INDIAN INSTITUTE OF TECHNOLOGY KANPUR

CS 633A Semester 2020–2021-II

Assignment I

Members
Hirak Mondal
Roll-20111022

Chandan Kumar
Roll-20111015

Professor:
Dr. Preeti Malakar
Department of Computer Science

CODE EXPLANATION

Approach to the problem

Say we have 9 processes and each process has 16 Data Points. So we can say that each process has a matrix of data points of size $4 * 4$. Now we can represent these 9 processes also in a matrix of $3 * 3$

$$\begin{bmatrix} P0 & P1 & P2 \\ P3 & P4 & P5 \\ P6 & P7 & P8 \end{bmatrix}$$

Now see that in the worst case, like P4, it has to do Halo Exchange with all 4 of its neighbours. And in the best case like the corner elements of the matrix for example P2 it has to do exchange only with P1 and P5. So this is the virtual diagram of the process's topology that we have used to solve this assignment. To receive elements from the neighbouring process we have allocate 4 buffers to each process (keeping in mind that in the worst case there are 4 HALO exchanges). The length of the buffer is kept same as the length of the rows and are initialised to 0. While doing stencil computation what we are doing is that, we are adding elements from these buffers and averaging them based on the number of their neighbours. That is, if that process have 4 neighbours, then we are diving it by 4. If it has 3 neighbours then we are dividing it by 3 and similarly by 2 if we have 2 neighbours.

We are repeating the communication for 50 times and after which we are noting the time to complete the entire exchange. This step is again repeated for 5 times, which means that we are recording 5 times for each method.

To generalize our logic, lets say we have P^2 processes and each Process has N^2 data points. So each Process P_i has a matrix of data point of size $N * N$. Now say, $D = \sqrt{P^2}$ i.e. $D = P$

Now a process P_i will only exchange data with P_{i-1} if and only if P_i is not present in the leftmost column. That is in the above matrix P0, P3, P6 cannot exchange data with their corresponding P_{i-1} since they are present in the left most column. We are checking this by using this logic

```
if(myrank%P!=0) //P= sqrt(number of processes)
{
//then this is not a process which is present in the leftmost
   column and has a left neighbour, hence we can do data exchange
   with the left neighbour
}
```

Listing 1: Logic to check whether there is a left neighbour or not

Similarly to check whether the process P_i is present in the rightmost column i.e. no HALO exchange is possible with P_{i+1} we have used this logic

```
if( (myrank+1)%P !=0) //P=sqrt(number of processes)
{
//then this is not a process which is present in the rightmost
   column hence we can do data exchange with the right neighbour
}
```

Listing 2: Logic to check whether there is a right neighbour or not

Now we also have to check whether there is a neighbour at the top or not, so we have checked it like this

```
if(myrank-D>=0)//D=sqrt(number of processes)
{
//then this is a process which has a top neighbour hence we can do
   data exchange with the top neighbour
}
```

Listing 3: Logic to check whether there is a top neighbour or not

and then we are checking if their is a neighbour at the bottom or not

```
if(myrank+D<size) //D=sqrt(number of process)
{
//then this is a process which has a bottom neighbour hence we can
   do data exchange with the bottom neighbour
}
```

Listing 4: Logic to check whether there is a bottom neighbour or not

Now once these are checked we are now ready to send data to the neighbouring processes using the following methods mentioned below.

Methods Used

• Multiple MPI_Sends, each MPI_Send transmits only 1 element (1 double in this case).

• MPI_Pack/MPI_Unpack and MPI_Send/MPI_Recv to transmit multiple elements at a time

• MPI_Send/Recv using MPI derived datatypes (contiguous, vector, ... whichever is suitable)

a. Multiple MPI_Sends

We have used non-blocking MPI_Isend() for this and we are sending the desired row/column to the neighbouring process's buffer transferring only 1 element at a time of that row, which gets stored in the buffer of the receiver. We are using MPI_Wait() after MPI_Irecv() to ensure that our transfer is successful.

An example is provided here

```
MPI_Isend (&m[i][j], 1 , MPI_DOUBLE, myrank-1, i, MPI_COMM_WORLD,&
    request[myrank-1]);
```

Listing 5: Sending a single Data Element using MPI_Isend()

```
MPI_Irecv(&buf3[i], 1, MPI_DOUBLE, myrank+1, i, MPI_COMM_WORLD, &
    request1[myrank+1]);
        MPI_Wait(&request1[myrank+1], &status[myrank+1]);
```

Listing 6: Receiving a single Data Element using MPI_Irecv()

b. MPI_Pack/MPI_Unpack and MPI_Send/MPI_Recv

While using MPI_Pack() along with MPI_Isend() we are now NOT sending single data elements one by one. When we are doing a HALO exchange with our neighbouring process where we need to send our row/column, we are packing the entire data together and sending it using MPI_Isend(). An example is attached here.

```

if(myrank-d>=0) //i.e. top neighbour exists
    {
      //we will send the last row to the buffer2 of that process
      for(int i=0;i<1;i++)
      {
        for(int j=0;j<count;j++)
        {
          MPI_Pack(&m[i][j], 1 , MPI_DOUBLE,snd_buf2,outs,&
    position2,MPI_COMM_WORLD);
        }
      }

      MPI_Isend (snd_buf2, count , MPI_PACKED, myrank-d, 2,
    MPI_COMM_WORLD,&request[myrank-d]);

    }
```

Listing 7: Sending Data using MPI_Pack() and MPI_Isend()

And on the receiving side we are unpacking it and then storing it in the buffer. An example is provided here

```

if(myrank+d<size)
    {

        MPI_Irecv(recv_buf4, count, MPI_PACKED, myrank+d, 2,
    MPI_COMM_WORLD, &request1[myrank+d]);
        MPI_Wait(&request1[myrank+d], &status[myrank+d]);

        for(int i=0;i<count;i++)
        {
         MPI_Unpack(recv_buf4,outs,&position4,&buf4[i],1, MPI_DOUBLE
    , MPI_COMM_WORLD);
        }
    }
```

Listing 8: Receiving Data using MPI_Unpack() and MPI_Irecv()

c. MPI_Send/Recv using MPI derived datatypes

In derived data type we have used MPI_type_contiguous() to serve our purpose. When transferring the rows to the neighbouring processes we are sending them contiguously from the current process's data matrix

```

MPI_Type_contiguous( count, MPI_DOUBLE, &type );
MPI_Type_commit(&type);


    if(myrank-d>=0)
    {
      //we will send the first row to the buffer2 of that process

      MPI_Isend (&m[0][0], 1 , type, myrank-d, 2, MPI_COMM_WORLD,&
    request[myrank-d]);

    }
```

Listing 9: MPI_type_contiguous() and MPI_Send()

But while sending the columns of the data matrix of the current process to the neighbouring process we cant directly send them using MPI_type_contiguous(), so we have first copied the boundary columns that we want to send to the neighbouring processes in a buffer, and then using MPI_type_contiguous() to send it.

```

```

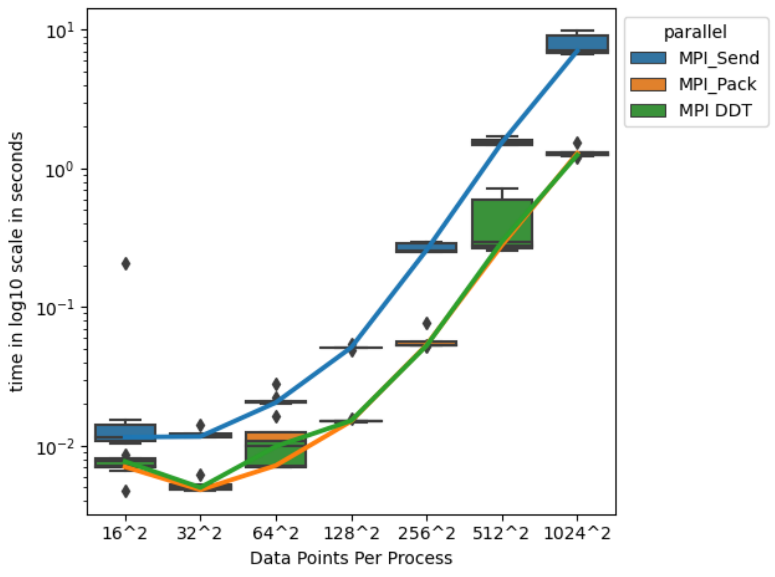
```

if(myrank%s!=0)
    {

      //we will send the left most column of this matrix using buf1
    of this process to the buffer3 of that process
      for(int i=0;i<count;i++)
      {
        for(int j=0;j<1;j++)
        {
          //MPI_Pack(&m[i][j], 1 , MPI_DOUBLE,snd_buf1,outs,&
   position1,MPI_COMM_WORLD);

          snd_buf1[i]=m[i][j];
        }
      }


      MPI_Isend (snd_buf1, 1 , type, myrank-1, 1, MPI_COMM_WORLD,&
   request[myrank-1]);
    }

```

Listing 10: MPI_type_contiguous() and MPI_Send()

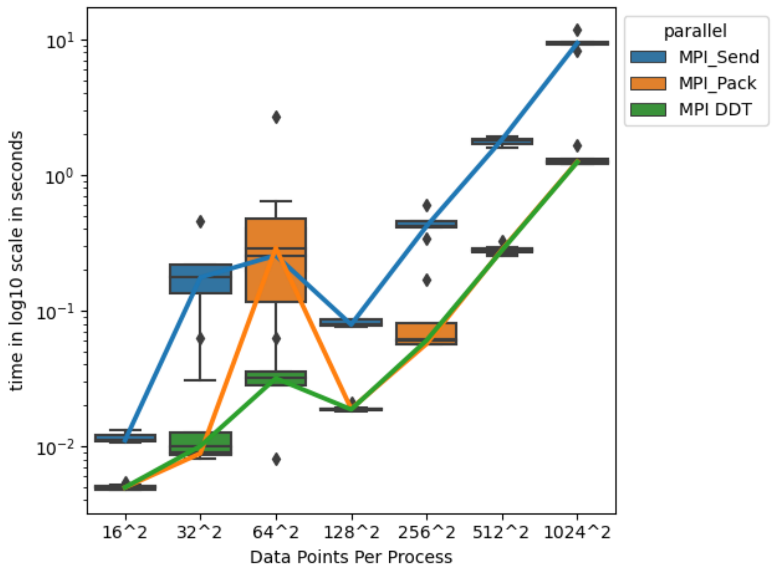
This is how we have implemented our 3 methods

PLOTS

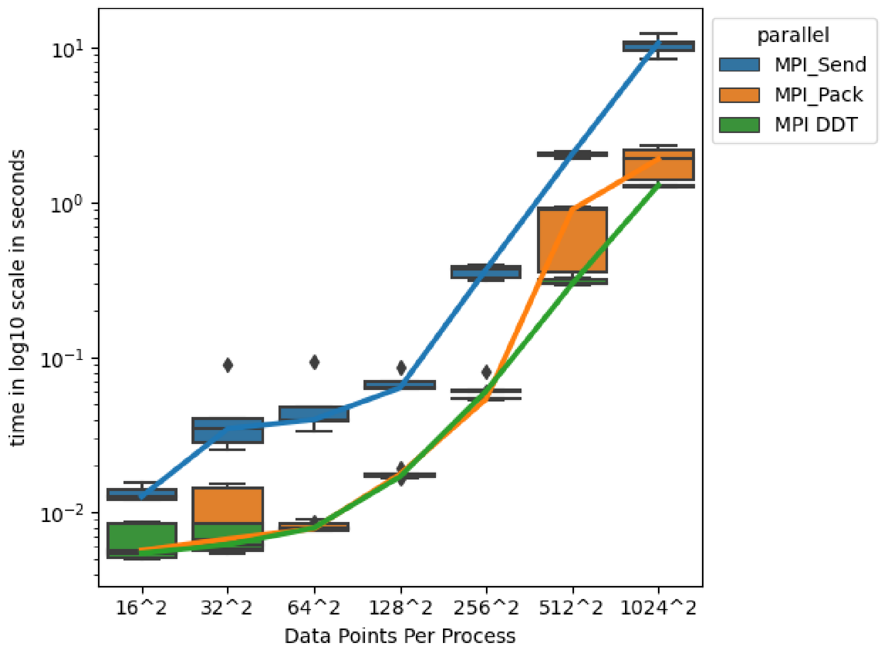
a.Plot for 16 Processes



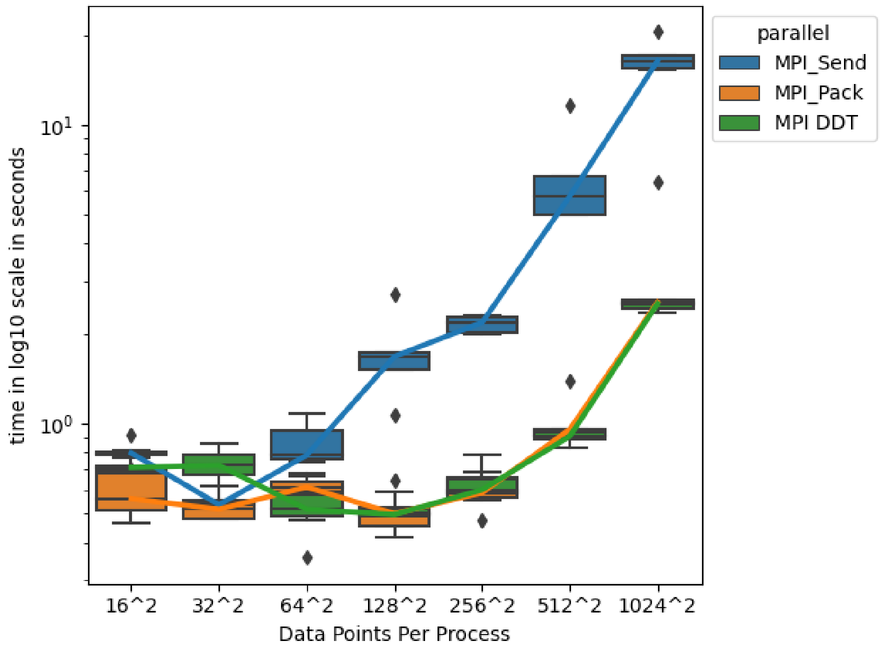
b.Plot for 36 Processes



c.Plot for 49 Processes



d.Plot for 64 Processes



Observations Regarding Performance From The Plots

As we can see that, in all the plots, using **MPI_Isend()** alone i.e. the method 1 is the slowest out of the other 2 methods where we use MPI_Pack() and MPI_type_contiguous(). This is because while using MPI_Isend() alone we are only sending one data element at a time. This **increases** the number of MPI calls across processes (which tend to be a parallelization bottleneck)

Other than this, MPI_Pack() and MPI_type_contiguous() takes more or less the same amount of time in each case, since we are packing and sending the entire row/column and also we are sending the entire row/column at a time using contiguous. The fact that MPI_pack() is some fraction of seconds slower than MPI_type_contiguous() can be attributed to the fact that packing and unpacking takes extra amount of time in MPI_Pack and MPI_Unpack().

In the second plot for $P = 36$ and for $N = 64^2$ we can see that there is a sharp rise in MPI_Pack() time which can is an unexpected case not observed during other runs and may be caused due to network overload.

NOTE$\rightarrow$ The line graphs are joining the median of each box plot