多處理機平行程式設計 2021 Homework 5

學號：F74076027
姓名：林政傑
系級：資訊 111

測試環境：Linux ubuntu 5.11.0-40-generic
CPU：Intel i7-10750H (6 core 12 thread) @ 2.6GHz
GCC version：9.3.0

1. Count Sort

Q1: If we try to parallelize the for i loop (the outer loop), which variables should be private and which should be shared?

`count`, `j`, `i` should be private. `a`, `temp`, `n` should be public.

Q2: 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.

不會有 loop-carried dependency。因為每個 for i loop 都可以獨自完成計算，不需要使用先前迴圈計算出來的結果。

Q3: Can we parallelize the call to memcpy? Can we modify the code so that this part of the function will be parallelizable?

不可以，因為每個 for j loop 都必須走訪陣列 `a` 的所有元素，如果使用平行化的 `memcpy`，會導致尚未完成的 thread 讀到錯誤的資訊。

Q4: Write a C program that includes a parallel implementation of Count sort.

在 h5_problem1.c，透過 `gcc h5_problem1.c -lpthread` 編譯，直接執行會用 4 個 thread 排序 10 個介於 0 到 9999999 的數，並且印出排序前後的陣列。透過 `argv[1]` 和 `argv[2]` 更改 n_thread 和 N，例如 `a.out 2 100` 使用 2 個 thread 排序 100 個介於 0 到 9999999 的數。

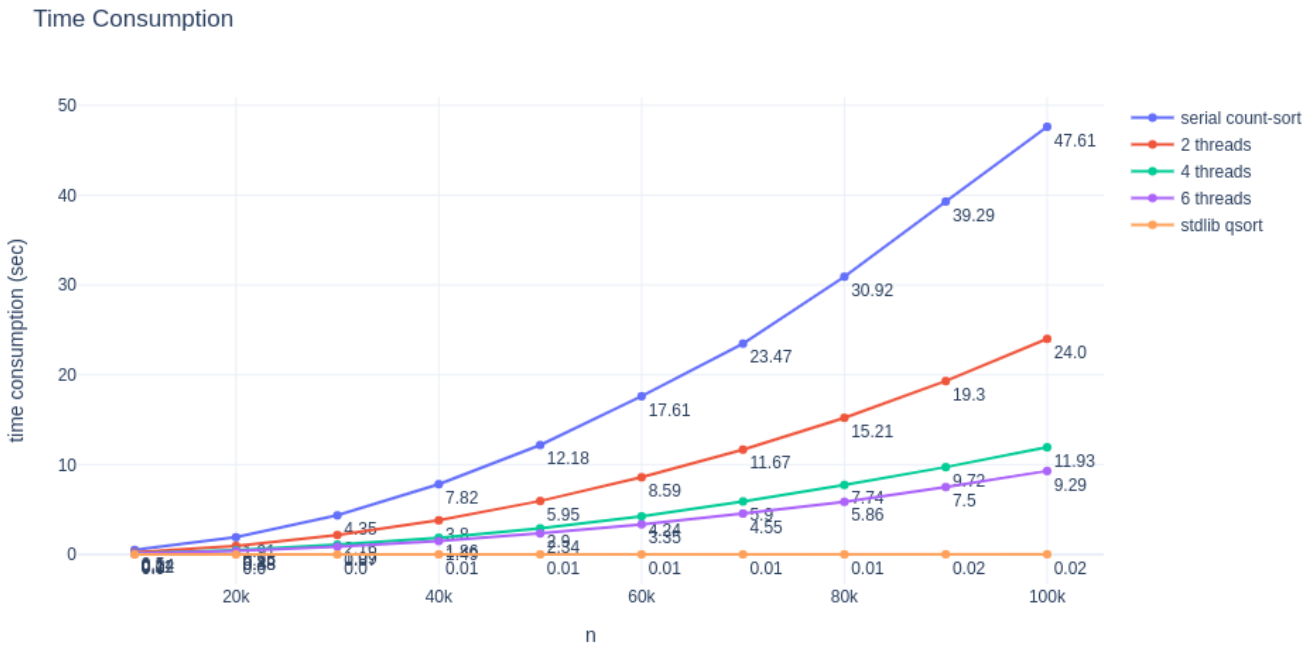
Q5: How does the performance of your parallelization of Count sort compare to serial Count sort? How does it compare to the serial qsort library function?

測試 parallel、serial count sort 與 qsort 執行時間的差距，使用 `clock_gettime` 測量時間。調整陣列長度 `n` 以及 thread 數量 `n_thread`，觀察兩者對於執行時間的影響，`n` 的範圍為 10000 到 100000、`n_thread` 的範圍為 1 到 6。這次比較忙，就不像以往那樣重複測試然後取 10% trimmed mean 了，每一種 `n` 與 `n_thread` 的組合只測試一次。腳本如下：

```
gcc h5_problem1.c -lpthread
gcc qsort.c -o qsort
for (( t = 1; t <= 6; t++ ))
do
    for (( n = 10000; n < 100000; n += 10000))
    do
```

```
        ./a.out "$t" "$n"
    done
done

for (( n = 10000; n < 100000; n += 10000))
do
    ./qsort "$n"
done
```



上圖每一條折線即為 `n` 對應 `n_thread` 的執行時間，當 n = 1 時即等同於 serial count-sort，初始化 pthread 的時間可以忽略不計。為了讓圖片不至於太過混亂，我只畫出 n = 1, 2, 4, 6 的執行時間。可以看到，使用 2 個 thread 的執行時間為 serial 的二分之一、使用 6 個 thread 則為五分之一到六分之一；C 語言內建的 qsort 為最底部的橘色線，與我自己實做的 count-sort 完全是不同檔次，最多只花了 0.02 秒。

我使用 perf 量測了 qsort 與 count-sort 執行的一些數據如下：

```
 Performance counter stats for './qsort 100000':

          979,5621      branches
           83,4525      branch-misses             #    8.52% of all
branches
            5,8497      cache-misses
         4745,0152      cycles
         5837,2890      instructions              #    1.23  insn per cycle
                2      context-switches

       0.029011647 seconds time elapsed
```

```
 Performance counter stats for './a.out 1 100000':

     300,1094,6452      branches
      26,8702,8462      branch-misses             #    8.95% of all
branches
          168,0405      cache-misses
    1699,3329,9801      cycles
    2600,6146,8518      instructions              #    1.53  insn per cycle
               230      context-switches

      46.250807377 seconds time elapsed
```

由上面 qsort 與 count-sort 的分析可以發現 count-sort 的 branch 和 cycle 遠遠多於 qsort，造成 count-sort 花費很長的時間。參考了這篇翻譯 glibc/stdlib/stdlib/qsort.c 註解的部落格 https://www.cnblogs.com/wkqblog/p/3220431.html ，點出了 qsort 實作的技巧：

- 不用遞迴，自己定義 stack 進行迭代
- 用 median-of-three 作為 pivot
- 大量使用 macro 代替 function，好處是 preprocess 以及省略 type checking
- 在 qsort 的 segment 元素數量小於 4 時改採用 insertion sort

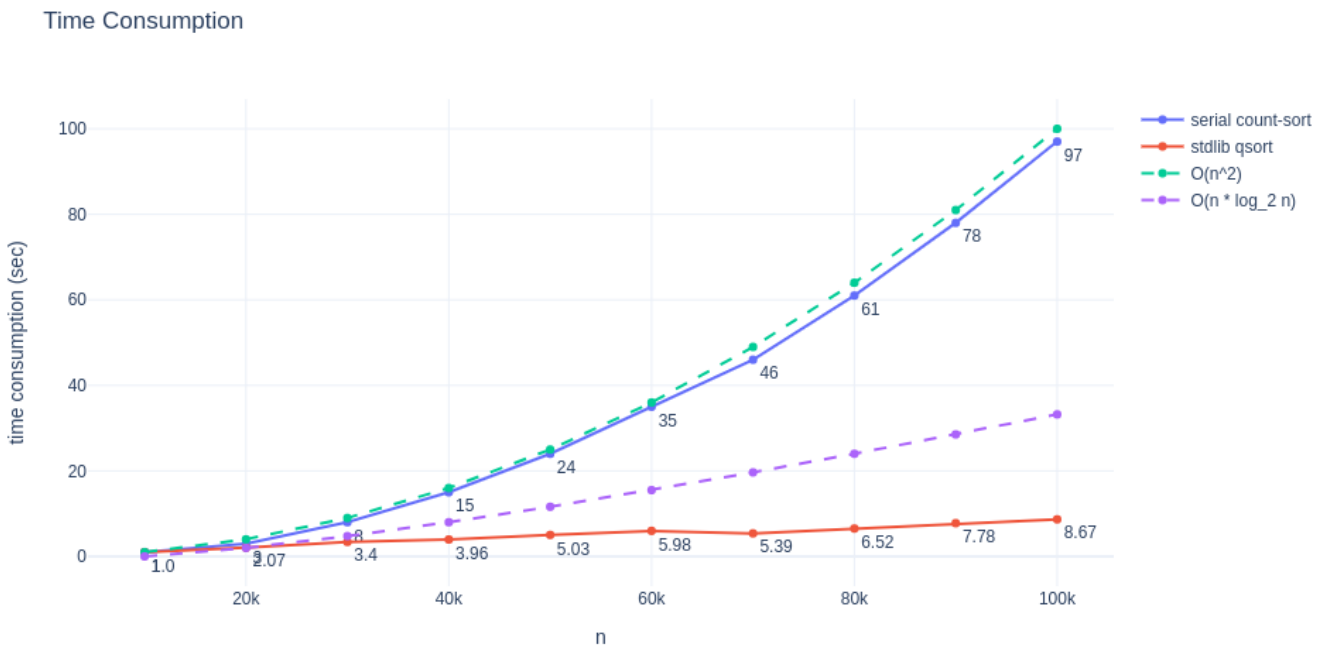
因此 qsort 比單純 O(n^2) 的 count-sort 快很多。

What have you done

所有 thread 共享陣列 `a` 以及 `tmp`，將 `a` 的區段平分給 thread 執行，也就是將原本程式碼的 `for (i = 0; i < n; i++)` 在每個 thread 轉化為 `for (i = start; i < end; i++)`。當所有 thread 都執行完畢後再統一使用 `memcpy` 把 `tmp` 覆寫到 `a`。

Analysis on your result

我將 qsort 與 count-sort 的執行時間根據 n = 10000 的執行時間標準化，與 O(n^2) 和 O(n * log_2 n) 比較，發現 count-sort 接近 O(n^2) 而 qsort 則遠低於 O(n * log_2 n)。



Any difficulites?

無

2. Producer-Consumer

What have you done

shared queue 與 keyword dictionary 使用同一個 queue 資料結構，queue 中的每個 node 有 integer data 和 string data。在 shared queue 只會使用 string data 來儲存一行 text，在 keyword dictionary 則會用 string data 儲存 keyword、用 integer data 儲存 keyword 的數量。

```
/* Queue node */
struct node {
    int idata;          // integer data
    char *sdata;        // string data
    struct node *next;
};

/* Queue data structure */
typedef struct queue {
    struct node *head;
    struct node *tail;
} queue;
```

thread-safe enqueue 與 dequeue 的實做：

```
/* Enqueue */
void enqueue(queue *q, char *data)
```

```
{
    struct node *_new = (struct node*)malloc(sizeof(struct node));
    _new->idata = 0;
    _new->sdata = strdup(data);
    _new->next = NULL;
    #pragma omp critical
    {
        if (q->tail == NULL) {
            q->head = _new;
            q->tail = _new;
        } else {
            q->tail->next = _new;
            q->tail = _new;
        }
    }
}

/* Dequeue */
char *dequeue(queue *q)
{
    char *data = NULL;
    #pragma omp critical
    {
        if (q->head != NULL) {
            data = q->head->sdata;
            struct node *del = q->head;
            q->head = q->head->next;
            free(del);
        }
        if (q->head == NULL) {
            q->tail = NULL;
        }
    }
    return data;
}
```

參考 chapter 5 投影producer-consumer 結構如下：

```
queue *lines = newqueue();  // shared queue of text lines
queue *dict = newqueue();   // result keyword dictionary
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < n_file; i++) {
        /* Producer */
        produce(lines, filename);

        /* Consumer */
        try_consume(lines, dict);
    }

    /* Consume remaining work */
```

```
    while (!done(lines))
        try_consume(lines, dict);
}
```

程式檔案結構如下，keywords.txt 定義 keyword，每個 keyword 以空格分開、words 是一個目錄，程式會讀取這個目錄中的所有檔案並計算 keyword 的數量：

```
├── h5_problem2.c
├── keywords.txt
└── words
    ├── 1.txt
    ├── 2.txt
    ├── 3.txt
    └── 4.txt
```

keywords.txt 的內容如下：

```
$ cat keywords.txt
this is openmp world hello number
```

透過以下指令編譯執行：

```
$ gcc h5_problem2.c -fopenmp
$ ./a.out
this 1
is 1
openmp 3
world 3
hello 7
number 0
```

使用 `argv[1]` 調整 `n_thread`，例如以下會使用 2 個 thread 執行程式。n_thread 預設為 4。

```
$ ./a.out 2
```

Analysis on your result

我從 nydailynews 複製幾篇報導當作 words，並挑了 20 個 keywords，透過調整 n_thread 並重複執行 10 次來觀察效能，指令如下：

```
perf stat --repeat 10 -e branches,branch-misses,cache-
misses,cycles,instructions,context-switches ./a.out [n_thread]
```

因為有其他作業要弄，就不畫圖了，展示 n_thread 介於 1 到 5 的 perf stat：

```
 Performance counter stats for './a.out 1' (10 runs):

          67,4934      branches
           1,5857      branch-misses          #    2.35% of all branches
           1,1790      cache-misses
         227,3098      cycles
         358,4501      instructions           #    1.58  insn per cycle
                0      context-switches

         0.002631 +- 0.000305 seconds time elapsed  ( +- 11.59% )

 Performance counter stats for './a.out 2' (10 runs):

          70,3717      branches
           1,7578      branch-misses          #    2.50% of all branches
           1,5301      cache-misses
         281,4109      cycles
         371,1494      instructions           #    1.32  insn per cycle
                2      context-switches

         0.001841 +- 0.000359 seconds time elapsed  ( +- 19.49% )

 Performance counter stats for './a.out 3' (10 runs):

          73,1255      branches
           1,8828      branch-misses          #    2.57% of all branches
           1,5823      cache-misses
         390,3688      cycles
         383,4286      instructions           #    0.98  insn per cycle
                4      context-switches

         0.001592 +- 0.000309 seconds time elapsed  ( +- 19.40% )

 Performance counter stats for './a.out 4' (10 runs):

          75,6943      branches
           1,9601      branch-misses          #    2.59% of all branches
           1,5348      cache-misses
         498,0114      cycles
         395,5646      instructions           #    0.79  insn per cycle
                7      context-switches

         0.001226 +- 0.000175 seconds time elapsed  ( +- 14.25% )

 Performance counter stats for './a.out 5' (10 runs):

          78,5179      branches
           2,1349      branch-misses          #    2.72% of all branches
           1,7999      cache-misses
```

```
          566,8570      cycles
          409,2336      instructions              #    0.72  insn per cycle
                10      context-switches

          0.002020 +- 0.000411 seconds time elapsed  ( +- 20.34% )

 Performance counter stats for './a.out 6' (10 runs):

           81,9455      branches
            2,3075      branch-misses             #    2.82% of all branches
            2,5179      cache-misses
          619,3265      cycles
          425,3139      instructions              #    0.69  insn per cycle
                14      context-switches

          0.002488 +- 0.000428 seconds time elapsed  ( +- 17.19% )
```

從實驗數據可以看出 `n_thread` 等於 1 到 4 時執行時間有下降的趨勢，但是 `n_thread` 等於 5、6 時執行時間又突然大幅增加，所以在 OpenMP 中並不是 n_thread 越多就越快，雖然 OpenMP 可以簡單的用 pragma 來撰寫，但背後的機制不像 pthread 可以讓程式開發者隨心所欲的控制。

context-switches、instructions、branch-misses 隨著 n_thread 增加而增加。insn per cycle 隨著 n_thread 增加而減少。

Any difficulites?

原本使用 `strtok` 來 tokenize line，但是發 `strtok` 的 pointer 是共用的，在平行化的程式會出錯，所以改用 while 迴圈實作。