南開大學
Nankai University

计算机学院

并行程序设计第 **5** 次作业

高斯消去法的 OpenMP 并行化

姓名：丁屹

学号：2013280

专业：计算机科学与技术

2022 年 5 月 21 日

目录

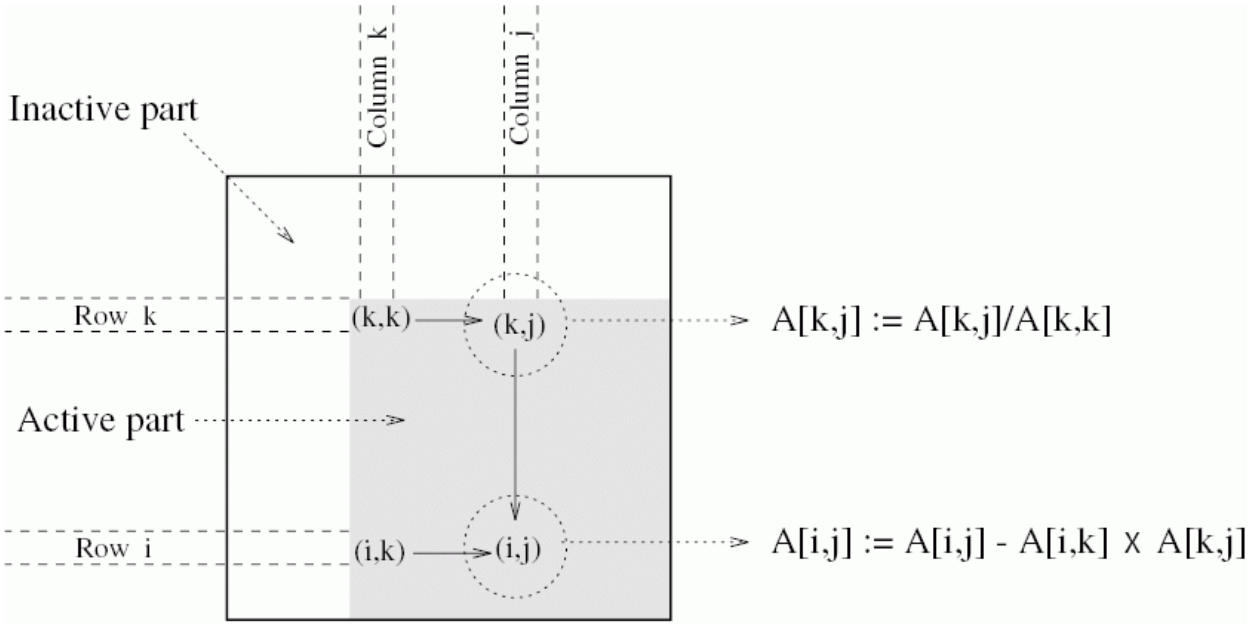

图 1.1: 高斯消去法示意图

1 问题描述

高斯消去的计算模式如图 1.1 所示，在第 k 步时，对第 k 行从 (k,k) 开始进行除法操作，并且将后续的 $k+1$ 至 N 行进行减去第 k 行的操作，串行算法如下面伪代码所示。

Algorithm 1 普通高斯消元算法伪代码

```
1:  function LU
2:      for k := 0 to n do
3:          for j := k + 1 to n do
4:              A[k, j] := A[k, j]/A[k, k]
5:          end for
6:          A[k, k] := 1.0
7:          for i := k + 1 to n do
8:              for j := k + 1 to n do
9:                  A[i, j] := A[i, j] − A[i, k] ∗ A[k, j]
10:             end for
11:             A[i, k] := 0
12:         end for
13:     end for
14: end function
```

观察高斯消去算法，注意到伪代码第 4，5 行第一个内嵌循环中的 $A[k,j] := A[k,j]/A[k,k]$ 以及伪代码第 8 9 10 行双层 for 循环中的 $A[i,j] := A[i,j]-A[i,k]\times A[k,j]$ 都是可以进行向量化的循环。可以通过 OpenMP 以及 SIMD 扩展指令对这两步进行并行优化。

2 OpenMP 算法设计

源码链接：https://github.com/ArcanusNEO/Parallel-Programming/tree/master/5

2.1 测试用例的确定

由于测试数据集较大，不便于各个平台同步，所以采用固定随机数种子为 12345687 的 mt19937 随机数生成器。经过实验发现不同规模下，所有元素独立生成，限制大小在 $[0, 100]$，能够生成可以被正确消元的矩阵。

代码如下：

测试数据集生成器

```
uniform_real_distribution<float> dist(0, 100);
mt19937 mt(12345687);
int n;
istringstream iss(argv[1]);
iss >> n;
cout << n << endl;
for (int i = 1; i <= n; ++i)
  for (int j = 1; j <= n; ++j) cout << dist(mt) << " \n"[j == n];
```

2.2 实验环境和相关配置

实验在华为鲲鹏 ARM 集群平台和本地 Arch Linux x86_64 平台完成；

华为鲲鹏 ARM 集群平台使用华为毕昇 clang++ 编译器，本地 Arch Linux x86_64 平台使用 GNU GCC 编译器；

使用 cmake 构建项目，编译开关如下：

```
set(CMAKE_CXX_FLAGS_RELEASE "-O3")
set(THREADS_PREFER_PTHREAD_FLAG ON)
find_package(OpenMP REQUIRED)
```

实验测试了 4、8、16 线程并行的运行数据。

2.3 算法设计

2.3.1 默认平凡算法

使用一维数组模拟矩阵，避免改变矩阵大小时第二维不方便调整、必须设成最大值的问题，可以减少 cache 失效；

使用 $\#define\ matrix(i, j)\ arr[(i) * n + (j)]$ 宏，增强可读性；

平凡算法

```
#define matrix(i, j) arr[(i) * n + (j)]
void func(int& ans, float arr[], int n) {
```

Scale	Reperat times	x86 ordinary (s)	arm ordinary (s)
8 × 8	100	0.000001330460	0.000000525400
16 × 16	50	0.000001706920	0.000001666000
32 × 32	50	0.000003640080	0.000007127000
64 × 64	20	0.000015253300	0.000037566500
128 × 128	15	0.000098880800	0.000231574000
256 × 256	10	0.000716408500	0.001820356000
512 × 512	10	0.006722607300	0.014974396000
1024 × 1024	5	0.064893815400	0.135511226000
2048 × 2048	3	1.400074583333	1.101775523333
4096 × 4096	1	10.705585484000	13.088073440000

表 1: 所有平台平凡算法结果对比

```
  for (int k = 0; k < n; ++k) {
    for (int j = k + 1; j < n; ++j) matrix(k, j) = matrix(k, j) / matrix(k, k);
    matrix(k, k) = 1.0;
    for (int i = k + 1; i < n; ++i) {
      for (int j = k + 1; j < n; ++j)
        matrix(i, j) = matrix(i, j) - matrix(i, k) * matrix(k, j);
      matrix(i, k) = 0;
    }
  }
#undef matrix
}
```

2.3.2 所有平台下只使用 OpenMP 4、8、16 线程并行化加速

OpenMP 并行化加速

```
  #define NUM_THREADS 8

  void func(int& ans, float arr[], int n) {
  #define matrix(i, j) arr[(i) *n + (j)]
    int   i, j, k;
    float tmp;
  #pragma omp parallel num_threads(NUM_THREADS), private(i, j, k, tmp)
    for (k = 0; k < n; ++k) {
      tmp = matrix(k, k);
  #pragma omp for
      for (j = k + 1; j < n; ++j) matrix(k, j) = matrix(k, j) / tmp;
      matrix(k, k) = 1.0;
  #pragma omp for
      for (i = k + 1; i < n; ++i) {
```

Scale	Reperat times	x86 OpenMP 4 threads (s)	x86 OpenMP 8 threads (s)	x86 OpenMP 16 threads (s)	arm OpenMP 4 threads (s)	arm OpenMP 8 threads (s)	arm OpenMP 16 threads (s)
8 × 8	100	0.000007841100	0.000011223470	0.000016886970	0.000092409600	0.000126543900	0.000168758200
16 × 16	50	0.000016620760	0.000019207720	0.000048819040	0.000179391200	0.000235206200	0.000287643600
32 × 32	50	0.000029769140	0.000038458880	0.000097313150	0.000357201000	0.000496609800	0.000712039600
64 × 64	20	0.000063419300	0.000077604000	0.000145570260	0.000716369500	0.000993004500	0.001336811000
128 × 128	15	0.000164951200	0.000178723933	0.000232990467	0.001481558000	0.001898954667	0.002348973333
256 × 256	10	0.000460212000	0.000523160100	0.001742868100	0.003256248000	0.004177293000	0.005598914000
512 × 512	10	0.001627336600	0.001560966700	0.051130604200	0.009262748000	0.009874281000	0.011944158000
1024 × 1024	5	0.012059842800	0.012384870200	0.743364184400	0.045375618000	0.031736468000	0.028419450000
2048 × 2048	3	1.195753347667	0.823777882667	2.923949854667	0.334706430000	0.275254946667	0.112933620000
4096 × 4096	1	11.418539925000	9.865224839000	11.725731761000	6.097139010000	3.971935890000	2.177672700000

表 2: 所有平台 OpenMP only 结果对比

```
15          tmp = matrix(i, k);
16          for (j = k + 1; j < n; ++j)
17            matrix(i, j) = matrix(i, j) - tmp * matrix(k, j);
18          matrix(i, k) = 0;
19        }
20      }
21    #undef matrix
22    }
```

其中修改 NUM_THREADS 宏定义可以指定不同的并行线程数。

测试了 4、8、16 的数据。

2.3.3 使用 OpenMP 及 SIMD 在 x86 平台 4、8 线程并行化加速

x86 OpenMP + AVX 并行化加速

```
1   #define NUM_THREADS 8
2
3   void func(int& ans, float arr[], int n) {
4   #define matrix(i, j)  arr[(i) *n + (j)]
5   #define pmatrix(i, j) (arr + (i * n + j))
6     int    i, j, k;
7     float  tmp;
8     __m256 vaik, vakj, vaij, vx;
9   #pragma omp parallel num_threads(NUM_THREADS), \
10    private(i, j, k, tmp, vaik, vakj, vaij, vx)
11    for (k = 0; k < n; ++k) {
12      tmp = matrix(k, k);
13  #pragma omp for
14      for (j = k + 1; j < n; ++j) matrix(k, j) = matrix(k, j) / tmp;
15      matrix(k, k) = 1.0;
16  #pragma omp for
17      for (i = k + 1; i < n; ++i) {
18        vaik =
19          _mm256_set_ps(matrix(i, k), matrix(i, k), matrix(i, k), matrix(i, k),
20                        matrix(i, k), matrix(i, k), matrix(i, k), matrix(i, k));
21        for (j = k + 1; j + 8 <= n; j += 8) {
22          vakj = _mm256_loadu_ps(pmatrix(k, j));
23          vaij = _mm256_loadu_ps(pmatrix(i, j));
24          vx   = _mm256_mul_ps(vakj, vaik);
```

Scale	Reperat times	x86 OpenMP 4 threads AVX (s)	x86 OpenMP 8 threads AVX (s)
8 × 8	100	0.000009138760	0.000011101290
16 × 16	50	0.000015564840	0.000018825000
32 × 32	50	0.000030945160	0.000039081760
64 × 64	20	0.000067078900	0.000079877600
128 × 128	15	0.000156863733	0.000172293600
256 × 256	10	0.000542010200	0.000501830600
512 × 512	10	0.001857792100	0.001428114600
1024 × 1024	5	0.015659546200	0.011327669000
2048 × 2048	3	1.033535896333	0.737357611000
4096 × 4096	1	10.552043834000	10.662585467000

表 3: x86 平台 OpenMP + AVX 结果对比

```
25            vaij = _mm256_sub_ps(vaij, vx);
26            _mm256_storeu_ps(pmatrix(i, j), vaij);
27          }
28          tmp = matrix(i, k);
29          for (; j < n; ++j) matrix(i, j) = matrix(i, j) - tmp * matrix(k, j);
30          matrix(i, k) = 0;
31        }
32      }
33  #undef matrix
34  }
```

其中修改 NUM_THREADS 宏定义可以指定不同的并行线程数。

测试了 4、8 的数据。

2.3.4 使用 OpenMP 及 SIMD 在 arm 平台 4、8 线程并行化加速

arm OpenMP + NEON 并行化加速

```
1   #define NUM_THREADS 8
2
3   void func(int& ans, float arr[], int n) {
4   #define matrix(i, j)  arr[(i) *n + (j)]
5   #define pmatrix(i, j) (arr + (i * n + j))
6     int         i, j, k;
7     float       tmp;
8     float32x4_t vaik, vakj, vaij, vx;
9   #pragma omp parallel num_threads(NUM_THREADS), \
10    private(i, j, k, tmp, vaik, vakj, vaij, vx)
11    for (k = 0; k < n; ++k) {
12      tmp = matrix(k, k);
13  #pragma omp for
14      for (j = k + 1; j < n; ++j) matrix(k, j) = matrix(k, j) / tmp;
15      matrix(k, k) = 1.0;
16  #pragma omp for
17      for (i = k + 1; i < n; ++i) {
```

Scale	Reperat times	arm OpenMP 4 threads NEON (s)	arm OpenMP 8 threads NEON (s)
8 × 8	100	0.000096044400	0.000150384800
16 × 16	50	0.000188844800	0.000279533400
32 × 32	50	0.000385884200	0.000604768200
64 × 64	20	0.000774845500	0.001232058000
128 × 128	15	0.001617558000	0.002336769333
256 × 256	10	0.003750217000	0.005148563000
512 × 512	10	0.011932396000	0.012737347000
1024 × 1024	5	0.062839944000	0.043704944000
2048 × 2048	3	0.504436153333	0.303472556667
4096 × 4096	1	7.025737910000	4.243570140000

表 4: arm 平台 OpenMP + NEON 结果对比

Scale	Reperat times	x86 ordinary (s)	x86 OpenMP 4 threads (s)	x86 OpenMP 8 threads (s)	x86 OpenMP 16 threads (s)	x86 OpenMP 4 threads AVX (s)	x86 OpenMP 8 threads AVX (s)
8 × 8	100	0.00000133046	0.0000078411	0.00001122347	0.00001688697	0.00000913876	0.00001110129
16 × 16	50	0.00000170692	0.00001662076	0.00001920772	0.00004881904	0.00001556484	0.000018825
32 × 32	50	0.00000364008	0.00002976914	0.00003845888	0.00009731315	0.00003094516	0.00003908176
64 × 64	20	0.0000152533	0.0000634193	0.000077604	0.00014557026	0.0000670789	0.0000798776
128 × 128	15	0.0000988808	0.0001649512	0.000178723933	0.000232990467	0.000156863733	0.0001722936
256 × 256	10	0.0007164085	0.000460212	0.0005231601	0.0017428681	0.0005420102	0.0005018306
512 × 512	10	0.0067226073	0.0016273366	0.0015609667	0.0511306042	0.0018577921	0.0014281146
1024 × 1024	5	0.0648938154	0.0120598428	0.0123848702	0.7433641844	0.0156595462	0.011327669
2048 × 2048	3	1.400074583333	1.195753347667	0.823777882667	2.923949854667	1.033535896333	0.737357611
4096 × 4096	1	10.705585484	11.418539925	9.865224839	11.725731761	10.552043834	10.662585467

表 5: x86 平台所有结果对比

```
18          vaik = vdupq_n_f32(matrix(i, k));
19          for (j = k + 1; j + 4 <= n; j += 4) {
20            vakj = vld1q_f32(pmatrix(k, j));
21            vaij = vld1q_f32(pmatrix(i, j));
22            vx   = vmulq_f32(vakj, vaik);
23            vaij = vsubq_f32(vaij, vx);
24            vst1q_f32(pmatrix(i, j), vaij);
25          }
26          tmp = matrix(i, k);
27          for (; j < n; ++j) matrix(i, j) = matrix(i, j) - tmp * matrix(k, j);
28          matrix(i, k) = 0;
29        }
30      }
31    #undef matrix
32  }
```

其中修改 NUM_THREADS 宏定义可以指定不同的并行线程数。

测试了 4、8 的数据。

3 实验结果分析

对比表格 1、2、3、4、5、6 可以发现, 对于 arm 平台,附加的 SIMD 加速效果不好,但是 OpenMP 加速效果很好: 最快的 4 线程对比平凡算法最大数据加速比达到了 $13.08807344 \div 6.09713901 = 2.146592587$, 最快的 8 线程对比平凡算法最大数据加速比达到了 $13.08807344 \div 3.97193589 = 3.295137123$, 最快的 16 线程对比平凡算法最大数据加速比达到了 $13.08807344 \div 2.1776727 = 6.010119611$。

对于 x86 平台，加速效果与数据规模关系很大，在小数据量和大数据量下平凡算法与优化算法速

Scale	Reperat times	arm ordinary (s)	arm OpenMP 4 threads (s)	arm OpenMP 8 threads (s)	arm OpenMP 16 threads (s)	arm OpenMP 4 threads NEON (s)	arm OpenMP 8 threads NEON (s)
8 × 8	100	0.0000005254	0.0000924096	0.0001265439	0.0001687582	0.0000960444	0.0001503848
16 × 16	50	0.000001666	0.0001793912	0.0002352062	0.0002876436	0.0001888448	0.0002795334
32 × 32	50	0.000007127	0.000357201	0.0004966098	0.0007120396	0.0003858842	0.0006047682
64 × 64	20	0.0000375665	0.0007163695	0.0009930045	0.001336811	0.0007748455	0.001232058
128 × 128	15	0.000231574	0.001481558	0.001898954667	0.002348973333	0.001617558	0.002336769333
256 × 256	10	0.001820356	0.003256248	0.004177293	0.005598914	0.003750217	0.005148563
512 × 512	10	0.014974396	0.009262748	0.009874281	0.011944158	0.011932396	0.012737347
1024 × 1024	5	0.135511226	0.045375618	0.031736468	0.02841945	0.062839944	0.043704944
2048 × 2048	3	1.101775523333	0.33470643	0.275254946667	0.11293362	0.504436153333	0.303472556667
4096 × 4096	1	13.08807344	6.09713901	3.97193589	2.1776727	7.02573791	4.24357014

表 6: arm 平台所有结果对比

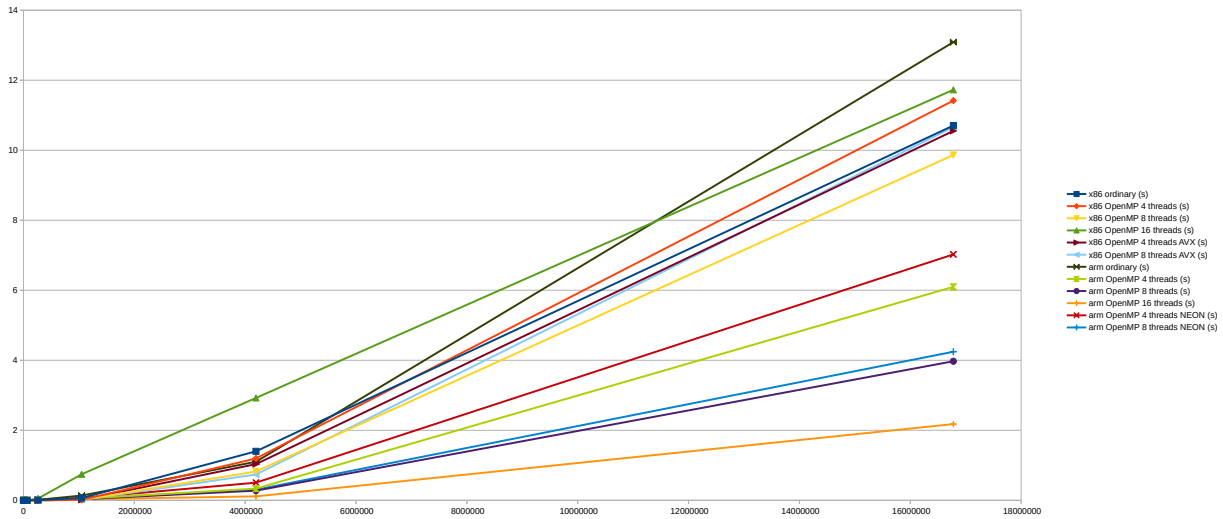

图 3.2: 所有平台所有结果对比折线图

度差异不大，在中等数据量下加速比有时能超越核心数的增加；值得注意的是，x86 平台的 16 线程加速效果很差，原因可能与处理器架构有关：使用了 AMD Ryzen 4800HS 的 CPU，8C16T，使用 1 个 CCD 2 个 CCX，两个 CCX 间不共享三级缓存，导致 16 线程加速带来过多的缓存失效。

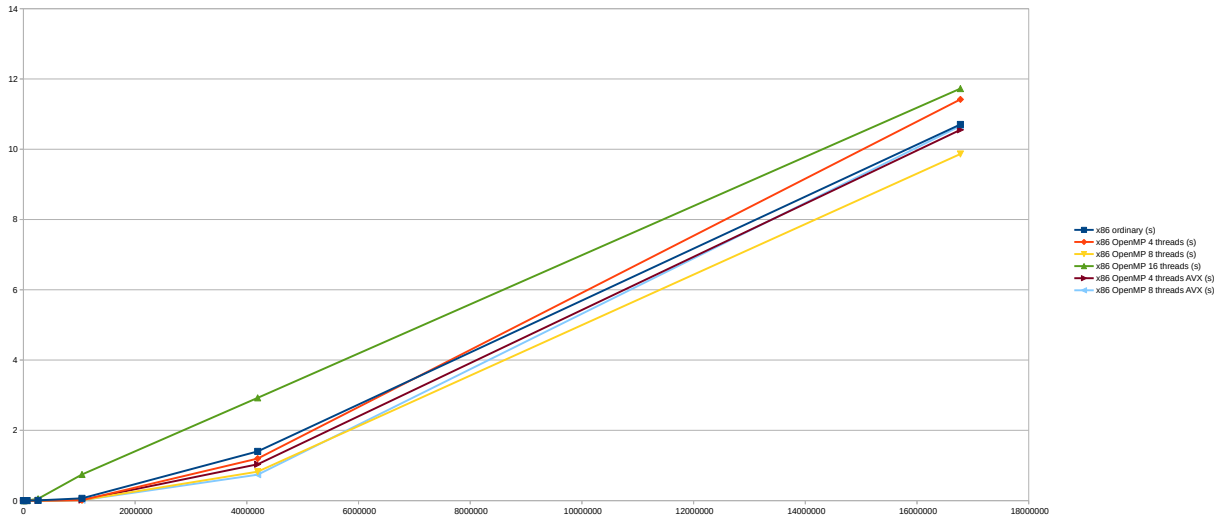

图 3.3: x86 平台所有结果对比折线图

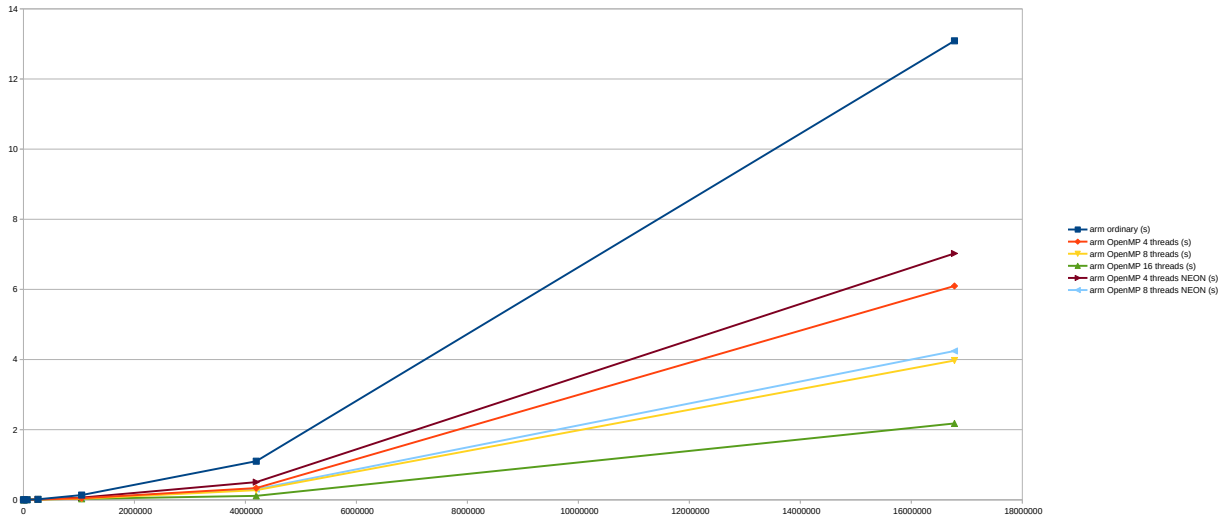

图 3.4: arm 平台所有结果对比折线图