《并行计算》上机报告

姓名:	范文	学号:	PB18111679	日期:	2021.5.15
上机题目:	OpenMP 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 4 中并行求 π 的算法:

1. 不同的线程交叉进行求和，在主线程上串行地把每个线程算出来的和相加。
2. 不同的线程分块进行求和，在主线程上串行地把每个线程算出来的和相加。
3. 不同的线程交叉进行求和，并且在临界区中把每个线程算出来的和相加。
4. 不同的线程分块进行求和，并使用加归约把每个线程算出来的和相加

事实上，如果不考虑计算误差，以上三种计算结果应该是一样的；另外由于调度分配开销，使用分块求和应该慢于交叉求和速度。由于竞争的原因，临界区或者加归约应该要比串行相加要慢。

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) 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，PSRS 算法的复杂度为 $O(n/p \lg(n/p) + p^2 \lg p)$

二、核心代码:

题目一:

(1) 对于第一种方法，可以开一个

```
#pragma omp parallel private(i)
```

来进行并行，而且在并行域中，令

```
i = i + NUM_THREADS
```

来进行交叉求和。

因此，核心代码为:

```
    // set 2 threads
    omp_set_num_threads(NUM_THREADS);
    // the start of parallelism domain for each thread
    #pragma omp parallel private(i)
    {
        double x;
        int id;
        id = omp_get_thread_num();
        // integral by getting sum of the interval
        for (i = id, sum[id] = 0.0;i < num_steps; i = i + NUM_THREADS){
            x = (i + 0.5)*step;
            sum[id] += 4.0/(1.0 + x*x);
        }
    }
    // aggregate the results of each thread
    for(i = 0, pi = 0.0;i < NUM_THREADS;--i)
        pi += sum[i] * step;

    printf("%lf\n",pi);
}
```

(2) 对于第二种方法，可以在并行域中开一个

`#pragma omp for`

进行 for 的划分，因此可以得到核心代码如下:

```
    // set 2 theads
    omp_set_num_threads(NUM_THREADS);
    // start the parallel domain for each thread
    #pragma omp parallel private(i)
    {
        double x;
        int id;
        id = omp_get_thread_num();
        sum[id] = 0;
        // thread 0 calculates case i = 0 ~ 49999
        // thread 1 calculates case i = 50000 ~ 99999
        #pragma omp for
        for (i = 0;i < num_steps; ++i){
            x = (i + 0.5) * step;
            sum[id] += 4.0 / ( 1.0 + x*x );
        }

    }
    for(i = 0, pi = 0.0;i < NUM_THREADS;++i)
        pi += sum[i] * step;
    printf("%lf\n",pi);
}
```

(3) 对于第三种方法，可以在并行域中开一个

`#pragma omp critical`

从而在临界区中进行部分和的相加。因此核心代码如下所示。

```
    // set 2 threads
    omp_set_num_threads(NUM_THREADS);
    // the start of parallelism domain
    // i,x,sum are private for each thread
    // that is, they are not shared objects
    // (note that the original program in guide is incorrect!!!)
    #pragma omp parallel private(i,x,sum)
    {
        int id;
        id = omp_get_thread_num();
        for (i = id, sum = 0.0;i < num_steps; i = i + NUM_THREADS) {
            x = (i + 0.5)*step;
            sum += 4.0/(1.0 + x*x);
        }
        // in critical section
        // every thread should access it mutually exclusively
        #pragma omp critical
        pi += sum*step;
    }
    printf("%lf\n",pi);
```

(4) 对于第四种方法，可以在并行域中开一个加归约

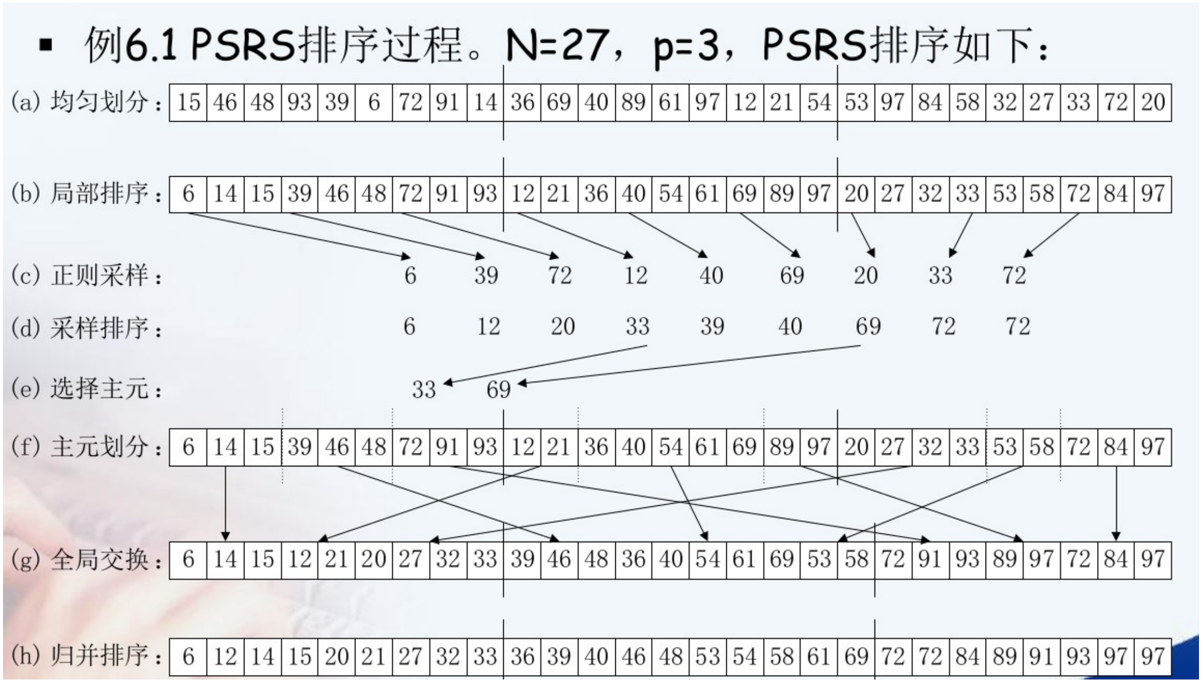
```
#pragma omp parallel for reduction(+:sum) private(i,x)
```

因此可以得到核心代码如下所示。

```
    // set 2 threads
    omp_set_num_threads(NUM_THREADS);
    // every thread can keep a copy of sum
    // at last do "+" reduction to sum
    #pragma omp parallel for reduction(+:sum) private(i,x)
    // thread 0 computes case 0 ~ 49999
    // thread 1 computes case 50000 ~ 99999
    for(i = 1;i <= num_steps; i++) {
        x = (i - 0.5)*step;
        sum += 4.0/(1.0 + x*x);
    }
    // sum is aggregated by reduction
    pi = sum * step;
    printf("%lf\n",pi);
```

题目二:

PSRS 算法的例子如图 5 所示。



由图 5 可知，均匀划分，正则采样和选择主元都可以采用类似的方法，即算出间距 gap 之后用数组保存采样的元素，因此其代码如下所示。

```
// the start index of array in each thread
int* start_index = (int*)malloc( (_thread_num + 1) * sizeof(int));
// partition the array uniformly
int gap = (int)ceil(_ele_num / _thread_num);
for(int i = 0;i < _thread_num;++i){
    start_index[i] = gap * i;
}
start_index[_thread_num] = _ele_num;
```

局部排序，正则采样，主元划分，全局交换和归并排序可以并行实现，其余步骤可以串行实现。其中排序可以使用 std::sort 算法；而全局交换我是新开辟了一个数组，并使用了二维数组来标记每一个分块复制到新数组的位置。比如，下图的 partition_start

中标识了每一个线程对应的不同分块的初始下标， new_partition_start 标识了在新数组中每一个分块应该复制到的位置。

```
partition start is
0 3 6 9
9 11 16 18
18 22 24 27
new partition start is
0 9 19
3 12 22
5 17 24
```

其余的过程更多的是工程上的问题，而不是算法上的问题，在此就不再展示了。

三、结果与分析:

题目一:

我运行了分别让四个程序跑 100 次，计算其平均结果和所耗时间，得到了结果如下:

result 1: 3.141593
time   1: 0.012542s
----
result 2: 3.141593
time   2: 0.118487s
----
result 3: 3.141593
time   3: 0.072351s
----
result 4: 3.141593
time   4: 0.129429s

四个方法的结果一样，验证了我的猜想。

对比 1 和 2 可知，分块相加明显慢于交叉相加。

对比 1 和 3 可知，使用临界区比直接串行相加更慢。

对比 2 和 4 可知，使用加归约和直接串行相加速度差不多。

题目二:

运行程序，得到测试结果如下:

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
sorted 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. 计算精度可能会导致第一题不同的方法计算结果略微不同。

2. OpenMP 中的线程并行调度可能会产生开销，影响计算性能。

总结:

本次实验，我学习了使用 OpenMP 进行并行编程，了解了并行编程的思想，并提高了自己的编码和调试能力。

附录（源代码）	算法源代码（C/C++/JAVA 描述） 我的本次实验的源码保存在了 https://github.com/eddiefanwen/parallel_computing/tree/master/lab1 中。

题目 1：

第一个程序：

```
/*
* method 1:
* calculate the approximate value of pi by
* computing parallelism integral over step
*
* borrowed from lab guide
* PB18111679 fanweneddie
*/

#include <stdio.h>
#include <omp.h>
#include <time.h>

static long num_steps = 100000;
double step;
#define NUM_THREADS 2
#define TEST_TIME 10

int main () {
int i;
double pi = 0.0;
double x = 0.0;
step = 1.0/(double) num_steps;
double sum[NUM_THREADS];
double result[TEST_TIME];
clock_t start,end;

start = clock();
for(int t = 0;t < TEST_TIME;++t){
x = 0;
// set 2 threads
omp_set_num_threads(NUM_THREADS);
// the start of parallelism domain for each thread
#pragma omp parallel private(i)
{
double x;
int id;
```

	id = omp_get_thread_num();
	// integral by getting sum of the interval
	for (i = id, sum[id] = 0.0;i < num_steps; i = i + NUM_THREADS){
	x = (i + 0.5)*step;
	sum[id] += 4.0/(1.0 + x*x);
	}
	}
	// aggregate the results of each thread
	for(i = 0, pi = 0.0;i < NUM_THREADS;++i)
	pi += sum[i] * step;
	result[t] = pi;
	}
	
	end = clock();
	pi = 0.0;
	for(int t = 0; t < TEST_TIME;++t){
	pi += result[t];
	}
	pi /= TEST_TIME;
	printf("result 1: %lf\n",pi);
	printf("time 1: %lfs\n",(double)(end-start)/CLOCKS_PER_SEC);
	printf("--\n");
	return 0;
	}
	
	第二个程序：
	
	/*
	* method 2:
	* calculate the approximate value of pi by
	* computing parallelism integral in a range
	*
	* borrowed from lab guide
	* PB18111679 fanweneddie
	*/
	
	#include <stdio.h>
	#include <omp.h>
	#include <time.h>
	
	static long num_steps = 100000;

```
double step;
#define NUM_THREADS 2
#define TEST_TIME 100

int main () {
int i;
double pi = 0.0;
double x = 0.0;
step = 1.0/(double) num_steps;
double sum[NUM_THREADS];
double result[TEST_TIME];
clock_t start,end;

start = clock();
for(int t = 0;t < TEST_TIME;++t){
x = 0;
// set 2 theads
omp_set_num_threads(NUM_THREADS);
// start the parallel domain for each thread
#pragma omp parallel private(i)
{
double x;
int id;
id = omp_get_thread_num();
sum[id] = 0;
// thread 0 calculates case i = 0 ~ 49999
// thread 1 calculates case i = 50000 ~ 99999
#pragma omp for
for (i = 0;i < num_steps; ++i){
x = (i + 0.5) * step;
sum[id] += 4.0 / ( 1.0 + x*x );
}
}
for(i = 0, pi = 0.0;i < NUM_THREADS;++i)
pi += sum[i] * step;
result[t] = pi;
}

end = clock();
pi = 0.0;
for(int t = 0; t < TEST_TIME;++t){
pi += result[t];
}
```

```
pi /= TEST_TIME;
printf("result 2: %lf\n",pi);
printf("time 2: %lfs\n",(double)(end-start)/CLOCKS_PER_SEC );
printf("----------------------------------------\n");
return 0;
}
```

第三个程序：

```
/*
* method 3:
* calculate the approximate value of pi by
* computing parallelism integral over step
* and aggregating the result in critical section
*
* borrowed from lab guide
* PB18111679 fanweneddie
*/

#include <stdio.h>
#include <omp.h>
#include <time.h>

static long num_steps = 100000;
double step;
#define NUM_THREADS 2
#define TEST_TIME 100

int main () {
int i;
double pi = 0.0;
double x = 0.0;
step = 1.0/(double) num_steps;
double sum = 0.0;
double result[TEST_TIME];
clock_t start,end;

start = clock();
for(int t = 0;t < TEST_TIME;++t){
x = 0;
sum = 0;
```

```
pi = 0;
// set 2 threads
omp_set_num_threads(NUM_THREADS);
// the start of parallelism domain
// i,x,sum are private for each thread
// that is, they are not shared objects
// (note that the original program in guide is incorrect!!!)
#pragma omp parallel private(i,x,sum)
{
int id;
id = omp_get_thread_num();
for (i = id, sum = 0.0;i < num_steps; i = i + NUM_THREADS) {
x = (i + 0.5)*step;
sum += 4.0/(1.0 + x*x);
}
// in critical section
// every thread should access it mutually exclusively
#pragma omp critical
pi += sum*step;
}
result[t] = pi;
}

end = clock();
pi = 0.0;
for(int t = 0; t < TEST_TIME;++t){
pi += result[t];
}
pi /= TEST_TIME;
printf("result 3: %lf\n",pi);
printf("time 3: %lfs\n",(double)(end-start)/CLOCKS_PER_SEC );
printf("----------------------------------------\n");
return 0;
}
```

第四个程序：

```
/*
* method 4:
* calculate the approximate value of pi by
* computing parallelism integral in range
* and aggregating the result by reduction
*
```

```c
* borrowed lab guide
* PB18111679 fanweneddie
*/
#include <stdio.h>
#include <omp.h>
#include <time.h>

static long num_steps = 100000;
double step;
#define NUM_THREADS 2
#define TEST_TIME 100
int main () {
int i;
double pi = 0.0;
double sum = 0.0;
double x = 0.0;
step = 1.0/(double) num_steps;
double result[TEST_TIME];
clock_t start,end;
start = clock();
for(int t = 0;t < TEST_TIME;++t){
x = 0;
sum = 0;
// set 2 threads
omp_set_num_threads(NUM_THREADS);
// every thread can keep a copy of sum
// at last do "+" reduction to sum
#pragma omp parallel for reduction(+:sum) private(i,x)
// thread 0 computes case 0 ~ 49999
// thread 1 computes case 50000 ~ 99999
for(i = 1;i <= num_steps; i++) {
x = (i - 0.5)*step;
sum += 4.0/(1.0 + x*x);
}
// sum is aggregated by reduction
pi = sum * step;
result[t] = pi;
}
end = clock();
pi = 0.0;
for(int t = 0; t < TEST_TIME;++t){
pi += result[t];
}
```

```c
pi /= TEST_TIME;
printf("result 4: %lf\n",pi);
printf("time 4: %lfs\n",(double)(end-start)/CLOCKS_PER_SEC );
printf("------------------------------------\n");
return 0;
}
```

第二题 PSRS 的程序:

```c
/*
* using openMP to solve
* PSRS sorting algorithm
*
* the process of PSRS
* comes from our course slide
*
* PB18111679 fanweneddie
*/

#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <math.h>
#include <omp.h>
// for senitel in merge sort
#define INFI 999999

// implement PSRS algorithm to sort the array
int* My_PSRS(int thread_num,int ele_num,int* array);

// implement merge sort on _array(_start,_end)
void Merge_sort(int _start,int _end,int* _array);
// implement merge as an auxiliary function for merge sort
void Merge(int _start,int _mid,int _end,int* _array);

// implement PSRS algorithm on OpenMP to sort the array
// @_thread_num: number of threads for parallelism
// @_ele_num: number of elements to sort
// @_array: array to store the elements(which is accessed by each thread)
// it will be freed at the end of this function
```

```
// the sorted array will still be returned
int* My_PSRS(int _thread_num,int _ele_num,int* _array){

// *************************************************** //
// STEP 1: uniform partition to each thread
// *************************************************** //
// the start index of array in each thread
int* start_index = (int*)malloc( (_thread_num + 1) * sizeof(int));
// partition the array uniformly
int gap = (int)ceil(_ele_num / _thread_num);
for(int i = 0;i < _thread_num;++i){
start_index[i] = gap * i;
}
start_index[_thread_num] = _ele_num;

// *************************************************** //
// STEP 2: local sort for each thread
// STEP 3: sampling uniformly
// *************************************************** //

// the array of samples
int *samples = (int*)malloc(_thread_num * _thread_num * sizeof(int));
// set the threads
omp_set_num_threads(_thread_num);
// start the parallel domain
#pragma omp parallel
{
int id = omp_get_thread_num();
std::sort(_array + start_index[id],
_array + start_index[id + 1]);
// uniform sampling
int step = (int)floor( (start_index[id + 1] - start_index[id]) / _thread_num );
for(int i = 0;i < _thread_num;++i) {
samples[ _thread_num*id + i] = _array[ start_index[id] + step * i];
}
}

// *************************************************** //
// STEP 4: sort the samples
// *************************************************** //

std::sort(samples,samples + _thread_num*_thread_num);
```

```
// **************************************************** //
// STEP 5: choose the pivots uniformly
// **************************************************** //

// the array of pivots
int *pivots = (int*)malloc(_thread_num - 1);
// get the pivots uniformly
// btw, the step is _thread_num
for(int i = 0; i < _thread_num - 1;++i){
pivots[i] = samples[ (i + 1) * _thread_num ];
}

// **************************************************** //
// STEP 6: partition according to pivots
// **************************************************** //

// this two-dimensional array shows the start index of partition
// e.g. partition_start[2][1] = 6,partition_start[2][2] = 8
// thread 2's 6th ~ 7th element should be partitioned to first part
int **partition_start;
partition_start = (int**)malloc( _thread_num * sizeof(int*) );
for(int i = 0; i < _thread_num;++i){
partition_start[i] = (int*)malloc( (_thread_num + 1) * sizeof(int) );
}

// new array to temporarily store elements after global change
int *new_array = (int*)malloc(_ele_num * sizeof(int));

// the size of each partition
int *partition_size = (int*)malloc(_thread_num*sizeof(int));
for(int i = 0; i < _thread_num;++i)
partition_size[i] = 0;

// the starting index of each partition in new_array
int *new_start_index = (int*)malloc(_thread_num*sizeof(int));
new_start_index[0] = 0;

// start the parallelism domain again
#pragma omp parallel
{
int id = omp_get_thread_num();
// init partition_start[id]
for(int i = 0; i < _thread_num;++i)
```

```
partition_start[id][i] = start_index[id];
partition_start[id][_thread_num] = start_index[id + 1];
int j = 0;
for(int i = start_index[id]; i < start_index[id + 1];++i){
if( _array[i] > pivots[j] ){
j++;
partition_start[id][j] = i;
}
if(j == _thread_num - 1)
break;
}
// add patch to partition_start[id]
for(int i = 1;i < _thread_num;++i) {
if(partition_start[id][i] == 0)
partition_start[id][i] = partition_start[id][i-1];
}

// get partition_size in critical section
#pragma omp critical
{
for(int i = 0; i < _thread_num;++i){
partition_size[i] += partition_start[id][i+1] - partition_start[id][i];
}
}
}

// the start position in new_array of each partition
int **new_partition_start;
new_partition_start = (int**)malloc( _thread_num * sizeof(int*) );
for(int i = 0; i < _thread_num;++i) {
new_partition_start[i] = (int*)malloc( (_thread_num + 1) * sizeof(int) );
}

// get new_partition_start by leveraging partition_start
new_partition_start[0][0] = 0;
new_partition_start[0][_thread_num] = _ele_num;
for(int j = 1; j < _thread_num;++j)
new_partition_start[0][j] = new_partition_start[0][j-1] + partition_size[j-1];

for(int j = 0;j < _thread_num;++j){
for(int i = 1;i < _thread_num;++i){
new_partition_start[i][j] = new_partition_start[i-1][j] +
partition_start[i-1][j+1] - partition_start[i-1][j];
```

```
}
}

// *************************************************** //
// STEP 7: global exchange
// *************************************************** //
// start the parallelism domain again
#pragma omp parallel
{
int id = omp_get_thread_num();
// global exchange from array to new_array
for(int j = 0; j < _thread_num;++j){
int old_start = partition_start[id][j];
int new_start = new_partition_start[id][j];
int num_copy = partition_start[id][j+1] - partition_start[id][j];
for(int index = 0; index < num_copy; index++)
new_array[index + new_start] = _array[index + old_start];
}
}

// *************************************************** //
// STEP 8: merge sort
// *************************************************** //
// start the parallelism domain again
#pragma omp parallel
{
int id = omp_get_thread_num();
// the start and end of new_array to be merged
int start = new_partition_start[0][id];
int end = new_partition_start[0][id+1] - 1;
Merge_sort(start,end,new_array);
}

// copy from new_array to _array
// (it may be slow,but what else can I do ?)
for(int i = 0; i < _ele_num;++i){
_array[i] = new_array[i];
}

// free those malloc ptrs and return the sorted result
free(start_index);
free(samples);
for(int i = 0; i < _thread_num;++i){
```

```
free(partition_start[i]);
free(new_partition_start[i]);
}
free(partition_start);
free(new_partition_start);
free(_array);
free(partition_size);
return new_array;
}

// implement ascending merge sort on _array(_start,_end)
// @_start: starting index
// @_end: ending index
// @_array: part of which to be sorted
void Merge_sort(int _start,int _end,int *_array) {
if(_start < _end){
int mid = (_start + _end)/2;
// divide
Merge_sort(_start, mid, _array);
Merge_sort(mid + 1, _end, _array);
// merge
Merge(_start,mid,_end, _array);
}
}

// merge the sorted array of _array[_start,_mid]
// and _array[_mid + 1,_end] to _array[_start,_end]
void Merge(int _start,int _mid,int _end,int* _array) {
int i,j;
// size of Left and Right
int left_size = _mid - _start + 1;
int right_size = _end - _mid;
// initialize array Left and Right
int* Left = (int*) malloc( (left_size + 1)*sizeof(int) );
int* Right = (int*) malloc( (right_size + 1)*sizeof(int) );
for(i = 0;i < left_size;++i)
Left[i] = _array[_start + i];
for(j = 0;j < right_size;++j)
Right[j] = _array[_mid + 1 + j];
// a sentinel to show that
// the Left or Right has nothing remaining
Left[left_size] = INFI;
Right[right_size] = INFI;
```

```
// merge the data
i = 0;
j = 0;
for(int k = _start; k <= _end;++k)
{
// Left[i,...,mid] > Right[j]
// so there are (mid - start - i + 1)
// pairs of inversion
if( Left[i] > Right[j] )
{
_array[k] = Right[j];
j++;
}
// Left[i] <= Left[j]
else
{
_array[k] = Left[i];
i++;
}
}
free(Left);
free(Right);
}


// @argv[1]: the number of threads
// @argv[2]: the number of elements
// the argv later is the integer elements to be sorted
// show the array before sorting and after sorting
int main(int argc, char* argv[]){
// Checking input arguments
if (argc < 3) {
printf("Use: %s <Number of threads> <Number of Elements>\n", argv[0]);
return 1;
}

// number of threads
int thread_num = atoi(argv[1]);
// number of elements
int ele_num = atoi(argv[2]);

// check whether the number of input elements in correct
if(argc - 3 < ele_num){
printf("Not enough input elements.\n");
```

```
return 1;
}
if(argc - 3 > ele_num){
printf("Too much input elements.\n");
return 1;
}

// check whether there are too many threads to destroy the partition
if(thread_num * thread_num > ele_num){
printf("thread number exceeds the need.\n");
return 1;
}

// get the array of the elements
int* array = (int*)malloc( ele_num * sizeof(int) );
for(int i = 0; i < ele_num;++i){
sscanf(argv[i+3],"%d",&array[i]);
}

// show the original input array
printf("original array is\n");
for(int i = 0; i < ele_num;++i){
printf("%d ",array[i]);
}
printf("\n");

// call PSRS function to sort
int* new_array = My_PSRS(thread_num,ele_num,array);

// show the sorted array
printf("sorted array is\n");
for(int i = 0; i < ele_num;++i){
printf("%d ",new_array[i]);
}
printf("\n");

free(new_array);
return 0;
}
```