Assignment-1

CS633

Readme

Execution Commnad : python3 run.py
Execution of this command leads to generation of 4 data files corresponding to each process when no of processes=4 and 4 plots corresponding to these text files. Matplotlib is used in the plotscript.py , so the node used for execution should have matplotlib installed in it.
==
The Assignment1 directory consists of the following files :

1. Makefile
2. plotscript.py
3. run.py
4. src.c
5. plot16.png
6. plot32.png
7. plot49.png
8. plot64.png
9. data_16.txt
10. data_32.txt
11. data_49.txt
12. data_64.txt
13. readme.pdf

plotscript.py - Python script that contains the code required to print the plots of the respective processes

run.py - Python script that compiles the code using the "make" command and activating the deamons . It is starting the monitor and also dynamically generating the host files(we used NodeAllocator for the same) and then running the code. Then once it finishes it has generated "4" text files , corresponding to the 4 processes , and then the plotscript is called which has the work of constructing the required boxplots.

src.c - This c file contains the code for the execution of the 3 methods required .

Then the 4 plots correspond to the respective processes and the 4 text files contain the respective time obtained for the given no of processes.The 4 text files are such that , the three contiguous entries of the file , correspond to the time taken by the different procedures used (Mpi_send/recv , packed/unpacked , vector derived datatype) for matrix size =N*N(N = $\sqrt{Datapointsperprocess}$. For example , for the entries of file data_16.txt, the first entry corresponds to procedure1 for matrix size = 16*16 , second entry to procedure2 for same matrix size and third entry corresponds to the timings corresponding to procedure3 . The 4th entry corresponds to the timing of procedure1 for matrix size = 32*32 and so on.

The assignment required us to perform halo exchange using 3 methods :

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

(2) MPI_Pack/MPI_Unpack and MPI_Send/MPI_Recv to transmit multiple elements at a time.

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

We considered 4 conditions which needed to be taken care of for the possible transmission to happen (The value of P = $\sqrt{totalprocesses}$:

a) ((myrank % P)-1 >=0) : This condition is used to check whether the process is able to send its leftmost column to the process with myrank = myrank - 1 and able to accept its rightmost column or not . If this condition is met , then only the required send and receive process is performed else not.

b) ((myrank%P)+1<P) : This condition is used to check whether the process is able to send its rightmost column to the process with myrank = myrank - P and able to accept its leftmost column or not . If this condition is met , then only the required send and receive process is performed else not.

c)(comp>0) : This condition is used to check whether the process is able to send its uppermost row to the process with myrank = myrank - P and able to accept its lowermost row or not . If this condition is met , then only the required send and receive process is performed else not.Here the value of comp is myrank/P.

d)(comp<P-1) : This condition is used to check whether the process is able to send its lowermost row to the process with myrank = myrank + P and able to accept its uppermost row or not . If this condition is met , then only the required send and receive process is performed else not.Here the value of comp is myrank/P.

In the communication part, the problem we faced was when we tried to execute our code for large values of no of processes or matrix size , it gave us errors. These errors were removed once we allocated Dynamic continuous memory allocation to the arrays.

Next we were required to perform the communication and the stencil computation .

For the communication part , we performed the following :

So for the (1) part , we use MPI_Send and MPI_Recv to transmit data. We transmitted 1 byte at a time from the sender to the receiver.

For the (2) part, we were required to send multiple bytes of data at a time , and hence we use MPI_Pack and MPI_Unpack for the same . The "N" data items(of the required row or column) were packed and sent to the receiver using MPI_Send and similarly received by the receiver using MPI_Recv and unpacked using MPI_Unpack to obtain the required transmitted bytes (of the required row or column). In this , we faced the problem of buffer overflow , which got resolved by initialising the position value as '0' before packing .

For the (3) part, we were required to use MPI derived datatypes , and hence we used vector datatype. We used MPI_Type_vector and created two different vectors for row values(rowtype) and column values(columntype).

Next for the stencil computation part , we considered different cases and computed the required cell values . At the time of computation we used 4 buffers to receive the values we are obtaining from the different neighbours ::

a)left_arr[i] - Used to store the values of the left column obtained from the process of myrank-1 , if my current process has rank=myrank.

b)right_arr[i] - Used to store the values of the left column obtained from the process of myrank+1 , if my current process has rank=myrank.

c) top_arr[i] - Used to store the values of the left column obtained from the process of myrank-P , if my current process has rank=myrank.

d) bottom_arr[i] - Used to store the values of the left column obtained from the process of myrank+P , if my current process has rank=myrank.

These values were then used to perform the stencil computation and find out the new values of the matrix.

The stencil computation was also different for different processes based on the boundary conditions . The following conditions were considered ::

1. myrank==0
2. myrank==P-1
3. myrank==P*(P-1)
4. myrank==P*P-1
5. myrank>0 myrank<P-1
6. myrank>P*(P-1) myrank<P*P-1
7. myrank%P==0 myrank>0 myrank<P*(P-1)
8. myrank%P==P-1 myrank>P-1 myrank<P*P-1
9. else for all the other cases

PLOT GENERATION ::

In order to create the plots, we used matplotlib library . We created the plotscript.py file , which contains the function fn(iter,myarr,matsize,process) wherein the input arguments , iter represents the number of iterations , myarr contains the time corresponding the process whose plots are currently being made , matsize refers to the number of matrices being considered(number of different data points per process) and process represents the number of processes for which the plot is currently being made.

This creates the required boxplots and also three lines which pass through each of the median of the boxplots. We used a deflection of "0.8" during the plotting , so that the boxplots for one x value don't get overlapped with each other.We used log scaling for the y-axis due to the small range of values.The x-axis of the graph represent N= $\sqrt{data\_points\_per\_process}$ and y-axis contain the time corresponding to each in log scaled value. The red line graph and boxplots correspond to the Type1 (Multiple Send/Recv) , the blue line graph and boxplots correspond to the Type2 (Packed/Unpacked) and green corresponds to the Type3 (Derived Datatypes i.e. vector).

We observed that the time taken by part(1) i.e. multiple MPI_Sends , is generally higher than when data is communicated using MPI_Pack / MPI_Unpack , which also generally reports higher time than when we use MPI_Type_vector . Hence , from most of the executions , we saw that MPI_Type_vector generally provides more efficient performance than when MPI_Pack / MPI_Unpack is used , which in turns performs better than when multiple MPI_Sends are used.