Lab 3
Concurrent Programming

Name : Suraj B Thite

Comparison between the ease of parallelization between pthreads and OpenMP.

The OpenMP simplifies parallelizing code, debugging, and tuning leading to fast code development as compared to the pthread. It avoids the programmer from rewriting the tricky thread synchronizations thus more perfectly tuned for speed in compute bound applications. OpenMP facilitates the development of thread-safe data structures. The OpenMP runtime scheduling mechanisms are highly tuned and save the overhead of writing own scheduling mechanism by assigning priorities to thread in execution. The pthread code turns out to be complex as compared to the OpenMP version of it. The directives can be added incrementally leading to gradual parallelization with some part of code run serially. Thus, the code is easier to understand, easy to maintain and parallelize the code as compared to pthread version.

Parallelization Strategy :

The input data is read from the file passed as argument and stored in a global array define in the heap. The #pragma omp parallel is used to parallelize the merge sort algorithm to a maximum of 10 threads in execution. The total number of available cores where threads can be executed without preemption can be determined using omp_get_num_threads() API. The #pragma master code then determines the length of data set to be divided into threads in execution using task structure. #pragma omp for is used to parallelize the for loop which parallelly executes mergesort() function with their respective arguments. The master thread then does the final merge. The parallel algorithm used is similar to the one used in the fork join method but the #pragma omp directives are implemented instead of pthread APIs. The #pragma Barrier is used to implement barrier for the synchronization between threads. After completion of the final merge the data is written to the file and the binary exits.

Code Organization :

The below tree describes the brief organization of the code for the Lab0 in SRC directory.

Lab3

|----main.c - Main Application File consisting sequential flow of operations.

- main()
- file_to_array()
- array_to_file()

|----sorts.c - Source File consisting of functions for Merge sort and Quick sort on input data.

- mergesort()
- merge ()

|----Makefile - Make file to build the project and application binary mysort.

Files Description:

1. **main.c :** The main application source file which consist of sequential flow of instruction to parse input from the user, fetch the data from the file, sort the data depending upon the argument passed and writeback to the output file. In case of –name argument is passed, the Author name i.e. Suraj Bajrang Thite is printed and the rest of the process is skipped. The getopt_long() function is used to parse the arguments from the user. The maximum number of threads is kept as 10 which can be changed using MAX_NUM_THREADS Macro defined.
2. **sorts.c :** The source application for merge and quick sort algorithms along with associated member functions. The function takes the array as input arguments and sorts it using merge sort algorithm
3. **Makefile :** The make file to build the source binary (mysort) using gnu C compiler running on Linux Platform.

Compilation Instructions :

Navigate to the working directory of the project and open a terminal.

Run make to compile the project and generate mysort binary file.

Run make clean to clean the application binary from the project folder.

Execution Instructions :

Usage :

./mysort [--name] [sourcefile.txt] [-o outputfile.txt]

If -- name is passed as argument, the program is terminated.

Failure to open the input file or write sorted data to the output file or incorrect no of arguments passed will stop the further execution printing appropriate messages on terminal.

Extant Bugs :

No bugs in the code as such, the project compiles successfully and passes the test case as mentioned in the assignment requirements.