

并行与分布式计算基础：第六讲

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

- ① MPI 基础知识回顾
- ② 点对点通信-1
- ③ 点对点通信-2

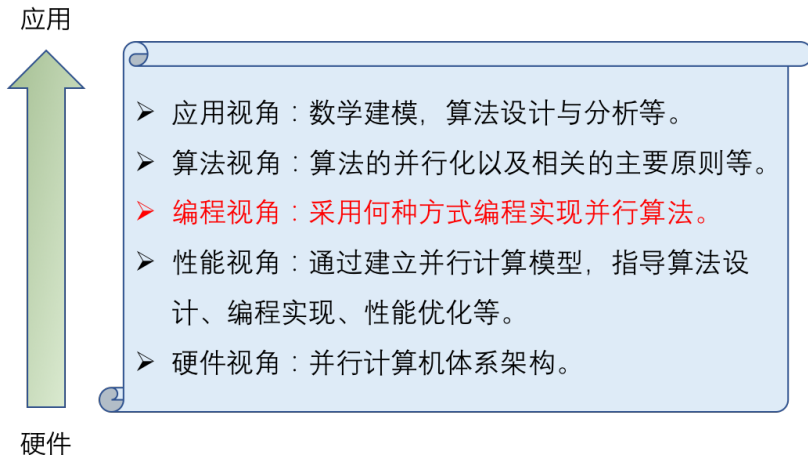
MPI 基础知识回顾

① MPI 基础知识回顾

② 点对点通信-1

③ 点对点通信-2

并行计算研究的几个主要视角

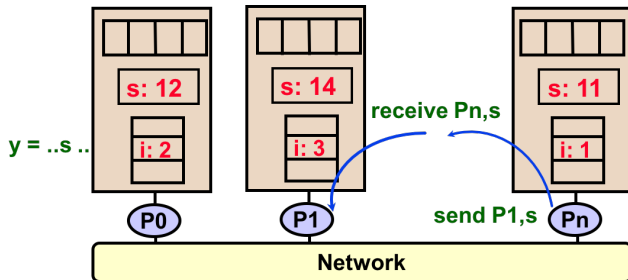


课程从应用与硬件切入，通过讨论算法与性能，最终聚焦并行编程。

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)
```


第一个 MPI 程序: hello world! (1)

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 }
```

第一个 MPI 程序: hello world! (2)

- 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!
```

点对点通信-1

① MPI 基础知识回顾

② 点对点通信-1

③ 点对点通信-2

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

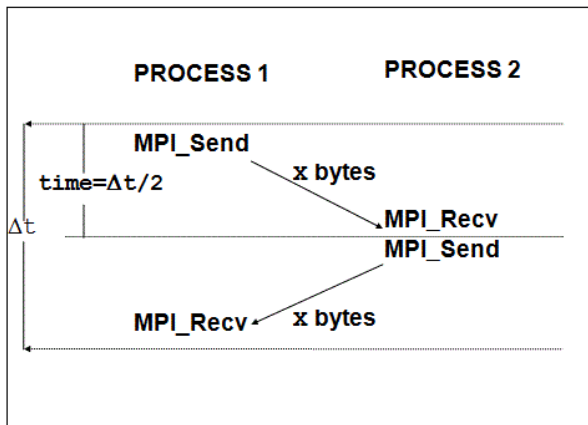
- 接收函数比发送函数多一个参数: `status`;
- 它保存了函数返回的通信状态, 可置为 `MPI_STATUS_IGNORE`;
- 其类型 `MPI_Status` 是一个 MPI 预定义的结构体:

```
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 (1)



示例程序：乒乓通信 (2)

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);
```


示例程序：乒乓通信 (3)

```
16     printf("Process %d ponged token %d from process %d\n", rank
    , token, 1-rank);
17 } else if (rank == 1) {
18     MPI_Recv(&token, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
19     MPI_Send(&token, 1, MPI_INT, 1-rank, 0, MPI_COMM_WORLD);
20 }
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 (1)

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);
```

示例程序：交换通信 (2)

```
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：

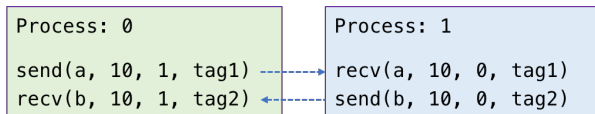
```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype  
    senddatatype, int dest, int sendtag, void *recvbuf, int  
    recvcount, MPI_Datatype recvdatatype, int source, int recvtag,  
    MPI_Comm comm, MPI_Status *status)
```

- 如果交换的数据共用一个 buffer，则可借助于如下操作：

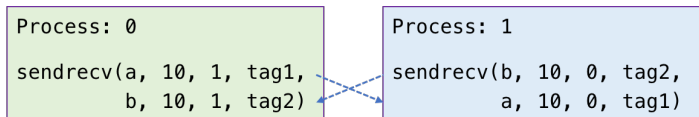
```
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype  
    datatype, int dest, int sendtag, int source, int recvtag,  
    MPI_Comm comm, MPI_Status *status)
```


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。

点对点通信-2

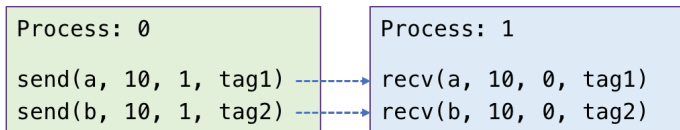
1 MPI 基础知识回顾

2 点对点通信-1

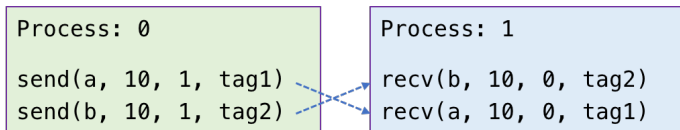
3 点对点通信-2

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  ...
```

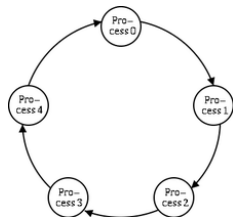


- 返回 MPI_Wtime 的时钟刻度:

```
double MPI_Wtick()
```

作业 1: 击鼓传花

- 完成击鼓传花程序，要求运用 MPI 点对点通信，在此基础上可以增加对延迟的测试等内容，程序需要有简单的说明；
- 截止时间：2019 年 10 月 23 日 24 点前。
- 提交方式：发送至助教邮箱。
- 正确性验证：数院机器，支持 2 至 16 进程（以 8 进程为例）



```
$ mpiexec -n 8 ./mpi_hw1
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
```