Timing Analysis of a Parallel Algorithm for Toeplitz Matrices on a MIMD Parallel machine

appeared in Parallel Computing, Vol. 17, pp. 563-577, 1991

I. Gohberg † I. Koltracht ‡ *
A. Averbuch † B. Shoham †

† School of Mathematical Sciences
Tel-Aviv University
Tel-Aviv 69978, Israel

‡ The College of Liberal Arts and Sciences
The University of Connecticut
Department of Mathematics
196 Auditorium Road
Storrs, Connecticut 06268, USA

E-Mail: amir@MATH.TAU.AC.IL

Abstract

In this paper performance analysis of a parallel Levinson-type algorithm for Toeplitz matrices is given. A modified version of the parallel algorithm is presented to improve performance. The algorithm is implemented on a shared-memory MIMD (non-vector) machine. The derivation of the parallel algorithm is presented. The speedup limitation is investigated along with the optimal number of processors needed to a given Toeplitz matrix.

1 Introduction

The solution of linear systems of equations with a Toeplitz coefficients matrix $R = \{r_{i-j}\}_{i,j=0}^{N}$ is important in many applications in science, engineering and mathematics (see [B] and references therein).

In some applications the Toeplitz matrix is positive definite (i.e. $R = \{r_{|i-j|}\}_{i,j=0}^{N}$) and the desired solution is the vector γ_N for which

$$R\gamma_N = e_N = [0, \ldots, 0, 1]^T. \tag{1}$$

*The work by the first two authors was supported by the NSF grant #DMS 8801961

In 1947 Levinson [L], proposed a fast algorithm for solving Eq. (1). We present it here in a form which is suitable for our forthcoming analysis. Since R is positive definite we assume without any loss of generality that $r_0 = 1$.

Levinson Algorithm:

1. Start with $\gamma_0(0) = 1$.

2. For $k = 1, \dots, N$ compute

$$c_k = -\sum_{j=0}^{k-1} r_{j+1}\gamma_{k-1}(j), \tag{2}$$

$$\gamma_k = \frac{1}{1-c_k^2}\left\{\begin{bmatrix} 0 \\ \gamma_{k-1}(0) \\ \vdots \\ \gamma_{k-1}(k-1) \end{bmatrix} + c_k \begin{bmatrix} \gamma_{k-1}(k-1) \\ \vdots \\ \gamma_{k-1}(0) \\ 0 \end{bmatrix}\right\} \tag{3}$$

3. Thus $R\gamma_N = e_N$.

The one advantage of this algorithm is its relatively low operations count and memory requirement ($O(N^2)$ and $O(N)$ respectively). It also computes the quantities $c_1, \dots, c_N$, known as the Schur or reflection coefficients, which are important in their right (see [KL], for example).

The improved speed of the Levinson algorithm (as opposed to $O(N^3)$ for general systems of equations), is important because of a possibly large size of Toeplitz systems and a large number of such systems to be solved. (Note that Toeplitz matrices are often not sparse).

A significant effort was recently put into further speed-up of the solution of Toeplitz systems, in particular, by the use of modern computer architectures (see [KH] and references in [B] for parallel architectures and [K] for vector machines).

One parallel version of the Levinson algorithm was suggested in [GKKL].

Parallel Levinson Algorithm:

1. Start with $\gamma_0 = 1,\ \gamma_0(j) = -r_j,\ j = \pm 1, \dots, \pm N$

2. For $k = 1, \dots, N$ compute

$$\beta_k = [1 - \gamma_{k-1}^2(-1)]^{-1}$$

$$\gamma_k(0) = \beta_k \gamma_{k-1}(-1)\gamma_{k-1}(k-1)$$

$$\gamma_k(k) = \beta_k \gamma_{k-1}(k-1)$$

$$\gamma_k(j) = \beta_k[\gamma_{k-1}(j-1) + \gamma_{k-1}(-1)\gamma_{k-1}(k-j-1)] \quad j = k-N, \dots, N,\ j \neq 0, k \tag{4}$$

3. Thus $R_N\gamma_N = e_N$

This algorithm requires $O(N)$ arithmetic operations on $O(N)$ parallel processors where the computation of $\gamma_k(j)$ components is done in parallel. The main difference between the parallel and the standard Levinson algorithms is that the reflection coefficients $c_k = \gamma_k(-1)$

do not have to be computed as inner products which, in principle, could allow more efficient implementation of the parallel algorithm on a parallel and vector machines.

In this paper we analyze such an implementation on a MIMD shared bus & shared memory multiprocessor architecture. Since the communication among the processors is done through a shared bus, (so that only one processor can use at a time as explained in section 3.1), the communication overhead is very significant in our implementation. Note that in the parallel Levinson algorithm, as presented in [GKKL] (Eq. (4)), the computation of the $j-th$ component of γ_k is done using the $(j-1)$, and $(k-j-1)$ components of γ_{k-1}. But in a shared bus architecture these components has to be transmitted through the shared bus to the processor that computes the $j-th$ component of γ_k. This increase the communication overhead and therefore increase the processing time.

In this paper we present a modification of the parallel Levinson algorithm that needs less communication which is related to the computation of γ_k, therefore the modified algorithm better exploits the machine and tailored to its architecture. Processing time of this algorithm for a fixed size Toeplitz matrix decreases when the number of processors increase, up to an optimal number of processors due to the limitation of the communication through the shared bus.

The paper is organized in the following way : In Section 2 we present the modified parallel Levinson algorithm. In Section 3 we introduce the parallel multiprocessor and the specific parallel implementation. Numerical results are presented in Section 4 and finally, Section 5 summarizes our results.

2 The Modified Parallel Levinson Algorithm

Following [GKKL] consider the sequence of $(N+1)\times(N+1)$ matrices,

$$R_k = \begin{bmatrix} R_k^l & \mid 0 \\ --- & --- \\ R_k^c & \mid I \end{bmatrix} \quad k = 0, \ldots, N \tag{5}$$

where $R_k^l = \{r_{|i-j|}\}_{i,j=0}^k$ and

$$R_k^c = \begin{bmatrix} r_{k+1} & \cdot & \cdot & \cdot & r_2 & r_1 \\ r_{k+2} & \cdot & \cdot & \cdot & r_3 & r_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ r_N & \cdot & \cdot & \cdot & r_{N-k+1} & r_{N-k} \end{bmatrix} \tag{6}$$

is a $(N-k)\times(k+1)$ Toeplitz matrix. Since $R_N = R$ is positive definite, it follows that R_k is invertible.

Consider also solution vectors γ_k and φ_k, for $k = 0, \ldots, N$ defined by:

$$R_k\gamma_k = e_k, \quad R_k\varphi_k = e_0, \tag{7}$$

where $e_k = [0, \ldots, 0, 1, 0, \ldots, 0]^T$ with 1 in the $k+1th$ position. It is shown in [GKKL] that

$$\varphi_{k-1}(k) = c_k = -\sum_{j=0}^{k-1} r_{j+1}\gamma_{k-1}(j), \tag{8}$$

and that

$$\gamma_k = \begin{bmatrix} 0 \\ \gamma_{k-1}(0) \\ \vdots \\ \gamma_{k-1}(N-1) \end{bmatrix} + c_k \varphi_k \triangleq \bar{\gamma}_{k-1} + c_k \varphi_k, \tag{9}$$

$$\varphi_k = \begin{bmatrix} \varphi_{k-1}(0) \\ \varphi_{k-1}(k-1) \\ 0 \\ \varphi_{k-1}(k+1) \\ \vdots \\ \varphi_{k-1}(N) \end{bmatrix} + c_k \gamma_k \triangleq \bar{\bar{\varphi}}_{k-1} + c_k \gamma_k. \tag{10}$$

(These equations can be verified directly). Thus

$$[\gamma_k \ , \ \varphi_k] \begin{bmatrix} 1 & -c_k \\ -c_k & 1 \end{bmatrix} = [\bar{\gamma}_{k-1} \ , \ \bar{\bar{\varphi}}_{k-1}] \tag{11}$$

and hence

$$[\gamma_k \ , \ \varphi_k] = \frac{1}{1-c_k^2} [\bar{\gamma}_{k-1} \ , \ J\bar{\bar{\varphi}}_{k-1}] \begin{bmatrix} 1 & c_k \\ c_k & 1 \end{bmatrix}, \tag{12}$$

where $J = \begin{bmatrix} 0 & \cdot & \cdot & \cdot & 0 & 1 \\ 0 & \cdot & \cdot & \cdot & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \cdot & \cdot & \cdot & 0 & 0 \end{bmatrix}$ of appropriate size.

The initialization of this algorithms is obtained from solving

$$\begin{bmatrix} 1 & 0 & \cdot & \cdot & \cdot & 0 \\ r_1 & 1 & 0 & \cdot & \cdot & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ r_N & 0 & \cdot & \cdot & 0 & 1 \end{bmatrix} \gamma_0 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \varphi_0 = \gamma_0 \tag{13}$$

Modified Parallel Levinson Algorithm

1. Start with

$$\begin{bmatrix} \gamma_0(0) & \varphi_0(0) \\ \vdots & \vdots \\ \gamma_0(N) & \varphi_0(N) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ -r_1 & -r_1 \\ \vdots & \vdots \\ -r_N & -r_N \end{bmatrix} \tag{14}$$

2. For $k = 1, \ldots, N$ let $c_k = \varphi_{k-1}(k)$ and compute

$$\begin{bmatrix} \gamma_k(0) & \varphi_k(0) \\ \gamma_k(1) & \varphi_k(1) \\ \vdots & \vdots \\ \gamma_k(k-1) & \varphi_k(k-1) \\ \gamma_k(k) & \varphi_k(k) \\ \gamma_k(k+1) & \varphi_k(k+1) \\ \vdots & \vdots \\ \gamma_k(N) & \varphi_k(N) \end{bmatrix} = \frac{1}{1-c_k^2} \begin{bmatrix} 0 & \varphi_{k-1}(0) \\ \gamma_{k-1}(0) & \varphi_{k-1}(1) \\ \vdots & \vdots \\ \gamma_{k-1}(k-2) & \varphi_{k-1}(k-1) \\ \gamma_{k-1}(k-1) & 0 \\ \gamma_{k-1}(k) & \varphi_{k-1}(k+1) \\ \vdots & \vdots \\ \gamma_{k-1}(N-1) & \varphi_{k-1}(N) \end{bmatrix} \begin{bmatrix} 1 & c_k \\ c_k & 1 \end{bmatrix} \tag{15}$$

3. Thus $R_N \gamma_N = e_N$ and $R_N \varphi_N = e_0$ where $e_0 = [1, 0, \ldots, 0]^T$.

This algorithm also requires $O(N)$ arithmetic operations on $O(N)$ processors. The communications are, however, simplified. Except for $c_k = \varphi_{k-1}(k)$, which has to be sent to the global memory in the beginning, and then to the local memories of each processor, the rest of communications is local. This is achieved, of course, on the expense of doubling the number of arithmetic operations in each local processor, but this is negligible because each processor is very powerful in particular to compare to the poor communications capabilities. We aim at increasing the locality of the computation and thus reducing the the communication overhead. (Note, though, that the number of required processors is reduced by half). Fig. 1 illustrates the parallel structure of the algorithm.

A general solution of linear system of Toeplitz equations $R_N x = f$ can be solved in parallel as follows (for derivation see [GKKL]).

1. Start with $x_0(0) = f_0$
   $x_0(j) = f_j - f_0 r_j, \;\; j = 1, \ldots, N$

2. For $k = 1, \ldots, N$ compute γ_k via Eq. 15 and with one step delay compute:

$$x_k(k) = x_{k-1}(k)\gamma_k(k),$$

$$x_k(j) = x_{k-1}(j) + x_{k-1}(k)\gamma_k(k), \;\; j = 0, \ldots, N \;\; j \neq k.$$

3 The Parallel Implementation

The parallel Toeplitz algorithms are implemented on a shared bus &shared memory MIMD (non-vector) multiprocessor using the software tool called Virtual Machine for MultiProcessors[VMMP].

3.1 The parallel multiprocessor

The parallel multiprocessor is an experimental shared memory multiprocessor built at the computer science department of Tel-Aviv University from commercial components. It was

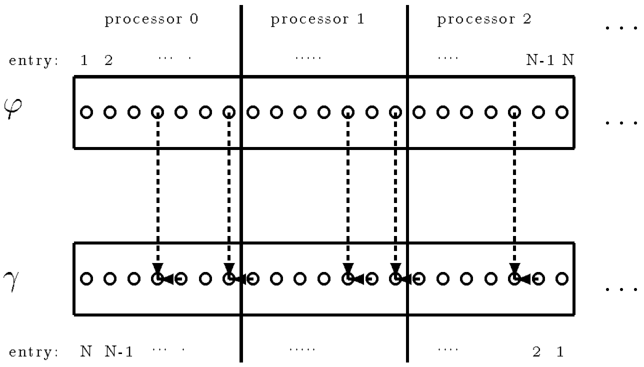


Figure 1: The implementation of the Modified Parallel Levinson algorithm

intended for research in software development for multiprocessors and not for hardware innovation.

The system consists of 8 National Semiconductor DB32532 processor boards connected by an industry standard bus (VME) to 8 MB of shared memory. Each processor board contains 4MB of local RAM. The parallel multiprocessor structure is illustrated in Fig. 2.

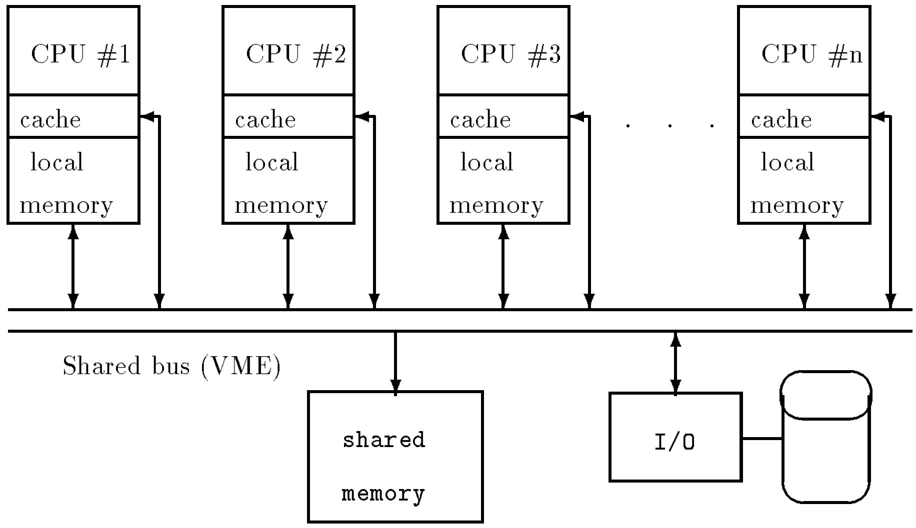


Figure 2: The Parallel Multiprocessor

3.1.1 The VMMP-Virtual Machine for MultiProcessors

The VMMP (Virtual Machine for MultiProcessors) is a software package, which provides a coherent set of services for parallel application programs running on diverse MIMD multiprocessors, both shared memory and message passing. It is intended to simplify parallel program writing and to promote portable and efficient programming. VMMP ensures high portability of application programs by implementing the same services on all target multiprocessors. It may also serve as a common intermediate language between very high level language compilers and various target multiprocessors.

VMMP supports two programming models: a virtual tree machine and a cooperating set of processes. Shared objects memory provides a flexible communication mechanism in addition to parameters and message passing. The combination of the VMMP services allows one to express many parallel algorithms naturally and easily. These, the same code can be implemented in a message passing machine using VMMP.

VMMP provides high level services for tree computations, crowd computations and shared objects memory. These high level services may be implemented efficiently on a wide variety of target machines. In addition, high level services ensure that their implementation details can be hidden effectively from the user. VMMP does not provide low level services, such as semaphores, because they are difficult to implement efficiently on all target machines (e.g. on a message passing multiprocessor).

VMMP may be used directly by the application programmer as an abstract parallel machine to aid portability without sacrificing efficiency. VMMP may also be used as a common intermediate language between high level programming tools, such as parallelizers and specification language processors, and the target machines.

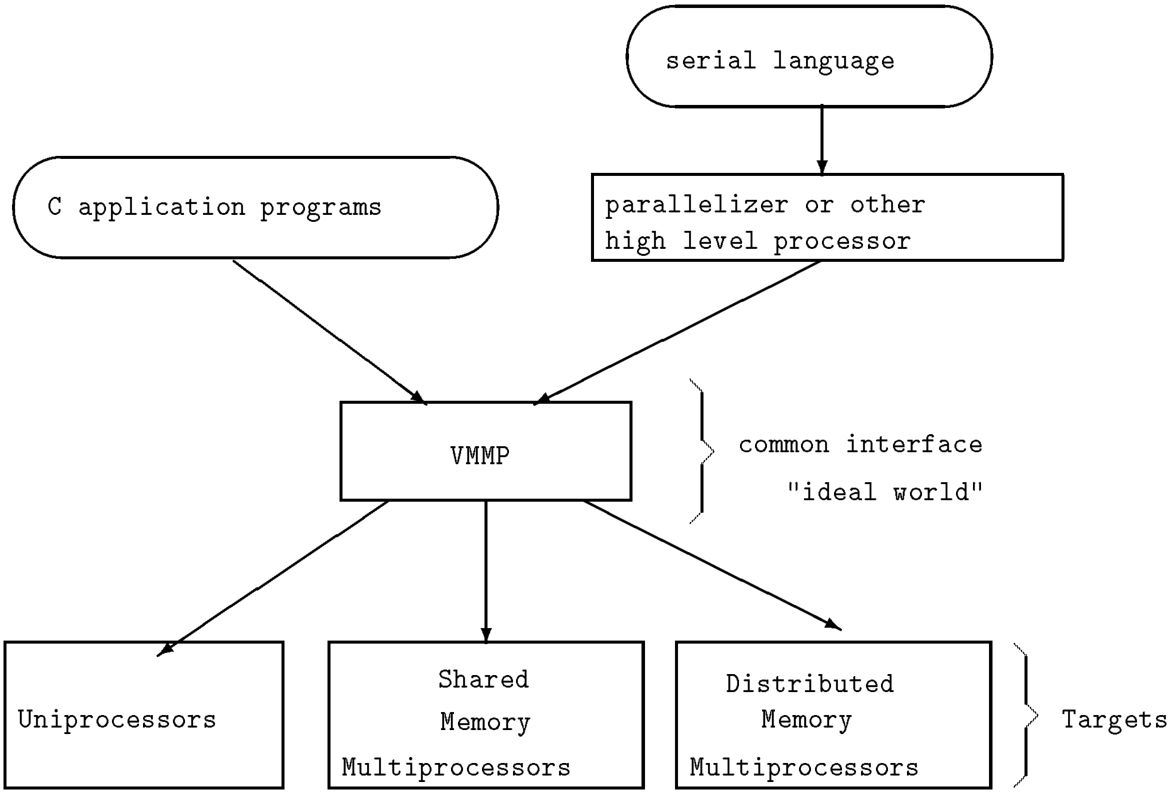


Figure 3: VMMP usage

VMMP Services

VMMP services follow closely the communication and synchronization requirements of parallel algorithms especially *tree computation* and *crowd computation*. VMMP provides a *virtual* *tree machine* to support tree computations and *crowd services* to support crowd computations. *Shared objects memory* provides an asynchronous and flexible communication mechanism for all computations. VMMP supports several data distribution strategies by some crowd operations and by shared objects operations. A crowd may create a tree computation and vice versa. Fig. 4 illustrates the relationship among VMMP services.

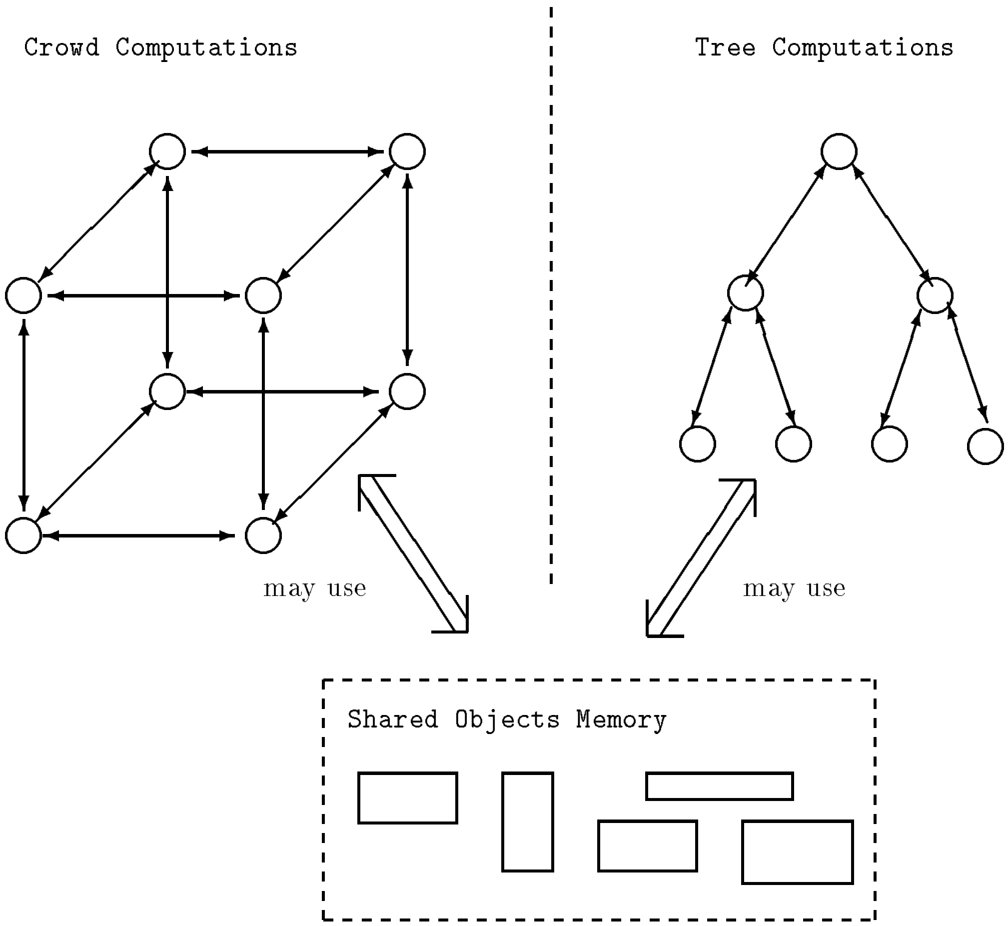


Figure 4: VMMP services

Tree computation- Tree computations are a general computation strategy, which solve the problem by breaking it into several simpler subproblems, which are solved recursively. The solution process looks like a tree of computations, in which the data flows down from the root into the leaves and solutions flow up towards the root. Some examples to tree computations are divide and conquer algorithms, game tree search and combinatorial algorithms.

Crowd computation- Many parallel algorithms are defined in terms of a set of cooperating processes. Each of these processes is an independent computational entity, which communicates and synchronizes with its peers. The processes communicate and synchronize mainly by message passing, which may be either blocking or non-blocking. In the following discussion these algorithms will be called *crowd algorithms*.

The communication structure of crowd algorithms normally follow some regular graph, such as ring, mesh, torus, hypercube or tree. Some algorithms change their communication structure during the computation and some require an unlimited communication between any pair of processes.

Message passing forces an implicit synchronization between the sending and the receiving processes, because the receiver normally waits for a message from the sender. Blocking send and receive operations imply a much stricter synchronization between the sender and the receiver.

Shared Objects Memory- VMMP implements a shared objects memory with the same semantics on all target multiprocessors. Shared objects may be regarded as virtually shared memory areas, which contain any type of data, and can be accessed only by secure operations. Shared objects can be created, deleted, passed by reference, read and written.

The shared objects are intended to distribute data, collect results and collect global results by some associative and commutative operations. Other uses of the shared objects, such as random updates to parts of a large object, are discouraged, because the implementation of these operations increase high overhead on a message passing multiprocessor.

Concurrent writes to the same shared object are permitted. The value of an object is the last value arrived at the *master copy* of the object. This value is then broadcasted to all other copies of the object (if any). A read following a write to the same object will return either the value written or a more recent value. It is possible that two distinct processors reading the same object at the same time will access different versions of the object value. This definition is necessary to guarantee consistent behavior also on message passing multiprocessors. A safe way to update shared objects is to apply an associative function, which guarantees correct result regardless of the application order.

The user must manipulate the shared objects only through VMMP services, which guarantee the correctness of the operations. The program must call VMMP explicitly after each change to the local copy of a shared object in order to broadcast the new value to other processors.

VMMP Implementation

VMMP is currently implemented on two shared memory multiprocessors and on three uniprocessors. VMMP was originally developed on MMX[G1,G2], an experimental shared memory multiprocessor built at Tel-Aviv University. VMMP has been later ported to ACE, a shared memory multiprocessor built at IBM T.J. Watson Research Center running Mach. VMMP has been ported also to three uniprocessors running versions of UNIX: SUN3 workstations, IBM RTs and on National Semiconductor's VME532 systems. Its implementation on a network of Transputers is currently on its way.

3.2 The specific parallel implementation

The algorithm is written in C using VMMP services. It supports any number of processors. The performance in shared memory & shared bus architecture has been degraded because of the need to access the global memory through the VME bus which is slower than the bus that is used by the local memory, and because access global memory by a processor while the bus is used by another processor cause the processor to wait until the bus is freed. Due to the above we tried to avoid a frequent use of the global memory by increase locality of computation.

To accomplish the above the following design decisions were taken into consideration:

1. Among the two types of tasks *global* and *local* we used the maximum number of local tasks because local task achieves better performance than a global task due to the fact that its stack is located in the local memory which belongs to the processor that executes this task not like the global task that located in the global memory. This is done using the VMMP services.

2. Priority was given to dynamic allocation of memory space in the local memory.

3. We tried to eliminate, as far as possible, the use of global variables.

4. In case a processor necessitates massive access to global memory, then the data in the global memory is copied to local memory, the computation is done in local memory and after completion of the computation the data is moved back to global memory.

5. We tried to eliminate, as far as possible, the use of message-passing among the tasks since message-passing is done through the use of global memory.

The implementation on the parallel machine using the VMMP software package is based on partitioning the vectors γ, φ and X into equal size slices, assign each slice to a processor, which computes the value of the components residing in its slice.

Data partitioning is done as follows:

1. Each processor computes slice of γ_k and φ_k in length of $\lceil \frac{2N+1}{p} \rceil$ (at most) and a slice of X_k (p is the number of processors in the system). If the slice of processor p_i contain the components i to j of γ then this processor will compute the components i to j of $J\varphi_k$ in order to prevent unnecessary broadcasts.

2. The partition of X_k is that the processor that computes $\gamma_k(-j)$ will compute $X_k(j)$.

4 Numerical Results and Performance Analysis

The Modified Parallel Toeplitz algorithm has been implemented on the parallel machine using the VMMP software package. For the purpose of parallel speedup measurements a serial program was executed on a single 32532 processor, without any system service call to VMMP. This serial algorithm is the fastest version we can derive.

The results given in Table 1 are for solving the Toeplitz system $R_N \gamma_N = e_N$ using the Modified Parallel Levinson Algorithm, and in Table 2 for the solution of the linear system $R_N x = f$ where f is any vector. The left column shows the size of the matrix. The other columns contain solution times for 1 to 8 processors. The speedup is defined as $\frac{serial\ time}{parallel\ time}$ and the efficiency is defined as $\frac{speedup}{number\ of\ processors}$.

The *speedup* and *efficiency* are given under each parallel time, while the serial time is given in the rightmost column. These measurements do not include I/O time.

Table 1: Processing time for solving the Toeplitz system $R_N \gamma_N = e_N$ using the modified parallel Levinson algorithm

Matrix size	Performance parameters	Parallel Processing 7 procs.	6 procs.	5 procs.	4 procs.	3 procs.	2 procs.	1 proc.	Serial Processing
512 × 512	Proc. time (sec.)	2.400	2.500	2.633	3.134	3.700	5.033	7.866	7.767
	Speed Up	3.24	3.11	2.95	2.48	2.10	1.54		
	Efficiency	0.46	0.52	0.59	0.62	0.70	0.77		
1024 × 1024	Proc. time (sec.)	7.766	8.666	9.534	11.500	14.233	19.934	31.433	31.167
	Speed Up	4.01	3.60	3.27	2.71	2.19	1.56		
	Efficiency	0.57	0.60	0.65	0.68	0.73	0.78		
2048 × 2048	Proc. time (sec.)	28.200	33.158	37.267	45.534	58.100	83.439	131.800	130.900
	Speed Up	4.64	3.95	3.51	2.88	2.25	1.57		
	Efficiency	0.66	0.66	0.70	0.72	0.75	0.79		
4096 × 4096	Proc. time (sec.)	103.434	119.534	141.067	174.033	223.667	318.667	511.600	505.100
	Speed Up	4.88	4.23	3.58	2.90	2.26	1.59		
	Efficiency	0.70	0.71	0.72	0.73	0.75	0.80		
8192 × 8192	Proc. time (sec.)	404.066	469.734	558.767	691.967	893.567	1275.267	2044.100	2023.367
	Speed Up	5.01	4.31	3.62	2.92	2.26	1.59		
	Efficiency	0.72	0.72	0.72	0.73	0.75	0.80		
16384 × 16384	Proc. time (sec.)	1593.766	1858.100	2198.066	2750.834	3544.300	4904.500	8211.006	8100.233
	Speed Up	5.08	4.36	3.69	2.95	2.29	1.65		
	Efficiency	0.73	0.73	0.74	0.74	0.76	0.83		

Table 2: Processing time for solving the Toeplitz system $Ax = f$ using the modified parallel Levinson algorithm

Matrix size	Performance parameters	Parallel Processing 7 procs.	6 procs.	5 procs.	4 procs.	3 procs.	2 procs.	1 proc.	Serial Processing
512 × 512	Proc. time (sec.)	1.967	2.180	2.200	2.500	3.067	4.500	6.467	6.334
	Speed Up	3.22	2.91	2.88	2.53	2.07	1.41		
	Efficiency	0.46	0.49	0.58	0.63	0.69	0.71		
1024 × 1024	Proc. time (sec.)	6.500	7.133	7.633	9.167	11.500	16.767	25.833	25.466
	Speed Up	3.92	3.57	3.34	2.78	2.21	1.57		
	Efficiency	0.56	0.60	0.67	0.70	0.74	0.76		
2048 × 2048	Proc. time (sec.)	22.167	25.033	28.134	34.666	44.934	66.367	103.466	102.564
	Speed Up	4.63	4.10	3.65	2.96	2.28	1.54		
	Efficiency	0.66	0.68	0.73	0.74	0.76	0.77		
4096 × 4096	Proc. time (sec.)	81.767	92.934	109.600	136.200	179.534	266.033	420.166	410.534
	Speed Up	5.02	4.42	3.75	3.01	2.29	1.54		
	Efficiency	0.72	0.74	0.75	0.75	0.76	0.77		
8192 × 8192	Proc. time (sec.)	315.433	366.200	437.867	543.700	719.166	1068.133	1680.200	1663.300
	Speed Up	5.26	4.53	3.79	3.05	2.31	1.55		
	Efficiency	0.75	0.76	0.76	0.76	0.77	0.78		
16384 × 16384	Proc. time (sec.)	1242.700	1442.033	1722.500	2149.100	2847.500	4087.266	6753.200	6610.800
	Speed Up	5.32	4.58	3.84	3.08	2.32	1.62		
	Efficiency	0.76	0.76	0.77	0.77	0.77	0.81		

The overhead from the use of the VMMP services is best demonstrated when we examine the performance of the modified parallel Levinson algorithm on a single processor.

Table 3: z and y values for the modified parallel Levinson algorithm (in thousands).

N	512	1024	2048	4096	8192	16384
z	0.3581	0.4841	0.6965	1.1120	2.1040	3.9065
y	0.0255	0.0256	0.0257	0.0260	0.0260	0.02600

We observe from Table 1 that the speedup increases with the problem size. This is due to the increase of the utilization of each processor. Said differently, as the problem size increases, the amount of time spent in synchronizing barriers decreases. Since the algorithm transfers data among global and local memories, the inefficiency of such transfer becomes insignificant when the problem size increases.

To get near peak performance from the machine we note that the delivered speedup is dramatically better for a larger matrices. In this case, the communication overhead is minimized because of the greater amount of work performed in the local processors. In addition, there is less traffic on the bus and less synchronization is required.

For each fixed $N = 512, 1024, 2048, 4096, 8192, 16384$ we consider the following processing timing model:

$$T_{p,N} = zPN + y\frac{N^2}{P} \tag{16}$$

where $T_{p,N}$ is the processing time for the Modified Parallel Levinson algorithm, and z is the overhead time for each of the N recursion steps as in Eq. 15 (e.g. the amount of time for passing around the value of $\varphi_k(k+1)$ plus the amount of time shifting N entries of γ_k is linear function of the number of processors). The constant y is the time needed for computing $\frac{N^2}{P}$ components of γ_k and φ_k (this is done N recursion times), and P is the number of processors.

For example (from Table 2), for $N = 512$

$$2.180 \approx T_3 = z \cdot 6 \cdot 512 + y \cdot \frac{512^2}{6}$$

Thus we have $T_{p,N} = zPN + y\frac{N^2}{P}$ for each N and $P = 1, 2, 3, 4, 5, 6, 7$.

Using Eq. (16) we assume equal time delays for all processors when they try to transmit data on the shared bus while another processor is occupying the bus. Under this assumption neither z nor y should depend on N or p. The values of z and y of Eq. (16) that provide the least squares fit to Table 1, for different values of N, are listed in Table 3.

The corresponding relative error $E_{p,N}$ is given by:

$$E_{p,N} = \frac{|T - zPN - y\frac{N^2}{P}|}{T} \tag{17}$$

where T is the corresponding real processing time. The values of E_p, for different values of p and N, are listed in Table 4.

Using Eq. (16) we can find P_{opt} for a given N:

$$\frac{\partial T}{\partial P} = zN - \frac{yN^2}{P^2} \Longrightarrow P_{opt} = \sqrt{\frac{y}{z}N}$$

Table 4: Relative errors for the modified parallel Levinson algorithm.

N	512	1024	2048	4096	8192	16384
p=1	0.062	0.058	0.055	0.049	0.048	0.043
p=2	0.175	0.140	0.145	0.145	0.150	0.114
p=3	0.094	0.092	0.105	0.114	0.119	0.115
p=4	0.038	0.051	0.058	0.006	0.070	0.068
p=5	0.024	0.028	0.019	0.003	0.006	0.003
p=6	0.015	0.044	0.059	0.076	0.076	0.072
p=7	0.137	0.123	0.145	0.152	0.172	0.162

Table 5: Optimal number of processors for the modified parallel Levinson algorithm.

N	256	512	4046	16384	5×10^4	10^5
P_{opt}	4	7	10	11	19	26

For example see Table 5.

Since Table 4 indicates good fit we can now use Eq. (16) and predict the processing time if the number of processors is optimal to a given N.

- T time is:

$$\begin{array}{llllll} If & N = 2028 & then & T = 25 & sec. & using\ 9\ proc. \\ If & N = 4096 & then & T = 89 & sec. & using\ 10\ proc. \\ If & N = 8192 & then & T = 347 & sec. & using\ 10\ proc. \\ If & N = 16384 & then & T = 1339 & sec. & using\ 11\ proc. \end{array}$$

During the implementation of this algorithm, the multiprocessor was upgraded from 4 week processors to 8 powerful one, and the bus was moved from MULTIBUS I to VME bus. Through this period we observe that the optimal number of processors needed to compute $10^4 \times 10^4$ matrix is 27. In other words, if we add more processors there is no improvement in the speedup. In the current upgraded multiprocessor the optimal number is 10. The difference comes from the fact that in the previous machine, there was a better balance between the computing power of each processor and the bandwidth of the bus. In the current system each processor is very powerful, and therefore there is no balance between CPU power and communication in this implementation.

We observe that processing time depends on N in a non-linear manner. The non-linearity in the speedup measurements can be understood for the parallel Levinson algorithm where $\gamma_k(k-j-1)$ and $\gamma_k(j-1)$ has to be sent to different processor and this done sequentially requiring $O(N)$ time units for each recursion steps. For the modified parallel Levinson algorithm the shifting of the components of γ in each iteration can be done simultaneously.

In general, the execution-time formula for the modified parallel Levinson algorithm is given by:

Let $T_{p,N}$ be the processing time for executing the modified parallel Levinson algorithm using p processors:

1. For $k = 1, \ldots, N$:

(a) t_1- time of reading $\varphi_{k-1}(k)$ into shared memory and sending it to each of the individual processors and the time of shifting vector γ_{k-1}, i.e.

$$\text{the vector } \gamma_{k-1} = \begin{bmatrix} \gamma_{k-1}(0) \\ \gamma_{k-1}(1) \\ \vdots \\ \gamma_{k-1}(N) \end{bmatrix} \text{ becomes } \gamma'_{k-1} = \begin{bmatrix} 0 \\ \gamma_{k-1}(0) \\ \vdots \\ \gamma_{k-1}(N-1) \end{bmatrix} \quad (18)$$

note that due to the shared bud architecture reading $\varphi_{k-1}(k)$ and shifting vector γ_{k-1} depend on the number of processors.

(b) t_2- time of computing the components $[\gamma_k(j), \varphi_k(j)]$. Thus for p processors $t_2\frac{N}{p}$ is how long it takes each processor to finish the job on $k-th$ recursion step. t_2 is the time of the following arithmetic

$$[\gamma_k(j)\ ,\ \varphi_k(j)] = \frac{1}{1-\varphi^2_{k-1}(k)}[\gamma_{k-1}(j-1)\ ,\ \varphi_{k-1}(j-1)]\begin{bmatrix} 1 & -\varphi_{k-1}(k) \\ \varphi_{k-1}(k) & 1 \end{bmatrix}$$

Thus the processing time $T_{p,N}$ is:

$$T_{p,N} = t_1 PN + t_2\frac{N}{p}N$$

The modified parallel Levinson algorithm that is described in this paper is fitted to this architecture and its features comprise a good match between the architecture (the increase of the locality of the computation) and the algorithm (shifting only the components of γ and not the components φ which reduces the amount of accesses to global memory). The locality of the computation and the need to shift only the components of γ enables this parallel algorithm to achieve fairly good speedup and hence demonstrates a good integration of the algorithm and the architecture.

From examination of the results and by the structure of the algorithm it is impossible to improve performance using the same method of shifting one vector γ in each iteration and transferring the entry $\varphi(k)$ using the global memory and the shared bus. In particular, we can see this in a small matrix where the domain of each processor is small and therefore the time of the local computation is short in comparison with the time spent on synchronization, message-passing among the processors and access to global memory.

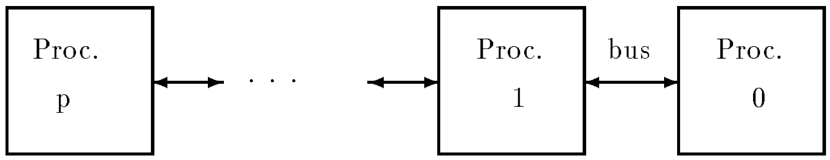
Since the partitioning of γ and φ is done as follow

$$
\begin{array}{c}
\textbf{Processor's slice} \\
processor\ \ 0 \\
\cdot \\
\text{--------------} \\
\vdots \\
processor\ \ i \\
\vdots \\
\text{--------------} \\
processor\ \ p \\
\vdots \\
\cdot
\end{array}
\quad
\left[\begin{array}{ll}
\textbf{Iteration} & \textbf{result} \\
\gamma_k(0) & \varphi_k(0) \\
\gamma_k(1) & \varphi_k(1) \\
\text{-------} & \text{--------} \\
\vdots & \vdots \\
\gamma_k(k-1) & \varphi_k(k-1) \\
\gamma_k(k) & \varphi_k(k) \\
\text{-------} & \text{--------} \\
\gamma_k(k+1) & \varphi_k(k+1) \\
\vdots & \\
\gamma_k(N) & \varphi_k(N)
\end{array}\right]
\Longleftarrow
\left[\begin{array}{ll}
\textbf{Iteration} & \textbf{input} \\
0 & \varphi_{k-1}(0) \\
\gamma_{k-1}(0) & \varphi_{k-1}(1) \\
\text{-------} & \text{--------} \\
\vdots & \vdots \\
\gamma_{k-1}(k-2) & \varphi_{k-1}(k-1) \\
\gamma_{k-1}(k-1) & 0 \\
\text{-------} & \text{--------} \\
\gamma_{k-1}(k) & \varphi_{k-1}(k+1) \\
\vdots & \\
\gamma_{k-1}(N-1) & \varphi_{k-1}(N)
\end{array}\right]
$$

where the number of data items for each processor are equal (± 2). Indicates that the value of $\varphi_{k-1}(k)$ must be transmitted from one processor to all the others, and shifting the components of γ is done via the global memory for components that are residing on the border of the processor slice, it seems that better performance will achieved on massage passing machine as shown bellow.

Moreover, increasing the number of processors will increase the number of broadcasts of data and thus we do not get a linear speedup (recall that the processors are connected by one shared data bus).

This indicates that using the same algorithm on a parallel massage passing machine where the processors are connected in line such as will give a linear speedup results since



the most consuming time part of the synchronization (shifting γ) will be done in a constant time regardless the number of processors using the local connection between the adjacent processors and their will be no dependence on the number of processors for communication overhead, i.e processing time will be:

$$T_{p,N} = t_1 N + t_2 \frac{N}{p} N$$

5 Summary

We have presented and implemented an efficient algorithm for solving Toeplitz system on a shared bus& shared memory parallel MIMD machine.

One of our main observations is that the ability to obtain a substantial speedup depends on the matrix size. For large matrix the efficiency is over 75%.

We observe that in order to get a speedup the size of the matrix should be large enough and that their is an optimal number of processor to a given matrix such that the speedup level will increase when increasing the number of processors.

Since the shared bus & shared memory multiprocessor is just one representative of the class of MIMD parallel machines, it would be useful to implement this algorithm on other types of multiprocessors, especially on a 1-D (linear) as explained in Sec. 4 since the most consuming time part of the synchronization will be done in a constant time regardless to the number of processors using the local connection between the adjacent processors. In this case, using the VMMP services will provide a smooth and easy implementation of our algorithm also on general message passing machine.

References

[L] N. Levinson (1947), The Wiener RMS error criterion in filter design and prediction, *J. Math. Physics, 25, pp. 261-270.*

[D] J. Durbin (1960), The fitting of time-series models, *Rev. Int. Inst. Statist., 28, pp. 233-243.*

[B] J. Bunch, Stability of methods for solving Toeplitz systems of equations, *SIAM J. Sci. Stat. Comput., (6:2), pp. 349-364.*

[KL] I. Koltracht, P. Lancaster (1988), Threshold algorithms for the prediction of reflection coefficients in a layered medium, *Geophysics, vol. 53, pp. 908-919.*

[GKKL] I. Gohberg, T. Kailath, I. Koltracht, P. Lancaster (1987), Parallel algorithms of linear complexity for machines with recursive structure, *Lin. Alg. Appl., 88, pp.271-315.*

[KH] S.Y. kung, Y.H. Hu (1983), A highly concurrent algorithms and pipelined architecture for solving Toeplitz systems, *IEEE Trans. ASSP, 31, no. 1.*

[K] I. Koltracht (1987), The Levinson algorithms on the CYBER 205, Cuper-C, *Newsletter, v.3:2, pp. 4-10.*

[G1] E. Gabber (1987), The MMX Parallel Operating System and Its Processor, *Dept. of Computer Science, Tel-Aviv University.*

[G2] E. Gabber (1987), MMX - Multiprocessor Multitasking eXecutive User's Manual *Dept. of Computer Science, Tel-Aviv University.*

[VMMP] E. Gabber, "A Virtual Machine for the Development of Portable and Efficient Programs for Multiprocessors", *Proc. of the 1989 int. conf. on parallel processing .*