**SOFT354 Report**

**Introduction**

The topic I am investigating for my coursework piece is in relation to text searching through a table within a database system. This project is built upon three main components:

1. Data storage and retrieval within a database system (which includes populating table(s) with data).
2. Searching through a given tables text fields to match a specified word/ phrase.
3. The population of the table with results from the search i.e. if a match has been found or not in a particular record.

In the current industry data storage and manipulation is a rapidly growing area. Relating this to big data, the application described above can be used for analytical purposes. Searching through large datasets to find specific key words/ phrases can later be used to extrapolate useful information with regards to the table in question. This project aims to not be restrictive with regards to the databases or tables(s) used and can easily be altered to accommodate any database/ table structure. The fact that SQL queries are used to create and update tables as well as extract the data required by embedding the queries in the code, means such queries can easily be altered to match the needs of the user. This in combination with the ability to change the search criteria (what word/ phrase to search for) results in a useful application for text searching through large data sets.

In a nutshell, with more detail to come, the base computing involved in this project for both the serial and parallel versions is as follows:

1. Define word/ phrase to be searched for.
2. Check if database and table have been created, if not, create both and populate the table with data.
3. Retrieve all data to be searched through.
4. Search through all data and record if match has been found or not.
5. Update IS\_PRESENT field to show if the search word/ phrase is present or not in the related text.

Given the option of either CUDA/ GPU or MPI I choose to implement MPI. As according to Rouse the definition of MPI is “the message passing interface (MPI) is a standardized means of exchanging messages between multiple computers/ processors running a parallel program across distributed memory.” (Rouse, 2013) In MPI each processor works on a portion of the overall computing problem. MPI works by exchanging data between the processors or nodes which are carrying out the work in parallel on different pieces of data. This results in a highly parallel process which can give huge speedup results when compared to the same serial process.

MPI is not tied to any specific programming languages. It has been designed so it can be used across a variety of different languages including, but not limited to, Java, C#, C/C++ (this case) and more. This flexibility and the fact that data can easily be passed from computer to computer has allowed MPI applications to be implemented in many of the world’s fastest supercomputers.

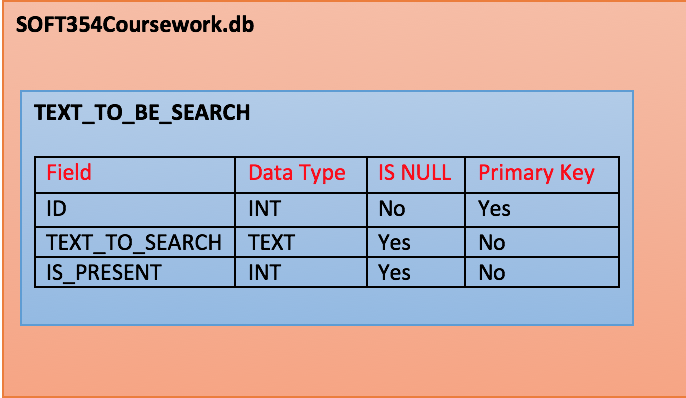
While using processes in MPI is a lot heavier with regards to memory, than when using threads, which results in higher overheads MPI is much more scalable. This can be said due to the fact that, if needed, the work load can be spread across multiple machines whereas when using threads, they all must be on the same machine. This is the reason why MPI was chosen for this project. Due to the fact that a database table is used, it is unclear as to how many records are being stored in the table being searched through. This number could vary from a few hundred to thousands, hundreds of thousands or more. The more records that are present in the table the more computational power is required to search through them all. If the number of records exceeds the number of processes that a single machine can handle, the workload can be spread across multiple machines with even more processes being used. In doing this the efficiency of the program can be maintained while the data set, and therefore the size of the computing problem, grows.

**Implementation**

With the storage, reading, writing and manipulation of a database system being at the core of this project, a suitable means of carrying out all this was required. A number of solutions were investigated and assessed for their suitability to this topic area and the needs of the project. Some constraints were in place for these database solutions including compatibility with C++/C languages and IDEs, cost of storage, and usability. One solution investigated, which is usually the first to be thought of for database solutions, was Oracle. Upon researching Oracle for this project, and its needs, it was found that there were compatibly issues in relation to C/C++ and Oracle as well as cost of storage on Oracle servers, which resulted in the use of Oracle being ruled out.

After further research SQLite/ SQLite3 was found. SQLite as stated by the creators is an in-process library that implements a [self-contained](https://sqlite.org/selfcontained.html), server less, zero-configuration, transactional SQL database engine (SQLite, 2017). The advantages of this solution are stated as such by the SQLite producers, “SQLite is an embedded SQL database engine. Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file.” (SQLite, 2017) Due to the fact that SQLite operates on a library meant that it could downloaded and the library and header files added to Visual Studios.

With the database solution in place a simple database was to be created. This database contains one table of the name ‘TEXT\_TO\_BE\_SEARCHED’ which itself is made up of three columns: ID, TEXT\_TO\_SEARCH and IS\_PRESENT. A visual representation of this can be seen in fig 1 below:

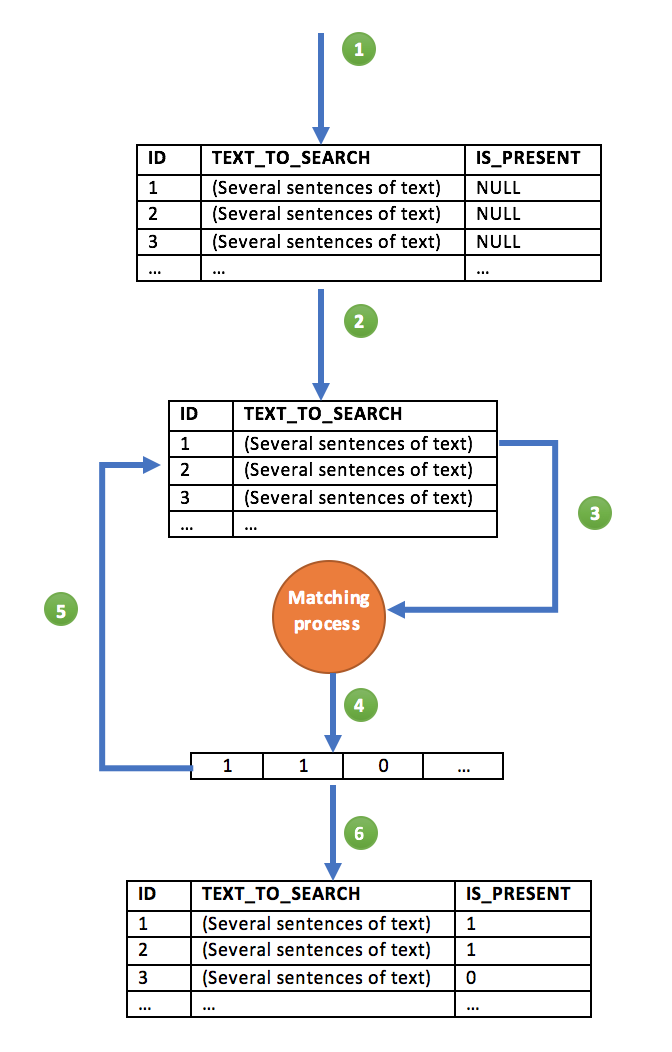


*Fig 1*

Although this is the database and table structure used for this implementation, the SQL queries and code can easily be altered to what is needed. This means that a more advanced database system can be put in place by using SQLite and changing the C++ code. In theory, by doing so multiple tables could be created in the database with their own fields and data which can be searched through independently.

After creation table(s) can be manipulated via the C++ code due to the fact that the database and table(s) are only created on the first iteration when they do not exist. Therefore, columns can be altered, added and removed and data can also be added, updated or removed as needed. This can all be done by writing simple SQL queries and calling the present function in order to run them. Any results i.e. select statements results, can be displayed in the console when run, if needs be. Error messages in relation to the SQL queries being run will also be displayed in the console. These messages are usually very clear and precise in what the problem that occurred is which allows for easier debugging of the code and queries.

The serial process is designed to retrieve and loop through all the data in the desired field and match its contents to the key word/ phrase which has been inputted. If a match is found a 1 is recorded and if not a 0 is recorded. A detailed diagram of this process can be seen in fig 2 with an explanation of each step following it.



*Fig 2*

1. If a database and table has not yet been created i.e. if it is the first run through or the table has been dropped, the database and table is created and populated with data in the format as seen above with IS\_PRESENT initially being NULL.
2. The columns needed are separated, these being the field with the ID and the field with the text to be searched. Once retrieved these two columns are placed into a vector of vectors. This means a vector denoting the columns, in this case 2, is created with the rows of the data retrieved making up the second vector which is inside the first.
3. The first row of the TEXT\_TO\_SEARCH field is retrieved and its contents passed to an ‘if’ statement where the data is checked for the matching word/ phrase.
4. The result of the matching process is written to an array of is\_present values, with the result being a 1 for a match or a 0 for no match. The first checked record is written to the array index 0, the second to 1, the third to 2 and so on.
5. A ‘for’ loop iterates through all the rows of the retrieved data and carries out steps 3 and 4 for each. Once all rows have been searched through, the end result is an array of 1’s and 0’s with the number of elements in the array matching the number of rows in the data searched through.
6. The final is\_present array is then used, in conjunction with a parameterised query, to update the IS\_PRESENT field in the table. To do this an UPDATE query is used with one being added to each index of the array due to the fact that the array is indexed at 0, whereas the table ID’s are indexed at 1. The end product is the original table now containing the results of the matching process denoting if matches were found.

This project is following the usual MPI process of single program multiple data (SPMD), which is where one program is written and multiple copies of it is spawned. The above serial implementation acts as the bases of the program with alterations being made to make it parallel with MPI.

The aim was to have it so the number of rows each processor takes to search is based on the number of rows in the data and the number of processes to be used. Therefore, the number of processes can be specified by the user upon running which will have an impact on the time taken to search through the data. The higher the number of processes the faster the computation is done. In calculating the number of rows and by using MPI\_Comm\_rank(), for processor rank, and MPI\_Comm\_size(), for communicator size, the start and end rows for each processor can be found. A worked example for this approach and a visual representation can be seen below. The root process (process 0) will take the retrieved data to be search and split it into the vector of vectors. The resulting vector of vectors produced is then broadcasted to all other processes using MPI\_Bcast(). Therefore, each process has a copy of the data in order for each to search through their corresponding record. Due to this approach the MPI usage of SPMD is being used in this case. This can be seen from the process described above where the same program is being used for each process but the program is being carried out on different data (the data associated with that process.)

**Worked example (small scale for demonstration purposes)**

*A table has 10 records to be searched and it has been decided this will be run with 5 processors. The table has already been split into the vector of vectors by process 0 and distributed to the other processes. The data to be searched is indexed at 0.*

***Communicator size*** *= 5*

***Number of records*** *= 10*

***Number of records for each processor*** *= Number of records / communicator size = 2*

***Starting row*** *= processor rank \* number of records for each processor*

***Ending ro****w = ((processor rank + 1) \* number of records for each processor) – 1*

*e.g.* ***processor*** *0*

***starting row*** *= 0 \* 2 = 0*

***ending row*** *= ((0 + 1) \* 2) - 1 = 2 - 1 = 1*

*e.g.* ***processor*** *1*

***starting row*** *= 1 \* 2 = 2*

***ending row*** *= ((1 + 1) \* 2) – 1 = 4 -1 = 3*

|  |  |  |
| --- | --- | --- |
| Processor rank | Starting row | Ending row |
| 0 | 0 | 1 |
| 1 | 2 | 3 |
| 2 | 4 | 5 |
| 3 | 6 | 7 |
| 4 | 8 | 9 |