Introduction to Parallel Computing

Fall Semester 2024

Assignment # 2

Po Yu Lai
email:lai00177@umn.edu

1 Parallelization Method

1.1 Steps

1. Generate N/P points inside each processor
2. Gather the value of each processor's last one in the array, broadcast the array to every processors and use non-parallel quicksort to find the median as the pivot.
3. Every processor use this pivot value to partition their own data into two different parts
4. Gather the number of first and second part in each processor and calculate the total number of each section
   (a) Use this total number to decide how many processors is need for each part.
   (b) Once determine the number of processor for each section, redistribute every section to their relative processor.
5. Separate the community into two parts and repeat 1 - 4 until only one precessor in each community.
   (a) Use non-parallel quicksort to solve the rest of part as the number of processor reaches to one.
6. Evenly distribute the points back to the status that contains N/P points inside each group.
7. Use rank 0 processor to gather all points and print out the sorting result.

1.2 MPI

Since MPI is based on NUMA, which means each processor can't directly access other's data, avoiding excess communication becomes an important issue. My methodology is to share the essential data and let each processor compute the value they need instead of using one processor to compute the value and broadcast it to others.

One section that needs communication is the finding of the Pivot part. I tried two different methods: one is to use the point near mean value as the pivot and the other is as above description: use the median. However, it turns out two runtimes are quite similar. Therefore, I use the median one as my final code.

Another part that requires communication is the redistribution of each element to its relative region. In this part, I use MPI_Scan with prefix sum and MPI_Allgather to get the prefix sum array which contains all processor's numbers of each part's element. Using this array, each processor can know how many elements they need to send and the offset from each processor if the are the receiver. The step is shown in fig.1. Finally, we need to redistribute every processor with their original number. Using this idea with prefix sum array, it can be achieved with the same routine in my code.

Last but not least, during the quicksort, the memory is varied based on the original data. However, the data is generated by random numbers and it is hard to determine the maximum size each process should create. I created a parameter called safety factor which multiplies the N/P size of memory for preventing the issue. From my experiment, the safety factor reached to 4 can

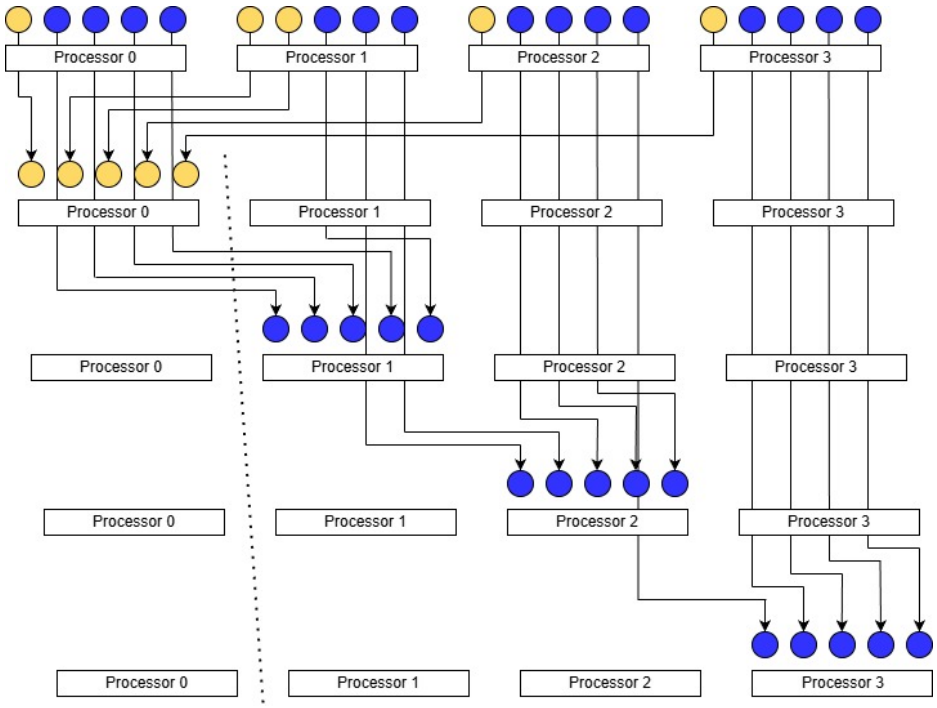

Figure 1: *The steps to redistribute elements to relative processor*

handle most of the cases. I also created a function to check whether the calculation need more space. If so, it will create new array with double size array, move data to new one and free the old one. This method is robust but it will increase the runtime. Therefore I use the safety factor one in my code.

2 Timing Results

# of elements	1 thread	2 threads	4 threads	8 threads	16 threads	speed-up
1,000,000	0.5400	0.2162	0.1379	0.0961	0.0595	9.0756
10,000,000	4.7447	3.2360	1.5436	0.9672	0.5952	7.9716
100,000,000	38.5040	34.8105	15.1890	7.5424	7.47696	8.5954

Table 1: *Timing results for MPI. Speed-up is defined as the time taken by 1 thread divided by the time-taken for 16 threads with a given number of elements, unit:sec*