

CHAPTER 7

FLOYD'S ALGORITHM

（弗洛伊德算法）



电子科技大学
University of Electronic Science and Technology of China

学习目标

- 创建二维数组
- 思考 "粒度" 问题
- 介绍点对点通信
- 读取和构建二维矩阵
- 分析计算和通信重叠时的性能

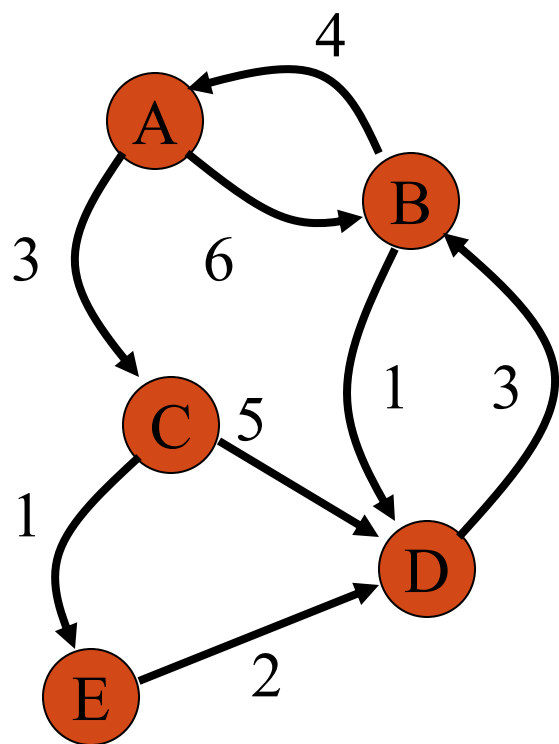


大纲

- 所有节点对之间的最短路径问题
- 动态二维数组
- 并行算法设计
- 点对点通信
- **I/O**块行矩阵**I/O**
- 分析和基准测试



所有节点对之间的最短路径问题



	A	B	C	D	E
A	0	6	3	6	4
B	4	0	7	10	8
C	12	6	0	3	1
D	7	3	10	0	11
E	9	5	12	2	0

结果包含距离的邻接矩阵

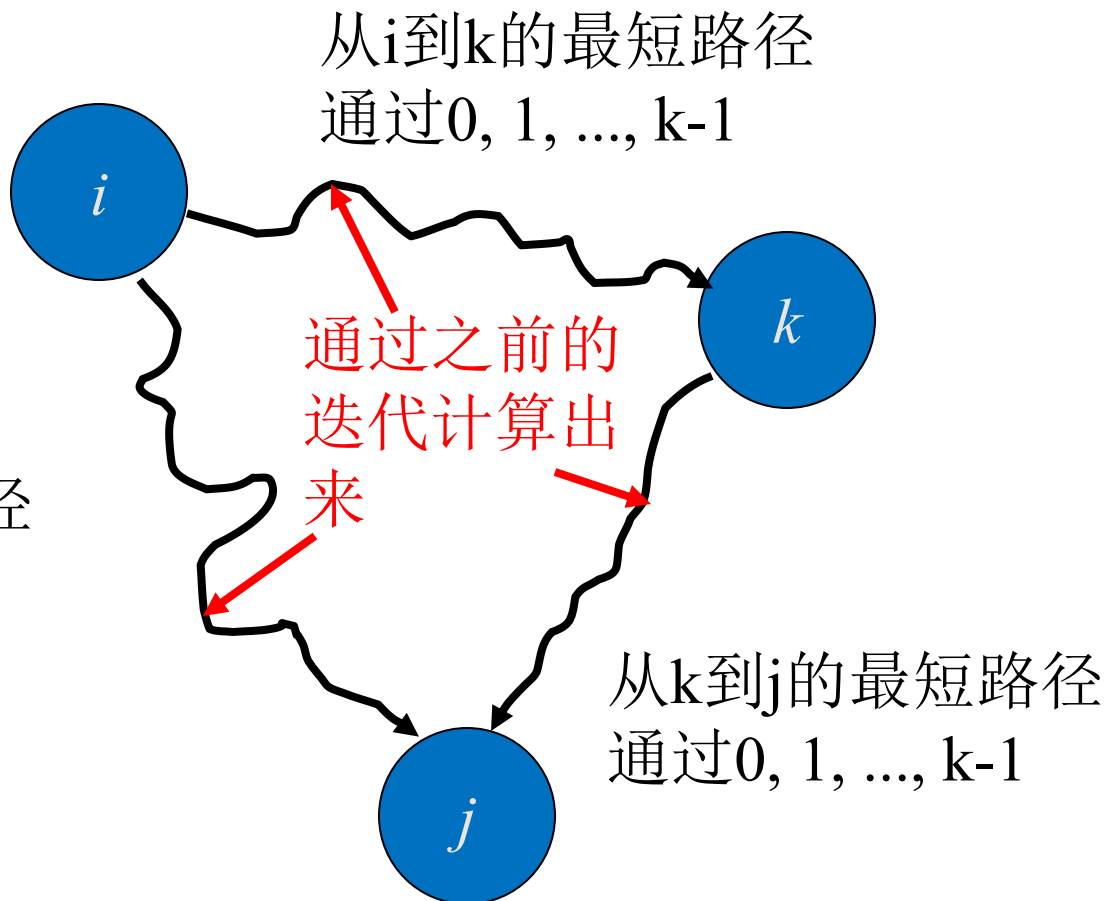
弗洛伊德算法

```
for  $k \leftarrow 0$  to  $n-1$ 
  for  $i \leftarrow 0$  to  $n-1$ 
    for  $j \leftarrow 0$  to  $n-1$ 
       $a[i,j] \leftarrow \min (a[i,j], a[i,k] + a[k,j])$ 
    endfor
  endfor
endfor
```

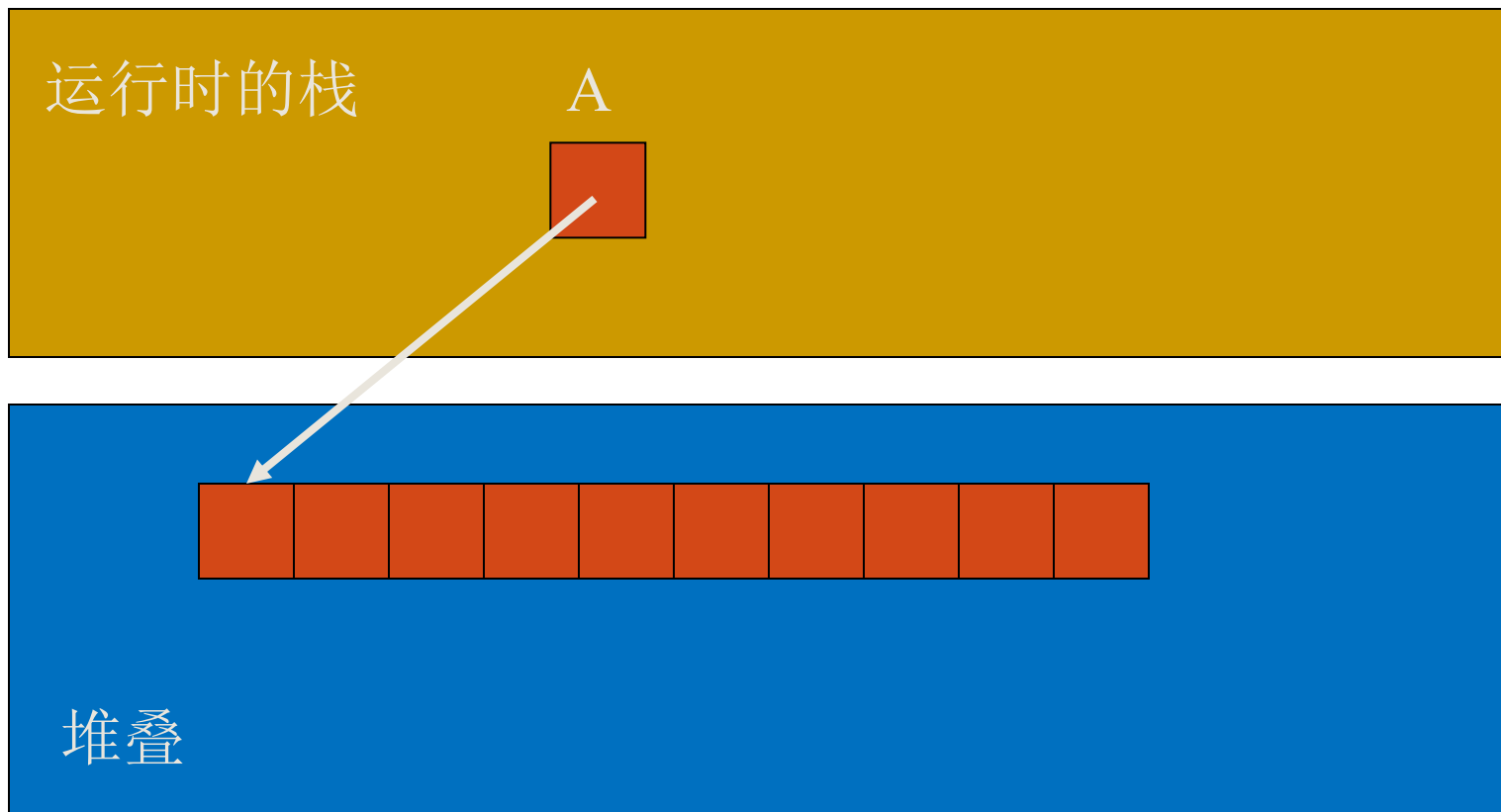


算法过程

从*i*到*j*的最短路径
通过0, 1, ..., *k*-1



创建动态一维数组



创建动态二维数组

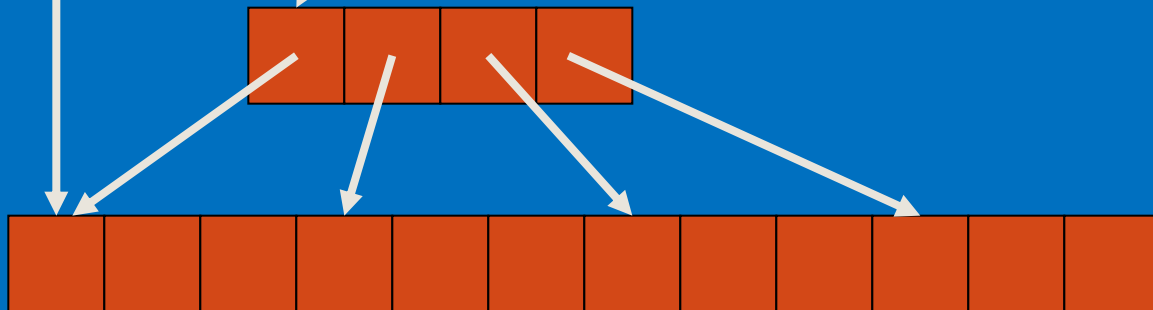
运行时的栈

Bstorage

B



堆叠



```
int *BStorage, **B;  
B = malloc(M * sizeof(int *));  
BStorage = malloc(SIZE * sizeof(int));  
for (int i = 0; i < M; i++)  
    B[i] = &BStorage[i * N];
```



并行算法设计

- 划分
- 通信
- 聚合和映射

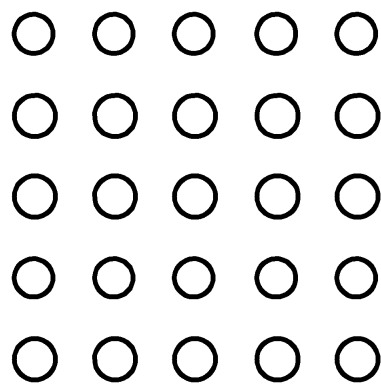
划分

- 领域划分还是功能划分？
 - 看一下伪代码
 - 同一赋值语句执行了 n^3 次
 - 没有功能并行
- 领域划分：将矩阵 **A** 分成它的 n^2 个元素

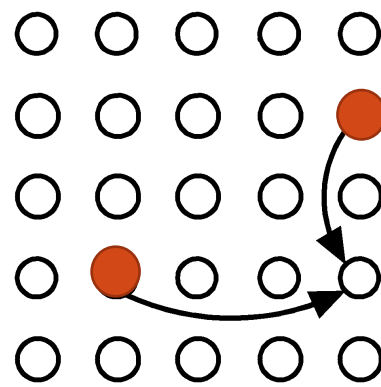
```
for  $k \leftarrow 0$  to  $n-1$ 
  for  $i \leftarrow 0$  to  $n-1$ 
    for  $j \leftarrow 0$  to  $n-1$ 
       $a[i,j] \leftarrow \min(a[i,j], a[i,k] + a[k,j])$ 
    endfor
  endfor
endfor
```

通信

原始任务



(a)

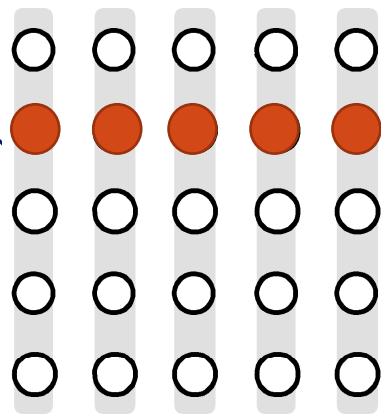


(b)

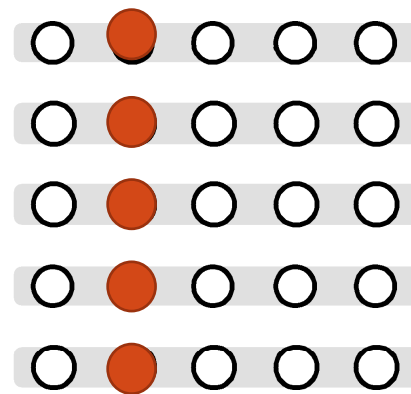
$k = 1$ 时更新
 $a[3,4]$

迭代 k :

第 k 行中的每个任务都会在任务列中广播它的值



(c)



(d)

迭代 k :

第 k 列中的每个任务都会在任务行中广播它的值

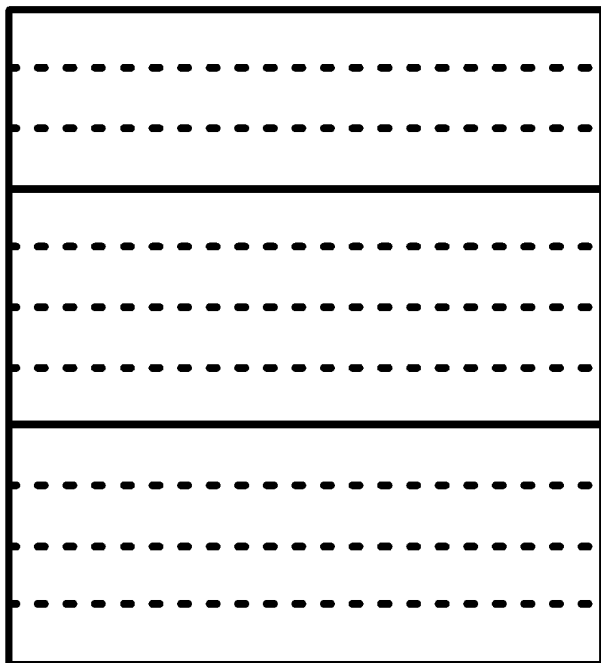


聚合和映射

- 任务的数量：静态
- 任务之间的通信：结构化
- 每个任务的计算时间：恒定
- 策略：
 - 聚合任务以减少通信
 - 每个**MPI**进程创建一个任务

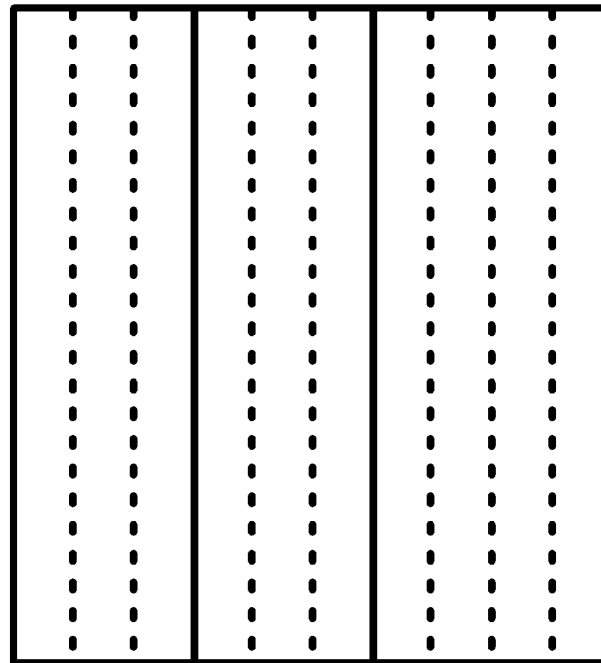
两种数据划分

行式块状条带化



(a)

列式块状条带化



(b)

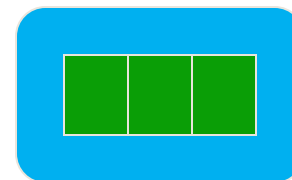
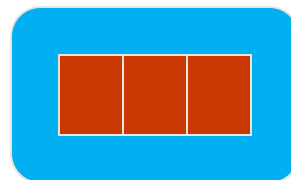
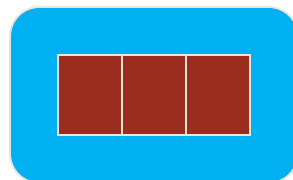
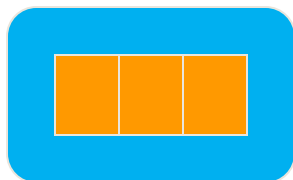
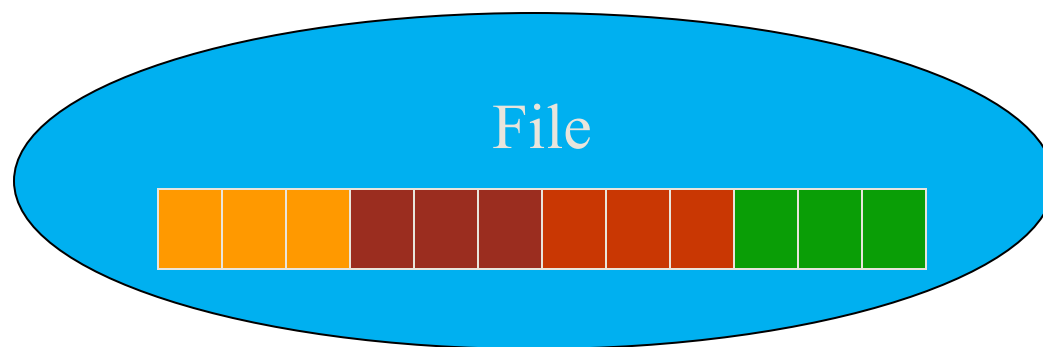


比较

- 列式块状条带化
 - 消除了列的广播
- 行式块状条带化
 - 消除了行内广播
 - 从文件中读取矩阵更简单
- 选择行式块状条带化划分



文件输入

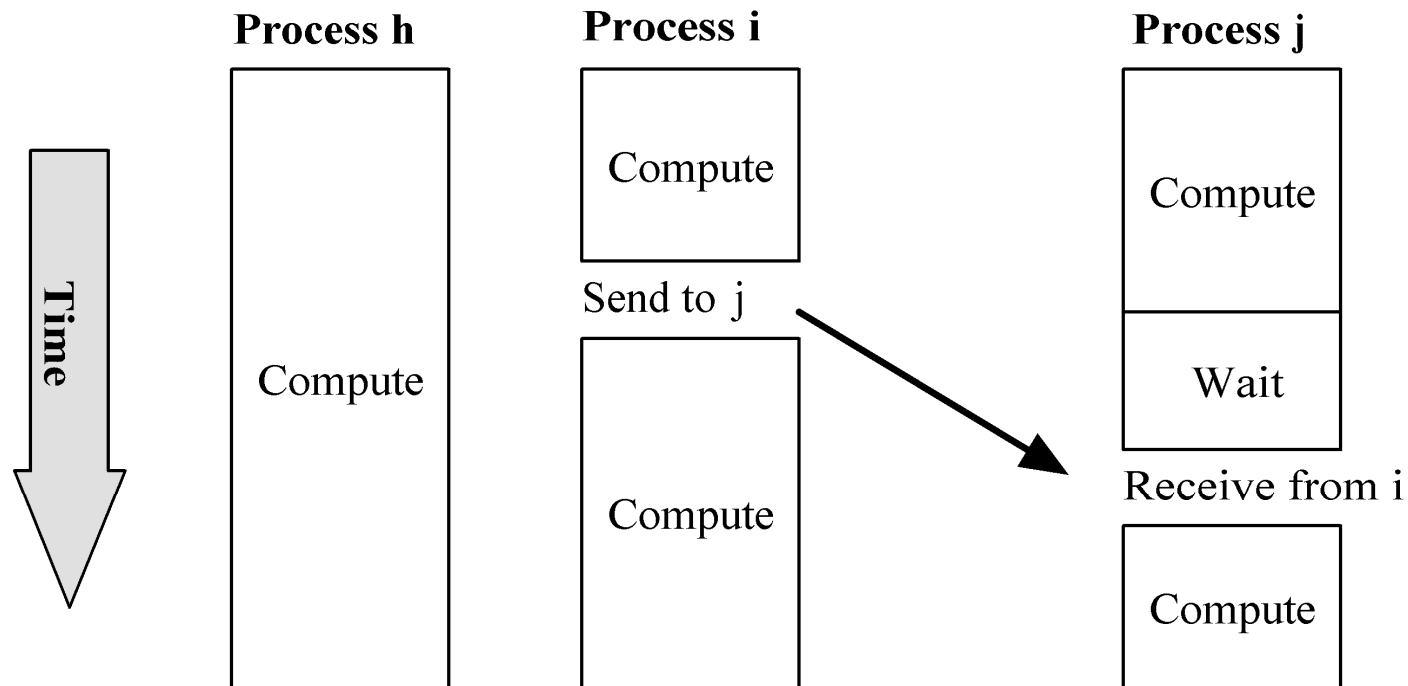


点对点通信

- 涉及到一对进程
- 一个进程发送一个消息
- 另一个进程接收该信息



发送/接收



函数 MPI_SEND

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



函数MPI_RECV

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



发送/接收代码

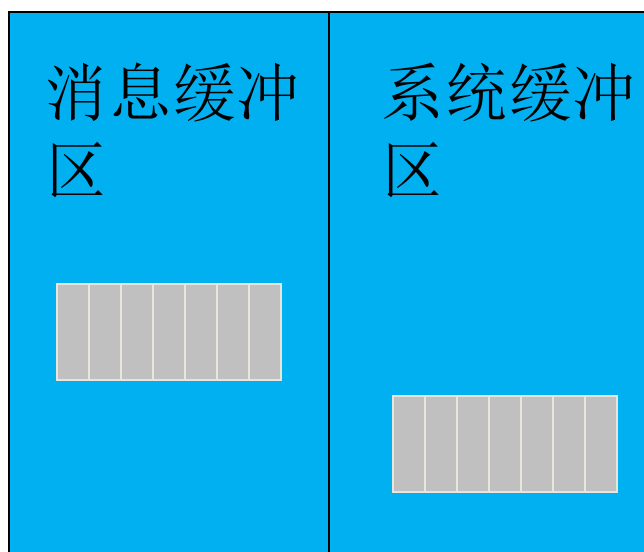
```
...  
if (ID == j) {  
    ...  
    Receive from I  
    ...  
}  
...  
if (ID == i) {  
    ...  
    Send to j  
    ...  
}  
...
```

接收在发送之前。
为什么这样做？



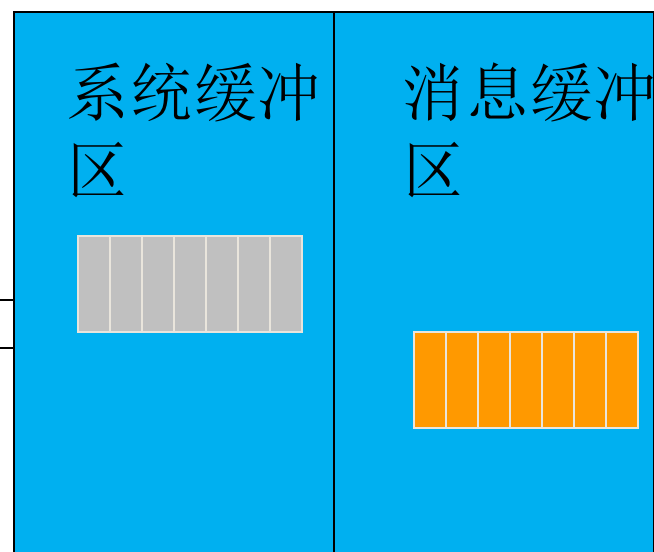
包括MPI_SEND和MPI_RECV

发送进程



MPI_Send

接收进程



MPI_Recv



RETURN FROM MPI_SEND

- 函数阻断，直到消息缓冲区空闲
- 消息缓冲区空闲的时间是
 - 消息被复制到系统缓冲区，或
 - 消息被传送
- 典型情况下
 - 消息被复制到系统缓冲区
 - 传输重叠的计算



RETURN FROM MPI_RECV

- 函数阻断，直到信息进入缓冲区
- 如果信息没有到达，函数就不会返回

死锁

- **死锁**: 等待一个永远不会成真的条件的过程
- 容易写出死锁的发送/接收代码
 - 两个进程: 都是先收后发
 - 发送标签与接收标签不一致
 - 进程向错误的目标进程发送消息

EXAMPLE 1

```
if (id == 0) {  
    MPI_Recv (&b, ...);  
    MPI_Send (&a, ...);  
    c = (a + b) / 2.0;  
} else if (id == 1) {  
    MPI_Recv (&a, ...);  
    MPI_Send (&b, ...);  
    c = (a + b) / 2.0;  
}
```

进程**0**阻塞等待来自**1**的信息，但**1**阻塞等待来自**0**的信息。

死锁!



EXAMPLE2-同样的场景

```
if (id ==0) {  
    MPI_Send(&a, ... 1,MPI_COMM_WORLD);  
    MPI_Recv(&b, ... 1, MPI_COMM_WORLD,&status);  
    c = (a+b)/2.0;  
}else if (id ==1) {  
    MPI_Send(&a, ... 0,MPI_COMM_WORLD);  
    MPI_Recv(&b, ... 0, MPI_COMM_WORLD,&status);  
    c = (a+b)/2.0;}
```

两个进程在试图接收之前都发送了，但它们仍然陷入僵局。

为什么？

标签是错误的。进程0试图接收一个1的标签，但进程1发送一个0的标签



发送/接收代码

```
...  
if (ID == j) {  
    ...  
    Receive from i  
    ...  
}  
...  
if (ID == i) {  
    ...  
    Send to j  
    ...  
}  
...
```

接收在发送之前。
为什么这样做？



MPI程序中的安全性

- 一个依赖**MPI**提供缓冲的程序被认为是**不安全**的
- 这样的程序对于一些输入集可能运行没有问题，但对于其他输入集可能会挂起或崩溃



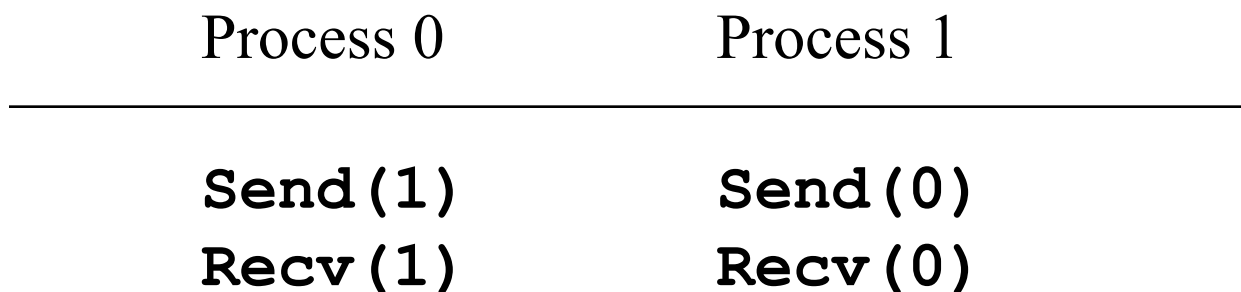
MPI程序中的安全性

- **MPI**标准允许**MPI_Send**以两种不同的方式行事：
 - 它可以简单地将消息复制到**MPI**管理的缓冲区并返回
 - 或者它可以阻塞，直到对**MPI_Recv**的匹配调用开始
- 许多**MPI**的实现都设置了一个阈值，在这个阈值上，系统会从缓冲区切换到阻塞区。
 - 相对较小的消息将由**MPI_Send**来缓冲
 - 较大的消息将导致其阻塞



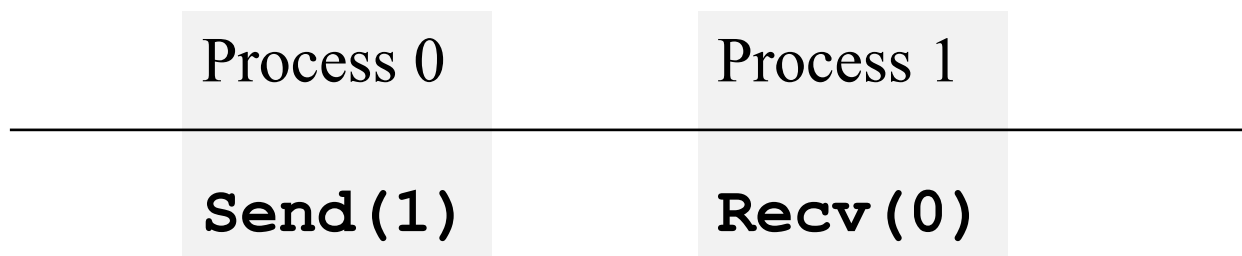
■ **Case 1** 从进程0向进程1发送一条很大的信息

- 如果在目的地没有足够的存储空间，发送必须等待用户提供存储空间（通过接收）



■ **Case 2** 生产者和消费者

- 如果流程0产生的大量信息需要发送给流程2，并且发送速度比接收速度快



MPI_SSEND

- **MPI**标准定义的**MPI_Send**的一个替代方案
- 额外的 "s "代表同步， **MPI_Send**保证阻塞直到匹配的接收开始

```
int MPI_Ssend(  
    void*          msg_buf_p      /* in */,  
    int            msg_size       /* in */,  
    MPI_Datatype    msg_type      /* in */,  
    int            dest           /* in */,  
    int            tag            /* in */,  
    MPI_Comm        communicator  /* in */);
```



重组通信

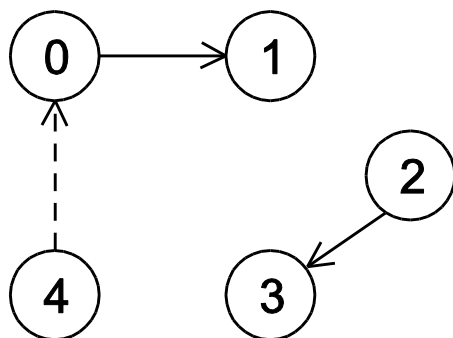
```
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);  
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,  
         0, comm, MPI_STATUS_IGNORE.
```



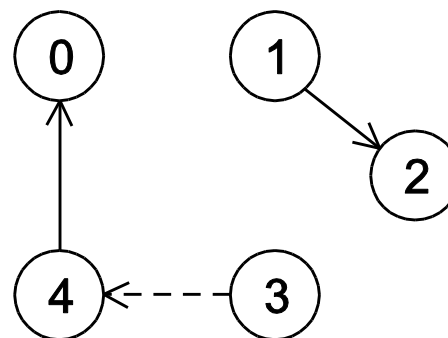
```
if (my_rank % 2 == 0) {  
    MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);  
    MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,  
            0, comm, MPI_STATUS_IGNORE.  
}  
else {  
    MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,  
            0, comm, MPI_STATUS_IGNORE.  
    MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);  
}
```



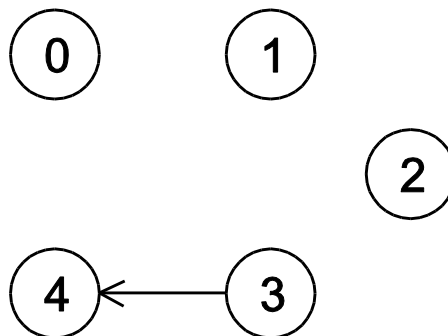
五个进程的安全通信



Time 0



Time 1



Time 2



MPI_SENDRECV

- 这是一个替代自己安排通信的方法
- 在一个单一的调用中执行一个阻塞的发送和接收
- 目的地和源可以是相同或不同的
- 特别有用的是**MPI**调度通信，这样程序就不会挂起或崩溃



MPI_SENDRECV

```
int MPI_Sendrecv(  
    void*          send_buf_p      /* in */,  
    int            send_buf_size   /* in */,  
    MPI_Datatype    send_buf_type  /* in */,  
    int            dest             /* in */,  
    int            send_tag         /* in */,  
    void*          recv_buf_p      /* out */,  
    int            recv_buf_size   /* in */,  
    MPI_Datatype    recv_buf_type  /* in */,  
    int            source           /* in */,  
    int            recv_tag         /* in */,  
    MPI_Comm        communicator    /* in */,  
    MPI_Status*     status_p        /* in */);
```



点对点通信VS集合通信

- 通信域中的所有进程必须调用相同的集体函数
 - 例如，如果一个程序试图将一个进程上对 **MPI_Reduce** 的调用与另一个进程上对 **MPI_Recv** 的调用相匹配，则是错误的，而且，很可能程序会挂起或崩溃



点对点通信VS集合通信

- 每个进程传递给**MPI**集合通信的参数必须是 "兼容的"
- 例如，如果一个进程传入0作为**dest_process**，而另一个进程传入1，那么调用**MPI_Reduce**的结果是错误的，而且，程序很可能再次挂起或崩溃



点对点通信VS集合通信

- **output_data_p**参数只在**dest_process**上使用
- 然而，所有的进程仍然需要传入一个与**output_data_p**相对应的实际参数，即使它只是**NULL**

点对点通信VS集合通信

- 点对点通信是在**标签和通信域**的基础上进行匹配
- 集合通信不使用标签
 - 它们只根据通信者和它们被调用的顺序来匹配

```

> int main (int argc, char *argv[]) {
    dtype** a;          /* Doubly-subscripted array */
    dtype* storage;     /* Local portion of array elements */
    int i, j, k;
    int id;             /* Process rank */
    int m;              /* Rows in matrix */
    int n;              /* Columns in matrix */
    int p;              /* Number of processes */
    double time, max_time;

    void compute_shortest_paths (int, int, int**, int);

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);

    read_row_stripped_matrix (argv[1], (void *) &a,
        (void *) &storage, MPI_TYPE, &m, &n, MPI_COMM_WORLD);

    if (m != n) terminate (id, "Matrix must be square\n");

    print_row_stripped_matrix ((void **) a, MPI_TYPE, m, n,
        MPI_COMM_WORLD);
    MPI_Barrier (MPI_COMM_WORLD);
    time = -MPI_Wtime();
    compute_shortest_paths (id, p, (dtype **) a, n);
    time += MPI_Wtime();
    MPI_Reduce (&time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0,
        MPI_COMM_WORLD);
    if (!id) printf ("Floyd, matrix size %d, %d processes: %6.2f seconds\n",
        n, p, max_time);
    print_row_stripped_matrix ((void **) a, MPI_TYPE, m, n,
        MPI_COMM_WORLD);
    MPI_Finalize();
}

```



```

void compute_shortest_paths (int id, int p, dtype **a, int n)
{
    int i, j, k;
    int offset;    /* Local index of broadcast row */
    int root;      /* Process controlling row to be bcast */
    int* tmp;      /* Holds the broadcast row */

    tmp = (dtype *) malloc (n * sizeof(dtype));
    for (k = 0; k < n; k++) {
        root = BLOCK_OWNER(k,p,n);
        if (root == id) {
            offset = k - BLOCK_LOW(id,p,n);
            for (j = 0; j < n; j++)
                tmp[j] = a[offset][j];
        }
        MPI_Bcast (tmp, n, MPI_TYPE, root, MPI_COMM_WORLD);
        for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
            for (j = 0; j < n; j++)
                a[i][j] = MIN(a[i][j], a[i][k] + tmp[j]);
    }
    free (tmp);
}

```



计算复杂度

- 最内层循环的复杂度为 $\Theta(n)$
- 中间循环最多执行 $\lceil n/p \rceil$ 次
- 外循环执行了 n 次
- 总体复杂度 $\Theta(n^3/p)$



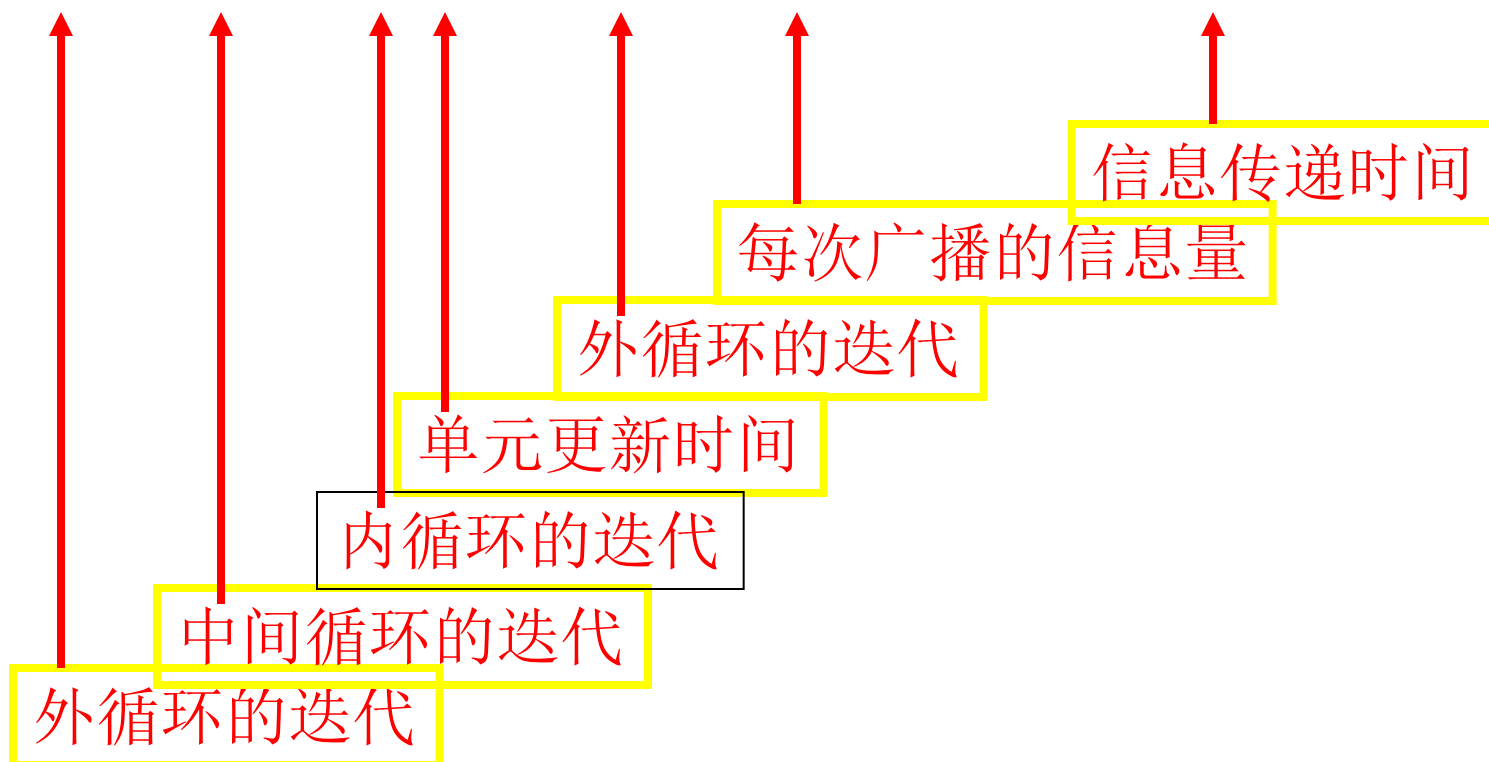
通信复杂度

- 内循环中没有交流
- 中循环没有通信
- 外循环中的广播 - 复杂度为 $\Theta(n \log p)$
- 总体复杂度 $\Theta(n^2 \log p)$

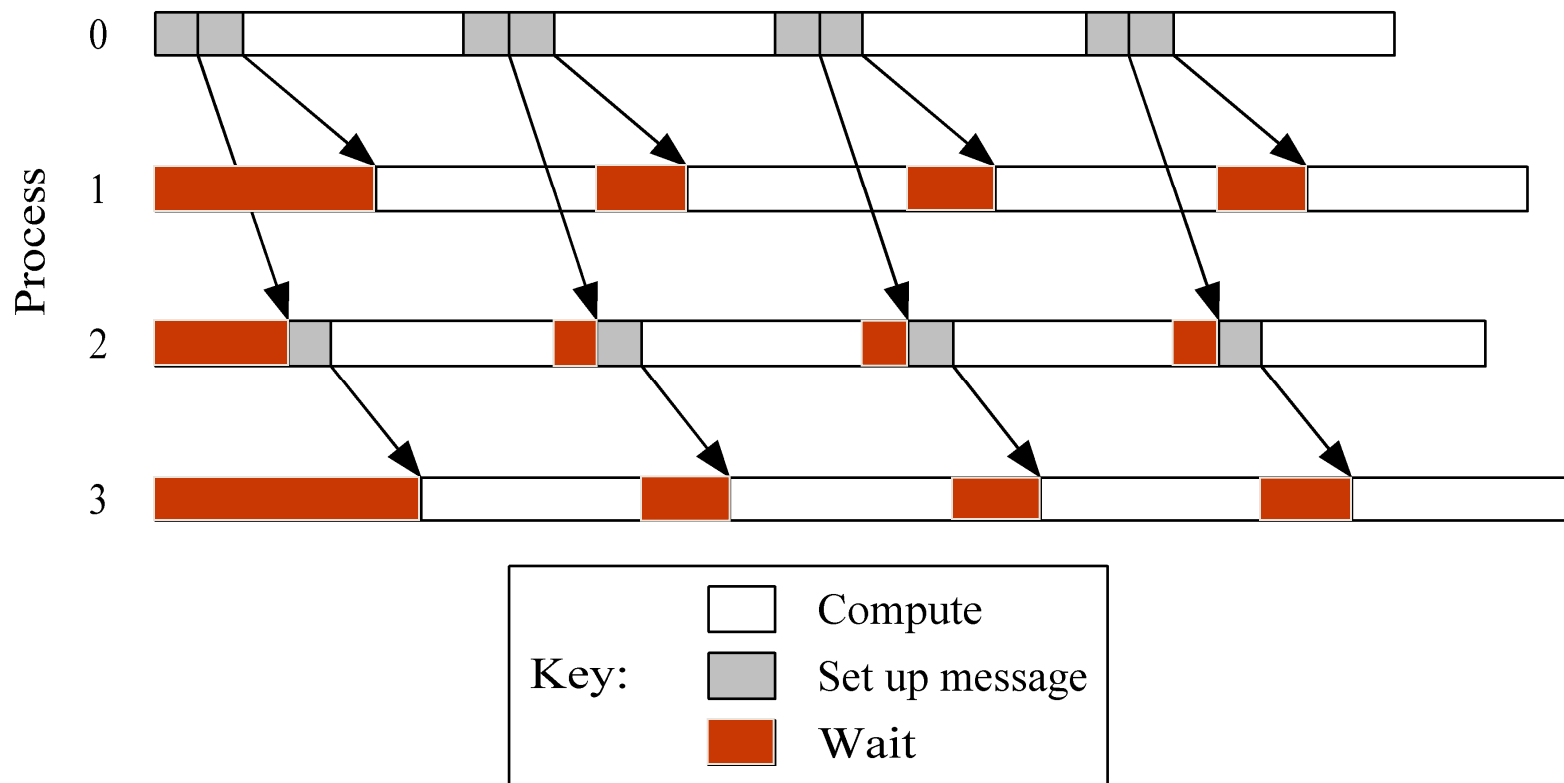


执行时间表达式 (1)

$$n \lceil n / p \rceil n\chi + n \lceil \log p \rceil (\lambda + 4n / \beta)$$

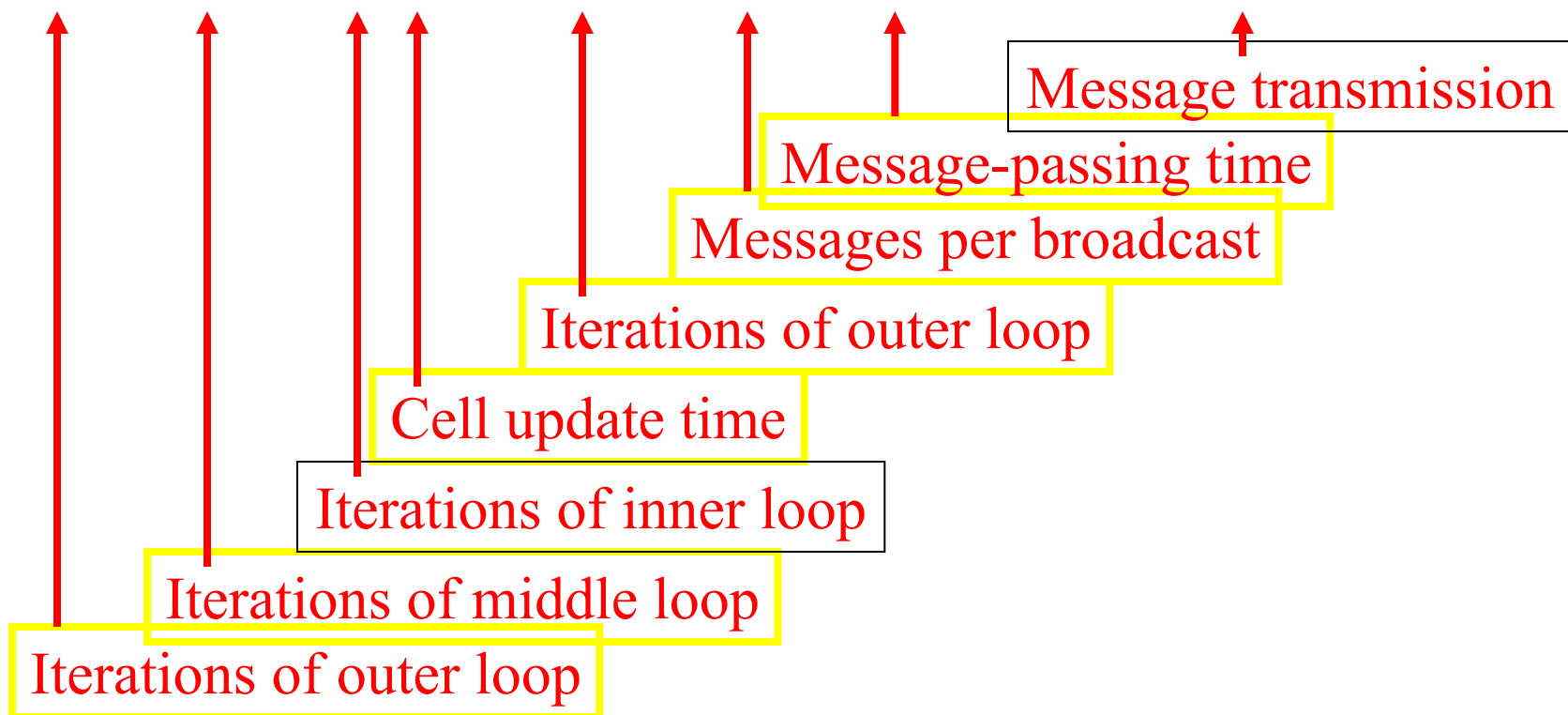


计算/通信叠加



执行时间表达式 (2)

$$n \lceil n / p \rceil n \chi + n \lceil \log p \rceil \lambda + \lceil \log p \rceil 4n / \beta$$



理论性能VS实际性能

Processes	Execution Time (sec)	
	Predicted	Actual
1	25.54	25.54
2	13.02	13.89
3	9.01	9.60
4	6.89	7.29
5	5.86	5.99
6	5.01	5.16
7	4.40	4.50
8	3.94	3.98



总结

- 两种矩阵划分法
 - 按行排列的块状划分
 - 按列排列的块状划分
- 分块发送/接收函数
 - MPI_Send
 - MPI_Recv
- 与计算重叠的通信

