Parallel Computation of Autocorrelation Matrices on a Limited Number of Processors

S.R. Subramanya Abdou Youssef
Department of Electrical Engineering and Computer Science,
The George Washington University,
Washington, DC 20052.
{subra,youssef}@seas.gwu.edu

Abstract

Autocorrelation matrices are used heavily in several areas including signal and image processing, where parallel and application-specific architectures are also being increasingly employed. Therefore, an efficient scheme to compute autocorrelation matrices on parallel architectures has considerable benefits.

In this paper, a parallel algorithm for the computation of autocorrelation matrices on a limited number of processors (compared to the matrix size) is presented. The computational requirements for the elements of the autocorrelation matrices are highly skewed and the proposed algorithm gives a scheme for even distribution of the computation load among the processors, resulting in better utilization of the processors and minimization of computation time. The algorithm is developed for an architecture used in several existing computers.

1 Introduction

The computation of the autocorrelation matrix is central to several applications including signal and image processing. For example, it is used for the computation of the coefficients of the ARMA (autoregressive moving average) model, used for modeling stationary signals [4]. Non-stationary signals are sometimes approximated by considering windows of the signal and modeling the signal windows as a stationary signal with suitable parameters.

Given a matrix $X = (x_{i,j})_{0 \le i,j \le N-1}$, the autocorrelation matrix is given by $A = (a_{k,l})_{0 \le k,l \le N-1}$, where,

$$a_{k,l} = \frac{1}{(N-k)(N-l)} \sum_{i=0}^{N-1-k} \sum_{j=0}^{N-1-l} x_{i,j} \cdot x_{i+k,j+l}$$

The computation requirements of the autocorrelation matrix elements is highly skewed - $A[0,0]$ takes N^2 multiplications and $N^2 - 1$ additions, while $A[N-1, N-1]$ takes just 1 multiplication and no addition, and the number of steps required for the remaining elements is something in between. In a parallel algorithm, if the computation of each $A[i,j]$ is assigned to every processor (in the case N^2 processors are available), or if the matrix is partitioned and assigned to processors in a straight-forward manner (when less than N^2 processors are available), then it results in a high imbalance of load on the processors, resulting in poor utilization of processors and increased computation time.

In [1], an algorithm for the computation of the autocorrelation matrix on 2-D meshes was presented, which had load-balancing embedded in the algorithm, and it was shown that the algorithm therein had a speedup of twice over a straight-forward algorithm and was within twice of the (trivial) lowerbound (optimal). The algorithm proposed in [2] for 2-D meshes achieves the optimal time for the matrix computation phase, using an offline algorithm to evenly distribute work among the processors in the mesh. Both the algorithms assume the availability of a 2-D mesh of size $N \times N$ equal to the size of the matrix, which could be restrictive. For example, even if the matrix is 100×100, the number of PEs required in the mesh is $10,000$ which may not be feasible with current technologies.

In this paper, this restriction is relaxed and an algorithm is developed for the computation of the autocorrelation matrix on a limited number of processors, $P : 1 < P < N^2$. Generally $P \ll N^2$.

The next section describes the architecture and gives a lowerbound on the matrix computation and the assumptions of the algorithm. Section 3 describes the proposed algorithm and the simulation results. Conclusions are presented in Section 4.

2 Architecture, Computation Requirements and Assumptions

2.1 Architecture

The proposed algorithm assumes the availability of a *host computer* to which a mesh of *processing elements* (PEs) is connected by a bi-directional communication link. This architecture is exemplified by several existing machines such as Intel Touchstone Delta, Paragon, and Ametek Series 2010.

The host is responsible for the following functions: (1) Receiving (or generating) the input matrix X, (2) Transmitting the matrix X to the PEs, (3) Partitioning the work among the PEs, and (4) Receiving the results of computation from the PEs and placing them appropriately in the resulting matrix A. The actual computation of the autocorrelation matrix is done by the mesh of PEs.

Each PE in the mesh has a very simple structure – a small local memory and basic *computation functions* such as *add* and *multiply*. It also has a *router* which handles very basic *communication functions* such as *send* and *receive*, (to and from the directly connected neighbors). The computations and communications in the mesh can be done concurrently. This is similar to the architecture described in [3].

2.2 Computation requirements and lowerbound

The computation of $A[k,l]$ requires the multiplication of the corresponding elements of two rectangular blocks of elements in X, of height $(N-k)$ and width $(N-l)$, one with its top-left corner at (k,l) and the other with its top-left corner at $(0,0)$, and the subsequent addition of those product terms. This entails $(N-k)(N-l)$ multiplications and $(N-k)(N-l)-1$ additions, for a total work of $2(N-k)(N-l)-1$. The computation requirement of two arbitrary elements is pictorially shown in Figure 1 below.

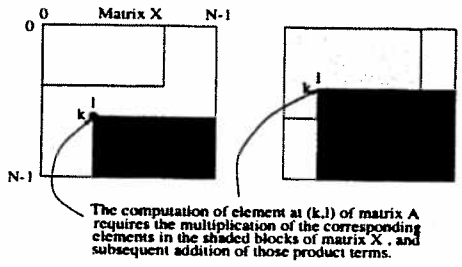

Figure 1: Computation requirements of two arbitrary $A[k,l]$s

The total *work* (multiplication and addition steps) required to compute the whole matrix is: $\sum_{k=0}^{N-1}\sum_{l=0}^{N-1}[2(N-k)(N-l)-1] = 2\left[\frac{N(N+1)}{2}\right]^2 - N^2$.

Assuming P PEs are available and all PEs are busy in computation, the average work load per processor is: $\lceil\left(\frac{N^2(N+1)^2}{2} - N^2\right) / P\rceil$.

The above quantity is a trivial lower bound on the matrix computation, and is denoted as the *balanced load*, W_B in our algorithm.

2.3 Assumptions

The matrix size in each dimension, N, is assumed to be even (the odd case is handled easily by a slight modification). The number of PEs P is such that $2 \leq P \leq \lfloor\frac{(N+1)^2}{2}\rfloor$. Note that P is at least 2, for otherwise, it reduces to the simple uniprocessor case. P is at most $\lfloor\frac{(N+1)^2}{2}\rfloor$ to ensure that the work done by a PE is always more than the work required to compute any $A[k,l]$. For $P > \lfloor\frac{(N+1)^2}{2}\rfloor$, the algorithm proposed in [2] could be used with slight modifications.

The physical topology of the P PEs is assumed to be a 2-D mesh of dimensions $n \times m$ such that the perimeter $2(n+m)$ is minimum for a given P (it is 'tightest' possible rectangle). (Note that the topology is a linear chain if P is prime.) In the algorithms, the PEs are indexed by $p \in \{0 \ldots P-1\}$. The indices in the individual dimensions $(i,j), 0 \leq i < n, 0 \leq j < m$, of the mesh can easily be derived from p.

In the determination of the computation and communication time, an arithmetic operation (add and multiply), and communication function (send and receive), each is assumed to take one step.

3 The Proposed Algorithm

In this section, we first give a brief overall description of the proposed algorithm, followed by the complexity analysis, and the performance evaluated by simulation.

3.1 Brief description of the algorithm

There are four main phases in the algorithm, which are described in the following subsections.

3.1.1 Distribution of elements of X by the host to the PEs

We assume that the whole matrix X is sent in a pipelined fashion from the host to one of the PEs and from there it is broadcast to all the other PEs in the mesh. This is shown to take $(n+m+N^2-2)$ steps [2].

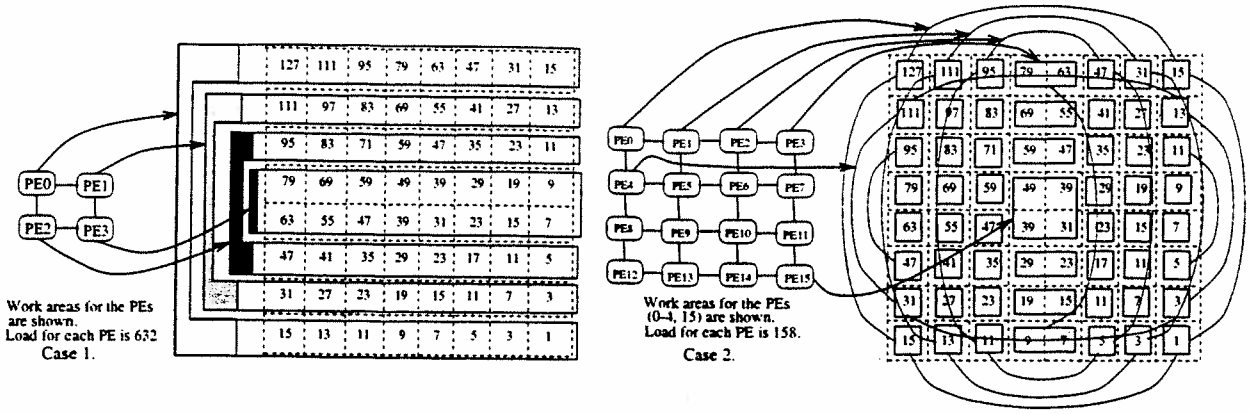

Figure 2: Distribution of work for 8×8 matrix among PEs for cases 1 and 2

3.1.2 Work partitioning among PEs by the host

This phase is executed by the host *offline* (and just once) for any given N and P (similar to the compilation of programs; thereafter, the matrix computation can be carried out any number of times). Based on the matrix size N and the number of PEs in the mesh P, the host partitions the work among the PEs in the mesh in a balanced way. We distinguish three 'work cases' as given below:

case	*Value of P*
1	P is a divisor of $\frac{N}{2}$.
2	P is a divisor of $\frac{N^2}{4}$.
3	P is other than in cases 1 and 2.

Based on the work case, the host determines the work required to be done by each PE and creates a tuple $\mathcal{W}$ containing the work description. The host then forms a message $\mathcal{V} = \langle \mathbf{case}, \mathcal{W} \rangle$, where **case** is the 'work case' and sends to the PE.

Work cases 1 and 2

For cases 1 and 2, the work is partitioned among the PEs in a perfectly balanced way, with each PE doing exactly W_B amount of work. In these two cases, the work description tuples $\mathcal{W}$ for the PEs are empty, since, based on the knowledge of just N and P, each PE can independently determine the elements of A that it should compute. Figure 2 describes the work areas for the PEs to compute an 8×8 matrix for cases 1 and 2, with 4 and 16 PEs respectively. The numbers in the grid represent the amount of work required to compute the element in that position of the autocorrelation matrix.

Work case 3

For case 3, the work description tuple for a PE, $\mathcal{W} = \langle i_1, j_1, W_1, i_2, j_2, W_2, f \rangle$ states that the PE needs to do:
(1) W_1 amount of (possibly *partial*) work toward the computation of $A[i_1, j_1]$,
(2) Complete computation of elements $A[i_1, j_1 + 1] \ldots$ etc., upto the element before $A[i_2, j_2]$, and
(3) W_2 amount of (possibly *partial*) work toward the computation of $A[i_2, j_2]$.

The possible scenarios for W_1 and W_2 are shown in Fig. 3. For the computation of an element, say $A[i_1, j_1]$, the complete rectangles in the figure represent the sub-rectangles in the matrix X with height $(N - i_1)$ and width $(N - j_1)$ with the top–left corner at (i_1, j_1), whose elements need to be multiplied with the corresponding elements of a similar sub-rectangle with the top–left corner at $(0, 0)$, and all those product terms to be added. The shaded areas represent the 'work areas' which indicate that only elements in that region of the sub-rectangle are used in the computation of the element by a PE. In case W_1 is only part of the computation done by a PE, say $PE(p)$, then the partial result is sent to $PE(p-1)$, where it is added to the partial result of its W_2, to obtain the element of A. This is handled by setting the flag f in $\mathcal{W}$.

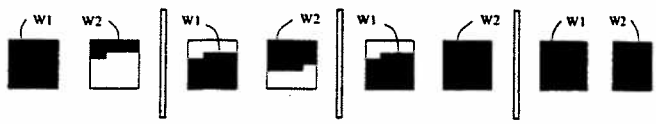

Figure 3: Scenarios for W_1 and W_2 for a particular PE

The work partitions for case 3, to compute a matrix of size 8×8 and with 12 PEs organized as a mesh of size 4×3 are shown in Fig. 4 and the work description tuples for the PEs are tabulated in Table 1.

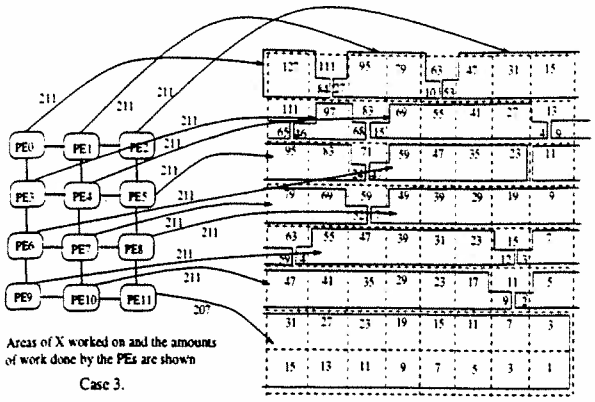

Figure 4: Distribution of work for 8 × 8 matrix among 12 PEs for case 3

PE	$\mathcal{W} = \langle i_1, j_1, W_1, i_2, j_2, W_2, f \rangle$
0	$\langle 0, 0, 127, 0, 1, 84, F \rangle$
1	$\langle 0, 1, 27, 0, 4, 10, T \rangle$
2	$\langle 0, 4, 53, 1, 0, 65, T \rangle$
3	$\langle 1, 0, 46, 1, 2, 68, T \rangle$
4	$\langle 1, 2, 15, 1, 7, 4, T \rangle$
5	$\langle 1, 7, 9, 2, 2, 24, T \rangle$
6	$\langle 2, 2, 47, 2, 6, 23, T \rangle$
7	$\langle 2, 7, 11, 3, 2, 52, F \rangle$
8	$\langle 3, 2, 7, 4, 0, 59, T \rangle$
9	$\langle 4, 0, 4, 4, 6, 12, T \rangle$
10	$\langle 4, 6, 3, 5, 6, 9, T \rangle$
11	$\langle 5, 6, 2, 7, 7, 1, T \rangle$

Table 1: Work description tuples for case 3 to compute 8 × 8 matrix using 12 PEs

After the host partitions the work among the PEs, the tuples of work descriptions are sent to the PEs. The tuple for every PE describes exactly what elements of A to compute (which areas of the X matrix to work on). Then the actual computation proceeds.

3.1.3 Computation of elements of A by the PEs

During the matrix computation phase, the actual computation of the terms required for the elements of the autocorrelation matrix A are computed. Before the start of this phase, each PE would have received $\mathcal{V}$ which contains the case and the tuple $\mathcal{W}$ which contains the complete work description (what elements to compute) for the PE. In this phase, all the PEs do their work independently and concurrently. There is at most one communication required by a PE in case W_1 is a partial computation (as described in Sec. 3.1.2) to send the partial result to its predecessor, at the end of computation.

3.1.4 Gathering of elements of A (result) by the host from the PEs

The algorithm assumes that, after a PE finishes computing all the elements that it is required to compute, it sends the list of results $\mathcal{R}$ to the host. $\mathcal{R} = \{[(i_1, j_1)], e_1, \ldots, e_n\}$ is an ordered list, where $e_1, \ldots, e_n$ are the elements of A computed by the PE. The positions of $e_1, \ldots, e_n$ in A is determined by the host based on the knowledge of the work case and the PE from which $\mathcal{R}$ was received. Only in the third case, (i_1, j_1) at the header of $\mathcal{R}$ denotes the row and column of the first element in the list (in cases 1 and 2, it is empty).

Note that if N^2 items are to be sent from the mesh to the host, and if every PE has at least one item, then the transfer of all the N^2 items can be done in N^2 time steps, using pipelining [2]. The host receives $\mathcal{R}$ from each PE then places the results in the appropriate positions in the A matrix.

The PEs could also send the results to the host as and when they finish the computation of an element and then proceed with the computation of the next one. By suitable scheduling of the communication of these results, the communication and computation can be overlapped and the overall time could be reduced. This, however, is not addressed in this paper.

3.2 Pseudocode of the proposed algorithm

The detailed pseudocode of the proposed algorithm is given in [2], in a top-down fashion. The upper levels enable easy comprehension of the overall scheme and the lower levels facilitate easy implementation of the algorithm in software and/or hardware. Only the topmost level is presented here.

Algorithm 3.1
COMPUTEAUTOCORRMATRIX
(in: X, N, P**, out:** A**)**
1. **begin**
   {**Concurrently Do** *2, 3, and 4*}
2. BROADCAST_X_TOMESH(X, N). {*by host*}
3. INIT_W_MATRIX(w, N). {*by mesh of PEs*}
4. DETERMINEWORKFORPEs(X, N, P). {*Host*}
   {**endDo**}
5. DOCOMPUTATION($X, N, P, \mathcal{V}$). {*by Mesh*}
6. GATHERRESULTS. {*Done by host*}
7. **end**

Complexity analysis: Each of the steps above has been completely described and analyzed in [2]. Steps 2, 3, and 4 take $n + m + N^2 - 2$, N^2, and $\mathcal{O}(N^2)$

time. Note that step 4 of the algorithm is done *offline* and just once, for any given value of N and P. Steps 5 and 6 take $\lceil \left(\frac{N^2(N+1)^2}{2} - N^2 \right) / P \rceil$ and N^2 steps, respectively.

3.3 Simulation results

The proposed algorithm has been simulated by a C program running on Sparcstation, for various values of N (matrix size is $N \times N$) and number of PEs, P to determine (1) the required communication steps, (2) time for partitioning work among the PEs, and (3) the time for actual computation of the autocorrelation matrix. The simulation results have been shown to agree with the analytical results [2].

As noted in Sec. 3.1.2, for cases 1 and 2, the load on all the PEs is perfectly balanced, with each PE doing the same amount of work, W_B. However, in case 3, the last PE, $PE(P-1)$ does less than W_B work, with all the rest doing W_B amount of work. We define the *imbalance*, $\mathcal{I} = \frac{W_B - W_{last}}{W_B} \cdot 100$, where W_{last} is the work done by $PE(P-1)$.

Note: The imbalance will be zero for cases 1 and 2. Imbalance occurs only in case 3, and the worst-case imbalance occurs when the number of processors, $P = \lfloor \frac{(N+1)^2}{4} \rfloor$, (see [2]).

A plot of the worst-case $\mathcal{I}$ against several values of N is shown in Fig. 5.

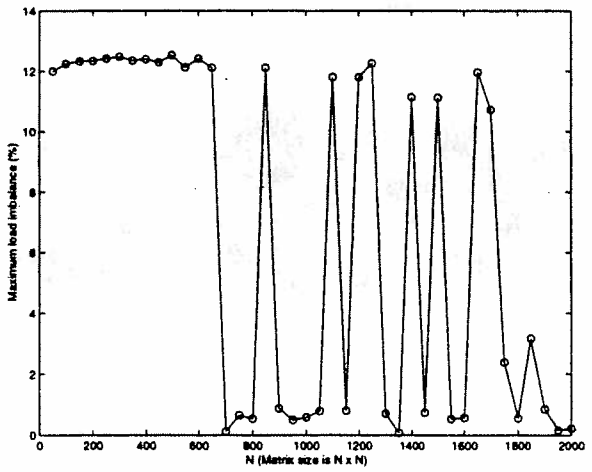

Figure 5: Worst-case imbalance

4 Conclusions

In this paper, a parallel algorithm for the computation of autocorrelation matrix on a limited number of processors (compared to the matrix size) was presented. Although the computation requirements for the elements of the autocorrelation matrix is highly skewed, the proposed algorithm evenly distributes the computation load among the processors, resulting in better utilization of the processors and minimization of computation time. The algorithm was developed based on the architecture of several existing machines. The computation and communication complexities were analyzed. The algorithm was simulated and the simulation results were found to agree with analytical results. The pseudocode based on the algorithm presented in a top-down fashion in [2] facilitates easy comprehension and also easy implementation in software and/or hardware.

References

[1] S.R.Subramanya. 'A Parallel Algorithm with Embedded Load Balancing for Autocorrelation Matrix Computation', *Int'l. Symposium on Parallel Architectures, Algorithms and Networks*, Taiwan, Dec. 1997.

[2] S.R.Subramanya. 'A Parallel Algorithm for Autocorrelation Matrix Computation on 2-D Meshes', *Technical Report*, IIST, George Washington University, 1997.

[3] Hwang, K. *Advanced Computer Architecture*, McGraw-Hill, 1993, pp370–375.

[4] Brockwell, P.J. and Davis, R.A. *Time Series: Theory and Methods*, Springer-Verlag, 1991.