Adaptive Mesh Refinement Implementation

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 effectively 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.

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: line 2:

line 3:

line 4: line 5: line 6: line 7: line 8:

line last:

<number of grid boxes> <num_grid_rows> <num_grid_cols>

<current_box_id> //starting with 0

<upper_left_y> <upper_left_x> <height> <width> //positions current box on underlying co-ordinate grid

<num_top_neighbors> vector<top_neighbor_ids> <num_bottom_neighbors> vector<bottom_neighbor_ids> <num_left_neighbors> vector<left_neighbor_ids> <num_right_neighbors> vector<right_neighbor_ids> <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

-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 it 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: corresponding box ids:

– 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 average adjacent temperature 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 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. corresponding box ids:

100	100	100
100	1000	100
100	100	100
0	1	2
3	4	5
6	7	8
100	100	100
100	1000	50
50		
100	100	100
0	1	2
3	4	5
6		
7	8	9

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 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: 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 (WAAT) by adding or sub- tracting (as appropriate) AFFECT_RATE% of the difference between the WAAT and the current box temperature.

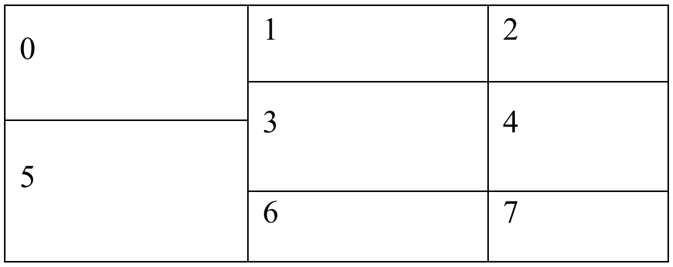
  – For example, if the current box temperature is 1000, the WAAT is 900, and AFFECT_RATE is 10%, then the new box temperature would be 1000 - (1000 - 900)*.1, or 990.

  – 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.

- 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"). – 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 to design your solution as you prefer, 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.

- 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.
- Programs are to be written in the C programming language in a version compatible with the standard compilers currently in use on the stdlinux systems.
- 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 link for setting up standard input and output from the shell command line http://www.linuxdevcenter.com/pub/a/linux/lpt/13_01.html
- 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 pro- grams. 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 appli- cation successfully converges for the "testgrid_400_12206" test data file, while running for approximately 3 - 6 minutes. This should cause your program to

run long enough to measure the impact of threads in future labs. NOTE: the purpose of this requirement is so that your program will run quickly enough to be convenient and also long enough for comparative measurements. Anything within this runtime range is perfectly fine. Performance on stdlinux can be variable. Try to meet the spirit of this requirement while allowing for some variation in specific run times.

- Print the number of convergence loop iterations performed, 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"** (with no arguments), i.e. the make file should be named as **"makefile"** within same directory.
- Use program file suffix ".c" for all program files.

Input Files

This section, which contains a characterization of input files proivded, 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 slight differences due to your specific program and your implementation of the algorithm. Should your results vary significantly from those below and you are using the same dissipation model, then you may be mis-interpreting the requirements or you may have a bug in your program.

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. Converges in 1,508 iterations testgrid_50_201 - data file with 201 variable overlap boxes on a 50x50. Converges in 2,286 iterations testgrid_200_1166 - data file with 1,166 variable overlap boxes on a 200x200. Converges in 14,461 iterations testgrid_400_1636 - data file with 1,636 variable overlap boxes on a 400x400. Converges in 22,283 iterations testgrid_400_12206 - data file with 12,206 variable overlap boxes on a 400x400. Converges in 75,269 iterations

Instrumentation

We will use multiple 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() 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

o Additionally, measure the convergence loop using realtime clock from POSIX real time library. To use this libarary, add <time.h> to your program and compile with **"-lrt"(dash, small-L, small-R, small-T)** option. Use clock_gettime(CLOCK_REALTIME, struct timespec) to get the timer value. Refer to the following link for more info. – http://linux.die.net/man/3/clock_gettime Usage Example struct timespec start, end;

```
o      double diff;
o      clock_gettime( CLOCK_REALTIME, &start);
o      {
o        <<< Scope of Evaluation >>>
o      }
o      clock_gettime( CLOCK_REALTIME, &end);
o      diff = (double)( ((end.tv_sec - start.tv_sec)*CLOCKS_PER_SEC) + \
o         ((end.tv_nsec -start.tv_nsec)/ NS_PER_US) )
```

- • Also, when executing your program, use the Unix time(1) utility to report elapsed clock, user and system times.

Program Output

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:naveen: time ./amr_csr_serial 0.1 0.1 </program/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:naveen:
```

Report Requirements

- Run your program against all of the test files provided (on stdlinux: /program/testgrid*), providing the param- eter, convergence and timing information as specified above.
- Summarize your timing results.