Информатика и применение компьютеров в научных исследованиях

Работа в операционной системе Linux и параллельные вычисления в среде MPI

Шориков Алексей Олегович

ISA: CISC и RISC, что эффективнее?

CISC (Complex Instruction Set Computing): x86

- нефиксированное значение длины команды;
- арифметические действия кодируются в одной команде;
- небольшое число регистров, каждый из которых выполняет строго определённую функцию.

RISC: (Restricted Instruction Set Computing) AIX, PowerPC

- Фиксированная длина машинных инструкций.
- Специализированные команды для операций с памятью — чтения или записи
- Большое количество регистров общего назначения
- Отсутствие поддержки операций вида «изменить» над укороченными типами данных — байт, 16-разрядное слово.
- Отсутствие микропрограмм внутри самого процессора.

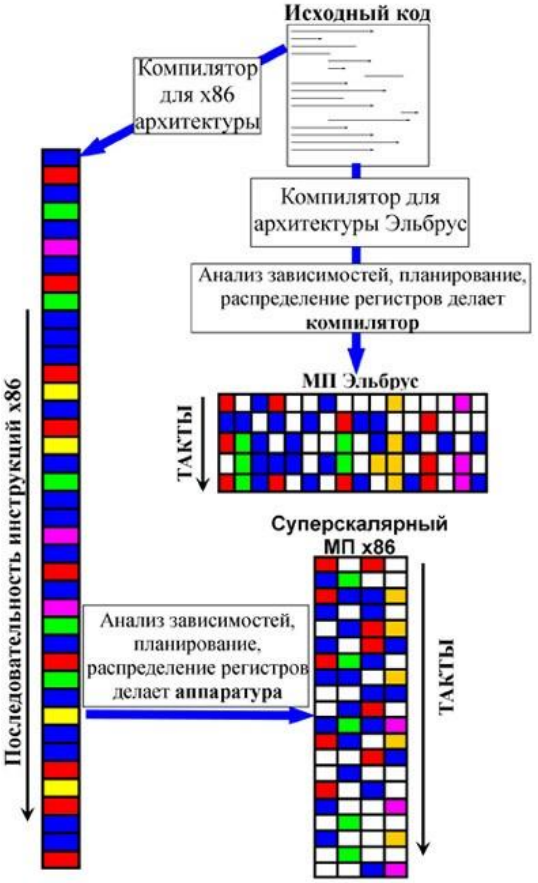
ARM: (Advansed RISC Machine) Snapdragon XXX (Qualcom), MTKXXXX/PXX (MediaTek), Kirin (HiSilicon/Huawei) AXX (Apple) и т.д.

VLIW: (Very Long Instruction Word) IA32, ЭЛЬБРУС

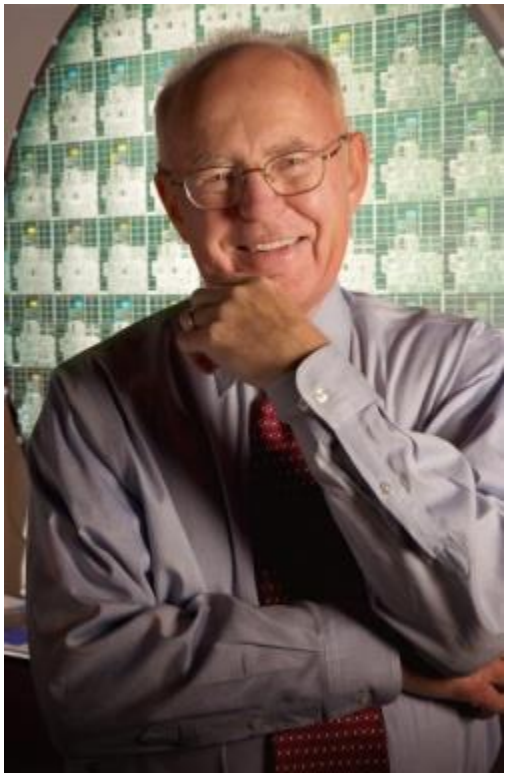
Суперскалаярные процессора (параллелизм на уровне инструкций) Intel Pentium, AMD Athlon и выше. гибридные CPU с RISC ядром. CISC-инструкции преобразовываются в набор внутренних RISC-команд.

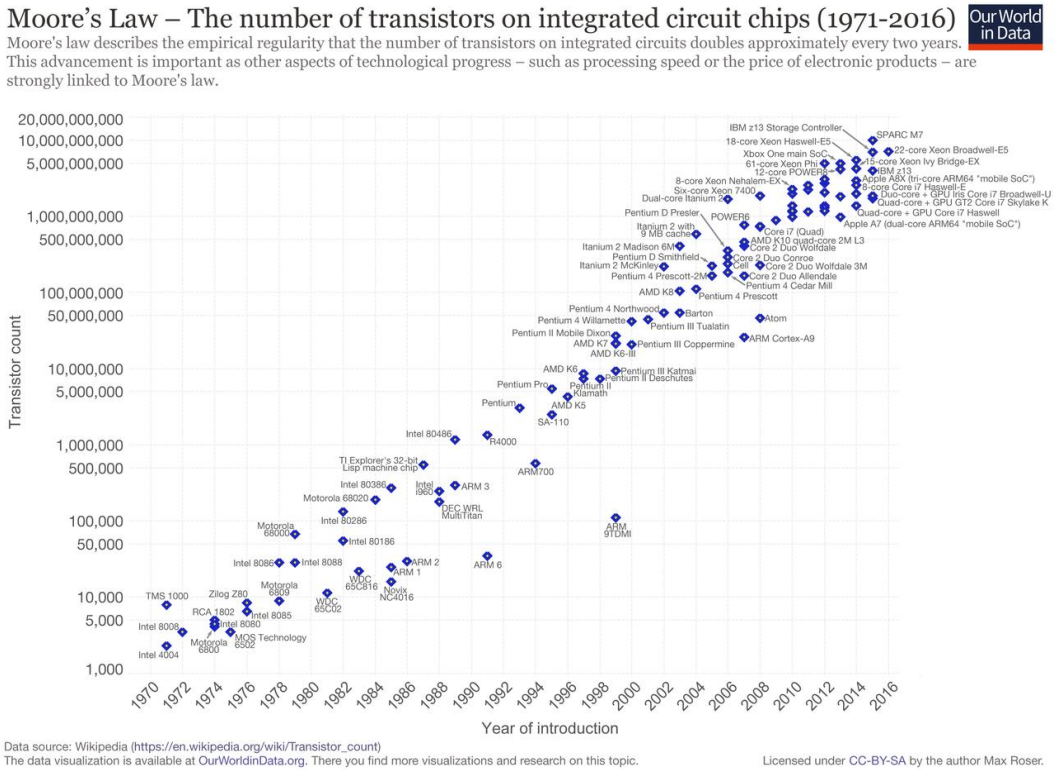
- внеочередное исполнение;
- переименование регистров;
- объединение нескольких команд в одну.

RISC-V: open-source hardware



Закон Мура





Закон Мура число транзисторов на микросхеме увеличивается в 2 раза за 2 года

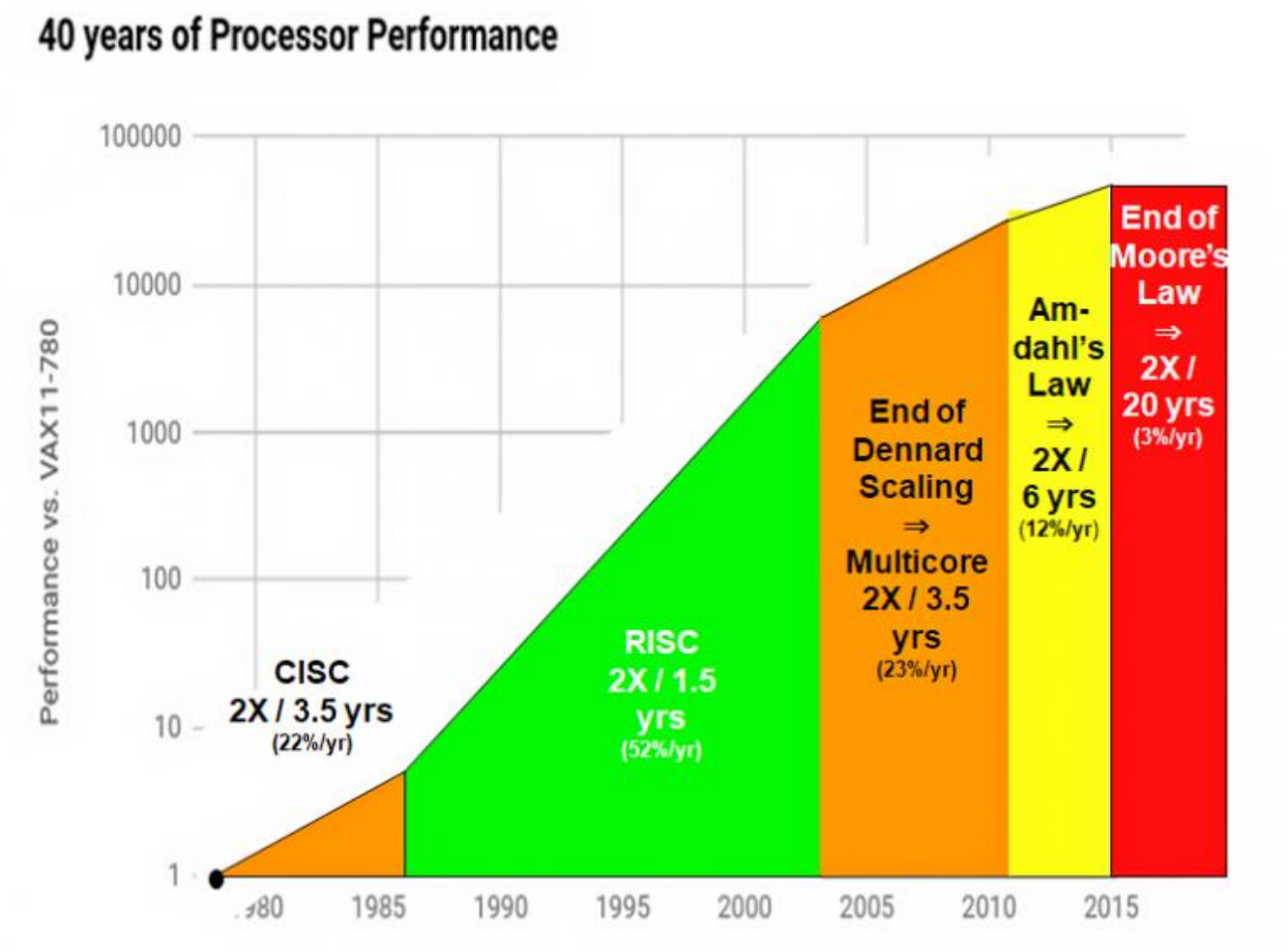
Закон Деннара «уменьшая размеры транзистора и повышая тактовую частоту процессора, мы можем легко повышать его производительность»

эффективность микропроцессоров увеличивается в 2 раза за 3.5 года

Закон Вирта — «Программы становятся медленнее куда шустрее, чем компьютеры становятся быстрее».

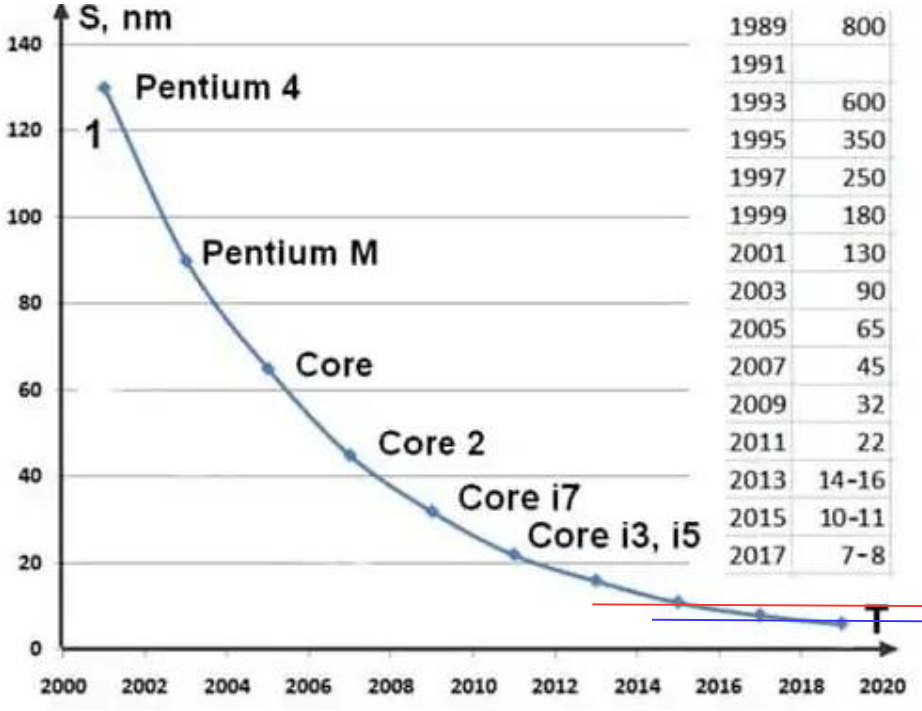
The hope is that the progress in hardware will cure all software ills. However, a critical observer may observe that software manages to outgrow hardware in size and sluggishness. Мартин Райзер.

Производительность CPU растет?



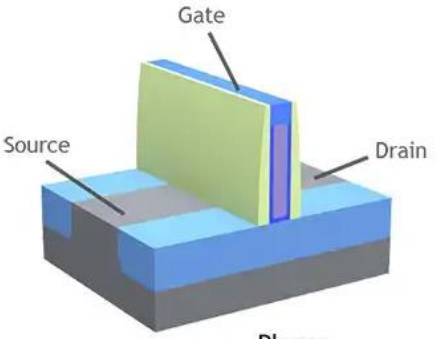
Производительность процессоров перестала расти в 2010-х. Прогресс идет за счет увеличения числа ядер, и повышения энегроэффективности за счет уменьшения техпроцесса.

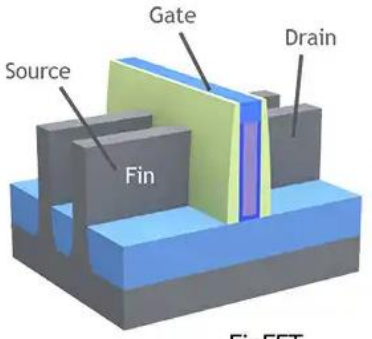
Техпроцесс



1989	800
1991	
1993	600
1995	350
1997	250
1999	180
2001	130
2003	90
2005	65
2007	45
2009	32
2011	22
2013	14-16
2015	10-11
2017	7-8

Si (Fd-3m), a = 5.4307 Å, d(Si-Si)= 2.34 Å



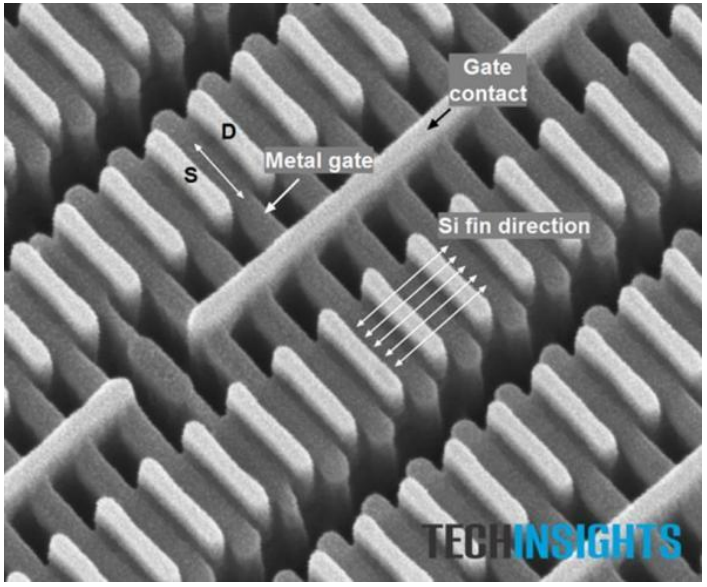


2019

Intel 10nm

Qualcomm, Huawei

7nm



TSMC:

7nm vs. 10 nm

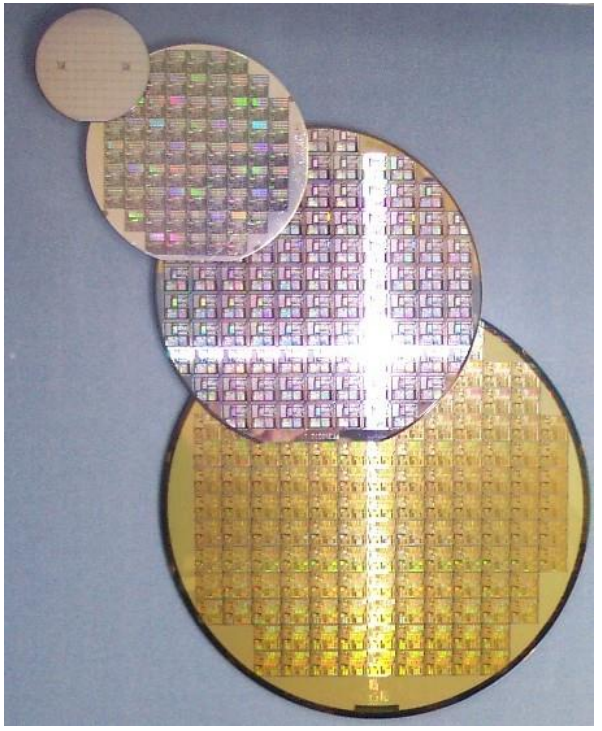
Производительность вырастет на 30%

Энергопотребление Уменьшится на 50%

Стоимость разработки $270M,

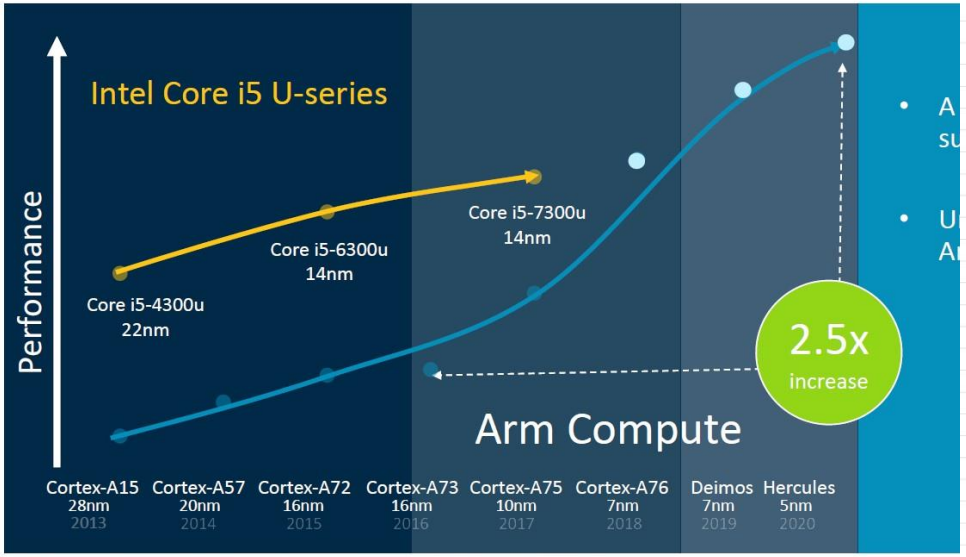
Строительство фабрики $1-1.5B

Объем производства 150M чипов/год

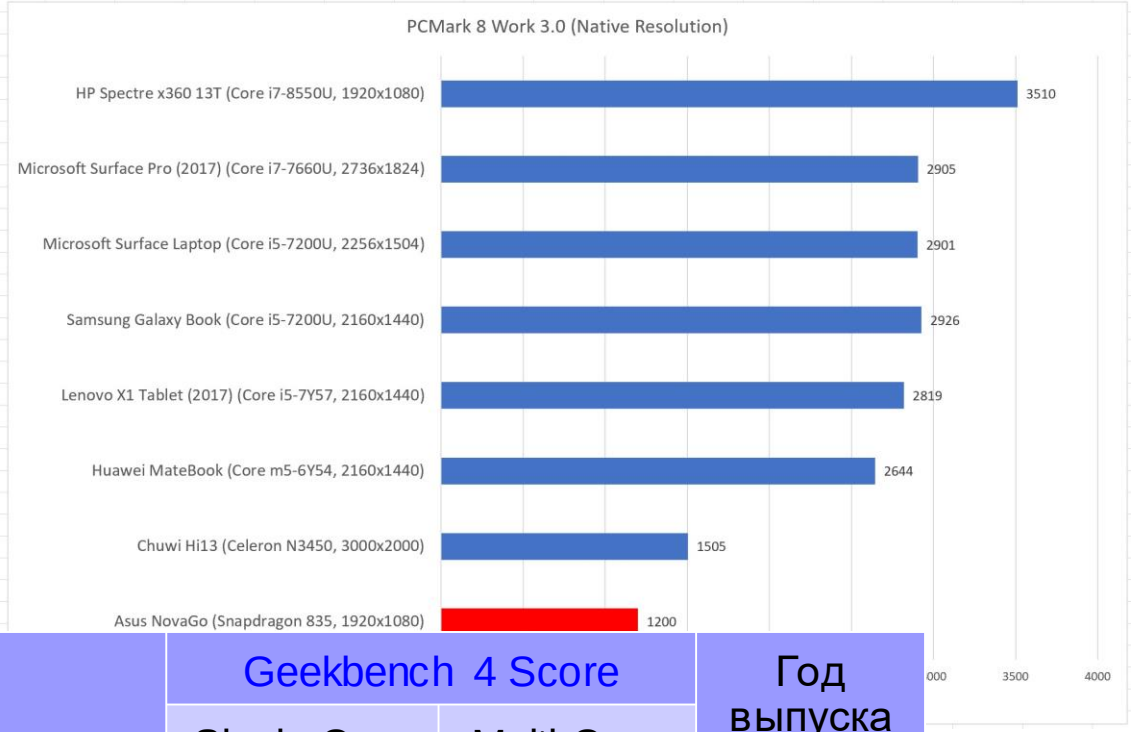


Техпроцесс

Path to Compute performance leadership with efficiency



Single-core performance based on SPECINT2k6, Intel measured, Arm Compute estimated
Graph not to scale



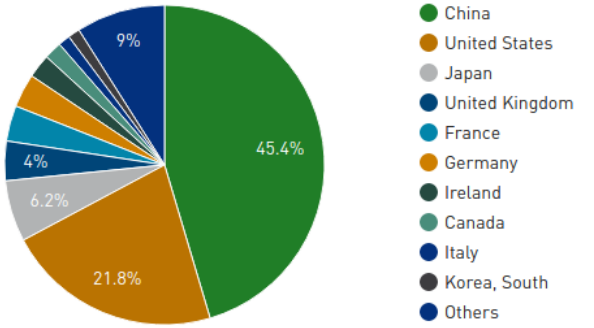
CPU	Geekbench 4 Score		Год выпуска
	Single-Core	Multi-Core	
Intel Core i9-9900K 3.6 GHz (8 cores)	6152	33259	Q4'2018
Qualcomm Snapdragon 845 1.8 GHz (8 cores)	2408	8361	Q4'2017
Intel Xeon E5640 2.7 GHz (4 cores)	2412	8886	Q1'2010
Intel Xeon E5-2650 2.8 GHz (8 cores)	2650	14685	Q1'2012
Intel Core i5-7300U 2.6 GHz (2 cores)	3751	7307	Q1'2017
Intel Core i3-6006U 2.0 GHz (2 cores)	2400	4600	Q4'2016

Суперкомпьютеры в мире: www.top500.org

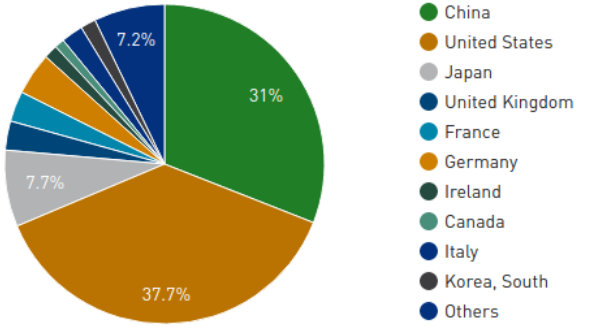
Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM	2,397,824	143,500.0	200,794.9	9,783
2	DOE/NNSA/LLNL United States	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM / NVIDIA / Mellanox	1,572,480	94,640.0	125,712.0	7,438
3	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
4	National Super Computer Center in Guangzhou China	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 NUDT	4,981,760	61,444.5	100,678.7	18,482
5	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 Cray Inc.	387,872	21,230.0	27,154.3	2,384

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
79	**Lomonosov 2** - T-Platform A-Class Cluster, Xeon E5-2697v3 14C 2.6GHz,Intel Xeon Gold 6126, Infiniband FDR, Nvidia K40m/P-100 , T-Platforms Moscow State University - Research Computing Center Russia	64,384	2,478.0	4,946.8	
283	Cray XC40, Xeon E5-2697v4 18C 2.3GHz, Aries interconnect , Cray Inc./T-Platforms Main Computing Center of Roshydromet Russia	35,136	1,200.3	1,293.0	
487	**Lomonosov** - T-Platforms T-Blade2/1.1, Xeon X5570/X5670/E5630 2.93/2.53 GHz, Nvidia 2070 GPU, PowerXCell 8i Infiniband QDR , T-Platforms Moscow State University - Research Computing Center Russia	78,660	901.9	1,700.2	2,800

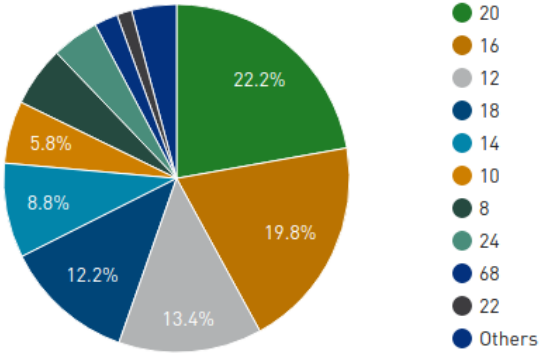
Countries System Share



Countries Performance Share



Cores per Socket System Share



Обзор архитектур МВС

Первая многопроцессорная вычислительная система (70гг.)
ILLIAC IV, 64 (в проекте до 256) процессорных элемента (ПЭ), работающих по единой программе, применяемой к содержимому собственной оперативной памяти каждого ПЭ. Обмен данными между процессорами осуществлялся через специальную матрицу коммуникационных каналов.

Многопроцессорные системы с массовым параллелизмом, или МВС с MPP-архитектурой (MPP – Massively Parallel Processing) - класс МВС с распределенной памятью и с произвольной коммуникационной системой

середина 80-х годов.
Первые промышленные образцы мультипроцессорных систем на базе векторно-конвейерных компьютеров (фирма Cray). Такие системы были чрезвычайно дорогими и производились небольшими сериями, имели от 2 до 16 процессоров, которые имели равноправный (симметричный) доступ к общей оперативной памяти (симметричные мультипроцессорные системы - Symmetric Multi-Processing – SMP).

Обзор архитектур МВС

SMP-архитектура имеет ограниченные возможнось по наращиванию числа процессоров (общая шина памяти)

NUMA-архитектуры (Non Uniform Memory Access) архиетктура МВС, в которых память физически разделена, но логически общедоступна. При этом время доступа к различным блокам памяти становится неодинаковым (Cray T3D время доступа к памяти другого процессора было в 6 раз больше, чем к своей собственной).

Существует четыре основные направления развития МВС:

- векторно-конвейерные суперкомпьютеры
- SMP системы
- MPP системы
- кластерные системы

Векторно-конвейерные суперкомпьютеры

Первый векторно-конвейерный компьютер Cray-1 (1976г.)
CRAY J90/T90, CRAY SV1, NEC SX-4/SX-5

- конвейерная организация обработки потока команд
- введение в систему команд набора векторных операций, которые позволяют оперировать с целыми массивами данных

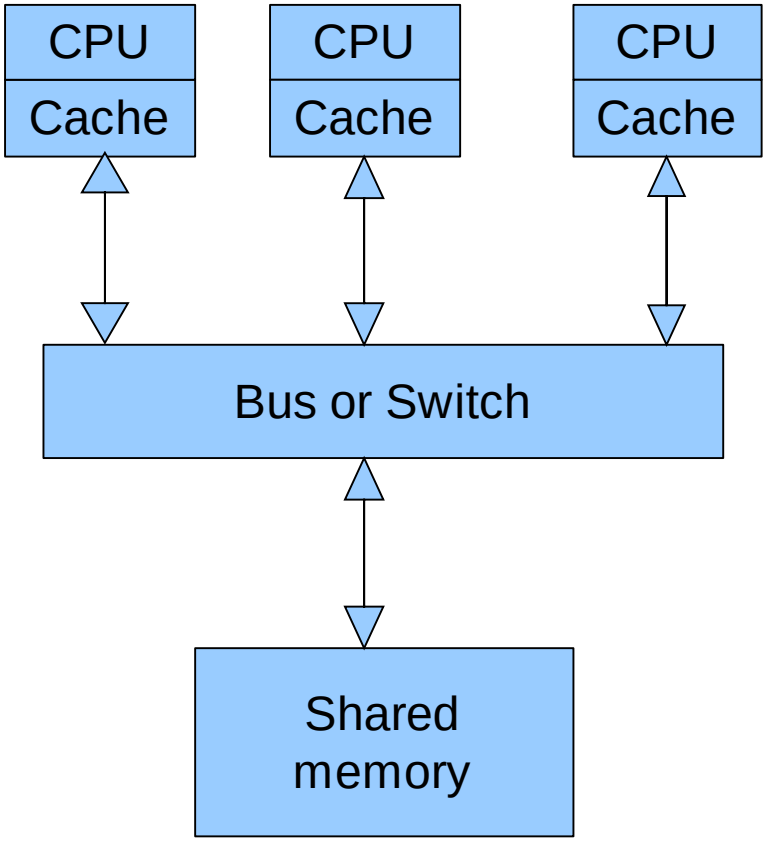
Векторные процессоры имеют сложную структуру и содержат множество арифметических устройств.
Основное назначение векторных операций состоит в распараллеливании выполнения операторов цикла

Распараллеливание происходит программно средствами компилятора - не требуется специальной технологии программирования

Вычислительный узел (2-16) CPU с SMP архитектурой
Узлы объединяются с помощью коммутаторов (NUMA или MPP архитектура)

Суперскалярные CPU – дешевая замена векторным процессорам.

Симметричные мультипроцессорные системы SMP



Проблема – большое число конфликтов при обращении к общей шине.
Решение - разделение памяти на блоки, подключение к которым с помощью коммутаторов позволило распараллелить обращения от различных процессоров.

Неприемлемо большие накладные расходы для систем более чем с 32 CPU

Симметричные мультипроцессорные системы SMP

Intel Core i7 2700k 3.2 GHz
пропускная способность кэш-памяти 28982 MB/s
латентность 5.6 ns
RAM DDR3-1600 (Sandy Bridge-E в двухканальном режимах)
пропускная способность 19740 MB/s
Латентность 49.1 ns

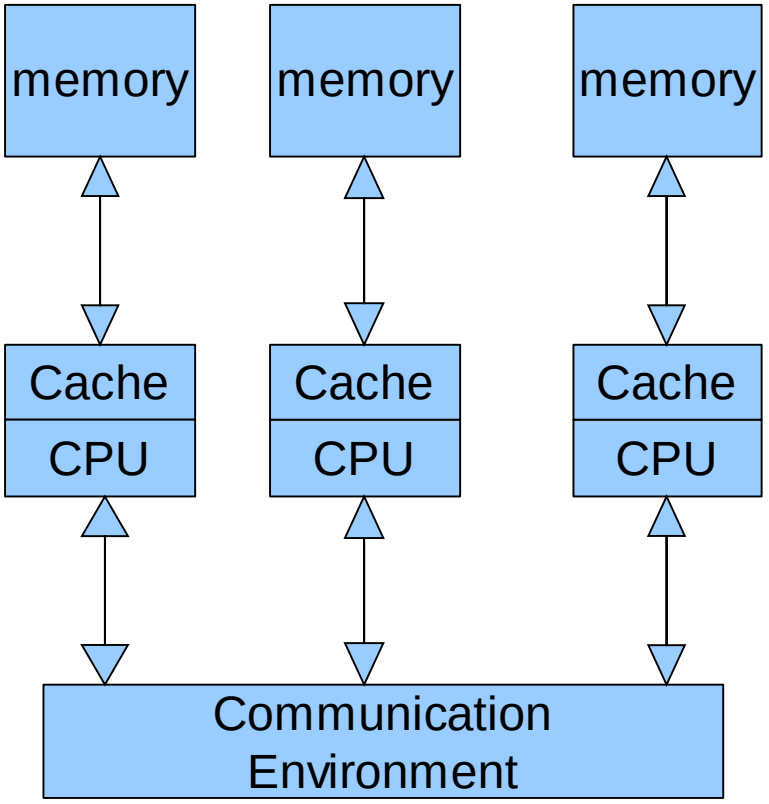
Fast Ethernet (100BASE-T) до 100 MB/s
Gigabit Ethernet (1000BASE-T) до 1000 MB/s
SATA Revision 3.0 до 6000 MB/s

В многопроцессорных системах, построенных на базе микропроцессоров со встроенной кэш-памятью, нарушается принцип равноправного доступа к любой точке памяти. Данные, находящиеся в кэш-памяти некоторого процессора, недоступны для других процессоров.
ccNUMA (cache coherent Non Uniform Memory Access) архитектура МВС, в которой память физически распределена, но логически общедоступна. Имеет 3 уровня памяти:

- Кэш-память процессора
- Локальная оперативная память
- Удаленная оперативня память

CcNUMA - до 256 CPU

Системы с массовым параллелизмом (MPP)



Каждый из узлов состоит из одного или нескольких CPU, собственной оперативной памяти, стемы ввода/вывода. На каждом узле может функционировать либо полноценна ОС (RS/6000 SP2), либо урезанныйвариант, оддерживающий только базовые функции ядра, а полноценная ОС работает на специальном управляющем компьютере (Cray T3E, nCUBE2).

Системы с массовым параллелизмом (MPP)

3 уровня памяти:

- Кэш-память процессора
- Локальная оперативная память
- оперативня память других узлов

Отсутствует возможность прямого доступа к данным, расположенным на других узлах. Доступ к памяти других узлов реализуется с помощью механизма передачи сообщений (MPI, PVM и т.д.).

Высокая степень масштабируемости - возможность практически неограниченного наращивания числа процессоров.

Недостаток MPP - проблема эффективности коммуникационной среды.

- В компьютерах Intel Paragon процессоры образовывали прямоугольную двумерную сетку (в каждом узле 4 коммуникационных канала). В компьютерах Cray T3D/T3E использовалась топология трехмерного тора (в каждом узле 6 коммуникационных каналов). В МВС nCUBE используется топология n-мерного гиперкуба.

При обмене данными между CPU, не являющимися ближайшими, происходит трансляция данных через промежуточные узлы (за счет дополнительных аппаратных средств)

- Иерархическая система высокоскоростных коммуаторов. Такая топология дает возможность прямого обмена данными между любыми узлами, без участия в этом промежуточных узлов.

Кластерные системы

В отличие от MPPв кластерных системах в качестве узла используются обычные серийно выпускаемые компьютеры, связанные с помощью стандартного высокоскоростного сетевого обуродования. Связи между узлами с помощью MPI

Гетерогенные кластеры — узлы имеют разную архитектру/состав
Распределенные кластеры — узлы расположенны географически в разных местах

Преимущества:
- Низкая стоимость
- Свободно распостроняемые программные продукты — MPI, диспетчеры задач, математические библиотеки и т.д.

Недостатки:
- Низкая скорость обмена информацией между узлами

ScaLapack
Скорость межпроцессорных обменов между двумя узлами, измеренная в Мб/сек, должна быть не менее 1/10 пиковой производительности вычислительного узла, измеренной в Mflops

Классификация вычислительных систем

1. SISD (Single Instruction Single Data) – единственный поток команд и единственный поток данных (классическая машина фон Неймана, все однопроцессорные системы).
2. SIMD (Single Instruction Multiple Data) – единственный поток команд и множественный поток данных. Матричные компьютеры, в которых все процессорные элементы выполняют одну и ту же программу, применяемую к своим локальным данным.
3. MISD (Multiple Instruction Single Date) – множественный поток команд и единственный поток данных. Нет ни одного примера реально существующей системы, работающей на этом принципе
4. MIMD (Multiple Instruction Multiple Date) – множественный поток команд и множественный поток данных. Практически все современные многопроцессорные системы.

Средства программирования МВС

Системы с общей памятью (SMP)
Для распараллеливания программ используется механизм порождения нитей

реализуется автоматически (средствами компилятора) или заданием директив (OpenMP)

Системы с распределенной памятью (MPP, кластеры)
единственно возможным механизмом взаимодействия между ними является механизм передачи сообщений (MPI,PVM).

Основная цель MPI – это обеспечение полной независимости приложений, написанных с использованием MPI, от архитектуры многопроцессорной системы, без какой-либо существенной потери производительности.

MPI – это библиотека функций, обеспечивающая взаимодействие параллельных процессов с помощью механизма передачи сообщений. Поддерживаются интерфейсы для языков C и FORTRAN (C++)

Альтернативный подход HPF (реализует идею инкрементального распараллеливания и модель общей памяти на системах с распределенной памятью)

Высокопроизводительные вычисления на MPP системах

Пути ускорения вычислений:

- использовать более производительную вычислительную систему с более быстрым процессором и более скоростной системной шиной;
- оптимизировать программу, например, в плане более эффективного использования скоростной кэш-памяти;
- распределить вычислительную работу между несколькими процессорами, т.е. перейти на параллельные технологии.

Параллельное программирование на MPP системах

- SIMD
- MIMD

Разработка параллельной программы подразумевает разбиение задачи на P подзадач, каждая из которых решается на отдельном процессоре.

```
if(myid.eq.0)then
        {блок вычислений 0 процесса}
elseif(myid.eq.1)then
        {блок вычислений 1 процесса}
...
endif
result = reduce(result_cpu0,result_cpu1, … )
```

Разбиение задачи на подзадачи:

```
do i=1,1000
   c(i)=c(i)+a(i)
enddo
```

- Все массивы целиком хранятся в каждом процессоре, каждый процесс вычисляет стартового и конечного значений переменной цикла и хранит свою копию всего массива, в которой будет модифицирована только часть элементов. В конце вычислений, потребуется сборка модифицированных частей со всех процессов.
- Все или часть массивов распределены по процессорам, т.е. в каждом процессе хранится часть массива. Требуется алгоритм установления связи индексов локального массива (на процессе #myid) с глобальными индексами всего массива,

Эффективное распределения данных по процессорам основной вопрос параллельного программирования

4 этапа разработки параллельного алгоритма:

- разбиение задачи на минимальные независимые подзадачи (partitioning);
- установление связей между подзадачами (communication);
- объединение подзадач с целью минимизации коммуникаций (agglomeration);
- распределение укрупненных подзадач по процессорам таким образом, чтобы обеспечить равномерную загрузку процессоров (mapping).

Высокопроизводительные вычисления на MPP системах

Минимально необходимый набор требуемых свойств, которыми должна обладать многопроцессорная система MPP типа для исполнения на ней параллельных программ

- **Процессоры в системе должны иметь уникальные идентификаторы (номера).**
- **Должна существовать функция идентификации процессором самого себя.**
- **Должны существовать функции обмена между двумя процессорами: посылка сообщения одним процессором и прием сообщения другим процессором.**

Эффективность параллельных программ

решение задачи на P процессорах не дает ускорения в P раз, чем на одном процессоре, и не позволяет решить задачу с объемами данных, в P раз большими.

закон Амдала

$$S \leq 1/(f+(1-f)/P),$$

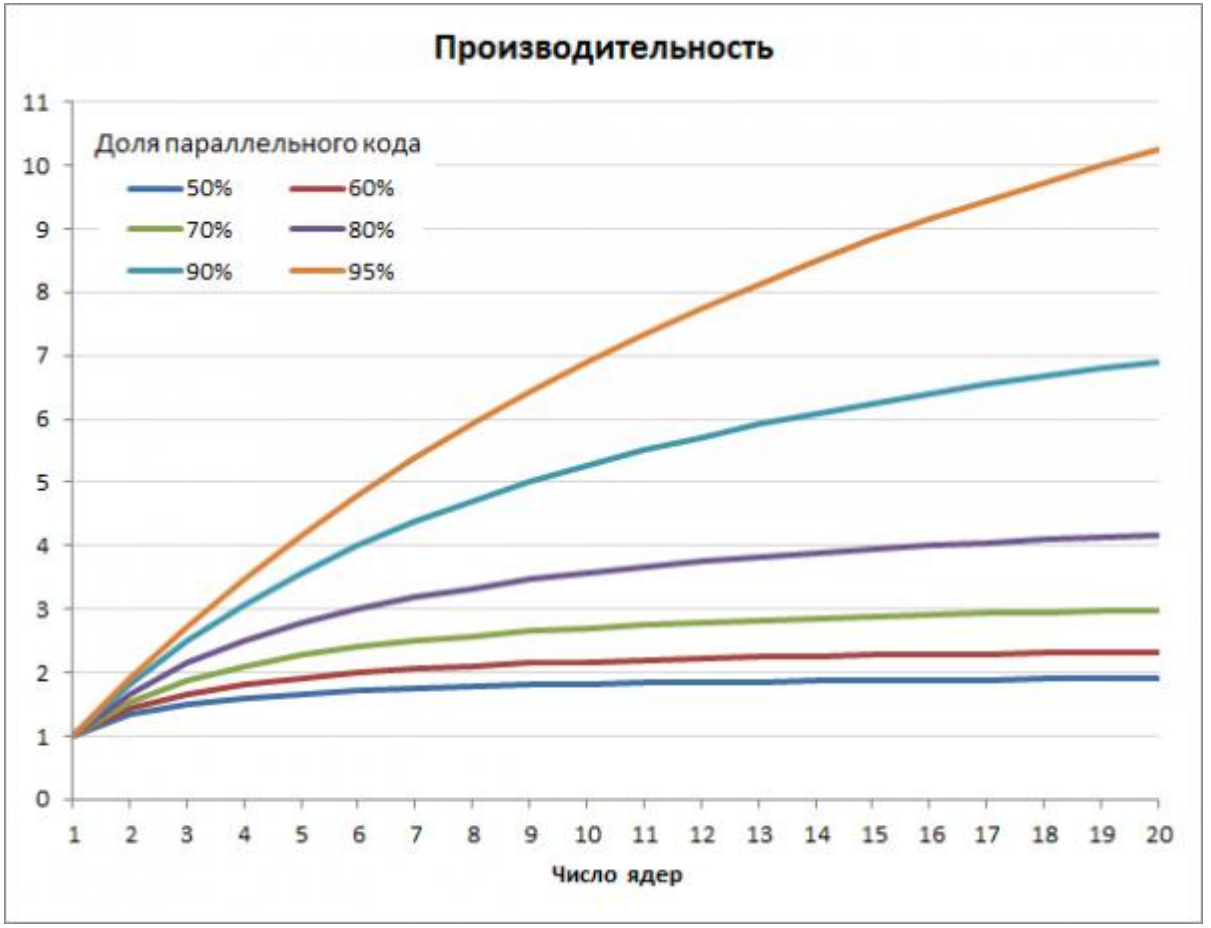
S – ускорение работы программы на P процессорах,
f – доля непараллельного кода в программе

Высокопроизводительные вычисления на MPP системах

Ускорение работы программы в зависимости от доли непараллельного кода

Число CPU	Доля последовательных вычислений,%				
	50	25	10	5	2
	Ускорение работы программы				
2	1.33	1.60	1.82	1.90	1.96
4	1.60	2.28	3.07	3.48	3.77
8	1.78	2.91	4.71	5.93	7.02
16	1.88	3.36	6.40	9.14	12.31
32	1.94	3.66	7.80	12.55	19.75
512	1.99	3.97	9.83	19.28	45.63
2048	2.00	3.99	9.96	19.82	48.83

Ускорение работы программы в зависимости от доли непараллельного кода



Использование высокопроизводительных технологий

распараллеливание программы это лишь одно из средств ускорения ее работы. Не меньший эффект, а иногда и больший, может дать оптимизация однопроцессорной программы. Причина большой разрыва в скорости работы кэш-памяти и основной памяти.

Перемножение двух квадратных матриц размерности N (2*N — 1)*N*N операций.

```
do i = 1,n
  do k = 1,n
    c(i,k)=0.d0
    do l = 1,n
       c(i,k) = c(i,k) + a(i,l)*b(l,k)
    enddo
  enddo
enddo

do i = 1,n
   do j = 1,n
      row(j)=a(i,j)
   enddo
do k = 1,n
   tmp=0.d0
   do l = 1,n
      tmp = tmp + row(l)*b(l,k)
   enddo
   c(i,k)=tmp
enddo
```

Среда параллельного программирования MPI

MPI — message passing interface , библиотека подпрограмм и функций (~130)
MPI-программа представляет собой набор независимых процессов, каждый из которых выполняет свою собственную программу (не обязательно одну и ту же), написанную на языке C или FORTRAN (C++)

поддерживает параллельне программы в стиле MIMD и SIMD

Процессы MPI-программы взаимодействуют друг с другом посредством вызова коммуникационных процедур.

MPI не предоставляет никаких средств для распределения процессов по вычислительным узлам и для запуска их на исполнение.

mpirun -np <число процессов> <имя программы>

компилятор mpif90 или mpif77

подключение библиотеки MPI в тексте программы

```
program main
implicit none
include 'mpif.h'
```

Среда параллельного программирования MPI

Основные понятия:

группа — часть или все параллельные процессы, на которые разделена программа.
Служит для локализации взаимодействия.
Процессы внутри группы нумеруются целым числом от 0 до «число_процессов» -1

область связи — виртуальный объект, описывающий группу процессов.

коммуникатор — объект библиотеки MPI, соответсвующий области связи группы.
Среда для взаимодействия процессов (тип integer)

при инициализации MPI-программы создаются области связи и соответсвующий им коммуникаторы:

MPI_COMM_WORLD
MPI_COMM_SELF
MPI_COMM_NULL

Структура библиотеки MPI

Содержит ~130 подпрограмм и функций

- функции инициализации и закрытия MPI-процессов;
- функции, реализующие коммуникационные операции типа точка-точка;
- функции, реализующие коллективные операции;
- функции для работы с группами процессов и коммуникаторами;
- функции для работы со структурами данных;
- функции формирования топологии процессов.

Любая параллельная программа может быть написана с использованием всего 6 MPI-функций, а достаточно полную и удобную среду программирования составляет набор из 24 функций

Структура библиотеки MPI

Каждая из MPI функций характеризуется способом выполнения:
1. Локальная функция – выполняется внутри вызывающего процесса. Ее завершение не требует коммуникаций.
2. Нелокальная функция – для ее завершения требуется выполнение MPI-процедуры другим процессом.
3. Глобальная функция – процедуру должны выполнять все процессы группы. Несоблюдение этого условия может приводить к зависанию задачи.
4. Блокирующая функция – возврат управления из процедуры гарантирует возможность повторного использования параметров, участвующих в вызове. Никаких изменений в состоянии процесса, вызвавшего блокирующий запрос, до выхода из процедуры не может происходить.
5. Неблокирующая функция – возврат из процедуры происходит немедленно, без ожидания окончания операции и до того, как будет разрешено повторное использование параметров, участвующих в запросе. Завершение неблокирующих операций осуществляется специальными функциями.

Структура библиотеки MPI

В языке FORTRAN большинство MPI-процедур являются подпрограммами (вызываются с помощью оператора CALL), а код ошибки возвращают через дополнительный последний параметр процедуры.

Соответствие типов данных

Тип MPI	Тип Fortran
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CARACTER	CARACTER
MPI_BYTE	-
MPI_PACKED	

Существует возможность создавать собственные типы данных на базе стандартых

Базовые функции MPI

Инициализация MPI-процессов

MPI_INIT(err)

INTEGER :: err

Завершения MPI-процессов

MPI_FINALIZE(err)

INTEGER :: err

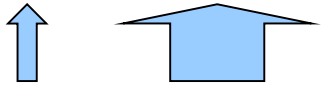
Функция определения числа процессов в области связи

MPI_COMM_SIZE(COMM, nproc, err)

INTEGER :: COMM, nproc, err

Функция определения номера процесса

MPI_COMM_RANK(COMM, myid,err)



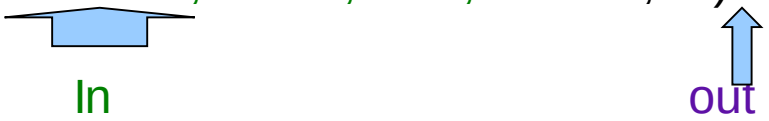
INTEGER :: COMM,myid,err

myid имеют значения [0,nproc-1]

Базовые функции MPI

Функция передачи сообщения

MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,err)



<type> :: BUF(*)

INTEGER :: COUNT, DATATYPE, DEST, TAG, COMM, err

Функция приема сообщения

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,STATUS,err)

<type> :: BUF(*)

INTEGER :: COUNT, DATATYPE, SOURCE, TAG, COMM,STATUS(MPI_STATUS_SIZE), err

Функция отсчета времени (таймер)

DOUBLE PRECISION MPI_WTIME()

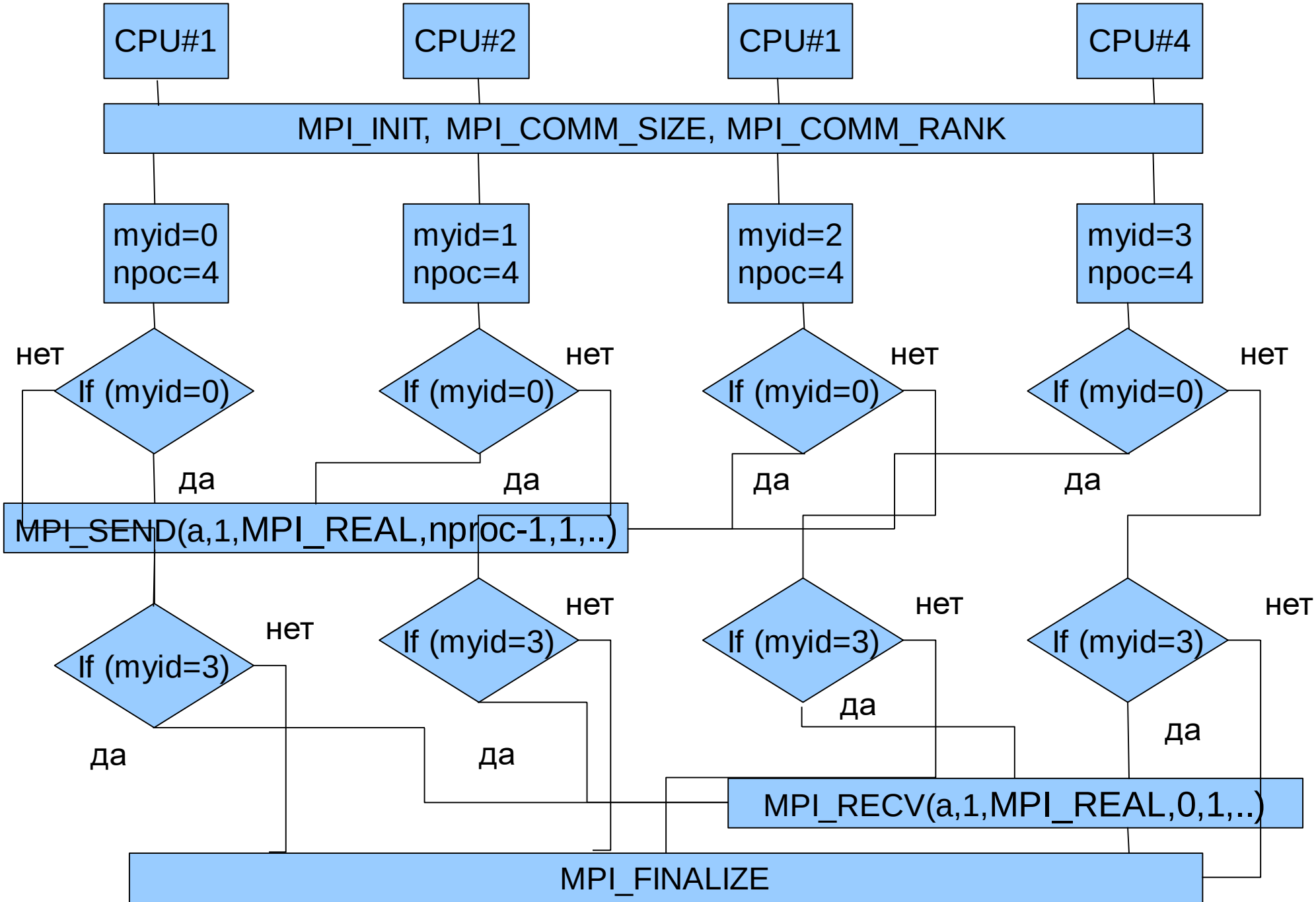
Функция возвращающая разрешение таймера(точность)

DOUBLE PRECISION MPI_WTICK()

Пример MPI-программы

```
program example
 implicit none
include 'mpif.h'
integer ::err,myid,nproc,stat(MPI_STATUS_SIZE)
real      ::a,b
double precision :: t1
call MPI_INIT(ierr)
t1=MPI_WTIME()
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,err)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,err)
write(6,*)'proc ',myid,' from ',nproc,' is alive'
if (myid.eq.0) then
     a = 10.d0
     call MPI_SEND(a,1,MPI_REAL,nproc-1,1,MPI_COMM_WORLD,err)
elseif(myid.eq.nproc-1) then
     call MPI_RECV(b,1,MPI_REAL,0,1,MPI_COMM_WORLD,stat,err)
     write(6,*)'proc ',myid,' get a = ',b
     write(6,*)'working time is',MPI_WTIME()-t1,'+/-',MPI_WTICK()
endif
call MPI_FINALIZE(ierr)
end
```

Пример MPI-программы



Коммуникационные операции типа точка-точка

В коммуникационных операциях типа точка-точка всегда участвуют не более двух процессов: передающий и принимающий.

Блокирующие функции подразумевают выход из них только после полного окончания операции, т.е. вызывающий процесс блокируется, пока операция не будет завершена.

Неблокирующие функции передачи и приема являются функциями инициализации соответствующих операций. Для опроса завершенности операции (и завершения) вводятся дополнительные функции.

Режим выполнения	С блокировкой	Без блокировки
Стандартная посылка	MPI_Send	MPI_Isend
Синхронная посылка	MPI_Ssend	MPI_Issend
Буферизованная посылка	MPI_Bsend	MPI_Ibsend
Согласованная посылка	MPI_Rsend	MPI_Irsend

Коммуникационные операции типа точка-точка

S (synchronous) – синхронный режим передачи данных. Операция передачи данных заканчивается только тогда, когда заканчивается прием данных. **Функция нелокальная.**

B (buffered) – буферизованный режим передачи данных. В адресном пространстве передающего процесса создается буфер обмена, который используется в операциях обмена. Операция посылки заканчивается, когда данные помещены в этот буфер. **Функция локальная**.

R (ready) – согласованный или подготовленный режим передачи данных. Операция передачи данных начинается только тогда, когда принимающий процессор выставил признак готовности приема данных, инициировав операцию приема. Функция нелокальная.

I (immediate) – относится к неблокирующим операциям.

Функции передачи, имеют совершенно одинаковый синтаксис и отличаются только внутренней реализацией.

Все функции передачи и приема сообщений могут использоваться в любой комбинации друг с другом.

Блокирующие коммуникационные операции

Стандартный режим с блокировкой

1. Передающая сторона формирует пакет сообщения, состоящий из передаваемой информации, адреса отправителя (source), адреса получателя (dest), идентификатора сообщения (tag) и коммуникатора (comm), Передает его в системный буфер. **На этом функция посылки сообщения заканчивается.**
2. Сообщение системными средствами передается адресату.
3. Принимающий процессор извлекает сообщение из системного буфера, когда у него появится потребность в этих данных. Содержательная часть сообщения помещается в параметр buf, служебная – в параметр status.

адресная часть принятого сообщения состоит из трех полей:
- коммуникатора (comm) (каждый процесс может одновременно входить в несколько областей связи)
- номера отправителя в этой области связи (source)
- идентификатора сообщения (tag)

Асимметрия операций передачи и приема сообщений

Параметр count (количество принимаемых элементов сообщения) в **процедуре приема** сообщения должен быть не меньше, чем длина принимаемого сообщения. Реально будет приниматься столько элементов, сколько находится в буфере.

Расширенные запросы(только в операциях чтения):

- для идентификаторов сообщений (MPI_ANY_TAG – читать сообщение с любым идентификатором)
- для адресов отправителя (MPI_ANY_SOURCE – читать сообщение от любого отправителя).

Не допускается расширенных запросов для коммуникаторов.

В качестве адресатов source и dest в операциях пересылки данных можно использовать специальный адрес MPI_PROC_NULL.

Поля массива статус	Способ обращения
Процесс-отправитель	status(MPI_SOURCE)
Идентификатор сообщения	status(MPI_TAG)

Блокирующие коммуникационные операции

определения числа фактически полученных элементов сообщения

MPI_GET_COUNT (STATUS, DATATYPE, COUNT, err)

INTEGER :: STATUS (MPI_STATUS_SIZE), DATATYPE, COUNT, err

Операция чтения безвозвратно уничтожает информацию в буфере приема. При этом попытка считать сообщение с параметром count меньше, чем число элементов в буфере, приводит к потере сообщения.

Определить параметры полученного сообщения без его чтения

MPI_PROBE(SOURCE, TAG, COMM, STATUS,err) **выполняется с блокировкой**

INTEGER :: SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),err

Блокирующие коммуникационные операции

Тупиковые ситуации

```
CALL MPI_COMM_RANK(comm, myid, err)
IF (myid.EQ.0) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, err)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, err)
ELSE IF (myid.EQ.1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, err)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, err)
END IF
```

Правильный вариант

```
CALL MPI_COMM_RANK(comm, myid, err)
IF (myid.EQ.0) THEN
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, err)
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, err)
ELSE IF (myid.EQ.1) THEN
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, err)
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, err)
END IF
```

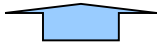
Блокирующие коммуникационные операции

В ситуациях, когда требуется выполнить **взаимный обмен данными** между процессами, безопаснее использовать **совмещенную операцию**

MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG,

RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM,

STATUS, ERR)



<type> :: SENDBUF(*), RECVBUF(*)

INTEGER :: SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), ERR

Или

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, ERR)

Неблокирующие коммуникационные операции

Использование неблокирующих коммуникационных операций повышает безопасность с точки зрения возникновения тупиковых ситуаций, а также может увеличить скорость работы программы за счет совмещения выполнения вычислительных и коммуникационных операций.

Неблокирующие операции используют специальный скрытый (opaque) объект "запрос обмена" (request) для связи между функциями обмена и функциями опроса их завершения.

Если операция обмена завершена, подпрограмма проверки снимает "запрос обмена", устанавливая его в значение MPI_REQUEST_NULL. Снять запрос без ожидания завершения операции можно подпрограммой MPI_Request_free

Неблокирующие коммуникационные операции

Функция передачи сообщения без блокировки

MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,REQUEST, ERR)

<type> :: BUF(*)

INTEGER :: COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, ERR

Функция приема сообщения без блокировки

MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST,ERR)

<type> :: BUF(*)

INTEGER ::COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, ERR

Возврат из подпрограммы происходит немедленно (immediate), без ожидания окончания передачи данных. Поэтому переменную buf повторно использовать нельзя до тех пор, пока не будет погашен “запрос обмена”. Это можно сделать с помощью подпрограмм MPI_Wait или MPI_Test, передав им параметр request.

Неблокирующие коммуникационные операции

Неблокирующая функция чтения параметров полученного сообщения

MPI_IPROBE (SOURCE, TAG, COMM, FLAG, STATUS, ERR)

In out

LOGICAL :: FLAG
INTEGER :: SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), ERR

Если flag=true, то операция завершилась, и в переменной status находятся атрибуты этого сообщения.

Имеется два типа функций завершения неблокирующих операций (ожидание завершения и проверки завершения):

1. Операции семейства WAIT блокируют работу процесса до полного завершения операции.
2. Операции семейства TEST возвращают значения TRUE или FALSE в зависимости от того, завершилась операция или нет. Они не блокируют работу процесса и полезны для предварительного определения факта завершения операции.

Неблокирующие коммуникационные операции

Функция ожидания завершения неблокирующей операции

MPI_WAIT(REQUEST, STATUS, ERR) **нелокальная блокирующая операция**

In out

INTEGER :: REQUEST, STATUS(MPI_STATUS_SIZE), ERR

Функция проверки завершения неблокирующей операции

MPI_TEST(REQUEST, FLAG, STATUS, ERR) **локальная неблокирующая операция**

InOut out

LOGICAL :: FLAG

INTEGER :: REQUEST, STATUS(MPI_STATUS_SIZE), ERR

Если связанная с запросом request операция **завершена**, возвращается flag = true, а status содержит информацию о завершенной операции. Если проверяемая операция **не завершена**, возвращается flag = false, а значение status в этом случае не определено.

Неблокирующие коммуникационные операции

```
CALL MPI_COMM_RANK(comm, rank, err)
IF (rank.EQ.0) THEN
      CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, err)
**** Выполнение вычислений во время передачи сообщения ****
      CALL MPI_WAIT(request, status, err)
ELSEIF (rank.EQ.1) THEN
      CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, err)
**** Выполнение вычислений во время приема сообщения ****
     CALL MPI_WAIT(request, status, err)
END IF
```

Функция снятия запроса без ожидания завершения неблокирующей операции

```
MPI_REQUEST_FREE(REQUEST, ERR)
                 ↑        ↑
                InOut    out
INTEGER :: REQUEST, ERR
```

Неблокирующие коммуникационные операции

- Использование неблокирующих коммуникационных операций повышает безопасность с точки зрения возникновения тупиковых ситуаций, а также может увеличить скорость работы программы за счет совмещения выполнения вычислительных и коммуникационных операций.

- Неблокирующие операции используют специальный скрытый (opaque) объект "запрос обмена" (request) для связи между функциями обмена и функциями опроса их завершения.

- Если операция обмена завершена, подпрограмма проверки снимает "запрос обмена", устанавливая его в значение MPI_REQUEST_NULL. Снять запрос без ожидания завершения операции можно подпрограммой MPI_Request_free

Выполняемая проверка	Функции ожидания (блокирующие)	Функции проверки (неблокирующие)
Завершились все операции	MPI_Waitall	MPI_Testall
Завершилась по крайней мере одна операция	MPI_Waitany	MPI_Testany
Завершилась одна из списка проверяемых	MPI_Waitsome	MPI_Testsome

Коллективные операции

Коллективные операций — коммуниационные операции, в которых участвуют все процессы, связанные с некоторым коммуникатором.

- Синхронизацию всех процессов с помощью барьеров (MPI_Barrier)
- Коллективные коммуникационные операции:
  - рассылка информации от одного процесса всем остальным членам некоторой области связи (MPI_Bcast)
  - сборка (gather) распределенного по процессам массива в один массив с сохранением его в адресном пространстве выделенного (root) процесса (MPI_Gather, MPI_Gatherv)
  - сборка (gather) 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 (MPI_Alltoall, MPI_Alltoallv)
- 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)

Коллективные операции

Все коммуникационные подпрограммы, за исключением MPI_Bcast, представлены в двух вариантах:

- простой вариант, когда все части передаваемого сообщения имеют одинаковую длину и занимают смежные области в адресном пространстве процессов;
- "векторный" вариант, в котором длины блоков, и размещение данных в адресном пространстве процессов произвольны.

особенности коллективных операций

- Коллективные коммуникации не взаимодействуют с коммуникациями типа точка-точка.
- Коллективные коммуникации выполняются в режиме с блокировкой. Возврат из подпрограммы в каждом процессе происходит тогда, когда его участие в коллективной операции завершилось, однако это не означает, что другие процессы завершили операцию.
- Количество получаемых данных должно быть равно количеству посланных данных.
- Типы элементов посылаемых и получаемых сообщений должны совпадать.
- Сообщения не имеют идентификаторов.

Коллективные операции

Функция синхронизации процессов

MPI_BARRIER(COMM, ERR)

In Out

INTEGER :: COMM, ERR

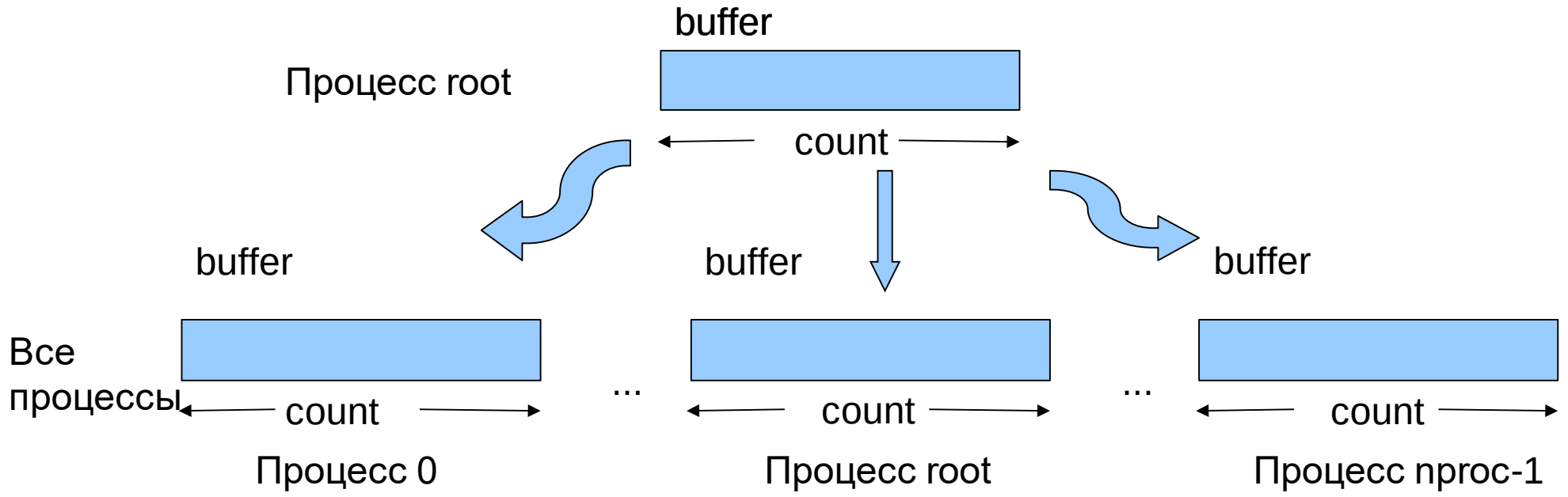
Широковещательная рассылка данных

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, ERR)

InOut In Out

<type> :: BUFFER(*)

INTEGER :: COUNT, DATATYPE, ROOT, COMM, ERR



Коллективные операции

```
IF ( MYID .eq. 0 ) THEN
    WRITE(6, *) 'ВВЕДИТЕ ПАРАМЕТР N : '
    READ(5, *) N
END IF
CALL MPI_BCAST(N, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ERR)
```

Функции сбора блоков данных от всех процессов группы

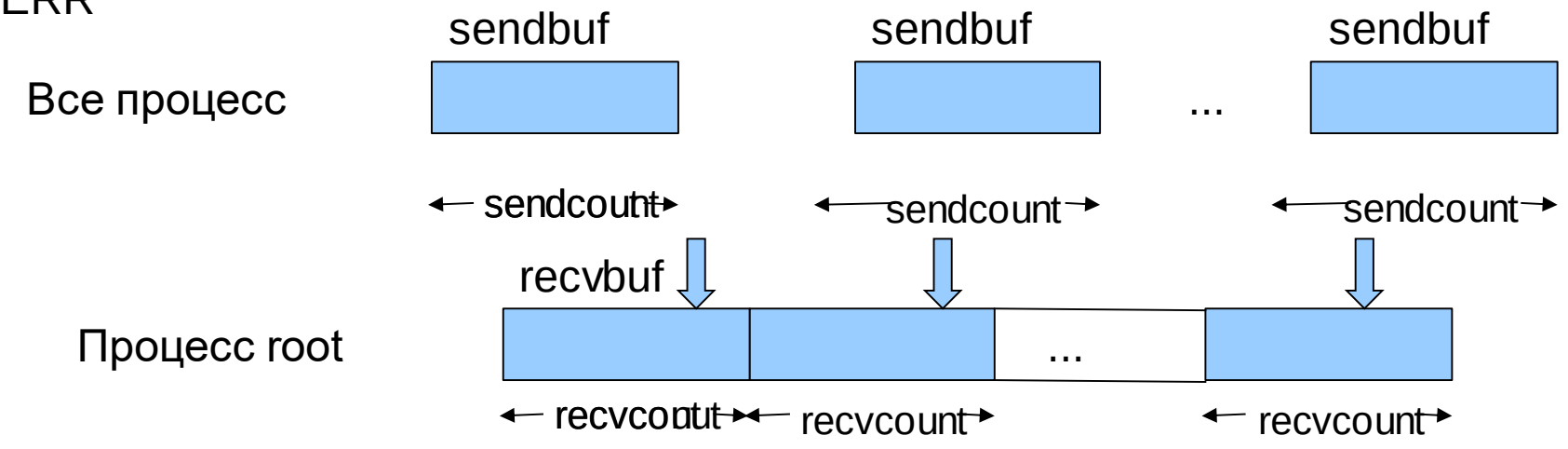
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,

In Out In

RECVTYPE, ROOT, COMM, ERR)

In Out

```
<type> :: SENDBUF(*), RECVBUF(*)
INTEGER :: SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM,
ERR
```



Коллективные операции

Функция MPI_Allgather выполняется так же, как MPI_Gather, но получателями являются все процессы группы. После завершения операции содержимое буферов приема recvbuf у всех процессов одинаково.

MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,

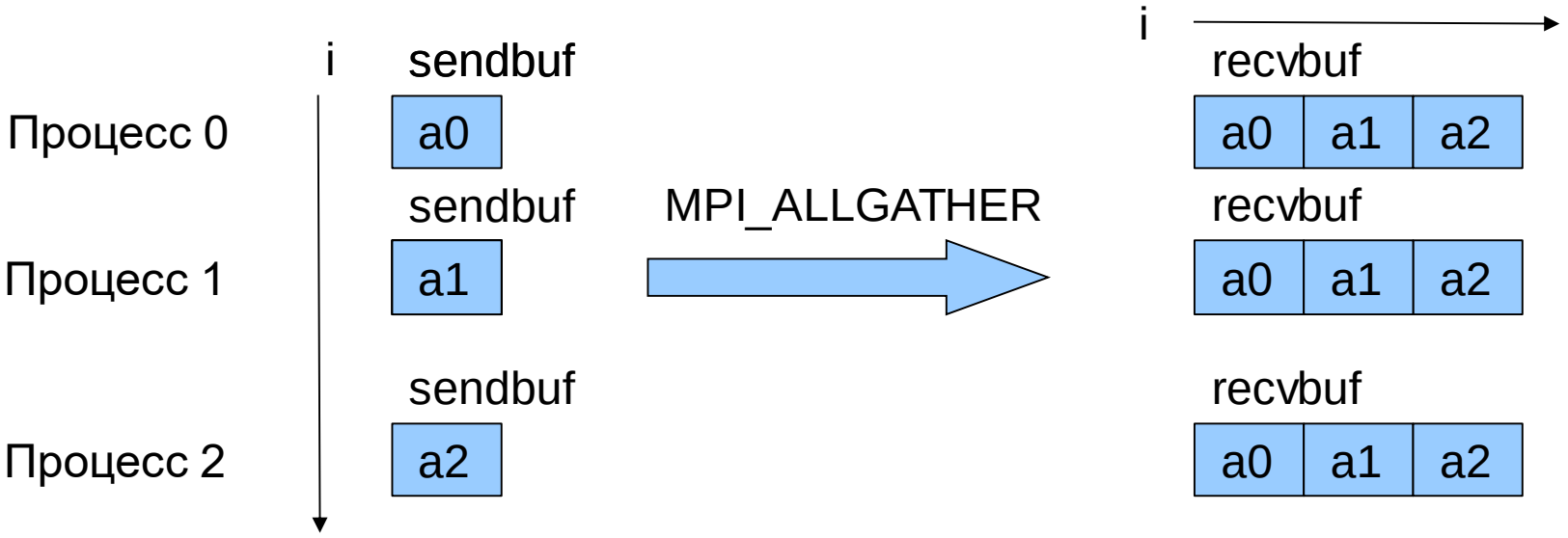
In Out

RECVCOUNT, RECVTYPE, COMM, ERR)

In Out

<type> :: SENDBUF(*), RECVBUF(*)

INTEGER :: SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, ERR



Коллективные операции

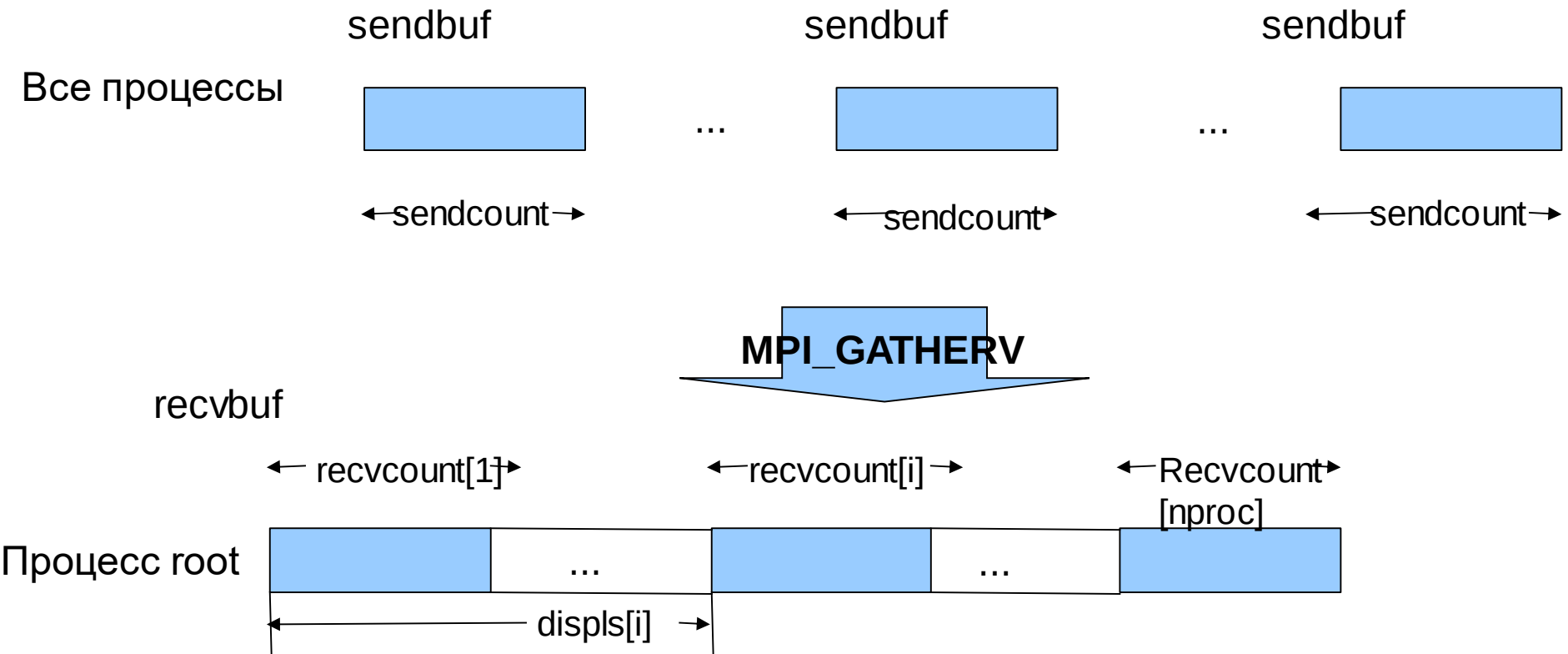
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RBUF,

In Out

RECVCOUNTS, DISPLS, RECVTYPE, ROOT, COMM, ERR)

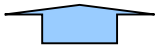
In Out

<type> :: SENDBUF(*), RBUF(*)
INTEGER :: SENDCOUNT, SENDTYPE, **RECVCOUNTS(*), DISPLS(*)**, RECVTYPE, ROOT, COMM, ERR

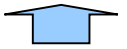


Коллективные операции

MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RBUF,

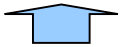


In



Out

RECVCOUNTS, DISPLS, RECVTYPE, COMM, ERR)



In

Out

<type> :: SENDBUF(*), RBUF(*)
INTEGER :: SENDCOUNT, SENDTYPE, **RECVCOUNTS(*), DISPLS(*)**, RECVTYPE, COMM, ERR

Функция MPI_Allgatherv является аналогом функции MPI_Gatherv, но сборка выполняется всеми процессами группы.

Коллективные операции

Функции распределения блоков данных по всем процессам группы

MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,

In Out

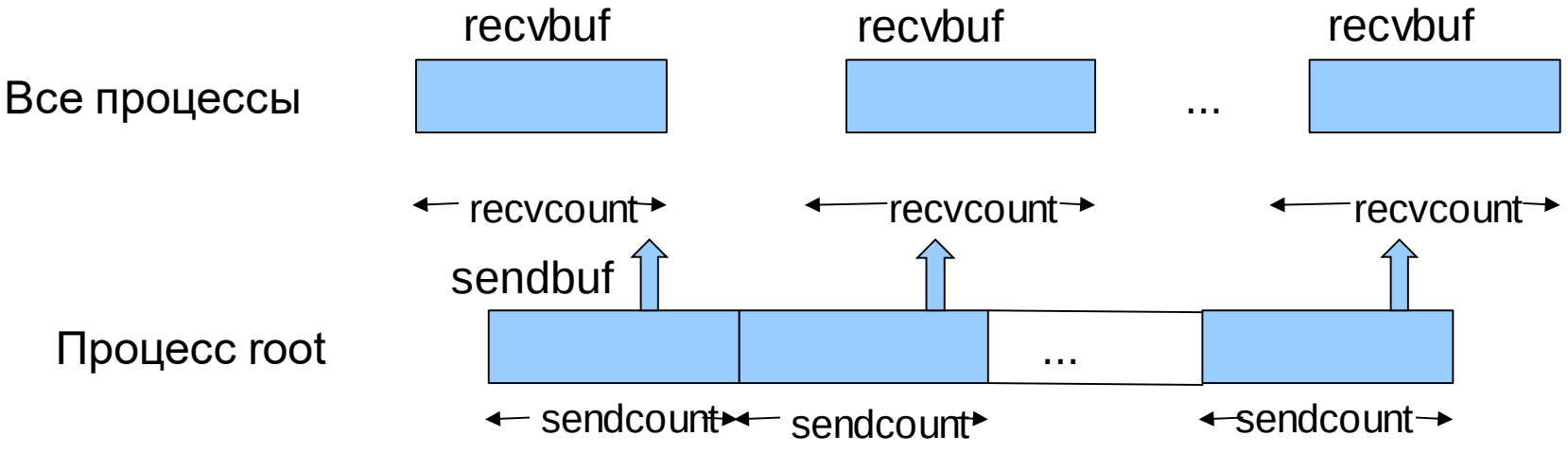
RECVCOUNT, RECVTYPE, ROOT, COMM, ERR)

In Out

<type> SENDBUF(*), RECVBUF(*)

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, ERR

Тип посылаемых элементов sendtype должен совпадать с recvtype получаемых элементов, а число посылаемых элементов sendcount должно равняться recvcount.



Коллективные операции

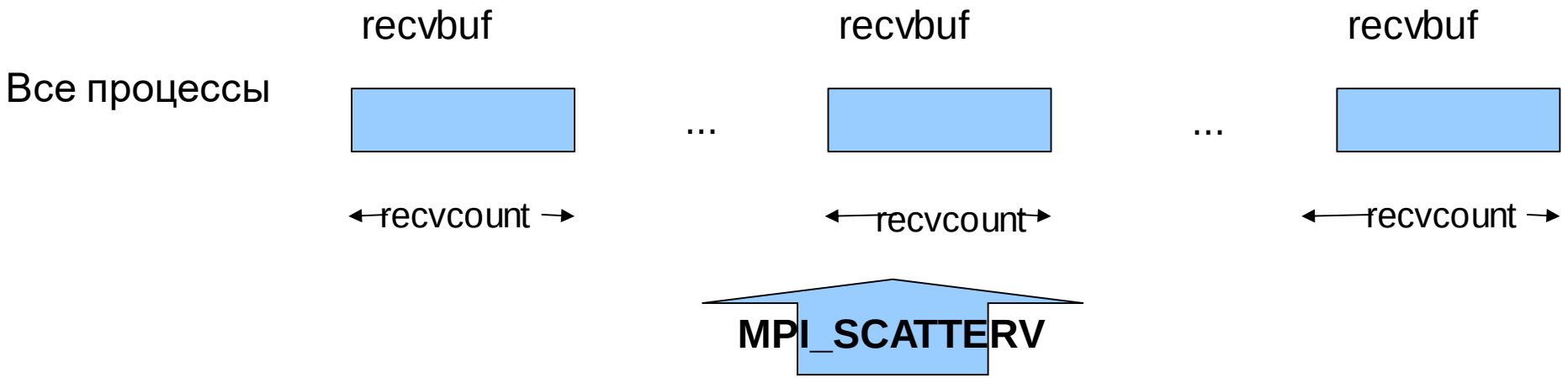
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,

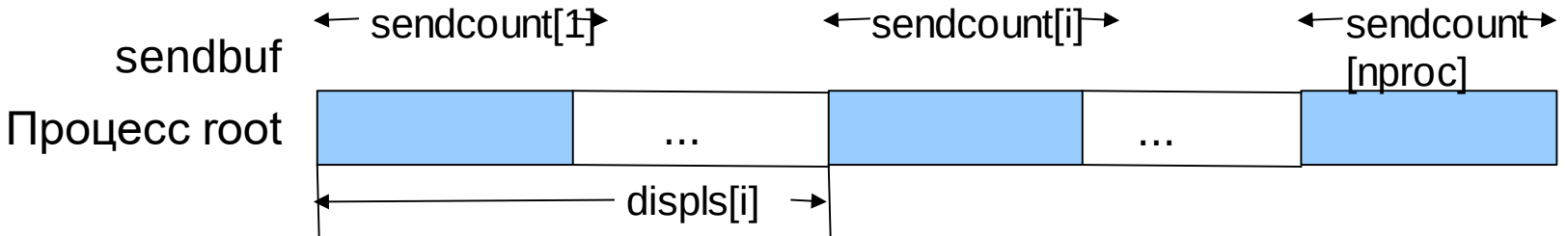
In Out

RECVCOUNT, RECVTYPE, ROOT, COMM, ERR)

In Out

<type> :: SENDBUF(*), RECVBUF(*)
INTEGER :: SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT,
RECVTYPE, ROOT, COMM, ERR





Коллективные операции

Совмещенные коллективные операции

Функция MPI_Alltoall (MPI_Alltoallv)совмещает в себе операции Scatter и Gather

MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,

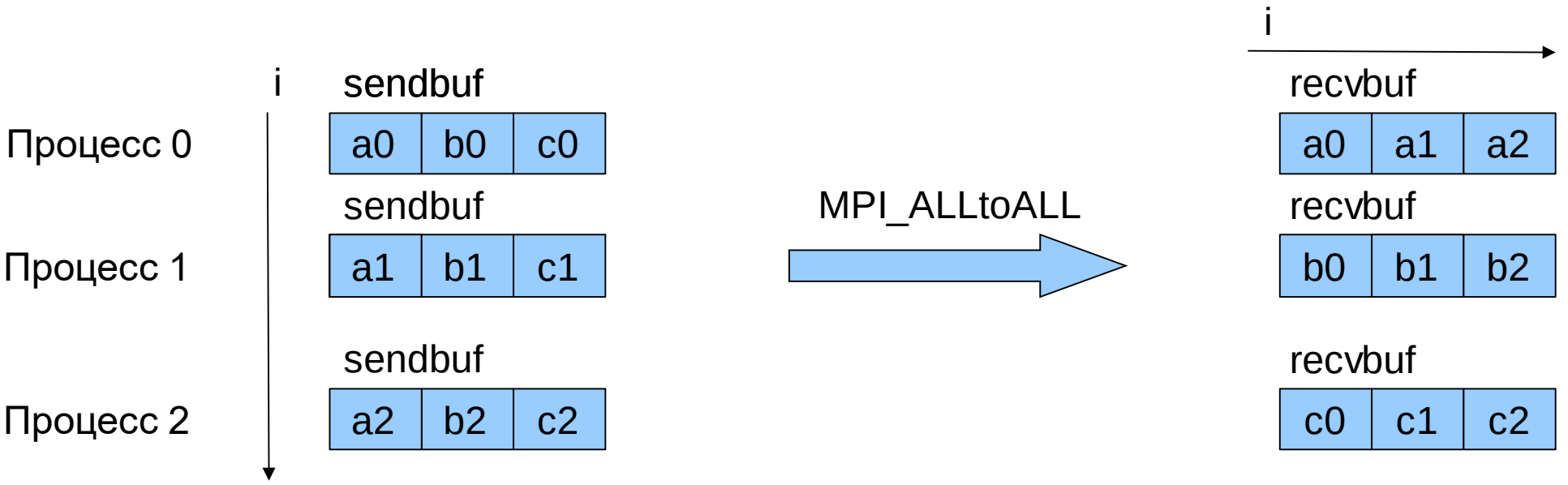
In Out

RECVCOUNT, RECVTYPE, COMM, ERR)

In Out

<type> :: SENDBUF(*), RECVBUF(*)

INTEGER ::SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, ERR



Коллективные операции

Глобальные вычислительные операции над распределенными данными

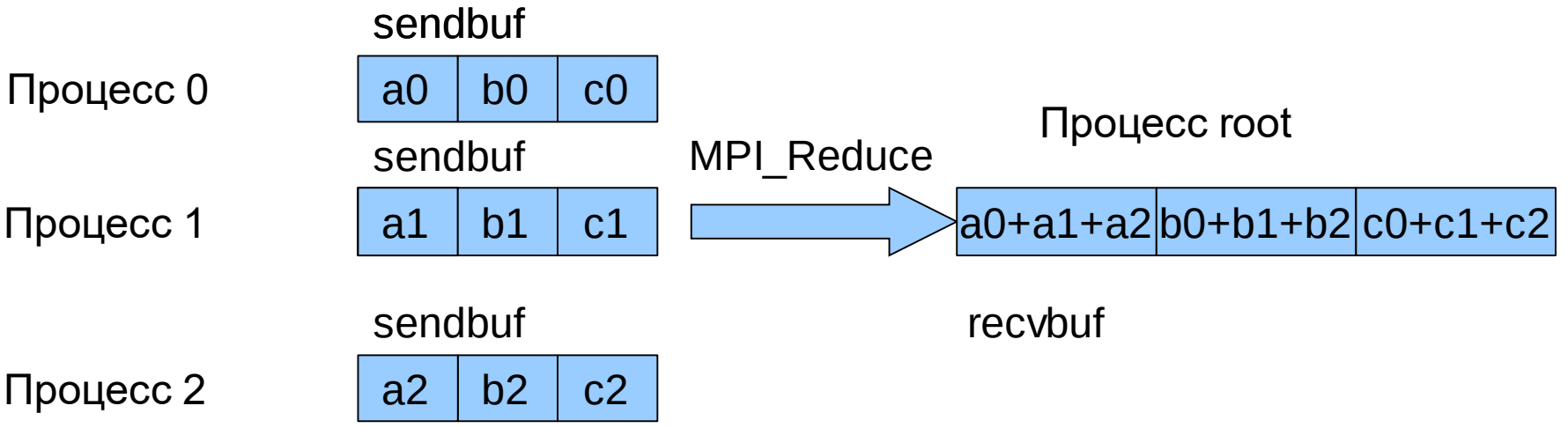
глобальные операции редукции - операции над блоками данных, распределенных по процессорам.

В общем случае операцией редукции называется операция, аргументом которой является вектор, а результатом – скалярная величина, полученная применением некоторой математической операции ко всем компонентам вектора.

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM,ERR)

In Out In Out

<type> ::SENDBUF(*), RECVBUF(*)
INTEGER ::COUNT, DATATYPE, OP, ROOT, COMM, ERR



Коллективные операции

Предопределенные операции в функциях редукции MPI

Название	операция	Разрешенные типы
MPI_MAX	Максимум	integer, Floating point
MPI_MIN	Минимум	
MPI_SUM	Сумма	integer, Floating point, Complex
MPI_PROD	Произведение	
MPI_LAND	Логическое AND	logical
MPI_LOR	Логическое OR	
MPI_LXOR	Логическое исключающее OR	
MPI_BAND	Поразрядное AND	MPI_BYTE
MPI_BOR	Поразрядное OR	
MPI_BXOR	Поразрядное исключающее OR	
MPI_MAXLOC	Максимальное значение и его индекс	MPI_2REAL, MPI_2DOUBLE_PRECISION, MPI_2INTEGER
MPI_MAXLOC	Минимальное значение и его индекс	

Коллективные операции

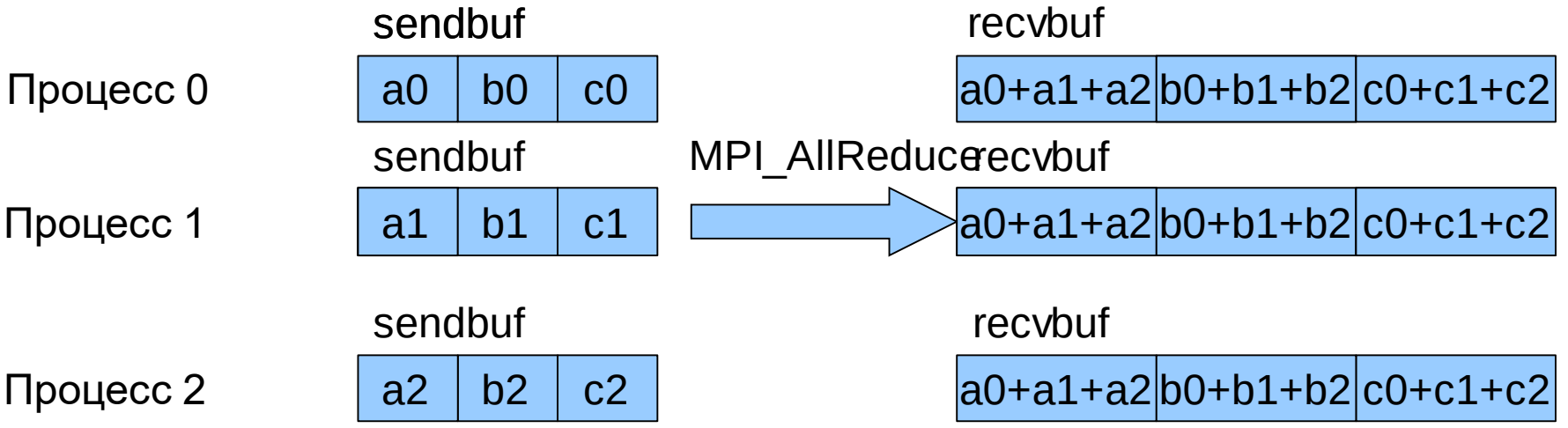
Функция MPI_Allreduce сохраняет результат редукции в адресном пространстве всех процессов

MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM,ERR)

In Out In Out

<type> ::SENDBUF(*), RECVBUF(*)

INTEGER ::COUNT, DATATYPE, OP, COMM, ERR



Коллективные операции

Функция MPI_Allreduce сохраняет результат редукции в адресном пространстве всех процессов

MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP,

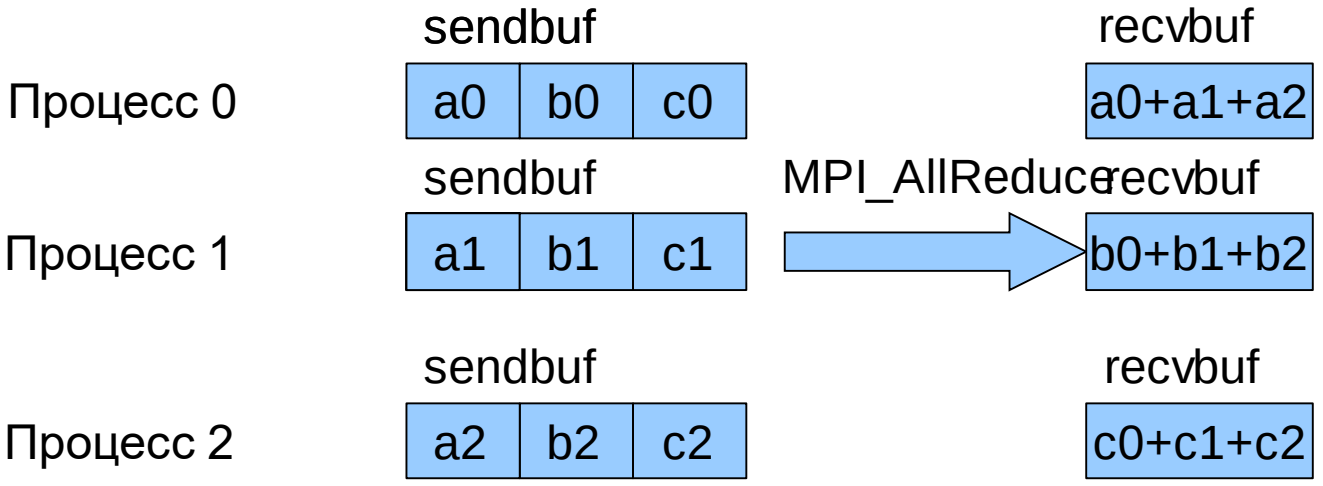
In Out In

COMM,ERR)

In Out

<type> ::SENDBUF(*), RECVBUF(*)

INTEGER ::RECVCOUNT, DATATYPE, OP, COMM, ERR



Коллективные операции

Префиксная редукция

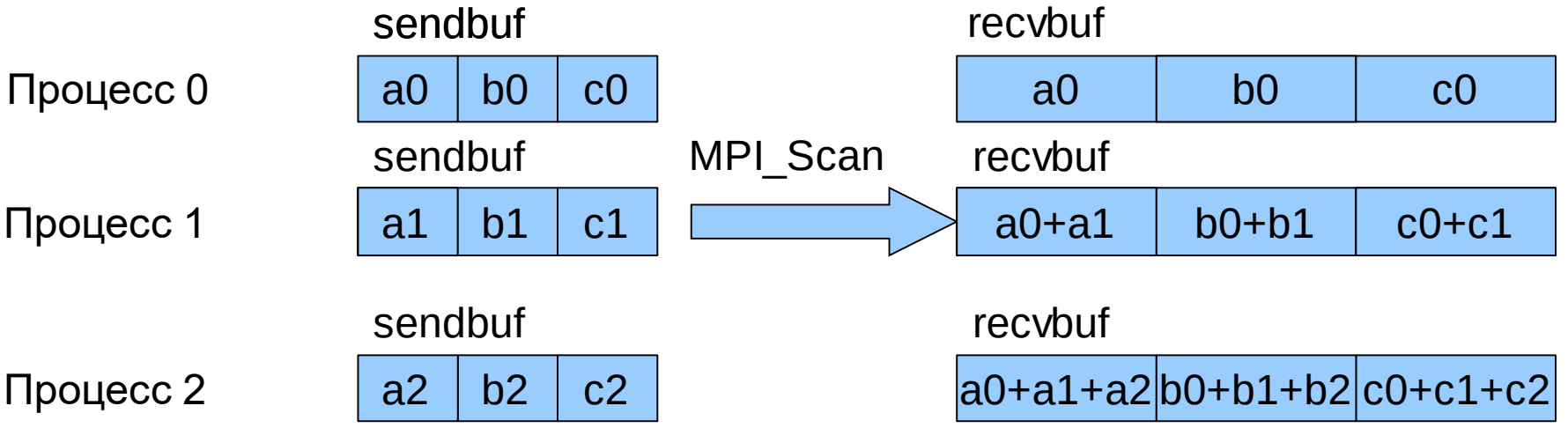
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM,ERR)

In Out In Out

<type> ::SENDBUF(*), RECVBUF(*)

INTEGER ::COUNT, DATATYPE, OP, COMM, ERR

Операция пересылает в буфер приема i-го процесса редукцию значений из входных буферов процессов с номерами 0, ... , i включительно



Производные типы данных, упакованный тип данных

Для передачи данных разных типов (например, структуры) или данных, расположенные в несмежных областях памяти (части массивов, не образующих непрерывную последовательность элементов) в MPI существует 2 механизма:

- создание производных типов для использования в коммуникационных операциях вместо предопределенных типов MPI

- пересылку упакованных данных (процесс-отправитель упаковывает пересылаемые данные перед их отправкой, а процесс-получатель распаковывает их после получения).

Производные типы данных

- Производные типы MPI не могут использоваться ни в каких других операциях, кроме коммуникационных.

- Производные типы MPI это описание расположения в памяти элементов базовых типов.

- Производный тип MPI - скрытый (opaque) объект, специфицирующий последовательность базовых типов и последовательность смещений.

- Последовательность таких пар - отображение (карта) типа:
Typemap = {(type0, disp0), ..., (typen-1, dispn-1)

Значения смещений не обязательно должны быть неотрицательными, различными и упорядоченными по возрастанию.

Производные типы данных

Определение и использование производных типов:

- Производный тип строится из предопределенных типов MPI и ранее определенных производных типов с помощью специальных функций-конструкторов MPI_Type_contiguous, MPI_Type_vector, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_hindexed, MPI_Type_struct.

- Новый производный тип регистрируется вызовом функции MPI_Type_commit. Только после регистрации новый производный тип можно использовать в коммуникационных подпрограммах и при конструировании других типов.

- Когда производный тип становится ненужным, он уничтожается функцией MPI_Type_free.

Производные типы данных

Любой тип данных в MPI имеет две характеристики: протяженность и размер, выраженные в байтах:

- Протяженность типа определяет, сколько байт переменная данного типа занимает в памяти. Эта величина может быть вычислена как адрес последней ячейки данных - адрес первой ячейки данных + длина последней ячейки данных

MPI_TYPE_EXTENT(DATATYPE, EXTENT, ERR)

INTEGER :: DATATYPE, EXTENT, ERR

- Размер типа определяет количество реально передаваемых байт в коммуникационных операциях. Эта величина равна сумме длин всех базовых элементов определяемого типа

MPI_TYPE_SIZE(DATATYPE, SIZE, ERR)

INTEGER :: DATATYPE, SIZE, ERR

Производные типы данных

MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, ERR)

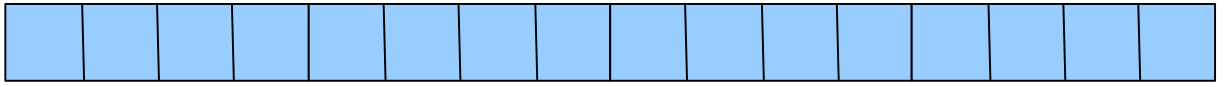
In Out

INTEGER :: COUNT, OLDTYPE, NEWTYPE, ERR

Oldtype = MPI_REAL

Count = 4

newtype



Производные типы данных

Конструктор MPI_Type_vector создает тип, элемент которого - несколько равноудаленных друг от друга блоков из одинакового числа смежных элементов базового типа.

MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, ERR)

Out

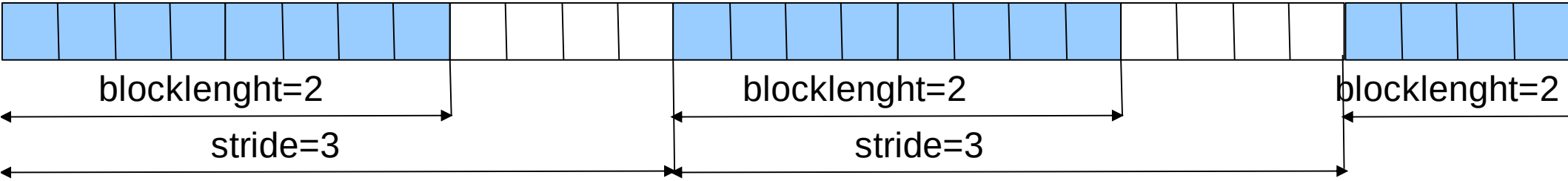
In

INTEGER :: COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, ERR

Oldtype = MPI_REAL

Count = 3, blocklenght=2, stride =3

newtype



Производные типы данных

Конструктор MPI_Type_hvector позволяя задавать произвольный шаг между началами блоков в байтах.

MPI_TYPE_PVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, ERR)

Out

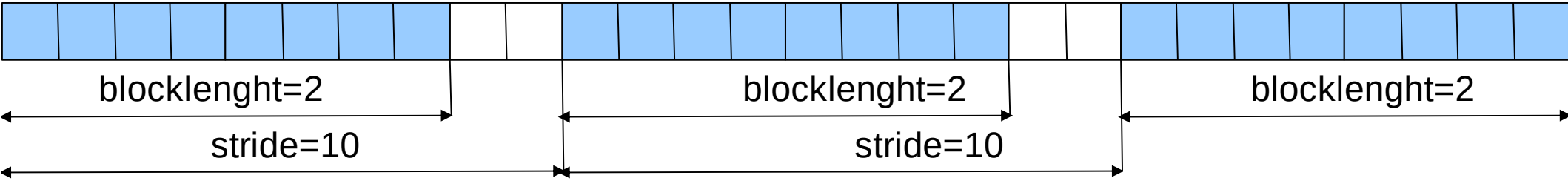
In

INTEGER :: COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, ERR

Oldtype = MPI_REAL

Count = 3, blocklenght=2, stride =10

newtype



Производные типы данных

- Конструктор типа MPI_Type_indexed создает элементы из произвольных по длине блоков с произвольным смещением блоков от начала размещения элемента. Смещения измеряются в элементах старого типа.

MPI_TYPE_INDEXED(COUNT,ARRAY_OF_BLOCKLENGTHS,ARRAY_OF_DISPLACEMENTS

In

In

Out

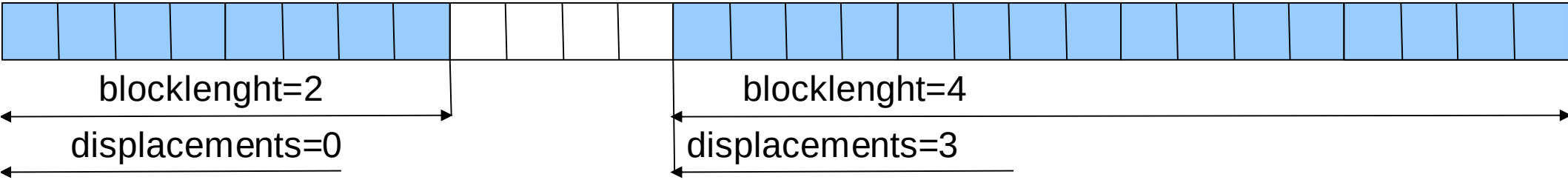
, OLDTYPE, NEWTYPE, ERR)

INTEGER :: COUNT, ARRAY_OF_BLOCKLENGTHS(*),
ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, ERR

Oldtype = MPI_REAL

Count = 2, blocklenght=(2,4) displacements =(0,3)

newtype



Производные типы данных

- Конструктор типа MPI_Type_hindexed смещения измеряются в байтах.

MPI_TYPE_INDEXED(COUNT,ARRAY_OF_BLOCKLENGTHS,ARRAY_OF_DISPLACEMENTS

In Out In

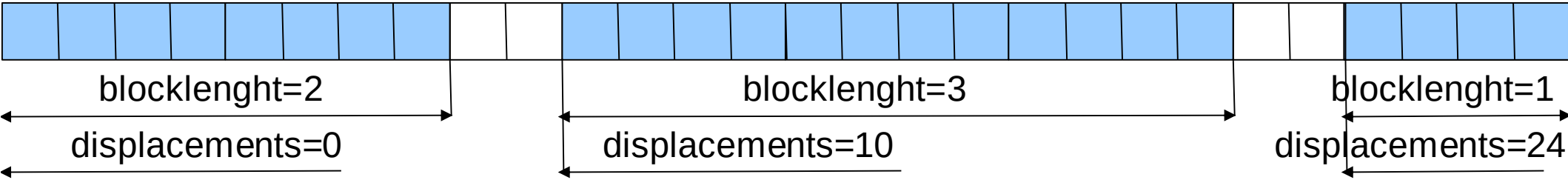
, OLDTYPE, NEWTYPE, ERR)

INTEGER :: COUNT, ARRAY_OF_BLOCKLENGTHS(*),
ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, ERR

Oldtype = MPI_REAL 

Count = 3, blocklenght=(2,3,1) displacements =(0,10,24)

newtype



Производные типы данных

Конструктор типа MPI_Type_struct – самый универсальный из всех конструкторов.

MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, ERR)

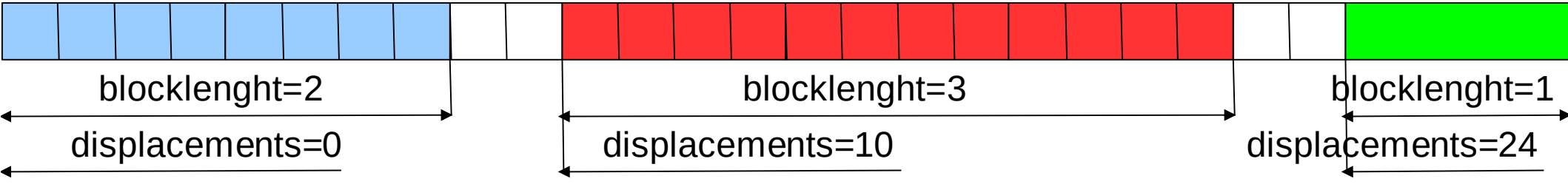
In Out In

INTEGER:: COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*), ARRAY_OF_TYPES(*), NEWTYPE, ERR

Oldtypes = (MPI_REAL, MPI_integer,MPI_CHARACTER)

Count = 3, blocklenght=(2,3,1) displacements =(0,10,24)

newtype



Производные типы данных

Функция MPI_Type_commit регистрирует созданный производный тип.

MPI_TYPE_COMMIT(DATATYPE, ERR)

In Out

INTEGER :: DATATYPE, ERR

Функция MPI_Type_free уничтожает описатель производного типа.

MPI_TYPE_FREE(DATATYPE, ERR)

In Out

INTEGER :: DATATYPE, ERR

Это не повлияет на выполняющиеся в данный момент коммуникационные операции с этим типом данных и на производные типы, которые ранее были определены через уничтоженный тип.

Функция MPI_Get_elements возвращает число элементов простых типов, содержащихся в сообщении. Если получено не целое число элементов, то она возвратит константу MPI_UNDEFINED

MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, ERR)

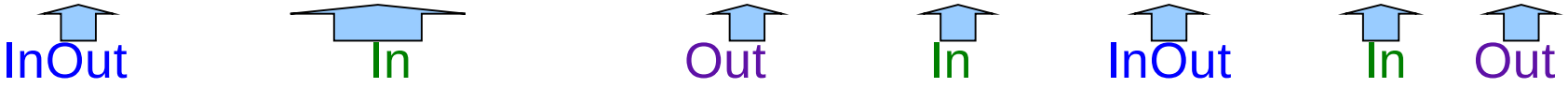
In Out

INTEGER :: STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, ERR

Передача упакованных данных

Функция MPI_Pack упаковывает элементы предопределенного или производного типа MPI, помещая их побайтное представление в выходной буфер

MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, ERR)



<type> :: INBUF(*), OUTBUF(*)

INTEGER :: INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, ERR

Функция MPI_Unpack извлекает заданное число элементов некоторого типа из побайтного представления элементов во входном массиве.

MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,DATATYPE, COMM, ERR)

<type> :: INBUF(*), OUTBUF(*)

INTEGER :: INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR

Упакованный буфер пересылается любыми коммуникационными операциями с указанием типа MPI_PACKED и **коммуникатора comm, который использовался при обращениях к функции MPI_Pack.**

Передача упакованных данных

Функция MPI_Pack_size определяет размер буфера, необходимый для упаковки некоторого количества данных типа datatype.

MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, ERR)

INTEGER :: INCOUNT, DATATYPE, COMM, SIZE, ERR

После обращения к функции параметр size будет содержать размер сообщения в байтах после его упаковки.

Передача упакованных данных

```
double precision  :: buff(100);
real              :: x, y
integer           :: position, a(2),err
call MPI_INIT(err)
call MPI_Comm_rank(MPI_COMM_WORLD, myid,err)
if (myid.eq.0) then
   ! Упаковка данных
     position = 0
     call MPI_Pack(x, 1, MPI_REAL, buff, 100, position, MPI_COMM_WORLD,err)
     call MPI_Pack(y, 1, MPI_REAL, buff, 100, position, MPI_COMM_WORLD,err)
     call MPI_Pack(a, 2, MPI_INTEGER, buff, 100, position, MPI_COMM_WORLD,err)
endif
! Рассылка упакованного сообщения
 call MPI_Bcast(position,1, MPI_INTEGER, 0, MPI_COMM_WORLD,err)
call MPI_Bcast(buff, position, MPI_PACKED, 0, MPI_COMM_WORLD,err)
! Распаковка сообщения во всех процессах
if (myid.ne. 0) then
    position = 0;
    call MPI_Unpack(buff, 100, position, x, 1, MPI_REAL, MPI_COMM_WORLD,err)
    call MPI_Unpack(buff, 100, position, y, 1, MPI_DOUBLE,MPI_COMM_WORLD,err)
    call MPI_Unpack(buff, 100, position, a, 2, MPI_INTEGER,MPI_COMM_WORLD,err)
endif
```

Работа с группами и коммуникаторами

Группа - упорядоченное множество процессов. Каждый процесс идентифицируется переменной целого типа.

Часто в приложениях возникает потребность ограничить область коммуникаций набором процессов, которые составляют подмножество исходного набора.

Существует две предопределенных группы:
MPI_GROUP_EMPTY – группа, не содержащая ни одного процесса;
MPI_GROUP_NULL – значение, возвращаемое в случае, когда группа не может быть создана.

При инициализации MPI не создается группы, соответствующей коммуникатору MPI_COMM_WORLD.

Одной и той же области связи может соответствовать несколько коммуникаторов. Они не являются тождественными и не могут участвовать во взаимном обмене сообщениями.

При инициализации MPI создается два предопределенных коммуникатора:
MPI_COMM_WORLD – описывает область связи, содержащую все процессы;
MPI_COMM_SELF – описывает область связи, состоящую из одного процесса.

Функции работы с группами

Функция определения числа процессов в группе

MPI_GROUP_SIZE(GROUP, SIZE, ERR)

INTEGER :: GROUP, SIZE, ERR

Если group = MPI_GROUP_EMPTY, тогда size = 0

Функция определения номера процесса в группе

MPI_GROUP_RANK(GROUP, RANK, ERR)

INTEGER GROUP, RANK, ERR

Функция установки соответствия между номерами процессов в двух группах

MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, ERR)

INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), ERR

n – число процессов, для которых устанавливается соответствие

Если процесс во второй группе отсутствует, то для него устанавливается значение MPI_UNDEFINED

Функции работы с группами

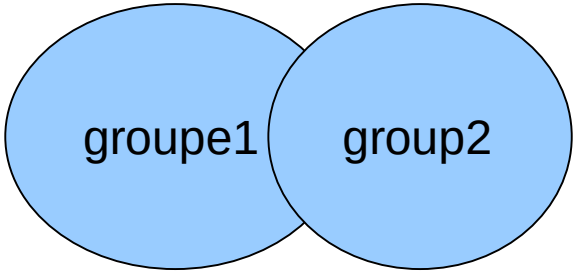
Функция создания группы с помощью коммуникатора

MPI_COMM_GROUP(COMM, GROUP, ERR)

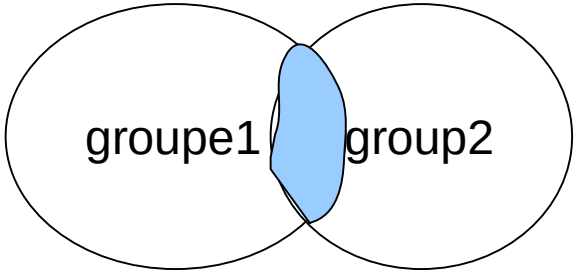
INTEGER COMM, GROUP, ERR

MPI_GROUP_UNION (GROUP1, GROUP2, NEWGROUP, ERR)
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, ERR)
MPI_GROUP_DIFFERENCE (GROUP1, GROUP2, NEWGROUP, ERR)

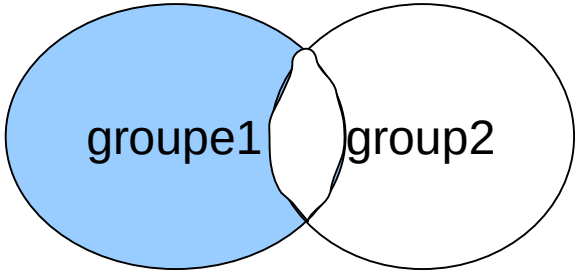
INTEGER ::GROUP1, GROUP2,NEWGROUP, ERR



Union (объединение)



Intersection (пересечеие)



Difference (дополнение)

Функции работы с группами

MPI_GROUP_INCL (GROUP, N, RANKS, NEWGROUP, ERR)
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, ERR)

INTEGER :: GROUP, N, RANKS(*), NEWGROUP, ERR

MPI_Group_incl создает новую группу, из процессов group, **перечисленных** в массиве ranks.

MPI_Group_excl создает новую группу из процессов group, которые **не перечислены** в массиве ranks.

Процессы упорядочиваются, как в группе group. Каждый элемент ranks должен иметь корректный номер в group, и среди них **не должно** быть совпадающих.

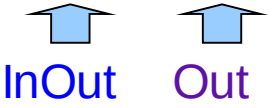
MPI_GROUP_RANGE_INCL (GROUP, N, RANGES, NEWGROUP, ERR)
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, ERR)

INTEGER :: GROUP, N, RANGES(3,*), NEWGROUP, ERR

RANGES(3,*) - тройки чисел: нижняя граница, верхняя граница, шаг

Уничтожение созданных групп

MPI_GROUP_FREE(GROUP, ERR)



INTEGER :: GROUP, ERR

Функции работы с коммуникаторами

MPI_COMM_RANK, MPI_COMM_SIZE

сравнение двух коммуникаторов

MPI_COMM_COMPARE(COMM1, COMM2, RESULT, ERR)

INTEGER :: COMM1, COMM2, RESULT, ERR

RESULT:

MPI_IDENT – коммуникаторы идентичны, представляют один и тот же объект;

MPI_CONGRUENT – коммуникаторы конгруэнтны, две области связи с одними и теми же атрибутами группы;

MPI_SIMILAR – коммуникаторы подобны, группы содержат одни и те же процессы, но другое упорядочивание;

MPI_UNEQUAL – во всех других случаях.

Функция дублирования коммуникатора

MPI_COMM_DUP(COMM, NEWCOMM, ERR)

INTEGER :: COMM, NEWCOMM, ERR

Функции работы с коммуникаторами

Функция создания коммуникатора

MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, ERR)

INTEGER :: COMM, GROUP, NEWCOMM, ERR

Для процессов, которые не являются членами группы, возвращается значение MPI_COMM_NULL. Функция возвращает код ошибки, если группа group **не является** подгруппой родительского коммуникатора.

Функция расщепления коммуникатора

MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, ERR)

INTEGER:: COMM, COLOR, KEY, NEWCOMM, ERR

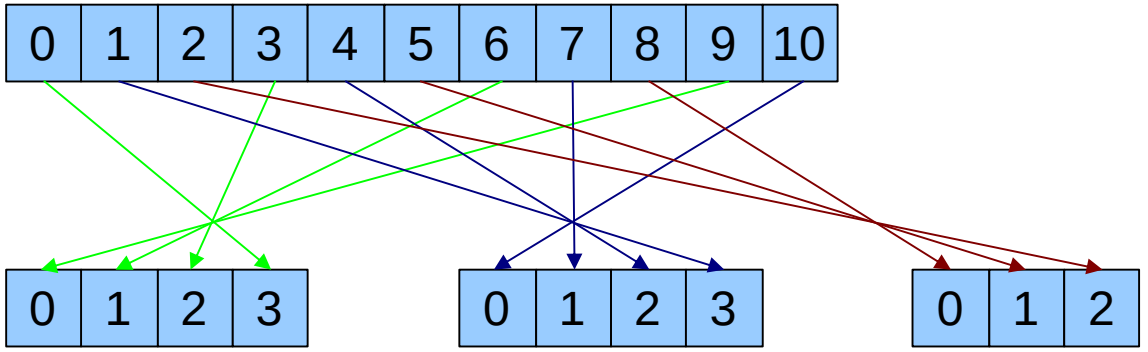
Функция уничтожения коммуникатора

MPI_COMM_FREE(COMM, ERR)

INTEGER :: COMM, ERR

Функции работы с коммуникаторами

```
integer              :: newcomm,err,myid
call MPI_INIT(err)
call MPI_Comm_rank(MPI_COMM_WORLD, myid,err)
call MPI_COMM_SPLIT(MPI_COMM_WORLD,mod(myid,3), nproc-myid, NEWCOMM, ERR)
```



mod(myid,3)=0 mod(myid,3)=1 mod(myid,3)=2

Топология процессов

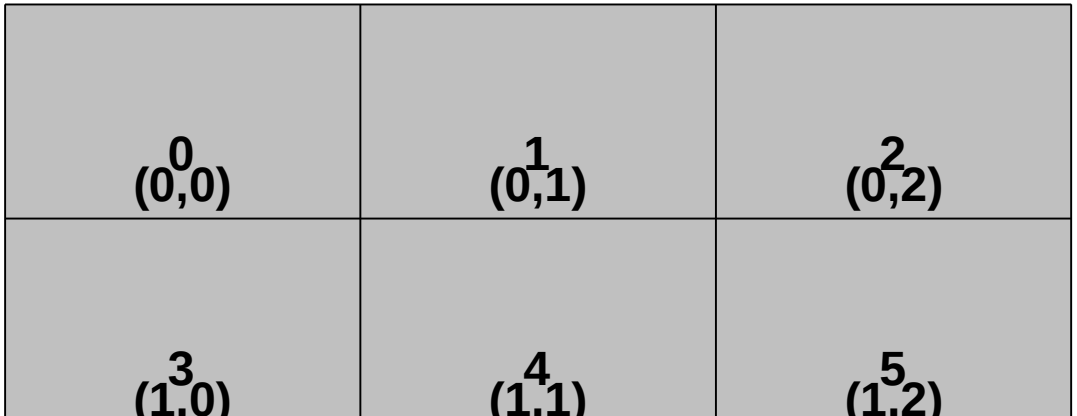
Топология процессов — альтернативный способ нумерации процессов

Топология процессов это один из необязательных атрибутов коммуникатора.
По умолчанию предполагается линейная топология, в которой процессы пронумерованы в диапазоне от 0 до nproc-1

Задание топологии позволяет учесть реальную топологию коммуникационной среды, или алгоритм задачи.

MPI поддерживает 2 топологии: топологию графов (узлы являются процессами, а грани – каналами связи между процессами) и декартову топологию.

В MPI используется row-major нумерация процессов, т.е. используется нумерация вдоль строки.



Декатрова топология

декартова топология - обобщение линейной и матричной топологий на произвольное число измерений

Функция создания коммуникатора с декартовой топологией

MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, ERR)


```
INTEGER :: COMM_OLD, NDIMS, DIMS(*), COMM_CART, ERR
LOGICAL :: PERIODS(*), REORDER
```

ndims – число измерений;

dims – массив размера ndims, в котором задается число процессов вдоль каждого измерения;

periods – логический массив размера ndims для задания граничных условий (true – периодические, false – непериодические)

reorder – логическая переменная указывает, производить перенумерацию процессов (true) или нет (false);

если какие-то процессы не попадают в новую группу, то для них возвращается результат MPI_COMM_NULL.

В случае, когда размеры заказываемой сетки больше имеющегося в группе числа процессов, то функция завершается аварийно.

Декартова топология

Функция определения оптимальной конфигурации сетки

MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, ERR)

INTEGER :: NNODES, NDIMS, DIMS(*), ERR

nnodes – общее число узлов в сетке;

ndims – число измерений;

dims – массив целого типа размерности ndims, в который помещается рекомендуемое число процессов вдоль каждого измерения.

Вычисляются только те компоненты dims[i], которые были равны 0.

dims перед вызовом	Вызов процедуры	dims после вызова
(0, 0, 0)	MPI_DIMS_CREATE(6, 3, dims, err)	(3, 2, 1)
(0, 0, 0)	MPI_DIMS_CREATE(7, 3, dims, err)	(7, 1, 1)
(0, 3, 0)	MPI_DIMS_CREATE(6, 3, dims, err)	(2, 3, 1)
(0, 3, 0)	MPI_DIMS_CREATE(7, 3, dims, err)	ошибка

Декатрова топология

Функция опроса числа измерений декартовой топологии

MPI_CARTDIM_GET(COMM, NDIMS, ERR)

INTEGER :: COMM, NDIMS, ERR

Получение более детальной информации о коммуникаторе

MPI_CART_GET(COMM, NDIMS, DIMS, PERIODS, COORDS, ERR)

INTEGER :: COMM, NDIMS, DIMS(*), COORDS(*), ERR

LOGICAL :: PERIODS(*)

Функция получения идентификатора процесса по его координатам

MPI_CART_RANK(COMM, COORDS, RANK, ERR)

INTEGER :: COMM, COORDS(*), RANK, ERR

Функция определения координат процесса по его идентификатору

MPI_CART_COORDS(COMM, RANK, NDIMS, COORDS, ERR)

INTEGER :: COMM, RANK, NDIMS, COORDS(*), ERR

Декатрова топология

Функция сдвига данных

MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, ERR)

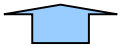
INTEGER :: COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, ERR

RANK_SOURCE и RANK_DEST затем могут быть использованы в коммуникационных операциях

В зависимости от граничных условий сдвиг может быть либо циклический, либо для граничных процессов возвращается MPI_PROC_NULL для переменной rank_source, или rank_dest.

Функция выделения подпространства в декартовой топологии

MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, ERR)



INTEGER :: COMM, NEWCOMM, ERR

LOGICAL :: REMAIN_DIMS(*)

имеется декартова решетка 2×3×4

remain_dims (true, false, true)

MPI_Cart_sub создаст три коммуникатора с топологией 2×4.

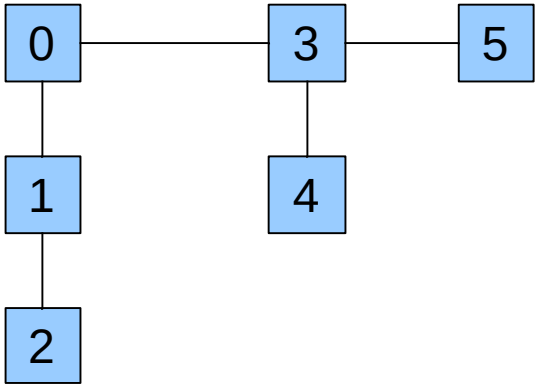
Топология графа

Функция создания коммуникатора с топологией графа

MPI_GRAPH_CREATE(COMM, NNODES, INDEX, EDGES, REORDER,COMM_GRAPH, ERR)

In Out

INTEGER :: COMM, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, ERR
LOGICAL :: REORDER



процесс	соседи
0	1,3
1	0,2
2	1
3	0,4,5
4	3
5	3

Nnodes = 6 — число вершин графа
Index = (2,4,5,8,9,10) — суммарное количество вершин
Edges = (1,3,0,2,1,0,4,5,3,3) - упорядоченный список номеров процессов-соседей

Топология графа

Определение числа соседей процесса

MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, ERR)

INTEGER :: COMM, RANK, NNEIGHBORS, ERR

Определение рангов соседей

MPI_GRAPH_NEIGHBORS(COMM, RANK, MAX, NEIGHBORS, ERR)

INTEGER :: COMM, RANK, MAX, NEIGHBORS(*), ERR

Определение числа вершин и ребер

MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, ERR)

INTEGER :: COMM, NNODES, NEDGES, ERR

Определение информации о топологии графа

MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, ERR)

INTEGER :: COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), ERR

Топология процессов

определение топологии коммуникатора

MPI_TOPO_TEST(COMM, STATUS, ERR)

INTEGER :: COMM, STATUS, ERR

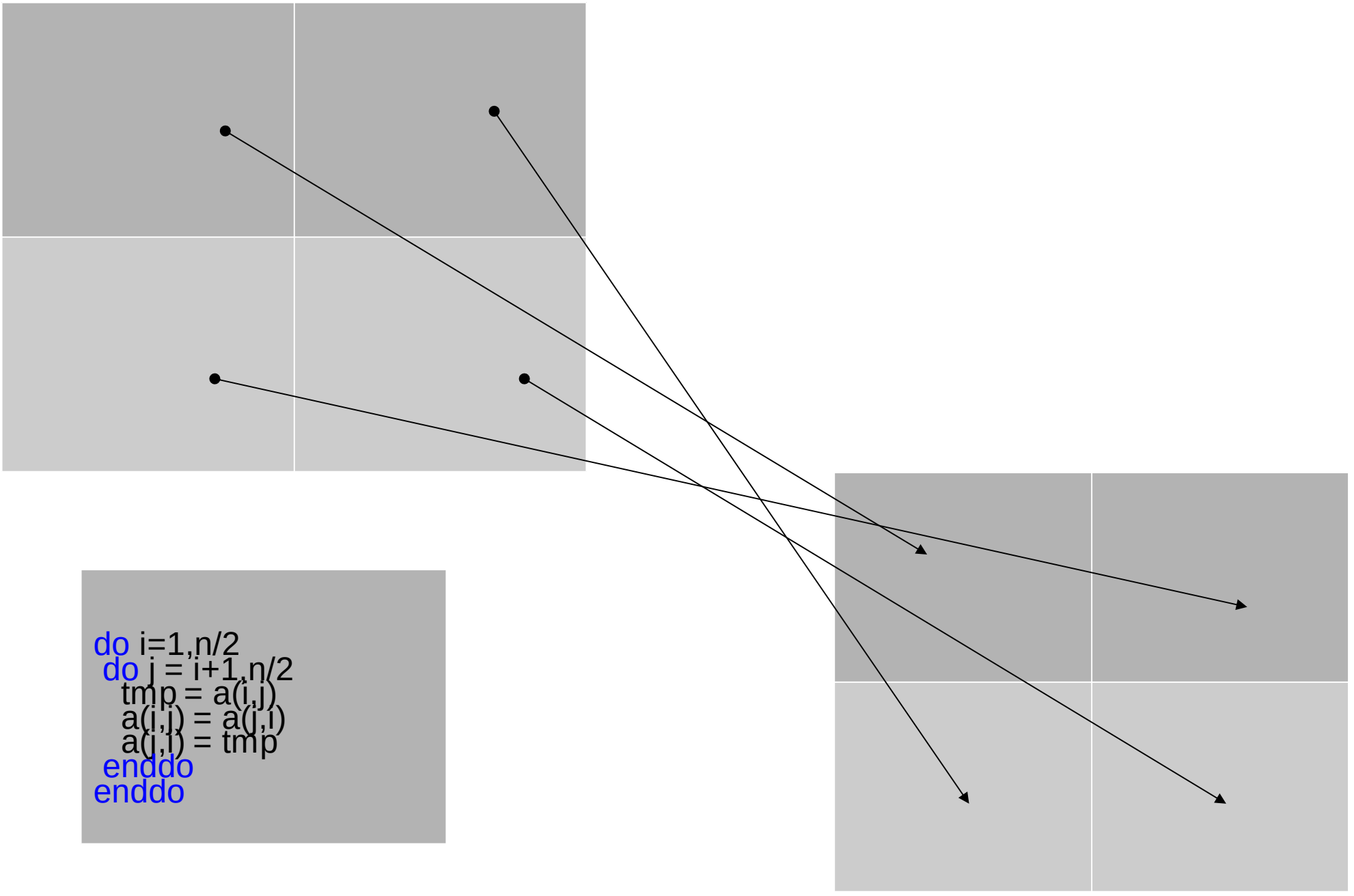
Status:

MPI_GRAPH – топология графа;

MPI_CART – декартова топология;

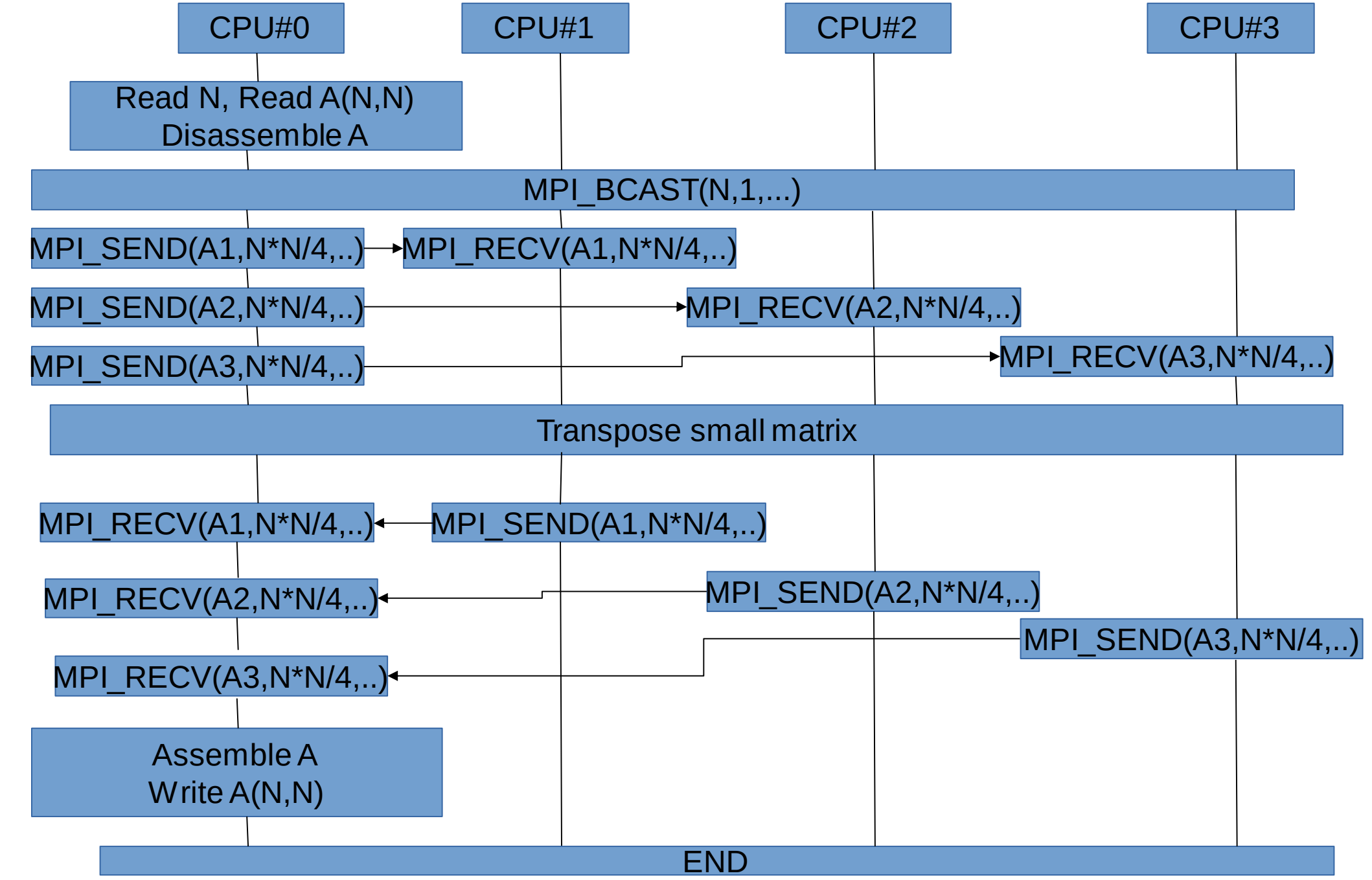
MPI_UNDEFINED – топология не задана

Транспонирование матрицы



```
do i=1,n/2
 do j = i+1,n/2
   tmp = a(i,j)
   a(i,j) = a(j,i)
   a(j,i) = tmp
 enddo
enddo
```

Транспонирование матрицы



Вычисление числа π

$$\operatorname{arctg}\ x = x - \frac{x^3}{3} + \frac{x^5}{5} - \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1}$$

$$\operatorname{arctg}\ x = \frac{x}{1+x^2} \sum_{n=0}^{\infty} \prod_{k=1}^{n} \frac{2kx^2}{(2k+1)(1+x^2)}$$

arctg(1)=π/4 $$\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

```
Pi=0.d0
do i=1,n
  pi=pi+(-1.d0)**i/(2.d0*i+1.d0)
enddo
pi = 4.d0*pi
```

```
mypi=0.d0
do i=nmin,nmax
  mypi=mypi+(-1.d0)**i/(2.d0*i+1.d0)
Enddo
call MPI_reduce(mypi,pi,1,mpi_double_precision,
mpi_sum,0,mpi_comm_world,err)
pi = 4.d0*pi
```

n=15

CPU#	0	1	2	3
nproc	4	4	4	4
nmin	1	5	9	13
nmax	4	8	12	15

nmin = f(myid,nproc,n) ?
nmax = f(myid,nproc,n) ?

Решение СЛАУ методом Якоби

Для решения СЛАУ приведем уравнение $A\vec{x} = \vec{b}$

к виду $\vec{x} = B\vec{x} + \vec{g}$ где:

$$B = E - D^{-1}A = D^{-1}(D - A), \quad \vec{g} = D^{-1}\vec{b};$$

$$B = -D^{-1}(L + U) = -D^{-1}(A - D), \quad \vec{g} = D^{-1}\vec{b}$$

$$D_{ii}^{-1} = 1/D_{ii}, D_{ii} \neq 0, i = 1, 2, ..., n \quad ;$$

D - матрица, у которой на главной диагонали стоят соответствующие элементы матрицы A, а все остальные нули; U и L верхняя и нижняя треугольные части A, на главной диагонали которых нули, E — единичная матрица.

Итерационная процедура нахождения решения

$$\vec{x}^{(k+1)} = B\vec{x}^{(k)} + \vec{g},$$

Условие остановки: $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| < \varepsilon$

Решение уравнения Лапласа методом Якоби

Элиптическое уравнение: Au=f, где A — элиптический оператор, u- неизвестная функция, а f — известная функция пространственных координат (например ур. Пуассона для потенциала электростатического поля: $\Delta\varphi=-\rho/\varepsilon_0$).
Краевая задача — дифференциальное уравнение (система дифференциальных уравнений) с заданными линейными соотношениями между значениями искомых функций на начале и конце интервала интегрирования.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Стационарное распределение температуры в тонкой пластине

Метод Якоби:

$$t^{(k+1)}_{i,j}=(t^{(k)}_{i+1,j}+t^{(k)}_{i-1,j}+t^{(k)}_{i,j+1}+t^{(k)}_{i,j-1})/4-h^2f_{i,j}/4$$

0

Условие остановки: $||\mathbf{t}^{(k+1)}-\mathbf{t}^{(k)}||<\varepsilon$