Hw3 report			
學號: 109062233	姓名: 蘇裕恆		繳交時間:

Implementation :

```
debug.py
job_tracker.cc
job_tracker.h
Makefile
mapreduce.cc
run.sh
task_tracker.cc
task_tracker.h
```

我總共創造了幾個 file，分別為 job_tracker , mapreduce 與 tasktracker，在比較容易 debug 的形況下比較符合真實在會出現的，而非只有一個單一的 file。

```
if(rank == size - 1){
    // show the default config
    std::cout << "[job tracker] job name : " << job_name << " \nnum_reducer : " << num_reducer << \
    "\ndelay :" << delay << "\ninput_filename : " << input_filename << "\nchunk_size : " << chunk_size << \
    "\nlocality_config : " << locality_config_filename << "\noutput_dir : " << output_dir  << std::endl;
    // assign the JobTracker for the first rank
    JobTracker job_Tracker(argv , cpu_num , size , log_out);
    job_Tracker.Assign_job();
    job_Tracker.Shuffle();
    job_Tracker.Assign_Reduce();
}
else{
    TaskTracker task_Tracker(argv , cpu_num , size , rank );
    task_Tracker.required_job();
    task_Tracker.required_reduce();
}
MPI_Finalize();
```

在 map reduce 裡面，我將最後一個 node 設為 job tracker，而剩餘的 node 都為 tasktracker。
我們可以從 spec 裡面發現。

“The jobtracker is responsible for generating the map tasks, reducing tasks of a MapReduce job and following the data-locality scheduling principle to dispatch tasks on worker nodes for execution.”

因此，我在 jobTracker 裡面總共有三個階段，AssignJob (代表著 assign map task) ， Shuffle (進行 hash 跟並將他結果存在 home directory) 和 Assign_Reduce (代表著 assign reduce task)。
同時，taskTracker 會一直先 required map job (require job) 與 required reduce (尋找 reduce job)。

我們先進入 job tracker

```cpp
JobTracker::JobTracker(char **argv , int cpu_num , int mpi_size , std::ofstream * log_out){
    this->job_name = std::string(argv[1]);
    this->num_reducer = std::stoi(argv[2]);
    this->delay = std::stoi(argv[3]);
    this->input_filename = std::string(argv[4]);
    this->chunk_size = std::stoi(argv[5]);
    this->locality_config_filename = std::string(argv[6]);
    this->output_dir = (argv[7]);
    this->cpu_num = cpu_num;
    this->mpi_size = mpi_size;
    this->num_of_data_trunk = 0;
    this->log_out = log_out;
    // this->global_start = std::chrono::steady_clock::now();
    *this->log_out << std::time(nullptr) << ",Start_Job," << job_name << "," << mpi_size << ","  << cpu_num <
    num_reducer << "," << delay << "," << input_filename << "," << chunk_size << "," << locality_config_filen

    // start read
    std::ifstream input_file(this->locality_config_filename);
    std::string line;
    while (getline(input_file, line)) {
        size_t pos = line.find(" ");
        int chunkID = stoi(line.substr(0, pos));
        int nodeID = stoi(line.substr(pos+1)) % (mpi_size-1);
        this->map_tasks.push_back(std::make_pair(chunkID, nodeID));
        num_of_data_trunk++;
    }
    std::cout << "[job tracker] the Job Tracker finish the split \n" ;
}
```

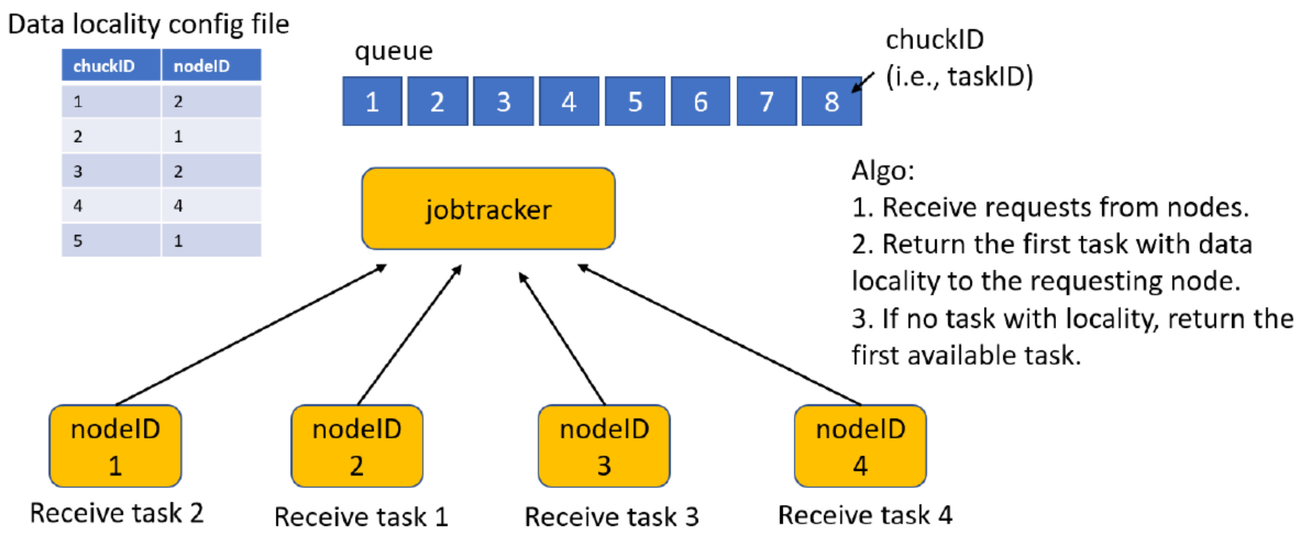
我們可以看到，除了一般的 initialize 以外，我們會直接從 input line readline 近來，並將他用空格做分開 因為(If the nodeID is larger than the number of worker nodes, mod the nodeID by the number of worker nodes.) 所以我們會將他 mode (size – 1) 這也是為甚麼我將 job tracker 設在 node 為 size – 1 得原因，最後將他統一放入 map_tasks 裡面 （ 代表總共有幾個 map tasks 要做 ） 在同時記錄有幾個 data trunk。

JobTtracker::Assign_job()

```cpp
while(!map_tasks.empty()) {
    MPI_Recv(&request_rank,1,MPI_INT , MPI_ANY_SOURCE , REQUEST , MPI_COMM_WORLD , &status);
    std::pair<int,int> target;
    bool find_the_same = false;
    // iter through the queue
    for(auto it = map_tasks.begin(); it != map_tasks.end(); it++){ // chunkID , nodeID
        if(it->second == request_rank){
            find_the_same = true;
            target = *it;
            map_tasks.erase(it);
            break;
        }
    }
    if(!find_the_same){
        target = map_tasks.front();
        // implement the pop front
        map_tasks.erase(map_tasks.begin());
    }
    send_req[0] = target.first;
    send_req[1] = target.second;
    std::cout << "[job tracker] send the chunk " << target.first << " with the number " <<  target.second << " to " <<  r
    *this->log_out << std::time(nullptr) << ",Dispatch_MapTask," << request_rank << "," << target.first  << "\n";
    MPI_Send(&send_req, 2 , MPI_INT , request_rank , MSG::DISPATCH_MAP,MPI_COMM_WORLD);
}
```

基本上，map task 跟 reduce task 的模式很接近，我都是使用 thread pool 的方式來實作。

在 job tracker 裡面，根據 spec，他會一直接收到 request。因此，我們會需要做 mpi_recv。而這邊需要進行的是將 nodeID 相同的在一起，我這邊是用 `std::vector<std::pair<int, int>>` 來實作，因為 vector 的 data type 比較適合做中間 index 的 pop。



如果找不到一樣的，就從一開始 pop 出一個 item 來 send 出去，給 request rank (node) 。

```
}
// end up sending
std::cout << "[job tracker] all map_job done \n" ;
send_req[0] = -1;
send_req[1] = -1;
for(int i = 1; i < mpi_size ; i++){
    MPI_Recv( &request_rank, 1, MPI_INT, MPI_ANY_SOURCE, REQUEST, MPI_COMM_WORLD, &status);
    time_span = std::chrono::duration_cast<std::chrono::duration<double>>(std::chrono::steady_clock::now() - start[
    *this->log_out << std::time(nullptr) << ",Complete_MapTask," << request_rank << "," << time_span.count() << "\
    MPI_Send( &send_req, 2 , MPI_INT , request_rank , MSG::DISPATCH_MAP,MPI_COMM_WORLD);
}
```

同時，如果今天 data 都已經分配出去並且處理完成，那就是將 -1 送出給 request node。

```
enum MSG{
    REQUEST , DISPATCH_MAP, REQUEST_REDUCE ,DISPATCH_REDUCE ,
};
```

在實作上，我使用了 enum 來做為哪中 type 的 send。在美觀的同時也保證了程式的可讀性。

將到他怎麼 assign job ，就必須提及 task tracker 是如何 send request 的。

```
void TaskTracker::required_job(){
    for(int i = 0 ; i < cpu_num - 1; i++){
        pthread_create(&this->Mapper_threads[i], nullptr, TaskTracker::map_pool, (void*)this);
    }
    int recv_arr[2];
```

在 tasktracker，我們會先將 mapper thread 全部 assign 到 map_pool 裡面。這邊沒有使用助教給的 threadpool 是因為當時誤會了，因為遇到了一點困難，到最後使用手刻。帶 difficulty 也會提及。

```
int to_work_on = -1;
while(true){
    pthread_mutex_lock(&Self->Mapper_mutex_job);
        while(Self->map_task_queue.empty()){
            pthread_cond_wait(&Self->Mapper_cond_job, &Self->Mapper_mutex_job);
        }
        to_work_on = Self->map_task_queue.front();
        Self->map_task_queue.pop();
    pthread_mutex_unlock(&Self->Mapper_mutex_job);
    if(to_work_on != -1){
        std::cout << "[Task tracker] Node : " << Self->which_node <<  " Work on :" << to_work_on << std::endl;
        auto return_from_split = Input_Split(to_work_on , Self->chunk_size , Self->input_filename);
        auto from_Map = MapFunction(return_from_split);

        std::ofstream out("./mapper_intermediate_" + std::to_string(to_work_on)  + ".txt");
        // output format word , value , hash value
        for(auto it : from_Map){
            out <<  it.first << " " << it.second << " "<< hash_function(it.first, Self->num_reducer) << "\n";
        }
        out.close();
        pthread_mutex_lock(&Self->Mapper_mutex_job);
        Self->Map_thread_cnt--;
        usleep(2000);
        pthread_cond_signal(&Self->Mapper_cond_com);
        pthread_mutex_unlock(&Self->Mapper_mutex_job);
    }
    else{
        std::cout << "[Task tracker] " << Self->which_node << " Received the end notation "  << std::endl;
        break;
    }
```

在 pool 裡面，若今天要處理的 queue 是 empty 的話，他就會 call condition wait 並將 threa put to sleep。相反的，有 task 要執行的話，他就會先去 fetch 那個 task id (注意，因為我們在 JOB TRACKER 結束後設為 -1 ，因此若是 task_queue 裡面有 -1 則代表整體結束。並會退出整個 while loop。) 接著將得到的資料已 (word , value , hash value) 的方式存到 intermediate file。

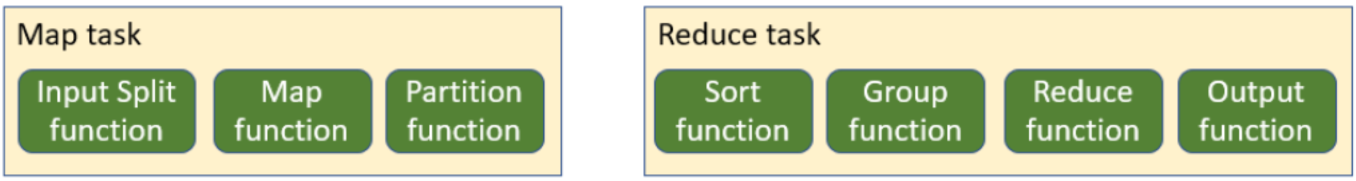
為了確保程式的正確性，我們在找到 task 與作整體而言有幾個 map thread 的部分，都是在 critical section 進行。

```
int recv_arr[2];
while(1){
    pthread_mutex_lock(&Mapper_mutex_com);
    if(Map_thread_cnt >= cpu_num - 1 ){
        pthread_cond_wait(&Mapper_cond_com, &Mapper_mutex_com);
    }
    pthread_mutex_unlock(&Mapper_mutex_com);
    MPI_Send(&(this->which_node),1,MPI_INT , job_tracker_node , MSG::REQUEST , MPI_COMM_WORLD);
    MPI_Recv(&recv_arr, 2 , MPI_INT , job_tracker_node , MSG::DISPATCH_MAP , MPI_COMM_WORLD , MPI_STATUS_IGNORE);
    if(recv_arr[0] == -1){
        pthread_mutex_lock(&Mapper_mutex_com);
        std::cout << "[Task tracker] " << this->which_node << " : finish receiving "  << std::endl;
        // to indicate the tasks are over
        this->map_task_queue.push(recv_arr[0]);
        usleep(2000);
        pthread_cond_signal(&Mapper_cond_job);
        pthread_mutex_unlock(&Mapper_mutex_com);
        break;
    }
    else{
        // Map_Args* arg = new Map_Args(recv_arr[0]);
        if(recv_arr[1] != this->which_node){
            sleep(delay);
            std::cout << "[Task tracker] delay !!!!" << this->which_node << " : received " << recv_arr[0] << std::endl;
        }
        else{
            std::cout << "[Task tracker] " << this->which_node << " : received " << recv_arr[0] << std::endl;
        }
        pthread_mutex_lock(&Mapper_mutex_com);
        // ( cpu_num - 1 ) mapper threads
        this->map_task_queue.push(recv_arr[0]);
        Map_thread_cnt++;
        // after the put into the queue wake up the processor
        usleep(2000);
        pthread_cond_signal(&Mapper_cond_job);
        std::cout << "[Task tracker] " << this->which_node << " : signal the task " << recv_arr[0] << std::endl;
        pthread_mutex_unlock(&Mapper_mutex_com);
        // Mapper_Pool->addTask(new ThreadPoolTask(&map_function, (void*)arg));
    }
}
```

再回到 function 本身，整體而言。會先看整體 map thread 是否大於上限 ，若是，則將整個 condition variable 進入 wait。並將他 halt 掉。

不然的話，會一直 send request 到本來的 job tracker 裡面。接著，接收從 job tracker 出來的 request，並且做一個 delay 的行為 (optional) 。

若是今天他有將 task push 進去 queue 裡面，則會同時 signal cond var。



接著講一下各自設定的 function

Input split function:

```
std::map<int, std::string> Input_Split(int chunk , int read_lines , std::string filename){
    std::map<int, std::string> buffer;
    std::ifstream input(filename);
    // ignore
    std::string line;
    for(int i = 0 ; i < read_lines*(chunk - 1) ; i++){
        std::getline(input, line);
    }
    for(int i = 0 ; i < read_lines ; i++){
        std::getline(input, line);
        buffer.insert({chunk * read_lines + i , line});
    }
    return buffer;
}
```

Input : num to be read , filename
Output : std::map<int, std::string>
作法先getline讀到應該要讀取的地方 (第一個for loop) 接著在將他跑過需要讀的lines並將他傳到buffer裡面。

Map function:

```
std::map<std::string, int> MapFunction(std::map<int, std::string> input) {
    std::map<std::string, int> output;
    int cnt = 0;
    for (const auto& record : input) {
        std::string line = record.second;
        std::istringstream iss(line);
        std::string word;
        while (iss >> word) {
            cnt++;
            if (output.count(word) > 0) {
                output[word]++;
            } else {
                output[word] = 1;
            }
        }
    }
    //std::cout <<  "[Function Info] : MapFunction : " <<  cnt << std::endl;
    return output;
}
```

You can assume the data type of the input records is int (i.e., line#), and string (i.e., line text), and the data type of the output records is string (i.e.,word), and int (i.e., count).
Input : num to be read , filename
因為之前是一行一行的讀近來，因此我們就會用一個 map 來看他有沒有到讀過，如果有的話就加一 沒有的話就 = 1

Partition function:

```
int hash_function(std::string input , int num_reducer){
    std::hash<std::string> hasher;
    return hasher(input) % num_reducer;
}
```

我使用的是 std::hash 他可以將任意 input 轉乘 hash。並且在最後會 % num_reducer。

```
std::ofstream out("./mapper_intermediate_" + std::to_string(to_work_on)  + ".txt");
// output format word , value , hash value
for(auto it : from_Map){
    out <<  it.first << " " << it.second << " "<< hash_function(it.first, Self->num_reducer) << "\n";
}
```

注意 ： 在 mapper 的最後，他會將 intermediate file 傳出去，並在之後回收使用。

```
void JobTracker::Shuffle(){
    std::cout << "[job tracker] : Start shuffle "<< std::endl;
    int kv_count = 0;
    int test_cnt = 0;
    std::vector<std::vector<std::pair<std::string,int>>> data( num_reducer , std::vector<std::pair<std::string,int>>());
    std::string word;
    int count;
    int hash;

    std::chrono::duration<double> time_span;
    auto start = std::chrono::steady_clock::now();

    for(int i = 1 ; i <= this->num_of_data_trunk ; i++){
        std::ifstream in("./mapper_intermediate_" + std::to_string(i)  + ".txt");
        while (in >> word >> count >> hash) {
            kv_count++;
            data[hash].push_back(std::make_pair(word, count));
        }
        in.close();
    }
    *this->log_out << std::time(nullptr) << ",Start_Shuffle," <<  kv_count <<  "\n";

    std::cout << "[job tracker] : KV Total count "<< kv_count <<std::endl;
    for(int i = 0 ; i < this->num_reducer ; i++){
        std::ofstream out("./mapper_reducer_" + std::to_string(i)  + ".txt");
        for(auto it : data[i]){
            test_cnt++;
            out << it.first << " " << it.second << "\n";
        }
        out.close();
        reduce_tasks.push(i);
    }
    std::cout << "[job tracker] : Test Total count "<< test_cnt <<std::endl;
    auto end = std::chrono::steady_clock::now();
    time_span = std::chrono::duration_cast<std::chrono::duration<double>>(end - start);
    *this->log_out << std::time(nullptr) << ",Finish_Shuffle," <<  (int)time_span.count() <<  "\n";
}
```

在 shuffle 這邊，我們會先將 word , count , hash 傳進去，而他的 data tpye 為。

```
std::vector<std::vector<std::pair<std::string,int>>>
```

因為我們需要依據 hash 的結果將 reducer 的結果產生到 home directory。接著再讓 reducer 使用這個 file 並做出相對應的處理。

待完成後就可以準備將 reducer task 傳給 reducer。

```
}
while(!reduce_tasks.empty()) {
    MPI_Recv(&request_rank,1,MPI_INT , MPI_ANY_SOURCE , MSG::REQUEST_REDUCE , MPI_COMM_WORLD , &status);
    target = reduce_tasks.front();
    reduce_tasks.pop();
    std::cout << "[job tracker] send the reduce " << target << " to " <<  request_rank << std::endl;
    *this->log_out << std::time(nullptr) << ",Dispatch_ReduceTask," << request_rank << "," << target  << "\n";
    MPI_Send(&target, 1 , MPI_INT , request_rank , MSG::DISPATCH_REDUCE,MPI_COMM_WORLD);
}
// end up sending
std::cout << "[job tracker] Reduce : all reduce done \n" ;
target = -1;
```

做法跟 map task 一樣，唯一差別是不需要將因為沒有 locality，因此不需要傳 node 數量給 tasktracker。

```
while(true){
    pthread_mutex_lock(&Self->Reducer_mutex_job);
        while(Self->assigned_task.empty()){
            pthread_cond_wait(&Self->Reducer_cond_job, &Self->Reducer_mutex_job);
        }
        to_work_on = Self->assigned_task.front();
        Self->assigned_task.pop();
    pthread_mutex_unlock(&Self->Reducer_mutex_job);
    if(to_work_on != -1){
        std::cout << "[Task tracker] Node : " << Self->which_node <<  " Work on Reduce:" << to_work_on << std::endl;
        auto data = extract_data(to_work_on , Self->output_dir);
        data = Sorting_function(data);
        auto group_results = group_function(data);
        auto final_output = reduce_function(group_results);
        std::ofstream out(Self->output_dir + "/" + Self->job_name + "-" + std::to_string(to_work_on) + ".out");
        for(auto it : final_output){
            out << it.first << " " << it.second << "\n";
        }
        out.close();
        pthread_mutex_lock(&Self->Reducer_mutex_job);
        Self->Reduce_thread_busy = false;
        usleep(2000);
        pthread_cond_signal(&Self->Reducer_cond_com);
        pthread_mutex_unlock(&Self->Reducer_mutex_job);
    }
```

因為這邊做法，基本上就會 map task 一樣。但唯一的差別就是他們的 function 不一樣。

```
std::vector<std::pair<std::string,int>> extract_data(int reducer , std::string output_dir){
    std::vector<std::pair<std::string,int>> output;
    std::string word;
    int count;
    int word_cnt;
    std::ifstream in("./mapper_reducer_" + std::to_string(reducer)  + ".txt");
    while (in >> word >> count ) {
        word_cnt++;
        output.push_back(std::make_pair(word,count));
    }
    in.close();
    std::cout << "[extract_data] :size " << word_cnt << "\n";
    return output;
}
```

首先 extract data ,

Input : reducer number

Output: std::vector<std::pair<std::string,int>>

基本上就是將之前的資料讀出來，(每個 reducer 讀取需要負責的 hash 檔案的 file)

Sort function:

```
std::vector<std::pair<std::string, int>> Sorting_function(std::vector<std::pair<std::string, int>> input) {
    std::sort(input.begin(), input.end(), [](const std::pair<std::string,int> &a, const std::pair<std::string,int> &b) {
        return a.first < b.first;
    });
    return input;
}
```

依據字典續將 word 由小排到大。

Group function:

```
std::map<std::string, std::vector<std::pair<std::string, int>>> group_function(std::vector<std::pair<std::string, int>> input
    std::map<std::string, std::vector<std::pair<std::string, int>>> output;
    std::cout << "[group_function] :size " <<input.size() << "\n";
    for(auto element: input) {
        if(output.find(element.first) == output.end())
            output[element.first] = std::vector<std::pair<std::string, int>>();
        output[element.first].push_back(element);
    }
    return output;
}
```

用一個 map 將 word 設為 key，並將一樣的 key 放到一起。並且回傳

Reduce function:

```
std::vector<std::pair<std::string, int>> reduce_function(const std::map<std::string, std::vector<std::pair<std::string, int>>> &input) {
    std::vector<std::pair<std::string, int>> output;
    for (auto it = input.begin(); it != input.end(); it++) {
        std::string key = it->first;
        int count = 0;
        auto value = it->second;
        for (const auto &p: value)
            count += p.second;
        output.push_back({key, count});
    }
    return output;
}
```

將同一個 group 裡面的東西加在一起，並且回傳一個 vector<string , int> 當成最終答案

最後再將整個 reduce 完的東西寫出去，整個 mapreduce 就完成了。

challenges in your implementation :

1. 型別問題:

   在 class 裡面，若要 class pthread 的話

   ```
   static void* map_pool(void* input) ;
   static void* reduce_pool(void* input) ;
   ```

   他的型別必須是要 static void* 而也是因為一開始搞不懂這點，因此才會拒絕選擇使用 ta 的 code。 (最後手刻到一半才發現好像差不多)

   " When you pass static void* map_function(void* arg) as the first argument to the ThreadPoolTask(void* (*f)(void*), void* arg) function, the type of the argument is void* (*)(void*).

   So, ThreadPoolTask function is expecting a function pointer as the first argument and the static void* map_function(void* arg) is a function pointer that meets that requirement.

   On the other hand, if void* map_function(void* arg) is passed as the first argument , the ThreadPoolTask function will receive a function, not a function pointer and it will not match the expected argument type and it will cause a compile error. "

2. Time measurement :
   也是同樣的問題，若是在 construct 的時候將 begin 設為

```
// this->global_start = std::chrono::steady_clock::now();
```

   並在最後 destuctor 的時候將他相減，不知道為甚麼他就是不會 work。
   Solution : 在 main 裡面做 calculation
3. Log File generating

```
// this->global_start = std::chrono::steady_clock::now();
*this->log_out << std::time(nullptr) << ",Start_Job," << job_name << "," << mpi_size << ","
num_reducer << "," << delay << "," << input_filename << "," << chunk_size << "," << locality_
```

   一開始也是出現了很多問題，因為就算是用 ofstream(…,std::ios::app) 他一樣會 overwrite 掉。造成 log file 無法正確。(因為在 main 裡面也需要將時間算出來並寫進去)
   最後，就是將 log file 當成 parameter 傳入 class 內，才解決這個問題。
4. 不同的型別問題

```
void* TaskTracker::reduce_pool(void *input){
    // keep on working
    TaskTracker* Self = (TaskTracker*)input;
```

   在 static 裡面，因為有諸多限制，因此若直接 call class 裡面的 variables 會有 error。因此，找了許多文獻後，發現要將自己放到裡面，之後 call 的時候從 self 來抓，才不會有問題。
5. Delay 問題
   因為有時候太快的話會產生一些 work on : -2345646
   推測是因為太快了，我在 signal 前都加了 usleep 來保證他不會因為還沒有 sync 就抓取資料。

Experiments

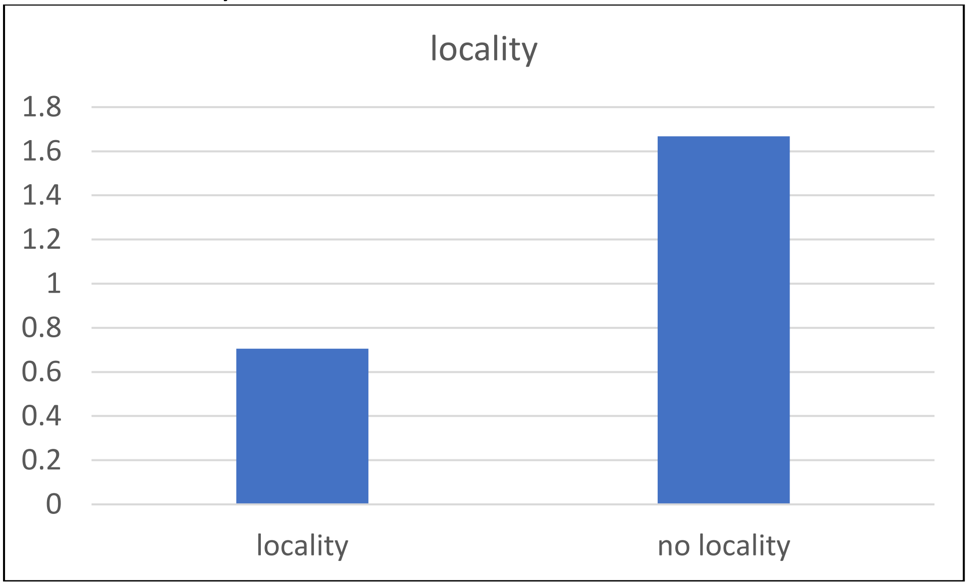
以下的testcase base:

```
Experiment 1 : {
    "NODES": 4,
    "CPUS": 8,
    "JOB_NAME": "TEST04",
    "NUM_REDUCER": 9,
    "DELAY": 1,
    "INPUT_FILE_NAME": "04.word",
    "CHUNK_SIZE": 10,
    "LOCALITY_CONFIG_FILENAME": x
}
```

```
Experiment 2 : {
    "NODES": x
    "CPUS": 8
    "JOB_NAME": "TEST06",
    "NUM_REDUCER": 9,
    "DELAY": 0
    "INPUT_FILE_NAME": "06.word",
    "CHUNK_SIZE": 20,
    "LOCALITY_CONFIG_FILENAME": "06.loc"
```

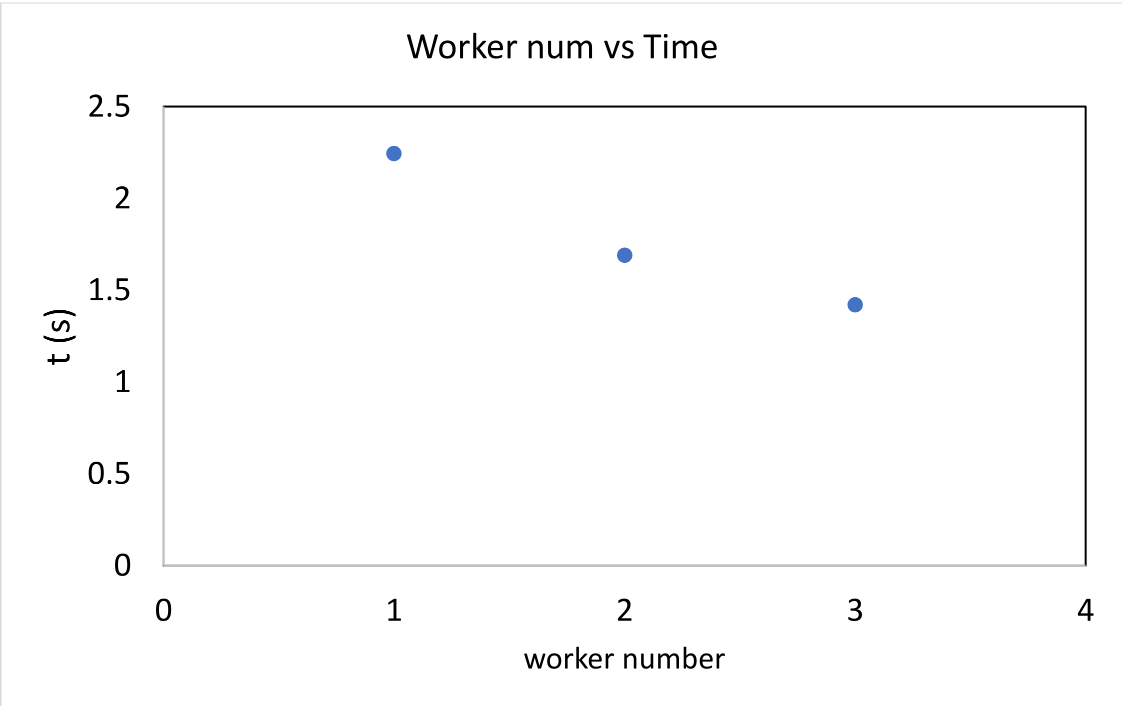
```
}
```

1. data locality

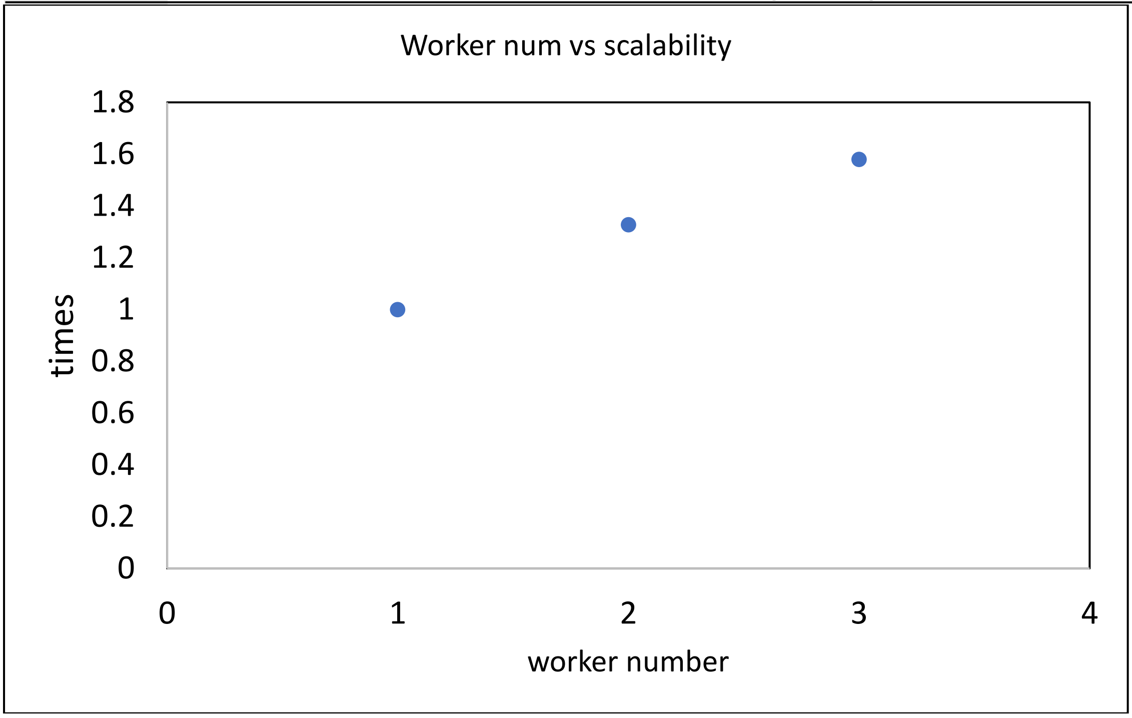


雖然說有差，但說到底的話就只是因為一個多做了一次 sleep 因而造成這樣的差距。(因為程式跑太快，因此當一個卡住以後，另外一個就把剩下的做完了) 不過我們還是可以看到 scalibility 的差距

2. scalability



可以看到他的時間依據 worker 數下降了



但是他的 scalability 不好，推測是因為我有 call usleep 因此讓她無法完全的展示 scalablity。

Experience & conclusion:

1. What have you learned from this homework?

學到了很多關於 c++ class 的細節，與一個 mapreduce framework 是如何被實踐的。