F74109016 葉惟欣 多處理機平行程式設計 HW5

第一題: Count Sort

Please answer the following questions:

1. If we try to parallelize the for i loop (the outer loop), which variables should be private and which should be shared?

Private 變數 : i (每個 thread 負責比較的範圍不同) , j (每個 thread 負責的範圍不同) , count(每個 thread 各自算自己負責的元素排序第幾，for 迴圈每次都會 initialize 為 0 不影響其他 thread 的計數過程)
共享 shared 變數 : n(總共要跑幾個回合),a(存放元素的陣列),temp(存放 sort 結果的陣列)

```
#pragma omp parallel for private(i,j,count) shared(n,a,temp)
    for(i = 0;i < n;i++){
        count = 0;
        for(j = 0;j < n;j++){
            if(a[j] < a[i]) count++;
            else if(a[j] == a[i] && j < i)count++;
        }
        temp[count] = a[i];
    }
#pragma omp parallel for
```

2. If we parallelize the for i loop using the scoping you specified in the previous part,are there any loop-carried dependences? Explain your answer.

如果將 for i loop 平行化，並不會有 loop-carried dependences，因為將 i 平行化，只代表每個 thread 是要負責幾個元素，去計算有幾個元素比他小而已。而其他 thread 也只是負責計算他負責的元素在整個陣列次排序第幾個而已。彼此間沒有資料相依。而且每次 for i loop 都會將 count 的計數 reset 為 0，所以如果一個 thread 可能負責很多個元素也彼此沒有關係。

3. Can we parallelize the call to memcpy? Can we modify the code so that this part of the function will be parallelizable?

可以，可以。
能夠將 call to memcpy 平行化，透過多次 call to memcpy 的方式。或是直接修改 memcpy 函數。

4. Write a C program that includes a parallel implementation of Count sort.

```
    /*** serial original count sort ***/

    /*** parallel count sort ***/
    // Implement with Open MP
    // We use the same concept of "serial original count sort mentioned aboved，but we parallizied the for i loop.
    // Thus we can assign every thread to cacluate some elements evenly.
    starttime = clock();
#pragma omp parallel for num_threads(numberOfThreads) private(i,j,count) shared(n,a,temp)
    for(i = 0;i < n;i++){
        count = 0;
        for(j = 0;j < n;j++){
            if(a[j] < a[i]) count++;
            else if(a[j] == a[i] && j < i)count++;
        }
        temp[count] = a[i];
    }
#pragma omp parallel for
    for(i = 0;i < n;i++)a[i] = temp[i];
    endtime = clock();
    cout << "Total time parallel count sort: " << (double)(endtime - starttime)/CLOCKS_PER_SEC<<endl;
    /*** parallel ***/
```

5. How does the performance of your parallelization of Count sort compare to serial Count sort? How does it compare to the serial qsort library function?

Qsort 效率比平行 count sort 更好，平行 count sort 又比循序 count sort 的效率還好。

Compile and execute

```
F74109016@sivslab-pn1:~/HW5$ g++ -o h5_problem1.exe h5_problem1.cpp -fopenmp
F74109016@sivslab-pn1:~/HW5$ ./h5_problem1.exe 80 1000
number of element : 1000
number of thread : 80
Total time serial-qsort: 0.003469
Total time original count sort: 0.011996
Total time parallel count sort: 0.02387
F74109016@sivslab-pn1:~/HW5$
```

問題 1 : What have you done?

生成初始陣列

```
srand(time(NULL));

int i,j,count;
int numberOfThreads = atoi(argv[1]); // first parameter is the number of thread
int n = atoi(argv[2]);               // second parameter is the number of element in array
int numberOfThreads = atoi(argv[1]);
// Print the info
printf("number of element : %d\nnumber of thread : %d\n",n,numberOfThreads);
// We use two array. a is the array of random number with range from 0 - 999
// c is the array of the count (rank) we will use in the count sort
int *a = new int[n];
int *c = new int[n];
int *temp = new int[n];
// create random array
for(i = 0;i < n;i++)  a[i] = rand()%(n);
for(i = 0;i < n;i++){
    c[i] = a[i];
}
clock_t starttime,endtime;
```

(1) Qsort

Just call the API of C library

```
/*** serial-qsort ***/
// Just call the API of C library
starttime = clock();
qsort(c,n,sizeof(int),cmp);
endtime = clock();
cout << "Total time serial-qsort: " << (double)(endtime - starttime)/CLOCKS_PER_SEC<<endl;

/*** serial-qsort ***/
```

(2) 基本 count sort (相當於一個 thread 的 count sort)

The count sort algo is to calculate the specific element's rank in total elements of array. After calculating, we push the specific element in their right position (index is just its rank - 1)

```
/*** serial original count sort ***/
// The count sort algo is to calculate the specific element's rank in total elements of array
// After calculating, we push the specific element in their right position (index is just its rank - 1 )
starttime = clock();
for(i = 0;i < n;i++){
    count = 0;
    for(j = 0;j < n;j++){
        if(a[j] < a[i]) count++;
        else if(a[j] == a[i] && j < i)count++;
    }
    temp[count] = a[i];
}
for(i = 0;i < n;i++)a[i] = temp[i];
endtime = clock();
cout << "Total time original count sort: " << (double)(endtime - starttime)/CLOCKS_PER_SEC<<endl;
/*** serial original count sort ***/
```

(3) 平行的 count sort (Open MP)

Implement with Open MP. We use the same concept of "serial original count sort mentioned aboved，but we parallizied the for i loop. Thus we can assign every thread to cacluate some elements evenly.

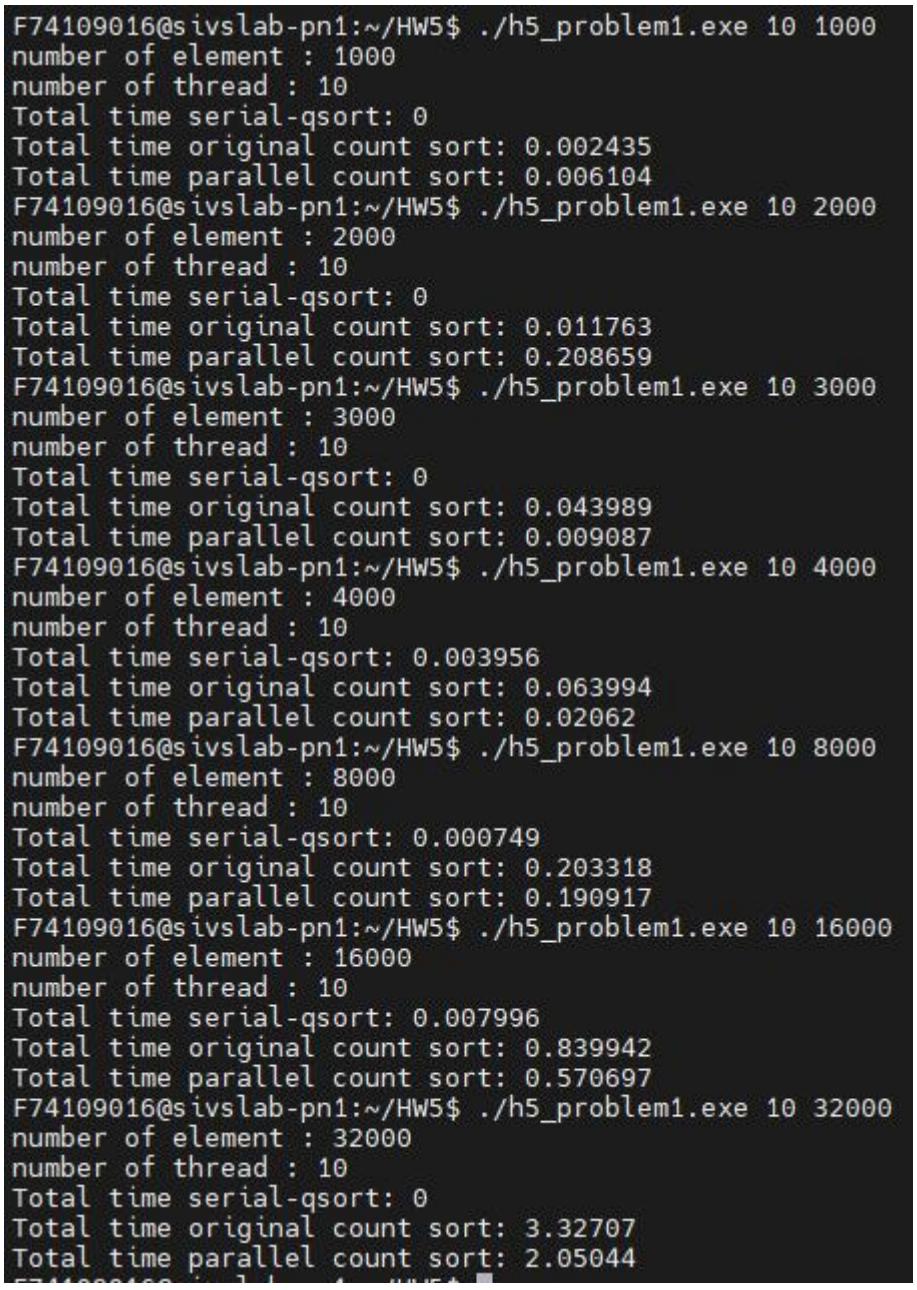
```
    /*** serial original count sort ***/

    /*** parallel count sort ***/
    // Implement with Open MP
    // We use the same concept of "serial original count sort mentioned aboved，but we parallizied the for i loop.
    // Thus we can assign every thread to cacluate some elements evenly.
    starttime = clock();
#pragma omp parallel for num_threads(numberOfThreads) private(i,j,count) shared(n,a,temp)
    for(i = 0;i < n;i++){
        count = 0;
        for(j = 0;j < n;j++){
            if(a[j] < a[i]) count++;
            else if(a[j] == a[i] && j < i)count++;
        }
        temp[count] = a[i];
    }
#pragma omp parallel for
    for(i = 0;i < n;i++)a[i] = temp[i];
    endtime = clock();
    cout << "Total time parallel count sort: " << (double)(endtime - starttime)/CLOCKS_PER_SEC<<endl;
    /*** parallel ***/
```

問題 2 : Analysis on your result

Qsort 效率比平行 count sort 更好，平行 count sort 又比循序 count sort 的效率告好

觀察一 ： 觀察隨著 number of elements 的增加，三個方式的效率比。



可以看到隨著 number of elements 的增加，達到 3000 個 elements，平行化 count sort 的時間會比 original count sort 還少。代表當數量變多平行化的效益會有比較顯著的提升。

觀察二 ： 隨著 number of threads 的數量增加，平行化的帶來的效益如何?

```
F74109016@sivslab-pn1:~/HW5$ ./h5_problem1.exe 10 1000
number of element : 1000
number of thread : 10
Total time serial-qsort: 0
Total time original count sort: 0.004
Total time parallel count sort: 0.004548
F74109016@sivslab-pn1:~/HW5$ ./h5_problem1.exe 20 1000
number of element : 1000
number of thread : 20
Total time serial-qsort: 0
Total time original count sort: 0.015656
Total time parallel count sort: 0.019341
F74109016@sivslab-pn1:~/HW5$ ./h5_problem1.exe 30 1000
number of element : 1000
number of thread : 30
Total time serial-qsort: 0
Total time original count sort: 0.003998
Total time parallel count sort: 0.00899
F74109016@sivslab-pn1:~/HW5$ ./h5_problem1.exe 40 1000
number of element : 1000
number of thread : 40
Total time serial-qsort: 0
Total time original count sort: 0.001032
Total time parallel count sort: 0.006539
F74109016@sivslab-pn1:~/HW5$ ./h5_problem1.exe 50 1000
number of element : 1000
number of thread : 50
Total time serial-qsort: 0
Total time original count sort: 0.00762
Total time parallel count sort: 0.009303
F74109016@sivslab-pn1:~/HW5$ ./h5_problem1.exe 60 1000
number of element : 1000
number of thread : 60
Total time serial-qsort: 0
Total time original count sort: 0.003999
Total time parallel count sort: 0.010044
F74109016@sivslab-pn1:~/HW5$ ./h5_problem1.exe 70 1000
number of element : 1000
number of thread : 70
Total time serial-qsort: 0
Total time original count sort: 0.004973
Total time parallel count sort: 0.006722
F74109016@sivslab-pn1:~/HW5$ ./h5_problem1.exe 80 1000
number of element : 1000
number of thread : 80
Total time serial-qsort: 0
Total time original count sort: 0.011962
Total time parallel count sort: 0.021326
```

在觀察這個我發現 ,我發現這個時間計算跟想像的不太一樣 ,因為 number of elements 的數量未改變，但 original count sort 的時間卻有劇烈變化，由此可知，當我們在算時間時仍要可慮其他可能的因素，而非只是 number of thread 與 process 的 number of element 兩個變數而已。

問題 3 : Any difficulites?

這次的作業比較困難的部分就是一開始寫 OpenMP 比較不知道語法查了一下。

第二題 :

Compile and execute :

```
F74109016@sivslab-pn1:~/HW5$ gcc -o h5_problem2.exe h5_problem2.c -fopenmp
F74109016@sivslab-pn1:~/HW5$ ./h5_problem2.exe 65 test100.txt
number of Threads : 65
frequency of hello : 3334
frequency of parallel : 3333
frequency of programming : 3333
```

問題 1 : What have you done?

inputproduce.c 生成 input file : test10.txt / test100.txt

```
srand(time(NULL));
char * token1 = "hello";
char * token2 = "parallel";
char * token3 =  "programming";
char * token[3] = {token1,token2,token3};
int num[3] = {3334,3333,3333};


FILE *writeMatrix = fopen("test100.txt","w");
fprintf(writeMatrix,"%d %d\n",100,100);
for(int i=0;i<100;i++){
     for(int j=0;j<100;j++){
         int rand;
         while(1){
             rand = random()%3;
             if(num[rand] > 0){
                 num[rand]--;
                 break;
             }
         }
         fprintf(writeMatrix,"%s ",token[rand]);
     }
   fprintf(writeMatrix,"\n");
}
fclose(writeMatrix);

return 0;
```

Input file 說明 :

亂數排序 “hello” “parallel” “programming” 三個 keywords。

第一行說明此 file 有幾行幾列

Test10 : 10 行 10 列

Frequency of “hello” : 34

Frequency of “parallel” : 33

Frequency of “programming” : 33

test10.txt

```
10 10
hello hello programming programming hello parallel parallel parallel hello hello
hello parallel hello parallel programming parallel hello programming hello programming
parallel programming hello hello hello hello parallel parallel hello hello
programming hello programming parallel hello programming parallel parallel parallel programming
programming programming hello programming parallel hello parallel programming hello hello
parallel programming hello programming parallel hello hello programming programming hello
hello parallel parallel programming hello parallel programming parallel hello parallel
parallel parallel parallel programming parallel hello hello parallel programming parallel
parallel programming parallel parallel programming programming programming programming programming parallel
parallel hello hello programming programming hello programming programming hello programming

```

Test10 : 10 行 10 列

Frequency of “hello” : 3334

Frequency of “parallel” : 3333

Frequency of “programming” : 3333

test100.txt

```
100 100
parallel parallel hello parallel programming parallel parallel hello programming hello parallel hello hello
 programming programming parallel parallel parallel programming hello programming parallel parallel parallel
 hello programming parallel parallel parallel programming parallel programming parallel parallel parallel hello
 parallel hello parallel hello parallel hello parallel parallel parallel programming hello programming hello
 programming programming hello programming programming programming programming parallel hello
 parallel hello parallel parallel parallel parallel hello programming programming hello parallel hello parallel
 programming parallel hello hello parallel hello hello parallel parallel hello programming hello hello parallel
 programming programming programming parallel hello hello hello programming programming parallel hello
 programming parallel parallel
hello programming hello hello parallel programming hello hello hello parallel hello parallel programming
 programming parallel programming programming programming programming parallel parallel hello hello parallel
 hello programming parallel hello parallel programming programming hello programming parallel hello parallel
 hello programming parallel hello parallel parallel programming hello programming parallel hello parallel
 parallel programming hello programming hello hello parallel hello hello programming programming programming
 programming programming hello programming hello parallel parallel parallel parallel hello programming
 programming programming programming programming programming hello hello hello programming hello parallel
 programming parallel programming hello programming hello hello parallel hello parallel programming hello hello
 programming programming hello parallel hello
parallel parallel programming hello hello programming parallel parallel hello parallel parallel parallel hello
 parallel programming hello programming programming parallel parallel parallel programming programming parallel
 parallel hello hello hello hello programming programming programming hello parallel hello parallel programming
 programming hello hello hello parallel parallel programming hello parallel hello hello parallel parallel
 parallel hello parallel parallel parallel programming programming programming programming parallel programming
 programming hello hello programming parallel programming parallel hello hello parallel programming programming
 parallel parallel hello programming parallel parallel hello programming hello parallel hello parallel
 programming hello parallel programming parallel programming programming hello parallel programming programming
 programming programming parallel hello
programming parallel hello hello programming parallel hello programming programming parallel parallel
 programming parallel programming parallel hello programming parallel parallel parallel programming programming
 parallel parallel hello hello hello programming parallel programming hello hello hello hello hello hello
```

Code implement with Open MP :

Function : operation of the buffer

```
int buffer_is_empty(){
    return fill == use;
}
int buffer_is_full(){
    return (fill+1)%BUFFER_SIZE == use;
}
void insert_item(char** item){
    buffer[fill] = item;
    fill = (fill+1)%BUFFER_SIZE;
}
char** remove_item(){
    char** item = buffer[use];
    use = (use+1)%BUFFER_SIZE;
    return item;
}
```

實作方法：

1.我平行化的是 for i loop 總次數為 row * 2 也就是 producer 要 produce row 個 line (insert to buffer),而 consumer 要 consume row 個 line(remover from buffer)。也因此這個總次數為 producer 與 consumer 總共要工作的次數。

解釋 Open Mp 的 thread task 將 i 去平行化，而分配給的 number of thread 為 execute 時給。如此，有些 thread 會去做 producer 有些 thread 會去做 consumer，更再來可能有些 thread 可能此時會去做 producer 接下來會去做 consumer，要看分配的 number of thread 為奇數或偶數，因為我在平行化時是透過 schedule(static,1) 來實作。

Case 奇數：則會有有些 thread 可能此時會去做 producer 接下來會去做 consumer 的情形出現。因為有可能 thread 這被 assign 的 task 的 i 為奇數或偶數

Case 偶數：則 thread 要不就是當 producer 要不就是當 consumer。

2. 接下來在 for 回圈內部我透過判斷現在每個 thread 分配的 i 為奇數還是偶數來判斷 thread 這次要當 producer 還是 consumer，如此一來才能正確地將 produce 的數量 = consume 的數量，而不會 consume 一直等待 produce 卻已經結束的不平衡情況出現。

3. thread safe 實作

第一個 critical section 為 buffer 的 insert 如果現在 buffer 為 full，producer 就必須等待。而此時因為已經進入 critical section (pro) ，所以沒有其他 buffer 可以在改變變數 fill 的值，也因此不會有兩個 thread 同時 insert 到同個位置的情形。

第二個 critical section 為 buffer 的 remove。如果現在 buffer 為 empty，consumer 就必須等待。而此時因為已經進入 critical section (con) ，所以沒有其他 buffer 可以在改變變數 use 的值，也因此不會有兩個 thread 同時 consume 到同個位置的情形。

4. tokenize 的實作

What happened? Recall that `strtok` caches the input line. It does this by declaring a variable to have `static` storage class. This causes the value stored in this variable to persist from one call to the next. Unfortunately for us, this cached string is shared, not private. Thus, it appears that thread 1's call to `strtok` with the second line has apparently overwritten the contents of thread 0's call with the first line. Even worse, thread 0 has found a token ("days") that should be in thread 1's output.

The `strtok` function is therefore *not* thread-safe: if multiple threads call it simultaneously, the output it produces may not be correct. Regrettably, it's not uncommon for C library functions to fail to be thread-safe. For example, neither the random number generator `random` in `stdlib.h` nor the time conversion function `localtime` in `time.h` is thread-safe. In some cases, the C standard specifies an alternate, thread-safe, version of a function. In fact, there is a thread-safe version of `strtok`:

```
char* strtok_r(
      char*       string      /* in/out */,
      const char* separators  /* in     */,
      char**      saveptr_p   /* in/out */);
```

The "_r" is supposed to suggest that the function is *re-entrant*, which is sometimes used as a synonym for thread-safe. The first two arguments have the same purpose as the arguments to `strtok`. The `saveptr_p` argument is used by `strtok_r` for keeping track of where the function is in the input string; it serves the purpose of the cached pointer in `strtok`. We can correct our original `Tokenize` function by replacing the calls to `strtok` with calls to `strtok_r`. We simply need to declare a `char*` variable to pass in for the third argument, and replace the calls in Line 17 and Line 20 with the calls

```
my_token = strtok_r(lines[i], " \t\n", &saveptr);
. . .
my_token = strtok_r(NULL, " \t\n", &saveptr);
```

respectively.

Because the "r" is supposed to suggest that the function is **re-entrant**, which is sometimes used as a synonym for thread-safe. The first two arguments have the same purpose as the arguments to strtok. The saveptr p argument is used by strtok r for keeping track of where the function is in the input string; it serves the purpose of the cached pointer in strtok. We can correct our original Tokenize function by replacing the calls to strtok with calls to strtok r.

也因此我的寫法為：

```
if(k==0)
    word = strtok_r(next," ",&saveptr);
else
    word = strtok_r(NULL," ",&saveptr);
```

5. Producer :

```
#pragma omp parallel for num_threads(numberOfThreads) schedule(static,1)
//The producers read text from a collection of files, one per producer.
for(int i=0;i<row*2;i++){
      int tid = omp_get_thread_num();
      if(i %2 == 0){
          # pragma omp critical (pro)
          {
            while(buffer_is_full()); //thread safe queue
            //The producers insert lines of text into a single shared queue
            //(please implement your own thread safe queue)
            insert_item(&lines[i/2]);
          }
      }
```

6. Consumer

```
else{
  char * my_token;
  char ** next_consumed;
  char * next;
  char * saveptr = NULL;
  # pragma omp critical (con)
  {
    while(buffer_is_empty());
    //The consumers take the lines of text and tokenize them.
    next_consumed = remove_item();
  }
  next = *next_consumed;

  char * word;
  for(int k=0;k<NITER;k++){
      //Tokens are “words” separated by white space.
      if(k==0)
          word = strtok_r(next," ",&saveptr);
      else
          word = strtok_r(NULL," ",&saveptr);
      //When a consumer finds a token that is keyword, the keyword count increases one.
      if(strcmp(word,token[0])==0){
          # pragma omp critical(hello)
          num[0]++;
      }
      else if(strcmp(word,token[1])==0){
          # pragma omp critical(parallel)
          num[1]++;
      }
      else if(strcmp(word,token[2])==0){
          # pragma omp critical(programming)
          num[2]++;
      }
  }
```

Consumer 在累加 keyword 出現的頻率也用 critical section 去做避免同時有兩個 thread 都在進行修改 fequency 的情況出現

問題 2 : Analysis on your result

Result is correct.

# of thread : 3

```
F74109016@sivslab-pn1:~/HW5$ gcc -o h5_problem2.exe h5_problem2.c -fopenmp
F74109016@sivslab-pn1:~/HW5$ ./h5_problem2.exe 3 test10.txt
number of Threads : 3
frequency of hello : 34
frequency of parallel : 33
frequency of programming : 33
```

# of thread : 3

```
F74109016@sivslab-pn1:~/HW5$ ./h5_problem2.exe 3 test100.txt
number of Threads : 3
frequency of hello : 3334
frequency of parallel : 3333
frequency of programming : 3333
```

# of thread : 4

```
F74109016@sivslab-pn1:~/HW5$ ./h5_problem2.exe 4 test100.txt
number of Threads : 4
frequency of hello : 3334
frequency of parallel : 3333
frequency of programming : 3333
```

# of thread : 5

```
F74109016@sivslab-pn1:~/HW5$ ./h5_problem2.exe 5 test100.txt
number of Threads : 5
frequency of hello : 3334
frequency of parallel : 3333
frequency of programming : 3333
```

# of thread : 16

```
F74109016@sivslab-pn1:~/HW5$ ./h5_problem2.exe 16 test100.txt
number of Threads : 16
frequency of hello : 3334
frequency of parallel : 3333
frequency of programming : 3333
```

# of threads : 17

```
F74109016@sivslab-pn1:~/HW5$ ./h5_problem2.exe 17 test100.txt
number of Threads : 17
frequency of hello : 3334
frequency of parallel : 3333
frequency of programming : 3333
```

```
F74109016@sivslab-pn1:~/HW5$ ./h5_problem2.exe 32 test100.txt
number of Threads : 32
frequency of hello : 3334
frequency of parallel : 3333
frequency of programming : 3333
F74109016@sivslab-pn1:~/HW5$ ./h5_problem2.exe 33 test100.txt
number of Threads : 33
frequency of hello : 3334
frequency of parallel : 3333
frequency of programming : 3333
F74109016@sivslab-pn1:~/HW5$ ./h5_problem2.exe 64 test100.txt
number of Threads : 64
frequency of hello : 3334
frequency of parallel : 3333
frequency of programming : 3333
F74109016@sivslab-pn1:~/HW5$ ./h5_problem2.exe 65 test100.txt
number of Threads : 65
frequency of hello : 3334
frequency of parallel : 3333
frequency of programming : 3333
```

Conclusion :

No matter the what is the number of thread the result is correct

問題 3 : Any difficulites?

這一題我花很久的時間寫，因為一開始我是先將 producer tokenize 完 將每個 keyword 放入 buffer 的寫法，後來再理解題意後發現是 consumer 要去 tokenize 如此以來才不會讓 producer 又要讀 line of file 又要 tokenize，而 consumer 只要 count the frequency 造成 load inbalance。

但修正成 producers insert lines of text into a single shared queue.The consumers take the lines of text and tokenize them. 發生很多 Segmentation fault 的問題。修了很久。

第二個困難的地方是，我原本是用 parallel section 去實作，後來上網查發現 parallel section 的作法是不同的 section 用不同的 thread 去實作，並非我想的不同的 section 用不同的很多 thread 再去平行化實作一樣。
section 的用意是希望讓不同 function 的 code 可以平行化，並非還可以分配很多 thread 去一起完成一個 section 。經過思考後我才改成現在這個方法。

參考 ： 簡易的程式平行化 - OpenMP（三）範例 parallel、section

https://kheresy.wordpress.com/2006/09/15/%E7%B0%A1%E6%98%93%E7%9A%84%E7%A8%8B%E5%BC%8F%E5%B9%B3%E8%A1%8C%E5%8C%96%EF%BC%8Dopenmp%EF%BC%88%E4%B8%89%EF%BC%89%E7%AF%84%E4%BE%8B-parallel%E3%80%81section/

https://chenhh.gitbooks.io/parallel_processing/content/openmp/openmp_setting.html

問題 4 : (optional) Feedback to TAs

第二題的作業很有趣，激發我很多實作的想法，因為 producer 跟 consumer 的工作都不一樣，同時 access shared variable 也要考慮 mutual exclusion 的問題。Buffer 是否同時被 insert 與 remove 的問題。寫完這次作業很有成就感謝謝助教。