中山大学计算机院本科生实验报告

（2025学年春季学期）

课程名称：并行程序设计
批改人：

实验	3-Pthreads并行矩阵乘法与数组求和	专业（方向）	计算机科学与技术
学号	22336087	姓名	胡瑞康
Email	hurk3@mail2.sysu.edu.cn	完成日期	2025.4.2

1 并行矩阵乘法

1.1 代码介绍

本实验实现了两种使用 Pthreads 实现的并行矩阵乘法方案，分别为“行划分”（`row_pthread.cpp`）和“块划分”（`block_pthread.cpp`）。两种方法都遵循如下基本流程：

1. 读取输入参数（线程数与矩阵维度）；
2. 动态分配内存，随机初始化矩阵 A（m×n）和 B（n×k）；
3. 创建若干线程，分别并行计算结果矩阵 C 的不同部分；
4. 等待所有线程完成后，统计运行时间；
5. 若矩阵较小，则输出计算结果矩阵 C；
6. 释放所有内存资源。

1.1.1 公共部分

- 矩阵存储与索引宏：

```
// 行优先存储的矩阵索引宏
#define IDX(i, j, cols) ((i) * (cols) + (j))

// 全局矩阵指针
double *A, *B, *C;
int thread_count, m, n, k;
```

采用一维数组模拟二维矩阵，`IDX`宏实现行优先访问，避免多维数组的内存不连续问题。

- 基础输入处理：

```
printf("请输入线程数 m n k (线程数范围1-16, 矩阵维度范围128~2048) : \n");
if (scanf("%d %d %d %d", &thread_count, &m, &n, &k) != 4) {
    printf("输入格式错误! \n");
    return 1;
}
```

统一输入格式校验，确保参数合法性。

- 内存管理：

```
// 分配内存
A = (double*)malloc(m * n * sizeof(double));
B = (double*)malloc(n * k * sizeof(double));
C = (double*)malloc(m * k * sizeof(double));

// 释放内存（最后执行）
free(A); free(B); free(C);
```

使用`malloc`动态分配内存，避免栈溢出风险。

- 矩阵初始化：

```
srand((unsigned)time(NULL));
for (int i = 0; i < m * n; i++) A[i] = (double)(rand() % 10);
for (int i = 0; i < n * k; i++) B[i] = (double)(rand() % 10);
```

使用时间种子生成0-9的随机数，保证每次运行数据不同但范围可控。

- 线程参数结构体：

```
typedef struct {
    int start_row;  // 起始行（两种方案共用）
    int end_row;    // 结束行（不包括）
    // 块划分特有字段...
} ThreadArg;
```

基础结构体包含行划分信息，块划分版本扩展了列坐标字段。

- 时间统计：

```
struct timeval start, end;
gettimeofday(&start, NULL);
/* ...并行计算... */
gettimeofday(&end, NULL);
double time_consumed = (end.tv_sec - start.tv_sec) +
                       (end.tv_usec - start.tv_usec) / 1e6;
```

使用`gettimeofday`获取微秒级计时，包含系统时间开销。

- 结果验证输出：

```
if (m <= 10 && n <= 10 && k <= 10) {
    printf("矩阵A:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
            printf("%6.2f ", A[IDX(i,j,n)]);
        printf("\n");
    }
    // 类似输出B和C...
}
```

小矩阵时打印完整内容，便于调试验证正确性。

1.1.2 行划分版本（`row_pthread.cpp`）

本方案按**行粒度**将矩阵 A 的多行划分给不同线程，每个线程计算其负责的所有行在结果矩阵 C 中的对应值。

- 线程任务划分

通过如下逻辑将 A 的所有行尽可能均匀地划分给多个线程，每个线程记录其 `start_row` 和 `end_row`：

```
int rows_per_thread = m / thread_count;
int remainder = m % thread_count;
```

前 remainder 个线程每人多分到 1 行，保证任务平均。

- 线程计算逻辑

每个线程调用 `thread_func`，对其负责的多行执行如下乘法计算：

```
for (int i = targ->start_row; i < targ->end_row; i++) {
    for (int j = 0; j < k; j++) {
        double sum = 0.0;
        for (int l = 0; l < n; l++) {
            sum += A[IDX(i, l, n)] * B[IDX(l, j, k)];
        }
        C[IDX(i, j, k)] = sum;
    }
}
```

这段代码逻辑为经典的三重循环矩阵乘法：A 的第 i 行 与 B 的第 j 列做点积得到 C 的 i 行 j 列。

- 线程创建与同步

主线程负责 `pthread_create()` 启动多个线程，并在之后使用 `pthread_join()` 等待所有线程完成。计算耗时使用 `gettimeofday()` 记录前后时间差。

- 输出与释放资源

若矩阵规模较小（如维度均小于等于 10），则输出矩阵 A、B 与 C；否则仅打印计算耗时。最后手动释放所有动态内存。

1.1.3 块划分版本（`block_pthread.cpp`）

块划分方法将结果矩阵 C 划分为若干小矩形块（二维子矩阵），每个线程负责一个块的计算。其实现主要在以下几个方面与行划分不同：

- 网格划分策略

将线程数分解为 `row_blocks × col_blocks` 的二维布局：

```
void compute_grid(...) {
    int r = sqrt(thread_count);
    while (r > 0) {
        if (thread_count % r == 0) {
            *row_blocks = r;
            *col_blocks = thread_count / r;
            return;
        }
        r--;
    }
```

这确保每个线程可以独立处理一个 `(block_row × block_col)` 的区域。

- 任务划分方式

每个线程计算一个子块，其坐标范围通过 `row_start ~ row_end` 与 `col_start ~ col_end` 指定。例如：

```
args[thread_id].row_start = current_row;
args[thread_id].row_end = current_row + block_rows;
args[thread_id].col_start = current_col;
args[thread_id].col_end = current_col + block_cols;
```

这种二维划分方式在理论上可以更好地利用 CPU 缓存局部性，特别是在多核多级缓存系统下。

- 计算逻辑

线程内部的计算逻辑与行划分几乎相同，只是行列范围为对应的块边界：

```
for (int i = targ->row_start; i < targ->row_end; i++) {
    for (int j = targ->col_start; j < targ->col_end; j++) {
        ...
    }
}
```

- 线程创建顺序

线程是以二维嵌套循环方式创建（每个块对应一个线程），而非按行顺序。

1.2 运行测试

使用修改版的`evaluate.py`脚本进行自动化测试，这里仅仅展示线程数和矩阵规模

```
implementations = {
    "行划分": {
        "source": "row_pthread.cpp",
        "binary": "row_exec",
        "thread_counts": [1, 2, 4, 8, 16],
        "matrix_sizes": [128, 256, 512, 1024, 2048]
    },
    "块划分": {
        "source": "block_pthread.cpp",
        "binary": "block_exec",
        "thread_counts": [1, 2, 4, 8, 16],
        "matrix_sizes": [128, 256, 512, 1024, 2048]
    }
}
```

1.3 表格展示

行划分 - 测试结果表格（单位：秒）

线程数	128	256	512	1024	2048
1	0.002228	0.029287	0.752014	6.345801	90.402978
2	0.001246	0.013822	0.355997	3.088190	52.618956
4	0.000960	0.007432	0.178382	1.452069	27.435417
8	0.000911	0.003992	0.081713	0.696357	15.473034
16	0.001144	0.003401	0.058410	0.377127	11.676839

块划分 - 测试结果表格（单位：秒）

线程数	128	256	512	1024	2048
1	0.002073	0.026610	0.742465	7.509419	92.433633
2	0.001693	0.013117	0.347714	3.021394	51.060233
4	0.001047	0.008375	0.177453	1.452103	26.221259
8	0.000725	0.003941	0.087281	0.782877	15.235888
16	0.001166	0.003471	0.050310	0.383902	11.370524

1.4 表格分析

1.4.1 行划分趋势分析

1. 单线程表现：
   - 执行时间随矩阵尺寸呈立方增长（符合矩阵乘法复杂度 $O(n^3)$）。例如，128×128 矩阵耗时 **0.0022秒**，而 2048×2048 矩阵耗时 **90.4秒**。
2. 多线程表现：
   - 加速比：
     - 线程数增加时，时间显著减少，但加速比逐渐趋缓。例如：
       - **128×128**：线程数从1增至16，时间减少 **50%**（0.0022 → 0.0011秒）。
       - **2048×2048**：线程数从1增至16，时间减少 **87%**（90.4 → 11.68秒）。
     - 高线程数（如16线程）时，加速比低于理想线性加速（如16线程理论加速比为16倍，实际为7.7倍），因线程竞争和内存访问冲突导致效率下降。
   - 线程数与性能拐点：
     - 在 小矩阵（如**128**）：线程数增加到16时，时间持续下降，无明显瓶颈。
     - 在 大矩阵（如**2048**）：线程数增至16后，时间继续下降，但加速比趋缓，可能因线程间数据竞争或调度开销增大。
3. 任务划分特点：
   - 行划分：将矩阵按行均分，每个线程独立计算其负责的行。
   - 劣势：
     - 缓存不友好：单行计算时，B矩阵的列数据需频繁从内存加载，导致缓存利用率低。
     - 任务不均衡：若某行计算复杂度较高（如矩阵尺寸不规则），可能导致负载不均。

1.4.2 块划分趋势分析

1. 单线程表现：
   - 与行划分类似，执行时间随矩阵尺寸增长呈立方关系，但 单线程性能略优。
     - 例如：128×128时，块划分耗时 **0.00207秒**，比行划分的0.0022秒快约6%。
2. 多线程表现：
   - 加速比：
     - 大矩阵（如**2048**）：线程数16时，时间 **11.37秒**，比行划分的11.68秒更快（加速比提升约2.6%）。
     - 小矩阵（如**128**）：线程数16时，时间 **0.00116秒**，与行划分的0.0011秒接近。
   - 优势：
     - 缓存局部性：块划分通过二维划分，使计算的A块和B块数据更可能被缓存复用，减少内存访问延迟。
     - 负载均衡：块划分任务更均匀，减少线程空闲时间。

3. 线程数与性能拐点：
   - 在 大矩阵（如**2048**）：线程数增至16时，时间持续下降，加速比优于行划分。
   - 在 中等矩阵（如**512**）：线程数8时，块划分耗时 **0.087秒**，略高于行划分的0.081秒，可能因块划分任务粒度过细导致线程创建开销占优。

1.4.3 两者的对比

对比维度	行划分	块划分
单线程性能	较弱（缓存效率低）	更优（缓存友好性更好）
多线程加速比	高线程数时加速比趋缓显著	高线程数时加速比下降更平缓
大矩阵表现	高线程数下仍有效，但效率低于块划分	更优（缓存复用减少内存延迟）
任务划分特点	行级划分，任务简单但缓存不友好	块级划分，任务均衡且缓存利用率高
小矩阵适用性	线程数较少时表现接近块划分	单线程时更优，但高线程时开销可能更大
关键优势	实现简单，适合小规模或低线程场景	高并行效率，适合大规模和高线程场景

2 并行数组求和

2.1 代码介绍

本实验实现了两种使用 Pthreads 实现的并行数组求和方案，分别为"**Mutex聚合**"（`sum_pthread_mutex.cpp`）和"**局部聚合**"（`sum_pthread_local.cpp`）。两种方法都遵循如下基本流程：

1. 读取输入参数（线程数与数组长度）；
2. 动态分配内存，随机初始化数组（取值0-9）；
3. 创建若干线程，分别并行计算数组的不同部分和；
4. 聚合部分和得到最终结果；
5. 统计运行时间并输出结果；
6. 释放所有内存资源。

2.1.1 公共部分

- 输入与初始化：
  程序首先会提示用户输入线程数和数组长度，具体代码如下：

```
printf("请输入线程数和数组长度 n（数组规模范围1M~128M）：\n");
if (scanf("%d %d", &thread_count, &n) != 2) {
    printf("输入格式错误！\n");
    return 1;
}
```

  接着，程序会动态分配数组内存，为数组`A`和存储每个线程部分和的数组`partial_sums`（在局部聚合版本中）分配内存空间。代码如下：

```
A = (int*) malloc(n * sizeof(int));
partial_sums = (long long*) malloc(thread_count * sizeof(long long));
if (A == NULL || partial_sums == NULL) {
    printf("内存分配失败！\n");
    return 1;
}
```

这里使用`malloc`函数分别为数组`A`分配了`n`个`int`类型大小的内存空间，为`partial_sums`数组分配了`thread_count`个`long long`类型大小的内存空间。`if`语句检查内存分配是否成功，如果`A`或`partial_sums`为`NULL`，说明内存分配失败，此时会输出错误提示信息并返回1，终止程序的执行。

然后，程序使用`rand()`函数生成0-9范围的随机数初始化数组`A`，具体代码为：

```
srand(time(NULL));
for (int i = 0; i < n; i++) {
    A[i] = rand() % 10;
}
```

`srand`函数用于设置随机数生成器的种子，这里使用当前时间`time(NULL)`作为种子，确保每次运行程序时生成的随机数序列不同。`for`循环遍历数组`A`的每个元素，使用`rand() % 10`生成0到9之间的随机整数，并将其赋值给数组`A`的对应元素，从而完成数组的初始化。

- 任务划分：
为了将数组尽可能均匀地划分给多个线程进行处理，程序采用以下逻辑进行任务划分：

```
int chunk = n / thread_count;
int remainder = n % thread_count;
```

在后续的线程创建过程中，会根据这个分配逻辑为每个线程设置其负责处理的数组元素范围。例如，在局部聚合版本中，代码如下：

```
int current = 0;
for (int i = 0; i < thread_count; i++) {
    args[i].thread_id = i;
    args[i].start = current;
    args[i].end = current + chunk + (i < remainder ? 1 : 0);
    current = args[i].end;
    pthread_create(&threads[i], NULL, thread_func, &args[i]);
}
```

这里的`current`变量用于记录当前已经分配的数组元素的位置。`for`循环遍历每个线程，为每个线程的参数结构体`args[i]`设置线程ID、负责处理的数组元素起始位置`start`和结束位置`end`（不包括`end`）。其中，`args[i].end`的计算逻辑考虑了余数的情况，如果当前线程序号`i`小于`remainder`，则该线程会多分配一个元素。最后，使用`pthread_create`函数创建线程，并将线程参数传递给线程函数`thread_func`。

2.1.2 Mutex聚合版本（`sum_pthread_mutex.cpp`）

- 关键机制

使用互斥锁（mutex）保护全局和变量：

```
pthread_mutex_t mutex;
long long global_sum = 0;
```

- 线程计算逻辑

每个线程先计算自己的部分和，然后通过互斥锁安全地更新全局和：

```
void* thread_func(void* arg) {
    // 计算部分和
    long long partial_sum = 0;
    for (int i = targ->start; i < targ->end; i++) {
        partial_sum += A[i];
    }
    // 加锁更新全局和
    pthread_mutex_lock(&mutex);
    global_sum += partial_sum;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

- 同步机制

  主线程通过 `pthread_join` 等待所有线程完成，无需额外的结果聚合步骤。

2.1.3 局部聚合版本（`sum_pthread_local.cpp`）

- 关键机制

  使用线程局部存储（每个线程独立的部分和数组）：

  ```
  long long *partial_sums; // 每个线程一个元素
  ```

- 线程计算逻辑

  每个线程将结果存入自己的部分和槽位：

  ```
  void* thread_func(void* arg) {
      long long sum = 0;
      for (int i = targ->start; i < targ->end; i++) {
          sum += A[i];
      }
      partial_sums[targ->thread_id] = sum;
      return NULL;
  }
  ```

- 结果聚合

  主线程在所有线程完成后，累加部分和数组：

  ```
  long long total_sum = 0;
  for (int i = 0; i < thread_count; i++) {
      total_sum += partial_sums[i];
  }
  ```

2.2 运行测试

使用 `evaluate2.py` 脚本进行自动化测试，测试配置如下：

```
implementations = {
    "Mutex聚合": {
        "source": "sum_pthread_mutex.cpp",
        "binary": "sum_mutex_exec",
        "thread_counts": [1, 2, 4, 8, 16],
        "array_sizes": [1000000, 4000000, 8000000, 16000000, 32000000, 64000000, 128000000]
    },
    "局部聚合": {
        "source": "sum_pthread_local.cpp",
```

```
        "binary": "sum_local_exec",
        "thread_counts": [1, 2, 4, 8, 16],
        "array_sizes": [1000000, 4000000, 8000000, 16000000, 32000000, 64000000, 128000000]
    }
}
```

2.3 表格展示

Mutex聚合 - 测试结果表格（单位：秒）

线程数	1000000	4000000	8000000	16000000	32000000	64000000	128000000
1	0.000913	0.001601	0.003262	0.005607	0.009670	0.017735	0.035513
2	0.000718	0.000764	0.001588	0.002913	0.005058	0.008594	0.041599
4	0.000513	0.000778	0.001315	0.002692	0.004283	0.010157	0.012636
8	0.000619	0.001054	0.001604	0.002871	0.003726	0.007854	0.012716
16	0.001499	0.001137	0.001299	0.002117	0.003860	0.008178	0.013597

局部聚合 - 测试结果表格（单位：秒）

线程数	1000000	4000000	8000000	16000000	32000000	64000000	128000000
1	0.000645	0.001578	0.003532	0.004738	0.009468	0.017550	0.045961
2	0.000427	0.001199	0.001435	0.003286	0.004928	0.008684	0.018588
4	0.000429	0.000782	0.001241	0.001952	0.003788	0.007899	0.012680
8	0.000546	0.001032	0.001453	0.001923	0.003457	0.006891	0.013164
16	0.001171	0.001672	0.001266	0.002298	0.004447	0.007489	0.011703

2.4 表格分析

2.4.1 Mutex聚合趋势分析

1. 单线程表现：
   - 单线程执行时间随数据量线性增长，符合预期（如1e6数据量耗时0.000913秒，1.28e8数据量耗时0.0355秒）。
2. 多线程表现：
   - **线程数增加初期**：执行时间显著下降（例如，数据量为1e6时，线程数从1增至2，时间减少约21%）。
   - **线程数增加后期**：当线程数超过一定阈值（如16线程）时，执行时间反而上升，尤其是大数据量（如1.28e8数据量下，16线程耗时0.0136秒，高于8线程的0.0127秒）。
   - **核心原因**：锁竞争导致线程频繁等待，同步开销超过并行计算收益，性能出现瓶颈。
3. 数据量与线程数的交互影响：
   - 在较小数据量（如1e6）时，线程数过多（如16线程）反而降低效率，因线程切换和锁竞争开销超过并行收益。
   - 大数据量（如1.28e8）时，线程数增至8后性能提升明显，但进一步增至16时因锁竞争导致性能下降。

2.4.2 局部聚合趋势分析

1. 单线程表现：
   - 单线程性能优于Mutex聚合（如1e6数据量耗时0.000645秒 vs Mutex的0.000913秒），因无锁开销。
2. 多线程表现：
   - **线程数增加初期**：执行时间显著下降（如数据量1e6时，线程数从1增至2，时间减少33%）。
   - **线程数增加后期**：
     - **小数据量**（如1e6）：线程数增至16时，时间反而上升（0.001171秒 vs 线程8的0.000546秒），因线程切换开销超过并行收益。
     - **大数据量**（如1.28e8）：线程数增至16时，时间持续下降（0.0117秒 vs 线程8的0.0131秒），因局部变量减少同步开销，线程并行效率更高。
3. 稳定性与扩展性：
   - 局部聚合在高线程数下表现更稳定，尤其在大数据量时性能随线程数增加持续提升，而Mutex因锁竞争导致性能波动。

2.4.3 两者对比

对比维度	**Mutex聚合**	局部聚合
单线程性能	较差（存在锁开销）	更优（无锁开销）
多线程性能	线程数过多时因锁竞争导致性能下降	线程数增加在大数据量时持续提升性能
锁竞争影响	显著（高线程数下性能瓶颈明显）	几乎无（局部变量减少同步需求）
最佳线程数阈值	需根据数据量调整（如8线程为临界点）	更适合高线程数（尤其在大数据量时）
小数据量表现	线程数过多时效率下降，但趋势较平滑	线程数过多时效率下降更明显