并行编程实验设计文档

实验项目名称：MPI

姓名：江家玮

班级：计科2204班

学号：22281188

自我评价：

在本次实验中，首先创建了Ubuntu的虚拟机，而后配置MPI的实验环境，在这个过程中，也遇到了许多 的报错，通过不断的搜索，也间接提高了我对Linux系统的指令的掌握程度，同时通过对MPI各个代码的调试，让我对于程序的各项进程的运行和管理有了更加清晰的认识。其中五个小实验中让我映像最深的是实验五的联机实验，因为需要建立SSH的免密链接等，方可继续运行程序，两台机子加起来可以同时运行八个进程，让我对并行程序的执行建立了清晰的认识。

成绩：

一、 实验一 MPI环境管理

✧ 运行代码

```
#include<mpi.h>
#include<stdio.h>
int main(int argc, char* argv[]) {
 int myid, numprocs;
 int namelen;
 char processor_name[MPI_MAX_PROCESSOR_NAME];
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myid);
 MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
 MPI_Get_processor_name(processor_name, &namelen);
 fprintf(stderr, "Hello World! Process %d of %d on %s\n",
 myid, numprocs, processor_name);
 MPI_Finalize();
}
```

✧ 运行结果：

```
mpi@jiawei-virtual-machine:~$ vim Lab1.c
mpi@jiawei-virtual-machine:~$ mpicc -o Lab1 Lab1.c
mpi@jiawei-virtual-machine:~$ mpirun -np 4 ./Lab1
Hello World! Process 0 of 4 on jiawei-virtual-machine
Hello World! Process 1 of 4 on jiawei-virtual-machine
Hello World! Process 2 of 4 on jiawei-virtual-machine
Hello World! Process 3 of 4 on jiawei-virtual-machine
```

✧ 前期文件配置：

```
mpi@jiawei-virtual-machine:~$ which mpicc
/home/mpi/mpich2/bin/mpicc
mpi@jiawei-virtual-machine:~$ which mpif90
/home/mpi/mpich2/bin/mpif90
```

✧ 运行结果分析：

代码成功运行，且MPI的安装与环境配置已经成功，前期检查完成。

■ **MPI基本函数的作用：**

1. MPI_Init(&argc, &argv);

是MPI程序的起始点。它初始化MPI执行环境，所有MPI的程序都必须从调用MPI_Init开始。它接收主函数的参数argc和argv，以便支持MPI实现可能会提供的命令行选项。

2. MPI_Comm_rank(MPI_COMM_WORLD, &myid);

这个函数用来获取当前进程在MPI_COMM_WORLD这个通信器中的排名或者ID。MPI_COMM_WORLD是所有MPI进程的默认通信器，每个进程都被赋予一个唯一的整数ID，称为rank。

3. MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

这个函数用来获取在指定通信器中的进程总数，在这里是MPI_COMM_WORLD。这个值表明有多少个进程参与了MPI并行计算。

4. MPI_Get_processor_name(processor_name, &namelen);

这个函数用来获取运行当前进程的物理或虚拟机器的名称。这可以帮助识别进程运行的具体节点。

5. MPI_Finalize();

这是MPI程序的结束点。它清理MPI环境，任何MPI程序必须在程序结束前调用`MPI_Finalize`。

✧ 实验结果分析：

从实验结果看，mpirun -np 4 ./Lab1 这个命令启动了一个有4个进程的MPI程序。-np 4表明运行程序的进程数为4。

- "Hello World! Process 0 of 4 on jiawei-virtual-machine"
- "Hello World! Process 1 of 4 on jiawei-virtual-machine"
- "Hello World! Process 2 of 4 on jiawei-virtual-machine"
- "Hello World! Process 3 of 4 on jiawei-virtual-machine"

每个输出行显示了一个不同的进程ID（从0到3），说明总共有4个进程。所有进程都在同一台机器上运行。Process %d of %d中的第一个%d被替换成了进程ID，第二个%d被替换成了总进程数，`%s`被替换成了机器名称。

二、 实验二 Hello World

```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
#include<mpi.h>
#include<stdio.h>
int main() {
MPI_Status status;
char string[] = "xxxxx";
int myid;
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
if (myid == 1)
MPI_Send("HELLO", 5, MPI_CHAR, 3, 1234, MPI_COMM_WORLD);
if (myid == 3) {
MPI_Recv(string, 5, MPI_CHAR, 1, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
printf("Got %s from P%d, tag %d\n",
string, status.MPI_SOURCE, status.MPI_TAG);
fflush(stdout);
}
MPI_Finalize();
}
```

✧ 运行结果

```
mpi@jiawei-virtual-machine:~$ touch Lab2.c
mpi@jiawei-virtual-machine:~$ vim Lab2.c
mpi@jiawei-virtual-machine:~$ mpicc -o Lab2 Lab2.c
mpi@jiawei-virtual-machine:~$ mpirun -np 4 ./Lab2
Got HELLO from P1, tag 1234
```

✧ 运行结果分析：

1. MPI_Send() 函数

MPI_Send()是MPI库中的标准发送消息函数，其作用是向指定的目标进程发送消息。

函数原型：int MPI_Send(void* data, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

参数解释：

- data: 发送数据的起始地址。
- count: 发送数据的元素数量。
- datatype: 数据元素的类型（例如`MPI_CHAR`）。
- dest: 目标进程的ID（rank）。
- tag: 消息的标签，用于区分不同的消息。
- comm: 使用的通信器，通常是`MPI_COMM_WORLD`，表示所有的MPI进程。

在此代码中，`MPI_Send()`被配置为当进程的ID为1时执行，它发送了一个"HELLO"字符串给ID为3的进程，消息的标签为1234。

2. MPI_Recv()函数

MPI_Recv()是MPI库中的标准接收消息函数，其作用是从指定的源进程接收消息。

函数原型：int MPI_Recv(void* data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status)

参数解释：

data: 接收数据的起始地址。

count: 最大接收数据的元素数量。

datatype: 数据元素的类型。

source: 数据源的进程ID，`MPI_ANY_SOURCE`表示可以接收任何进程的消息。

tag: 消息的标签，`MPI_ANY_TAG`表示可以接收任何标签的消息。

comm: 使用的通信器。

statu: 实际接收到的消息状态，如来源、标签等信息。

在此代码中，`MPI_Recv()`被配置为当进程的ID为3时执行，它从ID为任意的进程接收消息，并且接收任何标签的消息。

✧ 实验结果分析

运行结果显示了以下输出：

Got HELLO from P1, tag 1234

这说明进程3成功地接收了来自进程1的消息"HELLO"。`MPI_Recv()`函数中的`status`结构体被用来输出发送者的ID（通过`status.MPI_SOURCE`）和消息的标签（通过`status.MPI_TAG`）。这个实验结果表明了MPI中的点对点通信成功：进程1发送了一个字符串给进程3，进程3接收到了这个字符串并且打印出了发送者的ID和消息的标签。

三、 实验三 Nonblocking Send/Receive

```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
#include<mpi.h>
#include<stdio.h>
int main(int argc, char* argv[]) {
int numtasks, rank, dest, source, flag, tag = 1234;
char inmsg[] = "xxxxx", outmsg[] = "HELLO";
MPI_Status stats[2];
MPI_Request reqs[2];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
dest = 1;
MPI_Isend(&outmsg, 5, MPI_CHAR, dest, tag, MPI_COMM_WORLD, &reqs[0]);
printf("Task %d: Send %s \n", rank, outmsg);
fflush(stdout);
}
else if(rank == 1) {
source = 0;
MPI_Irecv(&inmsg, 5, MPI_CHAR, source, tag, MPI_COMM_WORLD, &reqs[1]);
printf("Task %d: Received %s \n", rank, inmsg);
fflush(stdout);
MPI_Wait(&reqs[1], &stats[1]);
printf("Task %d: Received %s from SOURCE %d with tag %d\n",
rank, inmsg, stats[1].MPI_SOURCE, stats[1].MPI_TAG);
fflush(stdout);
}
printf("Task %d inmsg=%s outmsg=%s reqs[%d] %d\n",
rank, inmsg, outmsg, rank, reqs[rank]);
fflush(stdout);
MPI_Finalize();
}
```

✧ 运行结果

```
mpi@jiawei-virtual-machine:~$ touch Lab3.c
mpi@jiawei-virtual-machine:~$ vim Lab3.c
mpi@jiawei-virtual-machine:~$ mpicc -o Lab3 Lab3.c
mpi@jiawei-virtual-machine:~$ mpirun -np 4 ./Lab3
Task 0: Send HELLO
Task 0 inmsg=xxxxx outmsg=HELLO reqs[0] -1409286144
Task 1: Received xxxxx
Task 1: Received HELLO from SOURCE 0 with tag 1234
Task 1 inmsg=HELLO outmsg=HELLO reqs[1] 738197504
Task 2 inmsg=xxxxx outmsg=HELLO reqs[2] 1
Task 3 inmsg=xxxxx outmsg=HELLO reqs[3] 0
```

✧ 运行结果分析

1. MPI_Isend()

函数原型：int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);

作用： 开始一个发送操作，但是不等待操作完成即返回。这允许程序继续进行计算或者进行其他通信操作。

2. MPI_Irecv()

函数原型：int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request);

作用：开始一个接收操作，但不等待操作的完成即返回。这允许程序在等待接收消息时执行其他操作。

3. MPI_Wait()

等待非阻塞操作完成的函数。

函数原型：int MPI_Wait(MPI_Request *request, MPI_Status *status);

作用：阻塞调用进程直到与特定的`MPI_Request`关联的非阻塞操作完成。

✧ 实验结果分析

- Task 0 : Send HELLO
- Task 1 Received HELLO from SOURCE 0 with tag 1234
- 其他任务显示初始字符串和请求ID。

输出表明：

- 任务0开始一个非阻塞发送操作，发送"HELLO"字符串。
- 任务1接收到来自任务0的"HELLO"消息，打印出接收到的信息，并且指明消息是来自源0且带有标签1234。
- MPI_Wait()被任务1使用来确保接收操作完成，然后打印出消息和发送者信息。
- 每个任务还打印出自己的inmsg和outmsg内容以及关联的请求对象的地址（或请求ID）。这说明MPI_Isend和MPI_Irecv都立即返回，程序继续执行，直到MPI_Wait()调用确保了通信操作的完成。

非阻塞通信对于提高并行程序的效率较高，因为它允许进行重叠计算和通信。即使发送或接收操作尚未完成，程序的执行也可以继续进行其他工作。从输出中可以看出，任务0成功发送了消息给任务1，并且任务1成功接收了该消息。非阻塞通信允许任务0和任务1在消息传递的同时继续执行其他代码，MPI_Wait确保在打印消息前接收操作已经完成。

四、 实验四 PI计算

```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
#include<mpi.h>
#include<stdio.h>
#define N 10000
int main() {
 int myid, numprocs, i, n;
 double mypi, pi, h, sum, x;
 n = N;
 MPI_Init(NULL, NULL);
 MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
 MPI_Comm_rank(MPI_COMM_WORLD, &myid);
 MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
 h = 1.0 / N;
 sum = 0.0;
 for (i = myid + 1; i <= N; i += numprocs) {
 x = h * ((double)i - 0.5);
 sum += (4.0 / (1.0 + x * x));
 }
 mypi = h * sum;
 MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
 if (myid == 0) {
 printf("pi is approximately %.16f\n", pi);
 fflush(stdout);
 }
 MPI_Finalize();
```

✧ 运行结果

```
mpi@jiawei-virtual-machine:~$ ^C
mpi@jiawei-virtual-machine:~$ mpicc -o Lab4 Lab4.c
mpi@jiawei-virtual-machine:~$ mpirun -np 4 ./Lab4
pi is approximately 3.1415926544231239
```

✧ 运行结果分析

MPI归约操作函数

MPI_Bcast

函数原型：int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);

作用：MPI_Bcast`用于从一个进程向所有其他进程发送数据。在此代码中，它被用于广播整数`n`的值（代表分割数）到所有的进程。

MPI_Reduce

函数原型：int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);

作用：MPI_Reduce`用于收集所有进程的数据，通过一个指定的操作（如求和，最大值，最小值等）将其归约为单一数据，并在根进程上存储结果。在此代码中，

它将所有进程计算的`mypi`的值相加，得到最终的π值，并将结果存储在根进程（进程0）的`pi`变量中。

✧ 实验结果分析

代码中的计算过程使用了数值积分的方法来估算π的值。每个进程计算它负责的矩形面积的一部分，然后通过MPI_Reduce操作将这些面积部分累加起来得到π的近似值。

✧ 程序输出显示：

pi is approximately 3.1415926544231239

这表示使用此MPI程序和所选择的分割数N=10000，所有的进程合作计算得到了π的一个近似值。由于π的实际值约为3.141592653589793，可以看出该近似值与实际值非常接近，说明程序是正确的。程序的精确度取决于分割数N；N值越大，近似值通常越精确。

五、 实验五 Synchronization Constructs

```
File  Edit  View  Search  Terminal  Help
#include<mpi.h>
#include<stdio.h>
#define N 10000
int main() {
int myid, numprocs, i, n, namelen;
double mypi, pi, h, sum, x;
char processor_name[MPI_MAX_PROCESSOR_NAME];
n = N;
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Get_processor_name(processor_name, &namelen);
fprintf(stdout,"Process %d of %d is on %s\n",
myid, numprocs, processor_name);
fflush(stdout);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
h = 1.0 / N;
sum = 0.0;
for (i = myid + 1; i <= N; i += numprocs) {
x = h * ((double)i - 0.5);
sum += (4.0 / (1.0 + x * x));
}
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
if (myid == 0) {
printf("pi is approximately %.16f\n", pi);
fflush(stdout);
}
```

✧ 运行结果

```
mpi@cloud-exp-63-22281168-0-6014:~$ mpiexec -f hosts.txt -n 5 ./Lab5
mpi@10.101.1.251's password:

Process 0 of 5 is on cloud-exp-63-22281168-0-6014
Process 4 of 5 is on cloud-exp-63-22281168-0-6014
Process 2 of 5 is on cloud-exp-63-22281168-0-6014
Process 1 of 5 is on cloud-exp-63-22281188-0-6022
Process 3 of 5 is on cloud-exp-63-22281188-0-6022
pi is approximately 3.1415926544231225
mpi@cloud-exp-63-22281168-0-6014:~$
mpi@cloud-exp-63-22281168-0-6014:~$ mpiexec -f hosts.txt -n 8 ./Lab5
mpi@10.101.1.251's password:
Process 2 of 8 is on cloud-exp-63-22281168-0-6014
Process 4 of 8 is on cloud-exp-63-22281168-0-6014
Process 0 of 8 is on cloud-exp-63-22281168-0-6014
Process 6 of 8 is on cloud-exp-63-22281168-0-6014
Process 1 of 8 is on cloud-exp-63-22281188-0-6022
Process 3 of 8 is on cloud-exp-63-22281188-0-6022
Process 5 of 8 is on cloud-exp-63-22281188-0-6022
Process 7 of 8 is on cloud-exp-63-22281188-0-6022
pi is approximately 3.1415926544231247
```

运行结果分析:

这个程序首先通过`MPI_Init`初始化MPI环境，然后用`MPI_Comm_size`和`MPI_Comm_rank`来确定进程的总数和每个进程的ID。还使用

`MPI_Get_processor_name`来获取并打印运行每个进程的机器名称。接着使用`MPI_Bcast`来广播变量`n`的值到所有进程，确保每个进程都将执行相同数量的迭代来计算π的一部分。

每个进程计算它的矩形面积的一部分。然后使用`MPI_Reduce`来将所有进程计算的部分π值相加，形成最终的π近似值，并在根进程（进程0）上打印这个值。

实验结果分析

8个不同的进程报告了它们在两台不同的云服务器上运行。

每个进程报告了它们是8个进程中的第几个，并打印出了它们所在的机器名称。

最后，根进程（进程0）打印出了计算出的π值：“pi is approximately 3.1415926544231247”。

则从结果中可以看出：

MPI程序成功地在多台机器上并行执行。

每个进程正确地计算了它应该计算的π的一部分，并将结果发送给根进程。

根进程成功地通过`MPI_Reduce`收集了所有的部分和，并计算出了π的近似值。

计算出的π值与π的实际值（3.14159265358979323846...）相当接近，表明并行计算是正确的。

Lab5实验出现的各种问题

首先第一步是进行两台机器的SSH的免密登录。在配置此过程中，我们出现了找不到knowns_hosts的文件，以及在ssh-copy-id的过程中，找不到sever2接收的authorized_key的密钥文件，之后我们通过ssh先从host进入sever2，而后一定要通过cd .ssh进入ssh的路径，方可找到authorized_keys。而且我们由于前期copy多次密钥传输至authorized_keys，而每次写入模式是追加而非覆盖，因此authorized_keys文件中记录了多次密钥。通过不断摸索修改，我们最后两台机器成功进行了SSH的互相免密登录。

在程序最后的运行阶段，我们发现无法运行，因此我们最后将hosts.txt中的外部ip改为内部ip，即可成功运行。