popular-terms Final Submission Report

Dinesh Bhirud, Prasad Kulkarni and Varada Kolhatkar

Department of Computer Science
University of Minnesota Duluth
{bhiru002,kulka052,kolha002}@d.umn.edu

December 14, 2008

1 Project URL

http://popular-terms.sourceforge.net/

2 Introduction

The aim of this project is to dig out a list of top R terms of varying lengths M through N that are especially interesting, from a large corpus of text. The goal is to do this as fast as possible with the use of multiple processors, but still retaining the accuracy. We are using the Gigaword corpus (a large archive of newspaper stories) as an input to this system. The basis of determination of what should be considered as interesting term is the $TF * IDF$ measure, where $TF(X)$ is the term frequency of the term X.

$$IDF(X) = \log(\frac{D}{DF(X)}),$$

where,

- IDF is Inverse Document Frequency,
- D is the total number of documents,
- $DF(X)$ is the document frequency of term X i.e. the number of documents in which the term X occurs at least once.

For example, suppose there are $D = 128$ articles in a corpus. Suppose the term we seek is *Washington DC* and that this occurs $(TF(X) = 10$ times in the corpus and it occurs in $(DF(X)) = 2$ different articles. Our calculation of
$TF * IDF$(*Washington DC*) is as follows:

$$\begin{aligned} TF * IDF(\textit{Washington DC}) &= 10 \times \log(\tfrac{128}{2}) \\ &= 10 \times \log(64) \\ &= 60 \end{aligned}$$

If you compare to the $TF * IDF$ of *is*, for example, you can see that *Washington DC* is more important. Suppose *is* occurs 110 times over all $(TF(X))$ and that it occurs in 120 different articles.

$$\begin{aligned} TF * IDF(the) &= 110 \times \log(\tfrac{120}{128}) \\ &= 110 \times \log(1.066) \end{aligned}$$

$= 10.142$ Note that since *is* is a very common word, it will occur in almost all articles.

Thus, the term *Washington DC* will be considered more interesting than *is*, as its $TF * IDF$ score is higher. As can be seen from the above example, higher is the number of documents that a term occurs in, less interesting it is considered. On the other hand, a higher term frequency also contributes to a term being "interesting". However,

since IDF is the logarithm of (D/DF(X)), the overall score for terms occurring in large number of documents is low.

As can be seen, the core of the processing done in this system is the counting of term frequency and document frequency of all the terms of varying lengths from M through N in the corpus. Since the Gigaword corpus is $\approx 10GB$, if we think of performing this on a serial computer, it will be very expensive both in terms of time and space. This is where parallelism can be exploited so that the interesting terms can be found out quickly on multiple processors and distributed memory. We do this using multiple processors performing the counting on different sections of the corpus simultaneously (Data level parallelism) and then combining the results to get the top interesting terms from the corpus.

Our approach to this problem, makes use of the data structures like Suffix array, LCP vectors, Class array and hash table. The work distribution is done using a manager-worker strategy wherein one processor is responsible for all I/O and distributes data to all other processors. The collection is done using a binomial tree communication pattern among processors so that none of the processor gets buried under a mountain of data.

3 Literature Review

Using Suffix Arrays to Computer Term Frequency and Document Frequency for All Substrings in a Corpus[YC01]

Summary by Dinesh Bhirud

This paper describes the data structures and techniques that can be used for counting term frequencies and document frequencies of n-grams in a very large corpus.

A string containing N words, would be having $N(N+1)/2$ substrings ie number of n-grams where n would vary from 1 to N. When the corpus is large, it would contain a large number of words, and as a result counting the tf of n-grams could become a very time or space intensive operation. This paper proposes the use of suffix arrays as a data structure that can be used to solve this problem, through partitioning the corpus into a smaller number of classes of suffixes ($2N-1$ at the most) which would reduce the space as well as time complexity.

Any string with N words, would be containing exactly N suffixes. A suffix array is nothing but an array that represents each one of these suffixes, in a sorted order. The representation could be done as a word index or character index into the string. The sorted order in the suffix array makes sure that all the suffixes that start with same words (prefixes) are at adjacent indices in the array, and this property is used to generate another data structure namely LCP (Longest Commom Prefix) vector. The LCP vector stores the length of the longest common prefix between the current and previous suffixes in the suffix array. We have implemented LCP array, which has a time and space complexity of $O(N)$.

In the next step, the corpus is divided into classes using the above 2 data structures. Each class is delimited by indices into the suffix array (i and j) and each class contains a set of substrings of these suffixes. These substrings are such that they are the prefix to each suffix in the class, but are prefix to no other suffix. Since, the suffixes in the suffix array are in a sorted order we can find out these classes with the help of LCP vector. All substrings in each class would share the same tf. Also, the classes would partition the corpus, and hence no substring would be left out.

Counting the tf of terms using above classes. If a term t belongs to a class $< i, j >$, then the term frequency of t would be $j - i + 1$. This is easy to figure out, since the suffix array is sorted and t belongs to class $< i, j >$, t can be a prefix to only elements i to j in the suffix array and no else. In the context of implementation for our project, our approach is to build these classes for each article in the corpus and then run through them each time we need to find out the number of n-grams of any length.

The term LBL (Longest Bounding LCP) used in the paper proves useful for this purpose. For a class $< i, j >$, the LBL is defined as $max(LCP[i], LCP[j+1])$. In simple words, it gives us the least number terms in any suffix in the class $< i, j >$.

Moreover, each class will have exactly one substring of any length. Hence each class can be checked if it contains a substring of length n, to find the tf for an n-gram.

Further in the paper, the authors have explained the results they obtained by using the above techniques on an English and a Japanese corpus. They have used MI (mutual information) and RIDF(Residual Inverse Document Frequency) measures to find the more interesting terms from the above mentioned corpora. MI is a measure based on a comparison of the actual tf of a n-gram as against what would be the frequency if these words in the n-gram were to combine by chance. The $RIDF$ measure compares the actual document frequency of an n-gram to what it would have been for a given tf if it was distributed randomly among the D documents. The observations show that the terms with a high $RIDF$ score give terms which are better for use as query to retrieve documents from the corpus.

Using Masks, Suffix Array-based Data Structures and Multidimensional Arrays to Compute Positional n-gram Statistics from Corpora[GD03]

Summary by Prasad Kulkarni

Gil and Dias discuss an implementation to compute positional n-gram statistics from corpora using masks, suffix array based data structures and multidimensional arrays. A mask is a representation of a text string in bits such that 1 represents a term and 0 represents a delimiter. The paper is divided in four parts: (1) the basic principles of positional n-grams and the mask representation to build the Virtual Corpus; (2) the suffix-array-based data structure which allows counting occurrences of positional n-grams; (3) use of a multidimensional array to efficiently compute Mutual Expectation; (4) results of the experiment over different size corpora.

Lexical relations such s collocations can be continuous or discontinuous sequences of words in a $(2.F + 1) - wordsize$ window context i.e. F words on either side of the pivot and the pivot itself. To compute all the positional n-grams, all the words as possible pivot words are taken into account. The authors have proposed a one-window context which shifts to the right throughout the corpus. To represent positional n-grams the authors propose reference representation rather than having an explicit structure for each n-gram. This representation is based on masks. A set of masks is used to identify all the valid sequences of words for a given window context. Each mask is a sequence of 1 and 0 where 1 stands for a word and 0 for a gap. An array of masks is built to identify each mask and to prepare the reference representation of positional n-grams. The suffix-array structure is sorted based on the lexicographic ordering for each mask in the array of masks.

The authors have then discussed a couple of equations to compute the Mutual Expectation using the sorted suffix-array-based data structure. For this purpose, the authors have highlighted use of a multidimensional array structure called Matrix. The techniques for accessing and performing operation on the Matrix have been discussed.

Towards the end of the paper the authors have discussed the experiments they conducted on large-scale text corpora to derive n-gram statistics for n-grams of various lengths. The experiment results have been listed of the program run on five different corpus of varying lengths.

The Virtual Corpus Approach of Deriving N-gram Statistics from Large Scale Corpora[KW98]

Summary by Varada Kolhatkar

The paper talks about using suffix array data structure for deriving n-gram statistics from a large-scale corpora. This paper is different from [YC01] in their implementation of calculating n-grams statistics. The implementation discussed in this paper assigns an integer code to each token in the corpus for fast processing. For finding n-gram statistics, a virtual corpus with pointers pointing to each word or character is created. The suffixes pointed by the pointers are then sorted. This is very much similar to the suffix arrays created in [YC01]. For sorting, Bucket radix sort is used which has the time complexity of ***O(***$N \log_n N$***)***. For counting the occurrences of a n-gram having length n is then counting the number of adjacent suffixes with the same prefix of length n. It seems like they didn't use equivalence classes as described in [YC01]. Then the paper presents some results of the experiments they conducted on PTB-II WSJ tag corpus and Brown corpus. They observed that for a large vocabulary size, quick sort is better than bucket radix sort and for a small vocabulary size, bucket sort works well.

We found this paper relevant because we thought this approach can scale well for a large amount of data as the number of distinct words(types) in any corpora will be significantly less compared to the total number of words and hence we found storing integer codes instead of text strings a neat idea. It is like creating a sorted list(to make search efficient) of unique words from the corpus and using their indices as integer codes. Since we are working in a multiprocessor environment, using such kind of unified basis for finding n-grams across and within processors might be helpful for our project. However, for a very large corpus such as the Gigaword corpus, assigning unique codes to words is not a trivial problem. Whenever we encounter a new word, in order to add it, every time we'll have to search for it in the sorted list. This search time is definitely an overhead.

Statistical Behavior of Fast Hashing of Variable-Length Text Strings[Sav90]

Summary by Varada Kolhatkar

Defining a good hash function is a challenging problem. It is best if this function is a one-to-one which requires least number of computations. However, if it is many-to-one, we should design it in such a way that there are not many $f(h) = f(h')$ such that $h = h'$. This paper discusses a number of hash functions for variable-length text strings. The Pearson's method described in the paper maps a variable-length text string to an integer. Moreover, the method takes into account the possibility of addition and deletion of the keys i.e. the key set is not static.

In the beginning, the Pearson's solution for the problem is discussed. In this solution an integer code is assigned to each character in the string. Suppose there are 256 unique characters in the corpus. The methods assigns $`a' = 1, `b' = 2$ etc for each character from 0 through 255. Then an auxiliary table T containing randomly generated integers from 0 through 255 is created. The integers in this table are generated using the shuffling algorithm. The proposed hashing algorithm uses bit XOR operation and returns a small integer, an element of the array T corresponding to the character string. This algorithm separates words like *honey* and *money*. However, it can't separate *ab* and *ba*. We can vary this table size when the corpora gets bigger.

Then the author mentions some of the collision handling schemes. He also discusses if the outcomes in the auxiliary table T are equally probable. Using the table size and the number of words in the dictionary, we can calculate the approximate number of collisions. Then there is a description of 8 different hashing algorithms for variable-length strings, among which HashSolution6, HashSolution7 and HashSolution8 can separate strings like *ab* and *ba*, which are called anagrams.

In our approach, before computing n-gram statistics, we need to build a dictionary from the whole corpus. Since we are dealing with the huge corpus, we thought it might be a good idea to build the dictionary, that is a list of types using such kind of hashes. The Pearson's method seems somewhat reasonable for our problem as it uses bit XOR operations which will be fast. In addition, it is not very likely that there are many strings like *'ab'* and *'ba'* in the corpus. However, there will be time and space overhead for the auxiliary table T.

A statistical interpretation of term specificity and its application in retrieval [Jon72]
Summary by Dinesh Bhirud

This paper presents the concepts of exhaustivity of document description and specificity of index terms, along with their implications and usage in retrieval systems. In retrieval systems that retrieve relevant/correlating text from collections, document descriptor for each document (index) and the index terms (searchable terms) are the key components. Exhaustivity of a document descriptor is the extent to which it wholly represents the topics covered in the document, through the index terms. The specificity of an index term is the level up to which it can distinguish a document from others. If the exhaustivity of a document descriptor is increased through the use of more index terms, the chance of the document matching a search request also increases.

Specificity of index terms is a property that depends more on meanings of the terms rather than statistics like their occurrence frequency, and hence more care should be taken while deciding on the index terms. The paper mentions two points that should be considered while deciding on the index vocabulary. One of them is the probable effect on document descriptors because eventually document descriptors are composed of index terms and this in turn affects the retrieval. Secondly, we should also consider the effect of index terms after they are used. This is because for each index terms there might be documents in the collection that match perfectly with it, some that do not match at all but still have an occurrence of that term and some that are on the fringe ie may or may not be matching. It is not an easy task to estimate the number or proportion of each of these categories of documents.

The aspects described above, tie the two concepts of exhaustivity of document descriptors and specificity of index terms together. The paper gives a simplistic definition of these two terms as ”the exhaustivity of a document description is the number of terms it contains, and the specificity of a term is the number of documents to which it pertains”. Clearly from this definition, exhaustivity of document descriptors have a direct impact on term specificities. If the descriptors are longer ie they contain more terms, the terms will be occurring in more document descriptors themselves. And terms that would be occurring in more document descriptors, would have a lower specificity value as they would be less capable of discriminating amongst documents.

The paper has also explained the effect of these two concepts on retrieval in detail. An observation shows that most of the request came for index terms that occurred more highly in document descriptors ie index terms with high document frequency. The paper proposes solutions like expanding the request by adding related terms to it and to weight terms based on their document frequencies.

We have chosen to read and study this paper, because we think that the concepts explained in this paper lead us to a better understanding of the tf*idf measure that we use. We think that the specificity property is something that can be directly related to the IDF of a term, and played a significant role in the study of stop words and decision of implementing pruning.

4 Algorithm/Approach Description

At the end of the alpha stage, we were successfully able to traverse the entire corpus using the manager-worker distribution and get the total number words from the corpus. Then the challenge for the beta stage was being able to traverse the entire GIGAWORD corpus and find out the counts of unique n-grams. For counting and identifying the unique n-grams, we used the Suffix Arrays method as proposed by [YC01]. The advantage of this method is that it identifies the n-grams within articles in $O(N)$ time complexity as against $O(N^2)$, which would be needed for a simplistic fetch-and-compare type of an approach. We were able to achieve the goal of the beta stage for $\approx 1.2GB$ of data. The goal of our final stage was to find out top T interesting terms using the counts obtained in the beta stage and the $TF \times IDF$ measure. The main challenge for us was to scale our solution from 1.2 GB data up to 10 GB data.

4.1 High level description of the algorithm

Processor 0 is designated to be the manager and the rest of the processors are the workers, who are responsible for the actual processing(identifying n-grams) of the data that is allocated to it. Each worker will do the following steps for each article that is allocated to it.

1. Find the suffix array and sort it.
2. Calculate the LCP vector from the suffix array built in step 1.
3. From the suffix array in step 1 and LCP vector in step 2, identify the classes.
4. In each class for each article, the worker will do the following tasks.
   (a) Identify if the class holds a n-gram of the required length.
   (b) If yes, then retrieve the text for this n-gram from the article.
   (c) Identify the dictionary id for each term in the n-gram found.
   (d) Check if all terms in the n-gram are in the stoplist
   (e) If yes, skip this n-gram and read the next n-gram (go to step a)
   (f) Check if the current n-gram is already inserted in the hash table of all unique n-grams for that worker. If yes, update the count of occurrence for that n-gram. If no, add the n-gram at appropriate location in the hash table.
5. After steps 1,2 and 3 is done by each worker, for all the articles that are assigned to it, each worker processor would have a hash table of all the unique n-grams that it has encountered.
6. Since n-grams with very low term frequency are not going to score high, we eliminate these n-grams here and free the memory associated with these n-grams.

7. The hash tables are then reduced (so that all the unique n-grams across the processors are found along with their counts), using a communication among processors that follow a binomial tree pattern.

Here, note that we use a word dictionary which is pre-built for this particular corpus. The dictionary is a big advantage to us for the following reasons:

1. Identifying unique n-gram counts across articles (within processors)
   The suffix classes and class arrays help us to find out unique n-grams within an article in linear time. However, to identify unique n-grams across articles, a simplistic fetch-and-compare algorithm would still be an exponential time complexity process. Hence, a dictionary here helps us in having a common and less memory consuming representation of the n-grams in articles. Also, having a dictionary means that the string comparisons are now reduced to integer comparisons.

2. Identifying unique n-gram counts across processors.
   If we can make sure, all the processors have the same dictionary, then reducing the unique n-grams across the processors would also become an easier task. In implementation terms, the task of finding unique n-grams is achieved using communication on the binomial tree pattern and performing the same process as that for identifying unique n-grams across articles. Hence, for the above 2 reasons, and to make sure that all the processors refer to the same word dictionary, we currently have a static dictionary to the program that acts as an input to it.

4.2 Detailed description of the modules

1. **Worker-Manager work distribution**
   by Prasad Kulkarni
   The manager (process 0) is responsible for reading the entire corpus and distributing it among the remaining processes (worker process). Given a path to the corpus the manager opens the directory and then opens and reads each file in that directory one after another. The manager reads the file line wise and sends the lines to the worker processes in a cyclic fashion. This ensures that all the worker processes receives some part of corpus to process. The manager and the worker processes exchange handshake signals for synchronization purpose. Before the manager sends a line to a worker process it waits for a ready message from that worker process. Similarly, once a worker is done processing on the line, it sends a ready signal to the manager to receive more data. Once the file I/O is complete the manager sends terminate signal to all worker processes to come out of the data receiving loop.

2. **Dictionary**
   by Varada Kolhatkar
   We observed that the total number of words in the Gigaword corpus according to n-gram Statistics Package (NSP) (http://www.d.umn.edu/ tpederse/nsp.html) package is 1831575402. But, the number of unique words(types) is just 1709628.

That is we have 0.09% types of the total number of words. This motivated us to build a dictionary for the entire corpus. We needed unique and sorted list of all words in the corpus to make our dictionary search faster so that we can search the dictionary in maximum $log_2(1709628) \approx 21$ steps. First, we tried to build the dictionary using binary search tree. However, it didn't scale for the huge 10GB data we have. We thought of using height balanced tree as well. But as per our analysis, it adds a lot of overhead for each insertion. We also tried a lot on working out a good hash function from variable length text-strings to integers. But all these attempts failed and we were not successful building the dictionary dynamically in our program at this stage. Currently the dictionary we use is static. We create a sorted list of unique words from the corpus using NSP and the system sort command.

The main advantage of the dictionary is the space efficiency. Moreover, if all processors have a single dictionary, it is easy to collect n-grams within processors and across processors. While collecting n-grams within and across processors, we use the indices of the dictionary instead of the actual words. This saves some space and allows us to compare integers rather than character strings. The hash function described later is also based on the dictionary indices.

3. **Suffix Array and LCP Vector creation for each article**
   by Varada Kolhatkar

   We implement suffix array and LCP vector as described by Yamamoto and Church [YC01] to compute n-gram statistics. Each worker gets a text line(which we call an article) from the manager for processing. We define a token as a nonempty sequence of characters not containing a space character. We define a sentence as a non empty sequence of tokens not containing the end of line character. For example, *"a double pointer is a pointer to a pointer"* is a sentence containing 8 different tokens namely 'a', 'double', 'pointer', 'is', 'a', 'pointer', 'to', 'a', 'pointer'. For n-grams statistics, we sort all suffixes in the input sentence using c qsort function and create a suffix array for each sentence/article. An element in the suffix array represents the starting index of the suffix. For example, continuing with the same sentence *"a double pointer is a pointer to a pointer"*. The sorted suffix array will be built as below

   s[0] = 0
   s[1] = 33
   s[2] = 20
   s[3] = 2
   s[4] = 17
   s[5] = 35
   s[6] = 9
   s[7] = 22
   s[8] = 30

which in fact represents

s[0] = a double pointer is a pointer to a pointer
s[1] = a pointer
s[2] = a pointer to a pointer
s[3] = double pointer is a pointer to a pointer
s[4] = is a pointer to a pointer
s[5] = pointer
s[6] = pointer is a pointer to a pointer
s[7] = pointer to a pointer
s[8] = to a pointer

Essentially, the size of the suffix array is the number of tokens in the given sentence. Note that the new line characters are eliminated as we process text line by line and build a suffix array for each line in the text. We allow tabs in a token. After building the suffix array we build LCP(Longest Common Prefixes) vector. LCP vector takes suffix array as input and an element in the LCP vector represents the number of common tokens with the previous element in the suffix array. The LCP vector is of size n+1, where n is the number of tokens in a sentence. The first and the last element of the LCP vector is 0. For example, for the above sentence the LCP vector is

LCP[0] = 0
LCP[1] = 1
LCP[2] = 2
LCP[3] = 0
LCP[4] = 0
LCP[5] = 0
LCP[6] = 1
LCP[7] = 1
LCP[8] = 0
LCP[9] = 0

We can see that $LCP[2]$ is 2, which means that there are 2 tokens common between $s[1]$ and $s[2]$. We define a n-gram as a sequence of one or more tokens. The frequency of a n-gram is how many times that particular n-gram occurs in the corpus. Further, the LCP vector is used to create equivalence classes of n-grams and their associated frequencies.

4. **Identifying unique n-grams within articles using data structures**
by Dinesh Bhirud
This stage kicks in after suffix array and LCP vector is generated from module 3, and the dictionary is available from module 2. The algorithm exploits the fact

that until the values stored in the LCP vector follow an increasing sequence, ie $LCP[m+2] > LCP[m+1] > LCP[m]$, the length of the common prefix of these suffixes is increasing. The left edge of the class is maintained through the use of a stack. For each index i in the suffix array, a trivial class $< i, i >$ would always exist. The algorithm proceeds by scanning through the LCP vector and pushing onto the stack the values of i and k (class representative, which is an index into the LCP vector that represents the length of the maximum length of substring in the class). Whenever, a value of LCP lesser than the previously pushed on stack value is found, means that the longest common prefix is found and corresponding class intervals are obtained by popping values from the stack. After all the classes are identified, we loop through all the classes for each article. For each class, it is identified if it holds an n-gram of the desired length. If yes, then the dictionary indices of all the words in this n-gram are found and this n-gram is passed to the next stage to be put in the hash table.
Continuing with the same example used in section 3, the following classes would be generated for the sentence *"a double pointer is a pointer to a pointer"*

$< 0, 0 >$
$< 1, 1 >$
$< 2, 2 >$
$< 1, 2 >$
$< 0, 2 >$
$< 3, 3 >$
$< 4, 4 >$
$< 5, 5 >$
$< 6, 6 >$
$< 7, 7 >$
$< 6, 7 >$
$< 5, 7 >$
$< 8, 8 >$
Note: Here the class $< 6, 7 >$ is not LCP-delimited as $LBL < SIL$ for this class and would not be considered in counting.

5. **Hashing to identify unique n-grams across articles.**
by Dinesh Bhirud
The hashing technique allows us to identify unique n-grams within and across processors. The hash value of each n-gram is calculated by taking the sum of the dictionary id of each word in the n-gram and taking a mod with the Hash table size. This identifies the bucket into which this n-gram will be placed. If that bucket is already occupied (hash clash), a chain of n-grams having the same hash value is created at that bucket. While adding, if the same n-gram is found, the count at the node is updated.
The advantage of this data structure is that reduces the time complexity of searching n-grams for finding uniqueness. We tried out a sequential search (as against the hash search) for finding uniqueness, and according to our estimates based on the experiments, that would have taken more than 12 hours to run on the corpus. Also, the same hashing technique is later used for reduction to find unique n-

grams across processors.

6. **Implementation of binomial Tree Reduction communication pattern.**
   by Prasad Kulkarni
   The worker-manager model suits well for data distribution. However, this model is not suitable for collecting results when each worker processor is done with the entire task assigned to it. If all the worker processors start giving their results to manager to combine the data then the manager will become a bottleneck. In fact, the manager won't remain manager anymore if it has to combine all data, perform reductions and find a unique solution to the given problem. Also, if all the workers start sending their own results to manager then it will create a huge serial component at the manager process to combine all the results. We won't really benefit from multiple processors available to us.
   We want some kind of reduction model that benefit from the multiple processors available, combine results in less number of steps and ultimately converge the results at process 0 (manager). The model that suits perfectly for these requirements is a binomial tree reduction pattern. From Chapter 3 of Dr. Quinn's book on Parallel Programming in C with MPI and OpenMP, "it is possible to perform a reduction of n values in log n message passing steps." In a binomial tree reduction, in the first step half of the processes send results and half of the processes receive the results and combine it with their own results. Now the sending processes go idle. In the next step, quarter of the processes send results and quarter of the processes receive the results and combine it with their own results. This procedures repeats with two sending processes and two receiving processes. Ultimately, the combined result is gathered at process 0 (manager). We have implemented this binomial tree communication pattern in our project to combine the results. The implementation of the binomial tree communication pattern is based on the assumption that the number of processors is a power of 2.

7. **Hash table reduce of unique n-grams following the binomial tree pattern**
   This task reduces the unique n-grams found by each processor into a list of all unique n-grams within the corpus. The hash table generated by the sender processes identified by the pattern from task 6, is sent to the receiver process, again as identified by task 6. Here, we have written code to loop through each n-gram sent and match them in the hash table of the receiver process. In the next iteration, the receiver processor becomes the sender and it sends this new modified hash table. Finally, the master process (process 0) would have a hash table that contains all the unique n-grams and their counts of the entire corpus.

8. **Finding interesting n-grams**
   By now process 0 has a huge hash containing all unique n-grams with the corresponding term and document frequencies. In this task we score all n-grams using $TF \times IDF$ measure and find out the requested number of top interesting terms using the formula described in the Introduction section.

9. **Scaling for the Gigaword corpus**
   With the system described above, we were able to find top interesting n-grams up

to 1.2 GB of data. The next big question was how can we scale our approach for a larger data set. Our main concern here was per process memory requirement.

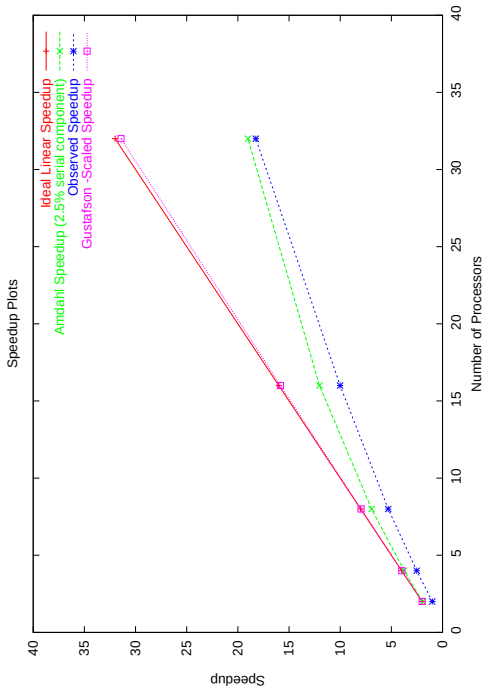
(a) **Pruning**
While carrying inter-process reduction, each receiving process has its own hash plus it gets a hash from the sender process in each iteration of the binomial tree reduction. This leads to increased per process memory requirement. As can be seen in the $TF \times IDF$ formula, n-grams with very low term frequency are not going to score high. We observed that if 1.2 GB of data is distributed among 32 processes, each process has $\approx 66\%$ n-grams having term frequency 1. We know that these terms are not going to score high. So our solution in order to scale for the larger data set was to eliminate such n-grams before inter-process reduction. We decide term frequency cut off depending upon the data set. We are compromising some accuracy here and our scores might not be perfect. To see if we are losing any top interesting terms, we carried few experiments on a smaller data set($\approx 1.2GB$) and observed that the top interesting terms remain same. One more concern here with this approach is if all 32 processes have a trigram say "THESE FEATURES OF" with a term frequency 1 and our cut off term frequency is 1. Each process will eliminate this trigram before inter-process reduction, which would have counted up to term frequency of 32 otherwise. However, we believe that for a large data set, term frequency of 32 is not at all competitive with the n-grams which would score high. Note that if any of the process has term frequency for this trigram > 1, it will NOT be eliminated and will be considered in the inter-process collection. For a very small data set, (data set below 1 GB) we suggest to keep our cut off frequency parameter value 0 i.e. no cut off frequency should be used for a small data set.

(b) **Stoplist**
The primary reason for the use of stoplist was the observation of not so interesting n-grams like "IN THE FIRST" scoring high and appearing in the top interesting terms. We thought of eliminating such n-grams while doing intra-process reduction. We created a stoplist containing words such as 'THE', 'OF', 'FIRST', 'IN', 'IS' etc and eliminated n-grams if all words in the n-gram are contained in the stoplist. For example, "IN THE FIRST" would be eliminated considering the above stoplist. This technique gave us really interesting terms in the output. There was an added advantage we got from this technique. As described in $TF \times IDF$ formula, n-grams with very high document frequency are not going to score high and we can get rid of those in order to reduce per process memory requirement. But it was not possible to do that until we were done with intra-process reduction. In order to scale our system for the Gigaword corpus, we wanted something adaptive which would help us eliminate not so interesting n-grams while doing intra-process reduction. And the stoplist technique gave us this advantage. Our stoplist is still evolving and currently it contains 160 words. Our stoplist is based on the dictionary and the stopwords should be coher-

ent with the dictionary.

All these techniques together reduced our per processor memory requirement and we could finally reach the Gigaword limit!

Figure 1: Speed Up Plots



5 Results and Performance Analysis

Figure 1 shows a snapshot of our output. These are first 5 interesting trigrams from the Gigaword corpus found by our system. The time taken by 32 processors was 1982.723 seconds. Figure 2 demonstrates our performance analysis. We used 1.2 GB data (first 50 files from the Gigaword corpus) to plot these graphs. The red line shows the ideal linear speed-up. The pink line, which is very close to the linear speedup line shows Gustafson's scaled speedup calculated using Gustafson's formula

$$\psi \leq p + (1 - p)s$$

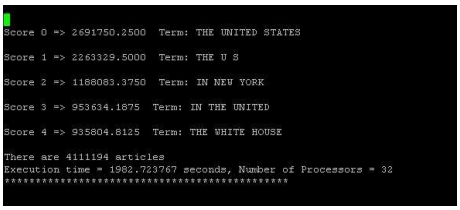
where,

- s is the fraction of total execution time spent in serial code
- p is the number of processors.

The blue line represents the observed speedup calculated using the formula

$$speedup = \frac{sequentialexecutiontime}{parallelexecutiontime}$$

Figure 2: Interesting trigrams from the Gigaword Corpus



5.1 Amdahl's Speedup - Serial component estimation

Amdahl's law states that Let f be the fraction of operations in a computation that must be executed sequentially, where $0 \leq f \leq 1$. The maximum speedup ψ achievable by a parallel computer with p processors performing computation is

$$\psi \leq \frac{1}{f+(1-f)/p}$$

Hence, for finding the speedup using Amdahl's law, estimating the serial fraction was the key. We identified the lines in the code that need to be executed sequentially and recorded the time needed for executing them. A ratio of this with total time gave us what we assumed as the serial fraction (f), which was around 1.5% However, the speedups according to Amdahl's law using 1.5% serial fraction were not in accordance with the speedups that we got experimentally for our system. This meant that there were certain serial portions of the code that were not timed or were not timeable. Hence, we revised the serial fraction to 2.5%, which exhibited speedups which were quite close to the observed ones. The difference between Amdahl's speedup and observed speedup it attributed to the communication overhead.

5.2 Space Complexity

Suppose there are n unique n-grams in each article and each processor processes m such articles Each process stores each unique n-gram of its share in the hash data structure. In worst case all n-grams across articles are unique. So we have to store $m \times n$ unique n-grams. Let l=number of unique n-grams / number of total n-grams. As number of unique n-grams increase, l approaches to 1. So overall memory requirement is $m \times n \times l$ For higher order n-grams, the number of unique n-grams increase which means l approaches 1. In that case the memory requirement becomes $m \times n$. This is the reason why we can't handle higher order n-grams for a very large data set.

6 Benchmarks

Following are the benchmarks for finding interesting trigrams on the Gigaword corpus.

Benchmarks

Processors	Execution Time (sec)
32	1982.723
64	1272.256
128	872.393
256	701.245

Note that we used the static dictionary created and therefore the timing of dictionary creation is not included.

7 Experiments

We carried various experiments in order to see the effects on the execution time.

7.1 Distribution Experiments

1. **Asynchronous distribution of data to all processors**
   The distribution strategy requires one processor to read the file sequentially and send the data to each processor in a cyclic fashion. In our implementation we are sending the data to the processors synchronously, i.e., before the manager processor (processor 0) sends data to processor i it waits for the acknowledgment from that particular processor. This synchronous approach might increase the execution time because as the manager waits for the acknowledgment, all other processors might go idle waiting for more data to come.
   To deal with this problem we tried sending the data asynchronously. The manager processor sends data to processor i without waiting for acknowledgement from it. The manager asynchronously sends data to each worker processor and the worker processor receives the data whenever it is ready. This asynchronous sends makes an inherent assumption that each processor has a message queue where the data coming from the manager will be pipelined.
   Result: This experiment did not turn out to be fruitful as the program seems to crash as some point.

2. **Manager probes for which worker is ready.**
   Instead of sending data synchronously to all worker processors in cyclic fashion, the manager probes for a worker processor that is ready to accept work. The manager does this by calling MPI_Probe. Once manager comes out of this function it knows the address of the processor which has sent the ready message. The manager then sends data to that processor only. The only concern in this approach was that some processors might starve. Since we do not know the implementation of MPI_Probe what if the manager sends data to only a subset of processors? If multiple processors send the ready message at same time how will the manager prioritize them?
   Result: This approach seems to take more execution time than the synchronous send receives.

3. **Strategy-2: The distribution strategy where each process knows its share**
   At first, we thought that all processors reading input files and choosing their share without communicating with each other might be an effective strategy. However, it turned out that multiple processors trying to read the same file at the same time makes the read operation incredibly slower. To quantify, it takes around 600s for 32 processors in contrast with 130s taken by the manager-worker model.

Though there is no communication, no wait time involved, it results in high execution time.
This lead us to an experiment where we made each processor read different file which significantly improved the timing from 600s to 359s. However, the manager-worker model gave the best execution time.

7.2 Other Experiments

1. **Can we reduce the huge serial component?** We observed that there is significant amount of serial component which is file i/o. Because of the high serial proportion in the computation, we are not able to see the effects of parallelism resulting in similar execution timings for 32,64,128 and 256 processors. We thought of and tried with two possibilities in order to improve file i/o timing

    - Instead of reading a file line by line using fgets, we read the entire file using fread. Then we work on the long string in memory. This didn't seem to give better results though. Moreover the potential problem with this strategy would be, if the entire text is contained in a single file, the memory requirement in that case would be unrealistic.
    - We have discussed before that when multiple processors are trying to read a file, the read operation slows down. We thought this might be the case with directory reading. Currently in a loop, we read a file in a directory, process it and go to the next file. During all this time the directory is open. We did an experiment with distribution strategy 2 to store all filenames along with their paths in an array of strings and then working on the array of strings. We observed a significant improvement in the execution timing.

2. **Finding word count by scanning the article string and counting spaces**
   For counting the number of words in an input article, we are using the C string function, strtok repeatedly, to get all the tokens separated by space. The experiment was to evaluate the relative performance of another strategy wherein I tried to count the number of words in an article by scanning through the string and counting the number of spaces in it. However, this required checking special conditions such as more than one spaces between 2 tokens/words, or a series of spaces followed by an end of line character.
   Result : This strategy was observed to be slower than the one which used "strtok", and hence discarded. Also, this strategy meant that the code needed to deal with uncertainties related to the formatting of the article, whereas the strtok function took care of all those for us.

3. **Creating all the 3 data structures suffix array, LCP vector and class array for each article and storing them along with the article text in memory for further use.**

The experiment was to find out the feasibility of creating all the data structures related to each article in memory and maintaining them in memory. Though this appeared possible in theory, actually the memory required for the storage was observed quite high. In fact, in an experiment, we had to allocate 3.5GB per processor (using the pmem parameter), to achieve this, which meant the memory requirement would be way too high to get through the queue at any decent speed. Hence, as a remedy to this, we have thought, not to store the data structures in memory. When we would be using them in the further stages of the projects (Beta stage, where we will be counting the number of n-grams), we would create the data structure per article, loop through its classes, generate counts and then free the memory associated with it. Hence, the result of this experiment was the conclusion that storing all the data structures in memory is not going to be a feasible option moving ahead. Work is in progress to generate counts from these data structures on the fly and then discard them.

8 Limitations

1. **Static Dictionary**
   Currently we are using a static dictionary, which has been built prior to the execution of the program, using NSP and the system sort command. We tried out creating a binary search tree representation of all the tokens in the corpus. However, we could not achieve this because the data was too large. We also tried creating the dictionary using an extra pass of the worker manager loop, wherein each worker called the system sort command. The plan was to combine the sorted output from each worker using binomial tree reduction. However, the program seemed to crash because of use of system call and we could not pursue this approach any further.

2. **Accepting the range M to N for size of n-grams fetched**
   We are currently able to get top n-grams of variable sizes (support to arguments M to N) only for a relatively small data set (up to 600MB). The reason for this is that, our data structures (especially the hash table) are currently not designed to handle variable sized n-grams. Hence, we have used the strategy of performing multiple passes of finding top interesting n-gram for each size and then sort these results to get top interesting n-grams of variable sizes. However, we do not think that this is a smart/efficient approach and believe that a reasonable sized change in the data structures can give multiple sized n-grams in a single pass. The code for our current implementation is in the “development” directory.

9 Conclusion

We have concluded from this project, that parallel systems like the IBM BladeCenter can be effectively used to perform computationally intensive operations like finding top interesting terms from large text corpora. We also, successfully demonstrated the value of data structures like suffix arrays, LCP vectors, classes and hash tables for such

problems. Our approach benefits with increasing number of processors ie we were able to successfully exploit parallelism on this system.

References

[GD03] Alexandre Gil and Gael Dias. Using masks, suffix array-based data structures and multidimensional arrays to compute positional ngram statistics from corpora. In *Proceedings of the ACL 2003 workshop on Multiword expressions: analysis, acquisition and treatment*, volume 18, pages 25–32. Association for Computational Linguistics, 2003.

[Jon72] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. In *J. Documentation*, volume 28, pages 11–21, 1972.

[KW98] Chunyu Kit and Yorick Wilks. The virtual corpus approach of deriving ngram statistics from large scale corpora. In *Proceedings of the 1998 International Conference on Chinese Information Precessing, Bejing, China*, pages 223–229, 1998.

[Sav90] Jacques Savoy. Statistical behavior of fast hashing of variable length test strings. *SIGIR Forum*, 24(3):62–71, 1990.

[YC01] Mikio Yamamoto and Kenneth W. Church. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Comput. Linguist.*, 27(1):1–30, 2001.