A PERFORMANCE EVALUATION OF TWO EQUI-JOIN MODELS FOR PROCESSING BIG DATA

Kayla-Jade Butkow (714227), Jared Ping (704447), Lara Timm (704157) and Matthew van Rooyen (706692)

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa

Abstract: This paper presents the design and implementation of a Hybrid equi-join using MPI and OpenMP, and an equi-join using Phoenix++ MapReduce. The two solutions were implemented in C++ to allow for low level memory management. After running benchmarks for the two implementations, it was discovered that the Hybrid solution has superior performance for input files larger than 5 GB. For future recommendations, a concurrent vector should be used to improve the performance of the Hybrid solution.

Key words: Equi-Join, Hash, MapReduce, Message Passing Interface

1. INTRODUCTION

A relational *join* operation describes the concatenation of tuples from two relations to form a new relation [1]. Along with the *select* and *project* operations, these form a base for normalisation which is critical for designing database relations [1]. A join is a product of relations followed by a restriction clause [1]. An equi-join follows this structure where the restriction clause set to be an equality operator [1]. For large datasets, this product is a computationally costly operation and as such, it must be optimized for efficient use.

This paper compares the performance of two programming models, namely an MPI-OpenMP hybrid and Phoenix++ MapReduce, for computing a parallel equi-join of two very large tables on their common join attributes. The design and implementation of each model is discussed and a comparative performance analysis between the models is given. Finally, future recommendations are given based on this discussion.

2. PROBLEM DESCRIPTION

2.1 Requirements and Success Criteria

The project requires the development and implementation of a hybrid programming model to compute a parallel equi-join of two very large relational tables. Additionally, a different algorithmic approach to solve the same problem must be developed and a performance comparison between the two implementations must be undertaken. The joined relation must be written to a file for the result to be corroborated.

The project will be considered a success if the

developed MPI-OpenMP Hybrid algorithm is able to compute the equi-join of two relational tables of approximately 1 GB, or larger, in size.

2.2 Assumptions

During the design and implementation of the Hybrid join algorithm, a number of assumptions were made. Firstly, performing an equi-join when the join attribute is a non-primary key, and where the chosen attribute is duplicated in one or both of the input relations, results in an output relation containing all possible permutations of the tuples in the input for that join attribute. A further assumption is made that big data constitutes a file size of 1 GB or larger.

3. PROJECT BACKGROUND

In order to process large amounts of data, join operations are often needed [2]. There is a great need for efficient join operations since the operation is very expensive in terms of both CPU usage and I/O costs [2].

3.1 Join Algorithms

In an equi-join operation, two relational tables $R_1(A, B)$ and $R_2(A, C)$ are joined on their common attribute A to produce a relation $R_3(A, B, C)$ [3]. To improve the performance and reduce the cost of implementing a relational join on large databases, the computational load can be shared across multiple processors [3]. This is achieved by implementing a parallel, distributed join algorithm which reduces the processing time and increases the efficiency of the algorithm for computations on a large scale [3].

The equi-join operation can be performed in a variety of ways. Two of these methods are a sort-merge join and a hash join.

3.1.1 Sort-Merge Join

The sort-merge join algorithm is based on the principle of sorting both relations according to their join attributes [3]. This is done in order to ensure that when scanning through both relations, related sets are encountered at similar times [3]. The algorithm consists of two phases, namely the sorting phase and the merging phase. For datasets being processed in parallel, the sorting phase occurs by making use of data partitioning [4].

In the first step, each thread partitions the input data [4]. The partitioning makes use of range based partitions to ensure that matching elements in both of the input relations are processed by the same node [4]. Within each range, the data is sorted and asynchronously sent to its target node for processing [4]. While the data transfer is taking place, the thread continues to sort the next chunk of input data [4]. The performance of the sort algorithm is thus limited by

two factors, the rate at which each chunk can be sorted and the network bandwidth available to each process at the given node [4].

After a node has sorted its input data, it waits until it has received all of the sorted data belonging to its range from all of the other nodes [4]. The algorithm then merges the sorted chunks into a sorted join result. Multiple iterations over the input data may be needed until both relations are fully sorted [4]. After both relations have been sorted, they are partitioned among all the nodes for merging. Merging constitutes computing the join result between the corresponding tuples [4].

The costs associated with the sort-merge join algorithm depend on the level of differentiation within the input files and the transmitted data that needs to be sent across the network [4].

3.1.2 Hash Join

In a hash join, rather than sorting the data, a hash table is created containing the (key,value) pairs [3]. The hash join algorithm has a build phase and a probe phase [5].

In the build phase, the algorithm generates a hash table using the smaller relation [6]. The hash table contains the join predicate, upon which the hash function is applied, and it's corresponding tuples of information [3, 6]. The build phase is completed when all the tuples of the initial relation have been stored in the hash table [3].

Once the table is created, the larger relation is read in and its tuples probe the hash table for a corresponding match [3]. This action forms the probe stage [6]. A hash join is used over a sort-merge join because searching the hash table for the join predicates is much more efficient than simply scanning through the original relation [6]. The entire process makes up a Simple Hash Join [6].

The Grace join algorithm differs from a simple hash join as it partitions both relations instead of scanning through the second, larger relation [7]. This is achieved by adding a third phase to the algorithm's make up. In the first phase, the algorithm partitions the smaller relation into a hash table [7]. In phase two, the larger relation is also partitioned into a table using the same hash function [6]. In the final phase, the algorithm joins the matching (key,value) pairs from both hash tables [6].

3.2 Message-Passing Interface

Message passing is a form of communication between two separate processes [8, 9]. By using message passing, processes can send and receive resources [8]. If data is transferred, the data must be moved from the local memory of the

sending process to that of the receiving process [8].

The Message-Passing Interface (MPI) is a library specification for message passing in parallel and distributed environments [10]. It is widely used in high-performance computing systems [9, 11]. The MPI library offers point-to-point communications, broadcast messages and parallel I/O [10].

MPI can be used for a variety of parallel computing models [10]. One such model is Single Instruction Multiple Data (SIMD) in which the same instruction is carried out simultaneously on multiple data sets [10]. Another model is Multiple Instruction Multiple Data (MIMD) in which different instructions are performed on separate sections of data [10]. The advantages of MPI is that it is a powerful, efficient method of executing parallel programs [8]. It is also portable, as message passing is implemented on most parallel platforms [8].

3.3 OpenMP

The OpenMP library is a multiprocessing application program inference (API) used to develop shared-memory parallel programs [10]. The library allows the same parallel code base to be run identically across a range of different operating systems [10]. OpenMP takes the form of a set of compiler directives (pragmas) which are used for thread creation and synchronization of operations [10].

The compiler directives allow for a serial program to be compiled into a multi-threaded program [12]. The OpenMP API makes use of the fork-join parallel design pattern, whereby in the parallel regions, threads are created to perform work concurrently [13]. Once the work has been completed, the threads join together to create a single result and to recreate the single control thread [13]. All of the threads within a program share a memory address space, thus allowing for efficient communication between the threads [10].

3.4 MapReduce

MapReduce is a programming paradigm for processing big data [10]. The model has been widely adopted for data processing, for the solving of practical problems, due to it's simplicity and ease of use [2, 10]. The algorithm consists of two distinct parts, namely the map function and reduce function [14].

The map function takes as an input a line (or chunk) of data, and emits a set of (key,value) pairs [14]. The map stage can be executed in parallel without any collaboration being required from threads, since each chunk of data is processed independently [10]. Once the key value pairs have been emitted, the resultant

keys are hashed and sorted, and then all values associated with a given key are grouped together [14]. The reduce function then aggregates all the values associated with a key and performs a function that is selected by the programmer [10].

The MapReduce model is well suited for data intensive processing for a number of reasons. Firstly, the model is able to sequentially work through the records in a data set without loading the entire set into memory [10]. The program is also convenient to use as the MapReduce model performs the hashing of the keys and the sorting of the (key,value) pairs inherently [14]. The programmer is able to override these if they wish, but they are not required to, thus enhancing the ease of using the paradigm. Furthermore, the parallel elements are also inherently included in the paradigm, which allows for the programmer to create easily scalable code [10].

Phoenix++ is a C++ based MapReduce implementation that makes use of PThreads to allow for scalability and efficient data processing [14].

4. SYSTEM DESIGN AND IMPLEMENTATION

The two implemented solutions are a Hybrid equi-join using MPI and OpenMP, and an equi-join implemented using MapReduce. The Hybrid solution makes use of the framework set out by the serial solution, but extends it using MPI and OpenMP. Both solutions were created using C++, as this allows for the low level memory management of C, but still allows for the use of advanced variable types, such as vectors.

4.1 Serial Equi-Join

An object-oriented solution was implemented for the serial equi-join, which makes use of a simple hash join algorithm. This algorithm was selected as it offers superior performance to the sort-merge join, and still allows for ease of programming (as opposed to the Grace hash-join). Within this solution, two classes were created, namely, `hashFunction` and `fileManager`. The `fileManager` class was responsible for any functionality relating to the management of files. Within the `fileManager` class, the input text files are read in and split by delimiter and by end of line character. The data was returned in the form of a vector which contains a key in every even index and the original line from the text file in every odd index.

Once the two input files have been read in and mapped to two vectors, an instance of the the `hashFunction` class is used to created a hash table. Since a simple hash-join algorithm was implemented, a hash table was only created for one of the input text files [6]. In creating the table, each key was hashed using Algorithm 1. By creating a hash of the key, a corresponding (index,(key, value)) pair is created, where the

index refers to the location of the value in the hash table.

In order to avoid the need for contiguous memory to store the hash table, the table was created as a series of pointers which each point to the next value in the table. If multiple keys are hashed to produce the same index, buckets are created which hold all of the (key,value) pairs with the same hash. The implication of using a hash table is that when searching for a key, the entire table does not need to be searched. Rather, the key of interest is hashed, and then only the bucket at the corresponding index in the table is searched for the required key. Once the first file has been hashed, the hash table must be probed by the tuples in the second file. To do this, the key is hashed to find its corresponding index in the hash table. Thereafter, each key in the hash table is checked for equality with the probing key. If they are found to be equal (thus satisfying the conditions of an equi-join), the (key,value) pair in the hash table is added to a vector of results. Once the whole bucket has been probed, the vector contains all of the (key, value) pairs that must be joined to the probing key, and it is returned.

Algorithm 1 Hash Function

```
function HASHFUNCTION(key)
    hashVal ← arrayInfo._array_0
    for i = 0 to length(key) do
        hashVal ← hashVal + int(key[i])
    end for
    index ← hashVal % tableSize
    return index

end function
```

Two functions were created to handle the joining of the two tables. In these functions, the value from each resulting (key, value) pair is appended onto the probing (key, value) pair in order to join the tables. To reduce repetition of the key in the joined table, the key is extracted from the resulting value prior to appending the strings. The joined strings are then appended to a vector. This process occurs for each (key,value) pair in the second file. Finally, the vector containing the joined table is returned and written to an output text file.

4.2 Hybrid Equi-Join

The first implemented solution utilises the MPI communications framework in conjunction with the OpenMP API. The MPI framework is used to create a scalable solution which can run on clusters of any size, allowing each node within the cluster to contribute to the computational load of the program [15]. The mpic++ compiler is utilised as it provides a wrapper that interfaces with the g++ compiler [16]. A 64 bit version of the g++ compiler is utilised, and as such the file buffer pointers used were 64 bit pointers [17]. By considering that the file buffer is a `signed long int` data type, this translates to the ability to parse files up to 8.5 EB in size [17].

Thus the memory bottleneck will most likely be enforced by the amount of RAM and page memory available within the cluster. This allows the file content to be read in completely, minimising data access overhead that would be incurred by reading in data chunks through the program execution.

OpenMP was selected for the Hybrid model since it makes the implementation of multithreaded programs much simpler. This is because the complier transforms the serial code into parallel according to the directives [10].

A `FileManager` object is utilised by the compute nodes to read in the two data files that are required to be joined. In order to reduce the overhead involved with the use of `MPI_send` with a master node `FileManager` object, the parallel architecture is exploited in reading in the input text files on each compute node. For the smaller input text file, the entire file is read in by each process and saved in a vector. For the larger input file, each process reads in the size of the input file, and calculates the chunk sizes by dividing the size of the file by the number of processes. Each process then reads in the file chunk relating to their rank.

Using the contents of the file chunk, a hash table is created on each compute node. When performing the hash, the value at the specified column index is set as the key and the line from which it was extracted is set as the value.

Each compute node utilises the OpenMP API to parallelise the hashing of the data segment. This allows the adding of keys to the hash table to be done in a parallel manner. The entire smaller file is used, by each node, to query the hash table. In order to parallelise the query and join components of the hash-join, vectors containing join results (corresponding to each unique key) are pushed back to a results vector. This allows for concurrent operations to occur. Thereafter, the vector of vectors is iterated through and a resultant vector is generated.

The use of a 2D vector to facilitate parallel operations is time expensive for large datasets. This design was selected since a `std::vector` container from the C++ Standard Template Library (STL) does not allow concurrent modification operations [18]. This presents a tradeoff in the design, which is discussed in detail in *Section 6.2*.

Once each compute node has completed its required hashing computations, the resulting joined data is sent back to the master node using `MPI_recv`. The order of received data is not important and thus the transmitted results from compute nodes can be received as they are completed. The master then proceeds to write the resultant data to an output file, as soon as each resultant data chunk is received. This write pro-

cess occurs on a single thread to prevent data corruption due to simultaneous writes to a single output file.

4.3 MapReduce Equi-Join

The second implemented method makes use of the Phoenix++ MapReduce framework. MapReduce was selected since it is the industry standard for the processing of large amounts of data [10]. The implemented algorithm is a general reducer-side join [2]. Within this algorithm, a single map and reduce stage is used. Within the map stage, an identifier corresponding to each file is attached to the value [2]. The (key, value) pair is then emitted. Within the reduce stage, data with the same key and a different tag are joined together [2].

In order to compensate for input text files that are too large to be held in RAM all at once, the input text files are mapped to memory. They can thus be read in when needed, in chunks that can be held in RAM. Since the memory map maps the data directly back to the text file, when the data is edited, the file is also edited. To avoid altering the input text file, two copies of each file are made - one used for isolating the key, and one for the value. Thus, four input files are mapped to memory. This presents a trade-off in the design, which is discussed in detail in *Section 6.2*.

After mapping the data to memory, the data is divided into chunks to be sent to the map processes. Checks are performed to ensure that each chunk only contains full lines of the text file. This is essential in ensuring that none of the (key, value) pairs are missed.

Within the mapper, the (key, value) pairs from the text file need to be isolated and emitted. In order to isolate the values, the file is iterated through until an end of line character is encountered. This end of line character is set to be the integer `0`, which is the integer value of the null terminator, and indicates where a string ends [14]. This string represents the value.

To isolate the keys, the column that contains the keys must be known. This information is obtained from the program's run-time arguments. When iterating through the text files, a check is done to determine whether each character is a delimiter or a normal text character. If the character is a delimiter, a counter of delimiters is incremented. When the delimiter counter is equal to the number of the column containing the keys, the start of the key is recorded as the following character. The string is then iterated through to find the next delimiter which marks the end of the key. This character is then set to `0`. The key and the value are then emitted to the reducer. In order to identify which (key, value) pair originated from which input file, the

pair is also emitted with an identifier (`0` for file 1 and `1` for file 2).

Once the mapping process is completed, the keys are hashed and the data is sorted. These processes were however not overwritten to implement the join and as such, the in-built Phoenix++ functionality was used.

Each key, and all of its corresponding values, are then sent to the reducer. Within the reducer, the values are split into two vectors based on their identifiers. Thereafter, for each combination of the two vectors, a string is created which contains the two values appended together. The process of appending the two values represents the joining of the row from the two tables. Together with the key, the joined value is then emitted from the reducer, and written to a text file.

Finally, to complete the program, the files are unmapped from memory and the four copied input files are deleted.

Phoenix++ inherently makes use of PThreads and as such, the equi-join is performed in parallel.

5. RESULTS

To test the performance of the two solutions, the solutions were benchmarked with different sized input files. These files were generated such that each file contained the same keys. As such, each line of the two input files was joined in the output file. For the Hybrid solution, the number of nodes was altered, and for the MapReduce solution, the number of threads was altered.

The Hybrid solution was run on two to eight nodes, in which one was a master node, and the remainder were compute nodes. Eight threads were used per node as this minimises caching load. The MapReduce solution was run on a single node, consisting of eight Intel i7 CPUs, each with two threads. The node had 24GB of RAM.

The benchmarks were run using three sets of input files of various sizes. Firstly, the implementations were run using input files with 11 881 376 lines and a size of 1.01 GB. Then files with 24 137 569 lines with a size of 2.00 GB were used. Finally, files containing 34 012 224 lines with a total size of 2.8 GB were used.

Table 1 provides the results of the Hybrid solution benchmarks using a range of input file sizes.

6. CRITICAL ANALYSIS

Figure 1 provides a graph of the time taken for the Hybrid equi-join per number of nodes for the 1, 2 and 2.8 GB input files. From the graph, it is clear that as the file size increases, the time to perform the join increases. It also indicates

Table 1: Comparison of time taken (in seconds) to compute the equi-join using the Hybrid solution with various input files over scaled numbers of nodes

File Size (GB)	Numbers of Nodes						
	2	3	4	5	6	7	8
1	44699.2736 s	21574.4287 s	13324.3839 s	9750.6720 s	6301.0722 s	4965.0615 s	4048.9356 s
2	86398.5473 s	41869.4638 s	26134.3246 s	18629.1303 s	12230.7858 s	9458.8837 s	8114.7422 s
2.8	120157.9663 s	58489.3100 s	36279.3885 s	24249.1338 s	16900.2848 s	13273.8530 s	10952.1978 s

Table 2: Comparison of time taken (in seconds) to compute the equi-join using MapReduce with different input files sizes over various numbers of threads

File Size (GB)	Number of Threads			
	1	4	8	16
1	58.179223 s	30.209526 s	27.320309 s	36.645406 s
2	121.903450 s	64.357541 s	54.217165 s	66.751160 s
2.8	12731.958691 s	6686.974704 s	8586.499522 s	10489.991414 s

that the performance gain of the algorithm is proportional to the number of nodes utilised for computation as the workload is shared between them. From the figure, it can be seen that performance improves as the number of nodes increases.

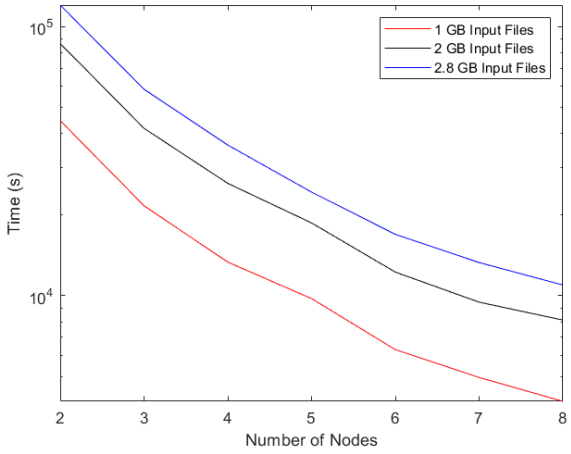

Figure 1: Time taken for the equi-join vs number of nodes for the Hybrid solution for various file sizes

Figure 2 is a speedup plot for the Hybrid equi-join. Speedup is defined as the ratio of the execution time for one node to that of N nodes [19]. The graph emphasises the conclusions made from *Figure 1*, being that as the number of nodes increases, performance increases. The performance increase occurs linearly as the number of nodes increases.

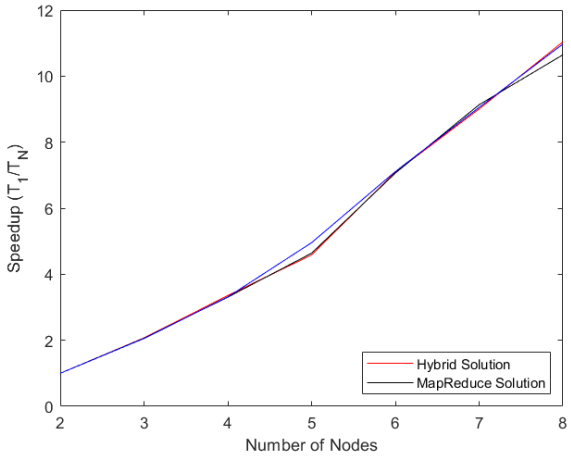

Figure 2: Speedup plot for the Hybrid solution

A scalability plot for the Hybrid solution is given in *Figure 3*. Scalability is defined as the ratio of the serial execution time to that of N nodes [19]. From the figure it is clear that between two and six nodes, the scalability increases linearly. However, from seven nodes onwards, the scalability seems to plateau. This is on account of the additional overhead for the MPI message passing. This may also be attributed to other bench-

marks being run concurrently on the additional node. As such, node resources were shared and thus a true measure of the performance of the model was not obtained.

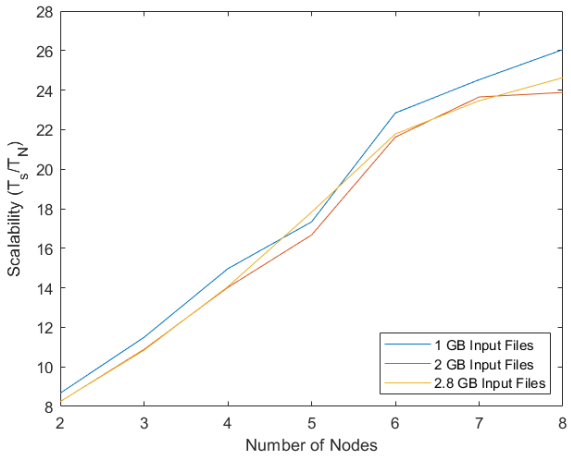

Figure 3: Scalability plot for the Hybrid solution

Figure 4 provides a graph of the time taken for the MapReduce equi-join per number of threads for the 1, 2 and 2.8 GB input files. From the graph, it is clear that as the file size increases, the time to perform the join increases. It also indicates that for the smaller input files, the optimal performance is achieved for eight threads. However, for the large input file, the best performance is achieved when four threads are used. Since the node used to run the benchmarks had eight hardware threads, it is expected that running the program with eight threads would give the best performance. It is thus an outlier that the best performance for the 2.8 GB input files was achieved for four threads. From the figure, it can also be inferred that when running the 2.8 GB input file, all of the node's RAM was used, and as such, hard drive space was used for virtual memory [20]. Since hard drive I/O speed is much slower than RAM I/O speeds, the use of virtual memory results in the drastic speed decrease seen in the graph [20].

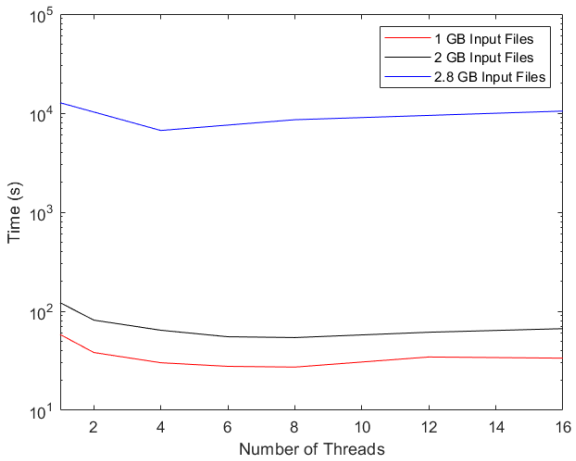

Figure 4: Time taken for the equi-join vs number of threads for the MapReduce solution for various file sizes

Figure 5 is a speedup plot for the MapReduce equi-join. The graph emphasises the conclusions made from *Figure 4*, being that as the number of threads increases, performance increases, until the maximum number of hardware threads per core is exceeded. Thereafter, performance decreases.

In order to obtain an accurate comparison additional benchmarks, above those previously mentioned, were run. By examining *Figure 6*, it is clear that for file sizes up to approximately 5 GB, the MapReduce solution out performs the Hybrid solution. However, thereafter the Hybrid solution has superior performance. The Hybrid solution also boasts linearity over in-

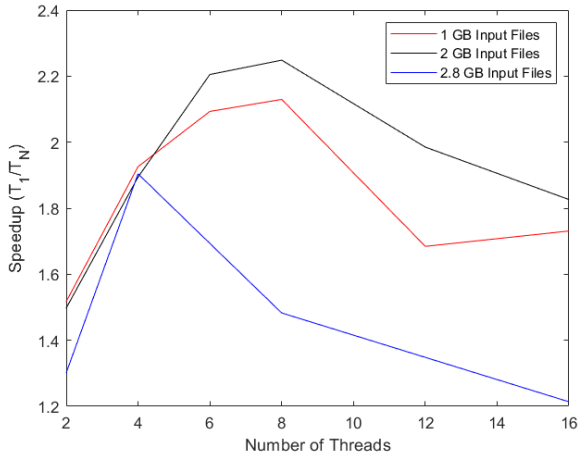

Figure 5: Speedup plot for the MapReduce solution

creasing file sizes, making it a more adaptable solution.

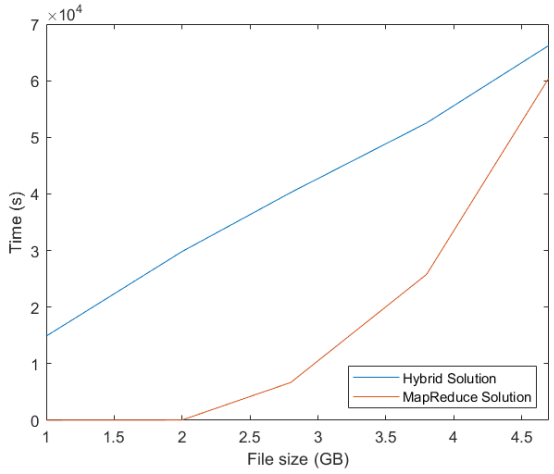

Figure 6: A comparison of the time taken for the two solutions over various file sizes

A major strength of the Hybrid solution is that it is both scalable over a large number of nodes and also adaptable to increasing file sizes. On the other hand, the MapReduce solution can only be scaled up to the maximum number of hardware threads, and thereafter performance declines. It is also not able to handle large file sizes with linearly increasing execution times. This presents a weakness of the MapReduce solution.

A strength of the designed solutions is that both solutions are able to perform equi-join operations on any column of a table. Furthermore, the Hybrid solution is able to handle any type of delimiter within a file.

6.1 Limitations

A limitation exists with regards to the network bandwidth of the cluster. This translates to incurred overhead when node communication commands, such as `MPI_send` and `MPI_recv`, are extensively used.

A limitation of the implemented MapReduce model is that an assumption is made that the reducer has sufficient memory to hold all of the values with the same key [2]. If the reducer does not have sufficient memory, the performance of the algorithm is greatly affected.

6.2 Tradeoffs

The use of a STL `std::vector` to store the hash join results is a large tradeoff of the system. While this usage allows for reduced computational complexity and improved platform compatibility, it creates redundancies in the efforts to parallelise all possible components of the Hybrid equi-join algorithm. Therefore this design choice creates a large inefficiency within the system.

The use of memory mapping to store the input text files for the MapReduce solution is a large tradeoff of the system. While this usage means that larger input files can be processed, it also lead to the need to duplicate each input text file twice, and allocate memory for each of the four text files. This creates a large inefficiency within the system.

7. FUTURE RECOMMENDATIONS

For future recommendations, a Grace or Hybrid hash-join algorithm should be implemented rather than a simple hash-join. This would improve performance at all levels of memory availability [6].

Furthermore, the performance of the Hybrid equi-join algorithm could be greatly improved through the use of a concurrent vector, such as the Thread Building Blocks (TBB) `tbb::concurrent_vector`, or by utilising the `std::mutex` class to manage thread synchronisation of modification operations to the STL `std::vector` [21, 22].

For future development, each of the benchmarks run above should be repeated multiple times and the results should be averaged. This will allow for more accurate benchmarks to be acquired as it will minimise the effect of outliers in the results.

8. CONCLUSION

The design and implementation of two different equi-join algorithms were presented. The first was a Hybrid model using MPI and OpenMP, that made use of a simple hash join to implement the equi-join. The second implemented the equi-join using the Phoenix++ MapReduce paradigm. Benchmarks were run by varying the input file size, along with the number of nodes and the number of threads being used by the two solutions. It was discovered that for input text files with sizes less than 5 GB, the MapReduce solution had superior performance. However, for larger file sizes, the Hybrid solution performs better. The Hybrid Solution was also found to be scalable over increasing node counts. For future work, a concurrent vector should be used in the Hybrid model to allow for parallel querying, thus drastically improving performance.

REFERENCES

[1] S. Stanczyk, B. Champion, and R. Leyton. *Theory and practice of relational databases*. CRC Press, 2001.

[2] A. Pigul. "Comparative Study Parallel Join Algorithms for MapReduce environment." *Proceedings of the Institute for System Programming*, vol. 23, pp. 285–306, 2012.

[3] T. Zurek. *Optimisation of Partitioned Temporal Joins*. PhD, University of Edinburgh, Edinburgh, Nov. 1997. URL `http://www.dcs.ed.ac.uk/home/tz/phd/index.htm`.

[4] C. Barthels et al. "Distributed Join Algorithms on Thousands of Cores."

[5] P. J. Blanas S. "Memory Footprint Matters: Efficient Equi-Join Algorithms for Main Memory Data Processing."

[6] D. A. Schneider and D. J. Dewitt. "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment." *ACM SIGMOD Record*, vol. 18, no. 2, pp. 110–121, 1989.

[7] E. Schikuta. "Performance Modeling of the Grace Hash Join on Cluster Architectures." *Proceedings of the Parallel and Distributed Processing Symposium*, 2003.

[8] IBM. "The message passing model." URL `https://www.ibm.com/support/knowledgecenter/en/SSF4ZA_9.1.3/pmpi_guide/mesage_passing_model.html`.

[9] N. K. Lee et al. "Implementation of Parallel Collection Equi-Join Using MPI." *International Workshop on Applied Parallel Computing*, pp. 217–226, 2002.

[10] S. J. Kang et al. "Performance Comparison of OpenMP, MPI, and MapReduce in Practical Problems." *Advances in Multimedia*, vol. 2015, 2015.

[11] C. Barthels et al. "Distributed Join Algorithms on Thousands of Cores." *Proceedings of the VLDB Endowment*, vol. 10, pp. 517–528, 2017.

[12] B. Kuhn et al. "OpenMP versus threading in C/C++." *Concurrency - Practice and Experience*, vol. 12, no. 12, pp. 1165–1176, 2000.

[13] T. Sterling et al. *High Performance Computing. Modern Systems and Practices*. Morgan Kauffman Publishers, 2018.

[14] CSinParallel Project. "Introducing Students to MapReduce with Phoenix++ Documentation." 2017.

[15] G. Bosilca et al. "On Scalability for MPI Runtime Systems." 2011.

[16] The Open MPI Project. "Compiling MPI applications.", 2017. URL `https://www.open-mpi.org/faq/?category=mpi-apps`. Last Accessed: 10/05/2018.

[17] T. O. Group. "64-bit and Data Size Neutrality.", 2015. URL `http://www.unix.org/whitepapers/64bit.html`. Last Accessed: 10/05/2018.

[18] A. Unknown. "Containers library.", 2018. URL `http://en.cppreference.com/w/cpp/container`. Last Accessed: 10/05/2018.

[19] S. Rajasekaran and J. Reif. *Handbook of Parallel Computing: Models, Algorithms and Applications*. CRC Press, 2007.

[20] "How Random Access Memory (RAM) affects performance.", 2017. URL `http://www.dell.com/support/article/za/en/zabsdt1/sln179266/how-random-access-memory-ram-affects-performance?lang=en`. Last Accessed: 14/05/2018.

[21] Intel. "concurrent_vector.", 2016. URL `https://software.intel.com/en-us/node/506203`. Last Accessed: 10/05/2018.

[22] A. Unknown. "std::mutex.", 2016. URL `http://en.cppreference.com/w/cpp/thread/mutex`. Last Accessed: 10/05/2018.