Parallel Programming
Homework 5: Page Rank

王鈺鎔 106062600

Page Rank:

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

Error:

$$err(k) = \sum_{i=1}^{N} \left| PR^{(k)}(x_i) - PR^{(k-1)}(x_i) \right|$$

Implementation

The implementation is composed of 4 phases: (i) parsing pages, (ii) appending outgoings, (iii) calculating ranks, and (iv) sorting ranks.

Parsing Pages (ParsePage)

This phase is to parse the original page codes and extract titles and links without missing links.

- **Mapper:** Mapper parse the original page codes and save titles and corresponding links as <link, title>, i.e. save the transpose graph. Furthermore, mapper also save <title, EXIST_TAG> for each title.
- **Partitioner:** To ensure all the pairs with same key are assigned to the same reducer, partitioner assigns all the pairs with same first char in key to the same reducer. For example, <mango, apple> and <monkey, elephant> will be assigned to the same reducer.
- **Reducer:** If the key has no EXIST_TAG, it is a missing node and reducer should drop out all the pairs with a missing link as their keys. Therefore, reducer only save the pairs with the same key when the key has an EXIST_TAG. The valid pairs will be store as <title, link>.

Appending Outgoings (AppendingOutgoing)

This phase is to save initial rank and append all the outgoings of a title. The output will be either <title, rank> or <title, rank|link1|link2|…>. The former is also a dangling node. The total number of nodes will also be calculated in this phase as a Hadoop counter.

Calculating Ranks (CalculateRank)

This phase is to iteratively calculate rank of each node.

- **Mapper:** Mapper save the rank of the title for each link as <link, rank> if the title is not the dangling node. Since mapper can distinguish dandling nodes and non-dangling nodes easily by splitting values with "|", mapper can calculate the sum of all dangling nodes' rank and store dangling sum as a Hadoop counter.

- **Partitioner:** Same as **ParsePage** and **AppendingOutgoing**.

- **Reducer:** For each key (title), reducer collects all the ranks of their ingoing links, and calculates the page rank for each key. The result page rank will be store with outgoing links of the title as <title, rank> or <title, rank|link1|link2|…> again. To determine when to terminate the iteration, he error is also be computed in this phase as a Hadoop counter.

Sorting Ranks (SortRank)

This phase is to sort the pages by their ranks in descending order or sort the pages lexicographically in ascending order. To sort page (word) and rank (double) at the same time, we implement our own kind datatype `SortPair` to save key, and the value is saved as null.

Performance Optimization

- **Transpose graph:** To easily get rid of missing links, we firstly save transpose edges, <links, title>, and exist tag, <title, EXIST_TAG>. Thus, reducer easily drop out the missing links which has no exist tag.

- **Partition by first char:** To distribute data on more reducers and avoid the information of the same key is distributed on different reducers. Therefore, partitioner assigns all the pairs whose keys' first char are the same to the reducer. There are 30 categories including "A" to "Z", "1", "2", other numbers and the remainder.

- **HashSet:** We use `HashSet` instead of `ArrayList` to get better performance.

- **StringBuffer:** Using `StringBuffer` instead of `ArrayList` and `String` to append strings.

Experiment & Analysis

Analyze the converge rate

Fig.1 shows the convergence of page rank. The y axis uses Logarithmic scale with base 2. We found that lines are like the liner, especially when the data size is getting smaller. That is, the error almost decreases exponentially in the next iteration, and t**he converge rate is almost exponential**. The reason is that the components of page rank multiply α and $(\alpha - 1)$ every iteration, so that page ranks has an **ordinary generating function**. The function can be seen as a **probability-generating function** since page rank can also be seen as the probability that you visit a page.

Another interesting thing is that the error decreases more sharply when the data size is more. The reason is that when the data size is getting more, there fewer dangling nodes in our dataset. The ratio of the number of total pages and dangling nodes is shown in fig. 2. There are **fewer dangling nodes** in the dataset with bigger data size, so it needs less iterations to converge, i.e. the **converge rate is higher**.

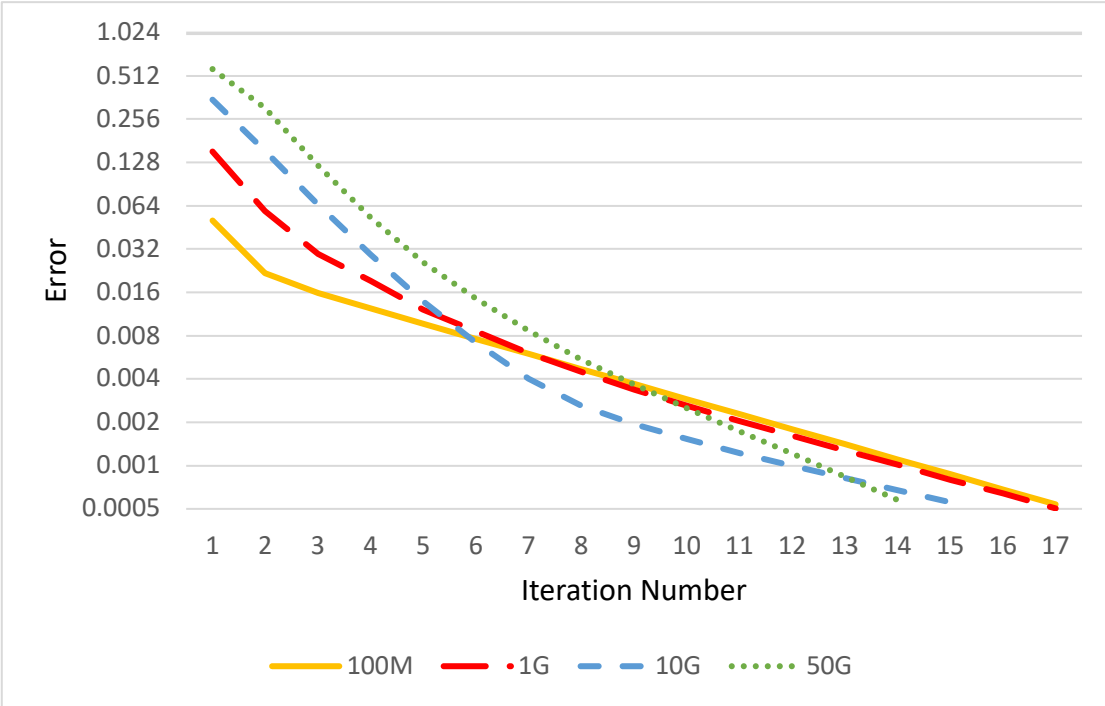

Figure 1. Convergence of Page Rank

Data	#page	#dangling node	Ratio
100M	30727	27832	0.906
1G	313500	246234	0.785
10G	3133027	1523520	0.486
50G	15982471	1030507	0.065

Figure 2. Number of pages and dangling nodes and Ratio

Experiences / Conclusion

In this homework, we implement page rank and analyze some wiki pages to learn how to handle big data with Hadoop. According our analysis, the converge rate is close to exponential and the fewer dangling nodes, the higher the converge rate.

In this homework, the most difficult thing is to get rid of missing links. The reason is that I tried to drop out all the missing links in one map-reduce process. However, it will cause too many missing links since the target pages of some links exist in other reducers but these links are judged as the missing links.