中山大学计算机院本科生实验报告

（2024学年秋季学期）

课程名称：高性能计算程序设计

实验	openmp实现并行程序设计	专业(方向)	信息与计算科学
学号	22336313	姓名	郑鸿鑫
Email	zhenghx57@mail2.sysu.edu.cn	完成日期	2024/11/15

1. 实验目的

通过OpenMP并行计算框架实现并优化通用矩阵乘法算法，掌握不同调度策略对并行程序性能的影响，并基于Pthreads库构建自定义的并行for循环机制。此外，学习如何在Linux系统中创建动态链接库，并将其应用于并行程序中，以提高程序的模块化和重用性。

2. 实验过程和核心代码

> 子任务1 通过OpenMP实现通用矩阵乘法（Lab1）的并行版本，OpenMP并行线程从1增加至8，矩阵规模从512增加至2048。

关键函数如下（完整代码详见code文件夹）：

```c
float** build_matrix(int m, int n) {
    float** matrix = (float**)malloc(m * sizeof(float*));
    #pragma omp parallel for
    for (int i = 0; i < m; i++) {
        matrix[i] = (float*)malloc(n * sizeof(float));
    }
    return matrix;
}
void fill_matrix(int m, int n, float** A) {
    #pragma omp parallel for
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            A[i][j] = (float)(rand() % 10); // 生成0到10之间的随机浮点数
        }
    }
}
void multiply_matrix(float** A, float** B, float** C, int m, int n, int k) {
    #pragma omp parallel for
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < k; j++) {
            C[i][j] = 0.0;
            for (int p = 0; p < n; p++) {
                C[i][j] += A[i][p] * B[p][j];
            }
        }
    }
}
```

> 子任务2 分别采用OpenMP的默认任务调度机制、静态调度schedule(static, 1)和动态调度schedule(dynamic,1)的性能，实现#pragma omp for，并比较其性能。

关键函数如下（完整代码详见code文件夹）：

```c
void multiply_matrix(float** A, float** B, float** C, int m, int n, int k) {
    #pragma omp parallel for
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < k; j++) {
            C[i][j] = 0.0;
            for (int p = 0; p < n; p++) {
                C[i][j] += A[i][p] * B[p][j];
            }
        }
    }
}
void multiply_matrix_static(float** A, float** B, float** C, int m, int n, int k) {
    #pragma omp parallel for schedule(static, 1)
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < k; j++) {
            C[i][j] = 0.0;
            for (int p = 0; p < n; p++) {
                C[i][j] += A[i][p] * B[p][j];
            }
        }
    }
}
void multiply_matrix_dynamic(float** A, float** B, float** C, int m, int n, int k) {
    #pragma omp parallel for schedule(dynamic, 1)
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < k; j++) {
            C[i][j] = 0.0;
            for (int p = 0; p < n; p++) {
                C[i][j] += A[i][p] * B[p][j];
            }
        }
    }
}
//main函数部分代码:
// Default scheduling
double start_default = omp_get_wtime();
multiply_matrix(A, B, C, m, n, k);
double end_default = omp_get_wtime();
printf("Default scheduling time: %lf s\n", end_default - start_default);
```

```c
// Static scheduling
double start_static = omp_get_wtime();
multiply_matrix_static(A, B, C, m, n, k);
double end_static = omp_get_wtime();
printf("Static scheduling time: %lf s\n", end_static - start_static);

// Dynamic scheduling
double start_dynamic = omp_get_wtime();
multiply_matrix_dynamic(A, B, C, m, n, k);
double end_dynamic = omp_get_wtime();
printf("Dynamic scheduling time: %lf s\n", end_dynamic - start_dynamic);
```

> 子任务3 构造基于Pthreads的并行for循环分解、分配和执行机制。

步骤1

编写parallel.h头文件，声明parallel_for函数，为后续可以给矩阵乘法程序调用

```c
// parallel.h
#ifndef PARALLEL_H
#define PARALLEL_H
void parallel_for(int start, int end, int increment, void *(*functor)(void*), void *arg, int num_t
#endif // PARALLEL_H
```

步骤2

编写parallel.c文件，补充parallel_for函数的具体实现,完整代码如下所示：

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include "parallel.h"
typedef struct {
    int start;
    int end;
    int increment;
    void*(*functor)(void*);
    void* arg;
} ThreadArgs;
void* thread_func(void* args) {
    ThreadArgs* t = (ThreadArgs*)args;
    for (int i = t->start; i < t->end; i += t->increment) {
        t->functor(t->arg); // 直接调用 functor
    }
    free(t); // 释放分配的参数
    return NULL;
}
void parallel_for(int start, int end, int increment, void*(*functor)(void*), void* arg, int num_th
    if (num_threads <= 0) {
        fprintf(stderr, "Invalid number of threads: %d\n", num_threads);
        exit(EXIT_FAILURE);
    }
    pthread_t threads[num_threads];
    int range = (end - start) / num_threads;
    for (int i = 0; i < num_threads; i++) {
        ThreadArgs* thread_args = (ThreadArgs*)malloc(sizeof(ThreadArgs));
        thread_args->start = start + i * range;
        thread_args->end = (i == num_threads - 1) ? end : (start + (i + 1) * range);
        thread_args->increment = increment;
        thread_args->functor = functor;
        thread_args->arg = arg;

        if (pthread_create(&threads[i], NULL, thread_func, thread_args) != 0) {
            perror("Failed to create thread");
            exit(1);
        }
    }
```

```
    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }
}
```

步骤3

在Linux系统中将parallel_for函数编译为.so文件,然后让

```
gcc -fPIC -shared -o libparallel.so parallel.c -lpthread
```

步骤4

编写test.c文件，对矩阵乘法中无数据依赖，循环依赖的for循环进行并行化

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "parallel.h"
#define N 512
float A[N][N], B[N][N], C[N][N];
typedef struct {
    int start;
    int end;
} MatrixArgs;
void* matrix_multiply_func(void* args) {
    MatrixArgs* margs = (MatrixArgs*)args;
    for (int i = margs->start; i < margs->end; i++) {
        for (int j = 0; j < N; j++) {
            float sum = 0.0;
            for (int k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
    return NULL; // 不需要释放内存，因为在主线程中管理
}
double get_wall_time() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec + ts.tv_nsec * 1e-9;
}
int main() {
    int num_threads = 8;
    // 初始化矩阵
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = rand() % 10;
            B[i][j] = rand() % 10;
            C[i][j] = 0;
        }
    }
```

```c
    double start_time = get_wall_time();
    printf("Size: %d,Threads: %d, calculating...\n",N,num_threads);
    // 调用 parallel_for
    for (int t = 0; t < num_threads; t++) {
        MatrixArgs* args = (MatrixArgs*)malloc(sizeof(MatrixArgs));
        args->start = t * (N / num_threads);
        args->end = (t == num_threads - 1) ? N : (t + 1) * (N / num_threads);
        parallel_for(args->start, args->end, 1, matrix_multiply_func, args, num_threads);
    }
    double end_time = get_wall_time();
    printf("Matrix multiplication took %.3lf seconds\n", end_time - start_time);
    return 0;
}
```

步骤5

编译test.c文件，并链接当前目录下的libparallel.so和标准的pthread库。将../test目录添加到LD_LIBRARY_PATH环境变量中，这样当运行动态链接的程序时，动态链接器会在这个目录下搜索需要的共享库文件。

```
gcc -o test test.c -L. -lparallel -lpthread
LD_LIBRARY_PATH=../test
```

3. 实验结果

> **子任务1 通过线程从1到8，矩阵大小从512到2048，运行结果如下：**

512X512:

```
n@XiaoxinPro:~/Lab4$ gcc -fopenmp -o MM matrix_multiply.c
n@XiaoxinPro:~/Lab4$ ./MM
512 1
Size:512, threads:1, Matrix_multiply takes 1.964 s
n@XiaoxinPro:~/Lab4$ ./MM
512 2
Size:512, threads:2, Matrix_multiply takes 1.359 s
n@XiaoxinPro:~/Lab4$ ./MM
512 4
Size:512, threads:4, Matrix_multiply takes 0.766 s
n@XiaoxinPro:~/Lab4$ ./MM
512 8
Size:512, threads:8, Matrix_multiply takes 0.546 s
```

1024x1024:

```
n@XiaoxinPro:~/Lab4$ ./MM
1024 1
Size:1024, threads:1, Matrix_multiply takes 19.458 s
n@XiaoxinPro:~/Lab4$ ./MM
1024 2
Size:1024, threads:2, Matrix_multiply takes 13.136 s
n@XiaoxinPro:~/Lab4$ ./MM
1024 4
Size:1024, threads:4, Matrix_multiply takes 7.049 s
n@XiaoxinPro:~/Lab4$ ./MM
1024 8
Size:1024, threads:8, Matrix_multiply takes 4.875 s
```

2048x2048:

```
n@XiaoxinPro:~/Lab4$ ./MM
2048 1
Size:2048, threads:1, Matrix_multiply takes 193.349 s
n@XiaoxinPro:~/Lab4$ ./MM
2048 2
Size:2048, threads:2, Matrix_multiply takes 140.968 s
n@XiaoxinPro:~/Lab4$ ./MM
2048 4
Size:2048, threads:4, Matrix_multiply takes 69.923 s
n@XiaoxinPro:~/Lab4$ ./MM
2048 8
Size:2048, threads:8, Matrix_multiply takes 52.126 s
```

Matrix Size	Threads	Time (s)	Speedup (vs 1 thread)
512x512	1	1.964	1.00
512x512	2	1.359	1.44
512x512	4	0.766	2.56
512x512	8	0.546	3.60
1024x1024	1	19.458	1.00
1024x1024	2	13.136	1.48
1024x1024	4	7.049	2.76
1024x1024	8	4.875	3.98
2048x2048	1	193.349	1.00
2048x2048	2	140.968	1.37
2048x2048	4	69.923	2.76
2048x2048	8	52.126	3.71

结果分析：随着线程数的增加，矩阵乘法的执行时间显著减少，不过性能的提升速度并没有像理

想加速比那么快，这是由于线程管理和通信开销等的限制

> **子任务2 三种不同调度方式的运行时间如下：**

```
n@XiaoxinPro:~/Lab4$ gcc -O3 -fopenmp -o SC schedule.c
n@XiaoxinPro:~/Lab4$ ./SC
Size: 2048,Thread: 8
Default scheduling time: 12.109388 s
Static scheduling time: 12.069830 s
Dynamic scheduling time: 12.009188 s
```

调度方式	运行时间 (秒)
默认调度 (Default)	12.109388
静态调度 (Static)	12.069830
动态调度 (Dynamic)	12.009188

结果分析：
在这个通用矩阵乘法的框架下，三种调度策略的运行时间相差不大，动态调度略优于静态调度，静态调度略优于默认调度。这可能是因为对于矩阵乘法这个进程，迭代时间相对均匀没有较大的差距，所以动态调度和静态调度比较接近，而且为了等待结果时间短，我们开启了-O3编译优化，所以可能使得差距不太明显。

> **子任务3 只给出矩阵大小为512x512的为例，结果如下：**

```
n@XiaoxinPro:~/Lab4$ gcc -o test test.c -L. -lparallel -lpthread
n@XiaoxinPro:~/Lab4$ LD_LIBRARY_PATH=. ./test
Size: 512,Threads: 8, calculating...
Matrix multiplication took 4.123 seconds
```

结果分析：
实验结果展示了使用pthreads库实现并行计算的可行性和有效性。通过自定义的parallel_for函数，实验成功地将矩阵乘法任务并行化，并通过动态库的形式提高了代码的可重用性。

4. 实验感想

通过这次openmp并行程序设计的实验，我深刻体会到了并行计算在处理大规模数据时的显著优势。实验中，我不仅学习了如何使用OpenMP来实现矩阵乘法的并行化，还亲自体验了不同线程

数和调度策略对性能的影响。从512到2048的矩阵规模扩展，以及从单线程到多线程的转变，代码的运行结果让我对并行计算的加速潜力有了更直观的认识。此外，通过将自定义的parallel_for函数编译为.so库并由其他程序调用，我不仅锻炼了系统编程的能力，也加深了对动态链接库工作机制的理解。这次实验不仅提升了我的编程技能，也让我对并行计算有了更全面的认识。