CSC367 Winter 2021
Assignment 3 Report

Shuting Zhang zhan4145
Muyuan Cao caomuyua

1. Implementation

1.1 Nested-loop join

The nested-loop method is implemented in the function join_nested() in join.c. It is very straightforward: for each entry of the Student table, check first if this student has a above 3 GPA; if this student does, then loop through the TA table to see if this student is also a TA. This is a brute force algorithm with M*N operations where M is the number of students and N is the number of TAs.

1.2 Sort-merge join

The sort-merge method is implemented in the function join_merge() in join.c. For this method to work, both the Student relation and the TA relation have to be sorted by sid in advance. We iterate through the Student relation and TA relation to look for matching pairs that satisfy the requirement---GPA > 3.0. After finding a matching pair, filter and count all of the following entries with the same sid because the TA table allows duplicate sids. Since we iterated through each relation once, this method has M + N operations in total.

1.3 Hash join

The hash join method is implemented in the function join_hash() in join.c. In order to minimize the operations needed for finding a match, we only hashed the sids for students with above 3.0 GPA, which is also unique and therefore could possibly reduce collision. Then looked for matches by iterating through entries of the TA relation and applying the same hash function on the attribute sid.

The hash table implementation is in hash.c, which has the same implementation as Lab3. We used the modulus function as our hash function. Since we tried to reduce collision by only

hashing the sids for students with above 3.0 GPA, and we iterated through each relation only once to look for matches, therefore ideally the hash join method costs N + M operations.

1.4 Fragment-and-replicate

The Fragment-and-replicate method is implemented in the fragment_replicate() function in join-omp.c. In this method, we partitioned the larger relation over all the threads created by OpenMP, and replicated the other relation entirely. Each thread works on its partial count. When the last thread finishes its work, the total matching count is computed by reduction.

1.5 Symmetric partitioning

The symmetric partitioning method is implemented in the symmetric_partitioning() function in join-omp.c. In this method, we partitioned the Students relation sids over all the threads created by OpenMP. Since both the student and the TA relation is sorted by sid, we had each thread to iterate through both the relations to find upper and lower indices corresponding to the upper and lower sids this thread got assigned to. Then the thread would proceed to find a partial matching count by joining the sub-relations. Again, because the sids are sorted, joining the sub relations independently would still give us the correct result. When the last thread finishes its work, the total matching count is computed by reduction.

2. Experiments

2.1 Experiment 1: Comparing the Different Sequential Join Methods

2.1.1 General Idea:

In this experiment, we ran different sequential methods using datasets ranging from dataset0 to dataset2 and recorded the time.

2.1.2 Experiment Setup:

We ran the command ./join-seq {-n | -m | -h} ../data/dataset{0|1|2}, and used a python program to record the run time and plot the diagram accordingly.
All the measurements are averaged over 10 runs for better accuracy.

2.1.3 Performance Comparison:

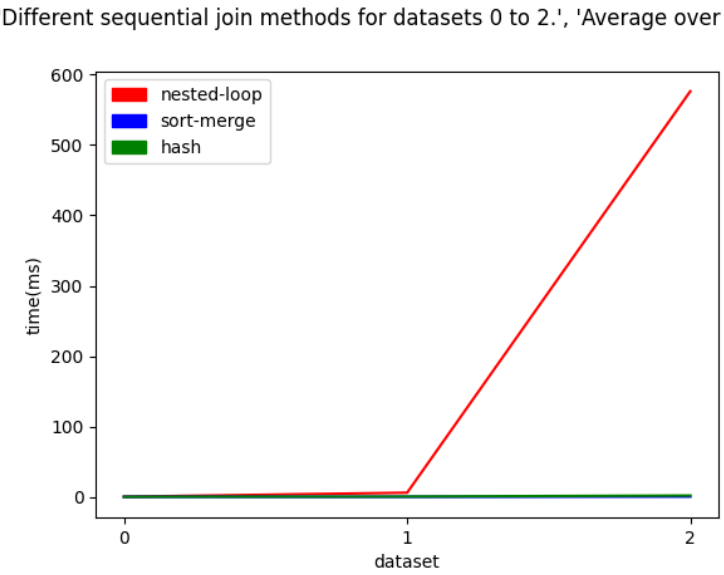

Figure 1: Performance for Different Sequential Methods

Figure 1 above is the result for each method's run time. As the figure shows, different colors represent different sequential methods. The amount of time used is shown on the y-axis, and different dataset numbers are on the x-axis. We see that, because the lines are overlapping, the three methods have similar performance when the dataset is relatively small (dataset0 and dataset1). However, as the sort-merge and hash methods continue to have good performance, the nested loop algorithm run time increases dramatically when moving on to dataset2. This is because dataset2 is way larger than dataset0 and dataset1. While the sort-merge and hash methods have approximately M + N operations, the nested-loop method has M*N operations, which is the cause for it being very inefficient when processing large datasets.

Although both the sort-merge and hash methods continue to have good performance, the green line, which stands for the hash join method, is a little above the blue line, which represents the merge sort method, on dataset2. While they both have approximately M + N operations, the hash method has additional overheads from initialing and cleaning up the hash table, and putting elements in and searching for elements in the hash table.

2.2 Experiment 2: Comparing pairs of Parallel and Partial Join Methods

2.2.1 General Idea:

In this experiment, we did experiments using datasets from 0 to 5. Since the datasets get large, we omitted the nested-loop method and only used the sort-merge and hash for partial join method.

Since we have 2 independent variables in this experiment---the parallel join method and the partial join method, we needed to fix one of the variables when conducting this experiment. In this experiment, we first fixed the parallel join methods and compared the performance of different sequential join methods. Then move on to other parallel join methods and do the same experiment. In the end, we are going to compare these different pairs of parallel and partial joins.

2.2.2 Experiment Setup:

We ran the command ./join-omp {-m | -h} {-r | -s} -t 8 ../data/dataset{0|1|2|3|4|5}, and used a python program to record the run time and plot the diagram accordingly.
All the measurements are averaged over 10 runs for better accuracy.

2.2.3 Performance Comparison

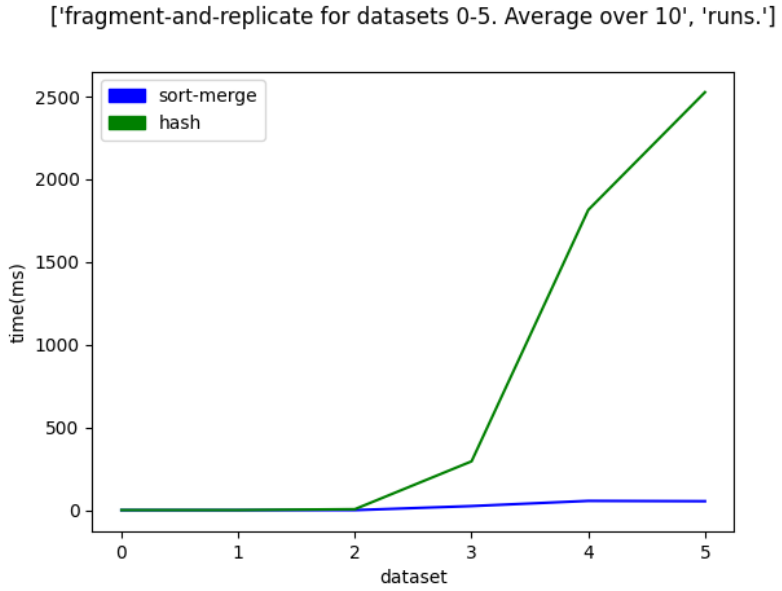

Figure 2: Comparing the hash and the sort-merge methods, using fragment-and-replicate as the parallel method.

According to figure3, we can conclude that when using fragment-and-replicate as the parallel method, the merge-sort method outperforms the hash method when the dataset is large. Although both the sort-merge and the hash methods have approximately M + N operations, the hash method has additional overheads from initialing and cleaning up the hash table, and putting elements in and searching for elements in the hash table. When the dataset is very large, although searching for a particular element is still efficient, the overhead of initialing and cleaning up the hash table becomes very large.
Another reason might be that when the dataset is very large, the hash table becomes very large and might not fit into the cache. As a result, the speed of accessing elements in the hash table

greatly slows down, due to the extra overhead introduced by excessive communication between slow memory and fast memory.

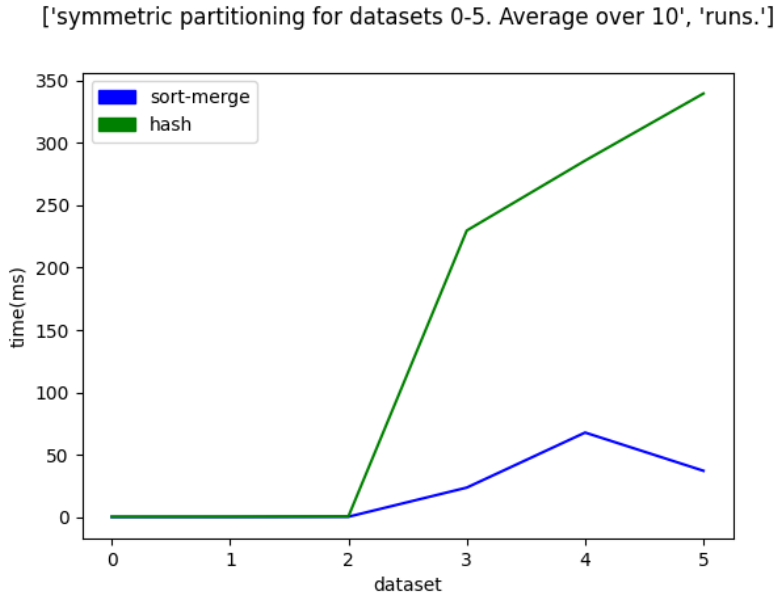

Figure 3: Comparing the hash and the sort-merge methods, using symmetric partitioning as the parallel method.

After switching the parallel method to be symmetric partitioning, a similar result can be observed from Figure3---the merge-sort method outperforms the hash method when the dataset is large. The reasoning is similar: when the dataset is very large the overhead of initialing and cleaning up the hash table becomes very large, while sort-merge does not have that problem.

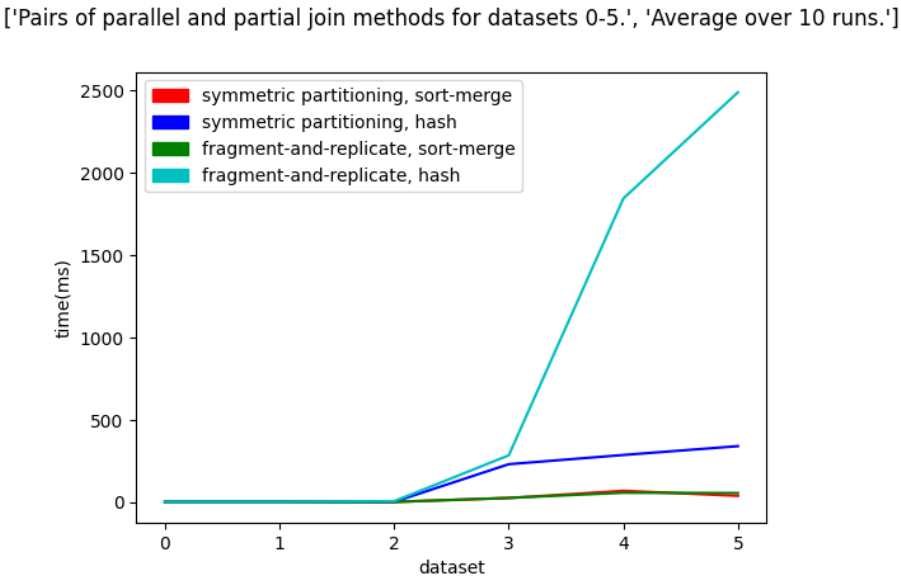

Figure 4: Comparing the hash and the sort-merge methods, using symmetric partitioning and fragment-and-replicate as the parallel method.

Now we compare the 4 combinations of parallel and partial join methods. According to Figure 4, the two merge-sort methods are the fastest, the hash + symmetric partitioning combination is slower, and the hash + fragment and replicate method is the slowest.

The merge-sort methods are faster and the hash methods are slower are as expected. We believe fragment-and-replicate + hash does not perform as well as symmetric partitioning + hash is because each thread in fragment-and-replicate + hash needs to do more work---it needs to join a small chunk of one relation with the other entirely replicated relation. While symmetric partitioning + hash does not have that problem. Each thread only needs to process two small chunks from the two tables.

2.3 Experiment 3: Scalability of Different Join Methods

2.3.1 General Idea:

In this experiment, we are going to analyze the scalabilities for different pairs of join methods. We will plot and analyze how the performance of the parallel methods scale for 1, 2, 4, and 8 threads.

2.3.2 Experiment Setup:

We ran the command ./join-omp {-m | -h} {-r | -s} -t {1|2|4|8} ../data/dataset{0|1|2|3|4|5}, and used a python program to record the run time and plot the diagram accordingly.
All the measurements are averaged over 10 runs for better accuracy.

2.3.3 Scalability

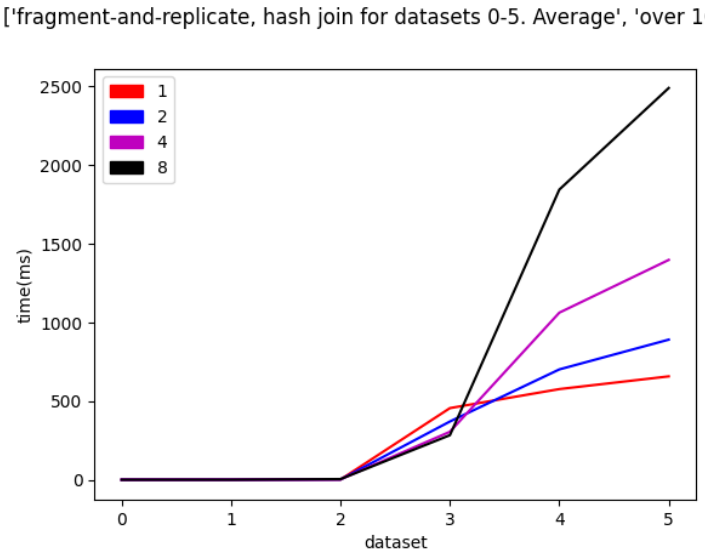

Figure 5: Scalability of fragment-and-replicate + hash

Although dataset5 is not necessarily larger than dataset4, because we only put a sid in the hash table if the student has an above 3.0 GPA, we think that the method takes longer on dataset5 is due to dataset5 having more students with above 3.0 GPA.

According to figure 5, to our surprise, with small datasets, more threads lead to better performance; with larger datasets, on the other hand, more threads lead to worse performance. We think this is due to our inefficient hash table implementation. For the fragment-and-replicate method, each thread needs to initialize a hash table and iterate through it to find matches. The more threads there are, the more overhead of initializing and iterating through the hash tables would be introduced, which then greatly slows down the program.

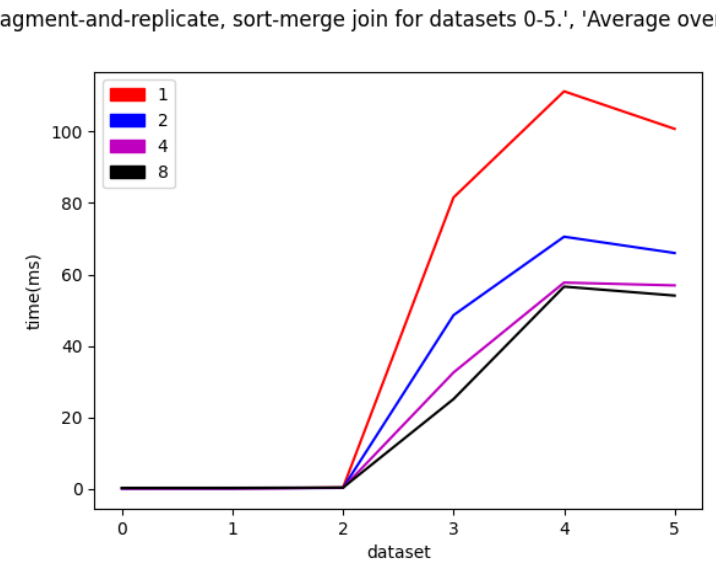

Figure 6: Scalability of fragment-and-replicate + sort-merge

According to figure 6, the fragment-and-replicate + sort-merge method consumes less time when there are more threads available, which is as expected. However, we would not say the fragment-and-replicate method has perfect scalability because, especially for dataset4, having 8 threads is only a little bit faster than having 4 threads.

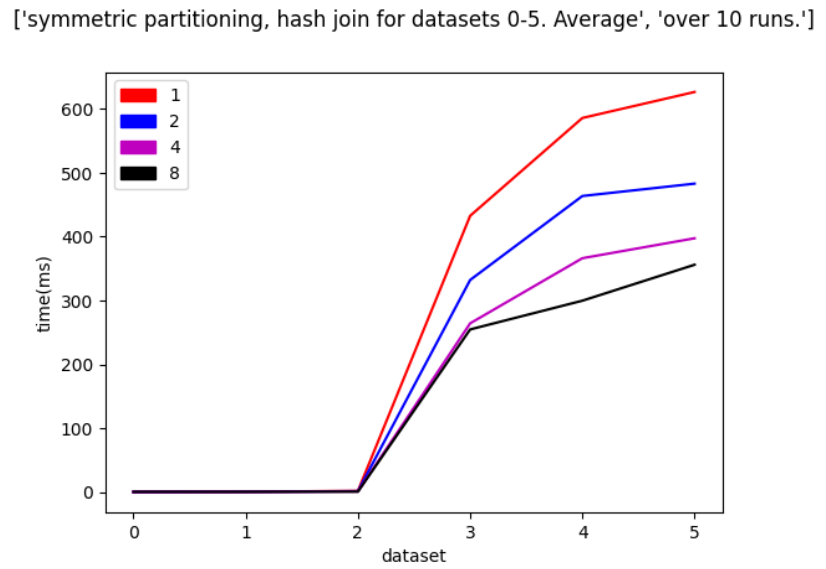

Figure 7: Scalability of symmetric partitioning + hash

According to figure 7, the symmetric partitioning + hash method consumes less time when there are more threads available, which is as expected. However, this method still does not scale perfectly since for dataset3, having 8 threads does not improve the performance much from having 4 threads. We think this is still caused by our inefficient implementation of the hash table, which introduced extra overhead.

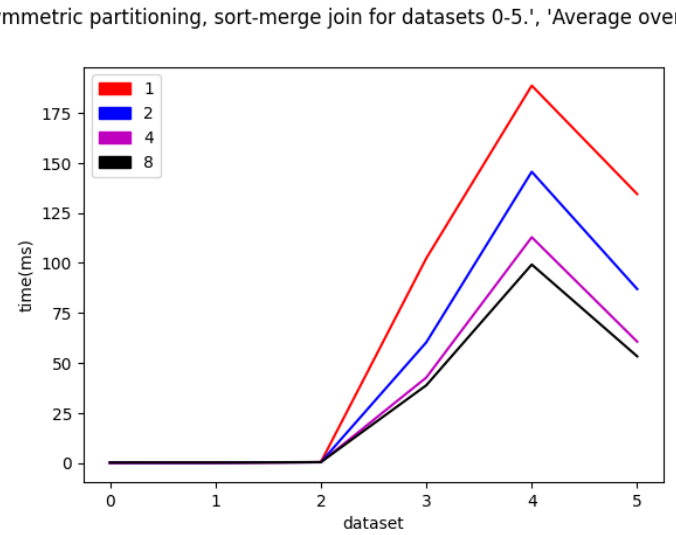

Figure 8: Scalability of symmetric partitioning + sort-merge

According to figure 8, although the scalability for the symmetric partitioning + sort-merge is still not perfect, it has better scalability compared to the other three methods. We think symmetric partitioning + sort-merge generally has better scalability because when more threads are created, each thread gets less work assigned compared to the fragment-and-replicate method, in which a single thread has to take on an entire table despite the number of threads available. Moreover, it does not use a hash table and therefore could avoid the overhead introduced by the hash table.

The scalability for the symmetric partitioning + sort-merge is, although better, still not perfect. We think this is due to the overhead introduced by having each thread iterating through the two tables to find upper and lower indices corresponding to the upper and lower sids this thread is assigned to.