UNIVERSITY OF WITWATERSRAND

ELEN4020

DATA INTENSIVE COMPUTING IN DATA SCIENCE

Group 9: Project 2018

Authors:
KISHAN NAROTAM - 717 931
JESAL CHANA - 603 177

Authors:
SYED HUSSAIN - 600 524
YUSUF ALLY - 604 973

15th May 2018

Abstract

The design and implementation of two parallel equi-join algorithms (MPI and OpenMP) have been presented. Two relational tables, F_1 and F_2 of a user-chosen size can be generated using a C++ program created called `tableGenerator.cpp` and these tables are written to external files, `F₁.txt` and `F₂.txt` respectively. Table F_1 contains two columns, a key (integer) and a randomly generated string (5 uppercase letters), while table F_2 contains two columns, a key (integer) and a randomly generated string (8 lowercase letters), and said tables must be joined on the common attribute, being the key and stored in table F_3 stored in `F₃.txt`. High-level algorithms are given for the table generator, MPI algorithm and OpenMP algorithm with various strengths and limitations presented. The experimental and testing environment of the built cluster and a benchmark device has been expanded on and results of various table sizes and various threads used have been tabulated and graphically presented in order to display the scalability and speed of the algorithms. Due to various time constraints, the full implementation of the MPI algorithm was not possible due to send and receive requests that was being blocked by the code and as a result of this, no testing could be done for the MPI algorithm. However, the MPI algorithm should be faster on the cluster than the OpenMP algorithm.

All source code and documentation can be found on the GitHub repository link: https://github.com/KNarotam/ELEN4020---Data-Intensive-Computing-In-Data-Science

I. Introduction

Large corporations such as *Facebook*, *Google*, *Microsoft* and *IBM* have constantly increasing databases. The maintenance of these databases require dedicated time and plentiful resources. As new entries are constantly added to the database, it continues to get larger. One method of maintaining the database would be to delete unwanted, unnecessary data [1].

In most cases, data is spread across various databases each storing specific types of data. For example, in an insurance company, one database table could be utilized for storing the records of all members that have life insurance, while another database table stores the records of all members that have medical insurance. In a situation like this, combining the two tables would aid the company in accessing all the relevant information stored for a specific member.

In the report that follows two algorithms are implemented using equi-join of two very large tables on their common join attributes. The first algorithm utilizes MPI (Message Passing Interface), while the second utilizes OpenMP. The algorithms are then tested on a cluster, which is explained, as well as a single separate computer that will be used as a benchmark for the algorithms, and the results presented.

II. Background

A. Requirements

The requirement of this project is develop two algorithms that implement a parallel equi-join of two very large tables on the common join attribute. The two tables are stored and maintained in two files, `F₁.txt` and `F₂.txt`. Given that table F_1 has attributes (A, B) and table F_2 with attributes (A, C), these tables must be joined on attribute A to produce relational table F_3 with attributes (A, B, C) which must then be written and stored to file `F₃.txt`.

One of the join algorithms utilized must use MPI. A straight forward MPI distributed programming model with one of of the other programming models, in this case, OpenMP, will be utilized in this project. The purpose of having two different programming models is to allow for comparative performance analysis.

B. Constraints

The join algorithms used must be equi-join and not any other type of join algorithms such as left join, inner join, etc. Additionally, one of the algorithms must use MPI and all testing must be done on the built cluster.

C. Success Criteria

The project and subsequently the algorithms can be considered a success if the requirements mentioned in Section II-A are met. Consequently, the joined tables must be tested on the cluster and the output files must be written to the file F_3.

III. Algorithms

A. Table Generator

The two relational tables, F_1 and F_2, that will be joined are generated using a user-created C++ program named `tableGenerator.cpp`. The function of this program generates the two tables with the user choosing the size of the table. In table F_1, stored in the external file `F₁.txt`, the first column is the key (the joining attribute), and the second column is a randomly generated string of five upper-case letters. Subsequently, in table F_2, stored in the external file `F₂.txt`, the first column is the key (joining attribute) and the second column is a randomly generated string of eight lower-case letters.

Within this file, four global variables are declared, and two functions that return random letters. The `main` function of the class calls these methods and writes the tables to the respective text files. Algorithm 1 shows a high level algorithm of the `tableGenerator.cpp` file.

Algorithm 1 High level algorithm of the `tableGenerator.cpp` class

```
Input: $tableSize$

create the upper-case letters char array
create the lower-case letters char array
calculate size of arrays

Function{getUpperChar}
    return random char from upper-case array
end Function
Function{getLowerChar}
    return random char from lower-case array
end Function

create the two text files $F_1.txt$ and $F_2.txt$
define string length for table 1 to be 5
define string length for table 2 to be 8
User input
for i = 0 till tableSize do
  Generate random string for table 1 and table 2
  Write key and string values to $F_1.txt$ and $F_2.txt$
end for
return 0
```

The user is able to define the size of the table that will be generated as this value will change for testing purposes.

B. MPI Algorithm

The main algorithm uses the MPI (Message Passing Interface Standard) which addresses the message-passing parallel programming model. As opposed to being an IEEE standard, MPI has become somewhat of an industry standard [4]. The master node reads in a single line of table F_1 and broadcasts that key to all the other nodes in the cluster using `MPI_Bcast()`. A second broadcast is done to send the value of table F_1 that corresponds to that key. This distributes the current key to every node to search for any equal keys in table F_1. Since all the slave nodes have access to the home directory each node will read a range of table F_2 corresponding to its rank using Equations 1, 2 and 3.

Then each slave calculates its search range by calculating the required lower and upper bounds by using its specific rank and Equation 1. The upper bound limit is compared to a global variable `numberOfEntries`. If the calculated upper limit is greater than `numberOfEntries` then the upper limit is set to equal `numberOfEntries` to prevent indexing errors. Once the upper and lower bounds have been calculated and the current key has been distributed using the broadcast each slave node can begin to iterate through the portions of the table that corresponds to its set search range and will compare the keys on every iteration to the distributed key from table F_1. On a successful equality of keys the slave node will send the master the index at which F_2 has the same as the key given from table F_1. The master then gets the key, the corresponding value from table F_1, and the corresponding value from table F_2. Then it joins the values from tables F_1 and F_2 and writes the result to a file in the master nodes home directory called `F`$_3$`.txt`.

Unfortunately, due to time constraints the full implementation of this algorithm was not possible and during send and receive requests the code was blocking. To counteract this a local memory version of the algorithm was implemented which would not be able to run on the distributed cluster, however, with changes to the `MPI_send` and `MPI_recieve` commands to handle it on a distributed system.

$$Chunk\ size = \frac{Number\ of\ Entries}{Number\ of\ Nodes} \tag{1}$$

where: $Chunk\ size$ defines the number of values that each node will iterate through; $Number\ of\ entries$ is the total number of entries for table F_2; and $Number\ of\ nodes$ is the number of nodes that will carry out searches.

$$Lower\ limit = Rank \times Chunk\ size \tag{2}$$

where: $Lower\ limit$ is the start of the search range for this node; $Rank$ is the unique rank given to the nodes that are searching; and $Chunk\ size$ defines the number of values that each node will iterate through.

$$Upper\ limit = ([Rank + 1] \times Chunk\ size) - 1 \tag{3}$$

where: $Upper\ limit$ defines the final element in the nodes search range; $Rank$ is the unique rank given to the nodes that are searching; and $Chunk\ size$ defines the number of values that each node will iterate through.

1) Strengths:

- The algorithm makes use of the parallel structure of the cluster efficiently, by delegating search ranges to each slave node. This allows F_2 to be read up to N times faster, where N is the number of slaves, this excludes delegation and send times by the master.
- The solution is scalable as the block size of F_2 is dependent on the number of available nodes. As more nodes are added, the master node is able to delegate blocks to maximize the number of nodes for a particular table size.

2) Limitations:

- Each slave has to read from the same master data and so this can cause a bottleneck if the communication bandwidth is insufficient.
- The number of elements needs to be divisible by the number of process, if not then the final process will receive less search elements to read through than the other slave nodes.
- Since the data is read by the master node into its local storage it means that the algorithm requires enough storage space to hold the table file in its original form as well as in the vector in memory.
- The size and elements of the Tables are required before accurate distribution of data elements and search ranges.

3) Improvements:

- Using scatter and a dynamic file read can improve performance issues relating to storage size and bandwidth problems. `MPI_Scatter()` partitions the given data By using a dynamic file read we can reduce the total size of the table in memory by removing entries that have been sent to a slave node via scatter.
- Using `structs` to broadcast information and to do storage and joins will lead to a more robust and flexible solution which can handle different types of values and data types.

C. OpenMP Algorithm

The second algorithm uses OpenMP with the nested `for` join. OpenMP (Open Multi-Processing) is an an application program interface (API) that is used specifically for parallel programming. It is regarded as a much higher level API, and portable allowing OpenMP to be utilized in a variety of languages such as C, C++ and Fortran. It is more easily scaled than PThreads, as it has the ability to divide the work across multiple threads with ease [5].

The algorithm allocates a master thread and slave threads, the total of which equal the number of processing nodes available. Table F_1 and Table F_2 are generated using the `tableGenerator.cpp` and are stored in the shared memory of the cluster, i.e. these files are accessed through all of the nodes. The tables are loaded from the shared memory via the master thread and is read in line by line.

Once the data is read in, an OpenMP lock is created. The purpose of the lock is to guarantee that the next set of instructions can only be performed one process at a time, i.e. it prevents different threads from simultaneously changing the same data [5]. A set of global variables are then shared between the slave threads. These variables are: a lock variable to allow a thread to lock a process, Table F_2, Table F_3, the size of Table F_1 and the size of Table F_2.

The join algorithm used with OpenMP is similar to that of the brute-force nested loop join with no partitioning stage. The nested `for` loop is used to iterate through the chunk given to each thread. A key value pair is stored in a temporary variable in the first `for` loop and is passed to a second `for` where the key of the temporary variable is compared to all the values in Table F_2. Using an `if` statement, the keys are tested for equality, and when the keys match, an entry is added Table F_3 with the corresponding key and string values from Table F_1 and Table F_2. This process is done on a locked thread. Algorithm 2 shows a high level algorithm of the `openMP.cpp` file.

Algorithm 2 High level algorithm of the `openMP.cpp` class

```
Input: $NUM\_THREADS$

read $F_1.txt$ and store in array of $key$-$value$ pairs
read $F_2.txt$ and store in array of $key$-$value$ pairs
ask user to set $NUM\_THREADS$
set number of threads
initialize omp_lock
initialize threads with shared variables
int count ← 0

for i = 0 till size of $F_1.txt$ do
  tempKeyA ← keyA[i]
  for j = 0 till size of $F_2.txt$ do
    tempKeyB ← keyB[j]
    if tempKeyA = tempKeyB then
      set omp_lock
      C[i] ← A[i]+ " " + B[i]
      count++
      unset omp_lock
    end if
  end for
end for
write joined table to file $F_3.txt$
return 0
```

IV. Experimental Environment

A. Cluster

A computer cluster was created for the testing purposes of the project. A computer cluster is defined as a single logical unit whereby multiple computers are linked together through a LAN. These networked computers now act as a single machine with more power, increasing processing speed, the storage capacity, improving the data integrity and reliability. The advantage of working within a cluster allows for large data processing, an increase in speed of data processing and provides users a high availability of resources [2]. Figure 1 shows the basis of the cluster design and architecture.

The cluster begins from RAID-1, or disk mirroring. A redundant array of independent disks (RAID) is defined as a method of grouping individual physical drives together in order to form a bigger drive called a RAID set. There are various types of RAID sets, which are defined by level numbers, whereby RAID-0, RAID-1 and RAID-5 are the most common. In this cluster, a RAID-1 set is used, which is regarded as the most reliable. It allows for no data to be lost across the hard disk drives and synchronously mirrors the data from each hard disk drive to an exact duplicate [3].

In order to access the cluster, the user must first Secure Shell (SSH) via a terminal such as *PuTTY*. The user will then log into *Jaguar1* or *Dica01*, and using a `machine` file, the user will be able to use the nodes *Dica02* till *Dica09*. The user can determine the number of nodes that will be utilized in processing the different algorithms.

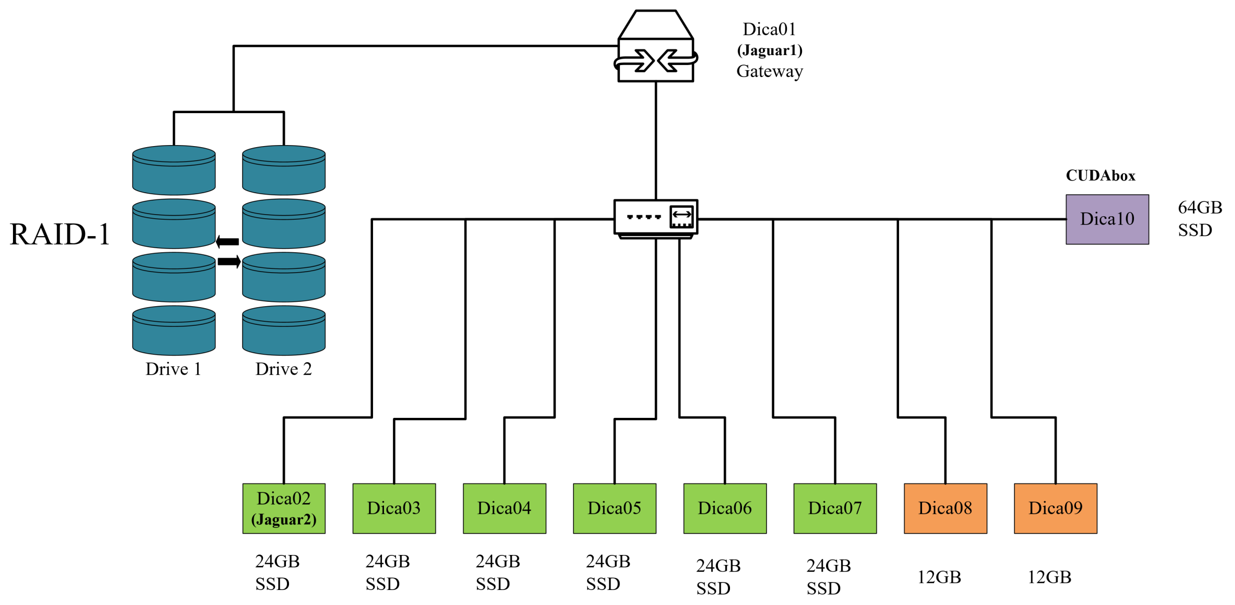

Fig. 1. Diagram showing the basis of the cluster, the nodes' memory and type of hard disk drives

B. ASUS

An ASUS N550JV high performance laptop was used as a "benchmark" for the OpenMP testing. The machine has an *Intel(R) Core(TM) i7-4700HQ CPU* with 12GB of RAM installed with a 64-bit operating system installed. The operating system installed is *Ubuntu 16.04 LTS*.

V. Results

A. MPI Results

Due to time constraints, and an unforeseen error on the cluster, thorough testing of the MPI algorithm could not be done. As the benchmark computer used could not simulate 2 nodes (processes) due to the error of the send and receive requests were being blocked by the code itself. Theoretically, the MPI algorithm, if programmed correctly should yield faster times.

B. OpenMP Results

The OpenMP algorithm, described in Section III-C was tested on the benchmark system mentioned in Section IV-B and on the Cluster, namely the node *Dica02*. In order to show the true scalability of the and speed of the systems, tables of increasing sizes are generated and the algorithm is run with varying number of threads. Table I shows the results of all the times recorded, in seconds, after being executed on the *ASUS* laptop and on *Dica02*. Although with a normal hard disk drive, the *ASUS* displayed faster times than *Dica02*'s execution times. This difference in time is due to the fact that at the time of testing and execution, *Dica02* had one zombie process, 284 processes being run and 23.1% of the memory was being used. This difference can be seen graphically in Figures 2, 3, 4, 5, 6 and 7.

VI. Conclusion

The implementation of the two parallel equi-join algorithms of various table sizes (large) have been presented. The criteria defined for success has been met, and the algorithms utilized two programming models, namely MPI and OpenMP. High-level algorithms of each method and model used have been presented and the algorithms were tested on a benchmark device and on the built cluster, and results were presented in various tables and various graphical representations. Due to

TABLE I
TABLE SHOWING THE TIME TAKEN (IN SECONDS) TO PERFORM THE JOIN ALGORITHM ON THE *ASUS* LAPTOP AND ON *Dica02* ON THE CLUSTER

Threads = 0			**Threads = 4**			**Threads = 8**		
Table Size	**ASUS**	**Cluster**	**Table Size**	**ASUS**	**Cluster**	**Table Size**	**ASUS**	**Cluster**
10	0.000505	0.000427	**10**	0.000691	0.00848	**10**	0.020215	0.029212
100	0.000451	0.000653	**100**	0.001585	0.008773	**100**	0.060356	0.028056
500	0.008197	0.00311	**500**	0.009749	0.014723	**500**	0.055863	0.036726
1000	0.019902	0.012039	**1000**	0.048367	0.028535	**1000**	0.052915	0.050996
2000	0.035553	0.042524	**2000**	0.077996	0.073102	**2000**	0.125781	0.106062
5000	0.161803	0.249228	**5000**	0.232968	0.375876	**5000**	0.324462	0.435307
10000	1.20043	0.977631	**10000**	0.844848	1.4016	**10000**	1.0934	1.68316
50000	15.6016	24.1215	**50000**	16.1613	32.6204	**50000**	27.6152	45.6045
100000	63.3261	97.2487	**100000**	98.6917	122.211	**100000**	136.779	186.858
Threads = 16			**Threads = 64**			**Threads = 128**		
Table Size	**ASUS**	**Cluster**	**Table Size**	**ASUS**	**Cluster**	**Table Size**	**ASUS**	**Cluster**
10	0.002822	0.00116	**10**	0.007291	0.001821	**10**	0.00649	0.003276
100	0.002869	0.001819	**100**	0.003474	0.003154	**100**	0.00605	0.006245
500	0.01392	0.005843	**500**	0.024972	0.006283	**500**	0.020082	0.00633
1000	0.043679	0.016396	**1000**	0.047186	0.017582	**1000**	0.044691	0.021302
2000	0.091387	0.074674	**2000**	0.089366	0.078589	**2000**	0.090983	0.077071
5000	0.268918	0.464784	**5000**	0.327753	0.436735	**5000**	0.285227	0.460076
10000	1.08484	1.87051	**10000**	1.10223	1.81869	**10000**	1.15194	1.88979
50000	28.314	46.6062	**50000**	28.7935	47.0111	**50000**	28.814	46.935
100000	127.945	187.502	**100000**	140.11	191.564	**100000**	129.064	188.037

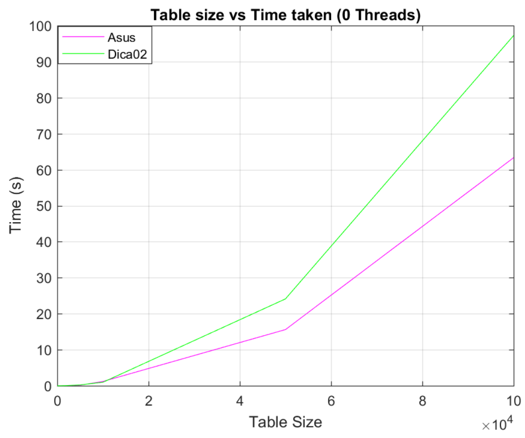

Fig. 2. Graph showing the *ASUS* and *Dica02* execution times of the OpenMP algorithm with 0 Threads

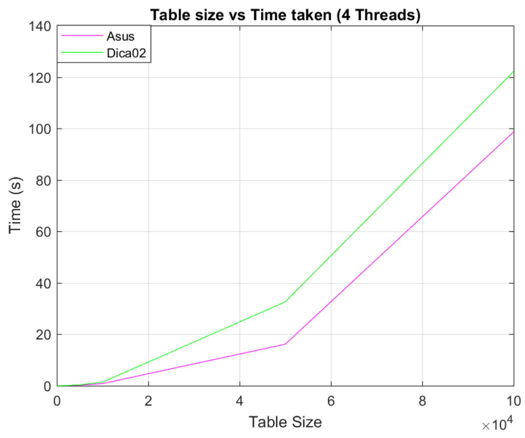

Fig. 3. Graph showing the *ASUS* and *Dica02* execution times of the OpenMP algorithm with 4 Threads

various time constraint factors, and no prior definition of *large* table size, tables of up to 100000 were generated and tested on as this size could be defined as a large data table set.

REFERENCES

[1] IBM; *Maintaining Databases*; http://publib.boulder.ibm.com/tividd/td/ITWSA/ITWSA_info45/en_US/HTML/guide/t-expire.html; Last Accessed: 07/05/2018

[2] Techopedia; *What is Computer Cluster? - Definition from Techopedia*; https://www.techopedia.com/definition/6581/computer-cluster; Last Accessed: 07/05/2018

[3] Rouse, M; *What is disk mirroring (RAID 1)? - Definition from WhatIs.com*; https://searchstorage.techtarget.com/definition/disk-mirroring; Last Accessed: 07/05/2018

[4] Barney, B; *Message Passing Interface (MPI)*; https://computing.llnl.gov/tutorials/mpi/; Last Accessed: 10/05/2018

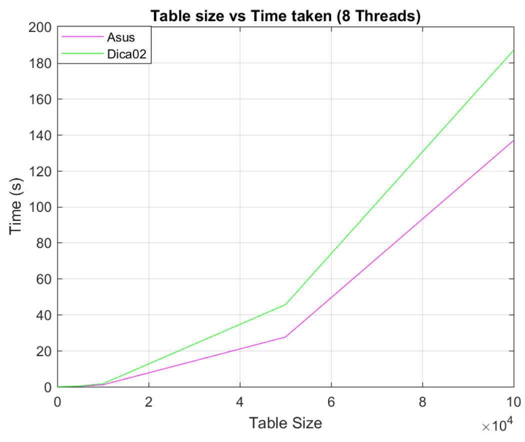

Fig. 4. Graph showing the *ASUS* and *Dica02* execution times of the OpenMP algorithm with 8 Threads

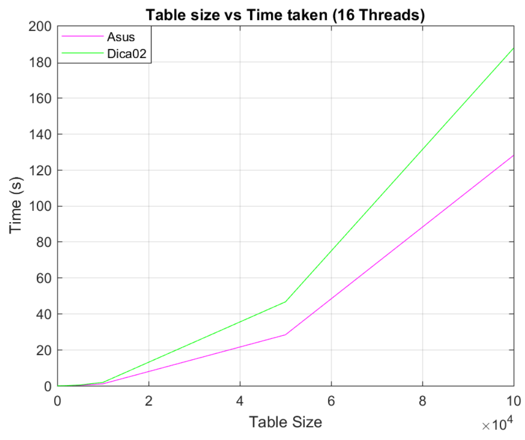

Fig. 5. Graph showing the *ASUS* and *Dica02* execution times of the OpenMP algorithm with 16 Threads

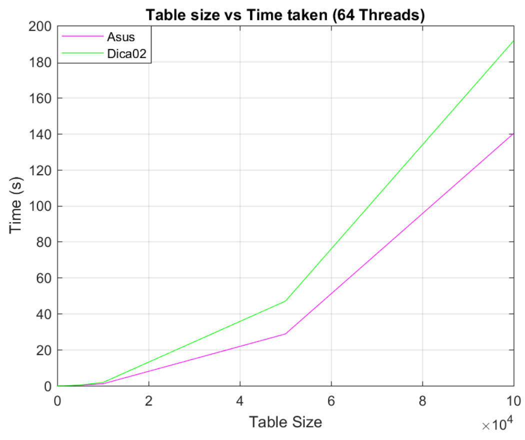

Fig. 6. Graph showing the *ASUS* and *Dica02* execution times of the OpenMP algorithm with 64 Threads

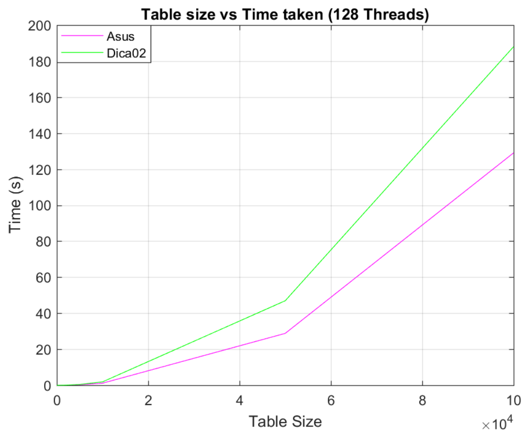

Fig. 7. Graph showing the *ASUS* and *Dica02* execution times of the OpenMP algorithm with 128 Threads

[5] Barney, B; *OpenMP*; https://computing.llnl.gov/tutorials/openMP/; Last Accessed: 10/05/2018