

ELEN 4020 Project - Comparison of Parallel Equi-Join using MPI and OpenMP

Uyanda Mphunga – 1168101, Darren Blanckensee – 1147279, Ashraf Omar – 710435
and Amprayil Joel Oommen – 843463

Abstract—Join operations are the most intensive operations in database queries. This project aims to compare two different approaches to performing an equi-join of two very large tables. The first approach is to use OpenMP to perform a merge join and the second is to use MPI to perform a hash join.

is described in Section II and the two different approaches discussed in Sections III and IV. The method of experimentation is outlined in Section V and the results of the experiment are analysed in Section VI. Section VII reviews the project and concludes this document.

I. INTRODUCTION

The join operation concerns the combining of two different tuples on a common join attribute [1]. It is important in information systems, in particular the use of databases since the join operation is the most expensive operation in database query operations in terms of time and data-intensity [2]. This project explores two different approaches to performing a join of two very large tables. The purpose of the project are twofold, the primary task is to use parallelism to implement two different join algorithms and the secondary is to compare the algorithms efficiency in joining. The problem

II. PROBLEM DESCRIPTION

The objective of this project is to perform an equi-join of two very large tables. An equi-join is a type of join that uses the equality operator as a basis for the join [3] that is if the join attribute in $R_1(A, B)$ is strictly equal to the join attribute in $R_2(A, C)$ the result of the join is inserted into a third table $R_3(A, B, C)$. An illustrated example can be seen in Figure 1. The join must be done using two different algorithms, one that is based in MPI (Message Passing Interface), and one that uses another high-level parallel programming model. The programming model chosen in this project is OpenMP. The two programs need to be compared with one another in terms of speed-up and scalability when increasing the number of

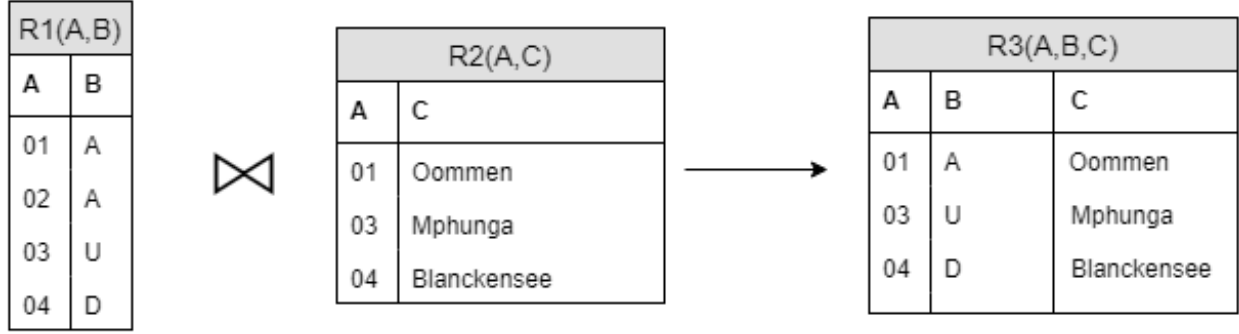


Fig. 1. An example of equi-join between two relational tables

processors and nodes that the program uses. The speed-up comparison is performed by running the two programs with increasing number of processors and the scalability comparison is conducted by increasing the number of nodes the program uses in a cluster. Based on the literature, there are multiple join algorithms. The algorithms chosen by the authors are the hash algorithm and the merge algorithm, which form part of the equi-join family of join algorithms and are widely used [4].

The join algorithms are designed to read data from file of a specific format and to perform the required operations as per specific algorithm. The algorithms read data of the same file in order to be able to compare their performance. The algorithms are both implemented in C++. This allows for accurate comparisons of the algorithm performances and consequently prevents the possible time lags that may arise from the use of different programming languages. C++ was used instead of C due to the authors finding easier to use C++ under the given time constraints. Another reason that why C++ was found more favourable is that it offers

more functionality than C and due to its Object Orientated Design (OOD) capabilities and the also due to the fact that the authors are familiar with C++. These algorithms were implemented using parallelism.

III. OPENMP JOIN ALGORITHM – MERGE-JOIN

The OpenMP join algorithm being implemented is merge-join also known as sort-merge join [5]. The two table are sorted by join attribute. Then the table are scanned and join attributes are compared to one another. For an equi-join, if the join attribute of one entry is strictly equal to the join attribute of the other table's entry, the entries are joined in the output table [6], [7]. The implemented algorithm assumes that the sorting of the data has already taken place and proceeds to only perform the join algorithm under that assumption.

References [4], [8] take into account possible errors that may occur due to data skew for the merge join algorithm. According to [4], data skew can result in some of the processors being over-utilised and others under-utilised. Reference [4]

proposes the use of a divide-and-conquer technique as a solution to handle data skew. Reference [8] approaches the issue of data skew in the merge algorithm by adding an extra scheduling phase. The implemented merge join does not take into account the possibility of data skew and instead relies on OpenMP's computational ability to process certain chunks of data and ability to make numerous threads [9]. This technique was chosen due to the resources available, namely a cluster, and time constraints.

An overview of the implemented code can be found in Algorithm 1. The program is implemented in C++ and makes use of OpenMP to implement parallelism. The “key” being referred to in this context is the join attribute and the “values” are the other two attributes.

Data: Table $R_1(A, B)$, Table $R_2(A, C)$

Result: $R_3(A, B, C)$

sort both tables on key;

read in keys from each table and store them in vectors;

for all key-value pairs in each vector **do**

if key from R_1 == key from R_2 **then**

 write key-value output to R_3 ;

end

end

Algorithm 1: OpenMP Implementation of merge-join

IV. MPI JOIN ALGORITHM – HASH-JOIN

A hash join can be generalised into two stages [10], the split phase and the join phase [11]. In the split phase the hash function is used to partition the data into a number of sets and in the join phase each partition is joined and the result is sent to a common table. Hash join algorithms are considered to be the most efficient join algorithms in terms of speed [2].

The reviewed literature for the hash join is also concerned about data skew. Some of the said literature include references [8], [10] and other documents. Reference [8] addresses the issue of data skew by implementing multiple hash phases along with the use of a heuristic optimisation algorithm in order to isolate skew elements[8].

An overview of the algorithm being used is listed in Algorithm 2. The program is implemented in C++ and makes use of MPI to send and receive messages between nodes. The hash function simply takes the sum of all the character's ASCII values to produce a unique hash value. As before, the “key” being referred to in this context is the join attribute and the “value” the other attributes.

V. EXPERIMENT DESCRIPTION

VI. ANALYSIS OF RESULTS

VII. CONCLUSION

Join operations are one of the most important operations in the use of databases, however, the operations are expensive in terms of time and data-intensity. In an effort to find an efficient algorithm,

Data: Table $R_1(A, B)$, Table $R_2(A, C)$

Result: $R_3(A, B, C)$

for all key-value pairs in R_1 **do**

 read key;
 calculate hash as sum(ASCII) of key;
 calculate (hash%(no. of processes - 1) + 1);

end

send key-value pairs of $R_1(A, B)$ to slave nodes in vector;

for all key-value pairs in R_2 **do**

 read key;
 calculate hash as sum(ASCII) of key;
 calculate (hash%(no. of processes - 1) + 1);
 //This is to determine which slave node the key-value pair is sent to.

end

send key-value pairs of $R_2(A, C)$ to slave nodes in vector;

IN EACH SLAVE NODE:

for all key-value pairs in each vector **do**

if key from R_1 == key from R_2 **then**
 write key-value output to R_3 ;

end

end

Algorithm 2: MPI Implementation of hash-join

two approaches were implemented and compared. The first approach uses OpenMP to perform a merge join on two large tables. The second performs a hash join using MPI as a basis for the code.

REFERENCES

- [1] C. Yu and W. Meng, *Principles of Database Query Processing for Advanced Applications*, ser. The Morgan Kaufmann series in data management systems. Morgan Kaufmann, 1998. [Online]. Available: <https://books.google.co.za/books?id=aBHRDhrrehYC>
- [2] P. Mishra and M. H. Eich, "Join processing in relational databases," *ACM Comput. Surv.*, vol. 24, no. 1, pp. 63–113, Mar. 1992. [Online]. Available: <http://doi.acm.org/10.1145/128762.128764>
- [3] w3resource.com. (2018) Sql equi join. Last Accessed: 2018-05-14. [Online]. Available: <https://www.w3resource.com/sql/joins/perform-an-equi-join.php>
- [4] J. L. Wolf, D. M. Dias, and P. S. Yu, "A parallel sort merge join algorithm for managing data skew," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 70–86, Jan 1993.
- [5] G. Graefe, "Sort-merge-join: An idea whose time has(h) passed?" in *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*, Feb 1994, pp. 406–417.
- [6] A. Pavlo. (2017) Lecture 19 parallel join algorithms (sorting). Carnegie Mellon University. Online Lecture Slides. [Online]. Available: <https://15721.courses.cs.cmu.edu/spring2017/slides/19-sortmergejoins.pdf>
- [7] G. Graefe, "Heap-filter merge join: A new algorithm for joining medium-size inputs," *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 979–982, Sep. 1991. [Online]. Available: <https://doi.org/10.1109/32.92919>
- [8] J. L. Wolf, D. M. Dias, and P. S. Yu, "An effective algorithm for parallelizing sort merge joins in the presence of data skew," in *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems*, ser. DPDS '90. New York, NY, USA: ACM, 1990, pp. 103–115. [Online]. Available: <http://doi.acm.org/10.1145/319057.319072>

- [9] B. Barney. (2017) Openmp. Lawrence Livermore National Laboratory. Last Accessed: 2018-05-15. [Online]. Available: <https://computing.llnl.gov/tutorials/openMP/>
- [10] A. M. Keller and S. Roy, “Adaptive parallel hash join in main-memory databases,” in *[1991] Proceedings of the First International Conference on Parallel and Distributed Information Systems*, Dec 1991, pp. 58–67.
- [11] M. Kitsuregawa, S. Tsudaka, and M. Nakano, “Parallel grace hash join on shared-everything multiprocessor: implementation and performance evaluation on symmetry s81,” in *[1992] Eighth International Conference on Data Engineering*, Feb 1992, pp. 256–264.