Jithin Joyson CS 546: Parallel and Distributed Processing September 24, 2019

Programming Assignment Report 1

1. Array Sorting Problem

Premise:

Given an array of size n (50 was the hardcoded size), try to sort the array with an arbitrary amount of processor. The initial implementation sort array with 2 processes.

Approach:

The approach is to divide the array into equal parts and assign portions to separate processors. Since blocking MPI commands are used (Send & Recv), it is easier to visualize a computation flow:

1. Initially create array of size s in process 0.
2. Sort processor 0's array. (Assignment specified below)
3. Divide the array into p processors in process 0 and send each piece to each processor which was achieved using a for loop over n processors.
    a. Problem arises when array is not divisible by amount of processors
    b. Naïve Approach:
        i. Integer divide the array size with number of processors.
        ii. Assign x amount to n-1 processors. (x = array_size/ # procs)
        iii. Assign left over data (array size – x*(n-1)) to last processor.
        iv. File: sort_2_procs_naive.c
    c. Computation Approach:
        i. Integer divide the array size with number of processors.
        ii. Mod the array size with number of processors.
        iii. Assign x+1 amount to m processors. (m = array_size % # procs)
        iv. Assign x amount to left over processors. (# procs – m)
        v. Benefit: Creates more spread array sizes
        vi. File: sort_2_procs.c
4. Receive from other processors. Other processors (not 0) receives, sorts data and sends back data.
5. At the end, merge all arrays from left to right based on portion in processor 0.

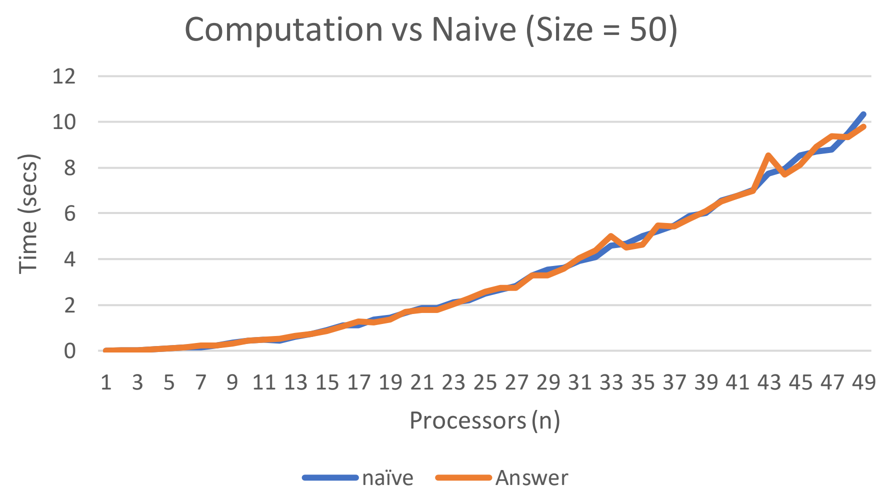
Insight Gained and Problems:

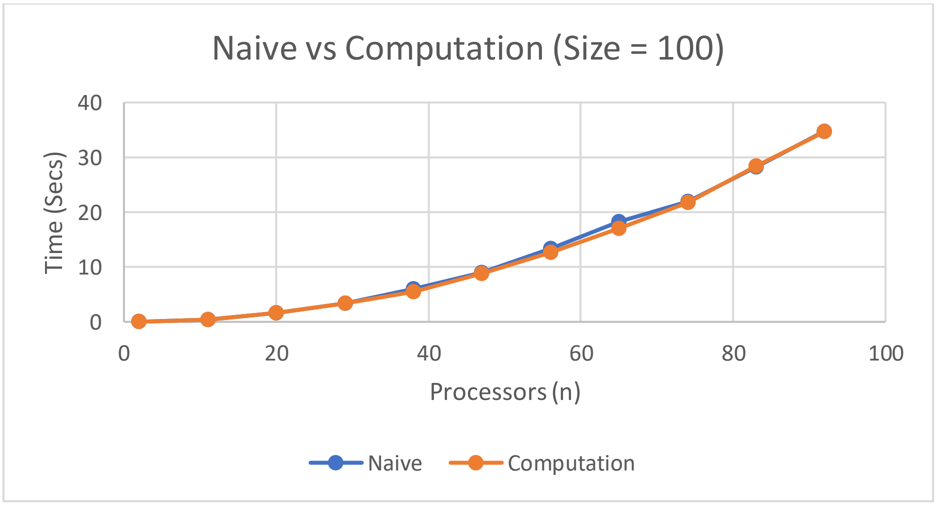
The biggest problem was dividing the initial array and determining how much data is assigned to what processor. After testing few ways to divide array (Math.ceiling and Math.floor), the mod arithmetic (Computation Approach) yielded in a good spread of arrays which might impact performance.

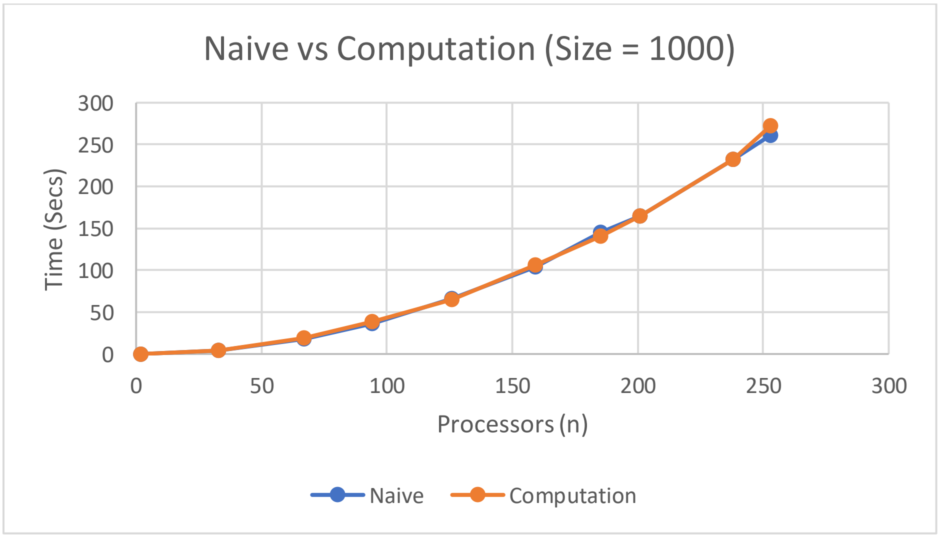
Optimizations:

1. Allowing for more than or less than (not including 0) 2 processors to sort the array
2. Allowing spread in array portioning before computation

Performance Results:

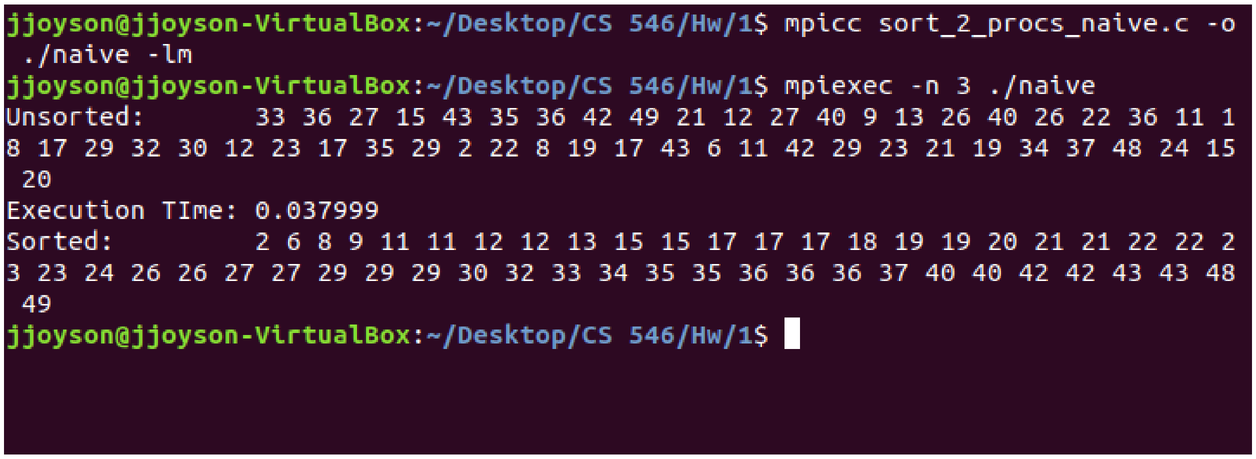






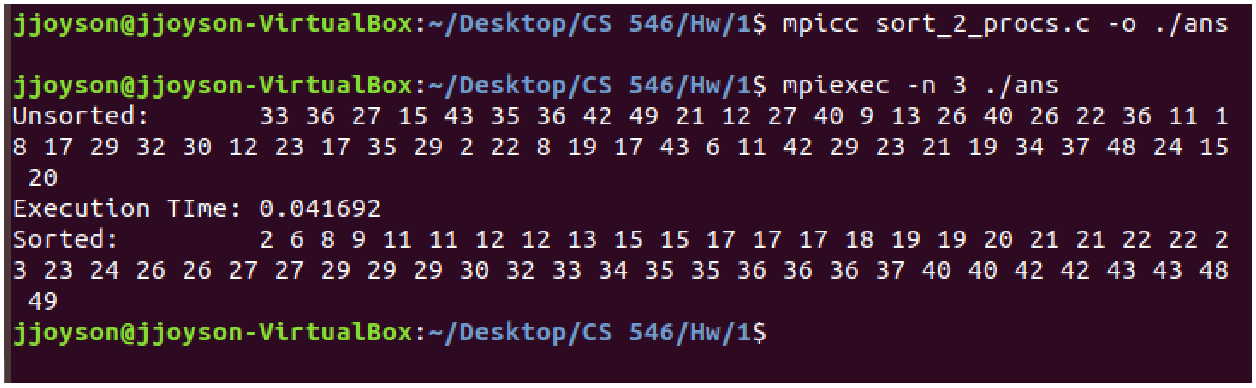
Output:

Running sort_2_procs_naive.c:



```
jjoyson@jjoyson-VirtualBox:~/Desktop/CS 546/Hw/1$ mpicc sort_2_procs_naive.c -o
 ./naive -lm
jjoyson@jjoyson-VirtualBox:~/Desktop/CS 546/Hw/1$ mpiexec -n 3 ./naive
Unsorted:       33 36 27 15 43 35 36 42 49 21 12 27 40 9 13 26 40 26 22 36 11 1
8 17 29 32 30 12 23 17 35 29 2 22 8 19 17 43 6 11 42 29 23 21 19 34 37 48 24 15
 20
Execution TIme: 0.037999
Sorted:         2 6 8 9 11 11 12 12 13 15 15 17 17 17 18 19 19 20 21 21 22 22 2
3 23 24 26 26 27 27 29 29 29 30 32 33 34 35 35 36 36 36 37 40 40 42 42 43 43 48
 49
jjoyson@jjoyson-VirtualBox:~/Desktop/CS 546/Hw/1$
```

Running sort_2_procs.c



```
jjoyson@jjoyson-VirtualBox:~/Desktop/CS 546/Hw/1$ mpicc sort_2_procs.c -o ./ans

jjoyson@jjoyson-VirtualBox:~/Desktop/CS 546/Hw/1$ mpiexec -n 3 ./ans
Unsorted:       33 36 27 15 43 35 36 42 49 21 12 27 40 9 13 26 40 26 22 36 11 1
8 17 29 32 30 12 23 17 35 29 2 22 8 19 17 43 6 11 42 29 23 21 19 34 37 48 24 15
 20
Execution TIme: 0.041692
Sorted:         2 6 8 9 11 11 12 12 13 15 15 17 17 17 18 19 19 20 21 21 22 22 2
3 23 24 26 26 27 27 29 29 29 30 32 33 34 35 35 36 36 36 37 40 40 42 42 43 43 48
 49
jjoyson@jjoyson-VirtualBox:~/Desktop/CS 546/Hw/1$
```

Analysis:

After the optimization to sort_2_procs.c to enable more the 2 processors, both the naïve approach and computation approach gave similar results. The naïve approach seems to do better when there are less processors but falls of as the number of processors increases as seen in graph 1 (Size = 50). All in all, the results are the same since the MPI commands are blocking each other.

2. Array Sorting Problem (Gather & Scatter)

<u>Premise:</u>

Given an array of size n (50 was the hardcoded size), try to sort the array with an arbitrary amount of processor. The initial implementation sort array with MPI_GET & MPI_SET. User MPI_Gather and MPI_Scatter instead.

<u>Approach:</u>

The approach is to divide the array into equal parts and assign portions to separate processors. Since scatter and gather spreads the data over the processors, there only needs to be scatter and gather implemented once.

1. Initially create array of size s in process 0.
2. Divide array into n-1 portions (n = # of processors) since processor 0 does not sort.
    a. Problem arises when array is not divisible by amount of processors
    b. Using Scatterv and Gatherv to choose array portions
    c. Sticking with naïve approach since there was not much difference in performance.
    d. Create count array and dis array to divide unsorted array.
    e. Initialize the counts to 0 for proc 0 (no sorting) and allow dis start at index 1.
    f. Scatterv data, sort, Gatherv data
3. At the end, merge all arrays from left to right based on portion in processor 0.

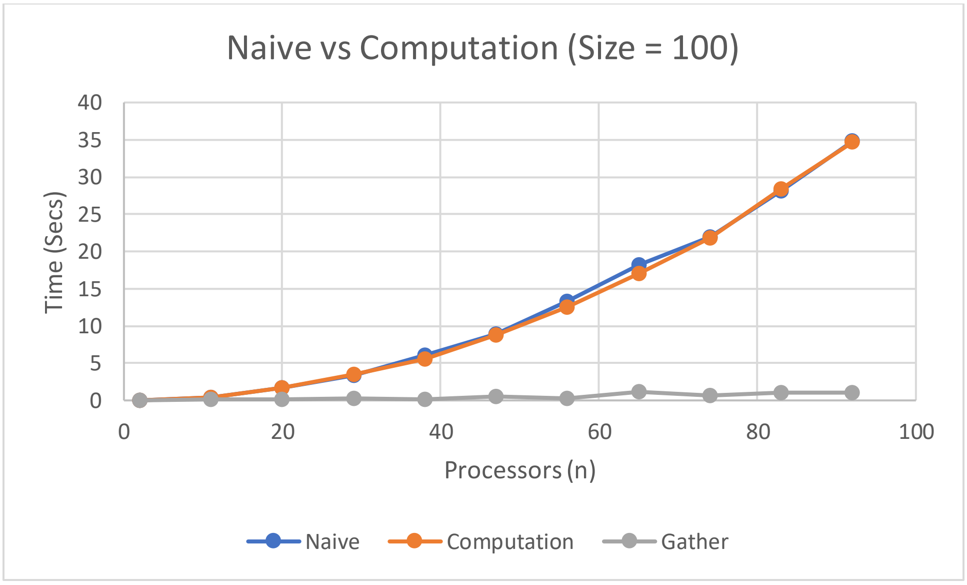
<u>Insight Gained and Problems:</u>

The biggest problem was dividing the initial array and determining how much data is assigned to what processor. After testing with Scatter and Gather, Scatterv and Gatherv were tested to allow for array portioning control.

<u>Optimizations:</u>

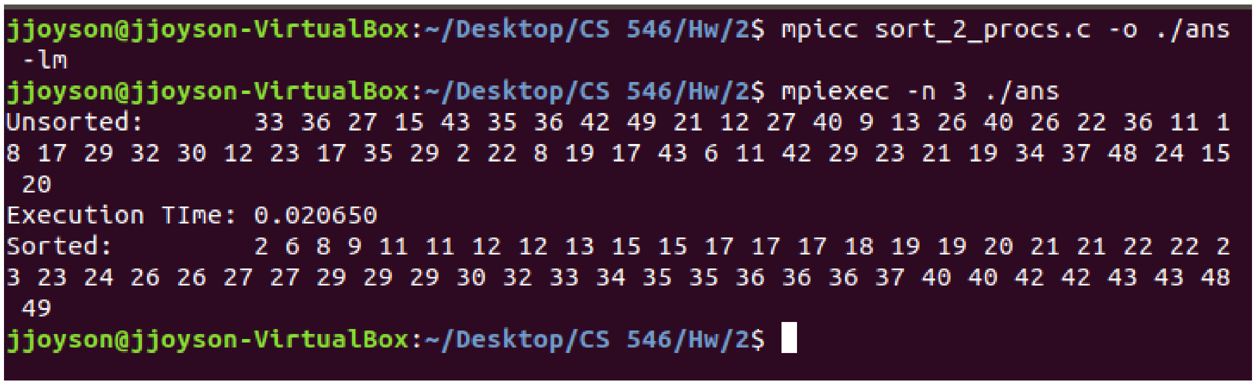
1. Replacing Get & Recv to allow for quick data access
2. Allowing control over array portioning through Scatterv and Gatherv

<u>Performance Results:</u>



Output:

Running sort_2_procs.c



```
jjoyson@jjoyson-VirtualBox:~/Desktop/CS 546/Hw/2$ mpicc sort_2_procs.c -o ./ans
 -lm
jjoyson@jjoyson-VirtualBox:~/Desktop/CS 546/Hw/2$ mpiexec -n 3 ./ans
Unsorted:       33 36 27 15 43 35 36 42 49 21 12 27 40 9 13 26 40 26 22 36 11 1
8 17 29 32 30 12 23 17 35 29 2 22 8 19 17 43 6 11 42 29 23 21 19 34 37 48 24 15
 20
Execution TIme: 0.020650
Sorted:         2 6 8 9 11 11 12 12 13 15 15 17 17 17 18 19 19 20 21 21 22 22 2
3 23 24 26 26 27 27 29 29 29 30 32 33 34 35 35 36 36 36 37 40 40 42 42 43 43 48
 49
jjoyson@jjoyson-VirtualBox:~/Desktop/CS 546/Hw/2$ 
```

Analysis:

After the optimization of Scatterv and Gatherv, the sorts have been running incredibly faster. As seen in the graph, the performance is considerably faster and does not follow a rising trend seen in the other methods as the processors increase.

3. 5-Point Stencil (Alltoallv)

Premise:

Given a 2-d array, apply the 5-point stencil algorithm to replace ISend and IRecv calls with AlltoALLv.

Approach:

The approach replace all the directional arrays with one huge array. Create a receiving buffer with the same size and do all the unpacking and packing with the updated data structures.

1. Create a data structure to hold all the ISend arrays
2. Create a data structure to hold all the IRecv arrays
3. Make a AlltoAllv with the new arrays such that the recv array can get hold of the sending array
4. Update dis and count arrays such that correct access is made after the passing of data
5. Make checks such that the request and error checking is done in the Alltoallv call.

Insight Gained and Problems:

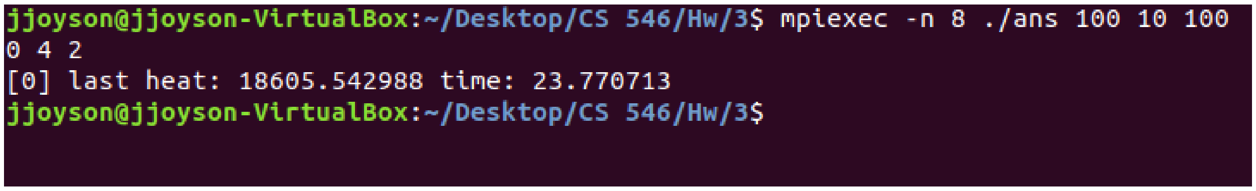
The biggest problem creating the right scounts and dis arrays such that the right data was being passed and received between the arrays. Another issue was to make the packing and unpacking of these arrays proper such that no segmentation faults are made. All the steps are commented in the code. Insight was gained regarding the AlltoAllv function.

Optimizations:

1. Replacing IGet & IRecv to allow for quick data access through Alltoallv.

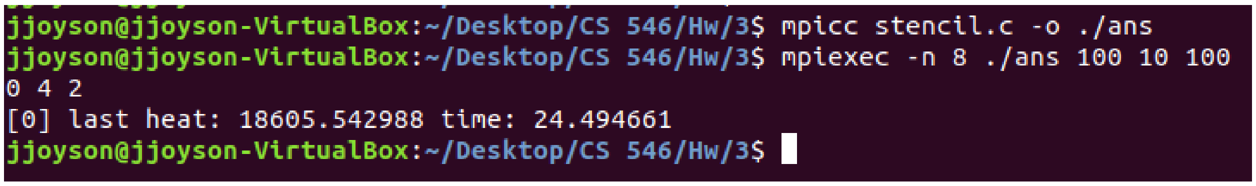
Performance Results & Output:

Default:



```
jjoyson@jjoyson-VirtualBox:~/Desktop/CS 546/Hw/3$ mpiexec -n 8 ./ans 100 10 100
0 4 2
[0] last heat: 18605.542988 time: 23.770713
jjoyson@jjoyson-VirtualBox:~/Desktop/CS 546/Hw/3$
```

Update:



```
jjoyson@jjoyson-VirtualBox:~/Desktop/CS 546/Hw/3$ mpicc stencil.c -o ./ans
jjoyson@jjoyson-VirtualBox:~/Desktop/CS 546/Hw/3$ mpiexec -n 8 ./ans 100 10 100
0 4 2
[0] last heat: 18605.542988 time: 24.494661
jjoyson@jjoyson-VirtualBox:~/Desktop/CS 546/Hw/3$
```

<u>Analysis:</u>

After the optimization of to Alltoallv, it seems that the program works just as fast as the orginal program. Further testing was not done due to the lack of time and input constraints but results match up