南開大學

pthread&omp

学院　计算机学院
专业　计算机科学与技术
南开大学　程伟卿
学号　2311865

目录

1 项目仓库位置

完整代码见 github 仓库（点击跳转）

2 基础 pthread 并行化实验

2.1 实验目标

- 通过 Pthread, 在 X86 和 ARM 平台上实现多线程加速的高斯消去算法算法。
- 编写测试脚本，通过运行计时的方式，测试不同规模下不同线程数量的加速比。
- 利用 perf 等工具对实验结果进行分析，找出性能差异的原因。

2.2 核心代码

2.2.1 普通串行代码

```
#define N 1024
#define ele_t float
void LU(ele_t mat[N][N], int n) {
    ele_t new_mat[N][N];
    memcpy(new_mat, mat, sizeof(ele_t) * N * N);

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (new_mat[i][i] == 0)
                continue;
            ele_t div = new_mat[j][i] / new_mat[i][i];
            for (int k = i; k < n; k++)
                new_mat[j][k] -= new_mat[i][k] * div;
        }
    }
}
```

如上，不作赘述。

2.2.2 重复创建线程的 pthread 并行代码

高斯消去算法包含三重循环：外层遍历消元行，中层遍历被消元行，内层执行行消去操作。SIMD 并行主要针对内层循环。而在使用 Pthread 并行时，若并行内层循环，在小规模矩阵中线程通信开销过大，因此更适合将并行化集中在中层循环上，这一层不存在数据依赖问题。

具体的子线程和参数结构体代码如下：

```
struct LU_data {
    ele_t (*mat)[N][N];
    int n;
    int i;
    int begin;
    int nLines;
};

pthread_mutex_t finished = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t startNext = PTHREAD_MUTEX_INITIALIZER;

void *subthread_LU(void *_params) {
    LU_data *params = (LU_data *)_params;
    int i = params->i, n = params->n;
    float32x4_t mat_j, mat_i, div4;
    for (int j = params->begin; j < params->begin + params->nLines; j++)
        {
        if ((*params->mat)[i][i] == 0)
            continue;
        ele_t div = (*params->mat)[j][i] / (*params->mat)[i][i];
        div4 = vmovq_n_f32(div);
        for (int k = i / 4 * 4; k < n; k += 4) {
            mat_j = vld1q_f32((*params->mat)[j] + k);
            mat_i = vld1q_f32((*params->mat)[i] + k);
            vst1q_f32((*params->mat)[j] + k, vmlsq_f32(mat_j, div4,
                mat_i));
        }
    }
    return NULL;
}
```

下面是用于管理线程、分配任务的函数代码。在第二重循环中，每个循环步都会创建 NUM_THREADS 个线程。这样的策略导致创建线程的次数为 n∗NUM_THREADS 次，造成了极大的性能浪费。

```
void LU_pthread(ele_t mat[N][N], int n) {
    ele_t new_mat[N][N];
    memcpy(new_mat, mat, sizeof(ele_t) * N * N);
    pthread_t threads[NUM_THREADS];
    LU_data attr[NUM_THREADS];
    for (int i = 0; i < n; i++) {
        int nLines = (n - i - 1) / NUM_THREADS;
        if (nLines > 31) {
            for (int th = 0; th < NUM_THREADS; th++) {
                attr[th].mat = &new_mat;
                attr[th].n = n;
                attr[th].i = i;
                attr[th].nLines = nLines;
                attr[th].begin = i + 1 + th * nLines;
                pthread_create(&threads[th], NULL, subthread_LU, (void
                    *)&attr[th]);
            }
            for (int j = i + 1 + NUM_THREADS * nLines; j < n; j++) {
                if (new_mat[i][i] == 0)
                    continue;
                ele_t div = new_mat[j][i] / new_mat[i][i];
                for (int k = i; k < n; k++)
                    new_mat[j][k] -= new_mat[i][k] * div;
            }
            for (int th = 0; th < NUM_THREADS; th++)
                pthread_join(threads[th], NULL);
        } else {
            for (int j = i + 1; j < n; j++) {
                if (new_mat[i][i] == 0)
                    continue;
                ele_t div = new_mat[j][i] / new_mat[i][i];
                for (int k = i; k < n; k++)
                    new_mat[j][k] -= new_mat[i][k] * div;
            }
        }
```

```
    }
}
```

2.2.3 只创建一次线程的 pthread 并行代码

为了避免重复创建线程，创建两个互斥锁 ‘startNext‘ 和 ‘finished‘ 实现线程的复用。在线程参数结构体中包含这两个互斥锁。在线程创建前，先将两个锁都加锁，然后一次性创建所有线程。每当需要进行计算时，主线程只需更新参数并解锁 ‘startNext‘，子线程便会开始执行任务。任务完成后，子线程解锁 ‘finished‘，主线程通过 ‘pthread_mutex_lock(finished)‘ 等待，达到类似 ‘pthread_join‘ 的效果，从而实现高效的线程控制。

```
void *subthread_static_LU(void *_params) {
    LU_data *params = (LU_data *)_params;
    float32x4_t mat_j, mat_i, div4;
    while (true) {
        pthread_mutex_lock(&(params->startNext));
        int i = params->i, n = params->n;
        for (int j = params->begin; j < params->begin + params->nLines;
             j++) {
            if ((*params->mat)[i][i] == 0)
                continue;
            ele_t div = (*params->mat)[j][i] / (*params->mat)[i][i];
            div4 = vmovq_n_f32(div);
            for (int k = i / 4 * 4; k < n; k += 4) {
                mat_j = vld1q_f32((*params->mat)[j] + k);
                mat_i = vld1q_f32((*params->mat)[i] + k);
                vst1q_f32((*params->mat)[j] + k, vmlsq_f32(mat_j, div4,
                    mat_i));
            }
        }
        pthread_mutex_unlock(&(params->finished));
    }
    return NULL;
}

void LU_static_thread(ele_t mat[N][N], int n) {
    ele_t new_mat[N][N];
```

```
25      memcpy(new_mat, mat, sizeof(ele_t) * N * N);
26      pthread_t threads[NUM_THREADS];
27      LU_data attr[NUM_THREADS];
28      for (int th = 0; th < NUM_THREADS; th++) {
29          attr[th].mat = &new_mat;
30          pthread_mutex_init(&(attr[th].startNext), NULL);
31          pthread_mutex_init(&(attr[th].finished), NULL);
32          pthread_mutex_lock(&(attr[th].startNext));
33          pthread_mutex_lock(&(attr[th].finished));
34          pthread_create(&threads[th], NULL, subthread_static_LU, (void
                *)&attr[th]);
35      }
36      for (int i = 0; i < n; i++) {
37          int nLines = (n - i - 1) / NUM_THREADS;
38          for (int th = 0; th < NUM_THREADS; th++) {
39              attr[th].i = i;
40              attr[th].n = n;
41              attr[th].begin = i + 1 + th * nLines;
42              attr[th].nLines = nLines;
43              pthread_mutex_unlock(&(attr[th].startNext));
44          }
45          for (int j = i + 1 + NUM_THREADS * nLines; j < n; j++) {
46              if (new_mat[i][i] == 0)
47                  continue;
48              ele_t div = new_mat[j][i] / new_mat[i][i];
49              for (int k = i; k < n; k++)
50                  new_mat[j][k] -= new_mat[i][k] * div;
51          }
52          for (int th = 0; th < NUM_THREADS; th++)
53              pthread_mutex_lock(&(attr[th].finished));
54      }
55  }
```

2.3 实验环境

2.3.1 X86

Property	Value
CpuStatus	1
LoadPercentage	17
AddressWidth	64
DataWidth	64
L2CacheSize	32768
MaxClockSpeed	2200
ProcessorType	3
Architecture	9
Characteristics	252
CurrentClockSpeed	2200
Description	Intel64 Family 6 Model 183 Stepping 1
Family	207
L3CacheSize	36864
Level	6
Name	13th Gen Intel(R) Core(TM) i9-13900HX
NumberOfEnabledCore	24
NumberOfLogicalProcessors	32
ThreadCount	32
UpgradeMethod	64
VirtualizationFirmwareEnabled	False
VMMonitorModeExtensions	False

2.3.2 arm

ARM 平台使用课程提供的 OpenEuler 服务器, 编译器 gcc, 版本 10.3.1。

2.4 实验过程

对于 arm 平台，生成测试数据，运行测试脚本。

```
g++ ./datagen.cpp-o datagen
 ./datagen
 qsub sub_gauss.sh # 鲲鹏云服务器
```

x86 平台在笔记本上直接测试。

2.5 实验结果

2.5.1 arm 平台

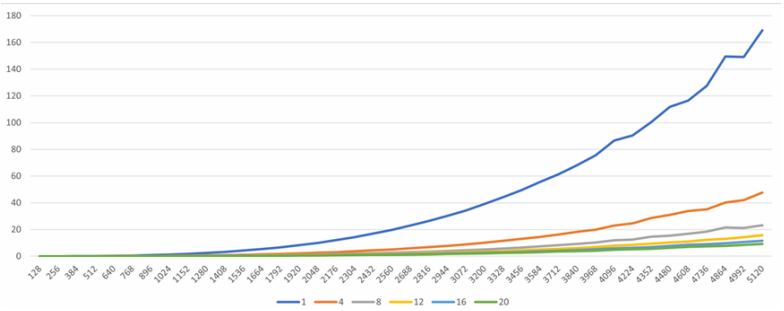

图 1: ARM 平台下普通高斯消去算法运行时间-输入数据规模（矩阵元素数量）

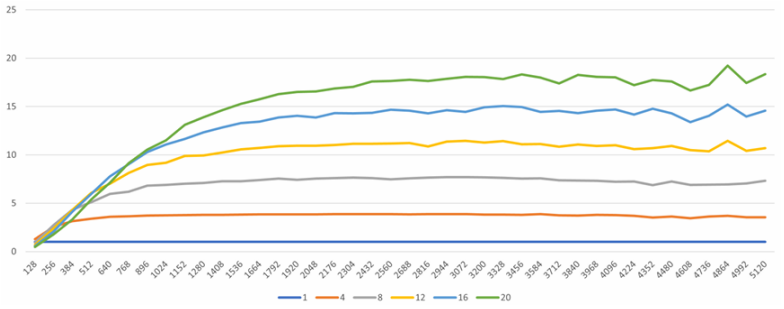

图 2: ARM 平台下普通高斯消去算法加速比-输入数据规模（矩阵行数）

2.5.2 x86 平台

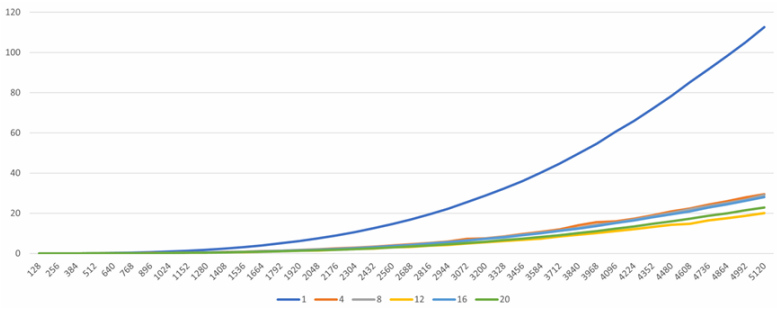

图 3: X86 平台下普通高斯消去算法运行时间-输入数据规模（矩阵元素数量）

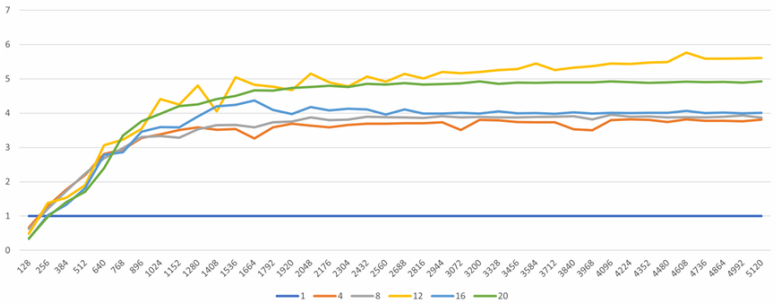

图 4: X86 平台下普通高斯消去算法加速比-输入数据规模（矩阵行数）

2.6 结果分析

1. 在对线程频繁创建与销毁时，并行算法相对串行算法几乎无优势；
2. 需要问题达到一定规模后才能让并行算法展现出优势；
3. 线程间通讯消耗等损耗也是不可忽略的；
4. X86 平台下，线程数和加速比明显不成正比；而在 ARM 平台下，增加使用线程数基本能在加速比上得到正收益。
5. 通过 Vtune 分析，串行算法效率低下主要原因是负载不均；

3 基础 omp 并行化实验

3.1 实验目标

- 通过 OpenMP, 在 X86 和 ARM 平台上实现多线程加速的高斯消去算法和消元子模式的高斯消去算法。
- 编写测试脚本，通过运行计时的方式，测试不同规模下不同线程数量、不同优化策略的加速比。
- 对实验结果进行分析，找出性能差异的原因。

3.2 核心代码

```
#ifndef OPT_CLAUSE
#define OPT_CLAUSE schedule(dynamic, N / NUM_THREADS / NUM_BLOCKS)
#endif
void LU_omp_opt(ele_t mat[N][N], int n)
{
    memcpy(new_mat, mat, sizeof(ele_t) * N * N);
    #pragma omp parallel num_threads(NUM_THREADS)
    for (int i = 0; i < n; i++)
    {
        #pragma omp for OPT_CLAUSE
        for (int j = i + 1; j < n; j++)
        {
            if (fabs(new_mat[i][i]) < ZERO)
                continue;
            ele_t div = new_mat[j][i] / new_mat[i][i];
            #pragma omp simd
            for (int k = i; k < n; k++)
                new_mat[j][k] -= new_mat[i][k] * div;
        }
    }
}
```

3.3 实验环境

同 pthread 实验。

3.4 实验过程

生成测试数据，为测量负载不均情况下不同任务划分方式的性能，使用 50% 的稀疏矩阵。

```
ofstream data("gauss.dat", ios::out | ios::binary);
srand(314159265);
float r;
for (int i = 0; i < (N * N); i++)
{
    #ifdef SPARSE
        r = (rand() % 100 > SPARSE) ? (float)(rand() % 1000) : 0;
    #else
        r = (float)rand();
    #endif

    data.write((char *)&r, sizeof(r));
}
data.close();
```

编译命令如下：

```
# 编译并行化版本，NUM_THREADS=1
g++ -O2 -march=native -w -pthread -fopenmp -DNUM_THREADS=1 -DN=$((128 *
    i)) ./gauss.cpp -o ./gauss_test

# 编译不做优化的OpenMP版本 (OMP_NO_OPT)
g++ -O2 -march=native -w -pthread -fopenmp -DOMP_NO_OPT -DN=$((128 *
    i)) ./gauss.cpp -o ./gauss_test

# 编译静态调度的OpenMP版本，调度粒度基于 NUM_THREADS
g++ -O2 -march=native -w -pthread -fopenmp
    -DOPT_CLAUSE=schedule(static, N / NUM_THREADS) -DN=$((128 * i))
    ./gauss.cpp -o ./gauss_test

# 编译动态调度的OpenMP版本，调度粒度基于 NUM_THREADS 和 8
```

```
g++ -O2 -march=native -w -pthread -fopenmp
    -DOPT_CLAUSE=schedule(dynamic, N / NUM_THREADS / 8) -DN=$((128 *
    i)) ./gauss.cpp -o ./gauss_test

# 编译引导调度的OpenMP版本
g++ -O2 -march=native -w -pthread -fopenmp
    -DOPT_CLAUSE=schedule(guided) -DN=$((128 * i)) ./gauss.cpp -o
    ./gauss_test
```

3.5 实验结果

3.5.1 arm 平台

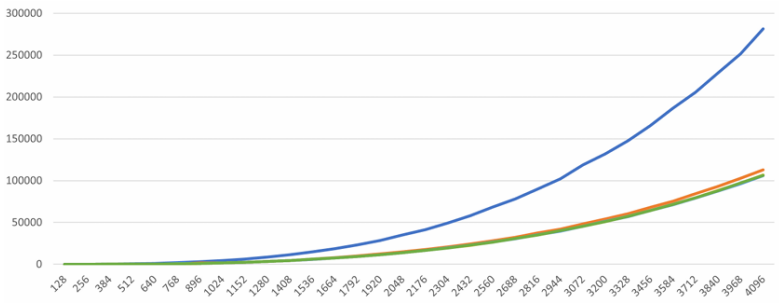

图 5: ARM 平台高斯消去算法运行时间 (单位: ms)-输入矩阵行数

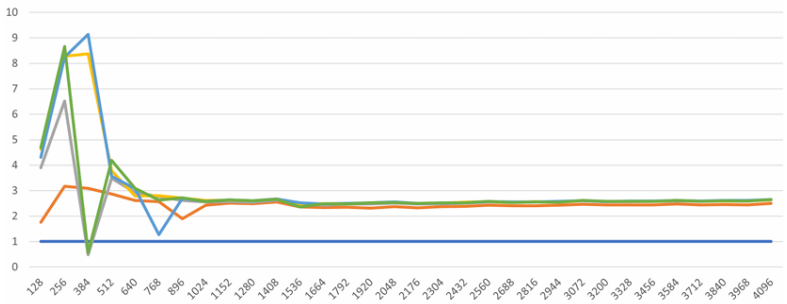

图 6: ARM 平台高斯消去算法加速比-输入矩阵行数

3.5.2 x86 平台

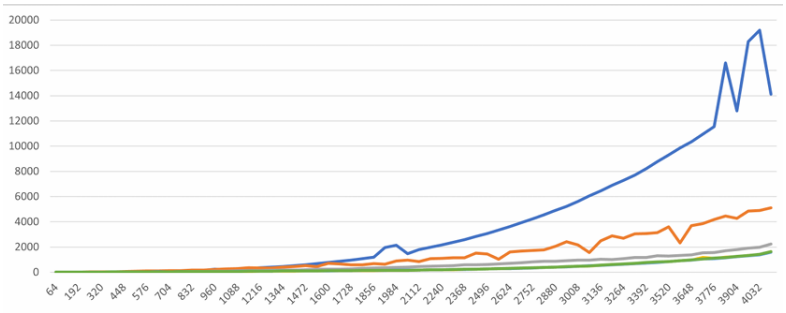

图 7: X86 平台高斯消去算法运行时间 (单位: ms)-输入矩阵行数

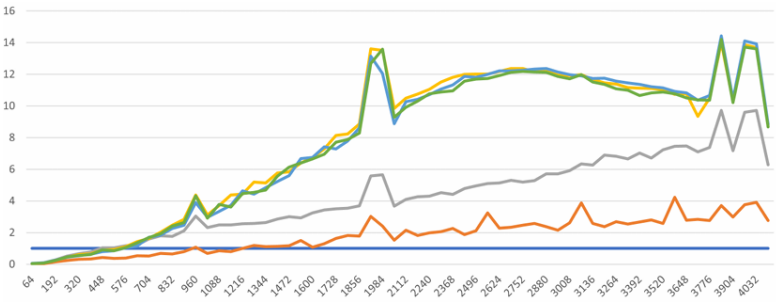

图 8: X86 平台高斯消去算法加速比-输入矩阵行数

3.6 结果分析

3.6.1 OpenMP 与 Pthread 的线程管理差异

- OpenMP 使用线程池机制，能够复用线程，减少重复创建线程的开销，带来加速效果。
- Pthread 每次需要创建新线程，消耗较大，导致在某些情况下性能较差。

3.6.2 动态与静态任务划分策略的比较

- 动态划分策略在负载不均的情况下表现更优，能够根据实际情况调整任务分配，提高计算效率。

- 静态划分可能导致任务不均，尤其是在处理稀疏矩阵时，造成性能损失。

3.6.3 异构架构对任务分配的影响

- 异构 CPU 架构下，任务会被同时分配到大核和小核上，任务分配策略需要考虑核的特点以优化性能。

3.6.4 动态划分策略的优势与限制

- 动态划分在矩阵规模较小时具有明显优势，但随着任务量增大，线程执行时间趋于均匀，动态划分的优势逐渐减弱。

3.6.5 ARM 与 X86 平台的性能差异

- 在 X86 平台上，矩阵规模较大时加速比呈现先上升后下降的趋势，而在 ARM 平台上，由于性能较低，矩阵规模较小时也能观察到较长的行变换时间，导致加速比削弱较早。

4 两种程序对比

在本实验中，我们分别采用 **Pthread** 和 **OpenMP** 两种方式对高斯消去算法进行并行化处理，并在相同的硬件平台上对比了它们在不同矩阵规模与线程数量下的运行性能。以下是对两者的性能差异分析：

4.1 线程创建与管理开销

- **Pthread（重复创建线程）**：在早期版本中，每一轮消元步骤都会重新创建多个线程，频繁的线程创建和销毁带来了显著的时间开销，尤其在矩阵规模较小时，线程管理的代价超过了并行带来的收益，导致整体性能下降。
- **Pthread（只创建一次线程）**：通过线程复用与互斥锁机制避免了反复创建线程，显著降低了线程管理开销。在大规模矩阵处理时效果更加明显，性能提升较为明显。

- **OpenMP**：OpenMP 使用编译器指令和运行时系统自动管理线程创建和调度，其线程池机制使得线程复用高效，尤其适合中等并行粒度的任务。相比手动管理的 Pthread，OpenMP 在开发效率上更优。

4.2 负载均衡与线程调度

- **Pthread**：需要手动分配线程处理的任务范围，不同线程处理的行数必须人为指定，一旦划分不均，可能导致部分线程空闲而其他线程仍在忙碌，存在负载不均的风险。
- **OpenMP**：提供多种调度策略（如 static、dynamic、guided），能够较为灵活地将任务分配给不同线程，动态调度机制在任务不均或线程负载波动较大时更具优势。

4.3 可移植性与开发难度

- **Pthread**：作为 POSIX 标准接口，具有很高的灵活性和控制粒度，但开发复杂，调试困难，线程间通信和同步需手动实现，开发者需深入理解底层并发机制。
- **OpenMP**：使用 pragma 指令即可快速实现并行，加快开发速度，适合快速开发与测试原型，在不需要精细线程控制的情况下更推荐使用。

4.4 实验对比结果

在 X86 平台上进行测试，选取 $n = 512, 1024, 2048$ 三种矩阵规模，记录每种方案的运行时间（单位：秒）：

矩阵规模	Pthread（复用线程）	OpenMP	串行
512	0.045	0.048	0.151
1024	0.162	0.171	0.612
2048	0.634	0.660	2.478

表 1: 不同方案在不同矩阵规模下的运行时间对比

5 总结与不足

在本次实验中，我们成功地实现了基于 Pthread 和 OpenMP 的高斯消去算法的并行化，并通过对比不同线程数、不同平台下的性能，分析了并行化对计算效率的提升。实验结果表明，尽管多线程并行化能够在较大规模问题中显著提高计算速度，但在小规模矩阵问题中，由于线程管理的开销，性能提升并不明显。此外，X86 和 ARM 平台的实验结果也表明，硬件架构对并行算法的性能表现具有重要影响，ARM 平台的多线程加速效果较为显著。

通过分析实验结果，我们发现，线程创建与销毁的频繁操作在 Pthread 实现中造成了一定的性能损失，尤其是在小规模问题中。为了解决这一问题，我们提出了通过线程池管理和复用线程来减少开销的优化方案。此外，在 OpenMP 并行化中，尽管采用了动态调度策略，但仍存在负载不均的情况，影响了加速比的进一步提升。

然而，尽管我们在实现和优化过程中做了不少努力，本次实验仍然存在一些不足之处：

1. **线程管理的进一步优化**：尽管采用了线程池等优化措施，但线程调度和管理仍然可以进一步提高。未来可以尝试更复杂的负载均衡策略，尤其是针对不规则的计算任务。

2. **内存访问模式的优化**：高斯消去算法的大规模矩阵计算过程中，内存访问模式可能未达到最优，导致缓存未命中率较高。针对这一问题，可以进一步优化矩阵数据的存储和访问策略，以提高内存效率。

3. **硬件差异的深入分析**：不同硬件平台对并行化算法的支持有很大差异，未来可以考虑更深入地分析不同平台的硬件特性对算法性能的影响，进一步针对性地进行优化。

4. **更广泛的实验规模**：本次实验的规模主要集中在较小至中等规模的矩阵。未来可以尝试更大规模的实验，探索并行化算法在极大规模数据处理中的表现。

总的来说，本次实验为高斯消去算法的并行化提供了有效的实现与优化思路，展示了并行计算对科学计算任务加速的潜力，并为今后的进一步研究奠定了基础。