

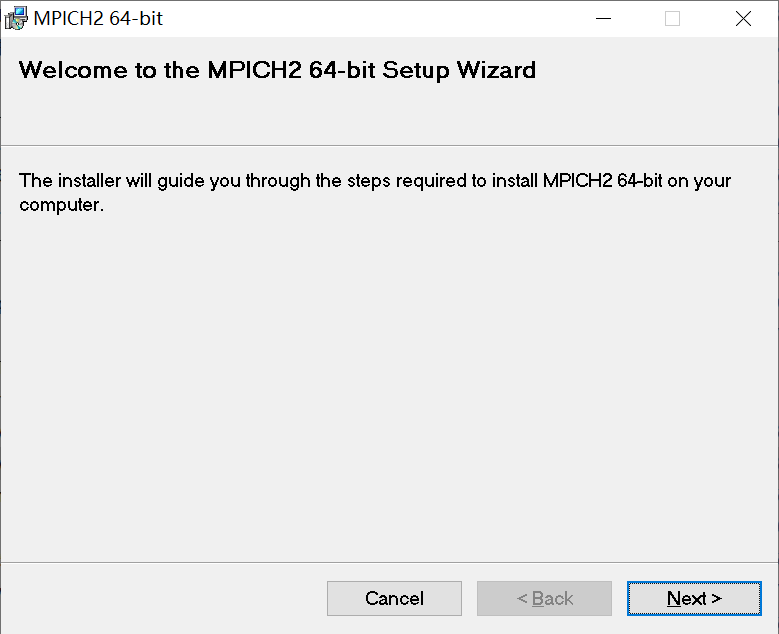
# 高性能计算导论(HPC)

|  |  |  |
| --- | --- | --- |
| 学号： | 17231163 | |
| 姓名： | 李一帆 | |
| 专业： | 计科1705 | |
| 学院： | 计算机与信息技术学院 | |
|  |  |  |
|  |  |  |
| 提交日期： | 2019年 11月 15日 | |

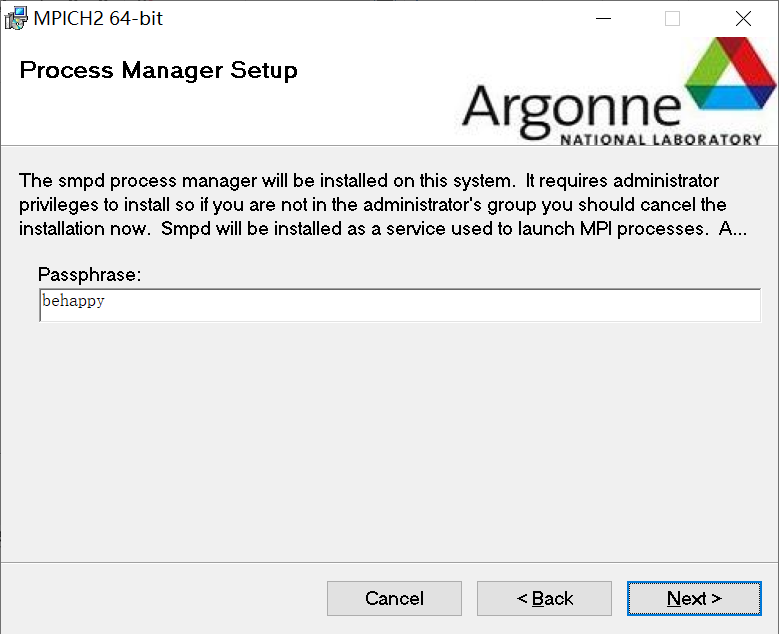
# MPI实验报告

## MPICH2 installation

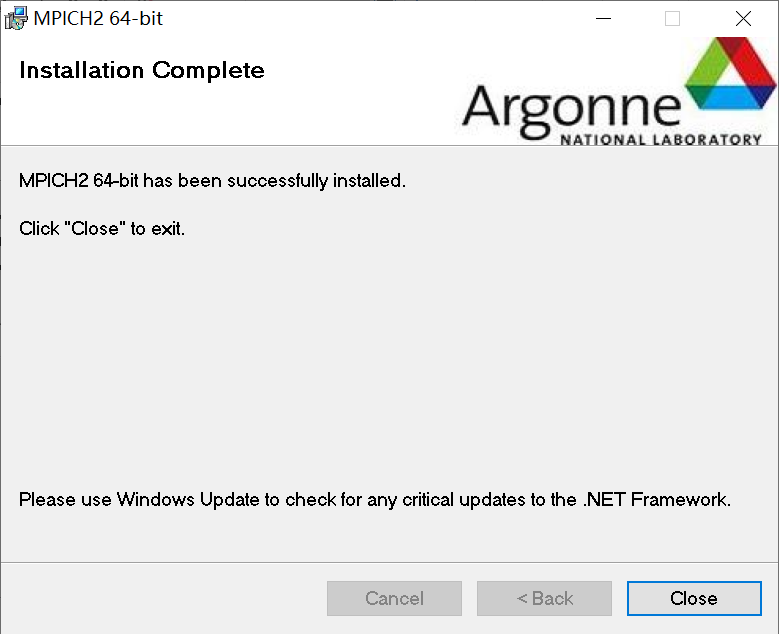
首先是打开安装包



一直next下一步，注意此过程设置的passphrase，这是smpd的执行口令



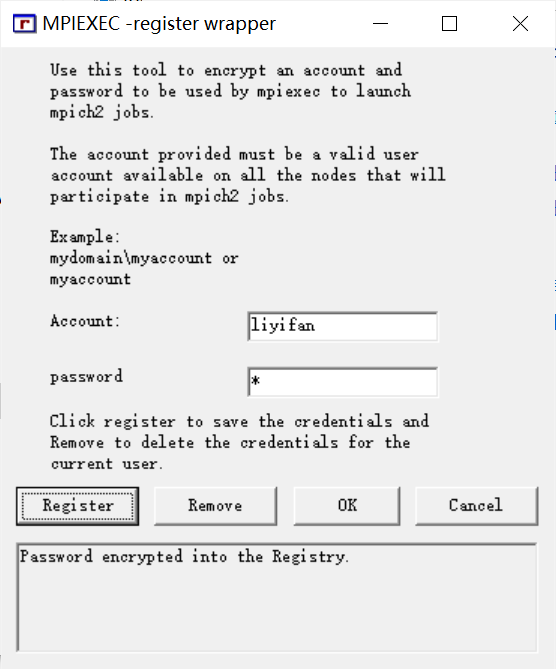
安装过程完成了



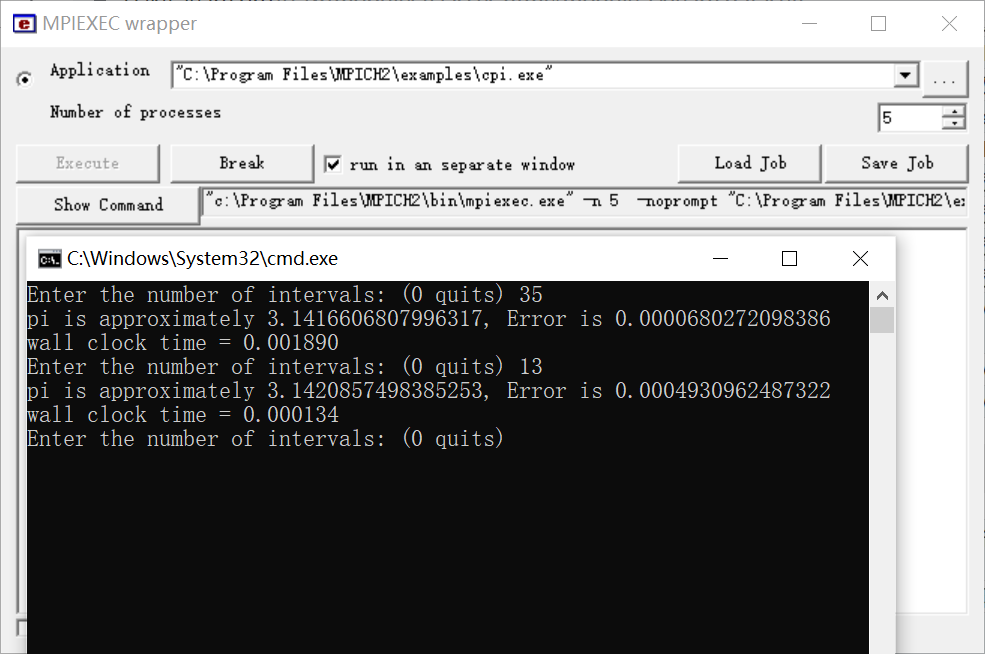
然后需要进行配置，先安装smpd



然后注册wrapper



测试自带的例子cpi.exe



可以将mpiexec的路径加入环境变量path里，使用命令行直接就可以操作。

编译器可以选择使用MinGW64，即Windows移植版的gcc/g++，使用命令

g++ xxx.cpp -I./include -L./lib -lmpi -o xxx.exe

就可以将cpp文件编译为exe了，使用gcc的话需要多加一个参数-lstdc++

## Blocking Send-Receive

### 实验代码

#include <stdio.h>

#include <mpi.h>

int main(){

MPI\_Status status;

char string[] = "xxxxx";

int myid;

MPI\_Init(NULL,NULL);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&myid);

if(myid == 0)

MPI\_Send((void\*)"HELLO",5,MPI\_CHAR,1,1234,MPI\_COMM\_WORLD);

if(myid == 1){

MPI\_Recv(string,5,MPI\_CHAR,0,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();

return 0;

}

### 运行结果截屏

单机调试



联机运行



### 结果分析

阻塞式通信，对每个进程来说，必须等待MPI\_Send/ MPI\_Recv函数执行完毕才会继续执行下面的语句，此通信模型相对简单，因为是同步通信，数据传输是完全匹配的，不会混乱。

## Nonblocking Send-Receive

### 实验代码

#include "mpi.h"

#include <stdio.h>

int main(int argc, char \*\*argv){

int numtasks, rank, dest, source, flag, count, 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);

MPI\_Wait(&reqs[0], &stats[0]); //按需注释

printf("Send succeed!\n"); //按需注释

fflush(stdout); //按需注释

}

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);

printf("task %d will finalize!\n", rank);

MPI\_Finalize();

printf("task %d exit!\n", rank);

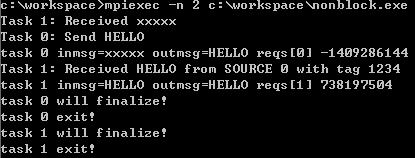
return 0;

}

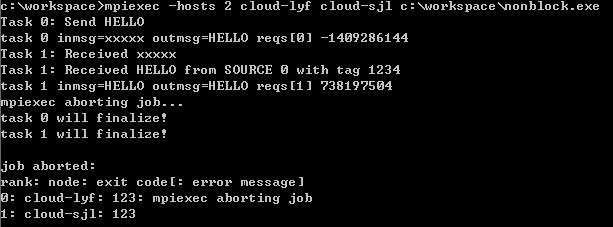
### 运行结果截屏

**发送方没有使用wait函数**

单机

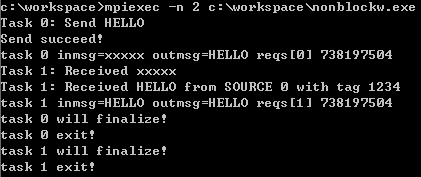


联机



**发送方使用了wait函数**

单机



联机



### 结果分析

该模型是非阻塞式通信，每个进程执行完MPI\_Isend/ MPI\_Irecv后继续往下执行，知道遇到MPI\_Wait才会暂停等待通信完成。

按照逻辑推理，发送方是可以不使用MPI\_Wait语句的，因为它只负责发送数据。但是测试结果并不是这样，通过结果截图可以看到，单机调试可以自动退出，但联机后程序无法自动退出，使用Ctrl+C才可以退出，并且会报错；而且发送方进程打印出来的请求句柄指针也是一个负数。

而发送方使用MPI\_Wait语句后，联机后程序可以正常退出，并且发送方打印出来的请求句柄指针也变成了正值，且与接收方的请求句柄指针相同。

说明MPI\_Wait会影响请求句柄，并且发送方的请求句柄会与接收方的请求句柄配对，由于是非阻塞式通信，发送方进程执行完发送语句就往下执行了，不等待的话打印出来的值就是野指针的值。而且发送方会等待MPI\_Wait响应后执行MPI\_Finalize，然后结束程序，否则就会一直等待下去。

## PI estimation

### 实验代码

#include <stdio.h>

#include <time.h>

#include "mpi.h"

#define N 100000

int main(){

clock\_t start = clock();

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;

for(int k=0; k<1000; ++k){

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);

clock\_t end = clock();

double duration = (double)(end-start)/CLOCKS\_PER\_SEC;

printf("prcess No.%d time consumption is %f\n", myid, duration);

}

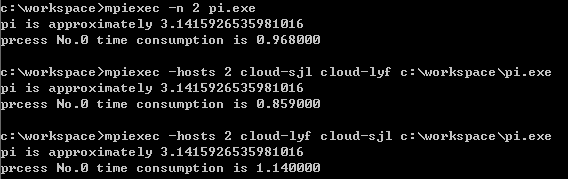
MPI\_Finalize();

return 0;

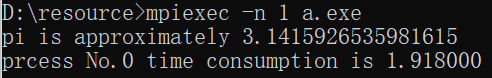
}

### 运行结果截屏

单机调试和联机运行的耗时对比



单进程运行



### 结果分析

使用广播进行圆周率的计算，联机运行的耗时差异较大，主要是来源是0号进程主进程是不是在本机上，在本机上的话耗时较短，因为该计算有个归纳过程，如果归纳在远端机器上，还需要耗时将其再传回来到本机上输出。如果没有这个过程，联机运算时间应该是比单机调试耗时要短。

比较单进程和多进程调试，可以看到消耗时间大幅减少，但如果指定的进程数多于机器物理核数，则耗时会增长。