PROJECT 2

Threading, or not?

CMPE 322

In this project, I have three '.c ' files. They are named as 'project21.c' , 'project22.c', 'project23.c'. They all compiled and created their executable objects with the command 'make'. In all three files, I sorted the random integer list with quicksort in advance. Same functions in different files are almost identical except for their return values. I used a global 'n' variable for given input N and a global integer pointer for the random integer array.

The 'project21.c' is the file which I used only one process. In the main, I get the return values of the functions and write them to the output1.txt file.

The "project22.c" is the file for which I used 10 threads. In this part of the project , I first attempted to implement a single thread function with a parameter named "arg" which takes the threadNumber and according to that threadNumber, it calls an operation function . After that, with a for loop, I created pthreads, and again with a for loop, I joined them. But after a few runs, I see that because of the if-else statements in the thread function and the for-loop to create pthreads with it, there is a lot of overhead, which makes the 10 threads stay below their potential. Then I decided to use it straight forward without a for loop and a thread function. I created 10 pthreads for 10 operation functions and joined them. I used global variables for every operation function, and changed their values with the threads. Finally as a lst step, I wrote the values of the global variables to the output2.txt file.

The 'project23.c' is the file which I used 5 threads similar to the 'project22'. Also in a similar way, I created global variables and assign their values with threads. The only difference between the two implementations is that I created five pthreads with five functions which call the paired operation functions.

In conclusion, I tested these three files and, what I observed was that the 'project21.c' which uses only one process was the slowest in general. Between 10 threads and 5 threads, the result was ambigious. After the first compilation and execution, 10 threads were usually faster compared to 5 threads. But after a few runs, the result turned out to be different in some cases. The reason why the result turned out this way was because the operations were not that complex and thread creating is not free. That is why it was sometimes slower to create five more threads than using just five.