CS633 PARALLEL COMPUTING

README

April 19, 2021

The execution command to run the code is python3 run.py
We are using allocator.out for generating the hostfile dynamically , so we require /UGP folder in **HOME** directory -> /user/mtech/$username

————————————————————————

The Assignment3 directory consists of the following files :

1. Makefile
2. run.py
3. src.c
4. output.txt
5. plot.png
6. readme.pdf

The Assignment required us to read the data from the csv file given i.e. 'tdata.csv' and find a) the minimum value from each of the column present i.e. the yearly minimum values b) the global minima value c) the maximum time among all the processes.

We first calculated the number of columns and rows present in the csv file and initialized an array **arr** of size no_of_rows*no_of_columns to store the csv data.

After that we checked that whether number of processes > 1 or not. If number of processes =1 , then we plainly calculate the minimum from each of the respective columns via the function **min_of_each_col**. This function takes in an array, number of rows and number of columns in that array and gives back an array which will have the minimum value from each of the present columns. After that we also found the global minima and along with these two information , the total time is then stored in the output.txt file.

If the number of processes != 1 , then we did the following steps ::

i) Broadcasted the value of number of rows and number of columns to each of the processes by using the **bcast** collective call.

ii) Next we sent different rows to different processes. We tried to make sure that each process must be getting as equal data as possible so that each of them has almost equal amount of load to handle and thus , for that we first distributed the data equally among them , such that each of them is getting rows/numtasks number of rows and for the rest of the rows , which are left , i.e. rows%numtasks , since this value will less than the total number of processes , i.e. numtasks , thus we distribute 1 row to the first rows%numtasks processes. This helped us make sure that each of the processes is getting almost similar number of rows to work on.

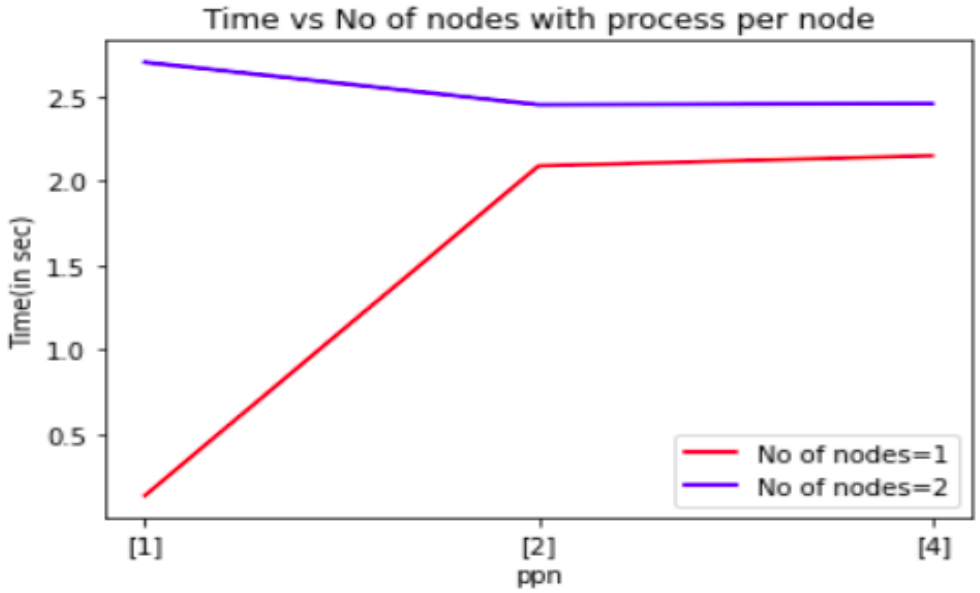
iii) If the number of rows is perfectly divisible by the number of processes , we simply used **scatter** collective call to send the part of the array to each process.
Else , we first used **scatter** collective call to tell each process what amount of data it would be receiving and then used **scatterv** collective call to send the different parts of the array to each process. (Vector collective call was required since here equal amount of data couldn't be shared among all the processes).

iv) Now , every data will call the function **min_of_each_col** to calculate the yearly minimum value of their respective arrays.

v) Next , we used **reduce** collective call to collectively get the final array of size = number of columns, which will contain the minimum value with respect to each of the columns.

vi) At last , we also calculated the global minima from this array and the maximum time taken and exported this result into the **output.txt** file.

The following is the plot which shows the time taken with respect to the various different configurations of nodes and cores when we performed 10 iterations for our code ::



The x-axis denotes the no of processes per node and the y-axis represents the time

taken by the respective set of processes. The blue line represents number of nodes=1 and the red line represents number of nodes=2

From the graph we imply that the time taken by node=1 and ppn=1 is lesser than that taken by ppn=2 and ppn=4 with node=1 as for the case of node=1 and ppn=1 we know that finding minimum of each column and then finding minimum of each column's minima result is very small since no amount of communication is happening here . So for ppn=2,4 the communication part is required to distribute the data among other processes is expensive than the computation benefits we can get over distributed data.Now for node=1, even though we are communicating less data among processes in ppn=4 than ppn =2, still ppn=2 performing slightly better than ppn=4. As computation benefit is very small and we have more intranode latency time for ppn=4 than ppn=2 which might be the reason for slight degradation in performance of ppn=4.

If we compare node=1 and node=2 for ppn=1,2,4 node=1 is performing better in each case. The reason for this is that in node=2 we will require internode communication to distribute the data which is very expensive and that degraded the performance.For node=2, ppn=1,2,4 we will have internode communication in each case and we have slight communication overhead for ppn=2,4 than ppn=1. Still we are getting slightly better performance for ppn=4 than ppn=2 , which in turn is having better performance than ppn=1 as benefits of doing computation on distributed data might overcome the slight communication overhead.

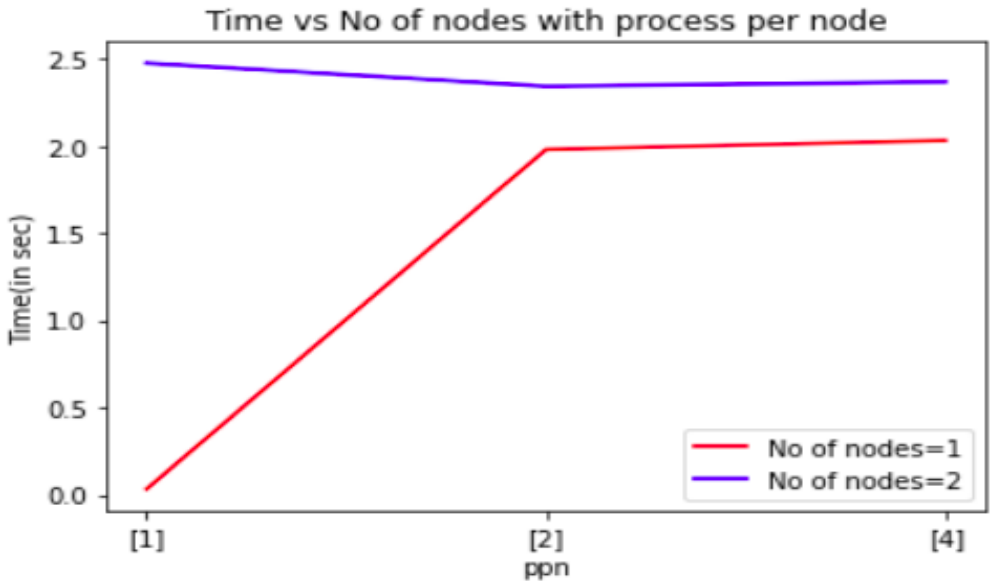
2nd Approach::

We also tried a 2nd approach , in which we basically store the data with rows and columns swapped i.e. as in the earlier case , if the number of rows = R and number of columns = C , then the matrix in which we are storing the data was $R \times C$, so in this case the matrix was $C \times R$. **The reason for storing the array in such a manner is because we are trying to write a cache-friendly code for each of the processes. So now, each process will be accessing the array to find the minima of a particular year by accessing the array in a row-major order instead of the column-major order we were accessing before.** And hence , now in order to distribute our data , we applied the same technique as above by first supplying equal number of rows of the matrix to each of the process and then for C%numtasks number of rows , it is distributed one by one to each of the different process. As above , this was done to ensure that almost equal amount of data is handled by each process.

After that , similar to the above case we used **scatter/scatterv** to distribute the data. Now , each process will have the required data . According to the problem statement , we are required to find the minimum of each column of the csv which is actually stored row-wise here. So , each of the process will call the function min_of_each_row and hence each of the processes will be storing the minimum value of some of the columns.

Now , at last , we can see that each of the process will be having some portion of the minimum values of the columns of the csv and hence, in order to get the total set of minimum value from each of the column, we need to somehow combine these , collecting them from each of the processes. This was done using the **Gather** collective call. After which we will have the final array consisting of minimum value of all the columns.

The following is the plot which shows the time taken with respect to the various different configurations of nodes and cores when we performed 10 iterations for our code ::



The x-axis denotes the no of processes per node and the y-axis represents the time taken by the respective set of processes. The blue line represents number of nodes=1 and the red line represents number of nodes=2

The graph here shows almost the similar results with respect to the previous approach with slight improvement. This is because the computation time incurred by each process would be lesser due to the cache-friendly code.

Why this approach was although better but wasn't used ::
If the number of columns and rows is not perfectly divisible by the total number of processes , then in that case the data distribution strategy among the processes in **2nd approach** would not be as good as what it would be in **1st approach**. So , in that case , some of the processes would get large amount of data for computation. We are not sure when the benefit of cache-friendly code gets overcome by the extra-computation + extra-communication(to transfer this extra data) cost it needs to do on a process. Hence ,we went further with the **1st approach** only.