PCA OpenMP Homework

David Langerman - Evan Gretok

February 2019

1 Homework (100 points)

Please submit all code files in a .tar, .7z, or .zip file uploaded to blackboard. Your short answer questions should be in a .pdf format.

1. (10) Explain shared-memory multiprocessing using OpenMP. Try to use non engineering terms; i.e. if you had to explain it to a non engineer, how would you go about it? Please consider some of the following matters: How might we visualize fork, join, and division of work? As all processes execute the same code, what methods can we use to assign tasks to one process? What is a race condition and how can we avoid them? What are the reasons, benefits, and challenges for shared and private variables between threads? What is scheduling and how does it apply?
2. (5) What are the three built-in OpenMP "get" functions that are used to identify and manage threads in almost every OpenMP program? Explain what each one does in a few words.
3. (5) What are five OpenMP parallel pragma parameters used to prepare variables or define behavior within the parallel region? Explain what each one does in a few words.
4. (20) Answer the following questions in one or two sentences. (5 points each)
   (a) What is one method to ensure serial execution of a certain segment of a program?
   (b) Suggest one example case where we would use block division instead of round-robin (cyclic) to divide a problem for parallelization.
   (c) Explain the difference between domain (data) and functional decomposition.
   (d) Detail one case where you would use critical regions and another where you would use atomic statements, contrasting the difference.
5. (20) Write a parallel matrix multiplication implementation using only OpenMP to execute on multiple processor cores. Your program must have 3 constants: M, N, and P. The program itself will multiply an M by N matrix with an N by P matrix (the output will be of size M x P). Run your program on the CRC and record speedup and parallel efficiency for the following configurations for 1, 2, 4, 8, and 16 cores. **Note**: Your code *must* handle an arbitrary number of execution nodes, so make sure to handle edge cases.
   - N=10, M=10, P=10
   - N=100, M=100, P=100
   - N=1000, M=1000, P=1000

   Modern supercomputer clusters are equipped with not only many nodes, but also multi- or many-core processors within each node. Software development for the latest high-performance computing must reflect this. Extend your previously developed MPI-parallelized matrix multiplication code with OpenMP for parallelization on multiple cores within the node. Matrices shall be generated on a single home node, then distributed. Run this program on the CRC and record speedup and parallel efficiency for the above configurations for 1, 2, and 4 nodes with 1, 2, and 4 cores. How do the performances of these trials compare with MPI-only and OpenMP-only parallelization?

6. (20) Parallelizing existing applications is an essential skill for increasing speed and efficiency as well as assessing parallelizability before employing further acceleration via GPUs (with CUDA) or FPGAs (with HLS or HDL). Scheduling can be applied to tune parallel performance for the most ideal fit to the application. Inspect the provided example program for Mandelbrot set fractal generation and parallelize it using OpenMP. Apply each of scheduling methodologies noted below. Run the program on the CRC using eight cores. The requirements for this program are below:

   - omp_get_wtime() shall be used to acquire timing information.
   - Plot your data in Excel/Libre Office and attach as a scatter plot with execution time on the Y-axis and scheduling type and chunk size on the X-axis.
   - You must gather enough data to accurately represent execution times using each of the following five methods of OpenMP parallelization:
     - Static, Typical
     - Static, Chunk Sizes 10, 100, and one of your choice.
     - Dynamic, Typical
     - Dynamic, Chunk Sizes 10, 100, and one of your choice.
     - Guided
   - It is strongly recommended that you gather at least ten runs of each configuration and average the timings of those ten runs for accuracy.

   Compare the results of the scheduling methods tested and summarize your findings.

7. (20) As a final challenge to test your capabilities in OpenMP, you are to implement and parallelize Conway's Game of Life. Conway's Game of Life is a cellular automaton played on a two-dimensional grid and consisting of cells that are either "alive" or "dead." Each cell interacts with the eight cells that surround it. The game is governed by three simple rules:

   - A cell with fewer than two living neighbor cells dies (underpopulation).
   - A cell with two or three living neighbor cells remains alive.
   - A cell with more than three living neighbor cells dies (overpopulation).
   - A cell with exactly three living neighbor cells is "born" (reproduction).

   Construct your own implementation and parallelize it with OpenMP. Compare the timing to process several iterations serially versus doing so with 1, 2, and 4 cores on the CRC. The requirements for this program are below:

   - Your program shall take a grid of characters in a text file as an input, 'o' means alive, 'x' means dead. Each cell is separated by a space, and the input file shall use UNIX line endings (LF). You may assume that the file is correctly formatted.
   - Your program must compile with gcc.
   - Your program must run given the following command:
     ./life <cores> <input file name> < number of iterations> <print frequency>
   - Your program must handle non-square grids.
   - Print your final output to a file using the same format as the input file.
   - Manage memory carefully. We might run your program for a very large number of iterations, so any memory leak will likely manifest as a crash.

   In your submission, please include a short summary of your parallelization method and an explanation of your results including runtimes, speedup, and parallel efficiencies for various grid sizes. Pay particular attention to how the shape of the grid and the initialization of the grid affects the runtime of this program.