CSCI 5451 Homework 1 Report

Benjamin Chu

October 2023

1 Parallelization

To determine how a parallel version of this program would be implemented, we can first consider how a serial implementation would work. A serial implementation of this program would take the following steps:

1. Randomly generate the array to sort
2. Apply sort the array with quicksort:
   (a) Select a pivot to partition the array around
   (b) Partition the array with that pivot
   (c) Recursively apply quicksort on the lower partition
   (d) Recursively apply quicksort on the upper partition
3. Print out timing results and save the sorted array

The serial version of this algorithm is implemented in `qs_mpi.c`.

To parallelize this algorithm, we can make the following modifications:

- For step 1, each individual process can work simultaneously work over a separate range to generate some of the array data. Then, each process's work can be combined with a `MPI_Allgatherv` operation.
- For step 2a, each process will randomly select an element of the array. Then, the processes will share the picked elements with a `MPI_Allgather` operation, and the median of those elements will be used as a pivot.
- For step 2b, each process can work simultaneously to partition a separate subsection of the array. Then, they can share their work with each other. First, a `MPI_Allgather` operation is used to share the sizes of the partitions. Then, a `MPI_Allgatherv` operation is used to share the array data. This is done twice for the lower and upper partitions.
- Steps 3c and 3d can be done simultaneously. Once the array is partitioned, we can split the process group into two subgroups. The first group will continue working on the lower partition, and the second group will continue working on the upper partition. Once each group has finished their work, the first process of the second group can send its data back to the first process of the first group.

For steps 1 and 2b, we can simply use linear interpolation to evenly divide the array. Each process begins from an index of `datasize*rank/size` (inclusive), and finishes at the index `datasize*(rank+1)/size` (uninclusive). This ensures that each process works on N/P elements, and that each range differs in size by no more than one element.

2 Results

Threads	Array Size		
	1000000	10000000	100000000
1	0.6248s	7.2618s	83.4536s
2	0.5088s	6.3571s	59.9910s
4	0.3304s	3.3479s	51.7194s
8	0.1794s	1.7134s	18.6531s
16	0.1457s	1.1765s	9.4370s