Parallel Programming HW4

108032053 陳凱揚

Implementation (highlight)

- Jobtracker & Tasktracker

  我將rank 0作為Jobtracker，其他則為Tasktracker，一開始Jobtracker會開一個大的queue和Tasktracker數量個queue，將locality config的資訊同時存到每個Tasktracker的queue和大的queue中，接著Tasktracker會發出request，Jobtracker就會優先從其對應的queue中拿chunk給他，如果沒有再從大的queue中拿沒使用過的chunk給他，而Tasktracker會一直發出request，直到Jobtracker的回應為terminate。

- Mapper

  Tasktracker拿到chunk後會開 `cpu_num-1` 個thread，將chunk中的所有record (line)平分給每個thread。

  - Map

    將每個record出現的word計算數量，存成一個 `unordered_map<string, int>` 回傳。

  - Partition

    使用 `std::hash<string>{}(s) % num_reducer` 來做 partition。

  - Store

    每個rank的每個reducer id都會開一個tmp file，例如在rank3且reduer id為5的word就會存到 `tmp3_5.txt` 中。由於在同個rank中的不同thread可能同時寫入相同的檔案，所以每個檔案都需要一個 `pthread_mutex_lock` 來保護。

- Shuffle

  需要先計算key-val總數量，所以在每個rank中，會有一個變數來加總Map回傳後的map size，此變數同樣需要 `pthread_mutex_lock` 來保護，所有mapper執行完後，做 `MPI_Reduce` 將值加總至Jobtracker。

  Jobtracker會將剛剛Mapper儲存的tmp file以reducer id來合併來完成shuffle。

- Reducer

  Reducer會用和mapper相同的方式來取得要做的reducer id，接著開啟 `1` 個thread來做以下的步驟。

  - Sort

    Sort的部分我定義了一個 `SortComparator` 的function來決定排序的方式，接著以 `std::sort` 來做排序。

  - Group

    我定義了一個 `GroupComparator` 的function來決定兩個key是否屬於同一個group，正常的情況下我們會以兩個string完全相等為條件，但也可以以其他的方式像是第一個字母為條件，因此我以較通用的方式來實作Group，也就是此group的key為其中最小的key，方法如下。

    首先開一個 `DisjointSet` ，每個string一開始屬於自己這個group，接著以剛剛的 `GroupComparator` 兩兩比較所有string，若屬於同一個group，則以 `SortComparator` 找出較小的string作為parent，將兩個string的group join起來，最後再將所有相同group對應的val存在一起，因此回傳的型態為 `vector<pair<string, vector<int>>>` 。

  - Reduce

    將前面回傳的group中，相同group裡的值加總起來，因此回傳型態為 `vector<pair<string, int>>` 。

  - Output

    將reduce後的所有key-val存至對應的reduce id的file中。

Implementation (detail)

- Jobtracker
  - Read locality config file

    如前述，會開一個大的queue和Tasktracker數量個的queue，讀取檔案後分別存到大的queue中和對應的node的queue中，同時也取得了chuck的數量，決定以後發chunk的次數。

    ```
    std::ifstream locality_config_file(locality_config_filename);
    int chunkID, nodeID, worker_num = mpi_num-1, chunk_num = 0, buf;
    std::vector<std::queue<int>> V(worker_num+1);
    std::queue<int> Q;
    while(locality_config_file >> chunkID >> nodeID){
        nodeID = (nodeID+worker_num-1)%worker_num+1;
        V[nodeID].push(chunkID);
        Q.push(chunkID);
        chunk_num++;
    }
    locality_config_file.close();
    ```

  - Receive mapper requests from nodes

    在這邊的步驟為從Tasktracker以 `MPI_Recv` 收到request，先去從發出request的worker id對應的queue找具有locality的chunk，若有的話就優先發送此task給他，若沒有的話再去大的queue中找其他的chunk，同時 `sleep(delay)` ，最後發完所有chunk後，還需要再接收worker數量個request，並發出teminate的訊息，讓worker知道所有chunk都已經處理完了。

    ```
    int remain_chunk = chunk_num;
    while(remain_chunk--){
        MPI_Recv(&buf, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        if(buf != -1) Log(log_file, "Complete_MapTask", buf, -1, start_time[buf]);
        nodeID = status.MPI_SOURCE;
        // Get chunk
        while(!V[nodeID].empty() && used[V[nodeID].front()]) V[nodeID].pop();
        // With data locality
        if(!V[nodeID].empty()){
            chunkID = V[nodeID].front(), V[nodeID].pop();
            used[chunkID] = true;
            start_time[chunkID] = time(nullptr);
            Log(log_file, "Dispatch_MapTask", chunkID, nodeID);
            MPI_Send(&chunkID, 1, MPI_INT, nodeID, 0, MPI_COMM_WORLD);
        }
        // Without data locality
        else{
            while(used[Q.front()]) Q.pop();
            chunkID = Q.front(), Q.pop();
            used[chunkID] = true;
            start_time[chunkID] = time(nullptr);
            Log(log_file, "Dispatch_MapTask", chunkID, nodeID);
            sleep(delay);
            MPI_Send(&chunkID, 1, MPI_INT, nodeID, 0, MPI_COMM_WORLD);
        }
    }
    // Send mapper task terminate flag
    int remain_worker = worker_num;
    while(remain_worker--){
        MPI_Recv(&buf, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        if(buf != -1) Log(log_file, "Complete_MapTask", buf, -1, start_time[buf]);
        nodeID = status.MPI_SOURCE, buf = -1;
        MPI_Send(&buf, 1, MPI_INT, nodeID, 0, MPI_COMM_WORLD);
    }
    ```

  - Receive reducer requests from nodes

    Reducer和mapper類似，等待Tasktracker發出request，再送reducer id過去，最後再發送一遍terminate的訊息。

- Tasktracker
  - Mapper thread

    以while迴圈來不斷向Jobtracker發出request，直到收到terminate的訊息，而收到的chunk會發送給開啟的pthread作執行。

```
int buf = -1;
MPI_Status status;
pthread_t *mapper = new pthread_t[cpu_num-1];
MapperArg *mapperArg = new MapperArg[cpu_num-1];
while(true){
    MPI_Send(&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    MPI_Recv(&buf, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    if(buf == -1) break;
    int line_start = (buf-1)*chunk_size;
    auto record_list = InputSplit(input, line_start);
    for(int i = 0; i < cpu_num-1; i++){
        mapperArg[i].threadId = i;
        mapperArg[i].record = &record_list;
        pthread_create(&mapper[i], nullptr, &ThreadJob, static_cast<void*>(&mapperArg[i]));
    }
    for(int i = 0; i < cpu_num-1; i++){
        pthread_join(mapper[i], nullptr);
    }
}
delete mapper;
delete mapperArg;
```

  - Reducer thread

    與mapper thread相似，用while迴圈不斷發出request，直到收到terminate訊息。

- Mapper

  每個mapper thread會平均分配chuck裡的每個record來執行，其中包含了map、partition和store等三個動作。

```
void* ThreadJob(void *arg){
    MapperArg A = *static_cast<MapperArg*>(arg);
    for(int i = A.threadId; i < chunk_size; i += cpu_num-1)
        MapperThread(&((*A.record)[i]));
    return nullptr;
}
void* MapperThread(void *arg){
    auto [line_id, line_text] = *static_cast<std::pair<int, std::string>*>(arg);
    // Map function
    auto mp = Map(line_text);
    pthread_mutex_lock(&pair_num_lock);
    pair_num += mp.size();
    pthread_mutex_unlock(&pair_num_lock);
    // Partition fucntion
    std::vector<std::vector<std::pair<std::string, int>>> V(num_reducer);
    for(auto& [str, cnt] : mp){
        int reducer_id = Partition(str);
        V[reducer_id].push_back({str, cnt});
    }
    // Store data to tmp_file
    for(int i = 0; i < num_reducer; i++){
        pthread_mutex_lock(&tmp_file_lock[i]);
        for(auto& [str, cnt] : V[i])
            tmp_file[i] << str << " " << cnt << "\n";
        pthread_mutex_unlock(&tmp_file_lock[i]);
    }
    return nullptr;
}
```

- Map

  在Map裡很簡單的將此record裡的string統計數量，存至一個map中回傳既可。

  ```
  std::unordered_map<std::string, int> Map(std::string& s){
      s += " ";
      std::unordered_map<std::string, int> res;
      int start = 0;
      for(int i = 0; i < s.size(); i++){
          if(isalpha(s[i])) continue;
          if(i != start) res[s.substr(start, i-start)]++;
          start = i+1;
      }
      return res;
  }
  ```

- Partition

  Partition我使用了內建的 `std::hash` ，再mod reducer的數量。

  ```
  int Partition(const std::string& s){
      return std::hash<std::string>{}(s)%num_reducer;
  }
  ```

- Store

  儲存時，需要用 `pthread_mutex_lock` 來保護每個檔案不被多個thread同時做write。

  ```
  // Store data to tmp_file
  for(int i = 0; i < num_reducer; i++){
      pthread_mutex_lock(&tmp_file_lock[i]);
      for(auto& [str, cnt] : V[i])
          tmp_file[i] << str << " " << cnt << "\n";
      pthread_mutex_unlock(&tmp_file_lock[i]);
  }
  ```

- Shuffle

  我這邊的Jobtracker所做的shuffle為將每個零碎的file以reducer id做合併起來。

  ```
  void Shuffle(){
      for(int i = 0; i < num_reducer; i++){
          std::string out_filename = std::string(output_dir)+"/tmp_"+std::to_string(i)+".txt";
          std::ofstream output_file(out_filename, std::ofstream::out|std::ofstream::trunc);
          for(int j = 1; j < mpi_num; j++){
              std::string in_filename = std::string(output_dir)+"/tmp_"+std::to_string(j)+"_"+std::to_string(i)+".txt";
              std::ifstream input_file(in_filename);
              std::string s;
              while(getline(input_file, s))
                  output_file << s << "\n";
              input_file.close();
              remove(in_filename.c_str());
          }
          output_file.close();
      }
  }
  ```

- Reducer

  Reducer thread需要執行sort、group、reduce和output等四個步驟。

```
void* ReducerThread(void *arg){
    ReducerArg A = *static_cast<ReducerArg*>(arg);
    auto data = *A.data;
    // Sort function
    std::sort(data.begin(), data.end(), SortComparator);
    // Group function
    auto group = Group(data);
    // Reduce function
    auto reduce = Reduce(group);
    // Output function
    Output(reduce, A.reducerId);
    return nullptr;
}
```

  - Sort

    我使用 `std::sort` 來做排序，並搭配 `SortComparator` 來決定排序方法，方便任意修改。

```
std::sort(data.begin(), data.end(), SortComparator);
bool SortComparator(const std::pair<std::string, int>& a, const std::pair<std::string, int>& b){
    return a.first < b.first;
}
```

  - Group

    我為了讓group可以在修改 `GroupComparator` 後自動決定此group的key為最小的key，因此我以一個 `DisjointSet` 搭配 `SortComparator` 來實作，每個group的key即為每個 `DisjointSet` 的parent。

```
std::vector<std::pair<std::string, std::vector<int>>> Group(
    std::vector<std::pair<std::string, int>>& data){
    int n = data.size();
    DisjointSet ds(SortComparator);
    for(int i = 0; i < n; i++) ds.set(data[i].first);
    for(int i = 0; i < n; i++)
        for(int j = i+1; j < n; j++)
            if(GroupComparator(data[i].first, data[j].first))
                ds.join(data[i].first, data[j].first);
    std::vector<std::pair<std::string, std::vector<int>>> res;
    std::unordered_map<std::string, int> seen;
    for(auto& [str, cnt] : data){
        std::string p = ds.find(str);
        if(!seen.count(p)){
            seen[p] = res.size();
            res.push_back({p, std::vector<int>{cnt}});
        }
        else res[seen[p]].second.push_back(cnt);
    }
    return res;
}
bool GroupComparator(const std::string& a, const std::string& b){
    return a == b;
}
```

  - Reduce

    Reduce的部分也相對簡單，直接將同個group裡的所有數值加總起來。

```
std::vector<std::pair<std::string, int>> Reduce(
    std::vector<std::pair<std::string, std::vector<int>>>& data){
    int n = data.size();
    std::vector<std::pair<std::string, int>> res(n);
    for(int i = 0; i < n; i++){
        res[i].first = data[i].first;
        res[i].second = std::accumulate(data[i].second.begin(), data[i].second.end(), 0);
    }
    return res;
}
```

- Output

  Output將資料寫進reducer id對應的file中。

```
void Output(std::vector<std::pair<std::string, int>>& data, int reducerId){
    std::string out_filename = std::string(output_dir)+"/"+std::string(job_name)+"-"+std::to_string(reducerId)+".out";
    std::ofstream output_file(out_filename, std::ofstream::out|std::ofstream::trunc);
    for(auto& [str, cnt] : data)
        output_file << str << " " << cnt << "\n";
    output_file.close();
}
```

- Log

  我log中的時間是直接以 `time(nullptr)` 並轉成 `unsigned long long` 取得，再輸出對應的資訊。

```
unsigned long long start = time(nullptr);
```

- Challenge
  - 發送task時，一開始不太知道該怎麼讓所有worker都知道chunk發完了，後來才想說讓每個worker都多發一次request，jobtracker再各別回復terminate，這樣就可以順利執行了。
  - Mapper thread的部分我一開始使用作業提供的Pthread Pool，並在每次chunk結束後teminate和join，但我發現這樣會讓一些pedding的task也一起被取消掉，所以我就改成自己create pthread和分配工作內容給每個thread，就能正常執行。
  - 寫tmp file時，一開始沒有發現不同thread同時寫的問題，讓每次執行的結果都不一樣，後來才發現要加上lock就沒問題了。
  - Group的部分，為了讓他可以隨意改 `GroupComparator` 後，也能產生對應的key，想了蠻久的，最後才決定用 `DisjointSet` 來實作。

Experiment & Analysis

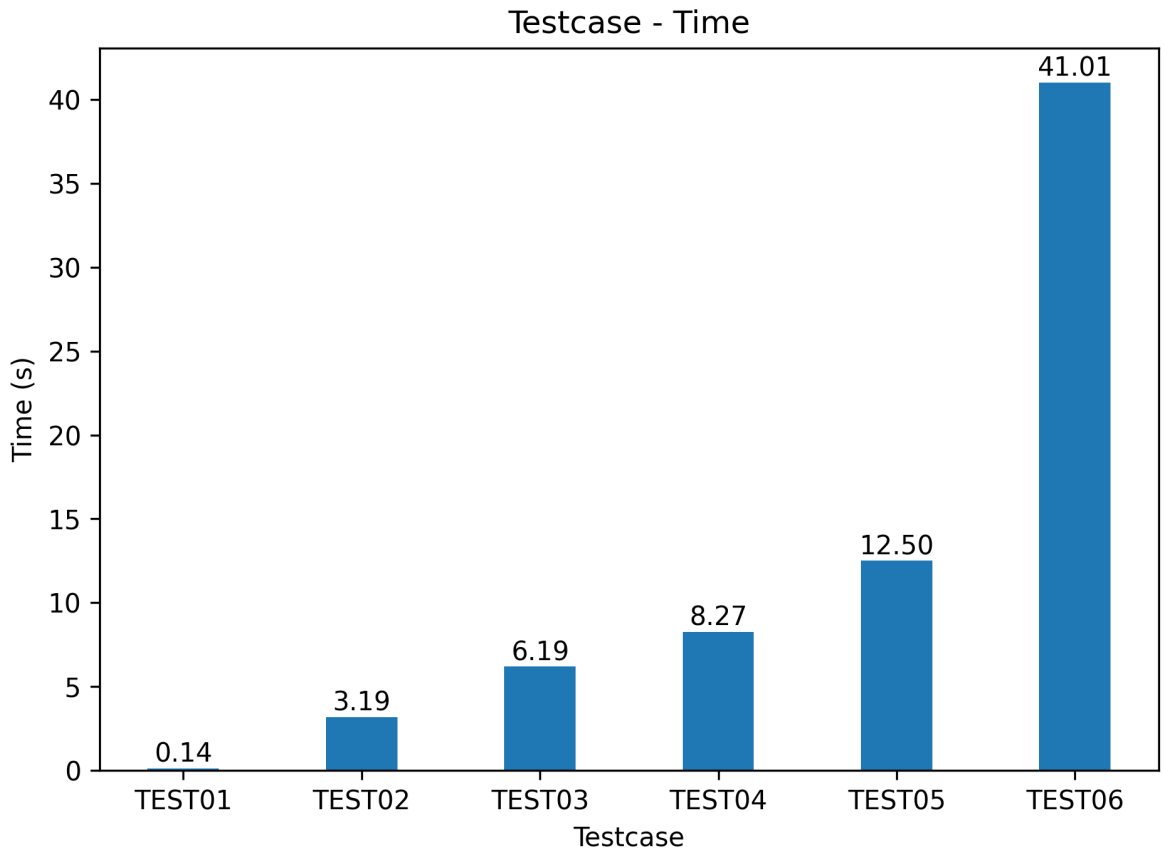
System Spec

我的測試是在課程所提供的 `apollo` 上做的。

Testcase

以下為作業所提供的6筆測資的執行時間。

- Result



Data Locality

以下為data locality的實驗，我定義locality強度的方式為將所有chunk平均分散在不同數量的node上，以下面的實驗為例，我使用的測資為 `TEST05` ，使用的node總數量為4，強度最弱的為將所有chunk放在其中1個node之中，強度最強的為將所有chunk平均放在4個node之中。

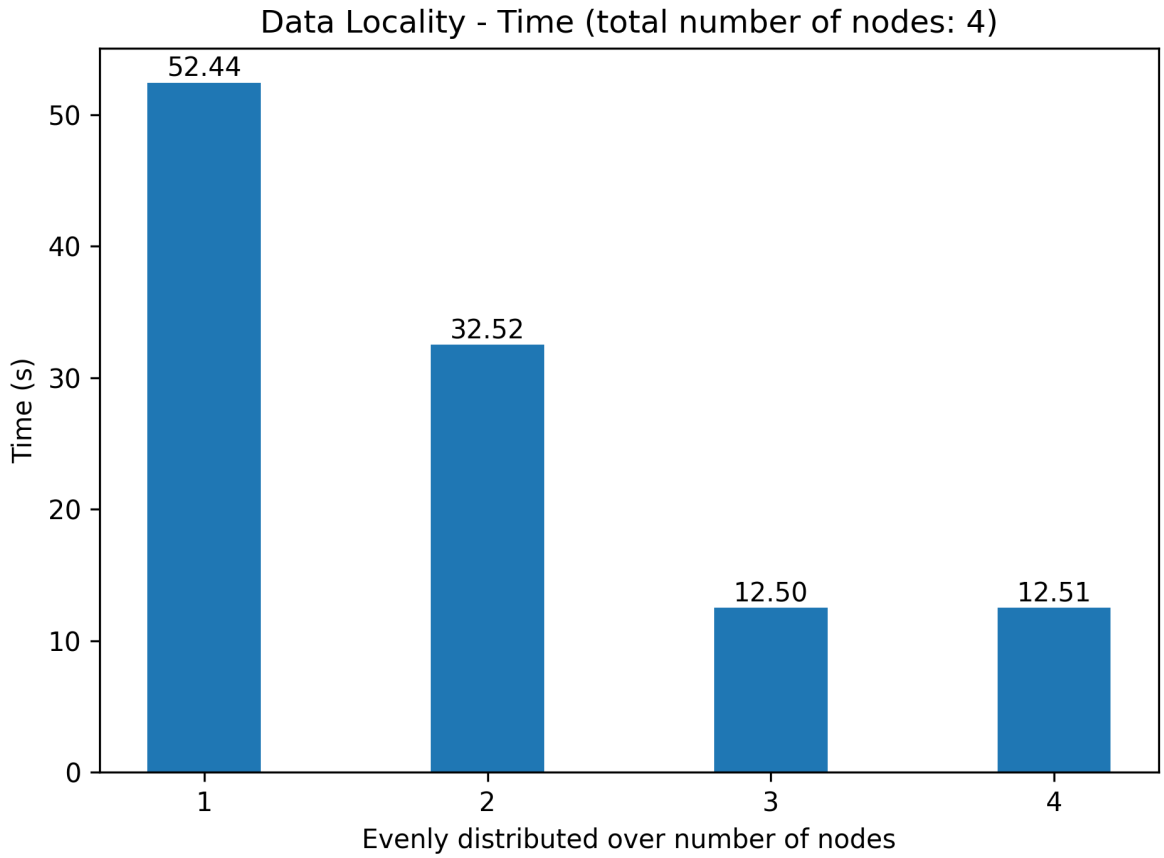
在執行時間上可以看到，data locality對於執行時間影響蠻大的，很差的locality會讓程式跑得很慢，隨著locality增加即有好轉，但當大部分node都有chunk的時候，再繼續分散chunk所能加快的程度有限，如以下分散在3個和4個node的執行時間是差不多的。

- Distribution of chunk

  在 `TEST05` 中共有20個chunk，因此每個node所擁有的chunk數量如下。

	Setting 1	Setting 2	Setting 3	Setting 4
Node 1	20	10	7	5
Node 2	0	10	7	5
Node 3	0	0	6	5
Node 4	0	0	0	5

- Result

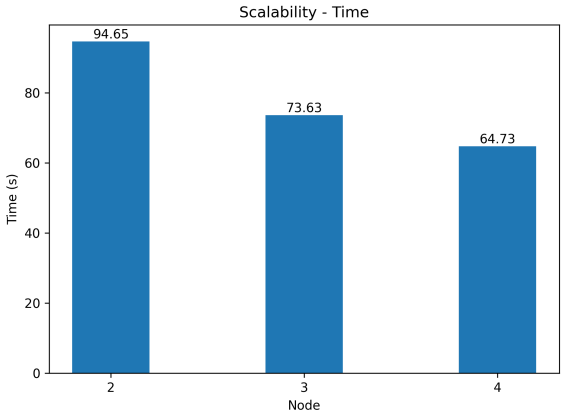


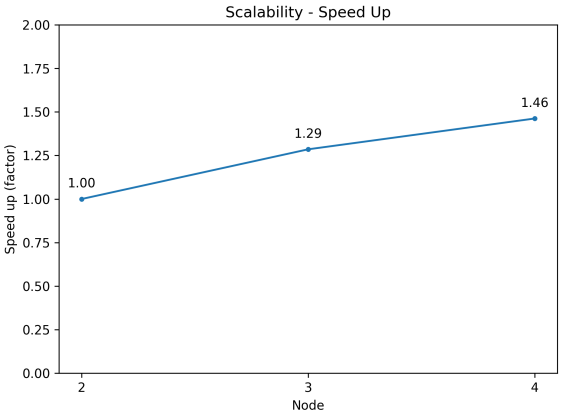
Scalability

以下為scalability的實驗，分別在2~4個node上執行，沒有1個node是因為其中1個node為jobtracker，至少必須再有1個node負責tasktracker。

在這裡我以 `TEST06` 作為測資，可以看出執行時間有隨著node增加而減少，但減少的程度並不高，speed up程度也偏低，我認為主要是因為在分散式計算中，bottleneck卡在node間的資料傳遞和溝通，越多node就代表locality的程度通常也越低，增加的communication也讓總執行時間的增加程度不高。

- Result





Experience & conclusion

What have you learned from this homework?

我覺得這次作業讓我對MapReduce有更深的了解，雖然課堂上的理論和概念都能了解，但在實作時就會發現很多小細節需要去處理，像是jobtracker和tasktracker之間的MPI溝通，一不小心就有可能讓程式無法正確結束，還有像是pthread pool的使用和thread之間可能需要用到一些lock等都需要清楚知道自己在做甚麼，才能夠確保程式的正確執行。而且在系統程式設計上必須更加清楚整個架構，每個小動作都可能連環影響到其他地方的執行。