# CS633 - Parallel Computing
# Assignment 3

**Shivam Kumar**
170669

In the given assignment, we were given temperature readings of several years for several thousand stations and we had to find:

- year-wise minimum temperature across all the stations for each year
- global minimum across all stations and all years

Sequential version of the solution is pretty simple: For each year, iterate over all the stations and find minimum temperature yearwise. Finally, minimum of the yearwise minimum can be calculated to get the global minimum.

However, we can make use of MPI algorithms to parallelize the computation to get speed up. We tried to do the same by dividing the working data among parallel processes, computing local minimum at each of the processes and then accumulating them to get the global minimum. Our approach and results are described in detail in the following sections.

## 1 Designing Parallel Computation

For designing parallel algorithm of minimum temperature computation, the first step required was to divide the data among the processes. Here, for the given dataset, it was stated that “there are several thousand stations", which comprise the rows in the dataset. On the other hand, the number of years, which comprise the columns, was said to be less than 100 always. In other words, the number of rows $\gg$ the number of columns. So, for better performance benefit, we divided the data along rows. Let N be the total number of rows i.e. stations and P be the number of processes, then each process was given data for N/P stations for all given years.

After each process got it’s working data, they independently computed local minimum temperature yearwise across all stations given to them. This is the part where execution happened in parallel across all processes.

After this, local minimum across all processes was reduced to get the global minimum yearwise. Once we have the yearwise minimum, we can sequentially compute the global minimum across all years. Note the fact that it’s not preferable to introduce parallelization in computing global minimum from yearwise minimum, since the maximum number of years in given dataset is quite small and so the overheads of parallelization will outweigh the benefits.

### 1.1 Code Description

The step by step description of the implementation of the above algorithm is as follows:

1. Input csv file is read, parsed and then actual temperature data yearwise for all stations is stored in dynamically allocated array *data* at the root process. Here, since the number of years and stations wasn’t known before, we first calculated the number of rows and columns by parsing the data. Thereafter in second iteration, actual data was stored in the *data* array. Note the fact that this step is performed for only root process.

2. If N is the number of stations, Y is the number of years and P is the number of processes, then size of every process working data would be (N/P)*Y floats.
3. MPI_Bcast was used to broadcast the number of rows and columns to all processes. This was required to dynamically allocate the *recvdata* array for locally storing the working data.
4. MPI_Scatter was used then to distribute the data among processes. This is the most time-consuming portion of the code, since data communication takes significant amount of time.
5. Local minimum yearwise is calculated in all processes by simply iterating over their own working data
6. MPI_Reduce call was used to reduce the computed local minimum across all processes and the yearwise global minimum was stored at root process.
7. If number of stations is not an integral multiple of the total number of processes i.e. N%P != 0, then there will be few rows left in the original data. Root process iterates over these rows to get the final global yearwise minimum.
8. Iterating over the yearwise minimum, the overall global minimum is computed at the root process.

In the implemented code, process with rank 0 was taken to be the root process. However, any process can be taken to be the root here.

## 2 Plots

In the assignment, we were asked to time all events after file reading stage. This comprised of data distribution, parallel minimum computation and then final reduction. To reduce the effect of noise in the time, the whole process(except file reading) was repeated 10 times and then average time across 10 runs were taken as the final time.

As per the given configurations, the code was executed on two values of number of nodes: 1 and 2, where number of nodes means number of systems on which the code was running. For each value of number of nodes, process per node was given three values: 1,2 and 4. So the code was executed for 6 values of the tuple (num_nodes,ppn): [(1,1),(1,2),(1,4),(2,1),(2,2),(2,4)]. The averaged value of execution time for each tuple(averaged over 10 runs) is shown in the plot.

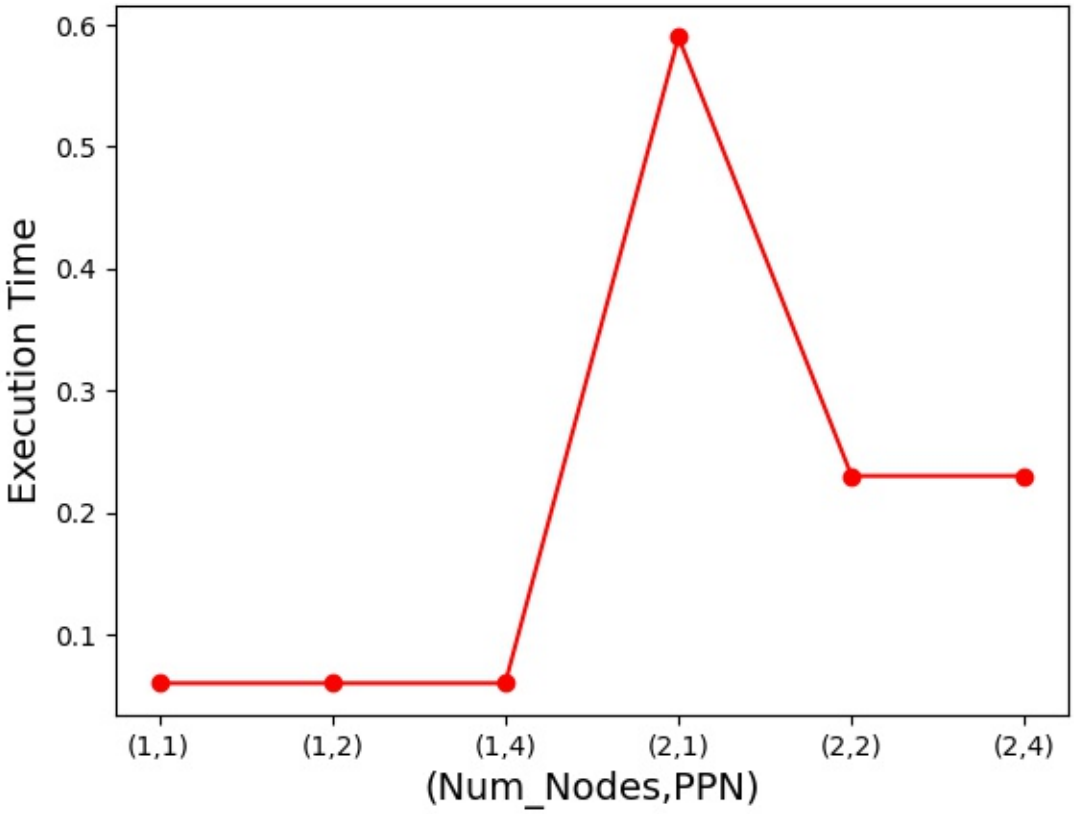


Figure 1: Plot for Parallel Computation

### 2.1 Observations

From the plots, it can be seen that there is no linear speedup observed. For process counts with number of nodes 1, the execution time is very much constant even after increasing process per node(ppn). This is because of the fact that data distribution adds overhead to the parallel algorithm and parallel computation tends to add speedup over sequential version. With num_nodes set as 1, on increasing the number of processes, the overhead of distribution and speedup due to parallelization seem to balance each other , thereby having very much constant time.

However, as soon as the number of nodes is set to 2, there is a big jump in the execution time. This is because the data transfer overhead is quite high when the transfer occurs between processes residing on different nodes. This distribution overhead completely outweighs the speedup obtained due to parallel portion and so the execution time increases significantly.

However, for num_nodes set to 2, when ppn is increased from 1 to 2, there is significant relative improvement. This is because of the sppedup obtained due to parallel portion. Note the fact that still the performance is poor with respect to sequential case i.e. (1,1) tuple, which again can be attributed to data transfer cost. Similarly when ppn is changed from 2 to 4, there is very slight improvement.

Overall it can be concluded that the actual speedup obtained is a result of the balancing between two opposing factors: data distribution(and reduction) overhead and benefit to parallel computation. Whichever is significant becomes the deciding factor in speedup.

## 3 Submission

The directory named Assignment3 is uploaded on git, which contains the source file (src.c), Makefile, plot script(plot.py) and plots. For ease of execution, a job script is also provided, which can be executed as follows:

```
$ bash run.sh
```

The job script compiles the code, executes the complete code on the given configurations and then executes the plot script for plots. Every single execution of src.c writes it's output in the file named as *output.txt*. However as mandated in the assignment, it is overwritten in every run. So to generate the plot, the time of every run is collected in a separate file named as *time.txt*.

The plot script plot.py uses the file *time.txt* to generate the plot for the given configurations.

### 3.1 Experimental Setup

- Install the matplotlib python library by the command (if not already present):

```
pip3 install matplotlib
```