Introduction to MPI

2023.7.10

陈书扬，ZJUSCT

Contents

什么是MPI？

消息传递接口（Message Passing Interface，MPI）是一个并行计算的应用程序接口，常在超级电脑、电脑集群等非共享内存环境程序设计。

MPI中提供了用于进程间通讯的标准接口，用于实现进程级别的并行。

MPI是一组接口标准，而不是实现。常见的MPI实现有:

- OpenMPI
- IntelMPI
- HPC-X

- MPI和OpenMP的区别？

MPI和OpenMP的区别？

```c
for (int i = 0; i < 64; ++i) {
    a[i] += i;
}
```

```c
#include <omp.h>
#pragma omp parallel for
for (int i = 0; i < 64; ++i) {
    a[i] += i;
}
```

✓

```c
#include <mpi.h>
MPI_Init(&argc, &argv);
int rank; // rank = [0, 64)
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
a[rank] += rank;
```

MPI和OpenMP的区别？

OpenMP	MPI
线程间并行	进程间并行
共享内存	独立内存
初始化开销小	初始化开销大
适用于单个节点内	适用于节点间
自动分配线程任务	手动分配进程任务

Hello, World: MPI的安装

- OpenMPI: Lab1
- IntelMPI: 一般与oneAPI一起安装，或者单独安装
- HPC-X: https://developer.nvidia.com/networking/hpc-x

Hello, World

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

int main(int argc, char* argv[]) {

    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello world from process %d in %d processes\n", rank, size);

    MPI_Finalize();
}
```

Hello, World -- compile and launch

compile:

```
mpicc mpi_example.c -o a.out
```

- mpicc并不是一个新的编译器
- 其他编译器：mpiicc, mpicxx, ...

launch:

```
mpirun -n 4 --host m601:2,m603:2 ./a.out
```

- 使用hostfile

应该使用多少进程？

- 理论上，最多能使用多少进程？

一般来说，进程的数量不应该超过核心的数量

```
$ lscpu
CPU(s):                    64      (逻辑CPU数)
On-line CPU(s) list:       0-63
Thread(s) per core:        2       (每个核的超线程数)
Core(s) per socket:        16      (每个物理CPU的核心数)
Socket(s):                 2       (物理CPU个数)
```

- 并不是在核心数量以下，进程越多越好。适当减少进程的数量可能会带来性能增益
- cpuinfo(intelmpi)

MPI通信函数

- 简单通信
- 非阻塞通信
- 集合通信

参考文档/网站

- https://www.open-mpi.org/doc/v4.0/
- mpitutorial.com
- https://rookiehpc.org/mpi/docs/index.html

简单通信

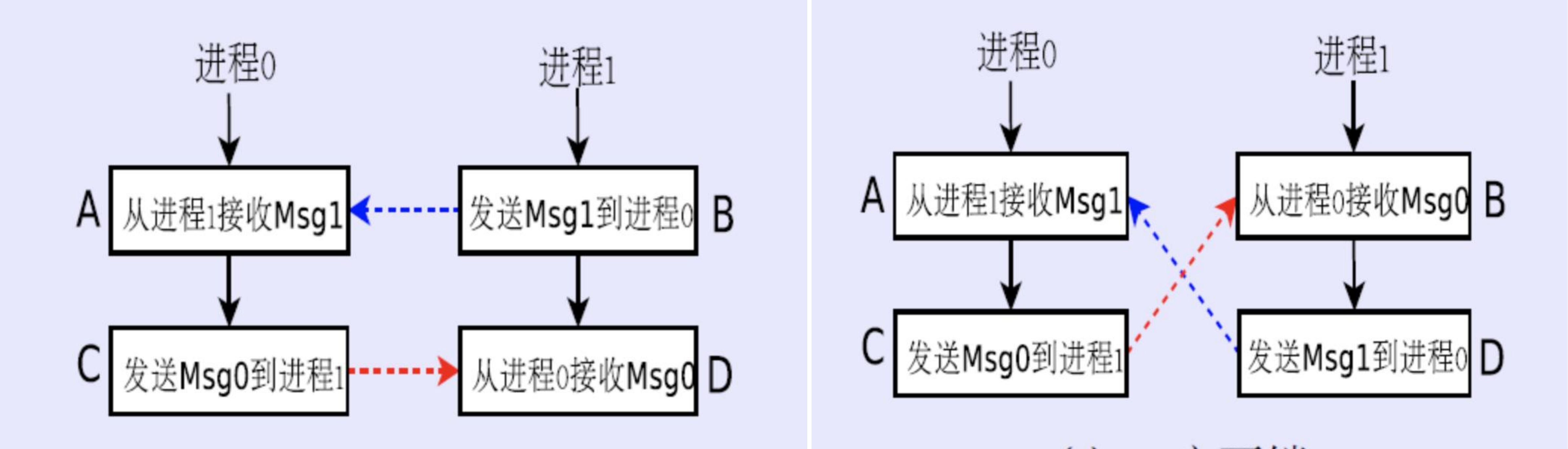
```
MPI_Send(void* data, int count, MPI_Datatype datatype, int destination,
int tag, MPI_Comm communicator)
```

data 发送数据的地址
count 发送数据的元素个数
datatype 发送数据类型(MPI_INT, MPI_CHAR, MPI_FLOAT, ...)
destination 接收方的rank
tag 用于标识信息，收发方的tag必须一致才能接收
communicator 通信域(一般为MPI_COMM_WORLD)

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

tag 可以为MPI_ANY_TAG
status 关于发送方的信息(可以为MPI_STATUS_IGNORE)

简单通信：死锁



(I) 不会死锁

(j) 一定死锁

简单通信：死锁的解决

```
int MPI_Sendrecv(void* buffer_send, int count_send, MPI_Datatype datatype_send,
                  int recipient, int tag_send,
                 void* buffer_recv, int count_recv, MPI_Datatype datatype_recv,
                 int sender,     int tag_recv,
                 MPI_Comm communicator, MPI_Status* status);
```

buffer_send和buffer_recv不能是同一个

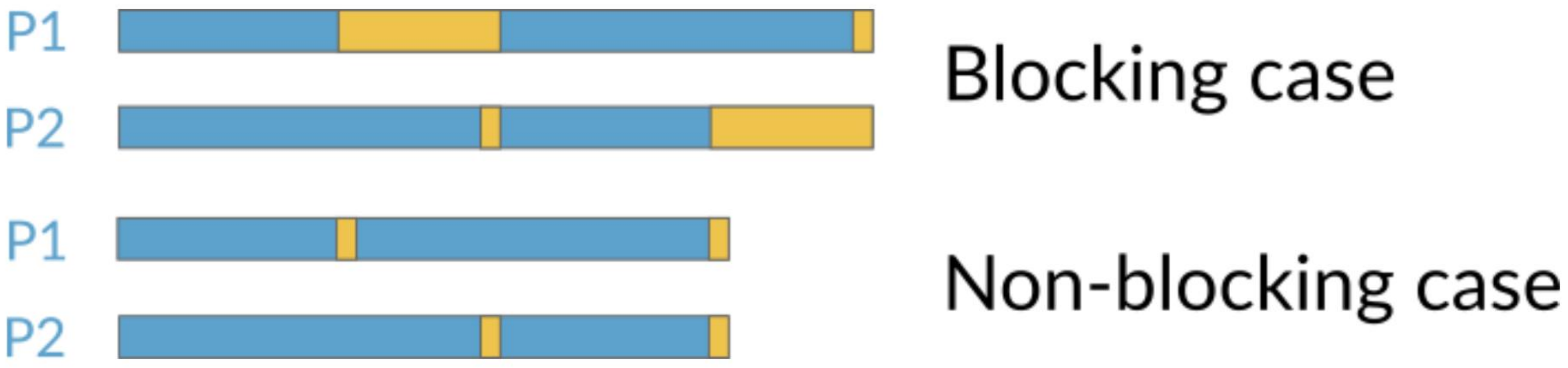
```
int MPI_Sendrecv_replace(void* buffer, int count_send, MPI_Datatype
datatype_send, int recipient, int tag_send, int sender, int tag_recv, MPI_Comm
communicator, MPI_Status* status);
```

简单通信：ping pong (from mpitutorial)

```
int count = 0;
int partner = (rank + 1) % 2;
while (count < LIMIT) {
    if (rank == count % 2) {
        // Increment the ping pong count before you send it
        count++;
        MPI_Send(&count, 1, MPI_INT, partner, 0, MPI_COMM_WORLD);
        printf("rank %d sent and incremented count %d to rank %d\n",
               rank, count, partner);
    }
    else {
        MPI_Recv(&count, 1, MPI_INT, partner, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        printf("rank %d received count %d from rank %d\n",
               rank, count, partner);
    }
}
```

非阻塞通信（异步通信）

为什么需要非阻塞通信？



非阻塞通信：相关接口

```
int MPI_Isend(void* buffer, int count, MPI_Datatype datatype, int
recipient, int tag, MPI_Comm communicator, MPI_Request* request);
```

```
int MPI_Irecv(void* buffer, int count, MPI_Datatype datatype, int sender,
int tag, MPI_Comm communicator, MPI_Request* request);
```

测试通信是否完成:

```
int MPI_Test(MPI_Request* request, int* flag, MPI_Status* status);
```

等待通信完成:

```
int MPI_Wait(MPI_Request* request, MPI_Status* status);
```

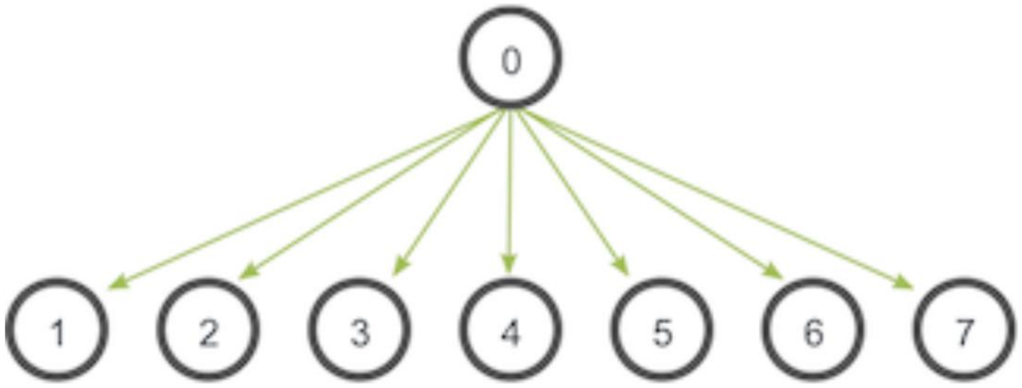
```
int MPI_Waitall(int count, MPI_Request requests[], MPI_Status statuses[]);
```

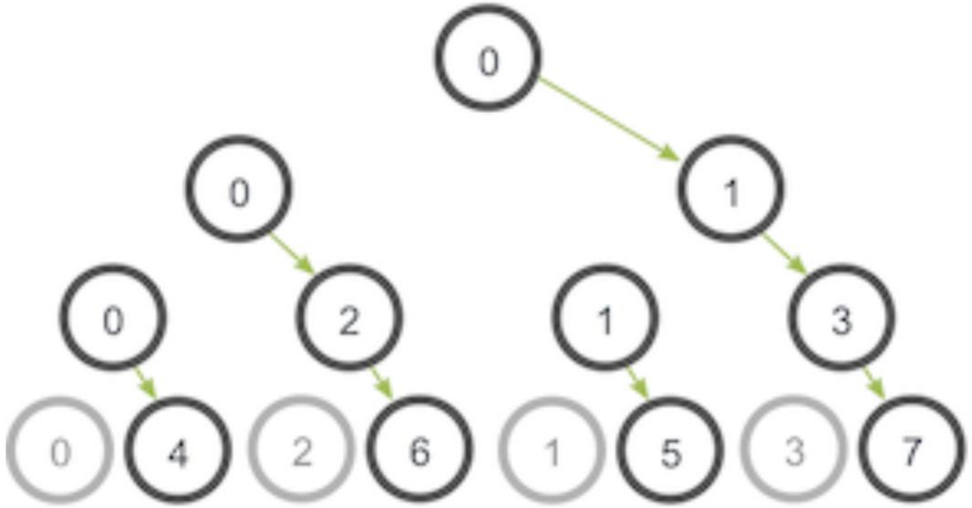
集合通信

点对点通信 --> 单点对多点通信
多点对单点通信
多点对多点通信

- 多点的通信都可以用单点通信来实现。为什么需要专门的集合通信？

考虑广播操作（从0号进程将数据分发给所有进程）：

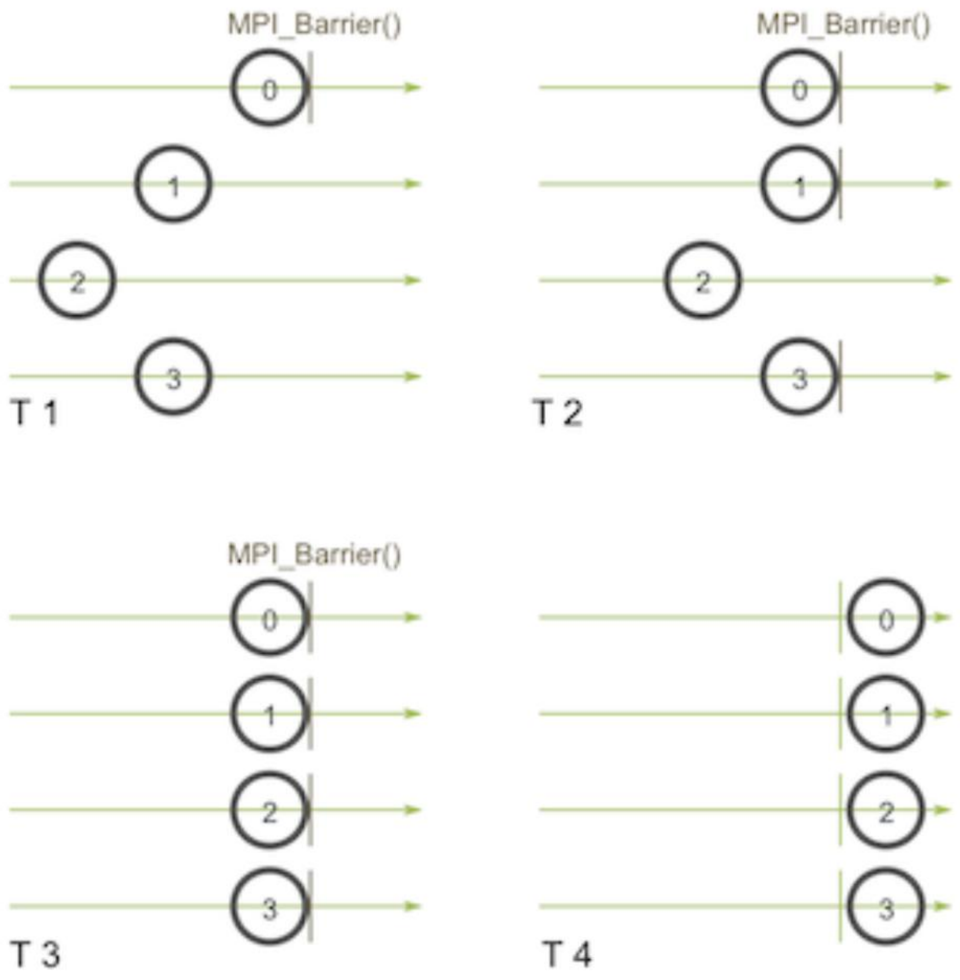




集合通信：接口

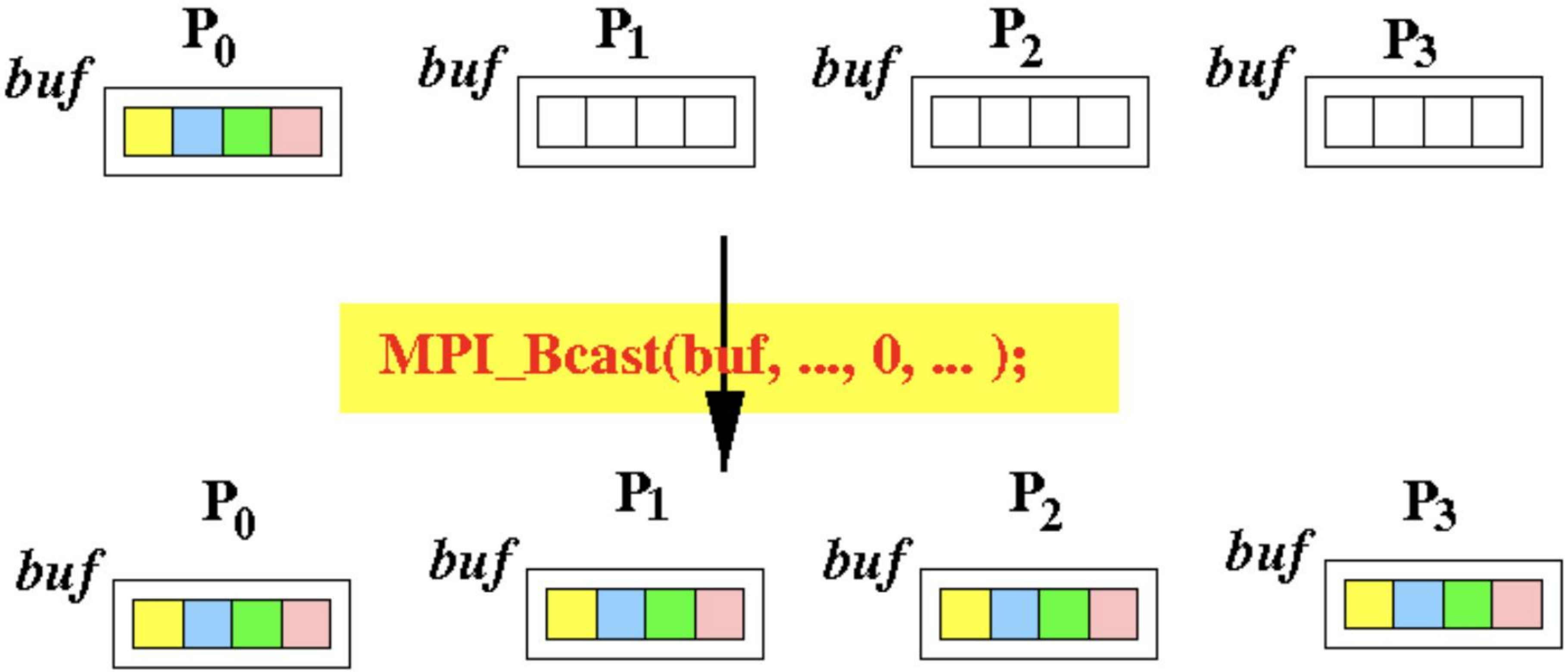
```
int MPI_Barrier(MPI_Comm communicator);
```

所有集合通讯在进行之前，发送数据的所有进程必须达到同一个时间点



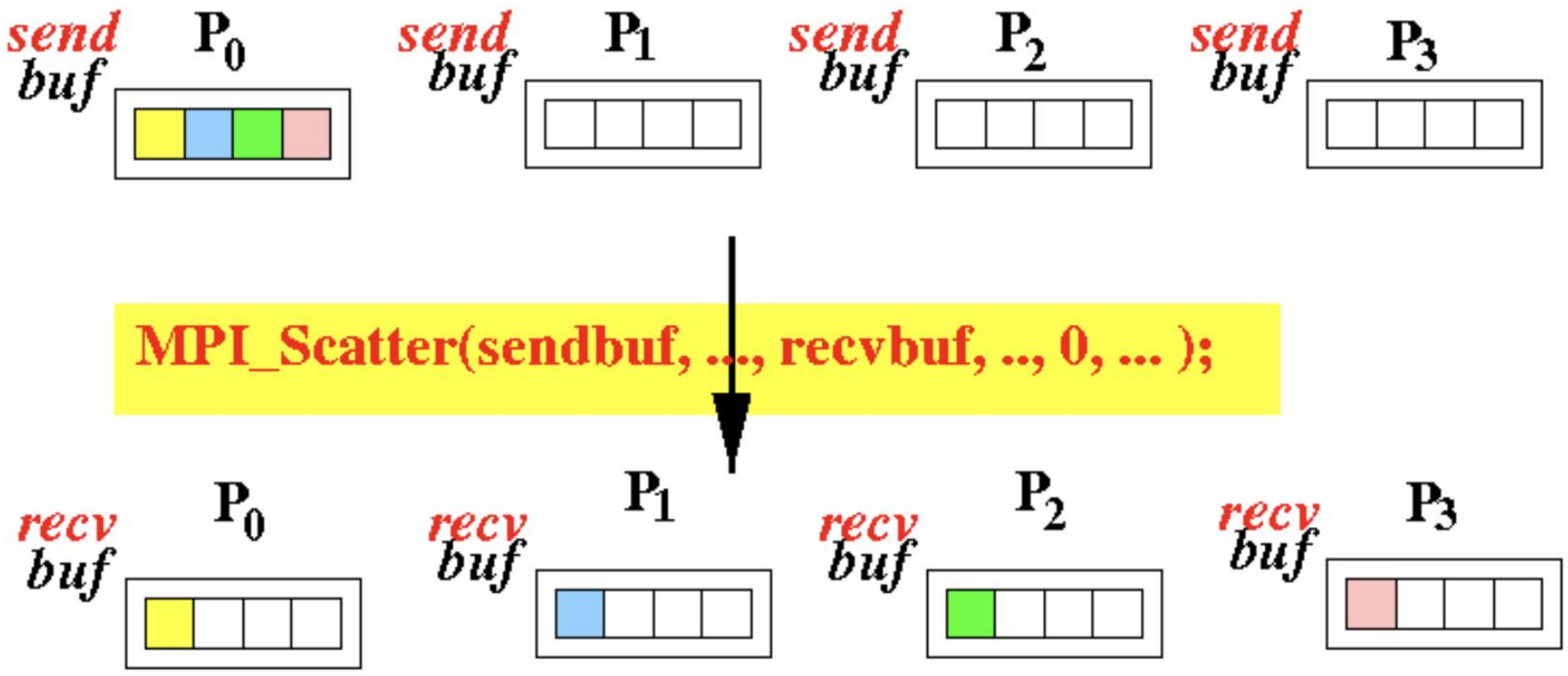
集合通信：接口

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
int emitter_rank, MPI_Comm communicator);
```



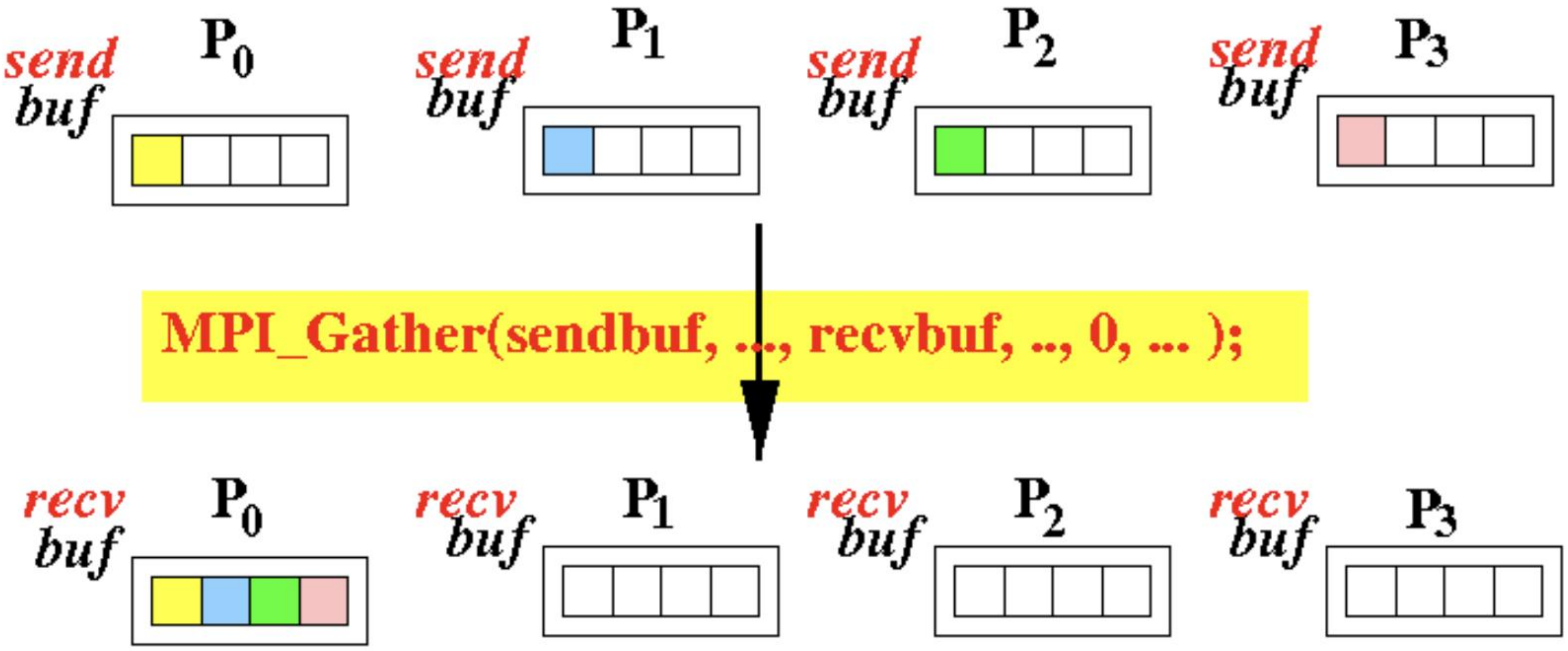
集合通信：接口

```
int MPI_Scatter(...);
```



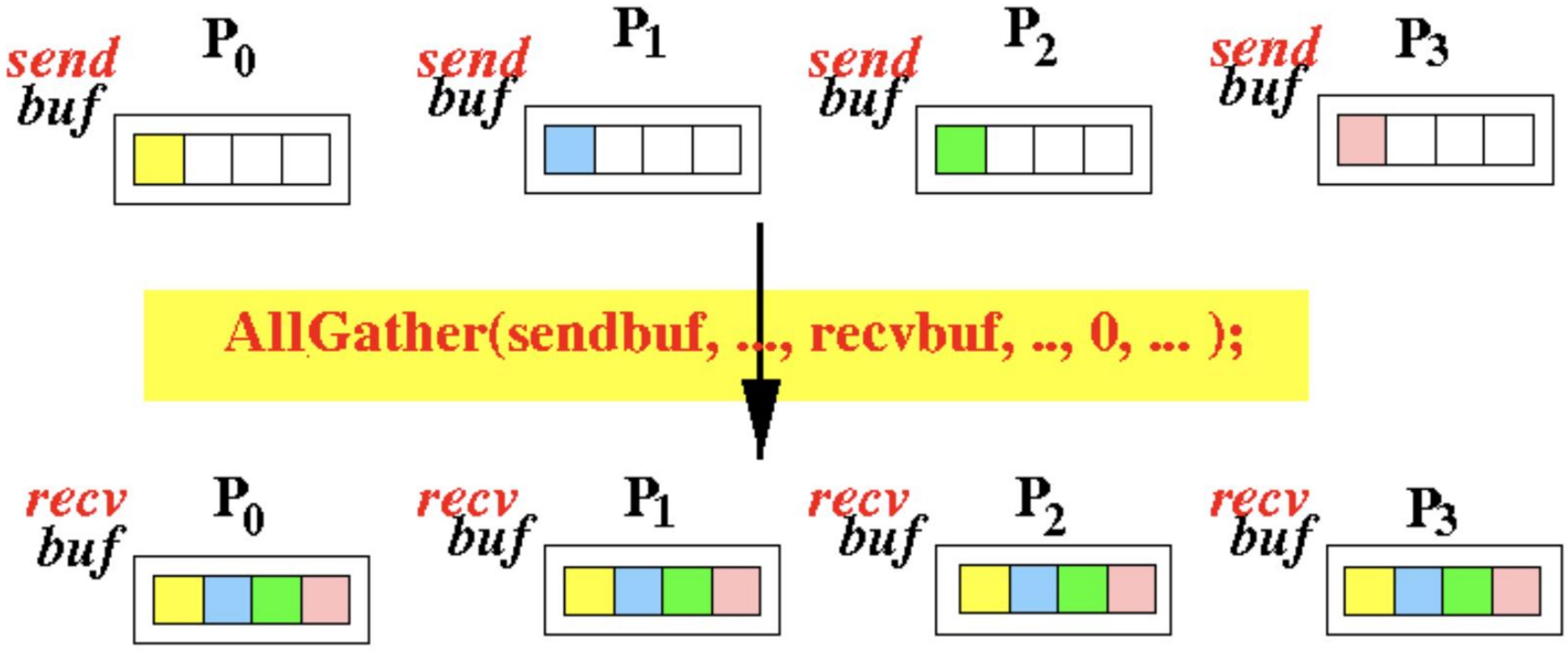
集合通信：接口

```
int MPI_Gather(...);
```



集合通信：接口

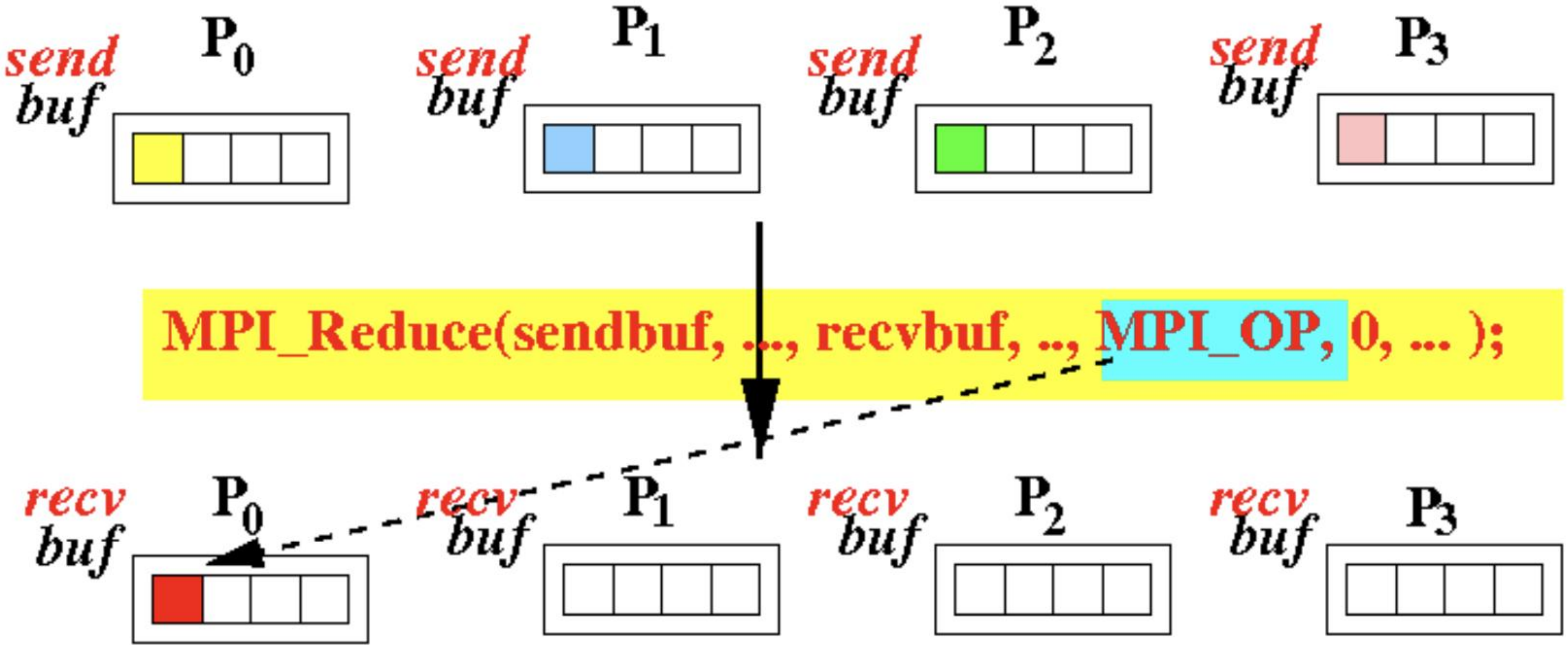
```
int MPI_Allgather(...);
```



集合通信：接口

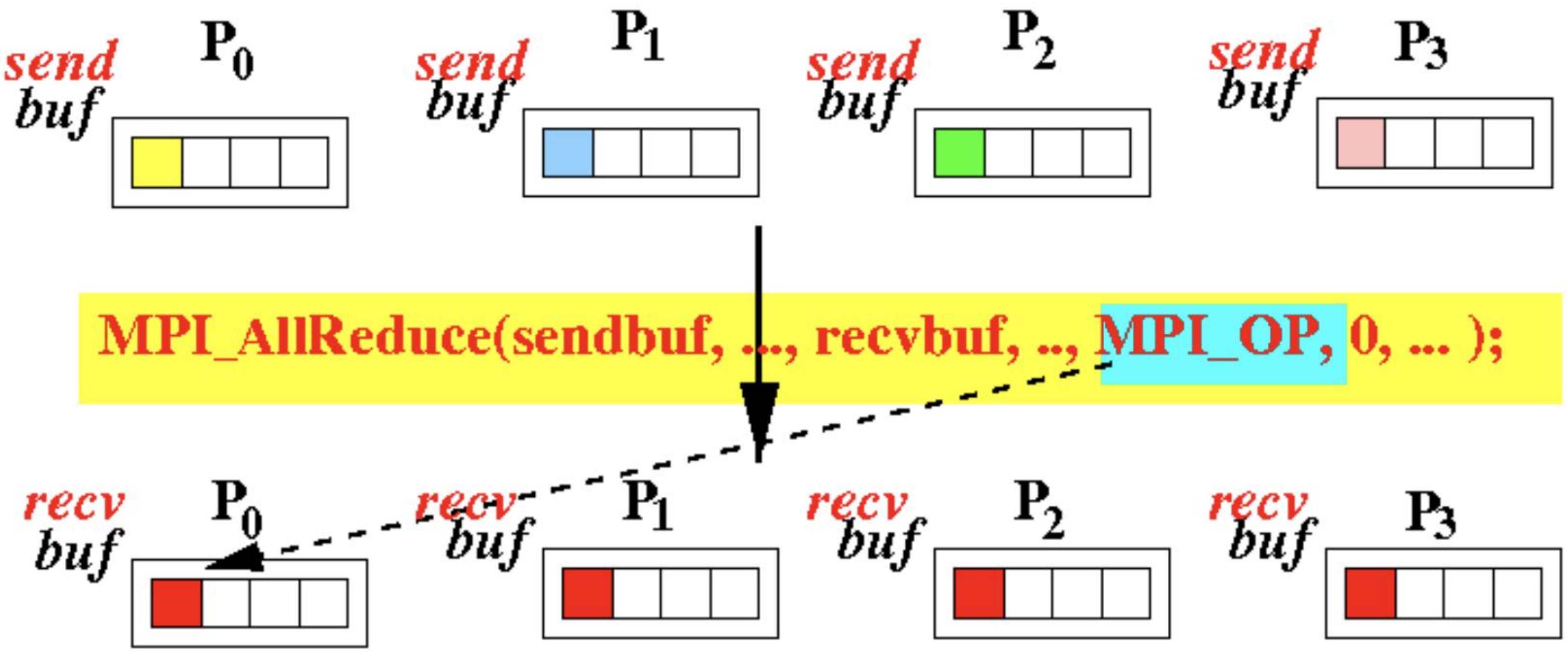
```
int MPI_Reduce(..., MPI_Op operation, ...);
```

MPI_Op可以为MPI_MAX，MPI_MIN，MPI_SUM，MPI_PROD(累乘)



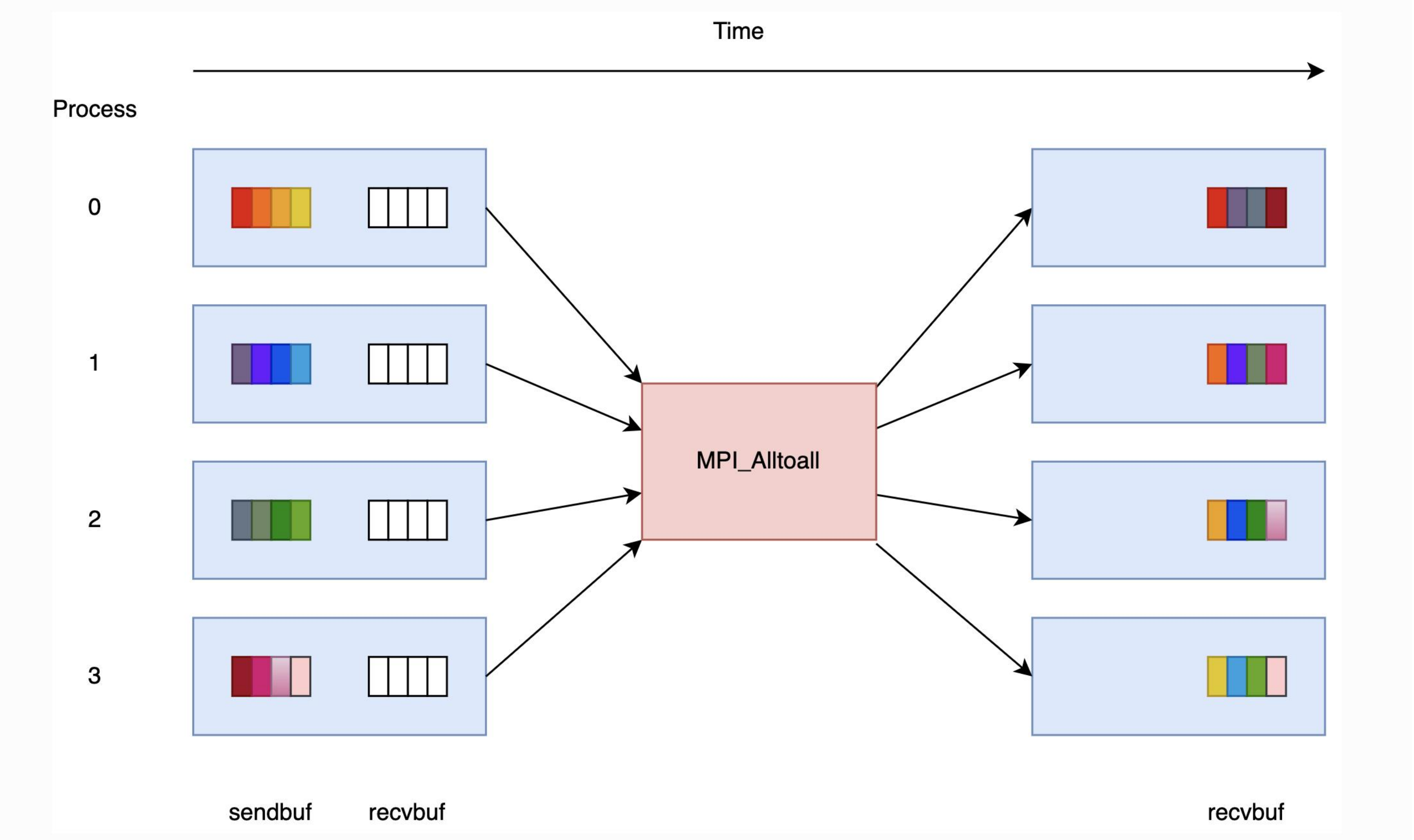
集合通信：接口

```
int MPI_Allreduce(...);
```



集合通信：接口

```
int MPI_Alltoall(...);
```



集合通信：总结

Operation	MPI Function	Synopsis
		Individual
Send	`MPI_Send`	One-to-one send
Receive	`MPI_Recv`	One-to-one receive
Send/Receive	`MPI_Sendrecv`	One-to-one send/receive
		Collective
Barrier	`MPI_Barrier`	All wait for stragglers
Broadcast	`MPI_Bcast`	Root to all, all data copied
Scatter	`MPI_Scatter`	Root to all, slices of data copied
Gather	`MPI_Gather`	All to root, slices ordered on Root
Reduce	`MPI_Reduce`	All to root, data reduced on Root
All-Gather	`MPI_Allgather`	All to all, data ordered
All-Reduce	`MPI_Allreduce`	All to all, data reduced

Profiling(性能分析)

通过Profiling，我们期望得到：

- 程序的瓶颈是哪一种类型？访存/计算/通信
- 程序的每一个模块/函数分别用时多少？消耗时间最多的是哪一部分？
- MPI通信的开销主要在哪些部分上？

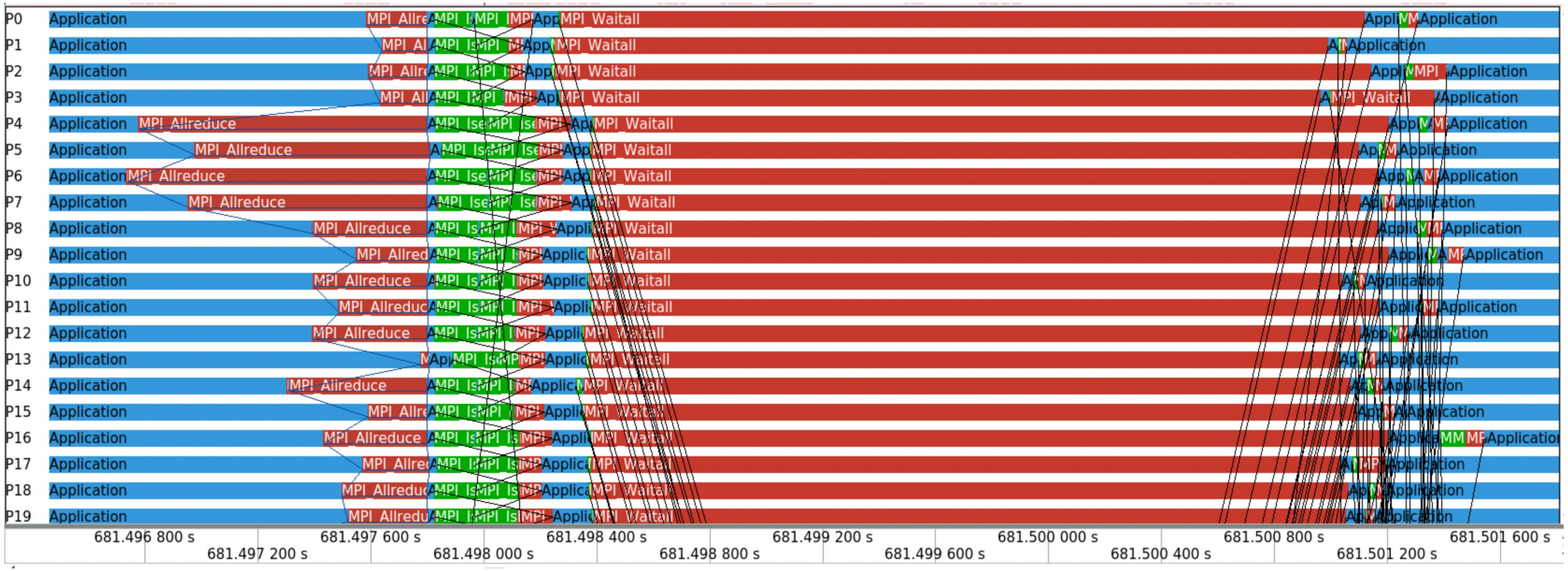
Profiling(性能分析)

IntelMPI：

- Application Performance Snapshot (APS)
- Intel Trace Analyzer and Collector (ITAC)
- VTune

HPC-X:

- IPM Profiler



其他

- Process Binding
- 变长通信
- 单边通信
- ...

预告：Lab4 MPI加速PCG算法

预处理共轭梯度法(Preconditioned Conjugate Gradient method, PCG)是一种求解线性方程组Ax = b的迭代算法，适用于稀疏矩阵的求解。广泛见于科学计算程序中。

在这个实验中，你需要:

- 使用MPI，将PCG算法并行化并尽可能加速
- 使用Profile工具对得到的MPI程序进行性能分析
- 使用Fortran完成这个实验(bonus)

Thanks for listening

2023.7.10