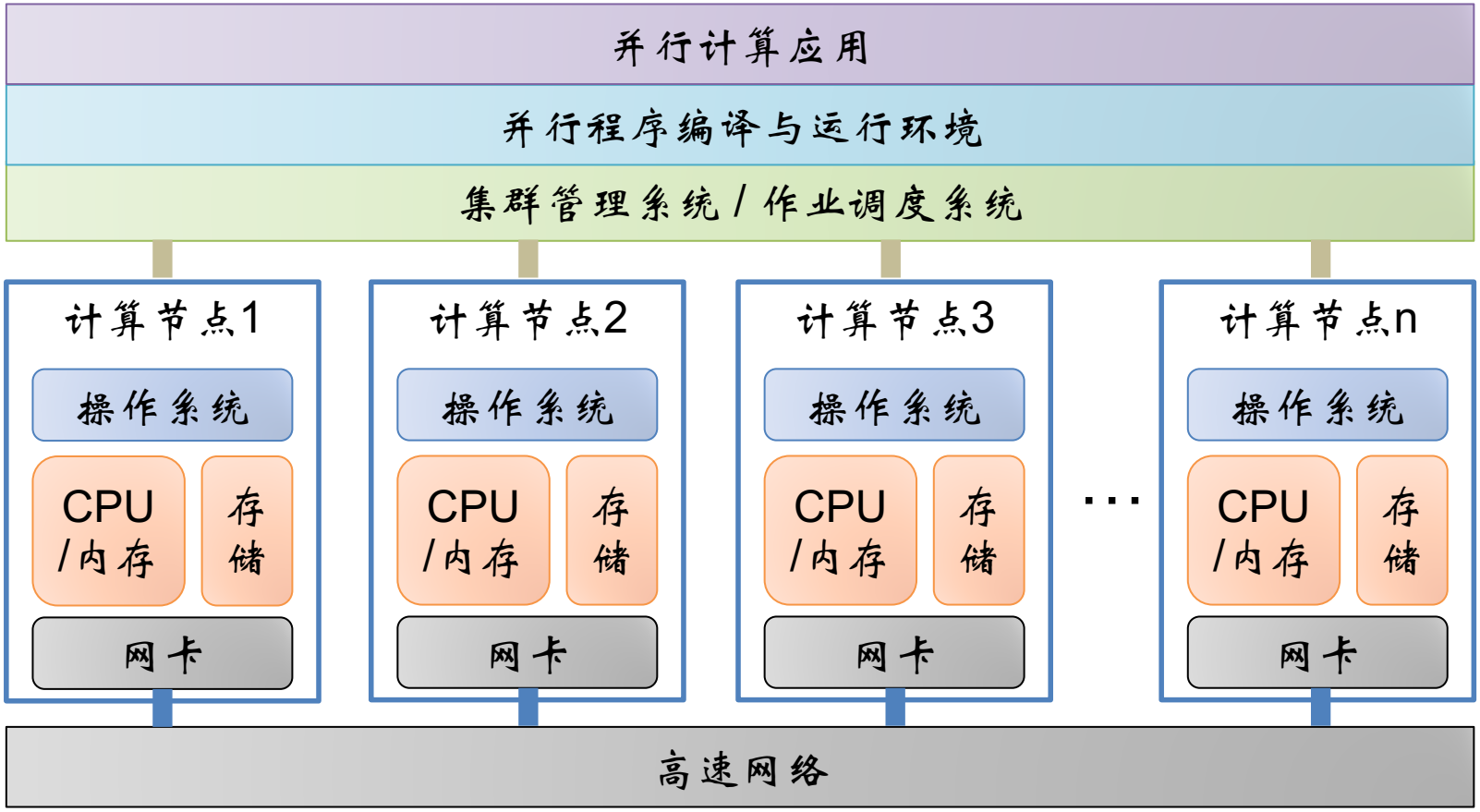
MPI基础

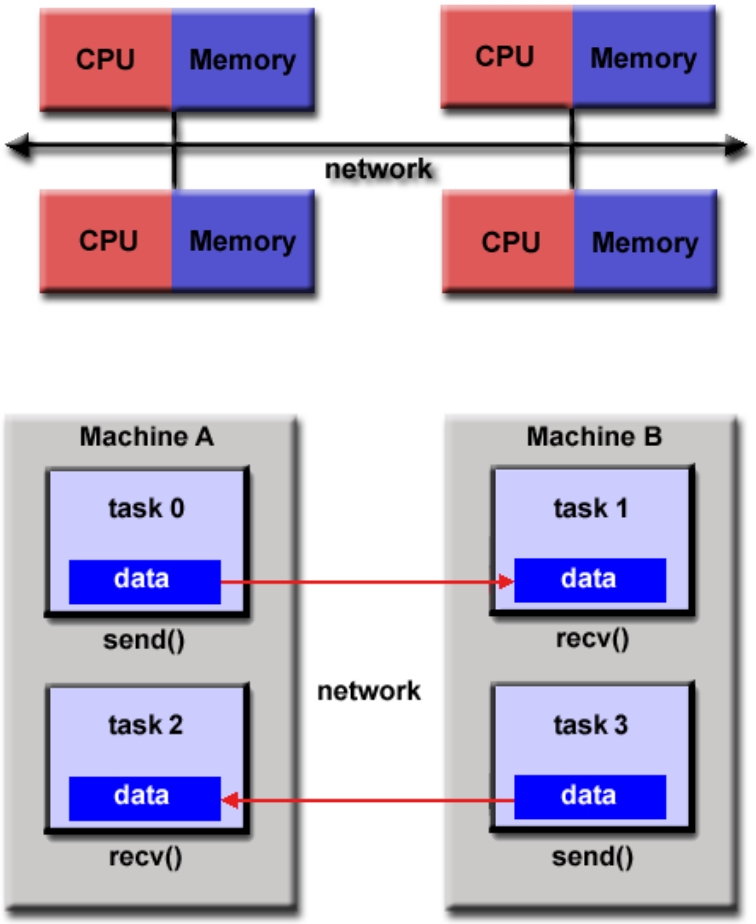
汤善江 副教授
天津大学智能与计算学部
tashj@tju.edu.cn
http://cic.tju.edu.cn/faculty/tangshanjiang/

集群逻辑体系结构


并行计算应用
并行程序编译与运行环境
集群管理系统 / 作业调度系统
计算节点1
操作系统
CPU /内存
存储
网卡
计算节点2
操作系统
CPU /内存
存储
网卡
计算节点3
操作系统
CPU /内存
存储
网卡
...
计算节点n
操作系统
CPU /内存
存储
网卡
高速网络

集群分类（按用途）

- 高性能计算
  - 科学计算，并行计算
  - 优先考虑计算性能
- 大数据分析
  - 分布式并行数据处理
  - 优先考虑IO与存储性能优化
- 高可用服务
  - 高可靠在线服务
  - 最大程度减少对外服务中断



Cluster1350

管理网络示例

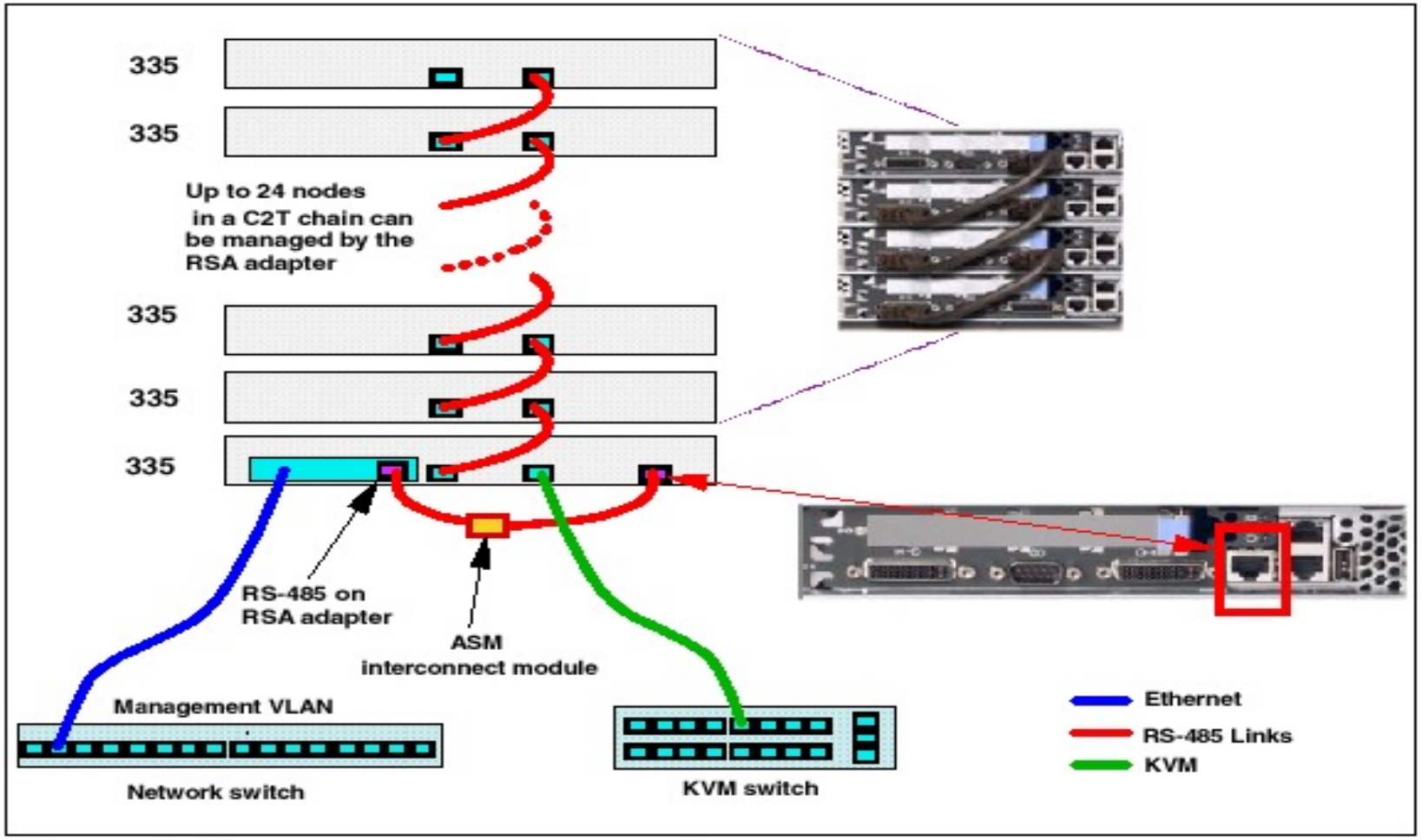


Figure 2-4 Management processor network

网络配置示例

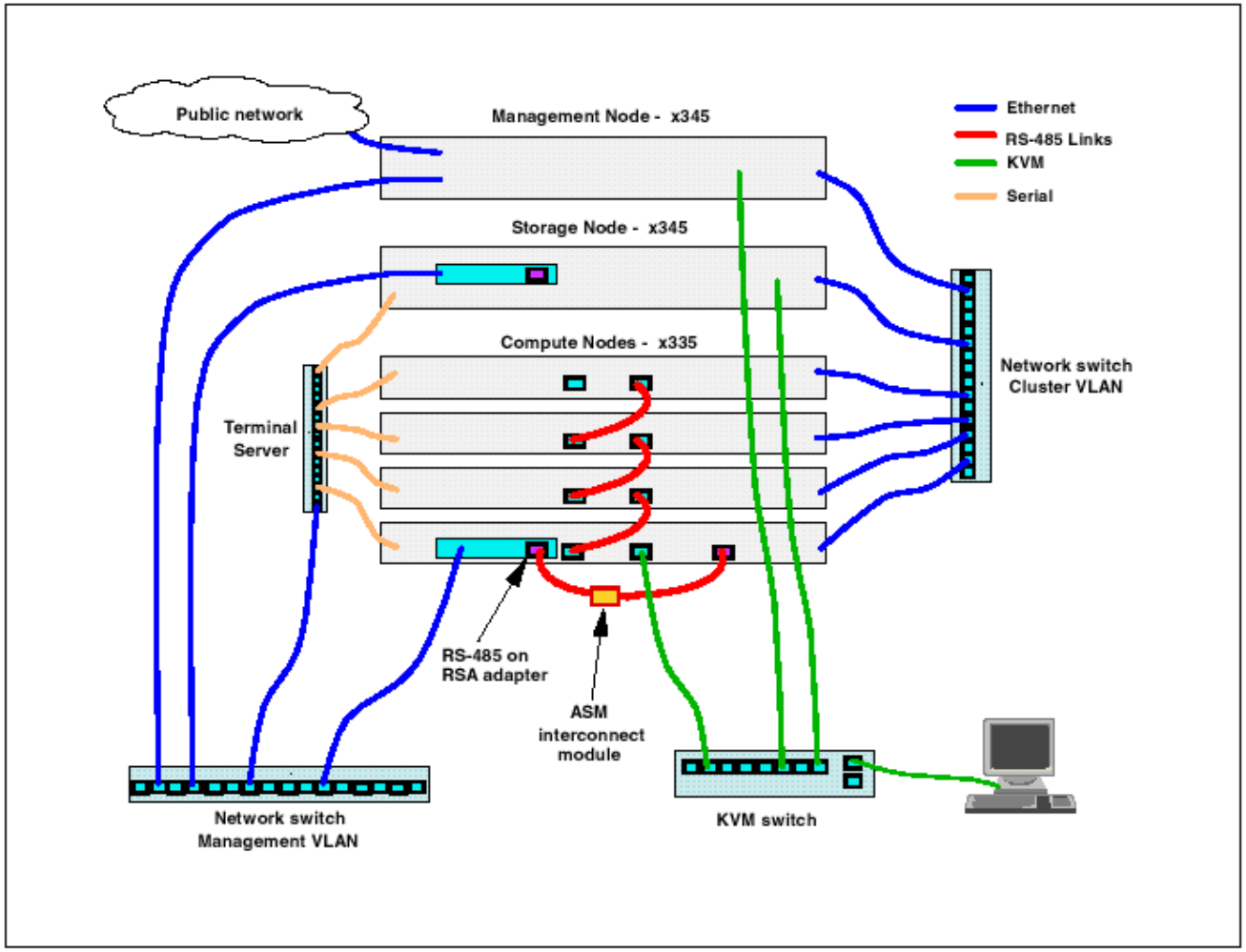


Figure 5-1 Lab cluster configuration

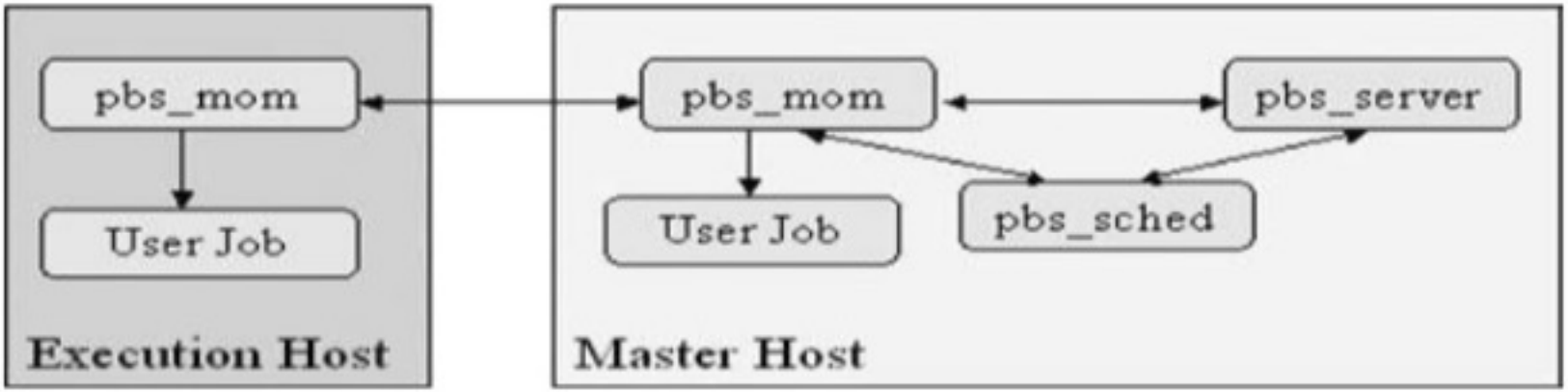
并行计算作业管理

- 在大规模集群或者超级计算机平台上，一般不能随意地直接运行用户的并行计算程序，而必须通过其上提供的作业管理系统来提交计算任务。
- 集群作业管理系统的功能
  - 统一管理和调度集群的软硬件资源
  - 保证用户作业公平合理地共享集群资源
  - 提高系统利用率和吞吐率。
- 常用的作业管理系统
  - PBS
  - Slurm
  - LSF

PBS

- PBS(Portable Batch System)最初由NASA的Ames研究中心开发，提供能满足异构计算网络需要的软件包，用于灵活的批处理，满足高性能计算的需要。
- PBS的主要特点有：
  - 代码开放，免费获取；
  - 支持批处理、交互式作业和串行、多种并行作业，如MPI、PVM、HPF、MPL；
  - PBS是功能最为齐全，历史最悠久，支持最广泛的本地集群调度器之一。
- PBS的目前包括openPBS, PBS Pro和Torque三个主要分支.
  - OpenPBS是最早的PBS系统，目前已经没有太多后续开发
  - PBS pro是PBS的商业版本，功能最为丰富
  - Torque是Clustering公司接管了OpenPBS，并给与后续支持的一个开源版本

PBS 逻辑架构



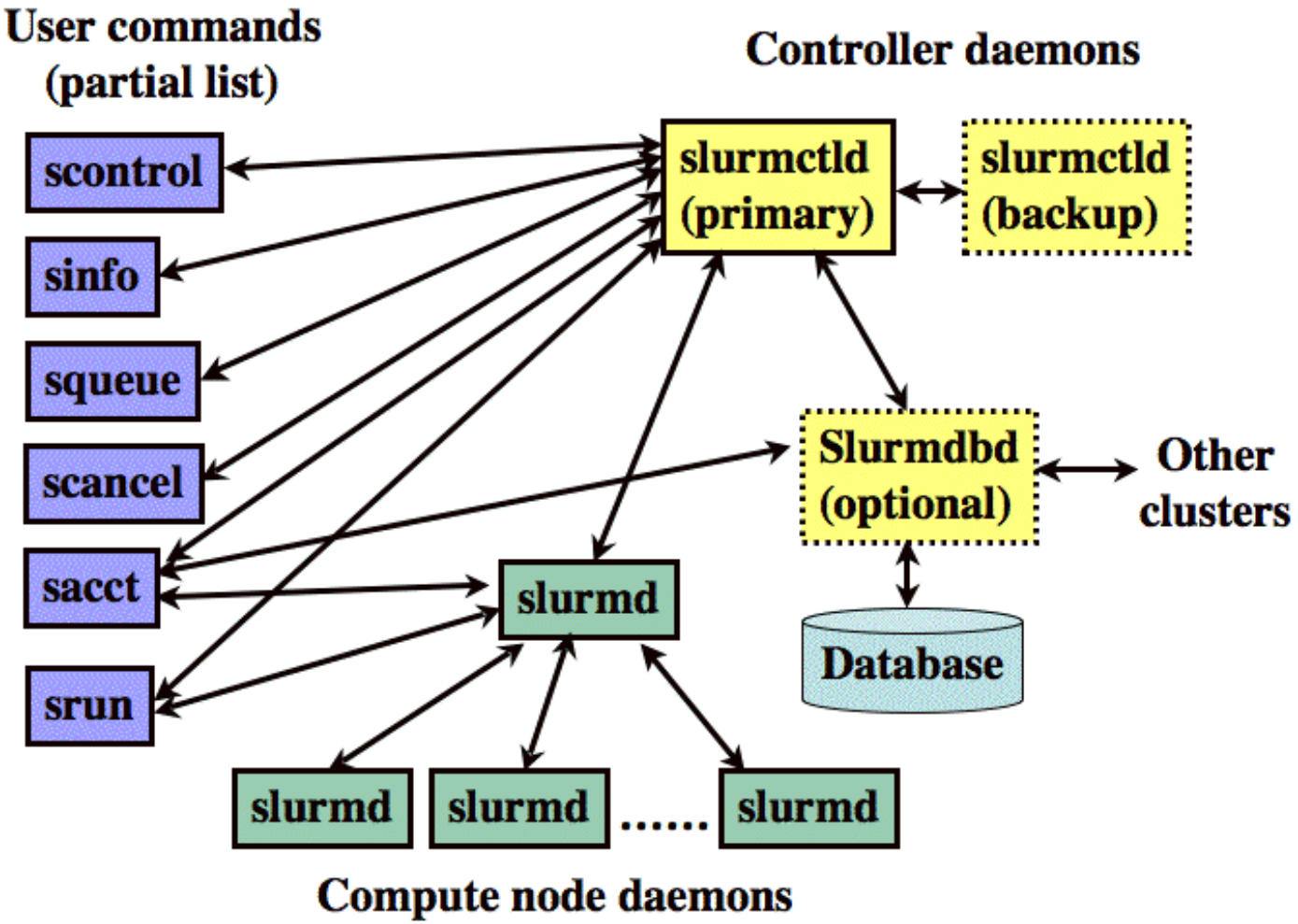
- pbs_mom：后台监控进程
- pbs_server：调度服务器
- pbs_sched：调度策略

Slurm

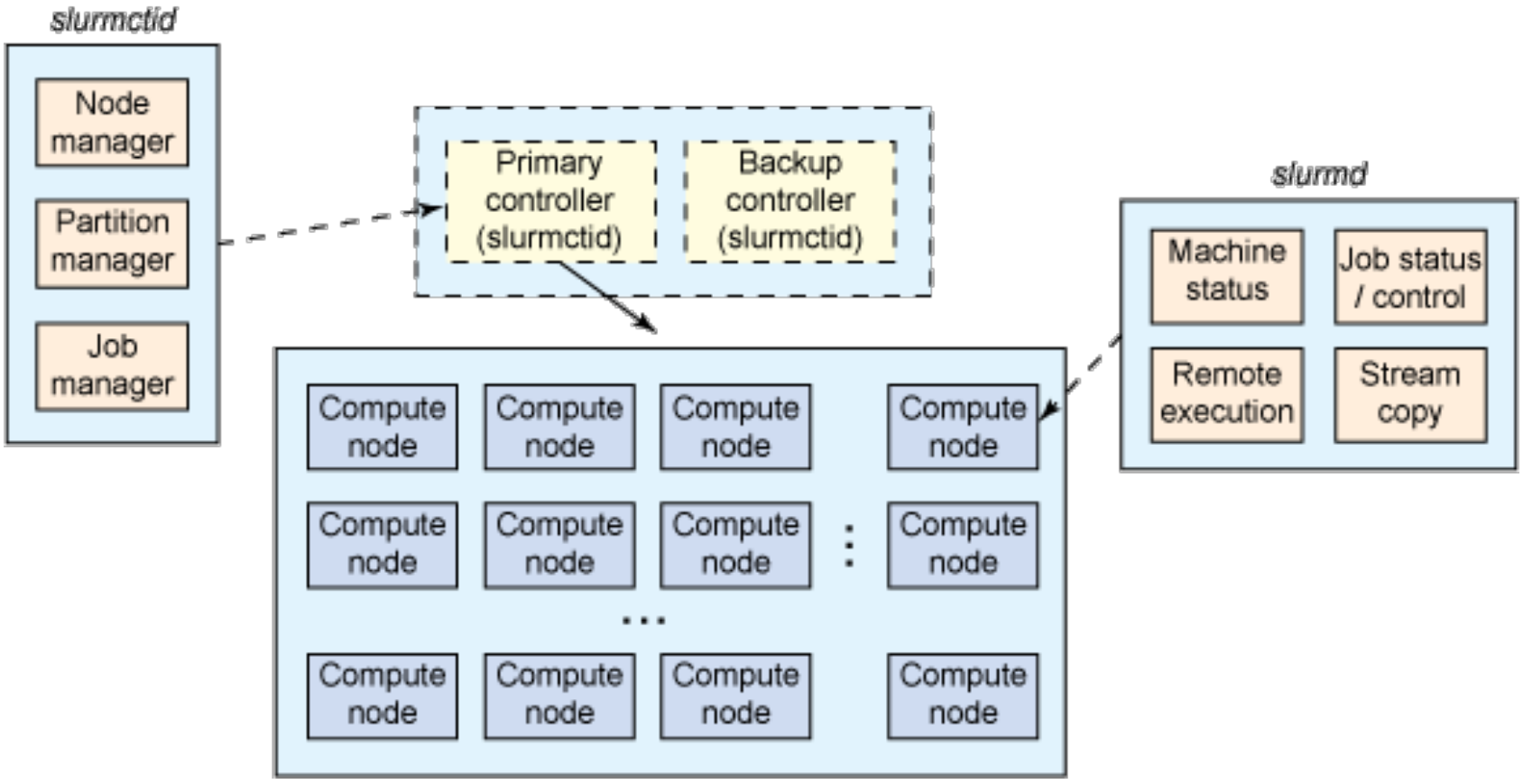
- SLURM （Simple Linux Utility for Resource Management）是一种可用于大型计算节点集群的高度可伸缩和容错的集群管理器和作业调度系统，主要功能包括：
  - 在一段时间内为用户**分配资源**（计算机节点）的独占和/或非独占访问权限，以便他们可以执行工作。
  - 提供了一个框架，用于在一组分配的节点上**启动，执行和监视**工作（通常是并行作业，例如MPI）。
  - 通过管理待处理作业队列来**仲裁资源争用**。
- Slurm是TOP500超级计算机中约60%的工作负载管理器，其中包括天河二号。
- Slurm使用基于希尔伯特曲线调度或胖树网络拓扑的最佳拟合算法来优化并行计算机上任务分配的局部性。

官网：https://slurm.schedmd.com/documentation.html

Slurm逻辑架构

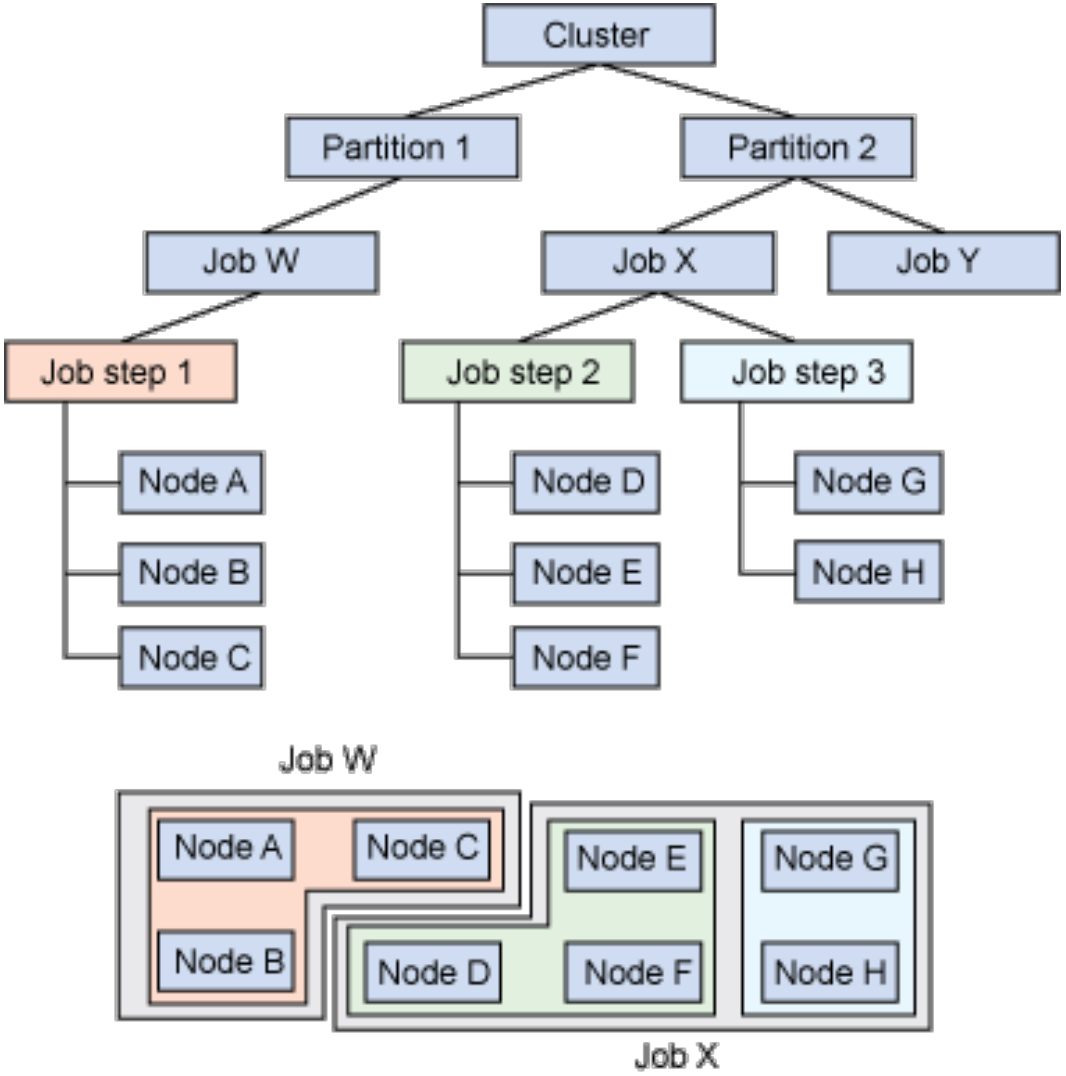

User commands
(partial list)
Controller daemons
scontrol
sinfo
squeue
scancel
sacct
srun
slurmctld
(primary)
slurmctld
(backup)
Slurmdbd
(optional)
Other
clusters
Database
slurmd
slurmd
slurmd
.....
slurmd
Compute node daemons

Slurm部署


slurmctid
Node manager
Partition manager
Job manager
Primary controller (slurmctid)
Backup controller (slurmctid)
slurmd
Machine status
Job status / control
Remote execution
Stream copy
Compute node

Slurm 资源分区

- 不同的节点的特性和硬件属性不同，设置分区可以帮助用户更好确定节点的特点，进而选择最适合自己的节点进行运算。
- 如果集群中部分机器是私有的，那么设置分区可以使得只有部分用户能在这个分区提交作业。
- 分区(Partition)可看做一系列节点的集合。



PBS 与 Slurm 的功能及命令对照

功能	PBS	SLURM
任务名称	#PBS -N name	#SBATCH -J name
指定队列/分区	#PBS -q cpu	#SBATCH -p cpu
指定 QoS	#PBS --qos=debug，需调度器支持	#SBATCH --qos=debug
最长运行时间	#PBS -l walltime=5:00	#SBATCH -t 5:00
指定节点数量	#PBS -l nodes=1	#SBATCH -N 1
指定 CPU 核心	#PBS -l ppn=4	#SBATCH --cpus-per-task=4
指定 GPU 卡	不支持	#SBATCH --gres=gpu:1
作业数组	#PBS -t 0-2	#SBATCH -a 0-2
输出文件	#PBS -o test.out	#SBATCH -o test.out
提交任务脚本	qsub run.pbs	sbatch run.slurm
查看任务状态	qstat	squeue
取消任务	qdel 1234	scancel 1234
交互式任务	qsub -I，自动切换	salloc，手动切换
指定特定节点	qsub -l nodes=comput1	#SBATCH --nodelist=comput1

Outline

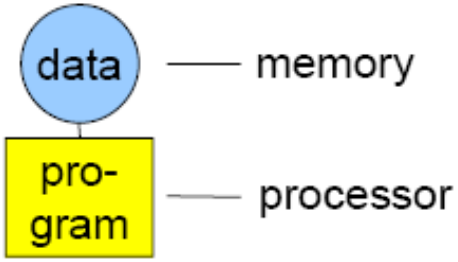
- MPI概述
- 点到点通信
- 组通信
- 阻塞通信模式

Outline

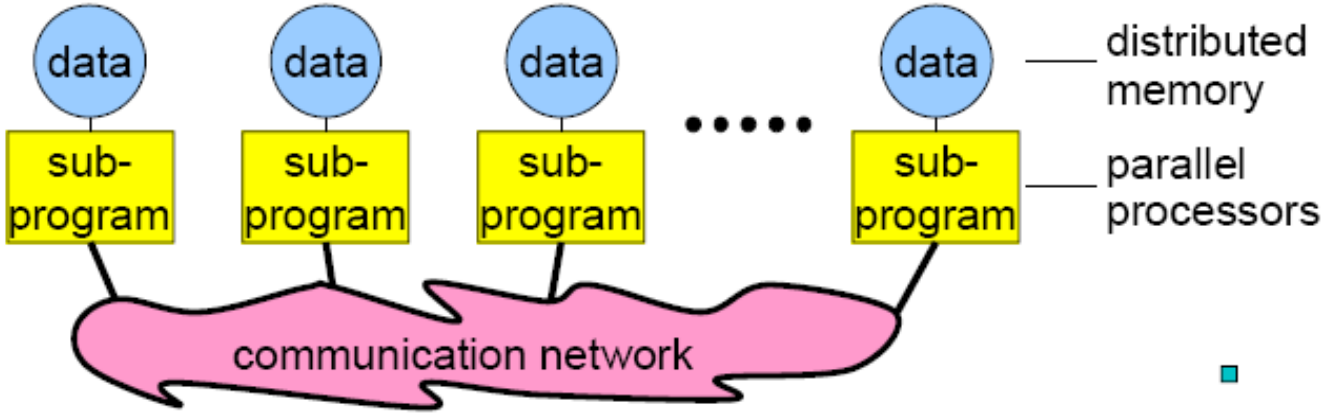
- **MPI概述**
- 点到点通信
- 组通信
- 阻塞通信模式

MPI概述

- 串行程序



- MPI并行程序



MPI (Message passing interface)

- MPI是一种标准或规范的代表，而不特指某一个对它的具体实现。MPI同时也是一种消息传递编程模型，并成为这种编程模型的代表和事实上的标准。
  - 迄今为止所有的并行计算机制造商都提供对MPI的支持，可以在网上免费得到MPI在不同并行计算机上的实现。
- MPI的实现是一个库，而不是一门语言。
  - 可以把FORTRAN+MPI或C+MPI 看作是一种在原来串行语言基础之上扩展后得到的并行语言。

MPI程序示例: Hello World!

Fortran

```
PROGRAM hello
  INCLUDE ‘mpif.h‘
  INTEGER err
  CALL MPI_INIT(err)
  PRINT *, “hello world!”
  CALL MPI_FINALIZE(err)

END
```

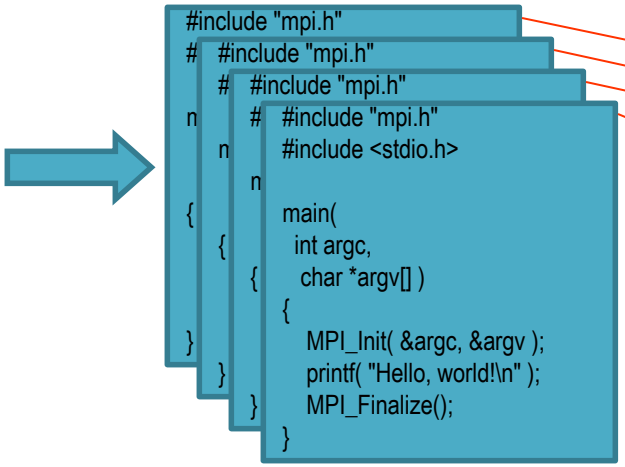
C

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

  err = MPI_Init(&argc, &argv);
  printf( “Hello world!\n” );
  err = MPI_Finalize();
}
```

MPI程序的执行

- SPMD: Single Program Multiple Data(MIMD)

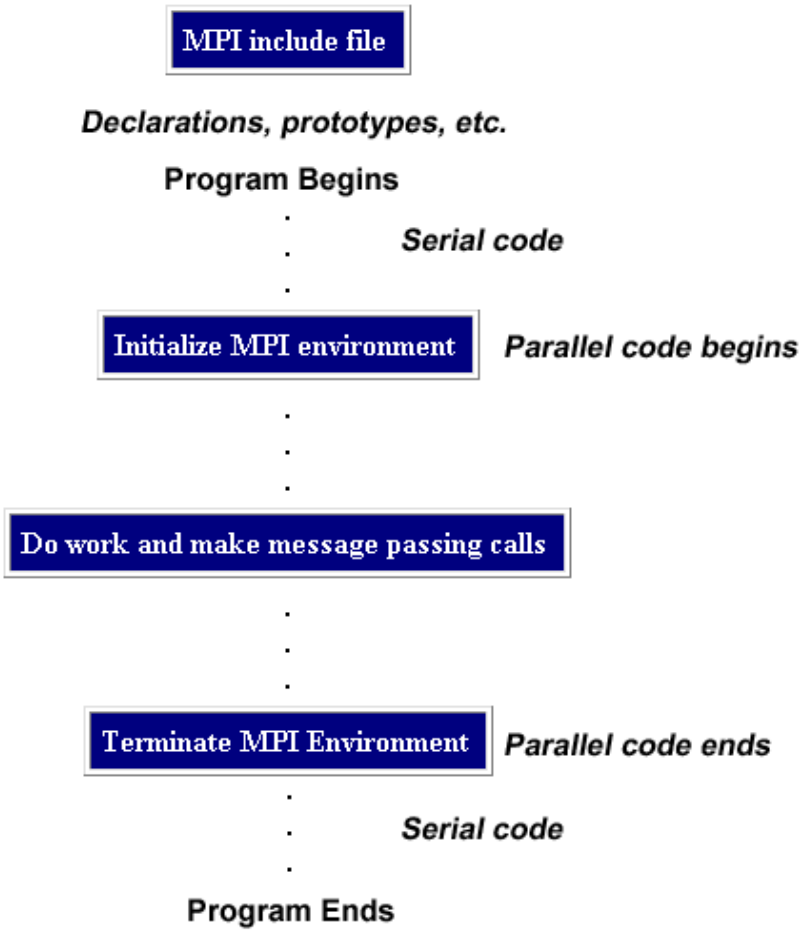
```
#include "mpi.h"
#include <stdio.h>

main(
  int argc,
  char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
}
```



```
#include "mpi.h"
    main(
      int argc,
      char *argv[] )
    {
        MPI_Init( &argc, &argv );
        printf( "Hello, world!\n" );
        MPI_Finalize();
    }
```

Hello World!
Hello World!
Hello World!
Hello World!

MPI程序结构


MPI include file
Declarations, prototypes, etc.
Program Begins
Serial code
Initialize MPI environment
Parallel code begins
Do work and make message passing calls
Terminate MPI Environment
Parallel code ends
Serial code
Program Ends

MPI 的六个基本接口

- 开始与结束
  - MPI_INIT
  - MPI_FINALIZE
- 进程身份标识
  - MPI_COMM_SIZE
  - MPI_COMM_RANK
- 发送与接收消息
  - MPI_SEND
  - MPI_RECV

MPI 程序的开始与结束

- MPI代码开始之前必须进行如下调用：

```
MPI_Init(&argc, &argv);
```

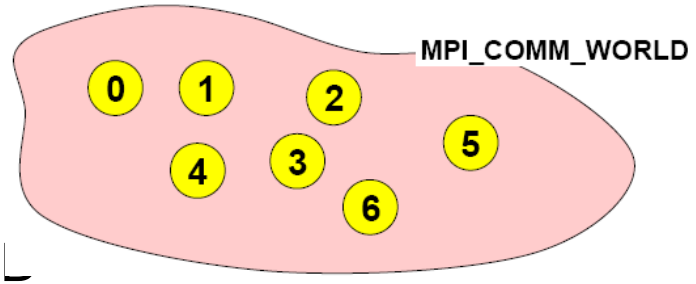
  - MPI系统将通过argc,argv得到命令行参数

- MPI代码的最后一行必须是：

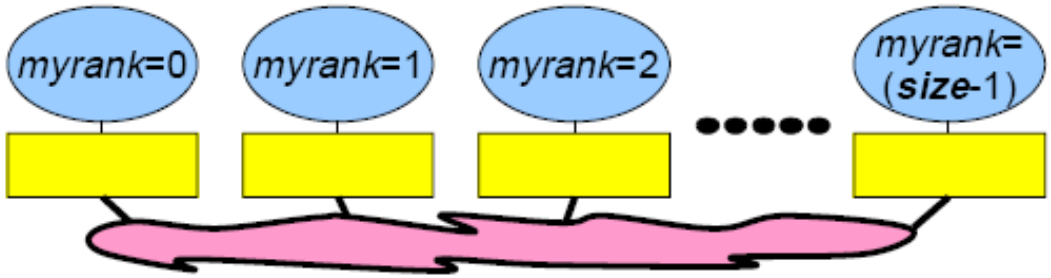
```
MPI_Finalize();
```

  - 如果没有此行，MPI程序将不会终止。

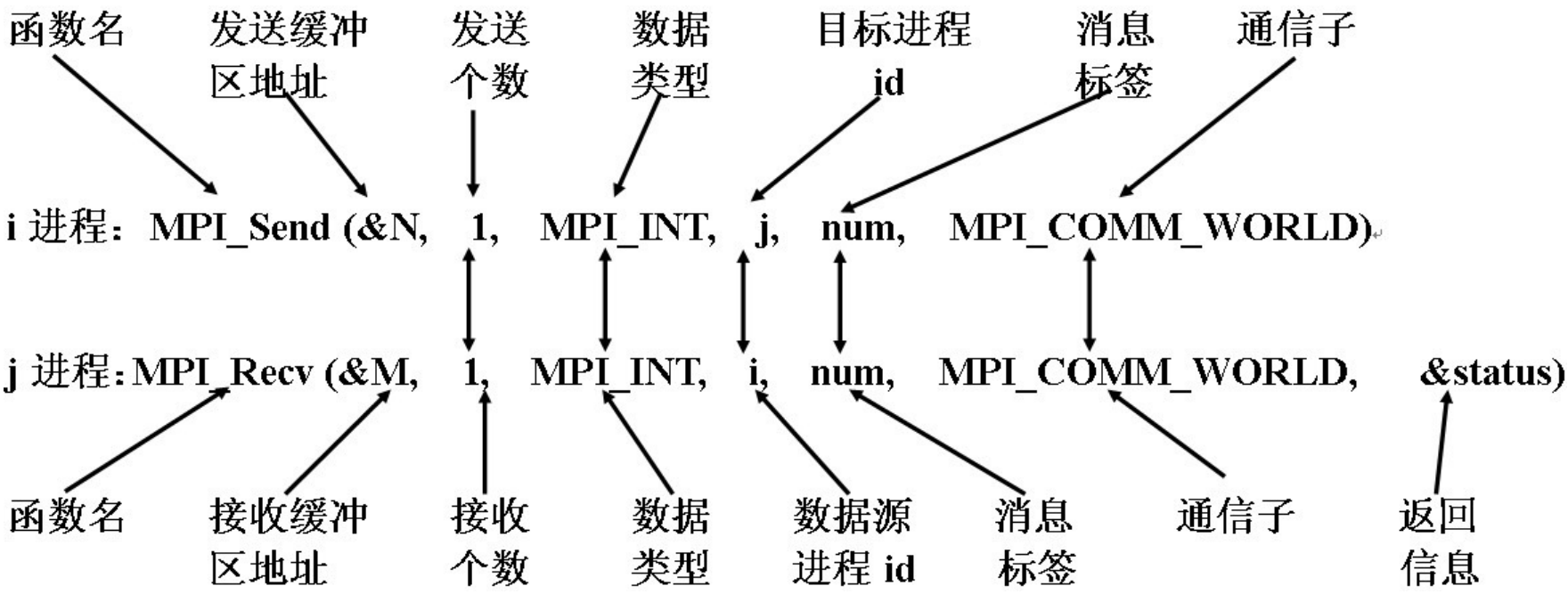
MPI进程身份标识



- 通信域
  - 缺省的通信域为MPI_COMM_WORLD
- MPI_Comm_size(MPI_COMM_WORLD, &size)
  - 获得缺省通信域内所有进程数目，赋值给size
- MPI_Comm_rank(MPI_COMM_WORLD, &myrank)
  - 获得进程在缺省通信域的编号，赋值给myrank



发送和接收消息


函数名
发送缓冲区地址
发送个数
数据类型
目标进程id
消息标签
通信子
i 进程：MPI_Send (&N, 1, MPI_INT, j, num, MPI_COMM_WORLD)
j 进程：MPI_Recv (&M, 1, MPI_INT, i, num, MPI_COMM_WORLD, &status)
函数名
接收缓冲区地址
接收个数
数据类型
数据源进程 id
消息标签
通信子
返回信息

一个计算∑foo(i)的MPI SPMD消息传递程序

```
#include "mpi.h"
int foo(i)
int i;
{...}
main(argc, argv)
int argc;
char* argv[]
{
int i, tmp, sum=0, group_size, my_rank, N;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &group_size);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank==0) {
   printf("Enter N:");
   scanf("%d",&N);
   for (i=1;i<group_size;i++)
      MPI_Send(&N, 1, MPI_INT, i, i, MPI_COMM_WORLD);
   for (i=my_rank;i<N;i=i+group_size) sum=sum+tmp;
   for (i=1;i<group_size;i++) {
      MPI_Recv(&tmp,1,MPI_INT,i,i,MPI_COMM_WORLD,&status);
     sum=sum+tmp;
     }
   printf("\n The result = %d", sum);
}
else {
   MPI_Recv(&N,1,MPI_INT,0,i,MPI_COMM_WORLD,&status);
   for (i-my_rank;i<N;i=i+group_size) sum=sum+foo(i);
   MPI_Send(&sum,1,MPI_INT,0,i,MPI_COMM_WORLD);
}
MPI_Finalize();
}
```

初始化MPI环境

获得总进程数

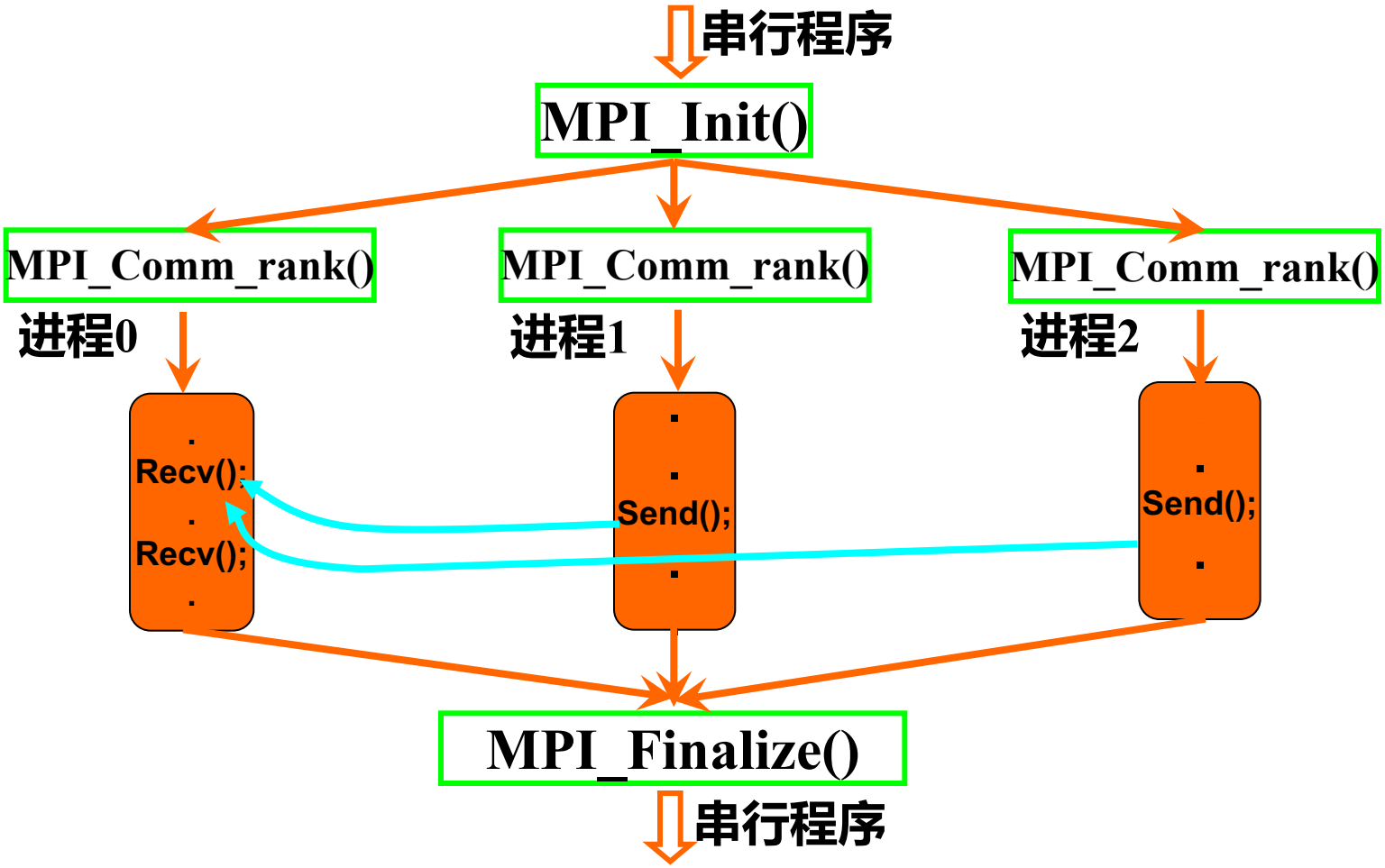
得到每个进程在组中的编号

发送消息

接收消息

终止MPI环境

消息传递的过程

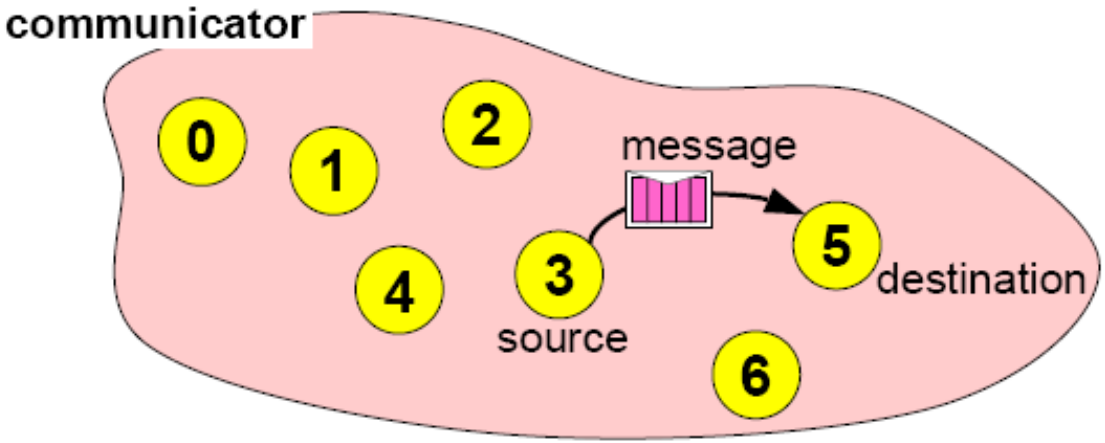

串行程序
MPI_Init()
MPI_Comm_rank()
MPI_Comm_rank()
MPI_Comm_rank()
进程0
进程1
进程2
Recv();
Recv();
Send();
Send();
MPI_Finalize()
串行程序

Outline

- MPI概述
- **点到点通信**
- 组通信
- 阻塞通信模式

点到点通信

- 对于某一消息
  - 唯一发送进程
  - 唯一接收进程



MPI_Send

MPI_Send(buffer, count, datatype, destination, tag, communicator)

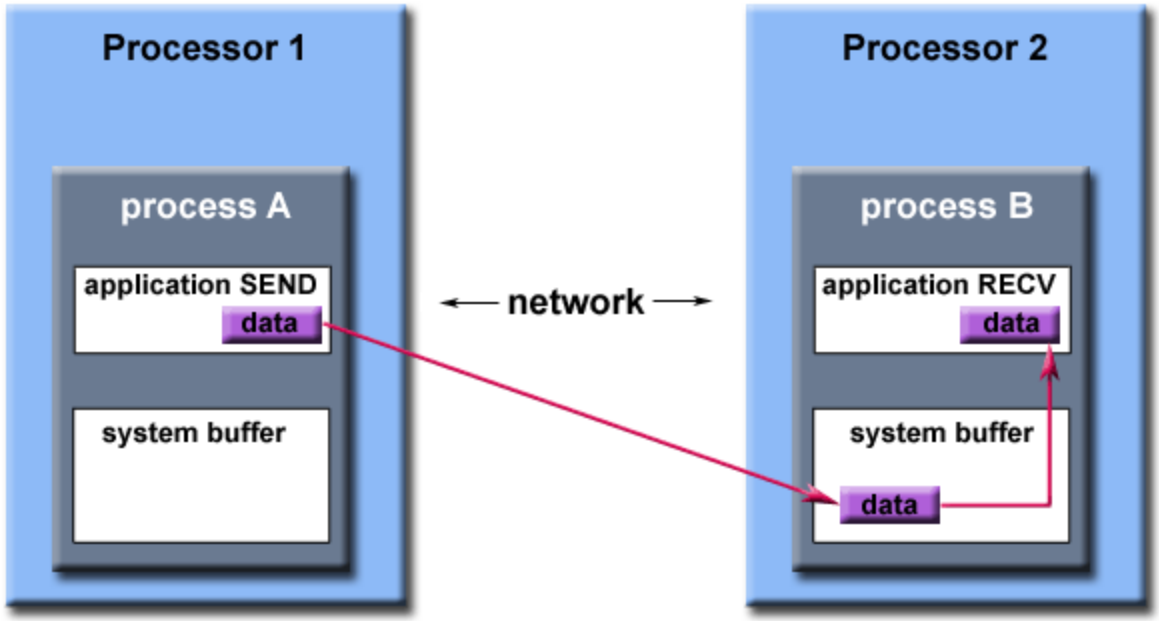
- MPI_Send(&N, 1, MPI_INT, i, i , MPI_COMM_WORLD);
- 第一个参数指明消息缓存的起始地址，即存放要发送的数据信息。
- 第二个参数指明消息中给定的数据类型有多少项，数据类型由第三个参数给定。
- 数据类型要么是基本数据类型，要么是导出数据类型，后者由用户生成指定一个可能是由混合数据类型组成的非连续数据项。
- 第四个参数是目的进程的标识符(进程编号)。
- 第五个是消息标签。
- 第六个参数标识进程组和上下文，即通信域。通常，消息只在同组的进程间传送。但是MPI允许通过intercommunicators在组间通信。

MPI_Receive

MPI_Recv(address, count, datatype,source, tag, communicator, status)

- MPI_Recv(&tmp, 1, MPI_INT, i, i, PI_COMM_WORLD,&Status)
- 第一个参数指明接收消息缓冲的起始地址，即存放接收消息的内存地址。
- 第二个参数指明给定数据类型可以被接收的最大项数。
- 第三个参数指明接收的数据类型。
- 第四个参数是源进程标识符 (编号)。
- 第五个是消息标签。
- 第六个参数标识一个通信域。
- 第七个参数是一个指针，指向一个结构：MPI_Status Status
  - 存放有关接收消息的各种信息。(Status.MPI_SOURCE, Status.MPI_TAG)
  - MPI_Get_count(&Status, MPI_INT, &C)读出实际接收到的数据项数。

消息的接收（系统缓存）



Path of a message buffered at the receiving process

标签的使用

为什么要使用消息标签(Tag)?

这段代码需要传送A的前32个字节进入X，传送B的前16个字节进入Y。但是，如果消息B尽管后发送但先到达进程Q，就会被第一个recv()接收在X中。

使用标签可以避免这个错误。

未使用标签

Process P:	Process Q:
send(A,32,Q)	recv(X, 32, P)
send(B,16,Q)	recv(Y, 16, P)

使用了标签

Process P:	Process Q:
send(A,32,Q,tag1)	recv (X, 32, P, tag1)
send(B,16,Q,tag2)	recv (Y, 16, P, tag2)

标签的使用

```
Process P:
send (request1,32, Q)

Process R:
send (request2, 32, Q)

Process Q:
while (true) {
      recv (received_request, 32, Any_Process);
      process received_request;
      }
```

使用标签的另一个原因是可以简化对下列情形的处理。
假定有两个客户进程P和R，每个发送一个服务请求消息给服务进程Q。

```
Process P:
send(request1, 32, Q, tag1)

Process R:
send(request2, 32, Q, tag2)

Process Q:
while (true){
      recv(received_request, 32, Any_Process, Any_Tag, Status);
      if (Status.Tag==tag1) process received_request in one way;
      if (Status.Tag==tag2) process received_request in another way;
      }
```

Outline

- MPI概述
- 点到点通信
- **组通信**
- 阻塞通信模式

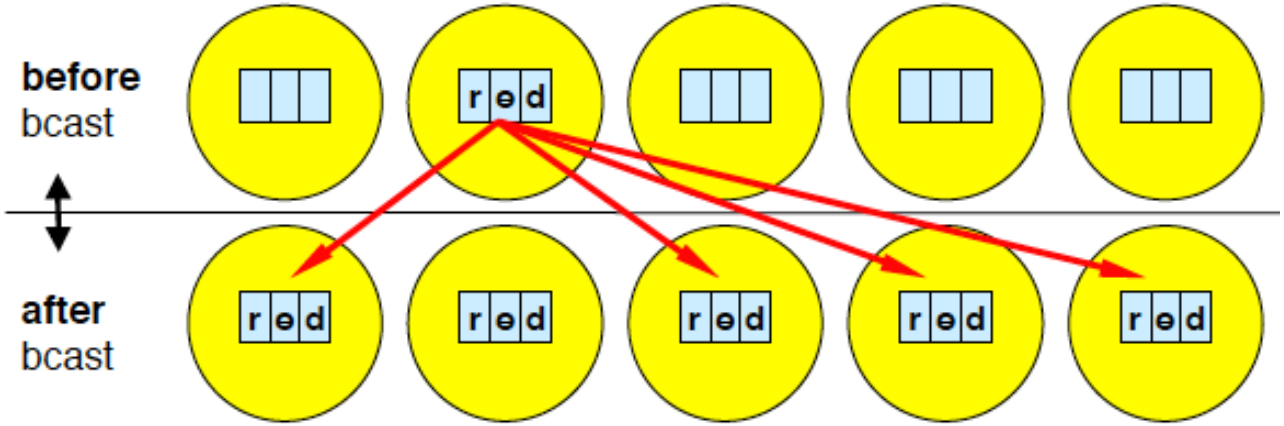
组通信

- 一到多 (Broadcast, Scatter)
- 多到一 (Reduce, Gather)
- 多到多 (Allreduce, Allgather)
- 同步 (Barrier)

广播（Broadcast）

MPI_Bcast(Address, Count, Datatype, *Root, Comm*)

- 标号为Root的进程发送相同的消息给标记为Comm的通信子中的所有进程。
- 消息的内容如同点对点通信一样由三元组(Address, Count, Datatype)标识。对Root进程来说，这个三元组既定义了发送缓冲也定义了接收缓冲。对其它进程来说，这个三元组只定义了接收缓冲。



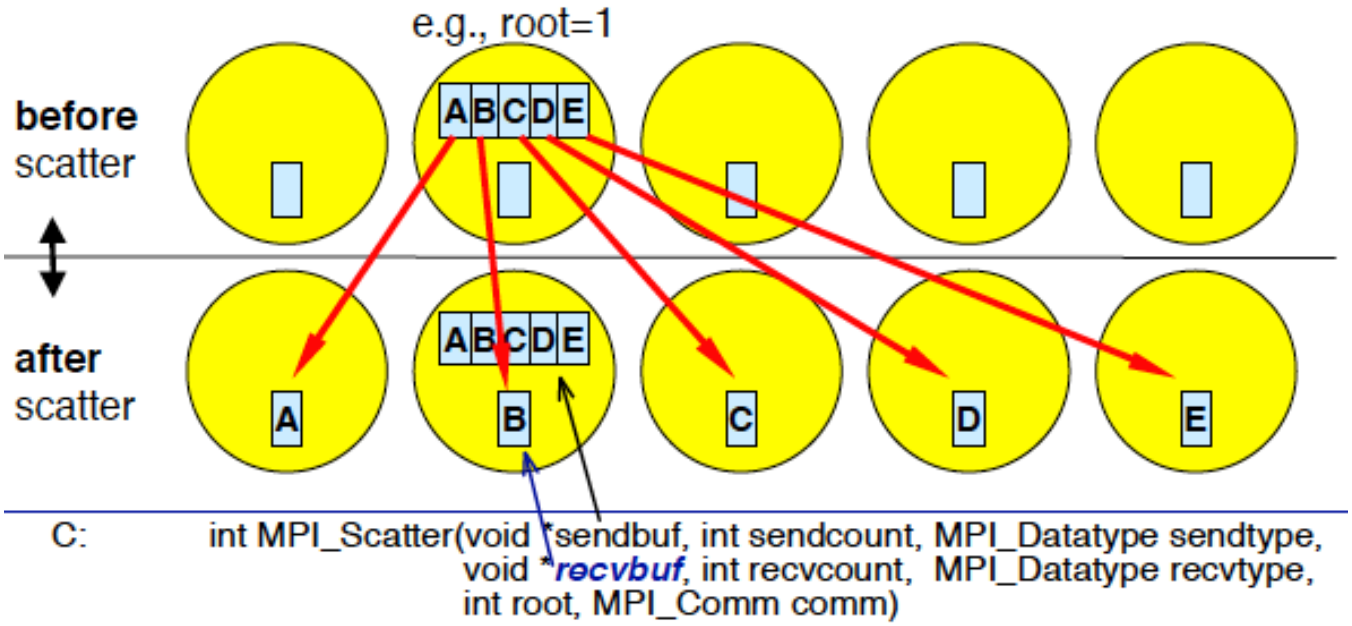
MPI_Bast

```
int argc;
char **argv;
{
int rank, value;
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
do {
if (rank == 0) /*进程0读入需要广播的数据*/
        scanf( "%d", &value );
MPI_Bcast( &value, 1, MPI_INT, 0, MPI_COMM_WORLD );/*将该数据广播出去*/
printf( "Process %d got %d\n", rank, value );/*各进程打印收到的数据*/
} while (value >= 0);
MPI_Finalize( );
return 0;
}
```

Scatter

MPI_Scatter (SendAddress,SendCount,SendDatatype,
RecvAddress,RecvCount,RecvDatatype,Root,Comm)

- Root进程发送给所有n个进程各发送一个不同的消息，包括自己。
- 这n个消息在Root进程的发送缓冲区中按标号的顺序有序地存放。每个接收缓冲由三元组(RecvAddress, RecvCount, RecvDatatype)标识。非Root进程忽略发送缓冲。
- 对Root进程，发送缓冲由三元组(SendAddress, SendCount, SendDatatype)标识.



MPI_Scatter

- 根进程向组内每个进程散播100个整型数据

```
MPI_Comm comm;
int gsize,*sendbuf;
int root,rbuf[100];
......
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
......
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

MPI_Scatterv

- 根进程向各个进程发送个数不等的数据

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

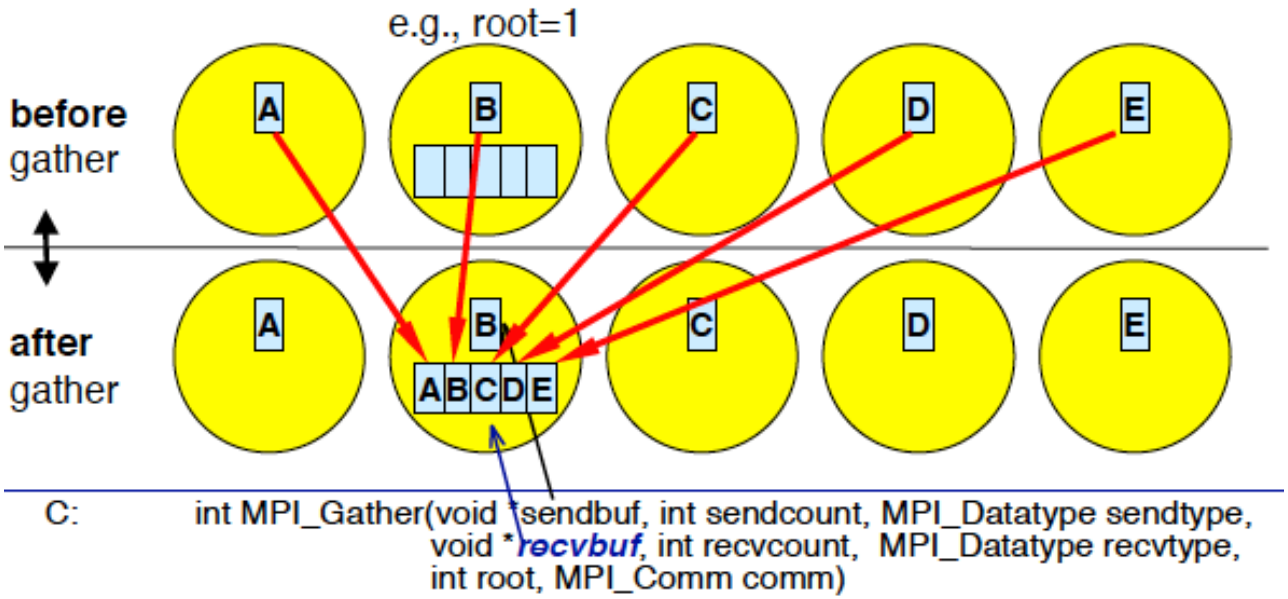
IN	sendbuf	发送消息缓冲区的起始地址(可选数据类型)
IN	sendcounts	发送数据的个数，整数数组 (整型)
IN	displs	发送数据偏移，整数数组(整型)
IN	sendtype	发送消息缓冲区中元素类型(句柄)
OUT	recvbuf	接收消息缓冲区的起始地址(可变)
IN	recvcount	接收消息缓冲区中数据的个数(整型)
IN	recvtype	接收消息缓冲区中元素的类型(句柄)
IN	root	发送进程的标识号(句柄)
IN	comm	通信域(句柄)

```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                 MPI_Comm comm)
```

Gather

MPI_Gather (SendAddress,SendCount,SendDatatype,
RecvAddress,RecvCount,RecvDatatype,Root,Comm)

- Root进程接收各个进程(包括它自己)的消息。这n个消息的连接按序号rank进行，存放在Root进程的接收缓冲中。
- 每个发送缓冲由三元组(SendAddress, SendCount, SendDatatype) 标识。
- 非Root进程忽略接收缓冲。对Root进程，发送缓冲由三元组(RecvAddress, RecvCount, RecvDatatype)标识。RecvCount是自每个进程接收数据个数。



MPI_Gather

- 自进程组中每个进程收集100个整型数给根进程

```
MPI_Comm comm;
int gsize, sendarray[100];
int root,*rbuf;
.....
MPI_Comm_size(comm,&gsize);
rbuf=(int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray,100,MPI_INT,rbuf,100,MPI_INT,root,comm);
```

MPI_Gatherv

- 从不同进程接收不同数量的数据

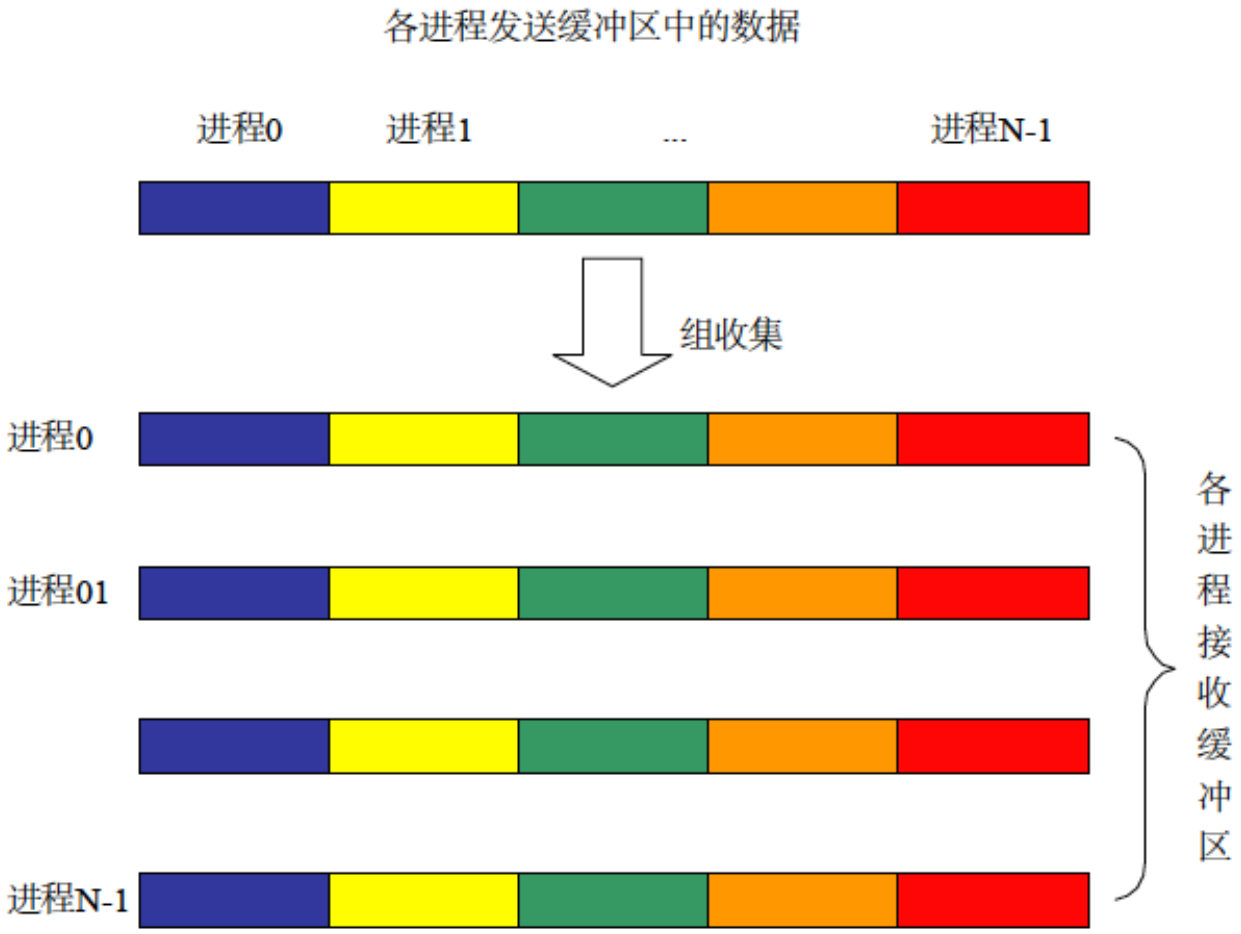
MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,recvtype,
root, comm)

IN	sendbuf	发送消息缓冲区的起始地址(可选数据类型)
IN	sendcount	发送消息缓冲区中的数据个数(整型)
IN	sendtype	发送消息缓冲区中的数据类型(句柄)
OUT	recvbuf	接收消息缓冲区的起始地址(可选数据类型,仅对于根进程有意义)
IN	recvcounts	整型数组(长度为组的大小), 其值为从每个进程接收的数据个数
IN	displs	整数数组,每个入口表示相对于recvbuf的位移
IN	recvtype	接收消息缓冲区中数据类型 (句柄)
IN	root	接收进程的标识号(句柄)
IN	comm	通信域(句柄)

```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf,
        int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Allgather

MPI_Allgather (SendAddress, SendCount, SendDatatype,
RecvAddress, RecvCount, RecvDatatype, Comm)



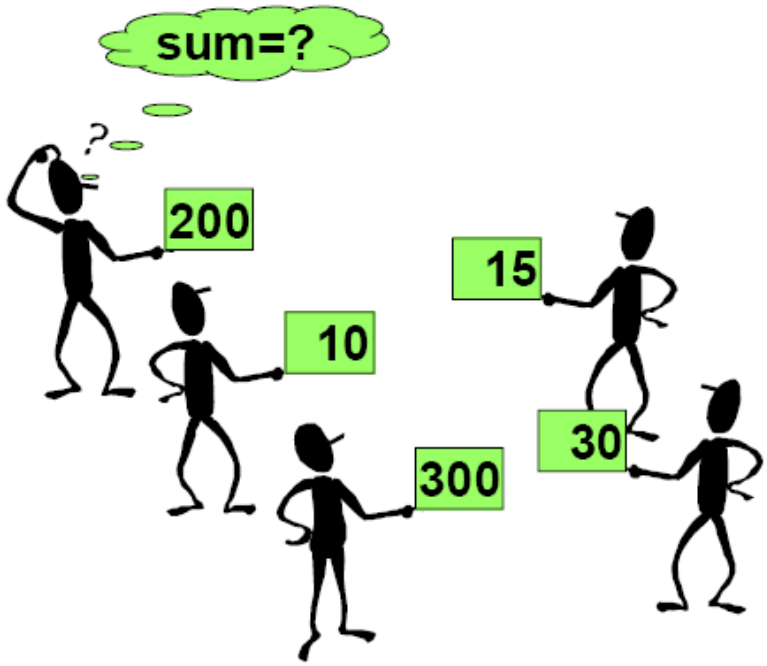
MPI_Allgather

- 每个进程都从其他进程收集100个数据，存入自己的缓冲区内

```
MPI_Comm comm;
int gsize,sendarray[100];
int *rbuf;
.....
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

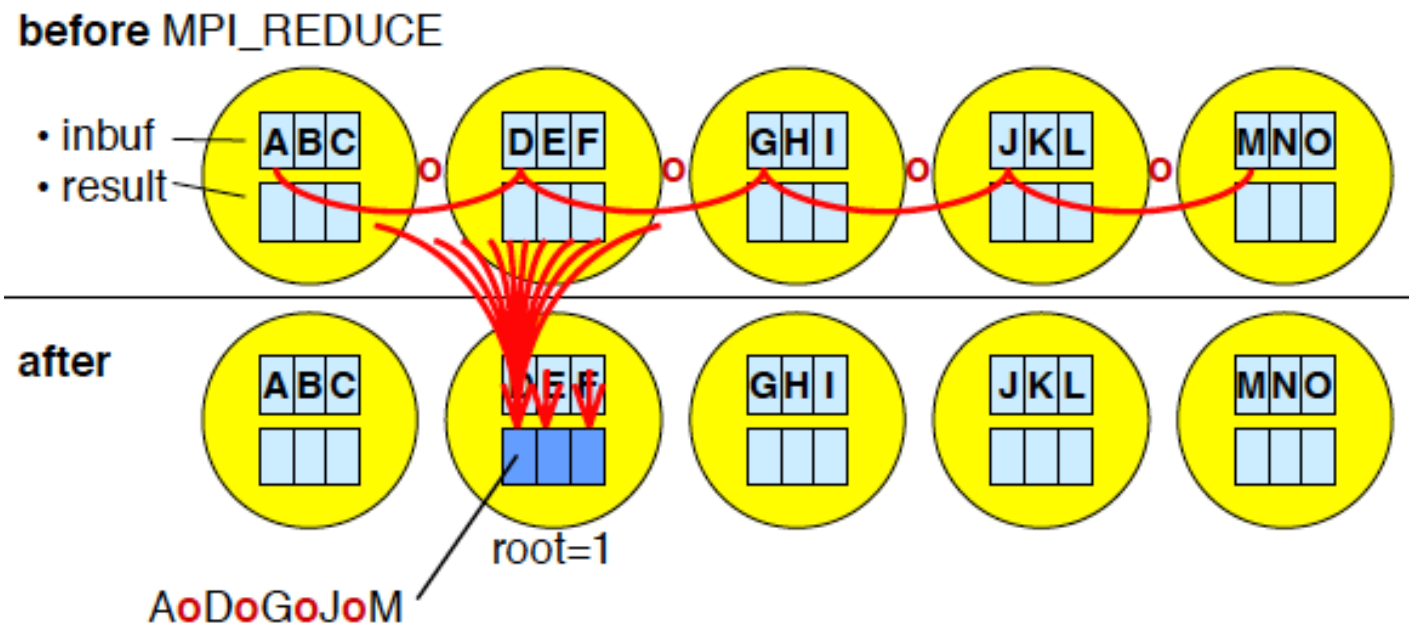
归约（Reduce）

- 所有进程向同一进程发送消息，与broadcast的消息发送方向相反。
- 接收进程对所有收到的消息进行归约处理。
- 归约操作：
  - MAX, MIN, SUM, PROD, LAND, BAND, LOR, BOR, LXOR, BXOR, MAXLOC, MINLOC



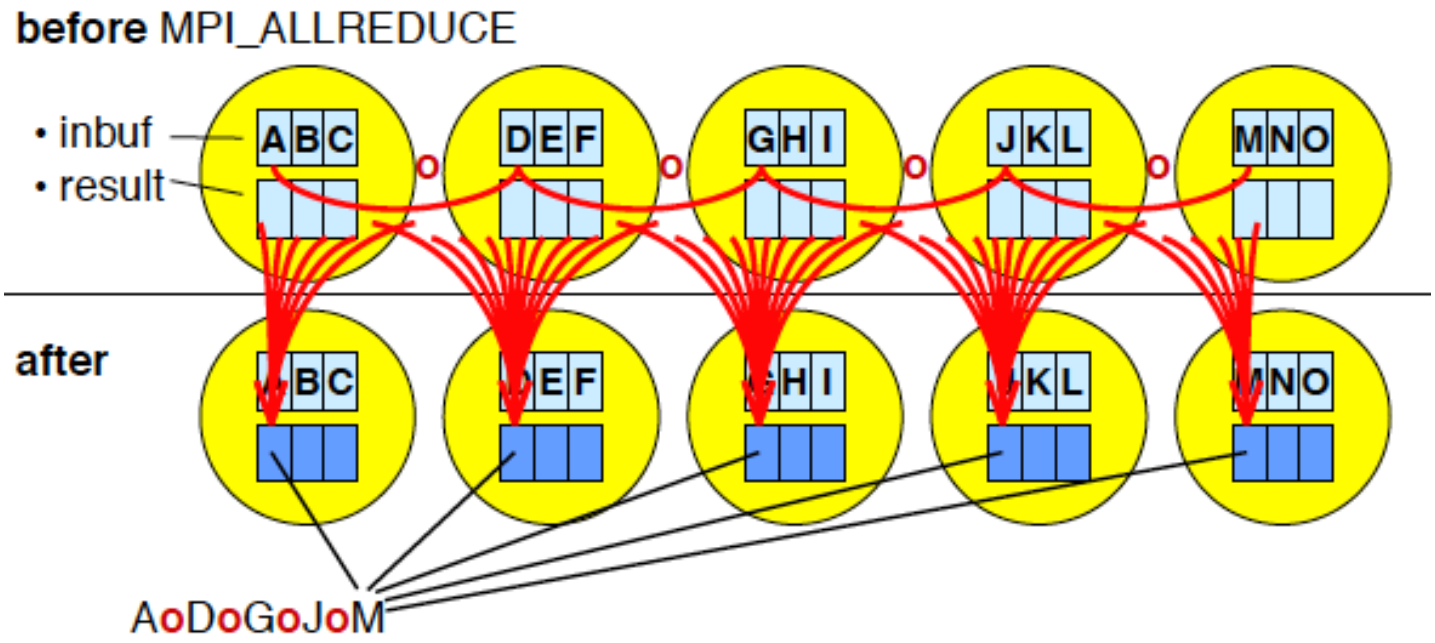
MPI_Reduce

MPI_REDUCE(inbuf,result,count,datatype,op,root,comm)



MPI_Allreduce

- 语法与reduce类似，但无root参数
- 所有进程都将获得结果



MPI_Reduce_scatter

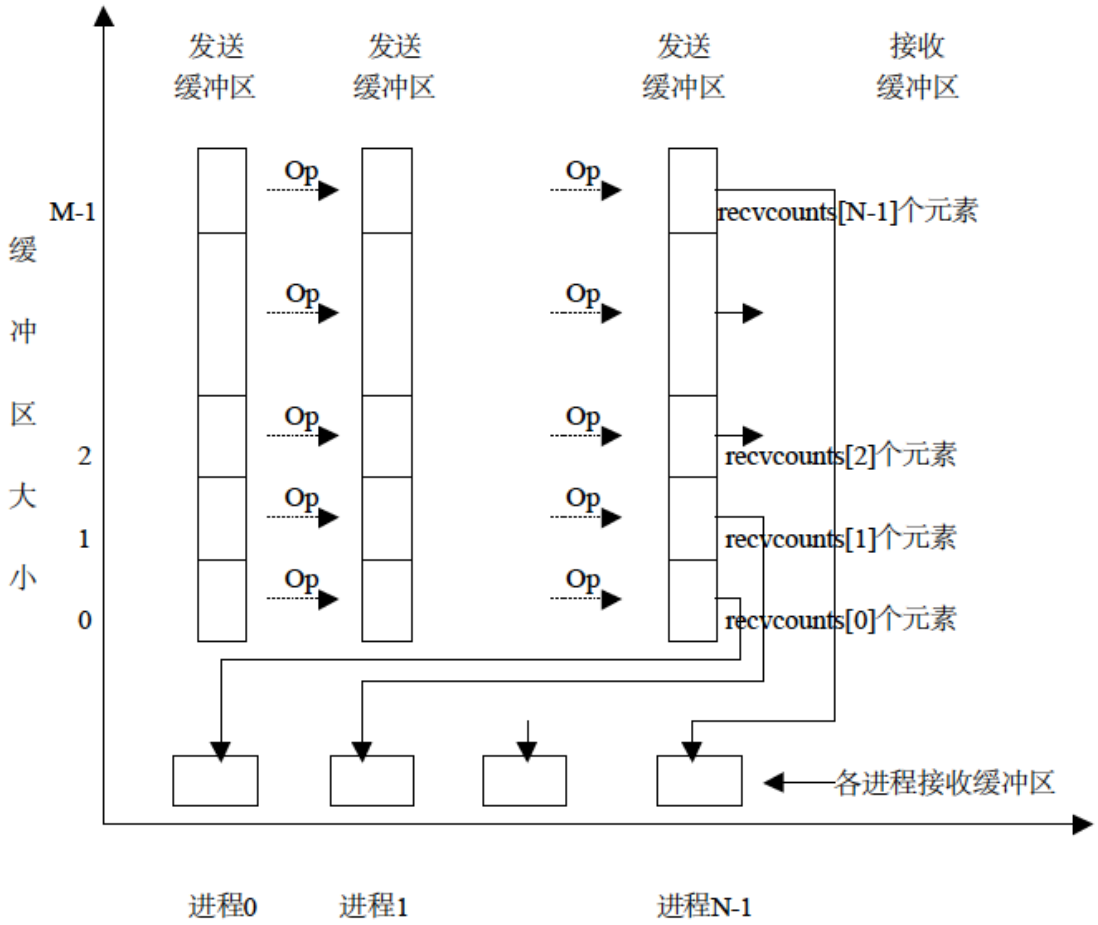
- 将归约结果散播到所有进程中

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)

IN	sendbuf	发送消息缓冲区的起始地址(可选数据类型)
OUT	recvbuf	接收消息缓冲区的起始地址(可选数据类型)
IN	recvcounts	接收数据个数〈整型数组〉
IN	datatype	发送缓冲区中的数据类型(句柄)
IN	op	操作(句柄)
IN	comm	通信域(句柄)

```
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts
                       MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

MPI_Reduce_scatter



MPI_Scan

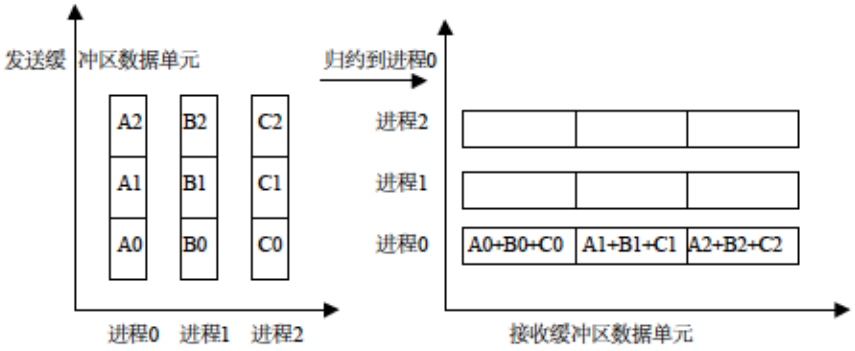
- 每一个进程都对排在它前面的进程进行归约操作。
- MPI_SCAN调用的结果是，对于每一个进程i，它对进程0,...,i的发送缓冲区的数据进行指定的归约操作，结果存入进程i的接收缓冲区。

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)

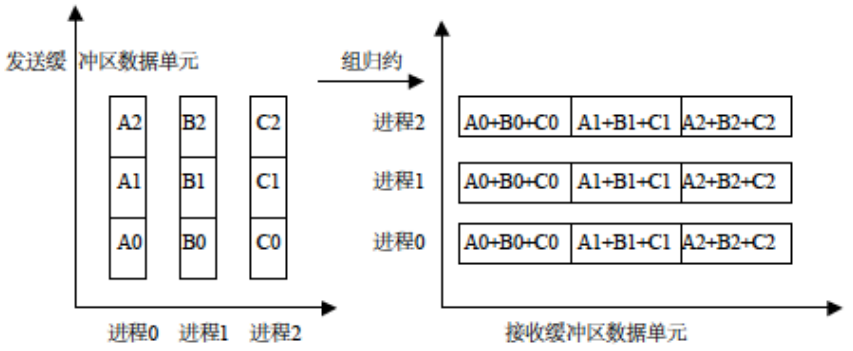
IN	sendbuf	发送消息缓冲区的起始地址(可选数据类型)
OUT	recvbuf	接收消息缓冲区的起始地址(可选数据类型)
IN	count	输入缓冲区中元素的个数(整型)
IN	datatype	输入缓冲区中元素的类型(句柄)
IN	op	操作(句柄)
IN	comm	通信域(句柄)

int MPI_Scan(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

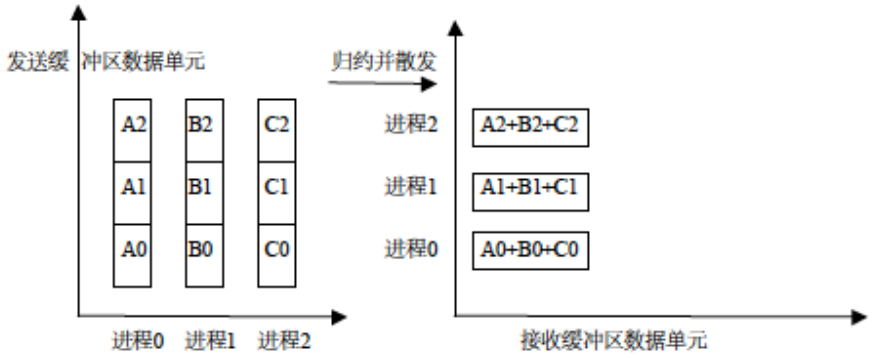
不同类型的归约操作对比



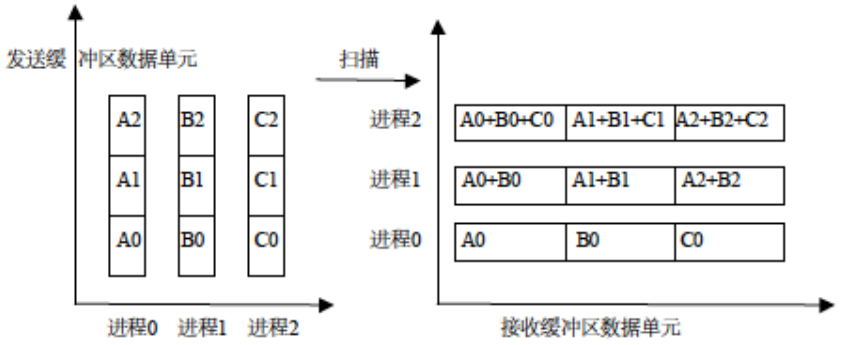
Reduce



Allreduce



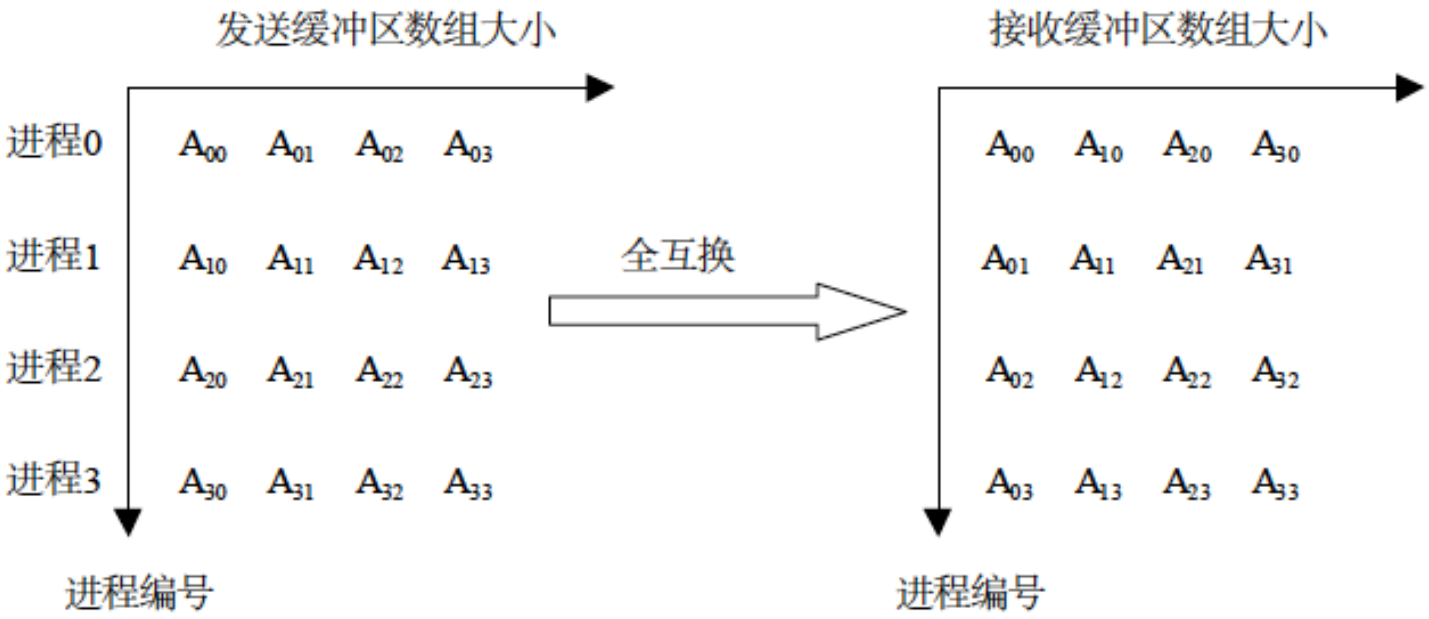
Reduce_scatter



Scan

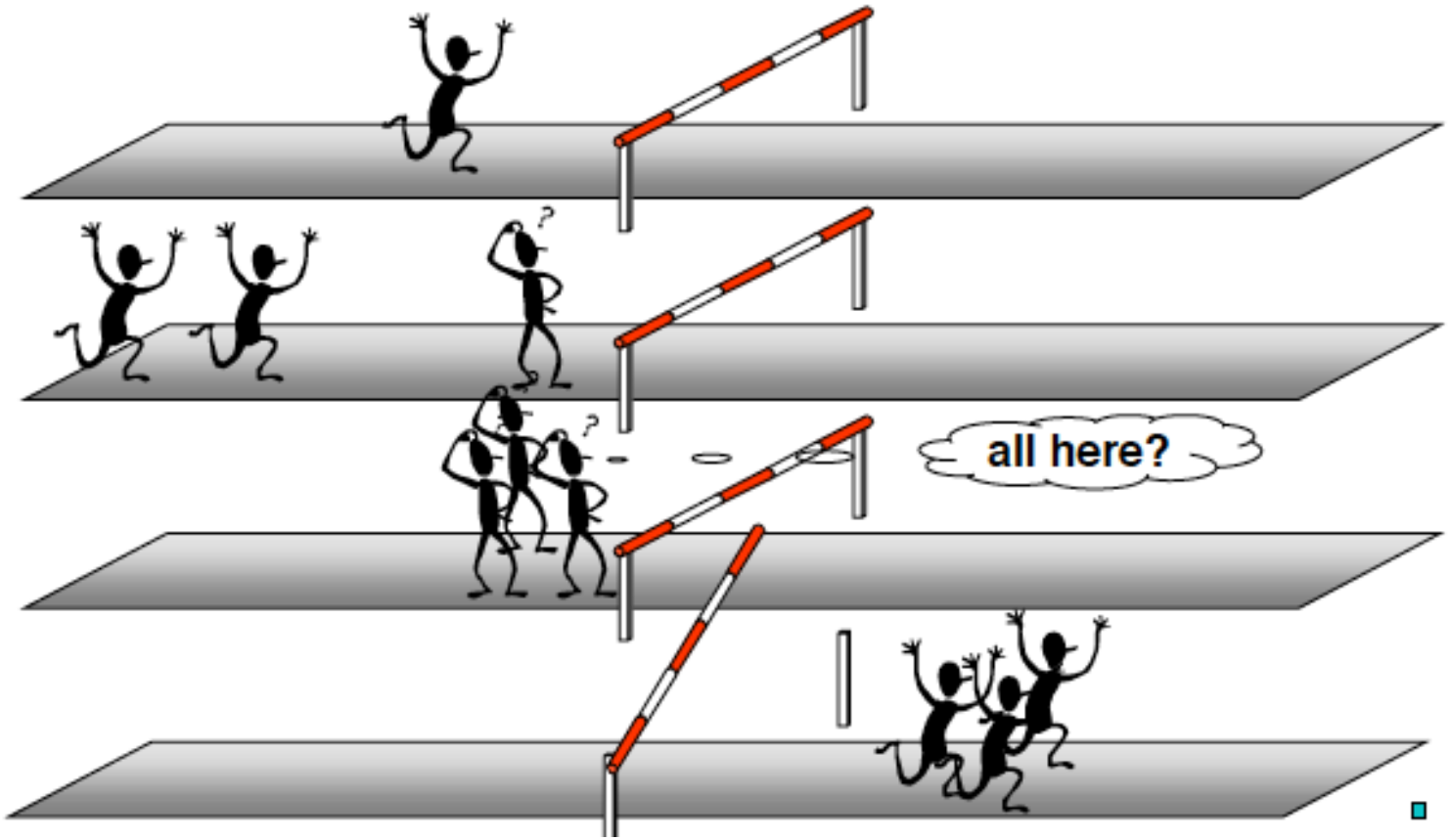
MPI_Alltoall

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

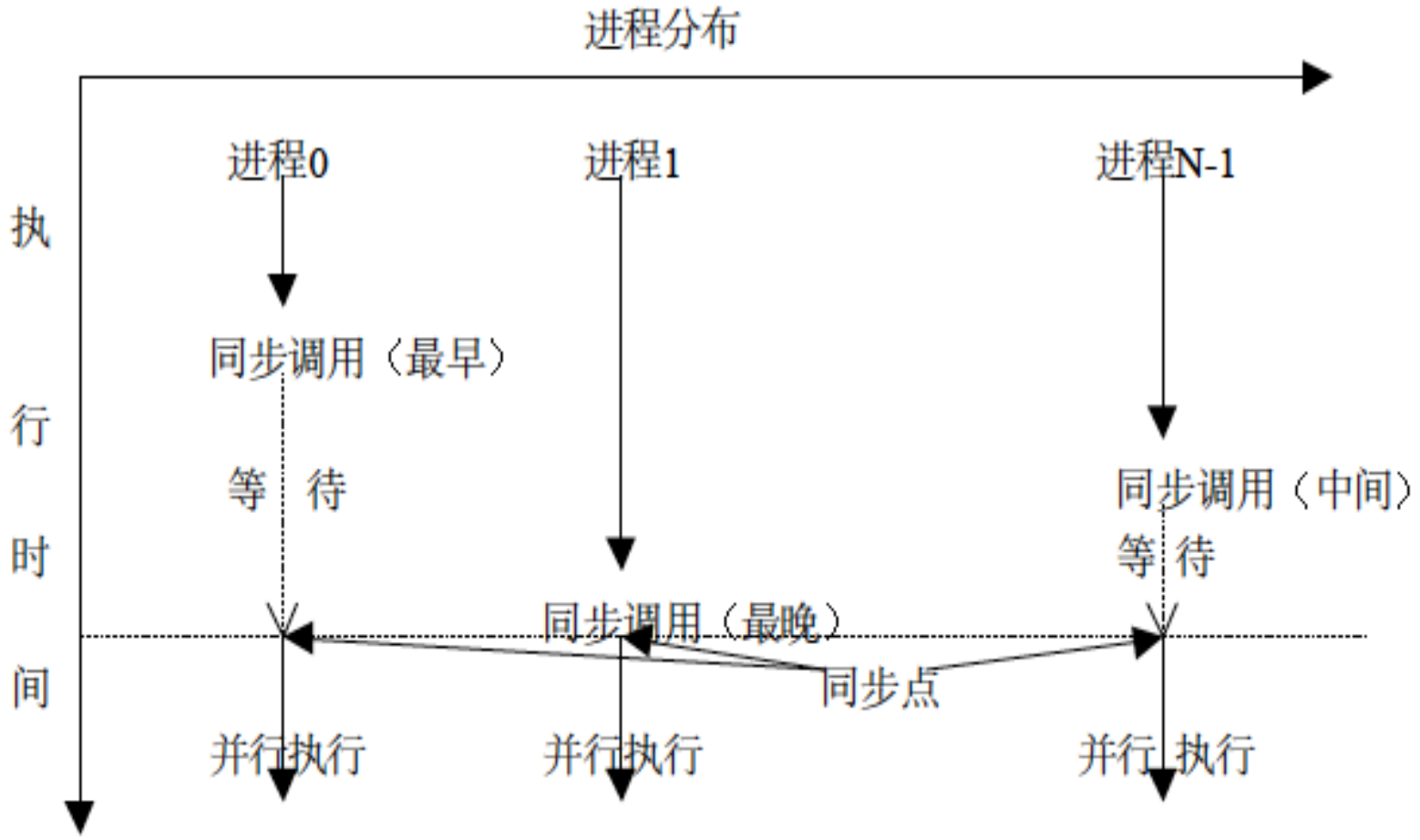


每个进程依次将它的发送缓冲区的第i块数据发送给第i个进程，同时每个进程又都依次从第j个进程接收数据放到各自接收缓冲区的第j块数据区的位置

MPI_Barrier



MPI_Barrier


进程分布
进程0
进程1
进程N-1
执
行
时
间
同步调用（最早）
等 待
同步调用（中间）
等 待
同步调用（最晚）
同步点
并行执行
并行执行
并行 执行

Outline

- MPI概述
- 点到点通信
- 组通信
- 阻塞通信模式

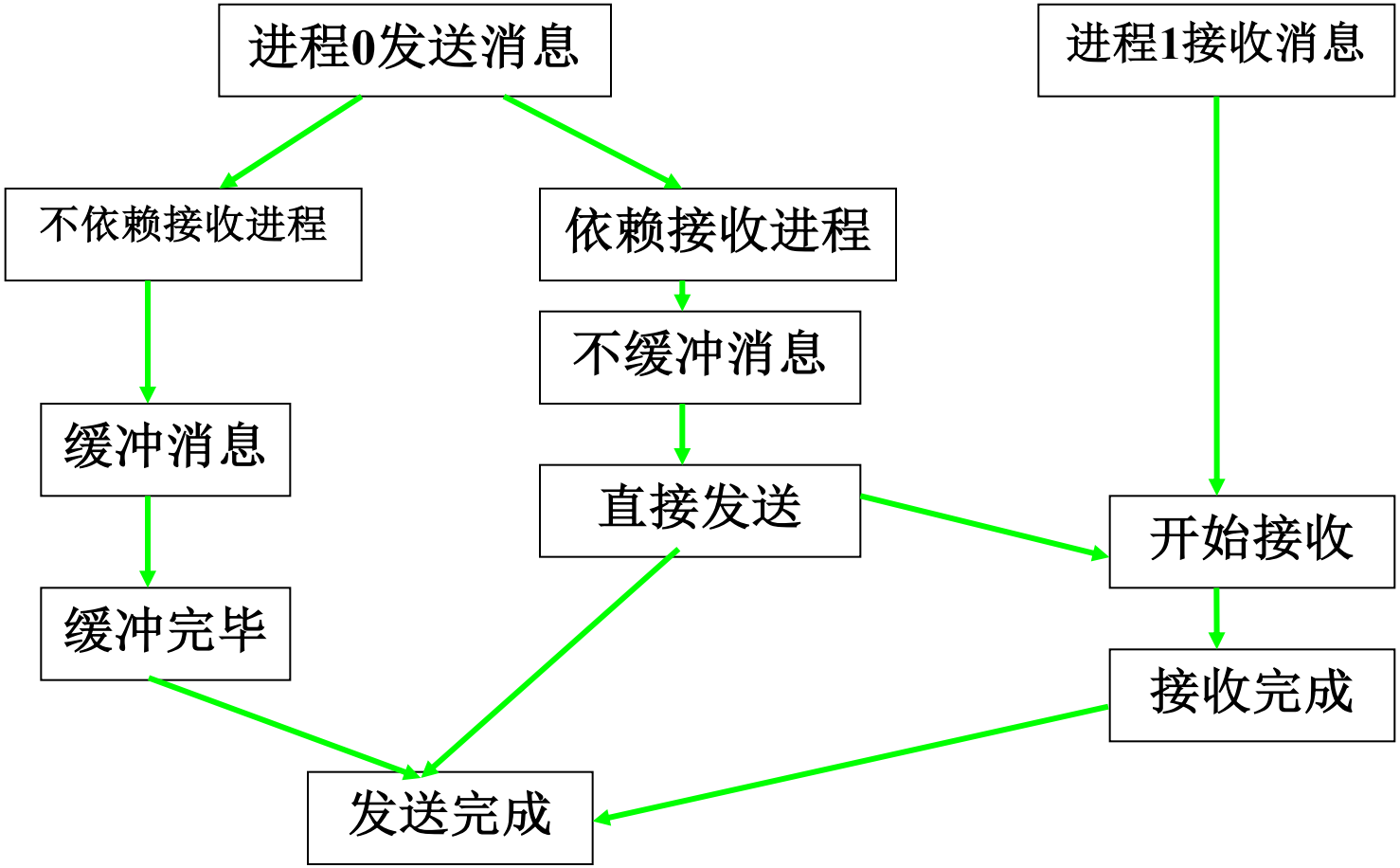
阻塞通信模式

- 标准通信模式
- 缓存通信模式
- 同步通信模式
- 就绪通信模式

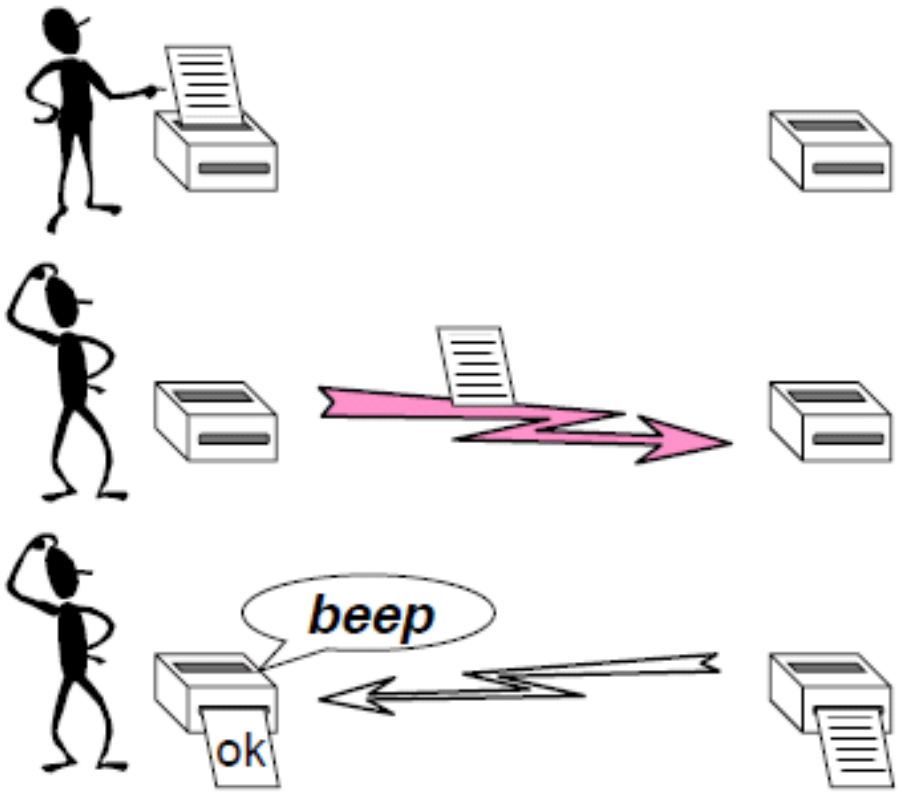
标准通信模式

- MPI_Send
- 在MPI采用标准通信模式时，是否对发送的数据进行缓存是由MPI自身决定的，而不是由并行程序员来控制。
- 如果MPI决定缓存将要发出的数据，发送操作不管接收操作是否执行，都可以进行，而且发送操作可以正确返回，而不要求接收操作收到发送的数据。

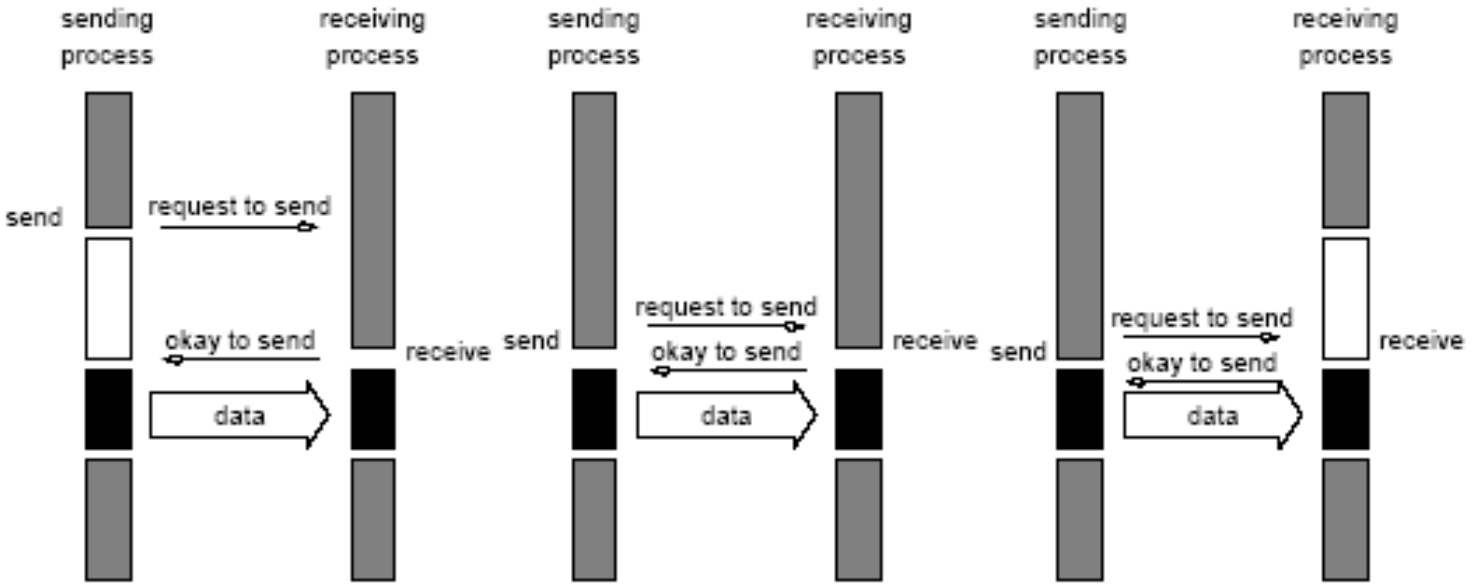
标准通信模式


进程0发送消息
进程1接收消息
不依赖接收进程
依赖接收进程
不缓冲消息
缓冲消息
直接发送
开始接收
缓冲完毕
接收完成
发送完成

不缓存发送



不缓存发送

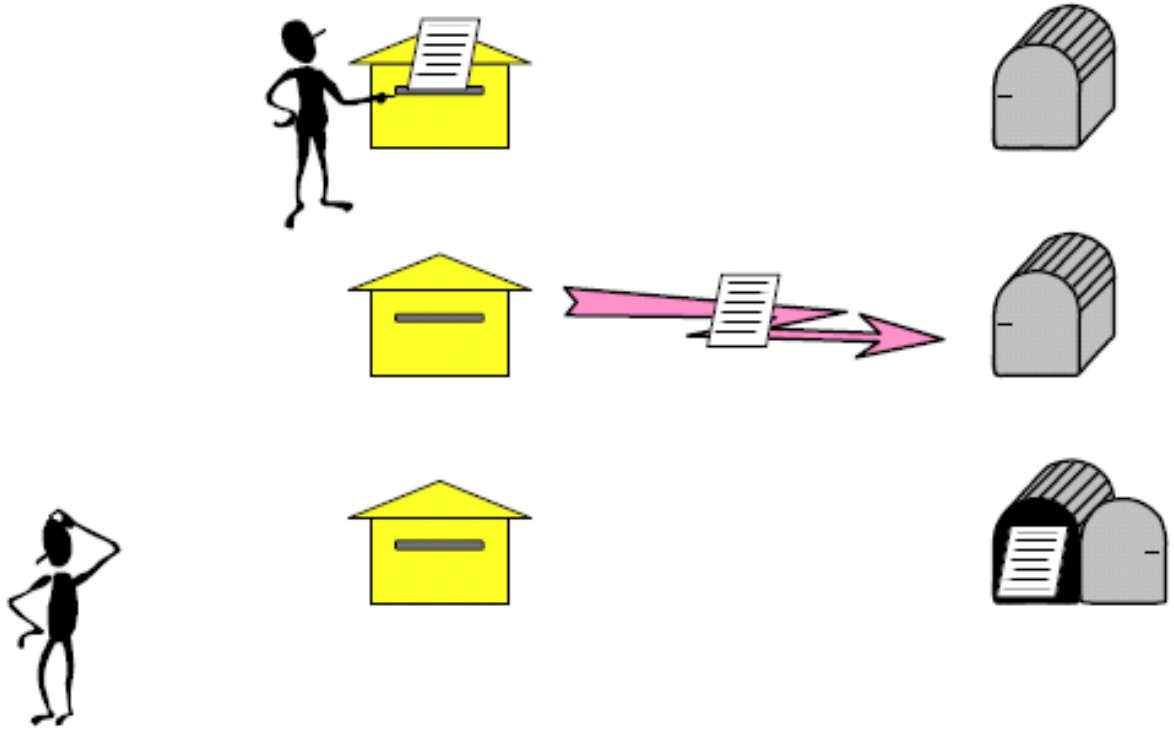


(a) Sender comes first; idling at sender

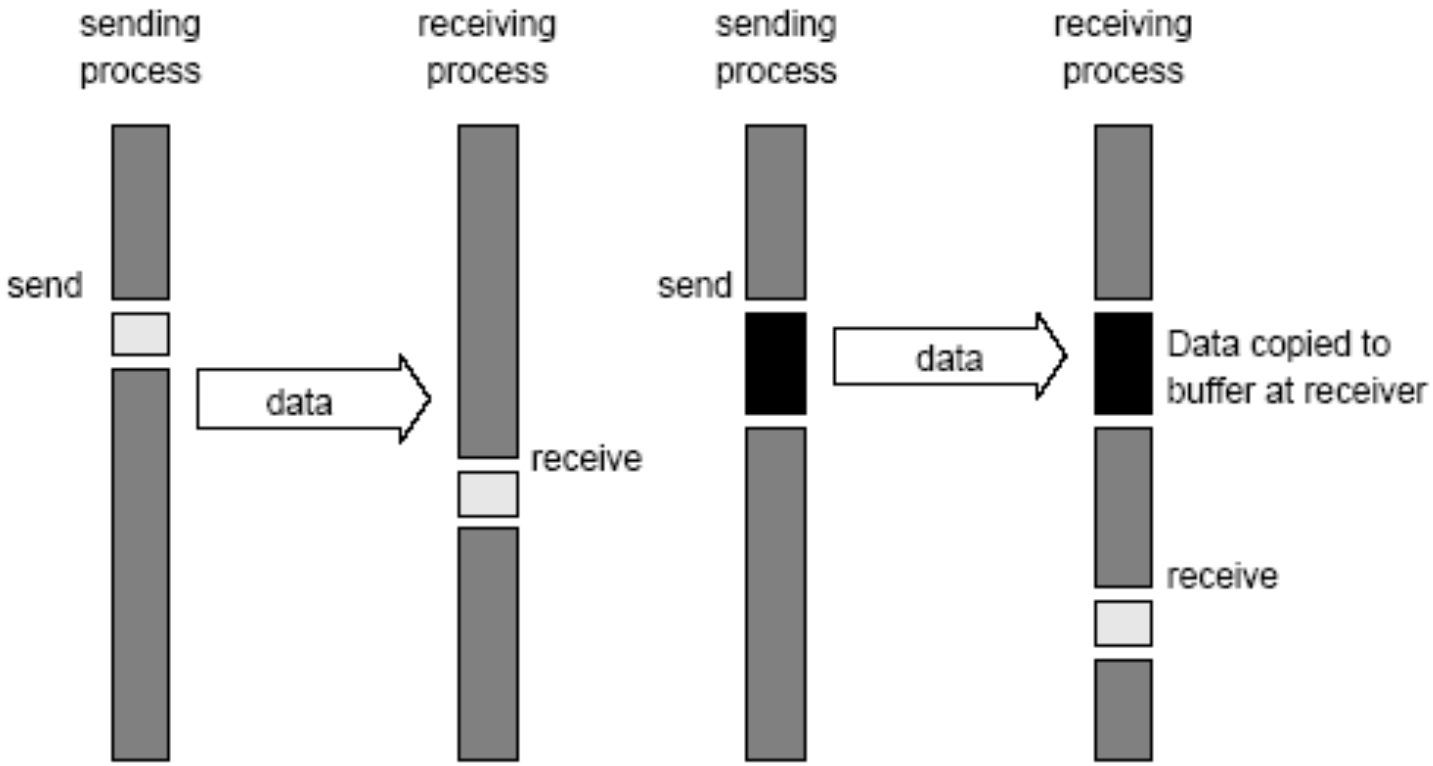
(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

缓存发送



缓存发送

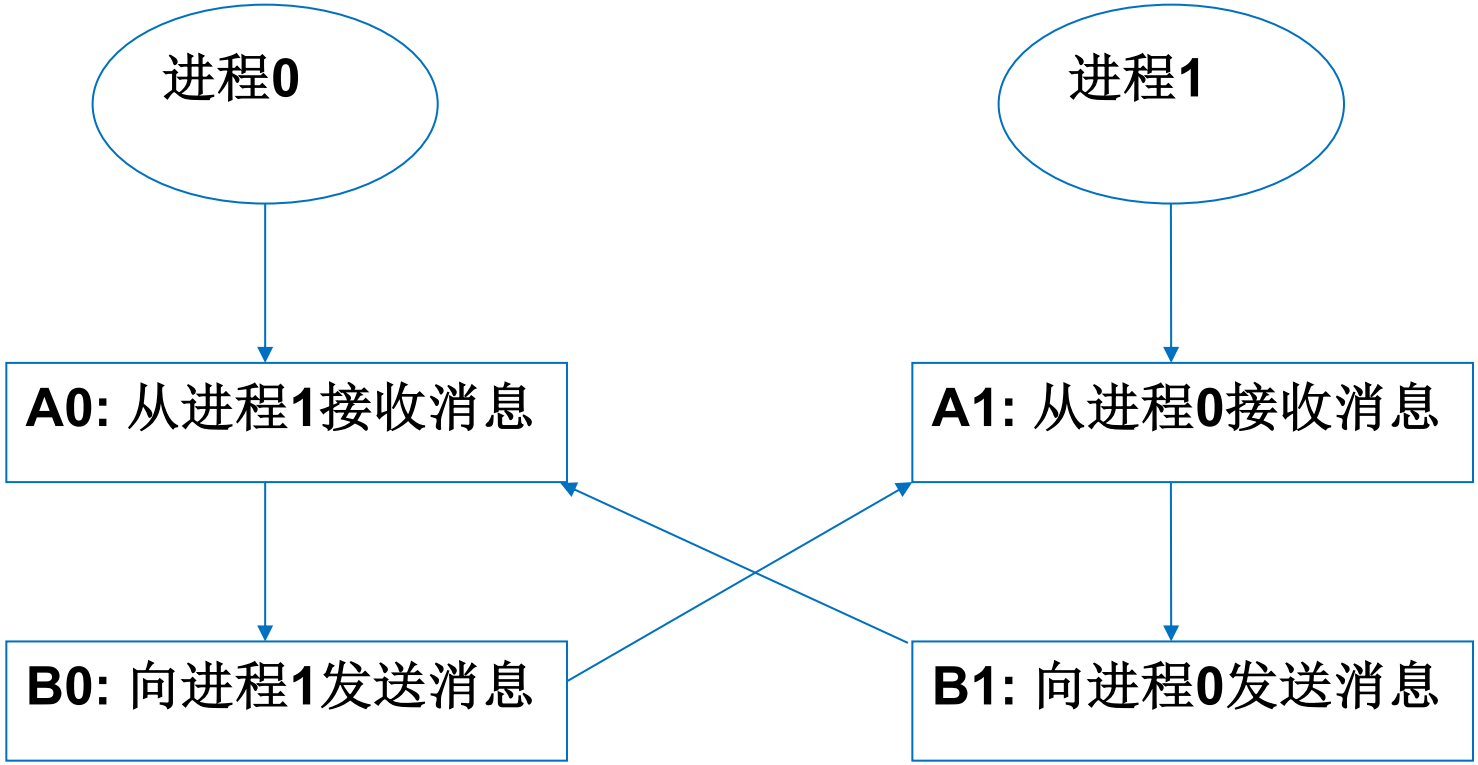


使用系统缓存　　不使用系统缓存

进程间通信的组织

```
……
MPI_Comm_dup (MPI_COMM_WORLD, &comm);
If ( myid==0) {
  MPI_Recv(bufA0,1,MPI_Float,1,101,comm,status);
  MPI_Send(bufB0,1,MPI_Float,1,100,comm);}
else if (myid==1){
  MPI_Recv(bufA1,1,MPI_Float,0,100,comm,status);
  MPI_Send(bufB1,1,MPI_Float,0,101,comm);}
……
```

进程间通信的组织

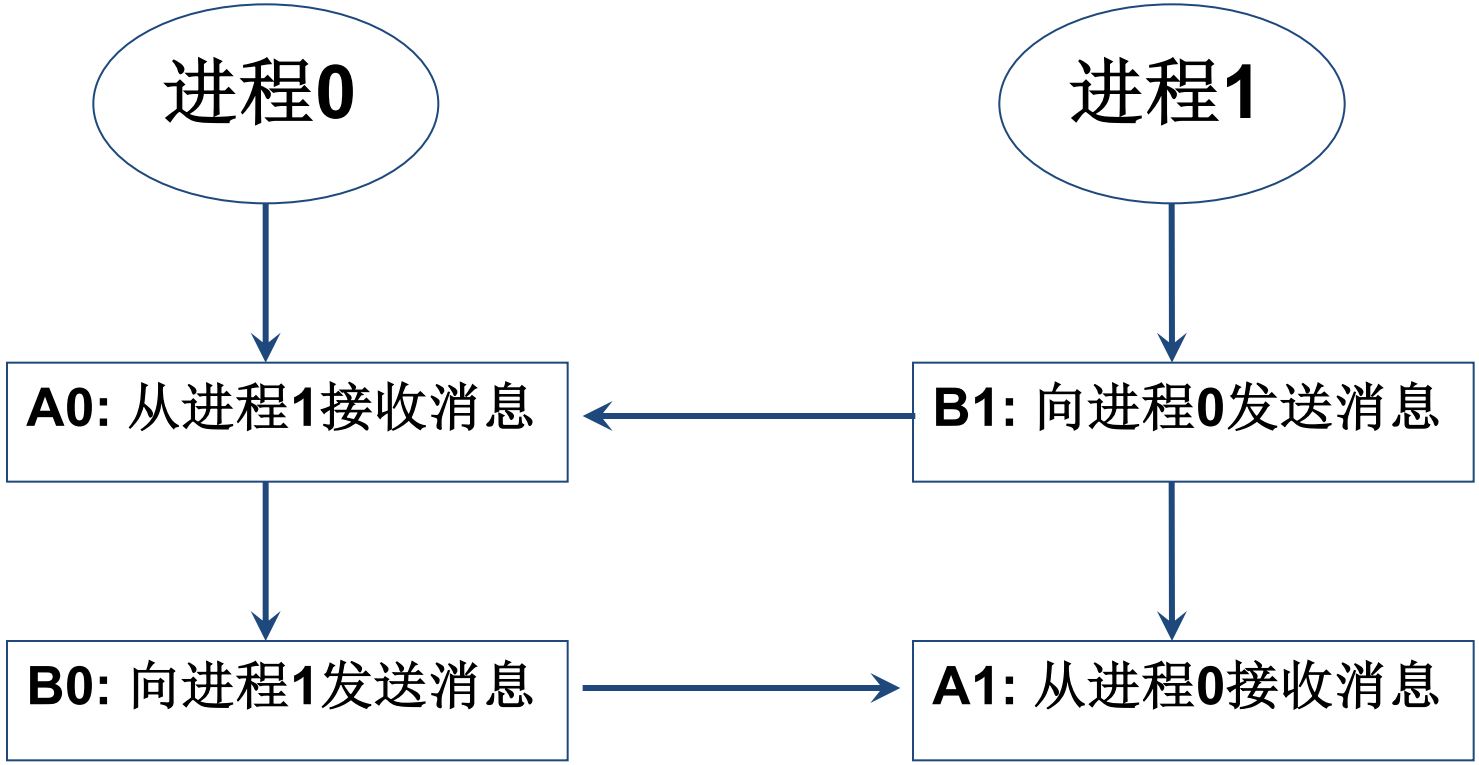


进程间通信的组织

```
……
MPI_Comm_dup (MPI_COMM_WORLD, &comm);
If ( myid==0) {

  MPI_Recv(bufA0,1,MPI_Float,1,101,comm,status);
  MPI_Send(bufB0,1,MPI_Float,1,100,comm);}

else if (myid==1){

  MPI_Recv(bufA1,1,MPI_Float,0,100,comm,status);
  MPI_Send(bufB1,1,MPI_Float,0,101,comm);}
……
```

死锁

死锁的避免



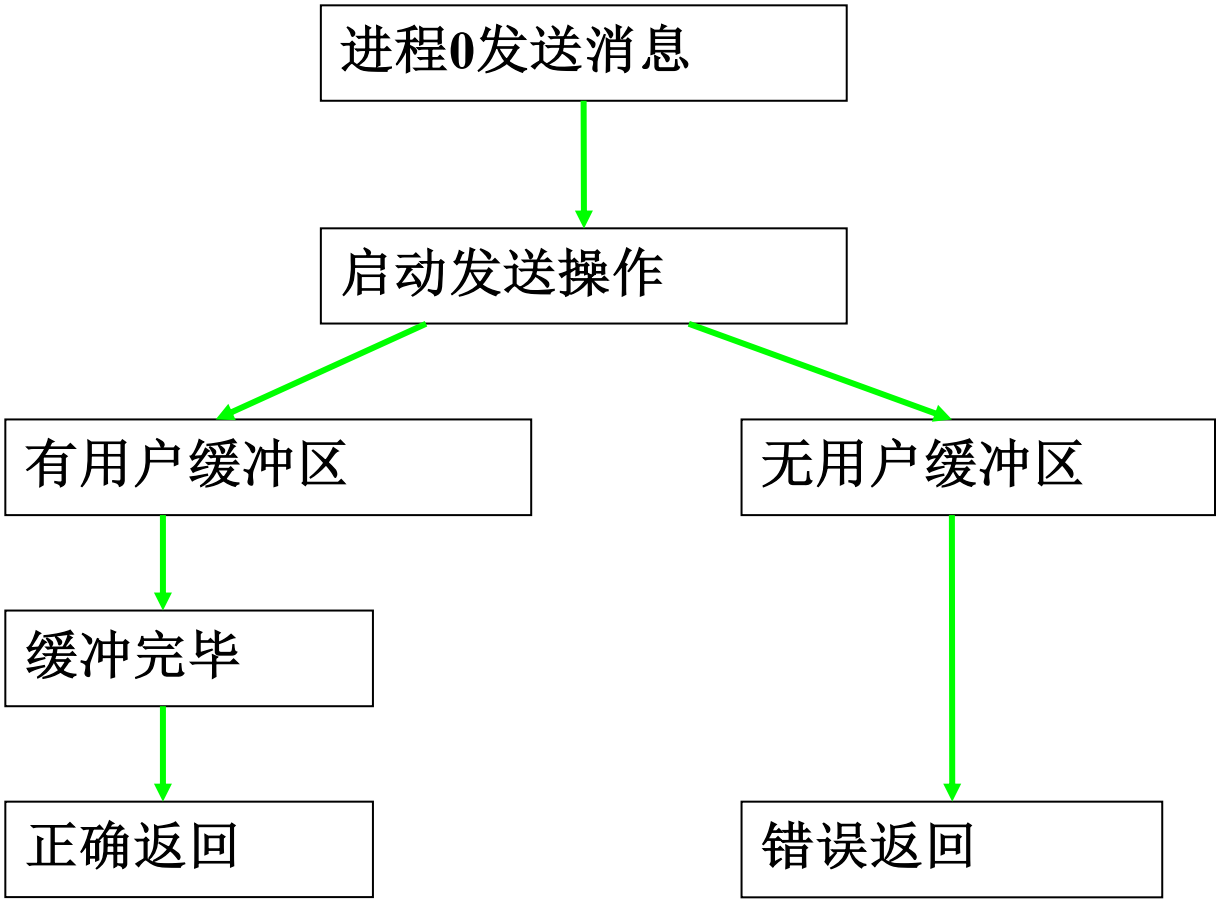
上例的修改

```
……
If ( myid==0) {

    MPI_Recv(bufA0,1,MPI_Float,1,101, comm, status);
    MPI_Send(bufB0,1,MPI_Float,1,100, comm);}

else if (myid==1){

    MPI_Send(bufB1,1,MPI_Float,0,101, comm);}
    MPI_Recv(bufA1,1,MPI_Float,0,100, comm, status);
    ……
```

缓存通信模式

- MPI_Bsend
  - 由用户直接对通信缓冲区进行申请、使用和释放。
  - 缓存模式下对通信缓冲区的合理与正确使用由程序设计人员自己保证。
- MPI_BSEND参数的含义和MPI_SEND的完全相同，不同之处仅表现在通信时是使用标准的系统提供的缓冲区还是用户自己提供的缓冲区。

缓存通信模式



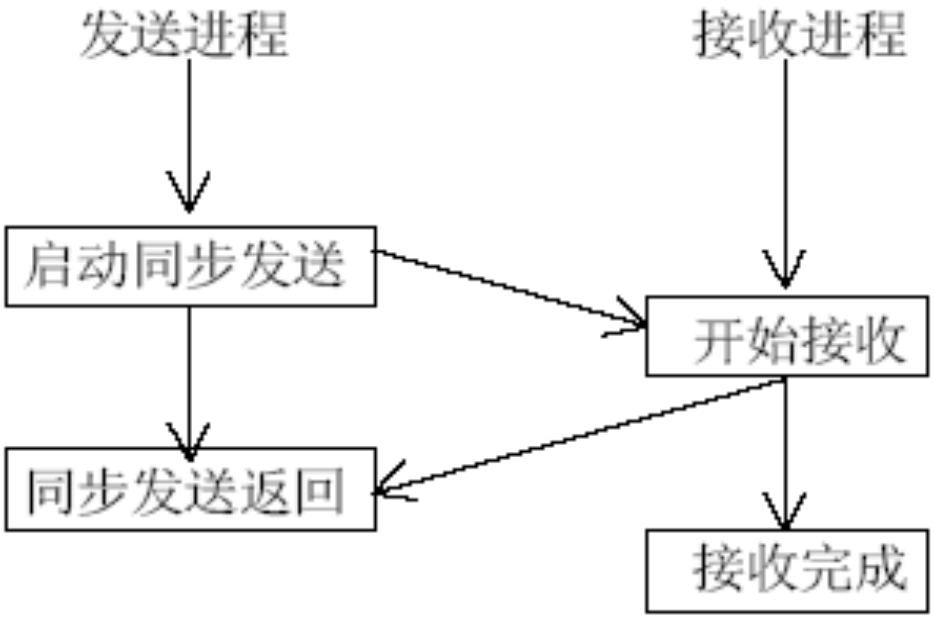
缓存通信模式

- MPI_BUFFER_ATTACH
  - 将大小为size的缓冲区递交给MPI，这样该缓冲区就可以作为缓存发送时的缓存来使用。
- MPI_BUFFER_DETACH
  - 将提交的大小为size的缓冲区buffer收回。
  - 该调用是阻塞调用它一直等到使用该缓存的消息发送完成后才返回，这一调用返回后用户可以重新使用该缓冲区，或者将这一缓冲区释放。

同步通信模式

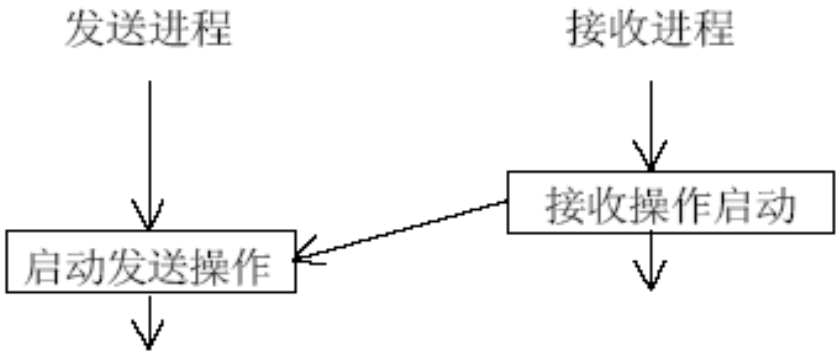
- MPI_Ssend
  - 同步通信模式的开始不依赖于接收进程相应的接收操作是否已经启动，但是**同步发送**必须等到相应的接收进程开始后才可以正确返回。
  - 同步发送返回后，意味着发送缓冲区中的数据已经全部被系统缓冲区缓存并且已经开始发送。
  - 当同步发送返回后发送缓冲区可以被释放或重新使用。

同步通信模式



就绪通信模式

- MPI_Rsend
  - 只有当接收进程的接收操作已经启动时，才可以在发送进程启动发送操作，否则，当发送操作启动而相应的接收还没有启动时，发送操作将出错。
  - 对于非阻塞发送操作的正确返回，并不意味着发送已完成；但对于阻塞发送的正确返回，则发送缓冲区可以重复使用。



就绪通信模式

- 一种安全的就绪通信模式