主要内容

1. MPI简介及环境搭建
2. MPI点对点通信
3. MPI集合通信
4. **MPI 派生数据类型**
5. **MPI程序的性能评估**
6. MPI实例——并行排序算法
7. 作业

太原理工大学
TAIYUAN UNIVERSITY OF TECHNOLOGY

MPI派生数据类型、程序性能评估及实例

岳俊宏

E-mail:yuejunhong@tyut.edu.cn; Tel:18234095983

目录

- MPI 派生数据类型
- MPI程序的性能评估


MPI 派生
数据类型

思考

- 在分布式系统中，通信与本地计算开销耗时情况；
- 用多条消息发送一定数量的数据与只用一条消息发送等量数据。

```
double x[1000];
. . .
if (my_rank == 0)
   for (i = 0; i < 1000; i++)
      MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);
else /* my_rank == 1 */
   for (i = 0; i < 1000; i++)
      MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0, comm, &status);

if (my_rank == 0)
   MPI_Send(x, 1000, MPI_DOUBLE, 1, 0, comm);
else  /* my_rank == 1 */
   MPI_Recv(x, 1000, MPI_DOUBLE, 0, 0, comm, &status);
```

思考

- MPI整合多条消息数据的方式：
  - ✓不同通信函数中的count参数
  - ✓派生数据类型
  - ✓MPI_Pack/Unpack函数

创建派生数据类型的方法

- 在MPI中，通过同时存储数据项的类型及它们在内存中的相对位置，**派生数据类型**可以用于表示内存中数据项的任意集合。
- **主要思想**：如果发送数据的函数知道数据项的类型及内存中数据项集合的相对位置，就可以在数据项被发送出去之前在内存中将数据项聚集起来。
- **接收函数**可以在数据项被接收后将数据项分发到它们在内存中正确的目标地址上。

实例——输入数据的广播

梯形积分：调用MPI_Bcast函数三次；

建立单独的派生数据类型：调用MPI_Bcast函数一次；

派生数据类型是由一系列的MPI基本数据类型和每个数据类型的偏移所组成的。

Variable	Address
a	24
b	40
n	48

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

每一对数据项的第一个元素表明数据类型，第二个元素是该数据项相对于起始位置的偏移。

创建派生数据类型的方法

MPI_Type_create_struct

- 可以用这个函数创建由不同基本数据类型的元素所组成的派生数据类型.

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

创建派生数据类型的方法

MPI_Get_address

- 用这个函数可以找到相关地址的值.
- 参数 MPI_Aint 是特殊的存储地址的类型.

```
int MPI_Get_address(
      void*       location_p   /* in  */,
      MPI_Aint*   address_p    /* out */);
```

创建派生数据类型的方法

MPI_Type_commit

- 允许 MPI 在通信函数内部使用新的数据类型，并优化数据类型内部表征.

```
int MPI_Type_commit(MPI_Datatype*  new_mpi_t_p  /* in/out */);
```

创建派生数据类型的方法

MPI_Type_free

- 结束新数据类型使用后，可以释放因此产生的存储空间

```
int MPI_Type_free(MPI_Datatype*  old_mpi_t_p  /* in/out */);
```

建立派生数据类型

```
void Build_mpi_type(
      double*          a_p              /* in  */,
      double*          b_p              /* in  */,
      int*             n_p              /* in  */,
      MPI_Datatype*    input_mpi_t_p    /* out */) {

   int array_of_blocklengths[3] = {1, 1, 1};
   MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
   MPI_Aint a_addr, b_addr, n_addr;
   MPI_Aint array_of_displacements[3] = {0};
```

建立派生数据类型

```
    MPI_Get_address(a_p, &a_addr);
    MPI_Get_address(b_p, &b_addr);
    MPI_Get_address(n_p, &n_addr);
    array_of_displacements[1] = b_addr-a_addr;
    array_of_displacements[2] = n_addr-a_addr;
    MPI_Type_create_struct(3, array_of_blocklengths,
          array_of_displacements, array_of_types,
          input_mpi_t_p);
    MPI_Type_commit(input_mpi_t_p);
}   /* Build_mpi_type */
```

建立派生数据类型

```
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
      int* n_p) {
   MPI_Datatype input_mpi_t;

   Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

   MPI_Type_free(&input_mpi_t);
}  /* Get_input */
```

作业

编写一个完整的并行梯形积分（要求： uses collective communications and MPI derived datatypes to distribute the input data and compute the global sum.）


MPI程序
性能评估

提纲

一 计时功能

二 同步路障

三 加速比和效率

四 MPI程序的安全性

一用MPI实现计时功能

1、MPI_Wtime调用的说明原型：

double MPI_Wtime(void)

功能：返回一个用浮点数表示的秒数，从某一时刻到调用时刻所经历的时间(s)

注意：调用产生结果的单位为秒，需要用户自己转换成其它时间单位。

一 用MPI实现计时功能

2、MPI_Wtime函数的用法

```
double start_time,end_time,total_time;
…… (变量说明、初始化与其它计算)
start_time=MPI_Wtime();
…….(需要计时的部分)
end_time=MPI_Wtime();
total_time=end_time-start_time;
Printf(“It  tooks %f  seconds\n”,total_time）;
```

一用MPI实现计时功能

3、MPI_Wtick调用的说明原型：

double MPI_Wtick(void)

功能：返回计算机时钟滴答一下所占用的时间(s){计算机所能分辨的最小时间}

程序示例

```
……(包含头文件 )
int main(int argc, char **argv)
{  double t1,t2,tick;    int  rank;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   t1=MPI_Wtime();       /*获取起始时间*/
   Sleep(1000);          /*系统睡眠一秒钟*/
   t2=MPI_Wtime();       /*获取终止时间*/
   tick=MPI_Wtick();    /*获取本机的时间分辨率*/
   printf("id=%d  totaltime=%f  tick=%10e2\n",rank,t2-t1,tick);
   MPI_Finalize();
}
```

二 同步路障

函数原型：

int MPI_Barrier(MPI_Comm comm)

说明：参数comm是输入

调用

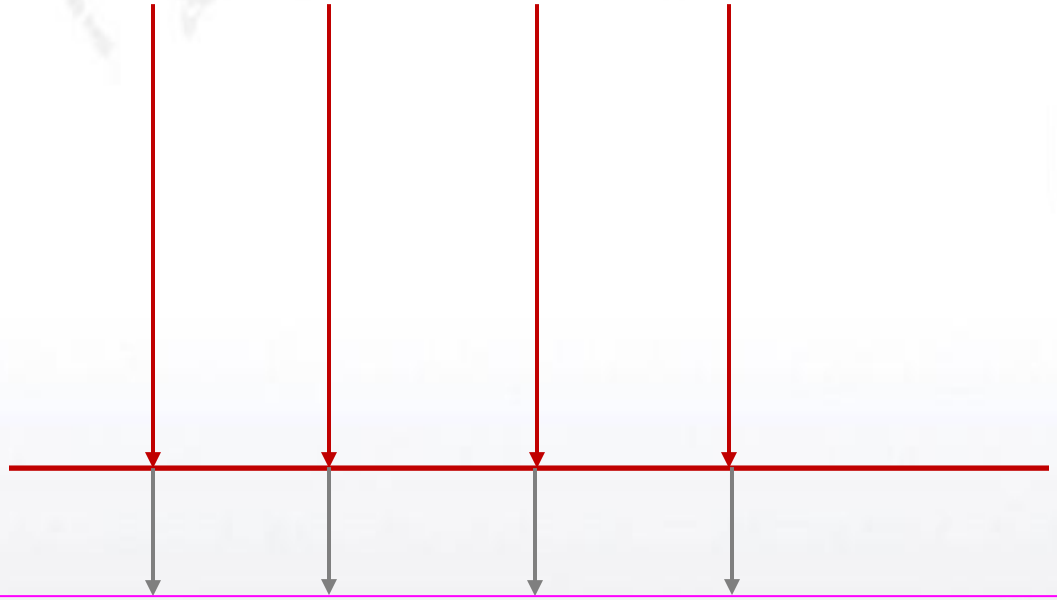
MPI_Barrier(comm)

Ensures that no process will return from calling it until every process in the communicator has started calling it.

二同步路障

功能：实现通信的同步功能

阻塞所有的进程，直到所有进程都调用了它



快的进程必须等待慢的进程，直到所有进程都执行到该语句后才可以向下进行

程序示例

```
double local_start, local_finish,local_elapsed,elapsed;
…
MPI_Barrier(comm);
local_start=MPI_Wtime();
/* Code to be timed */
local_finish=MPI_Wtime();
local_elapsed=local_finish-local_start;
MPI_Reduce(&local_elapsed,&elapsed,1,MPI_DOUBLE,
        MPI_MAX,0,comm);
if (my_rank ==0)
printf("Elapsed time = %e seconds\n",elapsed)
```

扩展：获取MPI版本号、机器名

调用说明

int MPI_Get_version(int *version, int *subversion）

OUT version MPI的主版本号

OUT subversion MPI的次版本号

功能：返回MPI的版本号

int MPI_Get_processor_name(char *name, int *resultlen);

/*得到机器名*/

MPI_Get_processor_name(processor_name,&namelen);

程序示例

```
#include <stdio.h>
#include <mpi.h>
void main(int argc, char **argv)
{       char name[MPI_MAX_PROCESSOR_NAME];
        int resultlen,version,subversion;
        MPI_Init(&argc,&argv);
        MPI_Get_processor_name(name,&resultlen);
        MPI_Get_version(&version,&subversion);
        printf("name=%s  version=%d  subversion=%d\n",
        name, version, subversion);
        MPI_Finalize();
}
```

三 加速比和效率

- **comm_sz固定**，n增大，那么矩阵的大小和程序的运行时间也会增加；
  - ✓ comm_sz 较小，n增大1倍，运行时间变为原来的4倍；
  - ✓ comm_sz 较大，公式不成立。
- **n固定**，comm_sz增大，运行时间减少
  - ✓ n较大：comm_sz增大，大约能减少一半的时间
  - ✓ n较小：comm_sz增大，效果甚微

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

矩阵向量乘法的串行和并行程序的运行时间

(Seconds)

三 加速比和效率

- 串行程序的运行时间与对应并行程序的运行时间的关系：

$$T_{并行}(n,p) = T_{串行}(n)/p + T_{开销}$$

- 在MPI程序中，并行计算的开销一般来自通信，也会受n和p的影响。
- p较小，n较大：公式中起主导地位的是 $T_{串行}(n)/p$
- n较小，p较大：公式中起主导地位的是 $T_{开销}$

加速比

- **加速比**是用来衡量串行运算和并行运算时间之间的关系，表示为串行时间与并行时间的比值

$$S(n,p)=\frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n,p)}$$

- S(n,p)=p时，称为线性加速比。

并行矩阵向量乘法的加速比

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

- 在p较小、n较大时，获得近似线性加速比；
- 在p较大、n较小时，加速比远远小于p。

效率

- **效率**也是评价并行性能的重要指标之一，它其实是每个进程的加速比：

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n,p)}$$

- 线性加速比相当于并行效率p/p=1.
- 通常效率都小于1.

并行矩阵向量乘法的效率

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

- 在p较小、n较大时，获得近似线性的效率；
- 在p较大、n较小时，远远达不到线性效率。

概念

- 可扩展性

如果问题的规模以一定的速率增大，但效率没有随着进程数的增加而降低。

- 实例

A程序比B程序更有扩展性！

A程序：效率恒定为0.75

B程序：效率为n/(625p), p>=2;1000<=n<<625p;

- ✓ n=1000,p=2:效率为0.8
- ✓ n=1000, p=4:效率为0.4
- ✓ n=2000,P=4:效率为0.8

概念

- 强可扩展性

  若程序可以在不增加问题规模的前提下维持恒定效率；

- 弱可扩展性

  当问题规模增加，通过增大进程数来维持程序效率的。

- 矩阵向量乘法

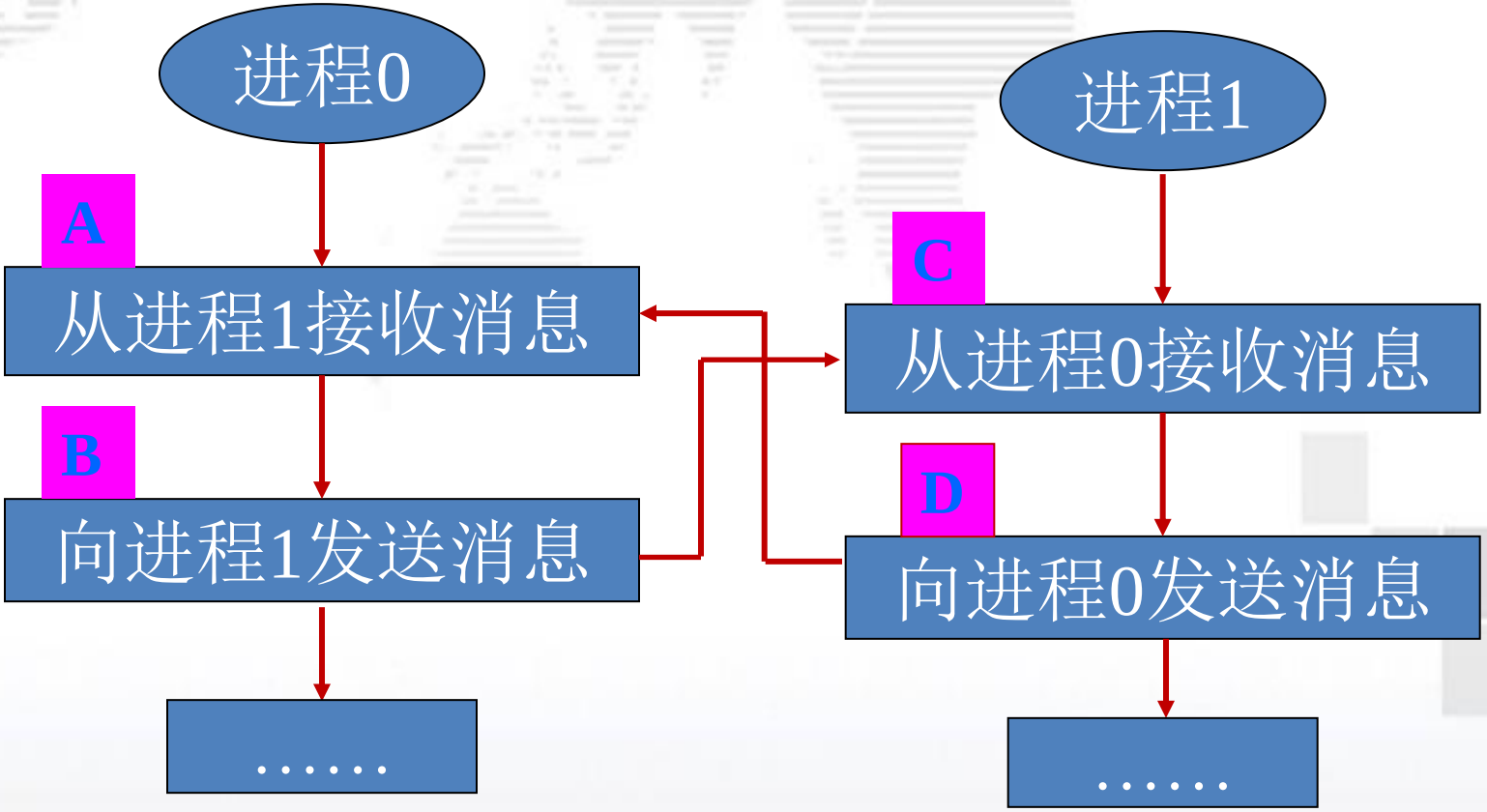
comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

四 MPI程序的安全性

假设程序启动时共产生两个进程

```
comm=MPI_COMM_WORLD;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
If(rank==0)
{   MPI_Recv(x2 , 3 , MPI_INT,1,tag,comm,&status);
    MPI_Send(x1 , 3 , MPI_INT,1,tag,comm);  }
If(rank==1)
{   MPI_Recv(x1, 3 ,MPI_INT,0,tag,comm,&status);
    MPI_Send(x2, 3 ,MPI_INT,0,tag,comm);  }
……
```

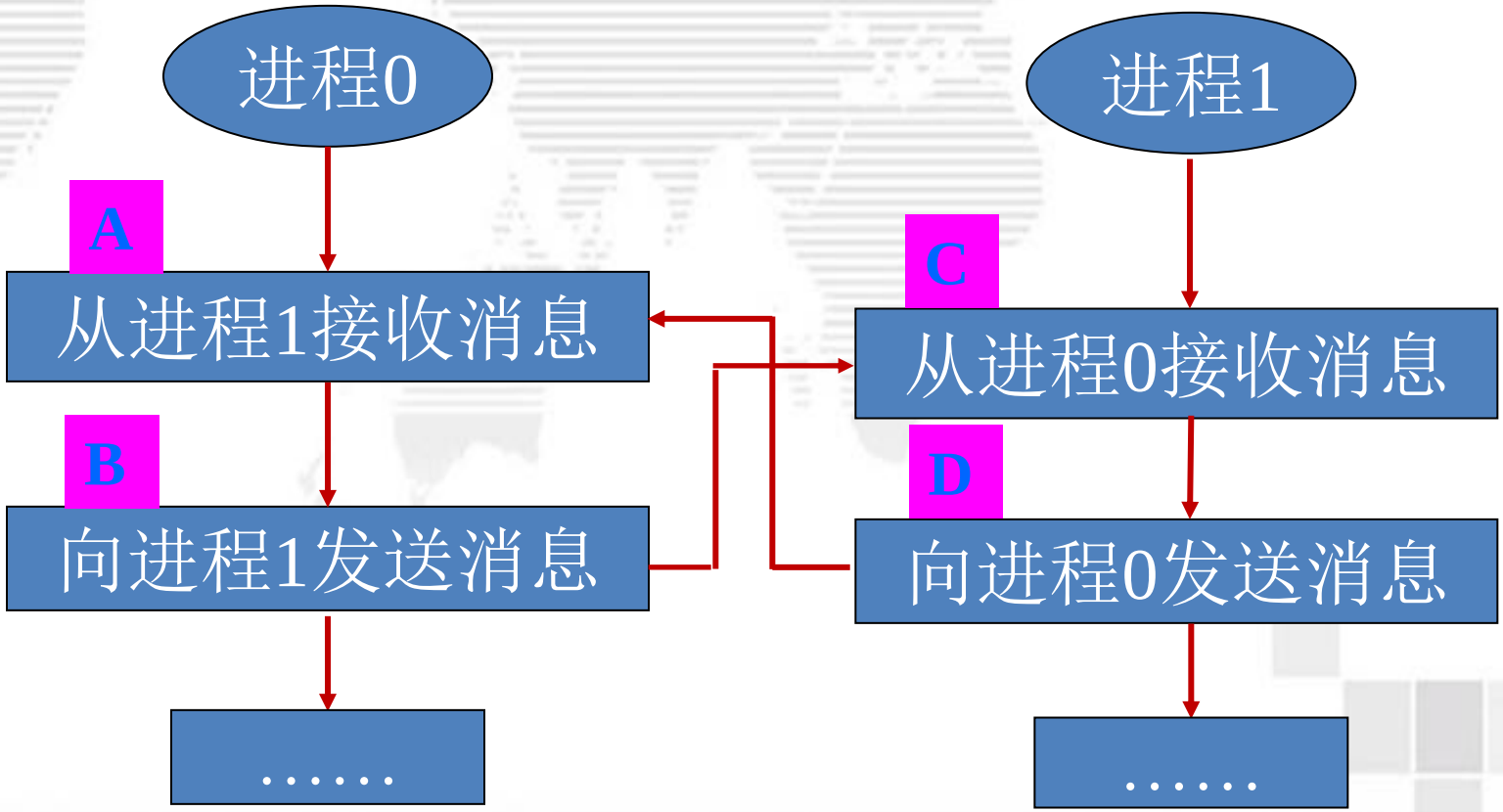
程序执行步骤图

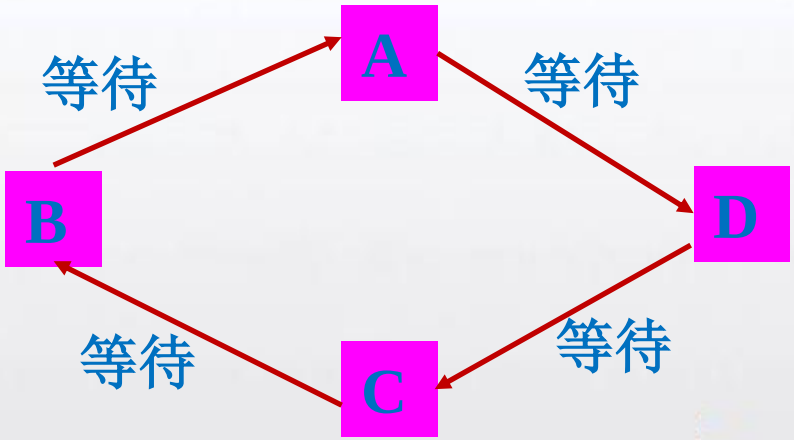


程序的预期目标：1、两个进程分别从对方接收一个消息，

2、向对方发送一个消息。

这个程序能够完成上述任务吗？





这样的程序能够正确执行吗?

消息死锁！

怎么改?

MPI程序的安全性

```
comm=MPI_COMM_WORLD;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
If(rank==0)
{   MPI_Recv(x2 , 3 , MPI_INT,1,tag,comm,&status);
    MPI_Send(x1 , 3 , MPI_INT,1,tag,comm);  }
If(rank==1)
{   MPI_Recv(x1, 3 ,MPI_INT,0,tag,comm,&status);
    MPI_Send(x2, 3 ,MPI_INT,0,tag,comm);  }
……
```

两个进程都是先接收，后发送。

没有发送，何谈接收?

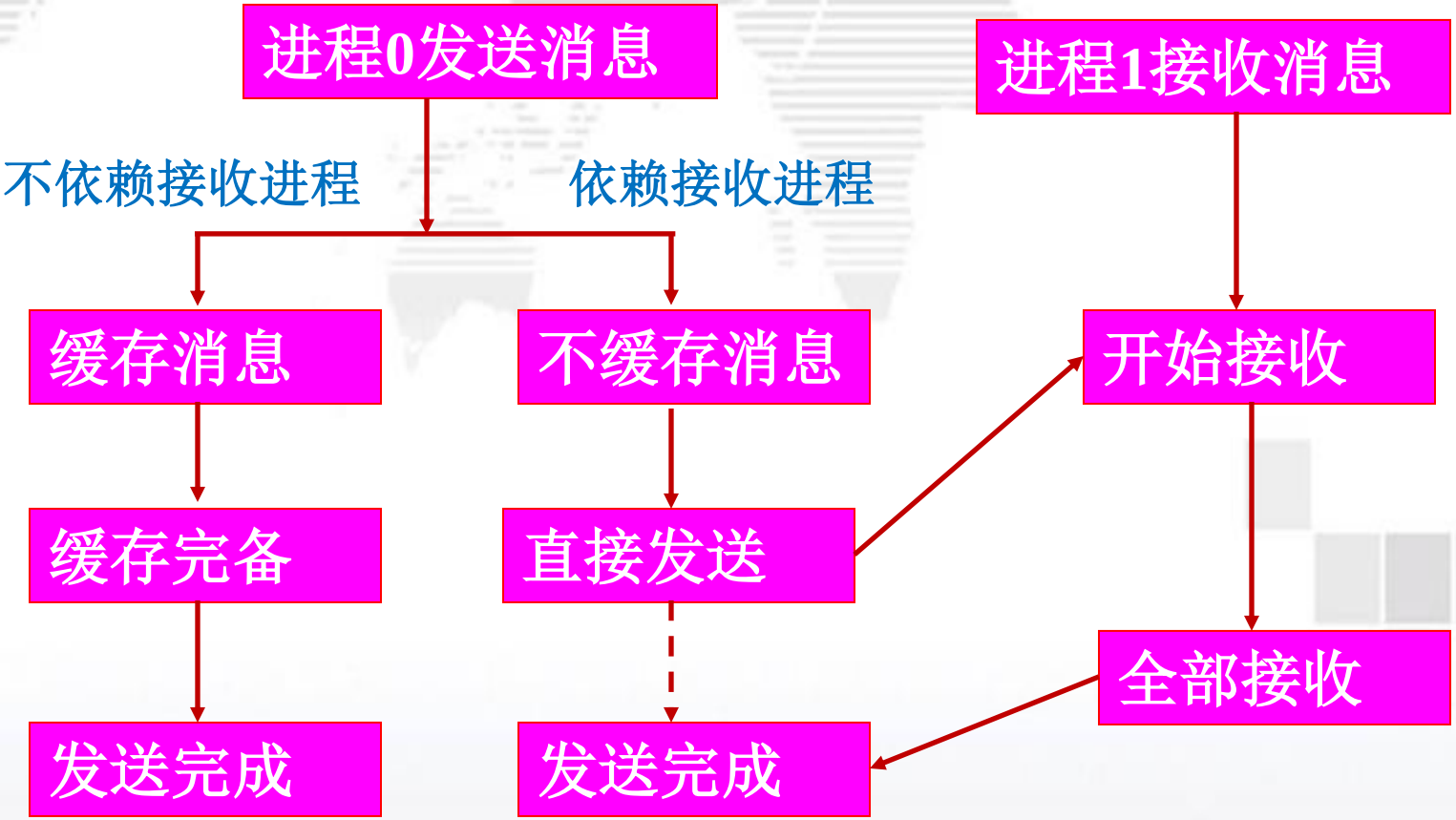
不合逻辑!

改成：先发送，后接收

```
comm=MPI_COMM_WORLD
MPI_Comm_rank(MPI_COMM_WORLD,&rank)
If(rank==0)
{   MPI_Send(sendbuf , 3 ,MPI_INT , 1 ,tag , comm);
    MPI_Recv(recbuf , 3 , MPI_INT , 1, tag , comm , &status);
}
If(rank==1)
{   MPI_Send(sendbuf , 3 , MPI_INT , 0 , tag , comm);
   MPI_Recv(recbuf, 3 , MPI_INT , 0 , tag ,comm , &status);
}
```

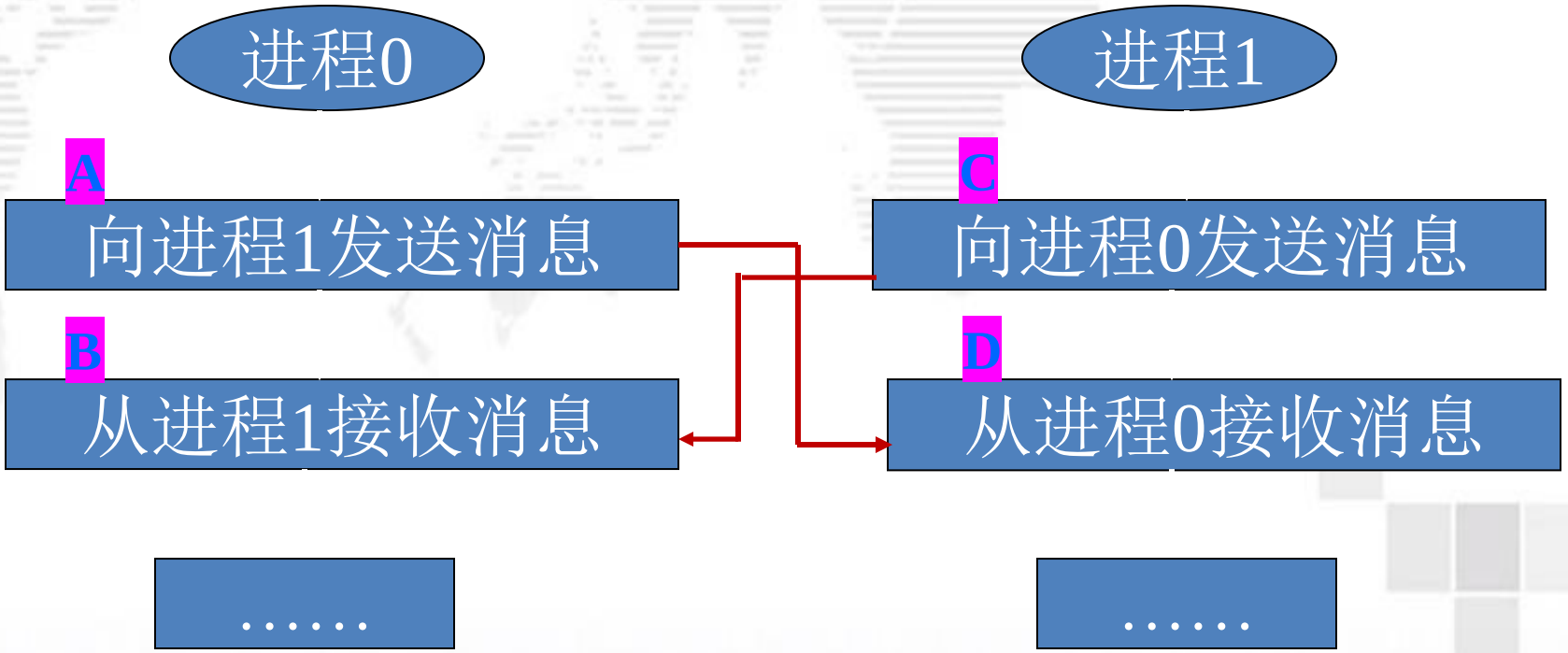
这个程序的性能如何?

发送与接收操作的执行过程



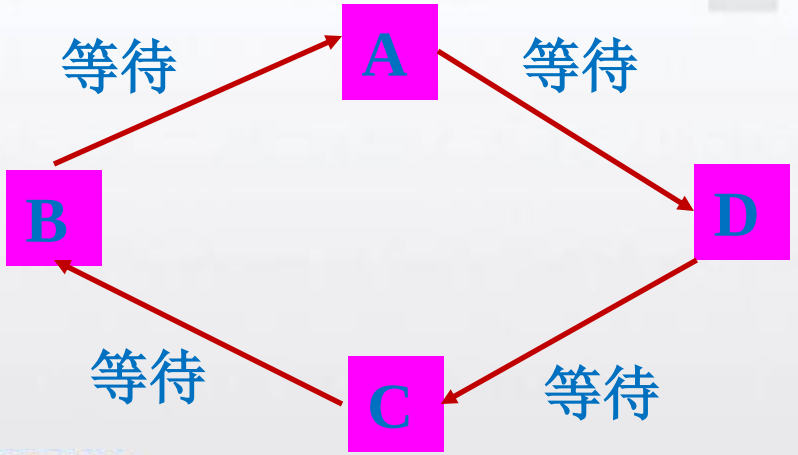
- ✓ 是否缓存数据由MPI决定；
- ✓ 如果MPI决定缓存该数据，则发送操作可正确返回而不要求接收操作收到发送的数据；
- ✓ 缓存数据要付出代价，缓冲区并非总是可以得到的。

修改后程序的执行步骤

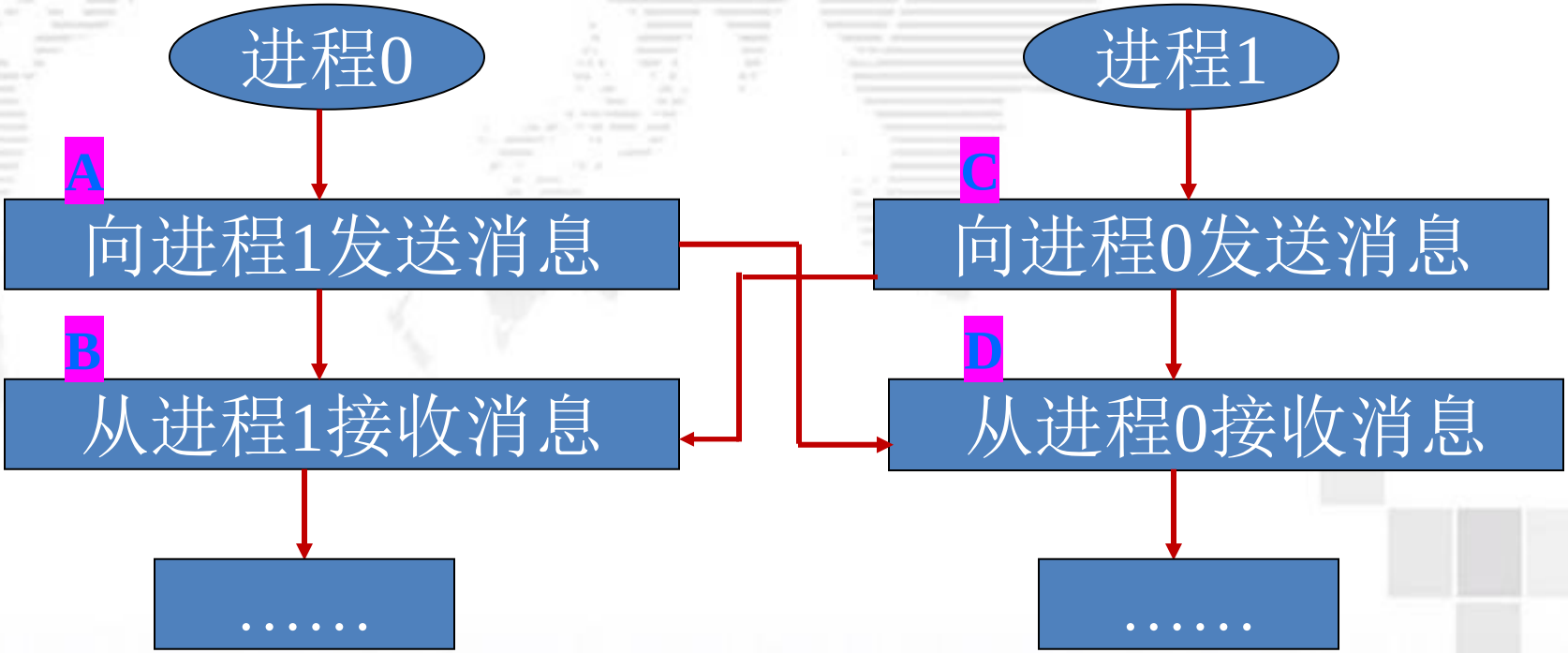


如果系统不缓存消息：

消息死锁!



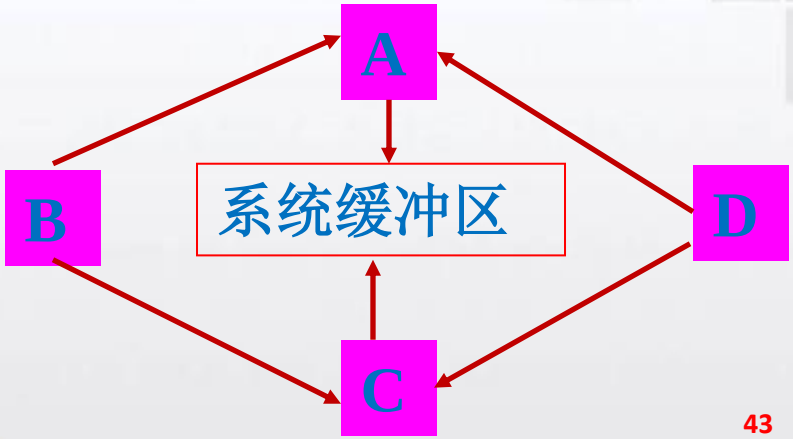
修改后程序的执行步骤



如果系统缓存消息：

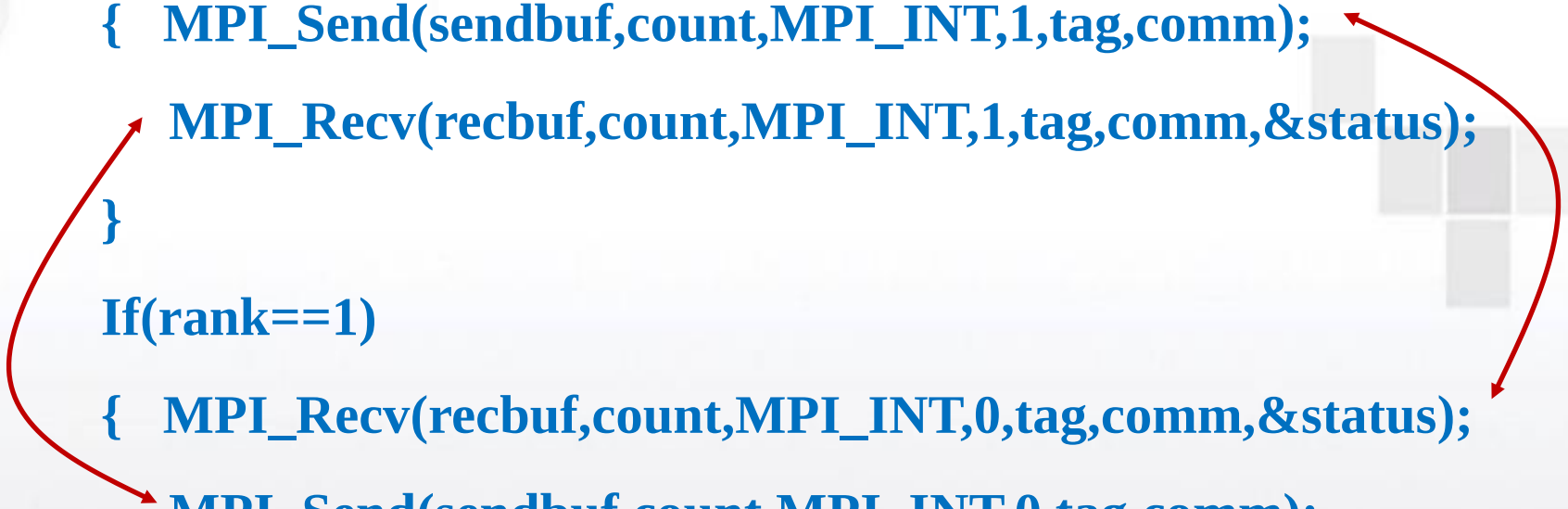
- ✓ A，C两个发送操作都需要系统缓冲区，如果系统缓存不足，消息传递将无法完成。

不安全的通信调用次序！

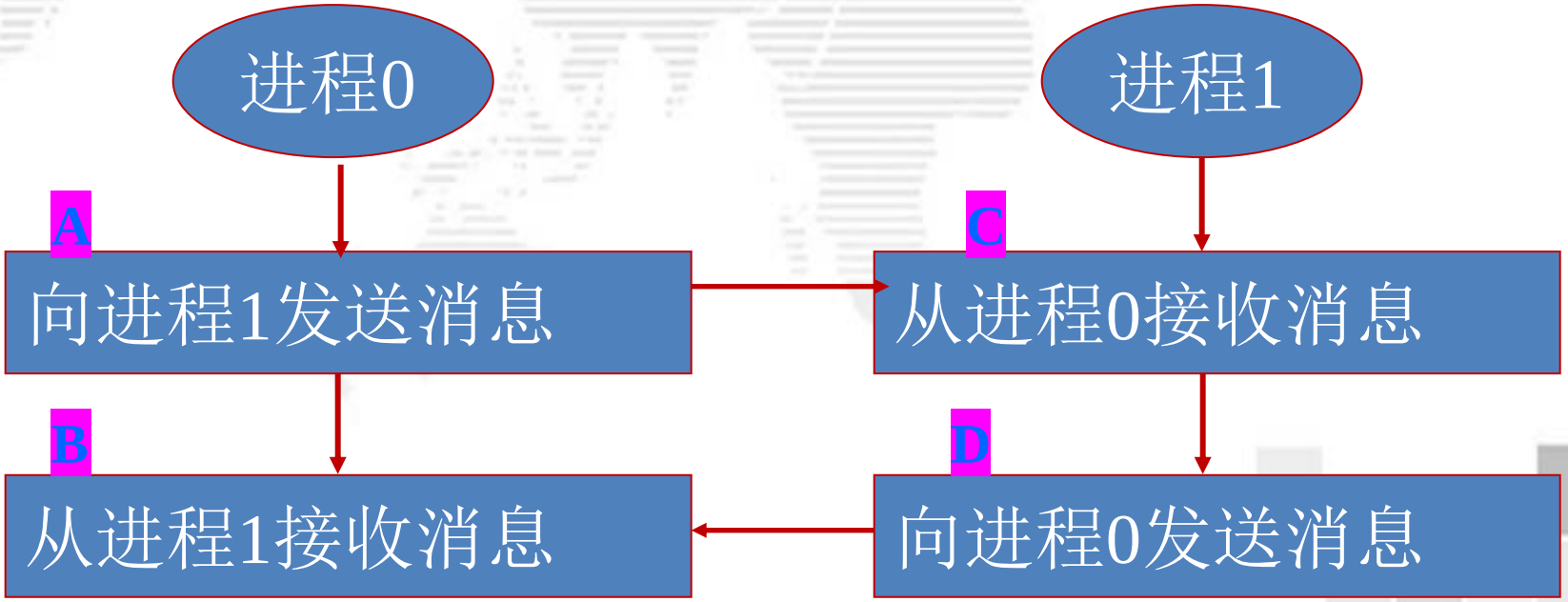


发送与接收按次序进行匹配

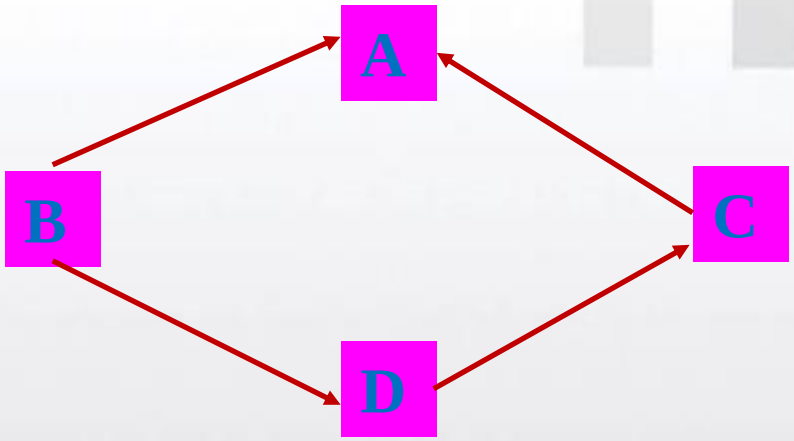
```
comm=MPI_COMM_WORLD
MPI_Comm_rank(MPI_COMM_WORLD,&rank)
If(rank==0)
{   MPI_Send(sendbuf,count,MPI_INT,1,tag,comm);
    MPI_Recv(recbuf,count,MPI_INT,1,tag,comm,&status);
}
If(rank==1)
{   MPI_Recv(recbuf,count,MPI_INT,0,tag,comm,&status);
    MPI_Send(sendbuf,count,MPI_INT,0,tag,comm);
}
```



安全通信！



- ✓ 只要C存在，则系统不提供缓冲区A也能够执行；
- ✓ C能够执行；
- ✓ A，C完成后，只要B存在，系统不提供缓冲区D也能够执行；
- ✓ B能够执行。



结论

- 将发送与接收操作按照次序进行匹配：

  一个进程的发送操作在前，接收操作在后；

  另一个进行的接收操作在前，发送操作在后；

- 若将两个进程的发送与接收操作次序互换，其消息传递过程仍是安全的

结束!