CSCE 435 Group project

0. Group number: 18

1. Group members:

1. Anna Hartman
2. Tate Moskal
3. Nicole Hernandez
4. Vasudev Agarwal

Our team will communicate primarily through Text message for real-time discussions and updates. We will also use GitHub to track code contributions and Trello for project management, ensuring transparency and accountability for task assignments.

2. Parallel Sorting Algorithms

2a. Brief project description

In this project, we will be comparing the efficiencies of different sorting algorithms running in parallel. The algorithms will be implemented and tested on parallel architectures, such as multi-core processors and possibly distributed systems. Each team member will focus on one of the following algorithms:

- Bitonic Sort: Anna Hartman
- Sample Sort: Vasudev Agarwal
- Merge Sort: Nicole Hernandez
- Radix Sort: Tate Moskal

The comparison will assess execution time, we will evaluate the efficiency on different input sizes (small, medium, large) to measure scalability, and analyzing how these algorithms leverage parallelism to improve sorting performance.

2b. Pseudocode for each parallel algorithm

2b.1 Bitonic Sort In Bitonic sort, a bitonic sequence is built (a sequence that first increases and then decreases) and sorted by merging. Within multiple threads, chunks of the given sequence are sorted into bitonic order. The merging process is carried out in parallel, merging each piece into a large bitonic sequence. Threads should be synchronized. The entire algorithm is carried out recursively in order to build the final bitonic sequence.

1. Divide the given array to be sorted into parallel chunks with corresponding threads
2. For each thread ands its piece (in parallel): sort the piece into bitonic order by recursively splitting the piece into two halves and sorting the first half into ascending order and the second into descending order, and then merging the two haves into a bitonic sequence.

3. Synchronize threads by ensuring each thread has completed its sorting before continuing on.
4. Merge all of the pieces bitonically (in parallel), for each chunk and thread: compare and swap such that if the current chunk is in the lower half, merge it in ascending order, and if it is in the upper half, merge it in descending order. Recursively merge the two halves of the sequence to ensure they are fully sorted in either ascending or descending order. This should be carried out log(array_size) times, as each chunk doubles in size after being recursively merged.
5. Output the result.

2b.2 Sample Sort In sample sort a large dataset is divided into smaller paritions, after which each partition is sorted independently, and then these sorted paritions are merged to obtain the final sorted result.

1. Splitting the given dataset into equally divided smaller segments, where each segement is given to a processor
2. Run basic sorting algorithm on each of the segments, where each processor is handling its piece independently.
3. From these sorted segments, each processor selects few samples. then, MPI communication is used to collect samples from all processors
4. Sort the selected samples, which will help us establish a global order among the samples.
5. From the sorted samples, we pick few speical elements. which act as a pivot. which are shared with all processors. MPI_Bcast is used to broadcast the pivots to all processors.
6. Each processor takes its ordered segment and divides it into subsegments based on the choicen pivot.
7. Now, processors share their ordered segments globally with the corresponding processor based on the segment number. using MPI_Alltoall
8. Finally, each processor merges and sorts the recieved elements.

2b.3 Merge Sort Parallel Merge Sort uses the divide and conquer technique, recursively dividing the dataset into smaller parts, sorting them, and merging the results where in parallel ver. it is distributed across multiple processors. Parallel Merge Sort will be implemented using MPI. Each processor will perform its own sorting idenpendently, merging sorted data using MPI communication.

1. Start MPI for Communication between processors.
2. Identify if the process is the master (rank 0) or worker (rank > 0)
3. The master splits the dataset into smaller parts
4. The master sends each part to a worker process
5. Each worker process sorts its part of the dataset independently
6. Workers send their sorted parts back to the master
7. The master merges all sorted parts into one sorted array
8. Close MPI after sorting is complete

2b.4 Radix Sort Radix Sort is an algorithm that sorts by processing through individual digits, sorting along the way. The process can be sped up by allowing each processor to handle a portion of the total array. By sorting a subarray and keeping note of the order of subarray chunks being sorted in each processor, they can be placed accordingly back into the main array. While this example sorts via binary, the process can account for numbers of any base as long as the # of arrays corresponds correctly.

1. Initialize MPI for multiprocessor communication
2. Convert array digits into binary (helps with initial implementation).
3. Find the maximum element and its # of digits.
4. Begin iterating through digits starting at the least significant digit up to the maximum digit significance.
5. Split the array into subarrays depending on the # of processors used and send to workers, keeping track of the order in which each subarray gets sent where.
6. Each worker will sort its subarray into 2 arrays, the first with digits that are 0, the second with digits that are 1.
7. Worker returns arrays and master combines the 0 array in order of worker process, then repeats for the 1 array.
8. Repeat with the next digit until all digits places have been parsed.
9. End MPI once complete.

2c. Evaluation plan - what and how will you measure and compare

- Input sizes, Input types (Alter input array to be: random, sorted, reverse sorted,...)
- Strong scaling (same problem size, increase number of processors/nodes)
- Weak scaling (increase problem size, increase number of processors)
- Parallelization strategies (master/worker vs SPMD, calculating speedup and runtime differences)
- Communication strategies (collectives vs point-to-point, measure runtime differences between code for each communication strategy)