COMP4007: 并行处理和体系结构

第六章：基于MPI的并行编程II

授课老师：王强、施少怀
助　　教：林稳翔、刘虎成

哈尔滨工业大学（深圳）

要点

- 阻塞与非阻塞通信
- MPI 派生数据类型

- 集合通信
  - 基础集合通信
  - 高级集合通信

- 示例学习

阻塞消息传递

- 一个阻塞消息传递的操作不会返回，直到这个操作完成
  - 它可以保证语义的正确性
  - 性能相对不高，因为在数据传输过程中，收发端不能做其他动作，即被阻塞

- MPI_Send 与 MPI_Recv 都是阻塞操作

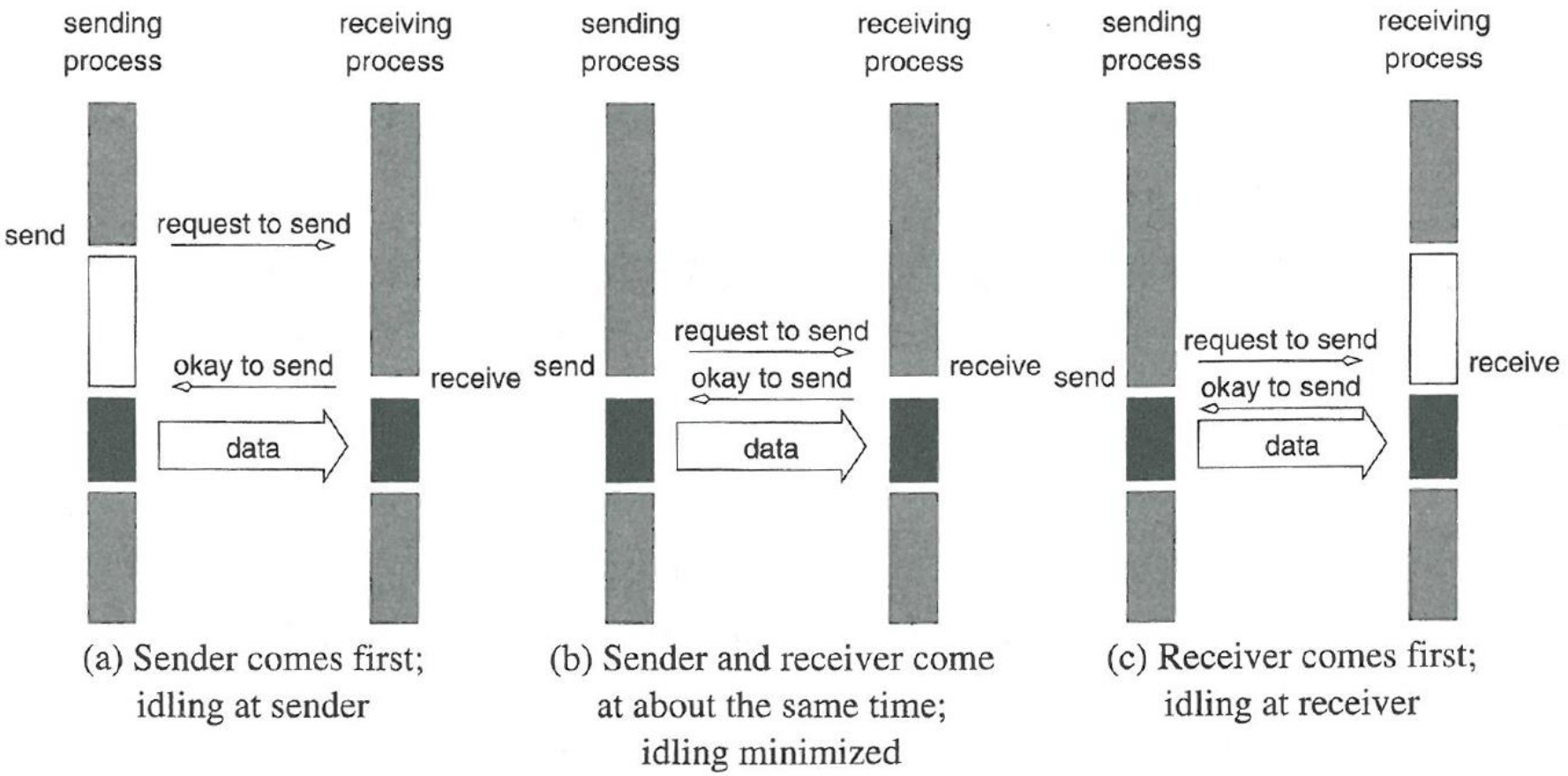
阻塞式收发

- 阻塞式操作的两个问题
  - 空转开销
    - 由于发送操作和接收操作可以在不同的时间执行，因此发送方或接收方中的一方将在一段时间内空闲，等待另一方
  - 死锁
    - 以下示例展示了一个死锁情景，其中两个进程尝试交换两个值，a 和 b

Process 0	Process 1
Send(&a, 1, 1); Receive(&b, 1, 1);	Send(&a, 1, 0); Receive(&b, 1, 0);

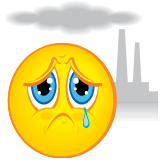
提问：如何纠正上述代码?

阻塞式收发

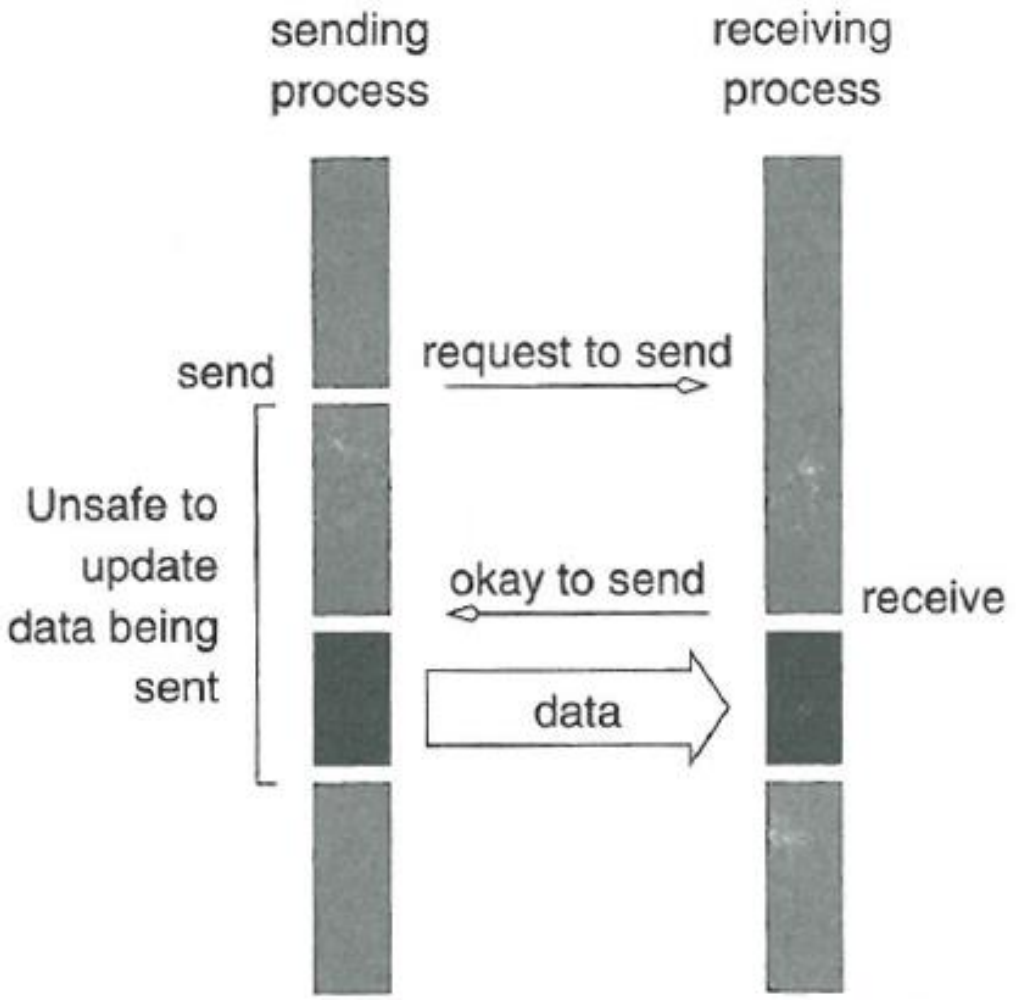


Source: Fig. 6.1 of Ref. [1].

非阻塞消息传递

- 非阻塞收发操作会立刻返回，即使通信过程还未完成
  - 它可以规避大部分死锁 
  - 它可以通过将通信与计算重叠进行，获得更高的性能 
  - 这需要复杂的程序设计 

非阻塞消息传递



Source: Fig. 6.4 of Ref. [1].

非阻塞消息传递

- 进程如何知道数据传输任务是否已经结束？
- 程序需要检查之前的非阻塞操作的状态
  - 如果数据传输完成，则可以开始处理数据
  - 如果没完成，则需要继续休眠或进行其他安全计算任务

MPI中的非阻塞消息传递

- MPI提供了一对函数用于非阻塞发送和接收操作：MPI_Isend和MPI_Irecv

- int MPI_Isend(void* buf,
  int count,
  MPI_Datatype datatype,
  int dest,
  int tag,
  MPI_Comm comm,
  MPI_Request *request /* out */);
  - <buf, count, datatype> 指定了消息
  - dest 指定应该接收消息的进程编号
  - tag（非负整数）被用于辨识消息
  - comm 指定进程组
  - request 是一个指向请求对象的指针，它被 MPI_Test 和 MPI_Wait 函数用于识别哪些是我们想要查询或等待其结束的操作

MPI中的非阻塞消息传递

- int MPI_Irecv(void* buf,
  int maxsize,
  MPI_Datatype datatype,
  int source,
  int tag,
  MPI_Comm comm,
  MPI_Request* request);
  - <buf, maxsize, datatype> 指定消息的存储位置
  - source 指定消息来源的进程编号
  - tag 应与发送端指定的tag相对应
  - comm 指定进程组
  - request 与 MPI_Isend 的定义相同

- 注意：没有status_p 参数。与接收操作相关的状态信息由MPI_Test和MPI_Wait函数返回

MPI_Test

- MPI_Test用于测试非阻塞发送或接收操作是否已完成
  - 发送或接收操作通过具有MPI_Request类型的请求对象进行标识

- int MPI_Test(MPI_Request *request,
  int *flag,
  MPI_Status *status);
  - **request** 用于识别非阻塞操作，如果操作已经完成，请求对象将被释放
  - **flag** 将被置为 true (non-zero in C/C++) 如果非阻塞操作已经完成，否则被置为false (0 in C/C++)
  - **status** 将被设置为包含有关该操作的信息如果非阻塞操作已经完成

MPI_Wait

- MPI_Wait会阻塞，直到非阻塞操作（由参数标识）完成为止

- int MPI_Wait(MPI_Request *request,
  MPI_Status *status);
  - request 用于识别非阻塞操作
  - status 用于包含操作的信息

一个简单例子


```
 3 int main(int argc, char *argv[])
 4 {
 5     int myid, numprocs, tag;
 6     int buffer;
 7     MPI_Status status;
 8     MPI_Request request;
 9
10     MPI_Init(&argc,&argv);
11     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
12     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
13     tag=1234;
14     if(myid == 0){
15         buffer = 5678;
16         MPI_Isend(&buffer, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &request);
17     }
18     if(myid == 1){
19         MPI_Irecv(&buffer, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &request);
20     }
21     MPI_Wait(&request, &status);
22     if(myid == 0){
23         printf("Process %d  sent %d\n", myid, buffer);
24     }
25     if(myid == 1){
26         printf("Process %d  got %d\n", myid, buffer);
27     }
28     MPI_Finalize();
29 }
```

Process 0 向 process 1 发送一个整数

都使用非阻塞传输

阻塞与非阻塞

- 非阻塞通信操作可以与相应的阻塞操作匹配

- 例如，一个进程使用非阻塞发送操作发送消息，而另一个进程可以使用阻塞接收操作接收该消息。

派生数据类型

- 以前，MPI程序只能交换以连续数组存储的数据，而消息中的所有数据元素必须具有相同的类型。

- Motivations
  - 在如今的分布式内存系统中，通信相比于本地计算开销要大很多
  - 将固定数量的数据分成多个消息发送的成本，远高于将相同数量的数据以单个消息发送的成本。

- 在MPI中，派生数据类型是一种将多个数据整合到单个消息中的方法，即使它们在内存中不是连续存储的
  - 一旦创建了派生数据类型，你可以像其他基本的MPI数据类型一样使用它

派生数据类型

- 派生数据类型用于表示内存中数据项的集合，通过存储这些数据项的类型和它们在内存中的相对位置来实现
  - 思路：如果一个发送数据的函数知道有关一组数据项的信息，它可以在发送之前从内存中收集这些数据项
  - 类似地，一个接收数据的函数可以在数据接收时将数据项分发到它们在内存中的正确位置

派生数据类型的例子

- 假设一个进程需要广播以下三个变量
  - double a, b;
  - int n.
- 简单解决方法：调用MPI_Bcast 三次
- 或者创建一个新的数据类型，合并a，b，n 进入一条消息，然后调用一次
  - MPI中具有MPI_Datatype类型，我们尝试创建一个具有MPI_Datatype类型的变量
- 形式上，派生数据类型由一系列基本MPI数据类型以及位移组成。
  - 每一个数据类型称为一个块，可以包含有若干个数据元素

Variable	Address
a	24
b	40
n	48

displacement

0

16

24

包含有3个块，
每一个有一个元素

$$\{(\mathtt{MPI\_DOUBLE}, 0), (\mathtt{MPI\_DOUBLE}, 16), (\mathtt{MPI\_INT}, 24)\}$$

MPI_Type_create_struct

- MPI_Type_create_struct() 可以建立一个派生数据类型，包含有若干具有不同基础数据类型的独立元素
- int MPI_Type_create_struct(

```
int         count, /* in */
int         array_of_blocklengths[], /* in */
MPI_Aint    array_of_displacements[], /* in */
MPI_Datatype array_of_types[], /* in */
MPI_Datatype*  new_type_p /* out */)
```

  - count 指定块数量
  - array_of_blocklengths[] 存储每个块中的元素个数
  - array_of_displacement[] 存储每个块的移位（单位字节）
  - array_of_types[] 存储每个块中各个元素的 MPI 数据类型
  - new_type_p 是指向 MPI 派生数据类型的指针

MPI_Get_address

- 如何确定移位的大小？
  - 通过 MPI_Get_address() 获取每个元素的内存地址；
  - 通过减法计算移位

- MPI_Aint 是一个特殊 MPI 类型用来存储内存地址

- int MPI_Get_address(

  void* location_p, /* in */

  MPI_Aint* address_p /* out */)
  - location_p 是指向目标变量的指针
  - address_p 是一个存储目标变量地址的变量

MPI_Type_commit 与 MPI_Type_free

- 一个派生数据类型在使用前必需通过 MPI_Type_commit() 来提交
  - MPI_Type_commit() 允许 MPI 实现来优化派生数据类型的表示， 以便在通信函数中使用
- int MPI_Type_commit(MPI_Datatype* new_type_p);
- 新的数据类型在使用后通过 MPI_Type_free() 来释放其存储占用
- int MPI_Type_free(MPI_Datatype* old_type_p);

派生数据类型示例


```
 1 #include "mpi.h"
 2 #include <stdio.h>
 3 struct mystruct {
 4     char x;
 5     double y[6];
 6     int z[4];
 7 };
 8
 9 int main(int argc, char *argv[]) {
10     struct mystruct mydata[1000];
11     int i, j, myid;
12     MPI_Status status;
13     MPI_Datatype mytype;
14     MPI_Datatype type[3] = {MPI_CHAR, MPI_DOUBLE, MPI_INT};
15     int blocklen[3] = {1, 6, 4};
16     MPI_Aint disp[3];
17     MPI_Init(&argc, &argv);
18     disp[0] = &mydata[0].x - &mydata[0];
19     disp[1] = &mydata[0].y - &mydata[0];
20     disp[2] = &mydata[0].z - &mydata[0];
21     MPI_Type_create_struct(3, blocklen, disp, type, &mytype);
22     MPI_Type_commit(&mytype);
23     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```

C/C++ structure for a single data element.
We will convert it into an MPI derived datatype.

派生数据类型示例


```
24      if (myid == 0) {
25          /* the code of initializing mydata[] should be put here */
26          MPI_Send(mydata, 1000, mytype, 1, 0, MPI_COMM_WORLD);
27      } else if (myid == 1){
28          MPI_Recv(mydata, 1000, mytype, 0, 0, MPI_COMM_WORLD, &status);
29      }
30      MPI_Type_free (mytype);
31      MPI_Finalize();
32      return 0;
33  }
```

集合通信（Collective Communication）

- 两个进程间的点对点通信

- 集合通信是一种涉及通信组中所有进程参与的通信方法
  - 例子1：进程0需要向其他进行广播一个向量
  - 例子2：进程0需要从其他所有进程中收取信息并进行计算

动机

- 考虑一下如何将分属8个进程的8个数相加8
  - 方法一：

```
for( i = 1; i < 8; i++)
    Process i sends its number to process 0;
```

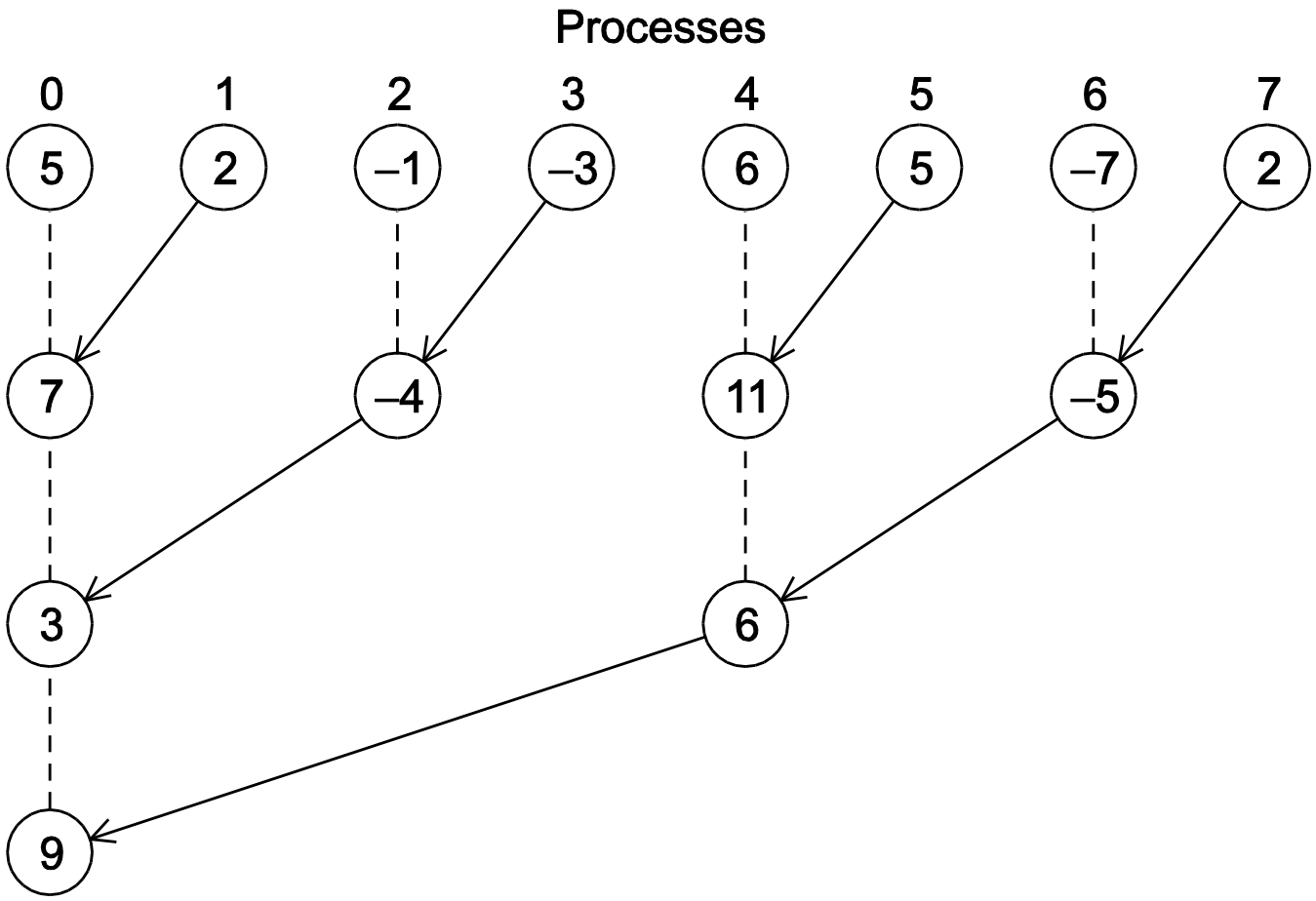
进程 0 分别将收到的 7 个数加到自己的数上

  - 分析：
    - 方法一涉及7次网络传输，所有的传输目标都是进程 0。在典型的集群环境中，这 7 次网络传输将按顺序进行（假设进程0所在的主机只有一个网络接口）

动机（续）

- 方法二
  - (1.a) 进程两两发送：1 -> 0,　3 -> 2,　5 -> 4,　7 -> 6
  - (1.b) 进程 0, 2, 4, 6 将收到的数加入自身
  - (2.a) 进程两两发送：　2 -> 0,　6 -> 4
  - (2.b) 进程 0, 4将收到的数加入自身
  - (3.a) 进程 4 将其最新值发送给 0：4 -> 0
  - (3.b) 进程 0将收到的数加入自身

方法二：树形全局相加



方法二分析

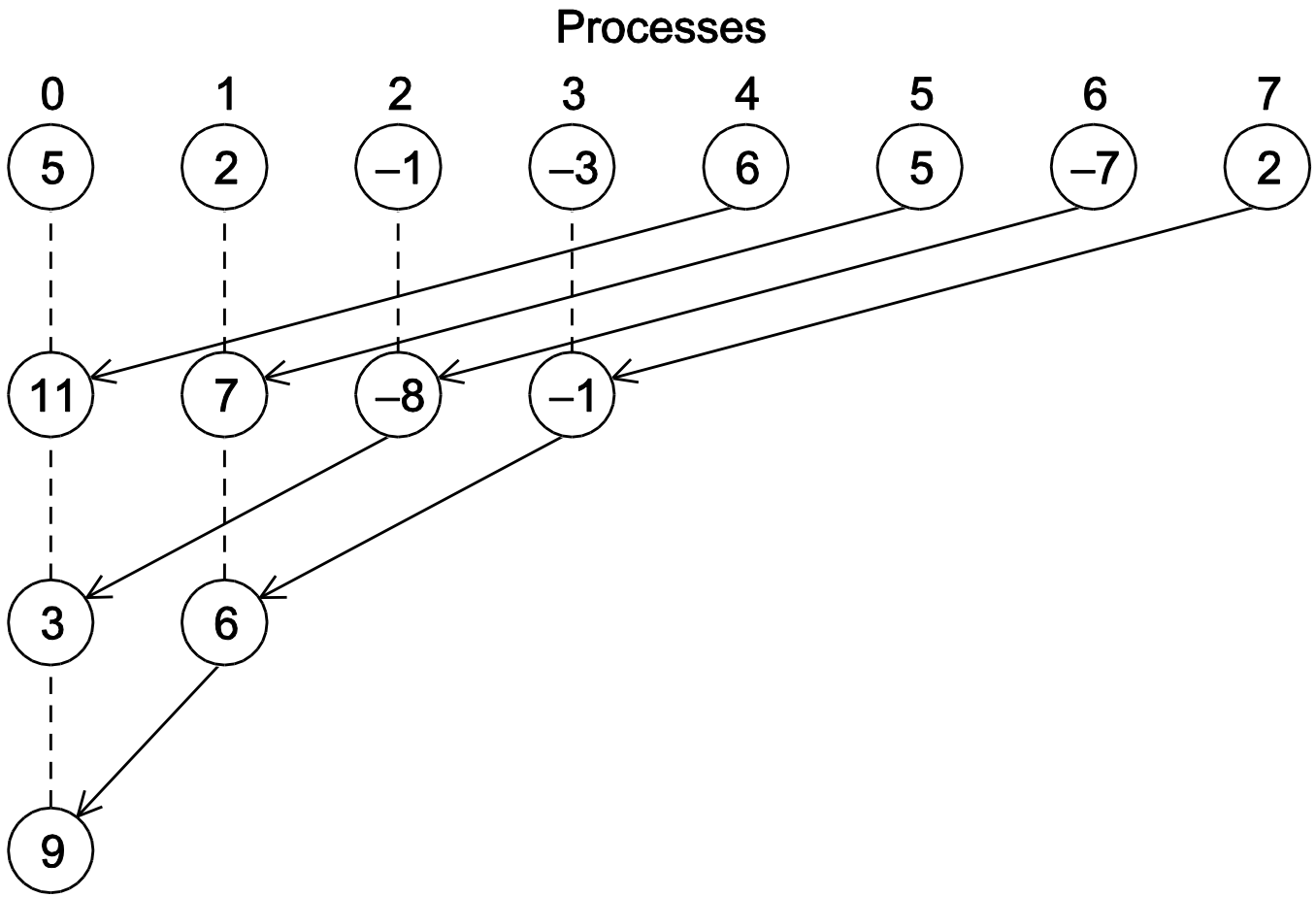
- 通过使用网络交换机，如果没有共同的源/目的地，可以同时进行多个数据通信

- 方法二同样需要总共7次数据通信

- 但这 7 次通信可以在 3 个时间步内完成
  - 因为部分通信可以并行执行

- 可见， n 个数相加仅需 $\log_2 n$ 个时间步，而方法一需要 n-1 个时间步

另一种备选方案

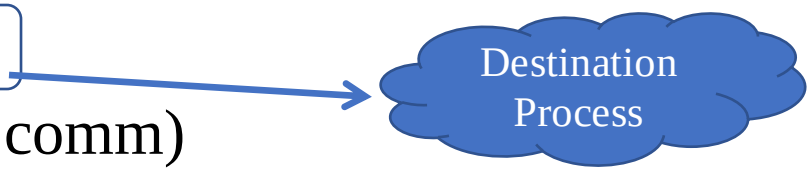


MPI 中的数据归约方法

- 前面的例子是一个典型的模式，称为“归约”(reduction)，即将一组消息合并成一个单一消息到一个进程组内。
  - 高效方法比一个简单方法好得多
  - 实现高效方法有很多不同实现方法
  - 一条消息可以包含多个数据元素。归约操作是逐个对每个数据元素应用的

MPI_Reduce

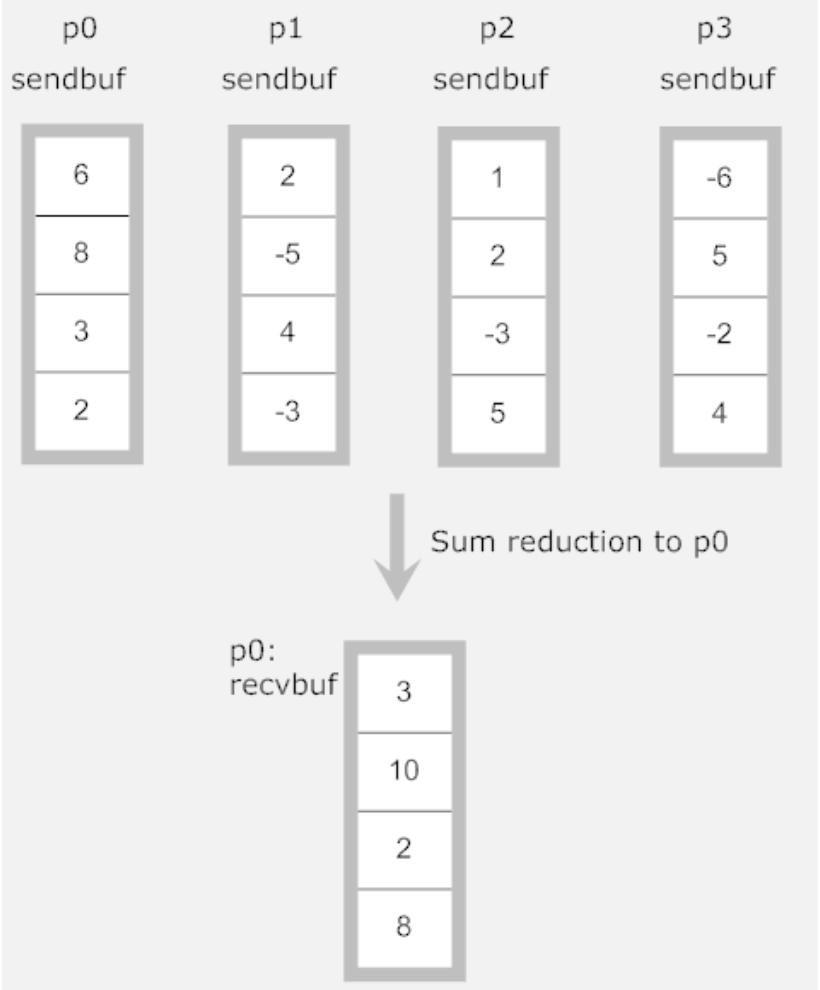
- MPI提供了一个经过优化的库函数MPI_Reduce，用于执行归约操作。归约的结果将会被存储在一个指定的“目标进程”中
- int MPI_Reduce(void *sendbuf,
  void *recvbuf,
  int count,
  MPI_Datatype datatype,
  MPI_Op operator,
  int root,
  MPI_Comm comm)



  - <sendbuf, count, datatype> 指定了 “message”
  - <recvbuf, root> 指定归约结果存储位置
  - operator 指定归约操作
  - comm 指定进程组

MPI_Reduce

- 归约操作是逐元素进行的
- 在进程0中，收缓存可以与发缓存是同一个吗？
  - 答案：NO!



MPI Reduction Operators

- 以下操作是MPI支持的：

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

集合通信和点对点通信

- 在MPI中，所有在同一个通信域（communicator）中的进程必须调用相同的聚合函数

- 例如，如果一个程序尝试在一个进程上调用MPI_Reduce，并在另一个进程上调用MPI_Recv来匹配它，这是错误的，而且很可能会导致程序挂起或崩溃

集合通信和点对点通信

- 每个进程传递给MPI集合通信的参数必须是“兼容的”
- 例如，如果一个进程将0作为根传递，而另一个进程将1作为根传递，那么对MPI_Reduce的调用结果是错误的，程序很可能会挂起或崩溃

集合通信和点对点通信

- recvbuf 参数只用在目标进程 dest_process 上
- 尽管如此，所有的进程仍然需要传递一个与 recvbuf 对应的实参，即使它只是 NULL

集合通信和点对点通信

- 点对点通信是基于标签（tags）和通信域（communicators）来匹配的
- 集合通信不使用 tags
- 点对点通信的匹配是仅基于通信器和调用顺序进行的

自我测试

- 假设每个进程都调用 MPI_Reduce，进行 MPI_SUM 操作，且指定目标进程为进程0
  - 进程0中 b 和 d 的最终值分别是多少？

Time	Process 0	Process 1	Process 2
0	a = 1; c = 2	a = 1; c = 2	a = 1; c = 2
1	MPI_Reduce(&a, &b, ...)	MPI_Reduce(&c, &d, ...)	MPI_Reduce(&a, &b, ...)
2	MPI_Reduce(&c, &d, ...)	MPI_Reduce(&a, &b, ...)	MPI_Reduce(&c, &d, ...)

  - MPI_Reduce(void *sendbuf, void *recvbuf, int count, …)

一些主要的集合通信操作

- MPI_Reduce
- MPI_Bcast
- MPI_Scatter
- MPI_Gather

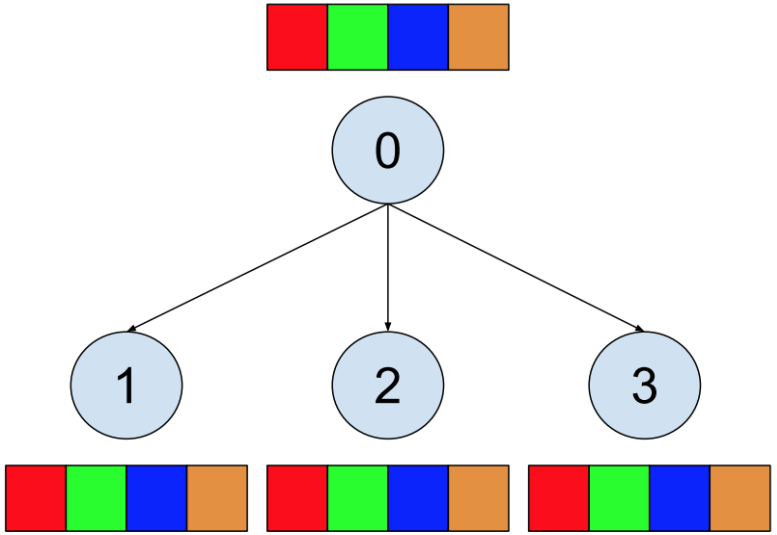
- MPI_Allgather
- MPI_Allreduce
- MPI_Alltoall
- MPI_Reduce_scatter

- MPI_Barrier

MPI_Bcast

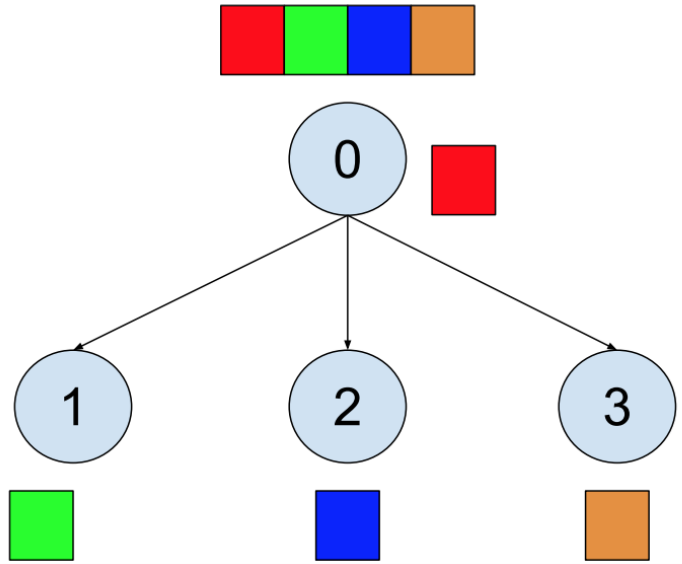
- 数据属于单个进程的情况下，可以使用广播（Broadcast）操作将数据发送到进程组中的所有进程

- int MPI_Bcast(void *buf,
  int count,
  MPI_Datatype datatype,
  int root,
  MPI_Comm comm)
  - <buf, count, datatype> 指定“message”
  - root 指定源进程的进程号
  - comm 指定进程组



MPI_Scatter

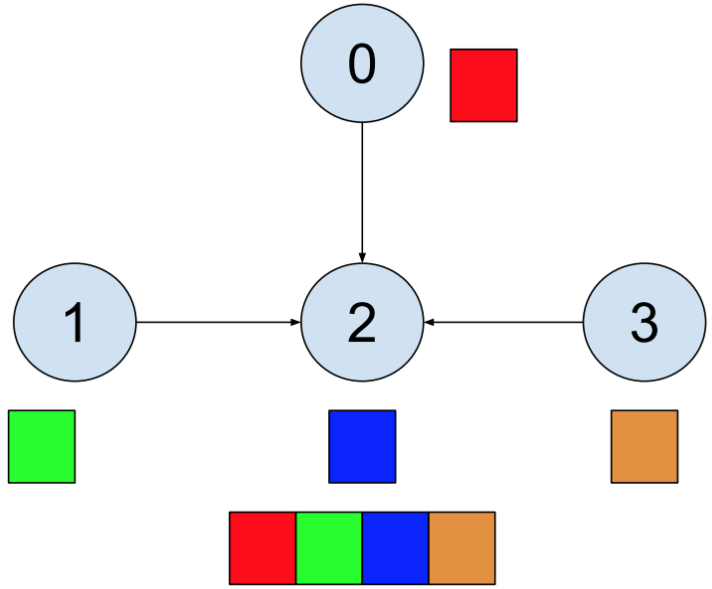
- MPI_Scatter 是一种聚合通信操作，它将属于根进程的数据向量均匀分割成若干份，并将每一块数据发送给其他进程中的对应位置
  - 份数等于进程组中进程的个数
  - 假设数据项数可以被进程数整除



- int MPI_Scatter(void *sendbuf, int sendcount,
  MPI_Datatype senddatatype,
  void *recvbuf, int recvcount,
  MPI_Datatype recvdatatype,
  int root, MPI_Comm comm)
  - <sendbuf, sendcount, senddatatype> 指定“source message”，其中 sendcount 指的是发送到每个进程的数据量，而不是发送缓存中的总数据
  - <recvbuf, recvcount, recvdatatype> 指定了 “received message”，其中 recvcount 是指每个进程收到的数据量
  - root 指定源进程的编号
  - comm 指定进程组

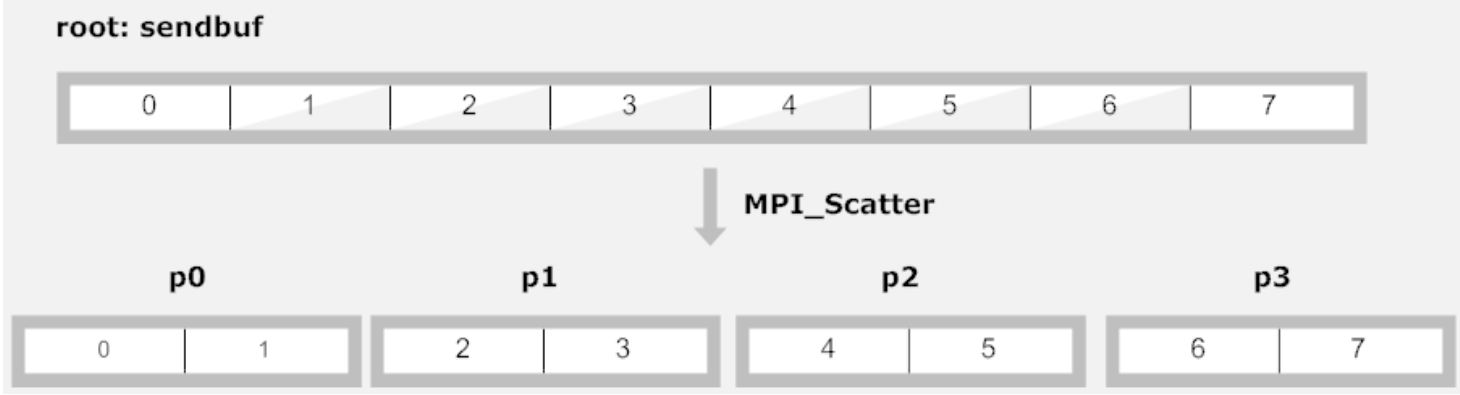
MPI_Gather

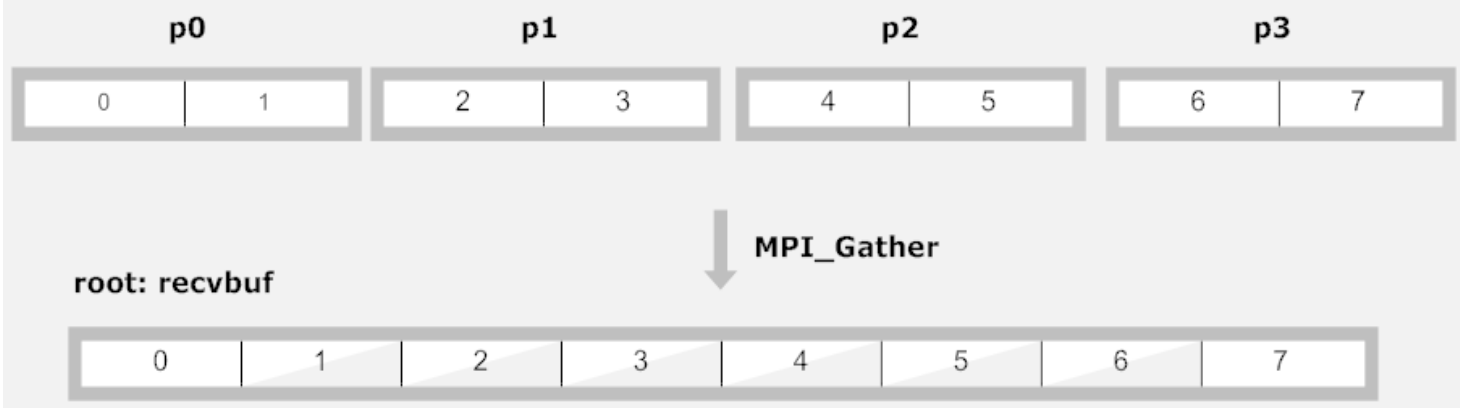
- MPI_Gather 从进程组中的各个进程上收集数据到一个进程上
  - 要求每个进程上的数据量相同



- int MPI_Gather(void *sendbuf,
  int sendcount,
  MPI_Datatype senddatatype,
  void *recvbuf,
  int recvcount,
  MPI_Datatype recvdatatype,
  int root,
  MPI_Comm comm)
  - <sendbuf, sendcount, senddatatype> 指定源消息，sendcount 指定每个进程发出的数据量
  - <recvbuf, recvcount, recvdatatype> 指定收到的消息，recvcount 指定从每个进程上收到的数据量
  - root 指定目标进程的编号
  - comm 指定进程组

MPI_Scatter 与 MPI_Gather





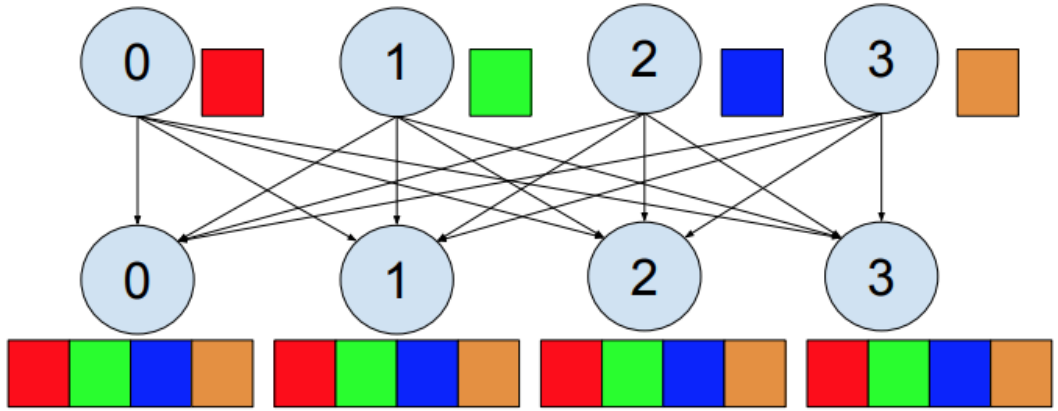
MPI_Scatter例子：分发一个数组


```
 4      int n, local_n, my_rank, comm_sz;
 5      MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
 6      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
 7      if (my_rank == 0) {
 8          scanf("%d", &n);
 9          MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
10      } else {
11          MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
12      }
13      local_n = n / comm_sz;
14      local_a = malloc(local_n * sizeof(double));
15      if (my_rank == 0) {
16          a = malloc(n * sizeof(double));
17          for (i = 0; i < n; i++)
18              scanf("%lf", &a[i]);
19          MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
20      } else {
21          MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
22      }
```

MPI_Allgather

- MPI_Allgather 连接各个进程的数据，并存储到各进程中

- int MPI_Allgather(void *sendbuf,
  int sendcount,
  MPI_Datatype senddatatype,
  void *recvbuf,
  int recvcount,
  MPI_Datatype recvdatatype,
  MPI_Comm comm)

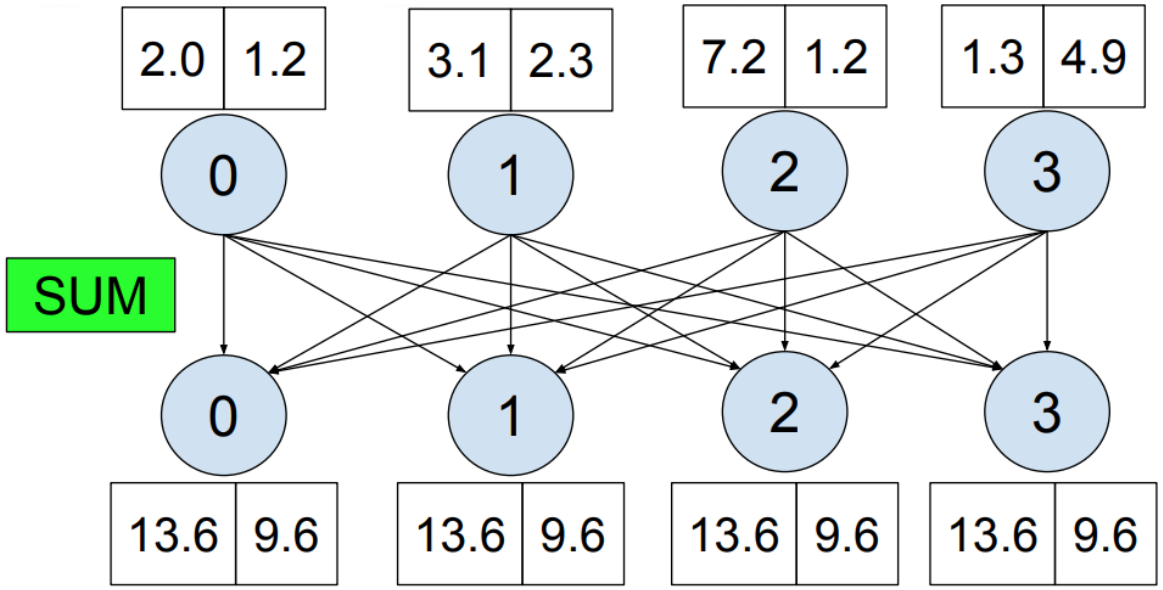


  - <sendbuf, sendcount, senddatatype> 指定源消息，sendcount 表示每个进程发出的数据量
  - <recvbuf, recvcount, recvdatatype> 指定接收到的消息，recvcount 表示从各个进程收到的数据量
  - comm 指定进程组

MPI_Allreduce

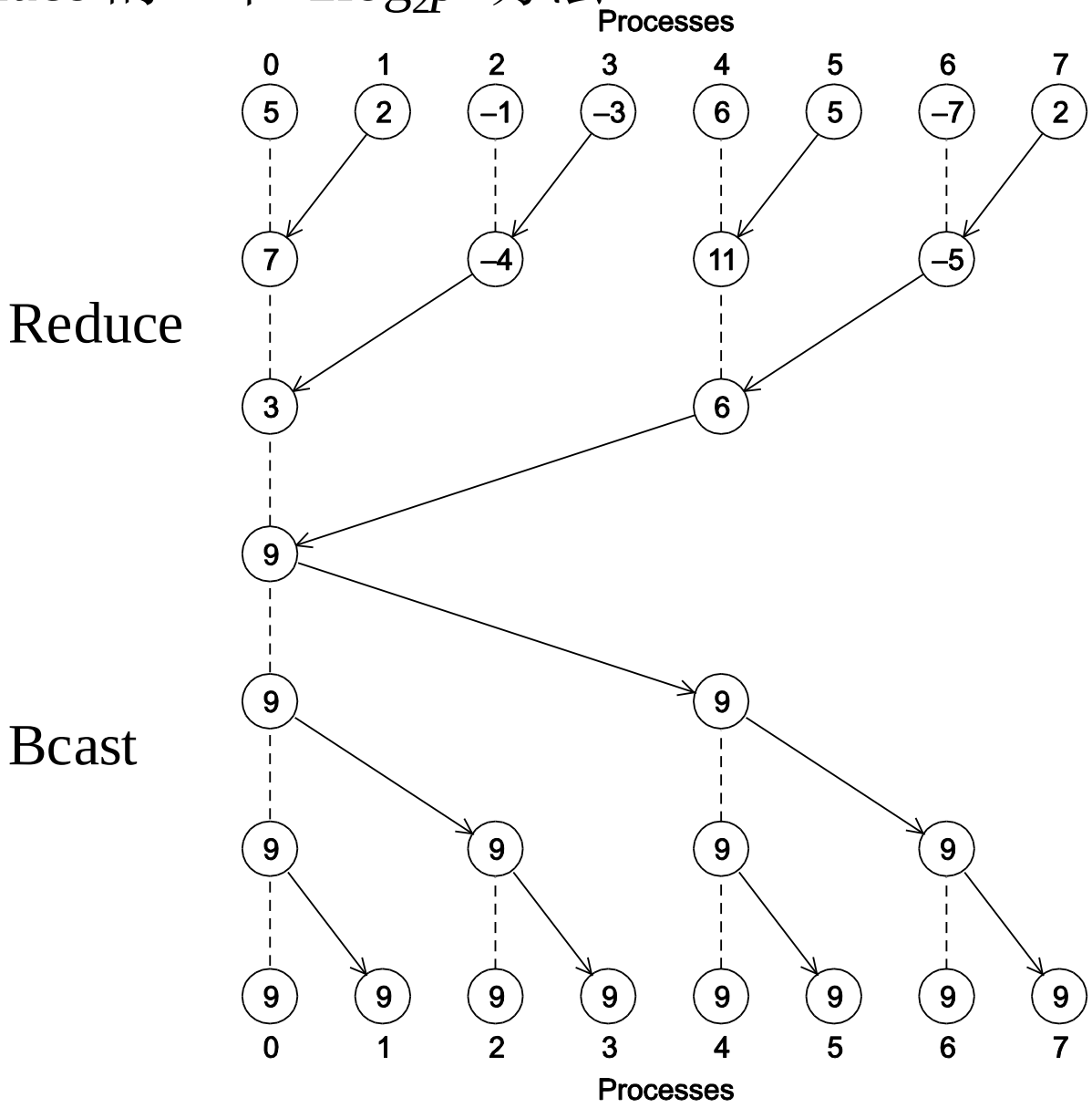
- int MPI_Allreduce(void *sendbuf,
  void *recvbuf,
  int count,
  MPI_Datatype datatype,
  MPI_Op operator,
  MPI_Comm comm)

- 与MPI_Reduce 相似，但归约得到的结果被存储到所有进程上



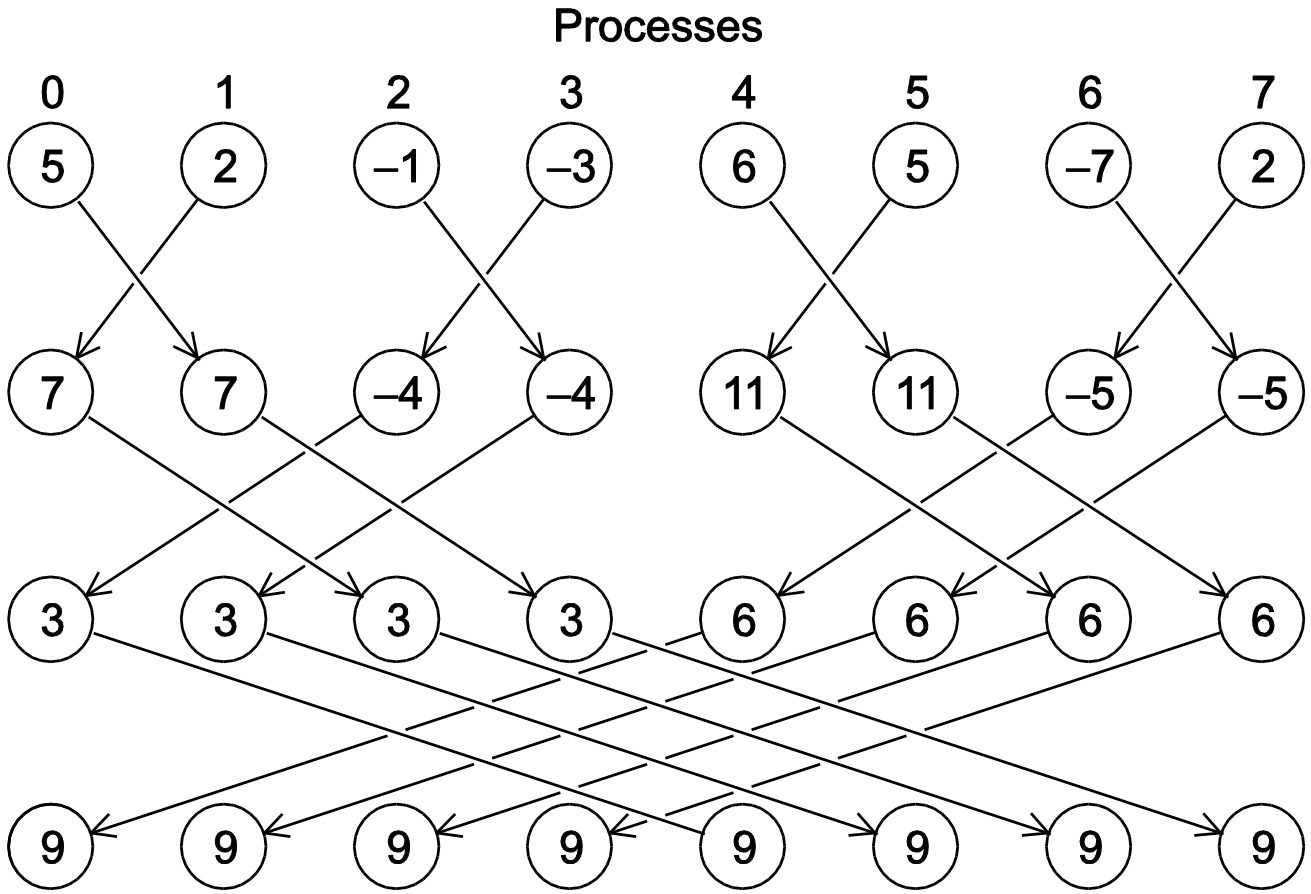
MPI_Allreduce：算法一

- MPI_Allreduce 的一个 $2\log_2 p$ 方法：



MPI_Allreduce：算法二

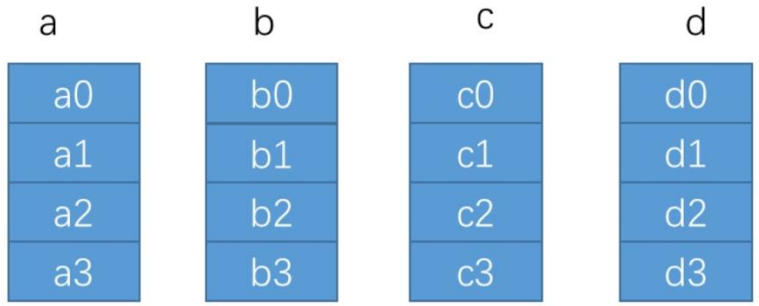
- $\log_2 p$ 方法：蝶形架构(butterfly-structure, or Rabenseifner's algorithm)
- 利用了当今的网卡和网络交换机支持同时发送和接收操作的特性，即全双工

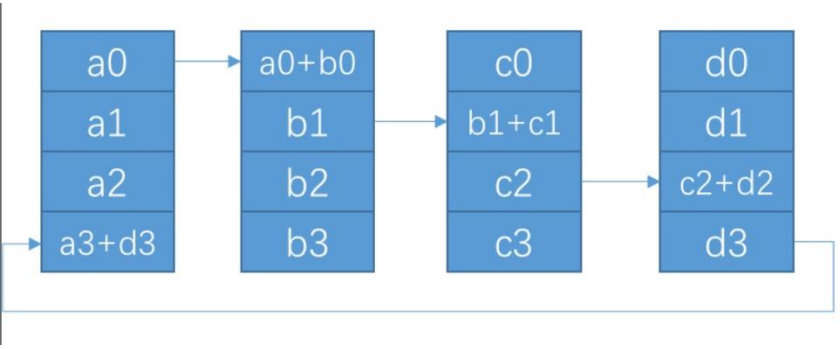


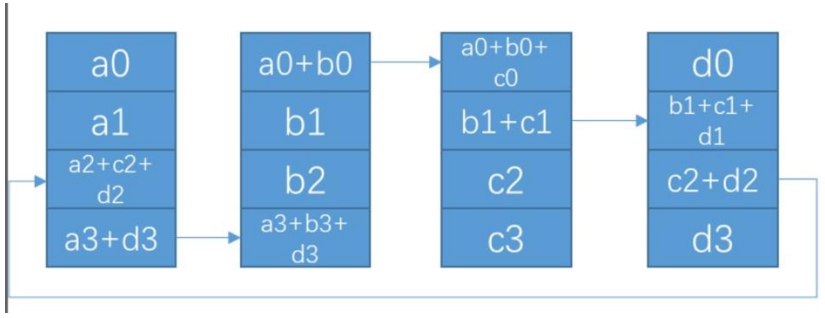
MPI_Allreduce：算法三

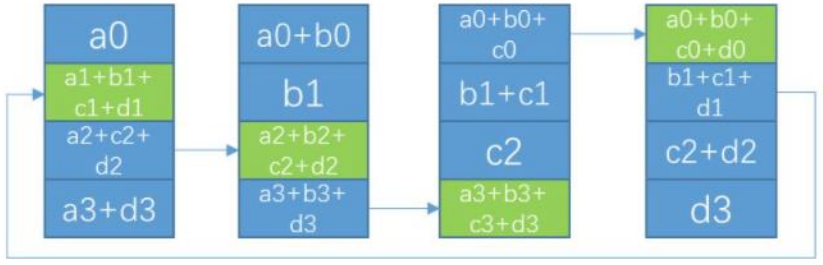
- 带宽友好型：ring-based

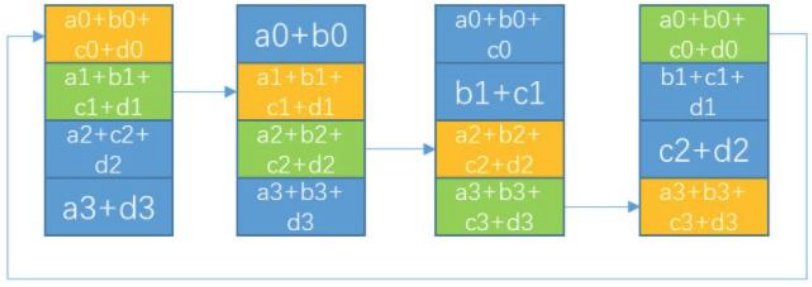
$$2(p-1)\alpha + 2(p-1)\beta m/p$$

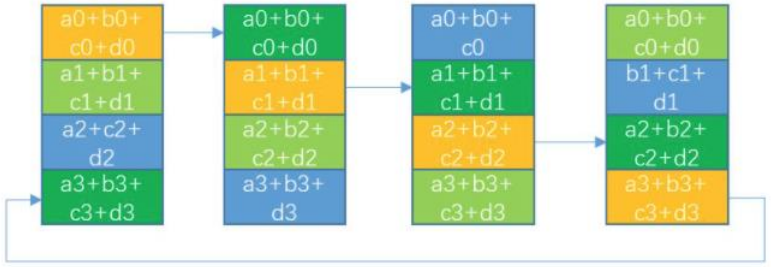


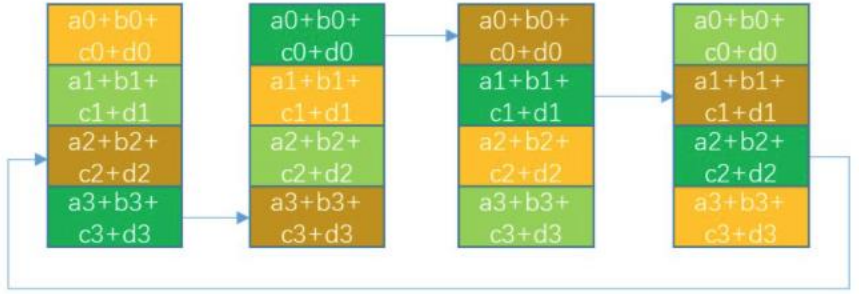






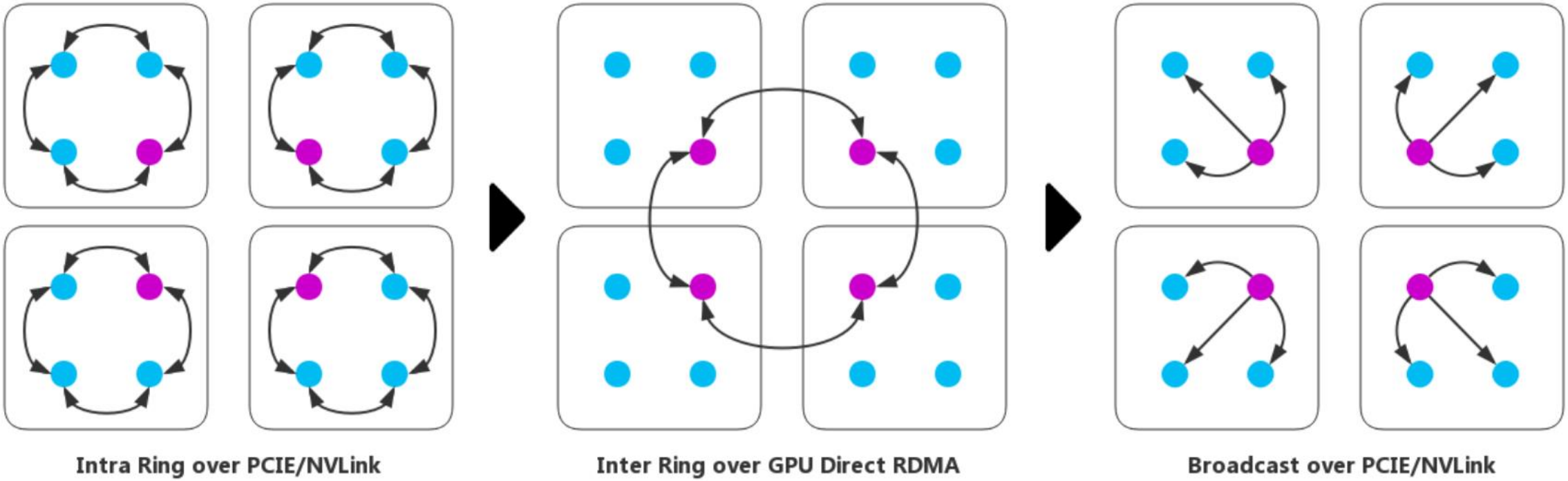






Allreduce：算法四

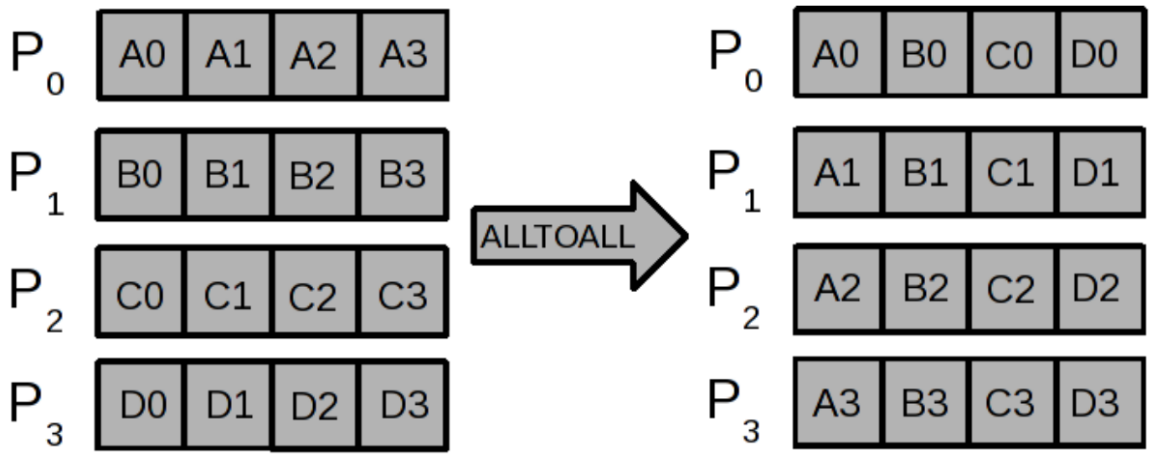
- 平衡延迟与带宽算法：hierarchical Allreduce [1]



[1] Jia et al., Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes, *NIPS Workshop 2018*

MPI_Alltoall

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



- 相当于转置
- 相当于同时的scatters 或 gathers 操作

MPI_Alltoall 例子

- 假设有包括root在内的四个进程，每个都有 u 和 v 两个数组
- 在 all-to-all 操作后

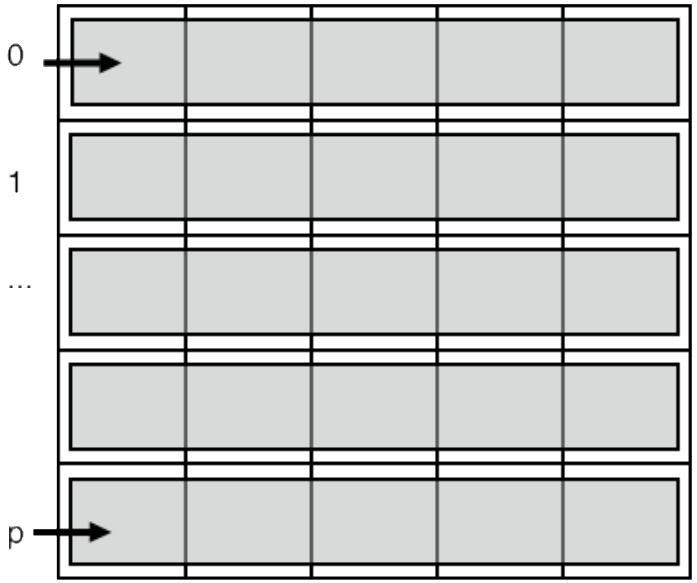
MPI_Alltoall(u, 2, MPI_INT, v, 2, MPI_INT, MPI_COMM_WORLD);

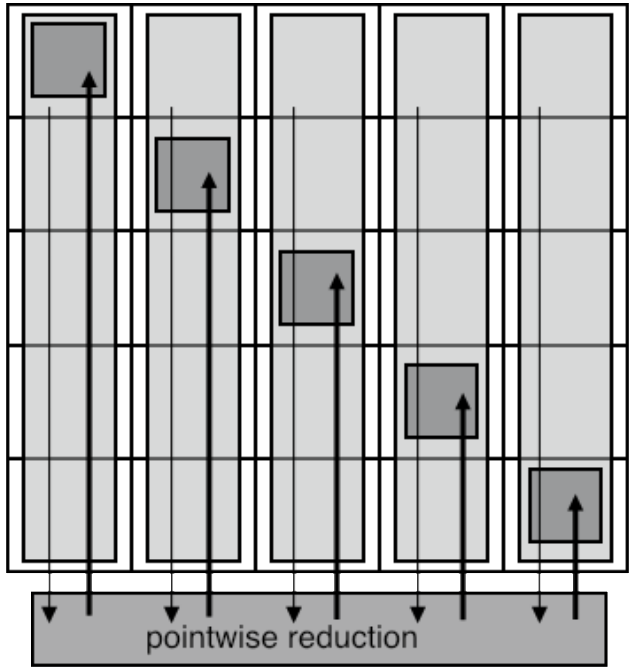
array u								Rank	array v							
10	11	12	13	14	15	16	17	0	10	11	20	21	30	31	40	41
20	21	22	23	24	25	26	27	1	12	13	22	23	32	33	42	43
30	31	32	33	34	35	36	37	2	14	15	24	25	34	35	44	45
40	41	42	43	44	45	46	47	3	16	17	26	27	36	37	46	47

Extension: https://developer.nvidia.com/blog/doubling-all2all-performance-with-nvidia-collective-communication-library-2-12/

MPI_Reduce_scatter

- int MPI_Reduce_scatter(void *sendbuf, void *recvbuf,
  const int recvcounts[], MPI_Datatype datatype,
  MPI_Op op, MPI_Comm comm)
  - *Sendbuf*：发送缓存的起始地址
  - *Recvbuf*：接收缓存的起始地址
  - *Recvcounts*：整数数组，指定结果中分发到个线程的元素个数，各调用进程中的数组必须是完全相同的





MPI_Barrier

- 屏障是一种常用于一组进程或线程中的同步方法
  - 每个成员需要在屏障处等待，直到组中的所有成员都到达屏障为止
- MPI_Barrier 实现了屏障同步操作
  - MPI_Barrier调用后直到组里所有进程都调用才会返回
- int MPI_Barrier(MPI_Comm comm)

矩阵-向量乘法：Scatter和Gather


```
 4 int main(int argc, char** argv)
 5 {
 6     int m = 0, n = 0, myid, numprocs, i;
 7     int srow = 0;
 8     double *A, *x, *y, *local_A, *local_y;
 9     double start, end;
10
11     MPI_Init(&argc, &argv);
12     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
13     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
14
15     if(myid == 0) {
16         while ( m <= 0 || n <= 0 || m%numprocs != 0) {
17             printf("Please input positive integers m and n: ");
18             scanf("%d %d", &m, &n);
19         }
20         A = (double*) malloc( m * n * sizeof(double) );
21         x = (double*) malloc(n * sizeof(double) );
22         y = (double*) malloc(m * sizeof(double) );
23         init_array(A, m * n);
24         init_array(x, n);
25     }
```

使用进程0对数据做预处理

矩阵-向量乘法：分散和收集

广播m和n

内存分配

分散矩阵 A

收集向量 y

```
27      MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
28      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
29
30      srow = m / numprocs;
31
32      local_A = (double*) malloc( srow * n * sizeof(double) );
33      local_y = (double*) malloc( srow * sizeof(double) );
34      if(myid != 0)     x = (double*) malloc(n * sizeof(double) );
35
36      MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
37
38      MPI_Scatter(A, srow * n, MPI_DOUBLE, local_A, srow * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
39
40      Mat_vect_mul(local_A, x, local_y, srow, n);
41
42      MPI_Gather(local_y, srow, MPI_DOUBLE, y, srow, MPI_DOUBLE, 0, MPI_COMM_WORLD);
43      if (myid == 0) { free(A); free(y); }
44      free(local_A); free(x); free(local_y);
45      MPI_Finalize();
46      return 0;
47  }
```

一个例子

- 用高斯消去法求解线性方程组

$$A x = b$$

- 它曾经被用于评估 TOP500 超算
  - http://www.top500.org/project/linpack/

线性方程组

- 考虑求解下列的线性方程组：

$$
\begin{array}{ccccccc}
a_{0,0}x_0 & + & a_{0,1}x_1 & + & \cdots & + & a_{0,n-1}x_{n-1} = b_0, \\
a_{1,0}x_0 & + & a_{1,1}x_1 & + & \cdots & + & a_{1,n-1}x_{n-1} = b_1, \\
\vdots & & \vdots & & & & \vdots \\
a_{n-1,0}x_0 & + & a_{n-1,1}x_1 & + & \cdots & + & a_{n-1,n-1}x_{n-1} = b_{n-1}.
\end{array}
$$

- 这可以被写作 $\boldsymbol{Ax} = \boldsymbol{b}$，其中 $\boldsymbol{A}$ 是 $n \times n$ 的矩阵，$\boldsymbol{A}[i, j] = a_{i,j}$，$\boldsymbol{b}$ 是一个 $n \times 1$ 的向量 $[\, b_0, b_1, \ldots, b_{n-1} \,]^{\mathrm{T}}$，$\boldsymbol{x}$ 是解

方法：高斯消元法

- 两步：(1) 化简 **A** 为上三角形式；(2) 回代

- 上三角的形式为：

$$\begin{aligned} x_0 + u_{0,1}x_1 + u_{0,2}x_2 + \cdots + u_{0,n-1}x_{n-1} &= y_0, \\ x_1 + u_{1,2}x_2 + \cdots + u_{1,n-1}x_{n-1} &= y_1, \\ &\vdots \qquad\qquad\qquad\quad \vdots \\ x_{n-1} &= y_{n-1}. \end{aligned}$$

- 我们将其写为：$\boldsymbol{Ux} = \boldsymbol{y}$. 我们选择的是一个单位上三角矩阵 $\boldsymbol{U}$，它的对角线元素都是 1

- 将一个给定矩阵转换为上三角矩阵的常用方法就是：高斯消去法

高斯消元法示例

$$
\begin{aligned}
4x_0 &+6x_1 &+2x_2 &-2x_3 &= 8\\
2x_0 & & +5x_2 &-2x_3 &= 4\\
-4x_0 &-3x_1 &-5x_2 &+4x_3 &= 1\\
8x_0 &+18x_1 &-2x_2 &+3x_3 &= 40
\end{aligned}
$$

高斯消元法示例

$$
\begin{array}{llllll}
x_0 & +1.5x_1 & +0.5x_2 & -0.5x_3 & = & 2 \\
2x_0 & & +5x_2 & -2x_3 & = & 4 \\
-4x_0 & -3x_1 & -5x_2 & +4x_3 & = & 1 \\
8x_0 & +18x_1 & -2x_2 & +3x_3 & = & 40
\end{array}
$$

高斯消元法示例

$$
\begin{array}{lllll}
x_0 & +1.5x_1 & +0.5x_2 & -0.5x_3 & = & 2 \\
 & -3x_1 & +4x_2 & -1x_3 & = & 0 \\
 & +3x_1 & -3x_2 & +2x_3 & = & 9 \\
 & +6x_1 & -6x_2 & +7x_3 & = & 24
\end{array}
$$

高斯消元法示例

$$
\begin{aligned}
x_0 \quad & +1.5x_1 \quad +0.5x_2 \quad -0.5x_3 \quad = \quad 2 \\
x_1 \quad & -4/3x_2 \quad +1/3x_3 \quad = \quad 0 \\
& +3x_1 \quad -3x_2 \quad +2x_3 \quad = \quad 9 \\
& +6x_1 \quad -6x_2 \quad +7x_3 \quad = \quad 24
\end{aligned}
$$

高斯消元法示例

$$
\begin{aligned}
x_0 \quad & +1.5x_1 \quad +0.5x_2 \quad -0.5x_3 \quad = \quad 2 \\
x_1 \quad & -4/3x_2 \quad +1/3x_3 \quad = \quad 0 \\
& 1x_2 \quad +1x_3 \quad = \quad 9 \\
& 2x_2 \quad +5x_3 \quad = \quad 24
\end{aligned}
$$

高斯消元法示例

$$
\begin{aligned}
x_0 \quad & +1.5x_1 \quad +0.5x_2 \quad -0.5x_3 \quad = \quad 2 \\
x_1 \quad & -4/3x_2 \quad +1/3x_3 \quad = \quad 0 \\
x_2 \quad & +x_3 \quad = \quad 9 \\
3x_3 \quad & = \quad 6
\end{aligned}
$$

高斯消元法示例

$$
\begin{aligned}
x_0 \quad & +1.5x_1 \quad +0.5x_2 \quad -0.5x_3 \quad = \quad 2 \\
x_1 \quad & -4/3x_2 \quad +1/3x_3 \quad = \quad 0 \\
x_2 \quad & +x_3 \quad = \quad 9 \\
x_3 \quad & = \quad 2
\end{aligned}
$$

回代

$$x_3 = 2$$

$$x_2 = 9 - x_3 = 7$$

$$x_1 = 4/3x_2 - 1/3x_3 = 26/3$$

$$x_0 = 2 - 1.5x_1 - 0.5x_2 + 0.5x_3 = -27/2$$

顺序高斯消元法

- 顺序高斯消元法将 ***Ax=b*** 转化为 ***Ux=y***

Input: ***A***, ***b***

Output: ***U***, ***y***

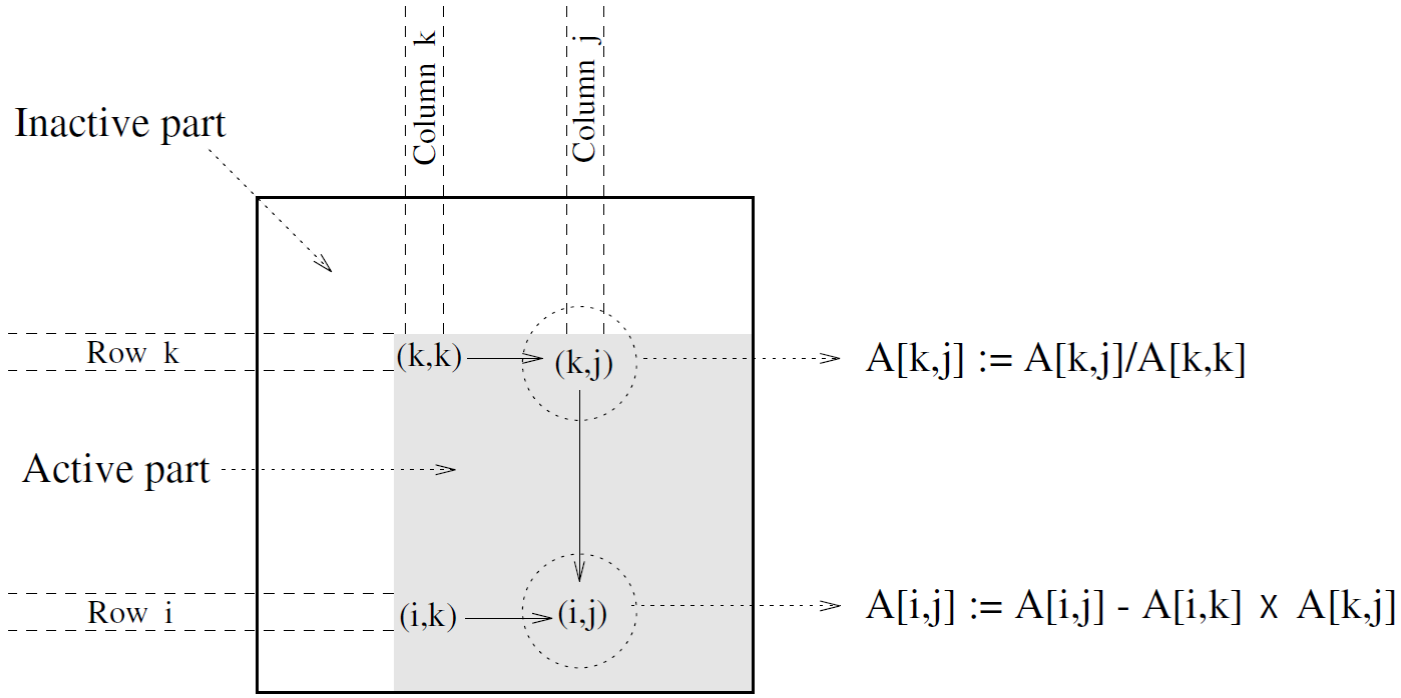
U is stored in the upper-triangular locations of ***A***.

```
1.  procedure GAUSSIAN_ELIMINATION (A, b, y)
2.  begin
3.      for k := 0 to n − 1 do                    /* Outer loop */
4.      begin
5.          for j := k + 1 to n − 1 do
6.              A[k, j] := A[k, j]/A[k, k];   /* Division step */
7.          y[k] := b[k]/A[k, k];
8.          A[k, k] := 1;
9.          for i := k + 1 to n − 1 do
10.         begin
11.             for j := k + 1 to n − 1 do
12.                 A[i, j] := A[i, j] − A[i, k] × A[k, j]; /* Elimination step */
13.             b[i] := b[i] − A[i, k] × y[k];
14.             A[i, k] := 0;
15.         endfor;                   /* Line 9 */
16.     endfor;                       /* Line 3 */
17. end GAUSSIAN_ELIMINATION
```

Source: Algorithm 8.4 of Ref. [1]

高斯消元法的经典计算

- 计算过程有 3 个嵌套循环
  - 在外层循环的第 k 次迭代，算法需要进行 $(n-k)^2$ 次计算
  - K 从 1 加到 n，要进行大约 $n^3/3$ 次“乘−减”运算，或者 $2n^3/3$ 次运算



Source: Figure 8.5 of Ref. [1]

回代

- 用顺序回代方法求解上三角方程组 ***Ux=y***
  - ***U*** 是一个单位上三角矩阵
  - 它大约进行 $n^2/2$ 次乘减运算

```
1.   procedure BACK_SUBSTITUTION (U, x, y)
2.   begin
3.       for k := n − 1 downto 0 do   /* Main loop */
4.           begin
5.               x[k] := y[k];
6.               for i := k − 1 downto 0 do
7.                   y[i] := y[i] − x[k] × U[i, k];
8.           endfor;
9.   end BACK_SUBSTITUTION
```

Source: Algorithm 8.5 of Ref. [1]

小测试

- 天河-2 是世界上最快的超算之一 (6th, 2020)
  - http://www.top500.org/system/177999

- 它的Linkpack性能 (i.e., R_{max}) 是 33,862.7 TFlop/s，就是通过求解9,960,000元线性方程组 (i.e., N_{max}) 测得的

- 能否估计出天河-2 求解该方程组的耗时是多少？

部分主元消元法

- 在之前的方法中，我们隐含地假设了$A[k, k]$ 不为零或接近零
  - 为什么？

- 部分主元消元法（通过列交换）
  - 在第 k 次迭代中外层循环开始时，选择第 i 列满足 $\boldsymbol{A}[k, i]$ 是 $\boldsymbol{A}[k, j], k \leq j < n$ 中最大的，我们称第 i 列为主元列
  - 在开始迭代前交换第 k 列和第 i 列

- 部分主元消元法（通过行交换）

并行高斯消元法：一维分割

- 我们可以在行维度上对系数矩阵 A 进行分割.

- 设每一行被分配给一个进程
  - 一个 n x n 矩阵需要 n 进程
  - 进程 i (P_i) 初始存储数据 $\boldsymbol{A}[i, j], 0 \leq j < n$

- 如何并行化 P68 的算法8.4?

并行高斯消元法：一维分割

- 在外层循环上(第3行)，每一次循环都依赖前次循坏(一般来说)
  - 这个层级很难并行化

- 内层循环中(9-15行)，每一行的消去步骤是相互独立的
  - 这些计算可由不同进程并行

例子：$n = 8, k = 3$

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Source: Figure 8.6 (a) of Ref. [1]

Lines 5-8:

```
for j := k+1 to n-1 do
    A[k,j] := A[k,j] / A[k,k];
y[k] := b[k] / A[k,k];
A[k,k] := 1;
```

上述代码仅由进程3 P_3 执行

不需要通信

例子：$n = 8, k = 3$

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Source: Figure 8.6 (b) of Ref. [1]

计算完 5-8 行后，数据 $A[k, j]$, $k<j<n$, 需要被广播到 $P_4 - P_7$

我们称这些数据激活数据 **“active data”**

例子：$n = 8, k = 3$

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Source: Figure 8.6 (c) of Ref. [1]

9-15 行：

```
for i := k+1 to n-1 do
begin
   for j := k+1 to n-1 do
      A[i,j] -= A[i,k] * A[k,j];
   b[i] -= A[i,k] * y[k];
   A[i,k] := 0;
end
```

这部分代码由 P_4-P_7 并行执行

不需要通信

分析具有 n 个处理进程的情况

- 第一步是归一化各行，这是一个序列化操作，在第 k 次迭代中耗时为(n-k-1)
- 第二步，将归一化行中的激活数据广播到(n-k-1)处理进程中，耗时

$$(t_s + t_w(n-k-1))\log n$$

- 每个进程独立从自身消去这一行，这需要(n-k-1)次乘减运算
- 最后相加，第 k 次迭代需要 3(n-k-1) 次计算和通信

$$(t_s + t_w(n-k-1))\log n$$

分析具有 n 个处理进程的情况

- 总的并行计算时间由 $k = 0 \ldots n\text{-}1$ 累加而来

$$T_P = \frac{3}{2}n(n-1) + t_s n \log n + \frac{1}{2}t_w n(n-1)\log n.$$

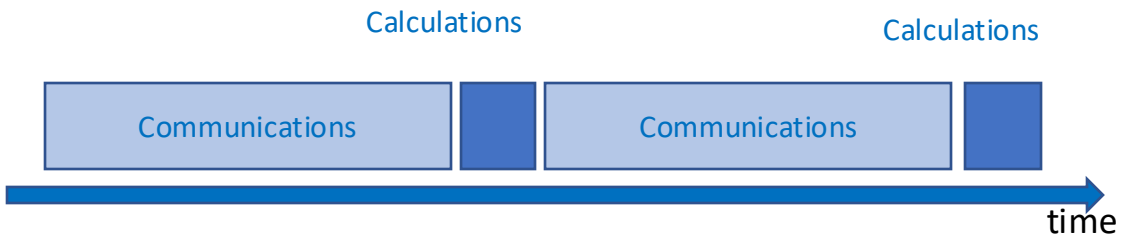
- 总耗时为 nT_p，时间复杂度为：

$$\Theta(n^3 \log n)$$

- 该方法不是代价最优的！

两个低效因素

- 通信开销
  - 第 k 次迭代中，每个进程花费 $\Theta(n\text{-}k)$ 时间在并行计算中，但通信时间为 $\Theta(n\text{-}k)\log_2 n$！



- 负载不均衡
  - 进程 P_i 在 $(i+1)$ 次迭代后完全空转
  - 进程 P_o 的工作负载最小，为 $\Theta(n)$；但 $P_{n\text{-}1}$ 的工作负载最大，达到 $\Theta(n^2)$

如何改进？

- 前述方法，第$(k+1)$次迭代只在第 k 次迭代的计算和通信完成后才开始
  - 是必须的吗？
- 在每一次迭代中，计算总在通信完成后开始
  - 是必须的吗？
- 要回答这个问题，我们需要对序列化的算法 8.4 进行深入分析

算法8.4深入分析

- 第 k 次迭代中，进程 P_k 将它的激活数据广播到其他进程 $P_{k+1}, \ldots, P_{n-1}$
  - 这可以通过顺序传递来实现：$P_k \rightarrow P_{k+1}$, $P_{k+1} \rightarrow P_{k+2}$, …, $P_{n-2} \rightarrow P_{n-1}$

- 对于进程 P_{k+1} 来说，在传递完激活数据到 P_{k+2} 后，不需要等待所有进程都收到，可以直接开始自己这部分的消去计算了(第12行)
  - 可以将通讯和计算重叠进行

- 当进程 P_{k+1} 完成它第 k 轮迭代的消去计算后，可以直接开始为第(k+1)轮迭代进行除法计算了(第6行)；然后它可以向进程 P_{k+2} 发送它的激活数据
  - 可以将第 k+1 次迭代与第 k 相重叠！而这又可以进一步重叠通信和计算，以及同时通信

- 下一步我们通过一个 5x5 的例子展示一个流水线 “pipelined” 并行方法

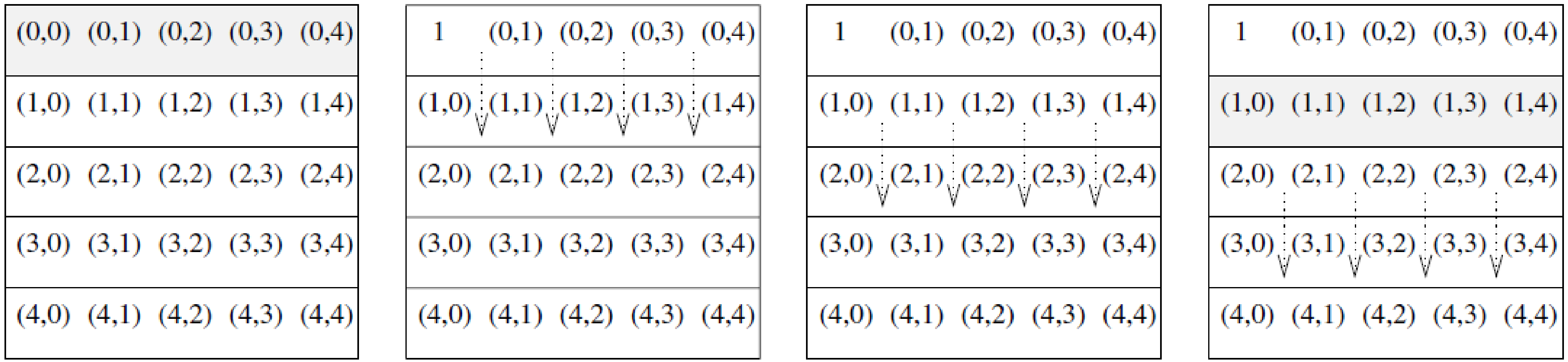
5x5 矩阵示例

(a) P_0 计算第 0 行的除法运算

(b) P_0 将它的激活数据传递给 P_1

(c) P_1 将它的激活数据传递给 P_2

(d) P_1 执行消去；同时， P_2 传递它的激活值给 P_3



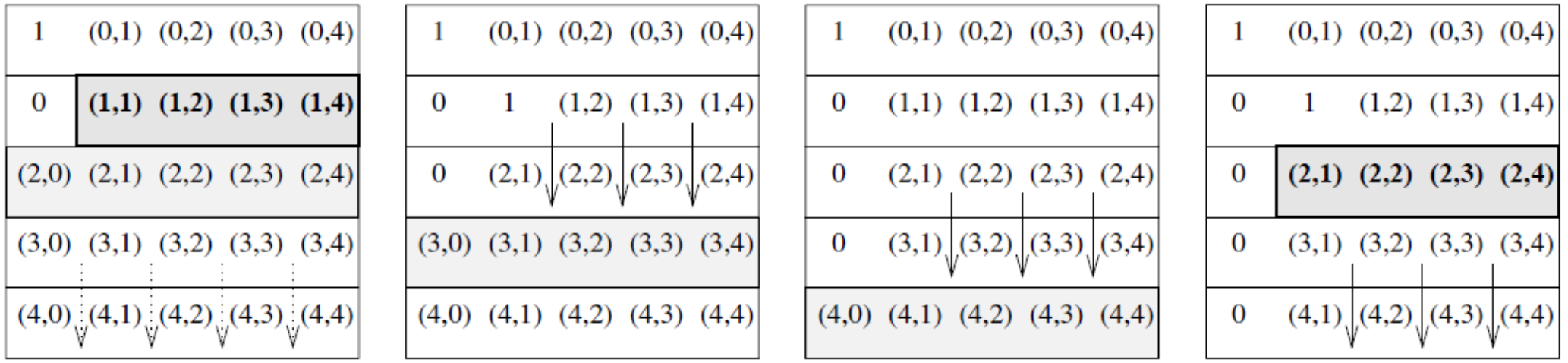
(a) Iteration k = 0 starts (b) (c) (d)

Source: Figure 8.7 (a-d) of Ref. [1]

5x5 矩阵示例

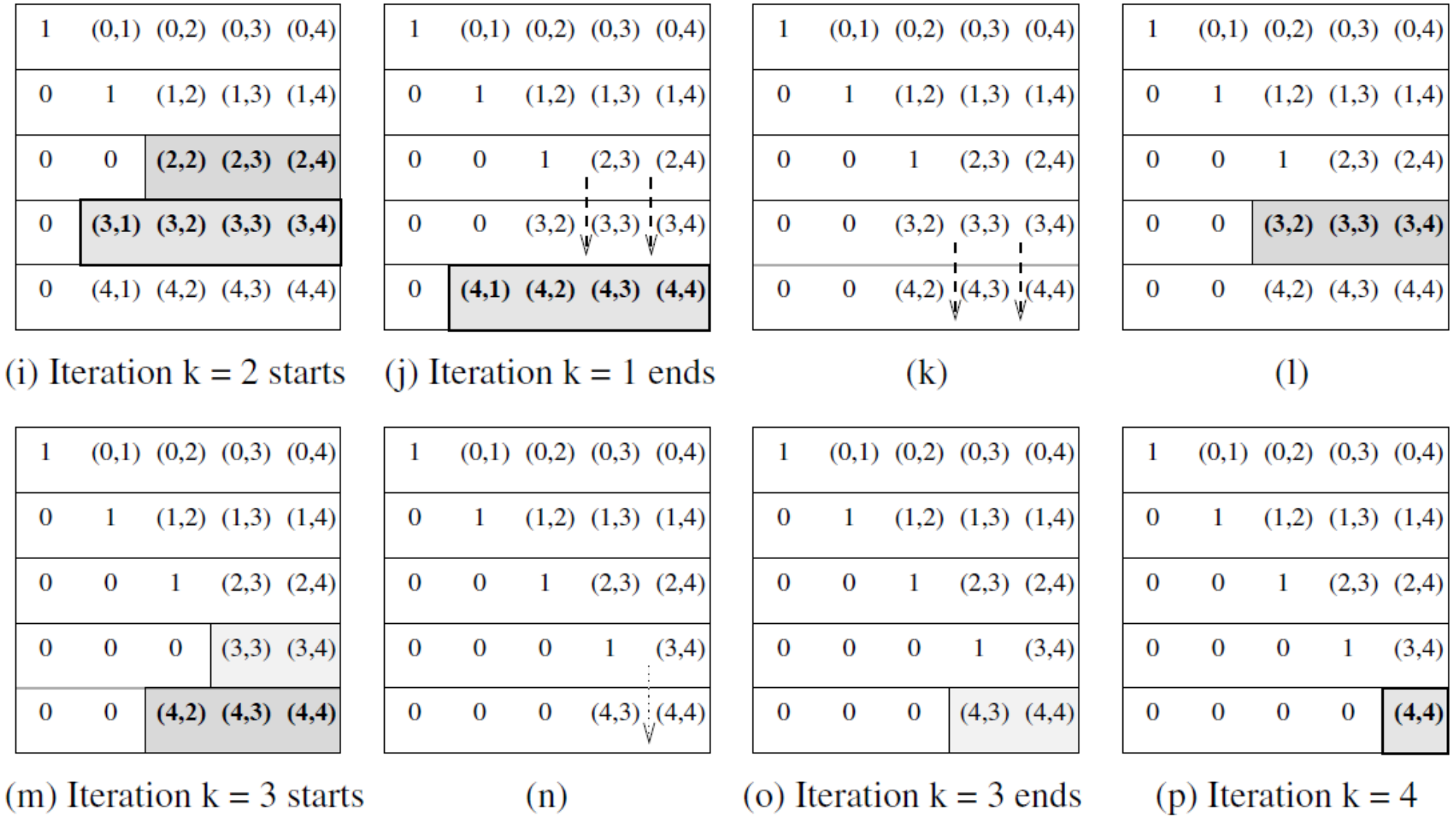
(e) P_2 执行消去；同时，P_3 传递它的激活值给 P_4；然后 P_1 计算第一行的除法运算 —— 第 1 次迭代在第 0 次迭代还未完成时就开始了

(f-p) 自行推理剩下部分



Source: Figure 8.7 (e-h) of Ref. [1]

5x5 矩阵示例



Source: Figure 8.7 (i-p) of Ref. [1]

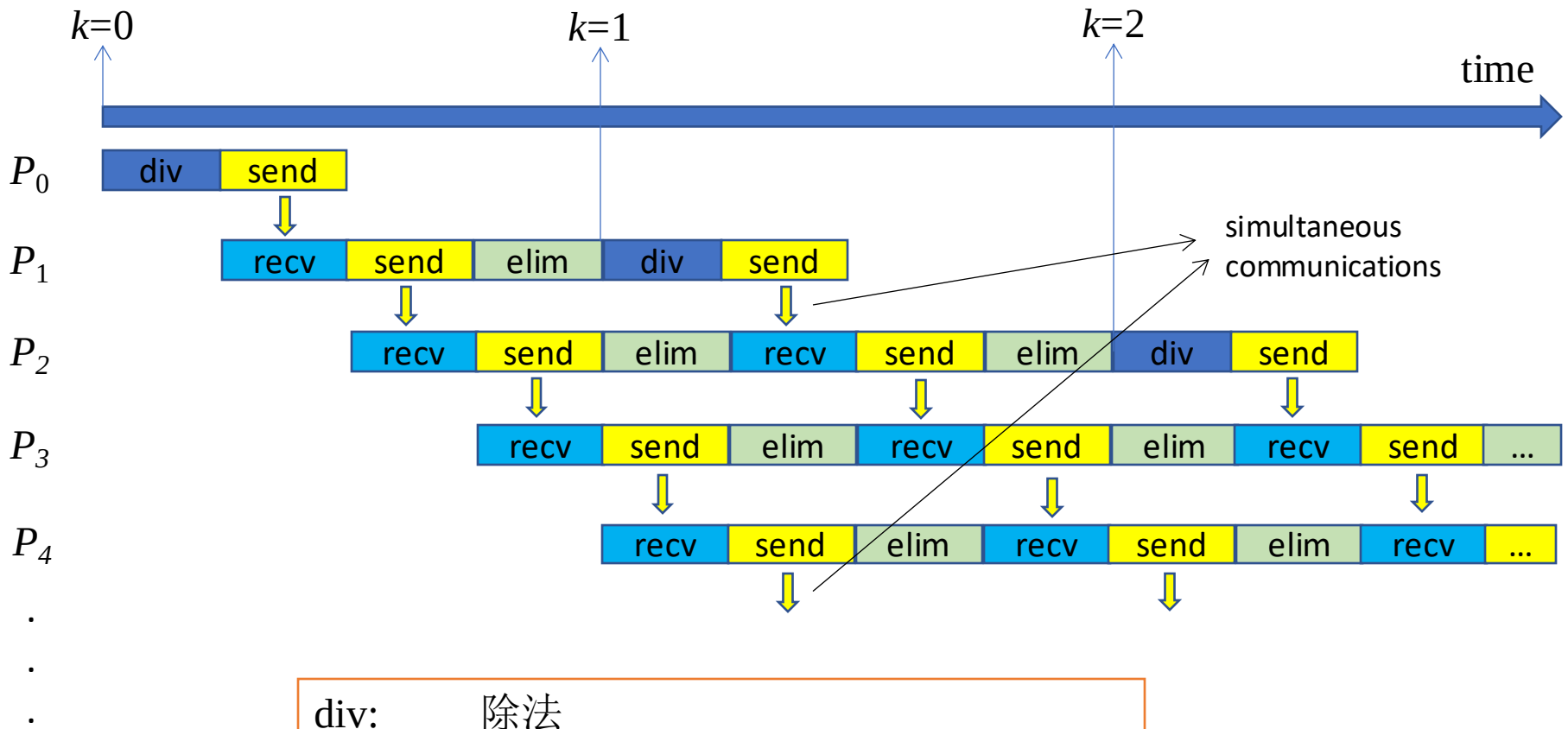
流水线并行方法

- 在流水线版本中，每个进程 P_i 做如下的事：

```
step = 0;
while (step < i) {
    Receive the "active data" from P_{i-1};
    Send the "active data" to P_{i+1} ;
    Perform the elimination ;
    step++;
}
perform the division;
send the active data to P_{i+1};
```

流水线方法分析

注意：我们这里先假设 div, recv, send, elim 这些操作的耗时相同



div:	除法
send:	向下一个进程发送数据
recv:	从前一个进程接收数据
elim:	消去

流水线方法分析

- 从前面的流水线图示中来看， 两次迭代间隔的时间为 $\Theta(n)$
- 总共有 n 次迭代，因此最后一次迭代从 $\Theta(n^2)$ 开始
- 最后一次迭代本身耗时 $\Theta(1)$
- 因此并行时间为 $\Theta(n^2)$，它是代价较小的方法

回到现实：p < n

- 之前，我们假设线程总数 $p = n$
- 在大多数情况下，线程数远小于 n
- 行一维块分割：每个进程被分配 n/p 行连续的矩阵块

一维块分割

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_1	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_2	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_3	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Source: Figure 8.8 of Ref. [1]

在第 3 次迭代中，P_1 上的激活数据需要被发送到 P_2 和 P_3

与 $p = n$ 的情况相比，通信开销缩减了 n/p 倍

当 $n >> p$，计算开销占支配地位

负载均衡问题

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_1	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
	0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
P_2	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_3	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(a) Block 1-D mapping

Source: Figure 8.9 (a) of Ref. [1]

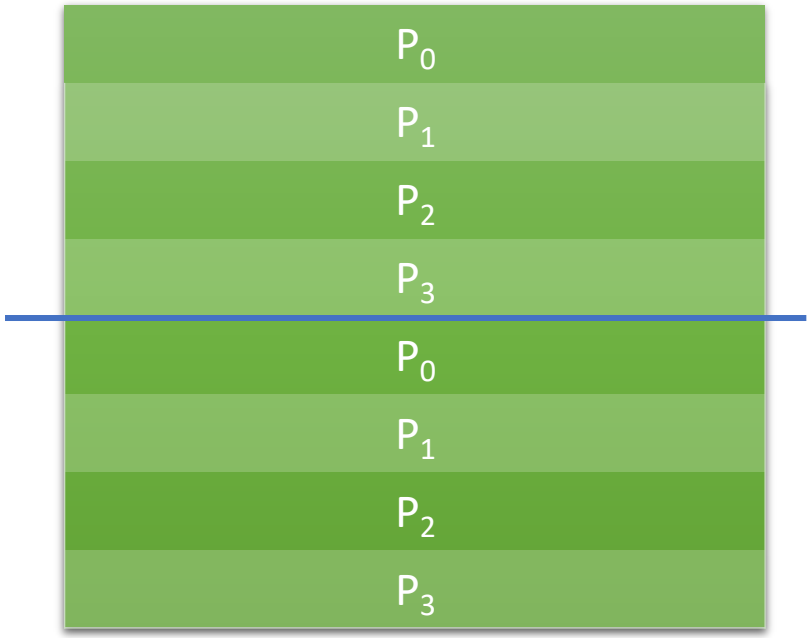
一维块分割的开销为 n^3，顺序执行算法的开销为 $2n^3/3$

这是 p 个线程间不平衡的负载分配导致的

左边的例子中，P_0 是完全空转的，P_1 部分负载，P_2 和 P_3 是满负载的

循环一维映射

- 循环映射 可以用于负载均衡
- 在每一次迭代中， 计算会被分配到所有线程



1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)	P_0
0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	
0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	P_1
0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	
0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	P_2
0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	
0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	P_3
0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	

(b) Cyclic 1-D mapping

Source: Figure 8.9 (a) of Ref. [1]

References

1. Ananth Grama, George Karypis, Vipin Kumar, Anshul Gupta, Introduction to Parallel Computing, 2nd Edition, Addison Wesley, 2003.

2. Peter S. Pacheco, An Introduction to Parallel Programming, Morgan Kaufmann, 2010.

3. William Gropp, Ewing Lush, Anthony Skjellum, Using MPI, 2nd Edition, The MIT Press, 1999.

4. C. Renggli et al., SparCML: High-Performance Sparse Communication for Machine Learning, SC, 2019