Singular Value Decomposition

High Performance Computing for Mathematical Models

Charles Marshall, Maxime Richard

February 14, 2020

1 Introduction

For this project we have chosen to study the *Singular Value Decomposition* (SVD), and how its computation may be accelerated using the multiprocessing API: `OpenMP`. In particular we compute the QR matrix factorization using householder reflections, and an adapted SVD algorithm to compute the eigenvalues, or *singular values* of a given matrix. This combination of algorithms is quite costly, and has room for parallelization of several reoccurring vector operations, and most importantly: matrix multiplications. As we will see, our parallel algorithm achieves substantial speedup compared to its sequential counterpart as the size of the matrix being decomposed grows.

In section 2 we describe our methodology and the concepts at play, followed by detailed descriptions of the algorithms, their bottlenecks, and our optimizations in section 3, results and analysis in section 4, and finally we conclude the report in section 5.

2 Methodology

In this section we will briefly formalize the SVD and the closely related QR decomposition, before describing in detail the householder algorithm which we have chosen to parallelize.

2.1 Singular Value Decomposition

The Singular Value Decomposition is a generalization of eigenvalue decomposition that holds for any real or complex matrix no matter the size. This factorization is extremely useful in regression settings with direct applications to least squares methods, PCA, and many other problems surrounding linear systems.

Let $A \in \mathbb{R}^{m \times n}$ be a matrix with m rows and n columns. SVD states that there exists a factorization for this matrix of the form $U\Sigma V^T$.

Due to its vast applicability, there exist a multitude of interpretations for this decomposition, but the most elegant in our opinion (and that which will respect the length of this report) is the following;

We consider our above matrix A to be some transformation resulting in an n-dimensional ellipsoid, residing in an m-dimensional space. Following this example, through SVD we obtain the following values:

- $\mathbf{U} \in \mathbb{R}^{m \times m}$ is an orthogonal matrix satisfying $U^T U = UU^T = I$ (unitary in the complex case), and describes the rotation effectuated by A in $\mathbb{R}^m$.
- $\mathbf{V}^T \in \mathbb{R}^{n \times n}$ like U, is an orthogonal matrix describing rotation in the space $\mathbb{R}^n$
- $\mathbf{\Sigma} \in \mathbb{R}^{m \times n}$ is a diagonal matrix containing the eigenvalues of A, of which there are n, all of which perform a scaling along the coordinate axes. These are known as the **singular values** and are of particular interest in our computation.

Although useful, computing the SVD in practice is not so simple. There exist many algorithms ranging in utility, but for this project we would like to direct our focus on extracting the singular values contained in Σ by using a QR decomposition.

2.2 QR Decomposition

Along the same grain as the SVD, a QR decomposition is a related method, applicable to any real or complex matrix, square or rectangular.

Again, let $A \in \mathbb{R}^{m \times n}$ be a matrix with m rows and n columns. A decomposition exists for this matrix of the form $A = QR$, where:

- $\mathbf{Q} \in \mathbb{R}^{m \times m}$ is an orthogonal (unitary in the general case) matrix with orthonormal columns.
- $\mathbf{R} \in \mathbb{R}^{m \times n}$ is an upper triangular matrix with nonzero values along the diagonal.

In particular, we will compute our decomposition using householder reflections [3]. In the following section, we provide precise details regarding our QR decomposition, which we will refer to as the *householder algorithm*, as well as our second algorithm for computing the eigenvalues of a given matrix, or equivalently the Σ of an SVD.

3 Algorithms

In this section we describe the two algorithms used in this project, and we identify the principal challenges and bottlenecks from a computational standpoint.

3.1 Householder

The householder QR algorithm makes use of a rather simple algebraic notion known as the householder transformation. We give our definition with some adaptations of [4]:

A householder transformation of a vector v with respect to a hyperplane with normal vector w is defined as;

$$Hx = v - 2ww^T v$$

Where Hx is a linear transformation of a matrix H applied to x. We say $H = I - 2ww^T$ is a householder matrix.

3.1.1 Householder Algorithm

In practice, computing QR using householder transformations assuming we have a matrix $A \in \mathbb{R}^{m \times n}$ and $m \geq n$ goes as follows:

1. Compute $P_i = I - w_i w_i^T$ for a given column i of the matrix A.
2. Repeat 1 n times and store $P_1 ... P_n$
3. Compute $R = P_1 P_2 ... P_n A$

This gives us our matrix R. Q is given implicitly, but we will compute it directly in order to next find Σ using SVD. We compute Q as follows:

$$Q = P_n P_{n-1} ... P_1$$

3.1.2 Bottlenecks

In this algorithm we make use of a number of vector operations. Namely, outer product, product, and subtraction, but we note that we must do a matrix-matrix multiplication n times to compute R, and then n more times to compute Q. This is by far the most expensive operation in the algorithm and is the main bottleneck we would like to parallelize.

3.1.3 Code

We adapted the C code from [3] for the previous algorithm, which is included in our submission as `householder.c`. We will speak more about this in section 3.3.

3.2 SVD

The main purpose of this section is to describe how we compute Σ given QR, and the original matrix A.

We will assume our matrix $A \in \mathbb{R}^{m \times n}$ contains real values and $m \geq n$. We also assume we have the QR decomposition described previously.

3.2.1 SVD Algorithm

The algorithm is divided into two main steps; first we reduce the initial matrix A to a bi-diagonal matrix, then we apply the Householder algorithm defined in the previous section.

The first step iterates n times over A and is split into two steps;

1. Introduce zeros below the diagonal of the i^{th} column where i is the iteration number.
2. introduce zeros to the right of the upper diagonal of the i^{th} row.

After n iterations we have a bi-diagonal matrix, denoted D, on which we are going to apply the second step of the algorithm, called *Wilkinson shift*. The second part iterates a non-fixed amount of times, depending on a convergence criterion which we explain in this section. The purpose is now to obtain Σ while making use of our householder algorithm.

The algorithm now considers the whole matrix D in the case where A is square, or the upper square matrix of size $(n \times n)$ if A is rectangular. The point of this part is to find the i^{th} singular value on the diagonal by repeatedly applying the two following QR transformations:

1. On the matrix $D^T D - s * I_i$ where D is the bi-diagonal matrix obtained by the previous step, I_i is the identity matrix of size n and s is a scalar.
2. On the matrix $A * Q$ where Q is the Q matrix resulting from the previous QR transformation.

The values of A are replaced by the values on the diagonal and upper diagonal of the resulting R_1 matrix obtained after the second QR transformation. This is repeated until the $[n-1, n]^{th}$ element of A is arbitrarily close to 0 (this is the convergence criterion, where we choose a very small number as the threshold). At this point, n is reduced by one and the same algorithm is applied on the sub matrix of A of size $(n-1 \times n-1)$ until we reach a 2 by 2 matrix. Once we have this, convergence is achieved and we obtain the diagonal matrix with singular values Σ

3.2.2 Bottlenecks

The two main parts of the SVD algorithm described above may not be parallelized. This is because each iteration of the first part depends on the result obtained in the previous part. Thus, the $i + 1^{th}$ step can not be computed before the i^{th} step is finished.

Similarly, for the second part each step depends on the previous one. In addition, the number of iterations is not fixed, so we decided to parallelize the most costly sub-functions called in these two steps and in the Householder algorithm. We will descibe these optimizations in section 4.

3.2.3 Code

The SVD algorithm described previously is an adaptation of the `svdgui` algorithm from [2] which we have translated from MATLAB to C.

3.3 Optimizations

In our submission is a file named `utils.c` which contains several mathematical operations we have decided to parallelize, and some which we have decided to leave sequential. We will describe these now.

- `matrix_mul` This is the matrix-matrix multiplication algorithm which dominated the complexity of both of our algorithms. We use static scheduling amongst threads to parallelize the $O(n^3)$ computation.
- `matrix_minor` This computes the minor of a matrix and is of the order $O(n^2)$. We parallelize this function.
- `vmul` Computes $I - vv^T$ which takes $O(n^2)$. We parallelize this computation
- `vnorm` Computes the norm of a vector. Parallelizing this function improves the performance a very small amount, but we do it nonetheless.
- `vdiv` Computes vector-scalar division. This case is similar to the previous function.
- `matrix_transpose` We use this function in order to check our results, and we parallelize it in order to speed up the $O(n^2)$ complexity.

There are several other functions such as vector matrix multiplication which we seldom use, and thus decided not to parallelize. This is apparent in the householder, and SVD functions of our code.

4 Results

Tests were performed on a machine with 8 cores clocked at 2.8GHz including 4 hyper-threaded with 32Go of ram.

We compute the execution time of our householder, followed by SVD algorithms, for different matrix sizes. We use both square and rectangular matrix sizes ranging from (10×10) to (150×150). The results for square matrices are summarized in figure 1. The choice of relatively small matrix sizes was motivated by the fact that computing our algorithm sequentially with a (300×300) matrix took more than 2 hours and 30 minutes to compute. In figure 1 we can see that the parallel algorithm achieves a $3x$ speed up for the largest, (150×150) matrix. For the smallest matrices, the sequential algorithm is approximately 5 times faster; timed at 0.002 seconds for the sequential algorithm and 0.0104 seconds for the parallel one for a (10×10) matrix.

We see room for an improvement to the parallel algorithm wherein we do parallel computations unless the size of the matrix or vector being computed is less than 10, then we force the algorithm to excute sequentially. We believe this could lead to a slight speed up of the algorithm.

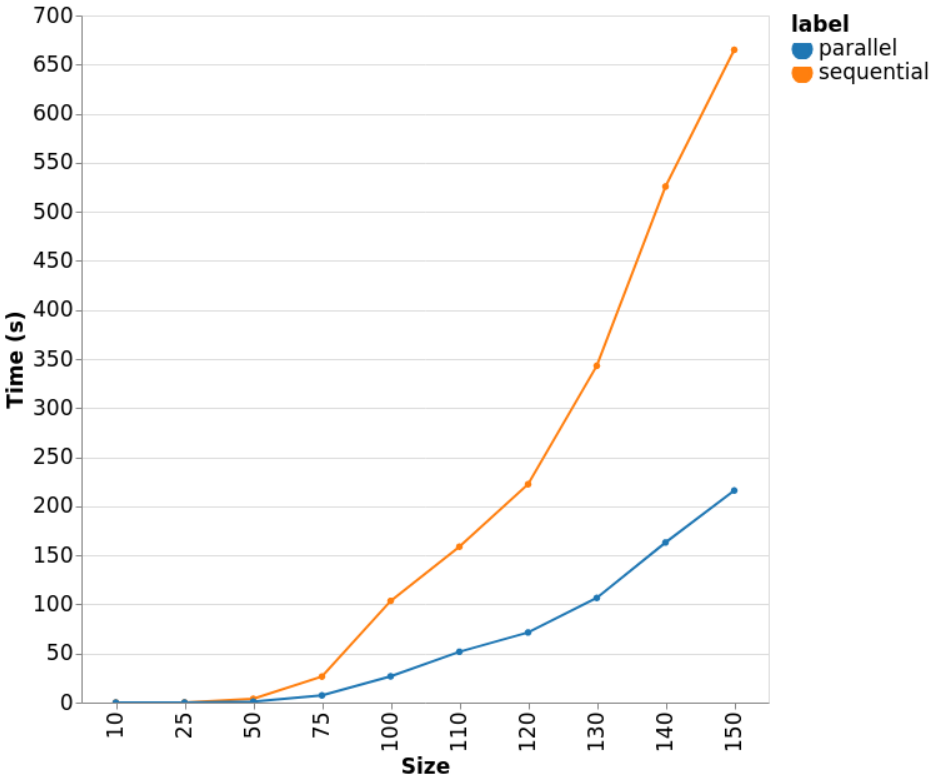

Figure 1: Evolution of the computation time of the algorithm with increasing size of the square matrix

The results regarding rectangular matrices are represented in tables 1 and 2 below for the sequential and parallel versions of the algorithm, respectively.

m \ n	5	10	25	35	40	50	55	60	70	80	100
10	0.0004										
25	0.004	0.0061									
50	0.0004	0.0024	0.4441	0.5873	1.1227						
75	0.0002	0.0025	0.1281	0.7519	1.4389	3.6474	5.271	8.3741	19.024		
110	0.0002	0.0027	0.1255	0.6259	1.3311	3.5418	6.2342	7.8828	23.855	38.007	97.618
120	0.0002	0.0027	0.1324	0.6339	1.2463	3.6110	6.5319	8.3697	18.575	30.923	93.295
130	0.0002	0.0027	0.1491	0.5817	1.1523	3.3454	6.0099	9.6198	17.968	33.117	93.715

Table 1: Computation time in seconds for the sequential algorithm for a matrix of size $(m \times n)$

m \ n	5	10	25	35	40	50	55	60	70	80	100
10	0.0027										
25	0.0017	0.0058									
50	0.0005	0.0029	0.0255	0.2281	0.4763						
75	0.0006	0.0028	0.0812	0.2212	0.3943	1.2234	1.7537	2.4294	5.3193		
110	0.0006	0.0030	0.0921	0.2284	0.4420	1.0100	2.0090	2.5460	5.3643	10.5073	30.1976
120	0.0009	0.0027	0.0783	0.2943	0.4237	1.1633	2.0022	2.7569	5.8660	10.5432	28.7099
130	0.0007	0.0034	0.0812	0.2597	0.5134	1.3083	1.8921	3.0915	5.2462	9.8610	32.9608

Table 2: Computation time in seconds for the parallel algorithm for a matrix of size $(m \times n)$

We first remark that the algorithm converges faster for rectangular matrices than square matrices. This is mainly due to the fact that when considering a rectangular matrix of size $(m \times n)$, the number of singular values is equal to the minimum between m and n (in our case n). For example, a (130×100) matrix has 100 singular values. We also observe that the time needed to compute the singular values for this matrix is significantly close, in the order of 2 seconds, to the time needed to compute the singular value for a (100×100) matrix.

If we now focus on the differences between the sequential and parallel versions of the algorithm, we can draw similar conclusions to those in the square matrix case. When $\texttt{min}(m, n) \leq 10$, the sequential algorithm achieves better performance, but when the size increase above 10, the parallel algorithm performs much better and achieves at least linear speedup for all matrix sizes.

From these two analyses we have empirically deduced that the parallelization of our algorithm has a huge impact on the computation time, especially as the matrix size grows. In practice, SVD is performed on large matrices, typically which have $m >> n$. Clearly, in a practical case such as this one, using our parallel algorithm is vastly favorable to a sequential alternative.

5 Conclusion

In this report we have discussed the *Singular Value Decomposition*, its importance, and how to compute it using the QR decomposition with householder reflections. We take a slightly unorthodox approach, as we combine a typical householder QR with an SVD algorithm designed to obtain only the singular values of a given matrix. We chose this path since we wanted to taste both the methods used for such a decomposition in practice, and because we wanted to obtain the exact (or closely approximated) eigenvalues of a matrix.

In the end, our results clearly demonstrate that our algorithm using `OpenMP` successfully parallelizes the two algorithms utilized, and achieves at least linear speedup in time complexity. Overall this project was incredibly instructive from a mathematical point of view, and helped both of us understand the practical and algorithmic points of view regarding SVD.

References

[1] Press, Teukolsky, Bethe, Vetterling, Flannery. Numerical Recipes, the Art of Scientific Computing. Third Edition. Cambridge University Press, 2007.

[2] Cleve Moler. Numerical Computing with MATLAB https://www.mathworks.com/matlabcentral/fileexchange/37976-numerical-computing-with-matlab, MATLAB Central File Exchange. Retrieved February 11, 2020.

[3] Householder QR algorithm. Rosetta Code. https://rosettacode.org/wiki/QR_decomposition

[4] The Householder Transformation. Harvey Mudd College. http://fourier.eng.hmc.edu/e176/lectures/NM/node10.html

[5] L. Vandenberghe. UCLA ECE133A Lecture Notes. http://www.seas.ucla.edu/~vandenbe/133A/lectures/qr.pdf. 2019.