programming assignment 1

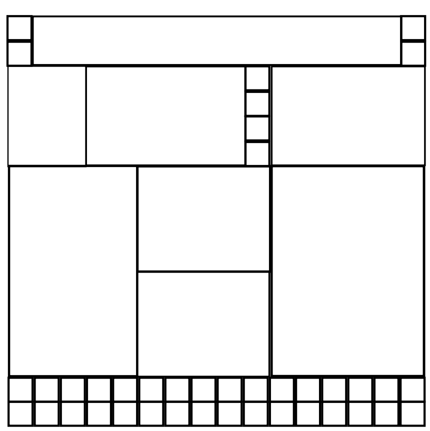
due date: Tue. 2/16

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 rules which include inherent updates to each box as well as some influence from the values of neighboring boxes. (This is referred to as a "stencil computation.") Updated values for all boxes are computed within a single "iteration," all based on the original DSV values. Then, the newly computed values replace the prior values at the same time. The iterative computations are repeated until "convergence," meaning that some threshold is reached, or that all DSVs are within some defined range.

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.



```
1 1                           1
1                             1
1     1           1 1
                  1
                  1
                  1
1         1         1



          1



1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Figure 1: representation of an arbitrary-granularity grid as a sparse matrix

Note that, from a programming standpoint, you may use whatever data structures you feel relevant to the problem, but it may be worth noting that a grid of arbitrary boxes may be represented by a sparse matrix of the upper-left corners (or some other reference points) of the boxes.

input data format

The input files for testing your program will be in the following format (counts 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

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.

You are free to design your own dissipation model (so long as you can justify is as either "reasonable" or "improved" or both), or you may use the simplified approach below.

Performing the following in each iteration:

- compute the weighted average adjacent temperature 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.
- the perimeter of box 0 is 4
- the average adjacent temperature for box 0 would be 100.
- the weighted sum adjacent temperature for box 4 would be (100*4) = 400.
- the perimeter of box 4 is 4
- the average adjacent temperature 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.

100	100	100
100	1000	50
50		
100	100	100

corresponding box ids:

0	1	2
3	4	5
6		
7	8	9

- the weighted sum adjacent temperature for box 3 would be (100*1) + (1000*1) + (50 * 1) + (100 * 1) = 1250.
- the perimeter of box 3 is 4
- the average adjacent temperature 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 average adjacent temperature for box 4 would be 75.
- note that some boxes my overlap irregularly, as in the following example:

box ids:

0	1	2
	3	4
5		
	6	7

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.

- migrate the current box temperature toward the weighted average adjacent temperature by adding or subtracting (as appropriate) AFFECT_RATE% of the difference to the current box.
  - For example, if the current box temperature is 1000, the weighted average adjacent temperature is 900, and AFFECT_RATE is 10%, then the new box temperature would be 1000 - (1000 - 900)*.1, or 990.
  - AFFECT_RATE will be specified as a command-line parameter to your program.

- 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.
- 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").

program requirements

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

```
repeat  until  converged

       for  each  container

              update  domain  specific  values  (DSV)

       communicate  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.

- Implement a sequential version of your program in C or C++.
  - if using C++, you may want to avoid using classes and member functions, but using other C++ features such as vectors, String class, etc. should not present significant problems.
- Input data files must be read from standard input.
- 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.
- Modify your values for EPSILON and AFFECT_RATE as needed (from the command line) in order for your application to successfully converge for the testgrid_400_12206 test data file while using 50k - 100k iterations. This should cause your program to run long enough to measure the impact of threads.
- Print the number of convergence loop iterations required, along with the last values for maximum and minimum DSV, on standard output.
- Compile your program with optimizer level3 (-O3).
- Provide a makefile with your program. Your program should build with the command **"make"**.
- If your program is in C, use program file suffix ".c", if in C++, use program file suffix ".cc".
- **instrumentation**

  Accurate measurement of program run times in C/C++ is non-standard and can be problematic, especially with multi-threaded or multi-process programs. This semester, we will evaluate four alternative measurement utilities and form an opinion as to their utility and usefulness for both sequential and parallel programs. Instrument your programs as follows:
  - Measure and report the run time of your convergence loop using both the Unix time() and clock() 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`

  - For C++ programs, also measure the convergence loop using the std::chrono::system_clock class methods (there is currently no chrono binding for C programs). Note that this may require compilation with the "-std=gnu++0x" or "-std=c++0x" compiler options.
    * http://www.cplusplus.com/reference/chrono/?kw=chrono
  - Also, when executing your program, use the Unix teim(1) utility to report elapsed clock, user and system times.

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

```
linux:jonejeff: time ./smr_csr_serial </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
elapsed convergence loop time (chrono): 41122.6
**********************************************************************


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

linux:jonejeff:
```

report requirements

- Run your program against all of the test files provided (/class/cse5441/testgrid* on stdlinux servers), providing the parameter, convergence and timing information as specified above.
- Summarize your timing results, being sure to answer the following questions (at a minimum). Which timing methods seem best for serial programs? Based on what you have observed so far, hypothesize as to which timing methods may be best suited for parallel programs.
- Submit all reports in .pdf format.

testing & submission instructions

- Test and submit your programs on the Oakley cluster.
- For small programs which run quickly to completion, an "interactive batch" (note the oxymoron ...) shell on OSC can be useful.
  - When benchmarking, it is advantageous to allocate an entire node so that your program will not be time-sharing with other programs, on Oakley we do this by requesting 12 processors per node. To allocate a dedicated node and begin an interactive computing session, use: **qsub -I -l walltime=0:59:00 -l nodes=1:ppn=12** (qsub -"capital eye" -"little ell" walltime . . . -"little ell" nodes . . .).
  - When testing and not benchmarking, it is not necessary to request an entire node. When OSC is busy, you may get an interactive session on a shared node quicker by using: **qsub -I -l walltime=0:59:00 -l nodes=1** (qsub -"capital eye" -"little ell" walltime . . . -"little ell" nodes . . .).

- For the time being, you will need to copy the test data files from stdlinux to OSC. (We hope to have an OSC repository created in time for the next lab.)
- submission:
  - Create a directory "cse5441_lab1" within your home directory on OSC.
  - Within this directory, place:
    - all program files (.c or .cc files);
    - makefile
    - report in .pdf format
  - Ensure that both your home directory and your cse5441_lab1 directory have group permission to read and execute.
  - Ensure that your submission files all have group read permissions.