Project Report: Needles in Gigastack

1. Index

2. Introduction

Author: Varun Sudhakar

2.1 Abstract

The aim of the project is to return the top ‘r’ ranked interesting distinct ngrams from a collection of formatted text data. Each ngram is associated with its term frequency which is the number of times the ngram occurs in the whole data set and its document frequency which is the total number of articles containing the ngram. The interesting terms are calculated on the basis of the TF*IDF measure. The program outputs the top ‘r’ interesting terms for range of ngram over M and N as input by the user.

2.2 Corpus

The corpus used is giga word corpus which is a collection of large text files (9.7 GB approx) consisting of a comprehensive archive of newswire text data. It is mainly used for development of programs dealing with natural language processing, information retrieval and language modeling. The data is very large and takes substantial computational power to process. The project utilizes the power of parallel computing to process data in short time.

*2.3 TF*IDF measure*

The TF*IDF measure is a statistical measure used to evaluate how important a term is in a collection. It is commonly used in information retrieval. The importance of a word increases with the number of times it occurs in a document but decreases with the number of times it appears over the entire corpus.

The TF*IDF is calculated using the formulae
TF(X) = the number of times the term X occurs in the corpus.
IDF(X) = log (D/DF(X)), where D is the total number of articles in the corpus, and
DF(X) is the total number of articles that contain one or more occurrences of the term X.
All logs must be to base 2.

|example: For corpus size of 162 MB from the giga word data and single ngrams, the term "CHINA" has TF = 125734 and IDF*TF = 233894.734375 while the term "PERCENT" has TF = 84994 and IDF*TF = 198745.109375. As we can see "CHINA" is more interesting than "PERCENT".

3. Related Work

Authors: Pavan Poluri, Varun Sudhakar, Siddharth Deokar

The paper by Mikio Yamamoto and Kenneth W. Church [1] talks about a new data structure called "suffix array" (introduced by Manber and Myers 1990). The paper has a detailed description of how a suffix array is created. Initially, a suffix array will have indices to all the alphabets in the corpus. Then the indices are sorted according to the starting alphabets they point to in the corpus. This sorting is done using quick sort. The reason of sorting the suffix array is to make it easy to calculate the term frequency and the document frequency.

[2]The paper by Alexandre and Dias describes a method to compute the positional ngram statistics based on masks, suffix arrays and multidimensional arrays. Most of the ngram models compute the continuous string frequencies which can lead to exponential rise in memory requirements as the corpus size increases. The key is to use masks to represent any combination of words in the corpus and storing just the starting words in a suffix array. Thus a Virtual Corpus is built representing the actual corpus with very less memory requirement. The occurrences of the positional ngrams are then counted using the suffix arrays.

[3]The paper by Chunyu Kit and Yorick Wilks describes the suffix array and sorting algorithms used. The bucket radix sort and q sort are described and their merits and demerits are evaluated. Based on this paper, we have decided to use a q sort to sort our suffix array since it seems to be well suited for sorting large suffix arrays.

The paper by Manber and Myers [4] says that suffix trees are very good data structures in terms of efficient string matching. A suffix tree for a text A of length N over alphabet Σ can be built in $O(N\log|\Sigma|)$ time and $O(N)$ space. Suffix trees have been tried and used successfully in many problems but are not efficient. This led to the introduction of suffix arrays. String searches using suffix arrays can be done in $O(P+\log N)$ where P is the length of the substring and N is the length of the total corpus. Basically, suffix array is an array of integers. The positions of all the characters in the corpus are stored in the suffix array. Once we store the positions we sort the suffix array positions basing on the strings each position points to in the corpus (in this case the string is starting from the character pointed by the position till the end of the corpus). The sorting is done using a complex variation of bucket - radix sort. After the suffix array is sorted, an array called longest common prefixes is created. Longest Common Prefix is an array that contains the length of the common prefix between two substrings. With the combination of the suffix arrays and the longest common prefix array, search for a substring can be answered in O (P + log N).

The paper by Puglisi, Smyth and Turpin [5] describes suffix arrays and lists their advantages over suffix trees. In particular suffix arrays are more memory efficient than suffix trees and are practical since any problem which can be solved by using suffix trees can also be solved using suffix arrays. The memory requirements of suffix arrays can further be reduced if they are compressed. Suffix arrays which store positions require more operations for comparisons since the file has to be read. Most papers which describe suffix arrays assume that the entire suffix array can be stored in the memory but this may not be true for a very large corpus. In such a case an inverted file structure should be used.

4. Architecture

Authors: Siddharth Deokar , Pavan Poluri, Varun Sudhakar

4.1 Stages

The project was divided into three stages Alpha, Beta and final. The Alpha stage requirements were to count the total number of words and articles in the corpus. The Beta stage involved eliminating the duplicate terms and finding the number of distinct terms along with their document and term frequency over the corpus. The number of distinct ngram terms had to be found over the user specified inputs of ‘m’ through ‘n’. The final stage involved calculating the TF*IDF measure and retrieving the top ‘r’ terms from the corpus.

4.1.1 Alpha

In the Alpha stage, we had to assume that the input could be present as any number of files of any sizes in the directory. In order to exploit parallelism our program scans the input directory and partitions files if they are too big or if the number of files is less than the number of processors. This ensures that we can distribute files to all the processors. Each processor reads its set of files and counts the number of words and articles in them. A reduce operation is then used to get the total number of words and articles in the entire corpus.

4.1.2 Beta

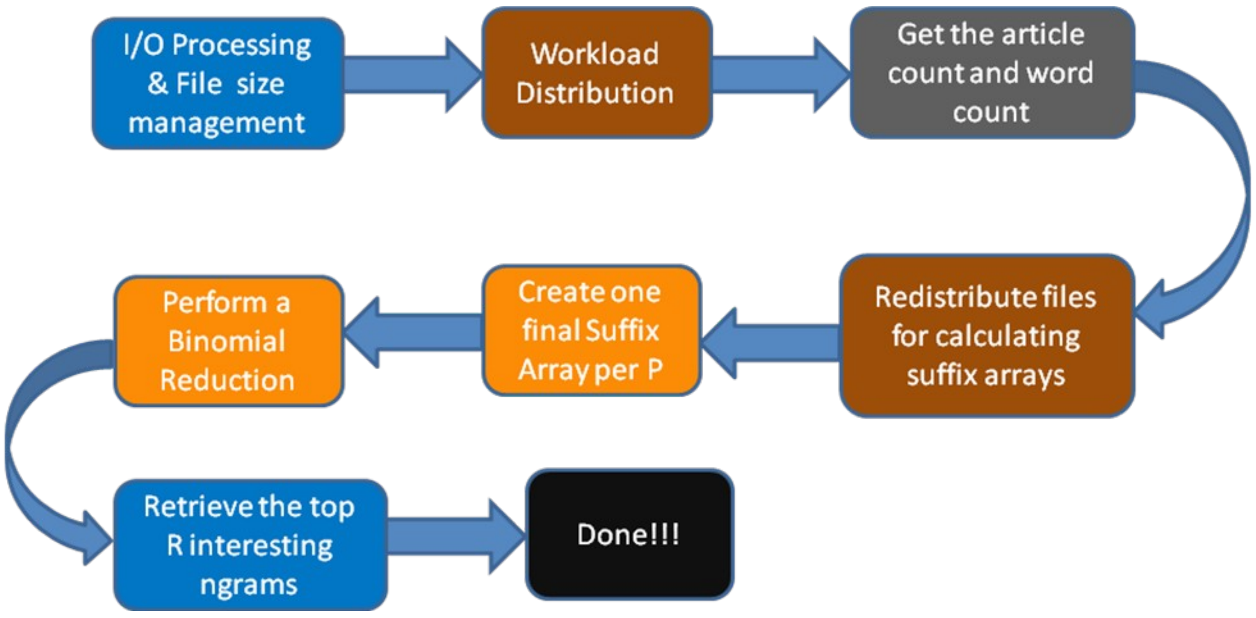
The input has already been structured in the Alpha stage and so the files can be distributed to the processors. Each processor creates a suffix array for every article in its set of files. The suffix array contains information about the starting position of the word, the file index and term and document frequency. Initially only information about the starting position and file index is collected. After this suffix array has been created, it is sorted using a quick sort and adjacent terms are compared to eliminate duplicate terms. The term and document frequencies are updated in this stage. The different suffix arrays are then written into files and these files are merged using a binomial tree reduction to form a single file containing all the suffix arrays calculated by the different processors. The adjacent terms are again checked to eliminate duplicate terms and term and document frequency are updated as needed. This final suffix array structure which contains the suffix arrays of distinct terms are written into a file. The program runs in a loop to calculate distinct terms from ‘m’ through ‘n’ and we get (n-m+1) files containing the suffix arrays of distinct terms.

4.1.3 Final

For the final stage, the top ‘R’ terms through m to n had to be retrieved. Our approach was modified to eliminate file writing. Instead the program stores the suffix arrays and merges them in memory. The suffix arrays are updated so that every element also has the calculated TF*IDF value. The sorting is now done based on the TF*IDF value. The merging is still done using a binomial reduction and using MPI pack and unpack commands to send the structure. Only the top ‘R’ terms for any suffix array are sent. This ensures that unnecessary data is not sent. Once merging of the entire suffix array is completed, the processor with id 0 picks out and displays the top ‘R’ terms.

4.2 Design

Authors: Siddharth Deokar, Pavan Poluri, Varun Sudhakar


I/O Processing & File size management
Workload Distribution
Get the article count and word count
Redistribute files for calculating suffix arrays
Create one final Suffix Array per P
Perform a Binomial Reduction
Retrieve the top R interesting ngrams
Done!!!

5 Suffix Arrays

Author: Pavan Poluri

The data structure we have used to solve this problem is *suffix arrays.* We are actually not using an array but a suffix structure for each word. We do not store the word in the suffix structure. Hence forth the term suffix array will be used to refer to a suffix structure and also arrays of suffix structures. We customized the suffix array to hold information according to our requirement. For detailed information regarding the suffix arrays please refer to Suffix Array Calculation [section 7.6].

6 Algorithm

Authors: Siddharth Deokar, Pavan Poluri, Varun Sudhakar

1. Process 0 reads the information about all the files in the given corpus (command line parameter) and stores the information in a structure.
2. Process 0 partitions any large files based on the average file size in the corpus or if the number of files are less than the number of processors. If there is a single large file in the corpus, then the file is partitioned into number of files equal to the number of processes.
3. After partitioning, the files' structure is broadcasted to all the processes by process 0.
4. The files are then distributed among the processors in an interleaved fashion. Each process identifies the file it has and counts the number of words and articles.
5. All the counts corresponding to different processes are then added up to give the total number of words and articles in the corpus.
6. In the next step the files are again allocated to different processes in an interleaved manner.

7. Each process takes the file assigned to it and calculates a suffix array for ngram = 1. Then it calculates the suffix array for ngram = M based on the suffix array for ngram = 1. The suffix array is then sorted using quick sort. Distinct terms are found after quick sort.

8. If the process has more than 1 file then the process merges all the suffix arrays corresponding to those files into a single suffix array. After merging the distinct terms are found again. By the end of this step, each process has a single suffix array with it.

9. Binomial tree is used for reduction of the suffix arrays. For example: If we have 8 processes p0, p1, p2, p3, p4, p5, p6 and p7 then in the first step p4, p5, p6 and p7 send their suffix arrays to p0, p1, p2 and p3. The processes which receive the suffix arrays then merge them into a single suffix array which is then modified to represent only the distinct terms. In the next step p2, p3 send their suffix arrays to p0 and p1 respectively and in the final step process p0 receives a suffix array from p1 and merges it into a single suffix array. At every level in the binomial tree we find the distinct terms after merging. At the end of this step, process 0 has one suffix array corresponding to ngram = M.

10. Process 0 then calculates the TF*IDF measure for all the terms in the suffix array. The terms in the suffix array are then sorted according to the TF*IDF measure. After sorting the top ‘r’ terms are retrieved from the suffix array. By the end of this step we have top ‘r’ interesting terms for ngram = M.

11. Repeat steps 6 to 10 for ngram = M + 1 through N. At the end of step 10, the top ‘r’ interesting terms for ngram = x and ngram = x-1 are merged together into a single suffix array of top ‘r’ terms. At the end of this step we have a single suffix array with the top ‘r’ interesting terms for ngram = M through N.

7 Implementation

7.1 I/O Processing

Author: Siddharth Deokar

Process 0 uses the directory reading API provided by C for UNIX for reading the corpus and getting the file information. This information is stored in a structure which is broadcasted to all the processes. Later, while merging and finding the distinct terms, the process doing that work reads the file for the word corresponding to the size of ngram for the comparison.

7.2 File Size Management

Author: Siddharth Deokar

7.2.1 Partitioning Files

Since the files in the corpus can be of any sizes, the files need to be made of comparable sizes before they can be assigned to the processors. If the corpus has a single huge file then for efficiency, the file needs to be partitioned so that all the processors can work on the file. So if there is a single file then the file is partitioned into number of files equal to the number of processors. While partitioning the file, we take advantage of all the processors that we have by distributing the partitioning work among the processors. If the corpus contains more than one file then we check the individual file sizes with the average file size. We partition a file only if the size of the file is greater than thrice the average size. If we partition for anything less than thrice the average size, then we spend more time in partitioning a file over reading and counting the intact file itself. Once the partitioning is done, we have an updated corpus with files with comparable sizes.

7.2.2 Communication

The work of partitioning files is shared by all the processors. For example: If we have abc.txt and 4 processors then processor 0 partitions abc.txt into abc_1.txt and abc_2.txt. Then again for the two files abc_1.txt and abc_2.txt, processor 0 partitions abc_1.txt into abc_1_1.txt and abc_1_2.txt while processor 1 partitions abc_2.txt into abc_2_1.txt and abc_2_2.txt so that we have four files(abc_1_1.txt, abc_1_2.txt, abc_2_1.txt, abc_2_2.txt) partitioned from abc.txt which is equal to the number of processors.

7.3 Workload Distribution

Author: Varun Sudhakar

The program is optimized to utilize all available processors and so the workload has to be evenly divided. An interleaved allocation scheme is used to divide the files among the processors. This works well since the input has already been structured in the partitioning of files stage and so no processor will be left idle. Once the files are distributed to all the processors, each processor works on processing the text in its set of files. This ensures that all processors are utilized efficiently.

7.4 Word and Article Calculation

Author: Pavan Poluri

Calculating the article count and word count is our alpha requirement. The article count is calculated by counting the number of '\n's in each document and adding it over all documents to get the total article count. To get the word count, first we just calculated the number of spaces per line and then added one to it to get the number of words for that line. But, we encountered some files which had multiple spaces between two words making our approach fail. So then we decided to keep track of spaces in order to avoid counting multiple spaces. At every instance of time we keep track of two characters. The first one is the current character and the second one is the previous character. So now we increment the word count whenever we the current character is an "alphabet" and the previous character is a "space". This way the word count is calculated.

7.5 Reduction

Author: Varun Sudhakar

The individual processors calculate the number of words and articles in their files and these subtotals are added by processor with id 0 using a MPI reduce to get the total count of all the words and articles in the entire corpus.

7.6 Suffix Array calculation

Author: Pavan Poluri

We have created a user defined structure for these suffix arrays to store information additional to just storing the positions. The other information that is stored is regarding term frequency, document frequency and the file index. In our project the basic unit is a word and not a character. So unlike the discussion about suffix arrays in [1] and [4] where positions of characters are stored, in this case we are storing the positions of words. So in short every word has a suffix array structure associated with it which has the information regarding the term's term frequency, document frequency and file index. The file index is unique to a file. The file index information is used to identify which file the word is from. Suppose a document has 100 words in it, distributed over 10 lines where each line has 10 words in it. We need to have 100 instances of the suffix array structure. The suffix array structure is a double dimensional array, where each row of it corresponds to the information of words in different lines and the column information corresponds to different words of the same line. Once the suffix arrays are created for a document we need to find if the document has any terms occurring more than once and if there are terms occurring more than once we need to update their term frequencies and document frequencies. For finding out the distinct terms, we need to sort the positions according to the words they point to. We had used quick sort to sort these positions. We tried understanding the sorting algorithm discussed in [4] but was not successful to the extent that we could implement it. So we resorted to quick sort which sorts in O(log N) time. For more details on sorting refer to Sorting Suffix Arrays [section 7.8].

7.7 Sorting Suffix Arrays

Author: Pavan Poluri

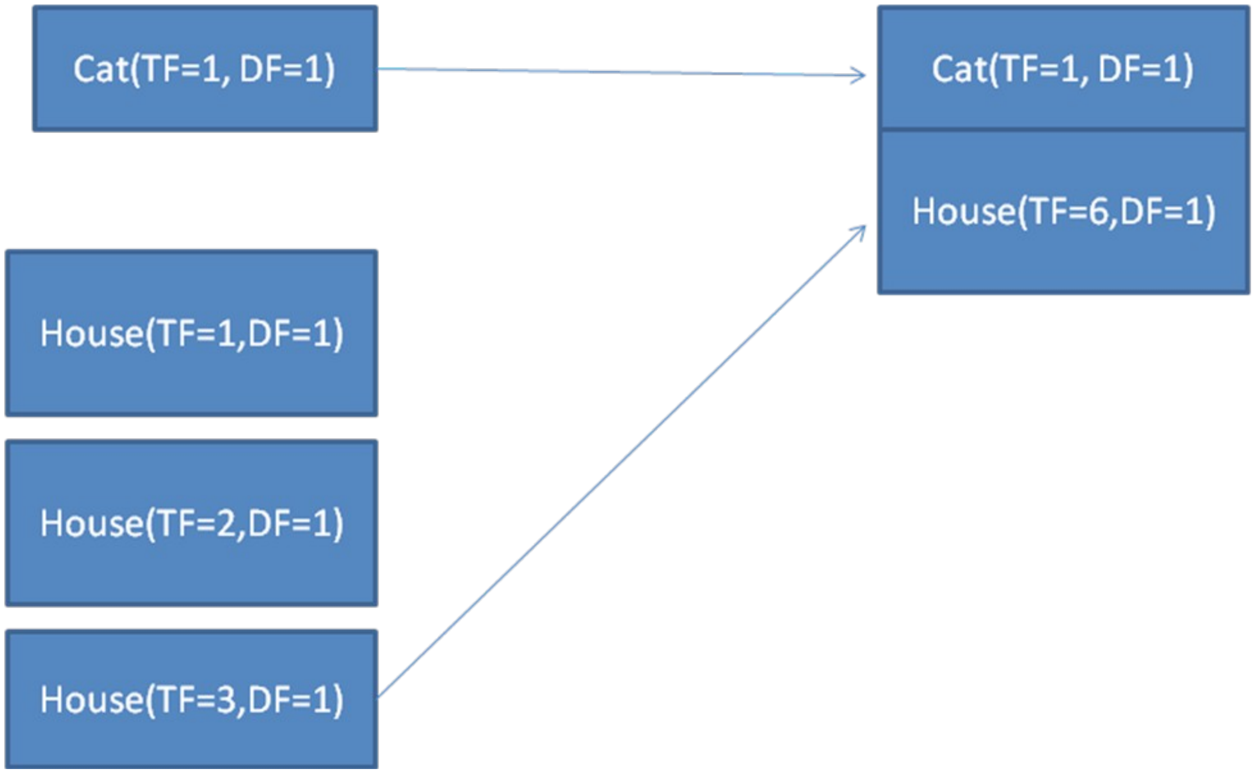
As mentioned in [section 7.6] we have used quick sort to sort our suffix arrays. Now we want to sort the suffix array. But all suffix array has is positions and we cannot tell which position is smaller than other without actually knowing the word the position is pointing to. So we need to do a file read on every particular position to know which word it is pointing to in order to sort them. We know that quick sort has its worst behavior when it is given a list that is already sorted or has lot of repetitions in it. Now if we add the overhead of file access also to quick sort, it takes more time. To counter this, we have used a function that actually stores the words in an array temporarily for sorting. We sort suffix arrays per line, find distinct terms in them, merge all the suffix arrays into one, sort the merged one and find the distinct terms and eliminate them and get a final suffix array per document. This approach is scalable from tokens of length one word to tokens of length multiple words also.

7.8 Finding Distinct Terms

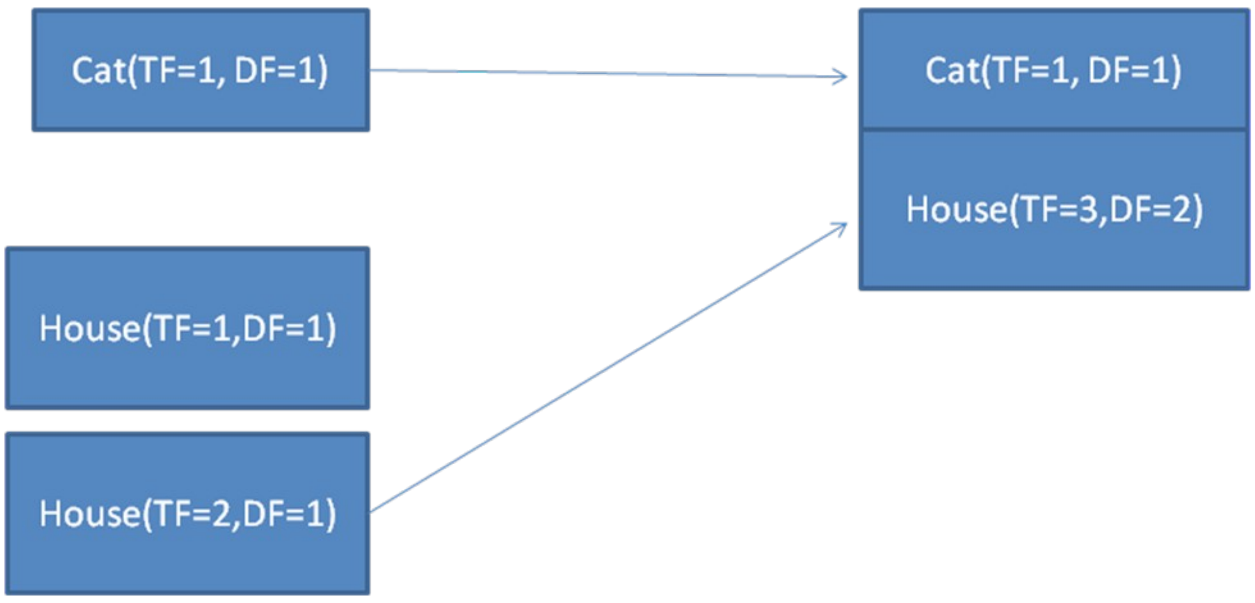
Author: Varun Sudhakar

Finding the distinct terms is done after sorting the suffix array and after merging the suffix arrays created by the different processors. In this step the duplicate terms are eliminated and the term and document frequencies are updated. This function works by comparing adjacent terms. If two or more adjacent terms are found to be duplicates, the term frequency and document frequencies are added into the last occurring duplicate and the previous entries are not stored. In this way the document and term frequencies are updated as and when needed. This function is called again after merging because duplicates found by different processors need to be eliminated in the same way.

Finding Distinct Terms in the same article



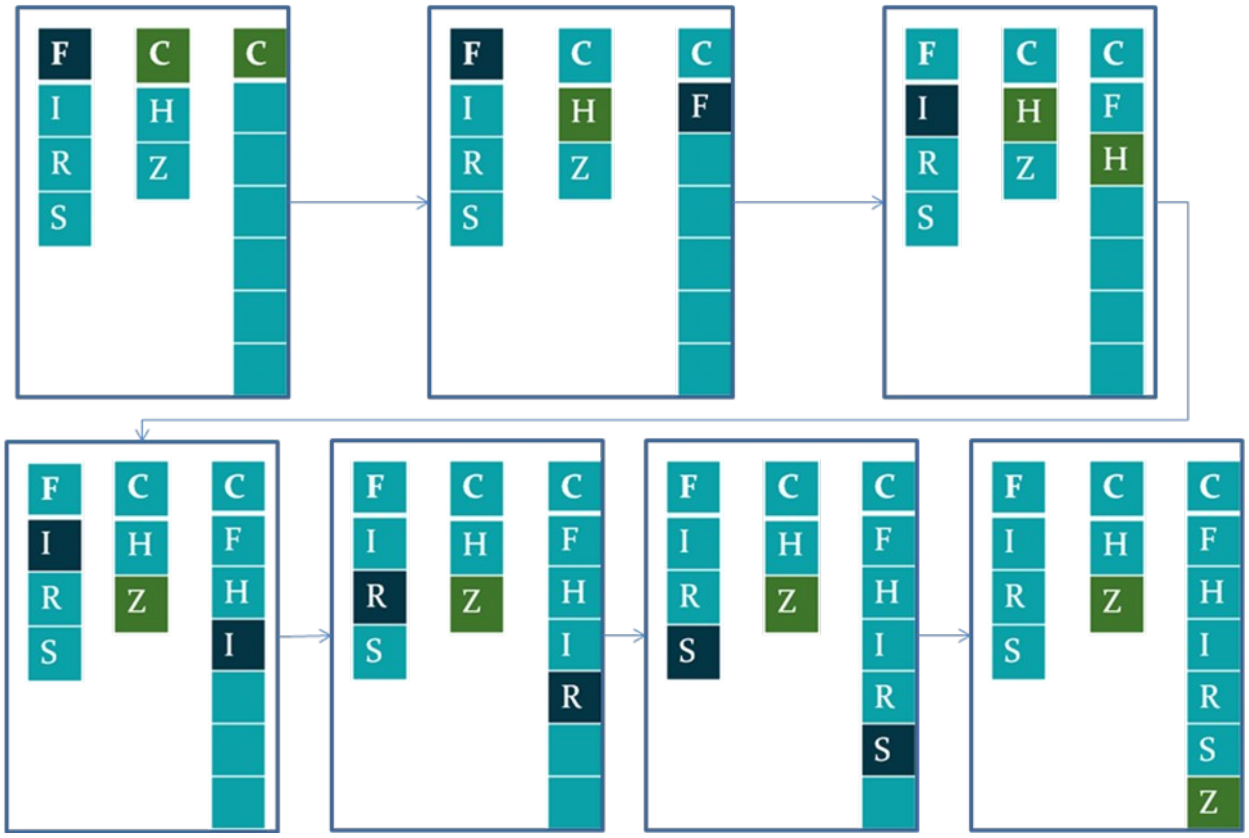
Finding distinct terms in different articles



7.9 Merging Suffix Arrays

Author: Siddharth Deokar

Merging is the process where two sorted suffix arrays are merged into a single suffix sorted suffix array. The suffix array stores the file index and the position of the ngram term. When merging, the process reads the position and file index, goes to the file corresponding to the file index and reads the ngram from the given position. The merge algorithm then compares the two ngrams and merges them in ascending order. The merged suffix array to be in sorted order must have the two suffix arrays to be merged in sorted order. The merge then ensures that the merged suffix array is in sorted order. Example: Consider two sorted arrays {F,I,R,S} and {C,H,Z} to merge. First we compare F and C. Since C < F, C is copied to the merge array {C}. Next we compare F and H. Since F < H, merge array = {C, F}. Similarly we follow till we have all the terms in the merged array.



7.10 Communication Strategies

During the initial phases of the project, the suffix arrays created by each process were stored in files on the hard disk. The I/O operations took considerable amount of time as compared to directly communicating the suffix array structures among the processes.

7.10.1 Reading and Writing Files

Author: Siddharth Deokar

Each processor gets its share of files to process upon. Every process creates a suffix array for every file that it has. After sorting, merging and finding distinct terms each process has a suffix array corresponding to all the files it has. The process then writes the suffix array to a file. For example: If a process, say process 2 is assigned with files, file1 and file2, then process 2 creates 2 suffix arrays 2_0 corresponding to file1 and 2_1 corresponding to file 2. Once these suffix arrays are created, they are sorted; distinct terms are found and then merged into a single suffix array "2" which corresponds to process id. Hence, before the binomial reduction can be performed, we have 1 suffix array corresponding to each process. For binomial reduction, we assume that the number of processes is always going to be a power of 2. So we divide the reduction in n = log2 p levels where p is the number of processors. For example: If we have p = 8 then n = 3. Then for n = 1, processors 0,1,2,3 merge with 4,5,6,7, for n = 2, processors 0, 1 merge with 2, 3 and for n = 3, processor 0 merges its suffix array with processor 1. At every step, the suffix array is written to a file. In the end we have a single file 0 corresponding to the suffix array for ngram = 1. This approach was later discarded in favor of communicating structures which is explained in the next section.

7.10.2 Communicating Structures

Author: Pavan Poluri

The reason for communication between the processors is to merge one processor's suffix array with the other and to do this in a binomial tree reduction fashion [section 7.12] so that at the end the processor with id 0 will have the suffix array for the whole data with duplicates eliminated where it can go ahead and start finding top R terms. Since reading and writing files took a lot of time we decided to communicate the structures (suffix arrays) between processors to save us the time from file overhead. Communicating structures is achieved using MPI functions like MPI_Pack_size( ), MPI_Pack( ), MPI_Unpack( ) along with MPI_Send( ) and MPI_Recv( ).

7.11 Finding Top R Interesting Terms

Author: Siddharth Deokar

7.11.1 Calculation and Storage

The top R interesting terms are found by the IDF*TF measure [Section 2.3]. A new suffix array structure is introduced which has the IDF*TF field which holds the value for the corresponding ngram term. Once we have the final suffix array for any ngram, process 0 calculates the IDF*TF measure for all the terms in that suffix array from the document frequency, term frequency and the number of articles in the corpus and stores the information in the new suffix array structure. By the end of this step we have a new suffix array with IDF*TF values.

7.11.2 Sorting

Process 0 uses quick sort on floating point values to get the new suffix array [Section 7.11.1] sorted on the IDF*TF measure. Once sorting is done, the top R terms are retrieved from the suffix array in a new suffix array structure which holds just the top R terms.

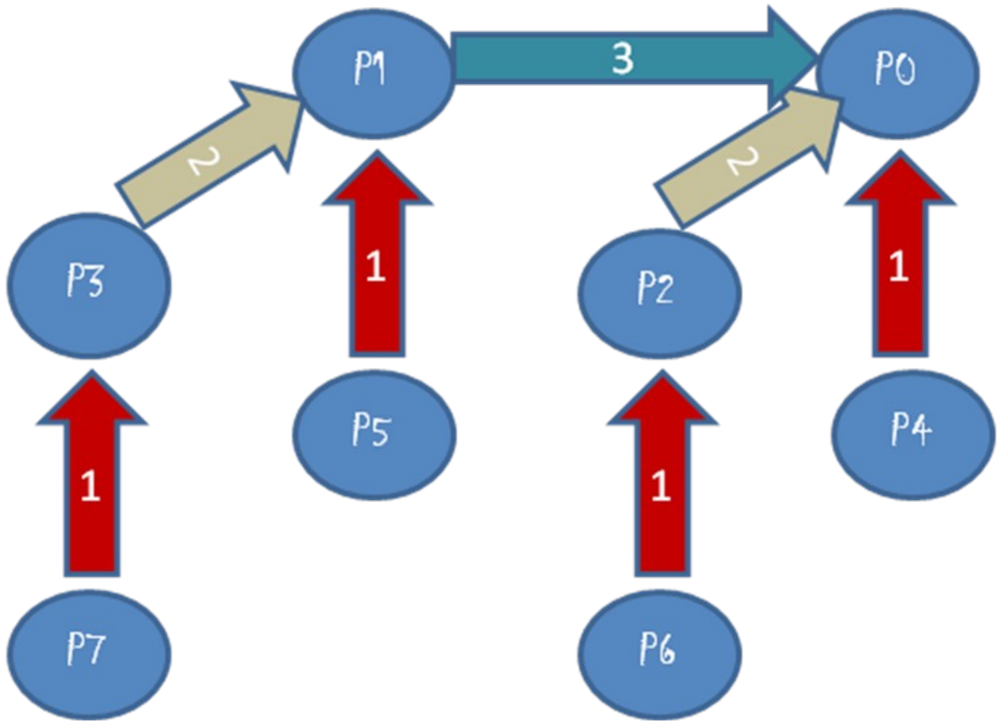
7.11.3 Merging

The top R terms need to be retrieved for ngram = M through N. Initially, for ngram = M, process 0 will have the top R interesting terms. Then the top R interesting terms are retrieved for ngram = M + 1 by process 0. Process 0 then merges these two suffix arrays and retrieves the top R terms from the combined suffix array. These top R terms represent the top R interesting terms for ngram = M and ngram = M + 1. As we go from ngram = M through ngram = N the top R terms retrieved for ngram = x are merged with the previous top R terms to get the new top R terms for ngram = M through x. This process continues as we reach ngram = N at the end of which we have the top R interesting terms for ngram = M through N.

7.12 Binomial Tree Reduction

Author: Pavan Poluri

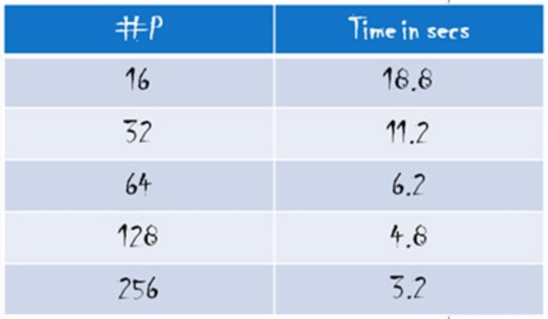
This is the kind of reduction we used when processors communicate. Our binomial tree reduction depends on the number of processors. The number of levels in our binomial tree is dependent on the number of processors (Number of levels = $\log_2(p)$ where p is the number of processors). For every iteration over each level, half the processors (whose ids are greater than half the number of *active* processors) will do a send operation and the other half do a receive operation.



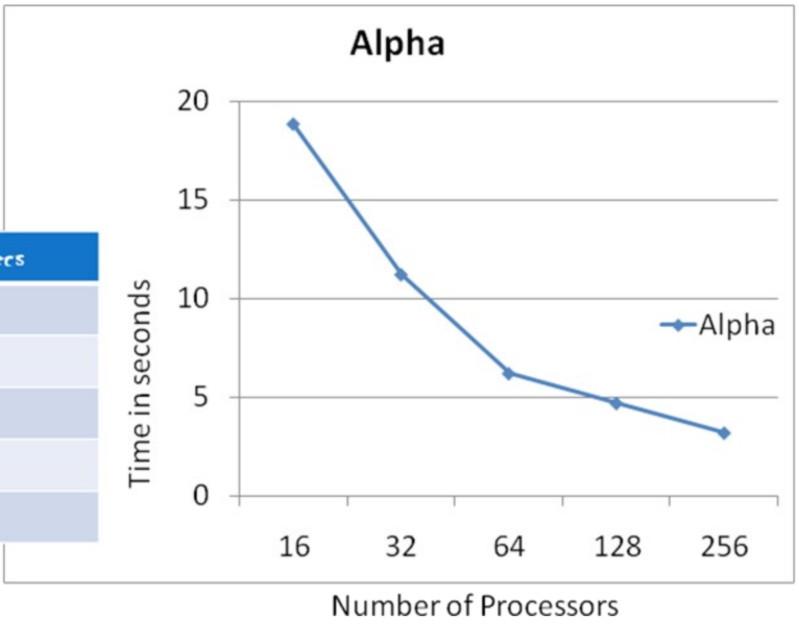
8 Results

Authors: Siddharth Deokar, Pavan Poluri, Varun Sudhakar

8.1 Alpha Results

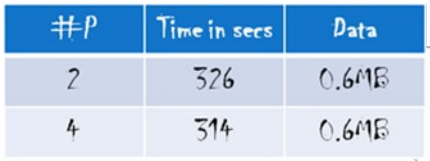


#P	Time in secs
16	18.8
32	11.2
64	6.2
128	4.8
256	3.2

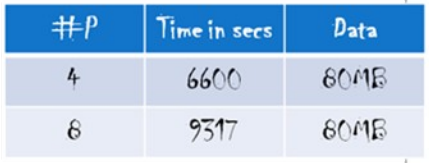


8.2 Final Results

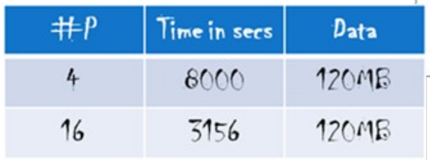
8.2.1 Analysis with Amdahl's Law



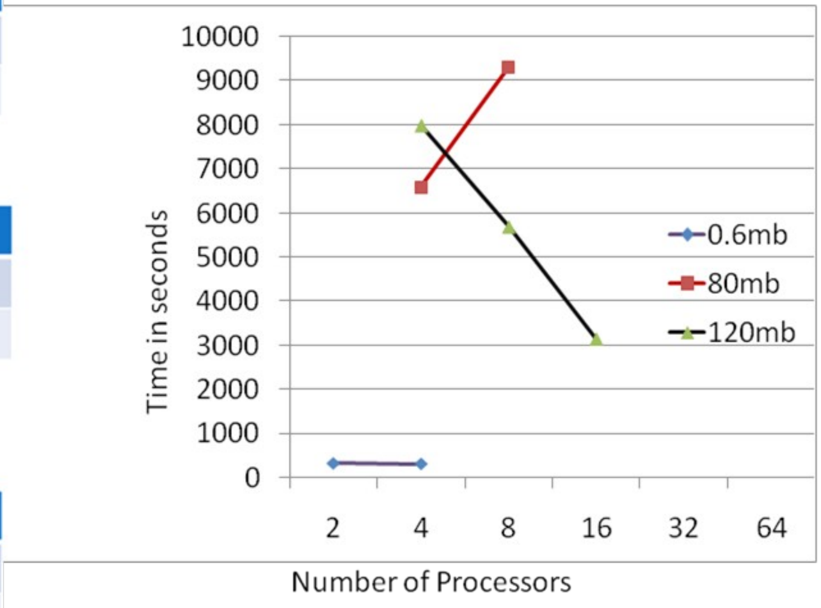
#P	Time in secs	Data
2	326	0.6MB
4	314	0.6MB



#P	Time in secs	Data
4	6600	80MB
8	9317	80MB



#P	Time in secs	Data
4	8000	120MB
16	3156	120MB

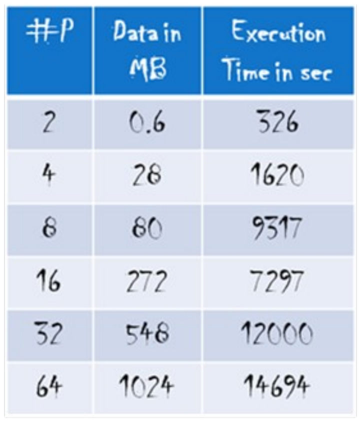


Using our results for data of size 120 MB

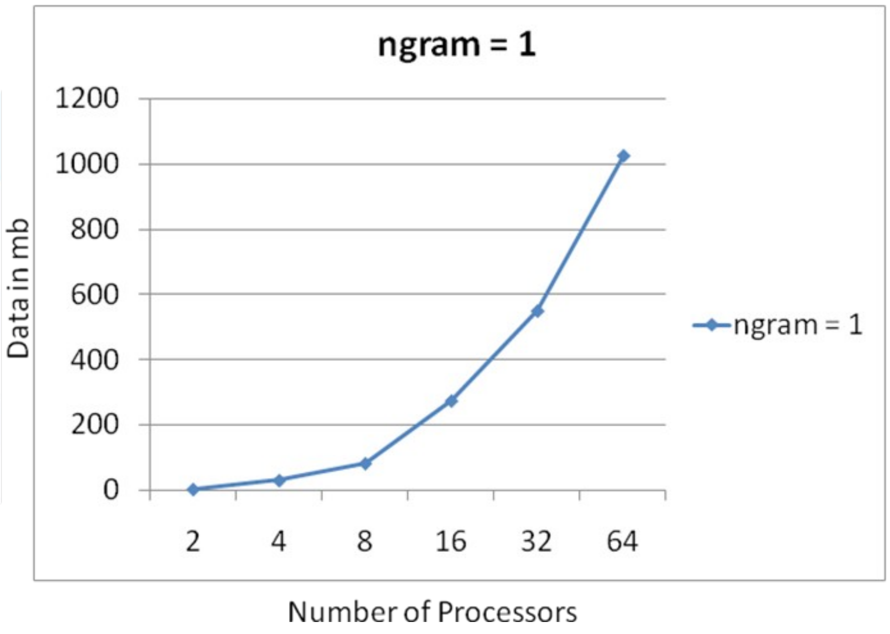
Speed up = 7680/3156=2.4

Considering the case where 4 processors as serial and 16 processors as parallel and using the formula for Amdahl's Law and substituting Ψ as 2.4 we get f = 0.22

8.2.2 Analysis with Gustafson-Barsis's Law



#P	Data in MB	Execution Time in sec
2	0.6	326
4	28	1620
8	80	9317
16	272	7297
32	548	12000
64	1024	14694



According to Gustafson's Law using s = 0.22, Ψ (scaled speed up) = 3.34

9 Open Issues

1. For small data size of corpus (approximately 8 MB), ngram works from 1 to 4. Needs to scale for large data.
2. Unpredictable results if number of files is greater than the number of processors. (Works well for small data).

10 Author Signature

Pavan Poluri
Siddharth Deokar
Varun Sudhakar

11 References

1. Mikio Yamamoto, Kenneth W. Church. Using Suffix Arrays to Compute Term Frequency and Document Frequency for All Substrings in a Corpus, Computational Linguistics, Vol. 27, Issue 1, March 2001, Pages 1 -30.

2. Alexandre Gil, Gaël Dias. Using Masks, Suffix Array based Data Structures and Multidimensional Arrays to Compute Positional Ngram Statistics from Corpora.

3. Chunyu Kit, Yorick Wilks. The Virtual Corpus Approach to deriving Ngram Statistics from Large Scale Corpora, Proceedings of the 1998 International Conference on Chinese Information Processing, Pages 223-229, November, Beijing.

4. Manber, Udi and Gene Myers. 1990. Suffix arrays: A new method for on-line string searches. In Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, pages 319-327. URL=http://glimpse.cs.arizona.edu/udi.html

5. Simon J. Puglisi, William F.Smyth, Andrew Turpin. Suffix Arrays: What Are They Good For?, ACM International Conference Proceeding Series,Vol. 170, 2006, Pages 17-18.