清華大學
Tsinghua University

高性能计算导论
第4讲：消息传递编程模型-2

翟季冬

计算机系

目录

- 点对点通信和缓冲区
- 点对点通信实现协议
- 通信死锁
- 进程映射
- 集合通信算法实现
- MPI 程序性能分析

点对点通信和缓冲区

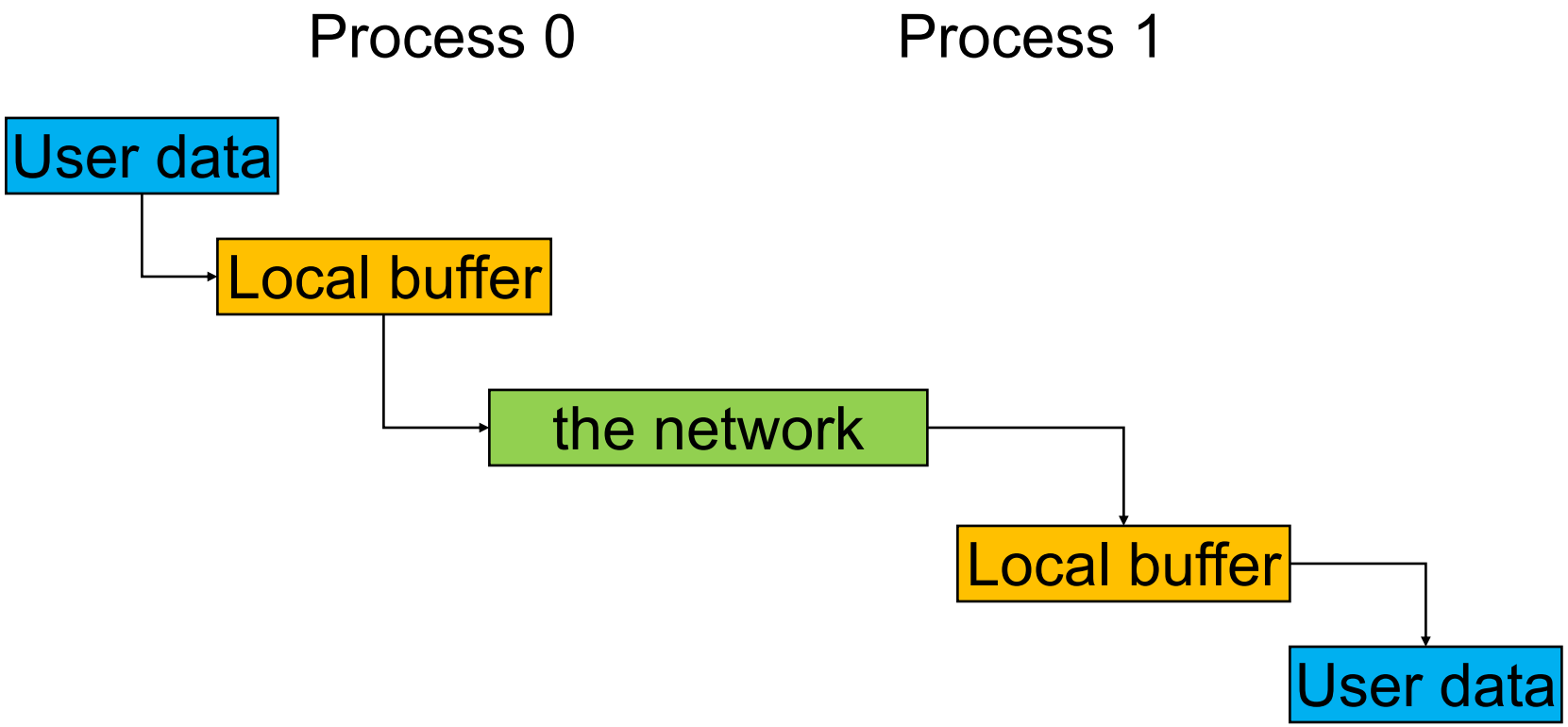
点对点通信缓冲区

阻塞发送	MPI_Send(buffer, count, type, dest, tag, comm)
非阻塞发送	MPI_Isend(buffer, count, type, dest, tag, comm, request)
阻塞接收	MPI_Recv(buffer, count, type, source, tag, comm, status)
非阻塞接收	MPI_Irecv(buffer, count, type, source, tag, comm, request)

- **buffer**：表示发送/接收数据的地址空间

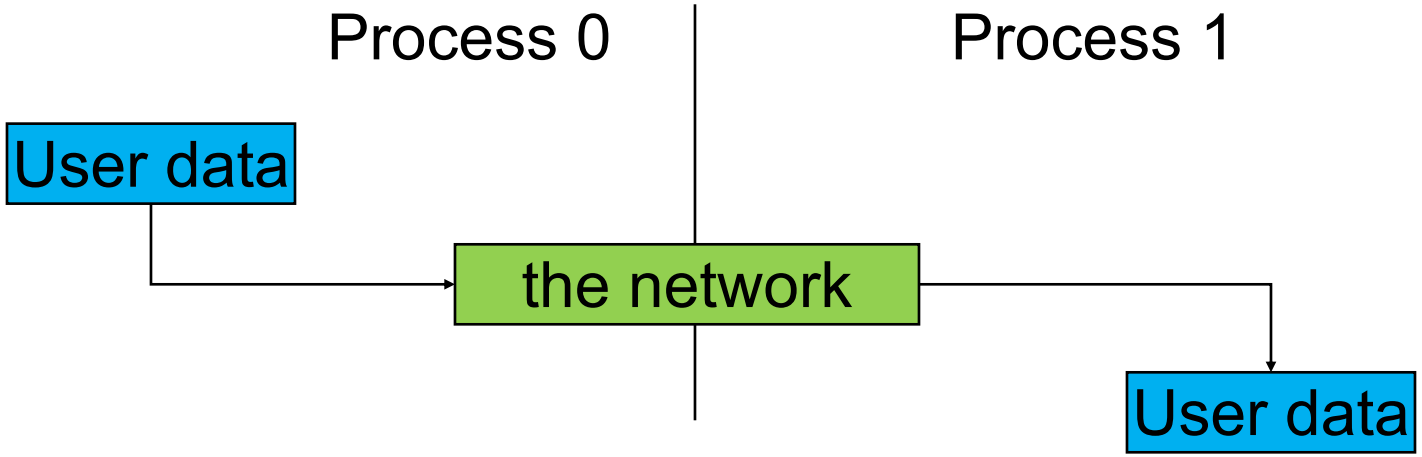
缓冲区

- 使用MPI发送数据时，数据会去哪里？
- 一种可能如下（Local buffer 也叫 System buffer）
- 潜在的问题是什么？



避免缓冲

- 避免到系统缓冲区的拷贝能够有效减少存储的使用
- 采用如下方式进行传输
- 潜在的问题是什么?



- 这要求发送进程在传输时等待（阻塞通信）
  - 或者发送进程在传输完成之前就返回，稍后再等待（非阻塞通信）

阻塞通信

- 阻塞通信的完成
  - MPI_Send 直到发送“buffer为空”之后才可以返回
    - 发送 buffer 可以再次被使用
  - MPI_Recv 直到消息被完整地接收之后才可以返回
    - 接收消息的 buffer 已经可用
- 阻塞通信的完成取决于
  - 传输消息的大小
  - 系统缓冲区大小

非阻塞通信

- 非阻塞通信会立即返回
- 需要 request handles 来进行等待（wait），等到操作完成

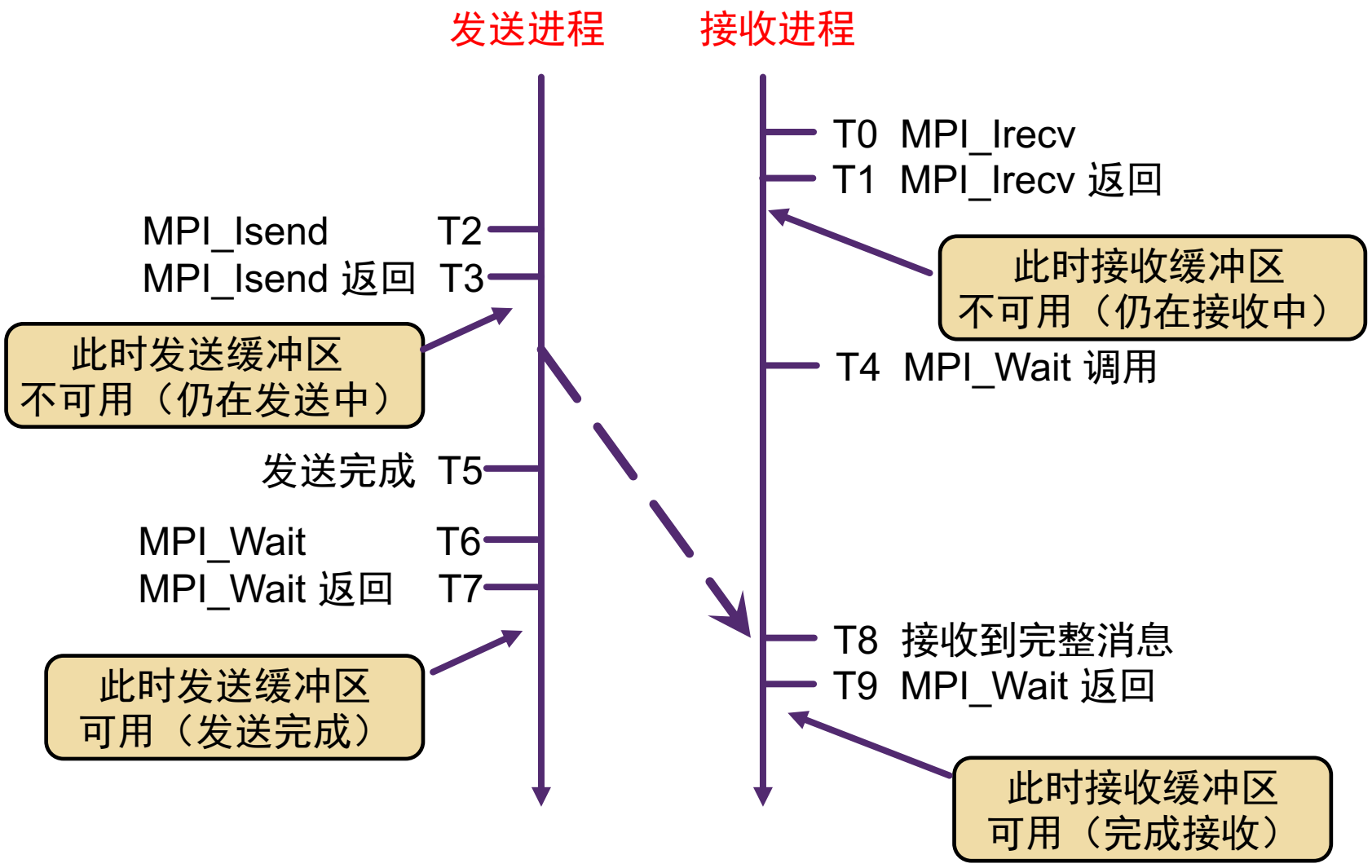
```
MPI_Request request;
MPI_Status status;
MPI_Isend(start, count, datatype, dest, tag, comm, &request);
```

或

```
MPI_Irecv(start, count, datatype, dest, tag, comm, &request);
MPI_Wait(&request, &status);
```

- 可以使用测试操作确认通信是否完成（MPI_Test）而不等待
  - `MPI_Test(&request, &flag, &status);`
- 在没有等到操作结束就访问数据缓冲区的行为是未定义的（危险！）

非阻塞通信示例


发送进程
接收进程
T0 MPI_Irecv
T1 MPI_Irecv 返回
MPI_Isend T2
MPI_Isend 返回 T3
此时接收缓冲区
不可用（仍在接收中）
此时发送缓冲区
不可用（仍在发送中）
T4 MPI_Wait 调用
发送完成 T5
MPI_Wait T6
MPI_Wait 返回 T7
T8 接收到完整消息
T9 MPI_Wait 返回
此时发送缓冲区
可用（发送完成）
此时接收缓冲区
可用（完成接收）

非阻塞通信示例 — 计算通信重叠

Non-Blocking send	MPI_ISend(buffer,count,type,dest,tag,comm,request)
Non-Blocking receive	MPI_IRecv(buffer,count,type,source,tag,comm,requst)

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);/* find process rank */
if (myrank == 0) {
   int x=10;
   MPI_Isend(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, req1);
   compute();
} else if (myrank == 1) {
   int x;
   MPI_Irecv(&x, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, req1);
}
MPI_Wait(req1, status);
```

等待多个非阻塞通信

- 有时需要同时等待多个requests：
  - `MPI_Waitall(count, array_of_requests, array_of_statuses)`
  - `MPI_Waitany(count, array_of_requests, &index, &status)`
  - `MPI_Waitsome(count, array_of_requests, array_of_indices, array_of_statuses)`

- 以上每个函数也有对应的检测（Test）函数

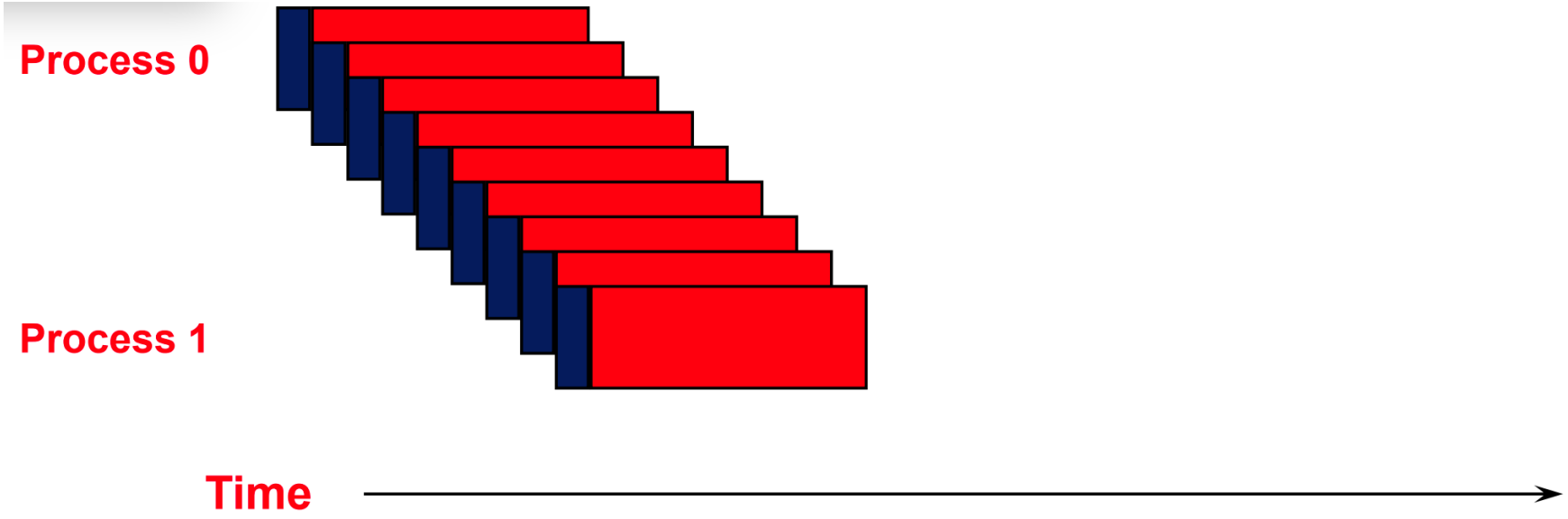
点对点通信实现协议

点对点通信协议

- Eager 协议
  - 直接发送到对方预设的缓冲区
  - 假定接收进程可以存储传入的数据
  - 延迟低，但需要接收进程有足够空间，所以一般用于传输小消息

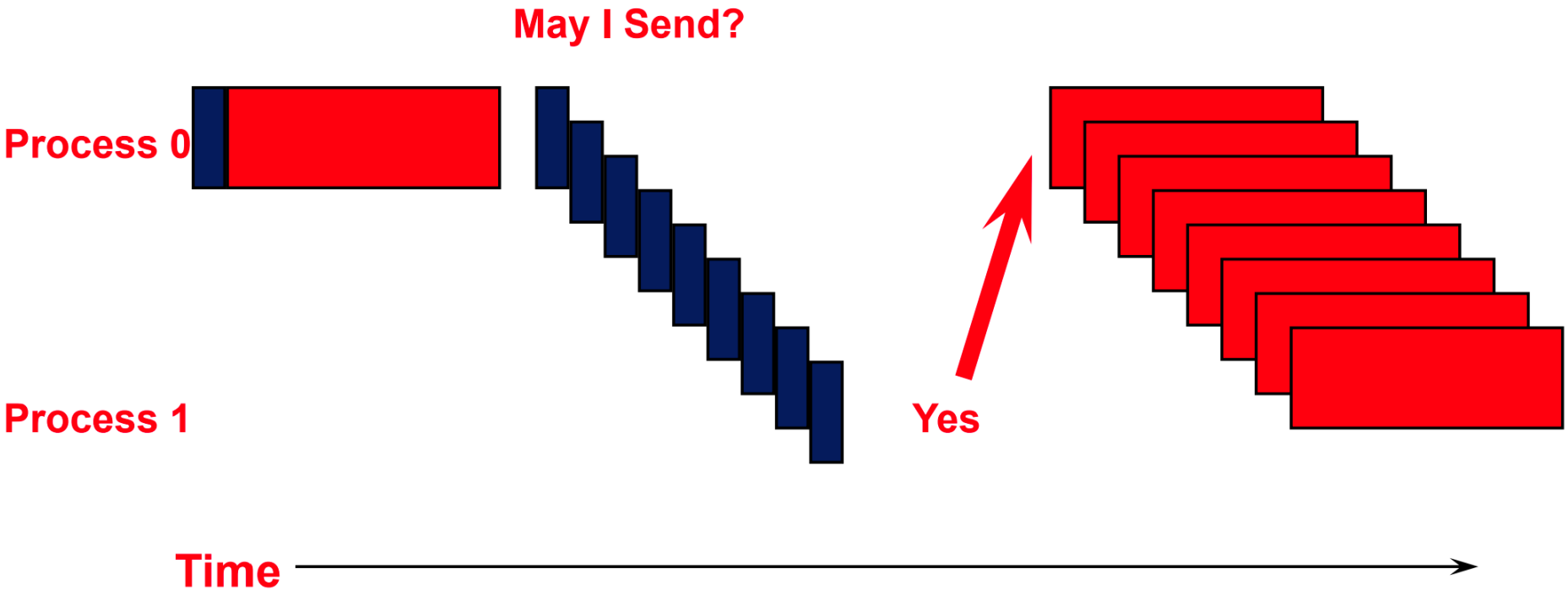
- Rendezvous 协议
  - 需要和接收进程确认（同步）
  - 在确定接收进程准备好接收数据后，发送者才开始传输数据
  - 延迟高，但不太需要额外空间，所以一般用于传输大消息

Eager协议



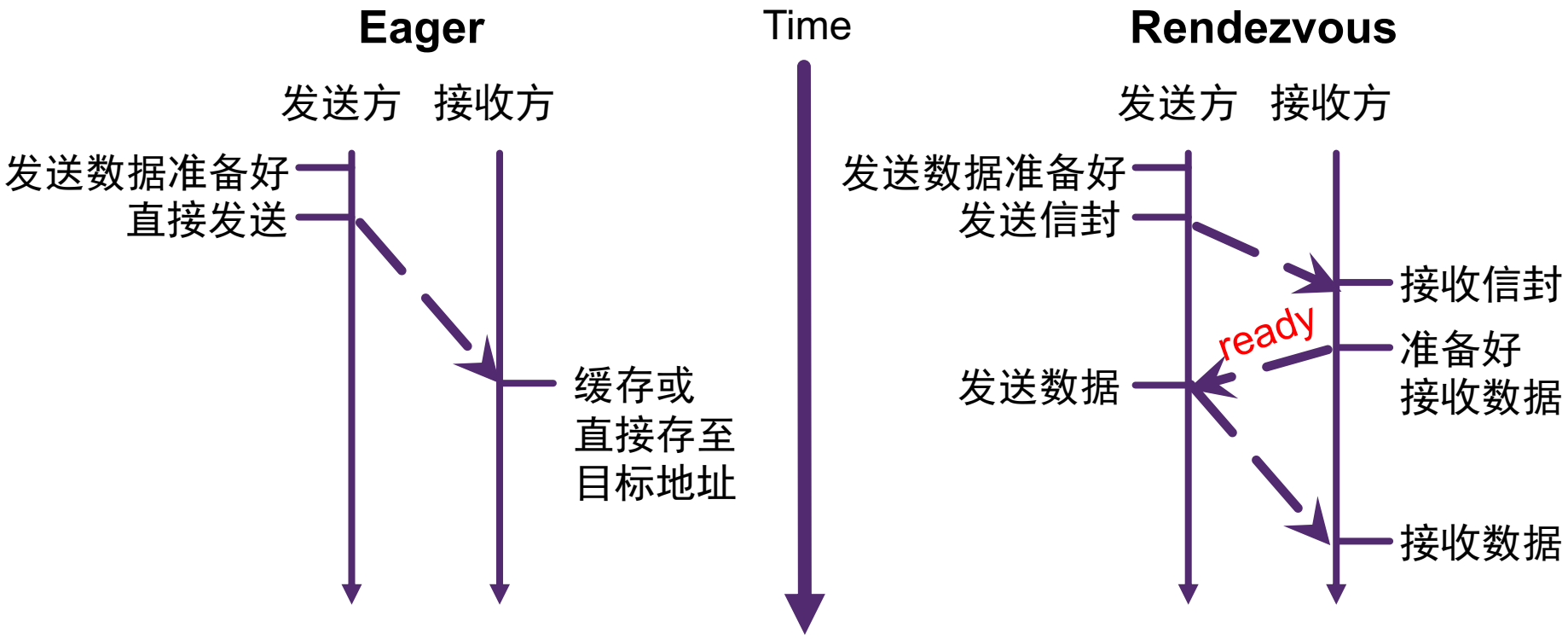
- 发送数据时假定接收者（进程1）可以存储传入的数据
- 发送到进程1的数据
  - 进程1一端可能存在no matching receive
  - 进程1必须缓存并拷贝这些数据

Rendezvous协议



- 首先发送envelopes（包含 tag, length, source 等信息）
- 当接收端（进程1）可以接收数据时（即用户缓冲区可用时），发送端（进程0）才发送数据
- 因此，只需要缓冲envelopes

Eager 协议和 Rendezvous 协议



- 发送数据时假定接收方可以存储传入的数据
- 发送到接收方的数据
  - 接收方可能还没有recv与发送来的数据对应
  - 接收方必须缓存并拷贝这些数据

- 首先发送信封（envelope，包含 tag, length, source 等信息）
- 当接收方可以接收其它数据时（即用户缓冲区可用时），发送方才发送数据
- 因此，只需要缓冲envelopes

Eager协议 vs. Rendezvous协议

Eager 特点

- 减少同步延迟
- 需要大量缓冲区
- 后台需要CPU的积极参与
- 引入额外拷贝（从缓冲区到最终目标地址）
- 适合小消息传输

Rendezvous 协议特点

- 鲁棒且安全
  - 除了envelopes的数量限制
- 移除潜在存储拷贝（直接从用户到用户）
- 可能引入同步延迟（等待接收端可以接收数据）
- 适合大消息传输

通信死锁

死锁的发生

- 从0号进程向1号进程发送一条大消息
  - 如果0号进程的系统存储空间（System buffer）不足，这次通信必须等到1号进程提供对应的存储空间（比如调用 MPI_Recv）
- 以下代码可能存在什么问题？

Process 0	Process 1
Send(1)	**Send(0)**
Recv(1)	**Recv(0)**

- *这段代码是不安全的！可能发生死锁*
- 因为这两次阻塞通信的完成取决于系统缓冲区是否可用，在接收数据前，系统缓冲区用于存储发送的数据

解决方案

- 方法一：安排合理的通信顺序

Process 0	Process 1
Send(1)	**Recv(0)**
Recv(1)	**Send(0)**

- 方法二：在发送的同时提供接收所用的缓冲区
  - **MPI_Sendrecv**：允许同时发送和接收

Process 0	Process 1
Sendrecv(1)	**Sendrecv(0)**

解决方案

- 方法三：提供自己的空间作为发送的缓冲区（MPI_Bsend）

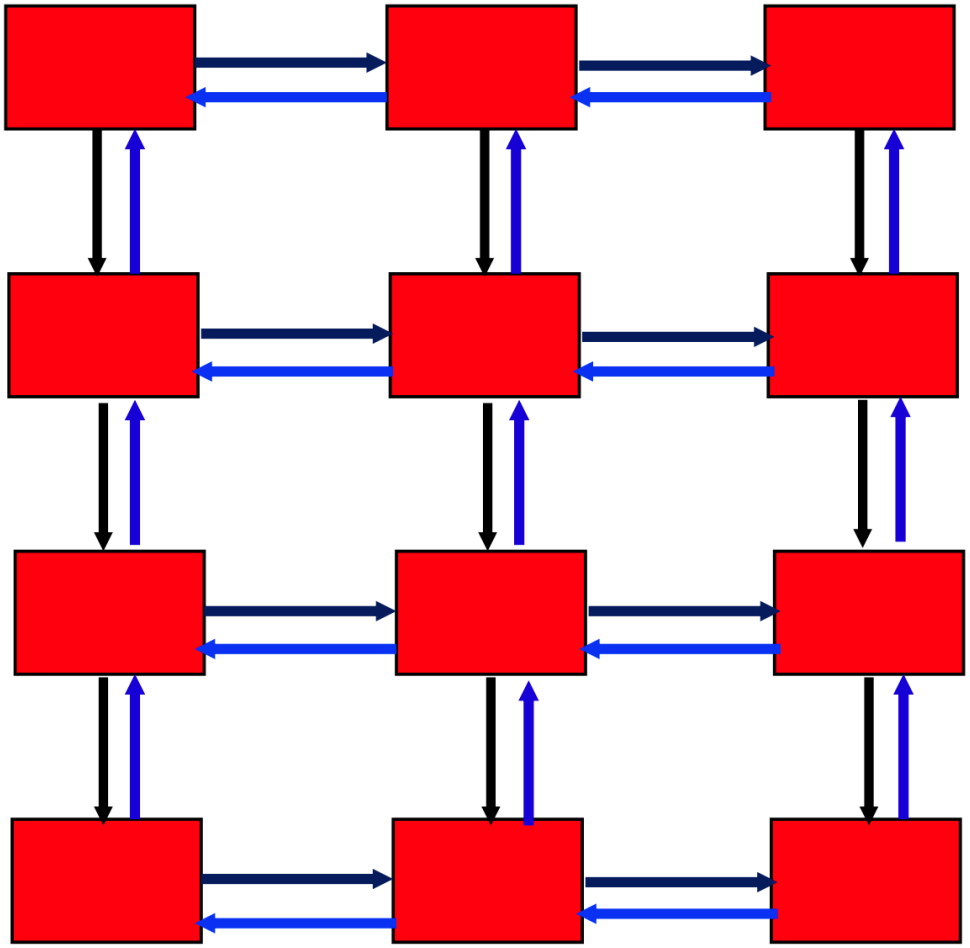
Process 0	Process 1
Bsend(1)	**Bsend(0)**
Recv(1)	**Recv(0)**

- 方法四：使用非阻塞通信（推荐使用）

Process 0	Process 1
Isend(1)	**Isend(0)**
Irecv(1)	**Irecv(0)**
Waitall	**Waitall**

一个稍微复杂的例子：网格交换（Mesh Exchange）

- 在一个2维网格上交互数据



一个可能的方案

```
for (int i = 0; i < n_neighbors; ++i)
   MPI_Send (edge, len, MPI_REAL, neighbor(i), tag, comm,
                                                    ierr);

for (int i = 0; i < n_neighbors; ++i)
   MPI_Recv (edge, len, MPI_REAL, neighbor(i), tag, comm,
                                                    ierr);
```

- 潜在的问题是什么？
  - 通信死锁

解决方案-1：顺序化

- 上页代码所有的发送可能都会被阻塞，等待一个接收与之匹配

- 稍作修改，以上下传输为例：

```
if (has_down_neighbor)
    MPI_Send(... down ...);

if (has_up_neighbor)
    MPI_Recv(... up ...);
```

- 问题是什么？
  - 将每一层的传输串行化
  - 最后一层不用send，直接recv
  - 倒数第二层send后，开始recv
  - 倒数第三层send后，开始recv
  - ……

解决方案-2：使用Irecv

```
for (int i = 0; i < n_neighbors; ++i)
    MPI_Irecv(edge, len, MPI_REAL, neighbor(i), tag,
                                   comm, request(i), ierr);

for (int i = 0; i < n_neighbors; ++i)
    MPI_Send(edge, len, MPI_REAL, neighbor(i), tag, comm,
                                                 ierr);

MPI_Waitall(n_neighbors, request, statuses, ierr);
```

- 如果发送方要发送的数据较大，接收方的buffer不够：
- 那么发送方会阻塞、暂缓发送，此时发送方剩余要发给其他进程的MPI_send 也都会被阻塞
  - 即，发送的顺序引入了延迟
  - 网络带宽会被严重浪费

解决方案-3：使用 Isend 和 Irecv

```
for (int i = 0; i < n_neighbors; ++i)
    MPI_Irecv(edge, len, MPI_REAL, neighbor(i), tag, comm,
                                                request(i), ierr);

for (int i = 0; i < n_neighbors; ++i)
    MPI_Isend(edge, len, MPI_REAL, neighbor(i), tag, comm,
                               request(n_neighbors + i), ierr);

MPI_Waitall(2 * n_neighbors, request, statuses, ierr);
```

- 减少了不必要同步操作
- 网络带宽被充分的利用

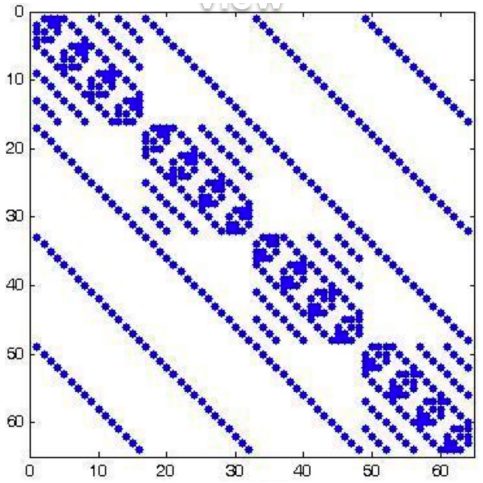
教训：推迟同步

- Send-Receive 配对操作隐含完成两个事情：
  - 数据传输
  - 进程同步
- 在很多情况下，一些同步是不必要的
- 使用非阻塞通信和 MPI_Waitall 来推迟同步

进程映射

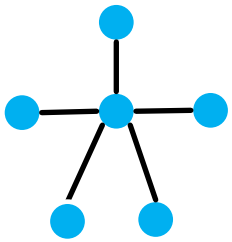
为什么进程映射

- 应用程序：应用的不同进程间通信需求不一致

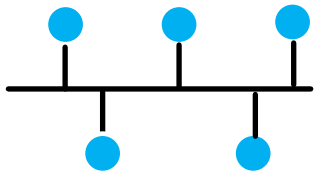


不同进程之间消息量分布

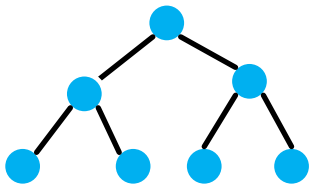
- 网络拓扑：集群内连接的网络拓扑与性能（带宽、延迟）不同



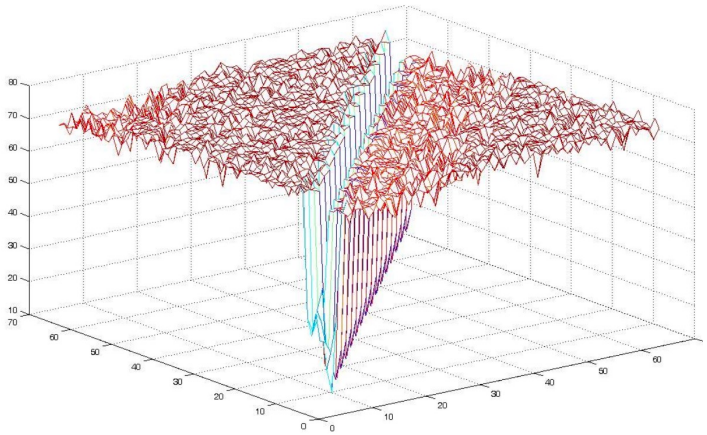
星型



总线型



树型

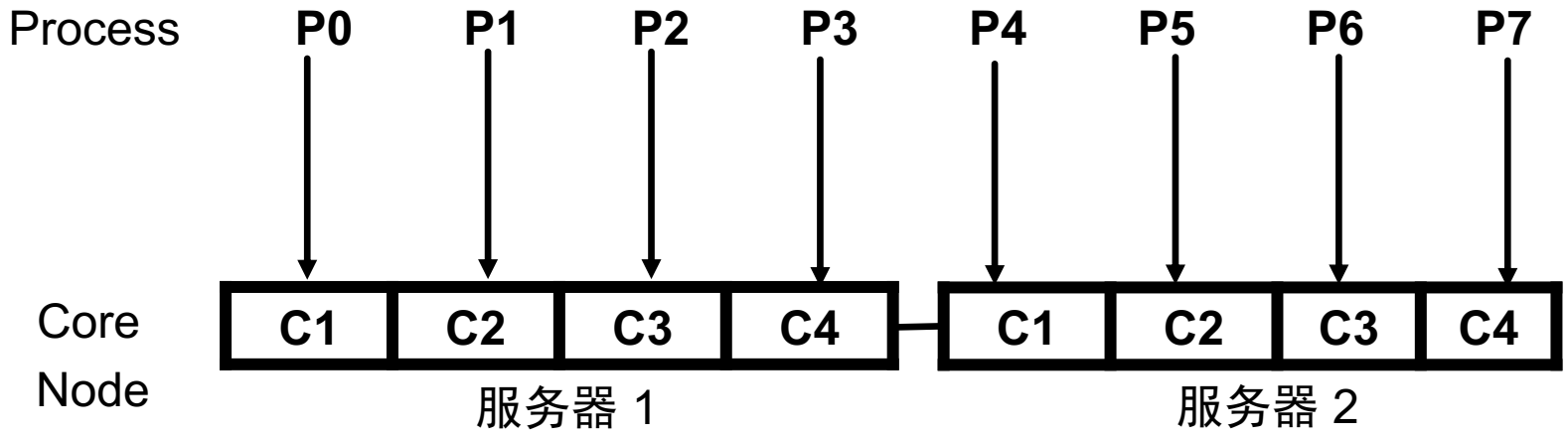


不同处理器之间延迟性能分布

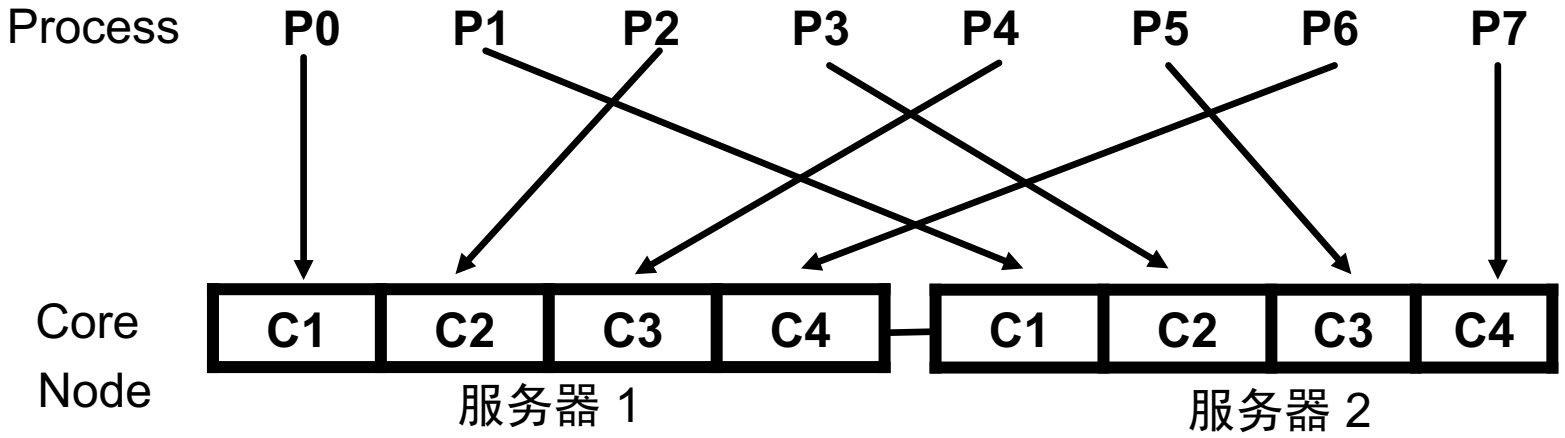
进程映射：将进程与物理节点做合理映射可以有效减少通信开销

两个简单映射方法: Block & Cyclic

- Block 映射方式：

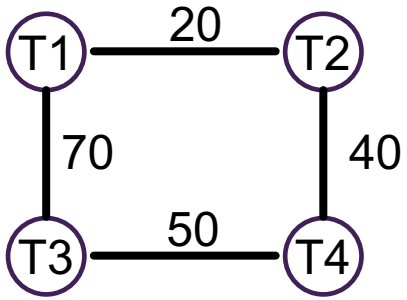


- Cyclic 映射方式：

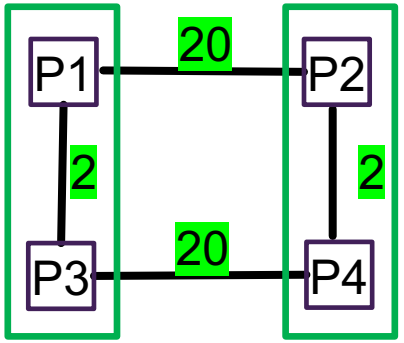


问题抽象：图映射问题

- 通信图：描述MPI应用程序的通信情况（图T）
- 系统拓扑图：描述集群内处理器的互连情况（图G）

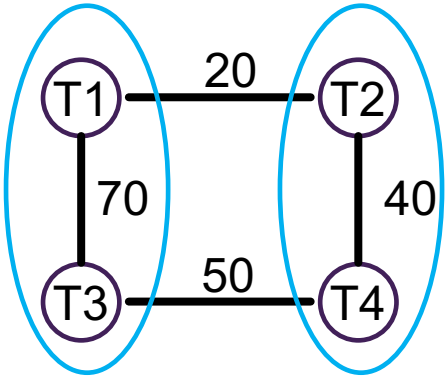


A：通信图T（通信量）

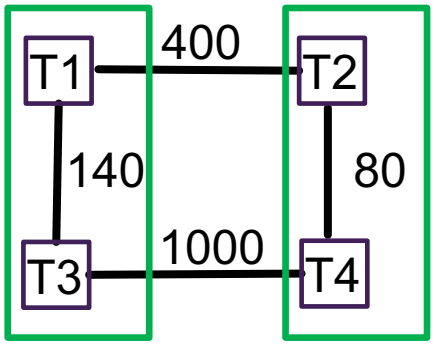


B：系统拓扑图G（延迟）

- 图问题：给定两个带有边权的图T和G，将T的点映射至G，使得一个目标成本函数F的值最小
  - NP完全问题
  - 启发式算法[1]
- 以函数F为映射后G和T对应边权乘积之和为例：



C：图划分



D：图映射

[1] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, H. Kuhn. MPIPP: An Automatic Profile-Guided Parallel Process Placement Toolset for SMP Clusters and Multi-Clusters. ICS 2006.

如何指定进程映射

- 以 OpenMPI 为例，machinefile / rankfile 指定节点及（可执行）进程数：
  - `host1 slots=24`
  - `host2 slots=16`
  - 如果启动30个进程，0~23进程将运行在host1，24~29进程将运行在host2。
  - 如果启动超过40个进程，后面的进程会被随机分配，可以使用`max_slots`进行限制。
- 命令行运行：`mpirun [-np X] -hostfile <filename> <program>`
- 更多请参考文档：
  https://www.open-mpi.org/doc/v4.0/man1/mpirun.1.php#sect6
- Intel MPI 称为 hostfile：
  https://software.intel.com/content/www/us/en/develop/articles/controlling-process-placement-with-the-intel-mpi-library.html

进程绑定

- 将进程绑定在固定的 CPU Core / Socket 上
  - 避免进程在核之间切换带来开销
  - 减少 cache miss
  - 避免跨 NUMA 内存访问
- 使用方法
  - MPI 自带进程绑定选项
    - OpenMPI 中的 `--bind-to-core` 和 `--bind-to-socket` 参数
    - Intel MPI 中的 `-binding` 选项：https://software.intel.com/content/www/us/en/develop/documentation/mpi-developer-reference-linux/top/binding-options.html
  - SLURM 等任务调度系统
    - 运行时使用 `--cpu-bind=sockets / cores` 选项，也支持根据进程数自动绑定
    - 详细文档： https://slurm.schedmd.com/mc_support.html
  - 操作系统工具
    - 如 `numactl -C 0 ./program` 将程序绑定在 CPU 0 上

集合通信算法实现

MPI集合通信

集合通信	操作内容
MPI_Bcast	将一个进程的消息广播至组内所有进程
MPI_Reduce	将组内所有进程的消息归约至一个进程
MPI_Gather	将组内每个进程的不同消息收集到一个目标进程
MPI_Scatter	将一个进程的不同消息分发到组内所有进程
MPI_Allgather	将消息串联到组内所有进程
MPI_Allreduce	执行归约操作并将结果存在组内所有进程中
MPI_Alltoall	将所有进程的数据转发至所有进程

- 典型的 MPI 集合通信算法底层通过点对点通信来实现
  - 针对不同消息大小、不同网络性能以及不同进程数如何设计高效率算法？

通信性能模型

通信性能模型

- 发送长度为n（字节）消息的时间大概为：

  Time = latency + n * 每字节传输开销

  = latency + n / 带宽

- 暂时不考虑网络拓扑带来的影响
- 经常被称为“α–β模型”： Time = α + n*β
- 通常α >> β >> 每浮点运算时间
  - 一个长消息的开销比许多短消息的总开销要少
  - $\alpha + n*\beta \ll n * (\alpha + 1*\beta)$
- 收发一条消息的开销可以做成百上千次浮点运算
  - 为了程序性能高效，计算-通信比例需要足够大
- LogP 模型 (**L**atency / **o**verhead / **g**ap / **P**roc)：更详细
  - David Culler, et al. LogP: towards a realistic model of parallel computation. PPOPP. 1993.

服务器上典型α–β参数

machine	α	β
T3E/Shm	1.2	0.003
T3E/MPI	6.7	0.003
IBM/LAPI	9.4	0.003
IBM/MPI	7.6	0.004
Quadrics/Get	3.267	0.00498
Quadrics/Shm	1.3	0.005
Quadrics/MPI	7.3	0.005
Myrinet/GM	7.7	0.005
Myrinet/MPI	7.2	0.006
Dolphin/MPI	7.767	0.00529
Giganet/VIPL	3.0	0.010
GigE/VIPL	4.6	0.008
GigE/MPI	5.854	0.00872

α 是延迟（单位是微秒）
β 是带宽（单位是微秒每字节）

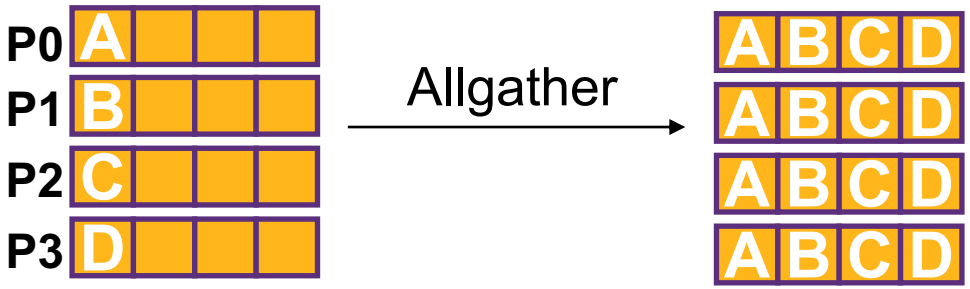
典型集合通信实现

- **MPI_Allgather**
- MPI_Bcast
- MPI_Alltoall
- MPI_Allreduce

MPI_Allgather 算法实现

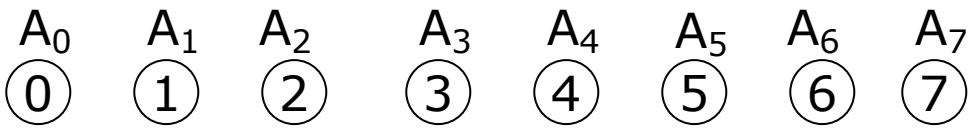
- Ring （环形算法）
- Bruck
- Recursive Doubling （递推倍增算法）

sendcnt = recvcnt = 1;



Ring 算法

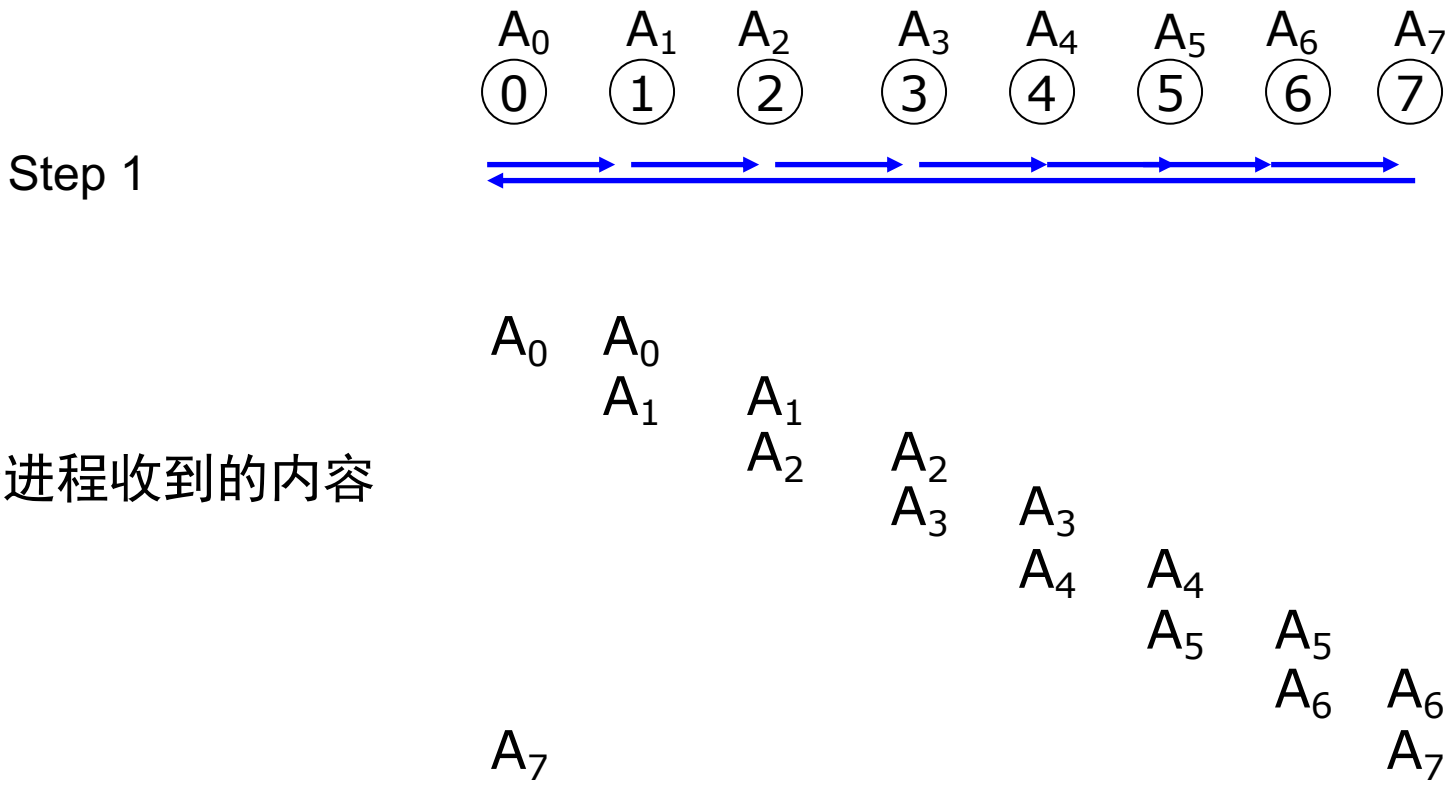
- 每一个进程都从自己的前一个进程收取信息，然后发送给下一个进程（第一个和最后一个进程需要特殊考虑）
- 假设有P个进程，该算法执行一次Allgather操作，需要 P-1 步
  - 例如，8个进程执行Allgather操作，共需要7步



每个进程收到的内容

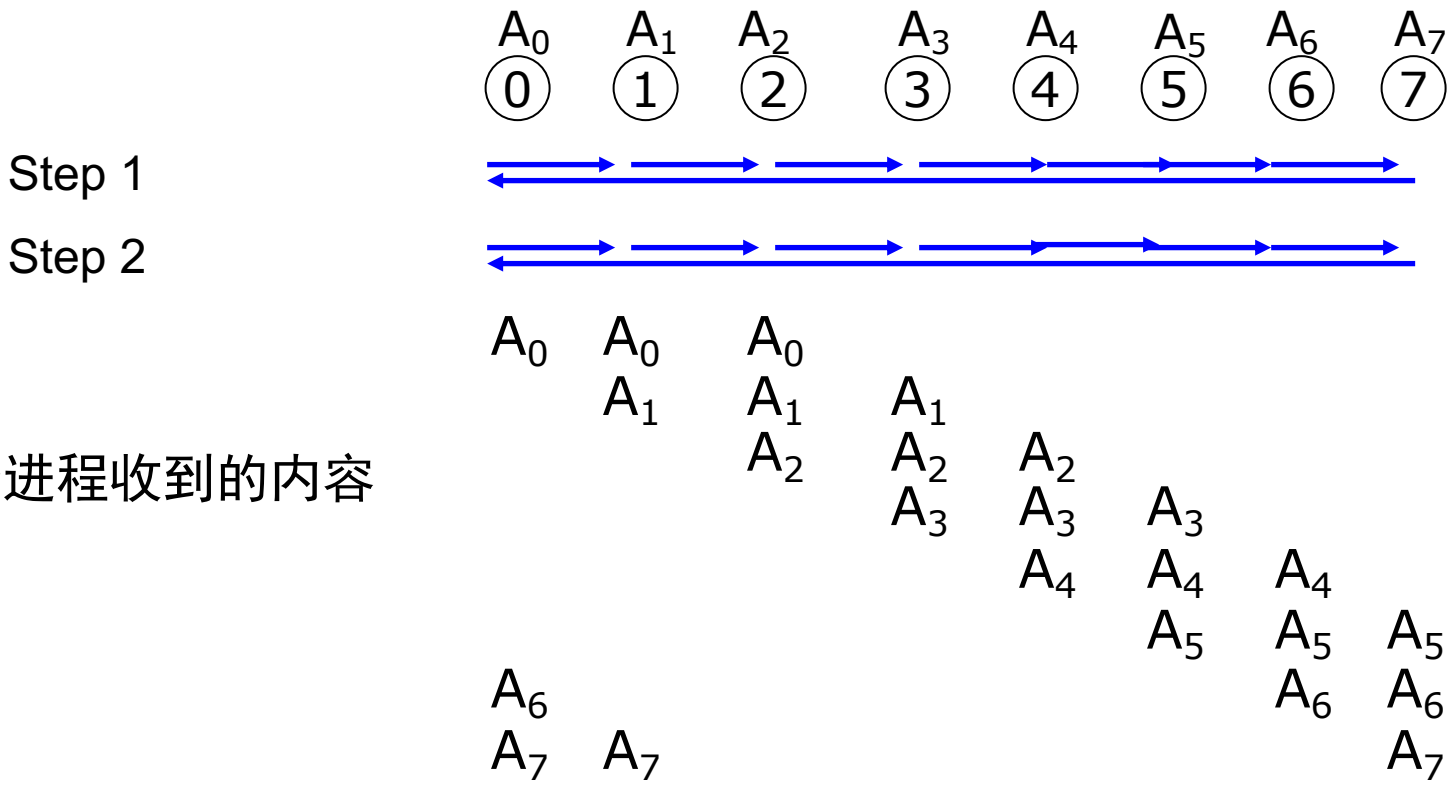
Ring 算法

- 每一个进程都从自己的前一个进程收取信息，然后发送给下一个进程（第一个和最后一个进程需要特殊考虑）
- 假设有P个进程，该算法执行一次Allgather操作，需要 P-1 步
  - 例如，8个进程执行Allgather操作，共需要7步



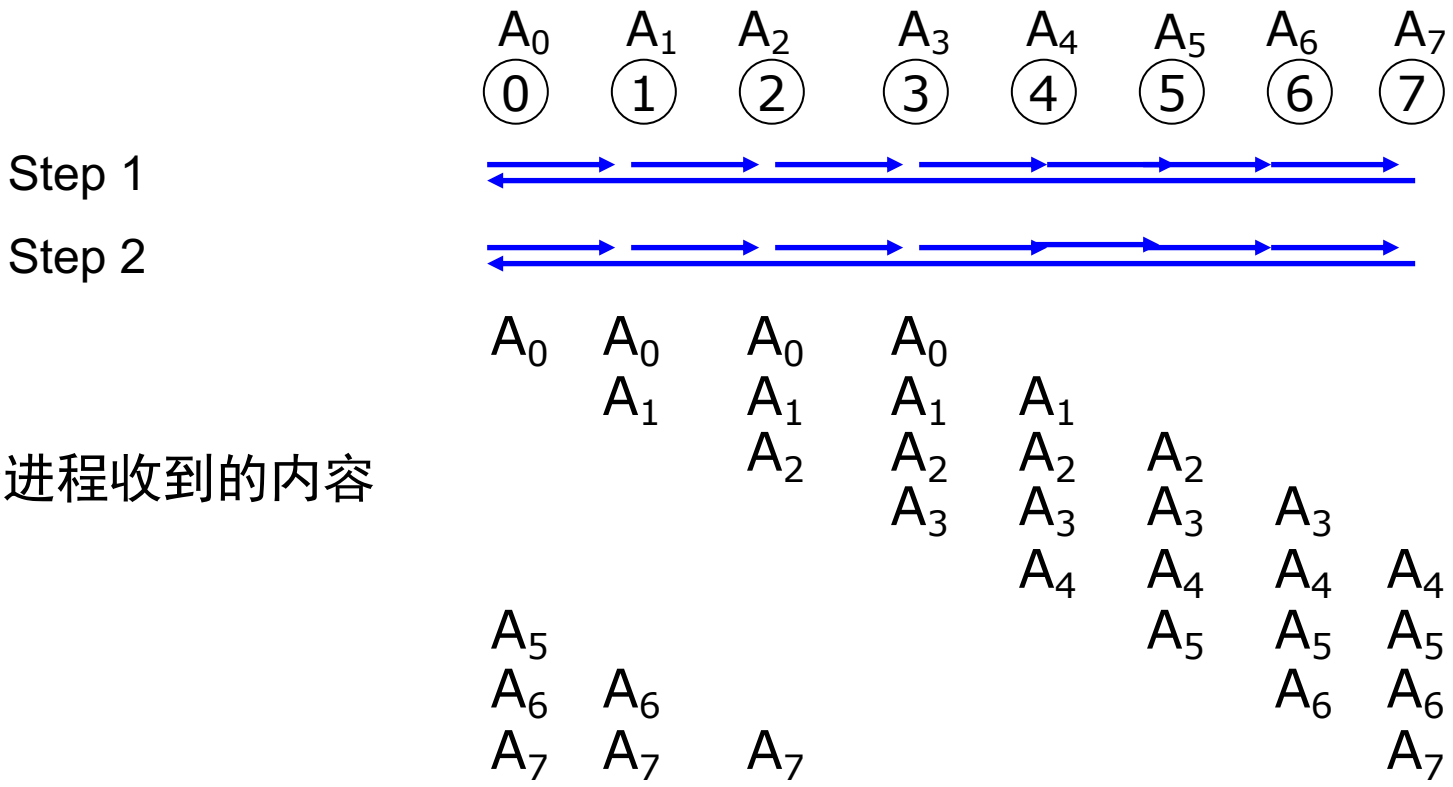
Ring 算法

- 每一个进程都从自己的前一个进程收取信息，然后发送给下一个进程（第一个和最后一个进程需要特殊考虑）
- 假设有P个进程，该算法执行一次Allgather操作，需要 P-1 步
  - 例如，8个进程执行Allgather操作，共需要7步



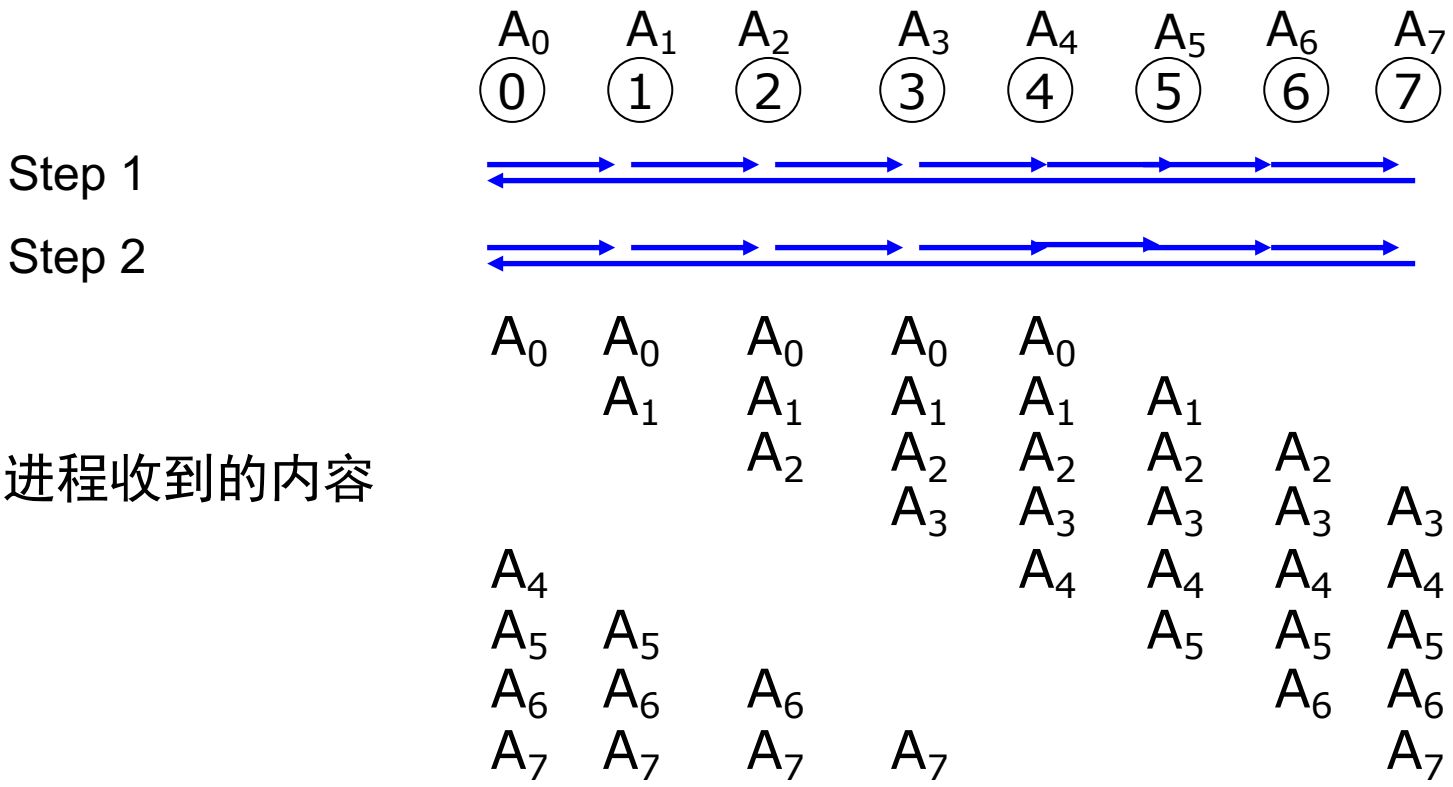
Ring 算法

- 每一个进程都从自己的前一个进程收取信息，然后发送给下一个进程（第一个和最后一个进程需要特殊考虑）
- 假设有P个进程，该算法执行一次Allgather操作，需要 P-1 步
  - 例如，8个进程执行Allgather操作，共需要7步



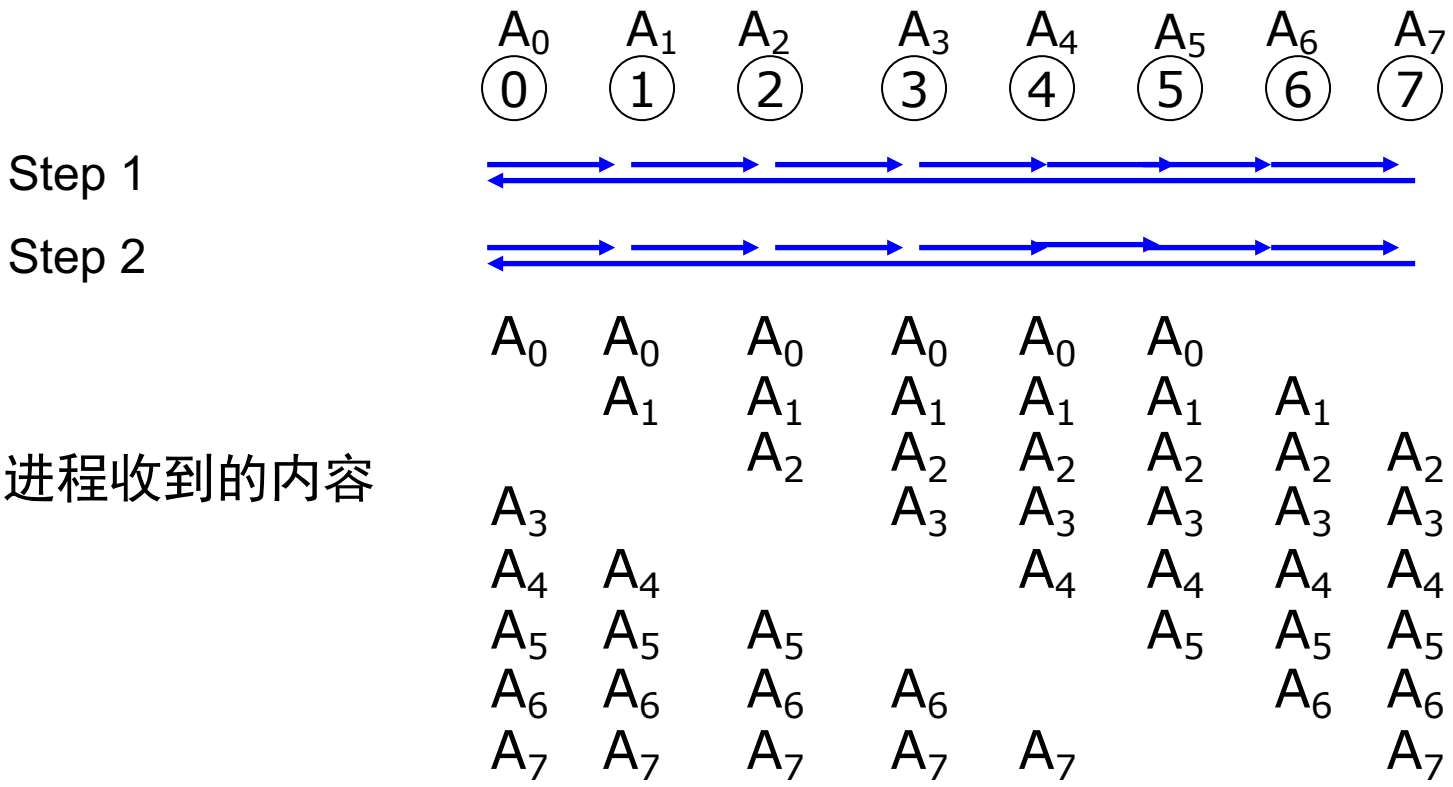
Ring 算法

- 每一个进程都从自己的前一个进程收取信息，然后发送给下一个进程（第一个和最后一个进程需要特殊考虑）
- 假设有P个进程，该算法执行一次Allgather操作，需要 P-1 步
  - 例如，8个进程执行Allgather操作，共需要7步



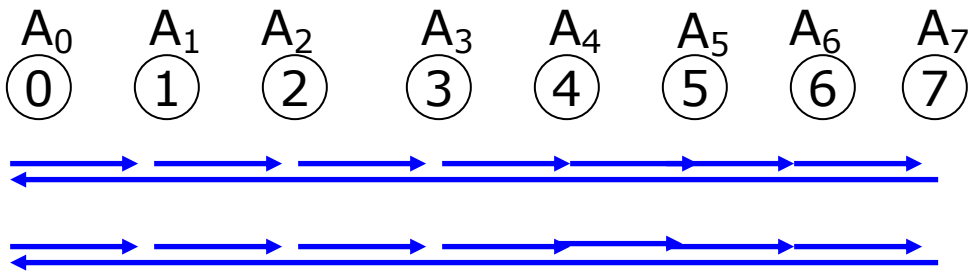
Ring 算法

- 每一个进程都从自己的前一个进程收取信息，然后发送给下一个进程（第一个和最后一个进程需要特殊考虑）
- 假设有P个进程，该算法执行一次Allgather操作，需要 P-1 步
  - 例如，8个进程执行Allgather操作，共需要7步



Ring 算法

- 每一个进程都从自己的前一个进程收取信息，然后发送给下一个进程（第一个和最后一个进程需要特殊考虑）
- 假设有P个进程，该算法执行一次Allgather操作，需要 P-1 步
  - 例如，8个进程执行Allgather操作，共需要7步

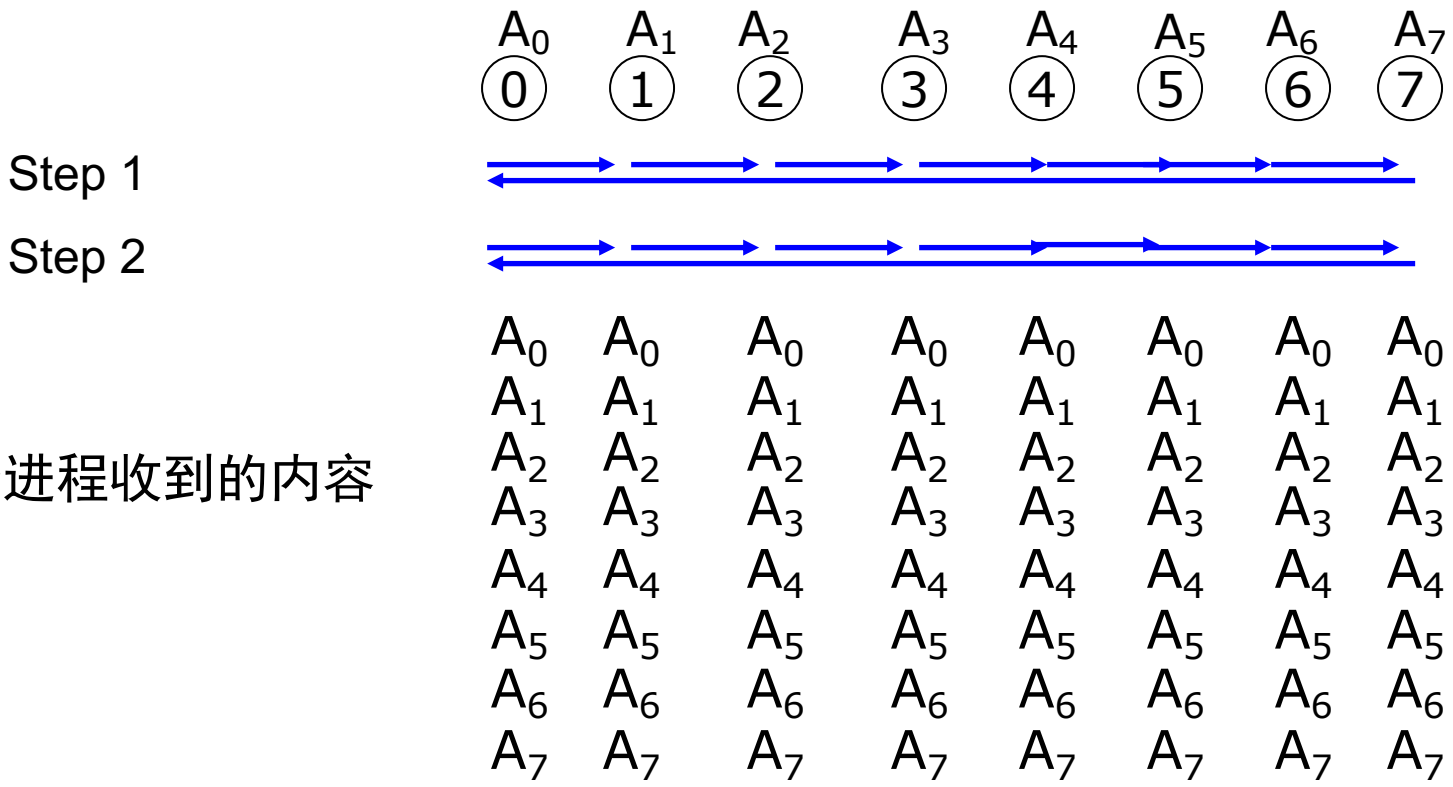


每个进程收到的内容

A_0	A_0	A_0	A_0	A_0	A_0	A_0	
	A_1	A_1	A_1	A_1	A_1	A_1	A_1
A_2		A_2	A_2	A_2	A_2	A_2	A_2
A_3	A_3		A_3	A_3	A_3	A_3	A_3
A_4	A_4	A_4		A_4	A_4	A_4	A_4
A_5	A_5	A_5	A_5		A_5	A_5	A_5
A_6	A_6	A_6	A_6	A_6		A_6	A_6
A_7	A_7	A_7	A_7	A_7	A_7		A_7

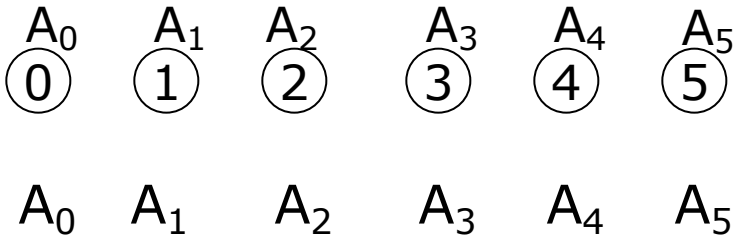
Ring 算法

- 每一个进程都从自己的前一个进程收取信息，然后发送给下一个进程（第一个和最后一个进程需要特殊考虑）
- 假设有P个进程，该算法执行一次Allgather操作，需要 P-1 步
  - 例如，8个进程执行Allgather操作，共需要7步



Bruck 算法

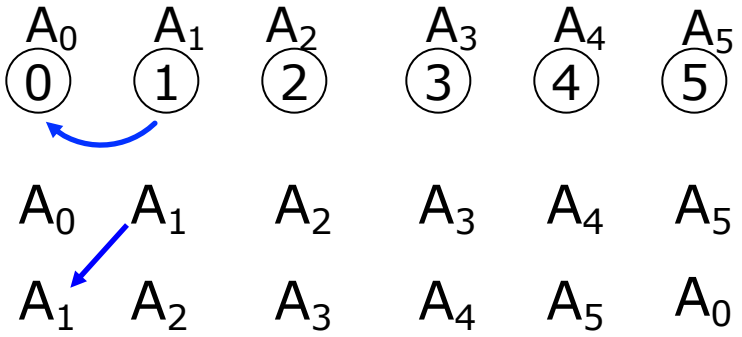
- 共需要 $\lceil \log_2 P \rceil$ 步。在第 k 步（k从0开始）中，进程 i 将数据发送给进程$(i-2^k)$，然后从进程 $(i+2^k)$ 上接收数据
- 需要注意的一点是，这样交互的结果是需要处理的，每个进程的输出 buffer 是需要调整的。进程i的输出 buffer 需要循环下移 i 个位置



每个进程收到的内容

Bruck 算法

- 共需要 $\lceil \log_2 P \rceil$ 步。在第 k 步（k从0开始）中，进程 i 将数据发送给进程(i-2^k)，然后从进程 (i+2^k) 上接收数据
- 需要注意的一点是，这样交互的结果是需要处理的，每个进程的输出 buffer 是需要调整的。进程i的输出 buffer 需要循环下移 i 个位置

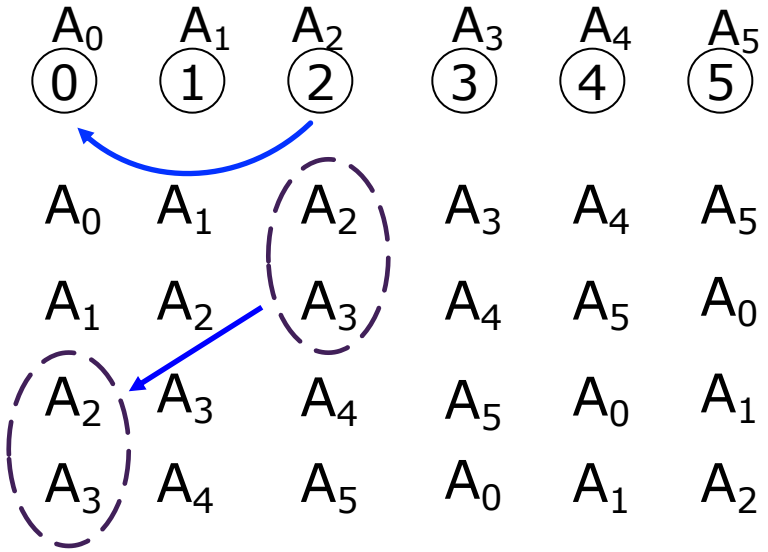


每个进程收到的内容

Bruck 算法

- 共需要 $\lceil \log_2 P \rceil$ 步。在第 k 步（k从0开始）中，进程 i 将数据发送给进程(i-2^k)，然后从进程 (i+2^k) 上接收数据
- 需要注意的一点是，这样交互的结果是需要处理的，每个进程的输出 buffer 是需要调整的。进程i的输出 buffer 需要循环下移 i 个位置

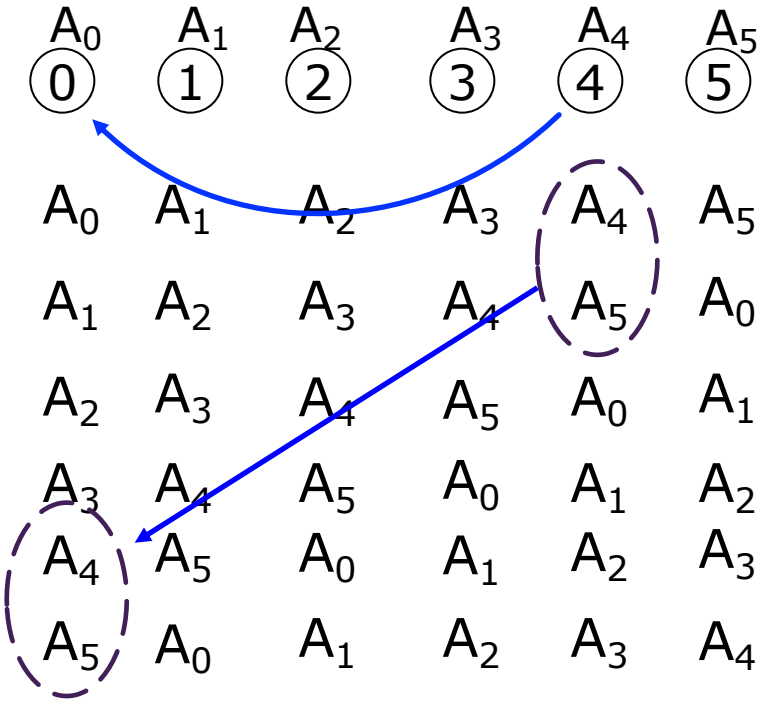
每个进程收到的内容



Bruck 算法

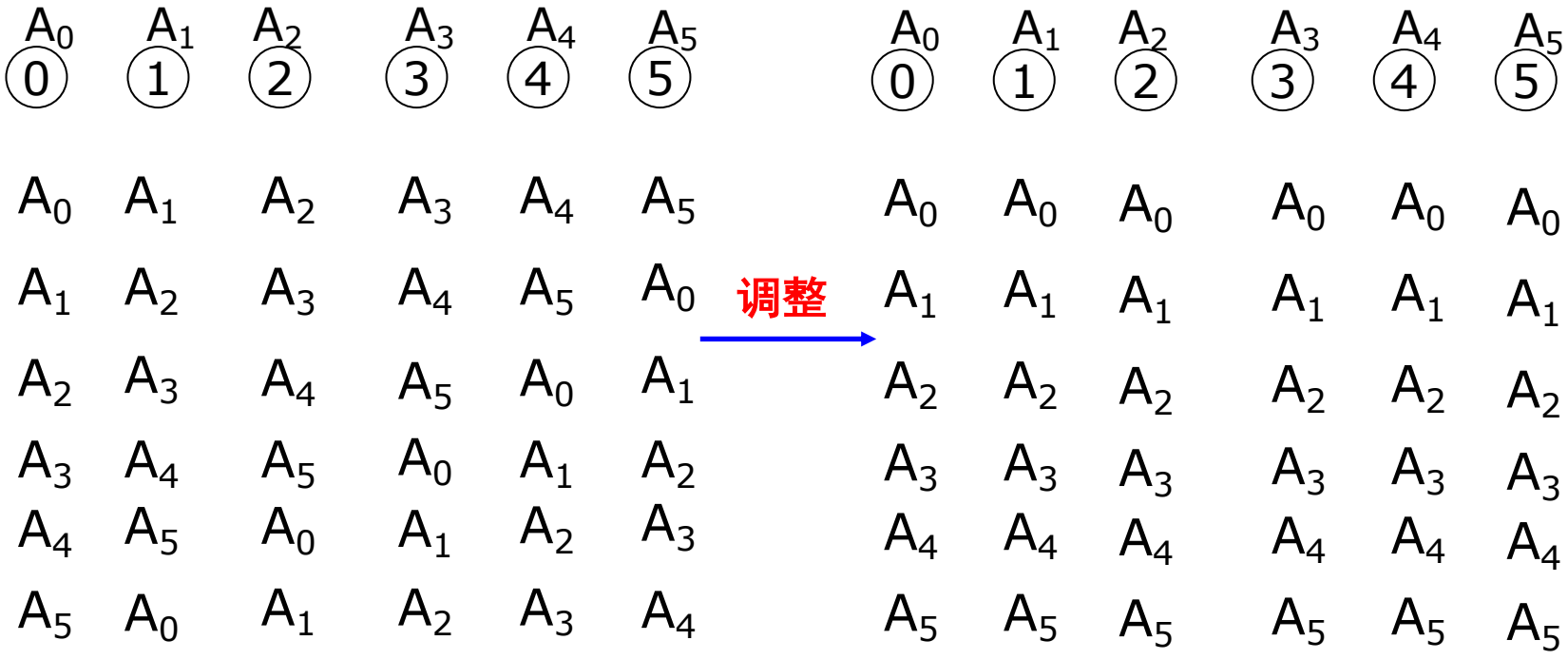
- 共需要 $\lceil \log_2 P \rceil$ 步。在第 k 步（k从0开始）中，进程 i 将数据发送给进程(i-2^k)，然后从进程 (i+2^k) 上接收数据
- 需要注意的一点是，这样交互的结果是需要处理的，每个进程的输出 buffer 是需要调整的。进程i的输出 buffer 需要循环下移 i 个位置

每个进程收到的内容



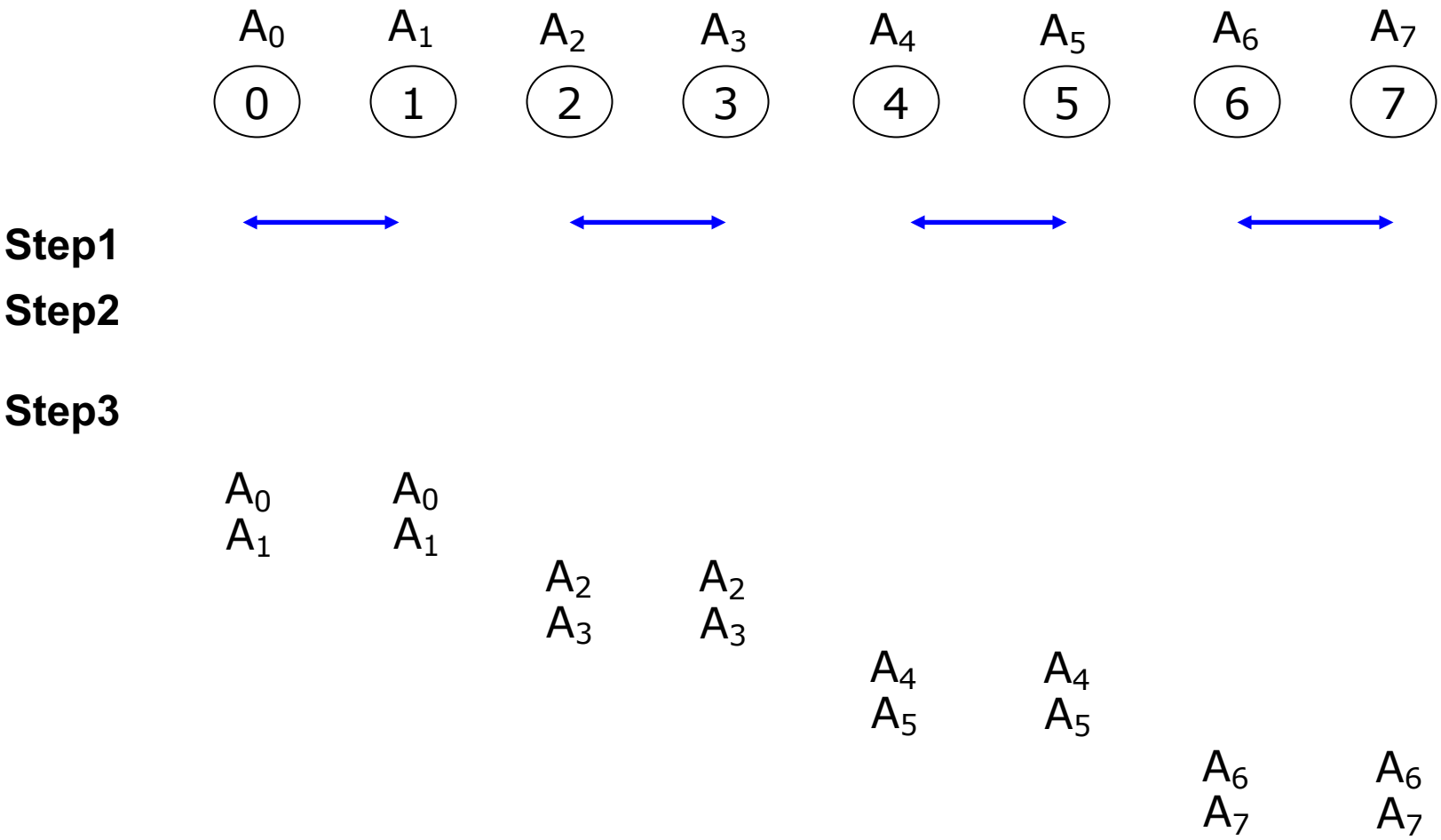
Bruck 算法

- 共需要 $\lceil \log_2 P \rceil$ 步。在第 k 步（k从0开始）中，进程 i 将数据发送给进程(i-2^k)，然后从进程 (i+2^k) 上接收数据
- 需要注意的一点是，这样交互的结果是需要处理的，每个进程的输出 buffer 是需要调整的。进程i的输出 buffer 需要循环下移 i 个位置



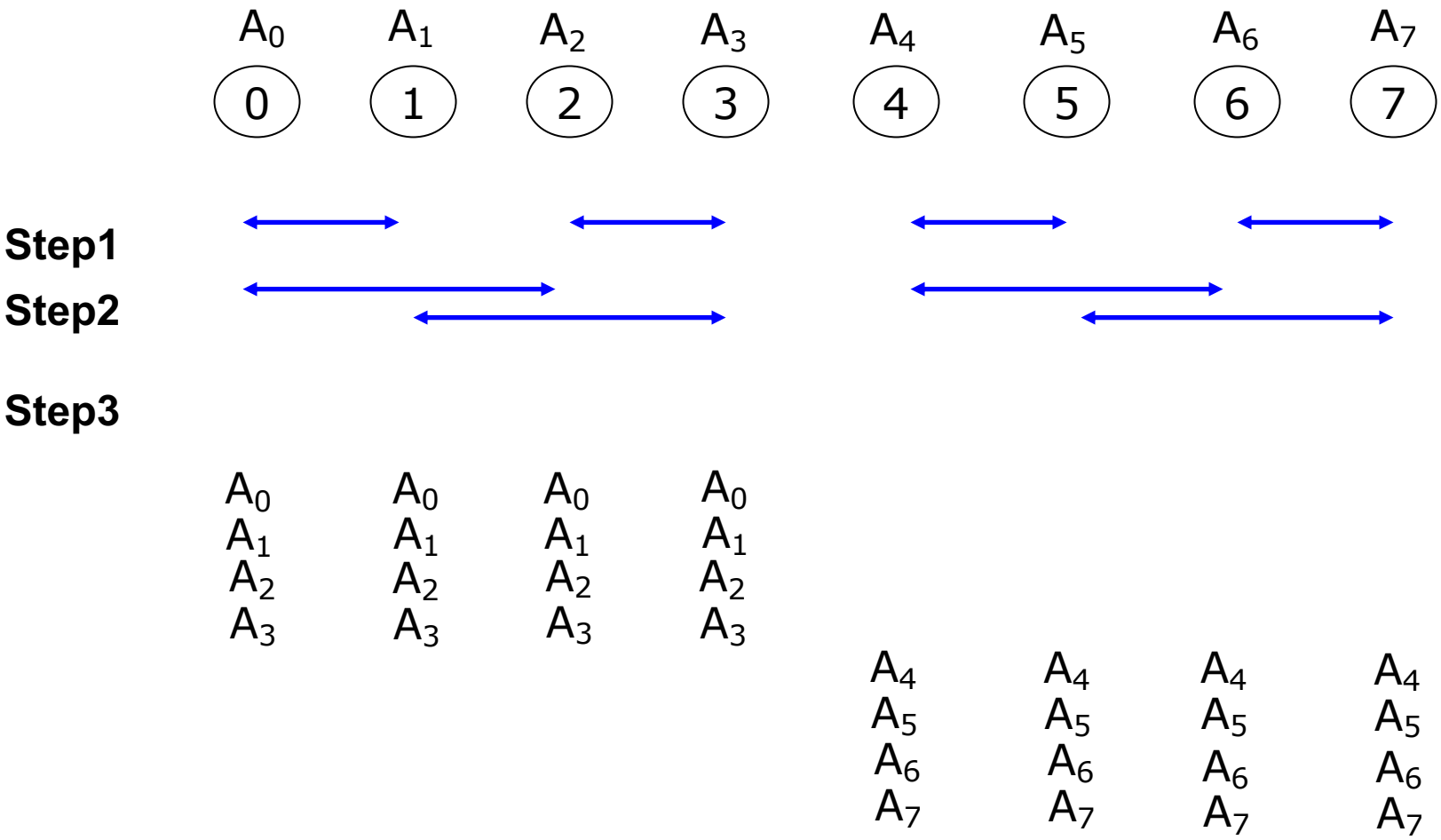
Recursive Doubling 算法

- 共需要 $\lceil \log_2 P \rceil$ 步
- 在第k步（k 从0开始）中，进程 i 与进程 (i+2^k) 或 (i-2^k) 交换数据



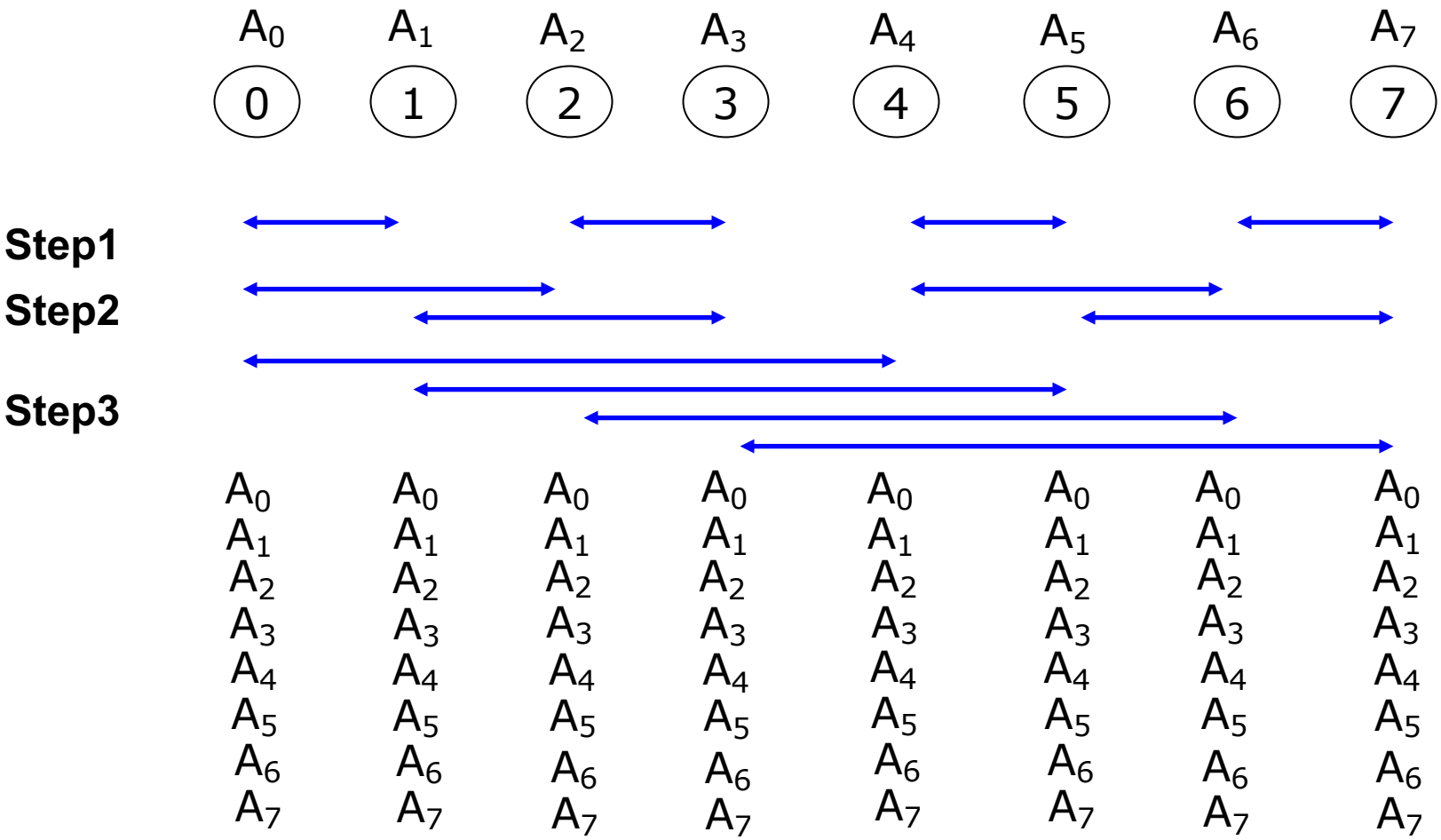
Recursive Doubling 算法

- 共需要 $\lceil \log_2 P \rceil$ 步
- 在第k步（k 从0开始）中，进程 i 与进程 (i+2^k) 或 (i-2^k) 交换数据



Recursive Doubling 算法

- 共需要 $\lceil \log_2 P \rceil$ 步
- 在第k步（k 从0开始）中，进程 i 与进程 (i+2^k) 或 (i-2^k) 交换数据



MPI_Allgather 算法性能分析

- 从几个维度去思考算法性能：
  - 通信所需的步数
  - 每次传输的通信量
  - 网络带宽的利用率

算法	开销	特点
Ring （环形算法）	$(\mathrm{P}-1)\alpha+\frac{\mathrm{P}-1}{\mathrm{P}}n\beta$	最简单，步数多，适合大消息
Bruck	$\lceil \lg \mathrm{P} \rceil \alpha+\frac{\mathrm{P}-1}{\mathrm{P}}n\beta$	最后需要额外调整，但在P为非2的正整数次幂时延迟比Recursive Doubling低
Recursive Doubling （递推倍增算法）假设P为2的正整数次幂	$\lceil \lg \mathrm{P} \rceil \alpha+\frac{\mathrm{P}-1}{\mathrm{P}}n\beta$	P为非2的正整数次幂时会有额外开销，步数至多为$2\lceil \lg \mathrm{P} \rceil$

MPI_Allgather 算法使用

- Infiniband 网络：
  - Recursive Doubling
- Ethernet 网络：
  - 短消息使用 Bruck
  - 中等长度消息使用 Recursive Doubling
  - 长消息使用 Ring

- 注：由于不同库的实现存在差异，本节的介绍主要基于 MPICH

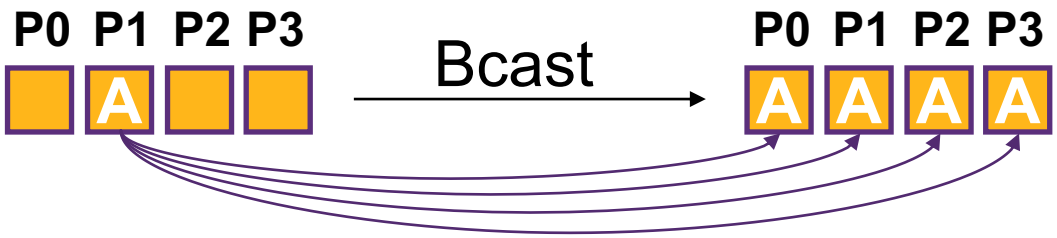
典型集合通信实现

- MPI_Allgather
- **MPI_Bcast**
- MPI_Alltoall
- MPI_Allreduce

MPI_Bcast 算法实现

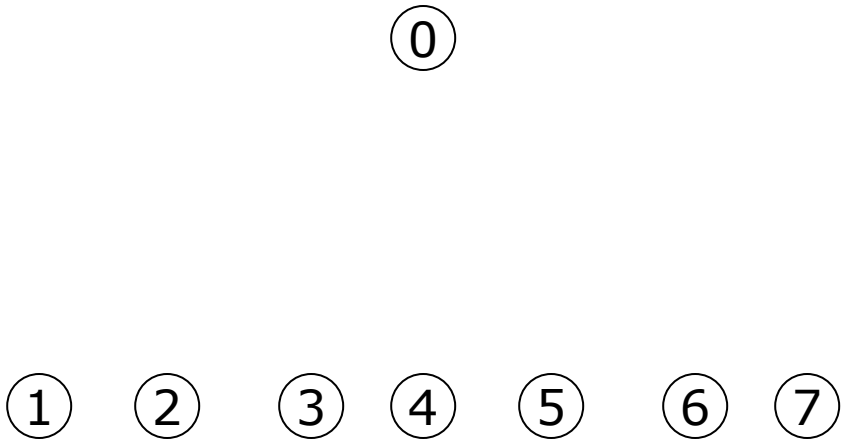
- Flat Tree （扁平树算法）
- Binomial Tree （二项树算法）
- Van de Geijn (Scatter-Allgather)

root = 1; count = 1;



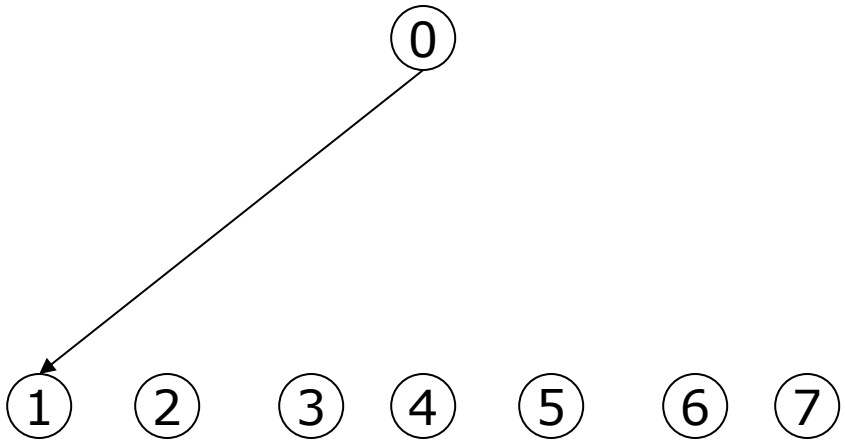
Flat Tree 算法

- root进程依次给其他进程发送消息



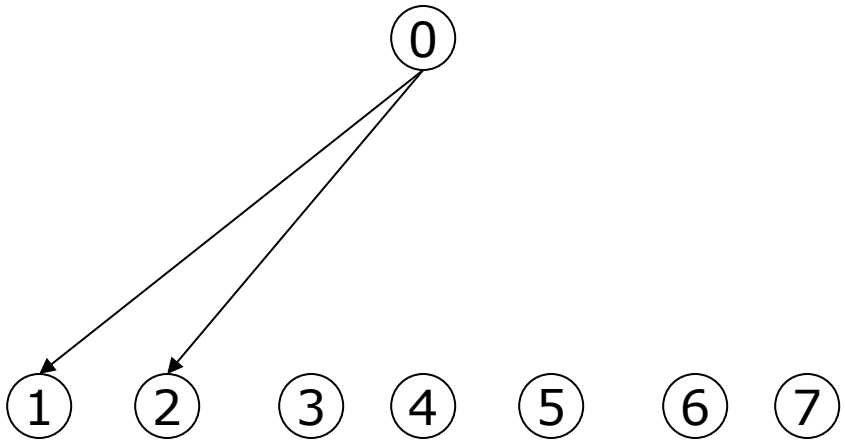
Flat Tree 算法

- root进程依次给其他进程发送消息



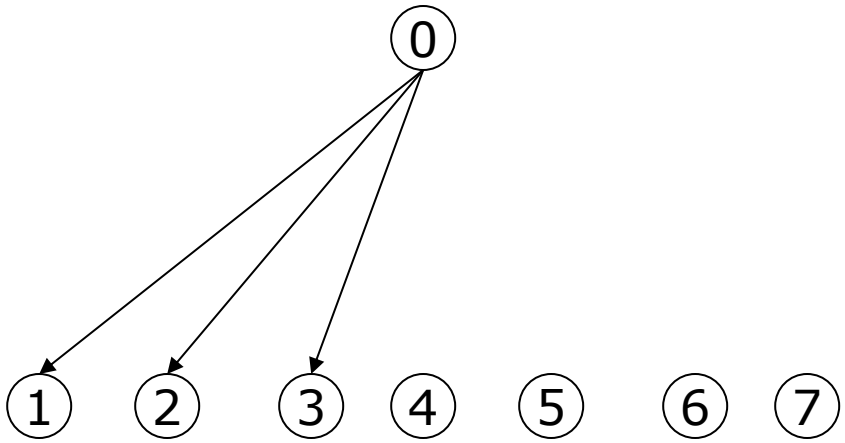
Flat Tree 算法

- root进程依次给其他进程发送消息



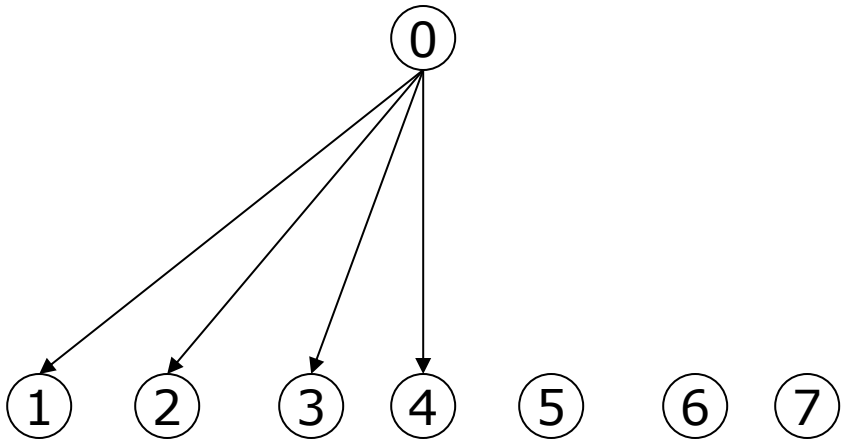
Flat Tree 算法

- root进程依次给其他进程发送消息



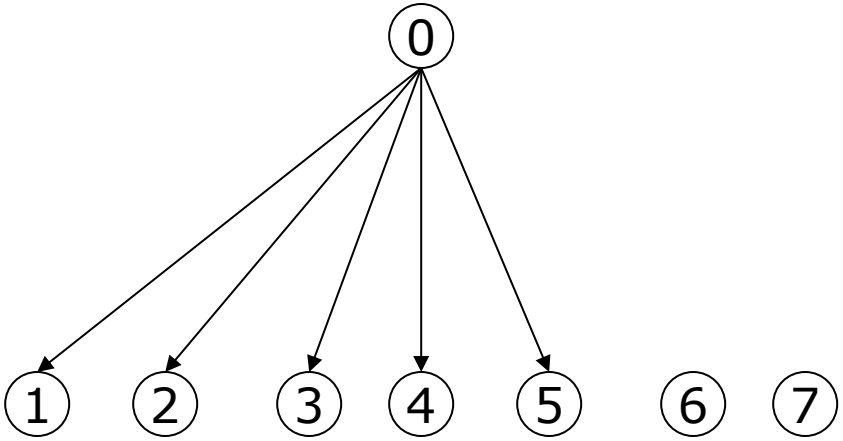
Flat Tree 算法

- root进程依次给其他进程发送消息



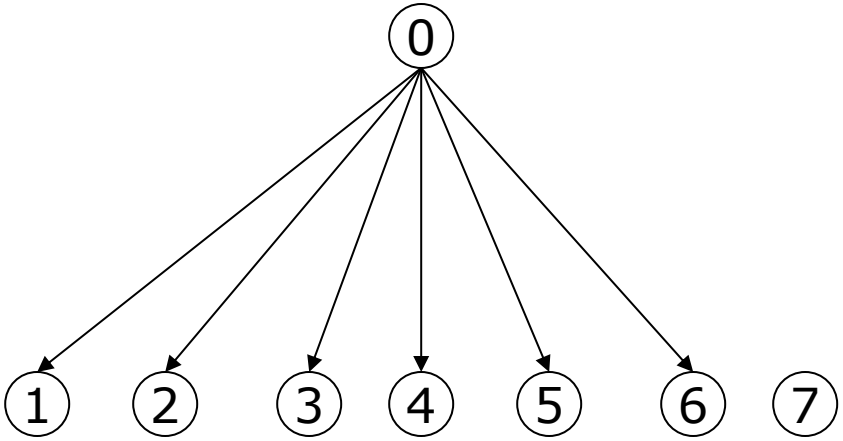
Flat Tree 算法

- root进程依次给其他进程发送消息



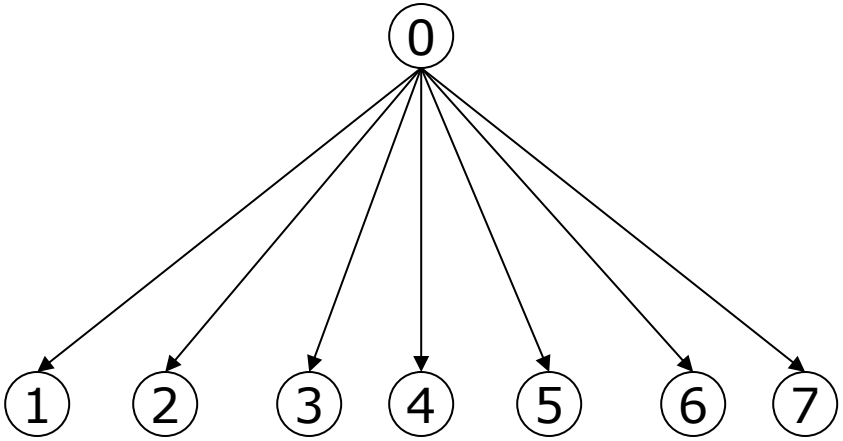
Flat Tree 算法

- root进程依次给其他进程发送消息



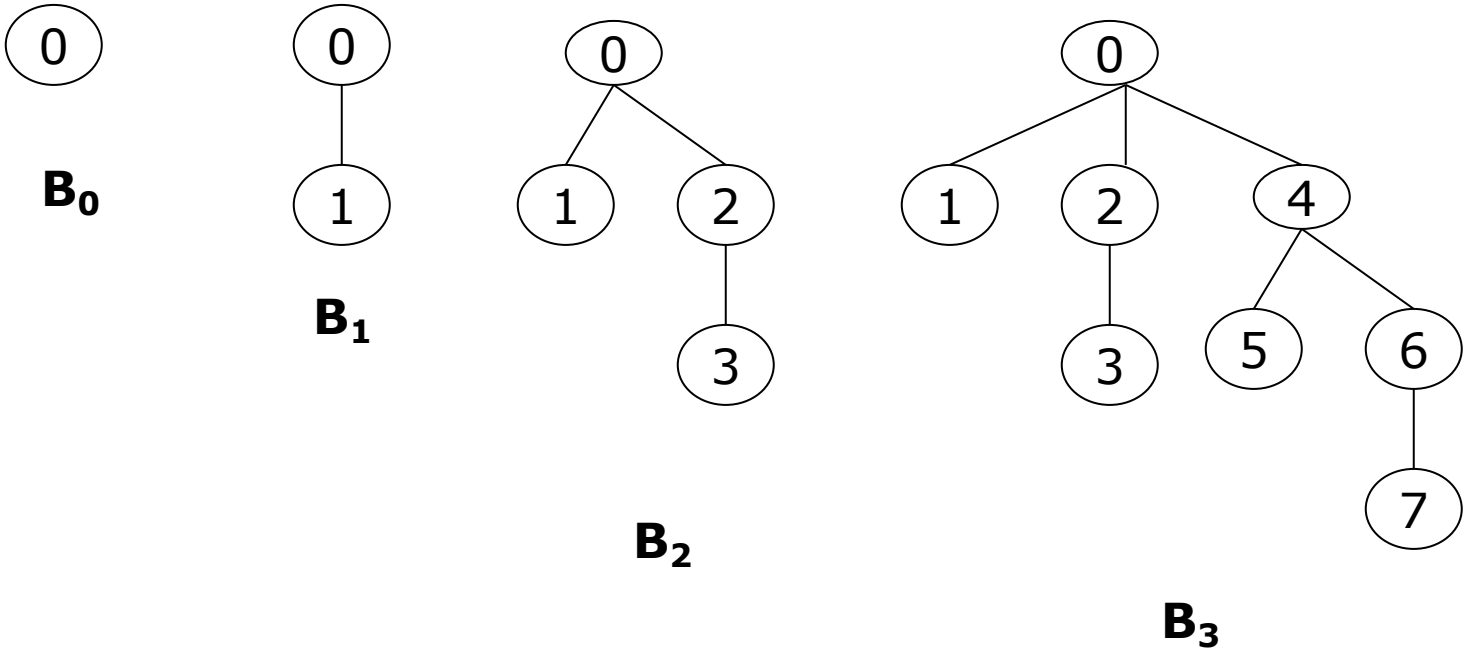
Flat Tree 算法

- root进程依次给其它进程发送消息
  - 网络带宽浪费
  - 步数多



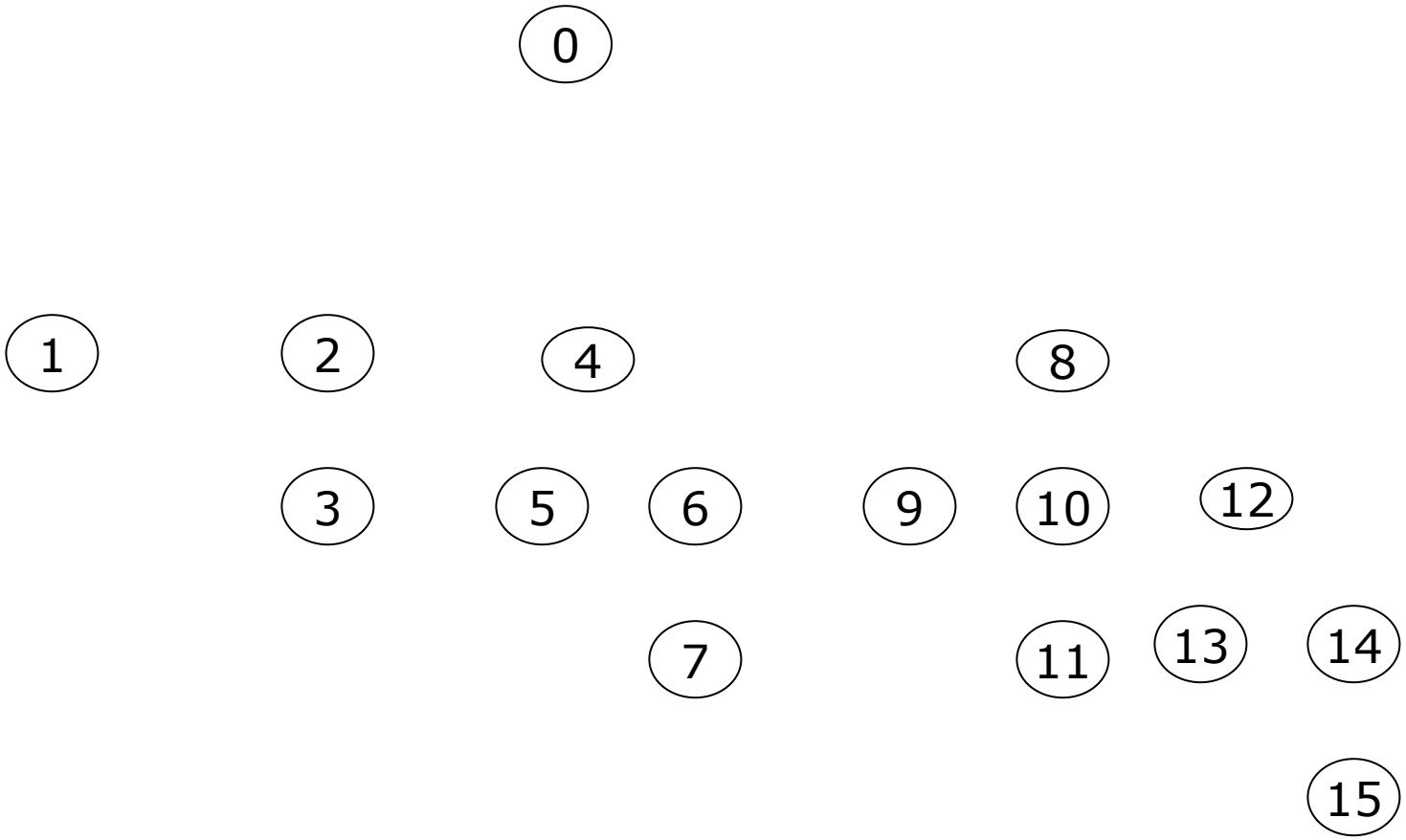
Binomial Tree

- **定义 (Binomial Tree，二项树)** 以R为根，度为k的二项树B_k 定义如下
  - 1. 如果 k=0, B_k ={R}. 即，度为0的二项树只包含一个节点R
  - 2. 如果 k>0, B_k ={R, B_0, B_1,...B_{k-1}}. 即，度为k的二项树有一个根节点R，R有k个子女，每个子女分别是度数为0, 1, 2,……, k-1 的二项树的根
- B_k 包含 2^k 个节点
- B_k 的高度为 k



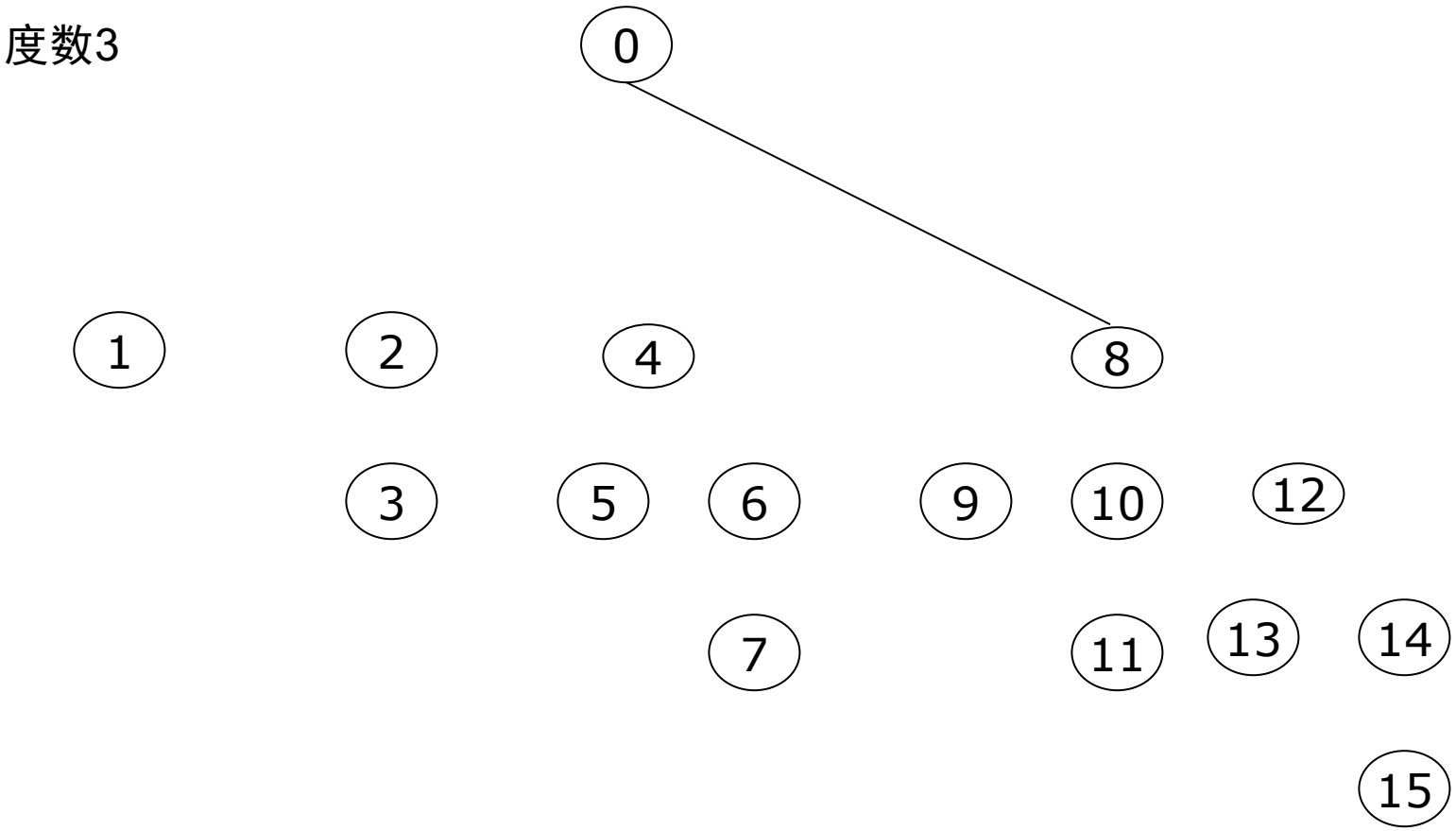
Binomial Tree 算法

- 当进程收到消息后，按照度数从大到小的顺序向子节点进程依次发送消息



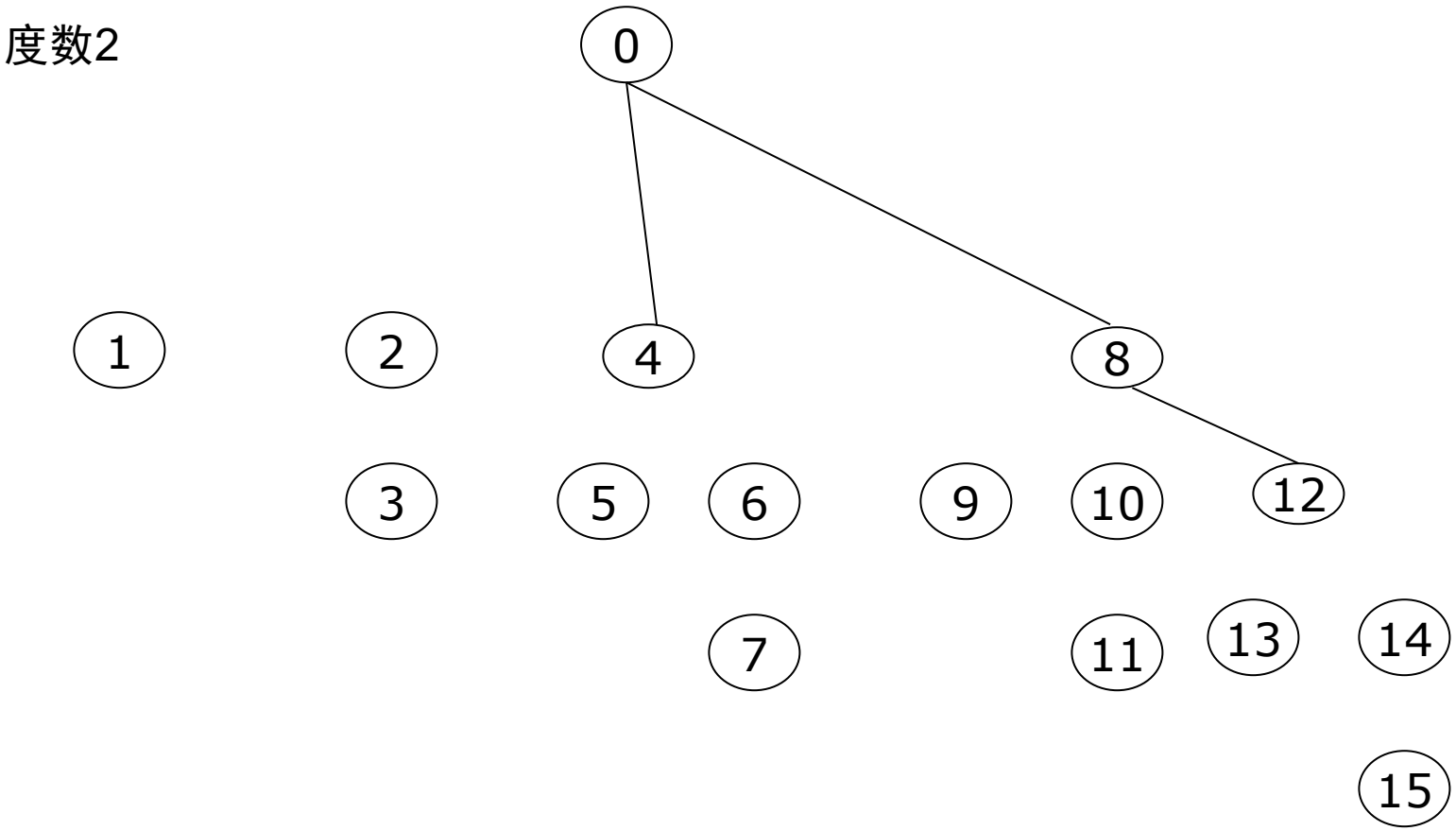
Binomial Tree 算法

- 当进程收到消息后，按照度数从大到小的顺序向子节点进程依次发送消息



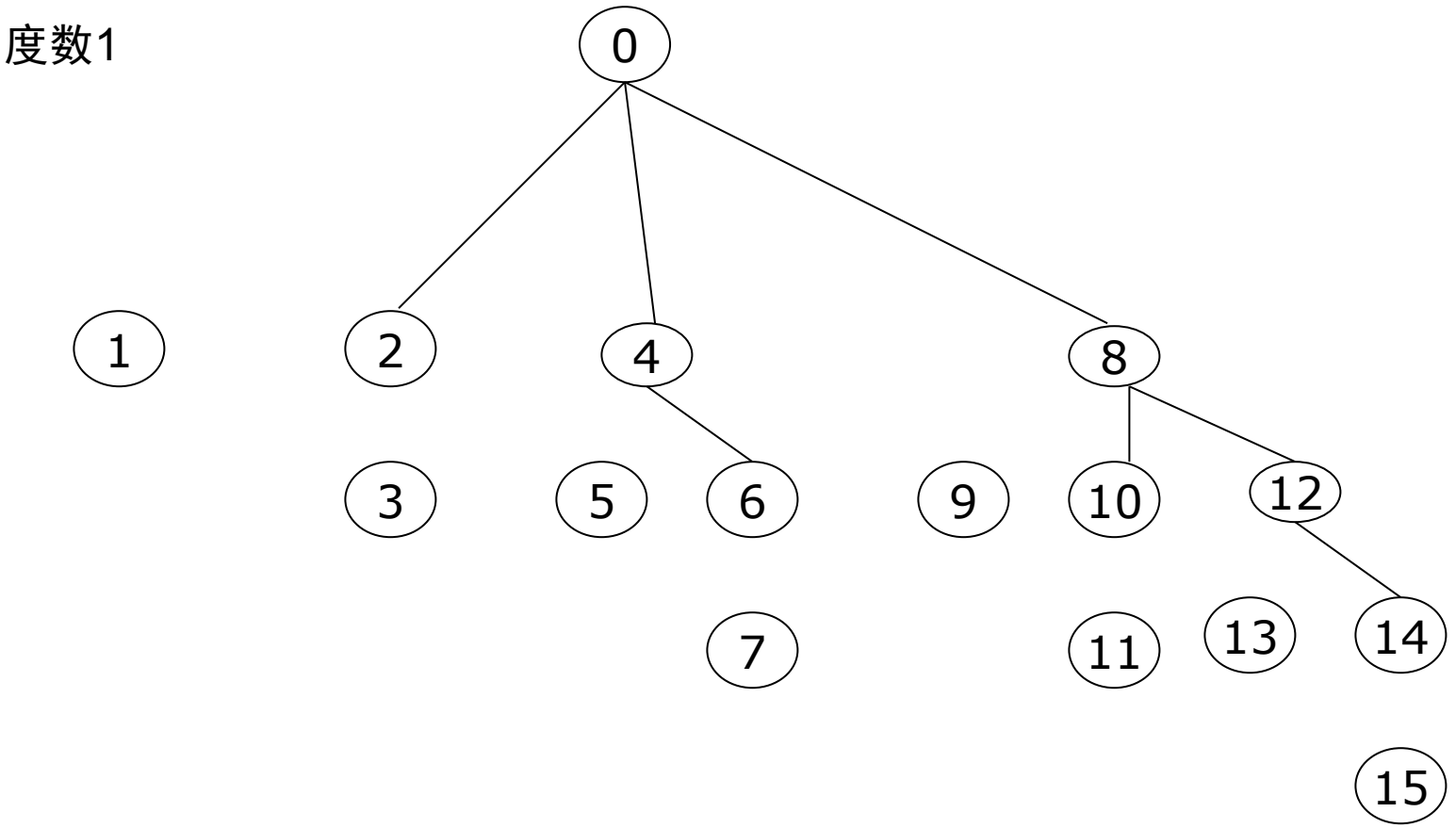
Binomial Tree 算法

- 当进程收到消息后，按照度数从大到小的顺序向子节点进程依次发送消息



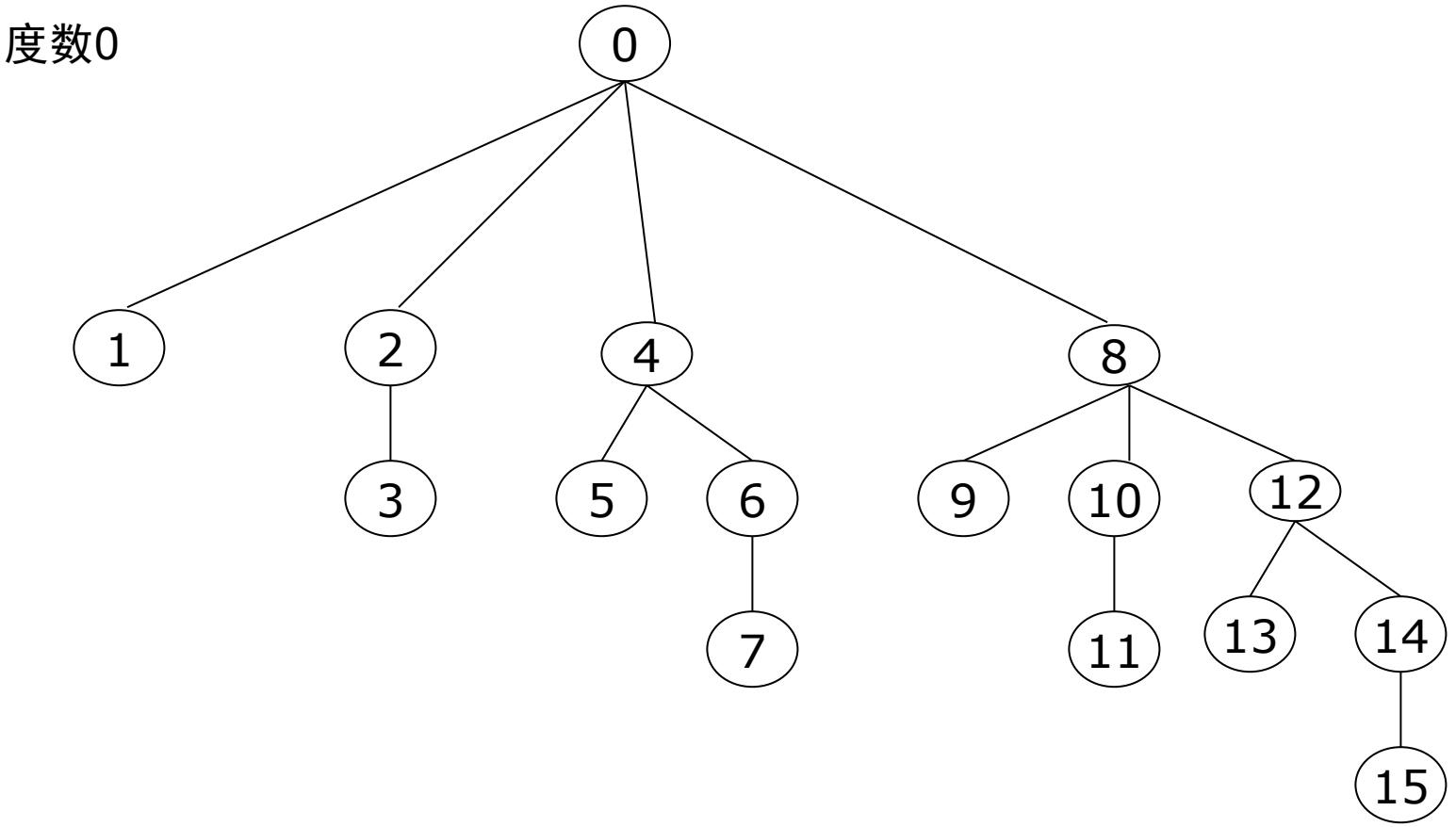
Binomial Tree 算法

- 当进程收到消息后，按照度数从大到小的顺序向子节点进程依次发送消息



Binomial Tree 算法

- 当进程收到消息后，按照度数从大到小的顺序向子节点进程依次发送消息



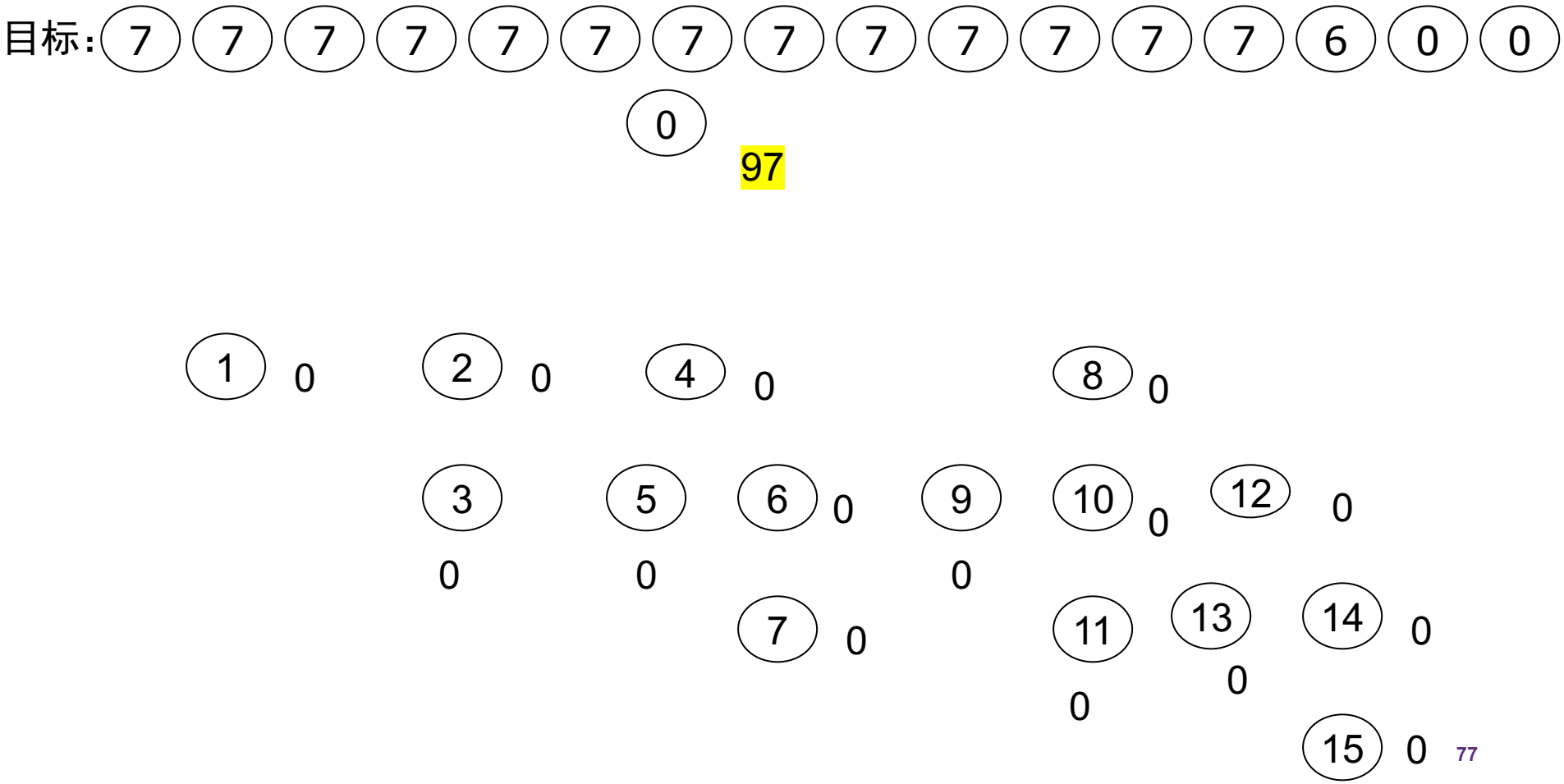
Van De Geijn 算法

- 将需要广播的数据首先均分成小块，然后均分给每一个进程
  - 类似于 MPI_Scatter，使用二项树算法
- 这些分配出去的数据最后再收集到每一个结点上
  - 类似于 MPI_Allgather
  - 消息较短：使用 Recursive Doubling 算法
  - 消息较长：使用 Ring 算法

假设 Bcast 的消息是（A0 A1 A2 A3）

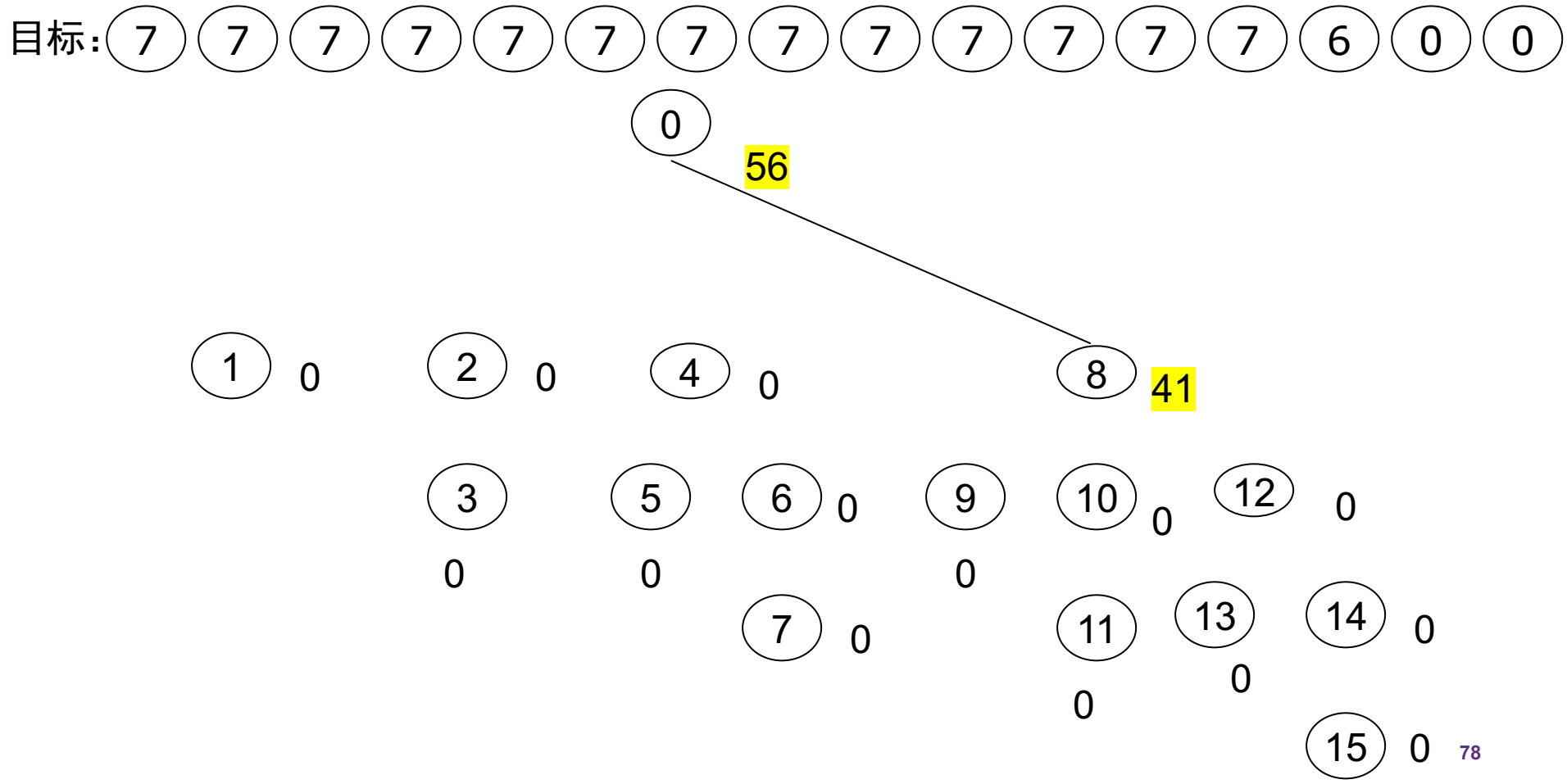
Van De Geijn 算法——Scatter部分

- Scatter 部分
  - scatter_size=(nbytes + nprocs -1) / nprocs
- nprocs =16; nbytes =97; scatter_size=7



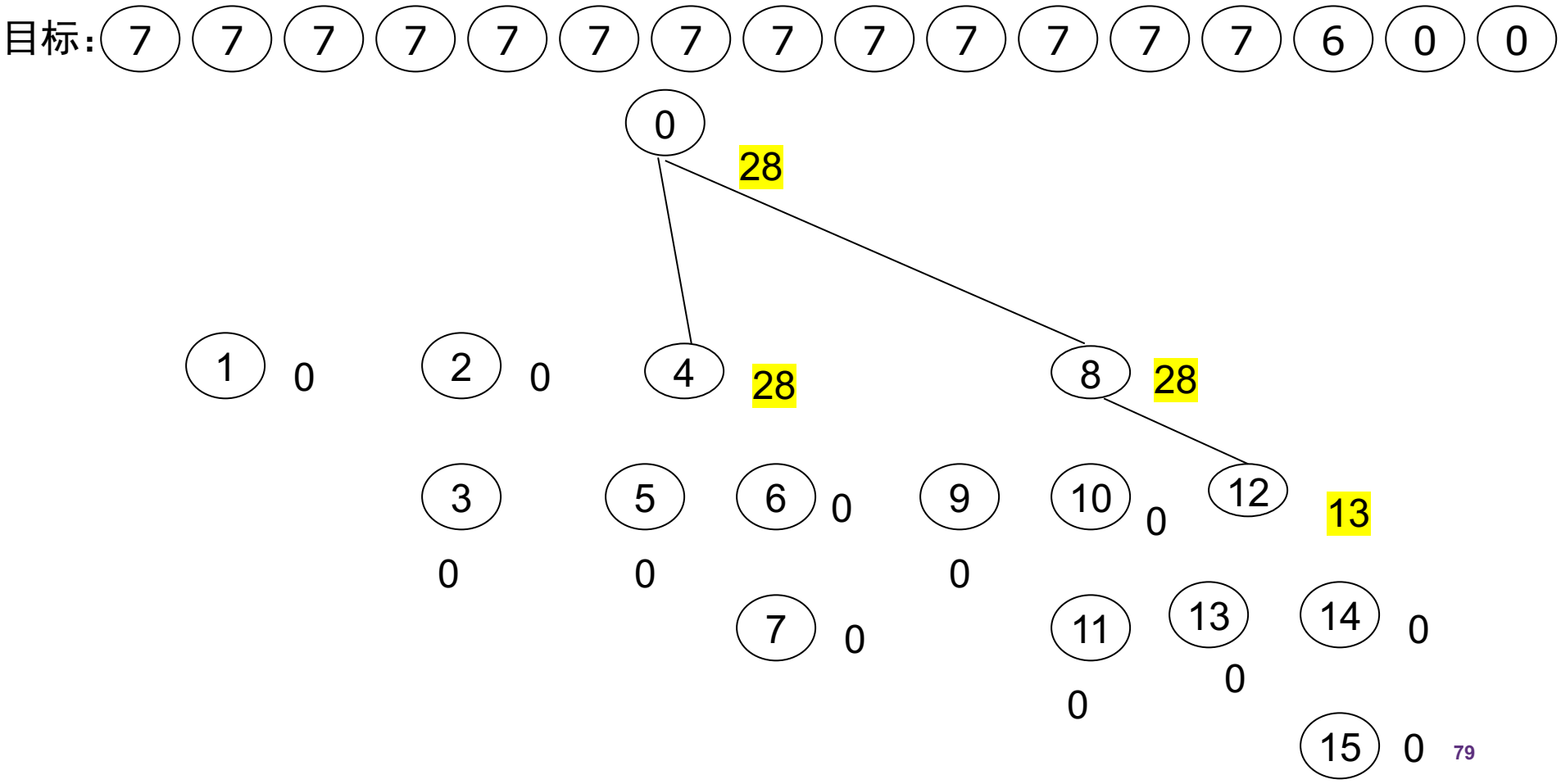
Van De Geijn 算法——Scatter部分

- Scatter 部分
  - scatter_size=(nbytes + nprocs -1) / nprocs
- nprocs =16; nbytes =97; scatter_size=7



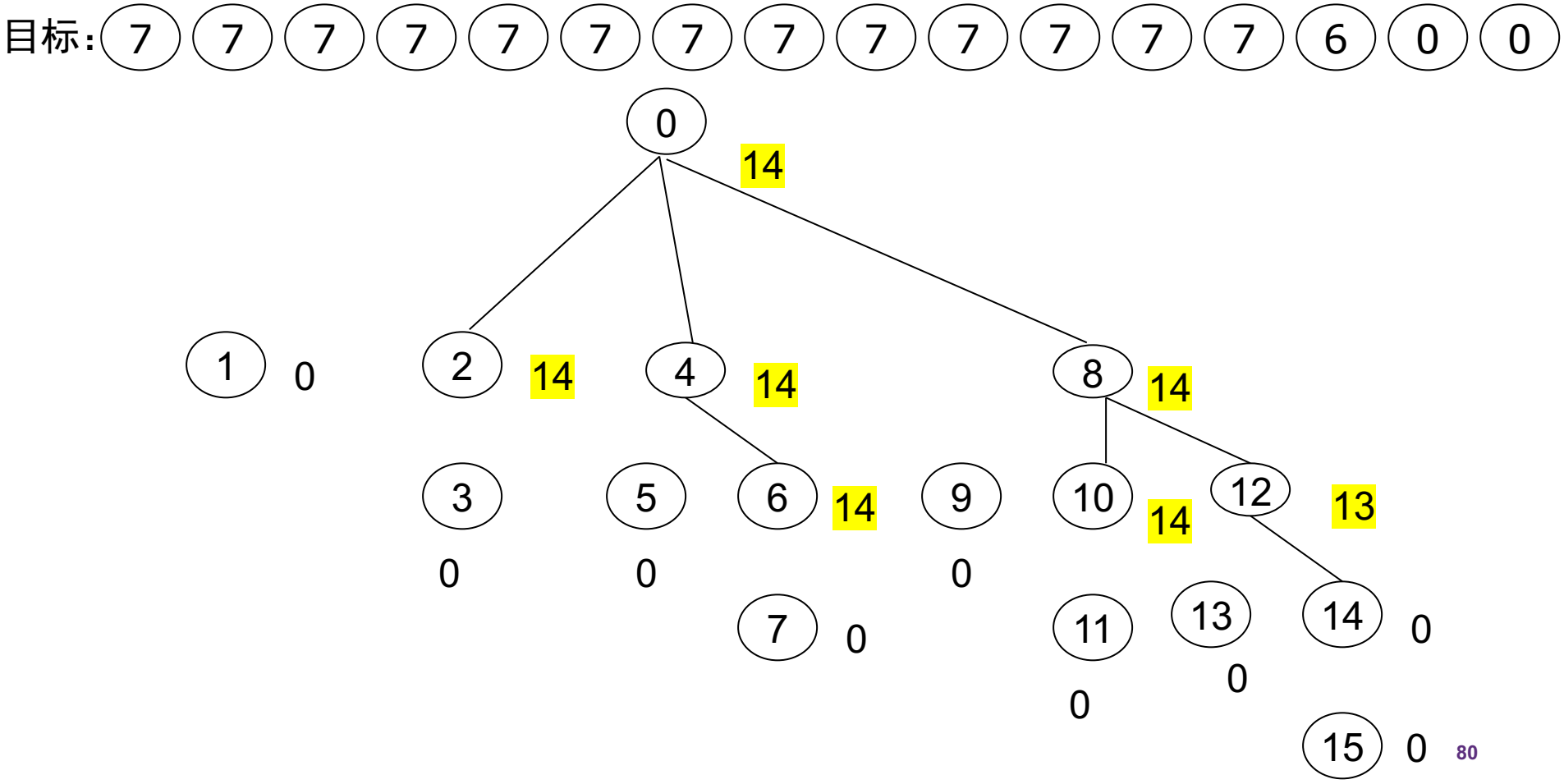
Van De Geijn 算法——Scatter部分

- Scatter 部分
  - scatter_size=(nbytes + nprocs -1) / nprocs
- nprocs =16; nbytes =97; scatter_size=7



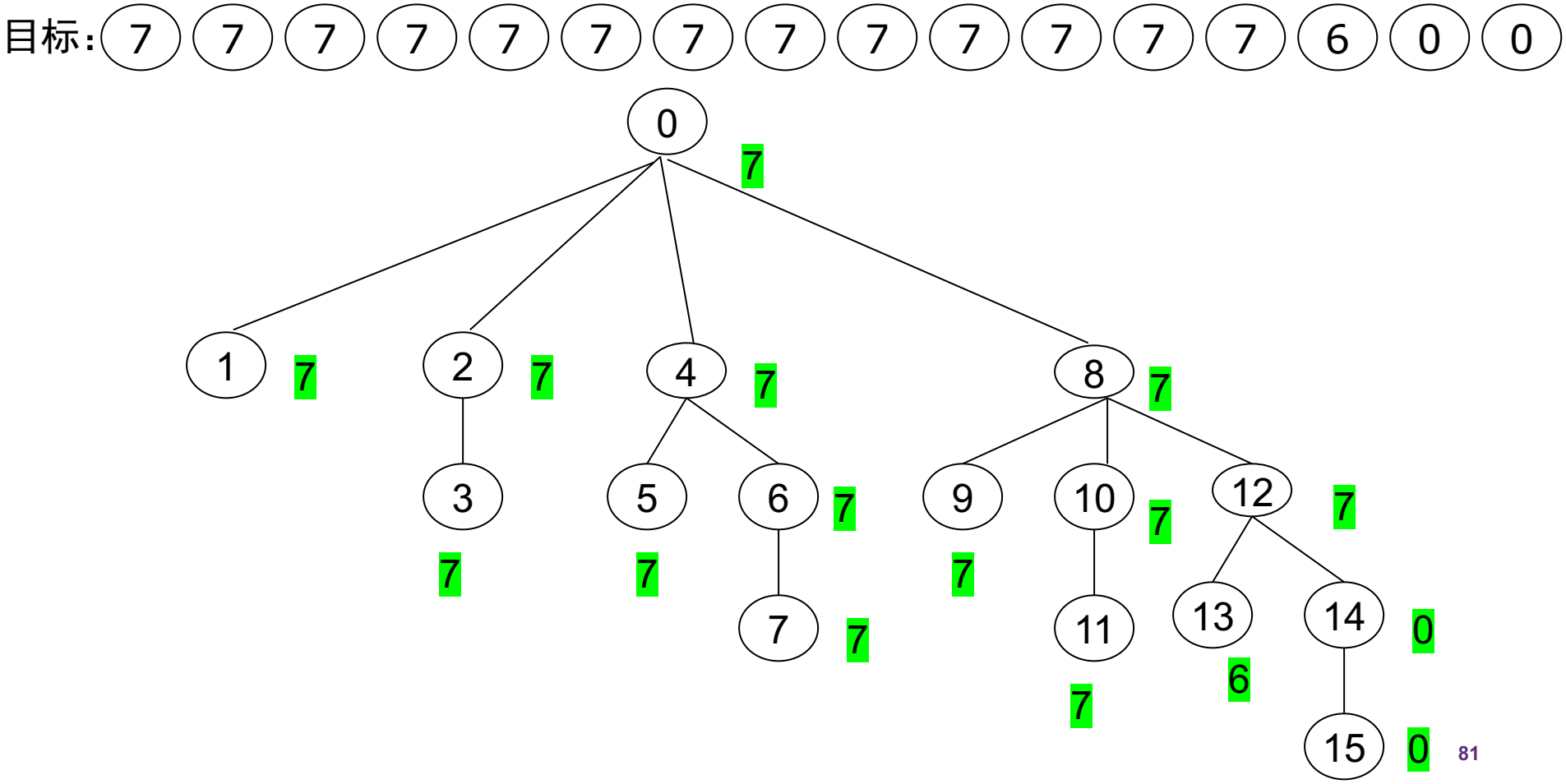
Van De Geijn 算法——Scatter部分

- Scatter 部分
  - scatter_size=(nbytes + nprocs -1) / nprocs
- nprocs =16; nbytes =97; scatter_size=7



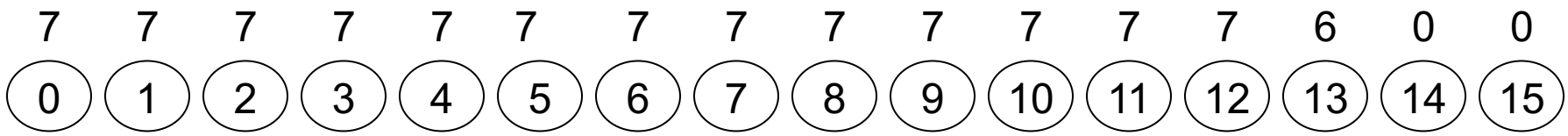
Van De Geijn 算法——Scatter部分

- Scatter 部分
  - scatter_size=(nbytes + nprocs -1) / nprocs
- nprocs =16; nbytes =97; scatter_size=7



Van De Geijn 算法——Allgather部分

- 假设使用 Recursive Doubling 算法



Step1

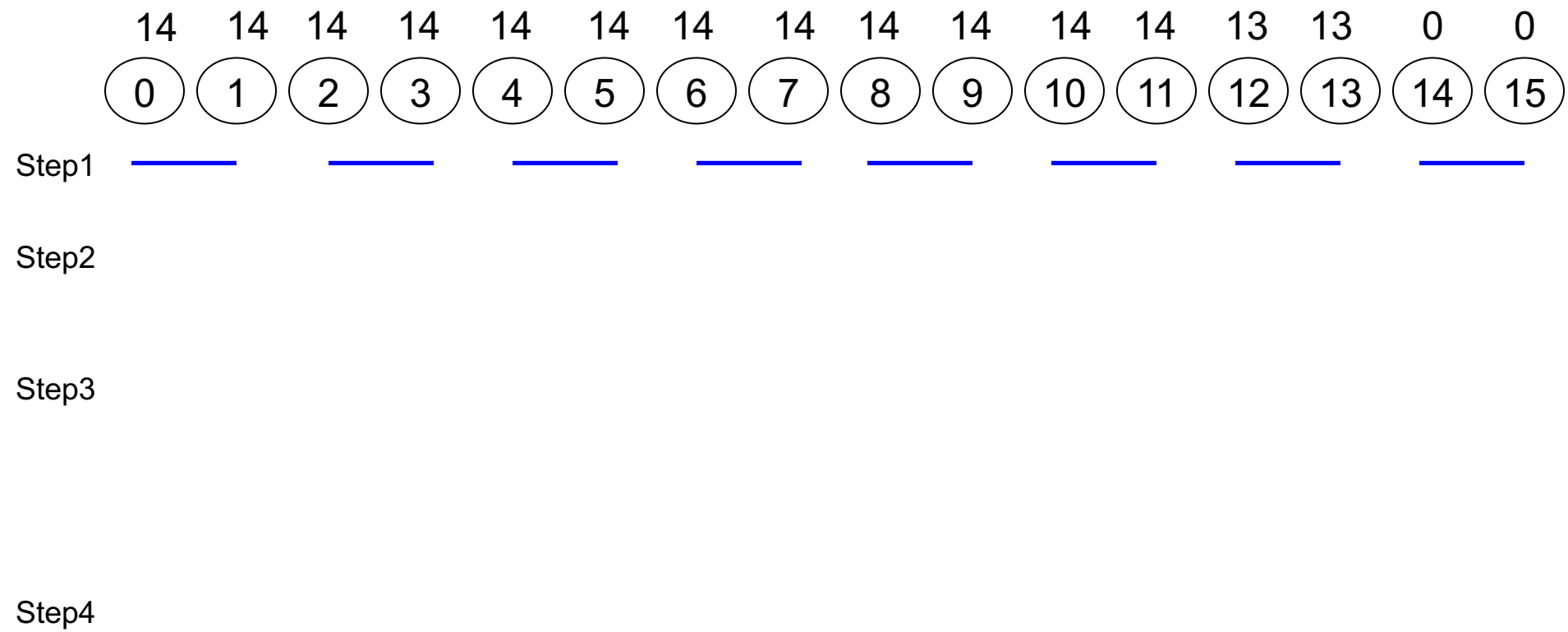
Step2

Step3

Step4

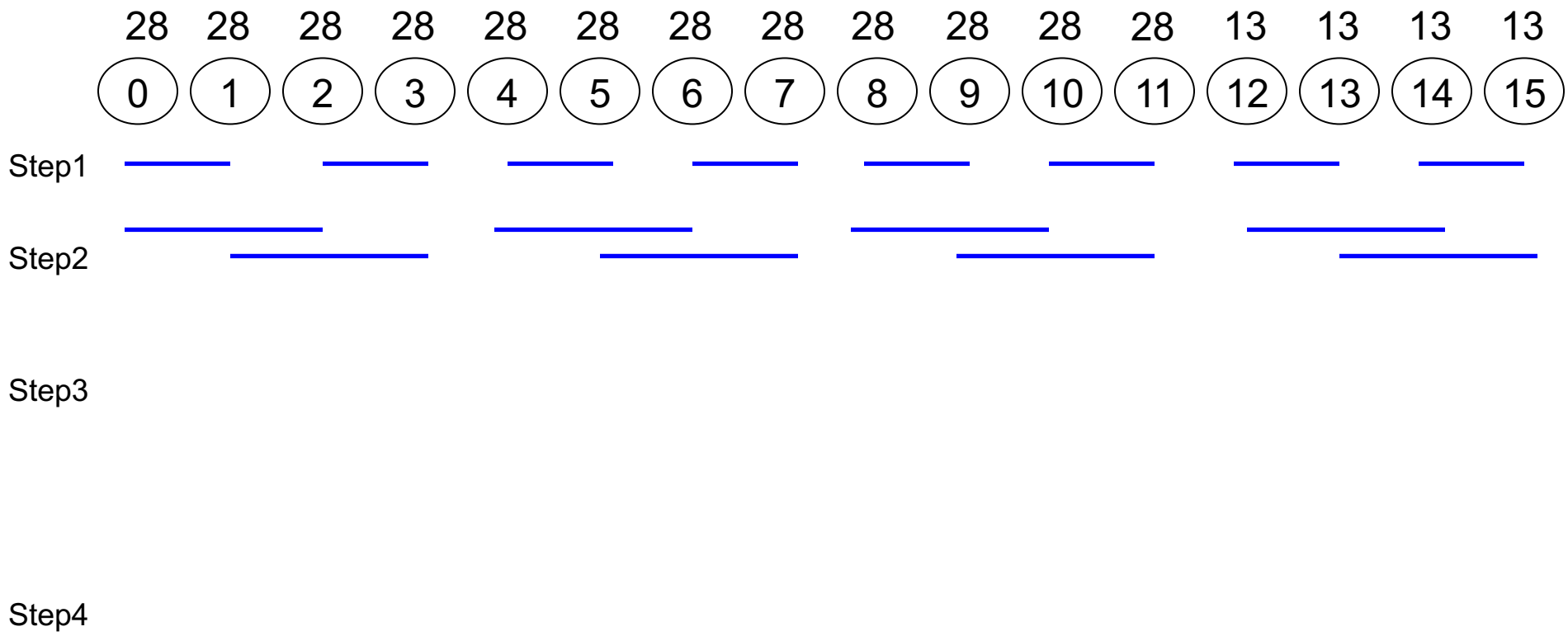
Van De Geijn 算法——Allgather部分

- 使用 Recursive Doubling 算法



Van De Geijn 算法——Allgather部分

- 使用 Recursive Doubling 算法

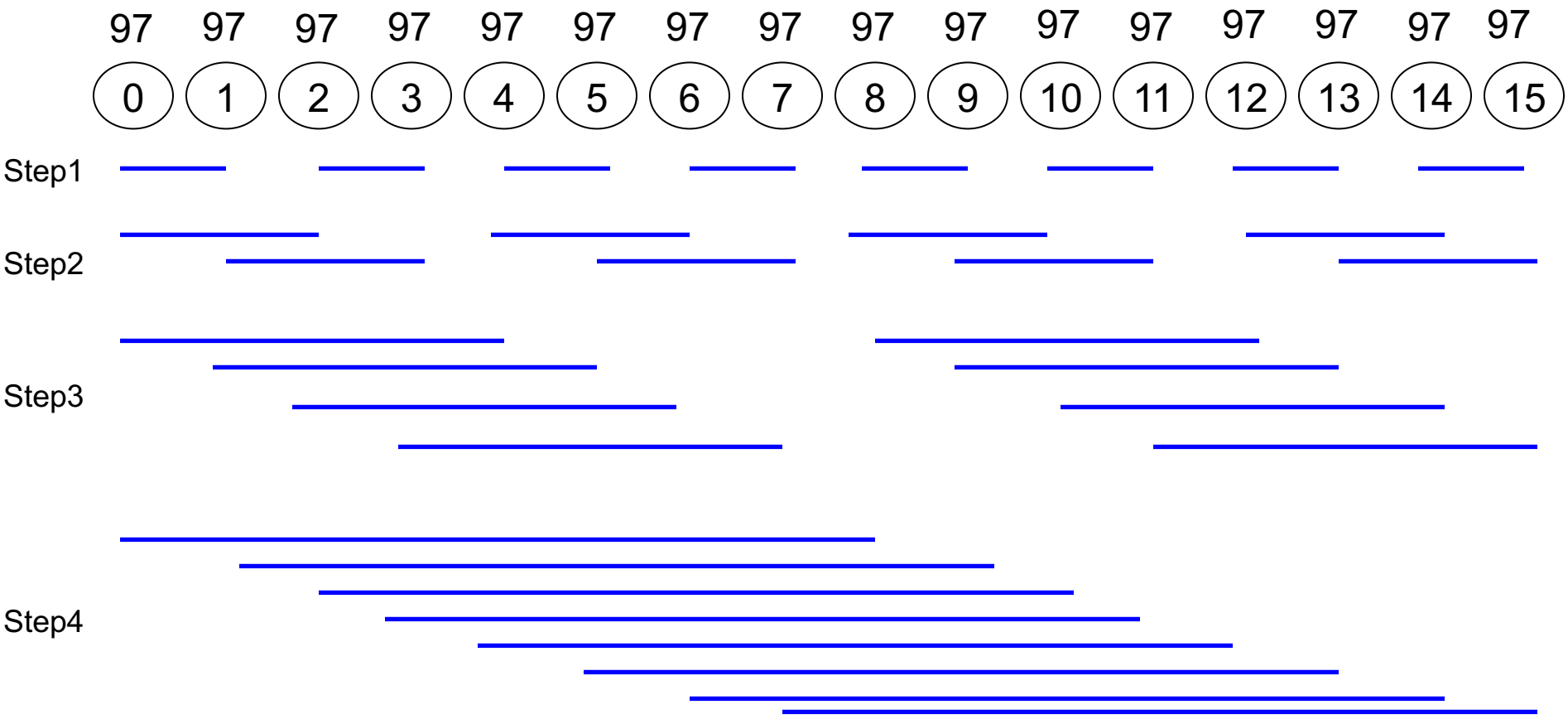


Van De Geijn 算法——Allgather部分

- 使用 Recursive Doubling 算法

Van De Geijn 算法——Allgather部分

- 使用 Recursive Doubling 算法



MPI_Bcast 算法性能分析

算法	开销	特点
Flat Tree （扁平树算法）	$(\mathrm{P}-1)(\alpha+n\beta)$	简单，但浪费带宽太多
Binomial Tree （二项树算法）	$\lceil \lg \mathrm{P} \rceil (\alpha+n\beta)$	步数较少，总延迟小，适合短消息
Van de Geijn (Scatter-Allgather) 假设消息极长，Allgather 使用 Ring 算法	$\lceil \lg \mathrm{P} \rceil \alpha + \frac{\mathrm{P}-1}{\mathrm{P}} n\beta$ + $(\mathrm{P}-1)\alpha + \frac{\mathrm{P}-1}{\mathrm{P}} n\beta$ = $(\lceil \lg \mathrm{P} \rceil + \mathrm{P} - 1)\alpha + 2\frac{\mathrm{P}-1}{\mathrm{P}} n\beta$	通信总量少，适合长消息。 进程越多、消息越长，较于二项树算法优势越明显

MPI_Bcast 算法使用

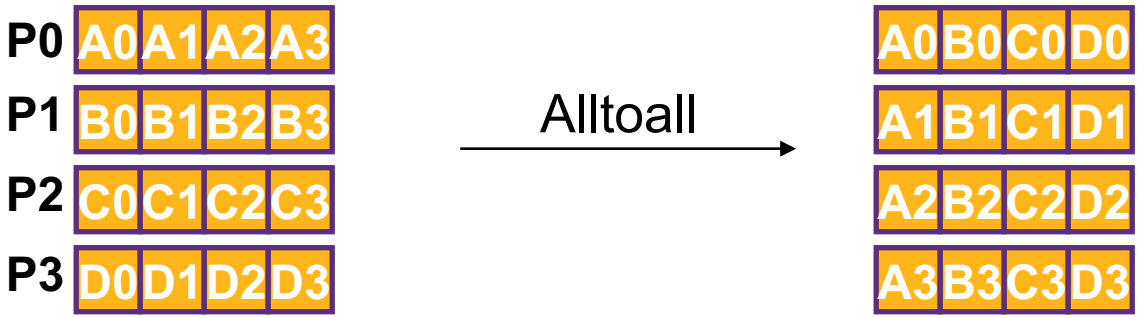
- Infiniband 网络：
  - 短消息和进程数小于等于8时使用 Binomial Tree
  - 其它情况使用 Van De Geijn
- Ethernet 网络：
  - 短消息或进程数小于等于8时使用 Binomial Tree
  - 中等长度消息或进程数为2的幂时使用 Van De Geijn
  - 其它情况使用 Ring

典型集合通信实现

- MPI_Allgather
- MPI_Bcast
- **MPI_Alltoall**
- MPI_Allreduce

MPI_Alltoall 算法实现

- Recursive Doubling （递推倍增算法）
- Bruck
- Irecv-Isend
- Pairwise Exchange （成对交换算法）



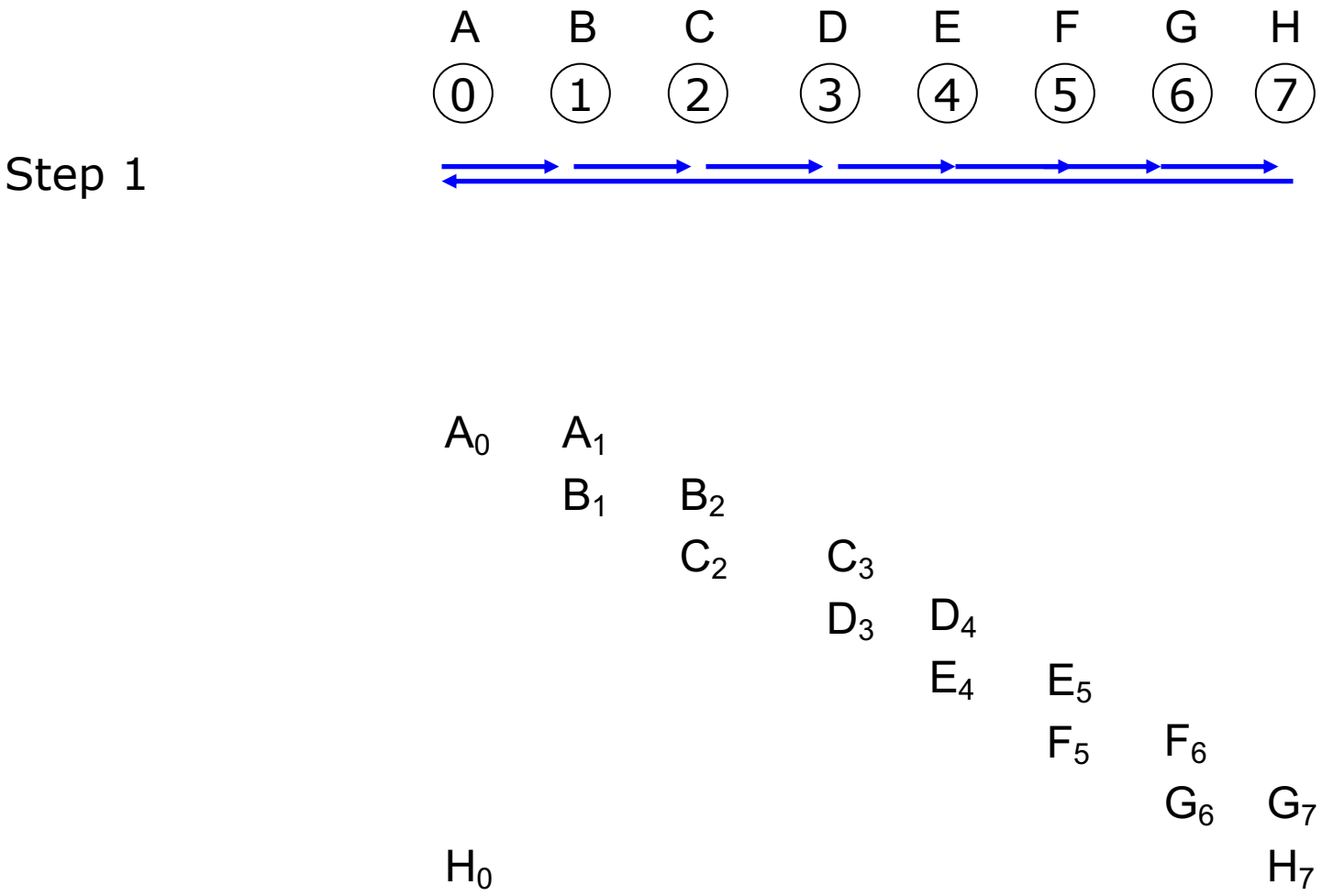
Irecv-Isend 算法

- 每个进程非阻塞地依次从其它进程接收数据，并依次向其它进程发送数据

0	1	2	3
Irecv from 1	**Irecv from 2**	**Irecv from 3**	**Irecv from 0**
Irecv from 2	**Irecv from 3**	**Irecv from 0**	**Irecv from 1**
Irecv from 3	**Irecv from 0**	**Irecv from 1**	**Irecv from 2**
Isend to 1	**Isend to 2**	**Isend to 3**	**Isend to 0**
Isend to 2	**Isend to 3**	**Isend to 0**	**Isend to 1**
Isend to 3	**Isend to 0**	**Isend to 1**	**Isend to 2**
Waitall	**Waitall**	**Waitall**	**Waitall**

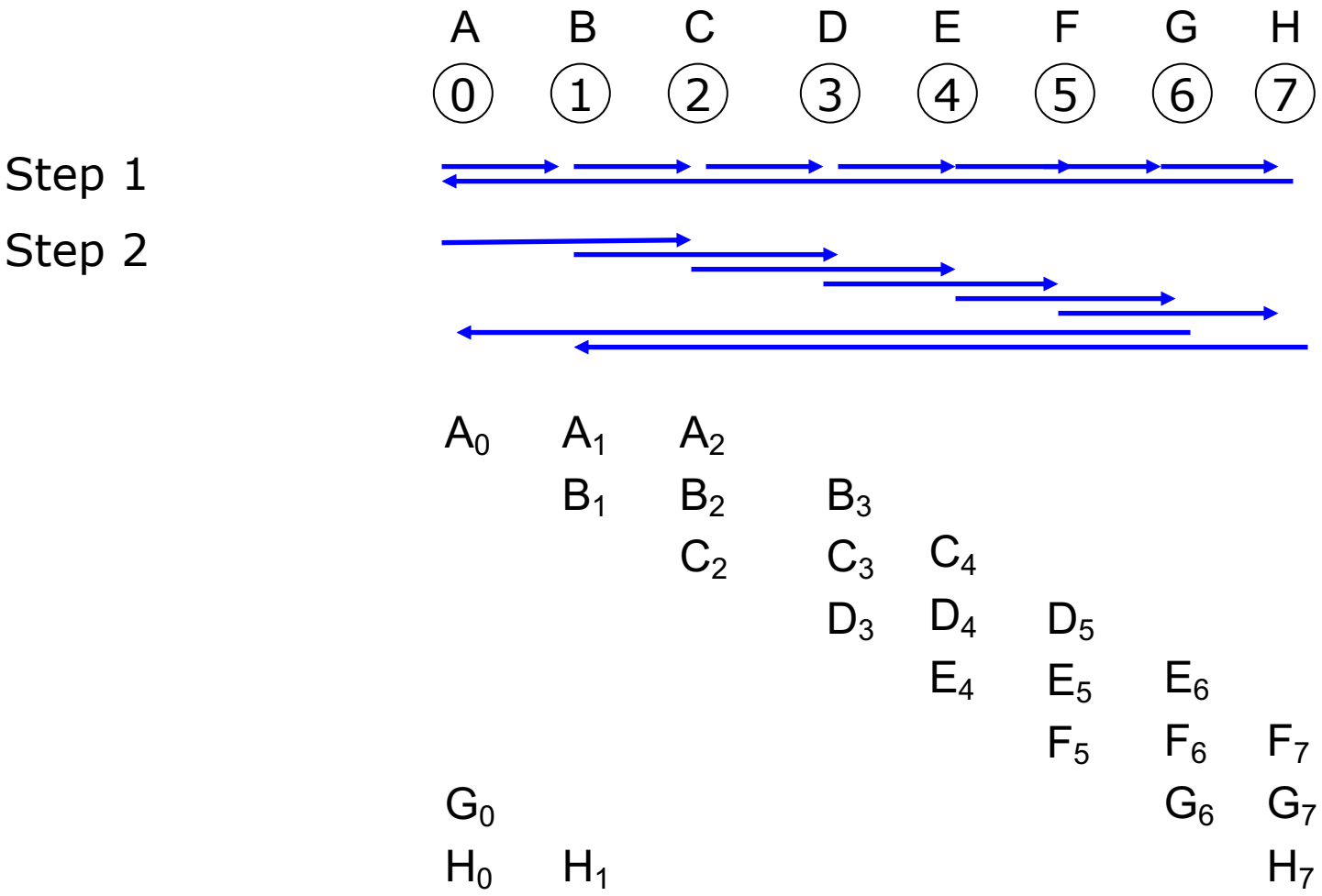
Pairwise Exchange 算法

- 在第k步，每个进程 i 从进程 i-k 接收数据，并向进程 i+k 发送数据



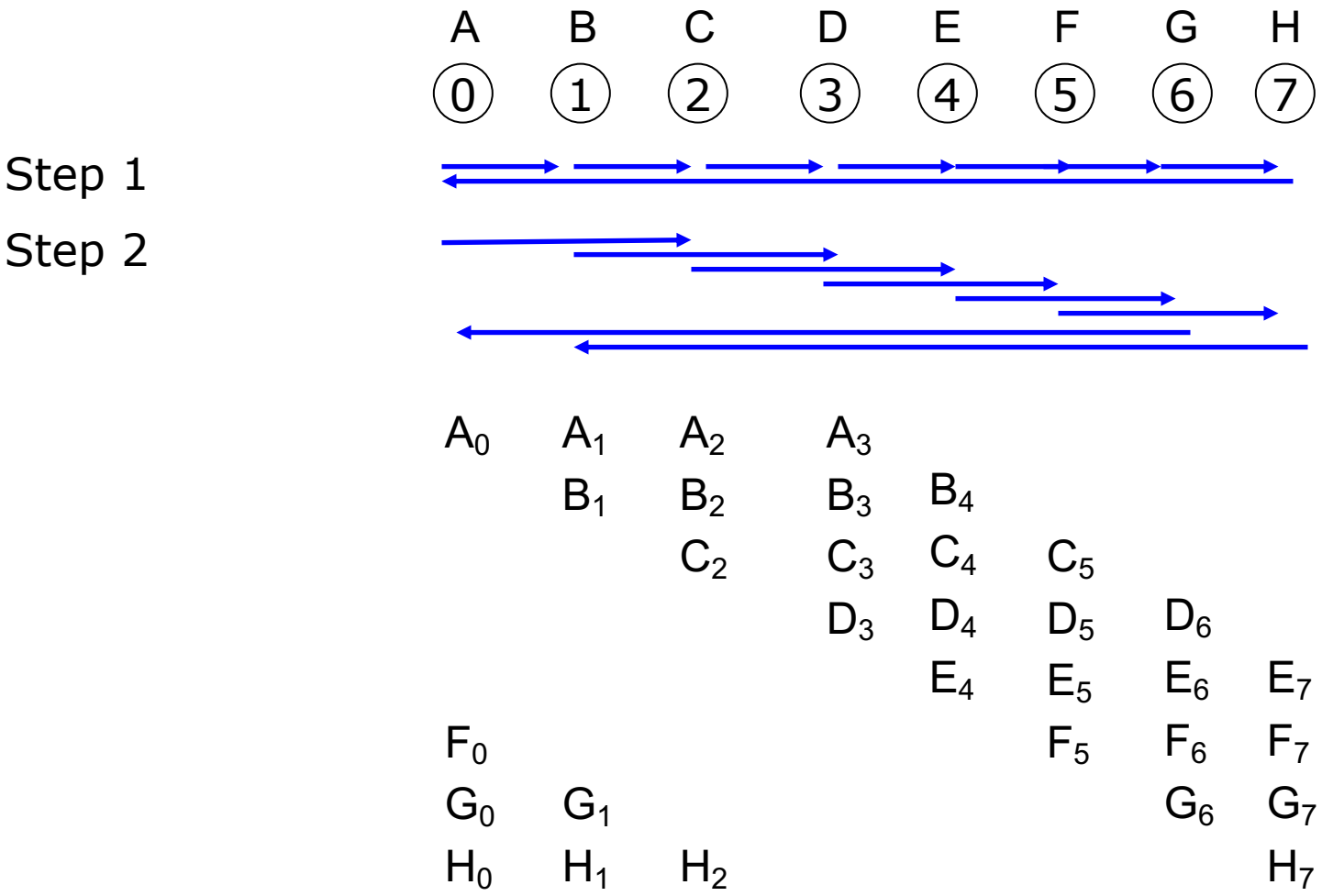
Pairwise Exchange 算法

- 在第k步，每个进程 i 从进程 i-k 接收数据，并向进程 i+k 发送数据



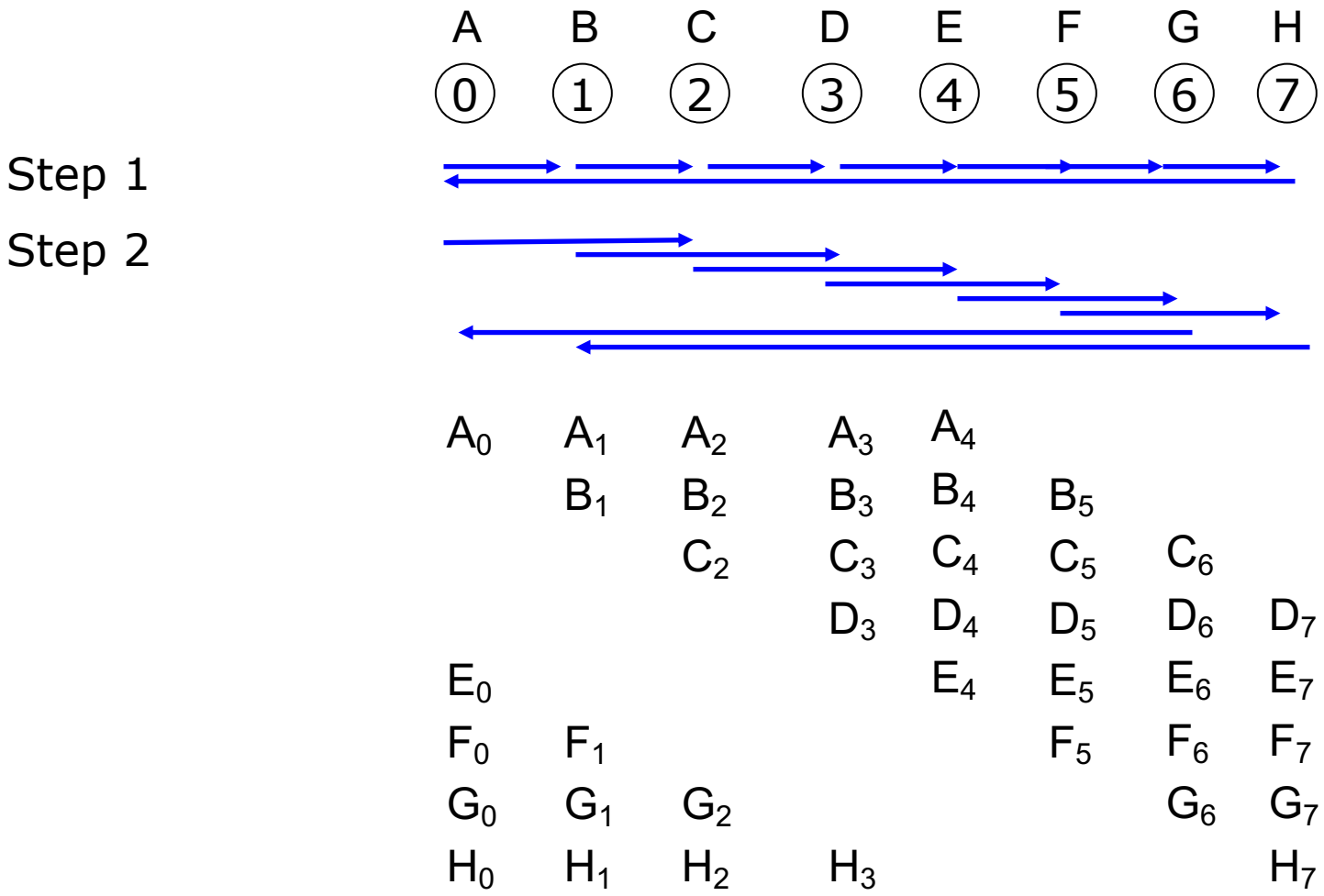
Pairwise Exchange 算法

- 在第k步，每个进程 i 从进程 i-k 接收数据，并向进程 i+k 发送数据



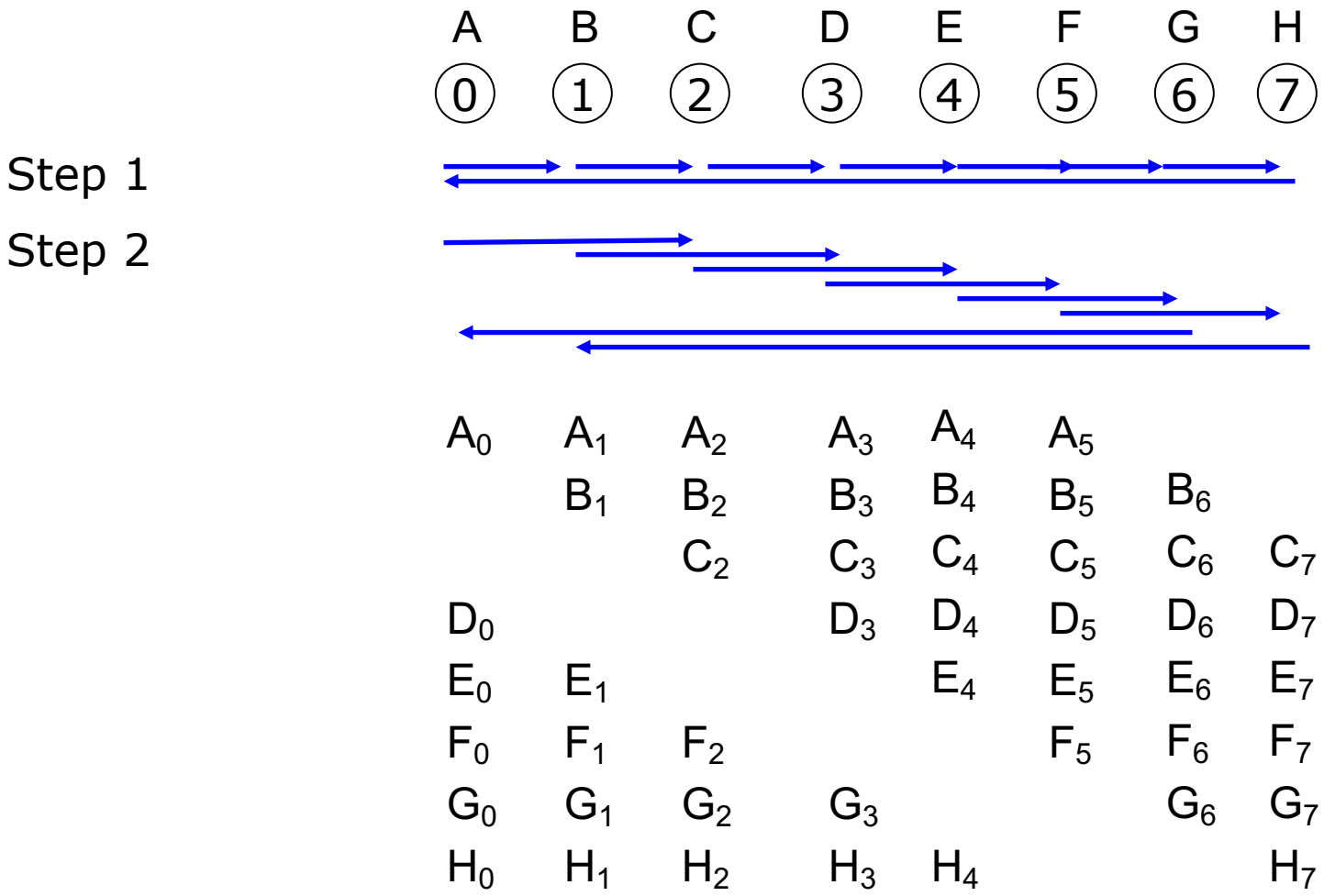
Pairwise Exchange 算法

- 在第k步，每个进程 i 从进程 i-k 接收数据，并向进程 i+k 发送数据



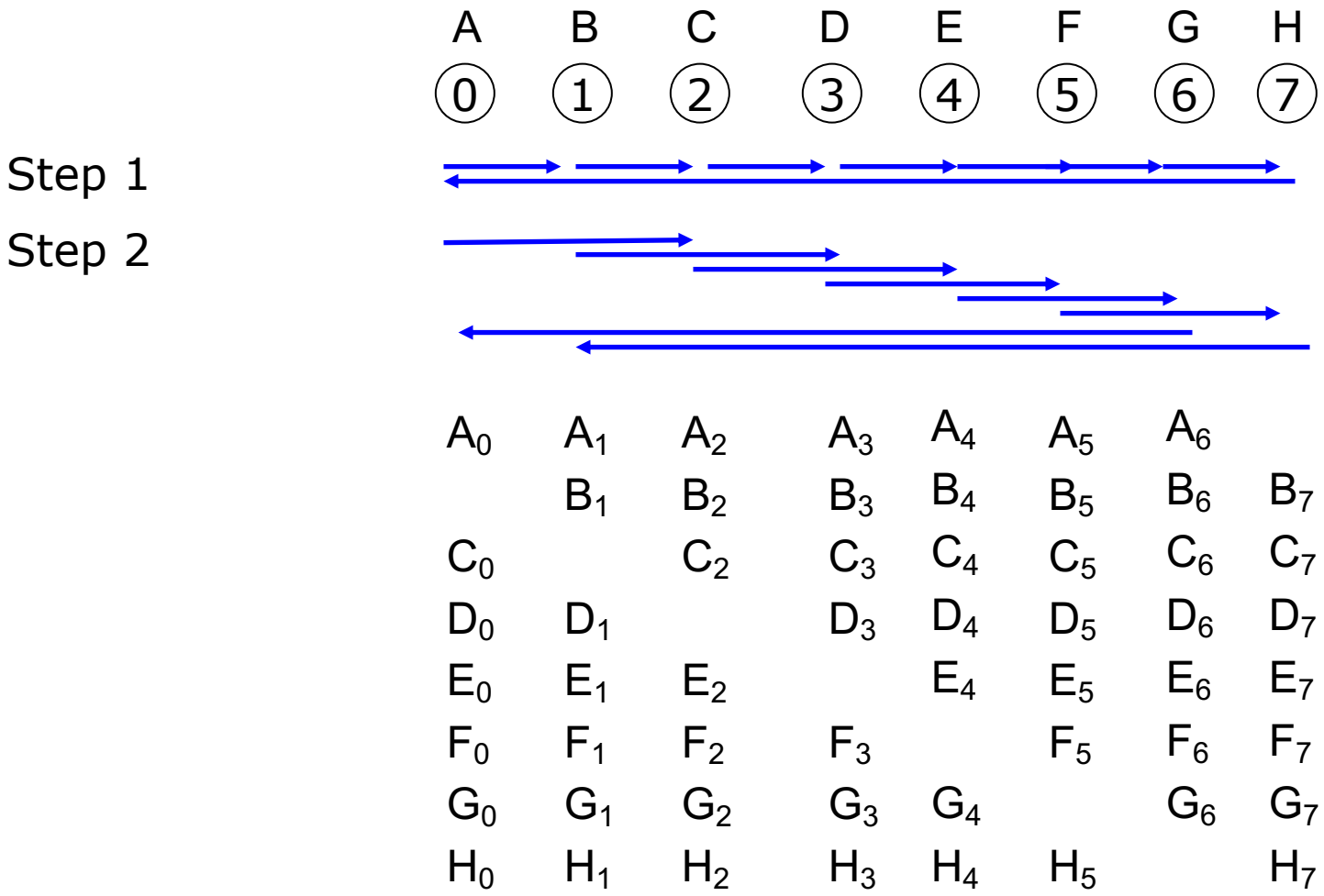
Pairwise Exchange 算法

- 在第k步，每个进程 i 从进程 i-k 接收数据，并向进程 i+k 发送数据



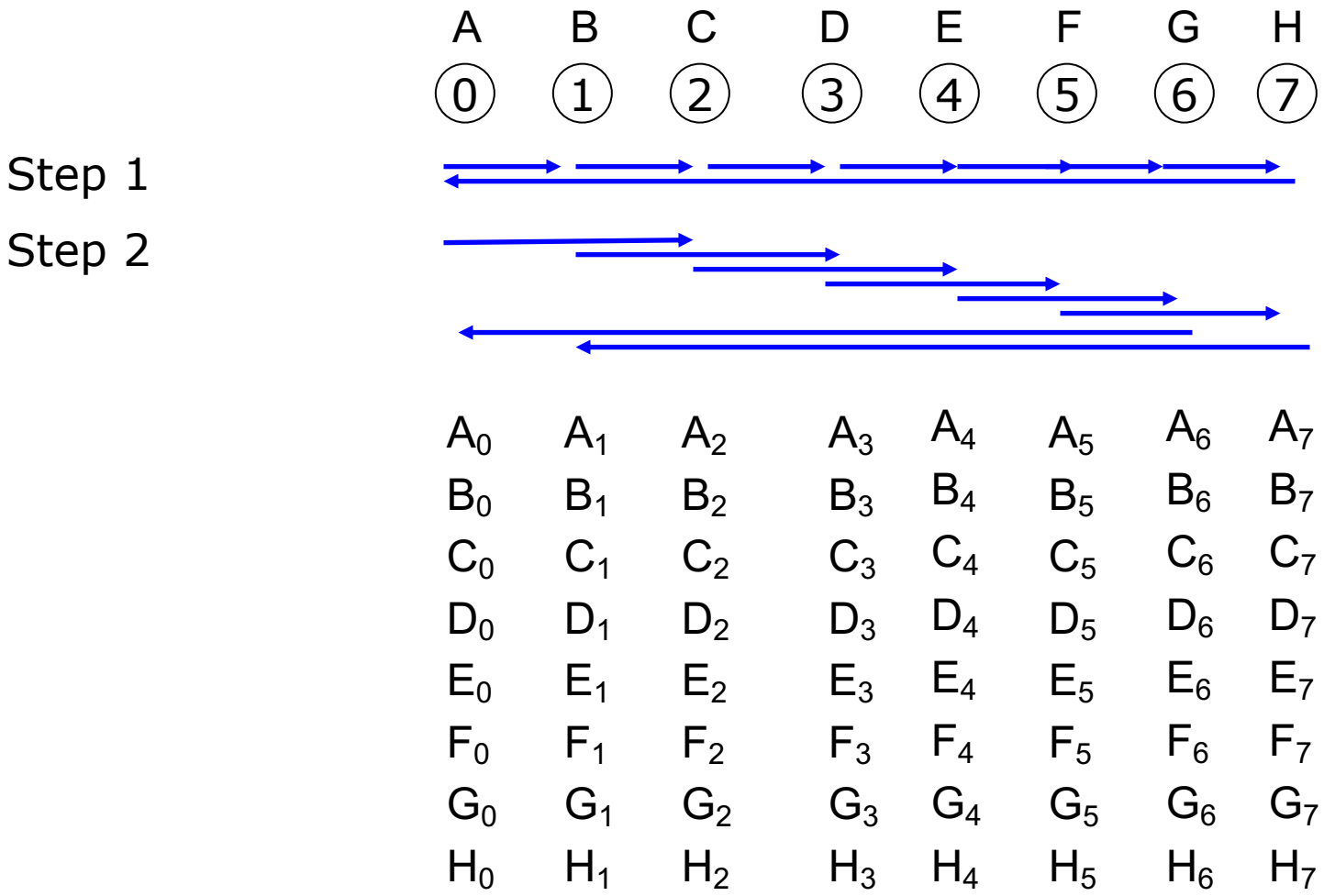
Pairwise Exchange 算法

- 在第k步，每个进程 i 从进程 i-k 接收数据，并向进程 i+k 发送数据



Pairwise Exchange 算法

- 在第k步，每个进程 i 从进程 i-k 接收数据，并向进程 i+k 发送数据



MPI_Alltoall 算法性能分析

算法	开销	特点
Recursive Doubling（递推倍增算法）	$lg\mathrm{P}\ \alpha + n\ \beta$	延迟低，但需要额外buffer，适合短消息
Bruck	$lg\mathrm{P}\ \alpha + \frac{n}{2} lg\mathrm{P}\ \beta$ （2的幂） $\lceil \mathrm{lgP} \rceil \alpha + (\frac{n}{2} \lceil \mathrm{lgP} \rceil + \frac{n}{\mathrm{P}} (P - 2^{\lfloor lg\mathrm{P} \rfloor}))\ \beta$（非2的幂）	延迟低，但需要最后调整，适合短消息
Irecv-Isend	$(\mathrm{P} - 1)\ \alpha + n\ \beta$	适合中等长度消息
Pairwise Exchange（成对交换算法）	$(\mathrm{P} - 1)\ \alpha + n\ \beta$	适合长消息

n 为一个进程需要发送或接收的数据总量

MPI_Alltoall 算法使用

- Infiniband 网络：
  - 短消息使用 Recursive Doubling
  - 中等长度消息使用 Irecv-Isend
  - 长消息使用 Pairwise Exchange
- Ethernet 网络：
  - 短消息使用 Bruck
  - 中等长度消息使用 Irecv-Isend
  - 长消息使用 Pairwise Exchange

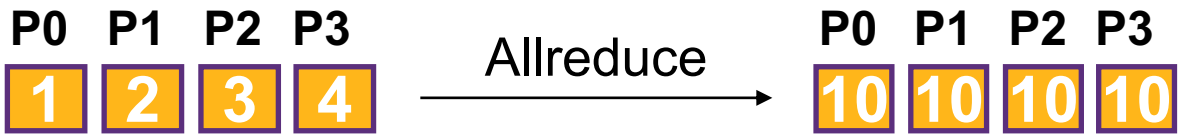
典型集合通信实现

- MPI_Allgather
- MPI_Bcast
- MPI_Alltoall
- **MPI_Allreduce**

MPI集合通信：MPI_Allreduce

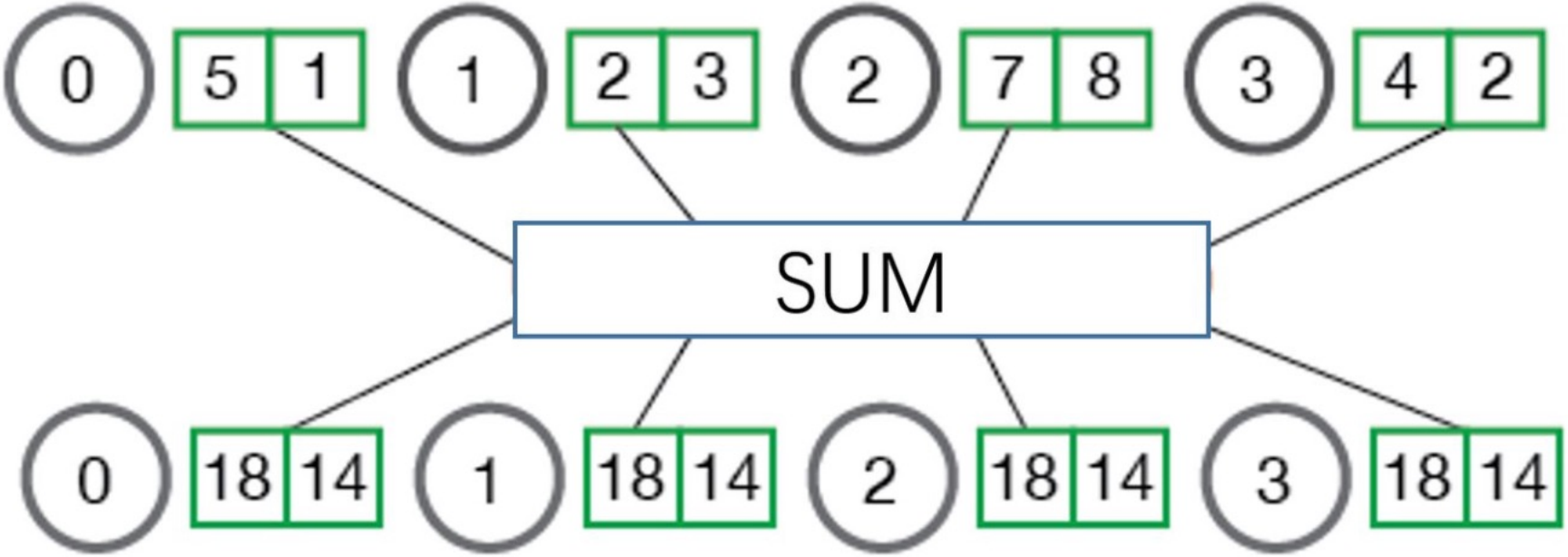
- MPI_Allreduce(&sendbuf, &recvbuf, count, datatype, op, comm)
  - 执行归约操作并将结果存至所有进程
  - 等效于MPI_Reduce操作后再进行MPI_Bcast操作

count = 1; op = MPI_SUM



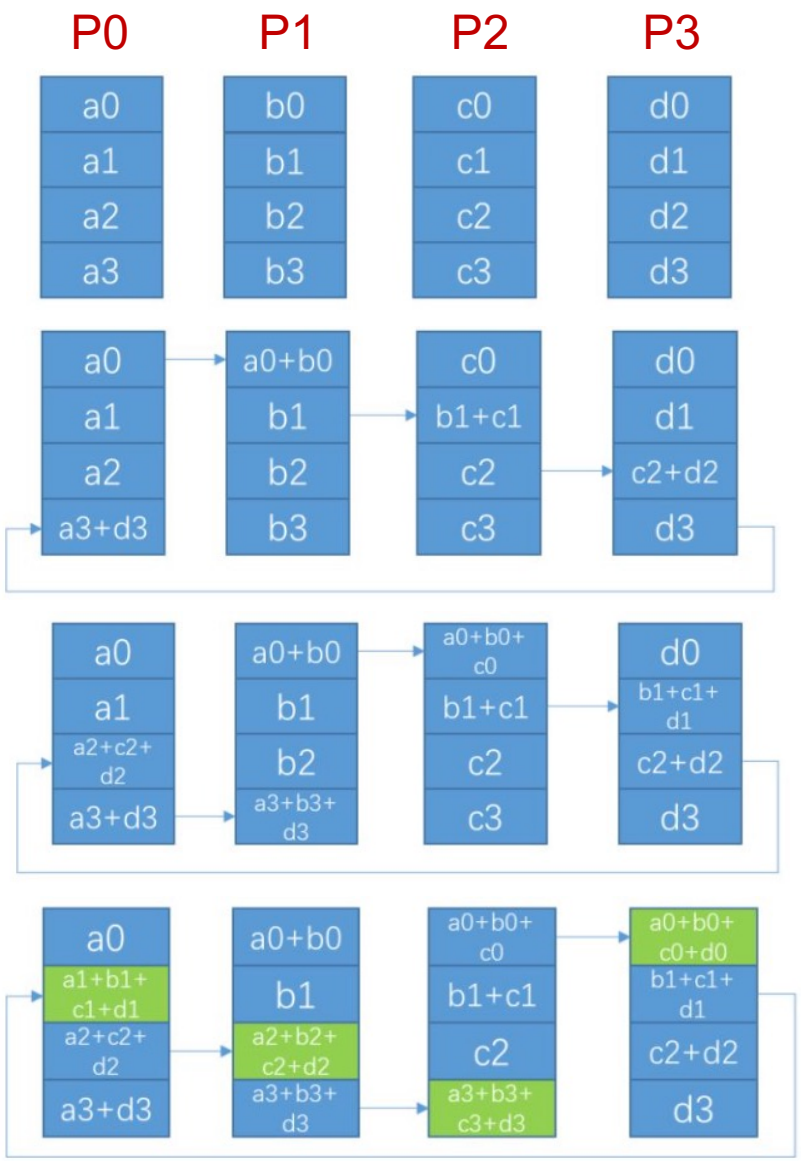
MPI_Allreduce

- 以SUM为例



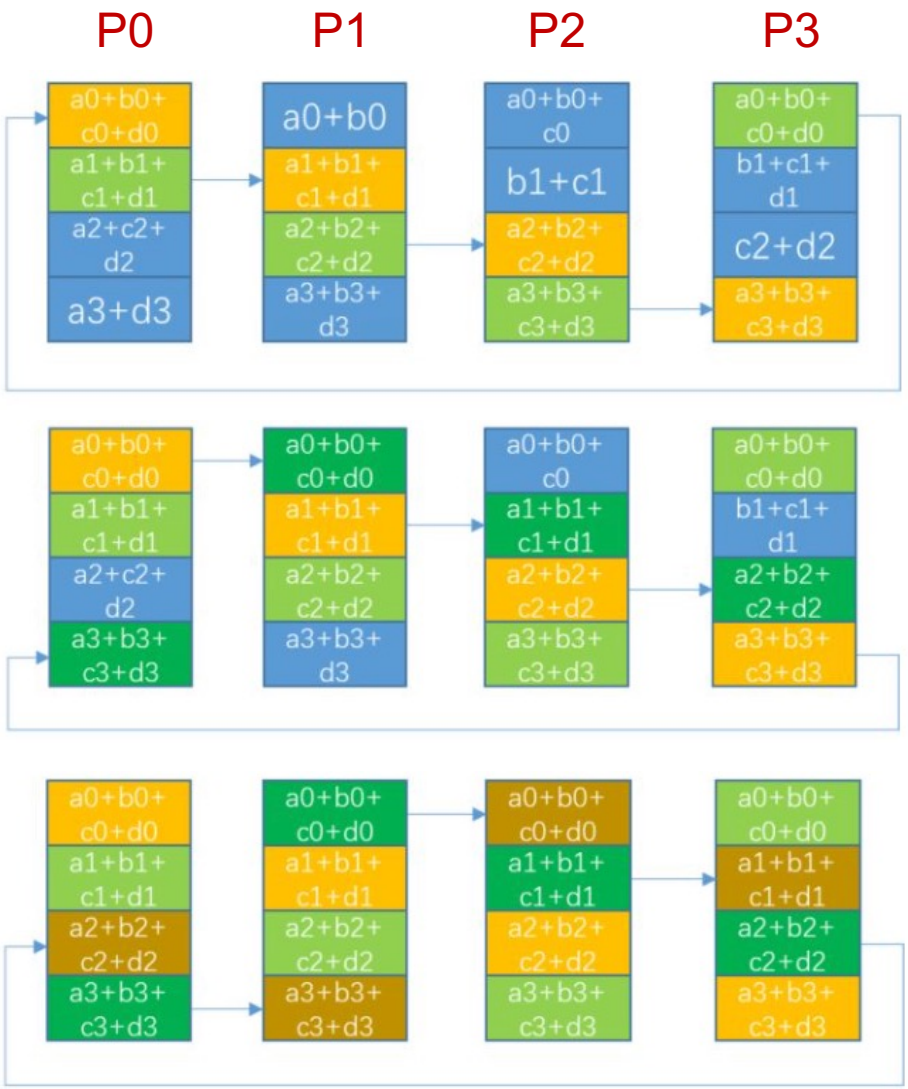
多个数据同时进行Allreduce

Ring 算法



- 首先将每个节点的数据分为P个数据块
- 第一阶段共(P-1)步。在第k步，第 i 个进程会将自己的第 (i - k) % P 对应数据块发送给第 i+1 个进程并累加
- 在(P-1)步结束后，第i个进程的第 (i + 1) % P 对应数据块保存着所有进程的第(i + 1) % P 个数据块的累加和
- 这一阶段也可视为**Reduce-scatter**

Ring 算法



- 第二阶段共(P-1)步。在第k步，第i个进程将自己的第(i+1-k) % P对应数据块发送给第(i+1)个进程
- 在(P-1)步结束后，每个进程的每个数据块都是所有进程的该数据块的累加和
- 这一阶段也可视为**allgather**

控制 MPI 集合通信算法

- 通常自动选择一般都很好，人工干预需要慎重
- Intel MPI 使用 I_MPI_ADJUST_** 环境变量控制（运行前 export）
  - https://software.intel.com/content/www/us/en/develop/documentation/mpi-developer-reference-linux/top/environment-variable-reference/i-mpi-adjust-family-environment-variables.html
  - I_MPI_ADJUST_<opname>="<algid>[:<conditions>][;<algid>:<conditions>[...]]" 表示对于 opname 指定的通信方式，在 conditions 的情况下，采用 algid 对应的算法（可以有多个条件）
  - I_MPI_ADJUST_<opname>_LIST=<algid1>[,<algid2>,...] 表示对于 opname 指定的通信方式，只使用列出的算法
  - 如 I_MPI_ADJUST_ALLTOALL_LIST=3：Alltoall 强制使用成对交换算法
- OpenMPI 的控制更加精细复杂，可查阅文档的 MCA 选项部分

I_MPI_ADJUST_ALLTOALL	MPI_Alltoall	1. Bruck's 2. Isend/Irecv + waitall 3. Pair wise exchange 4. Plum's

MPI程序性能分析

PMPI

- PMPI是MPI标准profiling接口
- 标准MPI函数均可以 **MPI_** 或 **PMPI_** 为前缀调用
  - 例如 **MPI_Send()**和 **PMPI_Send()**
- 实现上，MPI函数对PMPI函数进行封装

```
retType MPI_Func(Type1 arg1, Type2 arg2, ...){
    return PMPI_Func(arg1, arg2, ...);
}
```

- 通过在PMPI函数前后的插桩，实现相关信息的获取

统计函数调用次数

```
int MPI_Func(...){
    count ++;
    return PMPI_Func(...);
}
```

统计每次函数调用参数

```
int MPI_Func(...){
    ret = PMPI_Func(arg1, arg2, ...);
    GatherInfo(ret, arg1, arg2, ...);
    return ret;
}
```

统计函数执行总时间

```
int MPI_Func(...){
    start_time = getTime();
    ret = PMPI_Func(...);
    end_time = getTime();
    tot_time += end_time - start_time;
    return ret;
}
```

mpiP

- mpiP 是一个轻量级的 MPI 程序 profiling 库
  - 仅收集 MPI 函数的统计信息
- 使用方法
  - 根据说明编译 mpiP 支持库
  - 静态链接 MPI：
    - 编译时在 MPI 库前加入 mpiP 库，如 `-lmpiP -lmpi`
  - 动态链接 MPI：
    - 运行时使用 `LD_PRELOAD` 环境变量拦截 MPI 函数调用，例如
    - `srun -n 2 --export=LD_PRELOAD=[path to mpiP]/libmpiP.so [executable]`
- 相关链接
  - https://github.com/LLNL/mpiP
  - https://software.llnl.gov/mpiP/

mpiP 输出示例

- MPI 程序的整体信息

```
@ mpiP
@ Command : /mnt/home/jinyuyang/MY_PROJECT/ScalAna/NPB3.3-MPI/bin/./cg.C.8
@ Version                  : 3.5.0
@ MPIP Build date          : Feb  3 2021, 00:35:20
@ Start time               : 2021 03 02 11:37:27
@ Stop time                : 2021 03 02 11:38:19
@ Timer Used               : PMPI_Wtime
@ MPIP env var             : [null]
@ Collector Rank           : 0
@ Collector PID            : 115235
@ Final Output Dir         : .
@ Report generation        : Single collector task
@ MPI Task Assignment      : 0 gorgon2
@ MPI Task Assignment      : 1 gorgon2
@ MPI Task Assignment      : 2 gorgon2
@ MPI Task Assignment      : 3 gorgon2
@ MPI Task Assignment      : 4 gorgon2
@ MPI Task Assignment      : 5 gorgon2
@ MPI Task Assignment      : 6 gorgon2
@ MPI Task Assignment      : 7 gorgon2

---------------------------------------------------------------------------
@--- MPI Time (seconds) ---------------------------------------------------
---------------------------------------------------------------------------
Task    AppTime    MPITime     MPI%
   0         52       2.35     4.51
   1         52       2.47     4.74
   2         52       3.16     6.08
   3         52       1.42     2.73
   4         52       2.67     5.12
   5         52       1.59     3.05
   6         52       2.66     5.11
   7         52       2.45     4.71
   *        416       18.7     4.51
```

运行环境信息

各个进程通信时间统计信息

mpiP 输出示例

- MPI 调用索引表（以调用路径为粒度）：

索引号 函数栈等级 文件名　　行号　所在函数　　MPI 函数名

```
35 ---------------------------------------------------------------------------
36 @--- Callsites: 456 ---------------------------------------------------------
37 ---------------------------------------------------------------------------
38  ID Lev File/Address        Line Parent_Funct             MPI_Call
39   1   0 cg.f                1340 conj_grad_               Wait
40   2   0 cg.f                 414 cg                       Irecv
41   3   0 cg.f                1237 conj_grad_               Send
42   4   0 cg.f                1284 conj_grad_               Wait
43   5   0 cg.f                1159 conj_grad_               Wait
44   6   0 cg.f                1209 conj_grad_               Wait
45   7   0 cg.f                1237 conj_grad_               Send
46   8   0 cg.f                1284 conj_grad_               Wait
47   9   0 cg.f                1159 conj_grad_               Wait
```

mpiP 输出示例

- MPI 函数调用的详细信息

```
495 ---------------------------------------------------------------------------
496 @--- Aggregate Time (top twenty, descending, milliseconds) ----------------
497 ---------------------------------------------------------------------------
498 Call                 Site       Time    App%    MPI%      Count    COV
499 Send                  103   2.46e+03    0.59   13.10       3750   0.00
500 Send                  158   1.85e+03    0.44    9.86       3750   0.00
501 Send                   46   1.75e+03    0.42    9.35       3750   0.00
502 Send                  388   1.64e+03    0.40    8.77       3750   0.00
503 Send                  274    1.6e+03    0.39    8.55       3750   0.00
504 Send                  443   1.56e+03    0.38    8.34       3750   0.00
505 Send                  215        805    0.19    4.29       3750   0.00
506 Send                  329        718    0.17    3.83       3750   0.00
507 Wait                  351        680    0.16    3.63       3750   0.00
519 ---------------------------------------------------------------------------
520 @--- Aggregate Sent Message Size (top twenty, descending, bytes) ----------
521 ---------------------------------------------------------------------------
522 Call                 Site      Count      Total       Avrg  Sent%
523 Send                  329       3750   1.12e+09      3e+05   7.91
524 Send                  103       3750   1.12e+09      3e+05   7.91
525 Send                  274       3750   1.12e+09      3e+05   7.91
526 Send                   46       3750   1.12e+09      3e+05   7.91
527 Send                  215       3750   1.12e+09      3e+05   7.91
528 Send                  158       3750   1.12e+09      3e+05   7.91
529 Send                  388       3750   1.12e+09      3e+05   7.91
530 Send                  443       3750   1.12e+09      3e+05   7.91
531 Send                   65       1875   5.62e+08      3e+05   3.95
```

时间等信息

发送量等信息

MPI 程序计时

- MPI_Wtime 用于记录进程的运行时间
- 返回值是从过去的任意时间开始，以秒为单位的时间。
- MPI_Wtick 返回 MPI_Wtime 的精度
- 当 MPI_WTIME_IS_GLOBAL 设为 true 时，各个进程的时钟将会同步
- 使用方法

```
double start, end;
...
start = MPI_Wtime();
/** Code to be timed */
...
end = MPI_Wtime();
printf("Proc %d : Elapsed time = %e seconds\n",
                                    my_rank, end - start );
```

OSU 基准测试程序

- OSU测量和评估 MPI 点对点和集合通信操作的性能
  - 用于比较不同的 MPI 实现和基础网络环境的性能
- 使用方法
  - 通过点对点通信测试网络延时/带宽

    ```
    mpirun -np 2 ./mpi/pt2pt/osu_latency 或 osu_bw
    ```
  - 将发送和接收进程分别绑定在不同的节点上，测试集群不同跳数的延迟/带宽

    ```
    mpirun -np 2 --host N1:1,N2:1 ./mpi/pt2pt/osu_latency
    ```
  - 测试集合通信性能

    ```
    mpirun -np 32 ./mpi/collective/osu_alltoall
    ```
- 实验集群上的使用
  - 加载：`spack load osu-micro-benchmarks`
  - 点对点性能：`srun -N 2 -n 2 osu_bw / osu_latency`
  - 集合通信（可用 GPU）：`srun -N 4 -n 16 osu_allreduce [-d cuda]`
- 相关链接
  - http://mvapich.cse.ohio-state.edu/benchmarks/

OSU 基准测试程序输出

延迟

```
# OSU MPI Latency Test v5.4.1
# Size          Latency (us)
0                       0.26
1                       0.30
2                       0.30
4                       0.30
8                       0.29
16                      0.29
32                      0.31
64                      0.31
128                     0.39
256                     0.44
512                     0.55
1024                    0.66
2048                    0.81
4096                    1.30
8192                    1.49
16384                   2.18
32768                   3.15
65536                   5.21
131072                  9.95
262144                 22.06
524288                 45.76
1048576                94.46
2097152               196.49
4194304               391.09
```

带宽

```
# OSU MPI Bandwidth Test v5.4.1
# Size      Bandwidth (MB/s)
1                       5.10
2                      10.25
4                      19.90
8                      40.94
16                     80.05
32                    158.45
64                    251.38
128                   354.71
256                   577.66
512                  1020.16
1024                 2308.43
2048                 3703.74
4096                 2589.20
8192                 4548.39
16384                5810.28
32768                7743.17
65536                8555.32
131072               9052.95
262144              10496.45
524288              10133.83
1048576             10243.76
2097152             10192.72
4194304             10811.58
```

OSU 基准测试程序输出

集合通信 Alltoall 性能

```
# OSU MPI All-to-All Personalized Exchange Latency Test v5.4.1
# Size       Avg Latency(us)
1                       7.67
2                       7.57
4                       7.22
8                       7.09
16                      7.28
32                      7.27
64                      7.63
128                     8.71
256                     8.78
512                    10.58
1024                   17.49
2048                   23.47
4096                   69.72
8192                   85.52
16384                 119.00
32768                 178.54
65536                 296.67
131072                939.62
262144               2691.06
524288               5222.80
1048576             10225.12
```

总结

- 点对点通信和缓冲区
- 点对点通信实现协议
- 通信死锁
- 进程映射
- 集合通信算法实现
- MPI 程序性能分析

小作业-2

- Allreduce 通信是深度学习模型并行训练中重要通信函数
- 实现 Ring Allreduce，与默认MPI_Allreduce 以及 MPI_Reduce + MPI_Bcast对比性能
- 目的：
  - 熟悉 MPI 点对点通信与集合通信的调用方法
  - 熟悉 Ring Allreduce 的实现方法
- 详见网络学堂发布