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。

```
struct for_index {
    int start;
    int end;
    int increment;
    int mj;// need in cfft2 and step
    double *a;//need in step
    double *b;//need in step
```

```
    double *c;//need in step
    double *d;//need in step
    double *x;// need in cfft2
    double *y;// need in cfft2
    double *w;// need in cfft2 and step
    double sgn;// need in cfft2 and step
    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。
};
```

- **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。

```
void step_ ( int n, int mj, double a[], double b[], double c[],
  double d[], double w[], double sgn )
{
  // struct timeval val;
  // struct timeval newVal;
  // int ret = gettimeofday(&val, NULL);
  int lj;
  lj  = n / (mj*2);
  struct for_index first;
  first.mj = mj;
  first.a = a;
  first.b = b;
  first.c = c;
  first.d = d;
  first.w = w;
  first.sgn = sgn;

  parallel_for(0,lj,1,step_parallel,&first,num_thread);

  // ret = gettimeofday(&newVal, NULL);
  // unsigned long diff = (newVal.tv_sec * Converter + newVal.tv_usec) - (val.tv_sec * Converter + val.tv_usec);
  // printf("diff:  sec --- %ld, usec --- %ld\n", diff / Converter, diff % Converter);
  return;
}
```

- **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 )
```

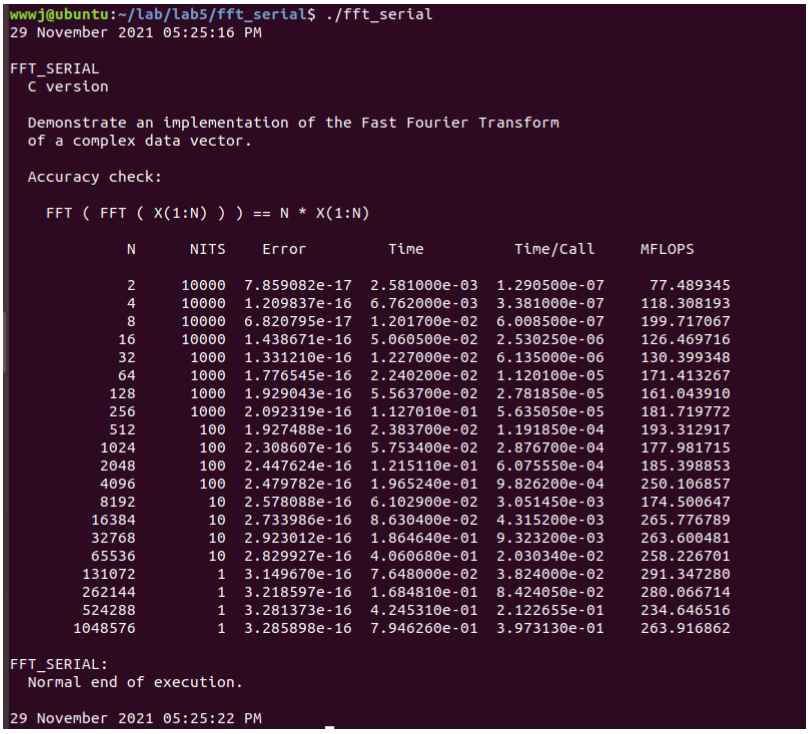
```
      {
        wjw[1] = - wjw[1];
      }

    for (int k = 0; k < thread_assign->mj; k++ )
    {
      thread_assign->c[(jc+k)*2+0] = thread_assign->a[(ja+k)*2+0] + thread_assign->b[(jb+k)*2+0];
      thread_assign->c[(jc+k)*2+1] = thread_assign->a[(ja+k)*2+1] + thread_assign->b[(jb+k)*2+1];

        ambr = thread_assign->a[(ja+k)*2+0] - thread_assign->b[(jb+k)*2+0];
        ambu = thread_assign->a[(ja+k)*2+1] - thread_assign->b[(jb+k)*2+1];

        thread_assign->d[(jd+k)*2+0] = wjw[0] * ambr - wjw[1] * ambu;
        thread_assign->d[(jd+k)*2+1] = wjw[1] * ambr + wjw[0] * ambu;
     }
  }

    ret = gettimeofday(&newVal, NULL);
    unsigned long diff = (newVal.tv_sec * Converter + newVal.tv_usec) - (val.tv_sec * Converter + val.tv_usec);
    thread_assign->time = diff ;
}
```

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：



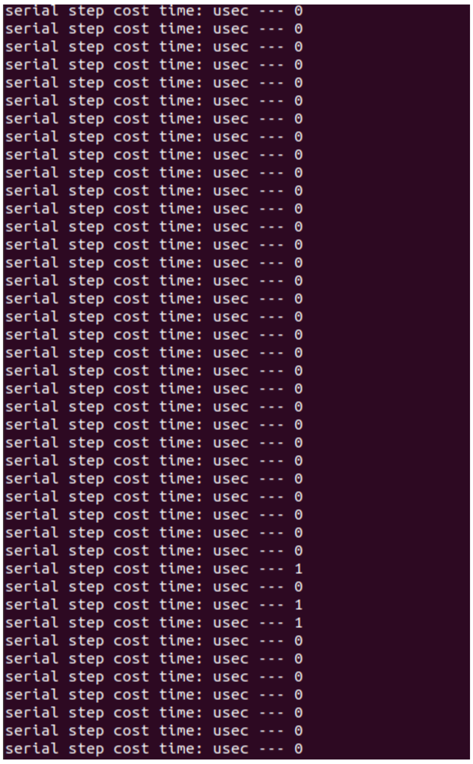
```
wwwj@ubuntu:~/lab/lab5/fft_serial$ ./fft_serial
29 November 2021 05:25:16 PM

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  2.581000e-03  1.290500e-07     77.489345
             4     10000  1.209837e-16  6.762000e-03  3.381000e-07    118.308193
             8     10000  6.820795e-17  1.201700e-02  6.008500e-07    199.717067
            16     10000  1.438671e-16  5.060500e-02  2.530250e-06    126.469716
            32      1000  1.331210e-16  1.227000e-02  6.135000e-06    130.399348
            64      1000  1.776545e-16  2.240200e-02  1.120100e-05    171.413267
           128      1000  1.929043e-16  5.563700e-02  2.781850e-05    161.043910
           256      1000  2.092319e-16  1.127010e-01  5.635050e-05    181.719772
           512       100  1.927488e-16  2.383700e-02  1.191850e-04    193.312917
          1024       100  2.308607e-16  5.753400e-02  2.876700e-04    177.981715
          2048       100  2.447624e-16  1.215110e-01  6.075550e-04    185.398853
          4096       100  2.479782e-16  1.965240e-01  9.826200e-04    250.106857
          8192        10  2.578088e-16  6.102900e-02  3.051450e-03    174.500647
         16384        10  2.733986e-16  8.630400e-02  4.315200e-03    265.776789
         32768        10  2.923012e-16  1.864640e-01  9.323200e-03    263.600481
         65536        10  2.829927e-16  4.060680e-01  2.030340e-02    258.226701
        131072         1  3.149670e-16  7.648000e-02  3.824000e-02    291.347280
        262144         1  3.218597e-16  1.684810e-01  8.424050e-02    280.066714
        524288         1  3.281373e-16  4.245310e-01  2.122655e-01    234.646516
       1048576         1  3.285898e-16  7.946260e-01  3.973130e-01    263.916862

FFT_SERIAL:
  Normal end of execution.

29 November 2021 05:25:22 PM
```

可以看到误差完全相同，说明算法是正确的。但是并行后的效果并不好，虽然在N=65536到N=1048576之间，并行的时间几乎是串行的一半，但是之前N更小的时候运算时间并未减小，猜测可能是线程创建的开销比较大，接下来分析各个部分的运行时间查看原因。

分析

在计时函数运行时间的方法已经在上面介绍过了，这里就不再赘述。

- 首先输出串行时每次调用函数step()所花费的时间，单位为微秒：



```
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 1
serial step cost time: usec --- 0
serial step cost time: usec --- 1
serial step cost time: usec --- 1
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
serial step cost time: usec --- 0
```

可以看到几乎为1。

- 然后输出并行时每次调用step()所花费的时间，单位为微秒：

```
    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   if ( n == 2 )
13   {
14     return;
15   }
16
17   struct for_index first;
18   first.n = n;
19   first.mj = mj;
20   first.x = x;
21   first.y = y;
22   first.w = w;
23   first.sgn = sgn;
24
25   parallel_for(0, m-2, 1, cfft2_parallel,&first, num_thread);
26
27   mj = mj * pow(2,m-2);
28
29   if ( (m-2)%2 == 0 )
30   {
```

```
    ccopy ( n, y, x );
  }

  mj = n / 2;
  step ( n, mj, &x[0*2+0], &x[(n/2)*2+0], &y[0*2+0], &y[mj*2+0], w, sgn
);

  return;
}
```

- **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。

```
void parallel_for_cfft2(int start, int end, int increment, void *
(*functor)(void*), void *arg , int num_threads){
    unsigned long time=0;

    struct for_index *first;
    first = (struct for_index*)arg;
    pthread_t pth[num_threads];
    int divid = (end - start )/num_threads;
    struct for_index thread_assign[num_threads];

    for(int i=0; i<num_threads; ++i){
        thread_assign[i].start = i*divid ;
        if(i == (num_threads - 1) ){
            thread_assign[i].end = end;
        }
        else{
            thread_assign[i].end = thread_assign[i].start + divid;
        }
        thread_assign[i].increment = increment;
        thread_assign[i].n = first->n;
        thread_assign[i].mj = first->mj;
        thread_assign[i].x =  first->x;
        thread_assign[i].y =  first->y;
        thread_assign[i].w =  first->w;
        thread_assign[i].sgn =first->sgn;
    }

    // struct timeval val;
    // struct timeval newVal;
    // int ret = gettimeofday(&val, NULL);

    for(int i=0; i<num_threads; ++i){
        pthread_create(&pth[i], NULL, functor, &thread_assign[i]);
    }

    for(int i=0; i<num_threads; ++i){
        // time += thread_assign[i].time;
        pthread_join(pth[i], NULL);
    }

    // ret = gettimeofday(&newVal, NULL);
    // unsigned long diff = (newVal.tv_sec * Converter + newVal.tv_usec)
- (val.tv_sec * Converter + val.tv_usec);
```

```
    // printf("diff:  sec --- %ld, usec --- %ld\n", diff / Converter,
diff % Converter - time);

}
```

- **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$mj = mj * 2^{start}$，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。

```
void *cfft2_parallel(void *arg){
    // struct timeval val;
    // struct timeval newVal;
    // int ret = gettimeofday(&val, NULL);

    struct for_index *thread_assign;
    thread_assign = (struct for_index*)arg;
    int mj = thread_assign->mj * pow(2,thread_assign->start);

    for (int j = thread_assign->start; j < thread_assign->end;
j=j+thread_assign->increment)
  {
      mj = mj * 2;
      if(j%2 == 0){
          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 );
      }
      else{
          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 );
      }
  }

    // ret = gettimeofday(&newVal, NULL);
    // unsigned long diff = (newVal.tv_sec * Converter + newVal.tv_usec)
- (val.tv_sec * Converter + val.tv_usec);
    // thread_assign->time = diff ;
}
```

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。

```
# define M 500
# define N 500

  double diff;
  double epsilon = 0.001;
  int i;
  int iterations;
  int iterations_print;
  int j;
  double mean;
  double my_diff;
  double u[M][N];
  double w[M][N];
  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：

```
  MPI_Init(NULL, NULL);
  int numprocess,rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  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：

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

```
    printf ( "  Number of threads =                %d\n",
omp_get_max_threads ( ) );
    /*
      Set the boundary values, which don't change.
    */
      mean = 0.0;

    #pragma omp parallel shared ( w ) private ( i, j )
      {
    #pragma omp for
        for ( i = 1; i < M - 1; i++ )
        {
          w[i][0] = 100.0;
        }
    #pragma omp for
        for ( i = 1; i < M - 1; i++ )
        {
          w[i][N-1] = 100.0;
        }
    #pragma omp for
        for ( j = 0; j < N; j++ )
        {
          w[M-1][j] = 100.0;
        }
    #pragma omp for
        for ( j = 0; j < N; j++ )
        {
          w[0][j] = 0.0;
        }
    /*
      Average the boundary values, to come up with a reasonable
      initial value for the interior.
    */
    #pragma omp for reduction ( + : mean )
        for ( i = 1; i < M - 1; i++ )
        {
          mean = mean + w[i][0] + w[i][N-1];
        }
    #pragma omp for reduction ( + : mean )
        for ( j = 0; j < N; j++ )
        {
          mean = mean + w[M-1][j] + w[0][j];
        }
      }
    /*
      OpenMP note:
      You cannot normalize MEAN inside the parallel region.  It
      only gets its correct value once you leave the parallel region.
      So we interrupt the parallel region, set MEAN, and go back in.
    */
      mean = mean / ( double ) ( 2 * M + 2 * N - 4 );
      printf ( "\n" );
      printf ( "  MEAN = %f\n", mean );
    /*
      Initialize the interior solution to the mean value.
    */
    #pragma omp parallel shared ( mean, w ) private ( i, j )
      {
```

```
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
```

```
      for(int i=1;i<numprocess;++i){
        MPI_Send(u,M*N,MPI_DOUBLE,i,1,MPI_COMM_WORLD);
        MPI_Send(w,M*N,MPI_DOUBLE,i,2,MPI_COMM_WORLD);
      }
    }

    else if (first == 1 && rank ! = 0) {
      first = 0;
      MPI_Recv(u,M*N,MPI_DOUBLE,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
      MPI_Recv(w,M*N,MPI_DOUBLE,0,2,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }

    else{
      if(rank>0){
      MPI_Recv(u[start-1],N,MPI_DOUBLE,rank-
1,3,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
      MPI_Send(u[start],N,MPI_DOUBLE,rank-1,4,MPI_COMM_WORLD);}
      if(rank<numprocess-1){
      MPI_Send(u[end-1],N,MPI_DOUBLE,rank+1,3,MPI_COMM_WORLD);

 MPI_Recv(u[end],N,MPI_DOUBLE,rank+1,4,MPI_COMM_WORLD,MPI_STATUS_IGNORE)
;}
    }



    # pragma omp parallel for private ( i, j ) shared ( u, w )
    for ( i = start; i < end; i++ )
      {
        for ( j = 1; j < N - 1; j++ )
        {
          w[i][j] = ( u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] ) /
4.0;
        }
    }



    diff = 0.0;
    my_diff = 0.0;

    for ( i = start; i < end; i++ )
    {
      for ( j = 1; j < N - 1; j++ )
      {
        if ( my_diff < fabs ( w[i][j] - u[i][j] ) )
        {
          my_diff = fabs ( w[i][j] - u[i][j] );
        }
      }
    }

    double all_diff[numprocess];

 MPI_Allgather(&my_diff,1,MPI_DOUBLE,all_diff,1,MPI_DOUBLE,MPI_COMM_WORL
D);

    for(int i=0;i<numprocess;++i){
        if(diff < all_diff[i]){
            diff = all_diff[i];
```

```
        }
    }

    if(rank == 0){
      iterations++;
      if ( iterations == iterations_print )
      {
        printf ( "  %8d  %f\n", iterations, diff );
       iterations_print = 2 * iterations_print;
      }
    }

  }
```

- 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：

```
  if(rank == 0){
    wtime = omp_get_wtime ( ) - wtime;
    printf ( "\n" );
    printf ( "  %8d  %f\n", iterations, diff );
    printf ( "\n" );
    printf ( "  Error tolerance achieved.\n" );
    printf ( "  Wallclock time = %f\n", wtime );
  /*
    Terminate.
  */
    printf ( "\n" );
    printf ( "HEATED_PLATE_OPENMP:\n" );
    printf ( "  Normal end of execution.\n" );
  }

  MPI_Finalize();

  return 0;

  # undef M
  # undef N
}
```

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$ mpiexec -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
    MPI_Recv(buffer,2*M*N,MPI_DOUBLE,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
21       MPI_Unpack(buffer, 2*M*N, &position, u, M*N, MPI_DOUBLE,
   MPI_COMM_WORLD);
22       MPI_Unpack(buffer, 2*M*N, &position, w, M*N, MPI_DOUBLE,
   MPI_COMM_WORLD);
23     }
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  }
```

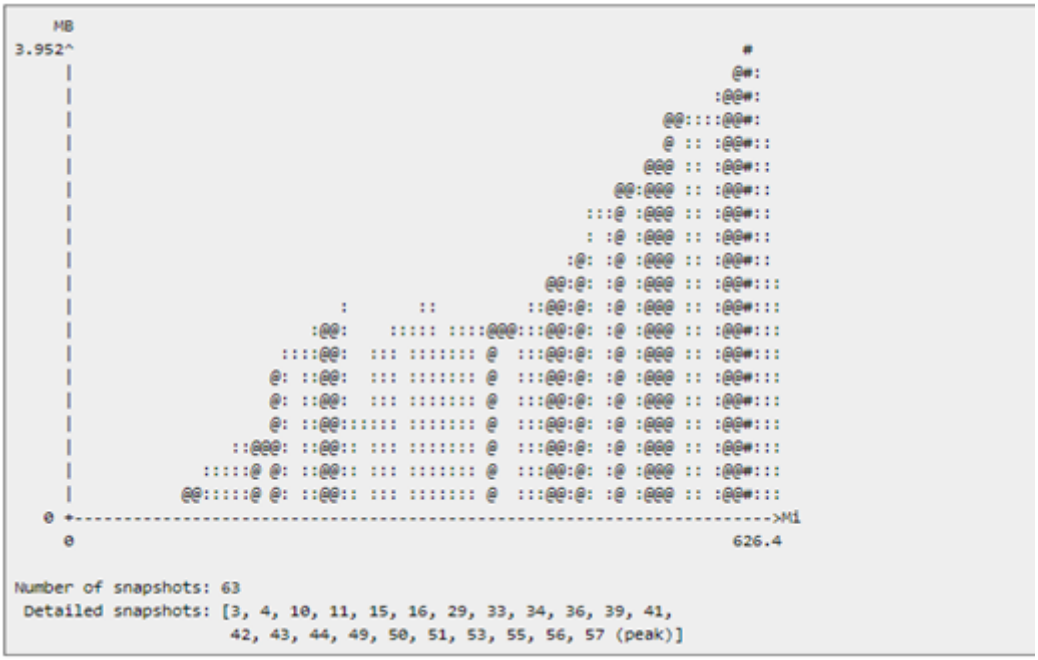
运行结果：

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

     16955  0.001000

  Error tolerance achieved.
  Wallclock time = 67.703033

HEATED_PLATE_OPENMP:
  Show Applications xecution.
```

任务3：

> 性能分析任务1和并行化fft应用，包括：
>
> 1）不同问题规模的并行化fft应用并行执行时间对比，其中问题规模定义为N变化范围2，4，6，8，16，32，64，128，……，2097152；并行规模为1，2，4，8 进/线程。
>
> 2）内存消耗对比，内存消耗采用 “valgrind massif”工具采集，注意命令valgrind命令中增加--stacks=yes 参数采集程序运行栈内内存消耗。Valgrind massif输出日志（massif.out.*pid*）经过ms_print打印后示例如下图，其中x轴为程序运行时间，y轴为内存消耗量：



参考文献：

*Valgrind massif：https://valgrind.org/docs/manual/ms-manual.html**(https://valgrind.org/docs/manual/ms-manual.html)

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：

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

进入对应文件夹：

```
$ cd valgrind-3.12.0
```

相关配置：

```
$ ./configure
```

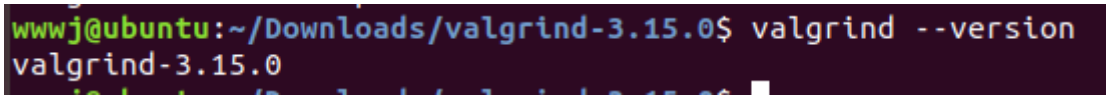
编译：

```
$ make
```

安装（给权限）：

```
$ sudo make install
```

最后检查安装情况：



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：

```
$ 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。

```
wwwj@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
```

```
wwwj@ubuntu:~/lab/lab5/fft_serial$ ms_print massif.out.67910
--------------------------------------------------------------------------------
Command:            ./fft_serial
Massif arguments:   --stacks=yes
ms_print arguments: massif.out.67910
--------------------------------------------------------------------------------


    MB
56.02^                                          :::::::::::  @:::::::@:::
     |                                      ##::::::::: ::::::@:::::::@:::
     |                                      # : :::::: ::::::@:::::::@:::
     |                                      # : :::::: ::::::@:::::::@:::
     |                                      # : :::::: ::::::@:::::::@:::
     |                                      # : :::::: ::::::@:::::::@:::
     |                                      # : :::::: ::::::@:::::::@:::
     |                                      # : :::::: ::::::@:::::::@:::
     |                                      # : :::::: ::::::@:::::::@:::
     |                                      # : :::::: ::::::@:::::::@:::
     |                                :::::::::@:# : :::::: ::::::@:::::::@:::
     |                                :::::::::@:# : :::::: ::::::@:::::::@:::
     |                                :::::::::@:# : :::::: ::::::@:::::::@:::
     |                                :::::::::@:# : :::::: ::::::@:::::::@:::
     |                                :::::::::@:# : :::::: ::::::@:::::::@:::
     |                           :::::@@:::::::::@:# : :::::: ::::::@:::::::@:::
     |                           :::::@ :::::::::@:# : :::::: ::::::@:::::::@:::
     |                           :::::@ :::::::::@:# : :::::: ::::::@:::::::@:::
     |                        ::@:::::@ :::::::::@:# : :::::: ::::::@:::::::@:::
     |                @::::::: @:::::@ :::::::::@:# : :::::: ::::::@:::::::@:::
   0 +------------------------------------------------------------------------>Gi
     0                                                                   15.80

Number of snapshots: 67
 Detailed snapshots: [1, 6, 11, 18, 24, 35, 37 (peak), 52, 62]
```

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) 0x108BEB: 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

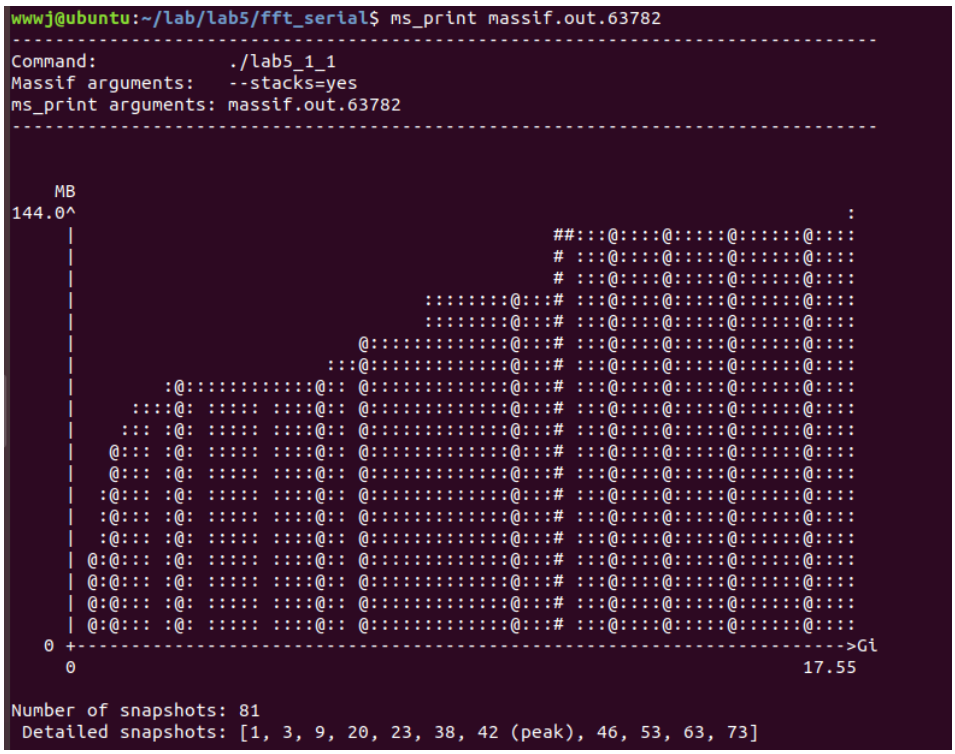
收集有关程序的堆分析信息：

```
$ 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：

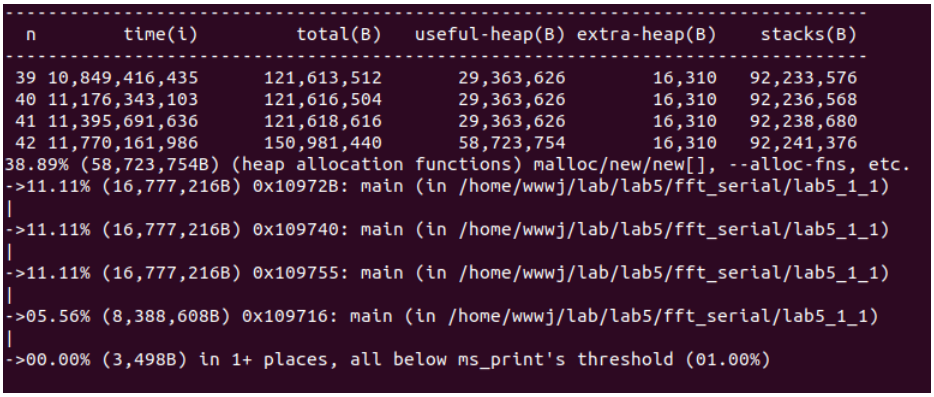
```
$ ms_print massif.out.63782
```



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：



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。