Text Relatedness using Google Trigrams: Efficient Parallelizations

Abstract. Text relatedness measure is an important Natural Language Processing task in many Text Mining applications. Google Trigram Method, an unsupervised corpus-based approach, has been shown to capture text relatedness well. The challenge is in making Google Trigram Method practical given its high computational complexity. This paper presents time and space efficient approaches for implementing Google Trigram Method to make it applicable for large amounts of text. In order to improve the performance, careful algorithmic engineering, data structure enhancement, and parallel computing are applied in the implementation. In particular, two parallel methods are discussed in this paper: shared memory multithreaded implementation and Hadoop-based implementation. Both parallel methods provide an order of magnitude improvement in accelerating the text relatedness measure.

1 Introduction

Text relatedness[1] plays an important role in many areas of Text Mining, such as information retrieval [11], text categorization [12], text summarization [6,10], text classification and text clustering [15]. There are two types of approaches broadly used for finding the relatedness between texts: corpus-based unsupervised approaches and lexical-based supervised approaches. The corpus-based unsupervised approaches have some advantages over their counterpart: first, to create, maintain, and update lexical databases or resources (e.g., WordNet or Roget's Thesaurus) require expertise and efforts; second, coverage of words in lexical resources is not enough for many Text Mining tasks; third, lexical resources are language specific whereas corpus-based approaches are, in general, language independent as long as there are enough texts of a language [8]. Google Trigram Method (GTM) published in Canadian AI 2012 [9], which determines the relatedness between two texts using Google Unigram and Trigram corpus [3], is one of the state-of-the-art methods of the corpus-based unsupervised approach.

However, applying GTM for large texts is expensive. For example, computing the relatedness between two texts each containing one hundred words takes approximately three hours. In addition, many Text Mining tasks, such as clustering or classification of N texts, require N to N text relatedness computation, in which $\frac{(N-1)N}{2}$ pairs of text relatedness need to be computed. Applying GTM in such scenarios is not practical.

[1] The term "similarity" has been used in the past to characterize the computation that is more accurate to be described as "relatedness", which subsumes similarity. Therefore, we use relatedness in this paper.

This paper mainly focuses on the efficient implementation of the GTM for text relatedness computation. Space and speed improvement are achieved by corpus preprocessing, data structure enhancement, and parallel computing. In-depth experimental evaluation and systematic tuning are also addressed in the paper. One of the key ideas to improve the real-time performance is to compute the intermediate data off-line, so that the intermediate data can be looked up rather than having to be computed on-line. For space and time efficient in-memory word relatedness lookups, different dictionary data structures are explored (Nested Hash, Hash with Concatenate Keys, Parallel Blocking Array). Two efficient parallel methods are designed for the N to N text relatedness task: the first uses shared memory mutlicore machines, and the second uses MapReduce model [5].

The rest of this paper is organized as follows: Section 2 presents a brief overview of the Google Trigram method; Section 3 describes the preprocessing step; Section 4 introduces the data structure enhancements; Section 5 discusses the two parallel methods: multi-core and MapReduce; Section 6 presents the performance evaluation of different data structure enhancements and the two parallel methods.

2 Google Trigram Method

GTM is an unsupervised corpus-based approach for measuring word and text relatedness. GTM uses unigrams and trigrams from the Google Web 1T N-gram corpus [3] to find the relatedness of a pair of words, and extends the word relatedness method to measure the text relatedness.

The Google Web 1T N-gram corpus, contributed by Google Inc., contains English word n-grams (from unigrams to 5-grams) and their observed frequency counts calculated over one trillion words from web page texts collected by Google in January 2006 [3].

2.1 Word Relatedness Based on Trigrams

GTM measures the relatedness of two words by considering the trigrams that start and end with the given pair of words and normalizing based on their unigram frequency. The main idea is to take into account all the trigrams that start and end with the given pair of words and then normalize their mean frequency using unigram frequency of each of the words as well as the most frequent unigram in the corpus used [8,9]. The notations used in the following equations are shown in Table 1.

$$\mathrm{GTM}(\omega_1,\omega_2) = \begin{cases} \dfrac{\log \frac{\mu_T(\omega_1,\omega_2)C_{\max}^2}{C(\omega_1)C(\omega_2)\min(C(\omega_1)C(\omega_2))}}{-2\times\log\frac{\min(C(\omega_1),C(\omega_2))}{C_{\max}}} & \text{if } \log\frac{\mu_T(\omega_1,\omega_2)C_{\max}^2}{C(\omega_1)C(\omega_2)\min(C(\omega_1)C(\omega_2))} > 1 \\ \dfrac{\log 1.01}{-2\times\log\frac{\min(C(\omega_1),C(\omega_2))}{C_{\max}}} & \text{if } \log\frac{\mu_T(\omega_1,\omega_2)C_{\max}^2}{C(\omega_1)C(\omega_2)\min(C(\omega_1)C(\omega_2))} \le 1 \\ 0 & \text{if } \mu_T(\omega_1,\omega_2) = 0 \end{cases}$$

2.2 Text Relatedness Based on Trigrams

The text relatedness measurement task is to derive a score between 0 and 1 that indicates the relatedness between two texts [9]. In general text relatedness can be expressed as a function of word relatedness. In GTM the main idea of measuring text relatedness is to find the relatedness using GTM on word relatedness between each word pair of the two texts [9]. For given documents P and R where $|P| \leq |R|$, first all the same words are removed, and then a matrix with the remaining words $P' = \{p_1, p_2, \cdots, p_m\}$ and $R' = \{r_1, r_2, \cdots, r_n\}$ is built where each entry $a_{ij} \leftarrow GTM(p_i, q_j)$.

$$M = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

From each row $M_i = \{a_{i1} \cdots a_{in}\}$ in the matrix, select the significant elements $A_i = \{a_{ij} | a_{ij} > \mu(M_i) + \sigma(M_i)\}$, where $\mu(M_i)$ and $\sigma(M_i)$ are the mean and standard deviation of row i. The sum of the means of all the rows is $\sum^m a_{i=1}\sigma(A_i)$. Then we can compute the document relatedness using

$$Rel(P, R) = \frac{(\delta + \sum^m a_{i=1}\sigma(A_i)) \times (m + n)}{2mn}$$

where δ is the number of removed words when generating P' or R'. The other notations used in the above equation are shown in Table 1

3 Word Relatedness Preprocessing

As shown in Section 2, finding word relatedness is the core operation for measuring text relatedness using GTM. Since using GTM to find text relatedness required finding relatedness between many different pairs of words, calculating word relatedness takes a long time. As the number and length of input texts increase, the cost of finding word relatedness on-line increases accordingly.

Notation	Description
$C(\omega)$	Frequency of the word ω.
$\mu_T(\omega_1, \omega_2)$	Mean frequency of trigrams which either start with ω_1 and end with ω_2, or start with ω_2 and end with ω_1.
$\sigma(a_1, \ldots, a_n)$	Standard deviation of numbers $a_1, \ldots, a_n$
$C_{\max}$	Maximum frequency among all unigrams.

Table 1: Notation used for GTM word relatedness measurement

Thereby, finding word relatedness off-line is a necessary step to make GTM applicable for text relatedness measurement in practise.

The preprocessing step finds all the relatedness between all the word pairs that exist in the Trigram corpus using GTM, which is an off-line step, so that a word relatedness dictionary file is derived and stored. In order to reduce the in-memory size of the word relatedness dictionary each word is assigned an unique numeric ID, and the word relatedness results are converted from plain text to binary code at the end of the preprocessing step because loading binary streams is faster than loading plain texts into memory. The pre-processing step significantly improves the real-time performance because the word relatedness can be looked up instead of computed on-line, and repetitive word relatedness computation are avoided.

4 Efficient Structures for Word Relatedness Lookups

In GTM text relatedness computation, word relatedness is intermediate data that can be precomputed off-line to reduce the time for on-line computation. To compute the text relatedness, the word relatedness dictionary needs to be loaded in memory for lookups. Since the word relatedness dictionary uses word IDs to identify the word relatedness, words in tokenized texts need to be converted to IDs for lookups in the dictionary. Efficient implementation of the classic dictionary data structures [4] are required for retrieving IDs for words, as well as storing and searching word relatedness. Both time and space efficiency are taken into consideration here. However, these two goals are countervailing sometimes. This section first introduces a dictionary structure for retrieving the ID for each word. For the construction of the dictionary of word relatedness in-memory, four potential data structures are explored.

4.1 Data Structure for Word ID and Frequency

Converting each word to an ID, retrieving word frequencies, and finding the ID of each word efficiently require an implementation of the dictionary structure. The design of the structure is shown in Figure 1.

Word Map is a word ID dictionary implemented by a hash table, mapping word to its ID which is a number in our implementation. The ID is unique for each word that exists in the Google unigrams. The keys of the hash table are the words, and the values are the IDs of the words.

Word Statistics is a dictionary implemented by an array that stores word frequencies from the Google unigrams. The indices of the array are the IDs of the words, and each entry contains the frequency of the corresponding word.

4.2 Data Structures for Word Relatedness

An in-memory dictionary storing word relatedness needs to be constructed for text relatedness measurement. In order to achieve fast lookup and minimize the memory cost, four dictionary structures are designed and explored as following:

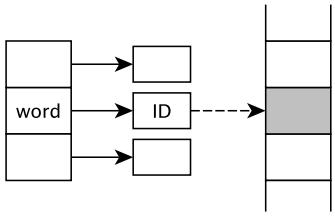
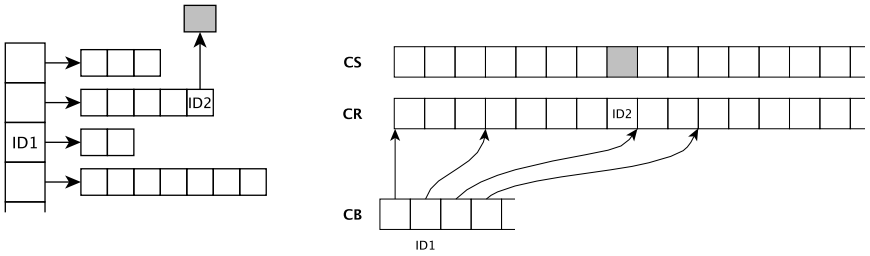


Fig. 1: Index structure for word statistics.

A two-dimensional array can be used to directly access the word relatedness (henceforth, Direct Access). In theory, Direct Access has the fastest retrieving speed, but the size of it is too large to fit into memory since the range of word ID is the number of words in the Google unigrams.

A more space-efficient structure is Nested Hash Map, implemented by two-level hash maps to store the word relatedness, as shown in Figure 2a. Nested Hash Map consists of a primary hash table indexed by the IDs of the first words in word pairs. The value associated with each index of the primary hash table is a reference to a secondary hash table indexed by the IDs of the second words in word pairs. Each value of the secondary hash table stores the corresponding word relatedness.



(a) Nested Hash Map. (b) Parallel Blocking Array.

Fig. 2: Word relatedness data structures.

To further decrease the size of the word relatedness dictionary in memory, Hash Map with Concatenated Keys is designed and implemented using only one hash table. This structure uses the concatenation of two words as the key and the relatedness between two words as the value in the hash table.

Another way to reduce the memory cost of the structures implemented by hash tables is to use parallel arrays. The Parallel Blocking Array structure is illustrated in Figure 2b. The first array, Range Array (RA), is indexed by the

IDs of the first words in word pairs, and declares the range of the blocks in the second array, Block Array (BA). Each block in the BA stores the second word ID associated with the key in the RA. The third array, Word Relatedness Array (WA) is parallel with the BA, and it stores the relatedness of two words of which IDs are the corresponding index of the RA and the corresponding value of the BA. Given two words, a lookup operation first finds the value indexed by the ID of the first word in the RA. The value found in the RA defines the starting index and the ending index of the associative block in the BA. Binary Search is performed on the BA to find the ID of the second word. If the ID of the second word is not found in the corresponding block in the WA, the operation returns 0. If the ID of the second word is found, the corresponding element in the WA is returned as the relatedness between the given two words.

5 Parallel Methods for Efficient Implementations

In this section, the two parallel methods for applying GTM on the N to N text relatedness task are explored: the first is a shared memory multithreaded parallel implementation and the second is Hadoop-based parallel implementation [7, 16], where Hadoop is a MapReduce Framework.

5.1 Shared Memory Multithreaded Implementation

The shared memory multithreaded implementation makes use of multicores of shared memory machines. In the implementation, a word relatedness dictionary is constructed using one of the dictionary data structures, and input texts are read into token arrays in the shared memory. The construction of the word relatedness dictionary and the text token arrays is the sequential part of this parallel implementation. In the preprocessing step, the word relatedness statistics are converted into binary format, so the word relatedness dictionary can be constructed faster using binary streams. The parallel part of this implementation uses threads to partition the workload to achieve speed-up on a multicore machine. A synchronized function, serving as a job scheduler, dynamically passes one pair of text IDs to each thread a time. Each thread fetches the corresponding texts from the shared memory and computes the relatedness between them. Word relatedness lookups are executed concurrently during the computation of text relatedness. All threads run in parallel until all pair-wise text relatedness are computed.

The limitation of this method is that it is implemented on a single multicore machine with all the texts stored in the memory of that machine. Thereby, the parallelism of this method is limited to the resources of one machine.

5.2 Hadooop Based Implementation

The second method employs the Hadoop framework to compute the pair-wise text relatedness in parallel. Each mapper is responsible for constructing a word

relatedness dictionary and calculating the pair-wise text relatedness within a subgroup of texts. Since the word relatedness dictionary takes large space in memory, each Hadoop node can only run one mapper task at a time. In order to fully make use of the multicores in one node, multithreaded mappers [16] are applied. Initially, the order of input texts is randomized so that it does not correlate with their length. As illustrated in Figures 3, the mapper task scheduler partitions the text pairs into square blocks, where a set of blocks are passed to a multithreaded mapper.

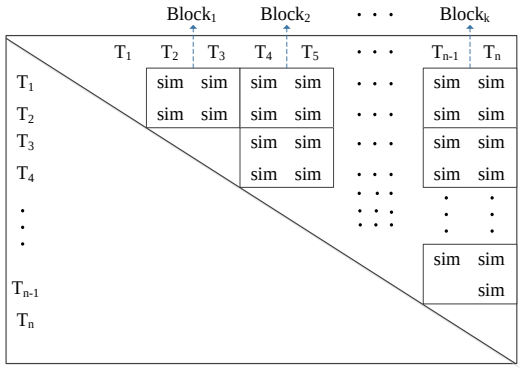


Fig. 3: The Up-triangle Text Relatedness Matrix used for Mapper task distribution.

Each mapper uses multithreads to partition the workload. Each thread of a mapper retrieves texts from the Hadoop Distributed File System [14] based on the block information and calculates the relatedness between each pair of texts. This task partitioning method minimizes the I/O time required by each mapper because the number of texts a mapper needs to retrieve from the distributed memory is minimized. Finally, the reducer [16] receives the output from each mapper, sorts the keys which are text IDs, and then writes the final results into one output file.

The following is a more detailed description of the mapper and the reducer with a particular focus on mapper tasks. A mapper task implements the Parallel Blocking Array Dictionary data structure and performs the word relatedness lookups, which are the critical components in terms of the computation efficiency.

Mapper: The mapper, shown in Algorithm 1, takes as input blocks derived from the texts matrix, each of which represents a subgroup of texts, and it retrieves the corresponding texts from the distributed file system. Each mapper also constructs an in-memory word relatedness dictionary for further lookups

Algorithm 1: Mapper in the Text Relatedness Application

input : $Blocks : \{B_1, B_2, \cdots, B_m\}$ where m is the number of blocks in a task, and m is greater than the number of threads in the mapper.
output: A list of entries $\{\mathrm{T}'_a\ \mathrm{T}'_b\ \mathbf{SIM}\}$ which are pairs of Texts IDs and their relatedness.

```
1  Construct the word relatedness dictionary from co-occurrence statistics using
   the Parallel Blocking Array data structure described in Section 4.
2  Create a concurrent text buffer to store text instances.
3  Parallel for each block, B_i in Blocks do
4      Retrieves the corresponding pre-processed texts from the distributed
       systems and stores them in the concurrent text buffer if the text is not in it
       yet .
5      for each pair of texts, T'_a, T'_b, instances in B_i do
6          Apply the Google Trigram Text Relatedness Algorithms to compute the
           relatedness between the texts:
7          SIM ← Google Trigram Text Relatedness( T'_a, T'_b)
8          Emit({T'_a T'_b SIM})
9      end
10 end
```

and text relatedness computation. Then the mapper processes the text pairs using multithreads in parallel, and thus keeps a concurrent buffer to store text information instead of repetitively loading the same texts. A mapper generates a set of key-value pairs, where the keys are the IDs of a pair of texts and the values are the relatedness scores between the text pairs.

Reducer: The reducer takes the key-value pairs generated by mappers as input, sorts the pairs by their IDs, and aggregates all the key-value pairs in one output file.

6 Experimental Evaluation

6.1 Experimental Setup

For the evaluation of candidate word relatedness dictionary structures and shared memory multithreaded parallel implementation, the experiments were performed on a Linux server containing 256 GB main memory and 16 Intel Xeon E5-2650 processors (32 logical cores). The Java version was 1.7.0.03 and the experiments were run when more than 90% CPU and memory were free.

For the evaluation of the Hadoop-based parallel implementation, the experiments were performed on Amazon EC2 m3.xlarge nodes [1], each of which had 4 cores and 15 GB. AWS EMR [2] was used in the experiments, and it was configured to ami-version 3.2.1 which is equivalent to Hadoop 2.4.0. In order to reserve enough RAM for the data structures in each mapper, each node is set to run one mapper a time and the Java heap size of each mapper task is set to 4 GB.

The experiment data set consists of documents generated from the abstracts of 10,000 papers published in ACM digital library where each abstract contains around 200 words. For the word relatedness dictionary structures evaluation, the input data are word pairs that randomly generated from the abstracts.

6.2 Word Relatedness Dictionary Structure Evaluation

The performance of the candidate word relatedness dictionary structures are evaluated in terms of time and space cost. The Direct Access structure cannot fit into the memory of the experimental machine, so Nested Hash Map, Hash Map with Concatenate Keys, and Parallel Blocking Array are compared in the evaluation.

In the experiments, relatedness of word pairs ranging from 100,000 to 30,000,000 are tested using the four candidate dictionary structures. Figure 4a shows the time taken in seconds as a function of the number of word relatedness lookups. It can be observed that linear growth is achieved by all the dictionary structures as the number of test cases increases. Table 2 compares the in-memory sizes of four word relatedness dictionary structures.

Since the implementation is written in Java, various JVM parameters that impact the garbage collection overhead and the performance data structures were tuned for experiments. These parameters include the initial heap size and the ratio between the amounts of space allocated to the young and old generations in Java's garbage collector. A higher initial heap size makes the program use more memory initially but reduces the number of times the heap needs to be resized as the heap space becomes insufficient to hold newly created objects [13]. Verified experimentally, with limited memory space, setting the initial heap size to 4 GB results in the best performance for using Parallel Blocking Array to store word relatedness dictionary, and setting the initial heap size to 10 GB results in the best performance for using Nested Hash Map or Hash Map with Concatenate Keys. A higher ratio between the young and old generation's space allocations reduces the frequency of minor garbage collection runs but may lead to old objects remaining in the young generation's space if there is no room left in the old generation's space [13]. Verified experimentally, setting the young-old ratio to 1:3 results in the best performance for the implementation.

Shown in the Table 2, using Parallel Blocking Array structure to store the word relatedness dictionary takes the least space in memory. Despite the fact

Structure	Size[GB]
Direct Access	NA
Nested Hash Map	9.1
Hash Map with Concatenate Keys	8.6
Parallel Blocking Array	3.7

Table 2: Comparison of the in-memory size of dictionary structures

that Parallel Blocking Array retrieves word relatedness slower than the other two structures, the word relatedness dictionary is constructed using the Parallel Blocking Array structure in the parallel implementations due to memory restriction.

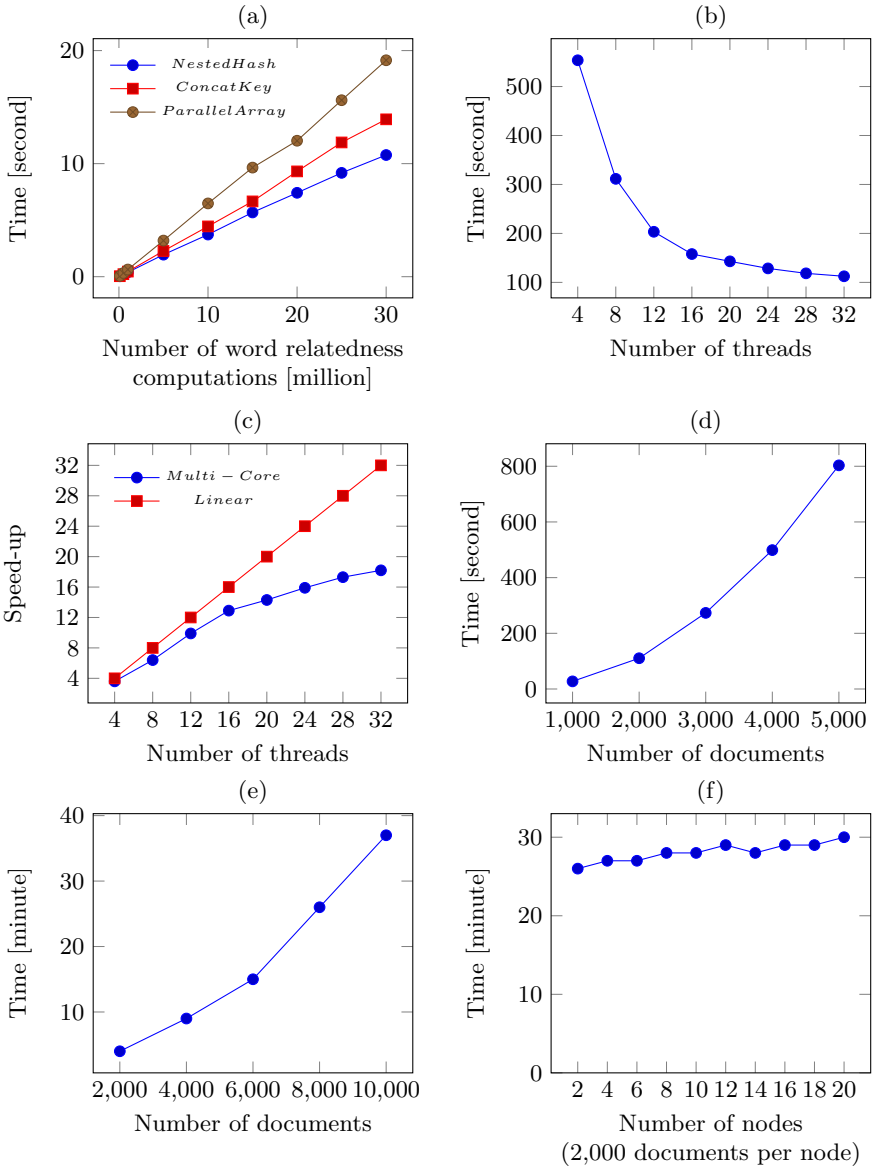


Fig. 4: Evaluation results.

6.3 Shared Memory Multithreaded Implementation Evaluation

The *speed-up performance* of the shared-memory multi-threaded implementation is the ratio between the running time it achieves on a single core and on P cores. *Linear speed-up* means that the speed-up for P cores is P, that is, the work is perfectly balanced across the cores. For the speed-up test, input size was fixed at 2000 files and the number of threads was increased from 1 to 32. Figures 4b and 4c show the running times and speed-up achieved, respectively. The construction of word relatedness dictionary is always sequential which is a fixed overhead. Even so, up to 16 threads used, the speed-up is close to linear. The speed-up between 16 and 32 threads is modest as the machine has only 16 physical cores. The multi-core implementation achieves a speed-up of 20 with 32 logical cores which is in line of our expectation.

The *size-up performance* shows how the running time increases with the input size for a fixed number of cores. We fixed the number of threads at 32 and increase the input size from 1,000 to 5,000. Since the number of the pair-wise document relatedness computation is $\frac{(N-1)N}{2}$ for N input documents, running time will increase with the order of $O(n^2)$. The size-up curve in Figure 4d shows that the running time is a polynomial function of the number of documents. The application can compute the relatedness within 1,000 documents in 30 seconds and 5,000 documents in 15 minutes.

6.4 Hadoop-based Implementation Evaluation

In order to demonstrate the *size-up performance* of the Hadoop-based implementation, the number of nodes were fixed at 20 and the number of input documents was increased from 2,000 to 10,000. Figure 4e shows the running time increases as expected. The result shows that the Hadoop-based implementation can process 10,000 documents in less than 40 minutes.

The *scale-up performance* shows the running time of the implementation while keeping the ratio between input size and nodes fixed. If the running time remains constant, it implies that the implementation can scale to process a larger input size. In order to demonstrate the *scale-up performance* of the Hadoop-based implementation, the number of document pairs processed by one node was set to 2,000 and the number of nodes was increased from 1 to 20. Figure 4f illustrates the scale-up running time which increases only slightly as the number of nodes increases. The slight increase is caused by the job scheduling of the Hadoop framework and the network traffic. The steady scale-up time demonstrates that the implementation is able to process a large number of documents.

7 Conclusion and Future Work

This paper shows that with careful algorithmic engineering and efficient parallelizations, GTM can be made practical and feasible for text relatedness computation. Two parallel implementations were explored, and both improved the

performance of GTM significantly. The shared memory multithreaded implementation achieves a close-to-linear speedup with the number of physical cores. The Hadoop-based implementation has a steady scale-up performance with the number of nodes.

In the future, the high-performance parallel implementations of GTM will be applied in text relatedness based applications, such as text clustering and classification.

References

1. Amazon Web Services, I.: Amazon elastic compute cloud (amazon ec2) (January 2015), http://aws.amazon.com/ec2/
2. Amazon Web Services, I.: Amazon elastic mapreduce (January 2015), http://aws.amazon.com/elasticmapreduce/
3. Brants, T., Franz, A.: Web 1t 5-gram corpus version 1.1. Google Inc (2006)
4. Cormen, T.H., Leiserson, C.E., et al.: Introduction to algorithms, vol. 2 (2001)
5. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
6. Erkan, G., Radev, D.R.: Lexrank: Graph-based lexical centrality as salience in text summarization. J. Artif. Int. Res. 22(1), 457–479 (Dec 2004)
7. Hadoop, A.: Hadoop (January 2015), http://hadoop. apache. org
8. Islam, A., Milios, E.E., Keselj, V.: Comparing word relatedness measures based on google n-grams. In: COLING (Posters). pp. 495–506 (2012)
9. Islam, A., Milios, E.E., Kešelj, V.: Text similarity using Google tri-grams. In: Canadian Conference on AI. Lecture Notes in Computer Science, vol. 7310, pp. 312–317. Springer Berlin Heidelberg (2012)
10. Lin, C.Y., Hovy, E.: Automatic evaluation of summaries using n-gram co-occurrence statistics. In: Proc. HLT-NAACL. p. 8 pages (2003)
11. Liu, T., Guo, J.: Text similarity computing based on standard deviation. In: Proceedings of the 2005 International Conference on Advances in Intelligent Computing - Volume Part I. pp. 456–464. ICIC'05, Springer-Verlag, Berlin, Heidelberg (2005)
12. Park, E.K., Ra, D.Y., Jang, M.G.: Techniques for improving web retrieval effectiveness. Inf. Process. Manage. 41(5), 1207–1223 (Sep 2005)
13. Rau-Chaplin, A., Yao, Z., Zeh, N.: Efficient data structures for risk modelling in portfolios of catastrophic risk using mapreduce. In: Proceedings of the International Conference on Computational Science, ICCS 2014, Cairns, Queensland, Australia, 10-12 June, 2014. pp. 1557–1568 (2014)
14. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on. pp. 1–10. IEEE (2010)
15. Wegrzyn-Wolska, K., Szczepaniak, P.S.: Classification of rss-formatted documents using full text similarity measures. In: Lowe, D.B., Gaedke, M. (eds.) ICWE. Lecture Notes in Computer Science, vol. 3579, pp. 400–405. Springer (2005)
16. White, T.: Hadoop: the definitive guide: the definitive guide. " O'Reilly Media, Inc." (2009)