Three Sorting Algorithms and a Cluster

[Author Name]

[Type the abstract of the document here. The abstract is typically a short summary of the contents of the document.]
Introduction
In this section give an overview of the contents of the paper (it is alright if it is short)

- Describe the basic problem – sorting integers with parallel algorithms using MPI on the NSCC.
- Describe the goals – e.g. to determine for which problem sizes and which original configurations which algorithm performs best (THINK ABOUT THIS PART!)

Methodology
In this section, describe more details about the methodology

- Introduce terminology – e.g. N is the number of elements in the array, P is the number of processes. You may not need to do this in a separate paragraph, but make sure you define N and P when you introduce them.
- Describe the basic methodology, i.e. we are sorting arrays of integers of sizes ???, ???, and ?? (you fill in the sizes, they should be at least 100K) using three sorting algorithms using 2, 4, and 8 processors. The arrays are generated
  - From a uniform distribution
  - To be in reverse order
  - From a uniform distribution but which are then squared.
- Mention that to simplify the investigation, the problem size is designed to fit neatly on the multiple processors. We ensure that the data can be evenly divided among the processors and that there is an even number of items on each proc.
- Write a couple of sentences that indicate that to predict the sorting times, we have timed the relevant MPI operations on NSCC and describe predicting functions here. We also timed the local sorting/data manipulation for the three algorithms and describe those in the appropriate sections.

Time to Exchange
Two of the three sorting algorithms perform data exchange operations in which pairs of processors send and receive data to and from each other. The time it takes to perform the exchange is a function of the number of integers exchanged N and of the number of processes P. Even though each process is sending data to only one other process, all processes are competing for space on the same bus. Thus, exchanges take longer in an 8-process application than in a 2-process application. However, the multiple-processor effect on the timing is negligible when compared to the effect of the amount of data, so we make the simplifying assumption that the time to exchange depends on N only. The overhead associated with communication dominates the time its takes to send and receive small amounts of data, so we computed different time constants for small and large N. We define the function $T_{\text{exchange}}(N)$ to predict the amount of time (in seconds) it takes to exchange N integers according to

\[ T_{\text{exchange}}(N) = 8e^{-6}N, \quad \text{for small } N \text{ (e.g. } < 1000) \text{ and} \]
\[ 1.3e^{-8}N, \quad \text{for large } N \]

Time to Gather/Scatter
The MPI_Scatter routine evenly divides data on one processor among all processors and the reverse operation, MPI_Gather, gathers data evenly divided among many processors onto one processor.
Because these operations require nearly identical communication strategies, we use the same prediction function for both. We make the further assumption that MPI_Gatherv, which gathers unevenly dispersed data, takes the same amount of time as MPI_Gather. The amount of time it takes to scatter or gather data is a function of the total amount of data N and of the number of processors P, though we have found the effect of the number of processors is negligible. Again, the overhead associated with communication dominates the time it takes to send and receive small amounts of data, so we computed different time constants for small and large N. We define the function $T_{	ext{scatgat}}(N)$ to predict the amount of time (in seconds) it takes to scatter or gather N integers according to

$$T_{	ext{scatgat}}(N) = \begin{cases} 2e-6 N, & \text{for small } N \text{ (e.g. } < 1000) \\ 2.3e-9 N, & \text{for large } N \end{cases}$$

**Odd-Even Interchange Sort**

In this section

- Briefly describe the algorithm. Note that this is not what is commonly called the parallel version (because that would be the “vector” version we discussed in class). It is a standard odd-even sort just implemented on multiple computers. I have included two copies of the pseudocode - one is written with a for loop and the other is written with the while loop we discussed in class. Remove the text associated with the algorithm you don’t use. Feel free to change your code to match the for-loop pseudo-code – it may make analysis easier because you know ahead of time how many times the loop will execute.

- Write the equation which predicts the run-time (which I have done for you)

- Show the results of several runs (using the sizes and initial distributions mentioned above)
  - For each of 3 problem sizes, use each of the three initial distributions on 2, 4, and 8 processors. This means there are at least 27 $(N,P,\text{initial distribution})$ configurations. You should run each configuration multiple times (e.g. 3) in order to make sure the time you report is representative.
  - Plot the observed runtimes along with the predicted runtimes.

**IF YOU USE A FOR LOOP, INCLUDE THIS PART:**

The pseudo-code for a single process with the given rank is

```python
scatter  // send N data to P processes
for i = 1 : N/2
  // iterate through all (odd-even pairs of) N/P elements, possibly swapping them
  pair_swap
  exchange  // exchange leftmost item with left neighbor
  exchange  // exchange rightmost item with right neighbor
  // iterate through all (even-odd pairs of) N/P elements, possibly swapping them
  pair_swap
end for
gather  // return sorted data to root process
```

The time to sort N items using the odd-even interchange sort on P processors is a function of the time to perform the local pair-swapping and the exchange of a single integer (between a pair of processors). The core of the algorithm loops $N/2$ times, calling pair_swap twice (once to swap odd/even pairs, once to swap even/odd pairs) and exchange twice (once with the left neighbor, once with the right neighbor) each iteration. Taking the initial scatter and final gather into account, the total runtime (in seconds) is:

$$T_{\text{oddeven}}(N,P) = 2 * T_{\text{scatgat}}(N) + N * T_{\text{pairswap}}(N/P) + N * T_{\text{exchange}}(1)$$

We estimate that $T_{\text{pairswap}}(N) = 1.7e-9 * N$. 


The pseudo-code for a single process with the given rank is

```plaintext
scatter // send N data to P processes
while not done
    // iterate through all (odd-even pairs of) N/P elements, possibly swapping them
    pair_swap
    exchange // exchange leftmost item with left neighbor
    exchange // exchange rightmost item with right neighbor
    // iterate through all (even-odd pairs of) N/P elements, possibly swapping them
    pair_swap
    all_reduce // determine how many processors are done
end while
gather // return sorted data to root process
```

The time to sort N items using the odd-even interchange sort on P processors is a function of the time to perform the local pair-swapping, the exchange of a single integer (between a pair of processors), and an MPI_Allreduce operation for a single integer. The core of the algorithm loops at most N/(2+1) times, calling pair-swap twice (once to swap odd/even pairs, once to swap even/odd pairs), exchange twice (once with the left neighbor, once with the right neighbor), and Allreduce once each iteration. We assume worst-case looping time. Taking the initial scatter and final gather into account, the total runtime (in seconds) is:

```
T_{odd even}(N,P) = T_{scatter}(N) + N*T_{pair swap}(N/P) + N*T_{exchange}(1) + N/2*T_{all reduce}(P) + T_{scatter}(N)
```

We estimate that T_{all reduce}(P) = 1e-6 * P and that T_{pair swap}(N) = 1.7e-9 * N.

**Bucket Sort**

In this section

- Briefly describe the algorithm
- Write the equation which predicts the run-time (which I have done for you below)
- Show the results of several runs (using the sizes and initial distributions mentioned above)
  - For each of the three problem sizes, use each of the three initial distributions on 2, 4, and 8 processors. This means there are at least 27 (N,P,initial_distribution) configurations. You should run each configuration multiple times (e.g. 3) in order to make sure the time you report is representative.
  - Plot the observed runtimes along with the predicted runtimes.

The pseudo-code for a single process with the given rank is

```plaintext
scatter // send N data to P processes
determine which of my elements belongs in which bucket
    // redistribution
    for i = 1 : P
        gather // all processes send their bucket counts to process i
        end for
    for i = 1 : P
        gather // all processes send the data belonging in bucket i to process i
        end for
    qsort   // sort N/P elements
    gather // send sorted data count to root process
    gather // return sorted data to root process
```
The time to sort $N$ integers using the bucket sort with $P$ processes is a function of the time to perform the initial scatter, the time to perform the local bucket-distribution, the time to redistribute the data to the appropriate processes, the time to perform a local quicksort, and the time to perform the final gather. We are making the simplifying assumption that each processor will end up with $N/P$ integers in each bucket. This assumption may not hold, but will give us a reasonable estimate for the amount of time it takes to run the algorithm when the data is uniformly distributed. We predict the runtime (in seconds) $T_{bucket}(N,P)$ is

$$T_{bucket}(N,P) = T_{scatter}(N) + P \cdot T_{scatter}(1) + P \cdot T_{scatter}(N/P) + T_{qsort}(N/P) + T_{scatter}(1) + T_{scatter}(N)$$

where we assume average case performance of the quicksort and estimate its runtime $T_{qsort}(N) = 6.6e-9 \cdot N \cdot \log_2(N)$.

**Bitonic Sort**

In this section

- Briefly describe the algorithm.
- Write the equation which predicts the runtime.
- Show the results of several runs (using the sizes and initial distributions mentioned above)
  - For each of 3 problem sizes, use each of the three initial distributions on 2, 4, and 8 processors. This means there are at least 27 $(N,P,initial\_distribution)$ configurations. You should run each configuration multiple times (e.g. 3) in order to make sure the time you report is representative.
  - Plot the observed runtimes along with the predicted runtimes.

The pseudo-code for a single process with the given rank is

```plaintext
scatter // send N data to P processes
qsort   // sort N/P elements with direction determined by 0th bit of rank
for depth = 1 : log2(P)
  for p = d-1 : -1 : 0
    exchange // exchange N/P elements with processor whose rank differs in pth bit
    bitonic_merge // local bitonic merge of N/P elements
  end for
qsort   // sort N/P elements with direction determined by dth bit of rank
end for
gather // return sorted data to root process
```

The time to sort $N$ integers using the bitonic sort with $P$ processes is a function of the time to perform the initial scatter, the time to perform the exchange of $N/P$ integers, the time to perform the quicksort, the time to perform the (local) bitonic merge, and the time to perform the final gather. The exchange and bitonic merge operations are performed in every iteration of the loop – i.e. $(\log_2(P)^2)/2$ times and the quicksort is performed $\log_2(P)+1$ times. We predict the runtime (in seconds) $T_{bitonic}(N,P)$ is

$$T_{bitonic}(N,P) = T_{scatter}(N) + (1+\log_2(P)) \cdot T_{qsort}(N/P) + (\log_2(P)^2)/2 \cdot (T_{exchange}(N/P) + T_{bitonic\_merge}(N/P)) + T_{scatter}(N)$$

where we estimate $T_{bitonic\_merge}(N) = 5.2e-9 \cdot N$ and, like above, we assume average case performance of the quicksort and estimate its runtime $T_{qsort}(N) = 6.6e-9 \cdot N \cdot \log_2(N)$.

**Discussion**

In this section
Discuss the results from the three sorts, using the three initial configurations, using the three (or more) problem sizes on 2, 4, and 8 processors. Yes, this means there are at least $3^3=27$ runs, and with duplicates (I suggested 3), that means there will be $27^3=19683$ runs. This shouldn’t take too long if you automate your runs. I ran bucket sort with $N=10000$ and it took ~0.2 seconds to run a problem. If your runs average 0.2 seconds, then all of the runs together take only $27^3*0.2=48s$. So, run it a lot.

Be sure to address any differences between predictions and observations.

For full credit, verify your explanations as best you can. For example, if the bucket-sort is taking varying amounts of time and you think that is because the bucket sizes are varying wildly, then print out the bucket sizes and add those results to your write-up.

End with a concluding paragraph that summarizes the “take-home message” of your work.