Improved Hestenes SVD Parallel Computing in Distributed System

Li Li, Shuai Chang

I. Introduction

Singular value decomposition (SVD) of matrix is an important and familiar problem in mathematics, science and engineering due to its generality and vast application fields [7]. However, SVD has been the bottleneck for many applicaitons, since the input size in most cases is extremely large (e.g. term-document matrix used in LSI as a input, which contains billion of rows and million of columns). Given a extremely large input, execution time could be intolerablely long though the running time complexity for SVD itself is polynomial [6], $O(m^2n + n^3)$, to be accurate. In this scenario, parallelized SVD is in demand for shortening the execution time. Yet, due to the nasty serial SVD algorithms, where calculations always depend on other components in the algorithm, the parallel version SVD hasn't been successfully established and implemented as far as we know.

In our project, we investigated a series of proposed parallel algorithms for SVD in the literature. And found the one in [9], which using a improved implementation of Hestenes method on a Householder transfermation and QR factorization based serial SVD algorithm, is a practical way of doing this. We decide to implement this algorithm on the HPC server and test its performance.

The rest of the report is formed in such order:

- Mathematically background and applications of SVD is introduced in section II.
- The bottleneck of current algorithms in literature and our motivation is mentioned in section III.
- Algorithms, serial version followed by parallelized version and improved version, are investiaged in detail in section IV.
- Our implementation is presented in section V.
- Experimental Results are provided in section VI.
- Analysis of the results are discussed in section VII.
- At last, section VIII concludes our project and report.

II. Theory of SVD

A. What is SVD

1) Theoretical Definition: The objective of SVD is to decompose a matrix A with rank r and dimension $m \times n$, into three matrices U with dimension $m \times k$, S with dimension $k \times k$ and V with dimension $k \times n$, such that $A = U \times S \times V^T$, where $k = \min(m, n)$ [4], as illustrated in figure 1. This decomposition can be done over any matirx A with any dimension or other mathematical properites. The output matrix S is diagonal matrix contains k singular values of A while matrices U and V are unitary matrices both contains a different set of singular vectors.

2) Geometic Meaning: Starting from the equation $A = U \times S \times V^T$, multiple it with its transpose, we have equation 1.

$$\begin{aligned} A_T \times A &= V \times S^T \times S \times V^T \\ A \times V &= U \times S \end{aligned} \tag{1}$$

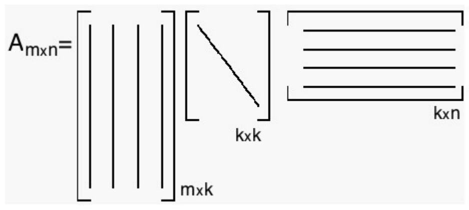


Fig. 1. Matices' relation in SVD

Recall the definition of eigenvector and eigenvalue,

- V can be seen as a set of eigenvectors of $A^T \times A$.
- U can be seen as a set of eigenvectors of $A \times A^T$.
- S contains a list of singular values which are also eigenvalues for both $A^T \times A$ and $A \times A^T$ that control the mapping on U and V

Now, the geometry interpretation of SVD is clear: a rotation, a scaling and another rotation, as shown in figure 2

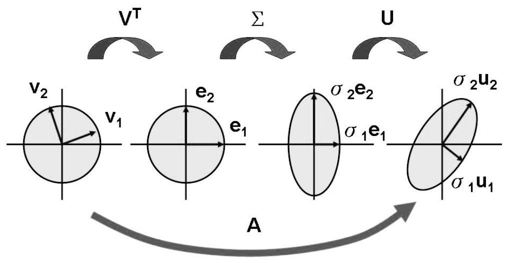


Fig. 2. Geometry Interpretation of SVD

B. Applications of SVD

Principal component analysis (PCA) has long been intensively adopted in large dataset analysis as it effectively cleans out redundant data and simplifies the system. SVD is a major method that enables the functionality of PCA [10], which is to say, whenever PCA is working, SVD is supporting. This exhibits the vast usage of SVD.

Another application of SVD is Latent Semantic Indexing (LSI). LSI is directly built on SVD by calling it to transfer the mapping between a text and word into the mapping between a text and certain combination of words [2]. From the set of distinct words to distinct set of certain combination of words is enssentially a change of basis, just as what SVD does. Search engines, like google or bing, are built on test similarity, where LSI is believed being able to offer better accuracy than co-occurrence indexing. In conclusion, SVD is widely used in data analysis.

III. Bottleneck and Motivation

The running time complexity of SVD is $O(m^2n + n^3)$, where m is the number of rows and n is the number of columns of the input matrix. It seems polynomial, cubic to be accurate. But in practise, m and n could be extremely large. Take its applicaiton on LSI for example, the input matrix always has billion of rows and million of columns. At this point, even a cubic time complexity results in inpractical execution time.

As the time complexity of SVD is inherent to its mathematical properties that is barely improvable, we consider shortening the execution time by improving the implementation method and using parallel programming techniques.

IV. Methodology: Algorithms for SVD

A. Serial Algorithm

There have several different methods on how to implement SVD in a serial fashion. The one to be discussed in detail here is developed in [3], [5], [1] and [4].

The algorithm goes in two steps: Householder transformation and QR factorization.

1) Step 1: Householder transformation: Householder transformation reduces the input matrix A to its bidiagonal form B, as in equation 2:

$$B = \tilde{U}^T A \tilde{V} = \begin{bmatrix} s_0 & e_0 & & & \\ & s_1 & e_1 & 0 & \\ & & \ddots & \ddots & \\ & 0 & & s_{p-1} & e_{p-1} \\ & & & & s_p \end{bmatrix} \tag{2}$$

where,:

$$\begin{aligned} \tilde{U} &= U_0 U_1 \ddots U_{kk-1}, kk = \min(n, m-1) \\ \tilde{V} &= V_0 V_1 \ddots V_{ll-1}, ll = \min(m, n-2) \end{aligned} \tag{3}$$

In equation 3, each $U_j, (j = 0, 1, \ddots, kk-1)$ turns elements in column j below the diagonal to be zero, while each $V_j, (j = 0, 1, \ddots, ll-1)$ turns elements in row j right to the diagonal to be zero.

Additionally, for each V_j, to avoid overflow and variation merging, it is represented as $I - \rho V_j V_j^T$, where ρ is a normalization factor.

As $V_j = (v_0, v_1, \ddots, v_{n-1})^T$ is a column vector, we have:

$$AV_j = A - \rho A V_j V_j^T = A - W V_j^T \tag{4}$$

where,

$$W = \rho A V_j = \rho(\sum_{i=0}^{n-1} v_i a_{0i}, \sum_{i=0}^{n-1} v_i a_{1i}, \ddots, \sum_{i=0}^{n-1} v_i a_{(m-1),i})^T \tag{5}$$

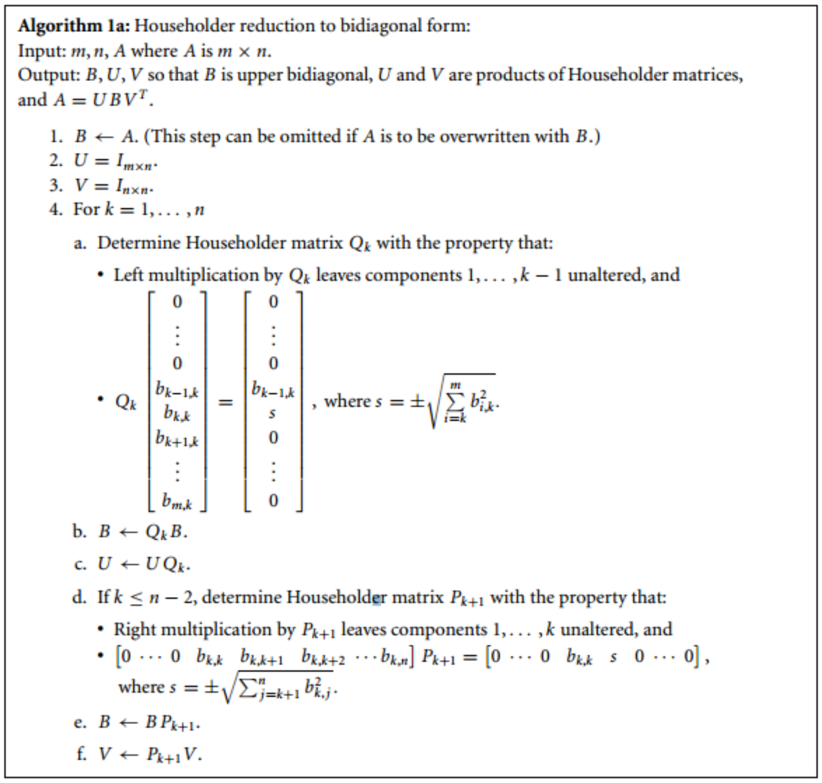
Psuedocode for this step is shown in figure 3.

2) Step 2: QR factorization: QR factorization is used then iteratively to calculate all singular values via a series of rotations on A's bidiagonal matrix B.

In each iteration, do

$$B' = U_{p-1,p}^T \ddots U_{1,2^T U_{0,1}^T B V_{0,1} V_{1,2} \ddots V_{m-2,m-1}} \tag{6}$$

where $U_{j,j+1}^T$ turns a element in column j below the main diagonal to zero, while, makes a non-zeros element on the right of the secondary diagonal in row j; at the mean time, $V_{j,j+1}$ turns a element in row $j-1$ on the right of the secondary diagonal to be zero, while makes a elment below the main diagonal on column j to be non-zeros, in B. After this, B' remains to be bidiagonal matrix after each



Algorithm 1a: Householder reduction to bidiagonal form:
Input: m, n, A where A is $m \times n$.
Output: B, U, V so that B is upper bidiagonal, U and V are products of Householder matrices, and $A = UBV^T$.

```
1. $B \leftarrow A$. (This step can be omitted if $A$ is to be overwritten with $B$.)
2. $U = I_{m \times n}$.
3. $V = I_{n \times n}$.
4. For $k = 1, \ldots, n$
   a. Determine Householder matrix $Q_k$ with the property that:
      • Left multiplication by $Q_k$ leaves components $1, \ldots, k-1$ unaltered, and
      • $Q_k \begin{bmatrix} 0 \\ \vdots \\ 0 \\ b_{k-1,k} \\ b_{k,k} \\ b_{k+1,k} \\ \vdots \\ b_{m,k} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ b_{k-1,k} \\ s \\ 0 \\ \vdots \\ 0 \end{bmatrix}$, where $s = \pm\sqrt{\sum_{i=k}^{m} b_{i,k}^2}$.
   b. $B \leftarrow Q_k B$.
   c. $U \leftarrow U Q_k$.
   d. If $k \leq n-2$, determine Householder matrix $P_{k+1}$ with the property that:
      • Right multiplication by $P_{k+1}$ leaves components $1, \ldots, k$ unaltered, and
      • $[0 \cdots 0 \;\; b_{k,k} \;\; b_{k,k+1} \;\; b_{k,k+2} \cdots b_{k,n}] P_{k+1} = [0 \cdots 0 \;\; b_{k,k} \;\; s \;\; 0 \cdots 0]$,
        where $s = \pm\sqrt{\sum_{j=k+1}^{n} b_{k,j}^2}$.
   e. $B \leftarrow B P_{k+1}$.
   f. $V \leftarrow P_{k+1} V$.
```

Fig. 3. Serial algorithms for SVD, a householder approach

iteration. But as more and more iteration has been done, B' will converge to diagonal matrix, where all elements on the diagonal are singular values.

An example of calucate shift value u for $V_{0,1}$ is shown in equation 7.

$$\begin{aligned} b &= [(s_{p-1} + s_p)(s_{p-1} - s_p) + e_{p-1}^2]/2 \\ c &= (s_p e_{p-1})^2 \\ d &= sign(b)\sqrt{b^2 + c} \\ u &= s_p^2 - \frac{c}{b+d} \end{aligned} \tag{7}$$

After all iteration, sort all singular value desendingly. Note that in practise, according to given requirement of accuracy, for any element e_j on the secondary diagonal, it can be regarded as zero if satisfies $|e_j| \leq \varepsilon(|s_{j+1} + |s_j|)$; for any element s_j on the main diagonal, it can be treated as zero if satisfies $|s_j| \leq \varepsilon(|e_{j-1}| + |e_j|)$

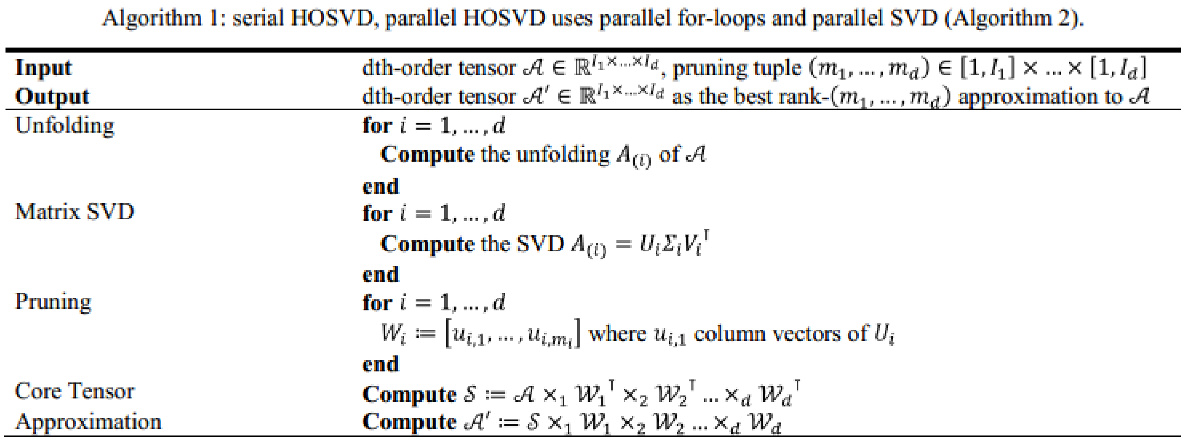
3) Sum up: Despite detailed calcuations given above, steps for the serial algorithm using Householder transformation and QR factorization can be sum up in figure 4. Divide the step into several calculation groups so that we can consider the possibility of parallelization on each grouph is stated in figure 5. Our parallel algorithm will built on this scratch.

B. Parallelization: Hestenes Method

Effective parallel algorithm for SVD is a hard problem. As far as I know, no literature solved it elegantly. Edward Chang, Director, Google Research, claimed they sucessfully implemented a parallel version of svd, but since Google uses SVD for LSI so that the input matrix is sparse, which still

- Starting from the beginning with a matrix A, we want to derive UWV^T
- Using Householder transformations: $A = PBS$
- Using QR factorization: $B = Q_0^T Q_1^T \ddots Q_w^T W Q_w \ddots Q_1 Q_0$
- Subsituting B into $A = PBS$: $A = PQ_0^T Q_1^T \ddots Q_w^T W Q_w \ddots Q_1 Q_0 S$
- With U being derived from: $U = PQ_0^T Q_1^T \ddots Q_w^T$
- And V_T being derived from: $V = Q_w \ddots Q_1 Q_0 S$
- Which results in the final SVD: $A = UWV^T$

Fig. 4. Steps sum up of Serial SVD


Algorithm 1: serial HOSVD, parallel HOSVD uses parallel for-loops and parallel SVD (Algorithm 2).

Input	dth-order tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times \ldots \times I_d}$, pruning tuple $(m_1, \ldots, m_d) \in [1, I_1] \times \ldots \times [1, I_d]$
Output	dth-order tensor $\mathcal{A}' \in \mathbb{R}^{I_1 \times \ldots \times I_d}$ as the best rank-$(m_1, \ldots, m_d)$ approximation to $\mathcal{A}$

Unfolding	**for** $i = 1, \ldots, d$
	 Compute the unfolding $A_{(i)}$ of $\mathcal{A}$
	 end

Matrix SVD	**for** $i = 1, \ldots, d$
	 Compute the SVD $A_{(i)} = U_i \Sigma_i V_i^{\mathsf{T}}$
	 end

Pruning	**for** $i = 1, \ldots, d$
	 $W_i := [u_{i,1}, \ldots, u_{i,m_i}]$ where $u_{i,1}$ column vectors of U_i
	 end

Core Tensor	**Compute** $\mathcal{S} := \mathcal{A} \times_1 W_1^{\mathsf{T}} \times_2 W_2^{\mathsf{T}} \ldots \times_d W_d^{\mathsf{T}}$
Approximation	**Compute** $\mathcal{A}' := \mathcal{S} \times_1 W_1 \times_2 W_2 \ldots \times_d W_d$

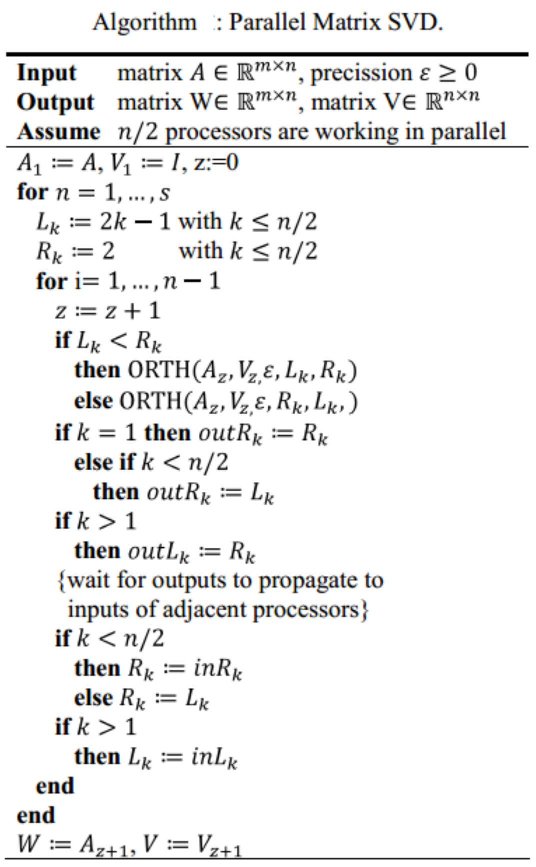
Fig. 5. Serial algorithm in computation groups for parallelization

left general parallel SVD unsolved. The difficulty is mostly due to the dependence among matrices calculated in step 1. The reduction to bidiagonal matrices is so hard to be parallelized since in each iteration, the input is the output of the last iteration, at the same time, values updated in each iteration depend on value of the input matrix. In this scenario, adding MPI or openMP simple into any make the algorithm invalid since you dont know if a value you used to update you current element has been updated in this iteration or not.

Given the serial SVD implemented using Householder transfermation and QR factorizaiton as we discussed above, Hestenes method is beleived to be a working way to parallelize it [8]. The idea of parallelize SVD based on algorithm 5 is as follow:

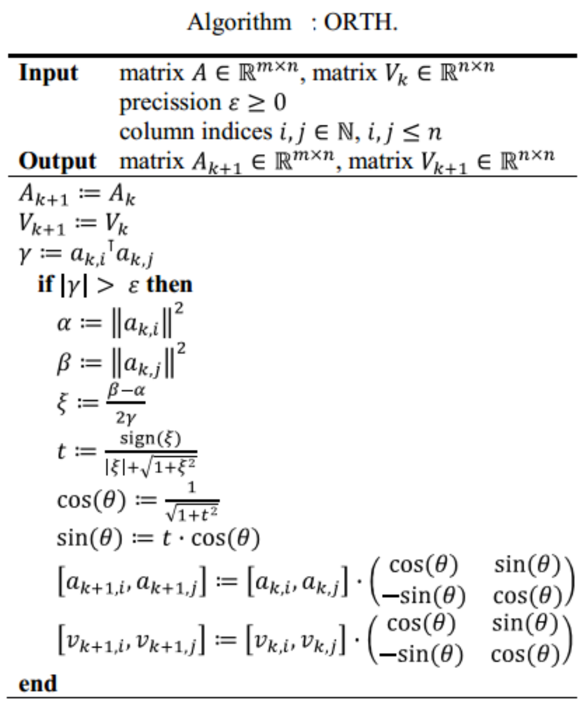
- Unfolding: this part can be parallelzied since A_i is independent from each other.
- Matrix SVD: This part can also be parallelzied thanks to the independence among A_i.
- Pruning: Pruning on each column vector U_i can be done in parallel.
- Core Tensor: This part is just a matrix multiplicaiton, so can be parallelized.
- Approximation: same as Core Tensor.

In the above discussion, one can see that, parallelizable part in SVD are essentially matrix multiplicatioin, through here number of matrices to be muliplied could be large. In other words, the parallelization of SVD is built on a effective matrix SVD algorithm using Hestenes method in figure 6.



Algorithm : Parallel Matrix SVD.

```
Input    matrix $A \in \mathbb{R}^{m\times n}$, precission $\varepsilon \geq 0$
Output   matrix W$\in \mathbb{R}^{m\times n}$, matrix V$\in \mathbb{R}^{n\times n}$
Assume   $n/2$ processors are working in parallel
$A_1 := A, V_1 := I$, z:=0
for $n = 1, \dots, s$
  $L_k := 2k - 1$ with $k \leq n/2$
  $R_k := 2$      with $k \leq n/2$
  for i= $1, \dots, n-1$
    $z := z + 1$
    if $L_k < R_k$
      then ORTH$(A_z, V_z, \varepsilon, L_k, R_k)$
      else ORTH$(A_z, V_z, \varepsilon, R_k, L_k,)$
    if $k = 1$ then $outR_k := R_k$
      else if $k < n/2$
        then $outR_k := L_k$
    if $k > 1$
      then $outL_k := R_k$
    {wait for outputs to propagate to
     inputs of adjacent processors}
    if $k < n/2$
      then $R_k := inR_k$
      else $R_k := L_k$
    if $k > 1$
      then $L_k := inL_k$
  end
end
$W := A_{z+1}, V := V_{z+1}$
```



Algorithm : ORTH.

```
Input    matrix $A \in \mathbb{R}^{m\times n}$, matrix $V_k \in \mathbb{R}^{n\times n}$
         precission $\varepsilon \geq 0$
         column indices $i, j \in \mathbb{N}$, $i, j \leq n$
Output   matrix $A_{k+1} \in \mathbb{R}^{m\times n}$, matrix $V_{k+1} \in \mathbb{R}^{n\times n}$
$A_{k+1} := A_k$
$V_{k+1} := V_k$
$\gamma := {a_{k,i}}^{\intercal} a_{k,j}$
  if $|\gamma| > \varepsilon$ then
    $\alpha := \|a_{k,i}\|^2$
    $\beta := \|a_{k,j}\|^2$
    $\xi := \frac{\beta - \alpha}{2\gamma}$
    $t := \frac{\text{sign}(\xi)}{|\xi| + \sqrt{1+\xi^2}}$
    $\cos(\theta) := \frac{1}{\sqrt{1+t^2}}$
    $\sin(\theta) := t \cdot \cos(\theta)$
    $[a_{k+1,i}, a_{k+1,j}] := [a_{k,i}, a_{k,j}] \cdot \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}$
    $[v_{k+1,i}, v_{k+1,j}] := [v_{k,i}, v_{k,j}] \cdot \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}$
end
```

Fig. 6. Parallel matrix SVD using Hentenes Method

C. Improved Implementation of Hestenes Method

Note the fact that W_{i-1} is a column orthogonal matrix (it is diagonal). So we can rewrite V_i as follow [9]:

$$V_k = V_{k-1} \prod_{w=1}^{\#sweeps} \left(\prod_{i=1}^{n-1} \prod_{j=i+1}^{n} V_{ij} \right). \tag{8}$$

From equation 8, we can use the orthogonal matrix from last iteration, which makes A_i near orthogonal, and then use Hestenes method. This way iteration times is reduced thus accelerates the algorithm.

V. Our Implementation

A. Software and Hardward Environment

1) MPI: In order to take advantage of a distributed computing system to solve the computation intensive application, which for this project, is the SVD computation, a parallel programming paradigm should be selected. For simplicity and based on our expertise, we selected MPI (Message Passing Interface) to implement the parallel processing of this problem. MPI can send and receive messages among different processes, work separately and gather together to produce result when all the parathion work is done.

2) HPC: The experiment environment we are using for this project is the HPC system from the University of Arizona. With this system, we are able to configure the number of nodes and memory that we need. Different nodes will communicate with each other with the MPI. Based on our size of test case and the constraint of HPC system, we decided to swipe from 1 nodes to 32 nodes (1, 4, 8, 16, 32) and the memory for each node is configured to be 4GB.

B. Code Structure

In order to achieve the best performance speed up, the serial code should be parallelized as much as possible. At the same time, the correctness of the computation after parallelized must be maintained as the serial implementation. Therefore, the place where the parallel computing can be safely performed must be an operation on a separated section of array, which does not depend on other parts of array that are being updated by other process.

Building on the serial version of the implementation, the section that we found would be the most appropriate to parallel execution is when the algorithm is updating ąřaąś mentioned in step one, where the Household Transform is being performed.

In terms of implementation, the source code is attached as appendix. The steps of implementation are listed are followings:

- The MPI is initialized in main.cpp
- The master process (Rank = 0) reads the input matrix from prepared input files.
- Master process initializes variables such as dimension information.
- Master sends the variables to slave processes, the slave processes received the message and allocate spaces for matrix the SVD will be performed on.
- Each process, including the masterąŕs process then ready to call the SVD compute function $dluav()$. The numNodes and myRank are passed as an identification for each process inside the function.
- The part of the array (input matrix) that an individual process will work on is determined by its rank, numRow, numCol and numNodes. The offset and length for each process is calculated as:

$$\begin{aligned} offset &= myRank \times numCols \times \frac{numRows}{numNodes} \\ length &= \frac{numRows}{numNodes} \times numCols \end{aligned} \tag{9}$$

- When the Household transform is complete, the partitions of array a need to be send back from slave processes to the masterąŕs process. The master updates the array a, and each process then uses this updated a to calculated output matrix u and v in parallel

VI. Experimental Results

The experiments we conducted are on randomly generated matrix with dimension 1024 by 1024, and 2048 by 2048. The output matrices are compared with the output of R and Matlab on SVD computation. The following matrices are the output of the computation, the output matrices are written into output files whose name can be specified in the command line arguments, the followings are the output matrices that are compared with R and Matlab:

- MarkSV: equal to $S \times V$, for personal usage
- MarkUS: equal to $U \times S$, for personal usage
- outLeftMatrix: matrix U
- outRightMatrix: matrix V
- reducedRightMatrix: matrix after customized dimension reduction, equal to $U_k \times S_k \times V_k^T$.
- simMatrix: similarity matrix for each row of input matrix, for personal usage.
- singVec: singular values

The correctness of the improved SVD algorithm implementation is verified by the output from R and Matlab for both serial version and the parallel implementation.

In terms of performance evaluation, we swiped number of nodes over two input matrices (1024 and 2048 double data type square matrices). Instead of doing profile for the whole program, we inserted time elapsed calculation, where we recorded the starting time and ending time in milliseconds within Rank0 before and after the Household Transform. The reason for this approach is that most of the file writes to output files after the result is computed is not related to the computation, in this project we focused on improvement of computation not the file IO performance.
Raw data are recorded in table I and table II respectively.
Curves describing execution time vs. number of cores are plotted in figure 7 and figure 8, respectively.

	1	4	8	16	32
JobID	54991	54992	54993	54994	54995
ExeTime(ms)	1794	1949	1970	3997	6573
Speedup Ratio	1	1.086399	1.098105	2.227982	3.66388

TABLE I

HPC EXPERIMENT RESULT FOR 1024 BY 1024 MATRIX SVD COMPUTATION OVER VARIOUS NUMBER OF NODES

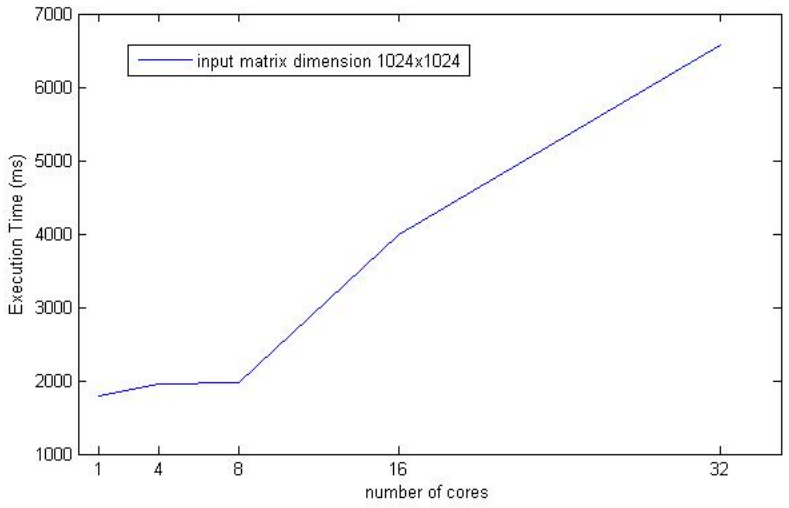


Fig. 7. Graph that shows the relationship between execution time over number of nodes for the 1024 by 1024 input matrices

VII. Analysis

To our surprise, the performance, measured in elapsed time, decreases when the number of core increases. However, this result is consistent with the circumstances happened in the programming homework throughout the semester _ the more nodes, the longer execution time for MPI implementation although the number of iterations is significant less. Since the output is correct in terms of functionality and the behavior matches that of the homework implementation, we are confident that the algorithm is successfully implemented in parallel and the distributed system indeed helped compute SVD. Some

	1	4	8	16	32
JobID	54997	54998	54999	55007	55008
ExeTime(ms)	13267	17716	13212	27306	43476
Speedup Ratio	1	1.335343	0.995854	2.058189	3.277003

TABLE II

HPC EXPERIMENT RESULT FOR 2048 BY 2048 MATRIX SVD COMPUTATION OVER VARIOUS NUMBER OF NODES

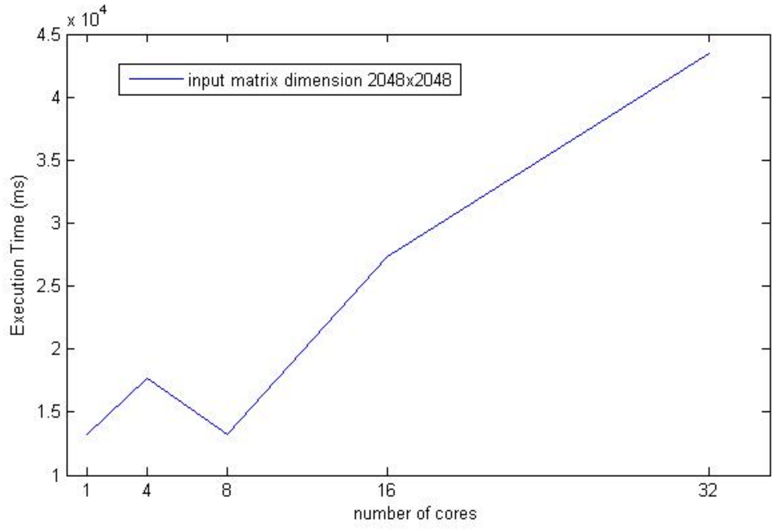


Fig. 8. Graph that shows the relationship between execution time over number of nodes for the 2048 by 2048 input matrices

other thoughts regarding this behavior could be that the MPI spends more time on communication for the selected test cases than computation. In MPI framework, we have to perform message passing among all processes, when number of processes increases, the time spent on communication also increase accordingly, therefore, it is reasonable to find the result we are getting.

VIII. Conclusions

In this project, we have investigated both serial and parallel version of SVD implementations. We compared the performance of the serial version discussed in section IV and the parallel version discussed in section IV.

We understood the difficuties of implementing serial SVD in a parallel version.

We implemented a parallel version SVD, which in theory should have good performance. Yet our results state that the performance is not as expected: parallel version was beaten by serial version in most of the cases. However, the performance also varies case by case according to the number of cores used.

The future work for this distributed system SVD algorithm implementation includes implementing the same algorithm with OpenMP, which instead of passing message around, uses shared memory to do computation. For the Household Transform part of SVD algorithm, OpenMP is a perfect candidates for parallelism. Without all the communication overhead, we are expecting the execution time will significantly be reduced while the number of cores are increasing.

Thanks to the project, we have a better sense on how complicate the parallel system is and how to crack it.

References

[1] Peter A Businger and Gene H Golub. Algorithm 358: singular value decomposition of a complex matrix [f1, 4, 5]. *Communications of the ACM*, 12(10):564–565, 1969.

[2] Scott C. Deerwester, Susan T Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6):391–407, 1990.

[3] Gene Golub and William Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial & Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224, 1965.

[4] Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5):403–420, 1970.

[5] Gene Howard Golub. Least squares, singular values and matrix approximations. *Aplikace matematiky*, 13(1):44–51, 1968.

[6] Thomas Hofmann. Probabilistic latent semantic analysis. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 289–296. Morgan Kaufmann Publishers Inc., 1999.

[7] Virginia Klema and Alan Laub. The singular value decomposition: Its computation and some applications. *Automatic Control, IEEE Transactions on*, 25(2):164–176, 1980.

[8] Philipp Shah, Christoph Wieser, and François Bry. Parallel higher-order svd for tag-recommendations.

[9] Zhang Shihui, Kong Lingfu, and Feng Liang. An improved hestenes svd method and its parallel computing and application in parallel robot [j]. *Journal of Computer Research and Development*, 4:023, 2008.

[10] Michael Wall, Andreas Rechtsteiner, and Luis Rocha. Singular value decomposition and principal component analysis. *A practical approach to microarray data analysis*, pages 91–109, 2003.