中山大學
SUN YAT-SEN UNIVERSITY

并行程序设计与算法实验

Lab5-基于 OpenMP 的并行矩阵乘法

姓 名 马岱

学 号 22336180

学 院 计算机学院

专 业 计算机科学与技术

2025 年 4 月 23 日

1 实验目的

- 掌握 OpenMP 编程的基本流程
- 掌握常见的 OpenMP 编译指令和运行库函数
- 分析调度方式对多线程程序的影响

2 实验内容

- 使用 OpenMP 实现并行通用矩阵乘法
- 设置线程数量（1-16）、矩阵规模（128-2048）、调度方式
  - 调度方式包括默认调度、静态调度、动态调度
- 根据运行时间，分析程序并行性能
- 选做：根据运行时间，对比使用 OpenMP 实现并行矩阵乘法与使用 Pthreads 实现并行矩阵乘法的性能差异，并讨论分析。

2.1 代码思路

本次实验不是很复杂，相较于 pthread，OpenMP 不需要我们定义线程、线程函数以及手动管理线程的创建与回收，也无需显式地设计互斥锁、条件变量等同步机制。只需通过在循环前添加 '#pragma omp parallel for' 指令，OpenMP 运行时便能自动将循环任务划分给不同线程并发执行。本次实验我们也仅需要在之前的 pthread 的代码基础上修改即可。接下来我将给出几个具体需要注意的点。

由于 openmp 的调度方式有多种，默认调度、静态调度和动态调度，其中默认调度通常由编译器根据系统环境和任务特性自动选择。静态调度会在程序开始时将所有迭代任务平均划分到各线程，调度开销较小。动态调度则是在运行时动态分配任务，每当某线程完成分配任务后，再从任务队列中领取新任务，能够有效避免线程空闲，提高资源利用率，但调度开销相对更大。

```
#pragma omp parallel for num_threads(THREAD_NUMS) //default schedule
#pragma omp parallel for num_threads(THREAD_NUMS) schedule(static, chunk_size)
#pragma omp parallel for num_threads(THREAD_NUMS) schedule(dynamic, chunk_size)
```

默认调度 默认调度实现并不复杂，只需在 `#pragma omp parallel for` 指令中指定线程数量，OpenMP 将自动采用默认策略进行调度。该方式无需手动指定调度策略，由编译器根据当前系统环境和线程数自动选择调度方式，适合任务耗时基本均衡的矩阵乘法计算场景。

```
#pragma omp parallel for num_threads(num_threads)
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < k; j++)
            {
                C[i * k + j] = 0.0;
                for (int p = 0; p < n; p++)
                {
                    C[i * k + j] += A[i * n + p] * B[p * k + j];
                }
            }
        }
```

静态调度 静态调度方式下，OpenMP 将循环迭代次数平均分配给各线程，每个线程在程序执行开始前就已经确定好自己所负责的迭代区间，适合任务耗时均衡的场景。这里我使用了 `schedule(static, 1)`，表示将迭代任务以块大小为 1 的方式，按顺序静态分配给各线程，保证各线程任务数量基本一致。

```
#pragma omp parallel for num_threads(num_threads) schedule(static, 1)
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < k; j++)
            {
                C[i * k + j] = 0.0;
                for (int p = 0; p < n; p++)
                {
                    C[i * k + j] += A[i * n + p] * B[p * k + j];
                }
            }
        }
```

动态调度 动态调度方式下，OpenMP 将迭代任务划分成若干块，线程在运行时动态申请任务块，其中，`schedule(dynamic, 1)` 表示将迭代任务以块大小为 1 的方式划分，线程在运行时动态获取任务块执行。

```
#pragma omp parallel for num_threads(num_threads) schedule(dynamic, 1)
for (int i = 0; i < m; i++)
{
    for (int j = 0; j < k; j++)
    {
        C[i * k + j] = 0.0;
        for (int p = 0; p < n; p++)
        {
            C[i * k + j] += A[i * n + p] * B[p * k + j];
        }
    }
}
```

2.2 影响因素

这个实验还有一个地方需要我们额外去考虑，就是 chunksize 大小的设置，chunksize 被称为块大小，指的是每次给线程分配的连续迭代次数。对于小的 chunk（如 1）能够让线程频繁地去“抢”或“平均”新任务，负载更加均衡，但每次分配都要执行 OpenMP 的内部调度逻辑，开销大。但对于大的 chunk（如 16、32）意味着每次分配更多迭代，调度次数减少，开销小，但如果某些迭代耗时比其它多，就会出现某个线程工作量过大，其它线程空闲的情况。因此我们有必要考虑和研究 chunksize 对实验结果的影响。

在本次实验中我分别对静态和动态调度在 4 线程 2048 矩阵规模下实验对比，分别设置 chunksize 大小为 1、2、4、8、16，这里给出运行脚本：

```
#!/bin/bash
# 自动化测试 OpenMP 矩阵乘法：4 线程、2048×2048 矩阵下 static vs dynamic 调度
↪  chunk_size 对比

exe="chunk"
src="chunk.c"
# 编译
gcc -O2 -fopenmp -o $exe $src
if [ $? -ne 0 ]; then
    echo "  编译失败，请检查 $src"
    exit 1
fi

# 参数固定为 4 线程、2048×2048 矩阵
M=2048; N=2048; K=2048; THREADS=4
CHUNKS=(1 2 4 8 16)
```

```
16
17 # 准备输出
18 log="chunk_compare.log"
19 csv="chunk_compare.csv"
20 echo "chunk,static_time,dynamic_time" > $csv
21 echo "OpenMP chunk_size 对比实验" > $log
22 echo " 测试日期: $(date)" >> $log
23 echo "================================" >> $log
24 for chunk in "${CHUNKS[@]}"; do
25     echo "## chunk_size = $chunk" | tee -a $log
26     out=$("./$exe" $M $N $K $THREADS $chunk)
27     echo "$out" | tee -a $log
28
29     static_t=$(echo "$out" | awk '/Static/  {print $2}')
30     dynamic_t=$(echo "$out" | awk '/Dynamic/ {print $2}')
31     echo "$chunk,$static_t,$dynamic_t" >> $csv
32
33     echo "--------------------------------" >> $log
34 done
35
36 echo " 日志: $log"
37 echo "CSV 数据: $csv"
```

表 1: 4 线程、2048×2048 矩阵下不同 chunk size 的调度时间对比（秒）

chunk size	static 调度	dynamic 调度
1	18.966281	15.491277
2	16.841290	14.799872
4	14.486974	13.975019
8	21.056800	14.027065
16	14.469276	13.851610

从表中可以看出，在 4 线程、2048×2048 矩阵规模下，静态调度在 chunk 为 1 时受到非常明显的调度开销和负载不均衡的双重影响，耗时接近 19 s，而当 chunk 扩大到 4 或 16 时，耗时可以降至约 14.5 s，说明较大的块大小能减少线程间切换和调度次数，从而提高缓存局部性并均衡工作量。但在 chunk=8 的情况下，静态调度反而出现了最差的 21 s，可能是块划分刚好与线程数不匹配，造成某些线程空闲而其他线程任务过载。

相比之下，动态调度的表现更加平稳和优秀：无论 chunk 大小如何，耗时总是保持在 13.8 s 至 15.5 s 之间，并在 chunk=16 时达到最优 13.85 s。，动态调度总体上都比静

态调度要快，且随着 chunk size 的增大，两者的性能差距更加明显。综合来看，在大规模矩阵乘法中使用动态调度并设置较大的 chunk（如 16），既能最大化并行吞吐，又能兼顾调度和缓存效益。

3 实验结果

表 2: 默认调度

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.001680 s	0.013984 s	0.198309 s	3.386374 s	39.344966 s
2	0.002125 s	0.007441 s	0.092215 s	1.122719 s	18.913649 s
4	0.001557 s	0.004022 s	0.088559 s	0.606000 s	11.480924 s
8	0.004375 s	0.005701 s	0.062188 s	0.610611 s	8.059403 s
16	0.001808 s	0.005982 s	0.066534 s	0.655350 s	6.541701 s

表 3: 静态调度

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.001681 s	0.011778 s	0.184108 s	2.914096 s	37.571292 s
2	0.001004 s	0.006985 s	0.088173 s	1.121244 s	15.184645 s
4	0.002352 s	0.015709 s	0.060903 s	0.666695 s	6.711124 s
8	0.001272 s	0.007103 s	0.071030 s	0.561011 s	6.867881 s
16	0.001621 s	0.004773 s	0.062754 s	0.595176 s	7.329311 s

表 4: 动态调度

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.001660 s	0.011374 s	0.190452 s	3.034337 s	37.839104 s
2	0.000959 s	0.006563 s	0.098543 s	1.138598 s	15.705387 s
4	0.000711 s	0.016499 s	0.065828 s	0.568955 s	6.518828 s
8	0.001022 s	0.007612 s	0.043779 s	0.551603 s	6.683374 s
16	0.002685 s	0.005735 s	0.051713 s	0.547189 s	7.135875 s

4 实验分析

- 根据运行时间，分析程序并行性能
- 回答：从实验结果的数据来看，三种调度方式其实差异并不是很大，但是默认调度相对来说能差一点。以 4 线程、2048×2048 为例，默认调度耗时约 11.48 s，而静态调度降至 6.71 s，动态调度又进一步略优，为 6.52 s，体现了显式划块和运行时平衡的优势。中等规模（512×512）下，静态调度耗时 0.0609 s，动态为 0.0658 s，默认则为 0.0886 s，可见静态略胜；但是在更细粒度（128×128），动态调度以 0.000711 s 领先于静态的 0.002352 s 和默认的 0.001557 s，说明当迭代开销不完全均匀时，动态抢占能更好地平衡负载。

  综上，静态调度适合大规模、计算量高度均匀的场景以最大化缓存利用和最小化调度开销；而动态调度则在负载波动或小问题上通过细粒度平衡略胜一筹；默认调度因依赖编译器策略，表现最不稳定。在线程数较多或任务不均时优先使用 schedule(dynamic)，在任务均匀且规模足够大时优先使用 schedule(static)。
- 选做题：根据运行时间，对比使用 OpenMP 实现并行矩阵乘法与使用 Pthreads 实现并行矩阵乘法的性能差异，并讨论分析。

表 5: pthread 在不同线程数下的运行时间

进程数	矩阵规模				
	128	256	512	1024	2048
1	0.01232 s	0.06298 s	0.60493 s	9.63817 s	101.69945 s
2	0.00545 s	0.04175 s	0.29455 s	2.73088 s	41.56598 s
4	0.00449 s	0.03200 s	0.15440 s	1.28812 s	20.11231 s
8	0.00533 s	0.02459 s	0.16169 s	1.20201 s	19.29964 s
16	0.01422 s	0.02913 s	0.14499 s	1.34612 s	20.05837 s

表 6: pthread 乘法加速比

进程数	矩阵规模				
	128	256	512	1024	2048
1	1.000	1.000	1.000	1.000	1.000
2	2.261	1.509	2.055	3.529	2.447
4	2.744	1.968	3.915	7.480	5.058
8	2.312	2.562	3.743	8.022	5.268
16	0.865	2.163	4.171	7.163	5.071

- 回答：对比表中 Pthreads（表 5）与 OpenMP（表 3 和 4）在相同线程数、相同矩阵规模下的运行时间，可以看出 OpenMP 实现整体上要快得多。以 4 线程为例 512×512: Pthreads 0.15440 s，OpenMP static 0.06090 s，dynamic 0.06583 s，快约 2.3×。2048×2048：Pthreads 20.11231 s，OpenMP static 6.71112 s，dynamic 6.51883 s，快约 3×。

  这种性能差异源于两方面：一是 OpenMP 在内层循环上使用 ‘collapse(2)‘ 将双重循环打平成一个大并行区，能够更高效地划分工作并利用 SIMD、缓存局部性；而 Pthreads 示例仅在外层按行划分工作，线程间负载与内存访问可优化空间更小。二是 OpenMP 的调度机制（static/dynamic）在运行时和编译器层面都做了高度优化，系统开销远低于手写的 Pthreads 线程创建、分配与同步。因此，在大多数矩阵规模和线程配置下，OpenMP 都能取得显著优于 Pthreads 的性能表现。