

# CSCE 435 Group project

---

0. Group number: 3

1. Group members:

1. Anjali Hole
2. Yahya Syed
3. Kyle Bundick
4. Peter Schlenker
5. Harsh Gangaramani

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

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

- **Bitonic Sort (Peter):** A divide-and-conquer algorithm implemented using MPI that sorts data into many bitonic sequences (the first half only increasing, the second half only decreasing). It then creates alternating increasing and decreasing sequences out of the bitonic sequences to create half as many bitonic sequences, but twice the size. It keeps repeating this process until there is one large bitonic sequence left, at which point it creates one final increasing sequence. For the parallel version I'm implementing, instead of one value each process will keep a sorted list, and when two processes compare lists the smaller sequence will hold a sorted list where all the elements are smaller than the elements in the bigger sequence.
- **Sample Sort (Kyle):** A divide-and-conquer algorithm implemented in MPI that splits the data into buckets based on data samples, sorts the buckets, and then recombines the data.
- **Merge Sort (Anjali):** A parallel divide-and-conquer algorithm implemented using MPI for efficient data distribution and merging where each process independently sorts a portion of the data, and MPI coordinates the merging of subarrays across multiple processors on the Grace cluster.
- **Radix Sort (Yahya):** A divide-and-conquer algorithm implemented with MPI that sorts an array of integers digit by digit, using a counting sort for each digit instead of direct comparisons to determine sorted order. Data distribution is determined by number values, with each process responsible for a certain range of values.
- **Column Sort (Harsh):** A multi-step matrix manipulation algorithm implemented using MPI that sorts a matrix by its columns, redistributes it through a series of transpositions, and applies strategic global row shifts

## Team Communication

- Team will communicate via Discord (for conferencing/meeting)
- Team will use the GitHub repo for reports, and Google Drive to share generated graphs/ report details

What versions do you plan to compare:

Communication strategies:

- a. Point-to-point communication (as shown in the pseudocode)
- b. Collective communication (using MPI\_Allgather or MPI\_Alltoall)

Parallelization strategies:

- a. SPMD (Single Program, Multiple Data) as shown in the pseudocode
- b. Master/Worker model

2b. Pseudocode for each parallel algorithm

- For MPI programs, include MPI calls you will use to coordinate between processes

## Bitonic Sort

```
// Assumes the total list size is a power of 2, and that comm_size is a power of 2 less than or equal to the list
// size, and that local_data size times comm_size is the total list size
function bitonic_sort(local_data, comm_size, rank):
    local_data = sequential_sort(local_data)

    for level = 0 to log2(comm_size) - 1:
        is_increasing = !rank.bit(level + 1)

        for current_bit = level to 0:
            other_rank = rank.flip_bit(current_bit)

            // While the data lives on two processes, only one needs to do the comparison.
            // For now the lower rank process will always do the comparison, though it might speed up the algorithm
            // if we try to balance who does the comparison more evenly.
```

```

        is_doing_comparison = rank < other_rank
        if (is_doing_comparison):
            other_data = MPI_Recv(other_rank)

            (smaller_half, larger_half) = merge(local_data, other_data)

            if (is_increasing):
                local_data = smaller_half
                MPI_Send(larger_half, other_rank)
            else:
                local_data = larger_half
                MPI_Send(smaller_half, other_rank)
        else:
            MPI_Send(local_data, other_rank)
            local_data = MPI_Recv(other_rank)

    return local_data

// Assumes data1 and data2 are the same size
function merge(data1, data2):
    array_size = sizeof(data1)

    lower_half = array size of array_size
    upper_half = array size of array_size

    index1 = index2 = 0

    while (index1 < array_size && index2 < array_size):
        output_index = index1 + index2
        choose_data1 = data1[index1] < data2[index2]

        if (choose_data1):
            value = data1[index1]
            index1++
        else:
            value = data2[index2]
            index2++

        if (output_index < array_size):
            lower_half[output_index] = value
        else:
            upper_half[output_index - array_size] = value

    // by this point we are guaranteed to be filling upper_half, since we have completely gone through one of the
    input arrays
    while (index1 < array_size):
        output_index = index1 + index2
        upper_half[output_index - array_size] = data1[index1]
        index1++

    while (index2 < array_size):
        output_index = index1 + index2
        upper_half[output_index - array_size] = data2[index2]
        index2++

    return (lower_half, upper_half)

function main():
    // Initialize MPI
    MPI_Init()
    comm_size = MPI_Comm_size(MPI_COMM_WORLD)
    rank = MPI_Comm_rank(MPI_COMM_WORLD)

    // Get local data
    local_data = read_or_generate_data(rank, comm_size)

    // Sort
    local_data = bitonic_sort(local_data, comm_size, rank)

    // Verify
    verify_sorted(local_data, comm_size, rank)

    // End program
    MPI_Finalize()

```

**MPI calls to be used:**

```

MPI_Init()
MPI_Comm_size()
MPI_Comm_rank()
MPI_Send()
MPI_Recv()
MPI_Finalize()

```

#### Other functions:

```

sequential_sort(data) - exact algorithm isn't relevant
integer.bit(n) - get the value of the nth bit of the integer as a bool
integer.flip_bit(n) - returns an integer with the same bits, except the nth bit is flipped
read_or_generate_data(rank, comm_size) - data generation function used for each sorting algorithm (to be implemented later)
verify_sorted(local_data, comm_size, rank) - function to verify local data is sorted and that this sequence is smaller than the one stored in the next highest rank (to be implemented later)

```

#### Sample Sort

```

function main(data, data_size, oversample_factor):

    MPI_Init()
    rank = MPI_Comm_rank(MPI_COMM_WORLD)
    size = MPI_Comm_size(MPI_COMM_WORLD)

    for sample = 0 to oversample_factor - 1
        samples.add(data.get_random_element())

    MPI_Gather(source = samples, count = oversample_factor, dest = oversample, root = MASTER)
    if (rank == MASTER):
        sort(oversample)
        splitters[0] = -inf
        for sample = 1 to size - 1:
            splitters[sample] = oversample[sample * oversample_factor]
        splitters[size] = inf
    MPI_Bcast(splitters)

    for each in data:
        choose bucket | splitters[bucket] < bucket && splitters[bucket + 1] > bucket

    for process = 0 to size - 1:
        if process == rank:
            for process = 0 to size - 1:
                Recv(new_data.end, process)
            Send(buckets[process], process)

    local_data = new_data
    sort(local_data)

    MPI_Finalize()

```

#### MPI calls to be used:

```

MPI_Init()
MPI_Comm_size()
MPI_Comm_rank()
MPI_Gather()
MPI_Bcast()
MPI_Send()
MPI_Recv()
MPI_Finalize()

```

#### Merge Sort

```

function parallel_merge_sort(local_data, comm_size, rank):

    // Sort local data using sequential merge sort
    local_data = sequential_merge_sort(local_data)

    // Parallel merge phase
    for step = 1 to log2(comm_size):
        partner = rank XOR (1 << (step - 1)) // Find the partner process
        if rank < partner:
            // Send local data to the partner and receive its data
            MPI_Send(local_data, partner)
            received_data = MPI_Recv(partner)
            // Merge local and received data
            local_data = merge(local_data, received_data)
        else:
            // Send local data to the partner and receive its data
            MPI_Send(local_data, partner)
            received_data = MPI_Recv(partner)
            // Merge received data first to maintain order
            local_data = merge(received_data, local_data)

    return local_data

function main():

    // Initialize MPI
    MPI_Init()
    comm_size = MPI_Comm_size(MPI_COMM_WORLD) // Get number of processes
    rank = MPI_Comm_rank(MPI_COMM_WORLD)      // Get process rank

    // Read or generate local data (each process generates or receives its own data)
    local_data = read_or_generate_data(rank, comm_size)

    // Perform parallel merge sort
    sorted_local_data = parallel_merge_sort(local_data, comm_size, rank)

    // Gather all sorted data at root process
    if rank == 0:
        global_sorted_data = MPI_Gather(sorted_local_data, root=0)
    else:
        MPI_Gather(sorted_local_data, root=0)

    // Finalize MPI
    MPI_Finalize()

```

#### MPI calls to be used:

```

MPI_Init()
MPI_Comm_size()
MPI_Comm_rank()
MPI_Send()
MPI_Recv()
MPI_Gather()
MPI_Finalize()

```

#### Radix Sort

```

// Function to do simple counting sort by the digit place specified by exp
function counting_sort(int arr, int n, int exp):
    output is array size n
    count is array of size 10

    for i from 0 to n:
        count[(arr[i] / exp) % 10]++

    for i from 1 to 10:
        count[i] += count[i - 1];

    for i from n-1 to 0:
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;

    for i from 0 to n:

```

```

arr[i] = output[i];

function radix_sort(local_data, local_size, comm_size, rank)
// Transfer data between processes such that each process has a correct range of values
local_max = max value of local_data
local_min = min value of local_data

// get global max and min values to determine split of numbers
int global_max, global_min;
MPI_Allreduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
MPI_Allreduce(&local_min, &global_min, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);

send_counts, send_offsets, recv_counts, recv_offsets are arrays of size comm_size

// calculate the range of values that each process will receive and send
range_size = (global_max - global_min + 1) / comm_size;

// vector to determine which data gets sent to which process
vector<vector<int>> buckets(comm_size);
for i from 0 to local_size:
    value = local_data[i];
    target_process = (value - global_min) / range_size;
    if target_process >= comm_size:
        target_process = comm_size - 1;
    buckets[target_proc].push_back(value);

// send the data to all processes
total_send = 0;
for i from 0 to comm_size:
    send_counts[i] = buckets[i].size();
    send_offsets[i] = total_send;
    total_send += send_counts[i];

send_data is array of size total_send
index = 0;
for i from 0 to comm_size:
    for j from 0 to buckets[i].size()
        send_data[index++] = buckets[i][j];

MPI_Alltoall(send_counts, 1, MPI_INT, recv_counts, 1, MPI_INT, MPI_COMM_WORLD);

// receive data from all processes
total_recv = 0;
for i from 0 to comm_size:
    recv_offsets[i] = total_recv;
    total_recv += recv_counts[i];

recv_data is array of size total_recv
MPI_Alltoallv(send_data, send_counts, send_offsets, MPI_INT, recv_data, recv_counts, recv_offsets, MPI_INT,
MPI_COMM_WORLD);

// copy received data to local data
local_size = total_recv;
copy(recv_data, recv_data + total_recv, local_data);

// radix sort now that all data are in correct processes
local_max is max element from local data
for exp from 1 to local_max / exp > 0, multiplying by 10:
    counting_sort(local_data, total_recv, exp);

function main():
// initialize MPI
MPI_Init()
rank = MPI_Comm_rank()
size = MPI_Comm_size()

// provide input and sort
input is array to sort
input_size is input size
radix_sort(input, input_size, rank, size)

// finalize MPI
MPI_Finalize()

```

MPI calls to be used:

```

MPI_Init()
MPI_Comm_rank()
MPI_Comm_size()
MPI_Finalize()
MPI_Allreduce()
MPI_Alltoall()
MPI_Alltoallv()

```

## Column Sort

```

// Function to extract a column from a 2D matrix
function get_column_data(local_data, col_index):
    // Initialize an empty list to store the column data
    column_data = []

    // Loop over each row in the local data
    for row in local_data:
        // Append the element at the column index to the column_data list
        column_data.append(row[col_index])

    return column_data

function parallel_column_sort(local_data, num_rows, num_cols, comm_size, rank):
    // Sort each column locally
    for col = 0 to num_cols - 1:
        local_column_data = get_column_data(local_data, col)
        sorted_column_data = sequential_sort(local_column_data) // can use any efficient sequential search such as
merge, bubble
        update_column(local_data, col, sorted_column_data)

    // Transpose the matrix
    local_data = transpose_local(local_data)

    // Perform all-to-all communication to redistribute columns as rows
    new_rows = MPI_Alltoallv(send_data=local_data, send_counts=calculate_send_counts(rank, comm_size),
        recv_data=new_matrix_space, recv_counts=calculate_recv_counts(rank, comm_size))

    // Sort all new rows received
    for row = 0 to num_rows - 1:
        sorted_row = sequential_sort(new_rows[row])
        new_rows[row] = sorted_row

    // Transpose the matrix back
    local_data = transpose_local(new_rows)

    // Another all-to-all communication to redistribute original rows
    final_matrix = MPI_Alltoallv(send_data=local_data, send_counts=calculate_send_counts(rank, comm_size),
        recv_data=final_matrix_space, recv_counts=calculate_recv_counts(rank, comm_size))

    // Final local sort of each column again
    for col = 0 to num_cols - 1:
        local_column_data = get_column_data(final_matrix, col)
        sorted_column_data = sequential_sort(local_column_data)
        update_column(final_matrix, col, sorted_column_data)

    return final_matrix

function main():

    // Initialize MPI
    MPI_Init()
    comm_size = MPI_Comm_size(MPI_COMM_WORLD) // Get number of processes
    rank = MPI_Comm_rank(MPI_COMM_WORLD) // Get process rank

    // Setup matrix dimensions and generate local data
    num_rows, num_cols = determine_dimensions(comm_size)
    local_data = read_or_generate_data(num_rows, num_cols, rank)

    // Perform parallel column sort
    sorted_matrix = parallel_column_sort(local_data, num_rows, num_cols, comm_size, rank)

    // Gather all sorted matrices at root process
    if rank == 0:
        global_sorted_matrix = MPI_Gather(sorted_matrix, root=0)
    else:
        MPI_Gather(sorted_matrix, root=0)

```

```
// Finalize MPI
MPI_Finalize()
```

#### MPI calls to be used

```
MPI_Init()
MPI_Comm_size()
MPI_Comm_rank()
MPI_Alltoallv() // Used for transposing the matrix
MPI_Gather()
MPI_Finalize()
```

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

#### Input:

- Input Sizes
  - $2^{16}$
  - $2^{18}$
  - $2^{20}$
  - $2^{22}$
  - $2^{24}$
  - $2^{26}$
  - $2^{28}$
- Input Types:
  - Sorted
  - Sorted with 1% perturbed
  - Random
  - Reverse sorted

#### Strong scaling (same problem size, increase number of processors/nodes)

- Fix problem size at  $2^{24}$  elements
- Increase number of processors: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
- Measure and compare:
  - Total execution time
  - Speedup ( $T_1 / T_n$ )
  - Parallel efficiency ( $(T_1 / T_n) / n$ )

#### Weak scaling (increase problem size, increase number of processors)

- Start with  $2^{16}$  elements per processor
- Increase both problem size and number of processors proportionally
  - (e.g., 2 processors:  $2 \times 2^{16}$ , 4 processors:  $4 \times 2^{16}$ , etc.)
- Measure and compare:
  - Execution time
  - Parallel efficiency

#### Performance Metrics (to be measured for all experiments):

- Total execution time
- Communication time
- Computation time
- Memory usage

### 3a. Caliper instrumentation

#### Bitonic Sort Calltree

```
0.002 main
├─ 0.000 data_init_runtime
│   └─ 0.000 data_perturbed_init_runtime
│       └─ 0.000 data_init_runtime
├─ 0.001 MPI_Recv
├─ 0.000 MPI_Send
├─ 0.000 correctness_check
│   └─ 0.000 comm
│       └─ 0.000 comm_small
```

```

    | 0.000 MPI_Recv
    | 0.000 MPI_Send
0.000 MPI_Finalize
0.000 MPI_Initialized
0.000 MPI_Finalized
0.000 MPI_Comm_dup

```

### Sample Sort Calltree

```

1.557 MPI_Comm_dup
0.000 MPI_Finalize
0.000 MPI_Finalized
0.000 MPI_Initialized
2.743 main
| 2.141 comp
| | 0.333 comm
| | | 0.318 comm_large
| | | | 0.272 MPI_Recv
| | | | 0.022 MPI_Send
| | | 0.015 comm_small
| | | | 0.014 MPI_Bcast
| | | | 0.001 MPI_Gather
| | 1.799 comp
| | | 1.799 comp_large
| | | 0.000 comp_small
| 0.510 correctness_check
| | 0.499 comm
| | | 0.498 comm_small
| | | | 0.515 MPI_Recv
| | | | 0.000 MPI_Send
| 0.091 data_init_runtime
| | 0.091 data_init_runtime

```

### Merge Sort Calltree

```

0.910 main
| 0.023 data_init_runtime
| | 0.023 data_perturbed_init_runtime
| | | 0.017 data_init_runtime
| 0.591 comp
| | 0.444 comp_large
| | 0.146 comp_small
| 0.059 comm
| | 0.013 comm_small
| | | 0.013 MPI_Sendrecv
| | 0.046 comm_large
| | | 0.046 MPI_Sendrecv
| 0.011 correctness_check
| | 0.001 comm
| | | 0.001 comm_small
| | | | 0.000 MPI_Recv
| | | | 0.000 MPI_Send
0.000 MPI_Finalize
0.000 MPI_Initialized
0.000 MPI_Finalized
0.003 MPI_Comm_dup

```

### Radix Sort Calltree

```

1.321 main
| 0.091 data_init_runtime
| | 0.091 data_init_runtime
| 0.049 comm
| | 0.008 comm_small
| | | 0.008 MPI_Allreduce
| | 0.040 comm_large
| | | 0.003 MPI_Alltoall
| | | 0.037 MPI_Alltoallv
| 0.910 comp
| | 0.897 comp_large
| 0.021 correctness_check
| | 0.011 comm

```



```
└─ 0.011 comm_small
  └─ 0.011 MPI_Recv
    └─ 0.000 MPI_Send
0.000 MPI_Finalize
0.000 MPI_Initialized
0.000 MPI_Finalized
0.131 MPI_Comm_dup
```

Column Sort Calltree

```
3.962 main
├─ 0.017 data_init_runtime
│   └─ 0.017 data_init_runtime
├─ 2.367 comp
│   └─ 2.367 comp_large
├─ 0.716 comm
│   └─ 0.716 comm_large
│       └─ 0.716 MPI_Alltoall
├─ 0.013 correctness_check
│   └─ 0.002 comm
│       └─ 0.002 comm_small
│           └─ 0.002 MPI_Recv
│               └─ 0.000 MPI_Send
0.000 MPI_Finalize
0.000 MPI_Initialized
0.000 MPI_Finalized
0.003 MPI_Comm_dup
```

3b. Collect Metadata

Bitonic Sort Metadata

Key	Value
cali.caliper.version	2.11.0
mpi.world.size	8
spot.metrics	min#inclusive#sum#time.duration,max#inclusive#sum#time.duration,avg#inclusive#sum#time.duration,sum#inclusive#sum#time.duration,v
spot.timeseries.metrics	
spot.format.version	2
spot.options	time.variance,profile.mpi,node.order,region.count,time.exclusive
spot.channels	regionprofile
cali.channel	spot
spot.node.order	true
spot.output	p8-a2048.cali
spot.profile.mpi	true
spot.region.count	true
spot.time.exclusive	true
spot.time.variance	true
launchdate	1729130249
libraries	[/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/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/prov/libmlx-fi.so, /lib64/libucp.so.0, /sw/eb/sw/zlib/1.2.11-GCCcore-8.3.0/li, /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/libps, /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/uc
cmdline	[./main, 0, 1, 2048]
cluster	c
algorithm	bitonic
programming_model	mpi
data_type	int
size_of_data_type	4

Key	Value
input_size	2048
input_type	1_perc_perturbed
num_procs	8
scalability	strong
group_num	3
implementation_source	handwritten

Sample Sort Metadata

Metadata Key	Value
cali.caliper.version	2.11.0
mpi.world.size	32
spot.metrics	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	time.variance, profile.mpi, node.order, region.count, time.exclusive
spot.format.version	2
spot.options	regionprofile
spot.channels	spot
cali.channel	true
spot.node.order	p32-a4194304.cali
spot.output	true
spot.profile.mpi	true
spot.region.count	true
spot.time.exclusive	true
spot.time.variance	true
launchdate	1729120737
libraries	/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-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/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	[./main, 1, 2, 4194304]
cluster	c
algorithm	sample
programming_model	mpi
data_type	int
size_of_data_type	4
input_size	4194304
input_type	Random
num_procs	32

Metadata Key	Value
scalability	strong
group_num	3
implementation_source	handwritten

Merge Sort Metadata

Metadata Key	Value
cali.caliper.version	2.11.0
mpi.world.size	32
spot.metrics	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	time.variance, profile.mpi, node.order, region.count, time.exclusive
spot.format.version	2
spot.options	regionprofile
spot.channels	spot
cali.channel	true
spot.node.order	p32-a4194304.cali
spot.output	true
spot.profile.mpi	true
spot.region.count	true
spot.time.exclusive	true
spot.time.variance	true
launchdate	1729116474
libraries	/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/libc.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/libbuct_ib.so.0, /usr/lib64/ucx/libbuct_rdmacm.so.0, /usr/lib64/ucx/libbuct_cma.so.0, /usr/lib64/ucx/libbuct_knem.so.0, /usr/lib64/ucx/libbuct_xpmem.so.0, /usr/lib64/libxpmem.so.0
cmdline	./main, 2, 1, 4194304
cluster	c
algorithm	merge
programming_model	mpi
data_type	int
size_of_data_type	4
input_size	4194304
input_type	1_perc_perturbed
num_procs	32
scalability	strong
group_num	3
implementation_source	handwritten

Radix Sort Metadata

Metadata Key	Value
cali.caliper.version	2.11.0
mpi.world.size	32
spot.metrics	min#inclusive#sum#time.duration,max#inclusive#sum#time.duration,avg#inclusive#sum#time.duration,sum#inclusive#sum#time.duration,v
spot.timeseries.metrics	
spot.format.version	2
spot.options	time.variance,profile.mpi,node.order,region.count,time.exclusive
spot.channels	regionprofile
cali.channel	spot
spot:node.order	true
spot:output	p32-a4194304.cali
spot:profile.mpi	true
spot:region.count	true
spot:time.exclusive	true
spot:time.variance	true
launchdate	1729119981
libraries	[/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/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/prov/libmlx-fi.so, /lib64/libucp.so.0, /sw/eb/sw/zlib/1.2.11-GCCcore-8.3.0/li /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/libps /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/uc
cmdline	[./main, 3, 2, 4194304]
cluster	c
algorithm	radix
programming_model	mpi
data_type	int
size_of_data_type	4
input_size	4194304
input_type	Random
num_procs	32
scalability	strong
group_num	3
implementation_source	handwritten

Column Sort Metadata

Key	Value
cali.caliper.version	2.11.0
mpi.world.size	32
spot.metrics	min#inclusive#sum#time.duration,max#inclusive#sum#time.duration,avg#inclusive#sum#time.duration,sum#inclusive#sum#time.duration,v
spot.timeseries.metrics	
spot.format.version	2
spot.options	time.variance,profile.mpi,node.order,region.count,time.exclusive
spot.channels	regionprofile
cali.channel	spot
spot:node.order	true
spot:output	p32-a4194304.cali
spot:profile.mpi	true

Key	Value
spot:region.count	true
spot:time.exclusive	true
spot:time.variance	true
launchdate	1729133553
libraries	[/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/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/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/libibverbs/libmlx5-rdmav34.so, /sw/eb/sw/impi/2019.9.304-iccifort-2020.4.304/intel64/libfabric/lib/prov/libpsmx2-fi.so, /lib64/libps /usr/lib64/ucx/libbuct_ib.so.0, /usr/lib64/ucx/libbuct_rdmacm.so.0, /usr/lib64/ucx/libbuct_cma.so.0, /usr/lib64/ucx/libbuct_knem.so.0, /usr/lib64/uc
cmdline	[./main, 4, 3, 4194304]
cluster	c
algorithm	column
programming_model	mpi
data_type	int
size_of_data_type	4
input_size	4194304
input_type	ReverseSorted
num_procs	32
scalability	strong
group_num	3
implementation_source	handwritten

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 4x7x10=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

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