

并行与分布式计算基础 II: MPI 编程与实践

杨超

chao_yang@pku.edu.cn

2023 秋



内容提纲

- 1 入门知识
- 2 一个简单的例子
- 3 点对点通信
- 4 死锁与非阻塞通信
- 5 点对点通信实现机制
- 6 集合通信
- 7 辅助函数
- 8 通信器与进程组
- 9 补遗与新特性

内容提纲

- 1 入门知识
- 2 一个简单的例子
- 3 点对点通信
- 4 死锁与非阻塞通信
- 5 点对点通信实现机制
- 6 集合通信
- 7 辅助函数
- 8 通信器与进程组
- 9 补遗与新特性

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。

- 定义了若干核心库函数的语法和涵义；
- 独立于编程语言，支持 C/C++、Fortran 语言的绑定；
- 独立于平台，学术界和厂商发展了若干高效、可靠的实现版；
- 支撑和推动了高性能计算软硬件生态的发展。

为什么用 MPI ?

- 目前几乎所有主流并行计算机上都提供了 MPI 的支持；
- MPI 没有依托于任何标准化组织，但已经成为事实上的工业标准；
- 除了 MPI，没有其他更好的选择！

MPI 的发展历史

- 1991 年夏季：首次讨论 MPI 的概念；
- 1992 年 4 月：首届 MPI 研讨会，MPI 标准工作组成立；
- 1992 年 11 月：MPI 1.0 初稿形成，MPI 论坛成立；
- 1993 年 1-9 月：MPI 标准工作组会议每六周开展一次；
- 1993 年 11 月：MPI 1.0 初稿在 SC93 大会上公布，征集公众意见；
- 1994 年 6 月：MPI 1.0 发布。

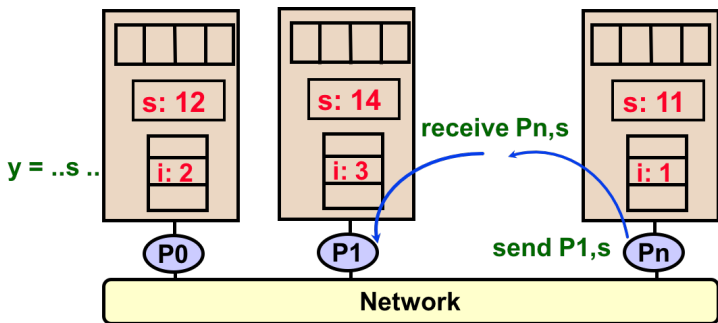
版本	最新版	增加主要功能	增加语言绑定
MPI-1	MPI 1.3	消息传递、静态运行空间	C/F77
MPI-2	MPI 2.2	并行 IO、动态进程等	C++/F90
MPI-3	MPI 3.1	容错、RMA 等 (主流版本)	F2008
MPI-4	MPI 4.0	混合编程等 (进行中)	-

MPI 的主要实现版本

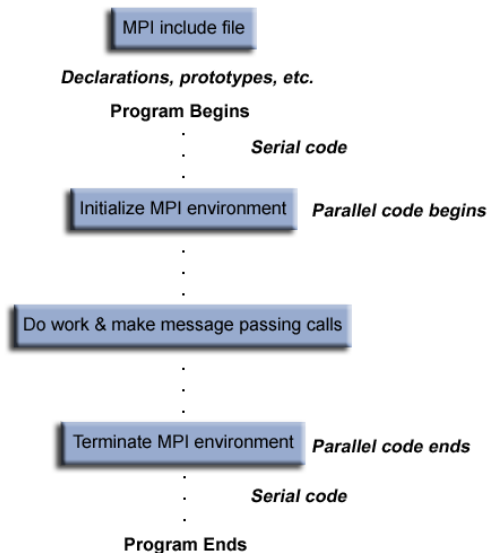
- MPICH
 - ▶ 由 Argonne 国家实验室与 Mississippi 州立大学联合开发；
 - ▶ 是最早的，也是目前最流行的 MPI 实现。
- MVAPICH
 - ▶ Ohio 州立大学开发；
 - ▶ 基于 MPICH 发展，强调对各类硬件和网络的个性化支持。
- OpenMPI
 - ▶ Stuttgart 大学等开发；
 - ▶ 是多个 MPI 开源实现的合并版。
- 商业版
 - ▶ Intel MPI、IBM MPI、Cray MPI、HP-MPI、MS-MPI、… …
- ABI (application binary interface) 兼容：由 MPICH 发起，保证各个 MPI 实现在底层数据类型上的兼容性。

MPI 的主要理念

- 机器由若干可以相互传递消息的进程 (process) 构成;
- 每个进程拥有私有的存储空间, 进程间无共享存储;
- 进程间的消息传递采用显式的发送/接收 (send/receive) 机制完成;
- 程序的运行模式主要有松散同步和完全异步两种;
- 程序往往采用 SPMD (single program multiple data) 方式编写。



MPI 程序的执行流程



MPI 程序的结构

- 头文件 (header file): 包含了 MPI 库提供的函数接口定义

C include file	Fortran include file
<code>#include "mpi.h"</code>	<code>include 'mpif.h'</code>

- 函数接口的调用格式
 - MPI 中函数、变量、参数均以 `MPI_` 作为前缀

C Binding	
Format:	<code>rc = MPI_Xxxxx(parameter, ...)</code>
Example:	<code>rc = MPI_Bsend(&buf, count, type, dest, tag, comm)</code>
Error code:	Returned as "rc". MPI_SUCCESS if successful
Fortran Binding	
Format:	<code>CALL MPI_XXXXX(parameter,..., ierr)</code> <code>call mpi_xxxxx(parameter,..., ierr)</code>
Example:	<code>CALL MPI_BSEND(buf, count, type, dest, tag, comm, ierr)</code>
Error code:	Returned as "ierr" parameter. MPI_SUCCESS if successful

内容提纲

- 1 入门知识
- 2 一个简单的例子
- 3 点对点通信
- 4 死锁与非阻塞通信
- 5 点对点通信实现机制
- 6 集合通信
- 7 辅助函数
- 8 通信器与进程组
- 9 补遗与新特性

MPI 定义的函数

- MPI-3 有超过 430 个函数 (包含 MPI-1、MPI-2 大部分函数);
- 从应用角度 (包括功能和性能) 上看, 只有 20 多个函数是常用的;
- 单纯从功能上说, 只需要 6 个函数就可以完成所有 MPI 功能。

The minimal set of MPI routines.

<code>MPI_Init</code>	Initializes MPI.
<code>MPI_Finalize</code>	Terminates MPI.
<code>MPI_Comm_size</code>	Determines the number of processes.
<code>MPI_Comm_rank</code>	Determines the label of the calling process.
<code>MPI_Send</code>	Sends a message.
<code>MPI_Recv</code>	Receives a message.

MPI 库的启动和结束

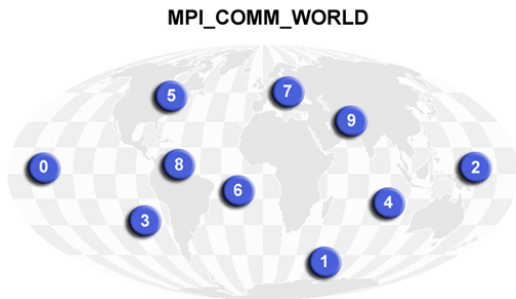
- MPI_Init 一般在主程序的开头，主要用于启动 MPI 环境；
- MPI_Finalize 一般在主程序末尾，主要用于终止 MPI 环境；
- 这两个函数的语法如下：

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

- 如果运行正常，所有 MPI 函数的返回值都是 MPI_SUCCESS 常量。

我在哪？——通信器 (communicator)

- 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)。



我是谁？——获取进程信息

- `MPI_Comm_size` 函数用来获得目标通信器的总进程数；
- `MPI_Comm_rank` 函数用来获得当前进程在目标通信器的进程号；
- 这两个函数的语法如下：

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

- `rank` 的范围： $0, 1, \dots, \text{size} - 1$.
- 同一个进程可以属于不同的通信器，因此 `rank` 也可能不同！

第一个 MPI 程序: hello world!

mpi_hello.c

```
1  #include <mpi.h> // mpi header file
2  #include <stdio.h> // standard I/O
3
4  int main(int argc, char *argv){
5      int size, rank;
6
7      MPI_Init(&argc, &argv); // initialize MPI
8      MPI_Comm_size(MPI_COMM_WORLD, &size); // get num of procs
9      MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get my rank
10     printf("From process %d of %d: Hello World!\n", rank, size);
11     if (rank == 0) printf("That's all, folks!\n");
12     MPI_Finalize(); // done with MPI
13     return 0;
14 }
```

程序的编译与运行

- 加载 MPI 库环境:

```
$ module add mpich
```

- 编译:

```
$ mpicc mpi_hello.c -o hello
```

- 运行 (请正确使用 sbatch 或者 salloc):

```
$ mpiexec -n 4 ./hello
```


运行结果

- 第一次运行:

```
From process 3 of 4: Hello World!  
From process 1 of 4: Hello World!  
From process 2 of 4: Hello World!  
From process 0 of 4: Hello World!  
That's all, folks!
```

- 第二次运行:

```
From process 2 of 4: Hello World!  
From process 1 of 4: Hello World!  
From process 0 of 4: Hello World!  
That's all, folks!  
From process 3 of 4: Hello World!
```

- 为什么?

内容提纲

- 1 入门知识
- 2 一个简单的例子
- 3 点对点通信**
- 4 死锁与非阻塞通信
- 5 点对点通信实现机制
- 6 集合通信
- 7 辅助函数
- 8 通信器与进程组
- 9 补遗与新特性

基本消息传递——发送与接收

- MPI_Send 函数用来发送消息；
- MPI_Recv 函数用来接收消息；
- 这两个函数语法如下：

```
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)
```

- tag 表示消息标签 (非负整数), 最大不超过常量 MPI_TAG_UB；
- buf 表示发送和接收的消息所对应的本地内存位置；
- count 表示发送和接收的消息长度；
- datatype 表示发送和接收的消息的数据类型；
- dest 和 source 分别表示接收端和发送端的进程号。

思考：接收端和发送端所设定的消息长度可以不一样吗？

MPI 数据类型

- MPI_Datatype 定义了若干常用数据类型：

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

- MPI_BYTE 是字节类型 (8 bits)；
- MPI_PACKED 是多个数据的聚合类型，用于不连续数据的打包。

MPI 通信状态

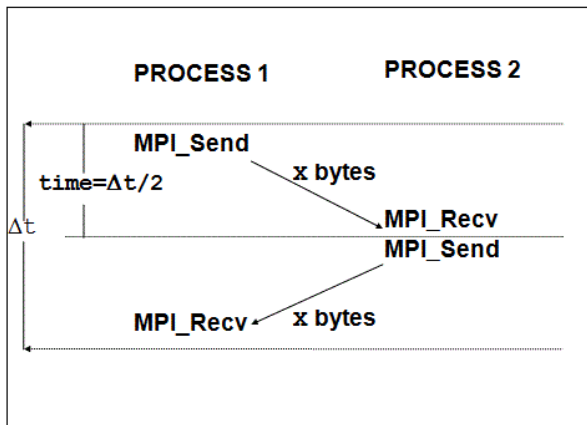
- 接收函数比发送函数多一个参数：status；
- 它保存了函数返回的通信状态，可置为 MPI_STATUS_IGNORE；
- 其类型 MPI_Status 是一个 MPI 预定义的结构体：

```
typedef struct MPI_Status {  
    int MPI_SOURCE;  
    int MPI_TAG;  
    int MPI_ERROR;  
    ...  
};
```

- 可以使用 MPI_Get_count 函数得到接收消息的大小：

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,  
                 int *count)
```

示例程序：乒乓通信



mpi_pingpong.c

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[]){
5      int token, size, rank;
6
7      MPI_Init(&argc, &argv);
8      MPI_Comm_size(MPI_COMM_WORLD, &size);
9      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10
11     if (rank == 0) {
12         token = -1;
13         MPI_Send(&token, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD);
14         printf("Process %d pinged token %d to process %d\n", rank,
15             token, 1-rank);
16         MPI_Recv(&token, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD,
17             MPI_STATUS_IGNORE);
18         printf("Process %d ponged token %d from process %d\n", rank
19             , token, 1-rank);
20     } else if (rank == 1) {
```

```
18     MPI_Recv(&token, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD,  
19             MPI_STATUS_IGNORE);  
20     MPI_Send(&token, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD);  
21 }  
22 MPI_Finalize();  
23 return 0;  
24 }
```


运行结果

- 运行 (请正确使用 sbatch 或者 salloc):

```
$ mpiexec -n 2 ./pingpong
```

- 运行结果:

```
Process 0 pinged token -1 to process 1  
Process 0 ponged token -1 from process 1
```

小练习

● 尝试利用 MPI_Status 检查通信状态

```
1  MPI_Status sta;
2  ...
3  if (rank == 0) {
4      token = -1;
5      MPI_Send(&token, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD);
6      printf("Process %d pinged token %d to process %d\n", rank
7          , token, 1-rank);
8      MPI_Recv(&token, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD, &
9          sta);
10     printf("Process %d ponged token %d from process %d\n", rank
11         , token, sta.MPI_SOURCE);
12 } else if (rank == 1) {
13     ...
14 }
```

示例程序：交换通信

mpi_exchange.c

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[]){
5      int a, b, size, rank;
6
7      MPI_Init(&argc, &argv);
8      MPI_Comm_size(MPI_COMM_WORLD, &size);
9      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10
11     if (rank == 0) {
12         a = -1;
13         MPI_Send(&a, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD);
14         printf("Process %d sent token %d to process %d\n", rank, a,
15             1-rank);
16         MPI_Recv(&b, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD,
17             MPI_STATUS_IGNORE);
```

```
16     printf("Process %d received token %d from process %d\n",  
17           rank, b, 1-rank);  
18 } else if (rank == 1) {  
19     a = 1;  
20     MPI_Recv(&b, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD,  
21             MPI_STATUS_IGNORE);  
22     MPI_Send(&a, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD);  
23 }  
24 MPI_Finalize();  
25 return 0;  
26 }
```

运行结果

- 运行 (请正确使用 sbatch 或者 salloc):

```
$ mpiexec -n 2 ./exchange
```

- 运行结果:

```
Process 0 sent token -1 to process 1  
Process 0 received token 1 from process 1
```

小练习

- 这部分代码：

```
1  } else if (rank == 1) {  
2      a = 1;  
3      MPI_Recv(&b, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD,  
4              MPI_STATUS_IGNORE);  
5      MPI_Send(&a, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD);  
6  }
```

- 是否可以修改为：

```
1  } else if (rank == 1) {  
2      b = 1;  
3      MPI_Recv(&a, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD,  
4              MPI_STATUS_IGNORE);  
5      MPI_Send(&b, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD);  
6  }
```

MPI 数据交换

- 如果两个进程需要进行数据交换，可以使用发送接收组合操作：

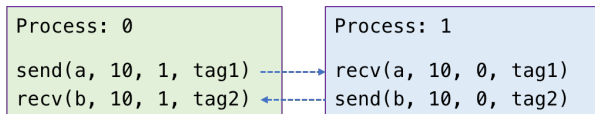
```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype  
    senddatatype, int dest, int sendtag, void *recvbuf, int  
    recvcount, MPI_Datatype recvdatatype, int source, int recvtag,  
    MPI_Comm comm, MPI_Status *status)
```

- 如果交换的数据共用一个 buffer，则可借助于如下操作：

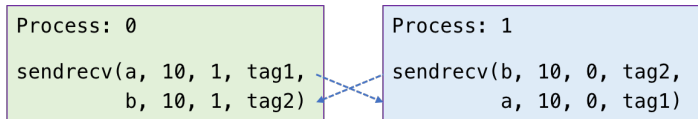
```
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype  
    datatype, int dest, int sendtag, int source, int recvtag,  
    MPI_Comm comm, MPI_Status *status)
```

MPI 数据交换示例

- 考虑下面的通信行为：



- 可以使用数据交换函数一次性完成：



小练习

- 尝试利用 `MPI_Sendrecv` 完成数据交换：

mpi_exchange2.c

```
1  ...
2  if (rank == 0) {
3      a = -1;
4      MPI_Sendrecv(&a, 1, MPI_INT, 1-rank, 0, &b, 1, MPI_INT
5      , 1-rank, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
6      printf("Process %d exchanged token %d to token %d with
7      process %d\n", rank, a, b, 1-rank);
8  } else if (rank == 1) {
9      a = 1;
10     MPI_Sendrecv(&a, 1, MPI_INT, 1-rank, 0, &b, 1, MPI_INT
11     , 1-rank, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
12 }
13 ...
```

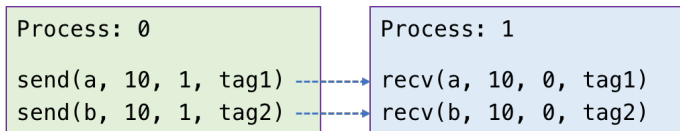
- 尝试利用 `MPI_Sendrecv_replace` 完成数据交换。

内容提纲

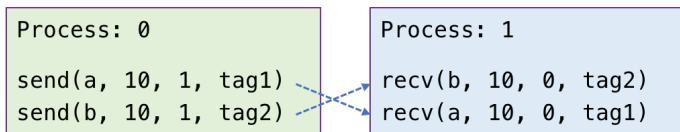
- 1 入门知识
- 2 一个简单的例子
- 3 点对点通信
- 4 死锁与非阻塞通信**
- 5 点对点通信实现机制
- 6 集合通信
- 7 辅助函数
- 8 通信器与进程组
- 9 补遗与新特性

死锁 (deadlock)

- 考虑下面的通信行为：



- 如果改变消息的接收顺序，会怎样？



- 产生了死锁 (deadlock)!
- 怎么避免？(1) 程序员把控；(2) 非阻塞 (non-blocking) 通信。

阻塞 (blocking) vs 非阻塞 (non-blocking) 通信

- 阻塞 (blocking) 通信：不成功不返回 (通信过程中该进程暂停)。

```
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)
```

- 非阻塞 (non-blocking) 通信：交上去不管了 (后台执行通信)。

```
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（立即返回，flag 值为 0 表示未结束）：

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

- 等待非阻塞通信结束（通信结束后函数返回）：

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

如果非阻塞通信没有结束，要小心使用 buf 中的数据！

- 不要修改 buf 中的发送数据；
- 不要使用 buf 中的接收数据。

非阻塞通信状态的批量检测与控制

- 检测/等待多个非阻塞通信：

```
int MPI_Testall(int count, MPI_Request requests[], int *flag,  
                MPI_Status statuses[])  
int MPI_Waitall(int count, MPI_Request requests[], MPI_Status  
                statuses[])
```

- 检测/等待任一个非阻塞通信：

```
int MPI_Testany(int count, MPI_Request requests[], int *index, int *  
                flag, MPI_Status *status)  
int MPI_Waitany(int count, MPI_Request requests[], int *index,  
                MPI_Status *status)
```

- 检测/等待任一些非阻塞通信：MPI_Testsome, MPI_Waitsome (略).

使用非阻塞通信避免死锁


- 使用非阻塞发送：

Process: 0

```
isend(a, 10, 1, tag1, &req1)
isend(b, 10, 1, tag2, &req2)
...
wait(&req1, &sta1)
wait(&req2, &sta2)
```

Process: 1

```
recv(b, 10, 0, tag2)
recv(a, 10, 0, tag1)
```




- 或者，使用非阻塞接收：

Process: 0

```
send(a, 10, 1, tag1)
send(b, 10, 1, tag2)
```

Process: 1

```
irecv(b, 10, 0, tag2, &req1)
irecv(a, 10, 0, tag1, &req2)
...
wait(&req1, &sta1)
wait(&req2, &sta2)
```



- 思考：wait 的顺序有影响吗？tag 可以去掉吗？

MPI 墙钟时间

- 返回当前进程的时钟时间:

```
double MPI_Wtime()
```

- 用法:

```
1  ...  
2  t0 = MPI_Wtime();  
3  ... // do some works  
4  t1 = MPI_Wtime();  
5  ...
```

- 返回 MPI_Wtime 的时钟刻度:

```
double MPI_Wtick()
```

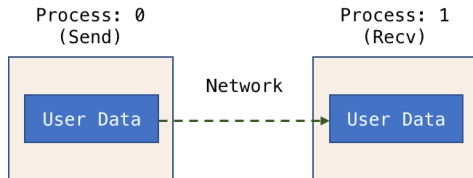


内容提纲

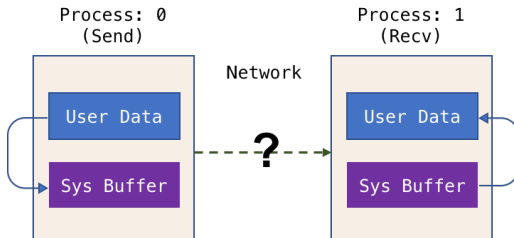
- 1 入门知识
- 2 一个简单的例子
- 3 点对点通信
- 4 死锁与非阻塞通信
- 5 点对点通信实现机制**
- 6 集合通信
- 7 辅助函数
- 8 通信器与进程组
- 9 补遗与新特性

MPI 系统缓冲 (system buffer)

- MPI Send/Recv 的接口：

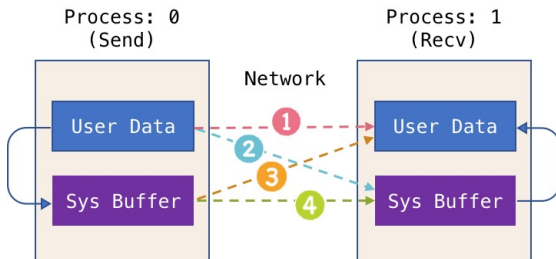


- 借助 MPI 系统缓冲, Send/Recv 的底层实现有几种方式？



基于 MPI 系统缓冲的 Send/Recv 的底层实现

- 借助 MPI 系统缓冲，Send/Recv 的底层实现包括：



- MPI 底层会根据具体实现、消息大小等选择合适的方式。

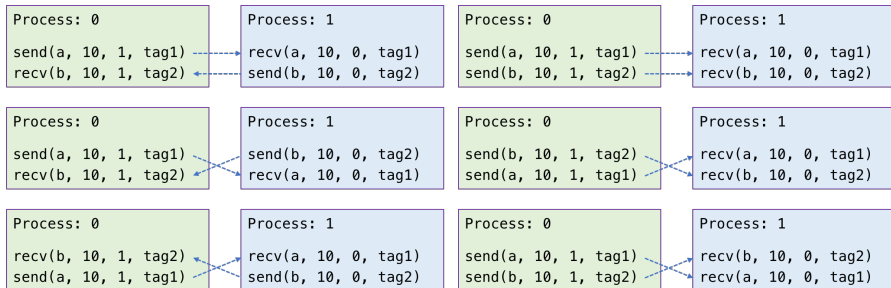
阻塞通信的再思考

- 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)

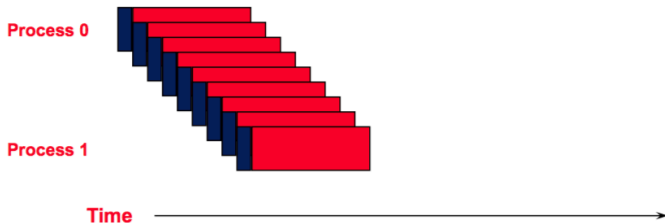
- 同步 (synchronous) 模式: `MPI_Ssend`
 - ▶ 无论接收端是否启动接收, 发送端可在任意时间启动发送;
 - ▶ 只有在接收端启动接收后, 发送端才返回;
 - ▶ 发送端返回不仅表示缓冲区可以使用, 还表示接收端已经到达了某个程序点 (进行了握手同步)。
- 就绪 (ready) 模式: `MPI_Rsend`
 - ▶ 仅当对方的接收操作启动且准备就绪, 才发送数据 (否则报错);
 - ▶ 语义上和同步发送完全一致, 避免了额外的缓冲区操作和发送接收方的握手操作。
- 缓冲 (buffered) 模式: `MPI_Bsend`
 - ▶ 发送端把数据拷贝到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

发送函数的分类 (2)

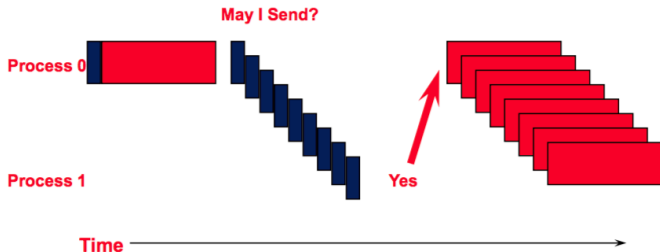
- 标准 (standard) 模式: `MPI_Send`
 - ▶ 可以是同步的或缓冲的, 给予系统以灵活选择的机会;
 - ★ 对短消息, 一般采用缓冲模式;
 - ★ 对长消息, 一般采用同步模式, 不同在于数据传输完成才返回;
 - ★ 长短消息的切换点, 可以配置。
- 非阻塞发送
 - ▶ 上述发送也有对应的非阻塞版本, 但是极少使用。
- 一些建议
 - ▶ 发送函数: 尽量采用标准模式的 `MPI_Send`, 除非知道自己在干什么;
 - ▶ 接收函数: 如果有必要, 采用非阻塞版本 `MPI_Irecv`, 并尽早发起。

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。

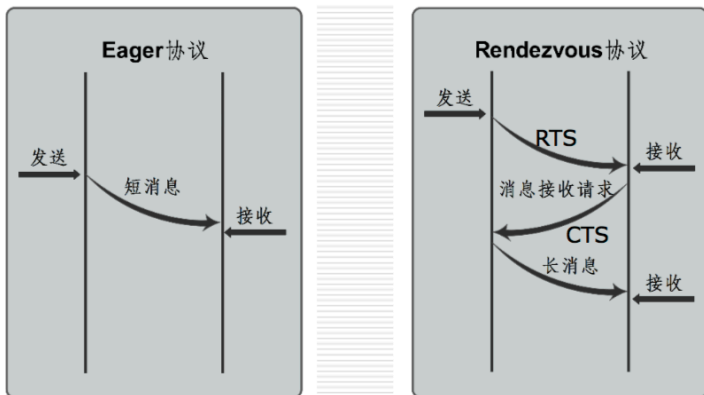


- 汇合 (rendezvous) 协议：双方均就绪才发送数据。



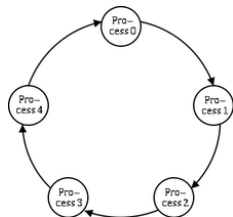
两种协议的比较

- 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：击鼓传花

- 支持任意 MPI 进程数
- 尝试使用不同类型的点对点通信
- 可增加对延迟的测试等内容



• 测试结果

```
$ mpiexec -n 8 ./mpi_ring
Process 1 received token -1 from process 0
Process 2 received token -1 from process 1
Process 3 received token -1 from process 2
Process 4 received token -1 from process 3
Process 5 received token -1 from process 4
Process 6 received token -1 from process 5
Process 7 received token -1 from process 6
Process 0 received token -1 from process 7
```

内容提纲

- 1 入门知识
- 2 一个简单的例子
- 3 点对点通信
- 4 死锁与非阻塞通信
- 5 点对点通信实现机制
- 6 集合通信**
- 7 辅助函数
- 8 通信器与进程组
- 9 补遗与新特性

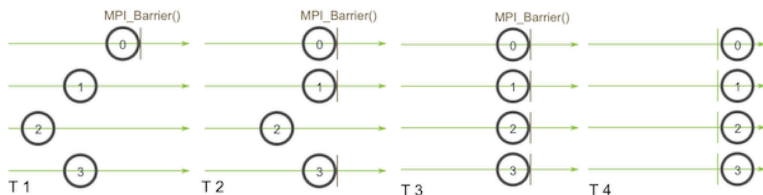
MPI 集合通信 (collective communication) 概述

- MPI 集合通信是指涉及到通信器中所有进程的通信，有三类：
 - ▶ 同步 (synchronization): barrier 等；
 - ▶ 数据移动: broadcast, scatter/gather, all to all 等；
 - ▶ 规约 (reduction): reduce, all reduce, reduce scatter 等。
- MPI 集合通信的一些特点：
 - ▶ 通信器中所有进程必须同时调用该操作；
 - ▶ 不需要指定 tag；
 - ▶ 所有的集合通信都是某种意义的同步操作。

MPI 栅栏同步

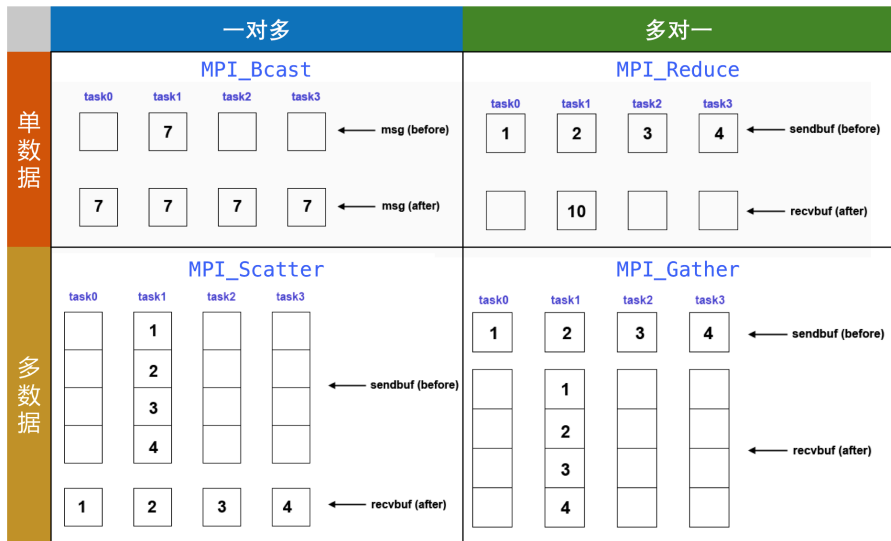
- 通信器中所有进程相互等待至某个同步点：

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



- 思考：这种栅栏同步底层是怎么实现的呢？

MPI “一对多”与“多对一”通信概览



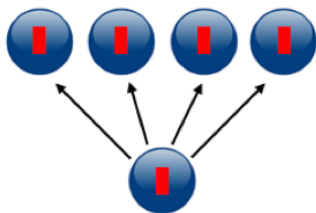
MPI 广播与规约

- 广播 (broadcast):

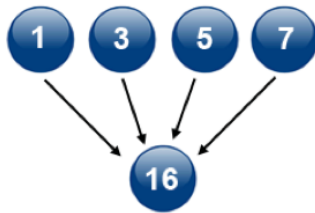
```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int  
source, MPI_Comm comm)
```

- 规约 (reduce):

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



broadcast



reduction

规约操作的类型

- MPI_Op 是 MPI 自定义操作，主要有：

Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating points
MPI_MIN	Minimum	C integers and floating points
MPI_SUM	Sums the elements	C integers and floating points
MPI_PROD	Multiplies the elements	C integers and floating points
MPI LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and bytes
MPI_LOR	Logical OR	C integers
MPI BOR	Bit-wise OR	C integers and bytes
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and bytes
MPI_MAXLOC	Max value and the rank	Data-pairs
MPI_MINLOC	Min value and the rank	Data-pairs

示例程序：广播与规约

mpi_bcast_reduce.c

```
1  ...
2  #define ROOT    0 // Rank of the root process
3
4  int main(int argc, char **argv){
5      ...
6      /* Set the data on root process */
7      if (rank == ROOT) {
8          data = 100;
9          printf("On root proc %d, data to be broadcast: %d\n", rank,
10               data);
11      }
12
13      /* Broadcast the data to everybody */
14      MPI_Bcast(&data, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
15
16      /* Everybody shows the broadcast data */
17      printf("On proc %d, data after broadcasting = %d\n", rank,
18           data);
```



```
17 MPI_Barrier(MPI_COMM_WORLD);
18
19 /* Modify the data to reduce */
20 data = data + rank;
21 printf("On proc %d, data to be reduced = %d\n", rank, data);
22
23 /* Reduce everybody's data to root */
24 MPI_Reduce(&data, &data_reduce, 1, MPI_INT, MPI_SUM, ROOT,
           MPI_COMM_WORLD);
25
26 /* On root, show the reduced data */
27 if (rank == ROOT) {
28     printf("On root proc %d, data reduced: %d\n", rank,
           data_reduce);
29 }
30 ...
```

- 运行 (请正确使用 sbatch 或者 salloc):

```
$ mpiexec -n 4 ./bcast_reduce

On root proc 0, data to be broadcast: 100
On proc 0, data after broadcasting = 100
On proc 2, data after broadcasting = 100
On proc 3, data after broadcasting = 100
On proc 1, data after broadcasting = 100
On proc 0, data to be reduced = 100
On proc 2, data to be reduced = 102
On proc 1, data to be reduced = 101
On proc 3, data to be reduced = 103
On root proc 0, data reduced: 406
```

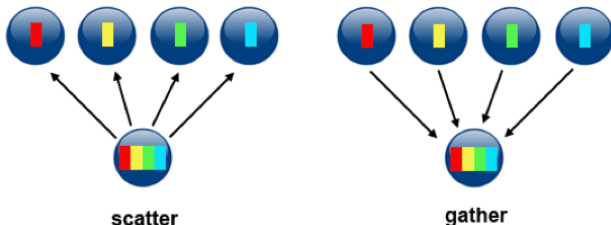
MPI 分发与集中

- 分发 (scatter):

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

- 集中 (gather):

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



示例程序：分发与集中

mpi_scatter_gather.c

```
1  ...
2  #define N      2 // Number of data per process
3  #define ROOT   0 // Rank of the root process
4
5  int main(int argc, char **argv){
6      ...
7      /* Set the data on root process */
8      if (rank == ROOT) {
9          printf("On root proc %d, data to be scattered:\n", rank);
10         for ( j=0; j<size; j++) {
11             for ( i=0; i<N; i++ ) {
12                 data_root[j][i] = j+i;
13                 printf("(%d, %d) = %d  ", j, i, data_root[j][i]);
14             }
15             printf("\n");
16         }
17     }
18 }
```

```
19  /* Scatter the data to everybody */
20  MPI_Scatter(&data_root[0][0], N, MPI_INT, &data[0], N,
    MPI_INT, ROOT, MPI_COMM_WORLD);
21
22  /* Everybody shows the scattered data */
23  for (i=0; i<N; i++) {
24      printf("On proc %d, scattered data[%d] = %d\n", rank, i,
    data[i]);
25  }
26  MPI_Barrier(MPI_COMM_WORLD);
27
28  /* Gather everybody's data to root */
29  MPI_Gather(&data[0], N, MPI_INT, &data_root[0][0], N, MPI_INT
    , ROOT, MPI_COMM_WORLD);
30
31  /* On root, show the gathered data */
32  if (rank == ROOT) {
33      printf("On root proc %d, data gathered:\n", rank);
34      for ( j=0; j<size; j++) {
35          for ( i=0; i<N; i++ ) {
36              printf("(%d, %d) = %d  ", j, i, data_root[j][i]);
37          }
```

```
38     printf("\n");
39 }
40 }
41 ...
```

- 运行 (请正确使用 sbatch 或者 salloc):

```
$ mpiexec -n 4 ./scatter_gather

Data to be scattered on root proc 0:
(0, 0) = 0   (0, 1) = 1
(1, 0) = 1   (1, 1) = 2
(2, 0) = 2   (2, 1) = 3
(3, 0) = 3   (3, 1) = 4
On proc 0, scattered data[0] = 0
On proc 0, scattered data[1] = 1
On proc 2, scattered data[0] = 2
On proc 2, scattered data[1] = 3
On proc 1, scattered data[0] = 1
On proc 3, scattered data[0] = 3
On proc 3, scattered data[1] = 4
On proc 1, scattered data[1] = 2
```

Data gathered on root proc 0:

$(0, 0) = 0$ $(0, 1) = 1$

$(1, 0) = 1$ $(1, 1) = 2$

$(2, 0) = 2$ $(2, 1) = 3$

$(3, 0) = 3$ $(3, 1) = 4$

MPI “多对多” 通信 (1)

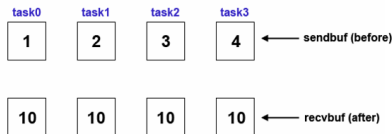
- 全规约 (allreduce):

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

MPI_Allreduce

Perform reduction and store result across all tasks in communicator

```
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT,
    MPI_SUM, MPI_COMM_WORLD);
```



MPI “多对多” 通信 (2)

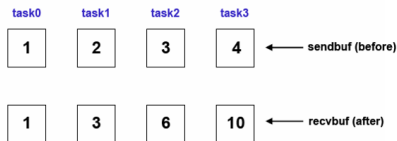
- 前缀和 (scan, prefix-sum):

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

MPI_Scan

Computes the scan (partial reductions) across all tasks in communicator

```
count = 1;  
MPI_Scan(sendbuf, recvbuf, count, MPI_INT,  
MPI_SUM, MPI_COMM_WORLD);
```



MPI “多对多” 通信 (3)

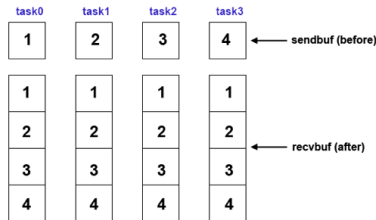
- 全集中 (allgather):

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

MPI_Allgather

Gathers data from all tasks and then distributes to all tasks in communicator

```
sendcnt = 1;  
recvcnt = 1;  
MPI_Allgather(sendbuf, sendcnt, MPI_INT  
    recvbuf, recvcnt, MPI_INT  
    MPI_COMM_WORLD);
```



MPI “多对多” 通信 (4)

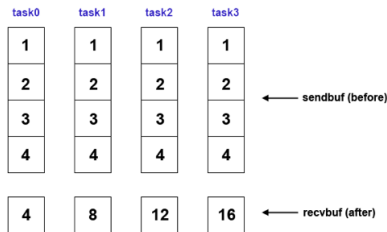
- 规约分发 (reduce_scatter):

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

MPI_Reduce_scatter

Perform reduction on vector elements and distribute segments of result vector across all tasks in communicator

```
recvcount = 1;
MPI_Reduce_scatter(sendbuf, recvbuf, recvcount,
    MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```



MPI “多对多” 通信 (5)

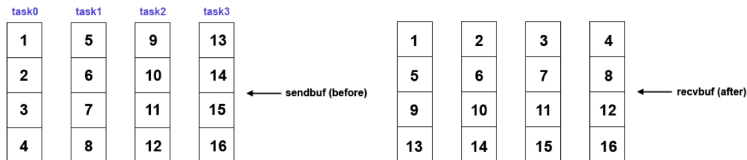
- 全交换 (alltoall):

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

MPI_Alltoall

Scatter data from all tasks to all tasks in communicator

```
sendcnt = 1;  
recvcnt = 1;  
MPI_Alltoall(sendbuf, sendcnt, MPI_INT  
recvbuf, recvcnt, MPI_INT  
MPI_COMM_WORLD);
```



内容提纲

- 1 入门知识
- 2 一个简单的例子
- 3 点对点通信
- 4 死锁与非阻塞通信
- 5 点对点通信实现机制
- 6 集合通信
- 7 辅助函数**
- 8 通信器与进程组
- 9 补遗与新特性

MPI 墙钟时间

- 返回当前进程的时钟时间：

```
double MPI_Wtime()
```

- 用法：

```
1  ...  
2  t0 = MPI_Wtime();  
3  ... // do some works  
4  t1 = MPI_Wtime();  
5  ...
```



- 返回 MPI_Wtime 的时钟刻度：

```
double MPI_Wtick()
```

一些其他辅助函数

- 判断 MPI 是否已经初始化：

```
int MPI_Initialized(int *flag)
```

- 中止 MPI 环境：

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

- 获取 MPI 版本号：

```
int MPI_Get_version(int *version, int *subversion)
```

- 获取处理器名：

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

示例程序：hello world 2!

mpi_hello2.c

```
1  ...
2  MPI_Init(&argc, &argv); // initialize MPI
3  MPI_Comm_size(MPI_COMM_WORLD, &size); // get num of procs
4  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get my rank
5  MPI_Get_processor_name(name, &len); // get node name
6  MPI_Get_version(&ver, &sver); // get mpi version
7
8  if ( size > 16 ) {
9      printf("Number of processes %d is too large. Abort MPI!\n",
10         size);
11      MPI_Abort(MPI_COMM_WORLD, 911); // abort mpi for large size
12  }
13
14  t0 = MPI_Wtime(); // tick
15  printf("On %s, from process %d of %d: Hello World!\n", name,
16         rank, size);
17  fflush(stdout); // flush standard output
```



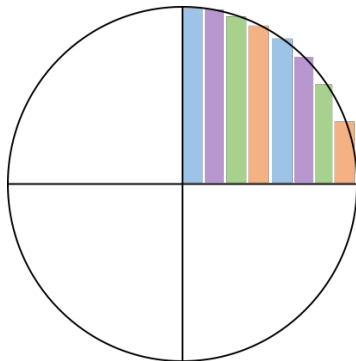
```
16 MPI_Barrier(MPI_COMM_WORLD); // barrier for the flush
17 t1 = MPI_Wtime(); // tock
18
19 if (rank == 0) printf("MPI version is %d.%d. Time ellapsed
    is %f.\nThat's all, folks!\n", ver, sver, t1-t0);
20 ...
```

● 运行结果:

```
On cu01, from process 2 of 4: Hello World!
On cu01, from process 0 of 4: Hello World!
On cu01, from process 3 of 4: Hello World!
On cu01, from process 1 of 4: Hello World!
MPI version is 3.1. Time elapsed is 0.000086.
That's all, folks!
```

示例程序：计算 π

- 计算依据：单位圆的面积。



$$\pi = 4 \int_0^1 \sqrt{1-x^2} dx$$
$$\approx 4h \sum_{i=0}^{N-1} \sqrt{1-x_i^2},$$

where

$$x_i = (i + \frac{1}{2})h, \quad h = \frac{1}{N}.$$

- 并行策略：round-robin。

mpi_cpi.c

```
1    ...
2    if (rank == 0) n = 10000000;
3    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
4
5    t0 = MPI_Wtime();
6    for (k = 0; k < REPEAT; k++) {
7        h = 1.0 / (double) n;
8        sum = 0.0;
9        for (i = rank + 1; i <= n; i += size) {
10            x = h * ((double)i - 0.5);
11            sum += 4.0 * sqrt(1.-x*x);
12        }
13        mypi = h * sum;
14        MPI_Allreduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,
15                      MPI_COMM_WORLD);
16    }
17    t1 = MPI_Wtime();
18
19    if (rank == 0) {
20        printf("Number of processes = %d\n", size);
```

```
20     printf(" pi is approximately %.16f\n", pi);
21     printf(" Error is %.16f\n", fabs(pi-PI25DT));
22     printf(" Wall clock time = %f\n", (t1-t0)/REPEAT);
23 }
24 ...
```

- 运行 (请正确使用 sbatch 或者 salloc):

```
$ mpiexec -n 1 ./cpi
$ mpiexec -n 2 ./cpi
$ mpiexec -n 4 ./cpi
$ mpiexec -n 8 ./cpi

Number of processes = 1
pi is approximately 3.1415926536003460
Error is 0.0000000000105529
Wall clock time = 0.079847
Number of processes = 2
pi is approximately 3.1415926536009313
Error is 0.0000000000111382
Wall clock time = 0.041626
Number of processes = 4
```

```
pi is approximately 3.1415926536006418
Error is 0.0000000000108487
Wall clock time = 0.023723
Number of processes = 8
pi is approximately 3.1415926536006591
Error is 0.0000000000108660
Wall clock time = 0.012587
```

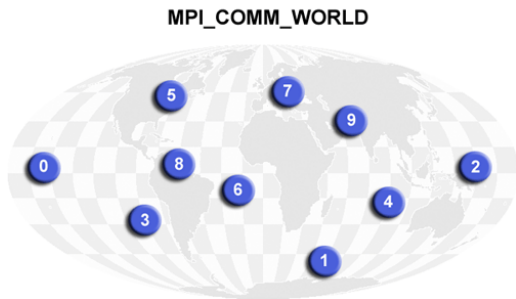
- 思考 1：为什么加速比不是线性的？
- 思考 2：为什么误差不同？

内容提纲

- 1 入门知识
- 2 一个简单的例子
- 3 点对点通信
- 4 死锁与非阻塞通信
- 5 点对点通信实现机制
- 6 集合通信
- 7 辅助函数
- 8 通信器与进程组**
- 9 补遗与新特性

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：

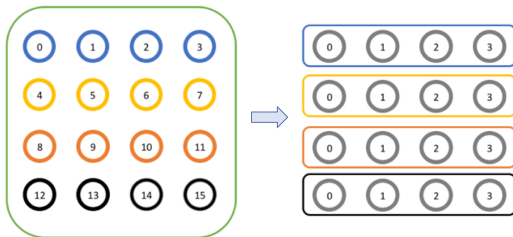
```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm*  
    newcomm)
```

- ▶ 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



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);
```

```
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 }
```

- 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
```

问题：不是分裂出了 4 个子通信器吗，为什么只有一个通信器变量 `sub_comm`？

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，但是可用集合操作创建新的进程组

- ▶ 子集、补集、交集、并集：

```
int MPI_Group_incl(MPI_Group group, int n, const int ranks
    [], MPI_Group* newgroup)
int MPI_Group_excl(MPI_Group group, int n, const int ranks
    [], MPI_Group* newgroup)
int MPI_Group_intersection(MPI_Group group1, MPI_Group
    group2, MPI_Group* newgroup)
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
    MPI_Group* newgroup)
```

- ▶ 若当前进程属于新进程组，newgroup 就是新创建的进程组；
 - ▶ 否则，newgroup 为 MPI_GROUP_EMPTY.
- 注意，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，可以创建通信器：

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

- ▶ comm 中所有进程都要调用本函数；
- ▶ 若当前进程在 group 中，newcomm 就是新创建的通信器；
- ▶ 否则，newcomm 为 MPI_COMM_NULL.

- 不使用时，需要释放进程组：

```
int MPI_Group_free(MPI_Group* group)
```

示例程序：进程组

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) {
```

```
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 }
```


- 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
```

内容提纲

- 1 入门知识
- 2 一个简单的例子
- 3 点对点通信
- 4 死锁与非阻塞通信
- 5 点对点通信实现机制
- 6 集合通信
- 7 辅助函数
- 8 通信器与进程组
- 9 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/>