上海大学
SHANGHAI UNIVERSITY

计算机系统结构
实验报告

姓　　名	严昕宇
学　　号	20121802
实验序号	实验二
日　　期	2023 年 4 月 3 日

目录

一　实验环境

表 1: 实验环境

实验设备	Lenovo Thinkbook16+ 2022
操作系统	Windows 11 22H2
	Ubuntu 22.04 LTS
开发语言	C
IDE	CLion 2022.3.3
编译器	GCC

二　实验 1 多环境 OpenMP 程序的编译和运行

1　实验目的

- 掌握 OpenMP 并行编程基础
- 掌握在 Linux 平台上编译和运行 OpenMP 程序
- 掌握在 Windows 平台上编译和运行 OpenMP 程序

2　实验步骤

2.1　Linux 下 OpenMP 程序的编译和运行

- **编译环境配置**

GNU 编译器集合是一系列用于语言开发的编译器和库的集合，包括: C、C++、Objective-C、Fortran、Ada 和 Go 等编程语言。很多开源项目，包括 Linux Kernel 和 GNU 工具，都是使用 GCC 进行编译的。

默认的 Ubuntu 软件源包含了一个软件包组, 名称为"build-essential", 它包含了 GNU 编辑器集合, GNU 调试器，和其他编译软件所必需的开发库和工具。想要安装开发工具软件包，以拥有 sudo 权限用户身份或者 root 身份运行下面的命令:

代码 1: 安装命令

```
sudo apt-get update
sudo apt-get install build-essential
```

这个命令将会安装一系列软件包，包括 GCC、G++ 和 Make。安装完成后，即可编译运行程序。

- **编译与运行**

编写串行、并行但代码中不设置线程数（即默认线程），以及代码中设置 8 个线程的 HelloWorld 程序，并用下述命令编译程序。

代码 2: 编译命令

```
gcc -fopenmp -O2 -o helloomp.out helloworld.c
```

编译完成后，输入./helloomp.out 即可运行程序，运行结果如下所示:

```
yanxinyu@Thinkbook16-2022:~$ gcc -fopenmp -O2 -o helloomp.out helloworld.c
yanxinyu@Thinkbook16-2022:~$ ./helloomp.out
Hello World from OMP thread 4
Hello World from OMP thread 1
Hello World from OMP thread 5
Hello World from OMP thread 7
Hello World from OMP thread 0
Number of threads is 8
Hello World from OMP thread 6
Hello World from OMP thread 3
Hello World from OMP thread 2
yanxinyu@Thinkbook16-2022:~$
```

图 1: Linux 下 helloworld.c 运行结果

2.2 Windows 下 OpenMP 程序的编译和运行

由于使用的开发环境是 CLion 和 CMake，因此需要在 CMakeLists 中添加**代码 3**中语句，将 OpenMP 与程序链接，便可执行程序。其中 find_package 指令可以查找 OpenMP 库的位置，target_link_libraries 指令可将 OpenMP 链接到具体项目。

代码 3: CMake 配置

```
1 FIND_PACKAGE(OpenMP REQUIRED)
2 if (OPENMP_FOUND)
3     message("OPENMP FOUND")
4     set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
5     set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
6 endif ()
```

Windows 下的运行结果，如下图所示：

```
E:\Code\CLionProjects\OpenMP\cmake-build-debug\OpenMP.exe
Hello World from OMP thread 1
Hello World from OMP thread 5
Hello World from OMP thread 4
Hello World from OMP thread 6
Hello World from OMP thread 0
Number of threads is 8
Hello World from OMP thread 2
Hello World from OMP thread 7
Hello World from OMP thread 3

进程已结束,退出代码0
```

图 2: Windows 下 helloworld.c 运行结果

3 实验结果分析

多次执行串行、并行但代码中不设置线程数（即默认线程），以及代码中设置 8 个线程的 HelloWorld 程序后，可以发现如下结果：

- **没有设置 OMP_NUM_THREADS 的情况下，系统默认线程数为逻辑处理器数目**

使用任务管理器观察，可以发现实验设备所使用的 AMD Ryzen 7 6800H 处理器内置 8 核 16 线程，与程序运行结果 Number of threads is 16 相同。

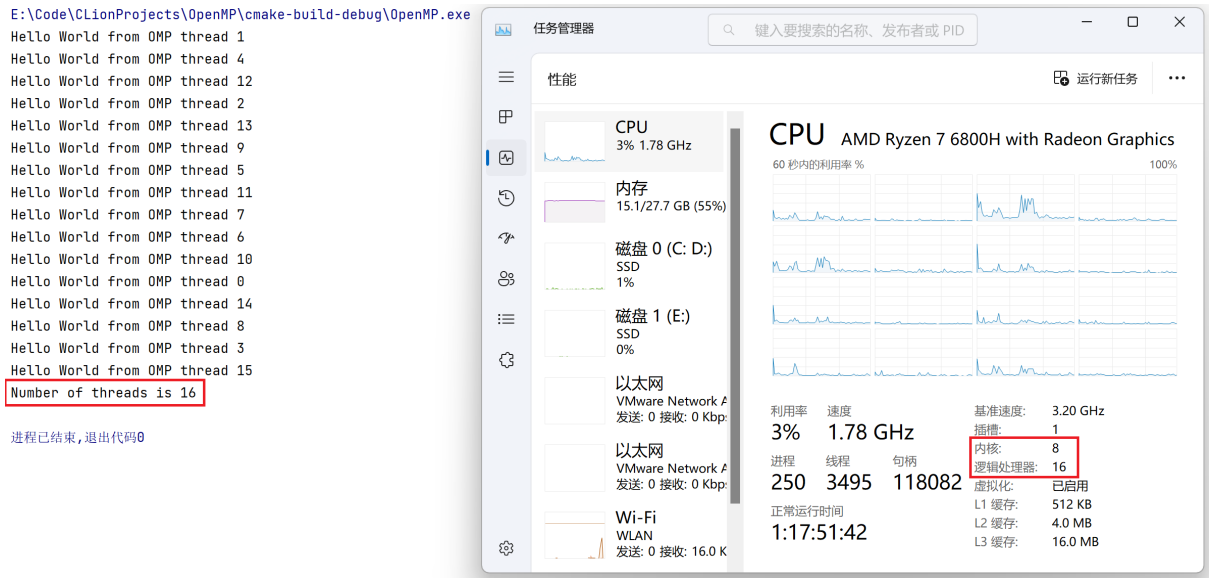

图 3: 默认线程数的运行结果

- **可以调用 omp_set_num_threads() 修改线程数**

为了调整线程数，可以通过调用 omp_set_num_threads 或设置 OMP_NUM_THREADS 环境变量来建立线程数的默认值，而该默认值可以通过在单个 parallel 指令上指定 num_threads 子句进行显式重写。且 omp_set_num_threads() 函数调用优先于 OMP_NUM_THREADS 环境变量。

本实验中，通过 HelloWorld 程序代码中设置 8 个线程，得到如下结果：

图 4: 修改线程数为 8 的运行结果

- **程序每次运行结果存在差异**

虽然线程都是一起开始运行，但实验中每次运行的结果都不一样，这个是因为每次每个线程结束的先后可能不一样的。所以每次运行的结果都是随机的。这是串行程序和并行程序不同的地方：串行程序可以重新运行，结果和之前一样；并行程序却因为执行次序无法控制可能导致每次的结果都不一样，如下图所示：

```
E:\Code\CLionProjects\OpenMP\cmake-build-debug\OpenMP.exe
Hello World from OMP thread 1
Hello World from OMP thread 2
Hello World from OMP thread 0
Number of threads is 8
Hello World from OMP thread 3
Hello World from OMP thread 7
Hello World from OMP thread 6
Hello World from OMP thread 4
Hello World from OMP thread 5
```

图 5: 运行结果 1

```
E:\Code\CLionProjects\OpenMP\cmake-build-debug\OpenMP.exe
Hello World from OMP thread 1
Hello World from OMP thread 0
Number of threads is 8
Hello World from OMP thread 5
Hello World from OMP thread 4
Hello World from OMP thread 6
Hello World from OMP thread 2
Hello World from OMP thread 7
Hello World from OMP thread 3
```

图 6: 运行结果 2

```
E:\Code\CLionProjects\OpenMP\cmake-build-debug\OpenMP.exe
Hello World from OMP thread 1
Hello World from OMP thread 4
Hello World from OMP thread 0
Number of threads is 8
Hello World from OMP thread 3
Hello World from OMP thread 2
Hello World from OMP thread 6
Hello World from OMP thread 7
Hello World from OMP thread 5
```

图 7: 运行结果 3

```
E:\Code\CLionProjects\OpenMP\cmake-build-debug\OpenMP.exe
Hello World from OMP thread 1
Hello World from OMP thread 4
Hello World from OMP thread 0
Number of threads is 8
Hello World from OMP thread 3
Hello World from OMP thread 2
Hello World from OMP thread 5
Hello World from OMP thread 7
Hello World from OMP thread 6
```

图 8: 运行结果 4

三　实验 2 矩阵乘法的 OpenMP 实现及性能分析

1　实验目的

- 用 OpenMP 实现最基本的数值算法“矩阵乘法”
- 掌握 for 编译制导语句
- 对并行程序进行简单的性能调优

2　实验步骤

代码 4: 矩阵乘法程序的主函数

```
int main() {
    int MatrixSize = 1000;
    double *x, *y, *z;
    for (int i = 0; i < 3; i++) {
        // 初始化
        x = (double *) malloc(sizeof(double) * MatrixSize * MatrixSize);
        y = (double *) malloc(sizeof(double) * MatrixSize * MatrixSize);
        z = (double *) malloc(sizeof(double) * MatrixSize * MatrixSize);
        MatrixGenerate(x, MatrixSize);
        MatrixGenerate(y, MatrixSize);
        printf("[%d * %d Matrix]\n", MatrixSize, MatrixSize);
        // 串行计算
        clock_t StartTime, EndTime;
        StartTime = clock();
        SerialMatrixMultiply(x, y, z, MatrixSize);
        EndTime = clock();
        double SerialTime = (double) (EndTime - StartTime) / CLOCKS_PER_SEC;
        printf("Serial Time: %.3lfs\n", SerialTime);
        // 并行计算
        omp_set_num_threads(NUM_THREADS);
        StartTime = clock();
        ParallelMatrixMultiply(x, y, z, MatrixSize);
        EndTime = clock();
        double ParallelTime = (double) (EndTime - StartTime) / CLOCKS_PER_SEC;
        printf("Parallel Time: %.3lfs\n", ParallelTime);
        printf("Speedup: %.3lf\n\n", SerialTime / ParallelTime);
        // 重置参数
        free(x);
        free(y);
        free(z);
        MatrixSize += 1000; // 更新矩阵尺寸
    }
    return 0;
}
```

主函数代码如**代码 4**所示, 其余代码详见**代码 26**，各函数功能如下所述：

- **MatrixGenerate**：使用随机函数生成 n 阶矩阵
- **SerialMatrixMultiply**：实现 n 阶矩阵串行相乘运算
- **ParallelMatrixMultiply**：实现 n 阶矩阵并行相乘运算
- **Main 主函数**：完成不同阶矩阵在串并计算下加速比的计算

3 实验结果分析

3.1 Windows 下矩阵乘法程序的运行

```
E:\Code\CLionProjects\OpenMP\cmake-build-debug\OpenMP.exe
[1000 * 1000 Matrix]
Serial Time: 3.083s
Parallel Time: 0.440s
Speedup: 7.007

[2000 * 2000 Matrix]
Serial Time: 38.081s
Parallel Time: 8.902s
Speedup: 4.278

[3000 * 3000 Matrix]
Serial Time: 246.880s
Parallel Time: 39.953s
Speedup: 6.179
```

图 9: Windows 下矩阵乘法程序的运行

3.2 Linux 下矩阵乘法程序的运行

```
yanxinyu@Thinkbook16-2022:~$ gcc -fopenmp -O2 -o Matrix.out Matrix.c
yanxinyu@Thinkbook16-2022:~$ ./Matrix.out
[1000 * 1000 Matrix]
Serial Time: 2.084s
Parallel Time: 3.102s
Speedup: 0.672

[2000 * 2000 Matrix]
Serial Time: 71.733s
Parallel Time: 99.752s
Speedup: 0.719

[3000 * 3000 Matrix]
Serial Time: 333.377s
Parallel Time: 408.958s
Speedup: 0.815
```

图 10: Linux 下矩阵乘法程序的错误运行

注意到在 Linux 环境下，程序运行结果中的加速比均小于 1。经过查找资料后发现，在 Linux 中，前后两个 clock() 函数相减产生的并不是实际耗费的时间，而是将各个线程的执行时间累加了，导致看起来好像串行比并行还要“快”，所以上述结果是不正确的。如果需要正确计时，可以使用 time 库中的 clock_gettime() 函数或 OpenMP 的 omp_get_wtime() 函数来计时。

clock_gettime() 函数是基于 Linux C 语言的时间函数，可以用于计算精度和纳秒，函数原型为：

代码 5: clock_gettime() 函数

```
1 int clock_gettime(clockid_t clk_id, struct timespec *tp);
```

故修改程序，得到结果如**图 11**所示，OpenMP 并行下的加速效果得以体现。

```
yanxinyu@Thinkbook16-2022:~$ gcc -fopenmp -O2 -o Matrix_Linux.out Matrix_Linux.c
yanxinyu@Thinkbook16-2022:~$ ./Matrix_Linux.out
[1000 * 1000 Matrix]
Serial Time: 1.438s
Parallel Time: 0.742s
Speedup: 1.937

[2000 * 2000 Matrix]
Serial Time: 26.738s
Parallel Time: 15.099s
Speedup: 1.771

[3000 * 3000 Matrix]
Serial Time: 235.255s
Parallel Time: 82.111s
Speedup: 2.865
```

图 11: Linux 下矩阵乘法程序的运行

可以发现，虚拟机中 OpenMP 运行的加速比低于 Windows，这可能是由于虚拟机仅分配了 4 核逻辑处理器与 2GB 内存，在矩阵空间变大时，内存空间的不足导致进程换入换出产生的频率和开销增加。

3.3 线程数和运行时间的关系及原因

总体而言，在两种运行环境下，加速比皆随着矩阵的规模的增大而增大。当矩阵较小时，并行计算下的程序运行时间甚至可能会大于串行运行时间，这是因为此时 OpenMP 本身的开销在此相比并行减少的时间占比更大。

实验数据如下所示：

表 2: 1000×1000 矩阵乘法

Thread Num	Serial Time	Parallel Time	Speedup
1	3.072s	2.935s	1.047
2	2.872s	1.460s	1.967
4	2.880s	0.817s	3.525
8	2.841s	0.446s	6.370
16	2.896s	0.264s	10.970
32	2.868s	0.269s	10.662

表 3: 2000 × 2000 矩阵乘法

Thread Num	Serial Time	Parallel Time	Speedup
1	31.488s	31.688s	0.994
2	30.687s	15.847s	1.936
4	31.503s	8.749s	3.601
8	31.001s	4.902s	6.324
16	31.299s	3.405s	9.192
32	31.171s	3.541s	8.803

表 4: 3000 × 3000 矩阵乘法

Thread Num	Serial Time	Parallel Time	Speedup
1	120.359s	121.550s	0.990
2	118.769s	60.563s	1.961
4	116.304s	33.012s	3.523
8	119.106s	20.038s	5.944
16	119.753s	14.107s	8.489
32	119.795s	14.876s	8.053

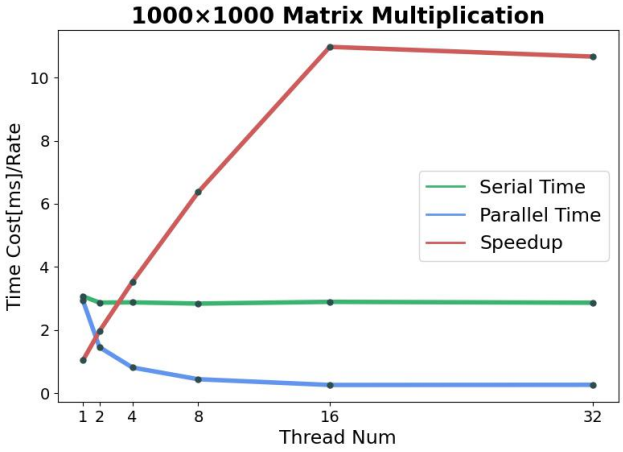

图 12: 1000 × 1000 矩阵乘法

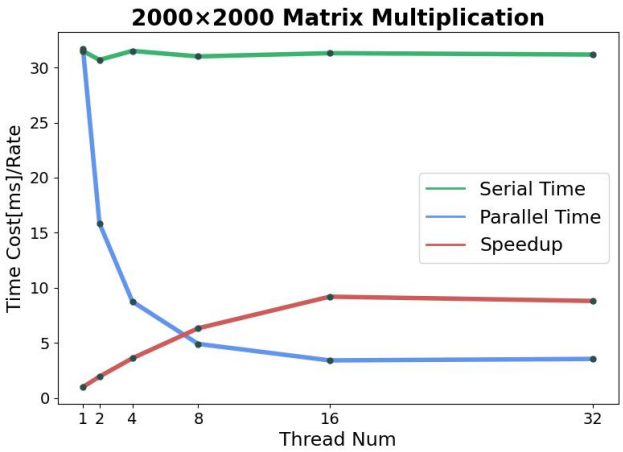

图 13: 2000 × 2000 矩阵乘法

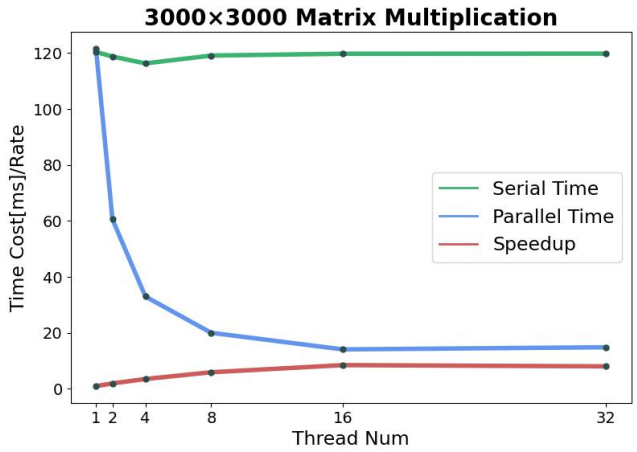

图 14: 3000 × 3000 矩阵乘法

查询资料后可知背后原因，OpenMP 的 parallel region 结束时，线程之间需要同步：即主线程需要等待所有其他线程完成工作之后才能继续，这个过程可以称做 barrier。一个简单的 barrier 的实现如下：

代码 6: Barrier

```
_Atomic int finished = 0;
int num_threads = N;
void barrier()
{
  atomic_add (&finished, 1);
  while (atomic_read(&finished) < num_threads);
  return;
}
```

在支持 compare-and-swap 指令的硬件平台上，atomic_add 可以用 cas 指令实现。在上面的 barrier 代码中，假定 threads 被不同的 CPU core 执行，那么一定会出现 CAS cache miss。假设只有两个 threads，分别被 CPU1 和 CPU2 执行。thread1 执行 atomic_add 时，假设正好满足 best-case CAS，即存放 finished 的 cache line 正好在 CPU1 的 local cache 里。那么在 thread2 执行 atomic_add 时就会发生 CAS cache miss，即 CPU2 要从 CPU1 的 cache 中先获得 finished 的 cache line 数据，然后再写入自己的结果。

CAS cache miss 需要 500 个 CPU cycles，现代体系结构的 CPU 每个 cycle 可以执行不止 1 条指令，所以 500 个 cycle 理想情况下可以执行 500 条以上的指令。所以，就算只有两个 threads，并且 threads 之间完全没有其他通信，如果 parallel region 单线程执行时需要不到 500 个 cycles，那么由于 barrier 的开销，OpenMP 多线程就会比单线程还慢。上述 barrier 只是一个简化了很多的示例，OpenMP 一个 parallel region 还有很多其他额外开销。另外，如果 threads 之间还要共享其他数据，并且访问共享数据时需要“锁”的保护，一次锁操作也需要至少一条 cas 指令，这又增加了额外的开销。

然而，随着矩阵规模的增大，并行带来的运算优化变多，在 OpenMP 本身开销相对固定的情况下，并行运算的优越性便越为明显。

四　实验 3 OpenMP 实例估算 PI 值

1　实验目的

- 调试 PPT 中串行算法及四种并行程序
- 检查四种并行程序是否有效提高加速比

2　实验步骤

- **矩形法则的数值积分方法估算 PI 的值**

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \approx \frac{1}{N} \sum_{i=1}^{N} f\left(\frac{i}{N} - \frac{1}{2N}\right) = \frac{1}{N} \sum_{i=1}^{N} f\left(\frac{i-0.5}{N}\right)$$

上式将区间 [0,1] 划分为尽可能小的区间，将区间中值对应的函数值乘以区间长度，即用矩形面积近似曲边梯形面积，再多次累加求和, 便可得到 π 的积分近似值。

在本实验程序中，将区间 [0,1] 划分成了 1000000000 个区间，并行线程数设置为 4，分别使用了串行模式下及 PPT 中提及的的四种并行模式下的程序，计算得到 π 的结果及对应程序的运行时间。不同模式下的计算代码如**代码 7**、**代码 8**、**代码 9**、**代码 10**、**代码 11**所示，完整代码见**代码 28**。

代码 7: 串行程序

```
double SerialPI(int Steps) {
    double sum = 0.0;
    double Step = 1.0 / (double) Steps;
    for (int i = 0; i < Steps; i++) {
        double x = (i + 0.5) * Step;
        sum = sum + 4.0 / (1.0 + x * x);
    }
    return Step * sum;
}
```

代码 8: 使用并行域并行化的程序

```
double ParallelPI_1(int Steps) {
    int i;
    double sum[THREAD_NUM * 16]; // 目的：减少False Sharing，实现加速比>1
    double result = 0.0;
    double Step = 1.0 / (double) Steps;
    omp_set_num_threads(THREAD_NUM);
#pragma omp parallel private(i)
    {
        double x;
        int id = omp_get_thread_num();
        for (i = id, sum[id] = 0.0; i < Steps; i += THREAD_NUM) {
            x = (i + 0.5) * Step;
            sum[id * 16] += 4.0 / (1.0 + x * x);
        }
    }
    for (i = 0; i < THREAD_NUM; ++i) {
```

```
        result += sum[i * 16];
    }
    return result * Step;
}
```

代码 9: 使用共享任务结构并行化的程序

```
double ParallelPI_2(int Steps) {
    double sum[THREAD_NUM * 16]; // 目的：减少False Sharing，实现加速比>1
    double result = 0.0;
    double Step = 1.0 / (double) Steps;
    omp_set_num_threads(THREAD_NUM);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        sum[id] = 0.0;
#pragma omp for
        for (int i = 0; i < Steps; i++) {
            double x = (i + 0.5) * Step;
            sum[id * 16] += 4.0 / (1.0 + x * x);
        }
    }
    for (int i = 0; i < THREAD_NUM; ++i) {
        result += sum[i * 16];
    }
    return result * Step;
}
```

代码 10: 使用 private 子句和 critical 部分并行化的程序

```
double ParallelPI_3(int Steps) {
    int i;
    double sum;
    double result = 0.0;
    double Step = 1.0 / (double) Steps;
    omp_set_num_threads(THREAD_NUM);
#pragma omp parallel private(sum, i)
    {
        int id = omp_get_thread_num();
        sum = 0.0;
        for (i = id; i < Steps; i += THREAD_NUM) {
            double x = (i + 0.5) * Step;
            sum += 4.0 / (1.0 + x * x);
        }
#pragma omp critical
        result += sum;
    };
    return result * Step;
```

```
}
```

代码 11: 使用并行归约并行化的程序

```
double ParallelPI_4(int Steps, int ThreadNum) {
    double sum;
    double Step = 1.0 / (double) Steps;
    omp_set_num_threads(ThreadNum);
#pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < Steps; i++) {
        double x = (i + 0.5) * Step;
        sum += 4.0 / (1.0 + x * x);
    }
    return Step * sum;
}
```

3 实验结果分析

分析实验结果，可以发现：OpenMP 估算 PI 值程序中，四种并行程序相对于串行程序的加速比有显著差异。其中，使用 private 子句和 critical 子句部分并行化的程序以及使用并行规约并行化的加速效果（后两种并行程序），明显优于单纯使用并行域并行化的程序以及使用共享任务并行化的程序（前两种并行程序），**甚至未修改前的前两种并行程序的加速比小于 1**。具体数据如**图 15**所示。

```
======Serial======
PI Result: 3.1415926535899708
Time: 2.7580000000000000s

======Parallel 1======
PI Result: 3.1415926535897682
Time: 14.4340000152587891s

======Parallel 2======
PI Result: 3.1415926535898211
Time: 14.6419999599456787s

======Parallel 3======
PI Result: 3.1415926535898211
Time: 0.7409999370574951s

======Parallel 4======
PI Result: 3.1415926535898211
Time: 0.7409999370574951s
```

图 15: 串行程序及 4 种并行程序修改前的执行结果

3.1 前两种并行程序的加速比异常分析

此处将前两种并行程序的中每一个线程的运行时间都打印出来，如**图 16**所示，可以发现：单线程运行时间和总进程运行时间几乎一致，因此并不存在线程拖垮整体进程时间，导致加速失败的问题。

通过查找资料后发现，产生此现象的原因是因为 OpenMP 的 **False Sharing 问题**，接下来对此问题进行介绍与分析。

```
======Parallel 1======
Thread 3: 14.074000
Thread 0: 14.081000
Thread 2: 14.095000
Thread 1: 14.111000
PI Result: 3.1415926535897682
Time: 14.1130001544952393s

======Parallel 2======
Thread 1: 16.325000
Thread 2: 16.325000
Thread 3: 16.324000
Thread 0: 16.324000
PI Result: 3.1415926535898211
Time: 16.3329999446868896s
```

图 16: 前两种并行程序修改前各线程的执行时间

- **False Sharing**

根据英文参考文献，对此 False Sharing 有如下定义：In symmetric multiprocessor (SMP) systems, each processor has a local cache.The memory system must guarantee cache coherence.False sharing occurs when threads on different processors modify variables that reside on the same cache line.This invalidates the cache line and forces an update, which hurts performance。翻译为中文，即：在多核系统上，每一个处理器都有自己的缓存。计算机体系结构中，必须要保证内存数据的一致性，当某个处理器对属于的它自己的缓存的变量执行更新操作时，糟糕的是那个变量所在的块也被其他的核放在了缓存里面，那么就会发生 False Sharing。

- **Cache Line**

CPU 处理指令时，由于“Locality of Reference”（局部性原理）原因，需要决定哪些数据需要加载到 CPU 的缓存中，以及如何预加载。因为不同的处理器有不同的规范，导致这部分工作具有不确定性。在加载的过程中，涉及到一个非常关键的术语：Cache Line。Cache Line 是能被 Cache 处理的内存 Chunks，Chunk 的大小即为 Cache Line Size，典型的大小为 32、64 及 128 Bytes。

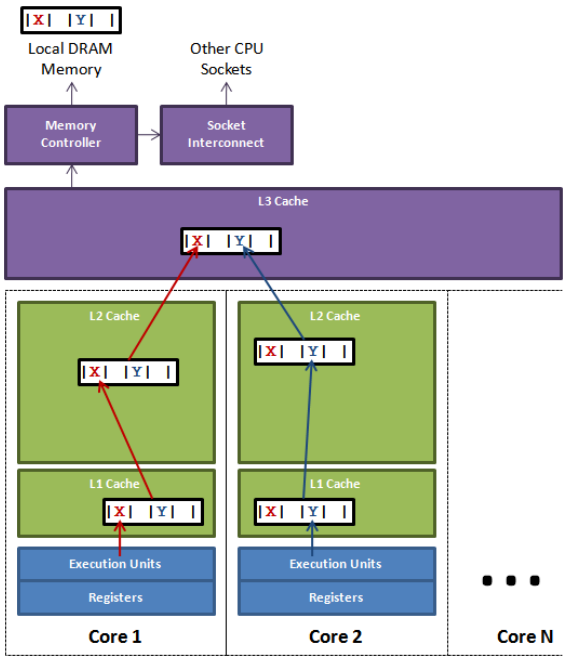

图 17: L1、L2、L3 Cache 的位置与关系

Cache 分为如下几级：

1) L1 Cache：L1 Cache 是 CPU 第一层高速缓存，分为数据缓存和指令缓存。内置的 L1 高速缓存的容量和结构对 CPU 的性能影响较大，不过高速缓冲存储器均由静态 RAM 组成，结构较复杂，在 CPU 管芯面积不能太大的情况下，L1 级高速缓存的容量不可能做得太大。

2) L2 Cache：由于 L1 级高速缓存容量的限制，为了再次提高 CPU 的运算速度，在 CPU 外部放置一高速存储器，即二级缓存。工作主频比较灵活，可与 CPU 同频，也可不同。CPU 在读取数据时，先在 L1 中寻找，再从 L2 寻找，然后是内存，在后是外存储器。所以 L2 对系统的影响也不容忽视。

3) L3 Cache：现在的都是内置的。而它的实际作用即是，L3 缓存的应用可以进一步降低内存延迟，同时提升大数据量计算时处理器的性能。降低内存延迟和提升大数据量计算能力对游戏都很有帮助。而在服务器领域增加 L3 缓存在性能方面仍然有显著的提升。比方具有较大 L3 缓存的配置利用物理内存会更有效，故它比较慢的磁盘 I/O 子系统可以处理更多的数据请求。具有较大 L3 缓存的处理器提供更有效的文件系统缓存行为及较短消息和处理器队列长度。

增加多级缓存的好处就是提高命中率，三级缓存的机器总体的命中率大约为 95%，也就是大约有 5% 的数据从内存中读取，这样就大大提高了 CPU 的使用率。

- **False Sharing 的产生过程**

以**图 17**为例子，其中的 Core1 拥有包含 X 和 Y 的数据块，Core1 会将其标记为 “Exclusive”，即专用。当 Core2 加载了相同的数据块后（操作系统的各个核之间的缓存调度是独立的），Core1 会将相同的块标记为 “Shared”，那么 Core2 里面的在加载的时候就会被标记 “Shared”。

此时，如果 Core1 要对 X 进行修改（如果 Core2 也要对 X 进行修改，那么会发生冲突，需要原子操作进行隔离，否则会发生错误），Core1 就对 XY 数据块标记为 “Modified”，并发送 “Invalid” 通知其他拥有相同数据块的处理器 Core。如果此时 Core2 要使用 XY 数据块，那么被 Core1 得知之后，Core1 就把它自己 Cache 里面的 XY 数据块回写到内存中，并将 Core1 Cache 里面的 XY 数据块重新标记为 “Shared”，而此时 Core2 Cache 里的 XY 数据块是 “Invalid”，即会产生一个 Miss。此时需要重新加载 XY 数据块，加载完成后将其标记为 “Shared”。

- **False Sharing 的解决方案**

参考 Intel 发表的技术文献Avoiding and Identifying False Sharing Among-Threads，三种解决方案如下：

1) 字节对齐：因为缓存遵循”Locality of Reference“, 所以只要避免多个处理器 Cache 里面的数据块尽量不要” Shared “就行了。如果上述的例子 Core1 只有 X 数据块, Ccore2 只有 Y 数据块, 那么就不会存在 False Sharing。在 Windows 的程序中使用 __declspec(align(64)) 加在变量核结构体之前，就能把变量或者结构体扩展成 64 个字节的数据，如果 Cache 的一个数据块是 64byte 的话，就只会加载一个变量，那么就不会发生 False Sharing 了，不过此方法会造成一定的资源浪费。而 Linux 中可以使用 __attribute__((aligned(64)))，亦可解决此问题。

2) 结构体填充：类似于方法 1，不过此方法是自己手动填充数据块。以下是一个将结构体填充成 64 Byte 的例子：

代码 12: 结构体填充

```
struct ThreadParams {
// For the following 4 variables: 4*4 = 16 bytes
    unsigned long thread_id;
    unsigned long v; // Frequent read/write access variable
    unsigned long start;
    unsigned long end;
// expand to 64 bytes
// (4 unsigned long variables + 12 padding)*4 = 64
```

```
    int padding[12];
}
```

3) 数据线程私有化：也就是把可能会产生 False Sharing 的数据块对每个线程 Ccopy 一份，并重新命名，作为每个线程私有的东西，并在线程的最后一步同步到主线程中去。

- **修改后的实验结果**

根据以上分析，将实验中的 sum 数组大小修改为 **sum[THREAD_NUM * 16]**（扩大 16 倍），具体代码如**代码 8**、**代码 9**。修改后的实验结果如下图所示：

```
======Serial======
PI Result: 3.1415926535899708
Time: 2.8039999999999998s

======Parallel 1======
PI Result: 3.1415926535897682
Time: 0.9470000267028809s

======Parallel 2======
PI Result: 3.9269908167375260
Time: 0.8180000782012939s
```

图 18: 前两种并行程序修改后的执行时间

对比**图 16**可以发现，前两种并行程序的执行时间显著缩短，且加速比已 >1，实现了并行加速的效果。

3.2 加速比与线程数的关系

此处选择使用并行规约并行化的程序（第四种程序）进行进一步分析。修改使用并行规约并行化的程序，通过将线程数作为并行程序的函数参数，在外部以循环变量的形式引入，以测试线程数为 1、2、4、8···256、512、1024、2048 时并行程序的加速比，分析并行程序的加速比与线程数的关系。外部循环代码如**代码 13**所示。

代码 13: 外部循环代码

```
for (int i = 1; i < 2048; i *= 2) {
    ParallelStartTime = omp_get_wtime();
    PI = ParallelPI_4(Steps, i);
    ParallelEndTime = omp_get_wtime();
    double t4 = ParallelEndTime - ParallelStartTime;
    printf("[Thread Num:%d]\n", i);
    printf("PI Result: %.16lf\nTime: %.16fs\nSpeedup: %.6f\n", PI, t4, t0 / t4);
}
```

分别在 16 核 CPU（Windows）和 4 核 CPU（Linux）环境下运行上述程序，实验结果如**图 19**、**图 22**所示：

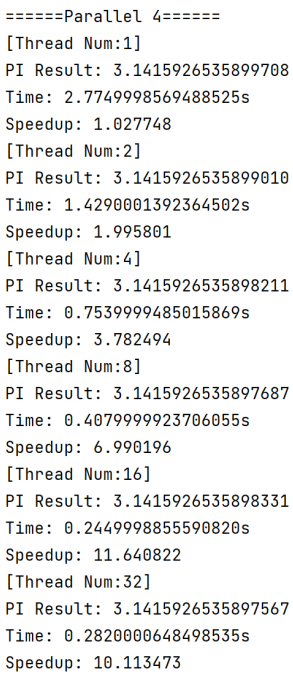

图 19: Windows 下估算 PI 程序的运行结果-1

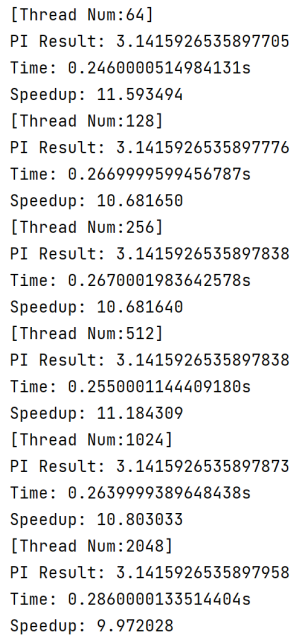

图 20: Windows 下估算 PI 程序的运行结果-2

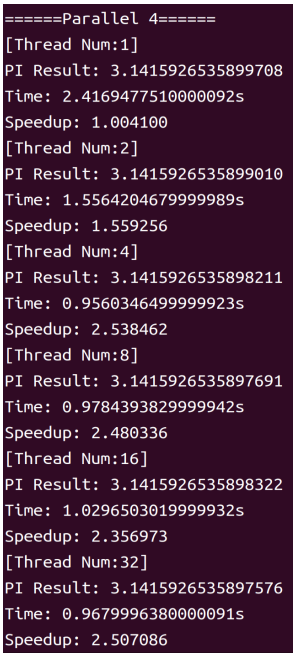

图 21: Linux 下估算 PI 程序的运行结果-1

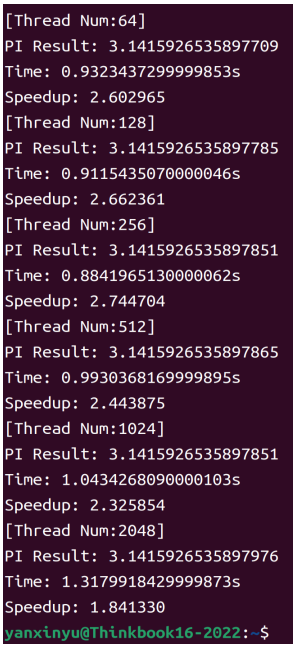

图 22: Linux 下估算 PI 程序的运行结果-2

为了便于分析实验结果，绘制结果数据对应的折线图，如**图 23**、**图 24**所示。随着线程数的增大，并行程序的执行时间在前期显著降低，与此同时加速比也显著提升，以 CPU 核数为分界，并行程序的执行时间及加速比便逐渐趋于稳定甚至略有下降。Linux 的 4 核环境下并行加速比的下降趋势相较于 Windows 的 16 核环境更为明显。

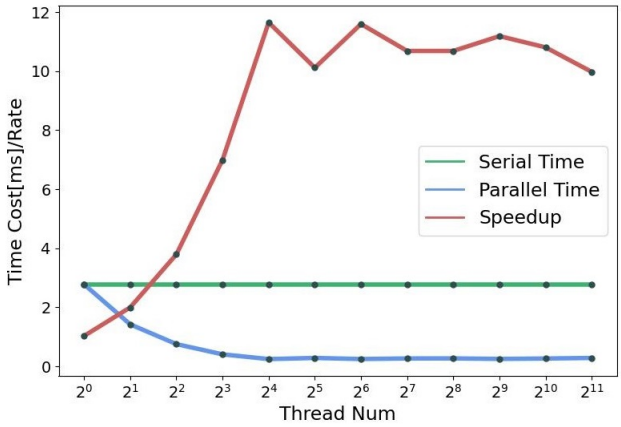

图 23: Windows 下程序的运行结果折线图

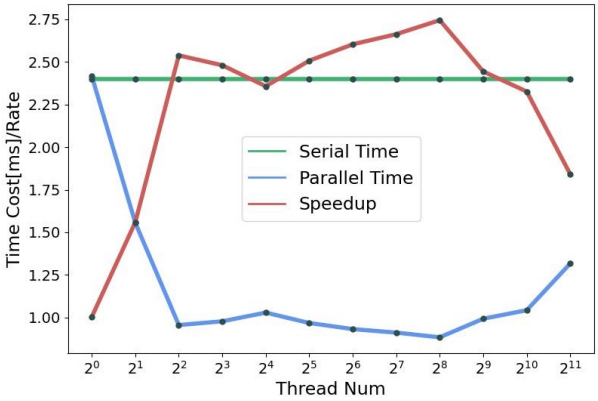

图 24: Linux 下程序的运行结果折线图

五　实验 4 OpenMP 变量的可共享性——private、shared 子句

1　private 子句

1.1　private 子句

private 子句可以将变量声明为线程私有。声明称线程私有变量以后，每个线程都有一个该变量的副本，线程之间不会互相影响，其他线程无法访问其他线程的副本。原变量在并行部分不起任何作用，也不会受到并行部分内部操作的影响。

代码 14: private 子句

```
int main(int argc, char *argv[]) {
    int i = 20;
#pragma omp parallel for private(i)
    for (i = 0; i < 10; i++) {
        printf("i = %d\n", i);
    }
    printf("outside i = %d\n", i);
    return 0;
}
```

```
i = 2
i = 1
i = 4
i = 5
i = 3
i = 6
i = 7
i = 0
i = 9
i = 8
outside i = 20
```

图 25: private 子句运行结果

1.2　firstprivate 子句

private 子句不能继承原变量的值，但是有时我们需要线程私有变量继承原来变量的值，这样我们就可以使用 firstprivate 子句来实现。

代码 15: firstprivate 子句

```
int main(int argc, char *argv[]) {
    int t = 20, i;
#pragma omp parallel for firstprivate(t)
    for (i = 0; i < 5; i++) {
        // 次数t被初始化为20
```

```
        t += i;
        printf("t = %d\n", t);
    }
    // 此时t=20
    printf("outside t = %d\n", t);
    return 0;
}
```

```
t = 23
t = 20
t = 21
t = 24
t = 22
outside t = 20
```

图 26: firstprivate 子句运行结果

1.3 lastprivate 子句

除了在进入并行部分时需要继承原变量的值外，有时我们还需要再退出并行部分时将计算结果赋值回原变量，lastprivate 子句就可以实现这个需求。

需要注意的是，根据 OpenMP 规范，在循环迭代中，是最后一次迭代的值赋值给原变量；如果是 section 结构，那么是程序语法上的最后一个 section 语句赋值给原变量。如果是类变量作为 lastprivate 的参数时，需要一个缺省构造函数，除非该变量也作为 firstprivate 子句的参数；此外还需要一个拷贝赋值操作符。

代码 16: lastprivate 子句

```
int main(int argc, char *argv[]) {
    int t = 20, i;
#pragma omp parallel for firstprivate(t), lastprivate(t)
    for (i = 0; i < 5; i++) {
        t += i;
        printf("t = %d\n", t);
    }
    printf("outside t = %d\n", t);
    return 0;
}
```

```
t = 21
t = 22
t = 23
t = 24
t = 20
outside t = 24
```

图 27: lastprivate 子句运行结果

需要注意的是，lastprivate 必须要搭配 firstprivate 一起使用。

1.4 threadprivate 子句

threadprivate 子句可以将一个变量复制一个私有的拷贝给各个线程，即各个线程具有各自私有的全局对象。

代码 17: threadprivate 子句

```
int g = 0;
#pragma omp threadprivate(g)

int main(int argc, char *argv[]) {
    int t = 20, i;
#pragma omp parallel
    {
        g = omp_get_thread_num();
    }
#pragma omp parallel
    {
        printf("thread id: %d g: %d\n", omp_get_thread_num(), g);
    }
    return 0;
}
```

```
thread id: 7 g: 7
thread id: 14 g: 14
thread id: 11 g: 11
thread id: 2 g: 2
thread id: 8 g: 8
thread id: 1 g: 1
thread id: 6 g: 6
thread id: 4 g: 4
thread id: 15 g: 15
thread id: 13 g: 13
thread id: 5 g: 5
thread id: 9 g: 9
thread id: 3 g: 3
thread id: 12 g: 12
thread id: 10 g: 10
thread id: 0 g: 0
```

图 28: threadprivate 子句运行结果

2 shared 子句

相对 private 子句，存在 shread 子句。share 子句可以将一个变量声明成共享变量，并且在多个线程内共享。需要注意的是，在并行部分进行写操作时，要求共享变量进行保护，否则不要随便使用共享变量，尽量将共享变量转换为私有变量使用。

代码 18: shared 子句

```
int main(int argc, char *argv[]) {
    int t = 20, i;
#pragma omp parallel for shared(t)
    for (i = 0; i < 10; i++) {
        if (i % 2 == 0)
            t++;
        printf("i = %d, t = %d\n", i, t);
    }
    return 0;
}
```

```
i = 3, t = 20
i = 1, t = 20
i = 2, t = 21
i = 4, t = 22
i = 6, t = 23
i = 9, t = 23
i = 5, t = 23
i = 0, t = 24
i = 7, t = 24
i = 8, t = 25
```

图 29: shared 子句运行结果

这里使用对 t 共享，实际上程序变成了串行。

3 OpenMP 中 private、share 的隐式规则

上面是通过 private、shared 子句显式地指定数据在线程之间的共享性，当我们不明确指定的时候，规则又是怎么样的呢？其实，OpenMP 有一组规则，可以推论出变量的数据共享属性。

例如，考虑以下代码：

代码 19: 实例代码 1

```
int i = 0;
int n = 10;
int a = 7;

#pragma omp parallel for
for (i = 0; i < n; i++){
    int b = a + i;
    ...
}
```

代码中有四个变量：i，n，a 与 b。通常在并行区域之外声明的变量的数据共享属性。因此，n 和 a 是共享变量。但是，循环迭代变量默认情况下是私有的。因此，i 是私有的。在并行区域内本地声明的变量

是私有的。因此 b 是私有的。

但是，更建议在并行区域内声明循环迭代变量。在这种情况下，此变量是私有的。修改后的代码如下所示：

代码 20: 实例代码 2

```
int n = 10;                 // shared
int a = 7;                  // shared

#pragma omp parallel for
for (int i = 0; i < n; i++) // i private
{
    int b = a + i;          // b private
    ...
}
```

4 通过 default 指定变量共享模式

default 指定并行区域内变量的属性，C++ 的 OpenMP 中 default 的参数只能为 shared 或 none，对于 Fortran，可以有 private 等参数，具体参考手册。

4.1 default(shared)

其表示并行区域内的共享变量在不指定的情况下都是 shared 属性。例如，在下面的代码中：

代码 21: 实例代码 3

```
int a, b, c, n;
...
#pragma omp parallel for default(shared)
for (int i = 0; i < n; i++){
    // using a, b, c
}
```

a、b、c 与 n 被共享变量。

default(shared) 子句的另一种用法是指定大多数变量的数据共享属性，然后另外定义私有变量。这种用法如下所示：

代码 22: 实例代码 4

```
int a, b, c, n;

#pragma omp parallel for default(shared) private(a, b)
for (int i = 0; i < n; i++){
    // a and b are private variables
    // c and n are shared variables
}
```

4.2 default(none)

其表示必须显式指定所有共享变量的数据属性，否则会报错，除非变量有明确的属性定义（如循环并行区域的循环迭代变量只能是私有的），如以下代码：

代码 23: 实例代码 5

```
int n = 10;
std::vector<int> vector(n);
int a = 10;

#pragma omp parallel for default(none) shared(n, vector, a)
for (int i = 0; i < n; i++){
    vector[i] = i * a;
}
```

4.3 两种模式对循环变量和在并行区域内声明的变量的影响

在 default(shared) 模式下，所有的并行区域外的变量都是 shared 的模式，对于循环变量和在并行区域内声明的变量，不受该 defautl(shared) 模式的影响，还是 private 的，除非指定为 shared 模式的；

在 default(none) 情况下，它们同样不受影响，该模式不会强制对循环变量和并行区域内变量进行指定，但需要强制对外部变量进行指定。比如下面，i 不会要求强制指定，默认私有，但是不指定 a 就会报错。

代码 24: 实例代码 6

```
int i = 20; int a = 1;
#pragma omp parallel for default(none) private(a)
for (i = 0; i < 10; i++){
    printf("i = %d\t%d\n", i, a);
}
printf("outside i = %d\n", i);
return 0;
```

5 总结

根据以上内容，可以总结出以下两个准则：

- **第一条准则**：始终使用 default(none) 子句编写并行区域。该准则迫使明确考虑所有变量的数据共享属性。
- **第二条准则**：在可能的情况下在并行区域内声明私有变量。该准则提高了代码的可读性，并使代码更清晰。

六　实验感想

OpenMP 是一种用于共享内存并行系统的多线程程序设计方案，支持的编程语言包括 C、C++ 和 Fortran，其提供了对并行算法的高层抽象描述，特别适合在多核 CPU 机器上的并行程序设计。使用 OpenMP 时，仅需要通过预编译指令以及一些简单的库函数，就可以让程序完成复杂的并行化运算，这是 OpenMP 最大的优点。它降低了开发并行程序的门槛，为程序员提供了便利。目前，在多核处理器广泛普及的背景下，仅需通过 OpenMP 并行化的处理，即可让需要大量计算的程序得到较大的性能上的提升，这是十分有意义。

不过，当计算规模非常小的时候，不适合使用多线程来计算。除此之外，还需要注意的是程序中设置的线程数，不宜超过物理机的核心数太多，否则线程切换带来的开销将会显著增加，结果就是性能提升将不明显。

七　附录

代码 25: HelloWorld 程序

```
#include <stdio.h>
#include <omp.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);
        }
    }
}
```

代码 26: Windows 下矩阵乘法程序

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

#define NUM_THREADS 16

void MatrixGenerate(double *MatSize, int n);

void SerialMatrixMultiply(double *a, double *b, double *c, int n);

void ParallelMatrixMultiply(double *a, double *b, double *c, int n);


int main() {
    int MatrixSize = 1000;
    double *x, *y, *z;
    clock_t StartTime, EndTime;
    for (int i = 0; i < 3; i++) {
        // 初始化
        x = (double *) malloc(sizeof(double) * MatrixSize * MatrixSize);
        y = (double *) malloc(sizeof(double) * MatrixSize * MatrixSize);
        z = (double *) malloc(sizeof(double) * MatrixSize * MatrixSize);
        MatrixGenerate(x, MatrixSize);
```

```
25          MatrixGenerate(y, MatrixSize);
26          printf("[%d * %d Matrix]\n", MatrixSize, MatrixSize);
27          // 串行计算
28          StartTime = clock();
29          SerialMatrixMultiply(x, y, z, MatrixSize);
30          EndTime = clock();
31          double SerialTime = (double) (EndTime - StartTime) / CLOCKS_PER_SEC;
32          printf("Serial Time: %.3lfs\n", SerialTime);
33          // 并行计算
34          omp_set_num_threads(NUM_THREADS);
35          StartTime = clock();
36          ParallelMatrixMultiply(x, y, z, MatrixSize);
37          EndTime = clock();
38          double ParallelTime = (double) (EndTime - StartTime) / CLOCKS_PER_SEC;
39          printf("Parallel Time: %.3lfs\n", ParallelTime);
40          printf("Speedup: %.3lf\n\n", SerialTime / ParallelTime);
41          // 重置参数
42          free(x);
43          free(y);
44          free(z);
45          MatrixSize += 1000; // 更新矩阵尺寸
46      }
47      return 0;
48  }
49
50  void MatrixGenerate(double *MatSize, int n) {
51      for (int i = 0; i < n; i++) {
52          for (int j = 0; j < n; j++) {
53              MatSize[i * n + j] = (double) rand() / RAND_MAX;
54          }
55      }
56  }
57
58  void SerialMatrixMultiply(double *a, double *b, double *c, int n) {
59      for (int i = 0; i < n; i++) {
60          for (int j = 0; j < n; j++) {
61              double t = 0;
62              for (int k = 0; k < n; k++) {
63                  t += (double) a[i * n + k] * b[k * n + j];
64              }
65              c[i * n + j] = t;
66          }
67      }
68  }
69
70  void ParallelMatrixMultiply(double *a, double *b, double *c, int n) {
71  #pragma omp parallel for
```

```
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            double t = 0;
            for (int k = 0; k < n; k++) {
                t += (double) a[i * n + k] * b[k * n + j];
            }
            c[i * n + j] = t;
        }
    }
}
```

代码 27: Linux 下矩阵乘法程序代码

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

#define NUM_THREADS 16

void MatrixGenerate(double *MatSize, int n);

void SerialMatrixMultiply(double *a, double *b, double *c, int n);

void ParallelMatrixMultiply(double *a, double *b, double *c, int n);


int main() {
    int MatrixSize = 1000;
    double *x, *y, *z;
    struct timespec StartTime;
    struct timespec EndTime;
    for (int i = 0; i < 3; i++) {
        // 初始化
        x = (double *) malloc(sizeof(double) * MatrixSize * MatrixSize);
        y = (double *) malloc(sizeof(double) * MatrixSize * MatrixSize);
        z = (double *) malloc(sizeof(double) * MatrixSize * MatrixSize);
        MatrixGenerate(x, MatrixSize);
        MatrixGenerate(y, MatrixSize);
        printf("[%d * %d Matrix]\n", MatrixSize, MatrixSize);
        // 串行计算
        clock_gettime(CLOCK_REALTIME, &StartTime);
        SerialMatrixMultiply(x, y, z, MatrixSize);
        clock_gettime(CLOCK_REALTIME, &EndTime);
        long SerialTime = 1000000*(EndTime.tv_sec - StartTime.tv_sec) + (EndTime
            .tv_nsec - StartTime.tv_nsec) / 1000;
        printf("Serial Time: %.3lfms\n", SerialTime);
```

```
34          // 并行计算
35          omp_set_num_threads(NUM_THREADS);
36          clock_gettime(CLOCK_REALTIME, &StartTime);
37          ParallelMatrixMultiply(x, y, z, MatrixSize);
38          clock_gettime(CLOCK_REALTIME, &EndTime);
39          long ParallelTime = 1000000*(EndTime.tv_sec - StartTime.tv_sec) + (
                EndTime.tv_nsec - StartTime.tv_nsec) / 1000;
40          printf("Parallel Time: %.3lfms\n", ParallelTime);
41          printf("Speedup: %.3lf\n\n", SerialTime / ParallelTime);
42          // 重置参数
43          free(x);
44          free(y);
45          free(z);
46          MatrixSize += 1000; // 更新矩阵尺寸
47      }
48      return 0;
49  }
50
51  void MatrixGenerate(double *MatSize, int n) {
52      for (int i = 0; i < n; i++) {
53          for (int j = 0; j < n; j++) {
54              MatSize[i * n + j] = (double) rand() / RAND_MAX;
55          }
56      }
57  }
58
59  void SerialMatrixMultiply(double *a, double *b, double *c, int n) {
60      for (int i = 0; i < n; i++) {
61          for (int j = 0; j < n; j++) {
62              double t = 0;
63              for (int k = 0; k < n; k++) {
64                  t += (double) a[i * n + k] * b[k * n + j];
65              }
66              c[i * n + j] = t;
67          }
68      }
69  }
70
71  void ParallelMatrixMultiply(double *a, double *b, double *c, int n) {
72  #pragma omp parallel for
73      for (int i = 0; i < n; i++) {
74          for (int j = 0; j < n; j++) {
75              double t = 0;
76              for (int k = 0; k < n; k++) {
77                  t += (double) a[i * n + k] * b[k * n + j];
78              }
79              c[i * n + j] = t;
```

```
        }
    }
}
```

代码 28: 估算 PI 值程序代码

```
#include <stdio.h>
#include <time.h>
#include <omp.h>

#define THREAD_NUM 4
static long num_steps = 100000;

double SerialPI(int Steps);

double ParallelPI_1(int Steps);

double ParallelPI_2(int Steps);

double ParallelPI_3(int Steps);

double ParallelPI_4(int Steps, int ThreadNum);

int main() {
    int Steps = 1000000000;
    double PI;
    clock_t SerialStartTime, SerialEndTime;

    SerialStartTime = clock();
    PI = SerialPI(Steps);
    SerialEndTime = clock();
    double t0 = (double) (SerialEndTime - SerialStartTime) / CLOCKS_PER_SEC;
    // 串行方法
    printf("======Serial======\n");
    printf("PI Result: %.16lf\nTime: %.16fs\n", PI, t0);
    // 并行方法1
    double ParallelStartTime, ParallelEndTime;
    ParallelStartTime = omp_get_wtime();
    PI = ParallelPI_1(Steps);
    ParallelEndTime = omp_get_wtime();
    double t1 = ParallelEndTime - ParallelStartTime;
    printf("\n======Parallel 1======\n");
    printf("PI Result: %.16lf\nTime: %.16fs\n", PI, t1);
    // 并行方法2
    ParallelStartTime = omp_get_wtime();
    PI = ParallelPI_2(Steps);
    ParallelEndTime = omp_get_wtime();
```

```
42      double t2 = ParallelEndTime - ParallelStartTime;
43      printf("\n======Parallel 2======\n");
44      printf("PI Result: %.16lf\nTime: %.16fs\n", PI, t2);
45      // 并行方法3
46      ParallelStartTime = omp_get_wtime();
47      ParallelPI_3(Steps);
48      ParallelEndTime = omp_get_wtime();
49      double t3 = ParallelEndTime - ParallelStartTime;
50      printf("\n======Parallel 3======\n");
51      printf("PI Result: %.16lf\nTime: %.16fs\n", PI, t3);
52      // 并行方法4（测试THREAD_NUM情况）
53  //    ParallelStartTime = omp_get_wtime();
54  //    ParallelPI_4(Steps,THREAD_NUM);
55  //    ParallelEndTime = omp_get_wtime();
56  //    double t4 = ParallelEndTime - ParallelStartTime;
57  //    printf("\n======Parallel 4======\n");
58  //    printf("PI Result: %.16lf\nTime: %.16fs\n", PI, t3);
59
60      printf("\n======Parallel 4======\n");
61      for (int i = 1; i <= 2048; i *= 2) {
62          ParallelStartTime = omp_get_wtime();
63          PI = ParallelPI_4(Steps, i);
64          ParallelEndTime = omp_get_wtime();
65          double t4 = ParallelEndTime - ParallelStartTime;
66          printf("[Thread Num:%d]\n", i);
67          printf("PI Result: %.16lf\nTime: %.16fs\nSpeedup: %.6f\n", PI, t4, t0 /
                t4);
68      }
69  }
70
71  double SerialPI(int Steps) {
72      double sum = 0.0;
73      double Step = 1.0 / (double) Steps;
74      for (int i = 0; i < Steps; i++) {
75          double x = (i + 0.5) * Step;
76          sum = sum + 4.0 / (1.0 + x * x);
77      }
78      return Step * sum;
79  }
80
81
82  double ParallelPI_1(int Steps) {
83      int i;
84      double sum[THREAD_NUM * 16]; // 目的：减少False Sharing，实现加速比>1
85      double result = 0.0;
86      double Step = 1.0 / (double) Steps;
87      omp_set_num_threads(THREAD_NUM);
```

```
88  #pragma omp parallel private(i)
89      {
90          double x;
91          int id = omp_get_thread_num();
92          for (i = id, sum[id] = 0.0; i < Steps; i += THREAD_NUM) {
93              x = (i + 0.5) * Step;
94              sum[id * 16] += 4.0 / (1.0 + x * x);
95          }
96      }
97      for (i = 0; i < THREAD_NUM; ++i) {
98          result += sum[i * 16];
99      }
100     return result * Step;
101 }
102
103
104 double ParallelPI_2(int Steps) {
105     double sum[THREAD_NUM * 16]; // 目的：减少False Sharing，实现加速比>1
106     double result = 0.0;
107     double Step = 1.0 / (double) Steps;
108     omp_set_num_threads(THREAD_NUM);
109 #pragma omp parallel
110     {
111         int id = omp_get_thread_num();
112         sum[id] = 0.0;
113 #pragma omp for
114         for (int i = 0; i < Steps; i++) {
115             double x = (i + 0.5) * Step;
116             sum[id * 16] += 4.0 / (1.0 + x * x);
117         }
118     }
119     for (int i = 0; i < THREAD_NUM; ++i) {
120         result += sum[i * 16];
121     }
122     return result * Step;
123 }
124
125
126 double ParallelPI_3(int Steps) {
127     int i;
128     double sum;
129     double result = 0.0;
130     double Step = 1.0 / (double) Steps;
131     omp_set_num_threads(THREAD_NUM);
132 #pragma omp parallel private(sum, i)
133     {
134         int id = omp_get_thread_num();
```

```
135         sum = 0.0;
136         for (i = id; i < Steps; i += THREAD_NUM) {
137             double x = (i + 0.5) * Step;
138             sum += 4.0 / (1.0 + x * x);
139         }
140 #pragma omp critical
141         result += sum;
142     };
143     return result * Step;
144 }
145
146
147 double ParallelPI_4(int Steps, int ThreadNum) {
148     double sum;
149     double Step = 1.0 / (double) Steps;
150     omp_set_num_threads(ThreadNum);
151 #pragma omp parallel for reduction(+:sum)
152     for (int i = 0; i < Steps; i++) {
153         double x = (i + 0.5) * Step;
154         sum += 4.0 / (1.0 + x * x);
155     }
156     return Step * sum;
157 }
```