南開大學
Nankai University

南　开　大　学

计 算 机 学 院

并行程序设计第三次实验报告

多线程Pthread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 5 日

摘要

关键字：并行；**Pthread**；**SIMD**；**SSE**；**AVX**；高斯消元

目录

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Pthread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。

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

```
function GaussianElimination(matrix A)
    for k = 1; k < n; i++ do
        for j = k;j < n;j++ do
            A[k, j] ← A[k, j]/A[k, k]
        end for
        for i = k + 1;i < n;i++ do
            for j = k + 1;j < n;j++ do
                A[i, j] ← A[i, j] − A[i, k] ∗ A[k, j]
            end for
            A[i, k] ← 0
        end for
    end for
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Pthread进行任务划分，可以对外层循环（按行）进行划分，也可以对内层循环（按列）进行划分。

（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（按行划分和按列划分），分析其性能。
- 与SIMD(SSE/AVX)算法相结合。

在上一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，本次实验将在上次实验的基础之上增加Pthread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，并使用VTune记录相关的性能。

按行进行划分代码如下：

Pthread优化（按行进行划分）

```
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 % NUM_THREADS) == 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];
                }
            }
        }
        pthread_barrier_wait(&barrier);
    }
    pthread_mutex_lock(&mutex);
    QueryPerformanceCounter((LARGE_INTEGER *)&tail);
    cout << "Thread " << r << ": " << (tail - head) * 1000.0 / freq << "ms"
       << endl;
    pthread_mutex_unlock(&mutex);
    pthread_exit(0);
    return 0;
}
```

按列进行划分代码如下：

Pthread优化（按列进行划分）

```
void *LU_pthread_matrix(void *parm) {
    threadParm_t *p = (threadParm_t *)parm;
    int r = p->threatId;
    long long tail;
    for (int k = 0; k < n; k++) {
        for (int j = k + 1 + r;j < n;j += NUM_THREADS) {
            B[k][j] = B[k][j] / A[k][k];
        }
        B[k][k] = 1;
        pthread_barrier_wait(&barrier);
        for (int i = k + 1 + r; i < n; i += NUM_THREADS) {
            for (int j = k + 1;j < n;j++) {
                B[i][j] = B[i][j] - B[i][k] * B[k][j];
            }
            B[i][k] = 0;
        }
        pthread_barrier_wait(&barrier);
    }
    pthread_mutex_lock(&mutex);
    QueryPerformanceCounter((LARGE_INTEGER *)&tail);
    cout << "Thread " << r << ": " << (tail - head) * 1000.0 / freq << "ms"
       << endl;
    pthread_mutex_unlock(&mutex);
```

```
    pthread_exit(0);
    return 0;
}
```

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Pthread优化结合SSE的SIMD方法

```
void* LU_pthread_sse(void *parm) {
    threadParm_t *p = (threadParm_t *)parm;
    int r = p->threatId;
    long long tail;
    __m128 t1, t2, t3;
    for (int k = 0; k < n;k++) {
        for (int i = k + 1; i < n; i++) {
            if ((i % NUM_THREADS) == 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_loadu_ps(B[k] + j);
                    t1 = _mm_loadu_ps(B[i] + j);
                    t2 = _mm_mul_ps(t2, t3);
                    t1 = _mm_sub_ps(t1, t2);
                    _mm_storeu_ps(B[i] + j, t1);
                }
            }
        }
        pthread_barrier_wait(&barrier);
    }
    pthread_mutex_lock(&mutex);
    QueryPerformanceCounter((LARGE_INTEGER *)&tail);
    cout << "Thread␣" << r << ":␣" << (tail - head) * 1000.0 / freq << "ms"
       << endl;
    pthread_mutex_unlock(&mutex);
    pthread_exit(0);
    return 0;
}
```

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Pthread优化结AVX的SIMD方法

```
void* LU_pthread_avx(void *parm) {
    threadParm_t *p = (threadParm_t *)parm;
    int r = p->threatId;
```

```
    long long tail;
    __m256 t1, t2, t3;
    for (int k = 0; k < n;k++) {
        for (int i = k + 1; i < n; i++) {
            if ((i % NUM_THREADS) == 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_loadu_ps(B[k] + j);
                    t1 = _mm256_loadu_ps(B[i] + j);
                    t2 = _mm256_mul_ps(t2, t3);
                    t1 = _mm256_sub_ps(t1, t2);
                    _mm256_storeu_ps(B[i] + j, t1);
                }
            }
        }
        pthread_barrier_wait(&barrier);
    }
    pthread_mutex_lock(&mutex);
    QueryPerformanceCounter((LARGE_INTEGER *)&tail);
    cout << "Thread " << r << ": " << (tail - head) * 1000.0 / freq << "ms"
        << endl;
    pthread_mutex_unlock(&mutex);
    pthread_exit(0);
    return 0;
}
```

（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-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：

对第一部分的代码进行测试，截取VTune相关数据并得到得到第一部分实验结果如下表：

测试项目\任务划分策略	朴素无优化	按列划分	按行划分
算法用时	1067.460ms	642.174ms/642.718ms	415.963ms/416.261ms
CPI	0.663	0.444	0.429
Cache命中次数	5827394150	5040627220	4880367910
L3 Cache缺失次数	440235	120985	62755
负载均衡	null	1:1	1:1
相对朴素算法的加速比	1:1	1.66:1	2.56:1
启用线程数	1	2	2
矩阵规模	1024	1024	1024

表 1: 第一部分实验结果（不同划分策略性能对比）

由第一部分实验结果可知，两种划分策略总体来说按行划分更优。猜测是访问在访问数据次数差不多的情况下按行划分策略的缓存缺失次数较少。两个种策略都能得到较好的负载均衡。因此接下来的第二部分实验将以按行划分的策略为基础结合SSE/AVX等SIMD方法对算法进行进一步的优化。

第二部分实验：

将使用多组不同的数据规模，对第二部分实验的代码进行测试。将从算法执行时间、加速比等多方面对结果进行分析比较。

对于行划分策略的Pthread结合SIMD方法的优化算法，进行以下操作。

将线程数$NUM_THREADS$调为2，得到以下结果：

数据规模\优化策略	无优化	仅使用Pthread	Pthread+SSE	Pthread+AVX
512	233.51	87.97	60.55	40.68
1024	1724.78	766.573	455.20	327.096
2048	14532.5	6002.67	3368.15	2568.87
4096	118799	50658.8	27172	21962.1

表 2: 性能测试结果(2线程)(单位:ms)

根据实验结果，求出加速比。绘制加速比统计图1如下：

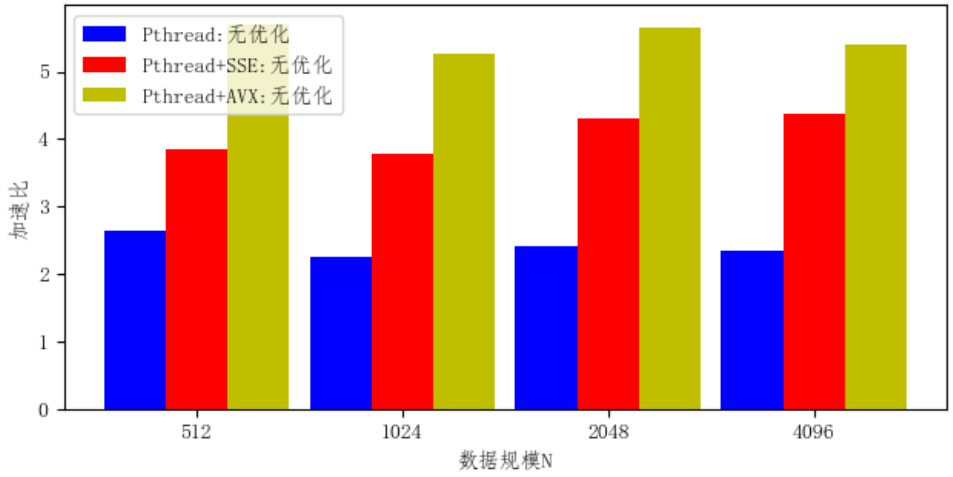

图 1: Pthread与SIMD方法结合优化算法获得的加速比(2线程)

根据得到的加速比结果可以知道，当线程数为2时，仅使用pthread优化算法大约能获得2.5倍左右的加速比；使用Pthread与SSE向量循环展开优化方法大约能获得3.5倍左右的加速比；使用Pthread与AVX向量循环展开的优化方法能获得5倍以上的加速比。

将线程数$NUM_THREADS$调为4，得到以下结果：

数据规模\优化策略	无优化	仅使用Pthread	Pthread+SSE	Pthread+AVX
512	236.51	59.71	35.64	28.98
1024	1865.37	429.69	273.77	202.72
2048	14900.1	3594.7	2135.39	1709.84
4096	117977ms	26487.3	16762.4	13541

表 3: 性能测试结果(4线程)(单位:ms)

根据实验结果，求出加速比。绘制加速比统计图2如下：

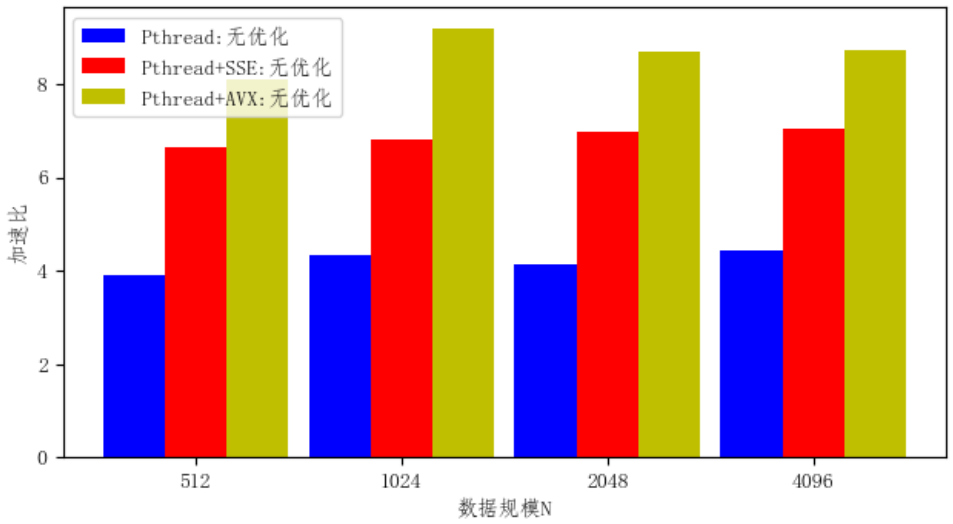

图 2: Pthread与SIMD方法结合优化算法获得的加速比(4线程)

根据得到的加速比结果可以知道，当线程数为4时，仅使用pthread优化算法大约能获得4.3倍左右的加速比；使用Pthread与SSE向量循环展开优化方法大约能获得7.0倍左右的加速比；使用Pthread与AVX向量循环展开的优化方法能获得9.0倍左右的加速比。

由于测试平台CPU为4核8线程，线程数过多系统线程分配开销过大反而影响性能，故采用2线程和4线程比较合理。将2线程与4线程做比较，探究在一定范围内线程数对性能的影响。

将线程数为2的测试结果和线程数2为4的测试结果比较，计算相应的加速比。得到加速比统计图3如下：

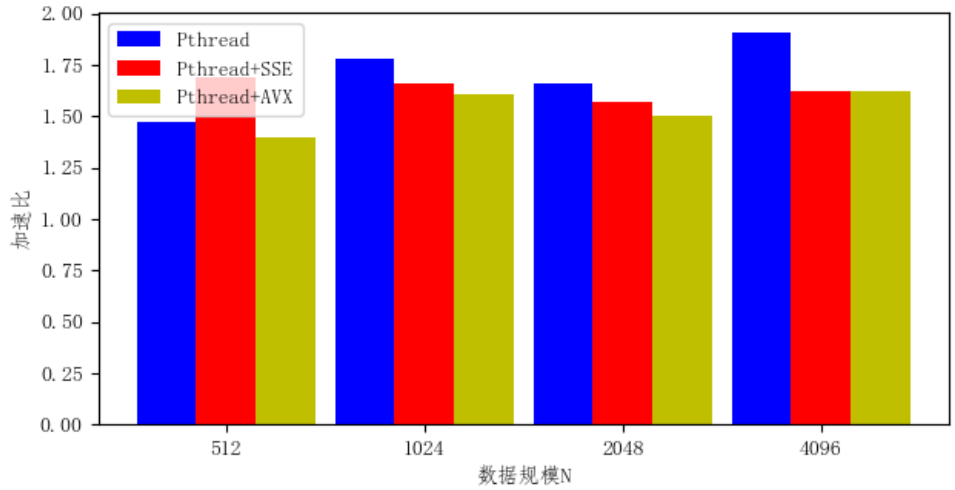

图 3: 线程数对程序性能影响结果(2线程:4线程)

根据得到的加速比结果可知，所有的加速比都在1.6倍左右。这说明线程数量对加速比影响比较恒定。本次实验条件所限，没有做更多线程数量的测试。但根据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，性能反而会下降。

二、 总结

本次实验首先使用Pthread多线程对高斯消元法进行优化，并探究不同的任务划分策略，即对外层循环（按行）进行划分和对内层循环（按列）进行划分对性能有多大的提升，经过测试发现按行进行划分性能更优。

在探究完不同划分策略对算法性能优化的影响之后，选择最优的优化策略与SIMD循环展开向量并行化结合，对算法进行进一步的优化。比较相关性能，并接着探究不同线程数对程序性能的影响，得出了一些结论。

- 在Pthread多线程任务划分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。
- 线程数对程序性能优化的影响比较恒定。在一定范围内，线程数量越多，对Pthread任务划分的优化算法能获得更大的加速比。