中山大学计算机学院

本科生实验报告

（2025学年春季学期）

课程名称: 并行程序设计

实验	计	专业(方向)	计算机科学与技术
学号	--	姓名	--
Email	--	完成日期	2025.4.19

实验目的

- 使用Pthreads多线程实现并行矩阵乘法
- 使用Pthreads多线程实现并行数组求和

实验过程和核心代码

矩阵乘法

将两个数组和结果数组设置为一维全局指针, 减少线程参数传递.
在获取到输入的线程数和矩阵规模后分配空间并随机初始化矩阵.

```
int thread_num = 1;
int matrix_size = 128;
if(argc >= 2) {
    try {
        thread_num = std::stoi(argv[1]);
    } catch (std::invalid_argument const& ex) {
        Usage(argv[0]);
    }
}
if(argc >= 3) {
    try {
        matrix_size = std::stoi(argv[2]);
    } catch (std::invalid_argument const& ex) {
        Usage(argv[0]);
    }
}
mat1 = new double[matrix_size * matrix_size];
mat2 = new double[matrix_size * matrix_size];
ans = new double[matrix_size * matrix_size]();
getRandomMat(mat1, matrix_size, matrix_size);
getRandomMat(mat2, matrix_size, matrix_size);
```

创建一个结构体作为线程函数的参数传递

```
struct matMulPara{
    int rows;
    int n;
    int thread_no;
    matMulPara(){rows = 0; n = 0; thread_no = 0;}
    matMulPara(int _rows, int _n, int _thread_no){
        rows = _rows;
        n = _n;
        thread_no = _thread_no;
    }
};
```

传递各行需要计算的mat1行数, 矩阵规模和线程编号.
创建各个参数后创建各个线程, 指定执行的函数并传递参数.
最后回收线程

```
int rows_each_thread = matrix_size / thread_num;
pthread_t* threads = new pthread_t[thread_num];

// 创建参数
matMulPara* threads_para = new matMulPara[thread_num];
for(int i = 0; i < thread_num; ++i) {
    threads_para[i] = matMulPara(rows_each_thread, matrix_size, i);
}

// 创建线程
auto begin = chrono::high_resolution_clock::now();
for(int i = 0; i < thread_num; ++i) {
    pthread_create(&threads[i], NULL, matMul, static_cast<void*>(&threads_para[i]));
}
// 回收线程
for(int i = 0; i < thread_num; ++i) {
    pthread_join(threads[i], NULL);
}
```

进行矩阵乘法的函数:

```
void* matMul(void* args) {
    matMulPara* para = (matMulPara*) args;
    int rows = para->rows;
    int n = para->n;
    int begin = para->thread_no * rows;
    int end = (para->thread_no + 1) * rows;
    for(int i = begin; i < end; ++i) {
        for(int j = 0; j < n; ++j) {
            for(int k = 0; k < n; ++k)
                ans[i*n + j] += mat1[i*n + k] * mat2[k*n + j];
        }
    }
}
```

最后回收分配的内存

```
delete [] mat1;
delete [] mat2;
delete [] ans;
delete [] threads;
delete [] threads_para;
```

在程序中将结果输出到文件中, 并编写一个脚本执行不同线程数和矩阵规模的程序.

```
program_name="matrixMul_pthread"

./$program_name 1 128
./$program_name 1 256
...
...
./$program_name 16 1024
./$program_name 16 2048
```

数组求和

声明一个全局arr指针表示要求和的数组, 一个int型ans表示求和结果
在获取到输入的线程数和数组规模后分配空间并随机初始化数组.

```
array_size *= (int)1e6;
arr = new int[array_size];
getRandomArray(arr, array_size);
```

创建一个结构体作为线程函数的参数传递

```
struct arraySumPara{
    int n;
    int total;
    int thread_no;
    arraySumPara(){n = 0; total = 0; thread_no = 0;}
    arraySumPara(int _n, int _total, int _thread_no){
        n = _n;
        total = _total;
        thread_no = _thread_no;
    }
};
```

传递各行需要计算的数组数量, 数组规模和线程编号.
创建各个参数后创建各个线程, 指定执行的函数并传递参数.
最后回收线程

```
int n_each_thread = array_size / thread_num;
pthread_t* threads = new pthread_t[thread_num];

// 创建参数
arraySumPara* threads_para = new arraySumPara[thread_num];
for(int i = 0; i < thread_num; ++i) {
    threads_para[i] = arraySumPara(n_each_thread ,array_size, i);
}

// 创建线程
auto begin = chrono::high_resolution_clock::now();
for(int i = 0; i < thread_num; ++i) {
    pthread_create(&threads[i], NULL, arraySum, static_cast<void*>(&threads_para[i]));
}
// 回收线程
for(int i = 0; i < thread_num; ++i) {
    pthread_join(threads[i], NULL);
}
```

在进行求和时, 由于会出现数组规模无法整除线程数的情况, 所以最后一个线程处理的最后数组下标要小于数组长度

```
int end = min((para->thread_no + 1) * n, para->total);
```

初始实现求和函数时, 各线程直接将数组项加到ans中, 这时需要一个互斥锁lock来避免同时写sum, 但这样冲突发生很频繁导致性能很低(求和函数注释部分v1). 随后改进v2, 每个线程设置一个临时变量, 先将要求和的数组项加到临时变量, 再利用互斥锁将临时变量加到ans中.
求和函数如下:

```
void* arraySum(void* args) {
    arraySumPara* para = (arraySumPara*) args;
    int n = para->n;
    int begin = para->thread_no * n;
    int end = min((para->thread_no + 1) * n, para->total);
    int tmp = 0;

    // v1
    // for(int i = begin; i < end; ++i) {
    //     pthread_mutex_lock(&lock);
    //     ans += arr[i];
    //     pthread_mutex_unlock(&lock);
    // }

    // v2
    for(int i = begin; i < end; ++i) {
        tmp += arr[i];
    }
    pthread_mutex_lock(&lock);
    ans += tmp;
    pthread_mutex_unlock(&lock);
    return NULL;
}
```

同时利用`std::accumulate()`来进行数组求和作为正确答案, 检查计算结果是否一致. 最后回收分配的空间.

```
int true_ans = std::accumulate(arr, arr + array_size, 0);
delete [] arr;
delete [] threads;
delete [] threads_para;
```

在程序中将结果输出到文件中, 并编写一个脚本执行不同线程数和数组规模的程序.

```
program_name="arraySum_pthread"

./$program_name 1 1
./$program_name 1 2
...
...
./$program_name 16 64
./$program_name 16 128
```

实验结果

虚拟机处理器个数: 4
时间单位: s

矩阵相乘

相乘矩阵mat1, mat2的大小都为: 矩阵规模 × 矩阵规模

线程数	矩阵规模				
	128	256	512	1024	2048
1	0.0083	0.0647	0.6850	7.4739	146.5982
2	0.0040	0.0333	0.3833	3.7757	63.4742
4	0.0020	0.0167	0.2175	2.1382	30.3483
8	0.0022	0.0167	0.2172	2.1205	41.8074
16	0.0023	0.0204	0.2803	2.5543	46.0103

结果分析
可以看到在进程数小于等于4时, 加速比近似为线程数比, 当线程数大于4时, 加速比几乎为1, 在矩阵规模较大时性能还更低.
理论上来说虚拟机有4个处理器, 那么应该可以同时执行8个线程, 应当在进程数为8时达到最佳的性能, 但是结果看来8线程与4线程的性能相近, 还略低一些. 考虑虚拟机其他应用(vscode ssh远程连接服务等)可能占据了一些处理器核心, 使某些线程进行乘法时要受cpu调度, 不能完全占有处理器核心, 所以发生这种情况.
另外可以横向对比之前多进程的实验结果, 发现本次实验结果在相同矩阵规模和线程数/进程数的情况下运行时间更短. 这是利用了**线程之间共享所属进程的内存资源**, 将矩阵设置为了全局变量, 线程之间共享, 可以减少消息传递的开销. 另外线程之间的调度比进程间的调度开销要更小.

数组求和

数组规模单位: 10^6

v1

线程数	数组规模							
	1	2	4	8	16	32	64	128
1	0.0171	0.0293	0.0712	0.1228	0.2317	0.4629	0.8977	1.8080
2	0.0579	0.1250	0.1996	0.5284	1.1069	2.2050	4.2751	9.4494
4	0.0647	0.1166	0.2285	0.5095	1.0666	2.2473	3.5986	8.3779
8	0.0475	0.0850	0.1782	0.3766	0.7605	1.4357	2.8097	5.8471
16	0.0361	0.0630	0.1244	0.2115	0.3450	0.7125	1.3614	2.4002

v2

线程数	数组规模							
	1	2	4	8	16	32	64	128
1	0.0023	0.0038	0.0087	0.0145	0.0288	0.0629	0.1154	0.2309

2	0.0011	0.0026	0.0038	0.0082	0.0143	0.0319	0.0600	0.1198
4	0.0007	0.0013	0.0034	0.0044	0.0082	0.0177	0.0300	0.0628
8	0.0007	0.0017	0.0032	0.0064	0.0120	0.0217	0.0313	0.0634
16	0.0009	0.0015	0.0025	0.0041	0.0076	0.0165	0.0341	0.0615

结果分析

同上, 在线程数小于等于4时加速比近似线程数比.

此外, v2的性能要明显优于v1, 因为v1版本线程每次加一个数都要请求锁来直接加到`ans`中. 性能损失非常大, 而v2每个线程只需要计算完毕后请求一次锁将结果加到`ans`中, 所以速度快很多.

当线程数为16超过虚拟机核数时, 需要进行线程的切换, 有额外开销. 但是这里16线程时在某些数组规模下性能甚至优于4线程. 考虑cpu分配时间片为几毫秒到几十毫秒, 那么在这种线程数多, 每个线程计算量较小情况下, 分给各个**线程的计算量可能在一个时间片内都能够完成**, 因此可以忽略线程切换的额外开销. 这样即使线程数大于虚拟机核数性能也不会降低.

实验感想

在进行多线程编程时, 可以合理利用线程具有共享进程内存空间的特性, 简化编程的复杂性. 在多线程与多进程之间抉择时, 要根据实际情况进行选择. 像这种计算任务还是使用多线程更优, 大量计算任务要耗费cpu, 要尽量减少调度的开销.

另外, 在多线程编程时要注意共享资源避免同时写, 因此还要考虑加锁来保证多线程的正确性.