# CSC367 Assignment 3 report

Tianchen Xu, Xuan Du

November 2021

## Implementation Description

1. Nested-loop join, Sort-merge join, Hash join

   For Nested-loop join and Sort-merge join, we simply follow the pseudo code given on assignment handout. For Hash join, we use chaining linked list implementation for hash-map. The hash-code is straight forward 1 : 10 distribution for SIDS.

   ```
   int hash32shift(int key) {
       return key%(size/10);
   }
   ```

2. Shared memory parallel join, Open-MP

   ```
   #pragma omp parallel reduction(+:count) shared(n, student_size, ta_size)
   ```

   For fragment-and-replicate, we split the larger data-set from students and tas. Otherwise, distributing a very large relation over the interconnect between the processing elements can be a major bottleneck. So firstly spit, then each thread join a partition and reduce the local count to global count.

   ```
   count += join_f(students, students_count, &tas[ta_start], new_ta_count);
   ```

   For symmetric partitioning, we split both students and tas with equal partition, expect for the last thread. Then do similar operation to fragment-and-replicate.

```
count += join_f(&students[first_ta], last_ta - first_ta + 1,
&tas[ta_start], new_ta_count);
```

3. Distributed parallel join, MPI

   For fragment-and-replicate, we merge the smaller data-set between students and tas. For merge the data-set from each processes, we message passing to add all partitions from processes together using MPI ALLGATHER.

```
// Create the datatype
MPI_Datatype student_type = create_student_type();
// message passing to add all partitions from processes together
all_students = (student_record*)malloc(all_students_count * sizeof(student_record));
MPI_Allgather(students, students_count, student_type,
all_students, students_count, student_type,
MPI_COMM_WORLD);
local_count += join_f(all_students, all_students_count, tas, tas_count);
// add local count to global count
MPI_Allreduce(&local_count, &count, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
```

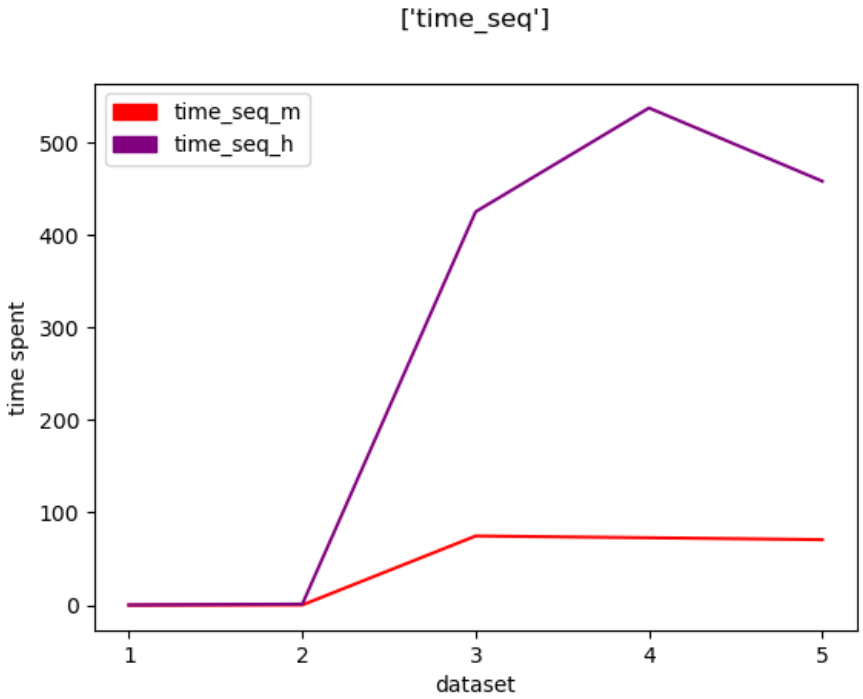
   For symmetric partitioning, since data-sets are already partitioned, we just join on each processes.

```
local_count += join_f(students, students_count, tas, tas_count);
// add local count to global count
MPI_Allreduce(&local_count, &count, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
```

# Performance

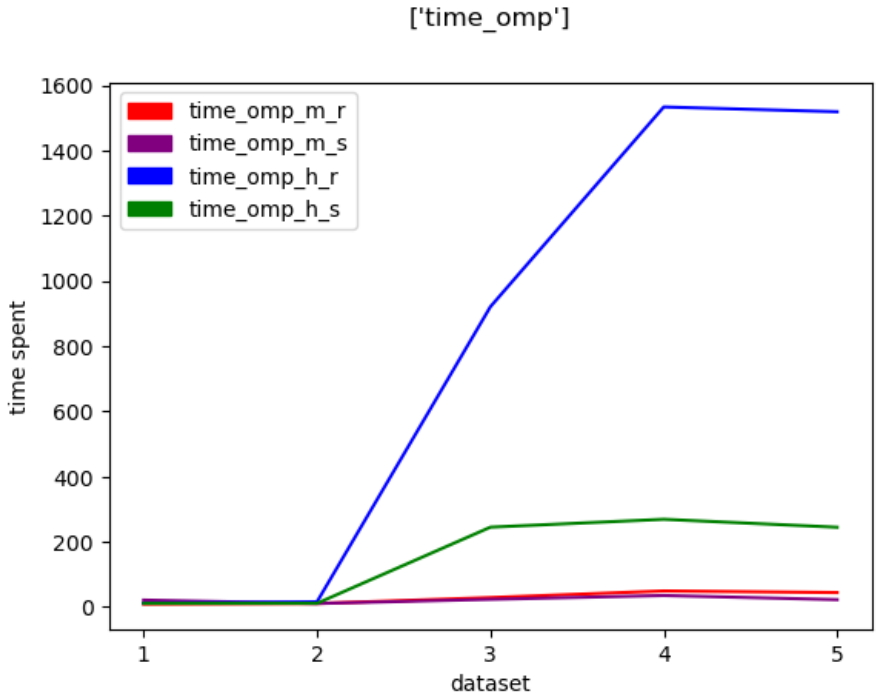
1. Seq

   Since nested loop take too long to process, we use merge join and hash join as benchmarks. Overall for join, merge join has better performance to hash join, because when data-set is large, hash join take times to build graph and finding elements in chaining linked list.



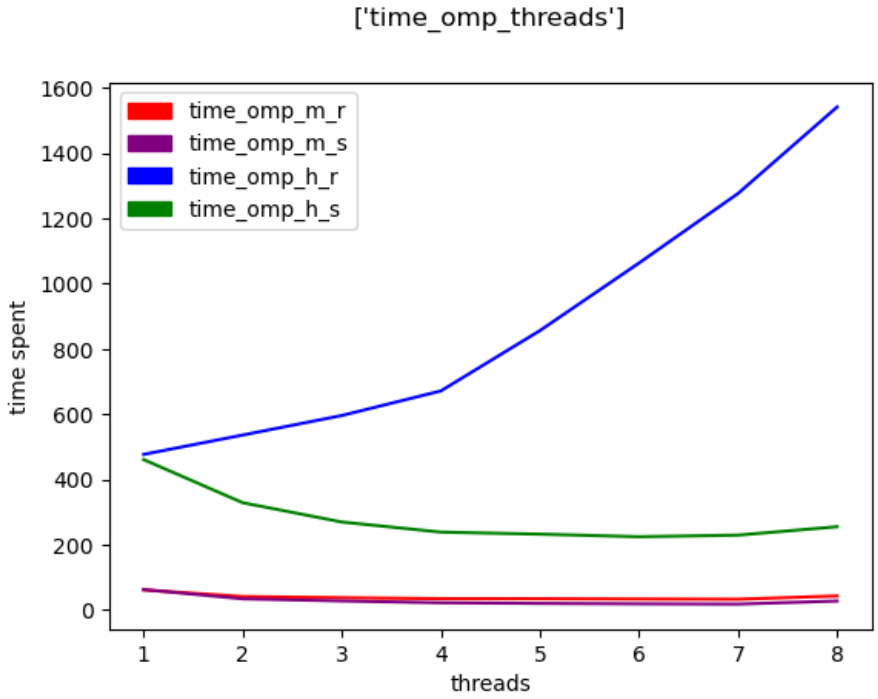

2. Omp, fixed threads=8, dynamic data-sets

   Firstly we fix the threads number to 8 and see how omp parallel method perform on different data-set2. Except hash join using Fragment-and-replicate, other methods are benefits from multi threads processing. The assumption here is the hashing fragment-and-replicate need to build the whole relation in each process which is unnecessary, wasting both time and space. The larger the data-set is, the more time and space this method wastes.




3. Omp, dynamic threads, fixed data-set=5

   Then we fix the data-set to data-set5 and see how omp parallel method perform on different threads. Except hash join using Fragment-and-replicate, other methods are benefits from multi threads processing. The reason is similar to the previous discussion.




4. Mpi4 and mpi8

   For mpi parallel method. We can clearly observe symmetric partitioning is benefiting from more threads, but fragment-and-replicate are not. More threads will only create more overhead for fragment-and-replicate since it take time to message passing to add all partitions from processes together.