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 or 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[ouput_index - array_size] = data1[index1]
        index1++

    while (index2 < array_size):
        output_index = index1 + index2
        upper_half[ouput_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 column_sort(local_data, local_data_size, comm_size, rank)
    Begin whole_column_sort

    // Step 1: Sort the local data
    Call sequential_sort(local_data, local_data_size)

    // Step 2: Transpose the matrix
    Initialize send_buf of size local_data_size
    Calculate subbuf_size as local_data_size / comm_size
    For i from 0 to local_data_size
        Calculate target process as i % comm_size
        Calculate target index as (subbuf_size * target process) + (i / comm_size)
        Place local_data[i] into send_buf[target index]
    Call MPI_Alltoall to redistribute send_buf into local_data using subbuf_size blocks
    Delete send_buf

    // Step 3: Sort the transposed data
    Call sequential_sort(local_data, local_data_size)

    // Step 4: "Untranspose" to restore original structure
    Call MPI_Alltoall with MPI_IN_PLACE to transpose data back in-place using subbuf_size blocks

    // Step 5: Sort the data again
    Call sequential_sort(local_data, local_data_size)

    // Step 6: Shift data to right neighboring process
    Initialize shift_buf of size (local_data_size / 2) * comm_size
    Determine half_local_size_ceil as (local_data_size + 1) / 2
    If rank == comm_size - 1
        Set target_rank to 0
    Else
        Set target_rank to rank + 1
    Calculate offset for filling shift_buf as (local_data_size / 2) * target_rank
    For i from half_local_size_ceil to local_data_size
        Place local_data[i] into shift_buf at offset + (i - half_local_size_ceil)
    Initialize receive_buf of same size as shift_buf
    Call MPI_Alltoall to exchange shift_buf into receive_buf using subbuf_size blocks
    If rank == 0
        Set receive_rank to comm_size - 1
    Else
        Set receive_rank to rank - 1
    Calculate receive offset as receive_rank * (local_data_size / 2)
    For i from half_local_size_ceil to local_data_size
        Update local_data[i] from receive_buf at receive offset + (i - half_local_size_ceil)
    Delete shift_buf and receive_buf

    // Step 7: Sort the data unless it's the first process
    If rank != 0
        Call sequential_sort(local_data, local_data_size)

    // Step 8: Reverse the shift done in step 6
    Reinitialize shift_buf and receive_buf
    Prepare data for reverse shift similar to step 6 but in opposite direction
    Call MPI_Alltoall to exchange data for unshifting
    Update local_data based on received data
    Delete shift_buf and receive_buf

    End whole_column_sort
End Function
```

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 (T1 / Tn)
  - Parallel efficiency ((T1 / Tn) / 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 x 2^{16}, 4 processors: 4 x 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

```
0.009 main
├─ 0.000 data_init_runtime
│  └─ 0.000 data_init_runtime
├─ 0.000 comp
│  ├─ 0.000 comp_large
│  └─ 0.000 comp_small
├─ 0.007 comm
│  ├─ 0.006 comm_small
│  │  └─ 0.006 MPI_Sendrecv
│  └─ 0.001 comm_large
│     ├─ 0.001 MPI_Sendrecv
│     └─ 0.000 MPI_Gather
└─ 0.001 correctness_check
   ├─ 0.001 MPI_Recv
   └─ 0.000 MPI_Send
0.000 MPI_Finalize
0.000 MPI_Initialized
0.000 MPI_Finalized
0.001 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
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-rdmav34.so, /sw/eb/sw/impi/2019.9.304-iccifort-2020.4.304/intel64/libfabric/lib/prov/libpsmx2-fi.so, /lib64/libpsm2.so.2, /sw/eb/sw/impi/2019.9.304-iccifort-2020.4.304/intel64/libfabric/lib/prov/libsockets-fi.so, /sw/eb/sw/impi/2019.9.304-iccifort-2020.4.304/intel64/libfabric/lib/prov/librxm-fi.so, /sw/eb/sw/impi/2019.9.304-iccifort-2020.4.304/intel64/libfabric/lib/prov/libtcp-fi.so, /usr/lib64/ucx/libuct_ib.so.0, /usr/lib64/ucx/libuct_rdmacm.so.0, /usr/lib64/ucx/libuct_cma.so.0, /usr/lib64/ucx/libuct_knem.so.0, /usr/lib64/ucx/libuct_xpmem.so.0, /usr/lib64/libxpmem.so.0
cmdline	[./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
scalability	strong
group_num	3
implementation_source	handwritten

Merge Sort Metadata

Metadata Key	Value
cali.caliper.version	2.11.0

Metadata Key	Value
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	2
spot.format.version	time.variance,profile.mpi,node.order,region.count,time.exclusive
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	1729408207
spot:time.variance	[/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/11.8.0/extras/CUPTI/lib64/libcupti.so.11.8, /sw/eb/sw/PAPI/6.0.0-GCCcore-8.3.0/lib/libpapi.so.6, /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
launchdate	[./main, 2, 2, 4194304]
libraries	c
cmdline	merge
cluster	mpi
algorithm	int
programming_model	4
data_type	4194304
size_of_data_type	Random
input_size	32
input_type	strong
num_procs	3
scalability	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

Metadata Key	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.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

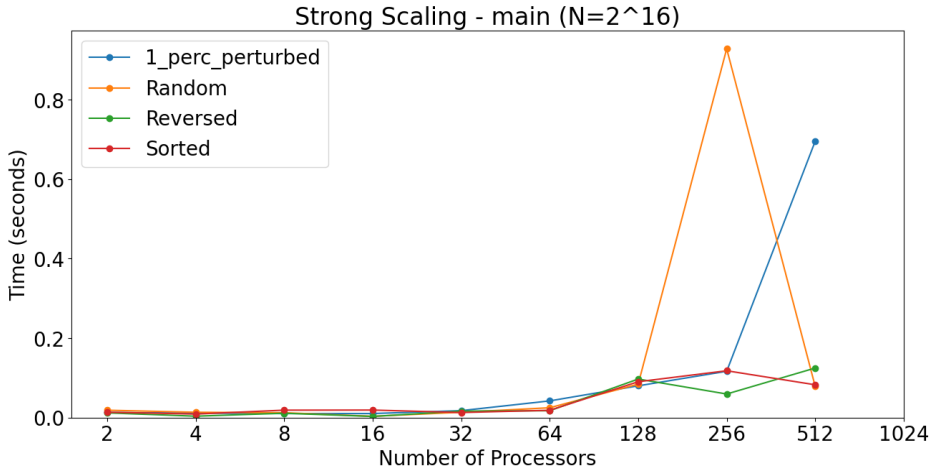
Column Sort Metadata

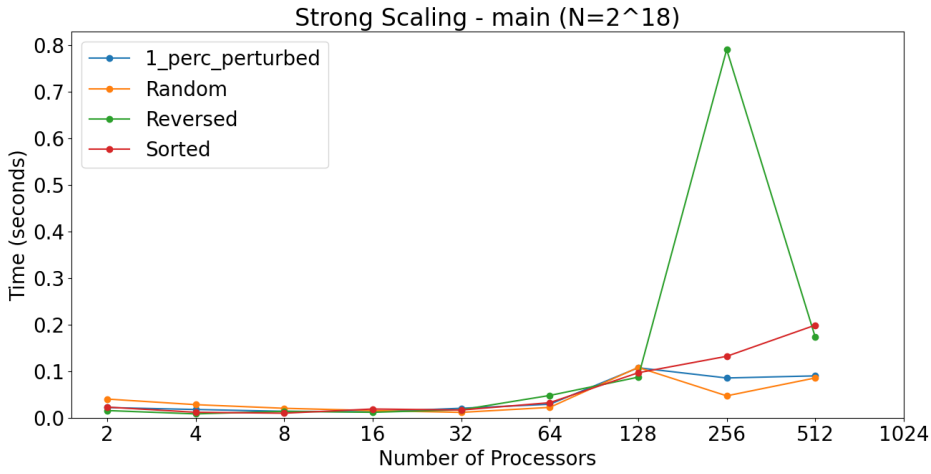
Key	Value
cali.caliper.version	2.11.0
mpi.world.size	32
spot.metrics	min#inclusive#sum#time.duration,max#inclusive#sum#time.duration,avg#inclusive#sum#time.duration,sum#inclusive#sum#time.duration,v
spot.timeseries.metrics	
spot.format.version	2
spot.options	time.variance,profile.mpi,node.order,region.count,time.exclusive
spot.channels	regionprofile
cali.channel	spot
spot:node.order	true
spot:output	p32-a4194304.cali
spot:profile.mpi	true
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/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, 4, 3, 4194304]
cluster	c
algorithm	column
programming_model	mpi
data_type	int
size_of_data_type	4
input_size	4194304
input_type	ReverseSorted

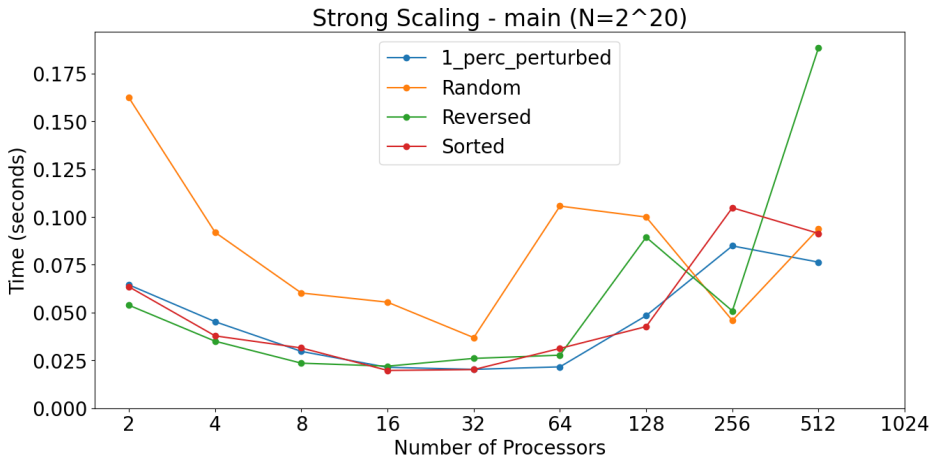
Key	Value
num_procs	32
scalability	strong
group_num	3
implementation_source	handwritten

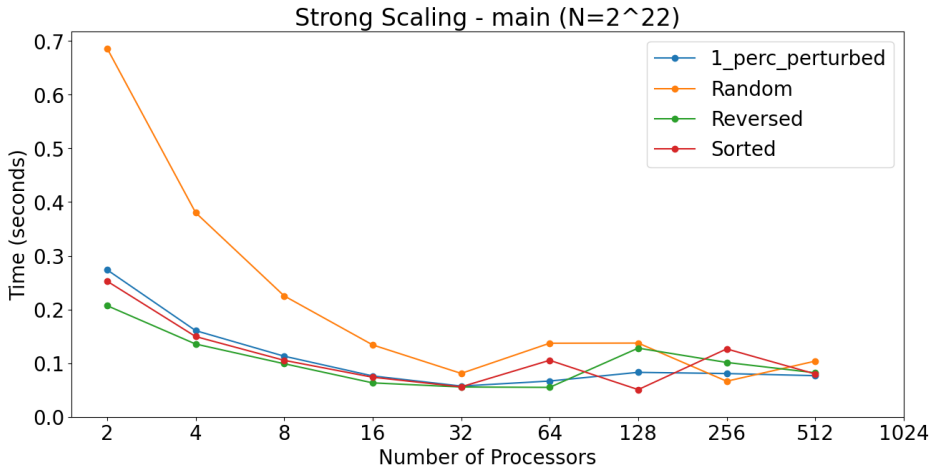
4. Performance evaluation

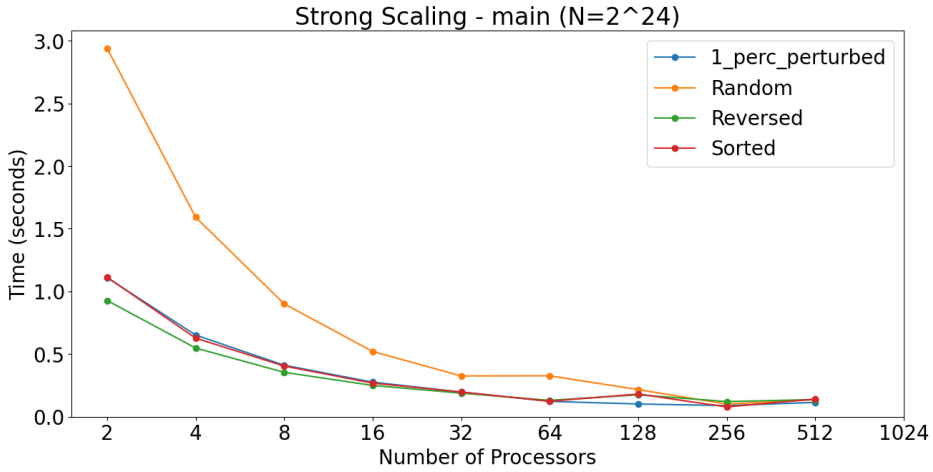
Bitonic Sort

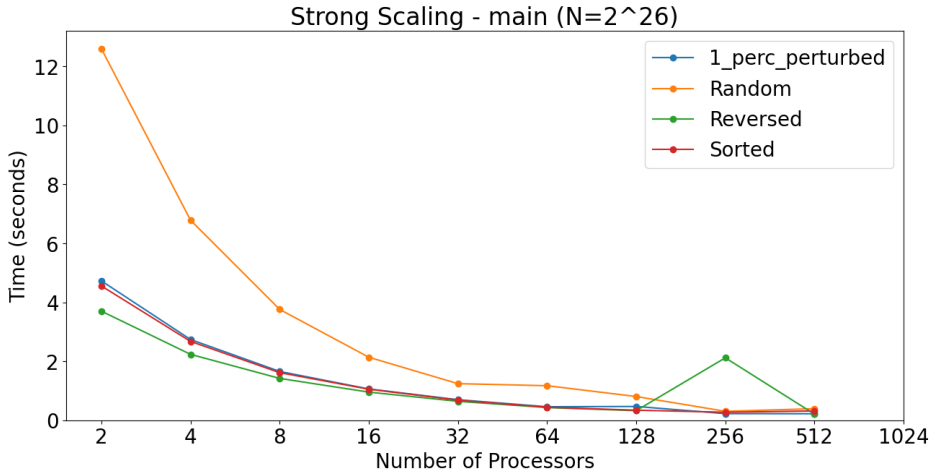


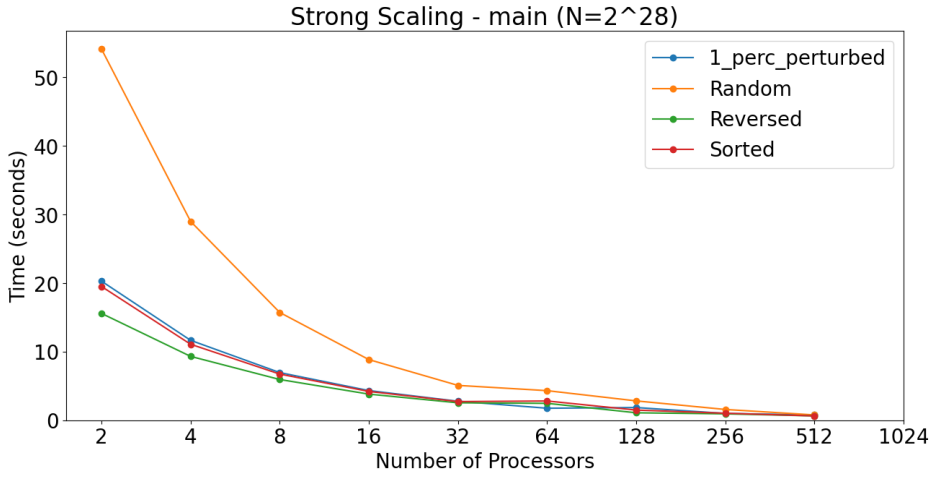


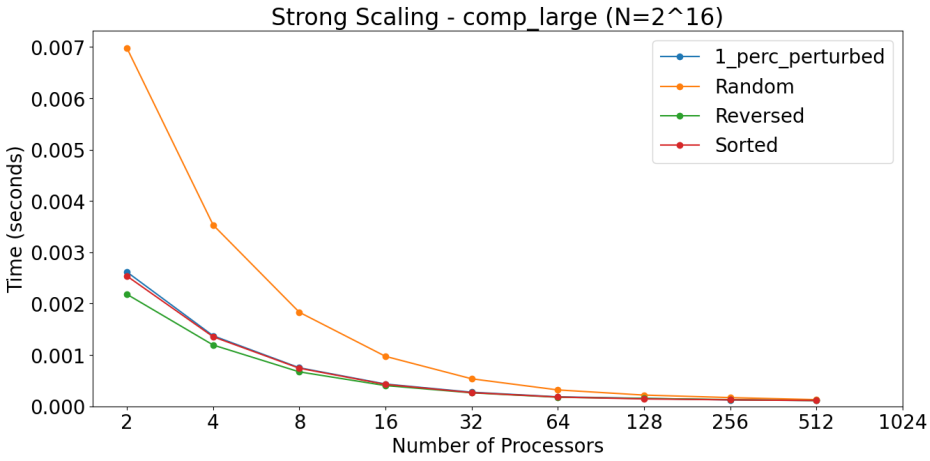


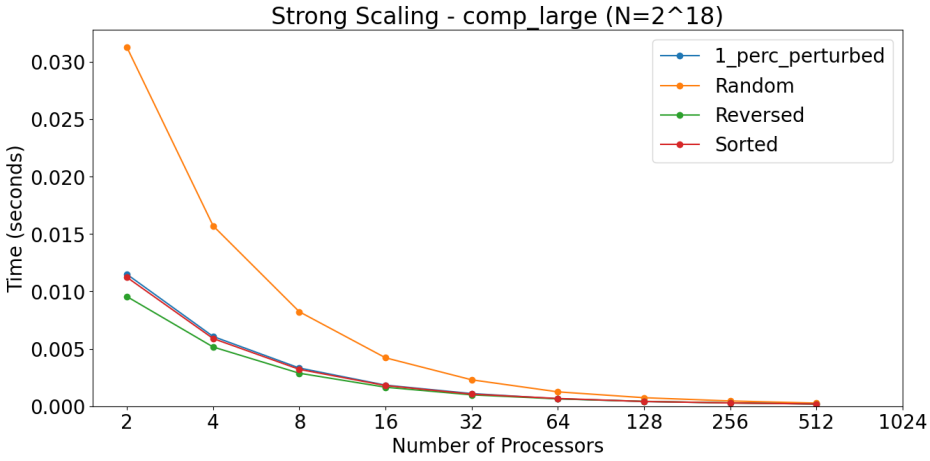

Strong Scaling - main (N=2^22)
Time (seconds)
Number of Processors
1_perc_perturbed
Random
Reversed
Sorted

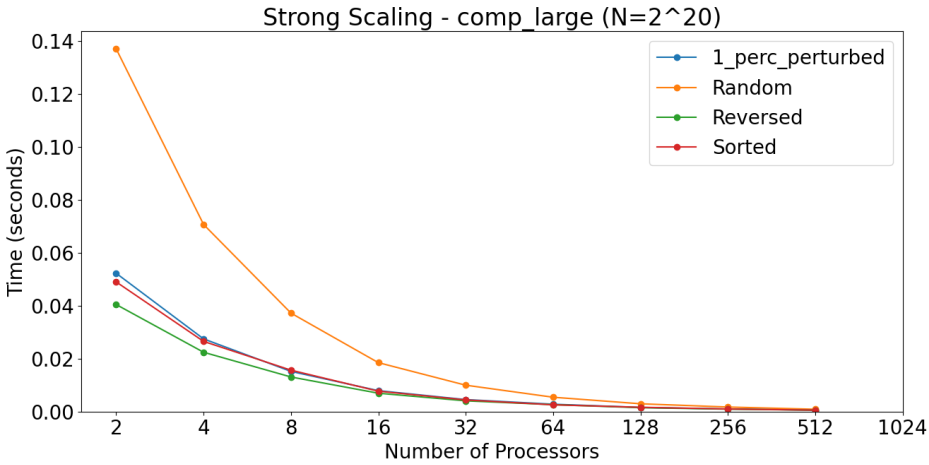

Strong Scaling - main (N=2^24)
Time (seconds)
Number of Processors
1_perc_perturbed
Random
Reversed
Sorted

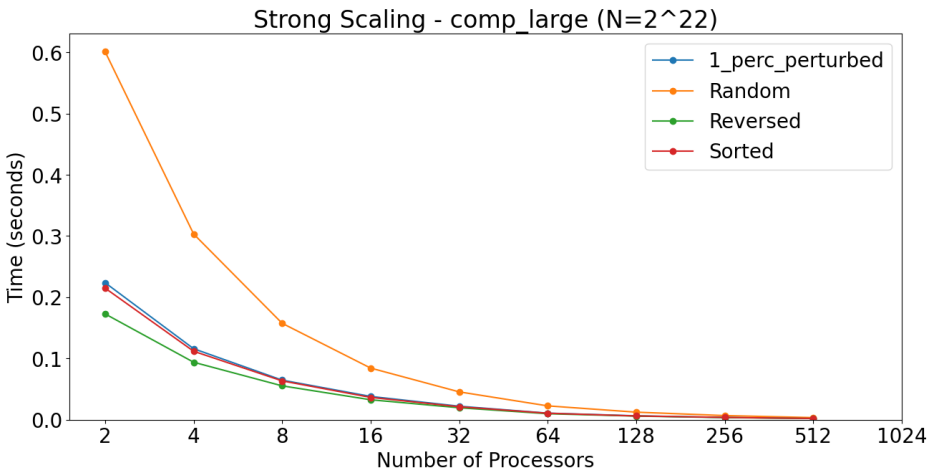

Strong Scaling - main (N=2^26)
Time (seconds)
Number of Processors
1_perc_perturbed
Random
Reversed
Sorted

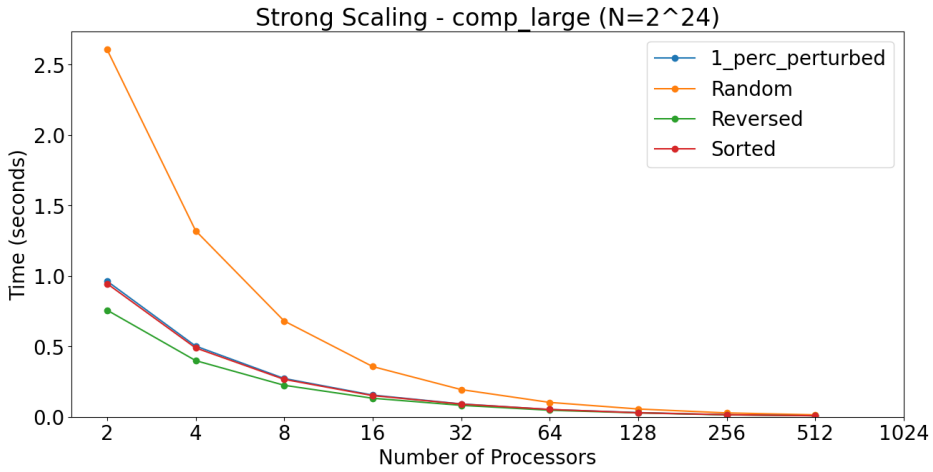

Strong Scaling - main (N=2^28)
Time (seconds)
Number of Processors
1_perc_perturbed
Random
Reversed
Sorted

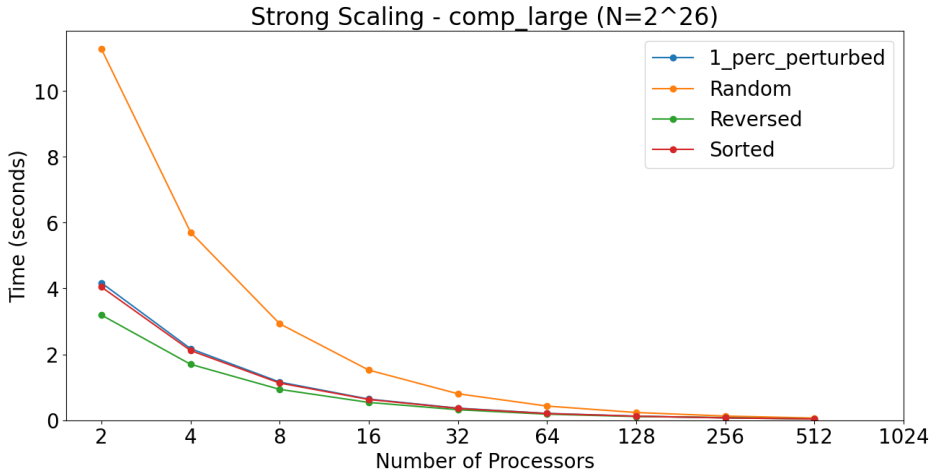

Strong Scaling - comp_large (N=2^16)
Time (seconds)
Number of Processors
1_perc_perturbed
Random
Reversed
Sorted

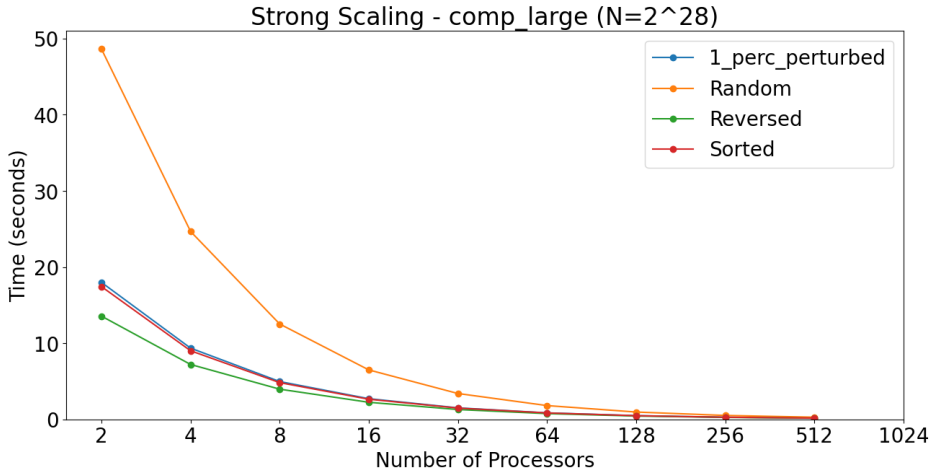

Strong Scaling - comp_large (N=2^18)
Time (seconds)
Number of Processors
1_perc_perturbed
Random
Reversed
Sorted

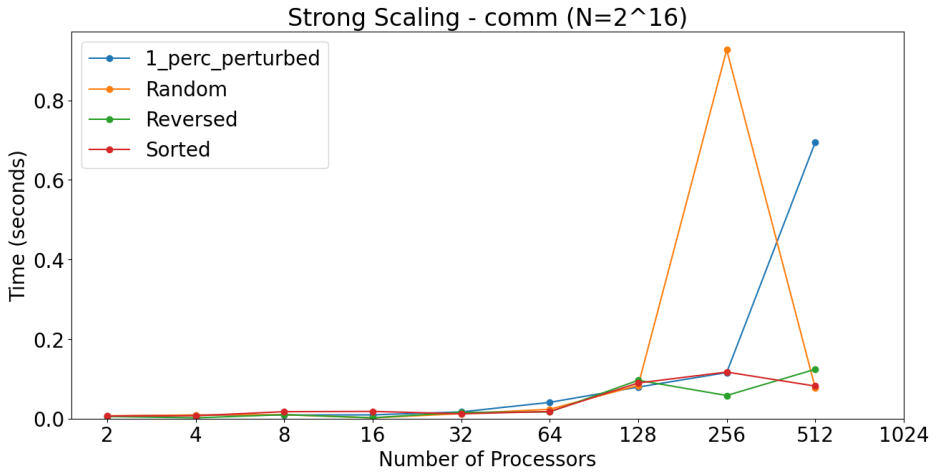

Strong Scaling - comp_large (N=2^20)
Time (seconds)
Number of Processors
1_perc_perturbed
Random
Reversed
Sorted

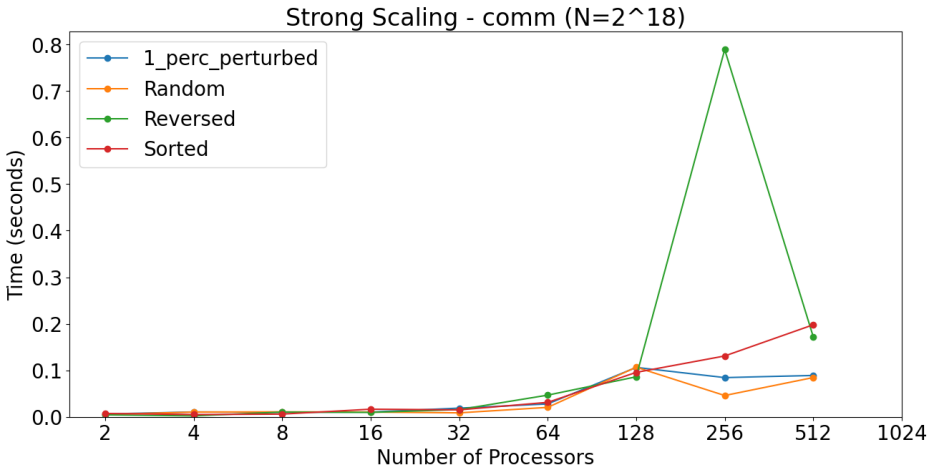

Strong Scaling - comp_large (N=2^22)
Time (seconds)
Number of Processors
1_perc_perturbed
Random
Reversed
Sorted

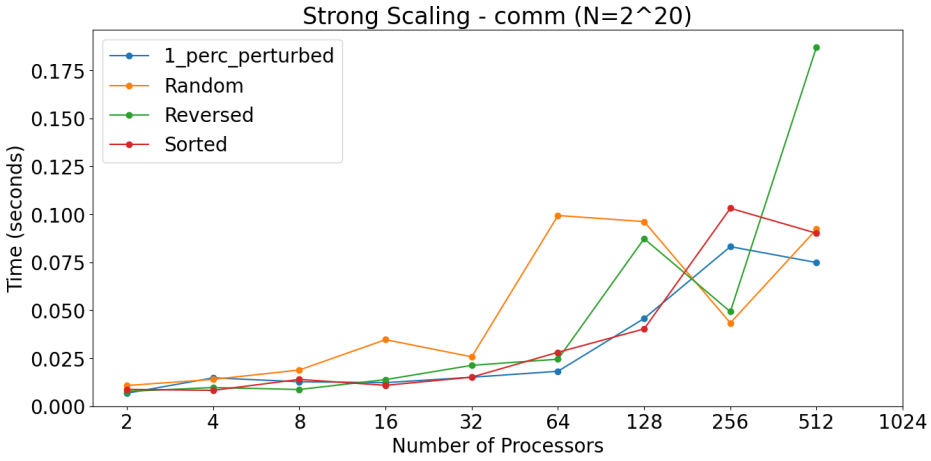

Strong Scaling - comp_large (N=2^24)
1_perc_perturbed
Random
Reversed
Sorted
Time (seconds)
Number of Processors
0.0
0.5
1.0
1.5
2.0
2.5
2
4
8
16
32
64
128
256
512
1024

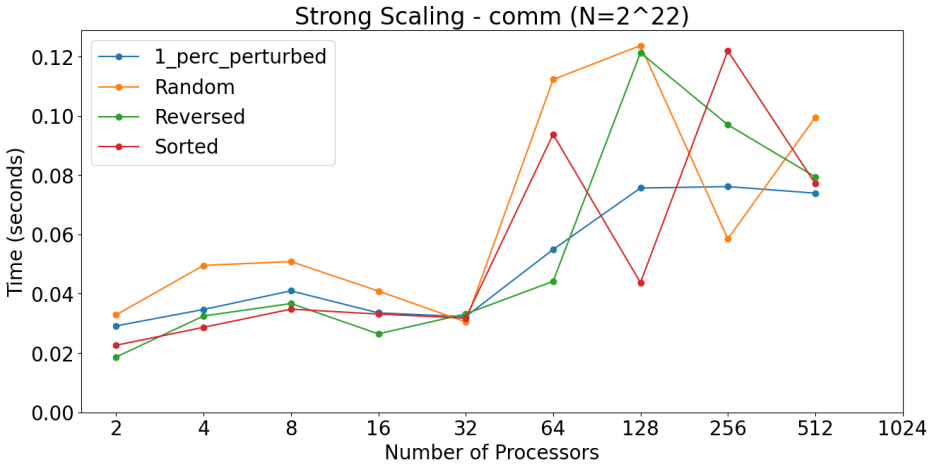

Strong Scaling - comp_large (N=2^26)
1_perc_perturbed
Random
Reversed
Sorted
Time (seconds)
Number of Processors
0
2
4
6
8
10
2
4
8
16
32
64
128
256
512
1024

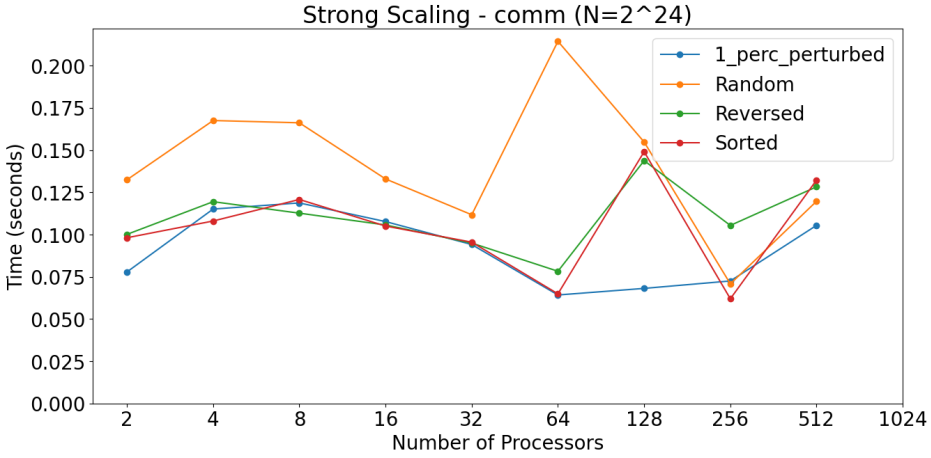

Strong Scaling - comp_large (N=2^28)
1_perc_perturbed
Random
Reversed
Sorted
Time (seconds)
Number of Processors
0
10
20
30
40
50
2
4
8
16
32
64
128
256
512
1024

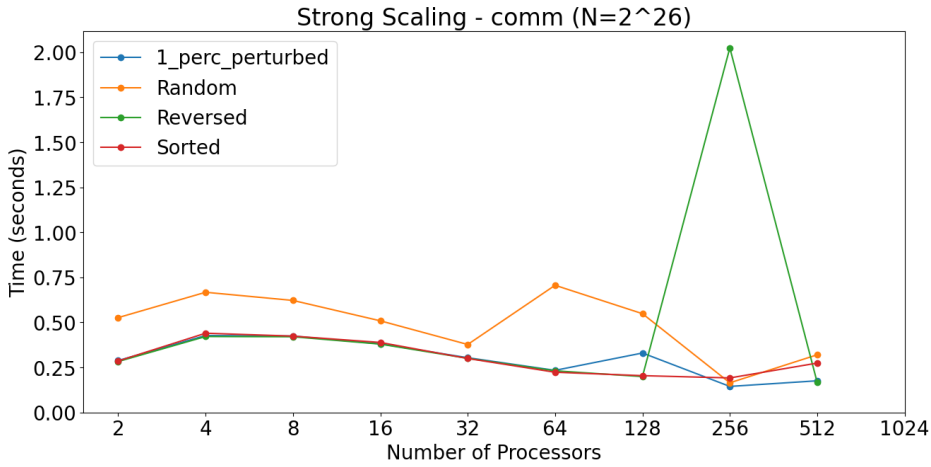

Strong Scaling - comm (N=2^16)
1_perc_perturbed
Random
Reversed
Sorted
Time (seconds)
Number of Processors
0.0
0.2
0.4
0.6
0.8
2
4
8
16
32
64
128
256
512
1024

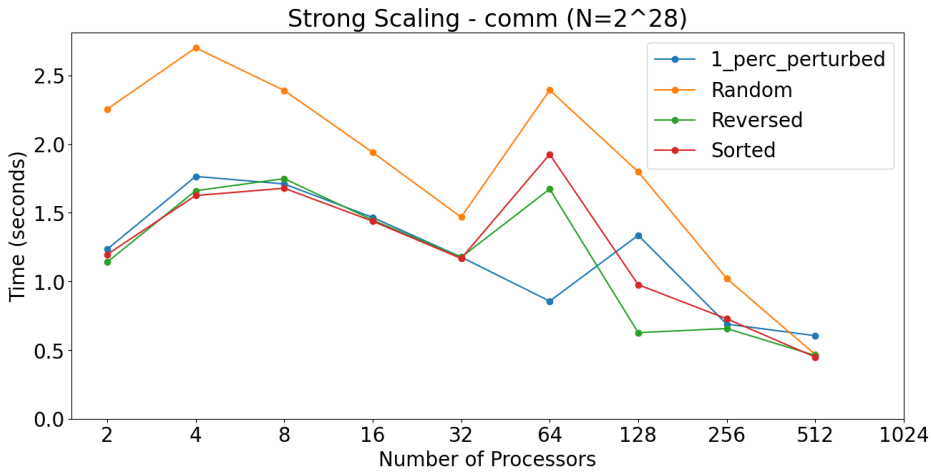

Strong Scaling - comm (N=2^18)
1_perc_perturbed
Random
Reversed
Sorted
Time (seconds)
Number of Processors


Strong Scaling - comm (N=2^20)
1_perc_perturbed
Random
Reversed
Sorted
Time (seconds)
Number of Processors

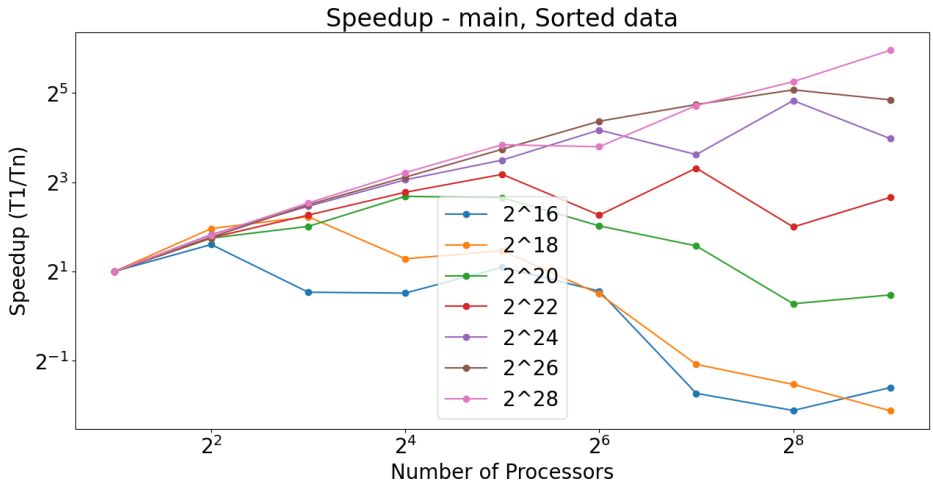

Strong Scaling - comm (N=2^22)
1_perc_perturbed
Random
Reversed
Sorted
Time (seconds)
Number of Processors

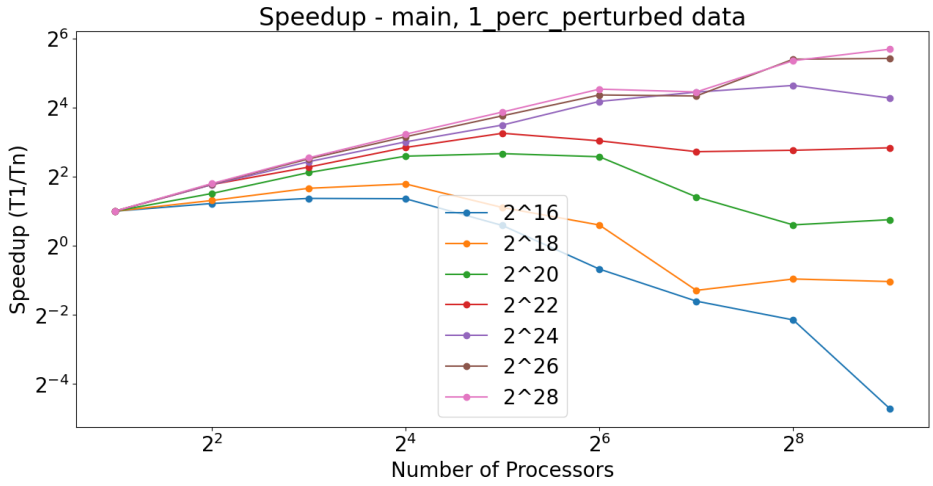

Strong Scaling - comm (N=2^24)
1_perc_perturbed
Random
Reversed
Sorted
Time (seconds)
Number of Processors

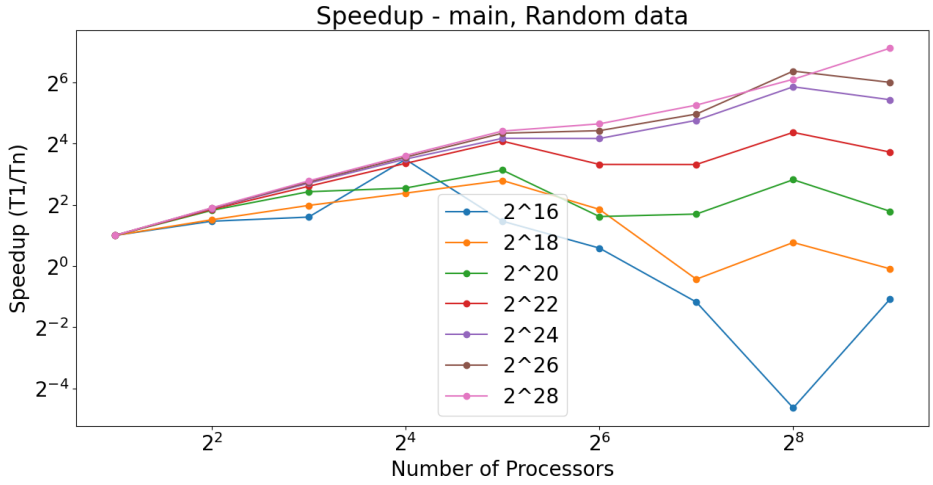


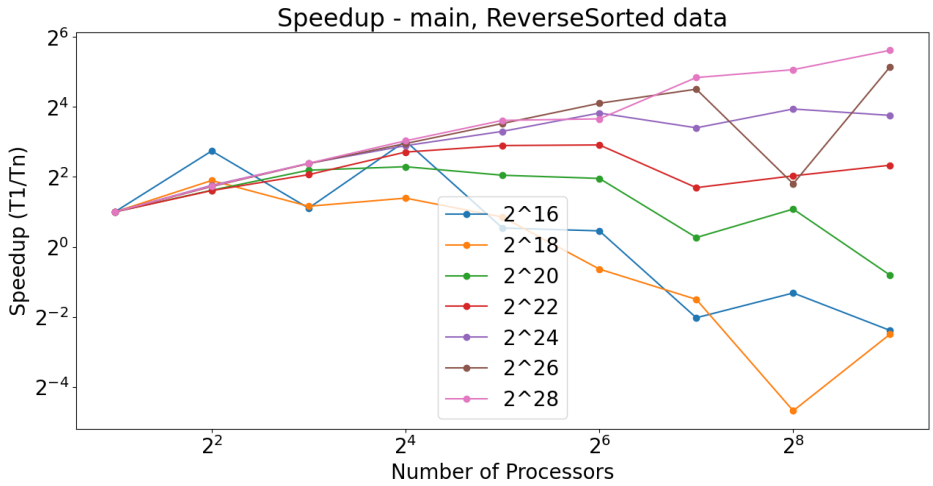


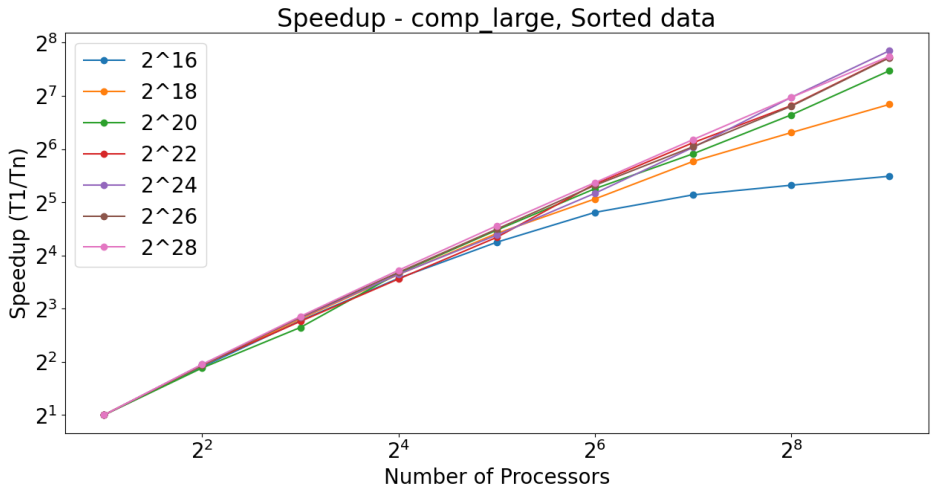
For the small input sizes, adding more processes hurts the runtime because the overhead of communicating outweighs the benefits of added parallelism. This is especially true because the bitonic sort algorithm has more steps when there are more processors. But for the large input sizes, the runtime decreases as we would expect from a parallel algorithm.

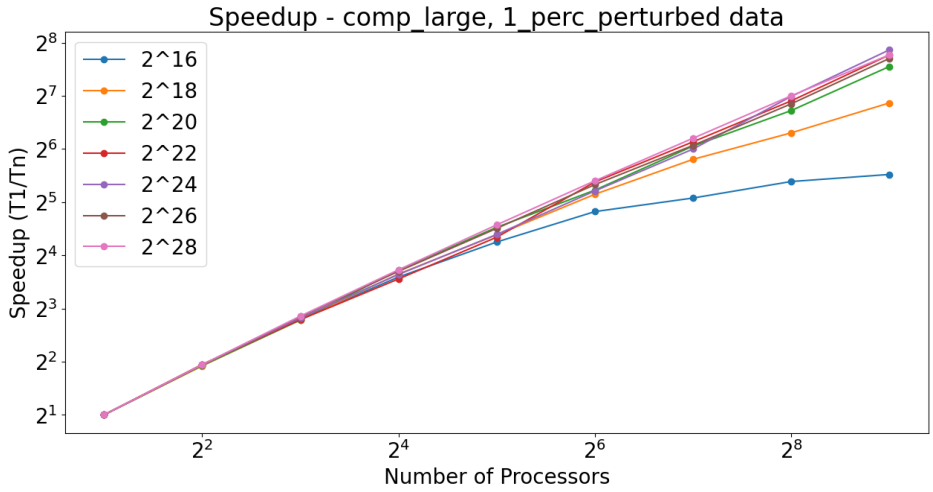

Speedup - main, Sorted data
Speedup (T1/Tn)
Number of Processors
2^16
2^18
2^20
2^22
2^24
2^26
2^28

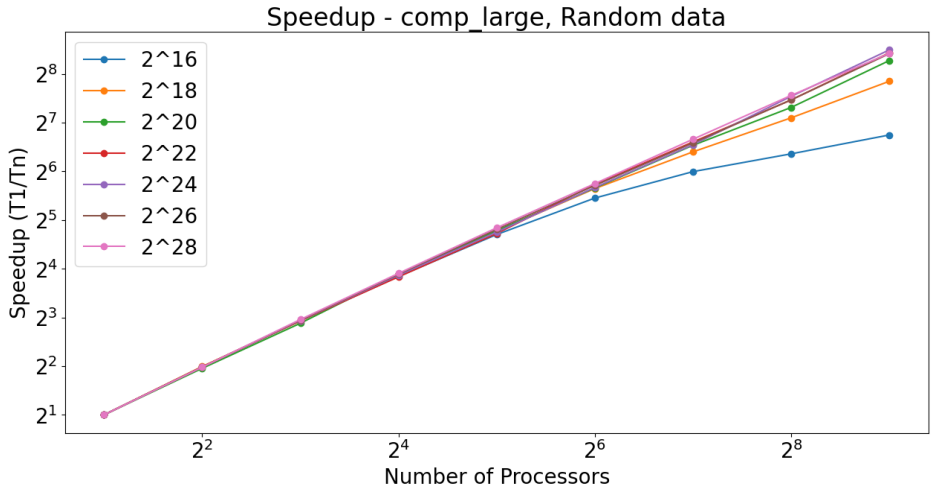

Speedup - main, 1_perc_perturbed data
Speedup (T1/Tn)
Number of Processors
2^16
2^18
2^20
2^22
2^24
2^26
2^28

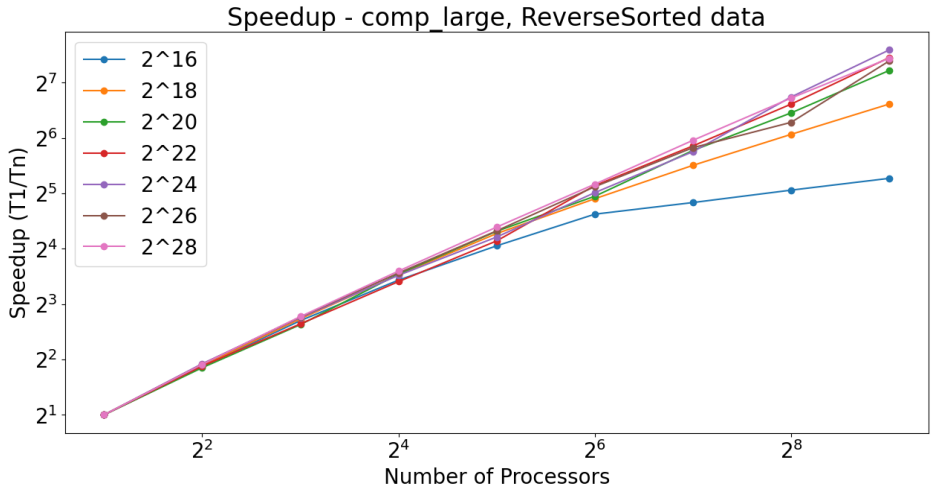

Speedup - main, Random data
Speedup (T1/Tn)
Number of Processors
2^16
2^18
2^20
2^22
2^24
2^26
2^28

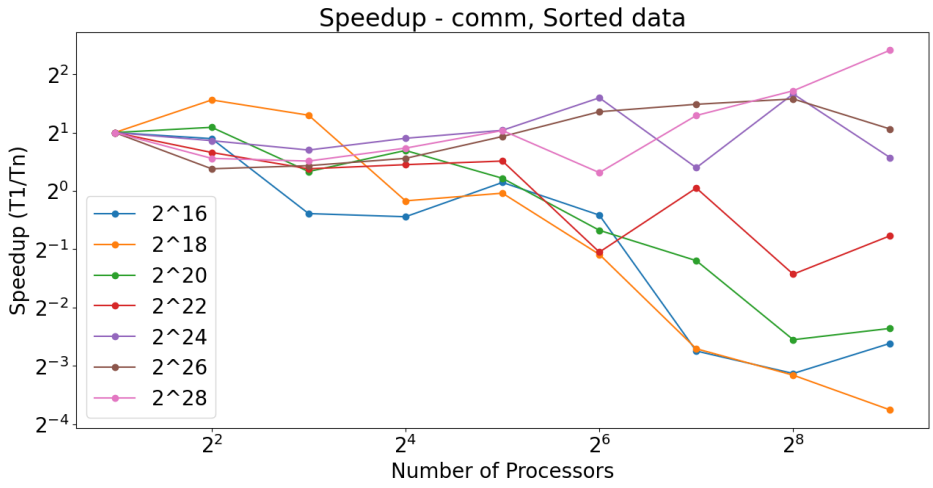

Speedup - main, ReverseSorted data
Speedup (T1/Tn)
Number of Processors
2^16
2^18
2^20
2^22
2^24
2^26
2^28

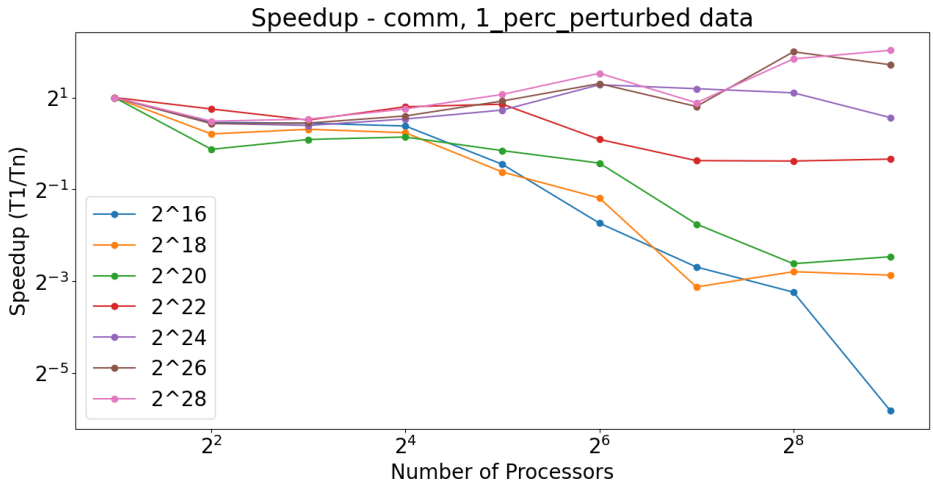

Speedup - comp_large, Sorted data
Speedup (T1/Tn)
Number of Processors
2^16
2^18
2^20
2^22
2^24
2^26
2^28

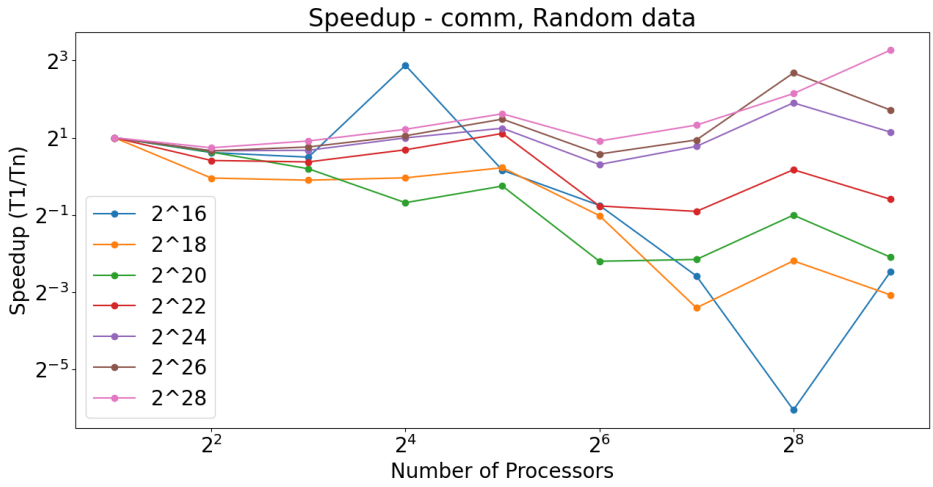

Speedup - comp_large, 1_perc_perturbed data
Speedup (T1/Tn)
Number of Processors
2^16
2^18
2^20
2^22
2^24
2^26
2^28

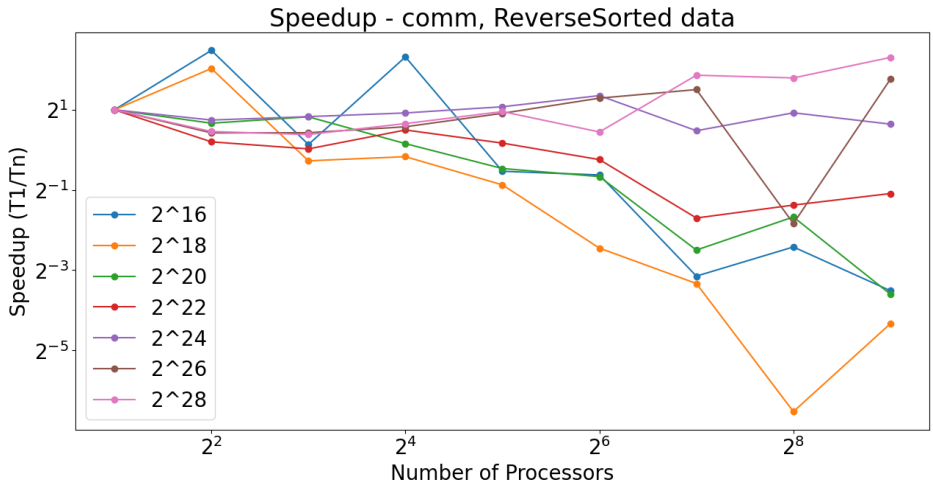

Speedup - comp_large, Random data
Speedup (T1/Tn)
Number of Processors
2^16
2^18
2^20
2^22
2^24
2^26
2^28


Speedup - comp_large, ReverseSorted data
Speedup (T1/Tn)
Number of Processors
2^16
2^18
2^20
2^22
2^24
2^26
2^28

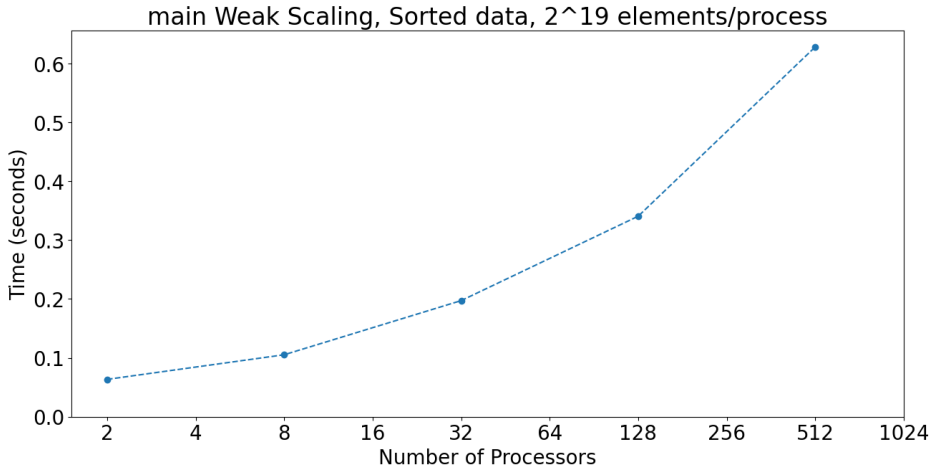

Speedup - comm, Sorted data
Speedup (T1/Tn)
Number of Processors
2^16
2^18
2^20
2^22
2^24
2^26
2^28

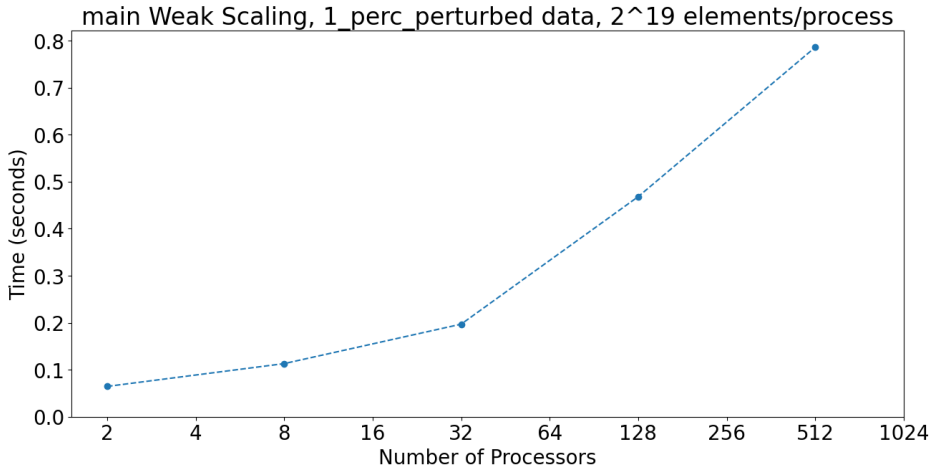

Speedup - comm, 1_perc_perturbed data
Speedup (T1/Tn)
Number of Processors
2^16
2^18
2^20
2^22
2^24
2^26
2^28

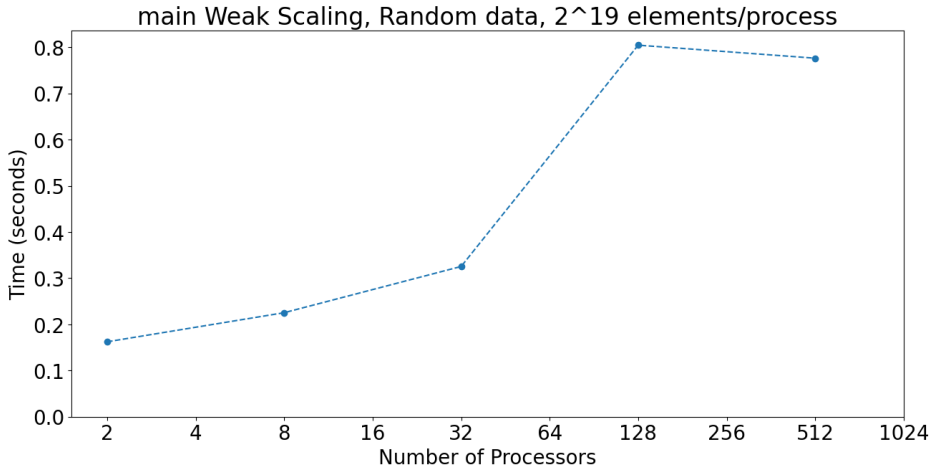

Speedup - comm, Random data
Speedup (T1/Tn)
Number of Processors
2^16
2^18
2^20
2^22
2^24
2^26
2^28

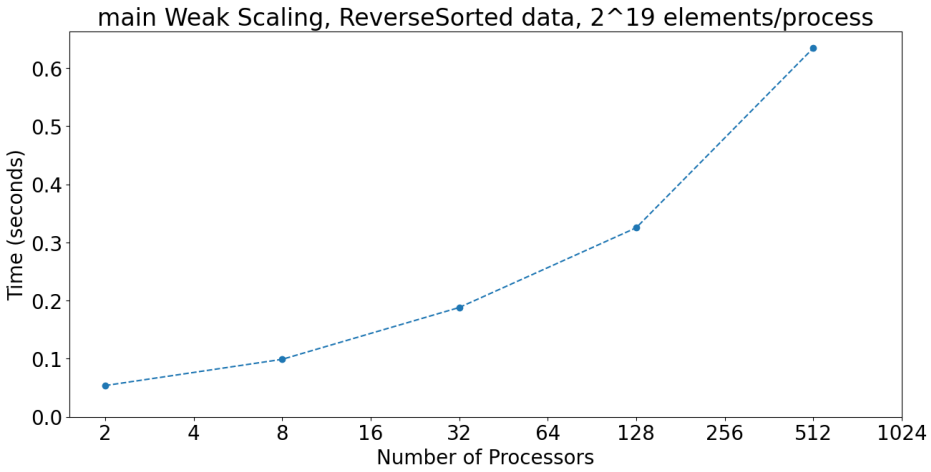

Speedup - comm, ReverseSorted data
Speedup (T1/Tn)
Number of Processors
2^16
2^18
2^20
2^22
2^24
2^26
2^28

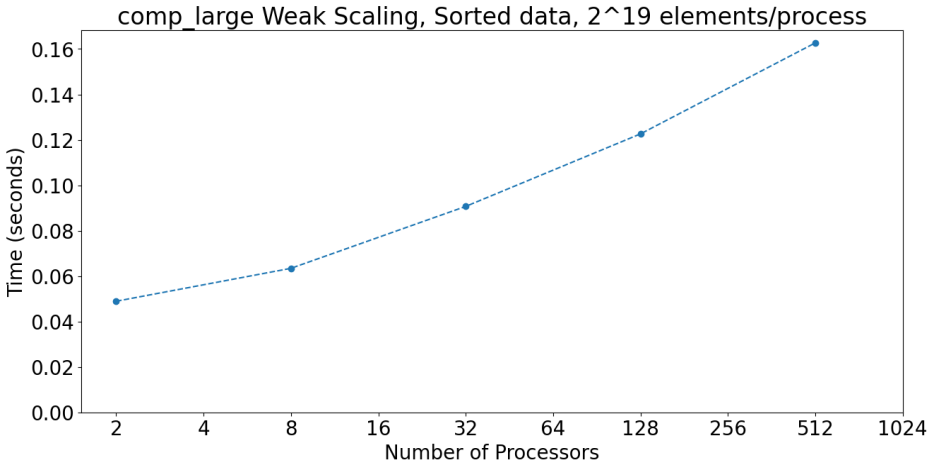
Just like with the last set of graphs, smaller input sizes do worse when adding more processors but larger input sizes do better. The speedup doesn't seem to level off for the 2^28 size, meaning it could run on even more processors before reaching peak parallelism. The computations speedup plots are smooth and increasing while the communication plots are jagged and only seem to stay level or go down. This is because computations are predictable and consistent, whereas the communication depends on what grace is doing and varies a lot. And the computation part is where the algorithm benefits from parallelism, not communication, so it makes sense the computation is increasing where the communication is not.

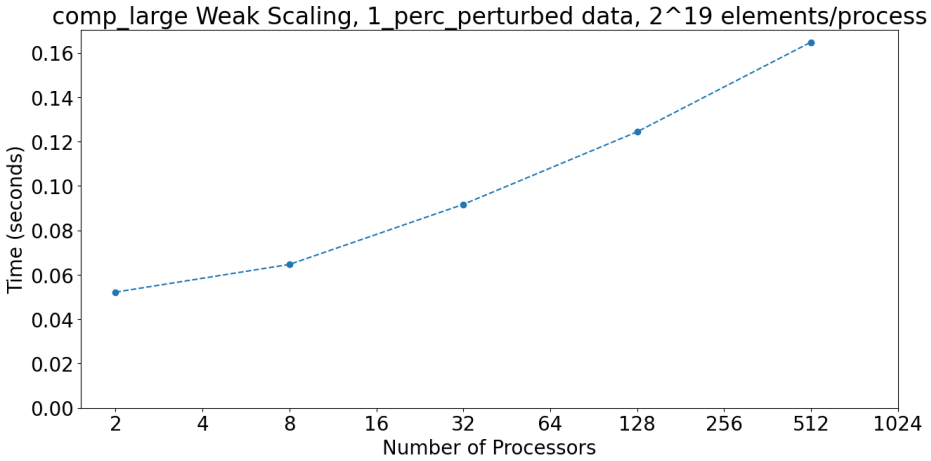


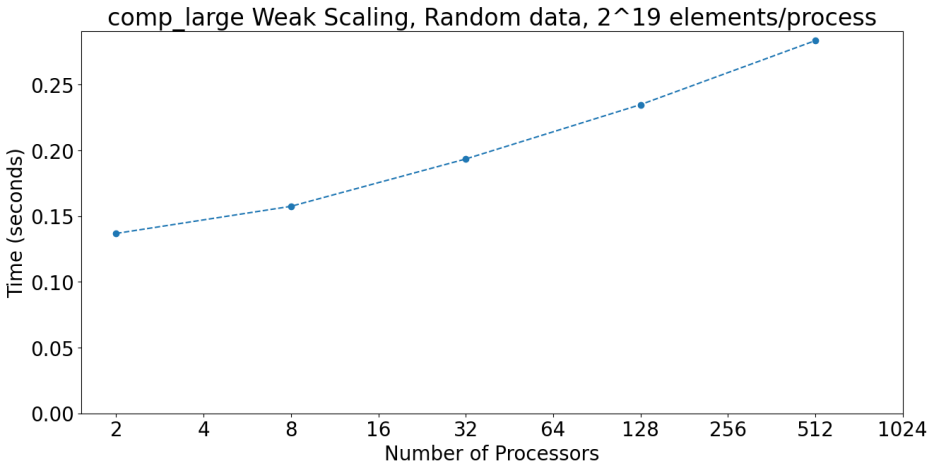


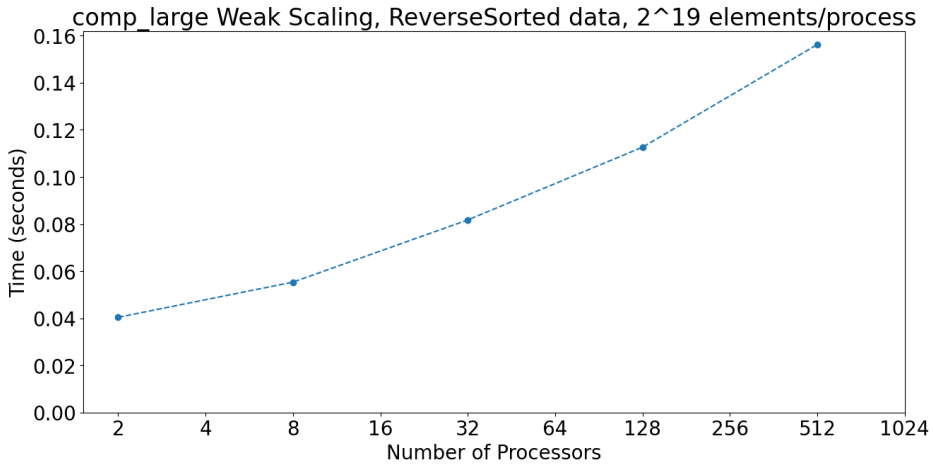


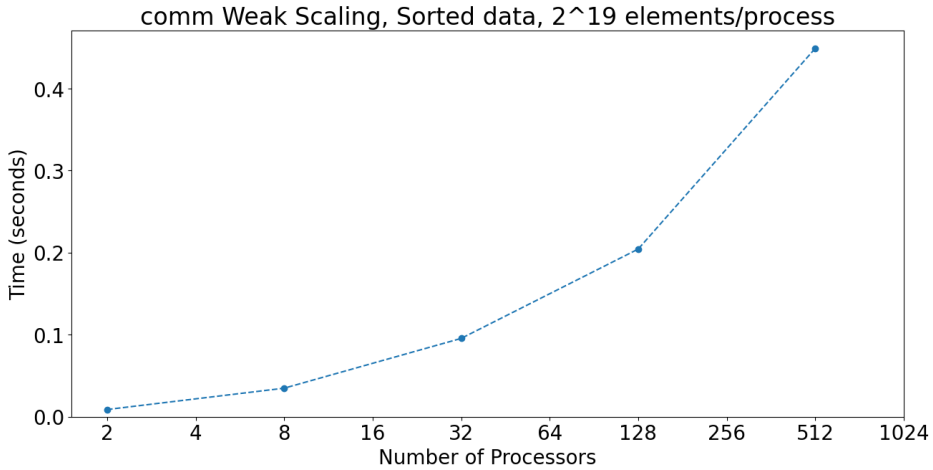

main Weak Scaling, ReverseSorted data, 2^19 elements/process
Time (seconds)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
2
4
8
16
32
64
128
256
512
1024
Number of Processors

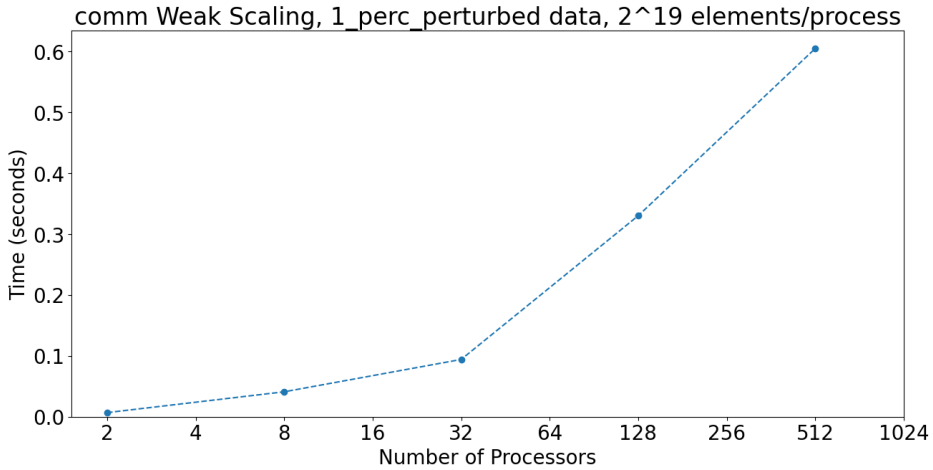

comp_large Weak Scaling, Sorted data, 2^19 elements/process
Time (seconds)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
0.16
2
4
8
16
32
64
128
256
512
1024
Number of Processors

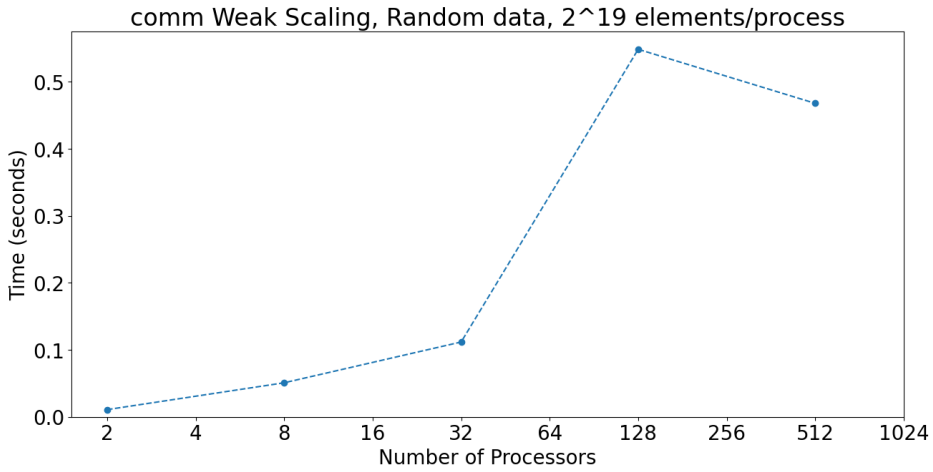

comp_large Weak Scaling, 1_perc_perturbed data, 2^19 elements/process
Time (seconds)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
0.16
2
4
8
16
32
64
128
256
512
1024
Number of Processors

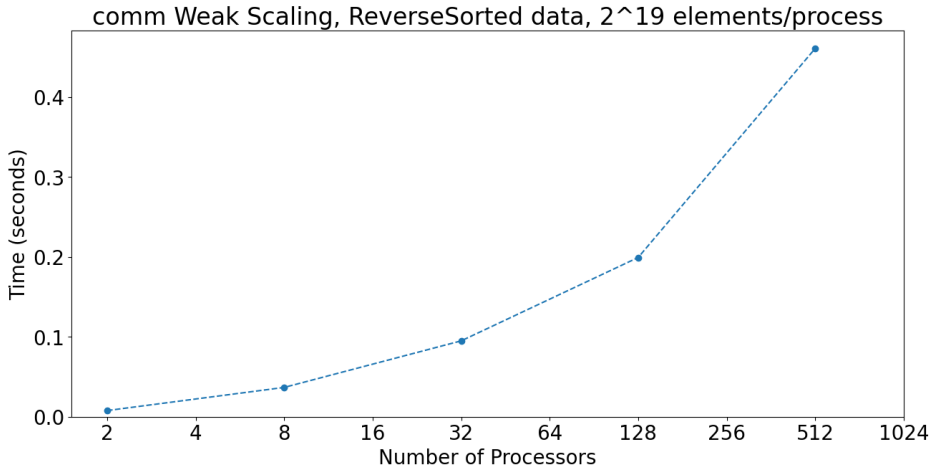

comp_large Weak Scaling, Random data, 2^19 elements/process
Time (seconds)
0.00
0.05
0.10
0.15
0.20
0.25
2
4
8
16
32
64
128
256
512
1024
Number of Processors


comp_large Weak Scaling, ReverseSorted data, 2^19 elements/process
Time (seconds)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
0.16
2
4
8
16
32
64
128
256
512
1024
Number of Processors

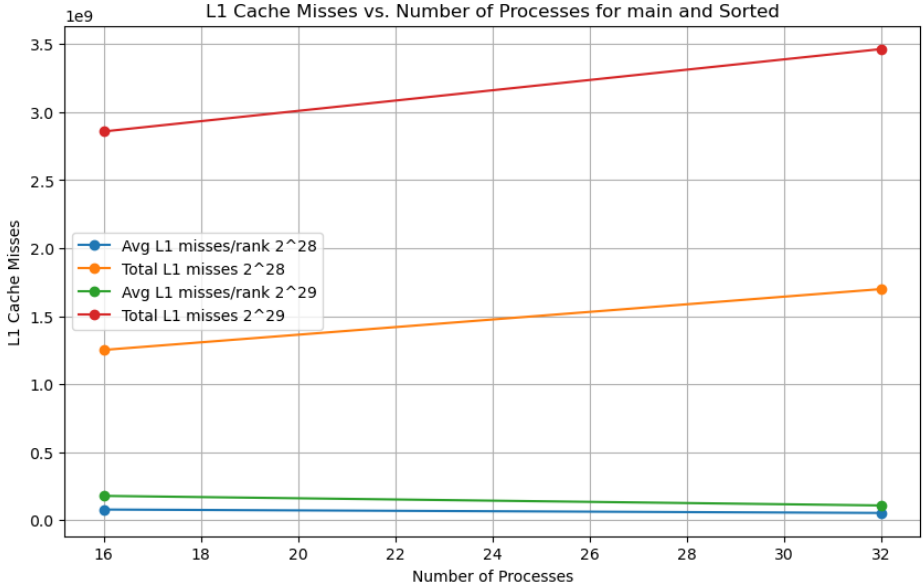

comm Weak Scaling, Sorted data, 2^19 elements/process
Time (seconds)
0.0
0.1
0.2
0.3
0.4
2
4
8
16
32
64
128
256
512
1024
Number of Processors

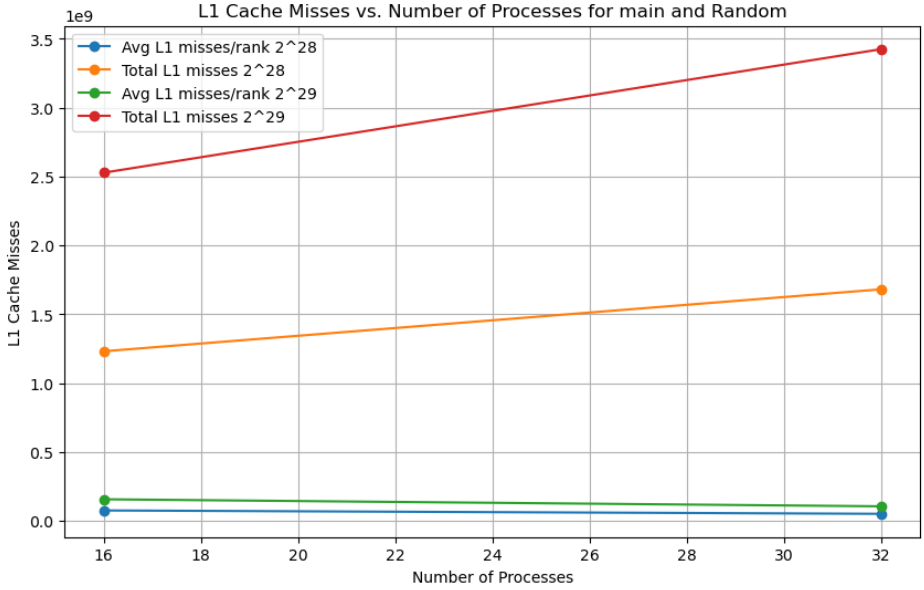

comm Weak Scaling, 1_perc_perturbed data, 2^19 elements/process
Time (seconds)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
2
4
8
16
32
64
128
256
512
1024
Number of Processors

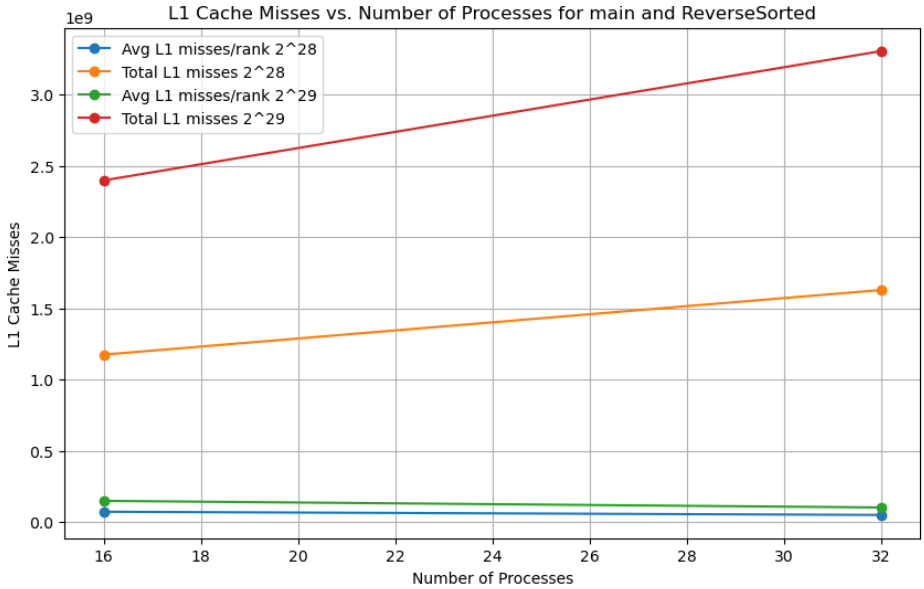


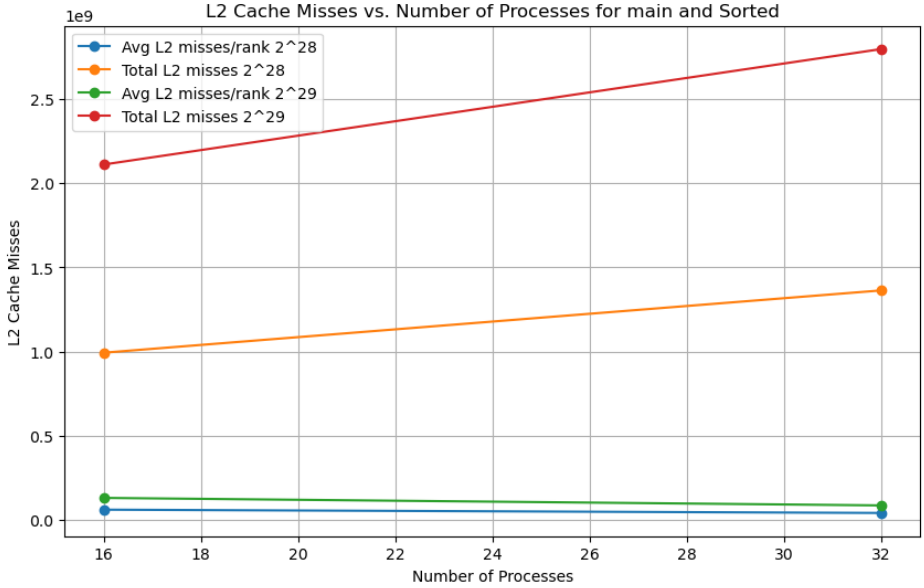


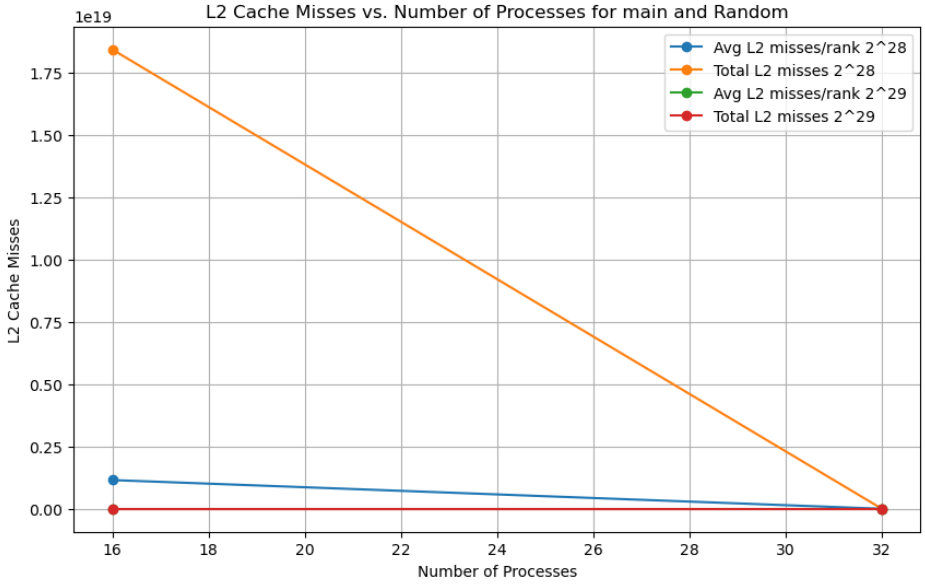
A perfectly parallel algorithm would show a level graph for the speedup, so it makes senses that these graphs are increasing. In fact, it wouldn't make sense for bitonic sort to have a level graph here because there are more communication steps needed for larger process counts.

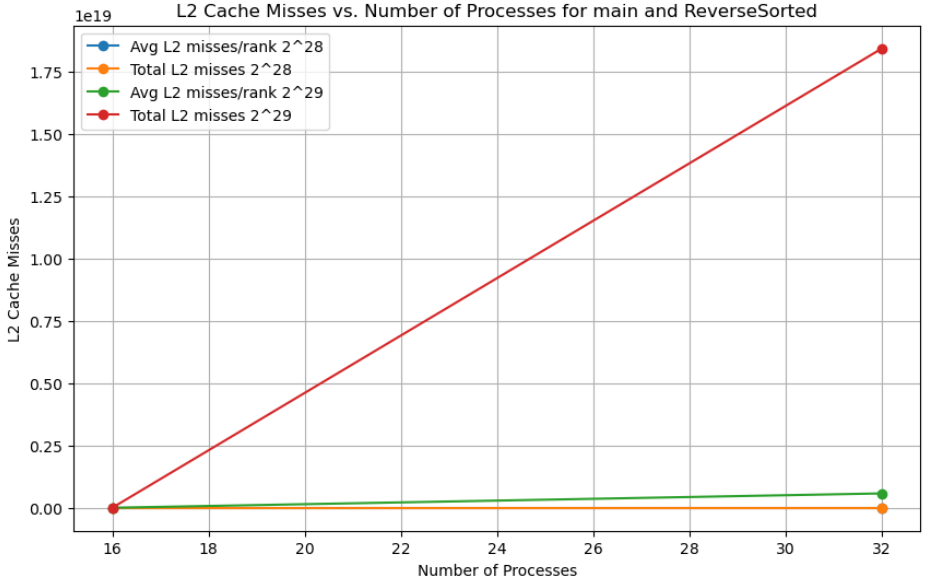

L1 Cache Misses vs. Number of Processes for main and Sorted
1e9
L1 Cache Misses
Number of Processes
Avg L1 misses/rank 2^28
Total L1 misses 2^28
Avg L1 misses/rank 2^29
Total L1 misses 2^29

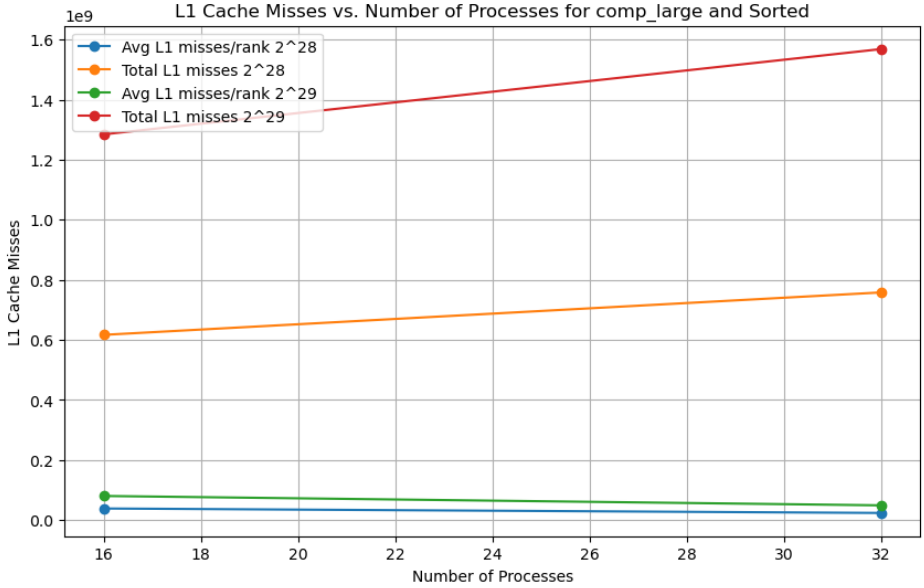

L1 Cache Misses vs. Number of Processes for main and Random
1e9
L1 Cache Misses
Number of Processes
Avg L1 misses/rank 2^28
Total L1 misses 2^28
Avg L1 misses/rank 2^29
Total L1 misses 2^29

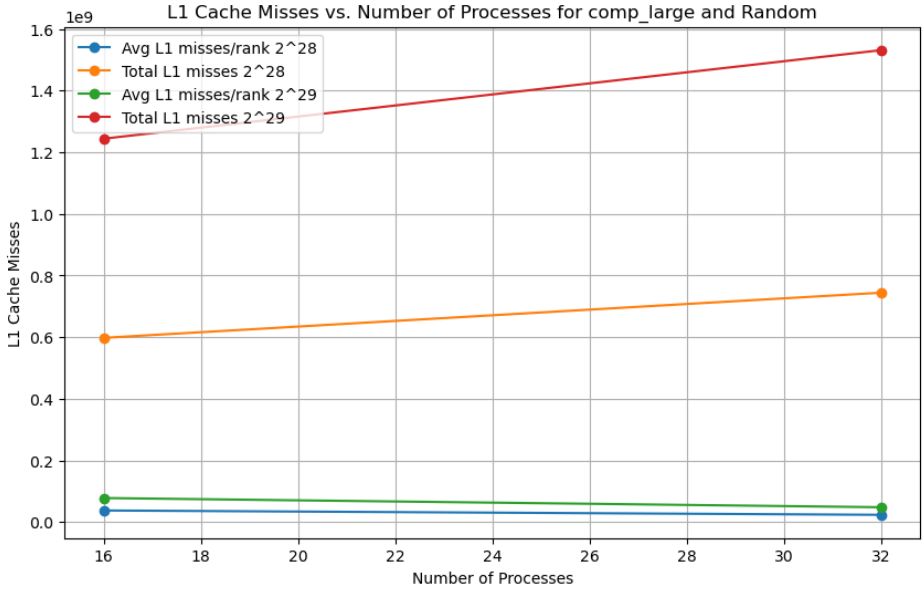

L1 Cache Misses vs. Number of Processes for main and ReverseSorted
1e9
L1 Cache Misses
Number of Processes
Avg L1 misses/rank 2^28
Total L1 misses 2^28
Avg L1 misses/rank 2^29
Total L1 misses 2^29

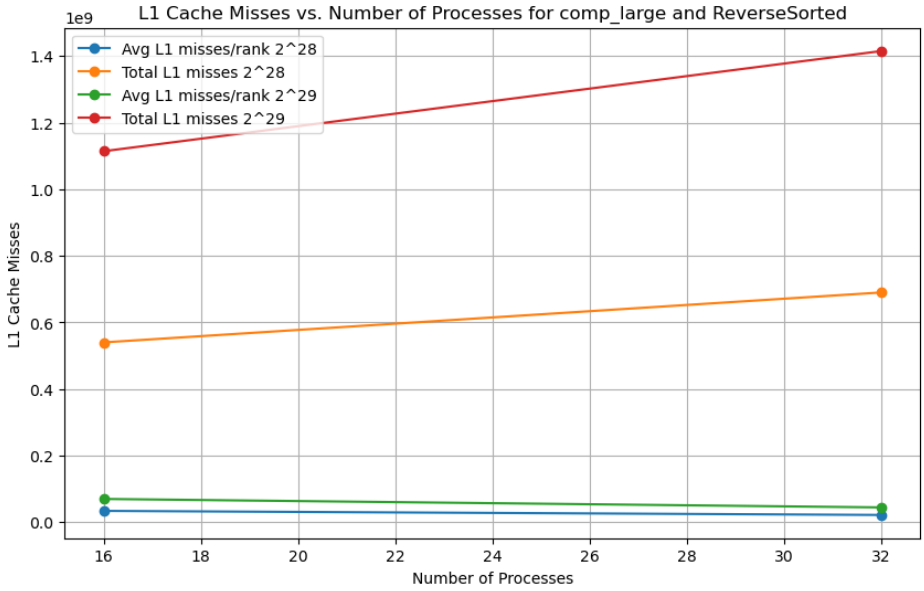

L2 Cache Misses vs. Number of Processes for main and Sorted
1e9
Avg L2 misses/rank 2^28
Total L2 misses 2^28
Avg L2 misses/rank 2^29
Total L2 misses 2^29
L2 Cache Misses
Number of Processes

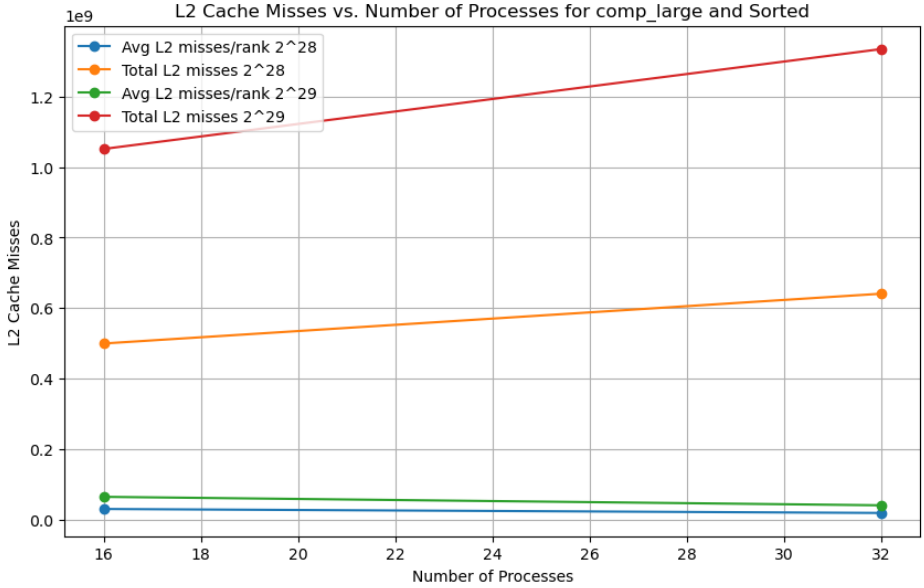

L2 Cache Misses vs. Number of Processes for main and Random
1e19
Avg L2 misses/rank 2^28
Total L2 misses 2^28
Avg L2 misses/rank 2^29
Total L2 misses 2^29
L2 Cache Misses
Number of Processes

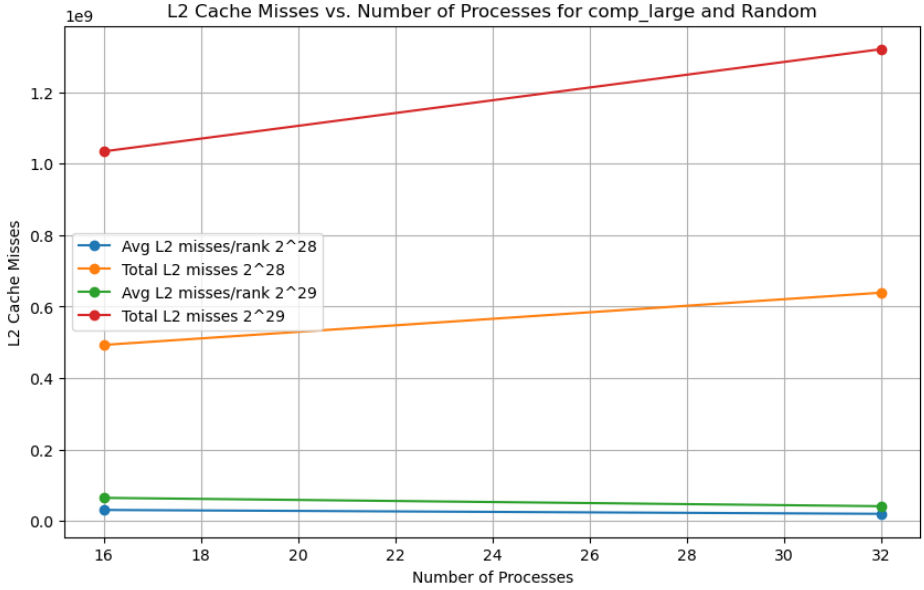

L2 Cache Misses vs. Number of Processes for main and ReverseSorted
1e19
Avg L2 misses/rank 2^28
Total L2 misses 2^28
Avg L2 misses/rank 2^29
Total L2 misses 2^29
L2 Cache Misses
Number of Processes

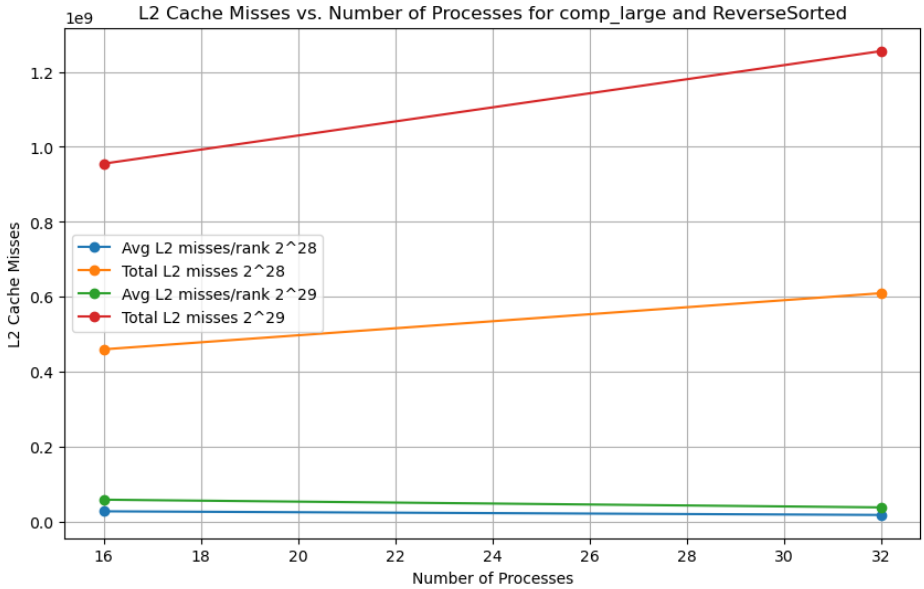

L1 Cache Misses vs. Number of Processes for comp_large and Sorted
1e9
Avg L1 misses/rank 2^28
Total L1 misses 2^28
Avg L1 misses/rank 2^29
Total L1 misses 2^29
L1 Cache Misses
Number of Processes

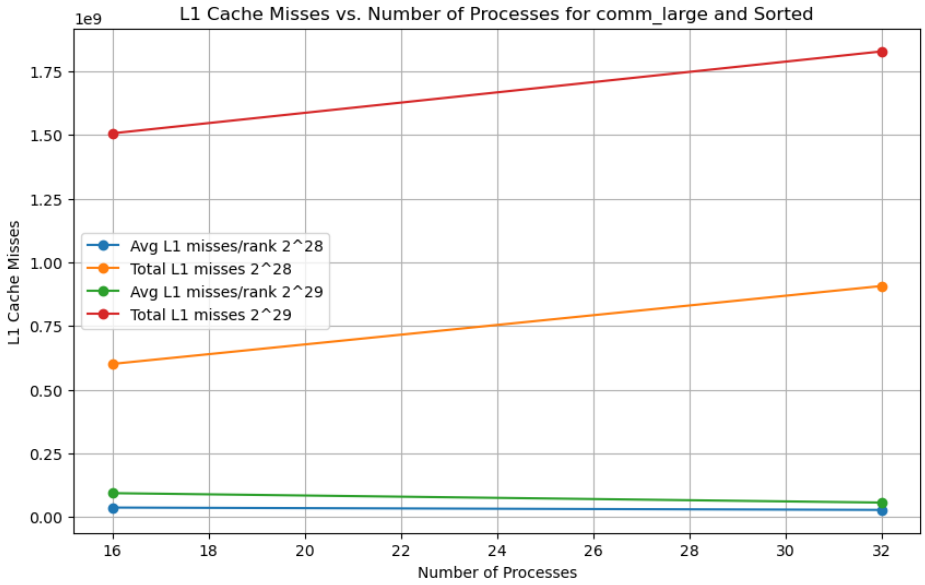

L1 Cache Misses vs. Number of Processes for comp_large and Random
1e9
Avg L1 misses/rank 2^28
Total L1 misses 2^28
Avg L1 misses/rank 2^29
Total L1 misses 2^29
L1 Cache Misses
Number of Processes

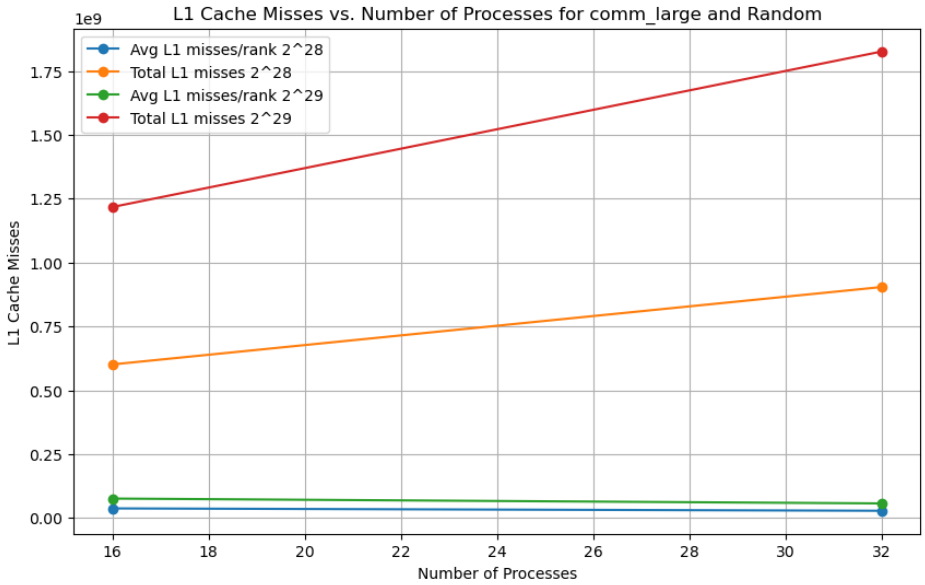

L1 Cache Misses vs. Number of Processes for comp_large and ReverseSorted
1e9
Avg L1 misses/rank 2^28
Total L1 misses 2^28
Avg L1 misses/rank 2^29
Total L1 misses 2^29
L1 Cache Misses
Number of Processes

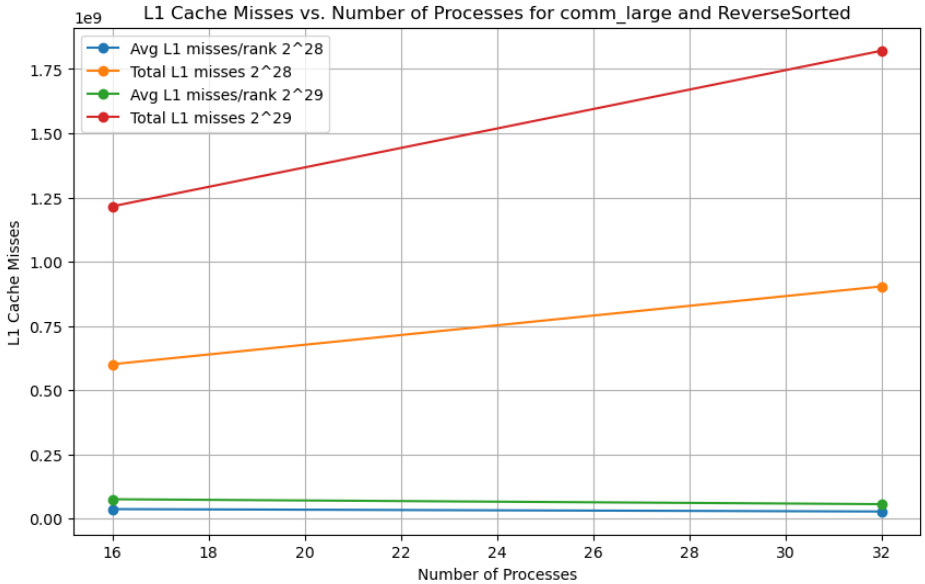

L2 Cache Misses vs. Number of Processes for comp_large and Sorted
1e9
Avg L2 misses/rank 2^28
Total L2 misses 2^28
Avg L2 misses/rank 2^29
Total L2 misses 2^29
L2 Cache Misses
Number of Processes

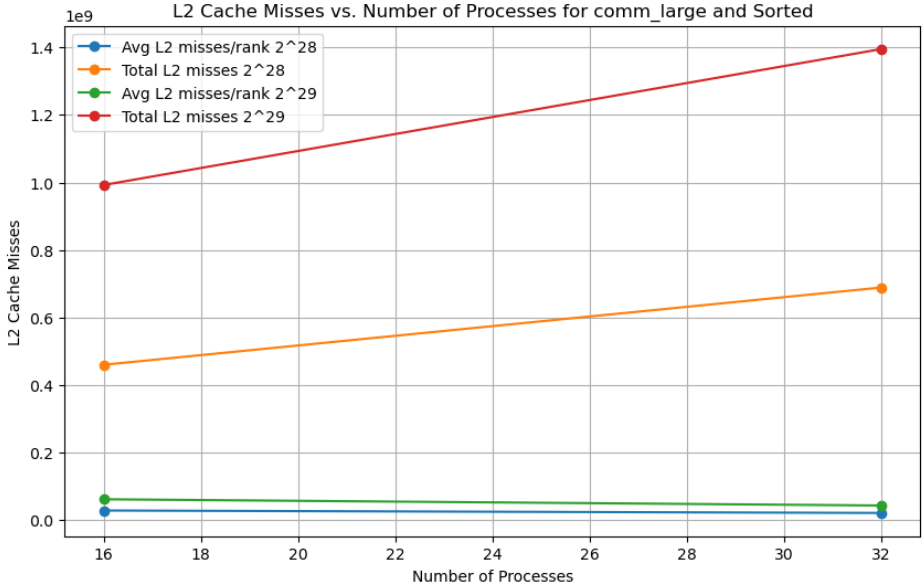

L2 Cache Misses vs. Number of Processes for comp_large and Random
1e9
Avg L2 misses/rank 2^28
Total L2 misses 2^28
Avg L2 misses/rank 2^29
Total L2 misses 2^29
L2 Cache Misses
Number of Processes

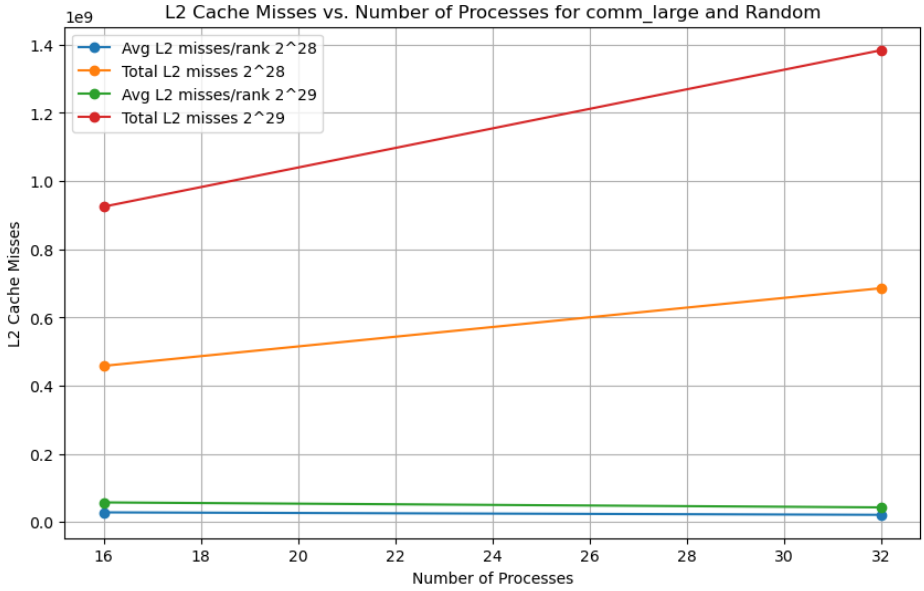

L2 Cache Misses vs. Number of Processes for comp_large and ReverseSorted
1e9
Avg L2 misses/rank 2^28
Total L2 misses 2^28
Avg L2 misses/rank 2^29
Total L2 misses 2^29
L2 Cache Misses
Number of Processes

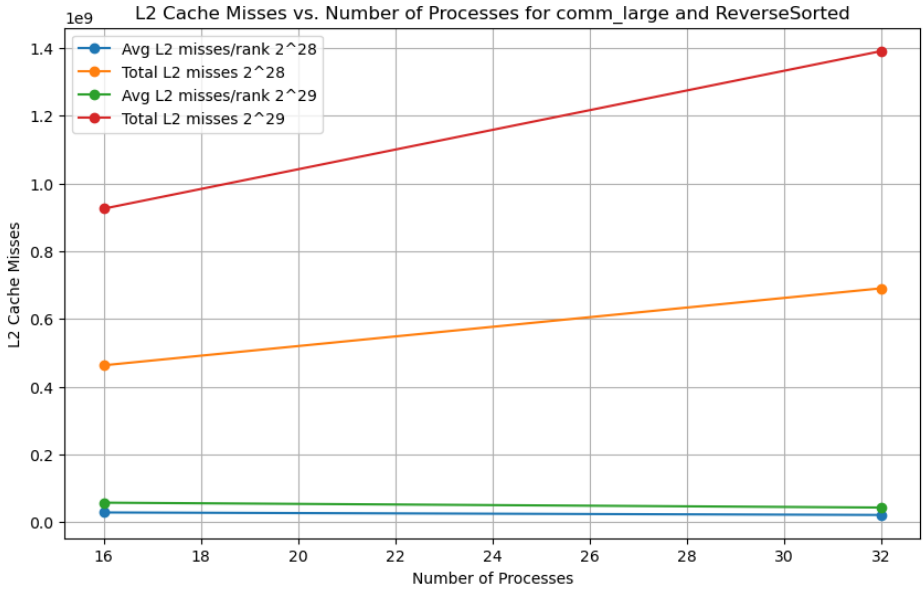

L1 Cache Misses vs. Number of Processes for comm_large and Sorted
1e9
L1 Cache Misses
Number of Processes
Avg L1 misses/rank 2^28
Total L1 misses 2^28
Avg L1 misses/rank 2^29
Total L1 misses 2^29


L1 Cache Misses vs. Number of Processes for comm_large and Random
1e9
L1 Cache Misses
Number of Processes
Avg L1 misses/rank 2^28
Total L1 misses 2^28
Avg L1 misses/rank 2^29
Total L1 misses 2^29


L1 Cache Misses vs. Number of Processes for comm_large and ReverseSorted
1e9
L1 Cache Misses
Number of Processes
Avg L1 misses/rank 2^28
Total L1 misses 2^28
Avg L1 misses/rank 2^29
Total L1 misses 2^29





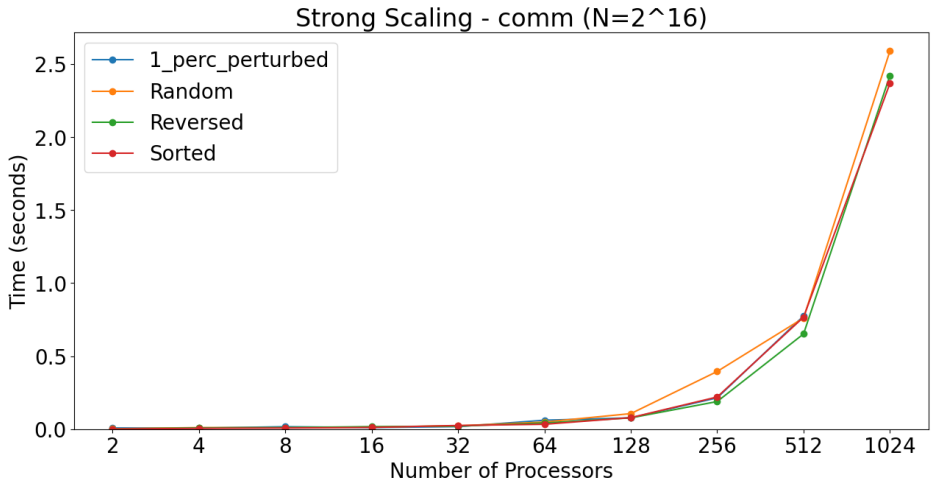


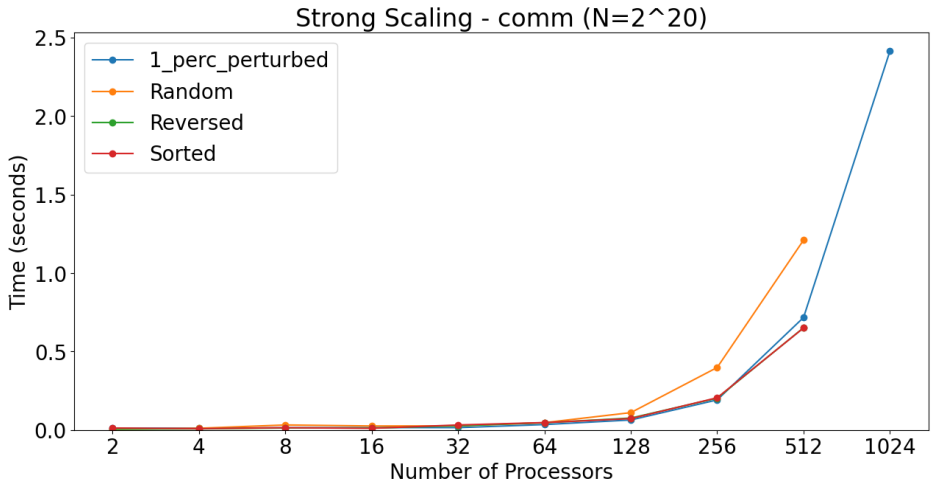
The total cache misses were higher for the larger number of processes, but the number of misses per rank were lower. This is because the problem size per processes is lower for the larger number of processes, but the larger number of processes means more steps so more computation and using memory. For some reason there were two outlier L2 cache misses just in main. This could be due to something weird connected to setting up MPI or something else outside our control.

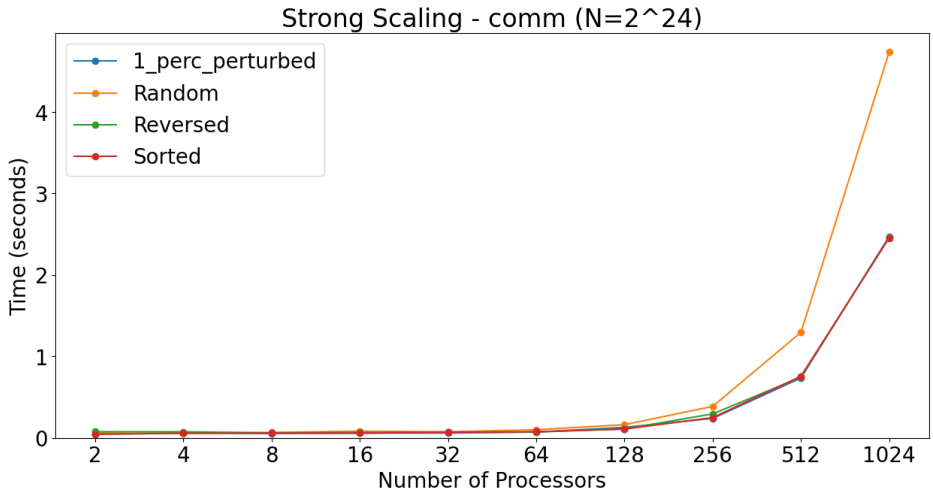
Sample Sort

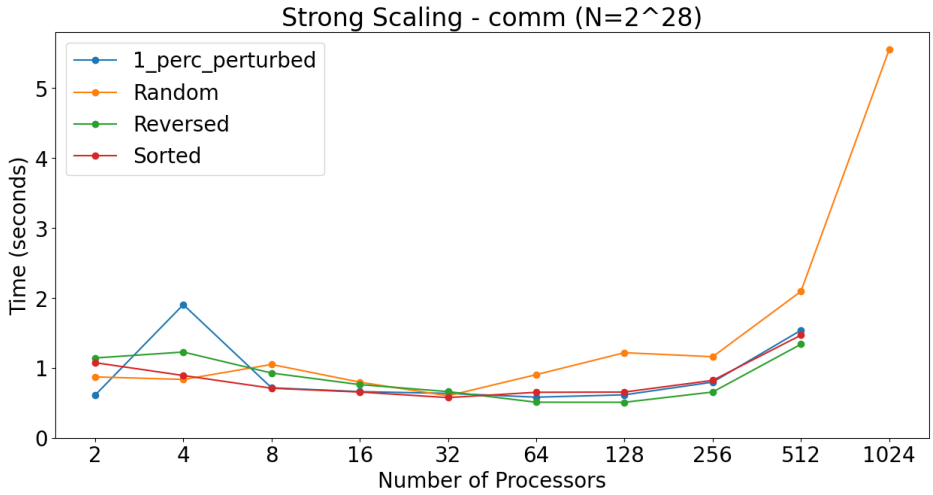
Communication

Strong Scaling



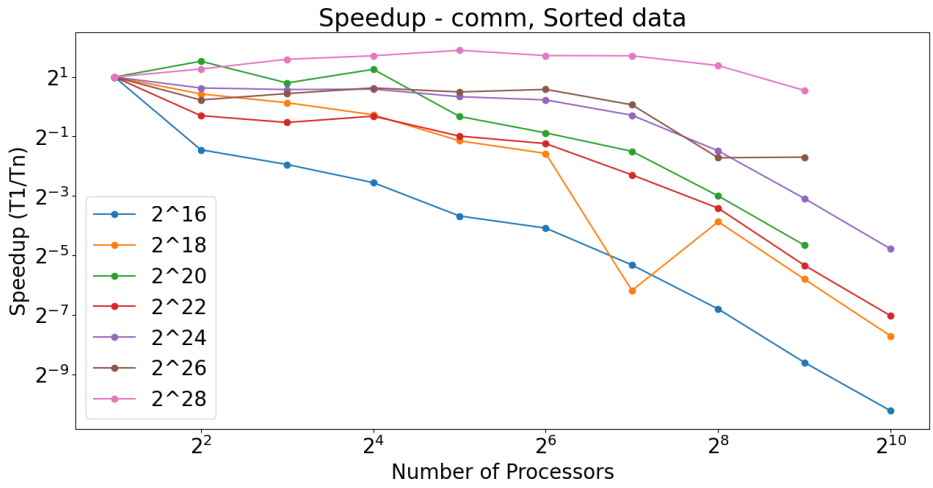


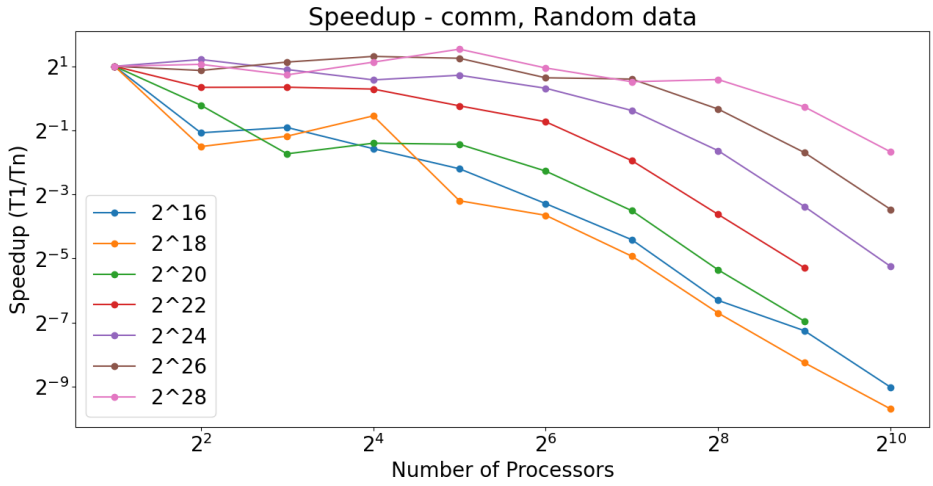




The time spent communicating is linearly proportional to the number of processes for all but the 2^28 graph, which is expected since each additional processor must communicate with all existing processors a constant amount of times. However, at this larger data size, the time is roughly constant when varying processors except for the single data point at 1024. This is likely because the bottleneck becomes the actual speed of sending data rather than forming and reforming connections.
From input size 2^16 to 2^20, the communication times do not meaningfully increase. This is because the overhead of establishing the channels accounts for most of the time. However, between the larger sizes, the time taken increases linearly with data size over most of the processor spectrum.
Comparing between the different input types, random is the slowest, with the other three types staying mostly clustered together. This is because for data sets with a predictable pattern, there will be a significant number of the processors that each process does not have to send data to.

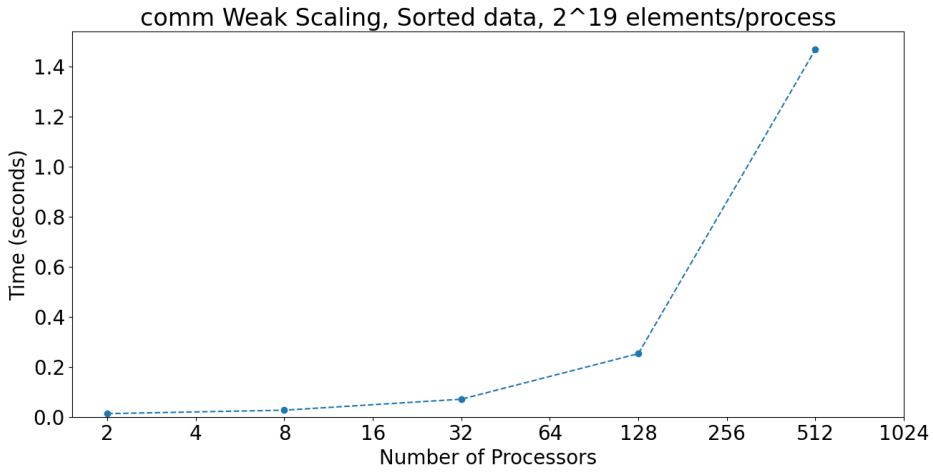
Speedup

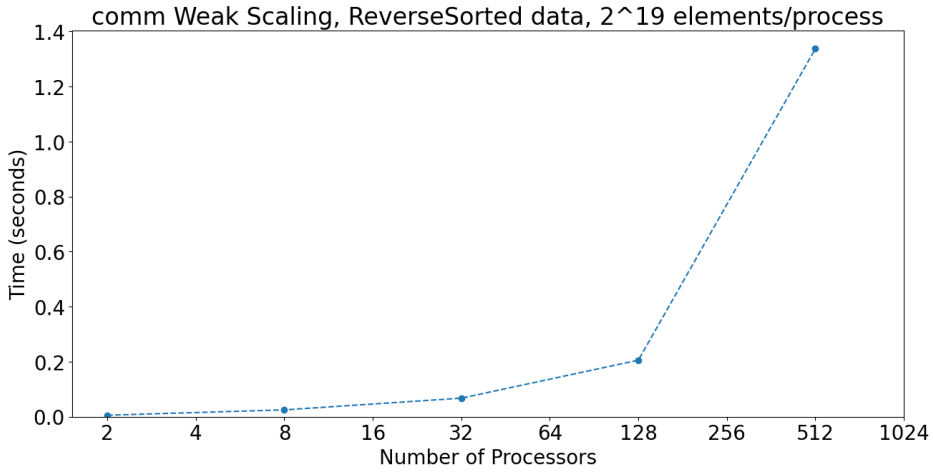




Communication time is minimized for a smaller number of processors, so almost all times do not improve over their time with 2 processors, resulting in a decreasing speedup for this step of the process.
This decline is worse for the smaller data sets, which is because they are setting up more communication channels but then not really taking advantage of them since only a small amount of data is sent.

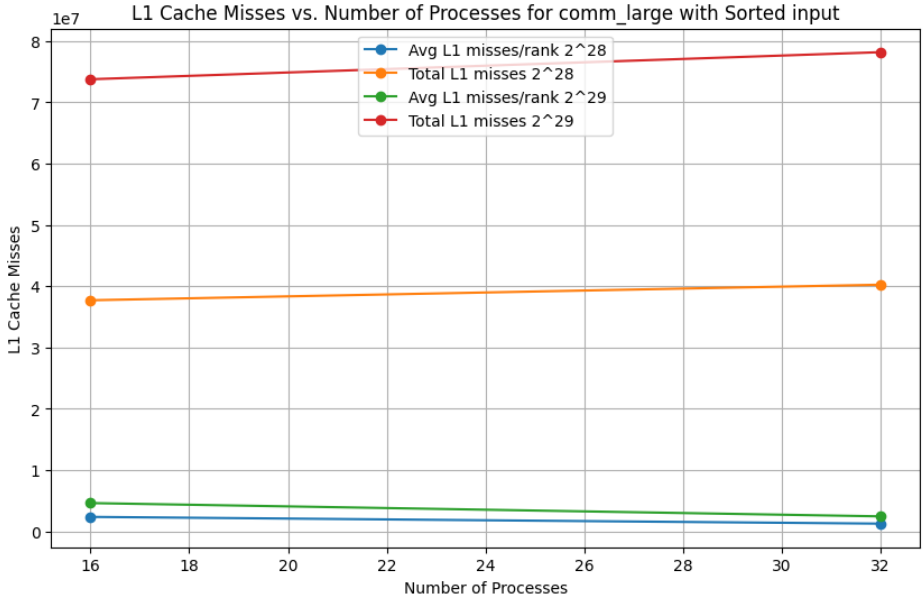
Weak Scaling

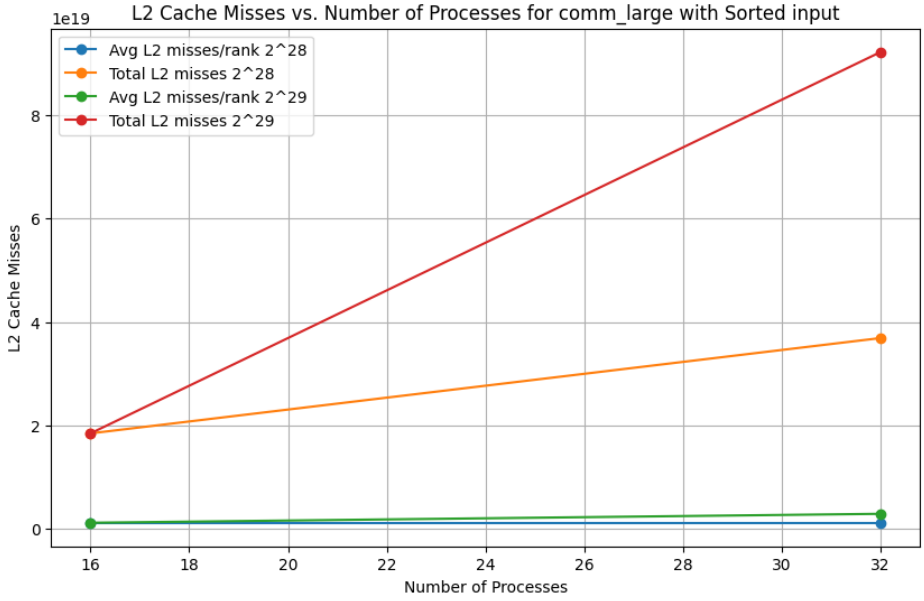


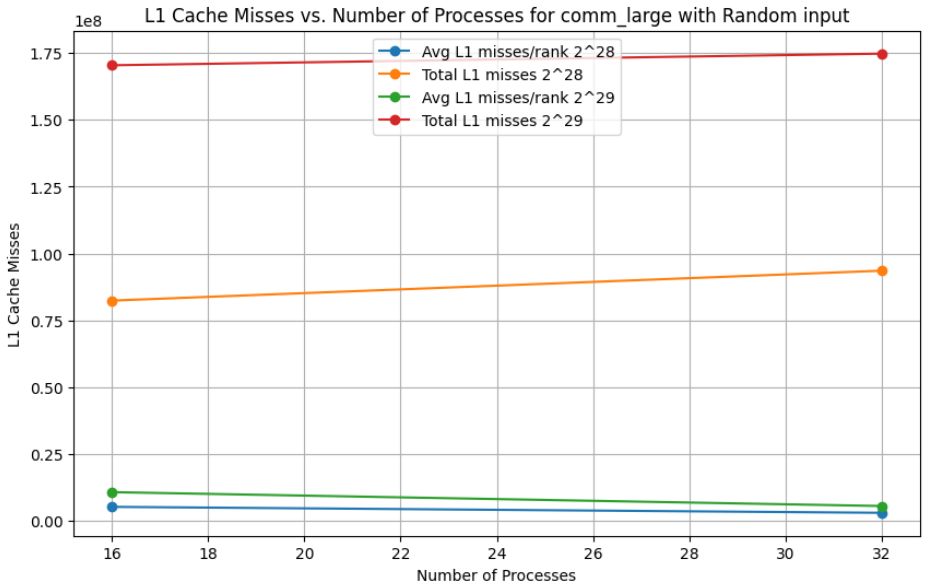


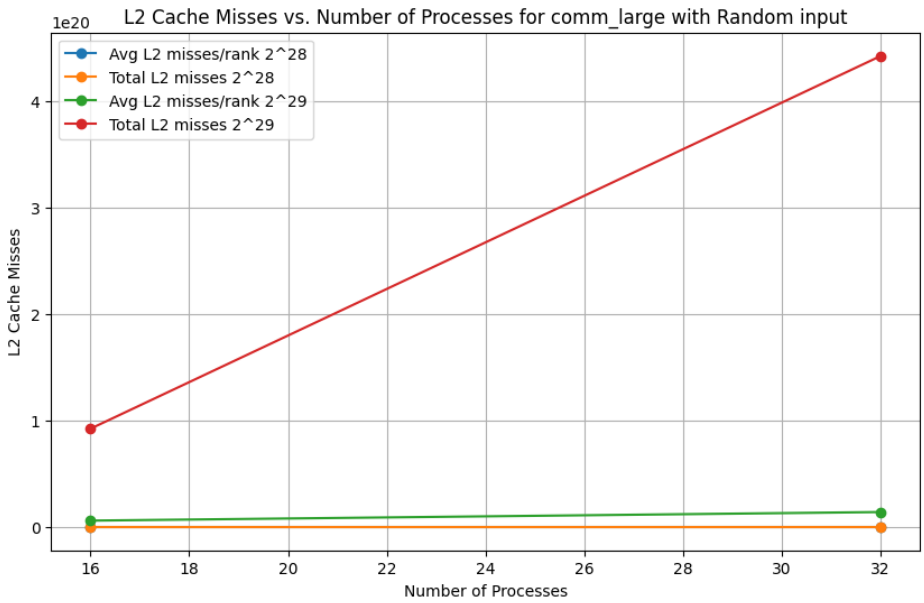
The time taken for communication increases linearly with the number of processors, since that means more send operations between processors.

Cache Misses


L1 Cache Misses vs. Number of Processes for comm_large with Sorted input
1e7
Avg L1 misses/rank 2^28
Total L1 misses 2^28
Avg L1 misses/rank 2^29
Total L1 misses 2^29
L1 Cache Misses
Number of Processes


L2 Cache Misses vs. Number of Processes for comm_large with Sorted input
1e19
Avg L2 misses/rank 2^28
Total L2 misses 2^28
Avg L2 misses/rank 2^29
Total L2 misses 2^29
L2 Cache Misses
Number of Processes

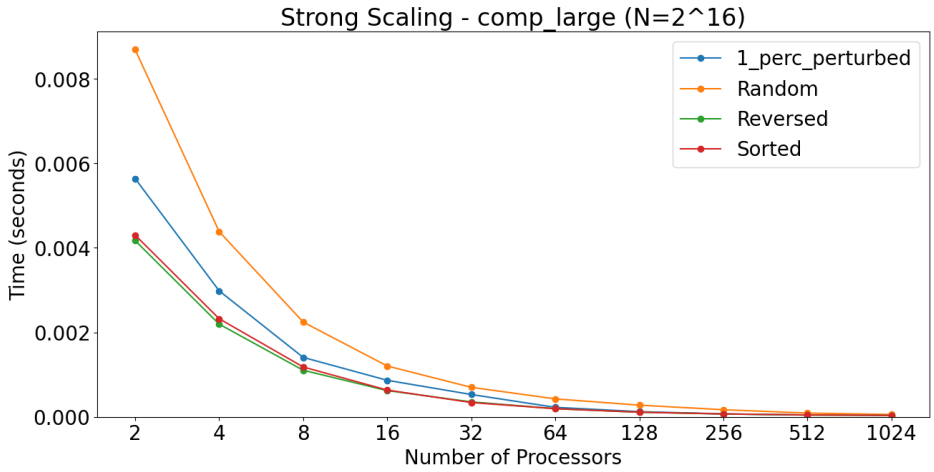


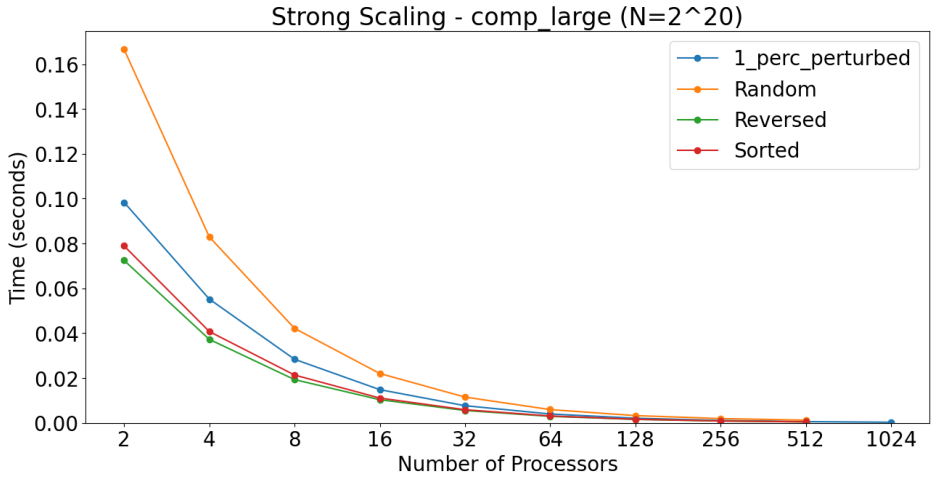


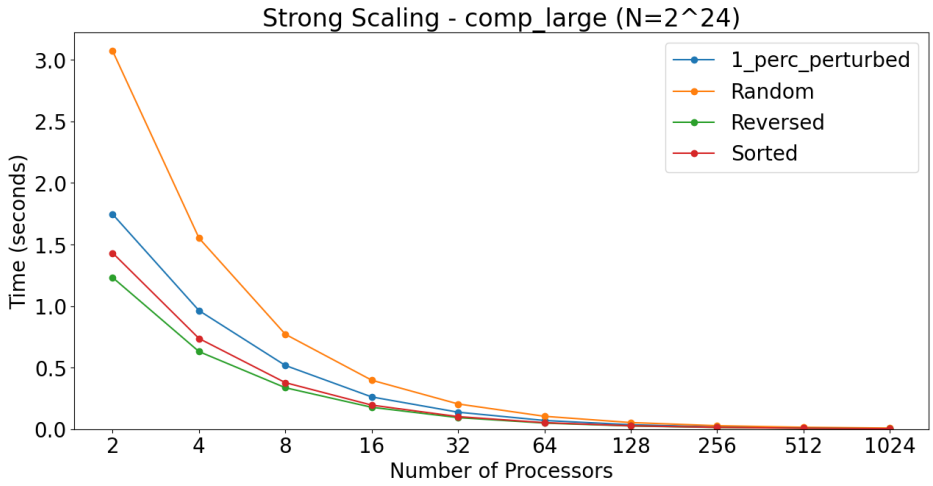
The L1 cache misses are effectively the same for the two different input types. They also do not change much for a different number of processors.
For the L2 cache, the misses for 2^29 elements are about the same, but sorted input leads to significantly more misses for 2^28. In all cases, there are many more cache misses at 32 processors compared to 16.

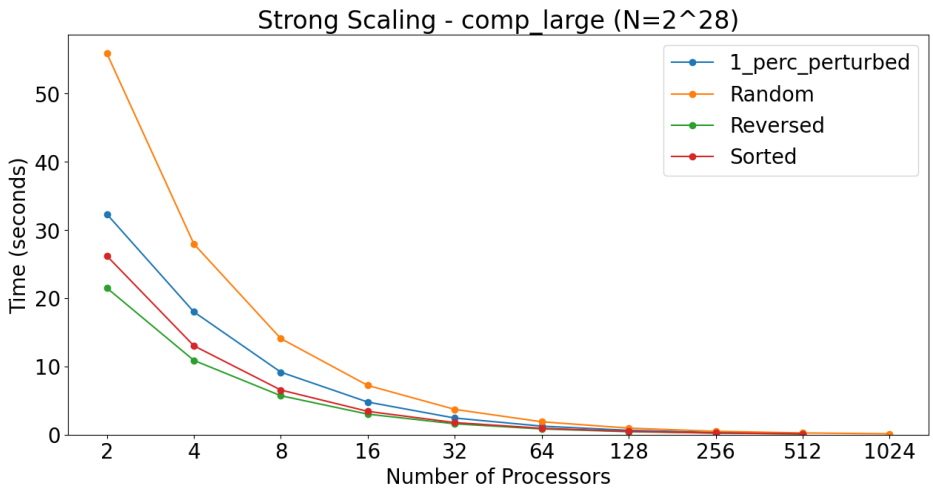
Computation

Strong Scaling





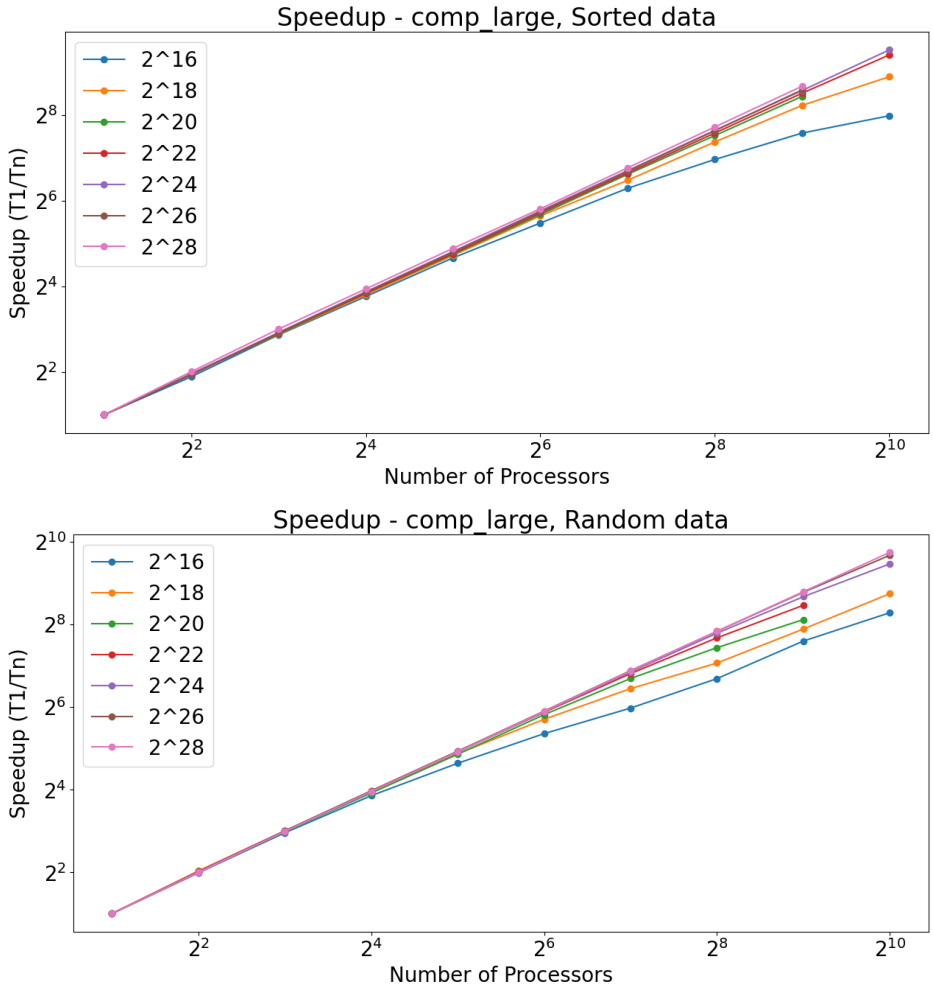




The time taken to sort the data decreases linearly with the number of processors, which indicates that the additional processors are being utilized effectively for the sorting.

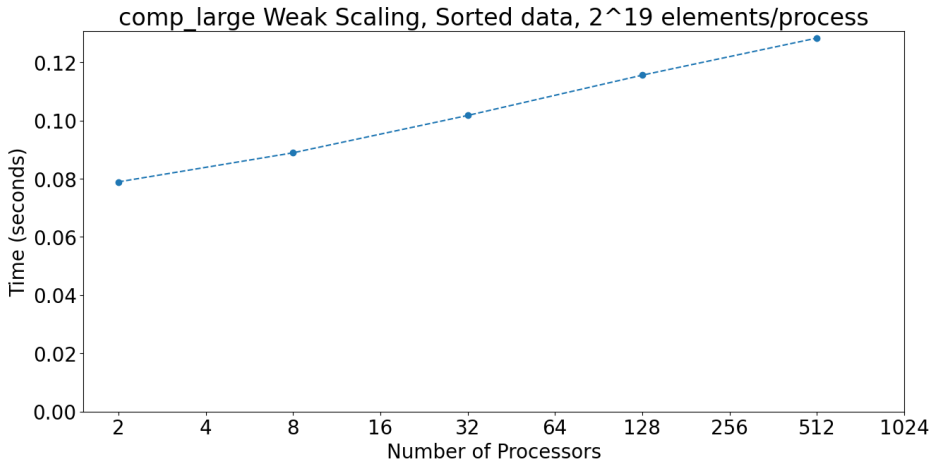
Between the different data sizes, the sorting time is scaling with nlogn. In most cases it is slightly better, jumping by about 20x instead of the 24x that would be expected for an 8x bigger input.
Random data is significantly slower, and perturbed data is slightly slower than sorted and reversed data. This is because there will be more variance in how many each elements get for each process since there is a guranteed number of splitters within each processor's starting data (3). This means that a processor can get 2 processor's worth of data at maximum for sorted and reversed data, while there is no such guarantee for random data. As more processors get added, the average amount of data each processor has to sort decreases, so the penalty for one processor getting too much data decreases, which is why the difference between random data and the other data types goes away for more processors.

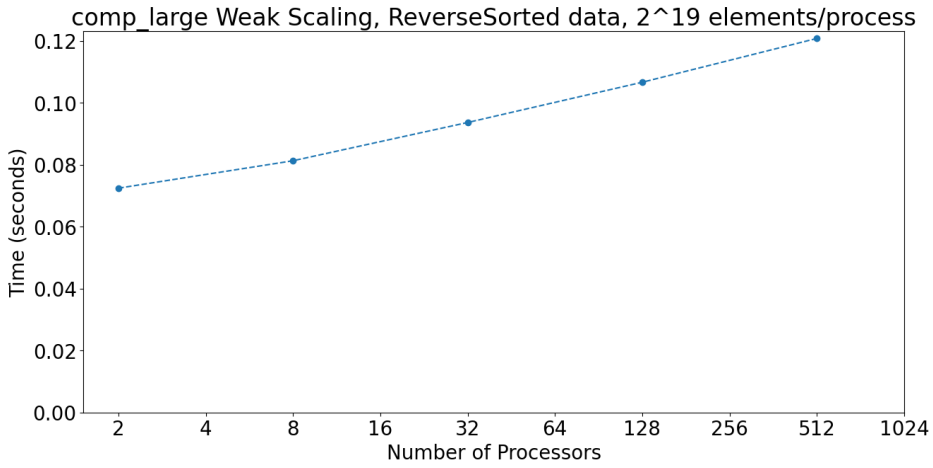
Speedup



The speedup does increase linearly across almost the entire range measured. For higher processor counts, the speedup gains do start to slightly decrease, and this effect is more pronounced for smaller inputs. This is because the size of the array of splitters becomes bigger, which means that the binary search to determine which processor should get each data element takes longer.
The speedups between the different data types are relatively consistent.

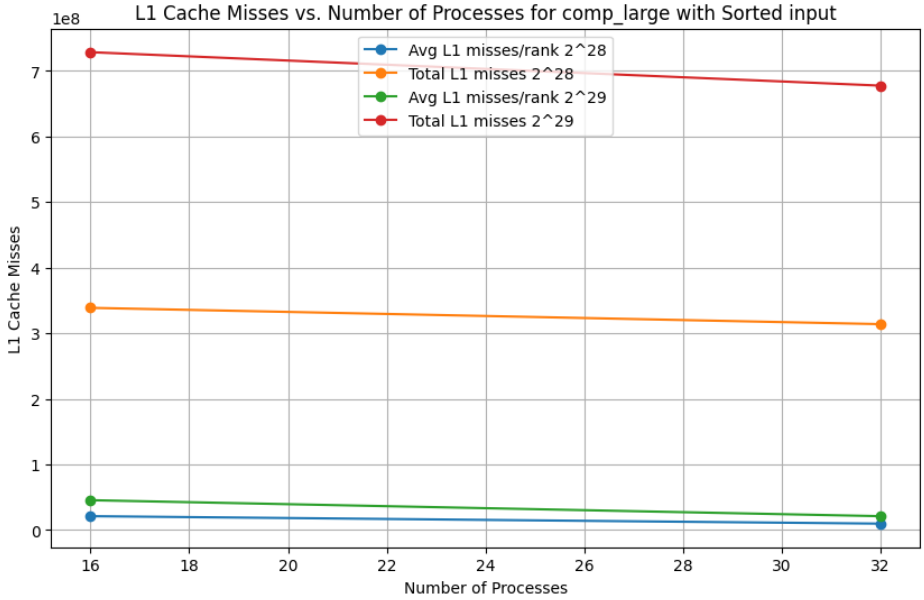
Weak Scaling

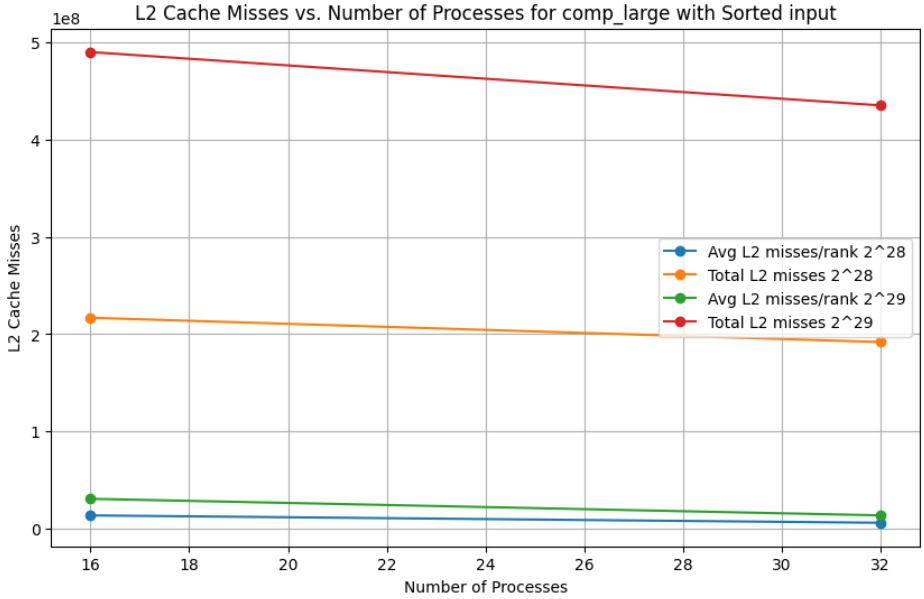


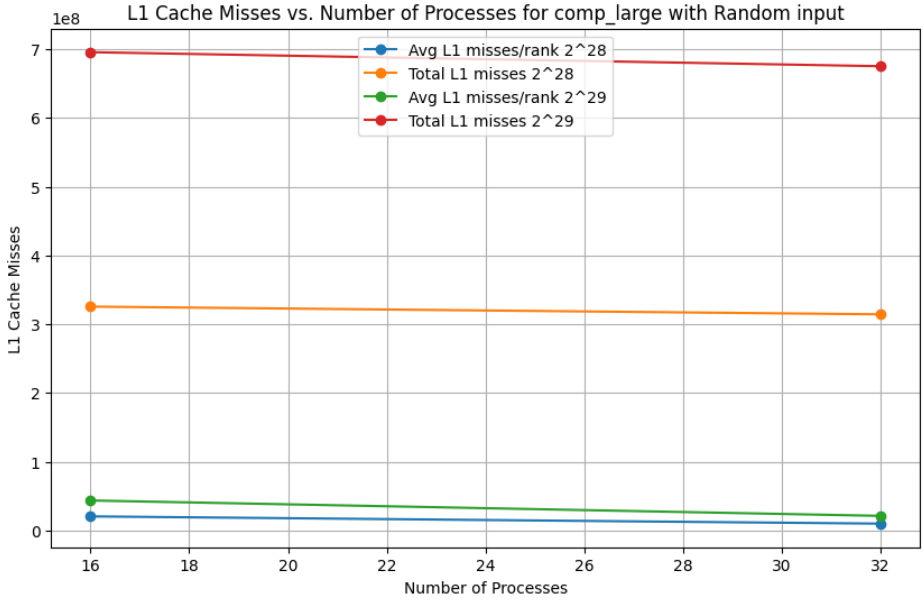


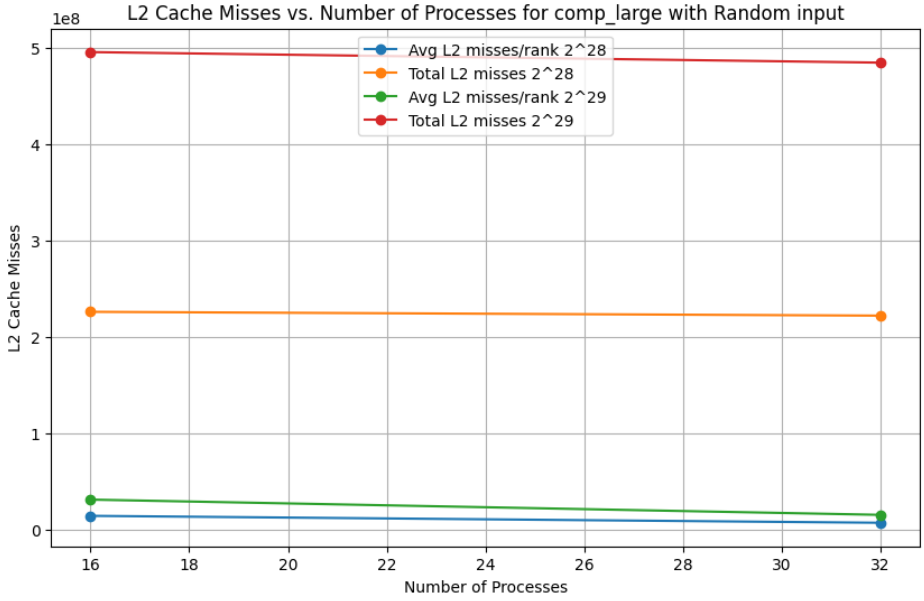
The computation time increases logarithmically with the number of processors used. Though the amount of time to sort the data should stay constant since the average sort size should stay the same, the time taken to partition the elements into buckets is proportional to log(p).
The type of data used does not significantly affect the weak scaling measurements.

Cache Misses


L1 Cache Misses vs. Number of Processes for comp_large with Sorted input
1e8
Avg L1 misses/rank 2^28
Total L1 misses 2^28
Avg L1 misses/rank 2^29
Total L1 misses 2^29
L1 Cache Misses
Number of Processes


L2 Cache Misses vs. Number of Processes for comp_large with Sorted input
1e8
Avg L2 misses/rank 2^28
Total L2 misses 2^28
Avg L2 misses/rank 2^29
Total L2 misses 2^29
L2 Cache Misses
Number of Processes

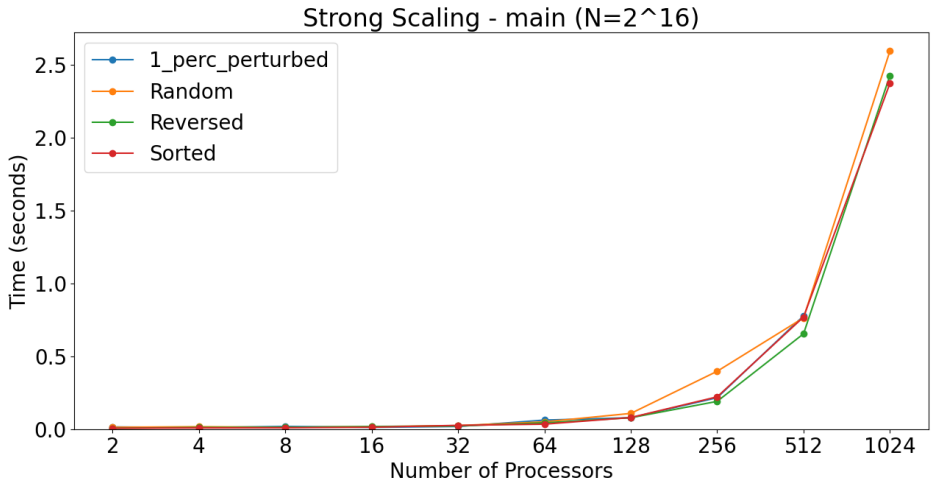


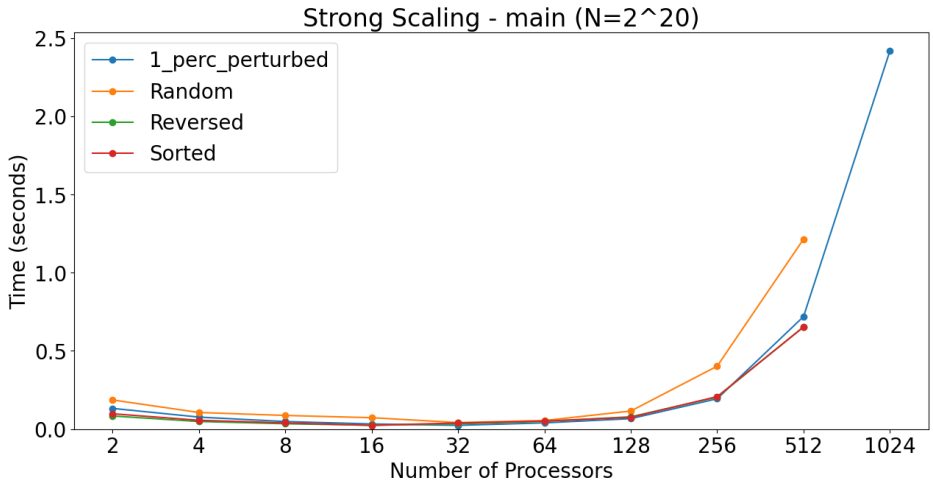


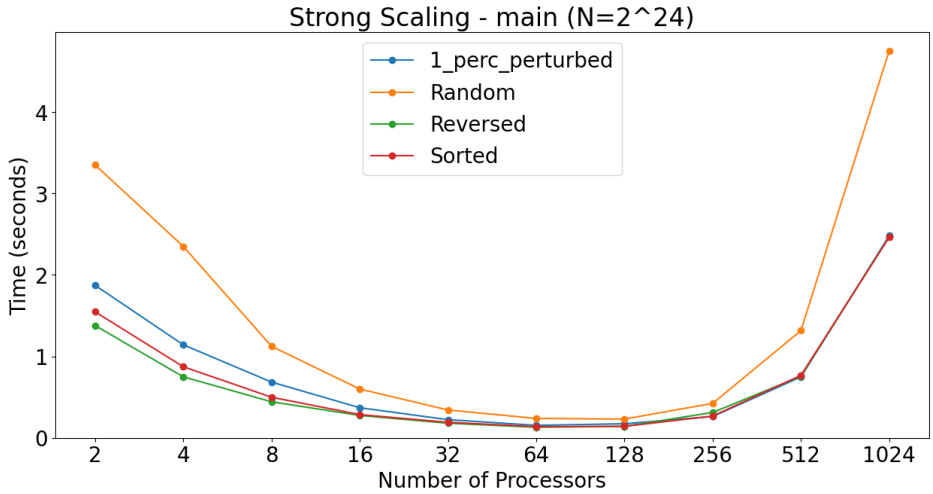
All of the cache misses for the computation show roughly the same pattern. Doubling the size of the array doubles the number of cache misses, and doubling the number of processors slightly decreases the amount of cache misses observed. This holds for different input types and for both the L1 and L2 caches.

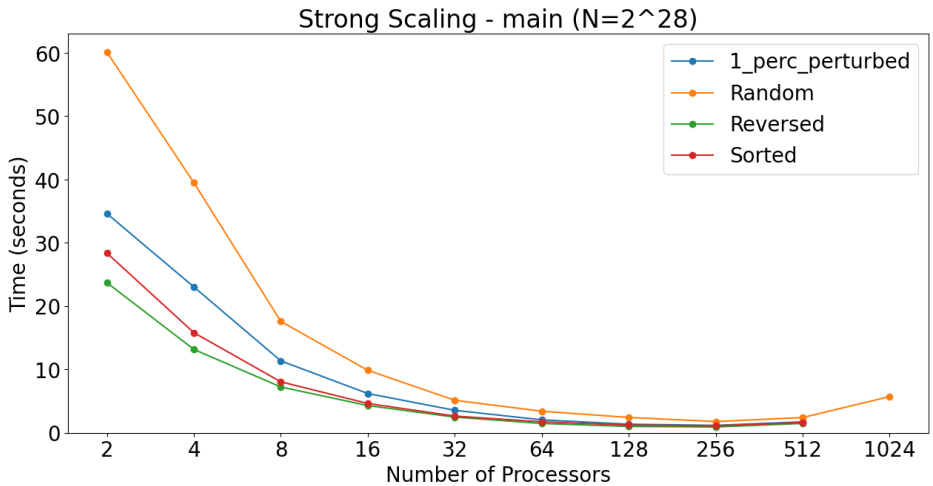
Main

Strong Scaling


Strong Scaling - main (N=2^16)
1_perc_perturbed
Random
Reversed
Sorted
Time (seconds)
Number of Processors
0.0
0.5
1.0
1.5
2.0
2.5
2
4
8
16
32
64
128
256
512
1024

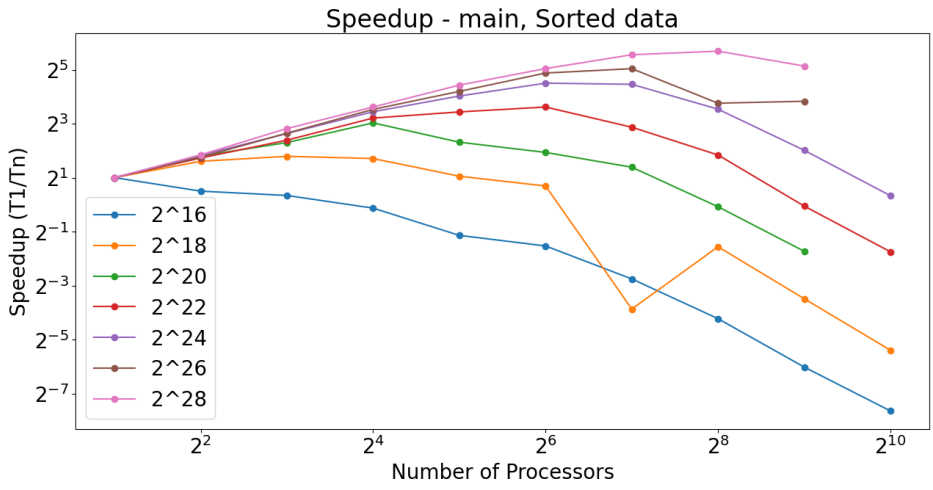

Strong Scaling - main (N=2^20)
1_perc_perturbed
Random
Reversed
Sorted
Time (seconds)
Number of Processors
0.0
0.5
1.0
1.5
2.0
2.5
2
4
8
16
32
64
128
256
512
1024

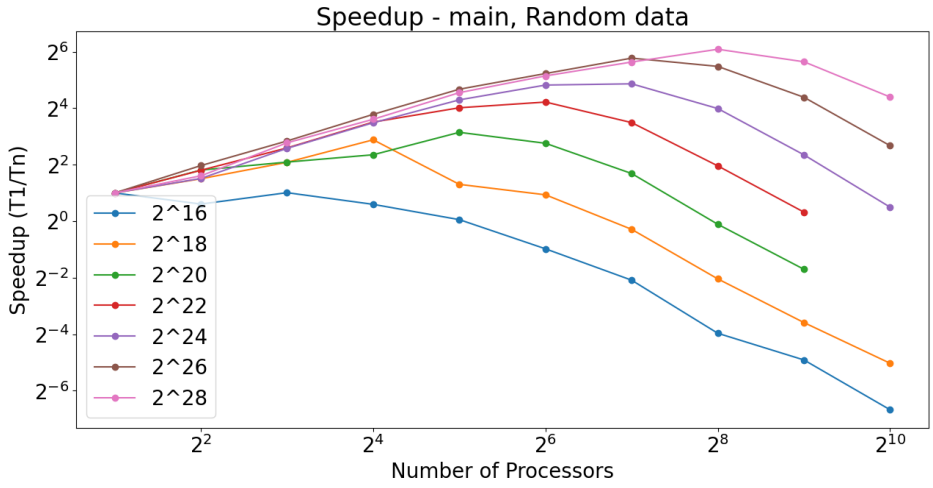

Strong Scaling - main (N=2^24)
1_perc_perturbed
Random
Reversed
Sorted
Time (seconds)
Number of Processors
0
1
2
3
4
2
4
8
16
32
64
128
256
512
1024


Strong Scaling - main (N=2^28)
1_perc_perturbed
Random
Reversed
Sorted
Time (seconds)
Number of Processors
0
10
20
30
40
50
60
2
4
8
16
32
64
128
256
512
1024

These graphs demonstrate the difference between processor numbers for different input sizes, and show that the optimal number of processors varies between the different input sizes. For 2^16, 2 processors is the fastest option available since there is not much data to sort in the first place, so communication overhead is counterproductive; for 2^20, about 32 processors is best, and adding the communication time for multiple nodes offsets the gains; for 2^24, about 128 processors is best; even for 2^28, anything beyond 256 processors does not help sort the data faster.
The additional time taken for the communication and computation with random data does manifest in the main graphs, with random taking noticeably longer to run.

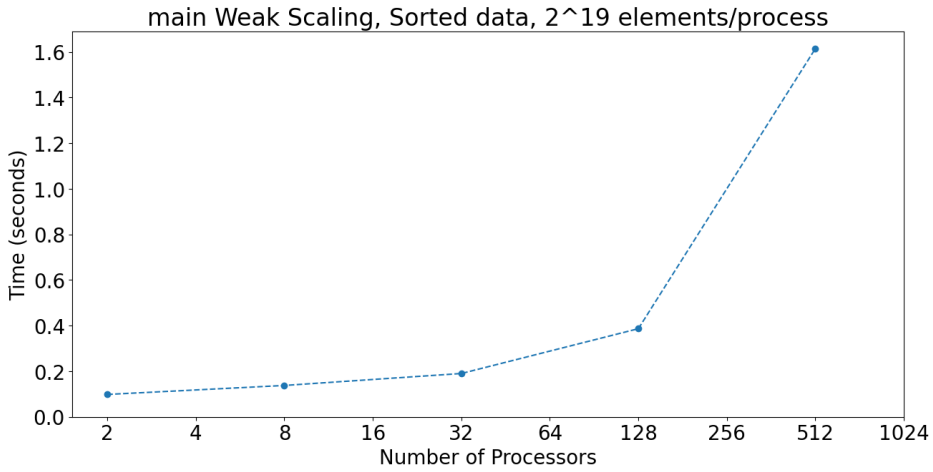
Speedup

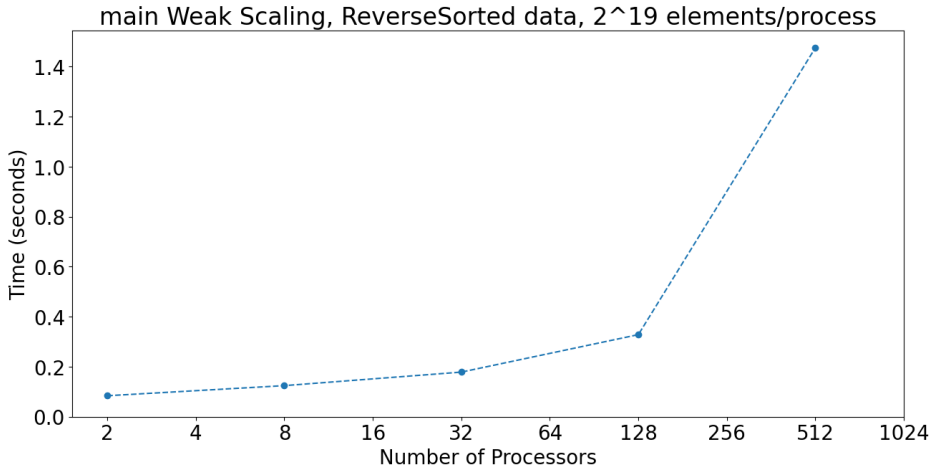




The speedup graphs show the strong scaling data in a slightly different format, and the gradual movement of the peaks towards the right shows the benefits of adding more processors as data size increases. However, no tested data size is large enough for 1024 processors to be practical, and so all of the plots start to decrease at some point.

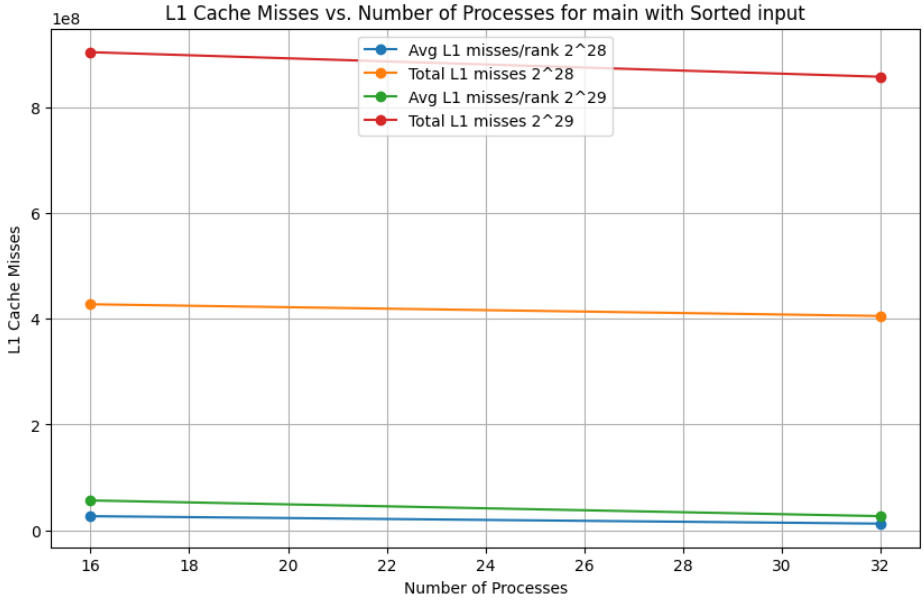
Weak Scaling

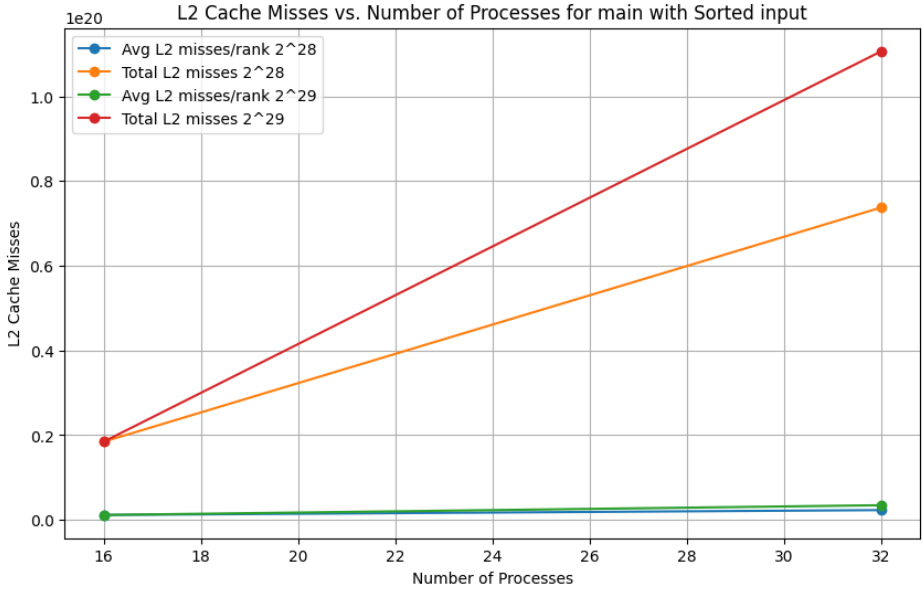


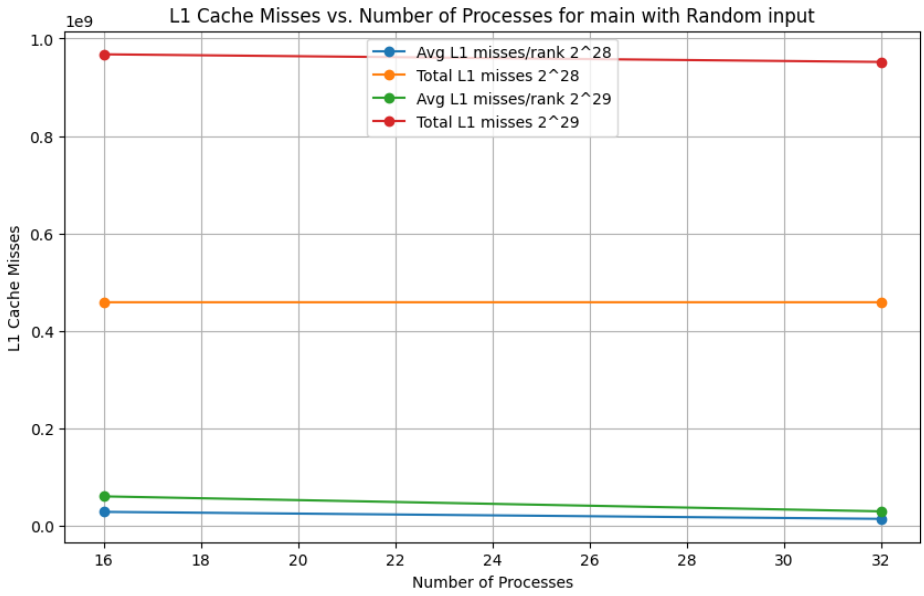


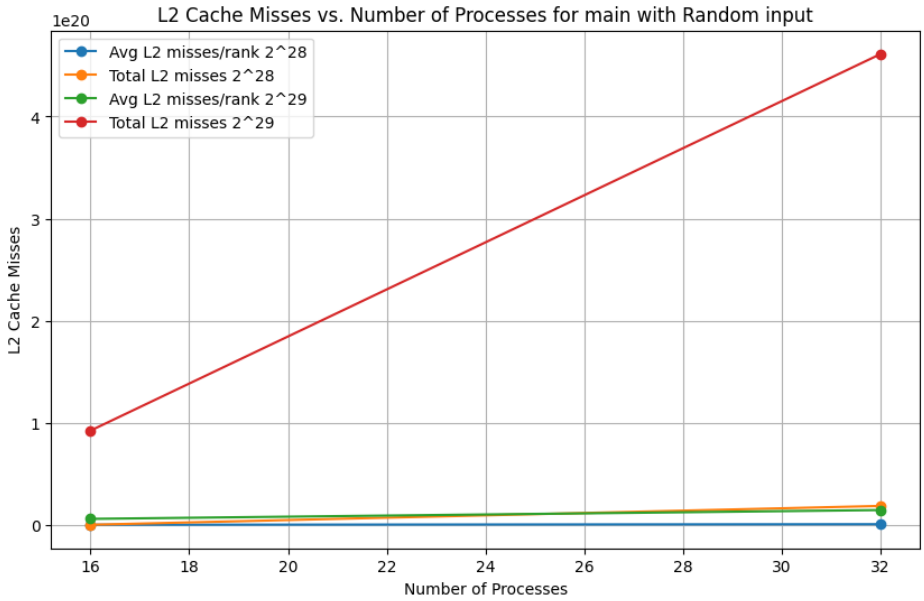
The linear increase in the communication time dominates the logarithmic increase of the computation time, and so the weak scaling graphs look most similar to those of the communication. This indicates the best place to look for further optimizations would be in the communication stage.

Cache Misses


L1 Cache Misses vs. Number of Processes for main with Sorted input
1e8
Avg L1 misses/rank 2^28
Total L1 misses 2^28
Avg L1 misses/rank 2^29
Total L1 misses 2^29
L1 Cache Misses
Number of Processes


L2 Cache Misses vs. Number of Processes for main with Sorted input
1e20
Avg L2 misses/rank 2^28
Total L2 misses 2^28
Avg L2 misses/rank 2^29
Total L2 misses 2^29
L2 Cache Misses
Number of Processes



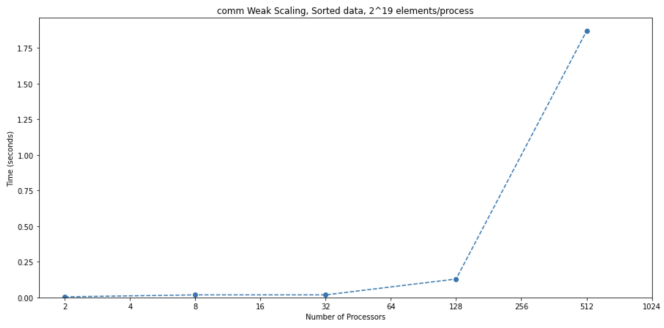


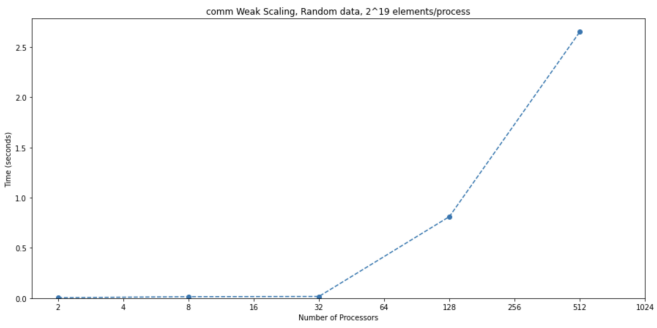
The L1 cache misses are relatively consistent across the different input types. They double when doubling the input size, and slightly decrease for more processors. They look most similar to the cache miss graphs for comp, since that is where about 90% of the L1 misses come from.
The L2 graphs are essentially the same as the comm graphs, since the computation accounts for less than a trillionth of the L2 misses.

Merge Sort

Communication

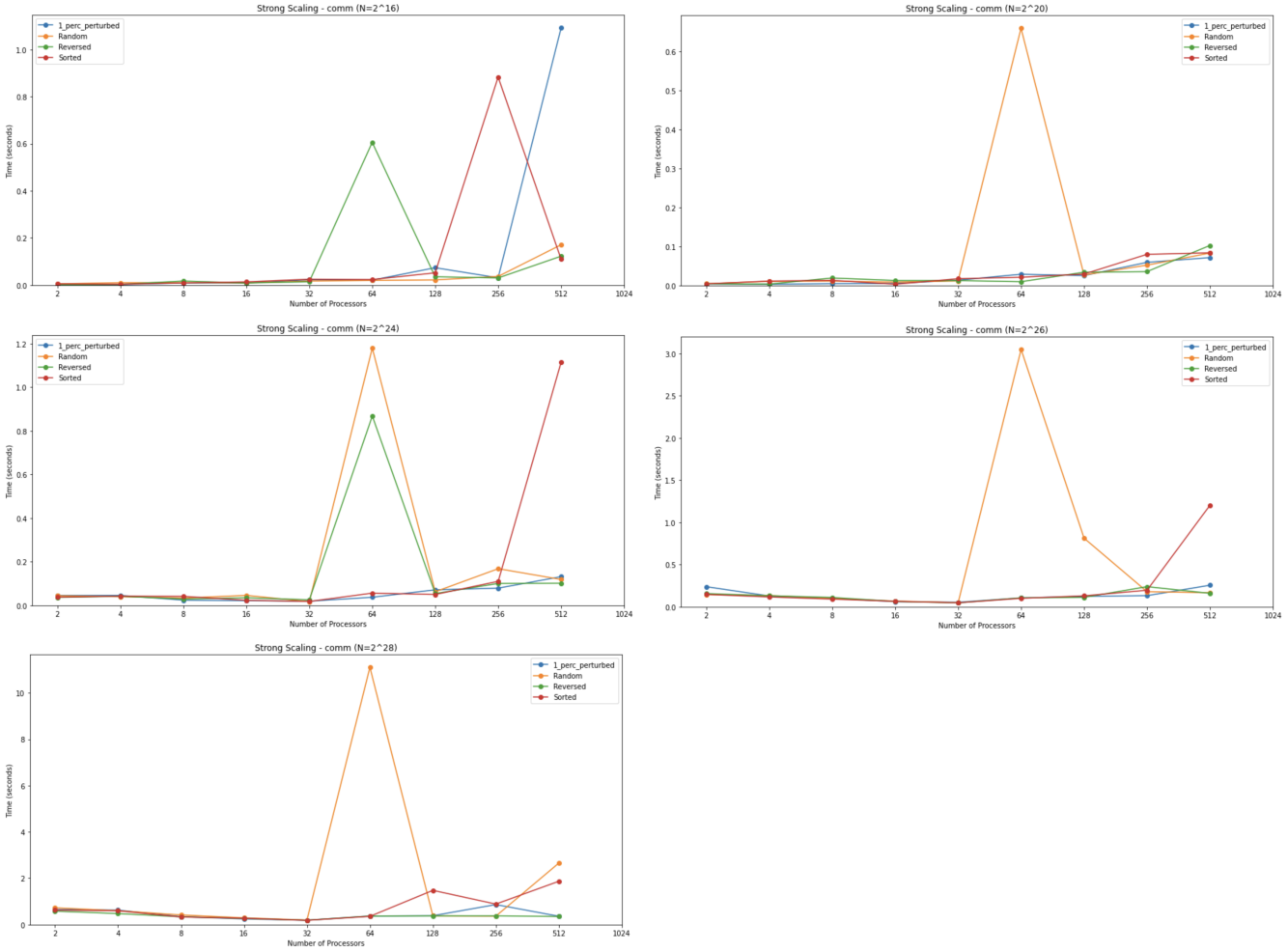
Weak Scaling





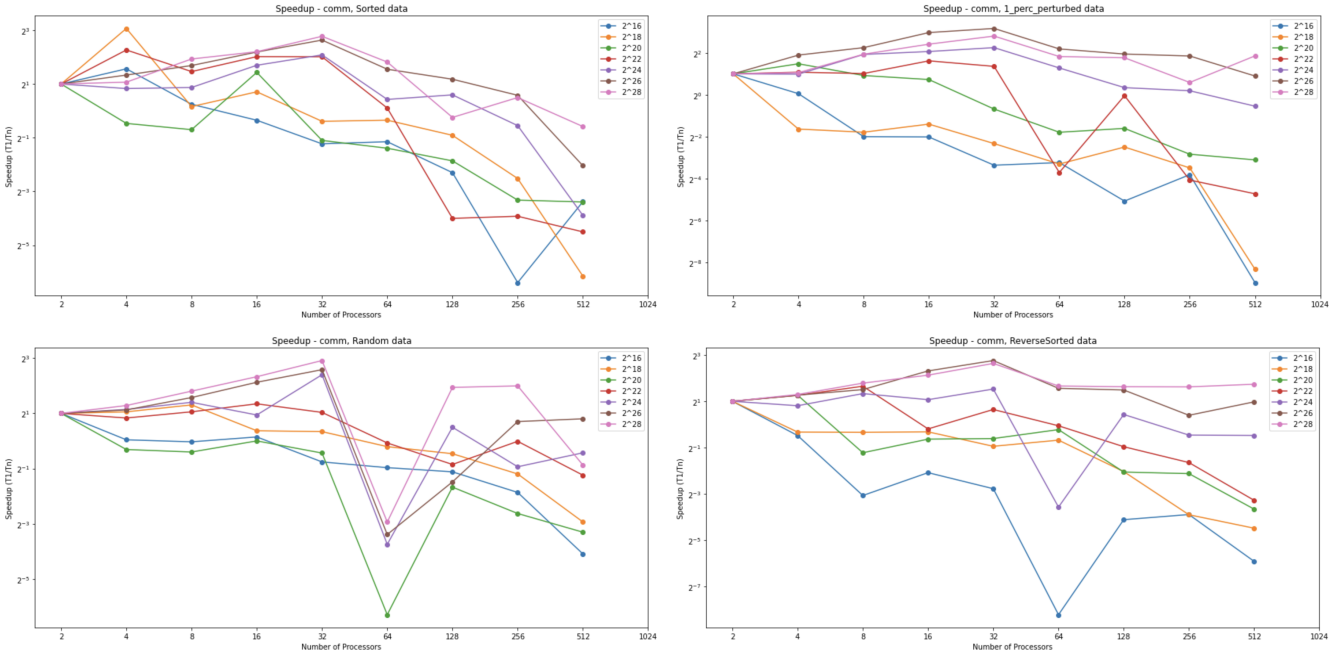
The weak scaling results demonstrate the implementation's ability to handle increased workload as processor count grows, with each processor maintaining a constant problem size. The graphs show relatively stable performance up to 64 processors, after which communication overhead becomes increasingly significant. All four input types (sorted, 1% perturbed, random, and reverse sorted) exhibit similar patterns, with communication time rising sharply beyond 64 processors. This behavior indicates that the communication overhead grows disproportionately to the number of processors, suggesting that the implementation's efficiency decreases at larger scales despite maintaining constant per-processor workload. A large reason for this communication overhead is likely to be the use of the MPI_Gather call.

Strong Scaling



For strong scaling, the most notable feature is the spikes around 64 processes, espcially for the random input type. This spike can be attributed to the switch from intranode communication to inter-node communication that happens when using 64 processes, resulting in incresed communication overhead and latency. All the graphs show increasing communication times with larger input sizes and and increased number of processes, likely as a result of the increased coordination required between the processes. Overall, the graphs suggest that the implementation's scaling efficiency is highly dependent on input distribution, with random data presenting particular challenges for load balancing and communication patterns.

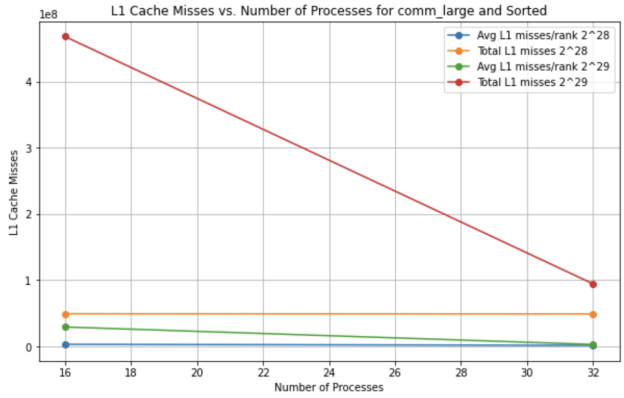
Speedup

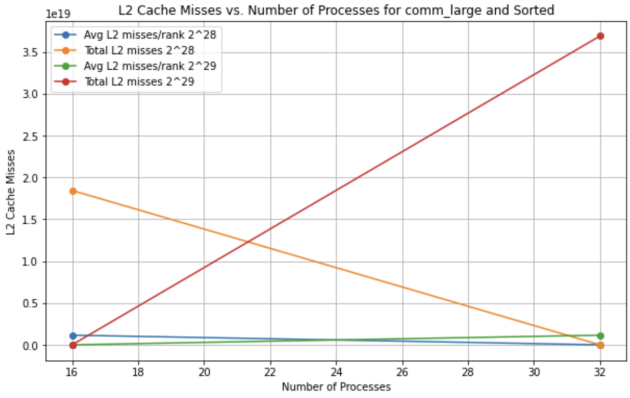


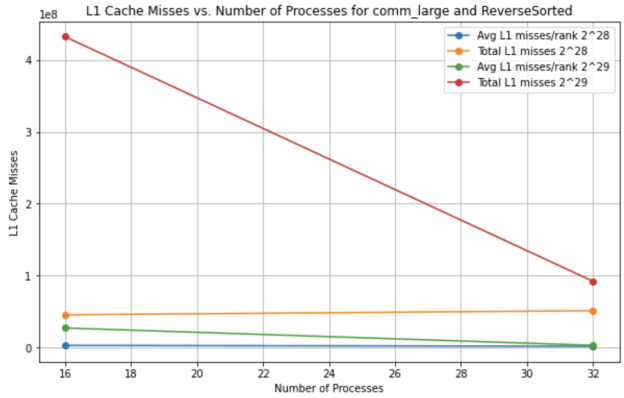
The speedup graphs reveal diminishing returns as processor count increases, with optimal performance typically achieved between 16-32 processors. All input types show declining speedup at higher processor counts, though reverse sorted data maintains better speedup compared to other input distributions. The graphs

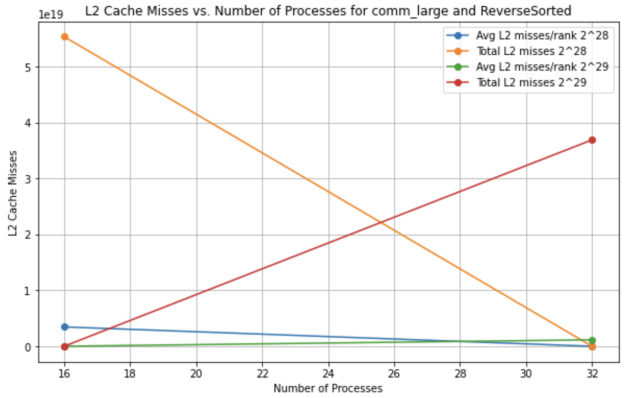
demonstrate parallel computing challenges, with Amdahl's Law limitations and communication overhead becoming increasingly significant at higher processor counts. The varying performance across different input types suggests that the implementation's efficiency is strongly influenced by data distribution, with more predictable distributions achieving better speedup characteristics than random or sorted data. The various dips in speedup can be attributed to inconsistent efficiency.

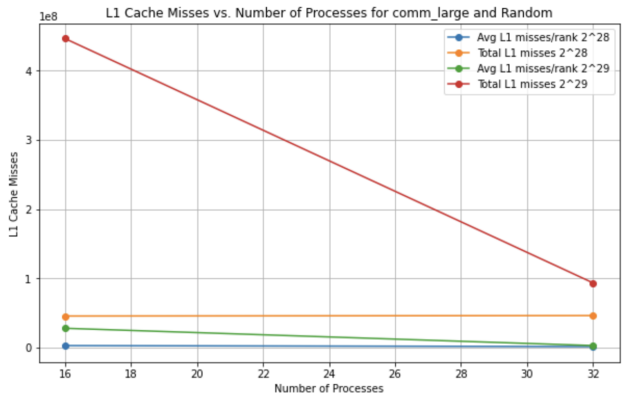
Cache Misses

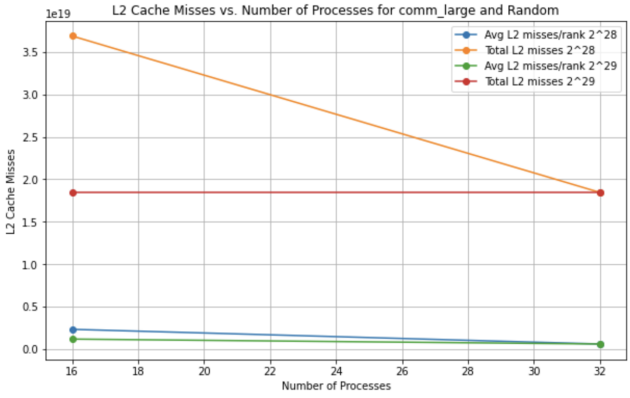








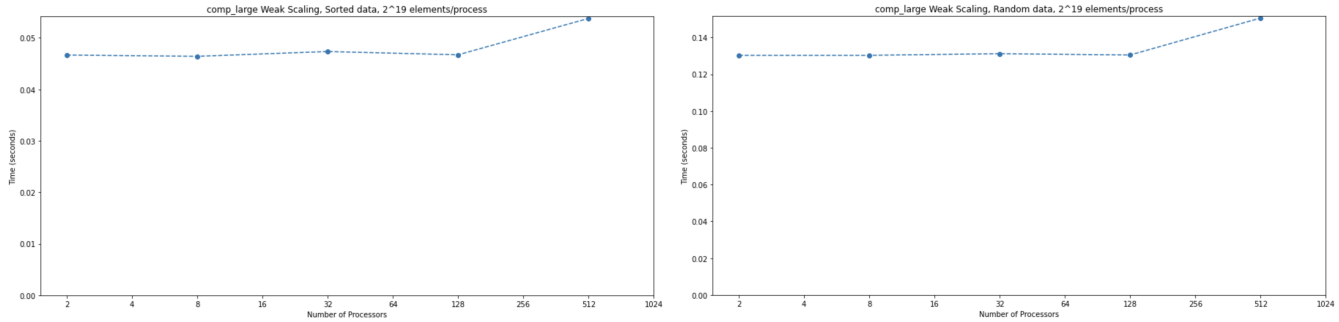




The cache miss graphs show distinct patterns between L1 and L2 cache behavior. L1 cache misses demonstrate a consistent decrease in total misses as process count increases, while average misses per rank remain relatively stable, indicating effective data locality at the local processing level. L2 cache behavior shows more complex patterns, particularly for larger data sizes (2^29), where total misses increase while per-rank averages remain stable. The crossover patterns observed in the ReverseSorted data suggest memory hierarchy bottlenecks and potential cache thrashing at larger data sizes. This cache behavior indicates that memory access patterns significantly impact performance, particularly when scaling to larger problem sizes.

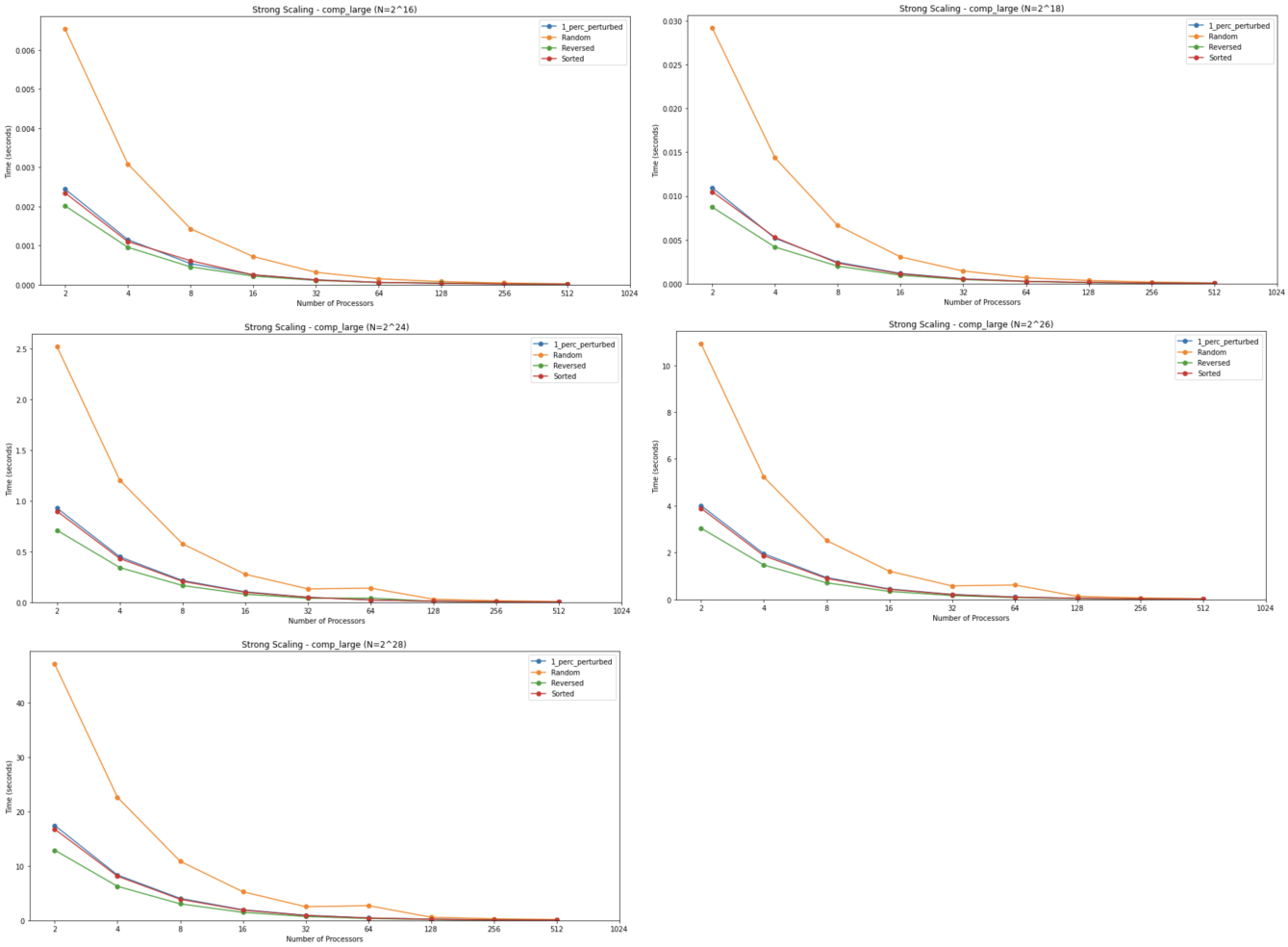
Computation

Weak Scaling



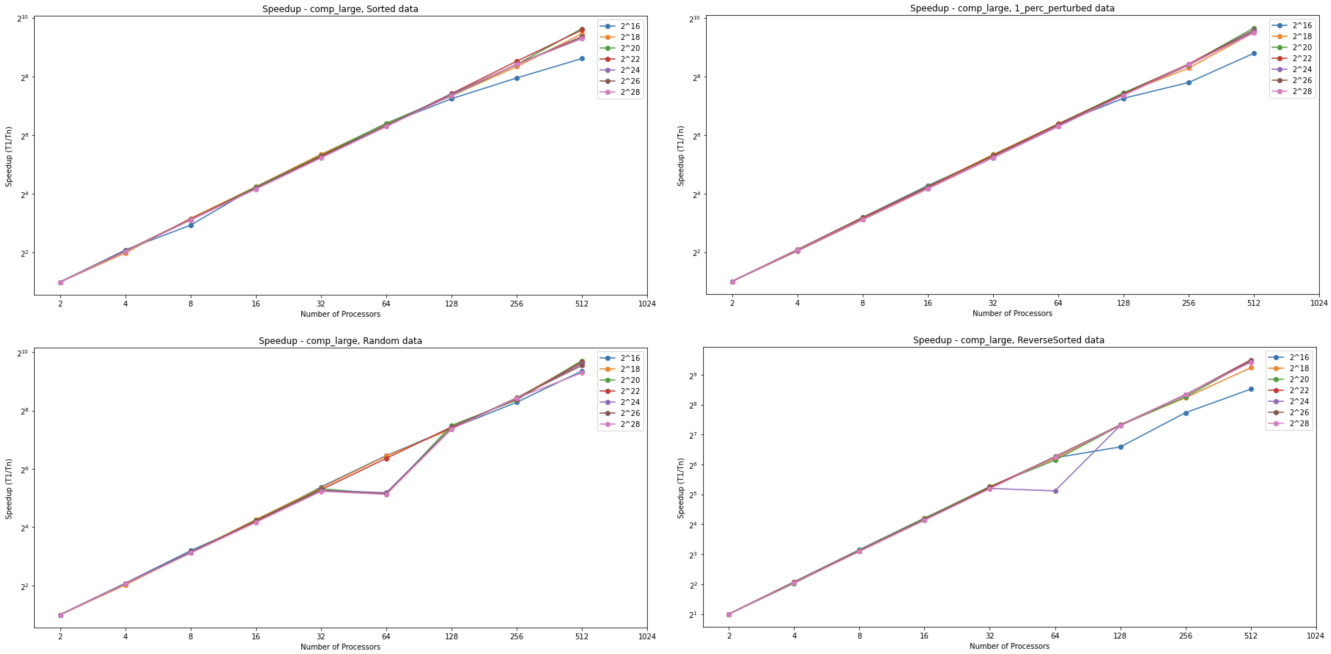
The weak scaling results demonstrate the algorithm's ability to maintain relatively consistent computation performance as the problem size and number of processors increase proportionally. For sorted, 1% perturbed, and reverse sorted data, the execution times remain fairly stable until reaching higher processor counts (256-512), where slight increases suggest growing communication overhead. Random data exhibits higher execution times and more variance, indicating that unordered data patterns create additional computational complexity. The overall trend suggests good but not perfect weak scaling, with performance degradation primarily occurring at higher processor counts likely synchronization overhead.

Strong Scaling



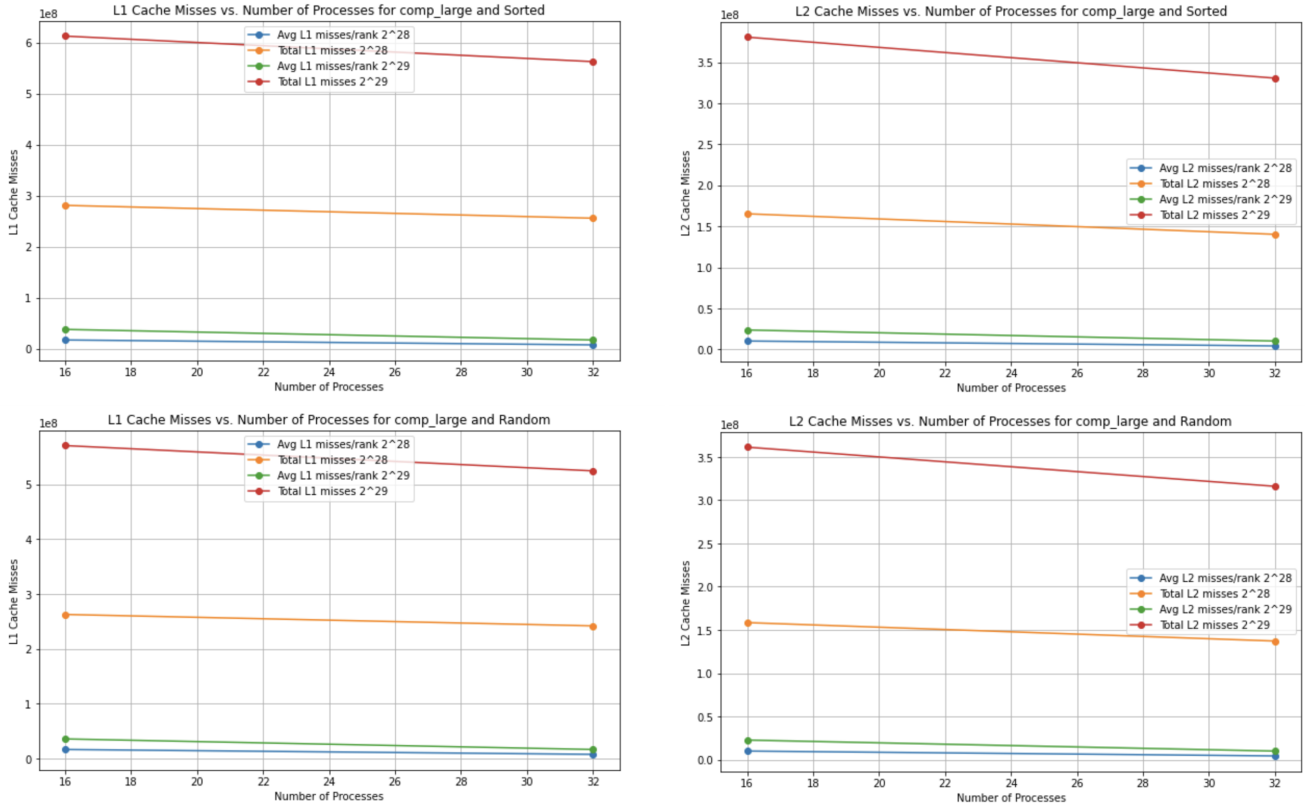
The strong scaling graphs reveal how execution time decreases with increased processor count for fixed total problem sizes (2^16 to 2^28). Random data consistently shows the poorest performance with significantly higher execution times compared to other data patterns, likely due to unpredictable memory access patterns and more complex merge operations. The curves demonstrate good scaling up to around 64 processors, after which diminishing returns become evident. This efficiency drop at higher processor counts is attributable to three main factors: increasing communication overhead, decreasing computation-to-communication ratio as work per processor shrinks, and potential load imbalances. Larger problem sizes (2^24 and above) exhibit better strong scaling characteristics due to maintaining a more favorable computation-to-communication ratio even at higher processor counts.

Speedup



The speedup graphs illustrate good speedup scaling up to 64-128 processors, with sorted and 1% perturbed data demonstrating the most consistent improvements. A notable dip in speedup occurs around 32-64 processors, particularly for random and reverse sorted data, likely due to a combination of increasing communication overhead and load imbalance effects. Larger problem sizes (2^24 to 2^28) achieve better speedup ratios, approaching ideal scaling more closely than smaller problems, demonstrating that the implementation benefits from larger datasets that better amortize parallelization overhead and maintain higher computation-to-communication ratios.

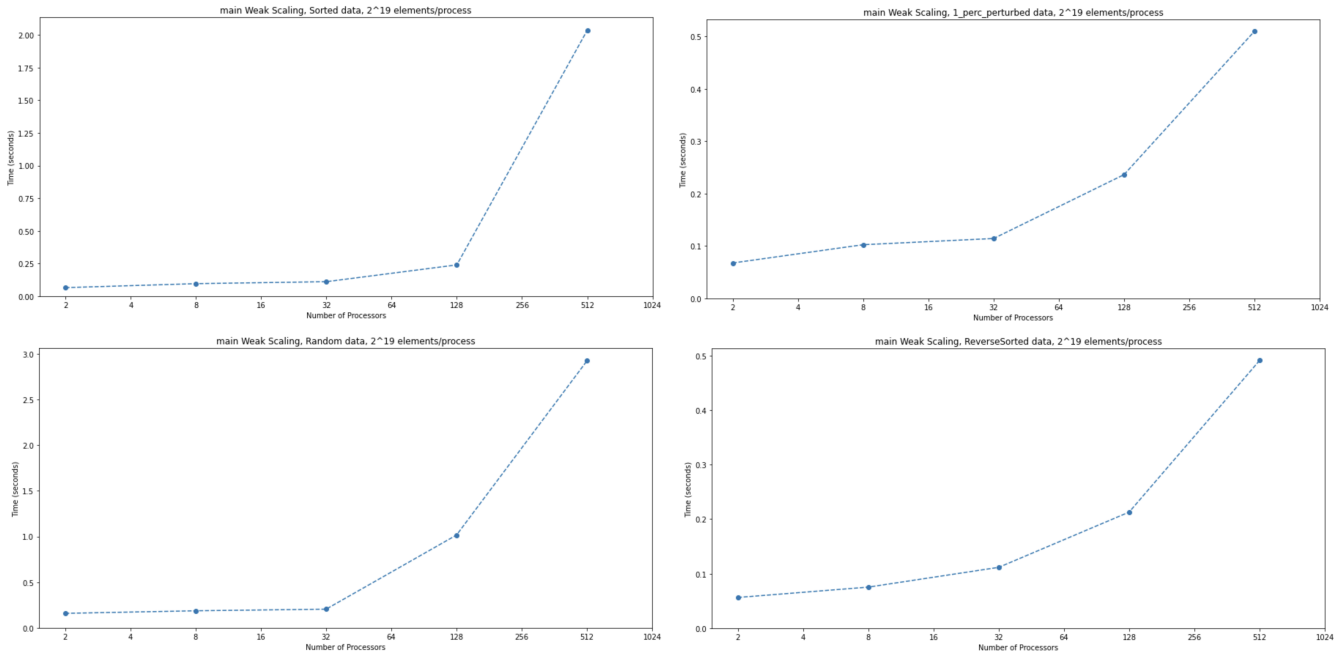
Cache Misses



The cache miss analysis for both L1 and L2 caches shows a gradual decrease in misses as the number of processors increases, which is expected as each processor handles a smaller portion of the data. The graphs demonstrate higher total misses for the larger dataset, but the average misses per rank decrease consistently with more processors. Interestingly, both sorted and random data exhibit similar cache miss patterns, suggesting that the cache behavior is more influenced by data size than data arrangement. The declining trend in cache misses with increased processor count indicates better cache utilization as local data sets become smaller, though this improvement doesn't necessarily translate to proportional performance gains due to other overhead factors.

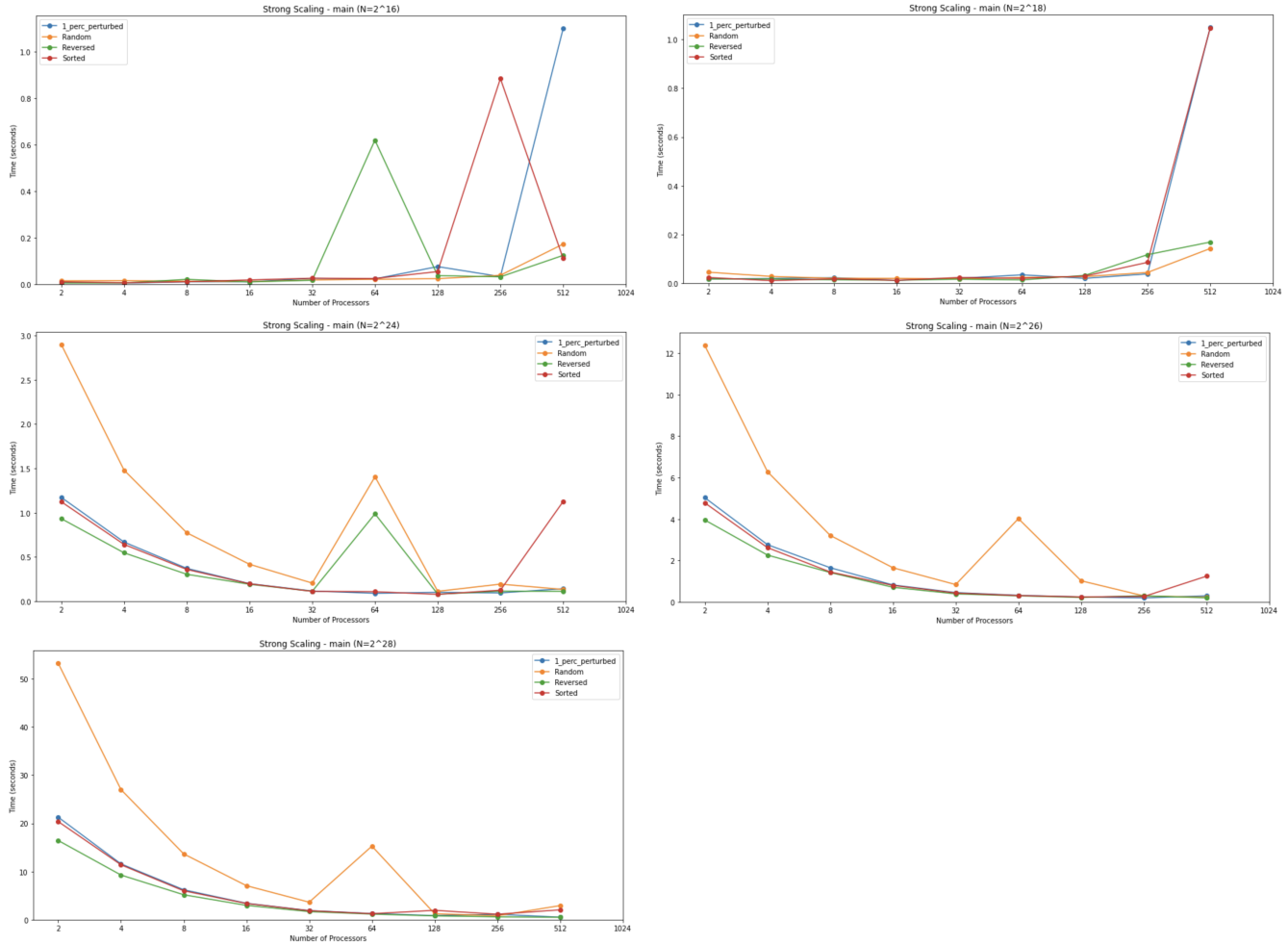
Main

Weak Scaling



The weak scaling results show that the parallel merge sort implementation struggles to maintain constant execution time as both problem size and processor count increase proportionally. This is evidenced by the upward trend in execution times, particularly pronounced after 128 processors. All input types (sorted, random, 1% perturbed, and reverse sorted) exhibit similar patterns, suggesting that the degradation is primarily due to increasing communication overhead and synchronization costs rather than data distribution characteristics. The steep increase in execution time at higher processor counts (256-512) indicates that the communication costs begin to dominate the computation benefits.

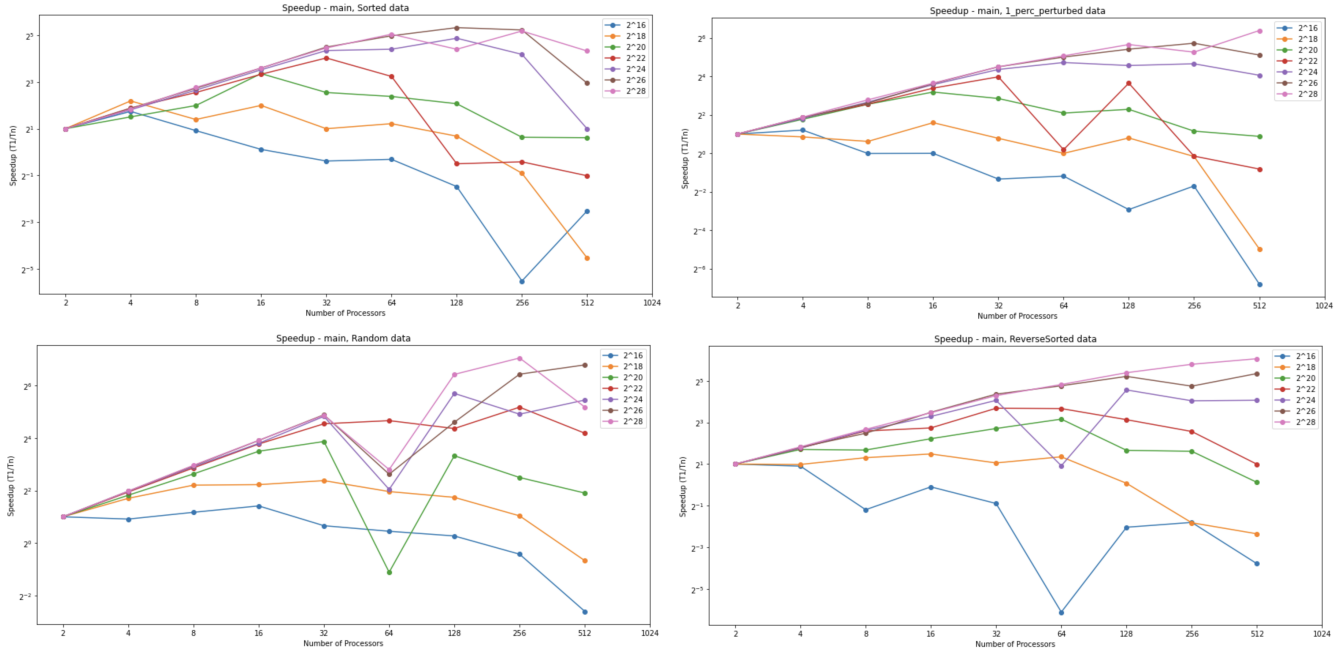
Strong Scaling



The strong scaling graphs reveal varying efficiency levels across different problem sizes (2^16 through 2^28). For smaller problem sizes (2^16, 2^18), the scaling is poor due to insufficient work per processor to offset communication overhead. Larger problem sizes (2^24 through 2^28) show better initial scaling but encounter efficiency drops at higher processor counts. Random data demonstrates consistently higher execution times but more predictable scaling behavior, while sorted and

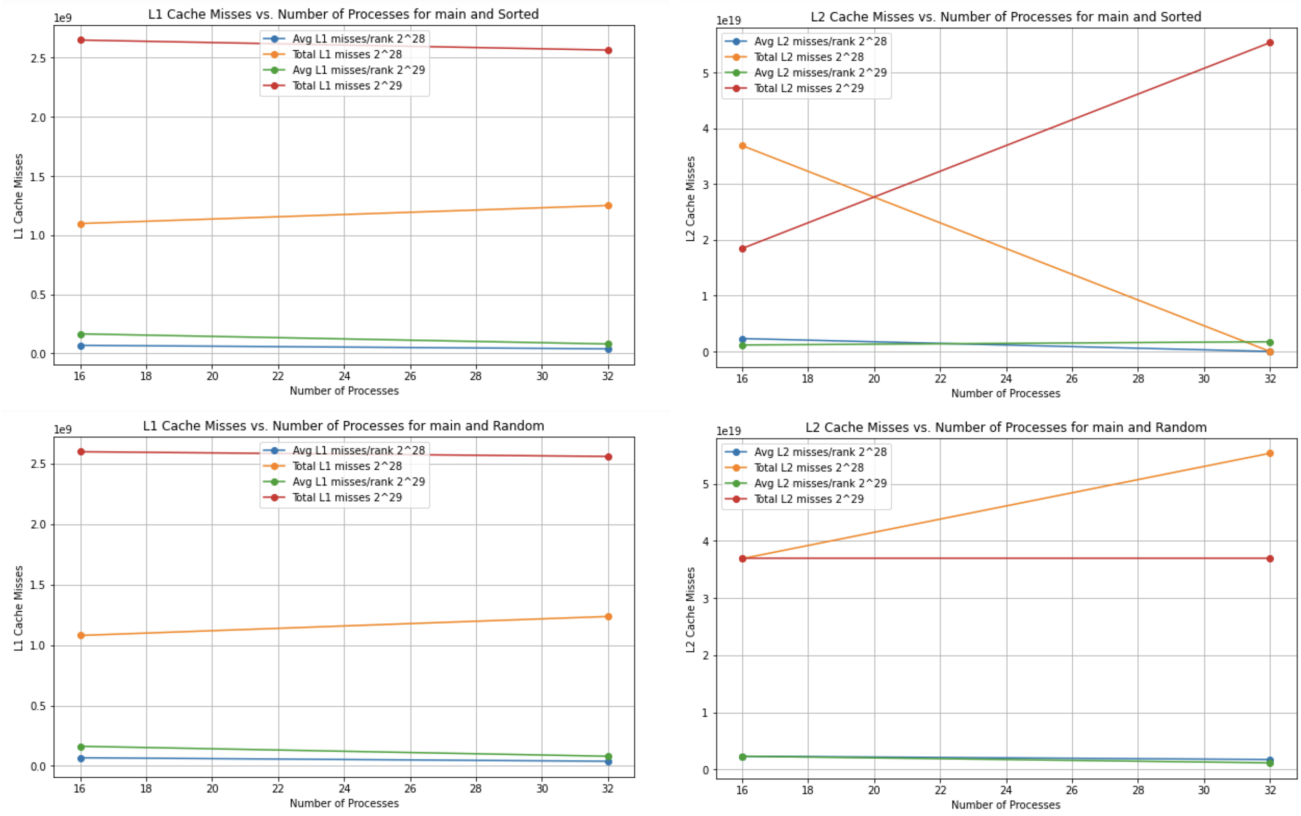
1-percent-perturbed data show irregular performance patterns with notable spikes at certain processor counts (particularly at 64 and 256 processors). These spikes likely result from cache alignment issues and load imbalance when dealing with already-sorted sequences.

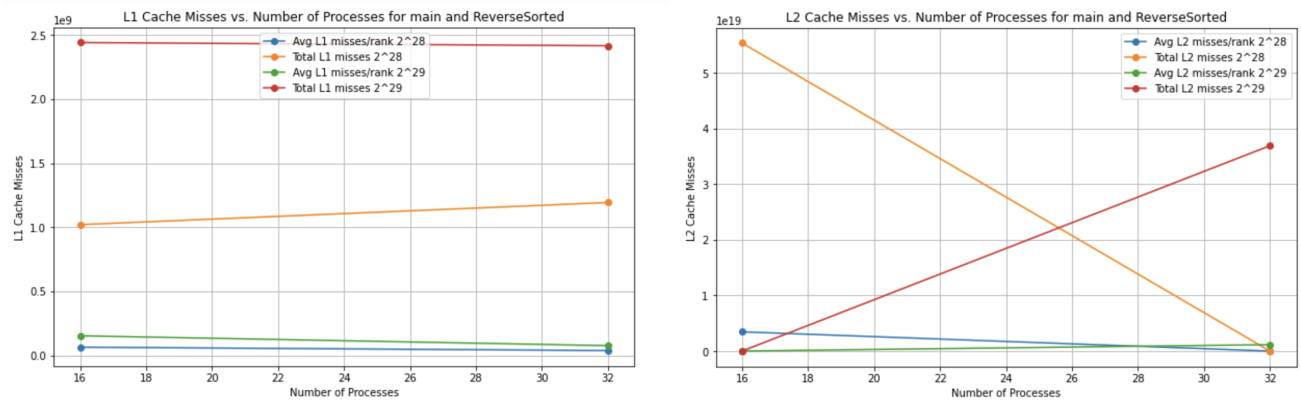
Speedup



The speedup graphs demonstrate sublinear scaling across all input types, with effectiveness varying by problem size and data distribution. Larger problem sizes (2^26, 2^28) achieve better speedup ratios due to their favorable computation-to-communication ratios. However, performance deteriorates sharply at high processor counts (256-512), indicating the dominance of communication overhead. Sorted data consistently shows lower speedup compared to random data, likely due to load imbalance issues.

Cache Misses

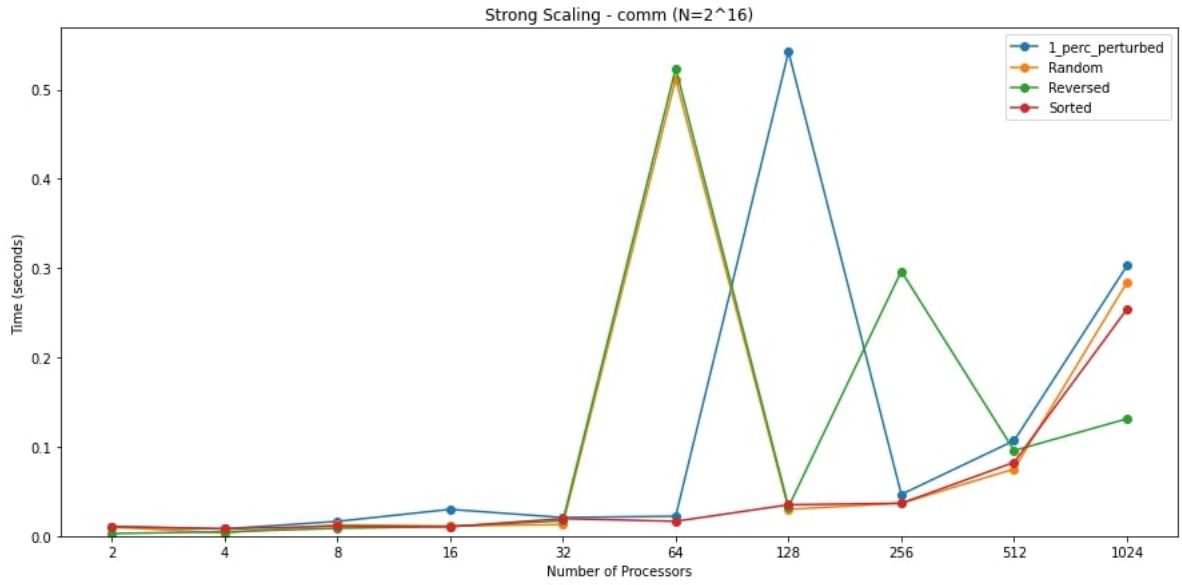


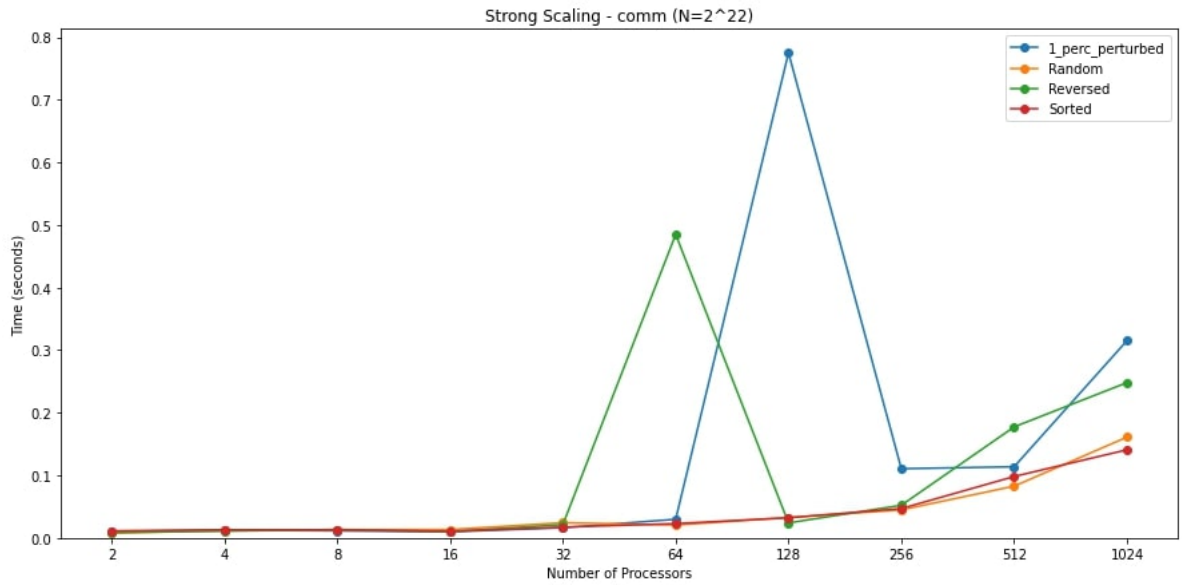


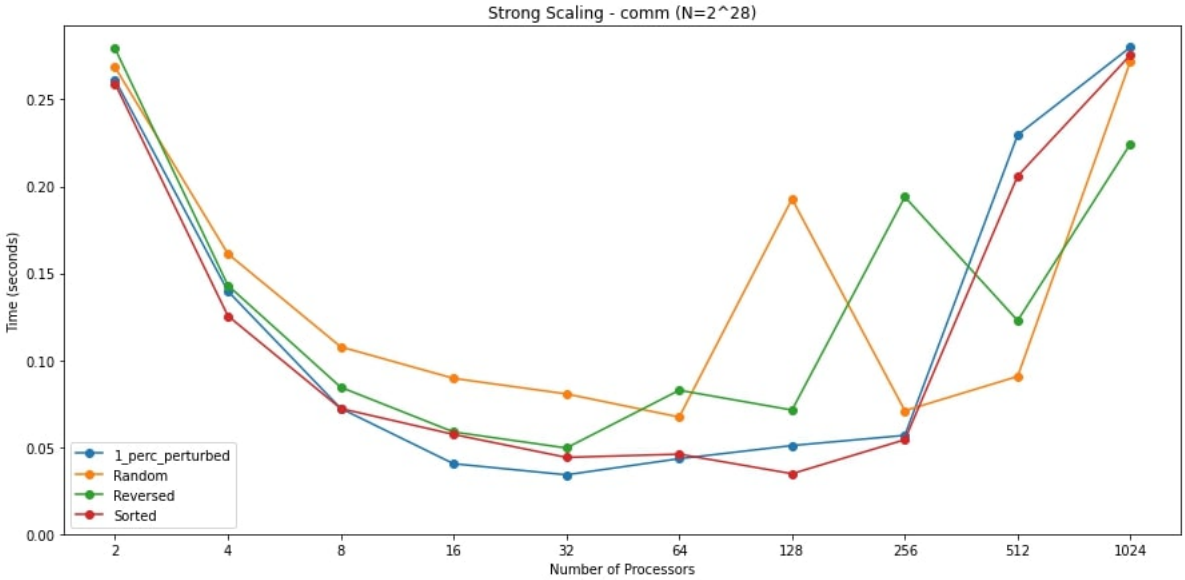
The cache behavior graphs highlight interesting patterns in both L1 and L2 cache performance. L1 cache misses remain relatively stable across processor counts, with slight variations between input types. However, L2 cache misses show more dramatic patterns, particularly for sorted data where there's a notable crossover pattern. Random data consistently shows higher L2 cache miss rates due to its unpredictable memory access patterns, while sorted data experiences more variable cache behavior. These cache patterns directly correlate with performance dips observed in the scaling graphs, emphasizing the critical role of memory access patterns in overall algorithm performance.

Radix Sort

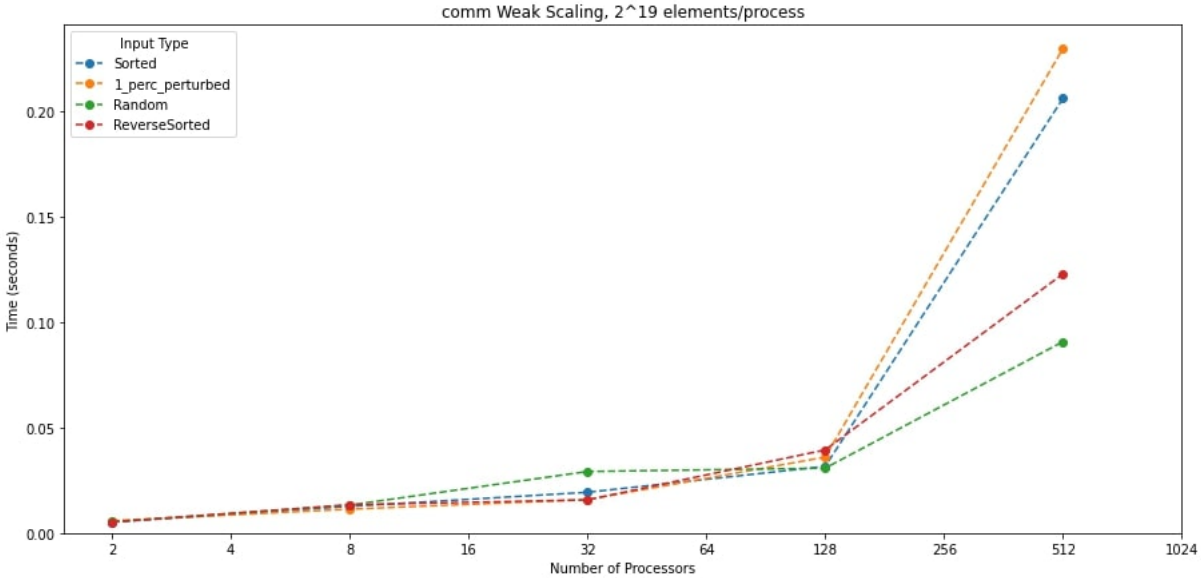
Communication



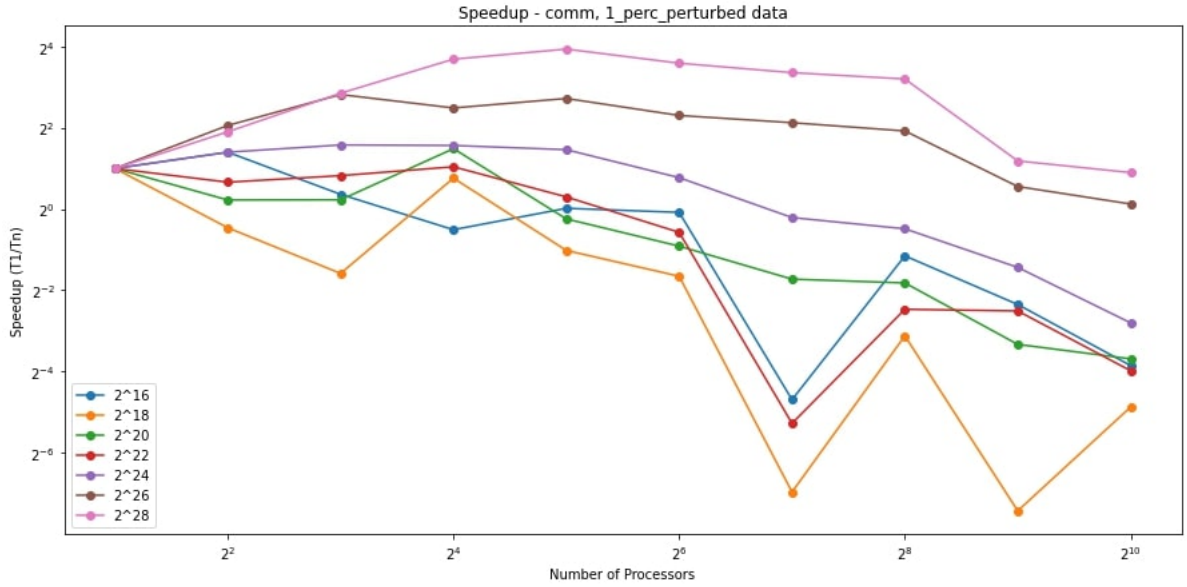


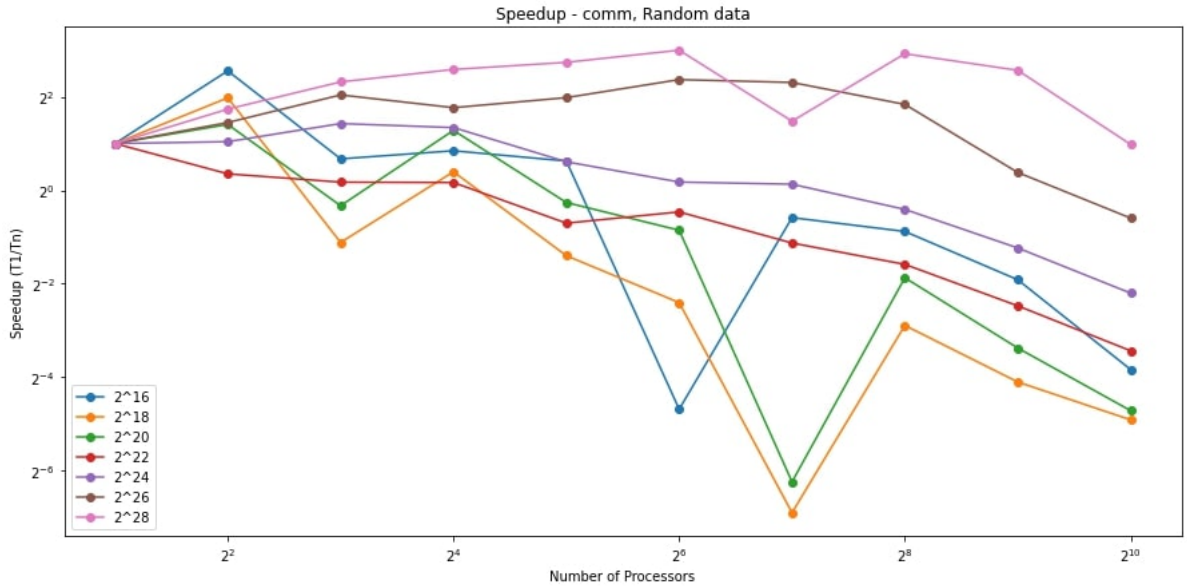


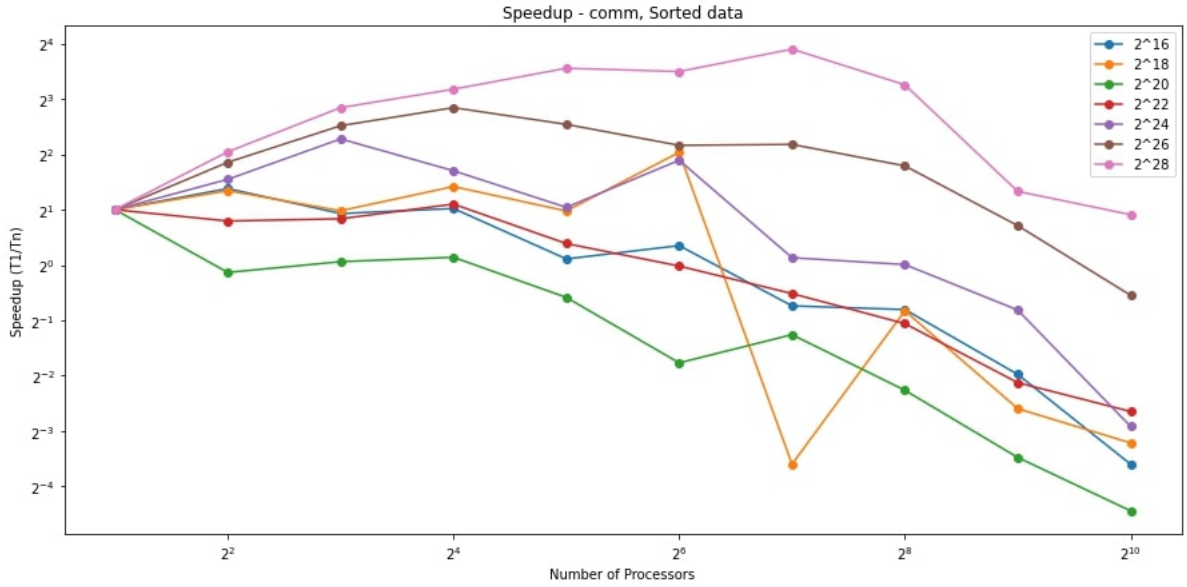
These are graphs of the average time for the strong scaling `comm_large` caliper section for the smallest, middle, and largest input sizes. Since the only time communication occurs in radix sort is to send all data from all processes to all processes using `MPI_Alltoall` or `MPI_Alltoallv`, there is no `comm_small` section. We see for the smallest input size that all input types are similar in time, with some spikes occasionally, which are likely due to unusual job runs. We see the same trend for the middle input size of 2^22. For the largest input size, we see that communication time actually decreases initially, hitting a minimum around 64 processes, then increasing again all the way to 1024 processes. The initial decrease is likely because so much data needs to be communicated between the 2 processes that it becomes inefficient. The increase at the end is likely because more processes need more time to communicate due to distance between the actual nodes and processors. We find a "sweet spot" around 64 processes to balance processor count and input size per process in communication time.

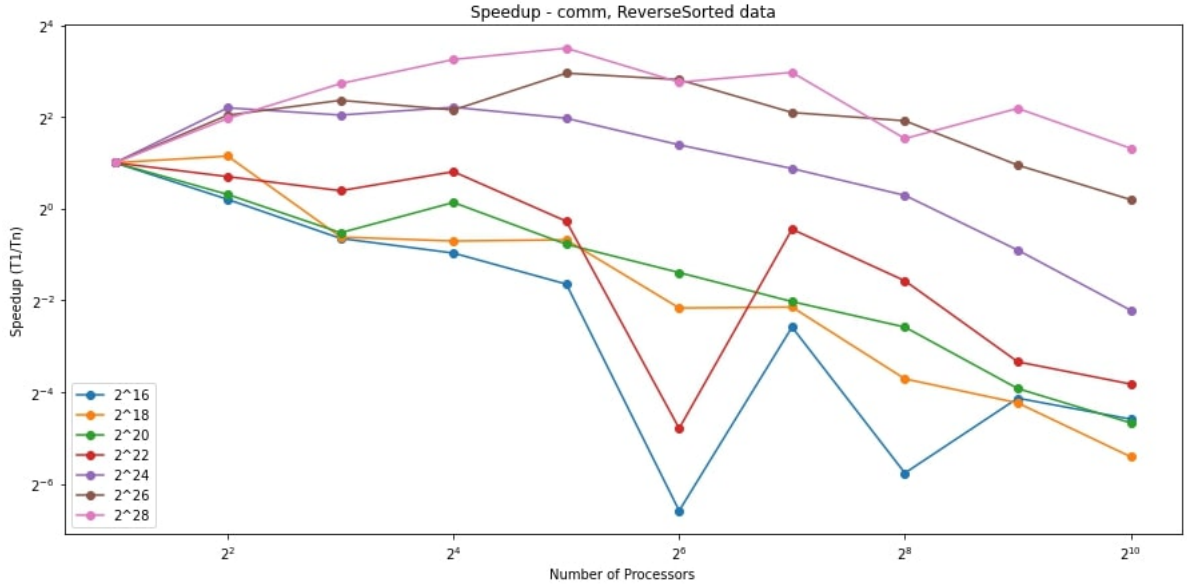


This is a weak scaling plot for communication with average time. We see that most input types are roughly equal in time until 512 processes, at which point the 1% perturbed and sorted data take the most communication time. This is likely because the data is initially in just a few buckets, so moving them around takes more time. Another possibility is that 512 processors have more variance in communication time, so these trends may just be due to chance.


Speedup - comm, 1_perc_perturbed data
Speedup (T1/Tn)
Number of Processors
2^16
2^18
2^20
2^22
2^24
2^26
2^28


Speedup - comm, Random data
Speedup (T1/Tn)
Number of Processors
2^16
2^18
2^20
2^22
2^24
2^26
2^28

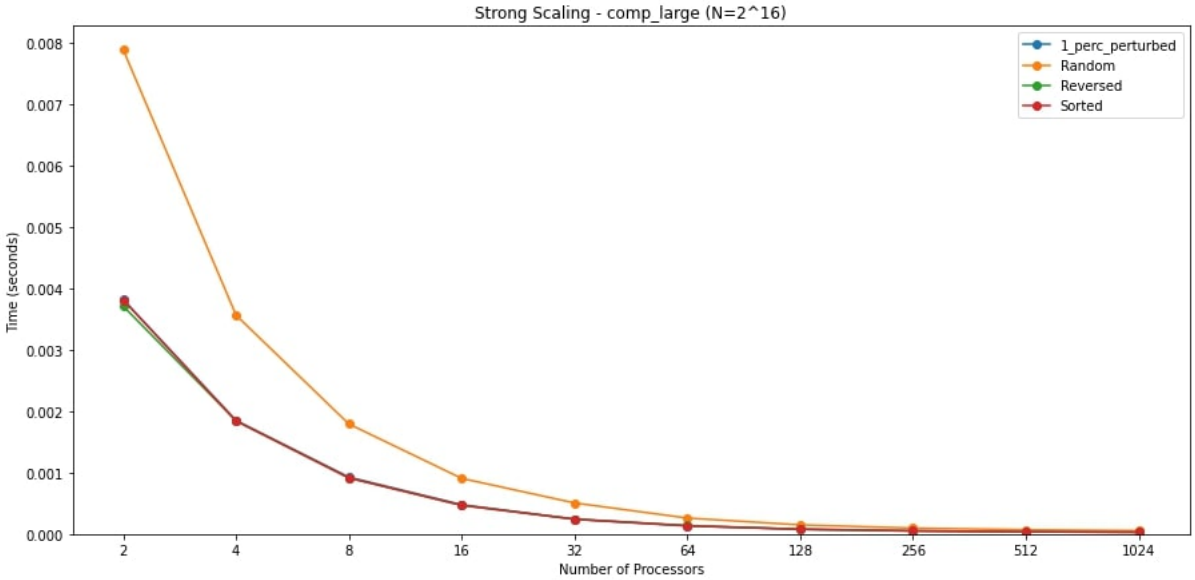


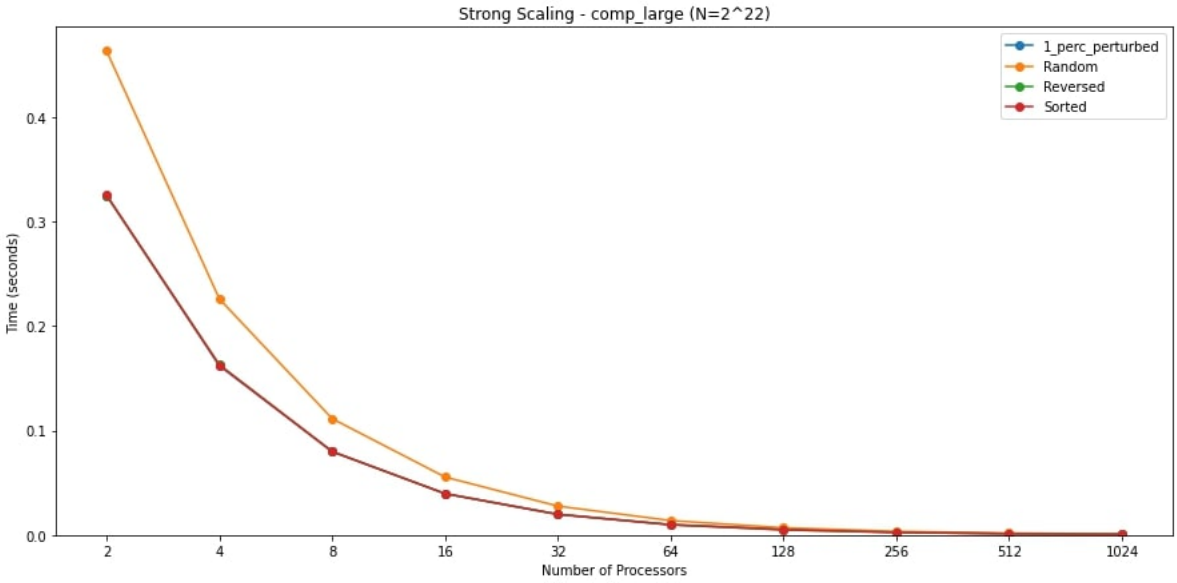


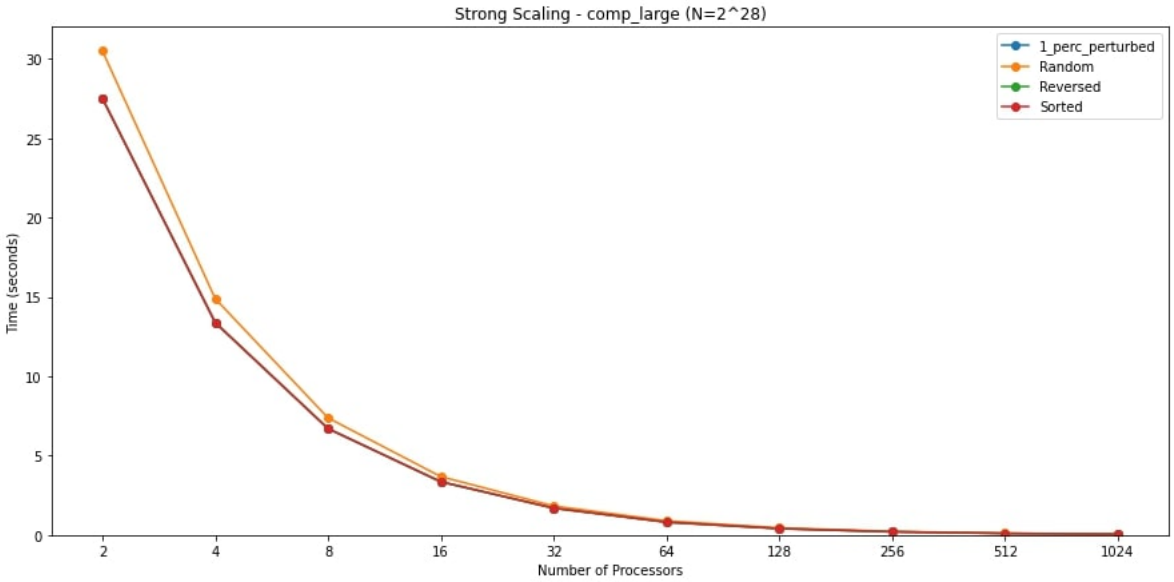
In terms of communication speedup, we see that all input types have roughly the same trends. The larger input sizes (2^24 and up) see a noticable speedup with number of processes, levelling off around 128 processes.

Communication time has a roughly linear relationship with the number of processes, regardless of input size and type. The magnitude of this relationship increases with input size, with larger inputs taking much longer to communicate than smaller ones. This makes sense as the algorithm uses `MPI_Alltoall` for communication, which is largely dependent on the number of processes performing the communication. There is not much difference between input types, which also makes sense because regardless of the data distribution the algorithm has to communicate its elements to all others.

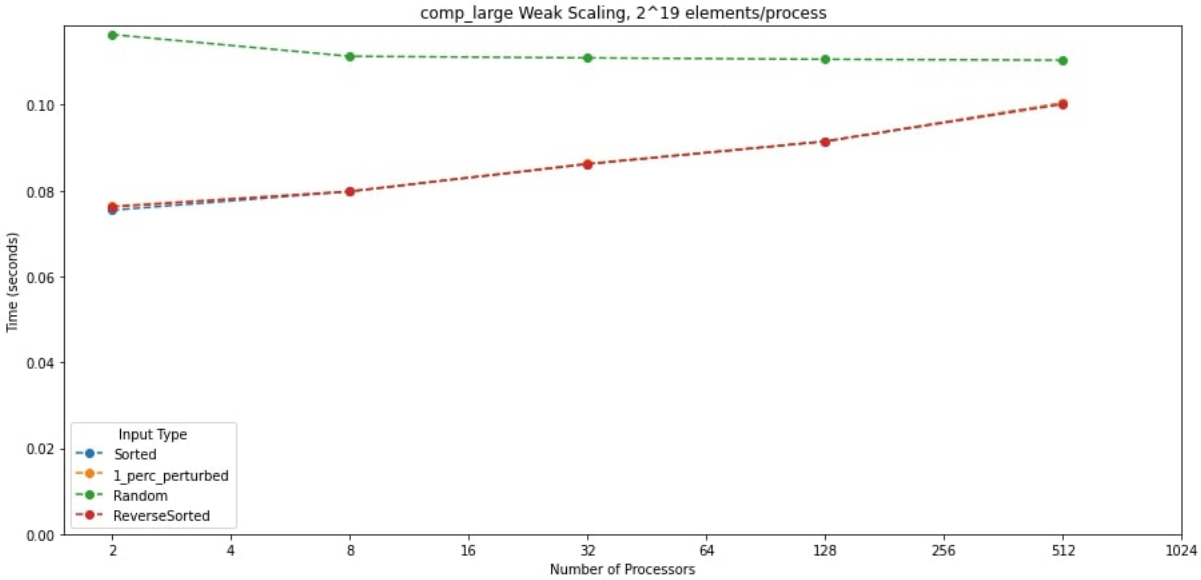
Computation



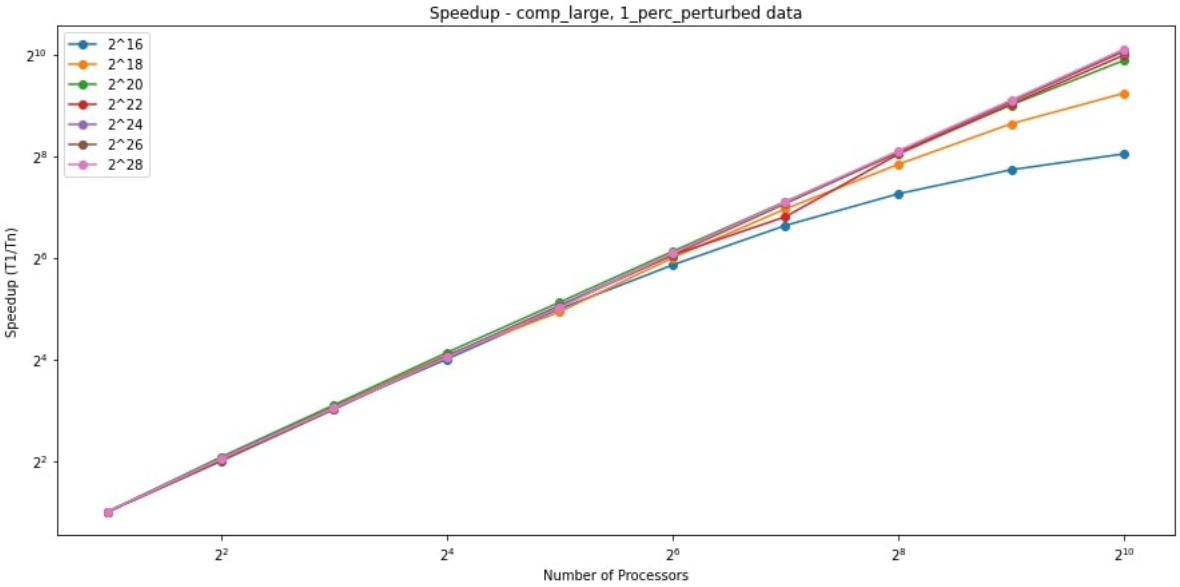


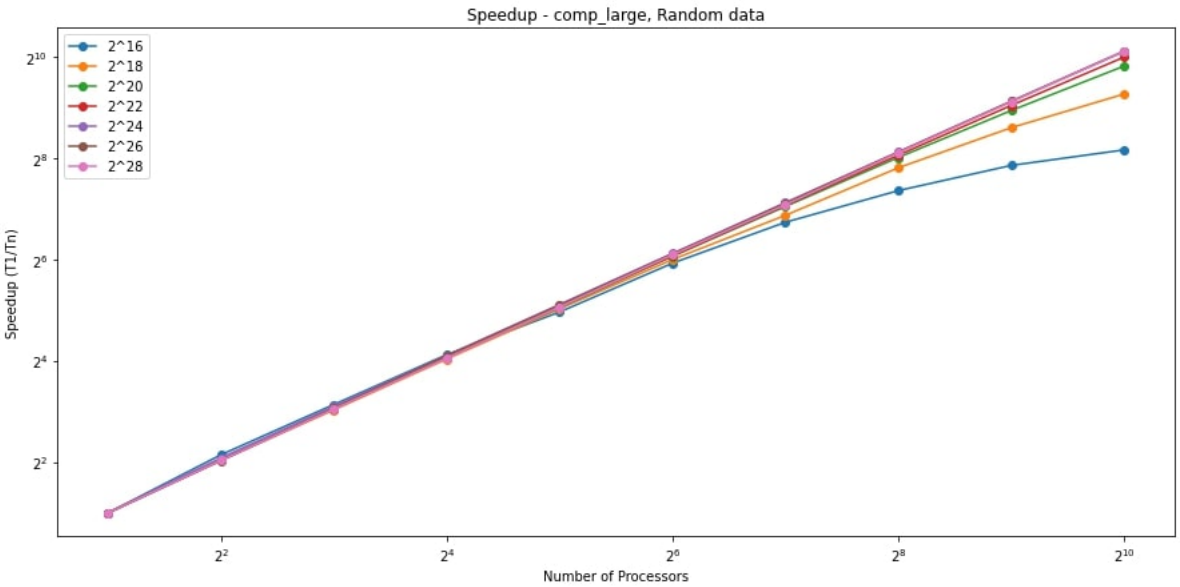


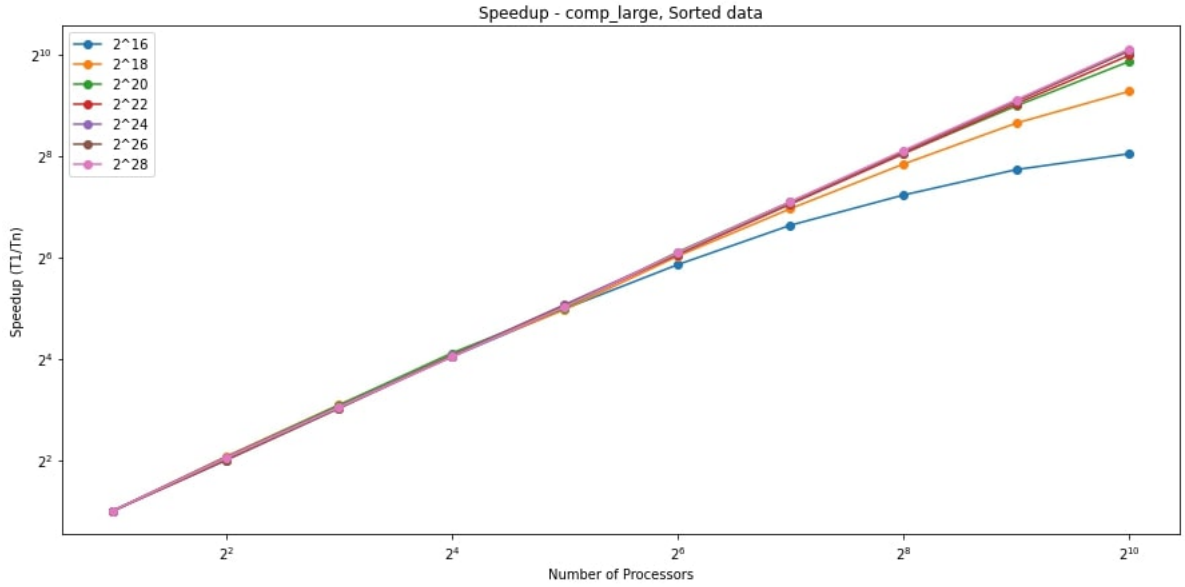
These are graphs of the average time for the `comp_large` caliper section using strong scaling with the smallest, middle, and largest input sizes. There is no `comp_small` caliper section since radix sort always sorts the entire local array, never just a portion of it. We can see that computation time does decrease logarithmically with the number of processes, which allows us to say that parallelization has made a significant difference in computation time. We can see that all input types have similar runtimes except for random, which takes a little longer. This is likely due to the distribution of the number of elements per process since the actual computation time should be almost the same regardless of input type. Radix sort is a noncomparison sorting algorithm, running in O(dn) time, where d is the max number of digits in the data and n is the number of elements. Since the max number of digits should be almost the same for each input type, the number of elements is the only thing that could cause this increase.

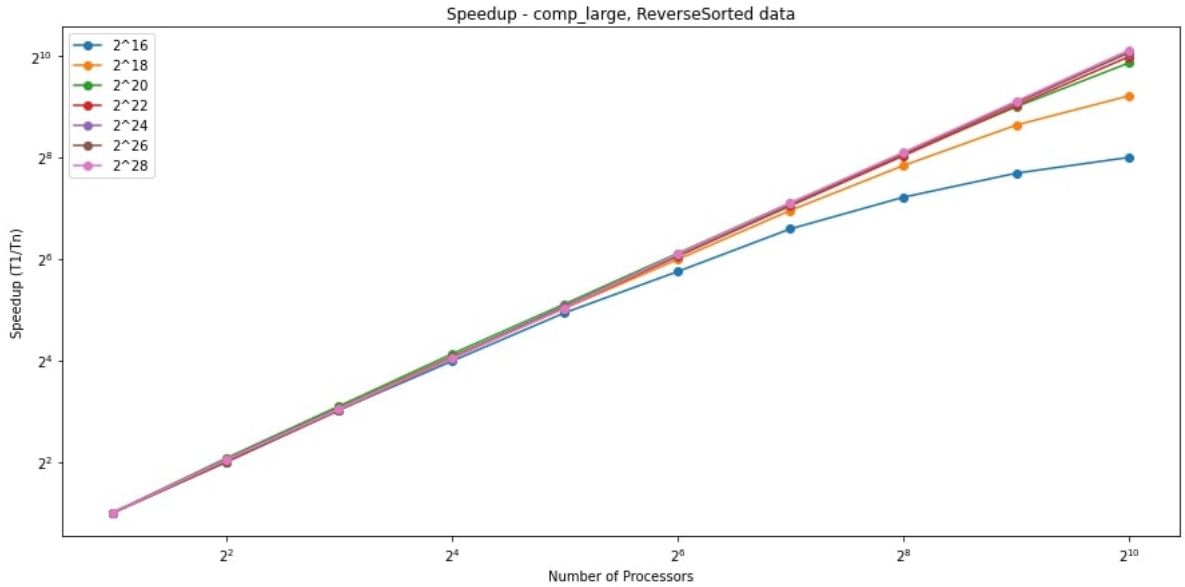


We can see that the weak scaling plot for computation stays relatively constant regardless of number of processes, increasing only slightly. Random again has the highest time, with the other 3 input types being almost the same. The consistency of this plot makes sense for the reason highlighted earlier for the strong scaling: radix sort only depends on the number of elements given to the process, so since this number is the same for all processes, their times reflect it.



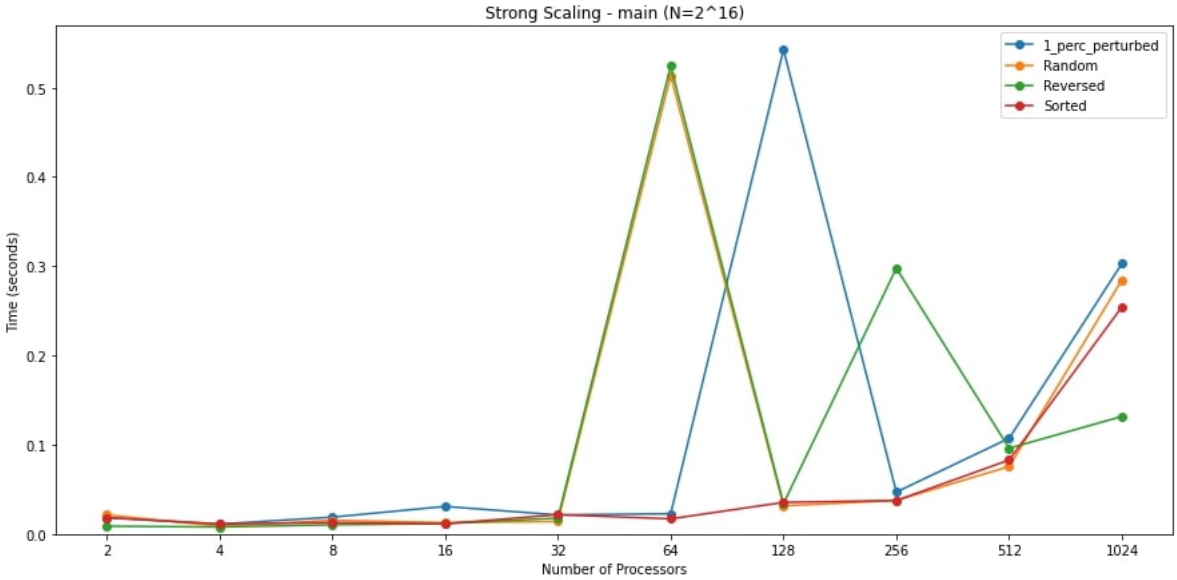


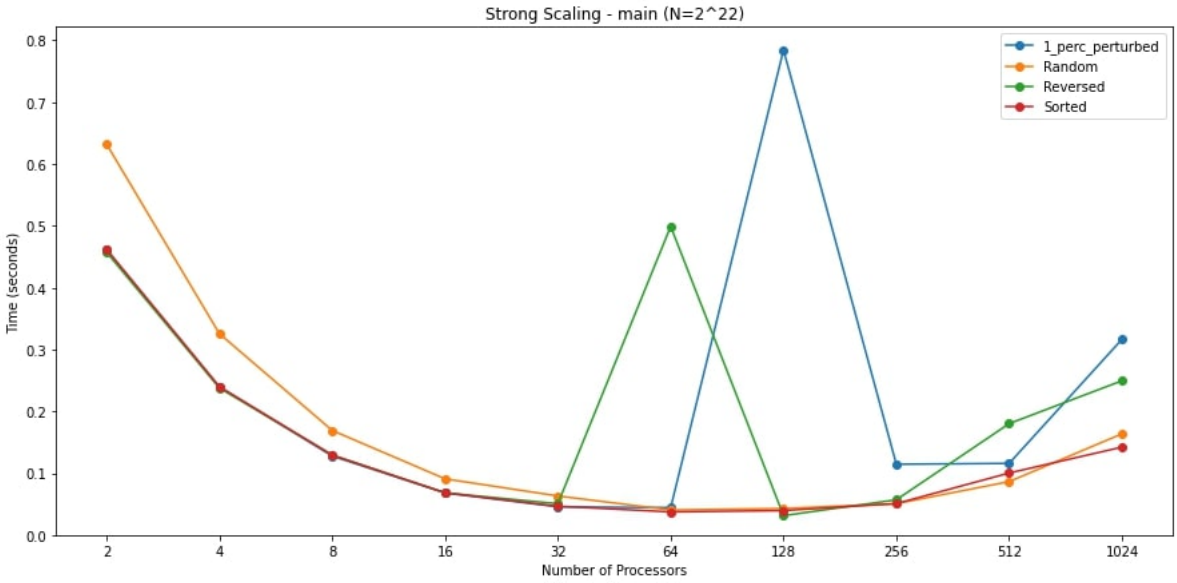


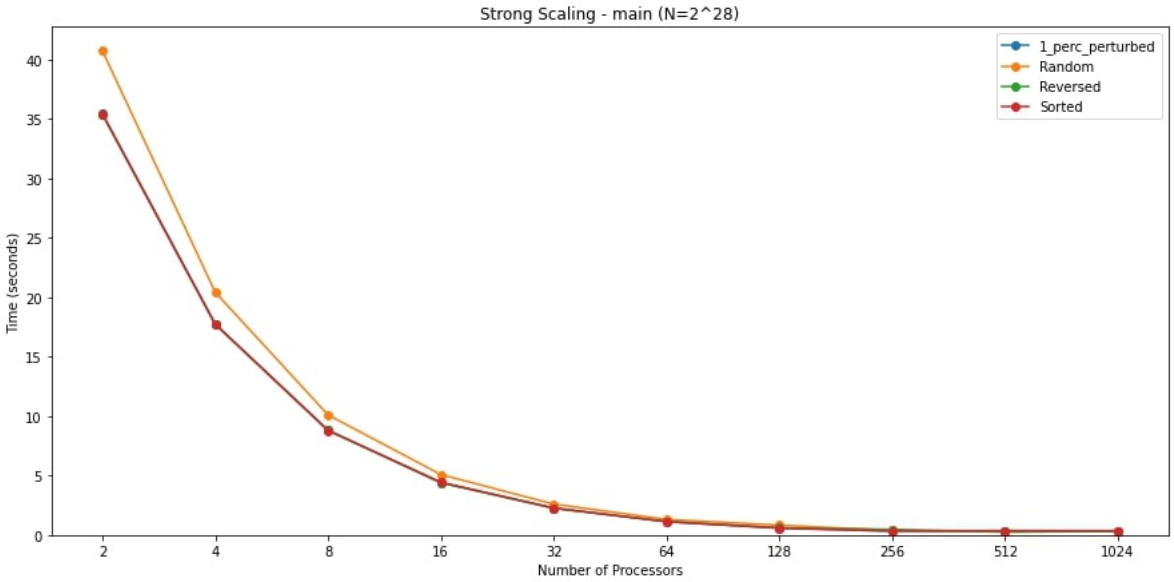


We can see that speedup for computation is the same for all input types. It is an almost diagonal line, dropping off only at 128 processes for the smallest input size. This means that we are always speeding up when there are more processes, which makes sense because the number of elements per process decreases, and that is the main thing affecting computation time. We see for smaller input sizes that we lose speedup around 128 processes, which is where there is no longer enough data per process to make further parallelization useful. For larger (2^20 and up) sizes, we still see significant speedup all the way to 1024 processes.

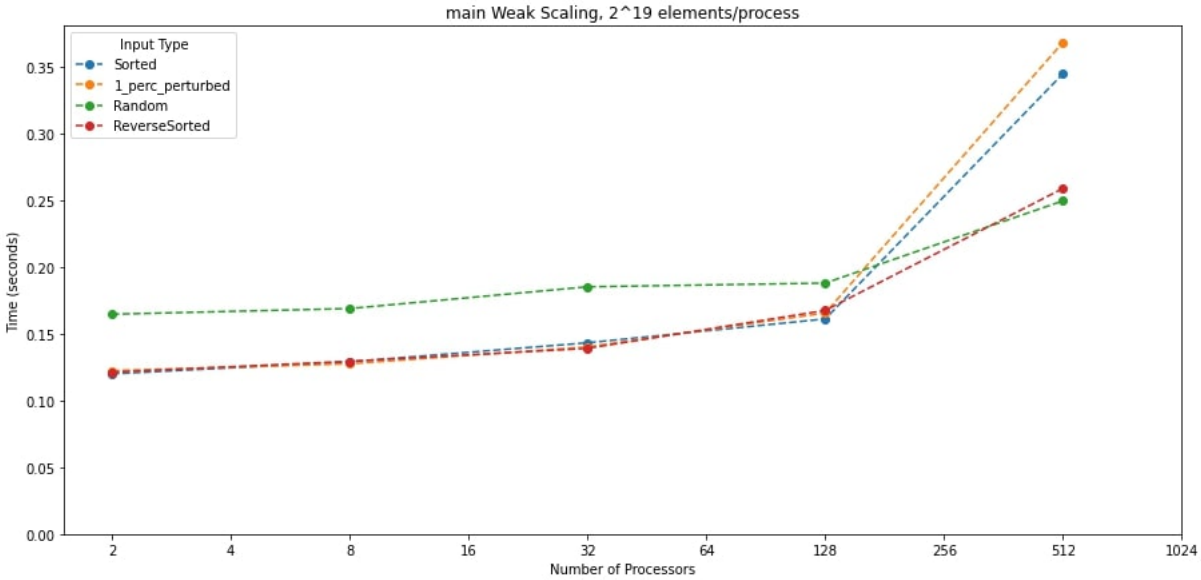
Main



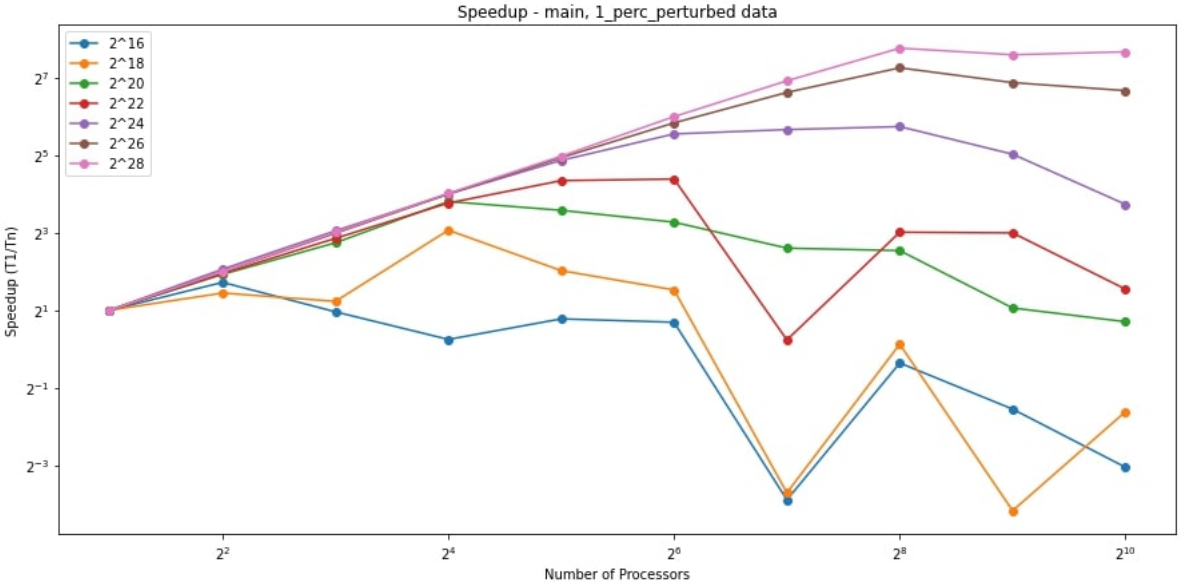


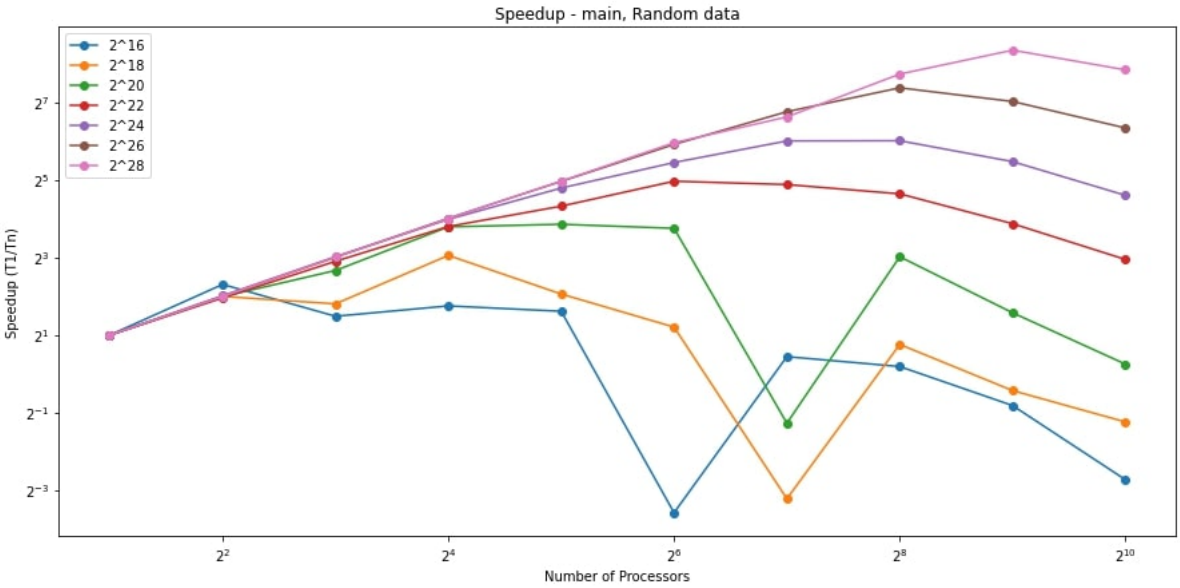


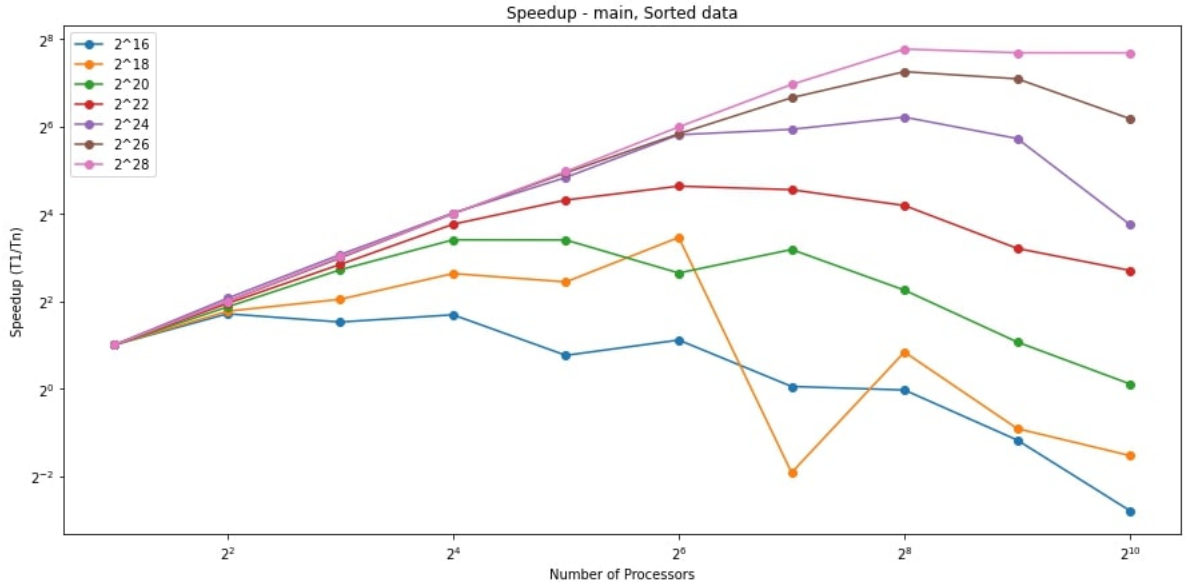
These are plots of full execution (main) time for radix sort for the smallest, middle, and largest input sizes. We see for the smallest one that it follows the communcation time strong scaling plot fairly closely, especially with the spikes in data occasionally. The middle size begins to combine the plots for communication and computation, with computation time being the bottleneck for small process counts and communication time taking over around 32 processes. The largest input size follows the computation plot almost exactly, with a slight increase at the end which can be attributed to the increase in communication time.

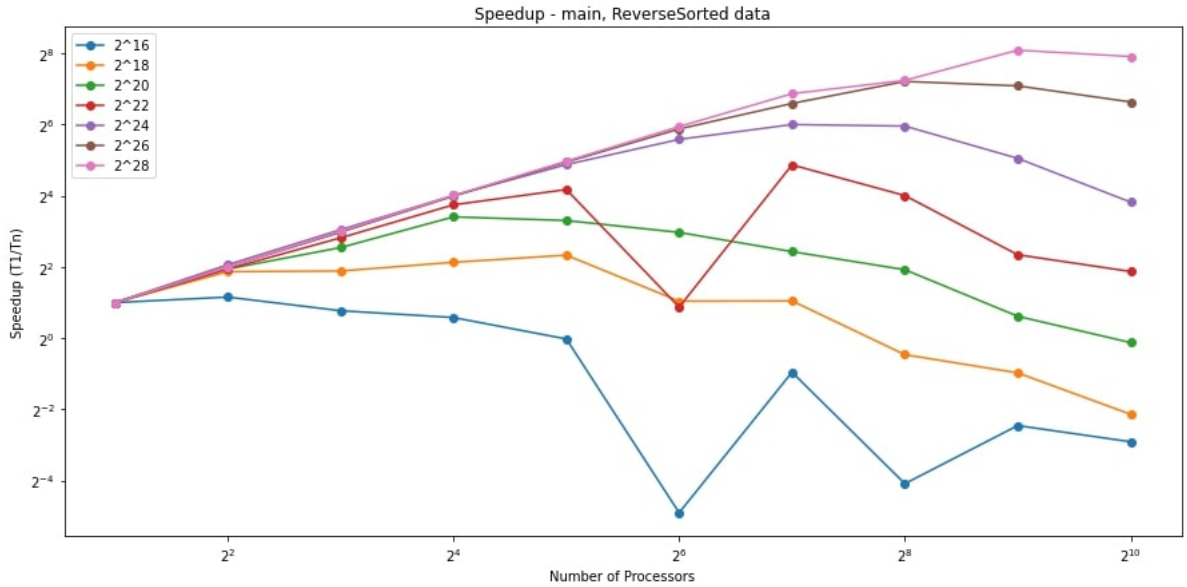


The weak scaling plot shows primarily the communication time increase at 128 processes, staying relatively constant before that.



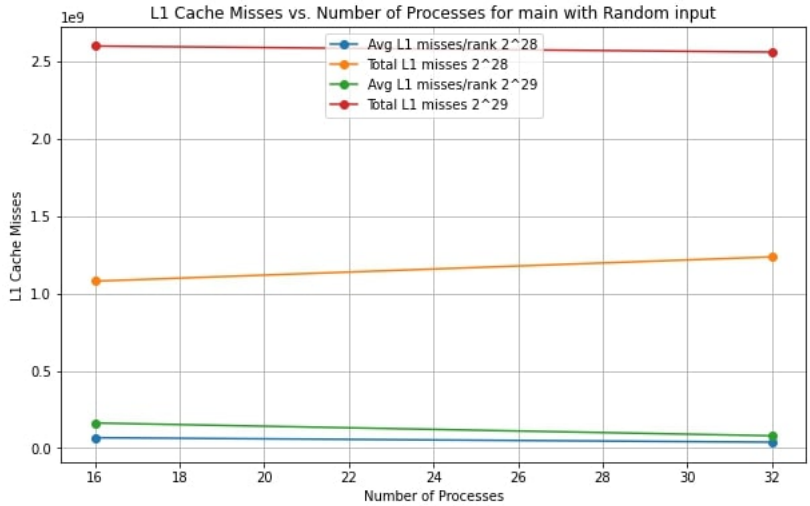


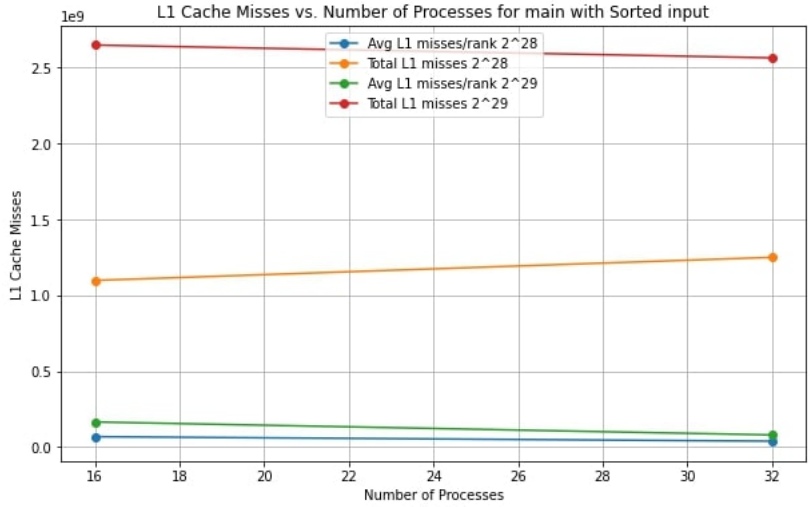


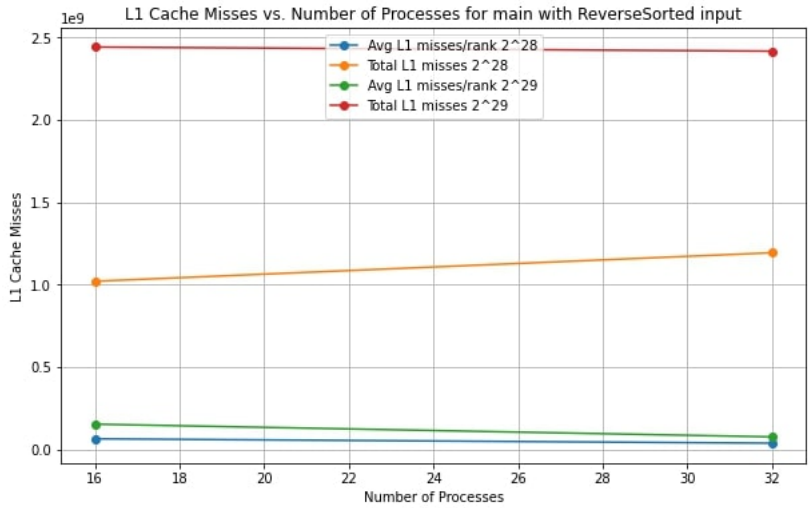


The speedup plots for main are also very similar regardless of input type. Speedup drops off in main at different times for each input size, with larger input sizes having drop offs at higher process counts. This makes sense because there is a balance between input size and process count which will actually make a difference.

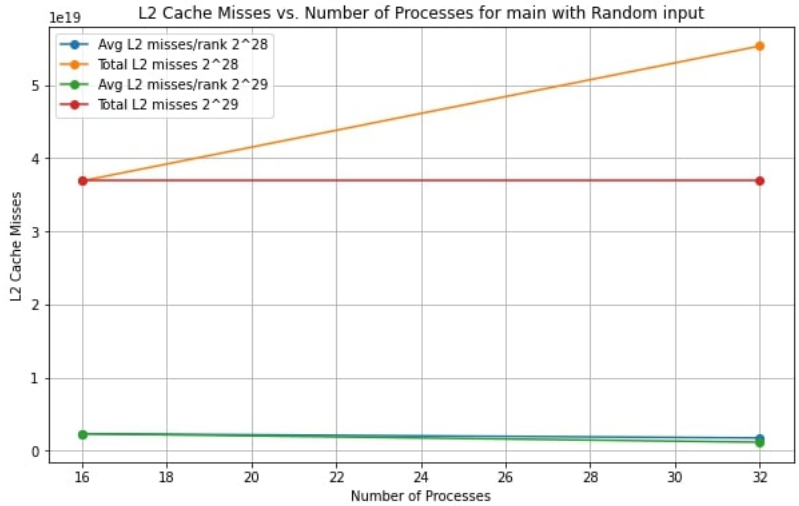
Cache Misses

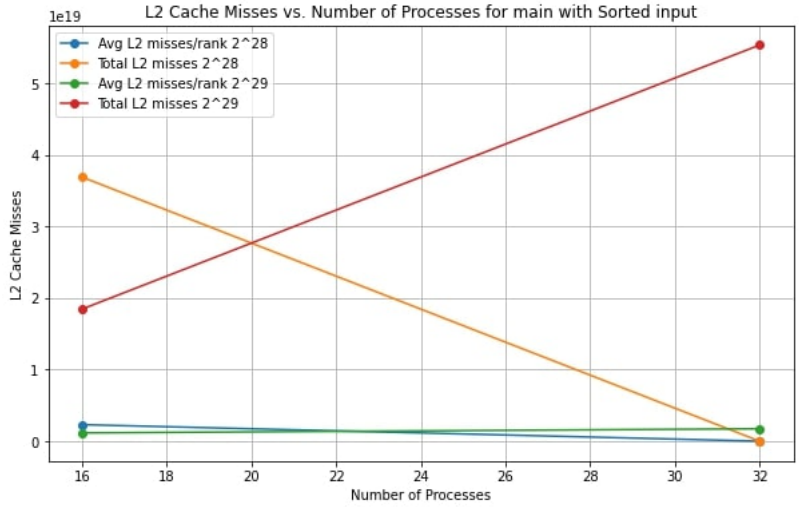


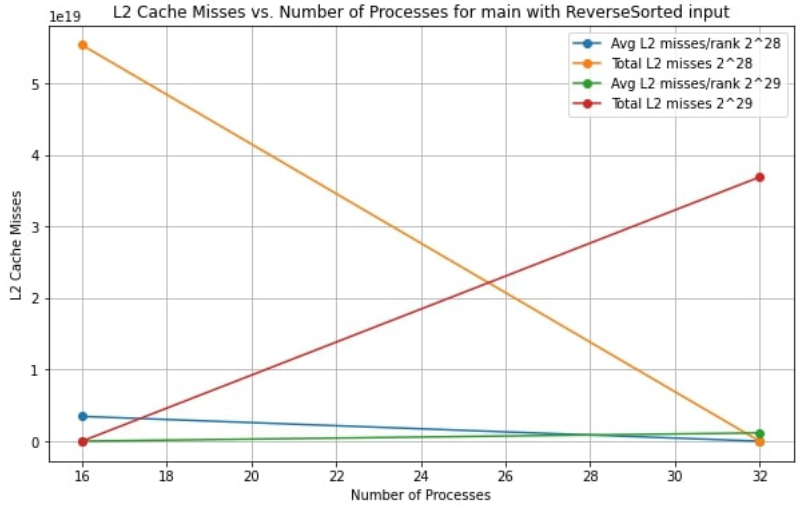




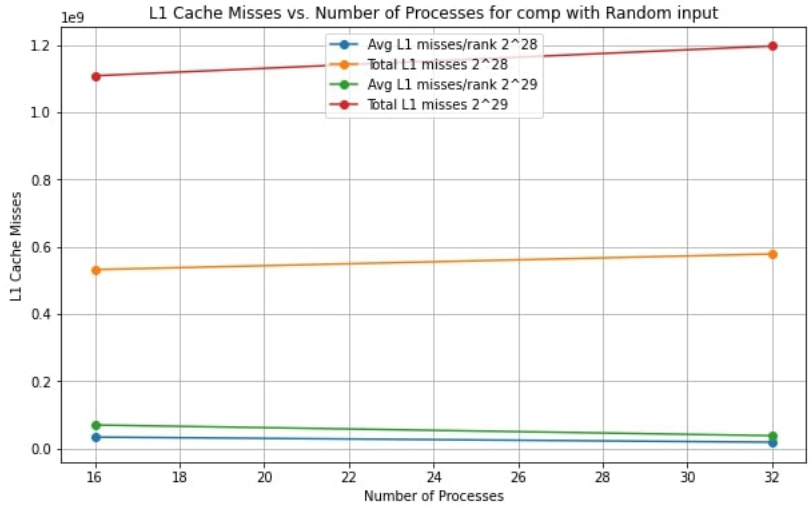
These are plots of the L1 cache misses on main for input types random, sorted, and reverse sorted. We can see that regardless of input type, the number of cache misses is the same on L1. There are more misses for the higher input size, which makes sense because there is more data that cannot be easily stored in the cache.

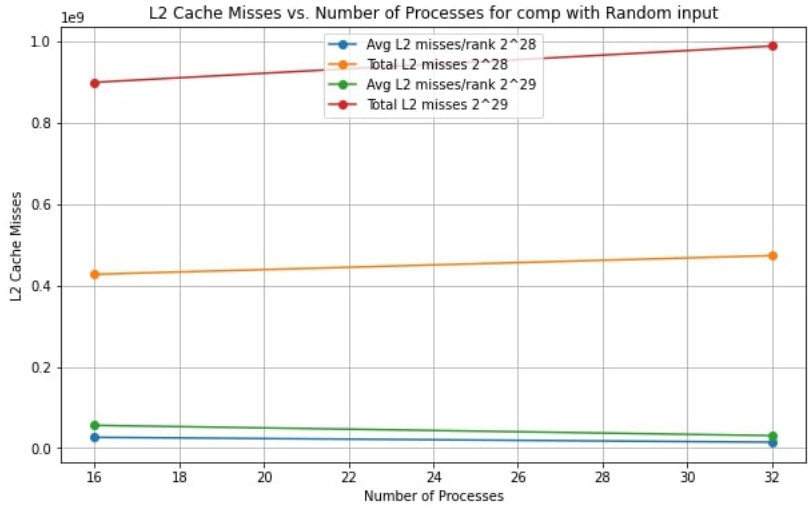




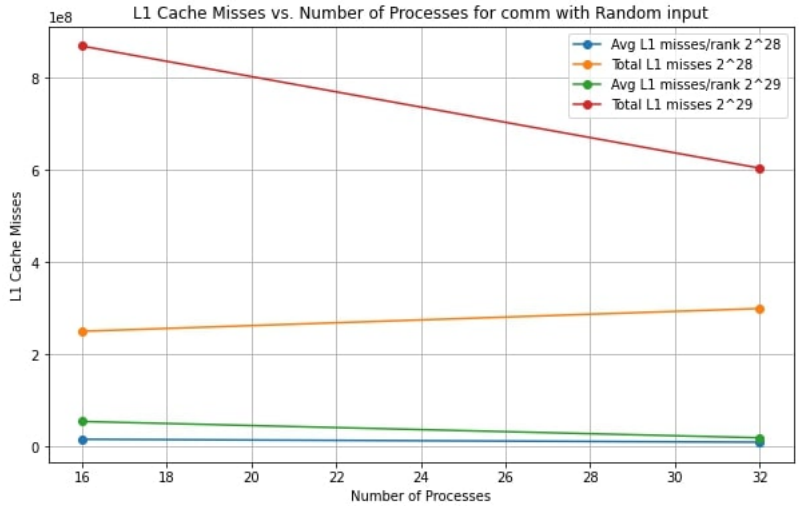


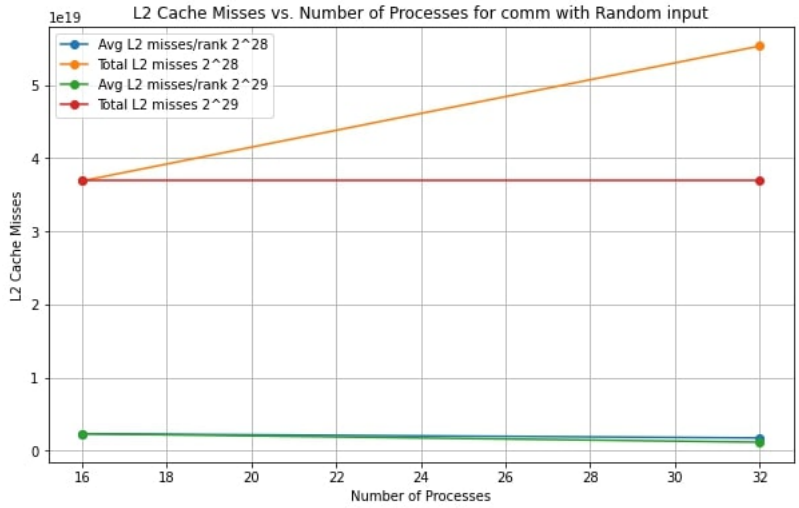
On L2, however, sorted and reverse sorted have similar trends, but random is different. This could be because random may require more communication than the other input types, so cache misses would go up with the number of processes for the smaller input size as compared to the other input types.





We can see that the L1 and L2 plots for computation are very similar, with a larger amount of cache misses for the larger input size. This makes sense because there is more data to handle, so a higher portion of it is not able to be stored in the cache.

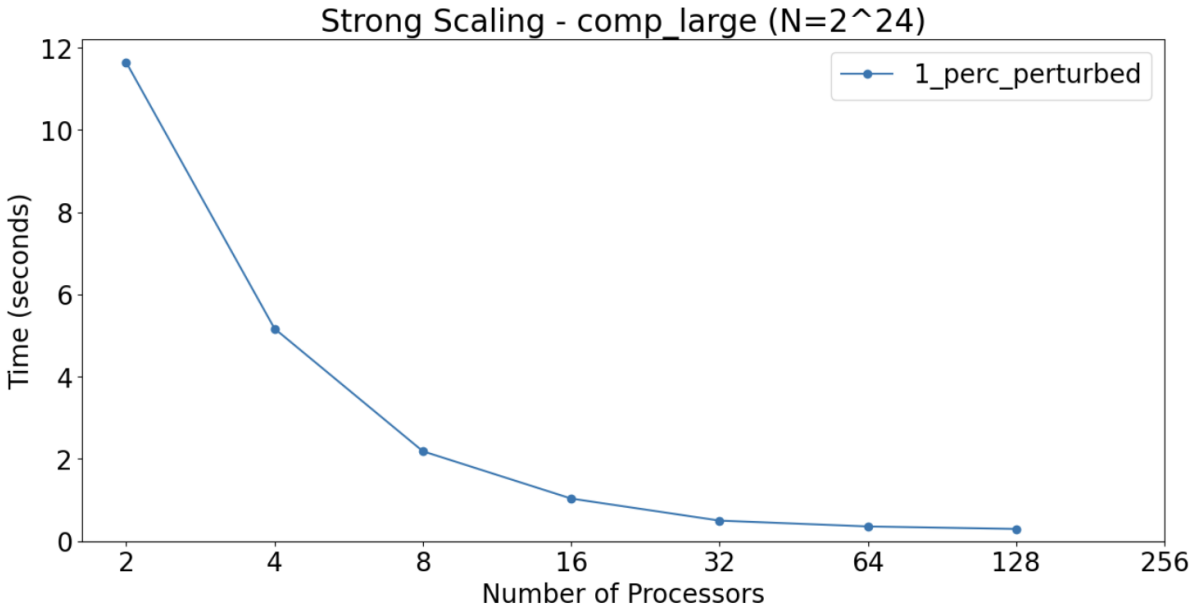


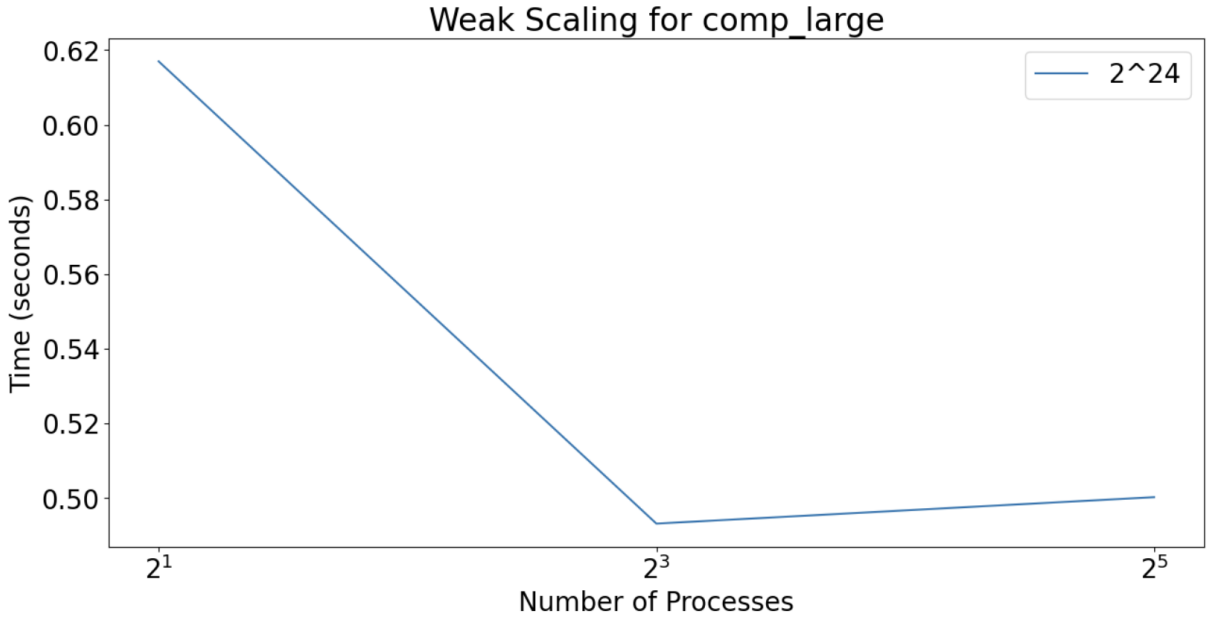


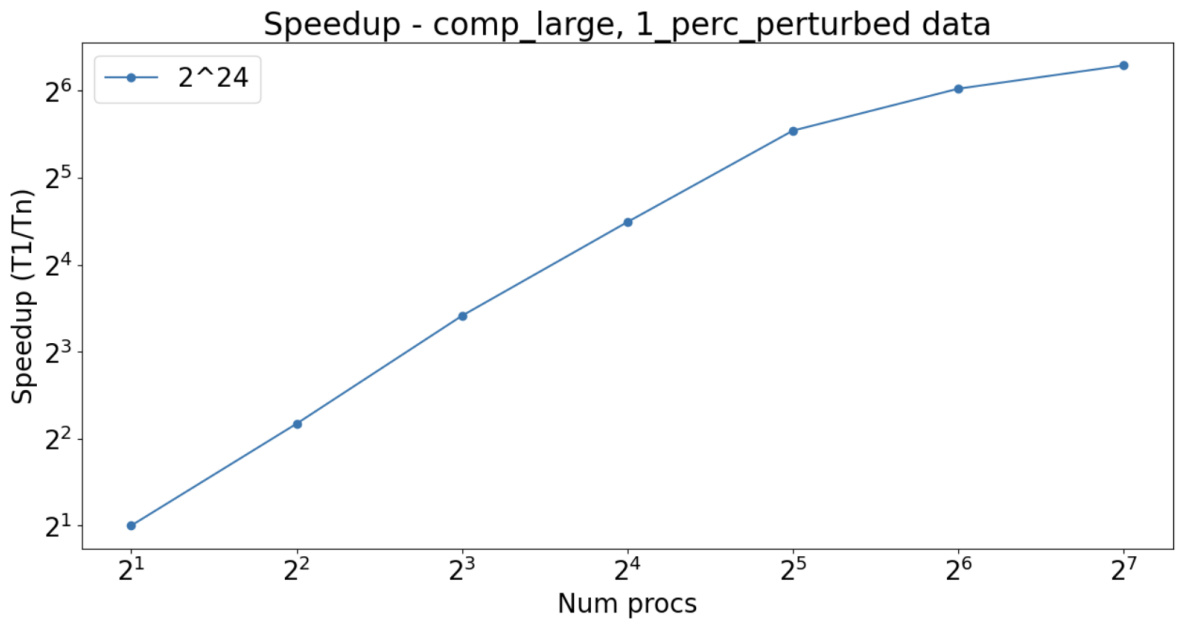
The communication plots show that on the L1 cache the higher input size has again a greater amount of cache misses. The L2 plot is different, with the smaller input size having more misses at 32 processes. This could be an anomaly resulting from a bad run.

Column Sort

Computation

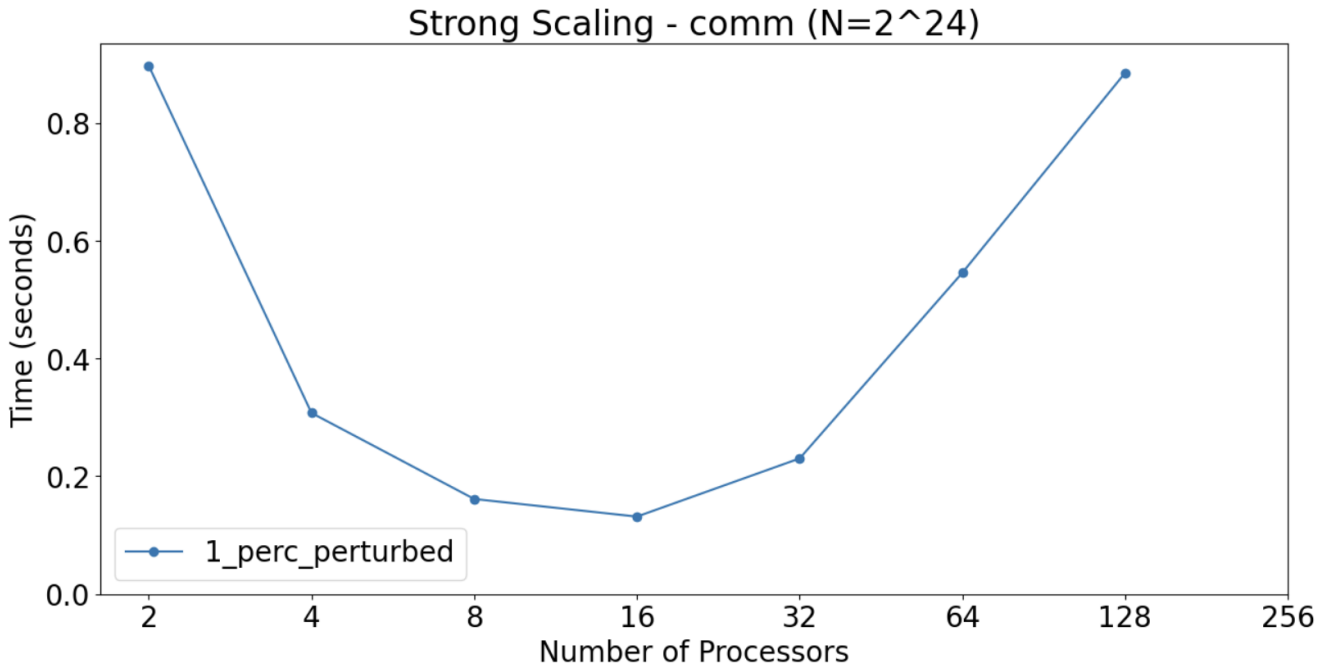

Strong Scaling - comp_large (N=2^24)
1_perc_perturbed
Time (seconds)
Number of Processors
0
2
4
6
8
10
12
2
4
8
16
32
64
128
256

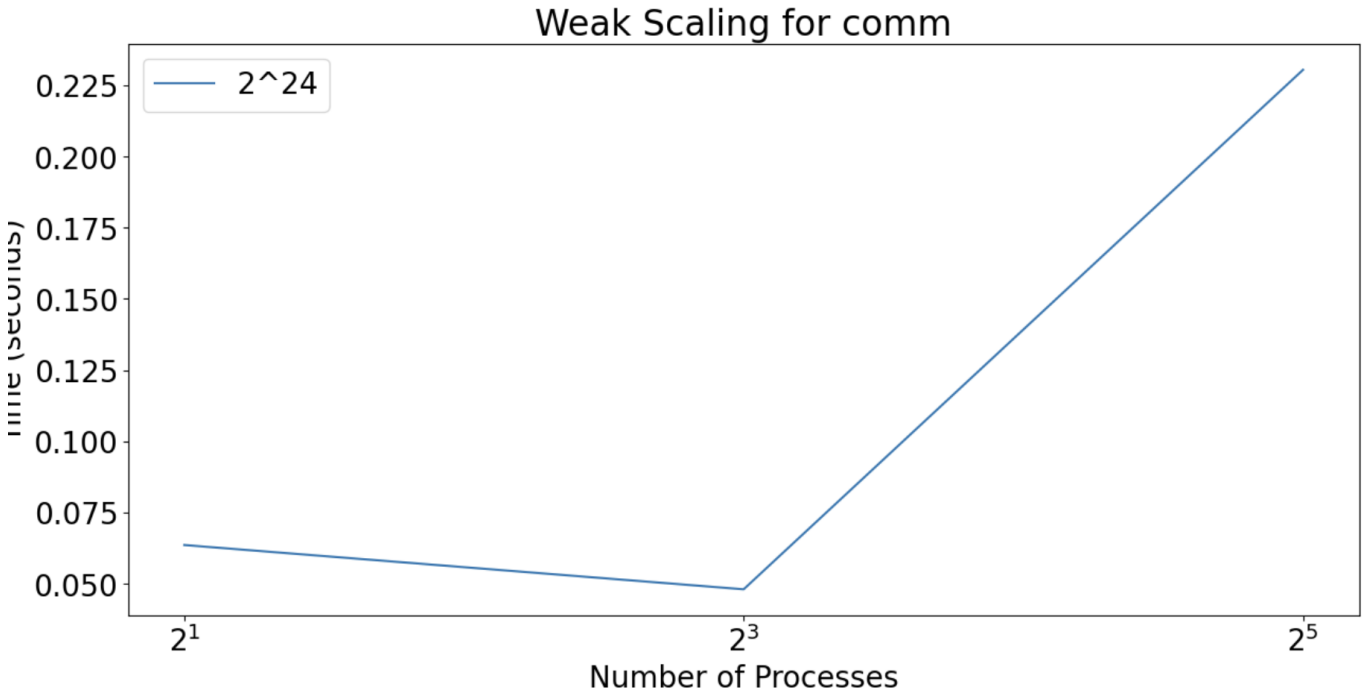

Weak Scaling for comp_large
2^24
Time (seconds)
Number of Processes
0.50
0.52
0.54
0.56
0.58
0.60
0.62
2^1
2^3
2^5



For computation, we can see the effects of parallel programming in column sort, as the computation time is decreasing exponentially with the number of processes. For weak scaling, it is interesting as computation is the only case where scaling up did not increase computation time. This may be due to the sequential nature of the algorithm. Looking at speedup, computation displayed the expected behavior as the speedup was logarithmically and positively related to the number of processes.

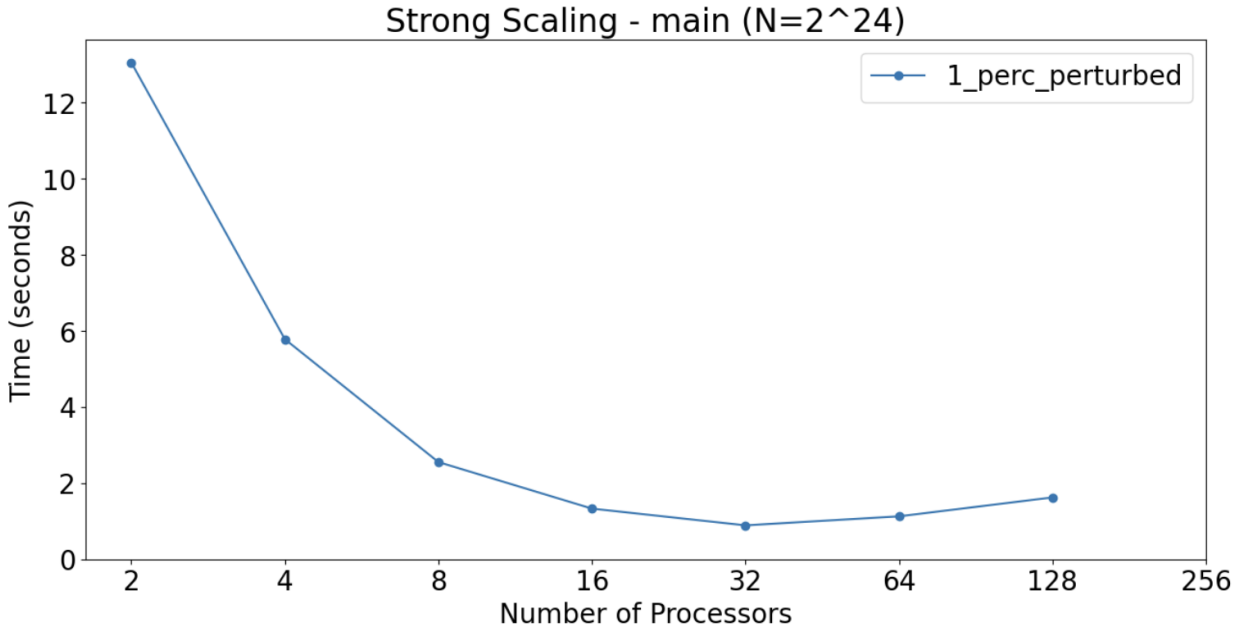
Communication

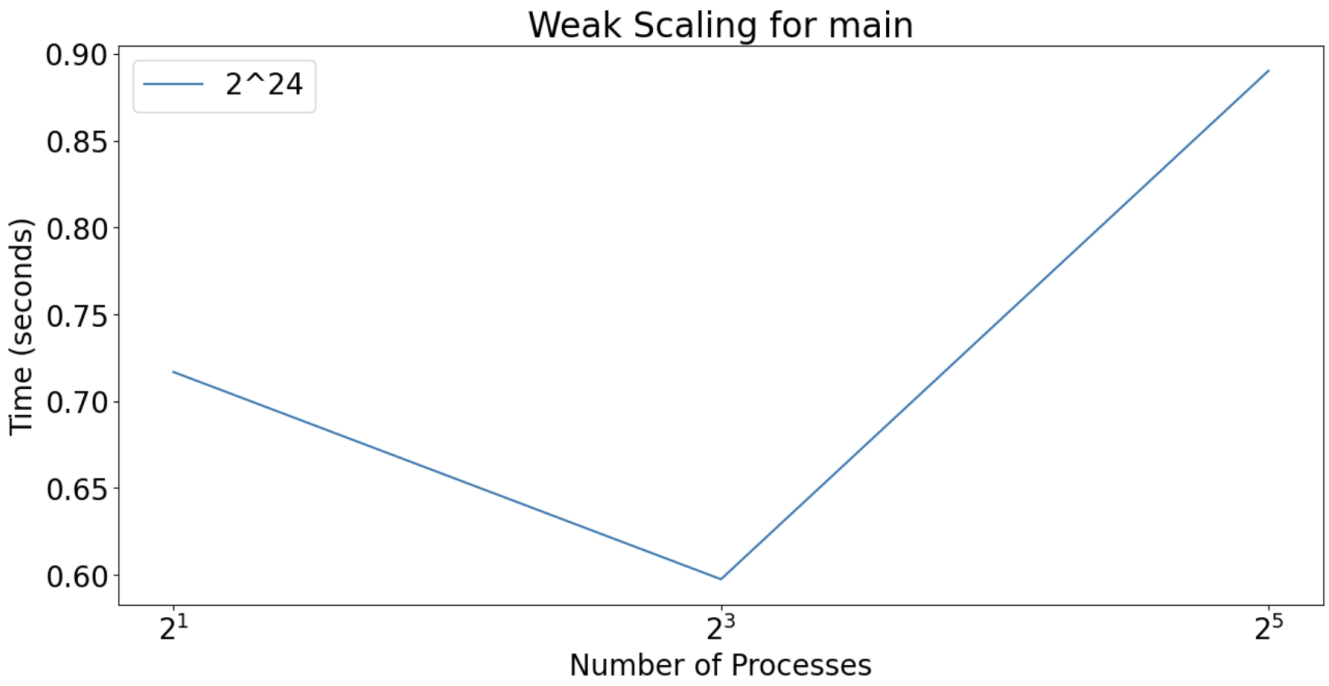


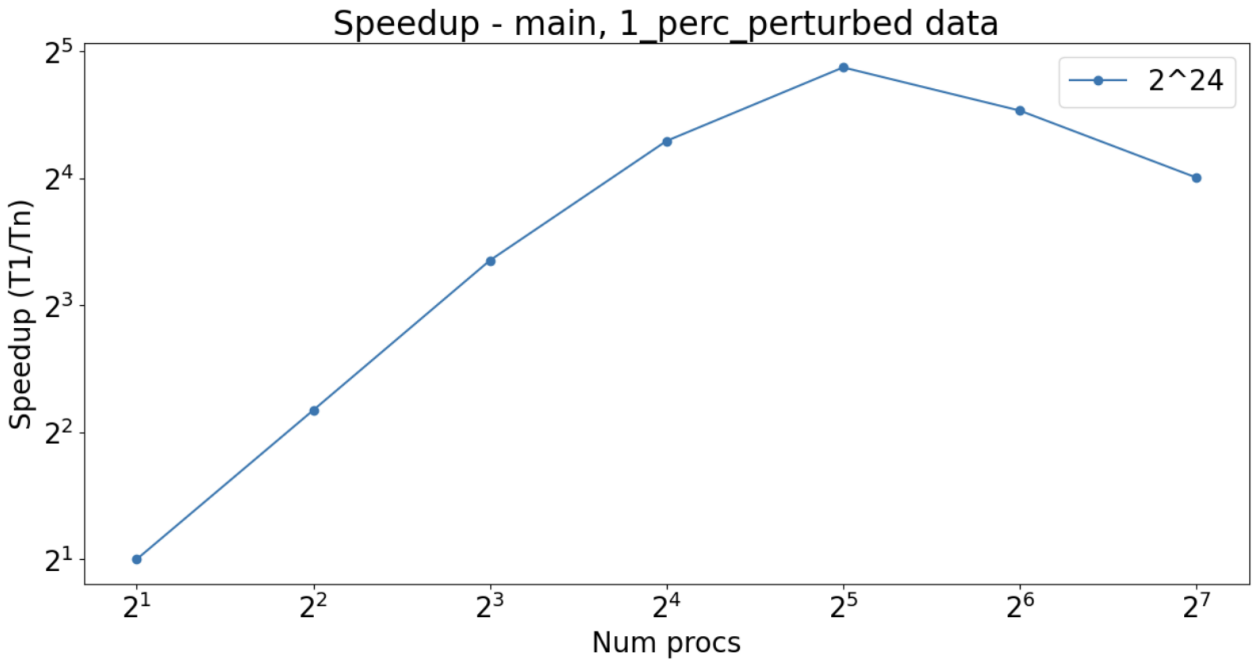


The way we implemented column sort was communication heavy, as there was 1 column assigned per processor and 4 MPIall_to_all calls for each of those. Hence, the trend in strong scaling seen is an inverse quadratic curve: where while processing time initially decreased, it plateaued around 16 processes and then increased. Weak scaling behavior was as expected, where increasing the number of processes while increasing input size increased computation time.

Main


Strong Scaling - main (N=2^24)
1_perc_perturbed
Time (seconds)
0
2
4
6
8
10
12
2
4
8
16
32
64
128
256
Number of Processors

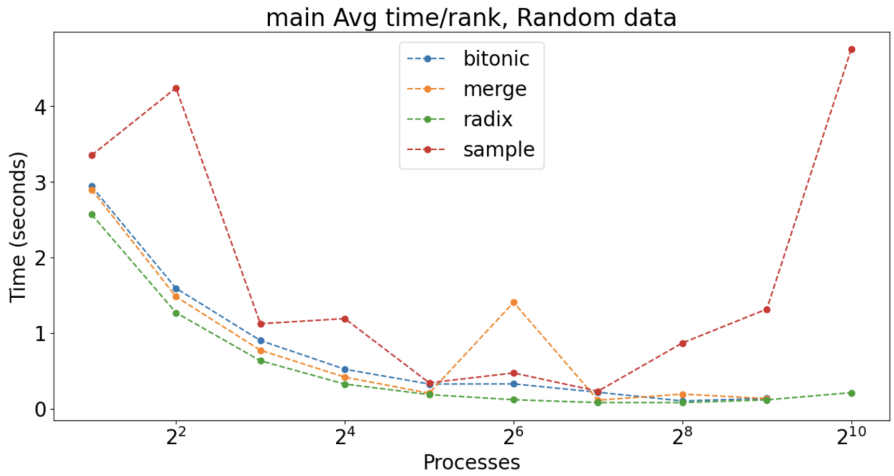

Weak Scaling for main
2^24
Time (seconds)
0.60
0.65
0.70
0.75
0.80
0.85
0.90
2^1
2^3
2^5
Number of Processes



Similar to computation, we can see the effects of parallel programming in column sort, as the computation time is decreasing exponentially with the number of processes. Weak scaling behavior is as expected where ncreasing the number of processes while increasing input size increased computation time. Looking at speedup, computation displayed the expected behavior as the speedup was logarithmically and positively related to the number of processes till about 32 processes, after which it started declining.

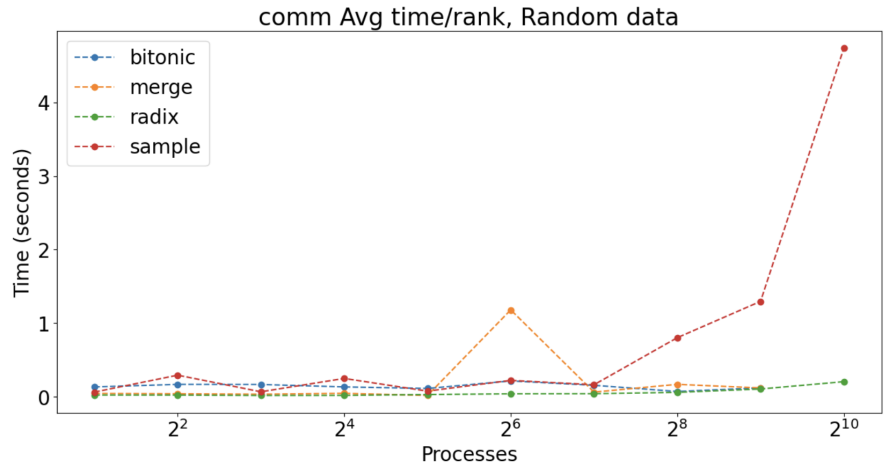
Algorithm Comparisons

Main



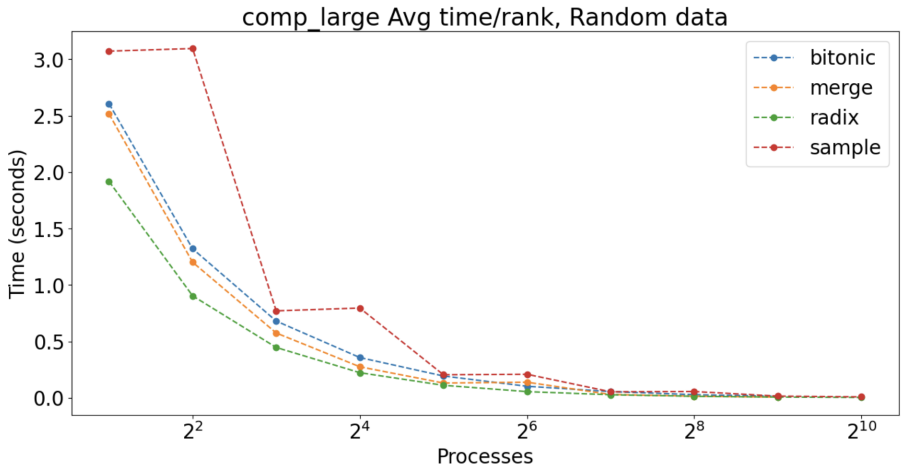
The main timing graph reveals interesting performance patterns across all sorting implementations. Starting at lower process counts, all algorithms show relatively high execution times around 3-4 seconds. Sample sort performs the worst, showing significant fluctuations in performance. As we move towards higher process counts, we see a general downward trend in execution time for all algorithms. Radix, merge, and bitonic sorts demonstrate very similar behavior after 16 processes, converging to nearly identical performance metrics. However, sample sort continues to show instability, with noticeable spikes in execution time even at higher process counts. Particularly interesting is the consistent performance of radix sort, which maintains the lowest and most stable execution times after 64 processes, suggesting better scalability than the other implemented algorithms.

Communication



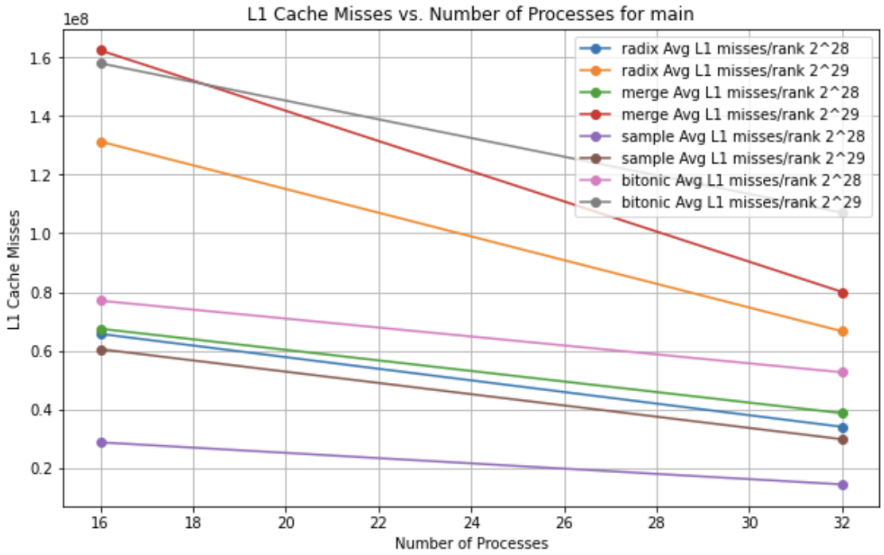
In the communication graph the most notable feature is the dramatic increase in communication time for sample sort at higher process counts, indicating poor communication scaling. Merge sort shows an interesting anomaly with a significant spike around 64 processes, suggesting a potential bottleneck in its communication pattern at this scale. In contrast, both radix and bitonic sorts maintain remarkably stable and low communication overhead throughout the process range. This graph particularly highlights how communication costs can become a dominant factor in parallel sorting performance as the number of processes increases, with radix sort showing the most efficient communication pattern overall.

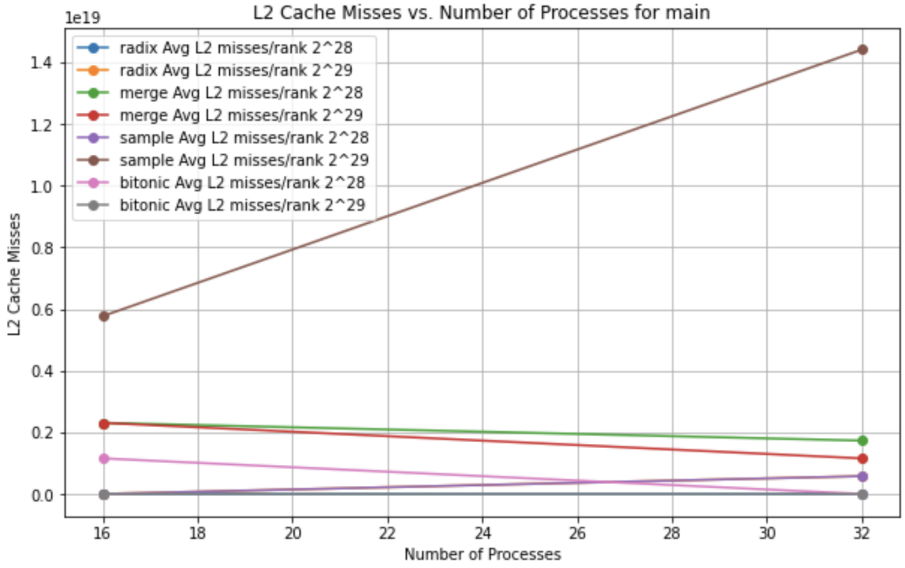
Computation



In terms of computation, initially, all algorithms show improvement with increased parallelization, starting from relatively high computation times. Sample sort again demonstrates the most irregular performance pattern, suggesting inconsistent load balancing. Radix sort consistently maintains better computation times throughout the process range, showing superior efficiency in handling larger datasets. Interestingly, all algorithms tend to converge in performance at very high process counts, indicating that the benefits of parallelization may plateau at these levels (diminishing returns from parallelization). The steady decline in computation time across all algorithms until this convergence point suggests that parallel implementation effectively reduces computational overhead up to a certain threshold.

Cache Misses





In the L1 cache graph, sample sort exhibits the highest miss rate, indicating less efficient memory access patterns. Bitonic and radix sorts demonstrate notably better cache utilization, with their miss rates consistently lower across different process counts. The L2 cache miss graph is particularly revealing, with radix sort showing remarkably superior cache performance compared to other algorithms. This suggests that radix sort's memory access pattern is more cache-friendly, likely contributing to its overall better performance. The general trend shows improving cache performance with increased process counts for most algorithms, though sample sort maintains relatively high miss rates throughout. This cache behavior helps explain the overall performance characteristics observed in the other graphs.