中山大学计算机学院本科生实验报告

课程名称：超级计算机组成原理　　任课教师：吴迪

年级	2019	专业（方向）	计科（超级计算）
学号	18324034	姓名	林天皓
开始日期	2021.4.30	完成日期	2021.5.10

一、实验题目

根据 7-Development.pdf 课件在 nbody 问题或者 tsp 问题中**二选一**进行实现，要求实现一个串行版本和 MPI，OpenMP，pthread 中的任意**两种**版本。

二、实验内容

1 实验原理

本次实验中选择 tsp 问题进行计算，使用 openmp 和 mpi 两种并行计算库进行并行计算，采用的数据为 tsp2.txt，通过动态选择其中前 n 行 n 列完成对数据规模的更改。使用程序的运行包括以下几个部分：

Step1:数据读取，先读取文件中的一行获取当前文件中有几列，然后通过二重循环读取文件中点到点的距离邻接矩阵。

```
//readData
FILE *init = fopen("tsp2.txt", "r");
if (init == NULL){
    printf("read file failed\n  EXIT!!\n");
    return 0;}
char *temp = NULL;
size_t len = 999;
getline(&temp, &len, init);
//查找一行有多少数字
for (int i = 0; i < len; i++)
{
    if (temp[i] >= '0' && temp[i] <= '9')
    {
```

```
            Count++;
            while (temp[i] >= '0' && temp[i] <= '9')
                i++;
        }
    }
    rewind(init);//重新设置文件指针到开头
    for (int i = 0; i < Count; i++)
    {
        for (int j = 0; j < Count; j++)
        {
            fscanf(init, "%lf", &dis[i][j]);//读取二维距离
        }
    }
```

Step2:串行与并行的测试循环：由于串行只需要在一个数据规模下执行一次，因此需要放在各种线程数量执行循环之外，本次程序一次性测试 6 种数据规模和 8 种线程数量测试。

```
#define test_case 7
#define thread_case 8
long long thread_tests[thread_case] = {1, 2, 4, 8, 18, 36, 54, 72};
long long tests[test_case] = {8, 9, 10, 11, 12, 13, 14};
```

一次性执行多种测试伪代码如下：

```
for (int test = 0; test < test_case; test++)
    {
        Count = tests[test];
         串行测试;
        for (int th = 0; th < thread_case; th++)
        {
            thread_count = thread_tests[th];
         Openmp 并行测试;
         Pthread 并行测试;
         }
     }
```

Step3:串行测试：设定全局最小值和全局最小路径，使用 dfs 的方法对不同的路径进行测试，当 dfs 到达对应层时候判断是否能更新全局最优解。考虑到 tsp 为一个环路，因此不论我们从哪个店出发都能得到最终的结果，因此默认选择从第 0 个点出发。具体运行代码如下：

```
void serial_TSP()
{
    int vis[N] = {1};
    road[0] = 0;
    for (int i = 1; i < Count; i++)
    {
        vis[i] = 1;
        road[1] = i;
        serial_dfs(i, 2, dis[0][i], road, vis);
        vis[i] = 0;
    }
}
```

串行部分上文中 serial_dfs 部分代码如下：

```
void serial_dfs(int v, int step, int cost, int *road, int *vis)
{
    if (step == Count && cost + dis[v][road[0]] < globalSerialMinCost)
    {
        {
            globalSerialMinCost = cost + dis[v][road[0]];
            memcpy(globalSerialMinRoad, road, sizeof(int) * Count);
        }
        return;
    }
    //if(cost>globalSerialMinCost)return;
    for (int i = 0; i < Count; i++)
    {
        if (vis[i] == 0)
        {
            vis[i] = 1;
            road[step] = i;
            serial_dfs(i, step + 1, cost + dis[v][i], road, vis);
            vis[i] = 0;
        }
    }
}
```

Step4:使用 openmp 进行并行计算。我们使用了静态规划的方法进行并行计算的规划。具体流程如下：枚举我们要经过的第 2 个点和第 3 个点，将程序分 secondLayer = (Count - 1) * (Count - 1)个部分，将这些部分分给定义的 thread_count = thread_tests[th]个线程。由于其中出现了多个线程同时执行的情况，在更新全局最优解时需要考虑临界区的互斥问题，防止获取

错误的数据。代码如下：

```
void Parallel_TSP()
{

#pragma omp parallel for shared(dis, Count) num_threads(thread_count)
    for (int i = 0; i < secondLayer; i++)
    {
        int first = i / (Count - 1) + 1;
        int second = i % (Count - 1) + 1;
        if (first == second)
            continue;
        int localvis[N] = {1};
        int localroad[N] = {0};
        int localParallelMinCost = 0x3f3f3f3f;
        int localParallelMinRoad[N] = {0};
        localvis[first] = 1;
        localvis[second] = 1;
        localroad[1] = first;
        localroad[2] = second;
        parallel_dfs(second, 3, dis[0][first] + dis[first][second], localroad, loc
alvis, localParallelMinRoad, &localParallelMinCost);
        localvis[first] = 0;
        localvis[second] = 0;

#pragma omp critial
        {
            if (globalParallelMinCost > localParallelMinCost)
            {
                globalParallelMinCost = localParallelMinCost;
                memcpy(globalParallelMinRoad, localParallelMinRoad, sizeof(int) *
Count);
            }
        }
    }
}
```

其中 paralleldfs 函数如下：于串行的 dfs 不同的是这里更新的最优解是更新局部的最优解，统一计算完成后再更新全局最优解，可以只需要在一个线程执行完毕才进入临界区更新全局最优解。

```
void parallel_dfs(int v, int step, int cost, int *road, int *vis, int *localSerial
MinRoad, int *localSerialMinCost_p)
{
    if (step == Count && cost + dis[v][road[0]] < *localSerialMinCost_p)
    {
        *localSerialMinCost_p = cost + dis[v][road[0]];
        memcpy(localSerialMinRoad, road, sizeof(int) * Count);
        return;
    }
    //if(cost>*localSerialMinCost_p)return;
    for (int i = 1; i < Count; i++)
    {
        if (vis[i] == 0)
        {
            vis[i] = 1;
            road[step] = i;
            parallel_dfs(i, step + 1, cost + dis[v][i], road, vis, localSerialMinR
oad, localSerialMinCost_p);
            vis[i] = 0;
        }
    }
}
```

Step5:pthread 的并行计算：使用与上文中相同的并行分配方式，不同的是我们这里需要自行计算每个线程中需要计算的 my_first_i 和 my_last_i,然后线程先计算得到自身局部的最优解，再通过信号量的方式解决更新全局解的互斥问题。实现代码如下：

```
void pthread_TSP()
{
    thread_handles = (pthread_t *)malloc(thread_count * sizeof(pthread_t));
    long thread;
    sem_init(&sem, 0, 1);
    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL,
                       (void *)pthread_TSP_func, (void *)thread);

    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);
}
    //if(cost>*localSerialMinCost_p)return;
    for (int i = 1; i < Count; i++)
    {
        if (vis[i] == 0)
```

```
        {
            vis[i] = 1;
            road[step] = i;
            pthread_dfs(i, step + 1, cost + dis[v][i], road, vis, localSerialMinRo
ad, localSerialMinCost_p);
            vis[i] = 0;
        }
    }
}
```

其中每个线程执行的函数如下，通过自己的 rank 计算自身获得的并行部分的局部最优解，然后通过 sem 互斥更新全局最优解。

```
void pthread_TSP_func(void *rank)
{
    long my_rank = (long)rank;
    long long i;
    long long my_n = secondLayer / thread_count;
    long long my_first_i = my_n * my_rank;
    long long my_last_i = my_first_i + my_n;
    for (i = my_first_i; i < my_last_i; i++)
    {
        int first = i / (Count - 1) + 1;
        int second = i % (Count - 1) + 1;
        if (first == second)
            continue;
        int localvis[N] = {1};
        int localroad[N] = {0};
        int localPthreadMinCost = 0x3f3f3f3f;
        int localPthreadMinRoad[N] = {0};
        localvis[first] = 1;
        localvis[second] = 1;
        localroad[1] = first;
        localroad[2] = second;
        parallel_dfs(second, 3, dis[0][first] + dis[first][second], localroad, loc
alvis, localPthreadMinRoad, &localPthreadMinCost);
        localvis[first] = 0;
        localvis[second] = 0;

        sem_wait(&sem);
        if (globalPthreadMinCost > localPthreadMinCost)
        {
            globalPthreadMinCost = localPthreadMinCost;
            memcpy(globalPthreadMinRoad, localPthreadMinRoad, sizeof(int) * Count)
;
        }
```

```
        sem_post(&sem);
    }
}
```

该部分中的 pthread_dfs 实现与在 openmp 中的 dfs 实现方式相同，此处不再赘述。

至此，我们完成了对实验代码的编写，下面运行测试。

三、实验结果

测试环境说明：

本次实验使用的 CPU 为 Intel(R) Xeon(R) Gold 6150 CPU @ 2.70GHz　18 核心 36 线程

编译器版本为 gcc version 9.2.0

本次实验中并行测试使用了 8 种线程数量，分别为 1，　2，4，8，18，36，54，72.

本次实验中测试使用了 7 种计算数量，分别为 8, 9, 10, 11, 12, 13, 14,。

使用的数据为老师提供的 tsp2.txt，根据计算数量分别取前 n 行 n 列。

编译运行

```
cpn281 ~/asc21/lth/hpc/hw5 =Σ(((  つ^ω^)つ gcc tsp.c -o tsp -lm -lpthread -fopenmp && ./tsp > result2.txt
<<<<<<<<<<<< start new Count: 8 >>>>>>>>>>
################## start new thread_nums: 1 #################
################## start new thread_nums: 2 #################
################## start new thread_nums: 4 #################
################## start new thread_nums: 8 #################
################## start new thread_nums: 18 ################
################## start new thread_nums: 36 ################
################## start new thread_nums: 54 ################
################## start new thread_nums: 72 ################
<<<<<<<<<<<< start new Count: 9 >>>>>>>>>>
################## start new thread_nums: 1 #################
################## start new thread_nums: 2 #################
################## start new thread_nums: 4 #################
################## start new thread_nums: 8 #################
################## start new thread_nums: 18 ################
################## start new thread_nums: 36 ################
################## start new thread_nums: 54 ################
################## start new thread_nums: 72 ################
<<<<<<<<<<<< start new Count: 10 >>>>>>>>>>
################## start new thread_nums: 1 #################
################## start new thread_nums: 2 #################
```

图 1-编译运行结果（节选）

查看结果（节选）：

```
ln101 ~/asc21/lth/hpc/hw5 =Σ((( つ^ω^)つ cat result2.txt
Count 17
<<<<<<<<<<<< start new Count: 8 >>>>>>>>>>
 ============  Serial time  ==============
 The elapsed time is 3.519058e-04 seconds
get New best cost1346
0 3 2 1 4 5 7 6
91 228 390 227 267 34 29 80

################## start new thread_nums: 1 #################
 ====  OpenMP time  thread_nums:1 ====
 The elapsed time is 3.778934e-04 seconds
get New best cost1346
0 3 2 1 4 5 7 6
91 228 390 227 267 34 29 80
```

图 2-查看运行结果（节选）

运行结果表如下：

各个数量的运算结果如下：

n	tsp 距离	路径
8	1346	0 3 2 1 4 5 7 6 0
9	1472	0 3 8 1 4 2 5 7 6 0
10	1637	0 3 8 4 1 9 2 5 7 6 0
11	1639	0 3 8 4 1 9 10 2 5 7 6 0
12	1799	0 3 11 8 4 1 9 10 2 5 7 6 0
13	1805	0 6 7 5 2 10 9 1 4 8 11 3 12 0
14	1872	0 6 7 5 13 2 10 9 1 4 8 11 3 12 0

表 1-运行结果汇总

同样的使用上次实验四中编写的 python 代码进行绘图，此处不再赘述。

串行排序时间如下：

运算规模	8	9	10	11	12	13	14
运行时间	3.52E-04	2.96E-03	2.85E-02	3.02E-01	3.49E+00	4.38E+01	6.01E+02

表 2-串行运行时间

openmp 并行排序运行时间如下

运算规模\ 线程数量\ 时间	8	9	10	11	12	13	14
1	3.78E-04	2.95E-03	2.74E-02	2.94E-01	3.45E+00	4.40E+01	5.98E+02
2	1.99E-04	1.73E-03	1.40E-02	1.47E-01	1.73E+00	2.20E+01	3.00E+02
4	1.33E-04	7.79E-04	6.96E-03	8.52E-02	8.78E-01	1.10E+01	1.50E+02
8	1.02E-04	4.25E-04	4.74E-03	4.14E-02	4.48E-01	5.67E+00	7.69E+01
18	1.65E-04	3.82E-04	2.79E-03	2.41E-02	2.25E-01	2.70E+00	3.84E+01
36	2.54E-02	2.18E-02	2.45E-02	2.17E-02	1.37E-01	1.62E+00	2.02E+01
54	4.53E-03	2.63E-03	2.32E-03	1.34E-02	1.39E-01	1.78E+00	2.22E+01
72	3.07E-04	4.88E-04	1.80E-03	1.33E-02	1.18E-01	1.47E+00	2.05E+01

表 3-openmp 各个线程数量与运算规模运行时间

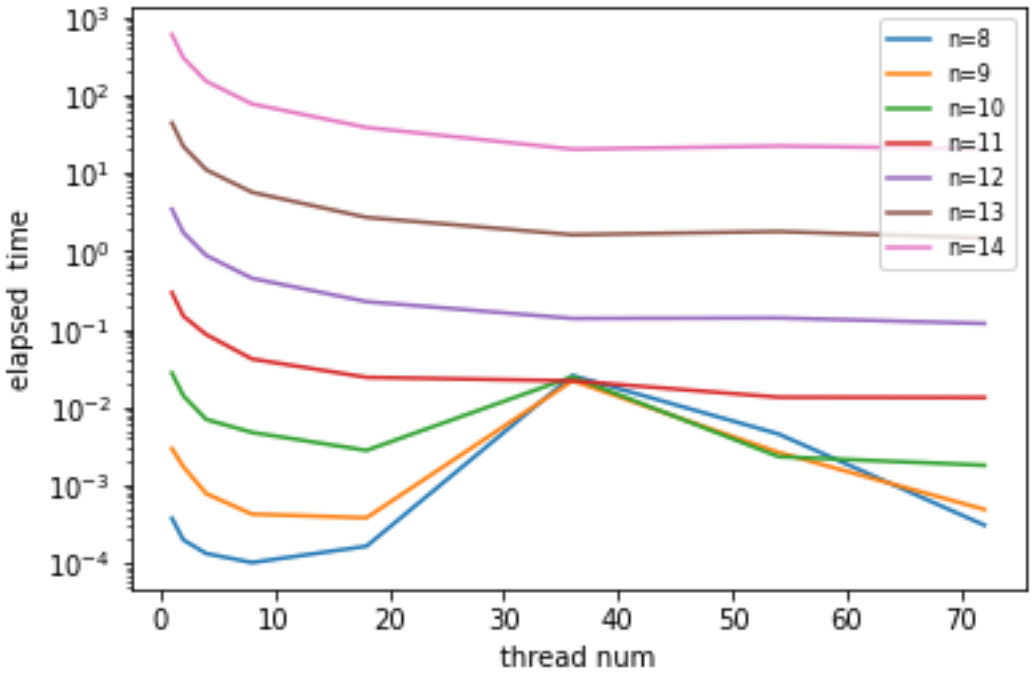

图 3 – openmp 运算数量与运行时间关系（对数坐标）

分析：n 越小，线程数量越小，运行时间越小，在线程数量为 36，n 较小时，与 CPU 线程数量相同时候时间运行时间最慢，原因未知。

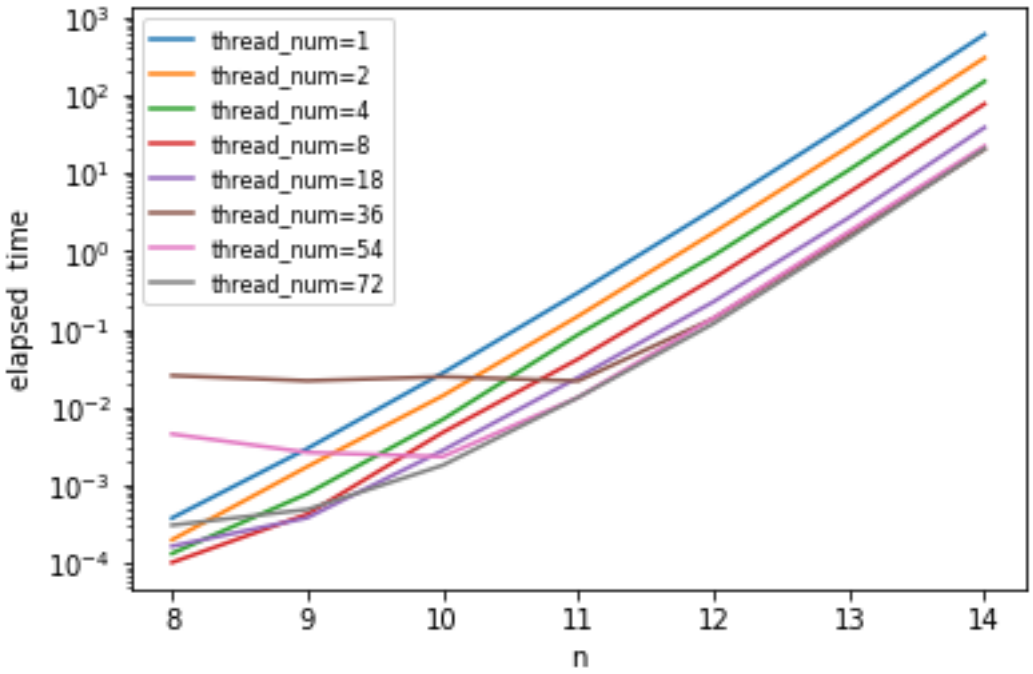

图 4 – openmp 线程数量与运行时间关系（对数坐标）

分析：线程数量越大，在 n 较小时由于线程创建开销导致运行时间较大。

pthread 并行排序运行时间如下

运算规模\线程数量\时间	8	9	10	11	12	13	14
1	4.38E-04	2.98E-03	3.73E-02	3.31E-01	3.47E+00	4.41E+01	5.99E+02
2	3.51E-04	1.53E-03	1.39E-02	1.49E-01	1.73E+00	2.20E+01	3.00E+02
4	2.38E-04	7.87E-04	9.45E-03	7.86E-02	8.81E-01	1.10E+01	1.50E+02
8	2.51E-04	6.17E-04	4.75E-03	5.34E-02	4.45E-01	5.67E+00	7.72E+01
18	3.60E-04	3.94E-03	2.35E-03	2.19E-02	1.96E-01	2.69E+00	3.49E+01
36	2.87E-03	7.97E-03	9.75E-03	2.28E-02	1.18E-01	1.55E+00	1.68E+01
54	1.50E-03	1.28E-03	1.51E-03	9.36E-03	1.01E-01	1.14E+00	1.90E+01
72	1.92E-03	1.85E-03	1.91E-03	7.47E-03	6.89E-02	1.40E+00	1.57E+01

表 4-openmp 各个线程数量与运算规模运行时间

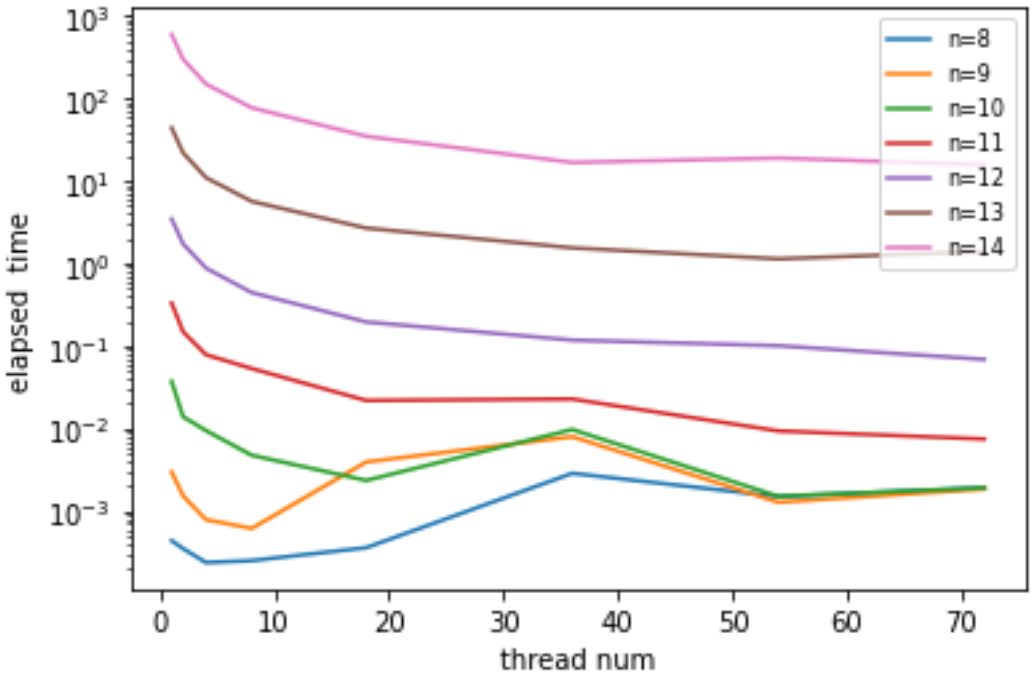

图 5 – pthread 运算数量与运行时间关系（对数坐标）

分析：n 越小，线程数量越小，运行时间越小，在线程数量为 36，n 较小时，与 CPU 线程数量相同时候时间运行时间最慢，原因未知。

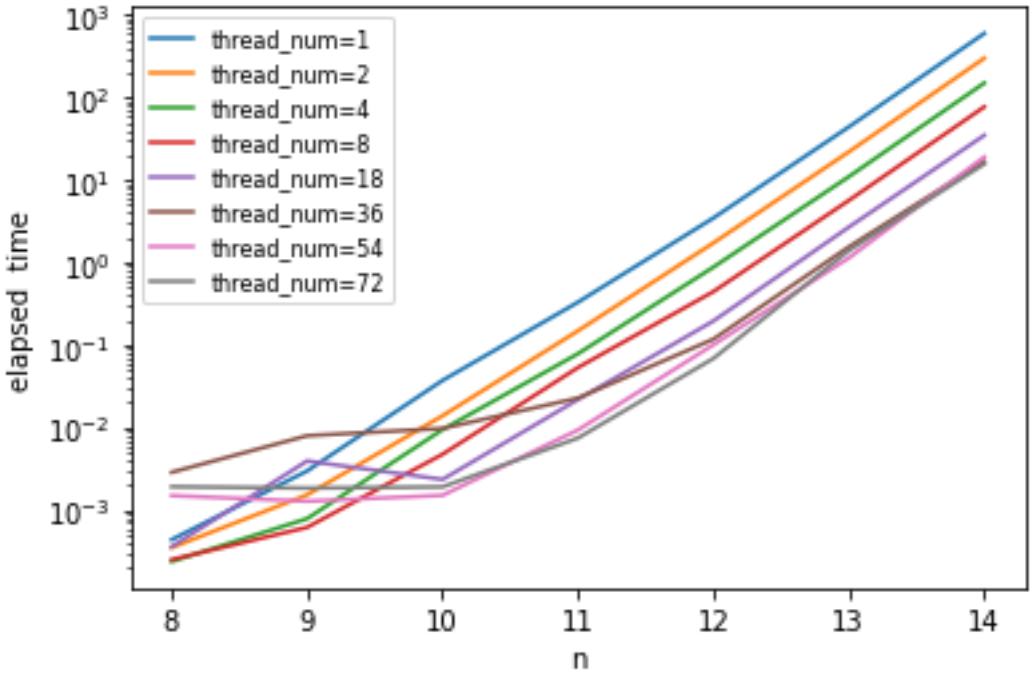

图 6 – pthread 线程数量与运行时间关系（对数坐标）

分析：线程数量越大，在 n 较小时由于线程创建开销导致运行时间较大。

openmp 并行排序运行加速比如下

运算规模\ 线程数量\ 时间	8	9	10	11	12	13	14
1	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00
2	1.90E+00	1.70E+00	1.96E+00	2.00E+00	2.00E+00	2.00E+00	2.00E+00
4	2.84E+00	3.79E+00	3.94E+00	3.45E+00	3.92E+00	3.99E+00	3.99E+00
8	3.71E+00	6.94E+00	5.79E+00	7.11E+00	7.69E+00	7.75E+00	7.77E+00
18	2.29E+00	7.73E+00	9.82E+00	1.22E+01	1.53E+01	1.63E+01	1.56E+01
36	1.49E-02	1.36E-01	1.12E+00	1.35E+01	2.51E+01	2.72E+01	2.96E+01
54	8.34E-02	1.12E+00	1.18E+01	2.19E+01	2.47E+01	2.47E+01	2.69E+01
72	1.23E+00	6.05E+00	1.52E+01	2.21E+01	2.92E+01	3.00E+01	2.92E+01

表 5-openmp 各个线程数量与运算规模加速比

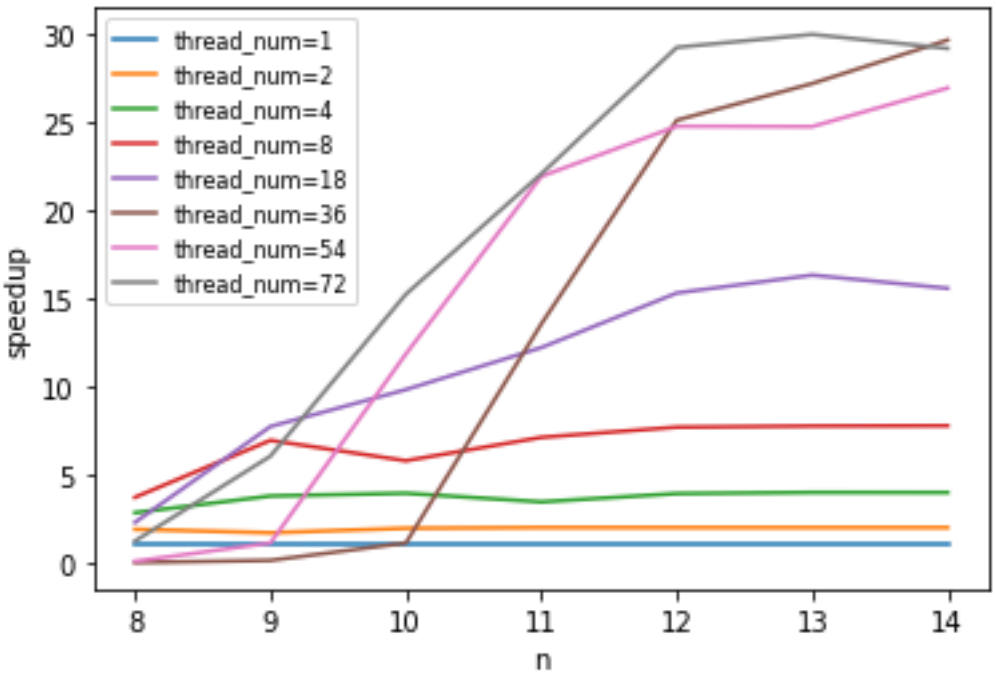

图 7 – openmp 线程数量与加速比关系

分析:

由图可得线程数越多，加速比近似越大。对于同一线程数，n 越大，加速比越大。

对于同一线程数量，当 n 越大时，加速比增加幅度越小。加速比不会超过 CPU 线程数量。

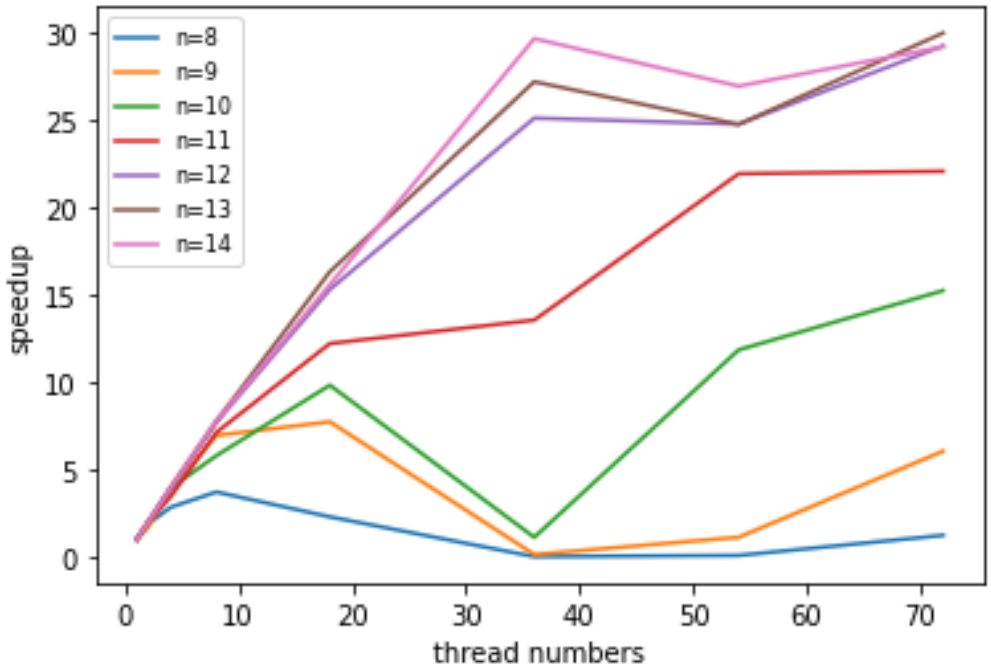

图 8 – openmp 线程数量与加速比关系

pthread 并行排序运行加速比如下

运算规模\线程数量\时间	8	9	10	11	12	13	14
1	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00
2	1.25E+00	1.94E+00	2.69E+00	2.22E+00	2.01E+00	2.00E+00	2.00E+00
4	1.84E+00	3.79E+00	3.94E+00	4.20E+00	3.94E+00	4.00E+00	3.99E+00
8	1.74E+00	4.83E+00	7.84E+00	6.20E+00	7.80E+00	7.77E+00	7.76E+00
18	1.22E+00	7.56E-01	1.58E+01	1.51E+01	1.77E+01	1.64E+01	1.72E+01
36	1.53E-01	3.74E-01	3.82E+00	1.45E+01	2.94E+01	2.84E+01	3.56E+01
54	2.92E-01	2.32E+00	2.46E+01	3.53E+01	3.43E+01	3.88E+01	3.15E+01
72	2.28E-01	1.61E+00	1.95E+01	4.43E+01	5.04E+01	3.15E+01	3.82E+01

表 6-pthread 各个线程数量与运算规模加速比

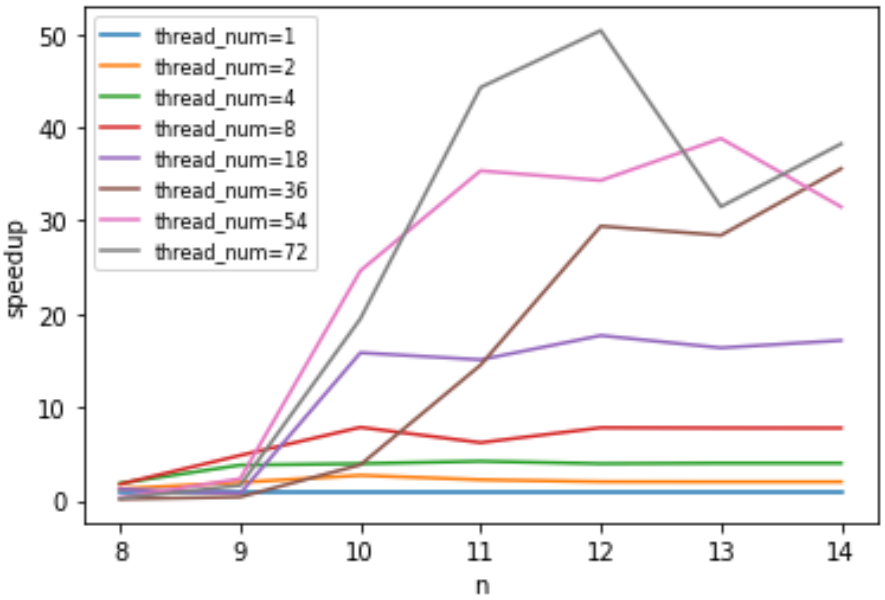

图 5 – pthread 线程数量与加速比关系

分析：由图可得线程数越多，加速比近似越大。对于同一线程数，n 越大，加速比越大。但是这里出现了超过 CPU 时间的加速比现象，原因是我们使用线程数量为 1 去作比较，由于某种原因导致线程数量为 1 运行时候的时间偏长，会导致这里的计算误差。

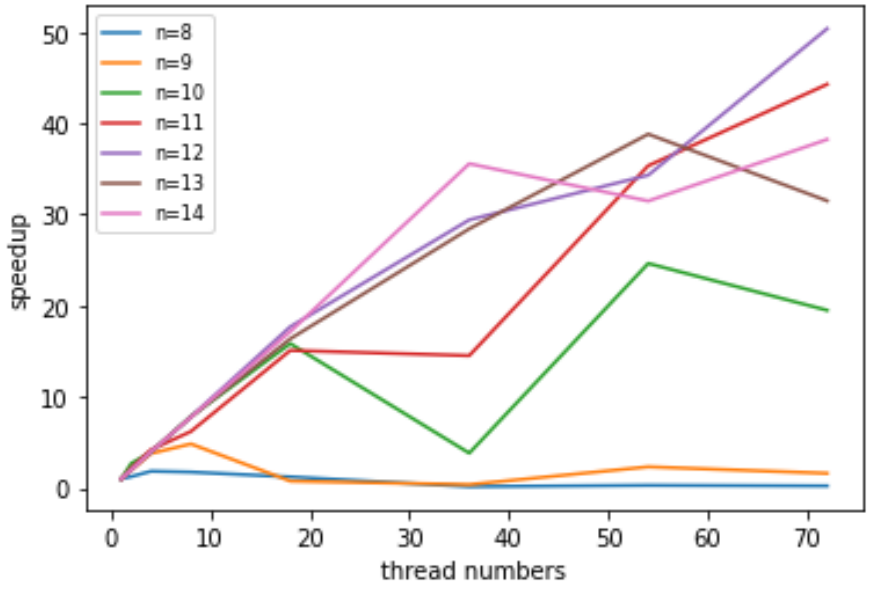

图 6 – pthread 运算规模与加速比关系

分析：由图可见当 n 较小时，加速比随着线程数先增加后减少，对于每一个 n 都有一个最优的线程数量，这个数量随着 n 的增加而增加。超过这个线程数量以后，因为过多线程调用的开销，会导致加速比下降。而 n 较大时，线程数量的增加仍然导致加速比的近似提升。

openmp 并行排序运行效率如下

运算规模\线程数量\时间	8	9	10	11	12	13	14
1	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00
2	9.49E-01	8.51E-01	9.80E-01	9.98E-01	9.98E-01	9.98E-01	9.98E-01
4	7.10E-01	9.47E-01	9.85E-01	8.64E-01	9.81E-01	9.97E-01	9.97E-01
8	4.64E-01	8.68E-01	7.23E-01	8.89E-01	9.61E-01	9.69E-01	9.71E-01
18	1.27E-01	4.29E-01	5.45E-01	6.78E-01	8.50E-01	9.06E-01	8.65E-01
36	4.13E-04	3.77E-03	3.12E-02	3.76E-01	6.97E-01	7.55E-01	8.23E-01
54	1.54E-03	2.08E-02	2.19E-01	4.06E-01	4.58E-01	4.58E-01	4.99E-01
72	1.71E-02	8.40E-02	2.11E-01	3.06E-01	4.06E-01	4.16E-01	4.05E-01

表 7-openmp 各个线程数量与运算规模效率

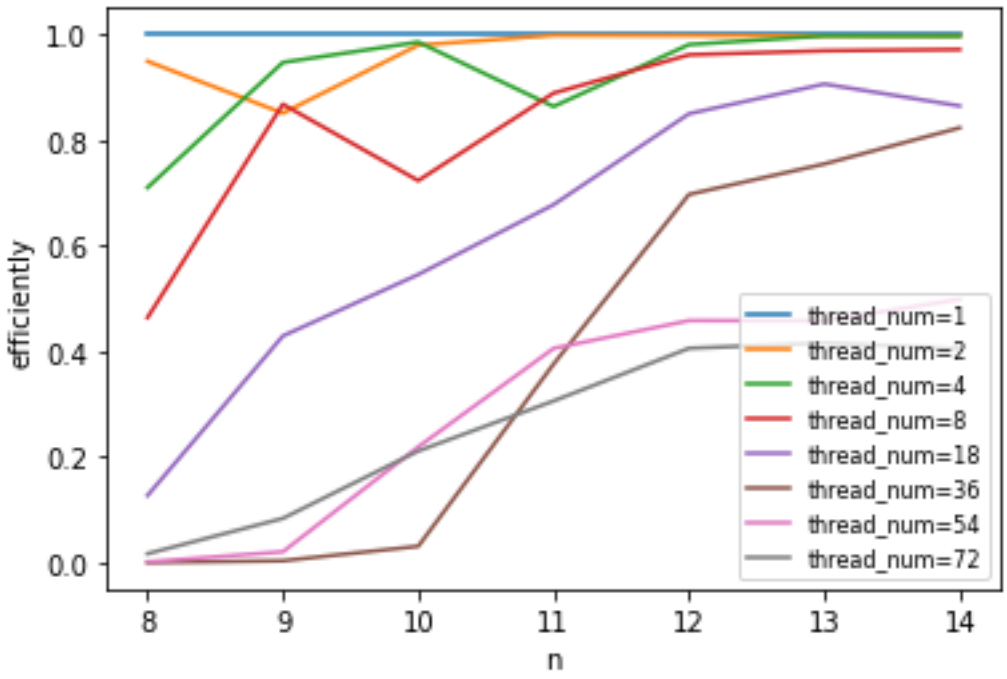

图 7 – openmp 线程数量与效率关系

分析：由图可见，当线程数越小时，效率越高。

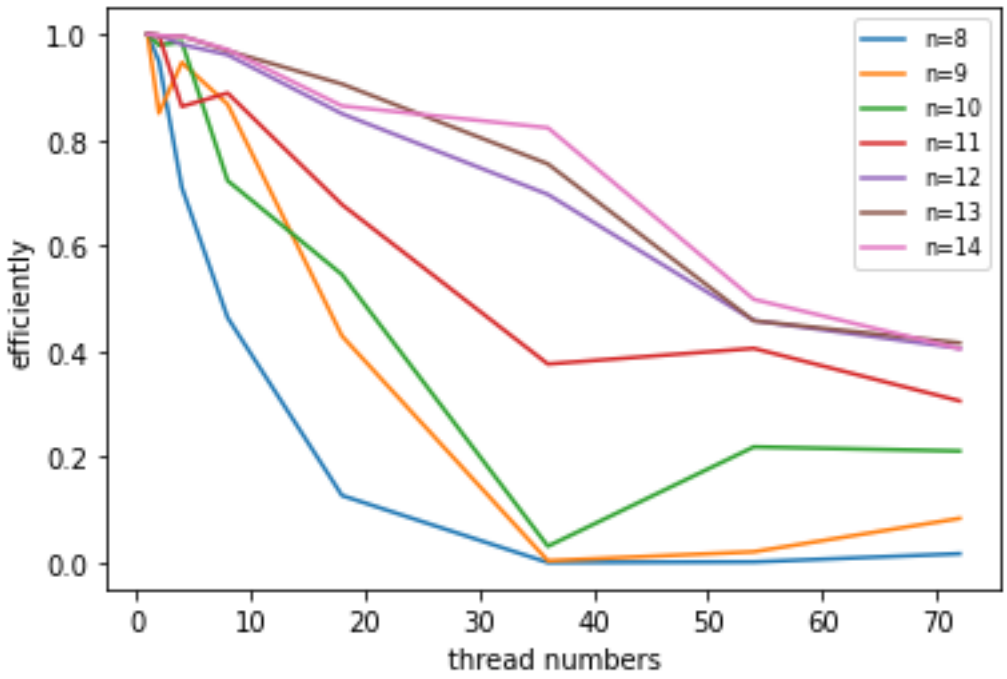

图 8－ openmp 运算数量与效率关系

分析：由图可见，当 n 越大时，效率近似越高。

pthread 并行排序运行效率如下

运算规模\线程数量\时间	8	9	10	11	12	13	14
1	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00
2	6.24E-01	9.71E-01	1.34E+00	1.11E+00	1.00E+00	1.00E+00	9.99E-01
4	4.60E-01	9.47E-01	9.85E-01	1.05E+00	9.86E-01	9.99E-01	9.98E-01
8	2.18E-01	6.04E-01	9.80E-01	7.75E-01	9.75E-01	9.71E-01	9.70E-01
18	6.76E-02	4.20E-02	8.80E-01	8.38E-01	9.82E-01	9.09E-01	9.53E-01
36	4.25E-03	1.04E-02	1.06E-01	4.03E-01	8.17E-01	7.89E-01	9.88E-01
54	5.40E-03	4.30E-02	4.56E-01	6.54E-01	6.35E-01	7.19E-01	5.83E-01
72	3.17E-03	2.24E-02	2.71E-01	6.15E-01	7.00E-01	4.37E-01	5.31E-01

表 8-pthread 各个线程数量与运算规模效率

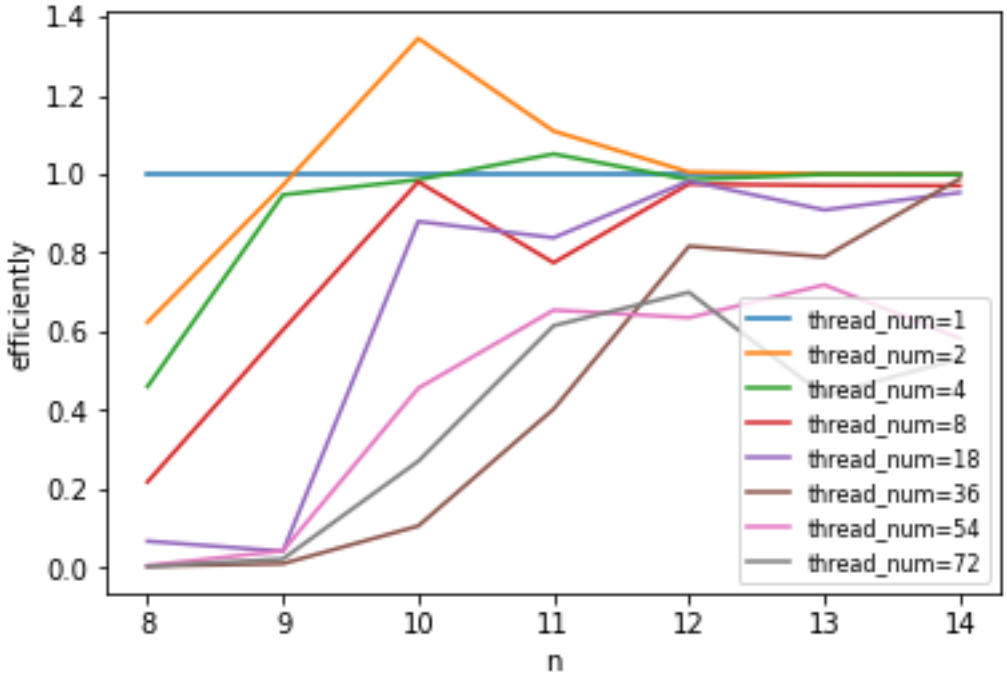

图 9 – pthread 线程数量与效率关系

分析：由图可见，当线程数越小时，效率越高。

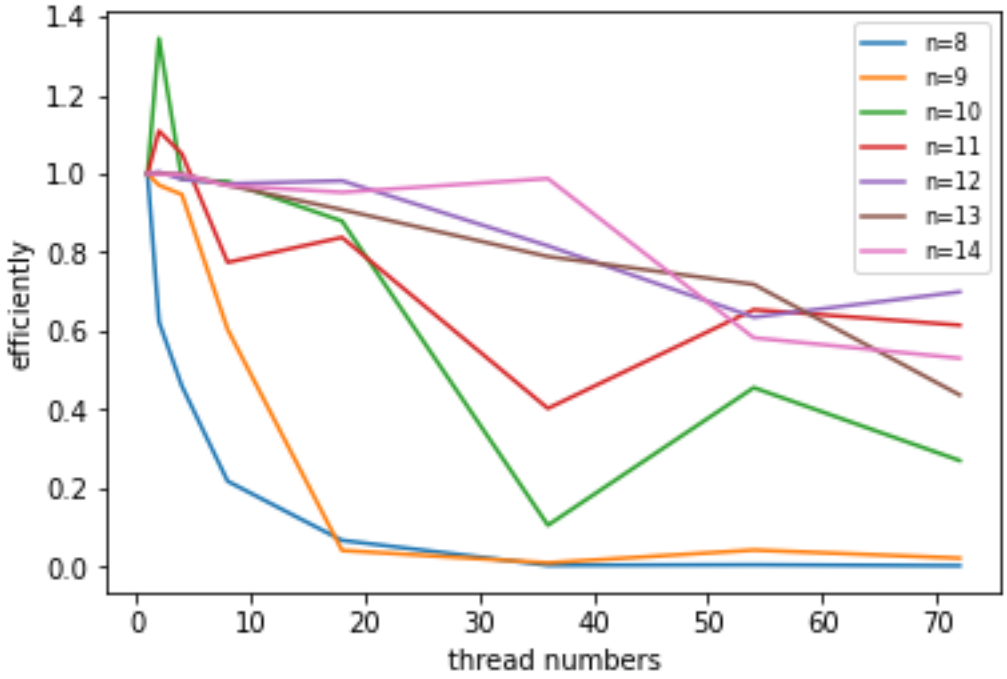

图 10 – pthread 线程数量与效率关系

分析：n 越大，效率越高。线程数量越多，效率越低。

总结：结合以上信息，使用 openmp 与 pthread 对比，在低线程数量高运算规模的情况下都可获得近似线性的加速比，而对于线程数量较多的情况，相同条件下 pthread 能获得约 15% 的性能提升。这可能和 pthread 的运算更加连续，对于空间局部性的把握更好，更有利于 CPU 缓存的命中。

综上，本次实验完成了使用深度优先遍历的方式计算 tsp 问题的串行版本，以及使用 pthread 和 openmp 的并行版本。

实验感想

本次实验使用 openmp 和 pthread 使用深度优先搜索算法进行并行计算 tsp 问题。本次使实验中，为了更好的对比并行算法，没有加入剪枝，从而使得并行和串行的计算量基本相同，更好的进行量化对比。

本次实验中使用静态规划并行任务的方式，相比于动态规划并行任务的方法，这种方式可以避免运行过程中对内存的动态分配和释放，而且不需要线程之间互相检测空闲或者发送任务。但是在运行过程中有一段时间会出现 cpu 不饱和的情况。

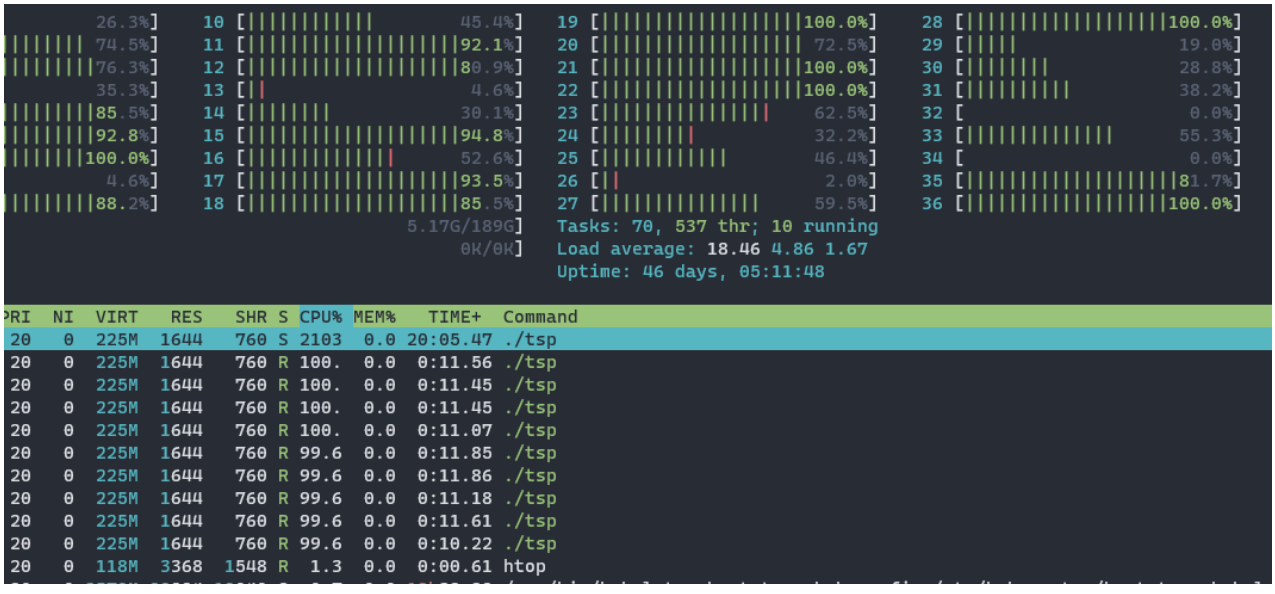

图 10-使用 54 线程时候不能全程跑满 CPU

这也是静态规划的缺点，在子任务数量不能被线程数量均摊时，会出现线程完成时间不一，CPU 闲置的情况，造成并行度的下降。

在使用 openmp 进行并行计算的过程十分方便，只需要指定并行部分的私有变量即可按照所需并行，然后可以通过 criticial 语句方便的添加临界区，省略了手动加锁解锁的麻烦。与 pthread 对比，pthread 需要自行计算每个线程所需要计算的部分，需要手动的给临界区加锁解锁，实现较为复杂。而使用 MPI 则更加复杂，因为 openmp 和 pthread 都可以在编译时自行制定线程数量，而 MPI 则需要运行时通过 mpirun 指定线程数量，因此我们编写的程序需要适应各种可能的线程数，对于并行算法的编写更复杂。