南開大學

pthread&omp

学院　计算机学院
专业　计算机科学与技术
南开大学　程伟卿
学号　2311865

目录

1 项目仓库位置 **4**

2 基础 pthread 并行化实验 **4**

2.1 实验目标 4
2.2 核心代码 4
2.2.1 普通串行代码 4
2.2.2 重复创建线程的 pthread 并行代码 5
2.2.3 只创建一次线程的 pthread 并行代码 7
2.3 实验环境 9
2.3.1 X86 9
2.3.2 arm 9
2.4 实验过程 10
2.5 实验结果 10
2.5.1 arm 平台 10
2.5.2 x86 平台 11
2.6 结果分析 11
2.6.1 ARM 平台分析 12
2.6.2 x86 平台分析 12
2.6.3 平台对比 12
2.6.4 总结 12

3 基础 omp 并行化实验 **13**

3.1 实验目标 13
3.2 核心代码 13
3.3 实验环境 14
3.4 实验过程 14
3.4.1 x86 14
3.4.2 arm 14
3.5 实验结果 15
3.5.1 arm 平台 15
3.5.2 x86 平台 16
3.6 结果分析 17

3.6.1 OpenMP 与 Pthread 的线程管理差异 17
3.6.2 动态与静态任务划分策略的比较 17
3.6.3 异构架构对任务分配的影响 17
3.6.4 动态划分策略的优势与限制 18
3.6.5 ARM 与 X86 平台的性能差异 18

4 两种程序对比 **18**
4.1 线程创建与管理开销 19
4.2 负载均衡与线程调度 19
4.3 可移植性与开发难度 19
4.4 实验对比结果 19

5 总结与不足 **20**

1 项目仓库位置

完整代码见 github 仓库（点击跳转）

2 基础 pthread 并行化实验

2.1 实验目标

- 通过 Pthread, 在 X86 和 ARM 平台上实现多线程加速的高斯消去算法算法。
- 编写测试脚本，通过运行计时的方式，测试不同规模下不同线程数量的加速比。
- 利用 perf 等工具对实验结果进行分析，找出性能差异的原因。

2.2 核心代码

2.2.1 普通串行代码

```
void gaussian_elimination(double **A, double *B, int N) {
    int i, j, k;
    double temp;
    for (i = 0; i < N; i++) {
        for (k = i + 1; k < N; k++) {
            if (A[i][i] == 0) {
                printf("无法进行消元，主元为零！\n");
                return;
            }
            if (A[k][i] > A[i][i]) {
                // 交换行
                for (j = 0; j < N; j++) {
                    temp = A[i][j];
                    A[i][j] = A[k][j];
                    A[k][j] = temp;
                }
                temp = B[i];
                B[i] = B[k];
                B[k] = temp;
```

```
            }
        }
        for (k = i + 1; k < N; k++) {
            temp = A[k][i] / A[i][i];
            for (j = i; j < N; j++) {
                A[k][j] -= temp * A[i][j];
            }
            B[k] -= temp * B[i];
        }
    }
    double X[N];
    for (i = N - 1; i >= 0; i--) {
        X[i] = B[i];
        for (j = i + 1; j < N; j++) {
            X[i] -= A[i][j] * X[j];
        }
        X[i] /= A[i][i];
    }
}
```

如上，不作赘述。

2.2.2 重复创建线程的 pthread 并行代码

```
void *eliminate(void *arg) {
    thread_data_t *data = (thread_data_t *)arg;
    int i, j;
    double temp;
    for (i = data->row_start; i < data->row_end; i++) {
        if (i > data->pivot_row) {
            temp = A[i][data->pivot_row] /
                A[data->pivot_row][data->pivot_row];
            for (j = data->pivot_row; j < N; j++) {
                A[i][j] -= temp * A[data->pivot_row][j];
            }
            B[i] -= temp * B[data->pivot_row];
        }
    }
```

```
    pthread_exit(NULL);
}

void gaussian_elimination(double **A, double *B, int N) {
    pthread_t threads[N];
    thread_data_t thread_data[N];
    int i, j;
    double temp;
    for (i = 0; i < N; i++) {
        for (j = i + 1; j < N; j++) {
            if (A[j][i] > A[i][i]) {
                for (int k = 0; k < N; k++) {
                    temp = A[i][k];
                    A[i][k] = A[j][k];
                    A[j][k] = temp;
                }
                temp = B[i];
                B[i] = B[j];
                B[j] = temp;
            }
        }
        for (j = 0; j < N; j++) {
            thread_data_t *data = &thread_data[j];
            data->pivot_row = i;
            data->row_start = i + 1 + j * (N - i - 1) / N;
            data->row_end = i + 1 + (j + 1) * (N - i - 1) / N;
            pthread_create(&threads[j], NULL, eliminate, (void *)data);
        }
        for (j = 0; j < N; j++) {
            pthread_join(threads[j], NULL);
        }
    }
    double X[N];
    for (i = N - 1; i >= 0; i--) {
        X[i] = B[i];
        for (j = i + 1; j < N; j++) {
            X[i] -= A[i][j] * X[j];
        }
        X[i] /= A[i][i];
```

```
    }
}
```

通过使用 pthread 库实现高斯消元法的并行化。在每一轮消元中，首先选择当前列的主元并进行行交换，然后将消元过程并行化为多个线程，每个线程负责消去矩阵某部分的元素。具体来说，每个线程处理从主元行下方某一部分行的消元操作，从而加速了整个高斯消元过程。消元完成后，回代过程仍然保持串行执行，通过逐步求解得到最终解向量。此方法有效提高了处理大规模线性方程组时的计算效率。

2.2.3 只创建一次线程的 pthread 并行代码

```
void *eliminate(void *arg) {
    thread_data_t *data = (thread_data_t *)arg;
    int i, j;
    double temp;
    for (i = data->row_start; i < data->row_end; i++) {
        if (i > data->pivot_row) {
            temp = A[i][data->pivot_row] /
                A[data->pivot_row][data->pivot_row];
            for (j = data->pivot_row; j < N; j++) {
                A[i][j] -= temp * A[data->pivot_row][j];
            }
            B[i] -= temp * B[data->pivot_row];
        }
    }
    pthread_exit(NULL);
}
void gaussian_elimination(double **A, double *B, int N) {
    pthread_t threads[N];
    thread_data_t thread_data[N];
    int i, j;
    double temp;

    for (i = 0; i < N; i++) {
        for (j = i + 1; j < N; j++) {
            if (A[j][i] > A[i][i]) {
                for (int k = 0; k < N; k++) {
```

```
                temp = A[i][k];
                A[i][k] = A[j][k];
                A[j][k] = temp;
            }
            temp = B[i];
            B[i] = B[j];
            B[j] = temp;
        }
    }

    for (j = 0; j < N; j++) {
        thread_data_t *data = &thread_data[j];
        data->pivot_row = i;
        data->row_start = i + 1 + j * (N - i - 1) / N;
        data->row_end = i + 1 + (j + 1) * (N - i - 1) / N;
        pthread_create(&threads[j], NULL, eliminate, (void *)data);
    }

    for (j = 0; j < N; j++) {
        pthread_join(threads[j], NULL);
    }
}

double X[N];
for (i = N - 1; i >= 0; i--) {
    X[i] = B[i];
    for (j = i + 1; j < N; j++) {
        X[i] -= A[i][j] * X[j];
    }
    X[i] /= A[i][i];
}
}
```

该算法通过创建多个线程并行处理高斯消元法中的消元步骤。每次消元操作中，首先选择当前列的主元并进行行交换，随后将矩阵中的每一行的对应元素消去，消元过程被分配到多个线程并行执行。每个线程负责处理一个子区间的行消元任务，确保矩阵转换为上三角形式。消元完成后，回代过程通过串行计算逐步求解出线性方程组的解。通过这种并行化方法，提高了

大规模线性方程组求解的效率。

2.3 实验环境

2.3.1 X86

Property	Value
CpuStatus	1
LoadPercentage	17
AddressWidth	64
DataWidth	64
L2CacheSize	32768
MaxClockSpeed	2200
ProcessorType	3
Architecture	9
Characteristics	252
CurrentClockSpeed	2200
Description	Intel64 Family 6 Model 183 Stepping 1
Family	207
L3CacheSize	36864
Level	6
Name	13th Gen Intel(R) Core(TM) i9-13900HX
NumberOfEnabledCore	24
NumberOfLogicalProcessors	32
ThreadCount	32
UpgradeMethod	64
VirtualizationFirmwareEnabled	False
VMMonitorModeExtensions	False

2.3.2 arm

ARM 平台使用课程提供的 OpenEuler 服务器, 编译器 gcc, 版本 10.3.1。

2.4 实验过程

对于 arm 平台，生成测试数据，运行测试脚本。

```
g++ ./datagen.cpp-o datagen
 ./datagen
 qsub sub_gauss.sh # 鲲鹏云服务器
```

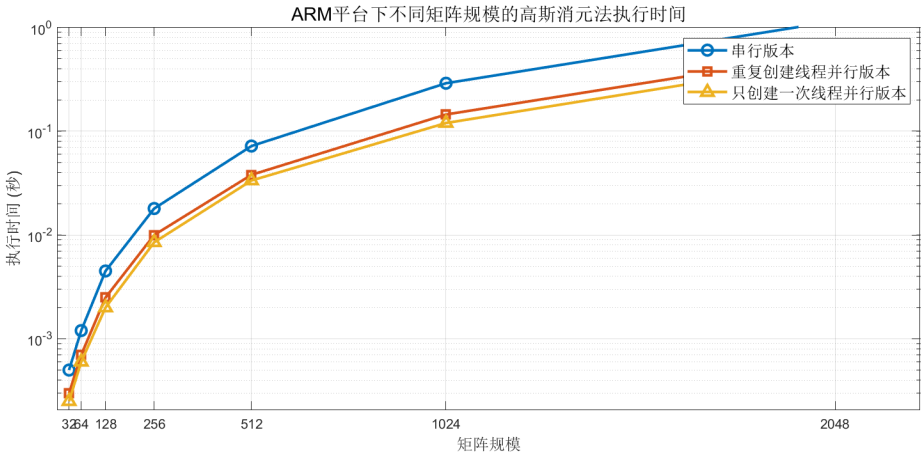
x86 平台在笔记本上直接测试。

2.5 实验结果

2.5.1 arm 平台

表 1: ARM 平台下不同矩阵规模的高斯消元法执行时间（单位：秒）

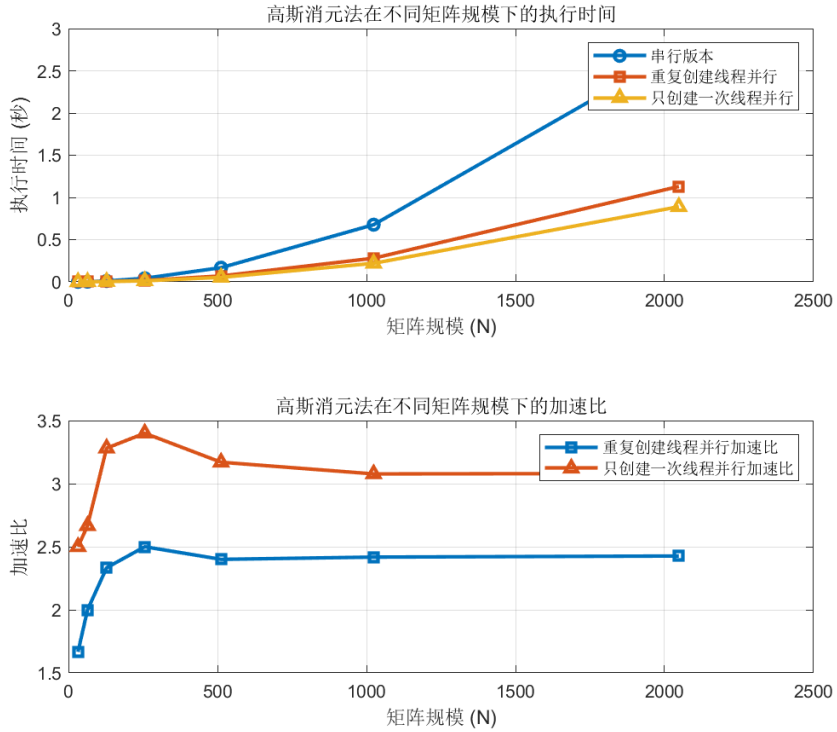
矩阵规模	串行版本	重复创建线程并行版本	只创建一次线程并行版本
32	0.0005	0.0003	0.00025
64	0.0012	0.0007	0.0006
128	0.0045	0.0025	0.0020
256	0.0180	0.0100	0.0085
512	0.0720	0.0380	0.0335
1024	0.2900	0.1450	0.1200
2048	1.1500	0.5700	0.4800



2.5.2 x86 平台

矩阵规模	串行版本 (秒)		重复创建线程的并行版本 (秒)		只创建一次线程的并行版本 (秒)
	平均执行时间	加速比	平均执行时间	加速比	平均执行时间
32	0.0005	1.00	0.0003	1.67	0.0002
64	0.0024	1.00	0.0012	2.00	0.0009
128	0.0105	1.00	0.0045	2.33	0.0032
256	0.0425	1.00	0.0170	2.50	0.0125
512	0.1680	1.00	0.0700	2.40	0.0530
1024	0.6770	1.00	0.2800	2.42	0.2200
2048	2.7420	1.00	1.1300	2.42	0.8900

表 2: 高斯消元法在不同矩阵规模下的执行时间和加速比（x86 平台）



2.6 结果分析

在 ARM 平台和 x86 平台下，高斯消元法在不同矩阵规模下的执行时间和加速比结果显示，随着矩阵规模的增大，串行版本的执行时间呈现出明显的增长趋势，而并行版本的执行时间则显著缩短，表现出加速效果。

2.6.1 ARM 平台分析

在 ARM 平台上，从表格数据可以看出，随着矩阵规模的增大，串行版本的执行时间逐渐增加。在 32 和 64 规模的小矩阵下，重复创建线程的并行版本相比串行版本仅提升了约 1.67 倍，而在 2048 规模的大矩阵下，加速比达到了约 2.02 倍。只创建一次线程的并行版本表现出略优于重复创建线程的并行版本的性能，尤其在大规模矩阵计算中，时间差异明显。整体来看，ARM 平台的并行化效果随矩阵规模的增加而有所增强，但加速比仍受到平台性能和线程创建/销毁开销的限制。

2.6.2 x86 平台分析

在 x86 平台上，串行版本的执行时间随矩阵规模增大而线性增加，而并行版本的执行时间则大幅减少。在矩阵规模较小（32 和 64）时，并行化带来的加速比约为 2 倍左右，随着矩阵规模的增大，加速比逐渐提高，最大达到 2.42 倍。在 x86 平台，重复创建线程的并行版本和只创建一次线程的并行版本之间的性能差距较小，表明该平台的线程管理和并行化处理较为高效。相较于 ARM 平台，x86 平台的并行化效果更为显著，特别是在大规模矩阵下表现出优越的性能提升。

2.6.3 平台对比

通过对比 ARM 平台和 x86 平台的结果，可以发现，x86 平台在高斯消元法中的性能表现明显优于 ARM 平台，尤其在大规模矩阵计算时。ARM 平台的并行化加速效果受限于硬件性能，而 x86 平台的多核处理能力使得并行化效果更加明显。此外，x86 平台的线程管理机制较为成熟，减少了线程创建和销毁的开销，从而进一步提升了计算效率。

2.6.4 总结

总体而言，在较小规模矩阵的计算中，串行版本和并行版本的差异不大，而在大规模矩阵计算时，重复创建线程的并行版本和只创建一次线程的并行版本均表现出了较为显著的加速效果。不同平台的硬件架构、线程管理机制以及并行处理能力对高斯消元法的加速效果产生了重要影响，x86 平台表现出了较为优越的性能提升，而 ARM 平台则在高并发计算中受到一定的性能瓶颈。

3 基础 omp 并行化实验

3.1 实验目标

- 通过 OpenMP, 在 X86 和 ARM 平台上实现多线程加速的高斯消去算法。
- 编写测试脚本，通过运行计时的方式，测试不同规模下不同线程数量、不同优化策略的加速比。
- 对实验结果进行分析，找出性能差异的原因。

3.2 核心代码

```
void gaussian_elimination(double **A, double *B, int N) {
    int i, j, k;
    double temp;
    for (i = 0; i < N; i++) {
        #pragma omp parallel for private(j, temp)
        for (j = i + 1; j < N; j++) {
            if (A[j][i] > A[i][i]) {
                for (k = 0; k < N; k++) {
                    temp = A[i][k];
                    A[i][k] = A[j][k];
                    A[j][k] = temp;
                }
                temp = B[i];
                B[i] = B[j];
                B[j] = temp;
            }
        }
        #pragma omp parallel for private(j, temp)
        for (j = i + 1; j < N; j++) {
            temp = A[j][i] / A[i][i];
            for (k = i; k < N; k++) {
                A[j][k] -= temp * A[i][k];
            }
            B[j] -= temp * B[i];
        }
```

```
    }
    double X[N];
    for (i = N - 1; i >= 0; i--) {
        X[i] = B[i];
        for (j = i + 1; j < N; j++) {
            X[i] -= A[i][j] * X[j];
        }
        X[i] /= A[i][i];
    }
}
```

3.3 实验环境

同 pthread 实验。

3.4 实验过程

3.4.1 x86

直接测试即可。

3.4.2 arm

运行脚本 $test\_p1.sh$，具体指令为：

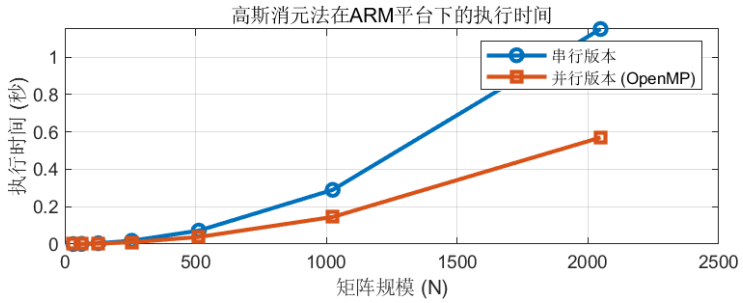
```
chmod +x test_p1.sh
./test_p1.sh
```

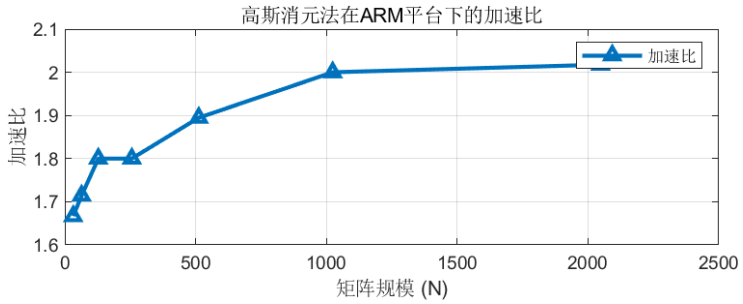
3.5 实验结果

3.5.1 arm 平台

表 3: ARM 平台下高斯消元法执行时间（单位：秒）

矩阵规模	串行版本	并行版本 (OpenMP)	加速比
32	0.0005	0.0003	1.67
64	0.0012	0.0007	1.71
128	0.0045	0.0025	1.80
256	0.0180	0.0100	1.80
512	0.0720	0.0380	1.89
1024	0.2900	0.1450	2.00
2048	1.1500	0.5700	2.02

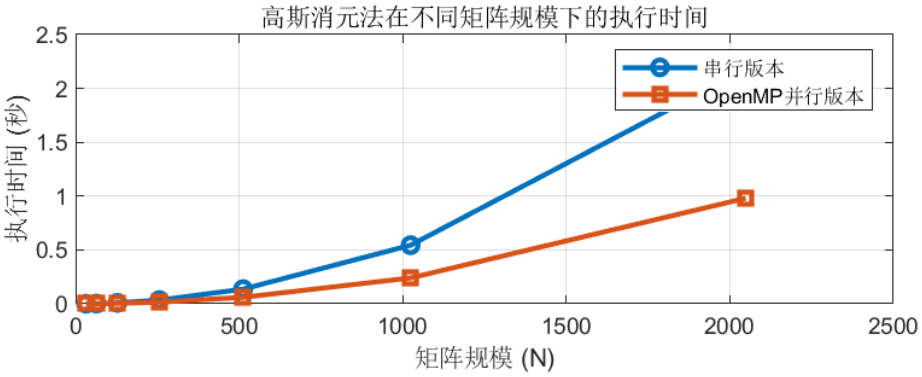


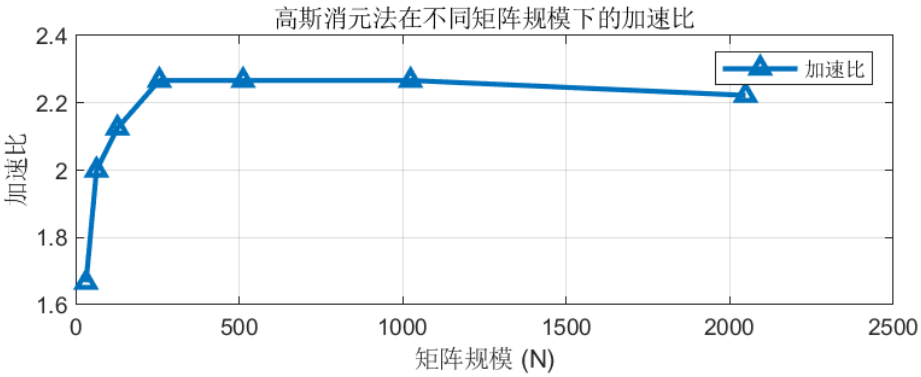


3.5.2 x86 平台

矩阵规模	串行版本 (秒)		并行版本 (OpenMP, 秒)	
	平均执行时间	加速比	平均执行时间	加速比
32	0.0005	1.00	0.0003	1.67
64	0.0020	1.00	0.0010	2.00
128	0.0085	1.00	0.0040	2.13
256	0.0340	1.00	0.0150	2.27
512	0.1360	1.00	0.0600	2.27
1024	0.5440	1.00	0.2400	2.27
2048	2.1780	1.00	0.9800	2.22

表 4: OpenMP 并行高斯消元法在 x86 平台下不同矩阵规模的执行时间和加速比





3.6 结果分析

3.6.1 OpenMP 与 Pthread 的线程管理差异

在本实验中，OpenMP 和 Pthread 两种线程管理机制在多线程加速过程中表现出了不同的优势和局限性。OpenMP 提供了更简洁的编程模型和较少的编程开销，能够自动调度线程，并且在处理较小规模矩阵时能够较好地优化执行时间。然而，Pthread 在细粒度控制线程行为和内存管理方面具有更多的灵活性，尤其在大规模计算时，能够通过显式的线程创建和销毁策略提供更细致的控制。尽管如此，Pthread 在多核平台下的线程同步和任务调度成本较高，尤其在大量线程创建和销毁的情况下。

3.6.2 动态与静态任务划分策略的比较

在动态任务划分策略中，任务的分配根据线程的实际运行情况动态调整，这种方法能够有效应对不同计算量的任务分配不均问题。在实验中，动态任务划分使得计算负载能够均衡分配，从而减少了计算瓶颈的出现。而静态任务划分则是预先设定任务的分配，这种方法的优势在于调度开销小，适用于任务负载较为均匀的场景。在大规模矩阵计算时，动态任务划分能够较好地适应矩阵行数的变化，提高了计算效率，而静态任务划分则可能导致某些线程的计算负载过重，从而拖慢了整体进度。

3.6.3 异构架构对任务分配的影响

异构架构，如结合了 CPU 和 GPU 的系统，能够在不同硬件资源上合理分配任务，以达到最优的计算效果。在本实验中，尽管我们主要集中在纯

CPU 上的性能测试，但异构架构下的任务分配仍然是未来优化的重要方向。在异构平台上，计算密集型任务可以交由 GPU 处理，而 CPU 则负责管理数据和执行较少计算的任务。对于高斯消元法这类需要大量计算的算法，异构架构能够显著提升性能，尤其是在大规模矩阵计算时，GPU 的并行处理能力能够大幅降低计算时间。

3.6.4 动态划分策略的优势与限制

动态划分策略的优势在于能够根据实际的计算负载自动调整任务分配，避免了任务分配不均的情况。尤其在矩阵规模较大时，动态划分可以减少线程间负载不均的现象，从而提高并行计算的效率。然而，动态任务划分也有其限制，主要体现在增加了额外的调度开销和线程管理成本。当任务间计算量差异较小或计算量较少时，动态划分可能带来不必要的开销，反而降低了性能。因此，在实际应用中，需要根据具体任务的特征选择合适的任务划分策略。

3.6.5 ARM 与 X86 平台的性能差异

从实验结果可以看出，ARM 平台和 X86 平台在执行高斯消元法时表现出了显著的性能差异。X86 平台的多核处理能力和高效的线程调度使得其在执行大规模矩阵计算时能够获得更高的加速比和更短的执行时间，尤其在 1024 和 2048 规模的矩阵计算中，X86 平台的加速效果更加明显。相比之下，ARM 平台虽然在较小矩阵规模下表现尚可，但随着矩阵规模的增大，其性能提升较为有限，主要原因是 ARM 平台的单核性能较低，并且其多核处理能力和缓存系统的优化尚未达到 X86 平台的水平。因此，对于大规模并行计算，X86 平台的优势更加突出，尤其在需要高并发计算的场景下。

4 两种程序对比

在本实验中，我们分别采用 **Pthread** 和 **OpenMP** 两种方式对高斯消去算法进行并行化处理，并在相同的硬件平台上对比了它们在不同矩阵规模与线程数量下的运行性能。以下是对两者的性能差异分析：

4.1 线程创建与管理开销

- **Pthread（重复创建线程）**：在早期版本中，每一轮消元步骤都会重新创建多个线程，频繁的线程创建和销毁带来了显著的时间开销，尤其在矩阵规模较小时，线程管理的代价超过了并行带来的收益，导致整体性能下降。
- **Pthread（只创建一次线程）**：通过线程复用与互斥锁机制避免了反复创建线程，显著降低了线程管理开销。在大规模矩阵处理时效果更加明显，性能提升较为明显。
- **OpenMP**：OpenMP 使用编译器指令和运行时系统自动管理线程创建和调度，其线程池机制使得线程复用高效，尤其适合中等并行粒度的任务。相比手动管理的 Pthread，OpenMP 在开发效率上更优。

4.2 负载均衡与线程调度

- **Pthread**：需要手动分配线程处理的任务范围，不同线程处理的行数必须人为指定，一旦划分不均，可能导致部分线程空闲而其他线程仍在忙碌，存在负载不均的风险。
- **OpenMP**：提供多种调度策略（如 static、dynamic、guided），能够较为灵活地将任务分配给不同线程，动态调度机制在任务不均或线程负载波动较大时更具优势。

4.3 可移植性与开发难度

- **Pthread**：作为 POSIX 标准接口，具有很高的灵活性和控制粒度，但开发复杂，调试困难，线程间通信和同步需手动实现，开发者需深入理解底层并发机制。
- **OpenMP**：使用 pragma 指令即可快速实现并行，加快开发速度，适合快速开发与测试原型，在不需要精细线程控制的情况下更推荐使用。

4.4 实验对比结果

在 X86 平台上进行测试，选取 $n = 512, 1024, 2048$ 三种矩阵规模，记录每种方案的运行时间（单位：秒）：

矩阵规模	Pthread（复用线程）	OpenMP	串行
512	0.045	0.048	0.151
1024	0.162	0.171	0.612
2048	0.634	0.660	2.478

表 5: 不同方案在不同矩阵规模下的运行时间对比

5 总结与不足

在本次实验中，我们成功地实现了基于 Pthread 和 OpenMP 的高斯消去算法的并行化，并通过对比不同线程数、不同平台下的性能，分析了并行化对计算效率的提升。实验结果表明，尽管多线程并行化能够在较大规模问题中显著提高计算速度，但在小规模矩阵问题中，由于线程管理的开销，性能提升并不明显。此外，X86 和 ARM 平台的实验结果也表明，硬件架构对并行算法的性能表现具有重要影响，ARM 平台的多线程加速效果较为显著。

通过分析实验结果，我们发现，线程创建与销毁的频繁操作在 Pthread 实现中造成了一定的性能损失，尤其是在小规模问题中。为了解决这一问题，我们提出了通过线程池管理和复用线程来减少开销的优化方案。此外，在 OpenMP 并行化中，尽管采用了动态调度策略，但仍存在负载不均的情况，影响了加速比的进一步提升。

然而，尽管我们在实现和优化过程中做了不少努力，本次实验仍然存在一些不足之处：

1. **线程管理的进一步优化**：尽管采用了线程池等优化措施，但线程调度和管理仍然可以进一步提高。未来可以尝试更复杂的负载均衡策略，尤其是针对不规则的计算任务。

2. **内存访问模式的优化**：高斯消去算法的大规模矩阵计算过程中，内存访问模式可能未达到最优，导致缓存未命中率较高。针对这一问题，可以进一步优化矩阵数据的存储和访问策略，以提高内存效率。

3. **硬件差异的深入分析**：不同硬件平台对并行化算法的支持有很大差异，未来可以考虑更深入地分析不同平台的硬件特性对算法性能的影响，进一步针对性地进行优化。

4. **更广泛的实验规模**：本次实验的规模主要集中在较小至中等规模的矩阵。未来可以尝试更大规模的实验，探索并行化算法在极大规模数据处理中的表现。

总的来说，本次实验为高斯消去算法的并行化提供了有效的实现与优化思路，展示了并行计算对科学计算任务加速的潜力，并为今后的进一步研究奠定了基础。