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
- $\qquad \qquad \boldsymbol{w}_{i,j}^{t+1} = \tfrac{1}{4}(w_{i-1,j-1}^t + w_{i-1,j+1}^t + w_{i+1,j-1}^t + w_{i+1,j+1}^t)$
- 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。

```
1
   #ifndef PARALLEL_FOR_H
 2
    #define PARALLEL_FOR_H
 4
   #include <stdlib.h>
 5
   #include <stdio.h>
    #include <math.h>
 6
 7
    #include <omp.h>
    #include <pthread.h>
 8
    typedef struct {
10
        int start;
        int end;
11
12
        int pos;
13
       int threadNums;
14
        int rank;
15
        double **W;
16
        double **U;
17
        double target;
18
    } Arg;
    typedef struct {
19
        int start;
20
21
        int end;
22
        int increment;
        void *(*functor)(void *);
23
24
        void *arg;
25
    } parg;
26
    void *toDo(void *arg) {
27
        parg myArg = *(parg *)arg;
        for (int i = myArg.start; i < myArg.end; i += myArg.increment) {</pre>
28
29
            (*myArg.functor)(myArg.arg);
30
        }
31
    void *paraller_for(int start, int end, int increment, void *(*functor)(void *), void
    *arg, int num_threads);
    #endif // PARALLEL_FOR_H
33
```

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
 1
    *arg, int num_threads) {
 2
        if ((end - start) / (increment * num_threads) < 1)</pre>
            num_threads = (end - start) / increment;
 3
        pthread_t thread_handles[num_threads];
 4
 5
        Arg args[num_threads];
        for (int i = 0; i < num\_threads; i++) {
 6
 7
            args[i] = *(Arg *)arg;
 8
            args[i].rank = i;
 9
        }
        parg tmp[num_threads];
10
11
        int local_size = (end - start) / num_threads;
12
        for (int i = 0; i < num threads - 1; i++) {
            tmp[i].start = i * local size;
13
            tmp[i].end = tmp[i].start + local_size;
14
15
            tmp[i].increment = increment;
            tmp[i].arg = (void *)&args[i];
16
17
            tmp[i].functor = functor;
            pthread_create(&thread_handles[i], NULL, toDo, (void *)&tmp[i]);
18
19
20
        tmp[num_threads - 1].start = (num_threads - 1) * local_size;
        tmp[num_threads - 1].end = end;
21
22
        tmp[num_threads - 1].increment = increment;
23
        tmp[num\_threads - 1].arg = (void *)&args[num\_threads - 1];
24
        tmp[num_threads - 1].functor = functor;
25
        pthread_create(&thread_handles[num_threads - 1], NULL, toDo, (void
    *) & tmp[num_threads - 1]);
26
        for (int i = 0; i < num_threads; i++)</pre>
27
            pthread_join(thread_handles[i], NULL);
28
    }
```

2.3 实现heated_plate

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

```
1
    #pragma omp for
 2
        for (i = 1; i < M - 1; i++)
 3
 4
          w[i][0] = 100.0;
 5
 6
 7
    #pragma omp for reduction ( + : mean )
        for (i = 1; i < M - 1; i++)
 8
 0
        {
10
          mean = mean + w[i][0] + w[i][N-1];
11
12
    . . .
```

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:

```
1  // openmp:
2  #pragma omp for
3  for ( i = 1; i < M - 1; i++ )
4  {
5  w[i][0] = 100.0;
6 }</pre>
```

我们需要changeRow and addRow:

```
1
    void *changeRow(void *a)
 2
 3
        Arg arg = *(Arg *)a;
        int size = (arg.end - arg.start) / arg.threadNums;
 4
 5
        int rank = arg.rank;
        int start = arg.start + size * rank;
 6
 7
8
        if (rank != arg.threadNums - 1)
9
             end = start + size;
10
        else
11
            end = arg.end;
        for (int i = start; i < end; i++)</pre>
12
            arg.W[arg.pos][i] = arg.target;
13
14
   }
```

对应到主函数:

```
1 | 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:

```
1
    #pragma omp for reduction ( + : mean )
2
        for (i = 1; i < M - 1; i++)
3
          mean = mean + w[i][0] + w[i][N-1];
4
5
        }
6
    #pragma omp for reduction ( + : mean )
7
        for (j = 0; j < N; j++)
8
9
          mean = mean + w[M-1][j] + w[0][j];
        }
10
      }
11
```

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)
 1
 2
 3
        Arg arg = *(Arg *)a;
 4
        int size = (arg.end - arg.start) / arg.threadNums;
 5
        int rank = arg.rank;
        int start = arg.start + size * rank;
 6
 7
        int end;
 8
        if (rank != arg.threadNums - 1)
9
            end = start + size;
10
        else
            end = arg.end;
11
12
        double tmp;
13
        for (int i = start; i < end; i++)
14
            tmp += arg.W[arg.pos][i];
15
        pthread_mutex_lock(&myLock);
16
        mean += tmp;
17
        pthread_mutex_unlock(&myLock);
18
19
    void *addCol(void *a)
20
21
        Arg arg = *(Arg *)a;
22
        int size = (arg.end - arg.start) / arg.threadNums;
23
        int rank = arg.rank;
24
        int start = arg.start + size * rank;
25
        int end;
        if (rank != arg.threadNums - 1)
26
27
            end = start + size;
28
        else
29
            end = arg.end;
30
        double tmp;
        for (int i = start; i < end; i++)
31
32
            tmp += arg.W[i][arg.pos];
33
        pthread_mutex_lock(&myLock);
34
        mean += tmp;
35
        pthread mutex unlock(&myLock);
36
   }
```

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:

```
1 | 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:

```
1
    #pragma omp parallel shared ( mean, w ) private ( i, j ) num_threads(8)
2
3
    #pragma omp for
4
        for (i = 1; i < M - 1; i++)
5
6
          for (j = 1; j < N - 1; j++)
7
8
            w[i][j] = mean;
9
10
11
      }
```

pthread实现:

```
1
    void *changeMat(void *a)
 2
 3
        Arg arg = *(Arg *)a;
 4
        int size = (arg.end - arg.start) / arg.threadNums;
 5
        int rank = arg.rank;
        int start = arg.start + size * rank;
 6
 7
        int end;
        if (rank != arg.threadNums - 1)
8
9
            end = start + size;
10
        else
            end = arg.end;
11
12
        for (int i = start; i < end; i++)
            for (int j = 1; j < arg.pos; j++)
13
                arg.W[i][j] = arg.target;
14
15
   }
```

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:

openmp实现:

```
while ( epsilon <= diff )
 1
 2
 3
    # pragma omp parallel shared ( u, w ) private ( i, j ) num_threads(8)
 4
 5
    /*
      Save the old solution in U.
 6
 7
 8
    # pragma omp for
9
          for (i = 0; i < M; i++)
10
            for (j = 0; j < N; j++)
11
12
13
              u[i][j] = w[i][j];
            }
14
15
          }
16
17
      Determine the new estimate of the solution at the interior points.
18
      The new solution W is the average of north, south, east and west neighbors.
19
    */
20
    # pragma omp for
21
          for (i = 1; i < M - 1; i++)
22
23
            for (j = 1; j < N - 1; j++)
24
25
              w[i][j] = (u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1]) / 4.0;
26
27
28
        }
29
30
      C and C++ cannot compute a maximum as a reduction operation.
31
32
      Therefore, we define a private variable MY_DIFF for each thread.
33
      Once they have all computed their values, we use a CRITICAL section
34
      to update DIFF.
35
36
        diff = 0.0;
37
    # pragma omp parallel shared ( diff, u, w ) private ( i, j, my_diff ) num_threads(8)
38
        {
39
          my_diff = 0.0;
40
    # pragma omp for
41
          for (i = 1; i < M - 1; i++)
42
            for (j = 1; j < N - 1; j++)
43
44
              if ( my_diff < fabs ( w[i][j] - u[i][j] ) )
45
46
                my_diff = fabs (w[i][j] - u[i][j]);
47
48
49
            }
50
    # pragma omp critical
```

```
52
53
             if ( diff < my_diff )</pre>
54
               diff = my_diff;
55
56
57
           }
58
        }
59
60
         iterations++;
61
         if ( iterations == iterations_print )
62
        {
           printf ( " %8d %f\n", iterations, diff );
63
           iterations_print = 2 * iterations_print;
         }
65
      }
66
```

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:

```
1
   void *copyMat(void *a)
 2
    {
 3
        Arg arg = *(Arg *)a;
        int size = (arg.end - arg.start) / arg.threadNums;
 4
 5
        int rank = arg.rank;
 6
        int start = arg.start + size * rank;
 7
        int end;
 8
        if (rank != arg.threadNums - 1)
 9
            end = start + size;
10
        else
11
            end = arg.end;
        for (int i = start; i < end; i++)
12
13
            for (int j = 0; j < arg.pos; j++)
14
                arg.U[i][j] = arg.W[i][j];
15
   }
```

更新矩阵:

```
1
   void *compute(void *a)
 2
 3
        Arg arg = *(Arg *)a;
 4
        int size = (arg.end - arg.start) / arg.threadNums;
 5
        int rank = arg.rank;
        int start = arg.start + size * rank;
 6
7
        int end;
        double **w = arg.W;
8
9
        double **u = arg.U;
10
        if (rank != arg.threadNums - 1)
11
            end = start + size;
```

求矩阵差异:

```
1
    void *findDiff(void *a)
 2
 3
         int key1, key2;
 4
        Arg arg = *(Arg *)a;
 5
         int size = (arg.end - arg.start) / arg.threadNums;
 6
         int rank = arg.rank;
 7
         int start = arg.start + size * rank;
 8
        int end;
 9
        double **w = arg.W;
10
        double **u = arg.U;
11
         if (rank != arg.threadNums - 1)
12
             end = start + size;
13
        else
14
             end = arg.end;
15
         double myDiff = 0.0;
         for (int i = start; i < end; i++)
16
17
         {
             for (int j = 1; j < arg.pos; j++)
18
19
                 if (myDiff < fabs(w[i][j] - u[i][j]))</pre>
20
21
                 {
22
                     myDiff = fabs(w[i][j] - u[i][j]);
23
                     key1 = i;
24
                     key2 = j;
                 }
25
26
             }
27
28
         pthread_mutex_lock(&myLock);
29
         if (diff < myDiff)</pre>
30
             diff = myDiff;
31
         pthread_mutex_unlock(&myLock);
32
   }
```

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:

```
2
 3
   HEATED_PLATE_OPENMP
 4
      C/OpenMP version
 5
      A program to solve for the steady state temperature distribution
 6
      over a rectangular plate.
 7
      Spatial grid of 250 by 250 points.
8
9
      The iteration will be repeated until the change is <= 1.000000e-03
      Number of processors available = 8
10
11
      Number of threads =
12
     MEAN = 74.899598
13
14
15
    Iteration Change
16
17
             1 18.724900
18
             2 9.362450
19
             4 4.096072
20
             8 2.288040
21
            16 1.135841
22
            32 0.567820
23
           64 0.282615
24
           128 0.141682
25
           256 0.070760
26
           512 0.035403
27
          1024 0.017695
          2048 0.008835
28
29
          4096 0.004164
30
          8192 0.001475
31
32
          9922 0.001000
33
34
      Error tolerance achieved.
35
      Wallclock time = 1.479547
36
37
   HEATED_PLATE_OPENMP:
    Normal end of execution.
38
```

N = M = 500:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./heated_plate_openmp.out
 1
2
 3
   HEATED_PLATE_OPENMP
4
     C/OpenMP version
 5
      A program to solve for the steady state temperature distribution
     over a rectangular plate.
 6
 7
 8
      Spatial grid of 500 by 500 points.
9
      The iteration will be repeated until the change is <= 1.000000e-03
10
      Number of processors available = 8
11
      Number of threads =
                                       8
12
13
      MEAN = 74.949900
14
```

```
15
     Iteration Change
16
17
             1 18.737475
18
             2 9.368737
19
            4 4.098823
20
            8 2.289577
            16 1.136604
21
22
            32 0.568201
23
           64 0.282805
24
           128 0.141777
25
           256 0.070808
26
          512 0.035427
27
          1024 0.017707
28
          2048 0.008856
29
         4096 0.004428
30
         8192 0.002210
31
        16384 0.001043
32
33
        16955 0.001000
34
35
      Error tolerance achieved.
36
     Wallclock time = 7.364223
37
   HEATED_PLATE_OPENMP:
38
39
     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:

```
1
    xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./main.out 250 250 1
 2
 3
   HEATED_PLATE_OPENMP
4
     C/Pthread version
 5
      A program to solve for the steady state temperature distribution
 6
    over a rectangular plate.
 7
8
      Spatial grid of 250 by 250 points.
      The iteration will be repeated until the change is <= 1.000000e-03
9
10
      Number of processors available = 1
11
12
     MEAN = 74.899598
13
14
    Iteration Change
15
16
             1 18.724900
17
             2 9.362450
18
             4 4.096072
19
            8 2.288040
20
            16 1.135841
21
           32 0.567820
22
            64 0.282615
23
           128 0.141682
```

```
24
           256 0.070760
25
          512 0.035403
26
          1024 0.017695
          2048 0.008835
27
28
          4096 0.004164
29
          8192 0.001475
30
31
          9922 0.001000
32
33
      Error tolerance achieved.
34
      Wallclock time = 16.726149
35
36
   HEATED_PLATE_OPENMP:
     Normal end of execution.
37
```

N = M = 250, thread_nums = 2:

```
1
   xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./main.out 250 250 2
 2
 3
   HEATED_PLATE_OPENMP
 4
     C/Pthread version
     A program to solve for the steady state temperature distribution
 5
 6
      over a rectangular plate.
 7
      Spatial grid of 250 by 250 points.
 8
 9
      The iteration will be repeated until the change is <= 1.000000e-03
10
      Number of processors available = 2
11
     MEAN = 74.899598
12
13
14
    Iteration Change
15
16
             1 18.724900
             2 9.362450
17
18
             4 4.096072
19
            8 2.288040
20
            16 1.135841
21
            32 0.567820
22
            64 0.282615
23
           128 0.141682
24
           256 0.070760
25
           512 0.035403
26
          1024 0.017695
27
          2048 0.008835
28
          4096 0.004164
29
          8192 0.001475
30
31
          9922 0.001000
32
33
      Error tolerance achieved.
34
      Wallclock time = 14.408228
35
36
   HEATED PLATE OPENMP:
37
      Normal end of execution.
```

```
1
    xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./main.out 250 250 4
 2
 3
    HEATED PLATE OPENMP
 4
      C/Pthread version
 5
      A program to solve for the steady state temperature distribution
 6
      over a rectangular plate.
 7
      Spatial grid of 250 by 250 points.
 8
 9
      The iteration will be repeated until the change is <= 1.000000e-03
      Number of processors available = 4
10
11
12
      MEAN = 74.899598
13
14
     Iteration Change
15
16
             1 18,724900
17
             2 9.362450
18
             4 4.096072
19
             8 2.288040
20
            16 1.135841
21
            32 0.567820
22
            64 0.282615
23
           128 0.141682
24
           256 0.070760
25
           512 0.035403
26
          1024 0.017695
27
          2048 0.008835
28
          4096 0.004164
29
          8192 0.001475
30
31
          9922 0.001000
32
33
      Error tolerance achieved.
34
      Wallclock time = 14.353445
35
36
    HEATED_PLATE_OPENMP:
37
      Normal end of execution.
```

N = M = 250, thread_nums = 8:

```
1
    xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./main.out 250 250 8
 2
 3
   HEATED_PLATE_OPENMP
 4
      C/Pthread version
 5
      A program to solve for the steady state temperature distribution
 6
      over a rectangular plate.
 7
 8
      Spatial grid of 250 by 250 points.
      The iteration will be repeated until the change is <= 1.000000e-03
9
      Number of processors available = 8
10
11
12
      MEAN = 74.899598
```

```
13
14
     Iteration Change
15
             1 18.724900
16
17
             2 9.362450
18
             4 4.096072
19
             8 2.288040
20
            16 1.135841
21
            32 0.567820
22
            64 0.282615
23
           128 0.141682
24
           256 0.070760
25
           512 0.035403
26
          1024 0.017695
27
          2048 0.008835
          4096 0.004164
28
29
          8192 0.001475
30
          9922 0.001000
31
32
33
      Error tolerance achieved.
      Wallclock time = 17.758982
34
35
   HEATED_PLATE_OPENMP:
36
37
      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:

```
1
   xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./main.out 500 500 1
 2
 3
    HEATED PLATE OPENMP
 4
     C/Pthread version
 5
      A program to solve for the steady state temperature distribution
 6
      over a rectangular plate.
 7
 8
      Spatial grid of 500 by 500 points.
 9
      The iteration will be repeated until the change is <= 1.000000e-03
      Number of processors available = 1
10
11
      MEAN = 74.949900
12
13
14
     Iteration Change
15
16
             1 18.737475
17
             2 9.368737
18
             4 4.098823
```

```
19
            8 2.289577
20
           16 1,136604
21
           32 0.568201
           64 0.282805
22
23
           128 0.141777
24
           256 0.070808
          512 0.035427
25
26
          1024 0.017707
          2048 0.008856
27
28
          4096 0.004428
29
          8192 0.002210
30
         16384 0.001043
31
32
         16955 0.001000
33
34
      Error tolerance achieved.
35
      Wallclock time = 46.787368
36
   HEATED_PLATE_OPENMP:
37
38
     Normal end of execution.
```

N = M = 500, thread_nums = 2:

```
1
    xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./main.out 500 500 2
 2
 3
    HEATED_PLATE_OPENMP
     C/Pthread version
 4
 5
      A program to solve for the steady state temperature distribution
      over a rectangular plate.
 6
 7
 8
      Spatial grid of 500 by 500 points.
      The iteration will be repeated until the change is <= 1.000000e-03
 9
10
      Number of processors available = 2
11
12
     MEAN = 74.949900
13
14
     Iteration Change
15
16
             1 18.737475
17
             2 9.368737
18
             4 4.098823
19
            8 2.289577
20
            16 1.136604
21
            32 0.568201
22
           64 0.282805
23
           128 0.141777
24
           256 0.070808
25
           512 0.035427
26
          1024 0.017707
27
          2048 0.008856
28
          4096 0.004428
29
          8192 0.002210
30
         16384 0.001043
31
```

```
32 16955 0.001000
33
34 Error tolerance achieved.
35 Wallclock time = 30.683121
36
37 HEATED_PLATE_OPENMP:
38 Normal end of execution.
```

```
N = M = 500, thread_nums = 4:
```

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP6$ ./main.out 500 500 4
 1
 2
 3
   HEATED_PLATE_OPENMP
     C/Pthread version
 4
 5
      A program to solve for the steady state temperature distribution
     over a rectangular plate.
 6
 7
 8
      Spatial grid of 500 by 500 points.
      The iteration will be repeated until the change is <= 1.000000e-03
 9
      Number of processors available = 4
10
11
     MEAN = 74.949900
12
13
14
    Iteration Change
15
16
             1 18.737475
17
             2 9.368737
18
             4 4.098823
19
             8 2.289577
20
            16 1.136604
21
            32 0.568201
22
           64 0.282805
23
           128 0.141777
24
           256 0.070808
25
           512 0.035427
          1024 0.017707
26
27
          2048 0.008856
28
          4096 0.004428
29
          8192 0.002210
         16384 0.001043
30
31
         16955 0.001000
32
33
      Error tolerance achieved.
34
35
      Wallclock time = 23.673142
36
37
   HEATED_PLATE_OPENMP:
38
      Normal end of execution.
```

N = M = 500, thread_nums = 8:

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

```
C/Pthread version
 4
 5
      A program to solve for the steady state temperature distribution
 6
      over a rectangular plate.
 7
 8
      Spatial grid of 500 by 500 points.
      The iteration will be repeated until the change is <= 1.000000e-03
 9
      Number of processors available = 8
10
11
     MEAN = 74.949900
12
13
14
     Iteration Change
15
16
             1 18.737475
17
             2 9.368737
18
             4 4.098823
19
            8 2.289577
20
            16 1.136604
            32 0.568201
21
22
           64 0.282805
23
           128 0.141777
24
           256 0.070808
25
           512 0.035427
26
          1024 0.017707
27
          2048 0.008856
28
          4096 0.004428
29
          8192 0.002210
30
         16384 0.001043
31
32
         16955 0.001000
33
      Error tolerance achieved.
34
35
      Wallclock time = 29.062525
36
   HEATED_PLATE_OPENMP:
37
```

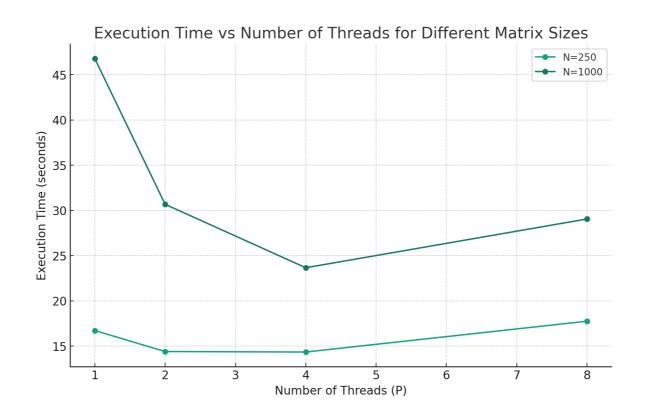
同样, 我们结果的正确性是可以保证的。

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:

$P \setminus N$	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。