CS3006 Parallel and Distributed Computing
Assignment 1-Pthreads

Submission deadline: 13th February 2024, 11:59PM

Instructions:

- For your assignment, make a code file and pdf file with Roll Number. For e.g. make ROLL-NUM_SECTION_A1.cpp (23i-0001_A_A1.c) and so on. Each file that you submit must contain your name, student-id, and assignment # on top of the file in comments.
- Combine all your work in one folder. The folder must contain only code and pdf file (no binaries, no exe files etc.).
- Rename the folder as ROLL-NUM_SECTION (e.g. 23i-0001_A) and compress the folder as a zip file. (e.g. 23i-0001_A.zip). Do not submit .rar file.
- All the submissions will be done on Google classroom within the deadline. Submissions other than Google classroom (e.g. email etc.) will not be accepted.
- The student is solely responsible to check the final zip files for issues like corrupt files, viruses in the file, mistakenly exe sent. If we cannot download the file from Google classroom due to any reason, it will lead to zero marks in the assignment.
- Displayed output should be well mannered and well presented. Use appropriate comments and indentation in your source code.
- **Be prepared for viva or anything else after the submission of assignment.**
- If there is a syntax error in code, zero marks will be awarded in that part of assignment.
- Understanding the assignment is also part of assignment.
- **Zero marks will be awarded to the students involved in plagiarism. (Copying from the internet is the easiest way to get caught).**
- **Late Submission** will **not** be entertained and **No retake** request will be accepted as per course policy.

Tip: For timely completion of the assignment, start as early as possible.
Note: Follow the given instruction to the letter, failing to do so will result in a zero.

This assignment consists of 2 parts, (i) serial solution and (ii) a parallel solution.

1. **Serial Version:** Students will implement a program to

   i. read a list of numbers from file and store them in an array,
   ii. insert the individual numbers into a linked list,
   iii. and sort the linked list using merge sort

```
      Serial version functions
void readRollNumbers(FILE* inputFile, int* Numbers, int num);
void addRollNumbersToList(Node** head, int* Numbers, int num);
Node* mergeSort(Node* head);
```

2. **Parallel Version with CPU Affinity:**
   - Create a pthread function to add roll numbers into the linked list concurrently. Each thread will handle a subset of the roll numbers.
   - Implement a pthread function to parallelize the merge sort algorithm on the linked list. Threads will work on different portions of the list during the merge sort process.
   - incorporate `pthread_setaffinity_np` for setting CPU affinity on a per-thread basis. Each thread in the parallel version will be mapped to a specific CPU core using `pthread_setaffinity_np`
     i. plan and implement the mapping of threads to specific CPU cores using the CPU affinity concept, exploiting `sched_setaffinity` function calls.
     ii. Utilize `pthread_setaffinity_np` to set CPU affinity for each thread, specifying the CPU cores they should be bound to. Details related to affinity available here (https://man7.org/linux/man-pages/man3/pthread_setaffinity_np.3.html).

```
#include <pthread.h>

// Parallel version functions with pthread_setaffinity_np
void* addRollNumbersToListParallel(void* arg);
void* mergeSortParallel(void* arg);
void setAffinity(pthread_t thread, int coreId);
```

Additional Requirements

1. **Thread Mapping Plan:**
   - Before implementing the parallel version, students should analyze the system's architecture and come up with a plan for mapping threads to specific CPU cores.
   - Consider factors such as the number of available cores, workload distribution, and potential resource contention.
   - Document the chosen mapping strategy and justify why it is suitable for the given scenario.

2. **Testing and Analysis:**
   - Measure the execution time of both serial and parallel versions for varying input sizes.
   - Analyze and compare the performance of the serial and parallel versions with and without CPU affinity.
   - Discuss any speedup, and possible bottlenecks introduced through CPU affinity. General speedup formulas are provided below you can use them as per your requirements.

$$Speedup_{actual} = \frac{T_1}{T_n}$$

where, $\boldsymbol{n}$ is the number of cores
$\boldsymbol{T_1}$ is the execution time on one core
$\boldsymbol{T_n}$ is the execution time on $\mathbf{n}$ cores

$$Speedup_{Amda\quad 's\ Law} = \frac{T_1}{T_n} = \frac{1}{\frac{f_p}{n} + f_s}$$

$\boldsymbol{f_p}$ *is the paralellizble fraction of the program*
$\boldsymbol{f_s} = 1 - \boldsymbol{f_p}$ *is the sequential fraction of the program*

Note: Keep in mind that Amdahl's Law is a theoretical model. The actual speedup measures how much faster the parallel execution is compared to the sequential (singleprocessor) execution.

3. **Documentation and Discussion:**
   - Document the code thoroughly, including the thread mapping plan (parallelization strategy) and the implementation of CPU affinity.
   - Discuss the impact of CPU affinity on the parallel version's performance, considering factors such as cache locality and reduced context switching.

This problem will help you learn hardware-level optimizations by manually controlling thread-to-core mappings, providing valuable insights into the intricacies of parallel programming and system-level considerations.