

高性能计算程序设计基础 秋季2021

任务1:

通过实验4构造的基于Pthreads的parallel_for函数替换fft_serial应用中的某些计算量较大的“for循环”,实现for循环分解、分配和线程并行执行。

1) 更改函数step()

代码

完整代码见lab5_1_1.c

代码思路:

- 在函数step中调用 parallel_for() 函数
- parallel_for() 中使用pthread创建线程
- 每个线程中调用 step_parallel() 函数。

并将函数 step_parallel() 指针作为参数传入还有并行中需要用到变量写在设置的一个结构体里, 也作为参数传入 parallel_for() 函数。并在 parallel_for() 函数中

- struct for_index

由于本次实验尝试了两种并行, 一种是在 step() 中调用 parallel_for() 实现并行, 还有一种是在 cfft2() 中调用 parallel_for() 实现并行, 因此里面有些变量是在 step() 中要用到的, 有些是在 cfft2() 中要用到的, 都写在这个函数中了, 没有用到变量并不影响计算。

```
1 struct for_index {
2     int start;
3     int end;
4     int increment;
5     int mj; // need in cfft2 and step
6     double *a; // need in step
7     double *b; // need in step
```

```

8     double *c;//need in step
9     double *d;//need in step
10    double *x;// need in cfft2
11    double *y;// need in cfft2
12    double *w;// need in cfft2 and step
13    double sgn;// need in cfft2 and step
14    unsigned long time; //用来传递每个线程的用时，在判断并行速度变慢时可以用到，
    一般不被用到。
15 };

```

• step()

这个函数主要做的就是定义一个结构体 `first`，并且将需要用到的 `mj`、`a`、`b`、`c`、`d`、`w`、`sgn` 等变量赋值给结构体中的变量（实际上像 `w`、`sgn` 等这些变量也可以设为全局变量，并不影响接下来的计算），最后调用函数 `parallel_for()`，简单介绍里面的参数会在接下来介绍 `parallel_for()` 的时候具体解释。

被注释掉的代码是用来计算调用这个函数花费的时间，精确到微秒。

```

1 void step_ ( int n, int mj, double a[], double b[], double c[],
2             double d[], double w[], double sgn )
3 {
4     // struct timeval val;
5     // struct timeval newVal;
6     // int ret = gettimeofday(&val, NULL);
7     int lj;
8     lj = n / (mj*2);
9     struct for_index first;
10    first.mj = mj;
11    first.a = a;
12    first.b = b;
13    first.c = c;
14    first.d = d;
15    first.w = w;
16    first.sgn = sgn;
17
18    parallel_for(0,lj,1,step_parallel,&first,num_thread);
19
20    // ret = gettimeofday(&newVal, NULL);
21    // unsigned long diff = (newVal.tv_sec * Converter + newVal.tv_usec) -
    (val.tv_sec * Converter + val.tv_usec);
22    // printf("diff:  sec --- %ld, usec --- %ld\n", diff / Converter, diff
    % Converter);
23    return;
24 }

```

• parallel_for()

- 这个函数的参数分别是并行的起始位置 `start`、并行的结束位置 `end`，每次迭代的增量 `increment`，线程调用的函数指针，需要用到的打包进来的参数 `arg`、以及线程数 `num_threads`。
- 第2-3行，由于这里传入的参数 `arg` 是一个结构体，里面的变量都要分配到每个线程上去，因此要进行类型转换；
- 第5-6行，将 $(end - start)$ 个迭代次数按照线程数进行划分，每个线程能分到最少 `divid` 个迭代次数。
- 第8-24行，给要传递到每个线程里的结构体 `thread_assign[i]` 赋值变量：

- 第10-12行是防止总迭代次数不能被线程数整除，即有一些迭代可能无法被分配给任一线程，因此将最后剩余未被分配的迭代次数全部分配给最后一个线程。
- 其余内容就是给结构体赋值，这些变量是在每个线程中都会用到的。
- 第31-40行，进行并行：
 - 第31行，给每个线程定义一个标识符；
 - 第33-35行，创建线程，传入的参数第一个是线程的标识符，第三个参数是每个线程调用的函数的指针，最后一个参数是调用的函数的参数；
 - 第37-40行，线程运行结束，释放资源；
- 被注释掉的内容实际上是在计算“创建线程的开销”，这是为了分析并行化后速度变慢的原因：
 - 第26-29行、第42-43行计算从线程创建到结束线程释放资源花费的时间 `diff`；
 - 进入线程调用的函数后也会对整个过程进行计时，将记录下来的每个线程调用的函数中进行计算所花费的时间写入传入的结构体的 `time` 里（这里传入的是一个指针，因此结构体里的变量更改后在被调用的函数外也可以被获得）；
 - 在第39行，释放每个线程的资源后，将所有线程真正在调用的函数中计算的时间进行求和，得到 `time`；
 - 第44行，输出微秒级的 `diff - time`，即输出的是创建线程和释放资源的开销时间；

```

1 void parallel_for(int start, int end, int increment, void *(*functor)
  (void*), void *arg, int num_threads){
2     struct for_index *first;
3     first = (struct for_index*)arg;
4
5     int divid = (end - start)/num_threads;
6     struct for_index thread_assign[num_threads];
7
8     for(int i=0; i<num_threads; ++i){
9         thread_assign[i].start = i*divid;
10        if(i == (num_threads - 1) ){
11            thread_assign[i].end = end;
12        }
13        else{
14            thread_assign[i].end = thread_assign[i].start + divid;
15        }
16        thread_assign[i].increment = increment;
17        thread_assign[i].mj = first->mj;
18        thread_assign[i].a = first->a;
19        thread_assign[i].b = first->b;
20        thread_assign[i].c = first->c;
21        thread_assign[i].d = first->d;
22        thread_assign[i].w = first->w;
23        thread_assign[i].sgn = first->sgn;
24    }
25
26    // struct timeval val;
27    // struct timeval newVal;
28    // unsigned long time = 0;
29    // int ret = gettimeofday(&val, NULL);
30
31    pthread_t pth[num_threads];
32
33    for(int i=0; i<num_threads; ++i){
34        pthread_create(&pth[i], NULL, functor, &thread_assign[i]);
35    }
36
37    for(int i=0; i<num_threads; ++i){

```

```

38     pthread_join(pth[i], NULL);
39     // time += thread_assign[i].time;
40 }
41
42 // ret = gettimeofday(&newVal, NULL);
43 // unsigned long diff = (newVal.tv_sec * Converter + newVal.tv_usec)
- (val.tv_sec * Converter + val.tv_usec);
44 // printf("creat pthread cost time :usec --- %ld\n", diff - time);
45
46 }

```

- **step_parallel()**

首先在第一行，表示这是一个函数指针，传入的参数是arg;

在原本的 step() 中并没有什么数据依赖，每个线程运行的过程互不相关，因此这部分代码相较于并行主要更改的地方是：

- 第21-48行的for循环中：
 - start、end、increment均是传入的参数中已经被分配好的变量;
 - 需要用到的之前赋值过的变量，已经被作为参数传入，只需要在前面加上 `thread_assign->` 表示使用的变量来自传入的结构体指针。
- 第2-4行、第50-52行计算整个函数运算所花费的时间，并写入结构体里的变量 `time` 中。

```

1 void *step_parallel(void *arg){
2     struct timeval val;
3     struct timeval newVal;
4     int ret = gettimeofday(&val, NULL);
5
6     double ambr;
7     double ambu;
8     int j;
9     int ja;
10    int jb;
11    int jc;
12    int jd;
13    int jw;
14    int k;
15    int lj;
16    double wjw[2];
17
18    struct for_index *thread_assign;
19    thread_assign = (struct for_index*)arg;
20
21    for (j = thread_assign->start; j < thread_assign->end;
j=j+thread_assign->increment )
22    {
23        jw = j * thread_assign->mj;
24        ja = jw;
25        jb = ja;
26        jc = j * (thread_assign->mj * 2);
27        jd = jc;
28
29        wjw[0] = thread_assign->w[jw*2+0];
30        wjw[1] = thread_assign->w[jw*2+1];
31
32        if ( thread_assign->sgn < 0.0 )

```

```

33     {
34         wjw[1] = - wjw[1];
35     }
36
37     for (int k = 0; k < thread_assign->mj; k++ )
38     {
39         thread_assign->c[(jc+k)*2+0] = thread_assign->a[(ja+k)*2+0] +
thread_assign->b[(jb+k)*2+0];
40         thread_assign->c[(jc+k)*2+1] = thread_assign->a[(ja+k)*2+1] +
thread_assign->b[(jb+k)*2+1];
41
42         ambr = thread_assign->a[(ja+k)*2+0] - thread_assign-
>b[(jb+k)*2+0];
43         ambu = thread_assign->a[(ja+k)*2+1] - thread_assign-
>b[(jb+k)*2+1];
44
45         thread_assign->d[(jd+k)*2+0] = wjw[0] * ambr - wjw[1] * ambu;
46         thread_assign->d[(jd+k)*2+1] = wjw[1] * ambr + wjw[0] * ambu;
47     }
48 }
49
50 ret = gettimeofday(&newVal, NULL);
51 unsigned long diff = (newVal.tv_sec * Converter + newVal.tv_usec) -
(val.tv_sec * Converter + val.tv_usec);
52 thread_assign->time = diff ;
53 }

```

运行结果

设置一个全局变量线程数为4，并运行：

```

wwwj@ubuntu:~/lab/lab5/fft_serial$ gcc -o lab5_1_1 lab5_1_1.c -lpthread -lm
wwwj@ubuntu:~/lab/lab5/fft_serial$ ./lab5_1_1
num_thread: 4
29 November 2021 04:44:13 AM

FFT_SERIAL
C version

Demonstrate an implementation of the Fast Fourier Transform
of a complex data vector.

Accuracy check:

FFT ( FFT ( X(1:N) ) ) == N * X(1:N)

      N      NITS      Error      Time      Time/Call      MFLOPS
      2      10000  7.859082e-17  5.221474e+00  2.610737e-04  0.038303
      4      10000  1.209837e-16  1.062326e+01  5.311630e-04  0.075306
      8      10000  6.820795e-17  1.566873e+01  7.834365e-04  0.153171
     16      10000  1.438671e-16  2.381166e+01  1.190583e-03  0.268776
     32      1000  1.331210e-16  3.318266e+00  1.659133e-03  0.482180
     64      1000  1.776545e-16  4.011962e+00  2.005981e-03  0.957138
    128      1000  1.929043e-16  4.684239e+00  2.342119e-03  1.912797
    256      1000  2.092319e-16  5.313004e+00  2.656502e-03  3.854693
    512      100  1.927488e-16  6.197160e-01  3.098580e-03  7.435664
   1024      100  2.308607e-16  6.999220e-01  3.499610e-03  14.630202
   2048      100  2.447624e-16  8.121880e-01  4.060940e-03  27.737420
   4096      100  2.479782e-16  8.895550e-01  4.447775e-03  55.254594
   8192      10  2.578088e-16  1.127540e-01  5.637700e-03  94.449864
  16384      10  2.733986e-16  1.516320e-01  7.581600e-03  151.271499
  32768      10  2.923012e-16  2.008800e-01  1.004400e-02  244.683393
  65536      10  2.829927e-16  3.405690e-01  1.702845e-02  307.889444
 131072      1  3.149670e-16  5.555500e-02  2.777750e-02  401.084331
 262144      1  3.218597e-16  1.060310e-01  5.301550e-02  445.020041
 524288      1  3.281373e-16  2.250950e-01  1.125475e-01  442.545236
1048576      1  3.285898e-16  4.961540e-01  2.480770e-01  422.681667

FFT_SERIAL:
Normal end of execution.
29 November 2021 04:44:59 AM

```

对比并行的运行结果：


```

FFT ( FFT ( X(1:N) ) ) == N * X(1:N)

      N      NITS      Error      Time      Time/Call      MFLOPS
step cost time: usec --- 421
step cost time: usec --- 233
      2      10000  7.859082e-17step cost time: usec --- 524
step cost time: usec --- 276
step cost time: usec --- 259
step cost time: usec --- 251
step cost time: usec --- 275
step cost time: usec --- 459
step cost time: usec --- 301
step cost time: usec --- 294
step cost time: usec --- 401
step cost time: usec --- 298
step cost time: usec --- 263
step cost time: usec --- 386
step cost time: usec --- 317
step cost time: usec --- 324
step cost time: usec --- 473
step cost time: usec --- 291
step cost time: usec --- 258
step cost time: usec --- 323
step cost time: usec --- 314
step cost time: usec --- 384
step cost time: usec --- 1006
step cost time: usec --- 812
step cost time: usec --- 1035
step cost time: usec --- 2756
step cost time: usec --- 430
step cost time: usec --- 312
step cost time: usec --- 362
step cost time: usec --- 480
step cost time: usec --- 425
step cost time: usec --- 615
step cost time: usec --- 371
step cost time: usec --- 410
step cost time: usec --- 379
step cost time: usec --- 317
step cost time: usec --- 365
step cost time: usec --- 316
step cost time: usec --- 285
step cost time: usec --- 377
step cost time: usec --- 286

```

结果都是几百微秒，非常慢。

- 然后计算并输出，pthread创建线程和释放空间所花费的时间（在介绍函数 `parallel_for()` 的时候解释过）：

```

FFT ( FFT ( X(1:N) ) ) == N * X(1:N)

      N      NITS      Error      Time      Time/Call      MFLOPS
creat pthread cost time :usec --- 405
creat pthread cost time :usec --- 348
      2      10000  7.859082e-17creat pthread cost time :usec --- 364
creat pthread cost time :usec --- 259
creat pthread cost time :usec --- 278
creat pthread cost time :usec --- 343
creat pthread cost time :usec --- 255
creat pthread cost time :usec --- 209
creat pthread cost time :usec --- 224
creat pthread cost time :usec --- 189
creat pthread cost time :usec --- 230
creat pthread cost time :usec --- 238
creat pthread cost time :usec --- 196
creat pthread cost time :usec --- 240
creat pthread cost time :usec --- 285
creat pthread cost time :usec --- 201
creat pthread cost time :usec --- 305
creat pthread cost time :usec --- 248
creat pthread cost time :usec --- 235
creat pthread cost time :usec --- 298
creat pthread cost time :usec --- 245
creat pthread cost time :usec --- 209
creat pthread cost time :usec --- 301
creat pthread cost time :usec --- 279
creat pthread cost time :usec --- 255
creat pthread cost time :usec --- 199
creat pthread cost time :usec --- 225
creat pthread cost time :usec --- 379
creat pthread cost time :usec --- 906
creat pthread cost time :usec --- 811
creat pthread cost time :usec --- 651
creat pthread cost time :usec --- 287
creat pthread cost time :usec --- 518
creat pthread cost time :usec --- 285
creat pthread cost time :usec --- 407
creat pthread cost time :usec --- 291
creat pthread cost time :usec --- 266
creat pthread cost time :usec --- 312
creat pthread cost time :usec --- 830
creat pthread cost time :usec --- 219

```

可以看出pthread创建线程和释放空间所花费的时间非常大。

- 结论

虽然我们输出的每次调用step()花费的时间和每次调用step后创建线程和释放空间花费的时间并不是在同一次运算时输出的。但是并行运算的结果较稳定，并且每调用一次step函数就会相应调用一次 `parallel_for()` 函数来创建销毁线程，因此我们仍可以比较得到一定的结果。可以看出实际上两者的花费时间相差不多都是几百甚至上千微秒，那么就说明运算中的很大一部分时间都是用在

了pthread创建线程和释放空间上。最后综合以上的情况可以看出，创建线程和销毁线程的开销大也许就是并行效果差的原因。

2) 更改函数cfft2

由于并行 step() 函数的效果较差，因此尝试并行 cfft2() 函数，由于 cfft2() 函数有数据依赖，因此在更改的时候需要注意。

代码

完整代码见lab5_1_2.c

- cfft2():
 - 在原来的 cfft2() 函数上进行更改，主要是将里面的for循环替换成下面的17-27行：
 - 第17-23行是将函数 parallel_for() 中需要用到的变量写入一个结构体中，曾作为参数传给函数 parallel_for() 进行并行；
 - 第25行，调用函数 parallel_for() 将原来的for循环进行并行化；
 - 由于原来的for循环中每迭代一次就给 mj 乘以2，为了不影响接下来的运算，那么当进行了 m-2 次循环后，mj应该乘 m-2 次2，即 2^{m-2} 。
 - 第29行的判断条件是一个变量tgle，这个变量是依赖for循环每一次的更改的，但是它实际上很有规律，当for循环里的j是双数的时候这次tgle为1然后变为0，单数的时候相反。因此可以判断结束循环后的tgle实际上由m-2的单双数决定。

```
1 void cfft2 ( int n, double x[], double y[], double w[], double sgn )
2 {
3     int j;
4     int m;
5     int mj;
6
7     m = ( int ) ( log ( ( double ) n ) / log ( 1.99 ) );
8     mj = 1;
9
10    step ( n, mj, &x[0*2+0], &x[(n/2)*2+0], &y[0*2+0], &y[mj*2+0], w, sgn
11    );
12
13    if ( n == 2 )
14    {
15        return;
16    }
17
18    struct for_index first;
19    first.n = n;
20    first.mj = mj;
21    first.x = x;
22    first.y = y;
23    first.w = w;
24    first.sgn = sgn;
25
26    parallel_for(0, m-2, 1, cfft2_parallel,&first, num_thread);
27
28    mj = mj * pow(2,m-2);
29
30    if ( (m-2)%2 == 0 )
31    {
```



```

31     ccopy ( n, y, x );
32 }
33
34 mj = n / 2;
35 step ( n, mj, &x[0*2+0], &x[(n/2)*2+0], &y[0*2+0], &y[mj*2+0], w, sgn
);
36
37 return;
38 }

```

- **parallel_for()**

和step()中调用的parallel_for()基本相同，只是这里将a、b、c、d的赋值更改为x、y的赋值。

```

1 void parallel_for_cfft2(int start, int end, int increment, void *
  (*functor)(void*), void *arg , int num_threads){
2     unsigned long time=0;
3
4     struct for_index *first;
5     first = (struct for_index*)arg;
6     pthread_t pth[num_threads];
7     int divid = (end - start )/num_threads;
8     struct for_index thread_assign[num_threads];
9
10    for(int i=0; i<num_threads; ++i){
11        thread_assign[i].start = i*divid ;
12        if(i == (num_threads - 1) ){
13            thread_assign[i].end = end;
14        }
15        else{
16            thread_assign[i].end = thread_assign[i].start + divid;
17        }
18        thread_assign[i].increment = increment;
19        thread_assign[i].n = first->n;
20        thread_assign[i].mj = first->mj;
21        thread_assign[i].x = first->x;
22        thread_assign[i].y = first->y;
23        thread_assign[i].w = first->w;
24        thread_assign[i].sgn =first->sgn;
25    }
26
27    // struct timeval val;
28    // struct timeval newVal;
29    // int ret = gettimeofday(&val, NULL);
30
31    for(int i=0; i<num_threads; ++i){
32        pthread_create(&pth[i], NULL, functor, &thread_assign[i]);
33    }
34
35    for(int i=0; i<num_threads; ++i){
36        // time += thread_assign[i].time;
37        pthread_join(pth[i], NULL);
38    }
39
40    // ret = gettimeofday(&newVal, NULL);
41    // unsigned long diff = (newVal.tv_sec * Converter + newVal.tv_usec)
  - (val.tv_sec * Converter + val.tv_usec);

```

```

42     // printf("diff:  sec --- %ld, usec --- %ld\n", diff / Converter,
    diff % Converter - time);
43
44 }

```

• cfft2()

cfft2()的串行版本的for函数中有两个数据依赖:

- mj, 每进行一次循环mj是上一次循环里mj的2倍, 在并行中每个线程的起始mj可以按照start来计算, start表示该线程从第几个循环开始计算, 那么在该线程中 $m_j = m_j * 2^{start}$, 然后再在该线程中的每个循环中将mj乘以2倍。
- for循环中有两个选择, 在刚开始的时候有一个标志 tgle 初始值设为1, 当tgle为1的时候执行if里的语句, 为0时执行else里的语句, 每执行一次tgle就变一次, 即原来为0变为1原来为1变为0, 也就是两个语句轮流执行。每个循环都要依赖上一次循环中tgle的值, 但其实这个算法也可以看作j为双数时执行if里的语句, j为单数时执行else里的语句, 这样就不再有数据依赖可以进行并行了。

解决了两个数据依赖其他部分和原来串行时的代码一样。最后, 作为参数传过来的结构体里有需要的变量, 在并行的时候直接使用就行。

```

1 void *cfft2_parallel(void *arg){
2     // struct timeval val;
3     // struct timeval newVal;
4     // int ret = gettimeofday(&val, NULL);
5
6     struct for_index *thread_assign;
7     thread_assign = (struct for_index*)arg;
8     int mj = thread_assign->mj * pow(2,thread_assign->start);
9
10    for (int j = thread_assign->start; j < thread_assign->end;
    j=j+thread_assign->increment)
11    {
12        mj = mj * 2;
13        if(j%2 == 0){
14            step ( thread_assign->n, mj,thread_assign->y+(0*2+0),
    thread_assign->y+((thread_assign->n/2)*2+0), thread_assign->x+(0*2+0),
    thread_assign->x+(mj*2+0), thread_assign->w, thread_assign->sgn );
15        }
16        else{
17            step ( thread_assign->n, mj, thread_assign->x+(0*2+0),
    thread_assign->x+((thread_assign->n/2)*2+0), thread_assign->y+(0*2+0),
    thread_assign->y+(mj*2+0), thread_assign->w, thread_assign->sgn );
18        }
19    }
20
21    // ret = gettimeofday(&newVal, NULL);
22    // unsigned long diff = (newVal.tv_sec * Converter + newVal.tv_usec)
    - (val.tv_sec * Converter + val.tv_usec);
23    // thread_assign->time = diff ;
24 }

```

运行结果

并行后的运行时间：

```
wwwj@ubuntu:~/lab/lab5/fft_serial$ gcc -o lab5_1 lab5_1.c -lpthread -lm
wwwj@ubuntu:~/lab/lab5/fft_serial$ ./lab5_1
29 November 2021 05:44:49 PM

FFT_SERIAL
C version

Demonstrate an implementation of the Fast Fourier Transform
of a complex data vector.

Accuracy check:

    FFT ( FFT ( X(1:N) ) ) == N * X(1:N)

      N      NITS      Error      Time      Time/Call      MFLOPS
      2      10000  7.859082e-17  2.712000e-03  1.356000e-07  73.746313
      4      10000  1.209837e-16  7.547315e+00  3.773657e-04  0.105998
      8      10000  6.820795e-17  6.823406e+00  3.411703e-04  0.351730
     16      10000  1.438671e-16  7.110043e+00  3.555021e-04  0.900135
     32      1000   1.331210e-16  7.386870e-01  3.693435e-04  2.166005
     64      1000   1.776545e-16  8.420980e-01  4.210490e-04  4.560039
    128      1000   1.929043e-16  8.432290e-01  4.216145e-04  10.625821
    256      1000   2.092319e-16  8.653230e-01  4.326615e-04  23.667463
    512      100    1.927488e-16  1.059710e-01  5.298550e-04  43.483595
   1024      100    2.308607e-16  1.445320e-01  7.226600e-04  70.849362
   2048      100    2.447624e-16  1.994920e-01  9.974600e-04  112.926834
   4096      100    2.479782e-16  3.477130e-01  1.738565e-03  141.357959
   8192      10    2.578088e-16  9.150100e-02  4.575050e-03  116.387799
  16384      10    2.733986e-16  1.497250e-01  7.486250e-03  153.198197
  32768      10    2.923012e-16  2.638360e-01  1.319180e-02  186.297548
  65536      10    2.829927e-16  4.323210e-01  2.161605e-02  242.545701
 131072      1    3.149670e-16  6.813200e-02  3.406600e-02  327.045148
 262144      1    3.218597e-16  1.635320e-01  8.176600e-02  288.542426
 524288      1    3.281373e-16  4.562470e-01  2.281235e-01  218.335069
1048576      1    3.285898e-16  7.922810e-01  3.961405e-01  264.698005

FFT_SERIAL:
Normal end of execution.

29 November 2021 05:45:17 PM
```

串行版本的运行时间：

```
wwwj@ubuntu:~/lab/lab5/fft_serial$ ./fft_serial
29 November 2021 05:46:01 PM

FFT_SERIAL
C version

Demonstrate an implementation of the Fast Fourier Transform
of a complex data vector.

Accuracy check:

    FFT ( FFT ( X(1:N) ) ) == N * X(1:N)

      N      NITS      Error      Time      Time/Call      MFLOPS
      2      10000  7.859082e-17  3.708000e-03  1.854000e-07  53.937433
      4      10000  1.209837e-16  8.382000e-03  4.191000e-07  95.442615
      8      10000  6.820795e-17  1.338200e-02  6.691000e-07  179.345389
     16      10000  1.438671e-16  3.138000e-02  1.569000e-06  203.951562
     32      1000   1.331210e-16  5.752000e-03  2.876000e-06  278.164117
     64      1000   1.776545e-16  1.690100e-02  8.450500e-06  227.205491
    128      1000   1.929043e-16  3.745700e-02  1.872850e-05  239.207625
    256      1000   2.092319e-16  8.123500e-02  4.061750e-05  252.108081
    512      100    1.927488e-16  2.346800e-02  1.173400e-04  196.352480
   1024      100    2.308607e-16  5.447900e-02  2.723950e-04  187.962334
   2048      100    2.447624e-16  8.724000e-02  4.362000e-04  258.230170
   4096      100    2.479782e-16  1.976940e-01  9.884700e-04  248.626665
   8192      10    2.578088e-16  5.938700e-02  2.969350e-03  179.325442
  16384      10    2.733986e-16  1.097460e-01  5.487300e-03  209.006251
  32768      10    2.923012e-16  1.710960e-01  8.554800e-03  287.277318
  65536      10    2.829927e-16  4.088870e-01  2.044435e-02  256.446402
 131072      1    3.149670e-16  1.183080e-01  5.915400e-02  188.340941
 262144      1    3.218597e-16  1.997920e-01  9.989600e-02  236.175222
 524288      1    3.281373e-16  6.528640e-01  3.264320e-01  152.581119
1048576      1    3.285898e-16  8.306920e-01  4.153460e-01  252.458432

FFT_SERIAL:
Normal end of execution.

29 November 2021 05:46:07 PM
```

可以看到最后并行在N=2以及N>65536的时候并行的效果要比串行的效果好，但仍不明显，基于并行step的经验，线程创建和销毁的开销占据很大一部分原因。

但是并行的线程增大后会发现Error改变，说明改变cfft2的算法还是有问题，反复查验后发现实际上在 `cfft2_parallel()` 并行的for循环中不仅仅有哪两处数据依赖，最大数据依赖的问题是 $x+(0*2+0)$ 和 $y+(0*2+0)$ 指向的两个数据会在每次循环后对下一次的数据产生影响，但是这里我并没有找到很好的解决方法，因此这部分的并行化是有错误的。

任务2: (二选一选做二)

1. 将fft_serial应用改造成基于MPI的进程并行应用（为了适合MPI的消息机制，可能需要对fft_serial的代码实现做一定调整）。Bonus:使用MPI_Pack/MPI_Unpack, 或MPI_Type_create_struct实现数据重组后的消息传递。
2. 将heated_plate_openmp应用改造成基于MPI的进程并行应用。Bonus:使用MPI_Pack/MPI_Unpack, 或MPI_Type_create_struct实现数据重组后的消息传递。

代码

完整代码见lab5_2.c

使用MPI进行并行化，主要在while循环中进行了更改，所有的代码都是在主函数中实现的：

- 第13-26行，定义一些需要在接下来用到的变量。

```
1  # define M 500
2  # define N 500
3
4  double diff;
5  double epsilon = 0.001;
6  int i;
7  int iterations;
8  int iterations_print;
9  int j;
10 double mean;
11 double my_diff;
12 double u[M][N];
13 double w[M][N];
14 double wtime;
```

- 首先需要进行MPI的初始化，获取总的进程数以及正在运行的进程号：

```
1  MPI_Init(NULL, NULL);
2  int numprocess, rank;
3  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4  MPI_Comm_size(MPI_COMM_WORLD, &numprocess);
```

- 输出信息以及初始化w并计算mean。这一部分没必要所有的进程都执行，会造成不必要的开销，因此只在主进程即rank=0的进程中执行，之后将其他进程需要的变量利用MPI的通信进行传输。并且在这部分内容中并不改变原本用openmp实现的线程并行：

```
1  if(rank == 0){
2      printf ( "\n" );
3      printf ( "HEATED_PLATE_OPENMP\n" );
4      printf ( "  C/OpenMP version\n" );
5      printf ( "  A program to solve for the steady state temperature
distribution\n" );
6      printf ( "  over a rectangular plate.\n" );
7      printf ( "\n" );
8      printf ( "  Spatial grid of %d by %d points.\n", M, N );
9      printf ( "  The iteration will be repeated until the change is <=
%e\n", epsilon );
10     printf ( "  Number of processors available = %d\n",
omp_get_num_procs ( ) );
```

```

11     printf ( "   Number of threads =           %d\n",
omp_get_max_threads ( ) );
12     /*
13         Set the boundary values, which don't change.
14     */
15     mean = 0.0;
16
17     #pragma omp parallel shared ( w ) private ( i, j )
18     {
19     #pragma omp for
20         for ( i = 1; i < M - 1; i++ )
21         {
22             w[i][0] = 100.0;
23         }
24     #pragma omp for
25         for ( i = 1; i < M - 1; i++ )
26         {
27             w[i][N-1] = 100.0;
28         }
29     #pragma omp for
30         for ( j = 0; j < N; j++ )
31         {
32             w[M-1][j] = 100.0;
33         }
34     #pragma omp for
35         for ( j = 0; j < N; j++ )
36         {
37             w[0][j] = 0.0;
38         }
39     /*
40         Average the boundary values, to come up with a reasonable
41         initial value for the interior.
42     */
43     #pragma omp for reduction ( + : mean )
44         for ( i = 1; i < M - 1; i++ )
45         {
46             mean = mean + w[i][0] + w[i][N-1];
47         }
48     #pragma omp for reduction ( + : mean )
49         for ( j = 0; j < N; j++ )
50         {
51             mean = mean + w[M-1][j] + w[0][j];
52         }
53     }
54     /*
55         OpenMP note:
56         You cannot normalize MEAN inside the parallel region. It
57         only gets its correct value once you leave the parallel region.
58         So we interrupt the parallel region, set MEAN, and go back in.
59     */
60     mean = mean / ( double ) ( 2 * M + 2 * N - 4 );
61     printf ( "\n" );
62     printf ( "   MEAN = %f\n", mean );
63     /*
64         Initialize the interior solution to the mean value.
65     */
66     #pragma omp parallel shared ( mean, w ) private ( i, j )
67     {

```

```

68     #pragma omp for
69         for ( i = 1; i < M - 1; i++ )
70         {
71             for ( j = 1; j < N - 1; j++ )
72             {
73                 w[i][j] = mean;
74             }
75         }
76     }
77     /*
78         iterate until the new solution w differs from the old solution u
79         by no more than EPSILON.
80     */
81     printf ( "\n" );
82     printf ( " Iteration Change\n" );
83     printf ( "\n" );
84
85
86     iterations = 0;
87     iterations_print = 1;
88
89     wtime = omp_get_wtime ( );
90 }

```

- 这部分所有进程都需要执行，主要进行：

- 第1-2行，给diff初始化，设置一个变量first用来判断是否是第一次进入while循环；
- 第4-13行，接下来要给while中的一个for循环进行MPI的并行化，这部分用来计算每个进程能分到多少个循环：一共要进行M-1-1次循环，由numprocess个进程，计算得到block；但是有可能循环并不能倍平均分配，因此前面几个线程都分配block个循环，最后将所有未被分配的循环都分配给最后一个线程，依据这个原则计算每个进程的start和end即循环开始和结束的定位；

```

1     diff = epsilon;
2     int first = 1;
3
4     int block = (M-1-1)/numprocess;
5     int start,end;
6     start = 1+rank*block;
7     if (rank==numprocess-1)
8     {
9         end =M-1;
10    }
11    else{
12        end = 1+(rank+1)*block;
13    }

```

- 进入while循环：

- 第6-13行，由于每次再进入循环后需要将原来w中的值对应保存在u中，于是利用每个进程中利用openmp将该进程负责的循环中的u进行赋值；
- 第15-45行，进行进程间的通信：
 - 第15-30行，如果是第一次进入while循环并且是进程0的话，首先将w赋值给u，然后利用MPI_Send将u和w传送给其他进程。起始地址是u、w，数量都是M*N，数据类型都是MPI_DOUBLE，目的进程是除0外的所有进程，tag分别为1、2，与传送的数据相对应，通信子为MPI_COMM_WORLD。

- 第32-36行，如果是第一次进入while循环并且不是进程0的话，就需要接收来自进程0的两个double类型的数据，这里的几个参数都比较简单，主要就是0表示消息来自进程0，tag为1、2与上面send中的tag相对应；
- 第38-45行，原本的for循环中有一点数据依赖就是需要用到前面一个循环的和后面一个循环的u，即每个进程都需要u[start-1]和u[end]，因此除了第一次进入while循环进行整体的消息传递外，每个进程还需要将自己的u[start]传递给上一个进程，作为上一个进程的u[end]；以及需要将自己的u[end-1]传递给下一个进程，作为下一个进程的u[start-1]。但是rank0不需要其他进程传递给他u[start-1]（u[0]固定不变），也不需要传递给其他进程他的u[start]；同样特殊的还有最后一个进程，他不需要其他进程传递给他u[end]，也不需要传递给其他进程他的u[end-1]。因此判断条件：当rank>0的时候，每个进程将自己的u[start]传递给上一个进程，作为上一个进程的u[end]；当rank<numprocess-1的时候，每个进程将自己的u[end-1]传递给下一个进程，作为下一个进程的u[start-1]。

这里需要注意的是，u是一个二维数组，每次传递的数据个数是N，并不是1，刚开始我没注意到这里最后的结果就是错的。

- 第48-55行，每个进程负责部分循环，根据u计算w（w并没有数据依赖，不需要进行消息传递），这部分也可以用openmp进行线程并行，不会产生数据竞争。
- 第58-79行，找到所有进程最大的diff：
 - 第58-58行，进行两个变量的初始化；
 - 第61-70行，每个进程比较得到各自负责的部分中最大的w[i][j]和u[i][j]之间的差异为diff；
 - 第72-73行，利用一个集合通信函数MPI_Allgather()，将所有进程的diff传递给每个进程，每个进程得到一个double类型的数组all_diff[numprocess]；
 - 第75-79行，比较得到所有的进程中最大的diff；
- 第81-88行，由进程0执行，输出迭代到第多少次（每 2^i 次迭代输出一次）和这次迭代过后得到的diff。

```

1  while ( epsilon <= diff )
2  {
3      /*
4       Save the old solution in u.
5      */
6      # pragma omp parallel for private ( i, j ) shared ( u, w )
7      for ( i = start; i < end; i++ )
8      {
9          for ( j = 0; j < N; j++ )
10         {
11             u[i][j] = w[i][j]; //u[i-1],u[i+1] impact
12         }
13     }
14
15     if(first == 1 && rank == 0){
16         first = 0;
17         # pragma omp parallel for private ( i, j ) shared ( u, w )
18         for ( i = 0; i < M; i++ )
19         {
20             for ( j = 0; j < N; j++ )
21             {
22                 u[i][j] = w[i][j]; //u[i-1],u[i+1] impact
23             }
24         }
25     }

```

```

26     for(int i=1;i<numprocess;++i){
27         MPI_Send(u,M*N,MPI_DOUBLE,i,1,MPI_COMM_WORLD);
28         MPI_Send(w,M*N,MPI_DOUBLE,i,2,MPI_COMM_WORLD);
29     }
30 }
31
32 else if (first == 1 && rank != 0) {
33     first = 0;
34     MPI_Recv(u,M*N,MPI_DOUBLE,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
35     MPI_Recv(w,M*N,MPI_DOUBLE,0,2,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
36 }
37
38 else{
39     if(rank>0){
40         MPI_Recv(u[start-1],N,MPI_DOUBLE,rank-
1,3,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
41         MPI_Send(u[start],N,MPI_DOUBLE,rank-1,4,MPI_COMM_WORLD);}
42         if(rank<numprocess-1){
43             MPI_Send(u[end-1],N,MPI_DOUBLE,rank+1,3,MPI_COMM_WORLD);
44
45             MPI_Recv(u[end],N,MPI_DOUBLE,rank+1,4,MPI_COMM_WORLD,MPI_STATUS_IGNORE)
46             ;}
47         }
48
49         # pragma omp parallel for private ( i, j ) shared ( u, w )
50         for ( i = start; i < end; i++ )
51         {
52             for ( j = 1; j < N - 1; j++ )
53             {
54                 w[i][j] = ( u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] ) /
55                 4.0;
56             }
57         }
58
59         diff = 0.0;
60         my_diff = 0.0;
61
62         for ( i = start; i < end; i++ )
63         {
64             for ( j = 1; j < N - 1; j++ )
65             {
66                 if ( my_diff < fabs ( w[i][j] - u[i][j] ) )
67                 {
68                     my_diff = fabs ( w[i][j] - u[i][j] );
69                 }
70             }
71         }
72
73         double all_diff[numprocess];
74
75         MPI_Allgather(&my_diff,1,MPI_DOUBLE,all_diff,1,MPI_DOUBLE,MPI_COMM_WORL
76         D);
77
78         for(int i=0;i<numprocess;++i){
79             if(diff < all_diff[i]){
80                 diff = all_diff[i];

```



```

78     }
79 }
80
81 if(rank == 0){
82     iterations++;
83     if ( iterations == iterations_print )
84     {
85         printf ( " %8d %f\n", iterations, diff );
86         iterations_print = 2 * iterations_print;
87     }
88 }
89
90 }

```

- 最后在diff<=0.001的时候跳出while循环，并由进程0计算运行的时间，并输出迭代了多少次以及花费的时间，最后清除MPI的所有状态：

```

1  if(rank == 0){
2      wtime = omp_get_wtime ( ) - wtime;
3      printf ( "\n" );
4      printf ( " %8d %f\n", iterations, diff );
5      printf ( "\n" );
6      printf ( " Error tolerance achieved.\n" );
7      printf ( " wallclock time = %f\n", wtime );
8      /*
9          Terminate.
10     */
11     printf ( "\n" );
12     printf ( "HEATED_PLATE_OPENMP:\n" );
13     printf ( " Normal end of execution.\n" );
14 }
15
16 MPI_Finalize();
17
18 return 0;
19
20 # undef M
21 # undef N
22 }

```

运行结果

对比原来只使用openmp进行并行的结果：

```

wwwj@ubuntu:~/lab/lab5$ ./heated_plate_openmp
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 = 4
Number of threads = 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 = 53.843794
HEATED_PLATE_OPENMP:
Normal end of execution.

```

最后的运行结果并不稳定，运行很多次时间可能在100s以内也有可能要100多s，下面是选取的两次效果较好的运行结果：

```

wwwj@ubuntu:~/lab/lab5$ mplexec -np 2 ./lab5_2
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 = 4
Number of threads = 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 = 57.337591
HEATED_PLATE_OPENMP:
Normal end of execution.

```

```

wwwj@ubuntu:~/Lab/Lab5$ mpiexec -np 1 ./lab5_2
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 = 4
Number of threads = 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 = 75.722362
HEATED_PLATE_OPENMP:
Normal end of execution.

wwwj@ubuntu:~/Lab/Lab5$ mpicc -o lab5_2 lab5_2.c -fopenmp
wwwj@ubuntu:~/Lab/Lab5$ ./lab5_2
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 = 4
Number of threads = 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 = 26.343478
HEATED_PLATE_OPENMP:
Normal end of execution.

```

首先看运行结果，在diff为0.001的时候迭代次数为16955，使用mpi后结果一样，所以算法并没有问题。但是从运行时间上看，可以看到虽然可能运行时间相差不大，但是使用mpi和openmp混合并行的效果没有只使用openmp的并行效果好，MPI的通信可能是花费了很多时间。

使用MPI_Pack/MPI_Unpack

完整代码见lab5_2_1.c

代码：

- 第10-11行，将两个变量进行打包，解释函数 `MPI_Pack()` 的参数：
 - u和w都是待打包数据的指针；
 - M*N是打包数据元素个数；
 - MPI_DOUBLE是打包数据的数据类型；
 - buffer是一个指向打包输出缓冲区的指针；

- $2*M*N*8$ 缓冲区大小（单位为 Byte）,这里需要注意的是这个单位为Byte指的是缓冲区的大小，而不是指缓冲区的存储的数据个数；一个double类型的数据是8字节64位，因此这里在buffer的数量 $2*M*N$ 后面再乘以8；
 - &position，用来输出缓冲区中第一个用于打包的位置（地址偏移量）；
 - MPI_COMM_WORLD是通信子
- 第13-15行，将打包成buffer的数据利用MPI_Send()传送到除进程0外的其他进程上去；
- 第20行，除进程0外的其他进程接收打包后的数据；
- 第21-22行，将buffer里的数据拿出来，里面的参数分别表示：指向待解包缓冲区的指针、缓冲区大小（单位为 Byte）、输出缓冲区中第一个用于打包的位置（地址偏移量）、指向解包后数据的指针、解包元素个数、数据类型、通信子；

```

1  while( epsilon <= diff ){
2      ...
3      int position;
4      double buffer[2*M*N];
5
6      if(first == 1 && rank==0){
7          first = 0;
8          ...
9
10         MPI_Pack(u, M*N, MPI_DOUBLE, buffer, 2*M*N*8, &position,
MPI_COMM_WORLD);
11         MPI_Pack(w, M*N, MPI_DOUBLE, buffer, 2*M*N*8, &position,
MPI_COMM_WORLD);
12
13         for(int i=1;i<numprocess;++i){
14             MPI_Send(buffer,2*M*N,MPI_DOUBLE,i,1,MPI_COMM_WORLD);
15         }
16     }
17
18     else if (first == 1 && rank!=0) {
19         first = 0;
20
21         MPI_Recv(buffer,2*M*N,MPI_DOUBLE,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
22         MPI_Unpack(buffer, 2*M*N, &position, u, M*N, MPI_DOUBLE,
MPI_COMM_WORLD);
23         MPI_Unpack(buffer, 2*M*N, &position, w, M*N, MPI_DOUBLE,
MPI_COMM_WORLD);
24     }
25     else{
26         ...
27     }
28     ...
29 }

```

验证结果：

与并行的计算结果一致，说明使用MPI_Pack/MPI_Unpack后的改编是正确的。

```

wwwj@ubuntu:~/lab/lab5$ mpiexec -np 2 ./3
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 = 4
Number of threads = 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 = 65.448750

HEATED_PLATE_OPENMP:
Normal end of execution.

```

使用MPI_Type_create_struct

完整代码见lab5_2_1.c

代码

- 创建一个结构体：
 - 第3-6行，定义一个结构体，里面用来存放即将传向各个进程的 `u`，以及 `w`。他们的大小已经确定，由于MPI函数中传递的元素的起始地址之后的元素被要求是连续的，因此在这里一定要确定他们的大小。
 - 第8行，是新数据类型中元素个数，这里是2（double, double）。
 - 第9行，是每个数据项的元素个数（`u`、`w`里都有`M*N`个）。
 - 第10行，是每个数据项距离消息起始位置的偏移量（`u`的偏移量是0，`w`的偏移量是`8*M*N`字节）
 - 第11行，是每个数据项的MPI类型。
 - 第12行，是一个自定义数据类型出口，命名为 `mytype`。
 - 第14行，函数 `MPI_Type_create_struct()`，它的每个参数前面几行都有介绍。
 - 第15行，函数 `MPI_Type_commit()` 表示允许MPI实现在通信函数内使用这一数据类型。

```

1  while( epsilon <= diff ){
2      ...
3      struct var{
4          double u[M][N];
5          double w[M][N];
6      };
7
8      int var_count=2;
9      int var_everycount[2]={M*N, M*N};
10     MPI_Aint var_displace[2]={0, 8*M*N};
11     MPI_Datatype var_type[2]={MPI_DOUBLE, MPI_DOUBLE};
12     MPI_Datatype mytype;
13
14     MPI_Type_create_struct(var_count, var_everycount, var_displace,
var_type, &mytype);

```

```

15     MPI_Type_commit(&mytype);
16     ...
17 }

```

- 然后利用自己创建的数据类型进行消息传递：
 - 第6-12行，将要传递的数据写入定义的结构体中；
 - 第14-16行，将写好的结构体作为自己创建的数据类型传递到除0外的其他进程中去；
 - 第21行，除0外的其他进程接收这个结构体；
 - 第22-27行，将收到的结构体的数据写入进程中定义的u、w中。

```

1  while( epsilon <= diff ){
2      ...
3      if(first == 1 && rank==0){
4          first = 0;
5          ...
6          struct var temp;
7          for(int i=0;i<M;++i){
8              for(int j=0;j<N;++j){
9                  temp.u[i][j]=u[i][j];
10                 temp.w[i][j]=w[i][j];
11             }
12         }
13
14         for(int i=1;i<numprocess;++i){
15             MPI_Send(&temp,1,mytype,i,1,MPI_COMM_WORLD);
16         }
17     }
18     else if (first == 1 && rank!=0) {
19         first = 0;
20         struct var temp;
21         MPI_Recv(&temp,1,mytype,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
22         for(int i=0;i<M;++i){
23             for(int j=0;j<N;++j){
24                 u[i][j]=temp.u[i][j];
25                 w[i][j]=temp.w[i][j];
26             }
27         }
28     }
29     else{
30         ...
31     }
32     ...
33 }

```

运行结果：

与并行的计算结果一致，说明使用MPI_Type_create_struct后的改编是正确的。

```

wwwj@ubuntu:~/lab/lab5$ mpicc -o 3 3.c -fopenmp
wwwj@ubuntu:~/lab/lab5$ mpiexec -np 2 ./3

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 = 4
Number of threads = 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
  32768   0.000521
  65536   0.000261
 129072   0.000130
 258144   0.000065
 516288   0.000032
1032576   0.000016
2065152   0.000008
4130304   0.000004
8260608   0.000002
16521216  0.000001
33042432  0.000000
66084864  0.000000
132169728 0.000000
264339456 0.000000
528678912 0.000000
1057357824 0.000000
2114715648 0.000000
4229431296 0.000000
8458862592 0.000000
16917725184 0.000000
33835450368 0.000000
67670900736 0.000000
135341801472 0.000000
270683602944 0.000000
541367205888 0.000000
1082734411776 0.000000
2165468823552 0.000000
4330937647104 0.000000
8661875294208 0.000000
17323750588416 0.000000
34647501176832 0.000000
69295002353664 0.000000
138590004707328 0.000000
277180009414656 0.000000
554360018829312 0.000000
1108720037658624 0.000000
2217440075317248 0.000000
4434880150634496 0.000000
8869760301268992 0.000000
17739520602537984 0.000000
35479041205075968 0.000000
70958082410151936 0.000000
141916164820303872 0.000000
283832329640607744 0.000000
567664659281215488 0.000000
1135329318562430976 0.000000
2270658637124861952 0.000000
4541317274249723904 0.000000
9082634548499447808 0.000000
18165269096998895616 0.000000
36330538193997791232 0.000000
72661076387995582464 0.000000
145322152775991164928 0.000000
290644305551982329856 0.000000
581288611103964659712 0.000000
1162577222207929319424 0.000000
2325154444415858638848 0.000000
4650308888831717277696 0.000000
9300617777663434555392 0.000000
18601235555326869110784 0.000000
37202471110653738221568 0.000000
74404942221307476443136 0.000000
148809884442614952886272 0.000000
297619768885229905772544 0.000000
595239537770459811545088 0.000000
1190479075540919623090176 0.000000
2380958151081839246180352 0.000000
4761916302163678492360704 0.000000
9523832604327356984721408 0.000000
19047665208654713969442816 0.000000
38095330417309427938885632 0.000000
76190660834618855877771264 0.000000
152381321669237711755542528 0.000000
304762643338475423511085056 0.000000
609525286676950847022170112 0.000000
1219050573353901694044340224 0.000000
2438101146707803388088680448 0.000000
4876202293415606776177360896 0.000000
9752404586831213552354721792 0.000000
19504809173662427104709443584 0.000000
39009618347324854209418887168 0.000000
78019236694649708418837774336 0.000000
156038473389299416837675548672 0.000000
312076946778598833675351097344 0.000000
624153893557197667350702194688 0.000000
1248307787114395334701404389376 0.000000
2496615574228790669402808778752 0.000000
4993231148457581338805617557504 0.000000
9986462296915162677611235115008 0.000000
19972924593830325355222470230016 0.000000
39945849187660650710444940460032 0.000000
79891698375321301420889880920064 0.000000
159783396750642602841779761840128 0.000000
319566793501285205683559523680256 0.000000
639133587002570411367119047360512 0.000000
1278267174005140822734238094721024 0.000000
2556534348010281645468476189442048 0.000000
5113068696020563290936952378884096 0.000000
10226137392041126581873904757768192 0.000000
20452274784082253163747809515536384 0.000000
40904549568164506327495619031072768 0.000000
81809099136329012654991238062145536 0.000000
163618198272658025309982476124291072 0.000000
327236396545316050619964952248582144 0.000000
654472793090632101239929904497164288 0.000000
1308945586181264202479859808994328576 0.000000
2617891172362528404959719617988657152 0.000000
5235782344725056809919439235977314304 0.000000
10471564689450113619838878471954628608 0.000000
20943129378900227239677756943909257216 0.000000
41886258757800454479355513887818514432 0.000000
83772517515600908958711027775637028864 0.000000
167545035031201817917422055551274057728 0.000000
335090070062403635834844111102548115456 0.000000
670180140124807271669688222205096230912 0.000000
1340360280249614543339376444410192461824 0.000000
2680720560499229086678752888820384923648 0.000000
5361441120998458173357505777640769847296 0.000000
10722882241996916346715011555281539694592 0.000000
21445764483993832693430023110563079389184 0.000000
42891528967987665386860046221126158778368 0.000000
85783057935975330773720092442252317556736 0.000000
171566115871950661547440184884504635113472 0.000000
343132231743901323094880369769009270226944 0.000000
686264463487802646189760739538018540453888 0.000000
1372528926975605292379521479076037080907776 0.000000
2745057853951210584759042958152074161815552 0.000000
5490115707902421169518085916304148323631104 0.000000
10980231415804842339036171832608296647262208 0.000000
21960462831609684678072343665216593294524416 0.000000
43920925663219369356144687330433186589048832 0.000000
87841851326438738712289374660866373178097664 0.000000
175683702652877477424578749321732746356195328 0.000000
351367405305754954849157498643465492712390656 0.000000
702734810611509909698314997286930985424781312 0.000000
1405469621223019819396629994573861970849562624 0.000000
2810939242446039638793259989147723941699125248 0.000000
5621878484892079277586519978295447883398250496 0.000000
11243756969784158555173039956590895766796500992 0.000000
22487513939568317110346079913181791533593001984 0.000000
44975027879136634220692159826363583067186003968 0.000000
89950055758273268441384319652727166134372007936 0.000000
179900111516546536882768639305454332268744015872 0.000000
359800223033093073765537278610908664537488031744 0.000000
719600446066186147531074557221817329074976063488 0.000000
1439200892132372295062149114443634658149952126976 0.000000
2878401784264744590124298228887269316299904253952 0.000000
5756803568529489180248596457774538632599808507904 0.000000
11513607137058978360497192915549077265199617015808 0.000000
23027214274117956720994385831098154530399234031616 0.000000
46054428548235913441988771662196309060798468063232 0.000000
92108857096471826883977543324392618121596936126464 0.000000
184217714192943653767955086648785236243193872252928 0.000000
368435428385887307535910173297570472486387744505856 0.000000
736870856771774615071820346595140944972775489011712 0.000000
1473741713543549230143640693190281889945550978023424 0.000000
2947483427087098460287281386380563779891101956046848 0.000000
5894966854174196920574562772761127559782203912093696 0.000000
11789933708348393841149125545522255119564407824187392 0.000000
23579867416696787682298251091044510239128815648374784 0.000000
47159734833393575364596502182089020478257631296749568 0.000000
94319469666787150729193004364178040956515262593499136 0.000000
188638939333574301458386008728356081913030525186998272 0.000000
377277878667148602916772017456712163826061050373996544 0.000000
754555757334297205833544034913424327652122100747993088 0.000000
1509111514668594411667088069826848655304244201495986176 0.000000
3018223029337188823334176139653697310608488402991972352 0.000000
6036446058674377646668352279307394621216976805983944704 0.000000
12072892117348755293336704558614789242433953611967889408 0.000000
24145784234697510586673409117229578484867907223935778816 0.000000
48291568469395021173346818234459156969735814447871557632 0.000000
96583136938790042346693636468918313939471628895743115264 0.000000
193166273877580084693387272937836627878943257791486230528 0.000000
386332547755160169386774545875673255757886515582972461056 0.000000
772665095510320338773549091751346511515773031165944922112 0.000000
1545330191020640677547098183502693023031546062331889844224 0.000000
3090660382041281355094196367005386046063092124663779688448 0.000000
6181320764082562710188392734010772092126184249327559376896 0.000000
12362641528165125420376785468021544184252368498655118753792 0.000000
24725283056330250840753570936043088368504736997310237507584 0.000000
49450566112660501681507141872086176737009473994620475015168 0.000000
98901132225321003363014283744172353474018947989240950030336 0.000000
197802264450642006726028567488344706948037895978481900060672 0.000000
395604528901284013452057134976689413896075791956963800121344 0.000000
791209057802568026904114269953378827792151583913927600242688 0.000000
1582418115605136053808228539906757655584303167827855200485376 0.000000
3164836231210272107616457079813515311168606335655710400970752 0.000000
6329672462420544215232914159627030622337212671311420801941504 0.000000
12659344924841088430465828319254061244674425342622841603883008 0.000000
25318689849682176860931656638508122489348850685245683207766016 0.000000
50637379699364353721863313277016244978697701370491366415532032 0.000000
101274759398728707443726626554032489957395402740982732831064064 0.000000
202549518797457414887453253108064979914790805481965465662128128 0.000000
405099037594914829774906506216129959829581610963930931324256256 0.000000
810198075189829659549813012432259919659163221927861862648512512 0.000000
1620396150379659319099626024864519839318326443855723725297025024 0.000000
3240792300759318638199252049729039678636652887711447450594050048 0.000000
6481584601518637276398504099458079357273305775422894901188100096 0.000000
12963169203037274552797008198916158714546611550845789802376200192 0.000000
25926338406074549105594016397832317429093223101691579604752400384 0.000000
51852676812149098211188032795664634858186446203383159209504800768 0.000000
103705353624298196422376065591329269716372892406766318419009601536 0.000000
207410707248596392844752131182658539432745784813532636838019203072 0.000000
414821414497192785689504262365317078865491569627065273676038406144 0.000000
829642828994385571379008524730634157730983139254130547352076812288 0.000000
1659285657988771142758017049461268315461966278508261094704153624576 0.000000
3318571315977542285516034098922536630923932557016522189408307249152 0.000000
6637142631955084571032068197845073261847865114033044378816614498304 0.000000
13274285263910169142064136395690146523695730228066088757633228996608 0.000000
26548570527820338284128272791380293047391460456132177515266457993216 0.000000
53097141055640676568256545582760586094782920912264355030532915986432 0.000000
106194282111281353136513091165521172189565841824528710061065831972864 0.000000
212388564222562706273026182331042344379131683649057420122131663945728 0.000000
424777128445125412546052364662084688758263367298114840244263327891456 0.000000
849554256890250825092104729324169377516526734596229680488526655782912 0.000000
1699108513780501650184209458648338755033053469192459360977053311565824 0.000000
3398217027561003300368418917296677510066106938384918721954106623131648 0.000000
6796434055122006600736837834593355020132213876769837443908213246263296 0.000000
13592868110244013201473675669186710040264427753539674887816426492526592 0.000000
27185736220488026402947351338373420080528855507079349775632852985053184 0.000000
54371472440976052805894702676746840161057711014158699551265705970106368 0.000000
108742944881952105611789405353493680322115422028317399102531411940212736 0.000000
217485889763904211223578810706987360644230844056634798205062823880425472 0.000000
434971779527808422447157621413974721288461688113269596410125647760850944 0.000000
869943559055616844894315242827949442576923376226539192820251295521701888 0.000000
1739887118111233689788630485655898885153846752453078385640502591043403776 0.000000
3479774236222467379577260971311797770307693504906156771281005182086807552 0.000000
6959548472444934759154521942623595540615387009812313542562010364173615104 0.000000
13919096944889869518309043885247191081230774019624627085124020728347230208 0.000000
27838193889779739036618087770494382162461548039249254170248041456694460416 0.000000
55676387779559478073236175540988764324923096078498508340496082913388920832 0.000000
111352775559118956146472351081977528649846192156997016680992165826777841664 0.000000
222705551118237912292944702163955057299692384313994033361984331653555683328 0.000000
445411102236475824585889404327910114599384768627988066723968663307111366656 0.000000
890822204472951649171778808655820229198769537255976133447937326614222733312 0.000000
1781644408945903298343557617311640458397539074511952266895874653228445466624 0.000000
356328881789180659668711523462328091679507814902390453379174930
```

1)

线程n=1:

```
wwwj@ubuntu:~/lab/lab5/fft_serial$ gcc -o lab5_1_1 lab5_1_1.c -lpthread -lm
wwwj@ubuntu:~/lab/lab5/fft_serial$ ./lab5_1_1
num_thread: 1
30 November 2021 12:36:22 AM

FFT_SERIAL
C version

Demonstrate an implementation of the Fast Fourier Transform
of a complex data vector.

Accuracy check:

FFT ( FFT ( X(1:N) ) ) == N * X(1:N)

      N      NITS      Error      Time      Time/Call      MFLOPS
      2      10000  7.859082e-17  1.817713e+00  9.088565e-05  0.110028
      4      10000  1.209837e-16  2.582764e+00  1.291382e-04  0.309746
      8      10000  6.820795e-17  4.372007e+00  2.186004e-04  0.548947
     16      10000  1.438671e-16  5.177398e+00  2.588699e-04  1.236142
     32      1000  1.331210e-16  6.404590e-01  3.202295e-04  2.498208
     64      1000  1.776545e-16  7.918920e-01  3.959460e-04  4.849146
    128      1000  1.929043e-16  1.095433e+00  5.477165e-04  8.179414
    256      1000  2.092319e-16  1.053606e+00  5.268030e-04  19.438006
    512      100  1.927488e-16  1.228970e-01  6.144850e-04  37.494813
   1024      100  2.308607e-16  1.471860e-01  7.359300e-04  69.571834
   2048      100  2.447624e-16  1.832270e-01  9.161350e-04  122.951312
   4096      100  2.479782e-16  2.351610e-01  1.175805e-03  209.014250
   8192      10  2.578088e-16  3.551500e-02  1.775750e-03  299.862030
  16384      10  2.733986e-16  6.601600e-02  3.300800e-03  347.455162
  32768      10  2.923012e-16  1.133820e-01  5.669100e-03  433.507964
  65536      10  2.829927e-16  1.990150e-01  9.950750e-03  526.882898
 131072      1  3.149670e-16  3.724500e-02  1.862250e-02  598.261243
 262144      1  3.218597e-16  7.417500e-02  3.708750e-02  636.143175
 524288      1  3.281373e-16  1.500940e-01  7.504700e-02  663.682226
1048576      1  3.285898e-16  3.151070e-01  1.575535e-01  665.536469

FFT_SERIAL:
Normal end of execution.

30 November 2021 12:36:39 AM
```

线程n=2:

```
wwwj@ubuntu:~/lab/lab5/fft_serial$ gcc -o lab5_1_1 lab5_1_1.c -lpthread -lm
wwwj@ubuntu:~/lab/lab5/fft_serial$ ./lab5_1_1
num_thread: 2
30 November 2021 12:36:46 AM

FFT_SERIAL
C version

Demonstrate an implementation of the Fast Fourier Transform
of a complex data vector.

Accuracy check:

FFT ( FFT ( X(1:N) ) ) == N * X(1:N)

      N      NITS      Error      Time      Time/Call      MFLOPS
      2      10000  7.859082e-17  2.460982e+00  1.230491e-04  0.081268
      4      10000  1.209837e-16  4.899251e+00  2.449625e-04  0.163290
      8      10000  6.820795e-17  7.452708e+00  3.726354e-04  0.322031
     16      10000  1.438671e-16  9.895716e+00  4.947858e-04  0.646745
     32      1000  1.331210e-16  1.354988e+00  6.774940e-04  1.180822
     64      1000  1.776545e-16  1.697846e+00  8.489230e-04  2.261689
    128      1000  1.929043e-16  2.028997e+00  1.014499e-03  4.415975
    256      1000  2.092319e-16  2.297585e+00  1.148793e-03  8.913707
    512      100  1.927488e-16  2.734880e-01  1.367440e-03  16.849003
   1024      100  2.308607e-16  3.010240e-01  1.505120e-03  34.017221
   2048      100  2.447624e-16  3.520900e-01  1.760450e-03  63.983641
   4096      100  2.479782e-16  4.579170e-01  2.289585e-03  107.338229
   8192      10  2.578088e-16  5.770300e-02  2.885150e-03  184.558862
  16384      10  2.733986e-16  8.079100e-02  4.039550e-03  283.912812
  32768      10  2.923012e-16  1.343220e-01  6.716100e-03  365.926654
  65536      10  2.829927e-16  2.271420e-01  1.135710e-02  461.638975
 131072      1  3.149670e-16  4.767500e-02  2.383750e-02  467.377871
 262144      1  3.218597e-16  8.880400e-02  4.440200e-02  531.349038
 524288      1  3.281373e-16  1.935140e-01  9.675700e-02  514.767510
1048576      1  3.285898e-16  3.675360e-01  1.837680e-01  570.597710

FFT_SERIAL:
Normal end of execution.

30 November 2021 12:37:10 AM
```

线程n=4:


```

wwwj@ubuntu:~/lab/lab5/fft_serial$ gcc -o lab5_1_1 lab5_1_1.c -lpthread -lm
wwwj@ubuntu:~/lab/lab5/fft_serial$ ./lab5_1_1
num_thread: 4
30 November 2021 12:37:57 AM

FFT_SERIAL
C version

Demonstrate an implementation of the Fast Fourier Transform
of a complex data vector.

Accuracy check:

FFT ( FFT ( X(1:N) ) ) == N * X(1:N)

      N      NITS      Error      Time      Time/Call      MFLOPS
      2      10000  7.859082e-17  4.383312e+00  2.191656e-04  0.045628
      4      10000  1.209837e-16  9.071765e+00  4.535882e-04  0.088186
      8      10000  6.820795e-17  1.339384e+01  6.696920e-04  0.179187
     16      10000  1.438671e-16  2.086990e+01  1.043495e-03  0.306662
     32      1000  1.331210e-16  2.653894e+00  1.326947e-03  0.602888
     64      1000  1.776545e-16  3.165751e+00  1.582876e-03  1.212982
    128      1000  1.929043e-16  3.742297e+00  1.871149e-03  2.394251
    256      1000  2.092319e-16  4.551291e+00  2.275645e-03  4.499822
    512      100  1.927488e-16  5.762700e-01  2.881350e-03  7.996252
   1024      100  2.308607e-16  6.428590e-01  3.214295e-03  15.928843
   2048      100  2.447624e-16  6.930630e-01  3.465315e-03  32.504982
   4096      100  2.479782e-16  7.530950e-01  3.765475e-03  65.266666
   8192      10  2.578088e-16  1.063520e-01  5.317600e-03  100.135399
  16384      10  2.733986e-16  1.474100e-01  7.370500e-03  155.604097
  32768      10  2.923012e-16  2.088910e-01  1.044455e-02  235.299750
  65536      10  2.829927e-16  3.009800e-01  1.504900e-02  348.387268
 131072      1  3.149670e-16  5.385600e-02  2.692800e-02  413.737374
 262144      1  3.218597e-16  1.138210e-01  5.691050e-02  414.562515
 524288      1  3.281373e-16  2.365150e-01  1.182575e-01  421.177177
1048576      1  3.285898e-16  5.092130e-01  2.546065e-01  411.841803

FFT_SERIAL:
Normal end of execution.

30 November 2021 12:38:36 AM

```

线程n=8:

```

wwwj@ubuntu:~/lab/lab5/fft_serial$ gcc -o lab5_1_1 lab5_1_1.c -lpthread -lm
wwwj@ubuntu:~/lab/lab5/fft_serial$ ./lab5_1_1
num_thread: 8
30 November 2021 12:38:45 AM

FFT_SERIAL
C version

Demonstrate an implementation of the Fast Fourier Transform
of a complex data vector.

Accuracy check:

FFT ( FFT ( X(1:N) ) ) == N * X(1:N)

      N      NITS      Error      Time      Time/Call      MFLOPS
      2      10000  7.859082e-17  1.071268e+01  5.356340e-04  0.018669
      4      10000  1.209837e-16  2.642171e+01  1.321085e-03  0.030278
      8      10000  6.820795e-17  4.500988e+01  2.250494e-03  0.053322
     16      10000  1.438671e-16  5.751579e+01  2.875790e-03  0.111274
     32      1000  1.331210e-16  7.556894e+00  3.778447e-03  0.211727
     64      1000  1.776545e-16  8.810876e+00  4.405438e-03  0.435825
    128      1000  1.929043e-16  1.017418e+01  5.087088e-03  0.880661
    256      1000  2.092319e-16  1.116737e+01  5.583684e-03  1.833915
    512      100  1.927488e-16  1.359933e+00  6.799665e-03  3.388402
   1024      100  2.308607e-16  1.440654e+00  7.203270e-03  7.107883
   2048      100  2.447624e-16  1.810876e+00  9.054380e-03  12.440388
   4096      100  2.479782e-16  2.203238e+00  1.101619e-02  22.308983
   8192      10  2.578088e-16  2.600920e-01  1.300460e-02  40.945512
  16384      10  2.733986e-16  3.286390e-01  1.643195e-02  69.795733
  32768      10  2.923012e-16  5.554090e-01  2.777045e-02  88.496945
  65536      10  2.829927e-16  7.090890e-01  3.545445e-02  147.876501
 131072      1  3.149670e-16  1.324620e-01  6.623100e-02  168.216092
 262144      1  3.218597e-16  1.794430e-01  8.972150e-02  262.957708
 524288      1  3.281373e-16  3.647140e-01  1.823570e-01  273.131056
1048576      1  3.285898e-16  6.020170e-01  3.010085e-01  348.354282

FFT_SERIAL:
Normal end of execution.

30 November 2021 12:40:36 AM

```

线程越多，开销越大，时间也越长，加速效果并不明显。

2)

Valgrind的安装

1. 进入网址<https://sourceware.org/pub/valgrind/>，找到合适的版本并下载，这里我下载了 valgrind-3.15.0.tar
2. 在文件夹中找到并进入下载的压缩包所在位置，在终端执行：
解压缩：

```
1 | $ tar -jxvf valgrind-3.12.0.tar.bz2
```

进入对应文件夹：

```
1 | $ cd valgrind-3.12.0
```

相关配置：

```
1 | $ ./configure
```

编译：

```
1 | $ make
```

安装（给权限）：

```
1 | $ sudo make install
```

最后检查安装情况：

```
wwwj@ubuntu:~/Downloads/valgrind-3.15.0$ valgrind --version
valgrind-3.15.0
```

fft串行版本内存消耗

首先利用指令来收集有关程序的堆分析信息：

```
1 | $ valgrind --tool=massif --stacks=yes ./fft_serial
```

注意：

由于默认情况下，堆栈分析处于关闭状态，因为它会大大减慢 Massif 的速度。因此，一般来说详细信息中的堆栈列为零。但是可以使用该选项打开堆栈分析：`--stacks=yes`。而且这里的指令一定是先写 `--stacks=yes`，再写 `./fft_serial`，如果命令写成 `valgrind --tool=massif ./fft_serial --stacks=yes`，那么仍不能打开堆栈分析。

```
www.j@ubuntu:~/lab/lab5/fft_serial$ valgrind --tool=massif --stacks=yes ./fft_serial
==67910== Massif, a heap profiler
==67910== Copyright (C) 2003-2017, and GNU GPL'd, by Nicholas Nethercote
==67910== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==67910== Command: ./fft_serial
==67910==
30 November 2021 02:44:35 AM

FFT_SERIAL
C version

Demonstrate an implementation of the Fast Fourier Transform
of a complex data vector.

Accuracy check:

  FFT ( FFT ( X(1:N) ) ) == N * X(1:N)

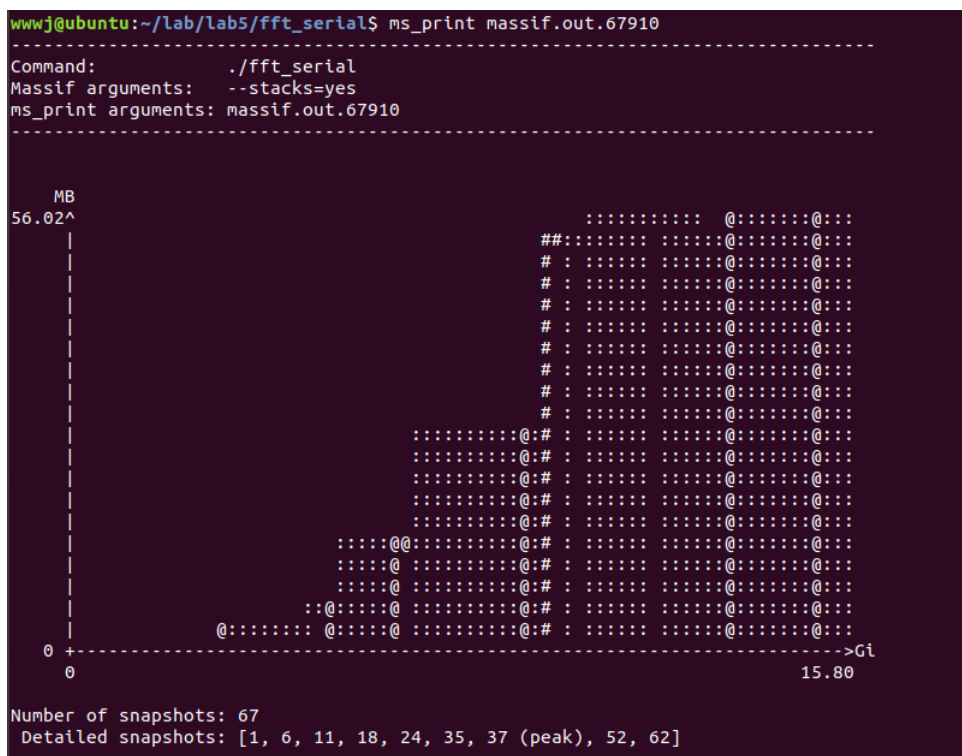
      N      NITS      Error      Time      Time/Call      MFLOPS
      2      10000  7.859082e-17  1.644300e-02  8.221500e-07  12.163231
      4      10000  1.209837e-16  2.498300e-02  1.249150e-06  32.021775
      8      10000  6.820795e-17  3.905900e-02  1.952950e-06  61.445506
     16      10000  1.438671e-16  7.216900e-02  3.608450e-06  88.680735
     32      1000   1.331210e-16  1.347100e-02  6.735500e-06  118.773662
     64      1000   1.776545e-16  3.172100e-02  1.586050e-05  121.055452
    128      1000   1.929043e-16  6.665200e-02  3.332600e-05  134.429575
    256      1000   2.092319e-16  1.448650e-01  7.243250e-05  141.373002
    512      100   1.927488e-16  3.236700e-02  1.618350e-04  142.367226
   1024      100   2.308607e-16  7.436600e-02  3.718300e-04  137.697335
   2048      100   2.447624e-16  1.666990e-01  8.334950e-04  135.141782
   4096      100   2.479782e-16  3.353110e-01  1.676555e-03  146.586303
   8192      10   2.578088e-16  7.348200e-02  3.674100e-03  144.928010
  16384      10   2.733986e-16  1.585820e-01  7.929100e-03  144.641889
  32768      10   2.923012e-16  3.264910e-01  1.632455e-02  150.546263
  65536      10   2.829927e-16  7.016460e-01  3.508230e-02  149.445162
 131072      1   3.149670e-16  1.465790e-01  7.328950e-02  152.015227
 262144      1   3.218597e-16  3.195460e-01  1.597730e-01  147.665500
 524288      1   3.281373e-16  6.652130e-01  3.326065e-01  149.748607
1048576      1   3.285898e-16  1.392694e+00  6.963470e-01  150.582396

FFT_SERIAL:
Normal end of execution.
```

可以看到Massif的所有分析数据都写入一个文件，默认情况下，此文件称为 `massif.out.<pid>`，其中<pid>是进程ID，这里显示了pid为67910。

然后打印生成的表图, `massif.out.67910` 是生成的文件：

```
1 | $ ms_print massif.out.67910
```



从底下的信息可以看到，共进行了67次快照。

下面给出了9个详细快照，详细的快照在图表中由"@"字符组成的条形表示。并在第37次达到了峰值，峰值快照在图表中由由"#"字符组成的条形表示。但这里的峰值快照中并没有后面消耗的内存高，老师给出的文章中也介绍了原因。这里看峰值处的详细信息：

n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
36	9,989,537,714	29,379,872	29,363,082	16,294	496
37	10,327,757,603	58,739,984	58,723,210	16,294	480
99.97% (58,723,210B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.					
->28.56% (16,777,216B) 0x108BC1: main (in /home/wwwj/lab/lab5/fft_serial/fft_serial)					
->28.56% (16,777,216B) 0x108BD6: main (in /home/wwwj/lab/lab5/fft_serial/fft_serial)					
->28.56% (16,777,216B) 0x108EB: main (in /home/wwwj/lab/lab5/fft_serial/fft_serial)					
->14.28% (8,388,608B) 0x108BAC: main (in /home/wwwj/lab/lab5/fft_serial/fft_serial)					
->00.01% (2,954B) in 1+ places, all below ms_print's threshold (01.00%)					

- 表中的数据是：它的编号、花费的时间、此时的总内存消耗量、此时分配的有用堆字节数（这反映了程序要求的字节数）、此时分配的额外堆字节数（反映了分配的字节数超过了程序要求的字节数）、堆栈的大小。
- 除了基本计数外，它还提供了一个分配树，该树准确指示哪些代码段负责分配堆内存。

fft并行版本内存消耗

收集有关程序的堆分析信息：

```
1 | $ valgrind --tool=massif --stacks=yes ./lab5_1_1
```

```
wwwj@ubuntu:~/lab/lab5/fft_serial$ valgrind --tool=massif --stacks=yes ./lab5_1_1
==63782== Massif, a heap profiler
==63782== Copyright (C) 2003-2017, and GNU GPL'd, by Nicholas Nethercote
==63782== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==63782== Command: ./lab5_1_1
==63782==
num_thread: 2
30 November 2021 02:40:12 AM

FFT_SERIAL
C version

Demonstrate an implementation of the Fast Fourier Transform
of a complex data vector.

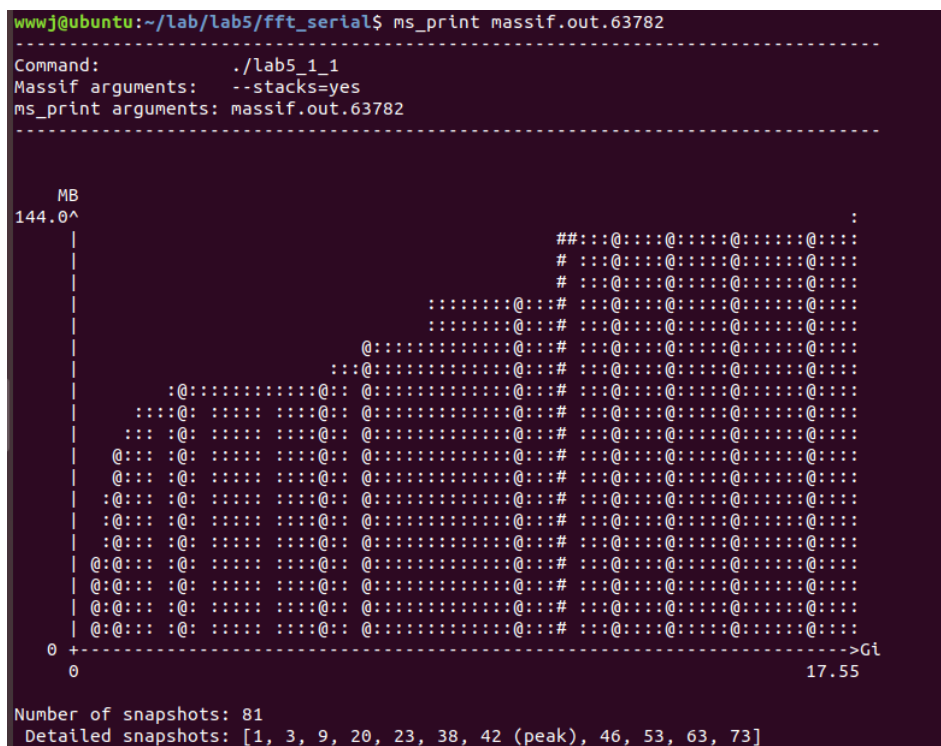
Accuracy check:

  FFT ( FFT ( X(1:N) ) ) == N * X(1:N)

      N      NITS      Error      Time      Time/Call      MFLOPS
      2      10000  7.859082e-17  5.679067e+00  2.839533e-04  0.035217
      4      10000  1.209837e-16  1.159773e+01  5.798867e-04  0.068979
      8      10000  6.820795e-17  1.733022e+01  8.665112e-04  0.138486
     16      10000  1.438671e-16  2.425550e+01  1.212775e-03  0.263858
     32      1000   1.331210e-16  3.105086e+00  1.552543e-03  0.515284
     64      1000   1.776545e-16  3.620905e+00  1.810453e-03  1.060508
    128      1000   1.929043e-16  4.473525e+00  2.236762e-03  2.002895
    256      1000   2.092319e-16  5.336657e+00  2.668329e-03  3.837608
    512      100   1.927488e-16  6.123080e-01  3.061540e-03  7.525624
   1024      100   2.308607e-16  7.029520e-01  3.514760e-03  14.567140
   2048      100   2.447624e-16  8.865630e-01  4.432815e-03  25.410490
   4096      100   2.479782e-16  1.180487e+00  5.902435e-03  41.637053
   8192      10   2.578088e-16  1.722090e-01  8.610450e-03  61.841135
  16384      10   2.733986e-16  2.673510e-01  1.336755e-02  85.795826
  32768      10   2.923012e-16  4.503310e-01  2.251655e-02  109.146383
  65536      10   2.829927e-16  8.435640e-01  4.217820e-02  124.303076
 131072      1   3.149670e-16  1.698390e-01  8.491950e-02  131.196251
 262144      1   3.218597e-16  3.417220e-01  1.708610e-01  138.082769
 524288      1   3.281373e-16  7.013170e-01  3.506585e-01  142.039506
1048576      1   3.285898e-16  1.450146e+00  7.250730e-01  144.616611
```

然后打印生成的表图,massif.out.63782 是生成的文件：

```
1 | $ ms_print massif.out.63782
```



从底下的信息可以看到，共进行了81次快照。可能是运行时间比串行更长的原因，进行的快照次数也就更多。

下面给出11个详细快照，在编号为42的快照次数中内存消耗达到峰值。

看峰值处的详细信息：

n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
39	10,849,416,435	121,613,512	29,363,626	16,310	92,233,576
40	11,176,343,103	121,616,504	29,363,626	16,310	92,236,568
41	11,395,691,636	121,618,616	29,363,626	16,310	92,238,680
42	11,770,161,986	150,981,440	58,723,754	16,310	92,241,376
38.89% (58,723,754B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.					
->11.11% (16,777,216B) 0x10972B: main (in /home/wwwj/lab/lab5/fft_serial/lab5_1_1)					
->11.11% (16,777,216B) 0x109740: main (in /home/wwwj/lab/lab5/fft_serial/lab5_1_1)					
->11.11% (16,777,216B) 0x109755: main (in /home/wwwj/lab/lab5/fft_serial/lab5_1_1)					
->05.56% (8,388,608B) 0x109716: main (in /home/wwwj/lab/lab5/fft_serial/lab5_1_1)					
->00.00% (3,498B) in 1+ places, all below ms_print's threshold (01.00%)					

对比

两个内存消耗的图表上可以清楚看出，并行版本的内存消耗要比串行版本的内存消耗要大得多，无论是从峰值上来看还是从整体上来看，并且并行版本的堆栈大小也非常大。因此可以看出并行版本内存代价是非常大的。

总结思考

这次作业的并行化效果实际不是很理想，而且不知道是什么原因在虚拟机上运行同一段代码不同时间可能有不一样的效果，有时候非常慢但过一段时间可能又会变快，这个问题之后可能还是需要再找一下原因。在这次作业中对MPI的运用有了更深一步的了解，可能之前就是认为MPI的并行化是从初始化开始(MPI_Init()), 释放资源结束(MPI_Finalize())。于是最初的想法是在while循环中多次调用MPI_Init() 和 MPI_Finalize()。但是实际上MPI是从程序的一开始就进行了并行化，MPI_Finalize() 之后就不再能进行MPI_Init() 初始化操作了，因此这个思路就是错误的。MPI的并

行化最重要的就是选择在不同进程上运行的内容以及不同进程间的通信，如果不规定运行的进程的话那么这部分内容所有的进程都需要运行。以及在这次实验中对数据依赖这部分内容有了更深的认识。最后希望通过以后的学习能将代码并行化以后有更好的效果。