Chapter 9 - Exercise 1

1. Before coding how much speed up do we think we will get?
   - If there was 0 delay in moving the information?
2. Code a serial and parallel solution for bubble sort.
   - The gaussian elimination MPI code makes a good starting point for parallel.
     - The loop that computes pivot – update it to do swaps and then it sends the updated elements to the next process (rather than all processes).
3. How much speed up did you see?

Notes on Implementation:
Each processor in the approach will only have some of the largest values. For example if there are 8 processors doing the sorting of 32 items then the first processor will have the largest values at the following indexes (assuming first index is 1): 8, 16, 24, and 32.

I did check and values don't get lost so you can send extra values and they will get picked up by the other processor fine. However, do be careful you read all the values you send. For example if I sent the values 1, 4, 5, 6 and then one pass only read the 1, 4 when I should have read 1, 4, 5 then when the next pass starts it would read a 5 instead of a 6. This comes into play when you need a processor to do additional passes.

I gave some thought to collecting these values together and the easiest approach I came up with is side aside one processor to wait for the values to come in. You know that the first value will come from the first processor, second from second processor, etc... and then it loops back to the first. You also know how many values you should get. For example, if you had 6 processors, then 1-5 could do the sorting and 6 collect the data or 1 collect the data and 2-6 do the sorting.

Another problem to keep in mind is the last few values, so when only 1-3 values left can create some special case situations. Use the general solution and then see where you might need some additional checks for near the end.

Graded Rubric
_____ (1 Point) Serial version of bubble sort
_____ (2 Points) Parallel version of bubble sort properly passes swap information to the next processor
_____ (2 Points) Parallel version of bubble sort properly collects all the sorted values together
_____ (1 Point) Parallel version able to sort more elements than number of cores - 1.
_____ (2 Points) Time cost measure for each approach on a large enough n (1000 items should be enough). Make sure to include number of processors and the n you used.

Answer:

1. **Expected Speedup (Ideal Case with No Delay):**
   • In an ideal case with no communication delay, the theoretical speedup would be close to the number of processors used. However, in practical parallel bubble sort, speedup tends to be less than linear due to inherent dependencies in sorting operations like how we saw in class last week.

2. **Code for Serial and Parallel Bubble Sort in C++:**

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <mpi.h>

using namespace std;

// Serial Bubble Sort
void bubbleSortSerial(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n − 1; i++) {
        for (int j = 0; j < n − i − 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

// Parallel Bubble Sort (using MPI)
void bubbleSortParallel(vector<int>& arr, int rank, int size) {
    int n = arr.size();
    int local_n = n / size;
    vector<int> local_arr(local_n);

    MPI_Scatter(arr.data(), local_n, MPI_INT, local_arr.data(),
local_n, MPI_INT, 0, MPI_COMM_WORLD);

    // Perform local bubble sort on each processor
    for (int i = 0; i < local_n − 1; i++) {
        for (int j = 0; j < local_n − i − 1; j++) {
            if (local_arr[j] > local_arr[j + 1]) {
                swap(local_arr[j], local_arr[j + 1]);
            }
        }
    }

    // Gather sorted subarrays back to root
    MPI_Gather(local_arr.data(), local_n, MPI_INT, arr.data(),
local_n, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        // Final sort on root processor to combine segments
        bubbleSortSerial(arr);
    }
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int n = 1000; // Array size
    vector<int> arr(n);
    double time_serial = 0.0;  // Declare time_serial here

    if (rank == 0) {
        for (int i = 0; i < n; i++) {
            arr[i] = rand() % 1000;
        }

        // Serial Bubble Sort (only on rank 0 for comparison)
        vector<int> arr_serial = arr;
        double start_serial = MPI_Wtime();
        bubbleSortSerial(arr_serial);
        double end_serial = MPI_Wtime();
        time_serial = end_serial - start_serial;
        cout << "Time taken for Serial Bubble Sort: " <<
time_serial << " seconds" << endl;
    }

    // Parallel Bubble Sort
    vector<int> arr_parallel = arr;
    double start_parallel = MPI_Wtime();
    bubbleSortParallel(arr_parallel, rank, size);
    double end_parallel = MPI_Wtime();
    double time_parallel = end_parallel - start_parallel;

    if (rank == 0) {
        cout << "Time taken for Parallel Bubble Sort: " <<
time_parallel << " seconds" << endl;
        cout << "Speedup: " << (time_serial / time_parallel) <<
endl;
    }

    MPI_Finalize();
    return 0;
}
```

Output:

```
sa7233@sloop:~/fall2024/HPC/chp9$ mpirun -n 8 ./exer1
Time taken for Serial Bubble Sort: 0.00430246 seconds
Time taken for Parallel Bubble Sort: 0.00364111 seconds
Speedup: 1.18163
sa7233@sloop:~/fall2024/HPC/chp9$ mpirun -n 8 ./exer1
Time taken for Serial Bubble Sort: 0.00374827 seconds
Time taken for Parallel Bubble Sort: 0.00367806 seconds
Speedup: 1.01909
sa7233@sloop:~/fall2024/HPC/chp9$ mpirun -n 8 ./exer1
Time taken for Serial Bubble Sort: 0.0037409 seconds
Time taken for Parallel Bubble Sort: 0.00362603 seconds
Speedup: 1.03168
sa7233@sloop:~/fall2024/HPC/chp9$ mpirun -n 8 ./exer1
Time taken for Serial Bubble Sort: 0.00374099 seconds
Time taken for Parallel Bubble Sort: 0.00355393 seconds
Speedup: 1.05264
sa7233@sloop:~/fall2024/HPC/chp9$ mpirun -n 8 ./exer1
Time taken for Serial Bubble Sort: 0.00429961 seconds
Time taken for Parallel Bubble Sort: 0.00365394 seconds
Speedup: 1.1767
sa7233@sloop:~/fall2024/HPC/chp9$ mpirun -n 8 ./exer1
Time taken for Serial Bubble Sort: 0.0042919 seconds
Time taken for Parallel Bubble Sort: 0.0036292 seconds
Speedup: 1.1826
sa7233@sloop:~/fall2024/HPC/chp9$
```

3. Measured Speedup:

From the multiple test runs, here are the measured times and speedups:

- **Serial Bubble Sort Time**: Ranged from approximately 0.0037 to 0.0043 seconds.
- **Parallel Bubble Sort Time (using 8 processors)**: Ranged from approximately 0.0035 to 0.0037 seconds.
- **Speedup**: Observed speedup values ranged from about 1.02 to 1.18.

Conclusion: The speedup achieved using 8 processors was modest, ranging from around 1.02 to 1.18. This limited speedup is likely due to the communication overhead in parallelizing a bubble sort, as bubble sort is not highly parallel-friendly because of its dependence on sequential comparisons and swaps.