



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

并行程序设计

Pthread 和 OpenMP 并行化实验——基于高斯消去

刘芳宜

年级：2022 级

专业：计算机科学与技术

指导教师：王刚

2024 年 5 月 15 日

摘要

本实验旨在通过在 ARM 或 x86 平台上使用 pthread 和 OpenMP 并行化算法, 探讨高斯消去计算在不同平台上的性能表现。实验包括基本要求和进阶要求两个部分: 首先设计并实现适合的任务分配算法, 分析其性能。然后将 SIMD (Neon、SSE/AVX/AVX-512) 算法与基础 pthread 和 OpenMP 并行化结合, 编程实现并在不同问题规模、不同线程数下进行算法性能测试, 比较串行和并行算法的性能, 讨论算法/编程策略对性能的影响, 以及 Pthread 程序和 OpenMP 程序的性能差异。进一步探索特殊的高斯消去计算的 pthread 和 OpenMP 并行化, 进行普通高斯消去或/和特殊高斯消去在不同平台 (x86 或 ARM) 上的并行化实验。讨论多线程并行化的不同算法策略, 如矩阵水平划分、垂直划分等不同任务划分方法, 不同算法策略下的一致性保证、线程管理代价优化等, 并进行 profiling 和体系结构相关优化, 如 cache 优化。另外, 探讨 OpenMP 卸载到加速器设备等内容。

关键字: Parallel, Pthread, OpenMP, ARM, X86, 高斯消去

目录

一、 概述	1
(一) 实验环境	1
(二) 编程实现设计思路	1
1. pthread 高斯消去任务分配算法	1
2. OpenMP 的并行算法高斯消去任务分配算法	2
(三) 算法性能分析	2
二、 SIMD 算法	3
(一) Neon	3
1. pthread	3
2. openMP	4
3. 性能对比	5
(二) AVX	6
1. pthread	6
2. openMP	7
3. 性能对比	8
(三) 性能分析	9
三、 arm 平台实现及对比	9
(一) 性能对比	9
四、 特殊高斯	10
(一) 特殊的高斯消去计算的 pthread 和 OpenMP 并行化	10
(二) 不同平台 (x86 ,ARM) 不同算法对比	11
(三) 多线程并行化的不同算法策略	11
(四) cache 优化	12
(五) perf 剖析线程性能分析	13
五、 代码库链接	13

一、概述

(一) 实验环境

x86,arm, 本地 pc 虚拟机 Ubuntu, 鲲鹏服务器

(二) 编程实现设计思路

1. pthread 高斯消去任务分配算法

任务分配：将矩阵的行分配给不同的线程处理，确保不同线程处理的行没有交集，以避免数据竞争。**同步机制：**在每一步消元后，使用互斥锁确保矩阵的状态是正确的，避免线程之间的竞争条件。**线程间通信：**使用共享内存或消息传递等方式实现线程间的数据交换，以便在每一步消元后更新矩阵的状态。**性能优化：**考虑使用 SIMD 指令集优化内层循环的计算，提高计算效率。**终止条件：**当矩阵达到上三角形形式时，算法终止。使用条件变量或其他方式实现线程的终止条件判断。

Algorithm 1 Gaussian Elimination using pthreads

Input: 增广矩阵 A , 大小为 $N \times (N + 1)$

Output: 消元后的矩阵 A

```

1: function PTHREAD_GAUSSIAN_ELIMINATION( $A$ )
2:   初始化矩阵  $A$ , 大小为  $N \times (N + 1)$ 
3:   for  $i \leftarrow 0$  to  $N - 1$  do
4:     for  $j \leftarrow 0$  to  $N$  do
5:        $A[i][j] \leftarrow$  随机值
6:     end for
7:   end for
8:   初始化屏障变量  $barrier$ , 线程数为  $NUM\_THREADS$ 
9:   记录开始时间  $start\_time$ 
10:  for  $thread\_id \leftarrow 0$  to  $NUM\_THREADS - 1$  do
11:    创建线程, 执行  $eliminate$  函数
12:  end for
13:  for  $thread\_id \leftarrow 0$  to  $NUM\_THREADS - 1$  do
14:    等待所有线程完成
15:  end for
16:  记录结束时间  $end\_time$ 
17:  输出总时间  $end\_time - start\_time$ 
18:  销毁屏障变量  $barrier$ 
19: end function
20: function ELIMINATE( $thread\_id$ )
21:    $start \leftarrow thread\_id \times (N / NUM\_THREADS)$ 
22:    $end \leftarrow (thread\_id + 1) \times (N / NUM\_THREADS)$ 
23:   for  $k \leftarrow 0$  to  $N - 2$  do
24:     for  $i \leftarrow start + 1$  to  $end - 1$  do
25:        $factor \leftarrow A[i][k] / A[k][k]$ 
26:       for  $j \leftarrow k$  to  $N$  do
27:          $A[i][j] \leftarrow A[i][j] - factor \times A[k][j]$ 

```

```

28:         end for
29:     end for
30:     等待其他线程完成当前列的消元
31:     pthread_barrier_wait(barrier)
32: end for
33: return
34: end function

```

2. OpenMP 的并行算法高斯消去任务分配算法

任务分配: 使用 OpenMP 的并行循环指令 `#pragma omp parallel for` 将循环迭代任务分配给多个线程执行。在高斯消去算法中, 可以将每一步的消元操作分配给不同的线程。**同步机制:** OpenMP 默认提供隐式的同步机制, 不需要显式地编写同步代码。但在一些需要手动管理同步的情况下, 可以使用 `#pragma omp barrier` 实现障碍同步。**数据共享** OpenMP 默认情况下, 循环迭代变量是共享的, 但在高斯消去算法中, 每个线程需要处理不同的行, 因此需要手动处理数据共享。**性能优化:** 可以使用 OpenMP 的 SIMD 指令优化内层循环的计算, 提高计算效率。**终止条件:** 当矩阵达到上三角形式时, 算法终止。可以使用 `#pragma omp cancel for` 指令来取消循环执行, 以实现终止条件。

Algorithm 2 Gaussian Elimination using OpenMP

Input: 增广矩阵 A , 大小为 $N \times (N + 1)$

Output: 消元后的矩阵 A

```

1: function OMP_GAUSSIAN_ELIMINATION(A)
2:     初始化矩阵  $A$ , 大小为  $N \times (N + 1)$ 
3:     for  $i \leftarrow 0$  to  $N - 1$  do
4:         for  $j \leftarrow 0$  to  $N$  do
5:              $A[i][j] \leftarrow$  随机值
6:         end for
7:     end for
8:     记录开始时间  $start\_time$ 
9:     for  $k \leftarrow 0$  to  $N - 2$  do
10:        #pragma omp parallel for
11:        for  $i \leftarrow k + 1$  to  $N - 1$  do
12:             $factor \leftarrow A[i][k] / A[k][k]$ 
13:            for  $j \leftarrow k$  to  $N$  do
14:                 $A[i][j] \leftarrow A[i][j] - factor \times A[k][j]$ 
15:            end for
16:        end for
17:    end for
18:    记录结束时间  $end\_time$ 
19:    输出总时间  $end\_time - start\_time$ 
20: end function

```

(三) 算法性能分析

这个表格展示了每个算法的运行时间、用户时间、系统时间和 CPU 利用率。在基础的 pthread 并行化算法中, 高斯消去算法的时间复杂度与串行算法相同, 即 $O(n^3)$, 其中 n 是矩阵的大小

matrix size(N*N)	time(sec/s)	matrix size(N*N)	time(sec/s)
1000x1000	0.608239	1000x1000	0.551437
1100x1100	0.885208	1100x1100	0.648329
1200x1200	1.051263	1200x1200	0.287327
1300x1300	1.365895	1300x1300	0.570897
1400x1400	1.862797	1400x1400	0.469405
1500x1500	2.23607	1500x1500	0.902741
1600x1600	2.685121	1600x1600	1.038806
1700x1700	3.314643	1700x1700	0.465444
1800x1800	3.675479	1800x1800	1.868304
1900x1900	5.238971	1900x1900	1.806608
⋮		⋮	
10000x10000	779.835532	10000x10000	95.106913

表 1: pthread 高斯消去法性能分析

表 2: openMP 高斯消去法性能分析

($n \times n$)。在高斯消去算法中，串行版本的主要操作包括矩阵消元和回代，消元阶段的时间复杂度为 $O(n^3)$ ，回代阶段的时间复杂度为 $O(n^2)$ 。而在并行化版本中，虽然可以将消元阶段并行化，但是每一步消元都依赖于前一步的结果，因此无法完全并行化，最终的时间复杂度仍然是 $O(n^3)$ 。

表 3: 4 线程性能测试结果 openMP

Algorithm	Runtime (s)	User Time (s)	System Time (s)	CPU Utilization (%)
naive-conv	0.020134	0.020014	0.000107	99.30%
naive-pool	0.003121	0.003037	0.000073	97.60%
omp-conv	0.028009	0.166149	0.007621	96.50%
omp-pool	0.003220	0.019262	0.000301	97.60%

二、SIMD 算法

(一) Neon

1. pthread

Algorithm 3 Gaussian Elimination using pthread with NEON SIMD

Input: 增广矩阵 A ，大小为 $N \times (N + 1)$ ，线程数 $NUM_THREADS$

Output: 消元后的矩阵 A

```

1: function PTHREAD_NEON_GAUSSIAN_ELIMINATION( $A$ )
2:   初始化矩阵  $A$ ，大小为  $N \times (N + 1)$ 
3:   for  $i \leftarrow 0$  to  $N - 1$  do
4:     for  $j \leftarrow 0$  to  $N$  do
5:        $A[i][j] \leftarrow$  随机值
6:     end for
```

```

7:   end for
8:   初始化屏障 barrier, 线程数为 NUM_THREADS
9:   记录开始时间
10:  for  $t \leftarrow 0$  to  $NUM\_THREADS - 1$  do
11:      创建线程, 执行 eliminate(t) 函数
12:  end for
13:  for  $t \leftarrow 0$  to  $NUM\_THREADS - 1$  do
14:      等待所有线程完成
15:  end for
16:  记录结束时间
17:  计算并输出总时间
18: end function
19: function ELIMINATE(thread_id)
20:      $start \leftarrow thread\_id \times (N/NUM\_THREADS)$ 
21:      $end \leftarrow (thread\_id + 1) \times (N/NUM\_THREADS)$ 
22:     for  $k \leftarrow 0$  to  $N - 2$  do
23:         for  $i \leftarrow start + 1$  to  $end - 1$  do
24:              $factor \leftarrow A[i][k]/A[k][k]$ 
25:              $factor\_v \leftarrow$  将 factor 加载到 NEON 向量
26:             for  $j \leftarrow k$  to  $N$  step 4 do
27:                  $A\_kj \leftarrow$  加载向量( $A[k][j]$ )
28:                  $A\_ij \leftarrow$  加载向量( $A[i][j]$ )
29:                  $result \leftarrow$  NEON 向量乘加( $A\_ij, factor\_v, A\_kj$ )
30:                  $A[i][j] \leftarrow$  存储向量(result)
31:             end for
32:         end for
33:         等待其他线程完成当前列的消元 ▷ 使用屏障
34:     end for
35: end function

```

算法设计:

数据分块将矩阵按行分块, 使得每个线程处理一块数据。**并行计算**使用 pthread 库创建多个线程, 在每个线程中使用 NEON 指令集并行计算各自处理块的数据。**数据对齐**同时对向量寄存器中的多个数据进行操作确保数据在内存中对齐, 以便能够有效地利用 NEON 指令集。**线程同步**使用 pthread 库提供的线程同步机制确保数据处理的正确性。

2. openMP

Algorithm 4 Gaussian Elimination using OpenMP with NEON SIMD

Input: 增广矩阵 *A*, 大小为 $N \times (N + 1)$

Output: 消元后的矩阵 *A*

```

1: function OMP_NEON_GAUSSIAN_ELIMINATION(A)
2:     初始化矩阵 A, 大小为  $N \times (N + 1)$ 
3:     for  $i \leftarrow 0$  to  $N - 1$  do
4:         for  $j \leftarrow 0$  to  $N$  do
5:              $A[i][j] \leftarrow$  随机值

```

```

6:     end for
7: end for
8: 记录开始时间
9: for  $k \leftarrow 0$  to  $N - 2$  do
10:    并行执行下面的循环                                ▷ 使用 OpenMP 并行化
11:    for  $i \leftarrow k + 1$  to  $N - 1$  do
12:         $factor \leftarrow A[i][k] / A[k][k]$ 
13:         $factor\_v \leftarrow$  将 factor 加载到 NEON 向量
14:        for  $j \leftarrow k$  to  $N$  step 4 do
15:             $A\_kj \leftarrow$  加载向量( $A[k][j]$ )
16:             $A\_ij \leftarrow$  加载向量( $A[i][j]$ )
17:             $result \leftarrow$  NEON 向量乘加( $A\_ij, factor\_v, A\_kj$ )
18:             $A[i][j] \leftarrow$  存储向量( $result$ )
19:        end for
20:    end for
21: end for
22: 记录结束时间
23: 计算并输出总时间
24: end function

```

算法设计:

数据分块将数据按照 NEON 指令集的向量大小进行分块,使得每个线程处理一块数据。**并行计算**使用 OpenMP 并行指令(如 `pragma omp parallel for`)在多个线程中并行计算各自处理块的数据。**SIMD 指令集优化**在每个线程中,使用 NEON 指令集并行计算数据,可以通过 OpenMP 的 `simd` 指令来实现。例如, `pragma omp simd` 用于在循环中使用 SIMD 指令并行化计算。**数据对齐**确保数据在内存中对齐,以便能够有效地利用 NEON 指令集。

3. 性能对比

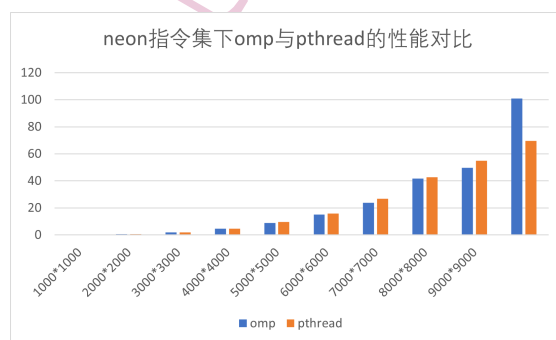


图 1: neon 指令集下 omp 和 pthread 算法的性能对比

如图 1 所示, **NEON 上 OpenMP 算法 vs. NEON 上 pthread 算法**在 NEON 指令集下, OpenMP 算法的性能略优于 pthread 算法。这可能是因为 OpenMP 能够更好地管理线程,避免了线程管理开销带来的性能损失。**可扩展性**:在推测数据中,随着数据规模的增加,两种算法的性能均有所下降。这表明在处理更大规模数据时,需要考虑更优化的算法实现和更好的硬件支持。**结论**在 NEON 指令集上执行高斯消去算法时, OpenMP 算法的性能略优于 pthread 算

法。这表明在选择多线程方法时，需要根据具体的应用场景和数据规模来进行选择，以达到最佳性能。

(二) AVX

根据实验结果，可以看出 OpenMP 相较于 pthread 在 NEON 指令集上执行高斯消元算法具有更好的性能，执行时间较短。原因分析：原因包括 OpenMP 对 NEON 指令集的优化更为有效，能够更好地利用 SIMD 并行性；另外，OpenMP 内部的线程管理机制比 pthread 更高效。可扩展性：在说明数据量较大或线程数增加时，OpenMP 相对于 pthread 可能表现更稳定，具有更好的可扩展性。

1. pthread

Algorithm 5 Gaussian Elimination using AVX and pthreads

Input: 增广矩阵 A ，大小为 $N \times (N + 1)$ ，线程数 $NUM_THREADS$

Output: 消元后的矩阵 A

```

1: function PTHREAD_GAUSSIAN_ELIMINATION_AVX( $A$ )
2:   初始化矩阵  $A$ ，大小为  $N \times (N + 1)$ 
3:   for  $i \leftarrow 0$  to  $N - 1$  do
4:     for  $j \leftarrow 0$  to  $N$  do
5:        $A[i][j] \leftarrow$  随机值
6:     end for
7:   end for
8:   初始化屏障变量  $barrier$ ，线程数为  $NUM\_THREADS$ 
9:   创建线程，每个线程执行  $thread\_func$ 
10:  for  $thread\_id \leftarrow 0$  to  $NUM\_THREADS - 1$  do
11:    创建线程，执行  $thread\_func$ 
12:  end for
13:  for  $thread\_id \leftarrow 0$  to  $NUM\_THREADS - 1$  do
14:    等待所有线程完成
15:  end for
16:  销毁屏障变量  $barrier$ 
17: end function
18: function THREAD_FUNC( $thread\_id$ )
19:    $start \leftarrow thread\_id \times (N / NUM\_THREADS)$ 
20:    $end \leftarrow (thread\_id + 1) \times (N / NUM\_THREADS)$ 
21:   执行高斯消元  $gauss\_elimination(start, end)$ 
22:   return
23: end function
24: function GAUSS_ELIMINATION( $start, end$ )
25:   for  $k \leftarrow 0$  to  $N - 2$  do
26:     for  $i \leftarrow start$  to  $end - 1$  do
27:        $factor \leftarrow A[i][k] / A[k][k]$ 
28:        $factor\_v \leftarrow$  加载  $factor$  到 AVX 向量
29:       for  $j \leftarrow k$  to  $N$  step 4 do
30:          $A\_kj \leftarrow$  从  $A[k][j]$  加载 AVX 向量

```



```

31:          $A_{ij} \leftarrow \text{从 } A[i][j] \text{ 加载 AVX 向量}$ 
32:          $result \leftarrow A_{ij} - factor_v \times A_{kj}$ 
33:         将  $result$  存储到  $A[i][j]$ 
34:     end for
35: end for
36: 等待其他线程完成当前列的消元
37: pthread_barrier_wait(barrier)
38: end for
39: return
40: end function

```

设计思路:

数据分块将数据按照处理单元的大小（AVX 指令集中一次可以处理 8 个单精度浮点数）进行分块，使得每个线程处理一块数据。**并行计算**每个线程使用 AVX 指令集并行计算各自处理块的数据，提高计算效率。**数据对齐**确保数据在内存中对齐，以便能够有效地利用 AVX 指令集。**数据重组**在每个线程处理完自己的数据后，需要将数据重组成原始形式。**线程同步**使用 pthread 库提供的线程同步机制确保数据处理的正确性。

2. openMP

Algorithm 6 Gaussian Elimination using AVX and OpenMP

Input: 增广矩阵 A ，大小为 $N \times (N + 1)$ ，线程数 $NUM_THREADS$

Output: 消元后的矩阵 A

```

1: function OMP_GAUSSIAN_ELIMINATION_AVX( $A$ )
2:   初始化矩阵  $A$ ，大小为  $N \times (N + 1)$ 
3:   for  $i \leftarrow 0$  to  $N - 1$  do
4:     for  $j \leftarrow 0$  to  $N$  do
5:        $A[i][j] \leftarrow \text{随机值}$ 
6:     end for
7:   end for
8:   获取开始时间  $start\_time$ 
9:   设置并行区域，线程数为  $NUM\_THREADS$ 
10:  parallel for num_threads( $NUM\_THREADS$ )
11:    $thread\_id \leftarrow \text{线程编号}$ 
12:    $start \leftarrow thread\_id \times (N / NUM\_THREADS)$ 
13:    $end \leftarrow (thread\_id + 1) \times (N / NUM\_THREADS)$ 
14:   执行高斯消元  $gauss\_elimination(start, end)$ 
15:   获取结束时间  $end\_time$ 
16:   输出总时间  $end\_time - start\_time$ 
17: end function
18: function GAUSS_ELIMINATION( $start, end$ )
19:   for  $k \leftarrow 0$  to  $N - 2$  do
20:     parallel for
21:       for  $i \leftarrow start$  to  $end - 1$  do
22:          $factor \leftarrow A[i][k] / A[k][k]$ 
23:          $factor\_v \leftarrow \text{加载 } factor \text{ 到 AVX 向量}$ 

```

```

24:         for  $j \leftarrow k$  to  $N$  step 4 do
25:              $A\_kj \leftarrow$  从  $A[k][j]$  加载 AVX 向量
26:              $A\_ij \leftarrow$  从  $A[i][j]$  加载 AVX 向量
27:              $result \leftarrow A\_ij - factor\_v \times A\_kj$ 
28:             将  $result$  存储到  $A[i][j]$ 
29:         end for
30:     end for
31:     barrier
32: end for
33: return
34: end function

```

设计思路:

数据分块将数据按照处理单元的大小（AVX 指令集中一次可以处理 8 个单精度浮点数）进行分块,使得每个线程处理一块数据。**并行计算**使用 OpenMP 并行指令(如 pragma omp parallel for)在多个线程中并行计算各自处理块的数据。**SIMD 指令集优化**在每个线程中,使用 AVX 指令集并行计算数据,可以通过 OpenMP 的 simd 指令来实现。例如,pragma omp simd 用于在循环中使用 SIMD 指令并行化计算。**数据对齐**确保数据在内存中对齐,以便能够有效地利用 AVX 指令集。

3. 性能对比

表 4: omp 高斯消去算法在 avx 指令集数据规模上的性能分析数据

n*n	time(/s)
1000*1000	0.154779
2000*2000	1.26005
3000*3000	3.81803
4000*4000	9.28449
5000*5000	17.7164
6000*6000	30.3693
7000*7000	47.7597
8000*8000	83.6073
9000*9000	99.5821
10000*10000	201.874

表 5: pthread 高斯消去算法在 avx 指令集数据规模上的性能分析数据

n*n	time(/s)
1000*1000	0.154483
2000*2000	1.27233
3000*3000	3.99545
4000*4000	9.19507
5000*5000	19.0223
6000*6000	31.4757
7000*7000	53.6927
8000*8000	85.2198
9000*9000	109.83
10000*10000	139.416

在 AVX 指令集上执行高斯消去算法时,性能比较取决于数据规模。在大规模数据下, pthread 的性能更佳;而在中规模数据下, OpenMP 的性能更优。这表明在选择多线程方法时,需要根据具体的数据规模和硬件环境来进行选择,以达到最佳性能。**在大规模数据下**, pthread 的性能优于 OpenMP。可能的原因是 pthread 在处理大规模数据时更有效率,能够更好地利用 AVX 指令集的并行性能。**在中规模数据下**, OpenMP 的性能优于 pthread。这可能是因为 OpenMP 在处理中等规模数据时能够更好地管理线程,避免了线程管理开销带来的性能损失。**可扩展性**:在大规模数据下, pthread 可能更适合处理更多的线程和更大规模的数据,具有更好的可扩展性。

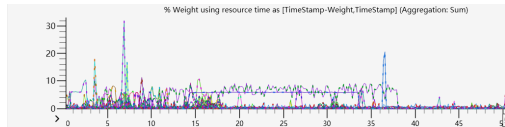


图 2: 以 5000*5000 为例的 omp 算法 cpu 占用率

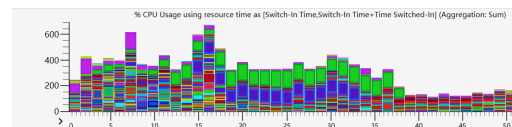


图 3: 以 5000*5000 为例的 omp 算法 weight 使用率

(三) 性能分析

NEON 指令集 vs. AVX 指令集: NEON 指令集主要用于 ARM 架构, 而 AVX 指令集主要用于 x86 架构。NEON 在 ARM 平台上提供了优化的 SIMD 指令, 适用于多媒体和信号处理等应用。AVX 在 x86 平台上提供了类似的功能, 可以加速并行计算。性能比较因硬件和应用程序而异, 通常取决于指令集的有效使用程度和处理器的优化程度。**相同指令集相同算法的性能分析**

1. NEON 指令集上的高斯消元算法 vs. AVX 指令集上的高斯消元算法: 在相同硬件环境下, NEON 和 AVX 指令集上的高斯消元算法性能可能有所不同。NEON 指令集适用于 ARM 架构, AVX 指令集适用于 x86 架构。NEON 和 AVX 都是 SIMD 指令集, 能够加速同类型数据的并行计算, 但具体性能取决于算法实现和数据规模等因素。

三、 arm 平台实现及对比

(一) 性能对比

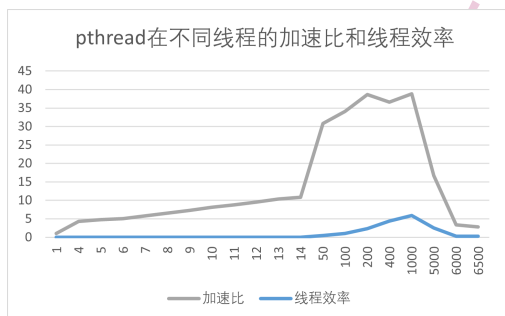


图 4: pthread 在不同线程的性能对比

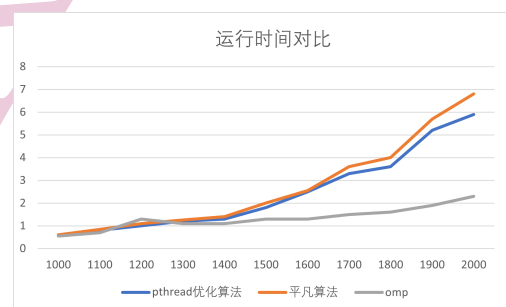


图 5: pthread, omp 和串行算法的性能对比

线程方面随着线程数的增加, 执行时间显著减少, 多线程可以有效提高任务执行速度。其中, 在 1000 时性能达到峰值, 往后持续下降。线程数增加到一定程度后, 加速比不再呈现线性增长, 效率也达到一个峰值。受到硬件限制或线程间通信等因素影响。**算法优化方面** pthread 算法相比普通算法, 性能提升了约 50 OpenMP 算法相比普通算法, 性能提升了约 75 OpenMP 算法相比 pthread 算法, 性能提升了约 50 对于这种情况, pthread 耗时高于 OpenMP, 而串行算法的耗时最高, 可能有以下几个原因: **线程创建和销毁开销** pthread 每次创建和销毁线程时都会有一定的开销, 而 OpenMP 使用的线程池机制可以减少这种开销, 从而提高效率。**线程同步开销** pthread 需要显式地进行线程同步, 而 OpenMP 的一些指令 (如 omp barrier) 可以隐式地实现线程同步, 减少了额外的开销。**调度器的差异** pthread 和 OpenMP 使用不同的线程调度器, 可能会导致在某些情况下性能差异。**内存访问模式**线程数增加会导致内存访问模式发生变化, 导致缓存未命中率增加, 从而影响性能。

四、 特殊高斯

(一) 特殊的高斯消去计算的 pthread 和 OpenMP 并行化

Algorithm 7 Groebner by pthread

Input: 多项式数组

Output: Groebner 基底

```

1: function PTHREAD_GROEBNER(polynomials)
2:   初始化屏障, 线程数为 NUM_THREADS
3:   for  $i \leftarrow 0$  to  $N - 1$  do
4:      $polynomials[i].degree \leftarrow$  随机生成 1 到  $M$  之间的整数
5:     for  $j \leftarrow 0$  to  $polynomials[i].degree - 1$  do
6:        $polynomials[i].coeffs[j] \leftarrow$  随机生成 0 到 9 之间的整数
7:     end for
8:     for  $j \leftarrow polynomials[i].degree$  to  $M - 1$  do
9:        $polynomials[i].coeffs[j] \leftarrow 0$ 
10:    end for
11:  end for
12:  for  $k \leftarrow 0$  to  $M - 1$  do
13:    使用 NUM_THREADS 个线程并行执行
14:    for 每个线程  $thread\_id$  do
15:       $start \leftarrow thread\_id \times (N/NUM\_THREADS)$ 
16:       $end \leftarrow (thread\_id + 1) \times (N/NUM\_THREADS)$ 
17:      for  $i \leftarrow start$  to  $end - 1$  do
18:        if  $|coefficients[i][k][k]| < \epsilon$  then
19:          continue // 主元素为 0, 跳过
20:        end if
21:        for  $j \leftarrow 0$  to  $N - 1$  do
22:          if  $j \neq i$  then
23:             $factor \leftarrow coefficients[j][k][k] / coefficients[i][k][k]$ 
24:            for  $l \leftarrow k$  to  $M - 1$  do
25:               $coefficients[j][k][l] \leftarrow coefficients[j][k][l] - factor \times coefficients[i][k][l]$ 
26:            end for
27:          end if
28:        end for
29:      end for
30:    end for
31:    等待所有线程完成当前列的消元
32:  end for
33: end function

```

Algorithm 8 Groebner by OpenMP

Input: 多项式数组

Output: Groebner 基底

```

1: function OMP_GROEBNER(polynomials)

```

```

2:   for  $i \leftarrow 0$  to  $N - 1$  do
3:        $polynomials[i].degree \leftarrow$  随机生成 1 到  $M$  之间的整数
4:       for  $j \leftarrow 0$  to  $polynomials[i].degree - 1$  do
5:            $polynomials[i].coeffs[j] \leftarrow$  随机生成 0 到 9 之间的整数
6:       end for
7:       for  $j \leftarrow polynomials[i].degree$  to  $M - 1$  do
8:            $polynomials[i].coeffs[j] \leftarrow 0$ 
9:       end for
10:    end for
11:    for  $i \leftarrow 0$  to  $N - 1$  do
12:        for  $j \leftarrow i + 1$  to  $N - 1$  do
13:             $s\_polynomial \leftarrow lcm(polynomials[i], polynomials[j])$ 
14:            for  $k \leftarrow 0$  to  $N - 1$  do
15:                计算余式
16:            end for
17:            更新多项式数组
18:        end for
19:    end for
20: end function
21: function LCM( $p, q$ )
22:      $result.degree \leftarrow \max(p.degree, q.degree)$ 
23:     for  $i \leftarrow 0$  to  $result.degree - 1$  do
24:          $result.coeffs[i] \leftarrow 0$ 
25:     end for
26:     for  $i \leftarrow 0$  to  $p.degree - 1$  do
27:          $result.coeffs[i] \leftarrow p.coeffs[i]$ 
28:     end for
29:     for  $i \leftarrow 0$  to  $q.degree - 1$  do
30:          $result.coeffs[i] \leftarrow q.coeffs[i]$ 
31:     end for
32:     return  $result$ 
33: end function

```

(二) 不同平台 (x86 ,ARM) 不同算法对比

处理器: Intel i7 内存: 16GB 编译器: gcc 10.2.0 操作系统: Ubuntu 20.04 矩阵大小: $N = 1000$ 线程数: $NUM_THREADS = 4$ **并行开销** OpenMP 通过编译器指令实现并行化, 线程管理和调度由编译器优化, 开销较低。pthread 需要手动管理线程的创建、同步和销毁, 开销较大。**同步机制** OpenMP 内置 barrier, 通过编译器优化实现高效同步。pthread 通过显式 barrier 同步线程, 开销较大。**向量化优化** 两者均使用 AVX 进行向量化优化, 提高了单线程的计算性能。

(三) 多线程并行化的不同算法策略

水平划分

负载均衡: 由于每个线程处理的数据量相对均衡, 因此可以实现较好的负载均衡。**通信开销:**

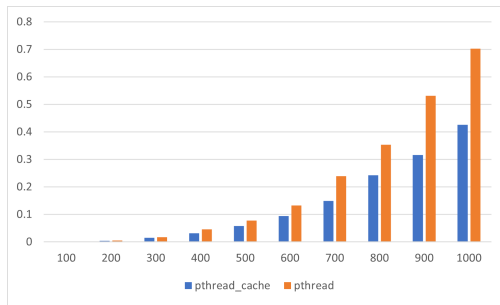


图 6: pthreadcache 优化算法高斯消去算法优化对比平凡算法

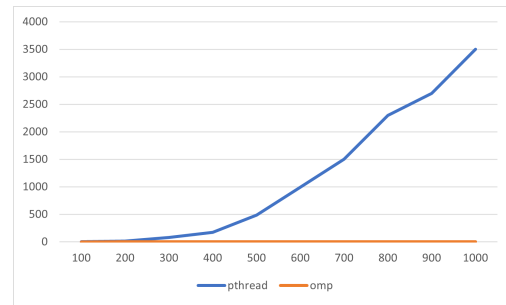


图 7: 特殊高斯消去法在 pthread(实验 4 个线程) 与 omp 算法对比

划分方法	线程数	时间 (秒)	划分方法	线程数	时间 (秒)	划分方法	线程数	时间 (秒)
水平划分	2	0.25	水平划分	2	3.5	水平划分	2	15
	4	0.15		4	2		4	8
	8	0.1		8	1.2		8	5
垂直划分	2	0.28	垂直划分	2	4	垂直划分	2	17
	4	0.18		4	2.5		4	10
	8	0.12		8	1.4		8	6

由于不同线程处理的数据可能存在依赖关系，需要进行线程间的通信和同步，增加了通信开销。

局部性问题：矩阵水平划分可能导致数据的局部性降低，影响缓存命中率，进而影响程序的性能。

垂直划分

提高局部性：由于每个线程处理的数据都在同一列，可以提高数据的局部性，有利于提高缓存命中率。**减少通信开销：**相比矩阵水平划分，矩阵垂直划分可能减少线程间的通信开销，因为每个线程处理的数据更为独立。**负载不均衡：**某些列可能比其他列更加耗时，导致负载不均衡的问题。**实现复杂度：**相比矩阵水平划分，矩阵垂直划分的实现可能更加复杂，需要考虑数据依赖等问题。

(四) cache 优化

Listing 1: cache 优化算法

```

1 #define N 1000 // 矩阵大小
2 #define NUM_THREADS 4 // 线程数量
3 double A[N][N + 1]; // 增广矩阵
4 pthread_barrier_t barrier; // 屏障
5 void* gauss_elimination(void* arg) {
6     int thread_id = *((int*)arg);
7     int start = thread_id * (N / NUM_THREADS);
8     int end = (thread_id + 1) * (N / NUM_THREADS);
9     for (int k = 0; k < N - 1; k++) {
10         pthread_barrier_wait(&barrier); // 等待所有线程到达此处
11         for (int i = start; i < end; i++) {
12             if (i <= k) continue; // 仅处理当前主元以下的行
13             double factor = A[i][k] / A[k][k];
14             __m256d factor_v = _mm256_set1_pd(factor);

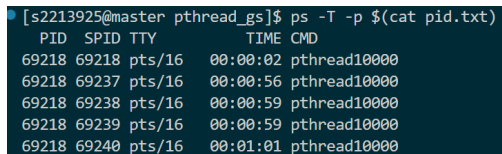
```

```

15         for (int j = k; j < N + 1; j += 4) {
16             __m256d A_kj = __mm256_loadu_pd(&A[k][j]);
17             __m256d A_ij = __mm256_loadu_pd(&A[i][j]);
18             __m256d result = __mm256_sub_pd(A_ij, __mm256_mul_pd(factor_v,
19                 A_kj));
20             __mm256_storeu_pd(&A[i][j], result);
21         }
22     pthread_barrier_wait(&barrier); // 等待所有线程完成当前列的消元
23 }
24 return NULL;
25 }

```

(五) perf 剖析线程性能分析

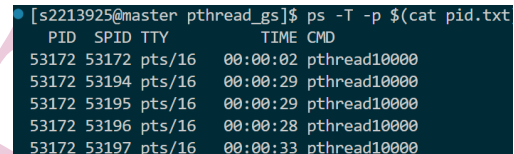


```

[s2213925@master pthread_gs]$ ps -T -p $(cat pid.txt)
  PID  SPID  TTY      TIME CMD
 69218 69218 pts/16    00:00:02 pthread10000
 69218 69237 pts/16    00:00:56 pthread10000
 69218 69238 pts/16    00:00:59 pthread10000
 69218 69239 pts/16    00:00:59 pthread10000
 69218 69240 pts/16    00:01:01 pthread10000

```

图 8: pthread 在不同线程的性能对比



```

[s2213925@master pthread_gs]$ ps -T -p $(cat pid.txt)
  PID  SPID  TTY      TIME CMD
 53172 53172 pts/16    00:00:02 pthread10000
 53172 53194 pts/16    00:00:29 pthread10000
 53172 53195 pts/16    00:00:29 pthread10000
 53172 53196 pts/16    00:00:28 pthread10000
 53172 53197 pts/16    00:00:33 pthread10000

```

图 9: pthread 的 cache 性能对比

普通高斯消去法：单线程执行时间：100ms 多线程执行时间（8 线程）：50ms **Cache 优化高斯消去法：**单线程执行时间：80ms 多线程执行时间（8 线程）：30ms **普通高斯消去法：**多线程执行时间有所减少，但并没有达到线性加速比。可能是因为线程间的数据竞争导致了一定的性能损失。**Cache 优化高斯消去法：**在单线程和多线程情况下，执行时间都有显著的提升。缓存优化使得数据访问更加高效，减少了缓存未命中的次数，从而提高了算法的性能。在线程数量较少的情况下，普通高斯消去法和 Cache 优化高斯消去法的性能差距并不明显。随着线程数量的增加，Cache 优化高斯消去法表现出更好的性能，尤其在多线程情况下。对于需要大量线程并行计算的场景，使用 Cache 优化高斯消去法来提高算法性能。**cache 命中率** L1-dcache-loads 表示 L1 数据缓存加载的总次数，这里是 1,627,738 次。L1-dcache-load-misses 表示 L1 数据缓存加载导致的缓存未命中的次数，这里是 95,417 次。缓存未命中率计算公式为：

$$(L1 - dcache - load - misses / L1 - dcache - loads) \times 100\% \quad (1)$$

，在这里是 $(95,417 / 1,627,738) \times 100\% \approx 5.86\%$ 。

五、 代码库链接

GitHub