OpenMP 并行编程实验报告

姓名

2024 年 5 月 30 日

1 实验环境

1. macOS Silicon：Apple M2 8 核心 (@3.49GHz 4× 高性能核心 + @2.42GHz 4× 能效核心)。
2. Windows Chip：Intel(R) Core(TM) i7-11800H 16 线程 @ 4.60GHz。
3. Docker: 运行于非 WSL2 的 Windows11 上，镜像地址为`openmpp/openmpp-build:ubuntu`。

2 实验目的

- 并行计算的性能评估与分析；
- 并行计算在复杂运算中的加速效果；
- 并行编程技术的实际应用效果评价。

3 实验步骤

1. 并行计算的性能评估与分析：通过 HelloWorld 程序的串行执行与并行执行（不设置线程数以及设置 8 个线程）的对比分析，旨在揭示并行计算在不同线程配置下的性能变化及其对程序执行时间的影响。此实验目的在于评估并行计算相较于串行计算在处理简单任务时的效率提升，并分析线程数对执行效率的具体影响，以及探索默认线程配置下的系统表现。
2. 并行计算在复杂运算中的加速效果：通过编程实现大规模向量的矩阵乘法并行计算，并在不同的线程数（1、2、4、8、16、32）下评估执行时间，以及在不同操作系统环境（Windows, Linux 及虚拟机下的 Linux 系统）中比较加速比。该实验目的在于深入分析并行计算在处理复杂数学运算时的性能提升，以及线程数与运行时间之间的关系，进一步理解操作系统环境对并行计算效率的影响。
3. 并行编程技术的实际应用效果评价：通过 OpenMP 实例估算 Pi 值的实验，调试并比较串行算法与四种不同的并行程序的加速比，检查并行编程是否有效提高计算效率。此外，探索其他实验内容，如私有变量和共有变量的性能对比，分析并行化的额外负担、线程负载均衡问题及线程同步问题。该实验目的旨在通过实际案例测试并行编程技术的应用效果，特别是在精确计算和资源调度方面的表现，为并行计算在科学研究和工程应用中的实践提供理论依据和技术支持。
4. 根据上面对 OpenMP 的学习使用 OpenMP 进行更多的实验来加深印象和理解。

4 编程及结果分析

4.1 HelloWorld 串行并行

通过本次的实验可以发现，串行结构下运行之后只会打印一遍 `HelloWorld`，如果在打印的语句之前加上 `#pragma omp parallel`，则会打印 8 遍 `HelloWorld`。这是因为 `#pragma omp parallel` 会创建一个并行区域，然后在这个区域中创建多个线程，因为打印语句之前没有加 `#pragma omp single` 或者 `#pragma omp master` 进行修饰，所以编译器会将这条打印语句分给所有的线程都执行一次，其中如果设置线程数为 10，效果如图 1d所示 (因为本身 CPU 的核心数就为 8)，如果不设置线程数，效果如图1e所示，可以看到如果不使用线程数的设置，那么创建的并行区域会默认使用电脑的全部 CPU 核心进行多线程的处理，此时 $num\_thread = \$\{nproc\}$，如果设置线程数，那么编译器会按照设定的线程数进行任务的分配。其中可以使用 `single` 和 `master` 子句来控制某个程序块中的代码只被一个线程执行一次，其中 `single` 和 `master` 的区别在于 `single` 会被所有线程中的某一个执行一次效果如图1a所示，而 `master` 则是只会被线程号为 0 的主线程执行一次，效果如图1b 所示。在后续的实验中，可以发现有些子句是互相嵌套的，比如 `for` 子句和 `section(s)` 子句都自带 `single` 的特点，只会被执行一次。

```
Hello World from OMP thread 2
Number of threads is 10

Process finished with exit code 0
```

(a) 使用 single 的打印结果

```
Hello World from OMP thread 0
Number of threads is 10

Process finished with exit code 0
```

(b) 使用 master 的打印结果

```
Hello World from OMP thread 0
Number of threads is 10

Process finished with exit code 0
```

(c) 串行的打印结果

```
Hello World from OMP thread 1
Hello World from OMP thread 4
Hello World from OMP thread 6
Hello World from OMP thread 3
Hello World from OMP thread 0
Hello World from OMP thread 2
Hello World from OMP thread 8
Hello World from OMP thread 9
Hello World from OMP thread 7
Number of threads is 10
Hello World from OMP thread 5

Process finished with exit code 0
```

(d) 设置线程数为 10 的打印结果

```
Hello World from OMP thread 1
Hello World from OMP thread 3
Hello World from OMP thread 2
Hello World from OMP thread 4
Hello World from OMP thread 5
Hello World from OMP thread 0
Number of threads is 8
Hello World from OMP thread 7
Hello World from OMP thread 6

Process finished with exit code 0
```

(e) 不设置线程数的打印结果

图 1: 不同设置下的 HelloWorld 的打印结果

4.2 关于矩阵乘法的实验

下面是本次实验使用的所有规格设备的运行串行矩阵乘法的用时，也是 Baseline 的数据。`Ubuntu(n)` 代表了运行在 Windows 上的 Docker 镜像，这里的镜像地址都来自于实验环境中的 Docker 镜像地址，其中的 `n` 代表的是核心数，通过 Docker Desktop 进行修改。其中红色标出了 Baseline 中在 1000×1000，2000×2000 和 3000×3000 大小矩阵的最快运行时间的设备。去除程序运行的偶然性，基本的运行时间误差在 5% 左右，但是可以看出镜像运行的速度是远高于宿主机和其他设备的这里考虑到 Docker 的特点可

以总结出以下几点原因：

1. 资源限制：Docker 容器可以配置 CPU 和内存使用限制。如果宿主机上运行很多其他程序，这些程序可能会竞争资源，导致你的程序运行缓慢。而在 Docker 容器中，你的程序可能通过资源限制获得了保证的 CPU 时间和内存，从而运行得更快。

2. 文件系统缓存：Docker 容器使用宿主机的文件系统，但也可以配置为使用特定的存储驱动来优化读写操作。在某些情况下，这可能会导致容器内的文件操作比直接在宿主机上执行时更高效。

3. 内核优化：Docker 容器直接运行在宿主机的内核上，但某些内核调优（如 TCP/IP 堆栈调优）可能只对容器内部的程序有益，这取决于 Docker 的配置和宿主机的系统设置。

Device \ Size	1000	2000	3000
Mac OS	3.62	31.74	112.17
Windows	2.71	31.91	143.90
Ubuntu (4)	2.14	***22.66***	82.43
Ubuntu (8)	***2.12***	22.84	82.11
Ubuntu (16)	2.14	23.03	***81.96***

表 1: 不同设备运行串行矩阵乘法的时间 (注：Ubuntu(n) 中 n 代表的是核心数，这里的 Ubuntu 没有特殊说明都是运行在 Windows 上的镜像)

Thread	Matrix Size	Use Reduction(s)	No Reduction(s)
1	1000	3.12 (0.87)	2.47 (1.1)
	2000	35.57 (0.9)	32.25 (0.99)
	3000	150.29 (0.96)	143.59 (1.0)
2	1000	1.66 (1.63)	1.39 (1.95)
	2000	18.61 (1.71)	16.50 (1.93)
	3000	77.42 (1.86)	73.27 (1.96)
4	1000	0.98 (2.77)	0.92 (2.95)
	2000	9.70 (3.29)	8.85 (3.61)
	3000	41.42 (3.47)	40.05 (3.59)
8	1000	0.62 (4.37)	0.55 (4.93)
	2000	5.94 (5.37)	5.40 (5.91)
	3000	25.26 (5.7)	24.21 (5.94)
16	1000	0.53 (5.11)	0.48 (5.65)
	2000	6.08 (5.25)	5.81 (5.49)
	3000	22.17 (6.49)	22.03 (6.53)
32	1000	0.53 (5.11)	0.47 (5.77)
	2000	5.86 (5.45)	5.67 (5.63)
	3000	22.29 (6.46)	21.72 (6.63)

表 2: Windows 和 macOS 不同大小矩阵和设定的线程数运行时间对比（包含加速比）

首先测试了是否使用 reduction 子句优化最内层的循环进行求和可以提高程序的运行速度。如表??所示，可以看到原本应该加速的程序并没有被加速，反而却整体运行速度都变慢了，总结有如下几点原因：

1. **reduction** 操作的开销： `reduction` 子句由于需要为每个线程维护 reduction 变量的私有副本并在并行区域结束时合并它们，引入了额外的开销。在最内层循环的情况下，这种开销可能会抵消并行化累加操作的好处。

2. 频繁的同步：在最内层循环中利用 `reduction` 会导致在循环结束时频繁的同步点，因为线程需要更新全局累加器。这种频繁的同步可能会阻碍并行执行效率，特别是当循环迭代计算量不大时。

3. 缓存和内存带宽的次优利用：最内层循环中的并行 reduction 可能导致缓存和内存带宽的使用不佳，因为该操作可能阻止有效地利用空间和时间局部性。结果可能是增加了缓存未命中和内存流量，这可能会减慢计算速度。

Thread	Matrix Size	MacOS(s)	Windows(s)
1	1000	3.61 (1.0)	2.47 (1.1)
	2000	30.72 (1.03)	32.25 (0.99)
	3000	107.18 (1.05)	143.59 (1.0)
2	1000	1.85 (1.96)	1.39 (1.95)
	2000	15.92 (1.99)	16.50 (1.93)
	3000	56.11 (2.0)	73.27 (1.96)
4	1000	0.98 (3.69)	0.92 (2.95)
	2000	8.20 (3.87)	8.85 (3.61)
	3000	29.35 (3.82)	40.05 (3.59)
8	1000	***0.65 (5.57)***	0.55 (4.93)
	2000	***5.90 (5.38)***	5.40 (5.91)
	3000	***23.73 (4.73)***	24.21 (5.94)
16	1000	0.64 (5.66)	***0.48 (5.65)***
	2000	5.63 (5.64)	***5.81 (5.49)***
	3000	23.82 (4.71)	***22.03 (6.53)***
32	1000	0.64 (5.66)	0.47 (5.77)
	2000	5.58 (5.69)	5.67 (5.63)
	3000	23.20 (4.83)	21.72 (6.63)

表 3: Windows 和 macOS 不同大小矩阵和设定的线程数运行时间对比（包含加速比）

然后本实验测试了在八核心的 macOS 和 16 线程的 Windows 上运行三种大小的矩阵乘法操作，然后通过 `omp_set_num_threads()` 设置不同的线程数，测量运行的时间，可以看到两个系统的运行时间在红色标出的线程数之后继续再增加线程数，速度提升不大。并且这里的线程数正好等于当前系统的 CPU 核心数 (macOS 为 8，Windows 为 16)，可以总结出以下的原因：

1. 硬件资源限制：每个 CPU 核心在同一时间只能有效执行一个线程。当线程数超过核心数时，额外的线程不得不等待 CPU 资源，导致部分线程在某些时刻处于闲置状态，从而减少了并行效率。

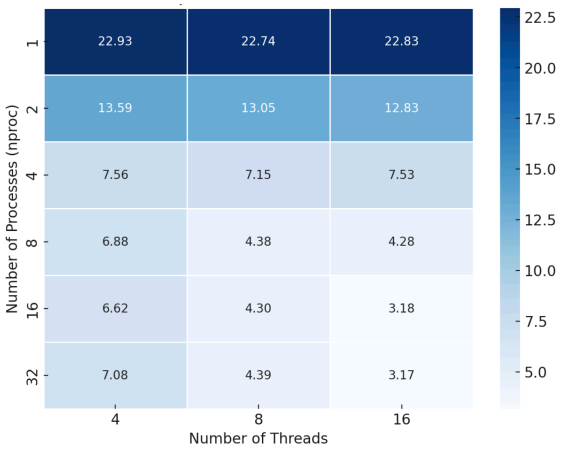
2. 上下文切换开销：当运行的线程数超过处理器核心数时，操作系统需要进行线程间的上下文切换，以

模拟多线程并发执行。这个上下文切换过程需要时间，特别是当线程数量大大超过核心数时，这种开销会显著影响程序的执行效率。

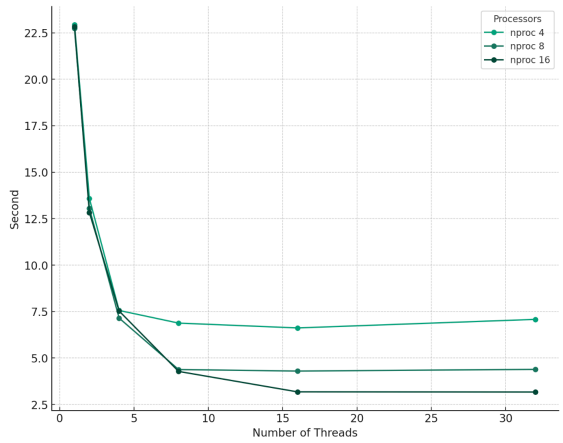
3. 线程管理与同步开销：随着线程数的增加，线程间的协调和通信成本也会增加。特别是在需要频繁进行数据同步和共享的并行程序中，线程越多，等待同步的时间就越长，这会抵消并行计算带来的部分性能增益。

nproc \ thread	Ubuntu(4)	Ubuntu(8)	Ubuntu(16)
1	22.93	22.74	22.83
2	13.59	13.05	12.83
4	***7.11***	7.15	7.53
8	6.88	***4.38***	4.28
16	6.62	4.30	***3.18***
32	7.08	4.39	3.17

表 4: 在不同的 nproc 和线程数下的性能数据 (sec)



(a) nprco 和 thread 热力图



(b) nproc 和 thread 的折线图

这里测试了使用 Docker 镜像的时候在设置不同的 CPU 核心的情况下运行时间的对比，可以看到在表**??**中每一个核心数的 Ubuntu 镜像在经过红色标注的数据之后随着设定的线程数增加，程序运行的效率提升变得不明显而且这时设定的线程数也正好等于镜像的虚拟核心数，可以看下面的热度图和折线图更加直观，热度图中可以看看到后面的数据颜色基本相同，而折线图中可以看到一开始曲线重合，后面出现了分裂。这也证明了上面的三个原因的正确性，虽然编译器在编译程序的时候创建了 N 多个线程，但是 CPU 的核心只有 n 个核心 ($n \leq N$)，CPU 最多只能让 n 个进程上处理机运行获得 CPU 核心的资源，在这样的情况下多余的线程只能停止等待 CPU 资源的释放，所以其实和创建 n 个进程时的效果差不多甚至还会更差一些，因为多线程的资源争夺会加剧操作系统的调度负担。

4.3 计算 π 值的实验

在这个实验中，为了避免程序运行的偶然性，对每个修改之后的串行和 OpenMP 程序都进行了五次运行，然后取时间的平均值作为参考值和计算加速比，结果如表5所示，可以看到不同的 OpenMP 程序都产生了约为 2 的加速比。

表 5: 不同方法的性能对比单位 $10^{-4}(sec)$

Times/Method	Serial	OMP1	OMP2	OMP3	OMP4
1	6.31	2.19	3.74	3.09	4.18
2	6.12	2.21	2.26	2.35	2.13
3	3.75	2.24	2.53	2.20	2.16
4	4.42	2.15	2.76	2.17	2.15
5	3.89	2.07	2.35	2.19	2.25
ave	4.898	2.172	2.728	2.400	2.574
SpeedUp	-	2.255	1.795	2.041	1.903

4.3.1 OpenMP 程序 1

该程序通过使用 OpenMP 并行化技术来加速圆周率的计算，采取了数值积分的方法来近似计算圆周率的值，并利用了以下并行计算思想和 OpenMP 技术：

1. 并行计算思想：圆周率的每一个微分元素的计算都是独立的，可以并行计算，并且这里的累加，加法具有交换律和结合律，先加哪一部分最后的结果都是相同的。

   - 运算无关性：圆周率的每一个微分元素的计算都是独立的，可以并行计算，并且这里的累加，加法具有交换律和结合律，先加哪一部分最后的结果都是相同的。
   - 工作共享与负载平衡：手动通过区分奇偶性给线程分配任务，确保两个程序有着共同的工作量，这样就可以最大化利用两个线程，同时使用数组来共享内存空间，还使用数组的位置坐标进行了两个线程的加法的结果的隔离，防止了线程访问临界资源的冲突。

2. 使用的 **OpenMP** 技术：

   - 设置线程数 (`omp_set_num_threads(NUM_THREADS)`)：指示使用的线程数量，根据处理器的核心数量进行调整，以最大化并行执行的效率。
   - 并行区域 (`#pragma omp parallel`)：标记并行执行的代码块，生成并行代码以便在多个线程上执行。
   - 私有变量 (`private(i,x)`)：确保每个线程有自己的循环计数器副本，避免数据竞争。
   - 计时 (`omp_get_wtime()`)：在并行计算开始和结束时获取时间，计算出计算圆周率所需的时间，评估并行化带来的性能提升。

3. 拓展：通过将迭代次数降低为 99999 进行测试，统计在两个线程上运行的任务数量，然后通过 `for (i = (id==1)?0:1, sum[id] = 0.0, cnt[id] = 0; i < num_steps; i = i + NUM_THREADS)` 修改每个线程的起始值来交换任务的配置数量，可以发现分配的任务数量改变了。

表 6: 任务分配情况统计

Thread \ Exchange	True	False
1	49999	50000
2	50000	49999

4.3.2 OpenMP 程序 2

1. 程序错误：

   - 数组变量无需在并行程序内部进行再次赋值，这类似于函数的传参，当传入普通变量则会进行拷贝构造，但是传入数组变量时传入的其实是指向存放数组位置的指针，这样在并行区域内对数组进行赋值操作其实是直接作用于原数组元素所在的内存地址，这样会直接修改原数组的地址，所以无需在内部再次进行赋值，这会浪费时间。
   - 虽然从逻辑上是没有问题的，但是 x 的声明可以放在并行区域的外部，并且声明为 private 变量。

2. **OpenMP** 思想：

   - 分而治之：程序将圆周率计算任务分解为多个子任务，每个线程处理一部分迭代，从而减少总体执行时间。
   - 工作共享与负载平衡：没有显示的分配给那些任务给哪些线程，而是由编译器进行选择实现负载均衡，在测试中两个线程各自进行 5000 次任务的计算和累加。

4.3.3 OpenMP 程序 3

1. **OpenMP** 思想：

   - 分而治之：程序将圆周率计算任务分解为多个子任务，每个线程处理一部分迭代，从而减少总体执行时间。
   - 工作共享与访问控制：通过编译器自动进行任务的分配，并且设置了一个临界区域，控制同一时间更新 π 的线程数量为 1，避免了修改出现冲突的情况。

2. 使用的 **OpenMP** 技术：

   - 设置线程数：通过 `omp_set_num_threads(NUM_THREADS)` 设置线程数量。
   - 并行区域：使用 `#pragma omp parallel` 指令创建并行区域。
   - 私有与共享变量：指定 `i`、`id`、`x` 和 `sum` 为私有变量，`pi` 作为共享变量在临界区域中更新。
   - 临界区：使用 `#pragma omp critical` 确保更新 `pi` 操作的一致性。
   - 时间测量：使用 `omp_get_wtime()` 计算并行区域的执行时间，评估性能提升。

4.3.4 OpenMP 程序 4

1. 并行计算思想：

   - 分工合作：将圆周率计算任务分解成多个子任务，并通过多个处理器核心并行执行这些任务，减少总体的计算时间。
   - 负载均衡：自动分配循环迭代给多个线程执行，帮助实现工作负载在各线程间的均匀分配，最大化处理器资源的利用。

2. 使用的 **OpenMP** 技术：

   - 设置线程数量：通过 `omp_set_num_threads(NUM_THREADS)` 指令指定并行执行时的线程数，优化并行性能。

- 并行循环：使用 `#pragma omp parallel for` 指令自动将 for 循环并行化，实现循环的快速执行。
- 归约操作：`reduction(+:sum)` 子句对变量 `sum` 执行归约操作，实现各线程计算的局部和的安全累加。
- 私有变量：`private(x)` 子句确保每个线程有自己的 `x` 变量副本，避免并行计算中的数据竞争。
- 性能测量：利用 `omp_get_wtime()` 函数在并行区域开始前后测量时间，评估并行加速的效果。

4.3.5 总结

从上面的实验结果可以看出：

1. 可以通过编程的手段进行任务的分配，实现负载的均衡。也可以通过编译器的自动分配实现负载的均衡。
2. 一些子句的特性可能结合了其他的子句特性，就像 java 中 SpringBoot 的注解一样。比如 `reduction` 子句的特性就结合了 `private` 和 `critical` 的特性。

4.4 其他实验

4.4.1 并行化的负担

通过运行下面的程序，下面的程序使用了一个空的并行化代码块，然后统计这个代码块执行的时间，就可以得出并行化的启动和准备的时间负担。

```
#include <stdio.h>
#include <omp.h>
int main() {
    double start_time = omp_get_wtime();
#pragma omp parallel num_threads(4)
    {

    }
    double time_taken = omp_get_wtime() - start_time;
    printf("Time taken for empty parallel region: %.8f seconds\n",
        time_taken);
    return 0;
}
```

Listing 1: 并行化的负担

在输出中可以发现改并行区域开始到结束耗费了 0.00025200s 的时间，对比一下上面的 π 值计算的时间，可以发现这个时间消耗还是比较大的。

4.4.2 私有变量和公有变量

通过运行下面的程序，其中 `shared_var` 是一个共享变量，对其的更新使用 `atomic` 关键字进行保护，`private_var` 是一个私有变量，每个线程都有自己的副本，将其和 `shared_var` 一起放入循环进行更新。

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

```
int main() {
    int shared_var = 0;
#pragma omp parallel num_threads(4)
    {
        int private_var = 0;
#pragma omp for
        for(int i = 0; i < 10; ++i) {
            private_var++;
#pragma omp atomic
            shared_var++;
        }
        printf("Thread %d, private_var = %d\n", omp_get_thread_num(),
            private_var);
    }

    printf("shared_var = %d\n", shared_var);
    return 0;
}
```

Listing 2: 私有变量和公有变量

输出结果如下所示，可以看到三个线程的 `private_var` 都不相同切相加起来等于 `shared_var`，同时也等于循环次数。

```
Thread 1, private_var = 3
Thread 3, private_var = 2
Thread 0, private_var = 3
Thread 2, private_var = 2
shared_var = 10
```

4.4.3 负载不均衡

通过运行下面的程序，对一些线程使用较轻的负担，对其他的线程使用很重的负担，比如循环很多次，来观察结束时间的快慢。

```
#include <stdio.h>
#include <omp.h>
int main() {
#pragma omp parallel num_threads(4)
    {
        int id = omp_get_thread_num();
        if(id < 2) {
            printf("Thread %d finished quickly.\n", id);
        } else {
            for(long i = 0; i < 100000000; ++i) {}
            printf("Thread %d finished later.\n", id);
        }
```

```
    }
    return 0;
}
```

Listing 3: 负载不均衡

结果如下所示，可以看到，两个负载较轻的线程提前完成了任务，但是两个负载重的线程都是最后完成的任务并且运行多次的结果上没有任何区别。

```
Thread 1 finished quickly.
Thread 0 finished quickly.
Thread 3 finished later.
Thread 2 finished later.
```

Listing 4: 负载不均衡

4.4.4 同步的负担

通过运行下面的程序，通过运行一个没有 `critical` 同步的程序，然后统计时间，然后再运行一个有 `critical` 同步的程序，然后统计时间，就可以得出同步的负担，并且使用 `volatile` 防止编译器的潜在优化。

```
#include <stdio.h>
#include <omp.h>
int main() {
    int i;
    double no_sync_time, sync_time, start, end;
    long num_steps = 1000000;
    volatile int sum = 0;
    start = omp_get_wtime();
#pragma omp parallel for num_threads(4) private(i)
    for(i = 0; i < num_steps; ++i) {
        sum += i;
    }
    end = omp_get_wtime();
    no_sync_time = end - start;
    sum = 0;
    start = omp_get_wtime();
#pragma omp parallel for num_threads(4) private(i)
    for(i = 0; i < num_steps; ++i) {
#pragma omp critical
        sum += i;
    }
    end = omp_get_wtime();
    sync_time = end - start;
    printf("Without synchronization: %.6f seconds\n", no_sync_time);
    printf("With synchronization: %.6f seconds\n", sync_time);
    return 0;
```

```
}
```

Listing 5: 同步的负担

结果如下所示，可以看到有同步的程序运行的时间更长一些。

```
    Without synchronization: 0.003153 seconds
    With synchronization: 0.035381 seconds
```

Listing 6: 同步的负担

5 实验感想

通过这系列的实验，我深入探索了 OpenMP 并行编程的几个核心概念，包括私有与共享变量的处理、并行化的额外负担、线程负载问题以及线程间同步的开销，从而获得了关于并行计算优化和挑战的深刻理解。这些实验不仅加深了我对并行编程理论的认识，而且通过实际代码实践，让我对如何在多线程环境中高效地管理数据和同步有了更直观的感受。

矩阵的乘法运算让我理解了 OpenMP 的线程并行和电脑处理器配置的一些关系，通过设置不同的线程数和核心数发现了设置的线程数多于处理器核心之后可能不会带来很大的性能提升甚至还可能会给操作系统产生大量的调度负担导致性能下降。

π 的计算实验让我更加了解了如何设计并行算法，如何实现负载均衡，如何进行线程同步和优化。

私有和共享变量的实验让我理解了在并行编程中数据封装和访问控制的重要性。通过合理地分配私有变量和共享变量，我们能够有效地减少数据竞争，提高程序的可靠性和性能。

通过测量带有同步操作和不带同步操作的程序运行时间，我深刻理解到了同步机制（如 critical 区域）对程序性能的影响。虽然同步操作对于保证数据的一致性和线程安全至关重要，但它们也可能引入显著的性能开销。因此，在设计并行程序时，寻找最小化同步开销的策略，如减少临界区的使用、利用归约操作等技术，是非常重要的。

线程负载不均衡的实验让我认识到了负载均衡对于并行程序性能的影响。理想情况下，所有线程应该尽可能均匀地分担计算任务，以充分利用所有可用的处理器资源。

最后，通过这些实验，我认识到了并行编程是一把双刃剑。正确和有效地使用 OpenMP 等并行编程工具可以显著加速计算密集型任务，特别是在现代多核处理器架构上。然而，也需要小心翼翼地处理数据共享和同步问题，避免引入过大的性能开销。实验过程中遇到的挑战和解决方案，加深了我对并行计算复杂性的理解，也激发了我进一步探索更高效并行算法的兴趣。总之，这是一次富有教育意义且启发性的实验过程，让我对并行编程有了更全面、更深入的认识。

A 附录

```
#include"omp.h"
#include"stdio.h"
int main(){
    int 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);
    }
}
```

Listing 7: 串行 Helloworld

```
#include"omp.h"
#include"stdio.h"
int main(){
    int nthreads,tid;
    #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);
        }
    }
}
```

Listing 8: 不设线程并行 Helloworld

```
#include"omp.h"
#include"stdio.h"
int main(){
    int nthreads,tid;
    omp_set_num_threads(10);
 #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);
        }
    }
}
```

Listing 9: 设置 10 个线程并行 Helloworld

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void generateMatrix(double* matrix, int size) {
    for (int i = 0; i < size * size; i++) {
        matrix[i] = rand() % 10;
    }
}
void multiplyMatrices(double* a, double* b, double* result, int size)
    {
    for (int row = 0; row < size; row++) {
        for (int col = 0; col < size; col++) {
            double sum = 0.0;
            for (int k = 0; k < size; k++) {
                sum += a[row * size + k] * b[k * size + col];
            }
            result[row * size + col] = sum;
        }
    }
}
int main() {
    srand(time(NULL));
    int sizes[] = {1000, 2000, 3000};
    for (int index = 0; index < 3; index++) {
        int size = sizes[index];
        printf("Generating and multiplying matrices of size %dx%d.\n"
           , size, size);
        double* a = malloc(size * size * sizeof(double));
        double* b = malloc(size * size * sizeof(double));
        double* result = malloc(size * size * sizeof(double));
        if (a == NULL || b == NULL || result == NULL) {
            printf("Memory allocation failed\n");
            exit(1);
        }
        generateMatrix(a, size);
        generateMatrix(b, size);
        clock_t start = clock();
        multiplyMatrices(a, b, result, size);
        clock_t end = clock();
        double time_spent = (double)(end - start) / CLOCKS_PER_SEC;
           // 计算运行时间
        printf("Done with size %dx%d. Time: %.2f seconds.\n", size,
           size, time_spent);
        free(a);
```

```
        free(b);
        free(result);
    }
    return 0;
}
```

Listing 10: 串行矩阵乘法

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
void generateMatrix(double* matrix, int size) {
    for (int i = 0; i < size * size; i++) {
        matrix[i] = rand() % 10;
    }
}
void multiplyMatrices(double* a, double* b, double* result, int size)
    {
    int i,j,k;
#pragma omp parallel for shared(a, b, result) private(i, j, k)
   schedule(dynamic, 10) num_threads(32)
    for (i = 0; i < size; i++) {
        for (j = 0; j < size; j++) {
            double sum = 0.0;
            for (k = 0; k < size; k++) {
                sum += a[i * size + k] * b[k * size + j];
            }
            result[i * size + j] = sum;
        }
    }
}
int main() {
    srand(time(NULL));

    int sizes[] = {1000, 2000, 3000};
    for (int index = 0; index < 3; index++) {
        int size = sizes[index];
        printf("Generating and multiplying matrices of size %dx%d.\n"
            , size, size);

        double* a = malloc(size * size * sizeof(double));
        double* b = malloc(size * size * sizeof(double));
        double* result = malloc(size * size * sizeof(double));

```

```
        if (a == NULL || b == NULL || result == NULL) {
            printf("Memory allocation failed\n");
            exit(1);
        }

        generateMatrix(a, size);
        generateMatrix(b, size);

        double start = omp_get_wtime();
        multiplyMatrices(a, b, result, size);
        double end = omp_get_wtime();

        double time_spent = end - start;
        printf("Done with size %dx%d. Time: %.2f seconds.\n", size,
           size, time_spent);

        free(a);
        free(b);
        free(result);
    }

    return 0;
}
```

Listing 11: 并行矩阵乘法

```
#include <stdio.h>
#include <time.h>
static long num_steps = 100000;
double step;
int main() {
    int i;
    double x, pi, sum = 0.0;
    clock_t start, end;
    double cpu_time_used;
    step = 1.0 / (double)num_steps;
    start = clock();
    for (i = 0; i < num_steps; i++) {
        x = (i + 0.5) * step;
        sum = sum + 4.0 / (1.0 + x * x);
    }
    pi = step * sum;
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

```

```
    printf("Value of Pi = %.16f\n", pi);
    printf("Time taken: %.8f seconds\n", cpu_time_used);

    return 0;
}
```

Listing 12: 串行 π 值计算

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
int main() {
    int i;
    double x, pi, sum[NUM_THREADS];
    step = 1.0 / (double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    int cnt[NUM_THREADS];
    double start_time = omp_get_wtime();
#pragma omp parallel private(x, i)
    {
        int id = omp_get_thread_num();
        for (i = (id==1)?0:1, sum[id] = 0.0, cnt[id] = 0; i <
            num_steps; i = i + NUM_THREADS) {
            cnt[id]++;
            x = (i + 0.5) * step;
            sum[id] += 4.0 / (1.0 + x * x);
        }
    }
    for (i = 0, pi = 0.0; i < NUM_THREADS; i++)
        pi += sum[i] * step;
    double end_time = omp_get_wtime();
    printf("Pi = %.16f\n", pi);
    printf("Time taken: %.8f seconds\n", end_time - start_time);
    printf("Thread1: %d\n",cnt[0]);
    printf("Thread2: %d",cnt[1]);
    return 0;
}
```

Listing 13: 并行 π 值计算 1

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000;
double step;
```

```
#define NUM_THREADS 2
int main() {
    int i;
    double x, pi = 0.0, sum[NUM_THREADS];
    step = 1.0 / (double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    for (i = 0; i < NUM_THREADS; i++) {
        sum[i] = 0.0;
    }

    double start_time = omp_get_wtime();
#pragma omp parallel private(x, i)
    {
        int id = omp_get_thread_num();
#pragma omp for
        for (i = 0; i < num_steps; i++) {
            x = (i + 0.5) * step;
            sum[id] += 4.0 / (1.0 + x * x);
        }
    }
    for (i = 0, pi = 0.0; i < NUM_THREADS; i++) {
        pi += sum[i] * step;
    }
    double end_time = omp_get_wtime();
    printf("Value of Pi = %.16f\n", pi);
    printf("Time taken: %.8f seconds\n", end_time - start_time);

    return 0;
}
```

Listing 14: 并行 π 值计算 2

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
int main() {
    int i, id;
    double x, sum, pi = 0.0;
    step = 1.0 / (double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    double start_time = omp_get_wtime();
#pragma omp parallel private(i, id, x, sum)
    {
```

```
        id = omp_get_thread_num();
        sum = 0.0;
        for (i = id; i < num_steps; i += NUM_THREADS) {
            x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }
#pragma omp critical
        pi += sum * step;
    }
    double end_time = omp_get_wtime();
    printf("Pi = %.16f\n", pi);
    printf("Time taken: %.8f seconds\n", end_time - start_time);
    return 0;
}
```

Listing 15: 并行 π 值计算 3

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
int main() {
    int i;
    double x, pi, sum = 0.0;
    step = 1.0 / (double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    double start_time = omp_get_wtime();
#pragma omp parallel for reduction(+:sum) private(x)
    for (i = 0; i < num_steps; i++) {
        x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }
    pi = step * sum;
    double end_time = omp_get_wtime();
    printf("Pi = %.16f\n", pi);
    printf("Time taken: %.8f seconds\n", end_time - start_time);

    return 0;
}
```

Listing 16: 并行 π 值计算 4