Term Project: Singular Value Decomposition (SVD)

Motivation and Problem Statement

Singular Value Decomposition is a matrix factorization technique used in linear algebra, statistics and machine learning. This technique factorizes a matrix into three matrices: U, Σ, and V^t, where U and V^t are orthogonal matrices where their columns and rows are the eigenvectors of AA^t and A^tA respectively, while Σ is a diagonal matrix where the values on the diagonal are the squares of the singular values of the original matrix. As it has a wide range of applications, such as Dimensionality Reduction, Data Compression, Image processing and more, it is a cornerstone of a multitude of programming techniques and applications. Typical computational activities involving matrices are prohibitively expensive.

In short, many functions that deal with matrices and computation, usually go over every element, using some type of nested for-loop, and depending on the given function up to another for-loop or two. This may cause a huge jump in overall runtime. For example, take basic matrix multiplication, the outer for-loop iterates over the rows of A, and the columns of B, while the inner for loop iterates over the columns of A and the rows of B, with another for-loop that takes care of the [i,j] calculation by looping over a row and a column. This is easily on the order of $O(n^3)$. With some abuse of structure, we can make this cache coherent to improve run-time.

Sequential Approach

The currently accepted efficient approach to solving this problem sequentially is done by using a handful of clever techniques. In a related part of linear algebra, we have the technique of QR factorization, where Q is an orthogonal matrix, and R is an upper triangular matrix. Without going into specifics of this factorization, it is used as a stepping stone towards a full SVD factorization. The technique currently being used, according to our findings, is Householder reflections of row and column vectors. Formally, we call this the Householder Bidiagonalization method. Utilizing this method allows us to get a start on performing linear algebra with matrices. Once we create a reflected vector, we then subtract its outer product from the ‘bottom right’ of the original matrix, and then we calculate the dot product of its outer product subtracted from an identity matrix against another identity matrix. We then repeat this for the horizontal vector starting from the same row, col index.

We continue this process, with a handful of more steps, and at each step we ‘move in’ another column. This makes the next problem ‘smaller’ than the previous one. With this technique the math holds and produces the correct SVD factorization. In practice, we have a m * n matrix, where we initially subtract an outer product of the 1x(m-i) vector from it. This is minimally an (m-i)*(m-i) operation. Then, we do essentially the same thing but subtract from an identity matrix, incurring the same cost, followed by a dot product of the same cost.

That is, the overall runtime is on the order of $O(n^3)$ in the best case, but likely closer to $O(n^4)$ if the structure of matrix multiplication isn't taken advantage of.

Parallel Approach

As far as the initial problem, outside of the Householder Bidiagonalization method, all involved calculations are done matrix wise. If the structure of a matrix multiplication problem is observed closely, we can see that it is possible to do it in a cache coherent way. That is, we keep row i of matrix A in cache, row j of matrix B in cache, and row i of matrix C in cache as long as possible. The factor of A[i][0] * B[0][j] can be applied to every element of matrix C's row i at index j.

```
a b c | j k l  [aj + bm + cp]  [ak + bn + cq]  [al + bo + cr]
d e f | m n o  [dj + em + fp]  [dk + en + fq]  [dl + eo + fr]
g h i | p q r  [gj + hm + ip]  [gk + hn + iq]  [gl + ho + ir]
```

This structure indicates that we can use a parallel approach to the calculation without any synchronization management. That is, no mutexes are needed to manage access to a 'shared resource', solely because of the structure of the problem.

That being said, our parallel approach is simple. We designed every matrix operation with care to make it cache coherent, and parallelizable. At the moment, it looks like a thread processes the rows of a matrix according to its own rank. That is rank 0 does row 0, rank 3 does row 3, and so on. Furthermore, they are cyclically assigned to prevent over saturation of threads when the problem size is small.

Implementation

As we kept our design mindful, we did not have need for most synchronization methods or tools. No mutexes were needed for controlling access, no semaphores were needed for signaling. What we did use, was an allotment of openMP threads per matrix operation. The user defines the maximum amount of threads to be allotted for this. We chose 10. The only synchronization we did make use of was the barrier method. Without this method, some threads would attempt to 'return' from a function before the others had finished processing.

Additionally, in the function make_house_vec, there is a for-loop that calculates the sum of squares of a matrix x and stores the result in a local variable sigma. We decided to implement an omp parallel for with reduction to speed this along if possible.

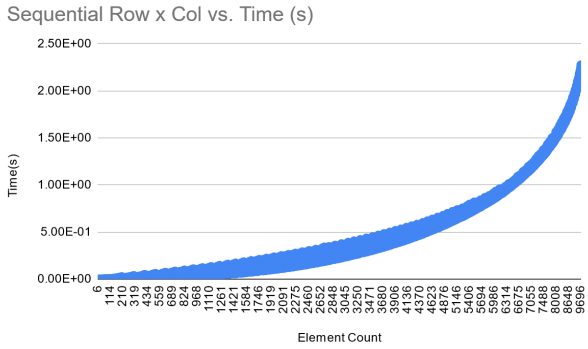
```
//Subtract (beta * outer_product) from matrix
#pragma omp parallel num_threads(max_threads)
{
    int i = omp_get_thread_num() + col;
    while(i < n){
        for (int i = col; i < n; i++) {
            for (int j = col; j < n; j++) {
                Q[i][j] -= beta * v[i-col] * v[j-col];
                if(abs(Q[i][j]) < 1e-6){
                    Q[i][j] = 0.0;
                }
            }
        }
        i+=max_threads;
    }
}

#pragma omp barrier

return Q;
```

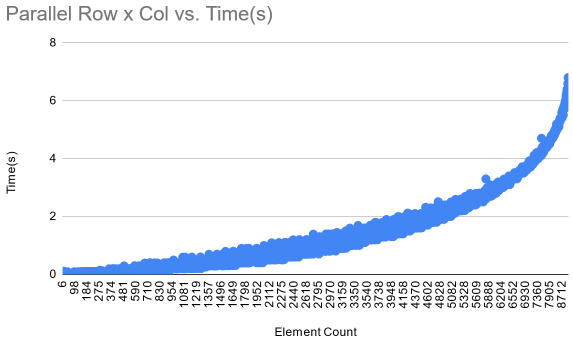
```
#pragma omp parallel num_threads(max_threads)
{
    int i = omp_get_thread_num();
    while(i < matrix_rows){
    // for(int i = 0; i < matrix_rows; i++){
        for(int j = 0; j < matrix_columns; j++){
            if(abs(mat[i][j]) < 1e-6){
                mat[i][j] = 0;
            }
            A[i+col][j+col] = mat[i][j];
        }
        i += max_threads;
    }
}
#pragma omp barrier
```

Results

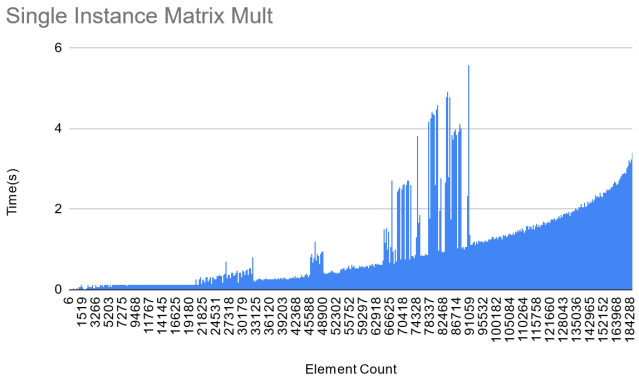
Interestingly enough, the parallel method, on average took about 10 times longer. This means that, per thread created, it increases the runtime factor by some factor k, where k is proportional to the thread count. In reflection, we determined that, due to the iterative nature of the algorithm, there is a point where we experience a diminishing tradeoff between number of threads and problem size. That is, we incur too much overhead during the small portions of the problem that we actually perform worse.

When plotted however, the growth rate for both sequential and iterative are equivalent.





In lieu of these findings, we initiated an investigation to determine where the ‘tradeoff’ point is, and to determine if we could prevent that slow-down from happening. To find this tradeoff, we implemented a single instance of an MxN matrix multiplication function call, and timed it from start to finish over several iterations where the size of the matrix increased first by column then by row.



Outside of a handful of outliers, the part we are looking for is when the runtime goes ‘constant’. More simply, we are looking for the rate of change in execution time over size to ‘zero out’. Disenheartengingly, we found this to be around 33,000 elements. While not out of the realm for some applications like Neural Networks and robust GPU’s and SVM’s, our physical devices just could not handle the load at those sized problems.

We repeat this process for the horizontal direction, and subtract from the matrix then subtracting their outer product, multiplied by a scalar ‘beta’, from the ‘bottom right’ side of the original matrix. In doing so, we perform a type of QR factorization, but when done repeatedly, where every iteration we go one more column in, we can achieve a full factorization where the remaining matrices.

Additional Investigation

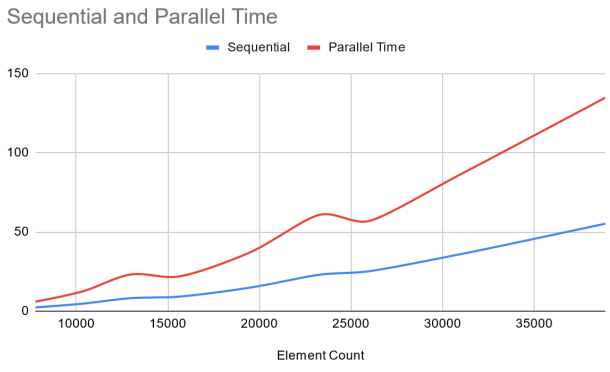
During an additional investigation to determine if we could improve the efficiency of the algorithm, we had attempted to determine the tradeoff point between overhead and speed improvements. Initially, this additional investigation had some promising results.As the algorithm itself enforces an iterative process, certain steps can still be done in parallel, and that is where we focused. That is the matrix operations. To test these matrix operations, we took several considerations into account, first was cache disturbances to timing. To deal with this, we enforced ‘cache clearing’ by allocating more memory than our cache between runs, then restarting the timer. Our timer is the base ctime library clock() function.
Additionally, the matrices are scaled on the order of the ‘thread-count’ squared. That is, each row has some constant i times the square of the thread count elements, and there are j times the square of thread count rows. This allowed even distribution of work to each thread, while scaling the matrices quickly.

Our initial results are as follows:

```
root@DESKTOP-G8P92T0:~/svd_dir# ./seq.o 5
75x50 Time in seconds: 1.01562
100x50 Time in seconds: 2.20312
100x75 Time in seconds: 3.82812
125x50 Time in seconds: 3.42188
125x75 Time in seconds: 6.57812
125x100 Time in seconds: 10.3594
150x50 Time in seconds: 5.79688
150x75 Time in seconds: 9.70312
150x100 Time in seconds: 15.1875
150x125 Time in seconds: 23.5312
^C
root@DESKTOP-G8P92T0:~/svd_dir# ./par.o 5
75x50 Time in seconds: 0.65625
100x50 Time in seconds: 1.28125
100x75 Time in seconds: 2.3125
125x50 Time in seconds: 2.04688
125x75 Time in seconds: 3.73438
125x100 Time in seconds: 6.0625
150x50 Time in seconds: 3.48438
150x75 Time in seconds: 5.65625
```

Where ./seq.o 5 is the sequential version being run with the matrix being scaled on the factor of 5^2, or 25, and ./par.o is the parallel version being run on the same scale. We see the matrix dimensions are equivalent, up until the 150x75 matrices. Execution was cut off out of runtime consideration.

In practice, we tried to make the cutoff relate to the thread count, as the above behavior tended to be repeatable for thread counts over 3. The formula was as follows Cutoff = (0.5 * thread_count^2), where the value being compared was the row count of the matrix being considered.



The results however, did not change as expected. However, the run time growth rate appears to be slower.

Conclusion

The householder bidiagonalization method, as is, is an efficient method for solving for SVD decomposition. While it is an active area of research concerning methods on how to parallelize it, our attempt did have several shortcomings. Our initial hypothesis stated that if we did a blanketed parallelization of any matrix operation, we should achieve speed up. In retrospect, we discovered this is most definitely not the case. Upon further analysis as to why that is, we see that as the householder method progresses, these matrix operations shrink. When matrices themselves are small enough, the overhead from producing a number of new threads for parallel processing will often outweigh the benefit. To that end, parallelizing matrix operations in general does typically give an improvement in performance time, however, in lieu of compiler optimizations, cache optimizations and coherence, among other things, much of the speed up to be gained through parallel processing, could potentially be gained by these other tools. Our attempt to find a threshold for when the costs overcome the benefits is worthwhile, and may still lead to improvements, but at the current time, that threshold has not successfully been discovered.

From there, it is also noted that not every matrix operation may need to be done in parallel. While we consider it for sizable problems, it may not be appropriate when compared to a better designed basic algorithmic approach that may not be parallelizable. There is room for parallelization in this method, however our current investigation into this is still young, and much is left to be investigated, discovered and applied.