ASHESI

ASHESI UNIVERSITY

PERFORMANCE ANALYSIS OF PARALLEL EQUI-JOIN IMPLEMENTATIONS

PARALLEL AND DISTRIBUTED COMPUTING

B.Sc. Computer Science

Mustapha T. Yussif and Jones Dari

2020

Abstract

This paper presents a hybrid programming implementation of equi-join and a MapReduce implementation. The hybrid programming model consists of MPI and OpenMP. The paper also conducts comparative performance analysis of the two solutions using scalability and speed up plots.

Introduction

The join operation is one of the fundamental relational database queries that facilitates the retrieval of information from two different relations using the Cartesian product of the two relations[5]. The join operation is the only relational algebra operation that requires joining of two relations [5]. The most frequently used join operation is the Equi-join. The equi-join is the join of source table S and a target table T on a an attribute A from S and B from T to form the result relation R by concatenating each tuple from s member of S and t member of T whenever the s[A] = t[B] [7]. The join operations in general are computationally expensive as a result, there are very difficult operations to implement efficiently[5]. Several parallel algorithms have been proposed to implement the equi-join operation. Some of these algorithms include: hashing, sort-merge, Grace hashing etc. In this paper, we discuss a variant of the hash join algorithm. This algorithm is discussed extensively later. A hybrid parallel programming model comprising OpenMP and MPI were used to implement the algorithm. Also, for comparative performance analysis, a MapReduce solution was also implemented using Mr Job. The design and implementation details of the two solutions are explain in the later sections.

Project Description

The main objective of this project is to develop and implement a parallel program for an equi-join of two very large tables (which are maintained as two independent files) on their common join attributes to produce another table and store the results in another file. This must be implemented with MPI-OpenMP hybrid parallel programming model. Additionally, a different programming model should be implemented and performance comparison between these approaches using speed up and scalability plots must be determined.

Assumptions

During the course of this project, some assumptions were made. Firstly, the number of processes were three. The first process was responsible for reading the input relations and distributing them to the other two relations. Secondly, we assumed that the three hash functions for filling the bloom filter would hit different distinct position. Thus, there will not result in collision. Lastly, when the join key is not the primary and where the chosen attribute is a duplicate in one or both the relation, the results is all possible permutation of the tuples for that join attribute.

Implementation

This section discusses the design and implementation of the equi-join algoirthm in this paper. The concepts (MapReduce, Bloom filter, MPI and OpenMP) used in this paper are briefly touched on. Also, one of the popular equi-join algorithms is explored before diving into explaining the main techniques used in this paper.

Bloom Filter

Bloom filter is a probabilistic data structure designed to memory-efficiently tell whether an item is present in a set or not. The base data structure of bloom filter is a bit vector. Size of the bloom filter impacts the false positive you get. A large size filter gives less false positives than a small size filter.

Hash Functions

Simple Hash function

This has function adds the Unicode of the characters in the key and finds the modulo "n" of the results. Where "n" is the maximum size of the bloom filter. to find the hash code. It is possible that two different keys (join attributes) will have the same hash code if the sum of the Unicode of its characters are the same. Further suppose that the bloom filter size is 1000. Suppose abc and bca are join attributes from table 1 and table 2 respectively. Summing the Unicode of the characters in abc is 294 (97 + 98 + 99) and bc is also 294 (98 + 99 + 97). Both keys will have the same hash positions even though they are different. To handle this collision, we multiply each Unicode number by the position of the character. We get (97 x 1) + (98 x 2) + (99 x 3) = for abc and (98 x1) +(99 x 2) +(97 x 3) = for bca. This result in different hash positions. However, when the number of join tuples in the relation increases, the possibility of having collisions increases. [ADD ALGORITHM HERE]

Djb2[6]

Its a string hash functions with fast speed and even distribution with any given keys and table sizes (bloom filter). It uses an initial hash code of 5381 which is tested and proven to have less collision. djb2 hash function uses shift operation, to make computation faster in modern CPUs.

Sdbm[6]

It is considered to perform well in scrambling of bits and with fewer splits. It has excellent distribution capabilities of keys. It uses an experimented value of 65599 which makes it more efficient because of its prime nature. Its a widely accepted algorithm used by Berkeley Lab.

OpenMP

OpenMP is a short form of Open specifications for Multi Processing of an API for explicit multi-threaded and shared memory programming[cite berkely resouce]. The OpenMP API has three concepts namely compiler directives, runtime library routines and environment variables. The compiler directives can be used to create teams of thread in order to multi-process a given task. The API makes use of the fork-join paradigm, where groups of threads are created (forked) to perform a task. Upon completion, these team of threads are joined together again to output a single result [cite berkely resouce]. In OpenMP, all the threads in the team share one common memory address location [cite berkely resouce].

MapReduce

After the release of MapReduce by Google, it has become the state-of-the-art for processing big data[8]. In its simple form, MapReduce comprises two phases; map phase, and reduce phase[8]. The map function takes as input each line in an input file and yields result in key-value pairs. In order implementations of the MapReduce, an additional phase called is added. This phase is responsible for hashing and sorting all the values associated with a particular key. However, the map function performs the work of the combiner if there is no combiner function. The reduce function finally aggregates the values associated with each key. At this point, depending on the task, programmers can manipulate the values to yield an expected outcome.

MPI

Message Passing Interface is an API for distributed memory programming[2]. With this API, processes can work on portions of a task as well as communicate with each other in order to exchange information [3]. In Message Passing, process share data between each other. If one process sends data to another, the receiving process must move the data from the sending process local memory to its own local memory [2].

Hash Join algorithm

Hash join is one the popular algorithms for performing equi-join. In hash join, a hash table is used to store the smaller table in key-value pairs manner [9]. This algorithm has two phase; build phase and probe phase[1]. The first phase is the build first. In the build phase, a hash function is applied on the attributes of the smaller table and store the table in key-value pair where the key is the hash code and values is the tuple from where the key was extracted. This phase is completed and the probe phase once all the values in the table are stored in the hash table. In the probe phase, the larger relation is read and the corresponding matching tuples that are likely to form a join with the smaller relation are looked up in the hash table and joined together[1]. Using hash table for store the values makes the algorithm efficient since the hash table hash is efficient for inserting and searching for items. In this paper, a variant of this algorithm is discussed. The difference between our implementation and the traditional parallel hash join is that, we employed bloom filter to reduce the number of communication between compute nodes.

Design

1. Partitioning: the computation to be performed and data to be used were divided into smaller tasks and distributed among the processes. In order to reduce the overhead by allowing each process to read the input files, process 0 was dedicated for that. It reads and distributes the relations to the other two compute nodes. Node 1 handles, requesting for possible candidate tuples from table 2 that are likely to form join with the tuples in table 1. It also does the joining. Node 2 sends the candidate tuples to node 1. Within each node, the task were further partitioned into smaller ones and executed by teams of threads.

2. Communication: all the three nodes communicate with one another. Node one (process 0) communicates with both node 1 and node 2 to send the tables. Node 1 and node 2 communicates with each other (after receiving the tables) to perform the equi-join. Refer to figure 1.

3. Aggregation: node 1 is made to perform the joining and write the joined relation to a text file.

Implementation

Hybrid Equi-Join

With the hybrid parallel equijoin implementation, a number of helper functions were created to modularize and make the work cleaner. As a result, three custom header files (filemanager.h, bloomFilter.h and queries.h) were created in this regard. Filemager header file responsible for reading input files and managing all I/O related task was created. Within this file, readfile function reads the content of an input file and stores the content as an array of strings by utilizing pointers. There is writeIntofile routine which takes an array of strings and dump it in a text file. bloomFilter.h manages all operations related to the hashing. Within bloomFilter.h, there are three hash functions/ These hash functions take the join attribute of the small relation and returns a corresponding hash value.

The master node (process 0) reads the two input files using the filemanager. Upon reading the files into arrays of strings, the node sends the small relation to node 1 (reason for sending the small relation to node one is explain later) and the larger relation to node 2. SendTable function was created to be responsible for sending the relation (array of string) using MPI_Send. The function first sends the total size of the relation to the receiving node first and follow by sending the size of each row and the row. It would have been nice to implement this in such a way that the relation will be send once. However, it was difficult sending the the entire relation because of how it was stored. Each node (node 1 and node 2) receive their respective tables

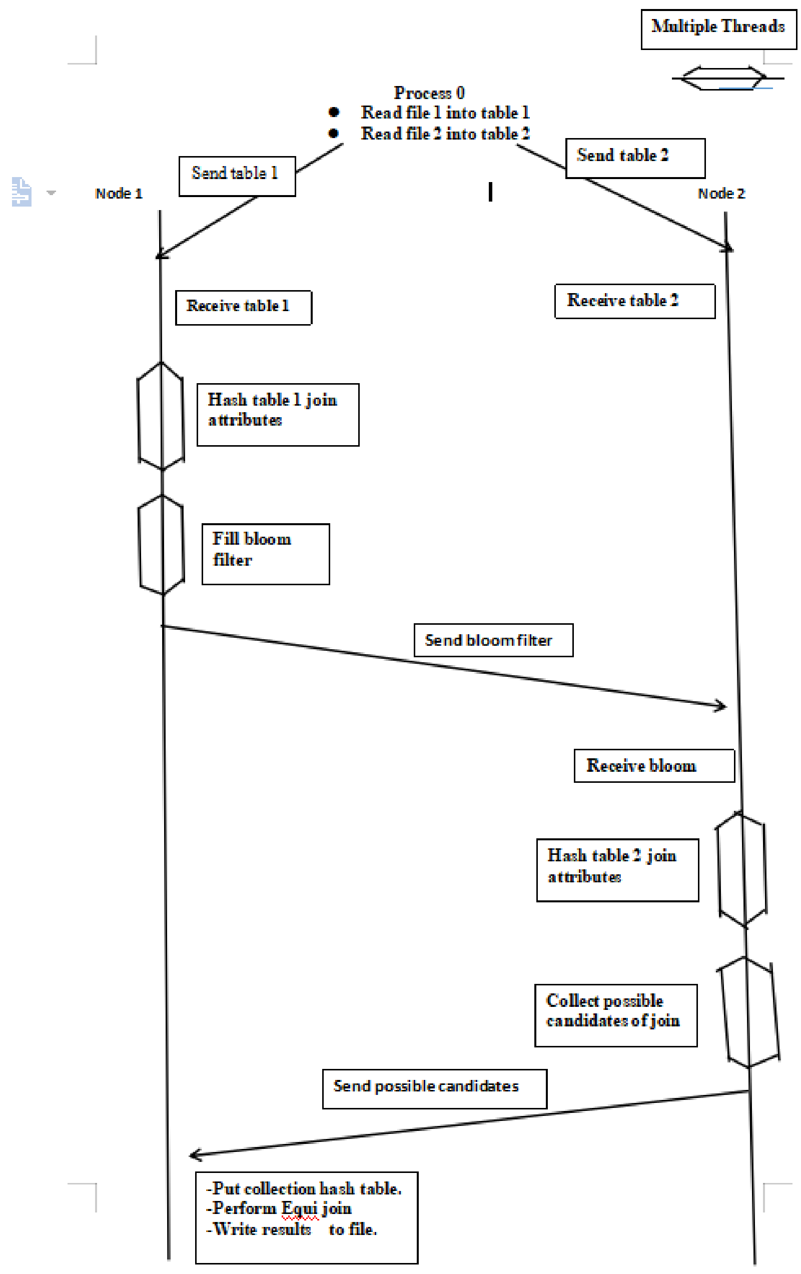


Figure 1: Hybrid Parallel design.

using a custom receiveTable function created. ReceiveTable first receives the total size of the table/relation using MPI_Recv. This will be used to determine the total number of rows in the table. The size of each row follow by the row itself are then received after the total size of the table is received. Having received them, node one hashes each value of the join attributes using the three hash functions discussed in this design. The outcome of these hash functions are numbers which are restricted to fall within [0, N], where N is the total size of the bloom filter (an array of integers initialized with zeroes). For every hash position in the bloom filter, the value is set to 1. Node one then sends the bloom filter to node two using MPI_Send.

After the node two receives the relation and the filled bloom filter, it repeats the same hashing exercise. If all the three hash positions are already set to 1s in the received bloom filter, the corresponding record in which the join attribute was extracted from is collected and save in array. All of such records on node 2 are collected and send to node 1 by utilizing the sendTable routine.

In order to avoid going through all the possible candidate to form the equi-join, the records are first stored in a hash table in ¡key, value¿ manner where the join attribute is the key and the value is the row from which the join attribute was extracted from. The hash table is a linked-list table where every index in the base array contains a linked list. This is to ensure when collision occurs because of duplicate join attribute, all the values of such attributes can still be stored in one location and retrieve using one key.

Node one then goes through the small relation sent to it. At this point, node 1 is dealing with the small relation and the possible candidates for the join. Both collections are small and therefore will be quite faster to go through. If the big relation was sent to node 1, node 1 would have to do more work by running through it, although the possible candidate for the join collection may be small. To reach join attribute (use as the key in the hash table) of the relation one, the corresponding tuples to form the join are merge together. By going through the relation, all the possible equi-join are form and save into a text file. The classical problem with this technique is collisions. Since we hash the values of the join attributes of R1 and R2, its not inconceivable that we can have h(v1)A=h(v2) for the different join attribute values, v1 and v2. A collision can lead to useless access of the bloom filter at the node-2 (at the second stage). To reduce collision, several hash functions are used, but all the hash functions are associated with one large bloom filter. Then for each join attribute value, the corresponding position in the bloom filter must be set to 1. The more hash functions used, the slower the bloom filter and the quicker the bloom filter fills up. If you have too small hash functions, however, you may suffer too many false positives. Function to help choose an optimal hash function = (m/n)ln(2).

MapReduce Equi-Join

The second implementation of the equi-join operation makes use of the mr job MapReduce framework. MapReduce is an industry standard framework for precosssing big data[4]. With this implementation, a mapper function and a reducer functions were used. The mapper function reads each line from both files in no particular order. At the map phase, the names of the files were determined using jobconf_from_env("map.input.file"). The key-value pair is then emitted. In the reducer function, the key-value pairs are separated into their respective tables. Equi-join is achieved by joining together, the data with the same key from both tables.

Experimental Results

The two solutions were bench-marked with different sized input files to compare their performance.

[In reporting the speed up and scalability of the parallel programs, we use the run-time of the serial program on which the parallel program was based on. This program was run on a single processor. No message passing or multi-threading functionality was added.] [For measuring the run time of the parallel program, we keep track of the start and end time on the master node. We then subtract the start time from the end time on the master node to derive the global time of the program.]

The solutions were bench-marked with different file sizes. Each file contains the same number of columns. The only different in those files were the number of rows. We have small, medium and large file size. The small files contained 11 and 10 lines, the medium files contained 6,485 and 6,023 lines. And the large files contained 12,365 and 12,560 lines.

The hybrid was run on three nodes (processes), in which the process 0 was the master node and the rest of the two nodes were compute nodes. 2, 3, 4, 5 and 6 threads were used each node had 2 threads when the program was ran for the first time. 3 threads were used per thread in the second run and so on.

RESULTS

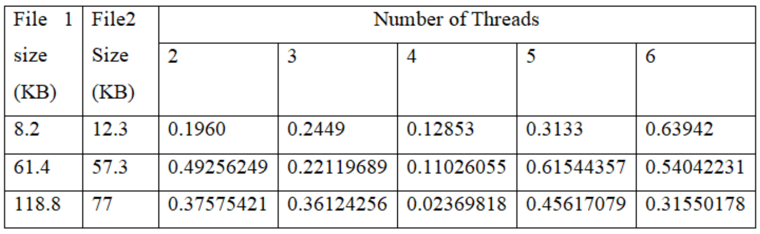
In reporting the speed up and scalability of the parallel programs, we use the run-time of the serial program on which the parallel program was based on. This program was run on a single processor. No message passing or multi-threading functionality was added. For measuring the run time of the parallel program, we keep track of the start and end time on the master node. We subtract the start time from the end time on the master node to derive the global time of the program. The solutions were bench-marked with different file sizes. Each file contains the same number of columns. The only different in those files were the number of rows. The program was tested on a Linux virtual machine with only 2GB RAM. We were limited by the size of file we could use to test. We have small, medium and large file size. The small files contained 40 and 80 lines, the medium files contained 487 and 430 lines. And the large files contained 947 and 615 lines.

The hybrid was run on three nodes (processes), in which the process 0 was the master node and the rest of the two nodes were compute nodes. However, the number of threads per process was varied in each execution of the program. We used 2, 3, 4, 5 and 6 threads per process. Each node had 2 threads when the program was run for the first time. 3 threads were used per thread in the second run and so on. The run times were measured. Table 2 summaries the run times for the hybrid program.

[TABLE 1 HERE]

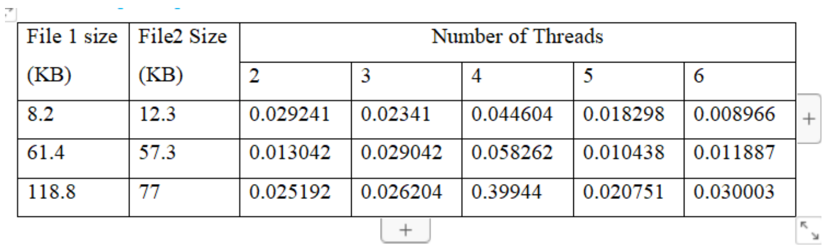
It was observed that the number hybrid run times were far worst than the serial version. It turned out that the hybrid solution was slow as compared to the serial version because the master thread sends the rows in the table instead of the entire table. Despite this unusual behavior this implementation was kept for comparison because we run out of time. As expected, the speed up values of the hybrid program were fractions less than 1. Table 2 summaries the speed up of the hybrid solution.

Also, we compared the run the MapReduce solution on the same input files and compare the run times of



File 1 size (KB)	File2 Size (KB)	Number of Threads				
		2	3	4	5	6
8.2	12.3	0.1960	0.2449	0.12853	0.3133	0.63942
61.4	57.3	0.49256249	0.22119689	0.11026055	0.61544357	0.54042231
118.8	77	0.37575421	0.36124256	0.02369818	0.45617079	0.31550178

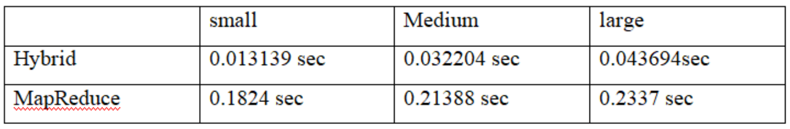
Figure 2: Hybrid Parallel design.



File 1 size (KB)	File2 Size (KB)	Number of Threads				
		2	3	4	5	6
8.2	12.3	0.029241	0.02341	0.044604	0.018298	0.008966
61.4	57.3	0.013042	0.029042	0.058262	0.010438	0.011887
118.8	77	0.025192	0.026204	0.39944	0.020751	0.030003

Figure 3: Hybrid Parallel design.

the two solutions. (see table 3)



	small	Medium	large
Hybrid	0.013139 sec	0.032204 sec	0.043694sec
MapReduce	0.1824 sec	0.21388 sec	0.2337 sec

Figure 4: Hybrid Parallel design.

Conclusion

The aim of this project was to implement a hybrid solution of the equi-join. Also, one of the requirements was to juxtapose the hybrid implementation with another solution (MapReduce) and perform performance analysis on them. The hybrid and MapReduce solutions were presented. Benchmarks were run by varying different file sizes and number of threads. It was discovered that the hybrid solution runs faster than the MapReduce solution bench-marked with the same input files. However, the hybrid solution performed poorly as compared to the serial version due to the reason explained above.

For future work, we will change how the relations (tables) are distributed to the compute nodes by the master node. We strongly believe by doing this, it will improve the performance significantly.

Appendix

How to run the MapReduce code

Python3.py mr_equi_join.py ¡path/to/R1.txt¿¡path/to/R2.txt¿ –column = ¡indexOfJoinColumn¿ ¿./R3.txt

Hybrid parallel

Compilation

```
mpicc ./main.c -o main -fopenmp -lgomp
```

Execution

```
mpiexec -n 3 ./main ¡path/to/R1.txt¿ ¡path/to/R2.txt¿ ¡indexOfJoinColumn¿
```

References

[1] BLANAS, S., AND PATEL, J. M. Memory footprint matters: efficient equi-join algorithms for main memory data processing. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013), pp. 1–16.

[2] DITTRICH, J., AND QUIANÉ-RUIZ, J.-A. Efficient big data processing in hadoop mapreduce. *Proceedings of the VLDB Endowment 5*, 12 (2012), 2014–2015.

[3] GROPP, W., GROPP, W. D., LUSK, E., SKJELLUM, A., AND LUSK, A. D. F. E. E. *Using MPI: portable parallel programming with the message-passing interface*, vol. 1. MIT press, 1999.

[4] KANG, S. J., LEE, S. Y., AND LEE, K. M. Performance comparison of openmp, mpi, and mapreduce in practical problems. *Advances in Multimedia 2015* (2015).

[5] MISHRA, P., AND EICH, M. H. Join processing in relational databases. *ACM Computing Surveys (CSUR) 24*, 1 (1992), 63–113.

[6] PARTOW, A. General purpose hash function algorithms, 2013.

[7] QADAH, G. Z. The equi-join operation on a multiprocessor database machine: Algorithms and the evaluation of their performance. In *Database Machines*. Springer, 1985, pp. 35–67.

[8] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment 2*, 2 (2009), 1626–1629.

[9] ZUREK, T. Optimisation of partitioned temporal joins. In *British National Conference on Databases* (1997), Springer, pp. 101–115.