Peter Stanton

CSS 434

5/5/2017

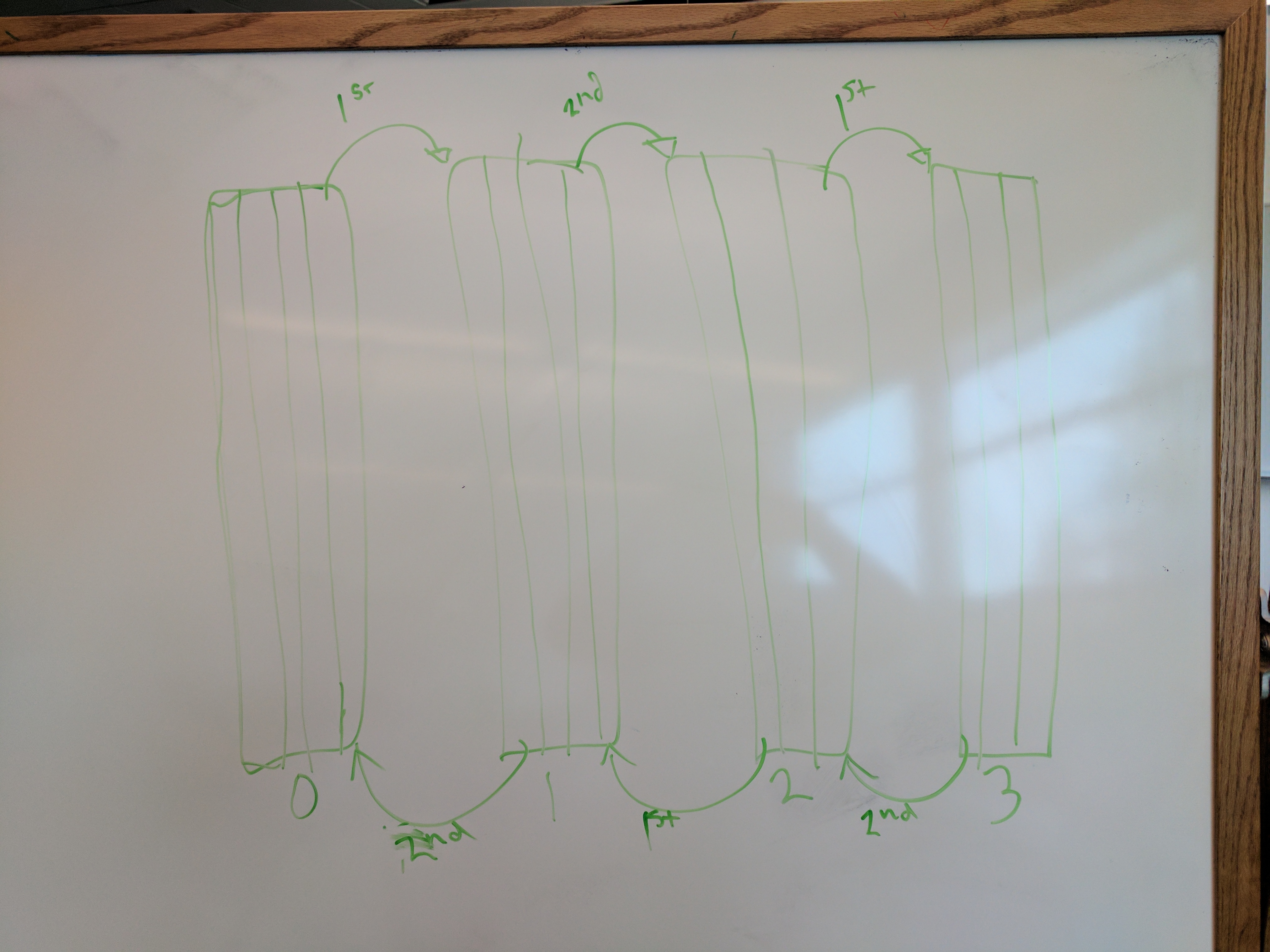
Assignment 2: MPI Java

This assignment covered the conversion of a sequential heat diffusion simulation to a parallel version utilizing MPI Java to run across as many as 4 computing nodes. First off, this requires several more variables than the sequential version in order to use Java MPI effectively. Unlike the sequential version, to be effective, the algorithm has to divide up the simulation space between N ranks. Because this simulation space is a square, that is rather simple. First, we divide the size of the space by the number of nodes. That may not divide evenly, so we also capture any remaining columns by taking the size of the space modulus the number of nodes. We then loop through an array first assigning the “average” number of columns to each rank, and then we add in an extra column for each rank until the orphaned, remainder ranks are depleted.

For simplicity, we are also going to do our work on a 1-dimensional space of 2 \* size \* size. We convert all the given functions that operate sequentially on a 3-d space to ones that operate on 1 dimension by using the function p \* size \* size + x \* size + y, where p is the 3-d z coordinate, size is the size of the square, x is the 3-d x coordinate, and y is the 3-d y coordinate.

We then calculate variables that will be needed later, such as colsUntilMe, which is the number of x-axis columns that precede a given rank on the square, to assist the rank in performing the Forward Euler calculation. Also, we calculate offsetPerRank so every rank knows how many cells the start of each rank is offset from the beginning. This is important for rank 0, the master, to be able to assemble a master array with the results from each rank for printing.

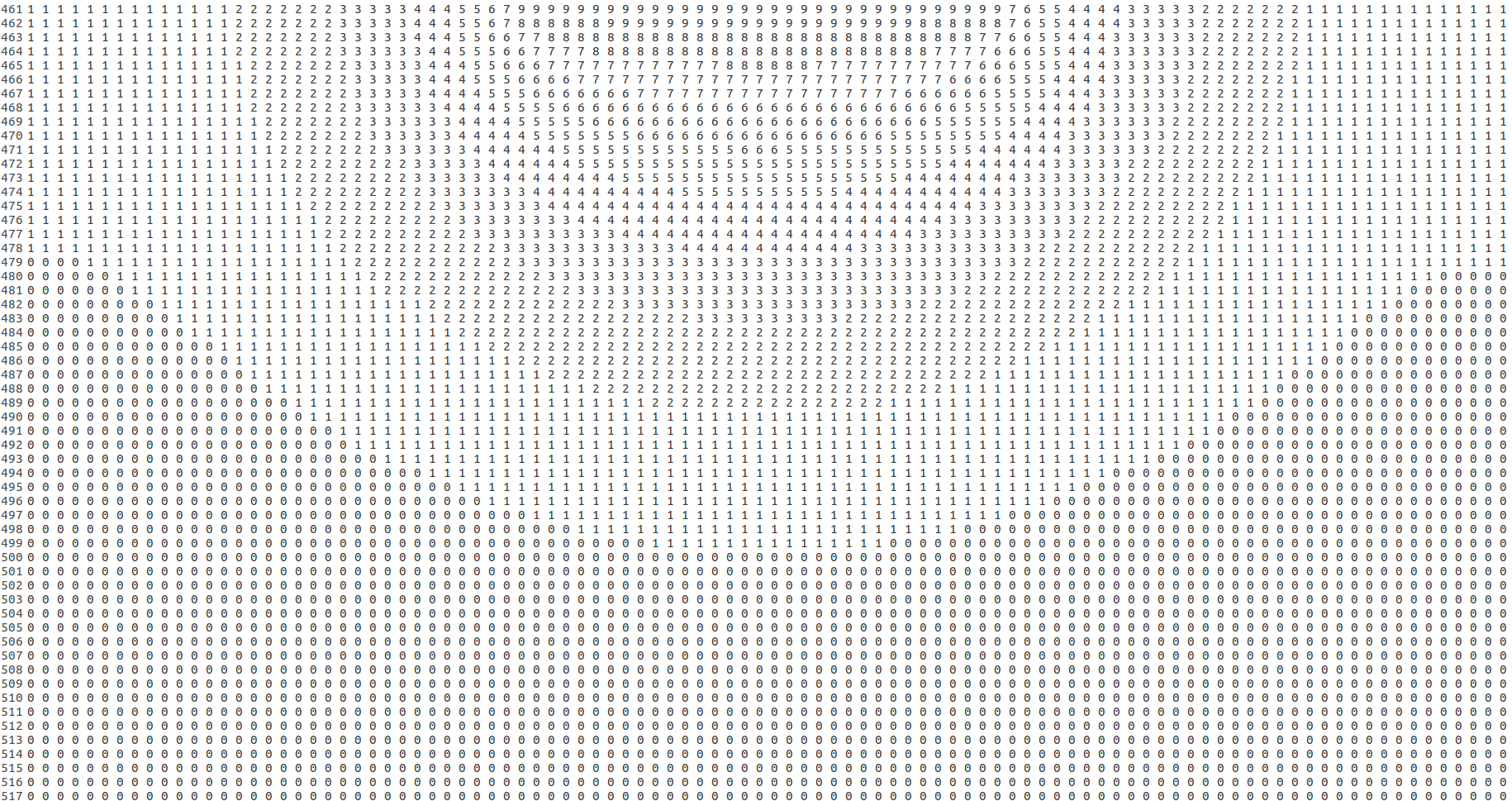
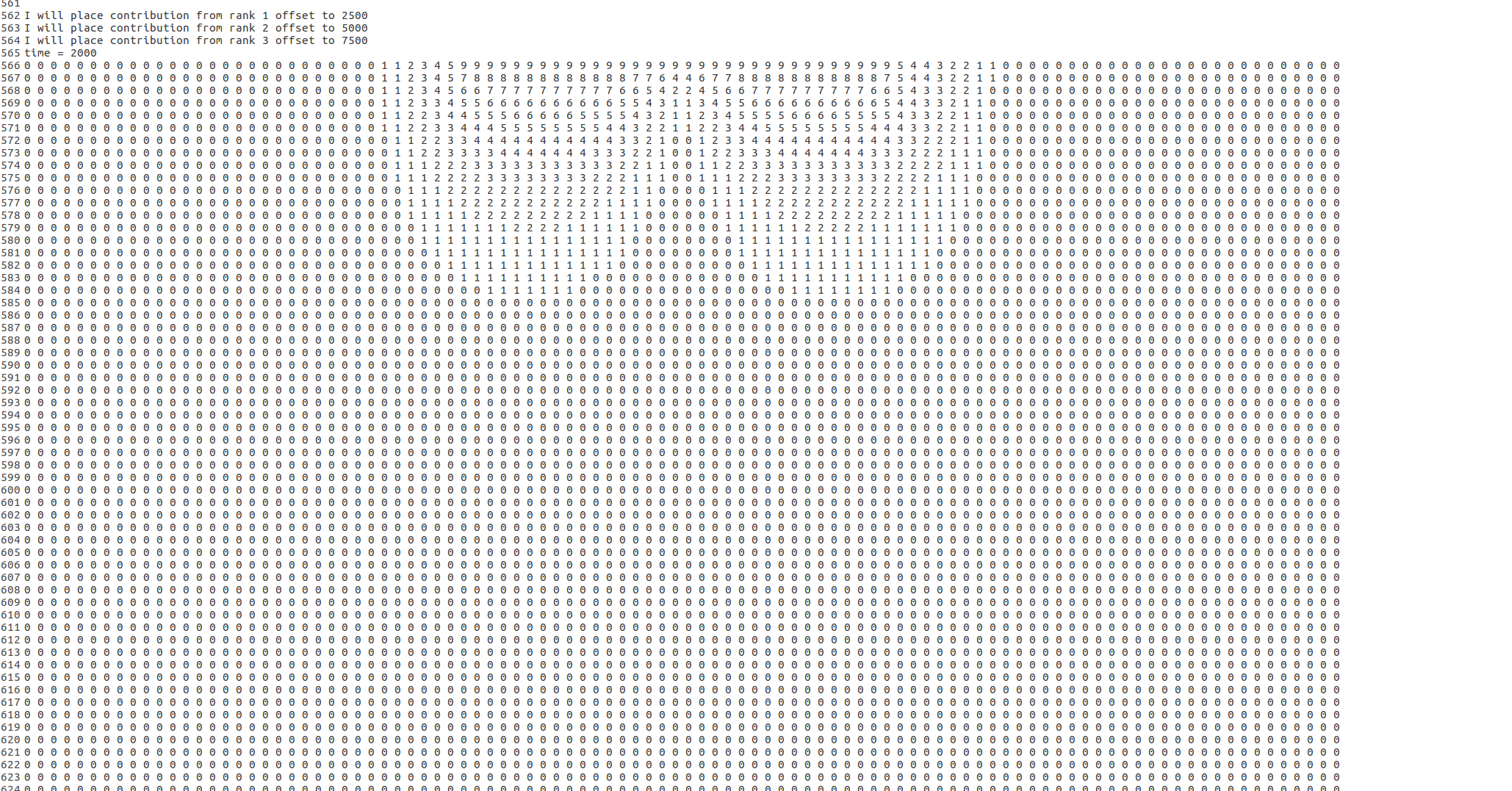
We subsequently pass through three loops responsible for heat diffusion and heating, which are too small to be parallelized and are done on the entire space of each rank.

Then the ranks exchange boundaries. First, even ranks send to the left, then send to the right, then they receive from the left, and receive from the right. At the same time, the odd ranks are receiving on the right (the even ranks’ left send) and then receive from the right (the even ranks’ right send), after which they send to the right and then send to the left to match the even ranks’ receives. Included in the offset calculation of these calls is (p\*size\*size), which will add a full size\*size square’s offset in the 1-d array if we should be acting on the second square. We have a myOffset variable that points to the start of each rank’s left-most column. Finding the start of the right-most column is a bit more complicated, but we can do by adding to offset the sum of size \* the number of columns in that rank, minus size to get from the start of the next rank’s first column back to the start of the right-most column.

When it is time to print, all ranks other than 0 send the portion of their array defined by their internal offset and (p\*size\*size) to the master. Rank 0, master, runs a loop the size of the cluster to receive all contributions from the slave nodes and consults it’s array containing offsets to determine where to place it. It then runs through a printing algorithm to display the results.

Finally, we perform the Forward Euler method on the next iteration’s square. This computation is parallelized so each rank performs it only on it’s own columns instead of the entire square. Column 0, and column N-1 are explicitly excluded from this calculation.

Discussion:

There are clearly some serious issues with my implementation of this program, namely the boundaries exchange, which appears to be sending data the incorrect direction somehow. This results in the strange output that looks like this instead of an even curve of advancing heat. It is also possible that this is an issue with my parallelization of my Forward Euler method, as when I replace my parallelized version of the outer X loop, for (int x = colsUntilMe; x <= ((colsUntilMe + myNumCols)-1); x++) {, with //for (int x = 1; x < size - 1; x++) {, so Euler is performed on the entire square for each rank instead of just each rank’s section, I get a far more reasonable output as shown in the image to the left.

In terms of performance, the issue is fairly obvious. As we increase the size of the simulation space, performance decreases in an inverse fashion. But as we add more simulation nodes, performance should increase. This is because with more nodes, the equation size / MPI.COMM\_WORLD.Size() has a smaller and smaller result, and each rank has less work to do in the Forward Euler Method and can iterate more quickly.

The limitation here is the MPI interface and the cluster hardware. We are limited, of course, to how quickly nodes can pass messages and communicate. In fact, as cluster size increases we are running the risk of one node failing and causing the entire simulation to hang up. This program is not fault tolerant.

We can improve performance by only allocating space to each rank equivalent to 2 \* (number of columns in this rank \* size) instead of a full square of 2\*size\*size, and only performing operations on that relevant space. Another significant improvement would be some sort of message inbox so some nodes aren’t stuck waiting at send and receive events because their counterparts haven’t caught up with them yet.