Parallel Programing

HW5 PageRank

程祥恩、ssh: pa128427359

1. Instruction

Compile

在 HW5 目錄底下執行 makefile

Run

刪除先前執行所留下的資料

```
$ hdfs dfs -rm -r Page_Rank/tmp
```

```
$ hdfs dfs -rm -r Page_Rank/Output
```

執行 Page_Rank.jar

```
$ hadoop jar Page_Rank.jar page_rank.Page_Rank INPUT_PATH OUTPUT_PATH ITER_TIMES
```

如未輸入 ITER_TIMES，則當兩次 iterations 之間的誤差小於 0.001 時視為收斂，並輸出結果，終止程式。

2. Implementation

我將 PageRank 分為四個 Job，分別為 Parse、Dangling node、Calculate、Sort，其中 Dangling node 和 Calculate 是需要迭代至收斂的兩個 Job，細節如下。

(a) Parse

Mapper

將讀入的檔案進行 title 和 link 的擷取，每一個 title 會擁有零個(dangling node)、一個或多個 link，**而只要被擷取為 title 的頁面，一定會是 existing page**，而本階段的輸出的<K, V>總共會有以下三種

Output

- <Page, Link>：表示某一 Page 對應到的某一 Link
- <Page, “”>：用來表示 dangling node，沒有任何 link
- <!Page, “Exist”>：在 Page 前面插入一個驚嘆號當前贅字，表示這個 Page 存在，在 Reducer 會用到

Reducer

首先，由於 hadoop 會在 Map->Reduce 這段過程中，自動把 key 根據字母順序排列，因此<!Page, “exist”>系列會被放在最前面。

我們**在 Reducer 新增一個 hashset，用來儲存 existing page**，Reducer 首先判斷只要 key 的前綴字為驚嘆號，就將該 title 加進 hashset 當中。等所有具備前綴驚嘆號的 Pair 都處理完畢後，就開始處理<Page, Link>以及<Page, “”>，Reducer 會判斷每一個 link，只要該 link 不存在(也就是不存在於 hashset)則忽略；若存在，則保留下來。最後將每一個 Page 各自的所有 link 做字串合併，並用分隔符號隔開，同時計算每個 Page 的初始 PageRank(1 / N)，即可一起寫入到 value 中。

Output

- <Page, PageRank 分隔符號 links>

(b) Dangling node

Mapper

將 Parse Reducer 最終輸出的，透過 value 的字串處理找出沒有 link 的 Page(即為 dangling node)，並把這些 Page 的 PageRank 加總，由於 Mapper 會在多台機器上執行，所以不能直接單純的加總，而是要**在個別的機器加總完後，透過 cleanup() 方法 Map 起來，並交給 Reducer 再做第二次的加總。**

Output

- <1, dangling node PageRank 加總> (key 隨意，只要一樣就好)

Reducer

把 Mapper 傳進來的所有 PageRank 再進行第二次的加總，最後透過 context.getCounter()方法設為全域變數，讓下一階段做使用。

(c) Calculate

Mapper

將 Parse Ruducer 輸出的<K, V>格式稍做處理，方便在 Reducer 計算每一個 Page 的 PageRank，輸出的<K, V>有三種格式

Output

- <Page, !>：代表該 Page 存在，用驚嘆號前綴字區分
- <Page, |Link>：代表 Page 連到 Link，用 OR 符號前綴字區分
- <Page, #PagePR>：代表 Page 與他的 PageRank，用井字號前綴字區分
- <Link, PagePR" \t" PageLinkCount>：表示 Page 有連到 Link，儲存 Page 的 PageRank 和 PageLinkCount 是為了計算 Link 的新 PageRank

Reducer

計算所有 Page 的 PageRank

$$PR^{(k)}(x) = (1-\alpha)\left(\frac{1}{N}\right) + \alpha \sum_{i=1}^{n} \frac{PR^{(k-1)}(t_i)}{C(t_i)} + \alpha \sum_{j=1}^{m} \frac{PR^{(k-1)}(d_j)}{N}$$

由於 hadoop 會自動將輸出以 key 排列，所以不用擔心到 Reducer 時順序會亂掉。

首先，若沒有收到<Page, !>，代表該 Page 應該忽略不處理；若收到的是<Page, |Link>，將所有 Link 透過字串合併起來；若收到的是 <Link, PagePR" \t" PageLinkCount>，則用 split()擷取出 PagePR 和 PageLinkCount，即可算出算式的第二項。最後再透過 context.getCounter()，即可取得(1/N)以及 dangling node PR 的和，就能夠算出第一項與第三項，即可算出目前 iteration 每一個 Page 的 PageRank。

最後，若收到的是<Page, #PagePR>，則將之與新算出來的 PR 計算誤差，即可得到 err，並用同樣的方法把 err 寫成全域變數，即可在最外層的迴圈中判斷是否收斂。

最終的輸出必須要把每個 Page 當作 key，PageRank 和 links 當作 value，才能進行下一個 iteration 的運算

Output

- <Page, PageRank 分隔符號 links>

(d) Sort

Mapper

分析 Calculate Reducer 的輸出，擷取出 Page 與 PagePR，並產生一個自訂的 PagePRPair 物件，把這兩個東西存進去，並存放在 key 中，方便 Reducer 排序。

Output

- <PagePRPair, NULL>

Reducer:

由於已經在 PagePRPair 實作 compareTo 方法，因此 hadoop 在載入至 reducer 前，會將所有 Page 按照 PageRank 和字母排列，最後把 PagePRPair 分解為 Page 和 PageRank，即可輸出結果。

Output

- <Page, PageRank>

3. Experiments and Analysis

(a) System spec

CPU: Intel Xeon Quad-Core 3.6GHz * 1

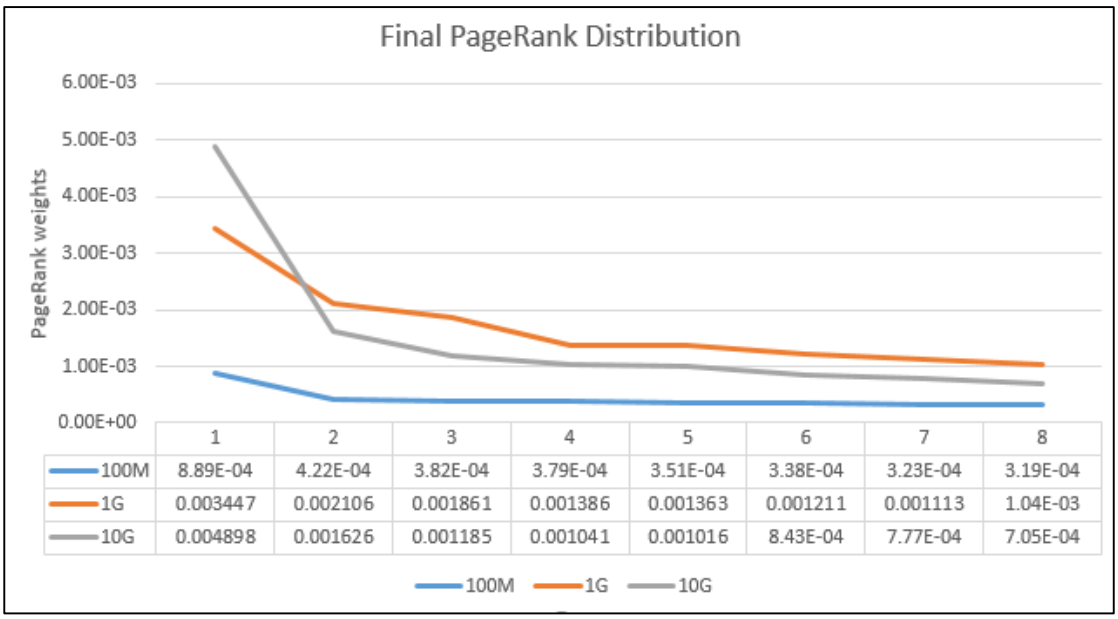
RAM: 16GB

HDD: 1TB

Slave: 6

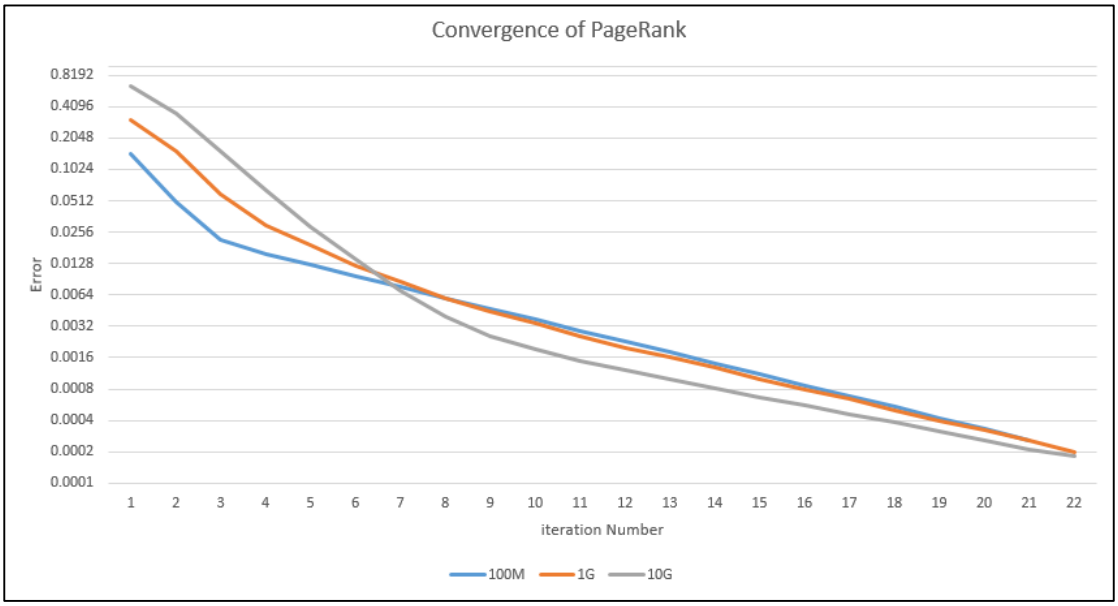
OS: Ubuntu-16.04

(b) Final PageRank Distribution



	1	2	3	4	5	6	7	8
100M	8.89E-04	4.22E-04	3.82E-04	3.79E-04	3.51E-04	3.38E-04	3.23E-04	3.19E-04
1G	0.003447	0.002106	0.001861	0.001386	0.001363	0.001211	0.001113	1.04E-03
10G	0.004898	0.001626	0.001185	0.001041	0.001016	8.43E-04	7.77E-04	7.05E-04

(c) Convergence of PageRank



4. Conclusion

Hadoop 的平行架構和之前幾次的 MPI、openmp、pthread、cuda 不太相同，比方說全域變數，資料的分割、結合和轉換，都需要經過特別的處理才能完成，在寫這份作業的初期是吃足了苦頭，一直被傳統的程式思維給限制住，結果其實很多事情 hadoop 的底層都已經做好了。作業寫到最後才慢慢掌握到 hadoop 的規則，獨立的平行運算交給 Mapper，要整合結果或加總數值就交給 Reducer，反正 hadoop 會自動幫你排列 key，所以不用擔心順序亂掉的問題。