A Parallel Implementation of Lanczos Algorithm in SVD Computation

SRIKANTH KALLURKAR

Abstract

This work presents a parallel implementation of Lanczos algorithm for SVD Computation using MPI. We investigate execution time as a function of matrix sparsity, matrix size, and the number of processes. We observe speedup and slowdown for parallelized Lanczos algorithm. Reasons for the speedups and the slowdowns are discussed in the report.

I. INTRODUCTION

Singular Value Decomposition (SVD) is an important classical technique which is used for many purposes [8]. Deerwester et al [5] described a new method for automatic indexing and retrieval using SVD wherein a large term by document matrix was decomposed into a set of 100 orthogonal factors from which the original matrix was approximated by linear combinations. The method was called Latent Semantic Indexing (LSI). In addition to a significant problem dimensionality reduction it was believed that the LSI had the potential to improve precision and recall. Studies [11] have shown that LSI based retrieval has not produced conclusively better results than vector-space modeled retrieval. Recently, however, there is a renewed interest in SVD based approach to Information Retrieval and Text Mining [4], [6].

The rapid increase in digital and online information has presented the task of efficient information processing and management. Standard term statistic based techniques may have potentially reached a limit for effective text retrieval. One of key new emerging areas for text retrieval is language models, that can provide not just structural but also semantic insights about the documents. Exploration of SVD based LSI is therefore a strategic research direction.

One of the problems that is faced during SVD computation is the that for very large matrices the amount of time and memory required is tremendous. This problem is accentuated when dealing with very large matrices that are also very sparse. It is observed that in general, term-document matrices are exactly such matrices. Since one of the advantages of LSI was dimension reduction, time and space constraints to SVD computation is almost detrimental.

In this work we explore the potential of parallelizing components in the SVD computation with the interest of quickening SVD computation. SVD has been implemented in a variety of methods over the year. One of the famous ones has been SVDPACK [3]. This is a suite of SVD computational packages. The code packages differ by type of Lanczos procedure’s implementation. In this work we selected the las2 implementation that computes SVD for a sparse matrix via Eigensystem of $AA^T$ symmetric matrix, for it being the simplest implementations of the Lanczos algorithm. A major portion of the work during SVD computation is performed in the Lanczos algorithm, and hence we explore the amount of speedup possible by parallelizing just the Lanczos step. We present experimental results that show that for very large matrices the speedup is directly proportional to amount of density of the matrices.

II. BACKGROUND

The usefulness of SVD has attracted many attempts for its parallelization. Many of these are based on SVDPACK code available at [10]. Parallel implementations also exist for both shared memory and distributed memory architectures [2], [1]. Amongst other notable attempts, SVD parallelization by Pan et al [9] use separate instructions using CORDIC arithmetic.
III. SVD

In this section we describe the steps involved in SVD computation and in particular the Lanczos step for computing the tridiagonal matrix. SVD is computed on a term-document matrix where the rows contain all unique terms that occurred in a collection of documents, represented in the columns. Thus a value $a_{ij}$ of the matrix tells us the weight of the $\text{term}_i$ in $\text{document}_j$. A simple form of the weight is number of occurrences of the term.

**Theorem A:** If $A$ is a symmetric $n \times n$ matrix, then there exists $n$ eigenpairs $(\lambda_i, v_i), i = 1, ..., n$, so that

1. all $\lambda_i$ are real and $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n$
2. the vectors $v_i$ form an orthogonal set, i.e. $v_i^T v_j = 0, \text{if } i \neq j$ and $v_i^T v_i = 1$
3. $A = V \text{diag}(\lambda_1, ..., \lambda_n)V^T$

**Theorem B:** If $B$ is real matrix then there exists orthogonal matrices $U = [u_1, ..., u_m] \in R^{m \times m}$, and $V = [v_1, ..., v_n] \in R^{n \times n}$ such that

$$U^T BV = \text{diag}(\sigma_1, ..., \sigma_p) \in R^{m \times m}, p = \min(m, n), \sigma_1 \geq ... \geq \sigma_p$$

Denoting $\text{diag}(\sigma_1, ..., \sigma_p)$ by $\Sigma$, we can write SVD as $\Sigma = U^T BV$. From this we can get

$$BB^T = U \text{diag}(\sigma_1^2, ..., \sigma_p^2)U^T$$

and

$$B^T B = V \text{diag}(\sigma_1^2, ..., \sigma_p^2)V^T$$

For SVD computation the following steps are required.

**Input:** $m \times n$ term document matrix $B$

**Output:** first $p \leq \min(m, n)$ Singular triplets($u_i, \sigma_i, v_i$) of $B$

1. Form symmetric matrix $A = BB^T$
2. Apply Lanczos algorithm to compute Tridiagonal matrix $T_k, p \leq k$
3. Apply QL method to obtain diagonal matrix $D$ from $T_k$
4. Compute first $p$ eigenvalues $\lambda_1, ..., \lambda_p$ of $D$
5. Compute first $p$ eigenvectors $x_1, ..., x_p$ of $D$ and approximations of eigenvectors $u_1, ..., u_p$ of $A$
6. if the eigenvalue approximations need improvement increase $k$ and GOTO step 2
   else
   stop

A. Lanczos Step

The Lanczos step computes a tridiagonal matrix. The following steps describe the Lanczos algorithm.

1. Set $\beta_0 = 0, q_0 = 0, b \neq 0$ arbitrary, $q_1 = b / \|b\|
2. Set $i = 1$
3. $v = Aq_i$
4. $\alpha_i = q_i^T v$
5. $v = v - \beta_i q_{i-1} \alpha_i q_i$
6. $\beta_i = \|v\|
7. $q_{n+1} = v / \beta_i$
8. Increment $i$ by 1
9. if $i < k$ goto step 3

IV. PARALLELIZATION

As described earlier, we used the las2 code package to implement a parallel Lanczos method. The input to las2 is a term-document matrix in a sparse format as represented by the Harwell-Boeing sparse matrix representation.
A. Sparse Matrix Representation

The Harwell-Boeing format is the most popular format for text-file exchange of sparse matrix data. The following information, taken from User’s Guide [7] for the Harwell-Boeing Sparse Matrix Collection provides a specification for this format.

Matrix data is held in an 80-column, fixed-length format for portability. Each matrix begins with a multiple line header block, which is followed by two, three, or four data blocks. The header block contains summary information on the storage formats and space requirements. From the header block alone, the user can determine how much space will be required to store the matrix. Information on the size of the representation in lines is given for ease in skipping past unwanted data.

If there are no right-hand-side vectors, the matrix has a four-line header block followed by two or three data blocks containing, in order, the column (or element) start pointers, the row (or variable) indices, and the numerical values. If right-hand sides are present, there is a fifth line in the header block and a fourth data block containing the right-hand side(s). The blocks containing the numerical values and right-hand side(s) are optional. The right-hand side(s) can be present only when the numerical values are present. If right-hand sides are present, then vectors for starting guesses and the solution can also be present; if so, they appear as separate full arrays in the right-hand side block following the right-hand side vector(s).

The first line contains the 72-character title and the 8-character identifier by which the matrix is referenced in our documentation. The second line contains the number of lines for each of the following data blocks as well as the total number of lines, excluding the header block. The third line contains a three character string denoting the matrix type as well as the number of rows, columns (or elements), entries, and, in the case of unassembled matrices, the total number of entries in elemental matrices. The fourth line contains the variable Fortran formats for the following data blocks. The fifth line is present only if there are right-hand sides. It contains a one character string denoting the storage format for the right-hand sides as well as the number of right-hand sides, and the number of row index entries (for the assembled case).

B. System design

Parallelization was achieved via MPI. The root process alone reads the matrix in the sparse format from file. This requires that the root process have as much memory required to read in all the row and column indices and all the non-zero entries. This was done deliberately so as to isolate as much as possible all user and data interactions on the root. The matrix was divided amongst the processors by blocks of columns. i.e for $m$ row by $n$ matrix, each processor should have had $m$ row $n/p$ columns for $p$ processors. Because of sparse nature of the matrices we would be dealing for retrieval purposes, it is entirely possible that such equal division of columns could lead to unbalanced load on the processors. We opted for a load balanced approach, where load would be quantified by the number of non-zero entries. However, assigning equal number of non-zero elements could result in fractional number of columns per processor. Such situations will lead to larger costs as the processes sharing the columns would have communicate to each the other the part of the answers they calculated. To solve this problem, the root first computed the number of non-zero entries per process, then found the nearest endpoint of a column. Thus the number of non-zero values per process was approximated to the optimal non-zero values, and complete columns were assigned to the processes (Table I).

The root then distributed the assigned columns to the processes, including itself. Each process performed the steps of the Lanczos step commensurate to the part of the matrix it has. The number of iterations is determined by the number of elements required for the tridiagonal matrix. Each process has the tridiagonal matrix computed per step of the iteration. To achieve orthogonalization, only the answers of matrix multiplications were gathered at all the processes. The original code used $C$ translations of BLAS rou-
tines. Whenever multiple arrays were required to be transferred, derived data types were used to reduce messaging, along with standard MPI global communication performatives.

V. EXPERIMENTS

It is generally observed that term document matrices are sparse. Thus we created an experimental setup wherein matrices of differing sizes and sparsity were created. This was done so as to simulate real term document matrices, without actually creating one. The matrices were created such that a matrix of and size could have the given percentage of sparsity. The matrix creation process was randomized, with values chosen carefully between 1 and 10. The experiments were conducted in two main parts. One for constant matrix size and different sparsity and, another of different matrix size and different sparsity. Table II provides the sizes and density of the matrices. Density of the matrices is dependent, amongst other factors, on the size of the documents. Larger documents tend to have more number of unique terms. Also, term-document matrices normally are not squares matrices. The growth of the term space is observed to be at least logarithmic and sometimes quadratic. This is because the term growth is limited by the language vocabulary.

Since only the Lanczos step was parallelized we did not measure the time for the entire SVD step. Instead we measured time from the starting of the program (las2), till the call to the Lanczos step, and then added the time spent for the Lanczos step, each time it was called.

A. Parallel hardware

The experiments were conducted on a Beowulf cluster (beowulf.cs.umbc.edu). The cluster has 9 nodes, 10/100 Mbps Ethernet and 1GB switch. The nodes have the following configuration:

- Master, nodes 1-4: Dual PIII, 600MHz, 256Kb cache and 1GB memory
- nodes 5-8: Dual PIII, 1000MHz, 256Kb cache and 2GB memory

<table>
<thead>
<tr>
<th>Number Processors</th>
<th>Nonzero Values Computed,(Actual Assigned)</th>
<th>Number Columns Computed,(Actual Assigned)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>82,(82)</td>
<td>1343,(1343)</td>
</tr>
<tr>
<td>2</td>
<td>41,(39, 43)</td>
<td>671.5,(666, 677)</td>
</tr>
<tr>
<td>4</td>
<td>20.5, (17, 22, 21, 22)</td>
<td>335.75, (337, 329, 335, 342)</td>
</tr>
<tr>
<td>8</td>
<td>10.25,(7, 10, 11, 11, 10, 11, 11, 11)</td>
<td>167.875,(161, 176, 170, 159, 167, 168, 175, 167)</td>
</tr>
</tbody>
</table>

TABLE I

COMPUTED AND ASSIGNED NONZERO VALUES AND COLUMNS

<table>
<thead>
<tr>
<th>Size</th>
<th>Non zero values</th>
<th>% Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000×100,000</td>
<td>4,450,321</td>
<td>0.04</td>
</tr>
<tr>
<td>100,000×100,000</td>
<td>7,452,351</td>
<td>0.07</td>
</tr>
<tr>
<td>100,000×100,000</td>
<td>11,203,051</td>
<td>0.11</td>
</tr>
<tr>
<td>100,000×100,000</td>
<td>19,928,901</td>
<td>0.20</td>
</tr>
<tr>
<td>100,000×125,000</td>
<td>29,928,901</td>
<td>0.30</td>
</tr>
<tr>
<td>100,000×250,000</td>
<td>38,400,549</td>
<td>0.18</td>
</tr>
<tr>
<td>100,000×250,000</td>
<td>49,846,209</td>
<td>0.28</td>
</tr>
</tbody>
</table>

TABLE II

MATRIX SIZES AND DENSITY
The experiments were conducted during minimal usage periods, although no restrictions were applied to other users of the system. The experiments were repeated to obtain the average times of execution. Compilation was done using gcc version 2.96 20000731 (Red Hat Linux 7.1 2.96 – 98) and LAM (version 6.5.4) implementation of MPI was used.

VI. RESULTS

The time and speedup plots are shown in Figures 1, 2 for the matrix of size 100,000X100,000. It is observed that for the matrix of smallest density, there was actually a slowdown with increase in number of processors. But as we increase the density of matrices there is almost a constant decrease in time, or constant gain in speedup. Thus it can be deduced that there needs to be enough work to offset the costs of parallelization. Two main factors affect the timings: (1) cost of distributing data and, (2) Messaging overhead during orthogonalization steps. The first factor is a function of the size of matrix, while the second factor is dependent both on the size of the matrix and number of eigenvectors required.

The figures suggest that by increasing the density of the matrix, we can impart enough load such that in a single processor case more work is performed than in a multiprocessor case where costs of work include parallelization overheads. For the second set of experiments we increased the number of rows disproportionately to the number of columns. The second set of experiments comply with the above mentioned results (Figures 3 and 4). Thus so far we have shown that using basic MPI communication methods there was some speedup.

VII. FUTURE WORK

In this work we presented a parallel implementation of the Lanczos step for SVD computation. It can be used as base for future work on extending the parallelization to other components of SVD computation and to other types of Lanczos algorithm. One of the steps that can be taken is the parallelization of the steps after Lanczos. It may be seen that the size of tridiagonal matrix is so small that parallelization may not achieve any intended speedups. Also the increase is network speeds and throughput is still nowhere close to
the memory speeds. Since most parallelization efforts start with this handicap, speedups is only achievable for large problems. High performance computing will have to address this issue.

VIII. ACKNOWLEDGMENTS

We thank Dr. M Berry\(^1\) for his words of encouragement. Dr J. Kogan\(^2\) for expert advice on computational IR and Dr. Nayakkankuppam\(^3\) for advice on parallelization.

\(^1\)University of Tennessee, Knoxville
\(^2\)UMBC
\(^3\)UMBC
Number of processors

Speedup

Matrix 110,000 X 250,000

Fig. 4. Speedup plot

REFERENCES