进程级并行

- 通过消息传递实现进程间交互
    - 网络连接
    - 内存不共享
- 与存储系统的协作
- 任务调度
    - 如何将任务映射到不同处理器
    - 消息传递、同步等是串行程序之外的开销

进程级并行

进程间不共享内存空间，如何交换数据？
怎样通过进程间并行完成一个计算任务？

- MPI(Message Passing Interface)
  - 消息传递操作
  - 聚合操作
  - 定义数据类型、消息传递域
  - ……
- 《并行计算导论》第三章
- 《MPI与OPENMP并行程序设计：C语言版》第四章

MPI程序基本结构

```
MPI_Init(&argc, &argv); //处理额外参数,
                        //应在命令行参数处理之前调用
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
... ...
MPI_Finalize();
...
```

我是谁?

我在哪?

我要干什么?

编译MPI程序:

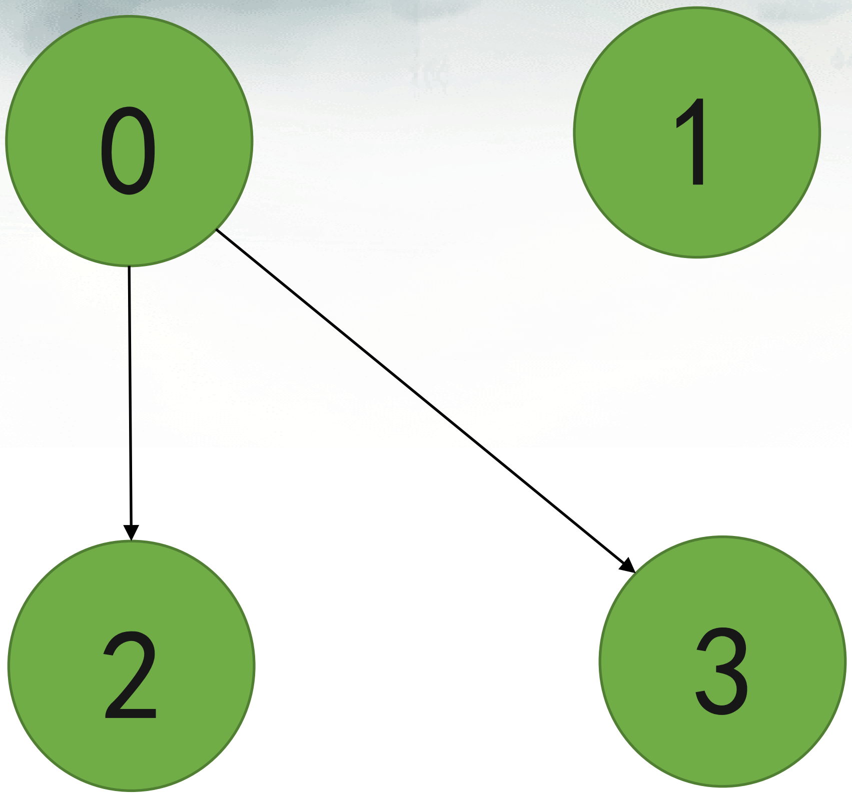
```
mpicc -o test ./test.c
```

运行MPI程序:

```
mpirun -np 4 -N 4 -hostfile hosts ./test 1024
```

进程级并行

- 点对点通信
  - 阻塞消息传递
  - 非阻塞消息传递
  - 死锁的问题
- 聚合通信



进程级并行

- MPI(Message Passing Interface)基本操作
  - int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);

参数：
buf：发送缓冲区的首地址
count：需要发送的数据项个数
datatype：每个被发送元素的数据类型
dest：目标进程的进程号（rank）
tag：消息标识（接收端要使用同样的标识）
comm：通信域（哪些进程参与通信）

进程级并行

- MPI(Message Passing Interface)基本操作
  - int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,MPI_Comm comm, MPI_Status *status)

参数：
buf：接收缓冲区的首地址
count：接收缓冲区最多存放多少个数据项
datatype：每个被接收元素的数据类型
source：发送进程的进程号（rank）
tag：消息标识
comm：通信域
status：MPI_Status指针，函数返回时存放发送方进程号、消息tag等

进程级并行

- 非阻塞消息传递
  - MPI_Isend, MPI_Irecv， 需要MPI_Request指针
    - 调用之后立即返回
    - 可以在等待完成期间进行其他计算
  - 用MPI_Wait()来等待完成， 或用MPI_Test来检验是否完成

```
int MPI_Isend( void *buf, int count, MPI_Datatype datatype, int dest, int tag,
               MPI_Comm comm, MPI_Request *request )

int MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int source,
               int tag, MPI_Comm comm, MPI_Request *request )
```

非阻塞消息传递

```
double buff[1024];
MPI_Request req;
MPI_Status status;
if(rank == 1) {
MPI_Irecv(buff, 512, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &req);
        //Do some computation
        MPI_Wait(&req, &status);
        //Use the data in buff
}
else {
        MPI_Isend(buff, 512, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &req);
        //Do some computation
        MPI_Wait(&req, &status);
        //Can use buff now.
}
```

非阻塞消息传递

```
double buff[1024];
MPI_Request req;
int flag = 0;
if(rank == 1) {
MPI_Irecv(buff, 512, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &req);
        //Do some computation
        while(!flag) {
                MPI_Test(&req, &flag, &status);
        }
        //Use the data in buff
}
else {
        MPI_Isend(buff, 512, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &req);
        //Do some computation
        while(!flag) {
                MPI_Test(&req, &flag, &status);
        }
```

MPI_Sendrecv()

- 《并行计算导论》3.2节

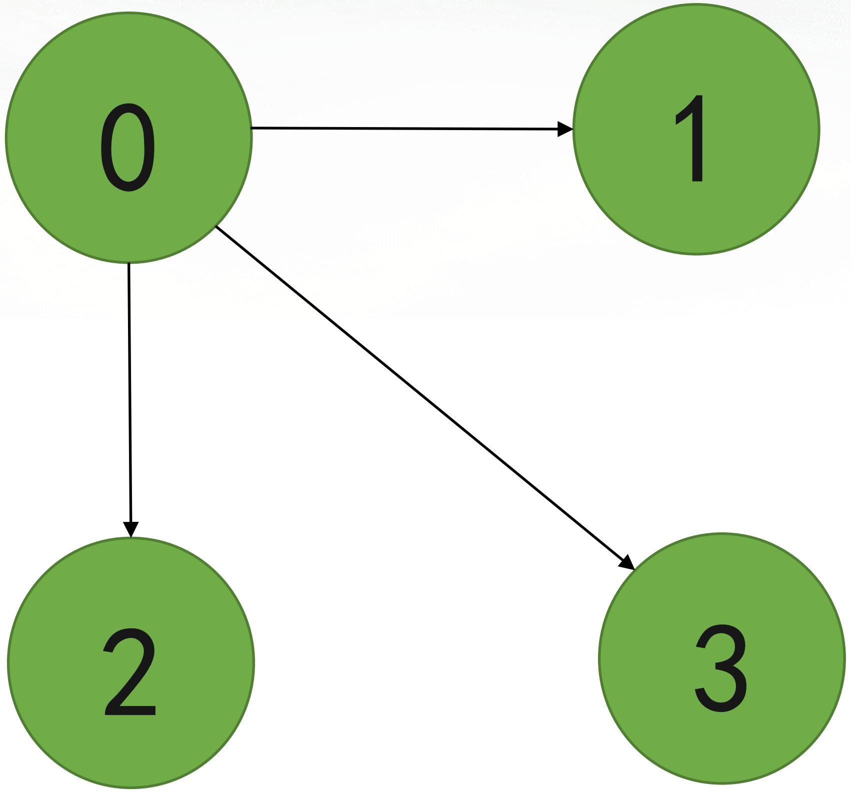
- 当发送数据很少时，MPI_Send会立即返回
- 当发送数据较多时，以下程序会死锁，因为所有进程都在等待目标程序接收。目标程序接收后MPI_Send才能返回
- 解决办法：使用MPI_Sendrecv()，同时发送与接收

```
static int buf1[SIZE], buf2[SIZE];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs); /* 获取总进程数
*/
MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* 获取本进程的进
程号*/
memset(buf1, 1, SIZE);
tag = 123;
dst = (rank > nprocs - 1) ? 0 : rank + 1;
src = (rank == 0) ? nprocs - 1 : rank - 1;
MPI_Send(buf1, SIZE, MPI_INT, dst, tag,MPI_COMM_WORLD);
MPI_Recv(buf2, SIZE, MPI_INT, src, tag, MPI_COMM_WORLD,
&status);

MPI_Finalize();
```

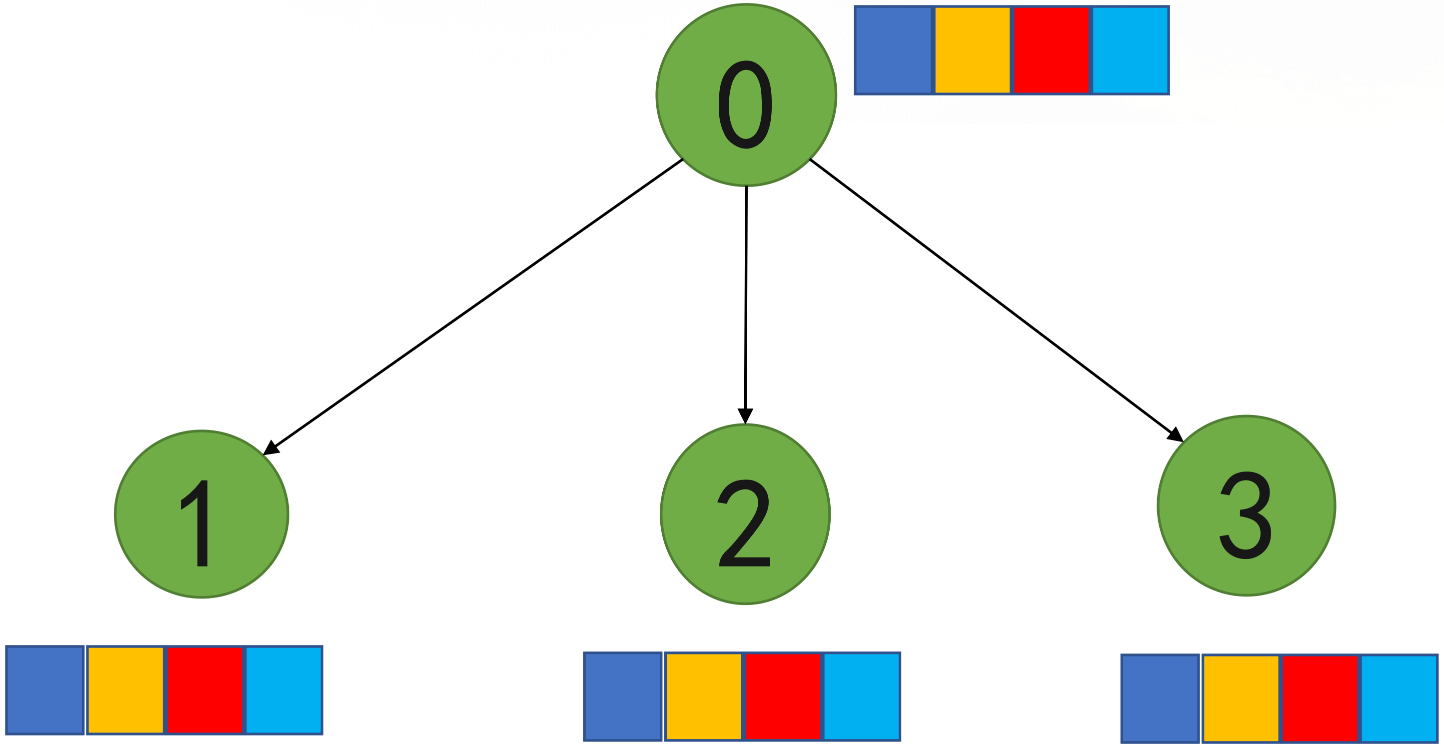
MPI聚合通信

- 需要多个进程参与
- 比较高效地完成较复杂的消息传递任务
    - MPI_Bcast
    - MPI_Scatter(v)
        - V是不等长版本
    - MPI_Gather(v)
    - MPI_Reduce
    - ……



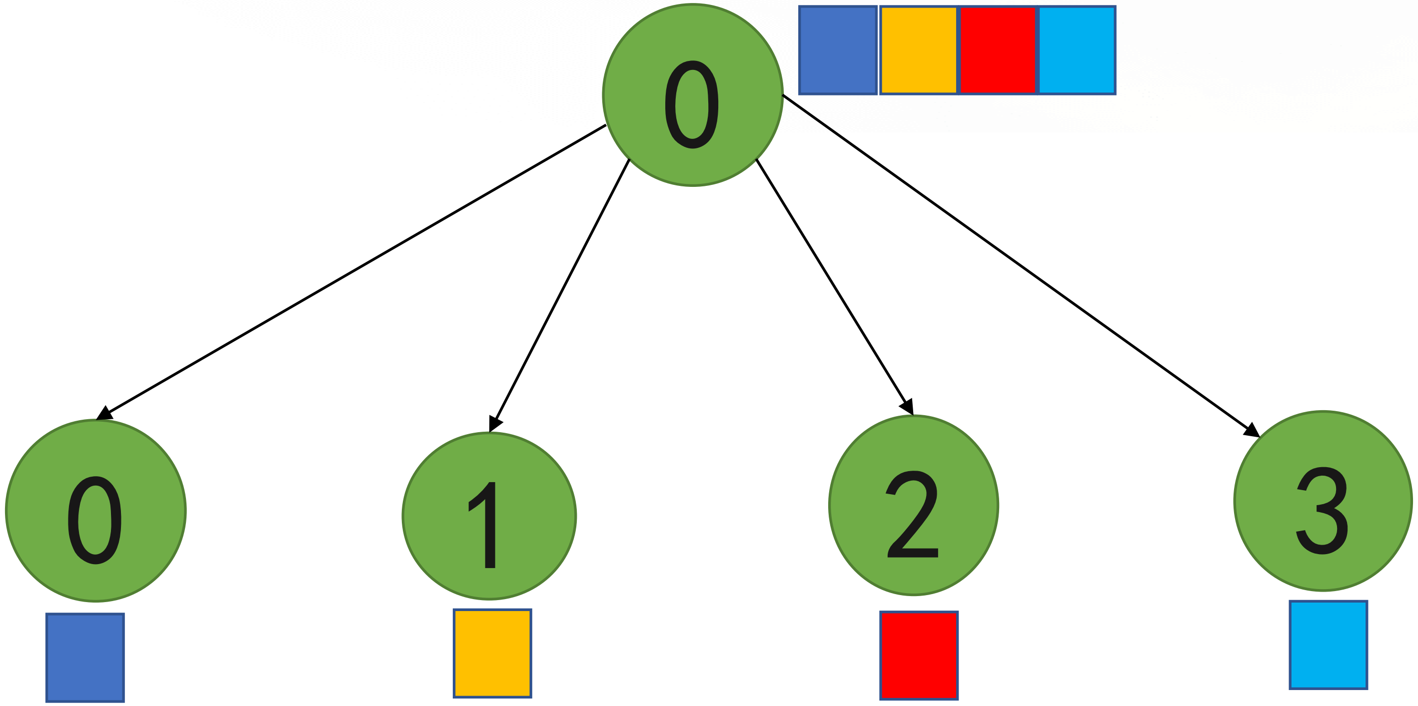
MPI_Bcast

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



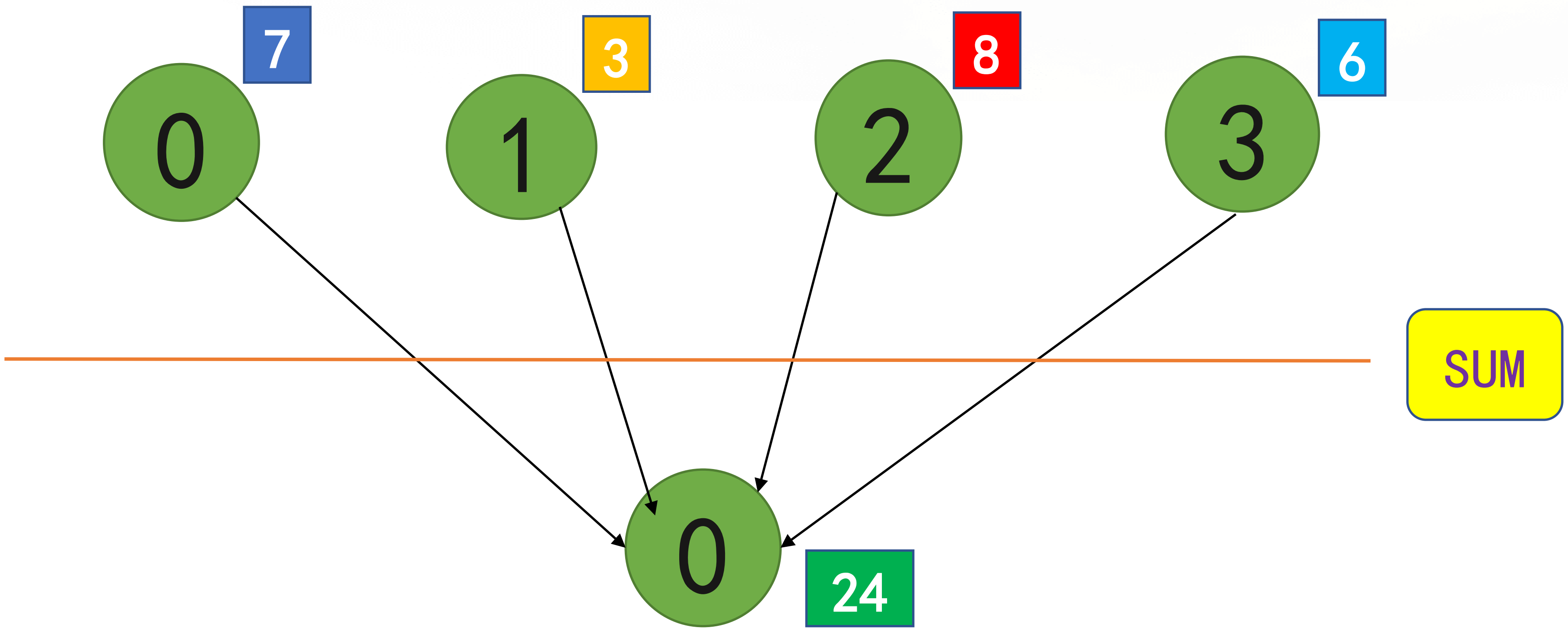
MPI_Scatter

int MPI_Scatter (void *sendbuf, int sendcnt, MPI_Datatype sendtype,
void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root,
MPI_Comm comm)



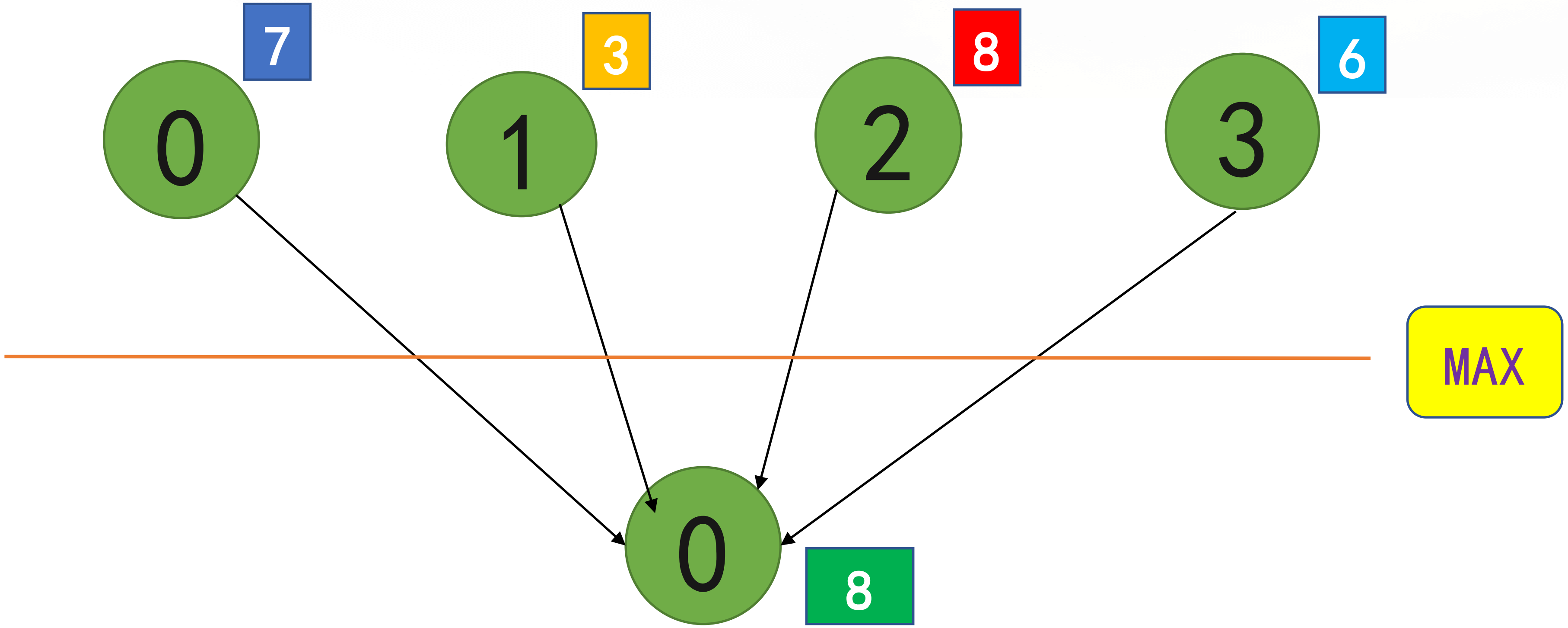
MPI_Reduce

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



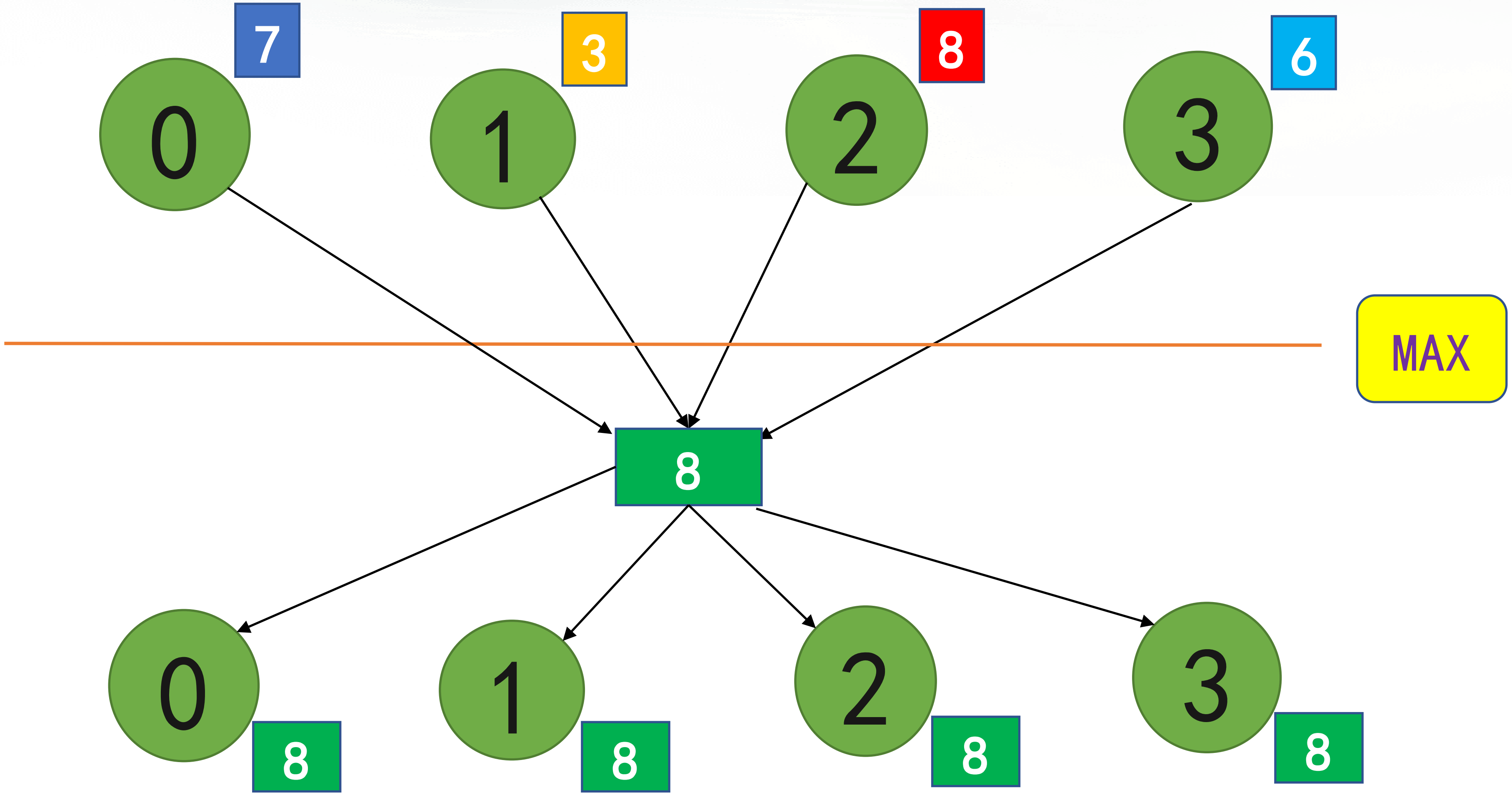
MPI_Reduce

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



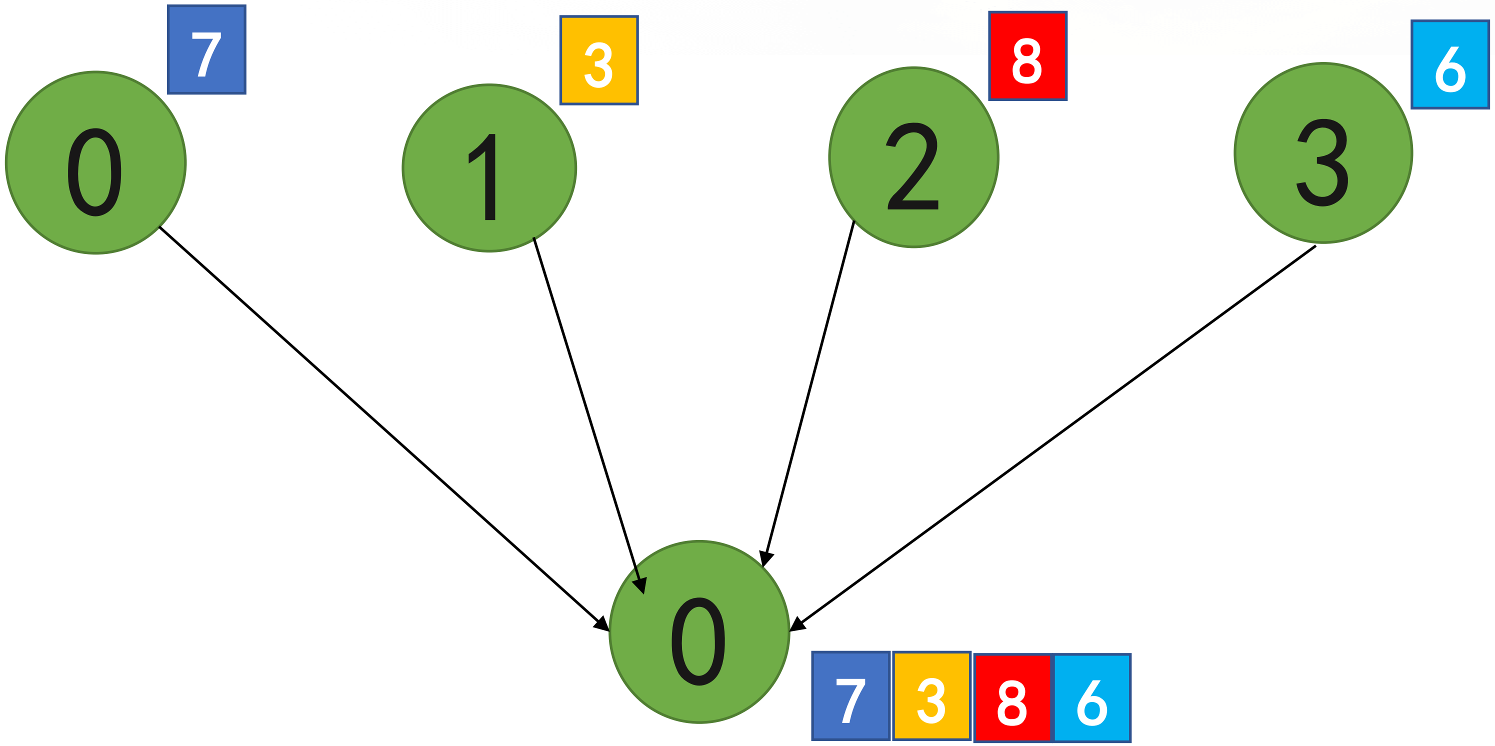
MPI_AllReduce

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



MPI_Gather

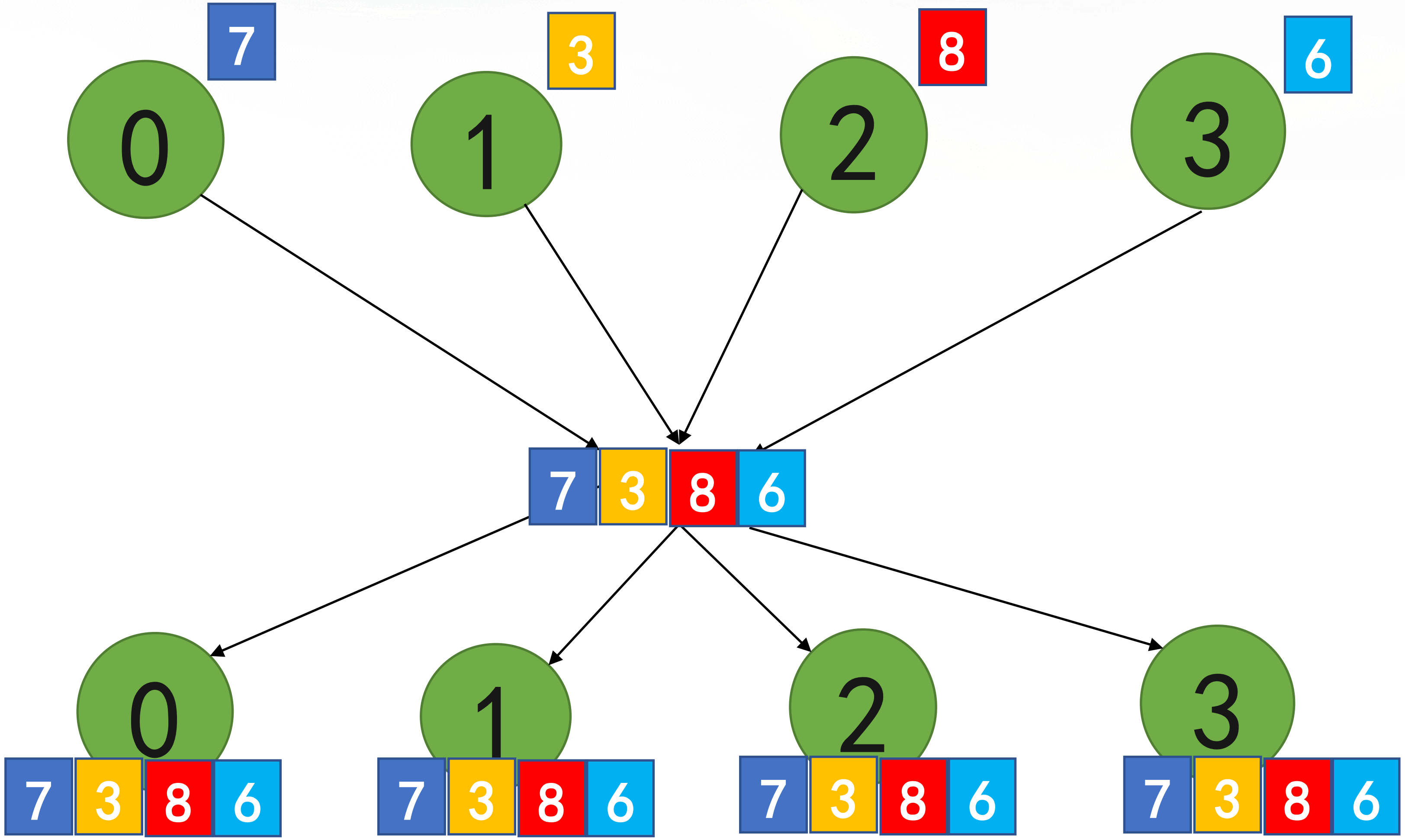
```
int MPI_Gather ( void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm )
```



root=0

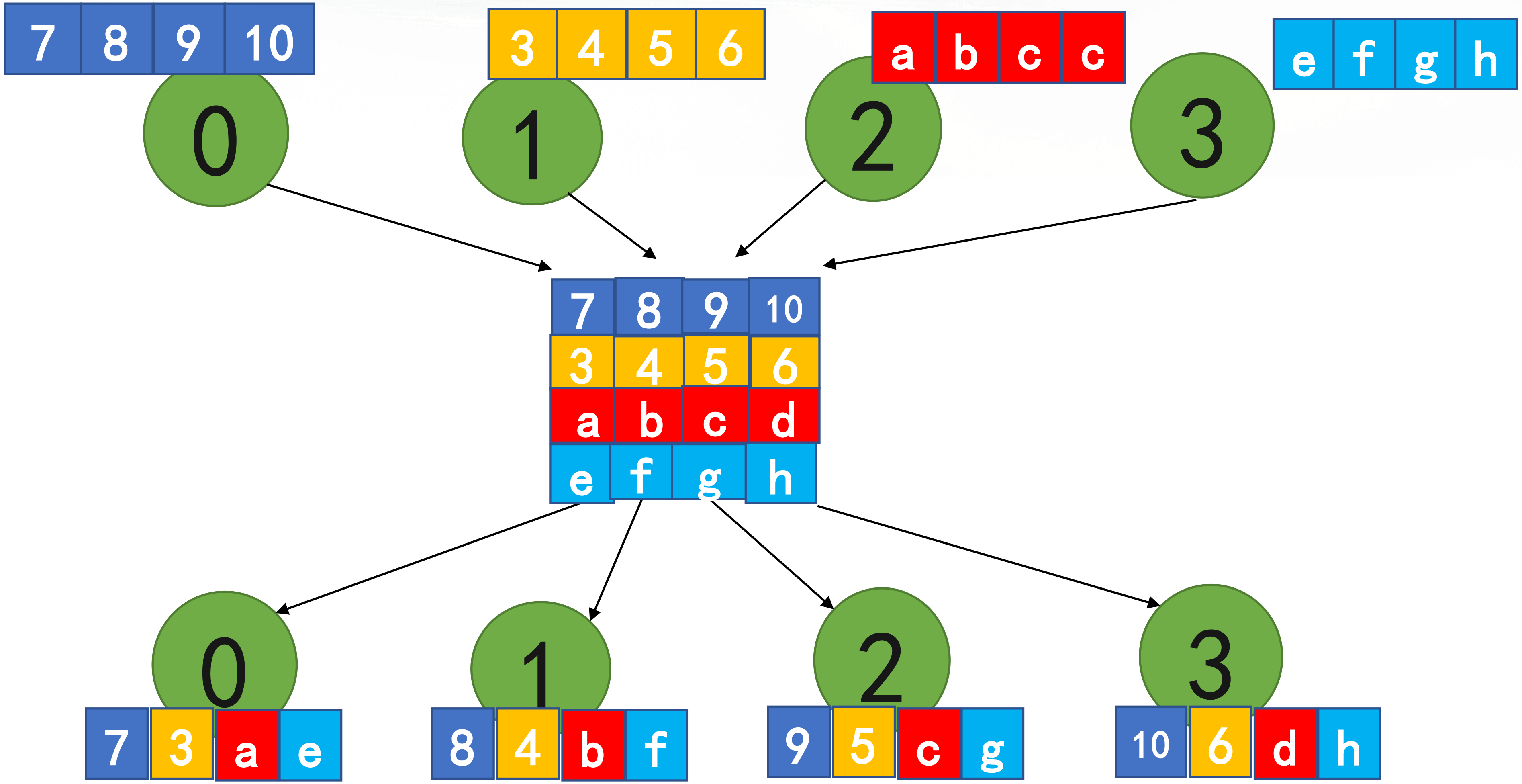
MPI_Allgather

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



MPI_Alltoall

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

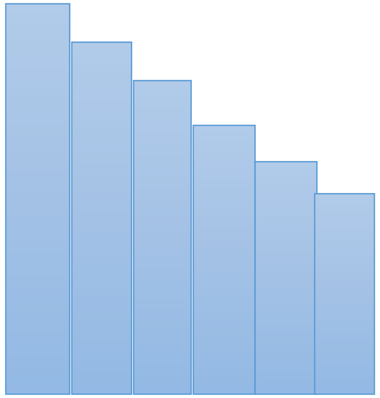


举个例子

- MPI程序举例：计算π

```
double f( double a ) { return (4.0 / (1.0 + a*a))};
int main(int argc, char *argv[]) {
     int n = 4, myid, I;
    double h, sum, x, mypi, startwtime;
    MPI_Init(argc, argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    startwtime = MPI_Wtime();
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid; i < n; i += numprocs) {
            x = h * ((double)i + 0.5);
            sum += f(x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0) {
            endwtime = MPI_Wtime();
            printf("pi is approximately %.16f\", pi);
            printf("wall clock time = %f\n", endwtime-startwtime);
    }
    MPI_Finalize();
}
```

$$x = \int_0^1 \frac{4}{1+x^2} dx$$ 计算π



再举个例子

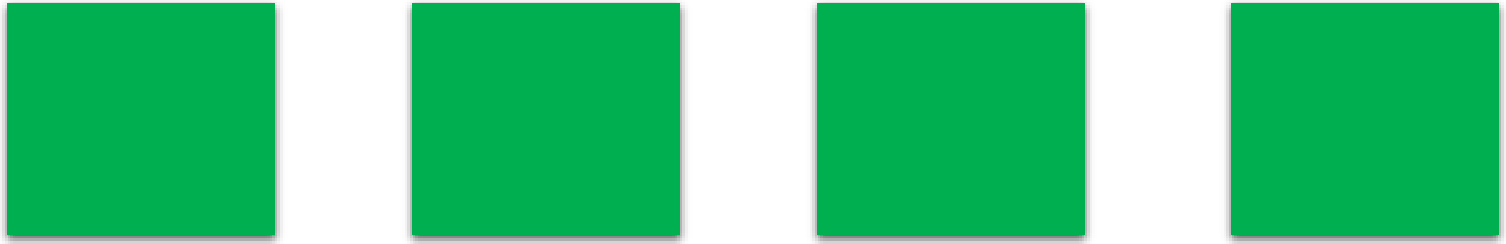
- Eratosthenes筛法计算素数的个数

给定自然数序列{2, 3, ……, N}，找出其中所有的素数
1. 给序列中所有数设置一个标记，并置0
2. 令k=2，执行以下操作
3. while (k^2<=N){
    将k^2到N之间的k的倍数的标记置为1；
    找出比k大的标记为0的数当中最小的一个（假设是m)，令k=m;
  }
4. 列表中标记为0的数都是素数

- Eratosthenes筛法计算素数的个数

需要处理以下数列：
2，3， 4， 5， 6， 7， ……， N

假设系统中有P个结点，如何用MPI实现并行处理?
每个结点有自己的内存空间；
结点间用高速网络相连

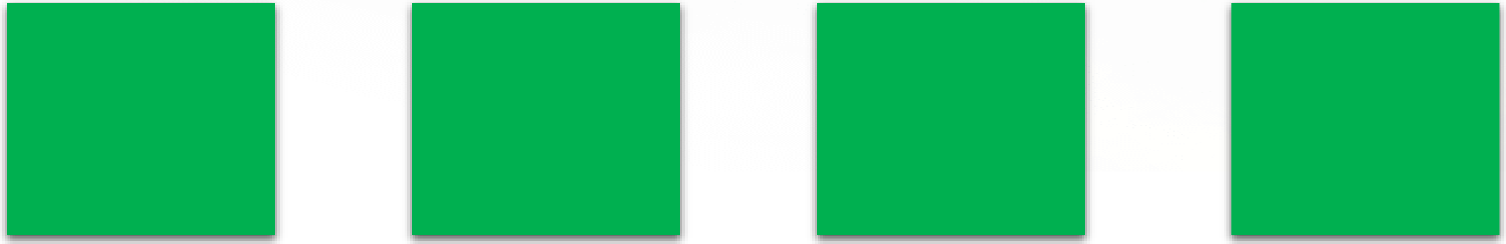


设N = 16, P = 4
N是P的倍数，所以每个结点处理N/P个数

每个结点一个进程
运行时mpirun –np 4 ./your_program

- Eratosthenes筛法计算素数的个数

考虑N不能被P整除的情况



设N = 14, P = 4

设N = 14, P = 4
可以分成｛4，4，3，3｝，也可以分成｛3，4，3，4｝
书中取用后者

```
#define BLOCK_LOW(id, p, n) ((id)*(n)/(p))
#define BLOCK_HIGH(id, p, n) (BLOCK_LOW((id+1),p,n)-1)
```

- Eratosthenes筛法计算素数的个数

```
MPI_Init(&argc, &argv);
int id, proc_num, local_size;
MPI_Comm_size(MPI_COMM_WORLD,
&proc_num);
MPI_Comm_world(MPI_COMM_WORLD, &id);
int low = 2+BLOCK_LOW(id,p,n-1);
int high = 2+BLOCK_HIGH(id,p,n-1);
Int local_size = BLOCK_SIZE(id,p,n-1);
char * mark = (char*)malloc(local_size);
memset(mark, 0, local_size);
if(!id ) index = 0;
int k = 2;
do {
   if(k*k > low) first = k*k – low; //前面k*k个已经处
理过，不用考虑
   else {
    //找到本地第一个k的倍数
    if((low%k) ==0) {        first = 0;    }
    else {
      first = k – (low % k);
    }
   }
    for(int i = first; i < local_size; i += k) {
       mark[i] = 1; //标记每个k的倍数
    }
    if(!id) {
       while(mark[++index]);
       k = index + 2;
    }
    MPI_Bcast(&k, 1, MPI_INT, 0, MPI_COMM_WORLD);
} while( k * k <= n) ;

int count = 0, global_count = 0;
for (i = 0; i < local_size; i++)
    count += 1-mark[i];
MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM,
0, MPI_COMM_WORLD);

if(!id) printf( "%d prime numbers are less than or equal to
%d\n" , global_count, n);
MPI_Finalize();
```

为什么只需要第一个进程做就可以了？（假设p<$\sqrt{n}$）

并行 Eratosthenes 筛法程序的分析

- 串行程序的时间
  - **素数个数：ln(ln(n)),即迭代次数；**
  - **每次迭代处理n个数；**
  - **处理每个数需要时间：** χ
  - **总时间：** χ **nln(ln(n))**
- 并行程序预期运行时间
  - **广播时间约为$\lambda\lceil \boldsymbol{log} \mathbf{P}\rceil$，λ是消息的延迟**
  - 2~n之间素数个数约为n/ln(n)，因此循环迭代次数约为$\sqrt{\boldsymbol{n}}/\boldsymbol{ln}\sqrt{\boldsymbol{n}}$
  - 并行算法时间为$\chi(\boldsymbol{nlnln}(\boldsymbol{n}))/\boldsymbol{p} + (\sqrt{\boldsymbol{n}}/\boldsymbol{ln}\sqrt{\boldsymbol{n}})\lambda\lceil \boldsymbol{logP}\rceil$

https://www.cnblogs.com/dc93/p/3930362.html 有对串行程序的较详细分析

- Eratosthene筛法计算素数的个数

优化一：去掉偶数

只需要处理一半的数据

从3开始

```
k=3;
local_size = n/proc_num/2;
......
for(int i = first; i < local_size; i += k) {
        mark[i] = 1;
}
if(!id) {
        while(mark[++index]);
        k = index + 3;
 }
```

总时间并不会减半，因为通信的开销并没有实质性减少（仅仅少了一次）---- P较大时优化效果不大

- Eratosthene筛法计算素数的个数

优化二：消除广播

可以令每个进程先算出$\sqrt{n}$之前的素数，这样每次循环中就不必再广播k；如果下面条件成立，消除广播操作会提高并行程序性能：

广播的时间

计算√n之前的素数的时间

$$(\sqrt{n}/\ln\sqrt{n})\lambda\lceil\log p\rceil > \chi\sqrt{n}\ln\ln\sqrt{n}$$

$$\Rightarrow (\lambda\lceil\log p\rceil)/\ln\sqrt{n} > \chi\ln\ln\sqrt{n}$$

$$\Rightarrow \lambda > \chi\ln\ln\sqrt{n}\ln\sqrt{n}/\lceil\log p\rceil$$

λ是消息延迟，χ是循环中每次迭代的时间

2到n之间有n/ln(n)个素数

串行执行f(n)的时间约为χ nln(ln(n))

- Eratosthene筛法计算素数的个数

优化三：改变循环顺序

cache！($)
假设cache line大小为4字节（4个char)
1-way association

3-99：

3的倍数：

9 15 21 27 33 39 45 51 57 63 69 75 81 87 93 99

5的倍数：

25 35 45 55 65 75 85 95

7的倍数：

49 63 77 91

3	5	7	9
11	13	15	17
......			
27	29	31	33

红：hit
黄：miss

- Eratosthene筛法计算素数的个数

优化三：改变循环顺序

3-17： 3的倍数： 9 15

19-33: 3、5的倍数： 21 27 33 25

35-49: 3、5、7的倍数： 39 45 35 45 49

51-65: 3、5、7的倍数： 51 57 63 55 65 63

67-81: 3、5、7的倍数： 63 69 75 81 75 77

83-97: 3、5、7的倍数： 87 93 85 95 91

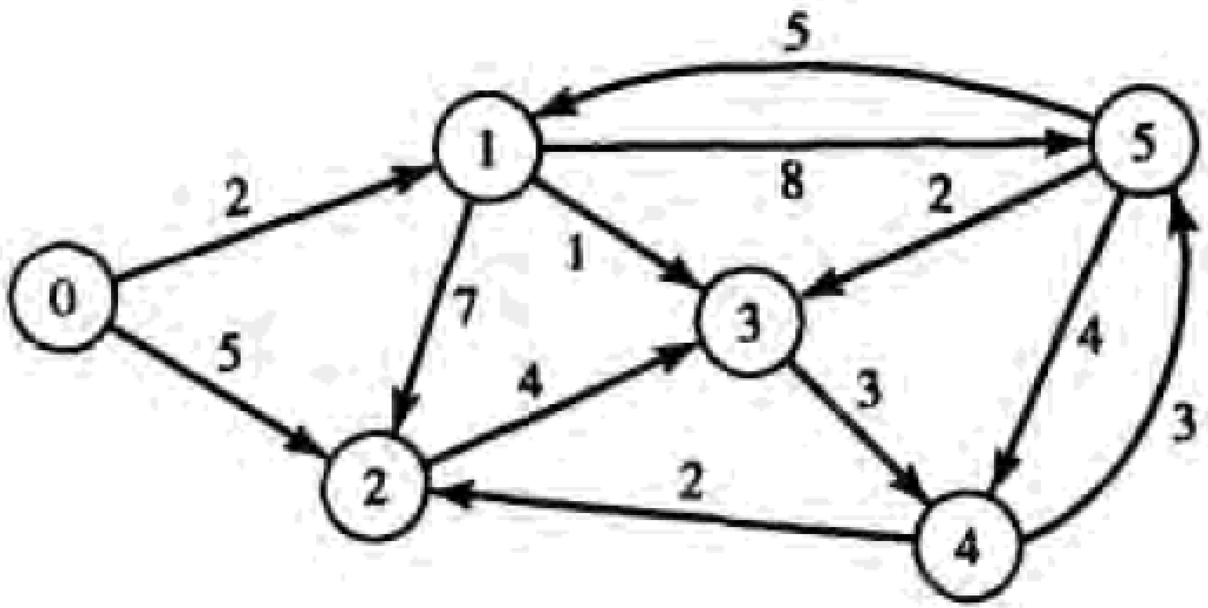
99: 3、5、7的倍数： 99

将数据块分段，每段内部运行现有循环;
Cache 命中率大幅提高！

红： hit
黄： miss

- Floyd算法的多进程并行

寻找有向加权图中所有点对间最短路径
图用邻接矩阵表示



	0	1	2	3	4	5
0	0	2	5	∞	∞	∞
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	∞	0	3
5	∞	5	∞	2	4	0

	0	1	2	3	4	5
0	0	2	5	3	6	9
1	∞	0	6	1	4	7
2	∞	15	0	4	7	10
3	∞	11	5	0	3	6
4	∞	8	2	5	0	3
5	∞	5	6	2	4	0

- Floyd算法的多进程并行

Floyd 算法

输入：n：顶点数

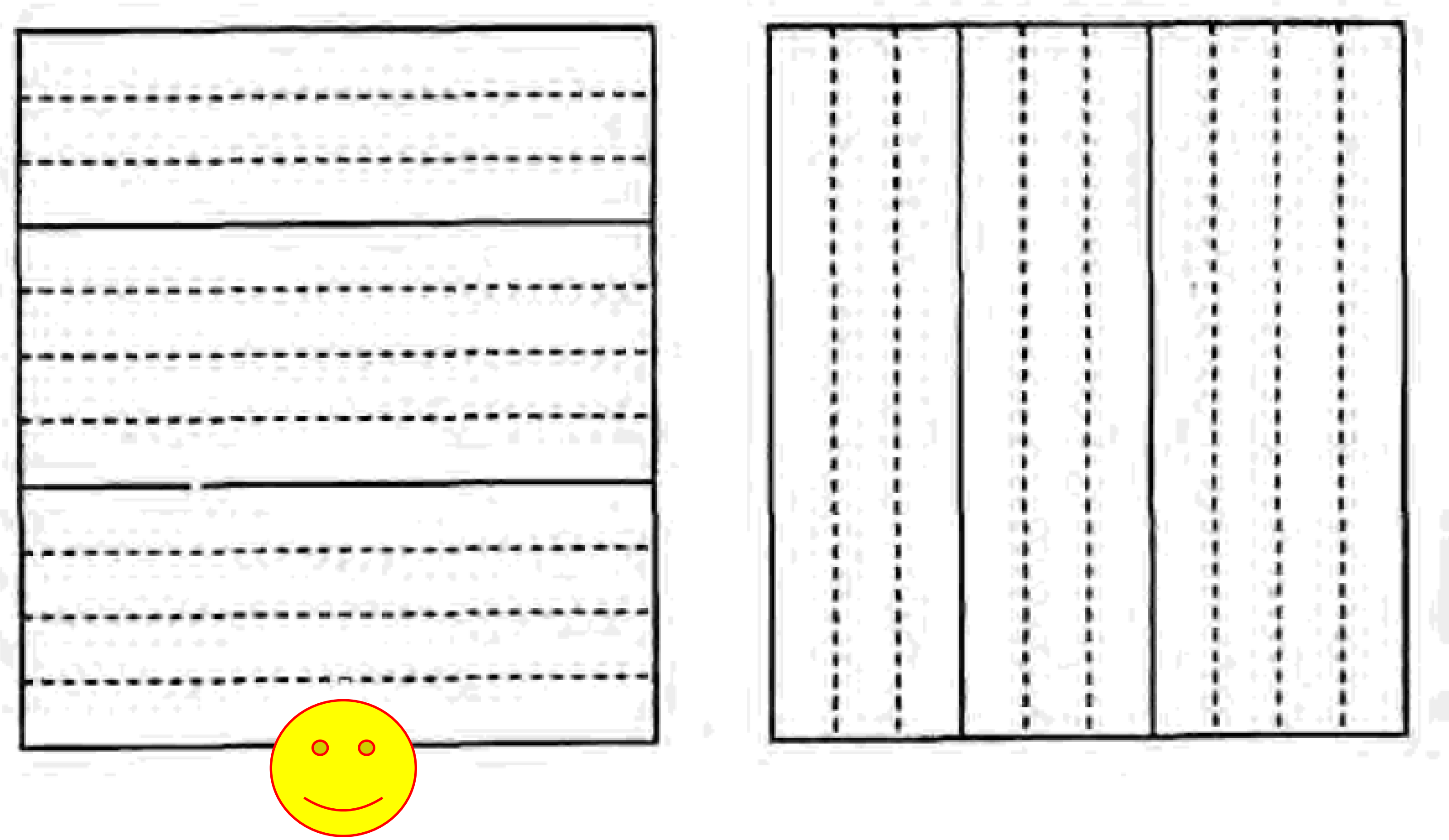
a[0..n-1, 0..n-1]：邻接矩阵

输出：变换后的矩阵a,其中包含最短路径长度

```
For k ← 0 to n – 1:
  for l ← 0 to n – 1:
    for j ← 0 to n – 1:
      a[l,j] ← min(a[l,j], a[l,k]+a[k,j])
    endfor
  endfor
endfor
```

- Floyd算法的多进程并行

按行划分or 按列划分?
按行存储（row major），所以按行划分，连续若干行分给一个进



- Floyd算法的多进程并行

P-1号进程读取数据；全部进程按行分块进行计算；0号进程打印初始邻居矩阵及结果

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

Read_row_striped_matrix(argv[1], (void*)&a, (void*)&storage, MPI_INT, &m, &n,
MPI_COMM_WORLD);
if(m!=n) terminate (id, "Matrix must be square\n");

print_row_striped_matrix((void**)a, MPI_INT, m, n, MPI_COMM_WORLD); //打印邻接矩阵

Compute_shortest_paths(id, p, (int**)a, n); //计算点对间最短路径

print_row_striped_matrix((void**)a, MPI_INT, m, n, MPI_COMM_WORLD); //打印结果

MPI_Finalize();
```

- Floyd算法的多进程并行

假设只有P-1号进程能从文件中读取数据

```
Read_row_striped_matrix(){
    ……
    if (id == (p-1)) {
        for (i = 0; i < p -1; i++) {
            x = fread(*storage, datum_size, BLOCK_SIZE(i, p, *m) * *n, infileptr);
            MPI_Send(*storage, BLOCK_SIZE(I, p, *m)* *n, MPI_INT, i, DATA_MSG, comm);
        }
        x = fread(*storage, datum_size, local_rows * *n, infileptr);
        fclose(infileptr);
    }
    else {
        MPI_Recv(*storage, local_rows * *n, MPI_INT, p-1, DATA_MSG, comm, &status);
    }
}
```

能否让数据在网络上的传输和文件读取同时进行?

- Floyd算法的多进程并行

非阻塞通信:

```
Read_row_striped_matrix(){
……
if (id == (p-1)) {
    for (i = 0; i < p -1; i++) {
        x = fread(*storage, datum_size, BLOCK_SIZE(i, p, *m) * *n, infileptr);
        MPI_Isend(*storage, BLOCK_SIZE(I, p, *m)* *n, MPI_INT, i, DATA_MSG, comm, *req);
    }
    x = fread(*storage, datum_size, local_rows * *n, infileptr);
    fclose(infileptr);
}
else {
    MPI_Recv(*storage, local_rows * *n, MPI_INT, p-1, DATA_MSG, comm, &status);
}
}
```

有什么问题?

- Floyd算法的多进程并行

双缓冲:

```
Read_row_striped_matrix(){
……
if (id == (p-1)) {
    cur_storage = storage[0], cur_req = request[0];
    for (i = 0; i < p -1; i++) {
        x = fread(*cur_storage, datum_size, BLOCK_SIZE(i, p, *m) * *n, infileptr);
        if(i>0) MPI_Wait(cur_req);
        MPI_Isend(*cur_storage, BLOCK_SIZE(I, p, *m)* *n,MPI_INT, i, DATA_MSG, comm, cur_req);
        prev_req = cur_req;
        cur_req = req[i%2];
        cur_storage = storage[i%2];
    }
    MPI_Wait(prev_req);
    x = fread(*cur_storage, datum_size, local_rows * *n, infileptr);
    fclose(infileptr);
}
else {
    MPI_Recv(*storage, local_rows * *n, MPI_INT, p-1, DATA_MSG, comm, &status);
}
}
```

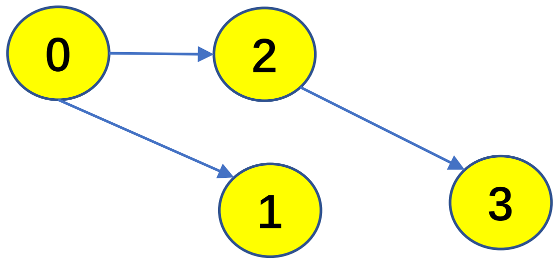
- Floyd算法的多进程并行

计算最短路径 Compute_shortest_paths:

```
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]);
}
```

MPI_Bcast是非阻塞的；
此处没有MPI_Barrier()

Broadcast tree

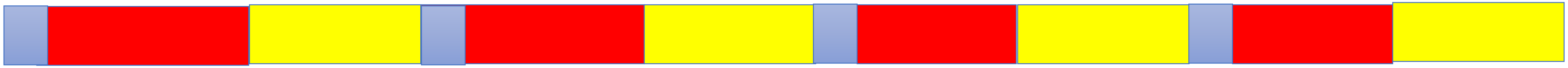


- Floyd算法的多进程并行

总执行时间分析

计算　创建消息　等待

不考虑计算与通信重叠：



χ是更新一个元素的时间

广播需要n次，每次按树形广播，需要logp次通信，通信数据长度是4n, λ是消息延迟, β是带宽

因此总时间如下：

$$n\lceil \log p\rceil(\lambda+4n/\beta)+n^2\lceil n/p\rceil\chi$$

实际运行时，会有计算与通信的重叠

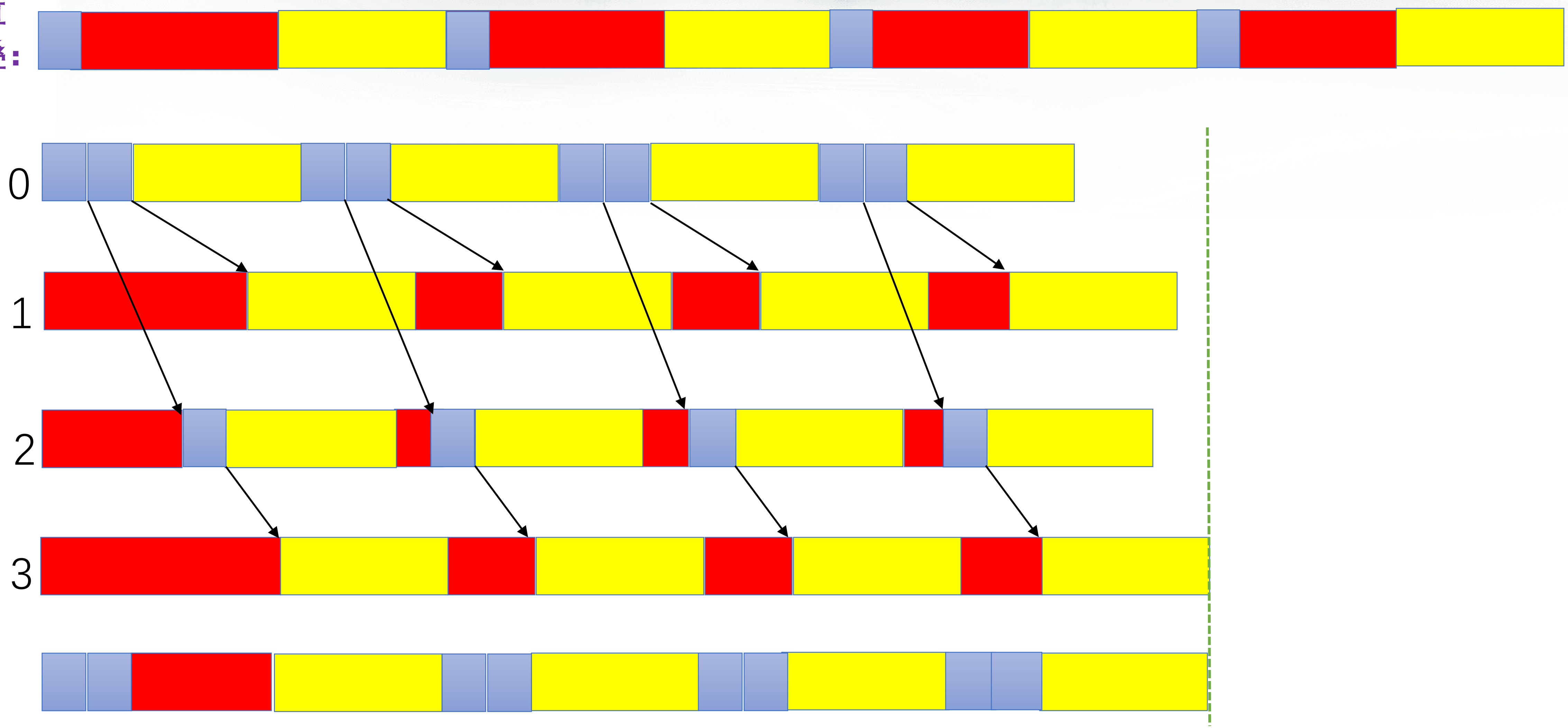
- Floyd算法的多进程并行

总执行时间分析

计算　创建消息　等待

不考虑计算与通信重叠：

考虑计算与通信重叠：



N较大时（计算时间掩盖通信时间）： $n\lceil \log p\rceil\lambda + \lceil \log p\rceil 4n/\beta + n^2\lceil n/p\rceil\chi$

- Floyd算法的多进程并行

```
void print_row_striped_matrix(void**a, MPI_Datatype dtype, int
m, int n, MPI_Comm comm){
if(!id) {
    print_submatrix(a,dtype, local_rows,n);
    datum_size = get_size(dtype);
    max_block_size = BLOCK_SIZE(p-1, p, m);
    bstorage = my_malloc(id, max_block_size *n * datum_size);
    b = (void**) my_malloc(id, max_block_size * sizeof(void*));
    b[0] = bstorage;
    for(i = 1; i < max_block_size; i++)
        b[i] = b[i-1] + n * datum_size;
    for(i=1; i < p; i++) {
        MPI_Send(&promp, 1, MPI_INT, i, PROMPT_MSG,
MPI_COMM_WORLD);
        MPI_Recv(bstorage, BLOCK_SIZE(I, p, m) * n, MPI_INT, I,
RESPONSE_MSG, MPI_COMM_WORLD, &status);
       print_submatrix(b, dtype, BLOCK_SIZE(I, p, m), n);
    }
  Free(b);
    Free(bstorage);
    putchar( '\n' );
} else {
    MPI_Recv(&promp, 1, MPI_INT, 0,
PROMPT_MSG, MPI_COMM_WORLD,
&status);
    MPI_Send(*a, local_rows * n, MPI_INT, 0,
RESPONSE_MSG, MPI_COMM_WORLD);
}
}
```

先发送一个消息，再传递数据：为了避免大量消息同时发送给0号进程。（还可以用什么操作？）

- Floyd算法的多进程并行

死锁问题（教科书6.5.3节）：

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

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

- Floyd算法的多进程并行

MPI_Recv中的Status参数

- 在MPI_Recv中已经指定了发送消息的进程号和tag，为什么还要在状态记录中还要查询这些信息?
  - 在指定参数时，如果接收进程是MPI_ANY_SOURCE，可接收任务进程发来的消息
  - 如果接收tag是MPI_ANY_TAG，可以接收任何标号的信息
  - 上述情况需要通过查询状态记录来确定发送方
  - status->MPI_source
  - status->MPI_tag
  - status->MPI_error
  - 注意！不同的MPI版本status记录的格式和大小可能不同