Parallel Singular Value Decomposition

Introduction

Singular Value Decomposition

The singular value decomposition of a matrix A is the factorization of A into the product of three matrices A = UDV^T where the columns of U and V are orthonormal and the matrix D is diagonal with positive real entries. The SVD is useful in many tasks. Here we mention one example. In many applications, the data matrix A is close to a matrix of low rank and it is useful to find a low rank matrix which is a good approximation to the data matrix .

Jacobi method

Jacobi algorithm was initially proposed by Jacobi C.G.J and is used to compute eigenvalues and eigenvectors. Based on Jacobi's algorithm, a fast and stable method was proposed by Golub-Kahan to decompose a matrix, rectangular or square, in its sub-matrices, as shown in

$$[U, S, V^T] = \text{Jacobi\_SVD}\,(M, \varepsilon)$$

where *M* shows sensitivity encoding matrix, *ε* is the tolerance level for SVD algorithm to converge. *U* and *V* are complex unitary matrices which contain left-singular and right-singular vectors of matrix *M,* respectively, whereas *S* is a diagonal matrix containing the positive singular values of *M,* as represented in

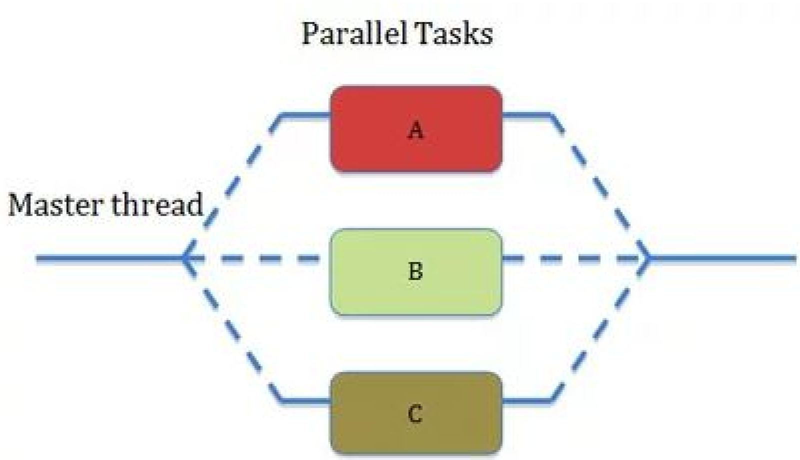
$$S = \begin{bmatrix} \sigma_x & 0 & \cdots & 0 \\ 0 & \sigma_{x-1} & 0 & 0 \\ \vdots & 0 & \ddots & \vdots \\ 0 & 0 & \ldots & \sigma_1 \end{bmatrix}$$

The diagonal values of *S* can be sorted in a descending order as:

$$\sigma_x > \sigma_{x-1} > \sigma_{x-2} > \cdots > \sigma_1$$

<u>OpenMP</u>

It is a directive based parallel programming model. OpenMP program is essentially a sequential program augmented with compiler directives to specify parallelism.



Jacobi SVD Algorithm

The pseudo-code of Jacobi SVD algorithm is given below. The matrix *U* in the algorithm is initialized by an input matrix *M,* while an initial guess of the desired eigenvectors (*V*) is taken as identity matrix.

repeat

  for all i<j

$$\alpha = \sum_{k=1}^{n} U_{ki}^2$$

$$\beta = \sum_{k=1}^{n} U_{kj}^2$$

$$\gamma = \sum_{k=1}^{n} U_{ki}\, U_{kj}$$

$$\zeta = \frac{(\beta - \alpha)}{2\gamma} \qquad \text{(Compute Jacobi rotation)}$$

$$t = signum \frac{(\zeta)}{(|\zeta| + \sqrt{1+\zeta^2})}$$

$$c = \frac{1}{\sqrt{1+t^2}}$$

$$s = ct$$

for all k<n (update matrix U)

  $t = U_{ki}$

  $U_{ki} = ct - sU_{kj}$

  $U_{kj} = st + cU_{kj}$

endfor

for all k < n (update matrix *V*)

  $t = Vki$

  $Vki = ct - s\, Vkj$

  $Vkj = st + c\, Vkj$

endfor

  endfor

until all $\frac{|c|}{\sqrt{\alpha\beta}} \leq \epsilon$

This algorithm repeatedly selects pairs of *i* < *j* from rows and columns of matrix MM^T. *N*-dimensional linear product space R_{kl} is given below. Jacobi rotation is applied on the 2D linear subspace of R_{kl}. The Jacobi rotation process is repeated until the convergence criterion ε is achieved. Orthogonal updates are applied during each iteration in Jacobi algorithm to make the matrix *M* more diagonal than it was in the previous iteration. As a result of the diagonal update, off-diagonal elements are small enough and can be replaced by zero where R_{kl} is an identity matrix except for *c* placed in the diagonal and *s* which are symmetrically placed off the diagonal. The entries of *c* and *s* can be computed by the algorithm. The singular values (*S*) are implicitly generated at convergence, and the right (*V*) and left (*U*) singular vectors are recovered by multiplying all the Jacobi rotations together.

The pseudo-inverse of matrix *M* with Jacobi SVD can be calculated using Eq. (10). The non-zero diagonal elements of *S* are inverted and multiplied with *V* and U^T to get the inverse of the rectangular matrix, *M*:

$$M^{-1} = V \times S^{-1} \times U^T$$

Observations

Following observations were made about the time taken after running the code on 300x300 and 400x400 matrix on a quadcore system:

	300x300	400x400
Serial SVD	4775.91 ms	12195.5 ms
Parallel SVD (3 threads)	2163.45 ms	5635.19 ms
Parallel SVD (4 threads)	1662.23 ms	4456.96 ms
Parallel SVD (8 threads)	2138.15 ms	5218.47 ms
Parallel SVD (16 threads)	1901.12 ms	4892.79 ms

Conclusion

Jacobi SVD algorithm is numerically stable and provides a good solution for large matrices to compute SVD . Jacobi SVD algorithm is computationally fast algorithm and is well-suited for implementation on parallel processors.

Team Members

Pranjali Ingole **(2018201029)**

Kratika Kothari **(2018201060)**

Shafiya Naaz **(2018201062)**