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	基于OpenMP的并行矩阵乘法	专业(方向)	计算机科学与技术
学号	--	姓名	--
Email	--	完成日期	2025.5.2

实验目的

1. 使用OpenMP实现并行通用矩阵乘法，并通过实验分析不同进程数量、矩阵规模、调度机制时该实现的性能
2. 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

实验过程和核心代码

OpenMP通用矩阵乘法

首先使用`omp_set_num_threads`来设置线程数

```
// set number of threads
omp_set_num_threads(thread_num);
```

在矩阵乘法中设置对应的OpenMP指令, 使用不同的调度方式来执行矩阵乘法. 分别执行并计时得到当前**线程数量**和**矩阵规模**下三种调度方式的执行时间

```
// default
#pragma omp parallel for collapse(2)

// static
//#pragma omp parallel for collapse(2) schedule(static)

// dynamic
//#pragma omp parallel for collapse(2) schedule(dynamic)
for(int i = 0; i < n; ++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];
        }
    }
}
```

这里使用了`collapse(2)`, 表示同时线程化接下来的两层循环. 即将两个相邻的嵌套循环合并成一个更大的迭代空间
如下:

```
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {

    }
}
// 使用 collapse(2)后, 迭代次数为 M * N, 展开为一维循环
for (int k = 0; k < M * N; k++) {
    int i = k / N;
    int j = k % N;

}
```

这样做的好处是

1. 如果外层循环很小, 不使用合并会导致有一些线程空闲
2. 相当于把M个大任务分解为M*N个小任务, 能够更均衡的将任务分配给线程

另外, 在了解到OpenMP的`shared`后, 我思考了是否能共享一些数据来加快效率. 首先`mat1`和`mat2`是必定能够共享的, 因为这**两个矩阵只需要读取, 不需要写值**. 然后考虑`ans`矩阵, 当我们并行外面两层循环后, 第三层循环一定是由某一个线程完整完成的, 也就是说`ans`的**某个位置的值只由一个线程计算完成**, 不会发生数据竞争情况. 所以`ans`也可以进行共享.
v2版本为共享数据版本:

```
// default
#pragma omp parallel for collapse(2) shared(ans, mat1, mat2)

// static
//#pragma omp parallel for collapse(2) schedule(static) shared(ans, mat1, mat2)

// dynamic
//#pragma omp parallel for collapse(2) schedule(dynamic) shared(ans, mat1, mat2)
for(int i = 0; i < n; ++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];
        }
    }
}
```

构造基于Pthreads的并行for循环分解、分配、执行机制

首先编写头文件`parallel_for.h`用于声明函数接口

```
#pragma once
void parallel_for(int start, int end, int inc, void* (*func)(int, void*), void* arg, int num_threads);
```

然后编写`parallel_for.cpp`实现并行逻辑.
先通过`start`, `end`, `inc`以及`num_therad`来计算每个线程需要执行的循环.

```
// 迭代次数
int iters = (end - start + inc - 1) / inc;
// 每个线程处理的迭代次数
int iters_per_thread = iters / num_threads;
```

然后创建线程来执行被分配的循环迭代. 考虑传入的`func`包含int型的参数index, 以及其他参数arg. 我们创建线程时先通过`threadFunc`来执行for循环, 然后在循环中依次调用`func`来执行单个迭代的逻辑. 由于创建线程时只能传入一个void*参数, 所以再创建一个结构体, 包含线程的id, start, end等信息, 再使用一个`void*`指针来存储传入的信息以及一个函数指针指向任务函数.

```
struct ThreadArgs{
    int thread_id;
    int start;
    int end;
    int inc;
    void* (*func)(int, void*);
    void* arg;
};

void* threadFunc(void* args) {
    ThreadArgs* thread_args = (ThreadArgs*)args;
    std::cout << "Thread " << thread_args->thread_id << ": start and end is " << thread_args->start << "," << thread_args->end << std::endl;
    for (int i = thread_args->start; i < thread_args->end; i += thread_args->inc) {
        thread_args->func(i, thread_args->arg);
    }

    delete thread_args;
    return nullptr;
}


// [start, end)
for (int i = 0; i < num_threads; ++i) {
    int thread_start = start + i * iters_per_thread * inc;
    int thread_end = (i == num_threads - 1) ? end : thread_start + iters_per_thread * inc;
    ThreadArgs* thread_args = new ThreadArgs{i, thread_start, thread_end, inc, func, arg};
    pthread_create(&threads[i], NULL, threadFunc, thread_args);
}
```

将`parallel.cpp`编译为动态链接库

```
g++ parallel_for.cpp -fPIC -shared -o libparallel.so -lpthread
```

先使用示例任务数组相加`use_parallel_for.cpp`来进行验证
编译链接

```
# compile with libparallel.so
g++ use_parallel_for.cpp -L. -lparallel -o use_parallel_for
# 将当前目录加入库搜索路径
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

执行结果如下

```
bo@bo-VirtualBox:~/vsc/parallel/lab5/parallel_for$ ./use_parallel_for
[5,5,5,1,9,10,4,4,7,3,]
[4,1,6,8,3,7,5,6,4,1,]
Thread 1: start and end is 5,10
Thread 0: start and end is 0,5
[9,6,11,9,12,17,9,10,11,4,]
```

运行结果正确.

然后使用paralle_for来实现通用矩阵乘法, 只并行化最外层循环

```
struct MatMulArgs {
    float* A;
    float* B;
    float* C;
    int n;
    int k;
};

void* matMul(int i, void* arg) {
    MatMulArgs* args = static_cast<MatMulArgs*>(arg);
    int n = args->n;
    int k = args->k;

    for(int j = 0; j < n; ++j) {
        for(int l = 0; l < k; ++l) {
            args->C[i*n + j] += args->A[i*k + l] * args->B[l*n + j];
        }
    }
    return nullptr;
}

parallel_for(0, m, 1, matMul, arg, thread_num);
```

执行结果见`matrixMul_result.txt`

实验结果

虚拟机处理器个数: 4
时间单位: s

OpenMP通用矩阵乘法

相乘矩阵mat1, mat2的大小都为: 矩阵规模 × 矩阵规模
运行结果分别在`matrixMul_result.txt`和`matrixMul_resultv2.txt`中

不使用共享数据:

线程数	调度方式	矩阵规模				
		128	256	512	1024	2048
1	默认调度	0.0131	0.1198	1.2982	11.6349	195.5958
	静态调度	0.0140	0.1564	1.2175	11.7686	196.3355

	动态调度	0.0143	0.1727	1.2208	11.2357	207.6958
2	默认调度	0.0066	0.0585	0.6112	6.1813	86.9964
	静态调度	0.0066	0.0670	0.5932	6.1616	86.6576
	动态调度	0.0164	0.1411	1.4086	12.6461	133.5897
4	默认调度	0.0038	0.0393	0.3753	3.5835	48.6474
	静态调度	0.0034	0.0449	0.3736	3.6845	48.7326
	动态调度	0.0180	0.1485	1.4175	11.7630	105.3714
8	默认调度	0.0038	0.0393	0.3949	3.5340	54.1694
	静态调度	0.0035	0.0389	0.3847	3.6018	56.6399
	动态调度	0.0229	0.1546	1.4207	12.9059	106.0736
16	默认调度	0.0038	0.0423	0.3999	3.6740	57.8780
	静态调度	0.0033	0.0377	0.3925	3.6789	57.7987
	动态调度	0.0211	0.1535	1.4214	11.9398	110.2016

使用共享数据

线程数	调度方式	矩阵规模				
		128	256	512	1024	2048
1	默认调度	0.0123	0.1068	1.3332	10.8154	191.7611
	静态调度	0.0121	0.1084	1.1535	10.2375	192.2813
	动态调度	0.0120	0.1145	1.1825	10.1733	191.2949
2	默认调度	0.0063	0.0536	0.5720	5.4719	76.7238
	静态调度	0.0061	0.0550	0.6424	5.4689	77.9113
	动态调度	0.0161	0.1276	1.1974	11.4968	118.7130
4	默认调度	0.0033	0.0307	0.3448	3.2286	42.4377
	静态调度	0.0030	0.0371	0.3561	3.3509	42.6698
	动态调度	0.0166	0.1432	1.2033	10.6415	93.7975
8	默认调度	0.0033	0.0413	0.3587	3.0520	41.5974
	静态调度	0.0032	0.0340	0.3645	3.0734	41.7401
	动态调度	0.0177	0.1454	1.2044	10.5947	95.8754
16	默认调度	0.0035	0.0335	0.3777	3.6773	40.2099
	静态调度	0.0031	0.0393	0.3559	3.6354	39.3034
	动态调度	0.0173	0.1458	1.2402	10.5386	93.7262

首先可以看到在线程数小于等于处理器核心数(4), 矩阵规模较大且调度方式不是动态调度时, 加速比接近理论值. 但是当线程数大于4时, 运行时间不但没有减少反而有所提升. 这是由于线程调度产生了额外的开销.

观察调度方式可以看到默认调度与静态调度的结果非常接近, 这是由于在使用**默认调度时, 系统就是按照静态方式调度**的. 观察动态调度的结果发现只有线程数为1时执行时间与其他调度方式接近, 线程数大于1时明显要慢于其他调度方式. 这是由于动态调度每次只分配一个循环给某个线程, 当其他线程启动或执行完毕后再分配一次迭代. **需要分配迭代的次数非常多, 所以分配的开销也大**, 导致运行速度最慢.

最后观察v1和v2, 发现v2共享数据版本比v1要快, 而且在矩阵规模越大时快的绝对值也越大. 所以在不会发生数据竞争时使用`shared`时对性能有明显提升.

构造基于Pthreads的并行for循环分解、分配、执行机制

由于本实验重点不在于分析线程数和矩阵规模等因素对运行时间的影响. 所以在程序中只使用了线程数为4, 矩阵规模为1024 * 1024来测试.
对比之前实验中对应线程数和矩阵规模的结果:

方法	parallel_for	OpenMP	Pthread	MPI(Send,Recv)	MPI(Scatter,Gather)
运行时间	2.1832	3.2286	2.1382	4.58821	2.8228

可以看到相对于直接使用Pthread会慢一些, 这是因为parallel_for多进行了一些封装, 运行过程有一些额外开销(如个迭代都需要函数调用, 需要对参数再进行一层封装等). 但是优点也在于此, 能够更方便的使用. 而且由于动态链接库的特性, 当parallel_for需要修改时, 只需要重新编译为动态链接库, 使用此链接库代码无需重新编译.

实验感想

首先学习了OpenMP的基本使用. 了解了 `collapse` 和 `shared` 的作用, 在特定场景下应用 `collapse` 和 `shared` 可以提高程序的性能. 还了解了不同调度方式的调度原理, 在设计并行时也要根据情景选择合适的调度策略. 然后学习了使用pthread来实现parallel_for并编译为动态链接库文件以便其他代码使用. 了解到动态链接库的特性与优点.