Homework 6: Parallel Implementation of Dense Matrix Chain Multiplication

Collin Olander

12/7/2018

1 Introduction

For our last assignment, I wanted to parallelize a problem in linear algebra that I thought would be able to benefit greatly from the concepts discussed in this course. In linear algebra, multiplying matrices is not communicative and when trying to do multiplication such as

$$A_{k,l} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,l} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,l} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k,1} & a_{k,2} & \cdots & a_{k,l} \end{pmatrix} B_{l,m} = \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,m} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{l,1} & b_{l,2} & \cdots & b_{l,m} \end{pmatrix}$$

is only valid when when the columns of the left hand matrix equal the rows of the right hand matrix. Furthermore, since the resulting matrix will be a k x m where the (i,j)th input is the dot product of the ith row of A and the jth row of B, this problem, I believe, could be categorized as an embarrassingly parallel problem, and solved as such.

2 Description

2.1 Input and Output

For my input files, I've include 30 csvs, divided into 3 directories, "Small", "Medium" and "Large". What these names are referring to are the sizes of the matrix dimensions. For the sake of keeping some variables constant, I've fixed each chain to a length of 10 matrices (more on this later). If you open one of the csvs, for example, /Small/csv0.csv, you'll see blocks of integers, separated at some point by "—". Each of these csv files represent on matrix chain, where all 10 of those matrices of varying dimensions will be multiplied together and outputted to a file call <correspondingcsv>_Result.csv, so in this example, "csv0_Result.csv".

2.2 Parallelization

In the parallel implementation, what I've done is set up a single go routine to listen for incoming data (the value and it's (i,j)th position) and write that to the resultant matrix, while for every row of the left hand matrix, we spawn a go routine to get the dot product of the appropriate column of the right hand side matrix. To make handling the data objects easier, I created two structs that you'll see in the code, called Element and Matrix. They're in /src/structs/matrix_structs.go and are pretty self explanatory.

And so you may be asking, well then I can't specify how many threads to use? I decided that I wanted to let the scheduler allocate threads as needed and not pigeon hold it to a set number, however, there is -p flag included in the usage statement. What -p does in my implementation allows the user to set an upper bound on the number of threads. While I was executing my program, I found that for smaller matrices, the program was using around 20 go routines at a time, but for larger operations, was using up to 500. So in the parallel execution, I created a channel buffer with a max size set to the -p thread cap. This will throttle the spawning of new threads at the limit set by the user. Setting the -p flag to an arbitrarily large number will allow the main process to spawn as many go routines as necessary for the problem size that its dealing with.

2.3 Decomposition

If I had more time, I would've liked to included a functional decomposition technique for the csv files themselves, but as it stands, my program relies on the embarrassingly parallel nature of matrix multiplication for data decomposition.

3 Challenges

3.1 Too Many Variables

When I first started out on this project, I wanted to include a lot of varying information and look at comparisons in the results. However, I quickly realized that in order to get results that make any sort of sense, I needed to fix a lot of my variables that previously were not constant. For example, with one run of this program, the data involved are the sizes of the matrices, the length of the matrix chain, the number of threads, the total run-time being recorded, and the order of operations in which the multiplication is done (see advanced section). After getting some initial results, I was confused, because it seemed like the medium size matrices were taking the same amount of time as the larger ones. I realized this might have something to do with the fact that I was allowing a varying matrix chain length between 2 and 10. So in one case, all 10 chains in the medium size directory could have been of length 9 and 10, while the lengths of the chains with larger matrices could have been 2 or 3. To combat this, I fixed every csv file to be a chain of length 10. I started to see much more intuitive results after this.

3.2 Run Time Slows Down Very Quickly

You'll notice in the generate script, that the largest dimension of the "Large" matrices is 1000, which in computer science context, is not large at all. However, I initially started out allowing dimensions up to 100000, and I realized that, because I'm multiplying 10 matrices together, the result is the program trying to process a matrix with a possible dimension 100000^{10} (or $1x10^5$ by $1x10^5$) for a total of $1x10^{10}$ inputs. Some of these programs were taking over an hour with the parallel version and I decided that I didn't want to wait around for the sequential version to complete, so I lowered the maximum dimensions for a single matrix down to 1000.

4 Machine Specs

model name : Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
operation system: Linux collin-HP 4.15.0-36-generic Ubuntu

4.1 Memory

collinol@collin-HP: cat /proc/meminfo
MemTotal: 16308648 kB
MemFree: 2878484 kB
MemAvailable: 9708688 kB
Buffers: 584316 kB
Cached: 6646500 kB
SwapCached: 0 kB

4.2 CPU

(paraphrased from command line)
collinol@collin-HP: lscpu
CPU(s): 4
Thread(s) per core: 2
Core(s) per socket: 2
Socket(s): 1

5 Hotspots and Bottlenecks

5.1 Hotspots

The bulk of the work done in this program occurs during the dot product operations when multiplying two matrices together. The analysis part of this program is a highly threaded execution of mathematical operations and because of this, the processing power concentrated at that section of the code.

5.2 Bottlenecks

There were a few bottle necks that I either couldn't address, or didn't quite get to. The first one being the I/O operations. Like I said earlier, I would've liked to have decomposed the tasks at the input point. I think this could have alleviated some of the slow down at this point, however, as you'll see in the advanced section, I think my time was better spent accomplishing an even greater speedup in a different way. It should also be noted that outputting the resultant matrices to result files was naturally a bottleneck as well.

Another bottleneck I encountered beyond just reading and writing to files, was building my matrices from the data. This added an **O**(3*10*10*n*m) run time operation to my program, 3 directories, 10 files per directory, 10 matrices per file of n by dimensions. I thought about how to get around this, and brainstormed about object storage like how Python has pickle, but decided that the payoff may not have been worth the effort

6 Plots

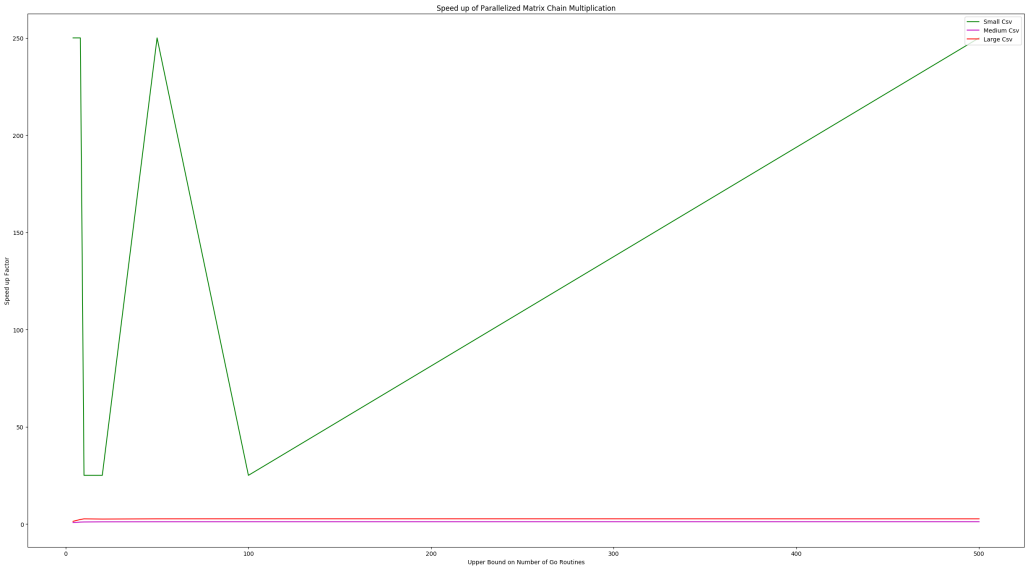

Figure 1: Speed ups for matrix multiplication

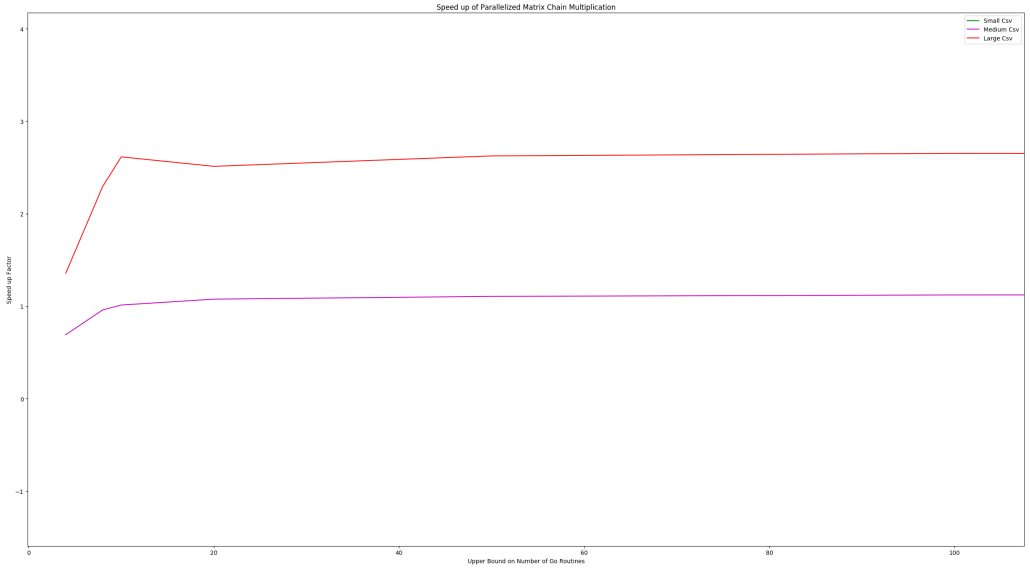

Figure 2: a close up of the Medium and Large csv speed ups

7 Analysis

In the first figure, we see speed up behavior by the smaller csv directory that oscilates wildly. With a low upper bound on the number of threads, the matrix multiplication in parallel was achieving a 250 times speed up.
In the second plot, I've zoomed into the medium and larger csv plots to get a better understanding of their respective speed ups. These larger matrices had a more consistent run time across a different number of upper bounds, but didn't achieve the same peak benefits that the smaller matrices did. However, beyond an upper bound of 10 threads, the speed up was greater than 1 for every size matrix, which was nice to see. This encouraged me to take a look at what might be limiting the speed up of the larger matrices and to get some more results.

8 Limiting Speedups

As mentioned in the bottleneck section, I believe the converting between csv inputs, to matrices and back to strings to write to a csv may have been one of the biggest limiting factors. However, there's an interesting trick hidden within multiplying a chain of matrices together that was limiting my potential speed up this whole time. Matrix multiplication may not be commutative, *but*, it is associative. Why is this important? Well let's look at this example. Say we want to multiply three matrices A B and C, whose dimensions are 10x30, 30x5 and 5x60, respectively. If we were to naively multiply straight through, with A*B and then the result of that times C, we'd be doing (10305) + (10560) = 1500 + 3000 = 4500 operations to get the result ABC. However, If we first multiplied B times C, and then multiply that product to the right hand side of A, we'd be doing (30560) + (103060) = 9000 + 18000 = 27000 operations to get the same product with one sixth the amount of operations.[1]

9 Advanced Features

9.1 Let's Talk About Algorithms

As mentioned in the last section, we can see that maybe there's a better way to get the same product matrix, if we can look at the operation cost across the chain and parenthesize them in a way that would incur the least amount of arthimetic. The matrix chain algorithm that I implemented is an $\mathbf{O}(n^3)$ dynamic programming algorithm which looks at each matrix along the chain, its cost to multiply to either its left or right neighbor, and decides if that is the cheapest option up to that point.

Initially I was concerned that needing to process an $\mathbf{O}(n^3)$ algorithm before even beginning to execute the multiplications would negate the benefit of knowing the most effecient order, but that turned out not to be the case.

9.2 Implementation

The algorithm itself wasn't too hard to implement; there were a lot of resources online that I've cited in my program. However, the difficult part was converting that knowledge of, ok great I know what order they should be done in, to actually executing them in that fashion. What I did was represented the chain as a string "ABCD...J" (10 matrices in each chain) and wrote a recursive function to parenthesize them in the mathematical sense. From there, it was a matter of converting that string, including the parentheses and "*" notation, into a list representing the cumulative order. Perhaps it'd be best to show an example.

[1] https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/

9.3 Example

Let's say we have matrices A,B,C,D,E,F and G, where after our algorithm and parenthesis annotating, we get that the most effecient order to multiply them in would be ((A*(B*C))*((D*(E*F))*G)). After reading in the csv data, I had built myself an array of matrix structs. If I had a way of knowing, not which letter, but which index of that list I should multiply, that would make this problem much easier. So if we convert that string to a string representing indices, we'd have ((0*(1*2))*((3*(4*5))*6)), again, where those numbers represent the matrix at position i. What I did first was look for matrices that should be paired up initially, in this case (1*2) and (4*5). So I calculated those products and saved them in a map, mapping the operation string to a matrix struct. Then, I had for loop tracing the string, checking from position 0 to n, with n decreasing, and checking from position 0 to n with 0 increasing. What this did was allowed me to find substrings that contained the key value of matrices in my map that I already calculated. For instance, By the time I found the substring "0*(1*2)" and could check and see that "(1*2)" was already a key in my map. Therefore, the product of those three matrices would be listOfMatrices[0]*mapOfMatrices["(1*2)"]. And so on, until the final substring is the entire substring and the value that that key represents is the product of the entire matrix chain. This was almost as hard to explain in words as it was come up with this, but let's see some results.

9.4 Results

The result I was most interested in was comparing the speed up between the sequential, naive operation to the parallel, effecient operation runtime.

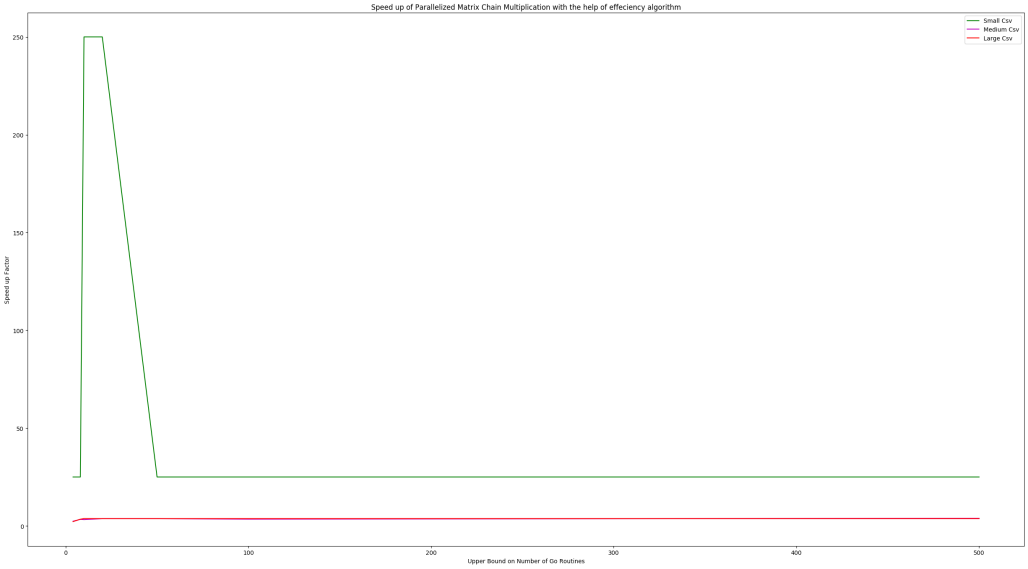

Figure 3: Non-optimized sequential and optimized parallel

We can see that the smaller csv files are still hitting speed ups of 250 times its original speed, except this time it seems to taper off with a thread increase and doesn't bounce back up, like when they're both not optimized. However, on our last graph we were concerned about increasing the speed up for the medium and large matrices, so let's take a look at that.

What I found interesting was that there still seems to a bit of ceiling on the speed up factor. It hovers around 4 times the original speed. However, as before the medium size matrices were slightly above and

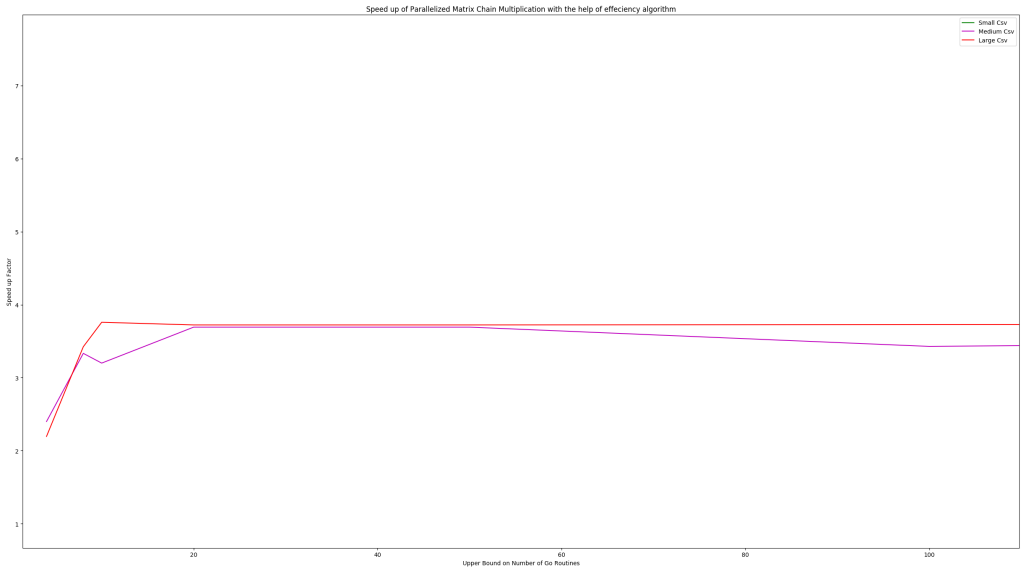

Figure 4: a close up of the Medium and Large csv speed ups

below 1 times the speed, the floor seems to have increased and now levels off at around 3.5 times the original run time, which I was happy to see.

9.5 What If They're Both Optimized?

Now let's see what happens if both the sequential version and the parallel version are allowed to find their most effecient order of operations.

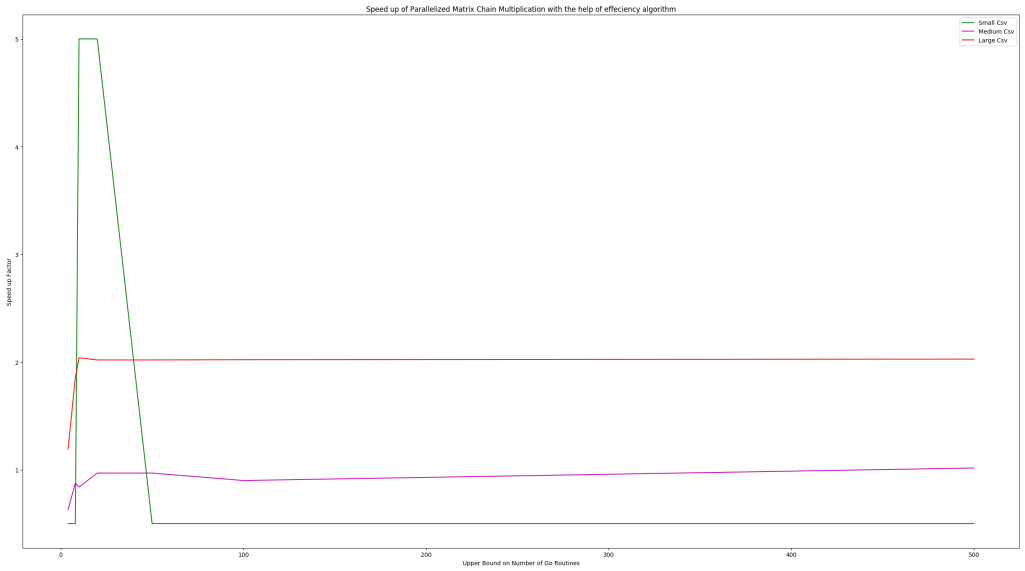

Figure 5: Both versions optimized

Again, we see that large spike in speed up with matrices of a very small size and a relatively low number of threads. However, because of the optimization of the sequential execution, the increased factor is greatly reduced from 250 to 5. Since any number of threads beyond 100 doesn't seem to improve the effeciency at all, let's shorten the x axis to a more relevant range.

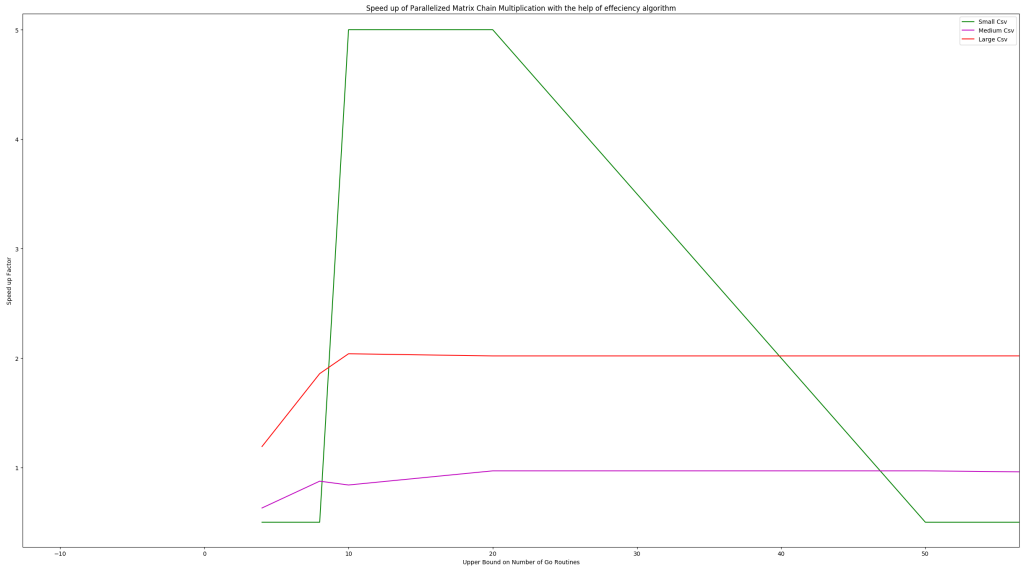

Figure 6: reduced x range

We still see that the medium size matrices struggle with achieving much of a speed up when the parallel

and sequential versions are competing at the same level (ie optimized or not optimized).

10 Key Take-aways

What I found most interesting was the consistent behavior of the directories across different factors. No matter if the one or both versions were optimized, the directory with small matrices benefitted tremendously from a relatively low number of threads (4-10) but seemed overly sensitive to any change in thread number. I believe this because there are so few opertations to do, mathematically, that the time spent elsewhere in the program (I/O, effeciency algorithm, etc) and context switching between threads, could affect the run time a lot more than it would if there were a large number of operations to be processed. Meanwhile, within that same range, the large matrices had a modest speed up that remained constant throughout any thread range, again, probably due to the number of calculations simply overshadowing any other processes that could slow it down. And the poor, poor, medium sized matrices. It seemed the only real advantage the files in that directory saw were when the parallel version was optimized and the sequential version wasnt.

11 Improvements

If I had more time (another week), the aspects I'd look to improve upon the most would be functionally decomposing the directories and csvs for the parallel execution. I'd also like to look into parallelizing the matrix chain algorithm itself. This wouldn't be as trivial as parallelizing the multiplication operation, since its memoizing the data, all the threads would need to read and write to the same table and would most likely spent time waiting for a row,column entry to come in, in order to process the calculation it's trying to do.