Indexing The Hansard XML Data Using Bitmaps

School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg 2050

Ntladi Mohajane-1599953, **Phetolo Malele**-1412331, **Zwavhudi Mulelu**-1110574

***Abstract*—This paper presents the design, implementation and analysis of a parallel program that calculates indexes for a given Hansard XML dataset. The bitmap technique is used to calculate these indices. Furthermore, the algorithms designed for parsing, indexing and searching the provided Hansard dataset are created using C++ and the OpenMP parallel processing library. Performance of the implemented parallel program is then tested and analysed to assess its performance. The obtained execution time results showed a poor performance with an increase in threads. Finally, future recommendations are therefore provided in order to improve the performance of the implemented program.**

I. Introduction

With the world moving towards digitization, hard-copy records such as debates in the Parliament of the Republic of South Africa (Hansards) are being transcribed into digital formats. This is done due to the increasing prevalence of internet-based access to information or records. The Hansard dataset is provided in the XML (Extensible Markup Language) format and it has a high information-density which makes searching the full dataset complex. It is therefore crucial to perform indexing on the dataset in order to provide a pre-processed representation of data which can be easily computed on. This thus enhances the performance of a search program.

This paper presents a design, implementation and analysis of a parallel program that uses the bitmap technique to calculate indexes for the provided Hansard data.

II. Background

Bitmaps are used in a variety of applications including data warehousing such as online analytic processing (OLAP) [1]. Bitmap indexing produces a bitmap index which is a collection of bit vectors generated for every distinct value in the indexed column. The amount of bit vectors in a bitmap index for a given indexed column is referred to as cardinality - the number of distinct values that appear in the indexed column. In order to obtain effective results for both low and high cardinality attributes, bitmaps are stored in a compressed form [1].

A. Byte-aligned Bitmap Code (BBC)

This is one of the two most popular schemes to compress a bitmap, known for its good compression and how it allows fast logical bit-wise operations [1]. The algorithm is based on the simple idea of run-length encoding [2]. A bitmap is first divided into bytes (hence the name), then the bytes are grouped into runs [2]. Each run then has a fill which is followed by literal bytes [3, 2]. The issue with this compressing scheme is that, though it offers fast logical operations and effective compressing, Answering queries is usually slower because most of the time is spent on the CPU [1, 2].

III. Constraints

The following limitations or constraints exist for this project:

- The group has elected to use C++ as an implementation language - this language is chosen because it has the benefit of fast run-time performance compared to interpreted languages such as Python. However, the compiled files are less portable as a result and may need to be recompiled from machine to machine.
- The group has elected to use OpenMP as a parallel processing library. This benefits the group because it abstracts away the low-level thread control, thus allowing to focus on the implementation itself. However, the group loses the opportunity for more granular optimization.
- The group has decided to only consider the Hansard file that accompanied the project brief. In doing so, the group can focus on optimizing for this specific dataset and therefore it does not have to add additional code in the parsing and indexing implementations in order to account for more general cases.
- The group has elected not to make use of the distributed-computing paradigm - this allows the group to test the implementation regardless of whether or not they have access to a multiprocessing computer cluster.

IV. Program overview

The program is separated into three sections, mainly the parser, indexer and the searcher as illustrated in Figure 1. These three sections run serially in the order stated. However, each section has subsections that run in parallel with one another.

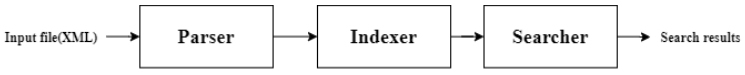

Figure 1. A representation of the program overview

The parser uses a library called PugiXML to format the data in a tree structure which is more easy to traverse through using linked lists. The indexer formats the parsed data in the form of a two dimensional (2D) Boolean array. This array is then exported to a binary file which represents a bitmap.

Finally, the searcher is able to load the bitmap into memory and reconstruct the 2D Boolean array. Furthermore, the searcher also uses maps from the indexer in order to identify which row and column represents which speaker and debate respectively. The searcher then uses this information to traverse through the bitmap and perform bit-wise operations on two

different rows or two different columns to determine which speakers appear in which debates and vice versa.

V. XML Parsing

A. PugiXML Library

The provided Hansard XML dataset has a relatively high-information density, most of which is not required in order to search for debates or speakers. As such, only the relevant data is extracted from the Hansard dataset. In order to extract the relevant information (debates and speakers) from the XML file, the PugiXML open-source parser library is used. By letting an external library handle parsing, the group reduced the overall amount of work required to implement the solution. Furthermore, the reliability of the program also increases because the library has been thoroughly tested by those with more time and experience. The parser section is broken up into the parsing debates and parsing speakers subsections. These sections each make use of parallel programming techniques to achieve their respective goals.

B. XML File Structure

Figure 2 shows the assumed structure of the XML file that is processed. This figure ignores several of the debate body sibling nodes because the group decided that they do not contain any relevant information. Each of debate body's child nodes represents a single debate. Each of these debates contains additional debate section child nodes. However, the structure of these nodes is inconsistent and sorting these nodes would add complexity to the parsing section. Furthermore, debate body's child nodes all have headers which can be easily used to identify debates by the date on which they occurred.

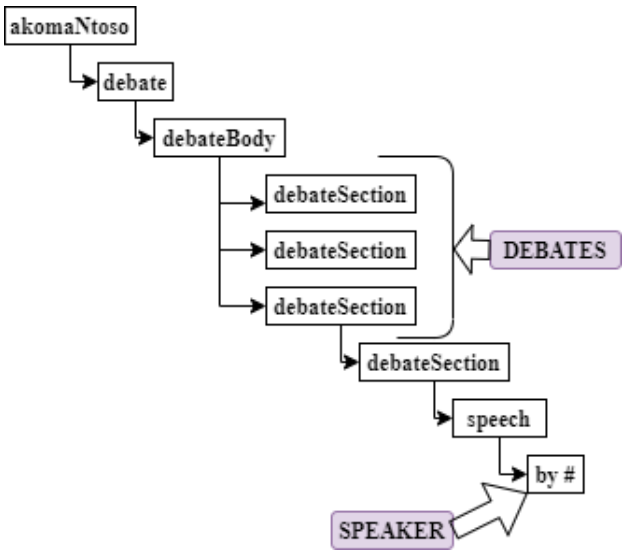

Figure 2. Assumed XML file structure

Each of the debate section nodes has additional child nodes which are also called debate section. However, not all of these nodes have speech child nodes. Therefore, the group has decided that a debate is only valid if it has at least one speech node as a child. Within each speech node is a tag labeled '#by' which is what the group uses to identify speakers in a debate.

C. Parsing debates

The XML tree is stored in the form of a linked list data structure. Traversing this data structure in parallel requires the address of the previous node in order to access the next node. This dependency makes concurrency difficult. However, OpenMP library provides a mean to solve this issue in the form of tasks. Tasks allow a single thread to allocate work to multiple threads before all the threads collapse back into a single thread. This process is illustrated in Figure 3 where *i* represents a debate that is being traversed through and *f*, a task being executed.

The tasks begin by finding all the debate section nodes within a specific debate and then adding the debate to a vector which represents a list of valid debates. The pseudo-code for the process is shown in Algorithm 1 and Algorithm 2. Furthermore, adding a debate to the list is done in the critical section. This prevents race conditions at the slight cost of performance. However, because the parsing of the debates happens in parallel, the order of the debates gets shuffled with each execution of the program.

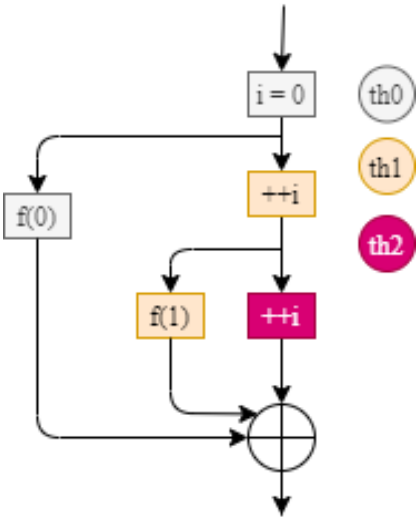

Figure 3. The process of parsing debates in parallel

Algorithm 1: `parseDebatesParallel`

```
1 #pragma omp parallel
2 #pragma omp single
3 debate = debateParentNode.first_child()
4 populateDebates(debate)
```

D. Parsing speakers

With the debates now stored in a vector container, their indices can now be accessed via the array index. This allows the group to parallelise simply by using for-loops rather than the more complex method described in Figure 3. The iterations are not uniform because different debates contain a different number of speakers to process and therefore the threads are dynamically allocated to each iteration of the for-loop.

Algorithm 2: `populateDebates`

```
input: debate
while traversing through debates do
    #pragma omp task
    find all debateSection children
    for child in children do
        if child == "speech" then
            store debate into vector
            #pragma omp critical
            debateChildNodes.push_back(debate) break
debate = debate.next_sibling("debateSection")
```

Therefore using dynamic allocation allows for the uniform distribution of work between threads.

Each thread looks for every speech in a specific debate and finds every speaker tag within those debates as illustrated in Algorithm 3. The thread then filters out all duplicates and stores the results in an array which is then stored in a larger multi-dimensional array. Furthermore, a map is used to link the inner arrays to their corresponding debates. This process is done within a critical block to ensure that threads will do the mapping to the correct debate one at a time. The parser also returns a list of unique speakers in order to make future processing easier.

Algorithm 3: `parseSpeakerParallel`

```
#pragma omp parallel for schedule(dynamic)
for i ← 0 to debateNodes.size() do
    access debates in the vector of pointers
    allSpeakers = organizeSpeakers(speech);
    #pragma omp critical
    add speakers to the speakers array
    Map speakers to specific debates
```

VI. Bitmap Indexing

The indexer section processes the data from the parser in order to generate a two dimensional bitmap. The tabular structure of the bitmap is illustrated in Table I. .This bitmap is stored in a 2D Boolean array in memory where the rows represent a speaker and the columns represent a debate. If a speaker appears in a specific debate, the corresponding index is marked as true, otherwise it is marked as false. By using an array of Booleans, the data can be stored with a reasonable degree of space efficiency because Booleans only take up 1 byte of memory per Boolean. This array is then exported to a binary file for future use and the size of the binary file is directly proportional to the size of the array.

Populating the file is done using a parallel for-loop that uses static thread allocation. Static thread allocation is used because the size of the buffer that each thread must process is uniform and can be divided into equal chunks. The process of how the binary file is written to is described by Algorithm 4.

Table I
Bitmap tabular structure

	debate 0	debate 1	debate 2
speaker 0			
speaker 1			
speaker 2			

Furthermore, the indexing section also returns a map of key-value pairs for all speakers and debates as well as the name of the output binary file, all of this information is returned in the form of a struct that will be processed in the searcher section.

Algorithm 4: `exportData`

```
input: fileName
Initialize output binary file using ofstream object
for i ← 0 to speakers.size() do
    bool buffer[data.size()]
    #pragma omp parallel for schedule(static)
    for j ← 0 to debates.size() do
        buffer[j] = index[i][j]
        write to output binary file
close output file
```

VII. Bitmap Searching

A. Loading the bitmap from the binary file

The structure of the bitmap in this section is the same as the one described in Table I, in that it is also a 2D Boolean array that stores indices that associate speakers with specific debates. Furthermore, the process in which the bitmap is loaded in memory is similar to how the indexer exports it. However, an *ifstream* is used instead of an *ofstream*. Moreover, when initializing a searcher object, it takes in the struct generated by the indexer as a a parameter. This allows the searcher object to identify the location of the exported bitmap and it also allows the size of the keys to be used as limits to populating the Boolean array.

B. Searching for debates

The speaker key-value pairs map each speaker tag as a key to a unique integer value. This value represents the row index of where the speaker is located on the bitmap. Therefore, in order to determine if two speakers are present in a debate, a function is written that accepts two speaker tags as an argument. These two arguments are checked against the list of keys and if those keys are present, the map returns the row indices of both speakers. These indices will be used to constrain the search to be between these rows. These rows are then traversed in parallel and a bit-wise operation is performed for each index. If the bit-wise operation evaluates to true, it represents that the two speakers are present in that debate and the debate is then printed out onto the console. This continues for all debates in those two rows. Algorithm 5 describes the process of how this is done.

Algorithm 5: `findDebates`

```
input: speaker1Tag
input: speaker2Tag
speaker1Index = speakers[speaker1Tag]
speaker2Index = speakers[speaker2Tag]
#pragma omp parallel for schedule(static)
for i ← 0 to debates.size() do
    if index[speaker1Index][i] and
      index[speaker2Index][i] then
        #pragma omp critical(cout)
        print the debate index
```

C. Searching for Speakers

Algorithm 6 describes the process of how all the speakers that participated in two debates can be found. The function takes in two debate column indices. In contrast to how debates are found, the indices act as keys to the map of debates. Furthermore, these indices will restrict the search algorithm to only look for speakers within these two columns. A bit-wise operation is performed much like when finding debates and if the expression evaluates to true, the index of the speaker will be printed to the screen.

Algorithm 6: `findSpeakers`

```
input: debate1Index
input: debate2Index
#pragma omp parallel for schedule(static)
for i ← 0 to speakers.size() do
    if index[i][debate1Index] and
      index[i][debate2Index] then
        #pragma omp critical(cout)
        print the speaker index
```

VIII. Results and Analysis

The program was bench-marked on a machine using the following specifications:

- Intel Core i7-6700 CPU @ 3.4GHz (4 CPU Cores, 8 Logical Threads due to HyperThreading).
- 8.00 GB 2133MHz Dual Channel DDR4 RAM
- Linux Ubuntu 18.04 LTS (64-bit)

A. Output

An example of the program's output after execution can be found in the *Appendix* . The program produced the correct output after every execution. Within the context of this program, a correct output is defined as the program returning the correct debates when querying for two different speakers, as well as returning the correct speakers when querying for two different debates.

The group noticed that although the rows of the bitmap were consistent with every run, the columns were not. This is because populating the bitmap is done in a nested for-loop and only the outer loop is parallelised. This means that it is not guaranteed that the columns will populate sequentially due to thread scheduling. The columns represent the debates, therefore a map for the debates must be exported whenever a new bitmap is generated.

The size of the bitmap in memory is exactly the same as the size of the exported file. The dataset used yielded 80 unique speakers from 21 debates and the size of the exported file is 1680 bytes. This means that every index occupies 1 byte in memory which corresponds to the size of a Boolean in Linux. However, the file takes up 4KB on disk because that is the default block size for the operating system.

B. Performance Results

Figure 4 shows the performance results of the program when given a variety of thread sizes.

The results show that the program scales poorly with an increasing number of threads. One possible reason for these results is that the data set used is relatively small at only 1680 indices, which magnifies the contribution of the overhead when initializing the threads.

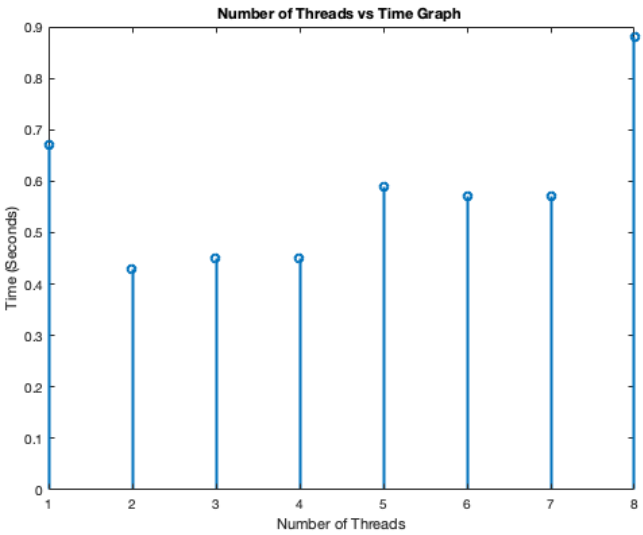

Figure 4. Performance results of the program with different thread sizes

There are many instances in the source code where *#pragma* parallel blocks are initialized. The group made the decision to design the program this way because it allowed them to test each parallel section individually, thus simplifying debugging. However, this comes with the tradeoff of introducing additional overhead into the program because threads are constantly being initialised and reinitialised.

IX. Recommendations

In order to improve the performance of the implemented bitmap indexing scheme, the group recommends the use of the WAH (word-aligned hybrid code) bitmap compression . WAH compressed indices have reasonable sizes that allow for a more

efficient way of answering queries [5]. WAH is also a CPU-friendly scheme. In comparison with the BBC compression scheme, a WAH performs logical operations approximately 12 times faster and it uses only 60% more space [5]. WAH also shows an improvement in the query processing speed [5].

Furthermore, it is recommended to store the bitmap indices in a bitset rather than a 2D binary array when the bitmap is loaded into memory. This is because a bitset is more space efficient because it compresses the bits such that they only occupy 1 bit each.

The program can be tested using different data sets. This will improve the reliability of the program because it will allow the group to test and account for more edge cases. Moreover, the program can be tested using larger data sets because this will minimise the effect of OpenMP's overhead when initialising the threads. Furthermore, other parallel programming environments such as PThreads can be explored because they allow access to lower level functionality and thus allow more chances for optimisation.

X. CONCLUSION

In conclusion, the design, implementation and analysis of a parallel program that computes indexes for the Hansards dataset using bitmaps is presented. This program makes use of the OpenMP parallel processing library in all the steps of the designed system; the parser, indexer and searcher. The parser step generates a pre-processed dataset that makes it easier and more efficient to index and search. The indexer computes indexes of the pre-processed dataset of debates and speakers, the bitmap indexes are then stored into a binary file. The searcher then makes use of the indexer's generated binary file to perform a search for all debates wherein two specific speakers participated or all speakers that participated in two specific debates. The implemented program is also tested and analysed and it was observed that it scales poorly with an increase of threads due to an overhead that arises when initializing threads.

XI. REFERENCES

[1] Kim S, Lee J, Satti SR, Moon B. SBH: Super byte-aligned hybrid bitmap compression. Information Systems. 2016 Dec 1;62:155-68.

[2] Wu, K., Otoo, E.J. and Shoshani, A., 2004. An efficient compression scheme for bitmap indices.

[3] website r. pugixml 1.10 manual [Internet]. Pugixml.org. 2020 [cited 20 June 2020]. Available from: https://pugixml.org/docs/manual.html

[4] Tousimojarad A, Vanderbauwhede W. Efficient Parallel Linked List Processing. Parallel Computing: On the Road to Exascale. 2016 Apr 28;27:295.

[5] Wu K, Otoo EJ, Shoshani A. Compressing bitmap indexes for faster search operations. InProceedings 14th International Conference on Scientific and Statistical Database Management 2002 Jul 24 (pp. 99-108). IEEE.

APPENDIX

This appendix illustrates a typical simplified output from the program and it is read from the left column going down. The program begins by listing the number of unique speakers followed by the number of debates. It then prints the bitmap that is exported to the output.bin file to the console for review. It then lists the key value pairs for both speakers as well as the debates. It then performs a search on the bitmap to find the debates that two speakers participated in. Finally, it performs a search to find all the speakers that participated in two given debates. The results of this last search vary from execution to execution, because the columns are indexed concurrently and their layout may vary due to thread scheduling.

Speakers: 80

Debates: 21

Bitmap: // Only the first 10 rows are shown

00000010101000001000

00000011100110001100

00101000001011100001

11110011101010010100

01000100000000110100

00001110000001000000

11101100011111011000

11100110111110000110000

00000000000000100000

00100010000110110000

Speaker Key-Value pairs: // Only the first 10 are shown

#acting-chairman = row 0

#acting-president = row 1

#berg = row 2

#bozas = row 3

#carr = row 4

#chairman = row 5

#crook = row 6

#dempsey = row 7

#deputy-chairman = row 8

#deputy-minister = row 9

Debate Key-Value pairs: //Only the first 10 are shown

col 0 = tuesday, 8 may 1979

col 1 = thursday, 10 may 1979

col 2 = wednesday, 23 may 1979

col 3 = tuesday, 5 june 1979

col 4 = tuesday, 22 may 1979

col 5 = wednesday, 16 may 1979

col 6 = thursday, 17 may 1979

col 7 = thursday, 7 june 1979

col 8 = wednesday, 6 june 1979

col 9 = monday, 11 june 1979

Debates where '#acting-chairman' and '#acting-president' appear:

thursday, 7 june 1979 col = 7

monday, 11 june 1979 col = 9

monday, 18 june 1979 col = 17

Row indeces of the speakers that appered in debates: tuesday, 8 may 1979 and thursday, 10 may 1979

3

6

7