# CSCE 435 Group project

## 0. Group number: 26

## 1. Group members:

1. Caroline Jia
   - Username: cjia2003
   - Algorithm: Radix Sort
2. Griffin Beaudreau
   - Username: CyberGriffin
   - Algorithm: Column Sort
3. Kaitlyn Griffin
   - Username: kaitlyngrif
   - Algorithm: Sample Sort
4. Samuel Bush
   - Username: SamShrubo
   - Algorithm: Merge Sort
5. Zhongyou Wu
   - Username: ZhongyouWuTAMU
   - Algorithm: Bitonic Sort

### 1a. Team Communication:

We will be using Discord for our team communications.

## 2. Project topic: Parallel Sorting Algorithms

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

- Bitonic Sort:

- Sample Sort: Will be implemented using MPI on the Grace cluster. The initial large problem array will split into multiple sub-arrays to be distributed across Grace's nodes and processors.

- Merge Sort: Implement using MPI on the Grace cluster, split the initial array into multiple sub-arrays to distribute across the network of nodes and processors

- Radix Sort: Impleneted using MPI on Grace cluster. The inital array will be split into multiple smaller arrays across the nodes and processors, will be using least significant digit version

- Column Sort: A parallel sorting algorithm that is well suited for sorting data arranged in a 2D grid. The matrix is sorted column-wise, transposed, and sorted again row-wise. This process is repeated until the matrix is sorted. This algorithm will be implemented using MPI on the Grace cluster.

### 2b. Pseudocode for each parallel algorithm

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

---Merge Sort Pseudocode---

```c++
# Full sorting algorithm
def MergeSort(Array, arraySize) {
  MPI_Init(arguments to set up mpi)
  MPI_Comm_rank(MPI_COMM_WORLD, rank)       # set rank to this process rank
  MPI_Comm_size(MPI_COMM_WORLD, numProcs)    # get number of processes

  # divide array into local sub-arrays for each process to sort individually
  if arraySize % numProcs != 0:
    # round up for array size and any unfilled space in the sub-array can be accounted for as null
    localArraySize = ceiling(arraySize / numProcs)
  else:
    localArraySize = arraySize / numProcs
  # make local array
  arrayOffset = rank * localArraySize
  localArray = array[localArraySize]    # create buffer for receiving scattered data


  # scatter to all processes
  MPI_Scatter(Array, localArraySize, MPI_INT, localArray, localArraySize, MPI_INT, root=0,
MPI_COMM_WORLD)


  # each process sorts locally using quicksort
  localQuickSort(localArray, localArraySize)


  # begin merging with neighbor processes
  step = 1
  while step < num_procs:
    # combine even and odd processes in the even process
    if (rank % (2 * step) == 0):
      if (rank + step < num_procs):
        # get sorted array from neighbor process
        receivedSize = localArraySize * step
        receivedArray = new array[receivedSize]


        MPI_Recv(receivedArray, receivedSize, MPI_INT, rank + step, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE)


        # Merge local array and received array using helper function defined below
        localArray = Merge(localArray, localArraySize, receivedArray, receivedSize)
        # Update local size after merge
        localArraySize = localArraySize + receivedSize
    elif (rank % step == 0):
      # Send local array to neighboring process
      MPI_Send(localArray, localArraySize, MPI_INT, rank - step, 0, MPI_COMM_WORLD)

    # double step size each iteration
    step = step * 2

  # only main process gets final sorted array
  if rank == 0
    sortedArray = array[arraySize]
  else
    sortedArray = null

  # from all scattered processes gather into final sorted array
  MPI_Gather(localArray, localArraySize, MPI_INT, sortedArray, localArraySize, MPI_INT, root=0,
MPI_COMM_WORLD)

  # only main can display / output the sorted array
  if rank == 0
    Display(sortedArray)

  # call in every process
  MPI_Finalize()
}

# helper function merges 2 already sorted arrays into 1
def Merge(Array1, array1Size, Array2, array2Size) {
  mergedArray = array[array1Size + array2Size]
  i = 0
  j = 0
  k = 0

  # compare each element until completed 1 array
  while (i < array1Size) and (j < array2Size)
    if Array1[i] < Array2[j]
      mergedArray[k] = Array1[i]
      i++
    else
      mergedArray[k] = Array2[j]
      j++
    k++

  # if unread elements in Array1 copy them to the end of mergedArray
  while i < array1Size
    mergedArray[k] = Array1[i]
    i++
    k++

  # if unread elements in Array2 copy them to the end of mergedArray
  while j < array2Size
    mergedArray[k] = Array2[j]
    j++
    k++

  return mergedArray
}
```

---Sample Sort Pseudocode---

```
# sample sort pseudocode here

```

---Radix Sort Pseudocode---

```
MPI_Init()
MPI_Comm_rank(comm, rank);
MPI_Comm_size(comm, size);

// Scatter the array across processes
if (rank == 0) {
    int *arr = generate_input_array(N);
    // Scatter the array to all processes
    MPI_Scatter(variables);
} else {
    // Other processes
    MPI_Scatter(variables);
}

// Perform radix sort on the local portion of the array

while i < max_digits {
    // Perform counting sort at current digit
    int local_count = counting_sort_by_digit(local_arr, local_size, digit_pos, base);

    // Gather global counts from other processes
    MPI_Allgather(local_count, base, MPI_INT, global_count, base, MPI_INT, comm);

    // Compute prefix sums on global counts to determine offsets
    int prefix_sum = compute_prefix_sums(global_count, base);

    // Redistribute elements based on the computed prefix sums
    int sorted_local_arr = redistribute_elements(local_arr, local_size, digit_pos, prefix_sum, base);

    // Replace local array with the newly sorted portion
    local_arr = sorted_local_arr;
}

// Gather the locally sorted arrays back into the root process
if (rank == 0) {
    MPI_Gather(local_arr, local_size, MPI_INT, sorted_arr, local_size, MPI_INT, 0,
MPI_COMM_WORLD);
} else {
    MPI_Gather(local_arr, local_size, MPI_INT, NULL, local_size, MPI_INT, 0, MPI_COMM_WORLD);
}
MPI_Finalize();
```

` ` `

---Column Sort Pseudocode---

` ` `

Steps:

1: Arrange data in a matrix with r rows and c columns, where r is the number of processors and c is the numner of items per process.

2: Sort each column using a sequential sorting algorithm (may change algorithm depending on input size).

```
 3: Transpose the matrix.
 4: Sort each row independently
 5: Sort each column again.
 6: Tranpose the matrix.
 7: Final column sort.
```
```c++
/*
* Include MPI header
* Include Caliper header
* Include any additional headers
* Define Constants (MASTER)
*/

int main(int argc, char *argv[]) {
  CALI_CXX_MARK_FUNCTION;

  if args invalid return 0;

  // Set up MPI environment (needs to include arguments)
  MPI_INIT();
  MPI_Comm_rank()
  MPI_Comm_size()

  // Initialize variables
  int N = atoi(argv[1])
  int P = num processors;
  int rows, N_padded, padding_size;
  // To handle cases where the total number of data elements isn't divisible by the number of
processors
  // Example: N = 10 and P = 4:
  // rows = (10 + 4 - 1) / 4 = 3
  // N_padded = 3 * 4 = 12 (what it will be when padded)
  // padding_size = 12 - 10 = 2 (num of padding elements)
  rows = (N + P - 1) / P;
  N_padded = rows * P;
  padding_size = N_padded - N;

  // Local data for each processor
  int *local_array = (int*)malloc(rows * sizeof(int));

  if (taskid == MASTER) {
    int *global_data = (int*)malloc(N_padded * sizeof(int));

    /* Add actual data to global_data */

    /* Add padded data */
    for (int i = N; i < N_padded; ++i) {
      global_data[i] = INT_MAX;
    }

    for (int p = 0; p < P; ++p) {
      if (p == MASTER) {
        // copy local data to global data
      } else {
        // MPI_Send
      }
    }
    free(global_data)
  } else {
    // MPI_Recv
  }

  /* local column sort, marked with caliper */

  /* transpose, marked with caliper */

  /* lcaol row sort, marked with caliper */

  /* transpose, marked with caliper */

  /* local column sort, marked with caliper */

  if (taskid == MASTER) {
    int *sorted_data = (int*)malloc(N * sizeof(int));
    // copy local_data into sorted_data

    foreach p = 1; p < column,
MPI_Recv(sorted_data[p*rows],rows,MPI_INIT,p,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

    // Verify that array is sorted

    free(sorted_data);
  } else {
    MPI_SNED(local_data, rows, MPI_INT, MASTER, 0, MPI_COMM_WORLD);
  }
  free(local_data);


  mgr.flush();
  MPI_Finalize();


  return 0;
}
```

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

- Evaluating with multiple process counts, the total process count should always be a power of 2 (2^n processors):
  - Processor count: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
- Using Caliper + thicket to calculate the total time taken, time per process, etc, for each algorithm with the same inputs and processor count across each
  - This method can allow us to determine which algorithms are fastest in what input context
- Adjust the following in each evaluation case to test each algorithm:
  - Input sizes, Input types:
    - Input sizes: 2^16, 2^18, 2^20, 2^22, 2^24, 2^26, 2^28
    - Input types: Sorted, Random, Reverse sorted, 1% perturbed
  - Strong scaling (same problem size, increase the number of processors/nodes)
  - Weak scaling (increase problem size, increase the number of processors)