COMS3008A Course Assignment (1)

Hand-out date: Thurday, Apr 18, 16:00, 2024
Due date: Monday, Apr 29, 16:00, 2024

Contents

1 Introduction

1. You are expected to work individually on this assignment.
2. Submission of code without report or report without code will result in 0 mark.
3. In your report, proper citations and references must be given where necessary.
4. Using Latex to write the report is required. A template tex file is provided.
5. Start early and plan your time effectively. Meet the deadline to avoid late submission penalties (20% to 40% depending on the time for overdue).

2 Problems

1. **Problem 1: Generating Julia Set Fractals in parallel** Julia sets are certain fractal sets in the complex plane that arise from the dynamics of complex polynomials. The Julia set is the boundary of a certain class of functions over complex numbers. For almost all values of the function's parameters, this boundary forms a fractal. The calculations involved in generating such a set are quite simple. At its heart, the Julia set evaluates a simple iterative equation for points in the complex plane. A point is not in the set if the process of iterating the equation diverges for that point. That is, if the sequence of values produced by iterating the equation grows toward infinity, a point is considered

outside the set. Conversely, if the values taken by the equation remain bounded, the point is in the set.

Computationally, the iterative equation to be evaluated is rather simple, which is $z_{n+1} = z_n^2 + c$, where c is a complex number (constant) that gives a specific set of fractal. Julia sets are generated by initializing a complex number $z = x + iy$ where $i^2 = -1$, x and y are image pixel coordinates scaled to the range of about -2 to 2. Performing this calculation for a whole 2D square grid, `M-by-M`, of pixels yields a fractal image (see Figure 1 for an example).

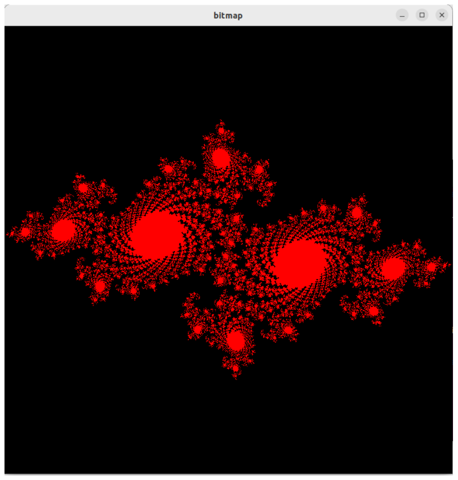

Figure 1: A fractal image example

Program `fractal.cpp` gives a serial implementation of computing the Julia set. In this baseline code, the maximum number of iterations is set to 300, and the threshold to determine whether or not a point is in the Julia set is set to 1000. That is, if the value of the iterative function at a certain point does not exceed the threshold (set to 1000 in the code), then the point remains in the Julia set and is assigned a certain color (red in the code). On the other hand, if the value exceeds the threshold within the maximum number of iterations (set to 300 in the code), it is considered to be diverging and hence not included in the Julia set (i.e., no color or black color is assigned in the code). In the resulting image, red color pixels belong to the particular Julia set, and the pixels with black color do not. In order to display the image in a terminal, you must first uncomment the line (No 18) that defines the macro named `DISPLAY` before compilation. **Note** that to compile and run this code, it is best to do so from a Linux terminal, where OpenGL/GLUT libraries are already installed in your Linux. If this is not the case, the code may not compile and run.

Write different parallel functions, respectively, for compute-intensive function `serial_fractal` (see program `fractal.cpp`) using OpenMP in the following way.

(a) 1D rowwise parallel. In this case, assuming there are T number of threads, then these threads process a group of T number of consecutive rows in parallel at a time, where Thread 0 is assigned the first row in the group, Thread 1 is assigned the second, and so on. The same process iterates until all the rows in the grid are processed. In this way, Thread 0 would process Row 0, Row T, Row 2T, and so on; Thread 1 would process Row 1, Row $T+1$, Row $2T+1$, and so on; and similarly for other threads in the team of T threads. [16%]

(b) 1D columnwise parallel. The process is similar to 1D rowwise parallel in Item 1a, however, instead of each thread processes one row at a time, it processes a column. [16%]

(c) 2D row-block parallel. In this case, assuming there are T number of threads, M/T (integer division) consecutive rows are assigned to a thread. For example, if the grid is 7×7, and there are three threads, then the above block division could give Rows 0 and 1 to Thead 0, Rows 2 and 3 to Thread 1, and the remaining rows to Thread 2. [16%]

(d) 2D column-block parallel. This case is similar to 2D row-block parallel in Item 1c, however, instead of dividing the grid into blocks of consecutive rows, it divides the grid into blocks of consecutive columns and each thread is assigned one of the blocks. [16%]

(e) Using OpenMP `for` construct. [16%]

Note that for the first 4 implementations above, you would need to do the task decomposition for parallel threads 'manually', like we did in the example of computing the number π (Example 4.2.3).

In your implementation, consider the following different values for constant c.

(a) $c = -0.123 + 0.745i$

(b) $c = -0.7269 + 0.1889i$

(c) $c = -0.50 - 0.56i$

Hand-in:

(a) A brief report that [20%]

  i. gives some details on your implementations for each method;

  ii. shows the performance of different parallelizations in a 2D line plot, where horizontal axis represents number of threads, and the vertical axis represents speedup. The number of threads to be used ranges in {1 (equivalent to sequential), 2, 4, 6, 8, 10, 12, 14, 16};

  iii. and presents discussions on your results and findings. In particular analyse your results based on the following questions, and give answers to them accordingly.

    A. Is there a difference, in general, in the performances of 1D rowwise parallel and 1D columnwise parallel? If there is, how do you explain the difference?

    B. Is there a difference, in general, in the performances of 2D row-block parallel and 2D column-block parallel? If there is, how do you explain the difference?

    C. Find the two top performers, and analyse their similarity and difference. Further, in your understanding, why the two outperform the others?

    D. Give any ideas that could further improve the performance of the top two performer. If one of the two top performers is from Item 1e, i.e., using OpenMP `for` construct, then you only need to give your ideas of improvement on the other one.

(b) A folder that contains your source codes, `Makefile` for compilation, and runscript for running. A `ReadMe` file is optional. (During marking, we simply assume your code is compiled using the `Makefile`, and run using the `run.sh` you provide in the same folder. If these support files are not provided, you must give clear instructions in a `ReadMe` file on how to compile and run your code.) No submission of any support files such as `Makefile` will result in deduction of the total marks even if we manage to compile and run your code eventually.

3 Academic Honesty

In order to maintain the highest standards of academic honesty, students are reminded to refrain from any form of plagiarism or unauthorized collaboration. All code and report submitted must be the result of individual effort and understanding.