Homework 3 | MPI | Team 42

Shivam Mittal 2022101105
Nikunj Garg 2021101021

1 Distributed K-Nearest Neighbours

1.1 Solution

For this question I have divided the given points evenly to all the processes and calculated the k nearest neighbours to each query among the points present in each process. Then the rank 0 process collects k points each from all other processes and finds the k nearest neighbours finally.

1.2 Pseudo Code

```
Algorithm
1. Initialize MPI environment
2. Get rank and size of the current process
3. If rank == 0:
    • Read $n$, $m$, $k$, $creal$, $cimg$ from input
    • Create 2D grid of points
    • Calculate total points and distribute them among processes
    • Broadcast $n$, $m$, $k$, $creal$, $cimg$ to all processes
    • Assign points to Process 0 and compute results
    • For each point in Process 0's range:
        – Compute whether point belongs to set using complex number formula
        – Store result in finalans
    • For each process $i$ from 1 to size-1:
        – Calculate start point and number of points for process $i$
        – Send start point and number of points to process $i$
    • For each process $i$ from 1 to size-1:
        – Receive results from process $i$
        – Append results to finalans
    • Print finalans as grid
4. Else:
    • Receive $n$, $m$, $k$, $creal$, $cimg$ from Process 0
    • Receive start point and number of points to process
    • Compute whether each assigned point belongs to set
    • Send results back to Process 0
5. Finalize MPI environment
```

1.3 Complexity

Given:
N: Number of points
M: Number of queries
K: Number of neighbors
p: Number of processors

Time Complexity:
$\mathcal{O}\left(\frac{M}{p} \cdot N\right)$

Space Complexity:
$\mathcal{O}\left(K\right)$

Message Complexity:
$\mathcal{O}(p)$

1.4 Results

Table 1: Execution Time Analysis for Distributing Dataset

No of Processes	**Execution Time (sec) (Distributing Dataset)**
1	1.7874
2	0.6578
3	0.5559
4	0.5874
5	0.5976
6	0.5854
7	0.5773
8	0.5899
9	0.5910
10	0.6596
11	0.6518
12	0.6658

2 Julia Set

2.1 Solution

In this question we split the computation for the n*m points between the p number of processes. First (n*m)mod(p) processes compute for (n*m)/p+1 points and rest of the processes compute for (n*m)/p number of points .We initially broadcast n,m,c to all the processes. After this all the processes compute what points they are allocated and find whether they are in Julia set or not and send the array of results for this to the root process. The root process then aggregates the results of all processes to get the final andswer.

2.2 Pseudo Code

Algorithm

```
1. Initialize MPI environment
2. Get rank and size of the current process
3. If rank == 0:
     • Read $n$, $m$, $k$, $creal$, $cimg$ from input
     • Create a 2D grid of points of size $n \times m$
     • Calculate total points and distribute them among processes
     • Broadcast $n$, $m$, $k$, $creal$, $cimg$ to all processes
     • Assign points to Process 0 and compute results:
         – For each point in Process 0's range:
             ∗ Compute real and imaginary parts
             ∗ Apply complex number iteration to check if the point belongs to
               the set
             ∗ Store result in finalans
     • For each process $i$ from 1 to size-1:
         – Calculate start point and number of points for process $i$
         – Send start point and number of points to process $i$
     • For each process $i$ from 1 to size-1:
         – Receive results from process $i$
         – Append results to finalans
     • Print finalans as grid
4. Else (if rank ¿ 0):
     • Receive $n$, $m$, $k$, $creal$, $cimg$ from Process 0
     • Receive start point and number of points to process
     • For each assigned point:
         – Compute real and imaginary parts
         – Apply complex number iteration to check if the point belongs to the
           set
         – Store result in ans
     • Send results back to Process 0
5. Finalize MPI environment
```

2.3 Complexity

Given:
N: Number of rows
M: Number of columns
K: Maximum number of iterations
p: Number of processors

Time Complexity:
$\mathcal{O}\left(\frac{N\cdot M}{P}\right)$

Space Complexity:

$\mathcal{O}(N \cdot M)$

Message Complexity:
$\mathcal{O}(P)$

2.4 Results

Table 2: Execution Time Analysis

No of Processes	**Execution Time (seconds)**
1	1.75
2	0.45
3	0.42
4	0.42
5	0.42
6	0.43
7	0.43
8	0.43
9	0.44
10	0.44
11	0.44
12	0.44

3 Prefix Sum

3.1 Solution

The provided code utilizes MPI (Message Passing Interface) to compute the prefix sum of an array in parallel. It begins by initializing the MPI environment and determining the rank and size of processes. The master process reads the input array from a file, ensuring the array size is divisible by the number of processes for even distribution. The array is then scattered across all processes, which compute the prefix sum of their respective chunks. Intermediate results are exchanged between processes to maintain a consistent prefix sum across the entire array. Finally, the results are gathered at the master process, which prints the final prefix sum and the elapsed time before finalizing the MPI environment. This parallel approach significantly enhances the performance of the prefix sum computation for large datasets.

3.2 Pseudo Code

Algorithm

```
1. Initialize MPI environment
2. Get rank of the current process
3. Read filename from command line arguments
4. If rank == MASTER_PROCESS:
     • Read the total size $n$ from the input file
     • Adjust $n$ to be divisible by the number of processes
     • Allocate memory for the array and prefix sum
     • Read input values into the array
5. Broadcast $n$ to all processes
6. Calculate the chunk size for each process:
     • Set chunkSize to $n/size$
7. Initialize recvBuffer for each process
8. Scatter the array into recvBuffer for each process using MPI_Scatter
9. Compute prefix sum for the received chunk:
     • Call compute_prefix_sum(recvBuffer, chunkSize)
10. If rank != 0:
     • Receive the value from the previous process
11. Set sendToken to the sum of received value and last element of recvBuffer
12. If rank is not the last process:
     • Send sendToken to the next process
13. Update recvBuffer by adding the received value
14. Gather the updated recvBuffer into prefix sum at MASTER_PROCESS using
    MPI_Gather
15. If rank == MASTER_PROCESS:
     • Print the prefix sum
     • Print the elapsed time
16. Finalize MPI environment
```

3.3 Complexity

Given:
N: Number of points
N: Number of processes

Time Complexity:
$\mathcal{O}\left(\frac{N}{P}\right)$

Space Complexity per process:
$\mathcal{O}(N/p)$

Message Complexity:

$\mathcal{O}(N)$

3.4 Results

Table 3: Performance Scale: For N = 12 (for n = 1e4)

No of Processes	**Execution Time (seconds)**
1	0.000184
2	0.000160
3	0.000165
4	0.000220
5	0.000216
6	0.000260
7	0.000226
8	0.000241
9	0.000289
10	0.000273
11	0.000281
12	0.000213

4 Inverse of a Matrix

4.1 Solution

In this question we found the inverse of matrix using the row reduction method .The input is taken in by the root process. A matrix of size N x 2N is constructed whose left part of size N X N is the initial matrix whose inverse is to be found and the right part of size N X N is the identity matrix which will store the inverse of this matrix at the end. Now I applied the row reduction method which uses row operations to convert the original input matrix to identity matrix applying the same operations to the initial identity matrix and at the end the initial identity matrix is converted into the inverse which we wanted. For parallelization , I have parallelized the row operations which include dividing a row and also the row addition/subtraction operations between the processes . Finally the answer is printed in the root process.

4.2 Pseudo Code

Algorithm

```
1. Initialize MPI environment
2. Get rank of the current process
3. Read filename from command line arguments
4. If rank == 0:
    • Read matrix size $n$ from the input file
5. Get the total number of processes
6. Broadcast $n$ to all processes
7. Calculate the number of rows each process will handle:
    • Initialize row counts for each process
    • Calculate the starting point for each process
8. Initialize local matrices for each process
9. If rank != 0:
    • Receive parts of identity matrix and input matrix using MPI_Scatterv
10. Else (if rank == 0):
    • Read the full matrix from the input file
    • Create an identity matrix of size $n \times n$
    • Scatter parts of identity matrix and input matrix to all processes using
      MPI_Scatterv
11. Perform Gaussian elimination for matrix inversion:
    • For each row $i$ from 0 to $n-1$:
        – If the current process owns the pivot row:
            ∗ Normalize the pivot row
            ∗ Broadcast pivot row to all processes
        – For each row in the local matrix:
            ∗ Eliminate the current column using the pivot row
12. Gather the inverted parts of the identity matrix using MPI_Gatherv
13. If rank == 0:
    • Reorder the rows to their correct global order
    • Print the inverted matrix
14. Finalize MPI environment
```

4.3 Complexity

Given:
N: Number of rows
P: Number of processors

Time Complexity:
$\mathcal{O}\left(\frac{N^3}{P}\right)$

Space Complexity:
$\mathcal{O}(N^2)$

Message Complexity:

$\mathcal{O}\left(\frac{N^2}{P}\right)$

4.4 Results

Table 4: Execution Time Analysis

No of Processes	**Execution Time (sec)**
1	1.5195
2	0.3668
3	0.3638
4	0.3717
5	0.3696
6	0.3701
7	0.3801
8	0.3850
9	0.3900
10	0.3923
11	0.3957
12	0.4054

5 Parallel Matrix Chain Multiplication

5.1 Solution

This MPI-based C program reads matrix data from a file, partitions it among multiple processes, and uses dynamic programming to solve a matrix chain multiplication problem in parallel. The master process (rank 0) reads the input, adjusts the data size for even distribution, and sends relevant partitions to slave processes. Each process computes part of the dynamic programming (DP) table, which stores the minimum cost of multiplying submatrices. Processes communicate DP values with neighboring processes to ensure consistency. Finally, the master process gathers the results, outputs the minimum multiplication cost, and reports the computation time.

5.2 Pseudo Code

Algorithm

```
1. Initialize MPI environment
2. Get rank and size of the current process
3. If rank == 0 and no input file is provided:
     • Print error message and abort MPI
4. If rank == MASTER_PROCESS:
     • Read input data from the file
     • Adjust the size of the input data to be divisible by the number of processes
     • Allocate memory for the input array
     • Fill the array with input data, padding with zeros if necessary
5. Broadcast the size of the array $n$ to all processes
6. Start timing the computation
7. Initialize the DP table with size $n \times n$
     • Set base cases in DP table where $i = j$ (diagonal elements) to 0
8. If rank == MASTER_PROCESS:
     • Send relevant data to slave processes
9. Else (if rank != MASTER_PROCESS):
     • Receive data from the master process
10. Perform dynamic programming (DP) to fill the DP table:
     • For each length $len$ from 2 to $n$:
         – Calculate the start index for the current process
         – For each row $i$ in the process's assigned partition:
             ∗ Calculate the column index $j$
             ∗ If $j \geq n$, break the loop
             ∗ Set $dp[i][j]$ to a large value (INF)
             ∗ For each possible split point $k$:
                 · Calculate the cost of multiplying the submatrices
                 · Update $dp[i][j]$ if a lower cost is found
             ∗ For each slave process within communication range:
                 · Receive updated DP values from the appropriate processes
             ∗ Send the updated DP values to the appropriate processes
11. End the timing of the computation
12. If rank == MASTER_PROCESS:
     • Print the final result from $dp[0][input\_size - 1]$
     • Print the elapsed time
13. Finalize the MPI environment
```

5.3 Complexity

Given:
N: Number of rows
P: Number of processors

Time Complexity:
$\mathcal{O}\left(\frac{N^3}{P}\right)$

Space Complexity:
$\mathcal{O}(N^2)$

Message Complexity:
$\mathcal{O}\left(N^2\right)$

5.4 Results

Table 5: Performance Scale: For N = 12 (for n = 1e3)

No of Processes	**Execution Time (seconds)**
1	1.997616
2	1.867445
3	10.06791
4	6.274964
5	4.561547
6	5.424868
7	4.830780
8	4.406806
9	12.04081
10	16.00027
11	14.00028
12	18.00021