Journal of Information & Communication Technology
Vol. 2, No. 2, (Fall 2008) 116-125

Parallel Processing on Pentium Machines: a Low cost Solution for High Performance Computing

Syed Misbahuddin[*]
Fazal Noor[*]
Saleh Zugail[*]
Department of Computer Science and Software Engineering
University of Hail, Hail, Saudi Arabia

ABSTRACT

Parallel processing is a form of computing in which a number of activities are carried out concurrently on multiple machines so that the time required to solve a given problem is minimized. The advent of low cost high performance Pentium machines and PC based LINUX operating system have attracted many researchers to explore parallel processing experiments on PCs. The PC based clusters provide low cost solution of high performance computing. In this paper, we provide detailed steps on setting up a LINUX cluster. Also, a cluster application has been discussed by mapping a compute intensive scientific algorithm. Furthermore, a scheme is presented for distance computing, in which the cluster based parallel processing facilities are extended to the remote users through the Internet.

INSPEC Classification : C4240P, C5130, C6110P, C6150N

1. INTRODUCTION

Scientific and engineering communities have been using Parallel computing techniques solve large scale complex problems. For parallel computing applications, parallel supercomputers have been used in the past. However, their popularity is declining due to factors like being expensive to purchase, run and maintain, slow to evolve in the factor of emerging hardware technologies, and difficult to upgrade without, generally, replacing the whole system etc. The parallel processing is also possible by making a cluster of microprocessor based low cost personal computers (Anderson, T.E., D.E. Culler and Peterson). This paper describes the experience of building a cluster of PCs by utilizing low cost Pentium machines and the LINUX Enterprise Edition operating system.

In parallel programming software, the process coordination by message passing is required to achieve the problem solution objective. Advanced software developers have designed and implemented several interesting programming models to help develop parallel applications. The most popular ones are OpenMP for shared memory programming and

[*] The material presented by the authors does not necessarily portray the viewpoint of the editors and the management of the Institute of Business and Technology (BIZTEK) or College of Computer Science & IT , Saudi Arabia

[*] Syed Misbahuddin : smisbah@uoh.edu.sa
[*] Fazal Noor : fnoor@uoh.edu.sa
[*] Saleh Zugail : zugail@uoh.edu.sa

MPI (Message Passing Interface) for distributed memory programming (Chandra, 2001), (Pacheco P., 1997). Recently, multi-threading programming on Symmetrical Multiprocessor (SMP) architectures and message-passing programming on distributed memory systems are becoming more popular and prevalent.

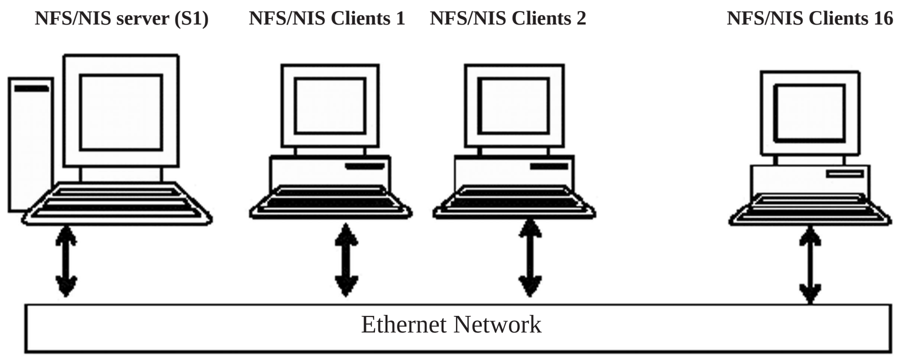
There are several MPI implementations such as MPICH, LAM (Edinburgh Parallel Computing Centre, 1991), Common High-level Interface to Message-Passing CHIMP(W.E. Trench, 1989). etc. In this paper, we use LAM implementation. LAM (Local Area Multicomputer) was developed at Ohio Supercomputer Center. LAM is a programming environment and development system for a message-passing parallel machine constituted with heterogeneous UNIX computers on a network The Message Passing Interface Standard (MPI) is a message passing library. It is widely used in writing programs in which distributed computing are necessary.

In section 2, we give the steps on how to build a PC cluster. Step by step process of developing Intel PC based parallel processing platform has been described in the Appendix at the end of this paper. Section 3 presents a compute intensive application in area of signal processing where MPI functions are utilized to parallelize an algorithm written in C++ and run under a LAM environment cluster. Finally, conclusion is given in section 4.

2. DEVELOPMENT OF A PC CLUSTER

Initially, a private local area network (LAN) was set up consisting of a server (which later became the head-node of the cluster-based system) and two clients. We linearly expanded the cluster by adding 14 clients to make a 17 node cluster (1 Server (master) + 16 Clients (slaves)). This scheme allowed us to make a Master-Slave model with 1 master and 16 slaves. We kept an even number of clients for graceful mapping of parallel algorithms on the cluster. Figure 2 shows 17 nodes cluster. In this cluster, the S1 machine is an NFS/NIS server and all the other machines (S2 – S17) are NFS/NIS clients. We used available a mix of P3 and P4 machines.

Figure 2.
A 17 node Cluster



The basic steps involved in building a cluster are summarized as following:

1) Installation of Linux Enterprise edition on all pentium machines.
2) Network configuration on Linux nodes.
3) Configuring NFS/NIS server and clients.
4) Configuring Local Area Multicomputer (LAM) environment for launching parallel applications on Linux nodes.

Complete detailed steps of building a cluster are provided in the Appendix.

3. USING LINUX CLUSTER FOR PARALLEL APPLICATIONS

A cluster of 17 PCs was used to run a program in parallel to compute computationally intensive complex problem. One such problem arises in array signal processing where a process is stationary and the covariance matrix is Hermitian Toeplitz. In this case matrices formed are of large order and problem reduces to compute eigenvalues. Algorithms exist in literature which for high order Hermitian Toeplitz matrices, are very compute intensive (Y. H HU, 1985), (F. Noor, 1993), (F. Noor, 1992). In this section, we have implemented algorithms on a cluster of Pentium machines using MPI functions within C++ software. The mapped algorithm run in parallel to compute all the eigenvalues of a given Hermitian Toeplitz matrices.. The parallel algorithm presented is an improvement over author's previous work (F. Noor, 1992). The parallel algorithm runs on a cluster of Pentium machines and further improves the rate of convergence. Next, we present mathematical development and summarize the sequential and the parallel algorithms.

Mathematical Development
Given a Hermitian Teoplitz matrix C*n* of order n,

$$C_n = \begin{bmatrix} c_0 & c_0 & \cdots & c^*_{n-1} \\ c_1 & c_1 & \cdots & c^*_{n-2} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ c_{n-1} & c_{n-2} & \cdots & c_0 \end{bmatrix}$$

Where c_0 is real and $c_1, c_2, \ldots, c_{n-1}$ are complex, the problem is to find the complete eigenspectrum. Since C_n is Hermitian, $c^*_{-i} = c_i$, $i = 0,1,\ldots, n-1$. The principal submatrix of C_n of order k is defined as $C_k = [c_{i-j} : 0 \# i, j \# k-1]$, $k=1,2,\ldots,n$.

Consider the following linear system involving the shifted Hermitian Toeplitz coefficient matrix C_n-λI$_n$:

$$(C_n - \lambda I_n) \begin{pmatrix} 1 \\ -\mathrm{i}_{n-1}(\lambda) \end{pmatrix} = [E_n(\lambda)\ 0 \ldots 0]^T$$

where λ is treated as a continuous real variable. We assume all principal submatrices , $C_k - \lambda I_k$, $k = 1, 2, \ldots, n$, are nonsingular. This implies that the Levinson-Durbin(L-D) recursion can be applied to $(C_{n-1} - \lambda I_n)\ \mathrm{i}_{n-1} = [c_0 c_1^* \ldots c^*_{n-1}]^T$

In order to find the elements i_{n-1i}, $i = 1,2, \ldots, n-1$, of $\mathrm{i}_{n-1}(\lambda)$ in a recursive fashion using the following algorithm:

Initialization:
$E_0 = c_0 - \lambda$,

For 1#*k*#:n-1
Compute:

$$\rho_k = \frac{1}{E_{k-1}} \left[c_k - \overset{k-1}{\underset{i-1}{3}} \mathrm{i}_{k-1,} c_{k-1} \right]$$

$$\mathrm{i}_{kk} = \rho_k$$

$\mathrm{i}_{kk} = \mathrm{i}_{k-1,i} - \rho_k \mathrm{i}^{*}_{k-1,}$ 1#*i*#*k* - 1
$E_k = E_{k-1}(1- {*}\rho_k{*}^2)$

The parameters ρ_k and E_k are known as the reflection coefficient and prediction error, respectively, at the *k*th recursive step. Note that all the quantities in equation (3) depend on the parameter λ through the initialization E_0 .

The following algorithm involves 3 steps to sequentially extract all the eigenvalues of a Hermitian Toeplitz matrices.

Modified-Algorithm-Hermitian Toeplitz Matrices (MAHTM)

Step 1-Select: Find the eigenvalues $\lambda_p, \lambda_{p+1},...,\lambda_q$, 1#*p*<*q*#*n*. Using trial and error, select an interval (*a*,*b*)by bisection such that $Neg_n(a)$#*p*-1, $Neg_n(b)\ \$q$

For *i* =*p*To *q*-1.

Step 2-Search: Search for the endpoint i $U\xi_i$ not captured by trial and error such that $(L\xi_i, U\xi_i)$ contains λ_i. This is done by bisection and by keeping count of the negative signs of $\{E_1(U\xi_i), E_2(U\xi_i),..., E_n(U\xi_i)\}$. During this search process, keep tightening, capturing, and storing the locations of other desired eigenvalues, while also retaining the values $E_n(L\xi_i)$, $E_n(U\xi_i)$, and $E_n(L\xi_{i+1})$.

Step 3-Refine: Once all the intervals $L\xi_i < \lambda_i < U\xi_i$, *p*#*i*#*q*, are obtained:
For *j*=*p* To *q*

(a) Set $\alpha = L\xi_j$, $E_a = E_n(L\xi_j)$ and $\beta = U\xi_j$, $E_\beta = E_n(U\xi_j)$.
(b) For multiple eigenvalues, set the matrix order *n* to *n-m+1* and work with the submatrix C_{n-m+1}. By trial and error, refine the interval $(\alpha,\beta)$to $(\alpha',\beta')$by bisection such that the following conditions hold:
i. $Neg_n(\alpha')$= *j*-1. and $Neg_n(\beta') = j$
ii. $E_n(\alpha')>0$ and $E_n(\beta')<0$.
(c) Switch to *MRQI* or *PEGASUS* method to find λ_j.
Next *j*
End.

Parallelism in the above algorithm can be done several ways. Some of the methods are as follows:

Method 1). The above algorithm *MAHTM* is executed in parallel on each computer (node) in the cluster for different range of eigenvalues. So in parallel each node finds for p=my_rank*n/psize +1 to q=(my_rank +1) * n/psize range of eigenvalues, where my_rank is the rank of each computer in the group, psize is the number of computers or nodes in a group, and n is the size of the matrix. We will call this method *MAHT-P*.

Method 2). Master Computer: Performs Step1-Select and Step2-Coarse Search then once all intervals are obtained then in parallel these intervals may be send to the slaves to extract the eigenvalues. Slave computers: Each slave$_x$ in parallel receives an interval and uses either MRQI or Pegaus method to find λ_j. In other words, parallelism is done at Step 3 of MAHTM and slaves are idle the whole time Master Computer performs step 1 and 2.

Method 3). Master and Slave Computers: All computers master and slave participate in obtaining the coarse intervals and pass the intervals to the master which acts as a coordinator. The master then sends one interval to each slave and waits for the result. Once the result is received it sends another interval to the same slave. The process is repeated till no more intervals to send.

Method 3 is summarized as follows:

MPI-Parallel-Eigen Algorithm Hermitian Toeplitz (MPEAHT) Matrices

*Step 1: In parallel each computer (Master and Slaves) finds the range of intervals from p to q each containing an eigenvalue according to their rank . Each has range from p= my_rank*n/psize + 1 to q=(my_rank + 1) * n / psize intervals to find, where my_rank is rank of each computer in the group, psize is the number of computers in the group, and n is the size of the matrix.*

Step 2: Intervals found are sent to Master Computer and then Master Computer sends an interval to a slave to find λ_j. Each slave receives a single interval at a time. The slave computes an eigenvalue and sends back a signal to master to send another single interval containing an eigenvalue. In this scenario, there is communication between master and slave but slaves are load balanced. Time spent by slave to compute an eigenvalue exceeds the time taken to communicate with the master.

In MPI-Parallel-Eigen (MPEAHT) the master and slave computers perform the coarse search in which non-contiguous intervals, in general, are obtained containing an eigenvalue per interval.

Below is the skeleton of the MPI program used with the above algorithm.

```
void intervals( int p, int q );
void roots( int kkk );

int main(int argc, char** argv)
{
int WORKTAG = 1;
int EXITTAG = 2;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &psize);

// p= my_rank*nn/psize + 1;
// q= (my_rank + 1) * nn / psize;
for ( rank = 0; rank < psize; ++rank)
{
if ( my_rank == rank )
{
intervals ( p, q );

if ( my_rank != 0 )
{
for ( kkk=p; kkk<=q; ++kkk)
{

// Each slave P_i have found intervals will send from p to q intervals to Master
// Sending intervals back to Master with rank = 0
MPI_Send(&work, 5, MPI_DOUBLE, 0, 22, MPI_COMM_WORLD);

} /* end-for kkk */
} /* end if rank not 0 */
} /* if my rank == rank */
} /* for loop rank = 0 */
```

```
if ( my_rank == 0 )
{
n1=nn/psize + 1;

for ( l=n1; l<=nn; ++l )
{
MPI_Recv(&work,5,MPI_DOUBLE, MPI_ANY_SOURCE,MPI_ANY_TAG,
MPI_COMM_WORLD, &status);

} /* end for l=n1 */
} /* end-if my rank == 0 */

// Range of eigenvalues to search
p = 1; q = nn;

// Master is finding the coarse intervals containing an eigenvalue
if ( my_rank == 0 )
{ intervals ( p, q );

// Sending intervals to processors with rank 1 to psize -1
for ( rank=1; rank < psize; ++rank)
{ MPI_Send(&work,5,MPI_DOUBLE, rank,WORKTAG, PI_COMM_WORLD);
}

// While there are intervals send them to find eigenvalues in them
i=psize;
while ( i != nn+1 )
{ MPI_Recv(&result,4,MPI_DOUBLE, MPI_ANY_SOURCE,MPI_ANY_TAG,
MPI_COMM_WORLD, &status);

MPI_Send(&work,5,MPI_DOUBLE, status.MPI_SOURCE,WORKTAG,
MPI_COMM_WORLD);
} /* end-while */

// No More Intervals left so recieve all that is still being worked upon
for ( rank=1; rank < psize; ++ rank)
{ MPI_Recv(&result,4,MPI_DOUBLE, MPI_ANY_SOURCE,MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
}

// Broadcast signal to slaves to indicate no more jobs therefore exit
for ( rank =1; rank < psize; ++rank)
{MPI_Send(0,0,MPI_INT,rank,EXITTAG, MPI_COMM_WORLD);
}

} /* end-if my rank equals 0 */

else {
// Slaves do the work; Processor i receives the lower and upper intervals
// slave returns with result array containing, eigenvalue, no of iterations, time elapsed in
seconds.

while ( 1 ) {

/* Receive information from Master */
MPI_Recv(&work,5,MPI_DOUBLE, 0,MPI_ANY_TAG,MPI_COMM_WORLD, &status);
/* Check the tag of the received message *

if ( status.MPI_TAG == EXITTAG )
{ MPI_Finalize( );
return 0;
}

/* Do the work */
roots( kkk );
MPI_Send(&result,4,MPI_DOUBLE,0,0, MPI_COMM_WORLD);
} /* end while ( 1 ) */

} /* end-if my rank equals 0 else */
MPI_Finalize( );
return ( 0 );
}
```

In MPEAHT the master dispatches intervals among the slaves and waits to receive result and then sends another interval. In this case load is balanced in time among the slaves, therefore some slaves may compute more eigenvalues than others.

We implemented MPEAHT in C++ with MPI functions and ran it on our cluster under LAM environment. With LAM, a LINUX cluster can act as one parallel computer solving one computeintensive problem as in our application.

Parallel programs are executed on LINUX cluster by two methods:

1. Interactive: User programs are directly executed at the login shell, and run immediately.
2. Batch: Users submit jobs to system program which will be executed according to the available resources and site policy.

In our cluster we used the Interactive option. In this method, a non-root user needs to log on to NFS server (S1 in our cluster) and launch LAM parallel programming environment (PPE). This environment can be established on all nodes in the cluster or on a subset of nodes. To initiate PPE, nodes' host names are written in text file and "lamboot" command is issued by non-root user. The syntax of initiated PPE is shown below:

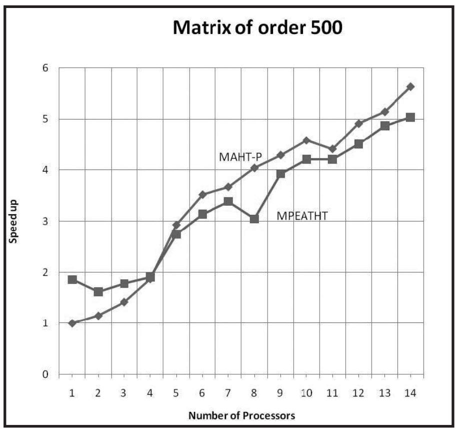
```
lamboot –v machinefile
```

In the above command, machinefile is the text file containing the names of machines on which PPE is to be launched. A parallel program can be written in C utilizing MPI functions. The parallel program will be compiled by a special compiler called Handle-C compiler (hcc). The executable image can be sent to run on n nodes in the cluster by following MPI command:

```
mpirun –np n < executable image name>
```

where n is the number of machines on which the executable image will be loaded and executed. For example, let the executable image name be "abc" and n=17, then the following command is issued on NFS server:

Figure 3
Speedup vs number of processors of MAHT-P and MPEAHT algorithms.



Experimental Results

We implemented the above algorithm in C++ with MPI and ran it on a 17 node cluster. Figure 3 shows the speedup of the two algorthims. Speedup is defined as T_1 / T_p where T_1 is the execution time on one processor and T_p is the execution time on P processors. We note from Figure 3, MAHT-P performs better than MPEAHT for small number of processors (less than 4) however as the processors are increased MPEAHT outperforms MAHT-P. Tables 1a and 1b, below show two sample runs of algorithm MPEAHT on six slaves machines in the cluster. The matrix size is 500 for this sample run. Note, same processor may have different number of intervals at different times resulting in imbalance in number of roots. Also note some processors may be slower than others, for example, P1 although has less number of iterations than P2, P1 takes more time than P2. Similar conclusions can be made for other processors.

MPEAHT

Table 1a.

	No. Roots	No. Iter	Seconds
P1	62	287	5.72
P2	86	410	5.5
P3	96	419	5.34
P4	93	431	5.52
P5	82	356	5.25
P6	81	406	5.7
	500	2309	5.505

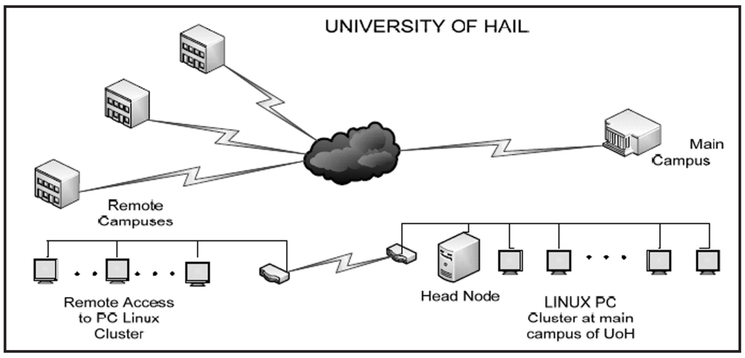
Table 1b.

	No. Roots	No. Iter	Seconds
P1	60	293	5.85
P2	96	443	5.56
P3	60	285	5.66
P4	96	438	5.51
P5	100	450	5.7
P6	88	400	5.35
	500	2309	5.605

4. REMOTE ACCESS OF CLUSTER FOR PARALLEL PROCESSING

Before parallel jobs could be started, a parallel programming environment (PPE) is launched on all participating node inside the cluster. To initiate PPE, a non root user logs on the NFS/NIS server node or "head node" and issues the necessary commands. Once PPE is running the parallel code is executed by issuing "mpirun" command specifying the number of nodes one which the parallel job is to be run. Therefore, for cluster's remote access, only the head node is made accessible to the remote sites as shown in Figure 4.

Figure 4
Remort users can access the head node of LINUX PC Cluster at Main Bldg via internet.


UNIVERSITY OF HAIL
Main Campus
Remote Campuses
Remote Access to PC Linux Cluster
Head Node
LINUX PC Cluster at main campus of UoH

Head node will have a TCP/IP connectivity to the LAN connecting all cluster nodes and to the corporate Intranet. The corporate Intranet is accessible by Internet through Virtual Private Network (VPN) set up on the Internet client machines. The VPN configuration on Internet client machines will allow users to connect to the corporate network. Once an Internet client is virtually connected to the corporate network, he or she can access the Linux cluster seamlessly. Other alternative approach is allocating a public IP address to the head node. This way, any Internet client can directly connect to the head using Telnet protocol over the Internet. However, the second approach is insecure. Therefore, former approach via VPN is more desirable.

CONCLUSION

Parallel computing is used for achieving fast computational results in variety of areas ranging from engineering applications to commercial applications. An economical solution for structuring parallel computing system is possible by connecting customary Intel based PCs by a network. This ensemble of PCs can form a cluster of workstations. By using Message Passing Interface MPI, this cluster can be used for implementing parallel algorithms. This paper has discussed the details of building a PC cluster. We have discussed an example of parallel programs to motivate users to utilize the PC cluster. Also remote access to the cluster further makes it cost effective solution of parallel computing solution for geographically dispersed users' community. Access of the cluster for parallel processing may be extended kingdom wide scientific community.

ANDERSON, T.E., D.E. CULLER AND PETERSON, " A case of NOW(Cluster of workstations", IEEE Micro Vol. 15, No.1, pp. 54-64

SYED MISBAHUDDIN and FAZAL NOOR, (2007.) "Hands-on workshop on parallel processing", Department of computer science and software engineering, University of Hail, Saudi Arabia, May 2007.

CHANDRA R., L. DAGUM D. KOHR. D. MAYDAN, J. MCDONALD and R. MENON (2001) "Parallel Programming in OpenMP", Margon Kaufmann Publishers, San Francisco, CA, 2001.

PACHECO P. (1997)"Parallel Programming with MPI", Morgan Kaufmann Publishers San Francisco, CA, 1997.

http://www.mcs.anl.gov/mpi/mpich/.

EDINBURGH PARALLEL COMPUTING CENTRE (1991), University of Edinburgh. CHIMP Concepts, June 1991.

W.F. TRENCH (1989), "Numerical solution of the eigenvalue problem for Hermitian Teoplitz matrices," SIAM J. Matrix Anal. Appl., vol. 10, no. 2, pp. 135-146, Apr.1989.

Y.H. HU and S. Y. KUNG (1985), "Toeplitz eigensytem solver," IEEE Trans. Acoust. Speech, Signal Processing, vol ASSP-33, pp. 1264-1271, Oct. 1985.

F. NOOR and S.D. MORGERA (1993), "Recursive and Iterative Algorithms for Computing Eigenvalues of Hermitian Teoplitz Matrices," IEEE Transcations on Signal Processing, vol. 41, no. 3, pp. 1272-1279, March 1993.

F. NOOR and S.D. MORGERA (1992), "Construction of a Hermitian Toeplitz matrix from an arbitrary set of eigenvalues," IEEE Trans. Siganl Processing, vol. 40, no. 8, pp. 2093-2094, Aug 1992.

Y.H. HU(1989), "Parallel eigenvalue decomposition for Teoplitz and related matrices," in Proc. ICASSP '89, Glasgow, Scotland, pp. 1107-1110.

G. CYBENKO and C. VAN LOAN (1986), "Computing the minimum eigenvalue of a symmetric positive definite Teoplitz matrix," SIAM J. Sci. Stat. Comput., vol. 7, pp. 123-131, 1986.

A.A. BEEX and M.P. FARGUES (1989), "Highly parallel recursive iterative Toeplitz eigenspace decomposition," IEEE Trans. Acoust. Speech, Signal Processing, vol. 37, no. 11. pp. 1765-1768, Nov. 1989.

E.H. GOLUB and C. VAN LOAN (1983), Matrix Computations, Baltimore, MD: John Hopkins, University Press, 1983, pp. 305-312.