中山大學
SUN YAT-SEN UNIVERSITY

并行程序设计与算法实验

Lab3-Pthreads 并行矩阵乘法与数组求和

姓 名 李源卿

学 号 22336128

学 院 计算机学院

专 业 计算机科学与技术

2025 年 4 月 9 日

1 实验目的

- Pthreads 程序编写、运行与调试
- 多线程并行矩阵乘法
- 多线程并行数组求和

2 实验内容

- 掌握 Pthreads 编程的基本流程
- 理解线程间通信与资源共享机制
- 通过性能分析明确线程数、数据规模与加速比的关系

2.1 并行矩阵乘法

- 使用 Pthreads 实现并行矩阵乘法
- 随机生成 $m \times n$ 的矩阵 A 及 $n \times k$ 的矩阵 B
- 通过多线程计算矩阵乘积 $C = A \times B$
- 调整线程数量 (1-16) 和矩阵规模 (128-2048)，记录计算时间
- 分析并行性能 (时间、效率、可扩展性)
- 选做：分析不同数据划分方式的影响
  - 请描述你的数据/任务划分方式。
  - 回答：.....

2.2 并行数组求和

- 使用 Pthreads 实现并行数组求和
- 随机生成长度为 n 的整型数组 A，n 取值范围 [1M, 128M]
- 通过多线程计算数组元素和 $s = \sum_{i=1}^{n} A_i$
- 调整线程数量 (1-16) 和数组规模 (1M-128M)，记录计算时间
- 分析并行性能 (时间、效率、可扩展性)

- 选做：分析不同聚合方式的影响
  - 请描述你的聚合方式。
  - 回答：.....

3 实验代码说明

3.1 并行矩阵乘法

维护如下结构体, 进行静态块切分：

```
typedef struct {
    int start_row;
    int end_row;
    int N;
    double *C;  // 结果矩阵C仍通过参数传递
} ThreadArg;
...
pthread_t threads[num_threads];
ThreadArg args[num_threads];
int rows_per_thread = N / num_threads;
int remaining = N % num_threads;
int current_start = 0;
for (int i = 0; i < num_threads; i++) {
    args[i].start_row = current_start;
    //如果不能刚好分配，编号小一些的线程回干多一些的活。
    args[i].end_row = current_start + rows_per_thread + (i <
       remaining ? 1 : 0);
    args[i].N = N;
    args[i].C = C_parallel;
    //创建线程，传入线程标识，线程属性（一般为NULL），入口函数，参数
       列表
    pthread_create(&threads[i], NULL, matrix_mult, &args[i]);
    current_start = args[i].end_row;
}
```

线程之间的通信相对进程来说要方便很多，由于我在实验中只会对 A 和 B 矩阵进行读操作，所以我把 A 矩阵和 B 矩阵存为了全局变量，并且由于 C 矩阵的每个元素的计算都是相互独立的，所以在把结果写入 C 矩阵的时候，我没有上锁。

```
void *matrix_mult(void *arg) {
```

```
    //指针类型转换
    ThreadArg *t_arg = (ThreadArg *)arg;
    for (int i = t_arg->start_row; i < t_arg->end_row; i++) {
        for (int j = 0; j < t_arg->N; j++) {
            double sum = 0.0;
            for (int k = 0; k < t_arg->N; k++) {
                // 直接访问全局变量A和B
                sum += A[i * t_arg->N + k] * B[k * t_arg->N + j];
            }
            t_arg->C[i * t_arg->N + j] = sum;
        }
    }
    pthread_exit(NULL);
}
```

3.2 并行数组求和

在该实验中，每个线程会将部分和的结果存入 partial_sums[thread_id]，然后在所有的子线程都终止后再由主线程汇总：

```
// 等待线程结束
for (int i = 0; i < num_threads; ++i) {
    pthread_join(threads[i], NULL);
}
// 汇总结果
long long total_sum = 0;
for (int i = 0; i < num_threads; ++i) {
    total_sum += partial_sums[i];
}
```

部分和的计算如下所示，由于是静态的块切分，所以得到答案之后就直接存到了 partial_sums 数组中，这里在后面使用动态的块循环切分的时候要加以修改。

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

```
}
```

3.3 动态块-循环切分

我对数据切分做了改进，我将原本的静态的块切分变为了动态的块-循环切分，一方面，每个线程不会一次性“领取到”自己的所有任务，而是当自己的任务做完之后，再去”领取任务”。

为了实现这一机制，在矩阵乘法中，我维护了两个变量：start_row 和 end_row。每个线程刚开始做计算或者做刚完一次计算的时候都会来读取这两个变量，来知道自己需要去算矩阵 C 的哪一行到哪一行，这就在矩阵乘法里实现了动态的块-循环切分。只不过读取这两个变量时要上锁，且应该在上锁期间将两个变量都加上一个 BLOCK_SIZE。

```
int start_row=-BLOCK_ROWS;
int end_row=0;
int N;
    while (1)
    {
        int st,ed;
        pthread_mutex_lock(&mtx);
        start_row += BLOCK_ROWS;
        end_row += BLOCK_ROWS;
        if (start_row >= N)
        {
            pthread_mutex_unlock(&mtx);
            pthread_exit(NULL);
        }
        end_row = end_row < N ? end_row : N;
        st = start_row; ed = end_row;
        printf("thread:%ld,start:%d,end:%d\n", pthread_self()%100,
           start_row, end_row);
        pthread_mutex_unlock(&mtx);
        ...
        （其余不变）
    }
```

至于数组求和，同样可以维护两个变量 start 和 end，线程读到这两个变量就会知道自己需要求和的区间。同理，每次访问这两个变量的时候需要上锁，访问完之后需要更新这两个变量。另外，局部求和的函数需要稍微修改。*(args->partial_sum) = sum; 需要改为 *(args->partial_sum) = *(args->partial_sum) + sum; 因为线程可能会“领取”多次任务。

```
int start=0;
int end=BLOCK_SIZE;
typedef struct {
    int *array;
    long long *partial_sum;
} ThreadArgs;
...
void* compute_sum(void *arg) {
    while (1)
    {
        int st, ed;
        pthread_mutex_lock(&mtx);
        if(start >= array_size){
            pthread_mutex_unlock(&mtx);
            pthread_exit(NULL);
        }
        st = start;
        ed = end < array_size ? end : array_size;
        start += BLOCK_SIZE;
        end += BLOCK_SIZE;
        // end =
        pthread_mutex_unlock(&mtx);
        ...(求和)
        *(args->partial_sum) = *(args->partial_sum) + sum;
    }
}
```

4 实验结果

4.1 结果验证

4.1.1 并行矩阵乘法

本次实验没有使用 numpy 验证，而是写一个简单的串行程序，用串行程序的结果和并行程序结果进行比较。

```
int correct = 1;
for (int i = 0; i < N * N; i++) {
    if (fabs(C_serial[i] - C_parallel[i]) != 0) {
```

```
        correct = 0;
        break;
    }
}
```

尝试了一些正常样例，比如 4×4 矩阵，2 线程；以及一些极端一些的样例，比如 79×79 矩阵,4 线程，都得到了正确答案。

4.1.2 并行数组求和

在此处我将数组中所有的数据都初始化为 1，所以只需要看结果是否等于数组大小即可

```
int *array = malloc(array_size * sizeof(int));
for (int i = 0; i < array_size; ++i) {
    array[i] = 1; // 全1数组方便验证结果
}
同样尝试了常规样例和极端样例，都得到了正确答案。
```

4.2 并行矩阵乘法

表 1: 并行矩阵乘法在不同线程数下的运行时间
从这一块开始，为了节省时间，我把串行部分的代码都注释掉了。

矩阵规模	1 线程	2 线程	4 线程	8 线程	16 线程
128×128	0.006606s	0.004127s	0.003338s	0.003072s	0.002736s
256×256	0.056041s	0.029643s	0.021376s	0.015199s	0.015636s
512×512	0.573044s	0.323702s	0.263435s	0.185210s	0.172837s
1024×1024	11.073968s	4.123620s	2.187723s	2.309699s	2.698541s
2048×2048	134.192716s	72.018253s	54.622610s	50.211454s	50.726002s

4.3 并行数组求和

4.4 动态块-循环切分

这部分的数据都是使用 4 线程跑出来的, 并且，块大小都是针对原划分（块切分）大小而言的，下面会用分数表示块大小（即原大小的几分之几）。

表 2: 数组求和不同线程数下的运行时间

数组规模	1 线程	2 线程	4 线程	8 线程	16 线程
1M	0.002371s	0.001300s	0.000843s	0.000968s	0.000971s
4M	0.009364s	0.004337s	0.002882s	0.002175s	0.003167s
16M	0.032915s	0.019150s	0.012836s	0.007171s	0.008061s
64M	0.145811s	0.086346s	0.041103s	0.034744s	0.035014s
128M	0.260360s	0.132760s	0.096561s	0.064181s	0.065544s

4.4.1 矩阵乘法

表 3: 四线程并行矩阵乘法在不同块大小下的运行时间

块大小 矩阵规模	1/16	1/8	1/4	1/2	1
128×128	0.002094s	0.003256s	0.001866s	0.002905s	0.002294s
256×256	0.019539s	0.018951s	0.019808s	0.019945s	0.020029s
512×512	0.156754s	0.148737s	0.147824s	0.151145s	0.149133s
1024×1024	2.498613s	2.603864s	2.288042s	2.540046s	2.710552s
2048×2048	58.875086s	57.863714s	57.127374s	53.276943s	57.176811s

4.4.2 数组求和

表 4: 四线程数组求和不同块大小下的运行时间

块大小 数组规模	1/16	1/8	1/4	1/2	1
1M	0.001208s	0.001018s	0.001253s	0.000878s	0.001469s
4M	0.002845s	0.002312s	0.002539s	0.003187s	0.003251s
16M	0.009681s	0.009835s	0.010636s	0.011589s	0.008545s
64M	0.036635s	0.040212s	0.036093s	0.036158s	0.034899s
128M	0.067312s	0.070636s	0.075666s	0.080182s	0.082353s

5 实验分析

这次的实验结果是我在自己的电脑上跑的，与之前使用 MPI 有较大的规律上的区别。一方面是因为我这一次的矩阵乘法没有调整循环顺序；另一方面，可能是操作系统

对线程和进程的调度策略不同。

5.1 并行矩阵乘法

（这是另外一个结果，是再次运行程序算出来的，不是由上面的直接得出）

- 线程数量对性能的影响分析：线程数增大为原来两倍，程序执行时间不一定减半，但是在线程小于物理核数的情况下，执行时间会减少，这是因为在这样的情况下，每多开一个线程都能分配到一个核，如果多于物理核数，就要调度。
- 矩阵规模对并行效率的影响：矩阵规模的增大，效率会略微上升。
- 可扩展性分析：由于随着核数的增大，问题规模不变的情况下，效率降低，所以是弱可拓展性的。
- (选做) 不同数据划分方式的比较：动态的块-循环切分在矩阵乘法的问题上不一定能胜过传统的静态块切分方法。一方面因为矩阵大小很规整，静态切分的负载是均衡的；另一方面不同时间跑的程序，其运行时间会和机器当时的状态有关（会有误差）。

表 5: 并行矩阵乘法在不同线程数下的效率

矩阵规模	2 线程	4 线程	8 线程	16 线程
128×128	79.41%	38.17%	29.09%	13.79%
256×256	79.52%	72.31%	36.11%	18.20%
512×512	75.60%	60.07%	36.64%	19.31%
1024×1024	84.96%	53.61%	46.97%	19.17%

5.2 并行数组求和

- 线程数量对性能的影响分析：与矩阵乘法一样，线程数加倍，时间达不到减半。但是在线程小于物理核数的情况下，执行时间会减少。
- 数组规模对并行效率的影响：数组规模增大，大部分情况计算得到的效率是上升的。
- 可扩展性分析：线程数增大，但是问题规模不变的情况下，效率降低，是弱可拓展性。
- 不同数据划分方式的比较：动态的块-循环切分在数组求和的问题上也不一定能胜过传统的静态块切分方法。原因也和上面的（矩阵乘法）是一致的。

表 6: 数组求和不同线程数下的效率

数组规模	2 线程	4 线程	8 线程	16 线程
1M	84.47%	57.85%	41.77%	8.92%
4M	88.18%	65.26%	51.69%	23.43%
16M	95.52%	74.78%	60.01%	29.59%
64M	96.25%s	91.74%	66.22%	27.52%
128M	97.38%s	84.31%	66.52%	31.29%