中山大学计算机学院本科生实验报告

（2024 学年秋季学期）

课程名称：高性能计算程序设计基础　　批改人：

实验	共享内存编程	**专业（方向）**	信息与计算科学
学号	22336049	**姓名**	陈日康
Email	chenih5@mail2.sysu.edu.cn	**完成日期**	2024 年 11 月 12 日

1. 实验目的

1. 通过 OpenMP 实现通用矩阵乘法
2. 基于 OpenMP 的通用矩阵乘法优化
3. 构造基于 Pthreads 的并行 for 循环分解、分配和执行机制。

2. 实验过程和核心代码

(1). 通过 OpenMP 实现通用矩阵乘法

通过 OpenMP 实现通用矩阵乘法（Lab1）的并行版本，OpenMP 并行线程从 1 增加至 8，矩阵规模从 512 增加至 2048。

(i). matrix_multiplication

```
void matrix_multiplication(const std::vector<std::vector<double>>& A,
                           const std::vector<std::vector<double>>& B,
                           std::vector<std::vector<double>>& C,
                           int size, int num_threads) {
    #pragma omp parallel for num_threads(num_threads) collapse(2)
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            double sum = 0.0;
            for (int k = 0; k < size; ++k) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}
```

执行矩阵乘法，将矩阵 A 和矩阵 B 相乘，结果存储在矩阵 C 中。size 表示矩阵的维度，num_threads 表示要使用的线程数。使用 OpenMP 的并行指令#pragma omp parallel for 以 num_threads 指定的线程数进行并行计算。collapse(2)表示将嵌套的两个循环合并成一个更大的循环，增加并行度。通过嵌套循环计算矩阵 C 的每个元素 C[i][j]，其值为矩阵 A 的第 i 行与矩阵 B 的第 j 列的点积。

(2). 基于 OpenMP 的通用矩阵乘法优化

分别采用 OpenMP 的默认任务调度机制、静态调度 schedule(static, 1)和动态调度 schedule(dynamic,1)的性能，实现#pragma omp for，并比较其性能。

(i). matrix_multiplication

```
void matrix_multiplication(const std::vector<std::vector<double>>& A,
                           const std::vector<std::vector<double>>& B,
                           std::vector<std::vector<double>>& C,
                           int M, int N, int K, int num_threads, int schedule_type) {
    switch (schedule_type) {
        case 1:
            omp_set_schedule(omp_sched_auto, 1);
            break;
        case 2:
            omp_set_schedule(omp_sched_static, 1);
            break;
        case 3:
            omp_set_schedule(omp_sched_dynamic, 1);
            break;
        default:
            std::cerr << "Invalid schedule type. Using default scheduling.\n";
            omp_set_schedule(omp_sched_auto, 1);
            break;
    }

    #pragma omp parallel for num_threads(num_threads) collapse(2)
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < K; ++j) {
            double sum = 0.0;
            for (int k = 0; k < N; ++k) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}
```

omp_set_schedule 根据 schedule_type 参数动态设置 OpenMP 的任务调度策略：omp_sched_auto (case 1)：由编译器和运行时系统决定。omp_sched_static (case 2)：均匀划分任务给线程，适合负载均匀的情况。omp_sched_dynamic (case 3)：线程完成任务后继续领取剩余任务，适合负载不均匀的情况。#pragma omp parallel for：启动 OpenMP 的并行区域，使用 num_threads 指定线程数。collapse(2)将外层两个循环（i 和 j）展平，线程可以同时处理多行多列的计算，提高负载均衡性。

(3). 构造基于 Pthreads 的并行 for 循环分解、分配和执行机制。

1). 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。

2). 在 Linux 系统中将 parallel_for 函数编译为.so 文件，由其他程序调用。

3). 将通用矩阵乘法的 for 循环，改造成基于 parallel_for 函数并行化的矩阵乘法，注意只改造可被并行执行的 for 循环（例如无 race condition、无数据依赖、无循环依赖等）。

(i). matrix_multiply_functor

```
void* matrix_multiply_functor(void* args) {
    matrix_args* mat_args = (matrix_args*)args;
    for (int i = mat_args->start_row; i < mat_args->end_row; ++i) {
        for (int j = 0; j < mat_args->size; ++j) {
            int sum = 0;  // 使用整数类型
            for (int k = 0; k < mat_args->size; ++k) {
                sum += (*mat_args->A)[i][k] * (*mat_args->B)[k][j];
            }
            (*mat_args->C)[i][j] = sum;
        }
    }
    return nullptr;
}
```

实现矩阵乘法逻辑，由线程并行调用，完成指定行范围内的矩阵运算。每个线程根据 start_row 和 end_row 参数只处理特定范围内的行。这种分块计算方式将矩阵的运算任务分解为多个子任务，便于多线程执行。具体操作中，外层 for 循环确定线程负责的行，确保每个线程只计算自己的分块。中间循环遍历列，内层循环完成矩阵元素的乘法累加，将结果存储到矩阵 C 中。

(ii). 多线程任务分配与执行

```
int rows_per_thread = size / num_threads;
std::vector<matrix_args> thread_args(num_threads);

for (int i = 0; i < num_threads; ++i) {
    int start_row = i * rows_per_thread;
    int end_row = (i == num_threads - 1) ? size : start_row + rows_per_thread; // 确保最后线程覆盖剩余行

    thread_args[i] = {&A, &B, &C, size, start_row, end_row};
}

// 使用 parallel_for 调用
for (int i = 0; i < num_threads; ++i) {
    parallel_for(i, i + 1, 1, matrix_multiply_functor, &thread_args[i], num_threads);
}
```

rows_per_thread 确定每个线程处理的行数。start_row 和 end_row 明确每个线程的工作范围，最后一个线程负责剩余行。parallel_for 调用 matrix_multiply_functor，完成每个线程的矩阵乘法任务。通过 thread_args[i]将矩阵及任务范围传递给对应线程。

(iii). parallel_for.cpp

```
void* thread_function(void* args) {
    struct for_index* index = (struct for_index*)args;
    return index->functor(index->arg); // 调用 functor
}
```

thread_function 是每个线程的入口函数，它解读传递的参数 args，args 是 for_index 结构体的指针，并调用指定的函数指针 functor，对传入的参数 index->arg 执行操作。

```
void parallel_for(int start, int end, int increment, void* (*functor)(void*), void* arg, int num_threads) {
    // 创建线程
    pthread_t threads[num_threads];
    struct for_index thread_args[num_threads];

    // 计算任务分配
    int chunk_size = (end - start) / num_threads;
    int remainder = (end - start) % num_threads; // 处理不能整除的情况

    for (int i = 0; i < num_threads; ++i) {
        thread_args[i].start = start + i * chunk_size + (i < remainder ? i : remainder);
        thread_args[i].end = thread_args[i].start + chunk_size + (i < remainder ? 1 : 0);
        thread_args[i].increment = increment;
        thread_args[i].functor = functor;
        thread_args[i].arg = arg;

        pthread_create(&threads[i], nullptr, thread_function, &thread_args[i]);
    }

    // 等待所有线程完成
    for (int i = 0; i < num_threads; ++i) {
        pthread_join(threads[i], nullptr);
    }
}
```

parallel_for 实现，将循环分配到多个线程中并行执行。使用 chunk_size 将总任务平分给线程。通过 remainder 处理无法整除的情况，前 remainder 个线程多处理一个任务。每个线程分配一个任务范围 [start, end)。通过 for_index 结构体传递任务范围、增量、用户定义函数 functor 和其参数 arg。调用 pthread_create 创建 num_threads 个线程，每个线程运行 thread_function，执行分配的任务。使用 pthread_join 确保线程全部执行完成，避免主线程过早退出。

(iii). parallel_for.h

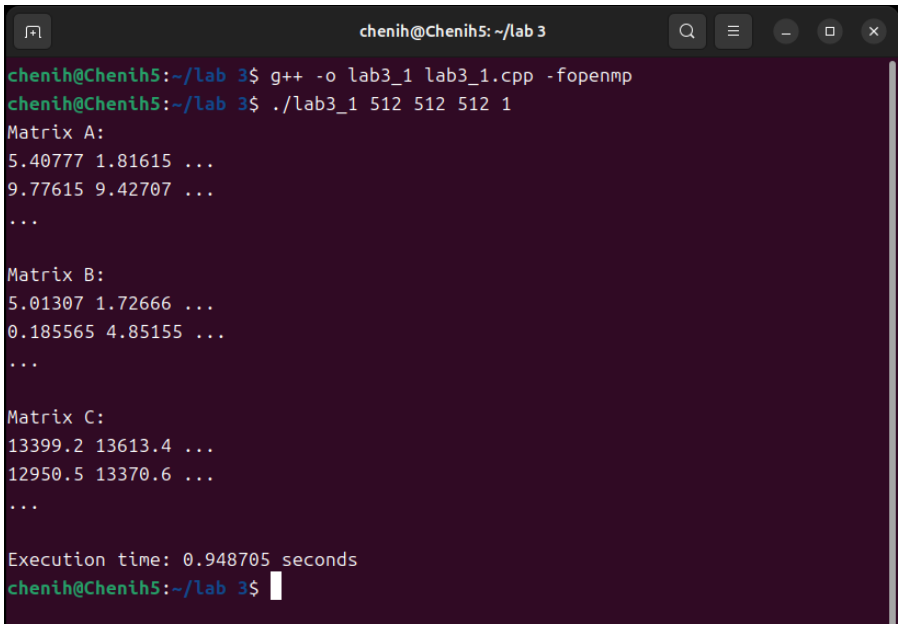
```
struct for_index {
    int start;                  // 起始索引
    int end;                    // 结束索引
    int increment;              // 增量
    void* (*functor)(void*);    // 函数指针，线程要执行的任务
    void* arg;                  // functor 的参数
};

// parallel_for 函数声明
void parallel_for(int start, int end, int increment, void* (*functor)(void*), void* arg, int num_threads);
```

struct for_index 定义线程任务的分工结构体，包括起始索引、结束索引、增量、任务函数指针(functor)，以及函数参数(arg)。parallel_for 声明一个通用多线程执行函数。

3. 实验结果

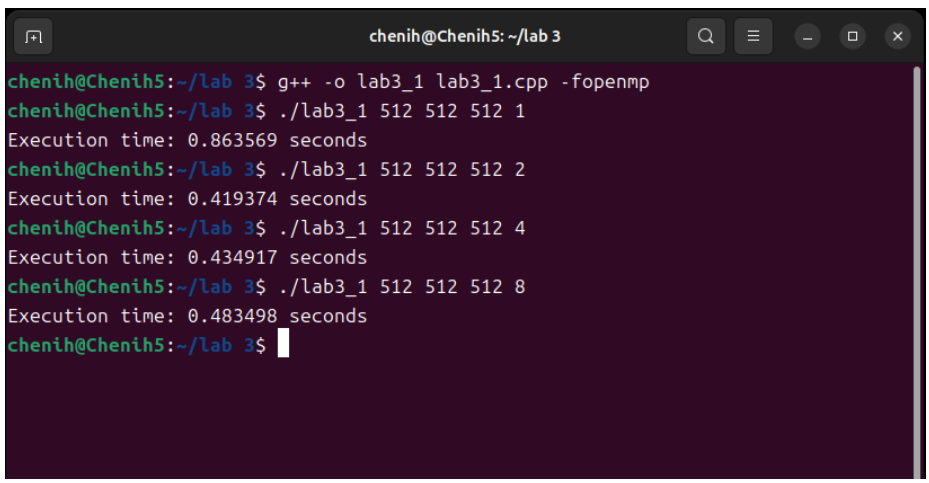
(1). 通过 OpenMP 实现通用矩阵乘法

打印运行结果：

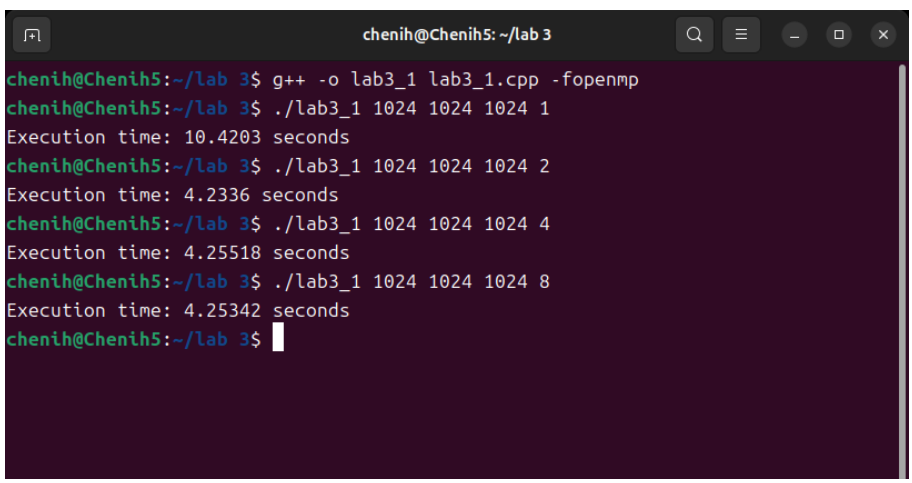


为简化结果只打印首 2*2 矩阵。由 print_matrix_flag 控制是否打印矩阵结果。

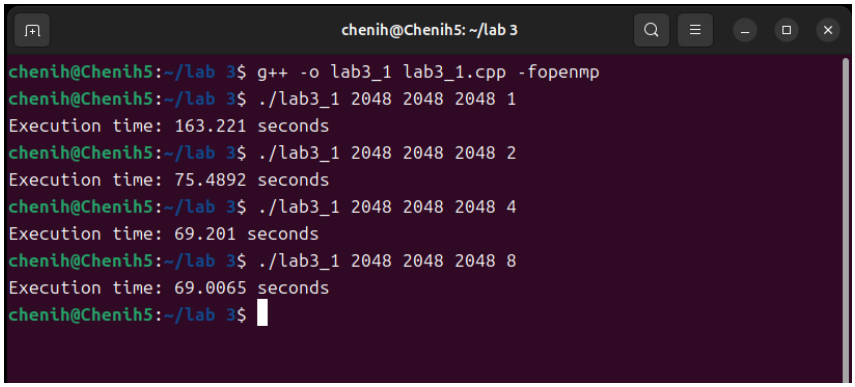
512 阶：



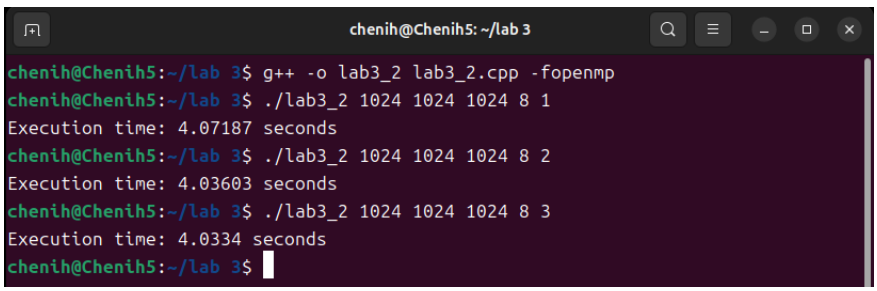
1024 阶：



2048 阶：



(2). 基于 OpenMP 的通用矩阵乘法优化



命令格式：`./lab3_2 <rows> <cols> <inner><threads> <schedule>`，其中 schedule 为调度策略：1=默认,，2=静态，3=动态。

以 1024 阶矩阵、线程数 8 作为测试。可以看出，动态调度的效果最佳，静态调度的表现优于默认调度。

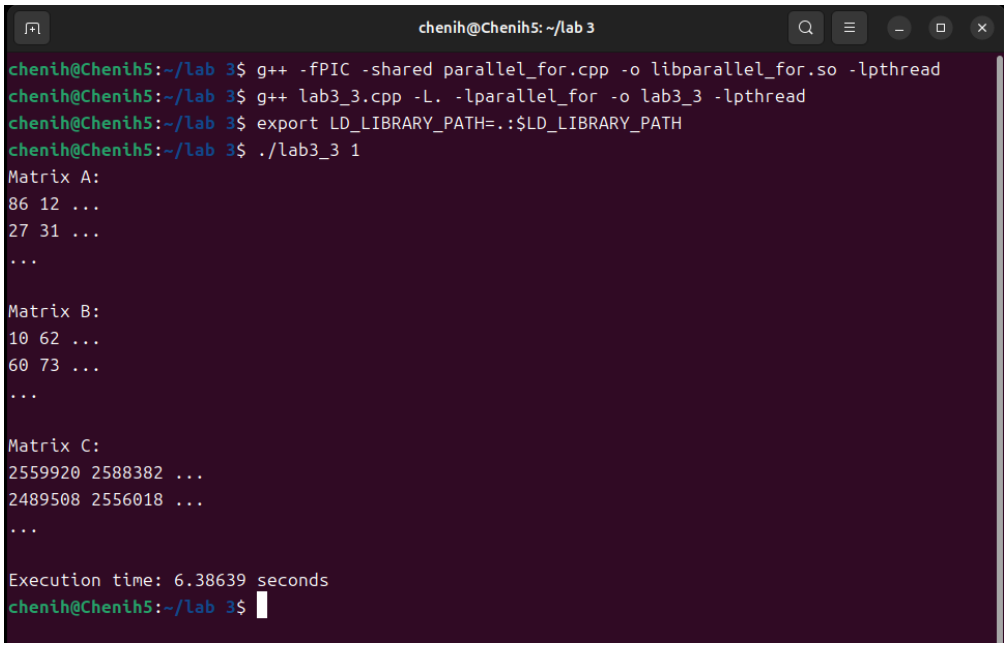
(3). 构造基于 Pthreads 的并行 for 循环分解、分配和执行机制。

编译 parallel_for.cpp 为共享库：`g++ -fPIC -shared parallel_for.cpp -o libparallel_for.so -lpthread`

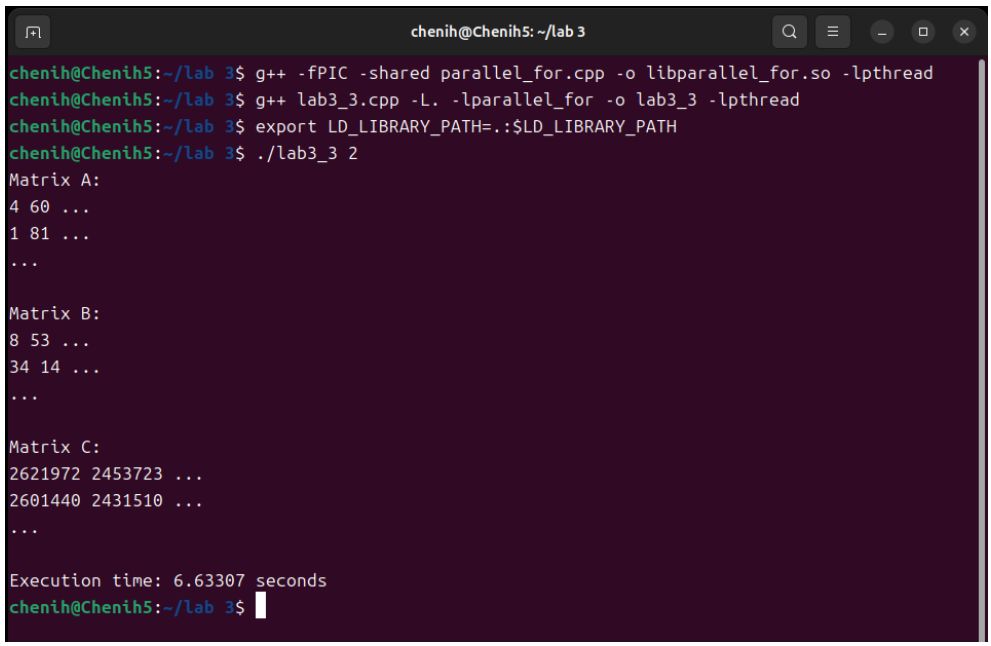
编译并链接 lab3_3.cpp 与共享库：`g++ lab3_3.cpp -L. -lparallel_for -o lab3_3 -lpthread`

设置共享库路径并运行程序：`export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH`

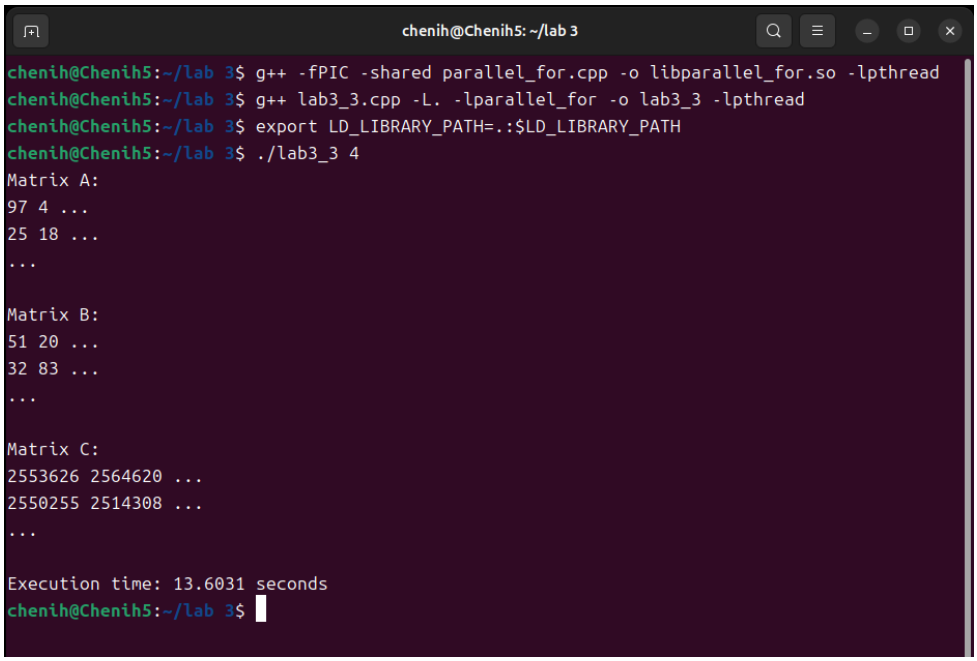
单线程：


chenih@Chenih5: ~/lab 3
chenih@Chenih5:~/lab 3$ g++ -fPIC -shared parallel_for.cpp -o libparallel_for.so -lpthread
chenih@Chenih5:~/lab 3$ g++ lab3_3.cpp -L. -lparallel_for -o lab3_3 -lpthread
chenih@Chenih5:~/lab 3$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
chenih@Chenih5:~/lab 3$ ./lab3_3 1
Matrix A:
86 12 ...
27 31 ...
...
Matrix B:
10 62 ...
60 73 ...
...
Matrix C:
2559920 2588382 ...
2489508 2556018 ...
...
Execution time: 6.38639 seconds
chenih@Chenih5:~/lab 3$

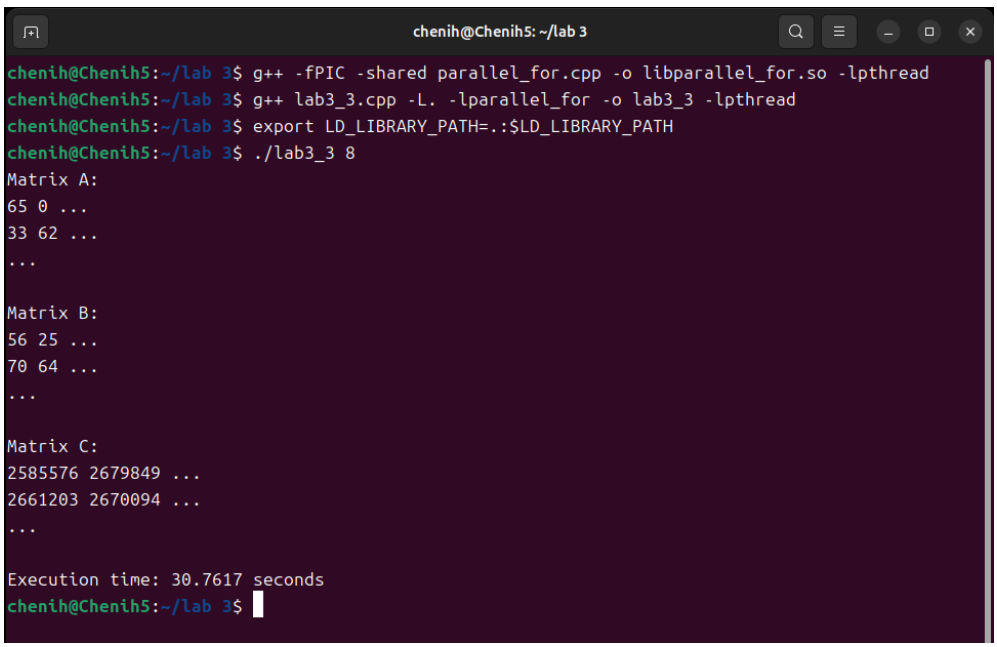
双线程：


chenih@Chenih5: ~/lab 3
chenih@Chenih5:~/lab 3$ g++ -fPIC -shared parallel_for.cpp -o libparallel_for.so -lpthread
chenih@Chenih5:~/lab 3$ g++ lab3_3.cpp -L. -lparallel_for -o lab3_3 -lpthread
chenih@Chenih5:~/lab 3$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
chenih@Chenih5:~/lab 3$ ./lab3_3 2
Matrix A:
4 60 ...
1 81 ...
...
Matrix B:
8 53 ...
34 14 ...
...
Matrix C:
2621972 2453723 ...
2601440 2431510 ...
...
Execution time: 6.63307 seconds
chenih@Chenih5:~/lab 3$

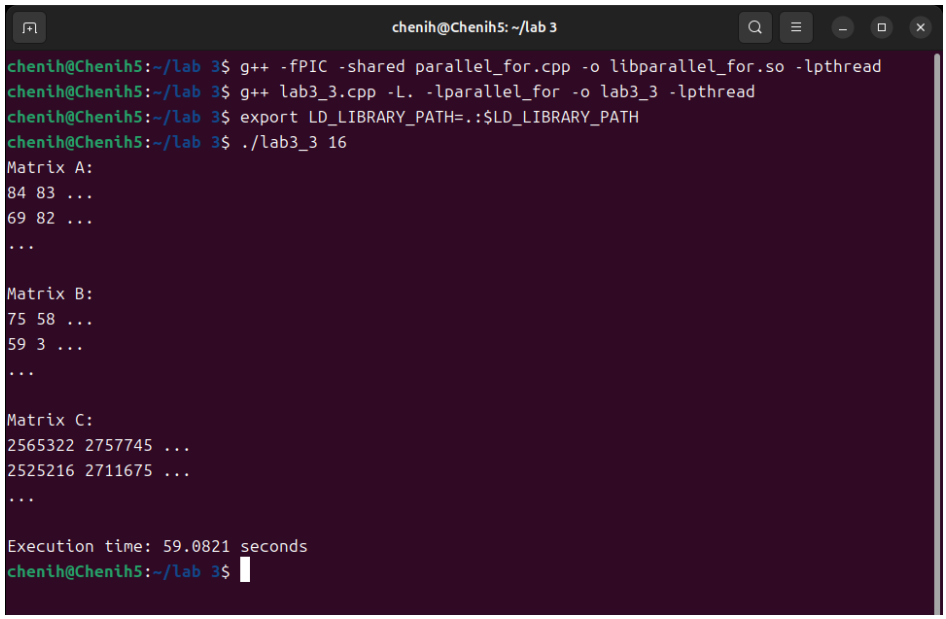
四线程：


chenih@Chenih5: ~/lab 3
chenih@Chenih5:~/lab 3$ g++ -fPIC -shared parallel_for.cpp -o libparallel_for.so -lpthread
chenih@Chenih5:~/lab 3$ g++ lab3_3.cpp -L. -lparallel_for -o lab3_3 -lpthread
chenih@Chenih5:~/lab 3$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
chenih@Chenih5:~/lab 3$ ./lab3_3 4
Matrix A:
97 4 ...
25 18 ...
...
Matrix B:
51 20 ...
32 83 ...
...
Matrix C:
2553626 2564620 ...
2550255 2514308 ...
...
Execution time: 13.6031 seconds
chenih@Chenih5:~/lab 3$

八线程：


chenih@Chenih5: ~/lab 3
chenih@Chenih5:~/lab 3$ g++ -fPIC -shared parallel_for.cpp -o libparallel_for.so -lpthread
chenih@Chenih5:~/lab 3$ g++ lab3_3.cpp -L. -lparallel_for -o lab3_3 -lpthread
chenih@Chenih5:~/lab 3$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
chenih@Chenih5:~/lab 3$ ./lab3_3 8
Matrix A:
65 0 ...
33 62 ...
...
Matrix B:
56 25 ...
70 64 ...
...
Matrix C:
2585576 2679849 ...
2661203 2670094 ...
...
Execution time: 30.7617 seconds
chenih@Chenih5:~/lab 3$

十六线程：



4. 实验感想

本次实验让我深入理解了通用矩阵乘法的并行计算原理。通过实现 OpenMP 的并行版本，我认识到在处理大规模数据时，采用并行计算能显著提升性能，尤其是在多核处理器环境中。在性能比较中，我观察到不同调度策略的影响，静态调度在负载均衡时效果较好，而动态调度则在应对不均匀负载时表现更佳。这让我意识到选择合适的调度策略是优化并行性能的关键。此外，构建基于 Pthreads 的 parallel_for 函数让我增强了对线程管理和同步的理解。在实现过程中，我确保了数据依赖和竞态条件的处理，提升了并行执行的安全性。总体而言，这次实验不仅提升了我的并行编程能力，也为我未来的学习和工作打下了良好的基础。