南開大學
Nankai University

计算机学院

并行程序设计调研报告

并行体系结构调研

姓名：苏胤华 赵元鸣

学号：2213893 2211757

专业：计算机科学与技术

2024 年 5 月 26 日

目录

1 普通高斯消元法 pthread 优化

1.1 设计思路

POSIX Threads 简称 Pthreads，POSIX 是"Portable Operating System Interface"（可移植操作系统接口）的缩写。POSIX 是 IEEE 为实现操作系统的可移植性而定义的一系列 API 标准，旨在兼容不同的操作系统，使得应用程序可以在不同平台之间无缝移植。Pthreads 是 POSIX 标准的一部分，主要用于多线程编程。在 C++/C 程序中，pthread 能让进程在运行时分叉为多个线程执行，并在程序的某个位置同步和合并，得到最终结果。通过 Pthreads，程序员可以创建、管理和同步多个线程，以实现并行计算和高效的资源利用。

在讨论并行计算时，Pthreads 提供了一种强大的工具。线程是轻量级的进程，具有独立的执行流，但共享相同的地址空间和全局变量。使用 Pthreads 可以在多核处理器上同时运行多个线程，从而提高程序的执行效率。

以高斯消元算法为例，算法的第二部分行消去的复杂度为 $O(n^2)$。在处理数据规模大的矩阵时，这部分的耗时占比很大，因此对这部分进行优化显得尤为重要。通过 Pthreads 进行并行优化，可以显著减少算法的执行时间。具体来说，当某一行的除法部分完成后，此后的每一行都会对其进行消元。我们可以利用多线程的思想，一次进行多个行的消去，实现加速。

在对 Pthreads 进行探究时，尝试了多种并行优化方法。例如，动态分配线程、静态分配线程结合信号量 sem_t 同步、静态 sem_t 同步加三重循环整合，以及静态 barrier 同步的 pthread 优化方法。每种方法都有其优缺点，需要根据具体的应用场景进行选择。此外，还尝试了将 pthread 和 SIMD 结合，以期获得更高的加速比。

在实际应用中，对比了 ARM 平台和 x86 平台上的性能表现。ARM 平台具有高效的能耗比和良好的并行计算能力，适用于嵌入式系统和移动设备。x86 平台则以其强大的计算能力和广泛的应用领域著称，常用于桌面计算和服务器领域。通过对这两种平台的测试，可以更全面地了解 pthread 优化的效果。

在 ARM 平台的测试中，时间单位为秒，而在 x86 平台的测试中，时间单位为毫秒。测试结果显示，经过 pthread 并行优化后，算法的执行时间显著减少。特别是在处理大规模数据时，多线程的优势更加明显。

具体来说，动态分配线程的方法灵活性较高，适用于负载不均衡的情况。但在实际实现中，线程的创建和销毁会带来额外的开销，影响整体性能。静态分配线程的方法通过预先创建好线程，避免了动态分配的开销，适合负载较均衡的场景。结合信号量 sem_t

另外，静态 sem_t 同步加三重循环整合的方法通过进一步优化循环结构，减少了同步操作的频率，提升了并行效率。而静态 barrier 同步的方法则通过设置线程同步点，使得所有线程在某一阶段完成后再进入下一阶段，适用于需要严格同步的场景。

最后，将 pthread 和 SIMD 结合的方法通过利用 SIMD 的并行计算能力，对数据进行分块处理，进一步提高了计算效率。在某些情况下，SIMD 加速比可以显著超过单纯的 pthread 并行优化，特别是在处理密集型计算任务时表现突出。

通过多种方法的尝试和测试，发现不同平台、不同场景下，pthread 并行优化的效果差异较大。具体应用时，需要结合实际需求和硬件特点，选择最优的并行优化策略。

1.2 编程实现

1.2.1 基于 ARM 平台的代码实现

ARM 平台的代码实现如下：

arm 平台上 neon 指令集的 pthread 优化

```
void neon_optimized() {                //neon优化算法
    for (int k = 0; k < N; k++) {
        float32x4_t vt = vdupq_n_f32(A[k][k]);
        int j = 0;
        for (j = k + 1; j + 4 <= N; j += 4) {
            float32x4_t va = vld1q_f32(&A[k][j]);
            va = vdivq_f32(va, vt);
            vst1q_f32(&A[k][j], va);
        }
        for (; j < N; j++) {
            A[k][j] = A[k][j] / A[k][k];
        }
        A[k][k] = 1.0;
        for (int i = k + 1; i < N; i++) {
            float32x4_t vaik = vdupq_n_f32(A[i][k]);
            for (j = k + 1; j + 4 <= N; j += 4) {
                float32x4_t vakj = vld1q_f32(&A[k][j]);
                float32x4_t vaij = vld1q_f32(&A[i][j]);
                float32x4_t vx = vmulq_f32(vakj, vaik);
                vaij = vsubq_f32(vaij, vx);
                vst1q_f32(&A[i][j], vaij);
            }
            for (; j < N; j++) {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;

        }
    }
}
```

该部分展示了使用 NEON 指令集进行向量化计算，从而加速矩阵的 LU 分解过程。下面部分展示了结合 NEON 优化和多线程的动态分配策略。通过动态分配线程，实现并行计算，从而进一步提高矩阵分解的效率。

动态分配多线程优化

```
void neon_dynamic_NUM_THREADS() {          //neon,动态分配8线程
    for (int k = 0; k < N; k++) {
        float32x4_t vt = vdupq_n_f32(A[k][k]);
        int j = 0;
        for (j = k + 1; j + 4 <= N; j += 4) {
            float32x4_t va = vld1q_f32(&A[k][j]);
```

```
            va = vdivq_f32(va, vt);
            vst1q_f32(&A[k][j], va);
        }
        for (; j < N; j++) {
            A[k][j] = A[k][j] / A[k][k];
        }
        A[k][k] = 1.0;

        int thread_cnt = NUM_THREADS;
        pthread_t* handle = (pthread_t*)malloc(thread_cnt * sizeof(pthread_t));
        threadParam_t* param = (threadParam_t*)malloc(thread_cnt *
            sizeof(threadParam_t));

        for (int t_id = 0; t_id < thread_cnt; t_id++) {
            param[t_id].k = k;
            param[t_id].t_id = t_id;
        }

        for (int t_id = 0; t_id < thread_cnt; t_id++) {
            pthread_create(&handle[t_id], NULL, neon_threadFunc_NUM_THREADS,
                &param[t_id]);
        }

        for (int t_id = 0; t_id < thread_cnt; t_id++) {
            pthread_join(handle[t_id], NULL);
        }
        free(handle);
        free(param);
    }
}
```

1.2.2 基于 X86 平台的代码实现

在 x86 平台上利用多线程和 AVX 指令集实现更高效的性能，以下是代码：

AVX 指令集下优化

```
void avx_optimized() {
    for (int k = 0; k < N; k++) {
        __m256 vt = _mm256_set1_ps(A[k][k]);
        int j = 0;
        for (j = k + 1; j + 8 <= N; j += 8) {
            __m256 va = _mm256_loadu_ps(&A[k][j]);
            va = _mm256_div_ps(va, vt);
            _mm256_storeu_ps(&A[k][j], va);
        }
        for (; j < N; j++) {
            A[k][j] /= A[k][k];
        }
```

```
        A[k][k] = 1.0f;
        for (int i = k + 1; i < N; i++) {
            __m256 vaik = _mm256_set1_ps(A[i][k]);
            for (j = k + 1; j + 8 <= N; j += 8) {
                __m256 vakj = _mm256_loadu_ps(&A[k][j]);
                __m256 vaij = _mm256_loadu_ps(&A[i][j]);
                __m256 vx = _mm256_mul_ps(vakj, vaik);
                vaij = _mm256_sub_ps(vaij, vx);
                _mm256_storeu_ps(&A[i][j], vaij);
            }
            for (; j < N; j++) {
                A[i][j] -= A[i][k] * A[k][j];
            }
            A[i][k] = 0.0f;
        }
    }
}
```

使用 pthread 创建多个线程，并分配工作给每个线程，通过动态负载平衡提升并行度

AVX 指令集下优化

```
void* avx_threadFunc(void* param) {
    threadParam_t* p = (threadParam_t*)param;
    int k = p->k;
    int t_id = p->t_id;
    for (int i = k + 1 + t_id; i < N; i += NUM_THREADS) {
        __m256 vaik = _mm256_set1_ps(A[i][k]);
        int j;
        for (j = k + 1; j + 8 <= N; j += 8) {
            __m256 vakj = _mm256_loadu_ps(&A[k][j]);
            __m256 vaij = _mm256_loadu_ps(&A[i][j]);
            __m256 vx = _mm256_mul_ps(vakj, vaik);
            vaij = _mm256_sub_ps(vaij, vx);
            _mm256_storeu_ps(&A[i][j], vaij);
        }
        for (; j < N; j++) {
            A[i][j] -= A[i][k] * A[k][j];
        }
        A[i][k] = 0.0f;
    }
    pthread_exit(NULL);
    return NULL;
}

void avx_dynamic_NUM_THREADS() {
    for (int k = 0; k < N; k++) {
        __m256 vt = _mm256_set1_ps(A[k][k]);
        int j = 0;
        for (j = k + 1; j + 8 <= N; j += 8) {
```

```
            __m256 va = _mm256_loadu_ps(&A[k][j]);
            va = _mm256_div_ps(va, vt);
            _mm256_storeu_ps(&A[k][j], va);
        }
        for (; j < N; j++) {
            A[k][j] /= A[k][k];
        }
        A[k][k] = 1.0f;

        pthread_t handle[NUM_THREADS];
        threadParam_t param[NUM_THREADS];
        for (int t_id = 0; t_id < NUM_THREADS; t_id++) {
            param[t_id].k = k;
            param[t_id].t_id = t_id;
            pthread_create(&handle[t_id], NULL, avx_threadFunc, &param[t_id]);
        }
        for (int t_id = 0; t_id < NUM_THREADS; t_id++) {
            pthread_join(handle[t_id], NULL);
        }
    }
}
```

1.3 性能测试

在研究 pthread 时，我进行了多种优化方法的尝试。首先，我探索了动态分配线程的方式，通过在运行时动态创建和管理线程，来提高程序的灵活性和资源利用效率。接着，我使用静态分配线程结合信号量 sem_t 进行同步，通过预先分配固定数量的线程并使用信号量来协调它们之间的工作，从而实现线程间的有序协作。此外，我还尝试了静态 sem_t 同步和三重循环整合的方法，通过结合循环结构和信号量同步机制，以进一步优化线程的执行效率和资源调度。最后，我使用静态 barrier 同步对 pthread 进行了优化，通过设置屏障点，确保所有线程在关键点同步，从而协调多线程的执行进度，避免资源竞争。

这些优化方法在 ARM 平台和 x86 平台上进行了测试。在 ARM 平台上，测试结果以秒为单位，显示了不同优化方法在该平台上的执行效率；而在 x86 平台上，测试结果以毫秒为单位，展示了各优化方法在该平台上的性能表现。测试结果如下

算法		500	1000	1500	2000	加速比
普通耗时	平凡算法	0.35	2.78	7.02	24.05	
	NEON 优化	0.33	2.12	7.36	19.47	1.18
动态分配	平凡算法	6.11	25.88	56.71	116.13	0.11
	NEON 优化	5.39	20.36	53.82	109.01	0.12
静态分配 + 信号量同步	平凡算法	0.13	0.71	2.32	6.12	3.93
	NEON 优化	0.11	0.68	1.96	5.32	4.35
静态分配 + 信号量 + 三重循环纳入线程	平凡算法	0.12	0.79	2.54	6.57	4.21
	NEON 优化	0.10	0.69	2.11	5.10	4.66
静态分配 +barrier	平凡算法	0.13	0.77	2.47	5.36	4.15
	NEON 优化	0.11	0.75	1.85	4.68	4.53

表 1: 不同算法在各个测试例中的执行时间（单位：s）和加速比

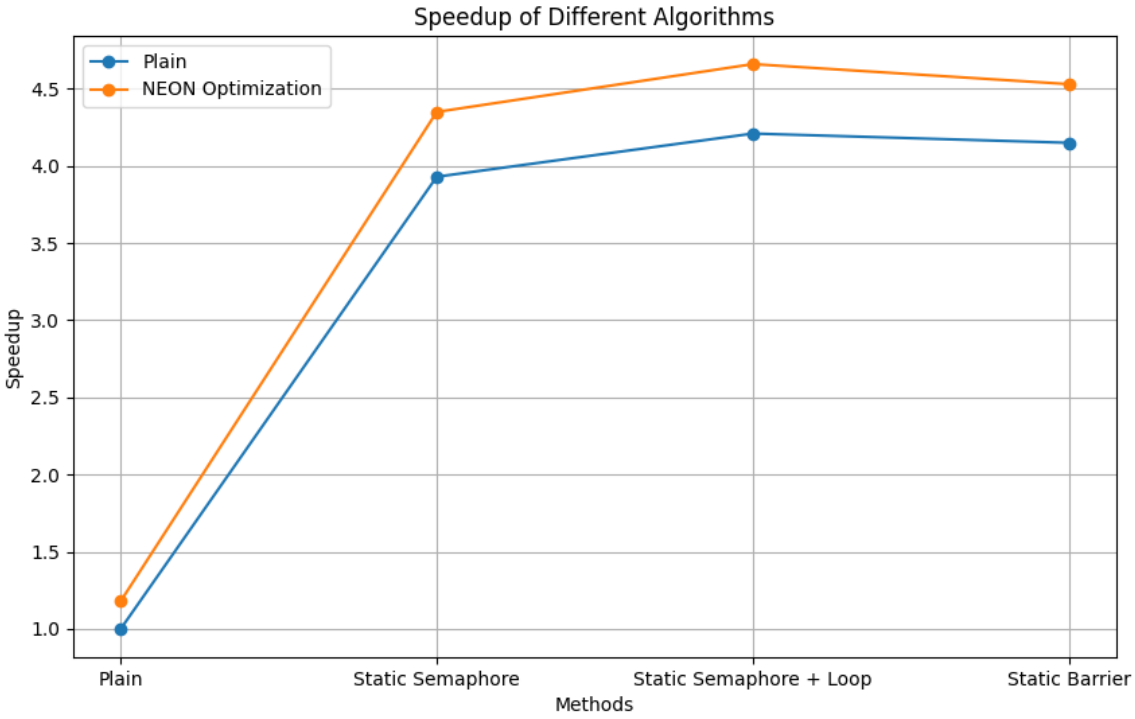


图 1.1: ARM 测试结果

算法		1000	1500	2000	2500	加速比
普通耗时	平凡算法	232	637	1701	3342	
	AVX 优化	57	168	620	1284	3.12
动态分配	平凡算法	18128	28358	63864	110130	0.028
	AVX 优化	16902	29000	48087	93623	0.030
静态分配 + 信号量同步	平凡算法	56	218	316	874	4.16
	AVX 优化	28	101	138	556	7.45
静态分配 + 信号量 + 三重循环纳入线程	平凡算法	53	183	307	731	4.62
	AVX 优化	27	82	179	645	7.61
静态分配 +barrier	平凡算法	43	169	361	841	4.52
	AVX 优化	21	84	121	481	8.79

表 2: X86 平台不同算法的执行时间（单位：ms）和加速比

算法		500	1000	1500	2000	加速比
无 OpenMP	平凡算法	28	191	756	1876	1
	AVX 优化	6	57	153	522	2.88
static	平凡算法	26	64	175	423	3.79
	AVX 优化	22	47	109	179	6.05
dynamic	平凡算法	23	68	181	341	3.82
	AVX 优化	22	59	89	160	5.92
guided	平凡算法	22	75	147	355	4.12
	AVX 优化	19	52	79	139	6.64

表 3: X86 平台不同算法的执行时间（单位：ms）和加速比

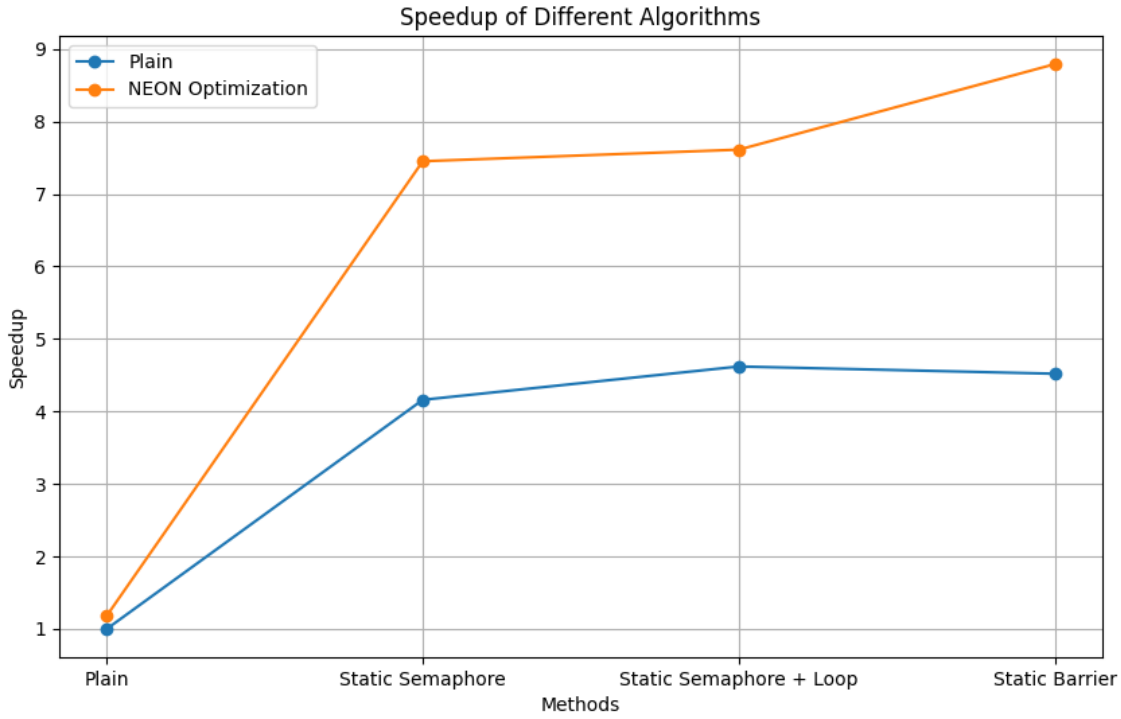


图 1.2: X86 测试结果

1.4 结果分析

在进行 pthread 优化时，我们发现其效果显著，其在 ARM 平台上也能够达到接近 4 的加速比，在 x86 平台上更能达到接近 5 的加速比。这表明 pthread 优化在不同平台上都有不错的效果，甚至有时比单纯的 SIMD 优化效果更好。然而，尽管如此，我们没有达到理论加速比 8 的原因是因为只对消去部分进行了并行优化。根据阿姆达尔定律，其余部分（如除法部分）的耗时决定了加速比的上限，因此无法达到理论加速比。此外，pthread 的线程管理也会带来时间的开销，增加耗时。

在 ARM 平台上，NEON 指令集对于整体加速比的提升较为有限，仅有 1.1 倍左右的提升幅度。这一结果也符合此前 SIMD 优化实验的结论，在 ARM 平台上的 NEON 指令集 SIMD 优化加速比在 1.1-1.2 之间。而在 x86 平台上结合 SIMD 后，也有将近翻倍的加速效果，这说明在各种情况下 SIMD 优化都真切起效了。

x86 平台上结合 AVX 后，优化效果更加显著。在使用表现最好的 barrier 同步时，加速比能达到 8.91，表现十分优异。

在这几种 pthread 优化方法中，barrier 同步的优化效果是最好的。推测这是由于 barrier 同步的方法简单易行，不需要多次的调用信号函数进行资源的分配，使得各线程工作效率更高。因此，在结合 pthread 优化的尝试中，也会优先考虑 barrier 方式的同步。

然而，由表格测试结果可知，动态分配线程的耗时巨大，耗时可以达到原先的 30-50 倍。在 x86 平台上，当数据规模为 1000 时，耗时甚至可以达到原先的 75 倍以上。因此，最后我们没有考虑动态分配线程的方式，因为其会负优化，没有实际意义。

2 普通高斯消元法 openMP 优化

2.1 设计思路

OpenMP 是一套并行编程框架，专注于 C++。其特点在于可以将串行代码最小程度地修改，便可实现自动转换为并行。这使得它具有广泛的适应性。本学期我们深入探究了 OpenMP 的并行优化方式。与 pthread 相比，OpenMP 的改动很少，工作量也相对较小，能够轻松快捷地实现并行加速。

在优化思路上，OpenMP 与 pthread 相似，都是针对行消去部分进行并行优化。不同之处在于，OpenMP 只需使用几行简单的编译语句即可将代码转换为并行程序：

首先是 ‘pragma omp parallel num_threads(NUM_THREADS),private(i,j,k,tmp)‘，该语句在外循环之外创建线程，避免线程的反复创建与销毁。同时，它设置了共享变量和私有变量，防止线程之间的冲突。

其次是 ‘pragma omp single‘，这句话表明只用一个线程来处理该部分代码，适用于除法部分等。

最后是 ‘pragma omp for schedule(static/dynamic/guided)‘，它指定了行消去部分任务的划分方式，并开始并行执行。这种方式用于行消去部分的优化。

具体的优化代码可见于 OpenMP 优化代码中。

2.2 编程实现

2.2.1 OpenMP 并行优化的 LU 分解

OpenMP 静态调度

```
void LU_omp() {
    int i = 0, j = 0, k = 0;
    float tmp = 0;
    #pragma omp parallel num_threads(NUM_THREADS), private(i, j, k, tmp)
    for (k = 0; k < N; k++) {
        #pragma omp single
        {
            tmp = A[k][k];
            for (j = k + 1; j < N; j++) {
                A[k][j] = A[k][j] / tmp;
            }
            A[k][k] = 1.0;
        }
        #pragma omp for
        for (i = k + 1; i < N; i++) {
            for (j = k + 1; j < N; j++) {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

OpenMP 动态调度

```
void LU_omp_dynamic() {
    int i = 0, j = 0, k = 0;
    float tmp = 0;
    #pragma omp parallel num_threads(NUM_THREADS), private(i, j, k, tmp)
    for (k = 0; k < N; k++) {
        #pragma omp single
        {
            tmp = A[k][k];
            for (j = k + 1; j < N; j++) {
                A[k][j] = A[k][j] / tmp;
            }
            A[k][k] = 1.0;
        }
        #pragma omp for schedule(dynamic)
        for (i = k + 1; i < N; i++) {
            for (j = k + 1; j < N; j++) {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

```
void LU_omp_guided() {
    int i = 0, j = 0, k = 0;
    float tmp = 0;
    #pragma omp parallel num_threads(NUM_THREADS), private(i, j, k, tmp)
    for (k = 0; k < N; k++) {
        #pragma omp single
        {
            tmp = A[k][k];
            for (j = k + 1; j < N; j++) {
                A[k][j] = A[k][j] / tmp;
            }
            A[k][k] = 1.0;
        }
        #pragma omp for schedule(guided)
        for (i = k + 1; i < N; i++) {
            for (j = k + 1; j < N; j++) {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

OpenMP 静态调度

```
void LU_omp() {
    int i = 0, j = 0, k = 0;
    float tmp = 0;
    #pragma omp parallel num_threads(NUM_THREADS), private(i, j, k, tmp)
    for (k = 0; k < N; k++) {
        #pragma omp single
        {
            tmp = A[k][k];
            for (j = k + 1; j < N; j++) {
                A[k][j] = A[k][j] / tmp;
            }
            A[k][k] = 1.0;
        }
        #pragma omp for
        for (i = k + 1; i < N; i++) {
            for (j = k + 1; j < N; j++) {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

2.2.2 AVX 指令集优化的 LU 分解

AVX + OpenMP 静态调度

```
void avx_omp_static() {
    int i = 0, j = 0, k = 0;
    #pragma omp parallel num_threads(NUM_THREADS), private(i, j, k)
    for (k = 0; k < N; k++) {
        #pragma omp single
        {
            __m256 t1 = _mm256_set1_ps(A[k][k]);
            for (j = k + 1; j + 8 <= N; j += 8) {
                __m256 t2 = _mm256_loadu_ps(&A[k][j]);
                t2 = _mm256_div_ps(t2, t1);
                _mm256_storeu_ps(&A[k][j], t2);
            }
            for (; j < N; j++) {
                A[k][j] = A[k][j] / A[k][k];
            }
            A[k][k] = 1.0;
        }
        #pragma omp for schedule(static)
        for (i = k + 1; i < N; i++) {
            __m256 vik = _mm256_set1_ps(A[i][k]);
```

```
            for (j = k + 1; j + 8 <= N; j += 8) {
                __m256 vkj = _mm256_loadu_ps(&A[k][j]);
                __m256 vij = _mm256_loadu_ps(&A[i][j]);
                __m256 vx = _mm256_mul_ps(vik, vkj);
                vij = _mm256_sub_ps(vij, vx);
                _mm256_storeu_ps(&A[i][j], vij);
            }
            for (; j < N; j++) {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

AVX + OpenMP 动态调度

```
void avx_omp_dynamic() {
    int i = 0, j = 0, k = 0;
    #pragma omp parallel num_threads(NUM_THREADS), private(i, j, k)
    for (k = 0; k < N; k++) {
        #pragma omp single
        {
            __m256 t1 = _mm256_set1_ps(A[k][k]);
            for (j = k + 1; j + 8 <= N; j += 8) {
                __m256 t2 = _mm256_loadu_ps(&A[k][j]);
                t2 = _mm256_div_ps(t2, t1);
                _mm256_storeu_ps(&A[k][j], t2);
            }
            for (; j < N; j++) {
                A[k][j] = A[k][j] / A[k][k];
            }
            A[k][k] = 1.0;
        }
        #pragma omp for schedule(dynamic)
        for (i = k + 1; i < N; i++) {
            __m256 vik = _mm256_set1_ps(A[i][k]);
            for (j = k + 1; j + 8 <= N; j += 8) {
                __m256 vkj = _mm256_loadu_ps(&A[k][j]);
                __m256 vij = _mm256_loadu_ps(&A[i][j]);
                __m256 vx = _mm256_mul_ps(vik, vkj);
                vij = _mm256_sub_ps(vij, vx);
                _mm256_storeu_ps(&A[i][j], vij);
            }
            for (; j < N; j++) {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
```

```
    }
}
```

AVX + OpenMP Guided 调度:

```
void avx_omp_guided() {
    int i = 0, j = 0, k = 0;
    #pragma omp parallel num_threads(NUM_THREADS), private(i, j, k)
    for (k = 0; k < N; k++) {
        #pragma omp single
        {
            __m256 t1 = _mm256_set1_ps(A[k][k]);
            for (j = k + 1; j + 8 <= N; j += 8) {
                __m256 t2 = _mm256_loadu_ps(&A[k][j]);
                t2 = _mm256_div_ps(t2, t1);
                _mm256_storeu_ps(&A[k][j], t2);
            }
            for (; j < N; j++) {
                A[k][j] = A[k][j] / A[k][k];
            }
            A[k][k] = 1.0;
        }
        #pragma omp for schedule(guided)
        for (i = k + 1; i < N; i++) {
            __m256 vik = _mm256_set1_ps(A[i][k]);
            for (j = k + 1; j + 8 <= N; j += 8) {
                __m256 vkj = _mm256_loadu_ps(&A[k][j]);
                __m256 vij = _mm256_loadu_ps(&A[i][j]);
                __m256 vx = _mm256_mul_ps(vik, vkj);
                vij = _mm256_sub_ps(vij, vx);
                _mm256_storeu_ps(&A[i][j], vij);
            }
            for (; j < N; j++) {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

纯 AVX 优化

```
void avx_optimized() {
    for (int k = 0; k < N; k++) {
        __m256 t1 = _mm256_set1_ps(A[k][k]);
        int j = 0;
        for (j = k + 1; j + 8 <= N; j += 8) {
            __m256 t2 = _mm256_loadu_ps(&A[k][j]);
            t2 = _mm256_div_ps(t2, t1);
```

```
            _mm256_storeu_ps(&A[k][j], t2);
        }
        for (; j < N; j++) {
            A[k][j] = A[k][j] / A[k][k];
        }
        A[k][k] = 1.0;
        for (int i = k + 1; i < N; i++) {
            __m256 vik = _mm256_set1_ps(A[i][k]);
            for (j = k + 1; j + 8 <= N; j += 8) {
                __m256 vkj = _mm256_loadu_ps(&A[k][j]);
                __m256 vij = _mm256_loadu_ps(&A[i][j]);
                __m256 vx = _mm256_mul_ps(vik, vkj);
                vij = _mm256_sub_ps(vij, vx);
                _mm256_storeu_ps(&A[i][j], vij);
            }
            for (; j < N; j++) {
                A[i][j] = A[i][j] - A[i][k] * A[k][j];
            }
            A[i][k] = 0;
        }
    }
}
```

2.3 性能测试

在 OpenMP 部分，我采用了 8 线程的并行粒度，在不同的数据规模下尝试了 static、dynamic、guided 三种不同的任务负载划分方式。最终的优化结果如下：

算法		500	1000	1500	2000	加速比
无 OpenMP	平凡算法	28	191	756	1876	1
	AVX 优化	6	57	153	522	2.88
static	平凡算法	26	64	175	423	3.79
	AVX 优化	22	47	109	179	6.05
dynamic	平凡算法	23	68	181	341	3.82
	AVX 优化	22	59	89	160	5.92
guided	平凡算法	22	75	147	355	4.12
	AVX 优化	19	52	79	139	6.64

表 4: X86 平台不同算法的执行时间（单位：ms）和加速比

各条件下执行时间随问题规模改变的 python 折线图呈现如下：

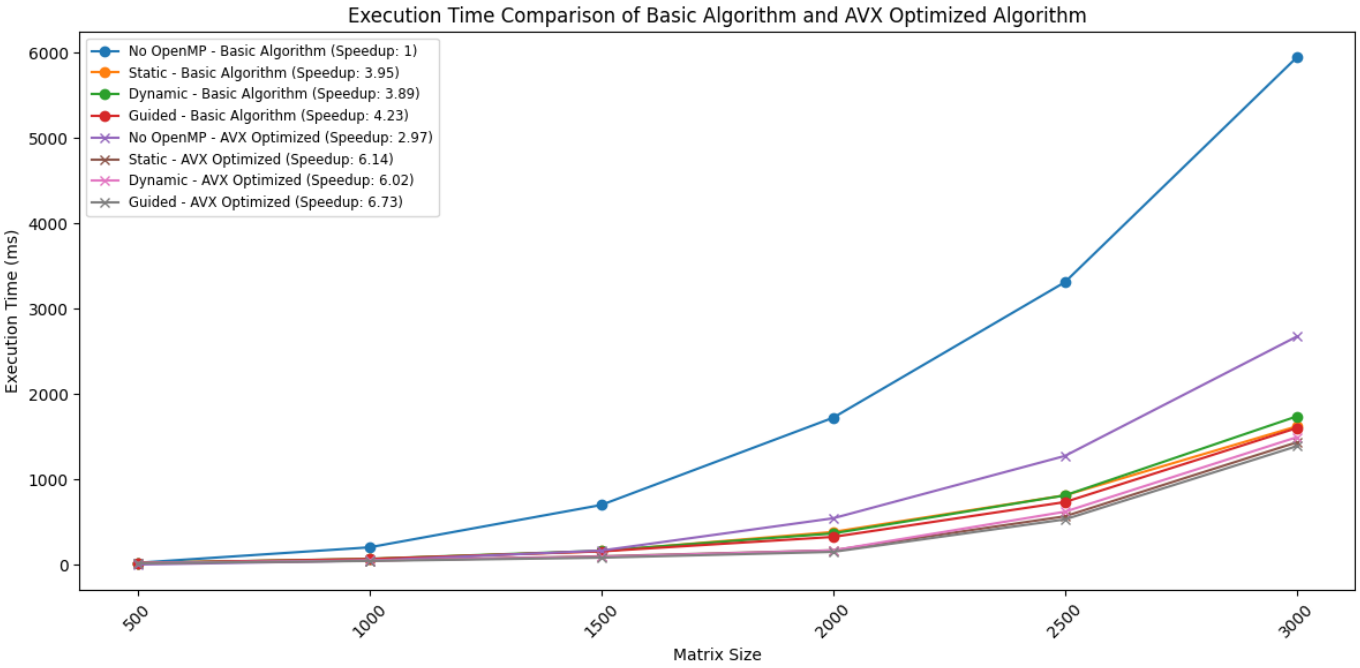


图 2.3: OpenMP 优化执行时间改变

而得到各条件下的加速比用图像方式呈现为:

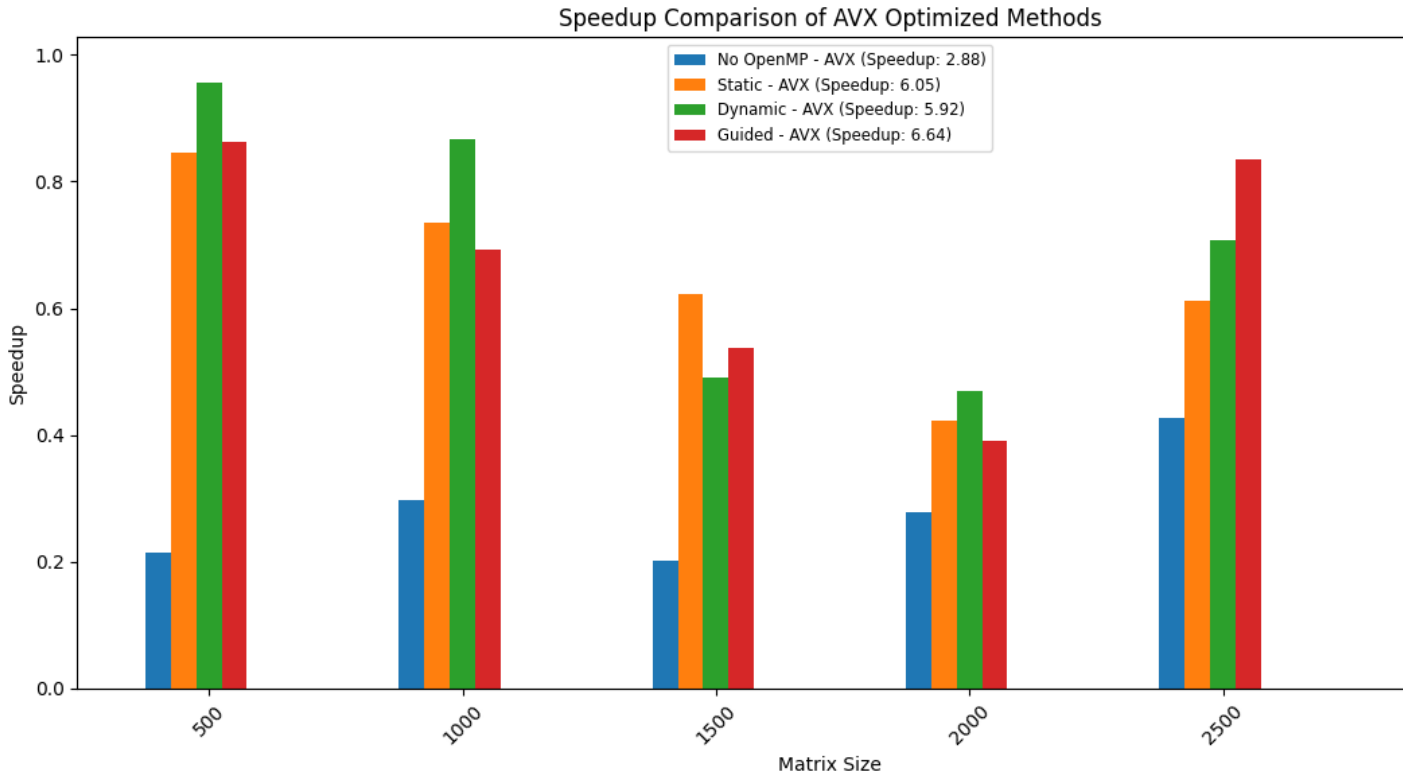


图 2.4: OpenMP 优化加速比

通过初步分析发现，当数据规模较小，即 500×500 时，OpenMP 的优化效果不显著。事实上，对于 AVX 的 SIMD 优化来说，OpenMP 反而起到了负面作用。这可能是由于 OpenMP 的线程管理和

销毁过程增加了额外的时间开销，导致线程划分后的整体效率下降。此外，对于较小规模的数据，多线程优化的实际效果并不明显，因此对于 500 规模的输入数据，不再进行进一步讨论。

2.4 结果分析

OpenMP 多线程优化对程序的加速效果明显，尤其是在采用平凡算法时，其加速比可以超过 4，远高于普通地结合 AVX 指令集的 SIMD 优化的效果。当结合 AVX 指令集后，加速比进一步增加，有时甚至能超过 10。通过加权计算后，总体加速比也能达到 6.64，说明 OpenMP 和 AVX 指令集的结合效果良好，能够显著提升程序的效率。

然而，在不同的优化方式中，dynamic 方式的优化效果表现较差，而 guided 划分方式则表现最好。这种差异可能是由于 dynamic 方式需要不断分配空闲线程，导致效果不佳，而 guided 方式则更加灵活，能够有效地均衡线程之间的负载。因此，在实际应用中，选择合适的优化方式对于提高程序性能至关重要。

虽然在某些情景下 OpenMP 的加速效果可能不如同线程数的 pthread，但其简单易用的特点使得我们能够更专注于程序的核心开发，具有巨大的实际价值。因此，对于需要进行并行编程的应用场景，OpenMP 仍然是一种非常有效的选择。

3 特殊高斯消元法 pthread 优化

3.1 算法设计

利用 pthread 对特殊高斯消元进行并行优化的核心在于使用多线程同时对多个被消元子进行消元，但是由于消元之后还要判断被消元子是否需要升格，多个线程可能会产生相同首项的需要升格的被消元子，这就会产生冲突，因此在算法当中加入了 mutex 互斥锁的机制，当某一行数据完成消元且需要升格时就要加锁，直到更新消元子的过程执行结束，之后其他正在排队的需要检查即将更新的消元子首项与刚刚更新的消元子首项是否相同，如果相同需要再次进行消元，如果不同则需要进行升格。

由于本次实验中有些文件数据过于庞大，为了取得更好的性能，使用了静态线程的创建方法，也就是一开始就定义好线程的创建数量，为了对比不同线程数量对程序性能的影响，本次实验也对不同线程数量条件进行了对比。

同时，为了进一步提高程序的性能，本次实验当中将 pthread 多线程与 AVX 结合实现对程序的并行化，在鲲鹏服务器和 codeblock 环境都进行了测试，在鲲鹏服务器中使用七个线程，采取以下的算法：basic(平凡算法)，neon，pthread，pthread+neon，在 codeblock 当中，对比 basic、AVX、pthread、AVX+pthread 算法，最后也对创建线程数量对于算法性能的影响进行了测试。

3.2 编程实现

特殊高斯消元的主体部分和原先相同，主要在于创建线程的部分，以下分别是在 codeblock 和鲲鹏服务器当中 pthread 并行算法，由于本次实验的对比方式，因此这里都以 pthread+neno（AVX）的方式进行展示，首先是 codeblock 环境下的代码实现：

AVX+Pthread 算法

```
1 void* AVX_lock_thread(void* param) {
2
3   threadParam_t* p = (threadParam_t*)param;
```

```
    int t_id = p->t_id;
    int num = p->num;

    for (int i = t_id; i + NUM_THREADS < num; i += NUM_THREADS) {
                //AVX特殊高斯消元算法
    }
    cout << t_id << "线程完毕" << endl;
    pthread_exit(NULL);
    return NULL;
}

void AVX_pthread() {
    int begin = 0;
    int flag;
    flag = readRowsFrom(begin);       //读取被消元行
    int num = (flag == -1) ? maxrow : flag;
        pthread_mutex_init(&lock, NULL);  //初始化锁
        pthread_t* handle = (pthread_t*)malloc(NUM_THREADS * sizeof(pthread_t));
        threadParam_t* param = (threadParam_t*)malloc(NUM_THREADS *
            sizeof(threadParam_t));
        for (int t_id = 0; t_id < NUM_THREADS; t_id++) {//分配任务
         param[t_id].t_id = t_id;
         param[t_id].num = num;
         pthread_create(&handle[t_id], NULL, AVX_lock_thread, &param[t_id]);
    }

    for (int t_id = 0; t_id < NUM_THREADS; t_id++) {
      pthread_join(handle[t_id], NULL);
    }
    free(handle);
    free(param);
    pthread_mutex_destroy(&lock);
}
```

其次是鲲鹏服务器 arm 架构下的编程实现：

neon+Pthread 算法

```
void* AVX_GE_thread(void* param) {
    threadParam_t* p = (threadParam_t*)param;
    int t_id = p->t_id;
    int num = p->num;
    for (int i = t_id; i + NUM_THREADS < num; i += NUM_THREADS) {
        while (findfirst(i) != -1) {
            int first = findfirst(i);
            pthread_mutex_lock(&lock);
            if (iToBasis.find(first) != iToBasis.end()) {
                int* basis = iToBasis.find(first)->second;
                pthread_mutex_unlock(&lock);
                int j = 0;
```

```
                for (; j + 8 < maxsize; j += 8) {
                    uint32x4_t vij1 = vld1q_u32((uint32_t*)&gRows[i][j]);
                    uint32x4_t vj1 = vld1q_u32((uint32_t*)&basis[j]);
                    uint32x4_t vx1 = veorq_u32(vij1, vj1);
                    vst1q_u32((uint32_t*)&gRows[i][j], vx1);

                    uint32x4_t vij2 = vld1q_u32((uint32_t*)&gRows[i][j + 4]);
                    uint32x4_t vj2 = vld1q_u32((uint32_t*)&basis[j + 4]);
                    uint32x4_t vx2 = veorq_u32(vij2, vj2);
                    vst1q_u32((uint32_t*)&gRows[i][j + 4], vx2);
                }
                for (; j < maxsize; j++) {
                    gRows[i][j] ^= basis[j];
                }
            } else {
                int j = 0;
                for (; j + 8 < maxsize; j += 8) {
                    uint32x4_t vij1 = vld1q_u32((uint32_t*)&gRows[i][j]);
                    vst1q_u32((uint32_t*)&gBasis[first][j], vij1);

                    uint32x4_t vij2 = vld1q_u32((uint32_t*)&gRows[i][j + 4]);
                    vst1q_u32((uint32_t*)&gBasis[first][j + 4], vij2);
                }
                for (; j < maxsize; j++) {
                    gBasis[first][j] = gRows[i][j];
                }
                iToBasis.insert(pair<int, int*>(first, gBasis[first]));
                ans.insert(pair<int, int*>(first, gBasis[first]));
                pthread_mutex_unlock(&lock);
                break;
            }
        }
    }
    pthread_exit(NULL);
}
void AVX_pthread() {
    int begin = 0;
    int flag = readRowsFrom(begin);
    int num = (flag == -1) ? maxrow : flag;
    auto start = chrono::high_resolution_clock::now();
    pthread_t handles[NUM_THREADS];
    threadParam_t param[NUM_THREADS];
    pthread_mutex_init(&lock, NULL);
    for (int t_id = 0; t_id < NUM_THREADS; t_id++) {
        param[t_id].t_id = t_id;
        param[t_id].num = num;
        pthread_create(&handles[t_id], NULL, AVX_GE_thread, (void*)&param[t_id]);
    }
    for (int t_id = 0; t_id < NUM_THREADS; t_id++) {
```

```
62          pthread_join(handles[t_id], NULL);
63      }
64      pthread_mutex_destroy(&lock);
65      auto finish = chrono::high_resolution_clock::now();
66      chrono::duration<double, milli> duration = finish - start;
67  }
```

对比两者算法可以发现本质上在 arm 架构和 windows 架构下进行并行算法本质思想区别不大，二者不同主要体现在使用的库函数已经变量的名称，编程实现主要就是创建线程，划分任务，结束线程。

3.3 性能测试

本次实验首先在 codeBlock 的 windows 架构下进行了平凡算法、pthread 算法（7 线程与 10 线程）、avx 算法、avx+pthread 算法，测试数据如下：

算法/ms	例 1	例 2	例 3	例 4	例 5	例 6	例 7	例 8	例 9	例 10	例 11
平凡算法	0	3	4	97	389	4602	32982	205744	319961	952471	477
AVX	0	2	2	71	304	3652	28266	132179	210296	692011	274
pthread(7 线程)	0	1	3	22	77	827	6194	48346	82360	265842	130
AVX+pthread(7 线程)	0	2	2	15	56	608	4787	36482	67843	203195	92
AVX+pthread(10 线程)	0	1	2	17	61	543	3317	26660	41141	125639	51

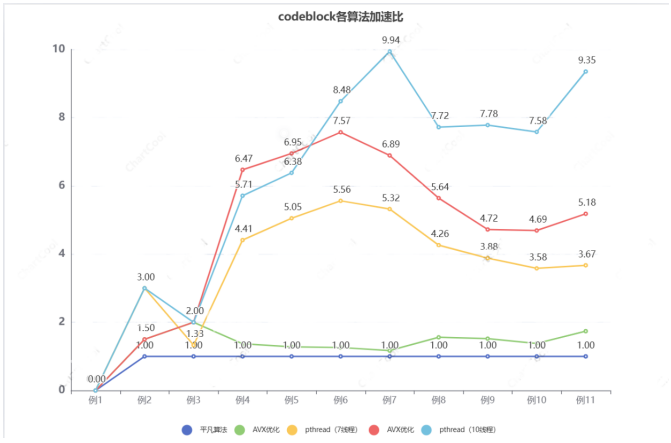


图 3.5: Windows

以下是在鲲鹏服务器进行平凡算法、neon 算法、pthread 算法、pthread+neno 算法的测试结果：

单位/ms	例 1	例 2	例 3	例 4	例 5	例 6	例 7	例 8	例 9	例 11
普通算法	0.458	17	16	140	1636	19591	123553	979128	1457030	1489
neon 优化	0.385	14	13	116	1353	15559	103536	730936	1077960	823
pthread 优化	0.567	4	4	38	284	3754	37998	168686	256741	347
neon+pthread 优化	0.362	3	3	34	228	2974	31452	140532	210063	364

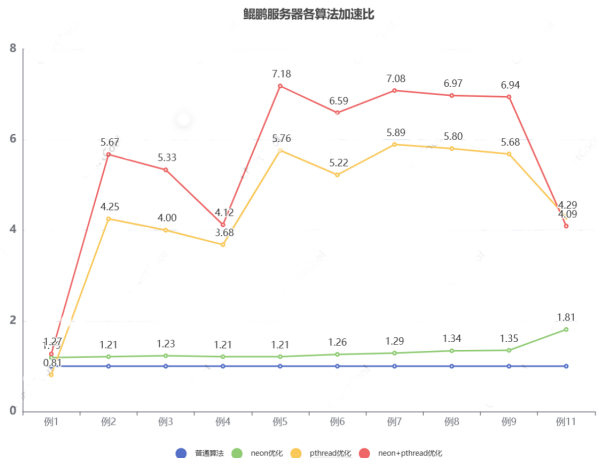
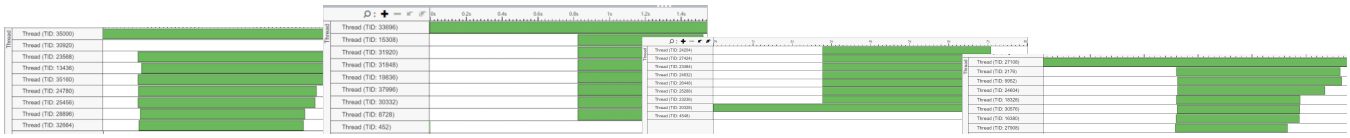


图 3.6: Arm

3.4 profiling

为了检测在并行过程中是否存在负载不均的问题，因此使用 Vtune 对各个线程的运行时间进行了统计，对 sample5、6、7、11 测试得到如下结果：从结果可以看出来，每个任务都使用了 7 个线程，整



(a) sample5 (b) sample6 (c) sample7 (d) sample11

图 3.7: vtune 下 profiling 结果

体情况来看，由于使用了静态线程的创建方法，因此各个线程开始的时间都相同，而结束时间上虽然有差异但是不大，对于前几个线程执行结束较晚，可能是消元子消元之后需要进行升格操作（后续被消元子消元之后可以被新增的消元子消去），因此时间更长。

3.5 结果分析

首先对于 windows 平台下的四个算法，单独从加速比大小来看，显然 AVX+pthread 的算法在性能上取得了最好的效果，在上一次实验当中，也就是 AVX 并行只能够取得 1.2 左右的加速比，而本次的 pthread 并行编程取得的加速比最高可以达到 5 倍左右. 可以看到本次实验也尝试将 pthread 与 AVX 并行方法相结合，也取得了不错的效果，在问题规模非常庞大的时候，AVX+pthread（10 线程）可以稳定在 8-10 倍的加速比，远远超过了其他算法的性能，对比 AVX 对平凡算法的优化和 AVX+pthread 的优化发现 AVX 对 pthread 优化比单纯使用 AVX 的优化时显著许多的，因此认为单纯的使用 AVX 指令集进行并行并不是一个非常好的方法，结合其他并行化方法可以更好的发挥 AVX 指令集的作用。最后，在实验当中也尝试了改变创建线程的数量进行实验，发现增加线程数量对于程序性能的提升幅度也很大，并且可以看到 AVX+pthread(7 线程) 随着问题规模的扩大加速效果是有一定程度的下降的，但是 10 线程数虽然也有下降，但是显然更加稳定。

其次，对于鲲鹏服务器下的 arm 架构，由于例 10 的输入文件过于庞大，在服务器总是读取失败，因此暂时没有考虑例 10 的样例，可以发现表现最好的算法还是 neon+pthread 的算法，在鲲鹏服务器

也选择创建了 7 个线程，最终可以得到 7 左右的加速比，同时发现随着问题规模的扩大，算法的加速比保持了一定的稳定性。而各个算法之间的加速比关系、趋势，可以参考 windows 架构下的几个算法，将 avx 类比 neon，基本是一致的。

而对于鲲鹏服务器下和 codeblock 下的两个架构的运行结果，发现不同于实验一当中对两个架构的对比，在实验一中相同任务相同代码的情况下，arm 架构性能是优于 windows 的，但是在本次特殊高斯消元法的实验当中，windows 从程序运行时间上来说是优于 arm 的，并且在对 arm 架构下程序测试的过程当中，增加创建的线程数量并不能对程序性能产生明显的优化，这也是与 windows 测试时不相同的，当然 arm 架构中 neon 指令集对于程序性能的优化力度是高于 avx 对于程序的优化力度的，并且在随着问题规模的扩大，arm 架构下的算法性能也比 windows 架构更加稳定，因此对于两种架构来说很难直接分出优劣，在特定的任务以及配置环境下，两种架构运行编程代码都会有自身的优势所在。

4 特殊高斯消元法 openMP 优化

4.1 算法设计

openMP 优化思路与 Pthread 相同，在本次实验中选择了 guided 的任务划分方式，使用 pragma omp for ordered schedule(guided) 结合 pragma omp ordered，保证了没有数据冲突冒险，即只有一个线程执行写入消元子，又保证了写入消元子的顺序依赖不被打破。因为写入消元子实际上也有一个顺序依赖——即靠前的消元行要先被写入。如果出现有两个消元行消元后首项相同，而靠后的一个却先执行升格为消元子时，就会引发错误。

本次实验当中，选择在 windows 下进行 openmp 以及 openmp+avx 的两种并行算法，以及鲲鹏服务器下的 openmp 算法进行测试。

4.2 编程实现

windows 下 openmp 算法：

openmp 算法

```
void GE_OpenMP() {
    int begin = 0;
    int flag;
    flag = readRowsFrom(begin);
    int num = (flag == -1) ? maxrow : flag;
    #pragma omp parallel for shared(iToBasis, gRows, gBasis) num_threads(NUM_THREADS)
    for (int i = 0; i < num; i++) {
        while (findfirst(i) != -1) {
            int first = findfirst(i);
            if (iToBasis.find(first) != iToBasis.end()) {
            with the leading term
                int* basis = iToBasis.find(first)->second;
                for (int j = 0; j < maxsize; j++) {
                    gRows[i][j] = gRows[i][j] ^ basis[j];
                    elimination
                }
            } else {
```

```
                #pragma omp critical
                {
                    for (int j = 0; j < maxsize; j++) {
                        gBasis[first][j] = gRows[i][j];
                    }
                    iToBasis.insert(pair<int, int*>(first, gBasis[first]));
                    ans.insert(pair<int, int*>(first, gBasis[first]));
                }
                break;
            }
        }
    }
}
```

windows 下 openmp+avx 算法:

openmp 算法

```
void AVX_GE_OpenMP() {
    int begin = 0;
    int flag;
    flag = readRowsFrom(begin);      // Read elimination rows
    int num = (flag == -1) ? maxrow : flag;
    #pragma omp parallel for shared(iToBasis, gRows, gBasis) num_threads(NUM_THREADS)
    for (int i = 0; i < num; i++) {
        while (findfirst(i) != -1) {
            int first = findfirst(i);
            if (iToBasis.find(first) != iToBasis.end()) {  // If a reducer exists
                int* basis = iToBasis.find(first)->second;
                int j = 0;
                for (; j + 8 < maxsize; j += 8) {
                    __m256i vij = _mm256_loadu_si256((__m256i*) & gRows[i][j]);
                    __m256i vj = _mm256_loadu_si256((__m256i*) & basis[j]);
                    __m256i vx = _mm256_xor_si256(vij, vj);
                    _mm256_storeu_si256((__m256i*) & gRows[i][j], vx);
                }
                for (; j < maxsize; j++) {
                    gRows[i][j] = gRows[i][j] ^ basis[j];
                }
            } else {
                #pragma omp critical
                {
                    int j = 0;
                    for (; j + 8 < maxsize; j += 8) {
                        __m256i vij = _mm256_loadu_si256((__m256i*) & gRows[i][j]);
                        _mm256_storeu_si256((__m256i*) & gBasis[first][j], vij);
                    }
                    for (; j < maxsize; j++) {
                        gBasis[first][j] = gRows[i][j];
                    }
```

```
33                      iToBasis.insert(pair<int, int*>(first, gBasis[first]));
34                      ans.insert(pair<int, int*>(first, gBasis[first]));
35                  }
36                  break;
37              }
38          }
39      }
40  }
```

4.3 性能测试

windows+openmp 运行时长：

算法	例 1	例 2	例 3	例 4	例 5	例 6	例 7	例 8	例 9	例 10	例 11
openop	0	1	1	16	68	785	4013	32672	53548	202366	80
openmp+avx	0	1	1	12	52	553	2890	27923	46152	152146	35

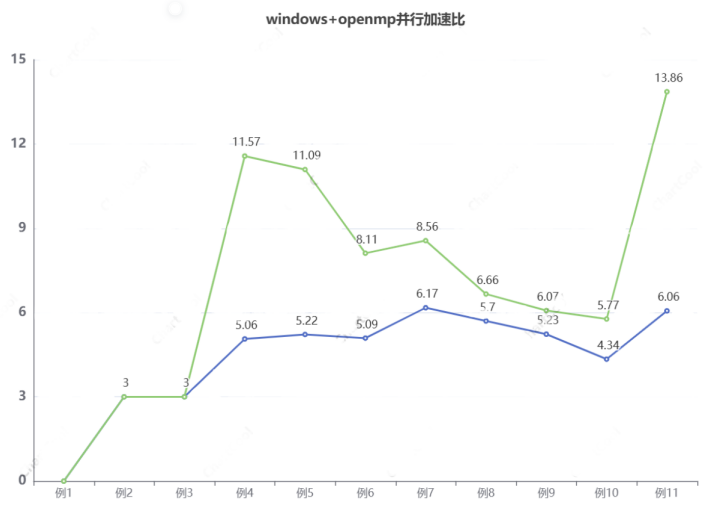


图 4.8: Windows

arm+openmp 运行时长：

算法	例 1	例 2	例 3	例 4	例 5	例 6	例 7	例 8	例 9	例 11
openmp	0.6	6	7	91	917	9401	66863	490378	631687	1427

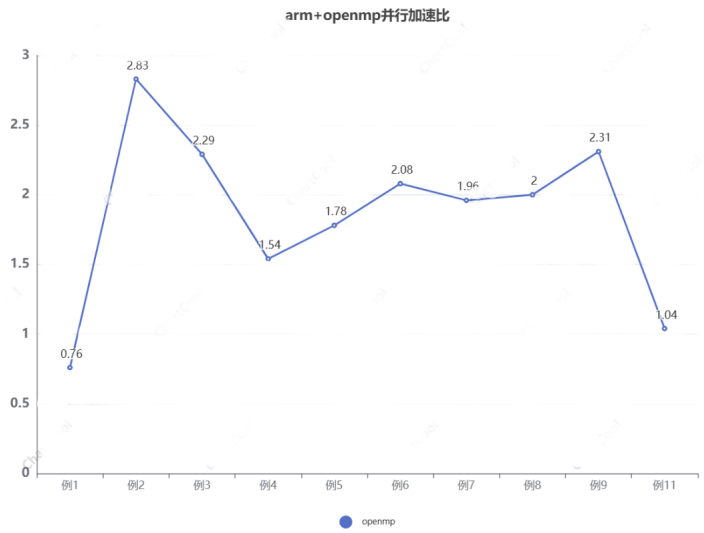


图 4.9: Arm

4.4 结果分析

可以看到在 windows 架构下，openmp 的优化加速比是比较稳定的在 5 倍左右，但是加入 avx 指令之后，对于不同问题规模的浮动就较大，而对比 arm 架构，openmp 的算法也是更优的，同时，对于例 1-例 5 由于问题规模较小，因此加速比不是很稳定，例 6-例 10 问题规模很大，对于算法的性能体现更加准确，这部分算法的性能表现也与之前 windows 架构与 arm 架构的对比是类似的，对于本次 openmp 实验中的特殊高斯消元法，windows 架构性能表现、优化力度更优，但是 arm 架构下的并行算法更加稳定。

4.5 预期优化

在 openmp 的实验当中，可以发现在 windows 架构下不加入 avx 指令的情况下程序的性能表现是比较稳定的，但是加入 avx 指令集之后加速比出现较大浮动的现象的原因值得探究，猜测与 avx 指令集实现并行方式本身的过程有关系。同时对于例 11 的稀疏矩阵任务，windows 架构相较于例 6-例 10 表现出了性能的提升，但是 arm 架构下却反而出现了下降，这个对比也值得进行进一步的探讨。

5 其他信息

我们的代码仓库地址是：https://github.com/NanGong-WenYa/Parallel_Programming
由赵元鸣进行普通高斯消去法的写作，苏胤华进行特殊高斯消去法的写作。