Assignment Three Report

Bizhe Bai, Muzhi Lyu

November, 2020

Part1-Sequential Implementation

We implemented three different methods of sequential join, Nested loop. Merge join and Hash join. Assume the array of student has m elements, and the array of ta has n elements. We can then compare the run time of each method.

1.1 Nested-Loop Join

The implementation for this part is basic, for every student, we first check the student's GPA, if it is greater than 3, we will loop through the array of ta, and count the number of ta with same id. No matter which array we check first, we will always have to loop through the other array for several times. Then Nested loop will need atotal of $m \times n$ operations.

1.2 Sort-Merge Join

We followed the pseudo code to implement merge sort: compare the value first, then check the student's GPA before adding to the result. We don't need to take extra time sorting the array this time, so Merge sort is the most efficient among all sequential implementation, with $m + n$ operations. We are going through both array at the same time, and there's no need to change the sequence based on the length of array.

1.3 Hash Join

We used linked list to store our hash table. Every node in the linked list stores a sid, a pointer to the next node, and a boolean value to track whether the node is occupied. Our hash function is designed that the hash table will have the same size as the array being hashed. When choosing the array, we first planned to chose the array of students, disregarding the size. This is because that the array of students will have a continuous sid, and we can make sure the size of every bucket is exactly one. However, to reduce the size we malloc in the heap, we still chose to hash the smaller array.

It takes m operations to hash the first array, and, in the best case, another n operations to look for n elements inside the hash table. There could be some buckets that have more element if the array of tas is hashed, but these should be a relatively rare case, and won't have too much cost. Therefore the total operations would be approximately $m + n$.

Part2-Parallel in OpenMP

We implemented two ways to parallelize our code in order to handle larger data sets. However, even if the work is done by several threads, $m \times n$ operations from the nested-loop would still be a heavy burden. Therefore we focused more on merge join and hash join when implementing this part of our code.

2.1 Fragment-and-Replicate

For both join techniques, if we decide to partition array of length m and replicate the length n array into k threads, we will have a $m + k \times n$ operations in total. Then we need m to be the larger array that we partition. Since we need to pass the number of students and tas into our join function, I calculated the length of partitioned array beforehand, and passed it as a shared variable. The count of match is initialized to 0 and passed as replicate variable. The following are the steps we took to implement this method:

1. Compare the size and choose which part to partition

2. Calculate the amount of work for each threads, using the total number of threads given. The last thread may have less work compared with other threads.

3. Pass both arrays and their size, together with the amount of work as shared variable, replicate count variable and start OpenMP parallelization.

4. Adjust the pointer to partitioned array with amount of work and thread id in each thread, call the assigned join function and record the output.

2.2 Symmetric Partitioning

We used the same technique to partition the array of students. For the array of tas, we have to match each group of tas to the corresponding group of students such that the students' sid is included in the range of tas' sid. To approach that, we set a lower bound and upper bound for the partition, and initialized it by the thread id. For example, when partitioning a group of 2000 tas for the second thread, with a total of eight threads, we initialize the lower bound to be $2000 \times \frac{1}{8}$, and upper bound will be $2000 \times \frac{2}{8}$. Then we increase upper bound and decrease lower bound until the ta's sid is greater(less) than the sid of last(first) student in this group, or the end of array is reached. In this way, we could partition the array of tas efficiently. Even though the groups of tas will intersect, as long as the groups of students have no intersection, the result will be accurate. We used the following steps to implement:

1. Partition the array of students, calculate the amount of work for each thread.

2. Pass both arrays, the size of arrays and the amount of work as shared variable, replicate count and start parallelization.

3. Initialize the partition of the tas array, set private bounds for each thread.

4. Compare the bound with the partition of students array it has. Use while loop for increments/decrements.

5. Adjust the pointer to both array, call the assigned join function.

Part3-Parallel in MPI

3.1 Fragment-and-Replicate

3.1.1 implementation

1. We need partition the bigger part of the whole dataset , so we need to decide which side has a larger size. This is done by gathering the students_count and tas_count from all processes and comparing the summation.

2. Once we decided which part to be partitioned, for example, if total student count is larger, we need gather all tas record from other process. This is implemented by
(a) Committing a MPI_Datatype person_type, as the data type we want to send is a struct, not a MPI primitive data type.
(b) Communicating with other processes, gathering and sending all tas' records by MPI_Allgatherv().

3. Since the number of ta count may differ from file to file , then we cannot use MPI_Allgather, and also we need to calculate the replacement for it.

4. Once we are done with the data collection . Just call join_f() on the data and compute the join result we have.
In the above example , we will call join_f() on partitioned students record (which means students record in each process is different) , and a replicated ta records(which means the ta records are the same in each process).

3.1.2 Some reason to choose implementation

1. At first, we need gather other process's student count or ta count . We chose the function MPI_Allgather() instead of MPI_gather(). The reason is that we need to malloc an array of the sum of counts right after. If we choose MPI_gather(), we will need another communication to send the malloc size to other processes.

2. We used MPI_Allgatherv() instead of MPI_Allgather() because the data size of each process may differ.

The efficiency for Fragment-and-Replicate Join on this scale of data is not good. I've tested the program without f_join function. Almost half of the running time is spent on communication.

3.2 Symmetric Partitioning

Pretty straight forward. Just run join_f on each process on their own data. Then collect the result from by MPI_Reduce. We don't need MPI_Allreduce here, since we only want root process to print out result.

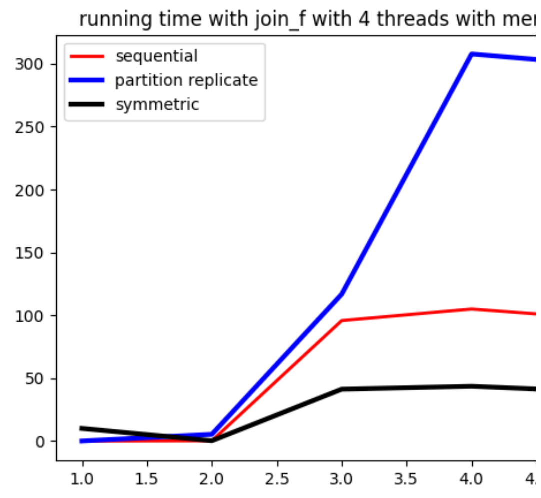

Figure 1: 4 threads

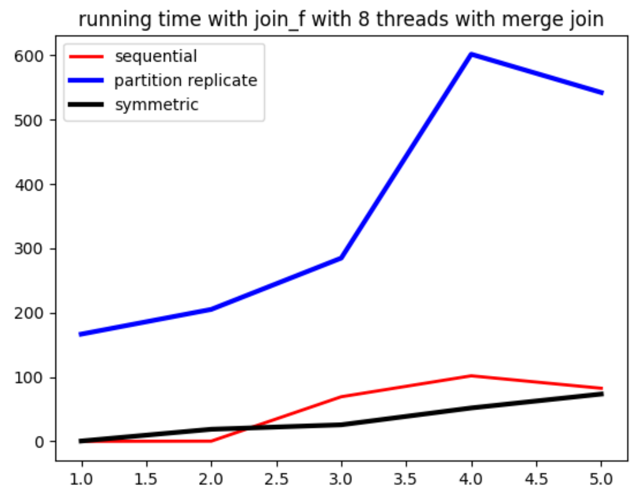

Figure 2: 8 threads

Part4-Analysis

4.1 Run Time Analysis

In Figure 1 and Figure 2 above, x-axis represents the index of dataset, and y-axis is run time taken in milliseconds. Figure 1 shows that, although with multi processes, replicate partitioning on 5 datasets is slower than sequential method and symmetric partitioning, and symmetric partitioning is always the fastest. I believe the slowness of replicate partitioning is caused by communication. Then I tried to run the test with 8 processes and end up at Figure 2. It seems like communication cost is pretty large even on the smallest dataset (dataset1).

Then I run TAU and pprof to conduct a performance analysis . The Figure 3 shows run time of each function for symmetric join. Most of the run time is dedicated by initialization step like load_data (shows I/O is even more expensive then communication.)

The Figure 4 shows the function running time for partition replicate method on the same data set. Beside the initialization, the most time consuming function is MPI_ALLgatherrv(). Which support the guess that communication between each process is expensive.

Thanks for your time .

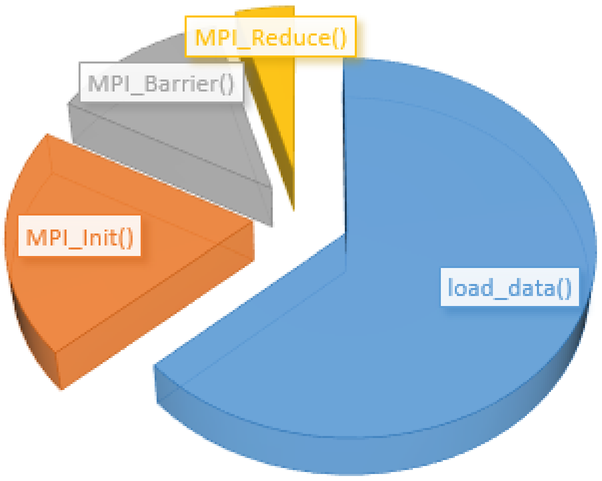

Figure 3:

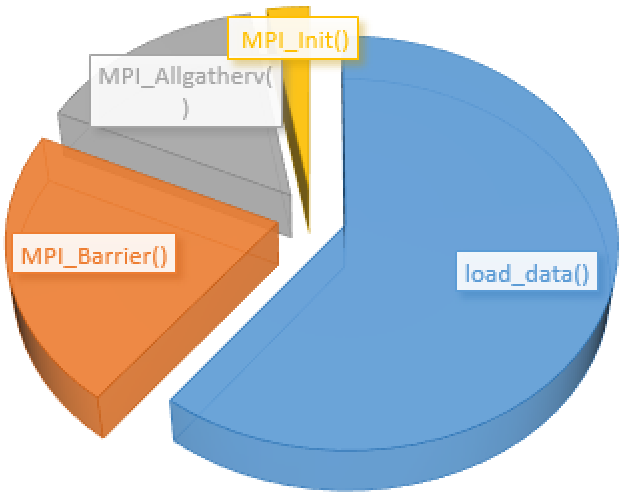

Figure 4: