Project Report

Google Star

Course Instructor: Dr. Ted Pedersen

Team Members:

Bhalekar Prafulla

Mehta Sarika

Naik Aneerudh

Nepalia Ankur

Index

Table of Contents:

Bhalekar, Prafulla:	Sections 2.3, 2.5, 2.9, 4.2, 4.3
Mehta, Sarika:	Sections 1.1, 2.1, 2.7, 2.11, 3, 4.1, 4.2
Naik, Aneerudh:	Sections 1.2, 1.3.2 – 1.3.5, 2.2, 2.6, 2.10, 4.2, 5
Nepalia, Ankur:	Sections 1.3.1, 2.4, 2.8, 4.1, 4.2

Abstract

The project aims at building a co-occurrence network from the Google n-gram data using the C programming language, MPI and OpenMP. The main goal is to be able to access the user defined data element combinations efficiently from the co-occurrence network for the purpose of some custom application. This could be either speech recognition or predicting certain trends in internet searching based on the frequency of the data returned by each search query in the Google search engine.

The amount of data provided in the dataset is very large and a lot of correlations can be drawn from this set. One particular operation that is being performed is checking how many words do form a network. Another operation could be to check how many disjoint networks for sub domains in terms of word patterns and search queries.

Ideally, the objective would be to specify a word and a number. This would search for the particular words and return all the words that are connected to it (in forward and reverse order) up to the specified number of nodes. Another variation is to print the disjoint networks (if any) present in the dataset. These would be the words that form a network but are connected to only a certain subset of words from the dataset. There can possibly be multiple disjoint networks in the dataset.

1. Introduction

1.1

The Google n-gram dataset is English words along with the frequency of occurrence, provided by Google Inc. The length of the n-gram ranges varies from unigram to five – grams. Suitable operations performed on the data could reveal possible patterns that provide new insights in statistical language modeling or for other uses.
The data was generated from approximately 1 trillion word tokens of text from publicly accessible web pages. All text provided is in the UTF-8 text encoding format.

Tokenization of data elements:

The data was tokenized in a manner similar to the tokenization of the Wall Street Journal portion of the Penn Treebank. Notable exceptions include the following:

1. Hyphenated word are usually separated, and hyphenated numbers usually form one token.

2. Sequences of numbers separated by slashes (e.g. in dates) form one token.

3. Sequences that look like urls or email addresses form one token.

Data sizes and specifications:

Number of tokens: 1,024,908,267,229

Number of sentences:	95,119,665,584
Number of unigrams:	13,588,391
Number of bi-grams:	314,843,401
Number of tri-grams:	977,069,902
Number of four-grams:	1,313,818,354
Number of five-grams:	1,176,470,663

1.2 Co-occurrence Networks:

In general, Co-occurrence networks are abstractions of data to gain insights into relationships among certain generic types of variables. The simple idea is that if the occurrence pattern of such variables is highly correlated, i.e. they occur multiple times within the dame data set, they are in some way related to each other. For each pair of objects, we count the number of co-occurrences. This data can be visualized as a graph, where nodes correspond to objects and edges describe found co-occurrences.

Usually, the co-occurring words have a weight associated with their occurrence. This might be the number of times the pair of words actually occurs or some other mathematical quantity dependant specifically on the particular domain in which the co-occurrence network is being used.

In terms of implementations, the network can be implemented in terms of either a weighted or non weighted graph.

1.3 Previous Work:

The following papers were referred to by the Fluminense team for gaining background knowledge and information of the general topics related to the Google Star project.

1.3.1 "Co-occurrence Vectors from Corpora v/s Distance Vectors from Dictionaries".

In this experiment a comparison was made of vectors from large text corpora and of vectors derived by measuring the inter word distances in dictionary definitions. The suggestion is that distance vectors contain some different semantic information that provided by co-occurrence vectors.

This paper showed by experiments that learning positive or negative meanings from example words, distance vectors provide a higher precision than co-occurrence vectors. Therefore there exists a possibility that distance vectors have some other semantic information as compared to co-occurrence networks. This increases the possibility that more information and trend predictions in machine learning and artificial intelligence could use distance vectors as the preferred input as compared to co – occurrence vectors.

1.3.2 "Choosing the Word Most Typical in Context Using a Lexical Co-occurrence Network".

This paper is about predicting the best possible synonym for a particular word, and doing it efficiently.

The author explores the possibility of using second order co-occurrence relations. By experimentation, it was proved that employing second order co-occurrence networks improved the reach of the lexical choice program.

1.3.3 "Discovering Word Senses from a Network of Lexical Co-occurrences".

This paper explores the possibility of defining the semantic and natural language 'sense' of words from a network of lexical co-occurrences built from a large data corpus. The 2 major languages that this was tested for was the English and French languages.

Semantic resources always prove to be useful in information retrieval and query expansion applications. The 3 main methods this is done is by is building a class of related words and not constraining ourselves in defining word sense strictly, using features in the word neighbourhood of a word and thirdly by deriving the word senses from the co occurrence pairs in a data corpus.

1.3.4 "Semantic Co-occurrence Networks".

This paper delves into the issue of correctly inferring the real sense of a word used in a context. Most often, there is a trigger word that is used to extrapolate to the word sense used in the context. Previous work in this field led to the conclusion that statistical co occurrence networks built for a large corpus would help in disambiguating the word sense for a word that could in fact have a different meaning for each context in which it was used. The paper describes 3 methods used to correctly disambiguate the sense of words. First, a neural network is used on a small collection of words, which does a good job assigning the correct meaning to the word. Second, the neural network is applied to a large corpus such as a dictionary of words, which certainly has more than one meaning for most of the words. Third, the same technique is used on a quarter million words of parallel French and English.

The basic working of the neural network is based on a set of real number values. One is the activation level associated with each node, and a bias. Then, there is a real number called weight associated with each edge in the neural network. Based on these values, the errors for each output term are calculated, which are then used to check whether the word sense was correctly assigned to the word that was fed in as input to the neural network. In applying the neural network to the dictionary, the training was conducted based on the financial and geographical meanings of the word 'bank', and certain context words that co occurred at least 3 times with the word 'bank'. In the third technique, pairs of English

words were mapped to their French translations. When a pair mapped to the same translation more then 85% of the time, the co occurrence was marked as significant. The conclusion from these experiments was that using neural networks makes it possible to generate correct senses of words in small to medium corpora.

1.3.5 "Conceptual Grouping in Word Co-occurrence Networks".

This paper describes a technique called conceptual grouping, which automatically distinguishes between the real word senses submitted in a user query, and then sorts the documents returned by that particular query. This can help users find what they want when a query is submitted and it improves the precision of results returned. Instead of providing users with results that seem too many and confusing to consider, it would be better to get results that are tailored to the specific query of the user.

The technique shown in this paper is that if a word is entered as a query, all semantically related words in that word's co-occurrence network are checked. The technique also uses broad 'groups' of 'concepts' under which most or all of the words come under. These concepts within 'groups' as they are called, can have over lapping words in them. Now, when a particular word has been entered as a query, we check the co-occurrence for that word and also check which words from the co-occurrence network occur in the same conceptual group under which the query word is. Empirically, it has been found that words that co-occur within the same conceptual group will be linked more strongly semantically in the co-occurrence network as well. That way, the real sense of a word

could be gauged by checking under which conceptual group that word and the words in its co-occurrence networks occur.

2. Approach:

2.1

The approach the Fluminense team chose was purely based on a interactive brainstorming session.

Our approach was to use a fairly simple concept which would make it easy to search and traverse a network when a particular word is used as a query. For building this concept into a program we started out with the 2 gram data files. This 2 gram data is formatted in such a way that there are 2 character strings per line with the frequency count of many times they occur in pairs. Each of these words is separated by a tab white space character.

2.2

A simple yet powerful hash function has been used in our program. The hash function used is an integral part of our tree creation and search algorithm as it reduces the time for traversal exponentially as compared to other traversal methods. We have used the “djb2” hashing function along with another core component of the “Lau” hash function. Both these hash functions are computationally similar, and are ideally suited for working with strings. As the time needed to come up with a custom hash function is very large when working within academic time constraints, it was decided to make use of existing hash functions.

It was first tested on the 1 gram data, just to check the total amount of collisions that occur when the same hash value is generated for different strings. When the hash function was tried on the 1 gram data, it was found that the maximum hash value was around 600 and the minimum value was 350. These figures were obtained on the premise that the first hash table would be around 10000 elements.

The first word on each line from the bi-gram data would be hashed.
Upon reaching the first word in the bi-gram data, the hash function is applied that returns the position in the hash table that will hold the structure for the first word.
Then the second word is also hashed and this time again the hash function returns the actual element number in the array where the second word's hash value is stored.

Each element in the hash table for the second string will have a binary search tree associated with it.
The use of binary search tree was chosen based on the following factors:
1. It takes log(n) time to search the depth of the tree.
2. The bi-gram data is already sorted so we could easily arrange and insert incoming new data into the child nodes in the correct order.
3. In order traversal is a highly efficient operation on Binary Search Trees.

2.2.1) Unigram Cut:

Unigram cut is the minimum number of times each unigram in a bigram must have occurred for a bigram to be included in the network.

2.2.2) Associativity Cut:

Assoc cut is the minimum "Association score" that a pair of words must have in order to be included in a path.

The Association score of a bigram

"W1W2" = Freq(W1,W2)/Freq(W1)*Freq(W2)

2.3

After the hash function has been applied on the first string in the bi-gram data file, the 'root node' of the binary search tree is created. This root node contains the 'occurrence node'.

The occurrence node is the second string that occurs with the first string in the bi-gram data file. Now there can be multiple combinations of second different second strings with the first string. For this purpose an array is dynamically declared for storing the initial second strings that occur with the first string.

The 'occurrence node' data structure has the following main elements. It has the initial second word that occurs with the first string. It also has the frequency count of those 2 strings as provided to us in the bi-gram data file. Then it has 2 flags called 'front' and

‘back’. These flags are used to indicate if the combination of the 2 words also occurs in reverse order in the bi-gram data file.

The operations of inserting into the nodes are fairly straightforward. The data element to be inserted is compared with the root element. If it is alphabetically smaller than the root element, then it becomes the left child of the root, else if it is bigger than the root element, it is made the right child of the root element.

The tree creation operation (part of the main network creation operation)

This function in the program is passed 2 strings on an entire line from the bi-gram data file as the input parameters. The line has 2 strings and the frequency of their occurrence.

Example:

Bi-gram data line:

Bombay	Tanvir	3457
Rambo	Rocket	285
Rocket	Rambo	76
Tanvir	Pilaf	970

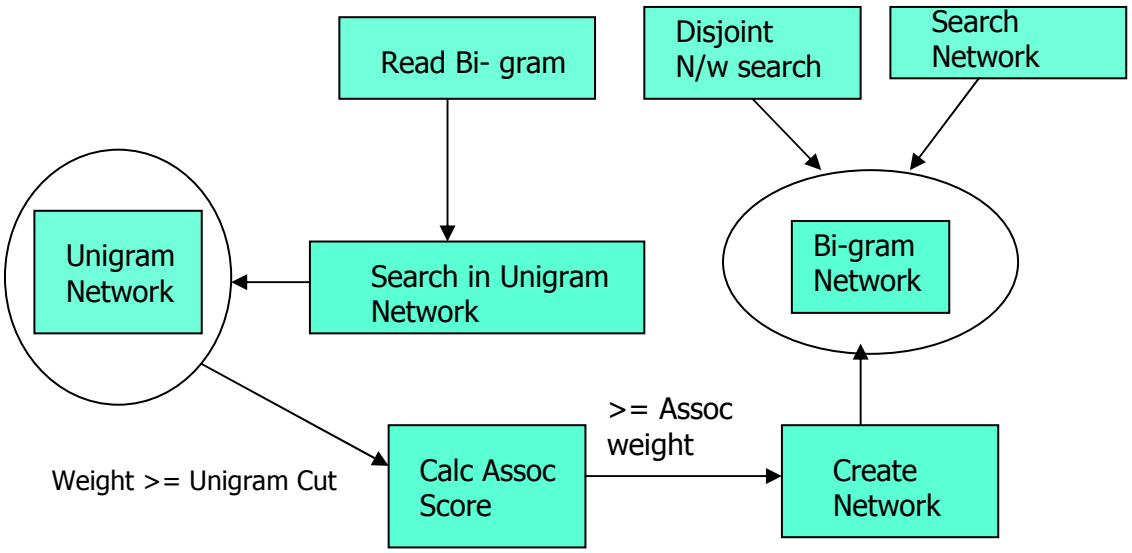
Whatever data is passed to the function, the following operations will take place.

The first word in the bi-gram file line is searched in the root node.

2.3.1

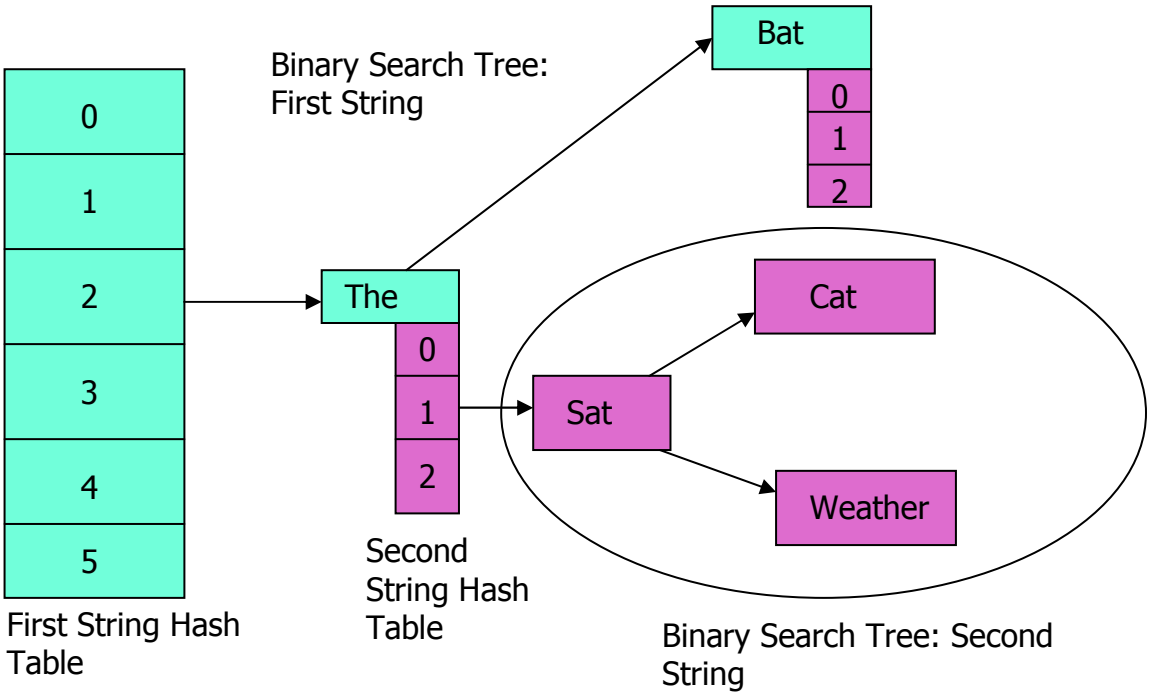
Architecture & Conceptual View

Architecture

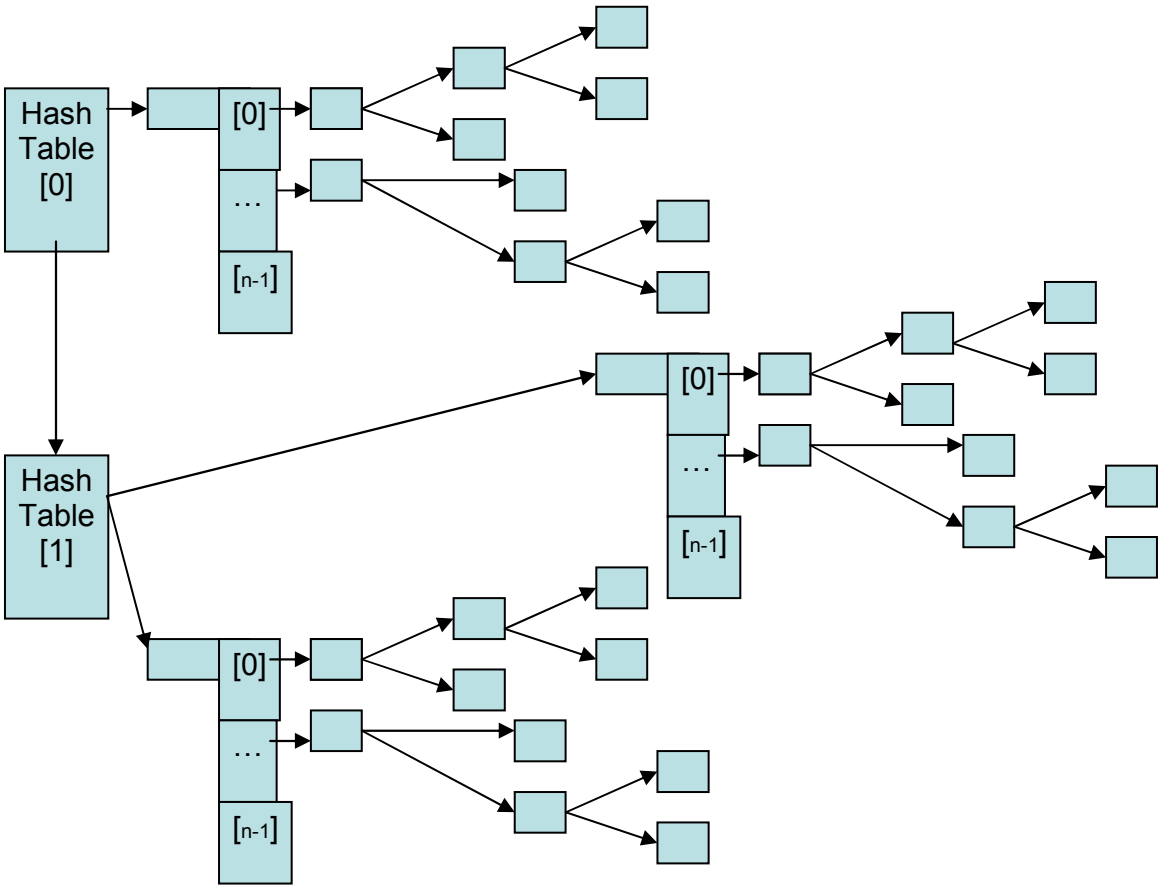


2.3.2

Data Structures used

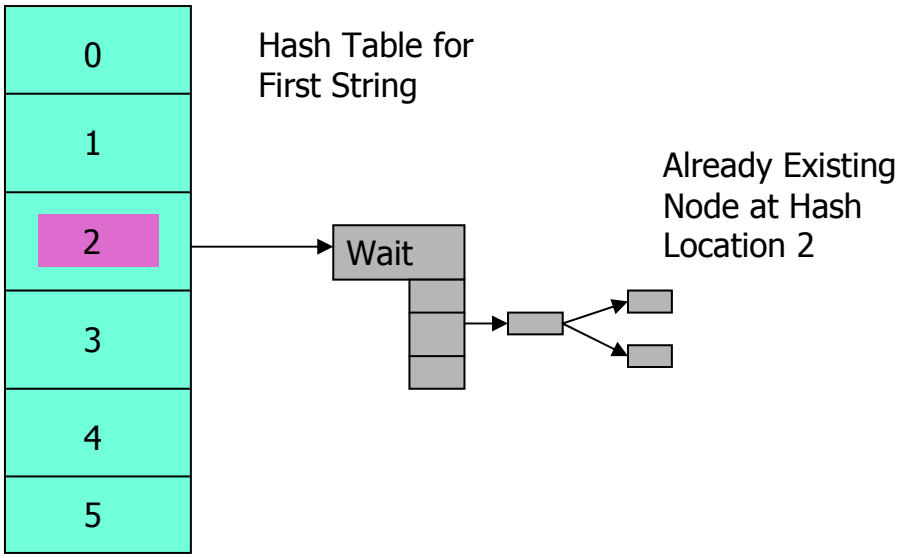


A diagram of the concept used in the base data structure.



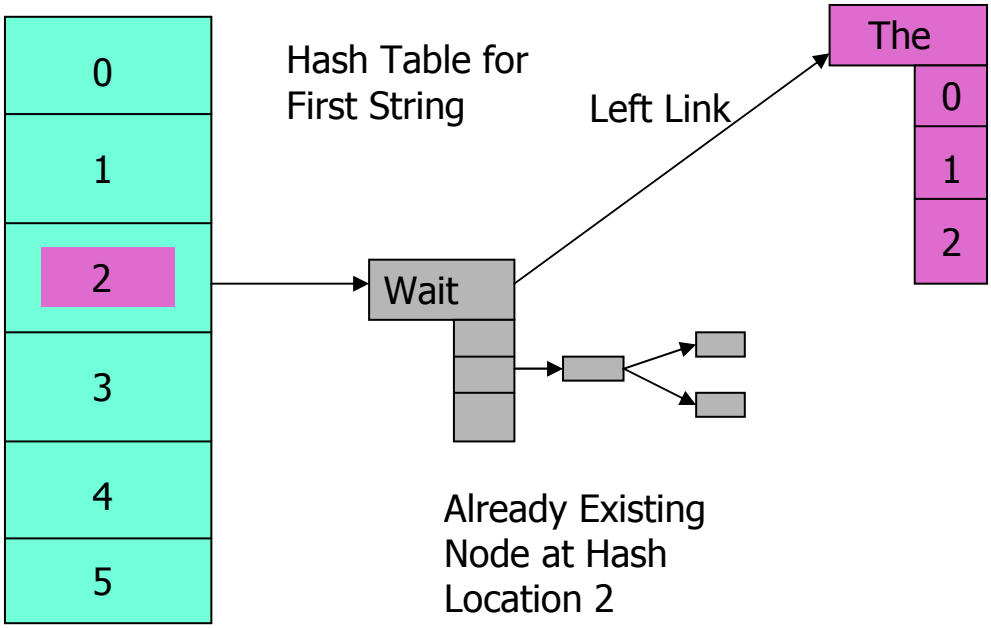
2.3.4

Hash (“The”) = 2

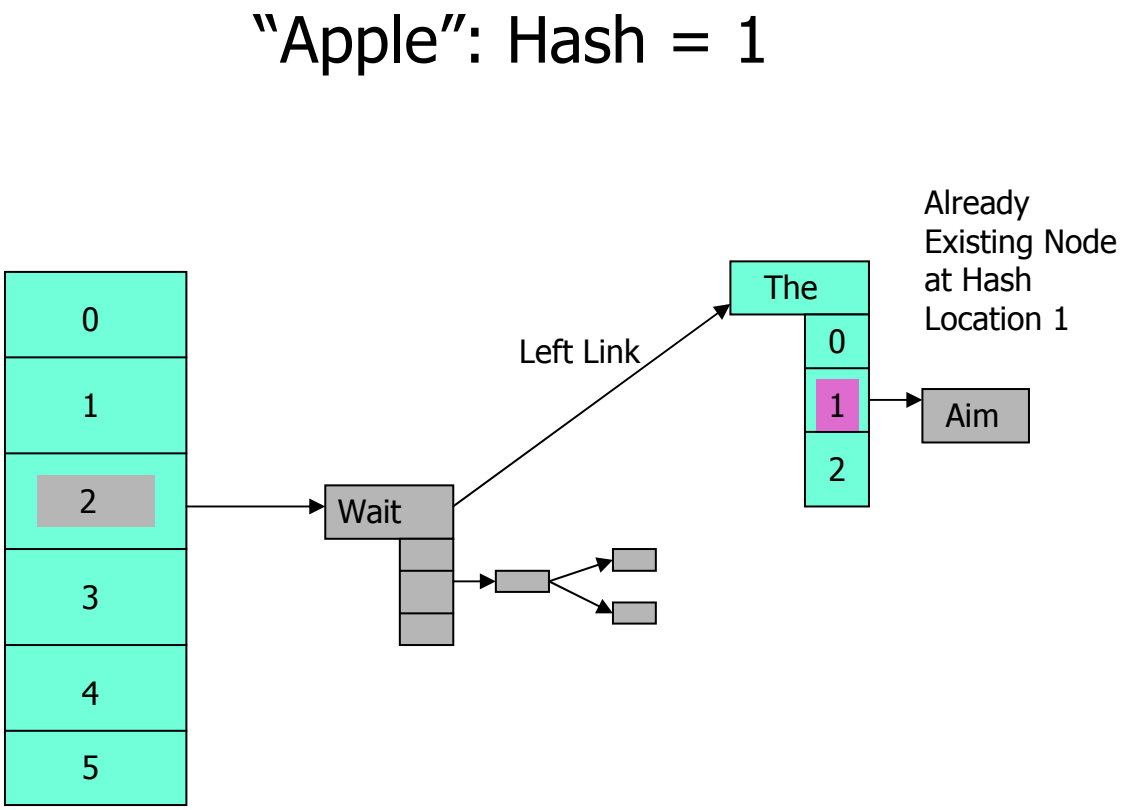


2.3.5

"The": Inserted

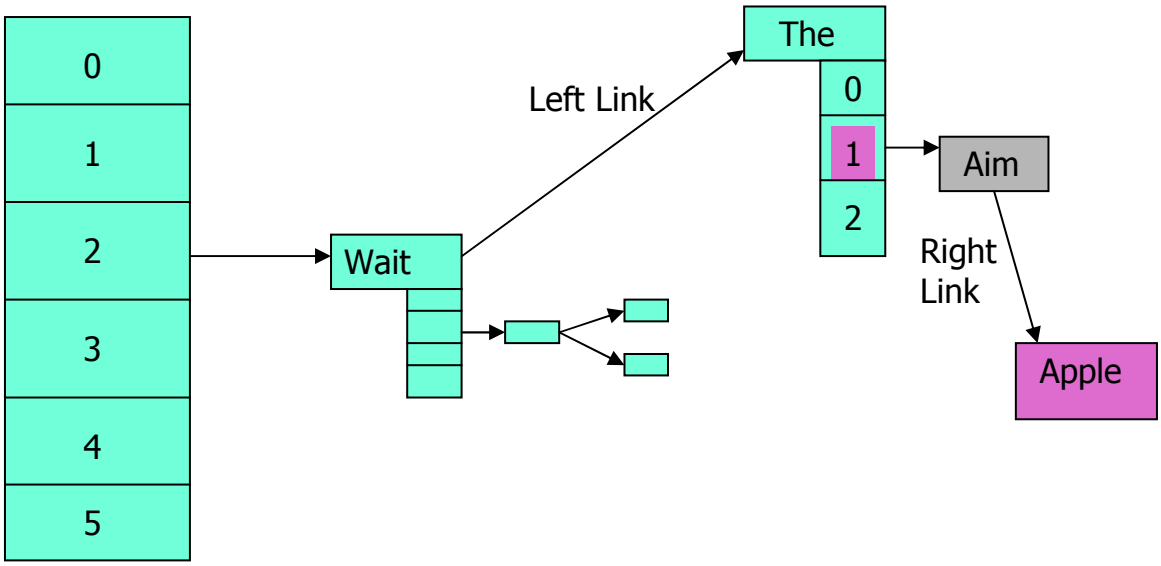


2.3.6


"Apple": Hash = 1
Already Existing Node at Hash Location 1
Left Link
0
1
2
3
4
5
Wait
The
0
1
2
Aim

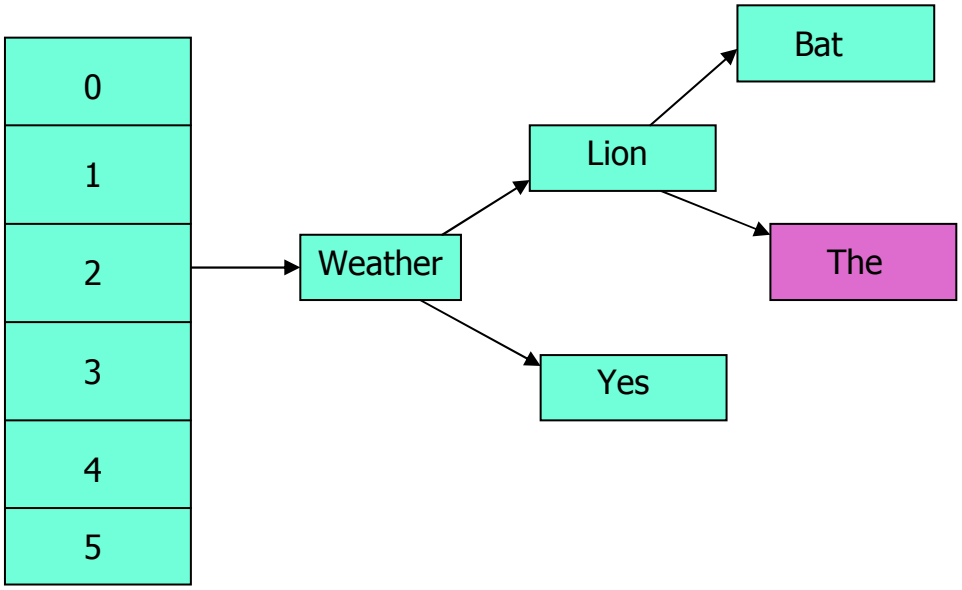
2.3.7

“Apple”: Inserted



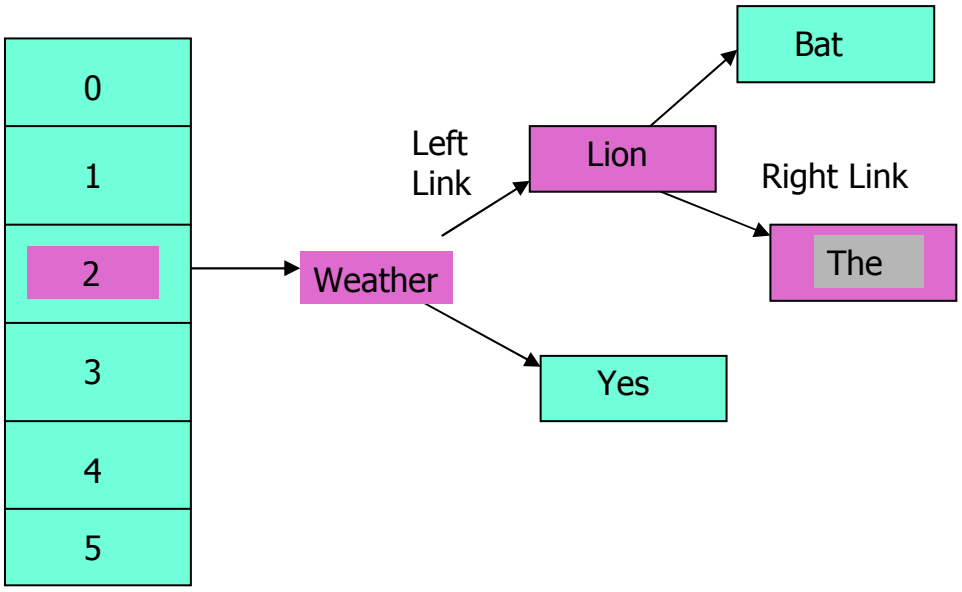
2.3.8

Search Method: “The”

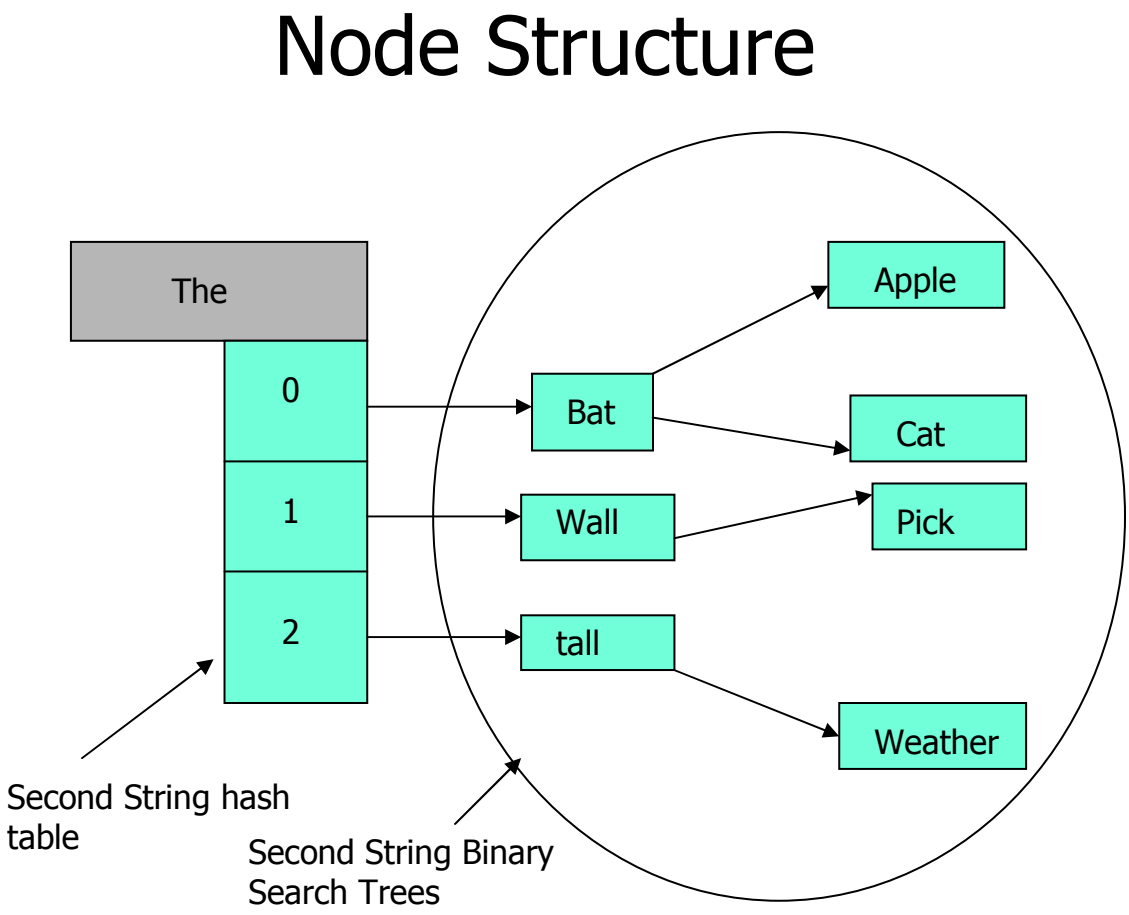


2.3.9

Search Method: “The”

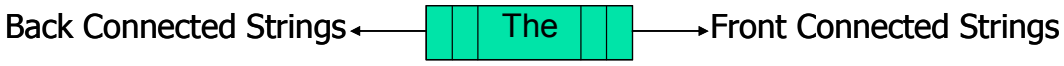


2.3.10

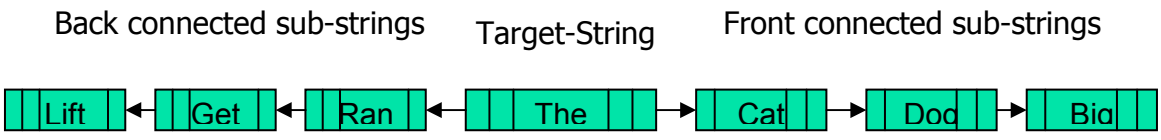

Node Structure
The
0
1
2
Bat
Wall
tall
Apple
Cat
Pick
Weather
Second String hash table
Second String Binary Search Trees

2.1.11

Creating Target-string network



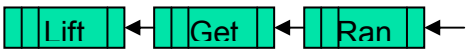
2.1.12



2.1.13

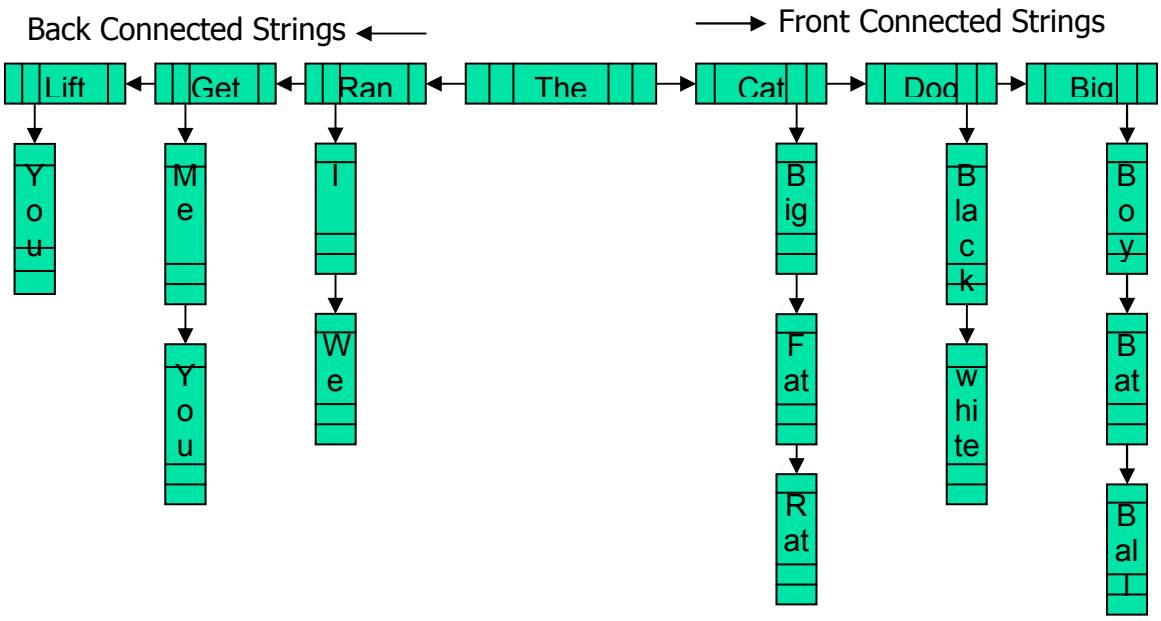
- Passing the substrings connected to the target string to the search function.
- Front Connected Sub-strings to ‘The’

- Back Connected Sub-strings to ‘The’



2.1.14

Search String: ‘The’ Path Length: 2



2.4

In our example, the word Bombay is searched in the root node.

Case 1: The word is not present in the root node.

The first string of the data in the bi-gram line is searched in the root node. The second word on the line is used as input in the second hash function. If the first word is not present as a child of the root node, then it is inserted into the root node. Then the front flag of the root is set to 1 and the back flag is set to 0. The frequency count of the root is set to the weight associated with the line. And the second string of the occurrence node for the received data is set to the second word on the passed line.

Example 1:

Root node does not have any word.

Line passed from bi-gram data file: Bombay Tanvir 3457

The word ‘Bombay’ will be searched in the root node, and when the search returns a null value, it is inserted into the root node. Front flag becomes 1 and back flag becomes 0. The word ‘Tanvir’ is stored as the second string of the occurrence root of the root node.

2.5

Case 2: The string is a word string in the root node's occurrence root, and

a) It is the first string associated with the occurrence root.

Here the word will not show up in the search of the occurrence root.
In that case, an insert operation takes place. The node is inserted either as a left child or right child of the parent node, under the array position returned by hashing the incoming data string.

b) It is another string associated with the occurrence root.

The word shows up in a search of the occurrence root. In that case, just set the front flag of this incoming data string as 1 and frequency on the bi-gram data line as the line weight.

After the first word has been detailed into the tree, the second word string on the bi-gram data file is evaluated.

Now we use a type of reverse operation. The same procedure is performed for the reverse root provided. The second word on the line is searched in the reverse root.

2.6

Case 1: It is not present in the reverse root node.

It is inserted by an insert operation into the node. This time the front flag is set to 0 and the back flag is set to 1. The second string of this reverse root's occurrence root is set to the actual first string on the bi-gram data file line.

Case 2: The string is a word string in the reverse root node's occurrence root, and

a) It is the first string associated with the occurrence root.

Here the word will not show up in the search of the reverse root's occurrence root. In that case, an insert operation takes place. The node is inserted either as a left child or right child of the parent node, under the array position returned by hashing the incoming data string.

b) It is another string associated with the occurrence root.

The word shows up in a search of the occurrence root. In that case, just the front back of this incoming data string is set as 1.

2.7

In the end of the alpha stage, the storing of the network created is handled thus:

The main program passes the locations of the first hash table entries along with the processor IDs. The path of the current working directory is obtained. Then the new directories are created per each processor and named with the processor IDs. Every processor creates one file per hash table entry (named as the hash table entry number). The contents of this file include the entire binary search tree attached to that particular hash table entry.

This convention has been used keeping in mind our initial approach to go about the beta stage and requirements and the input to the same.

2.8

To perform the operation of searching for the target string and any strings occurring in front of that string or behind it, we first have to traverse the co-occurrence file and store the strings that are connected.

The following operations take place to create the data structure for traversing the nodes:

1. Read the target-list file one line at a time.
2. Store the target string and the path-length.
3. Create a data structure for storing all the connected strings to the target string, both in front and back.
4. Pass the target string to the search function which searches all the connected strings to the target string both in front and back and returns back a connected link list of all the strings and the total count of the front and back connected strings.
5. Data structure is created for storing all the connected string, and then traverse the data structure one node at a time and pass the substrings connected to the target string to the search function.
6. This again gives back all the connected strings to the substring passed to the search function in the form of a connected link-list. This process is done for both the front and back connected substrings to the target string.
7. The process of passing the strings to the search function is limited by the path-length. Once the path-length is reached we have the connected string structure of the target

string, both in front and back directions. This structure spawns to the specified path-length.

8. Now I pass the address of the target string to the Display function which recursively prints the entire structure of substrings connected to the target string, both in front and back directions along with there frequency count.

1. Open the target-list file to read the target-strings and path lengths one at a time.

2. Passing the target-file to the “Read_display_connected_strings”.

3. Passing the target-string to the search function which returns all the substrings connected to it in the front and back directions along with the count of the number of front connected strings and number of back connected strings.

4. Using the path-length as the limit to create the target-string and connected-strings network, both in the front and back directions.

5. Once the target-string network is created the start node is passed to the display network which prints the target-string and all its substrings in the front and back directions along with there frequency count. Since we have created the network keeping the path-length limit, we just need to print the entire target-string network which will never exceed the path-length. This is one advantage of creating a target-string network.

6. Once the printing is done, flushing all the memory used for creating the target string network and taking new target-string and path-length as input from the target-list file and repeat step 1-4.

2.9

When given a string, and a certain path length, we search the forward and backward paths from that particular string and return the tree strutures of specified pathlength for each string in the forward and backward connected components of the original input string.

The input given to this phase is the string for which front and back connections need to be searched (and the specified pathlength).
The output from this phase would be:
list of strings containing the front connected strings, when forward connections are requested and/or list of strings containing the back connected strings when backward connections are requested.

This program searches the previously created hash table and binary search tree used to create the co-occurrence network.

Algorithm:

1. Hash the input string with the hashing function used for creation of the network
2. Search the input string in the binary search tree root at the hash location
3. Once the string is found,traverse the connected strings hash table and
   binary search tree.
4. Inorder traversal is used for the traversal and it adds the connected strings in the
   linked list structure to be returned.

As the network is distributed among the processors, this function also requires to gather the connected strings on other processors. For this it sends the input string to other processors.

Other processors implement the same algorithm and send the connected strings to this parent process using MPI communication.

When searching for the particular string among the co-occurrence nodes of other processes, if the front flag is set , we search the front connections of the input strings, otherwise, the back connections are traversed.

2.10

In the beta stage, we would check for the existence of disjoint networks within the data corpus. The disjoint network files are generated based on the number of disjoint networks found within the test data corpus.

Different processors create disjoint network files based on the data which they have worked on. The convention used is that we would have 2 types of files for the disjoint networks. The first file would be the actual disjoint network nodes file. In such a file, each separate node would be put on a new line. This file would have only nodes and not any figures related to the number of nodes or edges.

The second type of file would be the “stat” file. This file would contain the number of nodes and edges in the disjoint networks. It has only 2 figures placed on new lines. The first number is the number of nodes and the second number is the number of edges. Now, in order to the calculate the total number of disjoint networks among all the processors, we would have to check if any word occurs in common between any of the multiple number of disjoint network files.

The naming convention for the “nodes” and “stat” file is as follows:
Both files will have the first 3 characters of the name as numbers and the last 3 characters also as numbers. The first 3 numbers would be the processor that generated that particular disjoint network nodes and corresponding ‘stat’ file.

The last 3 numbers would be the actual file number to differentiate how many files are generated by each processor.

Examples:

1. 000network000, 000stat000: this means that the files were generated by process 0 and it is the first disjoint network node and ‘stat’ file.
2. 000network002, 000stat002: this means that the files were generated by process 0 and it is the third disjoint network node and ‘stat’ file.
3. 001network001, 001stat001: this means that the file was generated by process 1 and it is the second disjoint network node and ‘stat’ file.

For the processes and file numbers, our numbering index would start at zero.
The procedure is accomplished as follows: first we traverse each “stat” file and storing the number of nodes and edges occurring in the corresponding “network nodes” file.
A total network count variable records the total number of networks initially when all the processes create any disjoint networks that existed in their share of the data.

At the end of the disjoint network traversal stage, the total number of disjoint networks would be calculated among all the processes.
A simple algorithm used to calculate the total number of disjoint networks is as follows

1. Each file is compared, string (node) for string (node), to check if any node occurs in common among those files.

2. If at all any node is in common, that means those particular disjoint networks and in fact connected and not disjoint.

3. Once that happens, the total count of disjoint networks would reduce.

4. Then, the number of edges from each disjoint network would have to be added to get the total number of edges in the newly combined disjoint network.

5. The total number of nodes in the newly combined, larger disjoint network would be the sum of nodes from both the smaller disjoint networks minus the one node that occurred in common.

When this algorithm is tried, the output would be the total number of disjoint networks along with the final count of nodes and edges among the individual disjoint networks.

2.11

The unigram cut operation: This operation has been specifically optimized to run on only 2 processors in order to better utilize the memory resources while the other procedures are executing.

The operation will proceed as follows:

The ordered unigram file “vocab’ is used for this procedure. The file is cut into 2 parts and each part of the file is passed to one process. The hash function is used and for each value of the hash table, a binary tree is created for the particular string whose hash value corresponds to the hash table array. Using the hash function optimizes memory utilization and also speeds up the procedure considerably. A string is passed on the procedure that is supposed to search that particular string for occurrence in the unigram file. This search string is then passed onto the 2 separate processes.

The string is searched in both the processes and if the string is found the unigram weight is returned otherwise ‘-1’ is returned, indicating that the input search string is not found. Association cut-off takes the weights returned by the unigram search function and takes the bigram weight from the bigram file read function and calculates the association scores.

3. Resources

Hardware:

Blade cluster provided by the Minnesota Supercomputing Institute.

Processor type: Two dual-core 2.6 GHz AMD Opteron processors per node.

Memory: 8 GB per node (7 GB usable)

Software:

The Blade cluster uses the 2.6.5-7.244-smp version of the GNU/Linux operating system.

Language and Libraries:

The entire code has been written in the C programming language. The only libraries used are the MPI (Message Passing Interface) library and the OpenMP library for use in parallel programming.

4. Performance

4.1

Timing in seconds: Alpha Stage readings

	Processors	16	8
File Numbers			
0			93.65
1			63.945
2			16.98
3			16.19
4			18.62
5			88.4
6			260.129
7			11.85
8			11.6
9			11.169
10		33.015	11.51
11		8.968	11.76
12		12.43	11.39
13		13.73	
14		13.66	
15		24.26	
16		12.91	
17		12.92	
18		9.6	
19		8.23441	
20		11.108004	
21		8.842743	
22		8.198028	
23		8.489847	
24		9.076095	
25		8.447796	
26		10.484789	
27		8.056134	
28		7.860083	
29		11.223973	
30		9.218522	
31		4.758889	

We were able to test the code on differing number of single input files and also on the entire bi-gram data set. Due to the *bc* queue being busy during the writing of this section, not all the results could be posted.

Beta Stage Results

Unigram Cut off Network Creation Timing:

18.057747 seconds for the unigram cut off limit of 200

Unigram Cut off Network Search Time per string:

0.499875 seconds

Searching a target string for front and back connections to a user specified path:

1.667915 seconds (searching a string of specified length as 2)

4.2 Benchmarking:

End to End timing results

4.2.1 Network Creation & File I/O

Number of 2gram-files read	Number of 2gram-files read	Number of Processors (Execution Time in sec)
1	12(40.586217)	16(35.940412)
2	12(82.321112)	16(49.712990)
3	12(82.321112)	16(59.467261)
4	12(119.915033)	16(76.846659)
5q	15(125.675043)	18(109.447602)
29	64(975.952314)	100(1231.427653)
30	64(1015.643072)	100(1289.246602)
31	64(1064.134553)	100(1338.421798)

4.2.2 Searching, Display & File I/O

Number of Processors	Number of Files	Target String	Path Length	Searching+Display time(sec)
10	5	‘Plagiarism’	2	568.124860
16	5	‘Plagiarism’	2	102.300228
10	5	‘HELLO’	2	779.989307
16	5	‘HELLO’	2	34.778504
64	26	‘ubiquitous’	2	141.253852

4.3 Memory Requirements:

Network Creation: Memory Requirements per processor

Structures Used	*Bytes Required*
1. One line record	106
2. Occurrence node	77
3. Node	1213
4. Connected strings	36

Memory Calculations:

Hash table memory	= 1000 * 4	= 4000 bytes
Unique Node (Reverse)	= 5000000 * 1213	= 6065000000 bytes
Max Unique Nodes (Per File)	= 50000*1213	= 60650000 bytes
Occurrence Nodes	= 10000000 * 33	= 330000000 bytes
One Line Record	= 10000000 * 106	= 1060000000 bytes
Total	= 7545954000 bytes	

Total Maximum Memory Required Per Processor = 7.02772 GB

Unigram Creation: Memory Requirements

Structures Used	*Bytes Required*
1. Unigram-node	57

Memory Calculations:

Hash table memory	= 50000 * 4	= 200000 bytes
Node Memory	= 57 * 13588391	= 774538287 bytes
Extra Allocation	= 150 + 41 + 8	= 199 bytes
Total	= 774738486 bytes	

Total Maximum Memory Required = 0.774 GB

5. Testing Methodologies:

In the early development stage, we considered the SGI Altix cluster as another option, however the queue seemed to slow on that system too.

We attempted to move the code to the Calhoun cluster on the MSI systems but it was extremely slow. Also it has only a single job submittal queue and hence we decided to opt out of working on the Calhoun cluster.

In the beta stage, we have performed extensive testing on experimental data and in most cases, concrete conclusions could be drawn from trial data and single files of test 2 gram data.

Due to all the constraints mentioned above, we ran all our tests on the IBM Blade cluster.

Initial runs of our code produced a very persistent memory allocation error that was somehow corrupting the memory chunks of the run time heap. After steady debugging and making slight modifications in the way we handled pointers and certain data structures, the errors were resolved. Some files had to be broken down into multiple functions stored in separate C files.

The initial method the team used was a brute force method for time bounds testing for the sequential version of the code. However, what ended up being produced were errors in

reading the largest files of the data set. We kept varying the size of the first hash table and second hash table. This was showing us that varying amounts of lines were read depending on the sizes of the hash table.

We initially tried the sequential program on the largest file of the bi-gram collection. This was the “2gm-0000” data file. All the other files of the bi-gram data collection, except “2gm-0000”, “2gm-0001” and “2gm-0006” were stopping halfway through the program execution. The most common error was due to memory corruption.

6. Conclusion

As of this stage, our team is able to read the entire bi-gram data set and push the data into files. We are able to perform sequential as well as parallel file input output operations. However, some modifications are being performed as we are still facing some errors and irregularities in the file output. One reason could be the *bc* queue returning inconsistent output or no output when jobs are submitted and unusually high wait times.

7. Acknowledgments:

We are grateful to Dr. Ted Pedersen for providing us the opportunity to work on this project.

Thorsten Brants, Alex Franz.

(Google Research)

For providing documentation related to the data and its specifications.

For the purpose of dealing with this problem and a possible approach we read the following 3 published papers:

8. Bibliography

Papers:

Yoshiki Niwa and Yoshihiko Nitta. “Co-occurrence Vectors from Corpora v/s Distance Vectors from Dictionaries.” *15th International Conference on Computational Linguistics (COLING) 1994, Kyoto, Japan.*

Philip Edmonds. “Choosing the Word Most Typical in Context Using a Lexical Co-occurrence Network.” *Meeting of the Association for Computational Linguistics,(ACL) 1997, Madrid, Spain.*

Oliver Ferret. “Discovering Word Senses from a Network of Lexical Co-occurrences”. *20th International conference on Computational Linguistics, (COLING) 2004, Geneva Switzerland.*

Dan Higinbotham. “Semantic Co occurrence Networks”. *The Third International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages*, *(TMI), 1990, Linguistics Research Center University of Texas at Austin*.

Anne Veling, Peter van der Weerd. “Conceptual Grouping in Word Co-occurrence Networks”. *Sixteenth International Joint Conference on Artificial Intelligence, Stockholm, Sweden, 1999.*

Textbooks:

"Parallel Programming in C with MPI and OpenMP".

Michael J. Quinn.

McGraw-Hill Science/Engineering/Math, 1st edition, 2003.

"Introduction to Algorithms".

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.

The MIT Press, 2nd edition, 2001.

"The C Programming Language".

Brian W. Kernighan and Dennis M. Ritchie.

Prentice Hall PTR, 2nd edition, 1988.