Seth Tourish

Dr. Guillen

CS 4328

23 September 2024

Homework 1 - “Example\_Average\_Point\_to\_point\_Comm.py” Report

**Mathematical Operation**

The mathematical operation that the program is written to achieve is to find the sum of an array that has been transformed, using a second array to make the transformations. In the code, the dataset is predetermined to be an array A that is filled to a set size of 10 million of all value 1, with the program applying an element-wise affine transformation and returning the average of the transformed array.

The program works as follows:

1. The program starts with importing mpi4py and numpy.
2. The program sets the default MPI communicator(MPI.COMM\_WORLD), as world\_comm.
3. The program gets the total number of processes, world\_size.
4. The program sets each process to have a unique identifier, called rank.
5. The program sets the default size of the problem to 10 million.
6. The program creates an array to hold how many elements each process will handle. This is determined by dividing the problem size by how many processes are created to parallelize the computation.
   1. Since the division may not always be evenly divided, the program distributes the remainder to the first few processes until no remainder remains.
7. The program then assigns the section of the total array that the process will handle. For the first rank, the start is index 0, and the end is index 0 + the size of the workload for rank 0. For each following rank, the start index is the sum of all previous rank’s workloads, and the end is the start + the workload size of the current rank.
8. The program then fetches the wall time(MPI.Wtime()) to get the start time to time the initialization of array a.
9. The program for each rank sets the array a to length of the rank’s workload size to values of ones(a = np.ones( workloads[my\_rank] )).
10. The program then fetches the wall time(MPI.Wtime()) again to get the end time of array a’s initialization.
11. If the rank of the process is 0, then the program prints the initialization time by subtracting start time from end time.
12. The program then fetches the wall time(MPI.Wtime()) to get the start time to time the initialization of array b.
13. Array b is initialized as an array of zeros of the length of the current rank’s workload.
14. For each rank, the program then uses a loop to set each element of b to 1.0 + index + my\_start(the start value of the workload the process is handling). Each process only initializes an array with its portion of the transformation array.
15. The program then fetches the wall time(MPI.Wtime()) again to get the end time of array b’s initialization.
16. If the rank of the process is 0, then the program prints the initialization time by subtracting start time from end time.
17. The program then fetches the wall time(MPI.Wtime()) to get the start time to time the addition of the two arrays.
18. The program then uses the current rank’s workload size to iterate through both array a and array b, and add the values at each index of b to their respective index of array a.
19. The program then fetches the wall time(MPI.Wtime()) to get the end time to time the addition of the two arrays.
20. If the rank of the process is 0, then the program prints the summation time by subtracting start time from end time.
21. The program then fetches the wall time(MPI.Wtime()) to get the start time to time the average of the transformed array.
22. For each rank, the program loops through the transformed array a of each respective workload size and sums up the values and stores the value as sum.
23. If the process rank is 0:
    1. The program creates a variable world sum, to hold the sum of all processes and sets it to the value of rank 0’s sum.
    2. The program then loops from 1 to the number of processes running. For each rank:
       1. The program initializes an empty numpy array(sum\_np) and sets it to size 1.
       2. The program then uses the MPI communicator to receive the sum of the process’ transformed array, listening from source i(each process rank) and stores it into index 0 of sum\_np. It receives a numpy array of type double.
       3. The sum from sum\_np[0] is then added to the value of world\_sum.
    3. The average is then calculated by dividing the world\_sum by the problem size, N.
24. If the process is not rank 0:
    1. The program initializes a numpy array with the value of the local sum.
    2. The program then uses the MPI communicator to send the local sum array, sum\_np as type double, to destination at rank 0.
25. The program then fetches the wall time(MPI.Wtime()) to get the end time to time the average result.
26. If the rank of the process is 0:
    1. The program prints the average time by subtracting start time from end time.
    2. The program prints the average value as a string.

**MPI Parallelization Strategy**

The program uses MPI to parallelize the program and speed up the computation time for the mathematical calculation. It achieves this by:

1. Taking in an amount of processes to use for the calculation.
2. Assigning the workload size to each process so that each process only handles its own portion of the array transformation and averaging.
3. Using each process to initialize only its portion of original array and transformation array.
4. Using each process to then calculate the element-wise addition of the arrays, for only the workload size of the respective process.
5. Using each process to separately calculate its local array sum, and send that value to the process at rank 0.
6. Using the process at rank 0 to calculate its own local sum, and then receive each rank’s sum and add it to the world sum.
   1. These values are sent and received by the MPI communicator as a numpy array, as MPI needs data being sent as a buffer, and sending it as an array or contiguous blocks of memory are easier to manage.
7. Using rank 0 to divide the world sum by the total problem size and calculate the average of the transformed array.

**Execution Time Table**

| *Number of Processes* | *Initialization of a (sec)* | *Initialization of b (sec)* | *Addition of Arrays (sec)* | *Averaging*  *(sec)* |
| --- | --- | --- | --- | --- |
| 1 | 0.009116100001847371 | 1.3885025000054156 | 2.450641400006134 | 1.5541227999929106 |
| 2 | 0.004646300003514625 | 0.7597167999920202 | 1.2928547999908915 | 0.766707500006305 |
| 4 | 0.002433400004520081 | 0.365204999994603 | 0.6353985999885481 | 0.3850357999908738 |
| 8 | 0.0013736999972024933 | 0.18197319998580497 | 0.3376626999961445 | 0.19807280000532046 |