实验3-Pthreads并行矩阵乘法与数组求和

学号	姓名
20319045	刘冠麟

实验1

实验要求

使用Pthreads实现并行矩阵乘法，并通过实验分析其性能。

输入:

m, n, k三个整数，每个整数的取值范围均为$[128, 2048]$

问题描述:

随机生成$m \times n$的矩阵A及$n \times k$的矩阵B，并对这两个矩阵进行矩阵乘法运算，得到矩阵C.

输出:

A, B, C三个矩阵，及矩阵计算所消耗的时间t。

要求:

1. 使用Pthread创建多线程实现并行矩阵乘法，调整线程数量（1-16）及矩阵规模（128-2048），根据结果分析其并行性能（包括但不限于，时间、效率、可扩展性）。
2. 选做：可分析不同数据及任务划分方式的影响。

关键代码解释

代码中使用`pthread`库实现了多线程计算，实现了多线程程序。使用`pthread_create`创建线程，`pthread_join`等待线程完成。

矩阵乘法

```
// 矩阵乘法的线程函数
void *matrix_multiply(void *arg) {
    long pid = (long)arg;
    int row = M / thread_num;  // 每个线程计算的行数
    int rem = M % thread_num;  // 不能均分的额外行数
    int start_row = pid < rem ? pid * (row + 1) : pid * row + rem;
    int end_row = start_row + (pid < rem ? row + 1 : row);

    for (int i = start_row; i < end_row; ++i) {
        for (int j = 0; j < K; ++j) {
            C[i][j] = 0;
            for (int k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return NULL;
```

```
}
```

使用`matrix_multiply`函数来将要计算的矩阵分配给线程，将线程的 ID作为参数用于计算该线程应处理的矩阵行的范围。然后使用整除(`M / thread_num`)和取余(`M % thread_num`)来决定每个线程应该处理多少行，如果线程 ID 小于余数，则这个线程多处理一行，确保所有行都被平均分配到各个线程中。

最后遍历分配给该线程的每一行`i`，对于结果矩阵`C`的每一列`j`，计算`A[i][k] * B[k][j]`的累加和。

线程创建和同步

main函数中进行了线程创建和线程同步：

```
int main(int argc, char *argv[]) {
    if (argc != 5) {
        cout << "Usage: " << argv[0] << " M N K num_threads" << endl;
        return 1;
    }

    M = atoi(argv[1]);
    N = atoi(argv[2]);
    K = atoi(argv[3]);
    thread_num = atoi(argv[4]);

    initialize_matrices();

    pthread_t *threads = new pthread_t[thread_num];
    clock_t start_time = clock();

    for (int i = 0; i < thread_num; ++i) {
        pthread_create(&threads[i], NULL, matrix_multiply, (void *)i);
    }

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

    clock_t end_time = clock();
    double time_elapsed = static_cast<double>(end_time - start_time) /
CLOCKS_PER_SEC;

    cout << "Using time: " << time_elapsed << " s" << endl;

    delete[] threads;
    free_memory();  // 释放内存

    return 0;
}
```

线程创建：

- 创建`thread_num`个线程。

- 每个线程调用 `matrix_multiply` 函数并传递其线程 ID。

```
    pthread_t *threads = new pthread_t[thread_num];

    for (int i = 0; i < thread_num; ++i) {
        pthread_create(&threads[i], NULL, matrix_multiply, (void *)i);
    }
```

线程同步：

- `pthread_join` 确保所有线程都已完成它们的任务，主线程在所有线程完成后才继续执行。

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

编译运行

编译运行：

-lpthread表示编译器链接 POSIX 线程库：

```
g++ lab3_1.cpp -o lab3_1 -lpthread
```

运行结果如下：

```
(base) liuglin@WIN-6G3ESV0SHI5:/mnt/d/Paper/Communication-theory/并行程序设计与算法/实验作业/lab3$ ./lab3_1 512 512 512
16
Using time: 1.09135 s
(base) liuglin@WIN-6G3ESV0SHI5:/mnt/d/Paper/Communication-theory/并行程序设计与算法/实验作业/lab3$ ./lab3_1 512 512 512
8
Using time: 0.897088 s
```

改变线程数和矩阵规模后可得如下表格：

线程数\矩阵规模	128	256	512	1024	2048
1	0.01942	0.1581	1.585	19.01	187.6
2	0.02158	0.1758	1.667	18.54	182.9
4	0.02103	0.1885	1.754	16.98	154.5
8	0.02010	0.1620	1.789	14.63	148.2
16	0.02532	0.1564	1.695	14.46	147.3

根据表格计算得出加速比和效率如下：

加速比

线程数\矩阵规模	128	256	512	1024	2048
1	1.000	1.000	1.000	1.000	1.000
2	0.900	0.899	0.951	1.025	1.026

线程数\矩阵规模	128	256	512	1024	2048
4	0.923	0.839	0.904	1.120	1.214
8	0.966	0.976	0.886	1.299	1.266
16	0.767	1.011	0.935	1.315	1.274

根据运行时间，分析程序并行性能及扩展性

- **并行性能**
    - 可以看到实验随着进程数的增加加速比的提升并没有达到理想状态（即加速比应等于或接近于进程数）。
    - **对于较小的矩阵**，加速比反而随着线程数的增加而下降，这说明对于更小的矩阵（如128规模），增加进程数反而可能导致效率降低。这表明对于较小的任务，多进程可能不会带来预期的加速效果。**对于较大的任务**加速比的增加也非常有限，远未达到预期。
- **效率**
    - 效率随着进程数的增加而降低。对于较小的矩阵，效率显著下降，可能是因为程序是**弱拓展性**的，数据量不足以使所有进程都充分利用。
- **扩展性**
    - 可以看到随着线程规模的增加，加速比增加，可以说明该程序是**弱拓展性的。**
    - 对于较小矩阵，程序的扩展性较差。
- 在共享内存的情况下pthread通信代价小于MPI，因此计算速度普遍快于mpi。但是数据量增多时mpi通信将大大增加，因此在数据变多时pthread增长速度小于mpi增长速度。

实验二

实验要求

使用Pthreads创建多线程，实现并行数组求和

输入：整数n，取值范围为$[1M, 128M]$

问题描述：随机生成长度为n的整型数组A，计算其元素和$\sum_{i=1}^{n} A_i$

输出：数组A，元素和s，及求和计算所消耗的时间t。

要求：

1. 使用Pthreads实现并行数组求和，调整线程数量（1-16）及数组规模（1M, 128M），根据结果分析其并行性能（包括但不限于，时间、效率、可扩展性）.
2. 选做：可分析不同聚合方式的影响。

关键代码解释

求和函数

该函数是线程执行的主体，接收一个`SumThread`结构体作为参数，计算数组从`start_idx`到`end_idx`的元素之和，然后使用`__sync_fetch_and_add`原子操作将局部和加到全局变量`sum`上。

```
// 线程函数，用于计算数组部分的和
void* add(void* args) {
    SumThread* t = static_cast<SumThread*>(args);
    int partial_sum = 0;

    for (int i = t->start_idx; i < t->end_idx; ++i) {
        partial_sum += A[i];
    }
    __sync_fetch_and_add(&sum, partial_sum);  // 原子加操作，用于更新全局和
    return NULL;
}
```

线程创建和同步

在main函数中进行了线程的创建与同步，具体如下：

首先进行动态内存分配并计算每个线程的任务分配：

```
pthread_t *thread_handles = new pthread_t[thread_num];
SumThread *startend = new SumThread[thread_num];
int p = N / thread_num;
int q = N % thread_num;
```

线程创建：

```
for (int t = 0; t < thread_num; ++t) {
    startend[t].start_idx = start_idx;
    startend[t].end_idx = start_idx + p + (t < q ? 1 : 0);
    pthread_create(&thread_handles[t], NULL, add, &startend[t]);
    start_idx = startend[t].end_idx;
}
```

循环创建`thread_num`个线程，每个线程的开始和结束索引由`startend`数组中的对应元素指定。如果还有剩余的元素未分配（即`q`大于0），则前`q`个线程将多处理一个元素。`pthread_create`函数创建一个新线程，并将`add`函数作为线程执行的函数，`&startend[t]`作为参数传递给`add`函数。

线程同步：

```
for (int t = 0; t < thread_num; ++t) {
    pthread_join(thread_handles[t], NULL);
}
```

`pthread_join`函数等待指定的线程完成，循环确保所有创建的线程都执行完毕后主程序才继续执行。

编译运行

编译运行：

-lpthread表示编译器链接 POSIX 线程库：

```
g++ lab3_2.cpp -o lab3_2 -lpthread
```

运行结果如下：

```
(base) liuglin@WIN-6G3ESV0SHI5:/mnt/d/Paper/Communication-theory/并行程序设计与算法/实验作业/lab3$ ./lab3_2 1048576 1
Result: 51925467
Using time: 0.000112 s
```

改变线程数和矩阵规模后可得如下表格：

线程数\矩阵规模	1M	2M	4M	16M	32M	64M	128M
1	0.005587	0.01211	0.02208	0.07952	0.1485	0.3156	0.6295
2	0.006517	0.01265	0.02232	0.07467	0.1764	0.3285	0.7192
4	0.005784	0.01384	0.02389	0.06105	0.1732	0.2835	0.7536
8	0.006571	0.01378	0.02335	0.0884	0.1628	0.3241	0.6592
16	0.008053	0.01401	0.02793	0.0657	0.1682	0.3528	0.6512

加速比：

线程数\矩阵规模	1M	2M	4M	16M	32M	64M	128M
1	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2	0.857	0.957	0.989	1.065	0.842	0.961	0.875
4	0.966	0.875	0.924	1.303	0.857	1.113	0.835
8	0.850	0.879	0.946	0.900	0.912	0.974	0.955
16	0.694	0.864	0.791	1.210	0.883	0.895	0.967

根据运行时间，分析程序并行性能及扩展性

- **并行性能和拓展性**
    - 可以看到实验随着进程数的增加加速比并没有提升，反而有可能因为线程增多而变慢。这可能是由于多个线程竞争缓存，在总计算量不大的情况下创建、销毁线程的开销更多导致性能下降。