Procedia Computer Science

Volume 29, 2014, Pages 1876–1887

ICCS 2014. 14th International Conference on Computational Science

A Parallel Implementation of Singular Value Decomposition for Video-on-Demand Services Design Using Principal Component Analysis

Raul V. Ramirez-Velarde[1], Martin Roderus[2], Carlos Barba-Jimenez[1], and Raul Perez-Cazares[1]

[1] Tecnologico de Monterrey, Monterrey, Nuevo Leon, Mexico
rramirez@itesm.mx,cbarbaji@gmail.com,raul.perez@itesm.mx
[2] Institut fur Informatik. Technische Universitat Munchen, Munich, Germany
roderus@in.tum.de

Abstract
We have developed a mathematical model for video on demand server design based on principal component analysis. Singular value decomposition on the video correlation matrix is used to perform the PCA. The challenge is to counter the computational complexity, which grows proportionally to n^3, where n is the number of video streams. We present a solution from high performance computing, which splits the problem up and computes it in parallel on a distributed memory system.

Keywords: Parallel algorithms, Singular Value Decomposition, Principal Component Analysis

1 Introduction

Video on Demand (VOD) is an online service with a growing popularity. It can be used for many kinds of applications. Scientific fields like collaboration or E-Learning as well as commercial content like entertainment or news are only few examples of successful applications of this technology. With a growing worldwide network coverage and available bandwidth, it becomes more and more attractive for companies as well as for educational institutions to provide digital video content (DVC) online instead of via conventional channels of distribution, e.g. DVD disks. The advantages are obvious: More cost-efficient, easier to distribute and more economically friendly.

High-definition television (HDTV) is steadily gaining popularity as well. This technology enables video content to be shown in a much higher resolution than via the conventional PAL or NTSC standards. The result is a better image quality with more details and more realistic pictures. The frames of high definition videos do have a higher mean size due to their higher resolution, which increases the demand on the bandwidth simultaneously when a video stream is

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014

doi:10.1016/j.procs.2014.05.172

transmitted by a factor of 5.33 [9]. Today HDTV content is usually transmitted via conventional TV satellite and cable networks, but there are tendencies to distribute television channels also via IP-based networks.

VOD is the most demanding type of content distribution as quality of service (QoS) has to be assured to individual users (as opposed to video broadcasting). But this is very difficult since the characteristics of different video streams vary very much. Needless to say that this becomes even more challenging if HDTV content shall be offered on demand. The global Internet bandwidth of today is capable to carry this heavy load, but it also causes higher requirements on the video streaming servers. This work addresses the problem of buffer size prediction. [9] shows that the allocated buffer size for a single client with a high QoS can be up to 1 Gigabyte. Thus, it is of high interest to find an efficient way to calculate an optimal buffer size of a video stream regarding to its characteristics. The next step would be to predict the maximum number of simultaneous users per server with a constant level of QoS. This step is challenging, because a general statement has to be made about the characteristics of all provided videos. The next sections speak about how this shall be achieved and which methods are used in order to get a good prediction result.

2 Principal Component Analysis on Video Data

In [10], [9], [11], Raúl Ramírez-Velarde et al. introduced a new mathematical model to determine the maximum user load of a VBR video server. The model is based on statistical evaluation of performance measures of real MPEG-4 frame traces and should model the actual data behavior. In order to make a prediction of the required buffer size of a video, a statistical analysis of the video trace has to be performed. The analysis provides values like mean, variance, symmetry, kurtosis and self-similarity, which can be achieved with standard statistical methods. These values can be fed into the model in order to simulate the estimated buffer size.

On [8], we developed a mathematical model to design video on demand servers based on self-similarity and the Pareto Probability Distribution. We were also able to establish that principal component analysis can capture most of the behavior of a rather large collection of video files into one single video stream which we called the Characteristic Video Trace (CVT). We used principal component analysis to creat the CVT. Principal component analysis (PCA) is a common technique to reduce multidimensional data sets [7]. In this case, it is used to handle the different characteristics of the video traces, that is, each video is a dimension. As determined by the eigen values of correlation matrix, the CVT can capture increasing amounts of the total variability of the collection of data by adding principal components to the CVT.

Nevertheless one important question has not been addressed. Just how representative is the sample? In other words, if we have a sample size of n videos, what size of the population does it represent? This is important, because no doubt a video database for a video-on-demand service would contain a large collection of videos, probably in the range from 300 to 3,000 and would constantly grow. And although there are several strategies that we could use to keep the characteristic video file current, such as periodic re-computation of the characteristic video, if we knew the size of the population it represents, we believe it would not be necessary to re-compute the characteristic video until de population represented by the video taken to compute it is reached. It means for example, that if we used m videos to compute the characteristic video frame representing a population of 1,000 videos, but the video service only possesses 500 video, the characteristic video would not have to be recomputed until de video service could offer close to 1,000 videos to its users.

But determining sample size might prove difficult. Since we are using highly chaotic data

presuming it conforms to a long-tailed probability distribution, how are we to determine the appropriate samples size and population representation? One instance though, for which the sampling process is well understood is when the mean of each of the video samples is normally distributed, then we can determine sample size using well-known models.

According to the central limit theory [3], the arithmetic mean of a sufficiently large number of iterates of independent random variables, each with a well-defined expected value and well-defined variance will be approximately normally distributed. Therefore, a normality test was carried out on the mean video frame size for each video on a database of 23 videos and we found that mean video frame size is in fact normally distributed. In Figure 1 we can see the q-q plot which throws a correlation index against the normal distribution of 0.9747.

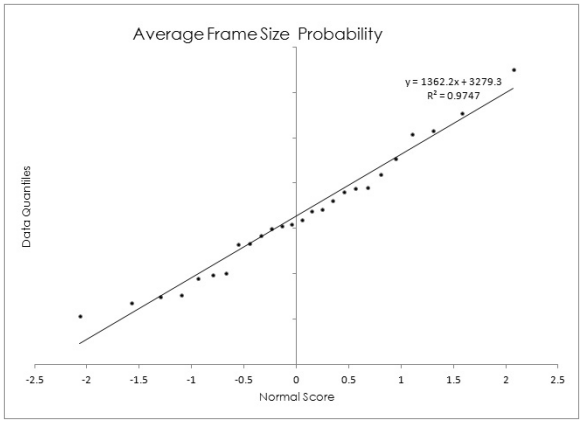

Figure 1: QQ plot for mean frame size

If the mean video frame size is normally distributed, then sample size and population sizes can be determined by well known formulas shown in table 1 [12]:

Population	**Margin of error**		
	10 %	**5 %**	**1 %**
100	50	80	99
500	81	218	476
1,000	88	278	906
10,000	96	370	4,900
100,000	96	383	8,763
+ 1,000,000	97	384	9,513

Table 1: Sample size for different population sizes

As we can see from table 1, a sample of 23 video with 5% Margin of error only represent the sample itself and not any significant population. This table shows that we need to obtain the characteristic video stream from 370 videos (assuming random selection) to represent a database of 10,000 videos, and that 384 videos would represent a video database of more than a million. So 384 is the magical number. A truly representative characteristic video would need to be obtained from at least 384 videos. This task requires of course the use of heavy computational power. As the main task of PCA is obtaining the SVD of the correlation matrix

we developed a parallel algorithm that could be implemented in computational clusters and grids.

2.1 Parallel PCA

We do PCA, as suggested by several authors, e.g. [2] or [7], by carrying out a Singular Value Decomposition (SVD) over the correlation matrix. The mentioned model to determine the maximum user load per video server has to be adaptable to virtually any number of streams. From a certain number of streams n on, this states a problem because the computational complexity of the SVD grows proportionally to n^3. Consequently, the problem becomes very difficult if one takes into consideration that about 64,000 movies have been created in the history of movie making.

Given the facts that the data load to be handled by the PCA procedure is quite large, that the eigenvalue problem (which is the core problem of the SVD) is principally parallelizable we decided to implement a parallel version of PCA. Although its true that there exists parallel very optimized implementations of SVD (see [4],[1] and [6]), we decided to implement our own parallel version since we need to handle specific aspects of SVD results in the context of PCA for video.

2.2 SVD for PCA

The singular value decomposition of a matrix $A \in \mathbb{R}^{m \times n}$ has the form

$$A = QSP^T$$

where Q is a $m \times m$ orthogonal matrix, S is a $m \times n$ rectangular diagonal matrix with non-zero elements and P is a $n \times n$ orthogonal matrix.

The elements on the main diagonal of S are referred to as the *singular values* of A, denoted $s_1, \ldots, s_l$, where $l = min(m, n)$. The columns of Q, $q_1, \ldots, q_m$, and of P, $p_1, \ldots, p_n$, are called the *singular vectors* of A. Usually, the singular value problem is solved in a way that the singular values in S (and their corresponding singular vectors in Q and P) are ordered descendingly, hence:

$$s_1 \leq s_2 \leq \cdots \leq s_l \leq 0$$

This is already very practical for the PCA, as the first singular value can already be used to extract the most important PC, called the scores, from the data set.

The singular vectors in Q are the eigenvectors of AA^T, the singular vectors in P are the eigenvectors of A^TA. The singular values stored on the main diagonal of S are the square roots of the corresponding eigenvalues (which are equal in both cases).

According to [5], an efficient way (and the most common method today) for the symmetric eigenvalue problem is to tridiagonalize the matrix before the eigenvalues are computed, so that $U_0^T A U_0 = T$ where A is the original matrix and T the tridiagonal part of the decomposition. Eigenvalues are computed from T and are equal with those of A. The eigenvectors of A can be found via reduction transformations.

Since correlation matrices are symmetric, the common way to store a symmetric matrix is to store only the upper or the lower triangular part and the main diagonal. So in order to tridiagonalize a symmetric matrix, we can also perform the overall process only in one of the both parts.

Algorithm 1 shows a Givens rotation for symmetric matrices, where the computational costs are almost halved compared to the traditional algorithm.

```
Data: A
Result: A'
j = rows(A);
for i = 1 : columns(A) − 2 do
    while j > 1 + i do
        /* rotating two appropriate rows                                  */
        [c s] = givenscs (A(j − 1, i), A(j, i));
        A(j − 1 : j, i : columns(A)) = givensrow(A(j − 1 : j, i : columns(A)), c, s);
        /* rotating the intersection square transposed                    */
        A(i − 1 : i, j − 1 : j) = givenscol(A(i − 1 : i, j − 1 : j), c, s);
        j − −;
    end
end
```

Algorithm 1: function $A' =$ **givenstridiagsymm**(A) for a symmetric matrix A

The overall process of tridiagonalizing a symmetric $n \times n$ matrix requires $\frac{8}{3}n^3$ flops. This are by the factor 2 more operations than the common algorithms, like say, Householder reflections require.

2.3 Data Dependencies and Parallel Concept

If we have a look at the inner *while*-loop of Algorithm 1, we can see that the single rows have to be processed *sequentially* and *bottom up*. Figure 2 shows the data dependencies when the elements $A(3:n,1)$ are annihilated. But what the illustration also shows is that there are no dependencies between the *single elements of a row*. And this is the key point of how parallelism can be introduced here. Instead of considering the zeroing of a single element, for what the Givens Rotations are usually used for, the overall process of zeroing a whole column up to its $(2+j)$th element has to be seen. Figure 3 shows a possible solution of how the annihilation of the elements $A(3:n,1)$ of a 5×5 matrix can be split up and distributed over *five* processors.

$$
\begin{array}{cc}
 & \begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \end{matrix} \\
\text{Step } 1 & \begin{matrix} a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \end{matrix} \quad 2 \\
3 & \begin{matrix} a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{matrix}
\end{array}
$$

Figure 2: Data Dependencies between the rows

In this way, a single step of annihilating the lower elements of the first column can be efficiently parallelized. But what the figure does not consider is that each node requires the both values c and s from the Givens matrix. Referring to the inner loop of algorithm 1, both values are computed in each iteration. This increases of course the communication overhead, since the first node has to broadcast the values in each iteration as well. One solution is to re-order the sequence of the events in the algorithm in a way that a "vector" of c and s values is computed and distributed by the first node. In this way, there is only one broadcasted message for annihilating one sub-column.

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}
p_1	p_2	p_3	p_4	p_5

Figure 3: Distributing the data over 5 processors

2.4 Tridiagonalization

Now we adapt the algorithm for the whole tridiagonalization process. The pseudocode in algorithm 2 shows how this can be done. It is a simplified version which should only show the parallelization related components.

```
Data: A
Result: A'
for i = 1 : columns(A) − 2 do
    if process contains the first column then
        Generate the c and s vector;
        Broadcast both vectors;
        forall the local columns do
            forall the elements in the column do
                Perform a rotation step;
            end
        end
        (local columns)-=1;
    else
        Receive vectors c and s;
        forall the local columns do
            forall the elements in the column do
                Perform a rotation step;
            end
        end
    end
end
```

Algorithm 2: An algorithm of the tridiagonalization process

The described parallel algorithm shows only the very principal concept of how the communication works and which data has to be processed. Some important details have been neglected and will be discussed later. Considering only this simplified algorithm, the number of sent or received messages for tridiagonalizing a $n \times n$ matrix would be $n - 2 \approx n$ for each node.

2.5 Intersection Square

The intersection square plays a special role. If an element in the lower triangular part of a symmetric matrix is zeroed, two rows are rotated. Since the matrix is symmetric, the same

process can be applied transposed in the two columns with a permuted row- and column-index. An exception is the intersection square. The rotation has to be applied here twice, once for the lower and once for the upper triangular part. How this can be solved for a sequential routine has already been shown, but if the data is distributed over several nodes, there are additional data dependencies arising. Let $S \in \mathbb{R}^{2\times 2}$ be a submatrix of $A \in \mathbb{R}^{n\times n}$:

$$S = \begin{pmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{pmatrix}$$

A rotation in the lower triangular part is performed by algorithm 3, the matching algorithm for the upper triangular part is shown in algorithm 4.

```
q = columns(A);
for j = 1:q do
    τ1 = A(1, j);
    τ2 = A(2, j);
    A(1, j) = cτ1 − sτ2;
    A(2, j) = sτ1 + cτ2;
end
```

Algorithm 3: A rotation in the lower triangular part

```
q = rows(A);
for i = 1:q do
    τ1 = A(i, 1);
    τ2 = A(i, 2);
    A(i, 1) = cτ1 − sτ2;
    A(i, 2) = sτ1 + cτ2;
end
```

Algorithm 4: A rotation in the upper triangular part

In algorithm 1, the transformation for the lower triangular part is applied first on the intersection square (**givensrow**), then the algorithm for the upper triangular part (**givenscol**). Figure 4 shows the data dependencies inside the intersection square graphically.

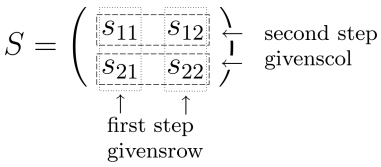

Figure 4: Data dependencies inside the intersection square

In §2.3 we assume that in the parallelization process the global matrix is split up *column wise* over the processes. This means that there exist additional data dependencies between the processes if the intersection square "lies between two processes". Figure 5 shows an example of a matrix $A \in \mathbb{R}^{8\times 8}$ distributed over 4 processes in which the $3-n$ elements of the first column are zeroed. The dashed squares represent the intersections without data dependencies, the ones with a solid line have dependencies between two processes.

```
Data: A
Result: A'
for i = 1 : columns(A) − 2 do
    if process contains the first column then
        Generate the c and s vector;
        Broadcast both vectors;
        forall the local columns do
            forall the elements in the column do
                Perform a rotation step;
            end
        end
        (local columns)-=1;
    else
        Receive vectors c and s;
        forall the local columns do
            forall the elements in the column do
                if Process contains column with the same column-index as row-index of
                element then
                    if first local column then
                        exchange first two column elements of the intersection square with
                        left neighbor
                    else
                        if last local column then
                            exchange second two column elements of the intersection square
                            with the right neighbor
                        end
                    end
                    Perform a transposed rotation step;
                end
                Perform a rotation step;
            end
        end
    end
end
```

Algorithm 5: An algorithm of the parallel tridiagonalization process considering the intersection square

Each time a process has to calculate a "half" intersection square, it has to exchange its two column-elements with those of his left or right neighboring process. The first and the last process (in the figure p_1 and p_4) play a special role because they have to exchange their elements only with one neighbor. This has to be considered for the algorithm as well. Algorithm 5 shows an improved version of the simplified communication algorithm 2 considering the intersection square.

With each annihilated column, each active process has to send 2 additional sent and 2 additional received messages. This increases the number messages of the overall parallel tridiagonalization process by averagely $4 \cdot \frac{n}{2} = 2n$. The *total* number of sent or received messages for tridiagonalizing a $n \times n$ matrix to approximately $n + 2n = 3n$.

$$
\begin{array}{c}
\left(\begin{array}{cc|cc|cc|cc}
a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\
a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} \\
a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\
a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} \\
a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} \\
a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\
a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\
a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88}
\end{array}\right) \\
\begin{array}{cccc} p_1 & p_2 & p_3 & p_4 \end{array}
\end{array}
$$

Figure 5: Intersection dependencies of a 8×8 matrix over 4 processes

2.6 Implementation

Since the goal is to develop an application which can perform the singular value decomposition as a step of the principal component analysis in parallel and in order to provide a program which is easy to adopt in different Linux and UNIX environments, the following programming standards were met for the implementation:

- Target platform: **GNU/Linux**
- Programming language: **ANSI C**
- Message passing API: **MPI**
- Except the below described parallel libraries, only libraries from the **C standard library** shall be used
- Usage of the parallel libraries:
  - **ScaLAPACK** for finding eigenvalues
  - **PBLAS** for matrix operations
  - **BLACS** for the data distribution
- From the used libraries above derive the following dependencies:
  - **LAPACK**
  - **BLAS**

3 Results

Benchmarks for several matrix sizes were made, to see how the implemented routines perform. For the benchmarks *all* singular values and their corresponding vectors were computed. The measurements are taken from matrices of several sizes, beginning with $N = 512$ up to $N = 3072$. In the usual case, the efficiency becomes better with an increasing problem size, because the

sequential part of the program, which is here the initialization and the data distribution- and gathering, grows slower with the problem size than the parallel part. Test on a larger cluster and even grid environments are on the works in order to test the implementation for really huge matrices.

The main conclusion of *Gustafson's Law* is: An appropriate efficiency will be obtained as long as the problem size is big enough. Generally we can say that *theoretically* the sequential part grows by $\mathcal{O}(N^2)$ and the parallel part by $\mathcal{O}(N^3)$. We've proven this by comparing the growth of runtimes of the both parts. Figure 6 shows a comparison between the two parts. The parallel part represents only the computation routines in parallel, i.e. the transposing, matrix multiplication and eigenvalue algorithm. The sequential part is a sum of the initialization time, the data distribution time and the data gathering time. To make it comparable, the both values are normalized to the first value.

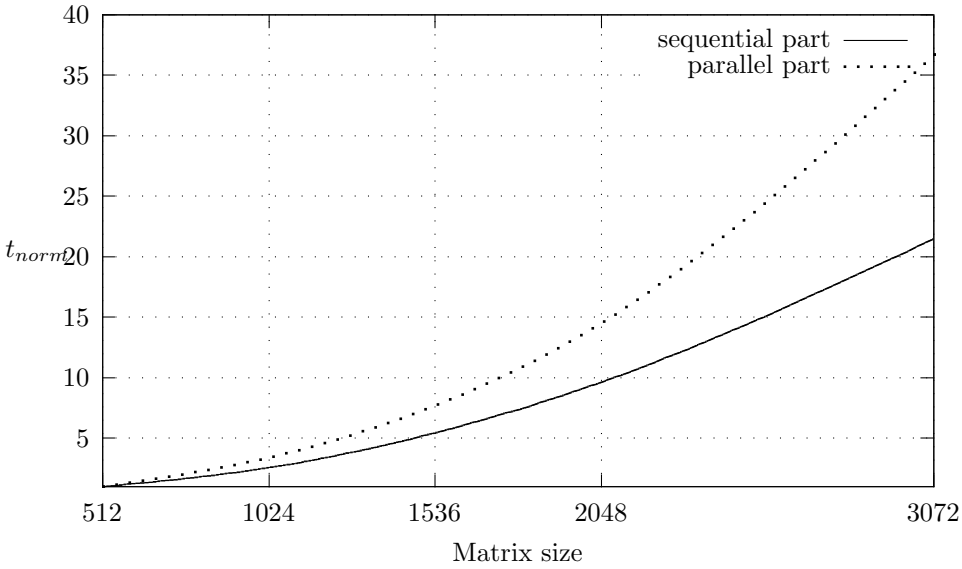

Figure 6: Growth of the sequential and the parallel part, normalized

We can see that the parallel part, represented by the dotted lines, grows faster than the sequential part. But the growth is much smaller than in the theoretical assumption, in which there is a difference of one polynomial degree. Let us have a look at another graph, shown in figure 7. It shows the speedup gained for different number of processors over the matrix size. In this way we can see very good how the speedup develops referring to Gustafson's Law.

Obviously the curves become better the more processing elements are used. The graph shows from which matrix sizes it makes sense to use parallelization. The first size, $N = 512$, has always a speedup below 1, so here it is better to run the process only on one processor. From $N = 1024$ on it starts to make sense to use more processing units. But since the speedup has more or less the same value, $p = 2$ has the best efficiency. With the growing size of the matrix, we can see that the speedup differs more and more between the number of used processors.

As a next step, we evaluated the efficiency. For a matrix of the size $N = 512$, the efficiency is obviously more than poor, since a number of processors of $p = 2$ has already a speedup below 1. The communication overhead exceeds the performance gain so that a speedup can not be reached.

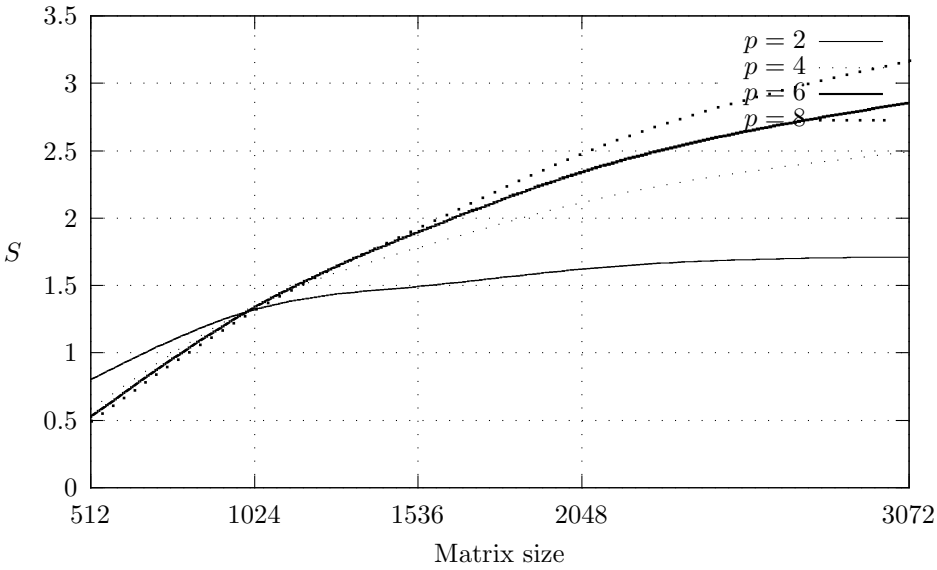

Figure 7: Speedup gained against the matrix size

The matrix size of $N = 1024$ shows another picture. Running the application on two processors ($p = 2$), the efficiency is $E = 0.67$, which is quite acceptable. With an increasing number of processors, the efficiency becomes worse and is not very feasible.

Matrix sizes $N = 1536$ and $N = 2048$ show a slightly improved picture. The efficiency do not fall as fast, but the results are still not very satisfying.

The only case with a reasonably good efficiency curve is the one of the matrix size of $N = 3072$. Still, the efficiency of $p = 8$ processing units lies below $E = 0.4$. We should also consider that the runtimes of this version already are between 20 and 5 minutes (depending on the number of used processors), which is quite long.

From the data above we can conclude that if the long latencies of the underlying *Gigabit Ethernet* could be avoided, efficiency could be improved for small matrices. However, with big matrix sizes, an appropriate speedup can be achieved and the presented implementation shows properly how the problem of finding the principal components of a video correlation matrix can be found in parallel.

4 Conclusions

We have shown a solution to compute the singular value decomposition in parallel. The focus is on performing the PCA, so only the greatest singular values respectively the principal components are searched. This makes the process more efficient, since superfluous information (the following PCs) are not considered. Data source are video correlation matrices, which are always symmetric. Thus, only routines for symmetric matrices are used, which speeds the overall process up, as redundant computation is avoided.

References

[1] Michal Aharon, Michael Elad, and Alfred Bruckstein. -svd: An algorithm for designing overcomplete dictionaries for sparse representation. *Signal Processing, IEEE Transactions on*, 54(11):4311–4322, 2006.

[2] John M. Chambers. *Computational Methods for Data Analysis*. Wiley, 1977.

[3] Richard Duret. *Probability: theory and examples (4th ed.)*. Cambridge University Press, 2004.

[4] GR Gao and SJ Thomas. An optimal parallel jacobi-like solution method for the singular value decomposition. In *Proc. Internat. Conf. Parallel Proc*, pages 47–53, 1988.

[5] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.

[6] Vicente Hernández, José E. Román, and Andrés Tomás. A robust and efficient parallel svd solver based on restarted lanczos bidiagonalization. *Electronic Transactions on Numerical Analysis*, 31:68–85, 2008.

[7] Ian T. Jolliffe. *Principal Component Analysis (second edition)*. Springer, 2002.

[8] Raul Ramirez-Velarde, Lorena Martinez-Elizalde, and Carlos Barba-Jimenez. Overcoming uncertainty on video-on-demand server design by using self-similarity and principal component analysis. *Procedia Computer Science*, 18(0):2327 – 2336, 2013. 2013 International Conference on Computational Science.

[9] Raúl V. Ramírez-Velarde. *Performance Analysis of a VBR Video Server With Self-Similar Gamma Distributed MPEG-4 Data*. PhD thesis, Instituto Tecnológico y de Estudios Superiores de Monterrey, 2004.

[10] Raúl V. Ramírez-Velarde and Ramón M. Rodríguez-Dagnino. A gamma fractal noise source model for variable bit rate video servers. *Computer Communications*, 27:1786–1798, 2004.

[11] Raúl V. Ramírez-Velarde and Ramón M. Rodríguez-Dagnino. Capacity planning of it infrastructure using pareto self-similar stochastic processes. Technical report, Instituto Tecnológico y de Estudios Superiores de Monterrey, 2006.

[12] Taro Yamane, editor. *Statistics: An Introductory Analysis (2nd Ed.)*. New York: Harper and Row, 1967.