Analysis of Query Techniques

Symmetric Partitioning

In this method of parallel join, we partition both relations into independent fragments, such that a fragment from one relation must only be joined with a singular fragment from the other relation, and the result totalled at the end of computation.

In our implementation, we assign shards of the `students` relation to each thread that is created. A thread maintains a start_index and end_index which are indices that represent the start and end position in the students relation array for which the thread is responsible for.

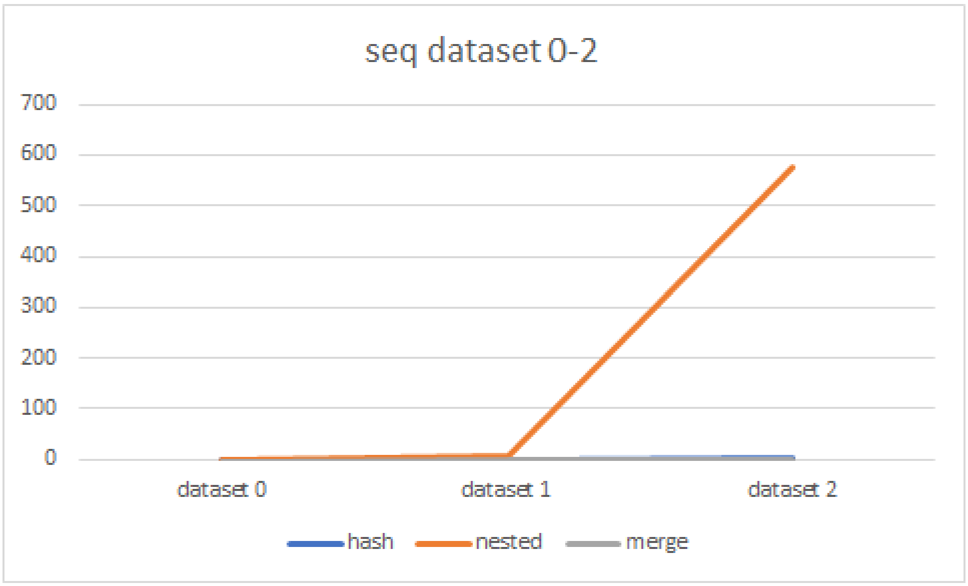
To determine which corresponding shard of `teacher_assistant` (`tas` variables in the code) the thread is responsible for, each thread calls the helper `data_partition_helper`. This is a simple, linear pass, through the entirety of the teacher_assistant relation. We must ensure that the shards of both relations contain only the same range of student id numbers (referred to as `sid` from here on out).

To ensure this constraint, in the `data_partition_helper` we take advantage of the fact that both relations are sorted on the `sid` field. As we pass through the `teacher_assistant` relation, we check if the sid at the current position is within the bounds of the `sid` at the beginning and end of the `students` array shard. If it is, we increment the index for the `teacher_assistant` shard. This continues until we reach a `sid` in the `teacher_assistant` relation which is bigger than that of the `sid` at the end of the student shard. The linear pass might not be the most efficient method, but as we noticed in some of the data sets, some shards of students have no corresponding `teacher_assistant` shards, which saves us computations that would have otherwise been wasteful in fragment & replicate.

After the partitions are calculated, the respective join function is applied on both. These are applied as described in the sequential section, apart from the hash join implementation, which differs slightly. For symmetric partitioning, we have the master thread create a large enough hash table to maintain each student id in their own bucket, where we use the student id itself as the key (no hash function applied). Since each student id is unique, there are no collisions, and since every sid in the given datasets are in the range of 0,..,students_count, we have a very convenient way of hashing that allows us to insert and get the value in O(1).

Data Analysis

Sequential Performance And Analysis



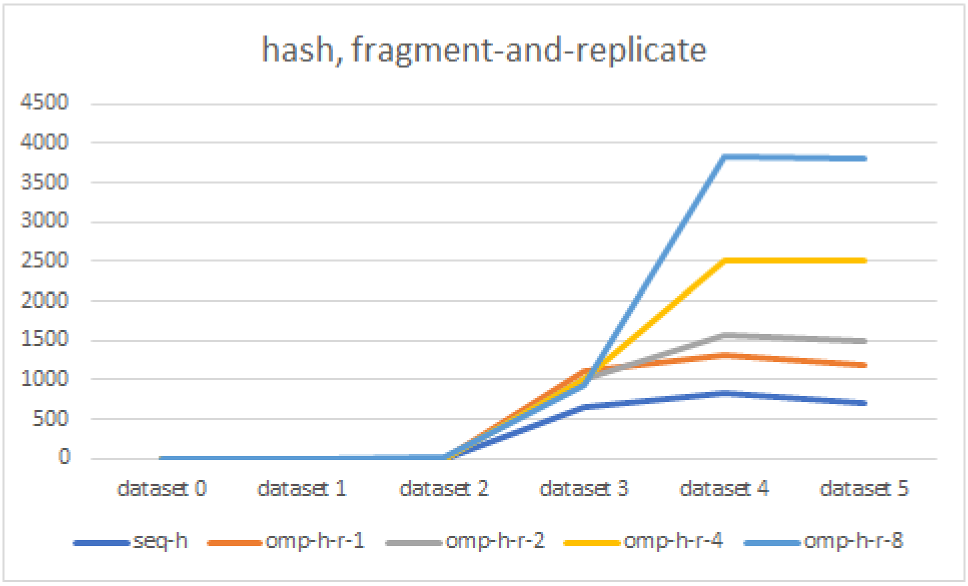
The sequential data is as expected. Sort-merge join performs the best, followed by Hash-join, with Nested-loop join being the slowest by a big margin.

This is expected as Nested-loop is a very naive approach, requiring n*m comparisons, where n is the number of entries in the student relation table, and m being the number of entries in the teacher assistant relation table. In comparison, although Hash-join requires two linear passes of both relation tables, we are able to take advantage of a hash table's O(1) average time performance. The Hash-join method is outperformed by the Sort-merge method due to the fact that before we can take advantage of the benefits of a hash table, we must allocate all the memory required, and hash each student, all of which take a relatively considerable amount of time.

In contrast with Sort-merge join, the reason it performs so well is due to the fact that in our assignment, both relations are already sorted, and thus Sort-merge is not responsible for the most computationally heavy operation - the sorting. Since the relations are already sorted, we get a time complexity of O(|R| + |S|), where |R| is the size of the first relation, and |S| is the size of the second relation.

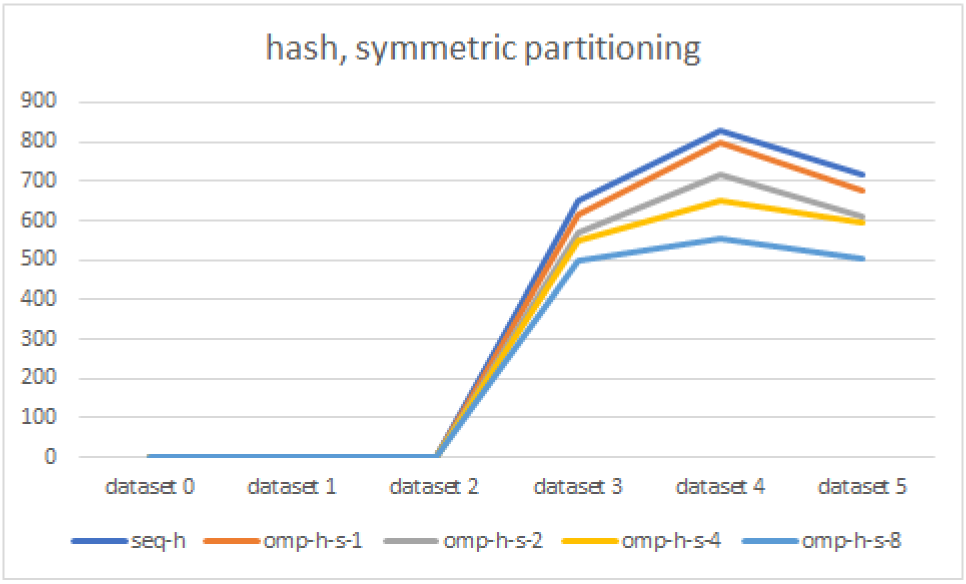
Parallel Performance And Analysis

Hash-join: Fragment-&-Replicate



Our data here is particularly interesting to us, as the sequential implementation of the Hash-join outperforms the parallelized version. Although this strikes us as odd, due to the manner of our implementation, as we increase thread count, the amount of hash tables (and thus hash operations) we must perform also increases. As described in the previous section of the report, creating the hash table and populating it is expensive, and it makes sense that the more times we have to do this, the more expensive the algorithm is. This suggests that our implementation is flawed in some way.

Hash-join: Symmetric Partitioning

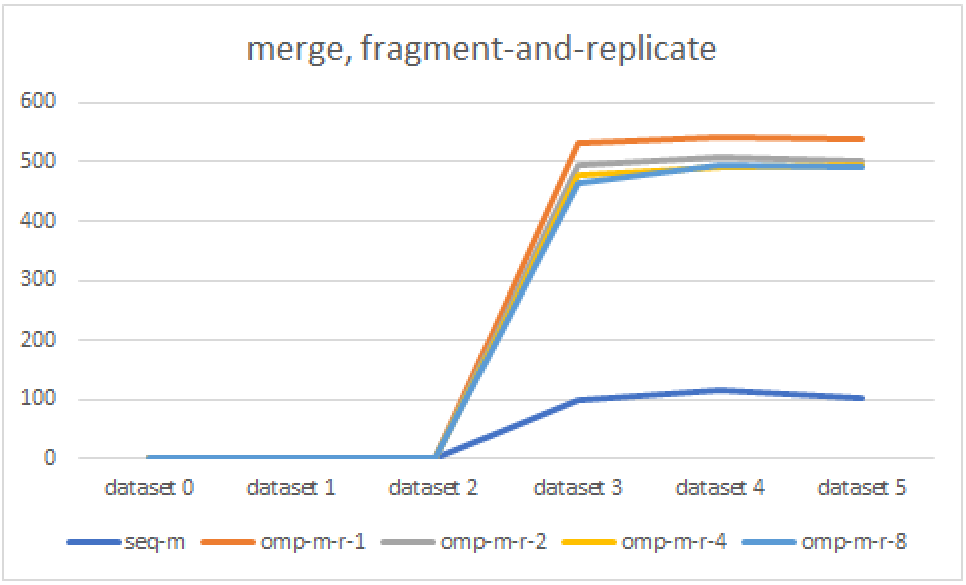


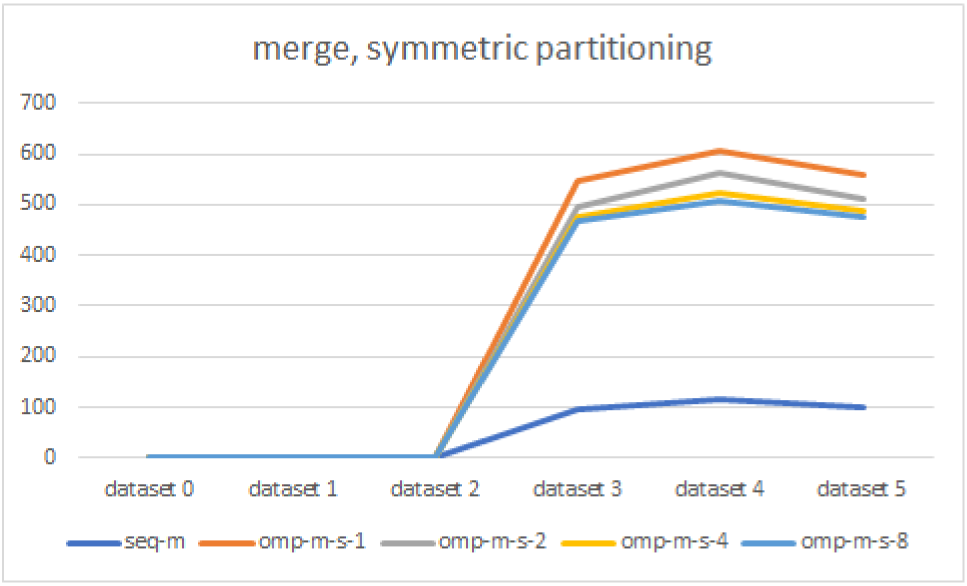
Unlike our data for the fragment-&-replicate implementation of Hash-join, this graph is consistent to what we expect to see. The sequential implementation performs the worst, and running the algorithm with 8 threads (the maximum number of cores on the machine), performs the best.

Compared to the fragment-&-replicate implementation, we only create one hash-table, which already negates the problem we talked about in the previous section. As well, as described in the implementation description section for this algorithm, we take advantage of the data set given to us to create a very efficient manner of hashing.

We can see that for dataset5, although it is a larger dataset than dataset4, the result of the join computation is in fact smaller than that of dataset4, so it is reasonable that dataset5 is processed faster than dataset4.

Sort-merge join: Fragment-&-Replicate + Symmetric Partitioning





We take note that for Sort-merge join, Fragment-&-Replicate outperforms Symmetric Partitioning. We think the reasoning behind this is because in Symmetric Partitioning we must do a linear pass of the teacher_assistants relation for every thread we create, so even though each thread does less over all computations with respect to the join itself, there is more overhead associated with assigning data to each respective thread.

As well as described with the Hash-join implementations, we have that dataset4, although smaller, is quicker than dataset5. This is for the same reason as described in the Hash-join analysis.