

科学计算导论

实验指导书

高性能计算实验室：刘涛、赵冬梅

西南科技大学 计算机科学与技术学院

2018 年 08 月

实 验 指 导 书

实验一 并行计算-MPI

实验目的：

初步了解并行计算的概念，能够在 Windows 平台上安装 MPI 库（OpenMPI 或者 MPICH2），并掌握在 Visual studio 中调用 MPI 头文件的方法；初步掌握使用 MPI 编制并行程序的步骤与核心函数的意义及其参数的含义。

实验内容：

- （1）介绍并行计算消息接口 MPI 的定义、应用背景与基本概念；
- （2）在教师指导下在 Windows 系统上安装 MPI，（鼓励学生在 Linux 操作系统上安装以及完成实验）；
- （3）教师讲解 MPI 程序编制的基本步骤，演示基于 MPI 的两个演示程序；
- （4）在教师指导下，学生完成基于 MPI 的两个演示程序（鼓励学生添加其他基于 MPI 的功能），完成相应的实验报告。

实验环境：

Windows 7 及以上操作系统，32/64 位系统均可，推荐 64 位；程序开发环境要求 Microsoft Visual Studio 2008 及以上，MPI 要求为：MPICH2；硬件要求为：I3 及以上的多核 CPU（对应 AMD 多核处理器亦可）。

【备注】：估计学生在 Linux 系统上完成实验，可根据学生自己的兴趣与爱好选择 Linux 版本。

实验介绍：

MPI 是一个跨语言的通讯协议，用于编写并行计算机。支持点对点和广播。MPI 是一个信息传递应用程序接口，包括协议和语义说明，它们指明其如何在各种实现中发挥其特性。MPI 的目标是高性能，大规模性，和可移植性。MPI 在今天仍为高性能计算的主要模型。

常见的 MPI 实现包括 OpenMPI，MPICH 等，可在不同平台上实现 MPI 的并行通信，常见的 MPI 函数如下：

1. `Mpi_init()` 初始化 MPI 执行环境，建立多个 MPI 进程之间的联系，为后续通信做准备；
2. `Mpi_finalize` 结束 MPI 执行环境；

3. `Mpi_comm_rank` 用来标识各个 MPI 进程的, 给出调用该函数的进程的进程号, 返回整型的错误值。两个参数: `MPI_Comm` 类型的通信域, 标识参与计算的 MPI 进程组; `&rank` 返回调用进程中的标识号;
4. `Mpi_comm_size` 用来标识相应进程组中有多少个进程;
5. `Mpi_send(buf, counter, datatype, dest, tag, comm)`: `buf`: 发送缓冲区的起始地址, 可以是数组或结构指针; `count`: 非负整数, 发送的数据个数; `datatype`: 发送数据的数据类型; `dest`: 整型, 目的的进程号; `tag`: 整型, 消息标志; `comm`: MPI 进程组所在的通信域。
 - a) 含义: 向通信域中的 `dest` 进程发送数据, 数据存放在 `buf` 中, 类型是 `datatype`, 个数是 `count`, 这个消息的标志是 `tag`, 用以和本进程向同一目的进程发送的其它消息区别开来。
6. `Mpi_recv(buf, count, datatype, source, tag, comm, status)`: `source`: 整型, 接收数据的来源, 即发送数据进程的进程号; `status`: `MPI_Status` 结构指针, 返回状态信息。

此外, 还包括如下定义或函数:

数据类型和预定义的量

用于作为参数的数据类型 `MPI_INT`, `MPI_DOUBLE`, `MPI_CHAR`, `MPI_Status`

预定义的量 `MPI_STATUSES_IGNORE`, `MPI_ANY_SOURCE`, `MPI_ANY_TAG`

集合通信

`MPI_Bcast` 广播, 使得数据有 `p` 份拷贝

`MPI_Scatter` 散发, 每份数据只拷贝一次

`MPI_Gather` 收集, 每份数据只拷贝一次

`MPI_Reduce` 归约

点到点通信函数

`MPI_Barrier(communicator)` 来完成同步

`MPI_Bsend(message_data, size, data_type, dest_id, tag, communicator)` 来发送数据, 需要预先注册一个缓冲区, 并调用 `MPI_Buffer_attach(buffer, buf_size)` 来供 MPI 环境使用

`MPI_Buffer_attach(buffer, size)` 来把缓冲区 `buffer` 提交给 MPI 环境, 其中 `buffer` 是通过 `malloc` 分配的内存块。

`MPI_Buffer_detach(&buffer, &size)` 来确保传输的完成, 尽量把 `detach` 和 `attach` 函数配对使用, 正如尽可能同时使用 `malloc` 和 `free`, 同时使用 `Init` 和 `Finalize`, 防止遗漏!

`MPI_Pack_size(size, data_type, communicator, &pack_size)` 来获取包装特定类型的数据所需要的缓冲区大小(还没有计入头部, 所以真正缓冲区大小 `buf_size = MPI_BSEND_OVERHEAD + pack_size`, 如果有多份数据发送, 则 `buf_size` 还要叠加)。

使用 `MPI_Recv(message, size, data_type, src_id, tag, communicator, status)` 来接收数据, 把已经到达接收缓冲区的数据解析到 `message` 数组中, 只有全部数据都解析出来时, 函数才返回。

获取当前时间

`double MPI_Wtime(void)` 取得当前时间, 计时的精度由 `double MPI_Wtick(void)` 取得。作为对比, 一般在 C/C++ 中, 插入 `time.h`, 通过 `clock_t clock(void)` 取得当前时间, 计时的精度由常数 `CLOCKS_PER_SEC` 定义。

演示程序 1:

传统C/C++程序学习中第一个程序为“Hello world!”,此实验中首先给出C/C++语言的实现版本;如下

C语言版:

```
#include<stdio.h>
int main(void)
{
    /*下面要输出 hello world*/
    printf("hello world!");
    return 0;
}
```

C++语言版:

```
#include "iostream"
using namespace std;
int main(void)
{
    cout<<"hello word!"<<endl;
    return 0;
}
```

MPI+C语言版:

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char * argv[])
{
    int myrank, nprocs;
    // 初始化 MPI 环境
    MPI_Init(&argc, &argv);
    // 获取当前进程在通信器 MPI_COMM_WORLD 中的进程号
    MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
    printf("Hellow, world! %dth of totalTaskNum = %d\n", myrank, nprocs);
    MPI_Finalize();
    return 0;
}
```

得到的输出结果如下:

```

D:\科学计算导论实验\C1\MPI\MPI\Debug>C:\Program Files (x86)\MPICH2\bin\mpiexec -n 8 MPIC.exe
User credentials needed to launch processes:
account (domain\user) [PC-20180529AICN\Administrator]: Administrator
password:
Hello world! 0th of TotalTaskNum = 8
Hello world! 3th of TotalTaskNum = 8
Hello world! 6th of TotalTaskNum = 8
Hello world! 1th of TotalTaskNum = 8
Hello world! 4th of TotalTaskNum = 8
Hello world! 5th of TotalTaskNum = 8
Hello world! 7th of TotalTaskNum = 8
Hello world! 2th of TotalTaskNum = 8

D:\科学计算导论实验\C1\MPI\MPI\Debug>C:\Program Files (x86)\MPICH2\bin\mpiexec -n 16 MPIC.exe
Hello world! 12th of TotalTaskNum = 16
Hello world! 1th of TotalTaskNum = 16
Hello world! 15th of TotalTaskNum = 16
Hello world! 2th of TotalTaskNum = 16
Hello world! 3th of TotalTaskNum = 16
Hello world! 9th of TotalTaskNum = 16
Hello world! 11th of TotalTaskNum = 16
Hello world! 14th of TotalTaskNum = 16
Hello world! 5th of TotalTaskNum = 16
Hello world! 4th of TotalTaskNum = 16
Hello world! 0th of TotalTaskNum = 16
Hello world! 13th of TotalTaskNum = 16
Hello world! 10th of TotalTaskNum = 16
Hello world! 7th of TotalTaskNum = 16
Hello world! 6th of TotalTaskNum = 16
Hello world! 8th of TotalTaskNum = 16

```

MPI+C++语言版:

```

#include "mpi.h"
#include <iostream>
using namespace std;

int main(void) {
    int rankID;
    int sizeNum;
    MPI_Init(0, 0);
    MPI_Comm_size(MPI_COMM_WORLD, &sizeNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &rankID);
    cout << "Hello world! " << rankID << " of total = " << sizeNum << endl;
    MPI_Finalize();
    return 0;
}

```

演示程序 2:

计时函数的使用:

```

#include "stdafx.h"
#include "mpi.h"
#include <stdio.h>
#include <windows.h>

int _tmain(int argc, _TCHAR* argv[])
{
    int myrank, nprocs, name_len, flag;
    double start_time, end_time;

```

```

char host_name[20];

MPI_Initialized(&flag);
fprintf(stderr, "flag:%d/n", flag);

MPI_Init(0,0);

MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_Comm_size(MPI_COMM_WORLD,&nprocs);

MPI_Get_processor_name(host_name, &name_len);

if (myrank == 0)
{
    fprintf(stderr, "Precision of MPI_WTIME(): %f.\n", MPI_Wtick());
    fprintf(stderr, "Host Name:%s\n", host_name);
}

start_time = MPI_Wtime();
Sleep(myrank * 3);
end_time = MPI_Wtime();

fprintf(stderr, "myrank: %d. I have slept %f seconds.\n", myrank, end_time-
start_time);

MPI_Finalize();

return 0;
}

```

结果如下图所示:

```

D:\科学计算导论实验\C1\MPI\MPITIME\Debug>"c:\Program Files (x86)\MPICH2\bin\mpiexec.exe" -n 16 MPITIME.exe
flag:0/nmyrank: 10. I have slept 0.031147 seconds.
flag:0/nmyrank: 3. I have slept 0.015565 seconds.
flag:0/nmyrank: 9. I have slept 0.031135 seconds.
flag:0/nmyrank: 7. I have slept 0.031261 seconds.
flag:0/nmyrank: 6. I have slept 0.031248 seconds.
flag:0/nmyrank: 1. I have slept 0.015608 seconds.
flag:0/nmyrank: 2. I have slept 0.015530 seconds.
flag:0/nmyrank: 11. I have slept 0.046815 seconds.
flag:0/nmyrank: 15. I have slept 0.046837 seconds.
flag:0/nmyrank: 5. I have slept 0.015515 seconds.
flag:0/nmyrank: 13. I have slept 0.046900 seconds.
flag:0/nmyrank: 4. I have slept 0.015624 seconds.
flag:0/nPrecision of MPI_WTIME(): 0.000000.
Host Name:PC-20180529AICN
myrank: 0. I have slept 0.000001 seconds.
flag:0/nmyrank: 8. I have slept 0.031261 seconds.
flag:0/nmyrank: 14. I have slept 0.046831 seconds.
flag:0/nmyrank: 12. I have slept 0.046858 seconds.

```

实验步骤:

【备注】: 在编译完成含 MPI 代码的程序之后, 直接运行该程序, 依然会出现与无 MPI 代码的程序一样的结果, 需要调用 `mpiexec` 命令来指定运行生成的可执行文件, 具体格式如下 (Windows 系统中需要调用 `cmd` 终端界面):

`Mpiexec -n 8 xx.exe`

需要注意的是, 如果未在系统环境变量中指定 `mpiexec` 的路径, 则需要在命令行中给出该命令的路径形式, “-n” 则是指定多少个进程数, “xx.exe” 则是自己编译得到的可执行文件名。

实验要求:

- (1) 独立完成两个演示程序。
- (2) 独立完成实验报告, 给出在操作系统上安装 MPI 的详细过程, 以及实现的代码, 着重分析 MPI 并行程序的优势以及可能的应用领域、普通 C/C++ 程序与 MPI 程序的区别, 在实验报告中应给出程序运行的结果截图。

实验二 MPI 程序设计

实验目的：

在第一个实验的基础上，采用 MPI 完成对于连续函数 $f(x)$ 的从起始位置到终止位置的面积积分，采用梯形计算近似面积。要求能够深入理解近似积分求面积的方法，掌握采用传统串行方法计算近似面积积分的方法，初步掌握 MPI 中消息传递机制（MPI_Send 和 MPI_Recv），能够理解采用 MPI 并行求解面积积分的思路和方法。

实验内容：

- （1）理解近似积分求面积的方法；
- （2）采用传统串行方法计算 $f(x) = 3.0 + 2.345 * x + 0.98372 * x^2 + 0.3221 * x^3$ 的指定区域面积，记录求解所消耗的时间（要求在不少于 10 种不同插样值下比较）；
- （3）采用 MPI 方法计算 $f(x) = 3.0 + 2.345 * x + 0.98372 * x^2 + 0.3221 * x^3$ 的指定区域面积，记录求解所消耗的时间（要求在不少于 10 种不同插样值下比较，该 10 种插样值与步骤（2）种保持一致）；
- （4）通过图表的方式对比两种方式的时间消耗曲线图，分析其背后的影响因素，分析两种版本计算得到的面积值为何存在差异。

实验环境：

Windows 7 及以上操作系统，32/64 位系统均可，推荐 64 位；程序开发环境要求 Microsoft Visual Studio 2008 及以上，MPI 要求为：；硬件要求为：I3 及以上的多核 CPU（对应 AMD 多核处理器亦可）。

【备注】：鼓励学生在 Linux 系统上完成实验，可根据学生自己的兴趣与爱好选择 Linux 版本。

实验示范代码：

要求的实验步骤如下：

- （1）传统串行代码

```
LARGE_INTEGER now;
LARGE_INTEGER then;
LARGE_INTEGER fr;
double estimat = 0.0;
double delta = 0.0;
const int fre = 50000000;
//int myrank, nprocs, flag;
```



```

QueryPerformanceFrequency(&fr);
printf("CPU version begins at %d!\n", fre);

QueryPerformanceCounter(&now);
delta = (end - start) / fre;
for (int i = 0; i < fre; i++)
{
    estimat = estimat + abs(delta*(f(delta*i) + f(delta*(i+1)))) / 2);
}

QueryPerformanceCounter(&then);
printf("Area is %f.\n", estimat);
printf("CPU耗时: %f毫秒\n", (double)(then.QuadPart - now.QuadPart) * 1000
/ (double)(fr.QuadPart));

system("pause");

```

(2) 核心并行程序代码

```

printf("MPI version begins at %d!\n", fre);
QueryPerformanceCounter(&now);

MPI_Initialized(&flag);
if (flag)
{
    printf("MPI cannot be initlized! Exit...\n");
    return 0;
}

int local_n;
double h, local_a, local_b;
double local_int, total_int;
int source;

MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

// 区间大小, 所有进程一样
h = (end - start) / fre;
local_n = fre / nprocs; // 每个processor需要处理的梯形的数目

/* Length of each process' interval of
* integration = local_n*h. So my interval

```

```

* starts at: */
Local_a = start + myrank*Local_n*h;
Local_b = Local_a + Local_n*h;
Local_int = Trap(Local_a, Local_b, Local_n, h);

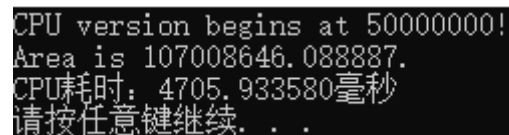
// 将所有进程的结果相加
if (myrank != 0) {
    MPI_Send(&Local_int, 1, MPI_DOUBLE, 0, 0,
            MPI_COMM_WORLD);
}
// 每个进程单独计算梯形面积
else {
    total_int = Local_int;
    for (source = 1; source < nprocs; source++) {
        MPI_Recv(&Local_int, 1, MPI_DOUBLE, source, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        total_int += Local_int;
    }
}

// 进程0打印结果
if (myrank == 0) {
    printf("Area from %f to %f = %f\n",
           start, end, total_int);
}
MPI_Finalize();

QueryPerformanceCounter(&then);
printf("MPI耗时: %f毫秒\n", (double)(then.QuadPart - now.QuadPart) * 1000
/ (double)(fr.QuadPart));
return 0;

```

传统并行的计算结果如下图所示：



```

CPU version begins at 500000000!
Area is 107008646.088887.
CPU耗时: 4705.933580毫秒
请按任意键继续. . .

```

并行的计算结果如下图所示：

```
D:\科学计算导论实验\C2\MPICode\Debug>MPICode.exe
MPI version begins at 50000000!
Area from 0.100000 to 190.000000 = 107232994.014603
MPI耗时: 4134.873365毫秒

D:\科学计算导论实验\C2\MPICode\Debug>"c:\Program Files (x86)\MPICH2\bin\mpiexec" -n 4 MPICode.exe
MPI version begins at 50000000!
Area from 0.100000 to 190.000000 = 107232994.014599
MPI耗时: 1428.867036毫秒
MPI version begins at 50000000!
MPI耗时: 1463.274298毫秒
MPI version begins at 50000000!
MPI耗时: 1442.264811毫秒
MPI version begins at 50000000!
MPI耗时: 1450.884738毫秒
```

实验要求:

- (1) 独立完成实验内容;
- (2) 独立完成实验报告, 分别完成在不同的函数求解区域(起始值和终止值)和不同插值数的计算耗时比较, 并通过数据图表对比在不同维度下不同计算方式的计算耗时, 分析两种程序的设计思路差异, 以及带来的计算性能的差异, 要求在实验报告最后附出源代码及注释。

实验三 GPU 加速矩阵计算

实验目的：

初步了解 GPU 与 CPU 计算的差异及各自的优缺点，在此基础上分别完成 CPU 和 GPU 版的矩阵计算程序，分别记录各自的计算时间并分析其中的差异。

实验内容：

矩阵元素是科学计算中基本的运算，其计算效率的高低直接决定了科学计算的时间成本，因此采用新的方法来加速矩阵运算是研究的一个重要方向，这其中包括了矩阵求解方法本身的研究，以及硬件加速求解的技术。

相对于 CPU，GPU 的结构更适合于大规模的浮点矩阵运算，理论上可以做到上百倍的加速比。本实验主要讲 GPU 引入到科学计算中，实现硬件加速矩阵求解。求解的矩阵方程如下：

$$aA+bB=C$$

其中， a 与 b 是矩阵的常系数， A 和 B 是两个同维度的矩阵， C 是存放求解结果的矩阵。

分别使用 CPU 和 GPU 完成该矩阵方程的计算过程，比较两种方式得到的结果矩阵是否一致，比较在不同矩阵维度下 GPU 的加速率，而后分析影响该加速率的影响因素。

【备注】：对于 32 位操作系统而言，矩阵的维度不要超过 11000。

实验环境：

Windows 7 及以上操作系统，32/64 位系统均可，推荐 64 位；程序开发环境要求 Microsoft Visual Studio 2008 及以上；硬件要求为：I3 及以上的多核 CPU（对应 AMD 多核处理器亦可），带独立 Nvidia 显卡（Geforce250 及以上），支持 CUDA 编程。

【备注】：估计学生在 Linux 系统上完成实验，可根据学生自己的兴趣与爱好选择 Linux 版本。

实验示范代码：

以下是 CPU 和 GPU 版本的核心示范代码。

```
.....  
#include "cublas.h"//需要额外引用Invidia开发包下的Include目录  
.....  
  
#define N 1000  
  
#pragma comment(Lib,"cublas.Lib")//需要额外指定引入库的目录
```

```

void simple_sgemm(int n, float alpha, float beta, float *a, float *b, float
*c)
{
    int i=0;
    int j=0;
    int k=0;

    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            float prod=0;
            for(k=0;k<n;k++)
            {
                prod+=a[k*n+i]*b[j*n+k];
            }
            c[j*n+i]=alpha*prod+beta*c[j*n+i];
        }
    }
}

double getDiff(LARGE_INTEGER n, LARGE_INTEGER t, LARGE_INTEGER f)
{
    .....
}

void main(int argc, char* argv[])
{
    printf("Cuda测试程序! By 刘涛 \n");
    printf("声明变量数组, 并分配内存! \n");

    .....

    h_A=(float*)malloc(n2*sizeof(float));
    if(h_A==0)
    {
        printf("h_A数组的内存分配失败, 程序将退出! \n");
        exit(1);
    }

    h_B=(float*)malloc(n2*sizeof(float));
    if(h_B==0)

```

```
{
    printf("h_B数组的内存分配失败, 程序将退出! \n");
    exit(1);
}

h_C=(float*)malloc(n2*sizeof(float));
if(h_C==0)
{
    printf("h_C数组的内存分配失败, 程序将退出! \n");
    exit(1);
}

printf("为ABC三个数组赋随机值! \n");

for(int i=0;i<n2;i++)
{
    h_A[i]=rand()/(float)RAND_MAX;
    h_B[i]=rand()/(float)RAND_MAX;
    h_C[i]=rand()/(float)RAND_MAX;
}

printf("初始化CUBLAS! \n");
status=cublasInit();
if(status!=CUBLAS_STATUS_SUCCESS)
{
    printf("初始化CUBLAS失败, 将退出程序! \n");
    exit(1);
}

QueryPerformanceFrequency(&fr);

printf("初始化d_A,d_B,d_C数组! \n");
status=cublasAlloc(n2,sizeof(d_A[0]),(void**)&d_A);
if(status!=CUBLAS_STATUS_SUCCESS)
{
    printf("初始化d_A失败, 将退出程序! \n");
    exit(1);
}

status=cublasAlloc(n2,sizeof(d_B[0]),(void**)&d_B);
if(status!=CUBLAS_STATUS_SUCCESS)
{
    printf("初始化d_B失败, 将退出程序! \n");
    exit(1);
}
```

```

}

status=cublasAlloc(n2,sizeof(d_C[0]),(void**)&d_C);
if(status!=CUBLAS_STATUS_SUCCESS)
{
    printf("初始化d_C失败, 将退出程序! \n");
    exit(1);
}

printf("传输数组数据到Device上! \n");
status=cublasSetVector(n2,sizeof(h_A[0]),h_A,1,d_A,1);
if(status!=CUBLAS_STATUS_SUCCESS)
{
    printf("传送数据h_A失败, 将退出程序! \n");
    exit(1);
}

status=cublasSetVector(n2,sizeof(h_B[0]),h_B,1,d_B,1);
if(status!=CUBLAS_STATUS_SUCCESS)
{
    printf("传送数据h_B失败, 将退出程序! \n");
    exit(1);
}

status=cublasSetVector(n2,sizeof(h_C[0]),h_C,1,d_C,1);
if(status!=CUBLAS_STATUS_SUCCESS)
{
    printf("传送数据h_C失败, 将退出程序! \n");
    exit(1);
}

printf("执行CPU版的sgemm函数! \n");
QueryPerformanceCounter(&now);
simple_sgemm(N,alpha,beta,h_A,h_B,h_C);
QueryPerformanceCounter(&then);
printf("CPU耗时: %f毫秒\n",getDiff(now,then,fr));
h_C_ref=h_C;

printf("执行CUBLAS版本的sgemm函数\n");
cublasGetError();
QueryPerformanceCounter(&now);
cublasSgemm('n','n',N,N,N,alpha,d_A,N,d_B,N,beta,d_C,N);
QueryPerformanceCounter(&then);
printf("GPU耗时: %f毫秒\n",getDiff(now,then,fr));

```

```

status=cublasGetError();
if(status!=CUBLAS_STATUS_SUCCESS)
{
    printf("调用cuBLas 计算失败, 将退出程序! \n");
    exit(1);
}

printf("重新为h_C分配内存, 并将cublas计算结果返回到h_C上! \n");
h_C=(float*)malloc(n2*sizeof(float));
if(h_C==0)
{
    printf("给h_C重新分配内存失败, 将退出程序! \n");
    exit(1);
}

status=cublasGetVector(n2, sizeof(h_C[0]), d_C, 1, h_C, 1);
if(status!=CUBLAS_STATUS_SUCCESS)
{
    printf("将数据传输到h_C上失败, 将退出程序! \n");
    exit(1);
}

printf("对比两者的计算结果! \n");
for(int i=0; i<n2; i++)
{
    diff=h_C_ref[i]-h_C[i];
    error_norm+=diff*diff;
    ref_norm+=h_C_ref[i]*h_C_ref[i];
}

error_norm=(float)sqrt((double)error_norm);
ref_norm=(float)sqrt((double)ref_norm);

printf("显示对比将结果\n");
printf("协方差: %f\n", error_norm);
printf("平方和: %f\n", ref_norm);

printf("对比结果: %s\n", (error_norm/ref_norm < 1e-6f)? "通过": "失败");

printf("释放内存! \n");
status=cublasFree(d_A);
if(status!=CUBLAS_STATUS_SUCCESS)
{
    printf("释放设备内存d_A失败, 将退出程序! \n");
}

```



```
        exit(1);
    }
    status=cublasFree(d_B);
    if(status!=CUBLAS_STATUS_SUCCESS)
    {
        printf("释放设备内存d_B失败, 将退出程序! \n");
        exit(1);
    }

    status=cublasFree(d_C);
    if(status!=CUBLAS_STATUS_SUCCESS)
    {
        printf("释放设备内存d_C失败, 将退出程序! \n");
        exit(1);
    }

    free(h_A);
    free(h_B);
    free(h_C);
    free(h_C_ref);

    printf("关闭计算设备! \n");
    status=cublasShutdown();
    if(status!=CUBLAS_STATUS_SUCCESS)
    {
        printf("关闭计算设备失败, 将退出程序! \n");
        exit(1);
    }

    system("pause");
    printf("程序运行结束! Bye\n");
}
```

当矩阵维度为 1000 时，计算得到的输出结果为：

```
Cuda测试程序! By刘涛 时间: 2012.7
声明变量数组, 并分配内存!
为ABC三个数组赋随机值!
初始化CUBLAS!
初始化d_A, d_B, d_C数组!
传输数组数据到Device上!
执行CPU版的sgemm函数!
CPU耗时: 6144.282959毫秒
执行CUBLAS版本的sgemm函数
GPU耗时: 0.126178毫秒
重新为h_C分配内存, 并将cublas计算结果返回到h_C上!
对比两者的计算结果!
显示对比结果:
协方差: 0.021045
平方和: 300506.156250
对比结果: 通过
释放内存!
关闭计算设备!
请按任意键继续. . .
```

实验步骤:

- (1) 教师讲解基本的矩阵运算;
- (2) 完成 CPU 版的矩阵运算代码;
- (3) 完成 GPU 版的矩阵运算代码;
- (4) 完成对于两个版本的计算结果的演算;
- (5) 分别比较在不同维度下两种计算方式的耗时及原因。

实验要求:

- (1) 1-2 人每组完成实验内容;
- (2) 独立完成实验报告, 要求完成 5 种维度以上 (32 位操作系统的维度不大于 1.2 万) 的串程序与 GPU 计算程序的矩阵加运算, 并通过数据图表对比在不同维度下不同计算方式的计算耗时, 分析两种程序的设计思路差异, 以及带来的计算性能的差异, 要求在实验报告最后附出源代码及注释。

实验四 热扩散方程求解与模拟

实验目的：

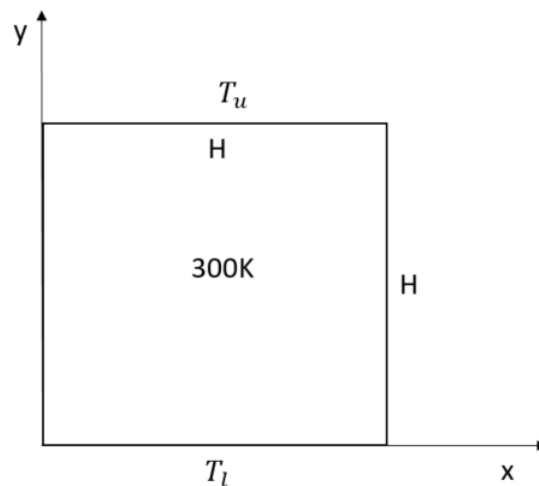
结合课程中的理论知识，完成二维热传导的数值计算程序，深入理解从物理现象到计算的过程和步骤，初步掌握简单物理现象的模拟方法，熟悉使用 C/C++ 来完成二维热传导的模拟程序。

实验内容：

假设一个二维空腔中充满空气，且上壁面为恒定低温，下壁面为恒定高温，忽略重力作用，热传导控制方程如下：

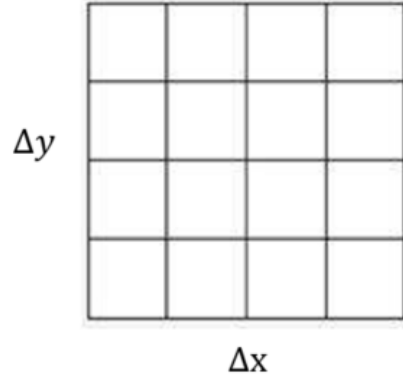
$$\rho c \frac{\partial T}{\partial t} = \lambda \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right)$$

方腔如下图所示：



空间离散

首先考虑空间离散，将以上区域离散为如下形式：



将正方形区域离散为如上图所示的网格， x 轴方向的距离步进值为 Δx ， y 轴方向为 Δy 。

内部节点的离散格式如下：

$$\left(\frac{\partial u}{\partial x}\right)_j^n = \frac{u_{j+1}^n - u_j^n}{\Delta x} + O(\Delta x)$$

$$\left(\frac{\partial v}{\partial y}\right)_j^n = \frac{v_{j+1}^n - v_j^n}{\Delta y} + O(\Delta y)$$

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_j^n = \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2} + O(\Delta x^2)$$

$$\left(\frac{\partial^2 v}{\partial y^2}\right)_j^n = \frac{v_{j+1}^n - 2v_j^n + v_{j-1}^n}{\Delta y^2} + O(\Delta y^2)$$

以上为空间域内部点的离散。

对于边界节点而言，在本问题中均为一类边界条件，即定值，故在计算循环之前，将其设定固定值即可。很显然，上边界的所有边界点被设定为 **400K**，下边界和左右边界的边界点均为 **300K**。

初次之外，在计算之前，还应该对内部所有的节点赋予初值，其物理意义为在 **0** 时刻时，计算域内部物理量的初始分布。很显然，根据题设，内部所有节点的温度初值均为 **300K**。

时间离散

控制方程中左侧为 $\frac{\partial T}{\partial t}$ ，一般而言，时间项的离散最多为二阶，在这里，为了简单起见，采用一阶显示来离散时间项，如下：

$$\frac{\partial T}{\partial t} = \frac{T_{j+1}^n - T_j^n}{\Delta t}$$

整理带入原控制方程，得到离散后的代数方程，如下：

$$T_{ij}^{k+1} = T_{ij}^k + \frac{\lambda}{\rho c} \left(\frac{\Delta t}{\Delta x^2} (T_{i+1,j}^k - 2T_{ij}^k + T_{i-1,j}^k) + \frac{\Delta t}{\Delta y^2} (T_{i,j+1}^k - 2T_{ij}^k + T_{i,j-1}^k) \right)$$

请注意方程左侧，为下一个时刻（经过 Δt ）的温度值，方程右侧 $T_{(i,j)}^k$ 则为当前计算时刻的温度值。

该离散方程为显式求解，在 Δt 与 Δx 和 Δy 之间存在一定的比例关系，否则整个方程随着时间的发展，计算的温度值会溢出，这就是在前面讲到的显示计算格式的条件稳定性，也就是说 Δt 与 Δx 和 Δy 之间一定要满足一定关系，此离散方程才不是发散，而是收敛的，这是采用显示求解需要密切关注的问题之一。

使用 C/C++ 来完成二维热传导的模拟程序，并对得到的数据文件进行可视化显示。

实验环境：

Windows 7 及以上操作系统，32/64 位系统均可，推荐 64 位；程序开发环境要求 Microsoft Visual Studio 2008 及以上，MPI 要求为：MPICH2；硬件要求为：I3 及以上的多核 CPU（对应 AMD 多核处理器亦可）。

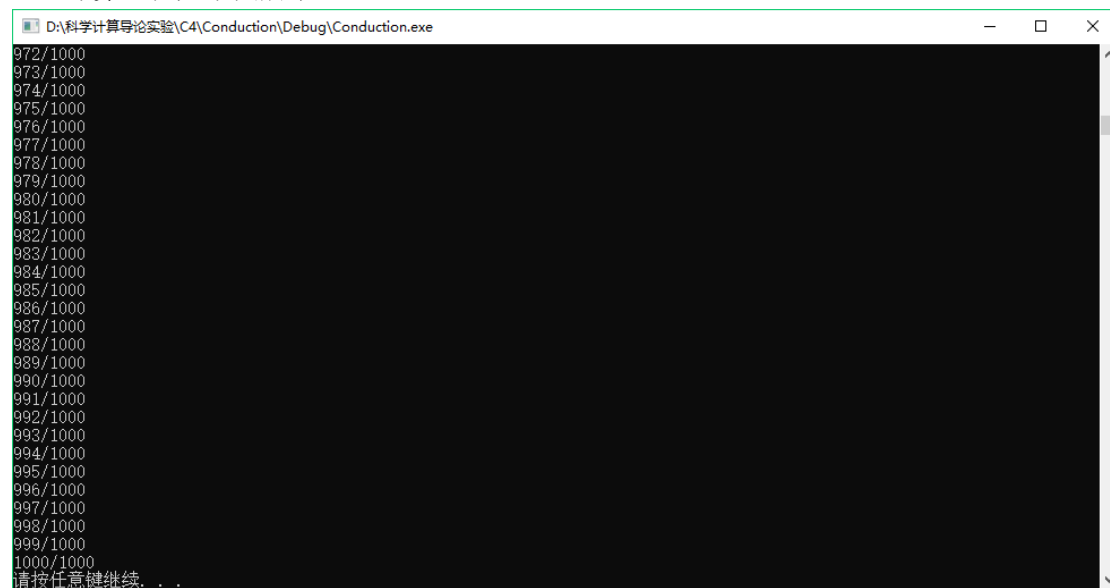
【备注】：估计学生在 Linux 系统上完成实验，可根据学生自己的兴趣与爱好选择 Linux 版本。

实验示范代码：

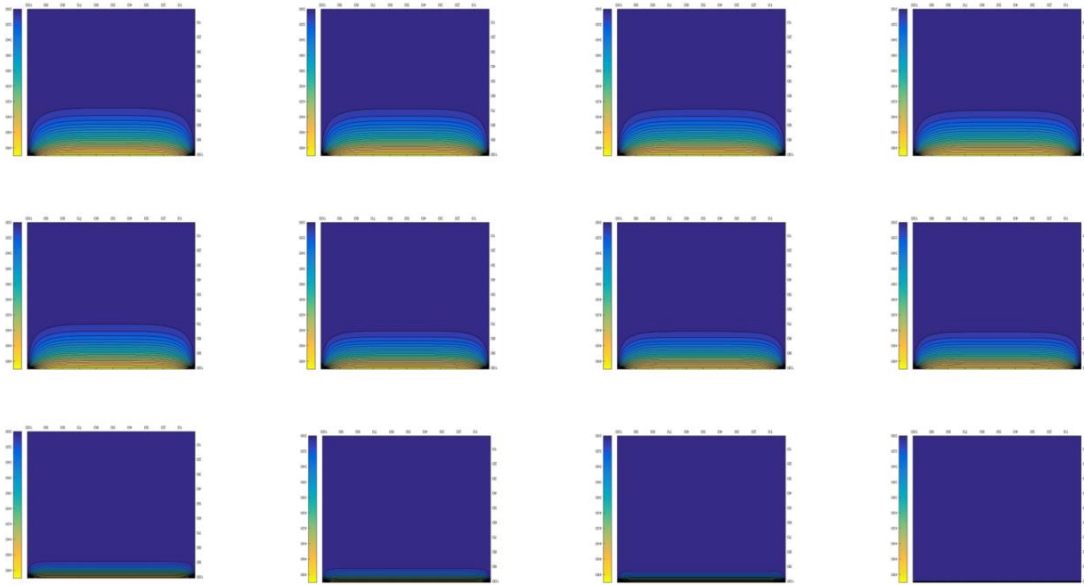
以下是试验示范代码：

.....
略

计算过程如下图所示：



可视化显示的结果如下所示：



可视化计算结果：

对实验数据文件进行可视化操作。鼓励同学使用自己的方式完成可视化显示。

下面给出基于 matlab 的后处理代码供参考：

```
function[] = getAlljpg(a,b,c)
%a- 为存储 csv 文件的目录，以结尾，b 为 csv 文件数量,c-循环增量，即间隔几个时间点
处理数据
for i=0:b:c
if i<10 f=[a,'000000',num2str(i),'.csv'];
elseif i<100 f=[a,'00000',num2str(i),'.csv'];
elseif i<1000 f=[a,'0000',num2str(i),'.csv'];
elseif i<10000 f=[a,'000',num2str(i),'.csv'];
elseif i<100000 f=[a,'00',num2str(i),'.csv'];
end;
m=csvread(f);
pcolor(m);
shading interp;
colorbar;
if i<10 saveas(gcf,[a,'wenduchang000000',num2str(i),'.jpg']);
elseif i<100 saveas(gcf,[a,'wenduchang00000',num2str(i),'.jpg']);
elseif i<1000 saveas(gcf,[a,'wenduchang0000',num2str(i),'.jpg']);
elseif i<10000 saveas(gcf,[a,'wenduchang000',num2str(i),'.jpg']);
elseif i<100000 saveas(gcf,[a,'wenduchang00',num2str(i),'.jpg']);
end
contourf(m,20);
colorbar;
if i<10 saveas(gcf,[a,'dengshixian000000',num2str(i),'.jpg']);
elseif i<100 saveas(gcf,[a,'dengshixian00000',num2str(i),'.jpg']);
```

```
elseif i<1000 saveas(gcf,[a,'dengshixian0000',num2str(i),'.jpg']);  
elseif i<10000 saveas(gcf,[a,'dengshixian000',num2str(i),'.jpg']);  
elseif i<100000 saveas(gcf,[a,'dengshixian00',num2str(i),'.jpg']);  
end  
end
```

实验要求：

- (1) 1-2 人每组完成实验内容；
- (2) 独立完成实验报告，要求在实验报告最后附出源代码及注释。

附录

科学计算导论

实验报告

学号	
姓名	
班级	
实验名称	
组号	(单人一组或未分组则无需填写)
报告时间	
成绩	

一、实验过程

1.1 实验基本内容与要求

1.2 实验过程（含流程分析、代码实现，以及必要的贴图）

二、实验结果

（包含实验结果的文字和图像描述）

三、实验分析

（结合实验过程中的内容对实验结果的分析，必须包含文字分析内容，可以附加相关的图表分析内容。）

四、参考文献

（鼓励在实验过程、分析中引用已经公开发表的文献，包括期刊论文、会议论文、出版物等。）

附录：（实现代码粘贴于此处，含不少于总代码量 1/2 的注释内容）

开发语言/版本	
编译环境/编译器	
运行环境（操作系统，版本，32/64 位）	

（代码/注释粘贴处）