SP Programming HW4 Report (Student ID: B09902046)

Question.1 比較利用 2 個 thread 與 20 個 thread 在 large test case 的執行時間，並給出自己的評論。

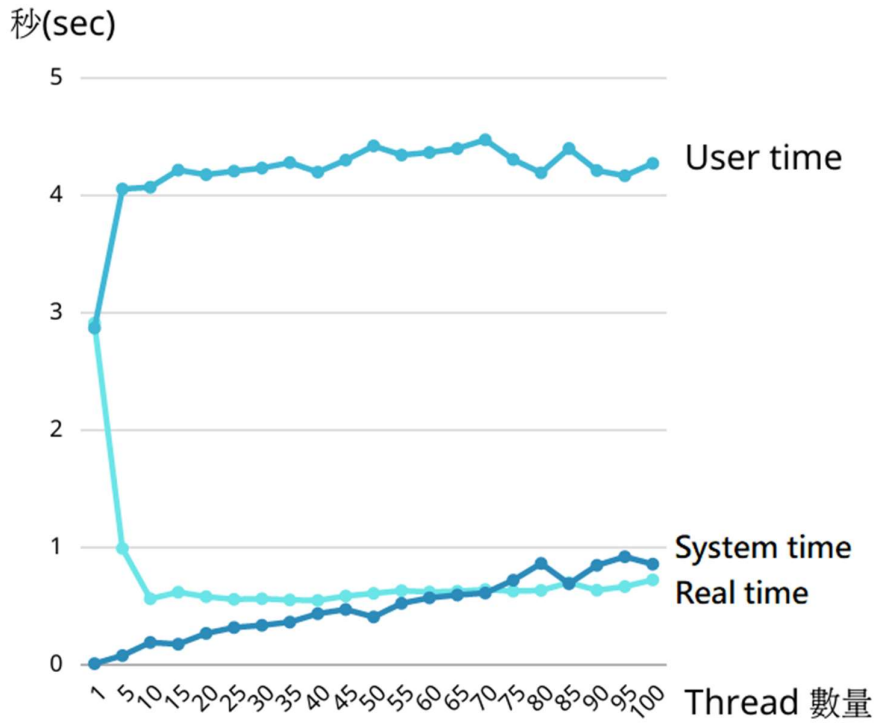
Thread 數量	Real time	User time	System time
2 threads	1.678s	3.078s	0.031s
20 threads	0.570s	4.271s	0.193s

Real time：若有較多 thread，可以讓每個 thread 所需要執行的工作量下降，並且進行更多的平行運算，故 20 threads 的 real time 就比 2 threads 的短。

User time：由於一開始只有 2 個 threads 時，在我的程式碼內，畫分工區塊的時間(user space)就會比較少，故 user time 會比 20 個 threads 時還要少。

System time：由於重複 create thread 與 join thread 需要動用許多 system call(本人作法)，故創建較多 thread 會導致 system time 的增加。

Question.2 利用 large test case 來去計算，當擁有 threads(= 1,5,10,..,100) 來執行時，所需要的執行時間，且畫出折線圖並給出自己的評論。



(Real time 為淡藍色，System time 為深藍色，User time 為藍色)

Real time：從 Real time 可觀察到，一開始當 thread 的數量增加時，Real time 的時間有很明顯的下降，因為有更多的 thread 來幫忙分擔工作量，進行平行運

算。但隨著 thread 開始增多，system time 會跟著增加，由於 threads 數量比 CPU 核心多太多的關係，就開始有許多的 context-switch overhead，real time 也隨之微幅上升。

System time：也因此，System time 的趨勢為，隨著 thread 的增多，system time 會越來越增加。因為 create 與 join thread 需要 system call，因此隨著需要製造跟等待的 thread 變多，context-switch overhead 也開始增加(見上段)，system time 也會跟著隨之增加，故 system time 是有越來越多的趨勢。

User time : 而 user time 之所以會在後期都一直呈現平緩，且微幅震盪的趨勢，就如同助教在討論區上所講的，是因為 system 沒辦法供應那麼多的 kernel level thread，才會導致 user time 出現這樣的情況。

Question.3 請寫出自己程式中的 critical parts。

以下主要分為分配 thread 工作區段的方法、每個 thread 要執行的內容以及如何運作各個 epoch 等三大部分來進行講解。

1. 分配 thread 工作區段的方法

```
int row_per = row/threadcount,rowmod = row%threadcount;
int Rowpart[threadcount][2];
if (row_per == 0){
    realworkingthreadcount = row;
    for (int i = 0; i < realworkingthreadcount; i++){
        Rowpart[i][0] = i+1; Rowpart[i][1] = i+1;
    }
    for (int i = realworkingthreadcount; i < threadcount; i++){
        Rowpart[i][0] = 2; Rowpart[i][1] = 1;
    }
}
else{
    realworkingthreadcount = threadcount;
    int modcount = 0;
    int acount = 1;
    for (int i = 0; i < threadcount; i++){
        if (modcount < rowmod){
            Rowpart[i][0] = acount; Rowpart[i][1] = acount+row_per;
            modcount++;
            acount += row_per+1;
        }
        else{
            Rowpart[i][0] = acount; Rowpart[i][1] = acount+row_per-1;
            acount += row_per;
        }
    }
}
for (int i = 0; i < threadcount; i++){
    workingla[i].begincolumn = 1;
    workingla[i].endcolumn = column;
    workingla[i].beginrow = Rowpart[i][0];
    workingla[i].endrow = Rowpart[i][1];
}
```

以上程式碼為當 row 數量>column 數量時的切法，若 row 數量<= column 數量時，就把上述 row 與 column 的關係對調即可。大致為將照 row/threads 的數量來分配每個 thread 要執行的 row 數，再加上考慮餘數的狀況。再將 thread 的工作區段，存進 workingla 的 struct 裡面，紀錄起始與終止的 row 與 column。

2. 每個 thread 要執行的內容

```
void* dealthread(void* vars){
    Workpart *needtowork = (Workpart*)vars;
    for (int i = needtowork->beginrow; i <= needtowork->endrow; i++){
        for (int j = needtowork->begincolumn; j <= needtowork->endcolumn; j++){
            int count = 0;
            for (int a = i-1; a <= i+1; a++){
                for (int b = j-1; b <= j+1; b++){
                    if (!(a == i && b == j)){
                        if (plate[a][b] == true) count++;
                    }
                }
            }
            if (plate[i][j] == true){ // live
                if (count == 2 || count == 3){
                    duplicateplate[i][j] = true;
                }
                else duplicateplate[i][j] = false;
            }
            else { //die
                if (count == 3){
                    duplicateplate[i][j] = true;
                }
                else duplicateplate[i][j] = false;
            }
        }
    }
    pthread_exit(NULL);
}
```

在我程式碼內，會在每個 epoch 重新 create 一個 thread，故每個 thread 只需要去更新自己工作區段內的 cell，且只需要考慮一個 epoch 即可。因此跑一個起始與終止點分別為 row 與 column 的雙重迴圈，再去統計周圍的 cell 是否存活，再根據自身 cell 為 live 還是 die，以及周圍有幾個 cell 存活的情況，將該 cell 要改變成的生命狀態，存到 duplicateplate 同樣的位置裏頭。

3. 如何運作各個 epoch

```
if (isthread == true){
    pthread_t threadarr[threadcount];
    for (int i = 0; i < epoch; i++){
        for (int j = 0; j < threadcount; j++){
            pthread_create(&threadarr[j],NULL,dealthread,(void*)&workingla[j]);
        }
        for (int j = 0; j < threadcount; j++){
            pthread_join(threadarr[j],NULL);
        }
        bool **tempplate;
        tempplate = plate;
        plate = duplicateplate;
        duplicateplate = tempplate;
    }
}
```

每個 epoch 會先重新 create thread，並讓每個 thread 去更新那些該完成的工作區段(上段內容)，之後再 join thread，確認每個 thread 的工作均已結束。之後我們準備一個 tempplate 的指標，使原本存到 duplicateplate 的新狀態，能夠轉移到本來的 plate，也就是 swap(plate,duplicateplate)的感覺。