中山大学计算机院本科生实验报告

（2023 学年春季学期）

课程名称：并行程序设计　　批改人：

实验	Pthreads 并行构造	专业（方向）	计算机科学与技术
学号	21307082	姓名	赖耿桂
Email	laigg@mail2.sysu.edu.cn	完成日期	2024.5.6

1. 实验目的（200 字以内）

实验一

模仿 OpenMP 的 omp_parallel_for 构造基于 Pthreads 的并行 for 循环分解、分配及执行机制并生成一个包含 parallel_for 函数的动态链接库（.so）文件，该函数创建多个 Pthreads 线程，并行执行 parallel_for 函数的参数所指定的内容。

实验二

使用此前构造的 parallel_for 并行结构，将 heated_plate_openmp 改造为基于 Pthreads 的并行应用。

2. 实验过程和核心代码（600 字左右，图文并茂）

实验一

本次实验仍使用此前实验所用到的矩阵生成函数 random_matrix_generator(在 Generator.h 中)和打印矩阵函数 printMatrix。

首先在 Parallel_for.h 中，对相关参数和函数进行声明：

```
#ifndef PARALLEL_FOR_H
#define PARALLEL_FOR_H

#include<vector>
#include<cstdio>
#include<set>
#include <chrono>
```

```
using namespace std;
extern int M,N,K,thread_num,Inc, missions_per_thread;   // 矩阵规模参数、线程数、步长、每个线程要完成的任务数量
extern set<int> unassigned;     // 记录还未被分配到线程的任务编号
extern vector<double> A, B, C;      // 矩阵A，B，C
void printMatrix(vector<double>& mat, int row, int col);    //按照row*col 的规格 打印矩阵
void* functor(void* args);      // 执行每个线程被分配到的任务
double parallel_for(int start, int end, int inc, void* (*functor_ptr)(void*), void*args, int thread_num);
// 模仿OpenMP 的omp_parallel_for 构造基于Pthreads 的并行for 循环分解、分配及执行

// 定义一个结构体用来存放每个线程执行任务所需的参数
struct  FUNCTOR_ARGS
{
    /*
        M、N、K：矩阵规模参数
        missions_per_thread：该线程需完成的任务数量
        Incr：步长
        Indices：每个线程要完成的任务的编号
        构造函数：对 M, N, K, missions_per_thread, Incr 进行赋值，然后先从 unassigned 里找到当前最小的任务编号 I，将其
        作为该线程要完成的第一个任务并记录其为 current_index，然后看看 current_index+Incr 是否会越界，若不越界则将
        current_index+Incr 放入 Indices 中并将 current_index 更新为 current_index+Incr，否则取 unassigned 中最小的任务
        编号放入 Indices 中，将其赋值给 current_index，继续迭代直到已经领取了 missions_per_thread 个任务。注意，在
        从 unassigned 中取任务到 Indices 中时要移除 unassigned 中对应的任务编号。
     */
    int M, N, K, missions_per_thread, Incr;
    vector<int> Indices;
    FUNCTOR_ARGS(int m,int n, int k, int missions, int inc){
        this->M = m;
        this->N = n;
        this->K = k;
        this->missions_per_thread = missions;
        this->Incr = inc;
        int current_index = *unassigned.begin();    // 刚领取到的任务编号
        this->Indices.emplace_back(current_index);
        unassigned.erase(current_index);
        int cnt = 1;    // 当前已经领取到的任务数量
        bool flag;      // 当前任务编号加上 Incr 后是否会越界
        while(cnt < this->missions_per_thread){
```

```
            flag = current_index + this->Incr < this->M;
            if(flag){
                current_index += this->Incr;
                this->Indices.emplace_back(current_index);
                unassigned.erase(current_index);
            }
            else{
                current_index = *unassigned.begin();
                this->Indices.emplace_back(current_index);
                unassigned.erase(current_index);
            }
            cnt++;
        }
    }
};

#endif
```

在 Parallel.cpp 中，实现 Parallel.h 中的相关函数：

```
#include "Parallel_for.h"
#include <pthread.h>


void printMatrix(vector<double>& mat, int row, int col){
    for(int i = 0; i < row; ++i){
        for(int j = 0; j < col; ++j){
            printf("%lf ",mat[i*col+j]);
        }
        printf("\n");
    }
}

void* functor( void* args){
    FUNCTOR_ARGS* Args = (FUNCTOR_ARGS*)args;
    int N = Args->N, K = Args->K;

    // 通用矩阵乘法
    for(int i = 0; i < Args->Indices.size(); ++i){
        int m = Args->Indices[i];   // 获取A 要参与计算的行号
        for(int k = 0; k < K; ++k){
```

```
            for(int n = 0; n < N; ++n){
                C[m*K + k] += A[m*N + n] * B[n*K + k];
            }
        }
    }
    pthread_exit(NULL);
}

double parallel_for(int start, int end, int inc, void* (*functor_ptr)(void*), void* args, int thread_num){
    pthread_t* handles = new pthread_t[thread_num];
    FUNCTOR_ARGS** missions = (FUNCTOR_ARGS**)args;
    auto start_time = chrono::high_resolution_clock::now();     // 开始计时
    for(int i = 0; i <thread_num; ++i){
        pthread_create(&handles[i],NULL, functor_ptr, (void*)missions[i]);  // 创建并启动线程
    }
    for(int i = 0; i < thread_num; ++i){
        pthread_join(handles[i],NULL);      // 等待线程结束并释放资源
    }
    auto end_time = chrono::high_resolution_clock::now();   // 结束计时
    auto using_time = (end_time-start_time).count()/1e9;
    return using_time;
}
```

在 main.cpp 中，依然是和前几次实验一样，初始化矩阵 A 和 B，然后调用 parallel_for 进行多线程矩阵乘法：

```
#include "Generator.h"
#include "Parallel_for.h"
int M,N,K,thread_num,Inc, missions_per_thread;
set<int> unassigned;
vector<double> A, B, C;
int main(){
    printf("请输入矩阵规模参数 M、N、K(矩阵 A 的大小为 M*N，矩阵 B 的大小为 N*K；在区间[128,2048]内的整数)：\n");
    scanf("%d %d %d",&M, &N, &K);
    printf("请输入线程数量(在区间[1,16]内的整数)： \n");
    scanf("%d",&thread_num);
    printf("请输入步长： \n");
    scanf("%d", &Inc);

    // 为矩阵 A，B，C 分配空间并将矩阵 C 初始化为全 0 矩阵
```

```
15.     A.resize(M*N);
16.     B.resize(N*K);
17.     C.assign(M*K, 0);
18.
19.     // 初始化矩阵A 和B 并打印它们
20.     A = random_matrix_generator(M,N);
21.     B = random_matrix_generator(N,K);
22.     printMatrix(A,M,N);
23.     printMatrix(B,N,K);
24.
25.     missions_per_thread = M / thread_num;   // 算出每个线程需要完成的任务数
26.     for(int i = 0; i < M; ++i){
27.         unassigned.insert(i);       // 初始化 unassigned，此时还未分配任务，所以0～M-1 均在unassigned 中
28.     }
29.
30.     FUNCTOR_ARGS** Args;
31.     Args = new FUNCTOR_ARGS*[thread_num];
32.     for(int i = 0; i < thread_num; ++i){
33.         Args[i] = new FUNCTOR_ARGS(M,N,K,missions_per_thread,Inc);  // 为每个线程分配任务
34.     }
35.
36.     double using_time = parallel_for(0, M, Inc, functor, (void*)Args, thread_num);
37.
38.     if(!unassigned.empty()){    // 此时对应M 无法被线程数整除的情况，需要计算剩余的行与矩阵B 的乘积，并加
    上这部分的用时
39.         auto start_time = chrono::high_resolution_clock::now();
40.         for(auto iter = unassigned.begin(); iter != unassigned.end(); iter++){
41.             int m = *iter;
42.             for(int k = 0; k < K; ++k){
43.                 for(int n = 0; n < N; ++n){
44.                     C[m*K+k] += A[m*N+n] * B[n*K+k];
45.                 }
46.             }
47.         }
48.         auto end_time = chrono::high_resolution_clock::now();
49.         using_time += (end_time-start_time).count()/1e9;
50.     }
51.     printf("Matrix C: \n");
52.     printMatrix(C, M, K);
53.     printf("在%d 个线程并行的情况下计算大小为%d*%d 的矩阵 A 和%d*%d 的矩阵 B 的乘积所用的时间
    为: %lfs\n",thread_num,M,N,N,K,using_time);
```

```
    return 0;
}
```

生成动态链接库（.so）文件：

```
g++ -fPIC -shared Parallel_for.cpp -o libparallelfor.so
```

将其复制到/usr/lib/目录下，以便编译时进行链接：

```
sudo cp libparallelfor.so /usr/lib
```

实验二

通过分析实验文件 heated_plate_openmp.c，可知该代码流程为：

①对矩阵 w 边界(外围)进行初始化(第一行全赋为 0，最后一行全赋为 100，最左和最右列[除了四个顶点 w[0][0]、w[0][N-1]、w[M-1][0]、w[M-1][N-1]也全赋为 0)并计算出边界的均值 mean，总运算量为(2*M+2*N-4)。

②将矩阵 w 内部即除了边界之外的部分全都赋值为 mean，总运算量为(M-2)*(N-2)。

③开始迭代后，将更新前的矩阵 w 的记录保存到矩阵 u，总运算量为 M*N。

④根据公式

$$w[i][j] = \frac{1}{4}(u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1])$$

更新矩阵 w 的值，总运算量为(M-2)*(N-2)。

⑤找出更新前后矩阵 w 当中所有元素变化量的最大值并与 epsilon 进行比较。

重复步骤③-⑤，直到矩阵 w 最大的变化量低于 epsilon。

因此根据以上 5 个步骤构造 5 个不同功能的函数，首先在 Parallel_for.h 中对相关变量和函数进行声明：

```
#ifndef PARALLEL_FOR_H
#define PARALLEL_FOR_H
// 矩阵参数
#define M 500
```

```
#define N 500

// 矩阵w和u（u用于记录上一次迭代的结果）
extern double w[M][N];
extern double u[M][N];

// 参数：diff用于记录每次迭代后矩阵中最大的变化值；mean用于为初始的矩阵非边界部分赋值
extern double diff;
extern double mean;

enum LOCATIONS{HORIZONTAL, VERTICAL};   // 为矩阵边界赋值时边界的方位（行：水平；列：竖直）
enum FUNCTION_TYPES{ASSIGN_BORDER, ASSIGN_INTERNAL, SAVE_LAST_MAT, UPDATE_MAT, UPDATE_DIFF};
//函数类型，用于在parallel_for中选择功能

void* assignBorder(void* args);     //给边界赋值并计算mean
void* assignInternal(void* args);   //给矩阵内部赋值为mean
void* saveLastMatrix(void* args);   //保存上一次迭代的矩阵信息
void* updateMatrix(void* args);     //根据热量扩散公式更新矩阵
void* updateDiff(void* args);   // 更新diff
void parallel_for(int type, int start,int end, int incr, void* (*functor_ptr)(void*), int thread_num );
// 使用pthread来模仿openmp的并行for循环分解、分配及执行机制
#endif
```

然后在 Parallel_for.c 中，针对 5 个不同功能的函数 assignBorder、assignInternal、saveLastMatrix、updateMatrix、updateDiff 构造 5 个不同的结构体，用于 pthread 执行不同功能的多线程任务时传递所需的参数：

```
struct BORDER_ARGS{
    /*
        start：起始索引
        end：终止索引（取不到）
        increment：步长
        location：方位，是给行边界赋值还是给列边界赋值
    */
    int start, end, increment;
    enum LOCATIONS location;
};
// 为矩阵内部赋值时所需参数
struct INTERNAL_ARGS{
    /*
```

```
        start：起始行号
        end：终止行号（取不到）
        increment：步长
        value：要赋的值（mean）
    */
    int start, end, increment;
    double value;
};
// 保存上一次迭代的矩阵信息所需参数
struct SAVE_LAST_MAT_ARGS{
    /*
        start：起始行号
        end：终止行号（取不到）
        increment：步长
    */
    int start, end, increment;
};
// 更新矩阵内部值时所需参数
struct UPDATE_MAT_ARGS{
    /*
        start：起始行号
        end：终止行号（取不到）
        increment：步长
    */
    int start, end, increment;
};
// 更新每次迭代的矩阵最大变化值所需参数
struct  UPDATE_DIFF_ARGS
{
    /*
        start：起始行号
        end：终止行号（取不到）
        increment：步长
        maxdiff：每个线程对应位置的矩阵元素最大变化值
    */
    int start, end, increment;
    double maxdiff;
};
```

assignBorder 函数用于对矩阵 w 边界进行初始化，将其分为水平(行边界)和竖直(列边界)两个方向：

```
void* assignBorder(void* args){
    struct BORDER_ARGS* Args = (struct BORDER_ARGS*)args;   // 进行类型转换，用于获取参数
    if(Args->location == HORIZONTAL){   // 要对顶上和地下两行赋值
        for(int i = Args->start; i < Args->end; i+=Args->increment){
            w[0][i] = 0;
            w[M-1][i] = 100;
            pthread_mutex_lock(&Mutex);
            mean += w[0][i] + w[M-1][i];    // 使用互斥锁保护mean 的计算
            pthread_mutex_unlock(&Mutex);
        }
    }
    else{   // 要对最左列和最右列两列赋值，注意不包含四个顶点
        for(int i = Args->start; i < Args->end; i+=Args->increment){
            w[i][0] = 100;
            w[i][N-1] = 100;
            pthread_mutex_lock(&Mutex);
            mean += w[i][0] + w[i][N-1];    // 使用互斥锁保护mean 的计算
            pthread_mutex_unlock(&Mutex);
        }
    }
    pthread_exit(NULL);     // 退出线程
}
```

assignInternal 则是将矩阵 w 的行划分给多个线程，将内部值赋值为 mean(存储在 value 参数中)：

```
void* assignInternal(void* args){
    struct INTERNAL_ARGS* Args = (struct INTERNAL_ARGS*)args;   // 进行类型转换，用于获取参数
    for(int i = Args->start; i < Args->end; i+=Args->increment){
        for(int j = 1; j < N-1; ++j){
            w[i][j] = Args->value;      // 矩阵内部赋值
        }
    }
    pthread_exit(NULL);     // 退出线程
}
```

saveLastMatrix 函数则是保存上一次迭代也即本次迭代更新前的矩阵 w 信息至矩阵 u，同样是对矩阵 w 的进行划分：

```
void* saveLastMatrix(void* args){
    struct SAVE_LAST_MAT_ARGS* Args = (struct SAVE_LAST_MAT_ARGS*)args;     // 进行类型转换，用于获取参数
    for(int i = Args->start; i < Args->end; i+=Args->increment){
        for(int j = 0; j < N; ++j){
            u[i][j] = w[i][j];      // 保存上一次迭代的结果
        }
    }
    pthread_exit(NULL);     // 退出线程
}
```

updateMatrix 函数则是根据公式更新矩阵内部的值，依旧对矩阵 w 的行进行划分：

```
void* updateMatrix(void* args){
    struct UPDATE_MAT_ARGS* Args = (struct UPDATE_MAT_ARGS*)args;    // 进行类型转换，用于获取参数
    for(int i = Args->start; i < Args->end; i+=Args->increment){
        for(int j = 1; j < N-1; ++j){
            w[i][j] = (u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1]) / 4;  // 根据公式更新矩阵内部的值
        }
    }
    pthread_exit(NULL);     // 退出线程
}
```

updateDiff 函数则是求出每个线程涉及的元素中的最大变化值，与当前迭代的最大变化值的比较放在后面的 parallel_for 中，过多使用 pthread 的互斥锁，该函数依然是对矩阵 w 的行进行划分：

```
void* updateDiff(void* args){
    struct UPDATE_DIFF_ARGS* Args = (struct UPDATE_DIFF_ARGS*)args;     // 进行类型转换，用于获取参数
    for(int i = Args->start; i < Args->end; i+=Args->increment){
        for(int j = 1; j < N-1; ++j){
            if(Args->maxdiff < fabs(w[i][j]-u[i][j])){
                Args->maxdiff = fabs(w[i][j]-u[i][j]);  // 更新矩阵内部最大的变化值
            }
        }
```

```
        }
        pthread_mutex_lock(&Mutex);
        if(diff < Args->maxdiff){
                    diff = Args->maxdiff;
                }
        pthread_mutex_unlock(&Mutex);
        pthread_exit(NULL);     // 退出线程
    }
```

接下来是核心部分 parallel_for 的实现，它可以对不同功能的循环进行分解、分配和执行，对于每个功能，最后一个线程都需要承担剩余所有任务(在无法为每个线程均匀分配任务的情况下；由于对边界赋值时边和列的起点不同，故没有用到 start 和 end 参数)：

```
    void parallel_for(int type, int start,int end, int incr, void* (*functor_ptr)(void*), int thread_num ){
        pthread_mutex_init(&Mutex,NULL);    // 初始化互斥锁
        if(type == ASSIGN_BORDER){
            pthread_t* handles = (pthread_t*)malloc(sizeof(pthread_t) * thread_num * 2);    // 为线程分配空间
            // 分成2*thread_num 个线程是因为行边界需要thread_num 个线程，列边界需要thread_num 个线程
            struct BORDER_ARGS** Args = (struct BORDER_ARGS**)malloc(sizeof(struct BORDER_ARGS*) * thread_num * 2);
            int rowChunksize = N / thread_num, colChunksize = (M-2) / thread_num;
            for(int i = 0; i < thread_num; ++i){
                // 初始化参数
                Args[i] = (struct BORDER_ARGS*)malloc(sizeof(struct BORDER_ARGS));
                Args[i+thread_num] = (struct BORDER_ARGS*)malloc(sizeof(struct BORDER_ARGS));
                Args[i]->location = HORIZONTAL;
                Args[i]->increment = incr;
                Args[i+thread_num]->location = VERTICAL;
                Args[i+thread_num]->increment = incr;
                if(i == 0){     // 第一个线程要把起点确定好，注意行和列的区别
                    Args[0]->start = 0;
                    Args[thread_num]->start = 1;
                    Args[0]->end = rowChunksize;
                    Args[thread_num]->end = colChunksize+1;
                }
                else if(i == thread_num-1){     // 最后一个线程要把剩余的所有任务都完成
                    Args[thread_num-1]->start = (thread_num-1)*rowChunksize;
```

```
                Args[2*thread_num-1]->start = (thread_num-1)*colChunksize+1;
                Args[thread_num-1]->end = N;
                Args[2*thread_num-1]->end = M-1;
            }
            else{   // 一般情况
                Args[i]->start = i*rowChunksize;
                Args[i]->end = Args[i]->start + rowChunksize;
                Args[i+thread_num]->start = i*colChunksize + 1;
                Args[i+thread_num]->end = Args[i+thread_num]->start + colChunksize;
            }

            // 创建并启动线程
            pthread_create(&handles[i],NULL,functor_ptr,Args[i]);
            pthread_create(&handles[i+thread_num],NULL,functor_ptr,Args[i+thread_num]);
        }
        for(int i = 0; i < thread_num; ++i){
            // 回收资源
            pthread_join(handles[i],NULL);
            pthread_join(handles[i+thread_num],NULL);
            free(Args[i]);
            free(Args[i+thread_num]);
        }
        free(Args);
    }
    else if(type == ASSIGN_INTERNAL){
        pthread_t* handles = (pthread_t*)malloc(sizeof(pthread_t) * thread_num);    // 为线程分配空间
        struct INTERNAL_ARGS** Args = (struct INTERNAL_ARGS**)malloc(sizeof(struct INTERNAL_ARGS*)*thread_num);
        int Chunksize = (M-2)/thread_num;
        for(int i = 0; i < thread_num; ++i){
            // 初始化参数
            Args[i] = (struct INTERNAL_ARGS*)malloc(sizeof(struct INTERNAL_ARGS));
            Args[i]->start = start + i*Chunksize;
            Args[i]->increment = incr;
            Args[i]->value = mean;
            if(i == thread_num-1){
                Args[i]->end = end;
            }
            else{
                Args[i]->end = Args[i]->start + Chunksize;
            }
```

```
64.             pthread_create(&handles[i],NULL,functor_ptr,Args[i]);       // 创建并启动线程
65.         }
66.         for(int i = 0; i < thread_num; ++i){
67.             // 回收资源
68.             pthread_join(handles[i],NULL);
69.             free(Args[i]);
70.         }
71.
72.         free(Args);
73.     }
74.     else if(type == SAVE_LAST_MAT){
75.         pthread_t* handles = (pthread_t*)malloc(sizeof(pthread_t) * thread_num);    // 为线程分配空
    间
76.         struct SAVE_LAST_MAT_ARGS** Args = (struct SAVE_LAST_MAT_ARGS**)malloc(sizeof(struct SAVE_L
    AST_MAT_ARGS*)*thread_num);
77.         int Chunksize = M/thread_num;
78.         for(int i = 0; i < thread_num; ++i){
79.             // 初始化参数
80.             Args[i] = (struct SAVE_LAST_MAT_ARGS*)malloc(sizeof(struct SAVE_LAST_MAT_ARGS));
81.             Args[i]->start = start + i*Chunksize;
82.             Args[i]->increment = incr;
83.             if(i == thread_num-1){
84.                 Args[i]->end = end;
85.             }
86.             else{
87.                 Args[i]->end = Args[i]->start + Chunksize;
88.             }
89.             pthread_create(&handles[i],NULL,functor_ptr,Args[i]);       // 创建并启动线程
90.         }
91.         for(int i = 0; i < thread_num; ++i){
92.             // 回收资源
93.             pthread_join(handles[i],NULL);
94.             free(Args[i]);
95.         }
96.         free(Args);
97.     }
98.     else if(type == UPDATE_MAT){
99.         pthread_t* handles = (pthread_t*)malloc(sizeof(pthread_t) * thread_num);    // 为线程分配空
    间
100.        struct UPDATE_MAT_ARGS** Args = (struct UPDATE_MAT_ARGS**)malloc(sizeof(struct UPDATE_MAT_A
    RGS*)*thread_num);
101.        int Chunksize = (M-2)/thread_num;
```

```
        for(int i = 0; i < thread_num; ++i){
            // 初始化参数
            Args[i] = (struct UPDATE_MAT_ARGS*)malloc(sizeof(struct UPDATE_MAT_ARGS));
            Args[i]->start = start + i*Chunksize;
            Args[i]->increment = incr;
            if(i == thread_num-1){
                Args[i]->end = end;
            }
            else{
                Args[i]->end = Args[i]->start + Chunksize;
            }
            pthread_create(&handles[i],NULL,functor_ptr,Args[i]);       // 创建并启动线程
        }
        for(int i = 0; i < thread_num; ++i){
            // 回收资源
            pthread_join(handles[i],NULL);
            free(Args[i]);
        }
        free(Args);
    }
    else{
        pthread_t* handles = (pthread_t*)malloc(sizeof(pthread_t) * thread_num);    // 为线程分配空间
        struct UPDATE_DIFF_ARGS** Args = (struct UPDATE_DIFF_ARGS**)malloc(sizeof(struct UPDATE_DIFF_ARGS*)*thread_num);
        int Chunksize = (M-2)/thread_num;
        for(int i = 0; i < thread_num; ++i){
            // 初始化参数
            Args[i] = (struct UPDATE_DIFF_ARGS*)malloc(sizeof(struct UPDATE_DIFF_ARGS));
            Args[i]->start = start + i*Chunksize;
            Args[i]->increment = incr;
            Args[i]->maxdiff = 0.0;
            if(i == thread_num-1){
                Args[i]->end = end;
            }
            else{
                Args[i]->end = Args[i]->start + Chunksize;
            }
            pthread_create(&handles[i],NULL,functor_ptr,Args[i]);       // 创建并启动线程
        }
        for(int i = 0; i < thread_num; ++i){
            // 回收资源
```

```
            pthread_join(handles[i],NULL);
            free(Args[i]);
        }
        free(Args);
    }
    pthread_mutex_destroy(&Mutex);
}
```

接下来在改进后的 heated_plate_openmp.c(更名为 heated_plate_pthread_for.c)中，将原来使用 openmp 的代码段更换成调用对应功能的 parallel_for 语句(为方便于原代码对比，将 increment 参数全置为 1)：

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <omp.h>
#include "Parallel_for.h"
# define M 500
# define N 500

  double diff;
  double epsilon = 0.001;
  int i;
  int iterations;
  int iterations_print;
  int j;
  double mean;
  double my_diff;
  double u[M][N];
  double w[M][N];
  double wtime;
int main (){
  int thread_num;
  printf("请输入线程数：\n");
  scanf("%d",&thread_num);
  printf ( "\n" );
  printf ( "HEATED_PLATE_OPENMP\n" );
  printf ( "  C/OpenMP version\n" );
  printf ( "  A program to solve for the steady state temperature distribution\n" );
  printf ( "  over a rectangular plate.\n" );
```

```
29.   printf ( "\n" );
30.   printf ( "  Spatial grid of %d by %d points.\n", M, N );
31.   printf ( "  The iteration will be repeated until the change is <= %e\n", epsilon );
32.   printf ( "  Number of processors available = %d\n", omp_get_num_procs ( ) );
33.   printf ( "  Number of threads =              %d\n", omp_get_max_threads ( ) );
34.
35.   mean = 0.0;
36.   parallel_for(ASSIGN_BORDER,0,M,1,assignBorder,thread_num);
37.
38.   mean = mean / ( double ) ( 2 * M + 2 * N - 4 );
39.   printf ( "\n" );
40.   printf ( "  MEAN = %f\n", mean );
41.
42.   parallel_for(ASSIGN_INTERNAL,1,M-1,1,assignInternal,thread_num);
43.
44.   iterations = 0;
45.   iterations_print = 1;
46.   printf ( "\n" );
47.   printf ( " Iteration  Change\n" );
48.   printf ( "\n" );
49.   wtime = omp_get_wtime ( );
50.
51.   diff = epsilon;
52.
53.   while ( epsilon <= diff )
54.   {
55.     parallel_for(SAVE_LAST_MAT,0,M,1,saveLastMatrix,thread_num);
56.     parallel_for(UPDATE_MAT,1,M-1,1,updateMatrix,thread_num);
57.     diff = 0.0;
58.     parallel_for(UPDATE_DIFF,1,M-1,1,updateDiff,thread_num);
59.
60.     iterations++;
61.     if ( iterations == iterations_print )
62.     {
63.       printf ( "  %8d  %f\n", iterations, diff );
64.       iterations_print = 2 * iterations_print;
65.     }
66.   }
67.   wtime = omp_get_wtime ( ) - wtime;
68.
69.   printf ( "\n" );
```

```
    printf ( "  %8d  %f\n", iterations, diff );
    printf ( "\n" );
    printf ( "  Error tolerance achieved.\n" );
    printf ( "  Wallclock time = %f\n", wtime );

    printf ( "\n" );
    printf ( "HEATED_PLATE_OPENMP:\n" );
    printf ( "  Normal end of execution.\n" );

    return 0;

# undef M
# undef N
}
```

生成动态链接库（.so）文件：

```
gcc Parallel_for.c -fPIC -shared -o libParallelFor2.so
```

将其复制到/usr/lib/目录下，以便编译时进行链接：

```
sudo cp libParallelFor2.so /usr/lib
```

3. 实验结果（500 字左右，图文并茂）

实验一

编译：

```
g++ main.cpp -L. -lparallelfor -o main -lpthread
```

运行：

```
./main
```

下为运行示例，展示程序正确性：

```
laigg@laigg-VirtualBox:~/codes/parallel/lab6/6_1$ ./main
请输入矩阵规模参数M、N、K(矩阵A的大小为M*N，矩阵B的大小为N*K；在区间[128,2048]内的整数):
4 4 4
请输入线程数量(在区间[1,16]内的整数):
1
请输入步长:
1
1.551286 -4.257329 1.457362 2.796868
-4.961518 -3.709996 -1.035028 -0.005524
-1.631311 -3.955646 -2.806898 -3.495523
-2.009049 2.484992 -4.733754 4.203128
1.972916 4.698080 4.738616 4.235579
2.501890 1.660524 0.342430 -1.024172
0.718790 0.096700 -1.288658 -1.492254
1.783611 3.960515 -3.088098 -3.581560
Matrix C:
-1.554752 11.436635 -4.621930 -1.261072
-19.824482 -29.592112 -23.430287 -15.650921
-21.367256 -28.347977 5.326958 13.849746
6.347666 10.876515 -15.548658 -19.044341
在1个线程并行的情况下计算大小为4*4的矩阵A和4*4的矩阵B的乘积所用的时间为: 0.000141s
```

(步长为 1)不同规模的矩阵在不同线程下的执行时间：

线程数	矩阵规模				
	128	256	512	1024	2048
1	0.012900	0.106791	0.997654	14.234169	222.517250
2	0.009930	0.054375	0.504169	6.722983	110.354257
4	0.003257	0.041024	0.317104	4.509141	65.879834
8	0.006068	0.046469	0.311147	3.106715	68.341060
16	0.004237	0.048225	0.289410	3.202291	70.704589

分析：

可以看到程序能够正常运行，且可以看到随着线程数的增加，总体上程序的效率呈现提高趋势且矩阵规模越大提高的效果越明显。虽然出现了部分线程数越多效率降低的情况，这可能是因为线程数越多，特别是 8 或 16 个线程，可能会造成资源竞争，且线程的创建和销毁开销也会相应增加，但是总体上相较于线程数较少的情况，效率还是要高上不少。

实验二

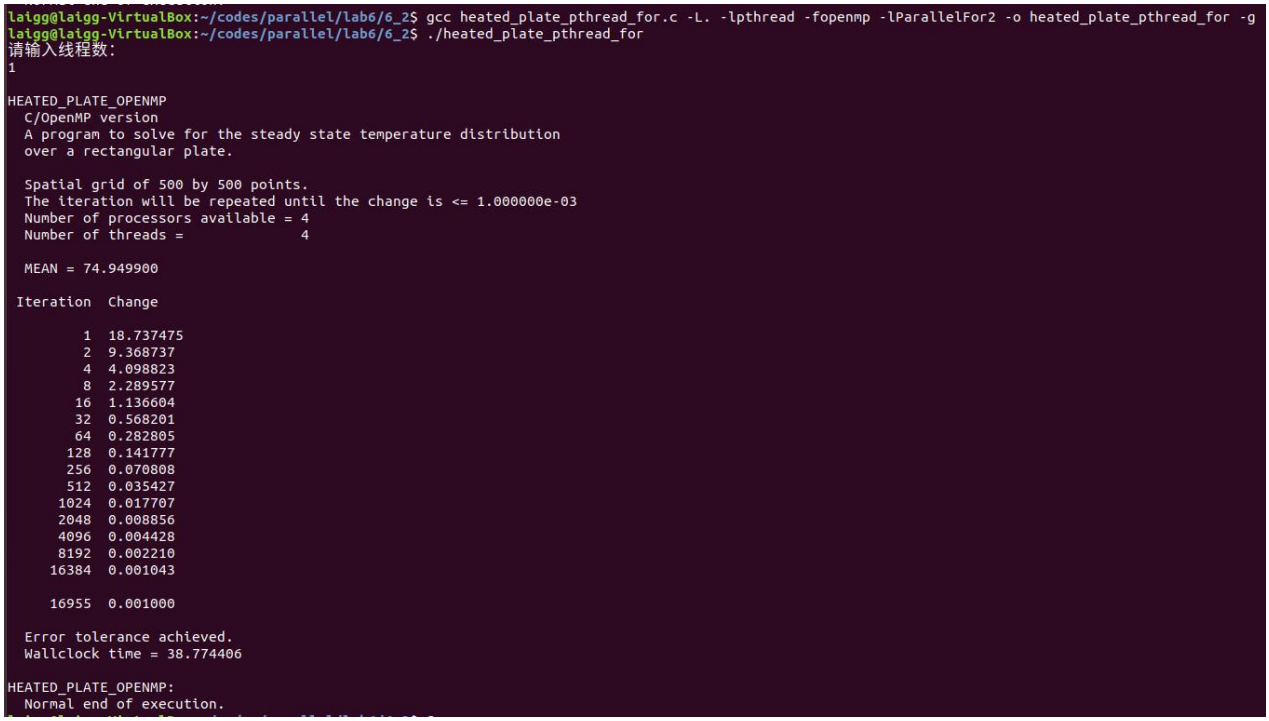
编译：

```
1.  gcc heated_plate_pthread_for.c -L. -lpthread -fopenmp -lParallelFor2 -o heated_plate_pthread_for -g
```

运行：

```
./heated_plate_pthread_for
```

下为 1 个线程运行截图：



pthread_for 不同线程下的执行时间：

线程数	1	2	4	8	16
执行时间(s)	38.774406	22.834356	20.575161	26.394939	39.746465

原代码执行时间为：14.925375s。

分析：

①可以看到，从 1 个线程增加到 4 个线程的过程中，执行时间逐渐提高，这是因为设备具有 4 个处理器，因此随着线程数的提高，资源能够得到充分利用，并行度提高，效率提高。而随着线程数的继续增加，可以看到执行时间不减反增，这是因为只有 4 个处理器，8/16 个线程会对资源进行竞争，加上随着线程数的增加，使用互斥锁的频率增加，线程的创建和销毁成本也随之增加，因此当线程增加到 8 个乃至 16 个时，并行度的提高远比不上这些原因带来的成本的提高。

②此外，对比原代码，由于使用 pthread，增加了线程的创建、销毁、互斥锁的使用以及可能出现的竞争现象，而原代码根据硬件条件合理地分配了不同步骤的线程数并进行适当调整，资源得到了最有效的利用，因此使用 pthread_for 远不及原来使用 openmp 的效率。

4. 实验感想（300 字左右）

在本次实验中我学习了解了如何使用 pthread 来模仿 OpenMP 的 omp_parallel_for 的并行 for 循环分解、分配及执行机制，对于 pthread 编程更加熟悉且对于 OpenMP 的 omp_parallel_for 的并行 for 循环分解、分配及执行机制有了更深入的理解。此外在本次实验中，我还学习了解了如何在 Linux 系统下生成动态链接库(.so 文件)，并使用它。