CSC367 Assignment 3 Report

Farzaneh Tabandeh Sitao Wang

March 2021

1 Sequential Database Join

1.1 Nested Loop

The simplest method to implement a database join operation is to loop through both tables and find all pairs of entries that meet the requirement.

This algorithm is extremely slow. For two tables each with size M and N, the algorithm need to loop through $M \times N$ pairs of entries, and thus the the time complexity of $\mathcal{O}(M \times N)$. For two tables each with size more than 5000000 entries, this algorithm could not complement in an hour. Thus, this algorithm, though simple, is not practical.

1.2 Sort Merge

To improve the runtime of the naive nested-loop join algorithm, sort merge could be used. Given two databases that are sorted on the shared column to be joined, we could iterate through both tables once.

For two tables each with size M and N, for each loop iteration one of the index into table advances, so the time complexity is $\mathcal{O}(M + N)$, which is much faster than the naive nested loop. However, the algorithm required the tables to be sorted. If the original tables are not sorted, then extra computations are needed to sort both tables. The time complexity to sort both tables is $\mathcal{O}(M \log(M) + N \log(N))$. Practically speaking, the total runtime with sorting is usually smaller than the naive methods.

In our experiments, all our tables are already sorted, so we expect the runtime to be $\mathcal{O}(M + N)$.

1.3 Hash Join

In this method, one of the tables is hashed and the number of matches are obtained by looping through the elements of the other table and looking it up in the hashed table. For sequential join, we used Students table to be hashed as we could make use of the uniqueness of sid as key to have shorter searches in each bucket. It takes $\mathcal{O}(M)$ to build the hash table from one of the tables and another $\mathcal{O}(N)$ to loop over the elements of the other table and look them

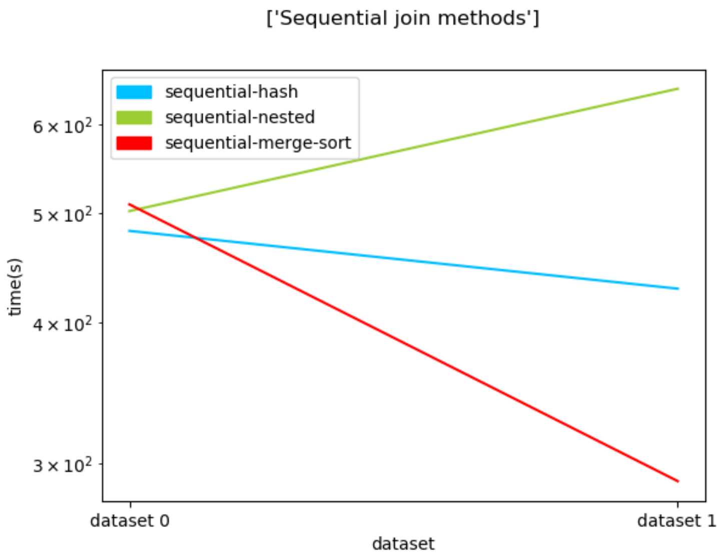


Figure 1: Comparison of Sequential Join Methods

up in the hash table. So overall the complexity of this method is $\mathcal{O}(M + N)$. Compared to the other methods above, this method have the smallest time complexity trading off extra space used for the hash table.

1.4 Comparison of Sequential Join Methods

As in Figure 1, we compared the performance of the 3 sequential join methods on data sets 0 and 1. As expected the run time of sequential nested join is the highest among all and is constantly high for two data sets. The sequential merge sort starts with the run time of around the same as the nested but significantly improves as the size of the data set increases. The sequential hash shows the best performance compared to all and its performance decreases as the size of the data set increases. The reason is that the difference of size of the Students table with the TA tables in data set 1 is much higher than the data set 0 and so the overhead of building the hash table dominates. Overall, hash join method can perform better than merge sort if the size of the hash table is small enough, otherwise the time complexity of both hash and merge will be quite similar. Again the merge sort's advantage in the case of the given data sets is that the data is already sorted, otherwise the hash map would potentially have better runt time.

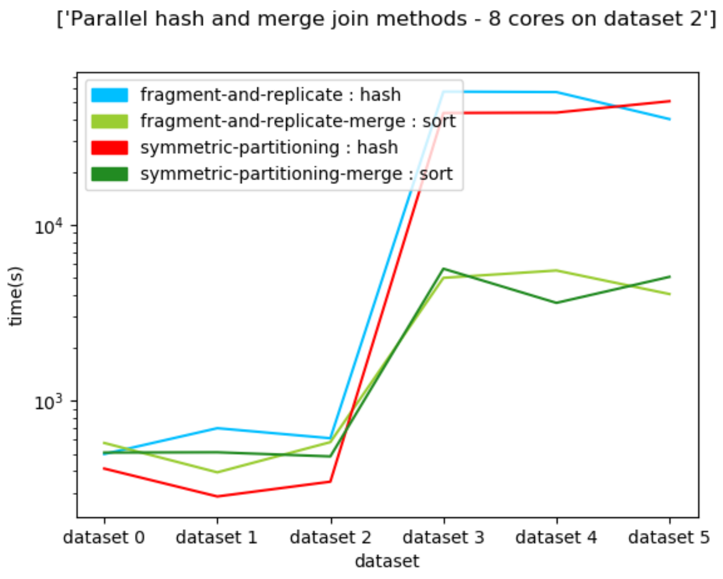


Figure 2: Fragment-Replicate and Symmetric on 8 cores

2 Parallel Database Join

2.1 Fragment and Replicate

In Fragment and Replicate parallelization method, we partitioned the larger table (among Students and TAs) and considered the smaller one shared between all threads. In particular, the partitioning is coarse grained in a way that each chunk of the larger table is compared to all elements of the smaller table. The partitioning can be done coarse grained or fine grained, meaning we never partition the smaller shared table. Figure 2 shows the results of Fragment and Replicate when coarse grained partitioning is done. We can see under this partitioning, the merge sort has better performance compared to hash and the main reason is that the data sets are sorted which benefits merge sort.

Overall, this fragmentation method's run time increases if the two data tables grows consistently together. There are two points that determine the run time in this partitioning method. One is that if the size of the shared table is much smaller than the bigger table that we partition, the performance increases greatly. In the case of Fragment and Replicate using merge sort this has happened for data set 1 compared to data set 0. This is because partitioning the bigger array provides similar sized chunks as the size of the smaller shared table, leading to a balanced load to all threads. The other point is the nature of the data sets and which table we choose to partition. For example, in the case of the hash join we can see the run time decreases from data set 1 to data set 2, although the overall size of the two tables has grown. The reason is that according to this partitioning method and the hash join implementation, we choose the smaller table to hash and consider it as shared and we partition the other larger table. In the case of data set 1, the hashed table is the TAs table while in the case of the data set 2 the hashed table is the Students table. Since hashing is based on `sid` which is not unique in TAs table, the hashing causes lots of collisions in TA table's hash. This causes an overhead in the case of data set 1, while data set 2 has the benefit of less collisions and so less time in search.

As an extra experiment, we also tried using a finer grained version of Fragment and replicate method on merge sort join,shown in Figure 3. Using a finer grain approach when the size of the shared table is too big to fit in the memory is recommended as it will partition the two tables into m and n chunks such that n * m == number of threads. Then it does the search by doing n*m times of comparisons such that each chunk of table one can be compared to all n chunks of table two. This way each such comparison job is assigned to one thread. We tried a fine grained partitioning in which we partitioned Students and TAs tables into m and n chunks, then we did n*m times of comparisons. As shown in the Figure 3, the fine grain Fragment and Replicate on merge sort improves the performance on data sets 2 and 5 compared to the coarse grain version. As described above, this is because in these data sets most of the matches are located in one specific portion of the Student table, and because in these two data sets the Student table will not be partitioned in coarse grain version, there is a load imbalance between threads. However, in the fine grain version, we divide both tables into m and n and then every one of the m*n threads will experience that portion of the Students table where most of matches exists. This way the imbalance of the chunks are corrected, hence the better performance.

We can see either versions of the Fragment and Replicate depending on the case of the data and the architecture on which the program is running can outperform the other.

2.1.1 Scalability of Fragment and Replicate

Overall, the performance of join operations in Fragment and Replicate partitioning shows a much better run-time improvements for the merge sort join compared to hash table join specially for larger data sets. In hash join (Figure 5), increasing the number of cores, does not necessarily increases the performance. In fact, if the performance gain of dividing up work iterations by having more partitions(threads) cannot cover the overhead of the thread management, there will not be a noticeable speed up. This can be seen in data set 1, that increasing threads to 4 has the best performance as in this case the search in the hash table potentially requires looping over the linked list in each bucket and so increasing number of threads has helped. On the other hand, for data sets 0 and 2, the best number of threads is 2. The reason for this is that in these data sets the overhead of search in the hash table is much less than the overhead of thread management. This point has caused the best number of threads for these two data sets to be 2 while having more threads does not increase performance. By the same reasoning data sets 3-5 also show their best performance at 2 cores,

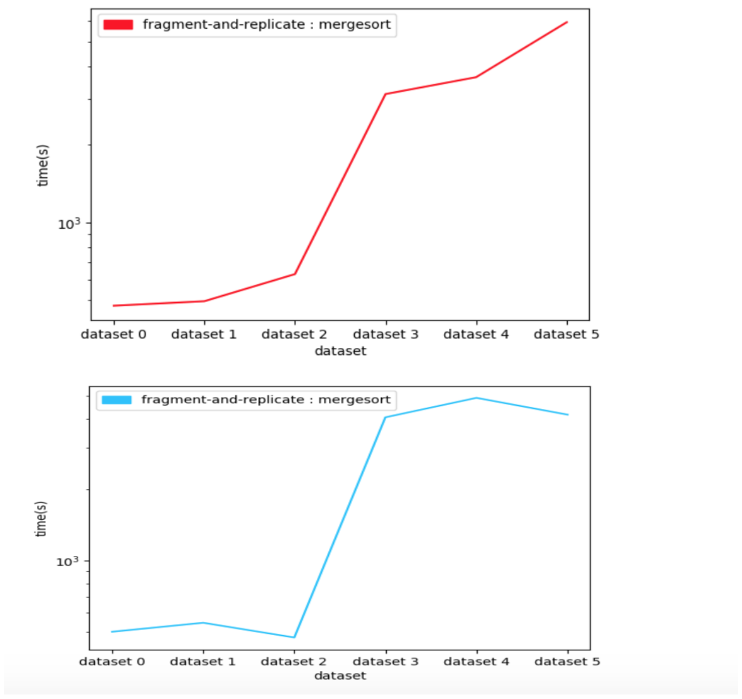


Figure 3: Coarse Grain vs Fine Grain Merge Sort (top and bottom respectively) Fragment-Replicate over 8 cores

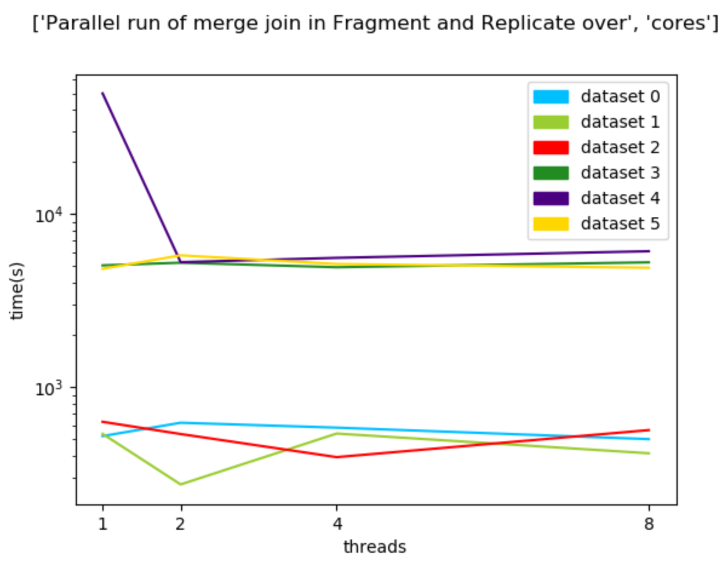


Figure 4: Fragment and Replicate Merge Sort

while increasing threads will not help so much. In Fragment and Replicate with hash join, clearly one other bottleneck is the sequential hash creation part that cannot be improved even if the number of cores increases.

In comparison, the effect of increasing cores on the Fragment and Replicate merge sort run time depends highly on whether or not the partitions of the bigger table creates quite equal sizes for the two tables that need to be merged(Figure 4). In particular, the number of cores/threads is desired that creates the similar sized partitions of the big table compared to the shared small table. That means because of the nature of this fragmentation, if the two tables assigned to each thread have similar sizes, the performance gain is much higher. For instance, data set 1 and 2 show improvement of performance over 2 and 4 number of cores respectively. The reason is that these number of cores can divide up the larger array into fragments that have similar sizes to the smaller table, leading to a balanced load among all threads. Increasing number of threads in bigger data sets 3-5 has a constant improvement compared to 1 cores but remains constant when core numbers increase. The other observation is that the increase of cores has an overall positive effect in performance on data sets that the size of the Students table is bigger than the size of the TAs table. The reason for that is that, bigger Students table compared to the TAs table means that most of the TAs will be in one specific portion of the Student table, while other portions will not contain any TAs and so will have slightly faster run over their jobs. That means if we chunk up the Student table more, we can divide up that busy portion(where there are more TA matches) and so balance the heavier load among more number of threads, hence improving performance.

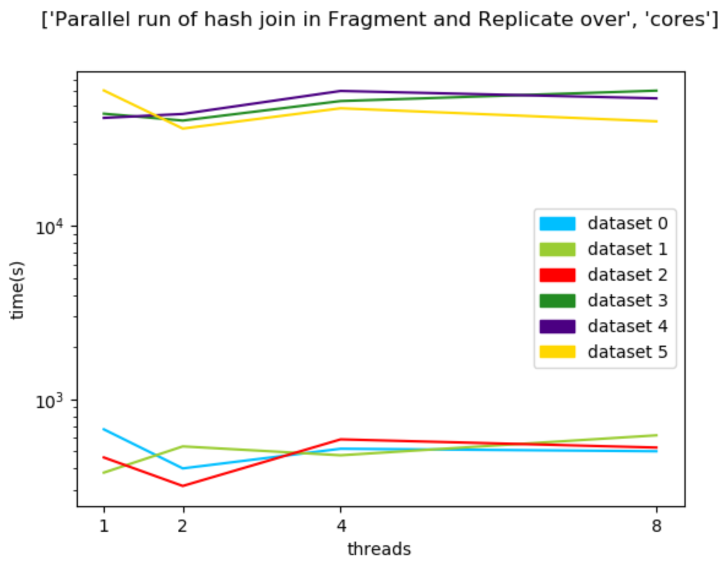


Figure 5: Fragment and Replicate Hash Table

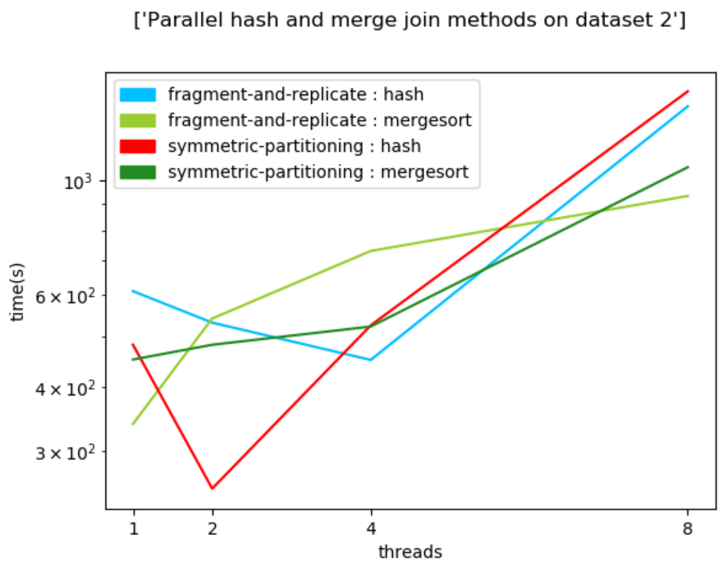


Figure 6: Fragment-Replicate and Symmetric in data set 2

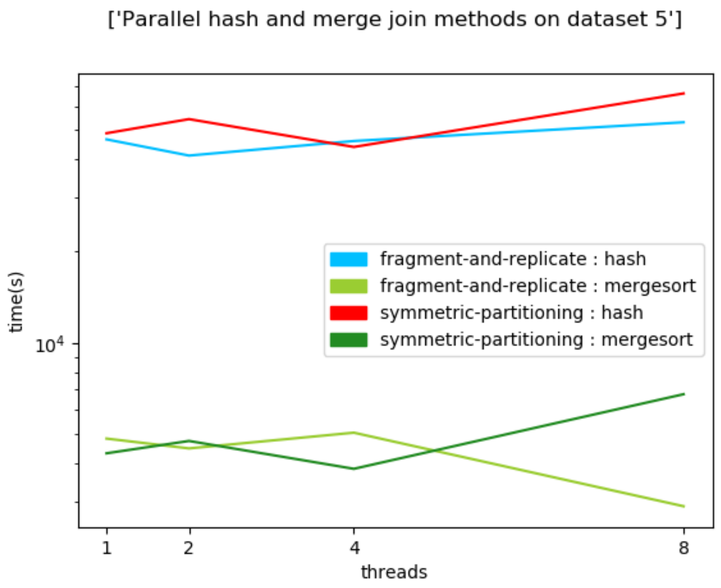


Figure 7: Fragment-Replicate and Symmetric in data set 5

2.2 Symmetric Partitioning

Another way to parallelize the database join is to partition both tables. For two tables, we segment both tables in n partitions in the way that we could get the result by just joining the pair of partitions with the same indices, thus the join of each pair of partitions could be parallelized and we only need to gather the results from the joins.

In our experiments, the two tables are sorted by `sid` and `sid` is the primary key of the student table, thus unique. So we segment the student table into n segments with equal sizes, and go over the ta table and get the matching partitions such table for each partition in ta table the max `sid` is not more than the max `sid` from corresponding partition from student table. This guarantees that there could not be matching entries across different pairs of partitions.

2.2.1 Partitioning the Hash table

For symmetric partitioning of hash join, we also tried to create two hash tables with the same size and hash function for both tables and partition the hash tables. Symmetric partitioning of such hash tables is simple. As long as two tables are partitioned in the same way, then the partition is symmetric because entries with different hashed indices could not meet the requirement. We could implement hash join directly on the partitions of hash tables.

We have conducted an experiment on the partitioning on hash table. In our experiment, we use 8 threads to utilize all 8 cores. The result of our experiment

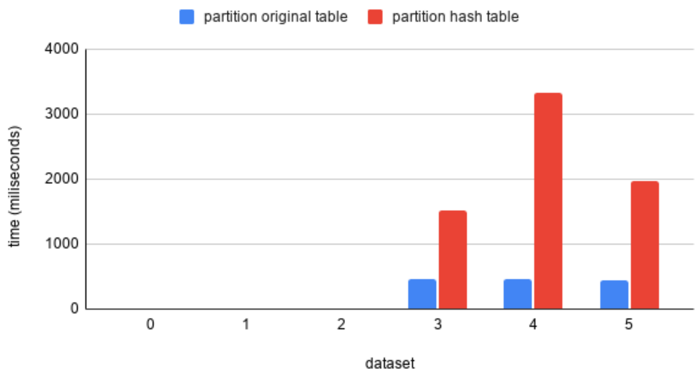
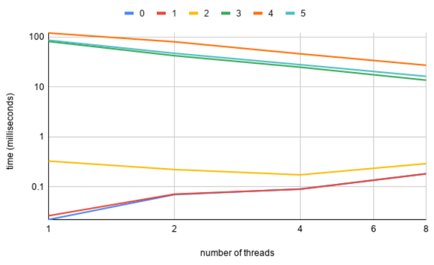
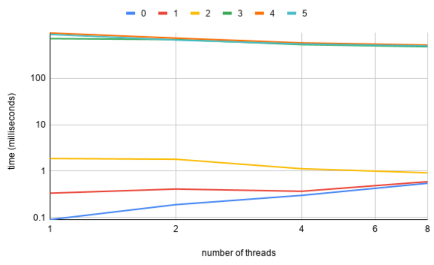


Figure 8: Symmetric Partition on Hash Table



(a) sort merge



(b) hash join

Figure 9: Scalability of Symmetric Partitioning

is shown in figure 8. In our experiments, this method is slower than the partitioning of the original tables for table with large sizes, and the main reason is that we have to create the large hash table sequentially, but for our original method, each thread only needs to create a small hash table and the hash table constructions are parallelized.

2.2.2 Scalability of Symmetric Partitioning

We have also conducted experiments on the scalability of the symmetric partitioning with sort-merge join and hash join. We run the symmetric partitioning of each join method on 1, 2, 4 and 8 threads for 5 times and take the average. For each experiment, number of partitions is equal to the number of threads.

The result of our experiments are presented in figure 9. For both graphs, both the horizontal axis and vertical axis are plotted in log scale to facilitate analysis on scalability.

From the result of sort merge in figure 9a, we could see that the runtime scales well with increased number of threads for datasets 3, 4 and 5. However, for smaller datasets, the runtime even increases with the increased number of threads.

The reason for the abnormal in small dataset is the overheads. To create the symmetric partitioning, we need to walk through the ta table once and computes the partition boundaries, and the number of partition is proportional to the number of threads. This computation has to be done in sequential before we spawn the threads to compute each partition. Creation and management of threads also create some overheads. These overheads consists of larger proportion when the runtime of each thread is small, thus for smaller datasets, these overhead dominates and runtime might increases with the increases number of threads. For larger datasets, theses overhead consists only a small part of the total runtime and thus due to Amdahl's Law, the speedup of each total runtime is close to the speed up of each thread, and thus the total runtime scales linearly with respect to the number of threads.

The result of hash join in figure 9b shows that symmetric partition does scale well for hash join. For small datasets, the runtime increases as with sort merge, and for large datasets, the runtime does decrease with the increased number of threads, but not with same proportion. The double of thread number only results in a slight decrease of the runtime.

2.3 Comparison of Parallel Join Methods

The major bottleneck of symmetric partition algorithm lies in the creation of hash table. In our profile of the experiments with 8 threads, 80.47% of the total runtime is spent in the `malloc` inside the hash table creation. The major reason is the `malloc` is thread-safe, and thus probably synchronized. When multiple threads call `malloc` at the start of the threads to insert records, theses calls create contention and limits the concurrency.

One way to solve this problem is to reduce the number of calls to `malloc`. Because the number of elements to insert is known, each thread needs only to call `malloc` for linked list nodes once to allocate an array for all nodes, and use the i^{th} element in array to insert the i^{th} record in the partition. Even if in the case where the number of elements are not known, we could estimate the number and allocate a buffer to reduce the call to `malloc`. However, we have not implemented this optimization in this assignment.

The Fragment and replicate is a good approach when the size of one of the tables are much smaller than the other since in this case partitioning the larger table makes well balanced jobs of comparing tow similar size arrays. Then for choosing the best join method, given the tables are already sorted, we can see the merge sort outperforms hash table method (Figure 2). The reason is that despite hash table join, it does not have the overhead of creating hash table which is both the sequential part of the code and requires calling `malloc` a lot. However, if the arrays are not sorted, the time complexity of merge sort will predictably be as high as the hash table.

As shown in Figure 10, comparing the speedup of the parallel join methods shows that for data set4, the merge sort gives the best linear speed up on both

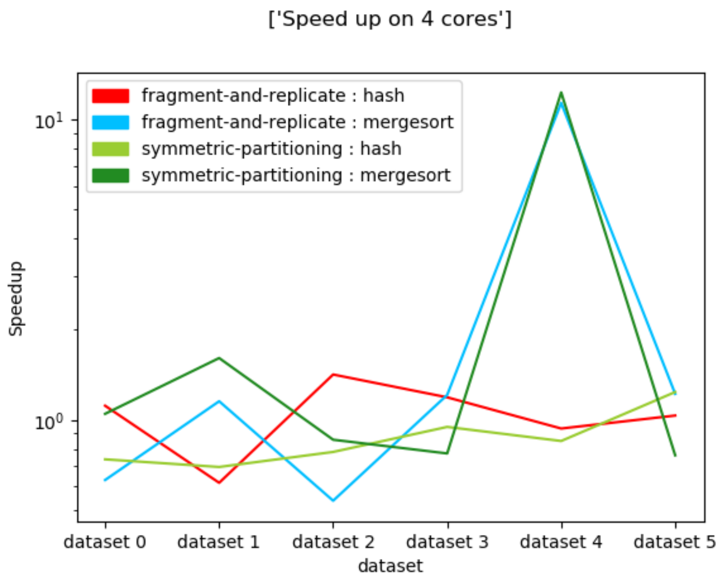


Figure 10: Parallel Methods Speedup on 4 cores

partitioning methods. Since in this data set, the size difference of Student and TA table is big, both partitioning methods help to assign equal load to threads and hence better speedup. Also in data set 2, best speedup is gained by fragment and replicate hash method. As explained above, data set 2 hashes Students table and so benefits from less collisions in search. So as shown using 2 threads to search the hash table happen to spread the load in a more balanced way compared to other methods.

We can conclude that the choice of the best partitioning and join methods for parallelization clearly depends on the type and nature of the data set and whether is the load among threads is balanced . We can see in Figure 6 that for data set 2 where the size of the Student table is higher than the TA table and as said above probably has this nature that one portion of the student table contains most of the matches, symmetric partitioning hash join method over 2 cores has the best performance. The reason is that only two comparing two threads to compare two chunks of the TA and Student array will creates a balanced load while increasing number of threads will create load imbalance. As another example for data set 5 using fragment and replicate hash join over 8 cores is the best as shown in Figure 7. Since in this data set the number of Student and TA tables are quite similarly large, diving both into smaller chunks and scheduling them over more cores has helped in improving performance a lot.