



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

OpenMP 编程实验

姓名：韩佳迅

学号：2012682

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2022 年 5 月 19 日

摘要

本实验基于两种高斯消去算法，进行 OpenMP 并行实验。分别对普通高斯消去和特殊高斯消去两种算法，实现了串行算法和 OpenMP 等的并行优化。

关键字：OpenMP，普通高斯消去，特殊高斯消去

目录

一、普通高斯消去	1
(一) 问题简介	1
(二) 实验设计	1
1. 总体思路	1
2. 设计思路	1
(三) 理论分析	2
(四) 实验内容及算法实现	3
(五) 实验结果及分析	7
1. 基于问题规模和线程数	7
2. 基于数据划分	7
3. 基于循环调度	8
4. 其他算法策略	9
5. OpenMP 自动向量化	10
6. 基于不同平台架构	10
7. 与 pthread 的对比	10
二、特殊高斯消去	11
(一) 问题简介	11
(二) 实验设计	11
(三) 理论分析	12
(四) 实验内容及算法实现	12
(五) 实验结果及分析	13
三、源代码	13

一、普通高斯消去

(一) 问题简介

给定一个满秩矩阵，从上到下依次对它的每行进行除法（除以每行的对角线元素），然后对矩阵该行右下角 $(n-k+1) \times (n-k)$ 的子矩阵进行消去，最后得到一个三角矩阵。

(二) 实验设计

1. 总体思路

在设计之前，首先从总体上理清普通高斯消去程序的依赖关系，并据此分别找出串并行的部分：

分析普通高斯程序的执行过程，如图1，程序由外层一个大循环和它内部的一个二重循环和一个三重循环构成。最外层大循环控制对矩阵每行的操作（假设某次循环到第 k 行），其中的二重循环表示对矩阵的第 k 行进行除法，三重循环表示用第 k 行对其右下角的 $(n-k+1) \times (n-k)$ 的子矩阵进行消去。

在每轮大循环中存在着先后关系：需要首先对该行进行除法，然后再对它右下角子矩阵消去。消去依赖于除法，而除法和消去内部没有必然的顺序关系。

因此，OpenMP 程序的整体思路为：取一个线程（或多个线程并行）先对矩阵第 k 行进行除法，除法结束后所有线程才可以进入并行消去环节。当所有线程完成消去后，一起进入下一轮，重复上述步骤。

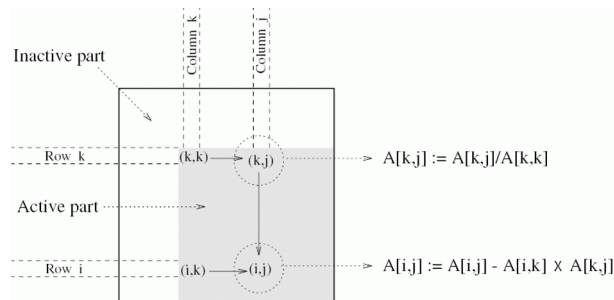


图 1: 总体思路

在《Pthread》实验中，我们已经编写了该算法的 pthread 多线程程序。而对于和它对应的 OpenMP 程序而言，本次实验将不再侧重信息交互、同步机制、多线程的具体实现等内容，转而将侧重对于任务划分、循环调度、负载均衡、体系结构、编程范式等的探究。

2. 设计思路

• 数据划分

设计不同粗细粒度的数据划分，研究数据划分的不同对体系结构、访存代价以及负载均衡的影响。

• 循环调度

OpenMP 为我们提供了不同的循环调度方法，例如静态划分、动态划分等。设计实现不同的划分方式，探究其对负载和访存的影响。

• 体系结构 / 访存

结合 cache 大小，设计不同问题规模 and 不同线程数，充分利用 cache。同时，在数据划分和循环调度的时候，也设计考虑 cache 等对其的影响。

改变算法，使得算法能够更加适应体系结构。

- 不同平台

OpenMP 支持多种操作系统，实验设计在 Linux 和 Windows 不同平台，结合不同 SIMD 指令集等进行对比。

- 与 SIMD 的结合

在编程时，手动将 OpenMP 程序与 SIMD 结合。并且尝试了 openmp 与 SIMD 自动结合的编译指令。

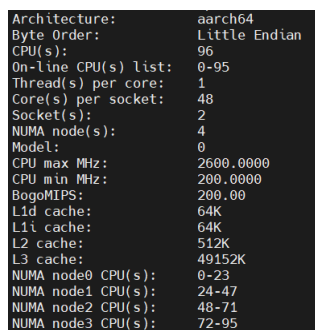
- 其他策略与编程范式

设计将除法也进行并行操作。以及使用 OpenMP 的多种范式。

(三) 理论分析

在设计多线程程序时，可以通过充分适应 CPU 的体系结构，来达到更好的实验效果。根据体系结构和组成原理的知识，服务器的一个节点可以有多个 CPU，一个 CPU 可以有多个 CPU 核心，一个 CPU 核心可以有一个以上的线程。一个核心在某个时间点只能执行一个进程，一个程序可以调用多个进程，一个进程可调用至少一个线程，如果一个进程同时调用的线程数超过 CPU 核心的线程数，则需要调用其他 CPU 核心实现并行。一个进程只能在本节点运行，线程是进程派生的并共享进程资源，所以多线程并行不能跨节点运行。分析实验使用的华为鲲鹏服务器，如图2，鲲鹏服务器有两个物理 CPU，每个物理 CPU 有 48 个核心，共有 96 颗逻辑 CPU。服务器有 4 个 node。鲲鹏每个 CPU 核心有各自的 L1 cache (64K)，L2 cache (512K)，共享 L3 cache (49152K)。

在鲲鹏服务器上运行 OpenMP 程序时，总共最多开启 8 个线程。



Architecture:	aarch64
Byte Order:	Little Endian
CPU(s):	96
On-line CPU(s) list:	0-95
Thread(s) per core:	1
Core(s) per socket:	48
Socket(s):	2
NUMA node(s):	4
Model:	0
CPU max MHz:	2600.0000
CPU min MHz:	200.0000
BogoMIPS:	200.00
L1d cache:	64K
L1i cache:	64K
L2 cache:	512K
L3 cache:	49152K
NUMA node0 CPU(s):	0-23
NUMA node1 CPU(s):	24-47
NUMA node2 CPU(s):	48-71
NUMA node3 CPU(s):	72-95

图 2: 鲲鹏服务器

对于 64k 大小的 L1 cache 来说，我们进行如下分析：

每个 float 类型的变量占 4 字节，L1 cache 的大小为 64k，当问题规模 n 足够大时，可忽略其他变量的空间，设 float 型变量总数为 N ，有 n^2 约等于 N 。因此，计算得出，占满 L1 cache 的 n 在 120 附近，占满 L2 cache 的 n 在 350 附近。

因此，当我们设计的问题规模较大时，如果能够使得每次分配给每个线程的数据量，不过小也不超过其 L1 cache 大小，就可能达到更好的命中率和更快的速度。

以处理的矩阵规模为 $\theta \times \theta$ 为例，将 $\theta \times \theta$ 的矩阵按行划分成若干任务，假设每个任务 h 行， $h \times \theta$ 接近 float 型变量总数为 N 即可。

但是，普通高斯每轮处理的消元矩阵大小为 $(n-k) \times (n-k)$ ，随着轮数 k 的增大而递减，因此，我们取其运算矩阵最大的规模 $n \times n$ 来计算任务行数，以提高整体命中率。

综上，我们得出结论：当每次划分的数据量过大，线程读取的数据会超出其 L1 cache 容量大小，降低命中率，速度减慢；当划分的数据量过小，空间读取不连续，由于空间局部性也会降低性能。因此，我们分析，当划分的数据大小接近 cache 容量可能会更好。

- 静态划分

openMP 默认的调度方式为静态划分，且默认的划分块大小为 $\text{ceil}(\text{iterations} / \text{threads})$ 。这种情况下，每个线程分配到的数据量不均衡，容易造成负载不均，时间浪费。如果减小划分粒度，极限情况下，每个线程负责的数据块只有一行/一个元素，会导致一个线程负责计算的数据几乎没有连续的，缺乏局部性，高度交互，通信量大。因此，高维块循环分配可能达到更好的效果。此时每个线程分配的数据量大小可以参考上一小节数据划分给出的计算结果。

- 动态划分

逻辑上相当于有一个任务池，包含所有迭代步。先分配给每个线程一个迭代步，一个线程完成任务后再为其分配迭代步。动态划分的好处是会让线程有空闲等待的时间。

- guided 动态划分

划分方式与普通的动态划分相同，但划分过程中，最初的组中的循环体执行数目较大，而后指数减小。

(四) 实验内容及算法实现

首先，结合 SIMD，我们在不同线程数、不同问题规模下采用不同的数据划分和循环调度方法进行了 OpenMP 实验，并且尝试了其他策略和平台。

限于篇幅,报告中只展示部分代码(可以在https://github.com/hanmaxmax/parallel_homework4中详见代码)。

结合 neon 的静态数据划分代码如下：

普通高斯消去 OpenMP 优化

```

1 void f_omp_static_neon()
2 {
3     float32x4_t va = vmovq_n_f32(0);
4     float32x4_t vx = vmovq_n_f32(0);
5     float32x4_t vaij = vmovq_n_f32(0);
6     float32x4_t vaik = vmovq_n_f32(0);
7     float32x4_t vakj = vmovq_n_f32(0);
8     #pragma omp parallel num_threads(NUM_THREADS), private(va, vx, vaij, vaik, vakj)
9         for (int k = 0; k < n; k++){
10         #pragma omp single //串行部分
11         {
12             float32x4_t vt=vmovq_n_f32(A[k][k]); int j;
13             for (j = k + 1; j < n; j++){

```

```

14         va=vld1q_f32(&(A[k][j]) );
15         va= vdivq_f32(va,vt);
16         vst1q_f32(&(A[k][j]) , va);
17     }
18     for(; j<n; j++)
19         A[k][j]=A[k][j]*1.0 / A[k][k];
20     A[k][k] = 1.0;
21 }
22 #pragma omp for schedule(static) //并行部分
23 for (int i = k + 1; i < n; i++){
24     vaik=vmovq_n_f32(A[i][k]); int j;
25     for (j = k + 1; j+4 <= n; j+=4){
26         vakj=vld1q_f32(&(A[k][j]) );
27         vaij=vld1q_f32(&(A[i][j]) );
28         vx=vmulq_f32(vakj,vaik);
29         vaij=vsubq_f32(vaij,vx);
30         vst1q_f32(&A[i][j] , vaij);
31     }
32     for(; j<n; j++)
33         A[i][j] = A[i][j] - A[i][k] * A[k][j];
34         A[i][k] = 0;
35     }
36 }
37 }

```

动态 dynamic 划分以及 guided 划分在此基础上，替换 static 和 [chunk] 即可。

此外，结合 pthread 编程的动态线程和静态线程区别，我们发现，将 *pragma omp parallel...* 语句放到整个大循环的外部，而在大循环内部再划分 *pragma omp single* 串行部分和 *pragma omp for* 并行部分，可以减少创建、销毁循环的开销。我们利用如下代码，探究若只在大循环内部的并行部分声明 *pragma omp parallel...* 而增加的创建、销毁线程开销。

普通高斯消去 OpenMP 优化（“动态线程”版）

```

1 void f_omp_static_neon_dynamicThreads()
2 {
3     float32x4_t va = vmovq_n_f32(0);
4     float32x4_t vx = vmovq_n_f32(0);
5     float32x4_t vaij = vmovq_n_f32(0);
6     float32x4_t vaik = vmovq_n_f32(0);
7     float32x4_t vakj = vmovq_n_f32(0);
8     for (int k = 0; k < n; k++){
9         //串行部分
10        {
11            float32x4_t vt=vmovq_n_f32(A[k][k]); int j;
12            for (j = k + 1; j < n; j++){
13                va=vld1q_f32(&(A[k][j]) );
14                va= vdivq_f32(va,vt);
15                vst1q_f32(&(A[k][j]) , va);
16            }

```

```

17         for (; j<n; j++)
18             A[k][j]=A[k][j]*1.0 / A[k][k];
19         A[k][k] = 1.0;
20     }
21     //并行部分
22     #pragma omp parallel for num_threads(NUM_THREADS), private(va, vx,
23         vaij, vaik, vakj), schedule(static)
24         for (int i = k + 1; i < n; i++){
25             vaik=vmovq_n_f32(A[i][k]);    int j;
26             for (j = k + 1; j+4 <= n; j+=4){
27                 vakj=vld1q_f32(&(A[k][j]));
28                 vaij=vld1q_f32(&(A[i][j]));
29                 vx=vmulq_f32(vakj, vaik);
30                 vaij=vsubq_f32(vaij, vx);
31                 vst1q_f32(&(A[i][j]), vaij);
32             }
33             for (; j<n; j++)
34                 A[i][j] = A[i][j] - A[i][k] * A[k][j];
35             A[i][k] = 0;
36         }
37     }

```

并且，我们尝试使用其他编程范式。使用 barrier 和 master 的代码如下：

普通高斯消去 OpenMP 优化（其他范式）

```

1 void f_omp_static_neon_barrier()
2 {
3     float32x4_t va = vmovq_n_f32(0);
4     float32x4_t vx = vmovq_n_f32(0);
5     float32x4_t vaij = vmovq_n_f32(0);
6     float32x4_t vaik = vmovq_n_f32(0);
7     float32x4_t vakj = vmovq_n_f32(0);
8     #pragma omp parallel num_threads(NUM_THREADS), private(va, vx, vaij, vaik
9         , vakj)
10         for (int k = 0; k < n; k++){
11             #pragma omp master //串行部分
12             {
13                 float32x4_t vt=vmovq_n_f32(A[k][k]);    int j;
14                 for (j = k + 1; j < n; j++){
15                     va=vld1q_f32(&(A[k][j])) );
16                     va= vdivq_f32(va, vt);
17                     vst1q_f32(&(A[k][j]), va);
18                 }
19                 for (; j<n; j++)
20                     A[k][j]=A[k][j]*1.0 / A[k][k];
21                 A[k][k] = 1.0;
22             }

```

```

23         //并行部分
24         #pragma omp barrier
25         #pragma omp for schedule(static)
26         for (int i = k + 1; i < n; i++){
27             vaik=vmovq_n_f32(A[i][k]);
28             int j;
29             for (j = k + 1; j+4 <= n; j+=4){
30                 vakj=vld1q_f32(&(A[k][j]));
31                 vaij=vld1q_f32(&(A[i][j]));
32                 vx=vmulq_f32(vakj, vaik);
33                 vaij=vsubq_f32(vaij, vx);
34                 vst1q_f32(&A[i][j], vaij);
35             }
36             for (; j<n; j++)
37                 A[i][j] = A[i][j] - A[i][k] * A[k][j];
38             A[i][k] = 0;
39         }
40     }
41 }

```

此外，我们还尝试了将除法也进行并行化，代码如下（由于与之前的代码大部分重合，故将重合部分用省略号替代）：

普通高斯消去 OpenMP 优化（除法优化）

```

1 void f_omp_static_neon_division()
2 {
3     float32x4_t va = vmovq_n_f32(0);
4     ...
5     #pragma omp parallel num_threads(NUM_THREADS), private(va, vx, vaij, vaik
6         , vakj)
7         for (int k = 0; k < n; k++)
8         {
9             //除法并行部分
10            #pragma omp for schedule(static)
11            {
12                ...
13            }
14            //消去并行部分
15            #pragma omp for schedule(static)
16            for (int i = k + 1; i < n; i++)
17            {
18                ...
19            }
20        }
21 }

```

由于 OpenMP 支持自动向量化，我们考虑去掉手写的 neon 代码，使用 omp 自动向量化的编程范式，测试其结果。openmp 自动向量化需要在循环前添加 `pragma omp for simd` 语句，由

于更改很少，代码不再赘述。

(五) 实验结果及分析

1. 基于问题规模和线程数

基于 OpenMP 静态块划分算法，结合 neon，在鲲鹏服务器上，我们进行了不同问题规模和线程数的实验，结果如图3。纯串行算法的结果如图4。

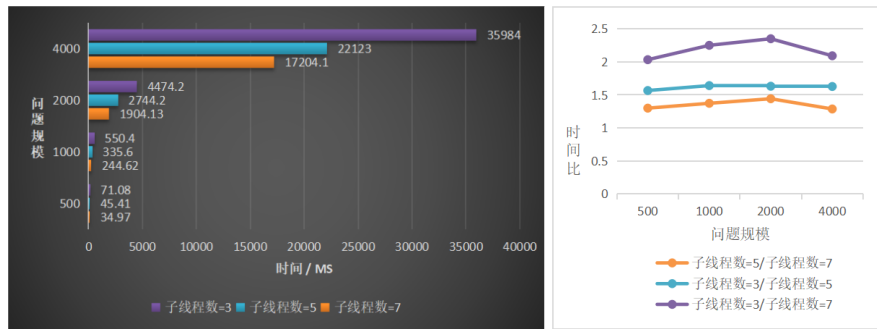


图 3: OpenMP 实验结果

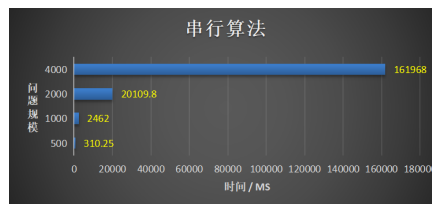


图 4: 串行算法结果

图3的左侧展示了不同规模和不同线程数下，OpenMP 算法运行的具体时间，右图展示了各规模下，不同线程数的运行时间比。分析数据我们可以发现，在各个规模下，不同线程数的时间比都近似等于线程个数之比。

再与串行算法对比，结合了 neon 的 OpenMP 多线程算法比纯串行算法快了 8 倍以上，达到了很好的加速效果，且加速效果在各个问题规模下都很明显。

2. 基于数据划分

在理论分析一节中，我们结合体系结构，对数据划分的粗细粒度进行了大致计算。下面我们问题规模 1000 为例，进行了实验。

根据我们之前的计算，问题规模为 1000 时，在最大的 $n \times n$ (1000 x 1000) 的矩阵中，能够填满 L1 cache 的划分粒度是 14 行，而随着最外层大循环的迭代，每次要处理的矩阵的大小会递减，能够填满 L1 cache 的划分行数也会增大。由于 schedule 编程范式只能设置 [chunk] 为常数值，我们不能更改大循环每次到达并行小循环的 [chunk] (即使使用 guided 范式，也只是针对并行小循环自己内部的划分粒度改变，不能改变大循环每次到达并行小循环的 [chunk])，因此我们取填满 $n \times n$ 的划分行数来实验。

因此，下面我们以问题规模为 1000，[chunk] 分别为 14、7、30 进行实验，测试结果如表1所示：

chunk	14	7	30
时间/ms	289.75	299.44	311.68
cache 命中率	86.56%	84.21%	83.63%

表 1: 基于数据划分的结果

可以看出, $\text{chunk} = 14$ 的划分方式, 要快于 $\text{chunk} = 7$ 和 30 的, 为了进一步分析其产生原因, 我们使用 `perf` 分析工具, 查看它们的 `cache` 命中率。在 `perf` 的结果中, $\text{chunk}=14$ 的 `L1-dcache` 命中率 $>$ $\text{chunk}=30$ 和 7 的 `L1-dcache` 命中率, 它们都达到了 80% 以上, 彼此相差 5% 以下。这说明我们之前对体系结构和粒度划分的分析是较合理的。

3. 基于循环调度

在问题规模为 1000 , 线程数 $=7$ 的情况下, 我们在消去的循环上分别采用 `static`、`dynamic`、`guided` 划分, 结果如表2。

调度方式	static	dynamic	guided
时间/ms	298.63	266.28	271.10

表 2: 基于循环调度的结果

可以看到, 动态划分线程和 `guided` 方式划分线程要快于静态划分, 我们利用 `vtune` 分析工具进行进一步分析, 其中 `static` 的七个线程运行时间如图5, `dynamic` 的七个线程运行时间如图6, `guided` 的如图7。

Thread / Function / Call...	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring ▽
▶ Thread (TID: 43980)	232.371ms	1,017,500,000	1,877,500,000	0.542	97.5%
▶ Thread (TID: 47716)	218.349ms	825,000,000	1,577,500,000	0.523	100.0%
▶ Thread (TID: 42728)	211.337ms	927,500,000	1,680,000,000	0.552	60.2%
▶ Thread (TID: 26308)	210.336ms	830,000,000	1,422,500,000	0.583	100.0%
▶ Thread (TID: 44180)	207.331ms	892,500,000	1,652,500,000	0.540	66.0%
▶ Thread (TID: 41632)	198.317ms	890,000,000	1,520,000,000	0.586	100.0%
▶ Thread (TID: 36008)	182.291ms	720,000,000	1,555,000,000	0.463	100.0%

图 5: static 的 vtune 结果

Thread / Function / Call...	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring ▽
▶ Thread (TID: 43076)	225.360ms	967,500,000	1,735,000,000	0.558	54.5%
▶ Thread (TID: 47512)	211.337ms	882,500,000	1,645,000,000	0.536	49.2%
▶ Thread (TID: 32708)	210.336ms	857,500,000	1,580,000,000	0.543	100.0%
▶ Thread (TID: 30676)	208.333ms	825,000,000	1,530,000,000	0.539	78.9%
▶ Thread (TID: 46444)	205.328ms	852,500,000	1,532,500,000	0.556	100.0%
▶ Thread (TID: 29764)	204.326ms	855,000,000	1,530,000,000	0.559	68.9%
▶ Thread (TID: 42860)	202.323ms	897,500,000	1,770,000,000	0.507	48.4%

图 6: dynamic 的 vtune 结果

Thread / Function / Call...	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring ▽
▶ Thread (TID: 32708)	240.384ms	972,500,000	1,760,000,000	0.553	28.7%
▶ Thread (TID: 3860)	228.364ms	905,000,000	1,515,000,000	0.597	82.2%
▶ Thread (TID: 37264)	207.331ms	885,000,000	1,717,500,000	0.515	56.0%
▶ Thread (TID: 46524)	207.331ms	892,500,000	1,600,000,000	0.558	100.0%
▶ Thread (TID: 46980)	203.324ms	862,500,000	1,682,500,000	0.513	97.0%
▶ Thread (TID: 19068)	195.312ms	855,000,000	1,570,000,000	0.545	87.0%
▶ Thread (TID: 1584)	184.294ms	827,500,000	1,530,000,000	0.541	74.9%

图 7: guided 的 vtune 结果

可以明显看出, dynamic 动态调度方式下, 每个线程的运行时间条明显更整齐, 说明这种方法下的负载更加均衡。这是因为: 动态划分策略下, 程序运行时的 7 个线程动态轮流抽取任务, 使得负载更加均衡, 减少了资源闲置浪费的情况。再看 static 的结果, 会有几个线程的运行时间明显少于其他线程, 这是因为资源分配是一开始就决定好的, 当一个线程运行完它负责的任务后, 只能闲置等待。在本实验的情境下, guided 的运行时间比 static 快但却没有 dynamic 快, 这是因为 guided 也是一种动态划分的方式, 要比 static 负载更均衡, 但它是随着循环的遍历, 划分的块大小递减。这样的划分方式对于随着循环的遍历而规模减小的循环更有利。但是本实验比较特殊, 虽然消去块是随着外层大循环的遍历而逐渐变小, 但是回观我们之前写的代码, openmp 的 for schedule 语句是写在并行消去小循环的外面, 只能控制消去循环内部的划分粒度, 不能控制外层大循环对每次消去部分的划分粒度, 因此, 这里的 guided 并没有更好的效果, 反而因为其刚开始划分的块过大, 导致最后的效果还没 dynamic 好。

4. 其他算法策略

对于 OpenMP 程序, 我们考虑并尝试了其他多种算法策略, 介绍如下 (具体代码已在上一节介绍):

- “动态”地创建线程

将 OpenMP 创建线程且分配任务的语句放到大循环内部, 相当于每次运行到并行部分, 都要创建、销毁线程。

- 其他 OpenMP 编程范式——barrier

使用 barrier 和 master 的范式编程。

- 将除法也进行并行化

把除法部分也分为多线程运算, 并且要注意除法和消去的同步。

结果如表3。

问题规模	1000	2000	4000
初始算法策略	242.48	1900.43	15936.8
“动态”创建线程	246.07	1919.28	16256.9
barrier 范式	241.77	1909.56	16016.3
除法也并行化	230.87	1854.22	15052.7

表 3: 其他策略的结果

从结果看出:

- 将 OpenMP 创建线程且分配任务的语句放到大循环内部会影响算法性能, 且问题规模越大, 影响程度越大, 这是由于随着问题规模的增大, 创建、销毁线程的额外开销会更明显。
- 两种范式 (single + for schedule 与 master + barrier) 运行时间几乎相当。
- 将除法也并行化, 会较明显地提升程序的性能, 达到更好的加速效果。

5. OpenMP 自动向量化

OpenMP4.0 以上支持自动结合 SIMD 并行编程。编译器通过分析程序中控制流和数据流的特征, 识别并选出可以向量化执行的代码, 并将标量指令自动转换为相应的 SIMD 指令的过程。即向量化的过程由编译器自动完成, 我们只要编写正常的 C++ 代码, 让编译器会自动分析代码结构, 将适合向量化的代码部分自动生成 SIMD 指令的向量化代码。而且这些代码可以跨平台编译, 针对不同的平台生成不同的 SIMD 指令。

查看鲲鹏服务器的 GCC 版本所对应的 OpenMP 已经可以支持自动向量化了, 因此我们进行了自动向量化的实验。

当线程数 =7 时, 不同问题规模下的结果如表4。

问题规模	500	1000	2000
OpenMP 无 SIMD/ms	37.87	272.42	2233.33
OpenMP 手动 SIMD/ms	33.03	242.17	1909.36
OpenMP 自动化 SIMD/ms	34.67	265.38	2010.72

表 4: 自动向量化

可以看到, 虽然相比于没有 SIMD 的程序, OpenMP 的自动化 SIMD 算法有一定的加速效果, 但效果并没有手动编写的代码好。

6. 基于不同平台架构

个人笔记本的配置如图8。可以看到, 个人笔记本的 L1 cache 比鲲鹏服务器的 cache 大, CPU 个数比鲲鹏少。因此, 在个人笔记本进行数据划分时, 需要重新设计划分粒度, 以适应不同的 cache 大小。除此之外, 在个人笔记本 (Windows) 运行程序时, 需要重新设置 GCC 编译器, 在其中添加-fopenmp 选项, 否则无法正确地多线程编译 openMP 程序。

```
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
Address sizes: 39 bits physical, 48 bits virtual
CPU(s): 8
On-line CPU(s) list: 0-7
Thread(s) per core: 2
Core(s) per socket: 4
Socket(s): 1
Vendor ID: GenuineIntel
CPU family: 6
Model: 165
Model name: Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz
Stepping: 2
CPU MHz: 2496.008
BogoMIPS: 4992.01
Virtualization: VT-x
Hypervisor vendor: Microsoft
Virtualization type: full
L1d cache: 128 KiB
L1i cache: 128 KiB
L2 cache: 1 MiB
L3 cache: 8 MiB
```

图 8: PC

在笔记本电脑上, 进行不同规模的实验, 例如, 我们设置新的划分粒度为 28 ($n=1000$) 时, 最后实验得到的结果整体比鲲鹏要慢, 线程数为 7 时, 个人笔记本的加速尚比未达到 6, 加速效果不如鲲鹏上高。结合之前的 SIMD 和 Pthread 实验, 都表现为本机运行速度比不上鲲鹏服务器。

7. 与 pthread 的对比

我们将 pthread 算法与 OpenMP 算法进行比较, 如图9。



图 9: 对比结果

可以看到，与自己写的 Pthread 程序相比，OpenMP 程序明显更快，并且 OpenMP 程序并不需要繁琐的代码编写，而是使用了简洁的范式，解放了程序员，从代码编写转向更加注重不同的算法策略。因此，OpenMP 的程序综合来看，更加优秀。

二、特殊高斯消去

(一) 问题简介

给定消元子和被消元行，依次使用消元子消去对应的被消元行，若某行消元子为空而有其对应的被消元行，则将被消元行升格成消元子，进入到后续的消元中。

(二) 实验设计

• 方法一

在《SIMD》一节中，我们完成了特殊高斯消去的串行设计以及结合 SIMD 并行的算法，方法如下：

- 1) 逐批次读取消元子 $Act[]$ 。
- 2) 对当前批次中每个被消元行 $Pas[]$ ，检查其首项 ($Pas[row][last]$) 是否有对应消元子；若有，则将与对应消元子做异或并更新首项 ($Pas[row][last]$)，重复此过程直至 $Pas[row][last]$ 不在范围内。
- 3) 运算中，若某行的首项被当前批次覆盖，但没有对应消元子，则将它“升格”为消元子，copy 到数组 Act 的对应行，并设标志位为 1 表示非空，然后结束对该被消元行的操作。
运算后若每行的首项不在当前批次覆盖范围内，则该批次计算完成；
- 4) 重复上述过程，直至所有批次都处理完毕。

在《SIMD》的设计中，我们分批对消元子进行读取，然后使用当前这批消元子对所有被消元行进行消去，若被消元行没有对应的消元子则直接将其升格，作为消元行进入到后续的消元中。然而，由于这种方法会在程序运行中间对某些被消元行进行升格，而升格成的这些消元子会影响到后续的消元环节，即“升格”的存在会造成程序前的后依赖关系。

因此，此种算法不适用于多线程编程，而我们在《pthread》多线程编程时，更换成了一种更适合多线程的算法：我们发现，如果不考虑升格，则消元子对不同的被消元行进行处理互不冲突，可以将被消元行划分给不同的线程，每个线程负责对一部分的被消元行进行处理。若采用划分被消元行的方式，如果按照串行的思路，若将升格放到线程函数内部，并对“升格”采用单线程方法，会导致某线程正在升格的消元子无法对其他线程之前出现的或正在处理的被消元行进行操作，影响到程序的正确性。

因此，在多线程编程时，我们考虑将升格操作单独拿出来，不参与到并行运算。具体方法如下：

- 1) 每一轮将被消元行划分给不同线程，多线程不升格地处理被消元行；
- 2) 所有线程处理完之后，一个线程对可能升格的被消元行升格，其他线程等待；
- 3) 该线程升格完之后，所有线程使用更新后的消元子进入下一轮的消去；
- 4) 直到所有被消元行都不用再升格，说明消去完成，退出程序。

• 方法二

结合之前《SIMD》一节并行操作的思路，我们只考虑对最内层循环——异或循环进行多线程划分。这样就不用更改原来的串行算法思路（即原来的串行算法不会有数据依赖被破坏），只需在异或之前划分多线程即可。

（三） 理论分析

对于方法一来说，我们已经在《pthread》中证实：该方法对应的串程序与之前设计的串程序相比，由于升格的单独化，消去的循环次数会增多，对被消元行的遍历升格次数也会增多，性能会下降。但是其对应的多线程并行算法相对于较好的串行算法可以达到优化的效果。即性能：不单独处理升格的串行算法 > 单独处理升格的串行算法；单独处理升格的并行算法 > 单独处理升格的串行算法；单独处理升格的并行算法 > 不单独处理升格的串行算法。由于我们在 OpenMP 的实验中也是采用这种多线程的方法，预期效果与 pthread 中一致。

而对于方法二来说，其一，由于串程序过于复杂化，存在很多分支条件，导致“异或”对程序的整体性能的影响并不占绝对作用；其二，由于存放矩阵是按照每 8 位一个字节存放，而异或计算的 for 循环是遍历每个 4 字节的 int 变量来多线程并行划分，因此实际上循环划分的矩阵规模并不大（原矩阵规模的 $1/32$ ），所以，多线程效果可能不显著。

（四） 实验内容及算法实现

结合 SIMD，我们对上面的设计进行实现。

由于方法一的串行和 pthread 版本在《pthread》中已详细介绍，方法二的串行版本在《SIMD》中也已详细介绍，且代码篇幅过长，故不在报告中赘述。（可在文末的 GitHub 仓库中查看具体代码）

方法一需在处理被消元行的循环之前，使用 `#pragma omp for schedule(static)` 划分被消元行给多线程，在升格操作前使用 `#pragma omp single` 表明单线程工作即可。

而方法二需要在异或循环之前添加 `#pragma omp for schedule(static)` 将其多线程化。

(五) 实验结果及分析

在工作线程个数为 7、测试环境为鲲鹏服务器并且结合 neon 编程的条件下，我们对以下样例进行实验，结果如表5。

测试样例 8：矩阵列数 23045，非零消元子 18748，被消元行 14325

测试样例 9：矩阵列数 37960，非零消元子 29304，被消元行 14921

测试样例 10：矩阵列数 43577，非零消元子 39477，被消元行 54274

测试样例	8	9	10
不单独处理升格的串行算法/ms	227.28	475.28	1984.17
单独处理升格的串行算法/ms	627.93	1158.85	4890.63
单独处理升格的 pthread 算法/ms	92.98	189.26	1521.16
OpenMP 算法一/ms	98.58	184.75	1101.74
OpenMP 算法二/ms	210.8	448.12	1897.35

表 5: 特殊高斯实验结果

从表5中可看出，预期结果与我们理论分析的结果大致相同，单独处理升格的串行算法要比不单独处理升格的串行算法慢，但是这种“较差”的串行算法要比“较好”的串行算法更适合应用于 OpenMP 编程，且其多线程处理过的程序相比于“较好”的串行算法要更快。相对于为了多线程而设计的“较差”串程序，OpenMP 版本可以达到超过 6、接近线程数 7 的加速比；而相对于“较好的”串程序，OpenMP 可以达到超过 2 的加速比。并且，OpenMP 在问题规模较小的时候与 Pthread 相差很小，在大规模下，要比 Pthread 更优，这与之前的普通高斯消去结果是一致的。

而对于只给“异或”多线程化的算法而言，其加速效果并不大，说明此种算法并不能充分利用多线程的优势。

三、 源代码

GitHub 仓库地址：https://github.com/hanmaxmax/parallel_homework4