PLSA Search Engine

2008 Parallel Programming

Department of Computer Science and Information Engineering,

National Taiwan University

Professor: Pangfeng Liu

Sponsored by Google

B92902036	**Shoou-Jong (John) Yu**	**余守中**
B94902007	**Shih-Ping (Kerry) Chang**	**張詩平**
B94902062	**Kai-Yang Chiang**	**江愷陽**
B94902063	**Yu-Ta (Michael) Lu**	**呂鈺達**
B94902065	**Shoou-I Yu**	**余守壹**

Contents

Introduction

Introduction to PLSA

Probabilistic latent semantic analysis (PLSA) is a statistical technique for the analysis of co-occurrence data. In contrast to standard latent semantic analysis, which stems from linear algebra and shrinks the size of occurrence tables (number of words occurring in some documents), PLSA is based on probability and statistics to derive a latent semantic model.

Instead of the traditional key-word based data classification, PLSA tries to classify data to its "latent semantic". It's about learning "what was intended" rather than just "what actually has been said or written". After performing the PLSA classification, words which often come together in a same document will be seen as highly connected to each other, and the documents which contain these words therefore will be classified into the same "topic". The whole PLSA process can be divided into two parts, which are corpus classification and the query fold-in. Both parts use the expectation maximization (EM) theory. After running tens of iterations, we can get the final result.

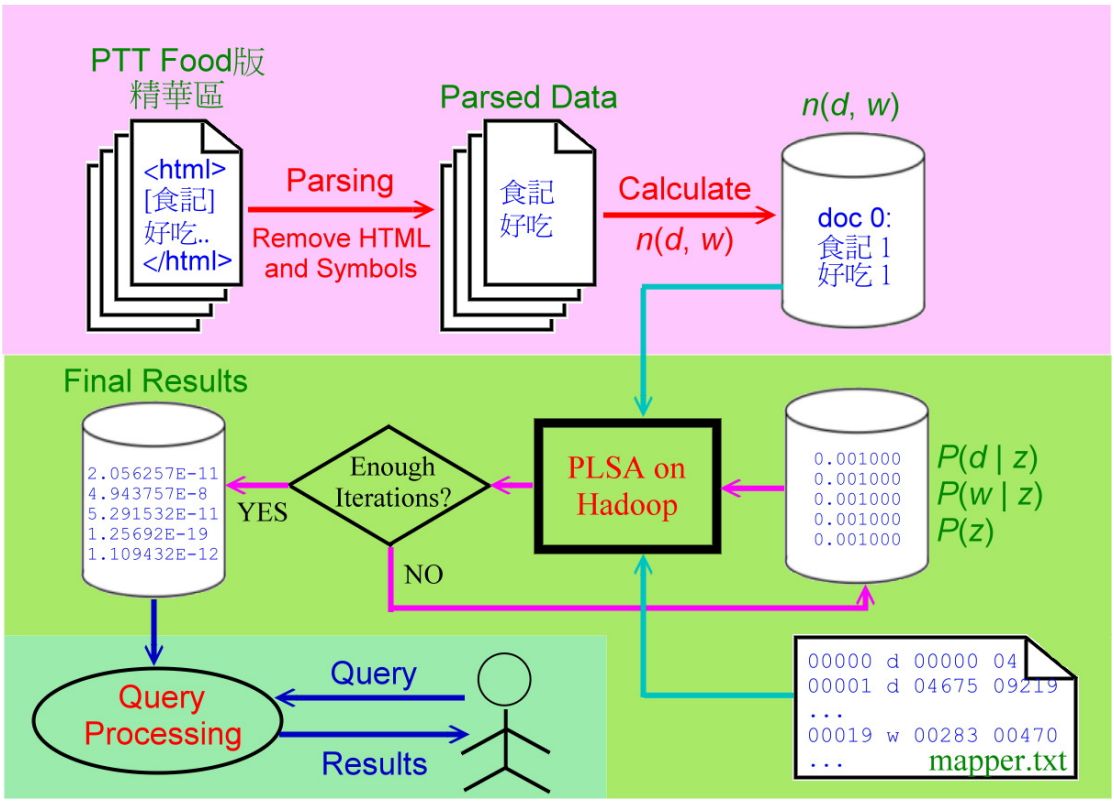
PLSA can be used in many areas, such as information retrieval or machine learning, to improve the original results.

Motivation

The PLSA search engine was the final project that two of our teammates planed to do for the Digital Speech Processing class in 2007. However, when they started to implement the theory into code, they found that it takes too long to obtain the PLSA model because it is constructed by a huge amount of data. In order to finish the project in limited time, they were forced to reduce the size of the corpus data. Because of that, the outcome of the search engine was not accurate enough and the topic size was also limited. So we came up with an idea of using parallel programming to speed it up and hopefully by using a larger corpus data, we can get a better result.

Implementation

The following diagram shows the flow of our project.



d: document, w: word, z: topic

We use the PTT Food board as the data source of our project. PTT is the biggest BBS site in Taiwan, and the Food board contains postings about people's experience in various restaurants around Taiwan. After parsing the data so that some useless symbols are removed, we divided the sentences into several two-letter words and calculated the $n(d,w)$ (number of word w appearing in document d) of each document. Then we use the information and the given initial values to construct a PLSA model. This is the part where parallel programming takes place. After running a fixed number of iterations, we can get the final results. Finally, we made a query interface for demonstration.

Below we shall explain in detail the flow and implementation of the whole project. Note that in the two sections that details the two algorithms we have implemented, each section will be organized as follows: "Overview" will provide a quick introduction of the algorithm; "Implementation" section will outline the detail of how PLSA is implemented under Map-Reduce; "Design Process" will discuss in detail the underlying philosophy of the implementation, along with other design tradeoffs and choices we have made that gave rise to the final implementation mentioned in the previous section

Pre-Processing

We wanted to get our training data from one of Taiwan's largest BBS site: PTT (telnet://ptt.cc). However, fetching data directly from BBS is inconvenient. Luckily every board in PTT has an approximate mapping on PTT's website(http://ptt.cc), we retrieve data from the website instead. Though it is not completely up to date with the BBS site, it is enough for our project.

The process is as follows.

1. Because we wanted to make a search engine to search the Food board, we used the tool "wget" on Linux to download all contents in the website's Food directory as well as its subdirectory to our own machine.
2. Now, we have gotten all web pages about food from the web site. Our next step is to extract the contents we need from the original web pages. Since every page is composed of text and html tags, we have to remove all the tags inside the pages. Luckily, because all contents we want are between the tag pair <pre> and </pre> and every page has only 1 tag pair of this kind, we can remove all contents in the pages but the text between <pre> and </pre>.
3. There are a large number of documents in our data folder. However, some of these documents are useless for our purpose, such as indexing pages. We found that most of the pages we need have character counts between some intervals. Therefore, we only keep documents within that interval.
4. Using all the documents, we build a dictionary consisting of 2-character words along with their appearing frequency in all the documents. We achieve this by scanning through all documents. Whenever a new character pair appears in a document, we add the word to our dictionary and set its frequency to be 1. When a character pair is already in the dictionary, we add its frequency by 1. After scanning all documents, we remove words with lower frequency and only keep 20,000 words in the dictionary.
5. The next step is to count how many times a word in the dictionary appears in 1 document. Finally, we have the statistics of the word count of each document. With these statistics and the dictionary file, we are now able to start training the PLSA model.

As for the initial probabilities of $P(d|z), P(w|z), P(z)$, $P(w|z)$ is initialized randomly, while $P(d_i|z)$ and $P(z_i)$ are initialized to $\frac{1}{d}$ and $\frac{1}{z}$ respectively.

Naïve PLSA

Overview

In this first version, we strive to be as intuitive and easy to code as possible. We observe that we have to compute four major components, namely the numerator and denominator of $P(z|d,w)$ and $P(d|z), P(w|z), P(z)$ (they require similar computation), in one iteration of EM process. So we just simply divide the work so that each component is done in one Map-Reduce job. Thus this approach offers us a method to compute one iteration of EM process with four Map-Reduce jobs.

Though this naïve algorithm is simple, it works correctly and this is our first version of PLSA. We will elaborate the implementation detail in next sub section.

Implementation

Consider one iteration in EM process. First, we compute $P(z|d,w)$ by giving $P(d|z), P(w|z), P(z)$. These values come from the outcomes of the previous EM process. After we have $P(z|d,w)$ for each (d,w) pair, we can compute the new $P(d|z), P(w|z), P(z)$ in parallel.

Let's consider the E-step. $P(z|d,w)$ can be computed by the equation:

$$P(z|d,w) = \frac{P(z)\times P(d|z)\times P(w|z)}{\sum_t P(z_t)\times P(d|z_t)\times P(w|z_t)} \quad (1)$$

The algorithm uses 2 rounds of Map-Reduce to obtain all $P(z|d,w)$, one for the numerator and the other for the denominator. The first Map-Reduce is for the denominator. Each mapper will be responsible for an interval of documents, and for each document d_j, it will compute all $P(z_t)\times P(d_j|z_t)\times P(w_i|z_t)$ for each w_i and z_t. Next we set the z value as

the key, thus the reducer can sum up all $P(z_t)\times P(d_j|z_t)\times P(w_i|z_t)$ for the same z_t, output the summation to GFS and finally for each z we obtain a $d\times w$ entry table recording the denominator.

The second pass computes the numerator and $P(z|d,w)$. We compute each $P(z)\times P(d|z)\times P(w|z)$ for a given z in the mapper, and divide the computed denominator and output $P(z|d,w)$ for a given z in the reducer. Therefore we complete the E-step in the first two Map-Reduce passes.

M-step, on the other hand, takes another two Map-Reduce passes. Consider three equations in M-step as the following:

$$P(w|z)=\frac{\sum_j n(d_j,w)\times P(z|d_j,w)}{\sum_j\sum_i n(d_j,w_i)\times P(z|d_j,w_i)} \tag{2}$$

$$P(d|z)=\frac{\sum_i n(d,w_i)\times P(z|d,w_i)}{\sum_j\sum_i n(d_j,w_i)\times P(z|d_j,w_i)} \tag{3}$$

$$P(z)=\frac{\sum_j\sum_i n(d_j,w_i)\times P(z|d_j,w_i)}{\sum_k\sum_j\sum_i n(d_j,w_i)\times P(z_k|d_j,w_i)} \tag{4}$$

The third pass Map-Reduce computes the numerator of $P(w|z)$ and $P(d|z)$. We divide z into several intervals for the mapper. For a given z, we compute $\sum_j n(d_j,w)\times P(z|d_j,w)$ and $\sum_i n(d,w_i)\times P(z|d,w_i)$, throwing their value to the reducer with its type (numerator for $P(w|z)$ or $P(d|z)$) and z as the keys respectively. Next we use a partitioner to partition intermediate value by its type so that we can deal with $P(w|z)$ and $P(d|z)$ in two different reducers. Finally, in the reducer, we output the numerator of $P(w|z)$ or $P(d|z)$ to their respective output files.

The fourth pass is used to compute the numerator of $P(z)$, and to normalize $P(w|z)$, $P(d|z)$ and $P(z)$ (that is, divided by the denominator). The mapper computes the

denominator of the three equations by adding all the $\sum_j n(d_j, w) \times P(z|d_j, w)$ computed in the previous Map-Reduce pass, the reducer does the normalization by loading in $P(d|z)$, $P(w|z)$, divide their value by the normalization factor, and output the normalized result back to the DFS.

Each pass will be sequentially executed by the main method. Except for combining four pass Map-Reduce routine stated above, the main method is also responsible for running the EM process iteratively and setting the input/output file's path for each Map-Reduce.

Design process

This is the first step in our project. Since we have no prior experience in Map-Reduce design, our design process is simple: observe the equation, try to divide the problem to sub-problem, and make it possible to fit the Map-Reduce model.

The main objective of the naïve algorithm is not its performance, but correctness. Therefore, we did not make much effort on optimizing time or space complexity. However, we achieved two important objectives with our naïve algorithm.

The first one is that we found a solution to our problem, proving that PLSA is solvable with the Map-Reduce model. It is important and realistic to find a basic approach at first, and try to improve it afterwards. Second, we got to become familiar with Hadoop and the Map-Reduce model by working on the naïve version. It gave us an opportunity to design our next Map-Reduce algorithm with more understanding of its working details. We were eventually able to call on the experience gained with naïve PLSA to develop the advanced PLSA.

Advanced PLSA

Overview

While the naïve version PLSA produces the correct results, it is obvious from the analysis that there is ample room for improvement. Therefore after completing the naïve version, we embarked on overhauling the implementation of PLSA. In this new version, the most marked improvement is that now one iteration of PLSA is now completed in one iteration, and along with other improvements to reduce computation and I/O, the advanced

PLSA showed an improvement of 74 times over the previous version.

Implementation

The input of mappers consists of two parts. First part is "mapper.txt", which is commonly read and divided amongst all the mappers. "Mapper.txt" looks like the following:

```
…
00002 d 09219 13937
00003 d 13937 18138
…
00020 w 00470 00722
00021 w 00722 01058
```

Fig 1: mapper.txt

Where "00002" denotes the id of the mapper which receives this line, "d 09219 13937" means that this mapper is responsible for calculating $P(d|z)$ for documents 9219 to 13937. Similarly, "00020 w 00470 00722" means mapper id no. 20 is responsible for computing the $P(w|z)$ for words 470 to 722. Other necessary information is read through the file system. These files include $n(d,w)$ for the given mapper, and $P(d|z)$, $P(w|z)$, $P(z)$ from the previous iteration.

Mapper computation is divided into two parts, and here we will use a mapper that is responsible for computing $P(d|z)$ for a given range as an example. Fig. 2 is a visualization of the most important component in the mapper, a 2-D array of topic×words, and each cell in the array represents $P(z|d_j,w_i)$ in equation 1. In the first part of computation, each cell is updated by computing the new $P(z|d_j,w_i)$ from corresponding values of the previous iteration. As each cell is computed in row-wise fashion, their values are also summed together to obtain the normalization factor for this row. Once every cell of a row is computed and the normalization factor obtained, the whole row is normalized, thus

completing the calculation for a row. This process is repeated for all the rows of the array to complete part one of the mapper computation.

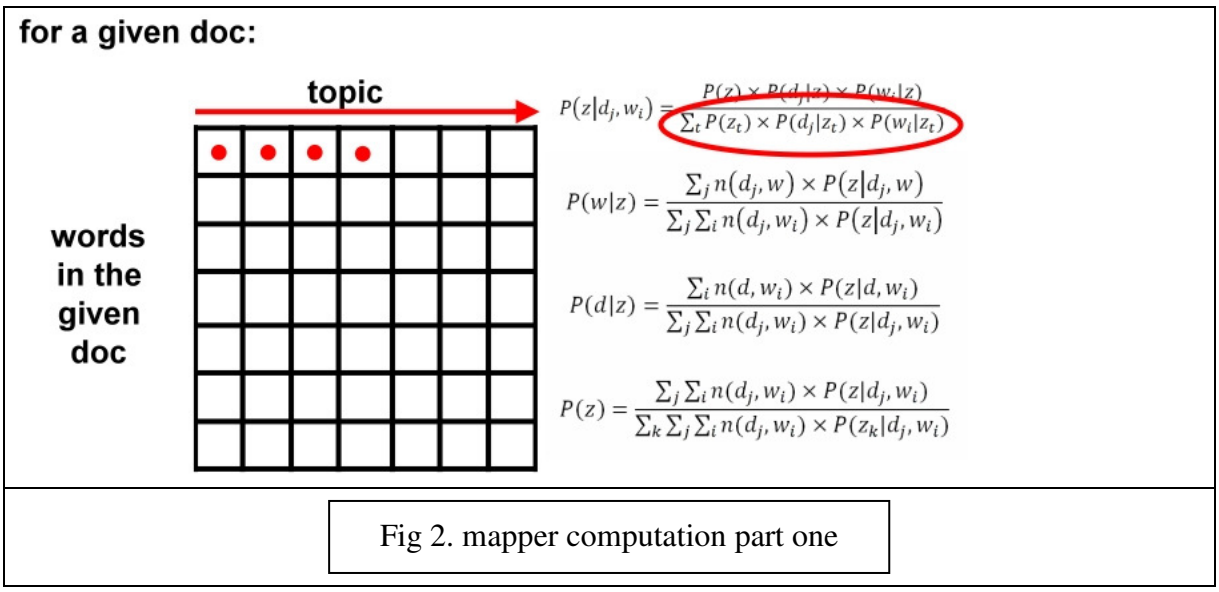


Fig 2. mapper computation part one

It is also worth noting that the x-axis of the 2-D array is words in the given document instead of total words. Since average words in a document (551 words) is a lot less than the number of words in the dictionary (20,000 in this project), this advance in representation will save enormous amount of memory and computation, which will be analyzed in later sections.

In part two of mapper computation, the matrix above is summed together column-wise to obtain the circled equation in Fig 3, and this value is then passed to the reducer. $P(w|z)$ is similarly obtained with the algorithm above.

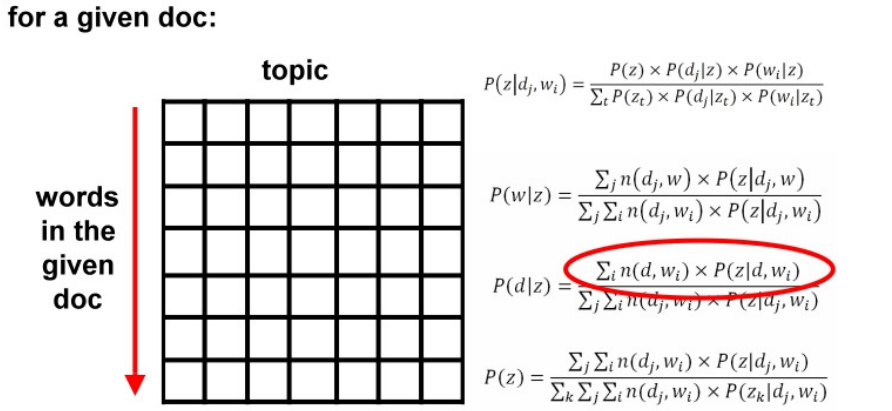


Fig 3. mapper computation part two

The communication between mapper and reducer is done with the following format:

Key: d/w

Value: # $P(x|z_1)$ $P(x|z_2)$ … where x = "d" or "w"

Ex:
Key: "d"
Value: 214 0.13 0.45 0.11 0.34 …
Ex:
Key: "w"

Fig 4. mapper reducer communication

Where the not yet normalized $P(d|z)$ and $P(w|z)$ is passed from the mapper to the reducer.

The responsibility of the reducer is to normalize the incoming $P(d|z)$ and $P(w|z)$. If we use the reducer responsible for $P(w|z)$ as an example, summing the *i*th column of the 2-D array shown in Fig. 5 would yield the normalization factor for that column of $P(d|z_i)$, in addition to the unnormalized value of $P(z_i)$. This process repeats until all the unnormalized $P(z_i)$ and normalized $P(d|z_i)$ is calculated and outputted to the file system for use by the next iteration. Finally, all the $P(z_i)$ is normalized and outputted to the file system as well. It is worth noting that there can be at a maximum of only two reducers, one responsible for $P(d|z)$ and one responsible for $P(w|z)$. More reducers are not possible due to the inability to concatenate to files in Hadoop 0.16.0.

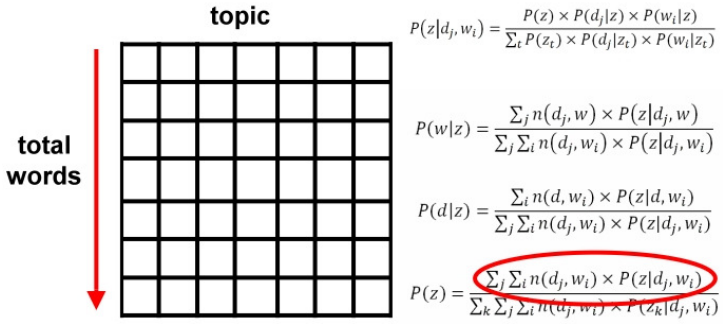


Fig 5. reducer computation

Design Process

While elated with the completion of the naïve version, which proved that PLSA is indeed solvable with the Map-Reduce model, we were totally appalled by the poor performance of the program, which was 83 times slower than a sequential version written with C. At the time, we believe that the problem was the inherit communication necessary between different rounds of Map-Reduce, which has to go through the file system, and the overhead accompanied in starting mapper and reducer tasks.

Now, let's partition the PLSA equations into smaller parts for the ease of explanation

$P(z)\times P(d_j|z)\times P(w_i|z)$ (numerator of (1)) (5)

$\sum_t P(z_t)\times P(d_j|z_t)\times P(w_i|z_t)$ (denominator of (1)) (6)

$\sum_j n(d_j,w)\times P(z|d_j,w)$ (numerator of (3)) (7)

$\sum_j \sum_i n(d_j,w_i)\times P(z|d_j,w_i)$ (denominator of (2), (3), numerator of (4)) (8)

$\sum_i n(d,w_i)\times P(z|d,w_i)$ (numerator of (2)) (9)

After scrutinizing the equations of the PLSA algorithm, we found a way to load data into a mapper such that we can compute equation 5, 6, 9 or equation 5, 6, 7 with a single mapper, and gracefully hand over $P(d|z)$ or $P(w|z)$ respectively to the reducer. However, if we choose to compute equation 5, 6, 9 and gracefully pass on $P(d|z)$, it is necessary to output a total of $O(d\times w\times z)$ pieces of data to piece together $P(w|z)$, and vice versa if we choose to calculate equation 5, 6, 7. To make matters worse, the process of piecing together $P(w|z)$ can only be done by a single reducer (otherwise more communication, thus more rounds of Map-Reduce is necessary).

Here we are faced with a dilemma, and there are two choices: one is to gracefully calculate either $P(w|z)$ or $P(d|z)$ and piece together the other one; the other is to make redundant calculations, so that both $P(w|z)$ and $P(d|z)$ can be passed gracefully to the

reducer, which would require only a total of $O((d+w)\times z)$ pieces of data. After lengthy consideration, we decided that if duplicating computation means we can avoid the inherently sequential work of piecing together $P(w|z)$ or $P(d|z)$, we should avoid the inherently sequential codes. In addition, this method is more intuitive, and with a relatively small test case of (d; w; z) = (2,000; 2,000; 100), we thought that having a reducer collect O(d*w*z) pieces of values seem like a huge bottleneck in terms of the whole project.

In hindsight, after running test cases in the range of (d; w; z) = (80,000; 20,000; 700), we realize that collecting $O(d\times w_{avg}\times z)$ pieces of values might not seem like a huge bottleneck after all, as mappers are now running in the range of 40 minutes per mapper. Doing away with the redundant computation would mean time and computation power to try to speed up the piecing together of $P(w|z)$ or $P(d|z)$, which might yield better performance.

Query Fold-In and Search Engine

With the result of the trained PLSA model, we can handle user queries in a search engine. We use PHP to build the search engine web page, and the following method is used to find the documents corresponding to the user's query.

1. We treat one query as one document. We decompose the query into many pairs of 2-character words, just as how we process documents.
2. We now have the word count of the query, denoted $n(q,w)$. We run EM algorithm for this query just like that we run EM for each document. The subtle difference is that $P(w|z)$ and $P(z)$ remain static during the process. $P(q|z)$ is the only thing that will change in M-step and will also result in the change in E-step.
3. We get what we need after step 2! We have $P(q|z)$ now and it means that we have gotten a vector indicating the proportion of each topic in this query. We also have $P(d|z)$ for all the documents in our training corpus. We take the inner product of $P(q|z)$ and $P(d|z)$ for each document as the relating score between the query and a document. Clearly, the document which produces the maximum score is the most related document with the query.

4. We sort the documents according to their score and only list the documents whose score is higher than some given threshold. Moreover, we choose topic z_1 which has the maximum $P(q|z_1)$ and list the words with the highest $P(w|z_1)$s as similar words and display the result to user.

Performance Analysis and Comparison

Performance Analysis for Advanced PLSA

Advanced PLSA Timing for 10000 documents, 20000 words, 1000 topics - One Iteration	
Processors	Average Time(s)
1	2387.0139404
2	1560.436385
4	1057.677541
8	842.8095478
16	731.2347188
32	650.9973154

Fig 6: Timing results with the Advanced PLSA

Above are the timing results for the same test data using different amount of processors. The test data has 10000 documents, 20000 words and 1000 topics. It is quite clear that the non-parallelizable part of the code takes up quite a lot of time, because the improvement amount decreases as we increase the number of processors used. We can make a slight estimate of the length of the non-parallelizable part of the code, which we assume is mostly I/O overhead. Let K denote the communication time, and L denote the parallelizable part of the code. Let p be the amount of processors used. An estimation formula is $K + \frac{L}{p}$. We derived that $K = 621$ and $L = 1766$ will give us the lowest absolute value difference compared to the experimental results. The non-parallelizable part of the code takes up about one-fourth of the whole sequential program! This is only a brief and not at all rigorous estimate, but we can get the idea that the I/O overhead of our program is still quite high.

Advanced Version Timing for Different Test Data - One Iteration				
Documents	Words	Topics	Computational Complexity	Time(s)
2000	20000	100	4E+09	58.624944
10000	20000	1000	2E+11	650.99732
82974	20000	250	4.149E+11	935.93066
82974	20000	700	1.162E+12	2559.9278

Fig 7: Timing Results for Different Test Data

Above are the timing results that we have run for each test data. For the PLSA model to be accurate, we ran all our test data for 50 iterations.

We would have liked to use more topics for the test data with 82974 documents, but due to memory constraints, we were not able to do so. Our advanced method requires that the whole $P(d|z)$ and $P(w|z)$ array be in memory, and Hadoop has an constraint that all mappers can only use 1 GB of memory. Therefore we cannot use as many topics as we liked.

Comparison Between Naïve and Advanced PLSA

To see the improvement of advanced PLSA over naïve PLSA, we timed both algorithms for input size d = 2000, w = 20000 and z = 100. The results are as follows:

Version	Machine Information	Time (min)
Naïve C with OpenMP	Grid02 8 cores	14
Naïve PLSA	Google cluster with 36 cores	74
Advanced PLSA	Google cluster with 32 cores	1

Fig 8: Performance Analysis of Advanced PLSA and Naïve PLSA

The followings are some possible factors contributing to this tremendous performance improvement. First is improvement of algorithm. Our advanced PLSA reduced the number of Map-Reduce pass to 1 instead of 4; that is, the advanced version eliminates many costs associated with running multiple Map-Reduce jobs. For instance, we now only need to synchronize once per iteration instead of four times, which means the amount of time wasted waiting for mappers/reducers completing their works. Additionally, the advanced version

avoids lots of local disk I/O and GFS access which were originally necessary to communicate between each Map-Reduce job. Advanced version keeps all the necessary information in memory, and is quickly and readily available for the next stage of computation.

Another important improvement is that the 2-D topic×words array now only includes words that appear in a given document instead of all words that appear in the dictionary. This improvement prevents wasting computation power for entries whose $n(d,w)=0$. Empirically, there are only 551 words on average in one document. While the naïve version still does the computation for all 20000 words, advanced version takes advantage of this "sparse" property of $n(d,w)$ and only computes those valid "w" entry. This means we have reduced the amount of computation by approximately 20,000/551 ~= 36 times.

A smaller improvement is changing the representation of $n(d,w)$. Since $n(d,w)$ is a sparse matrix, we changed the representation from a 2-D array to a pair-wise representation of (word label, $n(d,w)$). The reduction comes from the same fact mentioned above: a document contains relatively little words compared to the dictionary, and this change greatly reduces the amount of file I/O needed by the mapper to obtain the required $n(d,w)$ information.

Comparison with OpenMP and MPI

What if our project is done with OpenMP and/or MPI instead of Map-Reduce? OpenMP is indeed very suitable for computing PLSA, as all the information is stored on a common shared memory, minimizing communication costs between processors. Moreover, OpenMP is easy to parallelize, as all we have to do is write the sequential code and add the "pragma omp parallel for" clause in front of the "for" directive. In fact, we did write a sequential version using a naïve algorithm and parallelize it using OpenMP, and it performs a lot better than our naïve PLSA. However, despite its low coding complexity, the scalability of OpenMP is limited by the number of cores that is connected to a single shared memory. Hence, scalability is a critical drawback of OpenMP.

How about MPI? MPI is generally expected to have better performance than Map-Reduce because it is coded in C. Also, any processor can communicate with any other processor, with no limit on time and format, while in Map-Reduce processor can only communicate with others between the mapper and reducer, or via DFS. However, in MPI, any loss of passing message can result in deadlock or the failure of the whole computation, let alone sudden malfunction of machines. Map-Reduce, on the other hand, has a robust fault tolerance feature. The task can still be completed even if some machines in the cluster breaks down. Therefore, though MPI is faster and more flexible, it has high coding complexity and it is much more unreliable than Map-Reduce.

Difficulties

This section will primarily be focused on the difficulties we met when we tried to run our program on the Hadoop server. We met lots of problems when we implemented the naïve version of PLSA.

Blank output file written by the mapper to the filesystem

At first, we were not really familiar with the Map-Reduce concept; therefore we wrote our program using the MPI concept. Since speed was not our main concern, we used the file system as the means of communication. Therefore, our mappers will read files from the file system and, while not using any reducers, write files directly to the file system. However, for some unknown reason, some files written to the file system are blank. We spent quite some time debugging but could not find the reason. After discussing this phenomenon with Victor (Google Engineer), he said that it is odd for a mapper to be outputting files in the first place. Therefore we changed our code so that it fits more into the Map-Reduce concept: pass on the output to the reducer, and then let the reducer do the outputting for us.

Even with the same keys, the order of the values received at the reducer may be different from the sending order of the mapper.

We met another problem when we tried to send the data we computed from the mapper to the reducer. This problem occurred in the calculation of the $P(z|d,w)$ step. Each mapper will be given a topic number *n*, and that mapper will be responsible for calculating and outputting $P(z_n|d,w)$. In order to avoid the blank output file problem, we had to somehow send the data produced at the mapper to the reducer. Let *D* and *W* denote the number of documents and words in our test data respectively. The array calculated will be of size $D \times W$, which is quite big. Therefore, instead of sending the whole array once to the reducer, we decided to send data of length *W* *D* times to the reducer, thus cutting the memory requirement at the mapper from $D \times W$ to *W*. However, even though each mapper uses the topic number it is processing as its key, which will be unique, to send data to the reducer, the order of data received at the reducer is different from the order sent by the mapper. If the order is different, we would have to buffer all the $P(z|d,w)$ data in memory at the reducer

before writing it to disk. When we discovered this problem, we avoided the sending process by doing all our calculation in the reducer. All the mapper does then is send the topic it is responsible to the right reducer with a partitioner. With this method, we successfully avoided this problem and decreased the amount of traffic between the mapper and reducer greatly.

Task task_2008…0002_0 failed to report status for 601 seconds. Killing!

With the above two problems out of the way, our naïve version runs correctly on very small test data, but when we tried to run data with about 800 documents and 10000 words, all our tasks were killed by the above message. Our TA showed us what the script in /hadoop/hadoop-0.16.0/conf/hadoop-default.xml wrote:

```
<property>
<name>mapred.task.timeout</name>
<value>600000</value>
<description>The number of milliseconds before a task will be terminated if it neither reads an input,
writes an output, nor updates its status string.
</description>
</property>
```

Armed with this new knowledge, we updated the status string using *reporter.setStatus(any status string)* ever so often in our program and avoided the problem. And to our pleasant surprise, since the status string can be any string we like, it was something that we had been looking for ever since we started our project: a channel that the program can show us its status in real-time, like the *printf* in C. Our status showed to which document/word a mapper has processed, which was useful because at least we now know the program is running, ever so slowly in the case of the naïve version. Through this new tool, we can do some very simple profiling for our mapper and reducer. One important discovery, though quite obvious at hindsight, was that string concatenation in Java is slow, since it allocates a new string for every concatenation operation. We used string concatenation when we need to send an array of *double* from the mapper to the reducer, or when outputting data at the reducer. We did it by turning all *double*s into string representation separated by a space. In order to avoid this problem, we wrote a new function for string which manually manages the memory and the re-implemented string concatenation. We did try to write a new output value class, but we were not really successful and thus did not spend a lot of time on it.

Uploading the program takes a long time!

This is a very stupid problem, but it caused us a lot of frustration. Why does uploading a program take such a long time (3 minutes)? We discovered the reason when we tried to

create our own .jar file of the program. Why the test data folders we placed in the same directory as the code for convenience are in the .jar file too? After moving those data to another folder, uploading took about 10 seconds.

The Unkillable Job

Even now, we still do not know how to kill Hadoop jobs the "correct" way. Our TA told us to delete the .xml file of the job and the job should stop soon. However this method was useless on the Google Hadoop Cluster, and we resorted to removing the input files and directories of the mapper to stop the job. The job will throw an exception and halt when it cannot read the files it needs. Our *main* function that controls how many Map-Reduces we will run sequentially was written so that if one of the Map-Reduces fail, then the *main* function will exit too. It throws all exceptions thrown to it upwards to the JVM, and when the JVM catches an exception, the *main* function will halt. This is necessary because we do not know how to kill the *main* process on Hadoop, and if the *main* process continues to submit Map-Reduce jobs that are doomed to fail, we will waste a lot of resources.

"SCP Connection to hadoop server failed"

The above message is sometimes seen when we try to run our program in the Eclipse Map-Reduce Perspective. We were quite frustrated when we saw this message because the program successfully ran on the cluster a few minutes ago, and we did not change our code at all! After inquiring with Victor, the reason is that there is not enough space left in our home directory on the Google Cluster to Secure Copy our jar file. We learned from our TA later that there is 500GB in */tmp*, so a workaround was to manually make our job file, then upload it to */tmp* of the Google Cluster, and finally use *./hadoop jar jar-file* to execute our program. It was quite relieving when we could continue running our jobs again.

Above were the many problems that we faced when we implemented our naïve version. Even though most problems look extremely obvious, simple and sometimes stupid at hindsight, it is difficult to foresee these problems, and only the ones who have journeyed through this process will learn to avoid them in the future.

One of the purposes of writing this naïve version is to let us get familiar with the Hadoop environment, and I guess we really did become more comfortable with it. Thus when we implemented the advanced version, we didn't really have much trouble caused by being not familiar with the environment, and therefore we can focus ourselves on running our program using bigger and bigger test data. The only problem we had with the advanced version is described below.

The mapper skipped a line in mapper.txt!

We were checking our status on the job tracker webpage when we discovered a subtle bug.

When a job starts, the status field of each mapper will show which bytes of the input files it will process. The status of the first three mappers are shown below.

hdfs://10.1.130.99:9000/user/gamma/4/PLSA/for_mapper/mapper.txt:0+14

hdfs://10.1.130.99:9000/user/gamma/4/PLSA/for_mapper/mapper.txt:14+14

hdfs://10.1.130.99:9000/user/gamma/4/PLSA/for_mapper/mapper.txt:28+14

The contents of mapper.txt is shown below:

0 d 0 309

1 d 309 617

2 d 617 983

3 d 983 1280

4 d 1280 1566

5 d 1566 1857

The first line is 10 bytes long including the '\n'. The second and third lines are 12 bytes long, and the fourth line is 13 bytes long. All lines after the fourth line are 14 bytes long. After a few minutes, we examined the status of the mappers again, shown below.

Task	Complete	Status	Start Time
tip_200804221131_6155_m_000000	0.00%	[0, 309), processed to document 180	19-Jun-2008 01:10:52
tip_200804221131_6155_m_000001	0.00%	[617, 983), processed to document 857	19-Jun-2008 01:10:52
tip_200804221131_6155_m_000002	0.00%	[983, 1280), processed to document 1133	19-Jun-2008 01:10:52
tip_200804221131_6155_m_000003	0.00%	[1280, 1566), processed to document 1420	19-Jun-2008 01:10:52

For example, the status field of mapper zero means that it is responsible for processing document zero to document 308, and it has already processed to document 180. For some unknown reason, the range [309, 617) is not processed by any of the mappers. We are not really sure why does this happen, but we believe that this has something to do with the byte count of the first few lines are smaller than 14. Therefore, we avoided this problem by padding all the numbers in mapper.txt with zeros. For example, the first line of mapper.txt will be changed to:

"00000 d 00000 00309"

The problem is solved after this correction. Due to this problem, we have to run most of our data all over again.

Conclusion

We believe that PLSA, with its emphasis on returning "what the user intends" instead of "what the user inputs" is a very novel concept in querying, and as far as we know, almost no search engines provide similar querying properties. PLSA is extremely computationally expensive, so it might have been unrealistic in the ages of single core single thread computing. But with the maturing parallel computing environment, an already very powerful keyword search engine such as Google would be even more powerful if aided by a semantic search mechanism such as PLSA.

PLSA requires quite a lot of communication. Therefore it may not be very suitable for the Map-Reduce model. Our advanced version tried to minimize I/O overhead by using only one Map-Reduce per iteration. However, it suffers from memory problems. Therefore, developing a new method which is scalable and has minimal communication will be a great challenge.

Future Works

PLSA Implementation

The biggest problem we face with our advanced PLSA is that it uses too much memory, thus making it unscalable. Solving this problem would mean that a mapper cannot read the whole $P(d|z)$ and $P(w|z)$ into memory, which means more communications between each mapper-reducer nodes is necessary. However, the only means of communication in one Map-Reduce is between the mapper and the reducer. In our current implementation, this only window of communication is already used up by sending the not yet normalized $P(d|z)$ and $P(w|z)$ to the reducer. Therefore, this means that we need more than one Map-Reduce per PLSA iteration, which would mean increasing the already rather horrendous disk I/O overhead. Developing a more scalable algorithm that uses minimal disk I/O with

Map-Reduce will be of a great challenge.

In Query

Until now, the searching results we feedback to the user are the documents that are already in our database. If we want to break through this limitation on the data amount, we have to crawl the web for more documents and thus we can provide data that is on the fast-changing Internet world. Because our PLSA model has already been built up, whenever we found a new document on the Internet, we can treat this document as query and use our PLSA model to calculate its $P(q|z)$. We can save this $P(q|z)$ and the document's link in our database. Whenever a user query comes, it calculates the relating score not only with the documents saved in our machine but also the ones whose information ($P(q|z)$ and link) are available in our database. Thus the scope of searching can expand to a larger scale. However, this will also result in increasing search time and therefore the search process must also be parallelized.

What We Have Learned and Thoughts on Hadoop/Map-Reduce

By Michael Lu

It is quite fun to use Hadoop for the first time, since it's programming concept is different from the language that I learned before. It has some appealing features and mechanisms like fault tolerance in mapper. However, when the feeling fade out, I felt it inconvenient to use Hadoop. The most troublesome problem is that it is hard to debug under this mechanism. It takes some time to upload the source file and get it to start running; it takes some time to know that it does have bugs in the program; it also takes some time to kill the failing job. Maybe all the feelings come from the point that I haven't gotten used to Hadoop. Perhaps I will love it gradually in the future after I've overcome all the problems I met during this period.

Although it is not that convenient to use Hadoop; however, it is quite helpful to work things in parallel (except that it is still very hard to debug.) When the data amount becomes

larger and larger, to complete jobs in parallel style seems like the only solution. After all, it is impossible for a single machine (more specifically, a single core) to run some of the programs that require as many instructions as the number of sands on the earth. Parallel computing will be the trend in the future, and will also create a totally new approach toward problems.

By John Yu

In the process of working on this project, I learned a lot about the relatively new and unheard of Map-Reduce model. While getting used to Hadoop, and trying to fit the PLSA algorithm into the Map-Reduce model was a painstaking and sometimes very frustrating work, I was lucky to have a helpful TA and great partners that always worked together to resolve obstacles.

On the good side, I was very much attracted by the simplicity of coding once a task is translated into the Map-Reduce paradigm (and after getting through Hadoop's learning curve). We had relatively few bugs, coding was fast and straight forward in Java, and once these bugs are taken care of, the program always executes as expected. Also, ease of scaling up makes this model very attractive, as once this model is established for a given problem, an increase in computers and computing power almost always directly translates to increased speed, because it means more computers can share the burden of same amount of work. Finally, the fault tolerance feature really relieves us of concerns that a computer and its computation may crash, requiring us to rerun the whole program all over again.

However, we still met some annoying problems during this project, even though this might have more to do with the implementation of Map-Reduce rather than the model itself. First of, killing a job was practically impossible, which is a huge headache especially when we as novices might accidentally send up defective code. In the last few days of the project, someone accidentally sent up 1230 mappers which do not execute, and do not terminate for some reason despite efforts to kill it by their team. This rendered the Google cluster unusable for the days running up to the deadline, which is pretty serious. Then, there were not enough computers on the Google cluster to sufficiently demonstrate the advantage of the Map-Reduce model: lots of computers for a task. While 36 nodes is good enough to demonstrate the power and the prowess of the Map-Reduce model, I wonder if it is possible to open more computers during off-hours so that the power of Map-Reduce can be demonstrated in full? Finally, Google cluster also runs on Hadoop, which makes me wonder just how good is Google's implementation of the Map-Reduce model. If people can be attracted to the Map-Reduce model by using Hadoop (like me), then demonstrating how Google's implementation is say 10 times faster than Hadoop's implementation might attract prospective clients to run their programs on Google machines with Google implementation for a fee.

By Shih-Ping Chang

Using Hadoop to program is quite a challenge to me since most of our school works are in C language. It took me couple of weeks to review the JAVA language grammar, and even so, there were still a lot of problems when I really started to code. And the mechanism of uploading the source file first before running the program became a tedious job when debugging. However, if the one of the most annoying things of using Hadoop is it cost me a long time to code (and also wait for program to run), then one of the most exciting things of using Hadoop will be having an opportunity to get familiar with JAVA once again. In the end I felt that JAVA was not that hard (or tedious to use) as I thought before, and Hadoop itself is a convenient tool to parallelize a program, especially when the amount of data is very huge. It's also very exciting to find out how we can implement the Hadoop programming skills which we've been taught in class on a real problem in life through our final project, and see the huge improvement of the result. It's a very interesting experience.

By Kai-Yang Chiang

At the beginning, we had very little idea about how to do our project, including how to start and how to carry it out. The main reason is that we are not quite familiar with theMap-Reduce model, Hadoop and JAVA language. So we set our initial goal to be quite simple: just find a way to solve our problem with the Map-Reduce model by all means.

Thanks to the naïve algorithm, every team member had an opportunity to implement one part of the Map-Reduce routine. We did learn a lot about the model and some coding details under Hadoop. And fortunately, since we started our work earlier compared with other teams, we had more resource (grid and google cluster) and more time to improve our algorithm. That's the reason we can do much time improvement on our project, and experiment our program's performance by testing with different input size.

We did meet some difficulties during the project. For instance, we spent lots of time learning how to code and debug under the Hadoop environment. Debugging is quite hard in Map-Reduce. We cannot use System.Out.Console to verify our code. What's worse, DFS is not a friendly file system – we have to delete the output file and set the correct file path almost EVERY TIME, and re-upload the input file once we modify it.

Parallel programming is a fascinating idea and it is thriving. Even though it is not ubiquitous now, I believe that some day it will be a fundamental technique in computer science. In addition, Map-Reduce is an interesting model for parallelization. Despite the fact that the interface (under Hadoop) is a little inconvenient currently, it provides a new perspective to parallelize a program which is highly encapsulated and scalable. In conclusion,

I expect I can solve some other problem with parallelization after this class. After all, parallel programming seems to be a trend in the future!

By Shoou-I Yu

It was only a dream when I said that we can parse the data of the BBS board "Food". "If we were successful, we can group similar foods into groups", I said, "it is our ultimate and maybe a little far-fetched goal, because we have lots to do". I have done some parsing before, and understandd that parsing the data on the web is extremely difficult, because all sorts of odd problems occur. However, our team members all strived to make our final goal a reality. Our first meeting took place on 5/6/2008. At first, Michael Lu did the parsing and the other four implemented the naïve version, each being responsible for one Map-Reduce. Michael Lu successfully parsed the data, which consists of 82974 documents, in a week, which was a great boost to our morale, because now that we have our data ready, we can focus ourselves on implementing a PLSA that works and runs fast enough. The bugs of the naïve version caused us a lot of trouble, and we completed our naïve PLSA on 6/1. The naïve version took 74 minutes for each iteration using the 2000 document test data, which meant that if we tried to run the 5000 document test data, it would take about 740 minutes per iteration. Since we need to run at least 20 iterations to let the PLSA model be accurate enough, 740 minutes * 20 is just way too long. This seems to be the limit of our naïve version. I was quite disappointed then, because we were only able to use one-fortieth of the data we parsed.

In the meantime, we started to discuss and code our advanced version, which we completed on 6/3. I was extremely surprised and worried when I saw the timing results of our advanced version. It only took only one minute for each iteration using the 2000 document test data! There must be a bug somewhere. However, after comparing the output files with the previous versions, the results were correct. Therefore, we started to run bigger and bigger test data.

The mapper line skipping problem was fixed on 6/9. It was a pity that we could not run the 82974 document test data will enough topics, but compared to the naïve version, the advanced version is already a huge improvement! The final week before the presentation was devoted to building the webpage and preparing the slides for the presentation. To say the truth, unlike many of the projects I have done before, we completed this project in a quite leisurely pace.

Even though dealing with the details is always quite tedious and boring, in order to master a new environment, this process is inevitable. However, all the hard work paid off when seeing our "Fooooooood" webpage. It is just fantastic. The accuracy was a lot better than we have ever hoped. It is a pity though that our advanced version was not scalable,

therefore the database of the webpage consists of only 10000 documents. In conclusion, I must thank all our team members: “It was great working with you all, and thank you for making the “Fooooooood” dream a reality.”

Division of Labor

※ Pre-Processing: Michael, Shoou-I
※ Naïve PLSA: Kerry, Kai-Yang, Shoou-I, John
※ Advanced PLSA
  - Algorithm conception: Shoou-I
  - Mapper Implementation: Michael, Kai-Yang
  - Reducer Implementation : John
※ Running Map-Reduce experiments : Shoou-I, John
※ Website, query: Michael
  - Logo: Kerry
※ Report
  - Content: All team members
  - Editing: John
※ Oral Presentation: All team members

Reference

[1] Thomas Hofmann, Probabilistic Latent Semantic Indexing, *Proceedings of the Twenty-Second Annual International SIGIR Conference on Research and Development in Information Retrieval (SIGIR-99)*, 1999

[2] Wikipedia : Probabilistic Latent Semantic Analysis