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 **批改人：**

实验	**高性能程序设计3**	**专业（方向）**	**信息与计算科学**
学号	**22336173**		
Email	3133974071@qq.com	完成日期	**2024.11.05**

实验目的

> 了解openmp框架下的并行程序设计；比较openmp三种调度方式的性能；开发和使用linux共享库；

实验过程和核心代码

任务一：openmp下的矩阵乘法并行，

代码见task1.编译指令为gcc -fopenmp task1.c -o task1。运行指令是./task1 4 1024 第一个参数是线程数，第二个参数是矩阵维度。

核心代码如下：

```
            #pragma omp parallel for num_threads(THREAD_NUMS) //default schedule
            for (int i = 0; i < M; ++i) {
                for (int j = 0; j < K; ++j) {
                    C[i * K + j] = 0.0f;
                    for (int k = 0; k < N; ++k) {
                        C[i * K + j] += A[i * N + k] * B[k * K + j]; //由于并行的是最外层循环，所以不需要对C数组的写进行枷锁处理，因为同一个地址只有一个线程会访问，线程内又是串行的
                    }
                }
            }
```

```
(base) hongjie@kemove-ESC8000-G4:~/ml/hw4$ ./task1 4 128
Real time used: 0.0071 seconds
(base) hongjie@kemove-ESC8000-G4:~/ml/hw4$ ./task1 4 1024
段错误 (核心已转储)
```

```
    float *A = (float*)malloc(sizeof(float)*M*N);
    float *B = (float*)malloc(sizeof(float)*N*K);
    float *C = (float*)malloc(sizeof(float)*M*K);
```

在实验的开始出现了报错显示“段错误，核心已转储”，查阅相关资料，显示是栈溢出的问题，因为一开始使用的是直接分配（float A[M][N]），在main函数栈上分配内存的时候发生了栈溢出，导致段错误，查看栈空间的大小（ultimate -s）为8MB,当创建3个2048*2048的float矩阵的时候发生了栈溢出，解决方法可以使用（ultimate -s memory_size）来扩大栈空间，或者直接使用动态分配放在堆上。如上图所示。

任务二：比较openmp的三种调度机制，默认，静态和动态。

函数有三个参数，线程数，矩阵维度和调度模式（0是默认，1是静态，2是动态）

```
#pragma omp parallel for num_threads(THREAD_NUMS) //default schedule
#pragma omp parallel for num_threads(THREAD_NUMS) schedule(static, AVG_NUM)
#pragma omp parallel for num_threads(THREAD_NUMS) schedule(dynamic, 1)
```

默认是openmp的默认调度方式，静态是将任务均分，分配到线程上，优点是调度损耗更少，缺点是可能出现短板效应。动态是将任务分为更小粒度，然后每一个线程，执行完一次任务，若还有任务，则再执行剩下的任务，优点是效率比较高，缺点是调度损耗比较大。

任务三：使用pthreads构建一个“#pragma omp parallel”类似的函数parallel_for

基于pthreads的多线程库提供的基本函数，如线程创建、线程join、线程同步等，构建parallel_for函数，该函数实现对循环分解、分配和执行机制，函数参数包括但不限于(int start, int end, int increment, void *(functor)(void*), void *arg , int num_threads)；其中start为循环开始索引；end为结束索引；increment每次循环增加索引数；functor为函数指针，指向被并行执行的循环代码块；arg为functor的入口参数；num_threads为并行线程数

思路：**子线程要访问堆上的或者主函数栈上的三个矩阵地址，需要传参来访问**，使用arg传参，其他的**increment应该和num_threads相互对应**，否则会出现无法计算的剩余模块。同时注意pthread_create的传参需要num_threads个才能满足不同的start要求。

```
//parallel.c
/*
start:开始index, end:结束index, incre:线程的计算跳步步长，functor：并行函数，arg:矩阵地址之类
*/
int parallel_for(int start, int end, int increment, void *(*functor)(void*), void *arg , int num_threads){
    pthread_t *pthread_set = (pthread_t*)malloc(num_threads * sizeof(pthread_t));
    if(!pthread_set){
        printf("threads malloc failed!\n");
        return 0;
    }
    int end_i = end;
    // printf("%p\n", functor);
    struct for_index index_i[num_threads];
    for (int i = 0; i < num_threads; i++) {
        index_i[i].start= i;
        index_i[i].end = end_i;
        index_i[i].increment = increment;
        index_i[i].memory = arg;
        pthread_create(&pthread_set[i], NULL, functor, (void*) &index_i[i]); //开启并行
```

```
    }


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

    return 1;
}
```

```
//parallel.h
#ifndef PARARREL_H
#define PARARREL_H

struct for_index{
    int start;
    int end;
    int increment;
    void * memory;
};

int parallel_for(int start, int end, int increment, void *(*functor)(void*), void
*arg , int num_threads);
#endif
```

```
//task.c 也就是主函数
void * matrix_multiply_s(void * args) {
    int N = SIZE, K = SIZE, M = SIZE;
    struct for_index * index = (struct for_index *) args;
    double *A = index->memory;
    double *B = A + M * N;
    double *C = B + N * K; //获取堆上的数组指针
    // clock_t start, end;
    // start = clock();
    int increment = index->increment;
    // printf("%d %d %d\n", index->start, index->end, index->increment);
    for (int i = index->start; i < index->end; i+=increment) {
        for (int k = 0; k < N; k++){
            for (int j = 0; j < K; j++){
                C[i*K+j] += A[i*N+k] * B[k*K+j];
                // if (i == 0&& k == N-1 && j==K-1) printf("%.8f, %.4f,
%.4f\n",C[i*K+j], A[i*N+k], B[k*K+j]);
            }
        }
    }
int main(){
    .......
    double *A = (double *)malloc(sizeof(double) * M * N + sizeof(double) * N * K
+ sizeof(double) * M * K);
    double *B = A + M * N;
    double *C = B + N * K;
    ......
    parallel_for(0, M, increment, matrix_multiply_s, (void*)A, THREAD_NUMS);
```

```
}
```

编译脚本：

```
gcc -c -fPIC parallel.c -o parallel.o
gcc -shared -o libparallel_for.so parallel.o
cp libparallel_for.so ../lib
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/hongjie/ml/hw4/lib
gcc -o task3 task3.c -lparallel_for -L /home/hongjie/ml/hw4/lib
```

实验结果

实验环境： Intel(R) Xeon(R) Gold 6133 CPU @ 2.50GHz 40核心 80线程

任务一

核心数 / 矩阵维度	128 (秒)	256 (秒)	512 (秒)	1024 (秒)	2048 (秒)
1	0.0215	0.0867	0.6150	4.8933	39.9713
2	0.0138	0.0496	0.3235	2.4603	19.6516
4	0.0071	0.0344	0.1714	1.2461	9.8732
8	0.0040	0.0239	0.0961	0.6464	5.1082
16	0.0031	0.0148	0.0556	0.3235	3.4710
32	0.0029	0.0122	0.0400	0.1799	1.2533
64	0.0033	0.0093	0.0349	0.1678	1.2869
80	0.0438	0.0518	0.0502	0.2178	1.4757
128	0.0102	0.0158	0.0472	0.1699	1.2232

任务二

下面三个矩阵分别对应的是采用默认调度，静态调度（static， 1），动态调度（dynamic, 1）的计算时间表格。

核心数 / 矩阵维度	128 (秒)	256 (秒)	512 (秒)	1024 (秒)	2048 (秒)
1	0.0224	0.0804	0.6193	4.8899	39.2544
2	0.0130	0.0504	0.3236	2.5807	20.1340
4	0.0072	0.0338	0.1678	1.2592	9.9143
8	0.0041	0.0235	0.1038	0.7983	5.0594
16	0.0029	0.0225	0.0802	0.5324	2.5559

核心数 / 矩阵维度	128 (秒)	256 (秒)	512 (秒)	1024 (秒)	2048 (秒)
32	0.0033	0.0092	0.0377	0.1724	1.3960
64	0.0055	0.0114	0.0380	0.2082	1.4820
80	0.0453	0.0330	0.0444	0.2086	1.3864
128	0.0054	0.0157	0.0540	0.1666	1.2172

核心数 / 矩阵维度	128 (秒)	256 (秒)	512 (秒)	1024 (秒)	2048 (秒)
1	0.0197	0.0893	0.6136	5.0615	39.3100
2	0.0138	0.0525	0.3220	2.5460	19.9937
4	0.0073	0.0339	0.1622	1.2337	9.9558
8	0.0038	0.0224	0.0918	0.6403	5.1118
16	0.0033	0.0150	0.0554	0.3239	2.6715
32	0.0039	0.0106	0.0370	0.1746	1.3429
64	0.0056	0.0113	0.0358	0.1636	1.2968
80	0.0489	0.0318	0.0565	0.2460	2.1040
128	0.0067	0.0160	0.0374	0.2325	1.2229

核心数 / 矩阵维度	128 (秒)	256 (秒)	512 (秒)	1024 (秒)	2048 (秒)
1	0.0218	0.0915	0.6252	5.1313	39.3795
2	0.0096	0.0571	0.3281	2.5053	19.8531
4	0.0070	0.0340	0.1696	1.2620	9.9337
8	0.0029	0.0222	0.0971	0.6655	5.0953
16	0.0023	0.0142	0.0629	0.3314	2.5430
32	0.0034	0.0090	0.0397	0.1748	1.3294
64	0.0170	0.0050	0.0505	0.1891	1.4367
80	0.0326	0.0551	0.0514	0.1784	1.1311
128	0.0050	0.0141	0.0401	0.1613	1.1164

任务三：

根据表格，可以看到基本达到了Openmp的加速效果

行为核心数目 \ 矩阵维度	128	256	512	1024	2048
1	0.0238s	0.0929s	0.6332s	4.9298s	39.1135s

行为核心数目 \ 矩阵维度	128	256	512	1024	2048
2	0.0150s	0.0556s	0.3266s	2.4789s	19.9871s
4	0.0072s	0.0361s	0.1931s	1.3260s	10.0075s
8	0.0045s	0.0249s	0.1059s	0.6412s	4.9932s
16	0.0034s	0.0156s	0.0575s	0.3334s	2.6174s
32	0.0037s	0.0100s	0.0379s	0.1960s	1.3698s
64	0.0058s	0.0116s	0.0377s	0.1713s	1.2428s
80	0.0064s	0.0097s	0.0344s	0.2262s	1.3281s
128	0.0091s	0.0091s	0.0399s	0.1661s	1.3657s

实验感想

> //可以写写过程中遇到的问题，你是怎么解决的。以及可以写你对此次实验的一些理解.....

float数据类型经过大量的+*操作，会丢失数据精度，更多的有待查询资料。

Openmp在开启并行的时候，原来main函数栈上的元素如果被声明为共享，子线程会直接共享这个指针，如果是私有的变量，该指针在子线程里会被副本替代。

子线程访问堆上的函数需要使用传参或者声明堆上的元素是全局变量。

动态库的路径说明可以使用export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/hongjie/ml/hw4/lib或者在编译的时候加上参数-L directory来指定。