Programming Assignment 1

Due Date: Sun. 10/4, 11:59pm

Background

Adaptive Mesh Refinement (AMR) is a computing method by which natural phenomena are modeled. Typically, a grid of arbitrary resolution is divided into a number of "boxes," each of which contain some set of domain-specific values (DSVs). Each DSV is given some initial value corresponding to a start state, and then updated based upon some set of rules which include influencing effects from the values of neighboring boxes. (This is often referred to as a "stencil computation.") Updated values for all boxes are computed within each iteration of a controlling loop (called a “convergence loop”) based on the current set of DSV values. The newly computed set of values are then “committed,” meaning they replace the prior set of values at effectively the same time. These iterative computations are repeated until “convergence,” meaning that some threshold relating to a steady state is reached. This stop criteris is typically modeled by the DSVs all falling within some defined relative range, or that the DSVs enter a state of “changing slowly.”

It is important to note that, as stated above, the DSVs adhere to a “compute / commit” process. All current-iteration DSV updates are computed based upon the same (current) set of DSV values. Then, the DSVs are updated in bulk. This avoids the creation of ordering dependencies that would occurr if each DVS were updated individually as its new value was computed. As we will see later this semester, this independence of computational ordering is necessary to allow for parallel programs which achieve the same results (presumably, the correct ones) as their serial counterparts.

Problem Statement: A Simplified AMR Dissipation Problem

For our assignment, we will implement a simplified AMR problem. You will be given as input a file containing a description of a grid of arbitrarily-sized boxes, each containing a DSV for the starting “temperature” of the box. Your program will model the heat dissipation throughout the grid.

Input Data Format

The input files for testing your program will be in the following format (counts, ids and co-ordinates are integers, the DSV is a float and may appear with or without a decimal).

line 1:	<number of grid boxes> <num_grid_rows> <num_grid_cols>
line 2:	<current_box_id>
	//starting with 0
line 3:	<upper_left_y> <upper_left_x> <height> <width>
	//positions current box on underlying co-ordinate grid
line 4:	<num_top_neighbors> vector<top_neighbor_ids>
line 5:	<num_bottom_neighbors> vector<bottom_neighbor_ids>
line 6:	<num_left_neighbors> vector<left_neighbor_ids>
line 7:	<num_right_neighbors> vector<right_neighbor_ids>
line 8:	<box_dsv>
	//"temperature," be sure to store as a double-precision float
	repeat lines 2 - 8 for each subsequent box
	specify boxes sequentially in row major order
line last:	-1

While there is no “bad data” (data you would have to screen out) in the input files, your program should allow for an arbitrary number of tabs or spaces between entries, and the possibility of blank lines which should be ignored.

Dissipation Model

Your program will load a data file, reading it from standard input and setting the initial temperatures for each box as appropriate. Then, iteratively compute updated values for the temperature of each box to model the diffusion of "heat" through the boxes.

Your program should implement a heat dissipation model using approach described below. Your model will use two iniput parameters: AFFECT_RATE, which represents the weight of the influence from neighboring DSVs and determines the rate at which individual DSVs change; and EPSILON, which defines how closely in value all DSVs must be in order to be considered "converged."

Perform the following in each iteration of the convergence loop:

1. compute the weighted average adjacent temperature (WAAT) for each box, based upon the DSVs of the neighbors and their contact distance with the current box.
2. for each box, migrate the current box temperature toward its WAAT by adding or subtracting (as appropriate) AFFECT_RATE% of the difference between the WAAT and the current box temperature.
3. commit the results of this convergence loop iteration by replacing all current DSVs with the newly-computed DSVs.
4. check the stop condition, which is (max_DSV - min_DSV) < (EPSILON * max_DSV). That is, if the difference between the current maximally-valued DSV and the current minimally-valued DSV is less than EPSILON (which is specified as a fraction) times the maximally-valued DSV, then stop. Otherwise, continue with another loop iteration.

The remainder of this section provides additional detail on the above four steps.

1. compute the WAAT for each box, based upon the DSVs of the neighbors and their contact distance with the current box.

   Compute the average adjacent temperature for each "current box" as follows:

   - Assume the temperature outside of the grid is the same as the temperature for the current box.
   - Ignore diagonal neighbors.
   - Compute the sum of the temperature of each neighbor box times it's contact distance with the current box.
   - divide that sum by the perimeter of the current box yielding the average adjacent temperature of the current box.

   For example, consider this simple 3x3 grid, showing the initial temperatures of 9 1x1 boxes:

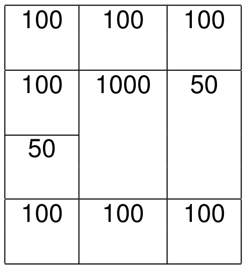
100	100	100
100	1000	100
100	100	100

corresponding box ids:

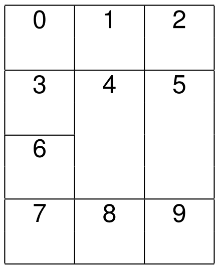
0	1	2
3	4	5
6	7	8

   - the weighted sum adjacent temperature for box 0 would be (100*4) = 400.
     (temp(box 1) + temp(box 3) + 2*temp(box 0))
   - the perimeter of box 0 is 4
   - the WAAT for box 0 would be 100.
   - the weighted sum adjacent temperature for box 4 would be (100*4) = 400.
     (temp(box 1) + temp(box 3) + temp(box 5) + temp(box 7))
   - the perimeter of box 4 is 4
   - the WAAT for box 4 would be 100.

Note that some boxes will have multiple neighbors in a given direction, and the temperature of each neighbor should be weighted by it's contact distance with the current box.



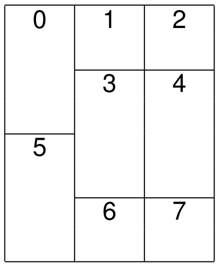
corresponding box ids:



- the weighted sum adjacent temperature for box 3 would be (100*1) + (1000*1) + (50 * 1) + (100 * 1) = 1250, (temp(box 0) * 1 unit contact + temp(box 4) * 1 unit contact + temp(box 6) * 1 unit contact + temp(box 3) * 1 unit border)
- the perimeter of box 3 is 4
- the WAAT for box 0 would be 312.5.
- note that box 4 has 5 neighbors: 1 to the top, 1 to the right, 1 to the bottom, and 2 to the left.
- the weighted sum adjacent temperature for box 4 would be (100*1) + (50 * 2) + (100*1) + (50 * 1) + (100 * 1) = 450.
- the perimeter of box 4 is 6
- the WAAT for box 4 would be 75.

Note that some boxes my overlap irregularly, as in the following example:

box ids:



- Although boxes 0, 3 and 5 all have height 2, the contact distance between boxes 0 and 3 is 1, as is also the case between boxes 3 and 5, as well as between boxes 0 and 5.

2. migrate the current box temperature toward the WAAT by adding or subtracting (as appropriate) AFFECT_RATE% of the difference between the WAAT and the current box temperature.
   - For example, if the current box temperature is 1000, the WAAT is 900, and AFFECT_RATE is 10%, then the new box temperature would be 1000 - (1000 - 900)*.1, or 990.
   - For example, if the current box temperature is 1000, the WAAT is 1100, and AFFECT_RATE is 10%, then the new box temperature would be 1000 + (1100 - 1000)*.1, or 1010.
   - AFFECT_RATE will be specified as a command-line parameter to your program.
3. Once updated DSV values for all boxes are computed, your program should commit these changes in preparation for the next iteration. Do not update DSVs individually as they are computed, this will lead to erroneous results and may impede your parallelization of the code.
4. Iterate the DSV update process until the difference between the maximum and minimum current box temperatures is no greater than EPSILON% of the highest current box temperature (this is called "convergence").
   - EPSILON will be specified as a command-line parameter to your program.

Program Requirements

Within this section you are provided several requirements including, but not limited to, an overall program structure (which is necessary for compatibility with future labs), language requirement, and also program build and submission requirements.

While this lab provides the freedom of expression for many aspects of the design of your solution (including your preference of data structures), treat all requirements in this section as firm. Do not deviate from the requirements of this section. Your work will be graded for strict conformity with the stated requirements.

For best performance and for easiest adaptation to the parallel APIs we will study, my suggestion is that you implement your program using the most basic and efficient data structures you can visualize. While having their own set of advantages, complex hierarchical data structures, such as objects, do not tend to perform as well and often must be de-constructed in order to attain desired performance.

- Your program should follow the following general form for scientific computing applications of this category:

  ```
  repeat  until  converged
         for  each  container
                 compute updated  domain  specific  values  (DSVs)
         commit  updated  DSVs
  ```

  In particular, be sure to have a convergence loop, and be sure to commit updated DSVs all at once (compute into temporaries, and then copy to primary data structure). This compute/commit organization will eliminate loop dependencies and allow for equivalent parallel programs.

- Programs are to be written in the C programming language in a version compatible with the Intel compiler ("icc") currently in use on OSC systems. Compile your programs on OSC systems with the icc compiler. Compile your program with optimizer level3 (-O3).

- Input data files must be read from standard input. See the link below for using Standard Input from within your C programs,
  `http://lessonsincoding.blogspot.com/2012/03/using-stdin-stdout-in-c-c.html`
  and the following links for setting up standard input and output from the shell command line
  `https://www.cs.bu.edu/teaching/c/file-io/intro/` , or,
  `https://www.tutorialspoint.com/cprogramming/c_input_output.htm`

  IMPORTANT: The use of Unix standard I/O (stdio) including the redirection of file paths from the command line is a strict requirement of this lab. Programs which do not implement this requirement will receive a score of 0 and your program will not be evaluated. Be sure that your program opens all input and output files in this way. Do not use open()/fopen() or other similar commands to open files from within your program.

  If you are unfamiliar with Unix standard I/O, including the automatic opening of files using redirectors on the command line, then you are responsible for acquiring this information prior to submitting your lab.

- To support tuning the run-time and comparison with parallel versions, your program must support two command-line parameters for AFFECT_RATE and EPSILON, using the standard argc and argv mechanism. Refer to the following links for an introduction to command line parameters and how to access them from within your C programs.
  `https://www.tutorialspoint.com/unix/unix-special-variables.htm`
  `https://www.tutorialspoint.com/cprogramming/c_command_line_arguments.htm`

- Modify your values for EPSILON and AFFECT_RATE as needed (from the command line), such that your application successfully converges for the "testgrid_400_12206" test data file, while running for between 3 and 6 minutes. This should cause your program to run long enough to measure the impact of threads in future labs while not running so long as to waste computational resources. Anything within this runtime range is perfectly fine. Note that, for many or most of you, this will mean trying to slow your program down.

- Print the number of convergence loop iterations performed, along with the last values for maximum and minimum DSV, on standard output.
- Provide a makefile with your program. Your program should build with the command **"make"** (with no arguments). (i.e. the make file should be named as **"makefile"** and reside within the same directory as your source code.)
- Use program file suffixes ".c" for program files and ".h" for (optional) user header files.
- Using the instrumentation tools described below, and testing with the largest test data file (testgrid_400_12206, described below), modify your values for AFFECT_RATE and/or EPSILON so that your serial program runs to convergence in between 4 and 6 minutes of elapsed clock time. This is necessary so that we can accurately measure the speed-up of the parallel versions of your program.

Input Files

This section, which contains a characterization of the test input files, is provided solely for your reference so that you might gauge the appropriateness of your results. You are not required to exactly match the convergence iterations, as it is expected there will be some difference due to your specific program and your implementation of the algorithm. However, should your results vary significantly from those below and you are using the required dissipation model, then you should suspect that you may have a bug in your program.

All files will be made available on stdlinux in directory /class/cse5441/amr_test_files.

Test grid input file information:
All run with AFFECT_RATE=.1 and EPSILON=.1

testgrid_1 - a simple 3x3 grid of unit-sized boxes may be useful for testing
testgrid_2 - converges in 245 iterations
testgrid_50_78 - data file with 78 variable overlap boxes on a 50x50 grid. Converges in 1,508 iterations
testgrid_50_201 - data file with 201 variable overlap boxes on a 50x50 grid. Converges in 2,286 iterations
testgrid_200_1166 - data file with 1,166 variable overlap boxes on a 200x200 grid. Converges in 14,461 iterations
testgrid_400_1636 - data file with 1,636 variable overlap boxes on a 400x400 grid. Converges in 22,283 iterations
testgrid_400_12206 - data file with 12,206 variable overlap boxes on a 400x400 grid. Converges in 75,269 iterations

Instrumentation

We will use the following methods to instrument the run-time of your initial serial program.

- Measure and report the run time of your convergence loop using the Unix time(2) and clock(2) system calls (the following work for both C and C++).
  - `http://www.cplusplus.com/reference/ctime/time/?kw=time`
  - `http://www.cplusplus.com/reference/ctime/clock/?kw=clock`
- Also, when executing your program, use the Unix time(1) utility to report elapsed clock, user and system times.

IMPORTANT The use of the common Unix instrumentation methods time(2), clock(2) and time(1), is a strict requirement of this lab. Programs which do not implement this requirement will receive a score of 0 for the performance measurement portion of the lab.

If you are unfamiliar with these instrumentation methods, then you are responsible for acquiring this information prior to submitting your lab. One or two short test programs is typically all that is required to understand these methods.

Program Output

You are not required to adhere to the following format precisely, but below is an example output for one specific test run containing all of the required elements for your assignment.

Notice that following the prompt is the invocation of time(1), followed by the program name, AFFECT_RATE and EPSILON parameters, the stdin file redirector, and lastly the input file name.

```
linux:jonejeff:   time ./amr_csr_serial 0.1 0.1 < /class/cse5441/testgrid_400_12206


*********************************************************************
dissipation converged in 75269 iterations,
    with max DSV = 0.0866714 and min DSV = 0.0780043
    affect rate  = 0.1;          epsilon = 0.1

elapsed convergence loop time  (clock): 40890000
elapsed convergence loop time   (time): 41
*********************************************************************

real 0m41.15s
user 0m40.87s
sys  0m0.05s

linux:jonejeff:
```

Note that the "user" time reported by time(1) essentially agrees with the clock(2) time, and represents the amount of time the program spent executing application (non-kernel) code.

Note that the "real" time reported by time(1) agrees with the time(2) time, and represents the elapsed time of program execution.

Report Requirements

Once your program is debugged and tuned to process the largest test file in the 4 - 6 minute timeframe as described above then run your program multiple times from a single batch file on Owens, one time per input file (excluding files testgrid_1 and testgrid_2 which are provided for debugging purposes only).

Provide the following information:

- the values for AFFECT_RATE and EPSILON used (should be the same for all test runs)
- and, for each test file:
  - the number of iterations required for convergence
  - the maximum and minimum DSV values
  - the elapsed execution time
- a short summary explanation of your timing results with comments regarding any unexpected results.

Testing & Submission Instructions

- Initial valuse of AFFECT_RATE = .1 and EPSILON = .1 often make useful starting points. You may use arbitrary values of you choosing to meet the performance window requirement (4 - 6 minutes for testgrid_400_12206).
- Use scp to copy the test files available on stdlinix to your local directory on OSC as needed.
- When working at OSC, be sure to do your program editing, compilation and debugging on the “log-in nodes.” Only submit your batch files to the Owens cluster when you are ready for performance testing and tuning.
- **IMPORTANT** Always use a run-time restriction of 30 minutes on all batch programs submitted to Owens. If your program is running longer than this, you have a bug in your program that you need to find and fix. We have a limited number of cpu hours available for each student on the super-computer. Allowing your programs to run for more than 30 minutes is simply wasting resources.
- As required above, ensure your program can be compiled with "make", before submitting.
- Instructions for using the OSC submission system will be posted on Carmen as they become available.