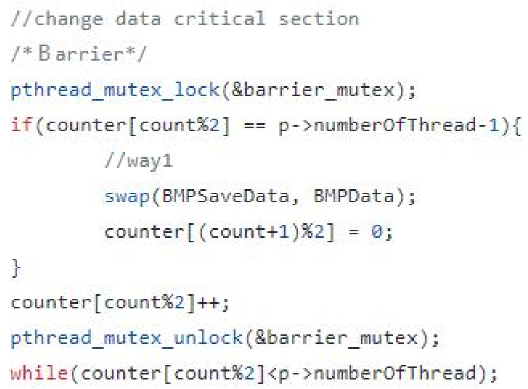
多處理機平行程式設計 作業四 F74109016 葉惟欣

一.What have you done?

這次的作業是在做 smooth，有三種 barrier 的其中一種來實現，為了想要了解每一個我嘗試了三種方法。

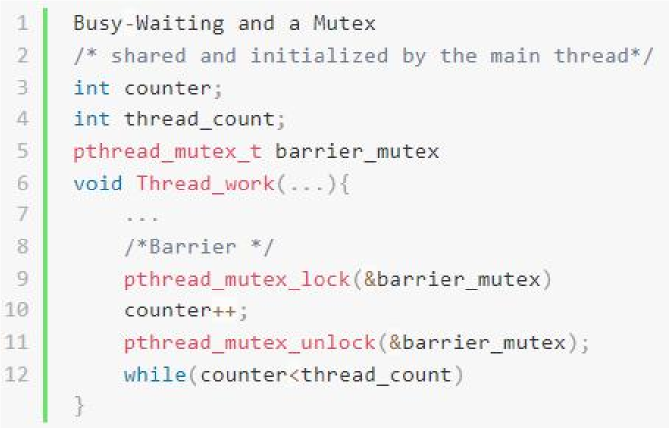
1. Busy Waiting And Mutex



```
//change data critical section
/*Barrier*/
pthread_mutex_lock(&barrier_mutex);
if(counter[count%2] == p->numberOfThread-1){
        //way1
        swap(BMPSaveData, BMPData);
        counter[(count+1)%2] = 0;
}
counter[count%2]++;
pthread_mutex_unlock(&barrier_mutex);
while(counter[count%2]<p->numberOfThread);
```

在最後一個 thread 要進來的時候才做 swap，且將下回的 counter[(count+1)%2] 初始化為 0。**這是用奇數與偶數的方法，來讓每一回合的 counter 是不一樣的**而每一次都只有一個 thread 能夠進入 pthread_mutex_lock 與 pthread_mutex_unlock 中間的區域。在那個 critical section 區域做完事後 unlock 讓下一個 thread 能夠去搶進入 critical section。

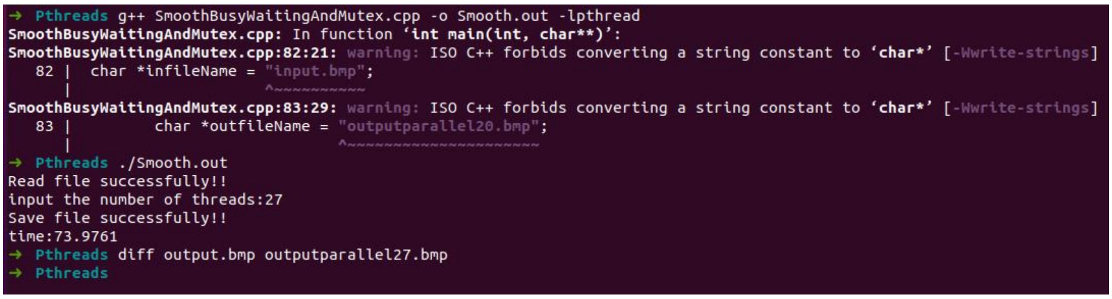
之後如果最後一個 thread 還沒有近來就在 while 那個地方等待，所有 thread 停在 while 那裏直到大家都執行到那行，也就是最後一個 thread 也做完 counter[count%2]++。



```
Busy-Waiting and a Mutex
/* shared and initialized by the main thread*/
int counter;
int thread_count;
pthread_mutex_t barrier_mutex
void Thread_work(...){
    ...
    /*Barrier */
    pthread_mutex_lock(&barrier_mutex)
    counter++;
    pthread_mutex_unlock(&barrier_mutex);
    while(counter<thread_count)
}
```

原本是只用 pthread_mutex_lock 去保護 shared variable “counter”，而這樣的作法在 second barrier 的時候 counter 的值已經為 thread_count，除非我們將 counter reset 為 0，不然 while 迴圈依然會有問題。但是 reset 為 0 也會有問題。如果最後一個 thread 在進入迴圈前試著將 counter reset 為 0，則有

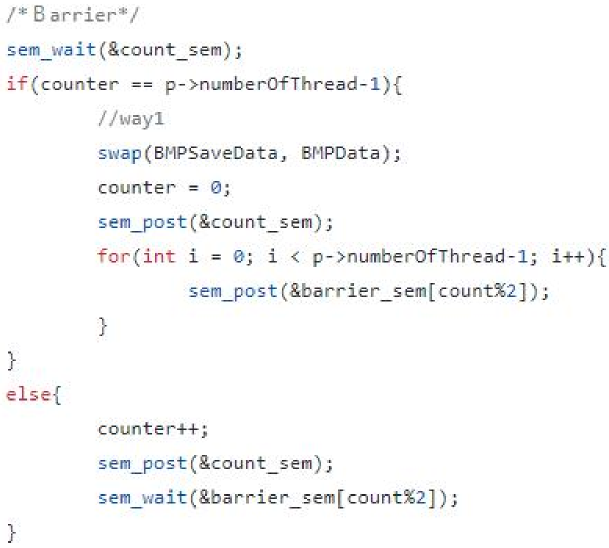
些 thread 從頭到尾無法看到 counter == thread_count 發生(相當於不會離開迴圈)。而如果有 thread 試著在 barrier 後 reset counter 為 0，則有些 thread 會在 counter reset 前 進入 second barrier，則他對 counter 做加 1 時的動作則會在 second barrier 沒有效果。如此一來所有的 thread 都會停在 second busy-wait loop。這也是為什麼我們在用每個 barrier，都需要一個 counter 變數的原因。



```
➜  Pthreads g++ SmoothBusyWaitingAndMutex.cpp -o Smooth.out -lpthread
SmoothBusyWaitingAndMutex.cpp: In function ‘int main(int, char**)’:
SmoothBusyWaitingAndMutex.cpp:82:21: warning: ISO C++ forbids converting a string constant to ‘char*’ [-Wwrite-strings]
   82 |  char *infileName = "input.bmp";
      |                     ^~~~~~~~~~~
SmoothBusyWaitingAndMutex.cpp:83:29: warning: ISO C++ forbids converting a string constant to ‘char*’ [-Wwrite-strings]
   83 |         char *outfileName = "outputparallel20.bmp";
      |                             ^~~~~~~~~~~~~~~~~~~~~~
➜  Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:27
Save file successfully!!
time:73.9761
➜  Pthreads diff output.bmp outputparallel27.bmp
➜  Pthreads
```

diff 後結果正確。

2. Semaphore



```
/*Barrier*/
sem_wait(&count_sem);
if(counter == p->numberOfThread-1){
        //way1
        swap(BMPSaveData, BMPData);
        counter = 0;
        sem_post(&count_sem);
        for(int i = 0; i < p->numberOfThread-1; i++){
                sem_post(&barrier_sem[count%2]);
        }
}
else{
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem[count%2]);
}
```

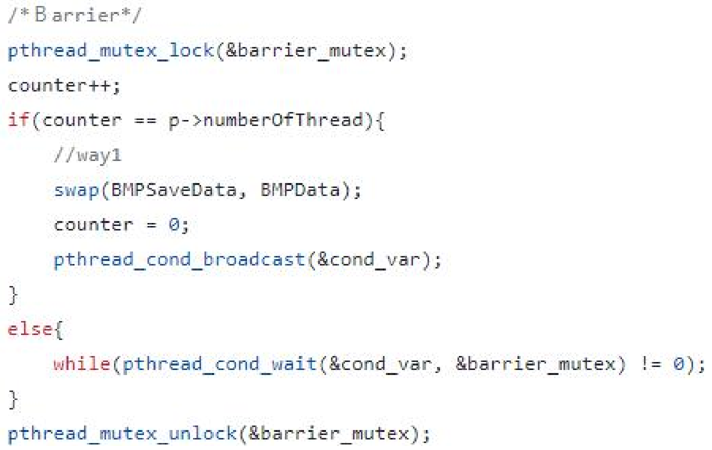
用兩個 semaphores : count_sem 用來保護 counter，然後用 barrier_sem 用來阻擋(block)threads 進入 barrier。 Count_sem 初始化為 1(unlock)，所以當第一個 thread 進到 barrier，他會 call sem_wair()來 block 其他的 thread，透過將 count_sem-1 變為 0，來讓其他 thread wait 在 barrier 外。而唯一一個進入 barrier 的 thread 就會去確認現在進到 barrier 的數量是否已經為總 thread 數量。

如果沒有則該 thread 將數字加一，且釋放 semaphore 然後 block 在 sem_wait。

如果是，則做最後一個 thread 該做的事情，將 counter 初始化，然後釋放 count_sem，且將釋放 sem_post 要 thread 的數量-1 次，因為每個有 thread 的數量-1 先前 wait，代表他們將 barrier_sem 減了 thread 的數量-1，現在透過 sem_post 加回去。

3.Condition Variable

A condition variable is a data object that allows a thread to suspend execution until a certain event or condition occurs.

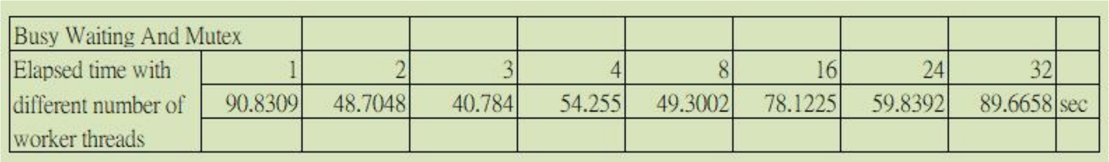


```
/*Barrier*/
pthread_mutex_lock(&barrier_mutex);
counter++;
if(counter == p->numberOfThread){
    //way1
    swap(BMPSaveData, BMPData);
    counter = 0;
    pthread_cond_broadcast(&cond_var);
}
else{
    while(pthread_cond_wait(&cond_var, &barrier_mutex) != 0);
}
pthread_mutex_unlock(&barrier_mutex);
```

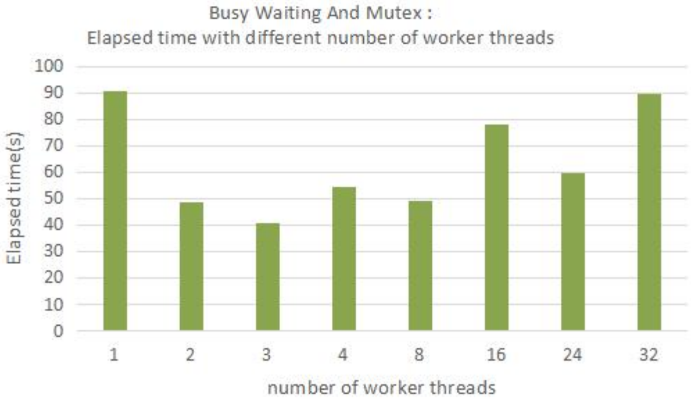
pthread_cond_wait() 用於阻塞當前線程，等待別的線程使用 pthread_cond_signal() 或 pthread_cond_broadcast 來喚醒它。 pthread_cond_wait() 必須與 pthread_mutex 配套使用。pthread_cond_wait() 函數一進入 wait 狀態就會**自動 release mutex**。**(所以當作到 while(pthread_cond_wait)時會釋放 barrier_mutex 來讓下一個 thread 可以得到 barrier_mutex 進入 critical section)**當其他線程通過 pthread_cond_signal() 或 **pthread_cond_broadcast** ，把該線程喚醒，使 pthread_cond_wait()通過（返回）時，該線程又自動獲得該 mutex 。

二.Analysis on your result。

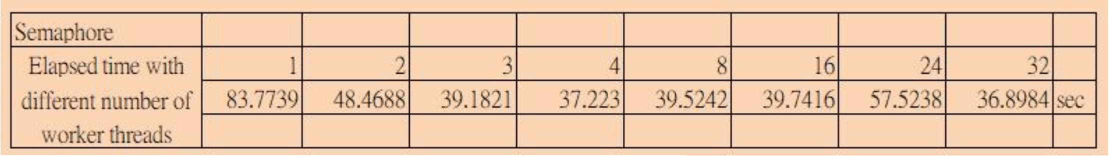
Busy wait and a mutex



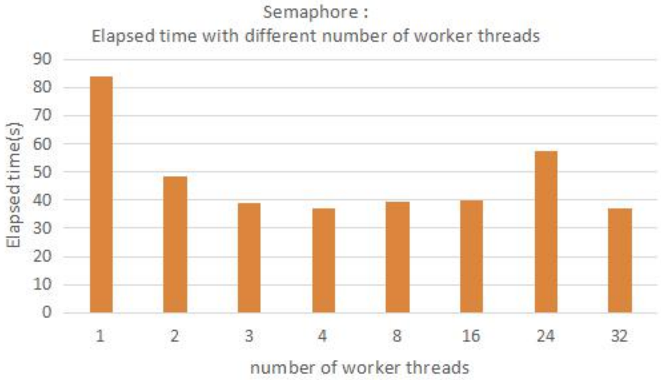
Busy Waiting And Mutex									
Elapsed time with	1	2	3	4	8	16	24	32	
different number of	90.8309	48.7048	40.784	54.255	49.3002	78.1225	59.8392	89.6658	sec
worker threads									



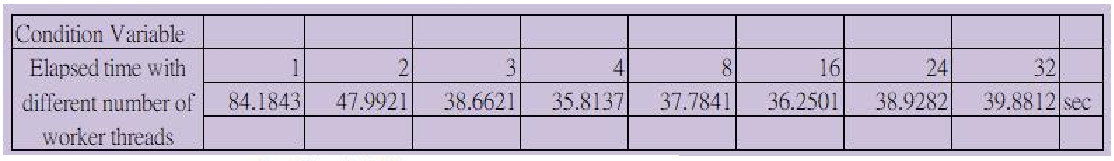
Busy Waiting And Mutex :
Elapsed time with different number of worker threads



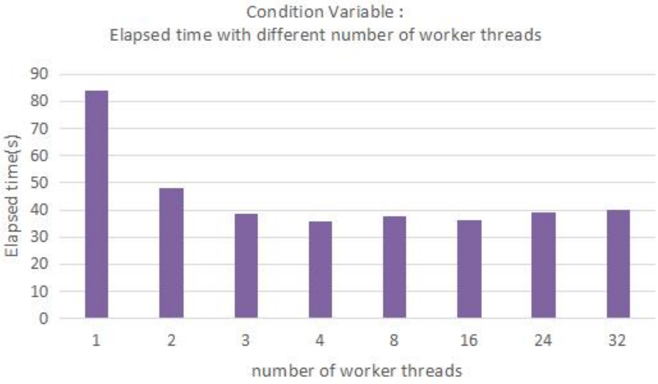
Semaphore									
Elapsed time with	1	2	3	4	8	16	24	32	
different number of	83.7739	48.4688	39.1821	37.223	39.5242	39.7416	57.5238	36.8984	sec
worker threads									



Semaphore :
Elapsed time with different number of worker threads



Condition Variable									
Elapsed time with	1	2	3	4	8	16	24	32	
different number of	84.1843	47.9921	38.6621	35.8137	37.7841	36.2501	38.9282	39.8812	sec
worker threads									



Condition Variable :
Elapsed time with different number of worker threads

分析一:

可以看到 Busy wait and a mutex 的實作會較花時間，因為會浪費 CPU cycle 在 busy-wait loop。而 semaphore 跟 Condition Variable 不用一直 waste CPU cycle 一直去檢查 counter == thread_count，所以花的時間普遍比較少。

分析二:

在 Busy wait and a mutex 中，隨著 thread 數量增加並沒有看到用時的減少，反而到最後趨近於跟循序相同，我認為可能的原因如下:

因為每個 thread 都要去檢查 counter == thread_count，這樣在我用虛擬機 4 個 core 的情況下。當 thread 的數量超過 core 時，每個 thread 在執行部分時間就要將 cpu 的 control 交給其他 thread。而這個交換稱為 context switch。Context switch is not free。必須要花時間儲存 CPU 的狀態，包刮暫存器的值得保存與更新新的暫存器的值，Program counter 的切換...等。這些都需要花時間。這樣的 overhead 可能 將 多平行處理帶來的效益降低，甚至超過，所以才有上述的現象發生。

分析三:

Semaphore 與 Condition variable 在 core 的數量小於 thread 的數量時，隨著 thread 增加，elapsed time 減少的幅度的變化量比較大。在 test1 這個減少的幅度還是線性減少的。

因為 Smooth 屬於 CPU bound 的 task，透過 multi thread 主要是提升 CPU 的使用率。對於虛擬機的 4core 來說，其實用 4 個 thread 就可達到很好的效果讓一個 thread 使用 1 個 core。所以我們也看到 4 個 test 執行時間最短的都大概為 4 個 thread。

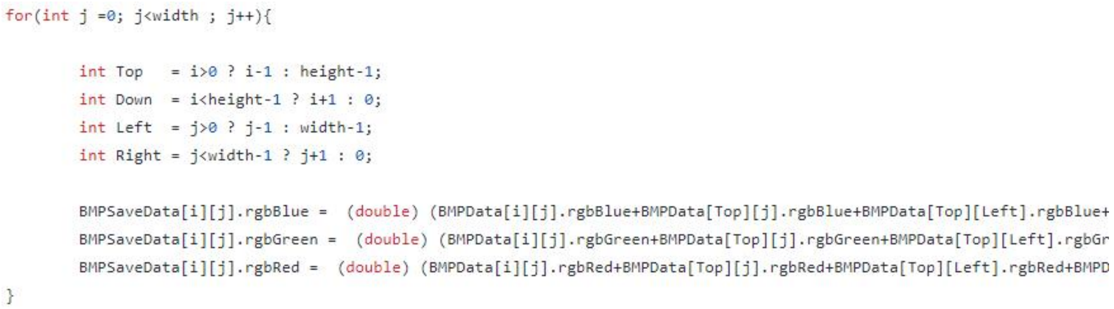
三.Any difficulties?

在這次作業我原先是將所有兩個矩陣做交換的部分，不是透過直接交換指標，而是在每次做完 smooth 後將新的資料直接 assign 給舊的資料。

法一:直接交換指標

讓兩個陣列都同步更新為新的 smooth 結果後，在去做下一次 smooth 的過程。

smooth 過程如下:

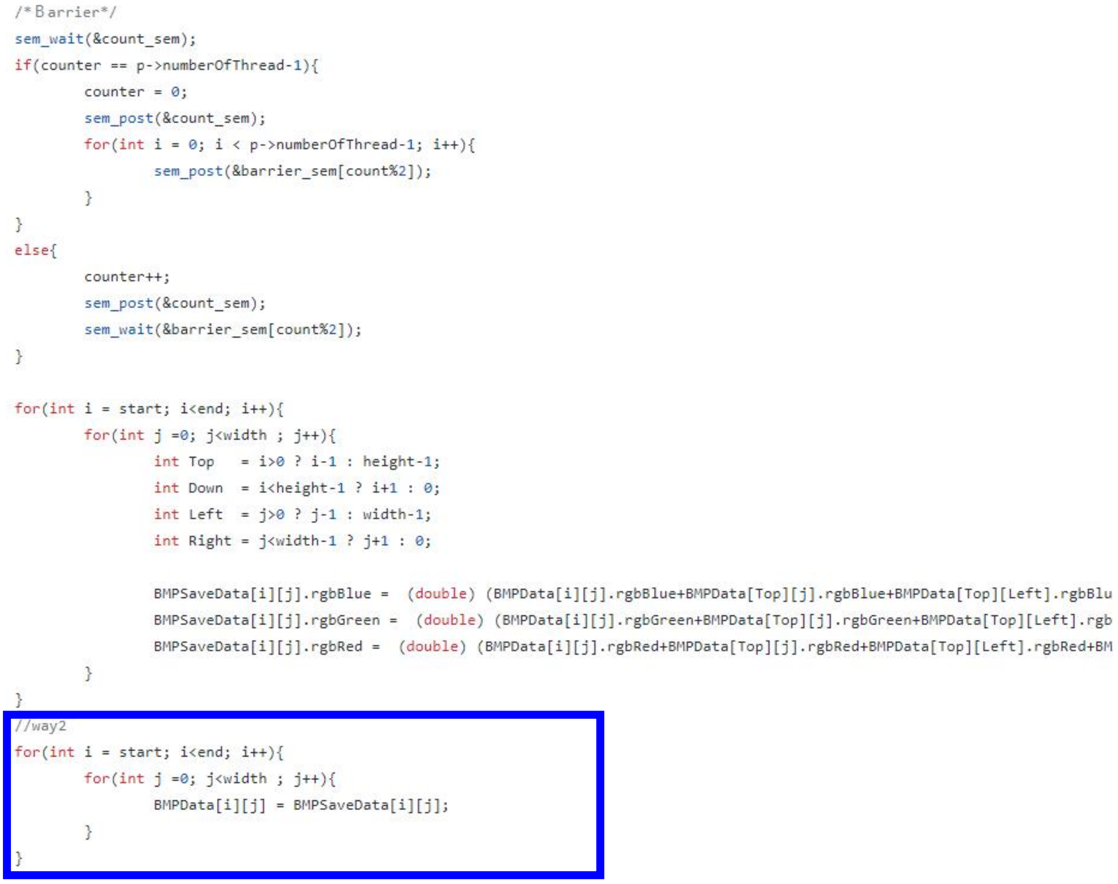


```
for(int j =0; j<width ; j++){

        int Top   = i>0 ? i-1 : height-1;
        int Down  = i<height-1 ? i+1 : 0;
        int Left  = j>0 ? j-1 : width-1;
        int Right = j<width-1 ? j+1 : 0;

        BMPSaveData[i][j].rgbBlue =  (double) (BMPData[i][j].rgbBlue+BMPData[Top][j].rgbBlue+BMPData[Top][Left].rgbBlue+
        BMPSaveData[i][j].rgbGreen =  (double) (BMPData[i][j].rgbGreen+BMPData[Top][j].rgbGreen+BMPData[Top][Left].rgbGr
        BMPSaveData[i][j].rgbRed =  (double) (BMPData[i][j].rgbRed+BMPData[Top][j].rgbRed+BMPData[Top][Left].rgbRed+BMPD
}
```

也就是拿 BMPData 去做 smooth 後 assign 給 BMPSaveData，最後 BMPSaveData ，如果還有在做 Smooth 再將兩個指標做交換，讓 BMPData 用上次 smooth 完的結果去做新的 Smooth。

法二:

這是我原先嘗試的作法，是在每次 Smooth 完後將剛剛 Smooth 的結果再 assign 給 BMPData 讓下一次做 smooth 的時候可以用新 update 的結果，而這裡是每個 thread 只做自己的部分 from start to end 負責區域的值的更新。



```
/*Barrier*/
sem_wait(&count_sem);
if(counter == p->numberOfThread-1){
        counter = 0;
        sem_post(&count_sem);
        for(int i = 0; i < p->numberOfThread-1; i++){
                sem_post(&barrier_sem[count%2]);
        }
}
else{
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem[count%2]);
}

for(int i = start; i<end; i++){
        for(int j =0; j<width ; j++){
                int Top   = i>0 ? i-1 : height-1;
                int Down  = i<height-1 ? i+1 : 0;
                int Left  = j>0 ? j-1 : width-1;
                int Right = j<width-1 ? j+1 : 0;

                BMPSaveData[i][j].rgbBlue =  (double) (BMPData[i][j].rgbBlue+BMPData[Top][j].rgbBlue+BMPData[Top][Left].rgbBlu
                BMPSaveData[i][j].rgbGreen =  (double) (BMPData[i][j].rgbGreen+BMPData[Top][j].rgbGreen+BMPData[Top][Left].rgb
                BMPSaveData[i][j].rgbRed =  (double) (BMPData[i][j].rgbRed+BMPData[Top][j].rgbRed+BMPData[Top][Left].rgbRed+BM
        }
}
//way2
for(int i = start; i<end; i++){
        for(int j =0; j<width ; j++){
                BMPData[i][j] = BMPSaveData[i][j];
        }
}
```

方法二的結果照理來說應該要是對的，但很奇怪的是用 sequential 的結果跟 diff 結果不一樣。關於這個問題我還在思考是甚麼原因。

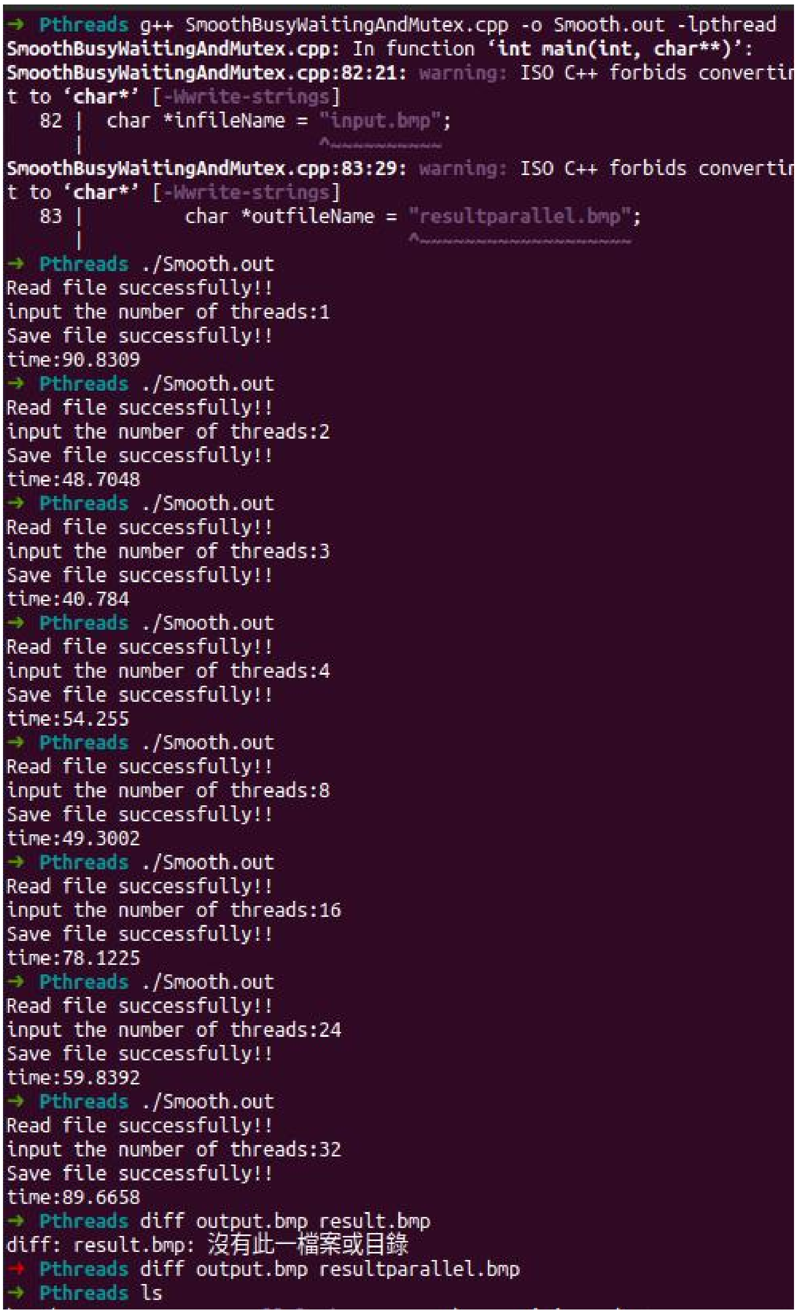
四.(optional) Feedback to TAs

這次的作業因為我不是很懂三個的差別，所以將三個都實作了，為了瞭解其中的機制去看了原文書，想把模糊的地方搞懂，雖然還是有些不懂的地方，但覺得原文書真的寫得很好，把

五.結果

Busy Waiting And Mutex

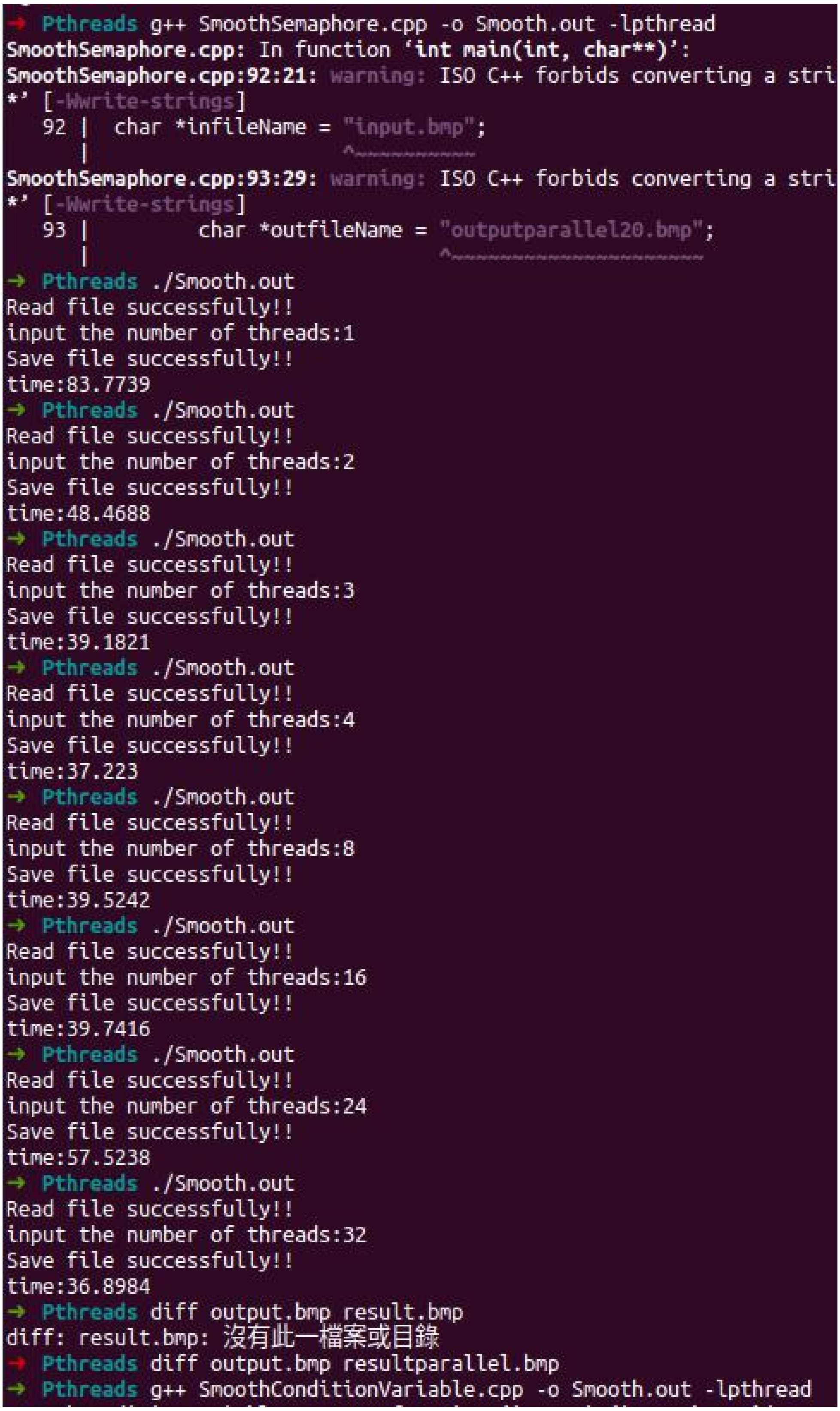
Diff 結果正確



```
➜  Pthreads g++ SmoothBusyWaitingAndMutex.cpp -o Smooth.out -lpthread
SmoothBusyWaitingAndMutex.cpp: In function ‘int main(int, char**)’:
SmoothBusyWaitingAndMutex.cpp:82:21: warning: ISO C++ forbids convertin
t to ‘char*’ [-Wwrite-strings]
   82 |  char *infileName = "input.bmp";
      |                     ^~~~~~~~~~~
SmoothBusyWaitingAndMutex.cpp:83:29: warning: ISO C++ forbids convertin
t to ‘char*’ [-Wwrite-strings]
   83 |         char *outfileName = "resultparallel.bmp";
      |                             ^~~~~~~~~~~~~~~~~~~~
➜  Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:1
Save file successfully!!
time:90.8309
➜  Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:2
Save file successfully!!
time:48.7048
➜  Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:3
Save file successfully!!
time:40.784
➜  Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:4
Save file successfully!!
time:54.255
➜  Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:8
Save file successfully!!
time:49.3002
➜  Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:16
Save file successfully!!
time:78.1225
➜  Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:24
Save file successfully!!
time:59.8392
➜  Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:32
Save file successfully!!
time:89.6658
➜  Pthreads diff output.bmp result.bmp
diff: result.bmp: 沒有此一檔案或目錄
➜  Pthreads diff output.bmp resultparallel.bmp
➜  Pthreads ls
```

Semaphore

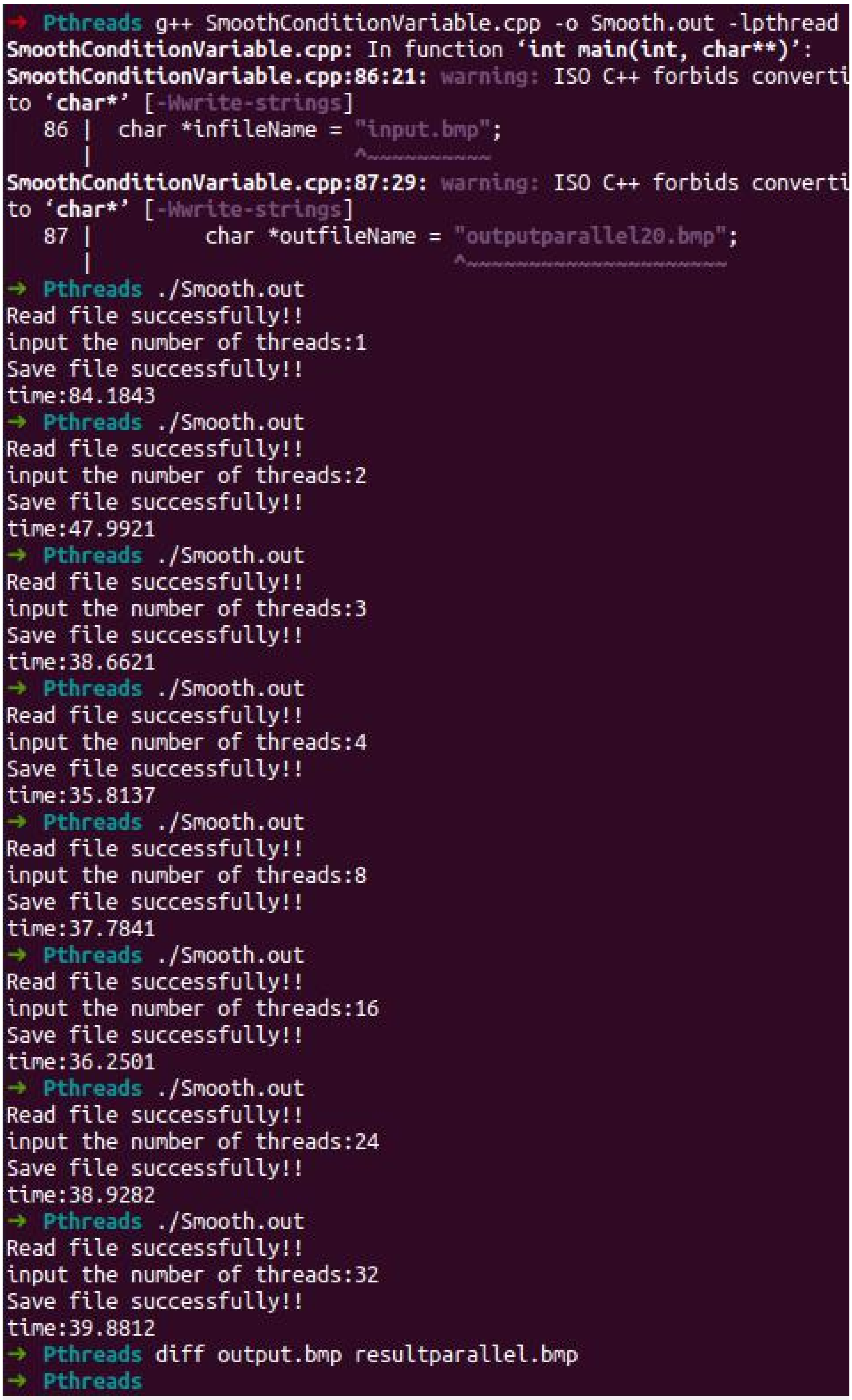
Diff 結果正確



```
➜ Pthreads g++ SmoothSemaphore.cpp -o Smooth.out -lpthread
SmoothSemaphore.cpp: In function ‘int main(int, char**)’:
SmoothSemaphore.cpp:92:21: warning: ISO C++ forbids converting a stri
*’ [-Wwrite-strings]
   92 |  char *infileName = "input.bmp";
      |                     ^~~~~~~~~~~
SmoothSemaphore.cpp:93:29: warning: ISO C++ forbids converting a stri
*’ [-Wwrite-strings]
   93 |         char *outfileName = "outputparallel20.bmp";
      |                             ^~~~~~~~~~~~~~~~~~~~~~
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:1
Save file successfully!!
time:83.7739
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:2
Save file successfully!!
time:48.4688
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:3
Save file successfully!!
time:39.1821
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:4
Save file successfully!!
time:37.223
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:8
Save file successfully!!
time:39.5242
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:16
Save file successfully!!
time:39.7416
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:24
Save file successfully!!
time:57.5238
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:32
Save file successfully!!
time:36.8984
➜ Pthreads diff output.bmp result.bmp
diff: result.bmp: 沒有此一檔案或目錄
➜ Pthreads diff output.bmp resultparallel.bmp
➜ Pthreads g++ SmoothConditionVariable.cpp -o Smooth.out -lpthread
```

Condition Variable

Diff 結果正確



```
➜ Pthreads g++ SmoothConditionVariable.cpp -o Smooth.out -lpthread
SmoothConditionVariable.cpp: In function ‘int main(int, char**)’:
SmoothConditionVariable.cpp:86:21: warning: ISO C++ forbids converti
to ‘char*’ [-Wwrite-strings]
   86 |  char *infileName = "input.bmp";
      |                     ^~~~~~~~~~~
SmoothConditionVariable.cpp:87:29: warning: ISO C++ forbids converti
to ‘char*’ [-Wwrite-strings]
   87 |         char *outfileName = "outputparallel20.bmp";
      |                             ^~~~~~~~~~~~~~~~~~~~~~
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:1
Save file successfully!!
time:84.1843
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:2
Save file successfully!!
time:47.9921
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:3
Save file successfully!!
time:38.6621
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:4
Save file successfully!!
time:35.8137
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:8
Save file successfully!!
time:37.7841
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:16
Save file successfully!!
time:36.2501
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:24
Save file successfully!!
time:38.9282
➜ Pthreads ./Smooth.out
Read file successfully!!
input the number of threads:32
Save file successfully!!
time:39.8812
➜ Pthreads diff output.bmp resultparallel.bmp
➜ Pthreads
```