

CSCE 435 Group project

0. Group number: 12

12

1. Group members:

1. Christion Bradley
2. Warren Wu
3. Sua Bae
4. Kunal Jain

- communication via: iMessage & Discord

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

Parallel sorting is incredibly important for processing large amounts of data efficiently on distributed systems/multicore processors.

This project focuses on various parallel sorting algorithms and their implementations, with the goal of understanding how they can be optimized for performance on parallel architecture. Specifically, we will compare the parallel performance, implementation complexity, and memory usage of Bitonic Sort, Sample Sort, Merge Sort, and Radix Sort using the Message Passing Interface.

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

- Bitonic Sort: Bitonic sort repeatedly creates bitonic sequences, which first monotonically increases and then monotonically decreases, then merges them into a fully sorted sequence through performing pairwise comparisons and swaps between elements. This sorting algorithm will be using MPI on TAMU Grace Cluster by dividing the array into multiple processes, locally sorting the subarrays within each process, then finally merging them in a parallel fashion, until the entire array is sorted.
- Sample Sort: An irregular algorithm similar to parallelized bucket sort except there are $p-1$ pivot points. Sample sort first splits an array of size n into different processes that is then sorted locally. From there the "splitters" are found and the local sequence is split into different sequences according to the splitters. Lastly, for each of the sequences they are merged locally
- Merge Sort: For a list of size n , Merge Sort recursively splits the list into n sublists, where each sublist contains a single element and is therefore sorted. These sublists are then combined into larger lists where smaller elements appear before larger elements. This is done until one list remains, which will be the final sorted list. We will be implementing this algorithm on the TAMU Grace Cluster using MPI to communicate between processes.
- Radix Sort: Radix sort sorts numbers digit by digit starting from the least to most significant digit. In parallelized radix sort, the array will be distributed among subprocesses, with each process performing the sorting for one or more digits. After sorting based on the current digit, the processes will communicate with each other to exchange data to ensure that the elements are properly distributed for the next iteration. This algorithm will also be implemented using MPI and ran on the Grace Cluster.

2b. Pseudocode for each parallel algorithm

Bitonic Sort:

```
void bitonic_merge(int a[], int low, int cnt, int dir)
{
    if (cnt > 1)
    {
        int k = cnt / 2;
        for (int i = low; i < low+k; i++)
            if (dir == (a[i] > a[i+k])) {
                int temp = arr[i];
                arr[i] = arr[i + k];
                arr[i + k] = temp;
            }
        bitonic_merge(a, low, k, dir);
        bitonic_merge(a, low+k, k, dir);
    }
}

void bitonic_sort(int a[], int low, int cnt, int dir)
{
    if (cnt > 1)
    {
        int k = cnt / 2;
        bitonic_sort(a, low, k, 1);
        bitonic_sort(a, low+k, k, 0);
        bitonic_merge(a, low, cnt, dir);
    }
}

int main() {
    int rank, size, n_workers, n;
    int* arr = array to be sorted;
    int* workers_arr = NULL;
    int* final_list = NULL;

    MPI_Init();
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n_workers);

    if (rank == 0) {
        n = sizeof(arr) / sizeof(arr[0]);
    }

    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    int local_size = n / n_workers;
    workers_arr = (int*)malloc(local_size * sizeof(int));

    MPI_Scatter(arr, local_size, MPI_INT, workers_arr, local_size, MPI_INT, 0,
MPI_COMM_WORLD);
    bitonic_sort(workers_arr, 0, local_size, 1);
```

```

    if (rank == 0) {
        final_list = (int*)malloc(n * sizeof(int));
    }

    MPI_Gather(workers_arr, local_size, MPI_INT, final_list, local_size, MPI_INT, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        bitonic_sort(final_list, 0, n, 1);
        free(final_list);
    }

    free(workers_arr);

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}

```

Merge Sort:

```

void merge(int arr[], int temp[], int left, int right) {
    int b = left;
    int l = left;
    int middle = (left + right) / 2;
    int r = middle + 1;

    while (l <= middle && r <= right) {
        if (arr[l] <= arr[r]) {
            temp[b++] = arr[l++];
        } else {
            temp[b++] = arr[r++];
        }
    }

    while (l <= middle) {
        temp[b++] = arr[l++];
    }

    while (r <= right) {
        temp[b++] = arr[r++];
    }

    for (int i = left; i <= right; i++) {
        arr[i] = temp[i];
    }
}

void merge_sort(int arr[], int temp[], int left, int right) {
    if (left < right) {

```

```

        int middle = (left + right) / 2;
        merge_sort(arr, temp, left, middle);
        merge_sort(arr, temp, middle + 1, right);
        merge(arr, temp, left, right);
    }
}

void merge_sort(int* arr, int* temp, int left, int right){
    if (left < right) {
        middle = (left + right) // 2;
        merge_sort(arr1, arr2, left, middle);
        merge_sort(arr1, arr2, middle + 1, right);
        merge(arr1, arr2, left, right);
    }
}

int main() {
    arr = array of elements to be sorted;
    MPI_Init();
    int rank, n_workers;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n_workers);

    int size = arr.size() / n_workers;
    int *workers_arr = malloc(size * sizeof(int));
    MPI_Scatter(arr, size, MPI_INT, workers_arr, size, MPI_INIT, 0, MPI_COMM_WORLD);

    int *w_arr = malloc(size * sizeof(int));
    mergeSort(workers_arr, w_arr, 0, size - 1);

    if (rank == 0) {
        final_list = malloc(arr.size() * sizeof(int));
    }

    MPI_Gather(worker_arr, size, MPI_INT, final_list, size, MPI_INT, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        int* helper_arr malloc(arr.size() * sizeof(int));
        mergeSort(final_list, helper_arr, 0, arr.size() - 1);
        print(final_list);
        free(final_list); free(helper_arr);
    }

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
}

```

Sample Sort

```
int main() {
```

```

arr = array of elements to be sorted;
MPI_Init();
MPI_Comm_rank(...);
MPI_Comm_size(...);

int localSize = arr.size() / size;
localData = sequence of elements of size localSize;
MPI_Scatter(arr, localSize, MPI_INT, localData, localSize, MPI_INT, 0,
MPI_COMM_WORLD);

sort(localData);

localSortedData[arr.size()];
MPI_Gather(localData, arr.size(), MPI_INT, localSortedData, arr.size(),
MPI_INT, 0, MPI_COMM_WORLD);

splitter[size-1];
for i=0 to size-1 {
    splitter[i] = localData[arr.size()/(size*size) * (i+1)];
}

allSplitters[size*(size-1)];
MPI_Gather(splitter, size-1, MPI_INT, allSplitters, size-1, MPI_INT, 0,
MPI_COMM_WORLD);

sort(allSplitters);
for i=0 to size-1 {
    splitter[i] = allSplitters[(size-1)*(i+1)];
}
MPI_Bcast(splitter, size-1, MPI_INT, 0, MPI_COMM_WORLD);

buckets[size];
for i=0 to localSize {
    int bucketIndex = 0;
    while (bucketIndex < size-1 and localData[i] > allSplitters[bucketIndex])
{
        bucketIndex++;
    }
    buckets[bucketIndex].append(localData[i]);
}

bucketBuffer[arr.size()+size]
MPI_Alltoall(buckets, localSize+1, MPI_INT, bucketBuffer, localSize+1,
MPI_INT, MPI_COMM_WORLD);

localBucket[2*arr.size()/size];
counter = 1;
for i=0 to size {
    incrementer = 1;
    for j=0 to bucketBuffer[(arr.size()/size + 1)*i] {
        localBucket[counter] = bucketBuffer[(arr.size()/size +
1)*i+incrementer];
        incrementer++;
        counter++

```

```

    }
}
localBucket[0] = counter-1;
sort(localBucket);

sortedArr[arr.size()];
MPI_Gather(localBucket, arr.size(), MPI_INT, sortedArr, arr.size() , MPI_INT,
0, MPI_COMM_WORLD);

MPI_Finalize();
return 0;
}

```

Radix Sort:

```

int getMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

void countSort(int arr[], int n, int exp) {
    int* output = (int*)malloc(n * sizeof(int));
    int count[10] = {0};

    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    for (int i = 0; i < n; i++)
        arr[i] = output[i];

    free(output);
}

void radixSort(int arr[], int n) {
    int m = getMax(arr, n);

    for (int exp = 1; m / exp > 0; exp *= 10) {
        countSort(arr, n, exp);
    }
}

```

```

int main(int argc, char** argv) {
    int rank, size, n;
    int* arr = NULL;
    int* local_arr = NULL;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        n = /* size of the array */;
        arr = (int*)malloc(n * sizeof(int));
    }

    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    int local_size = n / size;
    local_arr = (int*)malloc(local_size * sizeof(int));

    MPI_Scatter(arr, local_size, MPI_INT, local_arr, local_size, MPI_INT, 0,
MPI_COMM_WORLD);

    radixSort(local_arr, local_size);

    MPI_Gather(local_arr, local_size, MPI_INT, arr, local_size, MPI_INT, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        radixSort(arr, n);
    }

    free(local_arr);
    if (rank == 0)
        free(arr);

    MPI_Finalize();
    return 0;
}

```

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

- Input sizes, Input types: We will measure the time in seconds that it takes for each algorithm to sort a given array of elements. We will sort arrays of consisting of integers with size 128, 1024, and 8192 to determine the speed of each algorithm.
- Strong scaling (same problem size, increase number of processors/nodes) We will increase the number of processes by powers of two: 2, 4, 8, 16, 32, and 64 processes. If an algorithm parallelizes well, then increasing the number of processes should continue to decrease the computation time.

- Weak scaling (increase problem size, increase number of processors) We will increase the size of the array (128, 1024, and 8192) alongside increasing the number of processes (2, 4, 8, 16, 32, 64) in order to determine how well these algorithms can be parallelized.

3a. Caliper instrumentation

Bitonic Sort Implementation Description:

I used MPI for parallel processing to implement Bitonic Sort, distributing the original array into multiple worker processes to enable local sorting. The master process initializes the array based on user input of list type (random, sorted, reverse sorted, or 1% perturbed) and scatters the array among all processes. Each process performs local bitonic sorting. During the bitonic sort, the array is recursively divided into two parts: one sorted in ascending order and the other in descending order. They are then merged into a single sorted sequence during the bitonic merge, where pairs of elements from the two halves are compared and swapped based on the desired sorting order. This process is repeated recursively until the entire array is sorted. After each worker process finishes locally sorting its subarray, the sorted subarrays are gathered back at the master process, which performs a final bitonic sort to merge the results. After the final bitonic sort, the correctness of sorting is checked by going through the final list and checking if the array is actually sorted.

```
MPI Bitonic Sort Calltree:
0.011 main
├─ 0.001 data_init_runtime
├─ 0.008 MPI_Bcast
├─ 0.001 comm
│   └─ 0.001 comm_large
│       ├── 0.001 MPI_Scatter
│       └─ 0.000 MPI_Gather
├─ 0.002 comp
│   └─ 0.002 comp_large
└─ 0.000 correctness_check
0.037 MPI_Barrier
0.000 MPI_Finalize
0.000 MPI_Initialized
0.000 MPI_Finalized
0.001 MPI_Comm_dup
```

Merge Sort Implementation Description:

I used MPI to implement the Parallel Merge Sort Algorithm. I distribute a portion of the input array to each worker, and each worker uses merge sort to sort their portion of the array. This consists of recursively dividing an array of size n into n sub arrays. Each subarray contains one element, and is therefore sorted. Then each element is merged into one array, each process will return a single sorted array. Once all processes have sorted their portion, they then receive indices of the final array that they will then merge. Each level is then merged by a pair of processes until the entire final array has been sorted.

```
0.154 main
├─ 0.039 data_init_runtime
```



```

└─ 0.147 comm
  └─ 0.147 comm_large
    └─ 0.085 MPI_Scatter
    └─ 0.000 MPI_Gather
    └─ 0.062 MPI_Gatherv
└─ 0.006 comp
  └─ 0.004 comp_small
  └─ 0.003 comp_large
    └─ 0.002 MPI_Recv
    └─ 0.000 MPI_Send
└─ 0.003 correctness_check
0.000 MPI_Finalize
0.000 MPI_Initialized
0.000 MPI_Finalized
0.023 MPI_Comm_dup

```

Radix Sort Implementation Description:

I implemented Radix Sort using MPI to leverage parallelism. The master process is responsible for initializing the array and scatters it among all available processes. Each process receives a portion of the array, which it sorts locally using the Radix Sort algorithm. Radix Sort sorts the array based on digits, starting from the least significant digit (LSD) to the most significant (MSD). This is accomplished in multiple passes, where each pass uses the countSort function to sort the elements by a specific digit determined by the current exponent value. Each process sorts its subarray independently, and once each process completes the sorting of its local data, the subarrays are gathered back by the master process using MPI Gather. The master process performs a final Radix Sort on the gathered array to ensure that the entire data set is sorted.

```

0.017 main
└─ 0.000 data_init_runtime
└─ 0.002 MPI_Bcast
└─ 0.000 comm
  └─ 0.000 comm_small
  └─ 0.000 MPI_Scatter
  └─ 0.000 comm_large
    └─ 0.000 MPI_Gather
└─ 0.000 comp
  └─ 0.000 comp_small
  └─ 0.000 comp_large
└─ 0.000 correctness_check
0.000 MPI_Finalize
0.000 MPI_Initialized
0.000 MPI_Finalized
0.015 MPI_Comm_dup

```

Sample Sort Implementation Description:

Used MPI for parallel processing. Split original array by the number of processes and sorted the local arrays. Then sampled the data to determine which splitters should be used for determining the buckets. Once

splitters were calculated, array was partitioned such that each process was a bucket holding a value from a range between the splitters. Once all the processes had the correct values, they were copied back to the local array using Gather and Gatherv from MPI due to new local array size.

```
MPI Sample Sort Calltree:
0.011 main
├─ 0.000 data_init_runtime
├─ 0.001 comm
│   └─ 0.001 comm_large
│       └─ 0.001 MPI_Scatter
├─ 0.006 comp
│   └─ 0.006 comp_small
│       ├── 0.001 MPI_Gather
│       ├── 0.001 MPI_Bcast
│       ├── 0.001 MPI_Alltoall
│       └─ 0.001 MPI_Alltoallv
├─ 0.000 MPI_Gather
└─ 0.001 MPI_Gatherv
0.000 MPI_Finalize
0.000 MPI_Initialized
0.000 MPI_Finalized
0.001 MPI_Comm_dup
```

3b. Collect Metadata

Bitonic Sort:

```
{
  "cali.caliper.version": {
    "1793904175": "2.11.0"
  },
  "mpi.world.size": {
    "1793904175": 32
  },
  "spot.metrics": {
    "1793904175":
      "min#inclusive#sum#time.duration,max#inclusive#sum#time.duration,avg#inclusive#sum#time.duration,sum#inclusive#sum#time.duration,variance#inclusive#sum#time.duration,min#min#aggregate.slot,min#sum#rc.count,avg#sum#rc.count,max#sum#rc.count,sum#sum#rc.count,min#scale#sum#time.duration.ns,max#scale#sum#time.duration.ns,avg#scale#sum#time.duration.ns,sum#scale#sum#time.duration.ns"
  },
  "spot.timeseries.metrics": {
    "1793904175": ""
  },
  "spot.format.version": {
    "1793904175": 2
  },
  "spot.options": {
    "1793904175":
```

```

"time.variance,profile.mpi,node.order,region.count,time.exclusive"
  },
  "spot.channels": {
    "1793904175": "regionprofile"
  },
  "cali.channel": {
    "1793904175": "spot"
  },
  "spot:node.order": {
    "1793904175": "true"
  },
  "spot:output": {
    "1793904175": "p32-a65536.cali"
  },
  "spot:profile.mpi": {
    "1793904175": "true"
  },
  "spot:region.count": {
    "1793904175": "true"
  },
  "spot:time.exclusive": {
    "1793904175": "true"
  },
  "spot:time.variance": {
    "1793904175": "true"
  },
  "launchdate": {
    "1793904175": 1729101279
  },
  "libraries": {
    "1793904175": [
      "/scratch/group/csce435-f24/Caliper/caliper/lib64/libcaliper.so.2",
      "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/lib/libmpicxx.so.12",
      "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/lib/release/libmpi.so.12",
      "/lib64/librt.so.1",
      "/lib64/libpthread.so.0",
      "/lib64/libdl.so.2",
      "/sw/eb/sw/GCCcore/8.3.0/lib64/libstdc++.so.6",
      "/lib64/libm.so.6",
      "/sw/eb/sw/GCCcore/8.3.0/lib64/libgcc_s.so.1",
      "/lib64/libc.so.6",
      "/sw/eb/sw/CUDA/12.4.0/extras/CUPTI/lib64/libcupti.so.12",
      "/sw/eb/sw/PAPI/6.0.0-GCCcore-8.3.0/lib/libpapi.so.6.0",
      "/lib64/ld-linux-x86-64.so.2",
      "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/libfabric.so.1",
      "/lib64/libutil.so.1",
      "/sw/eb/sw/PAPI/6.0.0-GCCcore-8.3.0/lib/libpfm.so.4",
      "/lib64/libnuma.so",
      "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libshm-fi.so",
      "/sw/eb/sw/impi/2019.9.304-iccifort-

```

```

2020.4.304/intel64/libfabric/lib/prov/libmlx-fi.so",
    "/lib64/libucp.so.0",
    "/sw/eb/sw/zlib/1.2.11-GCCcore-8.3.0/lib/libz.so.1",
    "/usr/lib64/libuct.so.0",
    "/usr/lib64/libucs.so.0",
    "/usr/lib64/libucm.so.0",
    "/sw/eb/sw/mpi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libverbs-fi.so",
    "/lib64/librdmacm.so.1",
    "/lib64/libibverbs.so.1",
    "/lib64/libnl-3.so.200",
    "/lib64/libnl-route-3.so.200",
    "/usr/lib64/libibverbs/libmlx5-rdmav34.so",
    "/sw/eb/sw/mpi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libpsmx2-fi.so",
    "/lib64/libpsm2.so.2",
    "/sw/eb/sw/mpi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libsockets-fi.so",
    "/sw/eb/sw/mpi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/librxm-fi.so",
    "/sw/eb/sw/mpi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libtcp-fi.so",
    "/usr/lib64/ucx/libuct_ib.so.0",
    "/usr/lib64/ucx/libuct_rdmacm.so.0",
    "/usr/lib64/ucx/libuct_cma.so.0",
    "/usr/lib64/ucx/libuct_knem.so.0",
    "/usr/lib64/ucx/libuct_xpmem.so.0",
    "/usr/lib64/libxpmem.so.0"
  ]
},
"cmdline": {
  "1793904175": [
    "./bitonicsort",
    "65536",
    "random"
  ]
},
"cluster": {
  "1793904175": "c"
},
"algorithm": {
  "1793904175": "bitonic sort"
},
"programming_model": {
  "1793904175": "mpi"
},
"data_type": {
  "1793904175": "int"
},
"size_of_data_type": {
  "1793904175": 4
},
"input_size": {
  "1793904175": 65536
}

```

```

    },
    "input_type": {
      "1793904175": "random"
    },
    "num_procs": {
      "1793904175": 32
    },
    "scalability": {
      "1793904175": "strong"
    },
    "group_num": {
      "1793904175": 12
    },
    "implementation_source": {
      "1793904175": "ai"
    }
  }
}

```

Merge Sort:

```

{
  "cali.caliper.version": {
    "4285520407": "2.11.0"
  },
  "mpi.world.size": {
    "4285520407": 64
  },
  "spot.metrics": {
    "4285520407":
"min#inclusive#sum#time.duration,max#inclusive#sum#time.duration,avg#inclusive#sum
#time.duration,sum#inclusive#sum#time.duration,variance#inclusive#sum#time.duratio
n,min#min#aggregate.slot,min#sum#rc.count,avg#sum#rc.count,max#sum#rc.count,sum#su
m#rc.count,min#scale#sum#time.duration.ns,max#scale#sum#time.duration.ns,avg#scale
#sum#time.duration.ns,sum#scale#sum#time.duration.ns"
  },
  "spot.timeseries.metrics": {
    "4285520407": ""
  },
  "spot.format.version": {
    "4285520407": 2
  },
  "spot.options": {
    "4285520407":
"time.variance,profile.mpi,node.order,region.count,time.exclusive"
  },
  "spot.channels": {
    "4285520407": "regionprofile"
  },
  "cali.channel": {
    "4285520407": "spot"
  },
}

```

```

"spot:node.order": {
  "4285520407": "true"
},
"spot:output": {
  "4285520407": "p64-a1048576.cali"
},
"spot:profile.mpi": {
  "4285520407": "true"
},
"spot:region.count": {
  "4285520407": "true"
},
"spot:time.exclusive": {
  "4285520407": "true"
},
"spot:time.variance": {
  "4285520407": "true"
},
"launchdate": {
  "4285520407": 1729127740
},
"libraries": {
  "4285520407": [
    "/scratch/group/csce435-f24/Caliper/caliper/lib64/libcaliper.so.2",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/lib/libmpicxx.so.12",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/lib/release/libmpi.so.12",
    "/lib64/librt.so.1",
    "/lib64/libpthread.so.0",
    "/lib64/libdl.so.2",
    "/sw/eb/sw/GCCcore/8.3.0/lib64/libstdc++.so.6",
    "/lib64/libm.so.6",
    "/sw/eb/sw/GCCcore/8.3.0/lib64/libgcc_s.so.1",
    "/lib64/libc.so.6",
    "/sw/eb/sw/CUDA/12.4.0/extras/CUPTI/lib64/libcupti.so.12",
    "/sw/eb/sw/PAPI/6.0.0-GCCcore-8.3.0/lib/libpapi.so.6.0",
    "/lib64/ld-linux-x86-64.so.2",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/libfabric.so.1",
    "/lib64/libutil.so.1",
    "/sw/eb/sw/PAPI/6.0.0-GCCcore-8.3.0/lib/libpfm.so.4",
    "/lib64/libnuma.so",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libshm-fi.so",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libmlx-fi.so",
    "/lib64/libucp.so.0",
    "/sw/eb/sw/zlib/1.2.11-GCCcore-8.3.0/lib/libz.so.1",
    "/usr/lib64/libuct.so.0",
    "/usr/lib64/libucs.so.0",
    "/usr/lib64/libucm.so.0",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libverbs-fi.so",

```

```

        "/lib64/librdmacm.so.1",
        "/lib64/libibverbs.so.1",
        "/lib64/libnl-3.so.200",
        "/lib64/libnl-route-3.so.200",
        "/usr/lib64/libibverbs/libmlx5-rdmav34.so",
        "/sw/eb/sw/mpi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libpsmx2-fi.so",
        "/lib64/libpsm2.so.2",
        "/sw/eb/sw/mpi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libsockets-fi.so",
        "/sw/eb/sw/mpi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/librxm-fi.so",
        "/sw/eb/sw/mpi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libtcp-fi.so",
        "/usr/lib64/ucx/libuct_ib.so.0",
        "/usr/lib64/ucx/libuct_rdmacm.so.0",
        "/usr/lib64/ucx/libuct_cma.so.0",
        "/usr/lib64/ucx/libuct_knem.so.0",
        "/usr/lib64/ucx/libuct_xpmem.so.0",
        "/usr/lib64/libxpmem.so.0"
    ]
},
"cmdline": {
    "4285520407": [
        "./mergesort",
        "1048576",
        "random"
    ]
},
"cluster": {
    "4285520407": "c"
},
"algorithm": {
    "4285520407": "merge sort"
},
"programming_model": {
    "4285520407": "mpi"
},
"data_type": {
    "4285520407": "int"
},
"size_of_data_type": {
    "4285520407": 4
},
"input_size": {
    "4285520407": 1048576
},
"input_type": {
    "4285520407": "random"
},
"num_procs": {
    "4285520407": 64
},
"scalability": {

```

```

    "4285520407": "strong"
  },
  "group_num": {
    "4285520407": 12
  },
  "implementation_source": {
    "4285520407": "online"
  }
}

```

Radix Sort

```

{
  "cali.caliper.version": {
    "54582209": "2.11.0"
  },
  "mpi.world.size": {
    "54582209": 2
  },
  "spot.metrics": {
    "54582209":
"min#inclusive#sum#time.duration,max#inclusive#sum#time.duration,avg#inclusive#sum
#time.duration,sum#inclusive#sum#time.duration,variance#inclusive#sum#time.duratio
n,min#min#aggregate.slot,min#sum#rc.count,avg#sum#rc.count,max#sum#rc.count,sum#su
m#rc.count,min#scale#sum#time.duration.ns,max#scale#sum#time.duration.ns,avg#scale
#sum#time.duration.ns,sum#scale#sum#time.duration.ns"
  },
  "spot.timeseries.metrics": {
    "54582209": ""
  },
  "spot.format.version": {
    "54582209": 2
  },
  "spot.options": {
    "54582209":
"time.variance,profile.mpi,node.order,region.count,time.exclusive"
  },
  "spot.channels": {
    "54582209": "regionprofile"
  },
  "cali.channel": {
    "54582209": "spot"
  },
  "spot:node.order": {
    "54582209": "true"
  },
  "spot:output": {
    "54582209": "p2-a1024.cali"
  },
  "spot:profile.mpi": {
    "54582209": "true"
  }
}

```



```

    },
    "spot:region.count": {
      "54582209": "true"
    },
    "spot:time.exclusive": {
      "54582209": "true"
    },
    "spot:time.variance": {
      "54582209": "true"
    },
    "launchdate": {
      "54582209": 1729108987
    },
    "libraries": {
      "54582209": [
        "/scratch/group/csce435-f24/Caliper/caliper/lib64/libcaliper.so.2",
        "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/lib/libmpicxx.so.12",
        "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/lib/release/libmpi.so.12",
        "/lib64/librt.so.1",
        "/lib64/libpthread.so.0",
        "/lib64/libdl.so.2",
        "/sw/eb/sw/GCCcore/10.2.0/lib64/libstdc++.so.6",
        "/lib64/libm.so.6",
        "/sw/eb/sw/GCCcore/10.2.0/lib64/libgcc_s.so.1",
        "/lib64/libc.so.6",
        "/sw/eb/sw/CUDA/12.4.0/extras/CUPTI/lib64/libcupti.so.12",
        "/sw/eb/sw/PAPI/6.0.0-GCCcore-8.3.0/lib/libpapi.so.6.0",
        "/lib64/ld-linux-x86-64.so.2",
        "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/libfabric.so.1",
        "/lib64/libutil.so.1",
        "/sw/eb/sw/PAPI/6.0.0-GCCcore-8.3.0/lib/libpfm.so.4",
        "/sw/eb/sw/numactl/2.0.13-GCCcore-10.2.0/lib/libnuma.so",
        "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libshm-fi.so",
        "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libmlx-fi.so",
        "/sw/eb/sw/UCX/1.9.0-GCCcore-10.2.0/lib/libucp.so.0",
        "/sw/eb/sw/zlib/1.2.11-GCCcore-10.2.0/lib/libz.so.1",
        "/sw/eb/sw/UCX/1.9.0-GCCcore-10.2.0/lib/libuct.so.0",
        "/sw/eb/sw/UCX/1.9.0-GCCcore-10.2.0/lib/libucs.so.0",
        "/sw/eb/sw/UCX/1.9.0-GCCcore-10.2.0/lib/libucm.so.0",
        "/sw/eb/sw/binutils/2.35-GCCcore-10.2.0/lib/libbfd-2.35.so",
        "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libverbs-fi.so",
        "/lib64/librdmacm.so.1",
        "/lib64/libibverbs.so.1",
        "/lib64/libnl-3.so.200",
        "/lib64/libnl-route-3.so.200",
        "/usr/lib64/libibverbs/libmlx5-rdmacv34.so",
        "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libpsmx2-fi.so",

```

```

        "/lib64/libpsm2.so.2",
        "/sw/eb/sw/mpi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libsockets-fi.so",
        "/sw/eb/sw/mpi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/librxm-fi.so",
        "/sw/eb/sw/mpi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libtcp-fi.so",
        "/sw/eb/sw/UCX/1.9.0-GCCcore-10.2.0/lib/ucx/libuct_ib.so.0",
        "/sw/eb/sw/UCX/1.9.0-GCCcore-10.2.0/lib/ucx/libuct_rdmam.so.0",
        "/sw/eb/sw/UCX/1.9.0-GCCcore-10.2.0/lib/ucx/libuct_cma.so.0"
    ]
},
"cmdline": {
    "54582209": [
        "./radixsort",
        "1024",
        "random"
    ]
},
"cluster": {
    "54582209": "c"
},
"algorithm": {
    "54582209": "radix sort"
},
"programming_model": {
    "54582209": "mpi"
},
"data_type": {
    "54582209": "int"
},
"size_of_data_type": {
    "54582209": 4
},
"input_size": {
    "54582209": 1024
},
"input_type": {
    "54582209": "random"
},
"num_procs": {
    "54582209": 2
},
"scalability": {
    "54582209": "strong"
},
"group_num": {
    "54582209": 12
},
"implementation_source": {
    "54582209": "online"
}
}

```

Sample Sort:

```

{
  "cali.caliper.version": {
    "2189585451": "2.11.0"
  },
  "mpi.world.size": {
    "2189585451": 2
  },
  "spot.metrics": {
    "2189585451":
"min#inclusive#sum#time.duration,max#inclusive#sum#time.duration,avg#inclusive#sum
#time.duration,sum#inclusive#sum#time.duration,variance#inclusive#sum#time.duratio
n,min#min#aggregate.slot,min#sum#rc.count,avg#sum#rc.count,max#sum#rc.count,sum#su
m#rc.count,min#scale#sum#time.duration.ns,max#scale#sum#time.duration.ns,avg#scale
#sum#time.duration.ns,sum#scale#sum#time.duration.ns"
  },
  "spot.timeseries.metrics": {
    "2189585451": ""
  },
  "spot.format.version": {
    "2189585451": 2
  },
  "spot.options": {
    "2189585451":
"time.variance,profile.mpi,node.order,region.count,time.exclusive"
  },
  "spot.channels": {
    "2189585451": "regionprofile"
  },
  "cali.channel": {
    "2189585451": "spot"
  },
  "spot:node.order": {
    "2189585451": "true"
  },
  "spot:output": {
    "2189585451": "p2-a1024.cali"
  },
  "spot:profile.mpi": {
    "2189585451": "true"
  },
  "spot:region.count": {
    "2189585451": "true"
  },
  "spot:time.exclusive": {
    "2189585451": "true"
  },
  "spot:time.variance": {
    "2189585451": "true"
  },
}

```

```
"launchdate": {
  "2189585451": 1729139139
},
"libraries": {
  "2189585451": [
    "/scratch/group/csce435-f24/Caliper/caliper/lib64/libcaliper.so.2",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/lib/libmpicxx.so.12",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/lib/release/libmpi.so.12",
    "/lib64/librt.so.1",
    "/lib64/libpthread.so.0",
    "/lib64/libdl.so.2",
    "/sw/eb/sw/GCCcore/8.3.0/lib64/libstdc++.so.6",
    "/lib64/libm.so.6",
    "/sw/eb/sw/GCCcore/8.3.0/lib64/libgcc_s.so.1",
    "/lib64/libc.so.6",
    "/sw/eb/sw/CUDA/12.4.0/extras/CUPTI/lib64/libcupti.so.12",
    "/sw/eb/sw/PAPI/6.0.0-GCCcore-8.3.0/lib/libpapi.so.6.0",
    "/lib64/ld-linux-x86-64.so.2",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/libfabric.so.1",
    "/lib64/libutil.so.1",
    "/sw/eb/sw/PAPI/6.0.0-GCCcore-8.3.0/lib/libpfm.so.4",
    "/lib64/libnuma.so",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libshm-fi.so",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libmlx-fi.so",
    "/lib64/libucp.so.0",
    "/sw/eb/sw/zlib/1.2.11-GCCcore-8.3.0/lib/libz.so.1",
    "/usr/lib64/libuct.so.0",
    "/usr/lib64/libucs.so.0",
    "/usr/lib64/libucm.so.0",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libverbs-fi.so",
    "/lib64/librdmacm.so.1",
    "/lib64/libibverbs.so.1",
    "/lib64/libnl-3.so.200",
    "/lib64/libnl-route-3.so.200",
    "/usr/lib64/libibverbs/libmlx5-rdmav34.so",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libpsmx2-fi.so",
    "/lib64/libpsm2.so.2",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libsockets-fi.so",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/librxm-fi.so",
    "/sw/eb/sw/impi/2019.9.304-iccifort-
2020.4.304/intel64/libfabric/lib/prov/libtcp-fi.so",
    "/usr/lib64/ucx/libuct_ib.so.0",
    "/usr/lib64/ucx/libuct_rdmacm.so.0",
    "/usr/lib64/ucx/libuct_cma.so.0",
    "/usr/lib64/ucx/libuct_knem.so.0",
```

```

        "/usr/lib64/ucx/libuct_xpmem.so.0",
        "/usr/lib64/libxpmem.so.0"
    ]
},
"cmdline": {
    "2189585451": [
        "./samplesort",
        "1024",
        "random"
    ]
},
"cluster": {
    "2189585451": "c"
},
"algorithm": {
    "2189585451": "sample sort"
},
"programming_model": {
    "2189585451": "mpi"
},
"data_type": {
    "2189585451": "int"
},
"size_of_data_type": {
    "2189585451": 4
},
"input_size": {
    "2189585451": 1024
},
"input_type": {
    "2189585451": "random"
},
"num_procs": {
    "2189585451": 2
},
"scalability": {
    "2189585451": "strong"
},
"group_num": {
    "2189585451": 12
},
"implementation_source": {
    "2189585451": "online"
}
}

```

4. Performance evaluation

Include detailed analysis of computation performance, communication performance. Include figures and explanation of your analysis.

4a. Vary the following parameters

For input_size's:

- 2^{16} , 2^{18} , 2^{20} , 2^{22} , 2^{24} , 2^{26} , 2^{28}

For input_type's:

- Sorted, Random, Reverse sorted, 1%perturbed

MPI: num_procs:

- 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024

This should result in $4 \times 7 \times 10 = 280$ Caliper files for your MPI experiments.

4b. Hints for performance analysis

To automate running a set of experiments, parameterize your program.


- input_type: "Sorted" could generate a sorted input to pass into your algorithms
- algorithm: You can have a switch statement that calls the different algorithms and sets the Adiak variables accordingly
- num_procs: How many MPI ranks you are using


When your program works with these parameters, you can write a shell script that will run a for loop over the parameters above (e.g., on 64 processors, perform runs that invoke algorithm2 for Sorted, ReverseSorted, and Random data).


4c. You should measure the following performance metrics

- Time
 - Min time/rank
 - Max time/rank
 - Avg time/rank
 - Total time
 - Variance time/rank

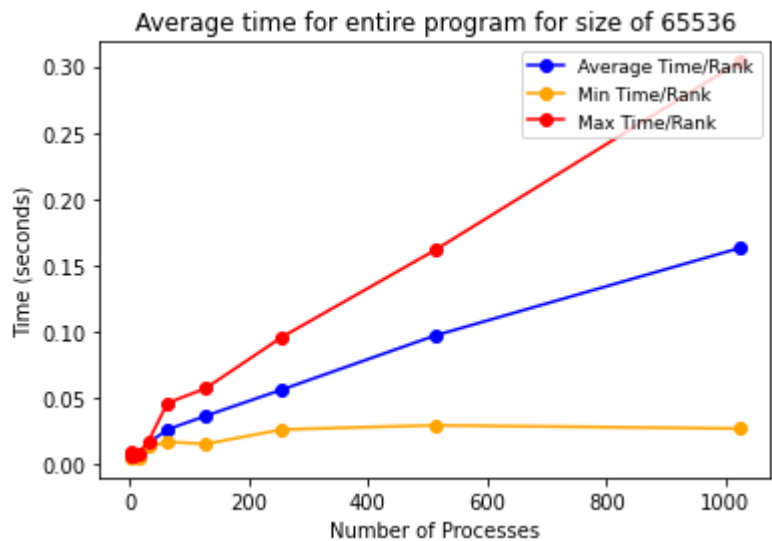
Bitonic Sort

 This is the total time for input size of 2^{16} . Since the input size is not that large, the communication overhead created by parallel processing outweighs the benefits of parallel processing, as shown in the graph by higher average time with larger number of processors.

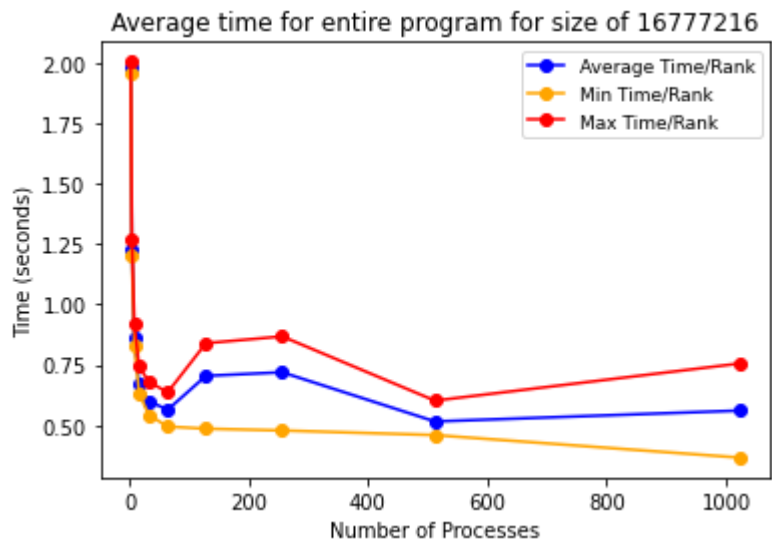
 This is the total time for input size of 2^{20} . This input size is justified for using parallel processing as the graph is becoming to look like an exponential decrease, meaning that the benefits of parallel processing started to outweigh the communication overhead. For this input size, using fewer than 128 processes makes sense.

 This is the total time for input size of 2^{28} . For a large input size like this, parallel processing is justified as the number of processors increases, the average time exponentially decreases since it's computationally more efficient to split tasks to a large number of worker processes.

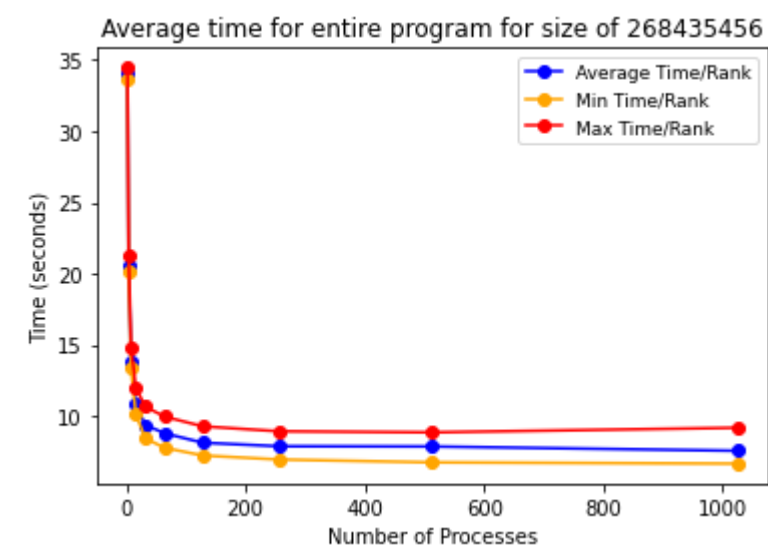
Merge Sort



This is the time taken for input size of 2^{16} . As you can tell, this input size is not nearly large enough to justify using parallel processing. More processes ends up taking longer due to overhead.



This is the time taken for 2^{24} . This is just approaching the optimal input size to justify using parallel processing. There is still too much overhead compared to the granted speedup at 256 and 512 processes. Using fewer or greater processes would make more sense.



This is the time taken for input size of 2^{28} . Merge Sort Appears to be highly scalable, as the time drops significantly as new processes are added. There is a great drop-off followed by a plateau.

Sample Sort



Here is the total time it took for the program for various number of processes for input size of 2^{16} . Based on the graph, it seems like there's a small initial dip, then the overhead needed to allocate the necessary resource becomes more expensive than the computation speed.



Here is the total time it took for the program for various number of processes for input size of 2^{20} . As you can see, it marks a turning point in the trend as there's a much more exponential decrease as the cost of computation becomes more expensive than the cost to allocate resources.



Here's the total time it took for the program for various number of processes for the max input size of 2^{28} . From the graph you can clearly see a strong exponential trend downward which makes the most sense as for such a large input size it's going to be way more computationally expensive to compute rather than allocate resources.

Radix Sort



This graph is the average time of the entire program for input size of 2^{16} . Once again, the input size is relatively small, which results in the commuication overhead of MPI outweighing the benefits that come with parallel computing, resulting in a higher average time for a greater number of processors.



This graph represents an input size of 2^{20} . This array size finally sees some benefits from parallel processing as the graph is neariang an inverse exponential curve. The benefits that come with parallel processing have

begun to overcome the communication overhead of MPI.



This graph represents the total time for an input array size of 2^{28} . The wonders of parallel processing can begin to be seen, as finally, as the number of processors increases, the average time for computation exponentially decreases. The communication overhead becomes negligible to the speedup.

5. Presentation

Plots for the presentation should be as follows:

- For each implementation:
 - For each of comp_large, comm, and main:
 - Strong scaling plots for each input_size with lines for input_type (7 plots - 4 lines each)
 - Strong scaling speedup plot for each input_type (4 plots)
 - Weak scaling plots for each input_type (4 plots)

Analyze these plots and choose a subset to present and explain in your presentation.

6. Final Report

Submit a zip named **TeamX.zip** where **X** is your team number. The zip should contain the following files:

- Algorithms: Directory of source code of your algorithms.
- Data: All **.cali** files used to generate the plots separated by algorithm/implementation.
- Jupyter notebook: The Jupyter notebook(s) used to generate the plots for the report.
- Report.md