Коллективные взаимодействия

НОЦ МФТИ 2013

Введение

- Коллективные операции используются для сбора и рассылки информации между несколькими процессами
- В общем случае, все операции по пересылке данных между процессами могут быть сделаны пересылками типа точка-точка
- Поскольку некоторые последовательности операций обмена встречаются довольно часто, MPI предоставляет набор коллективных операций
- Эти процедуры построены используя взаимодействия типа точка-точка
- Возможна реализация алгоритмов самостоятельно, однако для этого требуется довольно много затрат и MPI обычно предоставляет оптимальный вариант решения

Особенности

- Коллективные операции пересылают данные между всеми процессами коммуникатора
- В коллективных операциях отсутствует понятие tag сообщения
- Важен порядок исполнения на разных процессах. Пользователь должен следить за тем, чтобы одни и теже коллективные операции на разных процессах выполнялись в одной и той же последовательности
- Если необходимо произвести коллективный обмен данными только между частью процессов, необходимо их выделить в отдельный коммуникатор

Синхронизация

- Для большинства реализаций MPI вызов коллективной операции производит синхронизацию процессов. Однако стандарт не гарантирует синхронизации и при написании программ не стоит на это рассчитывать.
- Отдельная операция, MPI_Barrier, синхронизирует процессы, однако не производит обмена данными. Тем не менее она часто рассматривается как коллективное взаимодействие.

Типы

- Барьерная синхронизация всех процессов
- Рассылка одинаковых данных от одного процесса ко всем (broadcast)
- Глобальные операции по обработке данных такие как sum, min, max и другие. В частности могут задаваться пользователем (reduce)
- Сбор данных от всех процессов к одному (gather)
- Рассылка различных данных от одного всем (scatter)
- Набор операций когда все процессы получают результат от gather, scatter и reduce. Также существуют векторные варианты большинства операций, когда каждое сообщение имеет разные размер

Особенности

- Все операции блокирующие
- У сообщений нету идентификатора — tag
- Если необходимо выполнить для части процессов — необходимо сначала обьеденить их в отдельный коммуникатор
- Могут быть использованы только с собственными типами MPI

Барьерная синхронизация

- Используется для блокировки исполнения процессов до тех пор, пока все не вызовут одну и туже функцию. Функция вернет точку исполнения только после того как все процессы коммуникатора вызовут барьер.
- Иногда возникают ситуации когда некоторые процессы не могут продолжит исполнение, пока один не выполнит какой-либо набор инструкций. Общий пример, если главный процесс считывает входные данные перед их рассылкой между процессами.
- Функция реализована программно и может вносить некую задержку в исполнении. Поэтому стоит избегать частых использований и вставлять в код только там где она необходима.

Барьерная синхронизация

- Представлена одной функцией в MPI:

```
MPI_Barrier(MPI_Comm comm);
```

Broadcast

- Используется для рассылки сообщения от одного процесса, называемого «root», всем процессам в коммуникаторе, включая root.
- Аргумент главного процесса root должен быть одним и тем же у всех процессов.
- Данные с главного процесса будут скопированы ко всем.
- One-to-all коммуникация

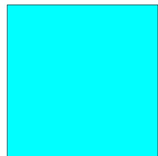
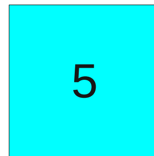
MPI_Bcast

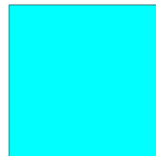
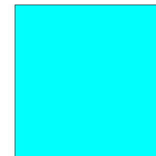
Пример использования

MPI_Bcast (&buffer,count,datatype,root,comm)

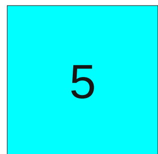
root = 1;
count = 1;

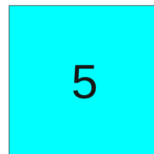
Task 0 **Task 1** **Task 2** **Task 3**

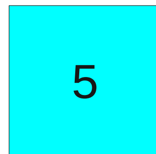



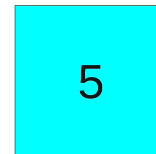



buffer до пересылки









buffer после пересылки

Синтаксис

MPI_Bcast (send_buffer, send_count, send_type, rank, comm)

Аргументы функции:

send_buffer указатель на область памяти, куда будут записаны данные, либо откуда считаны

send_count число элементов в сообщении

send_type тип данных в сообщении

rank rank главного процесса

comm MPI коммуникатор

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

Пример кода

```
#include <stdio.h>
#include <mpi.h>
void main(int argc, char *argv[]) {
    int rank;
    double param;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank==5)
        param=23.0;
    MPI_Bcast(&param,1,MPI_DOUBLE,5,MPI_COMM_WORLD);
    printf("P:%d after broadcast parameter is %f \n",rank,param);
    MPI_Finalize();
}
```

Пример кода

Вывод:

```
P: 0  after broadcast param is  23.
P: 5  after broadcast param is  23.
P: 2  after broadcast param is  23.
P: 3  after broadcast param is  23.
P: 4  after broadcast param is  23.
P: 1  after broadcast param is  23.
P: 6  after broadcast param is  23.
```

Gather

- Используется для сбора информации от каждого процесса и отсылки ее главному процессу в порядке ранков процессов
- All-to-one коммуникация
- Аргумент входящего сообщения имеет смысл только на главном процессе

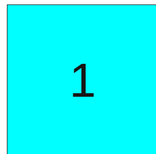
MPI_Gather

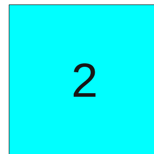
Сбор данных от всех процессов

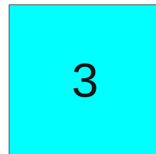
MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf,recvcount,recvtype,root,comm)

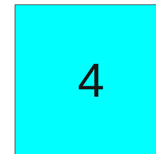
recvcnt = 1; sendcnt = 1; root = 1;

Task 0 **Task 1** **Task 2** **Task 3**

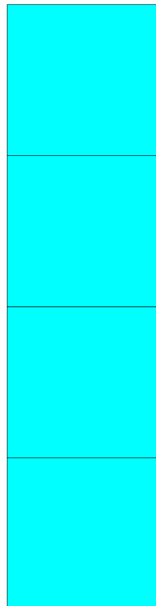
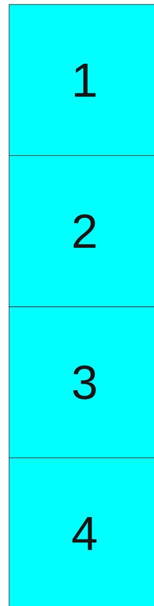


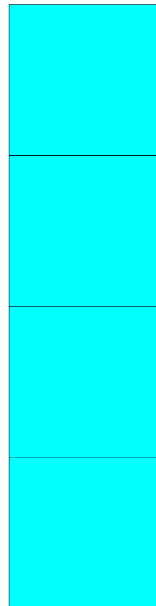
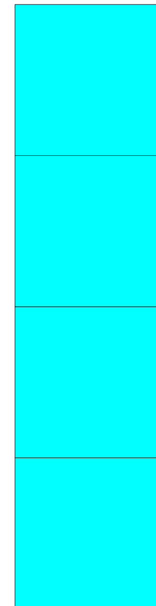






sendbuf до пересылки

recvbuf после пересылки

Синтаксис

MPI_Gather (send_buffer, send_count, send_type, recv_buffer, recv_count, recv_count, recv_rank, comm)

- Список аргументов:
  - send_buffer in адрес отсылаемого буфера
  - send_count in число элементов в буфере
  - send_type in тип данных
  - recv_buffer out адрес принимаемого буфера
  - recv_count in число элементов в буфере для одного приема
  - recv_type in тип принимаемых данных
  - recv_rank in rank главного процесса
  - comm in mpi коммуникатор

```
int MPI_Gather ( void* send_buffer, int send_count, MPI_datatype
    send_type, void* recv_buffer, int recv_count, MPI_Datatype recv_type,
    int rank, MPI_Comm comm )
```

Пример

```
#include   <stdio.h>
#include   <mpi.h>
void main(int argc, char *argv[])
{
    int rank,size;
    double param[16],mine;
    int sndcnt,rcvcnt;
    int i;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    sndcnt=1;
    mine=23.0+rank;
    if(rank==7) rcvcnt=1;


    MPI_Gather(&mine,sndcnt,MPI_DOUBLE,param,rcvcnt,MPI_DOUBLE,7,MPI_COMM_WORLD
    );

    if(rank==7)
    for(i=0;i<size;++i) printf("PE:%d param[%d] is %f \n",rank,i,param[i]);
    MPI_Finalize();
}
```

```
PE:7 param[0] is 23.000000
PE:7 param[1] is 24.000000
PE:7 param[2] is 25.000000
PE:7 param[3] is 26.000000
PE:7 param[4] is 27.000000
PE:7 param[5] is 28.000000
PE:7 param[6] is 29.000000
PE:7 param[7] is 30.000000
PE:7 param[8] is 31.000000
PE:7 param[9] is 32.000000
```

Scatter

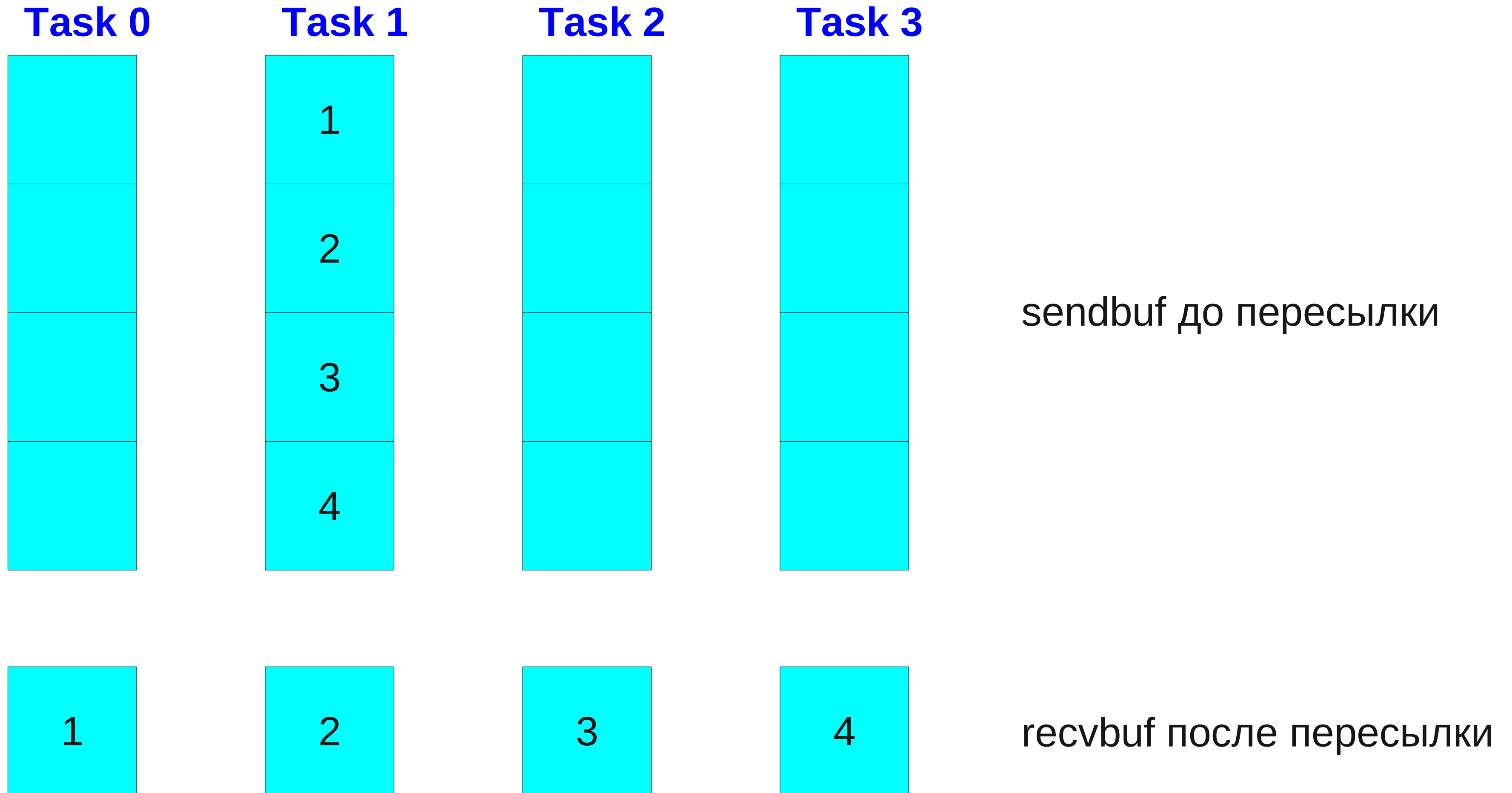
- Операция противоположная gather: распределить данные между процессами от главного процесса
- Данные берутся в порядке возрастания ранка процесса
- One-to-all коммуникация
- Аргумент отправляемого буфера имеет смысл только на главном процессе

MPI_Scatter

Рассылка от одного всем процессам в группе

MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf,recvcnt,recvtype,root,comm)

recvcnt = 1; sendcnt = 1; root = 1;



Синтаксис

MPI_Scatter (send_buffer, send_count, send_type, recv_buffer, recv_count, recv_type, rank, comm)

- Список аргументов:
  - send_buffer in адрес отсылаемого буфера
  - send_count in число элементов из буфера, отсылаемых каждому процессу (не размер всего буфера)
  - send_type in тип данных
  - recv_buffer out адрес принимающего буфера
  - recv_count in число элементов в принимаемом буфере
  - recv_type in тип данных
  - rank in rank главного процесса
  - Comm in mpi коммуникатор

int MPI_Scatter (void* send_buffer, int send_count, MPI_datatype send_type, void* recv_buffer, int recv_count, MPI_Datatype recv_type, int rank, MPI_Comm comm)

Пример

```
#include <mpi.h>
int main (int argc, char *argv[]) {
	int rank,size,i,j;
	double param[4],mine;
	int sndcnt,revcnt;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD,&rank);
	MPI_Comm_size(MPI_COMM_WORLD,&size);
	revcnt=1;
	if(rank==3){
		for(i=0;i<4;i++) param[i]=23.0+i;
		sndcnt=1;
	}
	MPI_Scatter(param,sndcnt,MPI_DOUBLE,&mine,revcnt,
	MPI_DOUBLE,3,MPI_COMM_WORLD);
	printf("P:%d mine is %f\n",rank,mine);
	MPI_Finalize();
}
```

```
P:0 mine is 23.000000
P:1 mine is 24.000000
P:2 mine is 25.000000
P:3 mine is 26.000000
```

Reduce

- Используется для обработки элементов данных с каждого процессора в один общий буфер используя некий оператор. Сохраняет полученный результат на главном процессе root.
- MPI_Reduce позволяет:
  - Собирать данные с каждого процесса
  - Обрабатывать данные по ходу сбора
  - Сохраняет в одно значение у главного процесса
- All-to-one коммуникация

MPI_Reduce

Сбор данных с их обработкой

MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)

```
root = 1;
count = 1;
op = MPI_SUM;
```

	Task 0	Task 1	Task 2	Task 3
sendbuffer до пересылки	1	2	3	4
recvbuffer после пересылки		10		

Список операторов

Оператор	Описание
MPI_MAX	максимум
MPI_MIN	минимум
MPI_SUM	сумма
MPI_PROD	произведение
MPI_LAND	Логическое И
MPI_BAND	Побитовое И
MPI_LOR	Логическое ИЛИ
MPI_BOR	Побитовое ИЛИ
MPI_LXOR	Логический xor
MPI_BXOR	Побитовый xor
MPI_MINLOC	Минимум и ранк процесса
MPI_MAXLOC	Максимум и ранк процесса

Синтаксис

MPI_Reduce (send_buffer, recv_buffer, count, datatype, operation, rank, comm)

- Аргументы:
  - send_buffer in отсылаемый буфер
  - recv_buffer out принимаемый буфер
  - count in число элементов в буфере
  - datatype in тип данных в сообщении
  - operation in операция
  - rank in rank главного процесса
  - comm in mpi коммуникатор

int MPI_Reduce (void* send_buffer, void* recv_buffer, int count, MPI_Datatype datatype, MPI_Op operation, int rank, MPI_Comm comm)

Пример

```
#include   <stdio.h>
#include   <mpi.h>
void main(int argc, char *argv[])
{
      int rank;
      int source,result,root;

      /* run on 10 processors */
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD,&rank);

      root=7;
      source=rank+1;
      MPI_Barrier(MPI_COMM_WORLD);

      MPI_Reduce(&source,&result,1,MPI_INT,MPI_PROD,root,MPI_COMM_WORLD);
      if(rank==root) printf("PE:%d MPI_PROD result is %d \n",rank,result);

      MPI_Finalize();
}
```

```
PE:7 MPI_PROD result is 362880
```

MPI_MINLOC, MPI_MAXLOC

- Разработаны для нахождения глобального минимума/максимума и индекса, который ассоциируется с экстремумом
- Если несколько экстремумов, берется первый.
- Используется на данными, хранящими значение и индекс

MPI_Datatypes:

C:

MPI_FLOAT_INT, MPI_DOUBLE_INT, MPI_LONG_INT, MPI_2INT,

MPI_SHORT_INT, MPI_LONG_DOUBLE_INT

Fortran:

MPI_2REAL, MPI_2DOUBLEPRECISION, MPI_2INTEGER

Пример

```
#include <mpi.h>
/* Run with 16 processes */
int main (int argc, char *argv[]) {
	int rank;
	struct {
		double value;
		int rank;
	} in, out;
	int root;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD,&rank);
	in.value=rank+1;
	in.rank=rank;
	root=7;
MPI_Reduce(&in,&out,1,MPI_DOUBLE_INT,MPI_MAXLOC,root,MPI_COMM_WORLD)
;
	if(rank==root) printf("PE:%d max=%lf at rank %d\n",rank,out.value,out.rank);
MPI_Reduce(&in,&out,1,MPI_DOUBLE_INT,MPI_MINLOC,root,MPI_COMM_WORLD);
	if(rank==root) printf("PE:%d min=%lf at rank %d\n",rank,out.value,out.rank);
	MPI_Finalize();
}
```

Вывод
P:7 max = 16.000000 at rank 15
P:7 min = 1.000000 at rank 0

Собственные операции

- Можно делать сои собственные reduce операции

C function of type MPI_User_function:

```
void my_operator (void *invec, void *inoutvec, int *len,
MPI_Datatype *datatype)
```

Fortran function of type:

```
FUNCTION MY_OPERATOR (INVEC(*), INOUTVEC(*),
LEN, DATATYPE)

<type> INVEC(LEN),INOUTVEC(LEN)
INTEGER LEN,DATATYPE
```

Собственные операторы

- Как должен работать оператор

  for (i=1 to len)

  inoutvec(i) = inoutvec(i) op invec(i)
- Должен не обязательно должен быть коммутирующим
- inoutvec используется как аргумент и результат оператора

Регистрация своего оператора

- Тип оператора MPI_Op или INTEGER
- Если commute TRUE, сбор данных может сильно ускорится

C:

```
int MPI_Op_create(MPI_User_function *function,
int commute, MPI_Op *op)
```

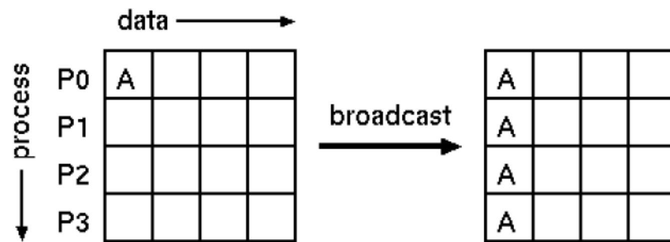
Fortran:

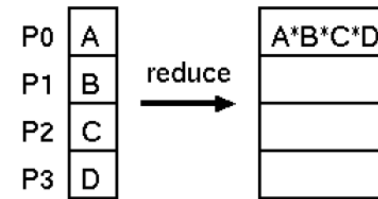
```
EXTERNAL FUNC
INTEGER OP,IERROR
LOGICAL COMMUTE
MPI_OP_CREATE (FUNC, COMMUTE, OP, IERROR)
```

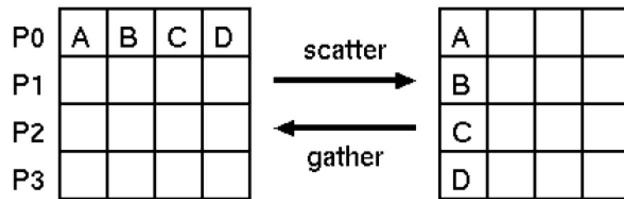
Операции All-to-all

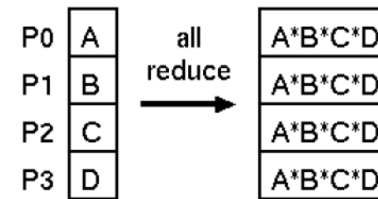
- MPI_ALLGATHER
- MPI_ALLTOALL
- MPI_ALLSCATTER
- MPI_ALLREDUCE
- MPI_REDUCE_SCATTER
- MPI_SCAN

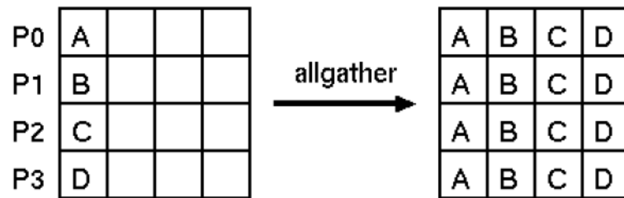
Сводная таблица операций

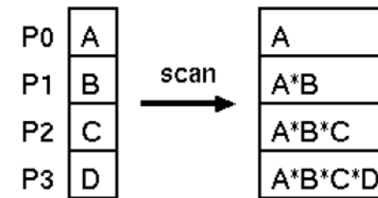


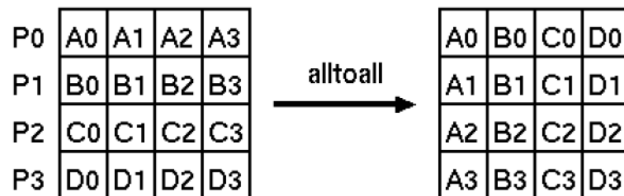


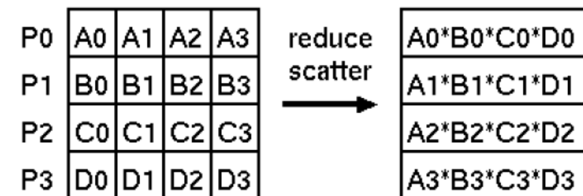












Вопросы.