

Introduction:

The objective of this experiment is to sort a structure by using multiple processes using MPI. We are using OpenMPI with C++. OpenMpi is one of the oldest and standard libraries for parallel processing. It is supported by various programming languages like C, C++, Java and Python (PyMPI). I have tried the python version of this library and found it very useful for beginners (Although it is not giving control over memory management).

Implementation:

We have implemented library sort.h with functions for merge sort, quick sort, radix sort, and heap sort. Each process with its own data calls the sort function and selects the numbers which need to be kept in memory. In this experiment, we are not using any additional memory and taken minimum communication with other processes which has a great impact on speed and scalability.

After calling the sort function with reference to data in it, each process calculate the index and offset of the data element to be kept in repeated manner. As there is an exact similar copy of data for each process every other process could easily guess the values of other process.

Results:

According to my program's output I can say that my program is completing all the basic requirement, very efficient in speed and also scaled very well. I have summarized the results below.

Scalability: Able to run it with 10^{10} size of data without any issue. I hope it will scale more than that.

Payload : I verified that the payload is intact after sorting and it also does not change among processes.

Sorting: Used merge sort, quick sort, radix sort and heap sort and all of them are working fine.

Experiment:

Create executable file using:

```
mpiCC -c -o sort.o sort.cpp
ar -rsclibsort.a sort.o
```

Run the file using:

```
mpirun -n 64 a.out
```

I have checked using a number of the process as 4,5,10,8,16,32,64 with test the data size of 10^3 to 10^{10} and it is working fine without taking much time.

Scalability:

I have checked this code up to 10^{10} data size and it is executing smoothly. Also, I tried to minimize allocating any extra memory and extra communication between processes so we can say that this code is very efficient in terms of speed and scale.