CSCE 435 Group project

0. Group number: 2 (Honors)

1. Group members:

1. Kimberly Chen
2. Spencer Le
3. Suhu Lavu
4. Andrew Mao
5. Jeff Ooi

We will communicate via a messaging group chat.

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

Parallel Sorting Algorithms

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

We will be comparing the following algorithms:

- Bitonic Sort - Kimberly Chen
  - Bitonic Sort is a parallel sorting algorithm well-suited for distributed systems where the number of processors is a power of 2. The idea behind bitonic sorting is to iteratively build "bitonic sequences" (sequences that are first increasing and then decreasing) and then merge them to make the sorted array. The algorithm works as follows:
    1. Locally sorting the processor's assigned portion of the array based on the processor's rank. Processors with even ranks sort in increasing order, while those with odd ranks sort in decreasing order.
    2. Bitonic merging where for each pair of processors, one sends its data to the other, and they compare their values. If the rank of the processor is less than its partner's, it keeps the smaller values; otherwise, it keeps the larger values.
    3. Recursive sorting where processors again sort their local arrays, but now based on the next bit of their rank so that the sorting order alternates between ascending and descending for different processors at each level
    4. Iterative merging where at each depth, the bitonic sequences are merged to form longer sorted sequences, with alternating sorting directions.
    5. Final gathering where the sorted subarrays held by each processor are gathered by the root processor.
  - The algorithm requires p = 2^k processors and for the number of elements to be 2^n. The time complexity of bitonic sort is O(n log^2 n), but since it is distributed among p processors, the parallel time complexity becomes O((n log^2 n) / p).
- Sample Sort - Spencer Le
  - Sample Sort is a highly scalable parallel sorting algorithm that divides the input data among multiple processors and efficiently sorts large datasets using a combination of local sorting, sampling, and redistribution. It is particularly well-suited for distributed-memory systems. The algorithm works as follows:
    1. Local Sorting: Each processor receives an equal-sized portion of the array to sort. The processors independently sort their local portions using a sequential sorting algorithm (e.g., quicksort). This step reduces the problem size locally before redistribution.
    2. Sampling: After the local sort, each processor selects a small sample of its locally sorted data. These samples are used to choose “splitters,” which will define the boundaries between the ranges of values to be assigned to each processor.
    3. Choosing Splitters: The samples from all processors are gathered on the root processor (or another dedicated processor), which sorts them and selects a set of splitters. These splitters divide the global array into buckets, where each bucket represents a range of values that should be sent to a specific processor.
    4. Redistribution (All-to-All Communication): Using the splitters, each processor sends its locally sorted data to the appropriate processor, ensuring that the data in each bucket is sent to the processor responsible for that range of values. This step involves an all-to-all communication to exchange data between processors.
    5. Final Sorting and Gathering: Once each processor has received its portion of data, it performs a final sort on the received data, ensuring that the data within each processor is fully sorted. Finally, the sorted subarrays are gathered by the root processor to form the final globally sorted array.
  - The algorithm is highly scalable because it minimizes the amount of inter-processor communication and focuses on local sorting and efficient data redistribution. The time complexity of Sample Sort is `O(nlog(n)/p)`, where `n` is the number of elements, and `p` is the number of processors
- Merge Sort - Suhu Lavu
  - Merge sort is a parallel sorting algorithm that uses a divide and conquer approach to sort an array by recursively dividing it into subarrays, sorting those subarrays, and then merging them backk together. The algorithm works as follows:
    1. The initial unsorted array is divided into smaller subarrays. Each processor is assigned a portion of this array.
    2. Each processor sorts their portion using a sequential merge sort where the array is divided into subarrays, sorted, and merged back together.
    3. Once the local arrays are sorted, processors merge their sorted subarrays. Merging involves comparing elements between 2 sorted arrays and combining them into a single sorted array.
    4. Sorted arrays from multiple processors are merged together in pairs, doubling the size of the sorted array at each step until there is a single sorted array with all elements.
    5. The root process gathers the final sorted array.
  - Sequential merge sort has a time complexity of `O(nlog(n))`. However, because this is distributed among `p` processors, the time complexity becomes `O(nlog(n)/p)` in parallel.
- Radix Sort - Andrew Mao
  - Radix sort is a non comparative sorting algorithm that sorts an array through evaluating and counting individual digits. We sort elements into buckets with a helper counting sort function. This is repeated multiple times starting from the least significant digit until the most significant digit. In contrast to serial radix sort, we must shuffle elements between processors' local buckets. The algorithm works as follows:
    1. Divide the whole unsorted array into local subarrays to be sorted by p processors.
    2. For each digit, from the least significant to the most significant, of the largest number in the array, run serial counting sort for the local array.
    3. Gather counts from all processes to calcaulte the prefix sums and total sums to find the position of each digit like in counting sort.
    4. Move elements within each processor to appropriate local buckets
    5. Send and receive elements to correct positions in local arrays of other processes.
    6. Gather all sorted local arrays back to master.
  - Sequential run time of decimal radix sort is `O(n * d)` where d is the max number of digits or `log10(maxVal)` and n is the number of elements in the array. An ideal parallel runtime would divide the time by p processors, which is `O(n * d / p)`

- Column Sort - Jeff Ooi
  - Column Sort is an eight step matrix parallel sorting algorithm, with the eight steps of the algorithm being as follows:
    1. Sort each column of the matrix.
    2. Transpose the matrix by reading the elements in Column-Major and writing back to the matrix in Row-Major, keeping the original shape and dimensions of the matrix
    3. Sort each column of the transposed matrix.
    4. Untranspose the matrix by reading the elements in Row-Major and writing back to the matrix in Column-Major, keeping the original shape and dimensions of the matrix.
    5. Sort each column of the untransposed matrix.
    6. Shift the elements of the matrix column-wise down by the floor of rows/2 and fill in the empty spaces in the first column with negative infinity. This will create a rows x (columns + 1) matrix. Fill in the remaining spaces in the final column with positive infinity.
    7. Sort each column of the shifted matrix.
    8. Unshift the elements of the matrix by deleting the infinities and shifting every element column-wise up by the floor of rows/2. This will return the matrix back to the original rows x columns dimensions and complete the sort.
  - The algorithm has the restriction that `rows >= 2 * (columns - 1) ^ 2` and `rows % cols == 0`. The longest step of the algorithm is the sorting of the columns, which runs in `O(nlog(n))`. However, because those steps are distributed among `p` processes, the runtime of the algorithm is `O(nlog(n)/p)` time, where `n` is the total number of elements in the matrix.

We will use the Grace cluster on the TAMU HPRC.

2b. Pseudocode for each parallel algorithm

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

```
func bitonic_sort(matrix, lowIndex, count, direction)
    Initialize MPI
    rank <- MPI rank
    num_procs <- MPI size

    if count > 1
        k = count / 2  // Divide the array into two halves

        // Sort the first half in ascending order
        bitonicSort(matrix, lowIndex, k, 1)

        // Sort the second half in descending order
        bitonicSort(matrix, lowIndex + k, k, 0)

        // Merge the two halves according to the 'direction'
        bitonicMerge(arr[], lowIndex, count, direction)

        for step = 1 to log2(num_procs)
            partner <- rank XOR step

            // Exchange sorted halves with partner process and merge
            if direction == 1 // this means it is ascending
                if rank < partner
                    MPI send local_array to partner
                    MPI receive partner_array from partner
                    matrix <- bitonic_merge_ascending(matrix, partner_array)
                else
                    MPI send local_array to partner
                    MPI receive partner_array from partner
                    matrix <- bitonic_merge_descending(matrix, partner_array)
            else // this means it is descending
                if rank < partner
                    MPI send local_array to partner
                    MPI receive partner_array from partner
                    matrix <- bitonic_merge_descending(matrix, partner_array)
                else
                    MPI send local_array to partner
                    MPI receive partner_array from partner
                    matrix <- bitonic_merge_ascending(matrix, partner_array)
            end if
        end for
    end if

    // Gather all sorted data at master
    if rank is master
        for i: 1 -> num_procs
            MPI receive sorted segments from all processes
        end for
        output "Final sorted array"
    end if

    Finalize MPI
end func
```

- Sample Sort Pseudocode

```
func parallel_sample_sort(local_data):
    // Initialize MPI environment
    MPI_Init()

    // Get total number of processes and rank of each process
    int num_procs, rank
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs)
    MPI_Comm_rank(MPI_COMM_WORLD, &rank)

    // Step 1: Sort local data locally
    local_data = sort(local_data)

    // Step 2: Choose 'p' samples from the sorted local data
    samples = select_samples(local_data, num_procs)

    // Step 3: Gather all samples to the root process
    all_samples = []
    MPI_Gather(samples, num_procs, MPI_INT, all_samples, num_procs, MPI_INT, root=0, MPI_COMM_WORLD)

    // Step 4: Root process sorts the gathered samples and selects splitters
    splitters = []
    if rank == 0:
        all_samples_sorted = sort(all_samples)
        splitters = select_splitters(all_samples_sorted, num_procs)

    // Step 5: Broadcast splitters to all processes
    MPI_Bcast(splitters, num_procs, MPI_INT, root=0, MPI_COMM_WORLD)

    // Step 6: Each process redistributes its local data based on the splitters
    buckets = distribute_data(local_data, splitters, num_procs)

    // Step 7: Each process sends its buckets to the appropriate processes
    for i = 0 to num_procs - 1:
        // Send and receive buckets to/from each process
        send_bucket_to_process(buckets[i], i)
        receive_bucket_from_process(i)

    // Step 8: Merge the received buckets
    local_data = merge(received_buckets)

    // Step 9: Perform local sort on the merged data
    local_data = sort(local_data)

    // Step 10: Gather sorted data on the root process
    sorted_data = []
    MPI_Gather(local_data, len(local_data), MPI_INT, sorted_data, len(local_data), MPI_INT, root=0, MPI_COMM_WORLD)

    // Finalize MPI environment
    MPI_Finalize()

    return sorted_data
end func
```

- Merge Sort Pseudocode

```
func parallel_merge_sort(array, n):
    Initialize MPI
    rank <- MPI rank
    size <- MPI size

    # calculate size of worker array and create local copy
    worker_size = n / size
    worker_arr = new array[worker_size]

    # send real array to all processes
    MPI_Scatter(array, worker_size, MPI_INT, worker_arr, worker_size, MPI_INT, root=0, MPI_COMM_WORLD)

    # sort local copy
    sorted_arr = merge_sort(worker_arr)

    # merge sorted pieces back together
    step = 1
    while step < size:
        if rank % (2 * step) == 0:
            if rank + step < size:
                # get sorted array from another process
```

```
                received_size = worker_size * step
                received_array = new array[received_size]
                MPI_Recv(received_array, received_size, MPI_INT, source=rank + step, tag=0, MPI_COMM_WORLD)

                # merge received array with local array
                sorted_arr = merge(sorted_arr, received_array)
                worker_size += received_size
        else:
            # send sorted array to another process
            target = rank - step
            MPI_Send(sorted_arr, worker_size, MPI_INT, dest=target, tag=0, MPI_COMM_WORLD)
            break out of loop
        step *= 2

    # master process gathers the final sorted array
    if rank == 0:
        MPI_Gather(sorted_arr, worker_size, MPI_INT, array, worker_size, MPI_INT, root=0, MPI_COMM_WORLD)

    Finalize MPI
end func

func merge(arr1, arr2):
    size_1 = size(arr1)
    size_2 = size(arr2)
    final = new array[size_1 + size_2]
    i, j, k = 0

    while i < size_1 and j < size_2:
        if arr1[i] < arr2[j]:
            final[k] = arr1[i]
            i += 1
        else:
            final[k] = arr2[j]
            j += 1
        k += 1

    # copy leftover elements
    while i < size_1:
        final[k] = arr1[i]
        i += 1
        k += 1

    while j < size_2:
        final[k] = arr2[j]
        j += 1
        k += 1

    return final
end func

func merge_sort(arr):
    # base case
    if size(arr) <= 1:
        return arr

    # find midpoint
    mid = size(arr) / 2

    # sort left array
    left_sorted = merge_sort(arr[0:mid])

    # sort right array
    right_sorted = merge_sort(arr[mid + 1:])

    # merge them together
    return merge(left_sorted, right_sorted)
end func
```

- Radix Sort Pseudocode

```
func radix_sort(matrix, lowIndex, count, direction)
    Initialize MPI
    rank <- MPI rank
    num_procs <- MPI size

        keysPerWorker <- numKeys / num_procs

        if rank is master:
            for i: 1 -> num_procs - 1:
                MPI send arr[i * keysPerWorker to (i + 1) * keysPerWorker - 1] to process i
            end for
        end if

        localbuffer <- [keysPerWorker]

        if rank is worker:
            MPI receive arr[i * keysPerWorker to (i + 1) * keysPerWorker - 1] into localbuffer
        end if

        # For each iteration, process g bits at a time (Radix Sorting)
        max_bits <- get bits max(arr)
        for i: 0 -> (max_bits / g) do:
            local_bucket_counts <- counting_sort(localbuffer, g, i)

            # Aggregate the result of counting sort, make global count of keys per bucket
            global_bucket_counts <- MPI all_to_all(local_bucket_counts)
            prefix_sums <- get prefix sums of global_bucket_counts
            bucketed_keys <- distribute_keys(localbuffer, g, i, prefix_sums)

            # update these results to preserve order
            exchanged_keys <- MPI all_to_all_exchange(bucketed_keys)
            sort_by_g_bits(exchanged_keys, g, i)
            localbuffer <- exchanged_keys
        end for

        if rank is worker:
            MPI send localbuffer to master
        end if

        if rank is master:
            for i: 1 -> num_procs - 1 do:
                MPI receive sorted buffer into arr
            end for

        Finalize MPI
    end func
```

- Column Sort Pseudocode

```
 Assumption: numRows >= 2 * (numCols - 1)^2
func column_sort(matrix, numRows, numCols)
    Initialize MPI
    rank <- MPI rank
    num_procs <- MPI size

    colsPerWorker = numCols / num_procs

    if rank is master
        for i: 1 -> num_procs
            MPI send matrix columns i * colsPerWorker to (i + 1) * colsPerWorker - 1 to process i
        end for

    localbuffer <- buffer with enough size to fit matrix columns i * colsPerWorker to (i + 1) * colsPerWorker - 1
    if rank is not master
        MPI receive matrix columns i * colsPerWorker to (i + 1) * colsPerWorker - 1 into localbuffer

    sort every column in localbuffer

    if rank is not master
        MPI send sorted localbuffer to master
    if rank is master
        for i: 1 -> num_procs
        MPI receive all localbuffers into matrix
        end for

        transpose matrix
```

```
        for i: 1 -> num_procs
            MPI send matrix columns i * colsPerWorker to (i + 1) * colsPerWorker - 1 to process i
        end for

    if rank is not master
        MPI receive matrix columns i * colsPerWorker to (i + 1) * colsPerWorker - 1 into localbuffer

    sort every column in localbuffer

    if rank is not master
        MPI send sorted localbuffer to master
    if rank is master
        for i: 1 -> num_procs
            MPI receive all localbuffers into matrix
        end for

        untranspose matrix

        for i: 1 -> num_procs
            MPI send matrix columns i * colsPerWorker to (i + 1) * colsPerWorker - 1 to process i
        end for

    if rank is not master
        MPI receive matrix columns i * colsPerWorker to (i + 1) * colsPerWorker - 1 into localbuffer

    sort every column in localbuffer

    if rank is not master
        MPI send sorted localbuffer to master
    if rank is master
        for i: 1 -> num_procs
            MPI receive all localbuffers into matrix
        end for

        shift matrix

        for i: 1 -> num_procs
            MPI send matrix columns i * colsPerWorker to (i + 1) * colsPerWorker - 1 to process i
        end for

    if rank is not master
        MPI receive matrix columns i * colsPerWorker to (i + 1) * colsPerWorker - 1 into localbuffer

    sort every column in localbuffer

    if rank is not master
        MPI send sorted localbuffer to master
    if rank is master
        sort last column in matrix

        unshift matrix // matrix should now be sorted in Column-Major order

        for i: 1 -> num_procs
            MPI send matrix columns i * colsPerWorker to (i + 1) * colsPerWorker -  to process i
        end for
    if rank is not master
        MPI receive matrix columns i * colsPerWorker to (i + 1) * colsPerWorker - 1 into localbuffer

    check if localbuffer is sorted
    sorted <- 1 if sorted, 0 if not
    if rank is master
        isSorted <- -1
        MPI reduce sorted to master with minimum sorted into isSorted
        if isSorted is 1
            output "matrix sorted"
        else
            output "matrix not sorted"

    Finalize MPI
end func
```

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

We will measure and compare the sorting times of:

- Different input sizes and input types
  - Input sizes are 2^16, 2^18, 2^20, 2^22, 2^24, 2^26, and 2^28
  - Input types are sorted, random, reverse sorted, and 1% perturbed
- Strong scaling (same problem size, increase number of processors)
  - Number of processes are 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024
- Weak scaling (increase problem size, increase number of processors)
  - Number of processes are 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024

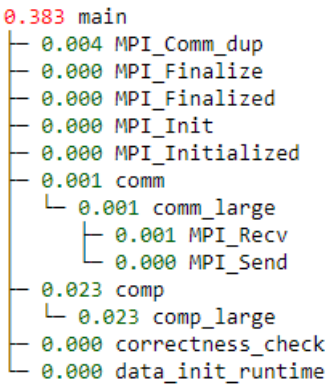
We will collect them using Caliper and compare them using Thicket.

3. Caliper Instrumentation

3a. Calltrees

- Bitonic Sort Calltree

- Sample Sort Calltree
- Merge Sort Calltree



- Radix Sort Calltree

```
  _____ _     _     _       _
 |_   _| |__ (_) ___| | _____| |_
   | | | '_ \| |/ __| |/ / _ \ __|
   | | | | | | | (__|   <  __/ |_
   |_| |_| |_|_|\___|_|\_\___|\__|  v2024.1.0

0.074 MPI_Comm_dup
0.000 MPI_Finalize
0.000 MPI_Finalized
0.000 MPI_Initialized
1.435 main
├─ 0.002 MPI_Barrier
├─ 1.425 comm
│  ├─ 0.000 MPI_Barrier
│  ├─ 0.248 comm_large
│  │  ├─ 0.238 MPI_Allgather
│  │  └─ 0.010 MPI_Gather
│  └─ 1.176 comm_small
│     ├─ 0.004 MPI_Bcast
│     ├─ 0.072 MPI_Isend
│     ├─ 0.981 MPI_Recv
│     └─ 0.000 MPI_Scatter
├─ 0.001 comp
│  ├─ 0.001 comp_large
│  └─ 0.000 comp_small
├─ 0.001 correctness_check
└─ 0.000 data_init_runtime

Legend (Metric: Avg time/rank Min: 0.00 Max: 1.43 indices: {'profile': 68957467})
1.29 - 1.43
1.00 - 1.29
0.72 - 1.00
0.43 - 0.72
0.14 - 0.43
0.00 - 0.14
```

- Column Sort Calltree
  `data_init_X` is `data_init_runtime`

```
0.930 main
├─ 0.004 MPI_Barrier
├─ 0.001 MPI_Comm_dup
├─ 0.000 MPI_Finalize
├─ 0.000 MPI_Finalized
├─ 0.000 MPI_Init
├─ 0.000 MPI_Initialized
├─ 0.013 comm
│  ├─ 0.013 comm_large
│  │  ├─ 0.001 MPI_Isend
│  │  ├─ 0.011 MPI_Recv
│  │  └─ 0.000 MPI_Wait
│  └─ 0.001 comm_small
│     ├─ 0.001 MPI_Recv
│     └─ 0.000 MPI_Send
├─ 0.136 comp
│  ├─ 0.131 comp_large
│  └─ 0.005 comp_small
├─ 0.002 correctness_check
│  ├─ 0.001 MPI_Recv
│  └─ 0.000 MPI_Send
└─ 0.002 data_init_X
```

3b. Metadata

- Bitonic Sort Metadata

programming_model	data_type	size_of_data_type	input_size	input_type	num_procs	scalability	group_num	implementation_source
mpi	int	4	67108864	Random	2	strong	2	online

- Sample Sort Metadata
- Merge Sort Metadata

algorithm	programming_model	data_type	size_of_data_type	input_size	input_type	num_procs	scalability	group_num	implementation_source
merge	mpi	int	4	65536	Sorted	16	strong	2	online

- Radix Sort Metadata

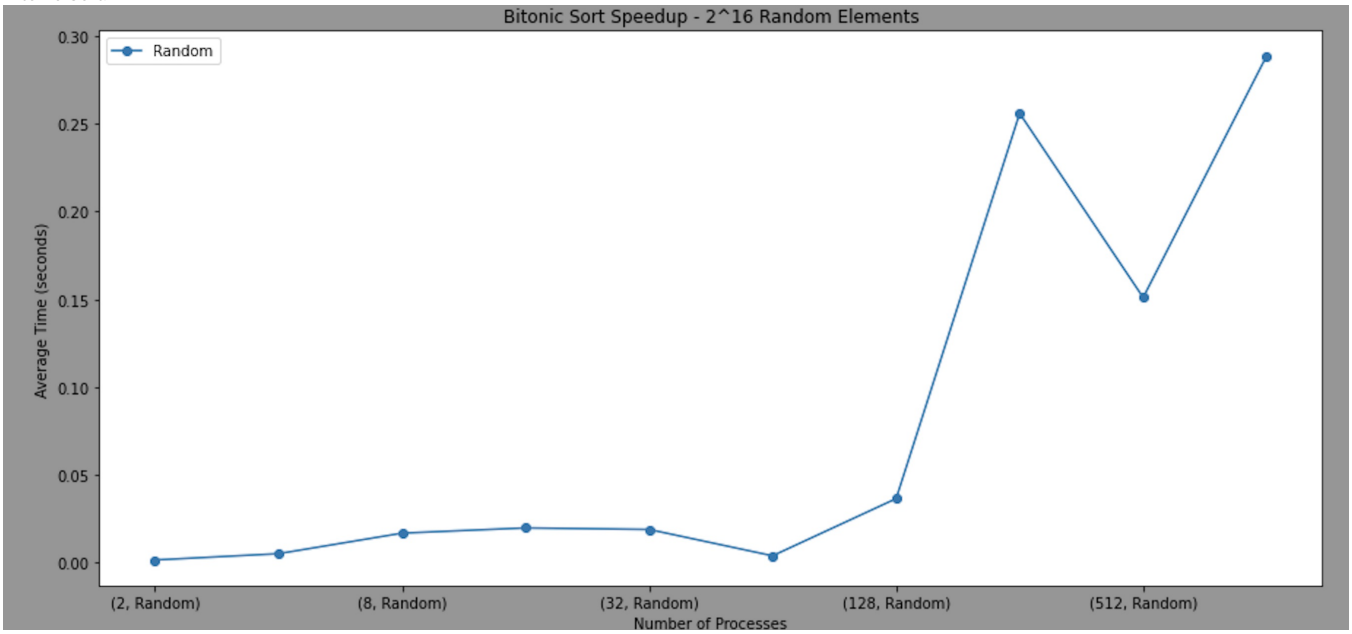
algorithm	programming_model	data_type	size_of_data_type	array_size	array_type	num_procs	scalability	group_num	implementation_source
radix	mpi	int	4	65536	Random	16	strong	2	online

- Column Sort Metadata

algorithm	programming_model	data_type	size_of_data_type	input_size	input_type	num_procs	scalability	group_num	implementation_source
column_sort	mpi	int	4	4194304	1_perc_perturbed	16	strong	2	handwritten

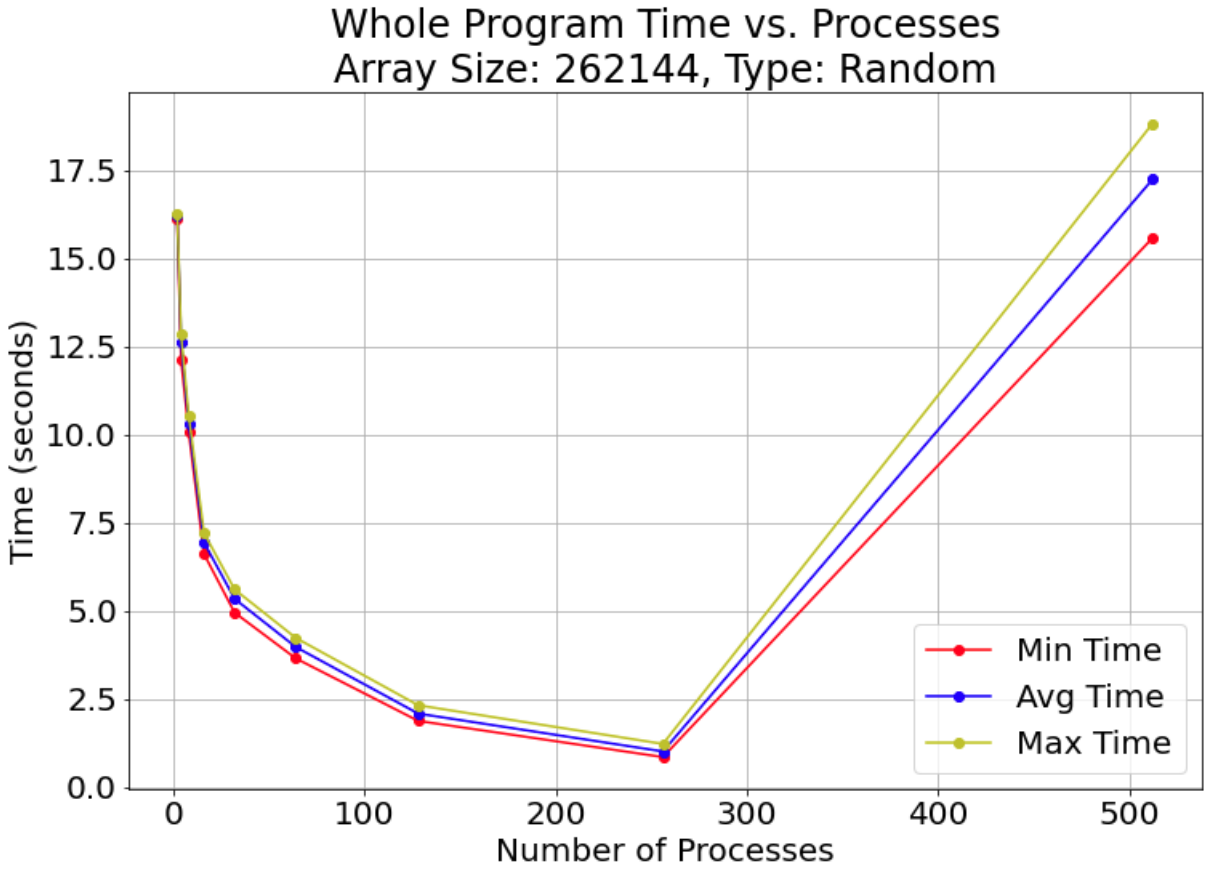
4. Performance Evaluation

- Bitonic Sort:

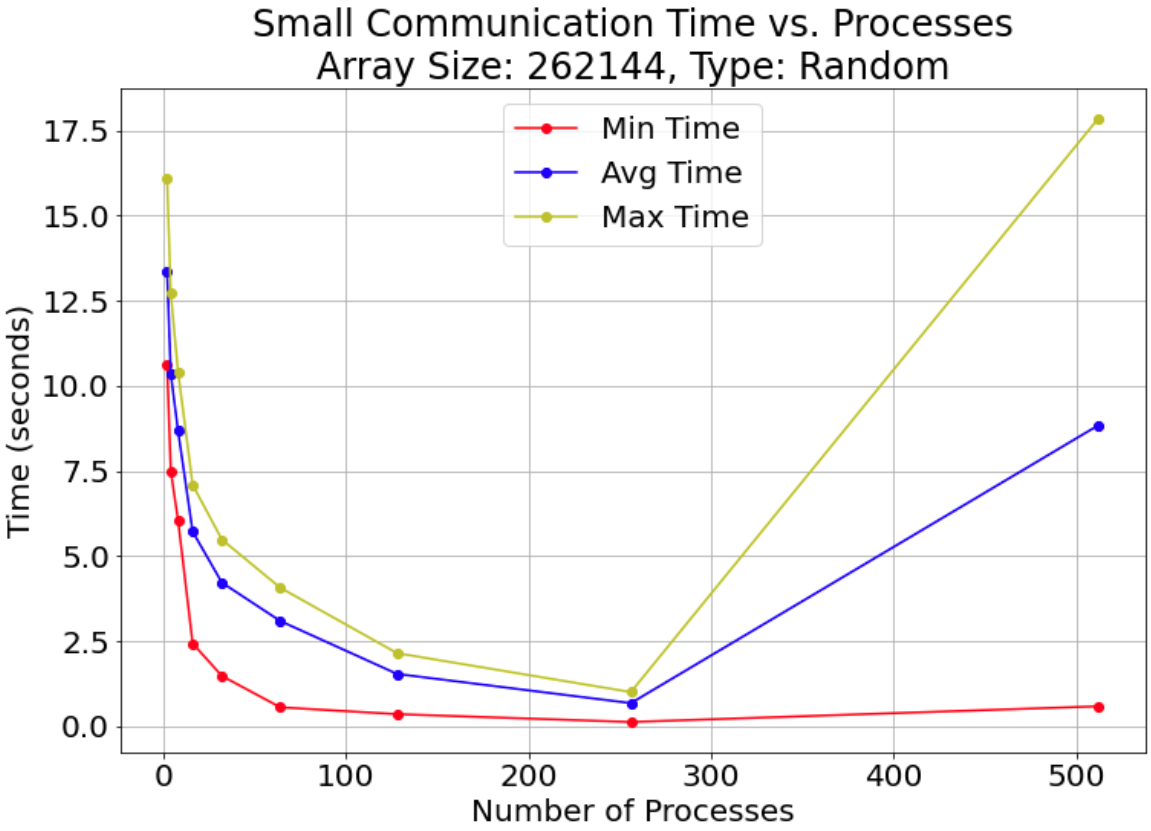


The graph indicates that the speedup maxes out around 512 processes, with the speedup become significantly higher at around 128 processes. We can see from this that using 512 processes would be the most efficient for sorting a randomized array size of 2^16.

- Radix Sort:
  - Limitations with the algorithm: Innefficiencies with the current implementation resulted in limited data for higher number of processors. Because of the nature of radix sort using counting sort under the hood, there is a lot of potential for uneven load balances between processors. (There can be more of one digits and a skewed distribution of buckets)

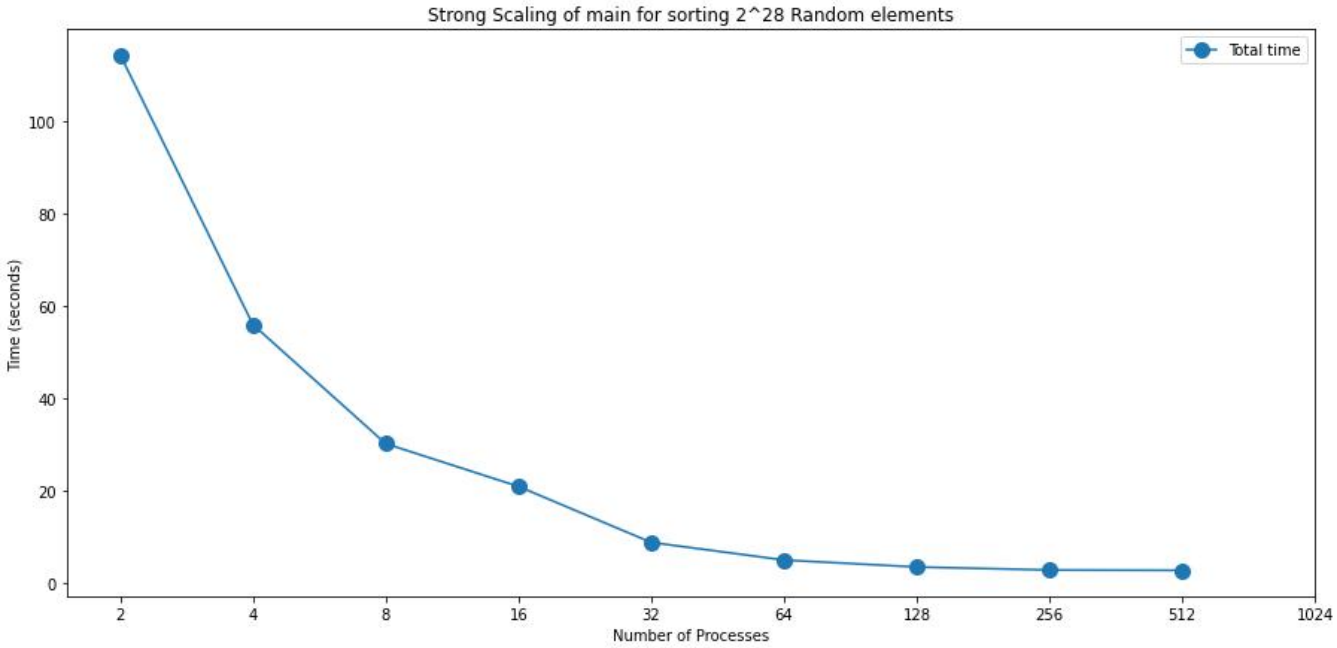


The graph indicates that the current algorithm is strongly scaled to a certain point where we see diminishing returns at 512+ processors. The high runtime can be attributed to the current inefficient communication.

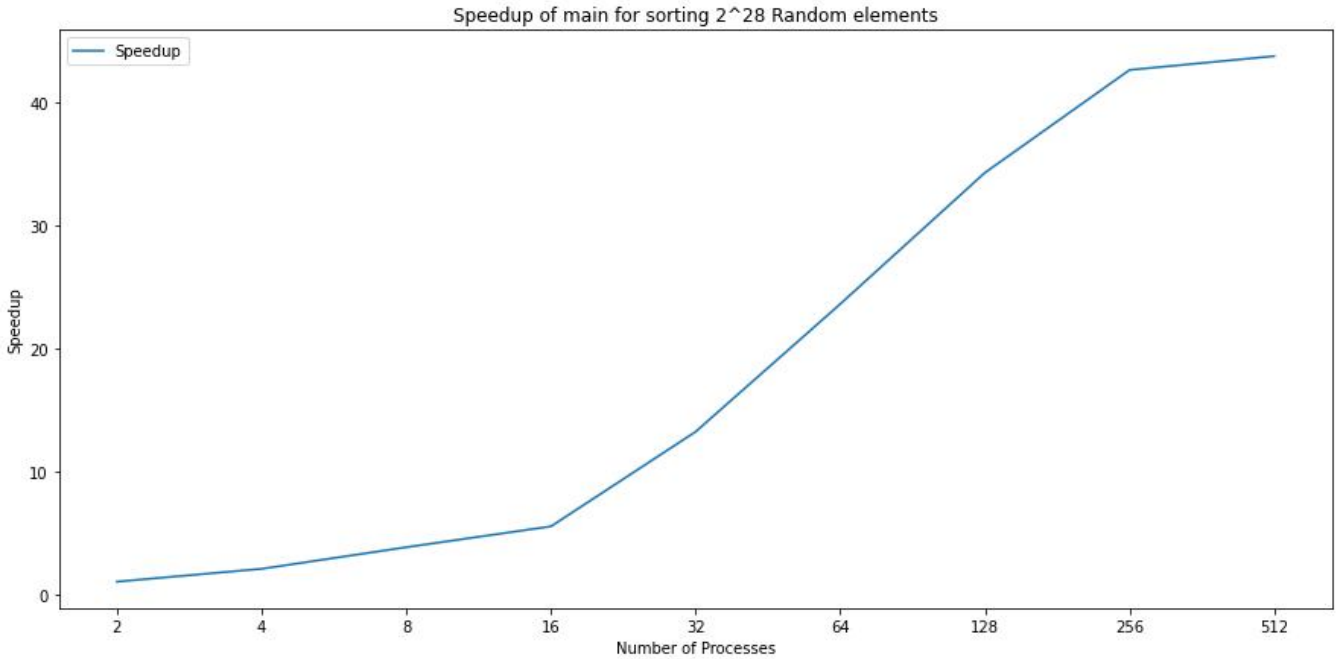


Labeled incorrecly as small

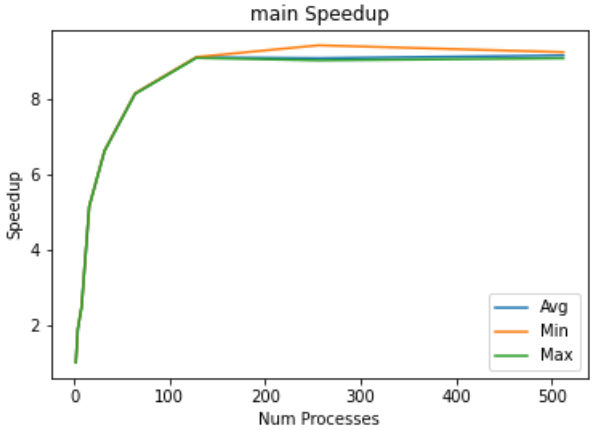
- Column Sort



The graph indicates that the algorithm is strongly scaled, with a bound of about 2 seconds due to the sequential runtime.



The graph indicates that the speedup maxes out at 512 processes, with about a 40x speedup compared to 2 processes. Because 512 is the maximum number of processes that can be used to sort 2^28 elements due to the restrictions, this means that using 512 processes would be the most efficient.
More analysis and plots in the column_sort folder.



- Merge Sort Here is an example of how parallelization can improve performance. The graph shows the speedup of the whole program for an input size of 2^26 and a reverse sorted input. We can see that increasing the number of processes allows for drastic improvements of up to 10x in runtime up to 128 processes, but then results in diminishing returns. More analysis and plots are available in the merge_sort folder.