Летняя школа по параллельному программированию
ИВМиМГ, НГУ, НГТУ
лето 2012

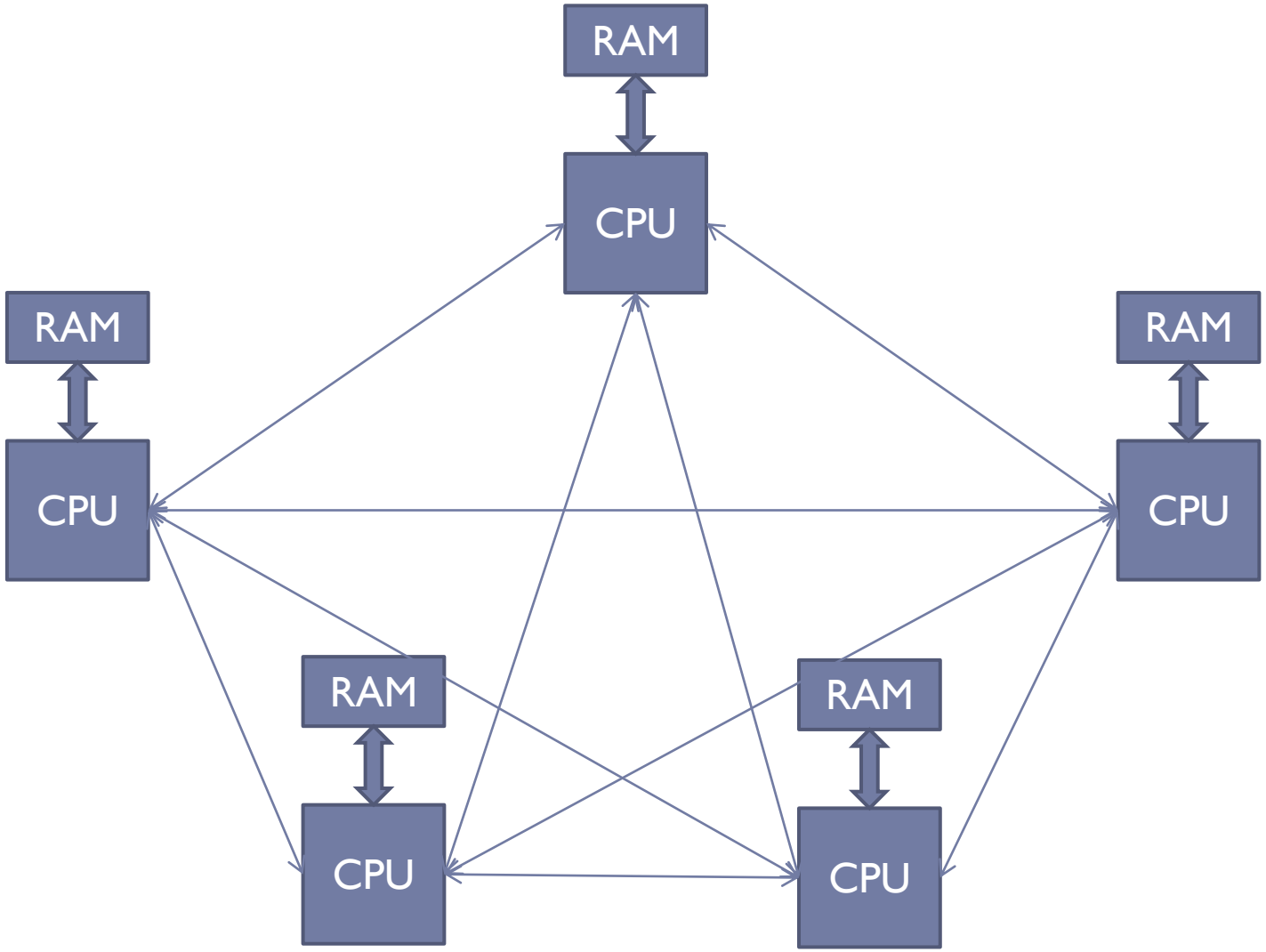
Основы MPI

Анна Черникова

План

1. Модель взаимодействия процессов
2. MPI
3. Актуальность MPI
4. Стандарты MPI
5. Простейшая программа на C
6. Компиляция и запуск
7. Функции отправки и приема сообщений
8. Пример программы
9. Соответствие типов
10. Пример. Портфель задач.
11. Пример. Модель клиенты-сервер.
12. Пример. Параллельная сортировка массива.
13. Коллективные взаимодействия
14. Пример программы
15. Параллельное умножение матриц в MPI
16. Литература
17. Практикум

Модель взаимодействия процессов



Модель взаимодействия процессов

- У каждого процесса собственная память
- Виртуальная топология – полный граф (по умолчанию)
- Процессы взаимодействуют посредством отправки и приема сообщений
- Эффективная реализация на MPP (Massive-Parallel Processing)

MPI

- MPI (Message Passing Interface) – программный интерфейс (API) для передачи информации между взаимодействующими процессами.
- Стандартизацией занимается MPI Forum
- Стандарты: MPI 1.1, MPI 2.0, MPI 2.1
- MPICH – самая распространенная бесплатная реализация, библиотека mpi.h

Актуальность MPI

- Актуальность MPI обусловлена тем, что интерфейс позволяет скрыть от пользователя отображение логической структуры программы на аппаратные ресурсы и воспринимать сложную, возможно неоднородную среду, как единый вычислительный ресурс.

Стандарты MPI

В MPI 1.1:

1. передача и получение сообщений между отдельными процессами;
2. коллективные взаимодействия процессов;
3. взаимодействия в группах процессов;
4. реализация топологий процессов;

В MPI 2.0 :

1. динамическое порождение процессов и управление процессами;
2. односторонние коммуникации (Get/Put);
3. параллельный ввод и вывод;
4. расширенные коллективные операции (процессы могут выполнять коллективные операции не только внутри одного коммуникатора, но и в рамках нескольких коммуникаторов).

Простейшая программа на С

```
int main(int argc, char** argv)
{
      int rank, size;
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);

      printf(“My rank is %d\n”, rank);
      printf(“Hello, World!\n”);

      MPI_Finalize();
      return 0;
}
```

Простейшая программа на С

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

первая функция MPI, которая должна быть вызвана. Параметры функции:

argc – указатель на количество параметров командной строки

argv – параметры командной строки, применяемые для инициализации виртуальной среды выполнения MPI-программы

```
int MPI_Finalize(void)
```

последняя вызываемая функция MPI

Компиляция и запуск

Компиляция:

```
mpicc file.c -o file.out -mpi.h
```

Запуск:

```
mpirun –np 3 ./file.out
```

```
int main(int argc, char** argv)
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(…);
  MPI_Comm_size(…);

  printf(“My rank is %d\n”, rank);
  printf(“Hello, World!\n”);

  MPI_Finalize();
  return 0;
}
```

```
int main(int argc, char** argv)
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(…);
  MPI_Comm_size(…);

  printf(“My rank is %d\n”, rank);
  printf(“Hello, World!\n”);

  MPI_Finalize();
  return 0;
}
```

```
int main(int argc, char** argv)
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(…);
  MPI_Comm_size(…);

  printf(“My rank is %d\n”, rank);
  printf(“Hello, World!\n”);

  MPI_Finalize();
  return 0;
}
```

```
My rank is 0
Hello, World!
My rank is 1
Hello, World!
My rank is 2
Hello, World!
```

Функции отправки и приема сообщений

Взаимодействие MPI-процессов происходит посредством передачи сообщений. Для парного взаимодействия необходимо, чтобы один процесс передал сообщение, а второй его принял. Для передачи сообщения в MPI разработана функция

```
int MPI_Send(buf, count, datatype, dest, tag, comm)
```

void* buf – адрес передаваемого буфера (откуда взять)

int count – количество передаваемых данных

MPI_Datatype datatype – тип передаваемых данных

int dest – номер процесса-получателя сообщения

int tag – тэг передаваемого сообщения

MPI_Comm comm – коммуникатор группы

Функция MPI_Send() осуществляет асинхронную блокированную передачу данных. Асинхронная неблокированная передача осуществляется посредством функции

```
int MPI_Isend(buf, count, type, dest, tag, comm, request)
```

Функции отправки и приема сообщений

Для получения сообщения в MPI разработана функция

```
int MPI_Recv(buf, count, type, source, tag, comm, status)
```

void* buf – адрес получаемого буфера (куда положить)

int count – количество данных

MPI_Datatype datatype – тип передаваемых данных

int source – номер процесса-получателя

int tag – тэг передаваемого сообщения

MPI_Comm comm – коммуникатор группы

MPI_Status status – статус сообщения

Неблокированный прием данных осуществляется посредством функции

```
int MPI_Irecv(buf, count, type, source, tag, comm, request)
```

Функции отправки и приема сообщений

```
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)
```

Функция MPI_Sendrecv() осуществляет блокированную асинхронную передачу данных от **source** к **dest**. Эта операция весьма полезна для выполнения сдвига по цепи процессов.

Пример программы

```
int main(int argc, char** argv)
{
        int rank, size;
        int a, b;
        a = 5, b = 0;
        int source,   dest;
        source = random()%size,  dest = random()%size;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Status st;
        if(rank = source)
                MPI_Send(&a, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        if(rank = dest)
                MPI_Recv(&b, 1, MPI_INT, source, 0, MPI_COMM_WORLD, st);
        MPI_Finalize();
        return 0;
}
```

Соответствие типов

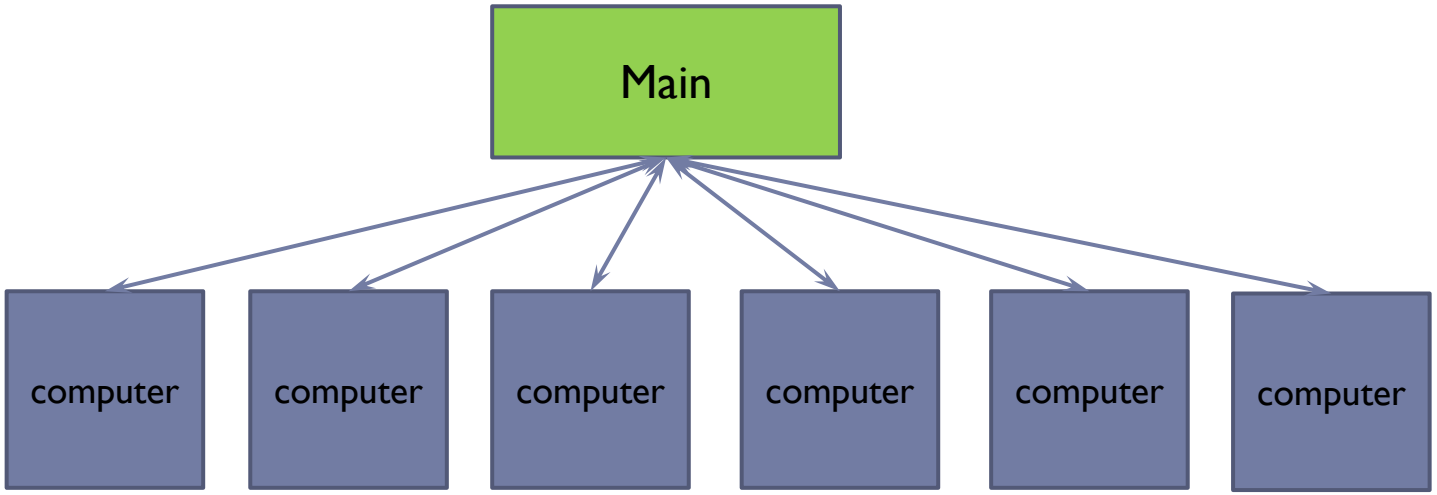
MPI_Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	
MPI_PACKED	

Соответствие типов

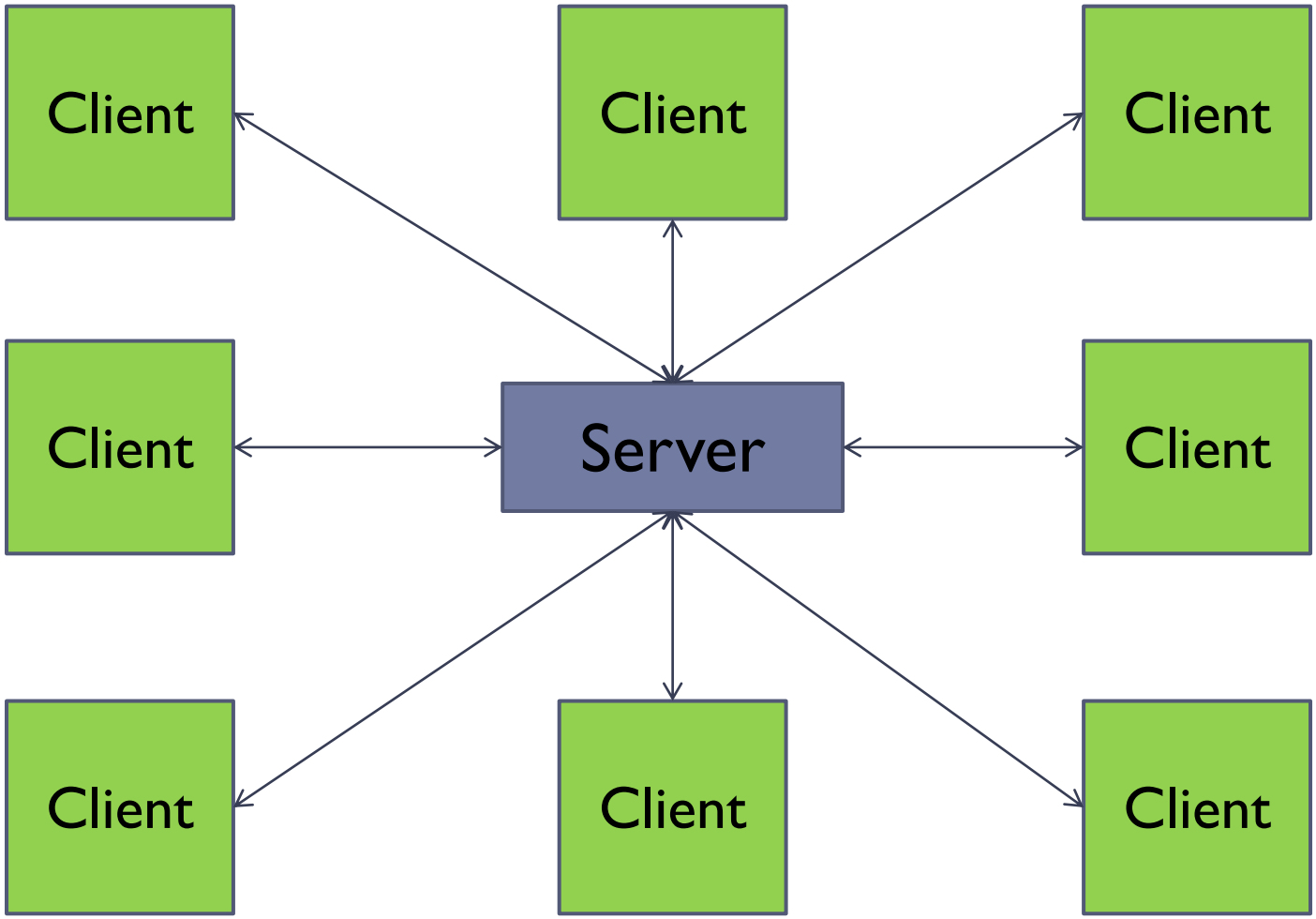
MPI_Datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	
MPI_PACKED	

Пример. Портфель задач.

Имеется один главный узел и несколько вычислительных узлов. Портфель задач (одна задача, разбитая на относительно независимые подзадачи) хранятся на главном узле. Главный узел распределяет работу между вычислительными узлами, вычислительная работа на главном узле сведена к минимуму, его основная работа – распределять работу на вычислительные узлы.

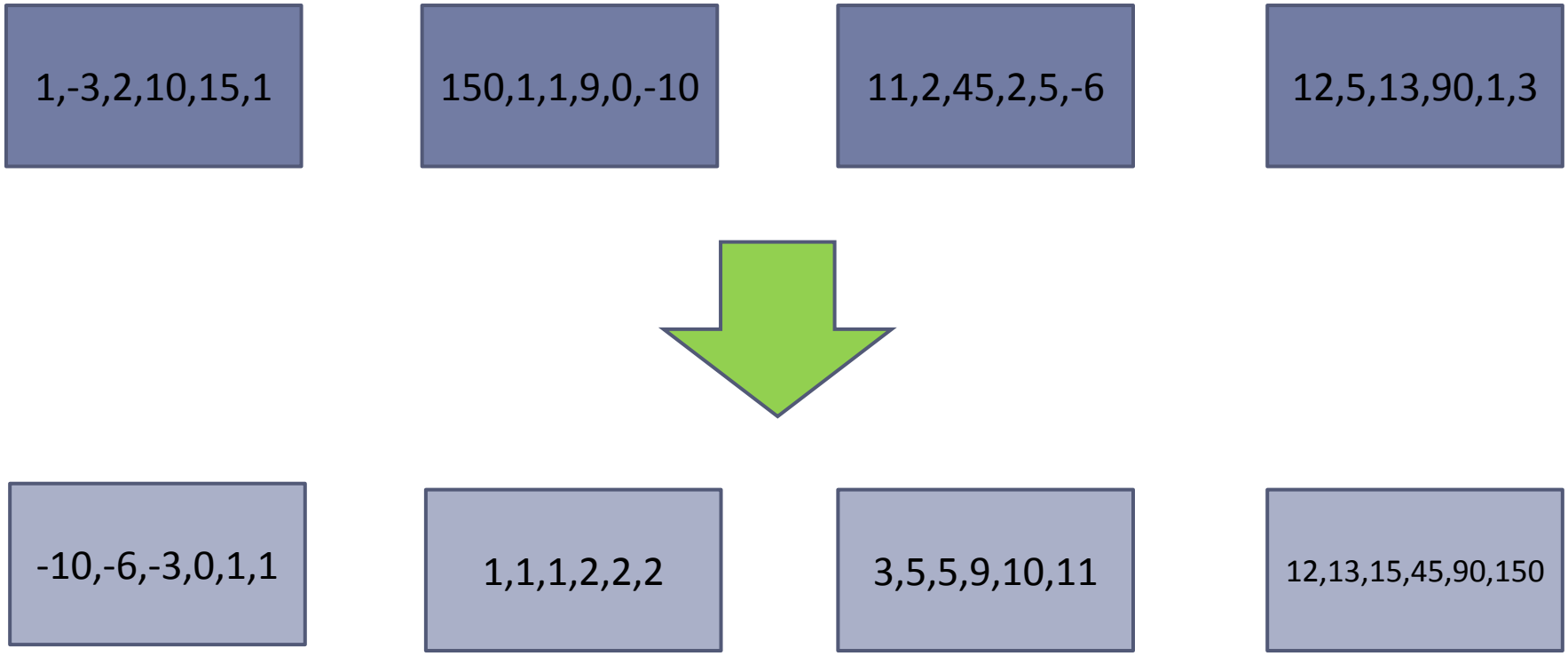


Пример. Модель клиенты-сервер.



Пример. Параллельная сортировка распределенного массива.

Задача. Расположить в невозрастающем порядке n элементов массива чисел, равномерно распределенных на p вычислительных узлах. Результат работы программы – отсортированная последовательность распределенно хранящаяся на p узлах, причем элементы массива с меньшими номерами должны быть размещены на узлах с меньшими номерами.



Пример. Параллельная сортировка распределенного массива.

Почему массив распределен по нескольким вычислительным узлам?

Одна из задач, решаемая параллельным программированием – хранение больших объемов данных на нескольких вычислительных узлах и доступ к ним.

Казалось бы, в настоящее время обеспечить компьютер большим объемом энергонезависимой памяти не - проблема. В то же время, рост оперативной памяти в современных устройствах хоть и вырос в тысячи раз по сравнению с первыми ЭВМ, на практике редко достигает более 10 Гб. Распределение данных по нескольким узлам позволяет использовать больше оперативной памяти и избегать дорогостоящих запросов к внешней памяти.

Пример. Параллельная сортировка распределенного массива.

Почему не стоит использовать последовательные алгоритмы для сортировки распределенного массива?

Рассмотрим на примере быстрой сортировки. Применение алгоритма к распределенным данным выявляет два основных недостатка: простаивание вычислительных узлов и большой объем коммуникаций (особенно на первых итерациях алгоритма).

Пример. Параллельная сортировка распределенного массива.

Проблемы при слиянии отсортированных подпоследовательностей.

Решать задачу параллельной сортировки имеет смысл в 2 этапа:

1. локальное упорядочивание элементов массива, т.е. сортировка подмассивов локально на каждом вычислительном узле;
2. слияние упорядоченных подмассивов.

Проблемы возникают на втором этапе, т.к. заранее неизвестно на каком узле должен храниться каждый из элементов массива и решение этой задачи весьма нетривиально.

Параллельный алгоритм быстрой сортировки

1. Выбрать каким-либо образом ведущий элемент и разослать его по всем процессорам системы (например, в качестве ведущего элемента можно взять среднее арифметическое элементов, расположенных на выбранном ведущем процессоре);
2. Разделить на каждом процессоре имеющийся блок данных на две части с использованием полученного ведущего элемента;
3. Образовать пары процессоров, для которых битовое представление номеров отличается только в позиции N (старший значащий бит), и осуществить взаимообмен данными между этими процессорами. Подмножество больших элементов ($S_>$) должно оказаться после обмена на процессорах с большими номерами, подмножество «меньше или равно» ($S_\leq$) на процессорах с меньшими номерами.

Параллельный алгоритм быстрой сортировки

- Подадим на вход алгоритма распределенный массив:

Node 0	Node 1	Node 2	Node 3
1,-3,2,10,15,1	150,1,1,9,0,-10	11,2,45,2,5,-6	12,5,13,90,1,3

В качестве ведущего элемента возьмем среднее арифметическое подмассива A_0 (=4).

Параллельный алгоритм быстрой сортировки

Итерация 0. Начало.

Node 0

1,-3,2,10,15,1

Node 1

150,1,1,9,0,-10

4

4

Node 2

11,2,45,2,5,-6

Node 3

12,5,13,90,1,3

Итерация 0. Конец.

Node 0

1,-3,2,1,2,2,-6

Node 1

1,1,0,-10,1,3

4

4

Node 2

11,45,5,10,15

Node 3

12,5,13,90,150,9

Параллельный алгоритм быстрой сортировки

Итерация 1.Начало.

Node 0

1,-3,2,1,2,2,-6

0

Node 1

1,1,0,-10,1,3

Node 2

11,45,5,10,15

14

Node 3

12,5,13,90,150,9

Итерация 1.Конец.

Node 0

-3,-6,0,-10

0

Node 1

1,1,1,3,1,2,1,2,2

Node 2

11,5,10,12,5,13,9

14

Node 3

90,150,45,15

Параллельный алгоритм быстрой сортировки

Упорядочим подмассивы и равномерно распределим их по узлам:

Node 0	Node 1	Node 2	Node 3
-10,-6,-3,0,1,1	1,1,1,2,2,2	3,5,5,9,10,11	12,13,15,45,90,150

Параллельный алгоритм быстрой сортировки. Анализ.

- Алгоритм работает эффективно, если входной массив - набор равномерно распределенных чисел.
- Что может произойти, если массив будет неравномерно распределенным? (например, несколько очень больших чисел, а остальные числа малы по сравнению с ними)
- Для того, чтобы гарантировать работу алгоритма необходимо памяти на каждом узле не менее, чем размер распределенного массива.

Коллективные взаимодействия

Коллективная связь осуществляет взаимодействия типа «один-ко-всем», «все-к-одному», «все-со-всеми». MPI имеет следующие функции коллективной связи:

1. Синхронизация (barrier) – синхронизирует все процессы группы

2. Глобальные функции связи

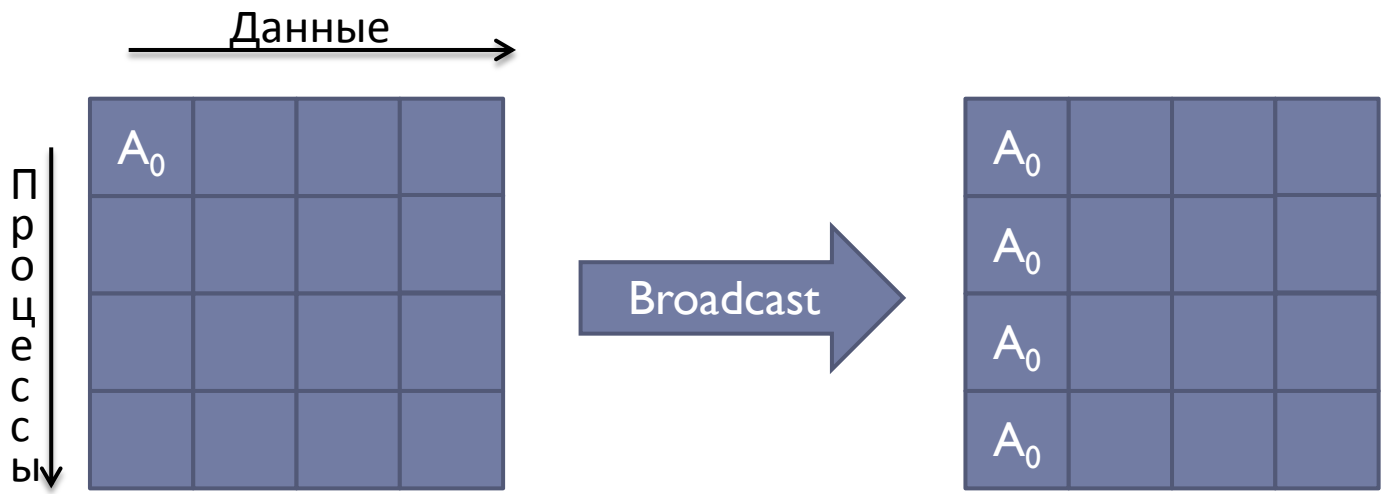
- broadcast – передача данных от одного процесса группы к остальным (в т.ч. самому себе)
- gather – сбор данных от всех процессов группы к одному процессу
- scatter – разброс данных от одного процесса группы ко всем остальным (в т.ч. самому себе)

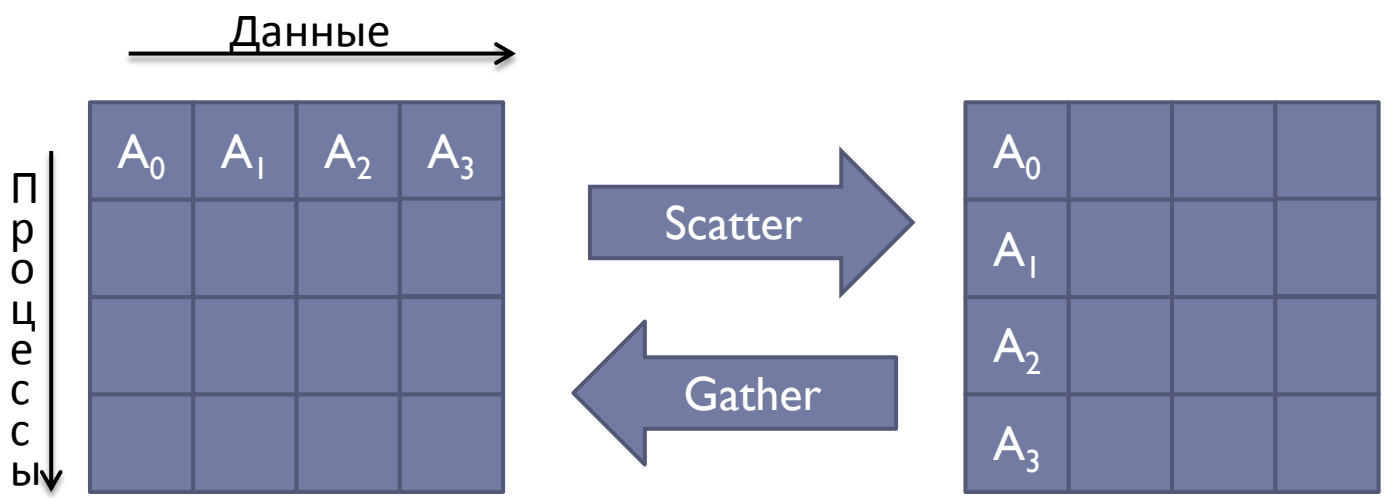
Коллективные взаимодействия

- allgather – сбор данных в цикле по всем процессам группы, все элементы группы получают результат от всех
- alltoall – разброс/сбор данных от всех элементов ко всем элементам группы (полный обмен)

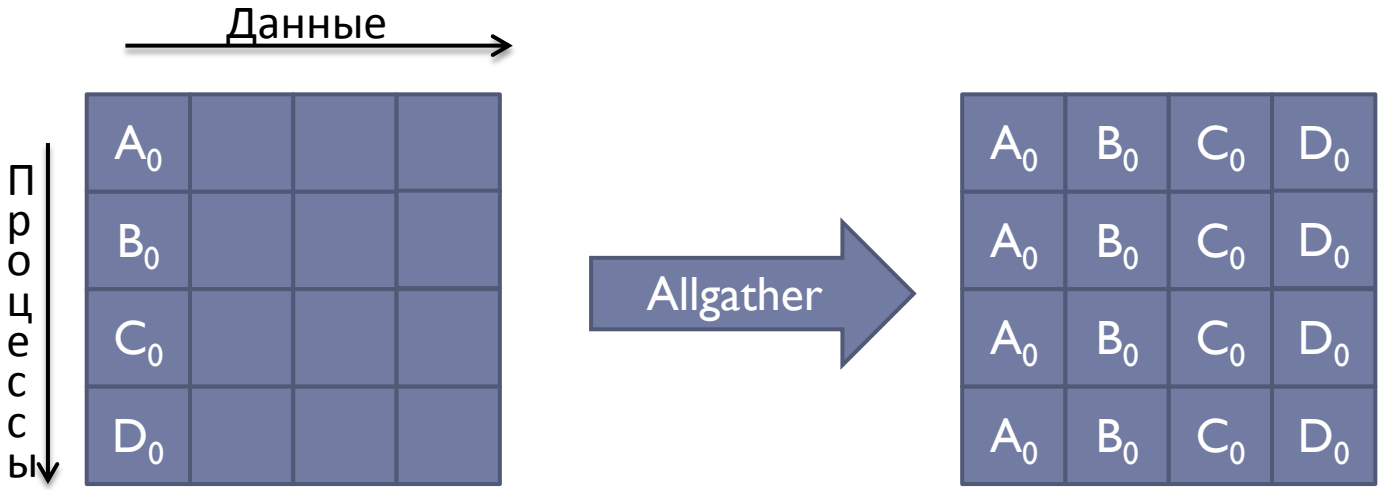
3. Глобальные операции редуцирования типа sum, max, min и определяемые пользователем функции

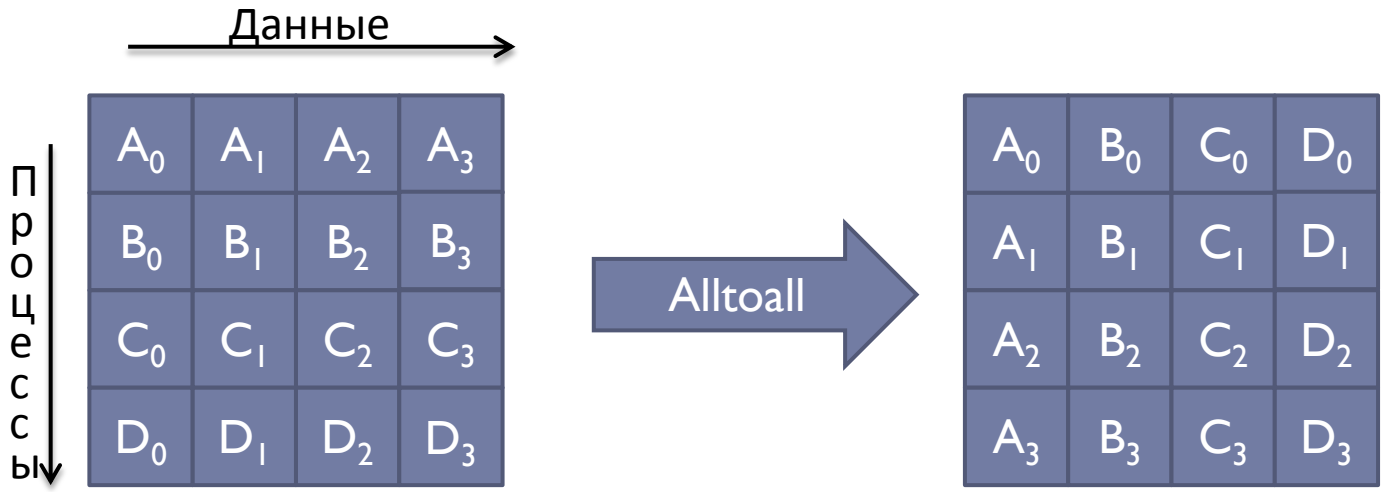
Коллективные взаимодействия





Коллективные взаимодействия





Коллективные взаимодействия

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

MPI_Comm comm – коммуникатор

Функция синхронизирует все процессы коммуникатора comm. До исполнения функции всеми процессами коммуникатора, управление блокируется.

Коллективные взаимодействия

```
int MPI_Bcast(buffer, count, datatype, root, comm)
```

void* buffer – адрес буфера

int count – количество элементов в буфере

MPI_Datatype datatype – тип данных

int root – номер корневого процесса

MPI_Comm comm – коммуникатор

Коллективные взаимодействия

```
int MPI_Gather(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, root, comm)

int MPI_Allgather(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, comm)

int MPI_Scatter(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, root, comm)

int MPI_Alltoall(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, comm)
```

Коллективные взаимодействия

void* sendbuf – адрес посылаемого буфера

int sendcount – количество посылаемых данных

MPI_Datatype sendtype – тип посылаемых данных

void* recvbuf – адрес буфера приема

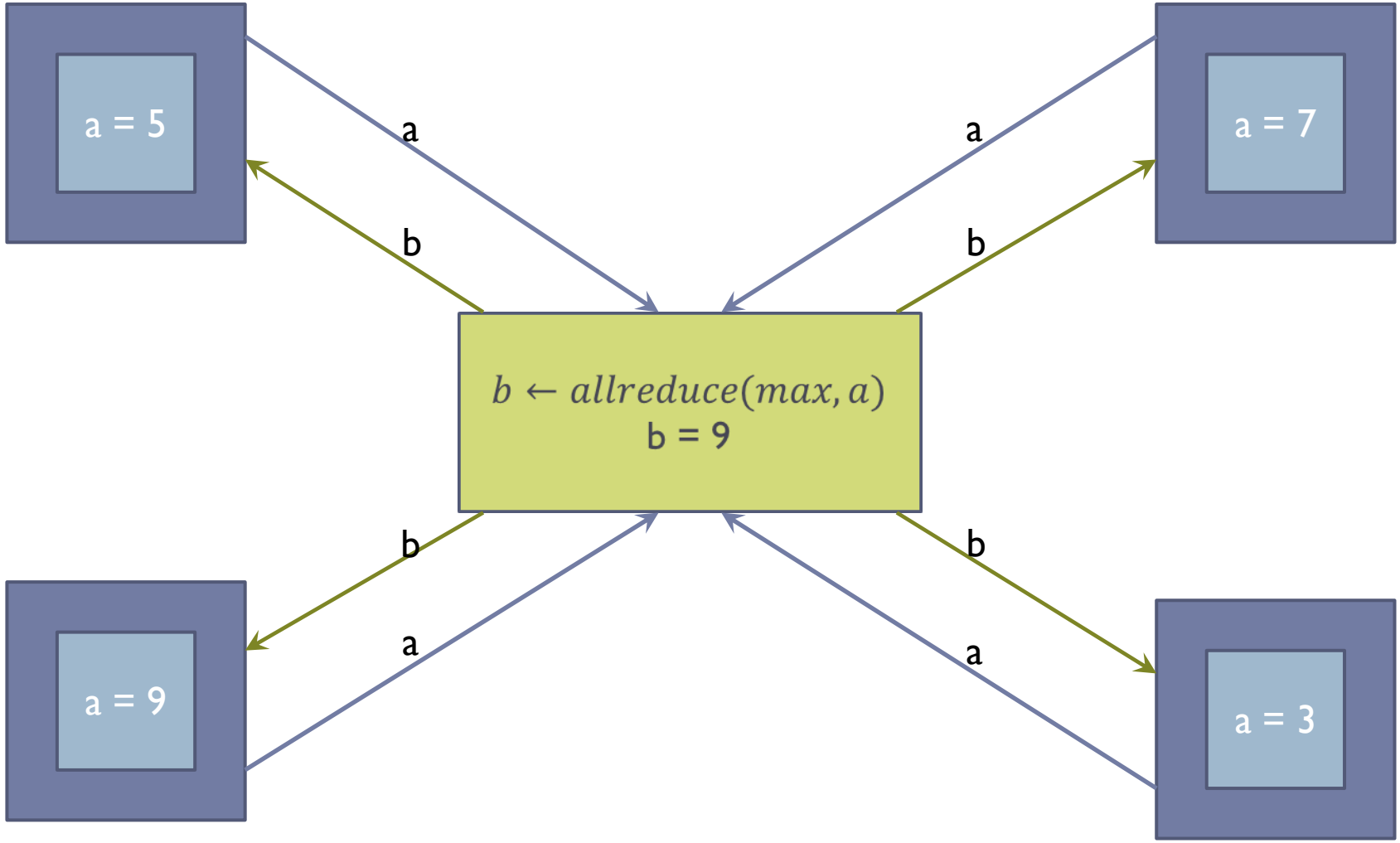
int recvcount – количество принимаемых данных

MPI_Datatype recvtype – тип принимаемых данных

int root – ранг процесса-отправителя

MPI_Comm comm - коммуникатор

Коллективные взаимодействия. Редукционные операции.



Коллективные взаимодействия. Редукционные операции.

```
int MPI_Reduce(sendbuf, recvbuf, count,
                        datatype, op, root, comm)
int MPI_Allreduce(sendbuf, recvbuf, count,
                        datatype, op, comm)
```

void* sendbuf – адрес передаваемого буфера

void* recvbuf – адрес буфера приема

int count – количество передаваемых элементов

MPI_Datatype datatype – тип данных

MPI_Op op – операция редукции

int root – номер корневого процесса

MPI_Comm comm - коммуникатор

Пример программы

Задача. На всех ли процессорах значение переменной a положительно?

Решение. Пусть переменная x принимает значение 1, если a положительна, и 0 в противном случае. Тогда задача сводится к поиску значения формулы: $\bigwedge_{i=0}^{n-1} x(i)$.

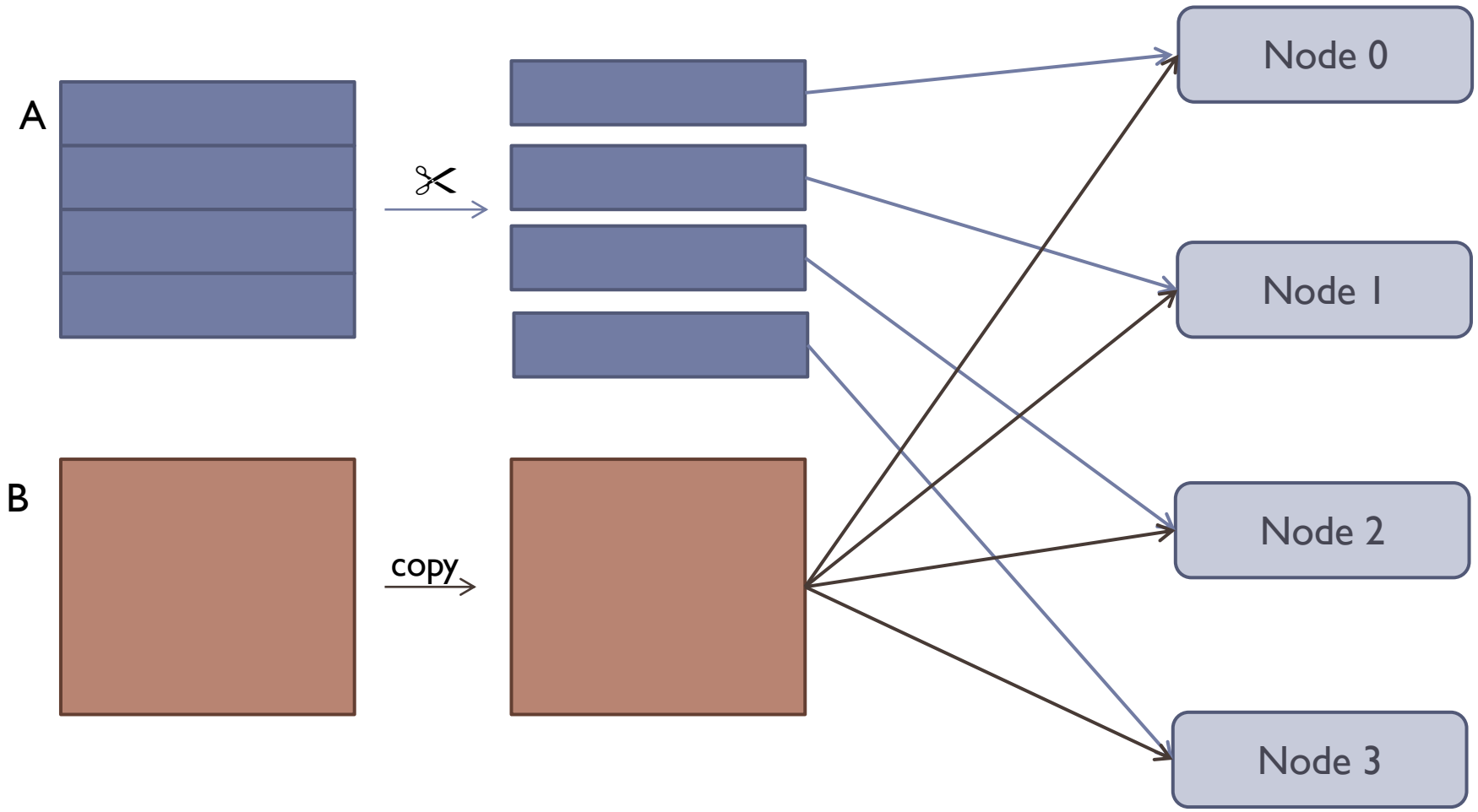
```
int x = 0;
if (a>0) x=1;
int Ans;
MPI_Reduce(&x, &Ans, 1, MPI_INT, MPI_LAND, root, comm);
```

Параллельное умножение матриц

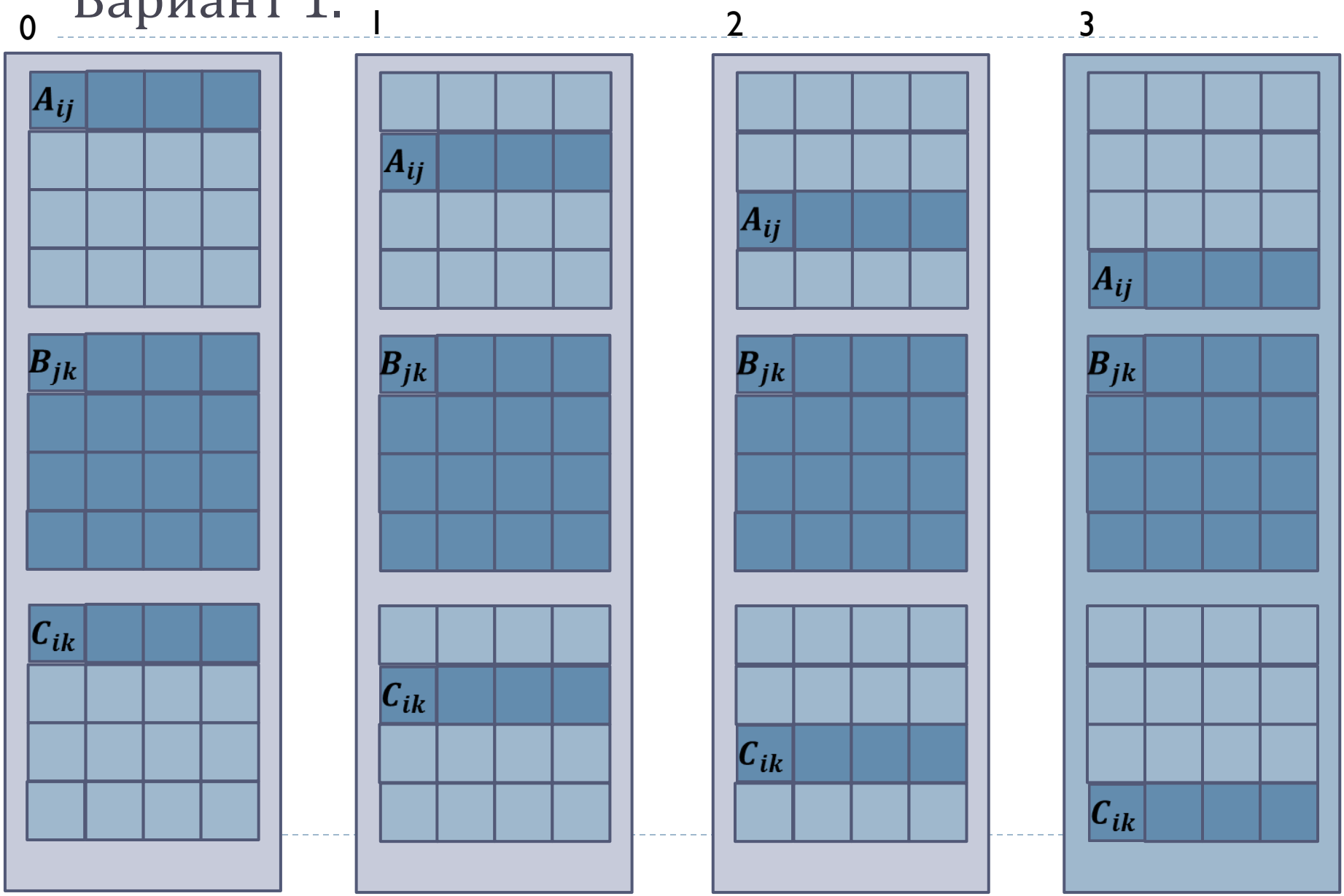
- Задача. $C = A \times B$

Решение. Т.к. решение задачи предполагает большое число независимых однотипных операций, то целесообразно проводить вычисления параллельно на нескольких вычислительных узлах.

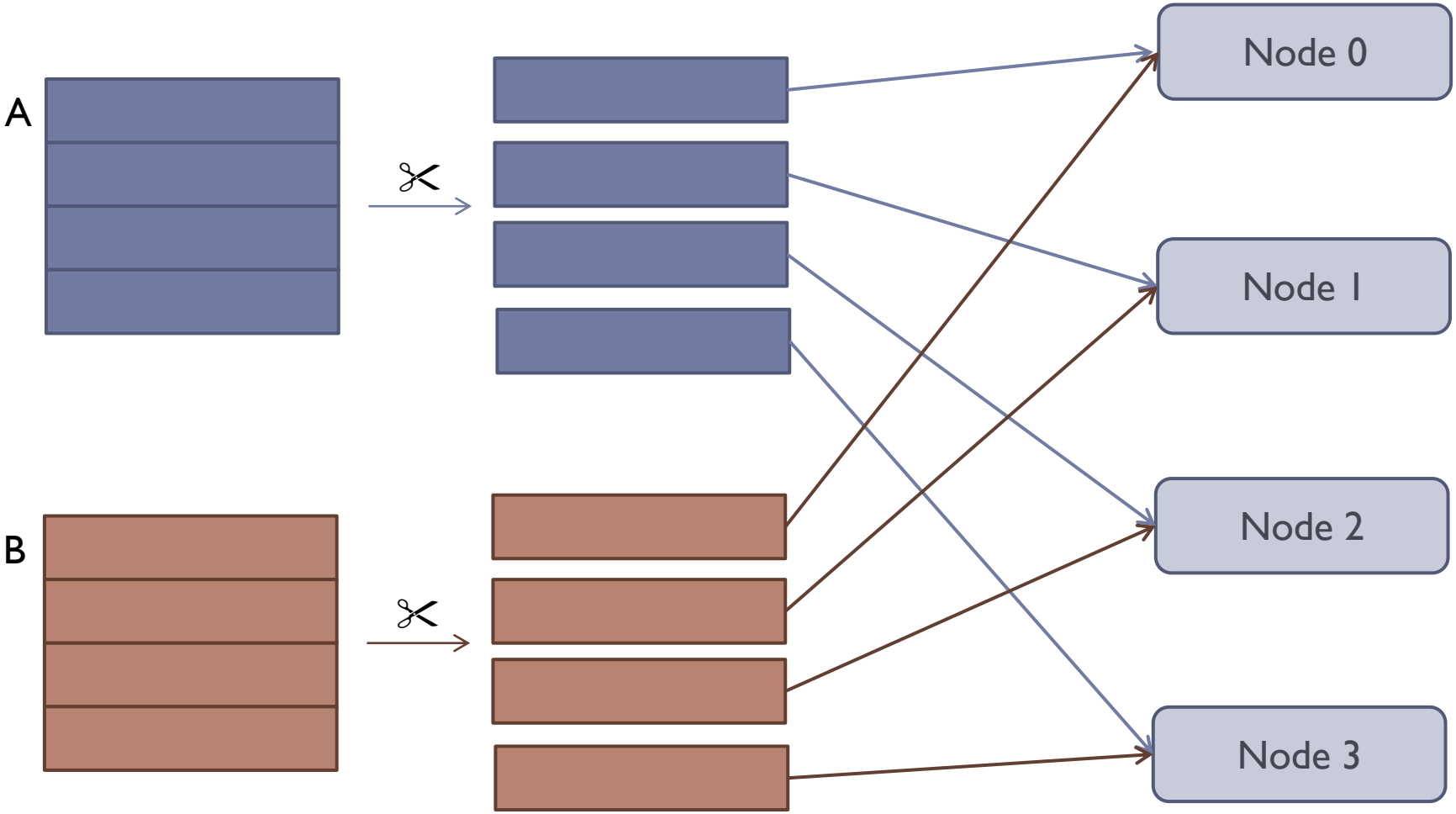
Параллельное умножение матриц. Вариант 1.



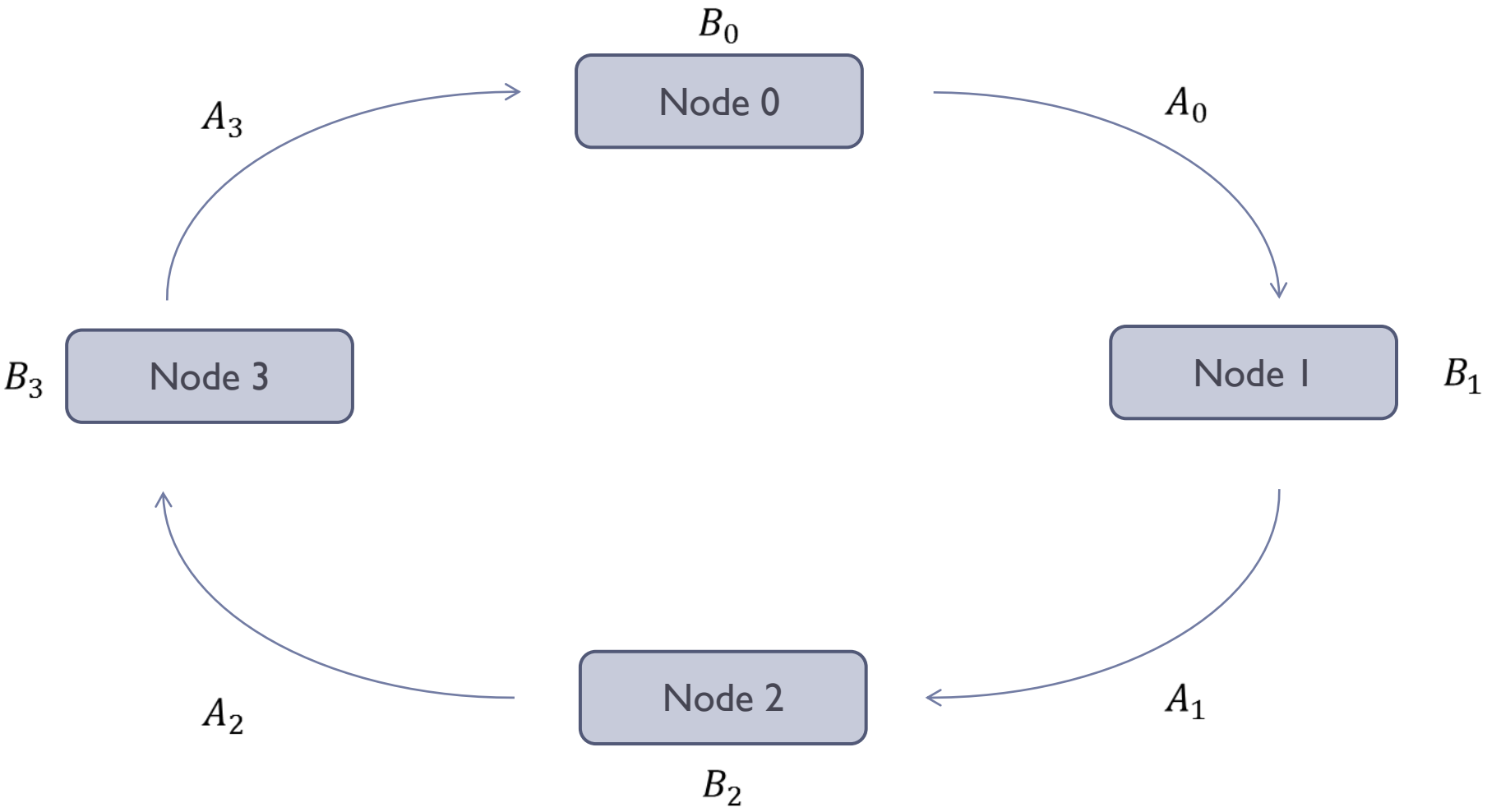
Параллельное умножение матриц. Вариант 1.



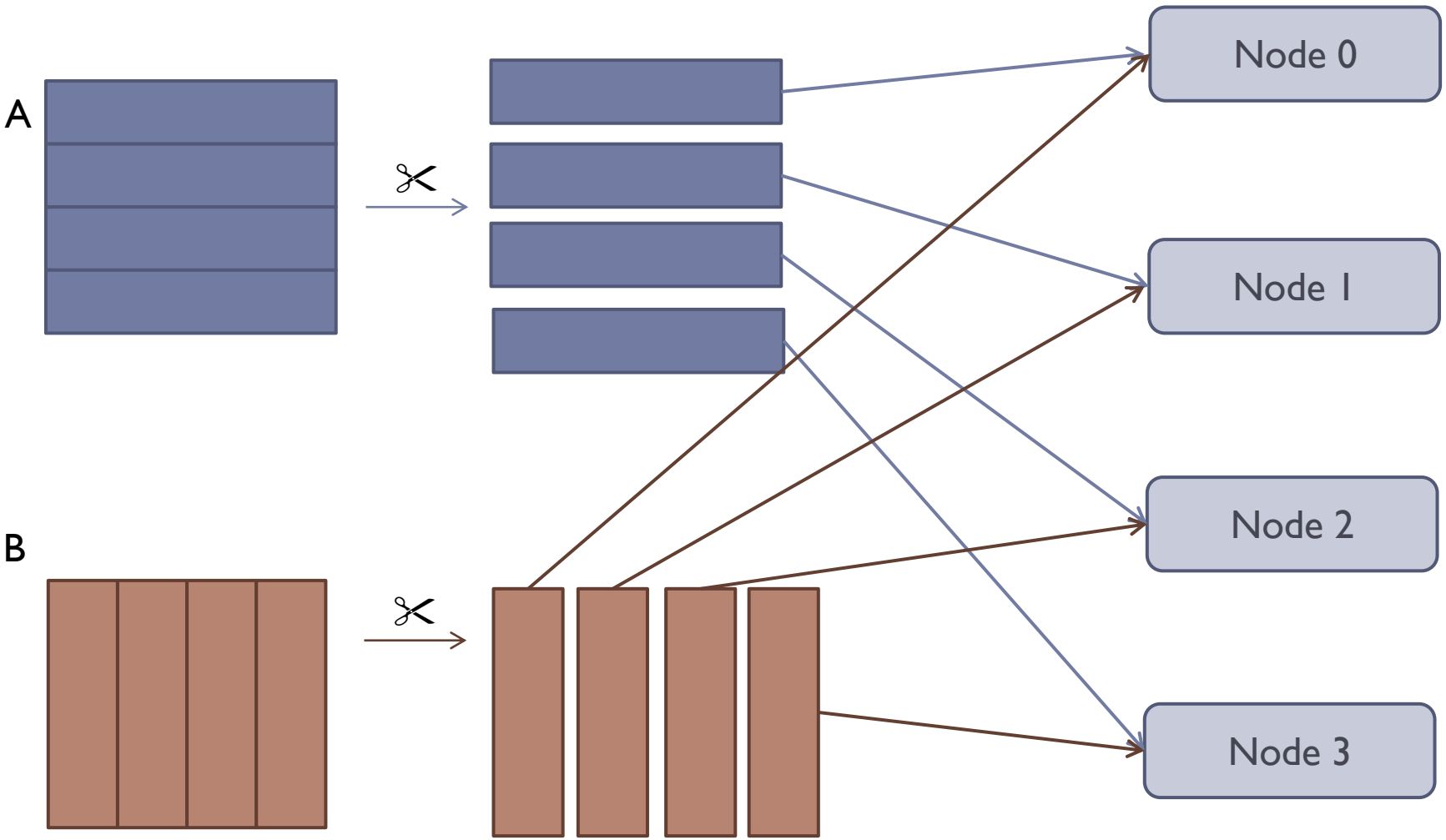
Параллельное умножение матриц. Вариант 2.



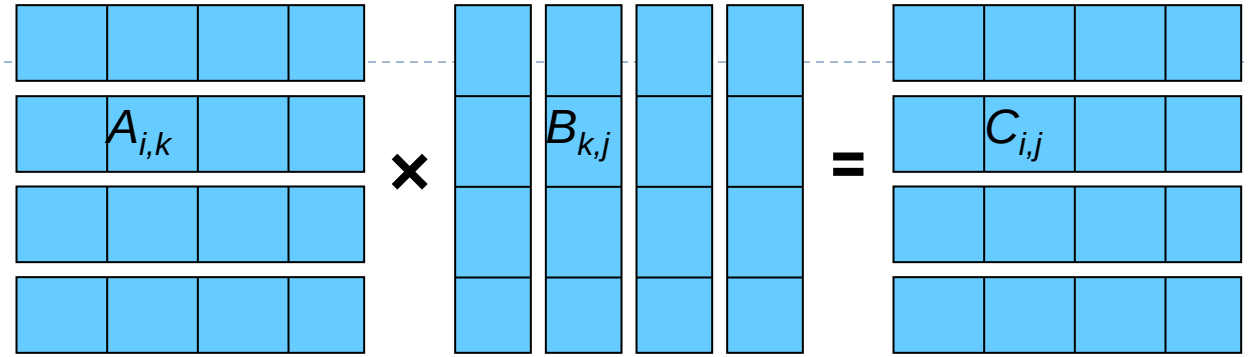
Параллельное умножение матриц. Вариант 2.

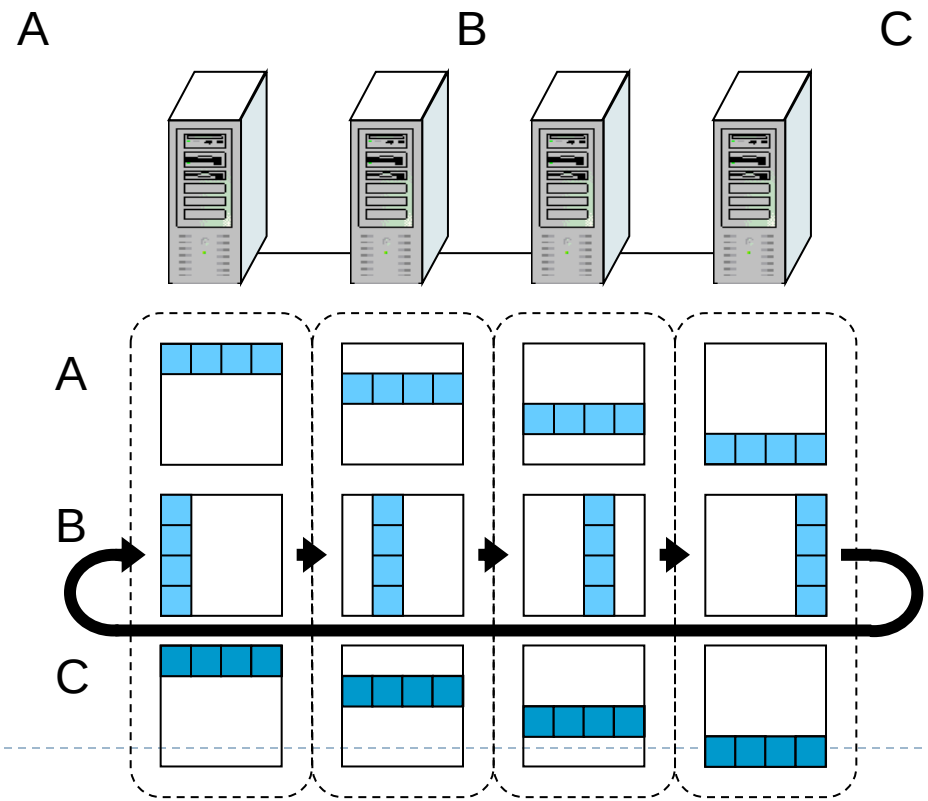


Параллельное умножение матриц. Вариант 3.



Параллельное умножение матриц. Вариант 2.





Практикум

1. Скомпилировать и запустить программу “Hello, World!”
2. Посчитать скалярное произведение векторов, распределенных на нескольких вычислительных узлах.
3. Параллельное умножение матрицы на вектор.
4. Параллельное умножение матрицы на матрицу.
5. Параллельная реализация метода простых итераций

Литература

- В.Д. Корнеев «Параллельное программирование в MPI»
- В.Д. Корнеев «Параллельное программирование кластеров»
- http://www.mcs.anl.gov/research/projects/mpi/www/

Спасибо за внимание!

Быстрая сортировка

- Выбрать элемент, называемый опорным.
- Сравнить все остальные элементы с опорным, на основании сравнения разбить множество на три подмножества — «меньшие опорного» ($S_<$), «равные» ($S_=$) и «большие» ($S_>$) , расположить их в порядке меньшие-равные-большие.
- Повторить рекурсивно для «меньших» и «больших».

Back

Битовое представление числа

Задача. Представить числа 0, 1, 2, 3 четырьмя битами.

Решение.

0 = 0000

1 = 0001

2 = 0010

3 = 0011

Back