《并行计算》上机报告

姓名:	范文	学号:	PB18111679	日期:	2021.5.22
上机题目:	MPI 并行编程实验				

实验环境:

CPU: 8 个 Intel(R) Core(TM) i5-8250U

内存: 8G（交换区 8G ）

操作系统: ubuntu 20.04（内核版本为 5.4.0-72-generic）

软件平台: VSCode,

一、算法设计与分析:

题目一:

根据 PPT 上的内容，可以在求和近似积分的基础上，设计基于消息传递的并行求 π 的算法:

(1) 非 0 号线程执行局部的求和，并将其计算结果传递给 0 号线程。

(2) 0 号线程将这些结果进行累加，即得到 π 的近似值。

相比于 OpenMP 中共享存储的方法进行局部结果的累加，mpi 通过消息传递的方式需要考虑更多的开销，因此更慢一些。

题目二:

根据教材上的内容，可以把 PSRS 分为以下 8 个阶段进行:

(1) 均匀划分: 将 n 个元素 A[1..n]均匀划分成 p 段,

(2) 局部排序: pi 调用串行排序算法对 A[(i-1)n/p+1..in/p]排序

(3) 选取样本: pi 从其有序子序列 A[(i-1)n/p+1..in/p]中选取 p 个样本元素

(4) 样本排序: 用一台处理器对 p^2 个样本元素进行串行排序

(5) 选择主元: 用一台处理器从排好序的样本序列中选取 p-1 个主元,并

播送给其他 pi

(6) 主元划分: pi 按主元将有序段 A[(i-1)n/p+1..in/p]划分成 p 段

(7) 全局交换: 各处理器将其有序段按段号交换到对应的处理器中

(8) 归并排序: 各处理器对接收到的元素进行归并排序

分析以上过程可知，(1)(4)(5)(7)(8)可以在一个线程上执行，(2)(3)(6)可以多个线程并行执行。而不同步骤之间需要进行一到多或者多到一的消息传递。

如果不考虑通信的开销，时间复杂度的分析和 OpenMP 的类似:

(1) 均匀划分需要 O(p)的复杂度。

(2) 局部排序需要 O(n/p lg(n/p))的复杂度。

(3) 选取样本需要 O(p)的复杂度。

(4) 样本排序需要 O(p^2lg p)的复杂度。

(5) 选择主元需要 O(p)的复杂度。

(6) 主元划分需要 O(p)的复杂度。

(7) 全局交换需要 O(p^2)的复杂度。

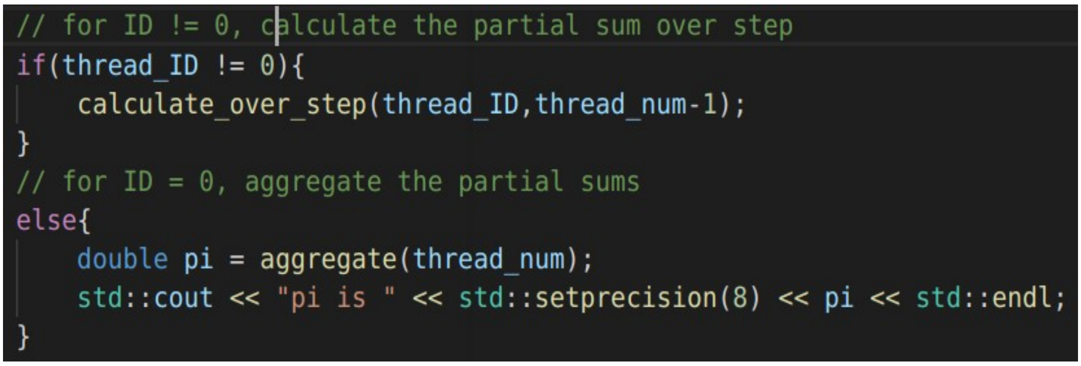
(8) 归并排序需要 O(n/p lg(n/p))的复杂度。

因此，不考虑通信开销，基于 mpi 的 PSRS 算法的复杂度为 O(n/p lg(n/p) + p^2lg p);但是，通信开销会使这里的复杂度继续增加。

二、核心代码:

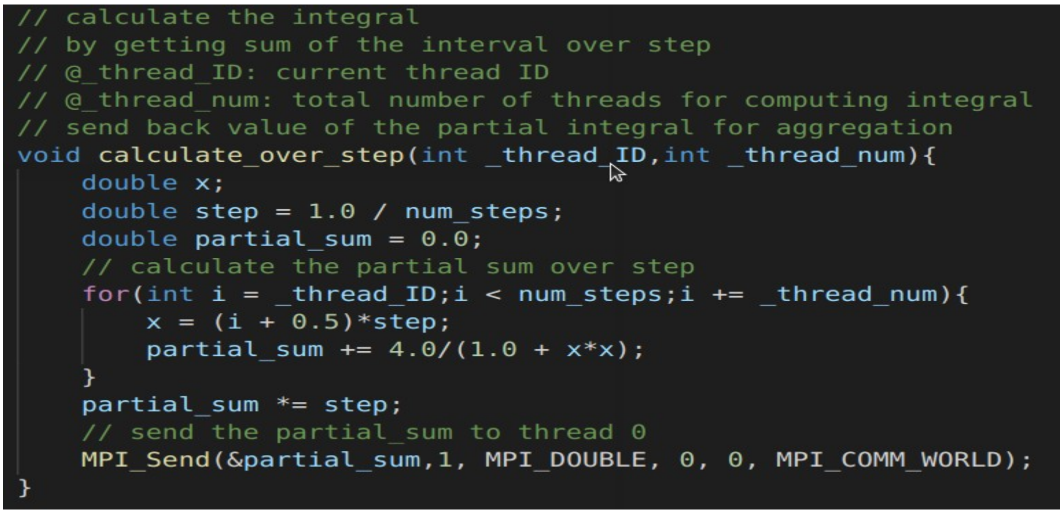
题目一:

(1) 在 main 函数中，控制不同线程号的线程执行不同的操作。



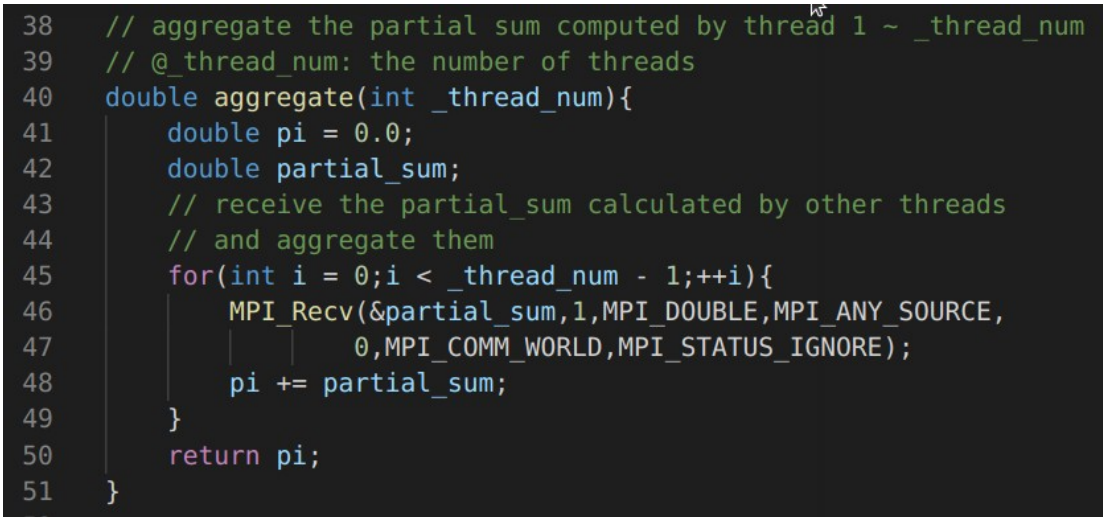
```
// for ID != 0, calculate the partial sum over step
if(thread_ID != 0){
    calculate_over_step(thread_ID,thread_num-1);
}
// for ID = 0, aggregate the partial sums
else{
    double pi = aggregate(thread_num);
    std::cout << "pi is " << std::setprecision(8) << pi << std::endl;
}
```

(2) 在非 0 号线程执行的 calculate_over_step()函数中，先进行循环的局部求和，再把求和的结果通过 MPI_Send()发给 0 号线程。



```
// calculate the integral
// by getting sum of the interval over step
// @_thread_ID: current thread ID
// @_thread_num: total number of threads for computing integral
// send back value of the partial integral for aggregation
void calculate_over_step(int _thread_ID,int _thread_num){
    double x;
    double step = 1.0 / num_steps;
    double partial_sum = 0.0;
    // calculate the partial sum over step
    for(int i = _thread_ID;i < num_steps;i += _thread_num){
        x = (i + 0.5)*step;
        partial_sum += 4.0/(1.0 + x*x);
    }
    partial_sum *= step;
    // send the partial_sum to thread 0
    MPI_Send(&partial_sum,1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}
```

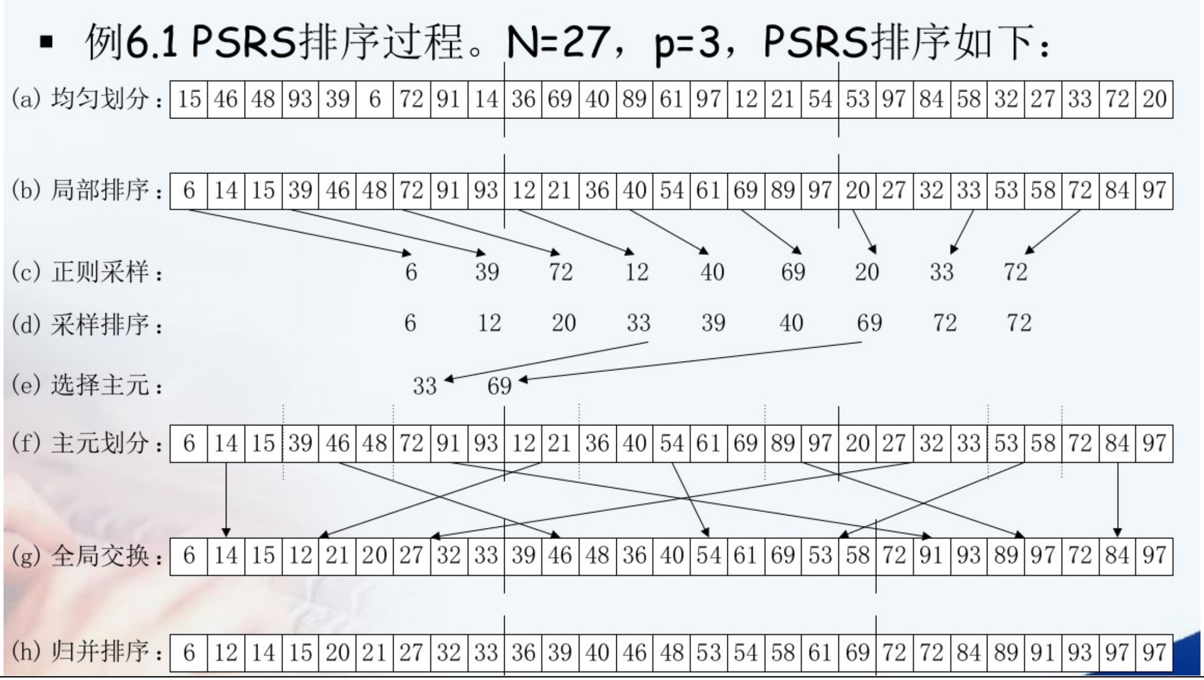
(3) 在 0 号线程执行的 aggregate()函数中,通过 MPI_Recv() 接收每个非 0 线程的求和的结果，并将这些局部的结果进行累加。



```
// aggregate the partial sum computed by thread 1 ~ _thread_num
// @_thread_num: the number of threads
double aggregate(int _thread_num){
    double pi = 0.0;
    double partial_sum;
    // receive the partial_sum calculated by other threads
    // and aggregate them
    for(int i = 0;i < _thread_num - 1;++i){
        MPI_Recv(&partial_sum,1,MPI_DOUBLE,MPI_ANY_SOURCE,
                0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        pi += partial_sum;
    }
    return pi;
}
```

题目二:

PSRS 算法的例子如下所示



- 例6.1 PSRS排序过程。N=27，p=3，PSRS排序如下：

(a) 均匀划分：15 46 48 93 39 6 72 91 14 | 36 69 40 89 61 97 12 21 54 | 53 97 84 58 32 27 33 72 20

(b) 局部排序：6 14 15 39 46 48 72 91 93 | 12 21 36 40 54 61 69 89 97 | 20 27 32 33 53 58 72 84 97

(c) 正则采样：6 39 72 12 40 69 20 33 72

(d) 采样排序：6 12 20 33 39 40 69 72 72

(e) 选择主元：33 69

(f) 主元划分：6 14 15 | 39 46 48 72 91 93 | 12 21 | 36 40 54 61 69 89 97 | 20 27 32 33 | 53 58 72 84 97

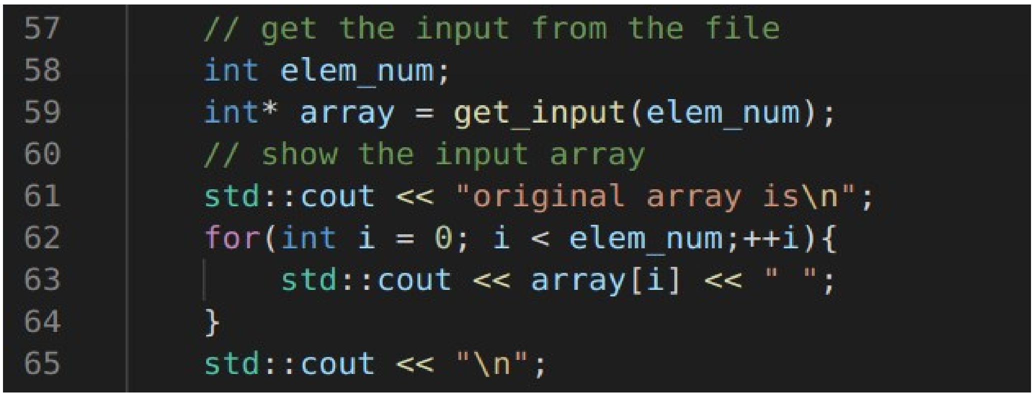
(g) 全局交换：6 14 15 12 21 20 27 32 33 | 39 46 48 36 40 54 61 69 53 58 | 72 91 93 89 97 72 84 97

(h) 归并排序：6 12 14 15 20 21 27 32 33 | 36 39 40 46 48 53 54 58 61 69 | 72 72 84 89 91 93 97 97

由此可知，(1)均匀划分、(4)采样排序、(5)选择主元、(7)全局交换、(8)归并排序都是在同一个线程里完成的。而(2)局部排序、(3)正则采样、(6)主元划分都是多线程并行完成

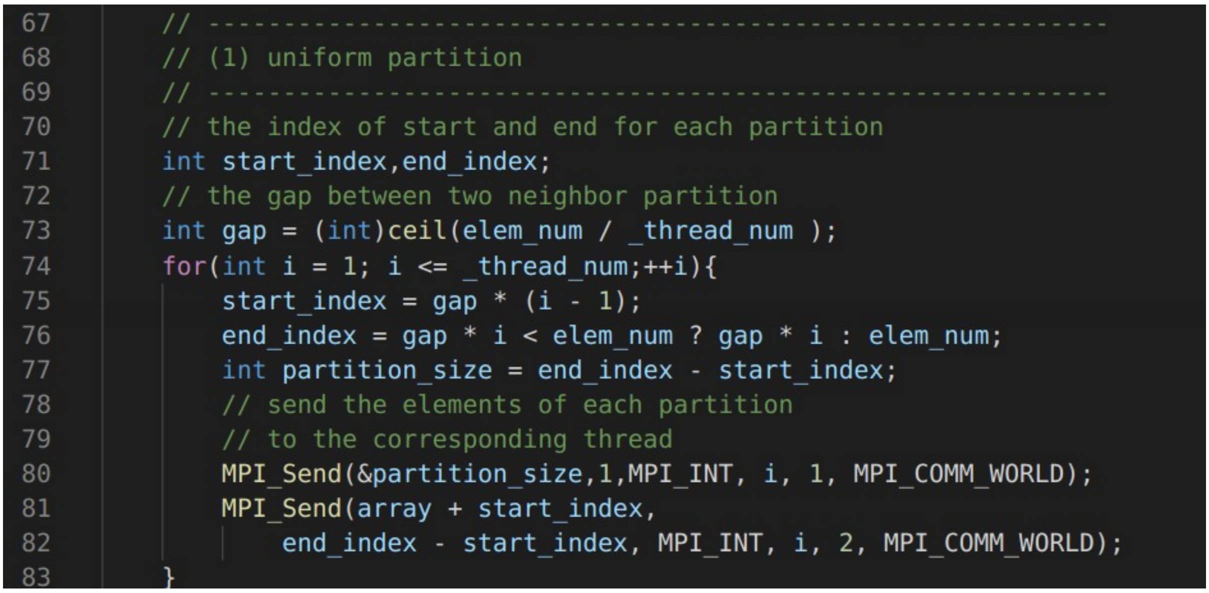
的。因此，我令 0 号线程执行 serial()串行函数，令非 0 号线程执行 parallel()并行函数，完成以上操作；而核心代码如下：

一．0 号线程读取输入。



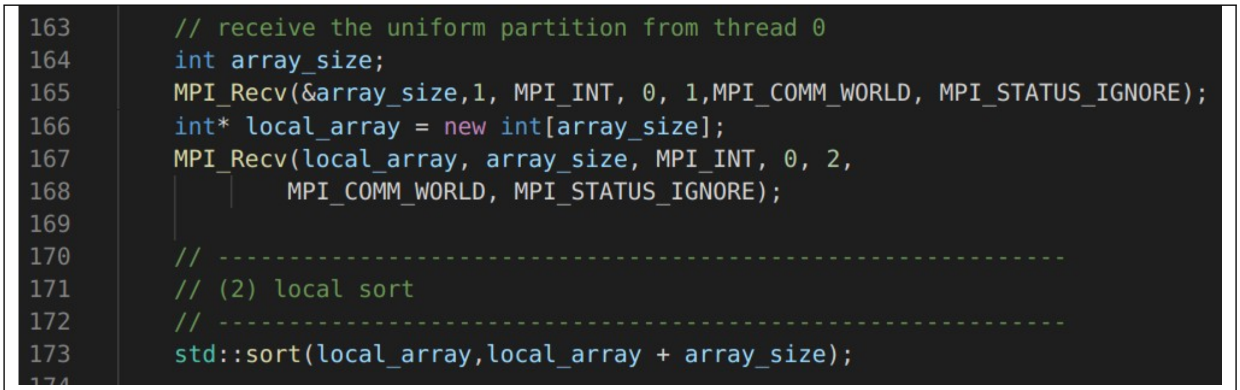
```
// get the input from the file
int elem_num;
int* array = get_input(elem_num);
// show the input array
std::cout << "original array is\n";
for(int i = 0; i < elem_num;++i){
    std::cout << array[i] << " ";
}
std::cout << "\n";
```

二．0 号线程对输入元素进



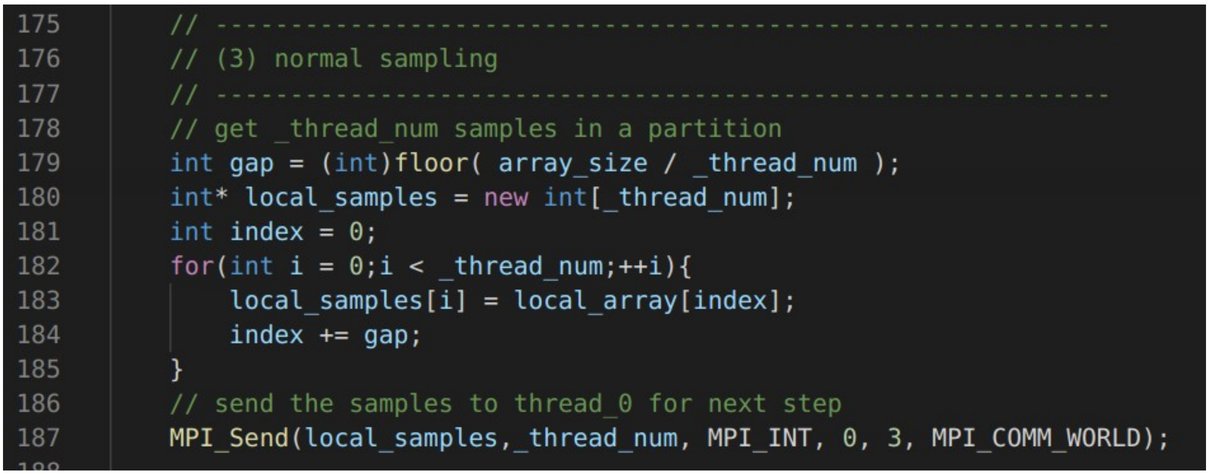
```
// ------------------------------------------------------------
// (1) uniform partition
// ------------------------------------------------------------
// the index of start and end for each partition
int start_index,end_index;
// the gap between two neighbor partition
int gap = (int)ceil(elem_num / _thread_num );
for(int i = 1; i <= _thread_num;++i){
    start_index = gap * (i - 1);
    end_index = gap * i < elem_num ? gap * i : elem_num;
    int partition_size = end_index - start_index;
    // send the elements of each partition
    // to the corresponding thread
    MPI_Send(&partition_size,1,MPI_INT, i, 1, MPI_COMM_WORLD);
    MPI_Send(array + start_index,
        end_index - start_index, MPI_INT, i, 2, MPI_COMM_WORLD);
}
```

三．每个非 0 号线程都得到得到它们对应的划分，并对它们进行排序。



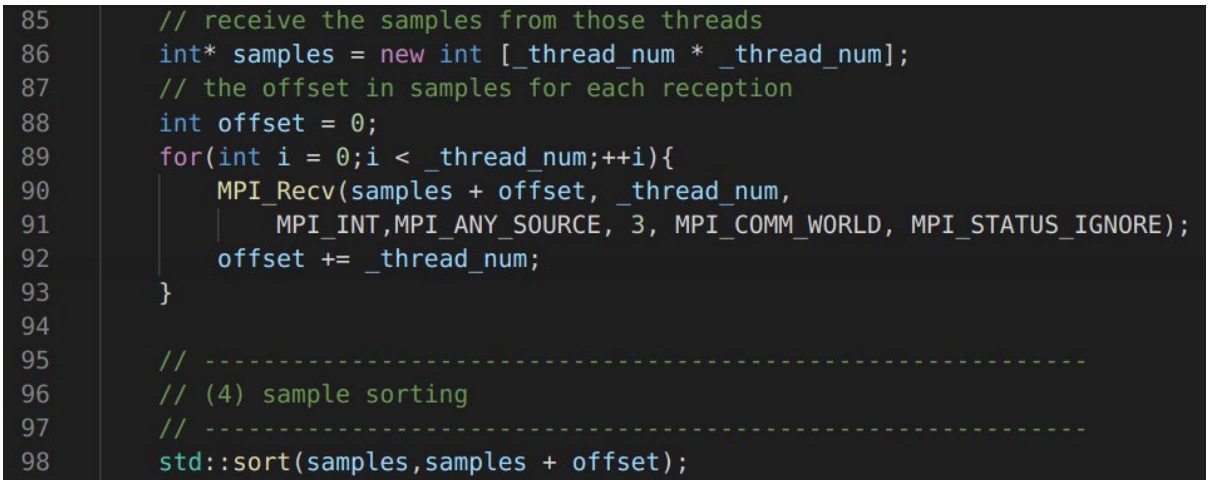
```
    // receive the uniform partition from thread 0
    int array_size;
    MPI_Recv(&array_size,1, MPI_INT, 0, 1,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    int* local_array = new int[array_size];
    MPI_Recv(local_array, array_size, MPI_INT, 0, 2,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // ------------------------------------------------------------
    // (2) local sort
    // ------------------------------------------------------------
    std::sort(local_array,local_array + array_size);
```

四．每一个非 0 号线程对划分进行采样，并传递给 0 号线程。



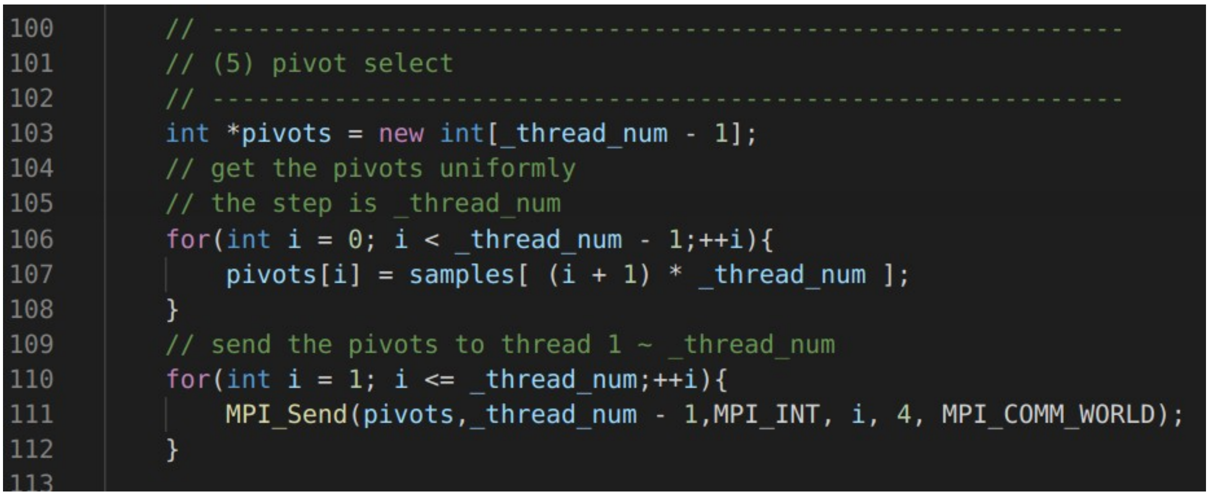
```
    // ------------------------------------------------------------
    // (3) normal sampling
    // ------------------------------------------------------------
    // get _thread_num samples in a partition
    int gap = (int)floor( array_size / _thread_num );
    int* local_samples = new int[_thread_num];
    int index = 0;
    for(int i = 0;i < _thread_num;++i){
        local_samples[i] = local_array[index];
        index += gap;
    }
    // send the samples to thread_0 for next step
    MPI_Send(local_samples,_thread_num, MPI_INT, 0, 3, MPI_COMM_WORLD);
```

五．0 号线程收集采样，并将采样进行排序



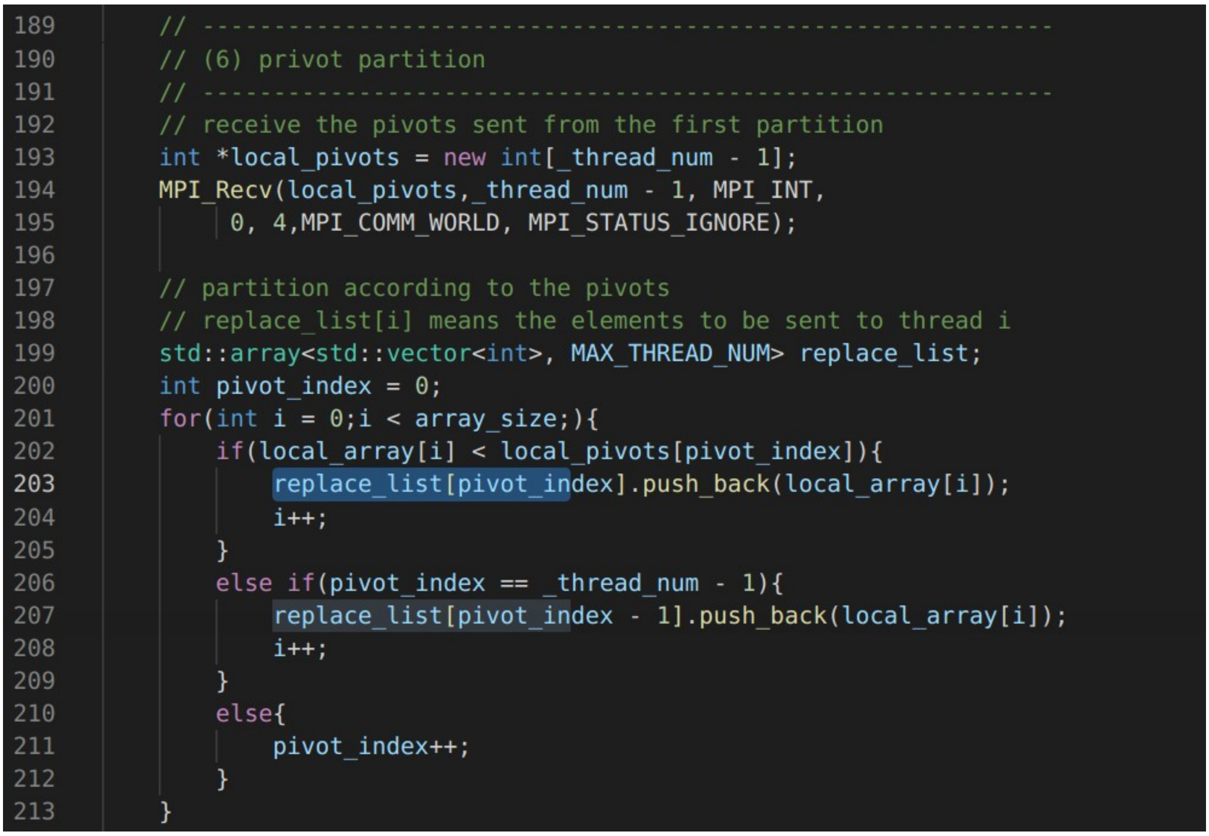
```
    // receive the samples from those threads
    int* samples = new int [_thread_num * _thread_num];
    // the offset in samples for each reception
    int offset = 0;
    for(int i = 0;i < _thread_num;++i){
        MPI_Recv(samples + offset, _thread_num,
            MPI_INT,MPI_ANY_SOURCE, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        offset += _thread_num;
    }

    // ------------------------------------------------------------
    // (4) sample sorting
    // ------------------------------------------------------------
    std::sort(samples,samples + offset);
```

六．0 号线程选择主元，并将主元传递给每个非 0 号线程。



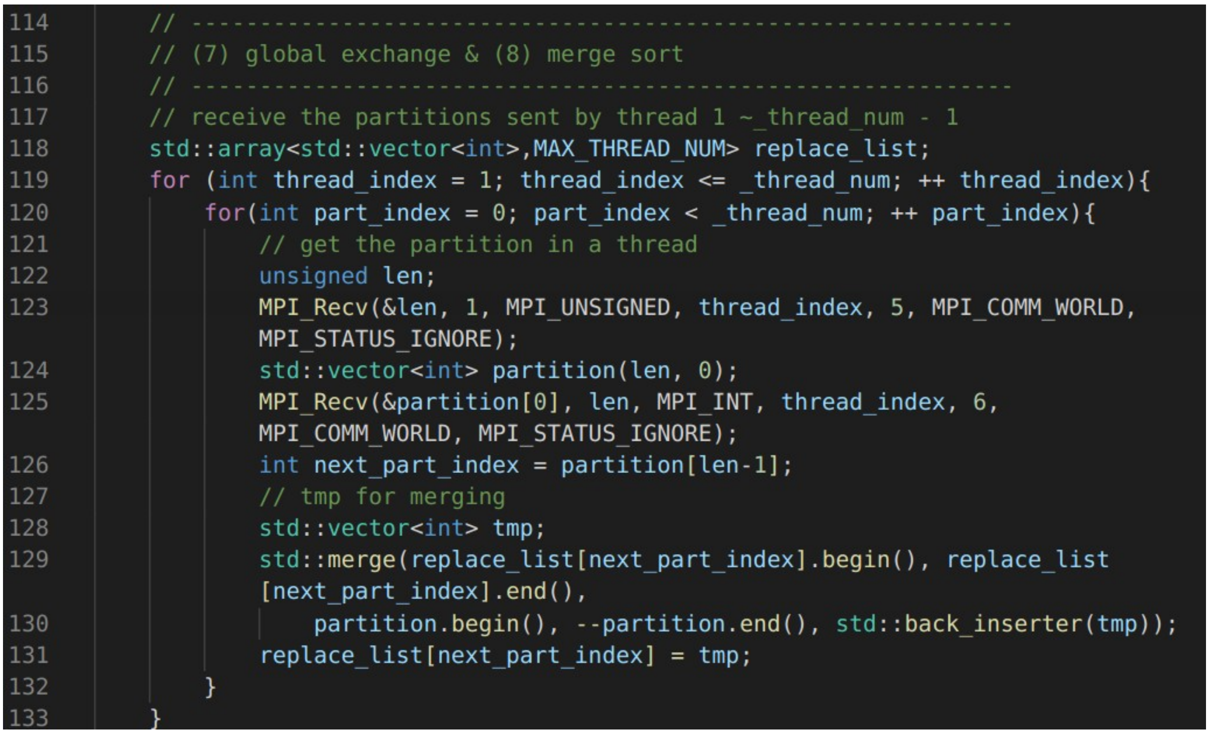
```
    // ------------------------------------------------------------
    // (5) pivot select
    // ------------------------------------------------------------
    int *pivots = new int[_thread_num - 1];
    // get the pivots uniformly
    // the step is _thread_num
    for(int i = 0; i < _thread_num - 1;++i){
        pivots[i] = samples[ (i + 1) * _thread_num ];
    }
    // send the pivots to thread 1 ~ _thread_num
    for(int i = 1; i <= _thread_num;++i){
        MPI_Send(pivots,_thread_num - 1,MPI_INT, i, 4, MPI_COMM_WORLD);
    }
```

七．每一个非 0 号线程接收主元，并根据主元再次进行划分，用一个二维数组存放，并将这些划分传递给 0 号线程。



```
    // ------------------------------------------------------------
    // (6) privot partition
    // ------------------------------------------------------------
    // receive the pivots sent from the first partition
    int *local_pivots = new int[_thread_num - 1];
    MPI_Recv(local_pivots,_thread_num - 1, MPI_INT,
        0, 4,MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // partition according to the pivots
    // replace_list[i] means the elements to be sent to thread i
    std::array<std::vector<int>, MAX_THREAD_NUM> replace_list;
    int pivot_index = 0;
    for(int i = 0;i < array_size;){
        if(local_array[i] < local_pivots[pivot_index]){
            replace_list[pivot_index].push_back(local_array[i]);
            i++;
        }
        else if(pivot_index == _thread_num - 1){
            replace_list[pivot_index - 1].push_back(local_array[i]);
            i++;
        }
        else{
            pivot_index++;
        }
    }
```

八．0 号线程接收这些进一步的划分，并对他们进行归并排序，得到了最终有序的数组。

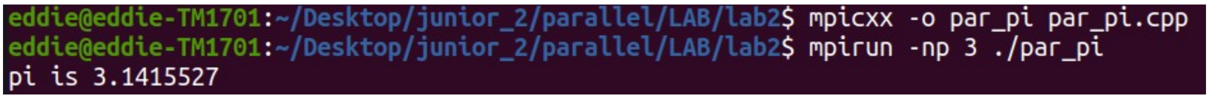


```
    // -----------------------------------------------------------
    // (7) global exchange & (8) merge sort
    // -----------------------------------------------------------
    // receive the partitions sent by thread 1 ~ _thread_num - 1
    std::array<std::vector<int>,MAX_THREAD_NUM> replace_list;
    for (int thread_index = 1; thread_index <= _thread_num; ++ thread_index){
        for(int part_index = 0; part_index < _thread_num; ++ part_index){
            // get the partition in a thread
            unsigned len;
            MPI_Recv(&len, 1, MPI_UNSIGNED, thread_index, 5, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            std::vector<int> partition(len, 0);
            MPI_Recv(&partition[0], len, MPI_INT, thread_index, 6, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            int next_part_index = partition[len-1];
            // tmp for merging
            std::vector<int> tmp;
            std::merge(replace_list[next_part_index].begin(), replace_list[next_part_index].end(),
                partition.begin(), --partition.end(), std::back_inserter(tmp));
            replace_list[next_part_index] = tmp;
        }
    }
```

三、结果与分析：

题目一:

我编译了程序之后，使用 3 个线程进行执行（实际上只有两个线程进行局部求和），得到了结果如下:

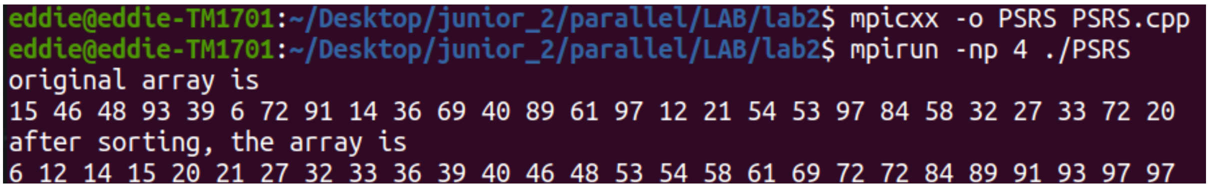


```
eddie@eddie-TM1701:~/Desktop/junior_2/parallel/LAB/lab2$ mpicxx -o par_pi par_pi.cpp
eddie@eddie-TM1701:~/Desktop/junior_2/parallel/LAB/lab2$ mpirun -np 3 ./par_pi
pi is 3.1415527
```

这里的结果精确到了小数点后 4 位。

题目二:

使用 PPT 上的例子，开 4 个线程执行，得到测试结果如下:



```
eddie@eddie-TM1701:~/Desktop/junior_2/parallel/LAB/lab2$ mpicxx -o PSRS PSRS.cpp
eddie@eddie-TM1701:~/Desktop/junior_2/parallel/LAB/lab2$ mpirun -np 4 ./PSRS
original array is
15 46 48 93 39 6 72 91 14 36 69 40 89 61 97 12 21 54 53 97 84 58 32 27 33 72 20
after sorting, the array is
6 12 14 15 20 21 27 32 33 36 39 40 46 48 53 54 58 61 69 72 72 84 89 91 93 97 97
```

这说明可以正确排序。

四、备注（* 可选）：

有可能影响结论的因素:

1. 计算精度会导致估计 π 的结果并不是很准确

2. mpi 中的线程的消息传递和并行调度会产生开销，导致计算并不是很快。（也可能是输入的数据量较小，没有明显体现出并行编程的优越性出来）

总结:

本次实验，我学习了使用 mpi 进行并行编程，了解了基于消息传递的并行编程的思想，并提高了自己的编码和调试能力。

附录（源代	算法源代码（C/C++/JAVA 描述）

码）

我的本次实验的源码保存在了 https://github.com/fanweneddie/parallel_computing/tree/master/lab2 中。

题目 1：

// par_pi.c

```
/*
* using MPI to calculate the approximate value of Pi
* by computing parallelism integral over step
*
* I have borrowed some ideas from myl7's github
* https://github.com/myl7/paracomp2021
*
* PB18111679 fanweneddie
*/
#include <iostream>
#include <iomanip>
#include <mpi.h>

static long num_steps = 100000;

void calculate_over_step(int _thread_ID,int _thread_num);
double aggregate(int _thread_num);

// calculate the integral
// by getting sum of the interval over step
// @_thread_ID: current thread ID
// @_thread_num: total number of threads for computing integral
// send back value of the partial integral for aggregation
void calculate_over_step(int _thread_ID,int _thread_num){
double x;
double step = 1.0 / num_steps;
double partial_sum = 0.0;
// calculate the partial sum over step
for(int i = _thread_ID;i < num_steps;i += _thread_num){
x = (i + 0.5)*step;
partial_sum += 4.0/(1.0 + x*x);
}
partial_sum *= step;
// send the partial_sum to thread 0
MPI_Send(&partial_sum,1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}

// aggregate the partial sum computed by thread 1 ~ _thread_num - 1
// @_thread_num: the number of threads
double aggregate(int _thread_num){
double pi = 0.0;
double partial_sum;
// receive the partial_sum calculated by other threads
// and aggregate them
for(int i = 0;i < _thread_num - 1;++i){
MPI_Recv(&partial_sum,1,MPI_DOUBLE,MPI_ANY_SOURCE,
0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
pi += partial_sum;
}
return pi;
}


int main(int argc, const char *const argv[]) {

MPI_Init(&argc, const_cast<char ***>(&argv));
// the thread ID of the current thread
int thread_ID;
MPI_Comm_rank(MPI_COMM_WORLD, &thread_ID);
// the total number of threads
int thread_num;
MPI_Comm_size(MPI_COMM_WORLD, &thread_num);

// for ID != 0, calculate the partial sum over step
if(thread_ID != 0){
calculate_over_step(thread_ID,thread_num-1);
}
// for ID = 0, aggregate the partial sums
else{
double pi = aggregate(thread_num);
std::cout << "pi is " << std::setprecision(8) << pi << std::endl;
}
MPI_Finalize();
return 0;
}
```

题目二:

```
/*
* use mpi to implement PSRS sorting algorithm
*
* the process of PSRS
* comes from our course slide
*
* I have borrowed some ideas from myl7's github
* https://github.com/myl7/paracomp2021
*
* PB18111679 fanweneddie
*/

#include <fstream>
#include <iostream>
#include <array>
#include <vector>
#include <math.h>
#include <algorithm>
#include <mpi.h>

constexpr auto MAX_THREAD_NUM = 10;
void serial(int _thread_num);
void parallel(int _thread_ID,int _thread_num);
int* get_input();

// get the input int array from the input file
// allocate the array and return it
// @_elem_num: the number of elements in the array
// we need to revise it in the function
int* get_input(int& _elem_num){
std::ios::sync_with_stdio(false);
std::ifstream in("input.txt");
if (!in) {
std::cerr << "Getting input failed\n";
}
in >> _elem_num;
int *array = new int[_elem_num];
for (int i = 0; i < _elem_num; i++) {
in >> array[i];
}
in.close();
```

```
return array;
}

// serial working part
// for thread 0
// do the serial work in PSRS,
// that is, to do
// (1) uniform partition
// (4) sample sorting
// (5) pivot select
// (7) global exchange
// (8) merge sort
// @_thread_num: number of parallel threads
void serial(int _thread_num){

// get the input from the file
int elem_num;
int* array = get_input(elem_num);
// show the input array
std::cout << "original array is\n";
for(int i = 0; i < elem_num;++i){
std::cout << array[i] << " ";
}
std::cout << "\n";

// ------------------------------------------------------------
// (1) uniform partition
// ------------------------------------------------------------
// the index of start and end for each partition
int start_index,end_index;
// the gap between two neighbor partition
int gap = (int)ceil(elem_num / _thread_num );
for(int i = 1; i <= _thread_num;++i){
start_index = gap * (i - 1);
end_index = gap * i < elem_num ? gap * i : elem_num;
int partition_size = end_index - start_index;
// send the elements of each partition
// to the corresponding thread
MPI_Send(&partition_size,1,MPI_INT, i, 1, MPI_COMM_WORLD);
MPI_Send(array + start_index,
end_index - start_index, MPI_INT, i, 2, MPI_COMM_WORLD);
}
```

```
// receive the samples from those threads
int* samples = new int [_thread_num * _thread_num];
// the offset in samples for each reception
int offset = 0;
for(int i = 0;i < _thread_num;++i){
MPI_Recv(samples + offset, _thread_num,
MPI_INT,MPI_ANY_SOURCE, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
offset += _thread_num;
}

// ------------------------------------------------------------
// (4) sample sorting
// ------------------------------------------------------------
std::sort(samples,samples + offset);

// ------------------------------------------------------------
// (5) pivot select
// ------------------------------------------------------------
int *pivots = new int[_thread_num - 1];
// get the pivots uniformly
// the step is _thread_num
for(int i = 0; i < _thread_num - 1;++i){
pivots[i] = samples[ (i + 1) * _thread_num ];
}
// send the pivots to thread 1 ~ _thread_num
for(int i = 1; i <= _thread_num;++i){
MPI_Send(pivots,_thread_num - 1,MPI_INT, i, 4, MPI_COMM_WORLD);
}

// ------------------------------------------------------------
// (7) global exchange & (8) merge sort
// ------------------------------------------------------------
// receive the partitions sent by thread 1 ~ _thread_num - 1
std::array<std::vector<int>,MAX_THREAD_NUM> replace_list;
for (int thread_index = 1; thread_index <= _thread_num; ++ thread_index){
for(int part_index = 0; part_index < _thread_num; ++ part_index){
// get the partition in a thread
unsigned len;
MPI_Recv(&len, 1, MPI_UNSIGNED, thread_index, 5, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
std::vector<int> partition(len, 0);
MPI_Recv(&partition[0], len, MPI_INT, thread_index, 6, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
int next_part_index = partition[len-1];
// tmp for merging
std::vector<int> tmp;
std::merge(replace_list[next_part_index].begin(), replace_list[next_part_index].end(),
partition.begin(), --partition.end(), std::back_inserter(tmp));
replace_list[next_part_index] = tmp;
}
}
// new array to store the sorted elements
std::vector<int> new_array;
// concat the replace_lists to get the new_array
for(int i = 0; i < _thread_num;++i){
new_array.insert(new_array.end(),replace_list[i].begin(),replace_list[i].end());
}

std::cout << "after sorting, the array is\n";
for (auto elem : new_array) {
std::cout << elem << " ";
}
std::cout << std::endl;

delete []array;
delete []samples;
delete []pivots;
}

// parallel working part
// for thread 1 ~ _thread_num - 1
// do the parallel work in PSRS
// that is, to do
// (2) local sort
// (3) uniform sampling
// (6) privot partition
void parallel(int _thread_ID,int _thread_num){
// receive the uniform partition from thread 0
int array_size;
MPI_Recv(&array_size,1, MPI_INT, 0, 1,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
int* local_array = new int[array_size];
MPI_Recv(local_array, array_size, MPI_INT, 0, 2,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

// ----------------------------------------------------------------
// (2) local sort
// ----------------------------------------------------------------
std::sort(local_array,local_array + array_size);
```

```
// -------------------------------------------------------------
// (3) normal sampling
// -------------------------------------------------------------
// get _thread_num samples in a partition
int gap = (int)floor( array_size / _thread_num );
int* local_samples = new int[_thread_num];
int index = 0;
for(int i = 0;i < _thread_num;++i){
local_samples[i] = local_array[index];
index += gap;
}
// send the samples to thread_0 for next step
MPI_Send(local_samples,_thread_num, MPI_INT, 0, 3, MPI_COMM_WORLD);

// -------------------------------------------------------------
// (6) privot partition
// -------------------------------------------------------------
// receive the pivots sent from the first partition
int *local_pivots = new int[_thread_num - 1];
MPI_Recv(local_pivots,_thread_num - 1, MPI_INT,
0, 4,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
// partition according to the pivots
// replace_list[i] means the elements to be sent to thread i
std::array<std::vector<int>, MAX_THREAD_NUM> replace_list;
int pivot_index = 0;
for(int i = 0;i < array_size;){
if(local_array[i] < local_pivots[pivot_index]){
replace_list[pivot_index].push_back(local_array[i]);
i++;
}
else if(pivot_index == _thread_num - 1){
replace_list[pivot_index - 1].push_back(local_array[i]);
i++;
}
else{
pivot_index++;
}
}
// send the result of partition to thread 0
for(int i = 0;i < _thread_num;++i){
int len = (unsigned)replace_list[i].size();
// marks the object of this partition.
replace_list[i].push_back(i);
len++;
MPI_Send(&len, 1, MPI_UNSIGNED, 0, 5, MPI_COMM_WORLD);
MPI_Send(&replace_list[i][0], len, MPI_INT, 0, 6, MPI_COMM_WORLD);
}

delete [] local_array;
delete [] local_samples;
delete [] local_pivots;
}

int main(int argc, const char *const argv[]) {
MPI_Init(&argc, const_cast<char ***>(&argv));
// the thread ID of the current thread
int thread_ID;
MPI_Comm_rank(MPI_COMM_WORLD, &thread_ID);
// the total number of threads
int thread_num;
MPI_Comm_size(MPI_COMM_WORLD, &thread_num);
thread_num = 4;
// for ID = 0, do the serial work
if(thread_ID == 0){
serial(thread_num - 1);
}
// for ID != 0, do parallel work
else{
parallel(thread_ID,thread_num - 1);
}
MPI_Finalize();
return 0;
}
```