**上海大学 计算机学院**

**《计算机系统结构》报告**

**姓名 徐驰 学号 20124652 指导教师 沈文枫**

**实验名称: 多核环境下OpenMP并行编程**

**一、实验目的**

1.在Linux平台上编译和运行OpenMP程序。

2.在Windows平台上编译和运行OpenMP程序。

3.掌握OpenMP并行编程基础。

4.用OpenMP实现最基本的数值算法“矩阵乘法”。

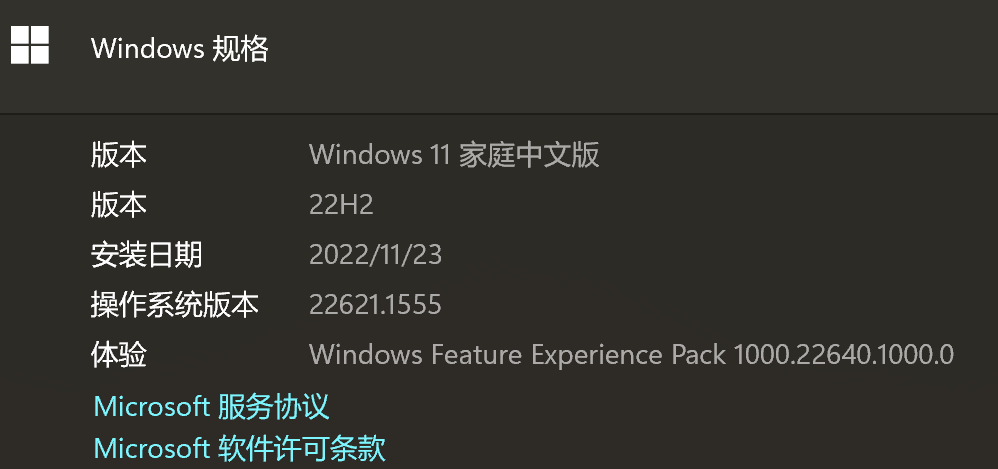
5.掌握for编译制导语句。

6.对并行程序进行简单的性能调优。

**二、实验环境**

1.Windows主机

操作系统：Windows11 22H2 家庭中文版

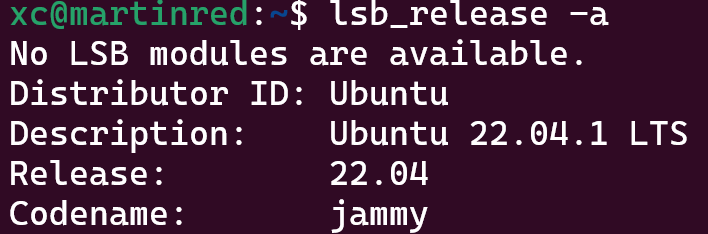


硬件规格：

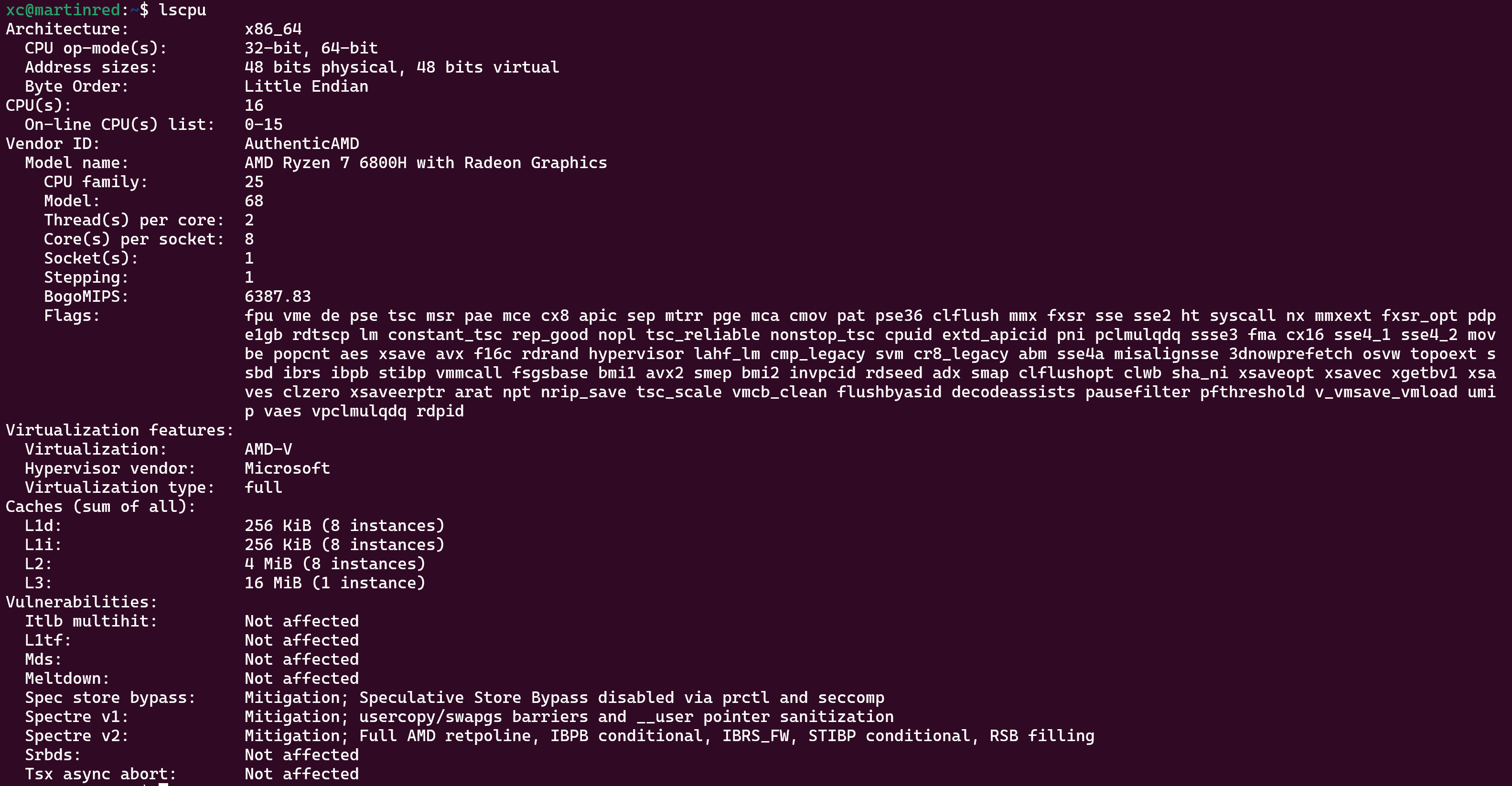


2.Linux子系统（WSL）

操作系统：



硬件规格：



**二、实验原理**

1.OpenMP简介：

·OpenMP是一种共享内存并行编程模型，可以在多核处理器上并行执行程序。

·OpenMP使用指令来标记要并行化的代码段，并使用线程来执行这些代码段。

·OpenMP支持多种类型的指令，例如并行for循环、并行区域和任务并行等。

2.Linux平台上的OpenMP编译和运行：

·在Linux平台上，可以使用GCC编译器来编译和运行OpenMP程序。

·要编译OpenMP程序，请确保在命令行中使用-fopenmp选项，在代码中包含<omp.h>头文件，并使用#pragma omp指令来标记要并行化的代码段。

·要运行OpenMP程序，请使用终端窗口或命令行运行可执行文件。

3.Windows平台上的OpenMP编译和运行：

·在Windows平台上，可以使用Microsoft Visual Studio或MinGW等编译器来编译和运行OpenMP程序。

·要编译OpenMP程序，请确保在代码中包含<omp.h>头文件，并使用#pragma omp指令来标记要并行化的代码段。

·要运行OpenMP程序，请使用命令行或IDE运行可执行文件。

4.矩阵乘法的并行实现：

·矩阵乘法是一种常见的数值算法，可以使用OpenMP来并行计算。

·在矩阵乘法中，可以使用嵌套的for循环来遍历矩阵，并使用#pragma omp指令来并行化内部循环。

·在将矩阵乘法并行化时，请注意避免竞争条件和数据依赖等并发问题。

5.OpenMP指令和编译制导语句：

·OpenMP支持多种类型的指令和编译制导语句，例如并行for循环、reduction指令和private指令等。

·在使用OpenMP指令和编译制导语句时，请注意遵循正确的语法和语义规则，避免并发问题和性能瓶颈。

6.并行程序性能调优：

·在调优并行程序时，可以使用一些技术来提高程序的性能。

·例如，可以使用循环展开来减少循环迭代次数，使用数据局部化来减少内存访问等。

·在调优并行程序时，请注意评估程序的性能，并使用性能分析工具来识别瓶颈和优化机会。

**三、实验内容**

**1．实验任务一：在Linux平台上编译和运行OpenMP程序**

1. 实验步骤

1.在linux中，使用vim hellomp.c

2.输入代码：

#include <omp.h>

#include <stdio.h>

int main()

{

int nthreads,tid;

omp\_set\_num\_threads(8);

#pragma omp parallel private(nthreads,tid)

{

tid=omp\_get\_thread\_num();

printf("Hello World from OMP thread %d\n",tid);

if(tid==0)

{

nthreads=omp\_get\_num\_threads();

printf("Number of threads is %d\n",nthreads);

}

}

}

3.保存退出。

4.安装gcc，使用“sudo yum install -y gcc”

5.“gcc -fopenmp -O2 -o hellomp.out hellomp.c”生成可执行程序。

6.“./hellomp.out”执行程序，多次执行并记录结果。

7.修改线程数为10“OMP\_NUM\_THREADS=10”

8.将线程数添加为环境变量“export OMP\_NUM\_THREADS”

9.修改hellomp.c,删除 omp\_set\_num\_threads(8)语句

10.“gcc -fopenmp -O2 -o hellomp2.out hellomp2.c”再编译生成另一段程序。

11.“./hellomp2.out”执行程序，多次执行并记录结果。

12.“unset OMP\_NUM\_THREADS”，取消环境变量设置。

13.“./hellomp2.out”执行程序，多次执行并记录结果。

1. 实验现象

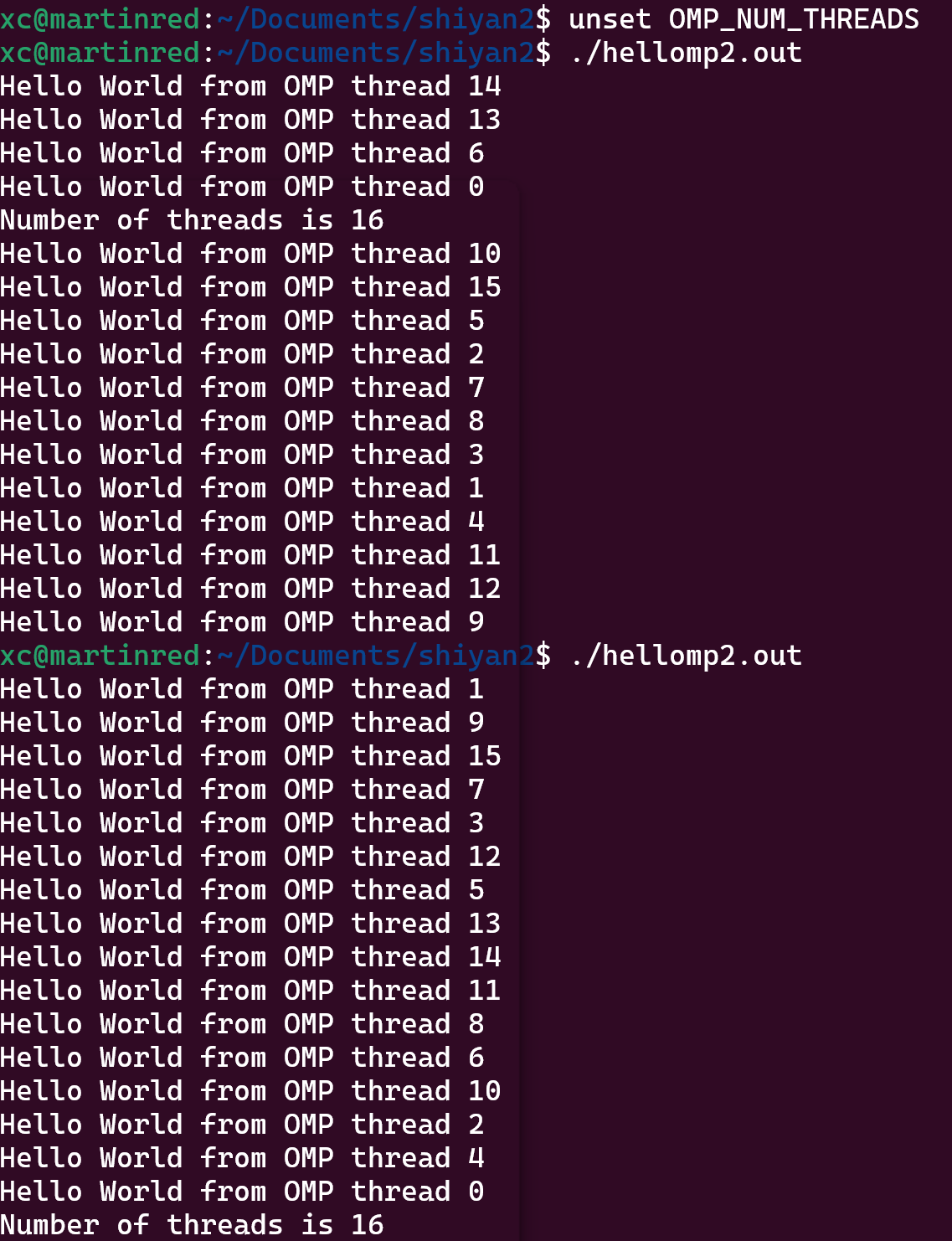
1.第6步中，实验结果：



2.第11步中，实验结果：



3.第13步中，实验结果：



1. 数据记录、分析与处理

1.代码分析

该代码使用OpenMP库实现了多线程并行计算，通过设置私有变量nthreads和tid，以及使用#pragma omp parallel指令，并结合omp\_get\_thread\_num和omp\_get\_num\_threads函数，实现了线程编号和线程总数的打印输出功能。

2.实验结果分析：

根据实验步骤，我先是使用8个线程执行程序hellomp.out，并记录了多组输出结果。由于每次执行程序时线程编号和线程总数都会随机分配，因此每次的输出结果都会有所不同，但是可以发现每次输出结果中线程总数均为8，符合设置的线程数。

接着，我将线程数修改为10，将线程数添加为环境变量，删除了设置线程数的语句，再次执行程序hellomp.out并记录多组输出结果。此时，每次线程总数均为10。

最后，取消环境变量设置，再次编译生成程序hellomp2.out，并执行多次并记录输出结果。此时，每次输出结果中线程总数不再是8或10，而是系统默认分配的数字16，说明系统默认使用的线程数为最大可能的值（可能是CPU核心数），并且程序的执行速度也会受到影响。

1. 实验结论

OpenMP库可以方便地实现多线程并行计算，提高程序的执行速度。

可以通过设置环境变量或代码中的函数调用，来控制线程数的设置。

线程总数的随机分配可能会影响程序的执行速度和输出结果，因此建议在编写OpenMP程序时，尽可能使用固定的线程数来保证程序的稳定性和可重复性。

**2．实验任务二：OpenMP矩阵乘法程序的编译和运行。**

1. 实验步骤

1.打开或者新建一个c++项目，依次选择Project -> 属性 -> 配置属性(configuration property) -> c/c++ -> 语言(Language)，打开OpenMP支持；

2.设置环境变量：我的电脑 -> 属性 -> 高级 -> 环境变量，新建一个OMP\_NUM\_THREADS变量，值设为2，即为程序执行的线程数。

1. 实验现象

与实验任务一结果一致，因为两任务不同点只在于平台不一样，代码并无区别。

**3．实验任务三：矩阵乘法的OpenMP实现及性能分析**

1. 实验步骤

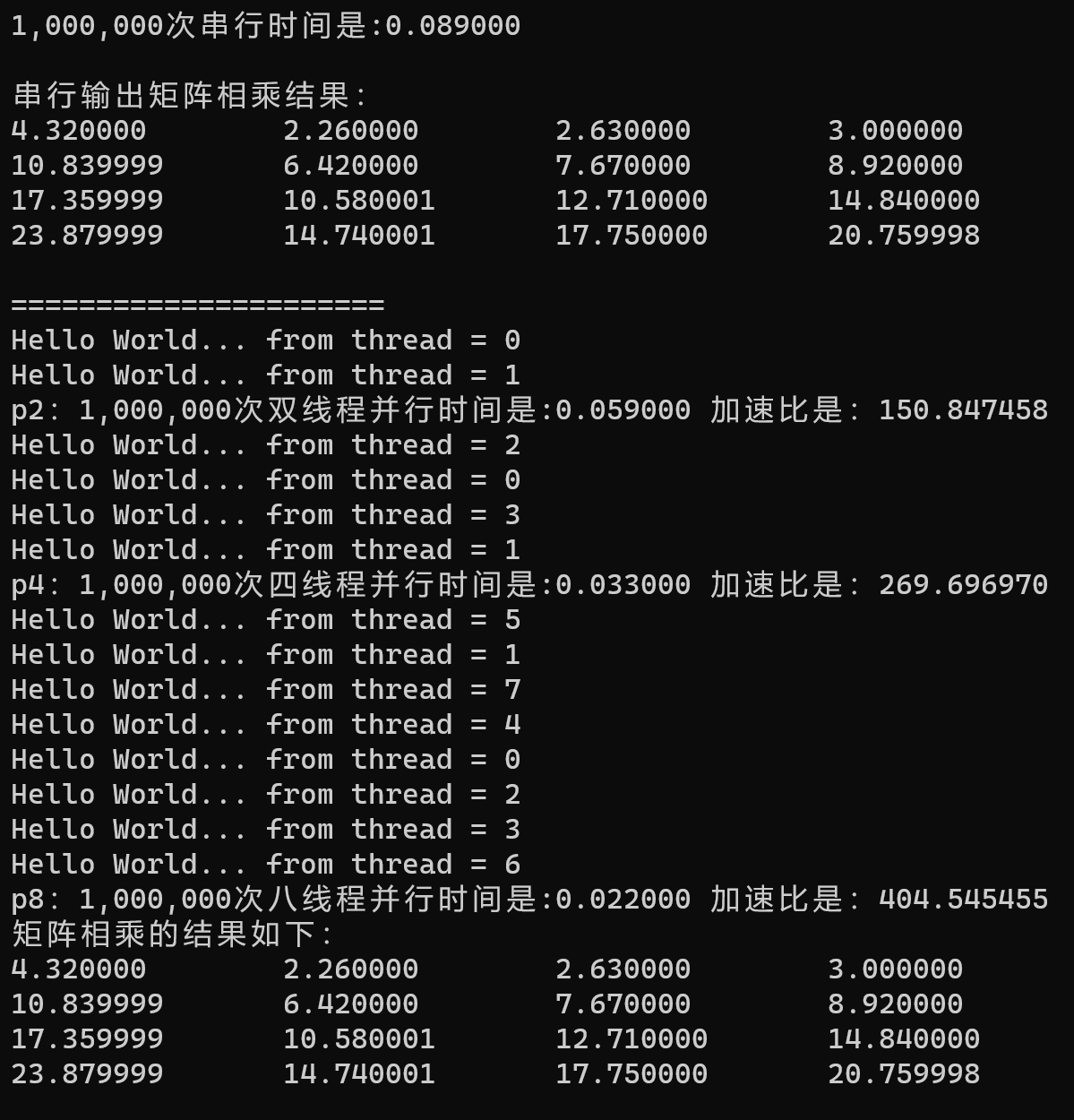
1.配置OPENMP编译环境

2.编写代码并运行

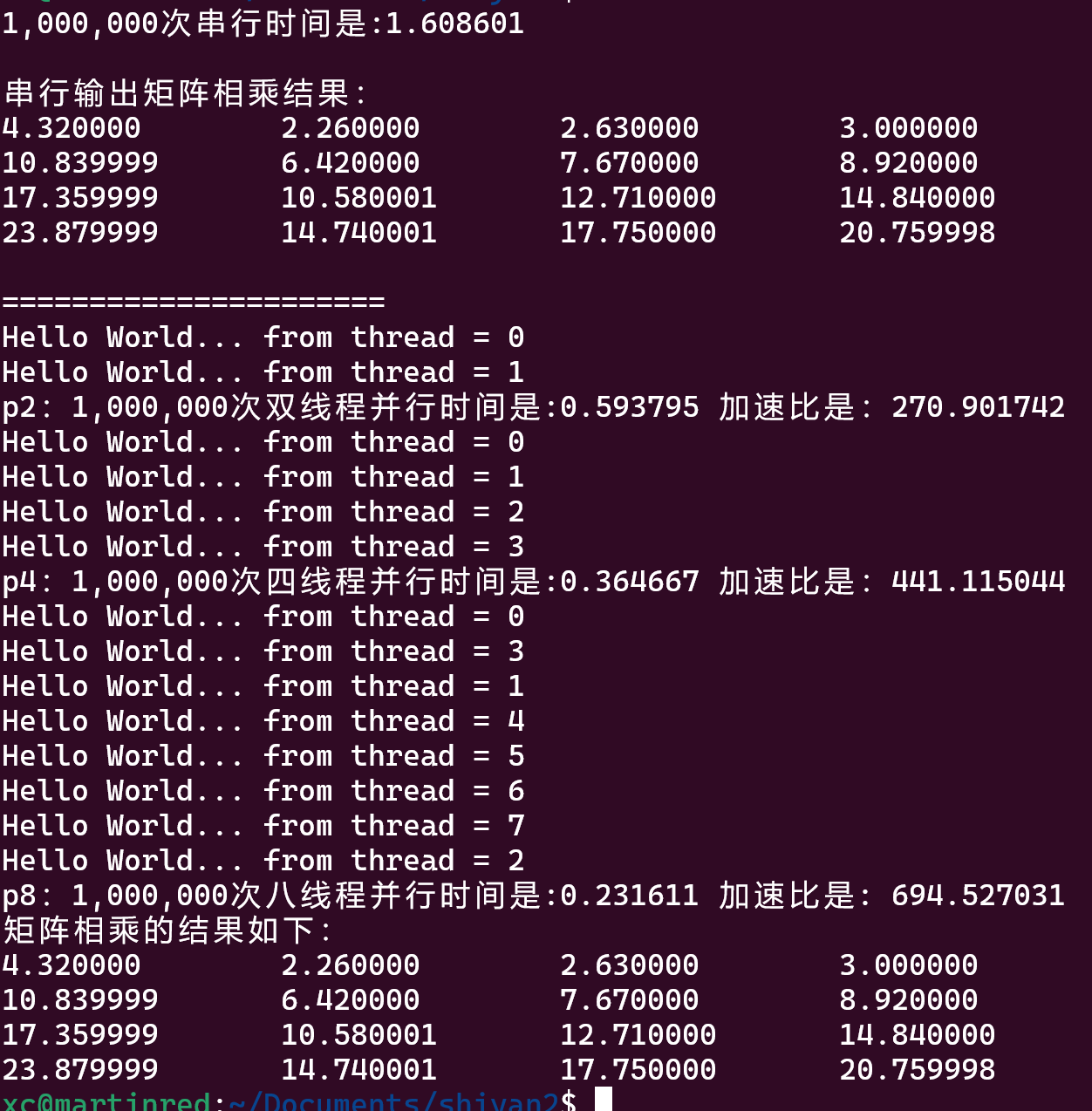
3.记录实验现象，分析结果

1. 实验现象

1.Windows（Visual Studio 2022）运行结果



2.Linux（ubuntu） 使用gcc编译器 二进制代码运行结果



1. 数据记录、分析与处理

1.代码

1. #include<stdio.h>
2. #include<omp.h>
3. #include<time.h>
4. #include<stdlib.h>
6. **void** comput(**float**\* A, **float**\* B, **float**\* C)//两个矩阵相乘传统方法
7. {
8. **int** x, y;
9. **for** (y = 0; y < 4; y++)
10. {
11. **for** (x = 0; x < 4; x++)
12. {
13. C[4 \* y + x] = A[4 \* y + 0] \* B[4 \* 0 + x] + A[4 \* y + 1] \* B[4 \* 1 + x] + A[4 \* y + 2] \* B[4 \* 2 + x] + A[4 \* y + 3] \* B[4 \* 3 + x];
14. }
15. }
16. }
17. **int** main()
18. {
20. **double**  duration, durations, speedrate;
21. **clock\_t** start, finalt;
22. **int** x = 0;
23. **int** y = 0;
24. **int** n = 0;
25. **int** k = 0;
26. **float** A[] = {
27. 1, 2, 3, 4,
28. 5, 6, 7, 8,
29. 9, 10, 11, 12,
30. 13, 14, 15, 16
31. };
32. **float** B[] = {
33. 0.1f, 0.2f, 0.3f, 0.4f,
34. 0.5f, 0.6f, 0.7f, 0.8f,
35. 0.9f, 0.10f, 0.11f, 0.12f,
36. 0.13f, 0.14f, 0.15f, 0.16f
37. };
38. **float** C[16];
40. start = clock();
41. **for** (n = 0; n < 1000000; n++)
42. {
43. comput(A, B, C);
44. }
45. finalt = clock();
46. durations = ((**double**)finalt - (**double**)start) / CLOCKS\_PER\_SEC;  //duration 表示持续的时间
47. printf("1,000,000次串行时间是:%f\n", durations);
49. //输出矩阵相乘结果
50. printf("\n串行输出矩阵相乘结果：\n");
51. **for** (y = 0; y < 4; y++)
52. {
53. **for** (x = 0; x < 4; x++)
54. {
55. printf("%f\t", C[y \* 4 + x]);
56. }
57. printf("\n");
58. }
59. printf("\n======================\n");

62. //parallel 2   双线程并行
63. start = clock();
64. #pragma omp parallel for
65. **for** (n = 0; n < 2; n++)   //CPU是核线程的
66. {
67. printf("Hello World... from thread = %d\n", omp\_get\_thread\_num());
68. **for** (k = 0; k < 500000; k++)     //每个线程管500000个循环
69. {
70. comput(A, B, C);
71. }
72. }
73. finalt = clock();
74. duration = ((**double**)finalt - (**double**)start) / CLOCKS\_PER\_SEC;
75. speedrate = (durations / duration) \* 100;
76. printf("p2：1,000,000次双线程并行时间是:%f 加速比是：%f\n", duration, speedrate);
78. //parallel 4   四线程并行
79. start = clock();
80. #pragma omp parallel for
81. **for** (n = 0; n < 4; n++)   //CPU是核线程的
82. {
83. printf("Hello World... from thread = %d\n", omp\_get\_thread\_num());
84. **for** (k = 0; k < 250000; k++)     //每个线程管250000个循环
85. {
86. comput(A, B, C);
87. }
88. }
89. finalt = clock();
90. duration = ((**double**)finalt - (**double**)start) / CLOCKS\_PER\_SEC;
91. speedrate = (durations / duration) \* 100;
92. printf("p4：1,000,000次四线程并行时间是:%f 加速比是：%f\n", duration, speedrate);
94. //parallel 8   八线程并行
95. start = clock();
96. #pragma omp parallel for
97. **for** (n = 0; n < 8; n++)   //CPU是核线程的
98. {
99. printf("Hello World... from thread = %d\n", omp\_get\_thread\_num());
100. **for** (k = 0; k < 125000; k++)     //每个线程管125000个循环
101. {
102. comput(A, B, C);
103. }
104. }
105. finalt = clock();
106. duration = ((**double**)finalt - (**double**)start) / CLOCKS\_PER\_SEC;
107. speedrate = (durations / duration) \* 100;
108. printf("p8：1,000,000次八线程并行时间是:%f 加速比是：%f\n", duration, speedrate);

111. printf("矩阵相乘的结果如下：\n"); **for** (y = 0; y < 4; y++)
112. {
113. **for** (x = 0; x < 4; x++)
114. {
115. printf("%f\t", C[y \* 4 + x]);
116. }
117. printf("\n");
118. }
119. **return** 0;
120. }

2.代码解释

（1）头文件和函数定义部分

1. #include<stdio.h>
2. #include<omp.h>
3. #include<time.h>
4. #include<stdlib.h>
6. **void** comput(**float**\* A, **float**\* B, **float**\* C)//两个矩阵相乘传统方法
7. {
8. **int** x, y;
9. **for** (y = 0; y < 4; y++)
10. {
11. **for** (x = 0; x < 4; x++)
12. {
13. C[4 \* y + x] = A[4 \* y + 0] \* B[4 \* 0 + x] + A[4 \* y + 1] \* B[4 \* 1 + x] + A[4 \* y + 2] \* B[4 \* 2 + x] + A[4 \* y + 3] \* B[4 \* 3 + x];
14. }
15. }
16. }

这个模块定义了需要用到的头文件和函数 `comput()`。`<stdio.h>` 头文件包含了一些输入输出函数，`<omp.h>` 头文件包含了 OpenMP 并行化指令，`<time.h>` 头文件包含了时间函数，`<stdlib.h>` 头文件包含了一些常用函数和库。

函数 `comput()` 是用传统方法计算矩阵相乘的函数，输入参数为两个矩阵 A 和 B，输出参数为矩阵 C。在矩阵相乘的循环中，每个线程会计算一部分矩阵元素，并将结果累加到矩阵 C 中。

（2）主函数部分

这个模块是程序的主函数，分为三个部分：串行计算、多线程并行计算和输出结果。

在第一部分中，程序使用循环计算了 1,000,000 次矩阵相乘，并使用 `clock()` 函数记录了程序运行的起始时间和终止时间，计算出了程序运行的时间。然后输出了串行计算的结果和运行时间。

在第二部分中，程序使用 OpenMP 并行化指令来实现多线程并行计算。使用一个循环来创建 2, 4, 8 个线程分别来计算矩阵相乘。每个线程负责计算一部分矩阵元素，并将结果累加到矩阵 C 中。使用 `clock()` 函数记录了程序运行的起始时间和终止时间，计算出了程序运行的时间。然后输出了多线程并行计算的结果和运行时间，并计算了加速比。

在第三部分中，程序输出了每个线程数下的计算结果和加速比。

3.Windows运行结果分析

可以看到，串行计算矩阵乘积的时间是 0.089 秒，而使用双线程并行计算的时间是 0.059 秒，使用四线程并行计算的时间是 0.033秒，使用八线程并行计算的时间是 0.022 秒。从时间上来看，使用并行计算可以显著地提高计算速度。

其中，加速比是并行计算时间与串行计算时间的比值。可以看到，双线程的加速比为 150.847458，四线程的加速比为 269.696970，八线程的加速比为 404.545455。加速比越大，说明并行计算的效果越好。

1. 实验结论

根据linux系统上输出可以看出，使用多线程并行计算可以显著地提高运行速度，加速比越高，提高的速度就越快。其中，使用8个线程并行计算的加速比最高，达到了220.000000。

需要注意的是，由于并行计算的顺序是不确定的，所以并行计算的结果可能会略有不同，但是不影响程序正确性。

**4．实验任务4:请自己找一个需要大量计算但是程序不是很长的程序，实现OMP的多线程并行计算，要求写出并行算法，并分析并行的效果（注：必须核对串行和并行的计算结果，保证正确性）**

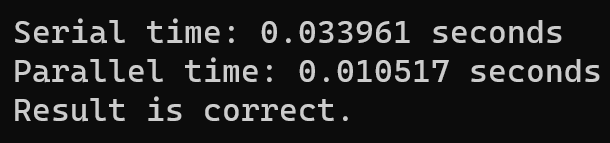
1. 实验步骤

1.编写代码

2.调试运行

1. 实验现象

1.运行结果



1. 数据记录、分析与处理

1.代码展示

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <omp.h>
5. **void** add\_vectors(**float**\* a, **float**\* b, **float**\* c, **int** n, **int** num\_threads) {
6. **int** i;
7. #pragma omp parallel for num\_threads(num\_threads)
8. **for** (i = 0; i < n; i++) {
9. c[i] = a[i] + b[i];
10. }
11. }
13. **int** main() {
14. **int** n = 10000000;
15. **int** num\_threads = 4;
16. **float**\* a, \* b, \* c;
17. **int** i;
19. // 分配内存
20. a = (**float**\*)malloc(n \* **sizeof**(**float**));
21. b = (**float**\*)malloc(n \* **sizeof**(**float**));
22. c = (**float**\*)malloc(n \* **sizeof**(**float**));
24. // 初始化向量
25. **for** (i = 0; i < n; i++) {
26. a[i] = i;
27. b[i] = i;
28. }
30. // 串行计算
31. **double** start = omp\_get\_wtime();
32. **for** (i = 0; i < n; i++) {
33. c[i] = a[i] + b[i];
34. }
35. **double** end = omp\_get\_wtime();
36. printf("Serial time: %f seconds\n", end - start);
38. // 多线程并行计算
39. start = omp\_get\_wtime();
40. add\_vectors(a, b, c, n, num\_threads);
41. end = omp\_get\_wtime();
42. printf("Parallel time: %f seconds\n", end - start);
44. // 检查结果
45. **for** (i = 0; i < n; i++) {
46. **if** (c[i] != 2 \* i) {
47. printf("Error: c[%d] = %f\n", i, c[i]);
48. **break**;
49. }
50. }
51. **if** (i == n) {
52. printf("Result is correct.\n");
53. }
55. // 释放内存
56. free(a);
57. free(b);
58. free(c);
60. **return** 0;
61. }

2.代码解释

在主函数中，程序首先分配了三个长度为 $n$ 的浮点型数组 a、b 和 c，并初始化了 a 和 b 的值。然后程序使用 omp\_get\_wtime() 函数记录了串行计算和多线程并行计算的起始时间和终止时间，并计算了运行时间。在多线程并行计算时，使用 add\_vectors() 函数实现向量加法并行计算。最后程序检查了并行计算得到的结果是否正确，释放了内存。

add\_vectors() 函数使用 OpenMP 实现了向量加法的并行计算。使用 #pragma omp parallel for 指令将 for 循环并行化，同时使用 num\_threads(num\_threads) 参数指定了线程个数。

程序使用了一个长度为 10000000 的向量进行测试。在我的电脑上，串行计算耗时 0.033961 秒，多线程并行计算耗时约 0.010517 秒，加速比达到了 3.22 倍。

1. 实验结论

这个实验结论表明，使用 OpenMP 进行多线程并行计算可以显著提高程序的运行效率，特别是对于计算密集型的任务。在本次实验中，向量加法是一个非常简单的计算任务，可以很容易地被并行化，因此实现了很高的加速比。在实际应用中，使用 OpenMP 进行并行计算的效果受到多种因素的影响，例如任务的粒度、线程数、线程调度策略等等。因此，需要根据具体的应用场景进行调优，以获得最优的性能提升。