并行与分布式计算基础：第九讲

杨超
chao_yang@pku.edu.cn

2019 秋

课程基本情况

- 课程名称：并行与分布式计算基础
- 授课教师：杨超（chao_yang@pku.edu.cn，理科 1 号楼 1520）
- 课程助教：尹鹏飞（pengfeiyin@pku.edu.cn）

授课内容（暂定）

- 引言
- 硬件体系架构
- 并行计算模型
- 编程与开发环境
- MPI 编程与实践
- OpenMP 编程与实践
- GPU 编程与实践
- 前沿问题选讲

上课时间（地点：二教 211）

上课时间	星期一	星期二	星期三	星期四	星期五
第 1 节 (8:00-8:50)					
第 2 节 (9:00-9:50)					
第 3 节 (10:10-11:00)				单周	
第 4 节 (11:10-12:00)				单周	
第 5 节 (13:00-13:50)		每周			
第 6 节 (14:00-14:50)		每周			
第 7 节 (15:10-16:00)					
第 8 节 (16:10-17:00)					
第 9 节 (17:10-18:00)					
第 10 节 (18:40-19:30)					
第 11 节 (19:40-20:30)					
第 12 节 (20:40-21:30)					

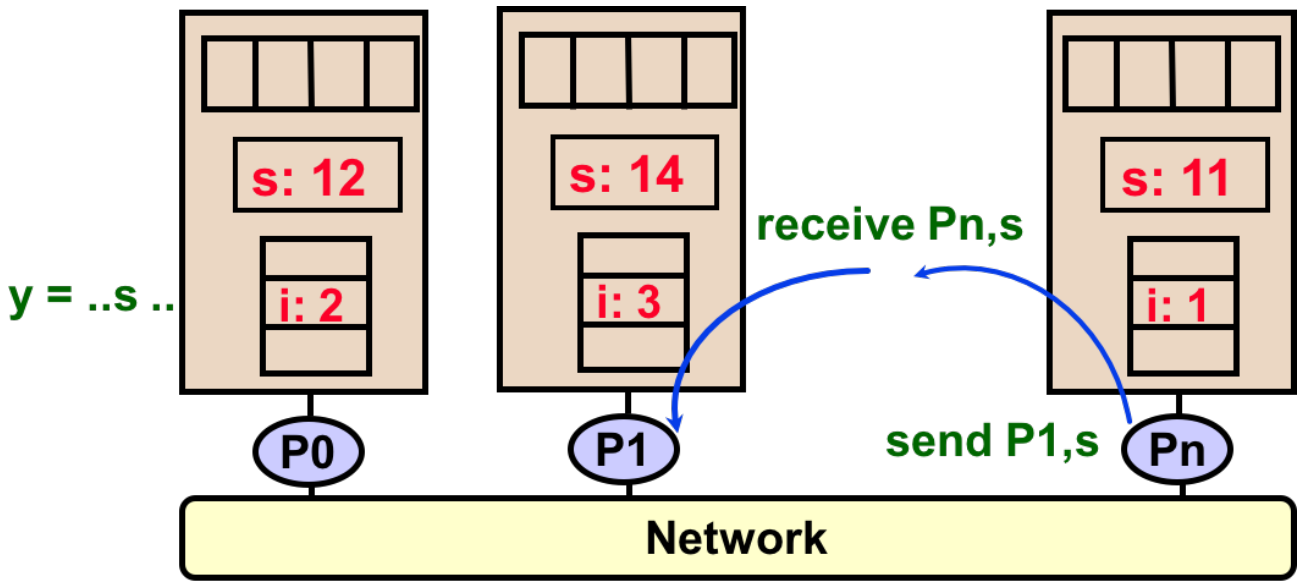
内容提纲

MPI 基础知识回顾

什么是 MPI?

MPI = Message Passing Interface

- 是一组由学术界和工业界联合发展的、面向主流并行计算机的、标准化和可移植的消息传递接口标准;
- 适用于目前几乎所有主流并行计算机，已经成为事实上的工业标准;
- 每个进程拥有私有的存储空间，进程间只能通过消息传递通信;
- 程序往往采用 SPMD (single program multiple data) 方式编写。



MPI 的六个基本函数

- 初始化/终止:

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

- 获取进程信息:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- 发送/接收消息:

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

MPI 点对点通信

- 两个进程进行数据交换:

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype
  senddatatype, int dest, int sendtag, void *recvbuf, int
  recvcount, MPI_Datatype  recvdatatype, int source, int recvtag,
   MPI_Comm comm, MPI_Status *status)
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype
  datatype, int dest, int sendtag, int source, int recvtag,
  MPI_Comm comm, MPI_Status *status)
```

- 非阻塞 (non-blocking) 通信:

```
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)
```

request 变量用来标记通信任务。

非阻塞通信状态的检测与控制

- 取消非阻塞通信:

```
int MPI_Cancel(MPI_Request *request)
```

- 检测非阻塞通信是否已经结束 (flag 值为 0 表示未结束):

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Testall(int count, MPI_Request requests[], int *flag,
    MPI_Status statuses[])
int MPI_Testany(int count, MPI_Request requests[], int *index, int *
    flag, MPI_Status *status)
```

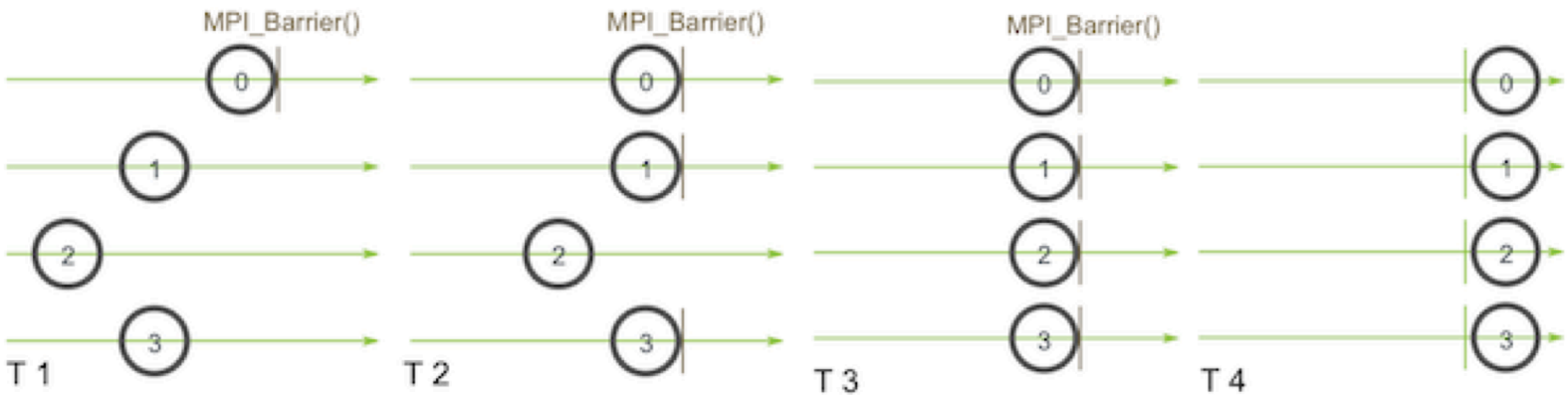
- 等待非阻塞通信结束:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
int MPI_Waitall(int count, MPI_Request requests[], MPI_Status
    statuses[])
int MPI_Waitany(int count, MPI_Request requests[], int *index,
    MPI_Status *status)
```

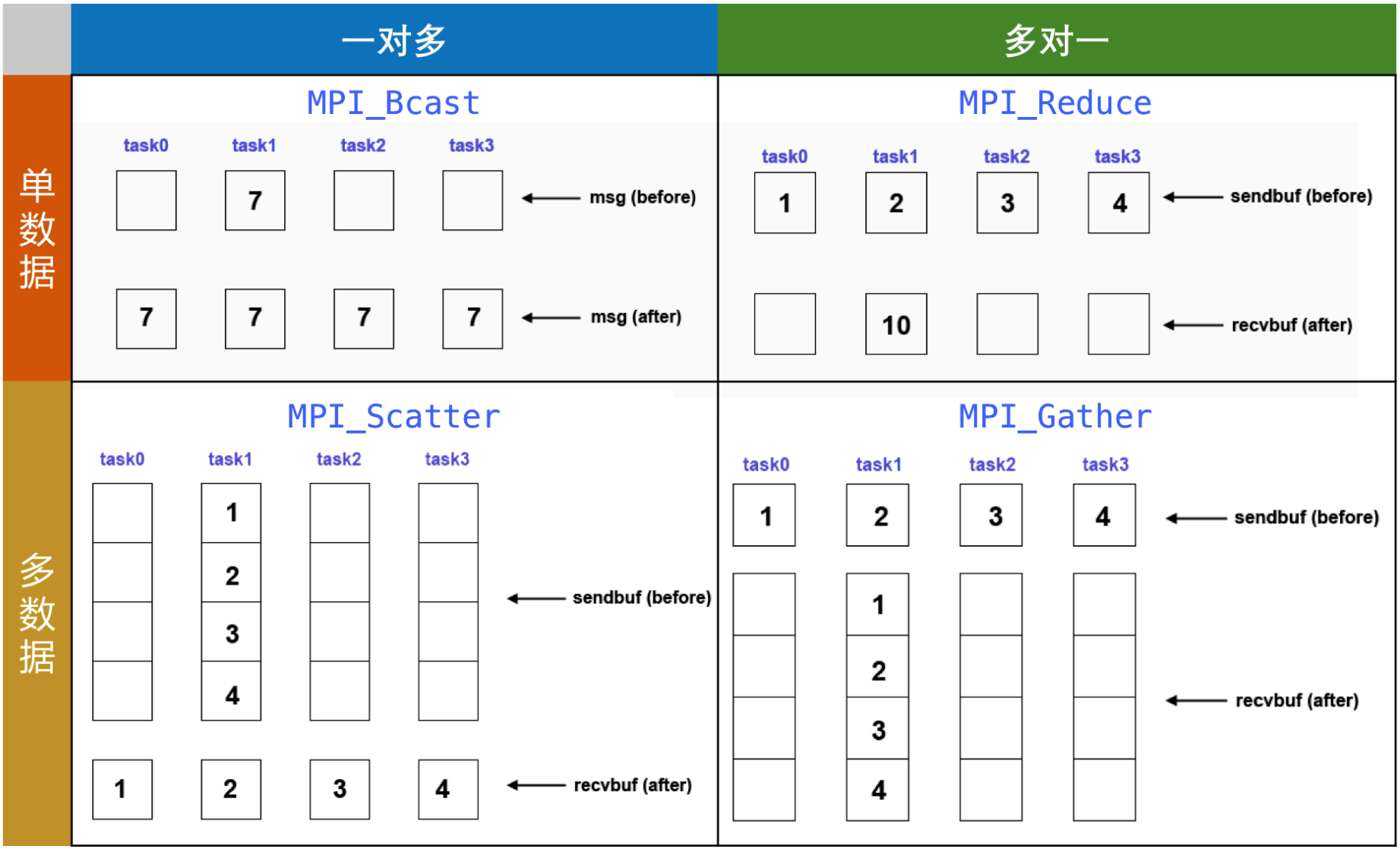
MPI 栅栏同步

- 通信器中所有进程相互等待至某个同步点：

```
int MPI_Barrier(MPI_Comm comm)
```



MPI “一对多”与“多对一”通信



MPI “多对多”通信

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

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

```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype
  senddatatype, void *recvbuf, int recvcount, MPI_Datatype
  recvdatatype, MPI_Comm comm)
```

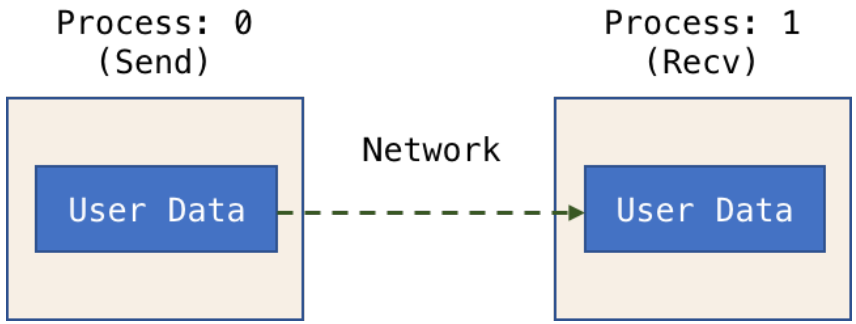
```
int MPI_Reduce_scatter(void *sendbuf, void *recvbuf, const int
  recvcount[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype
  senddatatype, void *recvbuf, int recvcount, MPI_Datatype
  recvdatatype, MPI_Comm comm)
```

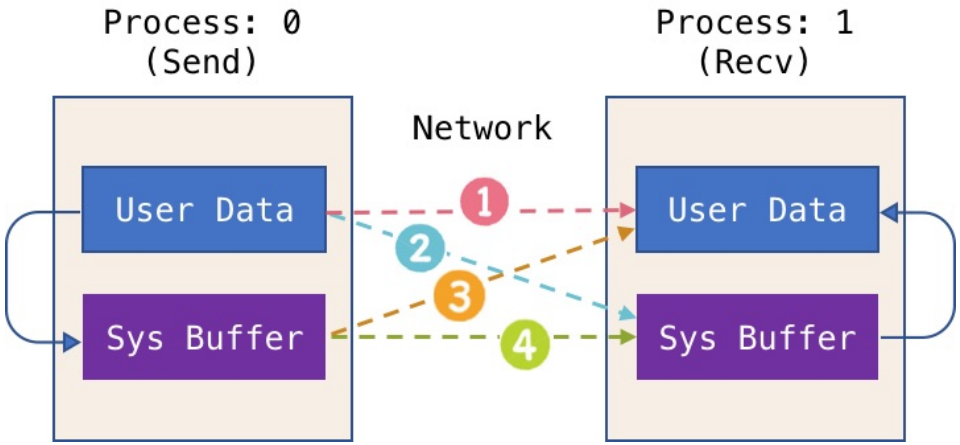
MPI 点对点通信-2

MPI 系统缓冲 (system buffer)

- MPI Send/Recv 的接口：



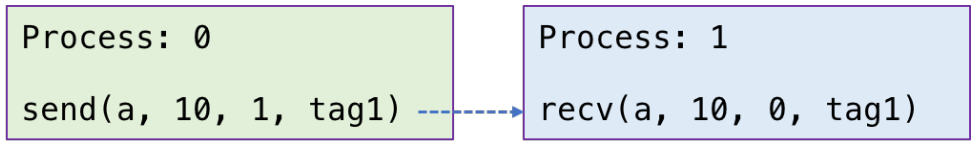
- 借助 MPI 系统缓冲，Send/Recv 的底层实现有多种方式：



- MPI 会根据情况 (底层实现、消息大小等) 选择合适的方式。

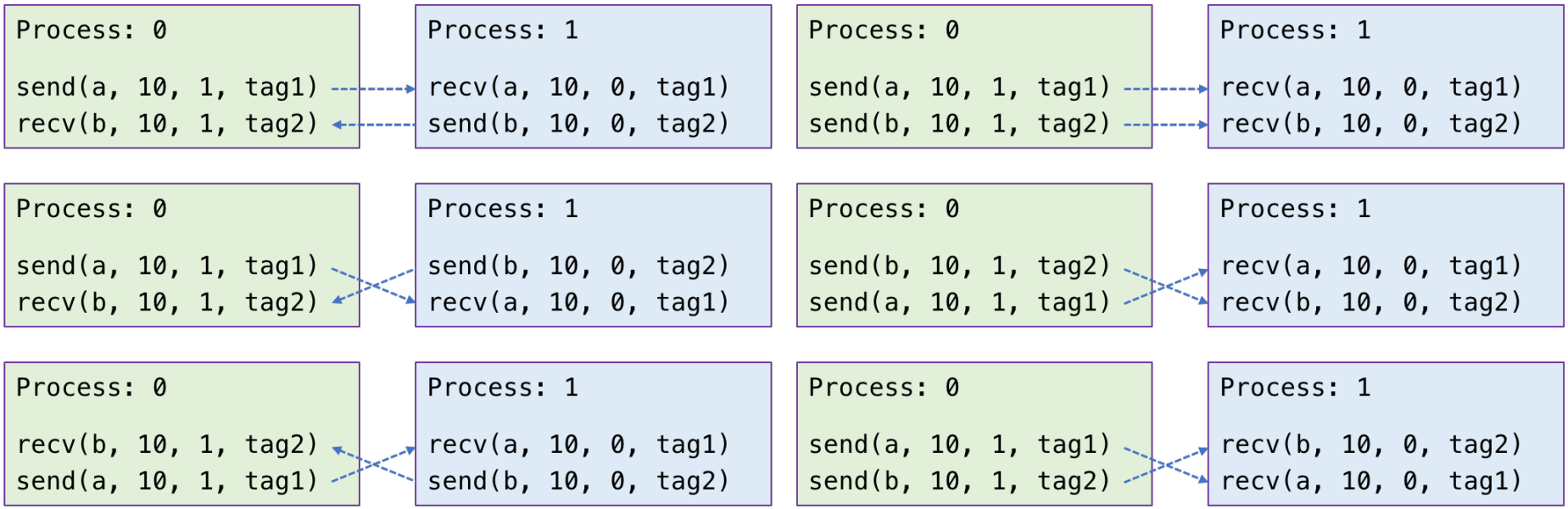
阻塞通信的再思考

- 阻塞发送/接收函数，哪个可以提前返回?



  - 发送函数：将消息拷贝至系统缓冲 (如可用！)；
  - 接收函数：收到消息。

- 讨论：如下哪些情况一定产生死锁? 哪些可能产生? 哪些一定不会?



发送函数的分类 (1)

- 同步 (synchronous) 模式：MPI_Ssend
  - 无论接收端是否启动接收，发送端可在任意时间启动发送；
  - 只有在接收端启动接收后，发送端才返回；
  - 发送端返回不仅表示缓冲区可以使用，还表示接收端已经到达了某个程序点 (进行了握手同步)。
- 就绪 (ready) 模式：MPI_Rsend
  - 仅当对方的接收操作启动且准备就绪，才发送数据 (否则报错)；
  - 语义上和同步发送完全一致，避免了额外的缓冲区操作和发送接收方的握手操作。
- 缓冲 (buffered) 模式：MPI_Bsend
  - 发送端把数据拷贝到用户提供的临时缓冲区；
  - 函数返回时，发送缓冲区可以用。
- 上述三种模式与 MPI_Send 语法完全相同。

缓冲发送的用法

- 缓冲发送函数 MPI_Bsend 在使用前后需要用户使用如下函数显式地指定和释放内部缓冲区：

```
int MPI_Buffer_attach(void* buffer, int length)
int MPI_Buffer_detach(void* buffer_addr, int* length_addr)
```

- 缓冲区应不小于发送数据所需的总空间，在此基础上还需要加上额外空间长度 MPI_BSEND_OVERHEAD.
- 对比区分：程序中的发送缓冲和用户指定的 bsend 缓冲区.

练习 (1)

- 研究并测试 mpi_bsend.c 程序，找出程序的问题.

```
1  ...
2  #define n 300
3  int main(int argc, char** argv) {
4    int size, rank, i, a[n], b[n];
5    int buf_len = MPI_BSEND_OVERHEAD + n*sizeof(int);
6    int *buf = malloc(buf_len);
7
8    MPI_Init(&argc, &argv);
9    MPI_Comm_size(MPI_COMM_WORLD, &size);
10   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11
12   if (rank == 0) {
13     for (i = 0; i < n; i++) {
14       a[i] = i+10; b[i] = -i-100;
15     }
16     MPI_Buffer_attach(buf, buf_len);
```

练习 (2)

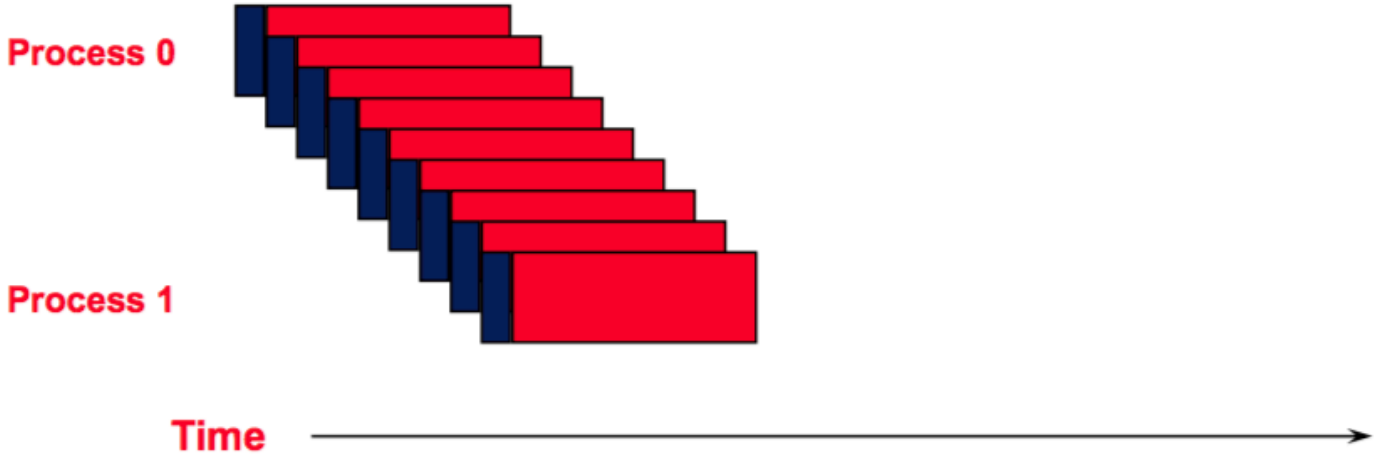
```
17      MPI_Bsend(a, n, MPI_INT, 1-rank, 0, MPI_COMM_WORLD);
18      MPI_Bsend(b, n, MPI_INT, 1-rank, 1, MPI_COMM_WORLD);
19      MPI_Buffer_detach(&buf, &buf_len);
20    } else if (rank == 1) {
21      MPI_Recv(b, n, MPI_INT, 1-rank, 1, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
22      printf("Process %d received %d intergers (first entry: %d)
        from process %d\n", rank, n, b[0], 1-rank);
23      MPI_Recv(a, n, MPI_INT, 1-rank, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
24      printf("Process %d received %d intergers (first entry: %d)
        from process %d\n", rank, n, a[0], 1-rank);
25    }
26    free(buf);
27    ...
```

发送函数的分类 (2)

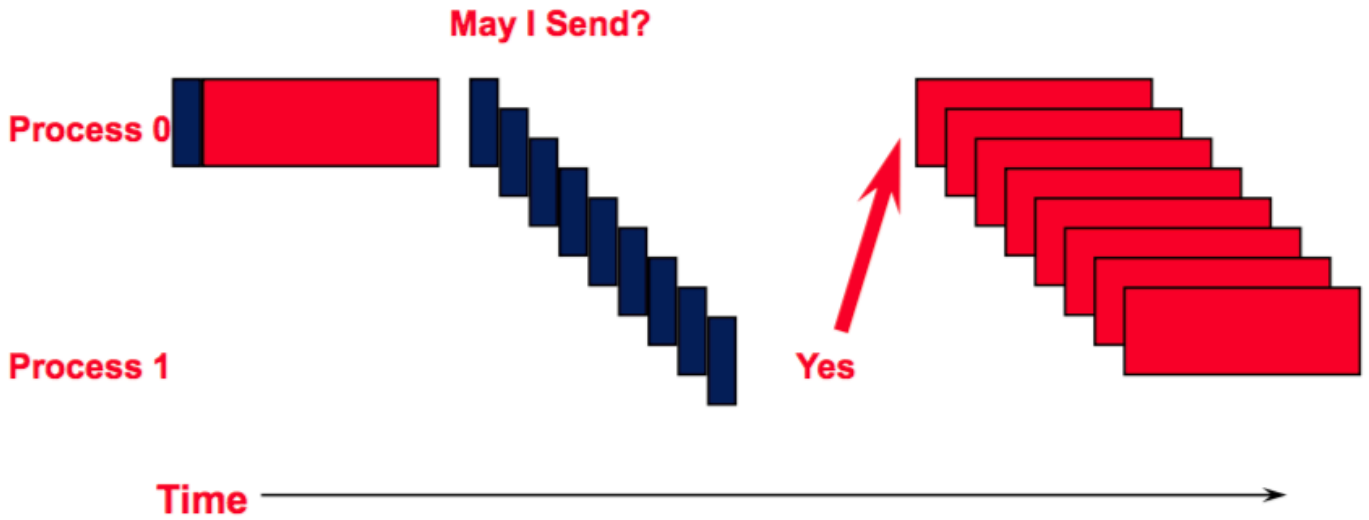
- 标准 (standard) 模式：MPI_Send
  - 可以是同步的或缓冲的，给予系统以灵活选择的机会；
    - ★ 对短消息，一般采用缓冲模式；
    - ★ 对长消息，一般采用同步模式，不同在于数据传输完成才返回；
    - ★ 长短消息的切换点，可以配置。
- 非阻塞发送
  - 上述发送也有对应的非阻塞版本，但是极少使用。
- 一些建议
  - 发送函数：尽量采用标准模式的 MPI_Send，除非知道自己在干什么；
  - 接收函数：如果有必要，采用非阻塞版本 MPI_Irecv，并尽早发起。

点对点通信的底层协议 (protocol)

- 急迫 (eager) 协议：发送方就绪就可以发送数据。

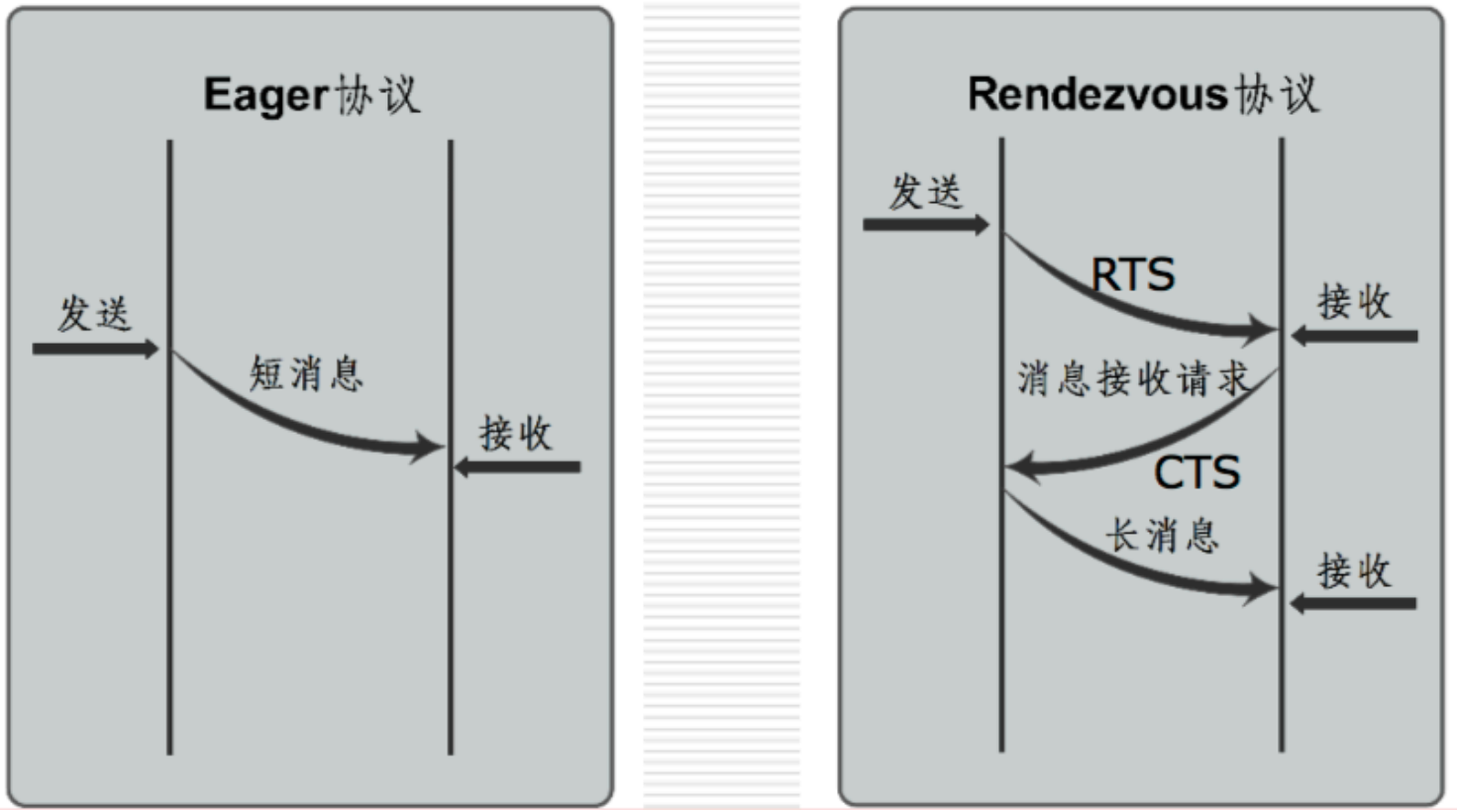


- 汇合 (rendezvous) 协议：双方均就绪才发送数据。



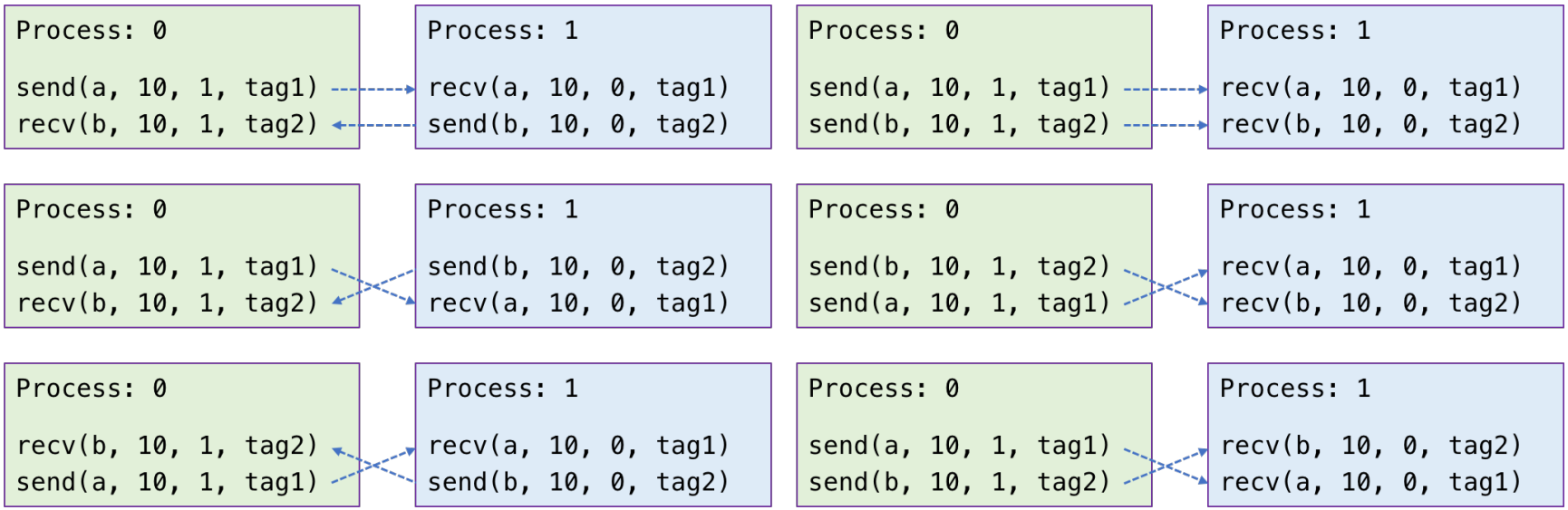
两种协议的比较

- 急迫协议：引入底层缓冲开销，减少同步开销，适合短消息传输。
- 汇合协议：可以避免缓冲，引入同步开销，适合长消息传输。
- MPI 系统自动选择，用户可调节切换策略 (如 EAGER_LIMIT 等)。



练习

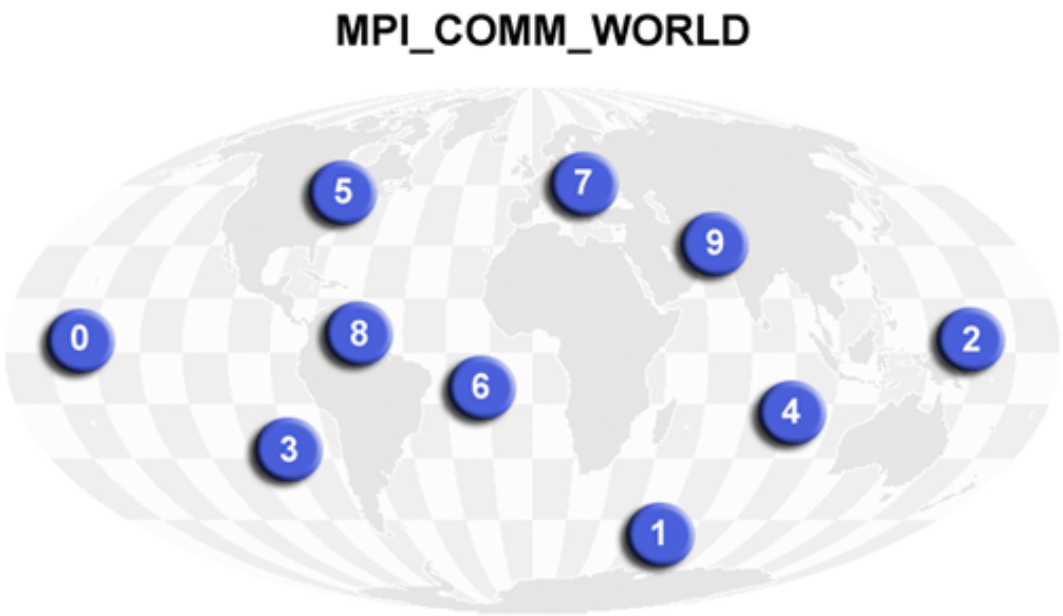
- 从下图中选择一种感兴趣的通信场景，测试不同的 MPI 发送/接收函数（可以利用 bsend 程序进行修改）：



MPI 通信器与进程组

通信器的再思考

- 定义了所有参与通信的进程的集合；
- 几乎所有的 MPI 函数都需要指定该函数所作用的通信器；
- 通信器变量的数据类型是 `MPI_Comm`；
- 默认通信器是 `MPI_COMM_WORLD` (所有进程)。



如果我们只打算对通信器中的部分进程进行集合通信怎么办?

通信器的分裂 (split)

- 基于通信器，分裂出新的子通信器：

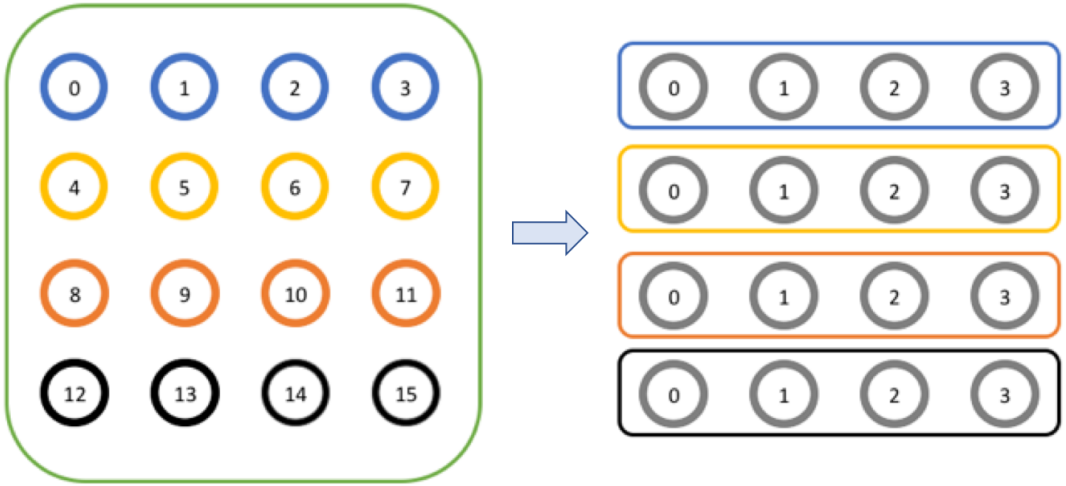
```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm*
    newcomm)
```

  - comm：原始通信器，没有消失；
  - color：决定该进程属于哪个子通信器 (MPI_UNDEFINED 排除)；
  - key：决定该进程在子通信器中的 rank (从小到大)；
  - newcomm：分裂出的子通信器。

- 释放不使用的通信器：

```
int MPI_Comm_free(MPI_Comm * comm)
```

示例程序：分裂 (1)



mpi_split.c

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char **argv){
5    int rank, size, color, sub_rank, sub_size;
6    MPI_Comm sub_comm;
7
8    MPI_Init(&argc, &argv);
```

示例程序：分裂 (2)

```
 9    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10    MPI_Comm_size(MPI_COMM_WORLD, &size);
11
12    color = rank/4;
13
14    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &sub_comm);
15
16    MPI_Comm_rank(sub_comm, &sub_rank);
17    MPI_Comm_size(sub_comm, &sub_size);
18
19    printf("World rank/size: %d/%d \t Sub rank/size: %d/%d\n",
        rank, size, sub_rank, sub_size);
20
21    MPI_Comm_free(&sub_comm);
22    MPI_Finalize();
23    return 0;
24  }
```

示例程序：分裂 (3)

- 运行 (请正确使用 sbatch 或者 salloc)：

```
 $ mpiexec -n 16 ./split
World rank/size: 8/16 -- Sub rank/size: 0/4
World rank/size: 9/16 -- Sub rank/size: 1/4
World rank/size: 12/16 -- Sub rank/size: 0/4
World rank/size: 14/16 -- Sub rank/size: 2/4
World rank/size: 13/16 -- Sub rank/size: 1/4
World rank/size: 2/16 -- Sub rank/size: 2/4
World rank/size: 3/16 -- Sub rank/size: 3/4
World rank/size: 4/16 -- Sub rank/size: 0/4
World rank/size: 5/16 -- Sub rank/size: 1/4
World rank/size: 10/16 -- Sub rank/size: 2/4
World rank/size: 0/16 -- Sub rank/size: 0/4
World rank/size: 11/16 -- Sub rank/size: 3/4
World rank/size: 15/16 -- Sub rank/size: 3/4
World rank/size: 1/16 -- Sub rank/size: 1/4
World rank/size: 6/16 -- Sub rank/size: 2/4
World rank/size: 7/16 -- Sub rank/size: 3/4
```

进程组 (group) (1)

- 每个通信器包含一个唯一的 ID (MPI 内部管理) 以及对应一个进程组 (group)，采用下面方式获取通信器的进程组：

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group* group)
```

- 进程组包含了该通信器的进程信息，因此可以获取 rank 和 size：

```
int MPI_Group_rank(MPI_Group group, int* rank)
int MPI_Group_size(MPI_Group group, int* size)
```

进程组 (group) (2)

- 进程组不可以用于通信，但是可以用于创建新的进程组
  - 两个进程组的并：

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
    MPI_Group* newgroup)
```

  - 两个进程组的交：

```
int MPI_Group_intersection(MPI_Group group1, MPI_Group
    group2, MPI_Group* newgroup)
```

  - 指定 n 个进程：

```
int MPI_Group_incl(MPI_Group group, int n, const int ranks
    [], MPI_Group* newgroup)
```

- 因此，同一个进程可以属于不同的进程组/通信器。

进程组 (group) (3)

- 基于进程组 (使用全局通信/局部通信)，可以创建通信器:

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm*
    newcomm)
int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int
    tag, MPI_Comm* newcomm)
```

示例程序：进程组 (1)

mpi_group.c

```
1  #include <mpi.h>
2  #include <stdio.h>
3  int main(int argc, char **argv){
4    int        rank, size, sub_rank, sub_size;
5    MPI_Group group, sub_group;
6    MPI_Comm  sub_comm;
7    const int ranks[4] = {2, 3, 5, 7};
8
9    MPI_Init(&argc, &argv);
10   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11   MPI_Comm_size(MPI_COMM_WORLD, &size);
12
13   MPI_Comm_group(MPI_COMM_WORLD, &group);
14   MPI_Group_incl(group, 4, ranks, &sub_group);
15
16   MPI_Comm_create(MPI_COMM_WORLD, sub_group, &sub_comm);
17
18   if (sub_comm != MPI_COMM_NULL) {
```

示例程序：进程组 (2)

```
19      MPI_Comm_rank(sub_comm, &sub_rank);
20      MPI_Comm_size(sub_comm, &sub_size);
21    } else {
22      sub_rank = -1;
23      sub_size = -1;
24    }
25
26    printf("World rank/size: %d/%d --- Sub rank/size: %d/%d\n",
        rank, size, sub_rank, sub_size);
27
28    MPI_Group_free(&group);
29    MPI_Group_free(&sub_group);
30
31    if (sub_comm != MPI_COMM_NULL) {
32      MPI_Comm_free(&sub_comm);
33    }
34    MPI_Finalize();
35    return 0;
36  }
```

示例程序：进程组 (3)

- 运行 (请正确使用 sbatch 或者 salloc)：

```
 $ mpiexec -n 8 ./group
World rank/size: 0/8 --- Sub rank/size: -1/-1
World rank/size: 1/8 --- Sub rank/size: -1/-1
World rank/size: 2/8 --- Sub rank/size: 0/4
World rank/size: 4/8 --- Sub rank/size: -1/-1
World rank/size: 5/8 --- Sub rank/size: 2/4
World rank/size: 3/8 --- Sub rank/size: 1/4
World rank/size: 6/8 --- Sub rank/size: -1/-1
World rank/size: 7/8 --- Sub rank/size: 3/4
```

MPI 补遗与新特性

MPI-1 的一些其他特性

- 扩展数据类型：
  - 除了多种预定义的数据类型，MPI 允许用户自定义数据类型；
  - 自定义数据类型的数据在内存中可以是连续存储的，也可以是不连续的。
- 虚拟拓扑：
  - 支持通信器/进程组中的进程按照某种拓扑方式排列；
  - 主要有笛卡尔 (Cartesian) 和图 (Graph) 两种；
  - 可以反映底层网络物理连接，也可以纯粹为了编程方便。

MPI-2 的一些重要新特性

- 动态进程：进程数可以动态改变；
- 单边通信：又称远程内存访问 (Remote Memory Access, RMA)；
- 增强的集合通信：允许跨通信器进行集合通信；
- 外部接口：允许用户在上层对 MPI 函数进行封装；
- 并行 I/O：支持并行文件的输入和输出。

MPI-3 的一些重要新特性

- 非阻塞集合通信：支持非阻塞形式执行集合通信操作，可实现计算通信重叠；
- 新的单边通信：可以更好地处理不同类型的内存模型；
- 邻居集合通信：定义在进程拓扑的基础上，实现邻居进程间的集合通信；
- 内部接口：允许用户通过 MPIT 工具接口 MPI 内部的变量信息。

MPI 标准的最新进展：

`http://www.mpi-forum.org/docs/`