродействии надо добиться любой ценой.

Совместные прием и передача

- ▶ Подпрограмма `MPI_Sendrecv` выполняет прием и передачу данных с блокировкой:

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

- ▶ Подпрограмма `MPI_Sendrecv_replace` выполняет прием и передачу данных, используя общий буфер для передачи и приёма:

```
int MPI_Sendrecv_replace(void *buf, int count,
MPI_Datatype datatype, int dest, int sendtag, int
source, int recvtag, MPI_Comm comm, MPI_Status
*status)
```

Неблокирующие обмены

- ▶ Вызов подпрограммы неблокирующей передачи инициирует, но не завершает ее. Завершиться выполнение подпрограммы может еще до того, как сообщение будет скопировано в буфер передачи.
- ▶ Применение неблокирующих операций улучшает производительность программы, поскольку в этом случае допускается перекрытие (то есть одновременное выполнение) вычислений и обменов. Передача данных из буфера или их считывание может происходить одновременно с выполнением процессом другой работы.

Неблокирующие обмены

- ▶ Для завершения неблокирующего обмена требуется вызов дополнительной процедуры, которая проверяет, скопированы ли данные в буфер передачи.
- ▶ **ВНИМАНИЕ!**
При неблокирующем обмене возвращение из подпрограммы обмена происходит сразу, но запись в буфер или считывание из него после этого производить нельзя – сообщение может быть еще не отправлено или не получено и работа с буфером может «испортить» его содержимое.

Неблокирующие обмены

Неблокирующий обмен выполняется в два этапа:

1. инициализация обмена;
2. проверка завершения обмена.

Разделение этих шагов делает необходимым *маркировку* каждой операции обмена, которая позволяет целенаправленно выполнять проверки завершения соответствующих операций.

Для маркировки в неблокирующих операциях используются *идентификаторы операций обмена*

Неблокирующие обмены

- ▶ Инициализация неблокирующей стандартной передачи выполняется подпрограммами `MPI_I[S, B, R]send`.
Стандартная неблокирующая передача выполняется подпрограммой:

```
int MPI_Isend(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm,
MPI_Request *request)
```

```
MPI_Isend(buf, count, datatype, dest, tag, comm,
request, ierr)
```

- ▶ Входные параметры этой подпрограммы аналогичны аргументам подпрограммы `MPI_Send`.
- ▶ Выходной параметр `request` – идентификатор операции.

Неблокирующие обмены

- ▶ Инициализация неблокирующего приема выполняется при вызове подпрограммы:

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

```
MPI_Irecv(buf, count, datatype, source, tag,  
comm, request, ierr)
```

- ▶ Назначение аргументов здесь такое же, как и в ранее рассмотренных подпрограммах, за исключением того, что указывается ранг не адресата, а источника сообщения (`source`).

Неблокирующие обмены

- ▶ Вызовы подпрограмм неблокирующего обмена формируют *запрос* на выполнение операции обмена и связывают его с идентификатором операции `request`.
- ▶ Запрос идентифицирует свойства операции обмена:
 - режим;
 - характеристики буфера обмена;
 - контекст;
 - тег и ранг.
- ▶ Запрос содержит информацию о состоянии ожидающих обработки операций обмена и может быть использован для получения информации о состоянии обмена или для ожидания его завершения.

Проверка выполнения обмена

- ▶ Проверка фактического выполнения передачи или приема в неблокирующем режиме осуществляется с помощью вызова *подпрограмм ожидания*, блокирующих работу процесса до завершения операции или неблокирующих подпрограмм проверки, возвращающих логическое значение «истина», если операция выполнена

Проверка выполнения обмена

- ▶ В том случае, когда одновременно несколько процессов обмениваются сообщениями, можно использовать проверки, которые применяются одновременно к нескольким обменам.
- ▶ Есть три типа таких проверок:
 1. проверка завершения всех обменов;
 2. проверка завершения любого обмена из нескольких;
 3. проверка завершения заданного обмена из нескольких.
- ▶ Каждая из этих проверок имеет две разновидности:
 1. «ожидание»;
 2. «проверка».

Блокирующие операции проверки

- ▶ Подпрограмма `MPI_Wait` блокирует работу процесса до завершения приема или передачи сообщения:

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

```
MPI_Wait(request, status, ierr)
```

- ▶ Входной параметр `request` — идентификатор операции обмена, выходной — статус (`status`).

Блокирующие операции проверки

- ▶ Успешное выполнение подпрограммы `MPI_Wait` после вызова `MPI_Ibsend` подразумевает, что буфер передачи можно использовать вновь, то есть пересылаемые данные отправлены или скопированы в буфер, выделенный при вызове подпрограммы `MPI_Buffer_attach`.
- ▶ В этот момент уже нельзя отменить передачу. Если не будет зарегистрирован соответствующий прием, буфер нельзя будет освободить. В этом случае можно применить подпрограмму `MPI_Cancel`, которая освобождает память, выделенную подсистеме коммуникаций.

Проверка завершения всех обменов

- ▶ Проверка завершения всех обменов выполняется подпрограммой:

```
int MPI_Waitall(int count, MPI_Request  
requests[], MPI_Status statuses[])
```

```
MPI_Waitall(count, requests, statuses, ierr)
```

- ▶ При вызове этой подпрограммы выполнение процесса блокируется до тех пор, пока **все** операции обмена, связанные с активными запросами в массиве `requests`, не будут выполнены. Возвращается статус этих операций. Статус обменов содержится в массиве `statuses`. `count` – количество запросов на обмен (размер массивов `requests` и `statuses`).

Проверка завершения всех обменов

- ▶ В результате выполнения подпрограммы `MPI_Waitall` запросы, сформированные неблокирующими операциями обмена, аннулируются, а соответствующим элементам массива присваивается значение `MPI_REQUEST_NULL`.
- ▶ В случае неуспешного выполнения одной или более операций обмена подпрограмма `MPI_Waitall` возвращает код ошибки `MPI_ERR_IN_STATUS` и присваивает полю ошибки статуса значение кода ошибки соответствующей операции.
- ▶ Если операция выполнена успешно, полю присваивается значение `MPI_SUCCESS`, а если не выполнена, но и не было ошибки – значение `MPI_ERR_PENDING`. Это соответствует наличию запросов на выполнение операции обмена, ожидающих обработки.

Алгоритм Якоби. Последовательная версия

```
/* Jacobi program */
#include <stdio.h>
#define L 1000
#define ITMAX 100
int i,j,it;
double A[L][L];
double B[L][L];
int main(int an, char **as)
{
    printf("JAC STARTED\n");
    for(i=0;i<=L-1;i++)
        for(j=0;j<=L-1;j++)
        {
            A[i][j]=0.;
            B[i][j]=1.+i+j;
        }
}
```

Алгоритм Якоби. Последовательная версия

```
/****** iteration loop *****/
for(it=1; it<ITMAX;it++)
{
    for(i=1;i<=L-2;i++)
        for(j=1;j<=L-2;j++)
            A[i][j] = B[i][j];
    for(i=1;i<=L-2;i++)
        for(j=1;j<=L-2;j++)
            B[i][j] = (A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4.;
}
return 0;
}
```


Алгоритм Якоби. MPI-версия

A_{00}	A_{01}	A_{02}	A_{03}	A_{04}	A_{05}	A_{06}	A_{07}	A_{08}
A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}	A_{16}	A_{17}	A_{18}
A_{20}	A_{21}	A_{22}	A_{23}	A_{24}	A_{25}	A_{26}	A_{27}	A_{28}
A_{30}	A_{31}	A_{32}	A_{33}	A_{34}	A_{35}	A_{36}	A_{37}	A_{38}



Shadow edges



Imported elements

A_{20}	A_{21}	A_{22}	A_{23}	A_{24}	A_{25}	A_{26}	A_{27}	A_{28}
A_{30}	A_{31}	A_{32}	A_{33}	A_{34}	A_{35}	A_{36}	A_{37}	A_{38}
A_{40}	A_{41}	A_{42}	A_{43}	A_{44}	A_{45}	A_{46}	A_{47}	A_{48}
A_{50}	A_{51}	A_{52}	A_{53}	A_{54}	A_{55}	A_{56}	A_{57}	A_{58}
A_{60}	A_{61}	A_{62}	A_{63}	A_{64}	A_{65}	A_{66}	A_{67}	A_{68}

A_{50}	A_{51}	A_{52}	A_{53}	A_{54}	A_{55}	A_{56}	A_{57}	A_{58}
A_{60}	A_{61}	A_{62}	A_{63}	A_{64}	A_{65}	A_{66}	A_{67}	A_{68}
A_{70}	A_{71}	A_{72}	A_{73}	A_{74}	A_{75}	A_{76}	A_{77}	A_{78}
A_{80}	A_{81}	A_{82}	A_{83}	A_{84}	A_{85}	A_{86}	A_{87}	A_{88}

Алгоритм Якоби. MPI-версия

```
/* Jacobi-1d program */  
#include <math.h>  
#include <stdlib.h>  
#include <stdio.h>  
#include "mpi.h"  
#define m_printf if (myrank==0)printf  
#define L 1000  
#define ITMAX 100  
  
int i,j,it,k;  
int ll,shift;  
double (* A)[L];  
double (* B)[L];
```

Алгоритм Якоби. MPI-версия

```
int main(int argc, char **argv)
{
    MPI_Request req[4];
    int myrank, ranksize;
    int startrow, lastrow, nrow;
    MPI_Status status[4];
    double t1, t2, time;
    MPI_Init (&argc, &argv); /* initialize MPI system */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* my place in MPI system */
    MPI_Comm_size (MPI_COMM_WORLD, &ranksize); /* size of MPI system */
    MPI_Barrier(MPI_COMM_WORLD);
    /* rows of matrix I have to process */
    startrow = (myrank * L) / ranksize;
    lastrow = (((myrank + 1) * L) / ranksize) - 1;
    nrow = lastrow - startrow + 1;
    m_printf("JAC1 STARTED\n");
```

Алгоритм Якоби. MPI-версия

```
/* dynamically allocate data structures */  
A = malloc ((nrow+2) * L * sizeof(double));  
B = malloc ((nrow) * L * sizeof(double));  
for(i=1; i<=nrow; i++)  
    for(j=0; j<=L-1; j++)  
    {  
        A[i][j]=0.;  
        B[i-1][j]=1.+startrow+i-1+j;  
    }
```

Алгоритм Якоби. MPI-версия

```
/****** iteration loop *****/
t1=MPI_Wtime();
for(it=1; it<=ITMAX; it++)
{
    for(i=1; i<=nrow; i++)
    {
        if (((i==1)&&(myrank==0))||((i==nrow)&&(myrank==ranksize-1)))
            continue;
        for(j=1; j<=L-2; j++)
        {
            A[i][j] = B[i-1][j];
        }
    }
}
```

Алгоритм Якоби. MPI-версия

```
if(myrank!=0)
    MPI_Irecv(&A[0][0],L,MPI_DOUBLE, myrank-1, 1215,
             MPI_COMM_WORLD, &req[0]);
if(myrank!=ranksize-1)
    MPI_Isend(&A[nrow][0],L,MPI_DOUBLE, myrank+1, 1215,
             MPI_COMM_WORLD,&req[2]);
if(myrank!=ranksize-1)
    MPI_Irecv(&A[nrow+1][0],L,MPI_DOUBLE, myrank+1, 1216,
             MPI_COMM_WORLD, &req[3]);
if(myrank!=0)
    MPI_Isend(&A[1][0],L,MPI_DOUBLE, myrank-1, 1216,
             MPI_COMM_WORLD,&req[1]);
ll=4; shift=0;
if (myrank==0) {ll=2;shift=2;}
if (myrank==ranksize-1) {ll=2;}
MPI_Waitall(ll,&req[shift],&status[0]);
```

Алгоритм Якоби. MPI-версия

```
if(myrank!=0)
    MPI_Recv(&A[0][0],L,MPI_DOUBLE, myrank-1, 1215,
            MPI_COMM_WORLD, &status[0]);
if(myrank!=ranksize-1)
    MPI_Send(&A[nrow][0],L,MPI_DOUBLE, myrank+1, 1215,
            MPI_COMM_WORLD);
if(myrank!=ranksize-1)
    MPI_Recv(&A[nrow+1][0],L,MPI_DOUBLE, myrank+1, 1216,
            MPI_COMM_WORLD, &status[0]);
if(myrank!=0)
    MPI_Send(&A[1][0],L,MPI_DOUBLE, myrank-1, 1216,
            MPI_COMM_WORLD);
```

Алгоритм Якоби. MPI-версия

```
for(i=1; i<=nrow; i++)
{
    if (((i==1)&&(myrank==0))||((i==nrow)&&(myrank==ranksize-1))) continue;
    for(j=1; j<=L-2; j++)
        B[i-1][j] = (A[i-1][j]+A[i+1][j]+
                    A[i][j-1]+A[i][j+1])/4.;
}
/*DO it*/
printf("%d: Time of task=%lf\n",myrank,MPI_Wtime()-t1);
MPI_Finalize ();
return 0;
}
```


Проверка завершения любого числа обменов

- ▶ Проверка завершения любого числа обменов выполняется подпрограммой:

```
int MPI_Waitany(int count, MPI_Request requests[],  
int *index, MPI_Status *status)
```

- ▶ Выполнение процесса блокируется до тех пор, пока, по крайней мере, один обмен из массива запросов (`requests`) не будет завершен.
- ▶ Входные параметры:
 - ❑ `requests` – запрос;
 - ❑ `count` – количество элементов в массиве `requests`.
- ▶ Выходные параметры:
 - ❑ `index` – индекс запроса (в языке C это целое число от 0 до `count - 1`) в массиве `requests`;
 - ❑ `status` – статус.

Неблокирующие процедуры проверки

- ▶ Подпрограмма `MPI_Test` выполняет неблокирующую проверку завершения приема или передачи сообщения:

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

- ▶ Входной параметр: идентификатор операции обмена `request`.
- ▶ Выходные параметры:
 - ❑ `flag` — «истина», если операция, заданная идентификатором `request`, выполнена;
 - ❑ `status` — статус выполненной операции.

Неблокирующая проверка завершения всех обменов

- ▶ Подпрограмма `MPI_Testall` выполняет неблокирующую проверку завершения приема или передачи всех сообщений:

```
int MPI_Testall(int count, MPI_Request requests[],  
int *flag, MPI_Status statuses[])  
MPI_Testall(count, requests, flag, statuses, ierr)
```

- ▶ При вызове возвращается значение флага (`flag`) «истина», если все обмены, связанные с активными запросами в массиве `requests`, выполнены. Если завершены не все обмены, флагу присваивается значение «ложь», а массив `statuses` не определен.
- ▶ Параметр `count` – количество запросов.
- ▶ Каждому статусу, соответствующему активному запросу, присваивается значение статуса соответствующего обмена.

Неблокирующая проверка любого числа обменов

- ▶ Подпрограмма `MPI_Testany` выполняет неблокирующую проверку завершения приема или передачи сообщения:

```
int MPI_Testany(int count, MPI_Request  
requests[], int *index, int *flag, MPI_Status  
*status)
```

- ▶ Смысл и назначение параметров этой подпрограммы те же, что и для подпрограммы `MPI_Waitany`.
Дополнительный аргумент `flag`, принимает значение «истина», если одна из операций завершена.
- ▶ Блокирующая подпрограмма `MPI_Waitany` и неблокирующая `MPI_Testany` взаимозаменяемы, как и другие аналогичные пары.

Другие операции проверки

- ▶ Подпрограммы `MPI_Waitsome` и `MPI_Testsome` действуют аналогично подпрограммам `MPI_Waitany` и `MPI_Testany`, кроме случая, когда завершается более одного обмена.
- ▶ В подпрограммах `MPI_Waitany` и `MPI_Testany` обмен из числа завершенных выбирается произвольно, именно для него и возвращается статус, а для `MPI_Waitsome` и `MPI_Testsome` статус возвращается для всех завершенных обменов.
- ▶ Эти подпрограммы можно использовать для определения, сколько обменов завершено.

Другие операции проверки

- ▶ Блокирующая проверка выполнения обменов:

```
int MPI_Waitsome(int incount, MPI_Request
requests[], int *outcount, int indices[],
MPI_Status statuses[])
```

- ▶ Здесь `incount` – количество запросов. В `outcount` возвращается количество выполненных запросов из массива `requests`, а в первых `outcount` элементах массива `indices` возвращаются индексы этих операций. В первых `outcount` элементах массива `statuses` возвращается статус завершенных операций. Если выполненный запрос был сформирован неблокирующей операцией обмена, он аннулируется. Если в списке нет активных запросов, выполнение подпрограммы завершается сразу, а параметру `outcount` присваивается значение `MPI_UNDEFINED`.

Другие операции проверки

- ▶ Неблокирующая проверка выполнения обменов:

```
int MPI_Testsome(int incount, MPI_Request
requests[], int *outcount, int indices[],
MPI_Status statuses[])
```

- ▶ Параметры такие же, как и у подпрограммы `MPI_Waitsome`. Эффективность подпрограммы `MPI_Testsome` выше, чем у `MPI_Testany`, поскольку первая возвращает информацию обо всех операциях, а для второй требуется новый вызов для каждой выполненной операции.

Проверка статуса операции приема сообщения

- ▶ Блокирующая проверка:

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status* status)
```

- ▶ Неблокирующая проверка:

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
```

- ▶ Входные параметры этой подпрограммы те же, что и у подпрограммы `MPI_Probe`.
- ▶ Выходные параметры:
 - `flag` – флаг;
 - `status` – статус.
- ▶ Если сообщение уже поступило и может быть принято, возвращается значение флага «истина».

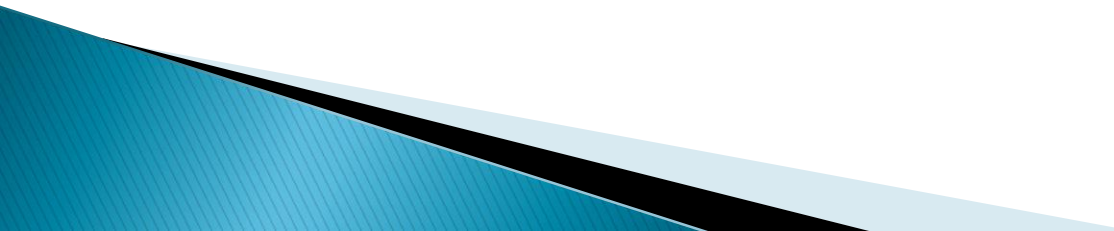
Проверка статуса операции приема сообщения (пример)

```
if (rank == 0) {
    MPI_Send(buf, size, MPI_INT, 1, 0, MPI_COMM_WORLD); // Send to process 1
    printf("0 sent %d numbers to 1 \n", size);
} else if (rank == 1) {
    MPI_Status status;
    MPI_Probe(0, 0, MPI_COMM_WORLD, &status); // Probe for an incoming msg.
    // When probe returns, the status object has the size and other
    // attributes of the incoming message. Get the size of the message.
    MPI_Get_count(&status, MPI_INT, &size);
    // Allocate a buffer just big enough to hold the incoming numbers
    int* bufnumber = (int*)malloc(sizeof(int) * size);
    // Now receive the message with the allocated buffer
    MPI_Recv(bufnumber, size, MPI_INT, 0, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("1 dynamically received %d numbers from 0. \n", size);
    free(bufnumber);
}
```

Коллективные операции

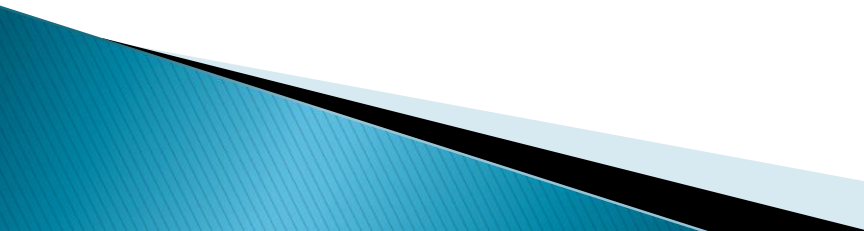
- ▶ Передача сообщений между группой процессов
- ▶ Вызываются ВСЕМИ процессами в коммутаторе
- ▶ Примеры:
 - Broadcast, scatter, gather (рассылка данных)
 - Global sum, global maximum, и.т.д. (редукционные операции)
 - Барьерная синхронизация

Характеристики коллективных передач

- ▶ Коллективные операции не являются помехой операциям типа точка–точка и наоборот
 - ▶ Все процессы коммутатора должны вызывать коллективную операцию
 - ▶ Синхронизация не гарантируется (за исключением барьера)
 - ▶ Нет тэгов
 - ▶ Принимающий буфер должен точно соответствовать размеру отсылаемого буфера
- 

Особенности коллективных передач

```
switch(rank) {  
  case 0:  
    MPI_Bcast(buf1, count, type, 0, comm);  
    MPI_Bcast(buf2, count, type, 1, comm);  
    break;  
  case 1:  
    MPI_Bcast(buf2, count, type, 1, comm);  
    MPI_Bcast(buf1, count, type, 0, comm);  
    break;  
}
```



Особенности коллективных передач

```
switch(rank) {
```

```
  case 0:
```

```
    MPI_Bcast(buf1, count, type, 0, comm0);
```

```
    MPI_Bcast(buf2, count, type, 2, comm2);
```

```
    break;
```

```
  case 1:
```

```
    MPI_Bcast(buf1, count, type, 1, comm1);
```

```
    MPI_Bcast(buf2, count, type, 0, comm0);
```

```
    break;
```

```
  case 2:
```

```
    MPI_Bcast(buf1, count, type, 2, comm2);
```

```
    MPI_Bcast(buf2, count, type, 1, comm1);
```

```
    break;
```

```
}
```

```
/* comm0 is {0,1}, comm1 is {1, 2} comm2 is {2,0} */
```

Особенности коллективных передач

```
switch(rank) {
```

```
  case 0:
```

```
    MPI_Bcast(buf1, count, type, 0, comm);
```

```
    MPI_Send(buf2, count, type, 1, tag, comm);
```

```
    break;
```

```
  case 1:
```

```
    MPI_Recv(buf2, count, type, 0, tag, comm, status);
```

```
    MPI_Bcast(buf1, count, type, 0, comm);
```

```
    break;
```

```
}
```



Обзор коллективных операций

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

Обзор коллективных операций

- ▶ Глобальные вычислительные операции (sum, min, max и др.) над данными, расположенными в адресных пространствах различных процессов:
 - с сохранением результата в адресном пространстве одного процесса (MPI_Reduce);
 - с рассылкой результата всем процессам (MPI_Allreduce);
 - совмещенная операция Reduce/Scatter (MPI_Reduce_scatter);
 - префиксная редукция (MPI_Scan).
- ▶ Все коммуникационные подпрограммы, за исключением MPI_Bcast, представлены в двух вариантах:
 - простой вариант, когда все части передаваемого сообщения имеют одинаковую длину;
 - "векторный" вариант, который снимает ограничения, присущие простому варианту, как в части длин блоков, так и в части размещения данных в адресном пространстве процессов. Векторные варианты отличаются дополнительным символом "v" в конце имени функции.

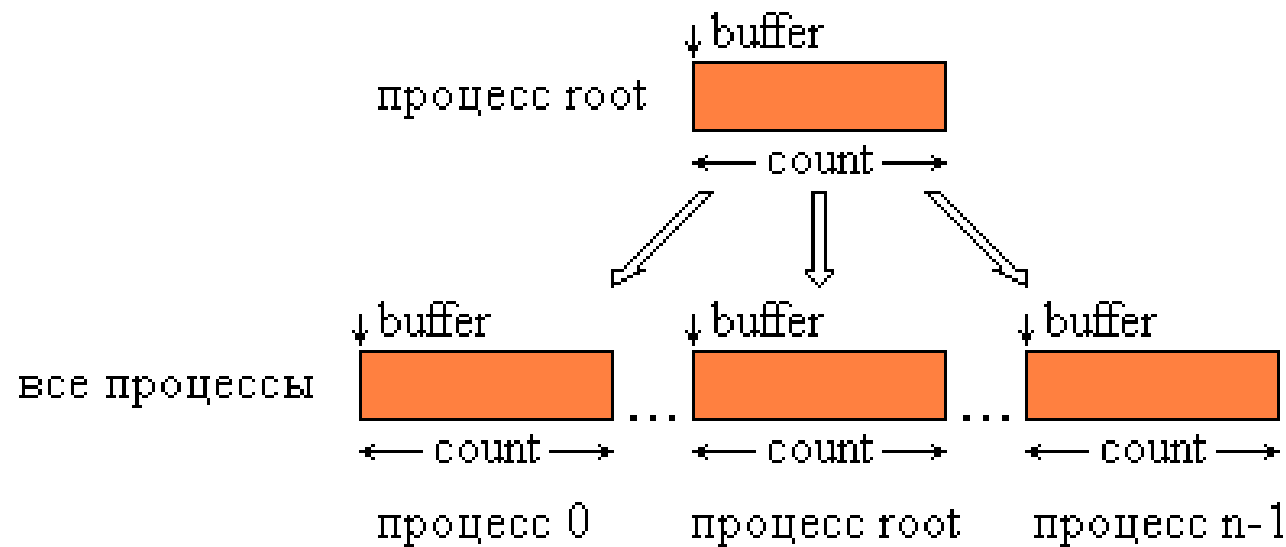
Широковещательная рассылка

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,  
MPI_Comm comm )
```

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

Процесс с номером root рассылает сообщение из своего буфера передачи всем процессам области связи коммуникатора comm.

Широковещательная рассылка



Сбор блоков данных от всех процессов группы

Сборка блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером `root`:

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

IN `sendbuf` - адрес начала размещения посылаемых данных;

IN `sendcount` - число посылаемых элементов;

IN `sendtype` - тип посылаемых элементов;

OUT `recvbuf` - адрес начала буфера приема (используется только в процессе-получателе `root`);

IN `recvcount` - число элементов, получаемых от каждого процесса (используется только в процессе-получателе `root`);

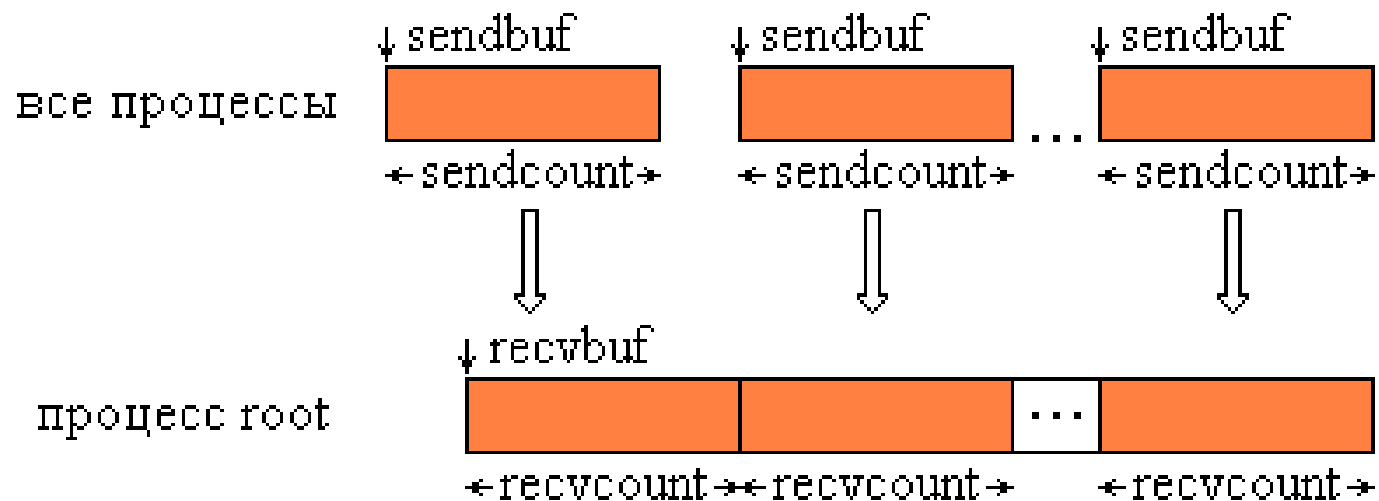
IN `recvttype` - тип получаемых элементов;

IN `root` - номер процесса-получателя; IN `comm` - коммуникатор.

Сбор блоков данных от всех процессов группы

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

Сбор блоков данных от всех процессов группы



Сбор блоков данных от всех процессов группы

Функция **MPI_Gatherv** позволяет собирать блоки с разным числом элементов:

```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,  
void* rbuf, int *recvcounts, int *displs, MPI_Datatype recvtype,  
int root, MPI_Comm comm)
```

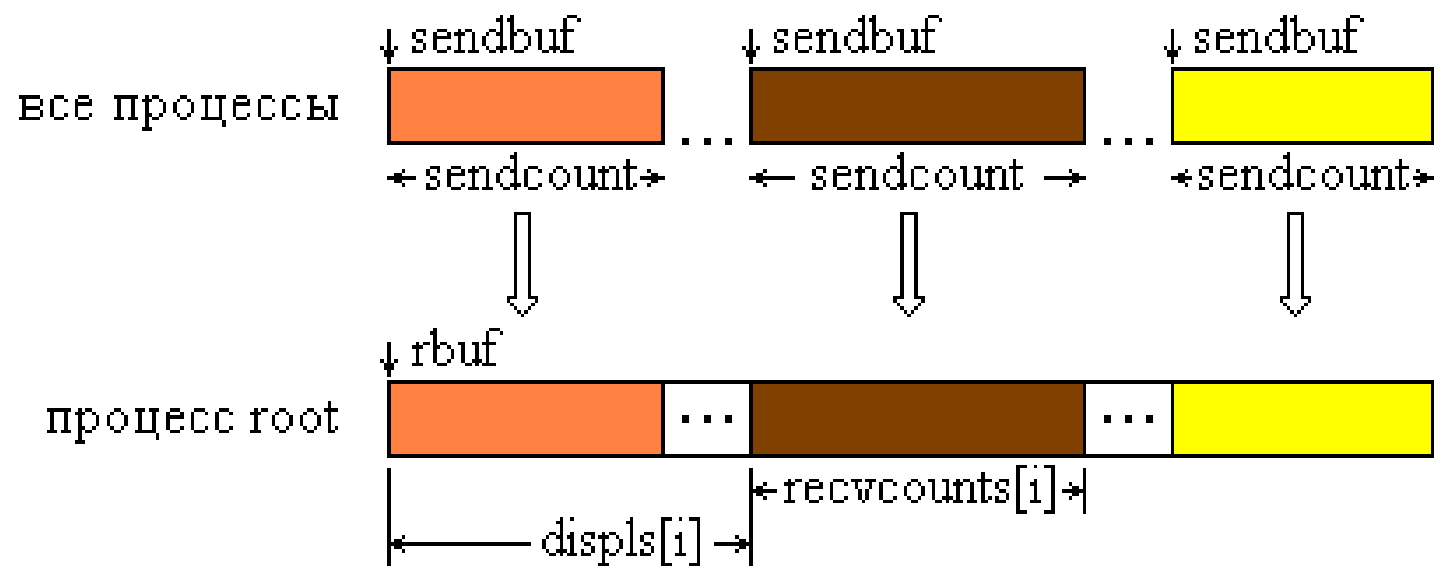
- IN sendbuf - адрес начала буфера передачи;
- IN sendcount - число посылаемых элементов;
- IN sendtype - тип посылаемых элементов;
- OUT rbuf - адрес начала буфера приема;
- IN recvcounts - целочисленный массив (размер равен числу процессов в группе), *i*-й элемент которого определяет число элементов, которое должно быть получено от процесса *i*;
- IN displs - целочисленный массив (размер равен числу процессов в группе), *i*-ое значение определяет смещение *i*-го блока данных относительно начала rbuf;
- IN recvtype - тип получаемых элементов;
- IN root - номер процесса-получателя;
- IN comm - коммуникатор.

Сбор блоков данных от всех процессов группы

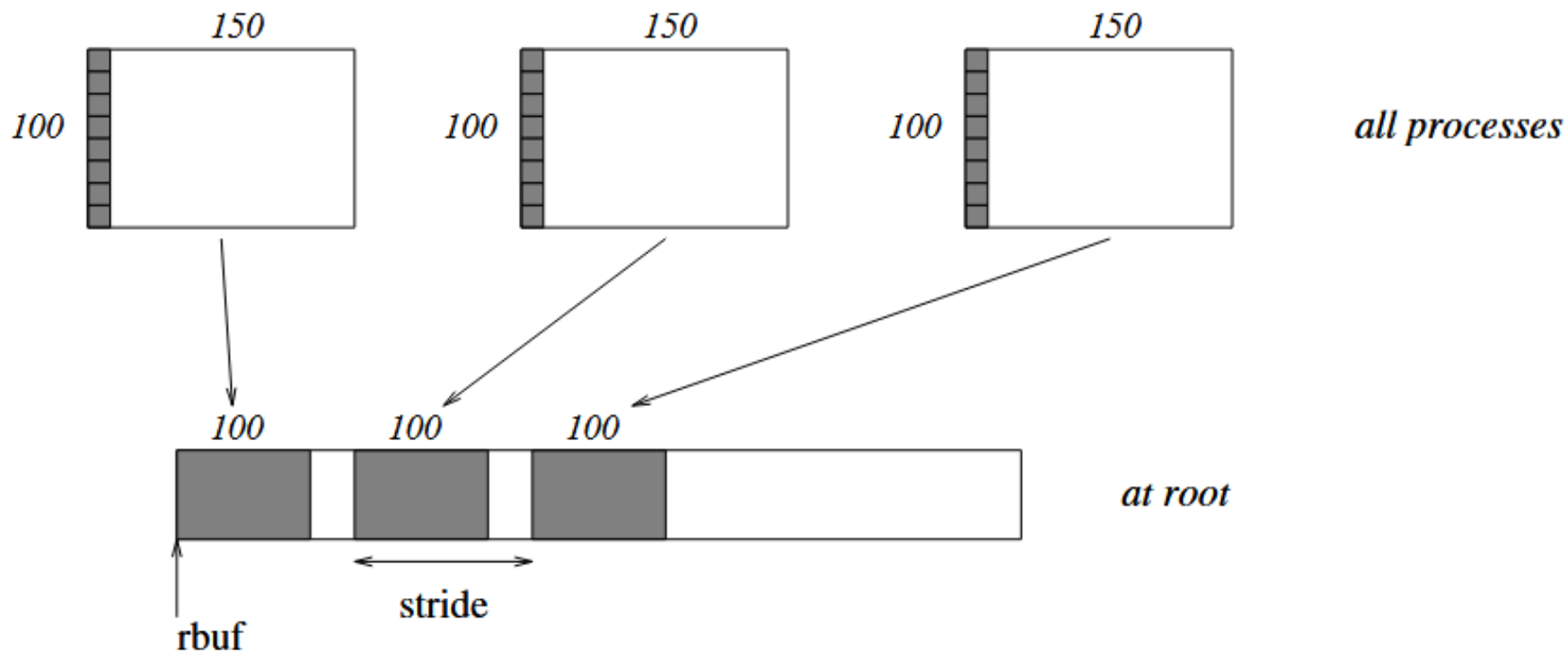
Функция **MPI_Gatherv** позволяет собирать блоки с разным числом элементов от каждого процесса, так как количество элементов, принимаемых от каждого процесса, задается индивидуально с помощью массива `recvcounts`. Эта функция обеспечивает также большую гибкость при размещении данных в процессе-получателе, благодаря введению в качестве параметра массива смещений `displs`.

Сообщения помещаются в буфер приема процесса `root` в соответствии с номерами посылающих процессов, а именно, данные, посланные процессом `i`, размещаются в адресном пространстве процесса `root`, начиная с адреса `rbuf + displs[i]`

Сбор блоков данных от всех процессов группы



Сбор блоков данных от всех процессов группы



Сбор блоков данных от всех процессов группы

```
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
/* Create datatype for 1 column of array */
MPI_Type_vector(100, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
```

```
MPI_Gatherv(sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT, root, comm);
```

Сбор блоков данных от всех процессов группы

Функция **MPI_Allgather** выполняется так же, как **MPI_Gather**, но получателями являются все процессы группы. Данные, посланные процессом i из своего буфера `sendbuf`, помещаются в i -ю порцию буфера `recvbuf` каждого процесса. После завершения операции содержимое буферов приема `recvbuf` у всех процессов одинаково.

C:

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

IN	<code>sendbuf</code>	-	адрес начала буфера посылки;
IN	<code>sendcount</code>	-	число посылаемых элементов;
IN	<code>sendtype</code>	-	тип посылаемых элементов;
OUT	<code>recvbuf</code>	-	адрес начала буфера приема;
IN	<code>recvcount</code>	-	число элементов, получаемых от каждого процесса;
IN	<code>recvtype</code>	-	тип получаемых элементов;
IN	<code>comm</code>	-	коммуникатор.

Сбор блоков данных от всех процессов группы

Функция **MPI_Allgatherv** является аналогом функции **MPI_Gatherv**, но сборка выполняется всеми процессами группы. Поэтому в списке параметров отсутствует параметр `root`.

```
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,  
void* rbuf, int *recvcounts, int *displs,  
MPI_Datatype recvtype, MPI_Comm comm)
```

- | | | |
|---------------|---|---|
| IN sendbuf | - | адрес начала буфера передачи; |
| IN sendcount | - | число посылаемых элементов; |
| IN sendtype | - | тип посылаемых элементов; |
| OUT rbuf | - | адрес начала буфера приема; |
| IN recvcounts | - | целочисленный массив (размер равен числу процессов в группе), содержащий число элементов, которое должно быть получено от каждого процесса; |
| IN displs | - | целочисленный массив (размер равен числу процессов в группе), <i>i</i> -ое значение определяет смещение относительно начала rbuf <i>i</i> -го блока данных; |
| IN recvtype | - | тип получаемых элементов; IN comm - коммуникатор. |

Рассылка блоков данных по всем процессам группы

Функция **MPI_Scatter** разбивает сообщение из буфера посылки процесса root на равные части размером sendcount и посылает i-ю часть в буфер приема процесса с номером i (в том числе и самому себе).

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

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

Рассылка блоков данных по всем процессам группы

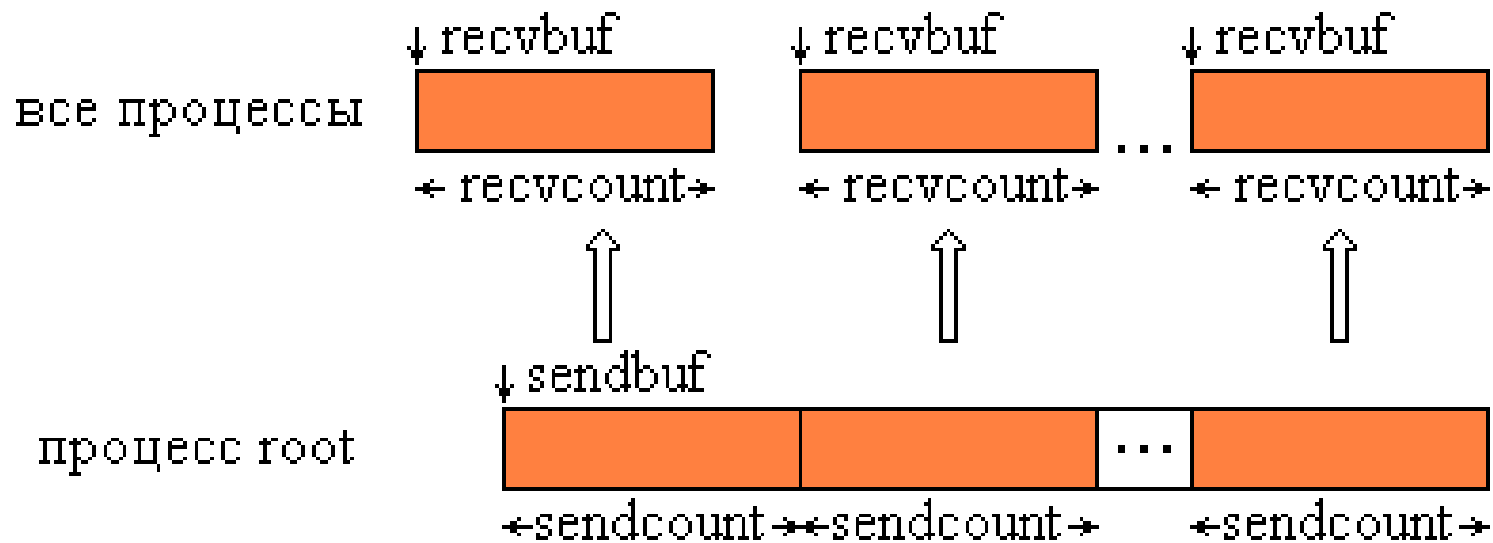
Функция **MPI_Scatter** разбивает сообщение из буфера отправки процесса root на равные части размером sendcount и посылает i-ю часть в буфер приема процесса с номером i (в том числе и самому себе).

Процесс root использует оба буфера (отправки и приема), поэтому в вызываемой им подпрограмме все параметры являются существенными. Остальные процессы группы с коммутатором comm являются только получателями, поэтому для них параметры, специфицирующие буфер отправки, не существенны.

Число посылаемых элементов sendcount должно равняться числу принимаемых recvcount.

Следует обратить внимание, что значение sendcount в вызове из процесса root - это число посылаемых каждому процессу элементов, а не общее их количество. Операция Scatter является обратной по отношению к Gather.

Рассылка блоков данных по всем процессам группы



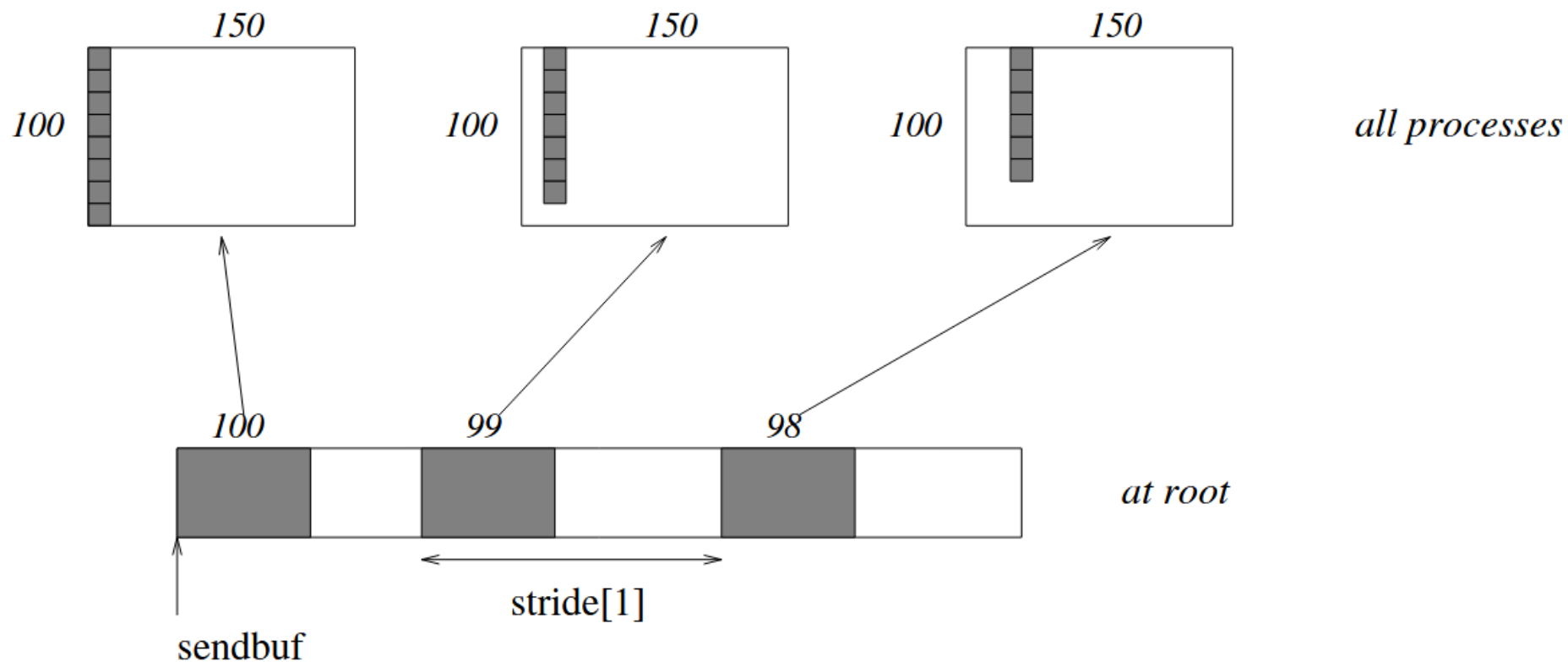
Рассылка блоков данных по всем процессам группы

Функция **MPI_Scatterv** является векторным вариантом функции **MPI_Scatter**, позволяющим посылать каждому процессу различное количество элементов. Начало расположения элементов блока, посылаемого *i*-му процессу, задается в массиве смещений `displs`, а число посылаемых элементов - в массиве `sendcounts`.

```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,  
                MPI_Datatype sendtype, void* recvbuf, int recvcount,  
                MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- IN `sendbuf` - адрес начала буфера отправки (используется только в процессе-отправителе `root`);
- IN `sendcounts` - целочисленный массив (размер равен числу процессов в группе), содержащий число элементов, посылаемых каждому процессу;
- IN `displs` - целочисленный массив (размер равен числу процессов в группе), *i*-ое значение определяет смещение относительно начала `sendbuf` для данных, посылаемых процессу *i*;
- IN `sendtype` - тип посылаемых элементов;
- OUT `recvbuf` - адрес начала буфера приема;
- IN `recvcount` - число получаемых элементов;
- IN `recvtype` - тип получаемых элементов;
- IN `root` - номер процесса-отправителя; IN `comm` - коммуникатор.

Рассылка блоков данных по всем процессам группы



Рассылка блоков данных по всем процессам группы

```
stride = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] for i = 0 to gsize-1 is set somehow sendbuf comes from elsewhere */
...
displs = (int *)malloc(gsize*sizeof(int));
counts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {
    displs[i] = offset;
    offset += stride[i];
    counts[i] = 100 - i;
}
/* Create datatype for the column we are receiving */
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &rtype);
MPI_Type_commit(&rtype);
rptr = &recvarray[0][myrank];
MPI_Scatterv(sendbuf, counts, displs, MPI_INT, rptr, 1, rtype, root, comm);
```

Рассылка данных от всех процессов всем процессам

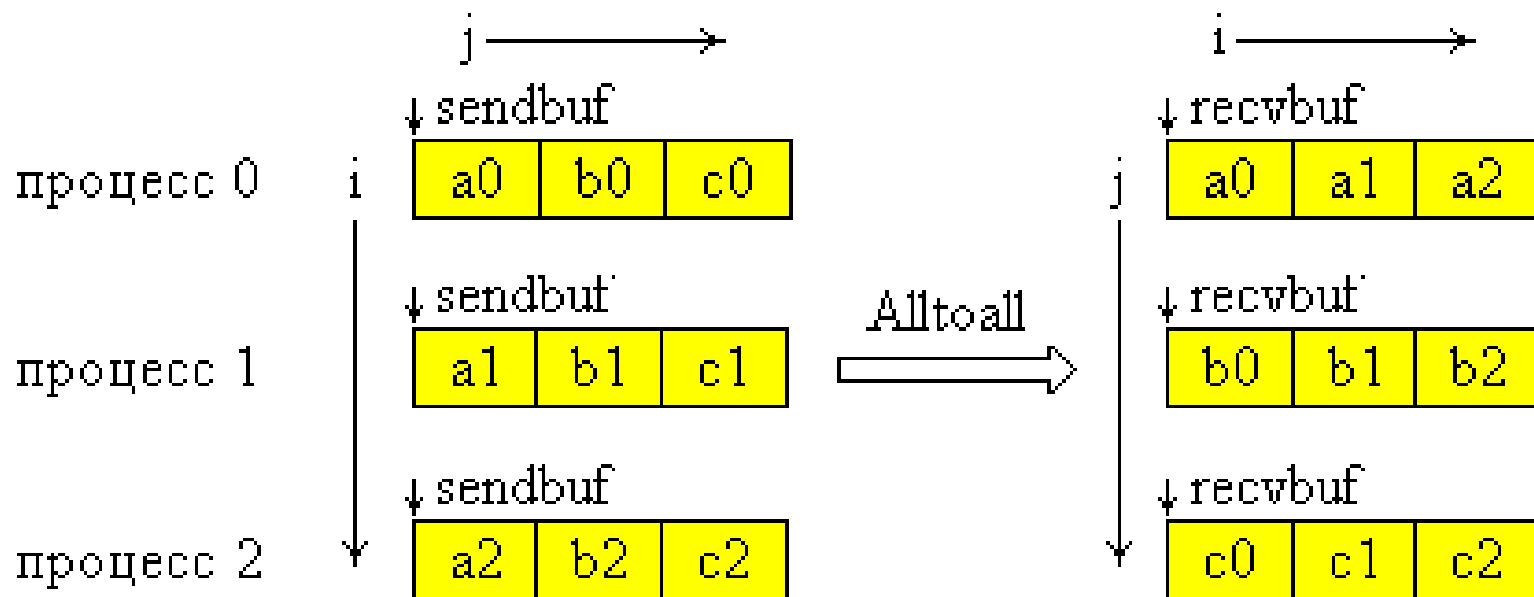
Функция **MPI_Alltoall** совмещает в себе операции Scatter и Gather и является по сути дела расширением операции Allgather, когда каждый процесс посылает различные данные разным получателям. Процесс i посылает j -ый блок своего буфера `sendbuf` процессу j , который помещает его в i -ый блок своего буфера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

int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvttype, MPI_Comm comm)

- IN `sendbuf` - адрес начала буфера отправки;
- IN `sendcount` - число посылаемых элементов;
- IN `sendtype` - тип посылаемых элементов;
- OUT `recvbuf` - адрес начала буфера приема;
- IN `recvcount` - число элементов, получаемых от каждого процесса;
- IN `recvttype` - тип получаемых элементов;
- IN `comm` - коммуникатор.

MPI_Alltoallv реализует векторный вариант операции Alltoall, допускающий передачу и прием блоков различной длины с более гибким размещением передаваемых и принимаемых данных.

Рассылка данных от всех процессов всем процессам



Глобальные вычислительные операции над распределенными данными

Операцией редукции называется операция, аргументом которой является вектор, а результатом - скалярная величина, полученная применением некоторой математической операции ко всем компонентам вектора распределенными по процессам коммутатора.

Операции редукции в MPI представлены в нескольких вариантах:

- с сохранением результата в адресном пространстве одного процесса (MPI_Reduce).
- с сохранением результата в адресном пространстве всех процессов (MPI_Allreduce).
- префиксная операция редукции, которая в качестве результата операции возвращает вектор. i -я компонента этого вектора является результатом редукции первых i компонент распределенного вектора (MPI_Scan).
- совмещенная операция Reduce/Scatter (MPI_Reduce_scatter).

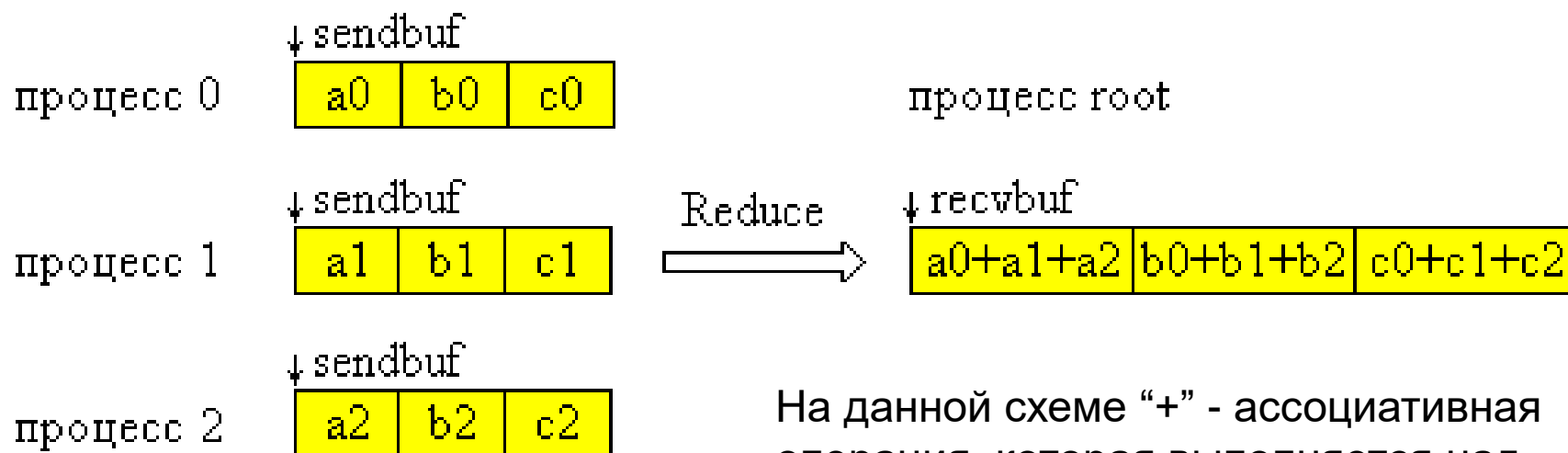
Глобальные вычислительные операции над распределенными данными

Функция **MPI_Reduce** выполняется следующим образом. Операция глобальной редукции, указанная параметром `op`, выполняется над первыми элементами входного буфера, и результат посылается в первый элемент буфера приема процесса `root`. Затем то же самое делается для вторых элементов буфера и т.д.

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

- IN `sendbuf` - адрес начала входного буфера;
- OUT `recvbuf` - адрес начала буфера результатов (используется только в процессе-получателе `root`);
- IN `count` - число элементов во входном буфере;
- IN `datatype` - тип элементов во входном буфере;
- IN `op` - операция, по которой выполняется редукция;
- IN `root` - номер процесса-получателя результата операции;
- IN `comm` - коммуникатор.

Глобальные вычислительные операции над распределенными данными



На данной схеме “+” - ассоциативная операция, которая выполняется над парой операндов типа datatype и возвращает результат того же типа:
MPI_MAX, MPI_MIN, MPI_SUM,
MPI_PROD, MPI_LAND, MPI_BAND,
MPI_LOR, MPI_BOR, MPI_LXOR,
MPI_BXOR, MPI_MAXLOC,
MPI_MINLOC

Глобальные вычислительные операции над распределенными данными

```
MPI_Comm_size ( comm , & groupsize );
MPI_Comm_rank ( comm , & rank );
if ( rank > 0 ) {
    MPI_Recv ( tempbuf , count , datatype , rank -1 , ...)
    User_reduce ( tempbuf , sendbuf , count , datatype );
}
if ( rank < groupsize -1 ) {
    MPI_Send ( sendbuf , count , datatype , rank +1 , ...);
}
/* answer now resides in MPI process groupsize -1 ...now send to root*/
if ( rank == root ) {
    MPI_Irecv ( recvbuf , count , datatype , groupsize -1 , ... , & req );
}
if ( rank == groupsize -1 ) {
    MPI_Send ( sendbuf , count , datatype , root , ...);
}
if ( rank == root ) {
    MPI_Wait ( & req , & status );
}
```

Cray MPI: параметры по умолчанию

MPI Environment Variable Name	1,000 PEs	10,000 PEs	50,000 PEs	100,000 Pes
MPICH_MAX_SHORT_MSG_SIZE (This size determines whether the message uses the Eager or Randervous protocol)	128,000 Bytes	20,480	4096	2048
MPICH_UNEX_BUFFER_SIZE (The buffer allocated to hold the unexpected Eager data)	60 MB	60 MB	150 MB	260 MB
MPICH_PTL_UNEX_EVENTS (Portals generates two events for each unexpected message received)	20,480 events	22,000	110,000	220,000
MPICH_PTL_OTHER_EVENTS (Portals send-side and expected events)	2048 events	2500	12,500	25,000

Глобальные вычислительные операции над распределенными данными

```
/* each MPI process has an array of 30 double : ain [30]*/
double ain [30] , aout [30];
int ind [30];
struct {
    double val ;
    int rank ;
} in [30] , out [30];
int i , myrank , root ;
MPI_Comm_rank ( comm , &myrank );
for ( i =0; i <30; ++ i ) {
    in [i ]. val = ain [i ];
    in [i ]. rank = myrank ;
}
MPI_Reduce (in , out , 30 , MPI_DOUBLE_INT , MPI_MAXLOC , root , comm );
/* At this point , the answer resides on root MPI process*/
if ( myrank == root ) { /* read ranks out*/
    for ( i =0; i <30; ++ i ) {
        aout [i] = out [i ]. val ;
        ind [i] = out [i ]. rank ;
    }
}
```

Глобальные вычислительные операции над распределенными данными

MPI_FLOAT_INT float and int

MPI_DOUBLE_INT double and int

MPI_LONG_INT long and int

MPI_2INT pair of int

MPI_SHORT_INT short and int

MPI_LONG_DOUBLE_INT long double and int

```
struct mystruct {  
    double val ;  
    uint64_t index ;  
};  
MPI_Datatype dtype ;  
MPI_Type_get_value_index ( MPI_DOUBLE , MPI_UINT64_T , & dtype );  
if ( dtype == MPI_DATATYPE_NULL ) {  
    // Handling for unsupported value - index type  
}
```

Глобальные вычислительные операции над распределенными данными

```
struct mystruct {  
    short val ;  
    int rank ;  
};  
type [0] = MPI_SHORT ;  
type [1] = MPI_INT ;  
disp [0] = 0;  
disp [1] = offsetof ( struct mystruct , rank );  
block [0] = 1;  
block [1] = 1;
```

```
MPI_Type_create_struct (2 , block , disp , type , &MPI_SHORT_INT );  
MPI_Type_commit ( &MPI_SHORT_INT );
```

Глобальные вычислительные операции над распределенными данными

```
typedef struct {  
    double real , imag ;  
} Complex ;  
  
/* the user - defined function*/  
void myProd ( void * inP , void * inoutP , int * len , MPI_Datatype * dptr )  
{  
    int i;  
    Complex c;  
    Complex * in = ( Complex *) inP , * inout = ( Complex *) inoutP ;  
    for (i =0; i < * len ; ++ i) {  
        c. real = inout -> real *in -> real - inout -> imag *in -> imag ;  
        c. imag = inout -> real *in -> imag + inout -> imag *in -> real ;  
        * inout = c;  
        in ++;  
        inout ++;  
    }  
}
```

Глобальные вычислительные операции над распределенными данными

```
...  
Complex a [100] , answer [100];  
/* each process has an array of 100 Complexes*/
```

```
MPI_Op myOp ;  
MPI_Datatype ctype ;  
/* explain to MPI how type Complex is defined*/
```

```
MPI_Type_contiguous (2 , MPI_DOUBLE , & ctype );  
MPI_Type_commit (& ctype );
```

```
/* create the complex - product user - op*/  
MPI_Op_create ( myProd , 1, & myOp );  
MPI_Reduce (a , answer , 100 , ctype , myOp , root , comm );
```

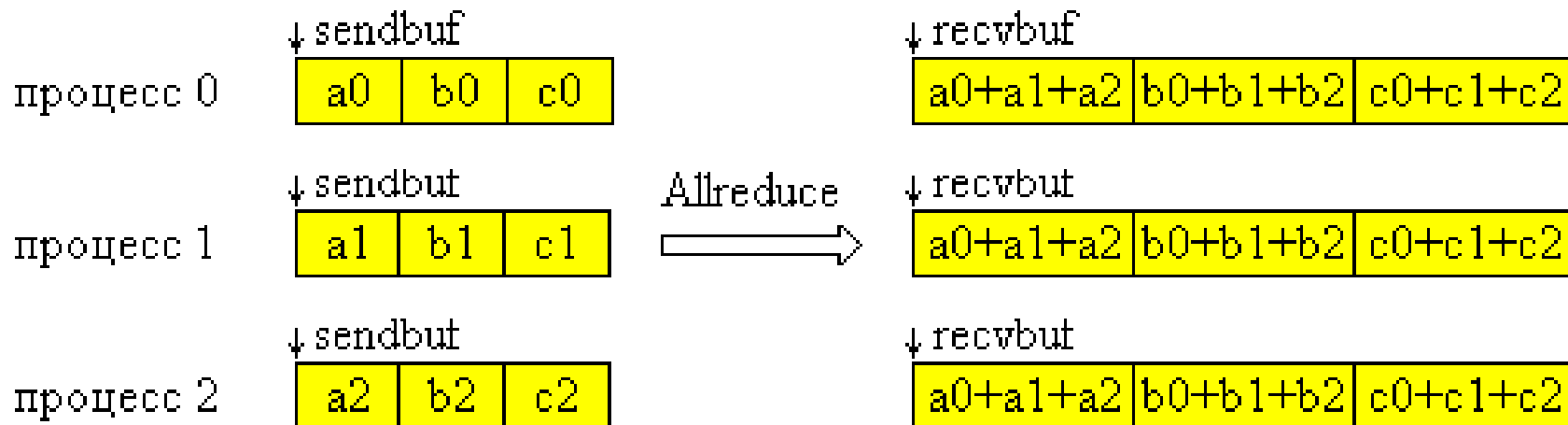
Глобальные вычислительные операции над распределенными данными

Функция **MPI_Allreduce** сохраняет результат редукции в адресном пространстве всех процессов, поэтому в списке параметров функции отсутствует идентификатор корневого процесса `root`. В остальном, набор параметров такой же, как и в предыдущей функции.

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,  
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

- IN `sendbuf` - адрес начала входного буфера;
- OUT `recvbuf` - адрес начала буфера приема;
- IN `count` - число элементов во входном буфере;
- IN `datatype` - тип элементов во входном буфере;
- IN `op` - операция, по которой выполняется редукция;
- IN `comm` - коммуникатор.

Глобальные вычислительные операции над распределенными данными



На данной схеме “+” - ассоциативная операция, которая выполняется над парой операндов типа datatype и возвращает результат того же типа:
MPI_MAX, MPI_MIN, MPI_SUM,
MPI_PROD, MPI_LAND, MPI_BAND,
MPI_LOR, MPI_BOR, MPI_LXOR,
MPI_BXOR, MPI_MAXLOC,
MPI_MINLOC

Глобальные вычислительные операции над распределенными данными

Функция **MPI_Reduce_scatter** совмещает в себе операции редукции и распределения результата по процессам.

MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

IN sendbuf - адрес начала входного буфера;

OUT recvbuf - адрес начала буфера приема;

IN recvcount - массив, в котором задаются размеры блоков, посылаемых процессам;

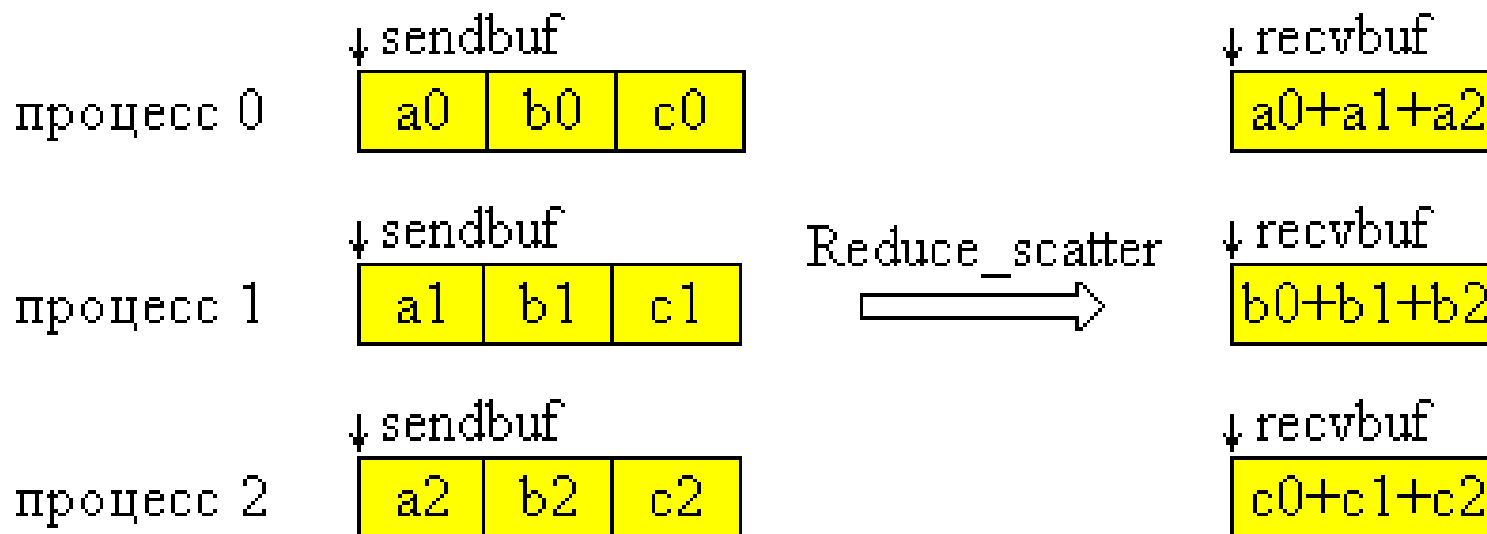
IN datatype - тип элементов во входном буфере;

IN op - операция, по которой выполняется редукция;

IN comm - коммунникатор.

Функция **MPI_Reduce_scatter** отличается от **MPI_Allreduce** тем, что результат операции разрезается на непересекающиеся части по числу процессов в группе, *i*-ая часть посылается *i*-ому процессу в его буфер приема. Длины этих частей задает третий параметр, являющийся массивом.

Глобальные вычислительные операции над распределенными данными



На данной схеме “+” - ассоциативная операция, которая выполняется над парой операндов типа datatype и возвращает результат того же типа:
MPI_MAX, MPI_MIN, MPI_SUM,
MPI_PROD, MPI_LAND, MPI_BAND,
MPI_LOR, MPI_BOR, MPI_LXOR,
MPI_BXOR, MPI_MAXLOC,
MPI_MINLOC

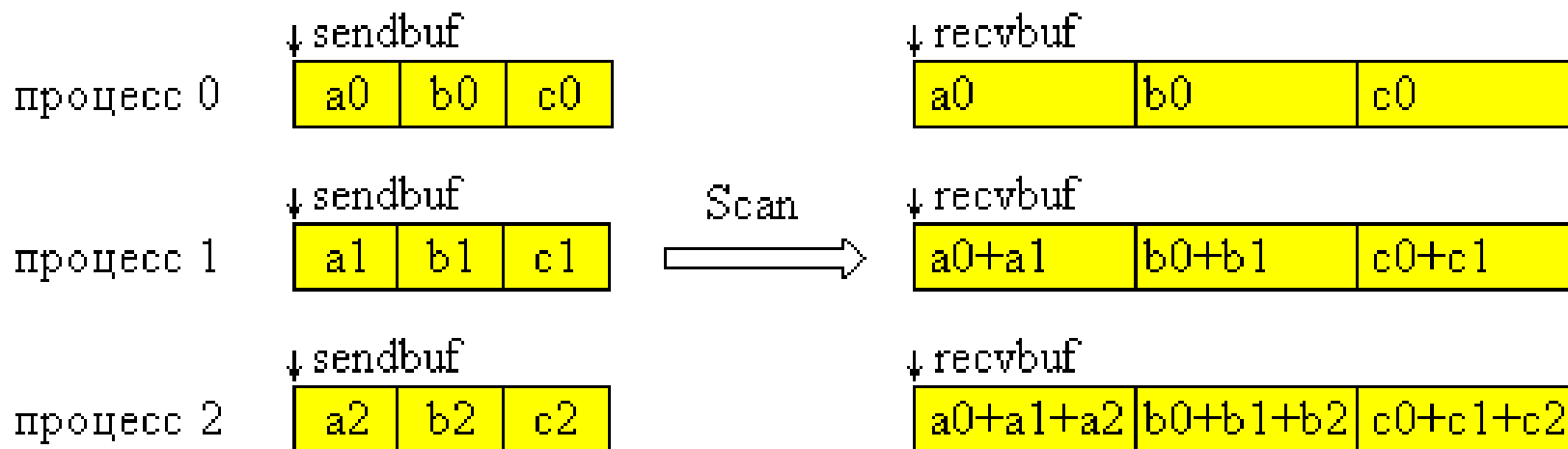
Глобальные вычислительные операции над распределенными данными

Функция **MPI_Scan** выполняет префиксную редукцию. Параметры такие же, как в **MPI_Allreduce**, но получаемые каждым процессом результаты отличаются друг от друга. Операция пересылает в буфер приема i -го процесса редукцию значений из входных буферов процессов с номерами $0, \dots, i$ включительно.

```
int MPI_Scan(void* sendbuf, void* recvbuf, int count,  
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

IN sendbuf	- адрес начала входного буфера
OUT recvbuf	- адрес начала буфера приема
IN count	- число элементов во входном буфере
IN datatype	- тип элементов во входном буфере
IN op	- операция, по которой выполняется редукция
IN comm	- коммуникатор

Глобальные вычислительные операции над распределенными данными



На данной схеме “+” - ассоциативная операция, которая выполняется над парой операндов типа datatype и возвращает результат того же типа:
MPI_MAX, MPI_MIN, MPI_SUM,
MPI_PROD, MPI_LAND, MPI_BAND,
MPI_LOR, MPI_BOR, MPI_LXOR,
MPI_BXOR, MPI_MAXLOC,
MPI_MINLOC

Коллективные асинхронные операции

- **MPI_Ibarrier**
- **MPI_Ibcast**
- **MPI_Igather**
- **MPI_Iscatter**
- **MPI_lallgather**
- **MPI_lalltoall**

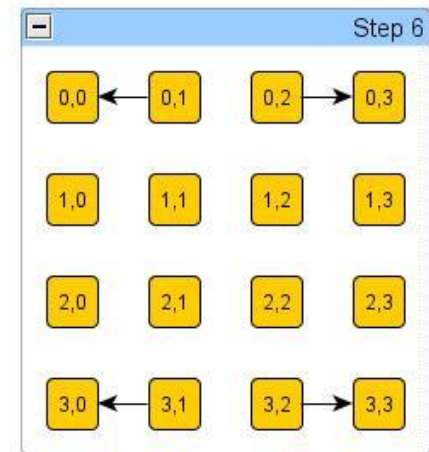
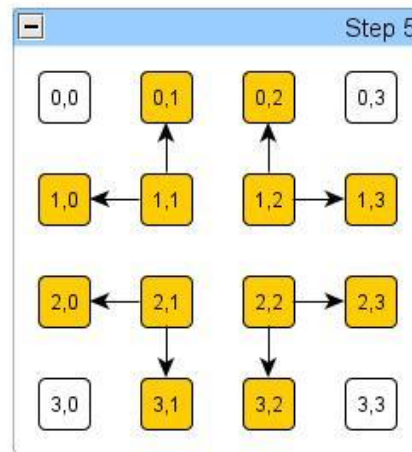
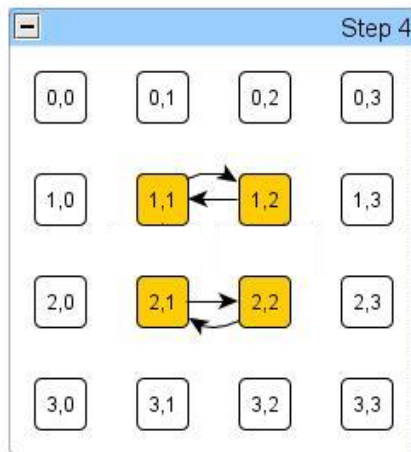
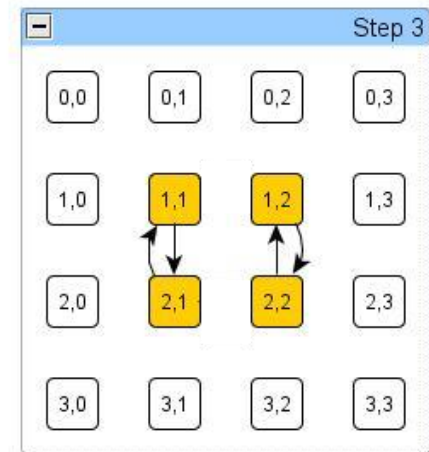
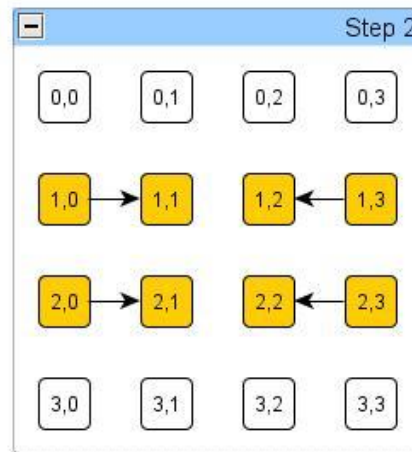
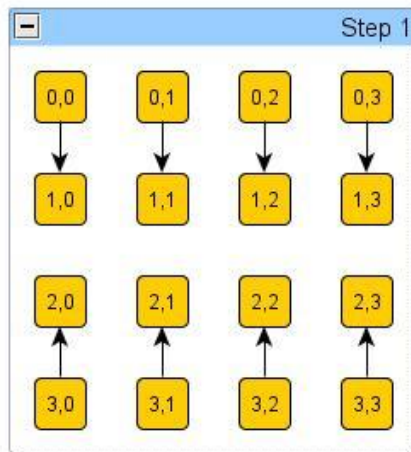
Редукционные операции

- **MPI_Ireduce**
- **MPI_lallreduce**
- **MPI_Ireduce_scatter**
- **MPI_Iscan**
-

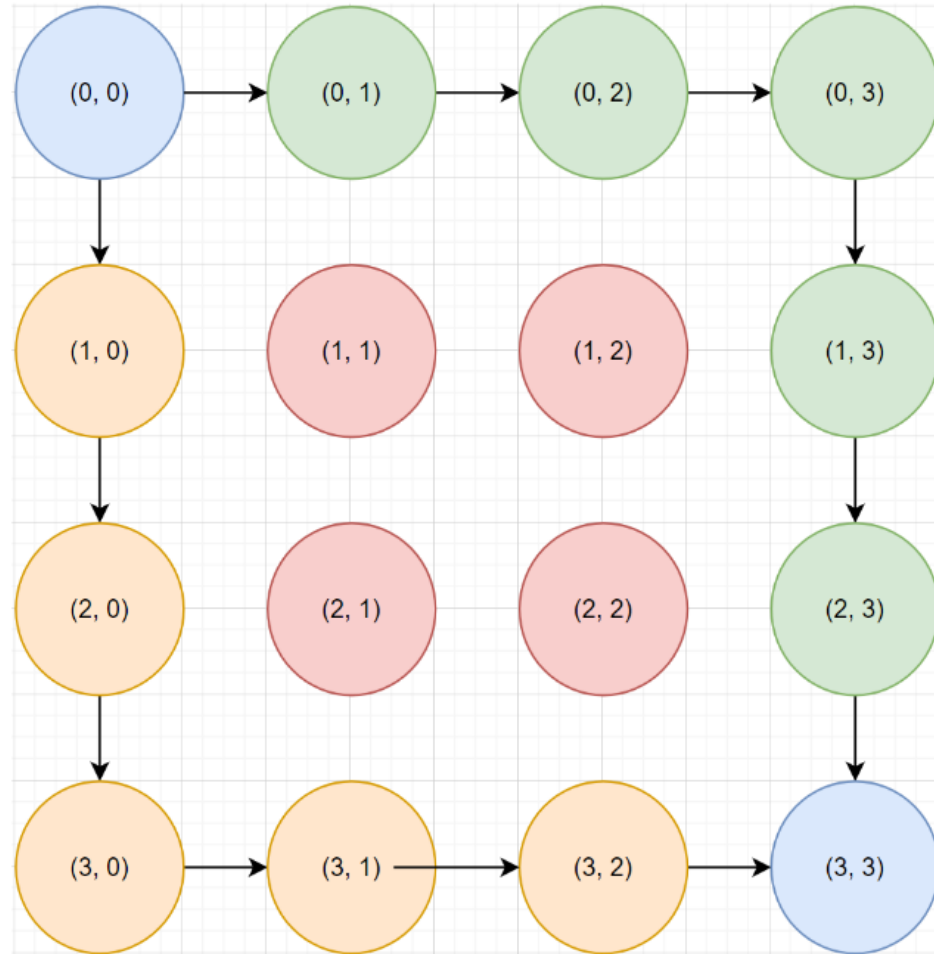
Коллективные асинхронные операции

```
MPI_Request req;
switch(rank) {
  case 0:
    MPI_lalltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, &req);
    MPI_Wait(&req, MPI_STATUS_IGNORE);
    break;
  case 1:
    MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm);
    break;
}
/* erroneous false matching of Alltoall and lalltoall */
```

Реализация MPI_Allreduce



Передача длинного сообщения



Partitioned Point-to-Point Communication

```
#include "mpi.h"
#define PARTITIONS 8
#define COUNT 5
int main(int argc, char *argv[])
{
    double message[PARTITIONS*COUNT];
    MPI_Count partitions = PARTITIONS;
    int source = 0, dest = 1, tag = 1, flag = 0;
    int myrank, i;
    int provided;
    MPI_Request request;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
    if (provided < MPI_THREAD_SERIALIZED)
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

Partitioned Point-to-Point Communication

```
if (myrank == 0)
{
    MPI_Psend_init(message, partitions, COUNT, MPI_DOUBLE, dest, tag,
        MPI_COMM_WORLD, MPI_INFO_NULL, &request);
    MPI_Start(&request);
    for(i = 0; i < partitions; ++i)
    {
        /* compute and fill partition #i, then mark ready: */
        MPI_Pready(i, request);
    }
    while(!flag)
    {
        /* do useful work #1 */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* do useful work #2 */
    }
    MPI_Request_free(&request);
}
```

Partitioned Point-to-Point Communication

```
else if (myrank == 1)
{
    MPI_Precv_init(message, partitions, COUNT, MPI_DOUBLE, source, tag,
        MPI_COMM_WORLD, MPI_INFO_NULL, &request);
    MPI_Start(&request);
    while(!flag)
    {
        /* do useful work #1 */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* do useful work #2 */
    }
    MPI_Request_free(&request);
}
MPI_Finalize();
return 0;
}
```

Упаковка и распаковка данных

Входящие в состав сообщения данные могут быть **упакованы** в буфер при помощи функции:

```
int MPI_Pack ( void *data, int count, MPI_Datatype type, void *buf, int  
bufsize, int *bufpos, MPI_Comm comm),
```

где `data` – буфер памяти с элементами для упаковки,

`count` – количество элементов в буфере,

`type` – тип данных для упаковываемых элементов,

`buf` - буфер памяти для упаковки,

`buflen` – размер буфера в байтах,

`bufpos` – позиция для начала записи в буфер (в байтах от начала буфера),

`comm` - коммуникатор для упакованного сообщения.

Упаковка и распаковка данных

Полученное сообщение может быть **распаковано** при помощи функции:

```
int MPI_Unpack (void *buf, int bufsize, int *bufpos, void *data, int count,  
MPI_Datatype type, MPI_Comm comm)
```

где `buf` - буфер памяти с упакованными данными,

`buflen` – размер буфера в байтах,

`bufpos` – позиция начала данных в буфере (в байтах от начала буфера),

`data` – буфер памяти для распаковываемых данных,

`count` – количество элементов в буфере,

`type` – тип распаковываемых данных,

`comm` - коммуникатор для упакованного сообщения.

Упаковка и распаковка данных

Для определения необходимого размера буфера для упаковки может быть использована функция:

```
int MPI_Pack_size (int count, MPI_Datatype type, MPI_Comm comm, int *size),
```

которая в параметре **size** указывает необходимый размер буфера для упаковки **count** элементов типа **type**.

После упаковки всех необходимых данных подготовленный буфер может быть использован в функциях передачи данных с указанием типа **MPI_PACKED**.

Упаковка и распаковка данных

```
char buff[100];
double x, y;
int position, a[2];
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) { /* Упаковка данных */
    position = 0;
    MPI_Pack(&x, 1, MPI_DOUBLE, buff, 100, &position, MPI_COMM_WORLD);
    MPI_Pack(&y, 1, MPI_DOUBLE, buff, 100, &position, MPI_COMM_WORLD);
    MPI_Pack(a, 2, MPI_INT, buff, 100, &position, MPI_COMM_WORLD);
}
MPI_Bcast(buff, position, MPI_PACKED, 0, MPI_COMM_WORLD);
if (myrank != 0) { /* Распаковка сообщения во всех процессах */
    position = 0;
    MPI_Unpack(buff, 100, &position, &x, 1, MPI_DOUBLE, MPI_COMM_WORLD);
    MPI_Unpack(buff, 100, &position, &y, 1, MPI_DOUBLE, MPI_COMM_WORLD);
    MPI_Unpack(buff, 100, &position, a, 2, MPI_INT, MPI_COMM_WORLD);
}
```


Point-to-Point Persistent Communication

```
MPI_Request recv_obj, send_obj;
MPI_Status status;
//Step 1) Initialize send/request objects
MPI_Recv_init (buf1, cnt, tp, src, tag, com, &recv_obj);
MPI_Send_init (buf2, cnt, tp, dst, tag, com, &send_obj);
for (i=1; i<MAXITER; i++)
{
    //Step 2) Use start in place of recv and send
    //MPI_Irecv (buf1, cnt, tp, src, tag, com, &recv_obj);
    MPI_Start (&recv_obj);
    do_work(buf1,buf2);
    //MPI_Isend (buf2, cnt, tp, dst, tag, com, &send_obj);
    MPI_Start (&send_obj);
    //Wait for send to complete
    MPI_Wait (&send_obj, status);
    //Wait for receive to finish (no deadlock!)
    MPI_Wait(&recv_obj, status);
}
//Step 3) Clean up the requests
MPI_Request_free (&recv_obj); MPI_Request_free (&send_obj);
```

Collective Persistent Communication

/* Nonblocking collectives API */

```
for (i = 0; i < MAXITER; i++) {  
    compute(bufA);  
    MPI_Ibcast(bufA, ..., rowcomm, &req[0]);  
    compute(bufB);  
    MPI_Ireduce(bufB, ..., colcomm, &req[1]);  
    MPI_Waitall(2, req, ...);  
}
```

/* Persistent collectives API */

```
MPI_Bcast_init(bufA, ..., rowcomm, &req[0]);  
MPI_Reduce_init(bufB, ..., colcomm, &req[1]);  
for (i = 0; i < MAXITER; i++) {  
    compute(bufA);  
    MPI_Start(req[0]);  
    compute(bufB);  
    MPI_Start(req[1]);  
    MPI_Waitall(2, req, ...);  
}
```

Моделирование сбоя во время работы MPI-программы

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int main(int argc, char *argv[])
{
    int rank, size, rc, len;
    char errstr[MPI_MAX_ERROR_STRING];
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == (size-1) ) raise(SIGKILL);
    rc = MPI_Barrier(MPI_COMM_WORLD);
    MPI_Error_string(rc, errstr, &len);
    printf("Rank %d / %d: Notified of error %s. Stayin' alive!\n", rank, size, errstr);
    MPI_Finalize();
}
```

Управление группами процессов

Для получения группы, связанной с существующим коммуникатором, используется функция:

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

Далее, 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).
```

Управление группами процессов

Для получения новых групп над имеющимися группами процессов могут быть выполнены операции объединения, пересечения и разности:

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 ).
```

При конструировании групп может оказаться полезной специальная пустая группа **MPI_COMM_EMPTY**.

Управление группами процессов

Ряд функций MPI обеспечивает 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 )
```

Создание нового коммуникатора из подмножества процессов существующего коммуникатора:

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

Управление группами процессов

```
MPI_Group WorldGroup, WorkerGroup;  
MPI_Comm Workers;  
int ranks[1];  
ranks[0] = 0;  
// получение группы процессов в MPI_COMM_WORLD  
MPI_Comm_group(MPI_COMM_WORLD, &WorldGroup);  
// создание группы без процесса с рангом 0  
MPI_Group_excl(WorldGroup, 1, ranks, &WorkerGroup);  
// Создание коммуникатора по группе  
MPI_Comm_create(MPI_COMM_WORLD, WorkerGroup, &Workers);  
  
...  
MPI_Group_free(&WorkerGroup);  
MPI_Comm_free(&Workers);
```

Моделирование сбоя во время работы MPI-программы

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int main(int argc, char *argv[])
{
    int rank, size, rc, len;
    char errstr[MPI_MAX_ERROR_STRING];
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
    MPI_Barrier(MPI_COMM_WORLD);
    if (rank == (size-1) ) raise(SIGKILL);
    rc = MPI_Barrier(MPI_COMM_WORLD);
    MPI_Error_string(rc, errstr, &len);
    printf("Rank %d / %d: Notified of error %s. Stayin' alive!\n", rank, size, errstr);
    MPI_Finalize();
}
```


Моделирование сбоя во время работы MPI-программы

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
static void verbose_errhandler(MPI_Comm* comm, int* err, ...) {
    int rank, size, len;
    char errstr[MPI_MAX_ERROR_STRING];
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Error_string( *err, errstr, &len );
    printf("Rank %d / %d: Notified of error %s\n",
        rank, size, errstr);
}
```

Моделирование сбоя во время работы MPI-программы

```
int main(int argc, char *argv[]) {  
    int rank, size;  
    MPI_Errhandler errh;  
    MPI_Init(NULL, NULL);  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
    MPI_Comm_size(MPI_COMM_WORLD, &size);  
    MPI_Comm_create_errhandler(verbose_errhandler, &errh);  
    MPI_Comm_set_errhandler(MPI_COMM_WORLD, errh);  
    MPI_Barrier(MPI_COMM_WORLD);  
    if( rank == (size-1) ) raise(SIGKILL);  
    MPI_Barrier(MPI_COMM_WORLD);  
    printf("Rank %d / %d: Stayin' alive!\n", rank, size);  
    MPI_Finalize();  
}
```