Homework 4: MapReduce

110062640 張德萱

1. List the highlights of your implementation.

- 將node分成scheduler與worker，scheduler負責分派task給worker，worker負責實際運作map task以及reduce task。
- worker與scheduler需要大量的溝通，透過`MPI_Send`, `MPI_Recv`來實現，不同種類型的訊息以不同tag來分類。
- worker內的worker threads溝通、存取變數，採`pthread mutex lock`保護，以確保一次只有一個 thread可以access。
- scheduler會create (NODES-1)個thread，每個dispatch thread只要負責相對應的rank id的worker即可。
- dynamic分派task，scheduler根據worker id以data locality-aware scheduling的方式分派task id給有send request的worker。

2. Detail your implementation of the whole MapReduce program.

首先讓rank 0的node作為scheduler，負責分派task給worker。剩下的node都作為worker，負責實作map task以及reduce task。

以`MPI_Send`, `MPI_Recv`來實現進行不同 node 之間的溝通。以`pthread mutex lock`保護scheduler、worker內變數，以確保一次只有一個 thread可以access。

以下會將scheduler與worker行為分成map phase和reduce phase敘述。

Map Phase

- Scheduler
  - scheduler會create (NODES-1)個dispatch thread，每個dispatch thread負責相對應的rank的worker，dispatch threads會在迴圈裡busy waiting接收有沒有worker send request來要map task，並根據worker id以data locality-aware scheduling的方式分派task id給worker，若task id為0代表所有task都分派完了，即可結束派送工作、終止這

個thread。另外，scheduler還會create (NODES-1)個check thread，check thread一樣負責相對應的rank的worker，來接收worker thread送來的complete message，將資訊寫入log file，若此check thread對應worker的工作都做完即可終止這個thread。

- 等所有threads都終止後，用 `MPI Reduce`統計分散在所有worker的data chunk key數量，完成 shuffle後開始Reduce phase。
- data locality-aware scheduling實作是從locality file從頭到尾找有沒有符合request worker node的node id，若有就回傳task id，並且刪除掉這筆task代表已派送了。若無符合的node id，就回傳現有的第一筆的task id * (-1)並刪掉這筆task，代表非此worker locality的task，讓worker收到後要模擬讀取remote file。若locality file沒有task了，就回傳0，代表task都派送完畢了。

```
int Scheduler::Dispatch_mapper(int target_rank) {

    pthread_mutex_lock(&lock);
    // all dispatched
    if (locality.size() == 0) {
        pthread_mutex_unlock(&lock);
        return 0 ;
    }

    // find locality
    for (auto i=locality.begin(); i!=locality.end(); i++) {
        if ((*i).second == target_rank) {
            task_chunkIdx = (*i).first;
            locality.erase(i);
            pthread_mutex_unlock(&lock);
            return task_chunkIdx;
        }
    }
    // no locality exists
    task_chunkIdx = locality[0].first;
    locality.erase(locality.begin());
    pthread_mutex_unlock(&lock);
    return -task_chunkIdx;
}
```

- Worker
  - worker會create (ncpus-1)個thread並且send request給scheduler要task，拿到task id先判斷不為0就給workers thread去搶這個task，若為0就代表task都做完了，將worker裡的參數done改為true，讓worker threads可以跳出迴圈不用再等。
  - worker threads會在迴圈裡busy waiting有沒有新的task，若搶到task id 後去執行 `Map_functions()`。會先判斷task id是否> 0，若是就去讀取相對應的chunk若id < 0就代表非locality的chunk，先`sleep(delay)`以模擬讀取 remote data 所耗費的時間，再去讀取chunk。執行`Map_functions()`裡面包含spec中所規範的`Input split`

`function`、`Map function`、`Partition function`，來計算這個chunk的word count，再把results hash給reducer。執行完成後送一個 complete message給 scheduler，以利 scheduler計算執行時間以及判斷結束條件。

Reduce Phase

- Scheduler
  - scheduler會create (NODES-1)個dispatch thread，dispatch threads會在迴圈裡busy waiting接收有沒有worker send request來要reduce task。scheduler會依序從task id 0 到 `num_reducer`開始派送reduce task，接著接收worker thread送來的complete message，將資訊寫入log file。scheduler以`reducer_to_dispatch`記錄現在派送到哪一個reduce task，每分派出一個task，`reducer_to_dispatch`就+1，直到 `reducer_to_dispatch >= num_reducer` 代表reduce task都分派完了，就回傳-1給 worker，即可結束派送工作、終止這個thread。
  - 等待所有dispatch thread都中止後，就輸出FinishJob到log file，結束整個map reduce 的任務。

- Worker
  - worker會send request給scheduler要task，拿到task id先判斷不為-1就去執行 `Reduce_functions()`來處理這個reduce task，若為-1就代表task都做完了，可以跳出迴圈不用再等。
  - `Reduce_functions()`包含spec中所規範的`Sort function`、`Group function`、`Reduce function`、`Output function`，來計算這個reduce task中的key count，並輸出到output file上。執行完成後送一個 complete message給 scheduler，以利 scheduler 計算執行時間以及判斷結束條件。

3. Conduct experiments to show the performance impact of data locality, and the scalability.

- System spec

使用課堂提供的apollo server

- Impact of data locality

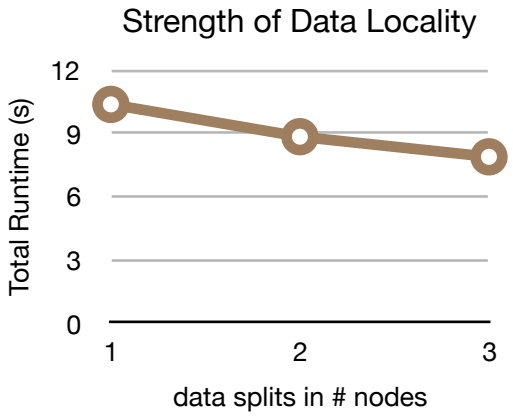
使用最大的test case 6作為測資，以一個scheduler、三個 worker，每個 worker 都只有一個worker thread的架構執行，測試執行時間。delay時間設為5，chunk size設為2。

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

srun -N4 -c2 ./hw4 {JOB_NAME} 9 5 {INPUT_FILENAME} 2 {LOCALITY_CONFIG_FILENAME} {OUTPUT_DIR}

Runtime (s)

data分散node數量	1	2	3
Total Runtime	10.382	8.848	7.893
Map Phase Runtime	9.462	7.934	6.747



單純只看map phase runtime可以看出data locality對執行時間的確有造成影響，當資料分散在越多node上面，執行時間越短。而整體時間幾乎也都是受map phase掌控，故呈現相同的趨勢。

- Scalability

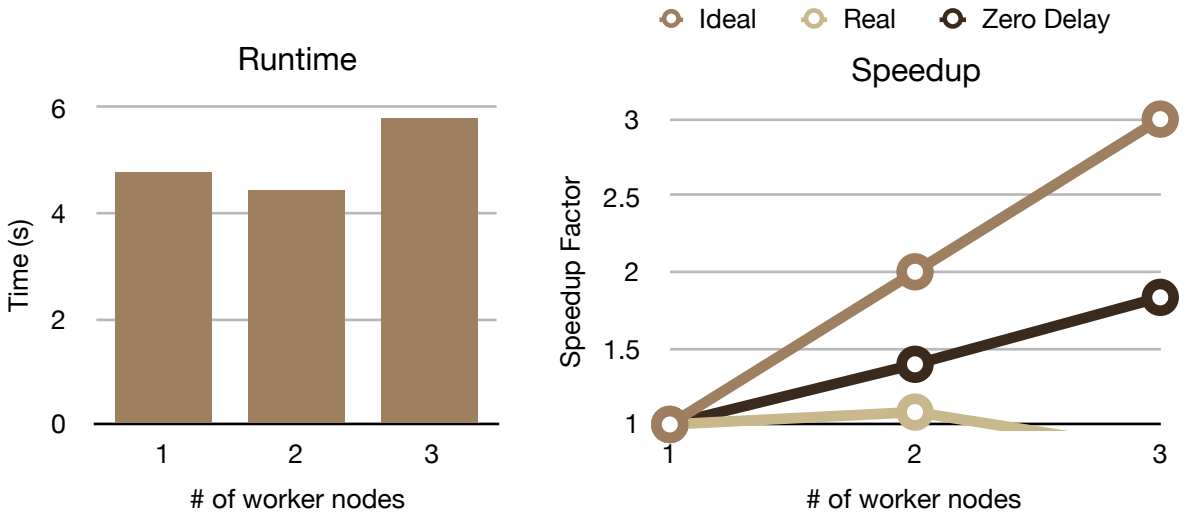
為了不讓delay時間佔執行時間太大部分，而影響scalability實驗效果，我把delay設為1

  - Number of nodes

```
srun -N4 -c2 ./hw4 {JOB_NAME} 9 1 {INPUT_FILENAME} 2 {LOCALITY_CONFIG_FILENAME} {OUTPUT_DIR}
```

Runtime (s) -1

worker node數量	1	2	3
Real (delay = 1)	4.771	4.413	5.767
Zero Delay	4.906	3.519	2.677



從圖表可看出當worker node增加為2時，執行時間比起1有減少，但是增加為3時，時間反而上升了，speedup factor小於1。推測為是因為node數量變多，而受到data locality的影響，隨著node數量變多，worker較有可能要去讀remote file，因此有delay時間所以影響到了scalibility。

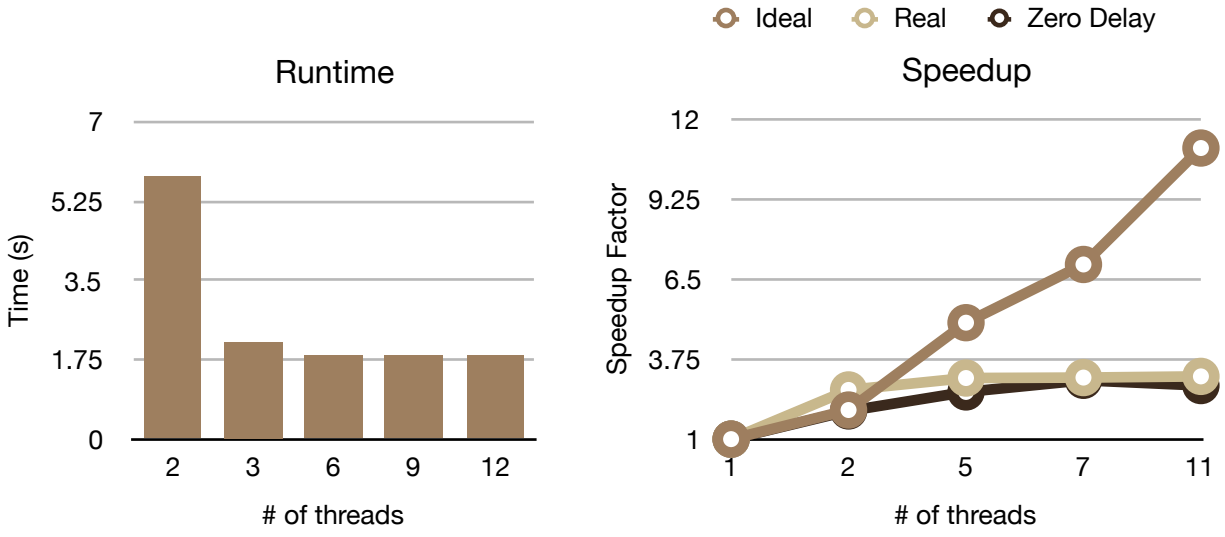
故我又做了delay time為0的實驗，可以看出執行時間有隨著worker node增加而減少，speed up factor也呈現上升的趨勢。

- Number of threads

```
srun -N4 -c3 ./hw4 {JOB_NAME} 9 1 {INPUT_FILENAME} 2 {LOCALITY_CONFIG_FILENAME} {OUTPUT_DIR}
```

Runtime (s)-2

threads 數量	1	2	5	7	11
Real (delay = 1)	5.806	2.154	1.873	1.864	1.840
Zero Delaye	2.716	1.379	1.036	0.899	0.956



當worker threads數量由1增加為2時，執行時間大幅減少，speed up factor超過ideal，但數量增加至5, 7, 11時執行時間都差異不大，並無明顯降幅。推測為threads數量由1增加為2時，不但可以平行化工作，也可以多拿一些屬於自己locality的task而少了些delay time，因此speed up factor能超過ideal。推測因為測資不夠大，所以當worker的threads太多時又會搶了不屬於自己locality的task，導致增加了delay time，原本擁有該locality的worker可能就沒有task可做而閒置。

4. Experience & conclusion

通過這次作業更瞭解 MapReduce 的架構，實作模擬遠距file system的分散平行處理。透過實驗看到了data locality所帶來的影響，對分散式運算的架構更有概念，也更加熟悉了 MPI 以及 pthread programming。