Comparing Message Passing and Multi-threading models for a Merge Join Algorithm

Arlo Eardley (1108472), Carel Ross (1106684) and Ryan Verpoort (1136745)
School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg 2050, South Africa

***Abstract*—This report describes and compares a Message Passing Interface (MPI) model with a multi-threading model to do an inner merge join on multiple tables. MPICH 3.2.1 is used for the MPI model and OpenMP 4.5 is used for the multi-threading model. The algorithm uses a merge sort to sort the tables and then joins the entries that have a common key. The MPI model utilizes the shared memory and efficiently allocates processes. The multi-threading model uses multiple fork-joins which increase the computation time due to the recursive nature of the algorithm. This means that for this implementation of the parallel inner merge join of tables, the MPI model is the most effective.**

I. Introduction

The aim of this report is to describe and compare a Message Passing Interface (MPI) model with a multi-threading model to do an inner merge join of multiple tables. MPICH 3.2.1 is used for the MPI model and OpenMP 4.5 is used for the multi-threading model. A literature review is conducted in Section II, the design and implementation is discussed in Section III, the results are discussed in Section IV, a critical analysis in Section V, limitations and improvements in Section VI and finally Section VII concludes the paper.

II. Literature Review

A. Tables

Database tables consist of rows and columns. Some columns contain keys, which are used to uniquely identify records in a table. Tables usually have primary keys to identify each record and is stored in an index [1]. Efficient database design would involve the use of multiple tables that are linked via foreign keys [2]. The tables containing foreign keys can have a one-to-one, one-to-many or many-to-one relationship with the table that contains the primary keys [3]. Duplicate entries of a foreign key is acceptable within a table, however all primary keys are unique within the same table. Figure 1 shows a one-to-one relationship between Table A and Table B are constructed.

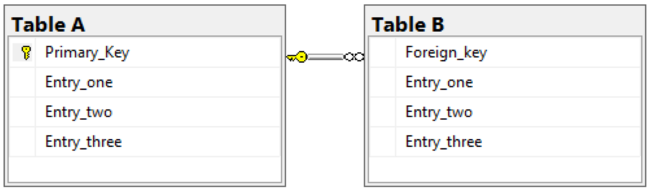


Fig. 1. Figure showing the one-to-one relationship of Table A and Table B

B. Inner Join

Within a set of tables, there needs to be a pair of matching columns that are based on a join attribute. For example, table A and table B both have a column of keys (these will be a set of primary/ foreign key pairs), this will be the join attribute [1]. The query then compares each record in A and B and joins the record entries, where the primary/ foreign key pairs are the same, into a new table C [4]. The records that are not common are not included, as described by the Venn diagram in Figure 2, this is known as an equi-join.

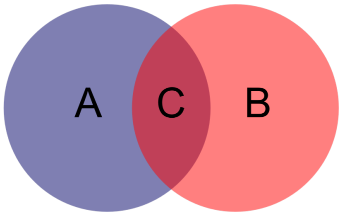
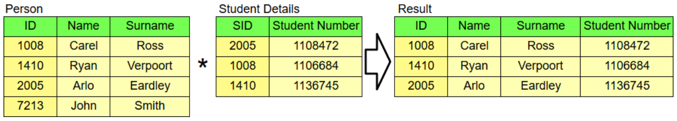


Fig. 2. Figure demonstrating the relations of table A, B and for an inner join

A few approaches to completing an inner join are as follows: Nested loop join, hash join and merge join. Figure 3 shows an example of an inner join of table A and table B, which result in the new table C.



Person

ID	Name	Surname
1008	Carel	Ross
1410	Ryan	Verpoort
2005	Arlo	Eardley
7213	John	Smith

*

Student Details

SID	Student Number
2005	1108472
1008	1106684
1410	1136745

Result

ID	Name	Surname	Student Number
1008	Carel	Ross	1108472
1410	Ryan	Verpoort	1106684
2005	Arlo	Eardley	1136745

Fig. 3. Figure showing an example of an inner join

C. Nested Loop Join

With the focus orientated towards the inner join algorithms, the nested loop join compares the primary key of each record in one table with the foreign key of each record in another table [5]. The matching primary/ foreign key pairs records are combined into the output table. This algorithm does not require any data structure information, however is very CPU intensive as its time complexity is O(N^2) [6]. This is due to the nested loop nature of the algorithm. This method is very memory efficient however, since the data can be directly operated on and does not require much memory [7]. This type of algorithm is primarily used for small subsets of data. This algorithm is commonly used with the resulting output table being smaller than 5000 rows [4]. With larger sets of data, the processing aspect of the algorithm will become very expensive and slow.

D. Hash Join

The hash join algorithm builds a temporary hash table of the inner table. The outer table is then scanned, while probing the hash table to join the appropriate rows [5]. This algorithm is primarily used for large tables that do not necessarily have common primary/ foreign key pairs [7]. This therefore uses a hash algorithm to create a common key based on the join condition to join the records of the tables. This improves computational time, however uses more memory as a result. The worst case time complexity of this algorithm is O(N^2), however the average case has a time complexity of O(N) [6]. This algorithm is very effective with relatively large data sets that have some sort of common attribute, but do not have predefined keys [4].

E. Merge Join

The merge join algorithm, merges tables together. The tables are joined by merging the primary/ foreign key pairs records that are presorted on the join attribute (in this case the primary/ foreign key pairs)[5]. This algorithm efficiently uses computational power and memory [7]. If the tables are presorted, then the maximum time complexity of the algorithm is O(N)[6]. The merge join algorithm also uses a minimal amount of memory to achieve merge tables. If the initial table is not sorted, then the complexity increases based on the sorting algorithms complexity. This algorithm is very effective with large data sets and is computationally very efficient [4].

F. Merge Sort

The merge join algorithm requires a sorted data set to effectively operate. The data therefore needs to be sorted first if it is in an unsorted state. The merge sort algorithm is a type of divide and conquer method that efficiently sorts large sets of data [8]. Merge sort continuously divides the array into equal halves until there are only pairs. The elements within the pairs are then compared and ordered accordingly. These new ordered sub-sets are then partially reconstructed. This process is recursively done until the array is rebuilt in an ordered manner [9]. For odd sized data sets, when the array is divided until a single element, then that element on its own is considered sorted as it has no other value to compare to. This process is demonstrated in Figure 4.

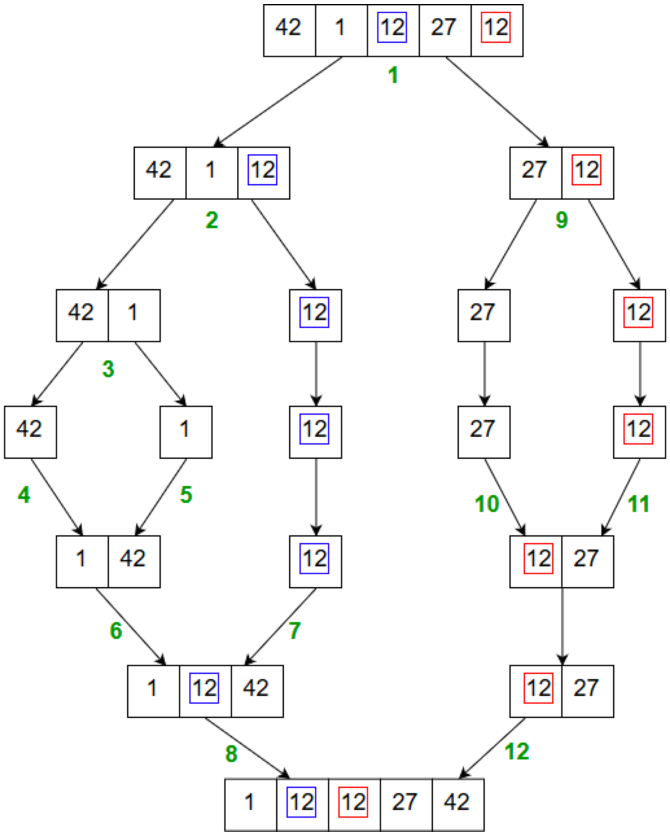


Fig. 4. Diagram illustrating the process of a merge sort algorithm

Figure 4 also illustrates the stable nature of the merge sort algorithm. This means that the repeated value of 12 remains in the same order in the array. Merge sort ensures that the 12 with the blue square will always be ordered first in the array, before the 12 with the red square, therefore proving its stability. The green text in Figure 4 demonstrates the order of processes. Merge sort has a best, average and worst case time complexity of O(nlogn). It also has a memory complexity of O(n) [10]. This can be improved to an O(1) memory complexity however, by using a hybrid block merge sort. The merge sort is also in theory parallelizable up to a time complexity of O(logn), as described by Coles parallel merge sort theory [10].

G. OpenMP

OpenMP is an Application Program Interface (API) that directs multi-threaded shared memory parallelism. Multiple threads can exist within a single process [11], however during the fork there are other threads that execute in parallel. When the threads complete their tasks, they synchronize and terminate [12]. This is known as a join. The master thread is now a single thread and process executing sequentially again [13]. This process is illustrated in Figure 5.

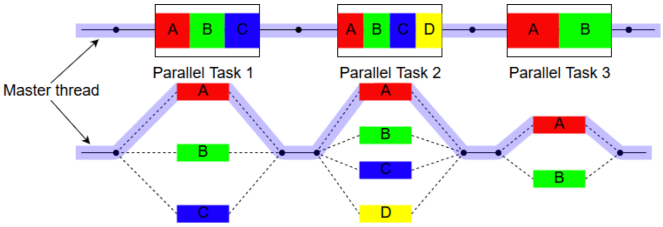


Fig. 5. Diagram showing the threading model OpenMP uses

Multiple fork-join processes can occur, depending on the code that needs to execute, as shown in Figure 5 [14]. The programmer is responsible for ensuring the I/O is correctly executed and that shared variables are to be explicitly defined [15].

H. MPI

Message Passing Interface (MPI) is a parallel programming model that is used in high-performance computing applications. Data is communicated between multiple processes via a communicator, where each process is assigned a unique rank [16]. Point-to-point communication can be described as follows: Each process can communicate with another process by the process rank. This communication allows processes to send and receive operations and messages to another process. Messages are sent by specifying the process rank and unique message tag [10]. A different process will then probe for a given message and handle the incoming message according to the process definition.

MPI does not require the master node to manage and control all communication with worker nodes. This means that the worker nodes can also communicate with one another to increase communication and process efficiency [17]. Collective communication is possible however, where the master node broadcasts messages to the worker nodes and receives an according response from the workers.

Point-to-point and collective communication are key aspects to the operations of MPI and can be used to manage and control efficient parallel processes [10]. MPI allows for multi-node processing. As shown in Figure 6 each node has its own memory and a number of processes.

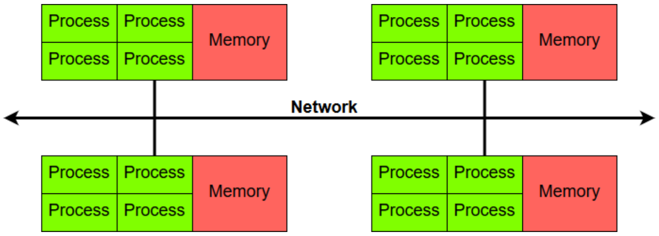


Fig. 6. Diagram describing the distributed memory and shared memory frameworks of MPI

These nodes can be linked into a larger network of distributed memory. This means that the MPI model can operate on either a distributed memory, shared memory or hybrid framework.

I. Select-Project-Join operations

Relational databases utilize the Select, Project and Join operations. These functions manipulate the data and performs select, insert, update and delete functions.

1) Select Operation: The select operation filters a table and removes rows that do not satisfy a selection condition [18]. As shown by Figure 7 the table on the left consists of five rows (R1-R5). Based on the selection condition, only rows R1 and R3 are kept. Rows R2, R4 and R5 did not meet the selection condition and are therefore discarded [19]. This is considered as a horizontal filter of the relation.

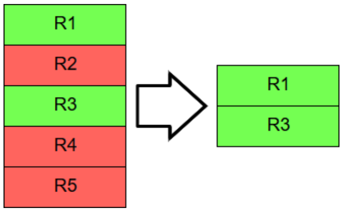


Fig. 7. Diagram demonstrating a select operation

2) Project Operation: The project operation is similar to the select operation as it also filters a table based on a condition. The project operation removes columns instead of rows however [18]. As shown by Figure 8 the table of the left consists of five columns (C1-C5). Based on the condition, only columns C2 and C4 are kept. Columns C1, C3 and C5 did not meet the condition and are therefore discarded [19]. This is considered as a vertical filter of the relation.

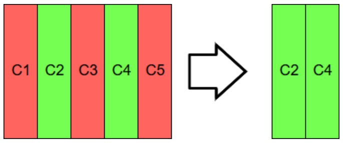


Fig. 8. Diagram demonstrating a project operation

3) Join Operation: The join operation merges independent tables that share a common column into a single table. With an inner join all the common entries are kept, while the other entries are discarded [18]. As shown by Figure 3 the table on the left consists of three columns and the middle table consists of two columns. These tables have a common column ID. When the tables are joined the common entries are kept and joined, while the other entries are discarded as illustrated by the table on the right [19]. The ID column is considered as the join condition and in this case is a common key to the related table.

III. Design and Implementation

The algorithm implemented is the inner join merge sort of an unsorted table. This means that the algorithm will sort the input data, before joining the tables. Selection, bubble, insertion and merge sort algorithms are all considered and compared, however the parallelization capability, stability and time complexity efficiency of merge sort indicates the advantage of this particular sorting method. The merge sort algorithm is implemented in both the multi-threaded and MPI versions of the code.

The table data is read in serially as it is prohibitively more difficult to read the table data in parallel when files contain an unknown amount of data for every line. If during the reading in of table data, an error occurs while indexing the file, the entire algorithm is stopped rather than correcting the error. This is not included in the time calculation as it will just introduce errors into the comparison between the multithreading and MPI approach. Once the tables are read and moved to memory, the data is sorted, joined and stored in an output file, all of which are done in parallel.

The input and output files are assumed to be in a CSV file format. The data is also assumed to have one to one relationships between the tables requiring all the keys to be unique within a table. The key also is required to be the first column of the table and an integer value that is limited to 10 characters. In both implementations, the data from the tables are read into a struct array. The struct consists of a Key which is the integer value, and a Value which is the proceeding data of the entry excluding the initial comma after the key.

A. OpenMP

This algorithm can join an undetermined number of tables into a single large table. It is however limited to the amount of contiguous memory available, since the data is read into a single large array. To describe the algorithm it will be assumed that two tables are used as input, which will be named A and B for the remainder of the discussion. The files containing the table data are opened and read into the struct array in a serial fashion. This results in the data for table B being appended to the data of table A in the newly created array. Once the array has been populated with all the required table data, the array is then run through a merge sorting algorithm which recursively breaks down the size of the array in order to perform the sort in the form of tasks. OpenMPs tasks are chosen since nested parallelism with recursion can cause more active threads than there are available CPUs. This will decrease the computational time of the parallel operation. By using OpenMPs tasks, manual handling of the threads parameter is not required since the threads will match the number CPUs available. These threads will therefore effectively work on the tasks.

The division of the array data is done in parallel using multithreading, resulting in threads being assigned portions of the original array during the recursive process. Once the array has been split up into chunk sizes of one thousand entries or less, the division process of the merge sort is no longer split across threads, instead the thread that received the chunk size now runs the rest of the division through to the sorting of the array data in serial on the single thread. This is due to the assumed size limit of one thousand entries being able to run much faster in serial on a single thread. This limits the amount of resources that are required to be allocated when assigning a single thread to each entry in the array had the entire merge sort been done in parallel.

Once all the threads are finished sorting their chunks of the original array, they are all combined together in parallel. Once the array has been sorted. An O(N) operation is performed to search through the array in parallel to find all the elements that match. This operation only has to be performed once due to the sorted nature of the final array. When elements are found to match they are outputted as the final result of the table join. Due to the stable nature of merge sort, the files columns will be appended in the order that the files were input. The results may not be in order due to some threads finishing faster than others based on their allocated workload and output computation time. This merge join algorithm therefore has a time complexity of Nlog(N) + O(N).

B. MPI

This algorithm can join an undetermined number of tables into a single large table. It is however limited to the amount of contiguous memory available, since the data is read into a single large array. At some point each node will also receive the full array, which means that each node also requires the same amount of contiguous memory. A MPI data type is created that matches the struct type described above. This allows the MPI module to communicate this data type between nodes and processes. The root node initiates and controls the I/O of the program.

To describe the algorithm it will be assumed that two tables are used as input, which will be named A and B for the remainder of the discussion. The files containing the table data are opened and read into the struct array in a serial fashion. This results in the data for table B being appended to the data of table A in the newly created array. Once the array has been populated with all the required table data, the array is then run through a merge sorting algorithm. Processes are then explicitly mapped to nodes in the recursion tree. This forms a virtual process tree where the root process is process 0. All other processes are known as helper processes. These helper processes communicate with each other to sort the algorithm by sending portions of the array to a child process. This allows the child process to receive and sort its segment of the array. The child process then sends the new sorted array back to the parent process. Figure 9 illustrates the process tree of a system with 4 processes.

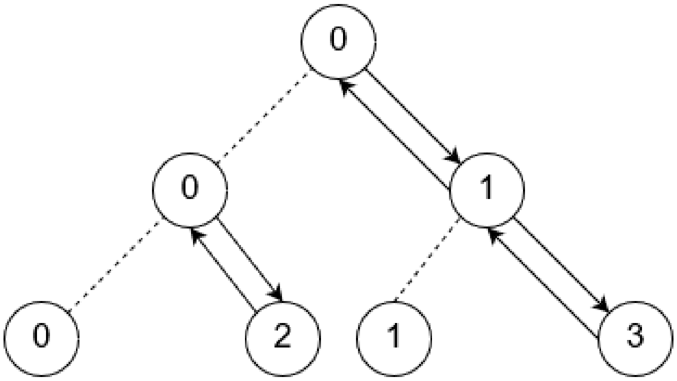


Fig. 9. Diagram showing the virtual process tree of the merge sort algorithm

The arrows demonstrate communication with a particular process, and the dotted line represents data stored for further use by the process. This process tree is further described as follows:

- The data is sent from process 0 to process 1
- Process 1 then recursively sends the data to process 3
- Process 3 sorts the data and replies to process 1 with a sorted array
- Process 1 sorts its data and replies to process 1 with the sorted half
- While this all happens process 0 also sends the data to process 2 which sorts the array and replies to process 0 with the sorted array.

The two halves of data are merged by its parent process. The division of the array data is done in parallel using MPI, this means that processes are being assigned portions of the original array during the recursive process. Once all the processes are finished sorting their chunks of the original array and finally a single sorted array has resulted in the root node, the root node broadcasts the data to all helper nodes directly. An O(N) operation is performed in parallel to search through the array to find all the elements that match. This operation only has to be performed once due to the sorted nature of the final array. When elements are found to match they are outputted as the final result of the table join. Due to the stable nature of merge sort, the files columns will be appended in the order that the files were input. The results may not be in order due to some threads finishing faster than others based on their allocated workload and output computation time. This merge join algorithm therefore has a time complexity of Nlog(N) + O(N).

IV. Results

Multiple Sets of data are run to compare the efficiency of each approach. The data sets are described as follows:

- Generated tables A and B each of the following sizes: 1,000, 10,000, 20,000, 40,000, 80,000, 120,000, 160,000, 200,000, 240,000 and 260,000
- Varying number of threads and processes: 2, 4, 6 and 8

These tables are limited in size due to the lack of contiguous memory available. The Central Processing Unit (CPU) used is an Intel Core i5-8400 Processor. This processor has a 9MB of cache memory, it also has 6 cores and 6 threads. This means that when 8 threads or processes are allocated, then two of the threads or processes will be virtual. The rest of the system specifications are as follows:

- RAM: 7.7 GiB
- CPU: Intel Core i5-8400 CPU @ 2.80GHz 6
- GPU: GeForce GTX 1050 Ti/PCIe/SSE2
- OS TYPE: 64-bit
- HDD: 35.6 GB

These are compared to compare the efficiency of physical and virtual threads and processes as well. Each entry requires 14 bytes of data. Figure 10 and Figure 11 show the various aspects discussed. This includes the relations of file size, computational time, number of threads/ processes and the multiprocessing model.

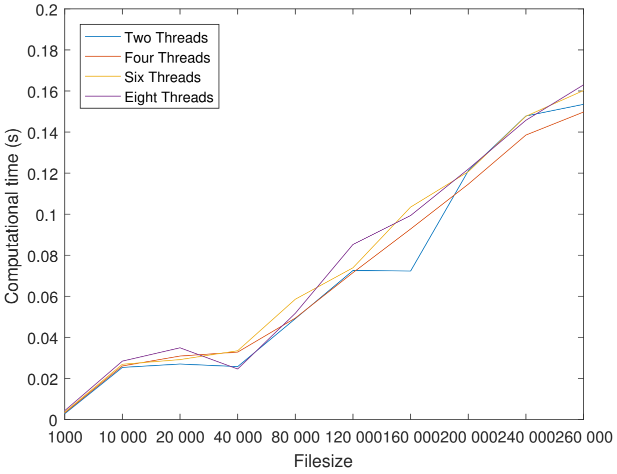


Fig. 10. Figure showing the computational time to file size for various number of threads using OpenMP

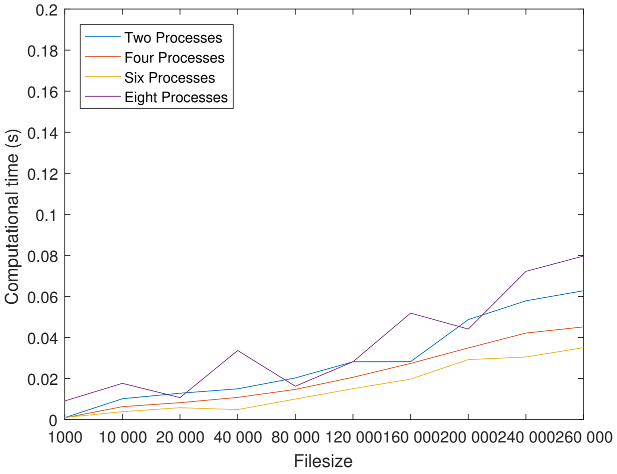


Fig. 11. Figure showing the computational time to file size for various number of processes using MPI

V. Critical analysis

The contiguous memory required for the final calculation is very large and limits the location that the data can be in memory. This means that a file with a maximum of 260,000 entries can be joined without causing a segmentation fault on the current hardware.

As shown by Figure 10, as the number of files increase, so does the amount of time required for computation. It also shows that for smaller file sizes, the computational time is faster for fewer threads. This is due to more computation being performed serially, which increases performance. As the file size increases to 260,000, the most efficient computational time is with four threads. If even larger file sizes are tested, then six threads will eventually execute faster than the rest. When the number of threads is eight however, the computational speed will not necessarily increase, since two threads are virtual. This means that it does not physically add more computational power to the system.

As shown by Figure 11, as the number of files increase, so does the amoun of time required for computation. It also shows that the number of physical processes indicate the speed of the computation. Since two physical processes is always slower than 4 physical processes. When 6 physical processes are used then the computation speed it at its most efficient. When virtual processes are introduced, such as with eight processes, the computational time has an extreme increase. This means that it is less efficient to use virtual processes and actually limits the computational strength of the parallel program.

Comparing the two graphs, the MPI model is consistently faster than the multithreading model. This is due to the MPI model utilizing shared memory and all the processes efficiently. The multithreading model however, continuously uses fork-joins which increases the computational time of the process. This means that for this implementation of the parallel inner merge join of tables, the MPI model is the most effective.

VI. Limitations and Improvements

The performance limitations of the multithreaded algorithm include the maximum amount of data allowable inside an entry in the table is limited to 10 characters (excluding the key and initial comma). This limits the amount of entries that are able to be joined. This design choice restricts the size of the array in memory which means that the amount of contiguous memory required is reduced.

The topology relies on having a large block of contiguous memory available for the allocation of the array as it is not a dynamic array. This can be altered into a pointer array to ensure all the memory is utilized for the merge join operation. The I/O aspect of the MPI design can be improved by ensuring that each process is sent only the data that it needs to output instead of broadcasting the entire array to all processes.

The maximum number of entries in a table is limited to the maximum size of an integer. This means that a maximum of 2,147,483,647 entries can be manipulated and stored, assuming enough contiguous memory is available.

When the output files become excessively large, some errors may occur during the parallel I/O of the system. Therefore a more robust error handling system should be implemented.

The input could also be parallelized by using memory mapping. This is a technique whereby the location of the file is allocated as virtual memory. This virtual memory will then load into physical memory while computation is already conducted. This will increase the read file time.

VII. Conclusion

Both implementations are scalable and are able to successfully execute a inner merge join of a number of tables. The total time complexity of the merge join in both cases is O(NlogN + N). Eventhough both have the same time complexity, the MPI model utilizes the shared memory more efficiently while allocates processes. This is partially due to the multi-threading model using multiple fork-joins due to the recursive nature of the algorithm, which increase the computation time. This means that for this implementation of the parallel inner merge join of tables, the MPI model is the most effective. The formatting of the input file is very important to ensuring the running of the application. Some improvements can be made, however the system is limited to the physical hardware.

References

[1] "what is the difference between a primary key and a foreign key?." https://www.essentialsql.com/what-is-the-difference-between-a-primary-key-and-a-foreign-key/, 2018. [Accessed 15-April-2018].

[2] "table keys — database primer." urlhttp://www.databaseprimer.com/pages/keys/, 2018. [Accessed 19-April-2018].

[3] "primary and foreign key constraints." https://docs.microsoft.com/en-us/sql/relational-databases/tables/primary-and-foreign-key-constraints?view=sql-server-2017, 2018. [Accessed 19-April-2018].

[4] D. Dieter, "Sql server join algorithms." http://sqlserverplanet.com/optimization/sql-server-join-algorithms, 2018. [Accessed 28-April-2018].

[5] "how joins work." https://www.periscopedata.com/blog/how-joins-work, 2018. [Accessed 02-May-2018].

[6] G. Kokosinski, "The fundamentals: join algorithms." http://prestodb.rocks/internals/the-fundamentals-join-algorithms/, 2018. [Accessed 28-April-2018].

[7] "ibm knowledge center." https://www.ibm.com/support/knowledgecenter/en/SSEP7J_10.1.1/com.ibm.swg.ba.cognos.vvm_user_guide.10.1.1.doc/c_join_algoritms.html, 2018. [Accessed 27-April-2018].

[8] "11.4 mergesort." http://www.mcs.anl.gov/~itf/dbpp/text/node127.html, 2018.

[9] "data structures and merge sort algorithms." https://www.tutorialspoint.com/data_structures_algorithms/merge_sort_algorithm.htm, 2018.

[10] A. Radenski, "Shared memory, message passing, and hybrid merge sorts for standalone and clustered smps," *In: The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 367–373, 2011.

[11] R. Rabenseifner, G. Hager, and G. Jost, *Hybrid Parallel Programming Hybrid MPI and OpenMP Parallel Programming*. 2013.

[12] *openmp 4.0 api c/c++ syntax quick reference card*. OpenMP, 4 ed., 2013.

[13] *lecture 12: introduction to openmp (part 1)*. 2018. [Accessed 23-April-2018].

[14] C. Barthels, I. Mleler, T. Schneider, G. Alonso, and T. Hoefler, "Distributed join algorithms on thousands of cores," *Proceedings of the VLDB Endowment*, vol. 10, no. 5, pp. 517–528, 2017.

[15] "parallel i/o introductory tutorial - documentation." https://www.sharcnet.ca/help/index.php/Parallel_I/O_introductory_tutorial, 2018. [Accessed 12-April-2018].

[16] "mpi tutorial introduction mpi tutorial." http://mpitutorial.com/tutorials/mpi-introduction/, 2018. [Accessed 12-April-2018].

[17] "mpi - c examples." http://people.sc.fsu.edu/~jburkardt/c_src/mpi/mpi.html, 2018. [Accessed 19-April-2018].

[18] "a model for sql - select project join operation." http://www.remote-dba.net/t_op_sql_model.htm, 2018. [Accessed 18-April-2018].

[19] "openstax cnx." https://cnx.org/contents/NAA4OPSt@1/Relational-Algebra, 2018. [Accessed 24-April-2018].