



南开大学  
Nankai University

南 开 大 学

计 算 机 学 院

并行程序设计实验报告

---

多线程 pthread&OpenMP 编程

---

蒋薇

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2023 年 5 月 14 日

# 摘要

关键字：Parallel

## 目录

<b>一、普通高斯消去基础 pthread/OpenMP 并行化实验</b>	<b>1</b>
(一) 设计分析 . . . . .	1
1. 划分 . . . . .	1
2. SIMD 结合 . . . . .	4
3. 复杂度分析 . . . . .	5
4. 串行 pthread1 实验对比 . . . . .	9
5. 串行 OpenMP 实验对比 . . . . .	9
6. pthread1OpenMP 实验对比 . . . . .	9
7. pthreadOpenMP 比较 . . . . .	15
<b>二、特殊高斯消去 pthread/OpenMP 并行化实验</b>	<b>15</b>
(一) 设计分析 . . . . .	15
1. 简介 . . . . .	15
2. 算法思想 . . . . .	15
3. 实现 . . . . .	16
4. 分析 . . . . .	16
<b>三、不同平台及算法策略的比较</b>	<b>17</b>
(一) ARM 与 X86 平台上比较 . . . . .	17
(二) 多线程并行化不同算法策略 . . . . .	18

## 一、普通高斯消去基础 pthread/OpenMP 并行化实验

### (一) 设计分析

#### pthread 部分:

外层循环共进行  $n$  轮消去步骤, 第一个内层循环第  $k$  轮执行完第  $k$  行除法后, 对于第二个内层循环的  $k+1$  至  $n$  行进行减去第  $k$  行的操作, 各行之间互不影响, 可采用多线程执行。

因为内外层循环之间并无计算, 外层循环的执行实际上只是在持续执行内层循环, 外层循环执行的全过程都是处于并行部分中, 因此将双重循环都置于线程函数中, 对外层循环保持不动, 将内层循环拆分分配工作线程。

将多重循环都纳入线程函数中, 消去操作对应的内层循环拆分, 分配给工作线程。对于除法操作, 可以只由一个线程执行, 也可拆分由所有工作线程并行执行。

**注意 1:** 除法和消去两个步骤后都要进行同步, 以保证进入下一步骤 (下一轮) 之前上一步骤的计算全部完成, 保证一致性。

同步问题可参考忙等待、互斥量和信号量、barrier 与条件变量等实现。

**注意 2:** 如果不得不划分出一些计算量大小不一的任务, 不便于手动分配给每个线程, 考虑构建任务池, 即任务队列, 把待完成的任务放进去。每当一个线程完成当前任务后, 立刻到任务池中再选取一个任务执行, 直到所有任务被完成。

**注意 3:** 均分任务时, 当问题规模不等于线程数倍数时, 对于分配最后一部分计算任务的线程不要直接使用  $my\_first+my\_n$  计算  $my\_last$ , 可根据线程 id 将  $n$  作为该线程的  $my\_last$ 。

高斯消去过程中的计算集中在第一个内层循环 (除法和第二个内层循环 (双重循环, 消去), 对应矩阵右下角  $(n-k+1) \times (n-k)$  的子矩阵。因此, 任务划分可以看作对此子矩阵的划分。对于除法部分, 只涉及一行, 只可能采用列划分, 而对于消去部分, 即可采用水平划分 (将其外层循环拆分, 即每个线程分配若干行), 也可采用垂直划分 (将其内层循环拆分, 即每个线程分配若干列)。

大致框架: 使用 `pthread_create` 函数创建工作线程, 将并行部分抽取出来形成线程函数, 工作线程执行线程函数进行并行计算, 主函数调用 `pthread_join` 函数等待工作线程完成。

#### 1. 划分

使用 pthreads 实现以下高斯消除算法的并行版本

理解顺序实现和 pthreads 基础上, 通过其并行版本构建逻辑的提示, 如线程工作分配标准, 循环并行化等, 继续探索。

划分时应注意, 矩阵中每一行的工作需要完成所有前面的行, 在一行中, 每一列都可以并行处理, 但需要注意的是第  $k$  列的原始值必须保存并用于其他列的计算. 这对应于  $j$  值。

j 上只有一个循环算法

```

1  procedure GAUSSIAN ELIMINATION (A, b, y)
2  begin
3      for k := 0 to n - 1 do /* Outer loop */
4          begin
5              for j := k + 1 to n - 1 do
6                  begin
7                      A[k, j] := A[k, j]/A[k, k]; /* Division step */
8                      for i := k + 1 to n - 1 do
9                          A[i, j] := A[i, j] - A[i, k] × A[k, j]; /* Elimination step
                                                                    */

```

```

10     endfor;
11     y[k] := b[k]/A[k, k];
12     A[k, k] := 1;
13     for i := k + 1 to n - 1 do
14     begin
15         b[i] := b[i] - A[i, k] × y[k];
16         A[i, k] := 0;
17     endfor;
18 endfor;
19 end GAUSSIAN ELIMINATION
20 }

```

j 上的循环体仅访问 j 和 k 列中的值，这是可以完成的循环在平行下。然后可以进一步注意到外循环的第二部分不依赖于第一部分，因此可以将外循环分成两部分：

#### 外循环分成两部分

```

1  procedure GAUSSIAN ELIMINATION (A, b, y)
2  begin
3      for k := 0 to n - 1 do /* Outer loop */
4      begin
5          for j := k + 1 to n - 1 do
6          begin
7              A[k, j] := A[k, j]/A[k, k]; /* Division step */
8              for i := k + 1 to n - 1 do
9                  A[i, j] := A[i, j] - A[i, k] × A[k, j]; /* Elimination step */
10             endfor;
11         endfor;
12     endfor;
13     for k := 0 to n - 1 do /* Outer loop, second pass */
14     begin
15         y[k] := b[k]/A[k, k];
16         A[k, k] := 1;
17         for i := k + 1 to n - 1 do
18         begin
19             b[i] := b[i] - A[i, k] × y[k];
20             A[i, k] := 0;
21         endfor;
22     endfor;
23 end GAUSSIAN ELIMINATION

```

预先创建线程，每个线程负责对从 0 到 j 值的子集执行 j 循环 n - 1。这些线程在处理每一行后将需要一个同步屏障，因为处理下一行需要前一行的所有列的结果。为此，可以使用 pthread\_barrier\_wait()。

并非每一列 (j 的值) 都有相同的工作。列 j 被该循环处理 j 次 (因此列 0 执行 0 次，而列 n - 1 执行 n - 1 次)。需要为线程分配列号，以便每个线程对每一行的工作量尽可能接近，这可以通过以循环方式分配列来完成。

例如。如果三个线程 A、B 和 C 以及从 0 到 9 的 10 列，分配它们：

Thread A: 0, 3, 6, 9

Thread B: 1, 4, 7,

Thread C: 2, 5, 8,

线程函数为:

#### 线程函数

```

1  for k := 0 to n - 1 do /* Outer loop */
2  begin
3      call pthread_barrier_wait(row_barrier);
4      for j := k + 1 + thread_number to n - 1 step n_threads do
5          begin
6              A[k, j] := A[k, j]/A[k, k]; /* Division step */
7              for i := k + 1 to n - 1 do
8                  A[i, j] := A[i, j] - A[i, k] × A[k, j]; /* Elimination step */
9              endfor;
10         endfor;
11     endfor;
12     call pthread_barrier_wait(phase1_barrier);

```

主要功能为:

#### 主要功能

```

1  procedure GAUSSIAN ELIMINATION (A, b, y)
2  begin
3      call start_threads;
4      call pthread_barrier_wait(phase1_barrier);
5
6      for k := 0 to n - 1 do /* Outer loop, second pass */
7          begin
8              y[k] := b[k]/A[k, k];
9              A[k, k] := 1;
10             for i := k + 1 to n - 1 do
11                 begin
12                     b[i] := b[i] - A[i, k] × y[k];
13                     A[i, k] := 0;
14                 endfor;
15             endfor;
16         end GAUSSIAN ELIMINATION

```

线程处理的数据是相对独立，可以避免数据竞争和死锁等问题。注意在合并处理结果时，使用同步机制来避免多个线程同时修改同一个变量的问题，使用互斥锁、条件变量等同步机制来实现线程间的同步。`pthread_create` 函数创建线程，同步机制合并到主线程中，`pthread_exit` 释放资源。

对于除法部分，只涉及一行，只可能采用垂直划分，即列划分，对于消去部分，可采用水平划分，将其外层循环拆分，即每个线程分配若干行，也可采用垂直划分，将其内层循环拆分，即每个线程分配若干列。

**OpenMP 部分:**

OpenMP 是跨平台的多核多线程编程的一套指导性的编译处理方案 (Compiler Directive), 指导编译器将代码编译为多线程程序。Openmp 的执行模型采用 fork-join 的形式, 其中 fork (派生) 创建新线程或者唤醒已有线程; join 即多线程的会合。

在变上三角的时候, 每一次清除一列元素时, 每一行的元素需要进行一次数字运算, 行与行之间没有数据依赖, 可以并行; 在上三角变对角线时同理并行。使用 omp.h 对 for 循环 (对每一行进行数学运算) 进行并行,

核心命令 `pragma omp parallel for num_threads( )`

OpenMP 规范中的指令:

`parallel`: 用在结构块之前, 表示这段代码将被多个线程并行执行; `for`: 用于 for 循环语句之前, 表示将循环计算任务分配到多个线程中并行执行, 以实现任务分担, 必须由编程人员自己保证每次循环之间无数据相关性; `sections`: 用在可被并行执行的代码段之前, 用于实现多个结构块语句的任务分担, 可并行执行的代码段各自用 `section` 指令标出 (注意区分 `sections` 和 `section`); `single`: 用在并行域内, 表示一段只被单个线程执行的代码; `critical`: 用在一段代码临界区之前, 保证每次只有一个 OpenMP 线程进入。

相应的 OpenMP 子句:

`private`: 指定一个或多个变量在每个线程中都有它自己的私有副本; `shared`, 声明一个或多个变量是共享变量; `reduction`: 用来指定一个或多个变量是私有的, 并且在并行处理结束后这些变量要执行指定的归约运算, 并将结果返回给主线程同名变量; `flush`: 确保同步时程序被正确写入, `flush` 指令将列表中的变量执行 `flush` 操作, 直到所有变量都已完成相关操作后才返回; `num_threads`

常用的环境变量:

`OMP_SCHEDULE`: 用于 for 循环并行化后的调度, 它的值就是循环调度的类型;

`OMP_NUM_THREADS`: 用于设置并行域中的线程数; `OMP_DYNAMIC`: 通过设定变量值, 来确定是否允许动态设定并行域内的线程数; `OMP_NESTED`: 指出是否可以并行嵌套。

**负载均衡**: 对于倒数第  $k$  行, 消去过程的计算量为  $O(k^2)$ , 使用纯块划分会使得处理头几行的线程负载要高于处理最后几行的线程, 在 OpenMP 的 `schedule` 子句默认的划分方式 (`static`) 就是这样, 通过设置 `chunk_size`, 使用循环划分或者块循环划分来均衡负载, 注意避免负载不均。

**实现方法**: 使用 `pragma omp parallel for` 指令来实现并行计算, 将循环体中的代码块并行化执行, 每次迭代中, 每个线程都可以独立地计算一个矩阵元素, 并更新矩阵的部分区域。OpenMP 可以自动将线程数调整到适当的数量, 无需手动指定线程数, 提高了程序的可移植性。

**注意**: 注意数据的共享和私有性, `private(j, t)` 指令声明  $j$  和  $t$  变量为私有变量, 避免了线程间的数据共享和竞争, 还需要注意线程间的同步和互斥, 避免数据的冲突和错误。

## 2. SIMD 结合

### pthread 部分:

使用 pthread 创建多个线程, 每个线程负责处理一个矩阵块。在每个线程中, 使用 SIMD 指令对矩阵块进行高斯消元操作。SIMD 只能将行内连续元素的运算打包进行向量化, 即只能对最内层循环进行展开、向量化。

在每个线程中, `pthread_barrier_wait()` 函数实现线程同步, 确保所有线程完成当前矩阵块的计算后再进入下一个矩阵块的计算。

**结合 SIMD pthread github 地址** [pthread 请点击这里](#)

### OpenMP 部分:

内层循环中的计算使用 SIMD 指令进行向量化, 使用 `pragma omp parallel for` 指令和 AVX 指令集中的 `__m256d` 数据类型来实现 SIMD 并行计算

## OpenMP 与 SIMD 结合简要代码

```

1  #pragma omp parallel for private(j,t)
2  for (i = k + 1; i < n; i++){
3      t = A[i][k]/A[k][k];
4      __m256d tt = __mm256_load_pd(&A[k][j]);
5      __m256d aij = __mm256_load_pd(&A[i][j]);
6      __m256d aijt = __mm256_mul_pd(tt, aij);
7      __m256d aijajt = __mm256_sub_pd(aij, aijt);
8      __m256_store_pd(&A[i][j], aijajt);
9  }

```

## 3. 复杂度分析

## pthread 部分:

时间复杂度 高斯消去算法的时间复杂度为  $O(n^3)$ ，使用 pthread 并行计算时，将矩阵的每一行分配给不同的线程，在每一行计算时使用 pthread\_barrier\_wait() 函数来同步线程的执行。

线程数和线程同步分析 线程数的选择需要考虑到系统的硬件资源和任务的并行性。一般情况下，线程数应该等于 CPU 核心数。如果线程数过多，会导致线程切换开销过大，反而降低了并行计算的效率。

使用 pthread\_barrier\_wait() 函数可以实现线程的同步，保证每一行计算完成后再进行下一步计算。此外，还可以使用 pthread\_mutex\_lock() 和 pthread\_mutex\_unlock() 函数实现线程之间的互斥操作，避免多个线程同时访问同一资源。

静态动态数据分配策略分析 静态分配策略指的是将矩阵的每一行按照顺序分配给线程，而动态分配策略则是根据每个线程的工作负载来动态分配矩阵的行。将矩阵的每一行分配到不同的线程上，采用静态分配策略或者动态分配策略提高效率。

## 具体实现

## 动态线程

首先从文件中读入矩阵，并根据矩阵的大小动态地创建线程。每个线程负责处理一个连续的行区间，从第  $i$  行开始，每隔  $n$  行处理一次，直到处理完整个矩阵。

## 静态线程 + 信号量同步版本

信号量数组 sem，每次线程完成一列的计算后，通过 sem\_post 函数向该列对应的信号量发送信号，以通知下一个线程开始计算。

## 静态线程 + 信号量同步版本 + 三重循环全部纳入线程函数

初始化  $N$  个信号量，每个信号量表示一个线程完成了一次高斯消去操作。接着，我们创建  $N$  个线程，每个线程执行高斯消去算法的一部分，并在完成后通过信号量通知主线程。最后，我们等待所有线程完成，销毁所有信号量，输出程序的运行时间。

将三重循环全部纳入线程函数，这使得每个线程需要处理  $N(N+1)/2$  个元素，因此线程数应该设置为  $N$ ，以充分利用多核处理器的并行性能。

静态线程 + 信号量同步版本 + 三重循环全部纳入线程函数 [github 地址 请点击这里](#)

## 静态线程 + barrier 同步

在主函数中初始化 barrier 同步对象，然后在高斯消元的每个阶段对其进行等待，以保证所有线程都完成了当前阶段的计算 pthread + 动态/静态实现高斯消元 [github 地址 请点击这里](#)

不同策略实现高斯消去求解程序中已添加测试程序运行时间代码。

## OpenMP 部分:

for 调度中的 schedule 子句: static 静态分配方式: pragma omp parallel for schedule(static, 2)



**dynamic 动态分配方式, 不指定 size:** `pragma omp parallel for schedule(dynamic)`

**dynamic 动态分配方式, 指定 size:** `pragma omp parallel for schedule(dynamic, 2)`

**for 调度:** 使用 `schedule` 子句来实现, 格式为: `schedule (type, size)`。四种 `type`: `static`、`dynamic`、`guided`、`runtime`, 如果没有指定 `size` 大小, 循环迭代会尽可能平均地分配给每个线程。

`static`: “静态”可以从逻辑上推断出哪几次迭代会在哪几个线程上运行, 对于一个  $N$  次迭代, 使用  $M$  个线程, 那么,  $[0, \text{size}-1]$  的 `size` 次的迭代是在第一个线程上运行,  $[\text{size}, \text{size} + \text{size} - 1]$  是在第二个线程上运行, 依次类推。

`dynamic`: 较快的线程抢到更多的任务, 没有 `size` 参数的情况下, 每个线程按先执行完先分配的方式执行 1 次循环; `dynamic` 也可以设置 `size` 参数, `size` 表示每次线程执行完 (空闲) 的时候给其一次分配的迭代的数量。

`guided`: 采用指导性的启发式自调度方式。

`runtime`: 表示根据环境变量确定上述调度策略中的某一种, 默认也是静态的, 控制 `schedule` 环境变量的是 `OMP_SCHEDULE` 环境变量。

**伸缩性分析:** 使用不同的线程数, `pragma omp parallel for num_threads`, 通过改变 `num_threads` 的值, 比较不同线程数下算法的运行时间和加速比来评估算法的伸缩性。

算法的加速比随着线程数的增加而增加, 说明算法的并行性好; 如果加速比达到峰值后开始下降, 说明算法的并行性存在瓶颈; 如果加速比很小, 说明算法的并行性很差。

**Cache 虚假共享:** 多级缓存架构下, 核 B 试图写入核 A 的 `cache` 中存在的某个高速缓存行, 使得核 A 中的该缓存行失效, 下一次核 A 访问该缓存行不命中, 该现象被称为虚假共享。使用在共享访问的变量间适当添加填充, 对齐到高速缓存行的大小, 或者在多线程访问连续内存时, 适当提高 `chunk_size`, 使得一个高速缓存行放在一个线程里解决。

线程数据结构及线程函数定义

```

1  typedef struct {
2      int k; // 消去的次数
3      int t_id; // 线程 id
4  } threadParam_t;
5
6
7  void *threadFunc(void * param) {
8      threadParam_t * p = (threadParam_t *)param;
9      int k = p->k; // 消去的次数
10     int t_id = p->t_id; // 线程编号
11     int i = k + t_id + 1; // 当前的计算任务
12 }

```

测试用例生成 初始化矩阵时可以首先初始化一个上三角矩阵, 然后随机的抽取若干行去将它们相加减然后执行若干次, 由于这些都是内部的线性组合, 这样的初始数据可以保证进行高斯消去时矩阵不会有 `inf` 和 `nan`。

### pthread 部分

串行

Gaussian 消去算法原型

```

1  procedure GAUSSIAN ELIMINATION (A, b, y)
2  begin
3      for k := 0 to n - 1 do /* Outer loop */

```



```

4   begin
5       for j := k + 1 to n - 1 do
6           A[k, j] := A[k, j]/A[k, k]; /* Division step */
7       y[k] := b[k]/A[k, k];
8       A[k, k] := 1;
9       for i := k + 1 to n - 1 do
10          begin
11              for j := k + 1 to n - 1 do
12                  A[i, j] := A[i, j] - A[i, k] × A[k, j]; /* Elimination step
13                      */
14                  b[i] := b[i] - A[i, k] × y[k];
15                  A[i, k] := 0;
16              endfor; /* Line 9 */
17          endfor; /* Line 3 */
18  end GAUSSIAN ELIMINATION
19  }

```

Forward Elimination, 把矩阵变为倒三角形, Back Substitution, 反向求出解。

github 地址 [串行请点击这里](#)

实验结果截图:

Matrix dimension	Elapsed time
200.	12.868 ms.
400.	63.572 ms.
600.	208.144 ms.
800.	561.739 ms.
1000.	1004.53 ms.
1200.	1517.96 ms.
1400.	2400.12 ms.
1600.	3713.21 ms.
1800.	5227.6 ms.
2000.	7702.5 ms.
2200.	10360.7 ms.
2400.	12893 ms.
2600.	16075.1 ms.
2800.	20239.5 ms.
3000.	25334.9 ms.
3200.	29976.6 ms.
3400.	35193.2 ms.
3600.	42814.5 ms.
3800.	50294.4 ms.
4000.	57756.6 ms.
4200.	68833.7 ms.
4400.	76231.6 ms.
4600.	87870.6 ms.
4800.	101829 ms.
5000.	110317 ms.
5200.	120172 ms.
5400.	148310 ms.
5600.	173519 ms.
5800.	191267 ms.
6000.	206010 ms.
6200.	219028 ms.
6400.	241775 ms.
6600.	267036 ms.
6800.	290154 ms.
7000.	311076 ms.
7200.	344624 ms.
7400.	377756 ms.
7600.	418067 ms.
7800.	428291 ms.
8000.	469039 ms.

pthread

将内层循环作为一个函数,每次创建一个线程就调用这个函数,也就是这个函数被 `pthread_create` 创建的线程调用,等每个线程将“0”算出,就把每个线程调用的函数之前申请的空间释放,然后将这个线程销毁。

不同规模

github 地址 [pthread 请点击这里](#)

实验结果截图:

```

parallel-threads: Matrix dimension: 200. Elapsed time = 6.081 ms.
parallel-threads: Matrix dimension: 400. Elapsed time = 56.487 ms.
parallel-threads: Matrix dimension: 600. Elapsed time = 150.043 ms.
parallel-threads: Matrix dimension: 800. Elapsed time = 302.29 ms.
parallel-threads: Matrix dimension: 1000. Elapsed time = 562.468 ms.
parallel-threads: Matrix dimension: 1200. Elapsed time = 883.419 ms.
parallel-threads: Matrix dimension: 1400. Elapsed time = 1566.63 ms.
parallel-threads: Matrix dimension: 1600. Elapsed time = 2061.79 ms.
parallel-threads: Matrix dimension: 1800. Elapsed time = 2942.43 ms.
parallel-threads: Matrix dimension: 2000. Elapsed time = 4238.44 ms.
parallel-threads: Matrix dimension: 2200. Elapsed time = 5801.84 ms.
parallel-threads: Matrix dimension: 2400. Elapsed time = 7447.86 ms.
parallel-threads: Matrix dimension: 2600. Elapsed time = 9812.56 ms.
parallel-threads: Matrix dimension: 2800. Elapsed time = 11817.7 ms.
parallel-threads: Matrix dimension: 3000. Elapsed time = 14116 ms.
parallel-threads: Matrix dimension: 3200. Elapsed time = 17027.8 ms.
parallel-threads: Matrix dimension: 3400. Elapsed time = 20124.9 ms.
parallel-threads: Matrix dimension: 3600. Elapsed time = 22979.4 ms.
parallel-threads: Matrix dimension: 3800. Elapsed time = 28056.7 ms.
parallel-threads: Matrix dimension: 4000. Elapsed time = 31946.7 ms.
parallel-threads: Matrix dimension: 4200. Elapsed time = 38093.3 ms.
parallel-threads: Matrix dimension: 4400. Elapsed time = 39272.9 ms.
parallel-threads: Matrix dimension: 4600. Elapsed time = 46056.8 ms.
parallel-threads: Matrix dimension: 4800. Elapsed time = 57512.2 ms.
parallel-threads: Matrix dimension: 5000. Elapsed time = 64510.9 ms.
parallel-threads: Matrix dimension: 5200. Elapsed time = 69016.1 ms.
parallel-threads: Matrix dimension: 5400. Elapsed time = 78921 ms.
parallel-threads: Matrix dimension: 5600. Elapsed time = 86343.6 ms.
parallel-threads: Matrix dimension: 5800. Elapsed time = 92716.5 ms.
parallel-threads: Matrix dimension: 6000. Elapsed time = 107230 ms.
parallel-threads: Matrix dimension: 6200. Elapsed time = 115462 ms.
parallel-threads: Matrix dimension: 6400. Elapsed time = 126204 ms.
parallel-threads: Matrix dimension: 6600. Elapsed time = 141988 ms.
parallel-threads: Matrix dimension: 6800. Elapsed time = 157298 ms.
parallel-threads: Matrix dimension: 7000. Elapsed time = 171461 ms.
parallel-threads: Matrix dimension: 7200. Elapsed time = 185597 ms.
parallel-threads: Matrix dimension: 7400. Elapsed time = 198133 ms.
parallel-threads: Matrix dimension: 7600. Elapsed time = 219699 ms.
parallel-threads: Matrix dimension: 7800. Elapsed time = 231182 ms.

```

可继续改进，实验结果截图：

```

parallel-chunk-thread: Matrix dimension: 200. Elapsed time = 58.863 ms.
parallel-chunk-thread: Matrix dimension: 400. Elapsed time = 101.928 ms.
parallel-chunk-thread: Matrix dimension: 600. Elapsed time = 239.691 ms.
parallel-chunk-thread: Matrix dimension: 800. Elapsed time = 445.113 ms.
parallel-chunk-thread: Matrix dimension: 1000. Elapsed time = 667.215 ms.
parallel-chunk-thread: Matrix dimension: 1200. Elapsed time = 1092.59 ms.
parallel-chunk-thread: Matrix dimension: 1400. Elapsed time = 1657.93 ms.
parallel-chunk-thread: Matrix dimension: 1600. Elapsed time = 2217.36 ms.
parallel-chunk-thread: Matrix dimension: 1800. Elapsed time = 3265.47 ms.
parallel-chunk-thread: Matrix dimension: 2000. Elapsed time = 4318.04 ms.
parallel-chunk-thread: Matrix dimension: 2200. Elapsed time = 6725.15 ms.
parallel-chunk-thread: Matrix dimension: 2400. Elapsed time = 7913.6 ms.
parallel-chunk-thread: Matrix dimension: 2600. Elapsed time = 9674.8 ms.
parallel-chunk-thread: Matrix dimension: 2800. Elapsed time = 11146 ms.
parallel-chunk-thread: Matrix dimension: 3000. Elapsed time = 14229.1 ms.
parallel-chunk-thread: Matrix dimension: 3200. Elapsed time = 17642 ms.
parallel-chunk-thread: Matrix dimension: 3400. Elapsed time = 20906.4 ms.
parallel-chunk-thread: Matrix dimension: 3600. Elapsed time = 24487.6 ms.
parallel-chunk-thread: Matrix dimension: 3800. Elapsed time = 29610.2 ms.
parallel-chunk-thread: Matrix dimension: 4000. Elapsed time = 33834.9 ms.
parallel-chunk-thread: Matrix dimension: 4200. Elapsed time = 41342.2 ms.
parallel-chunk-thread: Matrix dimension: 4400. Elapsed time = 46830.8 ms.
parallel-chunk-thread: Matrix dimension: 4600. Elapsed time = 53275.4 ms.
parallel-chunk-thread: Matrix dimension: 4800. Elapsed time = 61333.9 ms.
parallel-chunk-thread: Matrix dimension: 5000. Elapsed time = 71874.2 ms.
parallel-chunk-thread: Matrix dimension: 5200. Elapsed time = 79215.8 ms.
parallel-chunk-thread: Matrix dimension: 5400. Elapsed time = 87471.6 ms.
parallel-chunk-thread: Matrix dimension: 5600. Elapsed time = 98474.7 ms.
parallel-chunk-thread: Matrix dimension: 5800. Elapsed time = 111730 ms.
parallel-chunk-thread: Matrix dimension: 6000. Elapsed time = 122946 ms.
parallel-chunk-thread: Matrix dimension: 6200. Elapsed time = 134217 ms.
parallel-chunk-thread: Matrix dimension: 6400. Elapsed time = 149000 ms.
parallel-chunk-thread: Matrix dimension: 6600. Elapsed time = 162192 ms.
parallel-chunk-thread: Matrix dimension: 6800. Elapsed time = 180646 ms.
parallel-chunk-thread: Matrix dimension: 7000. Elapsed time = 196601 ms.
parallel-chunk-thread: Matrix dimension: 7200. Elapsed time = 213021 ms.
parallel-chunk-thread: Matrix dimension: 7400. Elapsed time = 236967 ms.
parallel-chunk-thread: Matrix dimension: 7600. Elapsed time = 252178 ms.
parallel-chunk-thread: Matrix dimension: 7800. Elapsed time = 274949 ms.

```

但其性能不升反降，从理论上分析，可能是内存太小，导致前期申请结构体数据的开销不足以弥补不用释放的开销，后期内存耗尽的开销影响不用释放内存的开销。

OpenMP:

在计算三角形，即 *ForwardElimination* 时，加一条 *pragma* 语句，将内层原本循环一次

生成一个“0”的部分并行化，但外层循环 *BackSubstitution* 因为存在数据依赖，不能并行化。

github 地址 [OpenMP](#) 请点击[这里](#)

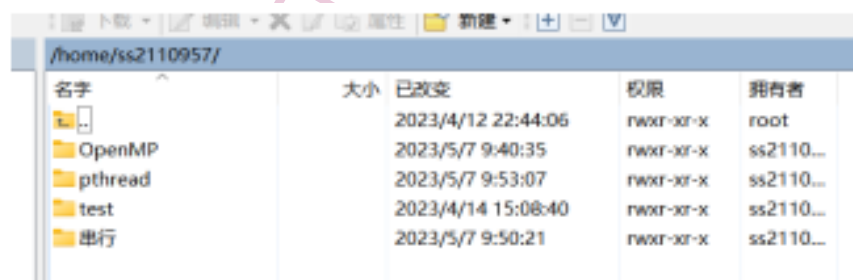
使用 `-fopenmp` 选项进行编译，实验结果截图如下：

```
parallel-openmp: Matrix dimension: 200. Elapsed time = 260.663 ms.
parallel-openmp: Matrix dimension: 400. Elapsed time = 99.42 ms.
parallel-openmp: Matrix dimension: 600. Elapsed time = 269.869 ms.
parallel-openmp: Matrix dimension: 800. Elapsed time = 531.745 ms.
parallel-openmp: Matrix dimension: 1000. Elapsed time = 959.613 ms.
parallel-openmp: Matrix dimension: 1200. Elapsed time = 1595.01 ms.
parallel-openmp: Matrix dimension: 1400. Elapsed time = 2004.03 ms.
parallel-openmp: Matrix dimension: 1600. Elapsed time = 3911.64 ms.
parallel-openmp: Matrix dimension: 1800. Elapsed time = 5286.26 ms.
parallel-openmp: Matrix dimension: 2000. Elapsed time = 6158.34 ms.
parallel-openmp: Matrix dimension: 2200. Elapsed time = 8205.18 ms.
parallel-openmp: Matrix dimension: 2400. Elapsed time = 10367.3 ms.
parallel-openmp: Matrix dimension: 2600. Elapsed time = 13586.3 ms.
parallel-openmp: Matrix dimension: 2800. Elapsed time = 14800.2 ms.
parallel-openmp: Matrix dimension: 3000. Elapsed time = 17151.8 ms.
parallel-openmp: Matrix dimension: 3200. Elapsed time = 20200.9 ms.
parallel-openmp: Matrix dimension: 3400. Elapsed time = 26477.5 ms.
parallel-openmp: Matrix dimension: 3600. Elapsed time = 29868.5 ms.
parallel-openmp: Matrix dimension: 3800. Elapsed time = 34162.9 ms.
parallel-openmp: Matrix dimension: 4000. Elapsed time = 38398.3 ms.
parallel-openmp: Matrix dimension: 4200. Elapsed time = 43832.1 ms.
parallel-openmp: Matrix dimension: 4400. Elapsed time = 50039.8 ms.
parallel-openmp: Matrix dimension: 4600. Elapsed time = 65387.5 ms.
parallel-openmp: Matrix dimension: 4800. Elapsed time = 68386.2 ms.
parallel-openmp: Matrix dimension: 5000. Elapsed time = 80138.9 ms.
parallel-openmp: Matrix dimension: 5200. Elapsed time = 88059.3 ms.
parallel-openmp: Matrix dimension: 5400. Elapsed time = 99289.1 ms.
parallel-openmp: Matrix dimension: 5600. Elapsed time = 106393 ms.
parallel-openmp: Matrix dimension: 5800. Elapsed time = 121666 ms.
parallel-openmp: Matrix dimension: 6000. Elapsed time = 134814 ms.
parallel-openmp: Matrix dimension: 6200. Elapsed time = 155779 ms.
parallel-openmp: Matrix dimension: 6400. Elapsed time = 161524 ms.
parallel-openmp: Matrix dimension: 6600. Elapsed time = 178107 ms.
parallel-openmp: Matrix dimension: 6800. Elapsed time = 200226 ms.
parallel-openmp: Matrix dimension: 7000. Elapsed time = 210357 ms.
parallel-openmp: Matrix dimension: 7200. Elapsed time = 220582 ms.
parallel-openmp: Matrix dimension: 7400. Elapsed time = 236068 ms.
parallel-openmp: Matrix dimension: 7600. Elapsed time = 256240 ms.
parallel-openmp: Matrix dimension: 7800. Elapsed time = 275508 ms.
```

#### 4. 串行 pthread1 实验对比

#### 5. 串行 OpenMP 实验对比

#### 6. pthread1OpenMP 实验对比



名字	大小	已改变	权限	所有者
OpenMP		2023/4/12 22:44:06	rwxr-xr-x	root
pthread		2023/5/7 9:40:35	rwxr-xr-x	ss2110...
test		2023/4/14 15:08:40	rwxr-xr-x	ss2110...
串行		2023/5/7 9:50:21	rwxr-xr-x	ss2110...

### 分析

#### pthread 部分

##### 同步开销

创建和销毁线程的次数过多，而线程创建、销毁的代价较大。通过信号量同步、barrier 同步等同步方式避免频繁的创建和销毁线程。

信号量同步，主线程执行除法，工作线程执行消去。主线程开始时建立多个工作线程，在每一轮消去过程中，工作线程先进入睡眠，主线程完成除法后将它们唤醒，自己进入睡眠；工作线程进行消去操作，完成后唤醒主线程，自己再进入睡眠；主线程被唤醒后进入下一轮消去过程，直至任务全部结束销毁工作线程；barrier 同步，直到所有线程都到达该点后才能继续执行。

矩阵规模 N	串行时间 (ms)	pthread1 时间 (ms)	加速比
200	12.868	6.081	2.116099326
400	63.572	56.487	1.125427089
600	208.144	150.043	1.387228994
800	561.739	302.29	1.858278474
1000	1004.53	562.468	1.785932711
1200	1517.96	883.419	1.718278642
1400	2400.12	1566.63	1.532027345
1600	3713.21	2061.79	1.800964211
1800	5227.6	2942.43	1.776626802
2000	7702.5	4238.44	1.817295986
2200	10360.7	5801.84	1.785761069
2400	12893	7447.86	1.731101283
2600	16075.1	9812.56	1.638216734
2800	20239.5	11817.7	1.7126429
3000	25334.9	14116	1.794764806
3200	29976.6	17027.8	1.760450557
3400	35193.2	20124.9	1.748739124
3600	42814.5	22979.4	1.863168751
3800	50294.4	28056.7	1.792598559
4000	57756.6	31946.7	1.807905042
4200	68833.7	38093.3	1.806976555
4400	76231.6	39272.9	1.94107387
4600	87870.6	46056.8	1.907874624
4800	101829	57512.2	1.770563463
5000	110317	64510.9	1.7100521
5200	120172	69016.1	1.741216904
5400	148310	78921	1.879220993
5600	173519	86343.6	2.009633603
5800	191267	92716.5	2.062922996
6000	206010	107230	1.921197426
6200	219028	115462	1.896970432
6400	241775	126104	1.917266701
6600	267036	141988	1.880694143
6800	290154	157298	1.844613409
7000	311076	171461	1.814266801
7200	344624	185597	1.856840358
7400	377756	198133	1.906577905
7600	418067	219699	1.90290807
7800	428291	231182	1.852613958

表 1: 串行 pthread1 实验对比

矩阵规模 N	串行时间 (ms)	OpenMP 时间 (ms)	加速比
200	12.868	260.663	0.049366423
400	63.572	99.42	0.639428686
600	208.144	269.869	0.771277916
800	561.739	531.745	1.056406736
1000	1004.53	959.613	1.046807411
1200	1517.96	1595.01	0.951693093
1400	2400.12	2004.03	1.197646742
1600	3713.21	3911.64	0.949271917
1800	5227.6	5286.26	0.988903308
2000	7702.5	6158.34	1.250742895
2200	10360.7	8205.18	1.262702342
2400	12893	10367.3	1.243621772
2600	16075.1	13586.3	1.183184531
2800	20239.5	14800.2	1.367515304
3000	25334.9	17151.8	1.477098614
3200	29976.6	20200.9	1.483923984
3400	35193.2	26477.5	1.329173827
3600	42814.5	29868.5	1.433433216
3800	50294.4	34162.9	1.47219352
4000	57756.6	38398.3	1.504144715
4200	68833.7	43832.1	1.570394756
4400	76231.6	50039.8	1.523419358
4600	87870.6	65387.5	1.343844007
4800	101829	68386.2	1.489028488
5000	110317	80138.9	1.376572426
5200	120172	88059.3	1.364671307
5400	148310	99289.1	1.493718847
5600	173519	106393	1.630924967
5800	191267	121666	1.572066148
6000	206010	134814	1.52810539
6200	219028	155779	1.406017499
6400	241775	161524	1.496836383
6600	267036	178107	1.499300982
6800	290154	200226	1.44913248
7000	311076	210357	1.478800325
7200	344624	220582	1.562339629
7400	377756	236068	1.600199942
7600	418067	256240	1.631544646
7800	428291	275508	1.55455014

表 2: 串行 OPenMP 对比



矩阵规模 N	pthread1 时间 (ms)	OpenMP 时间 (ms)	加速比
200	6.081	260.663	42.86515376
400	56.487	99.42	1.760050985
600	150.043	269.869	1.798611065
800	302.29	531.745	1.759055873
1000	562.468	959.613	1.706075723
1200	883.419	1595.01	1.805496599
1400	1566.63	2004.03	1.279198024
1600	2061.79	3911.64	1.897205826
1800	2942.43	5286.26	1.796562705
2000	4238.44	6158.34	1.452973264
2200	5801.84	8205.18	1.414237552
2400	7447.86	10367.3	1.391983738
2600	9812.56	13586.3	1.384582617
2800	11817.7	14800.2	1.252375674
3000	14116	17151.8	1.215060924
3200	17027.8	20200.9	1.186348207
3400	20124.9	26477.5	1.315658711
3600	22979.4	29868.5	1.299794599
3800	28056.7	34162.9	1.217637855
4000	31946.7	38398.3	1.201948871
4200	38093.3	43832.1	1.150651164
4400	39272.9	50039.8	1.27415597
4600	46056.8	65387.5	1.419714353
4800	57512.2	68386.2	1.189072927
5000	64510.9	80138.9	1.242253635
5200	69016.1	88059.3	1.275924024
5400	78921	99289.1	1.258082133
5600	86343.6	106393	1.232204819
5800	92716.5	121666	1.312236765
6000	107230	134814	1.257241444
6200	115462	155779	1.349179817
6400	126104	161524	1.280879274
6600	141988	178107	1.254380652
6800	157298	200226	1.272908746
7000	171461	210357	1.226850421
7200	185597	220582	1.188499814
7400	198133	236068	1.191462301
7600	219699	256240	1.166323015
7800	231182	275508	1.191736381

表 3: pthread1OpenMP 实验对比

线程切换，需要切换上下文，从用户态转换到内核态，导致一定的时间和计算资源的消耗；等待开销，需要阻塞等待其他线程完成；锁开销，一个线程访问共享资源时，它需要先获得锁，而其他线程则需要等待该线程释放锁之后才能访问共享资源，增加额外的时间和计算资源的消耗。

#### 空闲等待

```

1
2 while(true){
3     pthread_mutex_lock(&mutex);
4     while(tasks.empty()){
5         //等待任务可用
6         pthread_cond_wait(&cond);
7     }
8     //取出一个任务
9     int task = tasks.front();
10    tasks.pop();
11    pthread_mutex_unlock(&mutex);
12
13    //处理任务
14    do_task(task);
15 }
16 return NULL;

```

当任务队列中有任务时，线程会立即取出任务进行处理；当任务队列为空时，线程会等待在条件变量上，直到有新的任务被添加到队列中再开始工作。

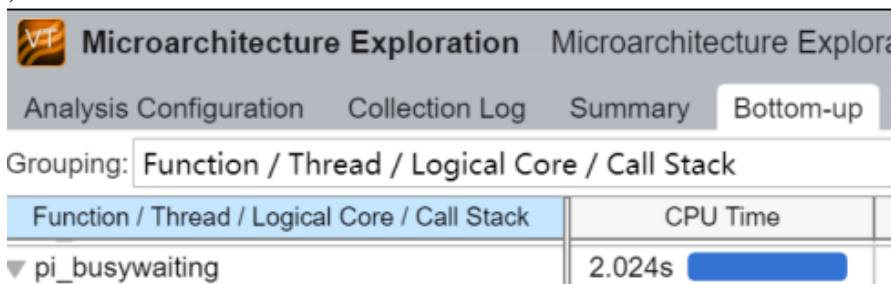
**Cache 命中：**将每个线程需要操作的行数存储在一个局部数组中，提高数据的局部性，增加 cache 命中率。

定义局部数据结构 `struct localdata double *row; double *pivot;`

使用数据对齐技术，使得每个线程需要访问的数据尽可能地在同一个 cache 行中，提高 cache 命中率。定义数据类型并进行对齐操作

`struct alignas(CACHELINE_SIZE) gauss_data double *rowptr; double pivot;`

**忙等待：**线程循环测试条件，直到满足条件，关闭编译器优化选项，每个线程的 id 及其运行信息，忙等待如下图所示：



根据线程 id 找到各线程的运行时间以及对应事件状态：`Sleep_Ex` 函数，该函数用于中止当前线程等待恢复运行：可以看出线程执行过该函数，结合其它线程，可分析线程的空闲等待。在线程数多的时候或是线程执行能力差异巨大的情况下尤为明显。

**互斥量：**线程使用忙等待会持续消耗 CPU 计算资源；互斥量是一种特殊的变量，使得同一时间只有一个线程可以访问临界区。当一个线程在使用临界区时，保证其它线程无法访问；Pthreads 的互斥量：`pthread_mutex_t`



使用 `pthread_mutex_t` 前, 必须由系统 `int pthread_mutex_init( pthread_mutex_t* mutex_p, const pthread_mutexattr_t* attr_p);`

当一个线程使用完互斥量后, 应该调用: `int pthread_mutex_destroy(pthread_mutex_t* mutex_p);`

要获得临界区的访问权, 线程需要调用: `int pthread_mutex_lock(pthread_mutex_t* mutex_p);`

当线程退出临界区后, 它应该调用 `int pthread_mutex_unlock(pthread_mutex_t* mutex_p);` `pthread_mutex_lock` 使线程等待, 直到没有其他线程进入临界区;

调用 `unlock` 则通知系统该线程已经完成了临界区中代码的执行。

### OpenMP 部分:

两种线程互斥机制: 互斥锁与事件同步机制。

#### critical 临界区

临界区用在可能产生数据访问竞争的地方, 用法: `pragma omp critical(name)(name )`, 保证每次只有一个线程进入。

注意: `critical` 语句不允许互相嵌套。例一个并行域内的 `for` 任务分担域中, 各个线程逐个进入到 `critical` 保护的区域内, 比较当前元素的最大值得关系并可能进行最大值的更替, 避免了数据竞争的情况:

```
pragma omp parallel for
for (i = 0; i < 8; i++)
pragma omp critical
if (arx[i] > max_x)
max_x = arx[i];
```

#### 原子操作

原子操作的功能是通过 `pragma omp atomic` 编译制导指令提供的。critical 临界区操作能够作用在任意大小的代码块上, 而原子操作只能作用在单条赋值语句中, C++ 中可用的原子操作如下: `+ - * / | « »`

`atomic` 在使用中需要注意:

当对一个数据进行原子操作的时候, 就不能对数据进行临界区的保护用户在针对同一个内存单元使用原子操作的时候, 需要在程序所有涉及到该变量并行赋值的部位都加入原子操作的保护。

#### 一致性问题

处理内存一致性问题可以用 `flush` 指令, 每个处理器 (processor) 都有自己的本地 (local) 存储单元: 寄存器和缓存, 当一个线程更新了共享变量之后, 新的值会首先存储到寄存器中, 然后更新到本地缓存中。这些更新并非立刻就可以被其他线程得知, 因此在其它处理器中运行的线程不能访问这些存储单元。如果一个线程不知道这些更新而使用共享变量的旧值就行运算, 就可能会得到错误的结果。通过使用 `flush` 指令, 可以保证线程读取到的共享变量的最新值。

语法形式: `pragma omp flush([list])`

结果正确性

**计算误差:** 问题规模和并行计算重排了指令执行顺序和计算机浮点数所导致误差。

**程序正确性:** 一次结果正确并不代表算法的正确性, 需要多进行实验降低错误概率, 不仅是同阶矩阵数据的多次实验, 还需要设计多组数据进行对比测试。

**负载均衡:** 计算量大小不一的任务, 不便于手动分配给每个线程, 考虑构建任务池, 即任务队列, 把待完成的任务放进去。每当一个线程完成当前任务后, 立刻到任务池中再选取一个任务执行, 直到所有任务被完成。在这一过程中, 一定要正确使用信号量、锁等方法来避免访问冲突。

## 7. pthreadOpenMP 比较

pthread 在程序启动时创建一束线程，将工作分配到线程上。然而，这种方法需要相当多的线程指定代码，而且不能保证能够随着可用处理器的数量而合理地进行扩充。

OpenMP，不需要指定数量，在有循环的地方加上代码，修改设置文件极客。OpenMP 非常方便，因为它不会将软件锁定在事先设定的线程数量中。

openmp 和 pthread 之间的区别主要在编译的方式上，openmp 的编译需要添加编译器预处理指令 pragma，创建线程等后续工作要编译器来完成。

而 pthread 就是一个库，所有的并行线程创建都需要我们自己完成。

pthread 全称是 POSIX THREAD，按照 POSIX 对线程的标准而设计的，两个版本：Linux Thread 和 NPTL。pthread 库是一套关于线程的 API，提供“遵循”（各平台实现各异）POSIX 标准的线程相关的功能。

openMP 不同于 pthread 的地方是，它是根植于编译器的（也要包含头文件 omp.h），而不是在各系统平台是做文章。貌似更偏向于将原来串行化的程序，通过加入一些适当的编译器指令（compiler directive）变成并行执行，从而提高代码运行的速率。

## 二、特殊高斯消去 pthread/OpenMP 并行化实验

### （一）设计分析

#### 1. 简介

高斯消元模块分两种行：消元子、被消元行（被消元多项式）。

其中消元子为根据系统中存储的多项式和其倍数多项式逐步导入，保证消元子首项各异但不能保证以每一项为首项的消元子都存在。其在高斯消元模块只充当“减数”而不会充当被减数，在高斯消元模块可以认为消元子已知。

被消元行在高斯消元过程中既充当“被减数”又充当“减数”，导入高斯消元模块的被消元行通常不是首项各异的。

高斯消元时，可视为消元子和被消元行合并为一个矩阵，该矩阵经过高斯消元之后非零行首项各异，消元结束后返回所有被消元行的消元结果。

#### 2. 算法思想

---

##### Algorithm 1 Grobe 基特殊高斯消元思想

---

```

1: for i := 0 to m - 1 do
2:   while E[i] != 0 do
3:     if R[lp(E[i])] != NULL
       then E[i] := E[i] - R[lp(E[i])]
        else
          R[lp(T[i])] := E[i][par] break
        end if
      end while
    end for
  return E

```

---

4:

### 3. 实现

将系数矩阵和右端向量分成多个块，每个块分配给一个线程；  
 在每个线程中，使用消元子和消元行的方式对分配给该线程的块进行消元；  
 在消元过程中，需要使用互斥锁来保护线程之间的共享变量，如主元和右端向量；  
 在回代过程中，同样需要使用互斥锁来保护线程之间的共享变量；  
 将计算结果合并得到最终解。

线程执行的任务函数

```

1  for i in range(n-1):
2      # 找到当前列绝对值最大的行，作为主元所在行
3      p = np.argmax(np.abs(Ab[i:, i])) + i
4      if Ab[p, i] == 0:
5          raise ValueError('The matrix is singular.')
6      # 交换主元所在行和当前行
7      Ab[[i, p]] = Ab[[p, i]]
8      # 计算消元子
9      m = Ab[i+1:, i:i+1] / Ab[i, i]
10     # 消元
11     Ab[i+1:] -= m * Ab[i:i+1, i+1:]
12 if Ab[n-1, n-1] == 0:
13     raise ValueError('The matrix is singular.')
14 # 回代求解
15 x = np.zeros(n)
16 x[n-1] = Ab[n-1, n] / Ab[n-1, n-1]
17 for i in range(n-2, -1, -1):
18     x[i] = (Ab[i, n] - np.dot(Ab[i, i+1:n], x[i+1:])) / Ab[i, i]
```

**pthread 部分**两个线程函数 `eliminate` 和 `back_substitute`，分别用于计算消元子和回代。`eliminate` 函数计算从  $k+1$  行到  $n-1$  行的消元子，然后更新矩阵  $A$  和向量  $B$ 。`back_substitute` 函数计算从  $(k+1)\text{num\_threads}-1$  行到  $\text{knum\_threads}-1$  行的未知数的值，然后更新向量  $x$ 。我们使用了 `pthread_barrier_t` 类型的 `barrier` 来同步线程的执行，确保消元子的计算完成后才进行回代。

**特殊高斯消元 + pthread** [github 地址 请点击这里](#)

#### OpenMP 部分

使用 OpenMP 中的 `pragma omp parallel for` 指令来并行化前向消元的循环。由于每次迭代中的计算是相互独立的，因此可以安全地并行化。使用消元子和消元行的技巧来减少计算量。在前向消元的过程中，我们将第  $k$  行以下每一行的第  $k$  列元素消为 0，并将消元系数存储在  $A$  数组中。

**特殊高斯消去 + OpenMP** [github 地址 请点击这里](#)

### 4. 分析

利用多个线程同时对不同的块进行计算，提高算法的计算效率。注意，实现过程中需要合理地设计线程的数量和块的大小，以便充分利用多核处理器的计算能力，避免过多的线程切换和锁竞争导致的性能损失。

随着矩阵规模的增加，使用 `pthread` 库并行化的算法的执行时间依然会逐渐增加，但增长速度会比串行情况下慢很多，并行化充分利用计算机的多核心处理能力，也减少了线程之间的同步操作。

### 三、不同平台及算法策略的比较

#### (一) ARM 与 X86 平台上比较

		Runtime (ms)	Cycles
400.perlbench	A12	217,019	541,213,307,212
	9900K	154,349	763,467,113,335
401.bzip2	A12	372,954	937,315,759,910
	9900K	254,523	1,260,402,131,339
403.gcc	A12	180,894	453,706,513,972
	9900K	119,148	589,521,733,262
429.mcf	A12 (Modified test)	115,758	290,867,619,932
	9900K	131,306	648,610,259,027
445.gobmk	A12	271,512	682,186,292,636
	9900K	239,435	1,184,327,814,901
456.hmmer	A12	208,570	498,341,625,944
	9900K	177,927	879,778,219,764
458.sjeng	A12	331,976	831,277,665,980
	9900K	251,560	1,243,406,921,178
462.libquantum	A12	182,229	414,483,072,624
	9900K	141,944	701,067,252,338
464.h264ref	A12	330,635	806,753,437,791
	9900K	251,796	1,246,607,303,564
471.omnetpp	A12	174,568	438,708,615,810
	9900K	157,770	779,498,936,960
473.astar	A12	260,617	648,134,176,855
	9900K	207,607	1,028,084,546,269
483.xalancbmk	A12	120,078	301,895,782,626
	9900K	98,729	488,373,655,922
433.milc	A12	219,456	548,678,476,220
	9900K	197,170	974,842,633,568

instruction 是运行该测试 cpu 执行的指令数量，可以看到，A12 的指令数量大约是 9900k 的 110%，也就是 ARM V8.4A 需要多 10% 的指令来达到和 X86 一样的效果。总体上，X86 还是要比 ARM 强 10%，同一段代码，ARM 要 110% 的 IPC（每时钟周期指令）才能达到和 X86 相同的 PPC（每时钟周期性能）

**功耗高一定比功耗低的性能强?:** 在 Spec2006 测试中, 9900k 分别以 2.7% 和 6.9% 的巨大性能优势打败了 A13, 并以 6.41 倍和 6.02 倍的功耗再次击败 A13。

图中 A12 的 IPC 达 9900K 的 170%, A13 还要比 A12 高上 15%, 综合下来 A13 拥有 9900K196% 的 IPC, 除去 X86 比 ARM 高 9.84% 的指令效率, 两者 PPC 相差 78.61%。才让 A13 可以以低频率打 9900K 高频率。

结论是, A13 利用强大的架构, 低的频率, 以及更优秀的制程, 才产生了 6 倍于 9900K 的能耗比。

关于 ICC:

Results: Vectorisation											
Application	% time	# Loops	Loops Vectorised SVE			Loops Vectorised NEON			Loops Vectorised AVX*		
			Arm	Cray	GCC	Arm	Cray	GCC	Intel	Cray	GCC
BUDE	98.6	4	4	3	3	3	4	3	4	4	3
TeaLeaf (cg)	87.2	8	5	6	8	5	6	8	8	6	6
TeaLeaf (ppcg)	91.2	6	6	6	6	6	6	6	6	6	6
CloverLeaf	62.5	10	9	10	6	8	9	6	10	9	8
MegaSweep	70.3	4	1	4	0	1	1	0	4	1	0

不同编译器和不同指令集在同一段代码下的优化程度。图中看出, 在 gcc 下, 无论是 x86 的 avx 还是 arm 的 neon、sve, 都有着同样的优化程度。而 arm 自己的编译器在 neon 下比 gcc 还差, sve 下强不少, 但也不如 ICC。如果使用 ICC 作为编译器的分数作为标准对比, 那就意味着 ICC 本身优化比 llvm 好。

**线程创建和销毁方式:** 在 ARM 和 X86 架构下, 线程的创建和销毁方式基本相同, 都是通过调用 pthread\_create() 函数和 pthread\_join() 函数来实现。但是, ARM 架构下的线程创建和销毁速度可能会更快, 因为 ARM 架构下的处理器通常具有更高的性能和更低的功耗。

**线程同步方式:** 在 ARM 和 X86 架构下, 线程同步方式也基本相同, 都是通过互斥锁、条件变量等机制来实现。但是, 在 ARM 架构下, 处理器的结构和特性不同, 可能会导致线程同步的性能和效率有所差异。

**浮点数运算的性能:** 在 ARM 和 X86 架构下, 浮点数运算的性能一般而言, 在相同的时钟速度下, X86 架构的处理器通常比 ARM 架构的处理器具有更高的浮点数计算性能, 影响高斯消去法算法的执行速度。

**内存访问的速度:** 在 ARM 和 X86 架构下, 一般而言, ARM 架构的处理器通常具有更高的内存访问速度和更低的功耗。

## (二) 多线程并行化不同算法策略

**手动分块**将矩阵分成多个小块, 每个线程处理一个小块, 可以充分利用多核处理器的并行性能, 提高算法的效率和性能, 但是手动分块需要手动调整块的大小和数量, 需要一定的经验。

**SIMD** 基于指令集的算法策略, 利用处理器的 SIMD 指令集来实现高效的并行化计算, 充分利用处理器的硬件并行性能。

### 一致性保证

**互斥锁:** 通过加锁和解锁来保证多个线程对共享数据的互斥访问, 避免多个线程同时访问共享数据而导致的数据竞争和错误, 但互斥锁会造成线程之间的等待和切换, 降低算法的效率。

**读写锁:** 分别对读操作和写操作进行加锁和解锁, 实现多个线程同时对共享数据进行读取, 而只有一个线程对共享数据进行写入。

**原子操作:** 保证对共享数据的操作是不可分割的, 保证多个线程对共享数据的一致性访问, 避免互斥锁和读写锁的等待和切换。

### 线程管理代价优化



**线程池：**预先创建一定数量的线程，并将任务分配给空闲的线程进行处理，避免频繁地创建和销毁线程，减少线程管理的代价，也可以通过控制线程池的大小和任务队列的长度来平衡线程的使用和任务的处理。

**线程绑定：**将线程和处理器核心进行绑定，使得线程可以在特定的处理器核心上运行，避免线程切换和处理器缓存的失效。

#### Cache 优化

**数据局部性优化：**将相关的数据放置在相邻的内存位置，对矩阵进行分块重排。

**数据预取优化：**预先将数据加载到 cache 中，以避免 cache 的缺失和延迟，在计算前预先加载相关的数据块。

**循环展开优化：**通过展开矩阵计算的循环，减少内部循环的指令和分支。

NIJUB