CSCE 435 Group project

0. Group number: 10

1. Group members:

1. Thomas Zheng
2. Albert Yin
3. Paul Bae
4. Krish Chhabra

For group communication, everyone in the group has exchanged phone numbers and is part of an iMessage group chat. This is how we will be communicating throughout the project. If the need to have a meeting arises, we will arrange to meet in-person.

2. Project topic (e.g., parallel sorting algorithms)

2a. Brief project description (what algorithms will you be comparing and on what architectures)

Bitonic Sort

Bitonic sort is a parallel sorting algorithm that works well on distributed systems that support parallel computation. It sorts a sequence of numbers using a series of compare-and-swap operations, following a divide-and-conquer approach. It works efficiently with bitonic sequences, which are sequences that first monotonically increase then monotonically decrease, also vice versa. The time complexity of Bitonic Sort is O(logˆ2 n) which is efficient for paralle computation compared to other algorithms like Bubble Sort O(nˆ2).

Sample Sort

Sample sort is a divide-and-conquer sorting algorithm that provides a more statistical approach to bucket sort. The efficiency of bucket sort is heavily dependent on the distribution of elements amongst the selected buckets; however, evenly distributing the buckets if you preselect bucket ranges would require domain knowledge on what elements the input arrays will contain. Instead, sample sort leverages sample collection in an attempt to estimate bucket ranges that will provide an even distribution of elements amongst buckets. Essentialy, we draw a sample of elements (with a certain number of elements being selected from each segment of the input array), sort this sample, select pivots from the sorted sample, distribute the elements amongst buckets delineated by these pivots, perform a comparitive sorting algorithm on each bucket, then merge the sorted buckets back together.

Merge Sort

Merge sort is a classical divide-and-conquer sorting algorithm that splits an input array into smaller subarrays, recursively sorts those subarrays, and then merges them back together to form a sorted array. The process begins by dividing the array into two halves repeatedly until each subarray contains only a single element. Once the array is split into minimal subarrays, the merge phase starts. During the merge phase, adjacent subarrays are combined in sorted order by comparing the smallest elements of each subarray and appending them into a new array in sequence. This merging process continues until the entire array is reassembled in sorted order. Merge sort operates with a consistent time complexity of O(n log n) and is highly efficient for large datasets, as it guarantees stable sorting. Its particularly advantageous when working with data that doesn't fit into memory all at once, as it can handle external sorting scenarios effectively. Unlike algorithms that rely on element comparisons for placement, merge sort inherently ensures ordered results by merging sorted partitions, making it a robust and dependable sorting strategy for distributed processing.

Radix Sort

Radix sort is a non-comparative sorting algorithm that orders elements by processing them digit by digit. Radix sort operates by sorting numbers based on individual digits, starting from the least significant digit (LSD) and moving towards the most significant digit (MSD). It is a non comparative sorting algorithm that ends up being slower than comparison algorithmns in most situations.

2b. Pseudocode for each parallel algorithm

Bitonic Sort

1. Initialize MPI environemnt
2. Distribute the input array into available processes
3. Local sort using a sequential version of Bitonic Sort
4. Bitonic merge between processes
5. After each process sorts its part, the results need to be gathered and redistrbuted across processes for the next sort
6. Finalize MPI

```
// Initialize MPI / rank / number of processes
// totalSize is length of the vector
if (rank == 0) {
    arr.resize(totalSize);
    for (int i = 0; i < totalSize; i++>){
        arr[i] = rand() % 1000;
    }
}

// Split the array based on numprocesses
```

```
int localSize = totalSize / numProcesses;
vector<int> localArr(localSize);

// Use MPI Scatter to distribute the array across processes
// Bitonic Sort and merge between processes
for (int k = 2; k <= numProcesses; k\*=2){
    for (int j = k/2; j > 0; j/=2) {
        partner = rank ^ j
        if (rank < partner) {
            mergeLow(localArr, recvArr);
        }
        else {
            mergeHigh(localArr, recvArr);
        }
    }
}

// MPI Gather sorted arrays on the root processses
// Output the sorted array
if (rank == 0) {
    cout << "Sorted array";
    for (int i = 0; i < totalSize; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

// Finalize MPI
```

Sample Sort

```
Choose Some Constant k to be the Oversampling Ratio (i.e. Number of Elements Sampled From Ea


Sample_Sort(arr, n, p, rank):
    Collect Personal Sample of k Elements as my_sample

    Allocate Array of Size p - 1 as pivots

    if rank == 0:
        Allocate Array of Size k * p as sample
        Gather Samples from All Processes into sample (MPI_Gather)

        Sort sample Using Some Comparison-Based Sort

        Set pivots to Contain [sample[k], sample[2k], ..., sample[(p - 1) * k]]
```

```
    Send pivots to Other Processes (MPI_Send)

else:
    Send my_sample to Process 0 (MPI_Gather)
    Recieve pivots from Process 0 (MPI_Send)

Allocate Array of p Vectors, Each of Size n / p^2 as buckets (Each Row Represents a Buck

for i from 0 to n / p - 1:
    elem = arr[i]
    found_bucket = false

    for bucket from 0 to p - 2:
        if elem < pivots[bucket]:
            found_bucket = true
            buckets[bucket].push_back(elem)
            break

    if not found_bucket:
        buckets[p - 1].push_back(elem)

my_bucket_size = 0
if rank == 0:
    Copy Elements of buckets[0] into arr
    my_bucket_size = buckets[0].size()

    for i from 1 to p - 1:
        Receive Size of Next Send from Process j into curr_size (MPI_Recv)
        if curr_size > 0:
            Receive curr_size Elements from Process j into arr + my_bucket_size (MPI_Rec
            my_bucket_size += curr_size

    for i from 1 to p - 1:
        Send Size of buckets[i] to Process i (MPI_Send)
        if buckets[i].size() > 0:
            Send Elements of buckets[i] to Process i (MPI_Send)

else:
    Send Size of buckets[0] to Process 0 (MPI_Send)
    Send Elements of buckets[0] to Process 0 (MPI_Send)

    my_bucket_cap = n / p
    for i from 1 to p - 1:
        if i == rank:
            if buckets[i].size() > my_bucket_cap:
                Resize arr to buckets[i].size()
```

```
                Copy Elements of buckets[i] into arr
                my_bucket_size = buckets[i].size()

                for j from 1 to p - 1:
                    Receive Size of Next Send from Process j into curr_size (MPI_Recv)
                    if curr_size > 0:
                        if my_bucket_size + curr_size > my_bucket_cap:
                            Resize arr to my_bucket_size + curr_size

                        Receive curr_size Elements from Process j into arr + my_bucket_size
                        my_bucket_size += curr_size

            else:
                Send Size of buckets[i] to Process i (MPI_Send)
                if buckets[i].size() > 0:
                    Send Elements of buckets[i] to Process i (MPI_Send)

    if my_bucket_size > 1:
        Sort [arr[0], arr[1], ..., arr[my_bucket_size - 1]] Using Some Comparison-Based Sort

    if rank == 0:
        arr[my_bucket_size++] = pivots[0]
        for i from 1 to p - 1:
            Receive Size of Next Send from Process i into curr_size (MPI_Recv)
            if curr_size > 0:
                Receive curr_size Elements from Process i into arr + my_bucket_size (MPI_Rec
                my_bucket_size += curr_size
            arr[my_bucket_size++] = pivots[i]

    else:
        Send my_bucket_size to Process 0 (MPI_Send)
        if my_bucket_size > 0:
            Send [arr[0], arr[1], ..., arr[my_bucket_size - 1]] to Process 0 (MPI_Send)


Main:
    Initialize MPI Environment (MPI_Init)
    Retrieve Number of Processes as p (MPI_Comm_Size)
    Retrieve Process Rank as rank (MPI_Comm_Rank)

    if rank == 0:
        Generate Array of Size n as arr

        for i from 0 to p-1:
            Send arr[i * p] Through arr[(i + 1) * p - 1] to Process i (MPI_Send)
```

```
        Sample_Sort(arr, n, p, rank, k)

        Verify arr is Properly Sorted

    else:
        Retrieve My Portion of Array (Size n / p) from Process 0 into arr (MPI_Recv)

        Sample_Sort(arr, n, p, rank, k)

    Finalize MPI (MPI_Finalize)
```

Merge Sort

1. Initialize MPI environment.
2. Divide the array of numbers among available processes.
3. Local Sorting using a sequential Merge Sort
4. Exchange and merge sorted arrays between processes.
5. Gather the sorted subarrays at the root process.
6. Output the sorted array at the root process.
7. Finalize MPI

```
// Initialize MPI / rank / number of processes
// totalSize is length of the vector
if (rank == 0) {
    arr.resize(totalSize);
    for (int i = 0; i < totalSize; i++>){
        arr[i] = rand() % 1000;
    }
}

// Split the array based on numprocesses
int localSize = totalSize / numProcesses;
vector<int> localArr(localSize);

// Use MPI Scatter to distribute the array across processes

// Merge Sort / Merge with other processes block
for (int step = 1; step < numProcesses; step *= 2) {
        if (rank % (2 * step) == 0) {
            if (rank + step < numProcesses) {

                vector<int> recvArr(localSize);
                // Use MPI Recv to get other array from partner process store in recvArr

                // Merge the received array with the local array
                vector<int> mergedArr(localArr.size() + recvArr.size());
```

```
                merge(localArr, recvArr, mergedArr);
                localArr = mergedArr;
            }
        }
        else {
            // Send local array to the partner process and exit the loop
            int partner = rank - step;
            // Use MPI send to send the sorted array to its partner process
            break;
        }
        // Double the size of the local size after merging
        localSize *= 2;
    }

// MPI Gather sorted arrays on the root processses

if (rank == 0) {
    // Output the sorted array (on root process)
    cout << "Sorted array: ";
    for (int i = 0; i < totalSize; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

// Finalize MPI
```

Radix Sort

1. Initialize MPI environment.
2. Divide the array of numbers among available processes.
3. Local Sorting (using Counting Sort by each process):
4. Each process sorts its local portion of the array based on the current digit (using a stable sort like counting sort).
5. After each process sorts its part, the results need to be gathered and redistributed across processes for the next digit sort.
6. Repeat for Each Digit

```
//Initialize MPI
//Initialize rank
//Initialize numprocesses
//totalSize is length of vector

 if (rank == 0) {
    // Generate or input the array on the root process
    for (int i = 0; i < totalSize; i++) {
```

```
        arr[i] = rand() % 1000;  // Example random values
    }
}

//Split the array based on the processes
int localSize = totalSize / numProcesses;
vector<int> localArr(localSize);
//Use MPI Scatter here to distribute the array to the different processes

//Perform Radix Sort
// Broadcast the maximum number in each Radix sort
// Sort each digit starting from the LSD
    // If rank == 0
        //MPI Gather all sub sorted arrays into one global array
    // Else
        //MPI Gather the individual process sorted array
    //MPI Scatter the sorted array to all processes for next iteration

//MPI Gather sorted arrays on the root process

if (rank == 0) {
    // Output the sorted array (on root process)
    cout << "Sorted array: ";
    for (int i = 0; i < totalSize; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

Repeat the local sort and gather steps for each digit, starting from the least significant to the most significant digit.

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

All evalution will be performed on TAMU's Grace. We will use Caliper for measuring execution time and Thicket for plotting and analysing measurements.

Input sizes

For testing, we will use input arrays of 7 different size: 2^16, 2^18, 2^20, 2^22, 2^24, 2^26, and 2^28 elements. This will allow us to evaluate our sorts using both strong and weak scaling as well as providing a good range of problem sizes.

Input types

In terms of ordering, our input arrays will be of 4 different types: random, sorted, reverse sorted, and sorted with 1% perturbed. This will allow us to observe the

strengths and weaknesses of each sorting algorithm and reason about what form input array each is more tailored to solving.

Each of our input arrays will be of an integer type, and the actual elements stored will be the same for each of the 4 orderings to avoid adding additional factors to our evaluation.

Example:

- Random: [5, 8, 4, 3, 1, 7, 2, 6]
- Sorted: [1, 2, 3, 4, 5, 6, 7, 8]
- Reverse Sorted: [8, 7, 6, 5, 4, 3, 2, 1]
- 1% Perturbed: [1, 2, 3, 7, 5, 6, 4, 8]

Scaling

In our performance analysis, we will use both strong scaling (comparing performance on same problem size as the number of processors increases) and weak scaling (comparing performance as both problem size and number of processors increase).

For each sort, we will collect data from the execution with 10 different numbers of processors: 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 processors.

- Keeping the number of processors as powers of 2 greatly simplifies the implementation of our algorithms.
- It is crucial to evaluate with at least 64 processors as this is the smallest power of two that requires more than one node to run (can see effect of inter-node communication).