Lab2 5451

MingxuanJiang
5588030

March 2020

1

We can split the HQuicksort function into 4 steps.
First, each pivot use list[i] as median, then split into two pieces. Second, processes P_i send large numbers in array A_i^{high} to P_{i+q} and send small numbers in array A_i^{low} to P_i. Third, sort numbers in each processes. Fourth, repeat the steps above.
To achieve the goal of quick sort, I write the HQuicksort function.

1. Initialize some parameters like MPI_Comm_rank and MPI_Comm_size to get myid and the number of processes.

2. Get the size of steps in total. For example, if the number of processes is 2, the size should be 1. It satisfies the number of steps is equal to the number of pivots.

3. Write a for loop to show the detail in each step. What is important is that in different steps, we divide pivots into different groups. The number of groups satisfies the formula 2^{step-1}. So I write a for loop of j to repeat 2^{step-1} times in each step. The number of groups is defined as the number after quitting the last bit.

4. In each pivot, sort numbers with a for loop of k. Then, we wanna to change the first bit in the first step, change the second bit in the second step and so on. Also, if the changed bit is equal to 0, it means receiving small numbers below list[0] and sending large numbers. If the changed bit is equal to 1, it means receiving large numbers and sending small numbers.
In this section, I use non-blocking Isend and Irecv function to send and receive data. It can send and receive immediately instead of waiting all complete. Then, use waitall function to wait all finish. After that, depending on the status and datatype, use MPI_Get_count to check whether status of the data is equal to Irecv. Meanwhile, use MPI_Barrier to create a barrier so need to wait all processes finish. Update the length and the data of array Abuf.

5. Finally, use MergeSort function to sort the numbers in each pivot.

2

number of processes	timing	standard deviation
2	6.924×10^{-3}	4.794×10^{2}
4	6.197×10^{-3}	3.609×10^{2}
8	4.079×10^{-3}	2.375×10^{2}
16	5.84710^{-3}	8.830×10^{2}

Here is the timing and standard deviations for sizes in the table above. I use the unbiased formula $s = \sqrt{\frac{\sum_{i=1}^{N}(x_i-\overline{x})^2}{N-1}}$ to calculate the standard deviation. If the sets are always perfectly balanced, the execution time is the sum of time that tasks down in the first pivot (because it participates in the most steps). We can find that when the number of processes is 2, the timing is the least.

3

We can find that with the increasing number of processes, the smaller the standard deviation is, the smaller timing will be. If the performance of load balancing is good, the timing is small. This achieves only when the tasks for each pivot is equal. However, if the tasks for each pivot is different, it is undoubted that the load balancing and timing will drop down. This is because the standard deviation will increase and pivots needs to wait all of them finish the tasks. Last but not least, we can find when using 16 processes, the timing is higher than using 8 processes. It may be because the number to sort and split is highly larger.

4

As I mentioned before, if the tasks for each pivot is equal, the load balancing will be improved. But it is hard to achieve. Maybe better production code about how to split the array helps. Also, with increasing number of processes, the performance will be better. Find a best number of processes is important. Therefore, we don't need to waste time in communication and comparisons and achieve the best timing.