Nov. 19, Assignment #4, due Dec. 2 @ 11:59 pm (10%)

Gardner Home | Course Home

Marking: For this assignment, the software is worth 60% and the documentation 40%. Software that does not build and execute on lockhart will receive 0%. The report does not have to be beautifully bound, but should be presentable and not be hand written.

Red/Blue simulation

The "Red/Blue computation" is described in textbook exercise #10 on p. 111, and you need to read that. **The algorithm is not fully described here!**

Be aware that, despite some similarities, this is not a "game of life": the respective quantities of coloured cells do not change as the program runs. Rather, the initial colours migrate around the board. Design a **scalable parallel algorithm** based on the factors we have studied. In particular, you can revisit the Successive Over-relaxation case study of Chap. 6, taking it as a model for applying considerations of work assignment, load balancing, granularity, and so on.

Clarification of the red/blue algorithm

Conceptually, all legal red moves must happen simultaneously for the entire board, followed by all legal blue moves. Of course, you have to produce that "conceptual effect" by means of the usual iterative program looping.

Consider the following 8-column row (@=space here, but you should print a real space):

[RRR@@BBB]

As a human in the 3rd dimension eyeballing the row, you can see that only one red is a candidate to move right. Thus, the legal result of the red half-step is:

[RR@R@BBB] *correct*

Don't go over the same row again and again to move reds into spaces that open up! That would produce:

[@@RRRBBB] *wrong*

Clarification of stopping conditions

The two conditions should be tested *after* each full step, and the density is tested first:

The maximum colour density condition is looking for a point where enough red or blue cells have shifted together so that *some* tile has become more than c% red or blue. We'll define this unambiguously so that our programs all stop at the same time when given the same initial board:

If a tile is t*t cells, then the maximum number of same-coloured cells for continuing the computation = t*t*c/100, **all done in integer arithmetic** .

For example, if tiles are 8 cells wide, and c is 70%, then up to 8*8*70/100 = 44 red cells or blue cells are allowed, but 45+ of either red or blue in any tile will stop the program.

Thinking through the effect, we know that random initial assignment means that any board will have around 1/3 of each colour cells overall, though for small tiles, their coloured fractions could vary widely. Therefore, specifying c<<33 will likely stop the computation immediately, while c>>33 will rarely be achieved--except with small tiles. So for large tiles, the useful range of c is probably a narrow band above 33. Try it and see.

Past experience has shown that for a given combination of board and tile sizes, there's a certain c% that can be reached "soon", meaning in under 200 iterations. Call that percentage "D" for max. density. If you want to go

above D, by even 1%, you won't reach it no matter how long you run. It would be very interesting to find some formula that gives D as a function of board and tile sizes.

The second condition, reaching a maximum number of full steps, is straightforward. It is only checked if the density condition was not triggered.

Java implementation

Code your solution to run on lockhart using Java 8 in NetBeans according to the program specification below. Don't forget that Parallel Amplifier claims to work with Java. It might help you optimize your program.

You can assume that the maximum no. of threads is 16, for the purpose of defining data structures. The TA will not run your program with >16 threads.

The program is called **jrb** , and takes 5-6 command line arguments, in any order. Each one is made of a leading character followed by an integer N. The first 5 are required.

- pN: no. of processors (threads) to use
- bN: width of board >1, so board size is N x N cells
- tN: width of one overlay tile, each N x N cells (b mod t=0)
- cN: max. colour density in integer percent, 1-100 (stopping condition)
- mN: max. no. of full steps (additional stopping condition)
- sN: *optional* random seed

Print an error and abort if p<1, b <2, the tile width t is not a factor of the board width b, or the colour percentage is out of range. If a *seed* is specified, use it with Random(*seed*) to generate the initial board layout (specified below), otherwise, obtain a seed by reading the system clock via Random(System.currentTimeMillis()). **Specifying a seed should result in the same output every time for the same set of numerical arguments on the same platform.** It is not certain whether Java's Random class running on lockhart will give the same pseudorandom sequence as on your own computer. Therefore, lockhart-produced outputs will be considered the "gold standard" for correctness checking.

"t" is an interesting parameter: With t1, the program will terminate immediately. (Why?) With tN=bN, running the program makes no difference to the colour density, so it will either terminate immediately or else run until the max. steps are exhausted. Also, note that "c100" effectively deactivates the colour stopping condition; nonetheless, checking for it is part of the required calculation (which can be parallelized) for each iteration.

To initialize the board we need a sequence of pseudorandom numbers based on a seed that can be specified. In Java, we can instantiate a Random(*seed*) object, then call **nextInt(3)** on it. Fill each cell in row major order based on **nextInt(3)** . The meanings of the values are: 0=white, 1=red, 2=blue. However, you do not have to fill the cells with integers 0, 1, and 2; you can think of another (more efficient?) encoding for the white, red, and blue if you wish, since we only care about the textual output.

The program will run until it satisfies either stopping condition, and then print the final board contents on the file **redblue.txt** . (Therefore, when a seed is specified, the output file can be diff'ed against a known good file. The Windows command like "diff" is **comp** . See "comp /?" for possible arguments.)

Follow this output format precisely: Print one row per line, terminated by \n, using the following character map: white=' ', red='>', blue='V' (uppercase vee). Also print, as the last line of the file *and* on stdout, the command line arguments, the number of (full step) iterations completed, the highest tile colour density in the final board (integer percent), and total execution time (to at least 2 decimal places), all on one line.

While you are checking and debugging your program on small boards, you will find it useful to make printouts on the console at each step, so you can eyeball whether the colours are shifting around correctly. If you do this kind of printing, make sure that it's suppressed from your submitted version.

For this assignment, the part that we want to time is just the simulation, excluding board initialization and final output. By disregarding a major inherently serial part in this way, more obvious speedup should be observed. There are several ways to carry out execution timing in Java. We will use **System.currentTimeMillis()** which reports the current time to milliseconds, more than adequate for our needs. Save/subtract those values to calculate intervals. Since Java runs are subject to arbitrary garbage collection "pauses," be sure to run each set of arguments several times taking the average or median, and you may throw out wild outliers that could be due to garbage collection.

Deliverables

- Source code via electronic submission using Subversion under tag **cis3090-A4** :

  1. Java program "jrb" in a NetBeans "project." Each .java source file that you create must include a **file header** containing your name and student number, the filename, and a brief description of the purpose of the file's class(es). Submissions lacking proper file headers will be penalized.

  2. Accompanying NetBeans files in the same directory, so that the project can be readily opened and built. You can commit all the files and subdirectories in the project-level "jrb" directory, except that we don't need the subdirectory named **build** .

- Documentation, printed and delivered to class.

  1. Write your impressions of working with Java threading. Compare and contrast with pthreads and OpenMP.

  2. Describe your application's parallel architecture and work distributions schemes. Explain how you went about utilizing threads in Java. Did you go for the "classic" approach or use a higher-level library?

  3. Make a timing graph of total execution time (Y axis) vs. number of threads/cores (1-16 with at least 8 data points) using the set of arguments given directly below. Also make speedup (with linear speedup line drawn) and efficiency graphs based on the same data. Write some observations about the graphs. **Given the speedup and efficiency results, how scalable was your parallel architecture?**

     p[varies 1-16] b875 t125 c100 m2000 s3090

     c% is intentionally set so that the full 2000 iterations run.

Bonus marks

Same terms as for A2, but no rematch provision. For contest purposes this time, we'll take 5 timings, throw out the worst one, and average the rest.

Marks for Assignment #4

For the software, mark is 0-6 based chiefly on functionality:

6 = Parallel program that produces correct results and reports timings for any tested args

1.5-5.5 = Same as 6, but with deductions according to deficiencies

1 = Parallel program that runs cleanly, and is coded to address the problem, but results are all incorrect or missing

0 = Fails to build, or crashes without producing any results

In addition, 1/2 mark penalties will be deducted for specific "sins": numerous Java warnings, file headers missing/incomplete, program name wrong, Java errors printed at run time.

The documentation starts with 4 marks, then deduct 1 mark for any item missing, or 1/2 mark for any lame or poorly-done item. If you didn't have any working software to graph, you'll lose at least 1 mark to be fair to those who conducted timing and produced their graphs.

Gardner Home | Course Home