中山大学计算机院本科生实验报告

（2024 学年秋季学期）

课程名称：高性能计算程序设计　　　　　　　　　　　　**批改人：**

实验	**修改基于 OpenMP 框架的多线程编程**	专业（方向）	**信息与计算科学**
学号	22336044	姓名	**陈圳煌**
Email	2576926165@qq.com	完成日期	2024.12.04

1. 实验目的

本次实验使用上次实验中自己构造的基于 Pthreads 的 parallel_for 函数替换 heated_plate_openmp 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。并尝试将 heated_plate_openmp 应用改造成基于 MPI 的进程并行应用，熟悉不同框架下的多线程编程。

2. 实验过程和核心代码

（本实验在 Windows 环境下进行，并且为了适应 parallel_fun 函数，因此将二维矩阵修改成二维指针）

task1 代码：（相关代码文件存于 task1 文件夹）

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <omp.h>
# include "parallel_fun.h"

int main ( int argc, char *argv[] );

int flag=1;

struct for_index{
    int start;
    int end;
    int increment;
    double **A;
    double **B;
    double **C;
    int scale1;
    int scale2;
    double mean;
};
```

```
void* functor1(void* arg){
    struct for_index *index=(struct for_index *)arg;
    int n=index->scale2;
    for(int i=index->start;i<index->end;i+=index->increment){
        for(int j=1;j<n-1;j++){
            index->B[i][j]=index->mean;
        }
    }

    return NULL;
}

void* functor2(void* arg){
    struct for_index *index=(struct for_index *)arg;
    int n=index->scale2;
    for(int i=index->start;i<index->end;i+=index->increment){
        for(int j=0;j<n;j++){
            index->A[i][j]=index->B[i][j];
        }
    }

    return NULL;
}

void* functor3(void* arg){
    struct for_index *index=(struct for_index *)arg;
    int n=index->scale2;
    for(int i=index->start;i<index->end;i+=index->increment){
        for(int j=1;j<n-1;j++){
            index->B[i][j]=(index->A[i-1][j]+index->A[i+1][j]+index->A[i
][j-1]+index->A[i][j+1])/4.0;
        }
    }

    return NULL;
}

void* functor4(void* arg){
    struct for_index *index=(struct for_index *)arg;
    int n=index->scale2;
    double my_diff=0.0;
    for(int i=index->start;i<index->end;i+=index->increment){
        for(int j=1;j<n-1;j++){
            if(my_diff<fabs(index->B[i][j]-index->A[i][j])){
                my_diff=fabs(index->B[i][j]-index->A[i][j]);
            }
        }
    }

    while(!flag);
    flag=0;
    if(index->C[0][0]<my_diff){
      index->C[0][0]=my_diff;
    }
    flag=1;

    return NULL;
}

/***********************************************************************
********/

int main ( int argc, char *argv[] )
{
# 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=malloc(sizeof(double*)*M);
  double **w=malloc(sizeof(double*)*M);
  for(int x=0;x<M;x++){
    u[x]=malloc(sizeof(double)*N);
    w[x]=malloc(sizeof(double)*N);
  }
  double wtime;

  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.
*/

  double **v=malloc(sizeof(double*));
  v[0]=malloc(sizeof(double));
  struct for_index* data=malloc(sizeof(struct for_index));
  data->A=u;
  data->B=w;
  data->C=v;
  data->scale1=M;
  data->scale2=N;
  data->mean=mean;

  parallel_for(1, M-1, 1, functor1, (void*)data, 16);
/*
  iterate until the  new solution W differs from the old solution U
  by no more than EPSILON.
*/
  iterations = 0;
  iterations_print = 1;
  printf ( "\n" );
  printf ( " Iteration  Change\n" );
  printf ( "\n" );
  wtime = omp_get_wtime ( );

  diff = epsilon;

  while ( epsilon <= diff )
  {
/*
  Save the old solution in U.
*/
  parallel_for(0, M, 1, functor2, (void*)data, 16);

/*
  Determine the new estimate of the solution at the interior points.
  The new solution W is the average of north, south, east and west neighbors.
*/
  parallel_for(1, M-1, 1, functor3, (void*)data, 16);

/*
  C and C++ cannot compute a maximum as a reduction operation.

  Therefore, we define a private variable MY_DIFF for each thread.
  Once they have all computed their values, we use a CRITICAL section
  to update DIFF.
*/
    diff = 0.0;
    data->C[0][0]=diff;
    parallel_for(1, M-1, 1, functor4, (void*)data, 16);

    diff=data->C[0][0];
    iterations++;
    if ( iterations == iterations_print )
    {
      printf ( "  %8d  %f\n", iterations, diff );
      iterations_print = 2 * iterations_print;
    }
  }
  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" );

  free(data);
  return 0;

# undef M
# undef N
}
```

在这个部分的实验中，使用构造的 parallel_fun 函数替代原始 heated_plate_openmp 文件中的二维矩阵运算。

Parallel_fun 实现如下:

```
struct for_index{
    int start;
    int end;
    int increment;
    double **A;
    double **B;
    double **C;
    int scale1;
    int scale2;
    double num1;
};

int min(int a, int b){
    if(a>=b){
        return a;
    }
    return b;
}

void parallel_for(int start, int end, int increment,
void*(functor)(void*) ,void *arg , int num_threads){

    pthread_t* thread_handles=malloc(num_threads*sizeof(pthread_t));
    struct for_index* indice=malloc(num_threads*sizeof(struct 
for_index));

    struct for_index *data=(struct for_index *)arg;

    for(int i=0;i<num_threads;i++){
        indice[i].start=start+(end-start)/num_threads*i;
        indice[i].end=min(start+(end-start)/num_threads*(i+1),end);
        indice[i].increment=increment;
        indice[i].A=data->A;
        indice[i].B=data->B;
        indice[i].C=data->C;
        indice[i].scale1=data->scale1;
        indice[i].scale2=data->scale2;
        indice[i].num1=data->num1;

        pthread_create(&thread_handles[i], NULL, functor, &indice[i]);
    }

    for(int i=0;i<num_threads;i++){
        pthread_join(thread_handles[i], NULL);
    }

    free(thread_handles);
    free(indice);
}
```

（注：lab3 提交的的实验报告和实验代码 parallel_fun.c 中，在最后多打了一个 free(data)，如果需要测试代码请自行删去。）

其中 functor1 替换以下部分的代码：

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

functor2 替换以下部分代码：

```
# pragma omp for
      for ( i = 0; i < M; i++ )
      {
        for ( j = 0; j < N; j++ )
        {
          u[i][j] = w[i][j];
        }
      }
```

functor3 替换以下部分代码：

```
# pragma omp for
      for ( i = 1; i < M - 1; i++ )
      {
        for ( j = 1; j < N - 1; j++ )
        {
          w[i][j] = ( u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] ) / 4.0;
        }
      }
```

functor4 替换以下部分代码：

```
# pragma omp parallel shared ( diff, u, w ) private ( i, j, my_diff )
    {
      my_diff = 0.0;
# pragma omp for
      for ( i = 1; i < M - 1; i++ )
      {
        for ( j = 1; j < N - 1; j++ )
        {
          if ( my_diff < fabs ( w[i][j] - u[i][j] ) )
          {
            my_diff = fabs ( w[i][j] - u[i][j] );
          }
        }
      }
# pragma omp critical
      {
        if ( diff < my_diff )
        {
          diff = my_diff;
        }
      }
    }
```
使用一个全局变量 flag 来保证临界区每次只能由一个线程进行访问。

运行文件为 build.bat，可以在 Windows 环境下进行：

```
@echo off
REM 设置源文件和目标文件的名称
set SRC_MAIN=heated_plate_openmp.c
set SRC_LIB=parallel_fun.c
set OBJ_LIB=parallel_fun.o
set EXEC_FILE=heated_plate_openmp.exe
set DLL_FILE=function.dll

REM 1. 编译 parallel_fun.c 为目标文件
echo Compiling %SRC_LIB% to %OBJ_LIB% ...
gcc -c %SRC_LIB%

REM 2. 使用 -shared 参数生成动态链接库 function.dll
echo Compile %SRC_LIB% generate %DLL_FILE% ...
gcc -shared -o %DLL_FILE% %SRC_LIB%

REM 3. 链接目标文件并生成最终可执行文件
echo Linking %OBJ_MAIN% and parallel_fun.o to generate %EXEC_FILE% ...
gcc -g -Wall -fopenmp -o %EXEC_FILE% %SRC_MAIN% -L. -lfunction

REM 4. 如果编译和链接成功，运行可执行文件
if %ERRORLEVEL% == 0 (
    echo Compilation and linking succeeded, running %EXEC_FILE% ...
    %EXEC_FILE%
) else (
    echo Compilation or linking failed!
)

pause
```

task2 代码：（相关文件见 task2 文件夹）

```
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <mpi.h>
# include <omp.h>
```

```
int main ( int argc, char *argv[] );}
/***********************************************************************
********/

int main ( int argc, char *argv[] )
{
# define M 500
# define N 500

  int comm_sz;
  int my_rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  double diff;
  double epsilon = 0.001;
  int i;
  int iterations;
  int iterations_print;
  int j;
  double mean;
  double my_diff;
  double* u=malloc(sizeof(double)*M*N);
  double* local_u=malloc(sizeof(double)*(M/comm_sz)*N);
  double* w=malloc(sizeof(double)*M*N);
  double* local_w=malloc(sizeof(double)*(M/comm_sz)*N);

  double wtime;
  if(my_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 processes =           %d\n", comm_sz );
  }
/*
  Set the boundary values, which don't change.
*/
  mean = 0.0;
  if(my_rank==0){
    for ( i = 1; i < M - 1; i++ )
    {
      w[i*N] = 100.0;
    }

    for ( i = 1; i < M - 1; i++ )
    {
      w[i*N+N-1] = 100.0;
    }

    for ( j = 0; j < N; j++ )
    {
      w[(M-1)*N+j] = 100.0;
    }

    for ( j = 0; j < N; j++ )
    {
      w[j] = 0.0;
    }

    for ( i = 1; i < M - 1; i++ )
    {
      mean = mean + w[i*N] + w[i*N+N-1];
    }

    for ( j = 0; j < N; j++ )
    {
      mean = mean + w[(M-1)*N+j] + w[j];
    }

    mean = mean / ( double ) ( 2 * M + 2 * N - 4 );
    printf ( "\n" );
    printf ( "  MEAN = %f\n", mean );
  }
  MPI_Barrier(MPI_COMM_WORLD);
/*
```

```
  Initialize the interior solution to the mean value.
*/
  MPI_Bcast(&mean, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Scatter(w, (M/comm_sz)*N, MPI_DOUBLE, local_w, (M/comm_sz)*N,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
  fuctor1(my_rank, comm_sz, M/comm_sz, N, local_w, mean);
  MPI_Gather(local_w, (M/comm_sz)*N, MPI_DOUBLE, w, (M/comm_sz)*N,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Barrier(MPI_COMM_WORLD);
/*
  iterate until the  new solution W differs from the old solution U
  by no more than EPSILON.
*/
  iterations = 0;
  iterations_print = 1;
  if(my_rank==0){
    printf ( "\n" );
    printf ( " Iteration  Change\n" );
    printf ( "\n" );
  }
  wtime = MPI_Wtime();

  diff = epsilon;

  while ( epsilon <= diff )
  {
/*
  Save the old solution in U.
*/
  MPI_Bcast(w, M*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Scatter(u, (M/comm_sz)*N, MPI_DOUBLE, local_u, (M/comm_sz)*N,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
  fuctor2(my_rank, M/comm_sz, N, local_u, w);
  MPI_Gather(local_u, (M/comm_sz)*N, MPI_DOUBLE, u, (M/comm_sz)*N,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Barrier(MPI_COMM_WORLD);

/*
  Determine the new estimate of the solution at the interior points.
  The new solution W is the average of north, south, east and west neighbors.
*/
  MPI_Bcast(u, M*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Scatter(w, (M/comm_sz)*N, MPI_DOUBLE, local_w, (M/comm_sz)*N,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
  fuctor3(my_rank, comm_sz, M/comm_sz, N, local_w, u);
  MPI_Gather(local_w, (M/comm_sz)*N, MPI_DOUBLE, w, (M/comm_sz)*N,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Barrier(MPI_COMM_WORLD);

/*
  C and C++ cannot compute a maximum as a reduction operation.

  Therefore, we define a private variable MY_DIFF for each thread.
  Once they have all computed their values, we use a CRITICAL section
  to update DIFF.
*/
  diff=0.0;
  my_diff=0.0;
  MPI_Scatter(w, (M/comm_sz)*N, MPI_DOUBLE, local_w, (M/comm_sz)*N,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Scatter(u, (M/comm_sz)*N, MPI_DOUBLE, local_u, (M/comm_sz)*N,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
  my_diff=fuctor4(my_rank, comm_sz, M/comm_sz, N, local_w, local_u,
my_diff);
  MPI_Reduce(&my_diff, &diff, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
  MPI_Bcast(&diff, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

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

      iterations_print = 2 * iterations_print;
    }
  MPI_Barrier(MPI_COMM_WORLD);
  }

  wtime = MPI_Wtime() - wtime;

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

  free(u);
  free(w);
  free(local_u);
  free(local_w);

  MPI_Finalize();
  return 0;

# undef M
# undef N
}
```

在 task1 中使用函数 functorx 的位置替换成 MPI 实现进程间集合通信，其中为了保证并行进程之间数据的准确性，在每一个修改部分结束前后都使用

```
MPI_Barrier(MPI_COMM_WORLD);
```

使所有进程一起启动。

最后，使用 build.bat 文件可以在 Windows 环境下运行，注意需要修改.bat 文件中的引用 MPI 的 include 文件和 bin 文件的路径。

```
@echo off
REM 设置MPI 的路径
set PATH1=D:\MPI\MPI_SDK\Lib\x64\
set PATH2=D:\MPI\MPI_SDK\Include\

REM 设置源文件和目标文件的名称
set SRC_MAIN=heated_plate_openmp.c
set EXEC_FILE=heated_plate_openmp


REM 1. 链接目标文件并生成最终可执行文件
echo Linking %SRC_MAIN% and parallel_fun.o to generate %EXEC_FILE% ...
REM gcc -g -Wall -fopenmp -o %EXEC_FILE% %SRC_MAIN%
gcc %SRC_MAIN% -o %EXEC_FILE% -fopenmp -l msmpi -L %PATH1%  -I %PATH2%
```

```
REM 2. 如果编译和链接成功，运行可执行文件
if %ERRORLEVEL% == 0 (
    echo Compilation and linking succeeded, running %EXEC_FILE% ...
    mpiexec -n 4 %EXEC_FILE%
    REM  %EXEC_FILE%
) else (
    echo Compilation or linking failed!
)

pause
```

task3:（相关文件在 task3 文件夹中）

使用 build.sh 文件可以在 Windows 环境下运行，在 task1 的基础上加上查看堆使用的空间。由于 **valgrind** 指令在 Windows 环境下不适应，因此该任务需要在 Linux 环境下运行，

```
#!/bin/bash

# 设置源文件和目标文件的名称
SRC_MAIN="heated_plate_openmp.c"
SRC_LIB="parallel_fun.c"
OBJ_LIB="parallel_fun.o"
EXEC_FILE="heated_plate_openmp"
DLL_FILE="libfunction.so"

# 1. 编译 parallel_fun.c 为目标文件
echo "Compiling $SRC_LIB to $OBJ_LIB ..."
gcc -c $SRC_LIB -o $OBJ_LIB -lpthread

# 2. 使用 -shared 参数生成动态链接库 function.so
echo "Compiling $SRC_LIB to generate $DLL_FILE ..."
gcc -shared $SRC_LIB -o $DLL_FILE -lpthread

#3. 编译 heated_plate_openmp.c 文件，生成目标文件，并链接 parallel_fun.o 和
function.so 生成最终的可执行文件
echo "Linking $SRC_MAIN and $OBJ_LIB to generate $EXEC_FILE ..."
gcc -g -Wall -fopenmp -o $EXEC_FILE $SRC_MAIN $OBJ_LIB -L. -lfunction
-lpthread

# 4. 如果编译和链接成功，运行可执行文件
if [ $? -eq 0 ]; then
```

```
    echo "Compilation and linking succeeded, running $EXEC_FILE ..."
    valgrind --tool=massif --time-unit=B --stacks=yes ./$EXEC_FILE
else
    echo "Compilation or linking failed!"
fi
```

3. 实验结果

原始 heated_plate_openmp 文件运行结果:

```
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 = 16
  Number of threads =              16

  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 = 8.435521

HEATED_PLATE_OPENMP:
  Normal end of execution.
Normal end of execution.
```

task1：

```
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-003
  Number of processors available = 16
  Number of threads =              16

  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 = 71.326000

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

相比 Openmp 框架，自己构建的 parallel_fun 函数运行结果一致，不过速度较慢，加速比仅有 0.12。

推测原因：自己构建的基于 Pthreads 的 parallel_fun 函数在每次调用时都会进行一次创建线程，执行完一次任务后就会销毁线程。在整个程序运行过程中，总共要进行 16955*3 次创建线程，因为造成了很大的开销，甚至比并行化提升的影响还大，所以运行速度降低了很多。

task2:

```
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-003
  Number of processors available = 16
  Number of processes =            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 = 37.136396

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

相比 Openmp 框架，在此基础上修改的 MPI 集合通信最后运行结果一致，速度比 Openmp 框架较慢，加速比为 0.23，接近自己构建的 parallel_fun 函数的两倍。

推测原因：使用 MPI 进行进程间的通信，在每次循环都要进行 2 次广播进行 u 和 w 整个矩阵的传递，影响了性能。

task3：

不同规模下并行化应用的执行时间

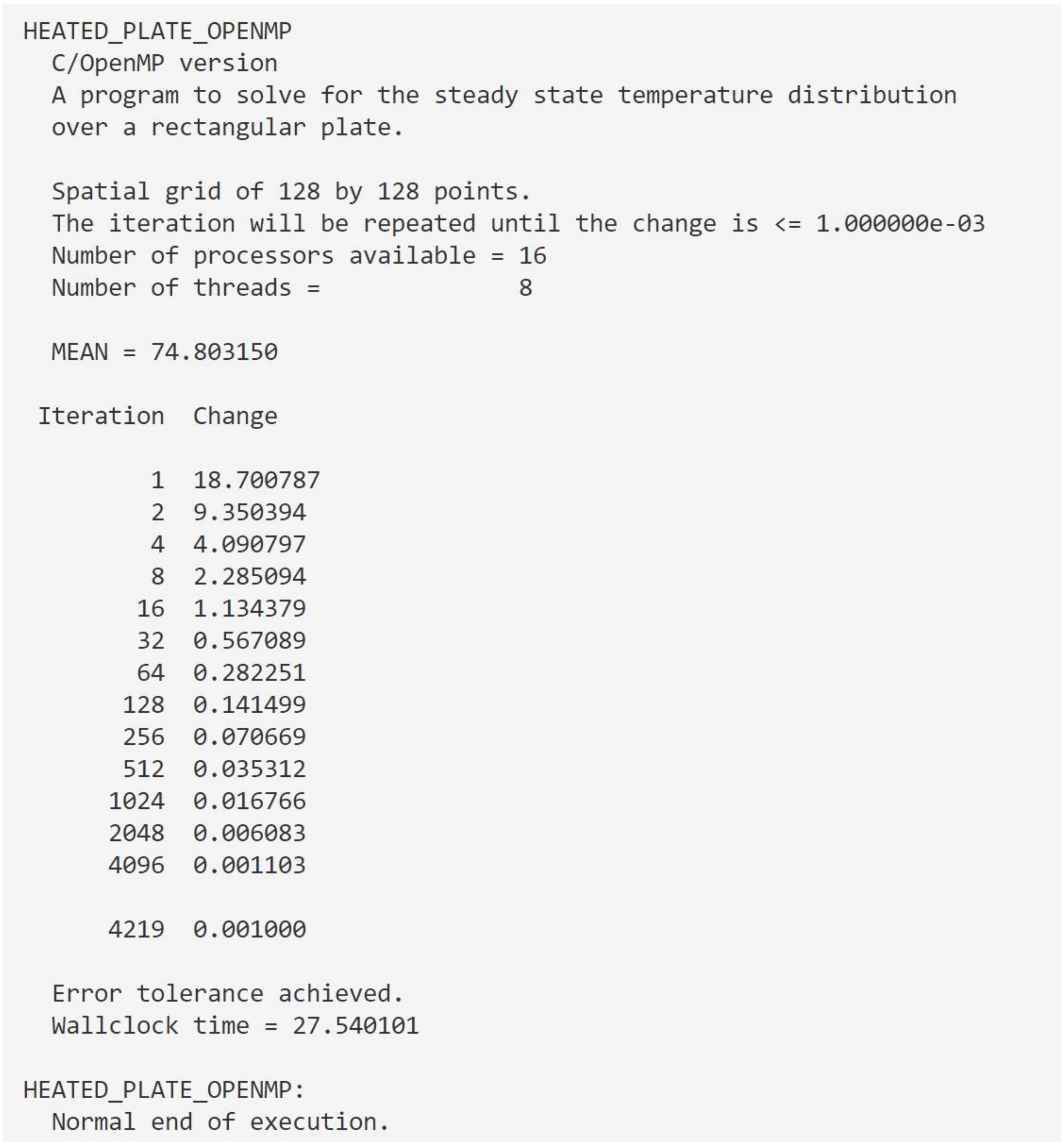
（由于规模超过 2048 后耗时过久，仅进行 8、64、512、2048 规模下的测试，以下结果基于原始的 heated_plate_openmp 文件进行测试）

Comm_size (num of threads)	Order of Matrix (Seconds)			
	16	64	512	2048
1	0.000170	0.040841	26.633184	569.175789
2	0.000610	0.025819	16.567998	299.834440
4	0.000613	0.016043	10.653334	233.593374
8	0.000885	0.016128	7.996018	188.379856
16	0.007690	0.023476	5.855366	212.488276

分析：关于 task1 中每次循环都要进行线程的创建和销毁的问题，在对比不同规模的并行化应用的执行时间时发现在同等问题规模下，使用更多线程的情况下在运行时间上要慢于只使用一个线程的（问题规模为 500*500 的情况下，使用一个线程的运行时间是 46s 左右，使用 16 个线程的运行时间是 73s 左右）。正如 task1 中的分析，频繁地创建和销毁线程会导致性能严重下降。我首先尝试将原本代码中使用的用 flag 标记循环等待来解决同步问题修改成使用 Pthreads 的 pthread_mutex_t 锁来处理同步问题，结果还是失败。最后想了一个解决思路是，在循环外创建一个线程池，每次调用 parallel_fun 函数进行计算时，从线程池中选取线程分配对应的任务。对于这种方法，思路与基于 MPI 进程通信相似，每次需要进行数据的分配，但是不需要进行频繁地创建和销毁线程，预计效率相比现有的 parallel_fun 有较大的提升。由于该方法的实现需要用到其他库函数并且涉及对 parallel_fun 函数的较大修改，不在本次实验中体现。

不同规模下的并行化应用的内存消耗对比

8 个线程，问题规模为 128 时（输出文件为 massif.out.378）



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

  Spatial grid of 128 by 128 points.
  The iteration will be repeated until the change is <= 1.000000e-03
  Number of processors available = 16
  Number of threads =              8

  MEAN = 74.803150

 Iteration  Change

         1  18.700787
         2  9.350394
         4  4.090797
         8  2.285094
        16  1.134379
        32  0.567089
        64  0.282251
       128  0.141499
       256  0.070669
       512  0.035312
      1024  0.016766
      2048  0.006083
      4096  0.001103

      4219  0.001000

  Error tolerance achieved.
  Wallclock time = 27.540101

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

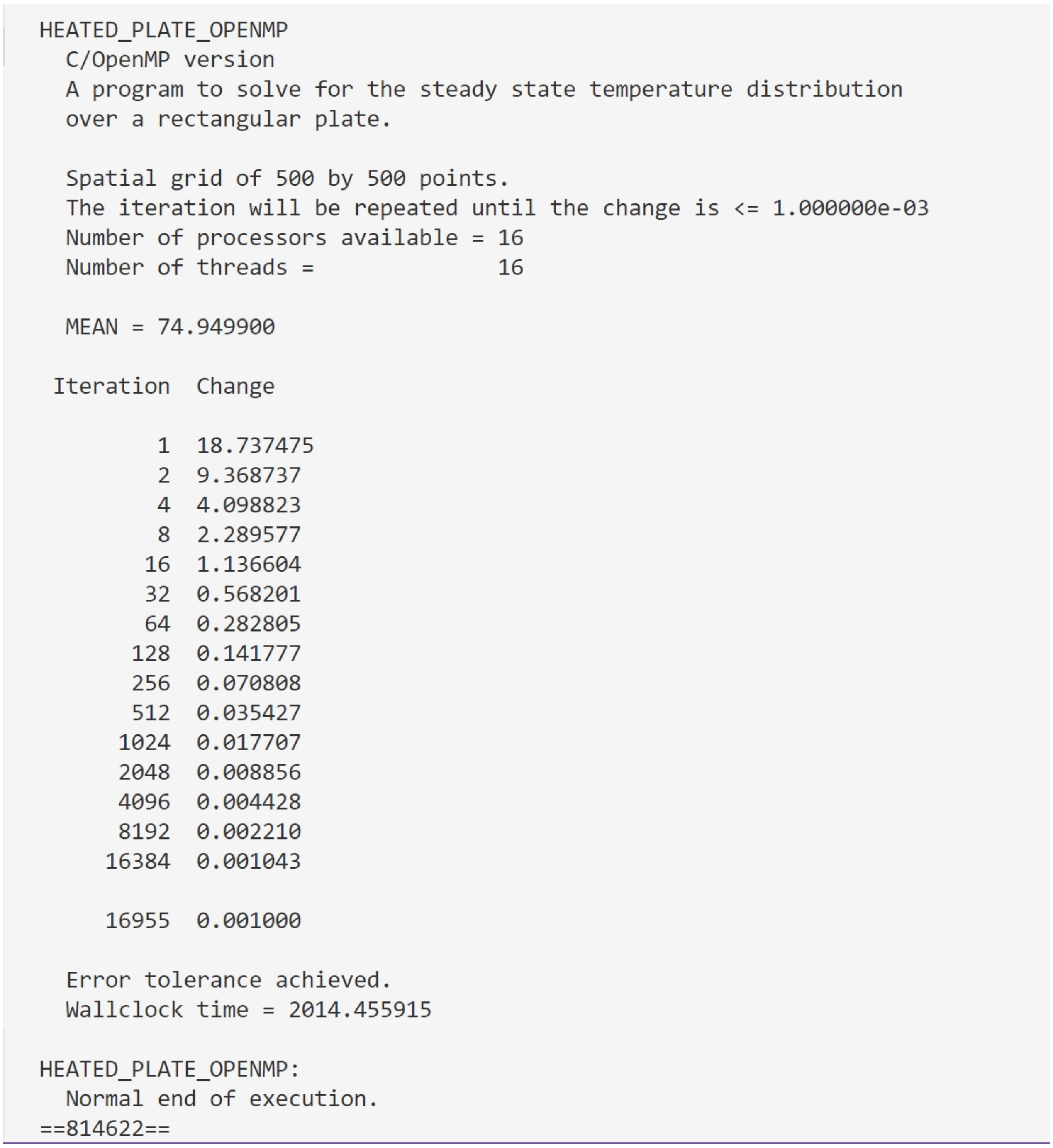
ms_print 图像

```
--------------------------------------------------------------------------------
Command:            ./heated_plate_openmp
Massif arguments:   --time-unit=B --stacks=yes
ms_print arguments: ./massif.out.378
--------------------------------------------------------------------------------


    MB
15.63^                                                                       #
     |                                                                     @@#
     |                                                                 ::::@@#
     |                                                             ::::::  @@#
     |                                                          @::: ::::  @@#
     |                                                      @@@@@::: ::::  @@#
     |                                                  @@@@@@ @@::: ::::  @@#
     |                                               :::@ @@@@ @@::: ::::  @@#
     |                                           :@::: :@ @@@@ @@::: ::::  @@#
     |                                       @@@@:@: : :@ @@@@ @@::: ::::  @@#
     |                                    @@@@ @ :@: : :@ @@@@ @@::: ::::  @@#
     |                                @:::@ @@ @ :@: : :@ @@@@ @@::: ::::  @@#
     |                            @@@@@:: @ @@ @ :@: : :@ @@@@ @@::: ::::  @@#
     |                        ::@@@@ @@:: @ @@ @ :@: : :@ @@@@ @@::: ::::  @@#
     |                    @@::: @@@@ @@:: @ @@ @ :@: : :@ @@@@ @@::: ::::  @@#
     |                 @@@@ : : @@@@ @@:: @ @@ @ :@: : :@ @@@@ @@::: ::::  @@#
     |              @@@@ @@ : : @@@@ @@:: @ @@ @ :@: : :@ @@@@ @@::: ::::  @@#
     |          @@::@ @@ @@ : : @@@@ @@:: @ @@ @ :@: : :@ @@@@ @@::: ::::  @@#
     |       @@@@ : @ @@ @@ : : @@@@ @@:: @ @@ @ :@: : :@ @@@@ @@::: ::::  @@#
     |   @@@:@@ @ : @ @@ @@ : : @@@@ @@:: @ @@ @ :@: : :@ @@@@ @@::: ::::  @@#
   0 +----------------------------------------------------------------------->GB
     0                                                                   1.102

Number of snapshots: 54
 Detailed snapshots: [2, 3, 4, 6, 7, 8, 10, 11, 12, 13, 14, 17, 18, 19, 20, 21, 22, 25, 26, 27,
 28, 30, 34, 35, 36, 37, 38, 39, 40, 48, 50, 51, 52, 53 (peak)]
```

16 个线程，问题规模为 500 时（输出文件为 massif.out.814622）



```
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 = 16
  Number of threads =              16

  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 = 2014.455915

HEATED_PLATE_OPENMP:
  Normal end of execution.
==814622==
```

ms_print 图像

```
--------------------------------------------------------------------------------
Command:            ./heated_plate_openmp
Massif arguments:   --time-unit=B
ms_print arguments: massif.out.814622
--------------------------------------------------------------------------------


    MB
3.846^#
     |#:::::@::::@:::::::::@@::@::::::::::@:::::::::::::::@@:::::::@@::::::::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
     |#::: :@::: @: :: : : @ ::@: :::::: :@:::  :::: :: : @ : :: : @ :: : :::@
   0 +----------------------------------------------------------------------->MB
     0                                                                   449.3

Number of snapshots: 54
 Detailed snapshots: [1 (peak), 6, 10, 16, 19, 28, 40, 45, 53]
```

4. 实验感想

在这次实验中，我进一步了解了 Openmp 的使用，并且在使用 Pthread 和 MPI 进行改造时，感受到了 Openmp 的遍历性，不需要我们自己进行任务分配即可带来较优的性能。同时，在实验过程中，我也遇到了问题，刚开始搞不清楚为什么会出现线程增多却性能下降的线程，经过向助教和同学请教后，我初步了解了问题，并且有一定的解决思路。另外，这次实验的强制环境要求让我给自己的电脑配置了 WSL，我第一次感受到在 Windows 下使用 Linux 环境的便利。