并行程序设计与算法实验 6

实验	**Pthreads并行应用**	专业	计算机科学与技术
学号	21311525	姓名	马梓培
Email	mazp@mail2.sysu.edu.cn	完成日期	2024/05/01

1. 实验要求

> - 使用自定义的parallel_for替代heated_plate_openmp中的并行构造
> - 该应用模拟规则网格上的热传导，每次循环中对邻域内热量平均
> - $w_{i,j}^{t+1} = \frac{1}{4}(w_{i-1,j-1}^{t} + w_{i-1,j+1}^{t} + w_{i+1,j-1}^{t} + w_{i+1,j+1}^{t})$
> - 测试不同线程、调度方式下的程序并行性能
> - 与原始heated_plate_openmp.c实现进行对比

2. 基于 Pthreads 的 parallel_for 函数替换omp parallel for

代码见附件或github.

2.1 parallel_for.h 头文件

我在头文件中引入一些库，例如C语言标准库，以及`<omp.h>`和`<pthread.h>`。

同时，定义了两个结构体，分别为`Arg`与`parg`，其中：

- `Arg` 结构体：包含用于某些任务的各种参数，例如线程的起始和结束位置、线程数、排名以及指向两个二维数组 `W` 和 `U` 的指针，还有一个 `double` 类型的 `target`。
- `parg` 结构体：封装了线程应该执行的范围（`start` 到 `end`），以及函数指针 `functor`，该函数指针指向每次迭代要执行的函数。`increment` 表示循环的步长。

最后，我定义了parallel_for函数与toDo()：

- `void *toDo(void *arg)`：这是一个线程函数，接受一个 `parg` 类型的指针。该函数遍历从 `start` 到 `end`，步长为 `increment` 的范围，并在每步调用 `functor` 指向的函数。

- `void *paraller_for(int start, int end, int increment, void *(*functor)(void *), void *arg, int num_threads);`：这个函数原型是用来创建并管理多线程的，功能类似于 OpenMP 的 `#pragma omp parallel for`。此函数的实现应负责分配 `start` 到 `end` 范围内的迭代到不同线程，并确保这些线程可以并行执行。

```h
#ifndef PARALLEL_FOR_H
#define PARALLEL_FOR_H

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <omp.h>
#include <pthread.h>
typedef struct {
    int start;
    int end;
    int pos;
    int threadNums;
    int rank;
    double **W;
    double **U;
    double target;
} Arg;
typedef struct {
    int start;
    int end;
    int increment;
    void *(*functor)(void *);
    void *arg;
} parg;
void *toDo(void *arg) {
    parg myArg = *(parg *)arg;
    for (int i = myArg.start; i < myArg.end; i += myArg.increment) {
        (*myArg.functor)(myArg.arg);
    }
}
void *paraller_for(int start, int end, int increment, void *(*functor)(void *), void *arg, int num_threads);
#endif // PARALLEL_FOR_H
```

2.2 parallel_for 函数

parallel_for函数则在parallel_for.c中实现，它也正是我们实验中要实现的.so文件：

`paraller_for`函数主要功能是将一个迭代范围（由 `start` 到 `end`），按照指定的步长 (`increment`) 和线程数量 (`num_threads`) 分配到多个线程上执行。我利用头文件中 `Arg` 类型的数组来存储每个线程的参数。这些参数是从主调用函数传入的 `arg` 复制而来，并为每个线程设置了一个唯一的 `rank`。同时，我准备 `parg` 结构体数组，用于传递给每个线程必要的信息，包括每个线程的起始和结束迭代位置、步长、要执行的函数（`functor`）以及传递给

functor 的参数。

然后就是普普通通的pthread_create与pthread_join了，这里比较麻烦的就是它前期的准备了。

```
void *paraller_for(int start, int end, int increment, void *(*functor)(void *), void *arg, int num_threads) {
    if ((end - start) / (increment * num_threads) < 1)
        num_threads = (end - start) / increment;
    pthread_t thread_handles[num_threads];
    Arg args[num_threads];
    for (int i = 0; i < num_threads; i++) {
        args[i] = *(Arg *)arg;
        args[i].rank = i;
    }
    parg tmp[num_threads];
    int local_size = (end - start) / num_threads;
    for (int i = 0; i < num_threads - 1; i++) {
        tmp[i].start = i * local_size;
        tmp[i].end = tmp[i].start + local_size;
        tmp[i].increment = increment;
        tmp[i].arg = (void *)&args[i];
        tmp[i].functor = functor;
        pthread_create(&thread_handles[i], NULL, toDo, (void *)&tmp[i]);
    }
    tmp[num_threads - 1].start = (num_threads - 1) * local_size;
    tmp[num_threads - 1].end = end;
    tmp[num_threads - 1].increment = increment;
    tmp[num_threads - 1].arg = (void *)&args[num_threads - 1];
    tmp[num_threads - 1].functor = functor;
    pthread_create(&thread_handles[num_threads - 1], NULL, toDo, (void *)&tmp[num_threads - 1]);
    for (int i = 0; i < num_threads; i++)
        pthread_join(thread_handles[i], NULL);
}
```

2.3 实现heated_plate

heated_plate_openmp.c中涉及了很多omp for，例如：

```
#pragma omp for
    for ( i = 1; i < M - 1; i++ )
    {
      w[i][0] = 100.0;
    }
...
#pragma omp for reduction ( + : mean )
    for ( i = 1; i < M - 1; i++ )
    {
      mean = mean + w[i][0] + w[i][N-1];
    }
...
```

```
# pragma omp critical
      {
        if ( diff < my_diff )
        {
          diff = my_diff;
        }
      }
```

2.3.1 初始化

对于w矩阵的i行0列，(i = 1; i < M - 1; i ++)，我是这样实现的：

```
// openmp:
#pragma omp for
    for ( i = 1; i < M - 1; i++ )
    {
      w[i][0] = 100.0;
    }
```

我们需要changeRow and addRow:

```
void *changeRow(void *a)
{
    Arg arg = *(Arg *)a;
    int size = (arg.end - arg.start) / arg.threadNums;
    int rank = arg.rank;
    int start = arg.start + size * rank;
    int end;
    if (rank != arg.threadNums - 1)
        end = start + size;
    else
        end = arg.end;
    for (int i = start; i < end; i++)
        arg.W[arg.pos][i] = arg.target;
}
```

对应到主函数：

```
paraller_for(0, threadNums, 1, changeRow, (void *)&arg, threadNums);
```

其余行与列类似，按照heated_plate_openmp.c实现即可。

2.3.2 更新矩阵内部的值

我们根据原始openmp的代码，实现出对应的pthreads的代码。

我略微解释一下heated_plate_openmp.c代码，并附上对应的pthreads实现：

```
#pragma omp for reduction ( + : mean )
    for ( i = 1; i < M - 1; i++ )
    {
      mean = mean + w[i][0] + w[i][N-1];
    }
#pragma omp for reduction ( + : mean )
    for ( j = 0; j < N; j++ )
    {
      mean = mean + w[M-1][j] + w[0][j];
    }
  }
```

这一部分将初始化赋值的那些元素加入到mean中，方便稍后求均值。

pthreads:

```
void *addRow(void *a)
{
    Arg arg = *(Arg *)a;
    int size = (arg.end - arg.start) / arg.threadNums;
    int rank = arg.rank;
    int start = arg.start + size * rank;
    int end;
    if (rank != arg.threadNums - 1)
        end = start + size;
    else
        end = arg.end;
    double tmp;
    for (int i = start; i < end; i++)
        tmp += arg.W[arg.pos][i];
    pthread_mutex_lock(&myLock);
    mean += tmp;
    pthread_mutex_unlock(&myLock);
}
void *addCol(void *a)
{
    Arg arg = *(Arg *)a;
    int size = (arg.end - arg.start) / arg.threadNums;
    int rank = arg.rank;
    int start = arg.start + size * rank;
    int end;
    if (rank != arg.threadNums - 1)
        end = start + size;
    else
        end = arg.end;
    double tmp;
    for (int i = start; i < end; i++)
        tmp += arg.W[i][arg.pos];
    pthread_mutex_lock(&myLock);
    mean += tmp;
    pthread_mutex_unlock(&myLock);
}
```

对应main.c

```
paraller_for(0, threadNums, 1, addRow, (void *)&arg, threadNums);
paraller_for(0, threadNums, 1, addCol, (void *)&arg, threadNums);
```

注意，由于pthreads是共享内存的实现方式，因此我们在将tmp加入到mean时，需要将mean上互斥锁。

求均值：

```
mean = mean / (double)(2 * M + 2 * N - 4);
```

这一部分openmp与pthreads实现相同，至此，均值求完，我们更新矩阵内部的值。

openmp实现：

```
#pragma omp parallel shared ( mean, w ) private ( i, j )  num_threads(8)
  {
#pragma omp for
    for ( i = 1; i < M - 1; i++ )
    {
      for ( j = 1; j < N - 1; j++ )
      {
        w[i][j] = mean;
      }
    }
  }
```

pthread实现：

```
void *changeMat(void *a)
{
    Arg arg = *(Arg *)a;
    int size = (arg.end - arg.start) / arg.threadNums;
    int rank = arg.rank;
    int start = arg.start + size * rank;
    int end;
    if (rank != arg.threadNums - 1)
        end = start + size;
    else
        end = arg.end;
    for (int i = start; i < end; i++)
        for (int j = 1; j < arg.pos; j++)
            arg.W[i][j] = arg.target;
}
```

2.3.3 迭代

根据物理热力学的知识，只要当前的系统没有其他额外的输入或影响，最终系统会逐渐趋于稳定，也就是说矩阵中同一位置处的值会基本不变。

对于某一位置的更新，则对应于该简化的公式：

$$w_{i,j}^{t+1} = \frac{1}{4}(w_{i-1,j-1}^{t} + w_{i-1,j+1}^{t} + w_{i+1,j-1}^{t} + w_{i+1,j+1}^{t}) \tag{1}$$

openmp实现：

```
  while ( epsilon <= diff )
  {
# pragma omp parallel shared ( u, w ) private ( i, j )  num_threads(8)
    {
/*
  Save the old solution in U.
*/
# pragma omp for
      for ( i = 0; i < M; i++ )
      {
        for ( j = 0; j < N; j++ )
        {
          u[i][j] = w[i][j];
        }
      }
/*
  Determine the new estimate of the solution at the interior points.
  The new solution W is the average of north, south, east and west neighbors.
*/
# pragma omp for
      for ( i = 1; i < M - 1; i++ )
      {
        for ( j = 1; j < N - 1; j++ )
        {
          w[i][j] = ( u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] ) / 4.0;
        }
      }
    }
/*
  C and C++ cannot compute a maximum as a reduction operation.

  Therefore, we define a private variable MY_DIFF for each thread.
  Once they have all computed their values, we use a CRITICAL section
  to update DIFF.
*/
    diff = 0.0;
# pragma omp parallel shared ( diff, u, w ) private ( i, j, my_diff ) num_threads(8)
    {
      my_diff = 0.0;
# pragma omp for
      for ( i = 1; i < M - 1; i++ )
      {
        for ( j = 1; j < N - 1; j++ )
        {
          if ( my_diff < fabs ( w[i][j] - u[i][j] ) )
          {
            my_diff = fabs ( w[i][j] - u[i][j] );
          }
        }
      }
# pragma omp critical
```

```
      {
        if ( diff < my_diff )
        {
          diff = my_diff;
        }
      }
    }

    iterations++;
    if ( iterations == iterations_print )
    {
      printf ( "  %8d  %f\n", iterations, diff );
      iterations_print = 2 * iterations_print;
    }
  }
```

pthreads更新矩阵元素：

我一步一步细讲：

在每一次更新时，是不能直接在原矩阵上更新的，因此我们在更新前需要先复制一个矩阵，在矩阵上求一个某一个元素的邻域均值，再赋值给w。

Copy matrix:

```
void *copyMat(void *a)
{
    Arg arg = *(Arg *)a;
    int size = (arg.end - arg.start) / arg.threadNums;
    int rank = arg.rank;
    int start = arg.start + size * rank;
    int end;
    if (rank != arg.threadNums - 1)
        end = start + size;
    else
        end = arg.end;
    for (int i = start; i < end; i++)
        for (int j = 0; j < arg.pos; j++)
            arg.U[i][j] = arg.W[i][j];
}
```

更新矩阵：

```
void *compute(void *a)
{
    Arg arg = *(Arg *)a;
    int size = (arg.end - arg.start) / arg.threadNums;
    int rank = arg.rank;
    int start = arg.start + size * rank;
    int end;
    double **w = arg.W;
    double **u = arg.U;
    if (rank != arg.threadNums - 1)
        end = start + size;
```

```
    else
        end = arg.end;
    for (int i = start; i < end; i++)
        for (int j = 1; j < arg.pos; j++)
            w[i][j] = (u[i - 1][j] + u[i + 1][j] + u[i][j - 1] + u[i][j + 1]) / 4.0;
}
```

求矩阵差异：

```
void *findDiff(void *a)
{
    int key1, key2;
    Arg arg = *(Arg *)a;
    int size = (arg.end - arg.start) / arg.threadNums;
    int rank = arg.rank;
    int start = arg.start + size * rank;
    int end;
    double **w = arg.W;
    double **u = arg.U;
    if (rank != arg.threadNums - 1)
        end = start + size;
    else
        end = arg.end;
    double myDiff = 0.0;
    for (int i = start; i < end; i++)
    {
        for (int j = 1; j < arg.pos; j++)
        {
            if (myDiff < fabs(w[i][j] - u[i][j]))
            {
                myDiff = fabs(w[i][j] - u[i][j]);
                key1 = i;
                key2 = j;
            }
        }
    }
    pthread_mutex_lock(&myLock);
    if (diff < myDiff)
        diff = myDiff;
    pthread_mutex_unlock(&myLock);
}
```

与求和类似，在更新最大diff时，需要用互斥锁互斥更新。

3. 实验结果

首先，我们先跑一下heated_plate_openmp.c的实验，用来作为一个baseline。

N = M = 250:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./heated_plate_openmp.out
```

```
HEATED_PLATE_OPENMP
  C/OpenMP version
  A program to solve for the steady state temperature distribution
  over a rectangular plate.

  Spatial grid of 250 by 250 points.
  The iteration will be repeated until the change is <= 1.000000e-03
  Number of processors available = 8
  Number of threads =              8

  MEAN = 74.899598

 Iteration  Change

         1  18.724900
         2  9.362450
         4  4.096072
         8  2.288040
        16  1.135841
        32  0.567820
        64  0.282615
       128  0.141682
       256  0.070760
       512  0.035403
      1024  0.017695
      2048  0.008835
      4096  0.004164
      8192  0.001475

      9922  0.001000

  Error tolerance achieved.
  Wallclock time = 1.479547

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

N = M = 500:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./heated_plate_openmp.out

HEATED_PLATE_OPENMP
  C/OpenMP version
  A program to solve for the steady state temperature distribution
  over a rectangular plate.

  Spatial grid of 500 by 500 points.
  The iteration will be repeated until the change is <= 1.000000e-03
  Number of processors available = 8
  Number of threads =              8

  MEAN = 74.949900

```

```
 Iteration  Change

        1  18.737475
        2  9.368737
        4  4.098823
        8  2.289577
       16  1.136604
       32  0.568201
       64  0.282805
      128  0.141777
      256  0.070808
      512  0.035427
     1024  0.017707
     2048  0.008856
     4096  0.004428
     8192  0.002210
    16384  0.001043

    16955  0.001000

  Error tolerance achieved.
  Wallclock time = 7.364223

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

可以看到，openmp的速度真的很快啊，250*250的矩阵1.47s就跑完，而500*500也就只要7.36s。

接下来，是pthread实现的parallel_for的评测：

N = M = 250, thread_nums = 1:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./main.out 250 250 1

HEATED_PLATE_OPENMP
  C/Pthread version
  A program to solve for the steady state temperature distribution
  over a rectangular plate.

  Spatial grid of 250 by 250 points.
  The iteration will be repeated until the change is <= 1.000000e-03
  Number of processors available = 1

  MEAN = 74.899598

 Iteration  Change

        1  18.724900
        2  9.362450
        4  4.096072
        8  2.288040
       16  1.135841
       32  0.567820
       64  0.282615
      128  0.141682
```

```
       256  0.070760
       512  0.035403
      1024  0.017695
      2048  0.008835
      4096  0.004164
      8192  0.001475

      9922  0.001000

  Error tolerance achieved.
  Wallclock time = 16.726149

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

N = M = 250, thread_nums = 2:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./main.out 250 250 2

HEATED_PLATE_OPENMP
  C/Pthread version
  A program to solve for the steady state temperature distribution
  over a rectangular plate.

  Spatial grid of 250 by 250 points.
  The iteration will be repeated until the change is <= 1.000000e-03
  Number of processors available = 2

  MEAN = 74.899598

 Iteration  Change

        1  18.724900
        2  9.362450
        4  4.096072
        8  2.288040
       16  1.135841
       32  0.567820
       64  0.282615
      128  0.141682
      256  0.070760
      512  0.035403
     1024  0.017695
     2048  0.008835
     4096  0.004164
     8192  0.001475

     9922  0.001000

  Error tolerance achieved.
  Wallclock time = 14.408228

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

N = M = 250, thread_nums = 4:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./main.out 250 250 4

HEATED_PLATE_OPENMP
  C/Pthread version
  A program to solve for the steady state temperature distribution
  over a rectangular plate.

  Spatial grid of 250 by 250 points.
  The iteration will be repeated until the change is <= 1.000000e-03
  Number of processors available = 4

  MEAN = 74.899598

 Iteration  Change

          1  18.724900
          2  9.362450
          4  4.096072
          8  2.288040
         16  1.135841
         32  0.567820
         64  0.282615
        128  0.141682
        256  0.070760
        512  0.035403
       1024  0.017695
       2048  0.008835
       4096  0.004164
       8192  0.001475

       9922  0.001000

  Error tolerance achieved.
  Wallclock time = 14.353445

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

N = M = 250, thread_nums = 8:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./main.out 250 250 8

HEATED_PLATE_OPENMP
  C/Pthread version
  A program to solve for the steady state temperature distribution
  over a rectangular plate.

  Spatial grid of 250 by 250 points.
  The iteration will be repeated until the change is <= 1.000000e-03
  Number of processors available = 8

  MEAN = 74.899598
```

```
 Iteration  Change

        1  18.724900
        2  9.362450
        4  4.096072
        8  2.288040
       16  1.135841
       32  0.567820
       64  0.282615
      128  0.141682
      256  0.070760
      512  0.035403
     1024  0.017695
     2048  0.008835
     4096  0.004164
     8192  0.001475

     9922  0.001000

  Error tolerance achieved.
  Wallclock time = 17.758982

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

首先，我们实验的正确性是可以保证的，求得的mean相同，并且最重要的，我们都是在9922这个iter结束迭代。

但是，可以看到，我的指标是明显不如openmp的。其中一个重要原因，就是王老师上课所提到的伪共享。在每一轮的更新中，由于不同线程可能会对同一个cache line进行反复的读写，即使这些变量彼此之间没有直接的数据依赖关系，每次修改也会导致其他核心上缓存行的无效化。这就意味着，即使各线程工作在不同的变量上，它们还是会互相影响对方的性能，因为每次修改都需要从主内存中重新加载整个缓存行。此外，我每调用一次自己实现的parallel_for函数，都会涉及到线程的创建与合并。这里其实可能也是导致运行减慢的一个重要因素。

N = M = 500, thread_nums = 1:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./main.out 500 500 1

HEATED_PLATE_OPENMP
  C/Pthread version
  A program to solve for the steady state temperature distribution
  over a rectangular plate.

  Spatial grid of 500 by 500 points.
  The iteration will be repeated until the change is <= 1.000000e-03
  Number of processors available = 1

  MEAN = 74.949900

 Iteration  Change

        1  18.737475
        2  9.368737
        4  4.098823
```

```
         8  2.289577
        16  1.136604
        32  0.568201
        64  0.282805
       128  0.141777
       256  0.070808
       512  0.035427
      1024  0.017707
      2048  0.008856
      4096  0.004428
      8192  0.002210
     16384  0.001043

     16955  0.001000

  Error tolerance achieved.
  Wallclock time = 46.787368

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

N = M = 500, thread_nums = 2:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./main.out 500 500 2

HEATED_PLATE_OPENMP
  C/Pthread version
  A program to solve for the steady state temperature distribution
  over a rectangular plate.

  Spatial grid of 500 by 500 points.
  The iteration will be repeated until the change is <= 1.000000e-03
  Number of processors available = 2

  MEAN = 74.949900

 Iteration  Change

         1  18.737475
         2  9.368737
         4  4.098823
         8  2.289577
        16  1.136604
        32  0.568201
        64  0.282805
       128  0.141777
       256  0.070808
       512  0.035427
      1024  0.017707
      2048  0.008856
      4096  0.004428
      8192  0.002210
     16384  0.001043

```

```
    16955  0.001000

  Error tolerance achieved.
  Wallclock time = 30.683121

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

N = M = 500, thread_nums = 4:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./main.out 500 500 4

HEATED_PLATE_OPENMP
  C/Pthread version
  A program to solve for the steady state temperature distribution
  over a rectangular plate.

  Spatial grid of 500 by 500 points.
  The iteration will be repeated until the change is <= 1.000000e-03
  Number of processors available = 4

  MEAN = 74.949900

 Iteration  Change

        1  18.737475
        2  9.368737
        4  4.098823
        8  2.289577
       16  1.136604
       32  0.568201
       64  0.282805
      128  0.141777
      256  0.070808
      512  0.035427
     1024  0.017707
     2048  0.008856
     4096  0.004428
     8192  0.002210
    16384  0.001043

    16955  0.001000

  Error tolerance achieved.
  Wallclock time = 23.673142

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

N = M = 500, thread_nums = 8:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./main.out 500 500 8

HEATED_PLATE_OPENMP
```

```
 C/Pthread version
 A program to solve for the steady state temperature distribution
 over a rectangular plate.

 Spatial grid of 500 by 500 points.
 The iteration will be repeated until the change is <= 1.000000e-03
 Number of processors available = 8

 MEAN = 74.949900

Iteration  Change

        1  18.737475
        2  9.368737
        4  4.098823
        8  2.289577
       16  1.136604
       32  0.568201
       64  0.282805
      128  0.141777
      256  0.070808
      512  0.035427
     1024  0.017707
     2048  0.008856
     4096  0.004428
     8192  0.002210
    16384  0.001043

    16955  0.001000

 Error tolerance achieved.
 Wallclock time = 29.062525

HEATED_PLATE_OPENMP:
```

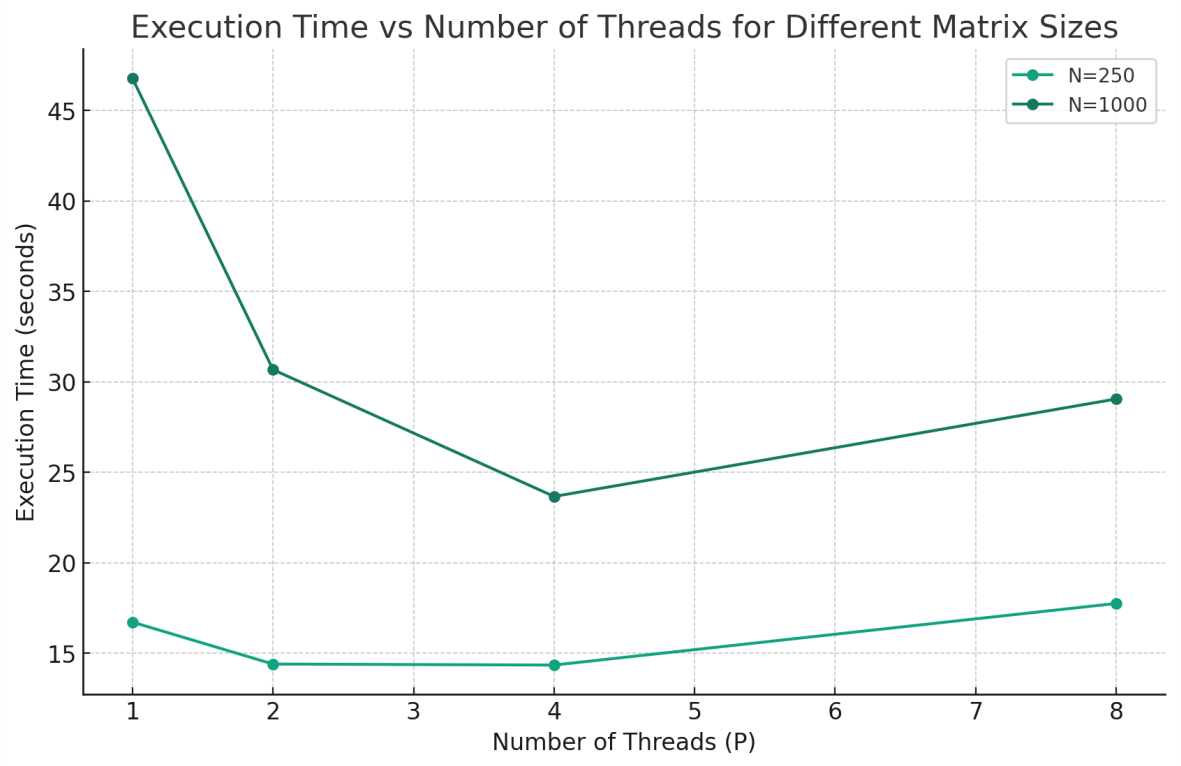
同样，我们结果的正确性是可以保证的。

并且，由于矩阵规模的增大，不同进程访问同一块cache line的概率减小，伪共享的影响有所减小。

表格：

P\N	250	1000
1	16.726149	46.787368
2	14.408228	30.683121
4	14.353445	23.673142
8	17.758982	29.062525

可视化：



4. 实验感想

在这次实验中，我通过使用Pthreads库实现了并行计算，特别是将OpenMP的并行循环结构替换为基于Pthreads的自定义parallel_for函数。通过这种方式，我们能够手动控制线程的创建、执行和同步，而不是依赖OpenMP的自动管理。这为深入理解并行计算的底层细节提供了极好的机会。

通过这次实验，我深刻体会到并行计算中的一些关键挑战，例如线程同步和数据共享的问题。尤其是在处理伪共享和线程管理开销时，这些因素对性能的影响非常显著。在实验中，通过不断调整线程数和计算分配策略，我观察到了性能的变化，并试图找到最优的并行策略。

此外，通过与OpenMP版本的对比，我更加明白了高级并行框架在简化并行编程和优化性能方面的优势。虽然手动实现提供了更多的控制和优化的空间，但也需要更深入的理解和更多的开发工作。