

## MPI 并行编程模型

### 1. 环境配置

- 1.1 安装MPICH2
- 1.2 VS中配置MPI环境
- 1.3 CentOS 安装OpenMPI3.1.0

### 2. MPI 编程实验

Task1: MPI环境管理\*

Task2: Hello World\*

Task3: 基本函数

3.1 Bcast

3.2 gather

3.3 scatter

3.4 reduce

Task3: Nonblocking send/receive\*

Task4: 计算Pi

4.1 MPI\_Bcast 课堂案例\*

4.2 点对点通讯

4.3 broadcast

4.4 gather

4.5 reduce

Task5: openMp和mpi 混合编程 \*

Task6 : 多节点测试\*

6.1算法设计

6.2 程序代码

6.3 单节点测试

6.4 多节点配置

6.4.1 host配置

6.4.2 配置ssh免密登录

6.5 多节点运行

6.6 性能分析

总结

参考

姓名：张煜玮

学号：19291255

班级：计科1905

注：目录中带星号的是作业要求必做的，其余的是拓展的内容

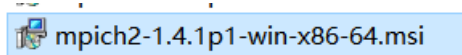
# MPI 并行编程模型

## 1. 环境配置

### 1.1 安装MPICH2

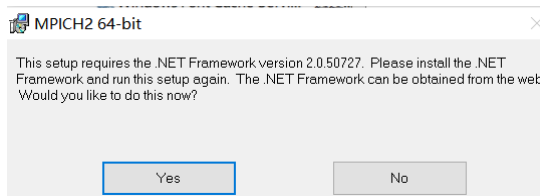
下载并安装MPICH2

1. 下载并打开“mpich2-1.4.1p1-win-ia64.msi”。

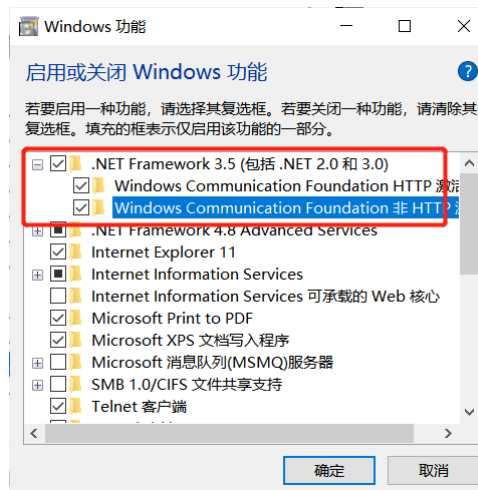


2. 安装64位的MPICH2

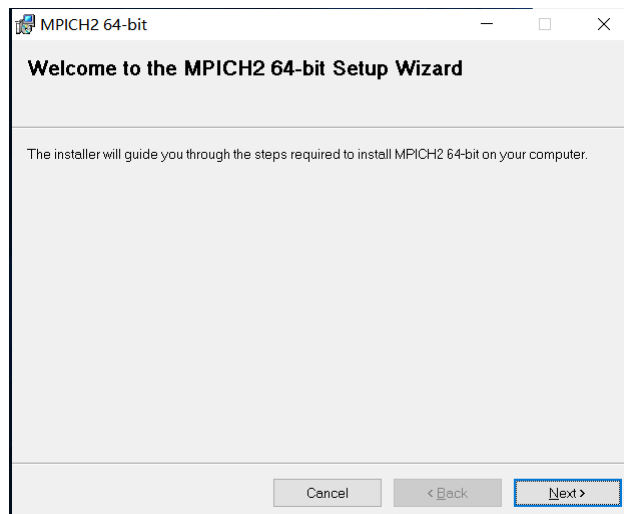
如果安装的时候，给出如下的错误提示，没有启动.NET 服务



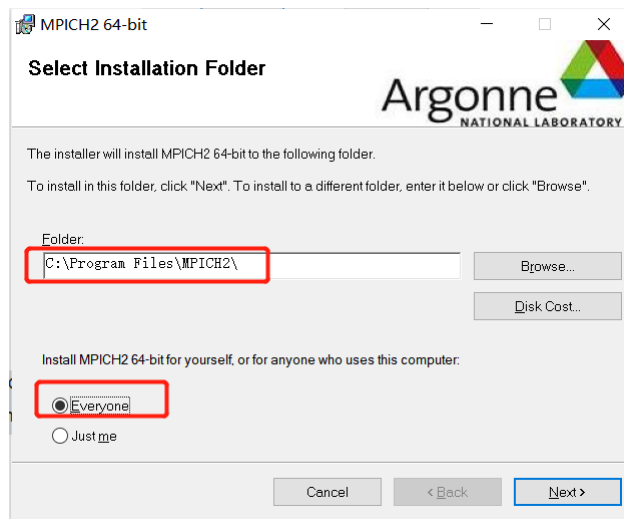
这是需要在“WINDOWS”功能中勾选“.NET FRAMEWORK”的选项



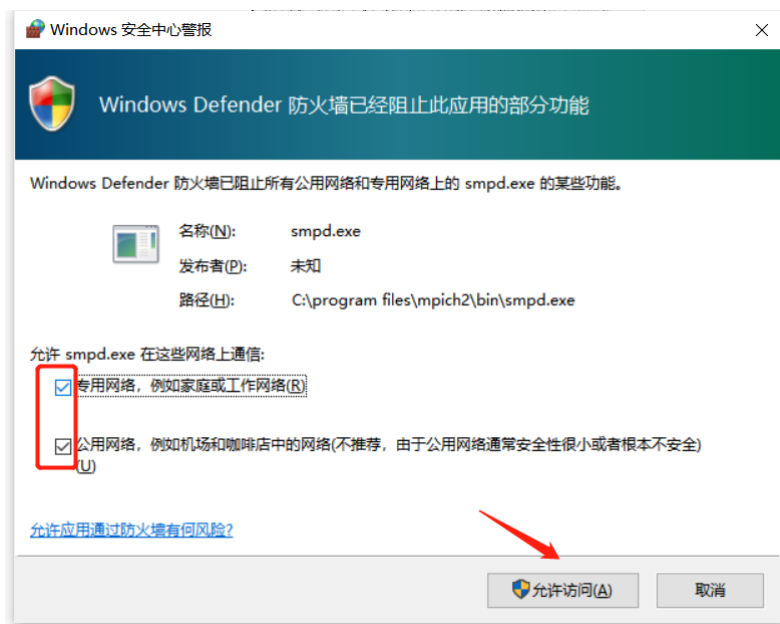
然后就可以点击“下一步”安装



选择默认的安装路径，勾选“Everyone”

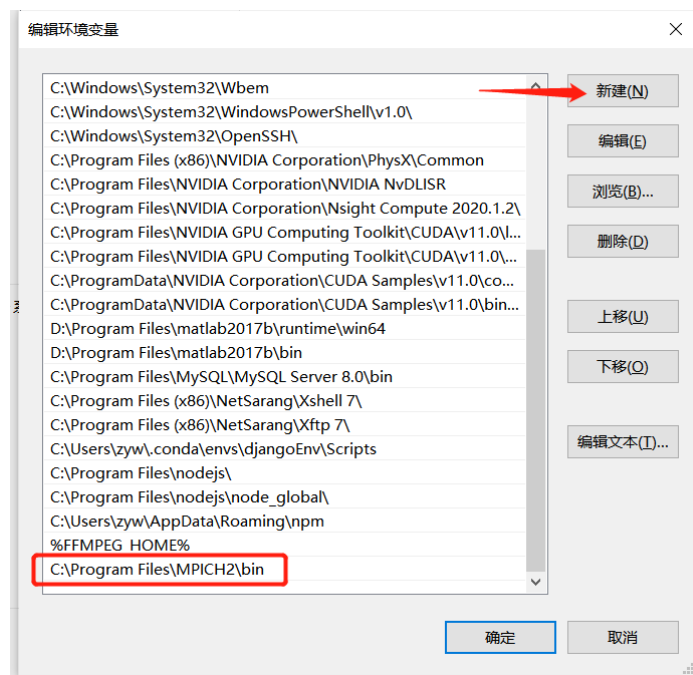


当出现防火墙提示时，专有网络和公用网络都要勾选



### 3. 添加环境变量

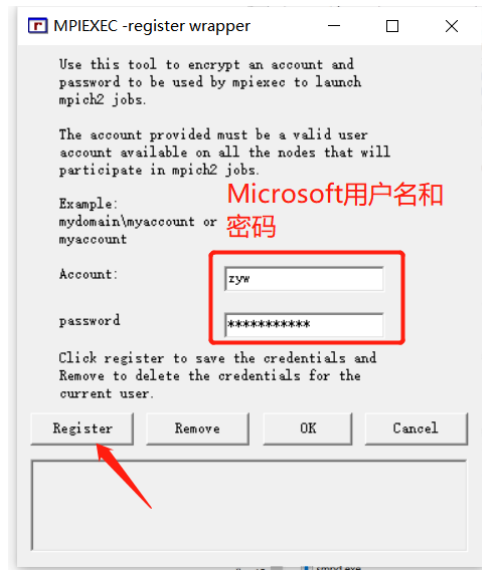
在系统变量中的path下添加 C:\Program Files\MPICH2\bin



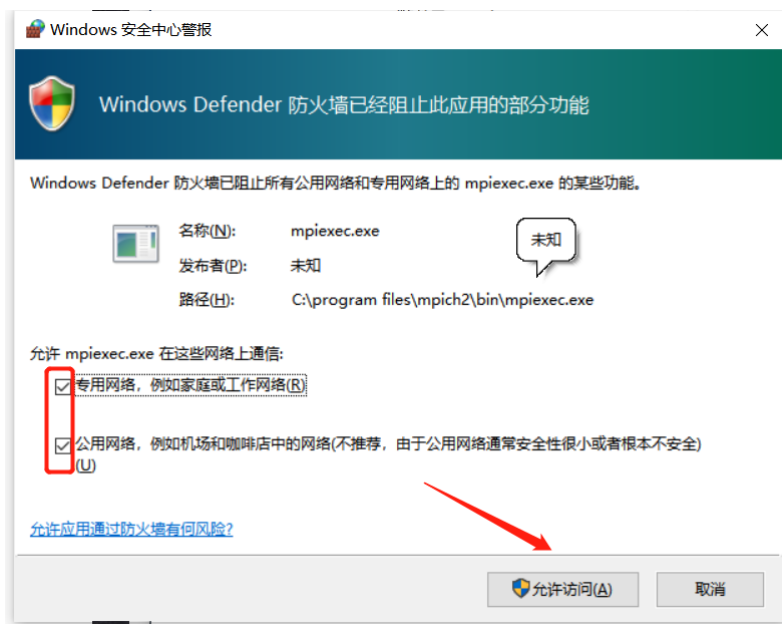
4. 打开路径“C:\Program Files\MPICH2\bin”，打开“wmpiregister.exe”文件

MPICH2 > bin			
搜索"bin"			
名称	修改日期	类型	
clog2Toslog2.jar	2011/6/29 15:38	JAR 文件	
irlog2rlog.exe	2011/9/1 15:30	应用程序	
jumpshot.jar	2011/6/29 15:38	JAR 文件	
jumpshot_launcher.jar	2011/6/29 15:38	JAR 文件	
mpiexec.exe	2011/9/1 15:00	应用程序	
smpd.exe	2011/9/1 15:00	应用程序	
traceToslog2.jar	2011/6/29 15:38	JAR 文件	
wmpiconfig.exe	2011/9/1 15:30	应用程序	
wmpiexec.exe	2011/9/1 15:30	应用程序	
wmpiregister.exe	2011/9/1 15:30	应用程序	

5. 输入Microsoft的用户名和密码（同锁屏时用户名和密码）



如果弹出防火墙警报，两个都要允许



6. 出现提示“Password encrypted into the Registry”，点击OK即可。



7. 以管理员的身份运行cmd



8. 输入命令 `smpd -install -phrase behappy` (注意: 这里的behappy需要与安装时的Passphrase一样), 如果出现提示“MPICH2 Process Manager, Argonne National Lab installed.”则说明安装成功。

输入命令 `smpd -status` 查询进程状态

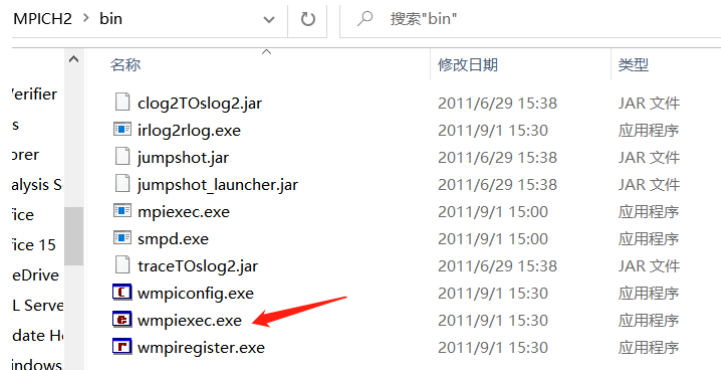
输入命令 `mpiexec -validate` 检测是否注册成功, 成功则会提示“SUCCESS”。

```
C:\Windows\system32>cd C:\Program Files\MPICH2\bin
C:\Program Files\MPICH2\bin>smpd -install -phrase behappy
MPICH2 Process Manager, Argonne National Lab installed.
C:\Program Files\MPICH2\bin>smpd -status
smpd running on LAPTOP-3H6PP4K0
C:\Program Files\MPICH2\bin>mpiexec -validate
SUCCESS
```

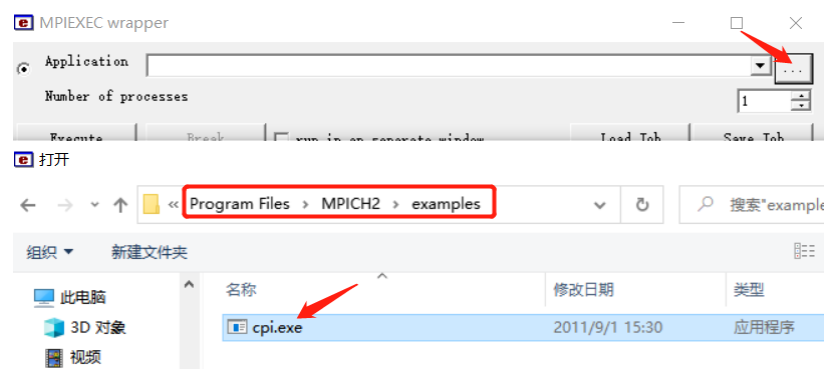
9. 进入文件夹 `C:\Program Files\MPICH2\examples`, 运行 `...\bin\mpiexec.exe -n 2 cpi.exe` 程序运行正常则说明MPICH2安装配置成功

```
C:\Program Files\MPICH2\bin>cd C:\Program Files\MPICH2\examples
C:\Program Files\MPICH2\examples>..\bin\mpiexec.exe -n 2 cpi.exe
Enter the number of intervals: (0 quits) 100000000
pi is approximately 3.1415926535900223, Error is 0.0000000000002292
wall clock time = 0.705926
```

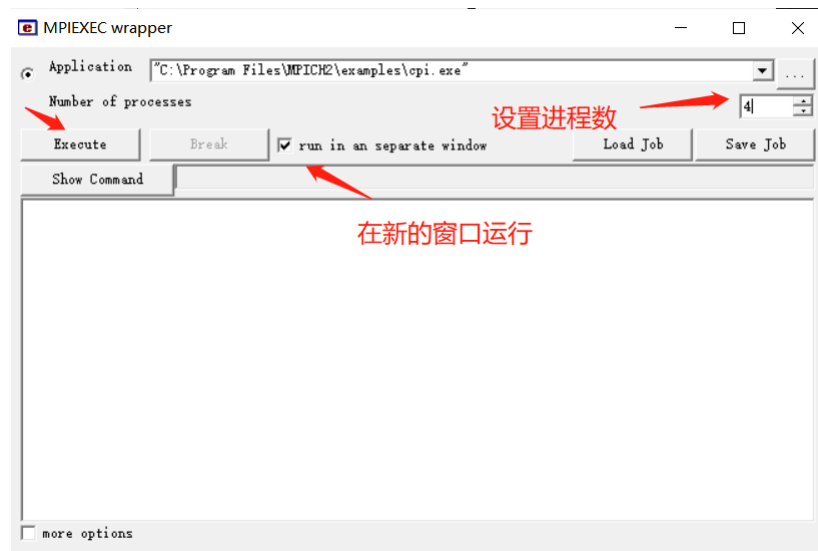
10. 打开路径 `C:\Program Files\MPICH2\bin` 下的“wmpiexec.exe”文件。



点右上角“三个点”,选择路径“C:\Program Files\MPICH2\examples”下的“cpi.exe”文件



勾选“run in an separate window”。设置进程数, 这里设置了4。点击Execute。

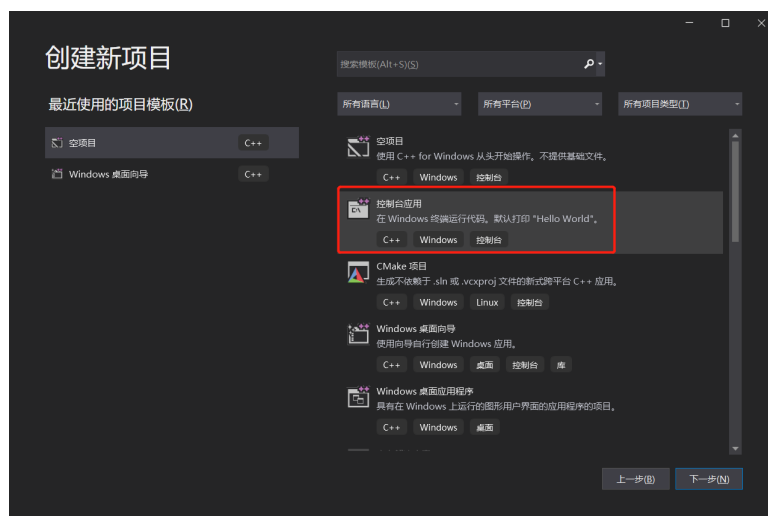


运行结果如下：

```
ca: C:\Windows\System32\cmd.exe
Enter the number of intervals: (0 quits) 4
pi is approximately 3.1468005183939427, Error is 0.0052078648041496
wall clock time = 0.000193
Enter the number of intervals: (0 quits) 0
请按任意键继续. . .
```

## 1.2 VS中配置MPI环境

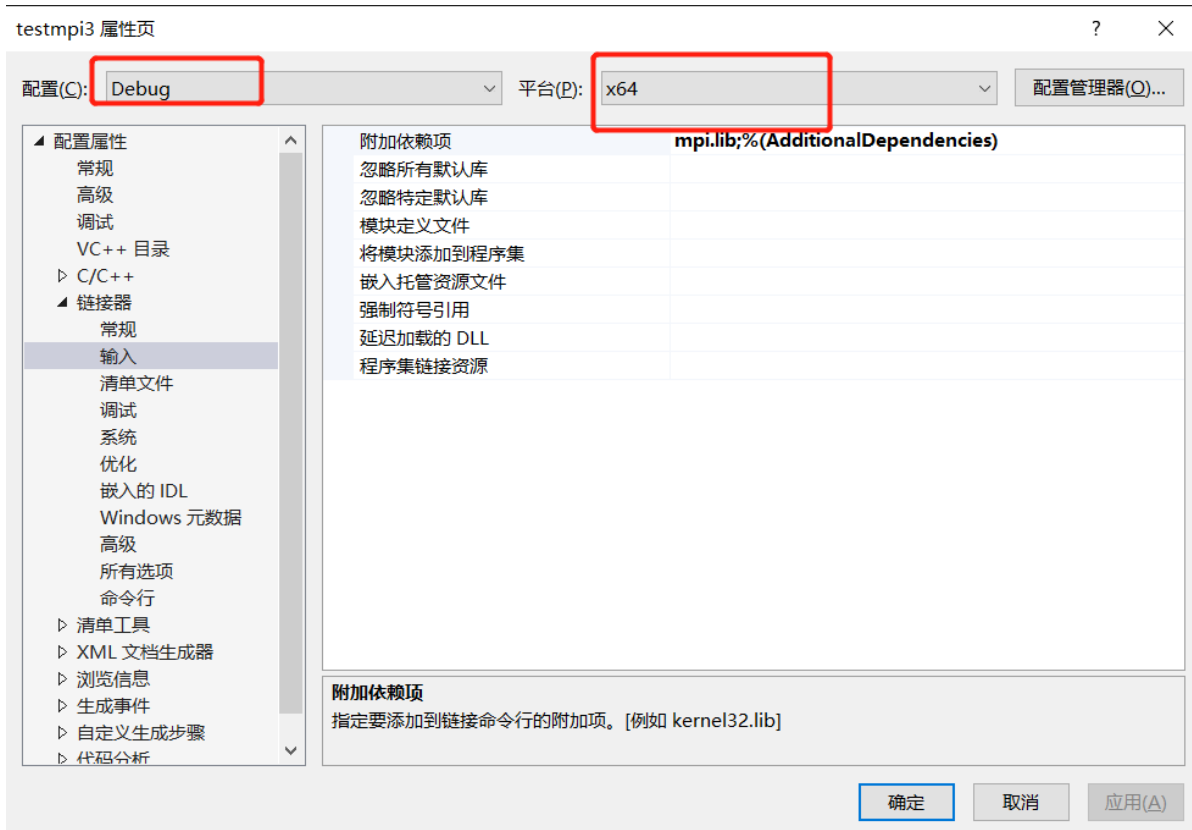
1. 打开VS，新建Visual C++里面的"Windows 控制台程序"



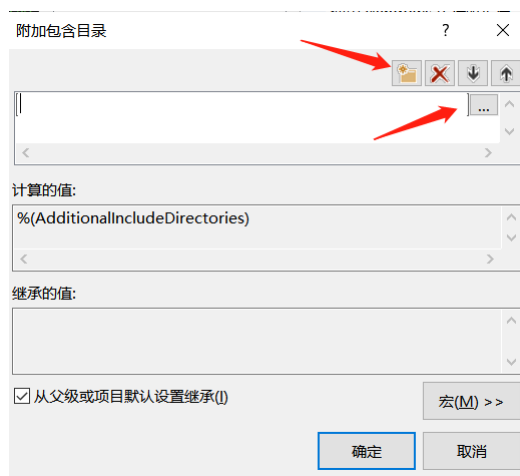
2. 打开项目的属性



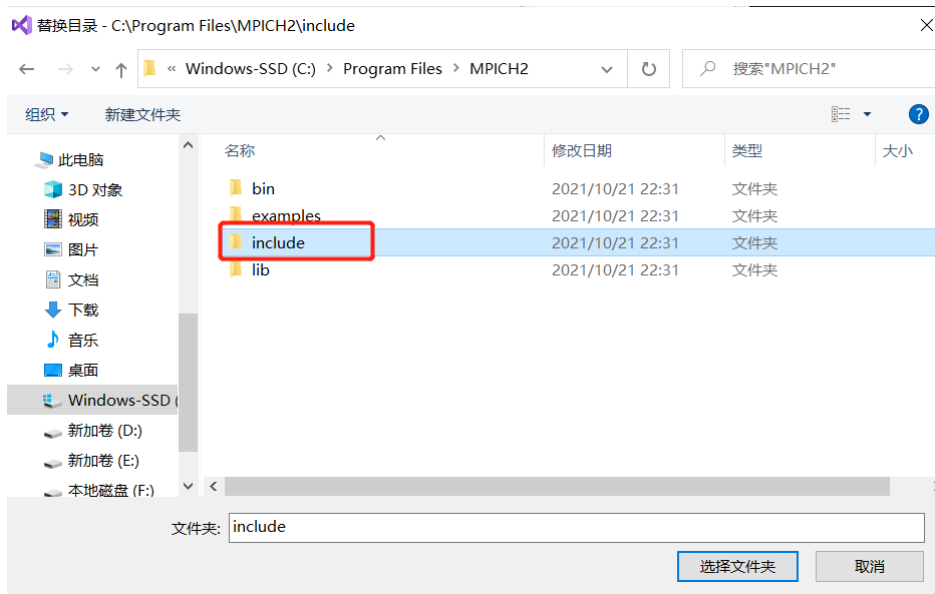
3. 先将上面的“配置”改为“debug”,平台选择X64, 然后点击左侧“C/C++”, 我们要修改的是“附加包含目录”, 点击“<编辑...>”



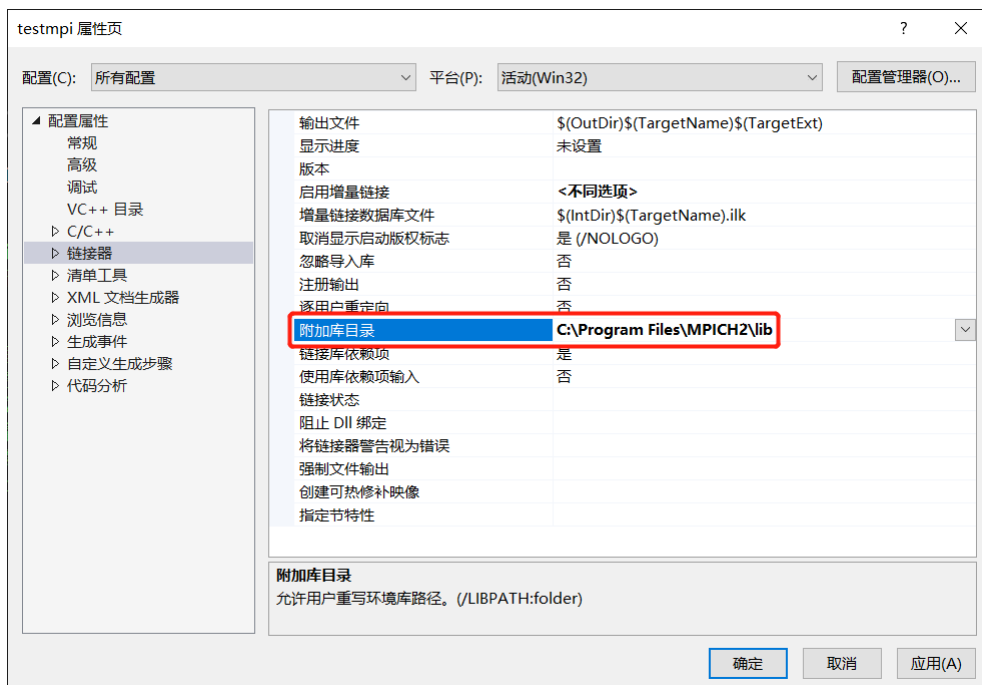
4. 点击“新建”图标, 然后点击“三个点”。



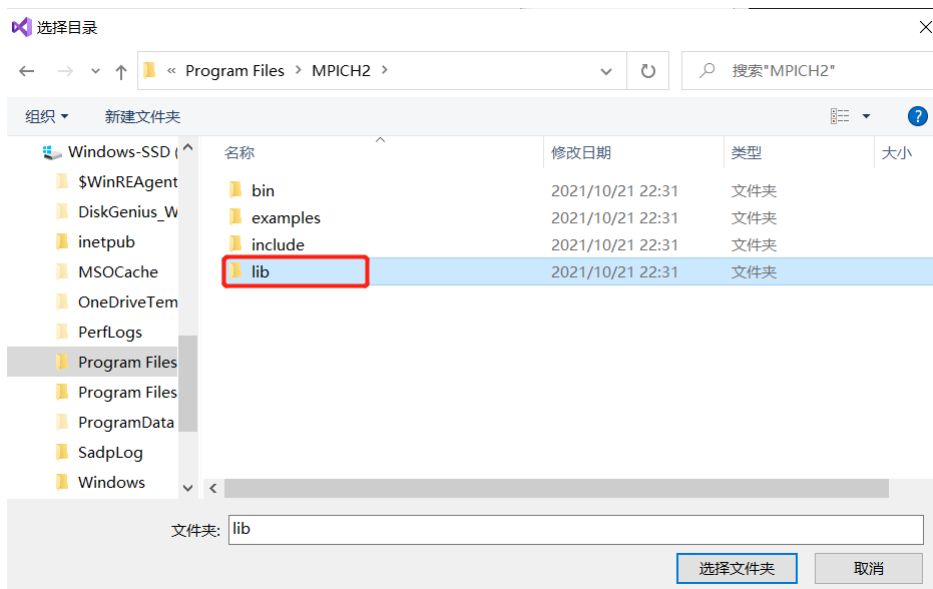
5. 选择路径“C:\Program Files (x86)\MPICH2\include”, 点击“选择文件夹”。然后“确定”。



6. 点击左侧“链接器”，这里我们要修改的是“附加库目录”。同上，点击“<编辑...>”，点击“新建”图标，然后点击“三个点”。

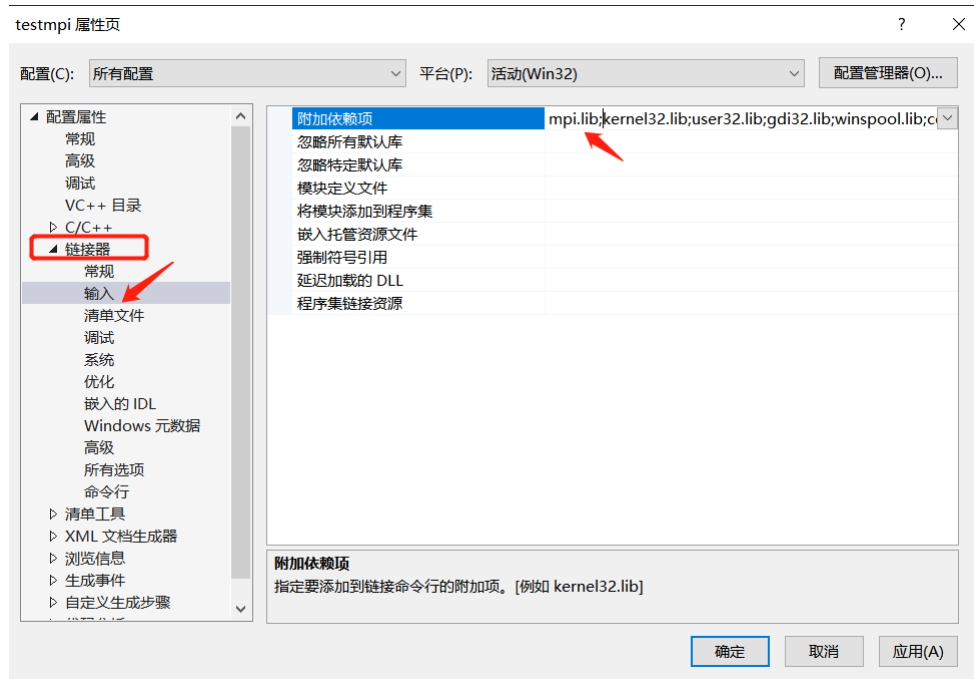


7. 选择路径“C:\Program Files\MPICH2\lib”，点击“选择文件夹”。然后“确定”。



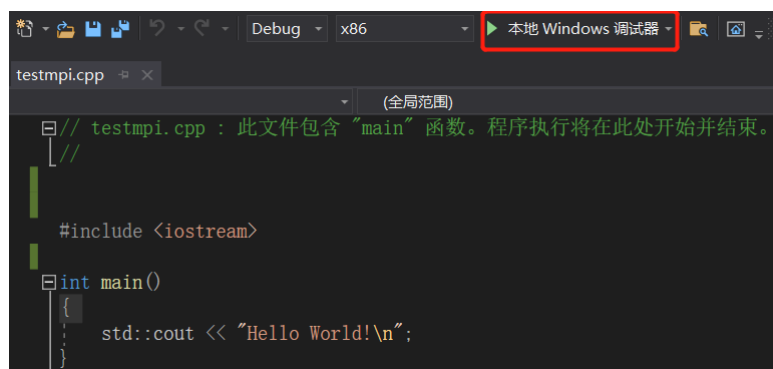


8. 展开“链接器”，点击“输入”，在右侧第一项“附加依赖项”，前面加上“mpi.lib;”，注意不要忘记分号。

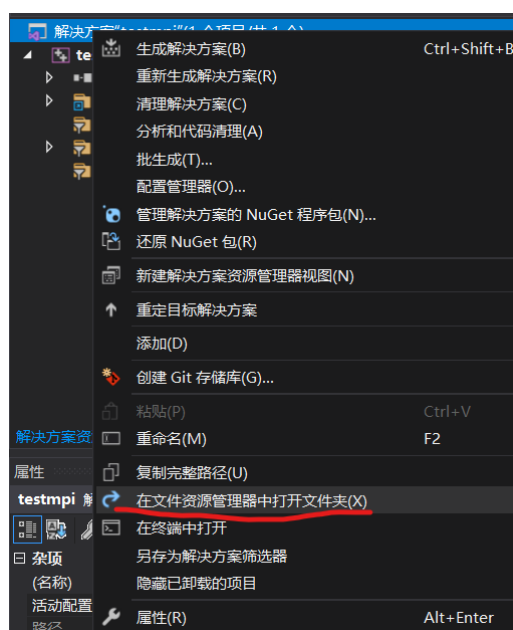


9. 点击“应用”、“确定”完成配置。

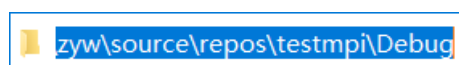
10. 写一个输出的程序，生成exe。



11. 右键单击“解决方案”，点击“在文件资源管理器中打开文件夹(X)”



12. 打开Debug文件夹，点击上方地址栏空白处，复制好路径。



13. 打开cmd，输入 `mpiexec -n 4 C:\Users\zyw\source\repos\testmpi\Debug\testmpi`，因为上一步复制过路径，所以该命令行中的路径处可以用粘贴方式输入。“-n 4”是参数，是指4个进程，后面就是路径+文件名

14. 运行结果如下，一共输出四个Hello World

```
C:\Users\zyw>mpiexec -n 4 C:\Users\zyw\source\repos\testmpi\Debug\testmpi
Hello World!
Hello World!
Hello World!
Hello World!
```

## 1.3 CentOS 安装OpenMPI3.1.0

此部分为拓展内容，在个人的阿里云服务器(1核)上配置OpenMp

1. 下载OpenMPI 源码

```
wget https://download.open-mpi.org/release/open-mpi/v3.1/openmpi-3.1.0.tar.gz
```

2. 解压缩OpenMPI源码

```
tar -zxvf openmpi-3.1.0.tar.gz
```

3. 安装OpenMPI

```
openmpi-3.1.0/
./configure --prefix=/usr/local/openmpi
make && make install
```

4. 配置环境变量

```
whereis openmpi
vim ~/.bash_profile
```

5. 将以下两句添加到.bash\_profile文件末尾位置，按Esc后:wq保存修改

```
1 export PATH=$PATH:/usr/local/openmpi/bin
2 export LD_LIBRARY_PATH=/usr/local/openmpi/lib
```

6. 使用source命令激活修改

```
source ~/.bash_profile
```

7. 验证安装

```
cd examples/
make
./hello_c
```

```
[root@iZ2ze7vmdw24up9ztboj6pZ examples]# ./hello_c
Hello, world, I am 0 of 1, (Open MPI v3.1.0, package: Open MPI root@iZ2ze7vmdw24up9ztboj6pZ Distribution, ident: 3.1.0, repo rev: v3.1.0, May 07, 2018, 123)
```

## 2. MPI 编程实验

## Task1: MPI环境管理\*

- 实验代码

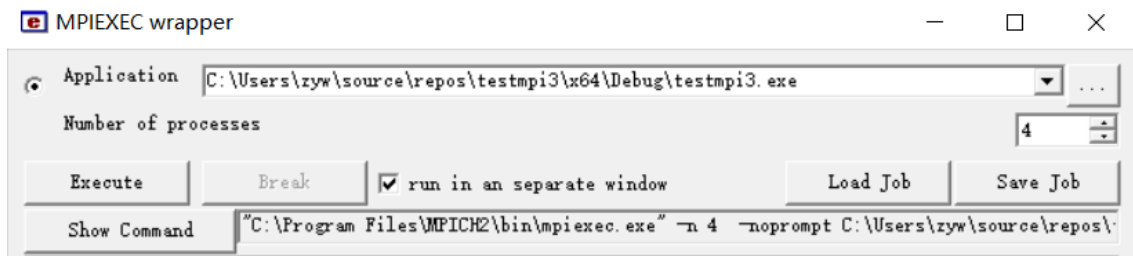
```
1  #include "stdio.h"
2  #include "mpi.h"
3
4  int main(){
5      int size,rank,namelen;
6      char name[MPI_MAX_PROCESSOR_NAME];
7      MPI_Init(NULL,NULL);
8      MPI_Comm_size(MPI_COMM_WORLD,&size);
9      MPI_Comm_rank(MPI_COMM_WORLD,&rank);
10     MPI_Get_processor_name(name,&namelen);
11     printf("size=%d,rank=%d,name=%s,len=%d \n",size,rank,name,namelen);
12     fflush(stdout);
13     MPI_Finalize();
14     return 0;
15 }
```

- 实验结果

1. 个人电脑:

【运行方法1】

1. 打开C:\Program Files\MPICH2\bin文件下的wmpiexec.exe文件
2. 打开VS项目下生成的exe文件, 在 Number of process 中设置线程数, 勾选 run in an seperate window ,最后点击 Execute



```
C:\Windows\System32\cmd.exe
size=4,rank=2,name=LAPTOP-3H6PP4K0,len=15
size=4,rank=0,name=LAPTOP-3H6PP4K0,len=15
size=4,rank=3,name=LAPTOP-3H6PP4K0,len=15
size=4,rank=1,name=LAPTOP-3H6PP4K0,len=15
请按任意键继续. . .
```

【运行方法2】

1. 打开cmd, 输入 `mpiexec -n 4 C:\Users\zyw\source\repos\testmpi3\Debug\testmpi3`

```
C:\Windows\System32\cmd.exe
size=4,rank=1,name=LAPTOP-3H6PP4K0,len=15
size=4,rank=3,name=LAPTOP-3H6PP4K0,len=15
size=4,rank=2,name=LAPTOP-3H6PP4K0,len=15
size=4,rank=0,name=LAPTOP-3H6PP4K0,len=15
请按任意键继续. . .
```

## 2. 服务器

输入命令 `./mpi.sh task1.c 4`

```
[student74@kunpeng192 ~]$ ./mpi.sh task1.c 4
size=4,rank=3,name=kunpeng192,len=10
size=4,rank=2,name=kunpeng192,len=10
size=4,rank=0,name=kunpeng192,len=10
size=4,rank=1,name=kunpeng192,len=10

real    0m0.008s
user    0m0.016s
sys     0m0.002s
```

### • 结果分析

上述代码体现了MPI编程的基本架构，具体来说包括一下几步：

#### 1. MPI程序的初始化

任何一个MPI程序在调用MPI函数之前，首先调用的是初始化函数 `MPI_INIT`

建立MPI的执行环境 `int MPI_Init(int *argc, char ***argv)`

#### 2. 获取通信域包含的进程数

`MPI_Comm_Size(MPI_Comm, int *size)`

#### 3. 获取当前进程标识

`MPI_Comm_rank(MPI_Comm comm, int *rank)`

#### 4. MPI 程序结束

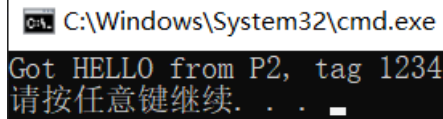
`MPI_Finalize()` :结束MPI程序的执行

其结果显示size=4, 即进程数为4; rank分别表示当前进程的编号。其中 `MPI_Get_processor_name` 用来获得本机的名称。len 表示的是name的长度。

## Task2: Hello World\*

### • 实验代码

```
1  #include <stdio.h>
2  #include <mpi.h>
3  int main() {
4      MPI_Status status;
5      char string[] = "xxxxx";
6      char buf[] = "HELLO";
7      //void* pbuf = buf;
8      int myid;
9      MPI_Init(NULL, NULL);
10     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
11     if (myid == 2)
12         MPI_Send(buf, 5, MPI_CHAR, 7, 1234, MPI_COMM_WORLD);
13     if (myid == 7) {
14         MPI_Recv(string, 5, MPI_CHAR, 2, MPI_ANY_TAG, MPI_COMM_WORLD,
15         &status);
16         printf("Got %s from P%d, tag %d\n", string, status.MPI_SOURCE,
17         status.MPI_TAG);
18         fflush(stdout);
19     }
20     MPI_Finalize();
21 }
```



```
C:\Windows\System32\cmd.exe
Got HELLO from P2, tag 1234
请按任意键继续. . .
```

```
mpiexec -n 8 C:\Users\zyw\source\repos\testmpi3\x64\Debug\testmpi3
```

## Task3: 基本函数

### 3.1 Bcast

- 实验代码

```
1  #include "mpi.h"
2  #include <stdio.h>
3  #define N 3
4
5  int main(int argc, char *argv[]) {
6      int i, myrank, nprocs;
7      int buffer[N];
8      MPI_Init(&argc, &argv);
9      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
10     MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
11     printf("myrank=%d \n before bcasting my buffer's data is: \n ", myrank);
12     for (int i = 0; i < N; i++) {
13         buffer[i] = myrank + i;
14         printf("buffer[%d]=%d\n", i, buffer[i]);
15     }
16     printf("\n");
17     MPI_Bcast(buffer, N, MPI_INT, 0, MPI_COMM_WORLD);
18
19     printf("after bcasting my buffer's data is:\n");
20     for (int i = 0; i < N; i++)
21         printf("buffer[%d]=%d\n", i, buffer[i]);
22     printf("\n");
23     MPI_Finalize();
24     return 0;
25 }
```

- 运行结果

```

[student74@kunpeng192 ~]$ ./mpi.sh bcast.c 3
myrank=2
  before bcasting my buffer's data is:
  buffer[0]=2
buffer[1]=3
buffer[2]=4

myrank=0
  before bcasting my buffer's data is:
  buffer[0]=0
buffer[1]=1
buffer[2]=2

myrank=1
  before bcasting my buffer's data is:
  buffer[0]=1
buffer[1]=2
buffer[2]=3

after bcasting my buffer's data is:
buffer[0]=0
buffer[1]=1
buffer[2]=2

after bcasting my buffer's data is:
buffer[0]=0
buffer[1]=1
buffer[2]=2

after bcasting my buffer's data is:
buffer[0]=0
buffer[1]=1
buffer[2]=2

real    0m0.007s
user    0m0.014s
sys     0m0.000s

```

### • 结果分析

广播 (broadcast) 是标准的集体通信技术之一。一个广播发生的时候，一个进程会把同样一份数据传递给一个 communicator 里的所有其他进程。广播的主要用途之一是把用户输入传递给一个分布式程序，或者把一些配置参数传递给所有的进程。

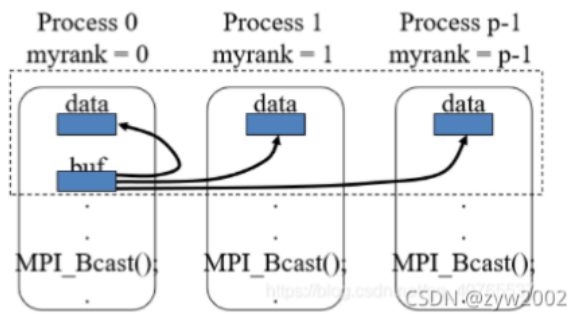
**MPI\_Bcast**：一对多广播同样的消息

```

1  int MPI_Bcast (
2      void buffer, /* 发送/接收buf/
3      int count, /*元素个数*/
4      MPI_Datatype datatype,
5      int root, /*指定根进程*/
6      MPI_Comm comm)

```

根进程既是发送缓冲区也是接收缓冲区。在本例中将0号进程作为root, 即把0号进程中的buffer值，分别为0, 1, 2 送到0号, 1号, 2号进程。所以在广播结束后，进程0, 1, 2 的 buffer[0]=0,buffer[1]=1,buffer[2]=2。



## 3.2 gather

- 实验代码

```

1  #include "mpi.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(int argc, char *argv[]) {
6      int size, rank;
7      int send[5];
8      int* recv;
9      int i = 0;
10     MPI_Init(&argc, &argv);
11     MPI_Comm_size(MPI_COMM_WORLD, &size);
12     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13
14     for (; i < 5; i++) {
15         send[i] = i + rank * 10;
16     }
17     printf("-----\n");
18     for (i = 0; i < 5; i++)
19         printf("%d ", send[i]);
20     printf("\n");
21     if (rank == 0)
22         recv = (int*)malloc(size * 5 * sizeof(int));
23     MPI_Gather(send, 5, MPI_INT, recv, 5, MPI_INT, 0, MPI_COMM_WORLD);
24     if (rank == 0) {
25         printf("I'm root process, and the data that i received is:\n");
26         for (i = 0; i < size * 5; i++)
27             printf("%d", recv[i]);
28     }
29     printf("\n");
30 }

```

- 运行结果

```

[student74@kunpeng192 ~]$ ./mpi.sh gather.c 3
-----
20 21 22 23 24

-----
-----
10 11 12 13 14
0 1 2 3 4

I'm root process, and the data that i received is:
0123410111213142021222324

real    0m0.007s
user    0m0.010s
sys     0m0.003s

```

- 结果分析

`MPI_gather`: 多对一收集各个进程的消息

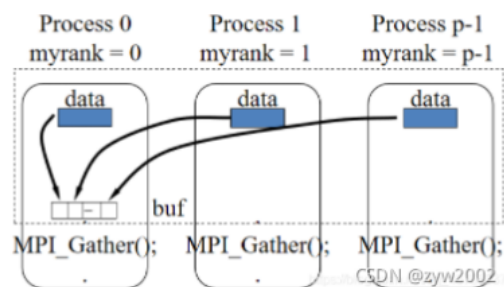
```

1  int MPI_Gather (
2      void *sendbuf,
3      int sendcnt,
4      MPI_Datatype sendtype,
5      void *recvbuf,
6      int recvcount,
7      MPI_Datatype recvtype,
8      int root,
9      MPI_Comm comm )

```

Root进程从n个进程的每一个接收各个进程(包括它自己)的消息. 这n个消息的连接按序号rank进行, 存放在Root进程的接收缓冲中。

在本例中, 开始通信前, 每个进程的send区域的值分别为0~4, 10~14, 20~24。在通信结束后跟进程的值分别为0~4、10~14、20~24, 即所有的进程把自己的消息发送给根进程的接收区。



### 3.3 scatter

- 实验代码

```

1  #include "mpi.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #define N 2
5  int main(int argc, char *argv[]) {
6      int size, rank;
7      int* send;
8      int* recv;
9      int i = 0;
10     int j = 0;
11     MPI_Init(&argc, &argv);

```



```

12 MPI_Comm_size(MPI_COMM_WORLD, &size);
13 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14 recv = (int*)malloc(N * sizeof(int));
15 for (; j < N; j++)
16     recv[j] = 0;
17 if (rank == 0) {
18     send = (int*)malloc(size * N * sizeof(int));
19     for (; i < size * N; i++) {
20         send[i] = i;
21     }
22 }
23 printf("-----\nrank=%d\n", rank);
24 for (j = 0; j < N; j++) {
25     printf("传输前recv: recv_buffer[%d]=%d\n", j, recv[j]);
26 }
27 printf("-----\n");
28 MPI_Scatter(send, N, MPI_INT, recv, N, MPI_INT, 0, MPI_COMM_WORLD);
29
30 printf("-----\nrank=%d\n", rank);
31 for (j = 0; j < N; j++)
32 {
33     printf("传输后recv: recv_buffer[%d]=%d\n", j, recv[j]);
34 }
35 printf("-----\n");
36 MPI_Finalize();
37 return 0;
38 }

```

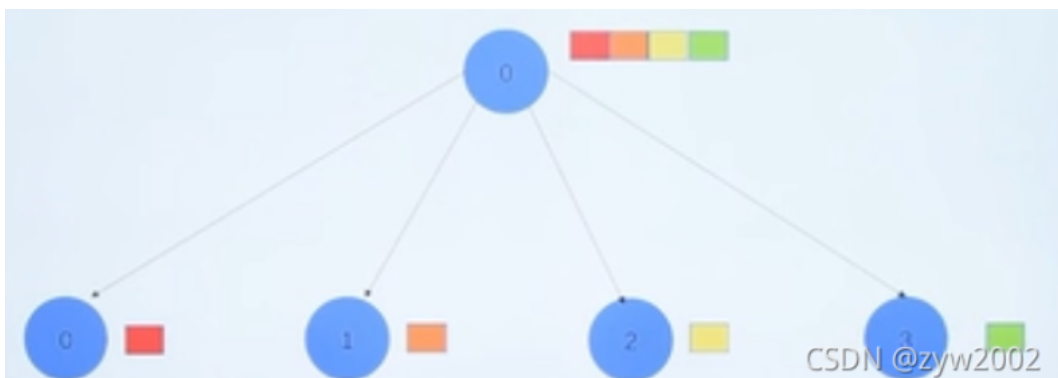
- 实验结果

```
[student74@kunpeng192 ~]$ ./mpi.sh scatter.c 3
-----
rank=2
传输前recv: recv_buffer[0]=0
传输前recv: recv_buffer[1]=0
-----
rank=1
传输前recv: recv_buffer[0]=0
传输前recv: recv_buffer[1]=0
-----
rank=0
传输前recv: recv_buffer[0]=0
传输前recv: recv_buffer[1]=0
-----
rank=0
传输后recv:recv_buffer[0]=0
传输后recv:recv_buffer[1]=1
-----
rank=1
传输后recv:recv_buffer[0]=2
传输后recv:recv_buffer[1]=3
-----
rank=2
传输后recv:recv_buffer[0]=4
传输后recv:recv_buffer[1]=5
-----
real    0m0.007s
user    0m0.005s
sys     0m0.009s
```

### • 结果分析

**MPI\_Scatter**:把指定根进程中的数据分散发送给组中的所有进程，包括自己

```
1 MPI_Scatter(*sendbuf, sendcnt, sendtype, *recvbuf, recvcnt, recvtype, root, comm)
```



在本例中，通信前各个进程的接收缓冲区的值均为0，root进程的接收缓冲区中的值分别为0~5，scatter之后，每个进程的接收区值分别为01, 23, 45，说明数据分散成功

## 3.4 reduce

### • 实验代码

```
1 #include "mpi.h"
2 #include <stdio.h>
3 #include <stdlib.h>
```

```

4
5 int main(int argc, char *argv[]) {
6     int myid, nprocs;
7     int i;
8     int send[3];
9     int recv[3];
10    MPI_Init(&argc, &argv);
11    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
12    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
13    printf("-----\n");
14    printf("myrank=%d\n", myid);
15    for (i = 0; i < 3; i++) {
16        send[i] = myid + i;
17        printf("send[i]=%d", send[i]);
18    }
19    printf("\n");
20    MPI_Reduce(send, recv, 3, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
21    if (myid == 0) {
22        for (i = 0; i < 3; i++)
23            printf("recv buf -sum:%d\n", recv[i]);
24    }
25    MPI_Finalize();
26    return 0;
27 }

```

- 运行结果

```

[student74@kunpeng192 ~]$ ./mpi.sh reduce.c 3
-----
myrank=2
send[i]=2send[i]=3send[i]=4
-----
myrank=0
-----
myrank=1
send[i]=1send[i]=2send[i]=3
send[i]=0send[i]=1send[i]=2
recv buf -sum:3
recv buf -sum:6
recv buf -sum:9

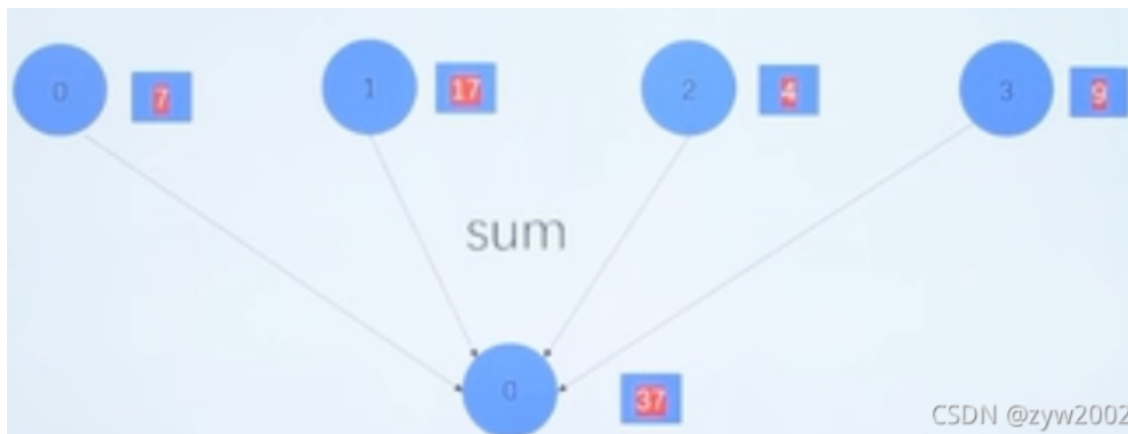
real    0m0.007s
user    0m0.013s
sys     0m0.000s

```

- 结果分析

**MPI\_Reduce**:在组内的所有进程中执行一个规约操作，并把结果存放在一个进程中。

```
1 MPI_Reduce(*sendbuf, *recvbuf, count, datatype, op, root, comm)
```



初始时，进程0，1，2的发送缓冲区中的值分别为0~2，1~3,2~4 在进行规约操作时，各个进程缓冲区中对应位置的值相加，其结果分别为3，6，9。

## Task3: Nonblocking send/receive\*

### • 实验代码

```

1  #include "mpi.h"
2  #include <stdio.h>
3  int main(int argc, char *argv[]) {
4      int numtasks, rank, dest, source, tag=1234;
5      char inmsg[]="xxxxx", outmsg[]="HELLO";
6      MPI_Status stats[2];
7      MPI_Request reqs[2];
8      MPI_Init(NULL, NULL);
9      MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
10     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11     if(rank==0) {
12         dest=1;
13         MPI_Isend(&outmsg, 5, MPI_CHAR, dest, tag, MPI_COMM_WORLD, &reqs[0]);
14         printf("Task %d: Send %s while inmsg=%s \n", rank, outmsg, inmsg);
15         fflush(stdout);
16         MPI_Wait(&reqs[0], &stats[0]);
17         printf("Task %d: Send %s while inmsg=%s reqs[0]=%d\n", rank, outmsg, inmsg, reqs[0]);
18         fflush(stdout);
19     }
20     else if(rank==1) {
21         source=0;
22         MPI_Irecv(&inmsg, 5, MPI_CHAR, source, tag, MPI_COMM_WORLD, &reqs[1]);
23         printf("Task %d: Received %s \n", rank, inmsg);
24         fflush(stdout);
25         MPI_Wait(&reqs[1], &stats[1]);
26         printf("Task %d: Received %s reqs[1]=%d \n", rank, inmsg, reqs[1]);
27         fflush(stdout);
28     }
29     MPI_Finalize();
30     return 0;
31 }

```

### • 实验结果

个人电脑：

C:\Windows\System32\cmd.exe

```
Task 0:Send HELLO while inmsg=xxxxx
Task 1:Received xxxxx
Task 0:Send HELLO while inmsg=xxxxx reqs[0]=738197504
Task 1:Received HELLO reqs[1]=738197504
请按任意键继续. . .
```

服务器:

```
[student74@kunpeng192 ~]$ ./mpi.sh nonblock.c 2
Task 0:Send HELLO while inmsg=xxxxx
Task 0:Send HELLO while inmsg=xxxxx reqs[0]=738197504
Task 1:Received xxxxx
Task 1:Received HELLO reqs[1]=738197504

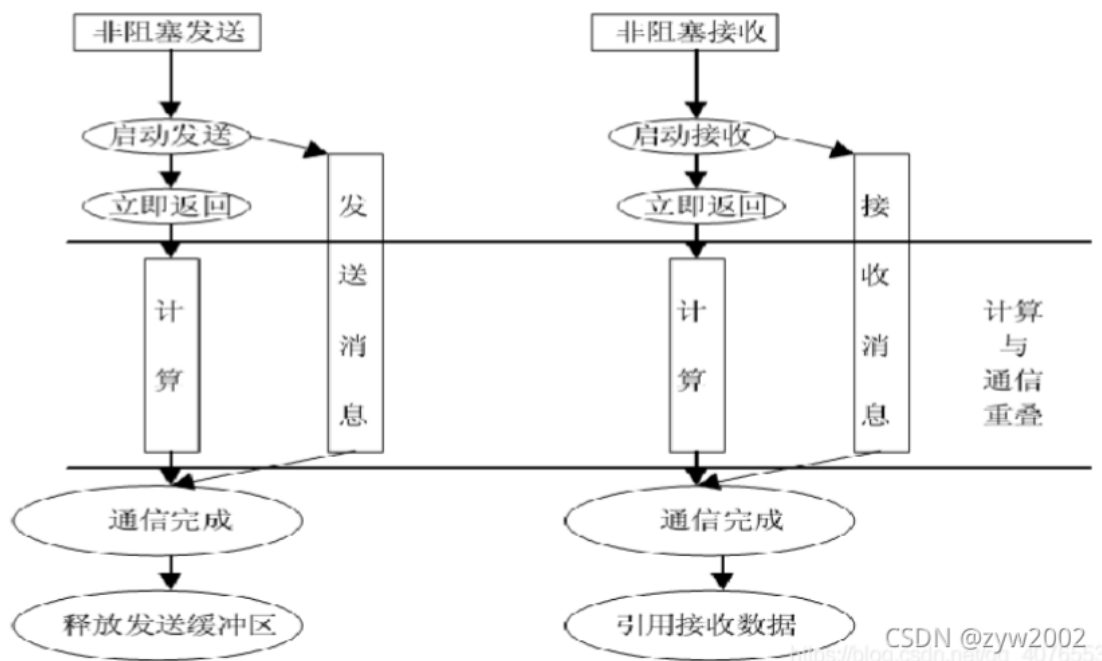
real    0m0.007s
user    0m0.010s
sys     0m0.000s
```

## • 结果分析

阻塞通信和非阻塞通信的区别:

**阻塞通信:** 进程自己要等待指定的操作实际完成, 或者所涉及的数据被MPI系统安全备份后, 函数才返回后才能进行下一步的操作。

**非阻塞通信:** 通信函数总是立即返回, 实际操作由MPI后台进行, 需要调用其它函数来查询通信是否完成, 通信与计算可重叠。



因为阻塞通信时保证了数据的安全性, 所以通信还是返回后, 其他的函数或者语句能够对这些数据资源直接访问。

但是非阻塞通信函数立即返回, 不能保证数据已经被传送出去或者被备份或者已经读入使用, 所以即使你没有阻塞, 后面的语句可以继续执行, 如果你要操纵上面所说的数据, 将会导致发送或接收产生错误。所以对于非阻塞操作, 要先调用等待 `MPI_Wait()` 或测试 `MPI_Test()` 函数来结束或判断该请求, 然后再向缓冲区中写入新内容或读取新内容。

## 非阻塞函数调用

发送语句的前缀由 `MPI_` 改为 `MPI_I_`, `I` 指的是 `immediate`, 即可改为非阻塞, 否则是阻塞。

`MPI_Isend(*buf, count, datatype, destination, tag, comm, MPI_Request *request)`: 非阻塞发送

`MPI_Irecv (*buf, count, datatype, source, tag, comm, MPI_Request*request)` : 非阻塞接收

`MPI_Wait(MPI_Request *request, MPI_Status *status)` : 通信检查, 必须等待指定的通信请求完成后才能返回, 成功返回时, status 中包含关于所完成的通信的消息, 相应的通信请求被释放, 即 request 被置成 MPI\_REQUEST\_NULL。

在本例中, 一共有两个进程, 0号进程进行非阻塞发送, 1号进程非阻塞式接收。首先发送方发送通信请求并输出“Task0: Send..”, 然后发送函数立刻返回, 此时接收方未接受到请求 输出“Task1:Received”。过一段时间后, 接收方接收到请求,此时发送方的wait函数才可以返回, 然后发送方输出“Task 0:Send HELLO”,request [0]被置成738197504最后接收方接收到信息后wait函数返回, 输出“Task1: Received HELLO”,request 被置成738197504

## Task4: 计算Pi

### 4.1 MPI\_Bcast 课堂案例\*

- 实验代码

```
1  #include <stdio.h>
2  #include <mpi.h>
3  #define N 100000
4  int main() {
5      int myid, numprocs, i, n;
6      double mypi, pi, h, sum, x, startTime, endTime;
7      n = N;
8      MPI_Init(NULL, NULL);
9      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
10     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
11     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
12     h = 1.0 / N;
13     sum = 0.0;
14     startTime = MPI_Wtime();
15     for (i = myid + 1; i <= N; i += numprocs) {
16         x = h * ((double)i - 0.5);
17         sum += (4.0 / (1.0 + x * x));
18     }
19     mypi = h * sum;
20     MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
21     if (myid == 0) {
22         endTime = MPI_Wtime();
23         printf("pi is approximately %.16f\n", pi);
24         printf("Time is :%.16f\n", endTime - startTime);
25         fflush(stdout);
26     }
27     MPI_Finalize();
28 }
```

- 实验结果

服务器结果:

```
[student74@kunpeng192 ~]$ ./mpi.sh pi_bcast.c 4
pi is approximately 3.1415926535981167
Time is :0.0002202987670898

real    0m0.008s
user    0m0.016s
sys     0m0.002s
```

个人电脑:

```
C:\Windows\System32\cmd.exe
pi is approximately 3.1415926535981167
Time is :0.0003833999999188
请按任意键继续. . .
```

- 结果分析

首先从性能上来说，华为服务器的运行速度约是本地电脑的2倍，但就精度而言两者无差别。

本代码首先通过广播机制，将n发送给通信域的每一个进程，然后每个进程i计算n被进程数num相除余数为i次的迭代，最后通过reduce的归一操作将所有的和累加在一起得到pi的值。

## 4.2 点对点通讯

- 实验代码

```
1  #include<stdio.h>
2  #include<math.h>
3  #include "mpi.h"
4  double func(double xi)
5  {
6      return (4.0 / (1.0 + xi*xi));
7  }
8  int main(int argc,char* argv[])
9  {
10     int n=1000000000,myid,numprocs,i;
11     double pi,h,xi,res,startTime,endTime;
12     pi=0.0;
13     h = 1.0/(double)n;
14     res = 0.0;
15     MPI_Init(&argc,&argv);
16     MPI_Status status;
17     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
18     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
19     MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
20     startTime=MPI_Wtime();
21     if(myid!=0)
22     {
23         for(int i=myid;i<=n;i+=(numprocs-1))
24         {
25             xi = h * ((double)i - 0.5);
26             res += func(xi);
27         }
28         res = h * res;
29         MPI_Send(&res, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD);
30     }
31     else
32     {for(i=1;i<numprocs;i++)
33     {
34         MPI_Recv(&res,1,MPI_DOUBLE,i,99,MPI_COMM_WORLD,&status);
35         pi=pi+res;}
36         endTime = MPI_Wtime();
37         printf("\nPI is %f\nTime is : %f\n",pi,endTime - startTime);}
38     MPI_Finalize();
39     return 0;
40 }
```

- 实验结果

华为服务器：

```
[student74@kunpeng192 ~]$ ./mpi.sh pi_p2p.c 4

PI is 3.141593
Time is : 2.363835

real    0m2.372s
user    0m9.436s
sys     0m0.032s
```

```
[student74@kunpeng192 ~]$ ./mpi.sh pi_p2p.c 120

PI is 3.141593
Time is : 0.203596

real    0m1.364s
user    1m8.812s
sys     0m9.232s
```

进程数越多，运算的速度就越快。

## 4.3 broadcast

- 实验代码

```
1  #include "mpi.h"
2  #include <stdio.h>
3  #include <math.h>
4  double f( double );
5  double f( double a )
6  {
7      return (4.0 / (1.0 + a*a));
8  }
9  int main( int argc, char *argv[])
10 { int done = 0, n, myid, numprocs, i;
11     double PI25DT = 3.141592653589793238462643;
12     double mypi, pi, h, sum, x;
13     double startwtime = 0.0, endwtime;
14     int namelen;
15     char processor_name[MPI_MAX_PROCESSOR_NAME];
16     MPI_Init(&argc,&argv);
17     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
18     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
19     MPI_Get_processor_name(processor_name,&namelen);
20     fprintf(stderr,"Process %d on %s\n", myid, processor_name);
21     n = 1000000000;
22     while (!done)
23     {
24         if (myid == 0)
25         {
26             startwtime = MPI_Wtime();
27         }
28         MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
29         if (n == 0)
30             done = 1;
31         else {
```



```

32     h = 1.0 / (double) n;
33     sum = 0.0;
34     for (i = myid + 1; i <= n; i += numprocs)
35     {
36         x = h * ((double)i - 0.5);
37         sum += f(x);
38     }
39     mypi = h * sum;
40     MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
41     done=1; }
42 }
43 if (myid == 0) {
44     printf("pi is approximately %.16f, Error is %.16f\n", pi,
45     fabs(pi - PI25DT));
46     endwtime = MPI_Wtime();
47     printf("wall clock time = %f\n",
48     endwtime-startwtime);
49 }
50 MPI_Finalize();
51 return 0;
52 }

```

### • 实验结果

华为服务器结果：

n=4

```

[student74@kunpeng192 ~]$ ./mpi.sh pi_broadcast.c 4
Process 0 on kunpeng192
Process 3 on kunpeng192
Process 1 on kunpeng192
Process 2 on kunpeng192
pi is approximately 3.1415926535897682, Error is 0.00000000000000249
wall clock time = 1.745385

real    0m1.753s
user    0m6.986s
sys     0m0.009s

```

n=120

```

pi is approximately 3.1415926535897953, Error is 0.00000000000000022
wall clock time = 0.620832

real    0m1.767s
user    1m31.677s
sys     0m9.944s

```

个人电脑结果：

C:\Windows\System32\cmd.exe

```

PI is 3.141593
Time is : 7.175711
请按任意键继续. . .

```

在同等进程数的情况下，华为服务器的速度是个人电脑的3~4倍，同时随着进程数的增加，误差减少，速度提升。

## 4.4 gather

- 实验代码

```
1  #include<stdio.h>
2  #include"mpi.h"
3  #include <stdlib.h>
4  #include<stddef.h>
5  double func(double xi)
6  {
7      return (4.0 / (1.0 + xi*xi));
8  }
9  int main(int argc,char *argv[])
10 {
11     int n=1000000000,myid,numprocs,root,i,sendnum;
12     double pi,h,xi,res,startTime,endTime,*recvbuf;
13     MPI_Init(&argc,&argv);
14     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
15     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
16     MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
17     root = 0;
18     pi=0.0;
19     h = 1.0/(double)n;
20     res = 0.0;
21     sendnum = numprocs;
22     startTime=MPI_Wtime();
23     for(int i=myid;i<n;i+=numprocs)
24     {
25         xi = h * ((double)i -0.5);
26         res += func(xi);
27     }
28     res = h * res;
29     if(myid==root)
30     {
31         recvbuf = (double *)malloc( sendnum * sizeof(double));
32         MPI_Gather(&res,1,MPI_DOUBLE,recvbuf,1,MPI_DOUBLE,root,MPI_COMM_WORLD);
33         MPI_Barrier(MPI_COMM_WORLD);
34         if (myid==root)
35             {for(i=0;i<numprocs;i++){
36 pi=pi+recvbuf[i];}
37             free(recvbuf);
38             printf("pi is approximately %f\n", pi);
39             endTime = MPI_Wtime();
40             printf("wall clock time = %f\n",
41 endTime-startTime);
42             }
43         MPI_Finalize();
44         return 0;
45     }
```

- 实验代码

服务器:

```
[student74@kunpeng192 ~]$ ./mpi.sh pi_gather.c 4
pi is approximately 3.141593
wall clock time = 1.778730

real    0m1.787s
user    0m7.116s
sys     0m0.014s
```

```
[student74@kunpeng192 ~]$ ./mpi.sh pi_gather.c 120
pi is approximately 3.141593
wall clock time = 0.263331

real    0m1.440s
user    1m14.951s
sys     0m9.040s
```

个人电脑:

```
C:\Windows\System32\cmd.exe
PI is 3.141593
Time is : 7.583272
请按任意键继续. . .
```

#### • 结果分析

该代码是先通过广播机制，将n发送给各个进程。然后各个进程分别分担一些迭代过程。再通过MPI\_Gather函数将每个进程的计算结果发送到根进程中，最后设置一个barrier,当所有进程的结果都发送到根进程后，将根进程中收集的所有迭代结果循环累加得到最终的pi值。

## 4.5 reduce

#### • 实验代码

```
1  #include<stdio.h>
2  #include<math.h>
3  #include "mpi.h"
4  double func(double xi)
5  {
6      return (4.0 / (1.0 + xi*xi));
7  }
8  int main(int argc,char* argv[])
9  {
10     int n=1000000000,myid,numprocs;
11     double pi,h,xi,res,startTime,endTime;
12     MPI_Init(&argc,&argv);
13     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
14     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
15     MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
16     if(myid==0)
17     {
18         startTime=MPI_Wtime();
19     }
20     h = 1.0/(double)n;
21     res = 0.0;
22     for(int i=myid;i<n;i+=numprocs)
23     {
24         xi = h * ((double)i - 0.5);
25         res += func(xi);
```

```

26     }
27     res = h * res;
28     MPI_Reduce(&res,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
29     if(myid==0)
30     {
31         endTime = MPI_Wtime();
32         printf("\nPI is %f\nTime is : %f\n",pi,endTime - startTime);
33     }
34     MPI_Finalize();
35     return 0;
36 }

```

- 实验结果

```

[student74@kunpeng192 ~]$ ./mpi.sh pi_reduce.c 4

PI is 3.141593
Time is : 1.779796

real    0m1.788s
user    0m7.134s
sys     0m0.000s

```

```

[student74@kunpeng192 ~]$ ./mpi.sh pi_reduce.c 120

PI is 3.141593
Time is : 0.327093

real    0m1.580s
user    1m21.075s
sys     0m8.974s

```

- 结果分析

使用reduce的作用是将不同进程的迭代结果进行规约操作，累加得到最终的pi值。

## Task5: openMp和mpi 混合编程 \*

- 实验代码

```

1  #include "stdio.h"
2  #include "mpi.h"
3  #include "omp.h"
4  #include "math.h"
5  #define NUM_THREADS 8
6  long int n=10000000;
7  int main(int argc,char*argv[])
8  {
9      int my_rank,numprocs;
10     long int i,my_n,my_first_i,my_last_i;
11     double my_pi=0.0,pi,h,x,startTime,endTime;
12     MPI_Init(&argc,&argv);
13     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
14     MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
15     h=1.0/n;
16     my_n=n/numprocs;
17     my_first_i=my_rank*my_n;

```

```

18     my_last_i=my_first_i+my_n;
19     omp_set_num_threads(NUM_THREADS);
20     startTime=MPI_Wtime();
21     for(i=my_first_i;i<my_last_i;i++)
22     {
23         x=(i+0.5)*h;
24         my_pi=my_pi+4.0/(1.0+x*x);
25     }
26     MPI_Reduce(&my_pi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
27     if(my_rank==0)
28     {
29         printf("Approximation of pi:%15.13f\n",pi*h);
30         endTime = MPI_Wtime();
31         printf("Time is : %f\n",endTime - startTime);
32     }
33     MPI_Finalize();
34     return 0;
35 }
36 // mpicc pi_mpi_omp.c -o pi_mpi_omp -fopenmp
37 // mpirun ./pi_mpi_omp -up 4

```

## • 实验结果

服务器结果：

```

[student74@kunpeng192 ~]$ mpicc pi_mpi_omp.c -o pi_mpi_omp -fopenmp
[student74@kunpeng192 ~]$ mpirun ./pi_mpi_omp -up 4
Approximation of pi:3.1415926535898
Time is : 0.009001

```

个人电脑结果：

```

C:\Windows\System32\cmd.exe
Approximation of pi:3.1415926535897
Time is : 0.030695
请按任意键继续. . .

```

## • 结果分析

当设置4个进程时，可以看出华为服务器的速度是个人电脑速度的3倍多。同时纵向比较，mpi和openmp混合使用后，其速度要比单独使用其中一个要快很多，大概有 $10^3$ 个数量级。

MPI主要实现的是多主机联网协作进行并行计算，OpenMP更适合单台计算机（多核）共享内存结构上的并行计算。MPI针对的是进程，openMP针对的是线程。通俗的说就是，在只使用MPI进行计算时，每个进程会分配一些迭代，即每个进程内部仍然使用了for循环。当使用了openmp后，每个进程for循环的迭代有不同的线程来分担，从而实现进程间核线程间的并行计算，速度变的更快了。

## Task6 : 多节点测试\*

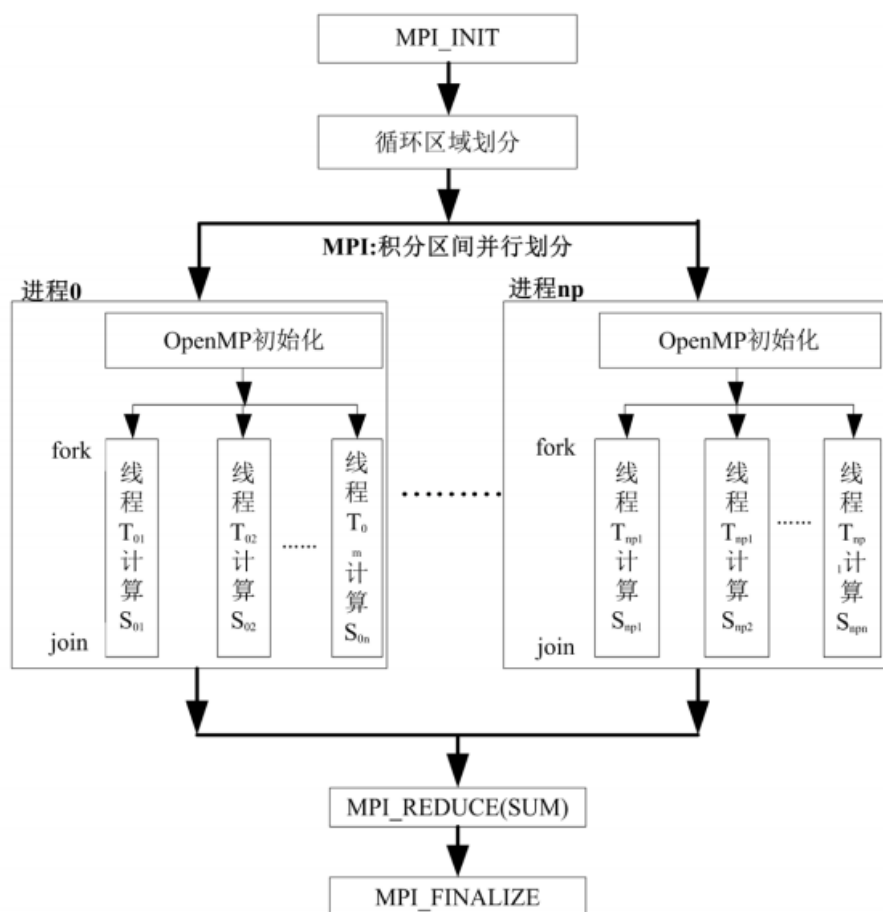
### 6.1算法设计

算法设计

求解思路主要围绕着循环计算各部分梯形面积展开，其算法设计也主要针对循环过程的二级并行划分开展。具体如下：

(1) 节点间任务划分 采用均匀划分或交叉划分方法，将计算区间S划分为np个小区间  $s_i, S = \bigcup s_i, i = 1, 2, \dots, np$ , 若采用交叉划分，每个区间所含的循环次数尽量为节点内核心数的整数倍。

(2) 节点内任务划分 在每个计算节点内调用OpenMP for并行制导语句，将所分配的循环分配给不同的线程，期间需要调整每个线程分配的循环次数，尽可能做到负载均衡，求解流程如下所示。



## 6.2 程序代码

编写代码 `pi_multinode.c`，实现mpi和openmp混合编程计算pi的值。并可以显示进程数和进程所运行的节点。

```

1  #include "mpi.h"
2  #include <stdio.h>
3  #include <math.h>
4  #include "omp.h"
5  double f(double);
6
7  double f(double a)
8  {
9      return (4.0 / (1.0 + a*a));
10 }
11
12 int main(int argc, char *argv[])
13 {
14     int    n, myid, numprocs, i;
15     double PI25DT = 3.141592653589793238462643;
16     double mypi, pi, h, sum, x;
17     double startwtime = 0.0, endwtime;
18     int    namelen;
19     char    processor_name[MPI_MAX_PROCESSOR_NAME];
20
21     MPI_Init(&argc, &argv);
  
```

```

22 MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
23 MPI_Comm_rank(MPI_COMM_WORLD,&myid);
24 MPI_Get_processor_name(processor_name,&namelen);
25
26 fprintf(stdout,"Process %d of %d is on %s\n",
27         myid, numprocs, processor_name);
28 fflush(stdout);
29
30 n = 10000;          /* default # of rectangles */
31 if (myid == 0)
32     starttime = MPI_Wtime();
33
34 MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
35
36 h = 1.0 / (double) n;
37 sum = 0.0;
38 /* A slightly better approach starts from large i and works back */
39 #pragma omp parallel for reduction(+:sum)private(x,i)
40 for (i = myid + 1; i <= n; i += numprocs)
41 {
42     x = h * ((double)i - 0.5);
43     sum += f(x);
44 }
45 mypi = h * sum;
46
47 MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
48
49 if (myid == 0) {
50     endwtime = MPI_Wtime();
51     printf("pi is approximately %.16f, Error is %.16f\n",
52           pi, fabs(pi - PI25DT));
53     printf("wall clock time = %f\n", endwtime-startwtime);
54     fflush(stdout);
55 }
56
57 MPI_Finalize();
58 return 0;
59 }

```

### 6.3 单节点测试

1. 通过xftp将代码上传
2. 在xshell 中连接两个节点 172.31.46.191 172.31.46.192
3. 然后kungpeng192的对话框窗口键入命令 `mpicc pi_multinode.c -o pi_multinode -fopenmp`, 生成名为pi\_multinode的可执行文件。
4. 然后kungpeng192的对话框窗口键入命令 `mpiexec -n 4 ./multinode`

出现如下图所示的结果，一共有四个进程，这四个进程都是运行在kungpeng192上的，说明单节点运行成功。

```

[student74@kungpeng192 ~]# mpiexec -n 4 ./pi_multinode
Process 2 of 4 is on kungpeng192
Process 3 of 4 is on kungpeng192
Process 0 of 4 is on kungpeng192
Process 1 of 4 is on kungpeng192
pi is approximately 3.1415926544231265, Error is 0.0000000008333334
wall clock time = 0.006114

```

### 6.4.1 host配置

```
1 172.31.46.191 kunpeng191
2 172.31.46.192 kunpeng192
```

[illegible]

## 1. kengpeng192生成公钥

```
1 cd ~/.ssh/ # 若没有该目录, 请先执行一次ssh localhost
2 ssh-keygen -t rsa # 会有提示, 接着连接3次回车
3 cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys #将自己的公钥追加到
authorized_keys里
```

## 2. kengpeng191生成公钥

kungpeng191的对话框口键入命令:



```
1 | cd ~/.ssh/ # 若没有该目录, 请先执行一次ssh localhost
2 | ssh-keygen -t rsa # 会有提示, 接着连按3次回车
3 | cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys #将自己的公钥追加到authorized_keys里
```

### 3. kungpeng191产生的公钥发送给kungpeng192

kungpeng191的对话框键入命令:

```
1 | cd ~/.ssh/ # 若没有该目录, 请先执行一次ssh localhost
2 | ssh-keygen -t rsa # 会有提示, 接着连按3次回车
3 | scp ./id_rsa.pub student74@kungpeng192:~/.ssh/kungpeng191_id_rsa.pub # 191复制自己的公钥给192
```

### 3. kungpeng192将kungpeng191发送过来的公钥追加到authorized\_keys

kungpeng192的对话框键入命令:

```
1 | cat ~/.ssh/kungpeng191_id_rsa.pub >> ~/.ssh/authorized_keys
```

### 4. kungpeng192修改文件权限并将authorized\_keys文件发送给kungpeng191

```
1 | chmod 600 ~/.ssh/authorized_keys #可能有时不修改也不影响无密登录但还是建议修改
2 | scp ./authorized_keys student74@kungpeng191:~/.ssh/authorized_keys
```

### 5. 删除kungpeng192下的kungpeng191\_id\_rsa.pub

kungpeng192的对话框键入命令:

```
1 | rm ~/.ssh/kungpeng191_id_rsa.pub # (好像这个删不删都可以。。)
```

### 6. 验证免密登录

```
1 | ssh kungpeng192 # kungpeng191的对话框键入
2 | ssh kungpeng191 # kungpeng192的对话框键入
```

出现如下图, 说明免密登录成功。

```
1 kunpeng191 x 2 kunpeng192 x +
[student74@kunpeng191 ~]$ ssh kunpeng192

Authorized users only. All activities may be monitored and reported.
Last login: Fri Nov 26 20:13:43 2021 from 172.31.46.191

Welcome to 4.19.90-2110.3.0.0116.oe1.aarch64

System information as of time:  Fri Nov 26 21:08:41 CST 2021

System load:      0.84
Processes:        741
Memory used:      48.5%
Swap used:        0.0%
Usage On:         10%
IP address:       172.31.46.192
Users online:     13

[student74@kunpeng192 ~]$
```

```
1 kunpeng191 x 2 kunpeng192 x +
Authorized users only. All activities may be monitored and reported.

Authorized users only. All activities may be monitored and reported.
Last login: Fri Nov 26 20:13:49 2021 from 172.31.46.192

Welcome to 4.19.90-2110.3.0.0116.oe1.aarch64

System information as of time:  Fri Nov 26 21:09:28 CST 2021

System load:      0.00
Processes:        679
Memory used:      1.2%
Swap used:        3.9%
Usage On:         6%
IP address:       172.31.46.191
Users online:     8

[student74@kunpeng191 ~]$
```

## 6.5 多节点运行

1. 然后将可执行文件cpi发送给kunpeng191

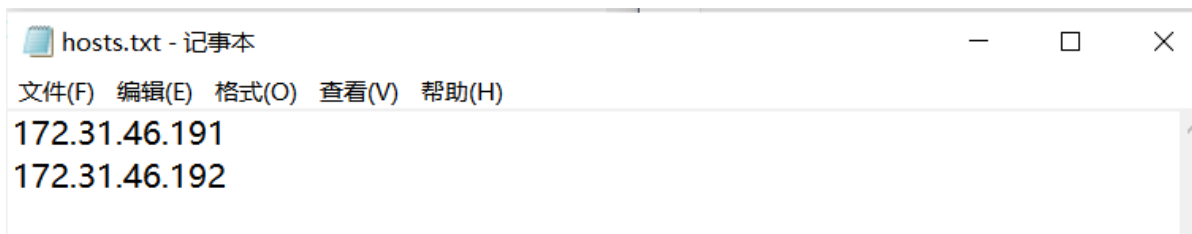
在kunpeng192中键入命令：

```
1 | scp -r /home/student74/pi_multinode student74@kunpeng191:/home/student74/

[student74@kunpeng192 ~]$ scp -r /home/student74/pi_multinode student74@kunpeng1
91:/home/student74/

Authorized users only. All activities may be monitored and reported.
pi_multinode                               100% 70KB 50.9MB/s 00:00
```

2. 编写hosts.txt文件，并用xftp上传hosts.txt文件至kunpeng192



名称	大小	类型
..		
bcast	70KB	文件
bcast.c	664 Bytes	C Source
cpi	70KB	文件
cpi.c	2KB	C Source
gather	70KB	文件
gather.c	735 Bytes	C Source
hello	70KB	文件
hosts.txt	30 Bytes	文本文档

3. 将hosts.txt文件发送给kunpeng191

在kunpeng192中键入命令：

```
1 | scp -r /home/student74/hosts.txt student74@kunpeng191:/home/student74/
```

```
[student74@kunpeng192 ~]$ scp -r /home/student74/hosts.txt student74@kunpeng191:/home/student74/
Authorized users only. All activities may be monitored and reported.
hosts.txt                                100% 30   162.2KB/s   00:00
```

出现如上图说明传输成功

4. 然后在kunpeng191或者kunpeng 192下执行如下语句

```
1 | mpiexec -f hosts.txt -n 4 ./pi_multinode
2 | mpiexec -n 1 ./pi_multinode
```

结果中既有kunpeng191又有192 说明多节点运行成功。而且在多节点上运行时，速度大概是单节点上速度的3倍。

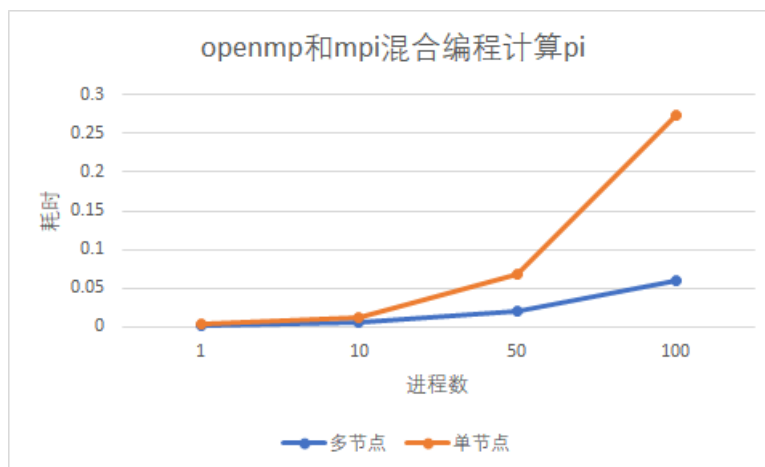
```
[student74@kunpeng192 ~]$ mpiexec -f hosts.txt -n 4 ./pi_multinode
Authorized users only. All activities may be monitored and reported.
Process 0 of 4 is on kunpeng191
Process 2 of 4 is on kunpeng191
Process 1 of 4 is on kunpeng192
Process 3 of 4 is on kunpeng192
pi is approximately 3.1415926544231261, Error is 0.0000000008333330
wall clock time = 0.002486
```

```
[student74@kunpeng191 ~]$ mpiexec -f hosts.txt -n 4 ./pi_multinode
Process 0 of 4 is on kunpeng191
Process 2 of 4 is on kunpeng191
Process 1 of 4 is on kunpeng192
Process 3 of 4 is on kunpeng192
pi is approximately 3.1415926544231265, Error is 0.0000000008333334
wall clock time = 0.002397
```

然后，分别针对单节点和多节点在不同的进程数下(分别取1, 10, 50, 100) 重复5次计算pi值，并得到其每种进程数下耗时的平均值，如下表所示：

multi	1	2	3	4	5	avg	error
1	<b>0.00177</b>	0.001661	0.001703	0.001881	0.001812	0.001765	8.33E-10
10	0.005317	0.008291	0.004871	0.005518	0.005188	0.005837	8.33E-10
50	0.020077	0.020126	0.019739	0.020044	0.019845	0.019966	8.33E-10
100	0.075079	0.063212	0.066827	0.048958	0.048966	0.060608	8.33E-10
single							
1	0.001817	0.001897	0.001835	0.001625	0.001685	0.001772	8.33E-10
10	0.006991	0.007054	0.006874	0.00708	0.007342	0.007068	8.33E-10
50	0.049091	0.050449	0.048606	0.049279	0.047565	0.048998	8.33E-10
100	0.19531	0.271491	0.193001	0.228831	0.174075	0.212542	8.33E-10

然后根据如上的实验结果，做出如下折线图：

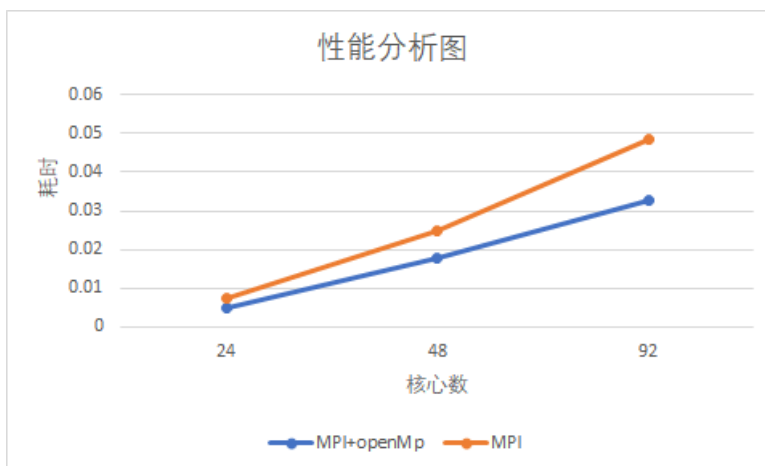


通过观察上述的图表，可以看出：

- 在进程数相同的情况下，单节点运算更耗时
- 无论是单节点还是多节点，随着进程数的增加，其消耗时间也随之增加
- 随着进程数的增加，单节点耗时增长的速度更快

## 6.6 性能分析

分别设置核心数为24, 48, 92分别测试其耗时（对于MPI + OpenMP程序，核心数 = 节点数 \* 单节点核心数；MPI程序，核心数 = 进程数。）



通过对比分析发现随着核心数的增加，其耗时都会增加。但是MPI+OpenMP混合编程时，效率要比只进行MPI时要高。

通过查阅资料发现MPI + OpenMP和MPI并程序执行过程中时间的分布按照：并行计算时间、通信时间两部分进行统计，随着计算规模的增加，MPI程序并行计算部分花费时间由11.95 s减少为0.64 s，降低近19倍，占总计算时间的百分比由91.0%减小为8.6%；进程间通信消耗的时间由1.09 s增加至6.77 s，提高6.2倍，占总计算时间的9.0%提高至91.4%。进程间通信开销远高于并行计算所花费的时间，极大的制约了并行效率的提高。

**Table 3.** Parallel computing distribute time of MPI parallel program  
**表 3.** MPI 并程序计算时间分配表

时间分布 \ 进程数	24	48	96	192	384	768
并行计算时间	11.95	6.10	3.14	2.17	1.43	0.64
进程通信时间	1.09	1.29	1.70	2.01	2.65	6.77
合计	13.14	7.39	4.84	4.18	4.08	7.41

备注：单节点启动 24 个进程

在同样的计算规模下，MPI + OpenMP程序并行部分计算花费时间由11.92 s减少为1.27秒，占总计算时间的百分比由99.7%减少为73.4%；进程通信部分消耗的时间由0.04 s增加为0.46 s，占总计算时间的百分比由0.3%提高至26.6%。采用MPI + OpenMP混合并程序方法，有效降低了进程间通信的时间开销，大大提高了程序的并行执行效率和可扩展性。

**Table 4.** Parallel computing distribute time of MPI + OpenMP parallel program  
**表 4.** MPI + OpenMP 并程序计算时间分配

时间分布 \ 进程数	1	2	4	8	16	32
并行计算时间	11.92	6.10	3.32	2.39	1.28	1.27
进程通信时间	0.04	0.15	0.14	0.16	0.21	0.46
合计	11.96	6.25	3.46	2.55	1.49	1.73

备注：单节点启动 1 个进程，24 个线程

因此可以得到如下的结论：

随着进程数的增加，MPI + OpenMP两种并行编程方式下，应用问题并行部分的计算时间均逐渐减小，但MPI + OpenMP并程序的通信消耗时间只占MPI并程序的6.8%。由此可见，MPI + OpenMP混合并程序在很好地继承了两种并行编程环境的优点的同时，可以有效克服两种并行编程环境的缺点，可以有效提高并程序的效率。

## 总结

- MPI **集体通信** (collective communication) 指的是一个涉及 communicator 里面所有进程的一个方法。这节课我们会解释集体通信以及一个标准的方法

MPI 根据参与集合通信双方的进程数目，可大致分为三类

1. 一对多：Bcast, Scatter, Scatterv
2. 多对一：Gather, Gatherv, Reduce
3. 多对多：Allgather, Allgatherv, Allreduce, Reduce\_scatter, Alltoall, Alltoallv, Alltoallw, Exscan

此外，集合通信还包括一个同步操作Barrier，同步栅格，即所有的进程到达后才可以继续执行。

- 执行命令总结

```
1 //MPI
2 mpicc/mpic++ -o mpi[可执行文件名] mpi.c/mpi.cpp # 编译命令
3 mpirun/mpiexec -np [线程数] ./mpi[可执行文件] # 执行命令
4
5 //OpenMP
6 gcc/g++ -o omp omp.c/omp.cpp -fopenmp # 编译命令
7 ./omp # 执行命令
8
9 //MPI + OpenMP
10 mpicc/mpic++ -o test omp.c/omp.cpp -fopenmp # 编译命令
11 ./test [参数1] [参数2] ... # 执行命令
```

## 参考

参考博客：

- 【1】[\(16条消息\)【mpich2】图文教程：mpich2的安装、配置、测试、vs配置、命令行测试（没有使用）苍狼的博客-CSDN博客mpich2配置与测试](#)
- 【2】[\(16条消息\)如何在Window7系统中安装MPICH2 怡暘-CSDN博客](#)
- 【3】[\(16条消息\) windows下安装mpich2执剑者罗辑的博客-CSDN博客mpich2](#)
- 【4】CentOS 7.6安装OpenMPI3.1.0 <https://drugai.blog.csdn.net/article/details/106787587>