SPATIAL DECOMPOSITION FOR PARALLEL CELLULAR AUTOMATA

1. The CA function

A cellular automaton (CA) is a computational model which operates on a grid of "cells", usually a two-dimensional grid but it could be one, or three, or some higher number of dimensions. The original definition (due to Conway - "Game of Life") used a 2D grid, with cell values set to either 0 (empty or "dead") or 1 ("live"). The grid is initially set to some pattern of 0s and 1s. At each iteration a new pattern for the whole grid is generated as follows: at each cell, add up the values of all adjacent cells (left, right, up, down, 4 cells at corners). Including the value of the central cell in the sum of the neighborhood is optional. A fixed rule then calculates a new value (0 or 1) of the cell based on the sum of the neighboring cells. This is done for all cells on the grid at each iteration. Iterations continue either until the system is deliberately halted or until it reaches a static state (if it does). The CA model has been shown to be equivalent in computational power to a Turing Machine, in which the starting state of the grid is the input, the grid itself is the machine’s tape and the cell calculation rule describes the machine’s finite set of states.

The CA definition can be extended in various ways - for example by using a range of integers or real numbers for cell values, by increasing the size or modifying the range of the neighborhood, by adding capabilities to the rules such as sensitivity to direction, by associating lists of values to each cell, with different rules for each value in the list, others. Since the original model is already Turing equivalent, these do not add theoretical compute power, but may make computation easier to support specific models.

In our research here, we still restrict each cell to binary values, but we extend the neighborhood and use a particular rule system that allows for very fine-grained control of the result. Specifically:

- We opt to include the value of the center cell in the sum of the neighborhood, this makes computation of the sum simpler and does not materially change the results.
- We use a square neighborhood $N$ around each cell (a Moore neighborhood) defined by a parameter $r$ (radius) - given a cell with grid coordinates $(x, y)$ (standard cartesian coordinates, where $x$ and $y$ are integers - grid position is a 2D array index); the neighborhood $N(x, y)$ includes all cells in the range $(x - r \leq i \leq x + r, y - r \leq j \leq x + r)$, of area $A_N = (2r+1)^2$. The possible values of $\sum N(x, y)$ are in the range $0 \ldots A_N$, giving $A_N + 1$ possibilities. (We say that cells in the neighborhood are at a "Manhattan distance" of $r$ using the analogy of each cell is a city block, and to get from one cell to another we must follow the streets and not cut in diagonal through the block).
- We compute our cellular automata on a square, $n \times n$ grid $G$, of cells $G(x, y)$. To compute one iteration we use a second grid $G_1$; one iteration consists
The Rule and the update function:

all cells within a (Manhattan) distance

where \( n \) we are unable to calculate cells at a distance

movement into update function

total time to compute each iteration; this is not an acceptable cost.

at top, bottom and each side). The border check could easily more than double the

grid of side \( r \). Note that all cells in the border region need to be held constant, since no cell

to pick a uniform border cell density by setting some fraction of all border cells to

computes the central region of the grid, ignoring the border.

the sphere does not preserve the shapes of neighborhoods from the grid).

A better way is to set the border to a constant value (usually 0, but it is possible

to pick a uniform border cell density by setting some fraction of all border cells to

\( f(G(x, y), \sum N(x, y)) \). That is, the grid \( G_1 \) is calculated, cell by cell, using

some function of the sum of the neighborhood of the corresponding cell in \( G \).

We use 2 arrays because we need the old values to calculate the new ones, so we can not update cells in \( G \) which we might need to calculate other cells in the next iteration. After each iteration, we exchange the arrays. Alternately, we alternate iterations - odd iterations calculate \( G_1 \) from \( G \), and even iterations calculate \( G \) from \( G_1 \).

• We are using CA to model growth of plants/bacteria/lichens in low-resource

environments (e.g. the Mojave desert); for this purpose we have found radii of interest to be between 3 and 7.

The Rule and the update function:

• Our rule is an array: \( int RULE(A_N + 1) \); where each entry contains a digit in \( \{0, 1, 2\} \).

• The function is: \( R = RULE(\sum N(x, y)) \); where \( R \in \{0, 1, 2\} \) then, \( if( R = 0) G_1(x, y) = 0; else if( R = 1) G_1(x, y) = 1; else if( R = 2) G_1(x, y) = G(x, y); \)

(That is, use the sum of the neighborhood as an index into the RULE array, if the value is 0 set the cell at \((x, y)\) in \( G_1 \) to 0, if the value is 1, set \( G_1 \) to 1, if the value is 2, copy cell \( G(x, y) \) into \( G_1(x, y) \) .

2. Edge problem:

Given a \( n \times n \) grids \( G, G_1 \) on which we calculate \( G_1(x, y) = f(G(x, y), \sum N(x, y)) \), where \( N \) is a square of side \( (2 \times r + 1) \) centered on \((x, y)\) it is evident that we need all cells within a (Manhattan) distance \( r \) of \((x, y)\) to calculate \( G_1(x, y) \). Therefore we are unable to calculate cells at a distance \( < r \) from any of the sides of \( G \) without making some assumption on the boundary conditions at or beyond the edge.

It might seem desirable to handle the border by introducing a conditional statement into update function \( f \) in: \( G_1(x, y) = f(G(x, y), \sum N(x, y)) \), to check if \((x, y)\) is in the border region and perform a different computation if so. However, in a grid of side \( n \) this conditional would have to be checked a minimum of \( n^2 \) times (and each of those times might require separate checks for \( x \) and \( y \) and for borders at top, bottom and each side). The border check could easily more than double the total time to compute each iteration; this is not an acceptable cost.

A better way is to set the border to a constant value (usually 0, but it is possible to pick a uniform border cell density by setting some fraction of all border cells to 1). Note that all cells in the border region need to be held constant, since no cell within \( r \) of an edge could be correctly calculated. The update loop that calculates \( G_i \) from \( G \) then runs from \( r \) to \( n - r + 1 \) in each dimension \( x \) and \( y \), and only computes the central region of the grid, ignoring the border.

The most useful thing to do, which generalizes well for parallelizing the project, is to implement a toroidal geometry - we connect the top edge to the bottom edge, and the right edge to the left edge. If the grid were a sheet of some rubbery material that we could easily twist and stretch, we are rolling the grid into a cylinder by taping the top edge to the bottom edge, and then we twist the cylinder to connect the right and left (open) ends; the result is an object shaped like a doughnut (torus), which has finite but unbounded surface area. (We might prefer a sphere, but it is a lot more complicated to distort a square grid into a sphere than into a torus, and the sphere does not preserve the shapes of neighborhoods from the grid).
The toroidal scheme generalizes to connecting $P$ grids into a rectangle or square (simplest is, set $P = q^2$, then we can make a $q \times q$ square out of the $p$ grids and again roll and twist the result into a torus, but now we are splitting the space into grids at $P$ separate processes). Having placed our $P$ grids into a toroidal arrangement, we note that each process (and therefore each grid) has one upper neighbor, one lower neighbor, one right and one left neighbor, and for neighbors at corners. For convenience of reference, I choose geographic directions to identify the neighbors of each process - thus the neighbor above is called $N$ for north, the upper right corner is $NE$ (northeast), the bottom neighbor is $S$ and so on. Where $G$ denotes the grid at a given process, here are the designations for neighbors.

<table>
<thead>
<tr>
<th>NW</th>
<th>N</th>
<th>NE</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>G</td>
<td>E</td>
</tr>
<tr>
<td>SW</td>
<td>S</td>
<td>SE</td>
</tr>
</tbody>
</table>

Processes are identified by process numbers from 0 to $P - 1$. So, for example for 16 processes we would use a $4 \times 4$ square of grids, as follows:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

For a side of $q$ (in this case $q = 4$), and numbering rows and columns from top left, for a given process $p$ the row number is $row = p \div q$ (integer division), and the column number is $col = p - q \times (p \div q)$. Inversely, $p = row \times q + col$. Knowing row and column for a particular process, $E$ and $W$ neighbors are $(col + 1) \mod q, (col - 1) \mod q$ respectively, same rule applies to $S$ and $N$ using $row$ instead of $col$. In this way, the 8 neighbor processes can be identified at each process. In the following discussion we will use the labels NW . . . SE to designate particular neighbors.

We now observe that, for gridsize $n \times n$ we are not calculating any cell within $r$ of an edge, instead we are assuming that these cells are calculate by one of our neighbor processes. That is, we assume the top edge of width $r$ of our grid overlaps the bottom strip of width $r$ of the calculated region of neighbor $N$, and the top of our calculated region overlaps the bottom (not-calculated) edge of $N$'s grid. The net result is that there is a border of width $2r$ on our grid; the outer half (width $r$) of that border overlaps the calculated inner grid of our neighbors, the inner part of the $2r$ border is calculated by our process, and is overlapped by the outer edge of our neighbors. Therefore at each iteration, we must send our inner edge region to each of our 8 neighbors, and we must receive the corresponding parts of their inner edge regions and use them to fill our own outer edge.

2.1. Data and program structures: At each process we have:

- The value of $r$, the function $f(G, RULE)$, the array $RULE$ of size $(2r + 1)^2 + 1$, the gridsize $n$, the values $P =$ number of processes, and $q$ such that $q$ is an integer and $p = q^2$
- assorted functions to compute identity of the 8 neighboring processes
- Two grids $G$ and $G_1$ of size $n^2$
- 8 data transfer arrays, sized as follows: 4 side arrays of size $(n - 2r) \times r$, 4 corner arrays of size $r^2$. You may want to have separate arrays for sending and receiving data for convenience in writing communications; or you may save memory by reusing the same transfer array for exchanges with different neighbors.
2.2. Iteration and data transfer logic at each process:

(1) Initialize the grid - this is easily done by setting a central region in 1 or more of the $p$ individual grids to some starting pattern, such as a line, square, or a random distribution of cells set to 1 with some selected probability. (One simple choice for edges is to set them all to 0 and let the patterns generated by the CA grow into the edges). We assume that all grids and edges are properly initialized.

(2) Compute $G_1$ from $G$: 
   \[ G_1(x, y) = f(G(x, y), \sum N(x, y)) \]
   Note that the boundaries have been adjusted to not compute the border region.

(3) For each side $N, E, W, S$: copy the region of $G_1$ from $r$ to $n - r + 1$, of width $r$ at a distance between $r$ and $2r$ from the edge into a side transfer array $st$. (`send`(`st` $\rightarrow$ $N$), `recv`(`st` $\leftarrow$ $N$)). Copy $st_1$ into the region of $G_1$ from $r$ to $n - r + 1$, of width $r$ at a distance less than $r$ from the edge (this is the outer overlap region, note that corners are excluded). Repeat for $E, W, S$.

(4) For each corner $NW, NE, SE, SW$: copy an $r \times r$ region from NW corner of the calculated region of $G_1$ into a corner transfer array $ct$. (`send`(`ct` $\rightarrow$ $NW$), `recv`(`ct` $\leftarrow$ $NW$)). Copy $ct_1$ into NW $r \times r$ corner of $G_1$. Repeat for $NE, SE, SW$.

(5) If printflag is set, output $G_1$ to disk

(6) Exchange $G_1$ and $G$ and goto step 2. Alternately, copy the code from steps 2-5, changing all references to $G$ into references to $G_1$, and all references to $G_1$ into references to $G$. Or use an equivalent programming construct.

(7) Repeat 2-6 some preset number of times $M$, then stop.

Since the number $M$ of iterations may be very large (possibly up to $10^6$), and the grids $G, G_1$ should have side $n$ possibly up to 1000, it is impractical to output everything. The printflag in step 5 is intended to allow printing of selected iterations, say one out of every 100, or the last 500 iterations.