TDT4200 Problem Set 4

Pthreads and OpenMP

Maren Wessel-Berg and Claudi Lleyda Moltó

Deadline: October 10, 2023 by 22:00
Evaluation: Pass/Fail

- **This assignment has two parts.** In the first part, you will solve programming tasks. These tasks are described in the Tasks section. In the second part, you will answer questions about your implementation and/or from the curriculum. These questions are stated in the Questions section.
- **This assignment is mandatory and will be evaluated as pass/fail.**
- **This assignment must be done individually and without help from anyone but the TDT4200 staff.** All sources found on the internet or elsewhere must be referenced. We encourage that you post clarification questions on Piazza so all can benefit. However, make sure you do not post full or partial solutions on Piazza.
- **The assignment requirements are explained in the Evaluation section.**
- **How and what to deliver is explained in the Delivery section.** Do not deliver any other files than those specified. Code should not use external dependencies aside from the ones already included in the handout code.

Finite difference approximation of the 2D heat equation using Pthreads and OpenMP

In this exercise you will take a sequential implementation of the Finite Difference Method (FDM) for solving the 2D heat equation and write parallel versions using Pthreads and OpenMP. The sequential implementation is provided. Skeletons for your Pthreads and OpenMP implementations are provided in `heat_pthreads.c` and `heat_omp.c`, respectively. Information on how to install the dependencies and run the program is provided in the `README.md` of the handout code.

Although we provide a Makefile in the handout code, we encourage you to inspect the Makefile to see how to compile and run the program.

Tasks

We recommend that you ensure the programs compile after completing each task even though the programs might not run correctly between tasks.

Sequential Pthreads

Start by implementing a sequential implementation that runs in a single pthread.

1. Extract the part of the sequential code that you want to parallelize into a separate function. Make sure that the function signature matches pthreads' expectations (`void* func(void*);`).
2. Use `pthreads_create` to spawn a thread for the function.
3. Use `pthreads_join` to join the thread.
4. Run the program and make sure it produces the correct result.

Parallel Pthreads

Parallelize the Pthreads implementation

1. Provide some data to the function, like the ID of the thread.
2. Create an array of pthreads handles.
3. Spawn several threads and give them unique IDs.
4. Join all the threads.

5. Optimize the function step by step by dividing the domain for each thread. You may either divide the domain into horizontal/vertical slices of the domain (the easier alternative) or you can divide the domain into a grid (the harder alternative), both are accepted solutions. Make sure that your solution is correct for one part before moving forward to the next part.

   Note You are allowed to assume that the grid size is divisible by the number of threads.

   Note The boundary condition code does not need to be parallelized (but you may do it if you want to).

   Note There is a dependency between the threads where a thread needs to use values from the other threads' sub-grids to perform time step computations in their own sub-grid. Therefore, the threads need to be synchronized to avoid race conditions. Add synchronization primitives like barriers to avoid race conditions in your code.

6. How much speedup did you get for this parallelized Pthreads solution?

OpenMP

1. Parallelize the sequential code by adding OpenMP pragmas to the for-loops in the code. Check that you still get the correct answer for each time you add a new pragma to a for-loop.

2. How much speedup did you get with this parallelized OpenMP solution?

Questions

- What is a critical section?
- How can a critical section be protected from race conditions?
- What is oversubscription?
- Suppose that you have a set of n pthreads, each with a local integer that represents each thread's individual index from 0 through $n - 1$, and a print statement at the end of its function body. How would you implement a method to ensure that the output of the print statements appears in the order of the thread indices? A pseudo-code description is sufficient to answer this question.

Evaluation

The code should run error-free and produce correct output. Your solution can be compared for correctness with the correctness script in the handout code.

Answer the questions with a couple of sentences. The purpose of the questions is for you to reflect on the work you have done and the curriculum.

Delivery

Deliver the files `heat_pthreads.c` and `heat_omp.c` in BlackBoard. Answers to the questions should also be delivered in BlackBoard in a separate PDF file. Do **NOT** upload a ZIP file.