Introduction:

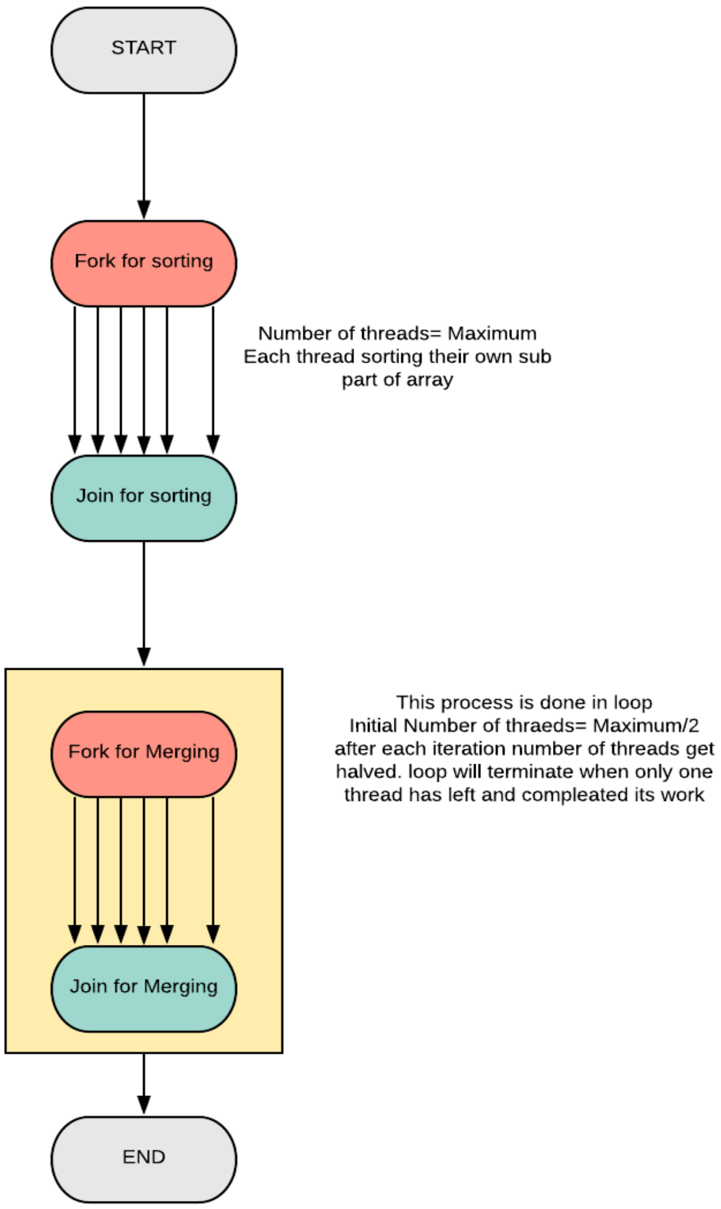
The objective of this experiment is to sort a structure by using multiple threads using OPENMP. We are using OpenMP with C++. OPENMP isone of the oldest and standard libraries for parallel processing. It is supported by various programming languages like C, C++, Java and Python.

Implementation:

We have implemented library sort.h with functions for merge sort, quick sort, radix sort, and heap sort. In this experiment, we are not using any additional memory and taken minimum communication with other thraeds 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.

A quick breakdown is given below in picture.


START
Fork for sorting
Number of threads= Maximum
Each thread sorting their own sub part of array
Join for sorting
Fork for Merging
This process is done in loop
Initial Number of thraeds= Maximum/2
after each iteration number of threads get halved. loop will terminate when only one thread has left and compleated its work
Join for Merging
END

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:

```
g++ -fPIC -c -Wall sort.cpp
gcc -shared -o libpsort.so sort.o
```

Run the file using:

```
bash test.sh //this will give an idea about speedup and thraed
```

```
I have checked using a number of the threads as 1 to 10, 13, 19, 20, 23 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 so we can say that this code is very efficient in terms of speed and scale.