并行与分布式计算基础：第七讲

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）

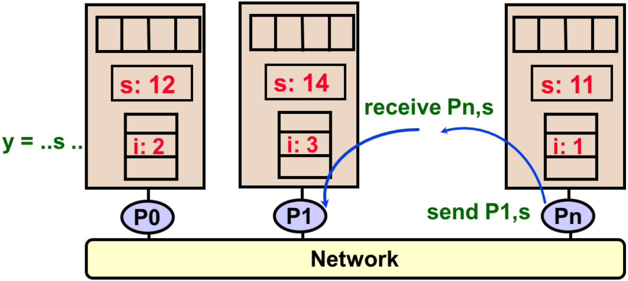
上课时间	星期一	星期二	星期三	星期四	星期五
第 1 节 (8:00-8:50)					
第 2 节 (9:00-9:50)					
第 3 节 (10:10-11:00)				单周	
第 4 节 (11:10-12:00)				单周	
第 5 节 (13:00-13:50)		每周			
第 6 节 (14:00-14:50)		每周			
第 7 节 (15:10-16:00)					
第 8 节 (16:10-17:00)					
第 9 节 (17:10-18:00)					
第 10 节 (18:40-19:30)					
第 11 节 (19:40-20:30)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

内容提纲

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)
```

使用非阻塞通信避免死锁

- 使用非阻塞发送避免死锁：

```
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，使用非阻塞接收避免死锁：

```
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)
```

MPI 墙钟时间

- 返回当前进程的时钟时间：

```
double MPI_Wtime()
```

- 用法：

```
...
t0 = MPI_Wtime();
... // do some works
t1 = MPI_Wtime();
...
```

- 返回 MPI_Wtime 的时钟刻度：

```
double MPI_Wtick()
```

MPI 集合通信-1

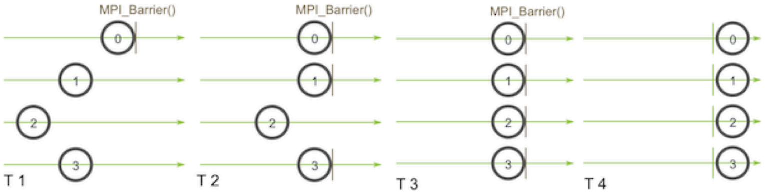
MPI 集合通信 (collective communication) 概述

- 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

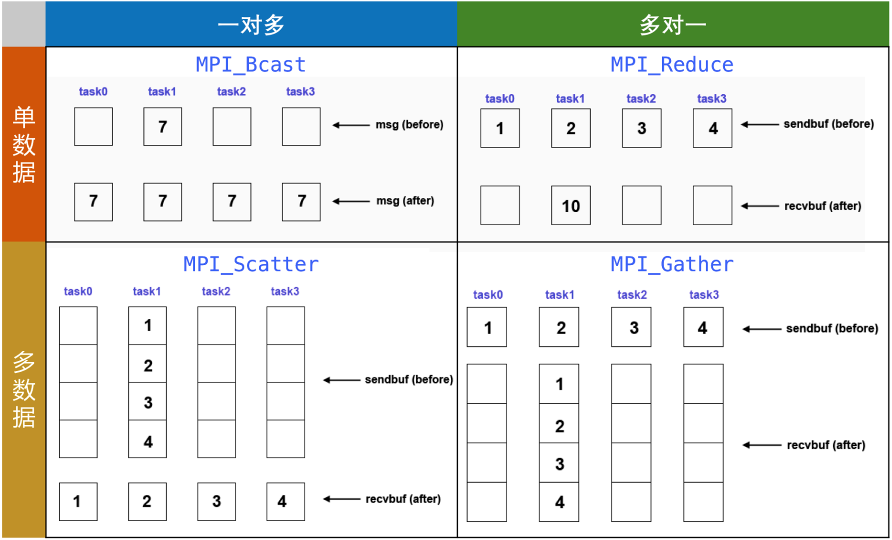
- 通信器中所有进程相互等待至某个同步点：

```
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



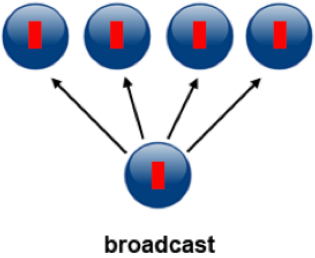
MPI 广播与规约

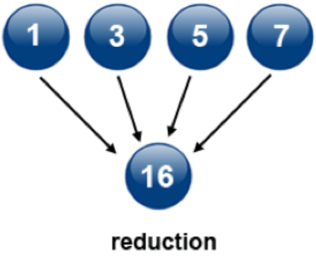
- 广播 (broadcast)：

```
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)
```





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 (1)

mpi_bcast_reduce.c

```c
...
#define ROOT   0 // Rank of the root process

int main(int argc, char **argv){
  ...
  /* Set the data on root process */
  if (rank == ROOT) {
    data = 100;
    printf("On root proc %d, data to be broadcast: %d\n", rank,
     data);
  }

  /* Broadcast the data to everybody */
  MPI_Bcast(&data, 1, MPI_INT, ROOT, MPI_COMM_WORLD);

  /* Everybody shows the broadcast data */
  printf("On proc %d, data after broadcasting = %d\n", rank,
    data);
```

示例程序：广播与规约 (2)

```
MPI_Barrier(MPI_COMM_WORLD);

/* Modify the data to reduce */
data = data + rank;
printf("On proc %d, data to be reduced = %d\n", rank, data);

/* Reduce everybody's data to root */
MPI_Reduce(&data, &data_reduce, 1, MPI_INT, MPI_SUM, ROOT,
  MPI_COMM_WORLD);

/* On root, show the reduced data */
if (rank == ROOT) {
  printf("On root proc %d, data reduced: %d\n", rank,
  data_reduce);
}
...
```

示例程序：广播与规约 (3)

- 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
```

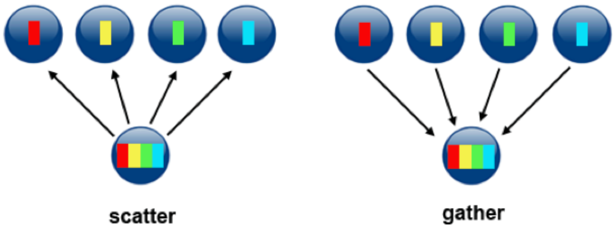
MPI 分发与集中

- 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 (1)

mpi_scatter_gather.c

```c
...
#define N       2 // Number of data per process
#define ROOT    0 // Rank of the root process

int main(int argc, char **argv){
  ...
  /* Set the data on root process */
  if (rank == ROOT) {
    printf("On root proc %d, data to be scattered:\n", rank);
    for ( j=0; j<size; j++) {
      for ( i=0; i<N; i++ ) {
        data_root[j][i] = j+i;
        printf("(%d, %d) = %d  ", j, i, data_root[j][i]);
      }
      printf("\n");
    }
  }

```

示例程序：分发与集中 (2)

```
/* Scatter the data to everybody */
MPI_Scatter(&data_root[0][0], N, MPI_INT, &data[0], N,
  MPI_INT, ROOT, MPI_COMM_WORLD);

/* Everybody shows the scatterd data */
for (i=0; i<N; i++) {
  printf("On proc %d, scattered data[%d] = %d\n", rank, i,
  data[i]);
}
MPI_Barrier(MPI_COMM_WORLD);

/* Gather everybody's data to root */
MPI_Gather(&data[0], N, MPI_INT, &data_root[0][0], N, MPI_INT
  , ROOT, MPI_COMM_WORLD);

/* On root, show the gathered data */
if (rank == ROOT) {
  printf("On root proc %d, data gathered:\n", rank);
  for ( j=0; j<size; j++) {
```

示例程序：分发与集中 (3)

```c
      for ( i=0; i<N; i++ ) {
        printf("(%d, %d) = %d  ", j, i, data_root[j][i]);
      }
      printf("\n");
    }
  }
  ...
```

- 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
```

示例程序：分发与集中 (4)

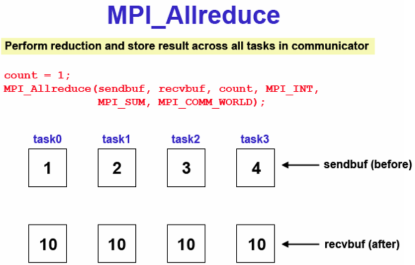
```
On proc 2, scattered data[0] = 2
On proc 2, scattered data[1] = 3
On proc 1, scattered data[0] = 1
On proc 3, scattered data[0] = 3
On proc 3, scattered data[1] = 4
On proc 1, scattered data[1] = 2
Data gathered on root proc 0:
(0, 0) = 0  (0, 1) = 1
(1, 0) = 1  (1, 1) = 2
(2, 0) = 2  (2, 1) = 3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

MPI 集合通信-2

MPI“多对多”通信 (1)

- 全规约 (allreduce)：

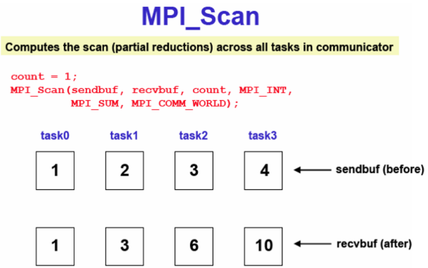
```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```



MPI“多对多”通信 (2)

- 前缀和 (scan, prefix-sum)：

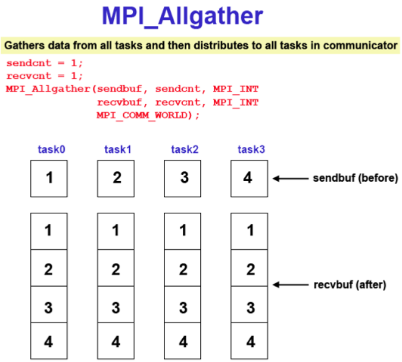
```
int MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype
    datatype, MPI_Op op, MPI_Comm comm)
```



MPI “多对多” 通信 (3)

- 全集中 (allgather):

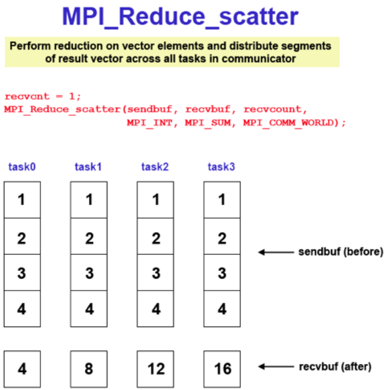
```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype
  senddatatype, void *recvbuf, int recvcount, MPI_Datatype
  recvdatatype, MPI_Comm comm)
```



MPI “多对多” 通信 (4)

- 规约分发 (reduce_scatter)：

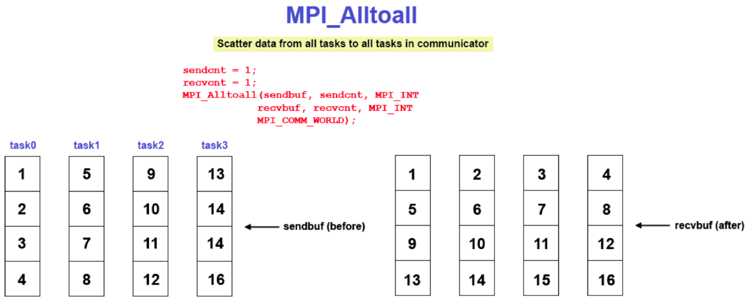
```
int MPI_Reduce_scatter(void *sendbuf, void *recvbuf, const int
    recvcount[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```



MPI "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)
```



辅助函数

一些重要的辅助函数

- 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！ (1)

mpi_hello2.c

```
...
MPI_Init(&argc, &argv); // initialize MPI
MPI_Comm_size(MPI_COMM_WORLD, &size); // get num of procs
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get my rank
MPI_Get_processor_name(name, &len); // get node name
MPI_Get_version(&ver, &sver); // get mpi version

if ( size > 16 ) {
  printf("Number of processes %d is too large. Abort MPI!\n",
    size);
  MPI_Abort(MPI_COMM_WORLD, 911); // abort mpi for large size
}

t0 = MPI_Wtime(); // tick
printf("On %s, from process %d of %d: Hello World!\n", name,
   rank, size);
fflush(stdout); // flush standard output
```

示例程序：hello world 2！ (2)

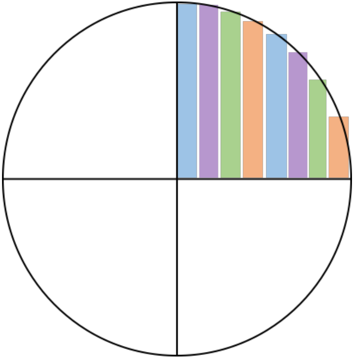
```
MPI_Barrier(MPI_COMM_WORLD); // barrier for the flush
t1 = MPI_Wtime(); // tock

if (rank == 0) printf("MPI version is %1d.%1d. Time ellapsed
  is %f.\nThat's all, folks!\n", ver, sver, t1-t0);
...
```

- 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 (1)

- 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。

示例程序：计算 π (2)

mpi_cpi.c

```c
...
if (rank == 0) n = 10000000;
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

t0 = MPI_Wtime();
for (k = 0; k < REPEAT; k++) {
  h = 1.0 / (double) n;
  sum = 0.0;
  for (i = rank + 1; i <= n; i += size) {
    x = h * ((double)i - 0.5);
    sum += 4.0 * sqrt(1.-x*x);
  }
  mypi = h * sum;
  MPI_Allreduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,
   MPI_COMM_WORLD);
}
t1 = MPI_Wtime();
```

示例程序：计算 π (3)

```c

if (rank == 0) {
  printf("Number of processes = %d\n", size);
  printf(" pi is approximately %.16f\n", pi);
  printf(" Error is %.16f\n", fabs(pi-PI25DT));
  printf(" Wall clock time = %f\n", (t1-t0)/REPEAT);
}
...
```

- 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
```

示例程序：计算 π (4)

```
 Wall clock time = 0.104462
Number of processes = 2
 pi is approximately 3.1415926536009313
 Error is 0.0000000000111382
 Wall clock time = 0.061566
Number of processes = 4
 pi is approximately 3.1415926536006418
 Error is 0.0000000000108487
 Wall clock time = 0.039160
Number of processes = 8
 pi is approximately 3.1415926536006591
 Error is 0.0000000000108660
 Wall clock time = 0.026021
```

- 思考 1：为什么加速比不是线性的？
- 思考 2：为什么误差不同？