

第6讲 MPI编程

----- (一) 基础部分



内容&阅读

- 消息传递编程，主要用于分布式内存系统
- 消息传递接口MPI(Message Passing Interface)
 - 最常用的分布式内存大规模计算编程语言
- 参考教材《并行程序设计导论》第三章 分布式内存编程——MPI
- 其他资源
 - 课程幻灯片
 - <http://www.mpich.org/documentation/guides/>
 - **Tutorials**部分有好几个PPT文件，推荐阅读



提纲

- MPI概念和基本原语
- MPI编程模型
- 点对点通信进阶
- 组通信
- 自定义数据类型
- 进程组与通信域
- 虚拟拓扑
- MPI新标准



提纲

- MPI概念和基本原语
- MPI编程模型
- 点对点通信进阶
- 组通信
- 自定义数据类型
- 进程组与通信域
- 虚拟拓扑
- MPI新标准



回顾：两类并行体系结构组织

○ 共享内存多处理器体系结构

- 一组自治的处理器，连接到单一内存系统
- 支持共享全局地址空间，每个处理器都能访问任何一个内存位置

○ 分布式内存体系结构

- 一组自治的系统，由一个互联结构连接
- 每个系统有自己独立的地址空间，处理器必须显式通信才能共享数据
- 商用网络连接的PC集群是最常见的例子



消息传递和MPI

- 消息传递是超级计算机和集群主要的编程模型
 - 可移植
 - 偏底层
- MPI是什么？
 - 消息传递编程模型标准，取代专有库
 - 编程角度，一个库，与Fortran、C/C++一起使用
 - 基于单程序多数据流（SPMD）
 - 隔离了独立地址空间
 - 不会有数据竞争，但可能有通信错误
 - 暴露了执行模型，迫使程序员思考局部性，两点对性能都有好处
 - 编程复杂，代码膨胀



消息传递库的特性

- 所有通信、同步都需调用函数完成

- 无共享变量

- 提供如下类别的函数

- 通信

- 点对点通信：消息从特定的发送进程（点A）发送到特定的接收进程（点B）

- 多处理器参与的组通信

- 移动数据：广播、散发/收集
 - 计算并移动数据：归约、全归约

- 同步

- 障碍

- 无锁机制，因为没有共享变量需要保护

- 查询

- 多少个进程？哪个是我？有处于等待状态的消息？



MPI参考资料

- MPI最权威的资料

- <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>



Hello World

- 关键问题：获得运行环境信息
 - 这次计算有多少进程参与？
 - 我是哪一个？
- MPI提供两个原语回答这两个问题：
 - **MPI_Comm_size**报告进程数
`int MPI_Comm_size(MPI_Comm comm, int *size)`
 - **MPI_Comm_rank**报告识别调用进程的**rank**,
值从0~size-1
`int MPI_Comm_rank(MPI_Comm comm, int *rank)`



Hello World (C版本)

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

```
int main(int argc, char *argv[]){
    int myid, numprocs;
    int namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Get_processor_name(processor_name,&namelen);
    fprintf(stderr,"Hello World! Process %d of %d on %s\n",
        myid, numprocs, processor_name);
    MPI_Finalize();
    return 0;
}
```

获得处理器名



Hello World (C++版本)

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI::Init(argc, argv);
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    std::cout << "Greetings from process " << rank << " of "
    << size << "\n";
    MPI::Finalize();
    return 0;
}
```

编译——Linux平台





`mpiexec -n <number of processes> <executable>`

`mpiexec -n 1 ./mpi_hello`

启动1个进程运行

`mpiexec -n 4 ./mpi_hello`

启动4个进程运行



运行结果

```
mpiexec -n 1 ./mpi_hello
```

Greetings from process 0 of 1 !

```
mpiexec -n 4 ./mpi_hello
```

Greetings from process 0 of 4 !

Greetings from process 1 of 4 !

Greetings from process 2 of 4 !

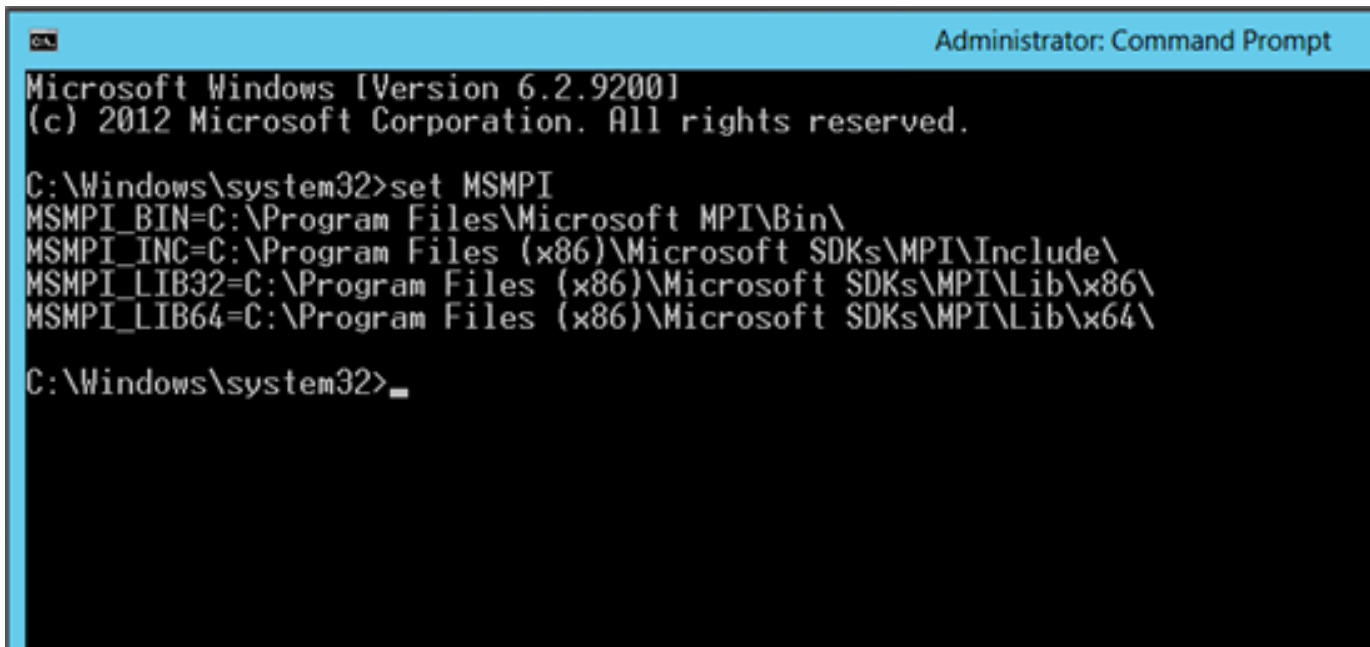
Greetings from process 3 of 4 !

Windows平台——MSMPI

- 安装MSMPI和MSMPI SDK

<https://www.microsoft.com/en-us/download/details.aspx?id=49926>

- 是否成功？可检查下面几个环境变量



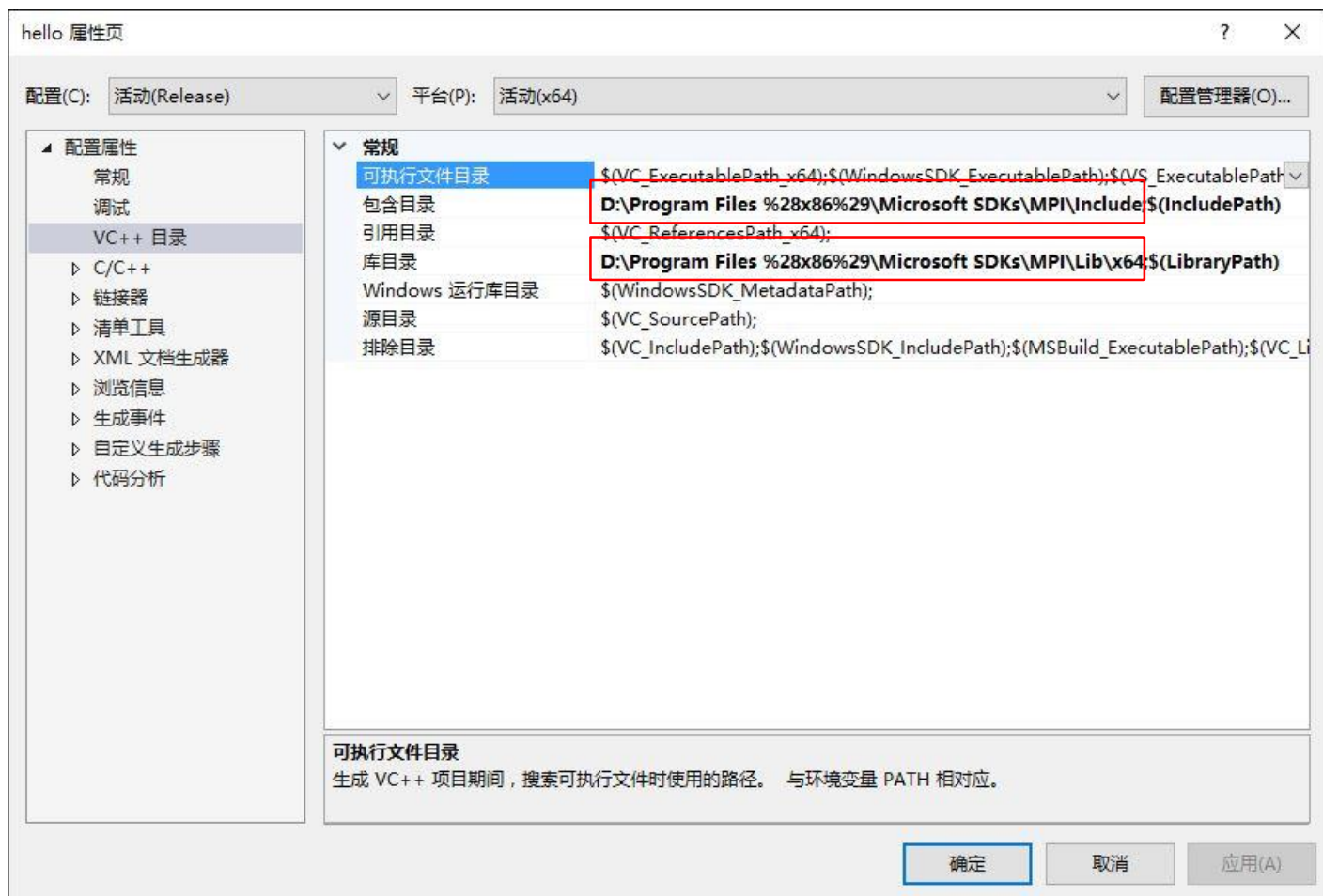
```
Administrator: Command Prompt
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

C:\Windows\system32>set MSMPI
MSMPI_BIN=C:\Program Files\Microsoft MPI\Bin\
MSMPI_INC=C:\Program Files (x86)\Microsoft SDKs\MPI\Include\
MSMPI_LIB32=C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x86\
MSMPI_LIB64=C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64\

C:\Windows\system32>
```

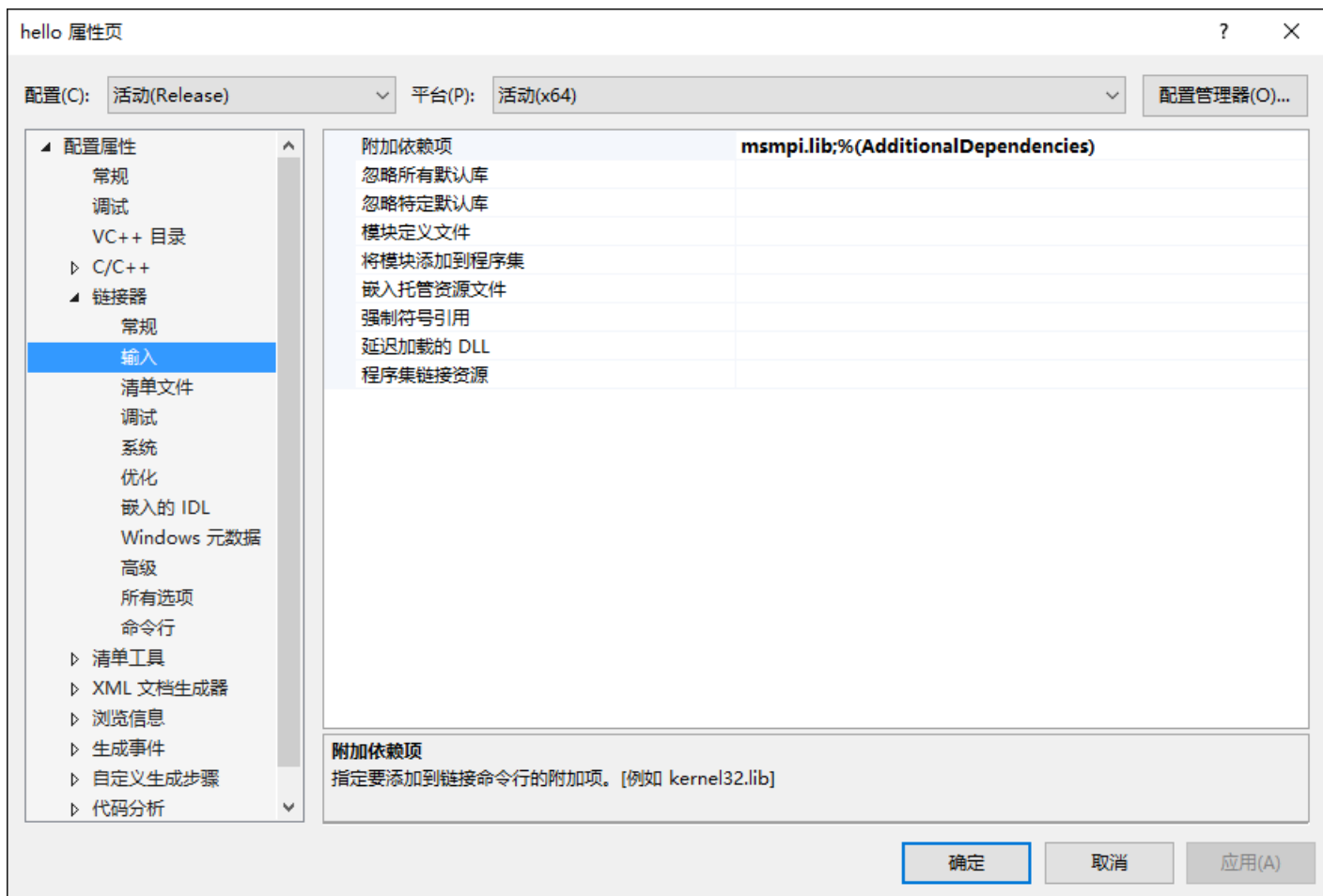
VS 设置

○ 头文件目录和库目录



VS设置 (2)

○ 链接库





运行MPI程序

- 后台运行MPI守护进程

`start /b smpd -d`

- 运行MPI程序

`mpiexec -hosts 1 localhost 4 .\hello.exe`



MPI初始化和结束处理

○ MPI_Init

- 令MPI进行必要的初始化工作

```
int MPI_Init(  
    int* argc_p /* 输入/输出参数 */,  
    char *** argv_p /* 输入/输出参数 */);
```

○ MPI_Finalize

- 告诉MPI程序已结束，进行清理工作

```
int MPI_Finalize(void);
```

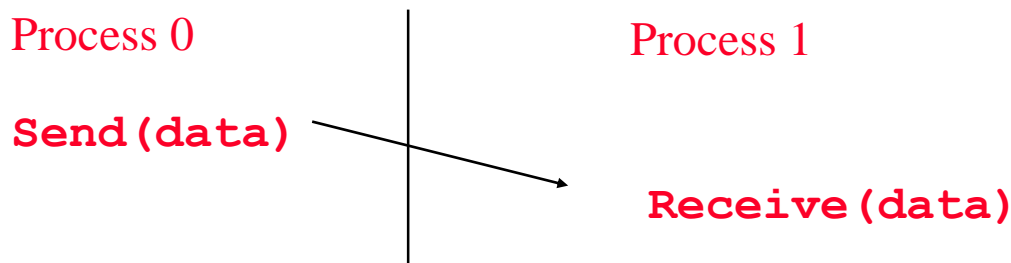


MPI程序基本结构

```
...  
#include <mpi.h>  
  
...  
int main(int argc, char *argv[]) {  
    ...  
    /* 这部分不应有MPI函数调用 */  
    MPI_Init(&argc, &argv);  
  
    ... /* MPI程序主体 */  
  
    MPI_Finalize();  
    /* 这部分不应有MPI函数调用 */  
    return 0;  
}
```

MPI消息传递

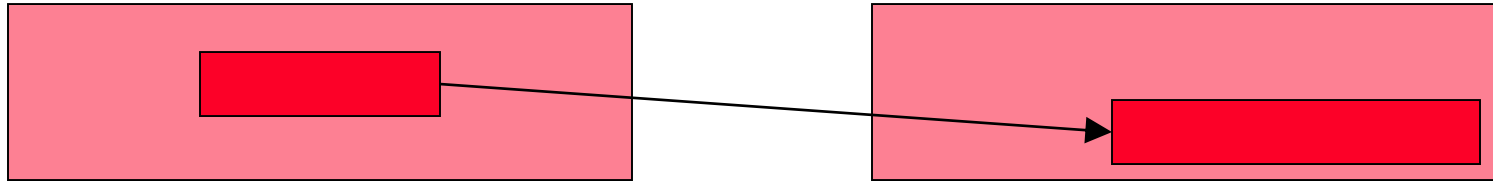
○ 基本的消息发送接收



○ 需要明确:

- 如何描述“数据”？
- 如何标识进程？
- 接收方如何识别消息？
- 操作完成意味着什么？

MPI基本（阻塞）发送



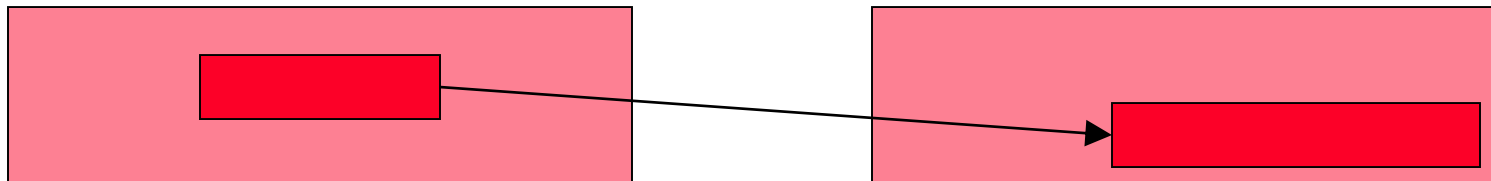
`MPI_Send(A, 10, MPI_DOUBLE, 1, ...)`

`MPI_Recv(B, 20, MPI_DOUBLE, 0, ...)`

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype,  
            int dest, int tag, MPI_Comm comm)
```

- 消息缓冲区(buf, count, datatype)
- 目的进程dest——目的进程在comm指定的通信域中的编号
- 阻塞发送——函数返回时，数据已经转给系统进行发送，缓冲区可作他用；消息可能还未送达目的进程

MPI基本（阻塞）接收



`MPI_Send(A, 10, MPI_DOUBLE, 1, ...)`

`MPI_Recv(B, 20, MPI_DOUBLE, 0, ...)`

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype,  
             int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- 阻塞接收——等待，直至收到匹配的（source和tag都相同）的消息，缓冲可作他用
- source可以是comm中的编号，或**MPI_ANY_SOURCE**
- tag为特定标签（需匹配）或**MPI_ANY_TAG**
- 接收的数据量比指定的少是允许的，但接收到更多数据就是错误
- **status**包含更多信息（如接收到的消息大小）



一些基本概念

- 如何组织进程
 - 进程可组成进程组
 - 每个消息都是在一个特定上下文中发送，必须在同一个上下文中接收
 - 进程组和上下文一起形成了通信域(communicator)
 - 进程用它在进程组（与某个通信域关联）中的编号标识
- **MPI_COMM_WORLD**: 默认通信域，其进程组包含所有初始进程



MPI数据类型

- MPI是强类型数据传输，区别于底层网络传输
- 消息中的数据用三元组(地址, 个数, 类型)描述
- MPI数据类型包括（递归定义）：
 - 对应高级语言类型的预定义类型（如MPI_INT、MPI_DOUBLE）
 - MPI数据类型的连续数组
 - 数据类型的跨越式块
 - 数据类型块的索引数组
 - 数据类型的任意结构
- MPI提供了自定义类型的函数

MPI预定义基本类型

MPI预定义数据类型与C数据类型的对应关系

MPI预定义数据类型	相应的C数据类型
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	无对应类型
MPI_PACKED	无对应类型



类型匹配

○ 含义

- 宿主语言的类型和消息所指定的类型相匹配
 - 例外: **MPI_BYTE**、**MPI_PACKED**
- 发送方和接收方的类型相匹配

○ 规则

- 有类型数据的通信，发送方和接收方均使用相同的数据类型
- 无类型数据的通信，发送方和接收方均以 **MPI_BYTE** 作为数据类型
- 打包数据的通信，发送方和接收方均使用 **MPI_PACKED**



MPI Tags

- 发送的消息都伴随一个用户定义的整数标签，帮助接收进程识别消息
- 接收方通过指定特定标签来筛选消息，或者指定**MPI_ANY_TAG**表示不筛选
- 其他一些消息传递平台也将tag称为“消息类型”。MPI称为tag，避免与数据类型混淆



MPI_STATUS

- 在C实现中，status至少是三个域组成的结构类型
 - status.MPI_SOURCE(MPI_ANY_SOURCE)
 - status.MPI_TAG(MPI_ANY_TAG)
 - status.MPI_ERROR
- 对status执行MPI_GET_COUNT调用还可以得到接收到的消息的长度信息



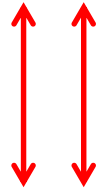
简单通信程序

```
#include<mpi.h>
#include<stdio.h>
#include<math.h>
int main(int argc,char *argv[])
{
    int myid, namelen;
    char message[20];
    MPI_Status status;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Get_processor_name(processor_name,&namelen);
```



简单通信程序(2)

```
if(myid == 0) {
    strcpy(message," Hello process 1");
    printf("process 0 on %s send: %s\n",
        processor_name, message);
    MPI_Send(message, 20, MPI_CHAR, 1, 99,
        MPI_COMM_WORLD);
}
else if (myid == 1) {
    MPI_Recv(message, 20, MPI_CHAR, 0, 99,
        MPI_COMM_WORLD, &status);
    printf("process 1 on %s received: %s\n",
        processor_name, message);
}
MPI_Finalize();
return 0;
}
```





简单通信程序(3)

○ 一个节点两个核心上的运行结果：

process 0 on USER-CULKTGAUF1 send: Hello process 1

process 1 on USER-CULKTGAUF1 received: Hello process 1



ANY_SOURCE & ANY_TAG

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, size, i, buf[1];
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
```



ANY_SOURCE & ANY_TAG(2)

```
if (rank == 0) {  
    for (i=0; i<(size-1); i++) {  
        MPI_Recv( buf, 1, MPI_INT, MPI_ANY_SOURCE,  
                  MPI_ANY_TAG, MPI_COMM_WORLD, &status );  
        printf( "Msg=%d from %d with tag %d\n",  
                buf[0], status.MPI_SOURCE, status.MPI_TAG );  
    }  
}  
else {  
    buf[0]=rank;  
    MPI_Send( buf, 1, MPI_INT, 0, 100-rank, MPI_COMM_WORLD );  
}  
MPI_Finalize();  
return 0;  
}
```



ANY_SOURCE & ANY_TAG(3)

○ 一个节点四个核心上的运行结果：

Msg=1 from 1 with tag 99

Msg=3 from 3 with tag 97

Msg=2 from 2 with tag 98

并行排序

Rank 0



$O(N \log N)$

Rank 0



Rank 1



$O(N/2 \log N/2)$

Rank 0



$O(N)$

Rank 0





并行排序使用Send/Recv

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char ** argv)
{
    int rank, a[1000], b[500];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        MPI_Send(&a[500], 500, MPI_INT, 1, 0, MPI_COMM_WORLD);
        sort(a, 500);
        MPI_Recv(b, 500, MPI_INT, 1, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);

        /* Serial: Merge array b and sorted part of array a */
    }
```

Introduction to MPI, Argonne (06/06/2014)



并行排序使用Send/Recv

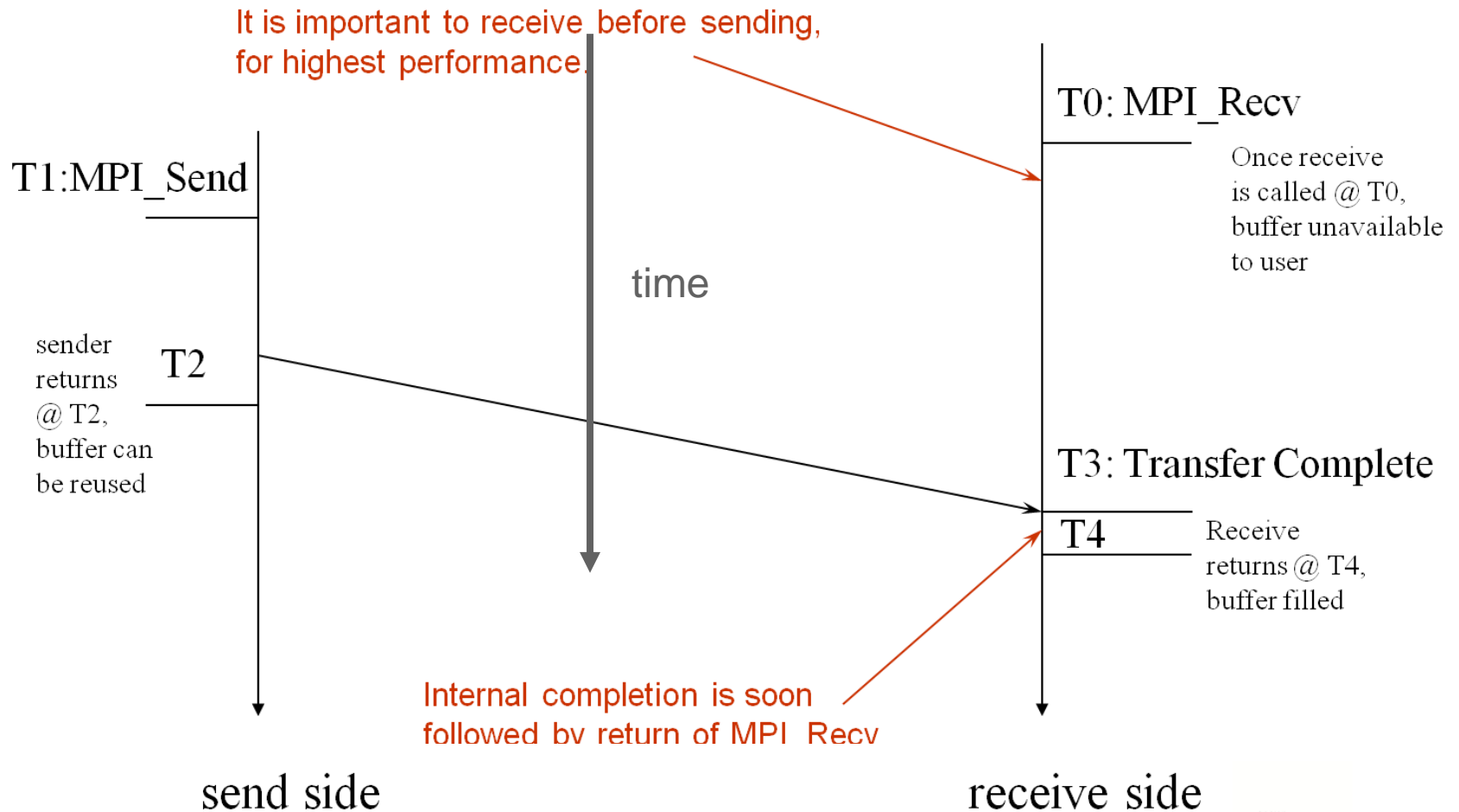
```
else if (rank == 1) {  
    MPI_Recv(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD,  
            MPI_STATUS_IGNORE);  
  
    sort(b, 500);  
    MPI_Send(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD);  
}  
  
MPI_Finalize(); return 0;  
}
```



提纲

- MPI概念和基本原语
- MPI编程模型
- 点对点通信进阶
- 组通信
- 自定义数据类型
- 进程组与通信域
- 虚拟拓扑
- MPI新标准

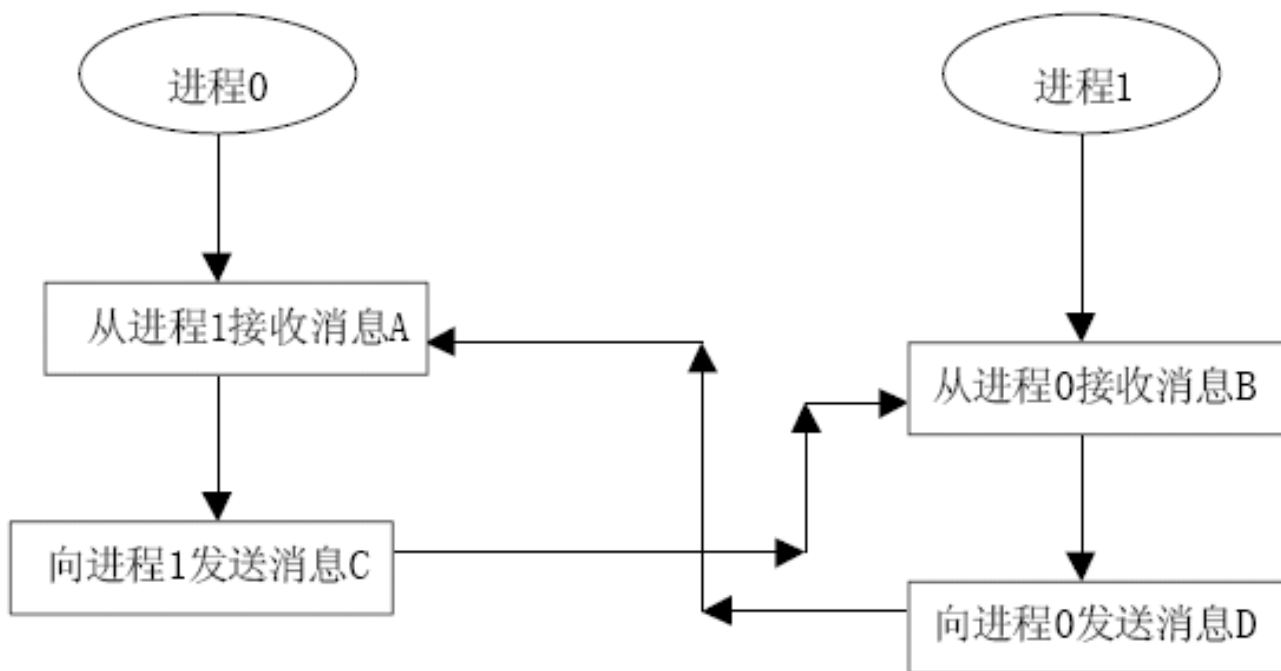
阻塞通信模式



39

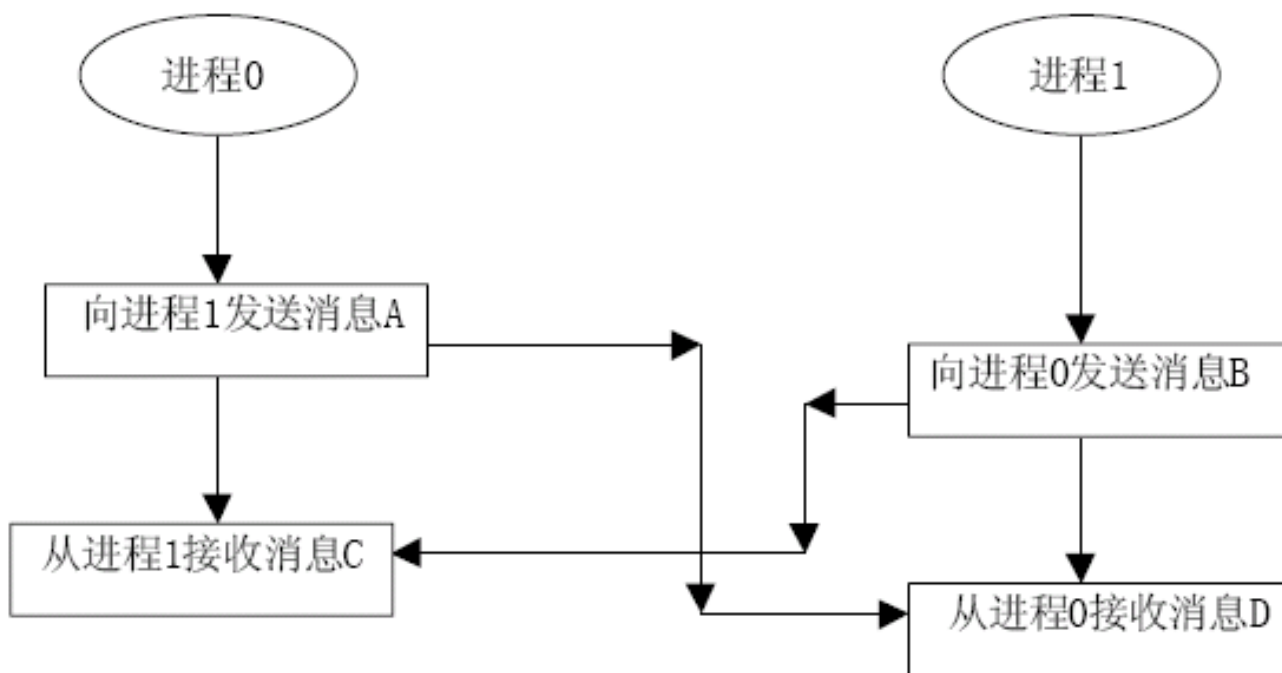
通信模式——避免死锁

○ 必然死锁的通信模式

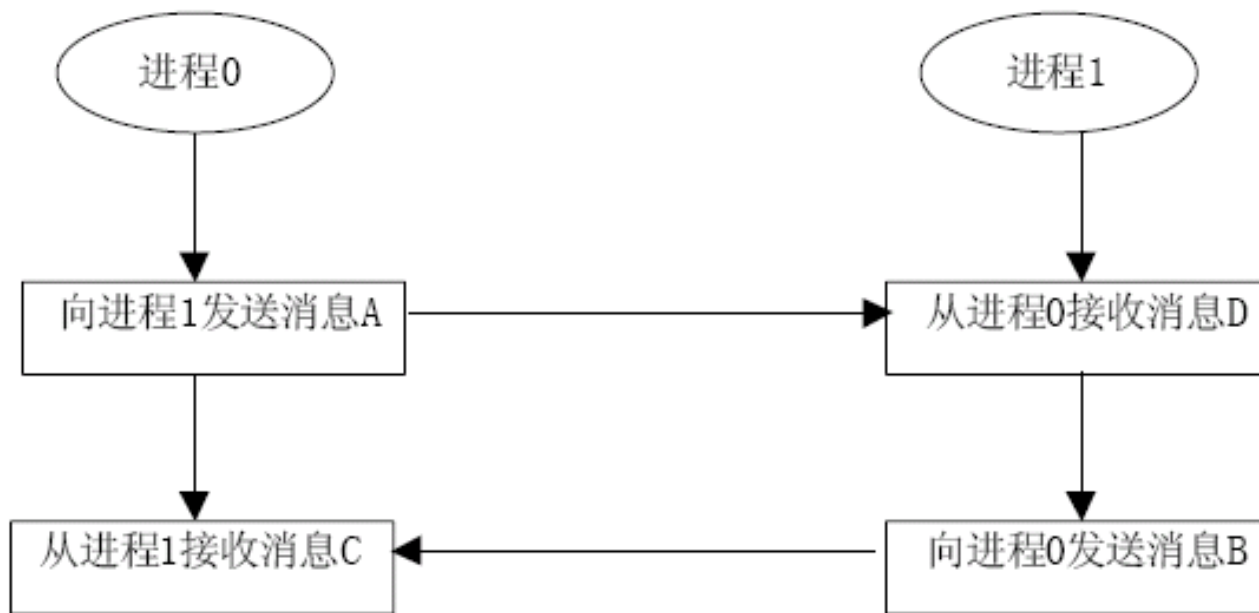


不安全的通信模式

○ 缓冲区问题可能导致死锁



安全的通信模式



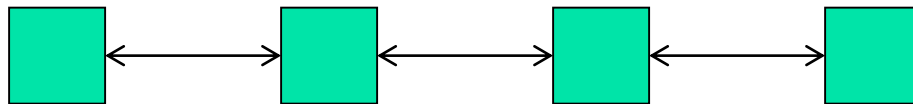


经验：推迟同步

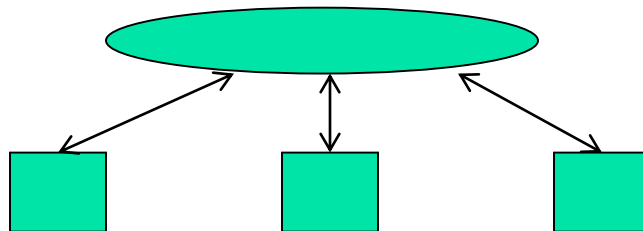
- Send-receive其实完成了两件事
 - 数据传输
 - 同步
- 很多情况下，我们不需要那么多同步
- 使用非阻塞通信操作和MPI_Waitall来推迟同步
- 一些可甄别性能问题的工具
 - MPE, Tau, HPCToolkit是流行的Profiling工具
 - Jumpshot可图示性能问题

消息传递两种编程模型

- 对等式（地位平等，功能相近）



- 主从式（地位不同，功能不同）

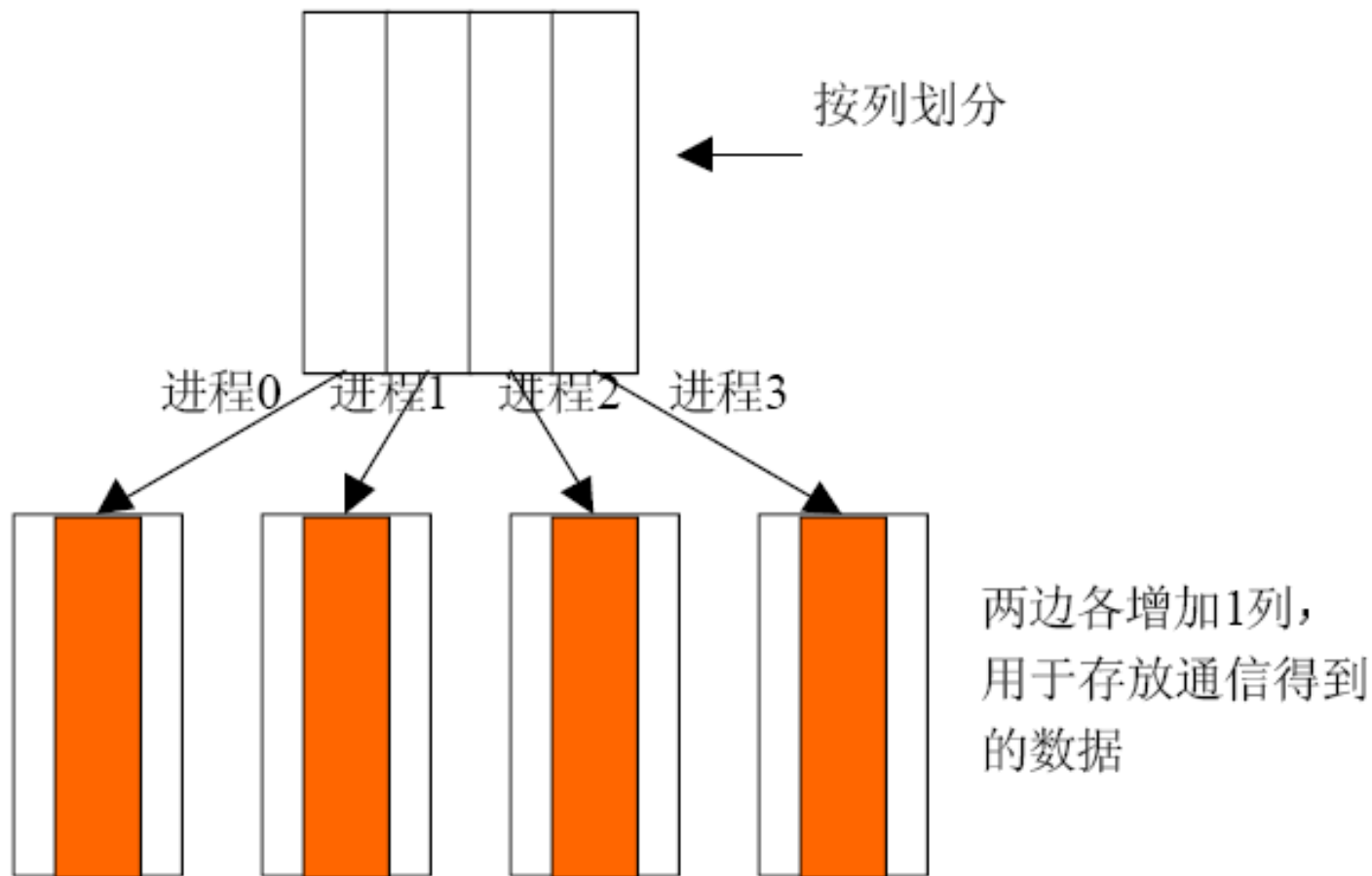




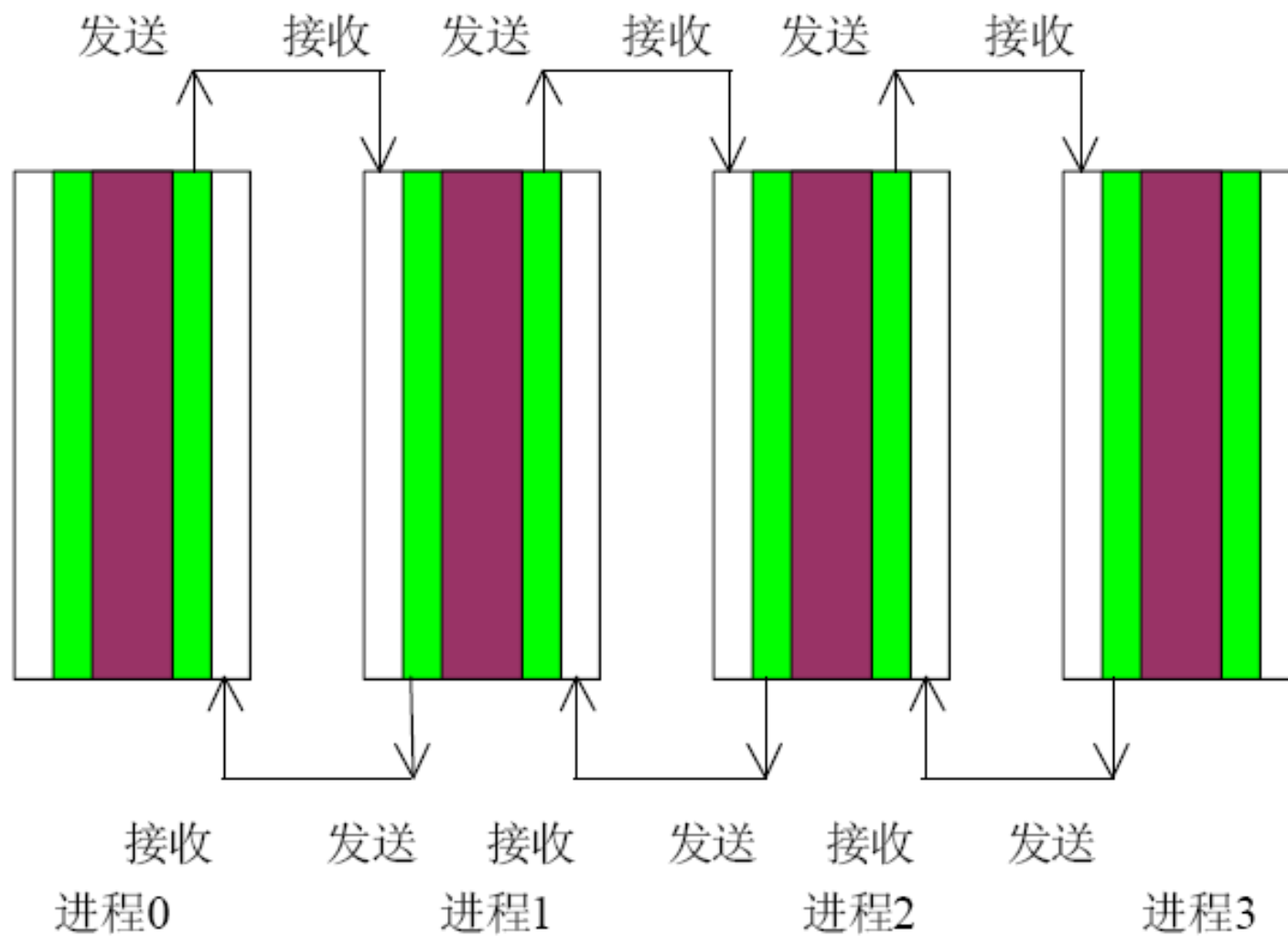
Jacobi迭代

```
for (int k = 0; k < steps; k+=2) {  
    for (int i = 1; i <= n; i++)  
        for (int j = 1; j <= n; j++)  
            B[i][j] = 0.25*(A[i - 1][j] + A[i + 1][j] + A[i][j - 1] + A[i][j + 1]);  
  
    for (int i = 1; i <= n; i++)  
        for (int j = 1; j <= n; j++)  
            A[i][j] = 0.25*(B[i - 1][j] + B[i + 1][j] + B[i][j - 1] + B[i][j + 1]);  
}
```

对等模式Jacobi迭代数据并行



通信方式



实现（C语言行划分）

```
for (int k = 0; k < steps; k += 2) {  
    if (rank < size - 1)  
        MPI_Recv(&A[my_n + 1], n, MPI_FLOAT, rank + 1, 10,  
                MPI_COMM_WORLD, &status);  
    if (rank > 0)  
        MPI_Send(&A[1], n, MPI_FLOAT, rank - 1, 10,  
                MPI_COMM_WORLD);  
    if (rank < size - 1)  
        MPI_Send(&A[my_n], n, MPI_FLOAT, rank + 1, 9,  
                MPI_COMM_WORLD);  
    if (rank > 0)  
        MPI_Recv(&A[0], n, MPI_FLOAT, rank - 1, 9,  
                MPI_COMM_WORLD, &status);
```

从下收

向上发

向下发

从上收



实现(2)

```
for (int i = 1; i <= my_n; i++) ← 各自计算n/size行
    for (int j = 1; j <= n; j++)
        B[i][j] = 0.25*(A[i - 1][j] + A[i + 1][j] + A[i][j - 1] + A[i][j + 1]);
    if (rank < size - 1)
        MPI_Recv(&B[my_n + 1], n, MPI_FLOAT, rank + 1, 10,
                 MPI_COMM_WORLD, &status);
    if (rank > 0)
        MPI_Send(&B[1], n, MPI_FLOAT, rank - 1, 10, MPI_COMM_WORLD);
    if (rank < size - 1)
        MPI_Send(&B[my_n], n, MPI_FLOAT, rank + 1, 9,
                 MPI_COMM_WORLD);
    if (rank > 0)
        MPI_Recv(&B[0], n, MPI_FLOAT, rank - 1, 9, MPI_COMM_WORLD,
                 &status);

    for (int i = 1; i <= my_n; i++)
        for (int j = 1; j <= n; j++)
            A[i][j] = 0.25*(B[i - 1][j] + B[i + 1][j] + B[i][j - 1] + B[i][j + 1]);
}
```

实现(3)——初始化和结果收集

```
if (rank == 0) {  
    ...  
    init(N);  
    QueryPerformanceCounter((LARGE_INTEGER *)&head);  
    my_n = N / size;  
    for (int i = 1; i < size; i++)  
        MPI_Send(&A[i*my_n + 1], N*my_n, MPI_FLOAT, i, 0,  
                 MPI_COMM_WORLD);  
    mpi_jacobi(rank, size, N, my_n, steps);  
    for (int i = 1; i < size; i++)  
        MPI_Recv(&A[i*my_n + 1], N*my_n, MPI_FLOAT, i, 1,  
                 MPI_COMM_WORLD, &status);  
    QueryPerformanceCounter((LARGE_INTEGER *)&tail);  
    cout << "MPI: " << (tail - head) * 1000.0 / freq << "ms" << endl;  
}
```

向从进程发送初始数据

接收迭代结果



实现(4)——初始化和结果收集

```
else {  
    my_n = N / size;  
    MPI_Recv(&A[1], N*my_n, MPI_FLOAT, 0, 0,  
            MPI_COMM_WORLD, &status);  
    mpi_jacobi(rank, size, N, my_n, steps);  
    MPI_Send(&A[1], N*my_n, MPI_FLOAT, 0, 1,  
            MPI_COMM_WORLD);  
}
```

从主进程接收初始数据

发送迭代结果



运行结果

`mpiexec -hosts 1 localhost 4 .\jacobi.exe 1000`

Sequential: 867.93ms

MPI: 304.379ms



简化通信—SENDRECV

`MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)`

IN sendbuf 发送缓冲区起始地址(可选数据类型)

IN sendcount 发送数据的个数(整型)

IN sendtype 发送数据的数据类型(句柄)

IN dest 目标进程标识(整型)

IN sendtag 发送消息标识(整型)

OUT recvbuf 接收缓冲区初始地址(可选数据类型)

IN recvcount 最大接收数据个数(整型)

IN recvtype 接收数据的数据类型(句柄)

IN source 源进程标识(整型)

IN recvtag 接收消息标识(整型)

IN comm 通信域(句柄)

OUT status 返回的状态(status)



SENDRECV_REPLACE

MPI_SENDRECV_REPLACE(buf,count,datatype,dest,sendtag,source,recvtag,comm, status)

INOUT buf 发送和接收缓冲区初始地址(可选数据类型)

IN count 发送和接收缓冲区中的数据个数(整型)

IN datatype 发送和接收缓冲区中数据的数据类型(句柄)

IN dest 目标进程标识(整型)

IN sendtag 发送消息标识(整型)

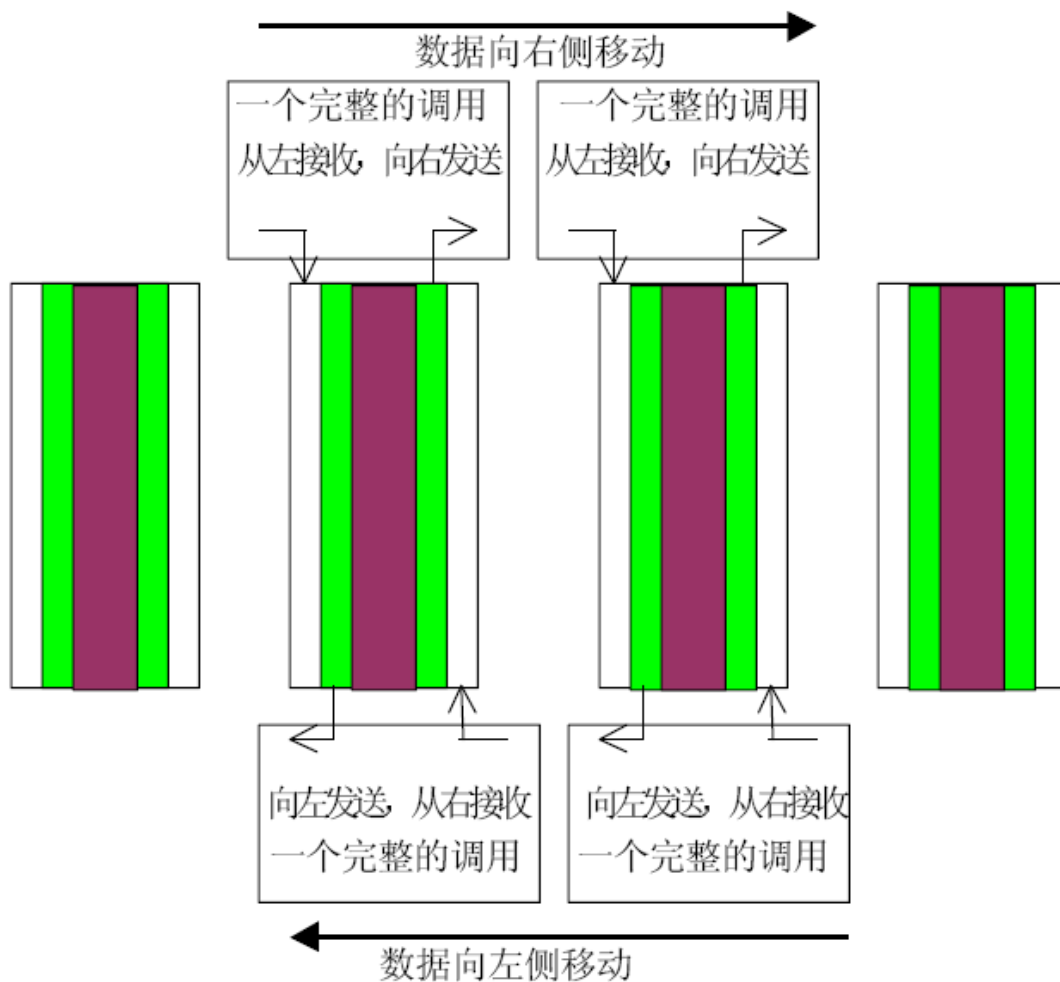
IN source 源进程标识(整型)

IN recvtag 接收消息标识(整型)

IN comm 发送进程和接收进程所在的通信域(句柄)

OUT status 状态目标(status)

Jacobi迭代程序改进





SENDRECV版本

```
for (int k = 0; k < steps; k += 2) {  
    if (rank == 0) // left to right  
        MPI_Send(&A[my_n], n, MPI_FLOAT, rank + 1, 9,  
MPI_COMM_WORLD);  
        else if (rank == size - 1) ← 边界进程特殊处理  
            MPI_Recv(&A[0], n, MPI_FLOAT, rank - 1, 9, MPI_COMM_WORLD,  
&status);  
        else MPI_Sendrecv(&A[my_n], n, MPI_FLOAT, rank + 1, 9, &A[0], n,  
MPI_FLOAT, rank - 1, 9, MPI_COMM_WORLD, &status);  
  
    if (rank == 0) // right to left  
        MPI_Recv(&A[my_n + 1], n, MPI_FLOAT, rank + 1, 10,  
MPI_COMM_WORLD, &status); ← 边界进程特殊处理  
        else if (rank == size - 1)  
            MPI_Send(&A[1], n, MPI_FLOAT, rank - 1, 10, MPI_COMM_WORLD);  
        else MPI_Sendrecv(&A[1], n, MPI_FLOAT, rank - 1, 10, &A[my_n + 1], n,  
MPI_FLOAT, rank + 1, 10, MPI_COMM_WORLD, &status);  
    ...  
}
```



SENDRECV版本(2)

```
if (rank == 0) // left to right
    MPI_Send(&B[my_n], n, MPI_FLOAT, rank + 1, 9,
MPI_COMM_WORLD);
    else if (rank == size - 1)
        MPI_Recv(&B[0], n, MPI_FLOAT, rank - 1, 9, MPI_COMM_WORLD,
&status);
    else MPI_Sendrecv(&B[my_n], n, MPI_FLOAT, rank + 1, 9, &B[0], n,
MPI_FLOAT, rank - 1, 9, MPI_COMM_WORLD, &status);

if (rank == 0) // right to left
    MPI_Recv(&B[my_n + 1], n, MPI_FLOAT, rank + 1, 10,
MPI_COMM_WORLD, &status);
    else if (rank == size - 1)
        MPI_Send(&B[1], n, MPI_FLOAT, rank - 1, 10, MPI_COMM_WORLD);
    else MPI_Sendrecv(&B[1], n, MPI_FLOAT, rank - 1, 10, &B[my_n + 1], n,
MPI_FLOAT, rank + 1, 10, MPI_COMM_WORLD, &status);
...
}
}
```



引入虚拟进程

```
if (rank > 0)
    left = rank - 1;
else left = MPI_PROC_NULL;
if (rank < size - 1)
    right = rank + 1;
else right = MPI_PROC_NULL
```

类比Linux系统中的/dev/null
像一个黑洞





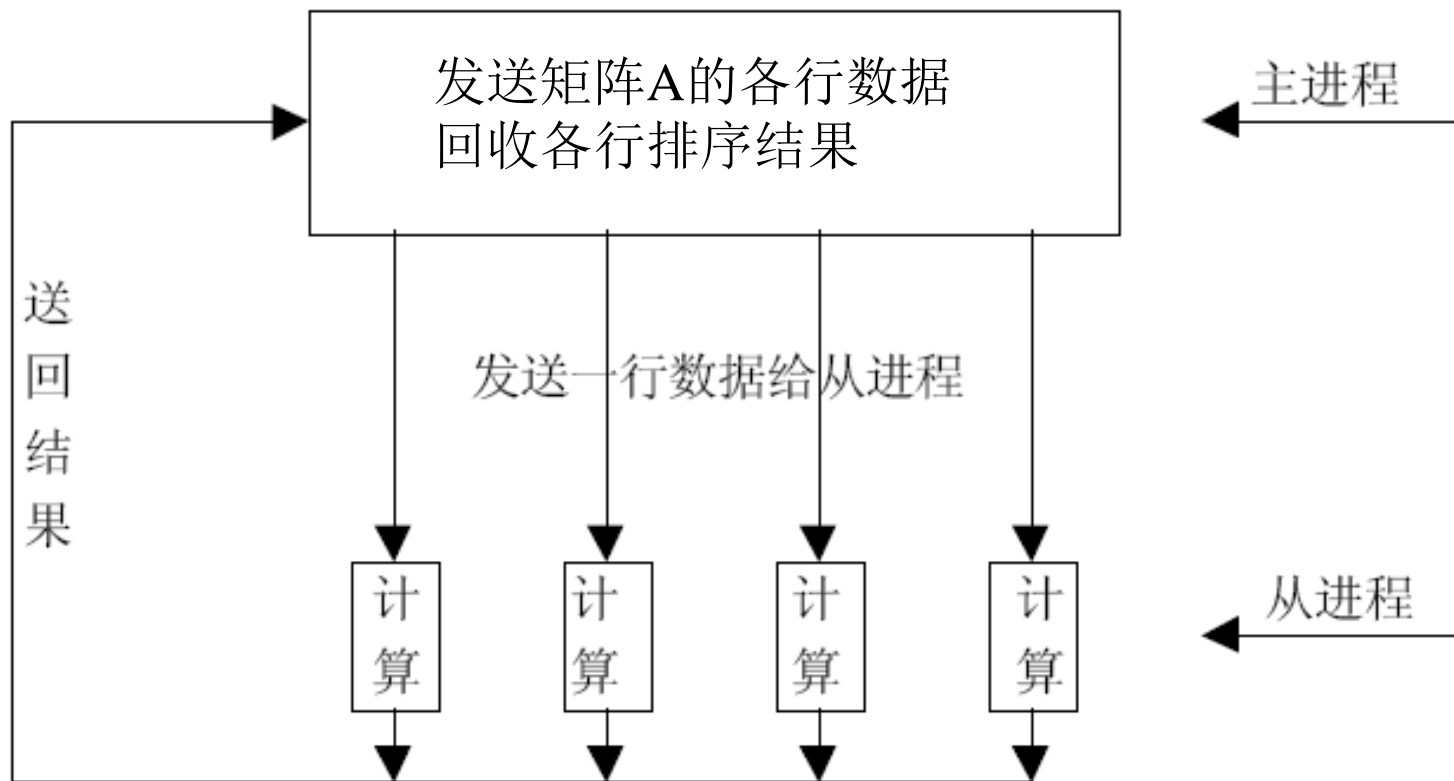
引入虚拟进程（2）

```
for (int k = 0; k < steps; k += 2) {  
    // left to right  
    MPI_Sendrecv(&A[my_n], n, MPI_FLOAT, right, 9, &A[0], n,  
MPI_FLOAT, left, 9, MPI_COMM_WORLD, &status);  
  
    // right to left  
    MPI_Sendrecv(&A[1], n, MPI_FLOAT, left, 10, &A[my_n + 1], n,  
MPI_FLOAT, right, 10, MPI_COMM_WORLD, &status);  
    ...  
}
```

边界进程不再需要特殊处理

主从模式MPI程序

○ 矩阵行排序



MPI动态任务分配

```
void mpi_sort(void)
{
    MPI_Status status;

    while (1) {
        MPI_Recv(&B[0][0], g * ARR_LEN, MPI_FLOAT, 0, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
        if (status.MPI_TAG >= ARR_NUM)
            return;
        for (int i = 0; i < g; i++)
            stable_sort(B[i].begin(), B[i].end());
        MPI_Send(&B[0][0], g * ARR_LEN, MPI_FLOAT, 0, status.MPI_TAG,
MPI_COMM_WORLD);
    }
}
```

接收任务（g行）

任务已全部完成

发送排序结果

MPI动态任务分配 (2)

```
if (rank == 0) {  
    int i, finished = 1;  
    for (i = 1; i < size; i++)  
        MPI_Send(&B[i * g][0], ARR_LEN * g, MPI_FLOAT, i, i * g,  
MPI_COMM_WORLD);  
    while (finished < size) {  
        MPI_Recv(&C[0][0], ARR_LEN * g, MPI_FLOAT, MPI_ANY_SOURCE,  
MPI_ANY_TAG, MPI_COMM_WORLD, &status);  
        memcpy(&B[status.MPI_TAG][0], &C[0][0], sizeof(float) * ARR_LEN *  
g);  
        if (i * g < ARR_NUM) {  
            MPI_Send(&B[i * g][0], ARR_LEN * g, MPI_FLOAT,  
status.MPI_SOURCE, i * g, MPI_COMM_WORLD);  
            i++;  
        }  
        else {  
            MPI_Send(&B[0][0], ARR_LEN * g, MPI_FLOAT,  
status.MPI_SOURCE, ARR_NUM+1, MPI_COMM_WORLD);  
            finished++;  
        }  
    }  
}
```

为每个进程发送一个任务，行号作为tag

接收结果，不能指定源进程号

若还有任务，发送给刚才的进程

任务都已完成



MPI动态任务分配 (3)

- 单机3线程不同粒度:

`mpiexec -hosts 1 localhost 3 .\master_slave.exe 1`

Sequential: 1387ms

MPI: 801.637ms

`mpiexec -hosts 1 localhost 3 .\master_slave.exe 10`

Sequential: 1375.58ms

MPI: 775.564ms

`mpiexec -hosts 1 localhost 3 .\master_slave.exe 50`

Sequential: 1382.78ms

MPI: 761.561ms



MPI动态任务分配（4）

- 不同线程数:

`mpiexec -hosts 1 localhost 3 .\master_slave.exe 50`

Sequential: 1382.78ms

MPI: 761.561ms

`mpiexec -hosts 1 localhost 4 .\master_slave.exe 50`

Sequential: 1446.71ms

MPI: 572.936ms

`mpiexec -hosts 1 localhost 5 .\master_slave.exe 50`

Sequential: 1443.42ms

MPI: 506.682ms

`mpiexec -hosts 1 localhost 6 .\master_slave.exe 50`

Sequential: 1480.65ms

MPI: 452.7ms



提纲

- MPI概念和基本原语
- MPI编程模型
- 组通信
- 自定义数据类型
- 进程组与通信域
- 虚拟拓扑
- MPI新标准



组通信

- Collective(Group) Communication
- 一个进程组（通信域）内的所有进程同时参加通信
- 所有参与进程的函数调用形式完全相同
- 哪些进程参加以及组通信的上下文都是由调用时指定的通信域限定的
- 在组通信中不需要通信消息标志参数
- 三个功能：通信、同步和计算
- 操作是成对的——互为逆操作



广播和归约

○ one-to-all broadcast

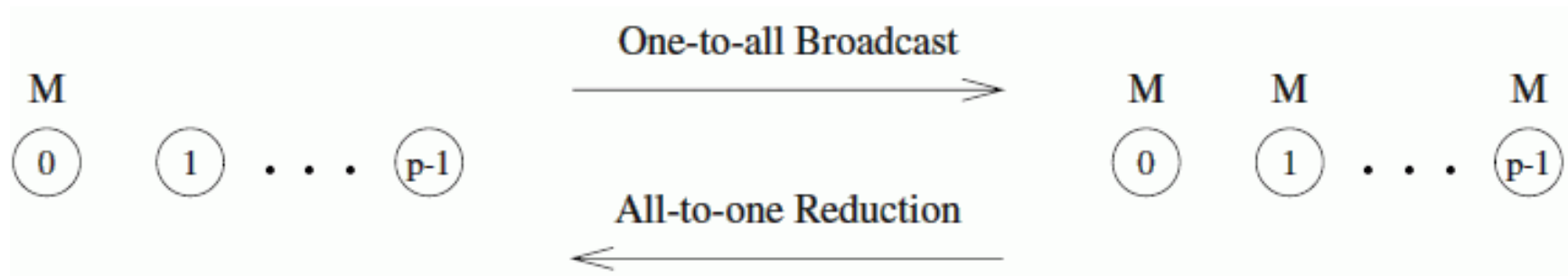
- 一个进程向其他所有进程发送相同数据
- 初始，只有源进程有一份 m 个字的数据
广播操作后，所有进程都有一份相同数据

○ 对应操作：all-to-one reduction

- 初始，每个进程都有一份 m 个字的数据
- 归约操作后， p 份数据经过计算（加、乘、...）
得到一份数据（结果），传送到目的进程

○ 应用：矩阵相乘、高斯消去、最短路径、向量内积...

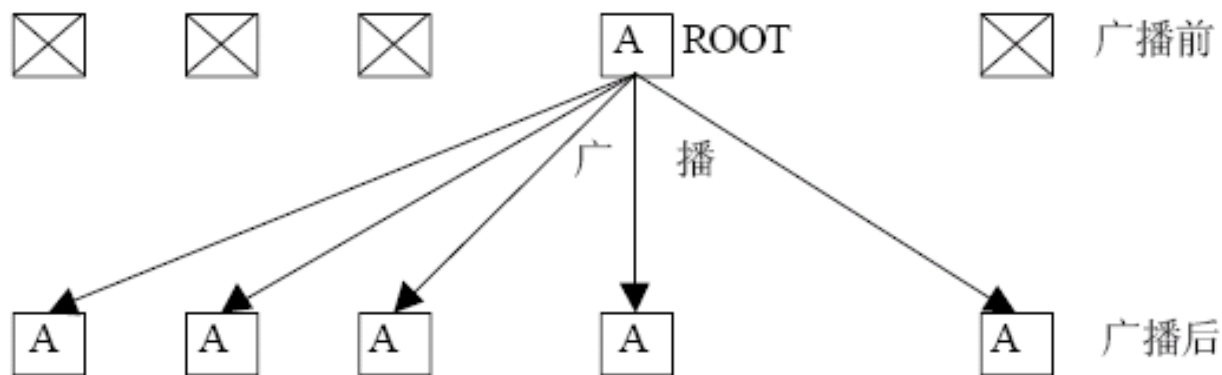
广播和归约效果



MPI广播

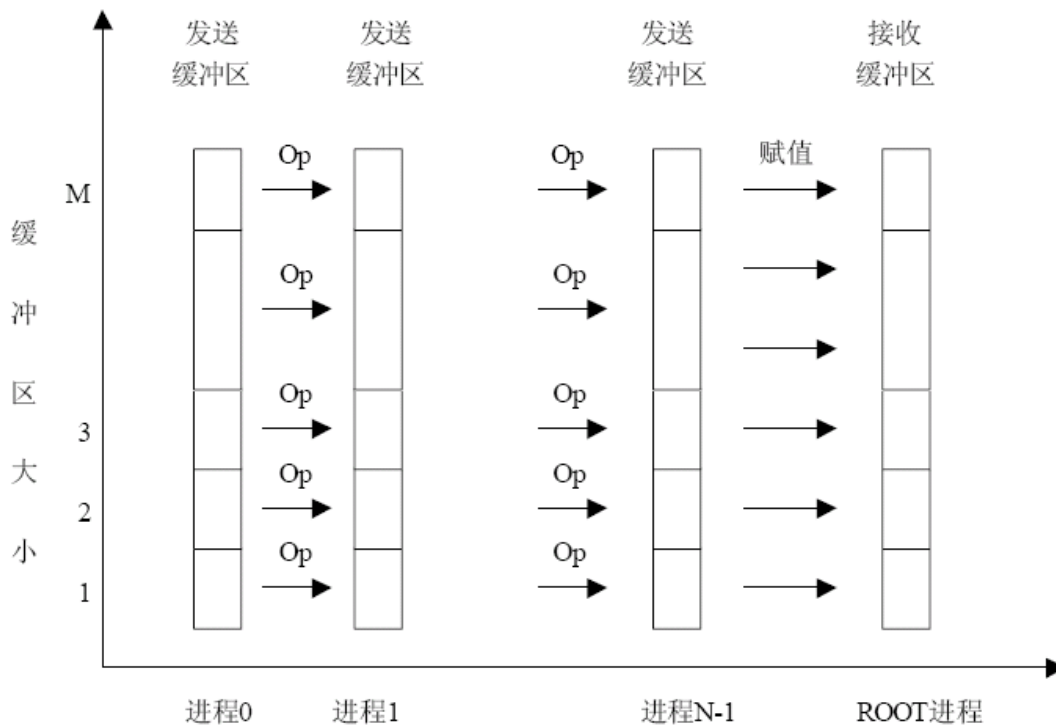
```
int MPI_Bcast(void* buffer,int count,MPI_Datatype  
              datatype,int root, MPI_Comm comm)
```

- root: 每个调用广播操作的进程中root都相同
- count: 待广播的数据的个数



MPI归约

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm
               comm)
```





MPI预定义归约操作

名字	含义
MPI_MAX	最大值
MPI_MIN	最小值
MPI_SUM	求和
MPI_PROD	求积
MPI_LAND	逻辑与
MPI_BAND	按位与
MPI_LOR	逻辑或
MPI_BOR	按位或
MPI_LXOR	逻辑异或
MPI_BXOR	按位异或
MPI_MAXLOC	最大值且相应位置
MPI_MINLOC	最小值且相应位置

操作	允许的数据类型
MPI_MAX, MPI_MIN	C整数, Fortran整数, 浮点数
MPI_SUM, MPI_PROD	C整数, Fortran整数, 浮点数, 复数
MPI_LAND, MPI_LOR, MPI_LXOR	C整数, 逻辑型
MPI_BAND, MPI_BOR, MPI_BXOR	C整数, Fortran整数, 字节型



环/线性阵列

○ 简单算法

- 源进程顺序将数据发送给其它 $p-1$ 个进程
- 低效，源进程成为瓶颈，网络利用不充分

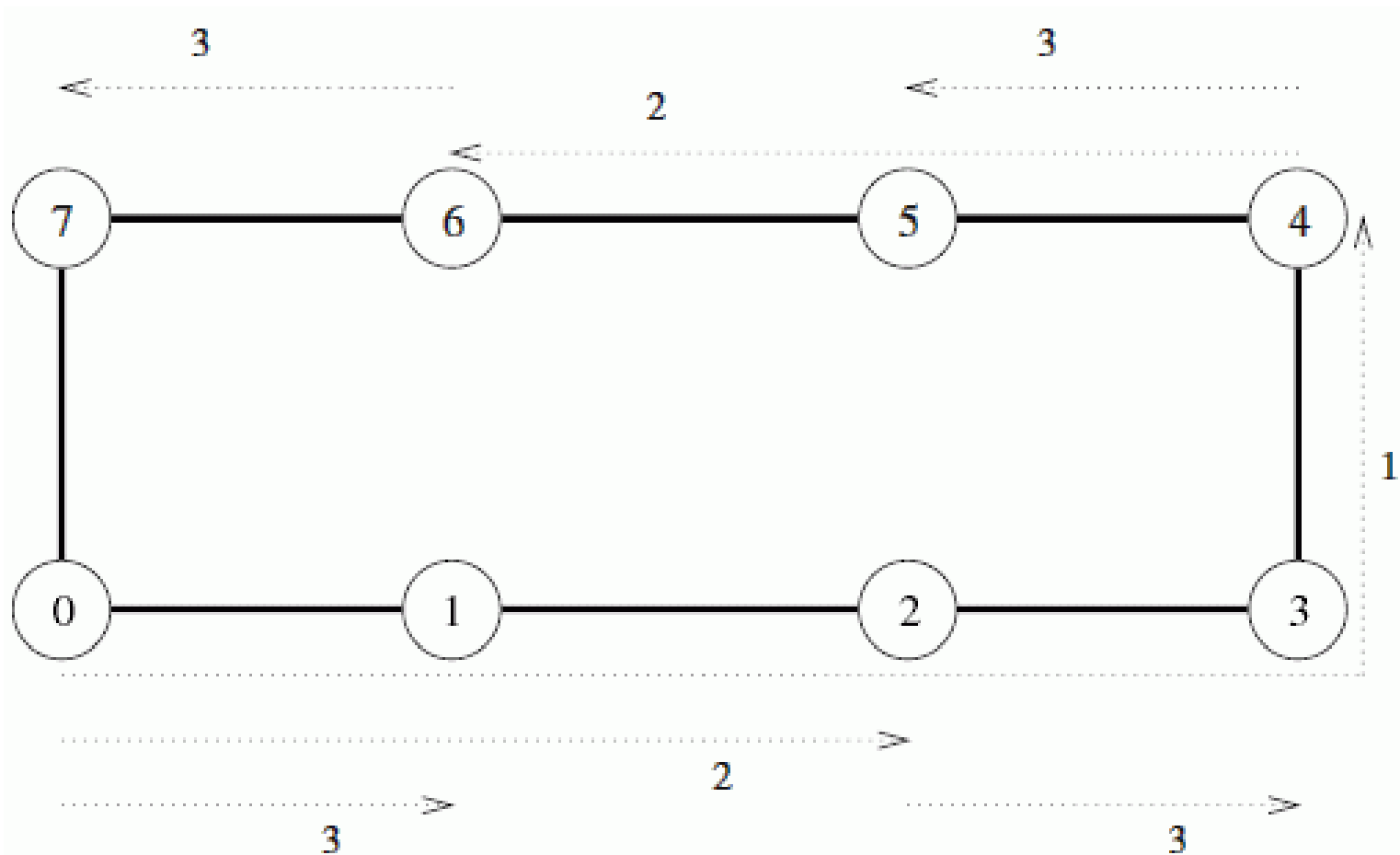
○ 递归加倍法

- 第一步：源进程 $P \rightarrow$ 进程 P'
- 第二步：进程 $P, P' \rightarrow$ 另外两个进程...

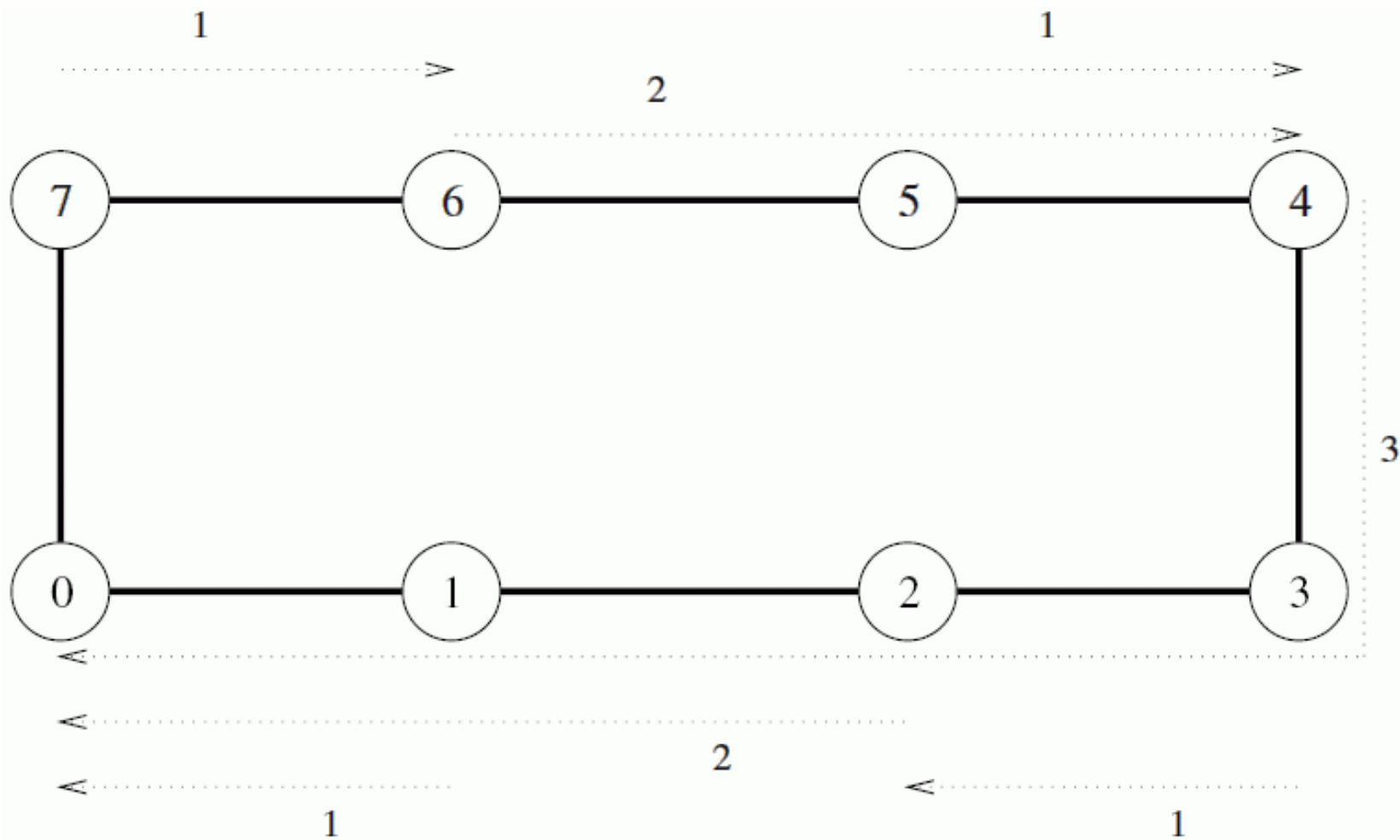
○ 避免冲突，环的实现

- 第一步：源进程 $P \rightarrow$ 距离最远 ($p/2$) 的进程
- 第二步：两个进程 \rightarrow 距离 $p/4$ 的进程...

广播实现——递归加倍法



多对一归约算法示意

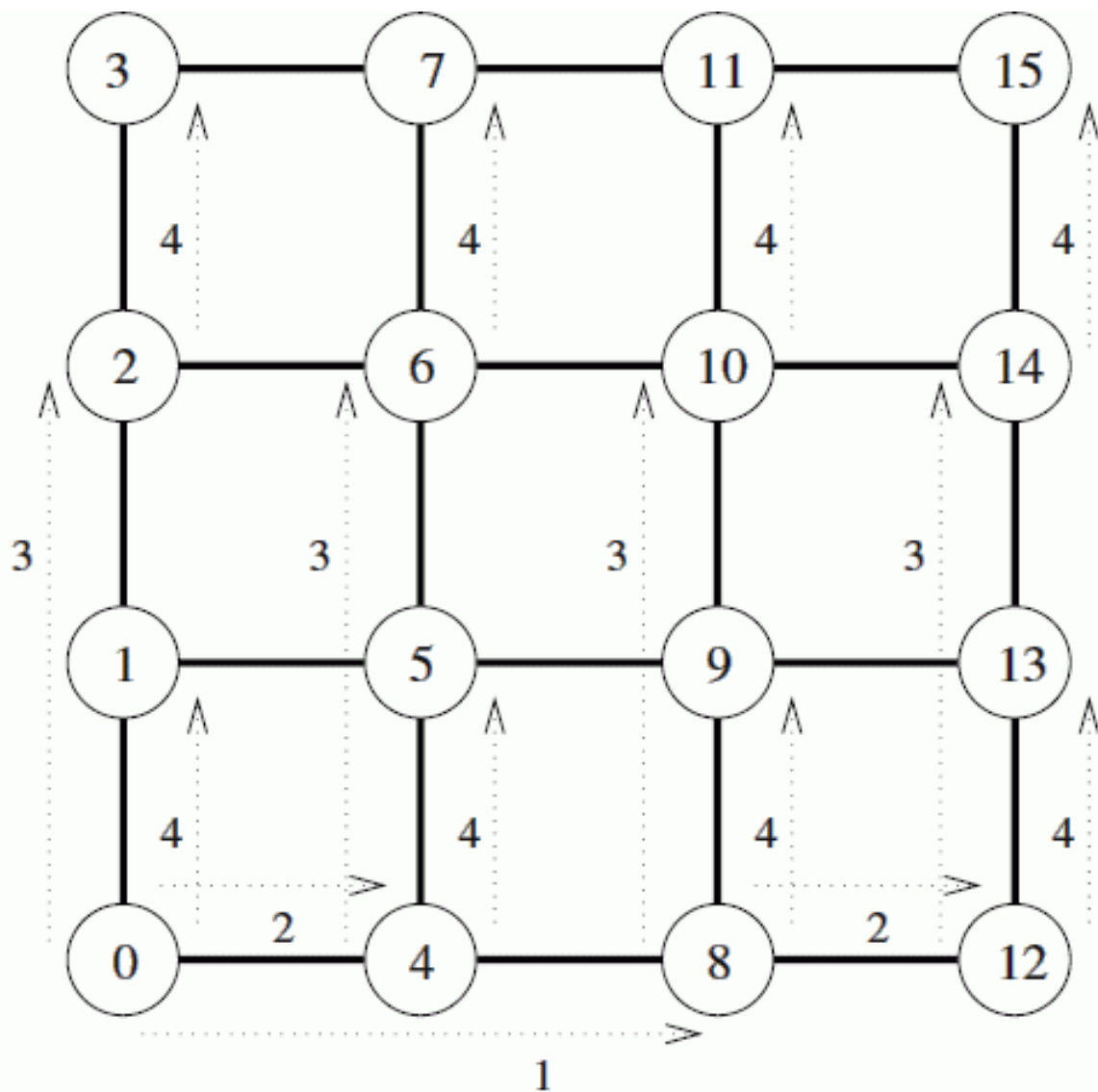




mesh

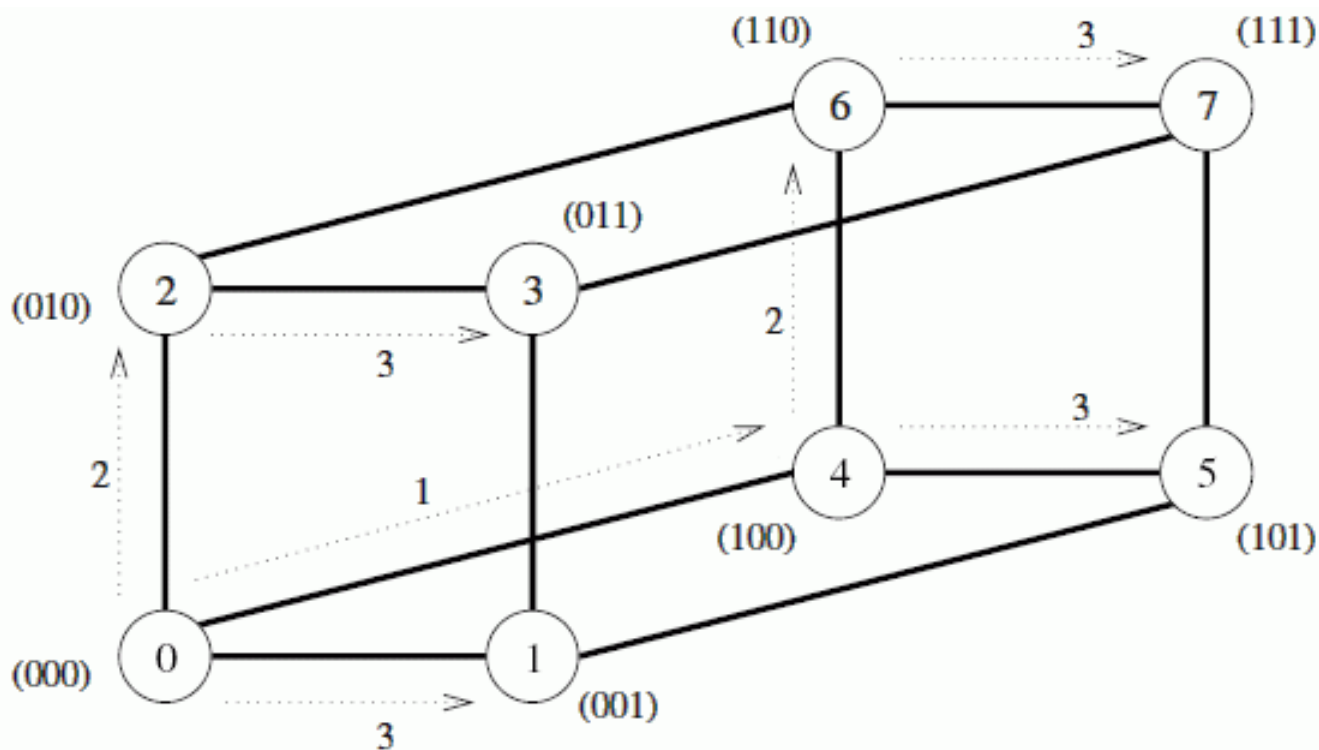
- 行——线性阵列，线性阵列算法的扩展
 - 第一步骤，行进行广播/归约
 - 第二步骤，列进行广播/归约
- 以广播为例
 1. 源进程→同一行其它 $\sqrt{p}-1$ 个进程广播
 2. 已有数据的 \sqrt{p} 个进程→同列节点广播
- 3维mesh广播
 - 每个维度 $p^{1/3}$ 个节点，线性阵列同样方法

mesh广播算法



超立方广播算法

- d维mesh, 每个维度2个节点
- 分为d个阶段, 每个阶段1个步骤



广播算法细节

○ 线性阵列、mesh、超立方上是一样的

○ 给出 2^d 超立方的算法——源节点0

1. **procedure** ONE_TO_ALL_BC($d, my\ id, X$)

2. **begin**

3. $mask := 2^d - 1;$

4. **for** $i := d - 1$ **downto** 0 **do**

5. $mask := mask \text{ XOR } 2^i;$

6. **if** ($my_id \text{ AND } mask$) = 0 **then**

7. **if** ($my_id \text{ AND } 2^i$) = 0 **then**

8. $msg_destination := my_id \text{ XOR } 2^i;$

9. **send** X to $msg_destination;$

第 i 步: 2^{d-i-1} 个进程 \rightarrow 另 2^{d-i-1} 个进程

$mask$ 值: $d-i$ 个0后接 i 个1

id 低 i 位全0 (与源相同) 的节点
(不同子立方同编号节点),
共 2^{d-i} 个, 参与本步骤通信

第 $i+1$ 位 (第 $i+1$ 维) 为0的节点
(共 2^{d-1-i} 个) 发送, 其他接收



超立方算法（续）

```
10.      else
11.          msg_source := my_id XOR  $2^i$ ;
12.          receive X from msg_source;
13.      endelse;
14.  endif;
15. endfor;
16. end ONE_TO_ALL_BC
```

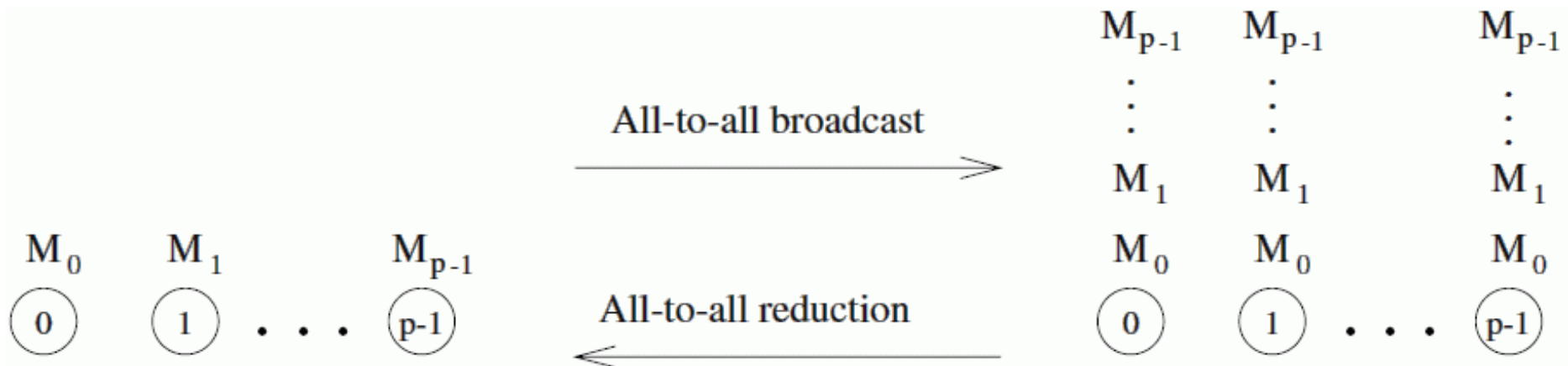
归约算法？

复杂性分析？

root不是0时怎么办？

All-to-All广播和归约

- All-to-All广播：所有进程同时发起一个广播，每个进程向所有其他进程发送m个字
- All-to-All归约
- 应用：矩阵相乘，矩阵向量相乘





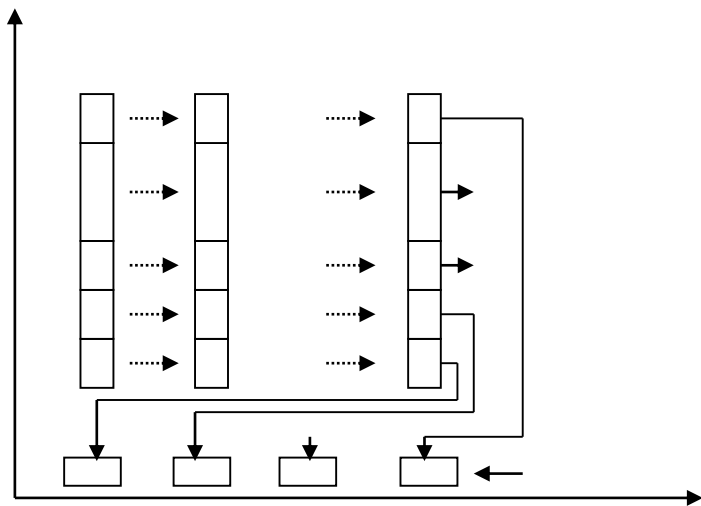
组收集

- `int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
- `MPI_Allgatherv`
- 相当于组内每个进程都执行一次收集
- All-to-All广播

归约并散发

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

- 归约后再执行一次散发操作
- All-to-all归约



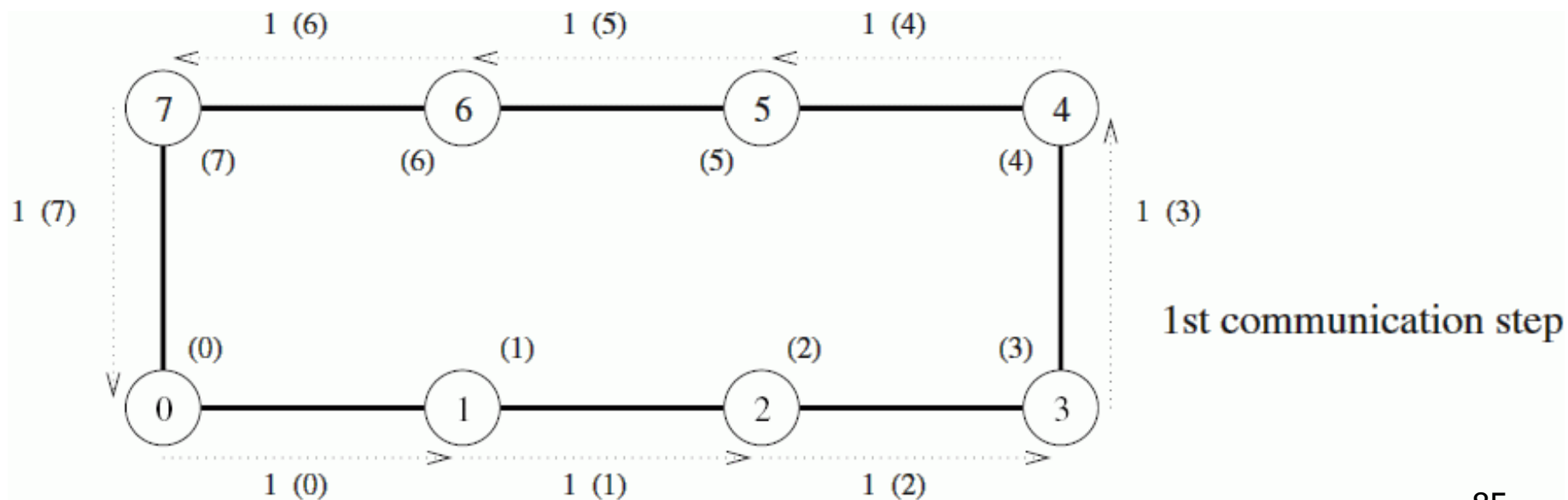


算法思想

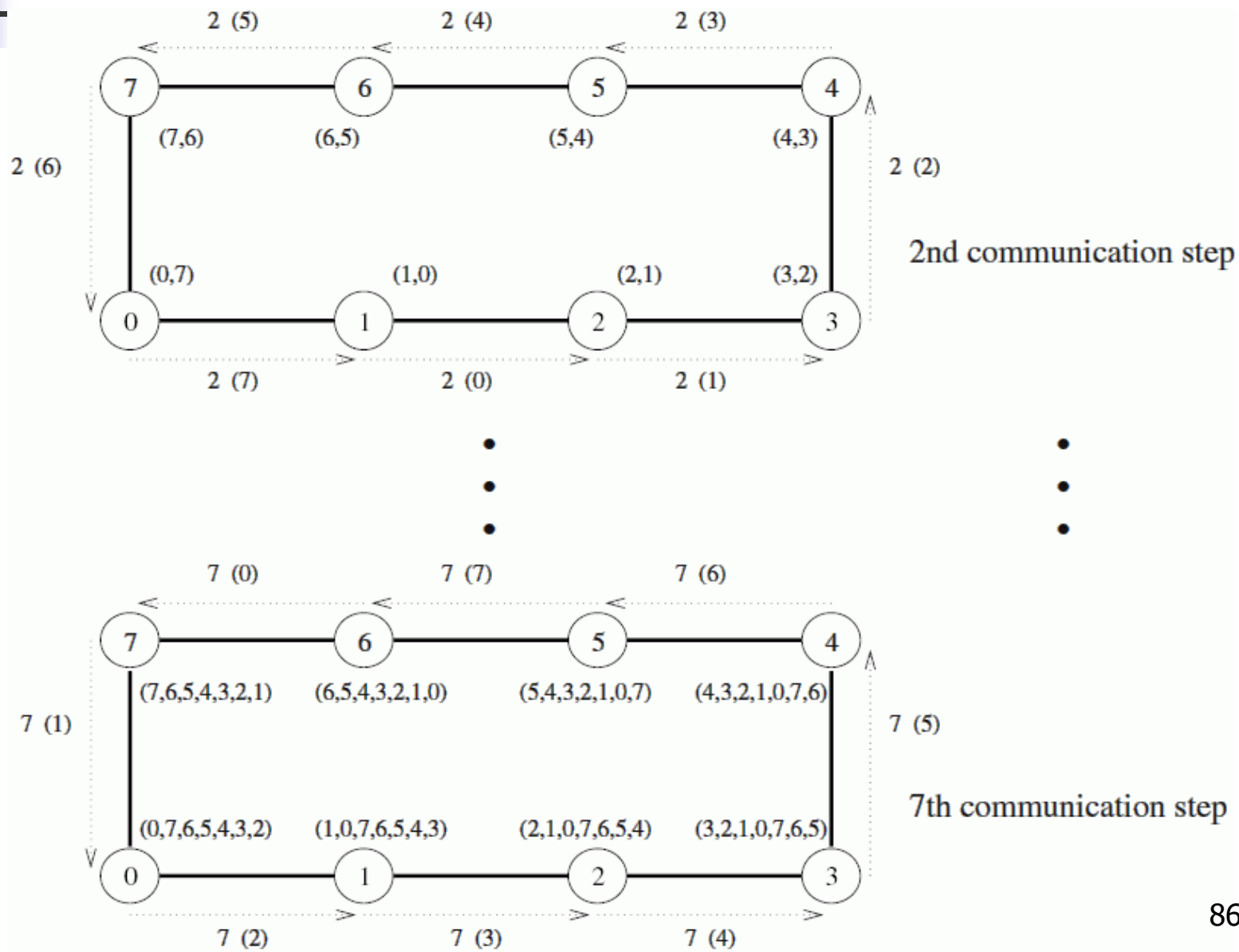
- 简单方法： p 个节点各执行一个 one-to-all 广播，性能差，可能时间是一个 one-to-all 广播的 p 倍
- 更有效地利用链路： p 个 all-to-all 广播同时进行，一个链路上同时传输的消息合并为大消息

线性阵列和环

- 保证每个节点总有数据传输给邻居——链路总保持忙
- 第一步：每个进程 → 某个邻居进程
- 第二步：每个进程将刚收到的数据转发到下一进程...



环的all-to-all广播（续）





环all-to-all广播算法

```
1. procedure ALL_TO_ALL_BC_RING(my_id, my_msg, p,  
   result)  
2. begin  
3.   left := (my_id - 1) mod p;  
4.   right := (my_id + 1) mod p;  
5.   result := my_msg;  
6.   msg := result;  
7.   for i := 1 to p - 1 do  
8.     send msg to right;  
9.     receive msg from left;  
10.    result := result ∪ msg;  
11.  endfor;  
12. end ALL_TO_ALL_BC_RING
```



环all-to-all归约

- 初始，每个进程都有 p 个消息（ m 个字）
- 每个消息会与其它 $p-1$ 个进程的各1个消息进行计算，最终保存在某个进程
- 实现：all-to-all广播实现的逆过程



环all-to-all归约的算法

```
1. procedure ALL_TO_ALL_RED_RING(my_id, my_msg, p,  
   result)  
2. begin  
3.   left := (my_id - 1) mod p;  
4.   right := (my_id + 1) mod p;  
5.   recv := 0;  
6.   for i := 1 to p - 1 do  
7.     j := (my_id + i) mod p;  
8.     temp := msg[j] + recv;  
9.     send temp to left;  
10.    receive recv from right;  
11.  endfor;  
12.  result := msg[my_id] + recv;  
13. end ALL_TO_ALL_RED_RING
```

其他网络?

复杂性分析?



All归约与前缀和

- all-to-one归约+one-to-all广播——完成后，每个进程都有相同的归约结果
- 与all-to-all归约不同——p个all-to-one归约，每个进程得到不同归约的结果
- 应用：同步操作实现
- 快速算法：利用all-to-all广播方法，每个步骤不是将消息组合，而是进行运算
$$T=(t_s+t_w m)\log p$$



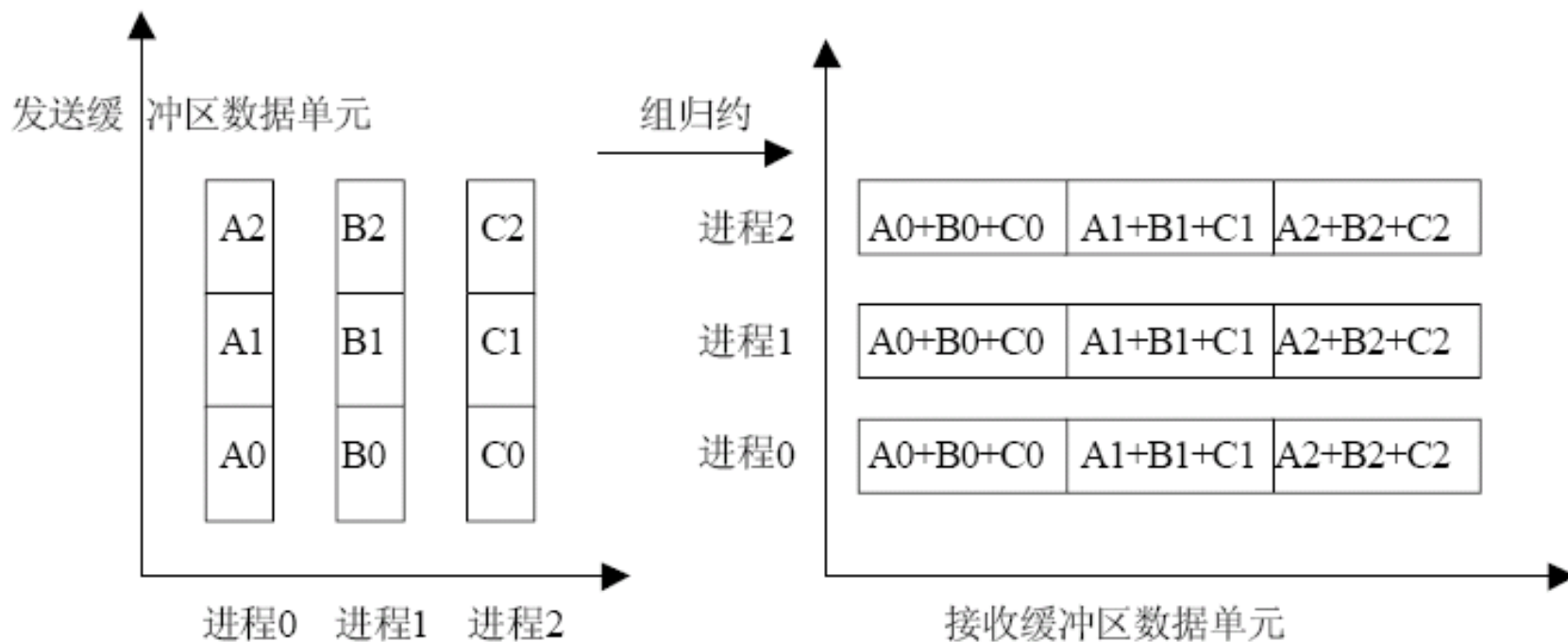
前缀和

- 操作之前，进程 P_k 保存数 n_k ，完成后前缀和操作后，保存 $\sum_{i=0}^k n_i$
- 利用all-to-all广播方法
 - 每个步骤进行加法，而不是组合消息
 - 两个缓冲
 - result——本节点最终结果，只累加编号小于自己的节点发送来的数据
 - 传输缓冲：累加所有收到的数据

全归约

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

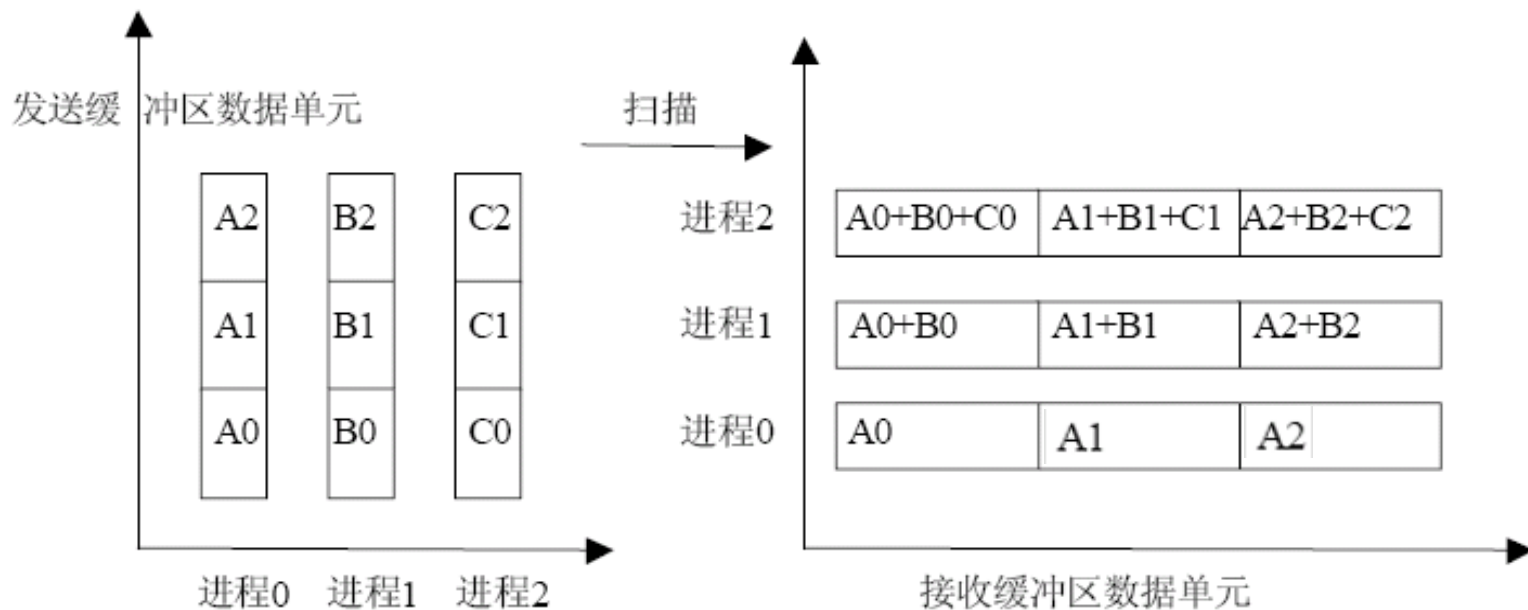
- 相当于每个进程都作为ROOT执行一次相同的归约操作



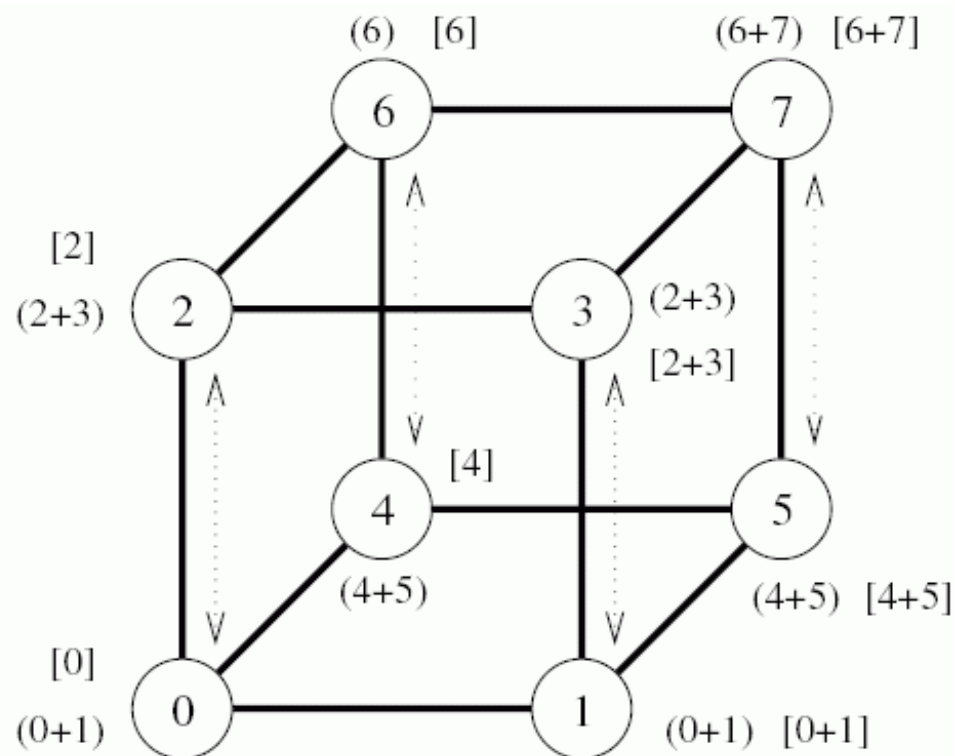
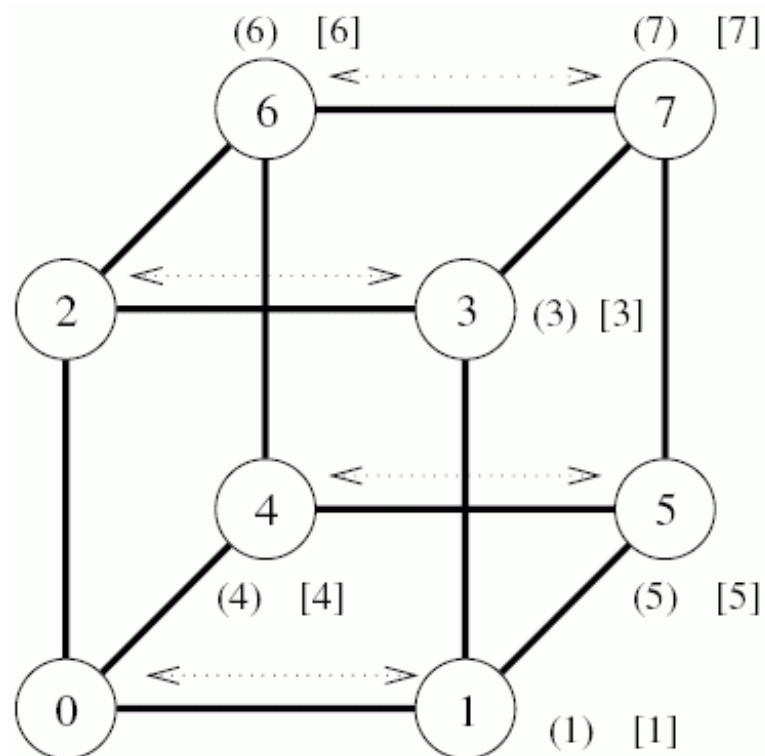
扫描

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

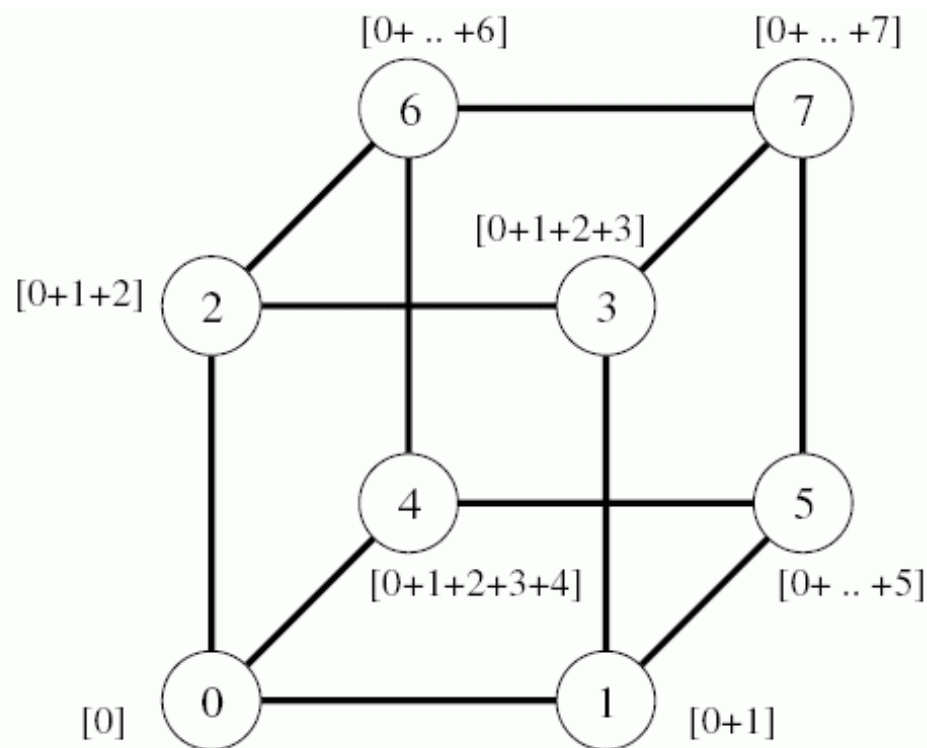
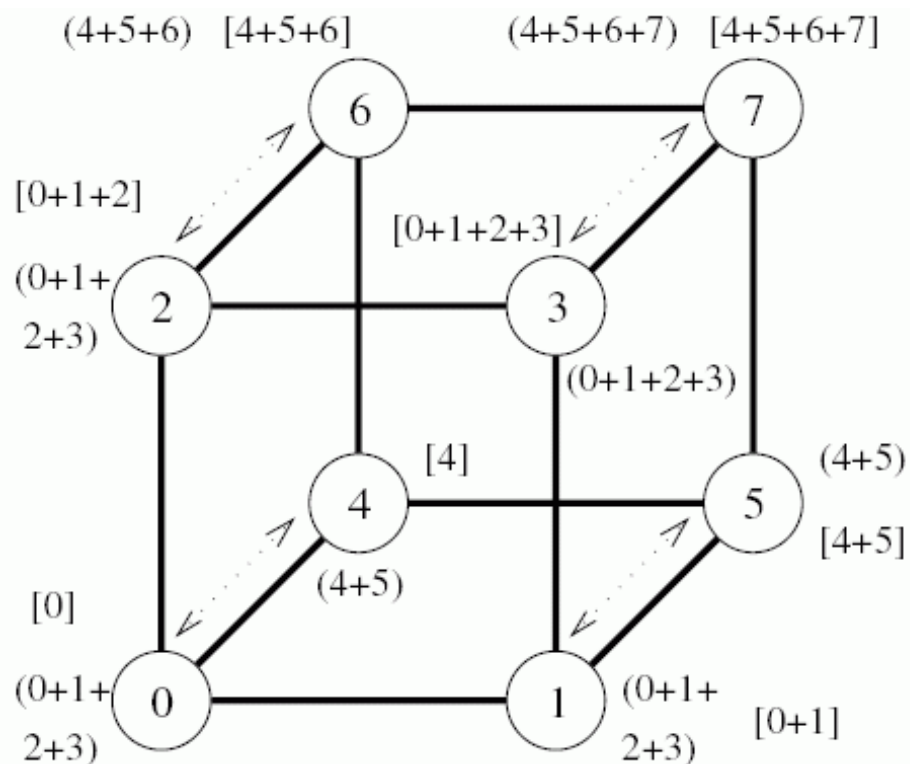
- 可以将扫描看作是一种特殊的归约，即每一个进程都对排在它前面的进程进行归约操作
- 前缀和



前綴和算法图示



前綴和算法图示（续）





算法

1. **procedure** PREFIX SUMS HCUBE(*my_id*, *my_number*, *d*,
 result)
2. **begin**
3. *result* := *my_number*;
4. *msg* := *result*;
5. **for** *i* := 0 **to** *d* - 1 **do**
6. *partner* := *my_id* XOR 2^i ;
7. **send** *msg* to *partner*;
8. **receive** *number* from *partner*;
9. *msg* := *msg* + *number*;
10. **if** (*partner* < *my_id*) **then** *result* := *result* + *number*;
11. **endfor**;
12. **end** PREFIX SUMS HCUBE



用户自定义归约操作

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

□ **commute**: 是否满足交换率

```
typedef void MPI_User_function(void *invec, void *inoutvec,  
                                int *len, MPI_Datatype *datatype);
```

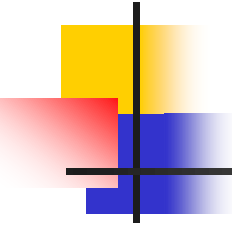
□ **invec**和**inoutvec**分别指出将要被归约的数据所在的缓冲区的首址,**len**指出将要归约的元素的个数,**datatype**指出归约对象的数据类型。

- 通常的数据类型可以传给用户自定义的参数,然而互不相邻的数据类型可能会导致低效率。
- 在用户自定义的函数中不能调用MPI中的通信函数。
- `int MPI_Op_free(MPI_Op *op)`



自定义归约操作的例：计算一个复数数组的积

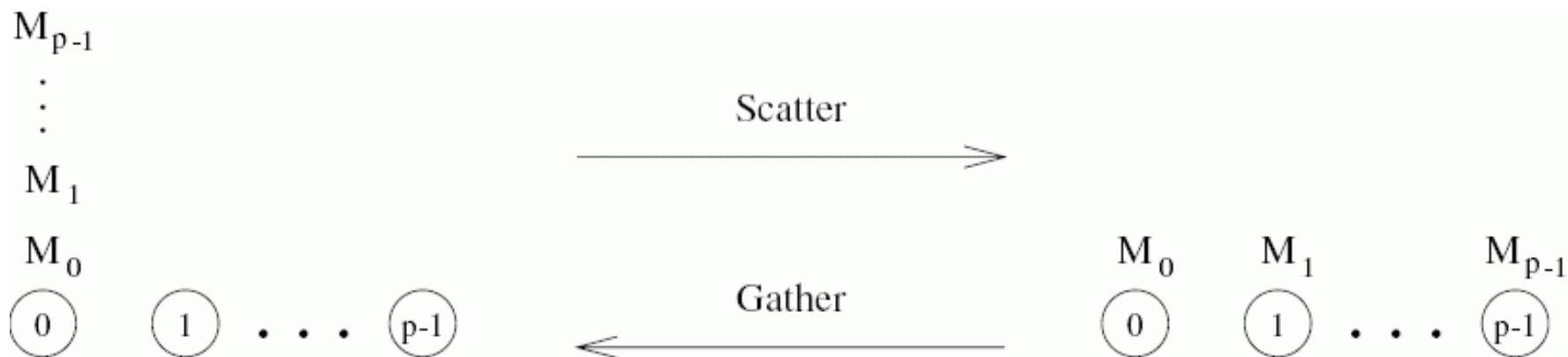
```
typedef struct {  
    double real,imag;  
} Complex;  
void myProd(Complex *in, Complex *inout, int *len, MPI_Datatype *dptr)  
{  
    int i;  
    Complex c;  
    for (i=0; i < *len; ++i) {  
        c.real = inout->real*in->real - inout->imag*in->imag;  
        c.imag = inout->real*in->imag + inout->imag*in->real;  
        *inout = c;  
        in++; inout++;  
    }  
}
```



```
/* 然后调用它 */
/* 每个进程都有一个100个元素的复数数组 */
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;
/* 告知MPI复数结构是如何定义的 */
MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
MPI_Type_commit(&ctype);
/* 生成用户定义的复数乘积操作 */
MPI_Op_create(myProd, True, &myOp);
MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);
/* 归约完成后计算结果(为100个复数)就已经存放在根进程 */
```

Scatter和Gather

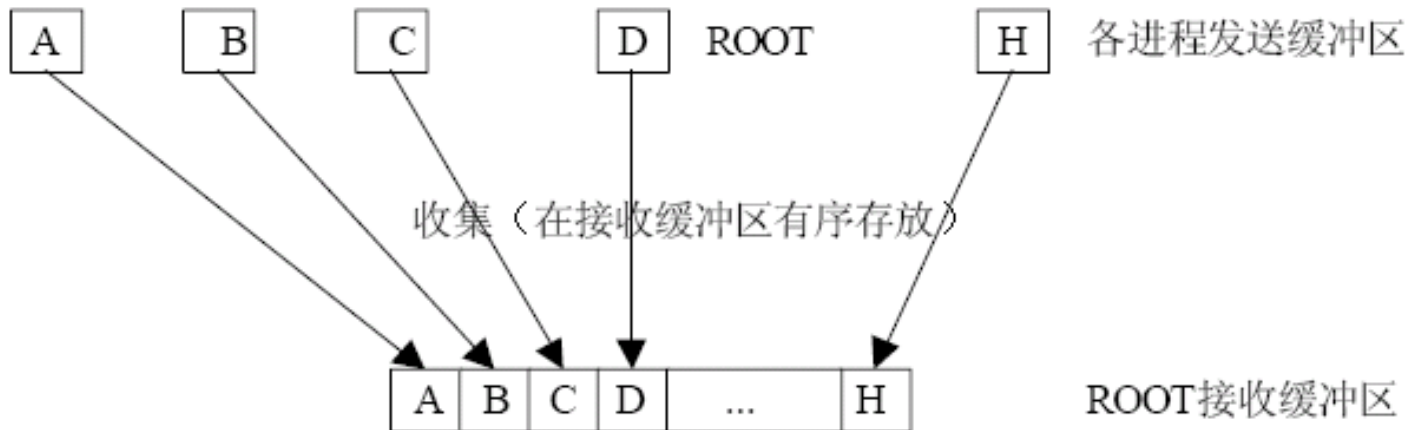
- **scatter**: 源节点向其它每个节点发送不同数据, one-to-all个体化通信
与one-to-all广播（发送相同数据）不同
- 对应操作: **gather, concatenation**
目的节点从其他每个节点接收不同数据
不同于all-to-one归约, 无合并/归约运算



MPI gather

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

- **recvcount**: 从每个进程接收的数据个数，而不是一共需要接收的数据个数





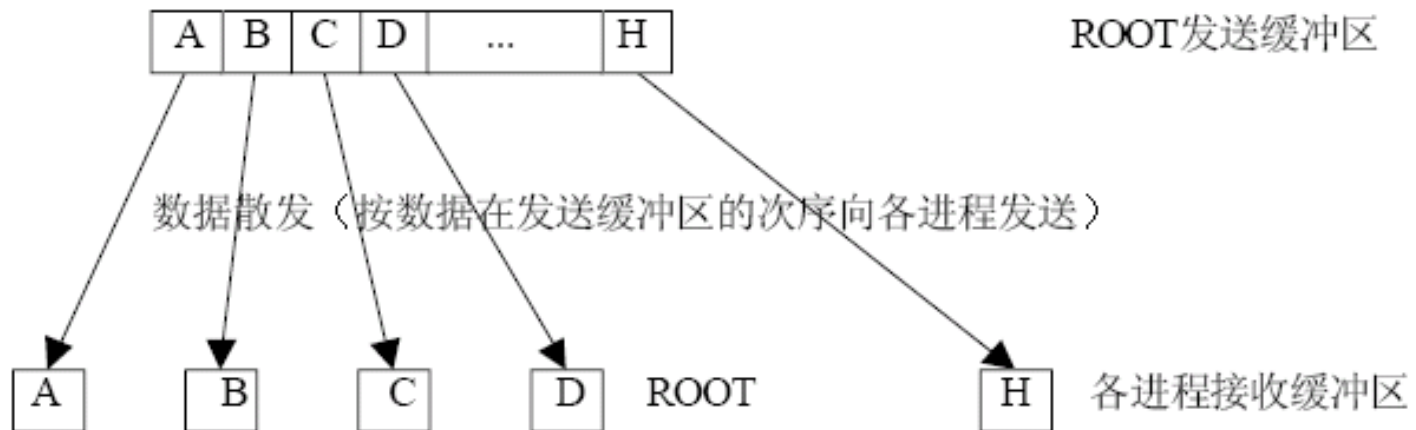
MPI gatherv

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

- 可以从不同的进程接收不同数量的数据
- **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
- **displs**: 整数数组, 每个入口表示相对于recvbuf的位移

MPI scatter

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





MPI scatterv

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

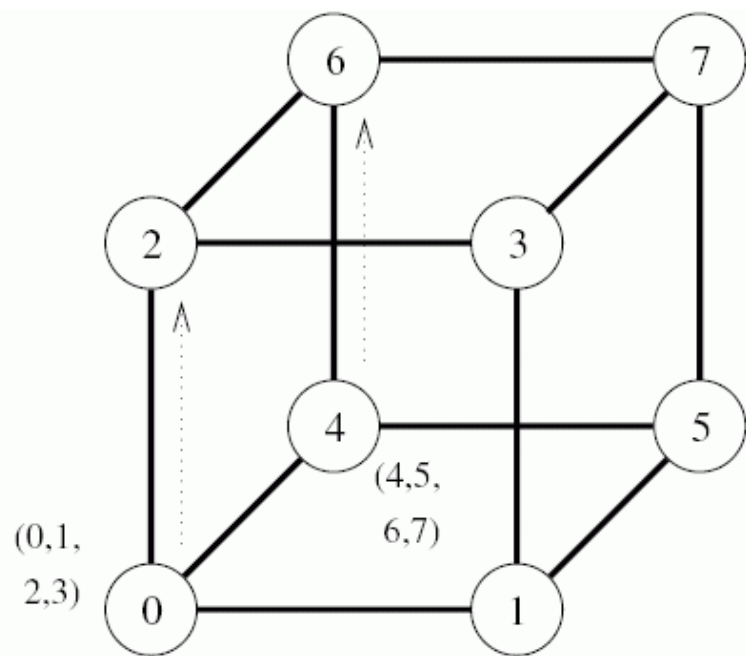
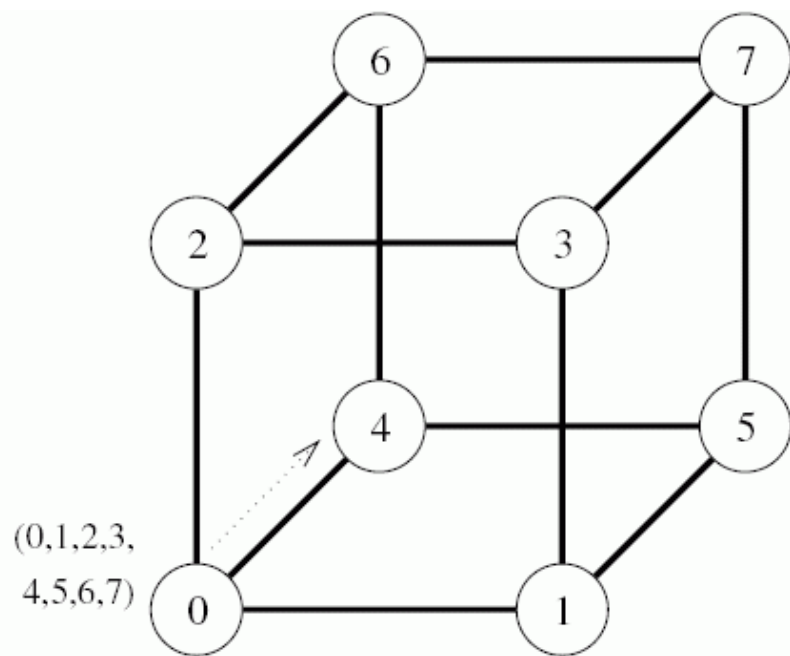
- 允许ROOT向各个进程发送个数不等的数据
- 与MPI_Gatherv相对应



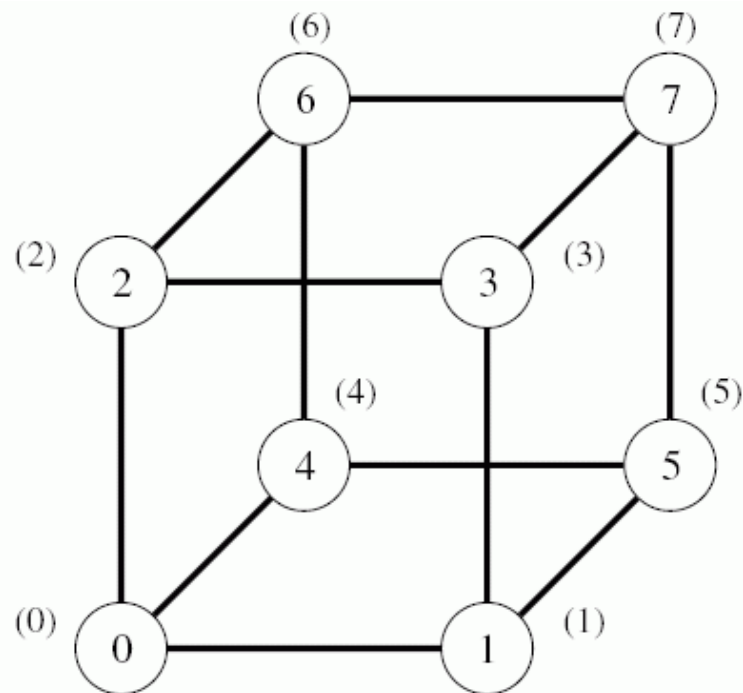
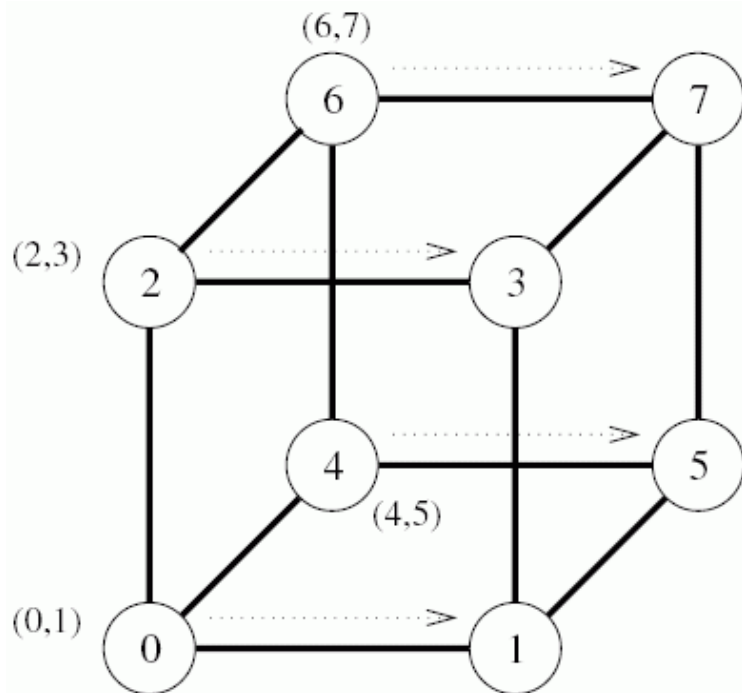
scatter实现算法

- one-to-all广播相同模式，但消息大小和内容不同——折半法
 - 第一步：源节点将p个消息的一半→邻居
 - 第二步：两个节点将各自消息的一半→邻居
- 消息用目的节点编号加以标记
- 每个步骤，节点i→邻居j，两个节点编号差异对应哪一维，所有消息就在那一维折半——为0的分为一组，为1的为另一组

scatter算法图示



scatter算法图示（续）





gather实现算法

○ scatter算法的逆操作

- 初始，每个节点保存一个消息
- 第一步：奇数节点→偶数节点，偶数节点将两个消息连接
- 第二步：不能被4整除的偶数节点→4倍数节点，消息连接...

○ 可应用于mesh和线性阵列

○ 复杂性分析

$$T = \sum_{i=0}^{\log p} (t_s + t_w mp / 2^{i+1}) = t_s \log p + t_w m(p-1)$$

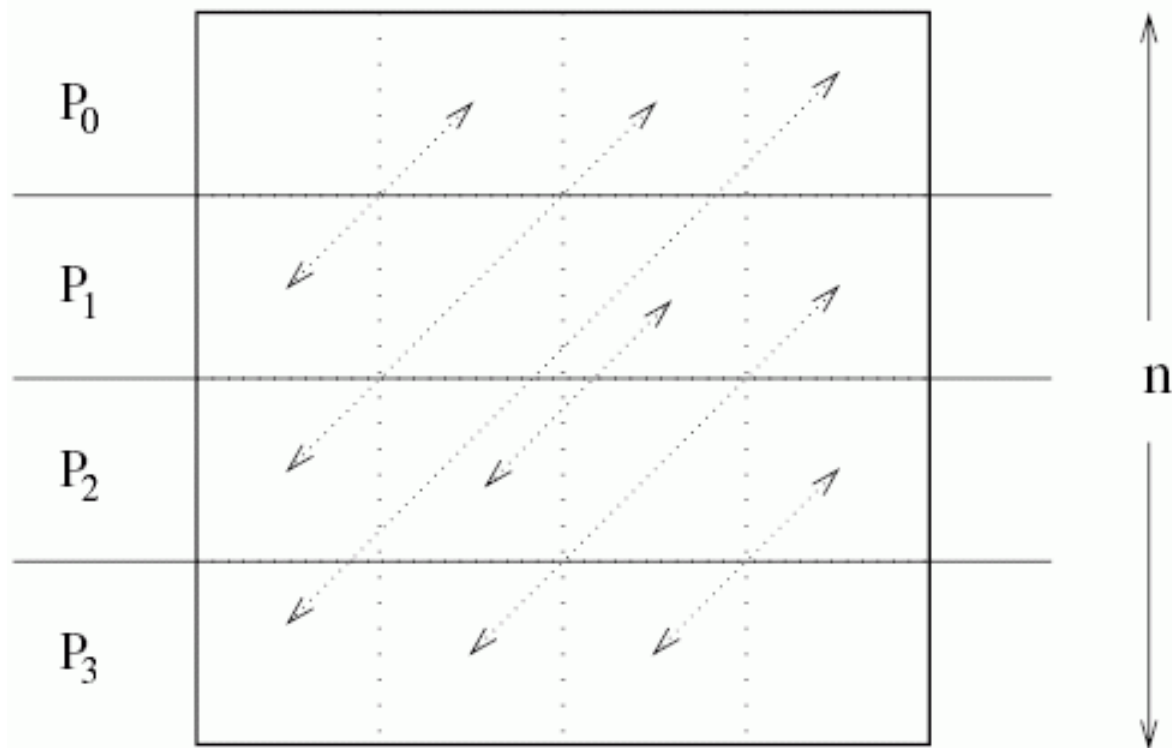
All-to-All个体化通信

- 每个节点都向其它每个节点发送一个不同的消息，all-to-all广播是发送相同消息
- 应用：FFT，矩阵转置，取样排序，并行数据库join操作



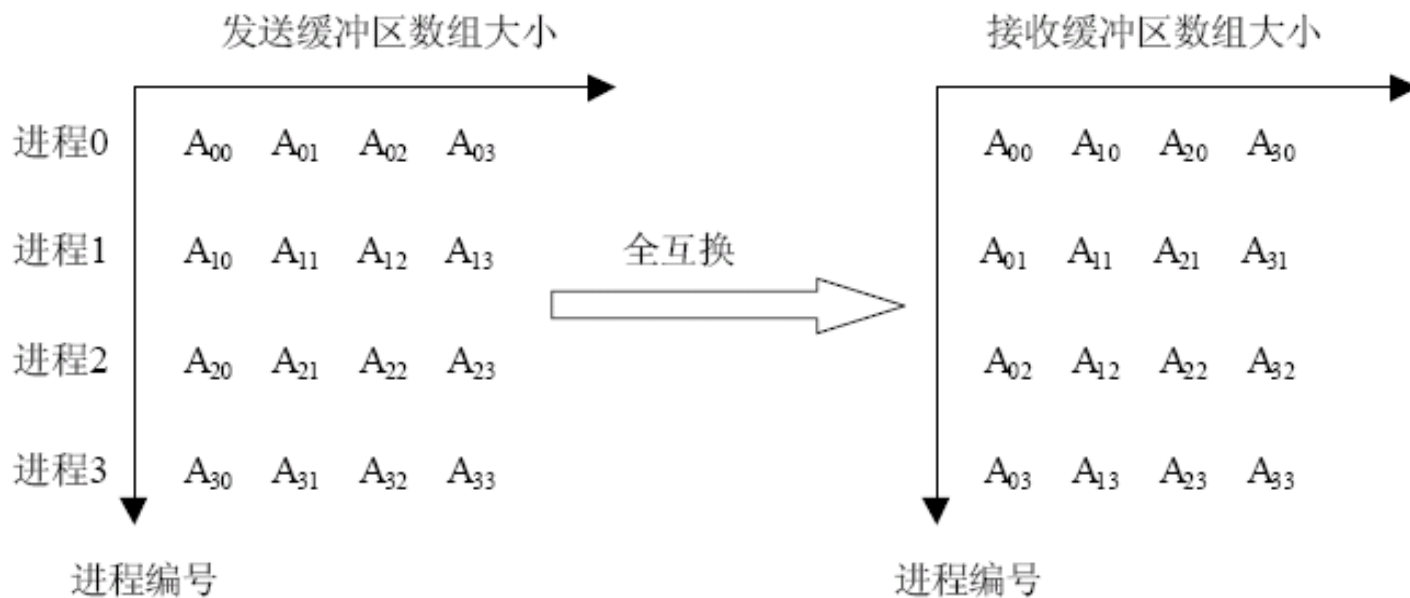
例 矩阵转置

- 每个进程保存矩阵一行: $[i, 0], [i, 1], \dots$
- 其中 $n-1$ 个要分别发送给其它 $n-1$ 个进程



全互换

- 组内进程之间完全的消息交换，其中每一个进程都向其它所有的进程发送消息，同时每一个进程都从其它所有的进程接收消息。
- 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
- All-to-all个体化通信

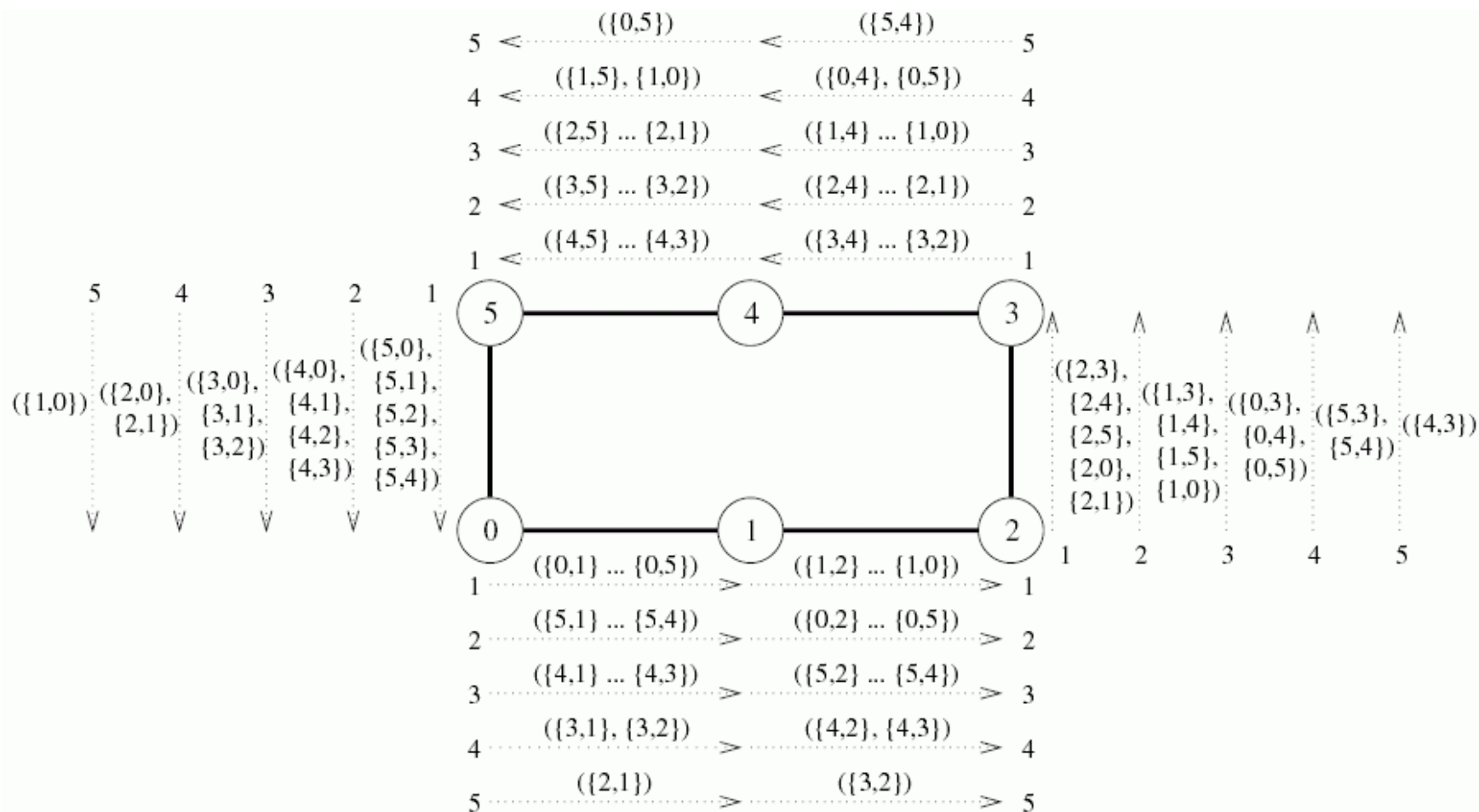




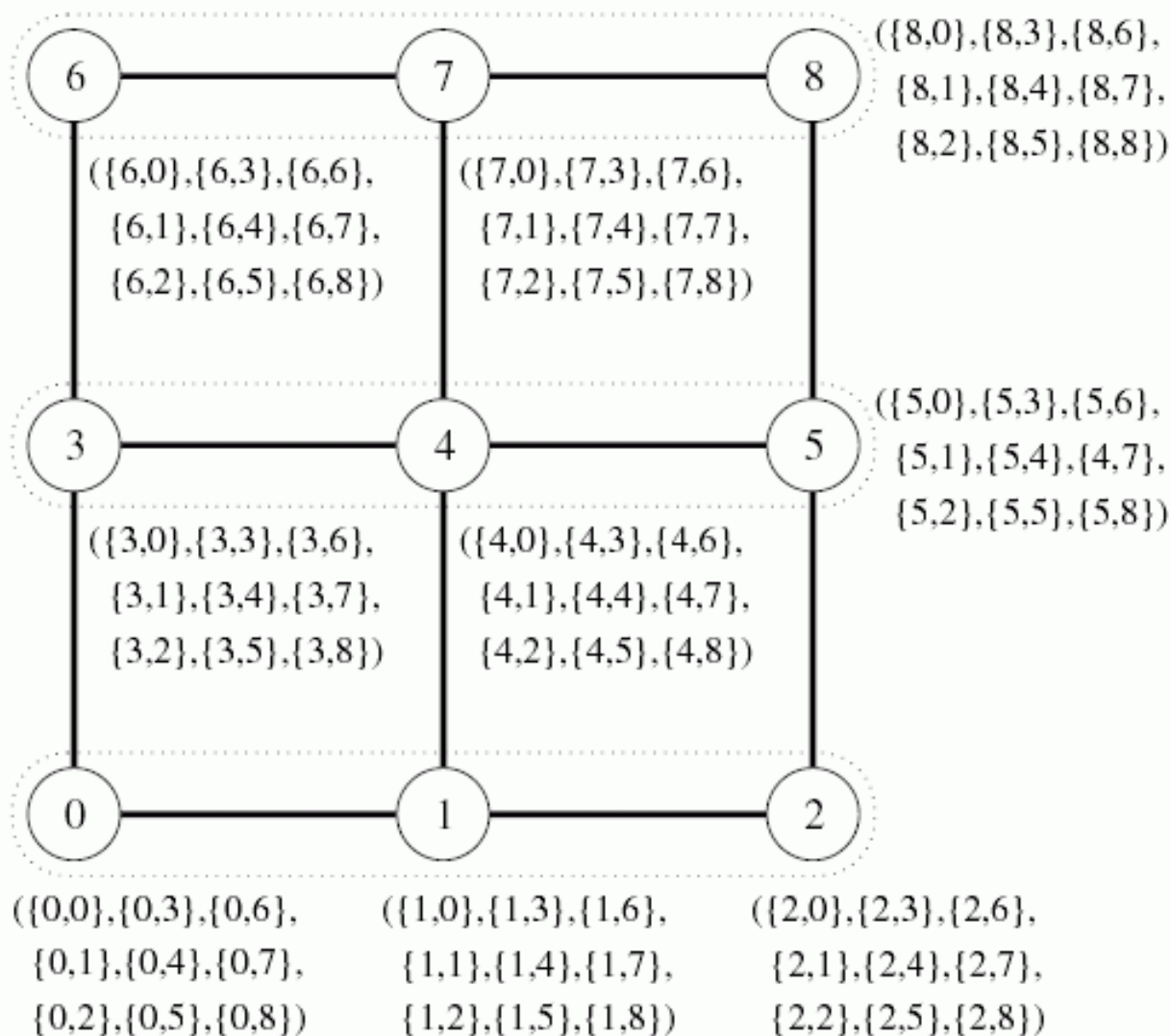
API

- `int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
- `int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)`

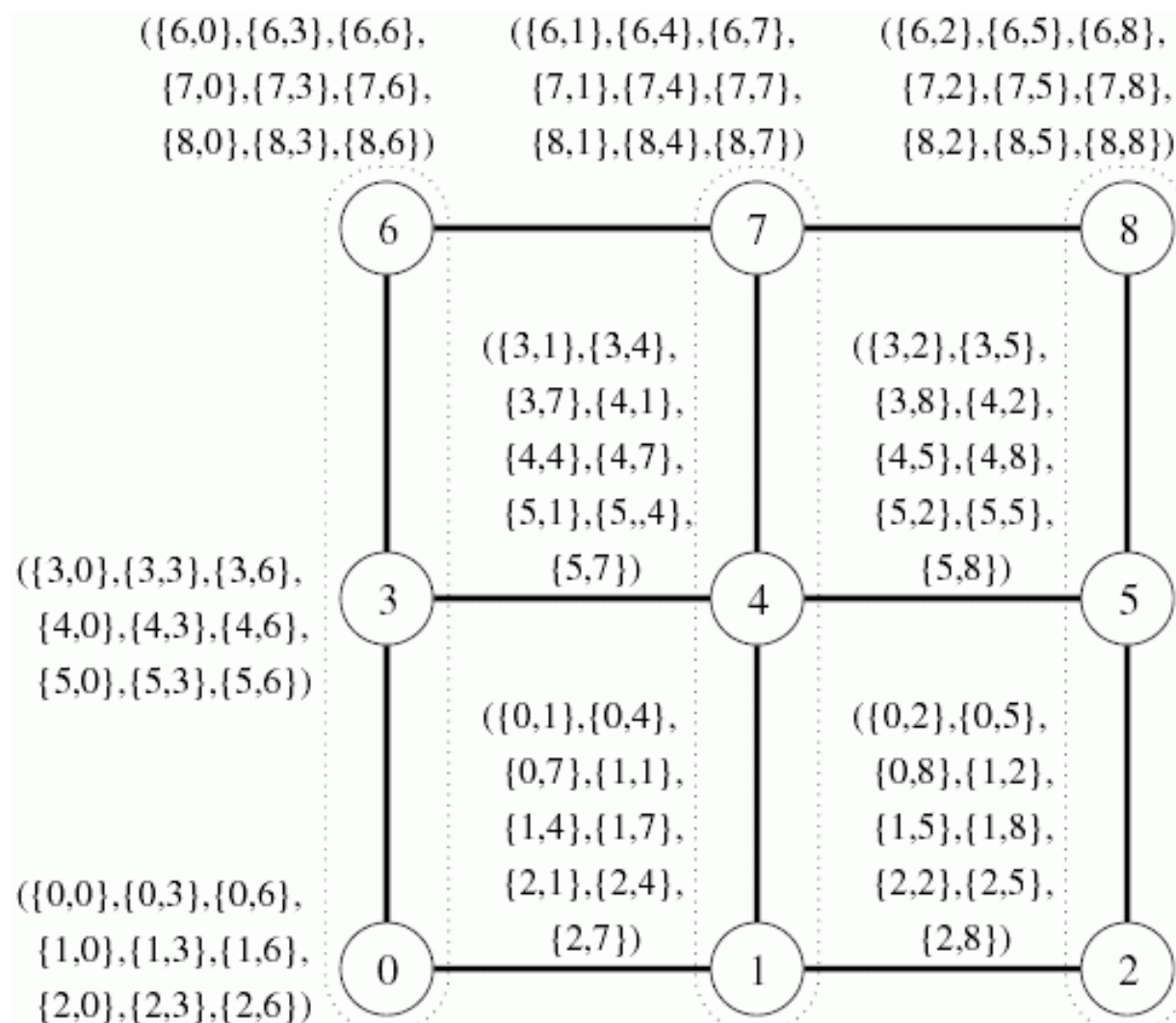
环算法——流水线



mesh算法——两个维度环

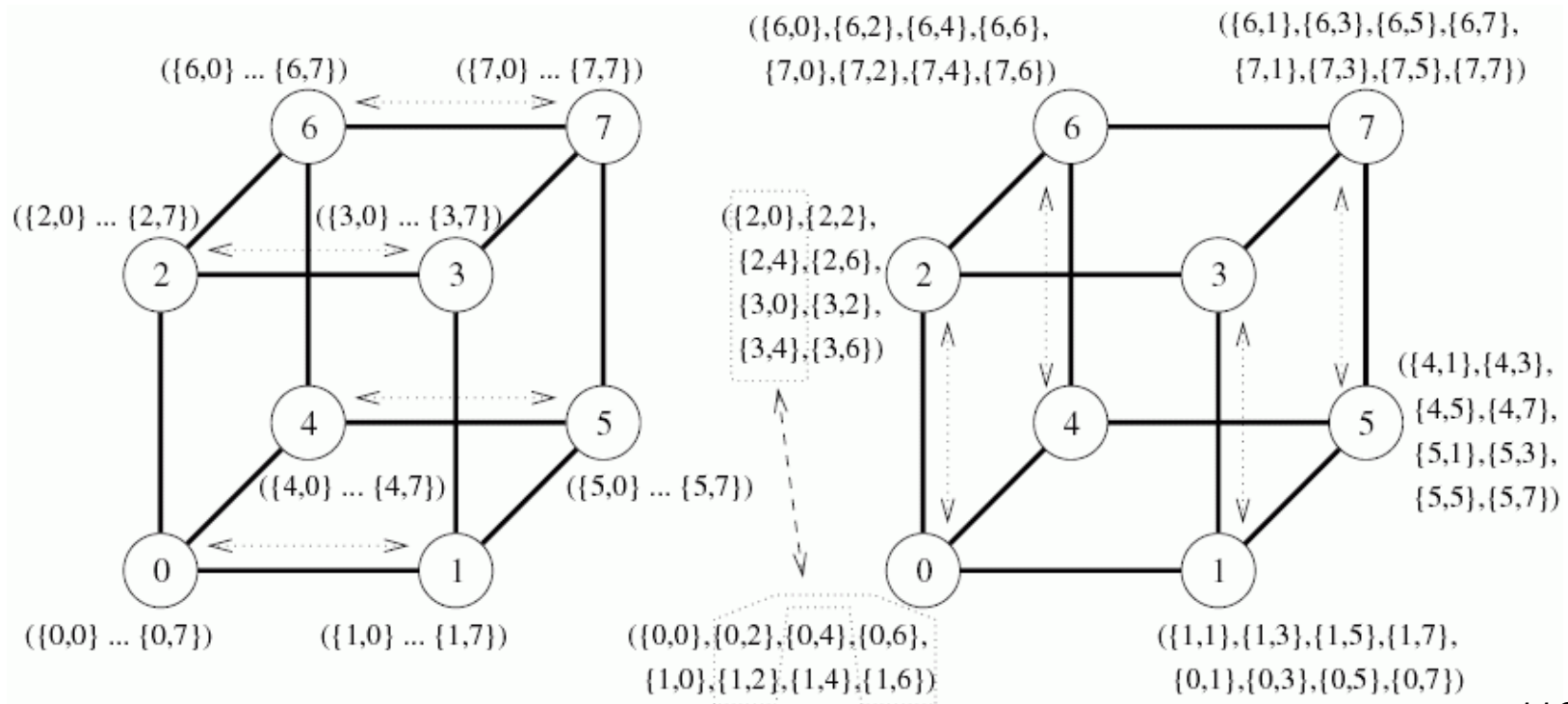


mesh算法（续）

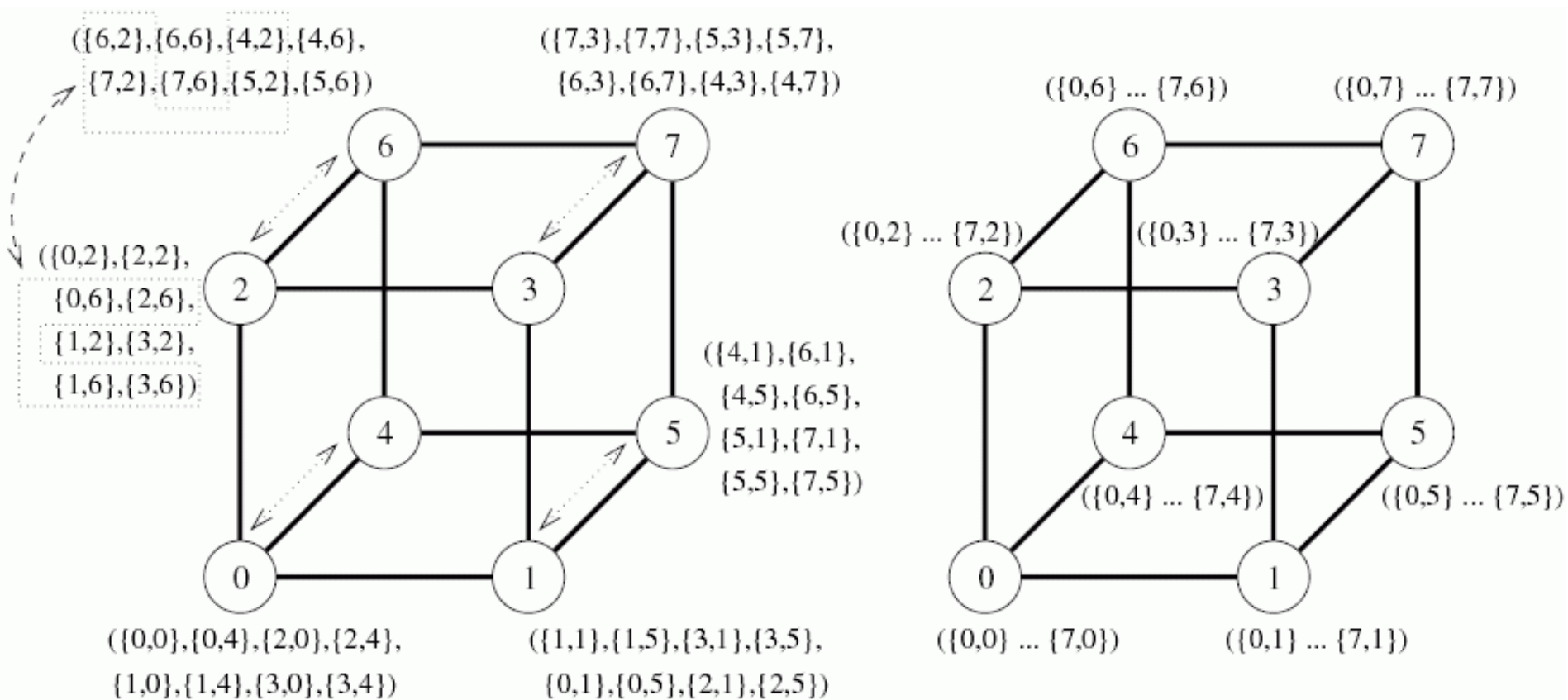


超立方算法

- $\log p$ 维的 mesh, $\log p$ 个步骤, 每步每个节点将自己的 p 个消息发送一半到某个相邻节点, 再接收相同大小数据



超立方算法（续）

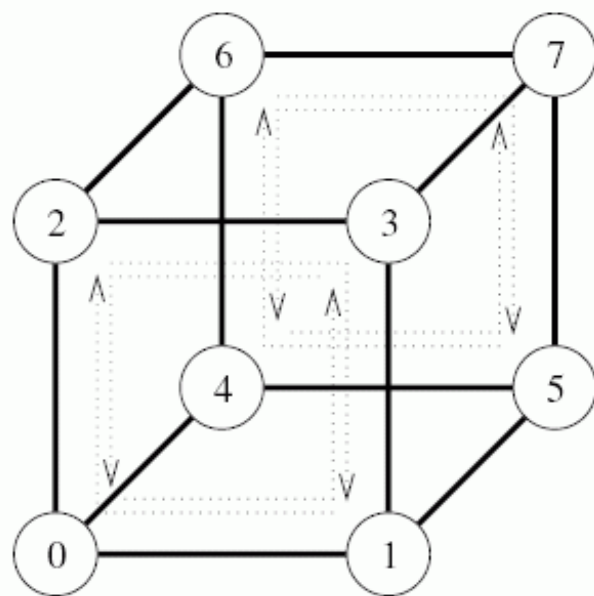
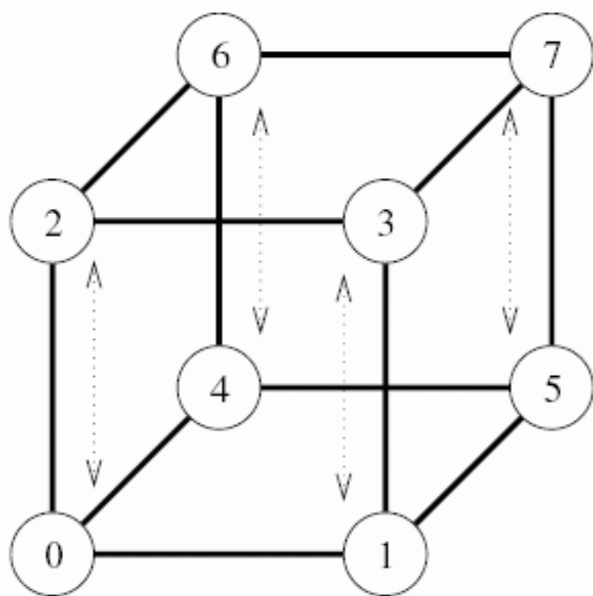
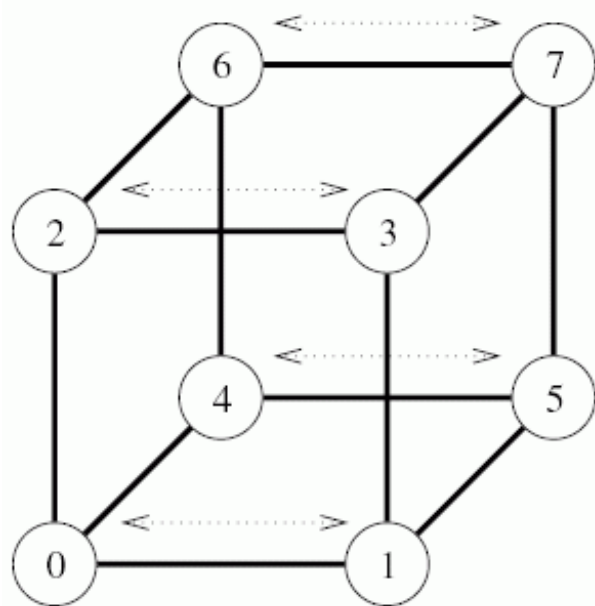




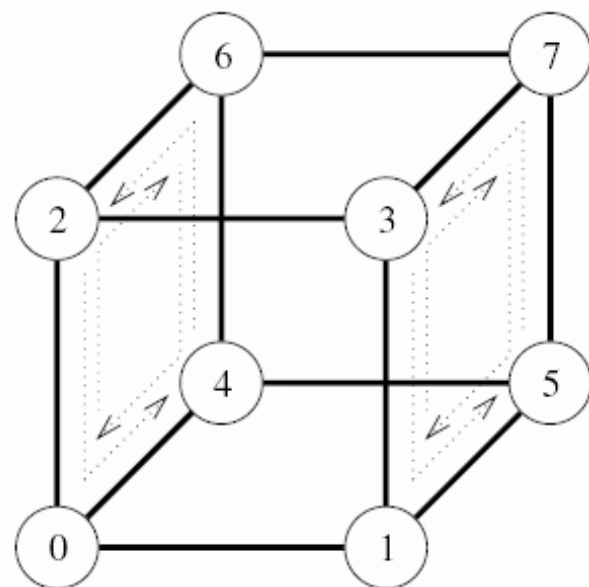
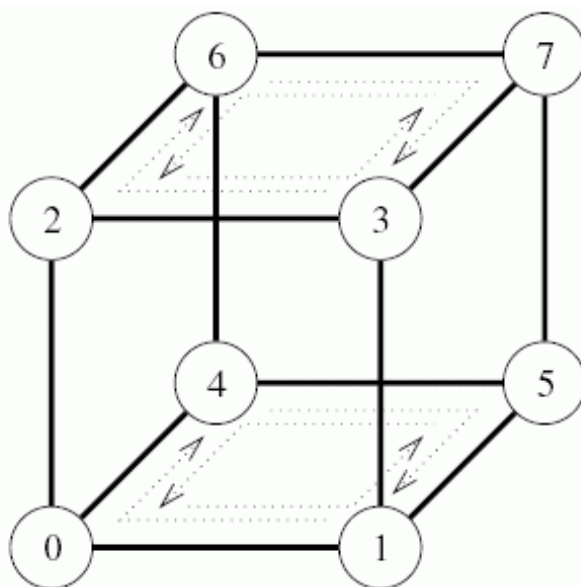
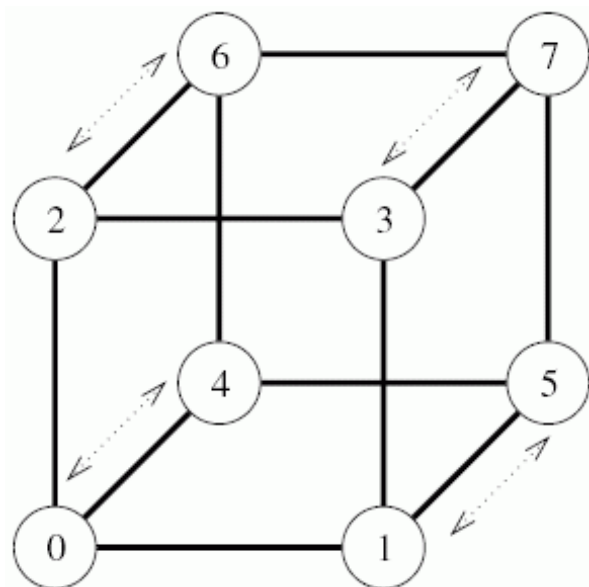
最优算法

- $p-1$ 个步骤，每个步骤每个节点将一个 m 字的消息直接发送到目的节点
- 关键：选择传输路径，避免冲突
- 第 j 步：节点 i 与节点 $i \text{ XOR } j$ 交换数据
- E-cube 路由方式：按维度递增的顺序，最先（后）考虑最低（高）位，对应位不同则在对应维度移动
- 不会产生拥塞

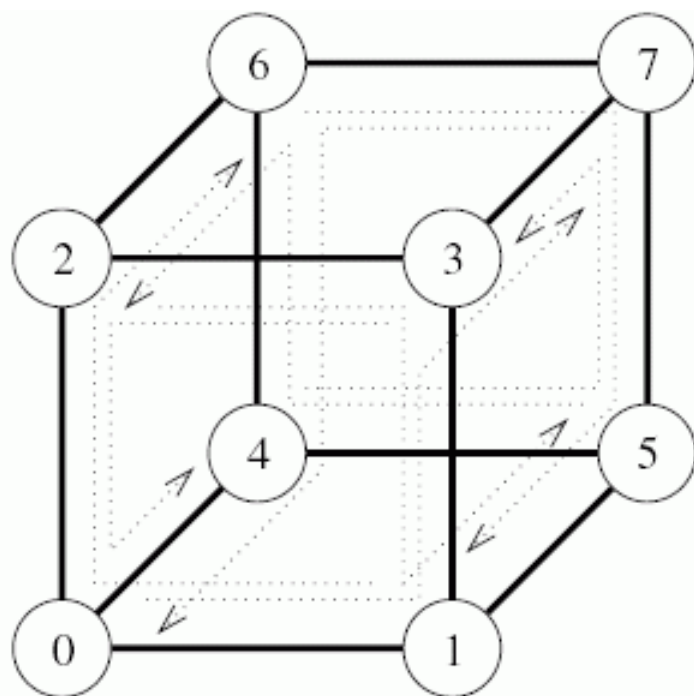
图示



图示（续）



图示（续）



0> 1> 3> 7
1> 0> 2> 6
2> 3> 1> 5
3> 2> 0> 4
4> 5> 7> 3
5> 4> 6> 2
6> 7> 5> 1
7> 6> 4> 0

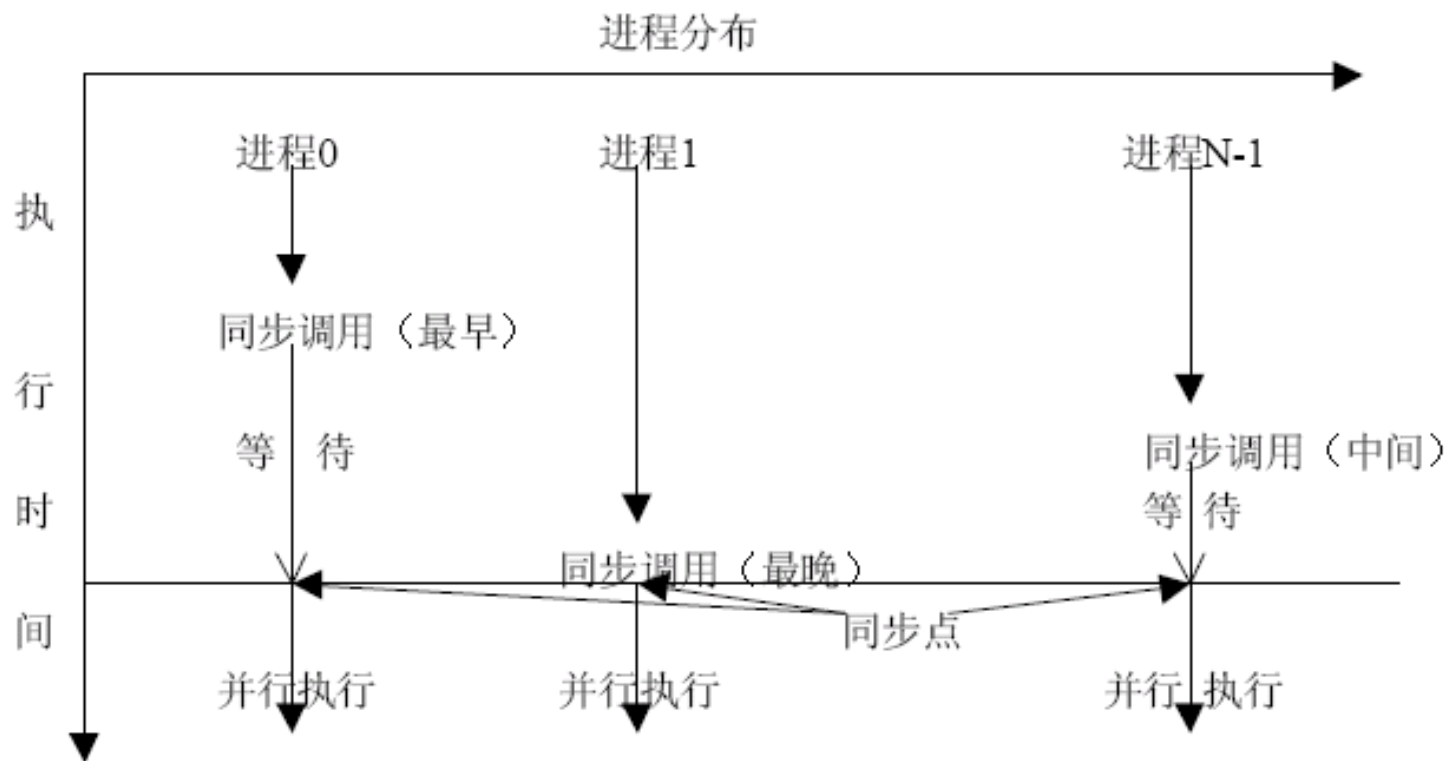


算法描述

```
1. procedure ALL_TO_ALL_PERSONAL( $d, my\_id$ )
2. begin
3.   for  $i := 1$  to  $2^d - 1$  do
4.     begin
5.        $partner := my\_id \text{ XOR } i$  ;
6.       send  $M_{my\_id, partner}$  to  $partner$ ;
7.       receive  $M_{partner, my\_id}$  from  $partner$ ;
8.     endfor;
9. end ALL_TO_ALL_PERSONAL
```

Barrier

- `int MPI_Barrier(MPI_Comm comm)`
- 只有当所有的语句都执行了该调用后才一起向下执行





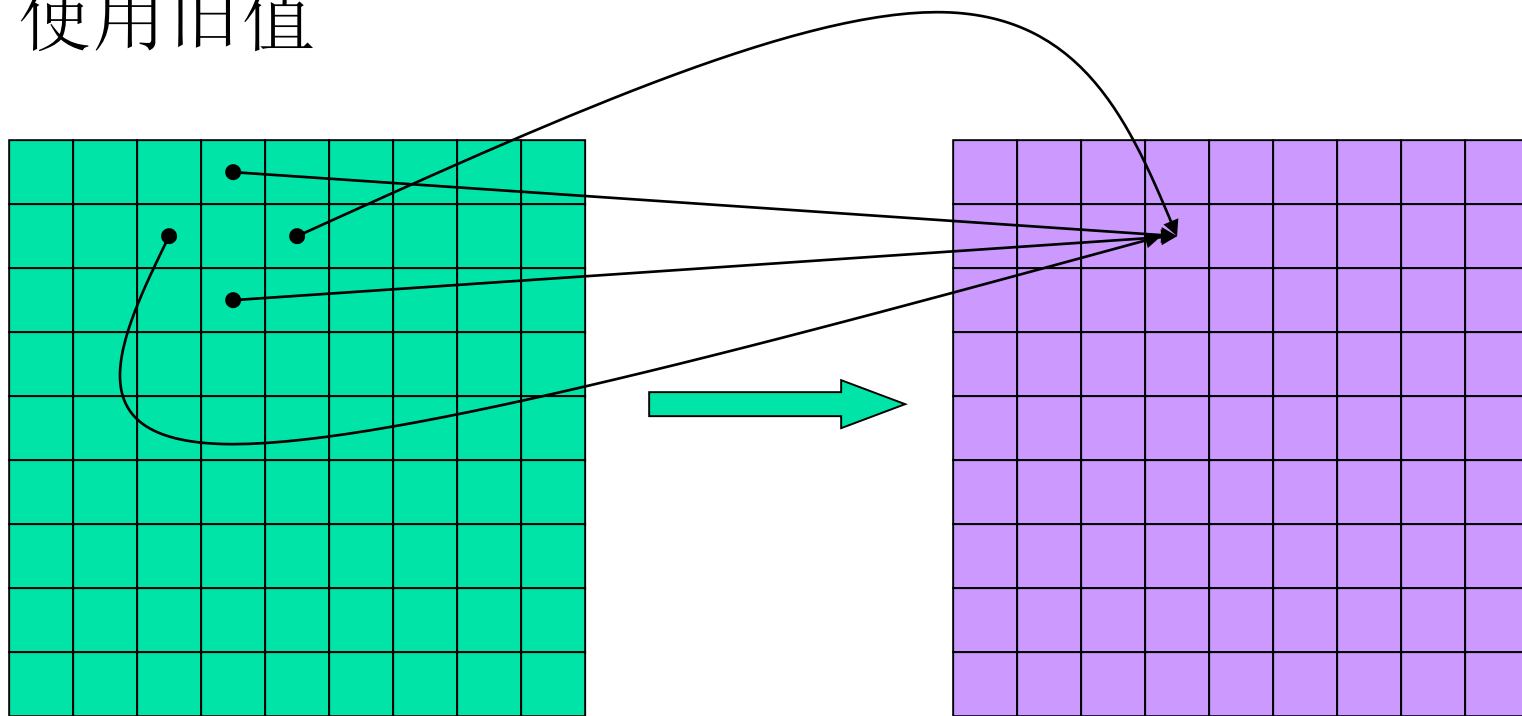
提纲

- MPI概念和基本原语
- MPI编程模型
- 点对点通信进阶
- 组通信
- 非阻塞通信
- 多线程混合编程
- 自定义数据类型
- 进程组与通信域

并行Jacobi迭代

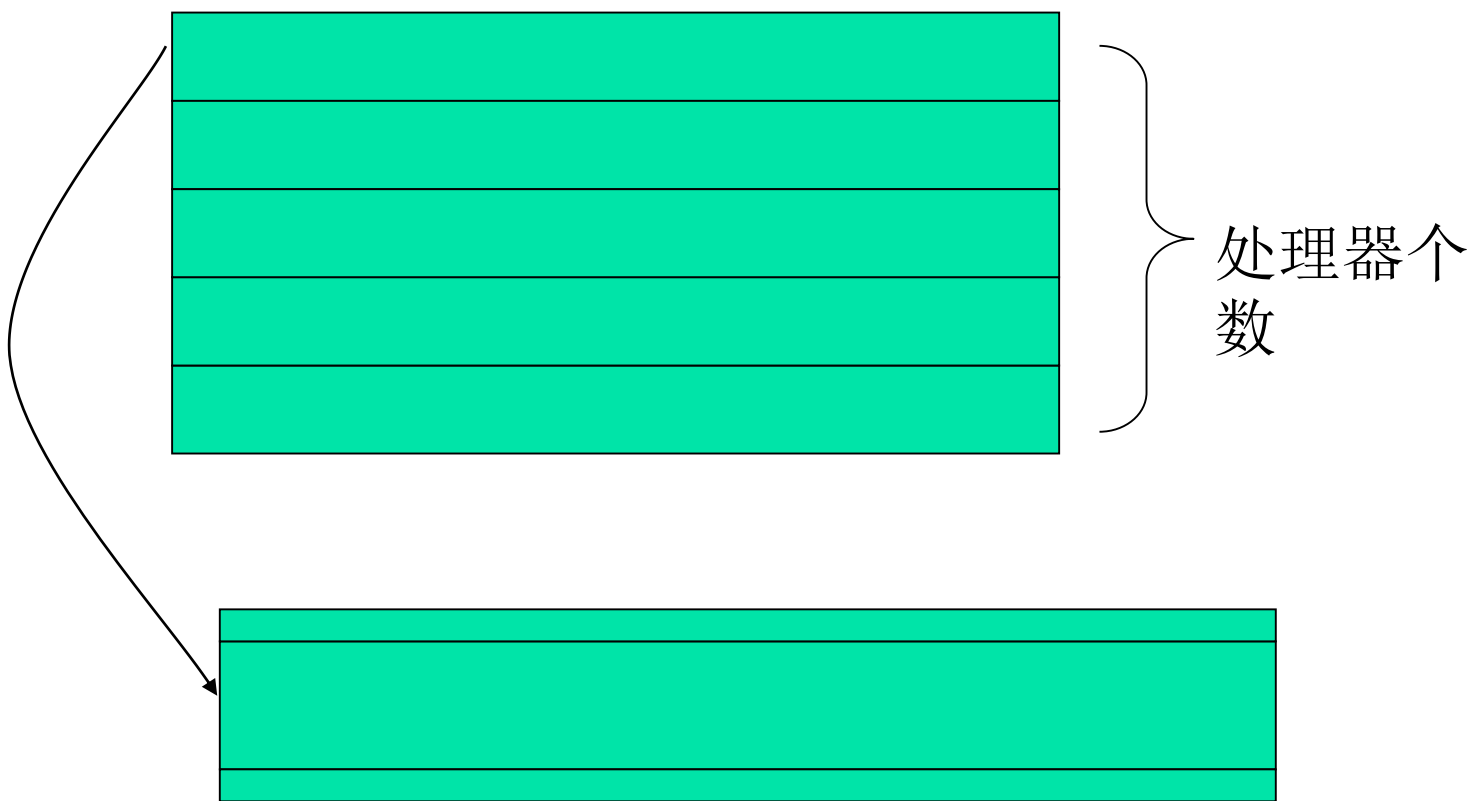
$$B(I,J)=(A(I,J+1)+A(I,J-1)+A(I-1,J)+A(I+1,J))/4$$

- 上下左右相加取平均
- 使用旧值





数据划分





回顾：阻塞通信解决Jacobi迭代

```
//设置进程标识
```

```
if(myid > 0) up = myid - 1;
```

```
else up = MPI_PROC_NULL;
```

```
if(myid < 3) down = myid + 1;
```

```
else down = MPI_PROC_NULL;
```

```
//对于每一次迭代都执行下面的通信操作：
```

```
// 从下侧的邻居得到数据
```

```
    MPI_Recv(a, size, MPI_FLOAT, down, 10, MPI_COMM_WORLD, &status);
```

```
// 向上侧的邻居发送数据
```

```
    MPI_Send(b, size, MPI_FLOAT, up, 10, MPI_COMM_WORLD);
```

```
// 向下侧的邻居发送数据
```

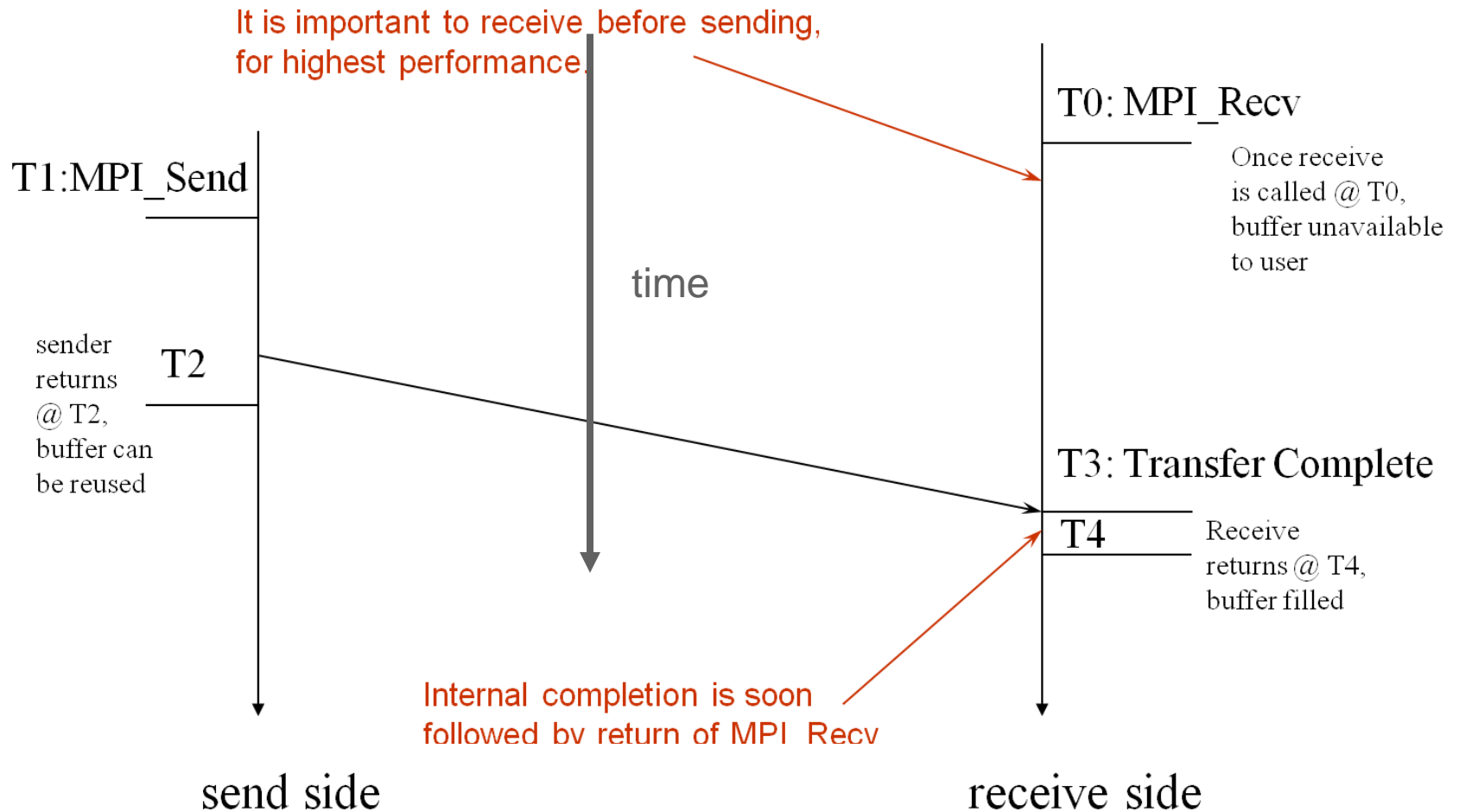
```
    MPI_Send(c, size, MPI_FLOAT, down, 9, MPI_COMM_WORLD);
```

```
// 从上侧的邻居接收数据
```

```
    MPI_Recv(d, size, MPI_FLOAT, up, 9, MPI_COMM_WORLD, &status);
```

```
// 计算
```

阻塞通信模式



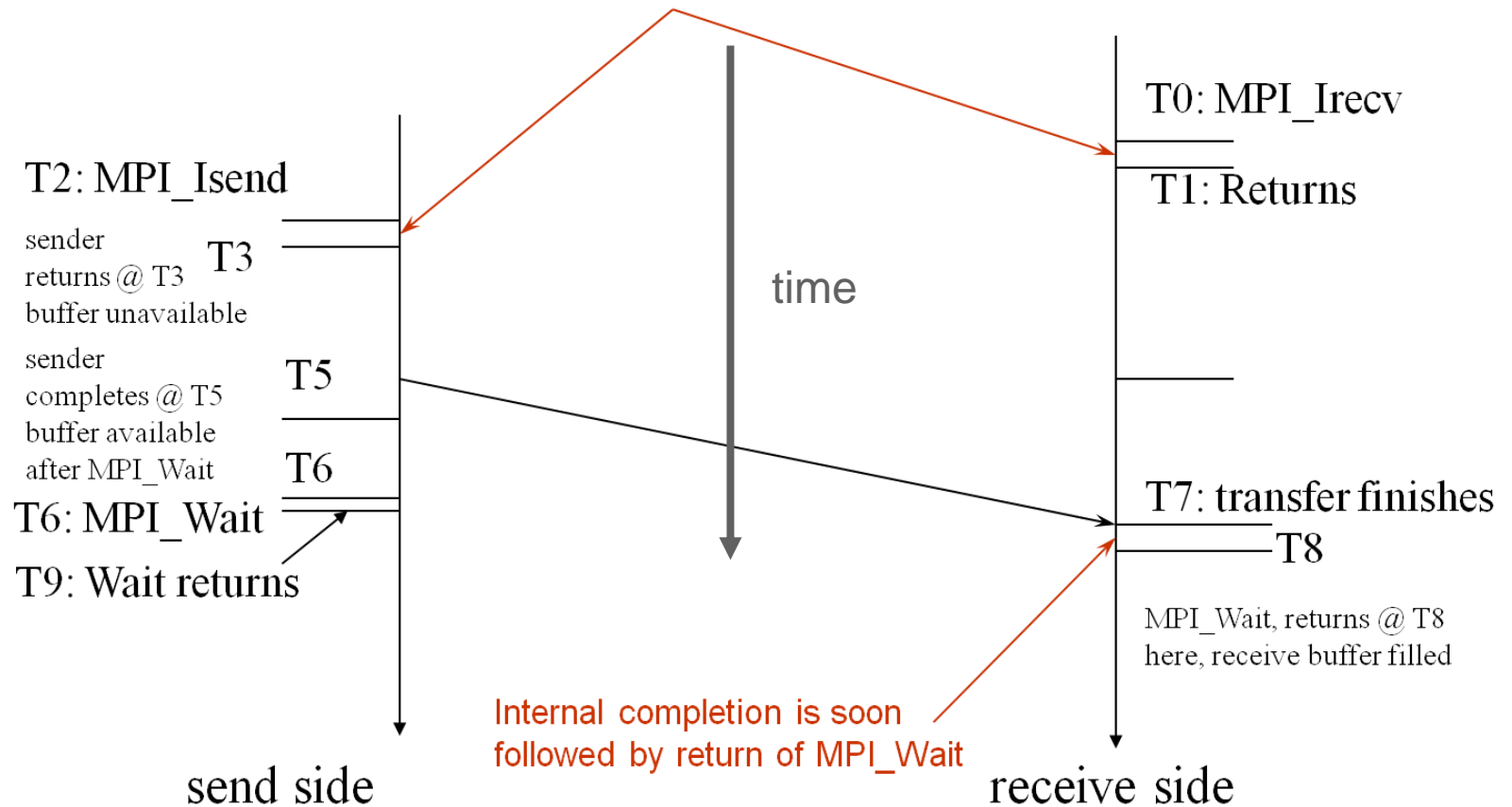


非阻塞通信

- 基本含义
 - 调用返回 \neq 通信完成
- 作用
 - 计算和通信的重叠
- 形式
 - 非阻塞发送和非阻塞接收

非阻塞通信示意

High Performance Implementations
Offer Low Overhead for Non-blocking Calls





消息传递与缓冲

- 当用户提供的buffer可被重用时，才表明send完成

```
*buf =3;  
MPI_Send(buf, 1, MPI_INT ...)  
*buf = 4; /* OK, 接收方总是会收到3 */
```

```
*buf =3;  
MPI_Isend(buf, 1, MPI_INT ...)  
*buf = 4; /* 不确定接收方收到3还是4*/  
MPI_Wait(...);
```

- send完成并不意味着receive也完成
 - 消息可能被系统缓冲
 - 消息可能还在传输中



实现非阻塞通信的难点

○ 系统层面

- 硬件支持通信在后台完成，不需要CPU参与
- 有相应的编程接口

○ 算法层面

- 要保证与通信重叠的计算与通信无依赖关系
- 很多时候要重构算法



发送与接收的接口调用

- 标准发送：
 - `int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
 - request: 非阻塞通信对象
- 缓存发送: 用户提供缓冲区管理
 - `MPI_Ibsend`
- 同步发送: 已开始接收
 - `MPI_Issend`
- 就绪发送: 接收已开始
 - `MPI_Irsend`
- 接收:
 - `int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)`



单一非阻塞通信的完成

- `int MPI_Wait(MPI_Request *request, MPI_Status *status)`
 - 等待完成并释放该对象，返回信息放在 `status` 中
 - 若为发送操作，则发送完成，发送缓冲区可以重用
 - 若为接收操作，则数据已经接收到，放在接收缓冲区中
- 若通信尚未完成，相当于回到阻塞状态



多个非阻塞通信的完成

- `int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status *status)`
 - `count`: 非阻塞通信对象的个数
 - `array_of_requests`: 非阻塞通信完成对象数组
 - `index`: 完成对象对应的句柄索引
 - `status`: 返回状态
- 任何一个对象完成就返回



多个非阻塞通信的完成(2)

- `int MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses)`
 - `count`: 非阻塞通信对象的个数
 - `array_of_requests`: 非阻塞通信完成对象数组
 - `array_of_statuses`: 状态数组
- 所有对象均完成才返回



多个非阻塞通信的完成(3)

- `int MPI_Waitsome(int incount, MPI_Request *array_of_request, int *outcount, int *array_of_indices, MPI_Status *array_of_statuses)`
 - `incount`: 非阻塞通信对象的个数
 - `array_of_request`: 非阻塞通信对象数组
 - `outcount`: 已完成对象的数目
 - `array_of_indices`: 已完成对象的下标数组
 - `array_of_statuses`: 已完成对象的状态数组
- 只要有不为0的对象完成就返回



非阻塞通信的检测

- 并不等到通信结束，立即返回状态
- 单个
 - `int MPI_Test(MPI_Request*request, int *flag, MPI_Status *status)`
 - `flag==true`，表示指定通信操作已完成
`flag==false`，表示指定通信操作还未结束
- 多个
 - 与Wait相对应，只是增加了flag
 - `MPI_Testsome`并无flag参数



非阻塞通信对象

- 识别各种通信操作，判断相应的非阻塞操作是否完成
- 释放
 - 确认一个非阻塞通信操作完成时，可直接释放该对象所占用的资源，而不是通过调用非阻塞通信完成操作来间接地释放
 - `int MPI_Request_free(MPI_Request * request)`
 - 如果与该非阻塞通信对象相联系的通信还没有完成则该对象的资源并不会立即释放



非阻塞通信的取消

- `int MPI_Cancel(MPI_Request *request)`
 - 若取消操作调用时相应的非阻塞通信已经开始则它会正常完成
 - 也必须调用非阻塞通信的完成操作或查询对象的释放操作来释放查询对象
- `int MPI_Test_cancelled(MPI_Status status, int *flag)`
 - 如果`flag=true`则表明该通信已经被成功取消



重复非阻塞通信

- 对于会被重复执行的通信操作，MPI对其进行优化以降低不必要的通信开销
 - 1. 通信的初始化，比如：MPI_SEND_INIT
 - 2. 启动通信：MPI_START
 - 激活通信对象
 - 3. 完成通信：MPI_WAIT
 - 这一步只是将通信状态置为不活跃
 - 4. 释放查询对象：MPI_REQUEST_FREE
 - 必须显式地调用释放



重复非阻塞通信接口调用

- MPI_SEND_INIT, MPI_RECV_INIT与非阻塞发送接收接口形式类似，只是不真正启动消息通信
- `int MPI_Start(MPI_Request *request)`
- `int MPI_Startall(int count, MPI_Request *array_of_requests)`



非阻塞通信解决Jacobi迭代

//设定进程标识和迭代的初始值

//对于每一次迭代都执行下面的通信操作:

// 执行非阻塞通信，将这次计算需要的边界数据先发 to 缓存区

MPI_Isend(a, size, MPI_FLOAT, down, 10, MPI_COMM_WORLD, &req[0]);

MPI_Isend(b, size, MPI_FLOAT, up, 11, MPI_COMM_WORLD, &req[1]);

MPI_Irecv(c, size, MPI_FLOAT, down, 10, MPI_COMM_WORLD, &req[2]);

MPI_Irecv(d, size, MPI_FLOAT, up, 11, MPI_COMM_WORLD, &req[3]);

// 计算除边界以外的剩余部分，更新数组

....

//完成非阻塞通信

for(i = 0; i < 4; i++)

MPI_Wait(&req[i], &status[i]);

//计算下次迭代需要通信的边界数据

....



利用重复模式

```
//设定进程标识和迭代的初始值
//初始化非阻塞通信
MPI_Send_init(a, size, MPI_FLOAT,down,10, MPI_COMM_WORLD, &req[0]);
MPI_Send_init(b, size, MPI_FLOAT, up ,11, MPI_COMM_WORLD, &req[1]);
MPI_Recv_init(c, size, MPI_FLOAT, down, 10, MPI_COMM_WORLD, &req[2]);
MPI_Recv_init(d, size, MPI_FLOAT, up, 11, MPI_COMM_WORLD, &req4[3]);
//对于每一次迭代都执行下面的通信操作：
// 激活非阻塞通信，将下一次计算需要的数据先发 to 缓存区
MPI_Startall(4, req);
// 计算除边界以外的剩余部分，更新数组
... ..
//完成非阻塞通信
MPI_Waitall(4, req, status);
//计算下次迭代需要通信的边界数据
... ..
//迭代完成后，释放非阻塞通信对象
for(i = 0; i < 4; i++)
    MPI_Request_free(&req[i]);
```




提纲

- MPI概念和基本原语
- MPI编程模型
- 组通信
- 非阻塞通信
- 多线程混合编程



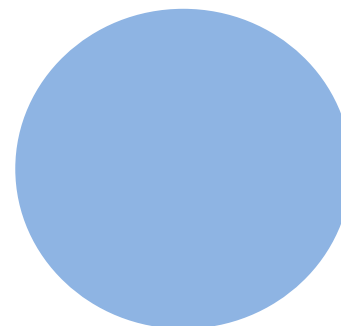
MPI和线程

- MPI描述了进程（独立地址空间）间并行
- 线程并行则是提供了一个进程内的共享内存模型
- 多核集群的常见编程方法
 - 纯MPI
 - 节点内和跨节点都采用MPI实现进程并行
 - 节点内MPI内部是采用共享内存来通信的
 - MPI + OpenMP
 - 节点内使用OpenMP，跨节点使用MPI
 - MPI + Pthreads
 - 节点内使用Pthreads，跨节点使用MPI
- 后两种方法被称为“混合编程”

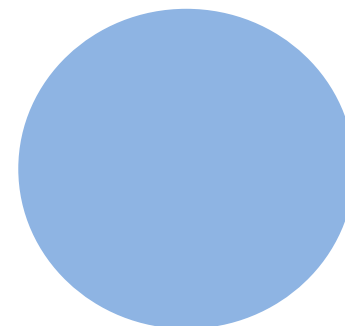
MPI + Threads混合编程

MPI-only Programming

- 在纯MPI编程中，
每个MPI进程有单一
程序计数器
- 在MPI+threads混合编程
中，可以有多个线程同时
执行



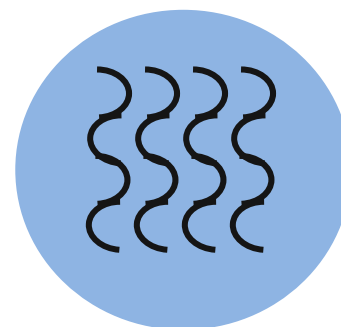
Rank 0



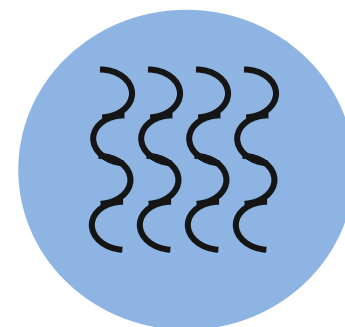
Rank 1

- 所有线程共享所有MPI对象
(通信域、请求、...)
- MPI可能需要采取措施
确保MPI栈的状态是
一致的

MPI+Threads Hybrid Programming



Rank 0



Rank 1



MPI四种线程安全级别

- MPI定义了四级线程安全级别——这其实是应用程序向MPI做出的承诺
 - `MPI_THREAD_SINGLE`: 应用中只有一个线程
 - `MPI_THREAD_FUNNELED`: 多线程, 但只有主线程会进行MPI调用 (调用`MPI_Init_thread`的那个线程)
 - `MPI_THREAD_SERIALIZED`: 多线程, 但*同时*只有一个线程会进行MPI调用
 - `MPI_THREAD_MULTIPLE`: 多线程, 且任何线程任何时候都会进行MPI调用 (有一些限制避免竞争条件)
- 线程安全级别是递增顺序的
 - 如果一个应用工作在FUNNELED模式, 它也工作在SERIALIZED模式
- MPI定义了一个MPI_Init的替代API
 - `MPI_Init_thread(requested, provided)`
 - 应用给出它希望的级别; MPI实现返回它支持的级别



MPI_THREAD_SINGLE

- 系统中无其他线程，如OMP并行区域

```
int main(int argc, char ** argv)
{
    int buf[100];

    MPI_Init(&argc, &argv);

    for (i = 0; i < 100; i++)
        compute(buf[i]);

    /* Do MPI stuff */

    MPI_Finalize();

    return 0;
}
```



MPI_THREAD_FUNNELED

○ 所有MPI调用都是主线程进行

□ 在OMP并行区域之外

□ 在OMP Master区域内

```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
    if (provided < MPI_THREAD_FUNNELED)
        MPI_Abort(MPI_COMM_WORLD, 1);

    #pragma omp parallel for
    for (i = 0; i < 100; i++)
        compute(buf[i]);

    /* Do MPI stuff */

    MPI_Finalize();
    return 0;
}
```



MPI_THREAD_SERIALIZED

- 一个时刻只有一个线程进行MPI调用

- 由OMP临界区保护

```
int main(int argc, char ** argv)
{
```

```
    int buf[100], provided;
```

```
    MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
```

```
    if (provided < MPI_THREAD_SERIALIZED)
```

```
        MPI_Abort(MPI_COMM_WORLD, 1);
```

```
    #pragma omp parallel for
```

```
        for (i = 0; i < 100; i++) {
```

```
            compute(buf[i]);
```

```
    #pragma omp critical
```

```
        /* Do MPI stuff */
```

```
    }
```

```
    MPI_Finalize();
```

```
    return 0;
```

```
}
```



MPI_THREAD_MULTIPLE

- 任何线程任何时候都可进行MPI调用（应用限制条件）

```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    if (provided < MPI_THREAD_MULTIPLE)
        MPI_Abort(MPI_COMM_WORLD, 1);

    #pragma omp parallel for
    for (i = 0; i < 100; i++) {
        compute(buf[i]);
        /* Do MPI stuff */
    }

    MPI_Finalize();

    return 0;
}
```




线程和MPI

- 并不要求一个MPI实现支持高于MPI_THREAD_SINGLE的级别；即，不要求实现是线程安全的
- 一个完全线程安全的实现会支持MPI_THREAD_MULTIPLE
- 一个调用MPI_Init（而非MPI_Init_thread）的程序应假定实现只支持MPI_THREAD_SINGLE
- 一个未调用MPI_Init_thread的多线程MPI程序是一个错误程序（很常见的错误）



MPI_THREAD_MULTIPLE规范

- **序：**当多个线程并发调用MPI API时，结果就像以某种（任意）顺序串行执行调用一样
 - 序是在每个线程内维持的
 - 程序员必须确保在相同通信域、窗口或文件上的组操作在线程间正确排序
 - 例如，不能在相同通信域上由一个线程调用广播操作、而同时另一个线程调用归约操作
 - 当一个应用内的线程进行冲突的MPI调用时，程序员应负责防止竞争条件
 - 例如，一个线程访问一个info对象、而同时另一个线程在释放它
- **阻塞：**阻塞MPI调用只会阻塞调用线程，而不会阻止其他线程的运行、也不会阻止它们调用MPI函数



MPI_THREAD_MULTIPLE中的序： 组通信错误例子

	<i>Process 0</i>	<i>Process 1</i>
Thread 1	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 2	MPI_Barrier(comm)	MPI_Barrier(comm)

- P0和P1上Bcast和Barrier的顺序可能不同
- 这里，程序员必须使用某种同步机制来确保在两个进程上两个线程的调度顺序相同
- 否则，在一个相同的通信域中，广播会与障碍匹配，在MPI中是不允许的



MPI_THREAD_MULTIPLE中的序： RMA错误例子

```
int main(int argc, char ** argv)
{
    /* Initialize MPI and RMA window */

    #pragma omp parallel for
    for (i = 0; i < 100; i++) {
        target = rand();
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, target, 0, win);
        MPI_Put(..., win);
        MPI_Win_unlock(target, win);
    }

    /* Free MPI and RMA window */

    return 0;
}
```

不同线程可能锁住相同进程，导致对相同target，在第一个锁解锁之前有多个锁锁住



MPI_THREAD_MULTIPLE中的序： 对象管理错误例子

	<i>Process 0</i>	<i>Process 1</i>
Thread 1	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 2	MPI_Comm_free(comm)	MPI_Comm_free(comm)

- 程序员应确保当一个线程在释放一个对象时，另一个线程不会使用它
 - 这本质上是一个顺序问题；对象可能在使用之前就释放了



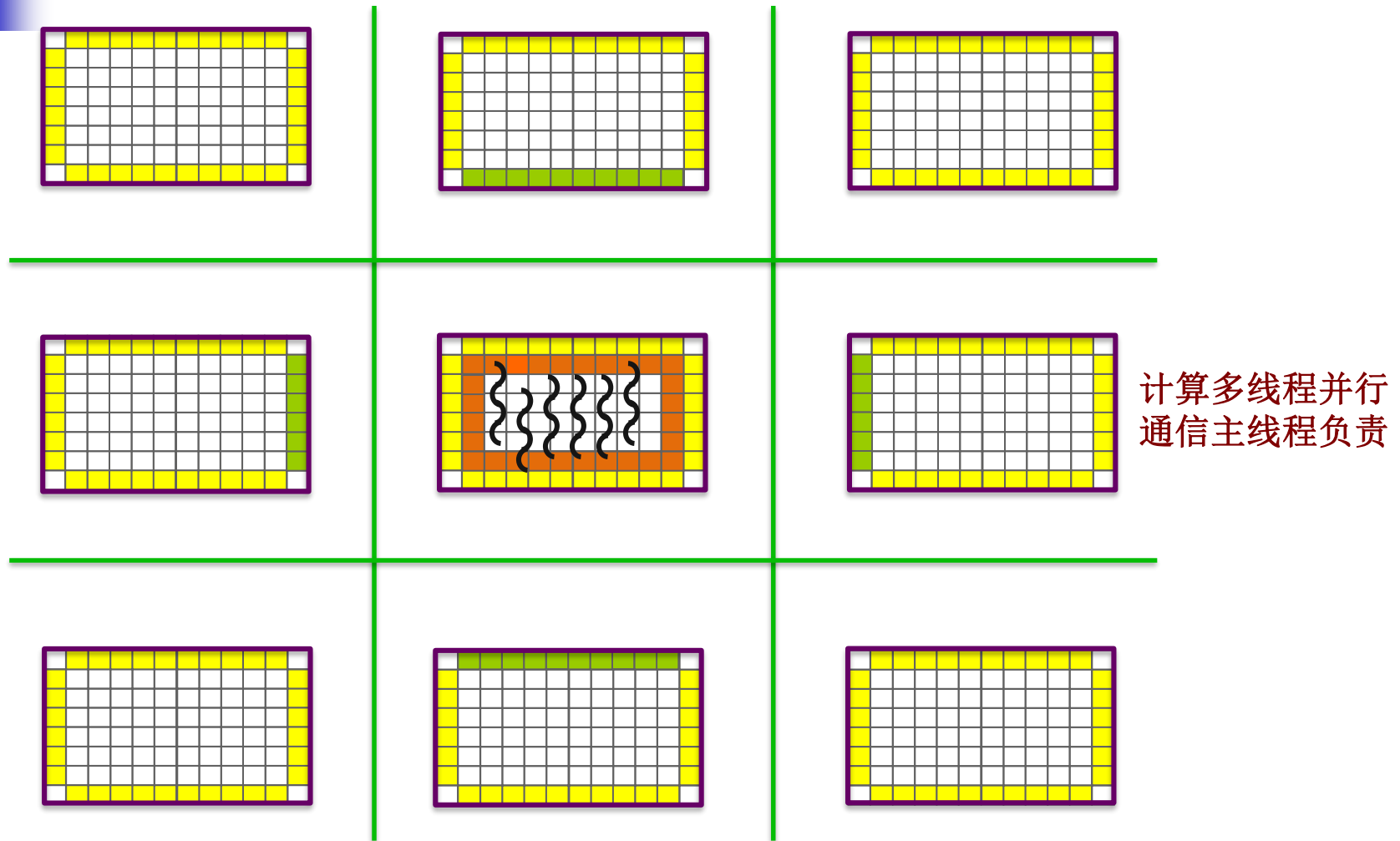
MPI_THREAD_MULTIPLE中的阻塞调用：正确例子

	<i>Process 0</i>	<i>Process 1</i>
Thread 1	MPI_Recv(src=1)	MPI_Recv(src=0)
Thread 2	MPI_Send(dst=1)	MPI_Send(dst=0)

- 一个MPI实现必须确保这个例子永远不会发生死锁，无论线程执行顺序是怎样的
- 这意味着一个实现在一个MPI函数内不能简单申请一个线程锁然后阻塞，它必须释放锁以允许其他线程能前进

MPI_THREAD_FUNNELED实现

Jacob迭代





Jacob迭代Funneled例

```
int main(int argc, char ** argv)
{
    ...
    /* initialize MPI environment */
    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
    if (provided < MPI_THREAD_FUNNELED)
        MPI_Abort(MPI_COMM_WORLD, 1);

    ...
    /* create north-south datatype */
    MPI_Datatype north_south_type;
    MPI_Type_contiguous(bx, MPI_DOUBLE, &north_south_type);
    MPI_Type_commit(&north_south_type);

    /* create east-west type */
    MPI_Datatype east_west_type;
    MPI_Type_vector(by, 1, bx+2, MPI_DOUBLE, &east_west_type);
    MPI_Type_commit(&east_west_type);}
```




Jacob迭代Funneled例

```
...
for (iter = 0; iter < niters; ++iter) {
    ...
    /* exchange data with neighbors */
    MPI_Request reqs[8];
    MPI_Isend(&aold[ind(1,1)] /* north */, 1, north_south_type, north, 9, MPI_COMM_WORLD, &reqs[0]);
    MPI_Isend(&aold[ind(1,by)] /* south */, 1, north_south_type, south, 9, MPI_COMM_WORLD, &reqs[1]);
    MPI_Isend(&aold[ind(bx,1)] /* east */, 1, east_west_type, east, 9, MPI_COMM_WORLD, &reqs[2]);
    MPI_Isend(&aold[ind(1,1)] /* west */, 1, east_west_type, west, 9, MPI_COMM_WORLD, &reqs[3]);
    MPI_Irecv(&aold[ind(1,0)] /* north */, 1, north_south_type, north, 9, MPI_COMM_WORLD, &reqs[4]);
    MPI_Irecv(&aold[ind(1,by+1)] /* south */, 1, north_south_type, south, 9, MPI_COMM_WORLD, &reqs[5]);
    MPI_Irecv(&aold[ind(bx+1,1)] /* west */, 1, east_west_type, east, 9, MPI_COMM_WORLD, &reqs[6]);
    MPI_Irecv(&aold[ind(0,1)] /* east */, 1, east_west_type, west, 9, MPI_COMM_WORLD, &reqs[7]);
    MPI_Waitall(8, reqs, MPI_STATUS_IGNORE);

    /* update grid points */
    update_grid(bx, by, aold, anew, &heat);
    ...
}
...
```



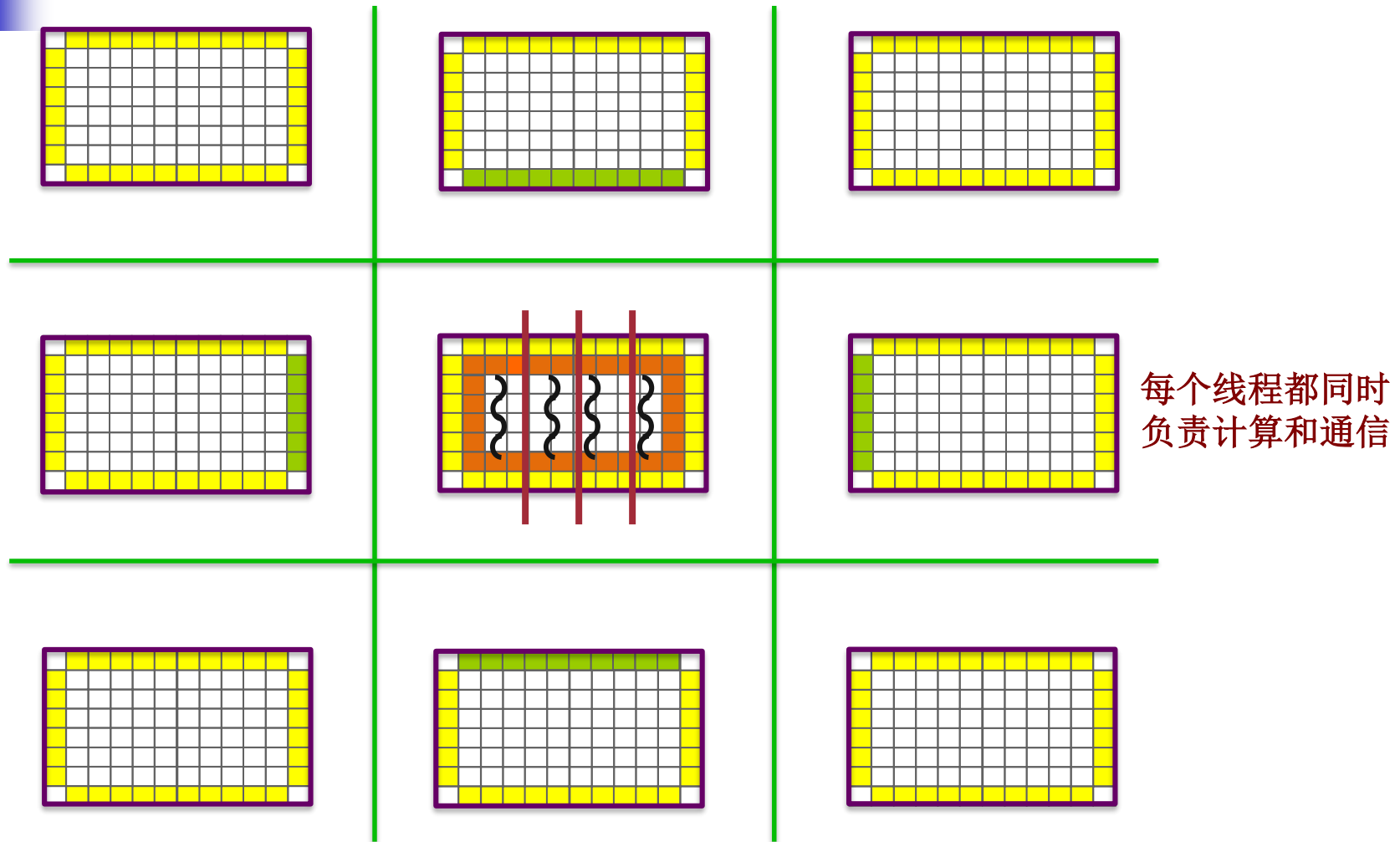
Jacob迭代Funneled例

```
void update_grid (int bx, int by, double *aold, double *anew, double *heat_ptr)
{
    int i, j;
    double heat = 0.0;

    for (i = 1; i < bx+1; ++i) {
        #pragma omp parallel for reduction(+:heat)
        for (j = 1; j < by+1; ++j) {
            anew[ind(i,j)] = anew[ind(i,j)]/2.0 + (aold[ind(i-1,j)] + aold[ind(i+1,j)] +
            aold[ind(i,j-1)] + aold[ind(i,j+1)])/4.0/2.0;
            heat += anew[ind(i,j)];
        }
    }

    (*heat_ptr) = heat;
}
```

MPI_THREAD_Multiple实现Jacob 迭代





Jacob迭代 Multiple例

```
int main(int argc, char ** argv)
{
    ...
    /* initialize MPI environment */
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    if (provided < MPI_THREAD_MULTIPLE)
        MPI_Abort(MPI_COMM_WORLD, 1);

    ...
    /* create north-south datatype */
    MPI_Datatype north_south_type;
    MPI_Type_contiguous(bx, MPI_DOUBLE, &north_south_type);
    MPI_Type_commit(&north_south_type);

    /* create east-west type */
    MPI_Datatype east_west_type;
    MPI_Type_vector(by, 1, bx+2, MPI_DOUBLE, &east_west_type);
    MPI_Type_commit(&east_west_type);}
```



Jacob迭代 Multiple例

```
...
for (iter = 0; iter < niters; ++iter) {
    ...
    #pragma omp parallel private(i,j) reduction(+:heat)
    {
        ...
        /* exchange data with neighbors */
        if (south >= 0) {
            MPI_Isend(&aold[ind(xstart,by)] /* south */, 1, north_south_type,
                      south, 9, MPI_COMM_WORLD, &south_reqs[0]);
            MPI_Irecv(&aold[ind(xstart,by+1)] /* south */, 1, north_south_type,
                      south, 9, MPI_COMM_WORLD, &south_reqs[1]);
            MPI_Waitall(2, south_reqs, MPI_STATUSES_IGNORE);
        }
        ...
        /* update grid */
        for (i = xstart; i < xend; ++i) {
            for (j = 1; j < by+1; ++j) {
                anew[ind(i,j)] = anew[ind(i,j)]/2.0 + (aold[ind(i-1,j)] + aold[ind(i+1,j)]
                + aold[ind(i,j-1)] + aold[ind(i,j+1)])/4.0/2.0;
                heat += anew[ind(i,j)];
            }
        }
    } /* end parallel region */
    ...
}
```



MPI实现对多线程的支持

- 所有MPI实现均支持MPI_THREAD_SINGLE
- 可能支持MPI_THREAD_FUNNELED即使不承认
 - 不需要线程安全的malloc
 - 在OpenMP程序中可能是OK的
- 很多（但不是所有）实现支持THREAD_MULTIPLE
 - 虽然高效实现很困难（锁粒度问题）
- “容易的” OpenMP程序（循环并行）只需FUNNELED
 - 因此很多混合程序不需要“线程安全” MPI
 - 但要注意Amdahl定律！



MPI_THREAD_MULTIPLE性能

- 线程安全不是免费的
- MPI实现必须用互斥量或临界区保护特定数据结构或代码片段
- 衡量性能影响：测试多线程 vs. 多进程的通信性能
 - Thakur/Gropp的论文：“Test Suite for Evaluating Performance of Multithreaded MPI Communication,” *Parallel Computing*, 2009



为何MPI_THREAD_MULTIPLE优化困难

- MPI内部要维护多种资源
- 由MPI语义，要求所有线程都可访问一些数据结构
 - 例如，thread 1可能发起一个Irecv，而thread 2可能等待其完成 – 因此请求队列必须被两个线程共享
 - 由于多线程访问此共享队列，就需要对其加锁 – 产生显著的额外开销



混合编程：正确性要求

- MPI+threads混合编程对降低多线程编程复杂性没有什么帮助
 - 程序还必须是一个正确的多线程程序
 - 在此之上，你还必须确保正确遵循MPI语义
- 有很多商用调试工具支持MPI+threads混合程序的调试（大多用于MPI+Pthreads和MPI+OpenMP）



一个实际例子

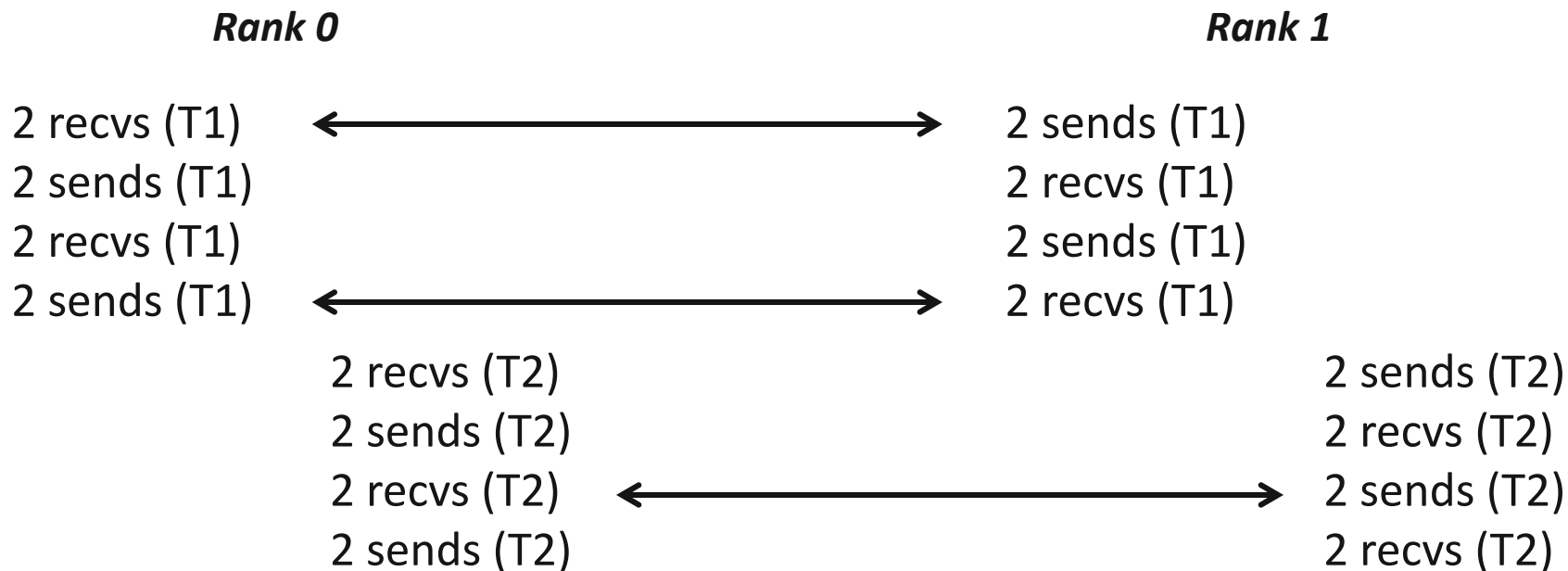
- 一个非常简单的多线程MPI程序锁死
- 2个进程
- 每个线程2个线程
- 两个线程都与其他进程的线程通信
- 调试花了很长时间，最终发现bug其实在用户程序中



2进程、2线程， 每个线程代码

```
for (j = 0; j < 2; j++) {  
    if (rank == 1) {  
        for (i = 0; i < 2; i++)  
            MPI_Send(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);  
        for (i = 0; i < 2; i++)  
            MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);  
    }  
    else { /* rank == 0 */  
        for (i = 0; i < 2; i++)  
            MPI_Recv(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &stat);  
        for (i = 0; i < 2; i++)  
            MPI_Send(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD);  
    }  
}
```

期望的操作顺序



- 每个send与另一个进程的receive匹配

实际可能的顺序

Rank 0

2 recvs (T1)
2 sends (T1)
1 recv (T1)

1 recv (T2)

1 recv (T1)

1 recv (T2)

2 sends (T1)

2 sends (T2)

2 recvs (T2)

2 sends (T2)

Rank 1

2 sends (T1)
1 recv (T1)

2 sends (T2)

1 recv (T2)

1 recv (T1)

1 recv (T2)

2 sends (T1)

2 sends (T2)

2 recvs (T1)

2 recvs (T2)

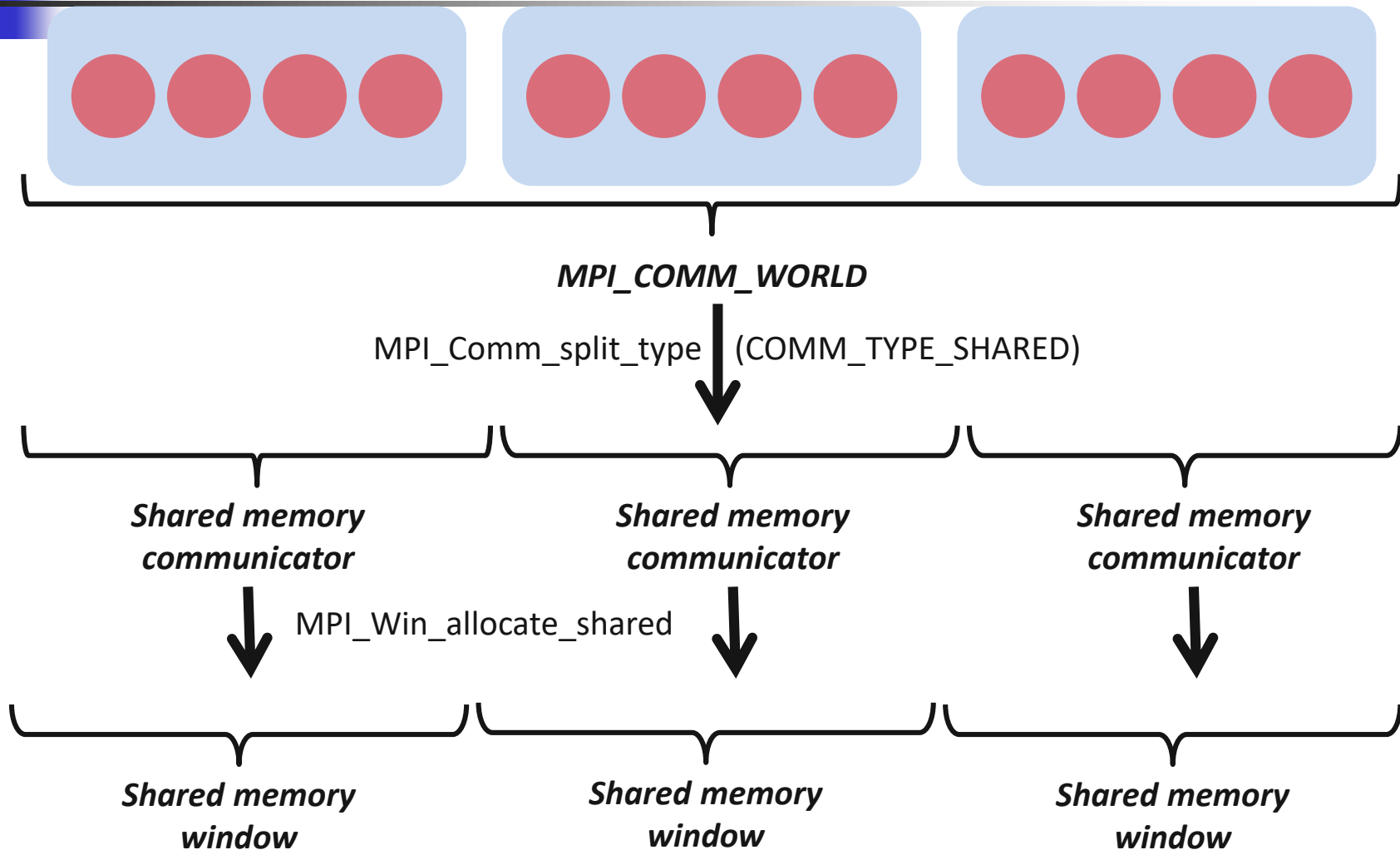
- 由于不同线程执行MPI操作的顺序是任意的，所以可能所有线程都阻塞于recv



使用共享内存的混合编程

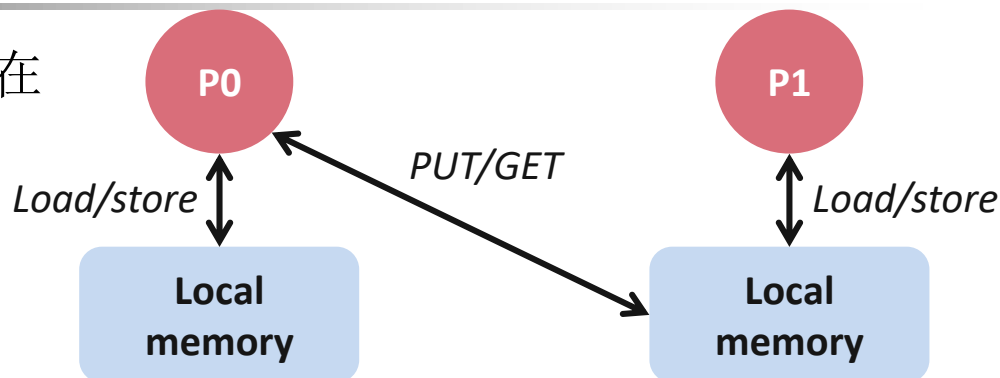
- MPI-3允许不同进程分配共享内存
 - `MPI_Win_allocate_shared`
- 使用了很多单边通信的概念
- 应用程序可以通过MPI进行混合编程或在共享内存窗口上load/store访问
- 可使用其他MPI同步对共享内存的访问
- 比多线程编程更为简单

在MPI中创建共享内存

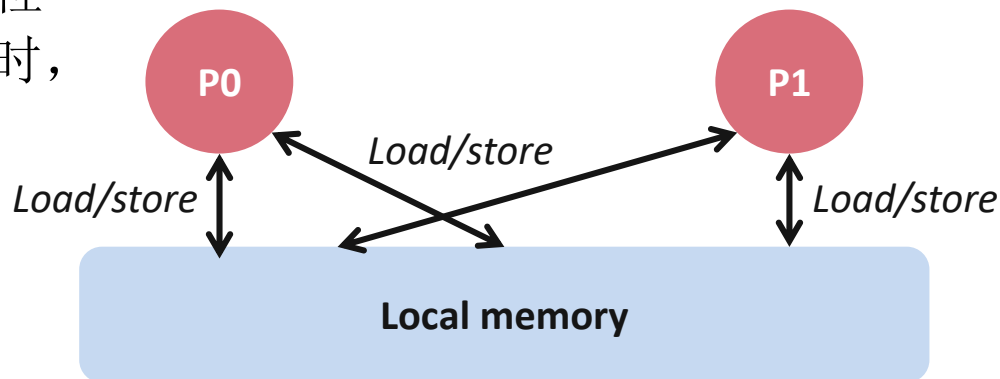


常规RMA窗口 vs. 共享内存

- 共享内存窗口允许进程在所有窗口内存上直接执行load/store操作
 - E.g., $x[100] = 10$
- 所有RMA函数也都可以用于这种内存，包括更高级的语义，如原子操作
- 如果进程只是想使用线程访问节点上的所有内存时，共享内存很有用
 - 你可以创建一个共享内存窗口，将共享数据存入其中



Traditional RMA windows



Shared memory windows



内存分配和放置

- 共享内存分配无需均匀分布在所有进程
 - 进程可以分配不同大小的内存（甚至零内存）
- MPI标准不指明内存放置在哪里（**pin**到哪块物理内存）
 - MPI实现可选择自己的策略，虽然我们期望它们努力将一个进程分配的内存放置到“靠近它”的位置
- 一个通信域中分配的全部共享内存默认连续
 - 用户可以传递一个称为“**noncontig**” info暗示，允许MPI实现将每个进程分配的内存对齐到恰当的边界，以帮助放置



共享内存窗口实现共享数组

```
int main(int argc, char ** argv)
{
    int buf[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_split_type(..., MPI_COMM_TYPE_SHARED, .., &comm);
    MPI_Win_allocate_shared(comm, ..., &win);

    MPI_Win_lockall(win);

    /* copy data to local part of shared memory */
    MPI_Win_sync(win);

    /* use shared memory */

    MPI_Win_unlock_all(win);

    MPI_Win_free(&win);
    MPI_Finalize();
    return 0;
}
```