Homework 4: MapReduce

107062120 姚潔恩

Implementation

Process Level

這邊討論 process level communication 的部分，也就是 job tracker 和 task tracker 的互動。job tracker 在分配 map task 和 reduce task 時都需要與 task tracker 溝通，溝通的流程大致分為三個階段：

1. job tracker 接收來自 task tracker 的 request 並回傳分配的工作

   在這個階段 job tracker 會先用 MPI_Recv 接收 request，來源會設定為 MPI_ANY_SOURCE 因此可以接收到任何一個 task tracker 的 request。其中接收到的 data 是一個 int，表示發送 request 的 process node ID，因此接下來 job tracker 就可以選擇要分配的工作並設定 MPI_Send 的目的地為發送 request 的 node ID，將工作傳給那一個 process。在工作派完之前都會持續進行這個階段，也就是 job tracker 會持續接收來自各個 process 的 request 並派出工作。

2. job tracker 通知 task tracker 工作已經分配完畢

   當所有工作都分配完畢以後，job tracker 會再做(# task tracker)次的 MPI_Recv 接收 request，在接收到 request 後將 job id 設為-1 並回傳。因此 task tracker 如果接收到 job id 為-1 就表示所有工作都分配完了，就不用在 request 新工作。

3. task tracker 回傳工作完成的資訊。

   當 task tracker 接收到工作分配完畢的通知以後，就會開始等所有被分配到的工作執行完畢。等所有被分配的工作都執行完畢 task tracker 就會開始傳送工作完成的資訊給 job tracker，包括工作的 id 以及執行時間等資訊。

因為有很多的 MPI_Send 和 MPI_Recv，為了避免錯誤我使用了六個 MPI 溝通的 tag，tag 0 / 1 / 2 分別是 map task 的 receive request / send job / receive complete information，3 / 4 / 5 就是 reduce task 的。

Thread Level

這邊討論 thread level 的部分，也就是使用 thread pool 運算 map task 的實作。基本上 thread pool 的實作包含兩部分，首先是要 create 很多 thread 成為 thread pool，這些 thread 會嘗試 lock mutex 並去 global 的 task queue 拿 task 並執行，如果 task queue 是空的就會 wait on 一個 condition variable，等到有工作了才會被 signal。實作如下圖。

```
pthread_mutex_lock(&mutex);
while (tasks.empty()) {
    pthread_cond_wait(&cond, &mutex);
}
task = tasks.front();
tasks.pop();
pthread_mutex_unlock(&mutex);
```

再來就是 process 這邊在接收到新的工作以後會將工作 submit 給 thread pool 的 thread。實作如下圖，把工作 push 到 job queue 以後就可以對 condition variable 發出 signal，通知 thread pool 中的 thread 有新工作。

```
pthread_mutex_lock(&mutex);
tasks.push(job);
pthread_mutex_unlock(&mutex);
pthread_cond_signal(&cond);
```

除了 thread pool 以外，控制 task tracker 向 job tracker 發送 job request 也需要用到 mutex 和 condition variable。具體來說，在發送 request 之前會檢查現在 thread pool 中有多少 job 還在執行，如果 thread pool 中每個 thread 都有 job，process 就會 wait on condition variable，暫時不發送 job request。等到 thread pool 中有工作結束就會對這個 condition variable signal，讓 process 去要下一個 job。

```
while (true) {
    pthread_mutex_lock(&mutex_complete);
    while (num_jobs == num_threads-1) {
        pthread_cond_wait(&cond_complete, &mutex_complete);
    }
    pthread_mutex_unlock(&mutex_complete);
```

Intermediate Result & Shuffle

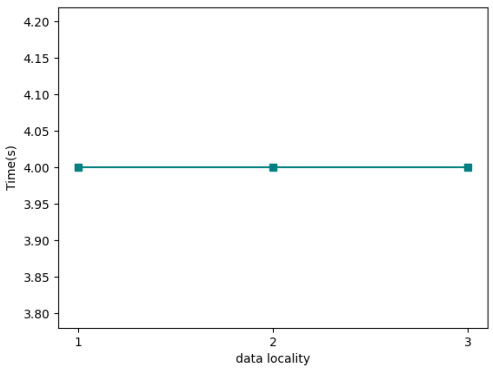
我會將每個 mapper task 最後 map 的結果都存為一個 intermediate result，因此有幾個 mapper task 就會有幾個 mapper 的 intermediate result。而接下來的 shuffle 我是實作在 job tracker 中。具體來說，在所有 task tracker 都完成工作以後 (也就是 job tracker 接收到所有 mapper task 的完成資訊後)，job tracker 會先創建(#reducer)個檔案，接著 job tracker 就會依序將每個 mapper task 的 intermediate result 讀進來，對每個 key value pair 呼叫 partition function 決定他們應該由哪個 reducer 負責，再寫到對應的檔案。最終得到的(#reducer)個檔案就是 reducer 在執行 reduce 的時候要讀的檔案。

Experiment & Analysis

A. System Spec

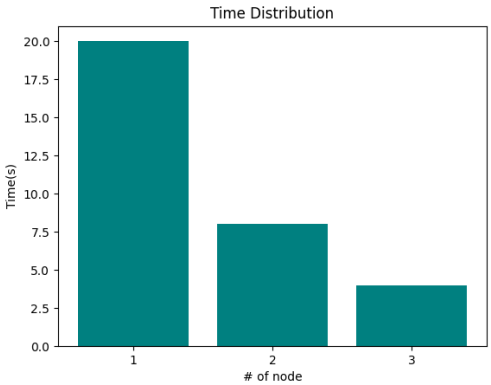
實驗環境使用課程提供的 server apollo。使用的測資行數為 9600，分為 30 個 chunk，chunk size 為 320。使用的 Node 數為 4，thread 數為 6，reducer 數為 10，locality 的 delay 為 4。

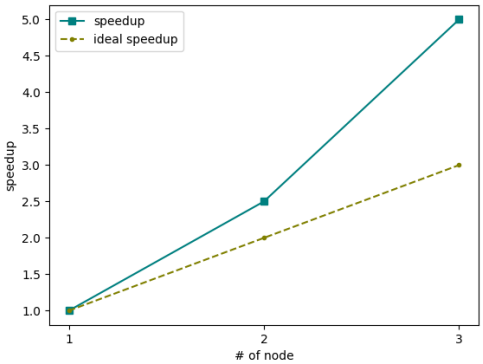
B. data locality



我使用助教提供的 generator.py 產生了三個 locality file 以測試不同 locality 的影響。第一個 file 是全部的 chunk 都在 node 0，第二個 file 是隨機分散在 node 0, 1，第三個 file 是隨機分散在 node 0, 1, 2。實驗結果如上圖，不管 locality 的是怎樣執行時間都是 delay 的秒數。會有這個現象的原因是現在的資料太小，所以 map 的執行時間幾乎是零秒，最後執行時間就變成只有 delay 的時間。以下舉例子進一步說明，例如現在所有的 chunk 都在 node 0，這時 node 1 和 node 2 將 thread pool 塞滿以後就需要等待 delay 秒才會再向 job tracker 要工作，但是 node 0 因為不用等待 delay 且 map 的執行幾乎不用時間，所以就會很快的把剩下的全部工作完成，所以最後的時間就是花 delay 秒。至於 chunk 隨機分散在 3 個 node 的實驗，因為是隨機分配所以每個 node 上的 chunk 數量並不平均。導致還是有 node 為了讓所有的 thread 都有工作，會拿到儲存在別的 node 的 chunk，所以還是要花 delay 秒。

C. scalability





scalability 實驗結果如上圖，可以看到時間隨著 node 數增加而遞減，而從 speedup 來看甚至超越了 ideal 的理想值。原因是因為 node 數增加不只是增加計算的平行度，也更能善用 data locality 的性質 (不管是幾個 node 的實驗我都會將 data chunk 隨機分散在 3 個 node 上)。

Conclusion

做完這一次作業不僅對 MapReduce 的架構更為熟悉，也扎實的練習了 master and slave 的架構以及 thread pool 的實作。