中山大学计算机院本科生实验报告

（2024 学年春季学期）

课程名称：并行程序设计	批改人:
实验：3	专业（方向）：计算机科学与技术
学号：22336226	姓名：王泓沣
Email：wanghf59@mail2.sysu.edu.cn	完成日期：2025/4/13

1 实验目的

1.1 并行矩阵乘法

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： 矩阵计算所消耗的时间。使用其他矩阵计算库来验算你的计算是否准确。

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。可以和其他高性能矩阵计算库比较性能

1.2 并行数组求和

使用 Pthreads 实现并行数组求和，并通过实验分析其性能。

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 $s = \sum_{i=1}^{n} A_i$

输出： 求和计算所消耗的时间 t。使用其他方法验证答案。

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。

2 实验过程和核心代码

2.1 并行矩阵乘法

线程结构体：

```
typedef struct {
    int thread_id;
    int start_row;
    int end_row;
} thread_data_t;
```

按行划分任务给每个线程，每个线程负责计算 C 的 [start_row, end_row] 区间：

```
void *thread_work(void *arg) {
    thread_data_t *data = (thread_data_t *)arg;
    for (int i = data->start_row; i < data->end_row; i++) {
        for (int j = 0; j < K; j++) {
            double sum = 0.0;
            for (int k = 0; k < N; k++) {
                sum += A[i * N + k] * B[k * K + j];
            }
            C[i * K + j] = sum;
        }
    }
    pthread_exit(NULL);
}
```

验证计算结果采用 macOS 原生的 Accelerate 框架验证

```
void verify_with_blas(const double *A, const double *B, const double *C,
   int M, int N, int K) {
    // 分配 C_blas
    double *C_blas = (double *)malloc(sizeof(double) * M * K);
    if (!C_blas) {
        fprintf(stderr, "Error: Failed to allocate C_blas\n");
        return;
    }
    // 初始化 C_blas
    for (int i = 0; i < M * K; i++) {
        C_blas[i] = 0.0;
    }

    // 使用新版 cblas_dgemm 进行矩阵乘法计算
    //  A: M x N, B: N x K, 结果 C_blas: M x K
```

```
    cblas_dgemm(
        CblasRowMajor,      // 数据按行存储
        CblasNoTrans,       // A 不转置
        CblasNoTrans,       // B 不转置
        (long long)M,       // A 的行数
        (long long)K,       // B 的列数（结果 C 的列数）
        (long long)N,       // 公共维度
        1.0,                // alpha
        A,                  // 矩阵 A
        (long long)N,       // lda = A 的列数
        B,                  // 矩阵 B
        (long long)K,       // ldb = B 的列数
        0.0,                // beta
        C_blas,             // 输出矩阵 C_blas
        (long long)K        // ldc = C_blas 的列数
    );

    // 计算并打印最大差异
    double max_diff = compare_results(C, C_blas, M, K);
    printf("  >> Max difference (Pthreads vs BLAS) = %e\n", max_diff);

    free(C_blas);
}
```

主函数计时部分

```
        double start_time = get_time();
        // 按行划分任务给每个线程
        int rows_per_thread = M / num_threads;
        int remainder = M % num_threads;
        int current_row = 0;
        for (int i = 0; i < num_threads; i++) {
            thread_data[i].thread_id = i;
            thread_data[i].start_row = current_row;
            int assigned_rows = rows_per_thread + ((i < remainder) ? 1 :
                0);
            thread_data[i].end_row = current_row + assigned_rows;
            current_row += assigned_rows;
            pthread_create(&threads[i], NULL, thread_work, &thread_data[i
                ]);
        }
```

```
        // 等待所有线程结束
        for (int i = 0; i < num_threads; i++) {
            pthread_join(threads[i], NULL);
        }
        // 结束计时
        double end_time = get_time();
        double elapsed = end_time - start_time;
```

2.2 并行数组求和

线程结构体:

```
typedef struct {
    int start;          // 当前线程处理起始下标
    int end;            // 当前线程处理结束下标
    long long partial_sum;  // 当前线程计算的局部和
} thread_arg_t;
```

每个线程使用无锁方式计算局部求和，最终由主线程汇聚:

```
void* parallel_sum_without_mutex(void* arg) {
    thread_arg_t *targ = (thread_arg_t*) arg;
    long long sum = 0;
    for (int i = targ->start; i < targ->end; i++) {
        sum += array[i];
    }
    targ->partial_sum = sum;
    return NULL;
}
```

主函数部分:

```
            int num_threads = thread_nums[k];
            pthread_t threads[num_threads];
            thread_arg_t thread_args[num_threads];
            int chunk = array_size / num_threads;

            struct timeval start, end;
            gettimeofday(&start, NULL);

            for (int t = 0; t < num_threads; t++) {
                thread_args[t].start = t * chunk;
                thread_args[t].end = (t == num_threads - 1) ? array_size
                    : (t + 1) * chunk;
```

```
        thread_args[t].partial_sum = 0;
        pthread_create(&threads[t], NULL,
            parallel_sum_without_mutex, &thread_args[t]);
    }

    long long sum_result = 0;
    for (int t = 0; t < num_threads; t++) {
        pthread_join(threads[t], NULL);
        sum_result += thread_args[t].partial_sum;
    }

    gettimeofday(&end, NULL);
    double time_spent = (end.tv_sec - start.tv_sec) * 1e6 + (end.
        tv_usec - start.tv_usec);
    time_spent /= 1000;  // Convert microseconds to milliseconds
```

3 实验结果

3.1 并行矩阵乘法

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.005995	0.021824	0.122613	1.023389	17.261944
2	0.001664	0.009010	0.063155	0.543305	10.909794
4	0.000940	0.004640	0.038930	0.313809	11.186346
8	0.000647	0.002713	0.025383	0.284973	7.941045
16	0.000554	0.002832	0.026191	0.272255	7.876508

表 1: 并行矩阵乘法实验结果 (s)

3.2 并行数组求和

进程数	数组规模（M）							
	1	2	4	8	16	32	64	128
1	0.815	0.021824	0.122613	1.023389	17.261944	34.523888	69.047776	138.095552
2	0.001664	0.009010	0.063155	0.543305	10.909794	21.819588	43.639176	87.278352
4	0.000940	0.004640	0.038930	0.313809	11.186346	22.372692	44.745384	89.490768
8	0.000647	0.002713	0.025383	0.284973	7.941045	15.882090	31.764180	63.528360
16	0.000554	0.002832	0.026191	0.272255	7.876508	15.753016	31.506032	63.012064

表 2: 并行数组求和实验结果 (ms)

4 实验感想