Parallel Programming HW4

107062312 郭濬睿

Implementation:

首先，我利用 MPI 讓 rank 為 0 的 node 成為 Scheduler，其他的都是 Worker，並讓其都執行各自的 Map phase、Reduce phase，完成大致的架構。

```
if (rank == 0) { // job tracker
    Scheduler scheduler(size, job_name, num_reducer, delay, input_
    scheduler.Map_phase() ;
    std::cout << "Scheculer finish map phase\n" ;
    scheduler.Reduce_phase() ;
    std::cout << "Scheculer finish reduce phase\n" ;
    MPI_Barrier(MPI_COMM_WORLD) ;
}
else { // worker
    Worker worker(size, rank, job_name, num_reducer, delay, input_
    worker.Map_phase() ;
    std::cout << "Mapper " << rank << " finish map phase\n" ;
    worker.Reduce_phase() ;
    std::cout << "Mapper " << rank << " finish reduce phase\n" ;
    MPI_Barrier(MPI_COMM_WORLD) ;
}
```

(in main)

- Map phase:
  - Worker:

    因為每個 worker node 都有 s-1 個 map thread 可以工作，因此我先 create 出這些負責實際完成 map functions 的 worker thread，並將 worker 自己當作 parameter 傳給這些 thread，以利之後 sync 的相關操作。

    這些 worker thread 需要做的事情是，先從收到的 argument (worker) 裡，嘗試拿到要計算的 map task id。如果目前 task 已經被其他 thread 拿走了，那就利用 pthread condition wait，否則就把 task 拿走，進行 map functions 的操作，並一直重複。而為了讓每個 thread 能正常的結束，會有一個變數儲存是否已經完成所有 map task，讓 thread 可以判斷是否要跳出無限迴圈。

    以上的這些判斷我都會用 pthread mutex lock 保護，以確定一次只有一個 thread 可以 access worker，取得 task 的資訊。

```c
void *worker_thread_func(void *args) {
    Worker *pool = (Worker *)args ;
    int task_chunkIdx ;

    for (; ; ) {
        pthread_mutex_lock(&(pool->work_lock)) ;

        while (pool->task[1] == 1 && !pool->done) {
            pool->waiting_threads++ ;
            pthread_cond_wait(&(pool->cond), &(pool->work_lock)) ;
        }

        if (pool->done) {
            break ;
        }

        task_chunkIdx = pool->task[0] ;
        pool->task[1] = 1 ;
        pool->waiting_threads-- ;
        pthread_mutex_unlock(&(pool->work_lock)) ;

        pool->Map_functions(task_chunkIdx) ;
    }

    pthread_mutex_unlock(&(pool->work_lock)) ;
    pthread_exit(NULL) ;
}
```

嘗試拿 task，否則 wait

至於 worker threads 一直拿的 map task id 是如何補充，我讓 main thread 負責與 Scheduler 溝通。Main thread 會以類似 busy waiting 的方式，一直去觀察目前的 task 被拿走過了沒，如果已經被拿走了且有 worker thread 正在等待，就 send request(tag = 0)給 scheduler 並收到新的 map task id，之後就可以 signal 一個正在等待的 worker thread 起來做工作。

而若收到的 map task id 是 0，代表 scheduler 是要告訴 worker 已經沒有 map 的工作了，此時我會先讓 worker 送一個 pseudo-complete message(tag = 1)給 scheduler，詳細原因將於之後解釋 scheduler 時表述。之後就可以把前文所提到「儲存是否已經完成所有 map task」的變數設為 True，喚醒所有等待中的 worker thread，令其結束。

```c
while (true) {
    pthread_mutex_lock(&work_lock) ;
    if (task[1] == 1 && waiting_threads > 0) {
        int task_chunkIdx ;
        // send request
        MPI_Send(&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD) ;
        // recv
        MPI_Recv(&task_chunkIdx, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE) ;

        // if done
        if (task_chunkIdx == 0) {
            done = true ;
            MPI_Send(&task_chunkIdx, 1, MPI_INT, 0, 1, MPI_COMM_WORLD) ;

            pthread_cond_broadcast(&cond) ;
            pthread_mutex_unlock(&work_lock) ;
            break ;
        }
        else {
            task[0] = task_chunkIdx ;
            task[1] = 0 ;
            pthread_cond_signal(&cond) ;
        }
    }
    pthread_mutex_unlock(&work_lock) ;
}
```

task 已被拿走，且有 worker thread 正在等待

(in Worker::Map_phase)

以上即為 Worker 在 map phase 時的架構。接下來則說明 worker thread 拿到 map task id 後執行的 map functions 分別做了什麼。

```
void Worker::Map_functions(int task_chunkIdx) {
    if (task_chunkIdx < 0) {
        task_chunkIdx = -task_chunkIdx ;
        sleep(delay) ;
    }
    std::vector<std::pair<int, std::string>> record = Input_split(task_chunkIdx) ;
    std::vector<std::pair<std::string, int>> out_record = Map(record) ;
    Partition(out_record) ;
    // done
    MPI_Send(&task_chunkIdx, 1, MPI_INT, 0, 1, MPI_COMM_WORLD) ;
}
```

首先，若是收到的 map task id < 0，代表 scheduler 告訴 worker 這個 task 沒有 locality，所以此時這個 worker thread 就會 sleep(delay)以模擬讀取 remote data 所耗費的時間。處理完可能的 locality 問題後，就會依序執行 Input split、Map、Partition，並在做完所有事後送一個 complete message(tag = 1)給 scheduler，以利 scheduler 計算執行時間以及判斷結束條件。

Input split、Map 都是按照 spec 上的要求進行實作，而 Partition function 我目前是以「開頭字母對 reducer 數量的餘數」進行 hash。其餘部分僅為單純程式實作，因此不贅述。

- Scheduler:

Scheduler 的部分我一開始的想法是 create worker 數量個 thread 出來，一個 thread 負責一個 worker 的溝通。然而因為在 map phase 時，一個 worker node 有可能有超過一個 worker thread，所以會發生「剛派了一個 task 給 worker A，此 task 尚未完成時 worker A 馬上又要一個新的」，比較難同時處理「派送任務」和「收到完成訊息」兩件事，因此我後來改成對每個 worker node 都 create 2 個 thread，分別處裡上述兩種任務。

其中，負責派送任務的 scheduler dispatch thread 做的事跟 worker main thread 對應，會以迴圈一直等待 worker send request(tag = 0)。收到 request 後則會從 scheduler 中拿一個 task 出來，send 給 worker。而若是此 task id 為 0，即前述 worker 中所提到「已結束」的訊息，此 thread 即可直接結束。

```
for (; ; ) {
    MPI_Recv(&buf, 1, MPI_INT, rank, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE) ;
    task_chunkIdx = scheduler->Dispatch_mapper(rank) ;
    MPI_Send(&task_chunkIdx, 1, MPI_INT, rank, 0, MPI_COMM_WORLD) ;
    if (task_chunkIdx == 0) { // done
        break ;
    }
}
```

拿 task 則是用 data locality aware scheduling 的方式：首先先看還有沒有 task，若還有 task，會先掃過所有 task 一遍找看看有沒有符合此 worker locality 的，若有則優

先派送此 task，否則就直接拿第一個 task。決定派送的 task 後，會紀錄此 task 的派送時間以及更新此 worker 已經負責了幾個 task，以利之後 log file 處理以及判斷結束的條件。同樣的，這些操作都會以 pthread mutex lock 保護。

而負責收到完成訊息的 scheduler check thread 則是簡單的一直 receive complete message(tag = 1)，並作相應的處理，直到判斷結束。

```
while (!done) {
    MPI_Recv(&task_chunkIdx, 1, MPI_INT, rank, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE) ;
    done = scheduler->Map_task_complete(rank, task_chunkIdx) ;
}
```

「相應的處理」方式為，若收到的不是 pseudo-complete message，代表真的有個 task 已被完成，就可以將其資訊寫入 log file，並更新此 worker 負責 task 數量。而最重要的是，會在這裡判斷結束條件：當此 worker 負責的 task 數已經為 0 且所有 task 都已經被派送完了，代表也已經收到所有曾經派給此 worker 的 task 的完成訊息，此時就可以 return 1 讓相應的 scheduler check thread 結束。

在這裡就可以解釋為何之前會要讓 worker 在要結束 map phase 前送一個 pseudo-complete message：因為當 map task 很少，每個 worker node 的 worker thread 很多時，是有可能發生某個 worker 完全沒拿到 task 就要結束的情況，例如若只有 2 個 map task，但有兩個 worker(A、B)各有 11 個 worker threads，結果 2 個 task 都給了 A，B 什麼都沒做到就被通知可以結束了。而此時若沒有 pseudo-complete message，負責 B 的 scheduler check thread 就會一直卡在第一個 receive，無法結束。

而在 Worker 及 Scheduler 都以 pthread join 等待 worker thread/scheduler dispatch thread/scheduler check thread 結束後，代表已成功完成所有 map tasks，此時即可以用 MPI Reduce 蒐集分散在所有 worker 的 data chunk key 數量，寫進 log file，之後完成 shuffle，開始 Reduce phase。

- Reduce phase:
  - Worker:

    因為每個 worker 只有一個 reducer thread，因此這部分較 map phase 簡單，只需要把之前 map phase 時 main thread 做的事搬過來，把之前是「收到 task 後叫醒一個 worker thread 執行 map functions」，改成「收到後自己執行 reduce functions」即可。同樣的，為了區分不同的訊息，我讓 request reduce task 的 tag = 2、reduce complete message 的 tag = 3。

```
while (true) {
    // send request
    MPI_Send(&rank, 1, MPI_INT, 0, 2, MPI_COMM_WORLD) ;
    // recv
    MPI_Recv(&reducer_task, 1, MPI_INT, 0, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE) ;

    // if done
    if (reducer_task < 0) {
        break ;
    }
    else {
        Reduce_functions(reducer_task) ;
        MPI_Send(&reducer_task, 1, MPI_INT, 0, 3, MPI_COMM_WORLD) ;
    }
}
```

而 reduce functions 的 Sort、Group、Reduce、Output，也都僅是按照 spec 的程式實作，因此不贅述。

```
void Worker::Reduce_functions(int reducer_task) {
    std::vector<std::pair<std::string, int>> records = Sort(reducer_task) ;
    std::map<std::string, std::vector<int>> group_records = Group(records) ;
    std::vector<std::pair<std::string, int>> results = Reduce(group_records) ;
    Output(results, reducer_task) ;
}
```

- Scheduler:

  同樣的，在 reduce phase 的 scheduler 也較為簡單，因為現在就不需要考慮之前提到的 worker 有多個 thread 的問題了，因此在 map phase 的兩種 thread 就可以直接併在一起，即只需要一個 thread 對應一個 worker 即可。

```
while (true) {
    MPI_Recv(&buf, 1, MPI_INT, rank, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE) ;
    reducer_task = scheduler->Dispatch_reducer(rank) ;
    MPI_Send(&reducer_task, 1, MPI_INT, rank, 2, MPI_COMM_WORLD) ;
    if (reducer_task < 0) {
        break ;
    }
    MPI_Recv(&reducer_task, 1, MPI_INT, rank, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE) ;
    scheduler->Reduce_task_complete(reducer_task) ;
}
```

每個 thread 做的事邏輯上跟 map phase 一模一樣：先等待 worker send request(tag = 2)，之後利用受到 pthread mutex lock 保護的方法拿到 reducer task id，傳給 worker。若 reducer task id < 0，代表已經沒有 task 要做了，此 thread 即可直接結束；否則就等待 complete message(tag = 3)，將資訊寫入 log file。

而在 reduce phase 也就不需要考慮 pseudo-complete message 的問題了，因為 scheduler thread 一定是有派送出一個有效的 task 才會試圖等待 complete message，就算有個 worker 完全沒有負責到 task，相應的 scheduler thread 在送出可以結束的訊息後自己就會立即 break，因此不會有卡住的問題。

以上即為我在這次作業中的架構與作法。

Experiment:

Impact of data locality:

我以 testcases 中 09.word 作為測資，測試以下三種 locality 影響執行時間的情況，分別為：所有 data 都只在一個 node 上、平均分散在兩個 node、平均分散在三個 node，並都以 1 個 scheduler 3 個 worker，每個 worker 都只有一個 worker thread 的架構執行，測試執行時間。

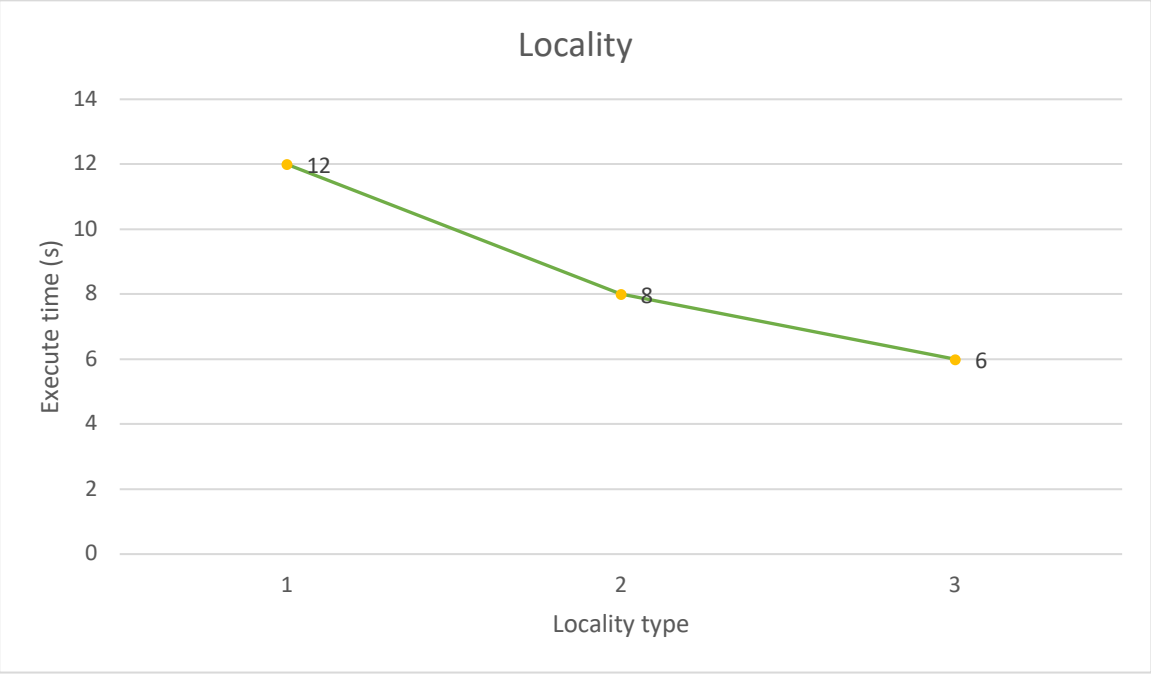
Command: srun -N4 -c2 ./hw4 {Job name} 7 3 {input} 5 {loc type}.txt {output dir}

(delay = 3)

Data 分散的數量	1	2	3
總執行時間	3 s	3 s	3 s
Map phase 執行時間	3 s	3 s	3 s

可以看到因為測試的 word count 檔案太小的關係，實際的執行時間都只有幾毫秒，這樣只要有派送一個沒有 locality 的 task，大家的執行時間都會被 delay 的時間 dominate，很難做相關的測試，因此我在 map functions 及 reduce functions 的最後都加上 sleep(1)，模擬假設每個工作需要一秒執行時間的情況。

Data 分散的數量	1	2	3
總執行時間	15 s	11 s	9 s
Map phase 執行時間	12 s	8 s	6 s



這樣就可以發現 locality 確實會影響執行時間。

Scalability:

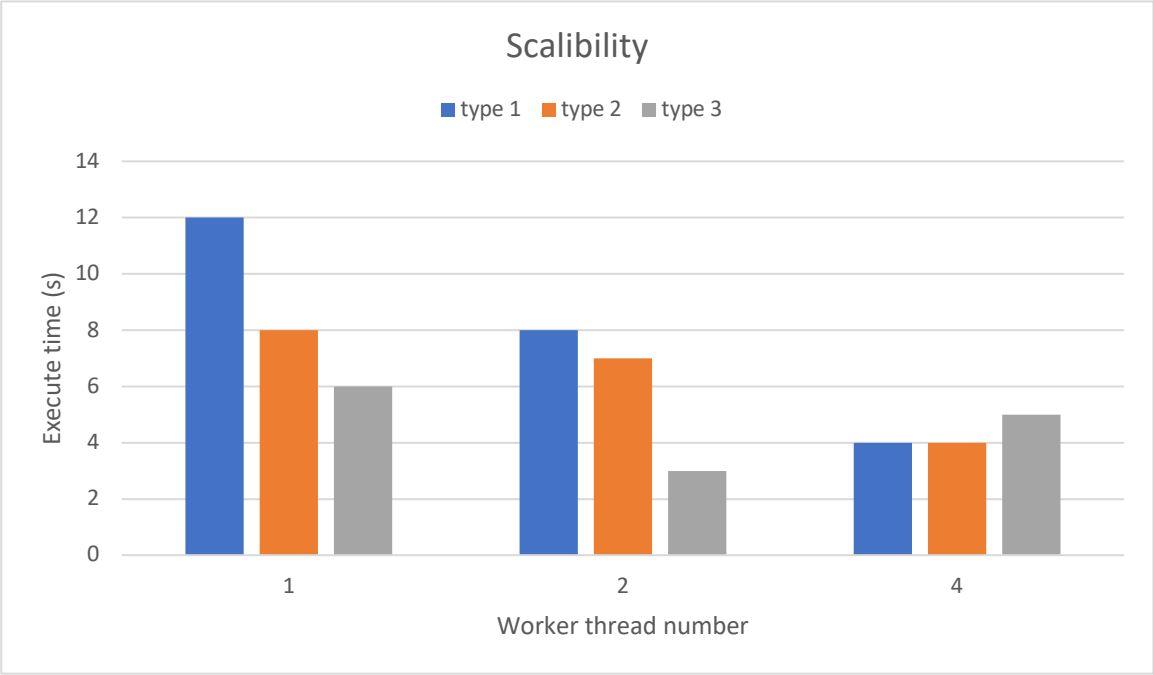
接下來我以同樣的設定測試一些不同 node 數量、cpu 數量得到的執行結果。

Command: srun -N4 -c3 ./hw4 {Job name} 7 3 {input} 5 {loc type}.txt {output dir}

Data 分散的數量	1	2	3
總執行時間	11 s	10 s	6 s
Map phase 執行時間	8 s	7 s	3 s

Command: srun -N4 -c5 ./hw4 {Job name} 7 3 {input} 5 {loc type}.txt {output dir}

Data 分散的數量	1	2	3
總執行時間	7 s	7 s	8 s
Map phase 執行時間	4 s	4 s	5 s



這裡我觀察到了一個有趣的現象：當每個 worker 的 worker thread 從 1 -> 2 時，會讓每個 worker 可以多拿一點屬於自己 locality 的工作來做，且就算不是自己的 locality 也可以因為一次執行兩個而減少總執行時間，因此執行時間不管是哪種 locality 都會下降。

然而當 worker thread 從 2 -> 4 時，在我目前的實作方法下，worker 會一直拿工作讓 4 個 thread 都有事做，但這反而會讓 worker 多搶到不屬於自己 locality 的工作，導致有 locality 的 worker 提早執行完只能等待，因此雖然比起 1 個 worker thread 執行時間仍有下降，但比起 2 個 worker thread，locality 3 的執行時間反而變久了。

Command: srun -N{N} -c3 ./hw4 {Job name} 7 3 {input} 5 {loc type}.txt {output dir}

Worker 數量	1	2	3
Data 分散的數量	1	1	1
總執行時間	16 s	12 s	11 s
Map phase 執行時間	9 s	8 s	8 s
Data 分散的數量		2	2
總執行時間		12 s	10 s
Map phase 執行時間		8 s	7 s
Data 分散的數量			3
總執行時間			6 s
Map phase 執行時間			3 s

Experience:

在這次作業中，我通過實作瞭解了 MapReduce 的架構，也更加熟悉了 MPI 以及 pthread programming，例如 thread pool 的實作概念等。在完成作業後看到符合預期的結果，也讓我很有成就感。