并行程序设计与算法实验3

实验	**Pthread并行实验**	专业	计算机科学与技术
学号	21311525	姓名	马梓培
Email	mazp@mail2.sysu.edu.cn	完成日期	2024/04/16

1. 实验目的

> 实验1：矩阵乘法
>
> - 使用Pthreads多线程实现并行矩阵乘法
> - 设置线程数量（1-16）及矩阵规模（128-2048）
> - 分析程序并行性能
> - 选做：可分析不同数据及任务划分方式的影响
>
> 实验2：数组求和
>
> - 使用Pthreads创建多线程，实现并行数组求和
> - 设置线程数量（1-16）及数组规模（1M-128M）
> - 分析程序并行性能及扩展性
> - 选做：可分析不同聚合方式的影响

2. 配置安装Pthreads

Linux 下的 GCC/G++:

pthread 库通常是 Linux 系统的默认库，所以通常不需要单独安装。但如果需要，可以使用包管理器安装。

```
# 对于基于 Debian 的系统 (如 Ubuntu)
sudo apt-get install libpthread-stubs0-dev

# 对于基于 Red Hat 的系统 (如 Fedora, CentOS)
sudo yum install pthread-stubs
```

在编译时，需要链接 pthread 库：

```
gcc -o myprogram myprogram.c -lpthread
```

3. Pthreads实现矩阵乘法

代码见附件或github.

首先，先进行一些初始化：

```c
long thread;
pthread_t* thread_handles;

// 初始化互斥锁和条件变量
pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond_var, NULL);

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

initialize_matrix(A, N, 0., 10.);
initialize_matrix(B, N, 0., 10.);

thread_cnt = strtol(argv[1], NULL, 10);
thread_handles = malloc(thread_cnt * sizeof(pthread_t));
avg_rows = N / thread_cnt;
```

我通过将矩阵A按行分块，矩阵B共享来计算矩阵C。

我们知道Pthreads采用的是共享内存的方式来实现。上述的矩阵乘法，并不会有临界区的情况。每个fork的线程都各自处理各自的数据，没有同时访问一块内存的情况。因此，可以直接fork出`thread_cnt`个线程，每个线程执行各自对应的矩阵乘法。

3.1 pthread_create

```c
for (thread = 0; thread < thread_cnt; thread ++) {
    pthread_create(&thread_handles[thread], NULL, matrix_mul, (void *) thread);
}
```

结合`pthread_create()`定义解释：

```c
int pthread_create(
    pthread_t*                 thread_p                 /* out */,
    const pthread_attr_t*      attr_p                   /* in  */,
    void*                      (*start_routine)(void*)  /* in  */,
    void*                      arg_p                    /* in  */);
```

首先，写一个for循环，从0遍历到thread_cnt-1，总共fork出thread_cnt个线程。pthread_create()第一个参数是一个指针，指向我们初始化已经分配好了的pthread_t对象，通过for循环的thread来索引。第二个参数不用，用NULL表示。接下来，*start_routine函数指针，我们传入matrix_mul，表示要并行的函数。而最后一个参数，为每一个线程赋予了唯一的int型参数rank，表示线程的编号。

3.2 pthread_joint

同样，既然有fork线程，我们就需要合并线程。

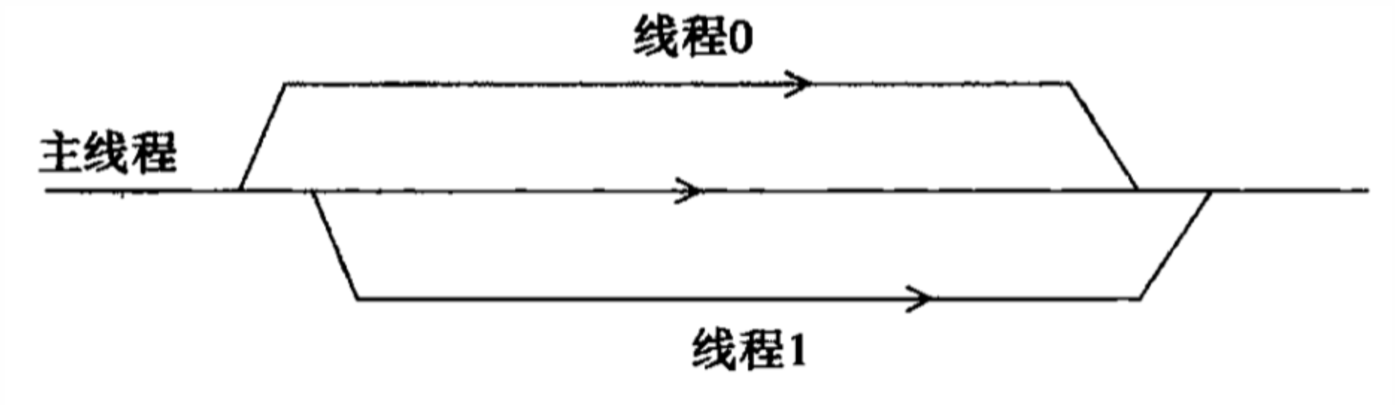

图 4-2　主线程派生与合并两个线程

```
for (thread = 0; thread < thread_cnt; thread ++) {
    pthread_join(thread_handles[thread], NULL);
}
```

结合pthread_join()定义解释：

```
int pthread_join(
    pthread_t       thread        /* in  */,
    void**          ret_val_p     /* out */);
```

首先，写一个for循环，从0遍历到thread_cnt - 1，总共fork出thread_cnt个线程。pthread_create()第一个参数表明，我们要合并哪个线程。第二个参数表示接受的返回值，我们不需要返回值，写上NULL。

3.3 运行计时

首先，我们需要设置一个barrier让所有进程在同一起跑线，copy自书上：

```c
// Barrier
pthread_mutex_lock(&mutex);
counter++;
if (counter == thread_cnt) {
    counter = 0;
    pthread_cond_broadcast(&cond_var);
}
else {
    while (pthread_cond_wait(&cond_var, &mutex) != 0);
}
pthread_mutex_unlock(&mutex);
```

计时，start_time 和 end_time 夹住矩阵相乘：

```c
clock_gettime(CLOCK_MONOTONIC, &start_time);
for (int i = start_rows; i < end_rows; ++i) {
    for (int j = 0; j < N; ++j) {
        float sum = 0;
        for (int x = 0; x < N; ++x) {
            sum += *(A + i * N + x) * *(B + x * N + j);
        }
        *(C + i * N + j) = sum;
    }
}
clock_gettime(CLOCK_MONOTONIC, &end_time);
```

在同步时间时，需要有互斥锁，因为访问的是同一个共享变量：

```c
// 更新全局最大时间
pthread_mutex_lock(&max_time_mutex); // 锁定互斥锁以安全更新
if (elapsed_time > max_elapsed_time) {
    max_elapsed_time = elapsed_time; // 更新最大时间
}
pthread_mutex_unlock(&max_time_mutex); // 解锁互斥锁
```

4. 通用矩阵乘法实验结果

N = 128

```
❯ ./mm.out 1
Max elapsed time among all threads: 0.013769 seconds.
❯ ./mm.out 2
Max elapsed time among all threads: 0.007233 seconds.
❯ ./mm.out 4
Max elapsed time among all threads: 0.003861 seconds.
❯ ./mm.out 8
Max elapsed time among all threads: 0.001570 seconds.
❯ ./mm.out 16
Max elapsed time among all threads: 0.001096 seconds.
```

N = 256

```
❯ ./mm.out 1
Max elapsed time among all threads: 0.084312 seconds.
❯ ./mm.out 2
Max elapsed time among all threads: 0.050017 seconds.
❯ ./mm.out 4
Max elapsed time among all threads: 0.025685 seconds.
❯ ./mm.out 8
Max elapsed time among all threads: 0.016018 seconds.
❯ ./mm.out 16
Max elapsed time among all threads: 0.008387 seconds.
```

N = 512

```
❯ ./mm.out 1
Max elapsed time among all threads: 0.481108 seconds.
❯ ./mm.out 2
Max elapsed time among all threads: 0.260021 seconds.
❯ ./mm.out 4
Max elapsed time among all threads: 0.146547 seconds.
❯ ./mm.out 8
Max elapsed time among all threads: 0.089492 seconds.
❯ ./mm.out 16
Max elapsed time among all threads: 0.092121 seconds.
```

N = 1024

```
❯ ./mm.out 1
Max elapsed time among all threads: 3.753396 seconds.
❯ ./mm.out 2
Max elapsed time among all threads: 1.961918 seconds.
❯ ./mm.out 4
Max elapsed time among all threads: 1.042281 seconds.
❯ ./mm.out 8
Max elapsed time among all threads: 0.903420 seconds.
❯ ./mm.out 16
Max elapsed time among all threads: 0.849642 seconds.
```

N = 2048

```
❯ ./mm.out 1
Max elapsed time among all threads: 32.983432 seconds.
❯ ./mm.out 2
Max elapsed time among all threads: 17.653807 seconds.
❯ ./mm.out 4
Max elapsed time among all threads: 10.603551 seconds.
❯ ./mm.out 8
Max elapsed time among all threads: 10.331040 seconds.
❯ ./mm.out 16
Max elapsed time among all threads: 9.993411 seconds.
```

绘制成表格：

P\N	128	256	512	1024	2048
1	0.013769	0.084312	0.481108	3.753396	32.983432
2	0.007233	0.050017	0.260021	1.961918	17.653807
4	0.003861	0.025685	0.146547	1.042281	10.603551
8	0.001570	0.016018	0.089492	0.903420	10.331040
16	0.001096	0.008387	0.092121	0.849642	9.993411

可视化：

5. 分块矩阵乘法实现

算法原理与上次MPI作业实现分块矩阵乘法基本类似。简短概括，我们可以将A矩阵拆成a个子矩阵，对应于(block_row = N / a)行N列的一个矩阵；将B矩阵拆成b个子矩阵，对应于(block_col = N / b)列N行的一个矩阵，即：

$$AB = \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{a-1} \end{bmatrix} \begin{bmatrix} B_0 & B_1 & \cdots & B_{b-1} \end{bmatrix} = \begin{bmatrix} A_0B_0 & A_0B_1 & \cdots & A_0B_{b-1} \\ A_1B_0 & A_1B_1 & \cdots & A_1B_{b-1} \\ \vdots & & \ddots & A_0B_{b-1} \\ A_{a-1}B_0 & A_{a-1}B_1 & \cdots & A_{a-1}B_{b-1} \end{bmatrix} \tag{1}$$

实现方式：

将可用于计算的进程数（thread_cnt）分解为a*b，然后将矩阵A全体行划分为a个部分，矩阵B全体列划分为b个部分，从而将整个矩阵划分为size相同的（thread_cnt）个块。每个子进程负责计算最终结果的一块。由于通用矩阵乘法的实现中，整个矩阵B都参与了运算，因此考虑到cache的空间局部性原理，分块矩阵乘法应该是要比通用矩阵乘法快上一些的。

5.1 代码实现

首先，根据thread_cnt确认每个线程需要处理的矩阵A和矩阵B大小：

```
if (thread_cnt == 2) {
    block_rows = N / 2;
}
else if (thread_cnt == 4) {
    block_rows = N / 2;
    block_cols = N / 2;
}
else if (thread_cnt == 8) {
    block_rows = N / 4;
    block_cols = N/ 2;
}
else if (thread_cnt == 16) {
    block_rows = N / 4;
    block_cols = N / 4;
}
```

根据分块的大小，得到每个thread需要处理的矩阵范围：

```
long start_row = my_rank * block_rows;
long end_row = (my_rank + 1) * block_rows;

long start_col = my_rank * block_cols;
long end_col = (my_rank + 1) * block_cols;
```

根据已经确定的矩阵范围，执行乘法：

```
for (int i = start_row; i < end_row; ++i) {
    for (int j = start_col; j < end_col; ++j) {
        float sum = 0;
        for (int x = 0; x < N; ++x) {
            sum += *(A + i * N + x) * *(B + x * N + j);
        }
        *(C + i * N + j) = sum;
    }
}
```

6. 分块矩阵乘法实验结果

N = 128

```
❯ ./mm_block.out 1
Max elapsed time among all threads: 0.014830 seconds.
❯ ./mm_block.out 2
Max elapsed time among all threads: 0.008628 seconds.
❯ ./mm_block.out 4
Max elapsed time among all threads: 0.005661 seconds.
❯ ./mm_block.out 8
Max elapsed time among all threads: 0.001791 seconds.
❯ ./mm_block.out 16
Max elapsed time among all threads: 0.001080 seconds.
```

N = 256

```
❯ ./mm_block.out 1
Max elapsed time among all threads: 0.088521 seconds.
❯ ./mm_block.out 2
Max elapsed time among all threads: 0.051135 seconds.
❯ ./mm_block.out 4
Max elapsed time among all threads: 0.031025 seconds.
❯ ./mm_block.out 8
Max elapsed time among all threads: 0.018194 seconds.
❯ ./mm_block.out 16
Max elapsed time among all threads: 0.008349 seconds.
```

N = 512

```
❯ ./mm_block.out 1
Max elapsed time among all threads: 0.486692 seconds.
❯ ./mm_block.out 2
Max elapsed time among all threads: 0.249893 seconds.
❯ ./mm_block.out 4
Max elapsed time among all threads: 0.133712 seconds.
❯ ./mm_block.out 8
Max elapsed time among all threads: 0.093087 seconds.
❯ ./mm_block.out 16
Max elapsed time among all threads: 0.083458 seconds.
```

N = 1024

```
❯ ./mm_block.out 1
Max elapsed time among all threads: 3.700232 seconds.
❯ ./mm_block.out 2
Max elapsed time among all threads: 1.947231 seconds.
❯ ./mm_block.out 4
Max elapsed time among all threads: 1.020938 seconds.
❯ ./mm_block.out 8
Max elapsed time among all threads: 0.864890 seconds.
❯ ./mm_block.out 16
Max elapsed time among all threads: 0.838746 seconds.
```

N = 2048

```
❯ ./mm_block.out 1
Max elapsed time among all threads: 32.724529 seconds.
❯ ./mm_block.out 2
Max elapsed time among all threads: 17.917027 seconds.
❯ ./mm_block.out 4
Max elapsed time among all threads: 10.302597 seconds.
❯ ./mm_block.out 8
Max elapsed time among all threads: 10.231490 seconds.
❯ ./mm_block.out 16
Max elapsed time among all threads: 9.894151 seconds.
```

绘制成表格：

P\N	128	256	512	1024	2048
1	0.014830	0.088521	0.486692	3.700232	32.724529
2	0.008628	0.051135	0.249893	1.947231	17.917027
4	0.005661	0.031025	0.133712	1.020938	10.302597
8	0.001791	0.018194	0.093087	0.864890	10.231490
16	0.001080	0.008349	0.083458	0.838746	9.894151

可视化:

7. Pthread实现数组求和

7.1 pthread_create和pthread_join

```
for (thread = 0; thread < thread_cnt; thread++) {
    threadData[thread].array = array;
    threadData[thread].start = thread * length_per_thread;
    threadData[thread].end = (thread + 1) * length_per_thread;
    pthread_create(&thread_handles[thread], NULL, sum_array,
(void*)&threadData[thread]);
}

for (thread = 0; thread < thread_cnt; thread++) {
    pthread_join(thread_handles[thread], NULL);
}
```

与矩阵乘法极为相似，不过多赘述。

7.2 array sum

```c
void* sum_array(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    long long sum = 0;
    for (long long i = data->start; i < data->end; i++) {
        sum += data->array[i];
    }

    pthread_mutex_lock(&mutex);
    global_sum += sum;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

首先，先将自己进程中的A[i]相加求和，得到局部和local_sum。然后降其加到共享变量中。然而，由于共享的缘故，我们必须互斥地访问改变量。因此，需要有一个互斥锁来保证互斥访问。

8. 实验结果

N = 1M

```
❯ ./as.out 1
Total sum: 49483647
Cost time: 0.001771 seconds
❯ ./as.out 2
Total sum: 49483647
Cost time: 0.000794 seconds
❯ ./as.out 4
Total sum: 49483647
Cost time: 0.000377 seconds
❯ ./as.out 8
Total sum: 49483647
Cost time: 0.000599 seconds
❯ ./as.out 16
Total sum: 49483647
Cost time: 0.000664 seconds
```

N = 2M

```
❯ ./as.out 1
Total sum: 98971657
Cost time: 0.003424 seconds
❯ ./as.out 2
Total sum: 98971657
Cost time: 0.001792 seconds
❯ ./as.out 4
Total sum: 98971657
Cost time: 0.000795 seconds
❯ ./as.out 8
Total sum: 98971657
Cost time: 0.000985 seconds
```

```
❯ ./as.out 16
Total sum: 98971657
Cost time: 0.000816 seconds
```

N = 4M

```
❯ ./as.out 1
Total sum: 198025019
Cost time: 0.005634 seconds
❯ ./as.out 2
Total sum: 198025019
Cost time: 0.002740 seconds
❯ ./as.out 4
Total sum: 198025019
Cost time: 0.001475 seconds
❯ ./as.out 8
Total sum: 198025019
Cost time: 0.001373 seconds
❯ ./as.out 16
Total sum: 198025019
Cost time: 0.001206 seconds
```

N = 8M

```
❯ ./as.out 1
Total sum: 396047077
Cost time: 0.008436 seconds
❯ ./as.out 2
Total sum: 396047077
Cost time: 0.004492 seconds
❯ ./as.out 4
Total sum: 396047077
Cost time: 0.002281 seconds
❯ ./as.out 8
Total sum: 396047077
Cost time: 0.001943 seconds
❯ ./as.out 16
Total sum: 396047077
Cost time: 0.001930 seconds
```

N = 16M

```
❯ ./as.out 1
Total sum: 792011723
Cost time: 0.017006 seconds
❯ ./as.out 2
Total sum: 792011723
Cost time: 0.008234 seconds
❯ ./as.out 4
Total sum: 792011723
Cost time: 0.004939 seconds
❯ ./as.out 8
Total sum: 792011723
```

```
Cost time: 0.004415 seconds
❯ ./as.out 16
Total sum: 792011723
Cost time: 0.003793 seconds
```

N = 32M

```
❯ ./as.out 1
Total sum: 1584318480
Cost time: 0.035475 seconds
❯ ./as.out 2
Total sum: 1584318480
Cost time: 0.017141 seconds
❯ ./as.out 4
Total sum: 1584318480
Cost time: 0.009986 seconds
❯ ./as.out 8
Total sum: 1584318480
Cost time: 0.007589 seconds
❯ ./as.out 16
Total sum: 1584318480
Cost time: 0.007616 seconds
```

N = 64M

```
❯ ./as.out 1
Total sum: 3168123411
Cost time: 0.074379 seconds
❯ ./as.out 2
Total sum: 3168123411
Cost time: 0.033816 seconds
❯ ./as.out 4
Total sum: 3168123411
Cost time: 0.018382 seconds
❯ ./as.out 8
Total sum: 3168123411
Cost time: 0.016177 seconds
❯ ./as.out 16
Total sum: 3168123411
Cost time: 0.015637 seconds
```

N = 128M

```
❯ ./as.out 1
Total sum: 6336129725
Cost time: 0.145537 seconds
❯ ./as.out 2
Total sum: 6336129725
Cost time: 0.075687 seconds
❯ ./as.out 4
Total sum: 6336129725
Cost time: 0.036094 seconds
❯ ./as.out 8
```

```
Total sum: 6336129725
Cost time: 0.034524 seconds
❯ ./as.out 16
Total sum: 6336129725
Cost time: 0.030697 seconds
```

P\N	1M	2M	4M	8M	16M	32M	64M	128M
1	0.001771	0.003424	0.005634	0.008436	0.017006	0.035475	0.074379	0.145537
2	0.000794	0.001792	0.002740	0.004492	0.008234	0.017141	0.033816	0.075687
4	0.000377	0.000795	0.001475	0.002281	0.004939	0.009986	0.018382	0.036094
8	0.000599	0.000985	0.001373	0.001943	0.004415	0.007589	0.016177	0.034524
16	0.000664	0.000816	0.001206	0.001930	0.003793	0.007616	0.015637	0.030697

可视化：

观察可视化结果，我们可以发现，当线程数增大时（比如4->8时），耗时反而会增大。合理猜测是因为多个线程都要抢互斥锁而导致的进程上下文切换，因此我们可以把结果先存储起来，然后直接相加。考虑到最多仅有16个线程，因此，我们至多只需要进行15次加法。聚合这一操作，并无并行的必要。所以，我直接一个for循环解决问题。

9. 无互斥锁实验结果

实现代码：

```c
for (thread = 0; thread < thread_cnt; thread++) {
    pthread_join(thread_handles[thread], NULL);
    total_sum += threadData[thread].result;
}
```

N = 1M

```
❯ ./as.out 1
Total sum: 49483647
Cost time: 0.002609 seconds
❯ ./as.out 2
Total sum: 49483647
Cost time: 0.001175 seconds
❯ ./as.out 4
Total sum: 49483647
Cost time: 0.000794 seconds
❯ ./as.out 8
Total sum: 49483647
Cost time: 0.000637 seconds
❯ ./as.out 16
Total sum: 49483647
Cost time: 0.000507 seconds
```

N = 2M

```
❯ ./as.out 1
Total sum: 98971657
Cost time: 0.003239 seconds
❯ ./as.out 2
Total sum: 98971657
Cost time: 0.001655 seconds
❯ ./as.out 4
Total sum: 98971657
Cost time: 0.001380 seconds
❯ ./as.out 8
Total sum: 98971657
Cost time: 0.000956 seconds
❯ ./as.out 16
Total sum: 98971657
Cost time: 0.000874 seconds
```

N = 4M

```
❯ ./as.out 1
Total sum: 198025019
Cost time: 0.005645 seconds
❯ ./as.out 2
Total sum: 198025019
Cost time: 0.002686 seconds
❯ ./as.out 4
Total sum: 198025019
Cost time: 0.001656 seconds
❯ ./as.out 8
Total sum: 198025019
Cost time: 0.001593 seconds
❯ ./as.out 16
Total sum: 198025019
Cost time: 0.001135 seconds
```

N = 8M

```
❯ ./as.out 1
Total sum: 396047077
Cost time: 0.008607 seconds
❯ ./as.out 2
Total sum: 396047077
Cost time: 0.004442 seconds
❯ ./as.out 4
Total sum: 396047077
Cost time: 0.002432 seconds
❯ ./as.out 8
Total sum: 396047077
Cost time: 0.002071 seconds
❯ ./as.out 16
Total sum: 396047077
Cost time: 0.001857 seconds
```

N = 16M

```
❯ ./as.out 1
Total sum: 792011723
Cost time: 0.017711 seconds
❯ ./as.out 2
Total sum: 792011723
Cost time: 0.008411 seconds
❯ ./as.out 4
Total sum: 792011723
Cost time: 0.005225 seconds
❯ ./as.out 8
Total sum: 792011723
Cost time: 0.004062 seconds
❯ ./as.out 16
Total sum: 792011723
Cost time: 0.004248 seconds
```

N = 32M

```
❯ ./as.out 1
Total sum: 1584318480
Cost time: 0.034818 seconds
❯ ./as.out 2
Total sum: 1584318480
Cost time: 0.017545 seconds
❯ ./as.out 4
Total sum: 1584318480
Cost time: 0.009814 seconds
❯ ./as.out 8
Total sum: 1584318480
Cost time: 0.008950 seconds
❯ ./as.out 16
Total sum: 1584318480
Cost time: 0.007547 seconds
```

N = 64M

```
❯ ./as.out 1
Total sum: 3168123411
Cost time: 0.070977 seconds
❯ ./as.out 2
Total sum: 3168123411
Cost time: 0.038256 seconds
❯ ./as.out 4
Total sum: 3168123411
Cost time: 0.020395 seconds
❯ ./as.out 8
Total sum: 3168123411
Cost time: 0.016129 seconds
❯ ./as.out 16
Total sum: 3168123411
Cost time: 0.015611 seconds
```

N = 128M

```
❯ ./as.out 1
Total sum: 6336129725
Cost time: 0.143842 seconds
❯ ./as.out 2
Total sum: 6336129725
Cost time: 0.067061 seconds
❯ ./as.out 4
Total sum: 6336129725
Cost time: 0.039102 seconds
❯ ./as.out 8
Total sum: 6336129725
Cost time: 0.030899 seconds
❯ ./as.out 16
Total sum: 6336129725
Cost time: 0.029182 seconds
```

P\N	1M	2M	4M	8M	16M	32M	64M	128M
1	0.002609	0.003239	0.005645	0.008607	0.017711	0.034818	0.070977	0.143842
2	0.001175	0.001655	0.002686	0.004442	0.008411	0.017545	0.038256	0.067061
4	0.000794	0.001380	0.001656	0.002432	0.005225	0.009814	0.020395	0.039102
8	0.000637	0.000956	0.001593	0.002071	0.004062	0.008950	0.016129	0.030899
16	0.000507	0.000874	0.001135	0.001857	0.004248	0.007547	0.015611	0.029182

可视化：

可以看到，先将进程求和的结果存起来，然后再相加，我们得到的曲线是十分正常的。随着进程数的增大，耗时也随之减少（8->16没什么变化，是因为我的电脑只有8核）。

10. 实验感想

在这次的并行程序设计与算法实验中，我深入探索了Pthreads的使用，特别是在矩阵乘法和数组求和的应用中。

在矩阵乘法部分，通过分块计算的方式，我学习到了如何有效地利用内存和处理器资源，这种方法不仅提高了计算速度，也优化了数据的存取效率。此外，我对不同线程分配策略下的性能差异有了更深入的了解，特别是在处理大矩阵时，合理的线程分配和任务划分显得尤为重要。

数组求和实验则让我体会到了在并行计算中同步和互斥的重要性。特别是在大规模数据操作中，正确地管理线程间的数据访问和资源竞争是确保程序正确运行和高效性能的关键。通过比较有互斥锁和无互斥锁的实验结果，我明白了互斥锁对于保护共享数据的重要性，同时也见证了过多的锁请求可能导致的性能瓶颈。