INDIAN INSTITUTE OF TECHNOLOGY KANPUR

CS 633A Semester 2020–2021-II

Assignment III

Members
Hirak Mondal
Roll-20111022

Chandan Kumar
Roll-20111015

Professor:
Dr. Preeti Malakar
Department of Computer Science

Approach to the problem

Say we have these data points which has 5 rows and 7 columns along with the headers.

$$\begin{bmatrix} Lat & Lon & 1960 & 1961 & 1962 & 1963 & 1964 \\ 44.6547 & -88.9592 & -5.67 & -8.26 & -10.7 & -14.13 & 5.15 \\ 51.8167 & -111.4667 & -14.23 & -9.64 & -12.35 & -16.82 & -10.76 \\ 41.4831 & -83.711 & -0.87 & -0.77 & -4.72 & -8 & -1.22 \\ 64.783 & 170.1166 & -25.19 & -21.96 & -20.88 & -19.4 & -34.94 \end{bmatrix}$$

At first we are reading the data with process 0 and storing it in a matrix without the headers and latitude and longitude columns as shown below.

$$\begin{bmatrix} -5.67 & -8.26 & -10.7 & -14.13 & 5.15 \\ -14.23 & -9.64 & -12.35 & -16.82 & -10.76 \\ -0.87 & -0.77 & -4.72 & -8 & -1.22 \\ -25.19 & -21.96 & -20.88 & -19.4 & -34.94 \end{bmatrix}$$

Now say we have 2 processes, **P0** and **P1**, so we must divide the work between them. For that we first need to send the data from P0 to P1. For Data Distribution we have taken the help of **MPI_Pack()** function. For 2 processes we have divided the columns. P0 operates on the first 2 columns

$$\begin{bmatrix} -5.67 & -8.26 \\ -14.23 & -9.64 \\ -0.87 & -0.77 \\ -25.19 & -21.96 \end{bmatrix}$$

It takes the minimum from each columns and store it in an array.

$$\text{arr} = \begin{bmatrix} -25.19 & -21.96 \end{bmatrix}$$

and P0 packs the next 3 columns using MPI_Pack and sends it to P1.

$$\begin{bmatrix} -10.7 & -14.13 & 5.15 \\ -12.35 & -16.82 & -10.76 \\ -4.72 & -8 & -1.22 \\ -20.88 & -19.4 & -34.94 \end{bmatrix}$$

Now P1 unpacks it and calculates minimum from each of the columns and stores the result in the array.

$$\text{arr}= \begin{bmatrix} -20.88 & -19.4 & -34.94 \end{bmatrix}$$

Now P1 send this array to P0 using MPI_Type_contiguous. So after receiving this, P0 has the array,

$$\text{arr}= \begin{bmatrix} -25.19 & -21.96 & -20.88 & -19.4 & -34.94 \end{bmatrix}$$

which contains the year-wise minimum across all stations. Which we are storing in the **output.txt** file and then from this array we are calculating the global minimum, which is nothing but the minimum element in this array.

Global Minimum (for this above example)= -34.94

Code Explanation

We will like to give the explanation for 4 processes. For 1,2,8 processes the function scales similarly.

At first we are reading the data using process 0.

```
if(size==4)
{
  if(myrank==0)
    {
    //Open the data file (name provided in command line input)
    //count number of rows and columns
    //store data (only temperatures) in a matrix
    }
}
```

Listing 1: Reading the data using P0

After that since we have 4 processes we are dividing the total number of columns into $\frac{columns}{4}$, for n processes it would have been $\frac{columns}{n}$. Now process 0 will keep 0 to (c/4 -1) columns to itself and will pack the remaining columns

```
1
2 //we are packing the remaining portions after c/4
3       for(long long int j=c/4;j<c;j++)
4       {
5         position=0; //reassigning position after each and every
      send
6
7         for(long long int i=0;i<r;i++)
8         {
9           MPI_Pack(&mat[i][j], 1 , MPI_DOUBLE,snd_buf[j-c/4],outs,&
      position,MPI_COMM_WORLD);
10        }
11      }
```

Listing 2: Packing the columns by P0

And then sending them to P1 P2 and P3

```
1
2 //sending to P1 and P2 from P0
3       for(int i=1;i<3;i++)
4       {
5
6         for(long long int j=(i*c/4); j<( (i*c/4) + c/4) ; j++)
7         {
8           MPI_Isend (snd_buf[j-c/4], outs , MPI_PACKED, i /*dest*/
      , j /*tag*/ , MPI_COMM_WORLD,&request[j]);
9         }
10      }
11
12
13
14      //sending to P3
15      for(long long int j=(3*(c/4)); j<c ; j++)
16      {
17        MPI_Isend (snd_buf[j-c/4], outs , MPI_PACKED, 3 /*dest*/ ,
      j /*tag*/ , MPI_COMM_WORLD,&request[j]);
18      }
```

Listing 3: Sending the values from P0

After that P0 finds the minimum elements from (0 to c/4-1) columns on which it is operating and stores them in an array.

```
1 //min1 will store the minimum for rank0
2       double min1=1000;
3
```

```
4       for(long long int j=0;j<c/4;j++)
5       {
6         min1=1000;
7
8         for(long long int i=0;i<r;i++)
9          {
10          if(mat[i][j]<min1)
11          min1=mat[i][j];
12         }
13
14          arr2[j]=min1;
15
16
17      }
```

Listing 4: Calculating minimum of 1/4th of the columns by P0

P0 also broadcasts the total number of rows and columns of the data matrix to other processes as well

After that P0 finds the minimum elements from (0 to c/4-1) columns on which it is operating and stores them in an array.

```
1
2 //Broadcasting the value of r and c to other process from the root
     process after reading the data
3
4    MPI_Bcast(&r, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
5    MPI_Bcast(&c, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Listing 5: Broadcasting the value of r and c to other process from the root process

Now process 1,2 and 3 receives and unpacks the data

```
1
2 if(myrank==1)
3     {
4
5        for(long long int j=c/4;j<(c/4 + c/4);j++)
6        {
7          MPI_Irecv(recv_buf[j-c/4], outs, MPI_PACKED, 0 /*src*/ , j
     /*tag*/, MPI_COMM_WORLD, &request1[j]);
8          MPI_Wait(&request1[j], &status[j]);
9        }
10
11      for(long long int j=c/4;j<(c/4 + c/4);j++)
12      {
```

```
13          position=0;
14           for(long long int i=0;i<r;i++)
15           {
16            MPI_Unpack(recv_buf[j-c/4],outs,&position,&buf[j-c/4][i],
    1/*outcount*/, MPI_DOUBLE, MPI_COMM_WORLD);
17           }
18        }
```

Listing 6: Code Snippet of Receiving the data by P1 followed by unpacking

And then each process calculates minimum of each column

```
1
2 if(myrank==1)
3   {
4
5       for(long long int j=c/4;j<(c/4 + c/4); j++)
6       {
7         min1=1000;
8
9         for(long long int i=0;i<r;i++)
10         {
11          if(buf[j-c/4][i]<min1)
12          min1=buf[j-c/4][i];
13         }
14
15          arr2[j]=min1;
16        // printf(" %lf", arr2[j]);
17
18
19      }
20  }
```

Listing 7: Code Snippet of calculating minimum of each column by P0

Each process then sends the array of minimum values by using MPI_Type_contiguous to the root process.

```
1
2 //for process 1 and 2, size will be c/4
3     MPI_Datatype type;
4       MPI_Type_contiguous( c/4 , MPI_DOUBLE, &type );
5       MPI_Type_commit(&type);
6
7
8 //now sending the data from P1
9     if(myrank==1)
```

```
10      {
11
12
13          MPI_Isend (&arr2[c/4], 1 , type, 0 /*dest*/ , 100 /*tag*/ ,
        MPI_COMM_WORLD ,&request2[1]);
14
15      }
```

Listing 8: Code Snippet of P1 sending data to P0

P0 receives them all in an array and writes them in the **output.txt file**.

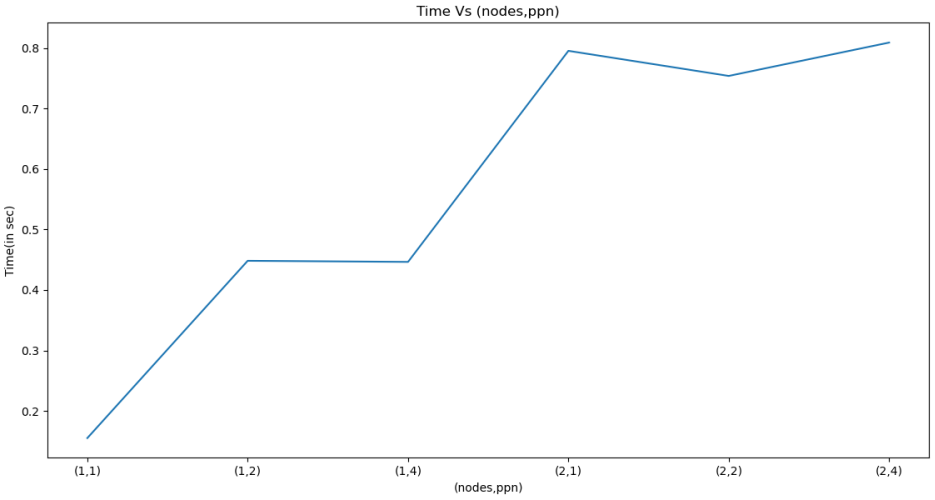
Moreover it also finds the minimum value of the array which is the global minimum across all stations and all years and stores it in output.txt file along with the time.

Reason for choosing MPI_Pack

At first we thought of using MPI derived data type, like MPI_Type_contiguous() or MPI_Type_vector(). But in the first assignment we saw that MPI_Pack and MPI_Type_contiguous() sharing almost the same time, so we have decided to go with MPI_Pack in this case. Moreover we felt its more convenient to pack the entire data and send it to a process rather than sending columns using MPI_Type_vector or MPI_Type_Contiguous one by one. One more problem with MPI Derived Datat Type is that if the buffer described by the derived datatype is not contiguous in memory, it may take longer to access.

Plot

We repeated each configuration 5 times and have taken the median value of them, to ensure that our data has no outlier.



Observations

• For one node the time is lowest when we have 1 node and 1 process, but time increases when we have 1 node 2 process and remains almost same for 1 node 4 process. In these cases the data distribution time dominates over the computation time.

• In case of 2 nodes,when each node has a process 2 processes, i.e. total 4 process then the time decreases from 2 nodes 1 ppn. This means that computation time is dominating over communication time and thus for 2 nodes and 2 ppn we are able to calculate a little bit faster than 2 nodes and 1 ppn. But the time again slows down for 2 nodes and 4 process per node as the data distribution time increases during that time.

•The scalability of a parallel algorithm on a parallel architecture is a measure of its capacity to effectively utilize an increasing number of processors. We can observe that for 1 node and 4 ppn the time taken is slightly lesser than 1 node and 2 ppn (looks almost same in the graph though), which means that by increasing the number of processors we are able to get a speed up, be it in very little amount. Again for 2

nodes and 2 ppn we can see that we get a good speed up than 2 node and 1 ppn. So here also our code is scaling good.