

Group 26

組員：楊峻銘，洪浚凱，簡佩如

Outline

1. 題目敘述
2. 方法一：Pthread
3. 方法二：MPI
4. 方法三：GPU
5. 實驗設置
6. 結論

題目敘述

給定一個 $m \times n$ 的網格，其中每個格子可以有以下三種狀態之一：

- 0 表示空格子；
- 1 表示新鮮橘子；
- 2 表示壞掉的橘子。

每分鐘，任何與壞橘子四個方向相鄰的新鮮橘子都會被感染變壞。

計算所有橘子變壞所需的最少時間（以分鐘為單位）。如果有橘子無法變壞，回傳 -1。

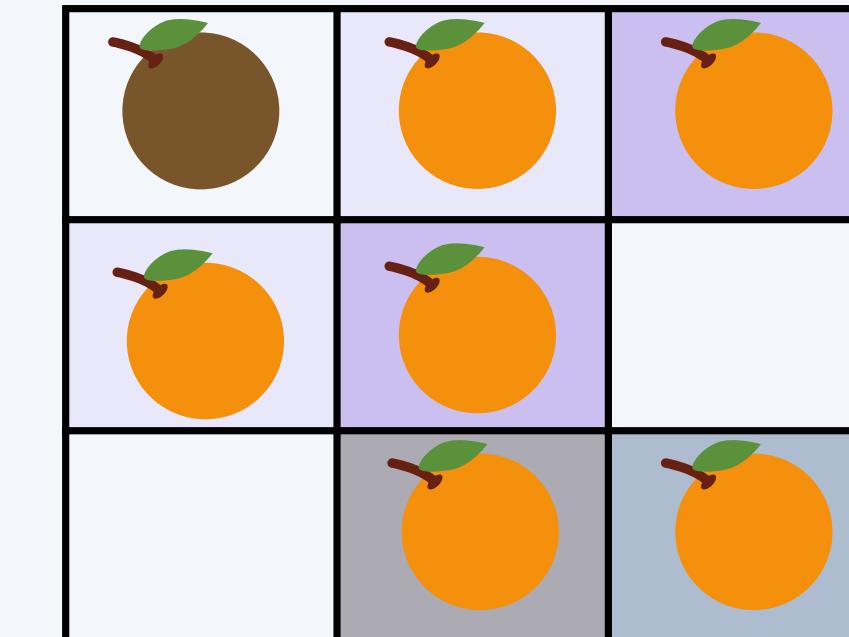
範例：

Input: grid = [[2,1,1],[1,1,0],[0,1,1]]

Output: 4

Input: grid = [[2,1,1],[0,1,1],[1,0,1]]

Output: -1



Sequential 解決方法

這題可以用 BFS (Breadth-first search) 解決，以下是步驟：

1. 初始化：

- 掃描整個網格，找到所有腐爛橘子的座標，將它們加入 BFS 的 queue 中。
- 記錄初始新鮮橘子的數量 (fresh_count)。

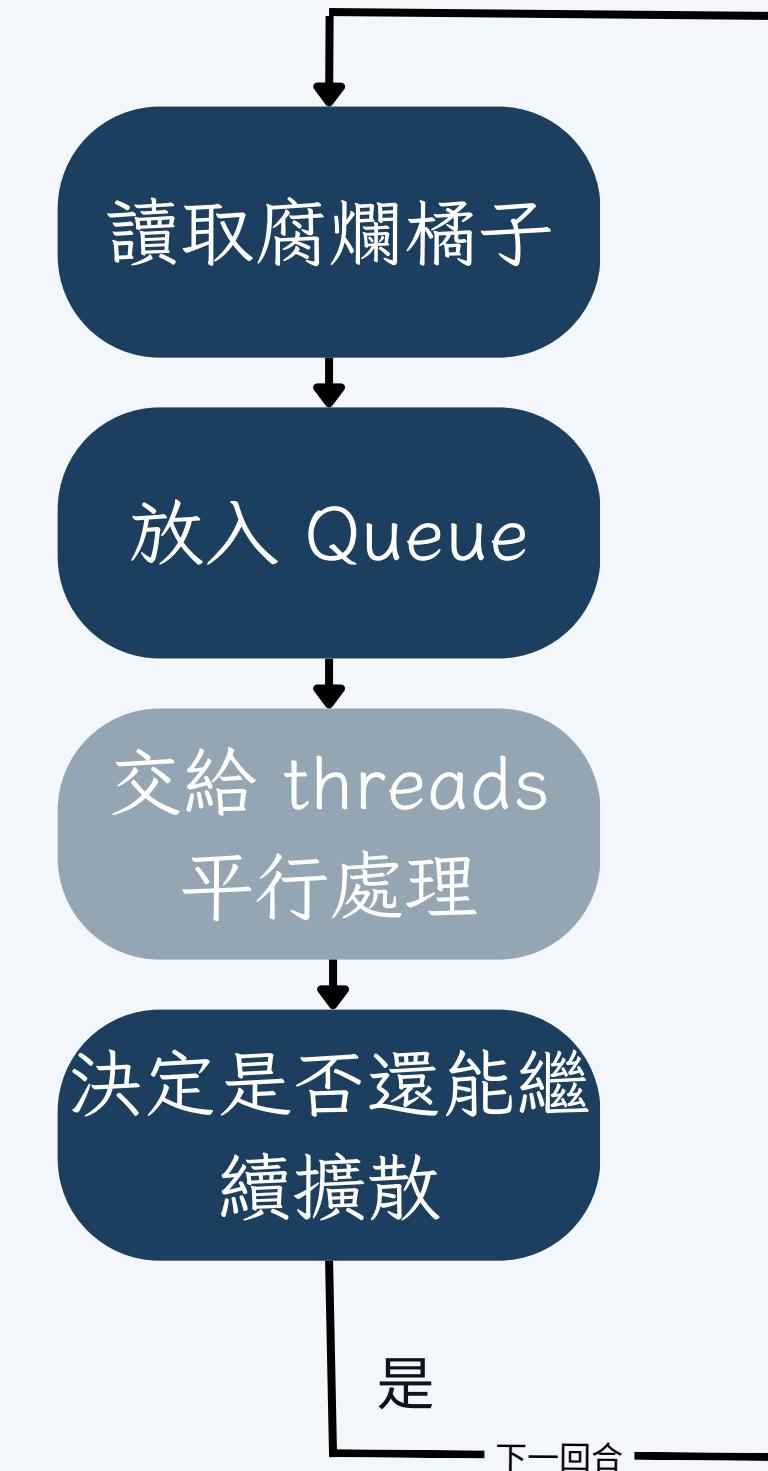
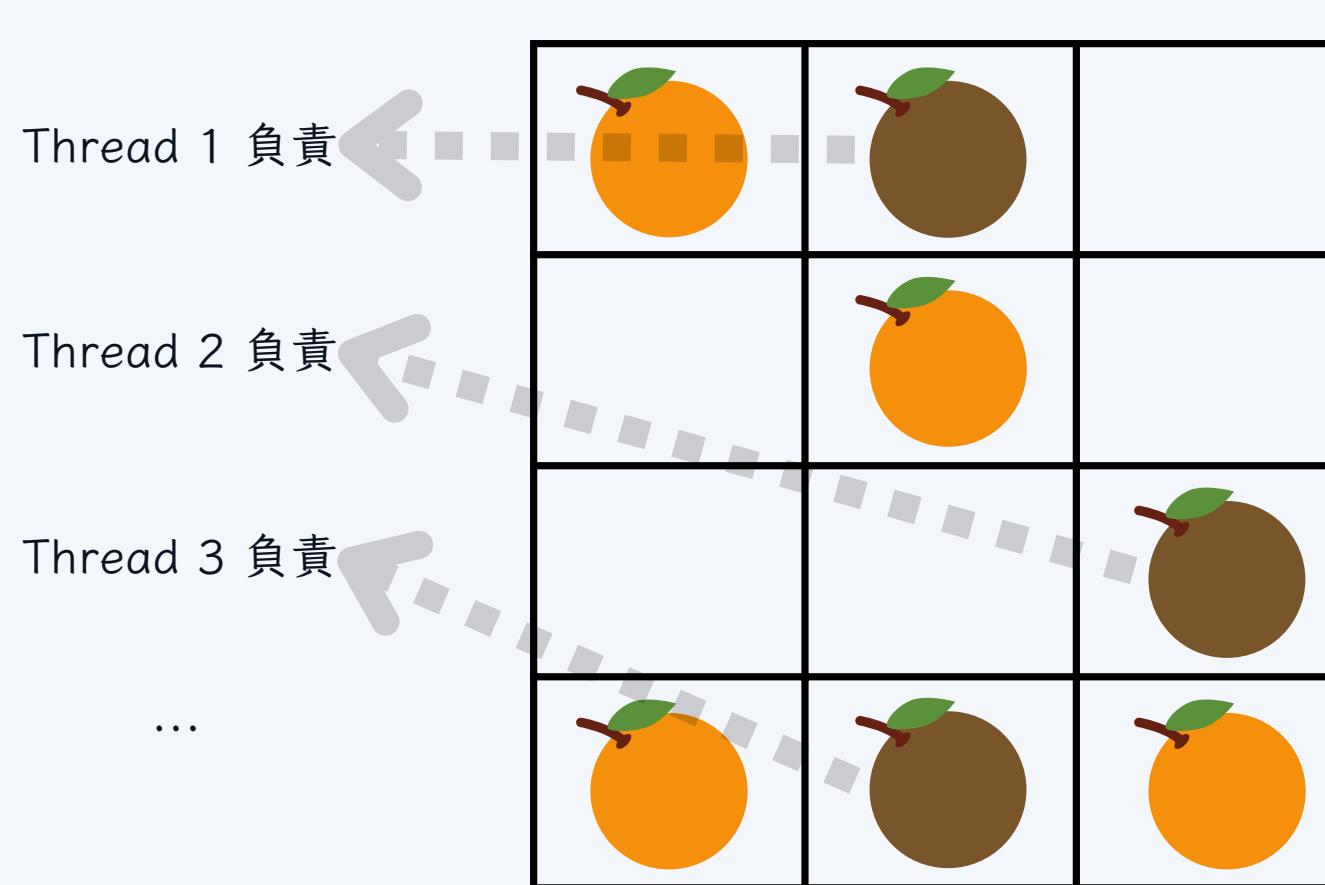
2. 執行 BFS：

- 從 queue 中取出一個腐爛橘子，檢查它的上下左右相鄰格子：
 - 如果相鄰格子是新鮮橘子 (1)，則將該橘子腐爛 (2)，並加入 queue，同時減少新鮮橘子的計數 (fresh_count 減 1)。
- 每進行一層 BFS，計數時間增加 1 分鐘。

3. 檢查結果：

- 如果 BFS 結束時， $\text{fresh_count} > 0$ ，表示有些橘子無法腐爛，返回 -1。
- 否則，返回腐爛所需的時間。

方法一：Pthread 平行思路



思考：

1. 如果每個 thread 都要讀寫 queue、fresh_count，則容易發生 race condition。
2. 在每一回合都做 pthread_create 跟 pthread_join的話，對於大量的 threads 會形成負擔。

方法一：Pthread 平行策略

Master (Thread 0) 負責整體流程管理：

- 監控腐爛進度
- 分配「腐爛橘子」的任務到共享 queue
- 動態分配：每一回合，根據腐爛橘子的位置，把工作分給 threads。

Slave (其他執行緒) 負責接收任務：

- 從共享 queue 中取出腐爛橘子座標。
- 處理相鄰格子（將新腐爛的橘子加入佇列）。
- 專注執行：每個執行緒只需專心處理當前任務。

方法一：Pthread 同步與資源管理

每一回合都創建 threads:

- 每一輪進行時 `pthread_create`。
- 當一輪完成後 `pthread_join`。
- 每次都需要執行緒的創建與回收。
- 同步效率低：依賴 `pthread_join`，無法高效管理多個 pthread 的資源。

Master-Slave 模式:

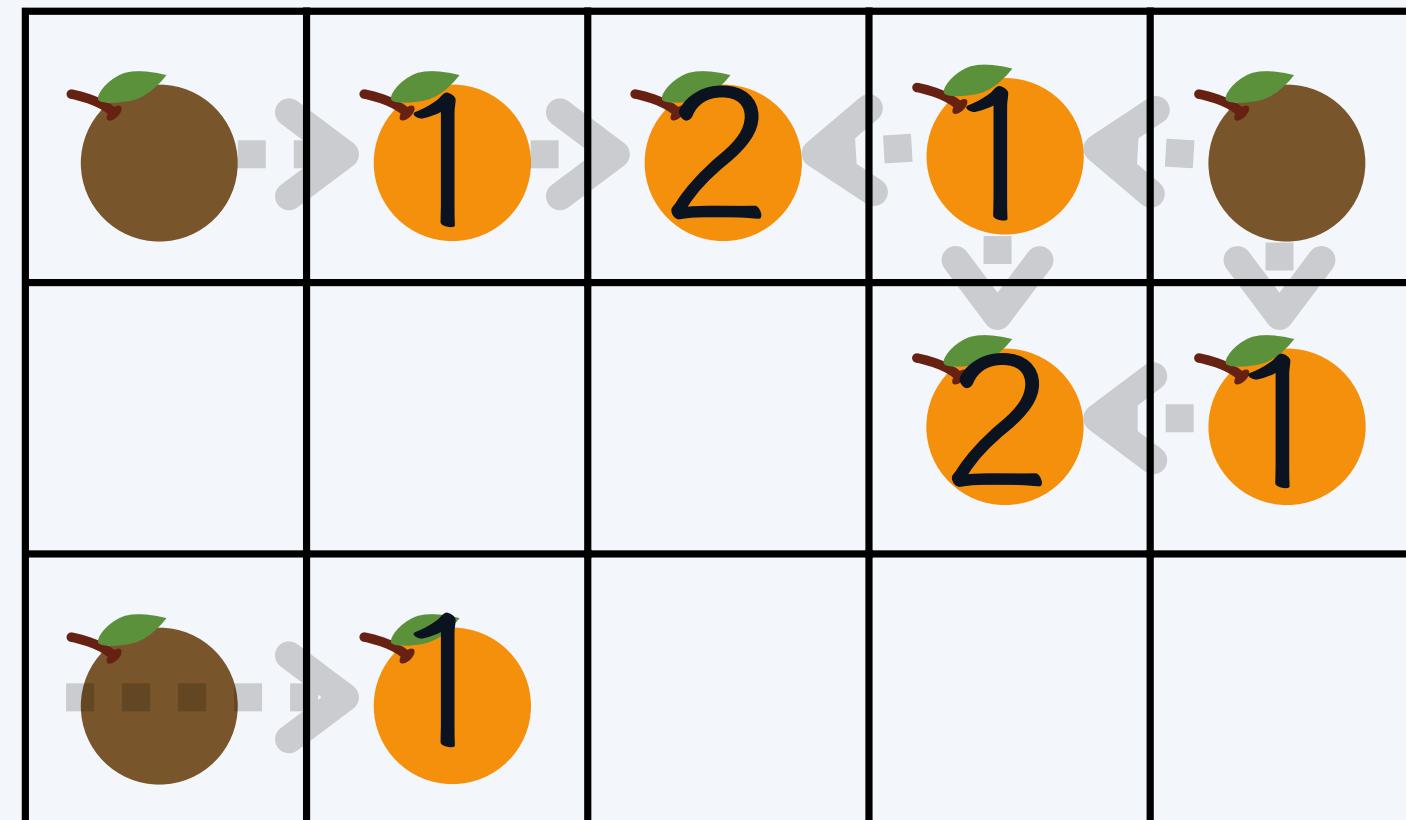
- 初次建立執行緒就建立固定數量的執行緒（Worker Threads）。
- 用 `pthread_mutex` 與 `pthread_barrier` 進行同步與資源管理。
- `pthread_mutex`：用於保護共享資源，避免多執行緒競爭造成資料不一致。
- `pthread_barrier`：確保所有 thread 在同一輪工作完成後同步。

方法二：MPI 平行策略

1. 多個起點同時執行Sequential 的腐爛橘子

每一個Process負責初始各自區域的腐爛橘子，每個Process同時執行BFS，並利用MPI_Allreduce來同步全局腐爛狀態

- 這個方法因為需要同步整張Grid的狀態，執行效率不佳，尤其是在Grid數量增加的情況下需要的時間會大幅增加

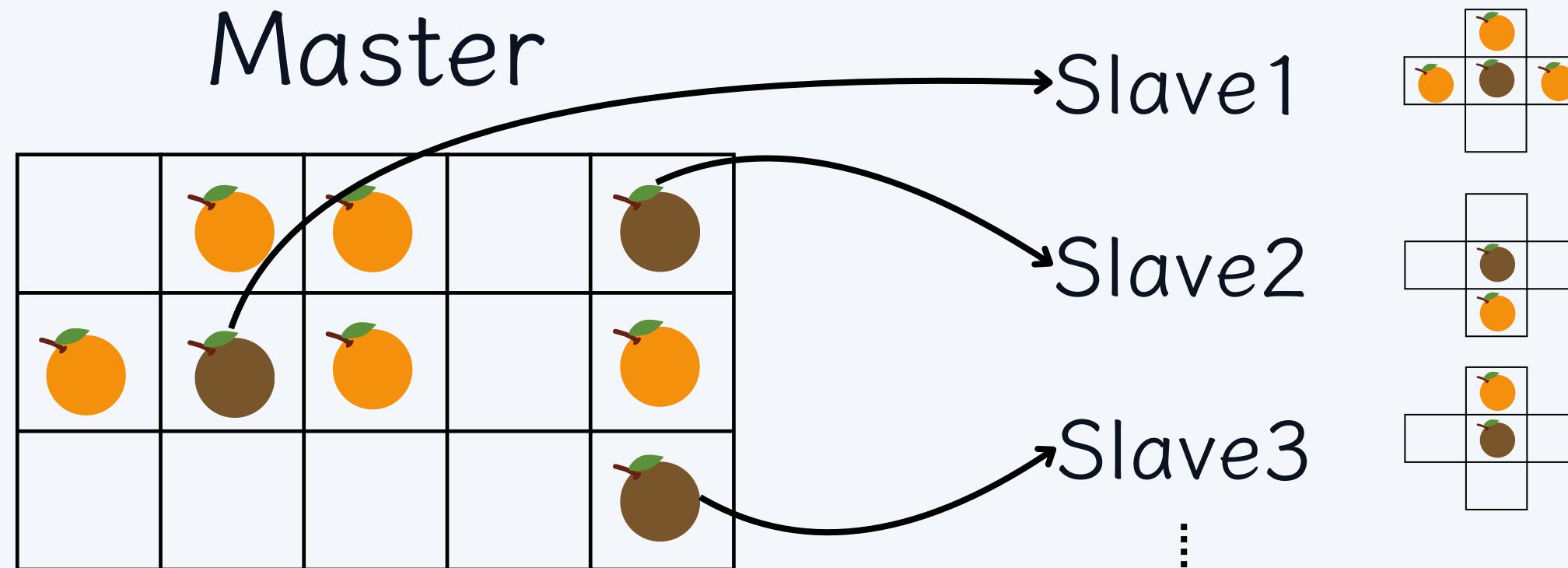


方法二：MPI 平行策略

2.Master-Slave方式分配任務

利用Master-Slave的平行化策略去分配任務，master將每一輪需要腐敗的橘子隊列分解成多個可獨立執行的任務交給slave去執行

- 避免了大型的數據交換，且不受初始腐爛橘子的區域限制
- 缺點是對於較小的計算數量會較為緩慢，因為process間的溝通時間占比會太大



方法二：MPI Master-Slave 實作細節

- 在 MPI 中，將結構體（例如 Point 結構）中的每個欄位分別傳送，會大大增加傳輸次數，並導致更高的溝通時間。而使用自定義的 MPI 類型的話，便能夠將整個結構作為一個單位傳送，減少傳輸次數和數據封裝的開銷。
- 由於BFS中的任務數量是不停變動的，為了確保在任務數量的高峰時可以有所有的進程參與計算，在低峰未參與任務的進程也必須和 Master溝通取得pending的訊號。如此一來能確保進程隨時準備接受新任務，即使這會使通信成本略為增加，但能提高整體資源的使用率，並減少高峰時的資源閒置。

方法三：GPU

Grid-based BFS

1. process_core Kernel:

- 將整個 `d_grid` 劃分 `block` 區塊，並每個載入到 `shared_mem`
- 如果鄰居格子是新鮮橘子，將其標記為「即將腐爛」狀態（值為 3），避免同一輪中被重複處理。
- 將所有即將腐爛的格子更新為真正腐爛（值為 2）。且使用 `atomicSub` 安全地減少 `fresh orange` 的值。

2. process_halo Kernel:

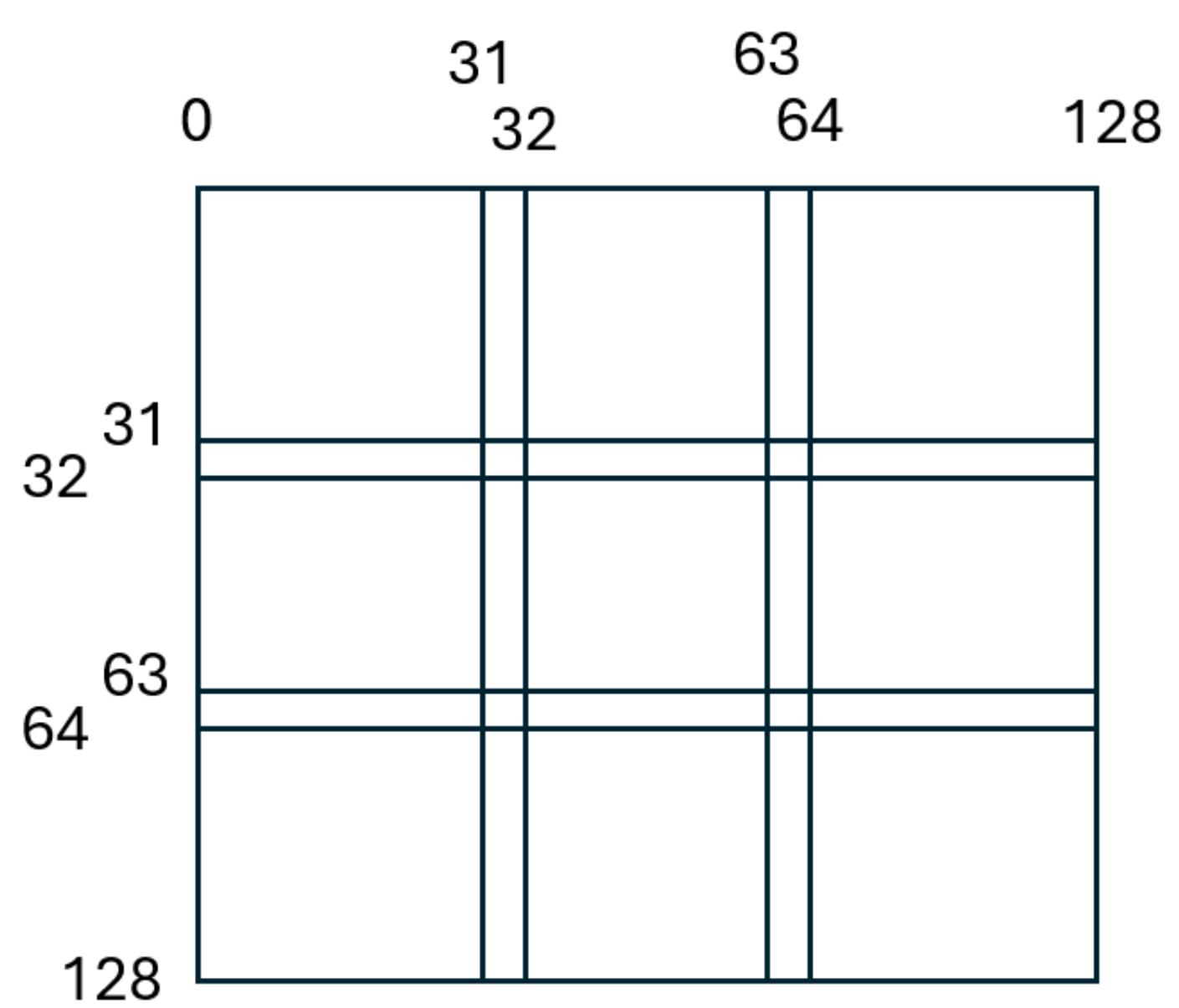
- 做的流程與 `process_core` 相同，只是只處理邊界上的格子

Queue-based BFS

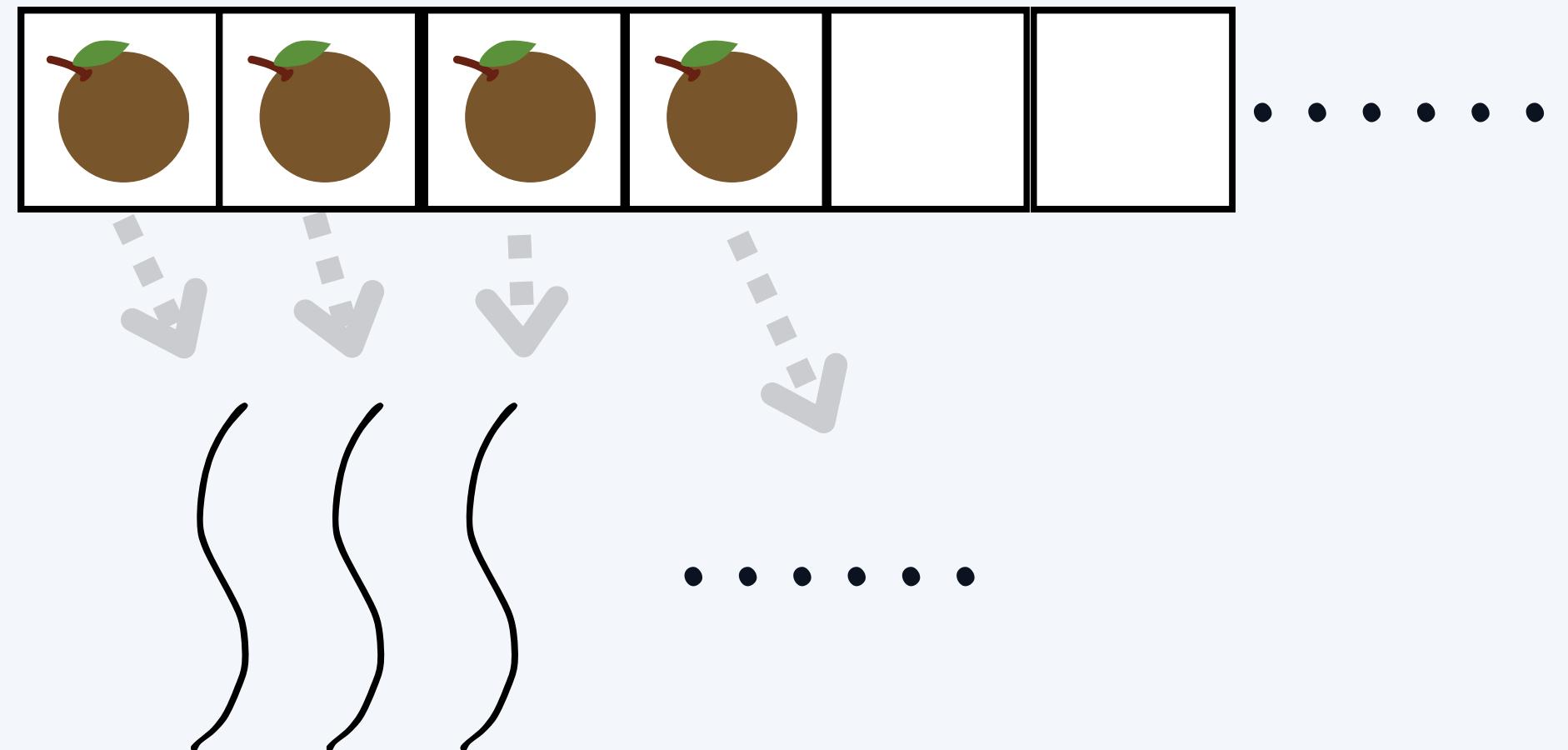
主要是透過並行化的方式加速 `queue` 的處理，使得每輪腐爛擴散的計算可以同時在多個 `thread` 上進行。

GPU Implement chart

Grid-based BFS



Queue-based BFS



GPU : Load Balance

Grid-based BFS

Data分布固定，所有 Block 都需要啟動，即使某些 Block 的工作量很小（例如完全沒有腐爛橘子的區域）。

Queue-based BFS

每輪只處理當前佇列 (`d_current_queue`) 中的腐爛橘子，根據腐爛橘子的數量動態分配執行緒。

```
while (fresh orange > 0){  
    ***  
  
    int blocks = (current_queue_size+threadsPerBlock -  
    1) / threadsPerBlock;  
  
    process_rotten_queue<<<blocks,  
    threadsPerBlock>>>  
  
    ***  
}
```

GPU : Advantage & disadvantage

	Grid-Based	Queue-Based
工作範圍	全域固定分配	動態分配，按需處理腐壞橘子的量
空閒threads比例	高（稀疏場景下）	低（幾乎所有執行緒有實際工作）
密集腐爛時的情況	負載均衡良好	負載可能不均，當queue數量> 256時，但總時間較短
稀疏腐爛場景	負載極不均勻，效率低	負載均衡良好，效率高
worst case 整體時間	159.90 seconds	6.74 seconds
Share memory	如同作業使用	無法使用腐爛橘子的queue的索引在網格中並不連續也不規律，也不一定局限於某個區域。

GPU : Compare to sequential CPU

- 腐爛橘子屬於簡單條件問題，只需鄰居檢查並更新，只需幾個邏輯便可完成，無法體現GPU大量浮點運算效能，成為了在此題目中最大的瓶頸。
- GPU 內部開消遠比CPU多太多，每次需要對兩個queue進行切換（ t 時間的queue 和 $t+1$ 時間的queue），並且寫入寫回datas，而CPU不需要處理threads內部commucation的問題，也沒有多餘的記憶體分配和同步操作。
- 每次迭代需要啟動 GPU Kernel，Kernel 啟動本身有一定的延遲。這題計算並沒有複雜到GPU加速整體效能比延遲還明顯，即使是worst case也是。

資料集

我們使用 Python 腳本產生多組隨機網格 (grid) 資料集，並為每個節點賦予預測值 0、1、2，依照不同機率分佈進行設計，以符合多元測試需求。資料集設計分為以下幾個情境：

small.txt：資料集範圍 $1 \leq m, n \leq 10$

此類資料集適合進行功能性驗證與基礎邏輯測試，確保算法的正確性。

large.txt：資料集範圍 $20 \leq m, n \leq 50$ ，並設定預測值 0:1:2 的出現機率為 [15%, 75%, 10%]
模擬常見的數據分佈情境，用於評估算法在中型規模問題中的表現。

huge.txt：資料集進一步擴展至 $50 \leq m, n \leq 1000$ ，機率分佈設定為 [5%, 90%, 5%]。
驗證算法在處理大數據規模時的穩定性與效率。

實驗設置

Qtc : MPI and Pthread method

Apollo gpu : single GPU

實驗加速結果



結論

1. Sequential 方法的執行時間最久，尤其是在大型測試案例（如 01.huge、03.huge 和 05.huge），表示平行的方法可以提高效率。
2. Pthread-ms 和 Pthread 方法在中小型測試案例中表現穩定且快速，相較於 Sequential 有明顯改善，而 Pthread-ms 的效率也確實高於 Pthread。
3. MPI 方法在小型案例中的表現並不理想，部分大型案例（如 02.huge）中稍具優勢，但仍不及 GPU 的表現。
4. GPU-grid 和 GPU-queue 方法的執行時間最短，幾乎在所有測試案例中都展現出強大的計算效能，尤其在小型測試案例時更為突出，顯示 GPU 平行運算對加速 BFS 算法有極大優勢。

結論：GPU-based 方法是處理 BFS 問題的首選，無論測試案例大小都展現出壓倒性的效能。而 Pthread 和 MPI 方法雖然有助於提升效能，但仍不如 GPU 技術高效。
因此，若在實務上需要追求最佳計算效率，導入 GPU 平行運算將會是最具效益的選擇！

Thank you
very much