Simulating One-Dimensional Cellular Automaton with the Majority Rule

In your final MP this semester, you are to simulate a one-dimensional cellular automaton (1D-CA) whose nodes update according to the Majority update rule.

Problem Description

For our purposes, a 1D-CA is defined as a linear array of cells or nodes, where each node is a simple finite state machine that can be in one of only two states: 0 or 1. Based on the current values of its neighbors to the left and to the right, as well as on its own current value, each node updates its state to either 0 or 1 based on its update rule. The update rule \( \delta \) we will consider is the \( \delta = \text{Majority} \) rule. We will consider two cases: (i) when the next state of a node depends only on its first neighbor to the left, first neighbor to the right, and itself, and (ii) when the next state of a node depends on its two nearest neighbors to the left, two nearest neighbors to the right, and itself. Thus, in case of (i), node \( i \) updates its state \( x[i] \) to 1 if and only if two (or more) of its three inputs are currently equal to 1, and it updates to 0, otherwise. Similarly, in case of (ii), a node updates to 1 if and only if at least three out of its five inputs are 1. Case (i) is referred to as a CA whose update rule radius is \( r = 1 \), since each node’s next state only depends on the nodes that are at distance at most 1 away from it. Analogously, in case (ii) the rule radius is defined to be \( r = 2 \).

More formally, the 1D Majority (S)CA with \( r = 1 \) have each of their nodes update its state \( x[i] \) according to the rule

\[
x[i] \leftarrow \delta(x[i-1], x[i], x[i+1]) = \begin{cases} 1, & \text{if } x[i-1] + x[i] + x[i+1] \geq 2 \\ 0, & \text{otherwise} \end{cases}
\]  

(1)

The mathematical formulation for \( r = 2 \) is similar.

The goal of this MP is to implement a 1D-CA with \( \delta = \text{Majority} \) rule for the cases (i) \( r = 1 \) and (ii) \( r = 2 \). You are to iteratively keep updating all the nodes’ values according to the given rules, starting from some initial configuration, until one of the following two scenarios happens:

- After some number of iterations, you reach what is called a stable configuration or fixed point. Namely, if \( t \) denotes the number of iterations or discrete time steps, after some number of step \( t' \), all cells in the array have stabilized: \( x^{t'} = x^{t'+1} = x^{t'+2} = \ldots \). In fact, once you encounter \( x^{t'} = x^{t'+1} \), it’s the matter of simple math to show that the values of the array \( x \) cannot change ever again.
what you need to do

you are to simulate the 1d majority ca with \( r = 1 \) and \( r = 2 \) in two regimes. one is the jacobili-like regime, where all the nodes (i.e., array elements) \( x[i] \) update synchronously in parallel: to compute its value at time \( t + 1 \), each node uses its own value and the values of its appropriate neighbors at time \( t \). the other regime is more gauss-seidel-like: using an appropriate random number generator, you at each sub-step pick a single \( i \) between 1 and \( N = \text{the number of nodes} \), and you update that node’s state \( x[i] \) only. to increase efficiency, you can parallelize by having each of multiple processors picking the random node values \( x[i] \) to update. to avoid collisions, you can assign each processor a segment of array \( x \) so that no two processors ever attempt to modify the same value. i am leaving to you to figure out what are the good ways of distributing array elements to processors in this scenario.

all of the runs should start from a randomly generated string of zeros and ones, \( x[0][1...N] \), where \( N \) is the array size (or, in CA terminology, the number of cells).

more specifically, your MP is made of the following subtasks:

**Part I:** implement the synchronous, jacobili-like node updates for 1d CA separately for \( r = 1 \) and for \( r = 2 \). in each case, simulate your synchronously updating CA using \( p = 1, 5, 10 \) and 20 processors. the number of cells (= array size) to test should be \( N = 40, 80, 100, 200, 400 \). for each array size, generate ten random inputs (i.e., boolean-valued arrays \( x[1...N] \)). report the average number of iterations in each scenario; do this just for the execution on a single processor (the number of iterations in this case shouldn’t depend on the number of processors anyway).

do also two performance plots (one for each \( r \in \{1, 2\} \)) that shows the running time as a function of the array size; plot a different performance curve for each number of processors. (so, two plots, each with four different curves - use different colors, or some other way of making different curves corresponding to different numbers of processors clearly distinguishable from each other).

finally, for \( N = 40 \), in each of the ten runs, output the resulting fixed point configuration \( x[1...40] \) that your cellular automaton, when it started from a random initial configuration, has converged to. (that’s right, i want you to actually print ten strings each made of forty boolean values.) your program should also be able to detect if by some chance, instead of a fixed point, you arrived at a two-cycle; in that case, output the two arrays that are forming the two-cycle.

**Part II:** repeat Part I, only this time the cells are to be updated asynchronously, in more of a gauss-seidel manner, as described above. again, for \( N = 40 \), output the fixed point (or two-cycle) in each of the ten runs. do also the performance comparisons, and summarize your findings in the report, as in Part I. remark: do not attempt comparisons between Part I and Part II, as (i) the notion of a single iteration is different in two cases, and (ii) in Part II, the execution time is going to be considerably affected by repeated calls to the random number generator. for the parallel version in Part II, describe carefully in your report (i) how did you split the array elements among the processors, and (ii) how did you address the issue of different processors sometimes needing array element values from each other (namely, for those \( x[i] \) some of whose neighbors are assigned to a different processor). insofar as reporting the number of iterations, consider a single iteration to be made of \( N \) individual updates of different \( x[i] \). (yes, since you’re picking the node to update at random, typically among \( N \) successive updates some elements will be updated more than once, while other won’t get their turn at all, but that is OK.) due to longer expected times until convergence, in Part II do ten runs for each of the array sizes \( N = 40, 80, 100, 200 \).
Part III: This is the bonus part. If you answer it correctly, you will get up to 20 extra points, on the top of whatever you get for the parts I and II. Everyone, regardless of how many credit hours he or she has registered, is welcome to give it a shot. This is my way of saying 'thank you' and also giving you a chance to add a few extra points to your overall MP scores, and possibly offset the poor midterm and/or lower score on one of the earlier MPs. If you are too busy or find the question hard, you don’t have to answer: you can still get the score of 100 if you get the first two parts done correctly.

Here is the question: have you encountered a single situation where, instead of reaching a stable, fixed point configuration, you actually ended up oscillating in a two-cycle? If yes, write down the two repetitive/cyclic configurations you got. If not, can you analytically figure out what $x[1...N]$ for $r = 1$ and $r = 2$ could be a cycle configuration? Is that configuration cyclic with respect to synchronous, Jacobi-like updating regime, or the asynchronous, Gauss-Seidel-like regime? Write in 2-3 simple, plain English sentences why you think one can or cannot get these two-cycles in one regime or the other (or perhaps both)?

The Extra Credit Part

This is the usual extra credit part, which is intended for the 4-credit-hours graduate students only.

Your task is to implement a block-parallel regime of updating $x[1...N]$. That is, split the array, and assign a chunk to each processor. Each processor then updates the array elements within its chunk in the synchronous, Jacobi-like manner. However, different processors may have different speeds. In this case, I specifically want you not to synchronize after every iteration; instead, any given processor should periodically ask for values from other processors that contain the array elements that this processor needs, but then the processor continues updating for a while, using those values similarly to how you treated boundary values in the finite difference exercise in MP4. For concreteness, let’s say that a processor asks its neighbors only every ten iterations, and in the meantime treats the values adjacent to its own chunk to be something like (temporarily fixed) boundary conditions.

Compare both the execution time and the number of iterations in this scenario vs. Part I above. Run five experiments for each of $r = 1$ and $r = 2$, for the array sizes $N = 40, 80, 100, 200$, and for $p = 1, 5, 10, 20$ processors. Plot separately the average execution time and the average number of iterations (the average is across the five runs for each set of parameter values) as a function of $N$; do this separately for $r = 1$ and $r = 2$. (So, in this exercise, there are four plots total.)

The Practical Matters

You should run your programs on the Turing cluster, which can be viewed as a distributed memory parallel system. For more on Turing, visit http://www.cse.uiuc.edu/turing/.

Choose either Fortran or C as your implementation language. The parallel part of your project is to be implemented using MPI. You are expected to submit (i) a single program file and (ii) a .doc or .pdf file with a brief explanation about your algorithm design (how you did the array partitioning among the processors, what is the inter-processor communication pattern, and the like – see [Quinn] for the standard stages of parallel algorithm design, and write a sentence or two for how you implemented each of those stages), brief performance analysis in terms of the speed of convergence to a stable fixed point, and the required performance plots.

Insofar as the content, shape and format of your report, please refer to the prior MP descriptions and also to the detailed instructions posted to the newsgroup earlier this semester.