CSCE 435 Group project

0. Group number: 10

1. Group members:

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

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

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

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

Bitonic Sort

Bitonic sort is a parallel sorting algorithm that works well on distributed systems that support parallel computation. It sorts a sequence of numbers using a series of compare-and-swap operations, following a divide-and-conquer approach. It works efficiently with bitonic sequences, which are sequences that first monotonically increase then monotonically decrease, also vice versa. The time complexity of Bitonic Sort is O(log^2 n) which is efficient for paralle computation compared to other algorithms like Bubble Sort O(n^2).

Sample Sort

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

Merge Sort

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

Radix Sort

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

2b. Pseudocode for each parallel algorithm

Bitonic Sort

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

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

// Split the array based on numprocesses
```

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

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

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

// Finalize MPI
```

Sample Sort

```
Choose Some Constant k to be the Oversampling Ratio (i.e. Number of Elements Sampled From Each Array Segment)


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

    Allocate Array of Size p - 1 as pivots

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

        Sort sample Using Some Comparison-Based Sort

        Set pivots to Contain [sample[k], sample[2k], ..., sample[(p - 1) * k]]
        Send pivots to Other Processes (MPI_Send)

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

    Allocate Array of p Vectors, Each of Size n / p^2 as buckets (Each Row Represents a Bucket, and We Allocate
Initial Memory Under the Assumption of Even Distribution)

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

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

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

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

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

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

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

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

                        Receive curr_size Elements from Process j into arr + my_bucket_size (MPI_Recv)
                        my_bucket_size += curr_size

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

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

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

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


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

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

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

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

        Verify arr is Properly Sorted

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

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

    Finalize MPI (MPI_Finalize)
```

Merge Sort

1. Initialize MPI environment.
2. Divide the array of numbers among available processes.
3. Local Sorting using a sequential Merge Sort
4. Exchange and merge sorted arrays between processes.

5. Gather the sorted subarrays at the root process.
6. Output the sorted array at the root process.
7. Finalize MPI

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

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

// Use MPI Scatter to distribute the array across processes

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

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

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

// MPI Gather sorted arrays on the root processses

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

// Finalize MPI
```

Radix Sort

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

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

 if (rank == 0) {
    // Generate or input the array on the root process
    for (int i = 0; i < totalSize; i++) {
        arr[i] = rand() % 1000;  // Example random values
    }
}
```

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

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

//MPI Gather sorted arrays on the root process

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

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

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

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

Input sizes

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

Input types

In terms of ordering, our input arrays will be of 4 different types: random, sorted, reverse sorted, and sorted with 1% perturbed. This will allow us to observe the strengths and weaknesses of each sorting algorithm and reason about what form input array each is more tailored to solving.

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

Example:

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

Scaling

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

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

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

3. Project implementation

3a. Caliper instrumentation

This section contains calltrees for each algorithm generated using Caliper and Thicket.

Bitonic Sort

The following tree comes from a run with 2^28 elements on 64 processors. The average time per rank is displayed on the tree.

```
10.125 main
├─ 0.145 MPI_Barrier
├─ 0.880 MPI_Comm_dup
├─ 0.000 MPI_Finalize
├─ 0.000 MPI_Finalized
├─ 0.000 MPI_Init
├─ 0.000 MPI_Initialized
├─ 0.110 MPI_Sendrecv
├─ 0.408 comm
│  └─ 0.408 comm_small
│     └─ 0.408 MPI_Scatter
├─ 6.299 comp
│  ├─ 0.155 comp_large
│  │  └─ 0.155 MPI_Gather
│  └─ 6.144 comp_small
├─ 0.769 correctness_check
└─ 0.717 data_init_runtime
```

Sample Sort

The following tree comes from a run with 2^28 randomly-organized elements on 64 processors. The average time per rank is displayed in the tree.

```
34.247 main
├─ 1.406 MPI_Barrier
├─ 0.770 MPI_Comm_dup
├─ 0.000 MPI_Finalize
├─ 0.000 MPI_Finalized
├─ 0.000 MPI_Init
├─ 0.000 MPI_Initialized
├─ 28.302 comm
│  ├─ 28.240 comm_large
│  │  ├─ 0.230 MPI_Recv
│  │  ├─ 26.362 MPI_Scatter
│  │  └─ 1.641 MPI_Send
│  └─ 0.062 comm_small
│     ├─ 0.059 MPI_Bcast
│     └─ 0.003 MPI_Gather
├─ 1.803 comp
│  ├─ 1.802 comp_large
│  └─ 0.000 comp_small
├─ 0.768 correctness_check
└─ 26.654 data_init_runtime
```

Merge Sort

The following tree comes from a run of merge sort with 2^28 random unsigned integer elements on 64 processors. The average time per rank is displayed in the tree.

```
155.835 main
├─ 24.103 MPI_Barrier
├─ 0.795 MPI_Comm_dup
├─ 0.000 MPI_Finalize
├─ 0.000 MPI_Finalized
├─ 0.000 MPI_Init
├─ 0.000 MPI_Initialized
├─ 26.264 comm
│  └─ 26.264 comm_large
│     ├─ 0.217 MPI_Gather
│     └─ 26.047 MPI_Scatter
├─ 102.152 comp
│  └─ 102.152 comp_large
│     └─ 79.123 comp
│        └─ 29.059 comp_small
├─ 0.771 correctness_check
└─ 26.836 data_init_runtime
```

Radix Sort

The following tree comes from a run with 2^28 Sorted-organized elements on 64 processors. The average time per rank is displayed in the tree.

```
4.900 main
├─ 0.000 MPI_Init
├─ 0.717 data_init_runtime
├─ 1.279 comm
│  ├─ 0.919 comm_small
│  │  ├─ 0.846 MPI_Scatter
│  │  └─ 0.073 MPI_Allreduce
│  └─ 0.360 comm_large
│     ├─ 0.312 MPI_Gather
│     └─ 0.044 MPI_Scatter
├─ 0.010 comp
│  ├─ 0.000 comp_small
│  └─ 0.010 comp_large
├─ 0.001 MPI_Barrier
├─ 0.000 MPI_Finalize
├─ 0.761 correctness_check
├─ 0.000 MPI_Initialized
├─ 0.000 MPI_Finalized
└─ 1.219 MPI_Comm_dup
```

3c. Collect metadata

This section contains metadata for each algorithm generated using Caliper and Thicket.

Bitonic Sort

This metadata is from the same run as the bitonic calltree above.

Category	Value
profile	4183253281
cali.caliper.version	2.11.0
mpi.world.size	64
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	Cali_Files/bitonic_Sorted_268435456_64.cali
spot:profile.mpi	TRUE
spot:region.count	TRUE
spot:time.exclusive	TRUE
spot:time.variance	TRUE
launchdate	1728927077
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.sc '/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. '/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/lib
cmdline	['./sort', 'bitonic', 'Sorted', '268435456']
cluster	c
algorithm	bitonic
programming_model	mpi
data_type	unsigned int
size_of_data_type	4
input_size	268435456
input_type	Sorted
num_procs	64
group_num	10
implementation_source	handwritten
scalability	strong

Sample Sort

This metadata is from the same run as the calltree above.

Category	Value
profile	4183253281
cali.caliper.version	2.11.0
mpi.world.size	64
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	Cali_Files/sample_Random_268435456_64.cali
spot:profile.mpi	TRUE
spot:region.count	TRUE
spot:time.exclusive	TRUE
spot:time.variance	TRUE
launchdate	1728927077
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.1 '/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 '/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. '/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/lib '/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/lib6
cmdline	['./sort', 'sample', 'Random', '268435456']
cluster	c
algorithm	sample
programming_model	mpi
data_type	unsigned int
size_of_data_type	4
input_size	268435456
input_type	Random
num_procs	64
group_num	10
implementation_source	handwritten
scalability	strong

Merge Sort

This metadata is from the same run as the Merge calltree above.

Category	Value
profile	1161980275
cali.caliper.version	2.11.0
mpi.world.size	64
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

Category	Value
spot.channels	regionprofile
cali.channel	spot
spot:node.order	TRUE
spot:output	Cali_Files/merge_Random_268435456_64.cali
spot:profile.mpi	TRUE
spot:region.count	TRUE
spot:time.exclusive	TRUE
spot:time.variance	TRUE
launchdate	1729134035
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.1 '/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 '/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. '/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/lib '/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/lib6
cmdline	['./sort', 'merge', 'Random', '268435456']
cluster	c
algorithm	merge
programming_model	mpi
data_type	unsigned int
size_of_data_type	4
input_size	268435456
input_type	Random
num_procs	64
group_num	10
implementation_source	handwritten
scalability	strong

Radix Sort

This metadata is from the same run as the Radix calltree above.

Category	Value
profile	4103088183
cali.caliper.version	2.11.0
mpi.world.size	64
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.order	TRUE
spot	Cali_Files/radix_Sorted_268435456_64.cali
spot.mpi	TRUE
spot.count	TRUE
spot.exclusive	TRUE
spot.variance	TRUE
launchdate	1728940918

Category	Value
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.1 '/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.sc '/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. '/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/lib '/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/lib6
cmdline	['./sort', 'radix', 'Sorted', '268435456']
cluster	c
algorithm	radix
programming_model	mpi
data_type	unsigned int
size_of_data_type	4
input_size	268435456
input_type	Sorted
num_procs	64
group_num	10
implementation_source	handwritten
scalability	strong or weak?

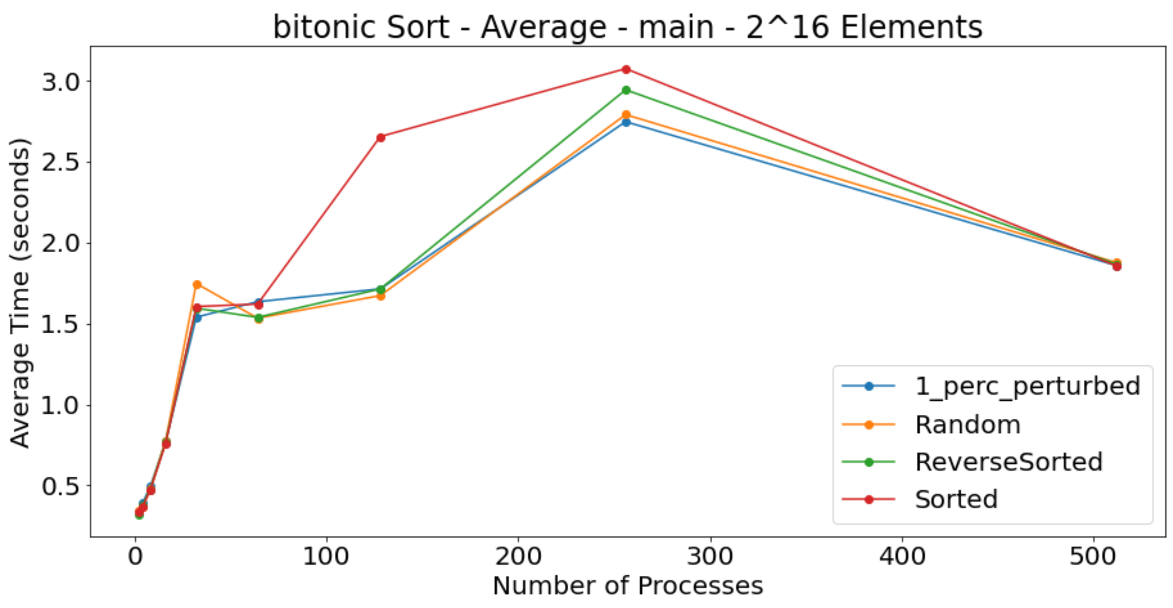
4. Performance evaluation

Bitonic Sort

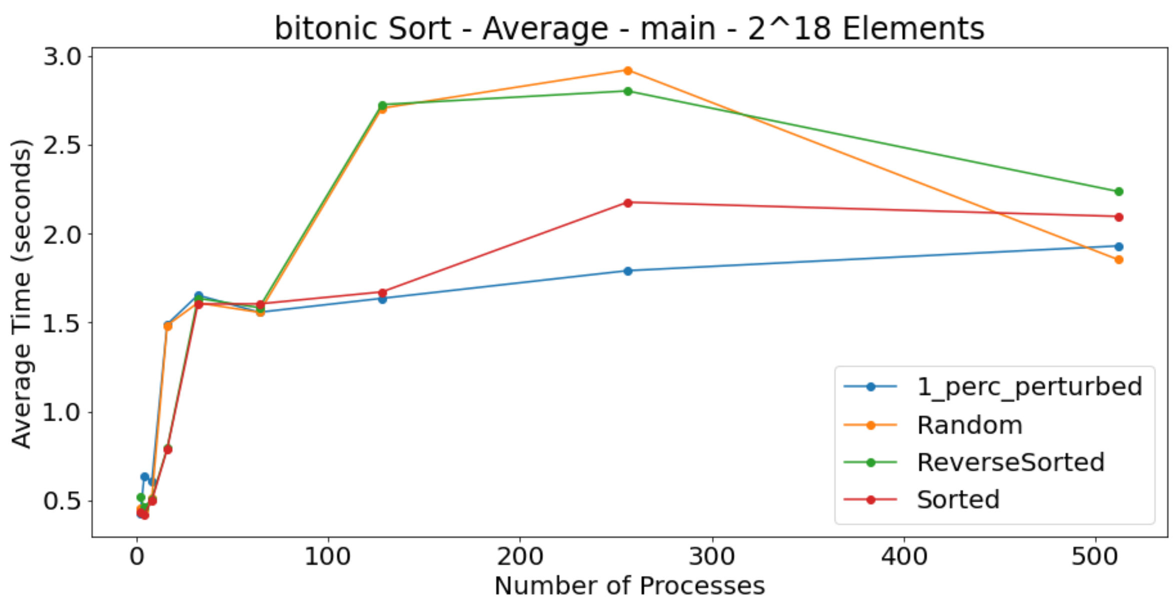
In the analysis of my bitonic sort's performance, I used a subset of graphs generated using Thicket. I was able to perform all runs and retrieve all Cali files besides the 1024 processes due to them timing out due ot grace resources not being available. All of the graphs I will present use strong scaling.

Strong Scale Main

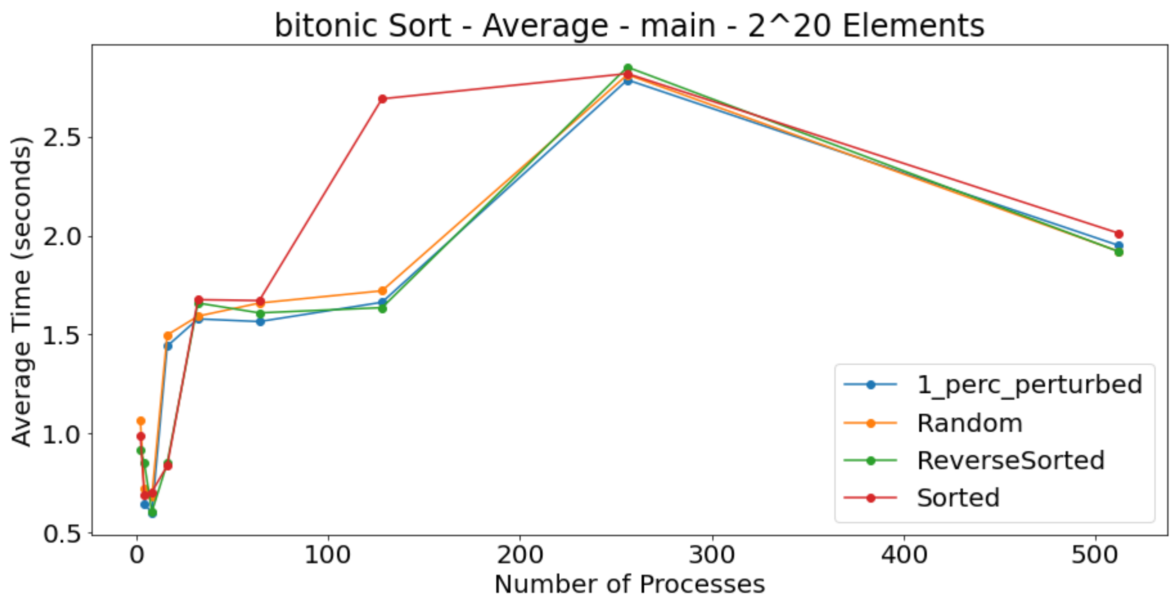
bitonic_Average_main_2^16



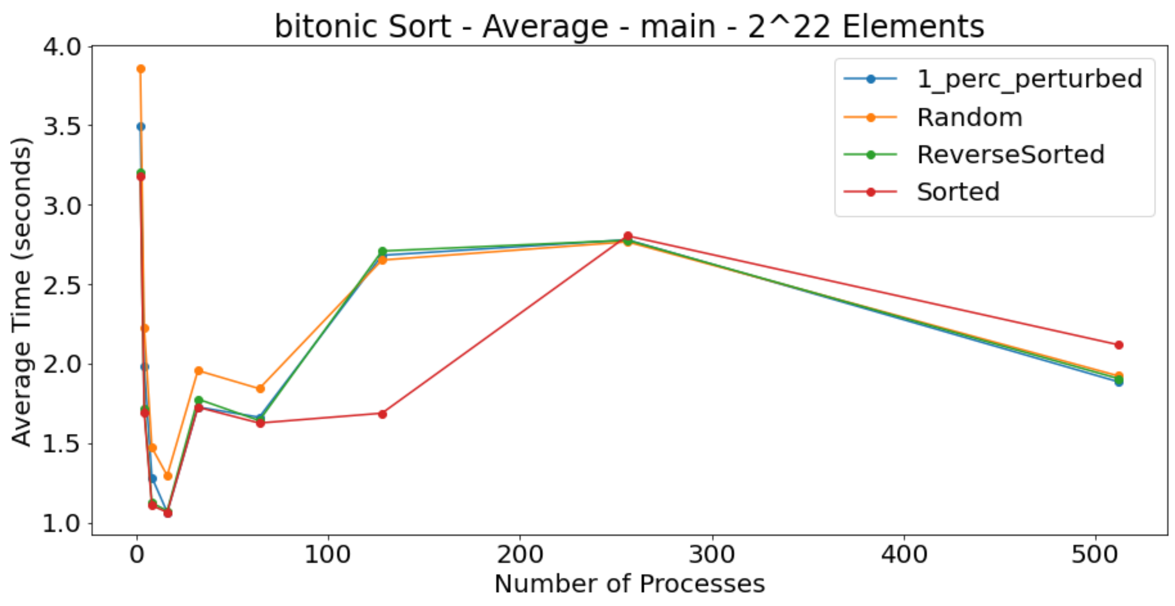
bitonic_Average_main_2^18



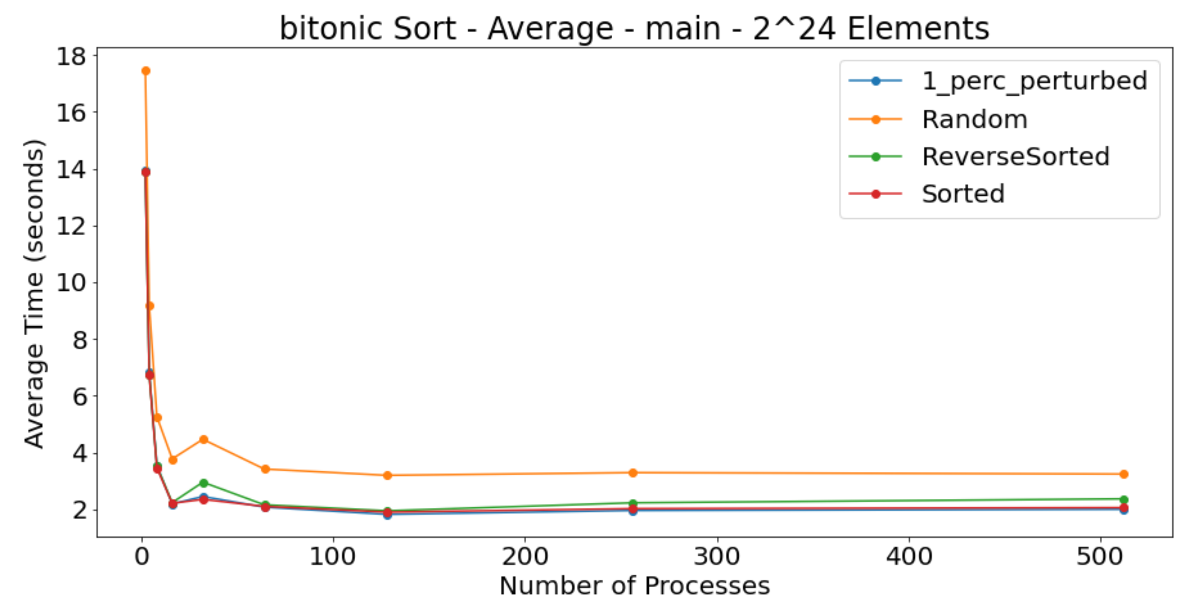
bitonic_Average_main_2^20



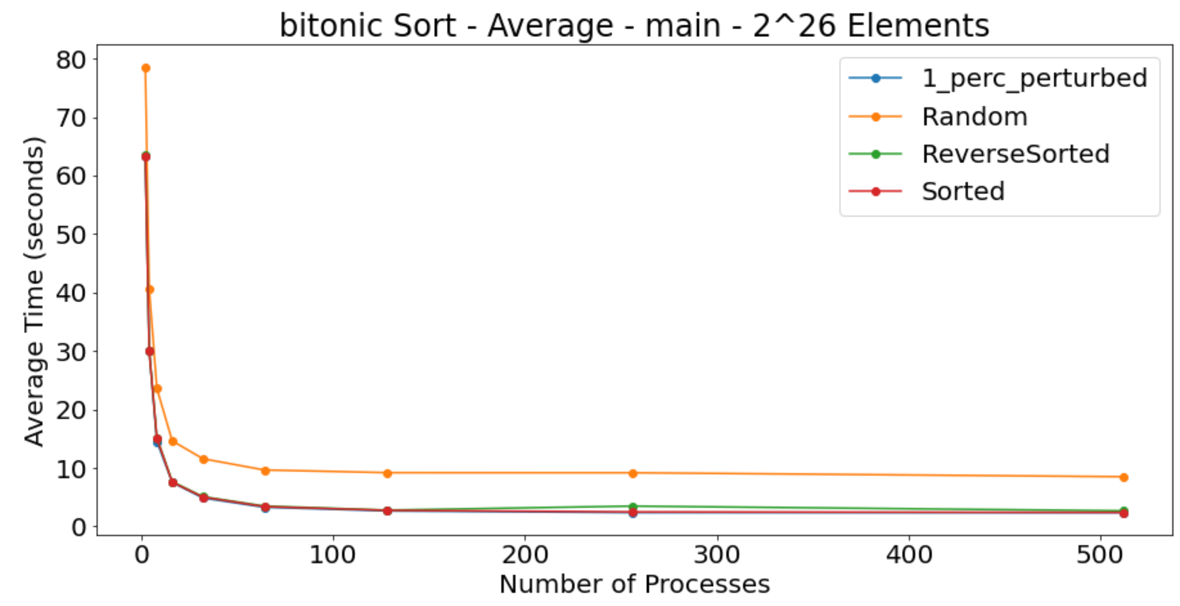
bitonic_Average_main_2^22



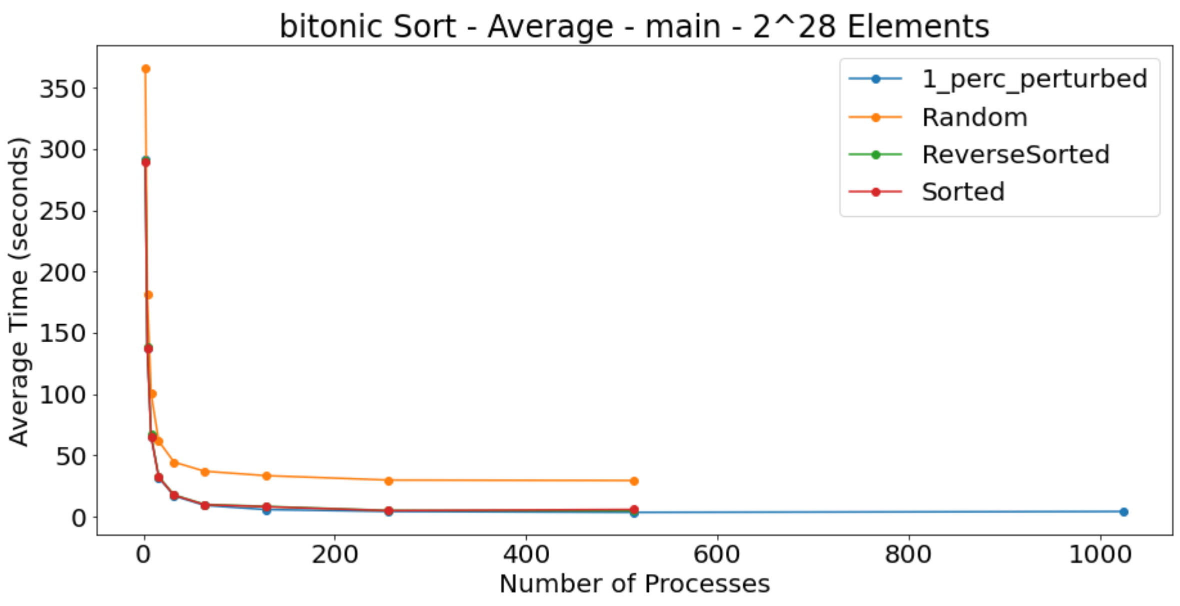
bitonic_Average_main_2^24



bitonic_Average_main_2^26



bitonic_Average_main_2^28

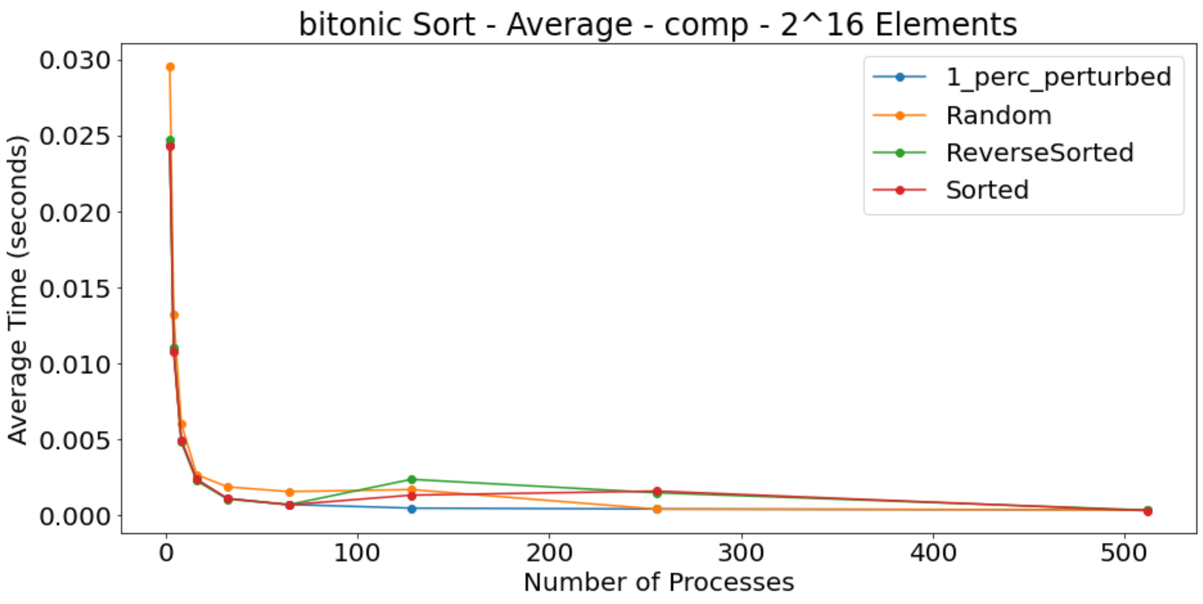


Observations

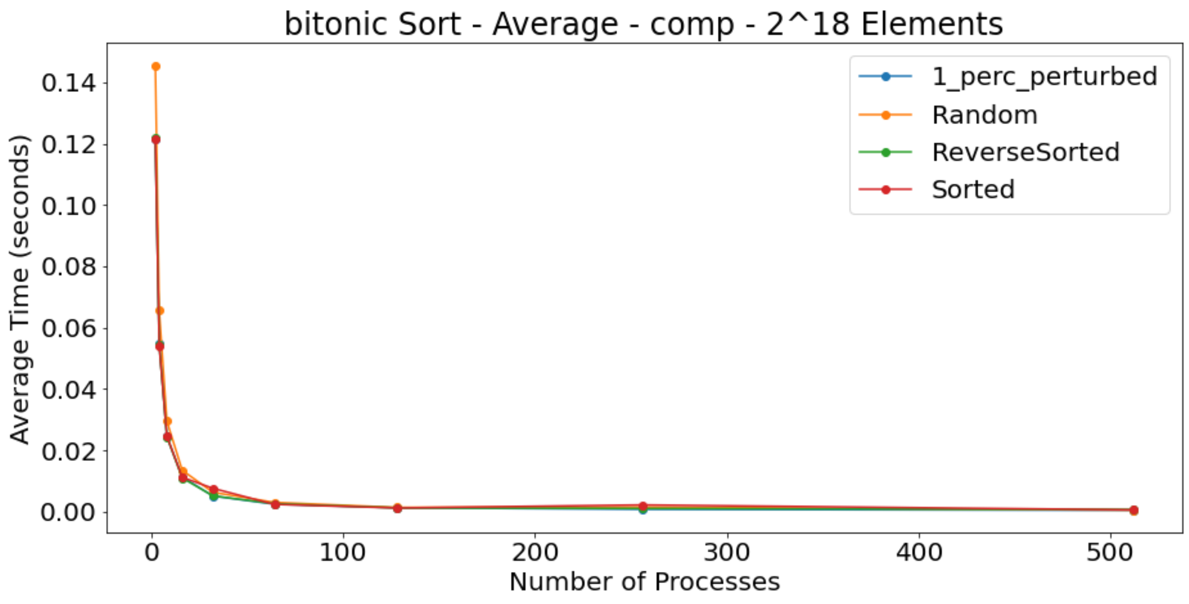
The trend between the graphs and different input sizes for the Main scale which is as the number of processes increases, there is an initial increase in time in sizes before 2^22. This could be because of the inefficiency of increasing the number of processes initially making it more costly than beneficial as the size doesn't require that many processes. However after 2^22, the average time decreases with the number of processes increasing. Random sorted did the best out of all the sorted methods which is increasing which shows a useful efficiency improvement to processes in the sorting process for random sorting as opposed to other input types. However, another interesting observation is the difference in performance across some input types: sorted, reverse sorted, and 1 % perturbed input arrays seem to have on average the same performance.

Strong Scale Comp

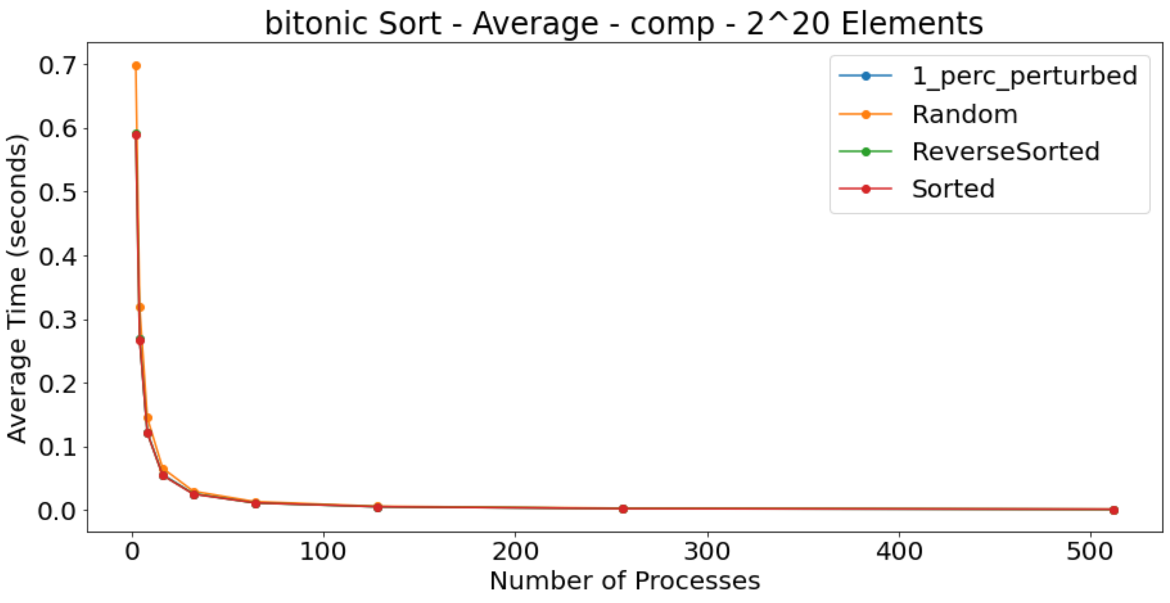
bitonic_Average_comp_2^16



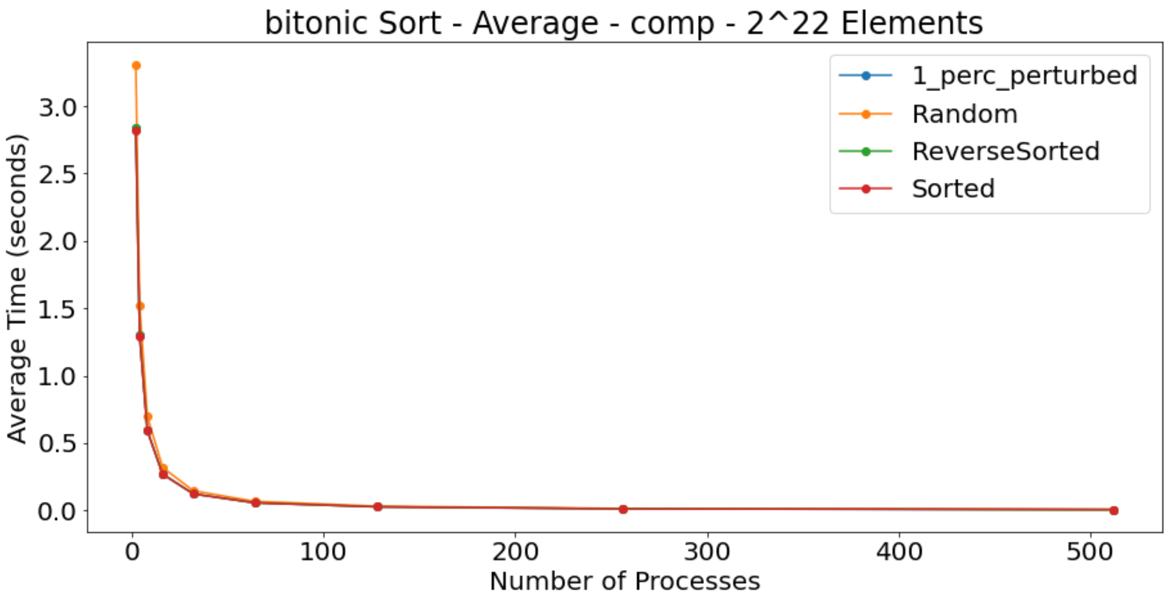
bitonic_Average_comp_2^18



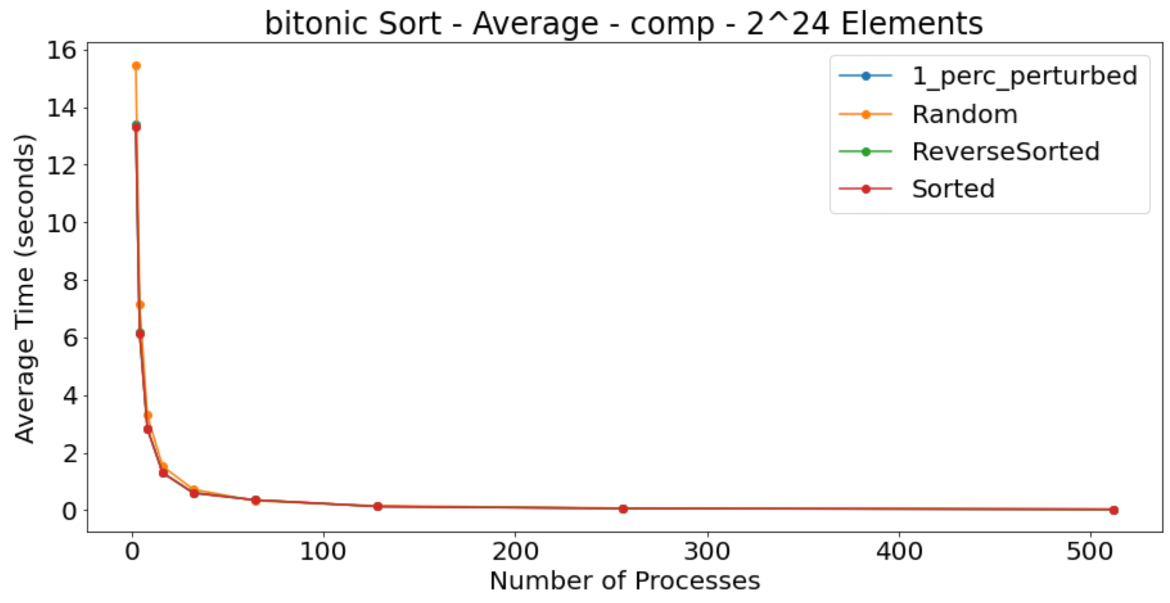
bitonic_Average_comp_2^20



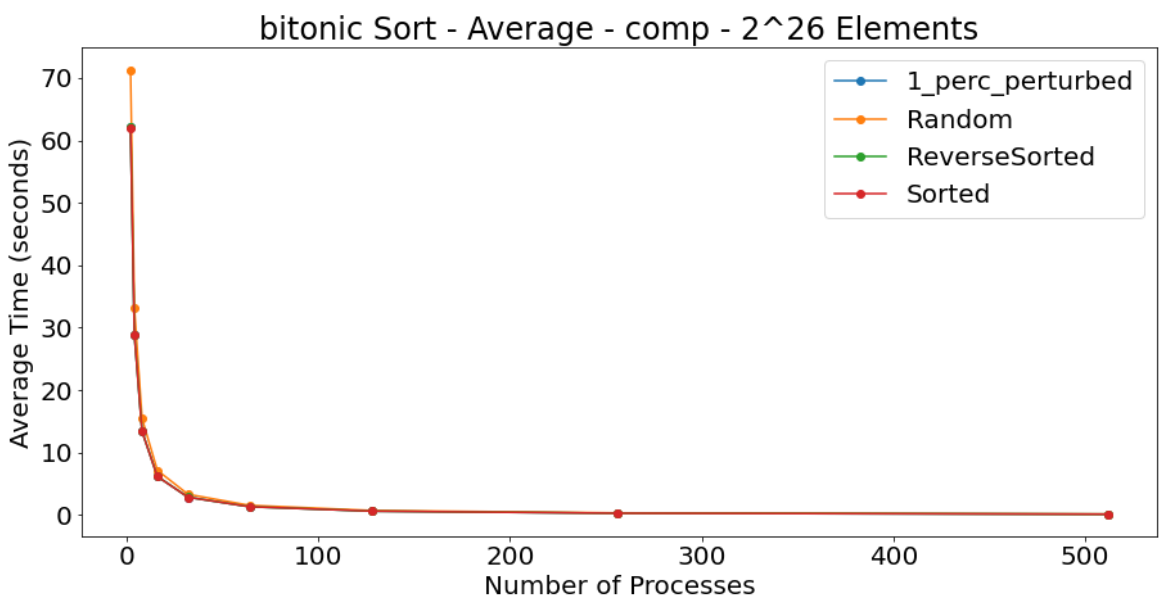
bitonic_Average_comp_2^22



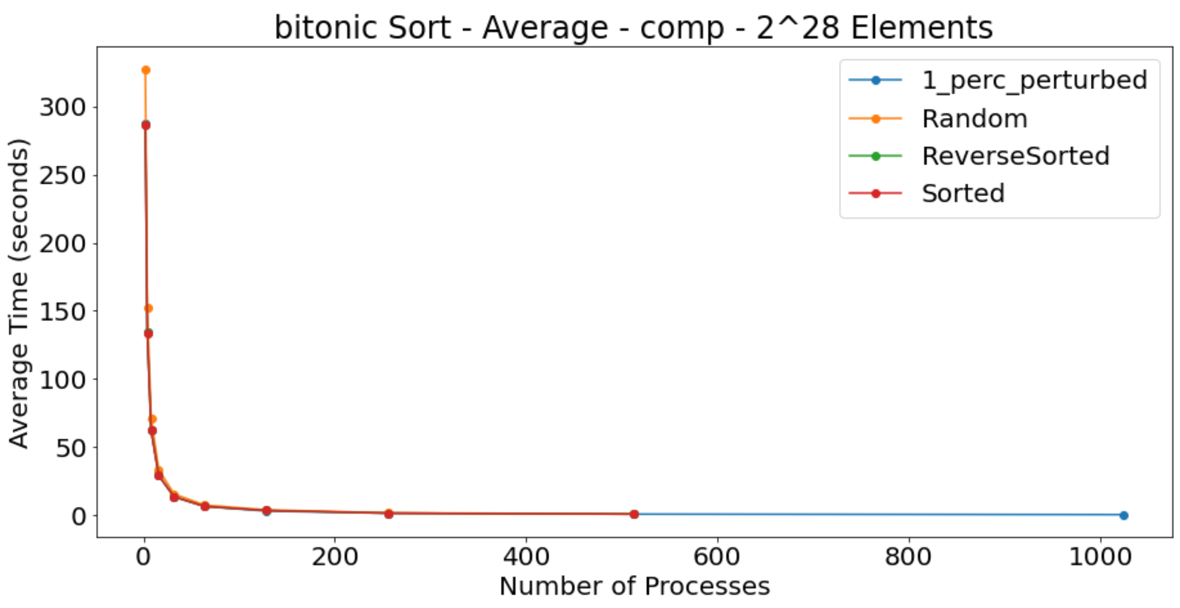
bitonic_Average_comp_2^24



bitonic_Average_comp_2^26



bitonic_Average_comp_2^28

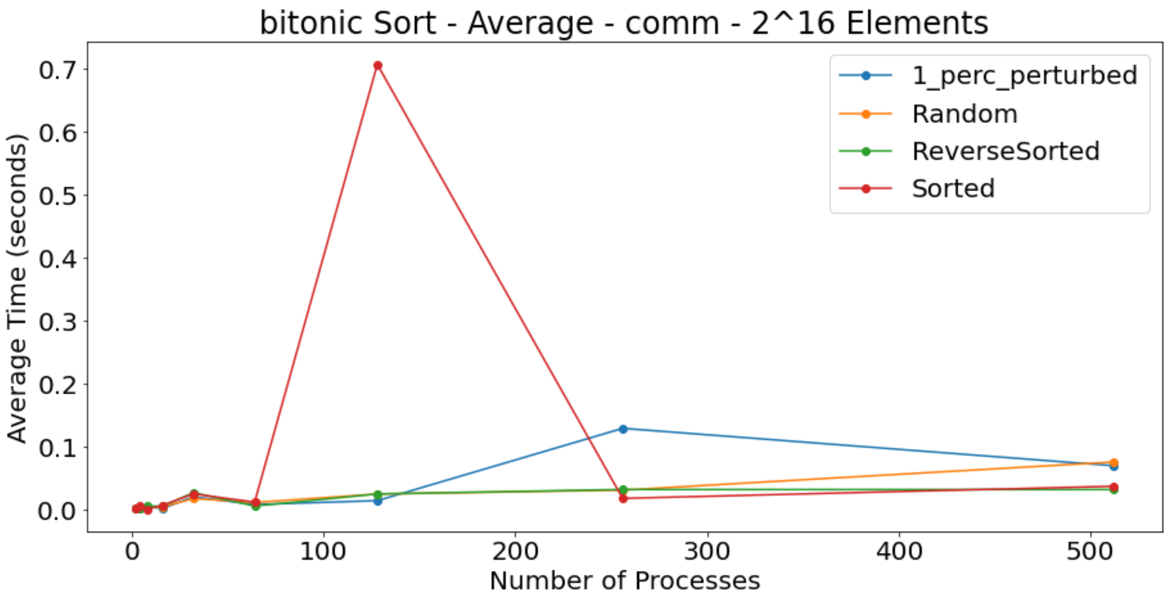


Observations

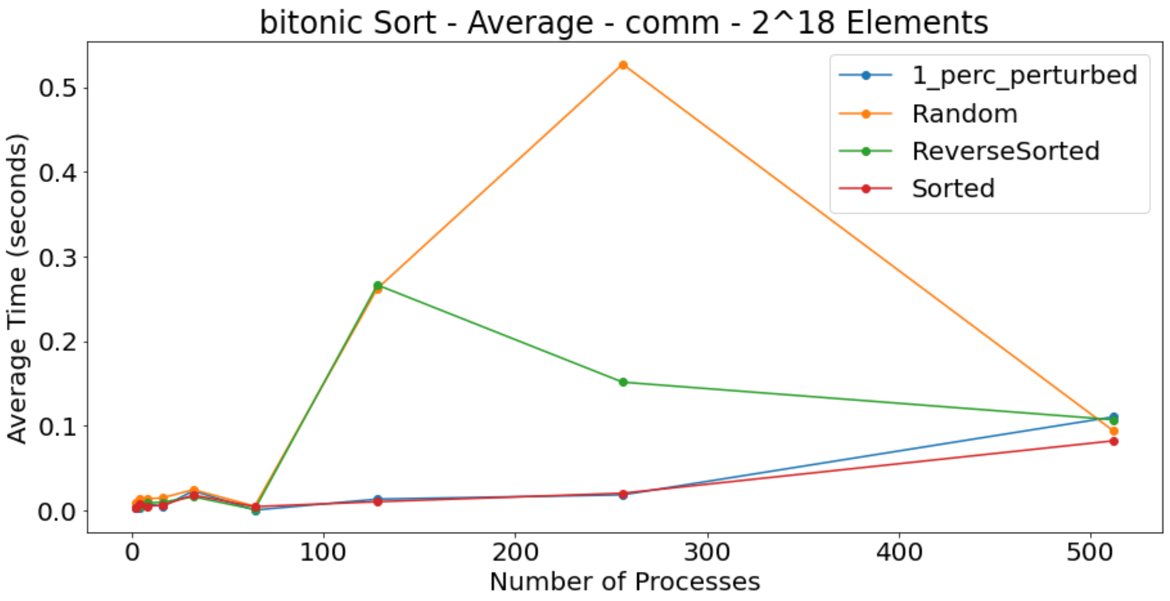
Overall the trend for these graphs are the computation time for each of the sizes seem to be exponentially decreasing with an increasing number of processes. For all of the graphs in bitonic sort this trend is consistent. The trend also shows that on average all of the input types roughly have the same performance without any deviations or spikes.

Strong Scale Comm

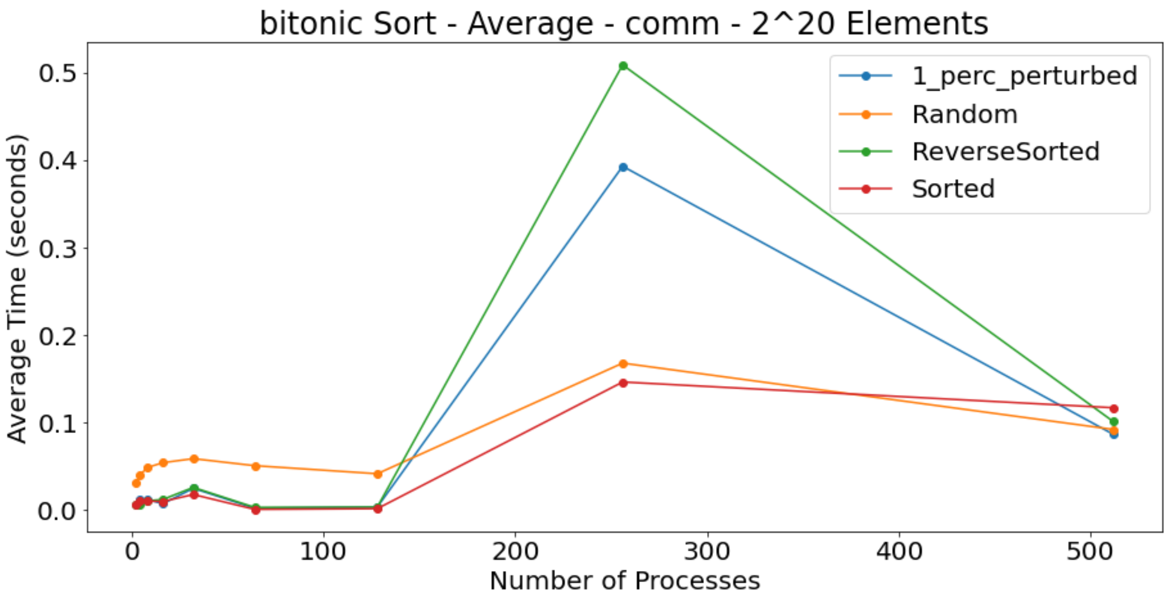
bitonic_Average_comm_2^16



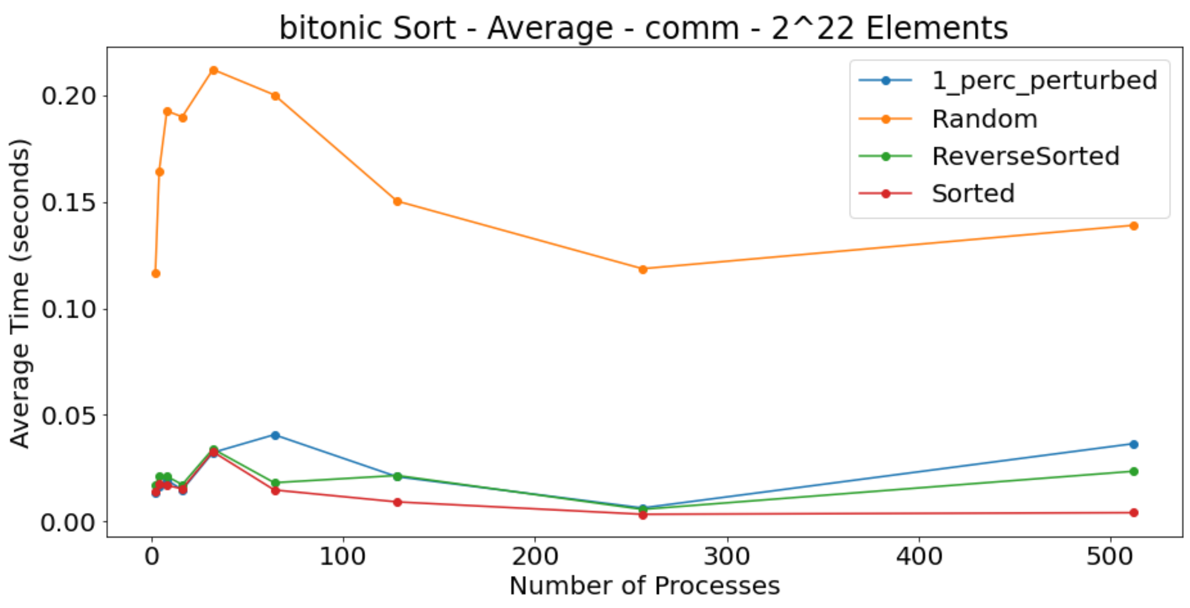
bitonic_Average_comm_2^18



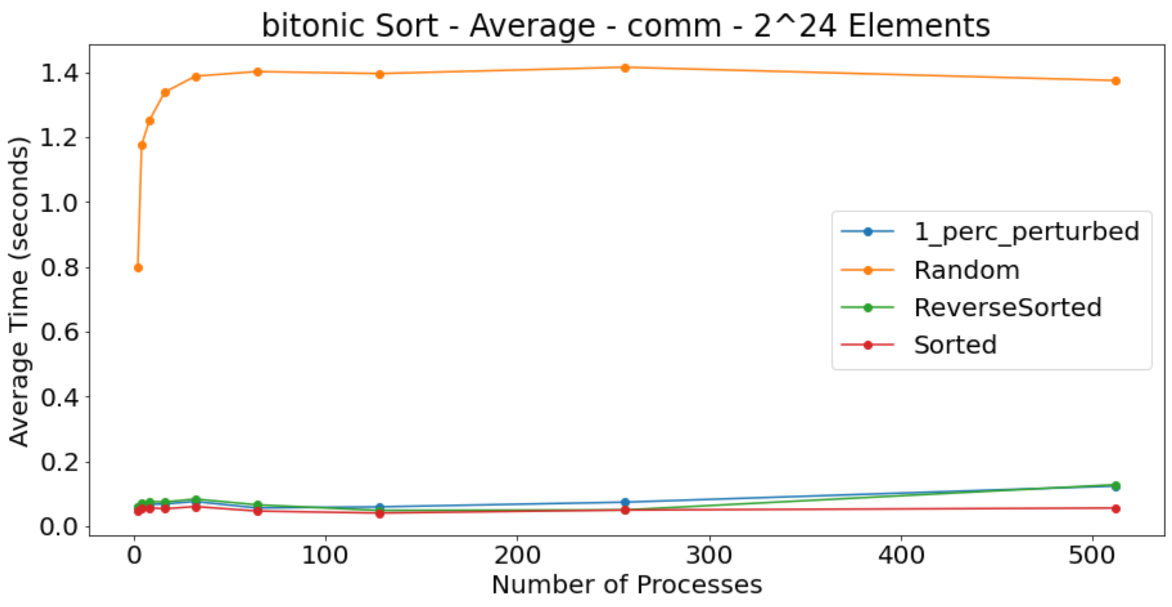
bitonic_Average_comm_2^20



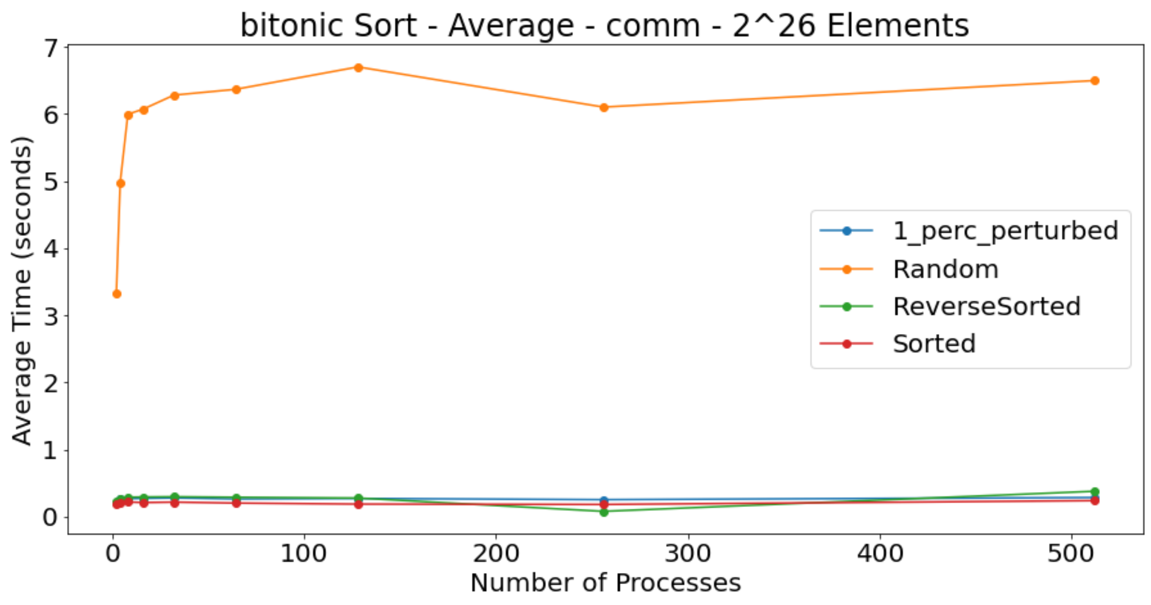
bitonic_Average_comm_2^22



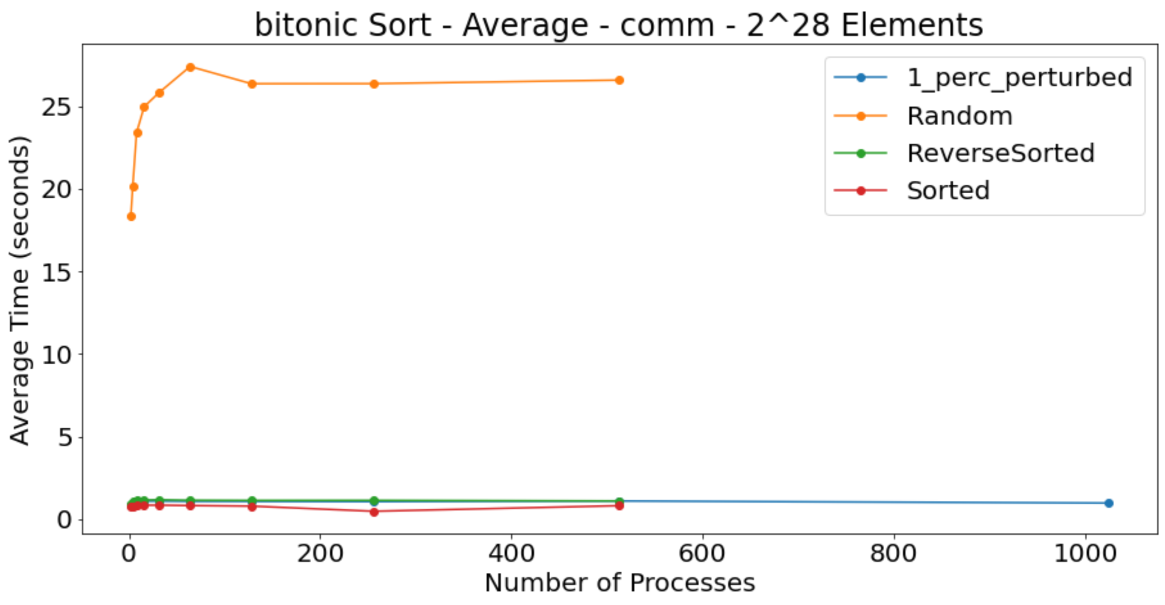
bitonic_Average_comm_2^24



bitonic_Average_comm_2^26



Radix_Average_comm_2^28



Observations

In graphs 2^22 and above, the time for random sorting is quite high, which is a general trends for the larger input sizes. This makes sense as the number of processes increase, teh time it takes to communicate between processes increases, thus overall increases the time. In the early graphs before 2^22, it shows a trend of spikes with reverse sorted having the highest spike in 2^20, sorted having a spike in 2^16 and random sorting being the highest in 2^18 while also having a spike.

Sample Sort

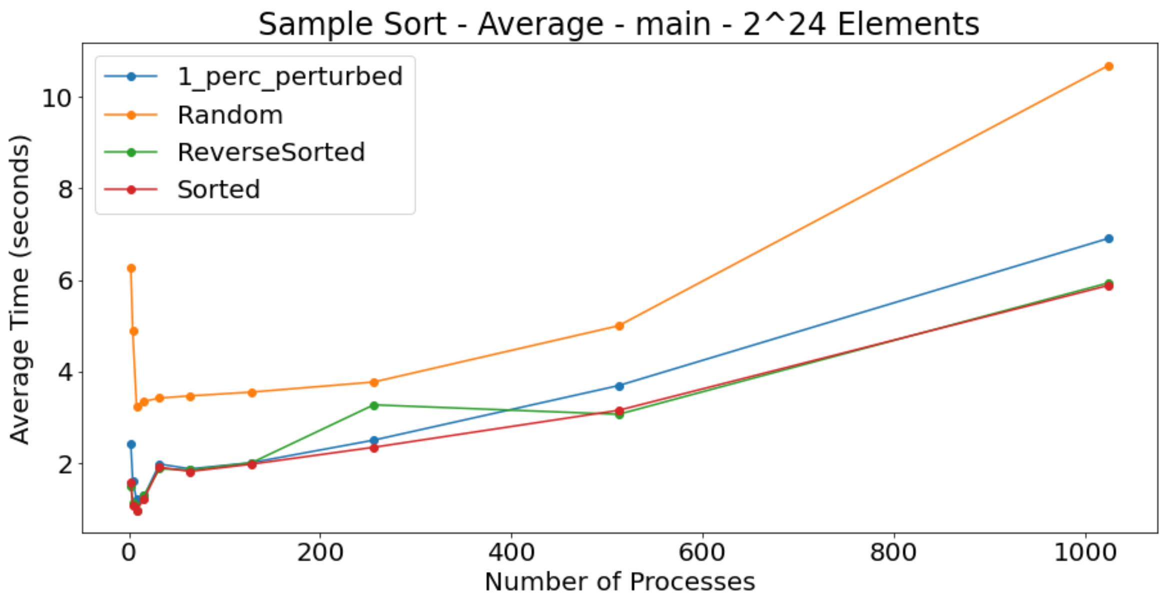
For the analysis of my sample sort's performance, I will be using a subset of graphs generated using Thicket. I was able to perform all 280 runs and retrieve all 280 Cali files. All of the graphs I will present use strong scaling.

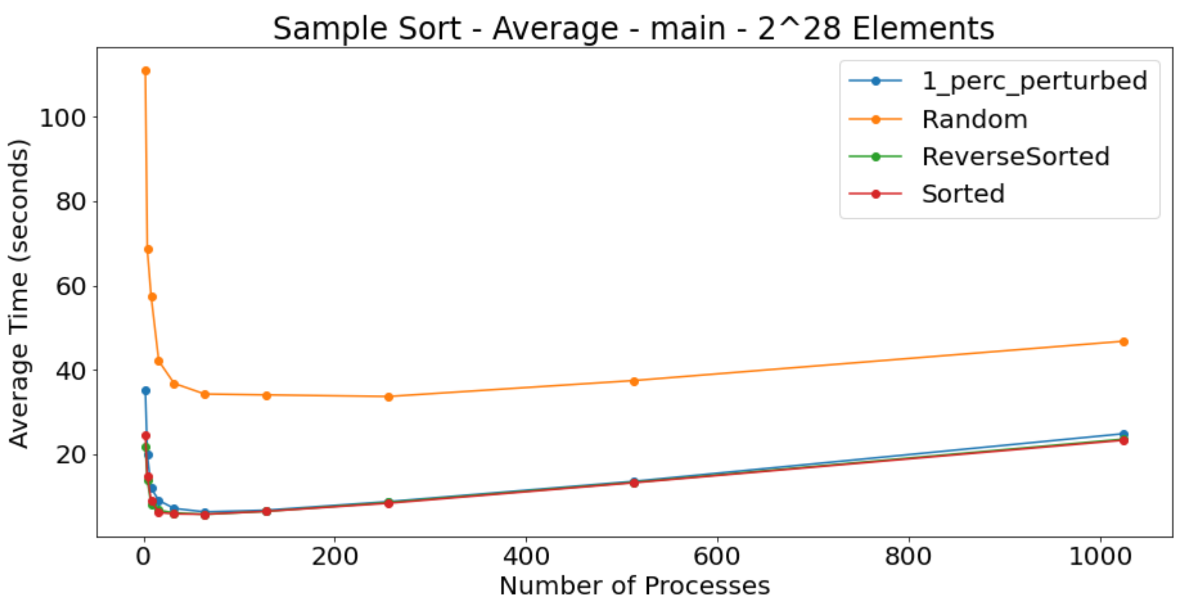
Overall Performance

We will begin by analizing the overall performance of the sort by looking at the main region. When looking at the main region, we see that there does seem to be some good speedup as the number of processes increase. After a certain number of processors, however, this speed up breaks down and we see our performance begin to get continualy poorer.

Another interesting observation is the difference in performance across the various input types. The sorted, reverse sorted, and 1 % perturbed input arrays seem to have roughly the same performance. On the other hand, the random input is significantly slower than the rest.

Keeping these two trends in mind, we will examine the performance of computation-based and communication-based regions of the code independently in order to isolate where our performance issues are coming from.





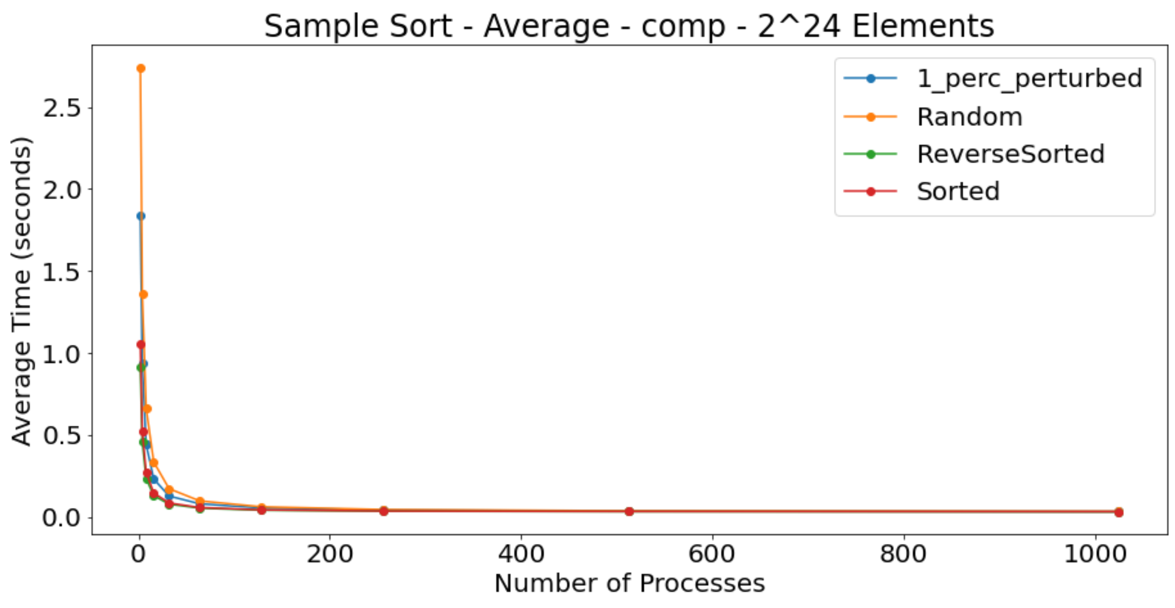
Computation Performance

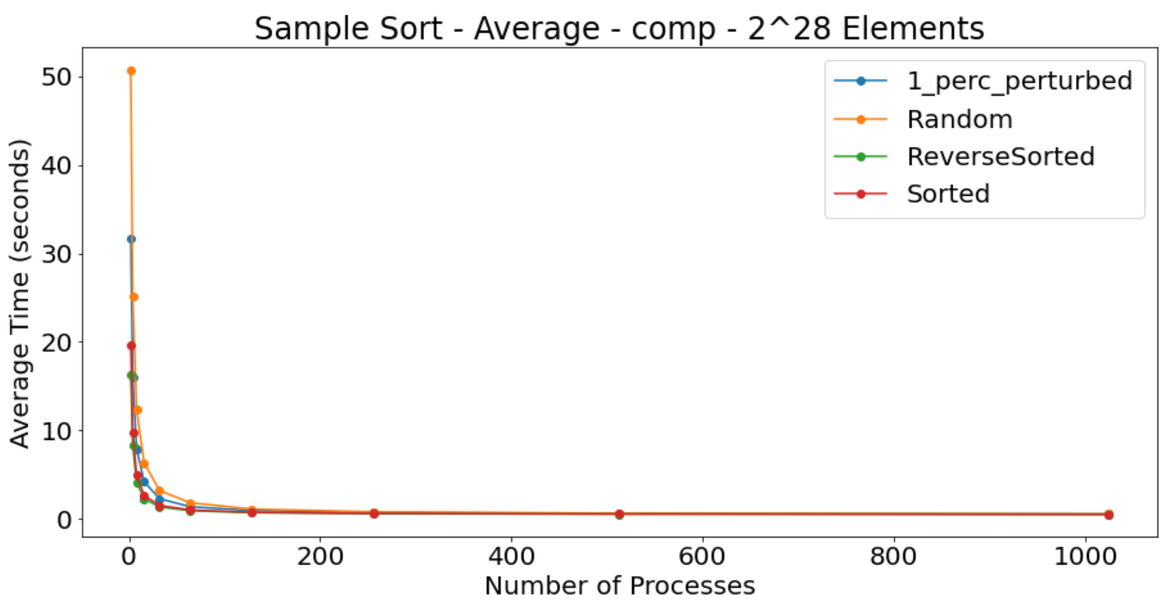
Looking at the computation-based performance, we actually see the trend we're looking for.

The speedup as the number of processes increases does not break down as the number of processes passes a certain threshold as we saw in the overall performance. We see roughly exponential decay in the average computation time as the number of processes grows exponentially. In the context of our sort, this means that the actual sorting of each bucket is speeding up, hinting at the fact that the elements in our array is being nicely distributed across our buckets.

This being said, it is important to notice that past a certain point the decrease in actual computation time is minimal as the sorting time for each bucket is so low. This hints at the fact that the decrease in computation time eventually becomes outweighed by a larger increase in communication time, but we will further explore this theory in the next section.

We don't see any significant difference in computation time when it comes to the input type. This is likely due to the fact that the process of sorting each bucket does not change and the sort underlying std::sort is probably well designed to handle inputs of various types/orientations.





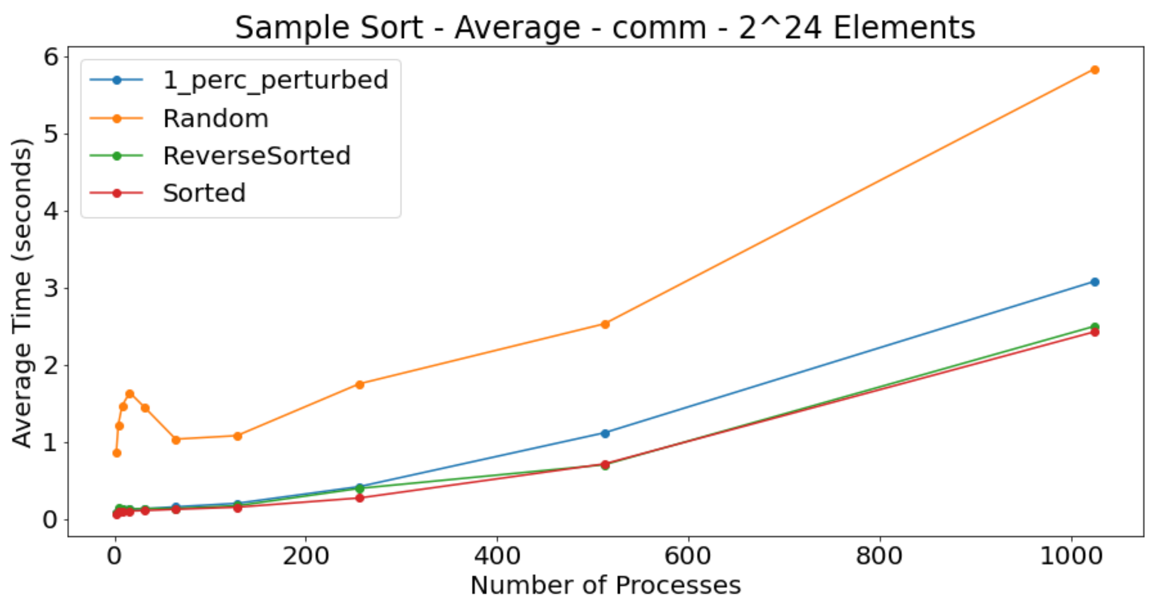
Communication Performance

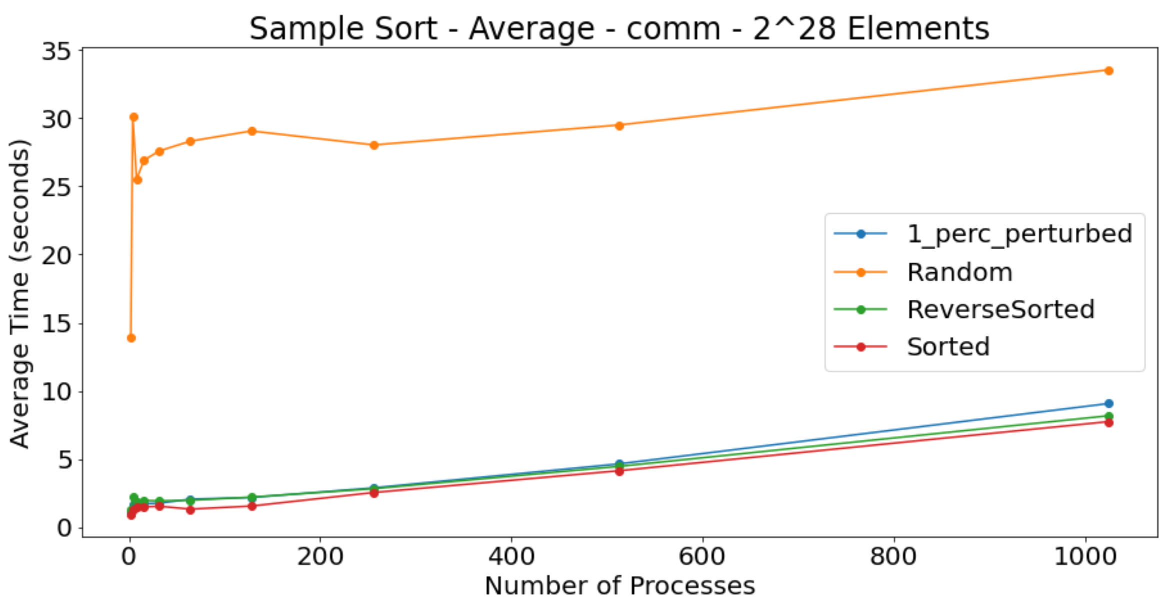
A quick look at the communication-based performance hints at where our performance issues are coming from.

As the number of processes grows, the average communication time of our algorithm roughly increases. This makes sense, as there will be more processes that need to communicate. Each process has a bucket, so the communication overhead to send each process their bucket and then merge each process's bucket back together is going to be quite large. Even if the size of each communication is decreasing as the number of processes increases, the amount of communication is greatly increases.

This verifies the theory brought up in the previous section. Though the computation time is continualy decreasing as the number of processes increases, the communication overhead is increasing. When the reduction in bucket size is significant, the reduction in computation time outweighs the increase in communication overhead and the overall performance improves. Once the bucket size reaches a certain "small enough" bucket size, though, the communication overhead outweighs the computation speedup and the overall performance suffers. The number of processors needed for this "small enough" bucket size increases as the input size increases, which is why the cutoff for overall performance increase exists at a higher number of processors for larger input sizes.

We also seem to have found the culprit for why the random input is taking longer than the other inputs. I am not exactly certain why this is the case. It could have something to do with the way the computational regions are resulting in the communications being reached. It could also have to do with less empty buckets resulting in more communication between processes to send their buckets (algorithm does not send empty buckets). I do not have a clear answer of why this is the case, but it is clear that this is the origin of the difference in random input performance that we saw when looking at the overall performance.

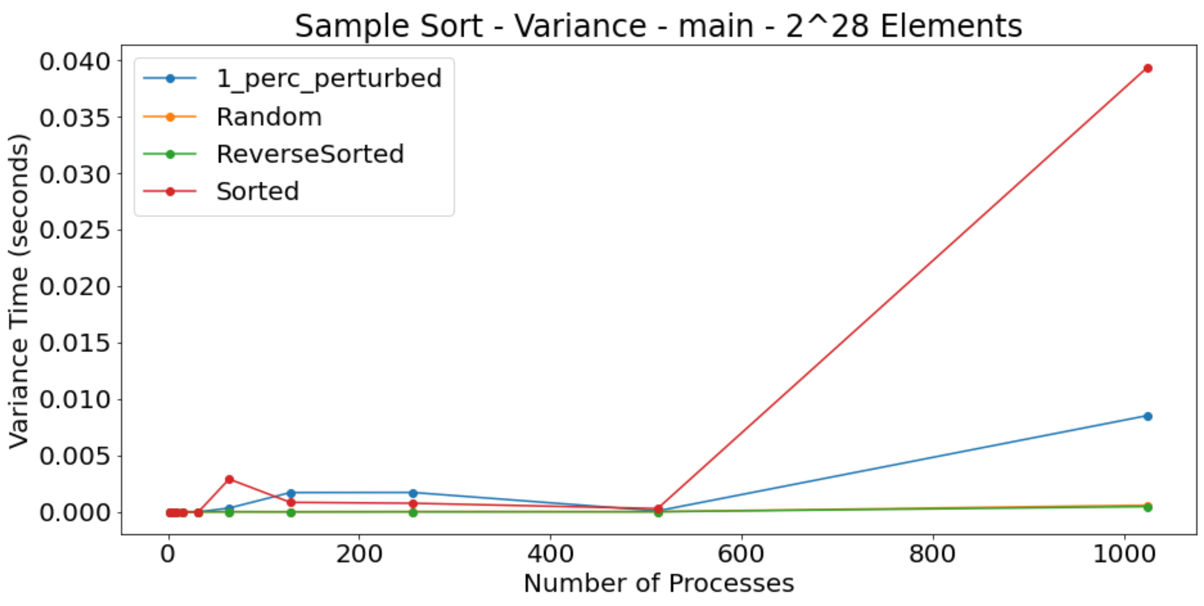


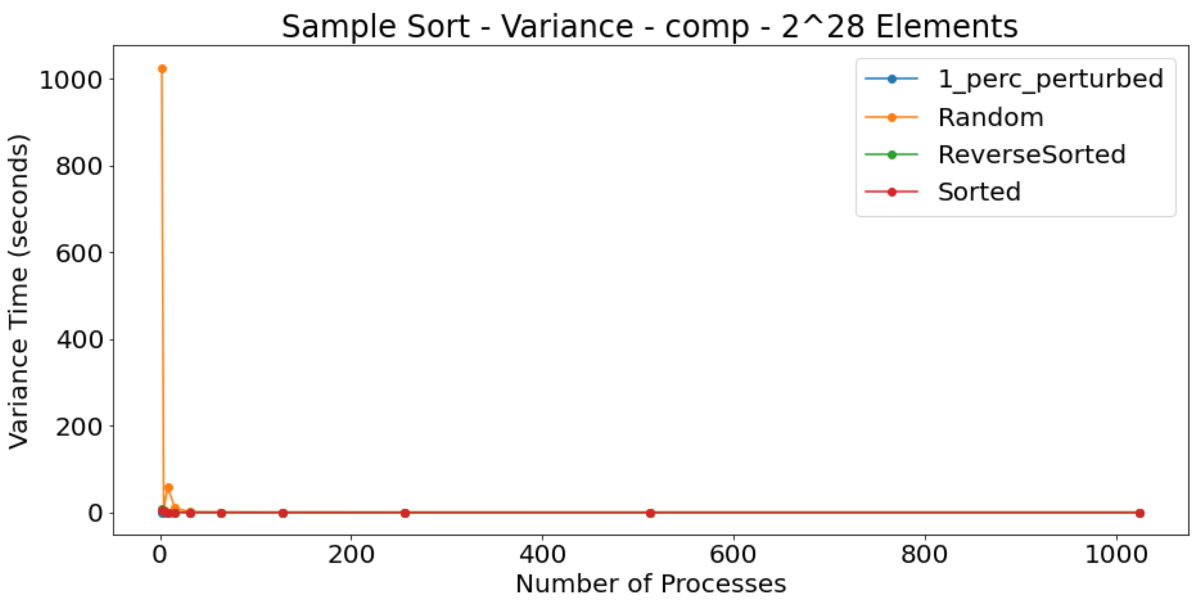


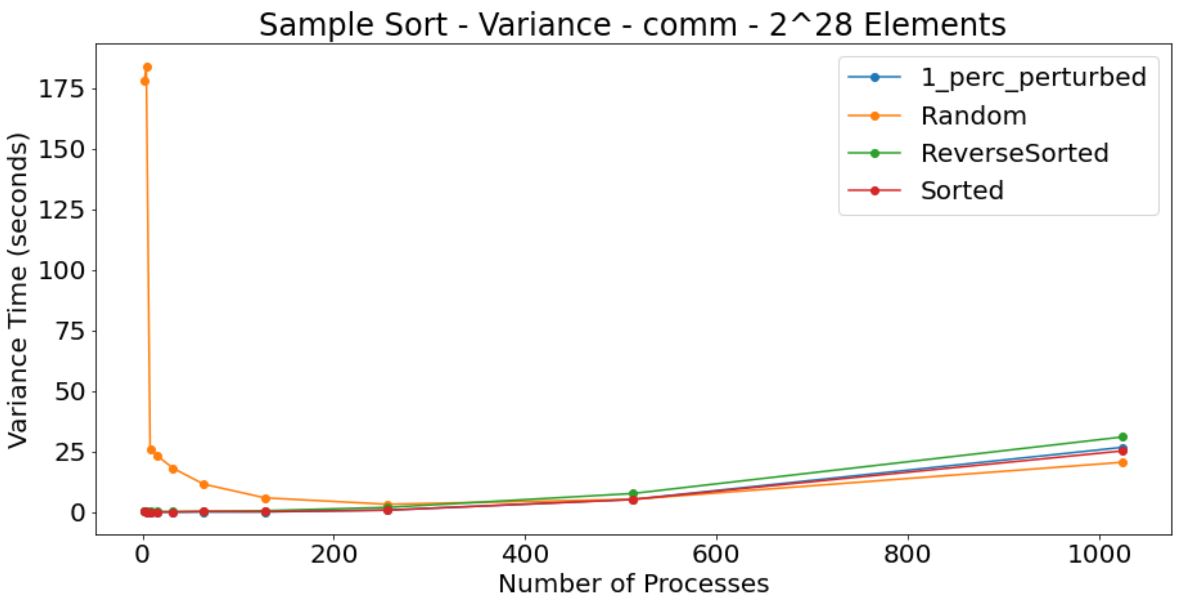
Variation in Performance Metrics

To finish off my analysis, we will take a quick look at the variance in performance time across processes.

There seems to be a slight increase in variance as the number of processes increase for each of the regions. With the lower number of processes, it is hard to interpret the variance as there are very few data points. The spike in the computation region is likely due to the root processes having more work to do to facilitate the sort.

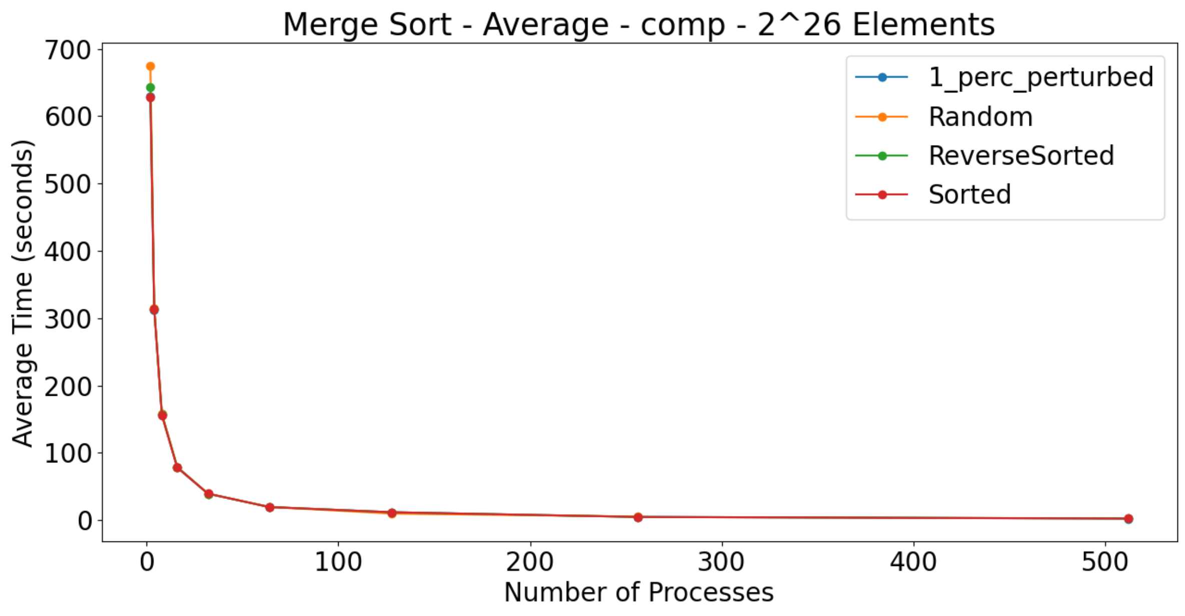




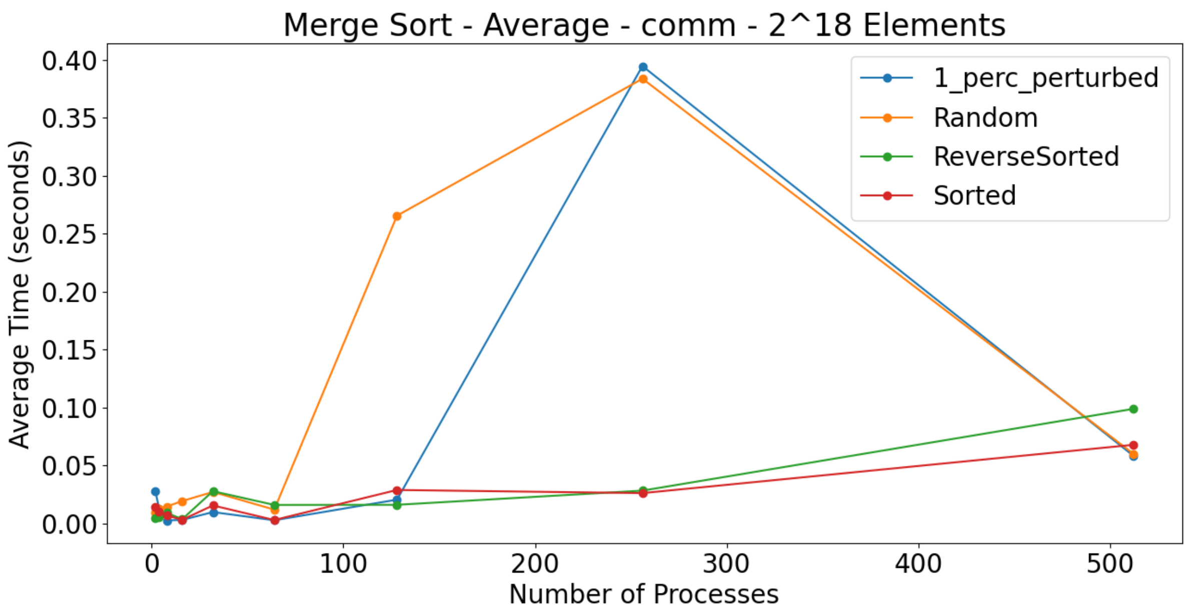


Merge Sort

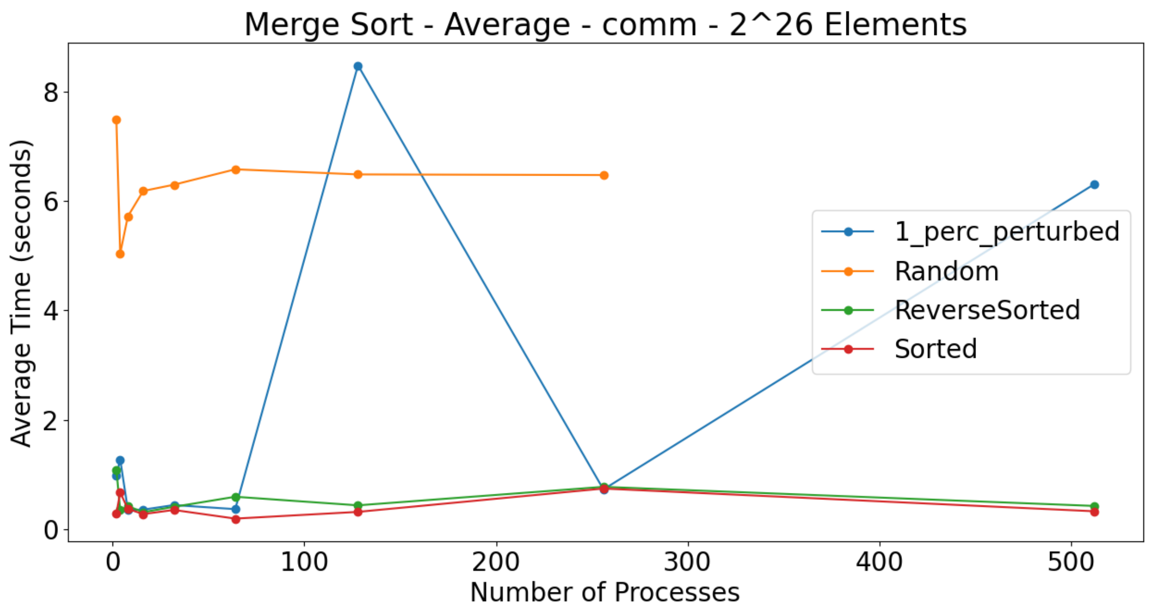
Computation Times: Merge Sort displayed expected behaviors with the computation times: as the number of processes increases, the average time of computation decreases, as more processors divide and conquer the sorting computation.



Communication Times: For the most part, merge sort displayed the expected trend of increasing of time taken with an increase in the number of processors. Take for example this figure:

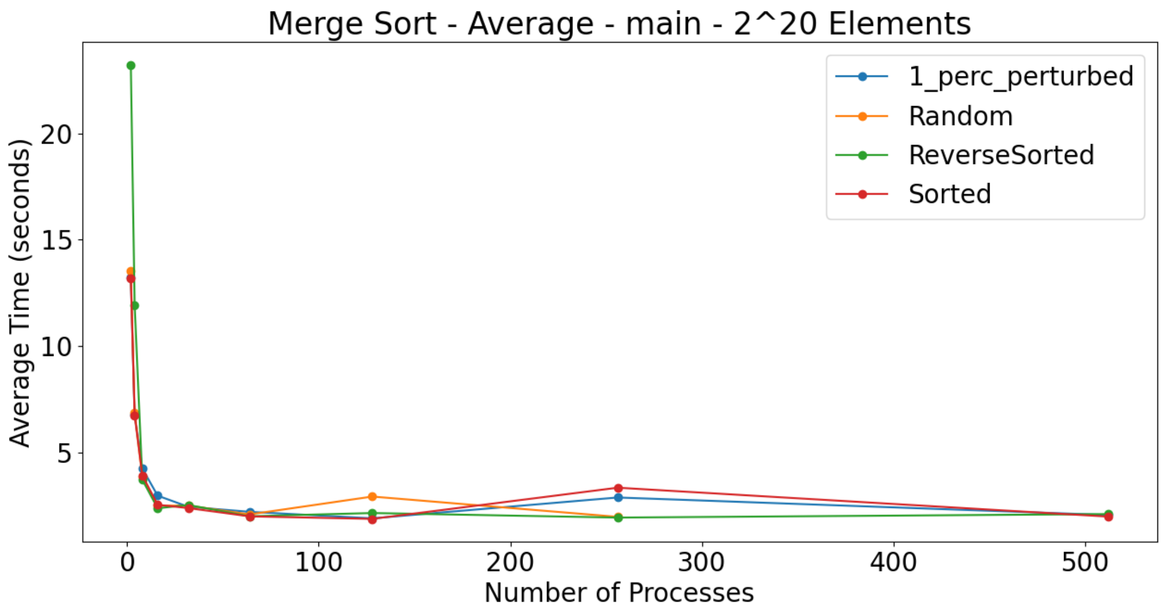


However, some of these numbers and figures seem odd. Such as this one figure:

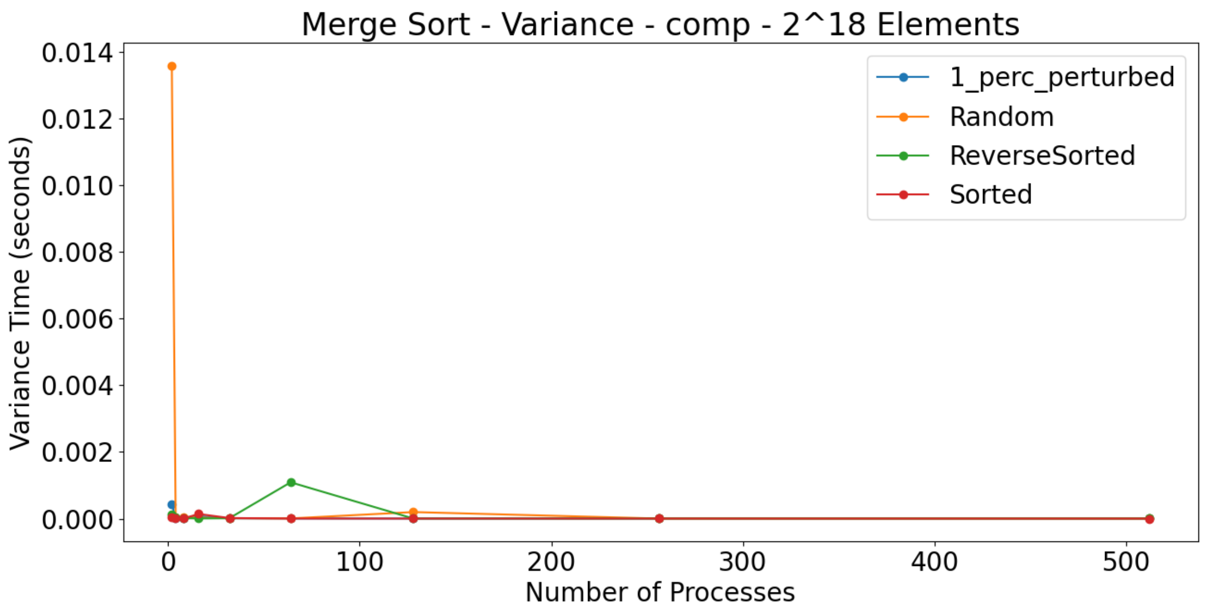


This could be due to a number of reasons, but we personally believe that this is due to the large amount of students rushing this assignment near the deadline, causing a surge in grace activity, causing possible congestion issues. This is supported by the variance of the merge sort graphs, which we will get into later in the report.

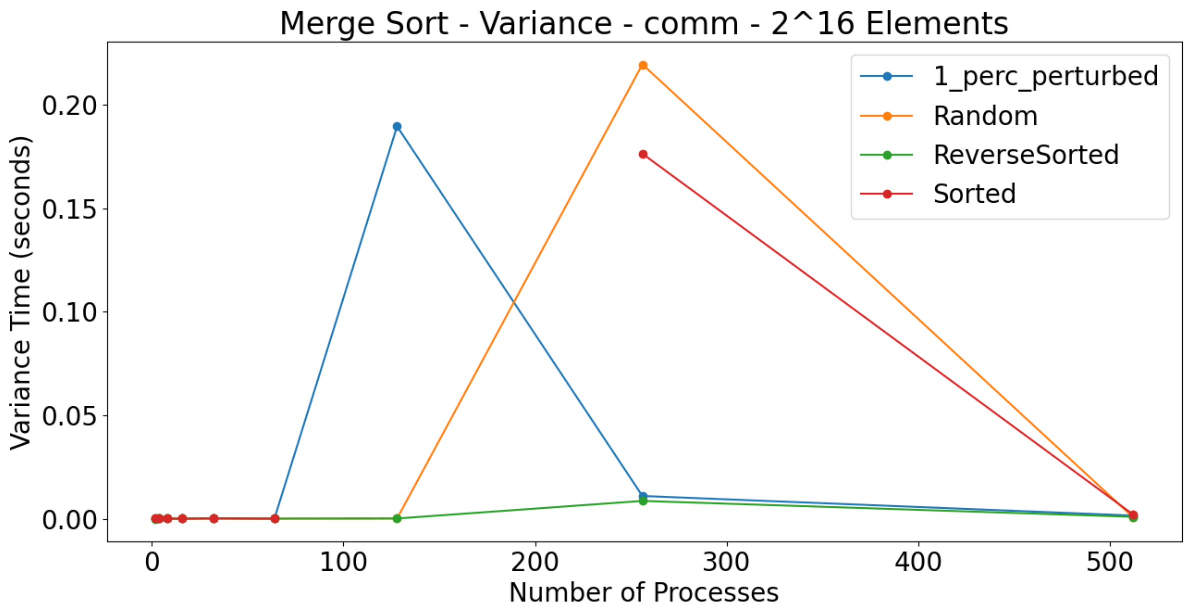
Overall (Main) Times: Although there were issues with the communication times between processes, the overall trend was as expected; as the number of processes increases, the time until completion decreases up until a point, when the line flattens and there are diminishing returns.



Variance: As mentioned before, Grace was very congested. Many of the jobs we put into grace did not even complete, such as the 1024 process sorts, which is why the graphs only go up to 512 processes. This is reflected in the Variance, where the computation variation was very low across the board,



but the communication variance was everywhere for some of the tests:

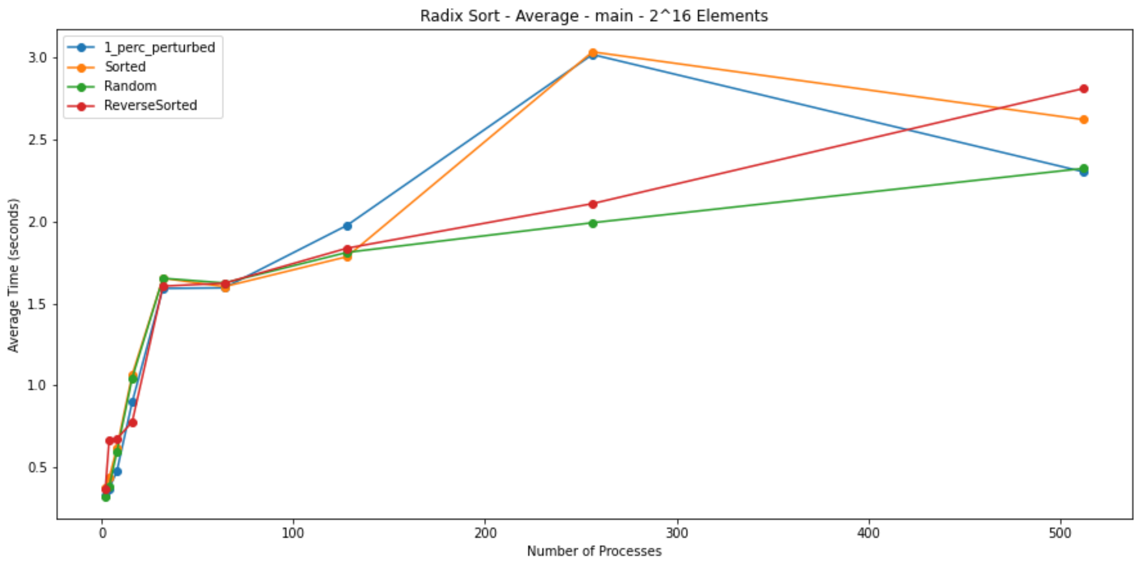


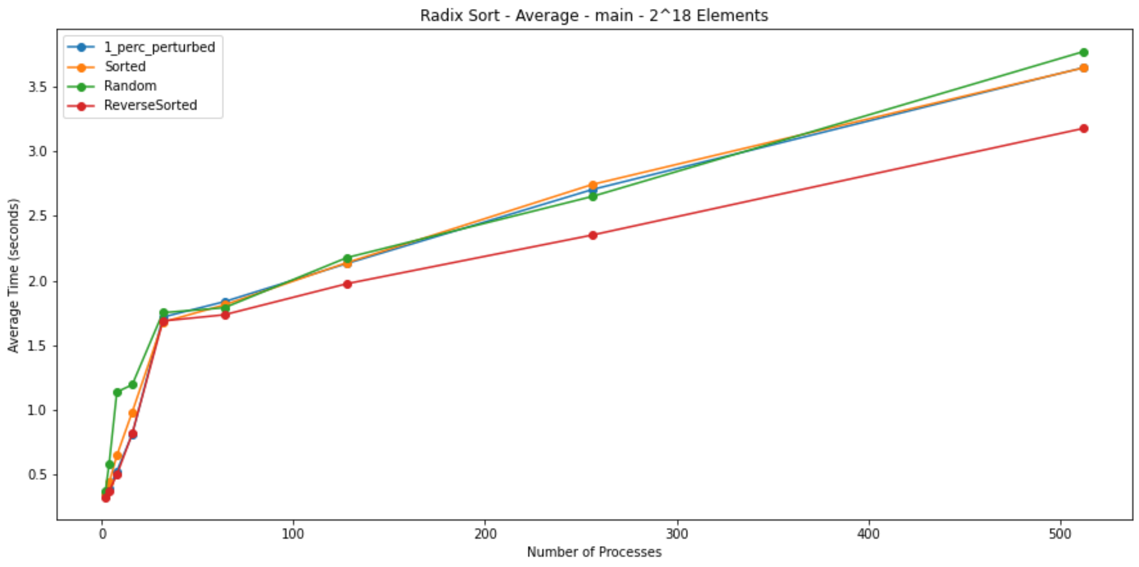
Overall, This benchmark was successful, but would have benefitted from less congestion. We will definitely run our sorts again tomorrow to see if our variance and communication graphs come out less unruly.

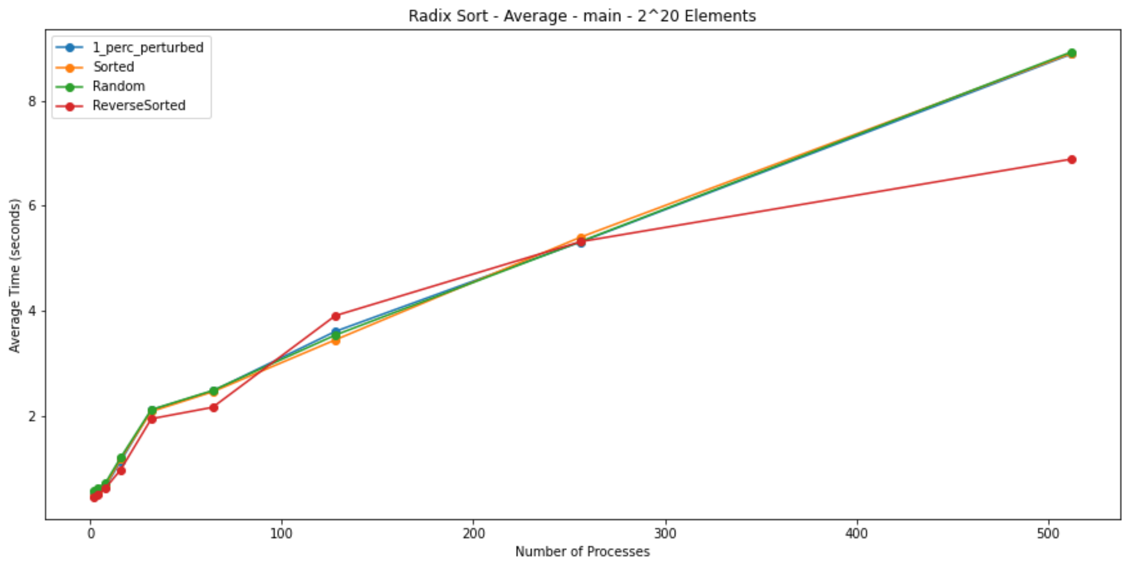
Radix Sort

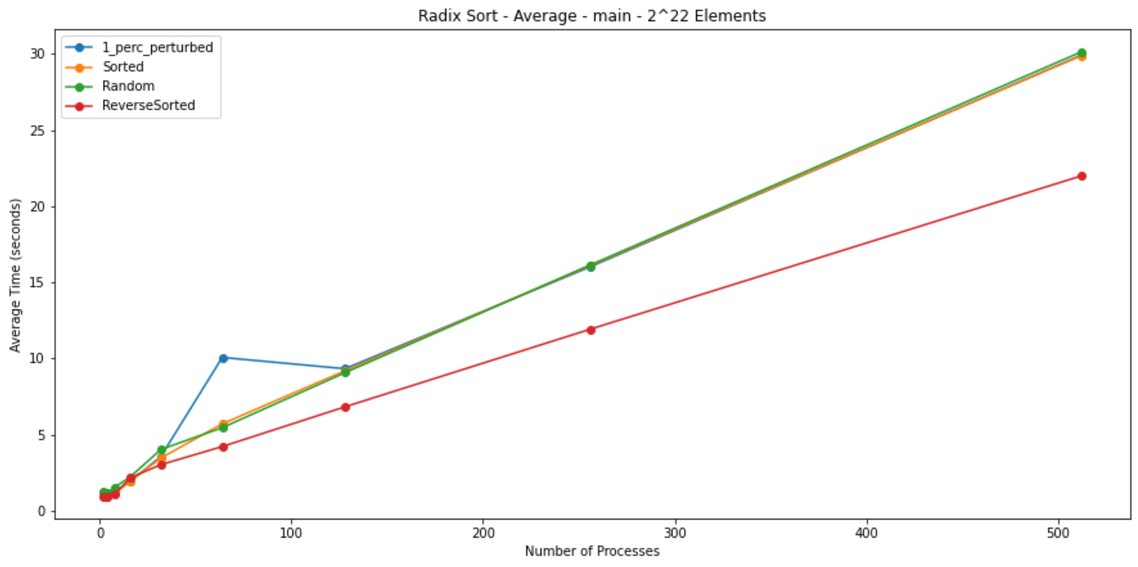
Strong Scale Analysis (Input Sizes, num procs, comp, comm, main), Note: 1024 processes were timing out due to grace resources not being available

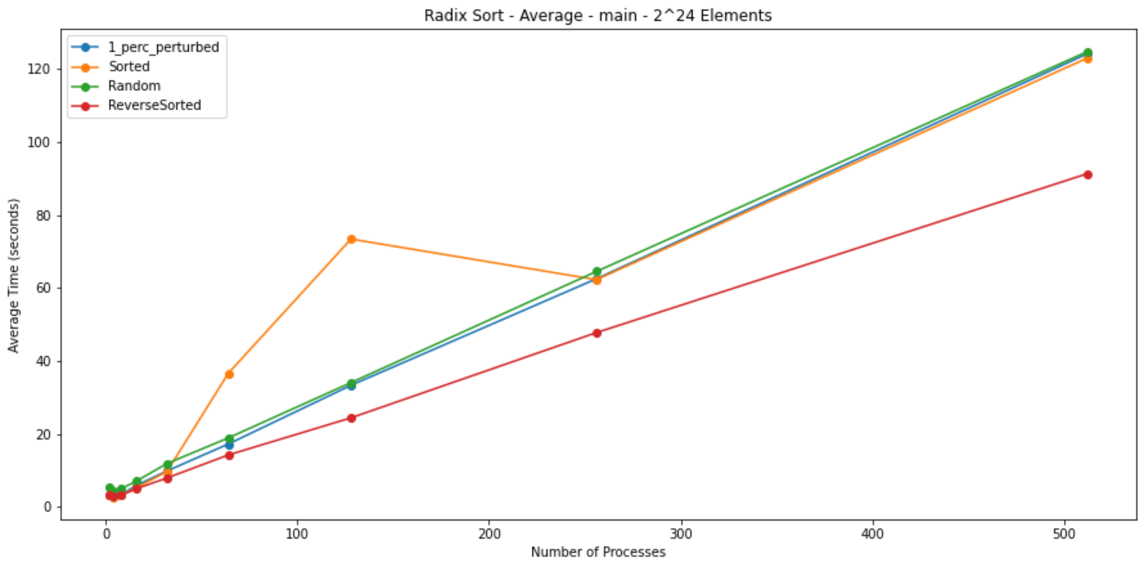
Strong Scale Main

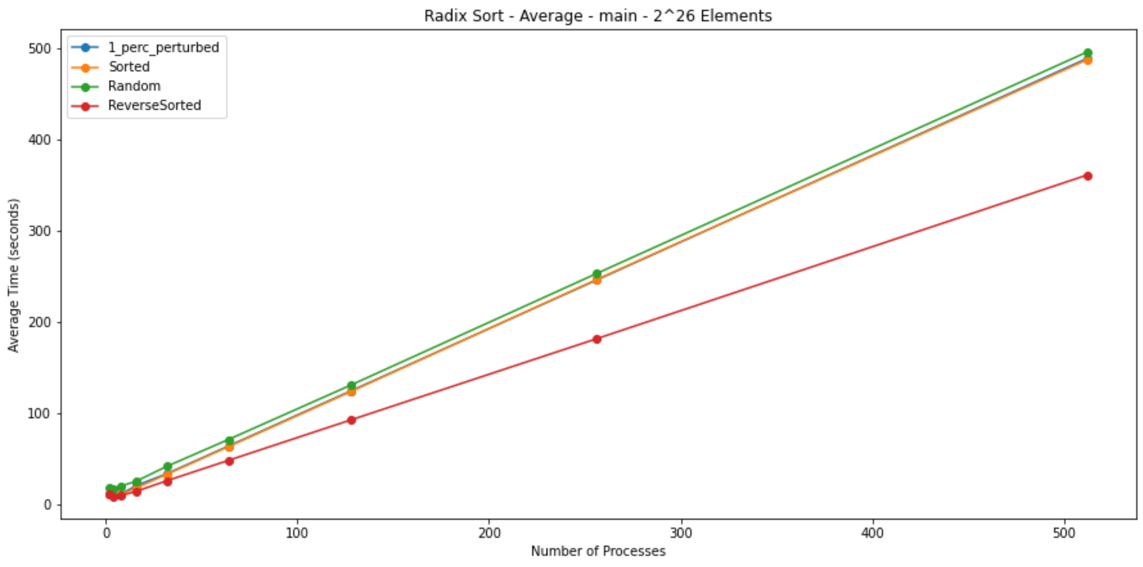

Radix Sort - Average - main - 2^16 Elements
1_perc_perturbed
Sorted
Random
ReverseSorted
Average Time (seconds)
Number of Processes
0
100
200
300
400
500
0.5
1.0
1.5
2.0
2.5
3.0

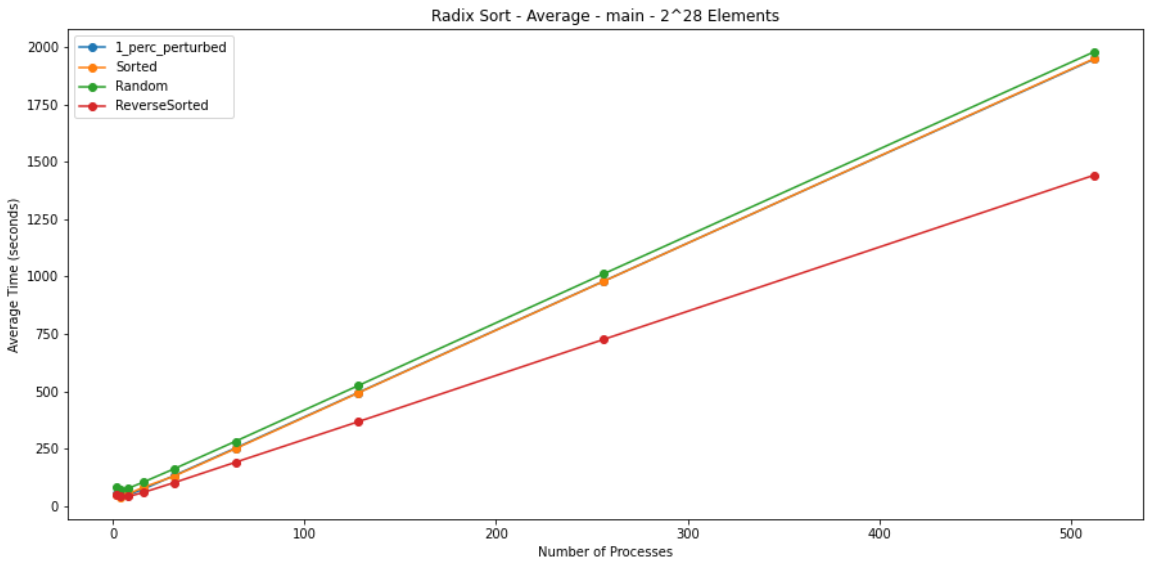

Radix Sort - Average - main - 2^18 Elements
1_perc_perturbed
Sorted
Random
ReverseSorted
Average Time (seconds)
Number of Processes
0
100
200
300
400
500
0.5
1.0
1.5
2.0
2.5
3.0
3.5


Radix Sort - Average - main - 2^20 Elements
1_perc_perturbed
Sorted
Random
ReverseSorted
Average Time (seconds)
Number of Processes
0
100
200
300
400
500
2
4
6
8


Radix Sort - Average - main - 2^22 Elements
1_perc_perturbed
Sorted
Random
ReverseSorted
Average Time (seconds)
Number of Processes


Radix Sort - Average - main - 2^24 Elements
1_perc_perturbed
Sorted
Random
ReverseSorted
Average Time (seconds)
Number of Processes



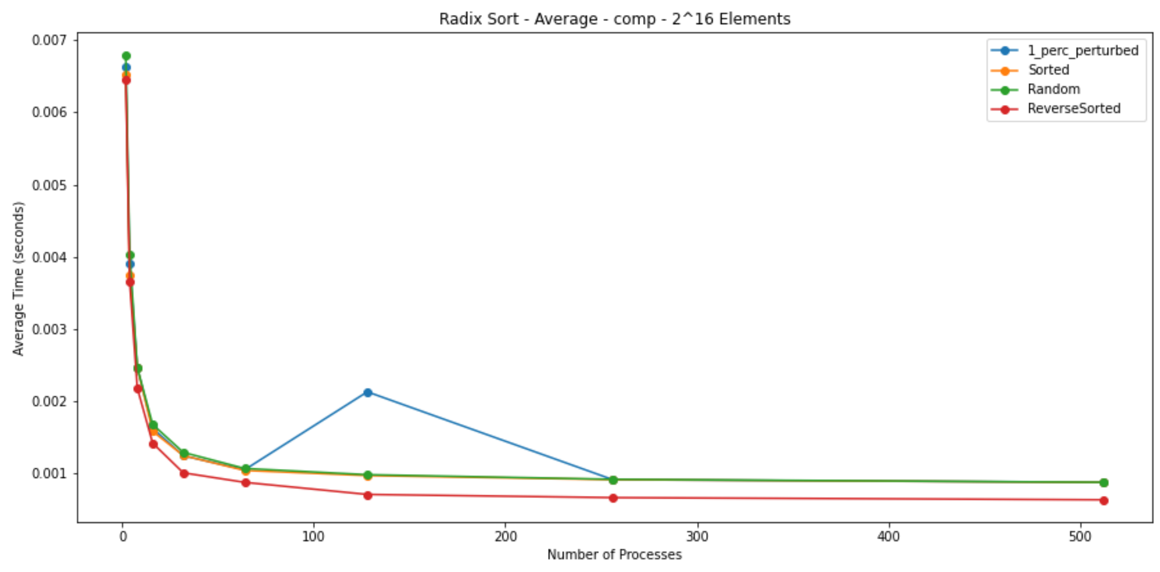


Observations

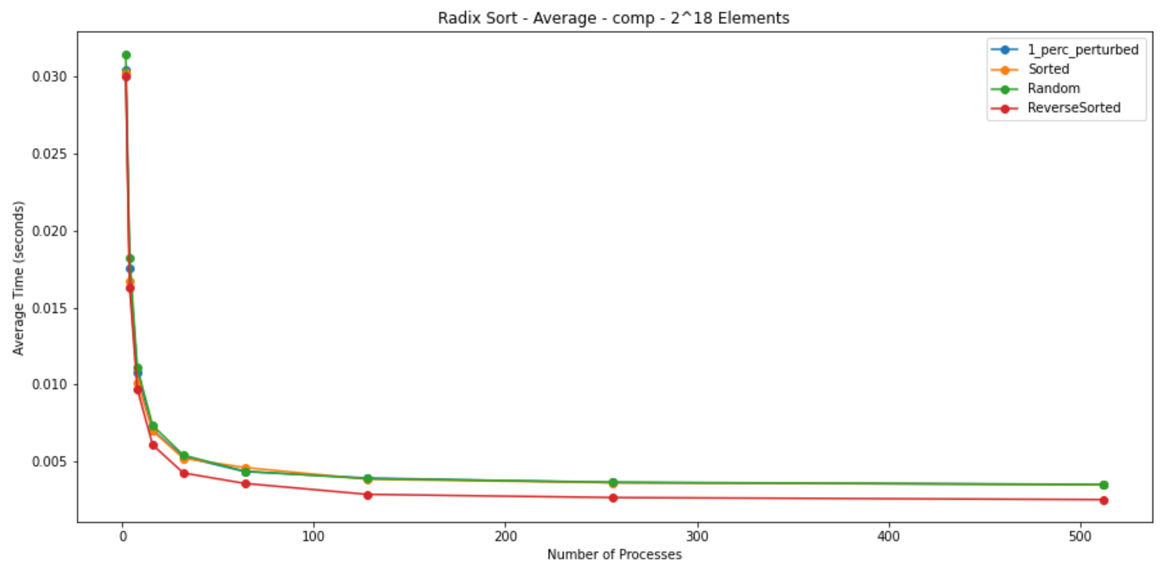
It seems that Across the different input sizes for the Main section time for each size as the number of processes increases, There is an initial increase in time in sizes beofre 2^20, This seems to show the inefficiency of increasing the number of processes past what seems like 32 making it more costly than beneficial as the size does not require that many processes. When we get past 2^20, the increase is more linear and steady. Interestingly enough, Reverse Sorting which should have had the worst results performed quite well through each of the inputs, which suggests a beneficial usefulness to processes in the sorting process for reverse sorting as opposed to the other input_types.

Strong Scale Comp

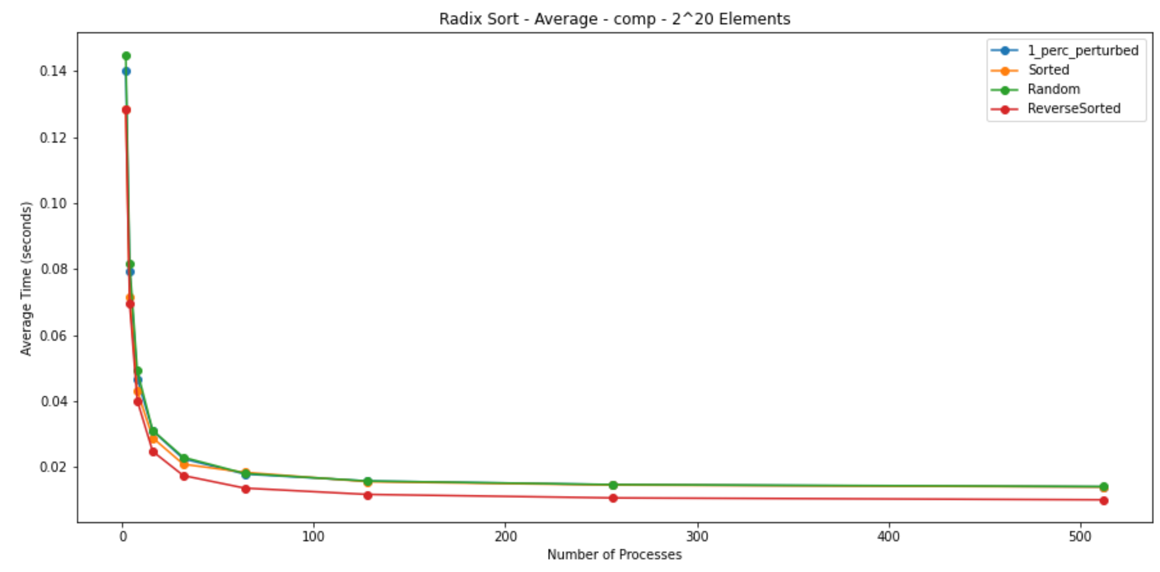
Radix_Average_comp_2^16



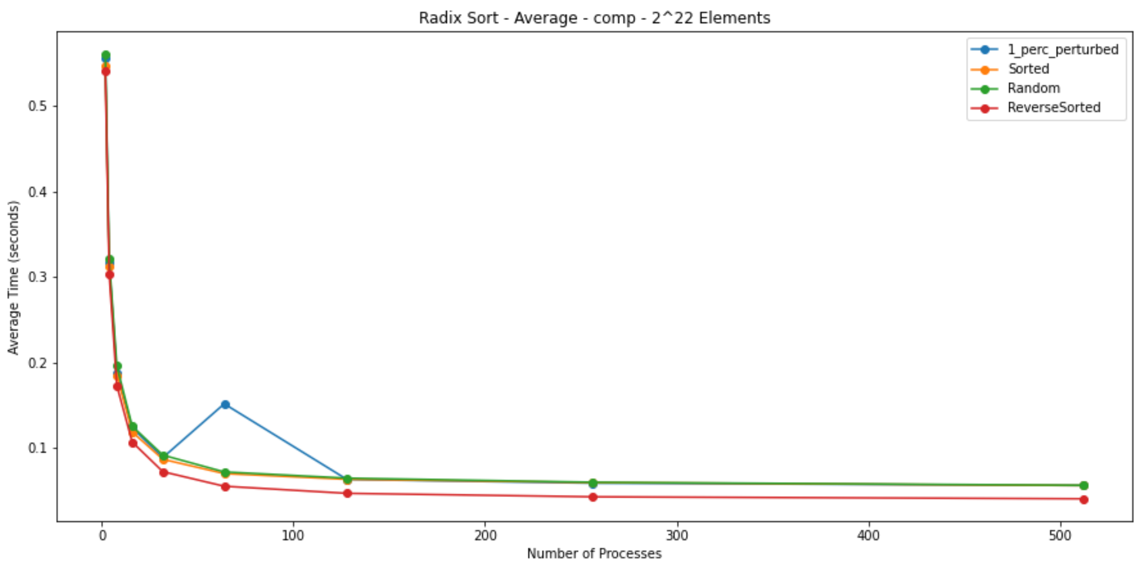
Radix_Average_comp_2^18



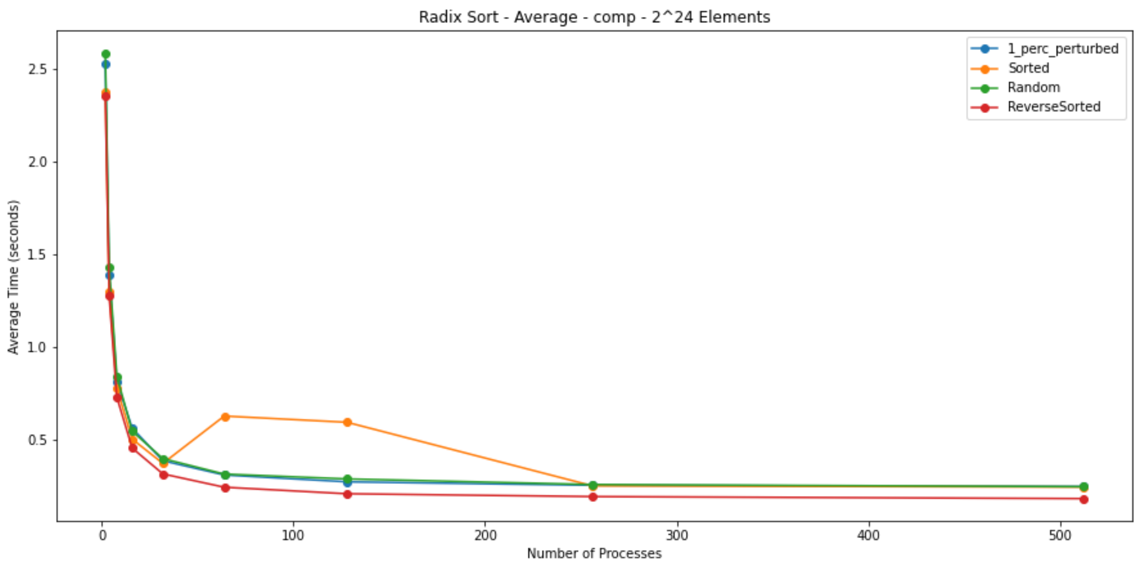
Radix_Average_comp_2^20



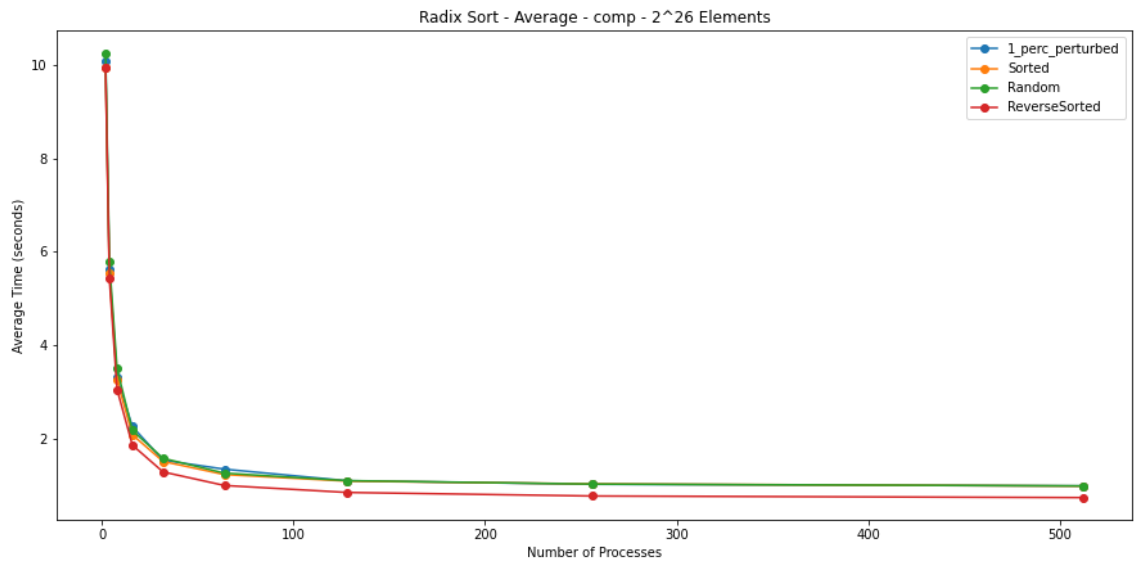
Radix_Average_comp_2^22



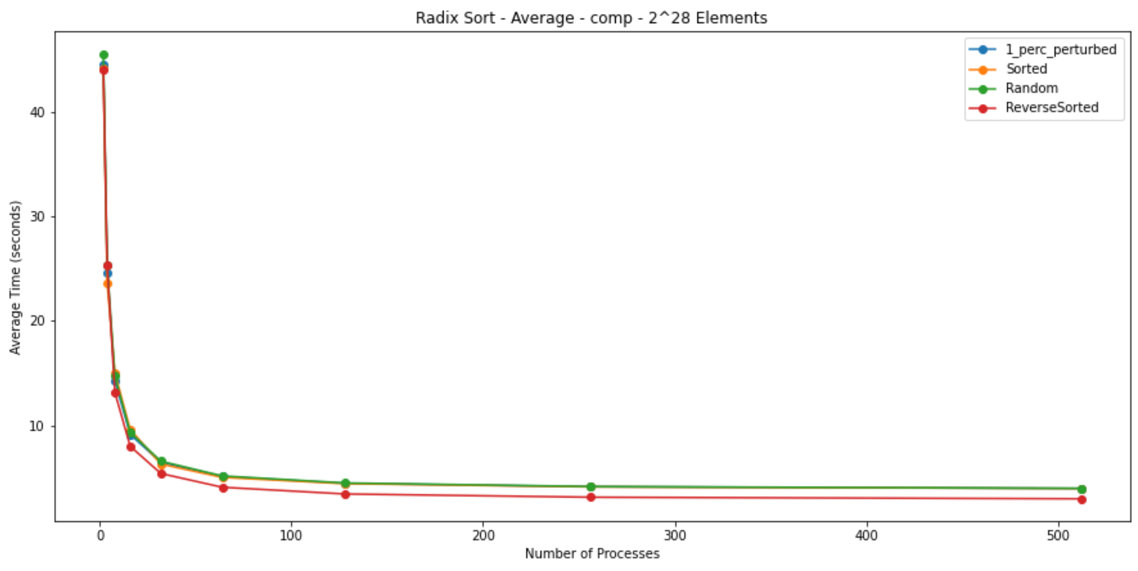
Radix_Average_comp_2^24



Radix_Average_comp_2^26



Radix_Average_comp_2^28

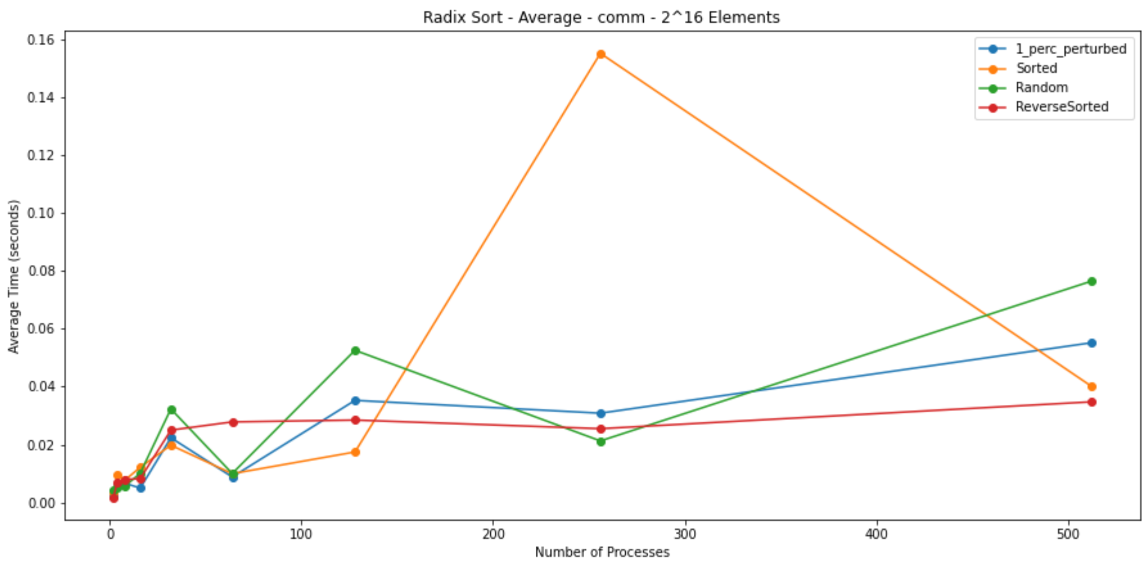


Observations

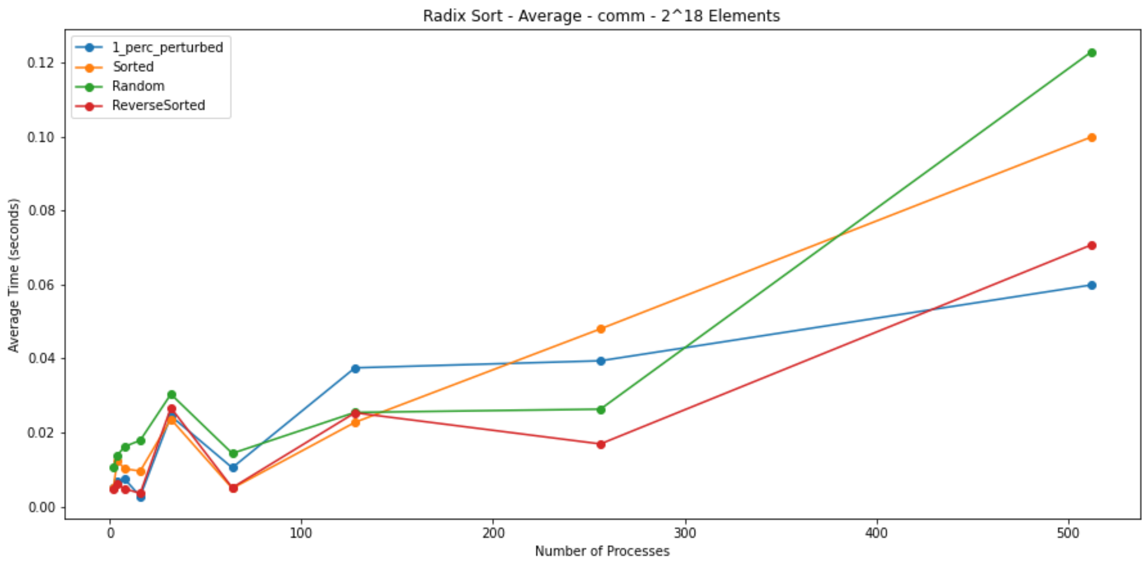
The Computation time for each of the sizes seems to the exponentially decreasing with an increasing number of processes, and for the radix sort this is consistent. We can also notice through strong scaling that for some instances such as 1_perc_perturbed at 2^22 and Sorted at 2^24 have a small amount of spike compared to the other types.

Strong Scale Comm

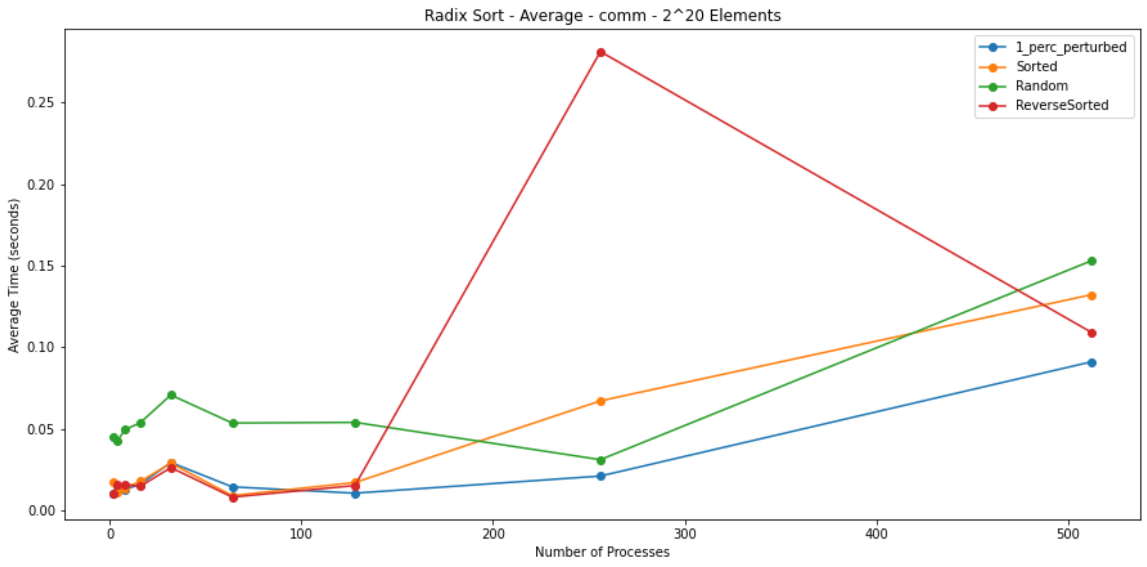
Radix_Average_comm_2^16



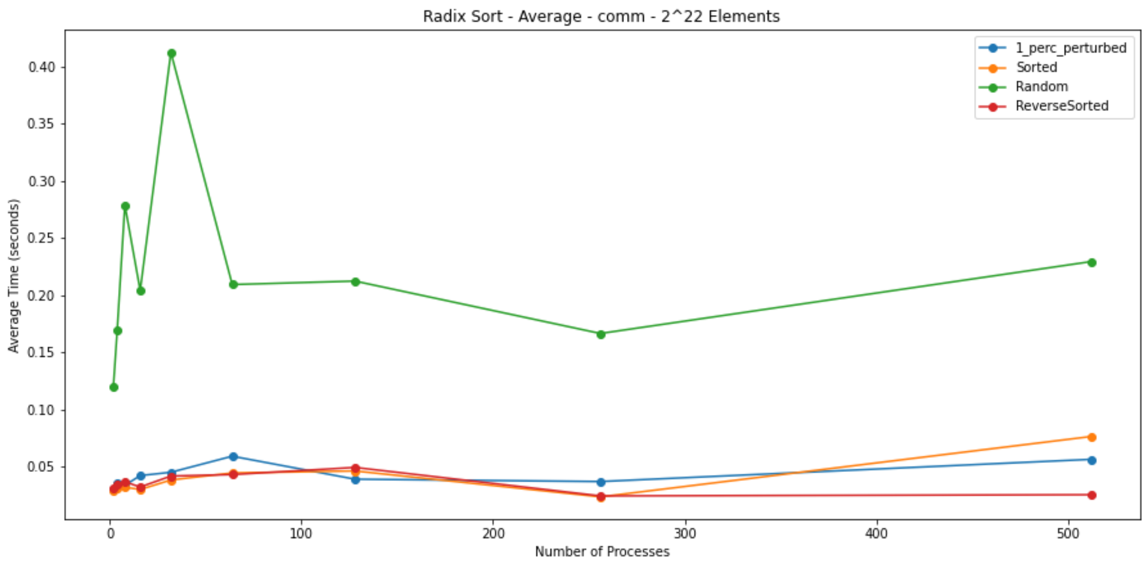
Radix_Average_comm_2^18



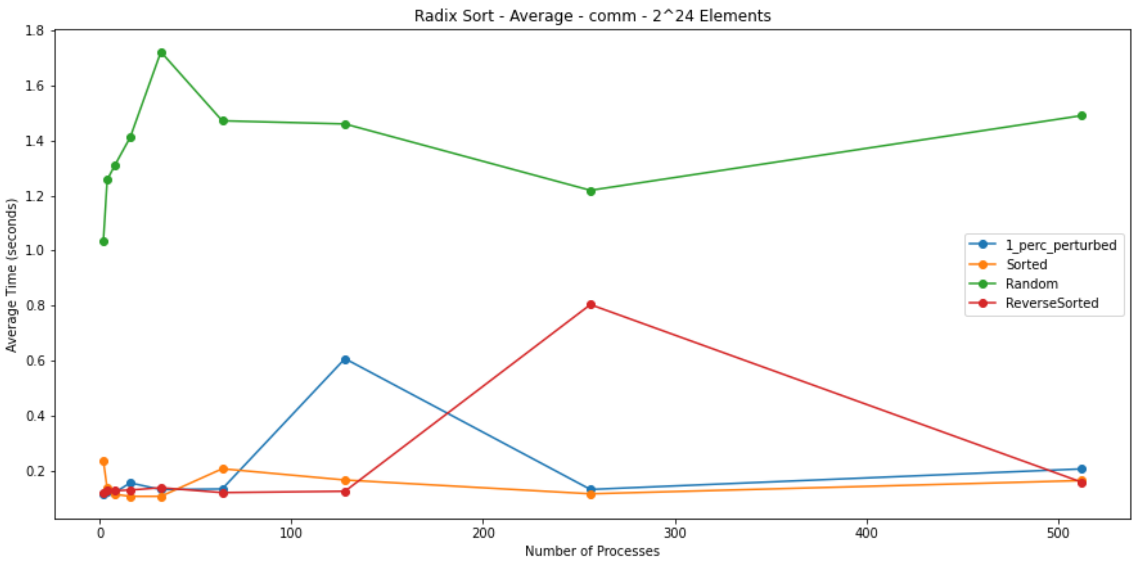
Radix_Average_comm_2^20



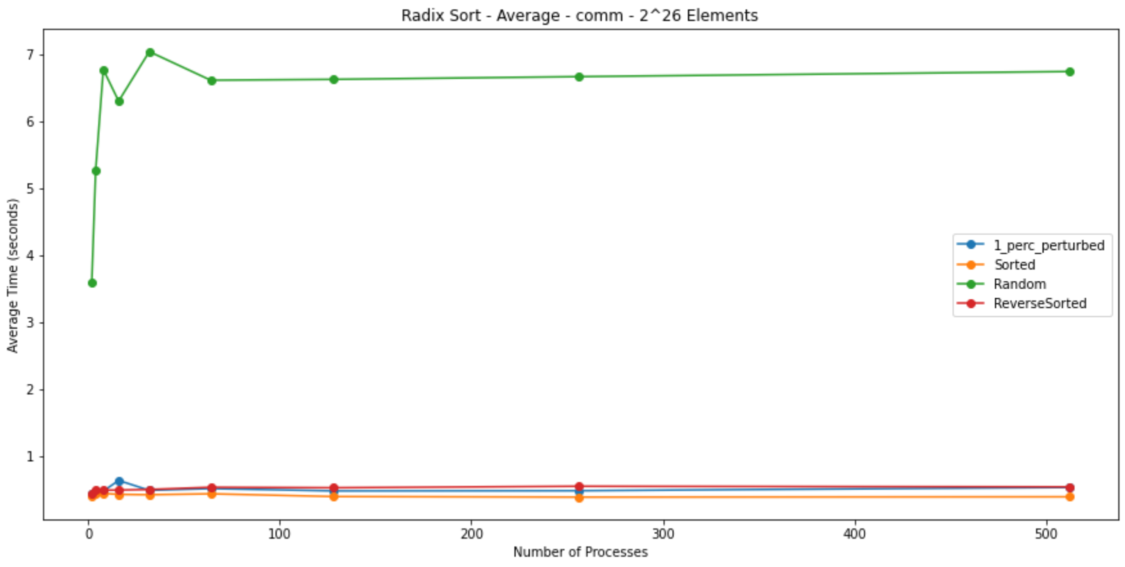
Radix_Average_comm_2^22



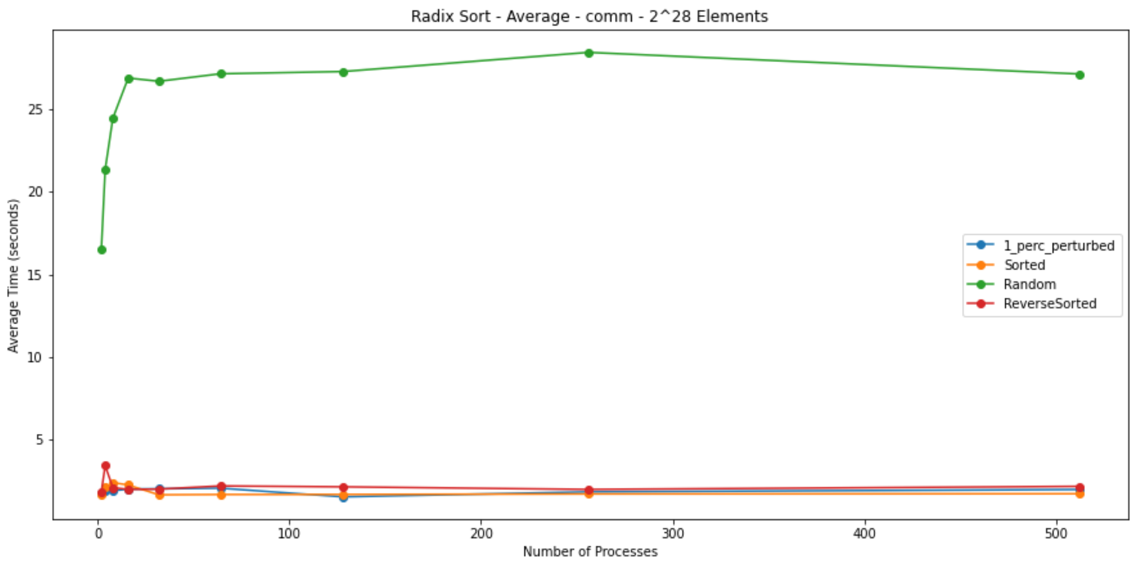
Radix_Average_comm_2^24



Radix_Average_comm_2^26



Radix_Average_comm_2^28



Observations

The communication time for the main section of the radix sort is quite high, especially for the larger input sizes, This is condusive with what we assumed. As the number of processes increases, the time it takes to commuinicate between processes increases. It seems to be an exponential increase as the number of processes increases