Brady Auen

Gaussian Elimination

My project ID was to parallelize Gaussian Elimination in order to speed up the computation of the answer. Gaussian Elimination is the process of converting a matrix into an upper triangular matrix through various row operations such as swaps, divides, and adding/subtracting other rows. An upper triangular matrix is define as a matrix where below the diagonal all entries are 0, for example:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 0 | 5 | 6 | 7 |
| 0 | 0 | 8 | 9 |

Note that this is an augmented matrix, the 4, 7, and 9 are the b vector in the equation of Ax = b where A would be the 3x3 region on the left, and x is just a vector of x1 x2 and x3 that we’re actually trying to solve forAll numbers on the diagonal are called “pivots”, so in this example the pivots would be 1, 5, and 8. The divide operation allows you to divide a whole row by a number of your choice, so if we wanted to divide the first row by ½ it would look like:

|  |  |  |  |
| --- | --- | --- | --- |
| ½ | 1 | 3/2 | 2 |
| 0 | 5 | 6 | 7 |
| 0 | 0 | 8 | 9 |

The swap operation allows us to swap two rows for each other, this isn’t always a mandatory operation, as it’s mainly used to make sure we have nonzero numbers at our pivot locations. In this example all of our pivots are nonzero numbers already so we wouldn’t have to.

Lastly our add/subtract operations allow us to take another row and add/subtract it from another, due to the nature of addition and subtraction we can also just multiply the row we’re adding/subtracting by a ratio and continue on, so say we want to subtract 5/2 times the first row from the second row, the result would be

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 0 | 0 | -3/2 | -3 |
| 0 | 0 | 8 | 9 |

As mentioned before, the purpose of Gaussian Elimination is to get an upper triangular matrix A like so:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 0 | 5 | 6 | 7 |
| 0 | 0 | 8 | 9 |

From here we can use back substitution to solve the system of equations which in this case would be:

X + 2y + 3z = 4  
5y + 6z = 7

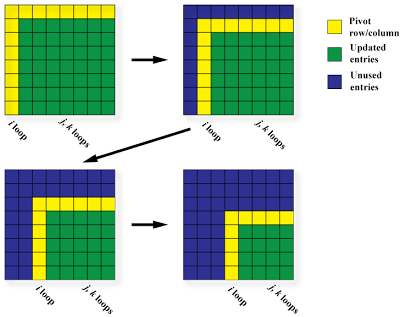
8z = 9

As you can see, putting it in upper triangular format allows us to solve for z, which allows us to solve for y, which allows us to solve for x.

An alternative approach can be to reduce the 3x3 matrix to an identity matrix which means a diagonal of 1’s with everything else set at 0, this eliminates the need for back substitution because it would read as x = int y = int z = int. This is called Gauss-Jordan which creates a row-reduced echelon form

Parallelizing Gaussian Elimination

I’ve spent most of my time so far thinking of an algorithm that might work where you would assign each thread a row and each row would have a lock. Each thread would be passed in its pivot coordinate with an overall shared matrix between them, from there they’d look to see if it’s pivot was nonzero, if not they’d want to swap with another row, the problem here was that as far as I can tell this would be either really difficult or impossible to write without causing major dead lock. A row could read that it’s pivot is nonzero and doesn’t need to switch while being a possible suitor for another row while that other row might only be suitable for the previous row that just said it doesn’t need to switch because it’s okay, or all threads might need to swap and in holding their own lock and requesting another we’d cause deadlock. To skip that headache I thought it would be way simpler to preempt the matrix in main before moving onto concurrent activity. From there everything went wrong due to data dependence in Gaussian Elimination which is visualized as so from <http://www.cs.rutgers.edu/~venugopa/parallel_summer2012/ge.html>



Generally my process for a row would’ve been as follows: divide the entire row by the pivot, and then use the addition/subtraction method to create 0’s below the pivot. So let’s say for example that the third pivot goes first to completion, everything below the third row in the third column is a zero, the row believes it’s work is done, and it returns. Then later the first pivot goes, it finds a nonzero number in the fourth row of the first column and it has a nonzero number in it’s own third column, so when it adds/subtracts to get rid of that nonzero in the first column, we’ve now entered a number into the third column which we’d previously managed to get to zero but the thread that handles that has already exited. Gaussian and Gauss-Jordan Elimination both require that the matrix is handled sequentially due to this issue of data dependence.

So what can we do? (Prototyping before I start coding)

My newest idea which I haven’t had the chance to produce code for yet is to have threads speed up the individual steps as opposed to handling entire rows. I still plan on preempting the matrix into nonzero pivot columns, but after that I could generate n threads (Typically when referring to matrices you use dimensions m x n to denote rows by columns) whose purpose is to divide each row by it’s pivot number. I would want this to happen first, for the moment I’m thinking I’ll just use a join and have those threads specialized for that single purpose but if I come up with anything more efficient that might be subject to change. After that we’d want to get to work on eliminating the numbers below the pivot point, the amount of threads generated could be up to the choice of the user or I might just say I want n threads each getting their own row, the thread would first check to see if it’s linked to the pivot row (and if it is it will do nothing), else it would look to subtract the pivot row from it’s own row so many times until it’s zero (unless it’s already zero in which case it will do nothing). After the column is done we move one column to the right and continue eventually creating a row-reduced echelon form matrix meaning we’ll have performed the Gauss-Jordan elimination method and we’ll have our answer. This method requires no locks as the pivot row would be the critical area but we only read from it, we don’t write to it (unless we’re doing the divide by pivot number step). The trickiest part will probably be making sure threads don’t advance to the next column. I could create m-2 (m-1 so we don’t count our included b/answer vector and m-2 because the first column wouldn’t require a stop) joins and just regenerate the threads for every column, but that could be really inefficient, I’m not sure. Another option would be to have a shared counter (protected by a lock) that’s incremented when a thread finishes making a 0 and when the counter hits a certain number the threads could be signaled to start working on the next column.

Final Product (Works!!)

I only have one test so far as I’m having trouble having my scanner work when I’m constructing my matrix so I had to fill it in with a for loop that goes from one and so on. But it works! The algorithm is to start at the first column and divide the pivot row by the pivot number, from there we generate a thread for every row, the thread first checks to see if it’s in the pivot row or if it’s already 0 and returns if it is, if not then it finds the ratio needed by dividing the number in it’s row by the pivot number and then doing the addition/subtraction across the entire row. All these threads are joined and then afterwards we move one column over and repeat. At first I thought I could help the program concurrently by distributing n threads (n = number of rows) and having them each divide, but it turns out you only need to divide the pivot row at the beginning of each column iteration and that you have to do this dividing step each time you enter a new column, so there was no need for it to be concurrent. If generating threads each time is inefficient it might be possible to adjust the algorithm so we won’t have to do that, but for now it works. I also need to figure out how to get a scanner working in the constructor so I can test more than one type of array.