




# Parallel Programming Hw4 Report

107062228 陳劭愷

NTHU-Parallel-Programming/hw4 at master · justin0u0/NTHU-Parallel-Programming  
CS, NTHU. 2021 CS542200, Parallel Programming. Contribute to justin0u0/NTHU-Parallel-Programming development by creating an account on GitHub.

 <https://github.com/justin0u0/NTHU-Parallel-Programming/tree/master/hw4>

justin0u0/**NTHU-Parallel-Programming**

CS, NTHU. 2021 CS542200, Parallel Programming

Rk 1 Contributor 0 Issues ☆ 0 Stars 0 Forks

## [Implementation](#)

[ThreadPool](#)

[Mapper](#)

[Split](#)

[Map](#)

[Partition](#)

[Write](#)

[Reducer](#)

[Fetch](#)

[Merge](#)

[Group](#)

[Reduce](#)

[Write](#)

[TaskTracker](#)

[JobTracker](#)

[Serve](#)

[Run](#)

[Others](#)

[Message](#)

[Experiment](#)

[Conclusion](#)

[References](#)

## Implementation

實作部分太簡單的部分就不多說明並且不貼出程式碼。

### ThreadPool

[https://github.com/justin0u0/NTHU-Parallel-Programming/blob/master/hw4/thread\\_pool.h](https://github.com/justin0u0/NTHU-Parallel-Programming/blob/master/hw4/thread_pool.h)

`class ThreadPool` 使用 pthreads 實作了 thread pool，並且對外提供 `terminate` 函數可以對 `ThreadPool` 進行 gracefully shutdown。對外提供 `addTask` 函數可以動態的新增工作（將工作加到

thread pool 中的 queue) 給 thread pools 使用。內部則是不斷的從 queue 中拿出工作並且執行。因為要執行的函數以及參數都需要被儲存在 queue 中，有使用到 function pointer 的儲存，queue 儲存的資料結構如下：

```
1  class ThreadPoolTask {
2      friend class ThreadPool;
3
4  public:
5      ThreadPoolTask(void* (*f)(void*), void* arg) : f(f), arg(arg) {}
6  private:
7      void* (*f)(void*);
8      void* arg;
9  };
10
```

因為多個 threads 會同時進行 `removeTask` 以拿到下一個要做的工作，因此 `removeTask` 透過 mutex lock 來保護 critical section。並且，若當前 queue 中工作為空，為避免 busy waiting，透過 conditional variable `removeCond` 來等待直到有 `addTask` 的執行。

當 `terminating` 被設成 true 時則直接 unlock 並回傳 `nullptr` 告知不要再繼續執行下一個 task。

```
1  ThreadPoolTask* removeTask() {
2      pthread_mutex_lock(&mutex);
3
4      while (tasks.empty() && !terminating) {
5          // sleep until addTask notify
6          pthread_cond_wait(&removeCond, &mutex);
7      }
8
9      if (terminating) {
10         pthread_mutex_unlock(&mutex);
11         return nullptr;
12     }
13
14     ThreadPoolTask* task = tasks.front();
15     tasks.pop();
16     pthread_cond_signal(&addCond);
17
18     pthread_mutex_unlock(&mutex);
19
20     return task;
21 }
```

`ThreadPool` 中，每一個 threads 執行的工作如下：

```
1 static void* run(void* arg) {
2     ThreadPool* pool = (ThreadPool*)arg;
3
4     while (!pool->terminating) {
5         ThreadPoolTask* task = pool->removeTask();
6
7         if (task != nullptr) {
8             // do the task works
9             (*(task->f))(task->arg);
10
11             delete task;
12         }
13     }
14
15     return nullptr;
16 }
```

對外提供的 `addTask` 與 `removeTask` 相似，在 queue 內工作數量  $\geq$  設定的 `bufferSize` 時，透過 conditional variable `addCond` 等待直到有 `removeTask` 執行。

```
1 void addTask(ThreadPoolTask* task) {
2     pthread_mutex_lock(&mutex);
3
4     while (tasks.size() >= bufferSize && !terminating) {
5         // sleep until removeTask notify
6         pthread_cond_wait(&addCond, &mutex);
7     }
8
9     if (terminating) {
10         pthread_mutex_unlock(&mutex);
11         return;
12     }
13
14     tasks.push(task);
15     pthread_cond_signal(&removeCond);
16
17     pthread_mutex_unlock(&mutex);
18 }
```

`ThreadPool::terminate` 函數會將 `terminating` 設成 `true` 並使用 `pthread_cond_broadcast` 將所有等待的 thread 喚醒。

## Mapper

<https://github.com/justin0u0/NTHU-Parallel-Programming/blob/master/hw4/mapper.h>

`class Mapper` 實作並提供 `run` 函數可以逐次執行 MapReduce 中 mapper phase 的 `split`、`map`、`partition` 以及最後寫入到 intermediate file 的 `write` 函數。

## Split

`Mapper::split` 首先讀出 input file，根據 spec 上面的說明若是當前 mapper 運行的 node 與資料不在同一個 node 上面，模擬等待  $D$  秒後再繼續。接著跳過不屬於自己的行數之後，將 `chunkSize` 行讀進 memory 中。

## Map

`Mapper::map` 將 split 讀進來的一個個 string 專換成自定義的 `KV` 資料結構。回傳一個 `KV` 的陣列。

```
1  class KV {
2  private:
3      unsigned int hashCode;
4  public:
5      std::string key;
6      int value;
7
8      KV() {}
9
10     KV(const std::string& key, int value) : key(key), value(value) {
11         hashCode = std::hash<std::string>()(key);
12     }
13
14     bool operator < (const KV& rhs) const {
15         return key.compare(rhs.key) < 0;
16     }
17
18     unsigned int hash(unsigned int modulus) const {
19         return hashCode % modulus;
20     }
21 };
```

## Partition

在 `KV` 資料結構中，定義了 hash function，透過 hash function 就可以得到此筆資料要分配給哪一個 reducer。

## Write

將 `Mapper::partition` 分組後的結果寫入到 intermediate file。

## Reducer

<https://github.com/justin0u0/NTHU-Parallel-Programming/blob/master/hw4/reducer.h>

`class Reducer` 實作並提供 `run` 函數可以逐次執行 MapReduce 中 reducer phase 的 `fetch`、`merge`、`group`、`reduce` 以及最後寫入到 output file 的 `write` 函數。

## Fetch

`Reducer::fetch` 會從所有的 mapper output intermediate files 中找到屬於自己要處理的 record 儲存到 local 的資料結構中。

使用的資料結構是 `std::vector<std::list<KV>>`。其中每一個 `std::list` 都儲存著一個 mapper output file 的結果。

## Merge

從 Hadoop 的 [source code](#) 中可以看到 reducer 在讀完 mapper 的 output file 後，因為每個 mapper 的輸出結果都是已經排序好的，因此可以透過線性時間的 merge 來將多個 mapper 的 output file 排序成一個排序好的 `KV` 資料（若有  $K$  個檔案且總長為  $N$ ，則花費  $O(KN)$  的時間）。

Hadoop 的實作中，reducer 並不需要等到所有 mapper 完成才會開始讀檔，但在我的實作中為了方便，reducer 會等到所有 mapper 完成後才開始讀檔。但是也因此可以透過每次將兩兩合併的方式將時間複雜度降到  $O(N \log_2 K)$ 。並且透過 inplace merge 的方式不使用額外的記憶體空間來合併資料，這也是為什麼儲存時會使用 `std::list` 這樣的資料結構。

```

1  VectorKV* merge(std::vector<ListKV*>* input) {
2      int len = (int)input->size();
3
4      for (int i = 1; i < len; i <= 1) {
5          for (int j = 0; j + i < len; j += (i <= 1)) {
6              // inplace merge list[j] and list[j + i] into list[j]
7
8              ListKV* lhs = input->at(j);
9              ListKV* rhs = input->at(j + i);
10
11             ListKV::iterator rit = rhs->begin();
12
13             for (ListKV::iterator lit = lhs->begin(); lit != lhs->end(); ++lit) {
14                 while (rit != rhs->end() && (*rit) < (*lit)) {
15                     lhs->emplace(lit, *rit);
16                     ++rit;
17                 }
18             }
19
20             while (rit != rhs->end()) {
21                 lhs->emplace(lhs->end(), *rit);
22                 ++rit;
23             }
24
25             delete rhs;
26         }
27     }
28
29     ListKV* l = input->at(0);
30     VectorKV* output = new VectorKV(l->begin(), l->end());
31
32     delete l;
33     delete input;
34
35     return output;
36 }

```

## Group

在 Hadoop 的 group 階段，reducer 會透過 `groupingComparator` 將資料進行分組，但是如果是全部排序後再進行分組，那先前排的順序不就沒有用途了嗎？透過觀察 Hadoop 的 source code 以及網路上的資訊，原來實際上 `group` 並不進行任何排序操作，只是依序取出 `KV` 後比較當前的 key 與前一個 key，若比較結果為 0 則認為是同一個 group。

```

1  VectorKVs* group(VectorKV* input) {
2      VectorKVs* output = new VectorKVs;
3
4      for (const KV& kv : (*input)) {
5          if (output->empty() || output->back().compare(kv) != 0) {
6              output->emplace_back(kv.key);
7          }
8          output->back().values.emplace_back(kv.value);
9      }
10
11     delete input;
12
13     return output;
14 }

```

## Reduce

對每個 group 的 `kvs` 分別進行 sum aggregation。

```

1  VectorKV* reduce(VectorKVs* input) {
2      VectorKV* output = new VectorKV;
3      output->reserve(input->size());
4
5      for (const KVs& kvs : (*input)) {
6          // reduce by sum aggregation
7          int sum = 0;
8          for (int value : kvs.values) {
9              sum += value;
10         }
11         output->emplace_back(kvs.key, sum);
12     }
13
14     delete input;
15
16     return output;
17 }

```

## Write

將 `Reducer::reduce` 的結果寫入到 output file 中。

## TaskTracker

[https://github.com/justin0u0/NTHU-Parallel-Programming/blob/master/hw4/task\\_tracker.h](https://github.com/justin0u0/NTHU-Parallel-Programming/blob/master/hw4/task_tracker.h)

`TaskTracker` 負責管理 `mapperPool` 以及 `reducerPool` 兩個 `ThreadPool`。

`TaskTracker` 在把 `mapperPool` 以及 `reducerPool` 啟動後，首先會向 `JobTracker` 請求一個 mapper 以及一個 reducer task，並進入 `TaskTracker::serve` 的無限迴圈中。在 `serve` 中，會不斷呼叫 `MPI_Recv` 等待 `JobTracker` 的訊息。

從 `JobTracker` 來的訊息會分為以下三種：

- `MessageType::MAP`： `JobTracker` 收到 mapper task 的請求後分配給此 `TaskTracker` 一個 mapper task，包含 mapper task 要處理的 `taskId` 以及 mapper task 本身的 `id`。收到此訊息後會創立一個新的 `Mapper` 並將 task 送到 `mapperPool` 中。
- `MessageType::REDUCE`：與上述一樣，不過是 reducer task 的部分。
- `MessageType::TERMINATE`：由 `JobTracker` 告知已經結束，不需要再等待並且關閉 `TaskTracker`。

```
1 while (!terminating) {
2     MPI_Recv(resp.raw, MESSAGE_SIZE, MPI_INT, JOB_TRACKER_NODE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
3
4     switch (resp.data.type) {
5         case MessageType::MAP: {
6             // dispatch mapper task
7
8             Mapper* mapper = new Mapper(resp.data.id, resp.data.taskId, nodeId, config, &TaskTracker::callback);
9             mapperPool->addTask(new ThreadPoolTask(&Mapper::run, mapper));
10
11             usleep(10000);
12             requestMapperTask();
13
14             break;
15         }
16         case MessageType::REDUCE: {
17             // dispatch reducer task
18
19             Reducer* reducer = new Reducer(resp.data.id, resp.data.taskId, config, &TaskTracker::callback);
20             reducerPool->addTask(new ThreadPoolTask(&Reducer::run, reducer));
21
22             usleep(10000);
23             requestReducerTask();
24
25             break;
26         }
27         case MessageType::TERMINATE: {
28             terminating = true;
29             break;
30         }
31     }
32 }
```

`TaskTracker` 在 `serve` 結束後會對底下的 `ThreadPool` 進行關閉的動作，等待 `ThreadPool` 關閉後結束進程並且通知 `JobTracker` 自己已經關閉。

## JobTracker

[https://github.com/justin0u0/NTHU-Parallel-Programming/blob/master/hw4/job\\_tracker.h](https://github.com/justin0u0/NTHU-Parallel-Programming/blob/master/hw4/job_tracker.h)

`JobTracker` 主要負責分配任務給 `TaskTracker`，實作中有兩個線程分別是 `run` 以及 `serve`。



## Serve

`JobTracker::serve` 會不斷的呼叫 `MPI_Recv` 等待任何 `TaskTracker` 的訊息。

從 `TaskTracker` 們來的訊息分別有：

- `MessageType::MAP`：來自 `TaskTracker` 的 mapper task 請求，將此請求加到 queue 中。
- `MessageType::SHUFFLE`：來自 mapper task，代表 shuffle 的開始。
- `MessageType::REDUCE`：來自 `TaskTracker` 的 reducer task 請求，將此請求加到 queue 中。
- `MessageType::MAP_DONE`：來自 mapper task 的結束訊息。
- `MessageType::SHUFFLE_DONE`：來自 reducer task，代表 shuffle 結束。
- `MessageType::REDUCE_DONE`：來自 reducer task 的結束訊息。
- `MessageType::TERMINATE`：來自 `TaskTracker` 的結束訊息。

`JobTracker::serve` 會在收到所有 `TaskTracker` 的結束訊息後關閉。

## Run

`JobTracker::Run` 負責分配任務。

首先不斷從 mapper task queue 中拿出 mapper task 請求，依照 data locality 分配。

```
1 // find data with locality
2 std::list<int>::iterator it;
3 for (it = tasks.begin(); it != tasks.end(); ++it) {
4     if (config->localityConfig[*it] == nodeId) {
5         break;
6     }
7 }
8
9 // if no matching data with the node, pick the first one
10 if (it == tasks.end()) {
11     it = tasks.begin();
12 }
13
14 int taskId = *it;
15 tasks.erase(it);
```

Scheduling 的 algorithm 依照 spec 上面的說明，優先選擇有 locality 的任務，若沒有則挑選第一個任務。挑選後送訊息給對應的 `TaskTracker`。

等待所有 mapper task 完成後，按照請求的順序分配 reducer task，最後等待 reducer task 完成後發送 `MessageType::TERIMATE` 的訊息給每一個 `TaskTracker`。最後等待 `serve` 關閉後結束。

## Others

### Message

<https://github.com/justinOu0/NTHU-Parallel-Programming/blob/master/hw4/types.h#L178-L195>

在使用 MPI 做 communication 時，因為想要傳送的訊息為 struct 但是 MPI 自定義 type 有點麻煩，因此使用 `union` 將想傳送的 struct map 到底層的 integer array 中。如此一來就可以直接使用 integer array 來送訊息，同時可以使用 struct 的資料結構。

```
1 // Message defines the message send between task tracker and job tracker
2 union Message {
3     struct {
4         MessageType type;
5
6         // mapper/reducer ID
7         int id;
8
9         // task ID
10        // - the chunk ID for the mapper
11        // - the partition ID for the reducer
12        int taskId;
13
14        // additional data carried
15        int data;
16    } data;
17
18    int raw[MESSAGE_SIZE];
19 };
```

## Experiment

# of Data Nodes	Job time (seconds)	With locality (%)
3	39	97.9%
2	47	66.7%
1	62	18.3%

實驗使用自己生成的大測資，delay 設為 3 秒。With locality 代表有多少 mapper 使用到有 locality 的資料。

可以發現當有 locality 越強時，執行時間就越短。

## Conclusion

這次 MapReduce 作業讓我學習到 MapReduce 這個框架的很多設計精髓，有很多原本不知道如何設計的部分在參考了 Hadoop 的一些實作後也更加清楚。另外還練習到了 multi-threads programming 的很多技巧，包含 thread pool 的使用，mutex lock 與 conditional variable 的搭配。受益良多。

另外實驗時發現偶爾會有突然 runtime error 的問題，追了一陣子後才發現原來 `MPI_Send` 預設並不是 `thread safe` 的，因此改了一些 code 讓 send 的地方有被 mutex lock 保護。

## References

---

<https://shengyu7697.github.io/std-mutex/>

[https://blog.csdn.net/qq\\_21794823/article/details/108348837](https://blog.csdn.net/qq_21794823/article/details/108348837)

<https://stackoverflow.com/questions/4295432/typedef-function-pointer>

<https://blog.csdn.net/hellojoy/article/details/80592811>

<https://www.cnblogs.com/DarrenChan/p/6773277.html>

<https://github.com/mbrossard/threadpool>