Lab 00

Varun Shankar Hoskere
111361326
September 4th 2024

Table of Contents

Parallel design

In order to parallelize the linear interpolation, I processed portions of the xval buffer on each processor. Each processor had access to the entire buffer, through the MPI_Gather method, but only computed the associated yval associated with its portion - albeit the computation was through an intermediary buffer yvallocal.

The processors and the portions of yval that they computed are as follows :

- Processor 0 computed yval[0] - yval[4]
- Processor 1 computed yval[5] - yval[9]
- Processor 2 computed yval[10] - yval[14]
- Processor 3 computed yval[15] - yval[19]

These values were stored in an intermediary buffer yvallocal, and these yvallocal buffers of the processors were gathered into the final yval buffer.

Self-evaluation

I had initially written the parallelizing code in the lookup function itself - instead of having a parallelized code and calling the lookup function individually. This posed a big problem. The lookup function in that initial design did not have a return value. I had wrongly thought that the buffers did not need to be broadcasted explicitly and had assumed that the global variable access would be sufficient to update the yval buffer. After working for a while I corrected my mistake and concluded that I needed to redesign my solution. After realizing that I had to use the MPI_Bcast( ) method, I also considered the use of the MPI_Scatter( ) method. Since, the question stated that each processor should have access to the entire buffer and not just segments, I discarded its potential.

In the redesign I decided to stick with the single lookup function - not make any changes to the function that was implemented in class. I moved the parallelizing portion of the code to the main( ) function instead of the lookup( ) function. Then I was able to incorporate the processor rank checks and compute the yval values accordingly.

Since the question prescribed the number of processors to be 4, the solution I have implemented only works with that number. There are ways to make this dynamic. The number that is passed while entering the command to run the .exe can be accessed within the file through the argv variable. That number represents the number of processors and, by implication, also the number of chunks we would need to create from xval and compute the associated yval chunk. Then comes the issue of computing the indexes required during the iteration of xval and computation of yval. To compute the starting index for each processor, all we need to do is multiple the rank of the processor with the chunk size. The chunk size would simply be the size of the xval and yval buffers, in this case m, divided by the number of chunks, n - specifically 4 in this scenario. Once we have each starting index the ending indexes can be computed by adding the chunk size to the starting index.

I thought of this dynamic generalization after completing the lab and decided that it is an optimization that can be implemented by making minor modifications to my code.

Another places where I had to relearn was with the creation of slurm scripts. I had to unlink the working of slurm scripts and makefiles in order to understand its working properly and come up with the correct slurm script.

Coming to the outcome, we can see that the observed output is the linear interpolation of the xval values. Linear interpolation is an approximation method that fits a straight line between two known points and calculates the y value for a given x value. From the results we can tell that the interpolation approximates the function $y = x^2$. By using parallel processing, the computation takes less time and this can be quantified by recording the start and end times for the parallel code. The code can then be run in sequential fashion without parallel implementation in order to compare the two approaches.

Appendix A : Parallel code

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

using std :: cout;
using std :: endl;

double lookup(int n, double *x, double *y, double xval)
{
  for ( int i = 0 ; i < n ; ++i)
      if ( xval >= x[i] && xval <= x[i+1] )
      return y[i] + (xval - x[i]) * (y[i+1]-y[i]) / (x[i+1]-
x[i]);
  return 0;
}

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

    int numPE, myPE;

    // Initializing number of processors and the rank of the
processor
    MPI_Comm_size(MPI_COMM_WORLD, &numPE);
    MPI_Comm_rank(MPI_COMM_WORLD, &myPE);


    //Initilazing the basic data structures that are needed for
the linear interpolation task
    int n = 100;
    double x[n], y[n];

    for (int i = 0; i<n; ++i) {
        x[i]=i;
        y[i]=i*i;
    }


    /*
     Creating the data structures for xval and yval and
yvallocal.
     yvallocal is used by each individual processor to store the
output of the linear interpolation.
```

```
    */

    int m = 20;
    double xval[m], yvallocal[5], yval[m];

    for (int i = 0; i<m; ++i) {
        xval[i] = 2*i;
        yval[i]=0;
    }

    /*
     Broadcasting the values x, y and the array xval so that
each other processor can use the values in order to compute
     the associated yval value.
     */

    MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(y, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(xval, m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
//    MPI_Bcast(yval, m, MPI_DOUBLE, 0, MPI_COMM_WORLD);



    /*

     Each process will only compute the yval values associated
with portions of xval. Since each processor should
     perform roughly the same amount of work, the total of m=20
values can be divided equally among the 4 processors.

     Each processor will compute 4 yval values and store them in
yvallocal buffer. All the yvallocal buffers are then finally
     gathered and merged into the yval buffer.


     */

    if( myPE == 0) {
        for (int j=0; j<5; ++j) {
            yvallocal[j] = lookup(n, x, y, xval[j]);
        }
    }

    /*
```

```
     Indices for the yvallocal buffer need to be computed
properly for processors 1,2 and 3.

     */

    else if ( myPE == 1) {
        for (int j=5; j<10; ++j) {
            yvallocal[j-5] = lookup(n, x, y, xval[j]);
//            cout << yvallocal[j] << endl;         Line used
for debugging
        }
    }
    else if ( myPE == 2) {
        for (int j=10; j<15; ++j) {
            yvallocal[j-10] = lookup(n, x, y, xval[j]);
        }
    }
    else if ( myPE == 3) {
        for (int j=15; j<20; ++j) {
            yvallocal[j-15] = lookup(n, x, y, xval[j]);
        }
    }

    // Gather the yvallocal buffers into the yval buffer at
process 0.
    MPI_Gather(yvallocal, 5, MPI_DOUBLE, yval, 5, MPI_DOUBLE,
0 , MPI_COMM_WORLD);

    if(myPE==0) {
        cout << "Final yval values: " << endl;
        for(int i=0;i<m;i++) {
            cout << "yval[" << i << "]: " << yval[i] << endl;
        }
    }

    MPI_Finalize();
    return 0;

}
```

Appendix B : Slurm script

```
#!/bin/bash

# Setting the slurm parameters

#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --time=00:01:00
#SBATCH --partition=amilan

#Loading the modules needed
module purge
module load gcc
module load openmpi


echo "|| Begin execution of lab0.exe slurm batch script."

# Running the executable and routing the output to an external file
mpirun -n 4 ./lab0.exe > lab0output.txt


echo "|| Execution of lab0.exe in slurm batch script complete."
```

Appendix C : Output

```
Final yval values:
yval[0]: 0
yval[1]: 4
yval[2]: 16
yval[3]: 36
yval[4]: 64
yval[5]: 100
yval[6]: 144
yval[7]: 196
yval[8]: 256
yval[9]: 324
yval[10]: 400
yval[11]: 484
yval[12]: 576
yval[13]: 676
yval[14]: 784
yval[15]: 900
yval[16]: 1024
yval[17]: 1156
yval[18]: 1296
yval[19]: 1444
```