南開大學
Nankai University

南　开　大　学

计 算 机 学 院

并行程序设计第四次实验报告

多线程OpenMP编程

丁彦添1911406

年级：2019级

专业：计算机科学与技术

指导教师：王刚

2021 年 5 月 21 日

摘要

本次实验将继续选择默认选题高斯消去法，使用OpenMP与SIMD方法结合对算法进行优化。

关键字：并行；OpenMP；SIMD；SSE；AVX；高斯消元

目录

一、 概述 **1**

（一） 问题描述 1

（二） 相关算法设计与实现 1

（三） 实验环境 4

（四） 实验结果及分析 5

二、 总结 **7**

一、 概述

（一） 问题描述

实现高斯消去法（LU分解），对比实现使用多线程OpenMP结合SSE/AVX等算法加速计算过程。研究不同划分策略（采用矩阵按行划分与按列划分等）对性能的影响。结合第二次实验，对比分析只用OpenMP优化和使用OpenMP与SSE/AVX方法优化效果。对比分析结合第三次实验，对比分析Pthread和OpenMP方法优化。

高斯消去法，是线性代数中的一个算法，主要是将矩阵中各行各列进行线性变化，将方阵化为主对角线全为1的三角方阵，便于后续分解运算。高斯消去法可用来求解线性方程组，并可以求出矩阵的秩，以及求出可逆方阵的逆矩阵。本次实验实现高斯消去法并探讨对其进行并行化优化。具体的伪代码如下：

Algorithm 1 朴素的高斯消去算法

Input: 方阵A

Output: 高斯消元后上三角方阵A

```
1:  function GaussianElimination(matrixA)
2:      for k = 1; k < n; i++ do
3:          for j = k;j < n;j++ do
4:              A[k, j] ← A[k, j]/A[k, k]
5:          end for
6:          for i = k + 1;i < n;i++ do
7:              for j = k + 1;j < n;j++ do
8:                  A[i, j] ← A[i, j] − A[i, k] ∗ A[k, j]
9:              end for
10:             A[i, k] ← 0
11:         end for
12:     end for
13: end function
```

观察高斯消去法的伪代码，可以得知该算法最多嵌套了对于n的三重循环，高斯消去法的时间复杂度为 $O(n^3)$。使用多线程OpenMP进行任务划分以达到加速的目的。

（二） 相关算法设计与实现

本次实验要求：

- 设计实现适合的任务分配算法，分析其性能。
- 与SIMD(SSE/AVX)算法相结合。
- 对比Pthread优化和OpenMP优化。
- 修改线程数，分析线程数对性能的影响。

在第二次实验中已经实现了SSE/AVX循环展开对高斯消去法算法的优化，本次实验将在第二次实验的基础之上增加OpenMP多线程优化方法。

本次实验的第一部分先对Pthread任务划分方式进行探究。观察代码，发现至少存在两种任务划分方式：按行划分和按列划分。实验的第一部分将对这两种划分方式进行探究。分别多次测试这两种划分方式以减少误差，并记录相关实验数据，以分析性能。

OpenMP优化代码如下：

OpenMP优化

```
#pragma omp parallel num_threads(THREAD_NUM) shared(k)
{
    #pragma omp barrier
    int r = omp_get_thread_num();
    for (int k = 0; k < n; k++) {
        for (int i = k + 1; i < n; i++) {
            if ((i % THREAD_NUM) == r) {
                B[i][k] = B[i][k] / B[k][k];
                for (int j = k + 1; j < n; j++) {
                    B[i][j] = B[i][j] - B[i][k] * B[k][j];
                }
            }
        }
    }
}
```

本次实验的第二部分，将采用的第一部分得出的较优划分方式的基础上结合SSE/AVX等SIMD方法。

按行进行划分并结合SSE循环展开代码如下：

使用按行进行划分OpenMP优化结合SSE的SIMD方法

```
#pragma omp parallel num_threads(THREAD_NUM) shared(k)
{
    #pragma omp barrier
    int r = omp_get_thread_num();
    __m128 t1_1, t2, t3;
    for (int k = 0; k < n; k++) {
        for (int i = k + 1; i < n; i++) {
            if ((i % THREAD_NUM) == r) {
                B[i][k] = B[i][k] / B[k][k];
                int offset = (n - k - 1) % 4;
                for (int j = k + 1; j < k + 1 + offset; j++) {
                    B[i][j] = B[i][j] - B[i][k] * B[k][j];
                }
                t2 = _mm_set_ps(B[i][k], B[i][k], B[i][k], B[i][k]);
                for (int j = k + 1 + offset ; j < n; j += 4) {
                    t3 = _mm_load_ps(B[k] + j);
                    t1_1 = _mm_load_ps(B[i] + j);
                    t2 = _mm_mul_ps(t2, t3);
                    t1_1 = _mm_sub_ps(t1_1, t2);
                    _mm_store_ps(B[i] + j, t1_1);
                }
            }
        }
    }
}
```

按行进行划分并结合AVX循环展开代码如下：

使用按行进行划分OpenMP优化结AVX的SIMD方法

```
#pragma omp parallel num_threads(THREAD_NUM) shared(k)
{
    #pragma omp barrier
    int r = omp_get_thread_num();
    __m256 t1_1, t2, t3;
    for (int k = 0; k < n; k++){
        for(int i = k + 1; i < n; i++){
            if(i%THREAD_NUM == r){
                B[i][k] = B[i][k] / B[k][k];
                int offset = (n - k - 1) % 8;
                for(int j = k + 1; j < k + 1 + offset; j ++){
                    B[i][j] = B[i][j] - B[i][k] * B[k][j];
                }
                t2 = _mm256_set_ps(B[i][k], B[i][k], B[i][k], B[i][k],
                B[i][k], B[i][k], B[i][k], B[i][k]);
                for (int j = k + 1 + offset ; j < n; j += 8) {
                    t3 = _mm256_load_ps(B[k] + j);
                    t1_1 = _mm256_load_ps(B[i] + j);
                    t2 = _mm256_mul_ps(t2, t3);
                    t1_1 = _mm256_sub_ps(t1_1, t2);
                    _mm256_store_ps(B[i] + j, t1_1);
                }
            }
        }
    }
}
```

本次实验的第三部分，将朴素的OpenMP优化方法与朴素的Pthread优化方法进行对比，分析性能。Pthread代码如下：

Pthread优化

```
void* LU_pthread(void *parm) {
    threadParm_t *p = (threadParm_t *)parm;
    int r = p->threatId;
    long long tail;
    for (int k = 0; k < n;k++) {
        for (int i = k + 1; i < n; i++) {
            if ((i % THREAD_NUM) == r) {
                B[i][k] = B[i][k] / B[k][k];
                for (int j = k + 1; j<n; j++) {
                    B[i][j] = B[i][j] - B[i][k] * B[k][j];
                }
            }
        }
```

```
14          pthread_barrier_wait(&barrier);
15      }
16      pthread_mutex_lock(&mutex);
17      QueryPerformanceCounter((LARGE_INTEGER *)&tail);
18      cout << "Thread " << r << ": " << (tail - head) * 1000.0 / freq << "ms"
            << endl;
19      pthread_mutex_unlock(&mutex);
20      pthread_exit(0);
21      return 0;
22  }
```

本次实验的第四部分，将修改线程数$THREAD_NUM$，分析线程数对性能的影响

（三） 实验环境

本实验环境如下：

- 操作系统：Windows 10
- CPU：AMD Ryzen 5 3500H with Radeon Vega Mobile Gfx 2.10GHz
- Codeblocks GNU GCC Compiler
- 关闭所有优化选项，增加-fopenmp（编译OpenMP）、-pthread（编译Pthread）和-march=native（编译SSE/AVX）

（四） 实验结果及分析

第一部分实验：

对第一、二部分的代码进行测试，或许相关实验数据，绘制第一、二部分实验结果如下表：

数据规模\优化策略	朴素	使用OpenMP	OpenMP+SSE	OpenMP+AVX
256	15.629ms	10.332ms	8.425ms	7.233ms
512	109.407ms	62.521ms	31.259ms	23.217ms
1024	1719.287ms	437.633ms	296.961ms	250.076ms
2048	13457.198ms	3516.688ms	2156.903ms	1797.184ms

表 1: 第一、二部分实验结果（线程数：4）

根据第一、二部分实验结果，绘制加速比统计图1如下表：

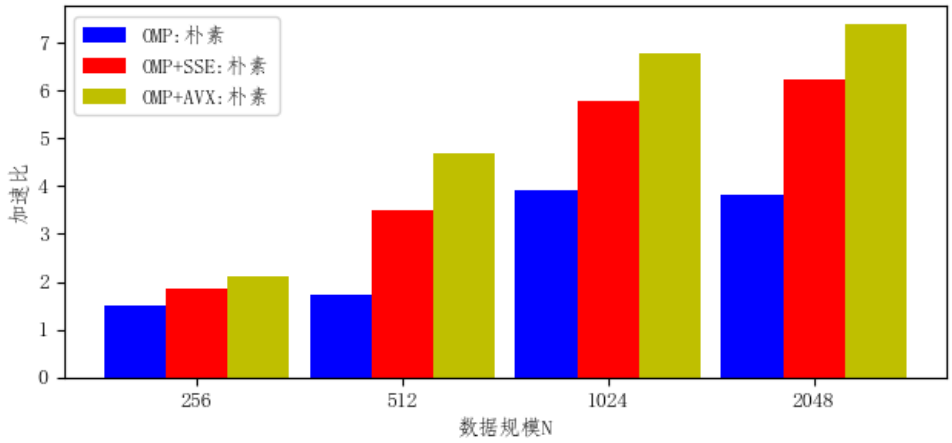

图 1: 实验一、二优化算法相对朴素算法获得的加速比(4线程)

由第一、二部分实验结果可知，在数据规模较小时，不管是只使用OpenMP优化还是OpenMP与SIMD结合优化获得的加速比都不太理想。当数据规模较大时，仅使用OpenMP进行优化能获得接近于四倍的加速比。由于本地平台CPU时AMD，可能对SIMD中的SSE/AVX支持不太好，所有结合上SIMD方法，本地测试优化幅度并不太大。

第三部分实验：对比仅使用OpenMP优化和仅使用Pthread优化，线程数保持为4。比较两者性能差别。对不同规模数据运行，记录算法运行时间。测试结果如下表：

数据规模\优化策略	朴素	使用OpenMP	使用Pthread
256	15.629ms	10.332ms	6.753ms
512	109.407ms	62.521ms	45.151ms
1024	1719.287ms	437.633ms	447.407ms
2048	13457.198ms	3516.688ms	3370.17ms

表 2: 第三部分实验结果（线程数：4）

根据测试结果，绘制加速比统计图2如下表：

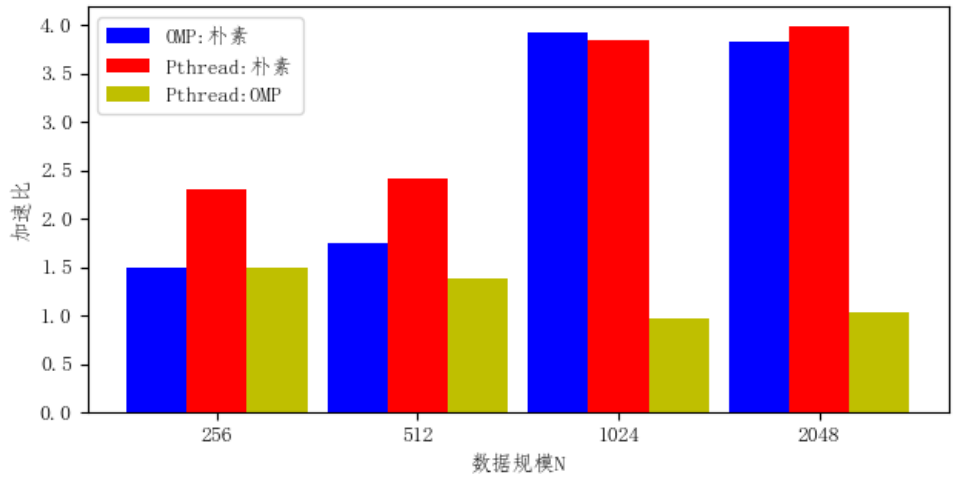

图 2: 实验三相关加速比(4线程)

第四部分实验：

将使用不同的线程数$NUM_THREADS$，固定数据规模为2048，对实验代码进行测试。将从算法执行时间、加速比等多方面对结果进行分析比较。注意到本地平台CPU为4核心8线程，因此线程数取值为：2、4、6、8、10、12。

数据规模\优化策略	朴素	使用OpenMP	OpenMP+SSE	OpenMP+AVX
2	13160.443	6455.077	3203.883	2625.811
4	13457.198	3516.688	2156.903	1797.184
6	13363.417	2547.646	1719.268	1359.766
8	13363.416	2031.865	1406.672	1156.597
10	13097.713	2000.606	1422.305	1156.598
12	13550.973	2047.494	1391.045	1194.078

表 3: 性能测试结果(数据规模2048)(单位:ms)

根据实验结果，求出加速比。绘制加速比统计图3如下：

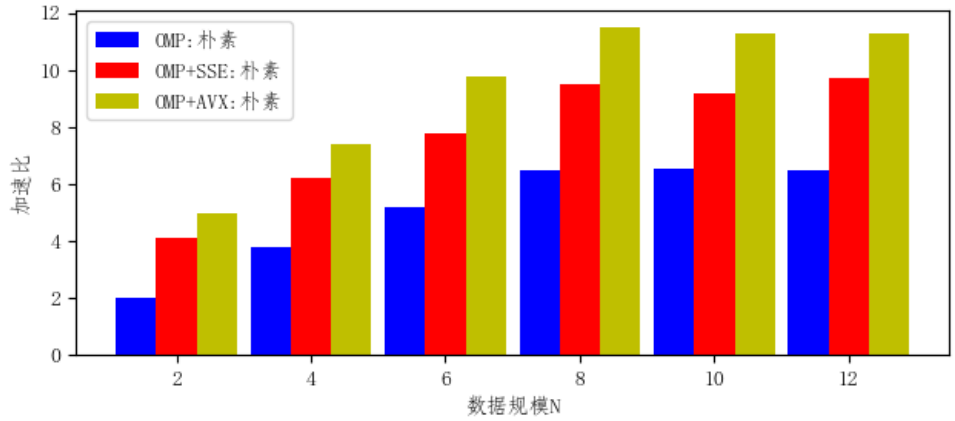

图 3: 不同线程数获得的加速比(数据规模:2048)

根据得到的加速比结果可以知道，在线程数小于8时，可以发现增加线程数可以使加速比稳步增长。在线程数超过8后，加速比基本没有什么变化。有时加速比甚至略微下降。结合体系结构和操作系统相关知识，当线程数过多必然导致大量操作系统调度操作，对性能可能有一定的影响。

二、 总结

本次实验首先使用OpenMP多线程对高斯消元法进行优化，并探究是否结合SIMD相关优化对性能的影响。发现使用OpenMP获得的加速比接近理想加速比值，但是结合SIMD方法并没有对性能有太大的提升。

此外，还比较了OpenMP和Pthread相关性能，发现在误差范围内两者的性能差别不大。并接着探究不同线程数对程序性能的影响，得出在一定范围内，线程数的增加能比较稳定地提升加速比，但线程数超过一定值后，加速比基本上没有变化。

- 在OpenMP多线程优化的基础上进行向量并行化能得到更大的加速比，更优的性能，但与理想加速比差别较大。
- 在误差范围内，认为OpenMP优化和Pthread优化性能差别不大。
- 在一定范围内，线程数的增加能比较稳定地提升加速比，但线程数超过一定值后，加速比基本上没有变化。