MAPPING SIGNAL PROCESSING ALGORITHMS ON PARALLEL ARCHITECTURES

By

NIDAL M. SAMMUR

Bachelor of Science
The University of Tulsa
Tulsa, Oklahoma
1984

Master of Science
The University of Tulsa
Tulsa, Oklahoma
1986

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
July, 1992

MAPPING SIGNAL PROCESSING ALGORITHMS

ON PARALLEL ARCHITECTURES

Thesis Approved:

Martin T Hagan

Thesis Advisor

C. M. Bacon

Huizhu Lu

Dean of the Graduate College

ACKNOWLEDGMENTS

I thank God for the persistence, determination, and faith He gave throughout my years of study. Like everyone who has reached this point in his education, I could not have gotten here without the help, encouragement, and support of a lot of people.

My deepest love, thankfulness and appreciation goes to my mother. She always stood by me, supported me, and inspired me. I dedicate this dissertation to her. The most important contributor to my success, however, did not live to see me complete my education - my father. He always stressed the value of education and hard work - this dissertation could not have been done without that example. My sincere love and appreciation goes to my sisters and my brother and the rest of my family.

I would like to express my appreciation to my adviser and friend Dr. Martin Hagan for the guidance and insight he provided me throughout my study. His delightful and informative discussions, his views, and his approaches added a depth to my understanding which could not have been obtained elsewhere. I have known Dr. Hagan for more than ten years and he never stopped amazing me. He is so knowledgeable, friendly, honest, and yet humble. Thank you Dr. Hagan for being such a wonderful friend and teacher. You are simply the best.

I would like to thank Dr. Bacon, Dr. Teague, Dr Baker, and Dr. Lu for serving on my committee.

I would like to acknowledge the machine time, technical support, and encouragement given by NASA/Goddard Space Flight Center and various members of its staff, including James Fischer and Judy Devaney. I would like to acknowledge the National Center for Supercomputing Applications and the Los Alamos National Laboratory for allowing us time on their machines.

Finally, I would like to thank all my wonderful friends in Stillwater and Tulsa for all their help, encouragement, and support.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Digital signal processing is a field of study concerned with the processing of information represented in digital form. Certain techniques in the field can be traced back to numerical algorithms performed in the seventeenth and eighteenth century. However, the advent of modern high-speed digital computing devices has caused a revolution in applications of the theory to a variety of problems. Signal processing is used in such areas as biomedical data processing [1], sonar and radar processing [2], speech processing [3], data communication [4], seismic signal processing [5], adaptive system identification [6], adaptive control applications [7], and a host of other applications [8-11]. One of the most interesting aspects of digital signal processing is this wide variety of applications. This has served to create a vitality in the field that is often missing in other scientific fields of study.

Digital signal processing has become an increasingly significant field because of the technology associated with digital computers. A digital computer used to process signals offers a tremendous advantage in flexibility. The emergence of parallel processing and very large scale integration (VLSI) motivated the researchers in the field of digital signal processing to find and explore new ways to

implement and design efficient and highly parallel algorithms [11-18].

The conventional sequential digital computers suffer from one serious drawback: the von Neumann bottleneck. This phenomenon accounts for the sometimes slow and inefficient use of conventional serial processor resources. In a sequential computer, a single memory buffer serves as the only gate between the high-speed memory and the central processing unit (CPU). This makes it necessary to organize all computational tasks in a strictly sequential fashion. Processing speed is limited not by the speed of the CPU but the narrow pathway between the CPU and memory. Parallelism is one of the major innovations in the hardware design of digital computers that have permitted the circumventing of the von Neumann bottleneck so as to attain high speeds [19-20].

A parallel processing computer, simply defined, is one that can perform operations using more than one processor simultaneously. The central problem parallel processing systems face is how to effectively and efficiently use more than one processor at the same time. The effectiveness of the system depends on whether one can identify a problem that lends itself to parallelism, determine the algorithm, and map it onto a suitable architecture. There are no established principles revealing how to automate the arduous manual task of partitioning any real-world problem so that it can be parcelled out to many processors simultaneously [21-22].

The main objective of this research is to explore the different techniques of mapping digital signal processing algorithms onto advanced computer architectures. It is impossible to cover all

algorithms and all architectures. The spectrum of the algorithms covered was limited to those which can be characterized as one dimensional, batch, and time domain. As for the architectures, the availability of such systems was the major limitation. The goal of this research is to discover the types of computer architectures which are best suited for signal processing.

This dissertation contains six chapters each of which is dedicated to present a concise set of ideas. The rest of this chapter concentrates on presenting a preliminary system identification theory that sets the stage for the second chapter. Chapter II discusses the batch algorithms that are used for inverse filtering and concentrates on the similarities between these algorithms. An algorithm is selected for parallel implementation since it is a good representative of this group of algorithms. Chapter III presents the different advanced computer architectures and discusses in detail those architectures that are used in this research. Chapter IV discusses the implementations of the algorithm chosen on a selected number of advanced computer architectures. Chapter V contains the performance analysis performed on the results obtained in chapter IV. A ranking of the machines is presented. Finally, chapter VI summarizes the main ideas presented in this work followed by some general conclusions.

Inverse Filtering

A problem of great importance is determining the parameters of a model given observations of the physical process being modeled

[23]. In control theory this problem is often called the system identification problem [6]; one of the most important applications of identification methods is adaptive estimation and control. Parameter identification problems also arise in several digital signal processing algorithms; applications include seismic signal processing and the analysis, coding, and synthesis of speech. In seismic signal processing the problem is termed deconvolution [3], while it is termed linear prediction in speech processing [5]. Other names have been used like parameteric spectrum analysis and inverse scattering [8-9]. Inverse filtering is a more natural term to use and is adopted throughout this report.

In this report we address the problem of inverse filtering for a particular underlying time series model, namely the autoregressive (AR) process . The general form for an AR process of order p is given by equation (1-1).

$$z_t = \phi_1 z_{t-1} + \ldots + \phi_p z_{t-p} + e_t \qquad (1\text{-}1)$$

The current value of the process z_t is expressed as a weighted sum of past values plus a random white noise e_t with a variance of σ_e^2. Thus z_t can be considered to be regressed on the p previous z's, hence the name. The weights on the previous z's are called the AR parameters. The right hand side of equation (1-1), excluding the white noise, is called the prediction of z_t based on z_{t-1} thru z_{t-p}, and the white noise is termed the prediction error. The inverse filtering problem is summarized as follows: given the set of

observations for the process, or the time series z_t, find the underlying AR parameters that best characterize the process.

The AR process given by equation (1-1) can be represented in a block diagram form as shown in Figure 1.1. The blocks with z^{-1} indicate unit delay. The structure depicted here is sometimes referred to as a tapped delay line or direct form I implementation.

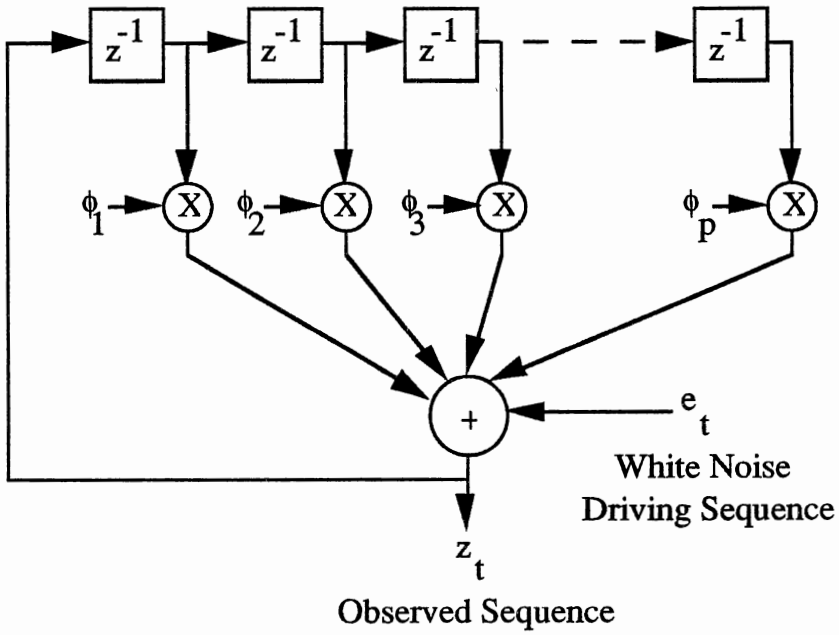


Figure 1.1 Autoregressive Process of Order p

It should be noted that an AR process has an all-pole transfer function given by equation (1-2). This property justifies the use of AR processes to model spectra with sharp peaks.

$$H(z) = \frac{1}{1 + \phi_1 z^{-1} + \phi_2 z^{-2} + \ldots + \phi_p z^{-p}} \qquad (1\text{-}2)$$

A plethora of inverse filtering methods are discussed in the literature which are based on the autocorrelation function, the partial autocorrelation function, and the generalized partial autocorrelation function [24-25]. The methods based on the first two functions are of major interest in this research.

The autocorrelation function of the AR process given by equation (1-1) can be determined by first multiplying equation (1-1) by z_{t-k} to get equation (1-3), then taking the expected values to get equation (1-4).

$$z_{t-k} z_t = \phi_1 z_{t-k} z_{t-1} + \ldots + \phi_p z_{t-k} z_{t-p} + z_{t-k} e_t \qquad (1\text{-}3)$$

$$\gamma_k = \phi_1 \gamma_{k-1} + \phi_2 \gamma_{k-1} + \ldots + \phi_p \gamma_{k-p} + \begin{cases} \sigma_e^2 & , k=0 \\ 0 & , k>0 \end{cases} \qquad (1\text{-}4)$$

If the estimated order of the process is denoted by m we can rewrite equation (1-4) with an extra index for the ϕ's as shown in equation (1-5).

$$\gamma_k = \phi_{m1} \gamma_{k-1} + \phi_{m2} \gamma_{k-1} + \ldots + \phi_{mm} \gamma_{k-m} + \begin{cases} \sigma_e^2 & , k=0 \\ 0 & , k>0 \end{cases} \qquad (1\text{-}5)$$

The parameter ϕ_{mm} is called the partial autocorrelation function at lag m. If we let k vary from 0 to m in equation (1-5) we get a set of

linear equations for the ϕ's in terms of the γ's. Equation (1-6) shows the resulting equations organized within matrices.

$$\begin{bmatrix} \gamma_0 & \gamma_1 & \cdots & \gamma_m \\ \gamma_1 & \gamma_0 & \cdots & \gamma_{m-1} \\ \vdots & \vdots & & \vdots \\ \gamma_m & \gamma_{m-1} & \cdots & \gamma_0 \end{bmatrix} \begin{bmatrix} 1 \\ \phi_{m1} \\ \vdots \\ \phi_{mm} \end{bmatrix} = \begin{bmatrix} \sigma_e^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \qquad (1\text{-}6)$$

Equation (1-6) is called the Yule-Walker or normal equation [23]. The matrix that contains the autocorrelations is called the autocorrelation matrix. This matrix is both symmetric and Toeplitz.

If m is the true order of the AR process (i.e., m=p) then $\phi_{m1}=\phi_1,\ldots,\phi_{mm}=\phi_m$. Therefore one technique for inverse filtering is to substitute estimated values for the autocorrelation function in equation (1-6) and to solve for $\phi_{m1},\ldots,\phi_{mm}$ and σ_e^2.

In chapter II, we explore several algorithms for inverse filtering. We will be investigating these algorithms looking for the commonality among them and choosing an algorithm that represents this class of algorithms.

CHAPTER II

BATCH INVERSE FILTERING ALGORITHMS

The previous chapter presented the problem of inverse filtering and the Yule-Walker or normal equation was discussed. This chapter presents some batch algorithms that solve the normal equation efficiently. By solving the normal equation, an estimate of the AR process parameters can be obtained thus achieving the goal of the inverse filtering problem.

The objective of this chapter is to study the batch algorithms, identify their similarities and differences, and finally, choose one algorithm that is representative of this class of algorithms. The selected algorithm will be the one to be implemented on advanced computer architectures. The results of implementing the selected algorithm on the different machines will be applicable to other batch algorithms.

The Levinson Algorithm

The Levinson algorithm is an efficient algorithm for solving the normal equation without inverting the autocorrelation matrix. Instead of solving the normal equation directly, the Levinson algorithm imbeds this problem into a whole class of similar problems; namely, of determining the best linear predictors of

ascending orders. The name linear prediction originate from the formulation of the model; the basic assumption is that the next sample in a sequence can be estimated from a linearly weighted sum of previous samples [26-28].

The solution is obtained in an iterative manner, by solving a family of matrix equations of lower dimensionality. Starting at the upper left corner of the autocorrelation matrix (the first element in the first row), i.e. first order equation, and successively increasing the order until the desired dimension is reached. The solution of each problem is obtained in terms of the solution of the previous one. In this manner, the final solution is gradually built up. In the process, one also finds all the lower order prediction error filters. Order determination is inherently performed by the Levinson algorithm.

The iteration is based on two key properties of the autocorrelation matrix: first, the autocorrelation matrix of a given size contains as subblocks all the lower order autocorrelation matrices; and second, the autocorrelation matrix is symmetric and Toeplitz, i.e., it is reflection invariant [29].

Equation (2-1) is the AR process of m^{th} order discussed in the previous chapter [30].

$$x_n + a_1^m x_{n-1} + \cdots + a_m^m x_{n-m} = e_n^m \qquad (2-1)$$

Using the approach followed in the previous chapter, the normal equation can be rewritten as shown in equation (2-2). This equation is equivalent to equation (1-6) but uses different variable names.

R(k) is the autocorrelation function, the a's are the AR parameters, and P_m is the variance of the white noise e_n^m .

$$\begin{bmatrix} R(0) & R(-1) & \cdots & R(-m) \\ R(1) & R(0) & \cdots & R(1-m) \\ \vdots & \vdots & \ddots & \vdots \\ R(m) & R(m-1) & \cdots & R(0) \end{bmatrix} \begin{bmatrix} 1 \\ a_1^m \\ \vdots \\ a_m^m \end{bmatrix} = \begin{bmatrix} P_m \\ 0 \\ \vdots \\ 0 \end{bmatrix} \qquad (2-2)$$

Consider the case of having solved equation (2-2) and wanting to increase the order of the model. Equation (2-3) shows the AR process of increased order , namely, m+1.

$$x_n + a_1^{m+1}x_{n-1} + \cdots + a_m^{m+1}x_{n-m} + a_{m+1}^{m+1}x_{n-m-1} = e_n^{m+1} \qquad (2-3)$$

Using equation (2-3) the new normal equation is shown by equation (2-4).

$$\begin{bmatrix} R(0) & R(-1) & \cdots & R(-m) & R(-m-1) \\ R(1) & R(0) & \cdots & R(1-m) & R(-m) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ R(m) & R(m-1) & \cdots & R(0) & R(-1) \\ R(m+1) & R(m) & \cdots & R(1) & R(0) \end{bmatrix} \begin{bmatrix} 1 \\ a_1^{m+1} \\ \vdots \\ a_m^{m+1} \\ a_{m+1}^{m+1} \end{bmatrix} = \begin{bmatrix} P_{m+1} \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \qquad (2-4)$$

Comparing equations (2-2) and (2-4) it is clear that the new autocorrelation matrix consists of the old matrix plus an extra row and column. Equation (2-4) can be rewritten as shown in equation (2-5). This expression is valid since the first and last rows of the autocorrelation matrix are reverses, the second and next to last rows are reverses, etc. (R(k) = R(-k)).

$$\begin{bmatrix} R(0) & R(-1) & \cdots & R(-m) & R(-m-1) \\ R(1) & R(0) & \cdots & R(1-m) & R(-m) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ R(m) & R(m-1) & \cdots & R(0) & R(-1) \\ R(m+1) & R(m) & \cdots & R(1) & R(0) \end{bmatrix} \left\{ \begin{bmatrix} 1 \\ a_1^m \\ \vdots \\ a_m^m \\ 0 \end{bmatrix} + c_{m+1} \begin{bmatrix} 0 \\ a_m^m \\ \vdots \\ a_1^m \\ 1 \end{bmatrix} \right\}$$

$$= \left\{ \begin{bmatrix} P_m \\ 0 \\ \vdots \\ 0 \\ Q_{m+1} \end{bmatrix} + c_{m+1} \begin{bmatrix} Q_{m+1} \\ 0 \\ \vdots \\ 0 \\ P_m \end{bmatrix} \right\} \qquad (2-5)$$

By inspection of equation (2-5) the following equation can be written:

$$Q_{m+1} = \sum_{n=0}^{m} R(m+1-n)\, a_n^m \qquad (2-6)$$

Now let

$$c_{m+1} = -\frac{Q_{m+1}}{P_m} \quad \text{or} \quad Q_{m+1} + c_{m+1} P_m = 0 \qquad (2-7)$$

Equation (2-5) will have the same form as equation (2-4), in which the first element of the left hand column vector is unity and the last m+1 elements of the right hand column vector are zero. We have thus found a solution to equation (2-4), and if we assume that the autocorrelation matrix is positive definite the solution must be unique [30] . The solution is given by equation (2-8):

$$\begin{bmatrix} 1 \\ a_1^{m+1} \\ \vdots \\ a_m^{m+1} \\ a_{m+1}^{m+1} \end{bmatrix} = \begin{bmatrix} 1 \\ a_1^m \\ \vdots \\ a_m^m \\ 0 \end{bmatrix} + c_{m+1} \begin{bmatrix} 0 \\ a_m^m \\ \vdots \\ a_1^m \\ 1 \end{bmatrix} \qquad (2-8)$$

where c_{m+1} is obtained from equation (2-7). By comparing equations (2-4) and (2-5) we can also see that:

$$P_{m+1} = P_m + c_{m+1} Q_{m+1} = P_m + c_{m+1} (-c_{m+1} P_m) = P_m \left(1 - c_{m+1}^2\right) \qquad (2-9)$$

The parameter c_{m+1} is called the reflection coefficient or the partial autocorrelation function. Equation (2-8) indicates the relationship of the reflection coefficients to the AR parameters. If the reflection coefficients are used instead of the AR parameters to realize an AR process, an interesting filter structure results as shown in Figure 2.1. Figure 2.1 is the lattice form realization of the AR process as opposed to Figure 1.1 that illustrates the direct form I realization. The lattice filter is an important structure in signal processing due to its modular structure and special features. The derivation of the Burg algorithm in the next section describes the equations that result in this interesting filter structure.

To summarize, the Levinson algorithm consists of equations (2-6), (2-7), (2-8), and (2-9). It is a recursive algorithm for estimating the coefficients of an AR process without a-priori knowledge of the process order. At the k-th iteration of the algorithm we obtain the AR coefficients of the k-th order model: $a_1^k, a_2^k, \cdots, a_k^k$.

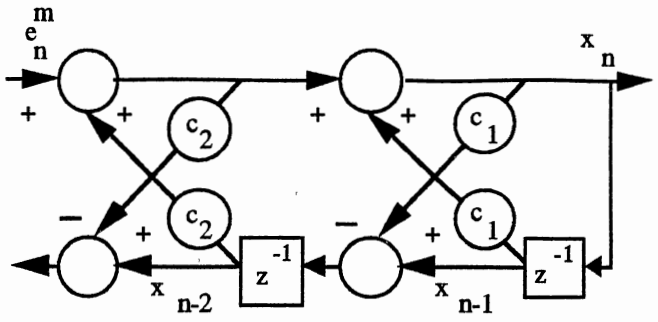


Figure 2.1 Lattice Filter Implementation of a 2nd Order AR process

The Burg Algorithm

The Burg algorithm is similar to the Levinson algorithm in that it estimates the coefficients of an AR process which are updated recursively using equation (2-8). The Burg differs from the Levinson algorithm in the way it calculates the reflection coefficients [30-31].

To derive the Burg algorithm, first consider the forward prediction error of the $m+1^{th}$ order AR model: (the prediction is forward in the sense that the prediction for the current data sample is a weighted sum of previous samples)

$$e_n^{m+1} = x_n + a_1^{m+1}x_{n-1} + \cdots + a_m^{m+1}x_{n-m} + a_{m+1}^{m+1}x_{n-m-1} \qquad (2-10)$$

If equation (2-8) is used to obtain the coefficients, equation (2-10) can be rewritten as:

$$e_n^{m+1} = x_n + \left(a_1^m + c_{m+1}\, a_m^m\right) x_{n-1} + \cdots$$

$$+\left(a_m^m + c_{m+1}\, a_1^m\right) x_{n-m} + c_{m+1}\, x_{n-m-1} \qquad (2-11)$$

Now consider the forward prediction error of the m^{th} order model:

$$e_n^m = x_n + a_1^m x_{n-1} + \cdots + a_m^m x_{n-m} \qquad (2-12)$$

There is an equivalent backward prediction model: (the prediction is backward in the sense that the prediction for the current data sample is a weighted sum of future samples)

$$b_{n-m-1}^m = x_{n-m-1} + a_1^m x_{n-m} + \cdots + a_m^m x_{n-1} \qquad (2-13)$$

It can be shown that the statistics of this model are equivalent to those of the forward prediction model.

By comparing equation (2-11) with equations (2-12) and (2-13), we can see that:

$$e_n^{m+1} = c_{m+1}\, b_{n-m-1}^m + e_n^m \qquad (2-14)$$

Likewise, we could show:

$$b_{n-m-1}^{m+1} = b_{n-m-1}^m + c_{m+1}\, e_n^m \qquad (2-15)$$

The Burg algorithm chooses c_{m+1} so as to minimize the sum of squares of the forward and backward prediction errors:

$$J = \sum_{n=m+2}^{M} \left(e_n^{m+1}\right)^2 + \left(b_{n-m-1}^{m+1}\right)^2 \qquad (2-16)$$

If equations (2-14) and (2-15) are used in equation (2-16) we can rewrite the latter as:

$$J = \sum_{n=m+2}^{M} \left(e_n^m + c_{m+1}\, b_{n-m-1}^m\right)^2 + \left(b_{n-m-1}^m + c_{m+1}\, e_n^m\right)^2 \qquad (2-17)$$

To minimize equation (2-17) we take its derivative with respect to c_{m+1} , set it equal to zero, and solve for c_{m+1} . The derivative is given by:

$$2\sum\left(e_n^m + c_{m+1}\, b_{n-m-1}^m\right) b_{n-m-1}^m + 2\sum\left(b_{n-m-1}^m + c_{m+1}\, e_n^m\right) e_n^m \qquad (2-18)$$

Rearranging equation (2-18) and setting it equal to zero result in equation (2-19).

$$\sum\left(e_n^m b_{n-m-1}^m + b_{n-m-1}^m e_n^m\right) + c_{m+1}\sum\left(\left(b_{n-m-1}^m\right)^2 + \left(e_n^m\right)^2\right) = 0 \qquad (2-19)$$

Solving for c_{m+1} result in the reflection coefficients that minimize the sum of squares of the forward and backward errors:

$$c_{m+1} = \frac{-2\sum_{n=m+2}^{M} e_n^m\, b_{n-m-1}^m}{\sum_{n=m+2}^{M}\left(e_n^m\right)^2 + \left(b_{n-m-1}^m\right)^2} \qquad (2-20)$$

To summarize, the Burg algorithm consists of two steps: 1) update the forward and backward prediction errors using equations (2-14) and (2-15); 2) calculate the reflection coefficient using equation (2-20) and then repeat step 1) . If the AR parameters are desired they are calculated using equation (2-8), as in the Levinson algorithm. This implies that the lattice filter implementation of the AR process is valid when the Burg algorithm is used. In fact, the forward and backward prediction errors given by equations (2-14) and (2-15) constitute the lattice structure shown in Figure 2.1.

The Burg algorithm uses equation (2-8) from the Levinson algorithm to update the coefficients of the AR model, but it differs from the Levinson algorithm in that it chooses the reflection coefficient, c_{m+1} , so as to minimize the sum of squares of the forward and backward prediction errors.

Notice that equations (2-14) and (2-15) involve a time shift of the b sequence relative to the e sequence, and that equation (2-20) involves three inner product operations. This combination of operations is found in all algorithms which use convolution or correlation.

The Method of Least Squares

The least squares method is one of the most popular and useful techniques for obtaining parameter estimates of an unknown system or signal model. The convergence properties of least squares estimates have been well established [23].

We begin by discussing the general problem and the proposed solution. Consider the problem of finding a vector $x \in \Re^n$ such that $Ax=b$ where $A \in \Re^{m \times n}$ and $b \in \Re^m$ are given and $m>n$. When there are more equations than unknowns, we say that the system $Ax=b$ is overdetermined. Usually an overdetermined system has no exact solution.

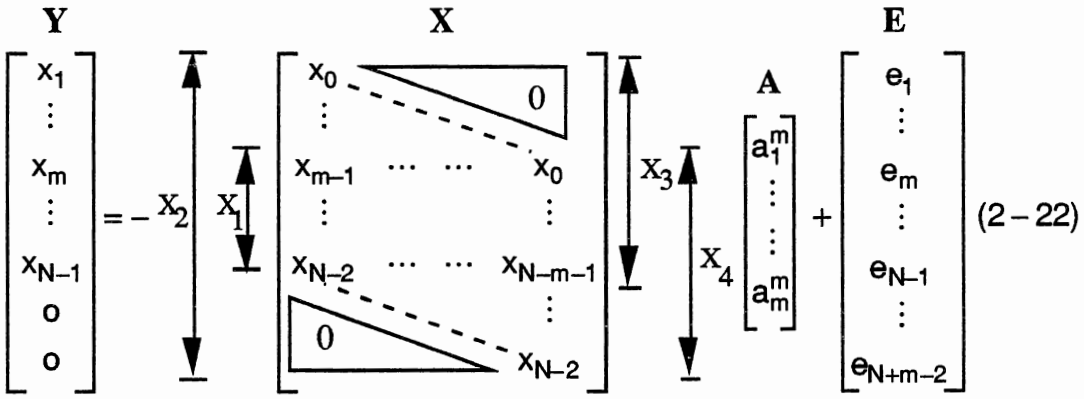
This suggests that we strive to minimize $|Ax-b|_p$ for some suitable choice of p. Different norms render different optimum solutions. Minimization in the 1-norm and ∞-norm is complicated by the fact that the function $|Ax-b|_p$ is not differentiable for those values of p. However, the next section discuss the case where $1 \le p < 2$ and present efficient techniques to solve the problem. On the other hand, $|Ax-b|_2$ is a continuously differentiable function of x [32].

The least squares formulation can be applied to the problem of estimating the parameters of an autoregressive process. Assume the data sequence $x_0, \ldots, x_{N-1}$ is used to find the m^{th} order AR parameter estimates. Recall equation (2-1) that describes an m^{th} order AR process. Equation (2-21) is equivalent to equation (2-1) rewritten to express the output in terms of the weighted sum of the previous output values and the white noise process.

$$x_n = -a_1^m x_{n-1} - \cdots - a_m^m x_{n-m} + e_n^m \qquad (2-21)$$

We can evaluate e_n^m in equation (2-21) for n=1 to n=N+m-2 if one assumes the terms outside the measurements are zero, i.e., $x_n=0$ for $n<0$ and $n>N-1$. Notice the existence of an implied

windowing of the data sequence in order to extend the index range from 1 to N+m-2. Using a matrix formulation we can rewrite equation (2-21) for the specified range as shown in equation (2-22).



Recall that the forward linear predictor will have the usual form:

$$\hat{x}_n = -(a_1^m x_{n-1} + \cdots + a_m^m x_{n-m}) = -\sum_{i=1}^{m} a_i^m \, x_{n-i} \qquad (2-23)$$

and the forward linear predictor error or residual is given by:

$$e_n^m = x_n - \hat{x}_n = -\sum_{i=0}^{m} a_i^m \, x_{n-i} \qquad ; \text{where } a_0^m = 1 \qquad (2-24)$$

Notice that the forward linear predictor error is equivalent to the white process, i.e. the E vector in equation (2-22), given that the process is autoregressive.

Using equation (2-22), we can solve for the residual vector:

$$E=Y-XA \qquad (2-25)$$

Notice that E^TE is simply the sum of squares of the residuals or errors. In the least squares method the objective is to minimize the sum of squares of the errors, i.e. E^TE. Thus the cost function to be minimized is given by:

$$J = E^TE = (Y-XA)^T(Y-XA) = Y^TY-Y^TXA-A^TX^TY+A^TX^TXA \qquad (2\text{-}26)$$

Differentiating J with respect to the vector A, setting it to zero, and rearranging will give the following system of normal equations:

$$(X_k^TX_k)\,A = X_k^TY \qquad (2\text{-}27)$$

The subscript k indicates which data matrix is used. There are four different possibilities, illustrated in equation (2-22), for the selection of the data matrix [28]. Hence, k takes on the values 1, 2, 3, or 4 depending on our windowing choice to indicate the selection of the covariance, the autocorrelation, the prewindowed, or the postwindowed formulation, respectively. Solving for A in equation (2-27) will result in the least squares solution given by:

$$A = (X^TX)^{-1}\,X^T\,Y \qquad (2\text{-}28)$$

In practice, the vector A is not computed using equation (2-28) since the computation of the inverse is fraught with numerical difficulties. Instead, the normal equation (2-27) is solved using numerically stable algorithms that involve orthogonal transformations. Hence,

equation (2-28) is a useful "theoretical" formula but is not a useful computational formula [23].

Equation (2-27) has the same structure as the Yule-Walker equations; however, the data matrix product ($X_k^T X_k$) is not necessarily Toeplitz as are the Yule-Walker equations [32]. Notice that the subscript k is used to indicate the data matrix selected as mentioned earlier.

If the data matrix X_1 is selected, the normal equations are termed the covariance equations or formulation, often encountered in linear predictive coding (LPC) of speech [28]. The ($X_1^T X_1$) matrix is symmetric but not Toeplitz. The square root method or Cholesky decomposition is used in this case for computing the A vector [33]. The Cholesky decomposition factors the data matrix product, which has the properties of a covariance matrix, to solve the system given by equation (2-22). Cholesky decomposition states that: if a matrix A is symmetric positive definite, then there exists a lower triangular matrix G with positive diagonal entries such that [32] :

$$A = L D L^T = (L D^{1/2})(D^{1/2} L^T) = G G^T \qquad (2\text{-}29)$$

To avoid square root computations, the factors L and D are computed rather than the factor G. L is a unit lower triangular and D is a diagonal matrix with positive elements. The elements of L and D can be determined by equating the elements of both sides in equation (2-29). If the first i-1 columns of L and D have been determined then the i-th column can be determined as:

$$d_i = a_{ii} - \sum_{k=1}^{i-1} d_k \, l_{ik}^2 \qquad (2-30)$$

$$l_{ji} = (a_{ji} - \sum_{k=1}^{i-1} d_k \, l_{jk} \, l_{ik}) / d_i \qquad i < j \leq n \qquad (2-31)$$

The Cholesky decomposition requires $n^3/6$ operations which is more than required by the Levinson algorithm, namely, $2n^2$ operations [32].

If the data matrix X_2 is selected, the normal equations are called the autocorrelation equation or formulation since the product matrix $(X_2{}^T X_2)/N$ reduces exactly to the Yule-Walker equations, for which the biased autocorrelation estimator has been used instead of the known autocorrelation function [28]. Notice that a data window has been assumed for this case. In this case, the $(X_2{}^T X_2)$ matrix is Toeplitz and the A vector can be solved for using the Levinson algorithm discussed in section 2.1 .

If the data matrix X_3 is selected, the normal equations are termed the prewindowed normal equations due to the zero value assumptions made for the missing data prior to x_o .

If the data matrix X_4 is selected, the normal equations are termed the postwindowed normal equations since a zero data assumption is made for the data beyond x_{N-1}.

It would appear that only the data matrix X_2 will yield normal equations with Toeplitz structure to permit an efficient recursive solution (namely, the Levinson algorithm). However, even though

the product matrix ($X_k^T X_k$) may not be Toeplitz, each of the four matrices X_k have Toeplitz structure.

The Lp Techniques

In the previous section the least squares method was investigated to explore the possibilities of using such method in inverse filtering. The main idea in setting up the least squares formulation is to minimize the sum of squared residuals or errors. The residuals are the difference between the actual data and the model. The solution obtained is the least squares solution which can be termed the L_2 solution. The number 2 indicates the residual terms are raised to the second power before summing.

In the previous section it was indicated that in general, one could raise the residual terms to some arbitrary p^{th} power and perform the minimization to get the L_p solution. In L_p techniques, the values of p other than two may offer some advantages in a number of ways. For example, the L_1 (absolute value) solution tends to ignore outliers while the L_2 solution tries to satisfy all points as best it can. In general, values of p between one and two blend these characteristics somewhat [34-39].

Other values for p, such as $p<1$ and even negative p, can be considered but unfortunately the results obtained for this range do not have a mathematical basis, as L_p is not a normed linear space. The solutions for $p>2$ are more sensitive to aberrant noise. The parameter p controls the trade-off between emphasizing and

deemphasizing aberrant noise. The L_1 solution is considered to be robust for its low sensitivity to aberrant noise [38].

The previous section described some of the algorithms to get the least squares solution efficiently. Unfortunately there is no simple solution for the L_p case but special iterative algorithms were developed to efficiently obtain the solution. Linear programming was used to get the L_1 solution but could not be used to obtain the general L_p solution. The iterative reweighted least squares (IRLS) algorithm can be used to get the L_p solution, but in general, the p-normed solution can be efficiently solved by using the residual steepest descent (RSD) algorithm which is a steepest descent method with an adaptive stepsize [34,37,39].

Linear programming formulations have two drawbacks: for a large data set linear programming requires an excessive amount of memory, in addition, it does not guarantee selection of a reasonable prediction error filter from the several possible solutions. By contrast, the IRLS algorithm starts from the least squares solution and iterate toward a solution from there. Each iteration solves a new L_2 problem by employing the weighted residuals of the previous iteration in the current one [34, 37, 39]. The rest of this section will describe the IRLS and the RSD algorithms.

The IRLS algorithm is based on the least squares solution. It is an iterative algorithm that uses a weighted least squares to solve the L_p formulation. The equation (2-22) , used in the previous section to set up the least squares problem, is also used here as the basis to set up the formulation. Equation (2-32) is equivalent to equation (2-22) repeated here in matrix notation.

$$Y = X A + E \tag{2-32}$$

The problem again is to estimate the A vector. The IRLS algorithm estimates the A vector for a selected p value iteratively. The first step in the IRLS is to compute [34,37,39]:

$$A(k+1) = (X^T W(k) X^T)^{-1} X^T W(k) Y \tag{2-33}$$

where W(k) is a diagonal matrix with its diagonal entries, $W_{ii}(k)$, given by:

$$W_{ii}(k) = \begin{cases} | r_i(k) |^{p-2}, & | r_i(k) | > \varepsilon \\ \varepsilon^{p-2}, & | r_i(k) | \le \varepsilon \end{cases} \tag{2-34}$$

where $r_i(k)$, the residual, is given by:

$$r_i(k) = (Y - X A(k))_i \tag{2-35}$$

and ε is some small positive number. Notice that if p=2, the W matrix will be equivalent to the identity matrix and equation (2-33) will be equivalent to the least squares solution given by equation (2-28). In fact the least squares solution can be used as an initial vector to solve for an arbitrary L_p solution.

Although the IRLS is a fast convergent algorithm, it still requires the computation of an inverse for a matrix at each stage k. Fast IRLS algorithms, based on fast Fourier transforms, were

developed to reduce the number of computations where the matrix X takes a special form [34].

The RSD algorithm uses fewer number of operations per iteration than the IRLS algorithm. The RSD solves the problem iteratively by the recursion [34,37,39]:

$$A(k+1) = A(k) - \Delta_k (X^T X)^{-1} X^T v(k) \tag{2-36}$$

where

$$v(k) = \text{col}\,[\, v_1(k) \;\; v_2(k) \;\; \ldots \;\; v_{N+m-2}(k) \,] \tag{2-37}$$

with

$$v_i(k) = \left| (X A(k) - Y)_i \right|^{p-1} \text{sgn}(X A(k) - Y)_i \tag{2-38}$$

where sgn(t)=+1 (-1) if t>0 (t<0). When t=0, one can arbitrarily choose sgn(t) to be either +1 or -1. The step size or the scale factor Δ_k is determined by minimizing:

$$\| -Y + X A(k) - \Delta_k X (X^T X)^{-1} X^T v(k) \|_p \tag{2-39}$$

with respect to Δ_k in the L_p sense. In equation (2-39), since the only unknown is Δ_k and it is a scalar, we can use the IRLS algorithm to solve for Δ_k . Notice that in the RSD algorithm we need to compute the matrix inverse only once, thus, reducing dramatically the number of computations required when compared to the IRLS algorithm.

Selection of a Representative Algorithm

In this chapter a description of the various batch inverse filtering algorithms was presented. The algorithms were derived to study their computational structure to select the algorithm that best represents this class of algorithms.

The algorithms seem to have a common computational structure, namely, a time shift/inner product operation. In fact, this operation is a key step in performing all signal processing algorithms which involve convolution or correlation.

The Burg algorithm is selected to represent this class of algorithms for several reasons. The time shift/inner product operation constitutes a large portion of the algorithm. The Burg algorithm generates models that are always stable and yield a solution in terms of reflection coefficients. The lattice structure embedded in the Burg algorithm makes it modular and stable.

The Levinson recursion was shown to be embedded in the Burg algorithm. The least squares solution using the autocorrelation formulation can be solved efficiently using the Levinson algorithm. Notice that the least squares autocorrelation formulation is always guaranteed to yield a stable filter, by contrast, the covariance formulation does not.

In the general L_p problem, no formulation mentioned so far can assure stability except for the autocorrelation form with p=2; other values of p may yield unstable models, no matter which method is used. In particular, the autocorrelation model is always stable for p less than three and greater or equal to two, but there may exist

some p_o in the interval $1<p_o<2$ for which the prediction filter may not be stable. In that case, stability is not assured for any model generated in the range $1<p<p_o$ [37,38]. Filter stability can be assured by using a different formulation of the linear prediction problem, namely, the lattice or Burg algorithm. In fact, generalized Burg algorithms which ensure filter stability for the L_1 solution were investigated in the literature [37,40].

CHAPTER III

PARALLEL PROCESSING COMPUTER ARCHITECTURES

The basic definition of parallelism is the ability to do more than one activity at once. Doing *n* different activities at once; doing one activity in *n* simultaneous parts; doing *n* activities staggered in time; using *k* resources for *n* jobs; and *k* resources for one job - all of the above represent instances of parallelism. The common thread that runs through these examples is the utilization of multiple resources in an instance of time to increase the amount of work performed per unit of time.

Despite early intellectual flirtations with parallelism, until recently it has remained largely a concept. During the past two decades, several parallel processor prototypes have been built. One of particular note was the ILLIAC IV, conceived at the University of Illinois by Daniel Slotnick in 1966 as quadrants of 64 processing elements, but reduced down to one by 1972 because of technical difficulties. Recently, we are starting to see more and more parallel designs successfully executed and commercially available.

This chapter describes a classification scheme for computers and selectively describes, in detail, seven advanced computer systems. The seven computer systems described represent different architectures: the Denelcor HEP, a shared memory (tightly coupled)

multiple-instruction stream multiple-data stream (MIMD) machine with switch network interconnect architecture; the Cray X-MP/48, a shared memory (tightly coupled) MIMD supercomputer with direct connect interconnect architecture; the Intel iPSC/2 hypercube computer, a distributed memory (loosely coupled) MIMD machine; the Alliant FX/8, a shared memory (tightly coupled) MIMD machine with a bus interconnect architecture; the NASA/Goodyear MPP, a massively parallel SIMD machine with mesh interconnect architecture; the Connection Machine model CM-2, a massively parallel SIMD machine with hypercube interconnect architecture; and the Cray-2 supercomputer, a tightly coupled MIMD machine with direct connect interconnect architecture and is the latest Cray to be produced. These architectures are considered to be representative of the commercially available parallel computer architectures. The selected algorithm was implemented on these machines and the results are reported in chapter IV.

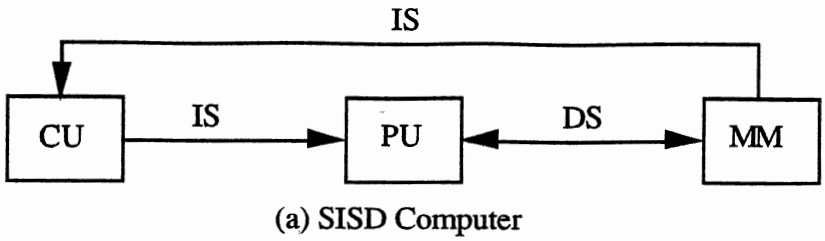
All of the architectures described in this chapter have one goal, to increase computational power by using replicated processing elements that are connected to and can communicate over some type of network. This goal results from the bounds on performance in traditional von Neumann architectures.

Computer Architecture Classification Scheme

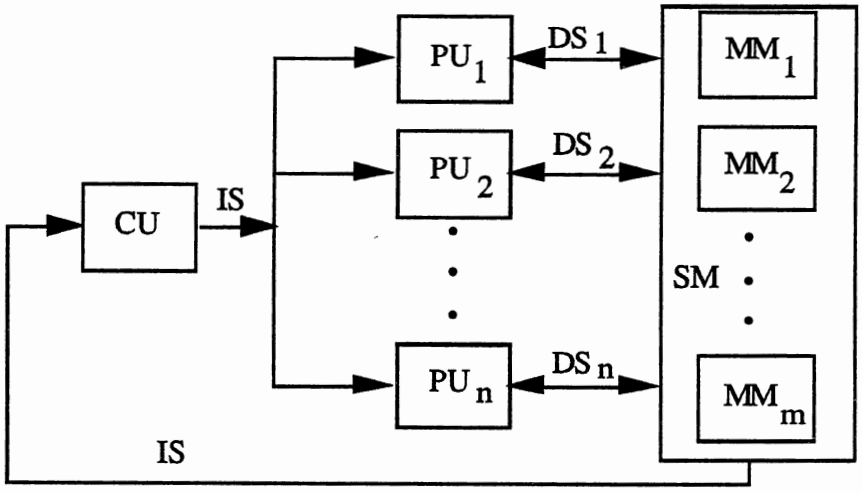
One of the oldest and still most widely used methods of classifying computer systems was developed by Flynn in 1966[19].

Figure 3.1 illustrates Flynn's classification which is based on program and data parallelism, i.e. the multiplicity of instruction streams and data streams in a computer system. In a conventional sequential computer, at any instant of time, there can be but a single command in the command register, and this command can effect an arithmetic or logical operation upon a single datum stored in the accumulator. Such a machine organization is termed single-instruction stream, single-data stream, or SISD [19]. Most SISD machines are pipelined and can have more than one functional unit under the supervision of one control unit.

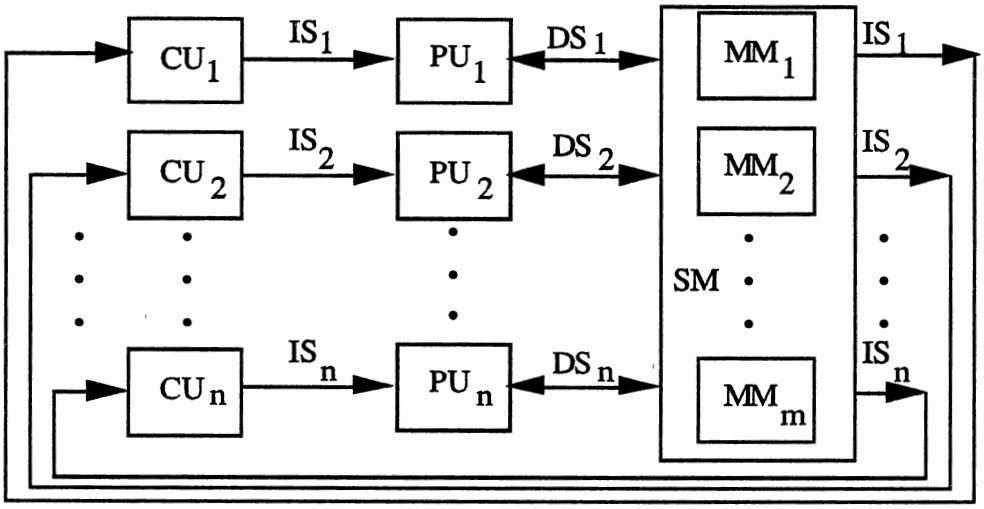
In one widely used approach to parallelism, a multiplicity of concurrently operating processing elements is provided, where each processing element consists of an ALU and a memory unit. The arithmetic and memory units are interconnected to form a network or an array. The system contains only one program control unit which can activate any or all of the arithmetic units. Each active element of the array performs the same arithmetic or logic operation under command of the control unit. Each arithmetic element may be operating on different data in executing the instruction resident in the control unit. For this reason, this type of structure is termed single-instruction stream, multiple-data stream, or SIMD. SIMD machines are also called array processors [19].



(a) SISD Computer



(b) SIMD Computer



(c) MIMD Computer

Figure 3.1 Flynn's Classification of Computer Architectures

In the third approach, each processing element contains a control unit as well as an ALU and memory unit. The elements of the network can therefore function as full-fledged independent digital computers, and during any instruction cycle each processing element can carry out a different arithmetic or logic operation. For this reason systems of this type are termed multiple-instruction stream, multiple-data stream, or MIMD. Most multiprocessor systems and multiple computer systems can be classified in this category. MIMD machines are considered tightly coupled if there is a shared memory and the degree of interactions among the processors is high, otherwise, they are considered loosely coupled. Loosely coupled systems employ distributed memory with a low degree of interactions among the processors.

Since Flynn published his classification scheme, new parallel computer architectures have emerged which incorporated a variety of new architectural concepts. Currently, Flynn's classification scheme is still used but other classification schemes were developed mainly to augment Flynn's classification scheme making it more complete and accurate when used to classify new architectures. Some of the classification schemes are based on data sharing mechanism, synchronicity of operation, or granularity of computations [41]. Other schemes have emphasized a particular class of machines, like MIMD machines categorizing them as either switched systems or networks [42-43].

The aforementioned classification schemes of advanced computer architectures are only a few of the currently existing schemes. A variety of other classification schemes exist that adds to

the complexity of placing a given computer system within a definite class. The science of computer classification schemes is by no means complete and the necessity for a more clear and accurate scheme still exists.

In this research, Flynn's classification is used along with a new classification scheme [21]. The classification scheme is based on three essential issues that must be considered for a parallel architecture: the granularity of the processing elements; the topology of the interconnections between processing elements; and the distribution of control across the processing elements. Granularity refers to the power of each processing element in the architecture ranging from many single-bit processors to a few powerful general purpose ones. Topology refers to the pattern and density of the connections that exist between the processing elements. Control distribution is concerned with allocating tasks to the processing elements and synchronizing their interactions. Figure 3.2 illustrates the so called organizational space of parallel computer systems with these three variables as the axes.

In describing each computer architecture in this chapter, an attempt is made to place each system in relative perspective by illustrating their approximate position within the space. The criteria used in placing these systems are somewhat subjective and qualitative. The architectures described are so different in their structure and operations that it is virtually impossible to establish a one-to-one comparison of their features. It should be emphasized that the placing criteria largely depend on the way each machine is used in this research.

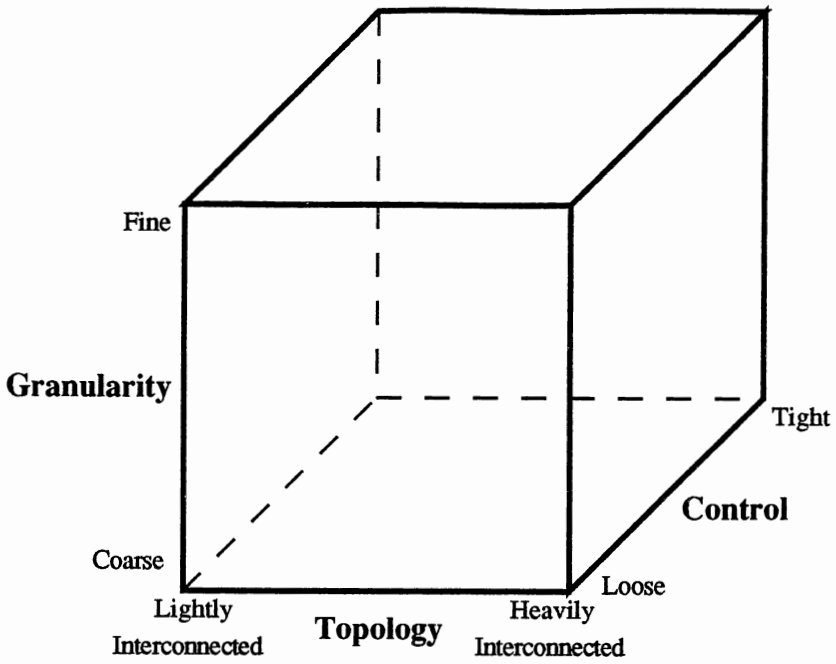


Figure 3.2 Organizational Space of Parallel Computer Systems

Description of Selected Advanced Computer Systems

In this section a selected group of advanced computer architectures are described in detail. The hardware, software, and classification of each system are discussed in detail. The computer systems described here represent a variety of interesting architectures.

The Heterogeneous Element Processor (HEP)

The Heterogeneous Element Processor (HEP) was first developed for the Army Ballistic Research Laboratories at Aberdeen by

Denelcor, Inc. The HEP is a large scale scientific parallel computer employing shared resource (tightly coupled) MIMD architecture. The processors used in the HEP are pipelined to support many concurrent processes, with each pipeline segment responsible for a different phase of instruction interpretation. Each processor has its own program memory, general purpose registers, and functional units; a number of these processors are connected to shared data memory modules by means of a very high speed pipelined packet switching network [19,44].

The extensive use of pipelining in conjunction with the shared resource idea result in a flexible and effective architecture. For example, the switch used to interconnect processors and memories is modular, and is designed to allow a given system to be expanded as needed. The increased memory access times that result from greater physical distances can be compensated for by using more processes in each processor because the switch is pipelined.

An overall block diagram of a typical HEP configuration is shown in Figure 3.3. The switch network shown has 28 nodes; it interconnects four processors, four data memory modules, and I/O processor and devices. Systems of this kind can be built to include as many as 16 processors and 128 data memory modules. Each processor performs 10 million instructions per second (MIPS), and the switch bandwidth is 10 million 64 bit words per second per network link. All instructions and data words in the HEP are 64 bits wide, although data references within each processor can access halfword, quarterword, and bytes [19,44].

Parallel processing and MIMD architecture, as implemented in HEP, allow up to 100 independent (or cooperating) instruction streams executing in parallel at any given time. In the HEP, these instructions are called processes, and 50 processes can be active at one time in a single processor. Each process can have its own unique data stream. With a number of processes executing concurrently, it is practical to separate an application problem into its component parts and execute the parts in parallel, with intermediate results passed between the cooperating processes as necessary [19,44,45].

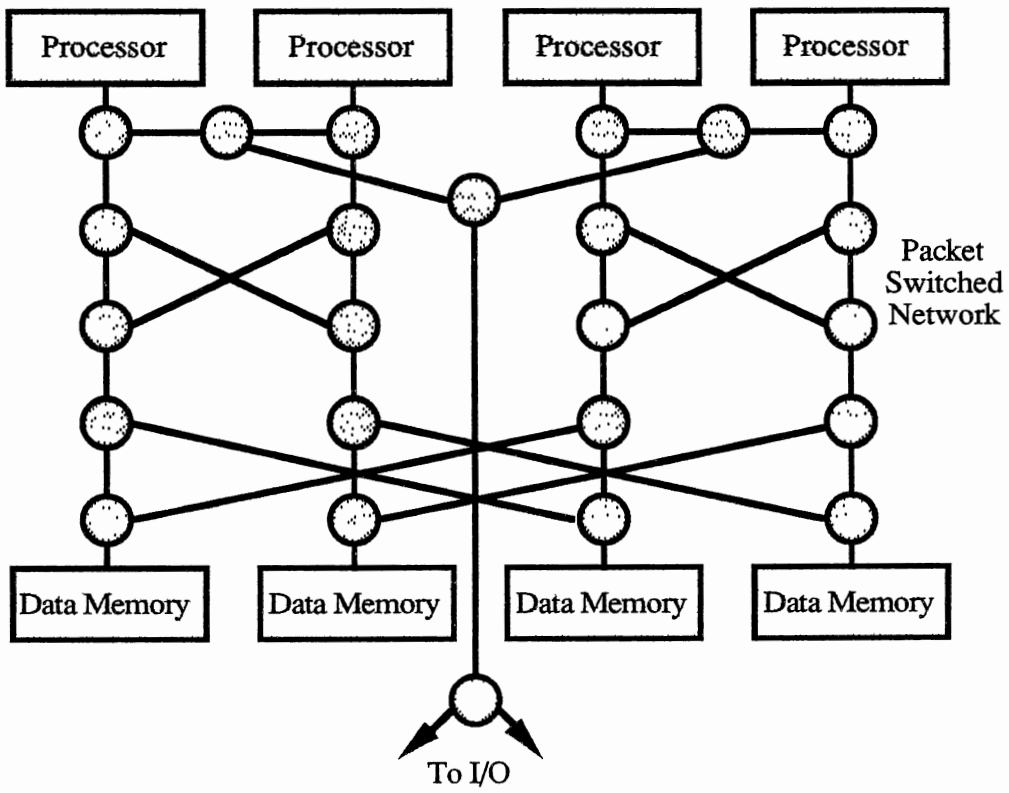


Figure 3.3 Four-Processor HEP System

Figure 3.4 illustrates the routing control in the bidirectional 3-ported switch node. The switch is synchronous and modular employing packet switching. Each node is connected by three full duplex ports. Each node receives three message every 100ns and route them for optimal destination, i.e. with minimal delay. Each node has three routing tables, one per port; tables are indexed by destination address and contain the identification of the preferred port out of which the packet should be sent [19,44].

A unique feature of each switching node is it does not enqueue messages in case a conflict for a port occurs; instead, it routes all messages immediately to output ports. It is the responsibility of the neighbors of the node to make sure that incorrectly routed messages eventually reach their correct destinations.

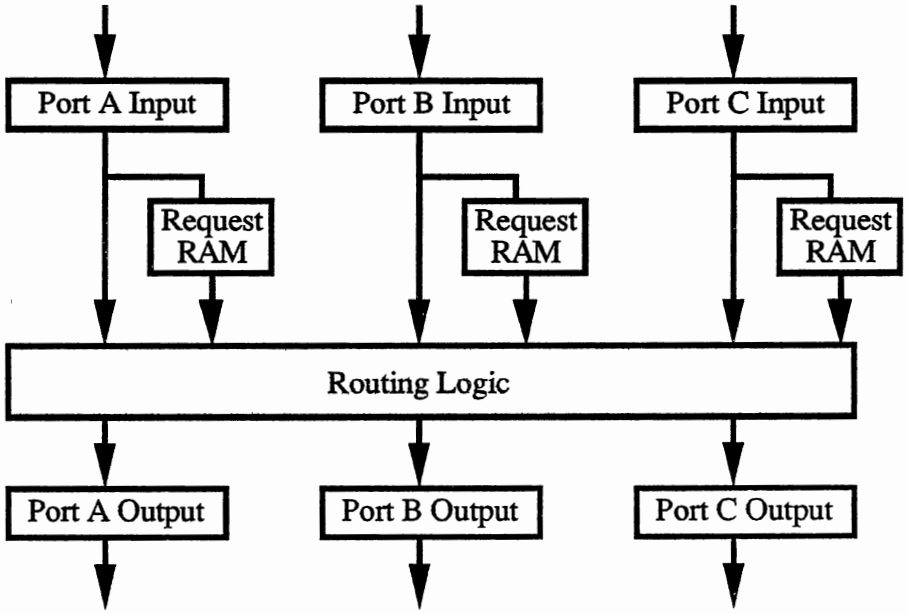


Figure 3.4 Routing Control in the 3-ported Switch Node

The HEP main programming language is HEP/UPX FORTRAN 77 which incorporates two kinds of extensions to FORTRAN 77: CREATE and RESUME statements which are syntactically equivalent to CALL and RETURN statements in FORTRAN 77 but used here for the creation and termination of processes, and access states of the asynchronous variables for synchronization between processes. Synchronization is required for handling data dependencies among user-created instruction streams. The user is responsible for establishing proper synchronization within his program using asynchronous variables that can be set to an access state, namely full or empty. PURGE statement is used to unconditionally set the access state of a synchronous variable to empty. Reading and writing to a synchronous variable will set it empty and full respectively. The HEP read and write instructions are controlled by these access states. By manipulating the access states, multiple instruction streams can be synchronized to access common memory locations [45].

Figure 3.5 illustrates the position of the HEP system in the organizational space. The packet switched network connecting the processors and the resources is considered to be lightly interconnected while the granularity of the HEP is relatively coarse since each 64 bit processor is general purpose. The resources within the system are shared making the control of communications among resources relatively tight.

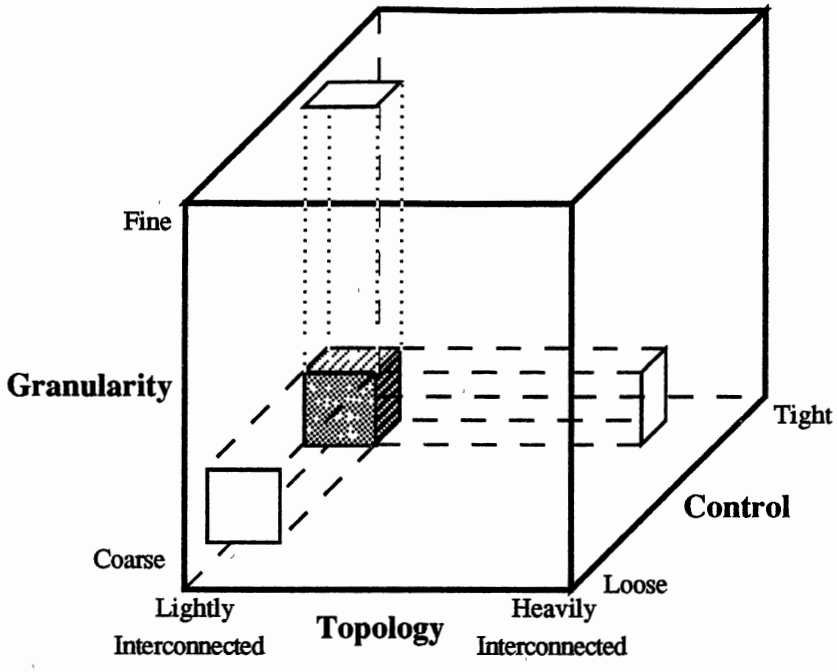


Figure 3.5 HEP Position in the Organizational Space

The Cray X-MP/48

The Cray-1 computer was first delivered in 1976 to Los Alamos National Laboratory and since then it has been the industry standard in very high-speed computing. The success of the Cray-1 can be attributed to its innovative vector architecture, dense packaging, and advanced cooling technology [19-22,46].

The Cray-1 design employs many state-of-the-art architectural features such as: pipelining in memory access and function units, utilization of vector registers and operations chaining, concurrent execution of multiple functional units, interleaved memory, instruction cache and lookahead, and massive use of parallel logic to shorten the execution time of functional units.

The Cray X-MP/48 or Experimental Multi-Processor is a multiprocessor extension of the Cray-1 that was completed in 1983. The Cray X-MP/48 is a tightly coupled MIMD supercomputer. It contains four Cray-1 like processors that share memory and I/O subsystems and has a clock cycle of about 9.5 nanoseconds (vs. 12.5 nanoseconds of Cray-1). Most often the four CPU's function independently, but their instruction streams can be synchronized.

Figure 3.6 illustrates the overall system of the X-MP/48. Although built upon the basic architecture of the Cray-1, the X-MP/48 processor is totally redesigned. All processors share a central memory of 8 million (64-bit) words, organized in interleaved memory banks. All banks can be accessed independently and in parallel during each machine clock period. Each processor has four parallel memory ports (four times that of Cray-1) connected to the central memory: two for memory loads, one for memory stores, and one for independent I/O operations.

The multiport memory has built-in conflict resolution hardware to minimize access delay and to maintain the integrity of all memory references from different ports to the same bank at the same time. The multiport memory design, coupled with a shorter memory cycle time, provides a high performance memory organization with up to 16 times the memory bandwidth of a Cray-1. The improved memory bandwidth balances the multiple-pipelined computing power of the CPU and the data streaming ability of the memory. For each processor, this capability, coupled with reduced clock period, gives a performance speedup over the Cray-1 of up to 4 [19-22,46].

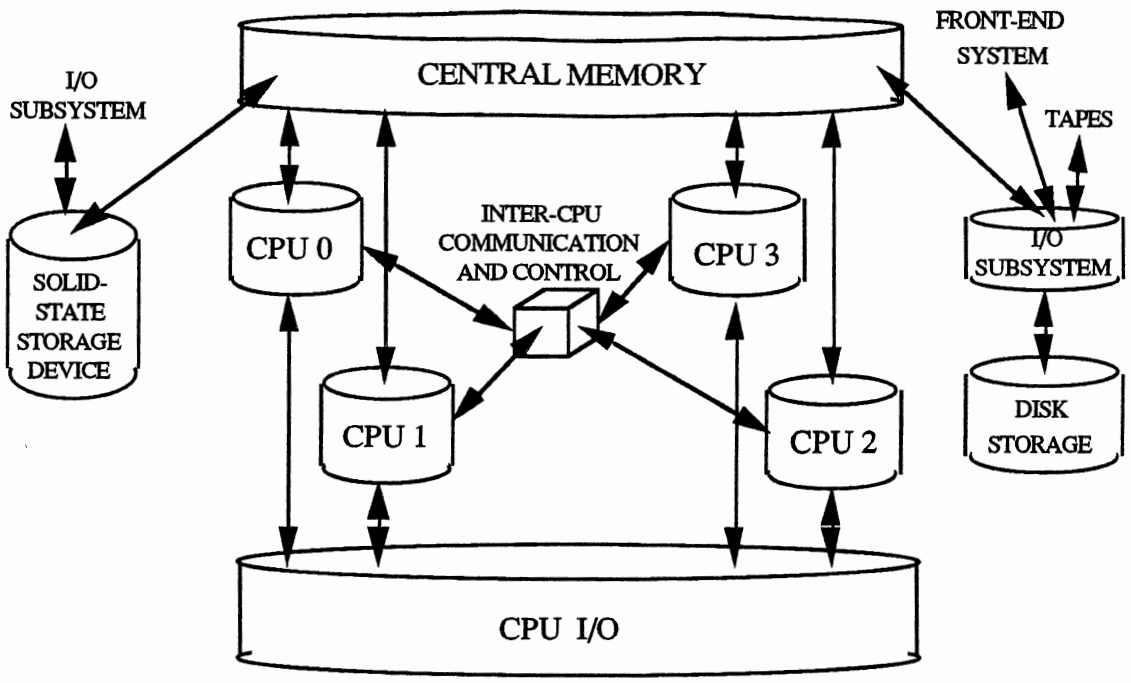


Figure 3.6 The Cray X-MP/48 Overall System Organization

The processors, which share the I/O ports, are controlled synchronously by a central clock. The scalar performance of each processor is improved through faster machine clock, shorter memory access, larger instruction buffers (twice that of the Cray-1), multiple data paths, and multiple processors. The vector performance of each processor is improved through faster machine clock, parallel memory ports, and a hardware automatic flexible chaining feature. The machine allows simultaneous memory fetches, a sequence of computations, and memory store in a series of related vector operations.

The Cray X-MP/48, like the Cray-1, achieves low-level parallelism through vectorization. The Cray FORTRAN compiler (CFT)

analyzes innermost DO loops to detect vectorizable sequences and then generates code to take advantage of the processor organization. The vectorization performed by the compiler is automatic, providing increased performance without restructuring or handcoding. In addition, the Cray X-MP/48 can achieve high-level parallelism via multitasking. All of the processors can cooperate to solve a problem by running separate tasks in parallel.

Any required synchronization must be specified by the programmer via calls to the multitasking library. Multitasking requires careful consideration of the algorithm at hand and data dependencies that may exist. A variety of facilities are provided to support multitasking: compiler linkage protocols, utilities, memory management facilities, and multitasking synchronization routines.

A task is defined as a program unit capable of being independently assigned a processor. All tasks of a program share the same FORTRAN common memory area, but each task is allocated a private environment for its local variables. All programs consist initially of one task. Any task can create a number of other tasks. All tasks created as descendants of the initial task run logically in parallel, but actual parallel processing across the two processors depends on instantaneous machine loading and available resources. Hence, it is not possible to easily determine for a particular run whether separate tasks actually ran in parallel.

Finally, Figure 3.7 illustrates the position of the Cray X-MP/48 in the organizational space. Low-level parallelism is used in determining the approximate placement location. The processors are of fine granularity because they operate on multibit elements. The

topology is fairly heavily interconnected, because communication is performed at a low level and without the need for contention to access a communication path. The operation of the X-MP/48 is tightly coupled.

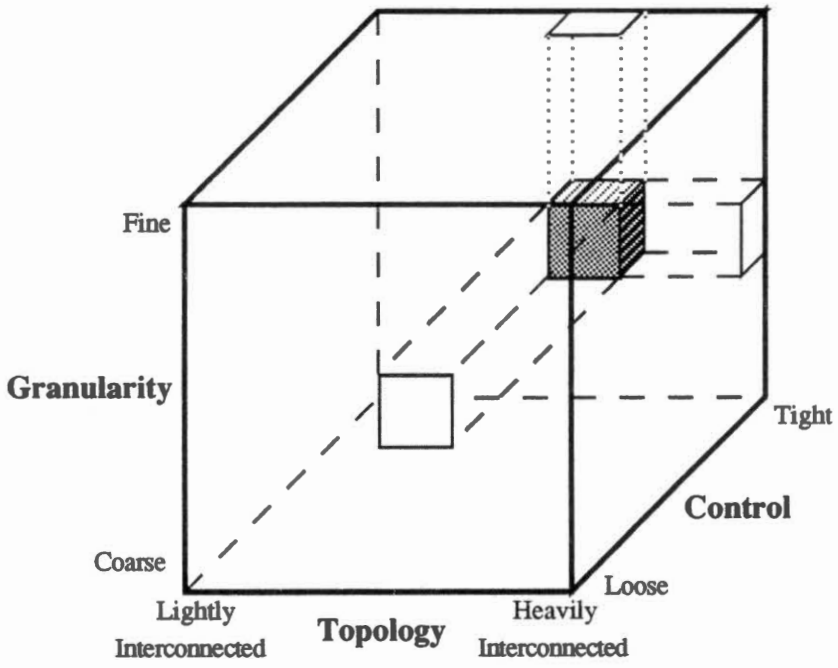


Figure 3.7 Cray Position in the Organizational Space

The Intel iPSC/2 Hypercube

A cube is defined as a set of n processors, where n is a power of two, that are interconnected in such a way that the processors are located at the corners of a cube and the interconnections form the cube edges. There are several types of cubes, all variations on the

basic architecture, called the Boolean n cube, or binary cube. Each of the n nodes contains log n connections to its neighbor nodes. Each node is numbered in such a way that there is one binary digit difference between any node and its log n neighbors.

Cubes with dimensions greater than three are generally called hypercubes. Higher dimension cubes/hypercubes are constructed using lower dimension cubes/hypercubes as shown in Figure 3.8.

Dimension	*Nodes*	*Channels/Node*	*Channels*	*Topology*
0 D	*1*	*0*	*0*	
1 D	*2*	*1*	*1*	
2 D	*4*	*2*	*4*	
3 D	*8*	*3*	*12*	
4 D	*16*	*4*	*32*	

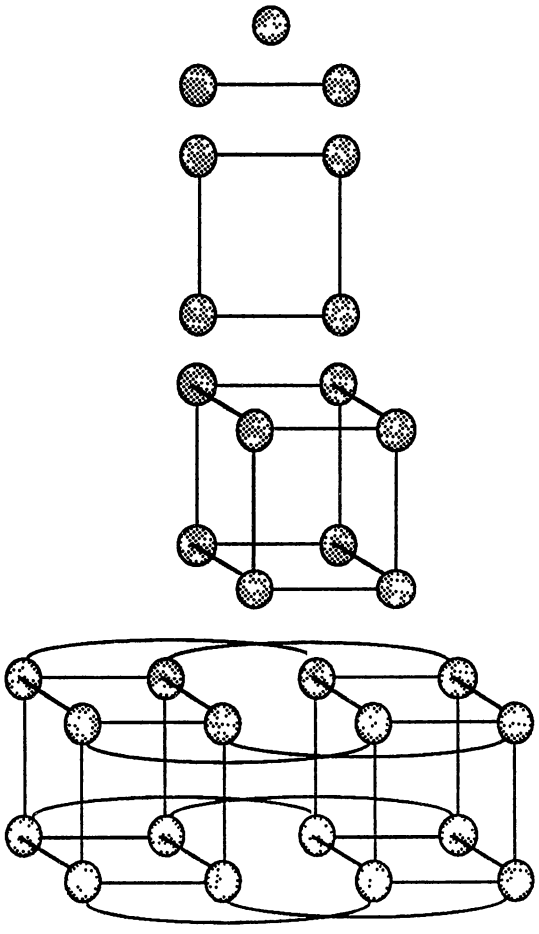


Figure 3.8 The Hypercube Topology

The basic concept for the Intel iPSC computer was proposed by and developed at the California Institute of Technology, under a project called the Cosmic Cube [47]. The project was sponsored by the United States Department of Energy and DARPA. Under a license from Caltech, Intel developed its own hypercube-based architecture, utilizing existing Intel microcomputer and communication components.

The Intel iPSC/2 computer consists of a hypercube based architecture along with an associated host processor called the cube manager. The iPSC/2 used in this research is a five dimensional hypercube. The iPSC/2 is expandable to a seven dimensional hypercube. The connection scheme of the hypercube is robust since there are several different paths that exist between any two nodes in the cube [21].

Each iPSC/2 node contains a 32-bit microcomputer based on the Intel 80386 processor with a fast scalar floating-point unit. Each node has its own memory and therefore the iPSC/2 is considered to be loosely coupled MIMD machine.

Nodes communicate with other nodes by sending and receiving messages. Message passing is the only means available for internode communication and synchronization, since the iPSC/2 has no shared memory. A Direct-Connect routing module is present in each node for high-speed message passing within the system's hypercube communication network. Each routing module provides an eighth channel for high-speed external communication (only seven are needed to connect up to 128 nodes). Messages on the iPSC/2 can either be synchronous or asynchronous. A call to the synchronous

message passing routines blocks until the message is sent or received before returning and allowing program execution to continue. On the other hand, a call to the asynchronous message passing routines returns immediately and does not block until the message is sent or received. The user has complete control over message passing using FORTRAN or C language extensions.

Figure 3.9 places the iPSC/2 hypercube architecture in the organizational space. The node processors are capable of performing a full range of operations since they incorporate general-purpose 32-bit microprocessors. For this reason, the iPSC/2 is considered to be fairly coarse grained. Since the complexity of the interconnection at each node is a maximum of five channels (for the 5-dimensional hypercube), and the communication between nodes does not have to be synchronized, the topology of the hypercube is of medium interconnection complexity. The iPSC/2 is an MIMD machine with each node having its own local instruction stream and communication is locally controlled making the overall system loosely coupled.

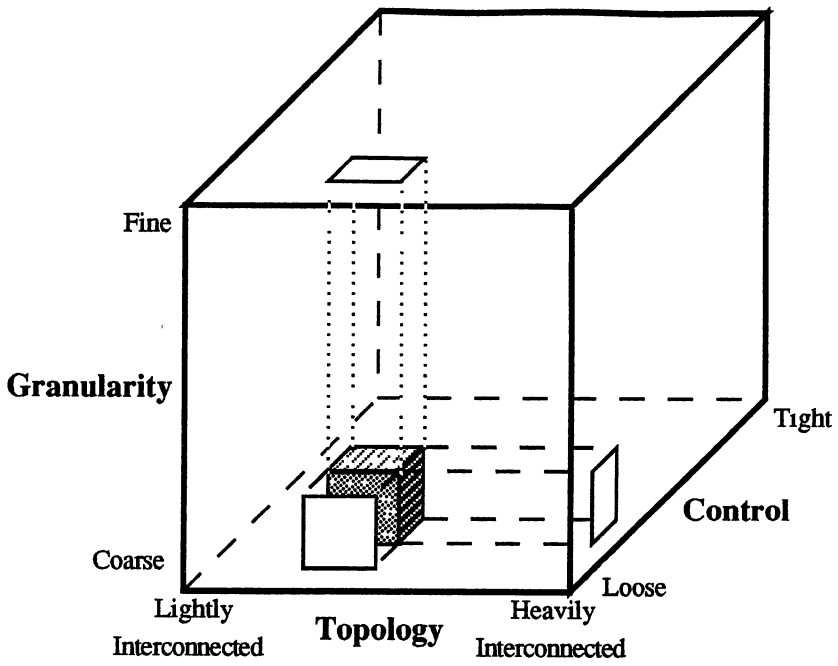


Figure 3.9 Hypercube Position in the Organizational Space

The Alliant FX/8

The Alliant FX/8 was designed to exploit parallelism found in scientific programs automatically. The intent was to allow parallel processing on existing FORTRAN programs with minimal or no changes to the source code. Figure 3.10 illustrates the hardware architecture of the Alliant FX/8. It consists of eight processors called Computational Elements (CEs), and 12 Interactive Processors (IPs). A common memory bus is employed for communication among resources. The CEs are connected via a crossbar switch to the cache modules attached to the memory bus. All access by the CEs and IPs to the bus occurs through cache memory modules. The CEs are also

connected directly to each other via a concurrency control bus. The Alliant FX/8 is a tightly coupled MIMD machine.

The CE is the computational building block of the Alliant FX/8 system. Each CE is a microprogrammed pipelined processor (compatible with MC68000 architecture) with integrated floating point and vector instruction sets. In general, the CEs are used to perform computation-intensive processes that can benefit from vectorization or loop-level concurrency, whereas the IPs are used to perform interactive processes and handle I/O between the memory and peripherals. The CEs are referred to as the computational complex and can be devoted to the execution of a single program.

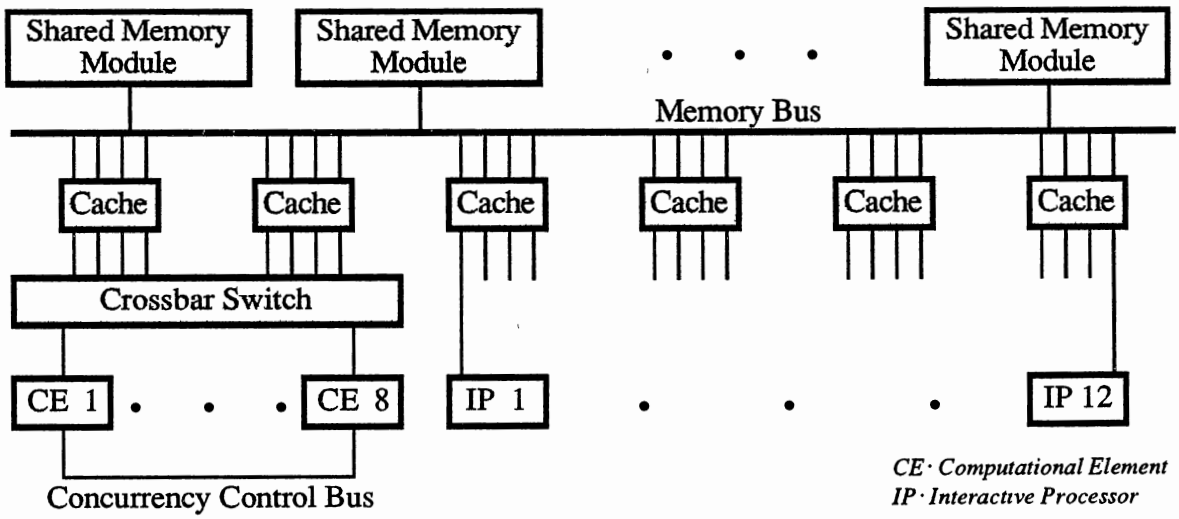


Figure 3.10 The Architecture of the Alliant FX/8

FORTRAN , C, and Pascal are supported on the FX/8 but only the FX/FORTRAN compiler provides optimization for concurrency and vectorization. Concurrency refers to the concurrent execution of loops and array operations by more than one CE. Vector operations are distributed across the CEs for concurrent execution in the same way as scalar operations.

When the FX/FORTRAN compiler recognizes an opportunity for concurrency, it generates concurrent code only as long as it can guarantee that this will not change the outcome of the program. In most cases, the compiler is very conservative in this regard. It bases its decision on the type of statements within a loop and the way variables are used, since the latter often affects the degree to which the iterations of the loop can be overlapped.

Concurrency is applied to DO loops by executing the different iterations on different CEs. Since there are eight processors, up to eight iterations can be active at one time. If necessary, the compiler inserts synchronization points into the object code to ensure that variables within a loop are updated and accessed in the correct order and to guarantee that the program statements following a loop do not execute until all iterations of the loop are finished.

Figure 3.11 illustrates the position of the Alliant FX/8 in the organizational space. The computer is a tightly coupled MIMD machine. The CEs are powerful processors that employ pipelining and vectorization suggesting that the FX/8 is of medium granularity. The interconnection between the CEs is simply a bus (concurrency bus). The CEs are connected via a crossbar switch to the cache memory modules which in turn connected to the shared memory via

a bus. For these reasons, the FX/8 is considered to be lightly connected. The control is fairly tight since a shared memory is used.

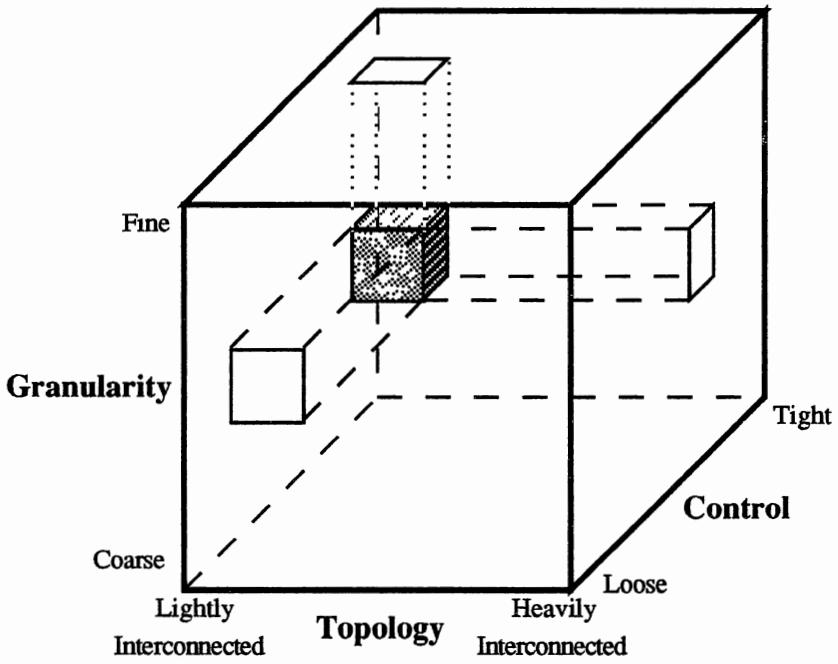


Figure 3.11 Alliant Position in the Organizational Space

The Massively Parallel Processor (MPP)

Among the experimental machines successfully built is the massively parallel processor (MPP) which was designed to process satellite imagery at high rates. In 1971, NASA Goddard Flight Center in Greenbelt,Md., initiated research for a high speed computer to process the data generated by orbital imaging sensors. Data rates of

1013 bits/day is expected to result in 109-1010 operations/sec workload. Designed and built for $6.7 million by the Goodyear Aerospace Corp. in Akron, Ohio, the MPP was delivered to NASA in May 1983.

One of MPP's first tests involved analyzing data from the "thematic mapper" aboard Landsat 4. By studying the million or so pixels making up a typical image, the MPP automatically finds out whether each spot represents water or land, forest or field, stream or street - all in 20 seconds. A conventional computer would take hours to analyze the same picture and produce a similar classification scheme.

The MPP Architecture [48]

The MPP owes its speed to the unusual way in which the machine's parts are organized. Its network of 16384 simple processors allows a problem to be divided up so that each processor performs the same operation on different pieces of data at the same time (SIMD architecture).

Figure 3.12 depicts the overall system block diagram. There are five main subsystems: The array unit (ARU), the array control unit (ACU), the program and data management unit (PDMU), the staging memories (SM), and the host computer.

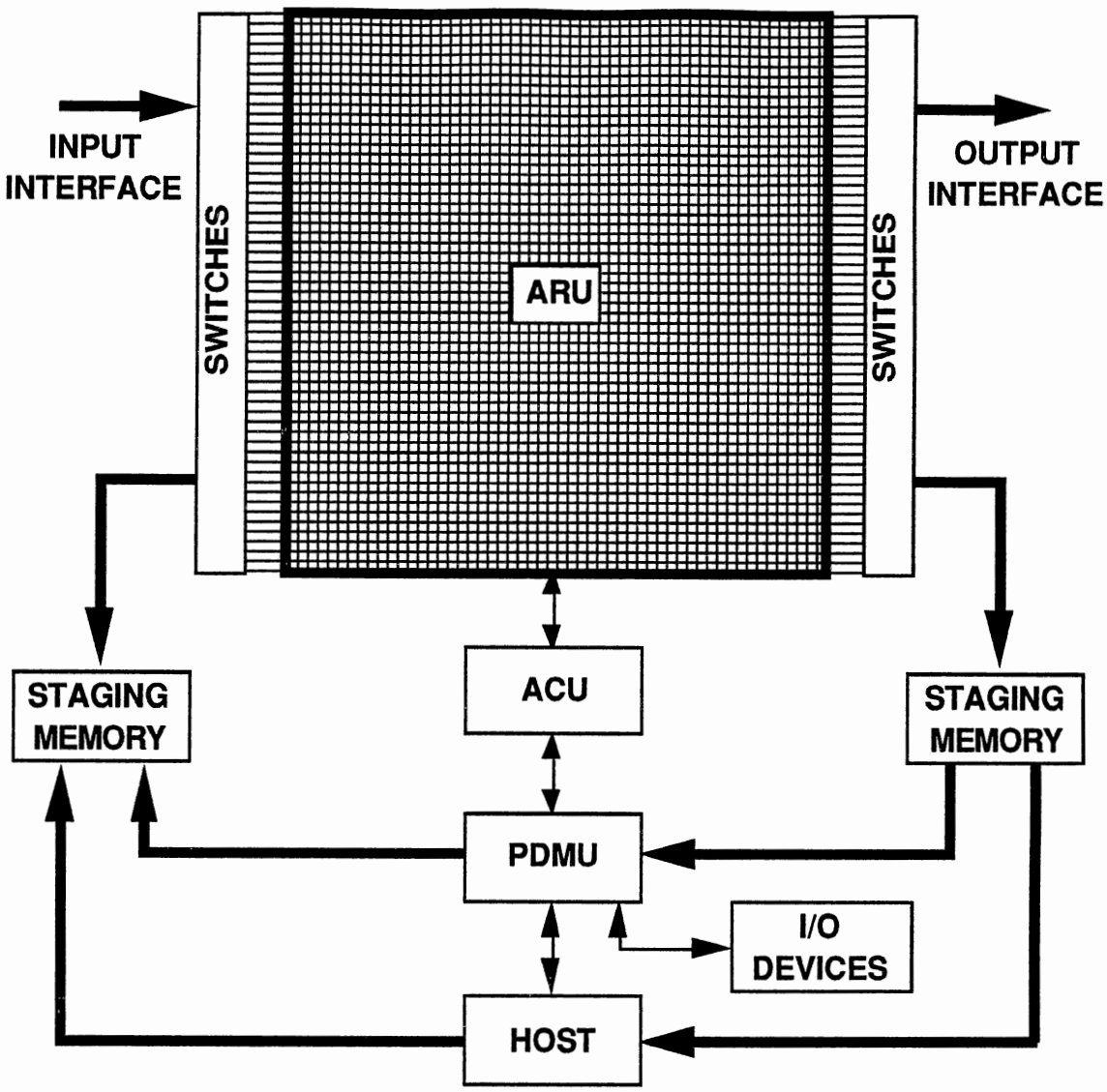


Figure 3.12 Overall Block Diagram of the MPP

The Array Unit. The ARU of the MPP is organized with a number of 16384 -element planes to handle the two-dimensional data processing at high speed. Each plane is a square with 128 rows and 128 columns. Figure 3.13 shows 1 column of the ARU. The ARU contains one S-plane (used to handle data input and output for the

ARU), 1024 memory planes, and 35 processing planes for a total of 1060 planes. Each plane also has 4 spare columns to bypass faulty hardware.

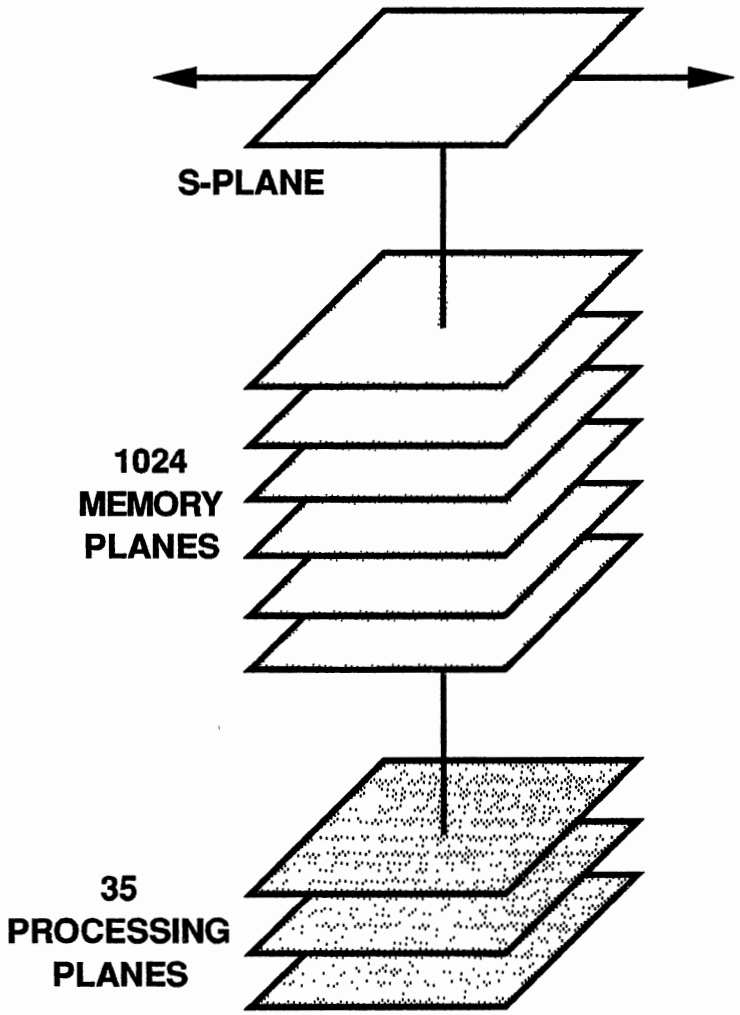


Figure 3.13 The ARU

Instructions operate on a whole plane of data in parallel. The ARU can treat data of arbitrary precision since the processing is bit-serial. Black-white images are stored and processed as arrays of single-bit pixels, images with 256 grey levels are stored and processed as arrays of 8-bit pixels.

Each plane is organized in a mesh with nearest neighbor communication between the elements in that plane. This architecture facilitates data accessability and is easy to implement in hardware. The edge connectivity is programmable offering the user eight different topologies. Figure 3.14 illustrates these topologies. For the East-West edges there are 4 possible options: open, cylindrical, open spiral, or closed spiral. For the North-South edges there are 2 possible options: open or connected.

Programmability is achieved using the topology register in the ACU. Note that topologies 4-7 convert the two-dimensional ARU into one-dimensional structure that can be used for one-dimensional signal processing problems.

Processing elements in the ARU are designed with two-row by four-column custom made VLSI chips (HCMOS technology). The processing element array has 128 rows and 132 columns that are divided into 33 groups, each of which consists of 128 rows by 4 columns. Each of the 33 groups has an independent group-disable control line from the ACU that is activated if a faulty processing element is detected. Arbitrary disable is used if no fault is detected. The programmer does not need to alter the program when the disabled group is changed since logical addresses are used.

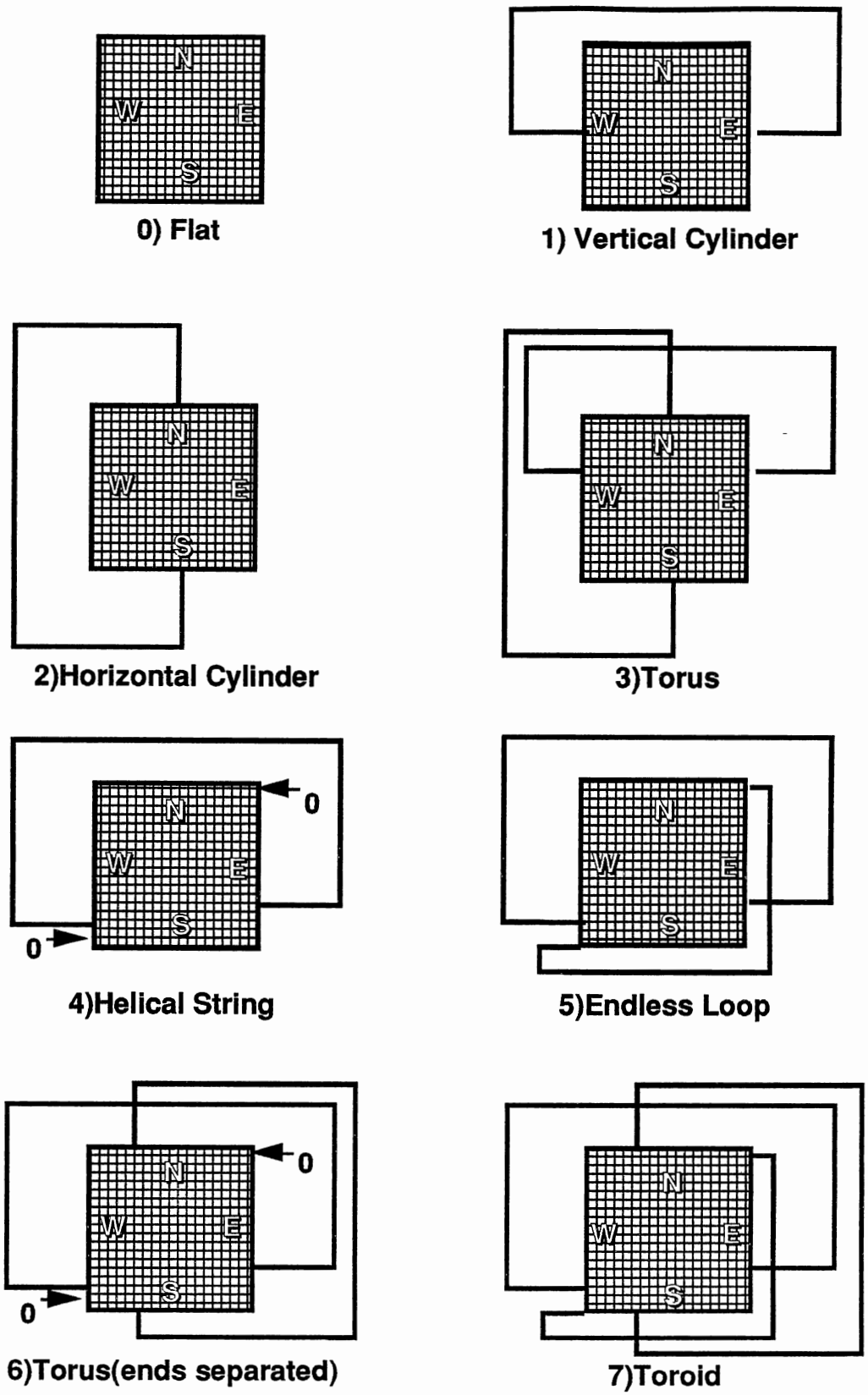


Figure 3.14 Topologies Available on the MPP's ARU

Figure 3.15 shows one processing element (PE) in the ARU. The PE has six 1-bit registers (A,B,C,G,P,and S), a planar shift register with a programmable length (2,6,10,14,18,23,26,or 30), RAM, data bus, full adder, and some logic circuits. The S-register is part of the S-plane that handles data input and output for the ARU. On input, the S-plane accumulates a plane of data, column by column and then transfers the data plane to a memory plane. On output, the S-plane receives the contents of a memory plane and then transfers the plane out column by column. Input and output can be handled simultaneously.

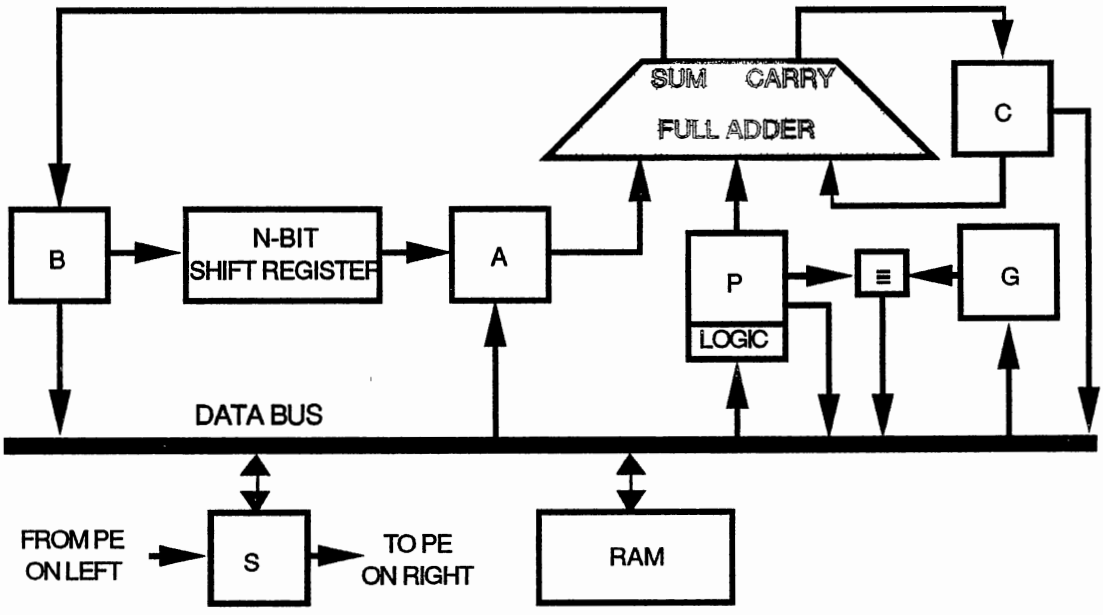


Figure 3.15 The Processing Element

The A-register is part of the A-plane that receives the output of the planar shift register. It can be considered to be a one-plane extension to the depth of the planar shift register. The B-plane is the sum plane in arithmetic operations, while the C-plane is the carry plane. The G-plane is used to mask activity in the other processing planes, while the P-plane is used for logic and routing operations.

The RAM stores 1024 bits per PE with addresses in the range 0 to 1023. The ACU generates 16-bit addresses so that ARU storage can be expanded to 65536 bits per PE. Memory faults are detected using parity check that sets an error flip-flop associated with the 2 by 4 subarray.

The S-plane and the processing planes are implemented with 2112 custom VLSI circuits. The memory planes are implemented with 4752 standard bipolar RAM integrated circuits- each RAM circuit contains 4 data bits or 4 parity bits of all 1024 memory planes. Twenty four VLSI circuits and 54 RAM circuits are packaged on one printed-circuit board to make up a 16 row by 12 column section of the ARU planes. The 128 row by 132 column ARU requires 88 printed-circuit boards. Another 8-boards are used for the topology switches around the edges of the P-plane and to distribute the control signals from the ACU.

The Array Control Unit. The ACU controls the operations in the ARU and performs the arithmetic on any scalars required to support operations on data arrays in the ARU. Figure 3.16 shows a block diagram of the ARU that consists of: the processing element control

unit (PECU), the input/output control unit (IOCU), the main control unit(MCU), a queue, and memory for both the PECU and the MCU.

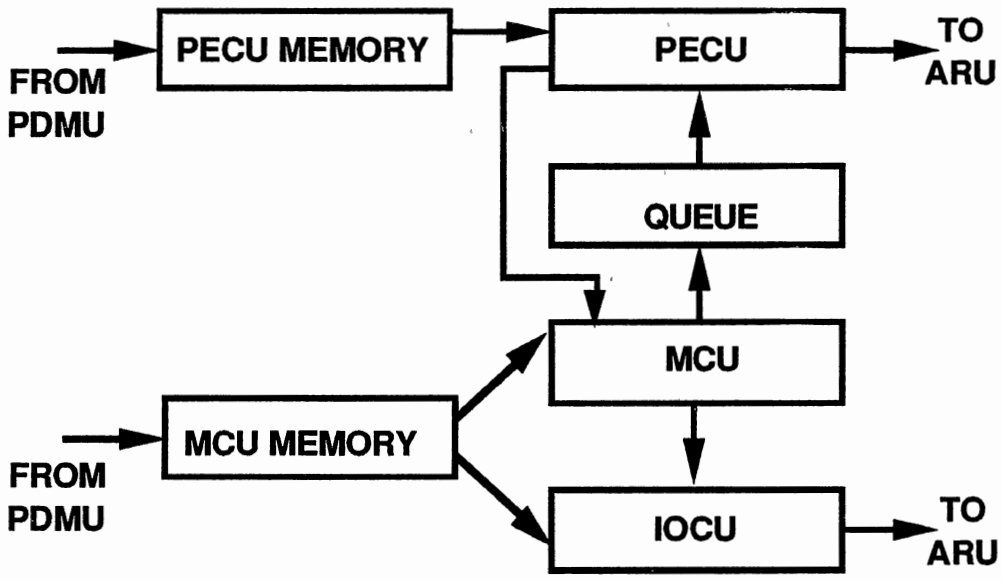


Figure 3.16 The Array Control Unit

The PECU controls operations in the processing planes of the ARU. It generates all ARU instructions except those pertaining to the S-register. It executes microcoded routines stored in its program memory to perform all array operations required by application programs. The PECU contains 8 index registers, a 64 bit common register for scalar data, a topology register, a program counter, a subroutine stack, and an instruction register.

The IOCU controls the shifting of I/O data through the ARU S-registers as well as the transfer of I/O data between the S-registers

and the ARU memory. It executes I/O channel control programs stored in the MCU program memory.

The MCU executes the application program stored in its program memory. It performs the scalar arithmetic operations required, calls the PECU for all array logic and arithmetic operations. Both sets of calls are queued to await execution while the MCU moves on to generate other calls.

The queue holds calls to the array processing routines until they are executed by the PECU. A call enters the queue when inserted by the MCU and remains there until the PECU has executed all previously called routines, then the PECU jumps to the called routine. Up to 16 calls can be held in the queue at one time.

The Program and Data Management Unit. The PDMU is a DEC PDP11/34A minicomputer. It controls the overall flow of programmed data in the system and it has the RSX-11M real time multiprogramming operating system. The PDMU executes the program development software package written in FORTRAN. This package includes the main assembler, the PE control assembler, a linker, and a control and debug module.

The main assembler is used to develop application programs executing in main control, while the PE control assembler is used to develop array processing routines for PE control. The linker is used to form load modules for the ACU. Finally, the control and debug module is used to load programs into the ACU, control and supervise the execution, and facilitate debugging.

The Staging Memories. The MPP system includes a staging memory for buffering ARU data. This memory provides both the "corner turning" function, which converts conventional byte or word oriented data into the bit plane form needed by the ARU, and the "multi-dimensional access" function which allows large multi-dimensional arrays of data located in the staging memory to be read out or written in along arbitrary orderings of array dimensions. The current capacity of the staging memory is 32 Mbytes and is upgradable to 64 Mbytes.

Data moves between the ARU and the staging memory via 128 parallel lines. The upper limit on the transfer rate is 1.28 billion bits/second. The MPP currently supports 64 billion bits/second. Data movement in both directions can be overlapped with processing.

The Host Computer. The host computer is a DEC VAX 11/780 minicomputer. It manages data flow between the MPP units, loads programs into the ACU, executes system test and diagnostic routines. The MPP is interfaced to the host through a 5Mbytes/second DR-780 channel. The custom interface is used to switch the MPP from PDMU to host and is facilitated by the DEC UNIBUS.

MPP Classification

The MPP is situated in the organizational space as shown in Figure 3.17. The MPP is a bit-slice machine with relatively special purpose processing elements and therefore considered finely grained. The topology of interconnection between processing elements is fairly dense and highly synchronized making the MPP

heavily interconnected. As previously mentioned, the operation of all processing elements is synchronized so that a single operation is performed on all data at once. The control of this action resides within the control unit at all times and is always synchronized with the main processor clock. For this reason, the MPP is classified as tight.

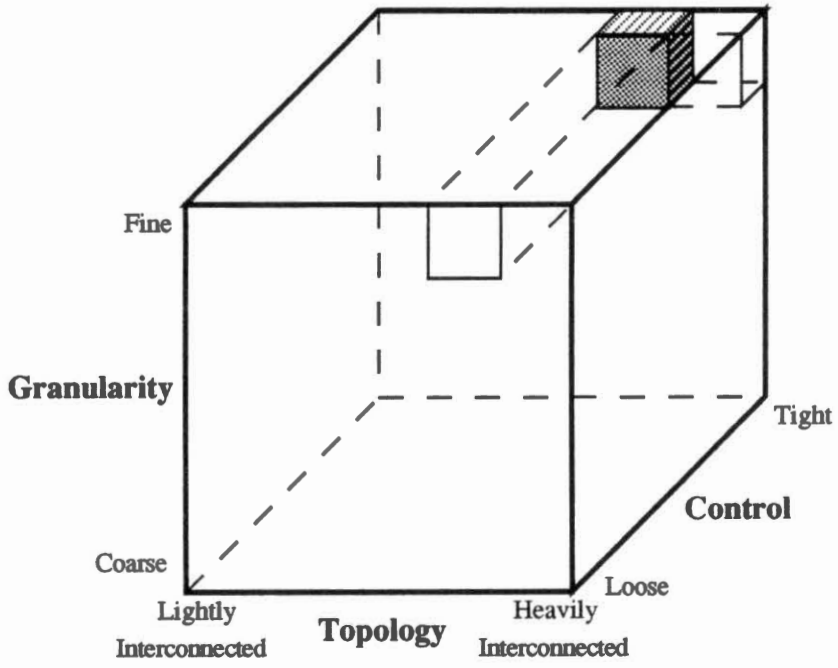


Figure 3.17 MPP Position in the Organizational Space

The MPP Programming Environment

The initial high level language implemented in 1983 was Parallel Pascal. It was designed to be independent of computer architecture,

thus allowing portability of application programs between diverse parallel computers having Parallel Pascal compilers. Experience in the development and use of this approach showed that the 128 by 128 square grid architecture of the MPP could not be hidden from the programmer using current compiler writing technology.

A modified language, MPP Pascal, was then implemented that is architecture dependent and that possesses important semantic features allowing the programmer to make efficient use of the hardware capabilities. This compiler is sufficiently flexible to allow easy modifications.

The compiler accepts most of the standard Pascal code but after some modifications to handle parallel arrays. A parallel array is stored in array memory and all operations on the array are performed in parallel. The following instruction declares "parray" as a parallel array of integer values with dimensions 128 by 128:

```
type
parray=parallel array [1..128,1..128] of integer;
```

Notice that the parallel arrays on the MPP must have the last two dimensions equal to 128. Figure 3.18 illustrates an example of a simple addition of two parallel arrays and the way they are stored in the ARU. Parallel array 'A' is a two dimensional array while 'B' is three dimensional.

```
Var
  A:Parallel Array [1..128,1..128] of Real;
  B:Parallel Array [1..2,1..128,1..128] of 1..256;
  I:Integer;

Begin
  A:=0.0;
  For I:=1 to 2 Do
     A:=A+B[I];
End
```

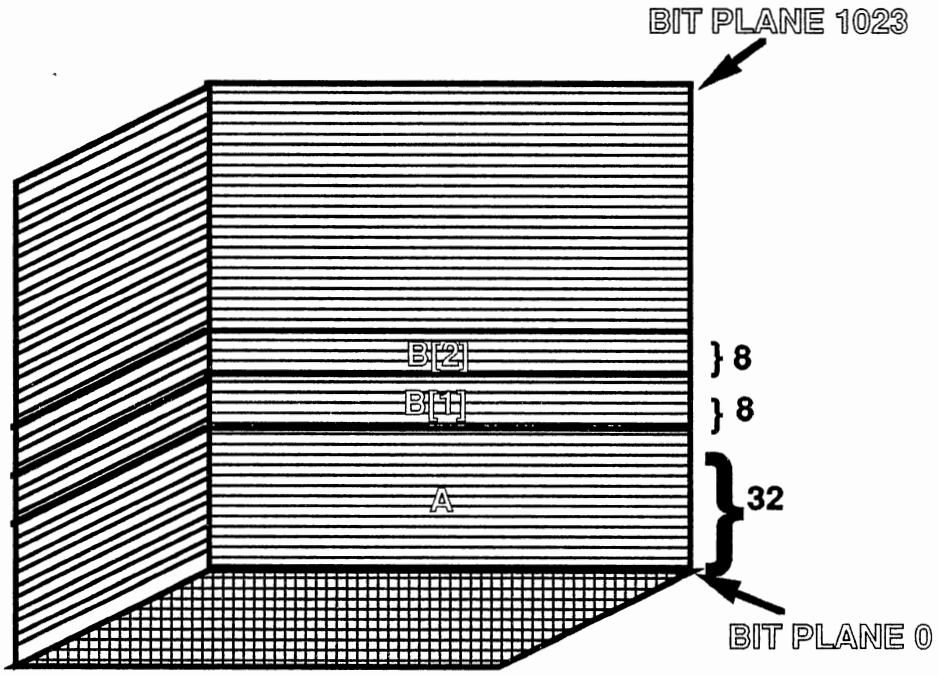


Figure 3.18 Example of MPP Pascal Code and Storage

There are two classes of standard functions that have been defined in Parallel Pascal: reduction functions and permutation

functions. Reduction functions reduce the rank of an array. The first argument of such a function specifies the array to be reduced and is followed by arguments that specify which dimensions are to be reduced. Table 3.1 summarizes these functions.

Table 3.1

Reduction Functions

Syntax	Meaning
sum(array, D1,...,Dn)	arithmetic sum
prod(array,D1,...,Dn)	arithmetic product
all(array,D1,..,Dn)	Boolean AND
any(array,D1,...,Dn)	Boolean OR
max(array,D1,...,Dn)	arithmetic maximum
min(array,D1,...,Dn)	arithmetic minimum

Permutation functions are primitive operations that involve data movement. Four functions are available and tabulated in Table 3.2.

Finally, special block structures were added to the Standard Pascal. An interesting one is the "when-do-otherwise" structure which is the equivalent to the combination of "for-do" and "if-then-else" structures.

Table 3.2

Permutation Functions

Syntax	Meaning
shift(array,S1,S2,1..,Sn)	End-off shift data in array
rotate(array,S1,S2,1..,Sn)	Circularly rotate data in array
transpose(array,D1,D2)	Transpose two dimensions
expand(array,dim,range)	Expand array along dimension

The Connection Machine Model CM-2

The Connection Machine was conceived at MIT's AI Laboratory for concurrent manipulation of knowledge stored in semantic networks. Figure 3.19 shows the block diagram of the system constructed by Thinking Machines Corporation at Cambridge, Massachusetts. This is an SIMD machine having 64K simple processor/memory cells linked by a 12-dimensional hypercube network. The hypercube topology is distinguished by its symmetry, small diameter, and multiplicity of paths between any two nodes. It is amenable to a layout with high packing density and short average wire length.

A full CM-2 is made up of four sections of 16K processors each. Each section is termed a sequencer and can be used separately, in pairs, or as one massive processor unit.

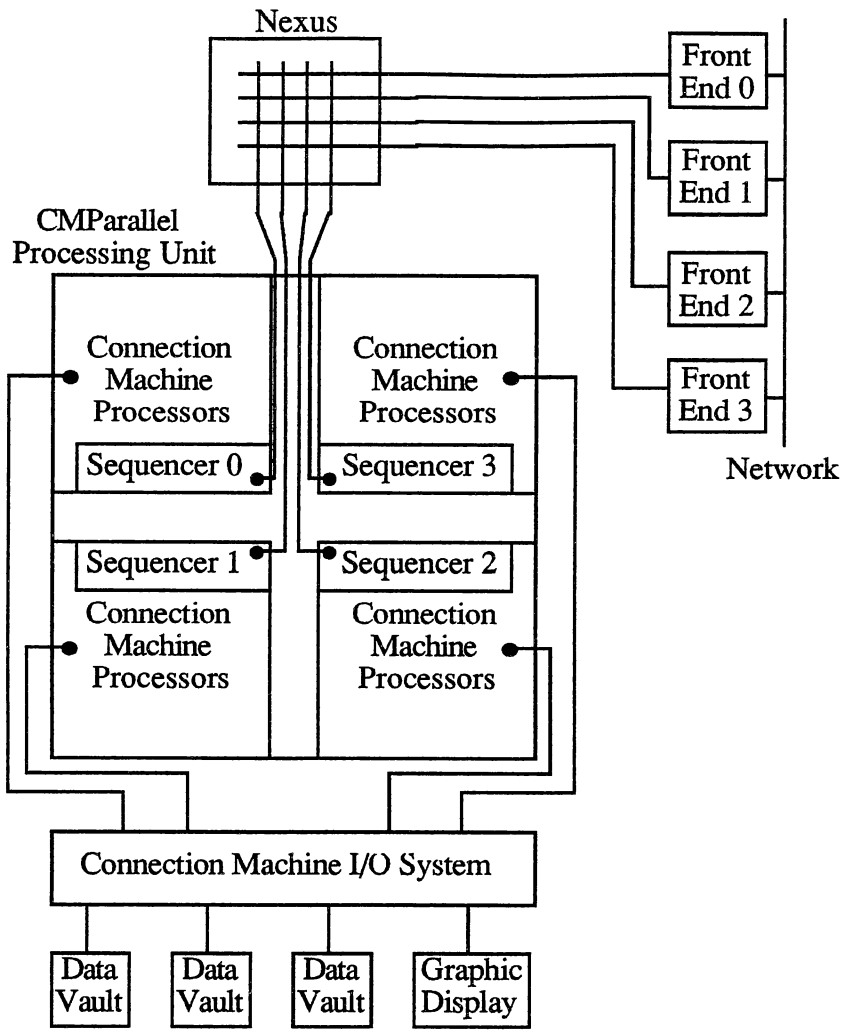


Figure 3.19 The Connection Machine CM-2 Overall Block Diagram

Aside from its processors, each section contains interprocessor communication hardware, a sequencer, and optionally, a set of peripherals. The interprocessor communication hardware is used for communication between any processors controlled by one host. A sequencer receives macroinstructions from the front end and broadcasts sequences of microinstructions to all the processors in its

section. A typical macroinstruction would be to add two 32-bit numbers and store them in a particular location. Because of the simplicity of the processors, each macroinstruction is typically implemented by many microinstructions. Thus the microcontroller acts effectively as a bandwidth amplifier for the instruction stream coming from the host.

The front end computer, or host, attaches to the microcontroller through a bidirectional crossbar called a Nexus, and controls one, two, or four 16K processor sections.

Although the front end can be used to supply data to the CM-2 computer, in many applications the CM-2 processors can process data much faster than the front end can supply it. For this reason, the CM-2 processors are connected to a high-speed bidirectional bus. Special disk drives, frame buffers, frame grabbers, and specialized I/O devices are connected to this high bandwidth bus.

Pointer referencing (or referencing the data in one data object from another) on the CM-2 computer requires interprocessor communication. Since a CM-2 processor needs to be able to access data from any other processor, this intercommunication system has to handle a large load at high speed. This is achieved by the use of a router which is integrated into the architecture so that every processor's memory is easily accessible to every other processor. The result is that the application programmer does not have to worry about physical processor geometry since the router handles all interprocessor communication efficiently regardless of layout. Figure 3.20 examines the sequencer architecture more closely.

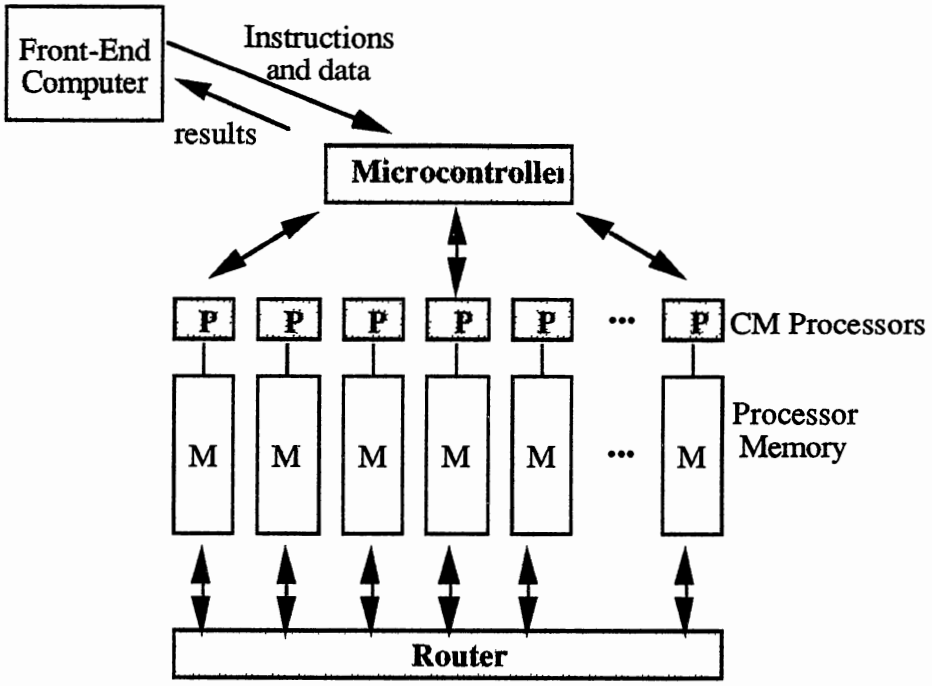


Figure 3.20 The Architecture of a Sequencer

Suppose a processor P(i,j) on chip i wants to communicate with processor P(k,l) on chip k. It first sends the message to the router R(i) on its own chip, using a simple hand-shaking mechanism. This router forwards the message to router R(k) on chip k. Finally, R(k) delivers the message to the appropriate memory location. The routing algorithm used by the router moves messages across each of the 12 dimensions of the hypercube in sequence. If there are no conflicts, a message will reach its destination within one cycle of this sequence, since any vertex of the cube can be reached from any other by traversing no more than 12 edges.

Along with general pointer referencing, two-dimensional interprocessor communication is supported. In this type of

communication, termed NEWS, each data object can communicate with its two-dimensional neighbor (to the north, east, west, or south). This intercommunication is handled by a slightly different mechanism and is faster than using the general communication mechanism. Image processing applications often use this ability to move data from one pixel the next.

The Connection Machine Model CM-2 is a data parallel computing system. Data parallel computing associates one processor with each data element. This computing style exploits the natural computational parallelism inherent in may data-intensive problems. It can significantly decrease the execution time of a problem, as well as simplify its programming. In the best cases, execution time can be reduced in proportion to the number of data elements in the computation.

The central element in the system is the CM-2 parallel processing unit, which contains:

- thousands of data processors
- an interprocessor communication network
- one or more sequencers
- one interface to one or more front-end computers
- zero or more I/O controllers and/or frame buffers

As previously mentioned, a parallel processing unit may contain 64K, 32K, or 16K data processors. Each data processor has 64 K bits (8Kbytes) of bit-addressable local memory and an arithmetic-logic unit that can operate on variable-length operands. Each data processor can access its memory at a rate of at least 5Mbits per

second. A fully configured CM-2 thus has 512 Mbytes of memory that can be read or written at about 300 gigabits per second. When 64K processors are operating in parallel, each performing a 32-bit integer addition, the CM-2 parallel processing unit operates at about 2500 Mips. In addition to the standard ALU, the CM-2 parallel processing unit has an optional parallel floating point accelerator that performs at 3500 MFlops (single precision) or 2500 MFlops (double precision).

The CM-2 parallel processing unit contains thousands of data processors. Each data processor contains:

- an ALU and associated latches
- 64K bits of bit-addressable memory
- four 1-bit flag registers
- optional floating point accelerator
- router interface
- NEWS grid interface
- I/O interface

The data processors are implemented using four chip types. A proprietary custom chip contains the ALU, flag bits, router interface, NEWS grid interface, and I/O interface for 16 data processors, and also contains proportionate pieces of the router and NEWS grid network controllers. The memory consists of commercial RAM chips. The floating point accelerator consists of a custom floating point interface chip and a floating point execution chip; one of each is required for every 32 data processors. A fully configured parallel processing unit contains 64K data processors, and therefore

contains 4096 processor chips, 2048 floating point interface chips, and 2048 floating point execution chips, and half a gigabyte of RAM.

The CM-2 ALU consists of a 3-input, 2-output logic element and associated latches and memory interface. The basic conceptual ALU cycle first reads two data bits from memory and one data bit from a flag; the logic element then computes two result bits from the three input bits. Finally, one of the two results is stored back into memory and the other result into a flag. One additional feature is that the entire operation is conditional on the value of a third flag; if the flag is zero, then the results for that data processor are not stored after all.

The logic element can compute any two boolean functions on three inputs; these functions are simply specified (by the sequencer) as two 8-bit bytes representing the truth tables for the two functions.

Since each cell can perform only extremely simple tasks, the real power of the CM derives from its ability to store information in the reconfigurable virtual interconnection patterns among the cells, and from the concurrent execution of the same simple operation on a very large number of cells.

The Connection Machine (CM-2) system consists of a collection of simple processors, each with its own memory, all acting under the direction of the front end. Since many data sets are larger than even the largest CM, the system uses a virtual processing mechanism, whereby each physical processor simulates some number of virtual processors by subdividing its memory, to ensure that a unique processor is assigned to each element of the data.

Virtual processors are a software abstraction, implemented at the microcode level, which allows a programmer to write programs that are independent of the number of physical processors that the CM-2 hardware contains. Virtual processors are implemented using three separate mechanisms: one for storage, one for processing, and another for communication. First, the memory of each physical processor is divided evenly among the virtual processors assigned to it. The number of virtual processors per physical processor is referred to as the virtual processor ratio.

The second mechanism necessary to support virtual processors is time-multiplexing of the physical processors among the virtual processors assigned to it. Every macroinstruction sent by the front end is run on each of the virtual processors within each physical processor. The overhead for switching context is extremely small (about the time to execute a 2-bit add) because each processor is so simple.

The third mechanism of communication allows the CM-2 processors to communicate with one another without regard to virtual processors. Grid communication is handled by the microcode by sharing the grid wires. General communication (pointer reference) is handled by the router hardware. The length of a processor address changes based on the number of virtual processors in the machine. This virtual address is used by the router hardware to deliver messages to the correct processor.

The CM-2 is situated in the organizational space as shown in Figure 3.21. The CM-2 is an SIMD machine with massive number of simple processors and therefore considered very finely grained. The

topology of interconnection is a 12 dimensional hypercube and is considered heavily interconnected. Notice that because there is only one instruction stream, the CM-2 processors are naturally synchronized. No processor can proceed to the next instruction until all have finished the current instruction. Therefore, the CM-2 is classified as tightly controlled architecture.

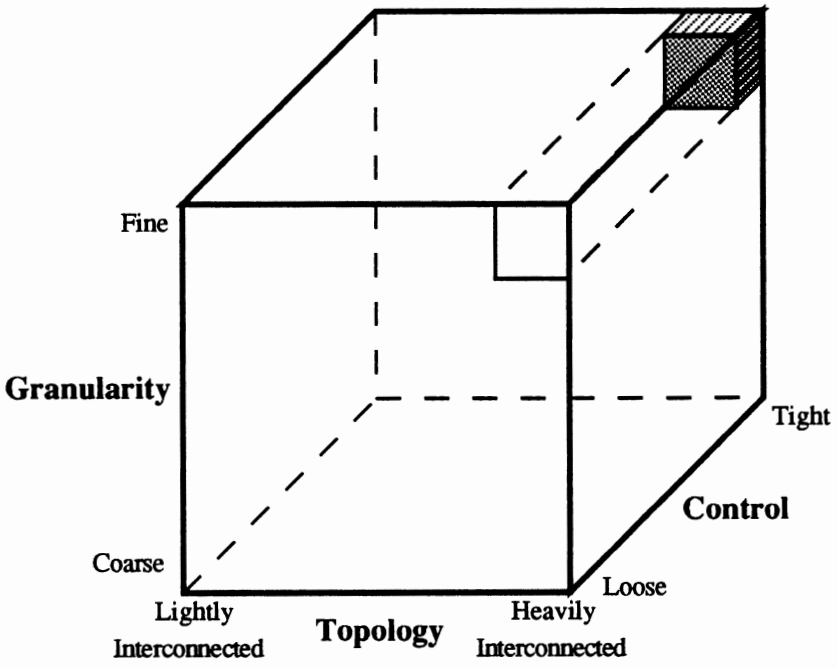


Figure 3.21 CM-2 Position in the Organizational Space

The Cray-2 Supercomputer

The Cray-2 supercomputer is the latest Cray to be produced. It is a tightly coupled MIMD supercomputer that contains four Cray-1

like processors that share memory and I/O subsystems and has a clock cycle of about 4.5 nanoseconds. The architecture is very similar to that of the Cray X-MP/48 but with much improved and faster vector processors. Table 3.3 compares the different Cray computer showing the number of CPU's used, individual processor potential (Cray-1 processor is used as a baseline), and finally the total system potential.

Table 3.3

Comparison of Cray Supercomputers

Model	# of CPU	Proc. Pot.	Tot. Sys. Pot.
Cray-1	1	1	1
Cray X-MP	2	1.32	2.64
Cray X-MP	4	1.47	5.88
Cray 2	4	3.05	12.20

The designer of the Cray supercomputers is Seymour Cray who concentrated on the science and art of arranging chips on circuit boards and plotting the interconnections among those boards in a way that minimizes the physical distance that electrical signals must

travel along the data path. The importance of packaging is magnified in supercomputers because the processors are so fast that one of the main constraints on speed is the time required for signals to pass through wires. Light travels about a foot per nanosecond, but due to the resistance of wiring, electrical pulses can manage 4 to 9 inches. The longest wire in a Cray-1 is 4 feet, but in the Cray-2 that distance has dropped to about 18 inches.

There is a price to pay for increased speed, however. The power consumption of the machine is enormous for its size and most of that is converted to heat. In order to dissipate the heat generated by the 300K integrated circuit chips in the Cray-1, the refrigerant Freon is circulated through channels in each layout board. To perform the same task in the more densely packed Cray-2, the entire system must be immersed in a bath of liquid fluorocarbon.

The position of the Cray-2 in the organizational space is similar to that of the Cray X-MP/48 illustrated in Figure 3.7. The main difference is the improved processor capability.

Chapter Summary

This chapter described a new classification scheme for computer architectures and described several commercially available machines. Throughout the chapter, these machines were classified according to the new classification scheme. The new classification scheme maps each architecture into a box in a defined three dimensional space. The results of the classification show that the architectures occupy a substantial volume of the three dimensional

space making them good representatives of the available computer systems.

In the next chapter, implementation results of the selected algorithm on the computer systems described in this chapter are reported. A preliminary judgement on performance is also reported.

CHAPTER IV

IMPLEMENTATIONS OF THE BURG ALGORITHM

The Burg algorithm, described in chapter II, is a procedure for fitting an autoregressive model to a time series data set. It is widely used in such areas as seismic data processing, spectral estimation, speech signal analysis, and biomedical signal processing. The main step in this algorithm is a time shift/inner product operation. This step is also a key to a number of other signal processing algorithms; any algorithm which involves a convolution or a correlation operation will have this step as a major component. Because it is representative of such a large class of signal processing algorithms, the Burg algorithm was chosen for this study of the mapping of batch signal processing algorithms onto general purpose parallel computers.

Seven parallel machines, described in chapter III, were chosen for this study, representing a variety of available modern computer architectures. The Burg algorithm has been implemented on each of these machines: the Intel iPSC/2 hypercube computer, a distributed memory (loosely coupled) MIMD machine; the Denelcor HEP, a shared memory (tightly coupled) MIMD machine with switch network interconnect architecture; the NASA/Goodyear MPP, a massively parallel SIMD machine with mesh interconnect

architecture; the Cray X-MP/48, a shared memory (tightly coupled) MIMD supercomputer with direct connect interconnect architecture; the Alliant FX/8, a shared memory (tightly coupled) MIMD machine with a bus interconnect architecture; the Connection Machine model CM-2, a massively parallel SIMD machine with hypercube interconnect architecture; and the Cray-2 supercomputer, a tightly coupled MIMD machine with direct connect interconnect architecture and is the latest Cray to be produced. These seven architectures provide a variety of interesting mapping problems for the algorithm. An analysis of the algorithm's performance on these machines will assist in the determination of the optimal architecture for this problem.

This chapter describes the parallel implementation of the algorithm on the seven architectures and gives an analysis of speedup characteristics. Finally, there will be a preliminary comparison of the performances of the seven machines and a discussion of the results.

Sequential Implementation [30,49-51]

A standard sequential implementation of the Burg algorithm, which was described in section 2.2 is shown in Figure 4.1. In this figure the individual computations, or tasks, are labeled $T_{in}(1)$, $T_n(1)$, $T_{in}(2)$, $T_{nn}(2)$, $T_{in}(3)$.

Tasks $T_{in}(1)$ are the computations of the inner products which are found in the numerator and denominator of Equation (2-20). Task $T_n(1)$ is the simple division needed to compute c_{m+1}. This task can cause a bottleneck in the parallel implementations, as is

discussed in later sections. Tasks $T_{in}(2)$ update the autoregressive coefficients. (These tasks may not always be necessary.) Tasks $T_{in}(3)$ update the forward and backward prediction errors using Equations (2-14) and (2-15).

```
1. INITIALIZATION
   FOR i=1 TO M DO
      e(i)=x(i)
      b(i)=x(i)

2. THE MAIN LOOP
   FOR n=1 TO MAX DO
      s1=0.0 ; s2=0.0
      FOR i=n+1 TO M DO
         s1=s1+e(i)*b(i-n)                 T_in(1)
         s2=s2+e(i)**2+b(i-n)**2
      c(n)=-2.0*s1/s2                      T_n(1)
      IF n>1 THEN DO
         FOR i=1 TO n-1 DO
            a1(i)=a(i)+c(n)*a(n-i)         T_in(2)
         FOR i=1 TO n-1 DO
            a(i)=a1(i)
      a(n)=c(n)                            T_nn(2)
      FOR i=n+1 TO M D
         temp=e(i)+c(n)*b(i-n)             T_in(3)
         b(i-n)=b(i-n)+c(n)*e(i)
         e(i)=temp
```

Figure 4.1 Sequential Implementation of the Burg Algorithm

Parallel Implementations [49,51]

To implement the Burg algorithm using parallel techniques we need to determine which tasks can be performed in parallel. Figure

4.2 illustrates the relationships between the various tasks for the case where there are 5 data points (M=5) and 3 coefficients to be calculated (MAX=3). Any tasks which are on the same level can be performed at the same time.

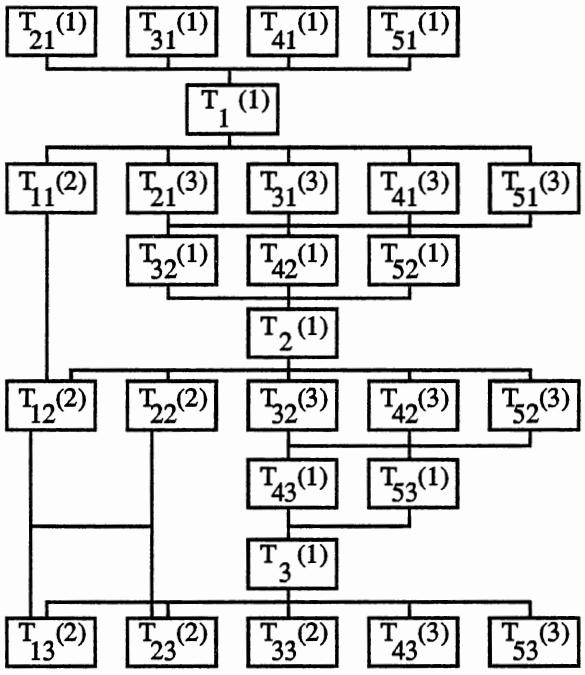


Figure 4.2 Maximally Parallel Graph for M=5 and MAX=3.

MPP Implementation [49,51]

As illustrated by the maximally parallel graph, we would need to have M processors (where M is the number of data values) to take full advantage of the parallel nature of the algorithm. This is clearly

infeasible on machines like the Denelcor HEP or the Intel iPSC/2 hypercube computer, but is feasible on the NASA/Goodyear MPP due to its massive number of processors - 16,384.

The MPP is a two dimensional mesh-connected architecture with nearest neighbor communication between the processing elements in the array unit (ARU). This type of architecture is most suitable for the processing of two-dimensional images; the Burg filter is a one-dimensional signal processing algorithm. The implementation problem reduces to finding a way to map the one-dimensional structure inherent in the Burg filter onto the two-dimensional architecture of the MPP.

A miniature (16 elements) ARU is shown in Figure 4.3, with arrows representing the required connections or communication channels needed to view the mesh architecture as a linear array of processors, which would be most suitable for this implementation of the Burg filter.

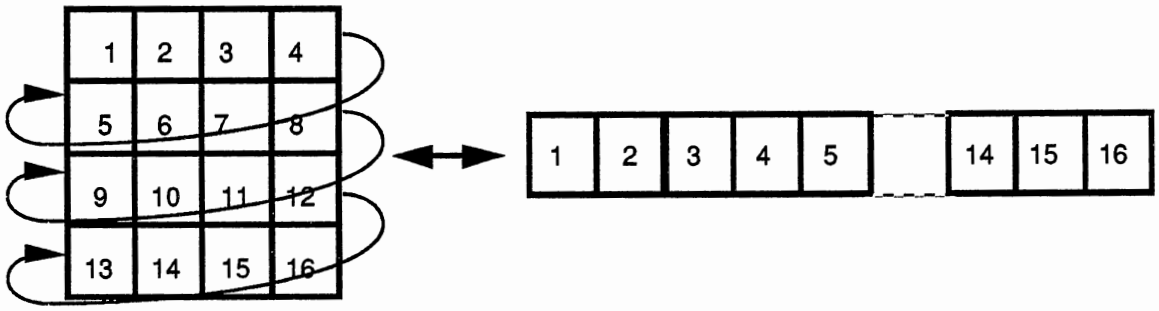


Figure 4.3 Mapping a Linear Array on a Mesh

Figure 4.4 illustrates the data movement for the Burg filter in a linear array of eight processing elements. In stage 0 the linear array is loaded with both the forward and backward prediction errors, which are equal to the observed time series at this stage. To calculate the first reflection coefficient the forward prediction errors are shifted to the left by one, as shown in stage 1. Now the reflection coefficients can be determined by forming the two sums; the first is the sum of the products of the two elements in each processing element, the second is the sum of the squares of the two elements in each processing element. Equation (2-20) can then be used to calculate the reflection coefficient that will be broadcast to all the processing elements, where it will be used to update the forward and backward prediction errors. To calculate the second reflection coefficient the updated forward prediction errors are again shifted to the left by one, as shown in stage 2. The above sequence of operations will be repeated until all reflection coefficients are computed.

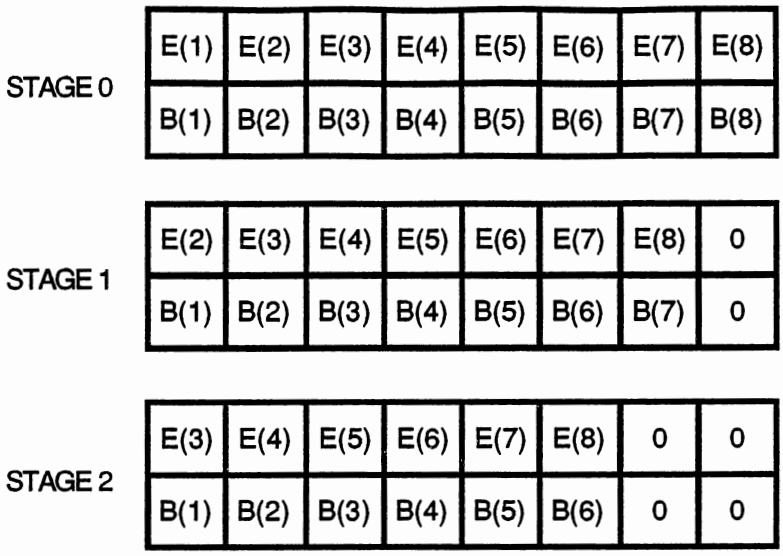


Figure 4.4 Data Movement For The Burg Filter in a Linear Array

The MPP Pascal code used to implement the described sequences is:

```
for n:=1 to max do
begin
   e:= snake_shift(e);
   where (col_index = 127) do
      where (row_index = 128-n) do
         b:= 0.0;
   s1:= e*b;
   s2:= sqr(e)+sqr(b);
   sum1:= sum(s1,1,2);
   sum2:= sum(s2,1,2);
   c[n]:= -2.0*sum1/sum2;
   temp:= c[n]*b+e;
   b:= c[n]*e+b;
   e:= temp;
end;
```

The **where** statement is similar to the **if** statement except the latter causes conditional execution while the former causes conditional

assignment. The **sum** function is a reduction function used to compute the arithmetic sum of a given parallel array.

The procedure snake_shift is used to simulate the effect of shifting the forward prediction errors to the left by one in the linear array mapped onto the MPP. The MPP Pascal code for the main part of snake_shift is:

```
r2:= shift(x,0,1);
r1:= rotate(x,1,1);
where (col_index=127) do
   where (row_index<127) do
      r2:=r1;
snake_shift:=r2;
```

The **shift** function performs an end-off shift of the entire array memory with data being lost in the rows or columns along the perimeter in the direction toward which the data movement is being done. The **rotate** function causes the array memory to be logically wrapped around so that data being shifted off an array edge is moved into the opposite edge of the array. Figure 4.5 illustrates the procedure snake_shift(x).

The parallel Burg algorithm was implemented on the MPP as described above. The number of data points was 16,384 and the number of reflection coefficients to be calculated was varied from one to 100. The results are shown in Table 4.1. Notice the linear relationship illustrated here.

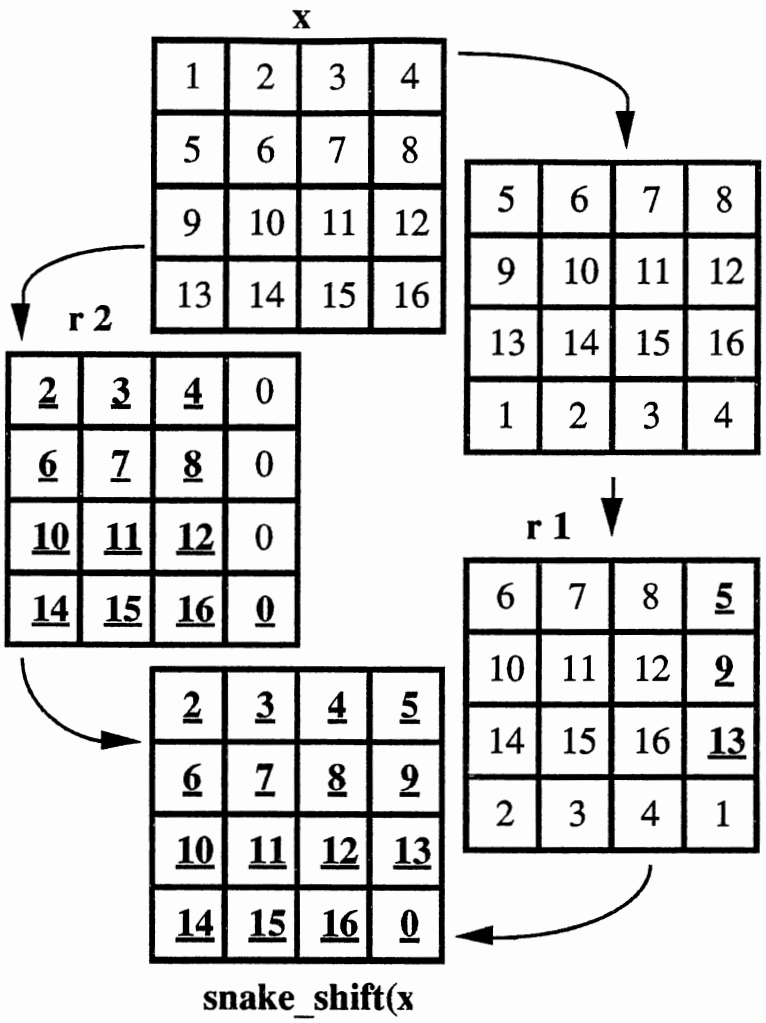


Figure 4.5 Procedure Snake-shift (x)

Table 4.1

Summary of Parallel Burg Algorithm on The MPP

MAX	1	2	5	10	40	80	100
TIME(msec)	5.5	11.1	27.6	55.2	220.8	441.6	552.1

Hypercube Implementation [49-51]

The Burg filter is a one-dimensional signal processing algorithm, while the hypercube is an n-dimensional architecture (n varies from 1 to 5 for the iPSC/2). The implementation problem reduces to finding a way to map the one-dimensional structure inherent in the Burg filter onto the n-dimensional architecture of the hypercube. Such a mapping is illustrated in Figure 4.6. The arrows represent the required connections, or communication channels, needed to view the hypercube architecture as a linear array of processors.

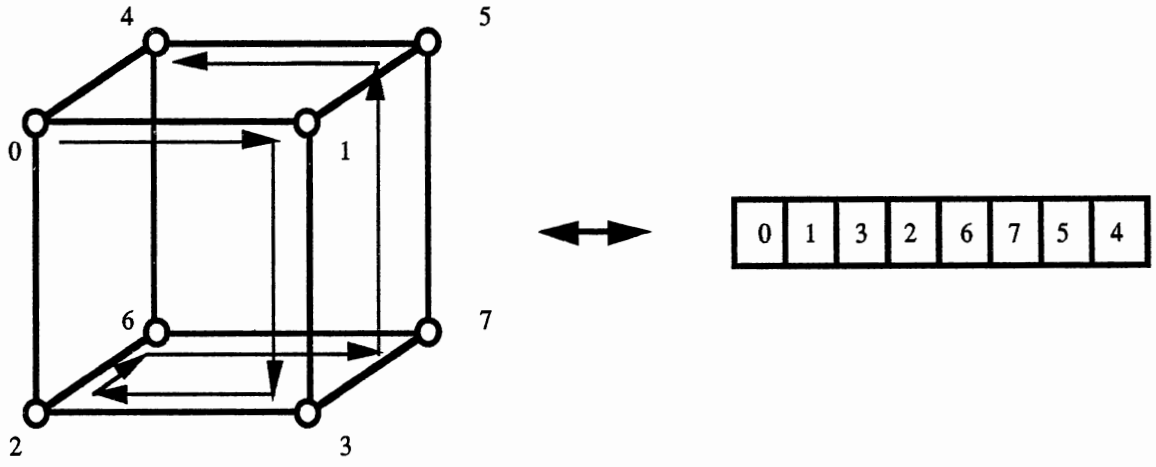


Figure 4.6. Mapping a Linear Array on a 3-D Hypercube

Figure 4.7 illustrates the data movement for the Burg filter if a linear array topology is mapped on a 2-D hypercube and the number

of data points is M (assume M is divisable by 4). In stage 0 the linear array is loaded with both the forward and backward prediction errors, which are equal to the observed time series. To calculate the first reflection coefficient the forward prediction errors are shifted to the left by one as shown in stage 1. Each node will calculate two partial sums: the first is that of the products of each pair of the forward and backward prediction errors and the second is that of their squares. Node zero receives the partial sums from the rest of the nodes (i.e. up-loading is performed) and produces the final sums needed to calculate the reflection coefficient given by equation(2-20). Each node receives the calculated reflection coefficient from node zero (i.e. down-loading is performed) and updates the forward and backward prediction errors. To calculate the second reflection coefficient the updated forward prediction errors are shifted to the left by one, as shown in stage 2. The above sequence of operations will be repeated until all reflection coefficients are computed.

Figure 4.7 also shows how the static load balancing is performed. All the nodes except the last one in the linear array will always perform the same number of computations every iteration. The number of computations performed by the last node in the linear array (node 2 in Figure 4.7) will be decremented every iteration. In real life applications the number of reflection coefficients (MAX) is much less than the length of the data sequence (M) so the last node will still be performing approximately the same number of computations as the rest of the nodes.

STAGE 0

Node 0	Node 1	Node 3	Node 2
e(1) ..e(M/4)	e(M/4+1) ...e(M/2)	e(M/2+1) .. e(3M/4)	e(3M/4+1)...e(M)
b(1) ...b(M/4)	b(M/4+1) ...b(M/2)	b(M/2+1) ...b(3M/4)	b(3M/4+1). .b(M)

STAGE 1

Node 0	Node 1	Node 3	Node 2
e(2) ...e(M/4+1)	e(M/4+2) ...e(M/2+1)	e(M/2+2) ...e(3M/4+1)	e(3M/4+2)...e(M) , 0
b(1) ..b(M/4)	b(M/4+1) ...b(M/2)	b(M/2+1) ...b(3M/4)	b(3M/4+1)...b(M-1), 0

STAGE 2

Node 0	Node 1	Node 3	Node 2
e(3) ...e(M/4+2)	e(M/4+3) ...e(M/2+2)	e(M/2+3) ...e(3M/4+2)	e(3M/4+3)...e(M) ,0,0
b(1) ...b(M/4)	b(M/4+1) ...b(M/2)	b(M/2+1) ...b(3M/4)	b(3M/4+1)...b(M-2),0,0

Figure 4.7 Data Movement for the Burg Filter in a Linear Array

Examining the maximally parallel graph it is noted that the part of the algorithm that calculates the reflection coefficient at each iteration is serial and creates a bottleneck that limits the algorithm performance. Another limiting factor is the four byte message between the nodes which has the worst communication overhead.

Figure 4.8a illustrates the speedup achieved by the algorithm on the hypercube (iPSC/2 with scalar processors). Linear speedup is only observed for low numbers of nodes and large numbers of data points. Speedup reaches a maximum then decreases, or stays at the same value.

Simply increasing the number of nodes will not result in a higher speedup due to communication overhead and bottleneck time. Figure 4.8b shows the speedup curves after subtracting the reflection

coefficient calculation time and the data shifting time. Notice the linear nature of the resulting modified speedup curves.

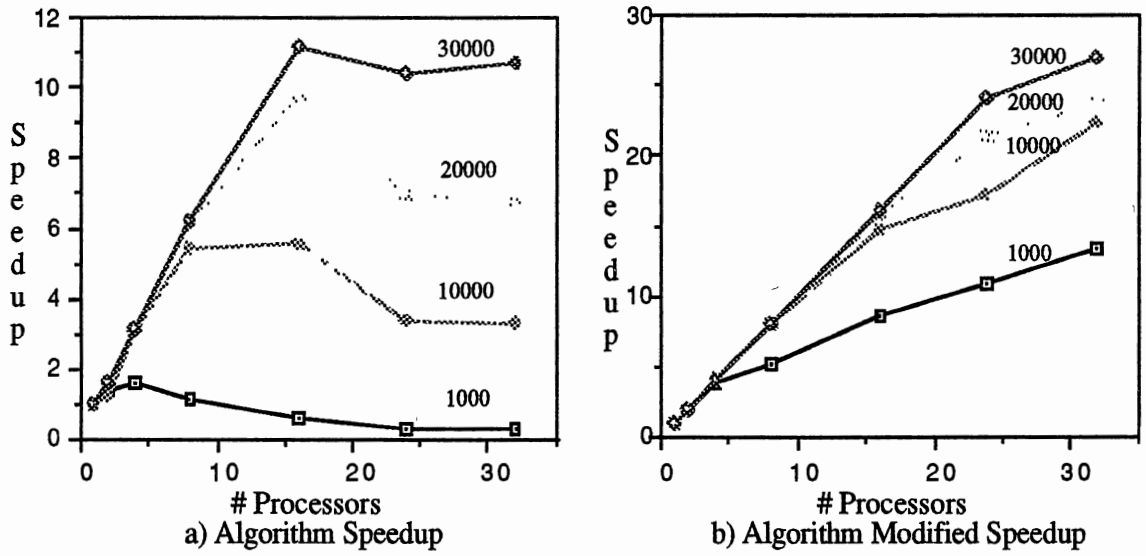


Figure 4.8. Speedup on The Hypercube (number of data points used are indicated on the plots)

Cray X-MP/48 Implementation [49,51]

The Cray X-MP/48 is used here without multitasking, only the vector processing capability is used. The compiler detects parallelism in the sequential program and converts the code appropriately. The major source of parallel code is found in DO-loops. The compiler analyzes the DO-loops found in the program and vectorizes them when possible. To be vectorizable, a DO-loop must manipulate or

perform computations on the contents of one or more arrays. Loops containing a GO TO, CALL, an I/O statment, or some form of IF statement are not vectorizable. In the case of the Burg algorithm all DO-loops are vectorizable.

Work loads on the Cray X-MP/48 are characterized by three types of execution requirements; scalar mode, vector mode, and concurrent mode. In the Burg implementation the vector mode is mainly used. In this mode the process code is executed in the vector section of the processor (process granularity is small).

The Cray X-MP/48 took 0.016887 sec to perform the Burg algorithm on 16,384 data points and 10 reflection coefficients.

HEP Implementation [30,49,51]]

The Burg algorithm was implemented on the HEP so that if NPROC processors are available, then those tasks which can be performed in parallel are performed NPROC at a time. The parallel implementation is:

```
PURGE $K,$DONE1,$DONE2
S1=0.0
S2=0.0
$K=NPROC
IF (NPROC.NE.1) THEN
    DO J=1,NPROC-1
        JJ(J)=J
        CREATE EB(JJ(J))
    ENDDO
CALL EB(NPROC)
DUMMY=$DONE1
C(N)=-2.0*S1/S2
    •
    •
    •
    •
    •
```

```
SUBROUTINE EB(J)
COMMON/EC/E(5000),B(5000),A(50),C(50),N,M,NPROC,$K,$DONE1,$DONE2
COMMON/EB1/S1,S2
SUM1=0.0
SUM2=0.0
DO I=N+J,M,NPROC
    SUM1=SUM1+E(I)*B(I-N)
    SUM2=SUM2+E(I)*E(I)+B(I-N)*B(I-N)
ENDDO
K1=$K-1
S1=S1+SUM1
S2=S2+SUM2
IF (K1.EQ.0) $DONE1=.TRUE.
$K=K1
RETURN
END
```

Each CREATE statement generates a new parallel process. The tasks are shuffled evenly between the processes.

Notice that the asynchronous variables $K and $DONE1 are used for synchroniztion on the HEP. Initially $K is set to NPROC and $DONE1 is purged so that it cannot be read. $K is decremented each time a process is completed. When $K is equal to zero (all processes have completed) the variable $DONE1 is set to .TRUE., thereby setting its state to full so that it can be read. Meanwhile, in the main program, the line - DUMMY=$DONE1 - acts as a barrier so that all processes will be completed before the program proceeds.

The algorithm was implemented on the HEP while varying the number of processors from one to twelve, and the number of data points from 1024 to 4096. Figure 4.9 illustrates the speedup obtained. It shows that this implementation of the Burg algorithm is highly parallel and that speedup increases linearly up to 6 or 7 processes. When more than 8 processes are used the speedup levels off, as the pipeline becomes full. The speedup of the algorithm increases somewhat as the number of data points increases. Each

process has more computations to perform, and a correspondingly smaller percentage of time would be spent on synchronization.

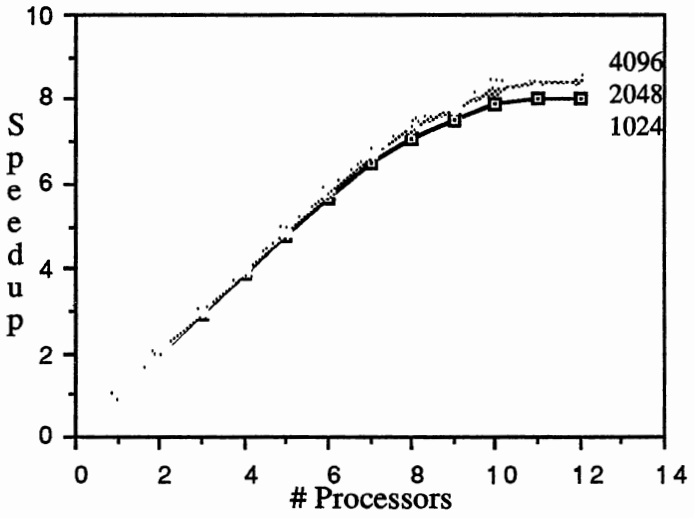


Figure 4.9 Algorithm Speedup on the HEP (number of data points used are indicated on the plots)

Alliant FX/8 Implementation [51]

The FX/8 is a tightly coupled MIMD machine with bus architecture. The results of implementing the Burg on the FX/8 should be used for comparison with those obtained on the HEP since the major difference in the two architectures is bus versus packet switched network.

The FX/8, through the use of FX/Fortran, allows the user to select between three forms of parallelism and other performance

enhancements: concurrency, vectorization, and scalar optimizations. Concurrency refers to the concurrent execution of loops and array operations by more than one processor. Vectorization refers to vector rather than scalar aggregates for processing data in loops and array constructs. And scalar optimizations refer to optimizations at the scalar level, such as redundant expression elimination and invariant code motion.

Program optimization typically increases the execution speed of a loop or array operation by a factor approaching the number of computational elements for concurrency and two to four for vectorization. The total increase in speed can be over 30 times the speed of the operation in scalar mode. Notice that the programmer has to use all the computational elements if concurrency is selected.

The Burg algorithm was implemented on the FX/8 while selecting different forms of parallelism. The results of the implementation are shown in Table 4.2 and Table 4.3. The results shown in Table 4.2 are those obtained when the Burg program was written using no FX/Fortran constructs, while the results shown in Table 4.3 are those obtained when using such constructs.

Table 4.2

Burg Implementation on the FX/8
(no FX/Fortran Constructs)

Optimization Type	Execution Time
None	4.064 sec
Global	1.369 sec
Global and vector	1.129 sec
Global and concurrency	0.465 sec
Global, vector, & concurrency	0.543 sec
With associative transformations	0.071 sec

Table 4.3

Burg Implementation on the FX/8
(FX/Fortran Constructs)

Optimization Type	Execution Time
None	0.835 sec
Global	0.573 sec
Global and vector	0.554 sec
Global and concurrency	0.422 sec
Global, vector, & concurrency	0.354 sec
With associative transformations	0.071 sec

Connection Machine Model CM-2 Implementation

The CM-2 is a massively parallel computer system that employs a 12-dimensional hypercube architecture. The efficient parallel implementation of the Burg algorithm requires the availability of an equal number of processors to that of the number of data points used. This concept was illustrated earlier using the maximally parallel graph. Thus, the CM-2 is a suitable machine for implementing the Burg algorithm and is expected to be very efficient.

The CM-2 used has 32K processors divided among four sequencers. Since the number of data points used in this study is 16K, we only used two sequencers. The programming language used is Fortran 90 which offers a rich selection of operations and intrinsic functions for manipulating arrays. An array can be referenced by name in an expression or assignment or passed as an argument to any Fortran intrinsic function, and the operation is performed on every element of the array.

The main loop of the Burg algorithm is given by:

```
DO N=1, MAX
    E=E(2:M-N+1)
    B=B(1:M-N)
    C(N)= -2.0 * SUM(E*B)/SUM(E**2+B**2)
    TEMP= E+C(N)*B
    B=B+C(N)*E
    E= TEMP
ENDDO
```

Notice that Fortran 90 supports selecting a particular section of an array which is used in the above code to actually replace do-loops.

The resulting code is efficient and very clear. The intrinsic function SUM performs the summation of all the elements of the array specified in the function argument.

The CM-2 Fortran program is actually directing two CM-2 system components with different memory organizations. An array can have its home either in the centralized memory of the front end or in the distributed memory of the CM-2. The CM-2 Fortran compiler allocates arrays on one machine or the other depending on how they are used. Arrays that are used only in Fortran 77 constructions in a program unit, and all scalar data, reside on the front end. Arrays that are used in array operations anywhere in a program unit reside on the CM-2. Programmers should avoid using an array both as an array object and as a subscripted array. Such an array has a CM-2 home, but the system moves it to the front end, one element at a time, to perform the serial operation. This data transfer is very expensive.

The E and B arrays have their home on the CM-2 distributed memory while the C array resides on the front end. Notice that the SUM intrinsic function returns a scalar, hence when calculating the reflection coefficients the two scalar numbers resulting from invoking the SUM function are passed to the front end where they are used to calculate the C array.

When using two sequencers for a total of 16,384 processors, the CM-2 completed the Burg algorithm successfully in 0.00763257 sec. The number of data points is 16,384 and the order of the filter is ten. When using one sequencer for a total of 8192 processors, the CM-2 completed the same Burg problem in 0.0101474 sec. Notice that in

the former case where two sequencers were used the CM-2 did not create any virtual processors, while in the latter case 8192 virtual processors were created. When comparing the two results it is obvious that the implementation of the virtual processors on the CM-2 is very fast and efficient.

The Cray-2 Implementation

Similar to the Cray X-MP/48, the Cray-2 is used here without multitasking but taking advantage of the powerful and improved vector processing capability. The discussion given in section 4.2.3 for the Cray X-MP/48 is also valid here for the Cray-2.

The Cray-2 took 0.00755184 sec to perform the Burg algorithm on 16,384 data points and 10 reflection coefficients.

Preliminary Comparison

A number of conclusions can be drawn from the results of this study. This section discusses some general conclusions about the relative performance of the seven machines used in this research. The next chapter, chapter V, builds on the preliminary comparison performed here and addresses more detailed aspects of the relative performance of the seven machines.

Table 4.4 is a comparison of algorithm execution times for the four machines, with 16,384 data points and 10 reflection coefficients.

Table 4.4

Comparison of Burg Execution Time

Machine	Execution Time (sec)
Denelcor HEP	1.679
Intel iPSC/2	0.24 (16 nodes)
Alliant FX/8	0.07084
NASA/Goodyear MPP	0.05522
Cray X-MP/48	0.016887
CM-2	0.00763257
Cray-2	0.00755184

Based on the mapping techniques used to implement the Burg algorithm on the seven machine and the underlying architectures, we introduce an implementation classification that will facilitate the discussion of the relative performance of the machines. Class one machines are the " true " MIMD computers and include the HEP, the iPSC/2, and the FX/8. Class two machines are the MIMD computers that were used for their vector processing capabilities and include the Cray X-MP/48 and the Cray-2 supercomputers. Finally, class three machines are the massively parallel SIMD computers and include the MPP and CM-2.

In class one machines, the number of processors is limited: 16 in the HEP, 32 in the iPSC/2, and 8 in the FX/8. The data set used contains many more data points than processors, so it was necessary to divide the data set equally among the processors. In doing so,

each processor must sequentially repeat the same calculations for the data points assigned to it.

Studying the results of mapping the Burg algorithm on class one machines, the FX/8 gave the fastest execution time followed by the iPSC/2 and finally the HEP. The FX/8 has a tightly coupled bus architecture that seems to work well for the Burg algorithm. The iPSC/2 is usually efficient when the interaction between the nodes is kept minimal, while the HEP can tolerate a higher degree of interactions between tasks without significant deterioration in performance. One of the limitations of the HEP is the switch network speed. Communication overhead imposed by the message passing communication scheme is a major disadvantage of the hypercube architecture. In the case of the HEP, communication is not a problem, but memory or bus contention is.

Figure 4.10 compares speedups achieved by the iPSC/2 and the HEP. The HEP shows much better speedup - almost linear. For both machines the speedup goes up as the number of data points increases. Since more data is loaded for computation on the processors, the processors spend more time computing partial sums than on communication or synchronization.

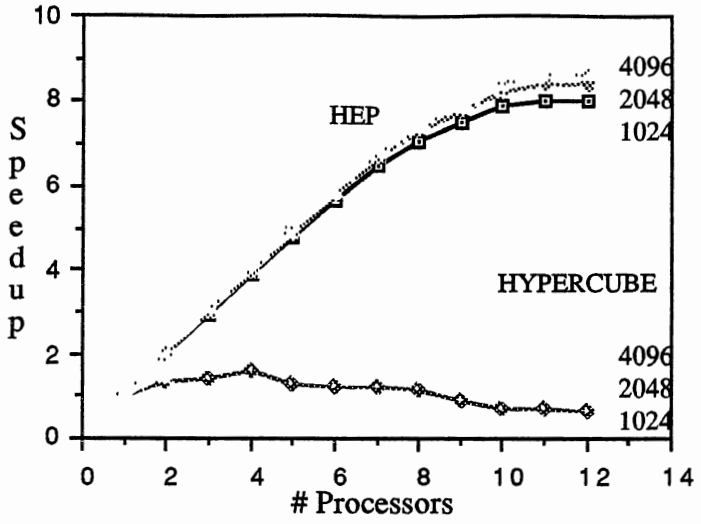


Figure 4.10 Comparison of HEP and iPSC/2 Speedups (number of data points used are indicated on the plots)

Class two machines are the Cray supercomputers: the Cray X-MP/48 and the Cray-2. These machines were used for their vector processing capabilities. The results show very fast execution times and indicate good utilization of the vector processing capabilities on the Cray computers. The performance of the Cray machines is limited by the vector size, namely 64. The main difference between the two Cray supercomputers used is the individual processor potential (see Table 3.3). The ratio of the processor potential of the Cray-2 to that of the X-MP/48 is about 2.08, i.e., the processor used in the Cray-2 has about double the capabilities or power of the processor used in the X-MP/48. Table 4.4 supports this ratio and in fact results in a higher number, namely, 2.17 .

Class three machines are the MPP and CM-2. Both machines are massively parallel and the results show fast execution times. Earlier in this chapter, the maximally parallel graph, given in Figure 4.2, suggested that we would need to have M processors (where M is the number of data points) to take the full advantage of the parallel nature of the Burg algorithm. When the Burg algorithm was implemented on class three machines, one data point was assigned to one processor so that maximum parallelism was possible. In fact, class three machines, due to their massive number of processors and flexibility, are the only machines that can implement the Burg algorithm with maximum parallelism. Therefore, class three machines are clearly the best machines for implementing the Burg algorithm.

This section discussed some preliminary conclusions drawn from the results reported in this chapter. The next chapter examines the relative performance of all machines used and presents an overall ranking.

CHAPTER V

PERFORMANCE ANALYSIS

In the previous chapter we concluded by presenting a preliminary comparison of the implementation results. The discussion was meant to setup the stage for a more thorough analysis of the relative performance of the seven machines. In this chapter we present the complete performance analysis based on the results reported in chapter IV.

This chapter is divided into three major sections: the first contains the serial and parallel timing equations, the second presents a ranking system for the seven machines used, and the third contains a discussion of some of the guidelines to be used when designing future machines.

Timing Equations

When implementing any algorithm on a parallel machine it is important to predict the time it takes to run the algorithm. Such time can only be calculated if appropriate equations are developed that take into consideration the underlying architecture and the speed at which it performs different operations needed to implement the algorithm. The importance of developing the timing equations is that they can be used to compare the performance of all the

architectures theoretically. The predicted time can be used along with the actual time to judge the efficiency of the underlying computer system. When examining the timing operations we can identify the parts of the algorithm that run efficiently and thus discuss the suitability of implementing the algorithm on that particular architecture.

The time to perform the sequential Burg algorithm is given by equation (5-1). This equation is considered to be the baseline timing equation when comparing the performance of the different advanced parallel architectures studied in this research.

$$T_{serial} = \left[5M * MAX + \sum_{i=1}^{MAX} (1-5i) \right] t_{mul} + \left[4M * MAX + \sum_{i=1}^{MAX} (4i+2) \right] t_{add} + t_{div} \qquad (5-1)$$

To check the validity of this equation, the iPSC/2 was used while activating only one node. Two examples were run on the iPSC/2: the first with M=1024 and MAX=10 and the second with M=30000 and MAX=10.

For the first example, equation (5-1) indicates that the predicted serial time is given by 91656 t_{flop} where t_{flop} is the time to perform a floating point operation like multiplication, addition, or division. Knowing that t_{flop} for the iPSC/2 with the Waitek chip is approximately 1.4 μsec, equation (5-1) predicts the time to be 0.128 sec. The actual program running time was 0.125 sec which corresponds to 2% error.

For the second example, equation (5-1) indicates that the predicted serial time is 2699490 t_{flop} or 3.7793 seconds. The actual program running time was 3.783 sec corresponding to 0.1% error. The two examples clearly verify equation (5-1).

In the next part of this section we examine the parallel timing equations for the seven machines used in this research. We first examine class one machines, namely, the iPSC/2, the HEP and the Alliant FX/8. As previously mentioned in chapter IV these machines are " true" MIMD machines with limited number of processors.

We start by presenting the iPSC/2 timing equation given by:

$$T_{parallel} = \frac{T_{serial}}{p} + (12*p-4)*t_{init} + (2*p+2)*t_{flop} \qquad (5-2)$$

where p is the number of processors used, t_{init} is the time to initiate a message (time to send a message is negligible), and t_{flop} is the time to perform a floating point operation (add, subtract, multiply, or divide). The values of t_{init} and t_{flop} are 0.5 msec and 1.4 μsec respectively.

The parallel time in equation (5-2) consists of two parts: The first part , which is the first term, simply represents the linear speedup and the second part , the remaining terms, represent the deviation from this linear speedup. In other words, simply increasing the number of processors will not result in a higher speedup due to the communication overhead. Table 5.1 gives the deviation values in terms of t_{init} and t_{flop} in one column and in

terms of seconds in another column while varying the number of processors.

Table 5.1

iPSC/2 Deviation Term

p	Deviation Expression	Deviation (sec)
2	$20\ t_{init} + 6\ t_{flop}$	0.01
4	$44\ t_{init} + 10\ t_{flop}$	0.022
8	$92\ t_{init} + 18\ t_{flop}$	0.046
16	$188\ t_{init} + 34\ t_{flop}$	0.094
32	$380\ t_{init} + 66\ t_{flop}$	0.1901

Finally, Table 5.2 compares actual to predicted execution times for the case of M=30000 and MAX=10. The table illustrates the validity of equation (5-2).

Equation (5-3) gives the parallel timing equation for the HEP. The structure of this equation is similar to that of the iPSC/2 but in this case the overhead time or the deviation from the linear speedup is caused by different factors.

$$T_{par} = T_{serial} / p + O_{HEP}(p) \qquad (5\text{-}3)$$

Table 5.2

iPSC/2 Actual and Predicted Times

p	Actual (sec)	Predicted (sec)	|Error|
1	3.783	3.78	0.003
2	1.985	1.9	0.085
4	1.029	0.967	0.062
8	0.611	0.519	0.092
16	0.338	0.33	0.008
32	0.354	0.31	0.044

In equation (5-3), the variable $O_{HEP}(p)$ is the overhead time and is a function of the number of processors used. As the number of processors increases, more time is needed to establish a path between the processors and memory.

In the case of the iPSC/2, a loosely coupled architecture with local memory, the technique for sharing information is mainly done using message passing that requires setting up a channel for communication thus requiring communication overhead. In the case of the HEP, a tightly coupled architecture with shared memory, the overhead is a result of setting up the processes, using the asynchronous variables, and memory contention through the packet switch network.

Equation (5-4) shows the parallel timing equation for the Alliant FX/8, a tightly coupled architecture with shared memory, which is similar in structure to the equations for the iPSC/2 and the HEP.

$$T_{par} = T_{serial} / 8 + O_{FX/8} \qquad (5\text{-}4)$$

In equation (5-4), the variable $O_{FX/8}$ is the overhead time. The overhead represents the time to access the shared memory through the crossbar switch and the memory bus and the time to synchronize the processors using the concurrency bus. A shortcoming of the implementation on the FX/8 is the inflexibility of choosing the number of processors to be used. The compiler uses the eight available processors most of the time.

This concludes the discussion of the timing equations for class one machines. It was demonstrated that the equations are similar in structure and an increase in the number of processors does not result in improved speedup since there is an overhead in implementing the algorithm. The overhead is a function of the number of processors used. Class one machine are therefore considered to be fairly suitable for implementing the selected algorithm.

Shifting our attention to class two machines we now consider the MPP and CM-2. Examining the Pascal code implementation of the algorithm we could write the following parallel timing equation for the MPP:

$$T_{par} = MAX [\ 6t_{pmul} + 6t_{pequ} + 2t_{padd} + 2t_{sum} + 2t_{shift} + 2t_{rotate} + t_{mul} + t_{add}\ ] \qquad (5\text{-}5)$$

where t_{pmul} is the time to execute a parallel multiplication (76 μsec), t_{pequ} time to equate, t_{padd} time to perform a parallel addition (39 μsec), t_{sum} time to perform the reduction function **sum**, t_{shift} time to perform the array memory manipulation function **shift**, t_{rotate} time to perform the function **rotate**, t_{mul} time to perform a serial multiplication, and t_{add} time to perform a serial addition. T_{par} is not a function of the number of data points unless it exceeds 16384. Typical stationary time series length will not reach this upper bound. Notice that MAX is the only algorithm variable used in equation (5-5).

Examining Table 4.1 it is clear that a linear relationship exists and validates equation (5-5). It is obvious that the MPP implementation of the Burg filter is an efficient one due to its massive number of processors and its SIMD architecture.

Equation (5-6) gives the parallel timing equation for the CM-2. It is similar in structure to that of the MPP. All the parallel operations are done on the CM-2 while the scalar multiply and divide were performed on the front end computer.

$$T_{par} = MAX\ [\ 5t_{pmul} + 3t_{pequ} + 3t_{padd} + 2t_{sum} + t_{mul} + t_{div}\ ] \qquad (5\text{-}6)$$

In general, class two machines seem to be the most suitable machines for implementing the Burg algorithm. The flexibility of these machines and their massive number of processors enable them to use one processor per data point thus achieving maximum

parallelism taking full advantage of the parallel nature of the Burg algorithm.

We now consider the implementation on the Cray machines, namely, the Cray X-MP/48 and the Cray-2. These machines were classified as class three machines. The vector mode was mainly used on these machines where the process code was executed in the vector section of the processor (process granularity is small). Time to perform operations, in this mode, on vectors of length N (assume that N is a multiple of 64) is given by:

$$T = T_{start} + N \left(T_{startstrip} / 64 \quad + \quad T_{vcomp} \right) \qquad (5\text{-}7)$$

where T_{start} is the startup time for the vector operation, $T_{startstrip}$ is the startup time for stripmining the vectors (vectors of length greater than 64 are stripmined in sections of length 64), and T_{vcomp} is the single vector computation time per element. It can be stated that class three machine are not suitable for implementing the Burg filter since only the vector capabilities were used while the multiprocessing capabilities were not used for their high overhead cost.

Ranking of Machines

In chapter IV, Table 4.4 showed that the Cray-2 is the fastest machine to execute the Burg filter. Is the speed the only measure we should use to evaluate the machines? The answer is definitely no. In the previous section we argued that class two machines are the

most suitable machines to implement the Burg filter from an architectural point of view. Other important factors are the cost of each machine and the technology used in the design. To make the comparison fair, all the aforementioned factors must be considered.

It is important to consider the speed of execution since some applications have time limitations on the execution time. It is vital to consider the suitability of implementation since it is a measure of the efficiency of the underlying architecture and because general purpose machines were used. The technology is important since it can compensate for the reduction in execution speed. Finally, the cost is of major importance, whether buying the system or buying time.

In this section we take all these factors into consideration and try to come up with a ranking system for the different machines used. Table 5.3 gives some comparative figures that are used in creating the ranking system.

Table 5.3

Comparative Figures for the Computers Used

	X-MP	Cray-2	HEP	MPP	iPSC/2	FX/8	CM-2
Year	83	85	81	82	87	87	88
# Proc	4	4	16	16K	32	8	64K
Price	10M	15M	3M	3M	1M	1M	3M
Network	Fully	Fully	Switch	Mesh	h-cube	Bus	h-cube
Arch.	MIMD	MIMD	MIMD	SIMD	MIMD	MIMD	SIMD
Time	0.06887	0.00755	1.679	.05522	0.24	0.07084	0.00763

Table 5.4 shows the comparative figures for all the machines used with relation to technology, cost, suitability, and execution time. To come up with the figures, each factor is mapped between zero and one and each machine is assigned a number proportional to its relative location within this interval. For example, the year 88 is assigned a zero while year 81 is assigned a one. In this way old technologies are compensated for their age. Suitability figures were assigned based on the discussion in the previous section: class two machines are the best followed by class one and finally class three machines.

Table 5.4

Comparative Figures in Terms of the Four Factors

	X-MP	Cray-2	HEP	MPP	iPSC/2	FX/8	CM-2
Tech.	0.714	0.429	1	0.857	0.143	0.143	0
Cost	0.357	0	0.857	0.857	1	1	0.857
Suit.	0	0	0.5	1	0.5	0.5	1
Time	0.994	1	0	0.971	0.861	0.962	0.99995

To setup the ranking system six different performance measures are used where each measure corresponds to a different weighting system of the four factors given in Table 5.4. Each measure has the following computational structure:

$$M_x = \alpha_1 \text{ Tech.} + \alpha_2 \text{ Cost} + \alpha_3 \text{ Suit.} + \alpha_4 \text{ Time} \qquad (5\text{-}8)$$

where α_1, α_2, α_3, and α_4 are weighting constants and their sum is one. The variable x represents the measure number and can be one through six to indicate the six measures used.

The first measure, M_1, is simply the execution time, where the values of α_1 thru α_3 are zero and the value of α_4 is one. This is to emphasize the importance of the execution time.

The second measure, M_2, weighs the first three factors equivalently and weighs the execution time twice as much. This measure corresponds to the following assignment of the weighting constants: $\alpha_1=\alpha_2=\alpha_3=0.2$ and $\alpha_4=0.4$.

The third measure, M_3, weighs all the four factors equivalently. All of the weights are simply assigned to 0.25.

The fourth measure, M_4, is similar to M_3 except that the suitability factor is neglected. The measure in this case corresponds to the following assignment: $\alpha_1=\alpha_2=\alpha_4=0.3$ and $\alpha_3=0$.

The fifth measure, M_5, weighs cost and execution time twice as much as the technology and suitability. The constants are assigned as: $\alpha_1=\alpha_3=1/6$ and $\alpha_2=\alpha_4=1/3$.

The sixth and final measure, M_6, is similar to M_5 except that the suitability factor is neglected. This measure corresponds to using the following values: $\alpha_1=0.2$, $\alpha_2=\alpha_4=0.4$, and $\alpha_3=0$. Table 5.5 summarizes the six measures.

The results of these measures are tabulated in Table 5.6. The last entry in the table is simply the average of all the measures for each machine.

Table 5.5

The Six Measures

	α_1	α_2	α_3	α_4
M_1	0	0	0	1
M_2	1/5	1/5	1/5	2/5
M_3	1/4	1/4	1/4	1/4
M_4	1/3	1/3	0	1/3
M_5	1/6	1/3	1/6	1/3
M_6	1/5	2/5	0	2/5

Table 5.6

Results of Applying the Six Measures (All Machines)

	X-MP	Cray-2	HEP	MPP	iPSC/2	FX/8	CM-2
M_1	0.994	1	0	0.971	0.861	0.962	0.99995
M_2	0.6118	0.4858	0.4714	0.9312	0.673	0.7134	0.77138
M_3	0.51625	0.35725	0.58925	0.92125	0.626	0.65125	0.71424
M_4	0.68833	0.47633	0.619	0.895	0.668	0.70167	0.61898
M_5	0.56933	0.40483	0.53567	0.91883	0.7275	0.76117	0.78565
M_6	0.6832	0.4858	0.5428	0.9026	0.773	0.8134	0.74278
Avg.	0.67715	0.535	0.45969	0.92331	0.72142	0.76715	0.77216

The measures used were selected for their importance and that does not imply that other measures can not be used. We found these measures fair and representative of the comparative performance of the seven machines used. Table 5.7 shows the ranking of all the machines and is directly created from Table 5.6.

Table 5.7

The Ranking Results (All Machines)

	1	2	3	4	5	6	7
M_1	Cray-2	CM-2	X-MP	MPP	FX/8	iPSC/2	HEP
M_2	MPP	CM-2	FX/8	iPSC/2	X-MP	Cray-2	HEP
M_3	MPP	CM-2	FX/8	iPSC/2	HEP	X-MP	Cray-2
M_4	MPP	FX/8	X-MP	iPSC/2	HEP	CM-2	Cray-2
M_5	MPP	CM-2	FX/8	iPSC/2	X-MP	HEP	Cray-2
M_6	MPP	FX/8	iPSC/2	CM-2	X-MP	HEP	Cray-2
Avg.	MPP	CM-2	FX/8	iPSC/2	X-MP	Cray-2	HEP

Clearly the MPP and the CM-2 are ranked first and second respectively almost consistently. This enforces the earlier conclusion about class two machines that indicates their efficiency in implementing the algorithm due to their architecture that incorporates massive number of processors.

Examining Table 5.6 one can argue that the HEP, with its old technology and its slow execution time, represents an outlier that affects the comparison. Tables 5.8 through 5.10 are similar to Tables 5.5 through 5.7 respectively except that the HEP is not included in the comparison.

Table 5.8

Comparative Figures (without HEP)

	X-MP	Cray-2	MPP	iPSC/2	FX/8	CM-2
Tech.	0.833	0.5	1	0.167	0.167	0
Cost	0.357	0	0.857	1	1	0.857
Suit.	0	0	1	0.5	0.5	1
Time	0.9598	1	0.7949	0	0.7277	0.99995

Table 5.9

Results of the Six Measures (without HEP)

	X-MP	Cray-2	MPP	iPSC/2	FX/8	CM-2
M_1	0.9598	1	0.7949	0	0.7277	0.99995
M_2	0.62192	0.5	0.88936	0.3334	0.62448	0.77138
M_3	0.53745	0.375	0.91298	0.41675	0.59868	0.71424
M_4	0.7166	0.5	0.88397	0.389	0.63157	0.61898
M_5	0.5777	0.41667	0.88397	0.4445	0.68707	0.78565
M_6	0.69332	0.5	0.86076	0.4334	0.72448	0.74278
Avg.	0.68448	0.54861	0.87099	0.33618	0.66556	0.77216

Notice that the results of performing the complete analysis without considering the HEP are similar to those done earlier where all the machines were considered. Class two machines are again the top performers. In fact, the CM-2 numbers improved compared to the FX/8, i.e., the gap between the CM-2 and the FX/8 was amplified.

Table 5.10

The Ranking Results (without HEP)

	1	2	3	4	5	6
M_1	Cray-2	CM-2	X-MP	MPP	FX/8	iPSC/2
M_2	MPP	CM-2	FX/8	X-MP	Cray-2	iPSC/2
M_3	MPP	CM-2	FX/8	X-Mp	iPSC/2	Cray-2
M_4	MPP	X-MP	FX/8	CM-2	Cray-2	iPSC/2
M_5	MPP	CM-2	FX/8	X-MP	iPSC/2	Cray-2
M_6	MPP	CM-2	FX/8	X-MP	Cray-2	iPSC/2
Avg.	MPP	CM-2	X-MP	FX/8	Cray-2	iPSC/2

We believe that the final ranking given by Table 5.7 is a reasonable ranking and should be adopted. It is consistent with the preliminary analysis performed in chapter IV and the analysis performed in the previous section. The MPP is clearly the best machine to implement the Burg algorithm followed by the CM-2. In

general, small-grain massively parallel SIMD architectures are the most suitable machines for implementing the selected algorithm.

Future Machines

In this section, we discuss some of the guidelines that should be used when designing future machines. The guidelines are based mostly on the experience acquired throughout this research and directed towards solutions that will improve the mapping of the class of signal processing algorithms discussed in this research.

Parallel processing is the key to high performance in modern advanced computer architectures. All architectures used in this research employ parallel processing in several different ways and proved to be fast and efficient. Future machines must continue to use parallel processing techniques to achieve higher speed and improved performance.

Newer and faster technologies must be developed, investigated, and used in future machines to improve the overall computational efficiency of parallel processing machines. The concept of wafer scale integration (WSI) appears to have a great potential in designing regularly structured computers that can implement the class of signal processing algorithms addressed in this work. Lower submicron technologies that use shorter interconnects should be investigated for their potential in increasing the speed and decreasing the overall computer size.

Examining the overall performance results presented earlier in this chapter, it is clear that small-grain massively parallel SIMD

architectures are the most suitable for the selected algorithm. Although a linear architecture is most suitable for the selected algorithm, a hypercube architecture appears to be a better choice for its ability of matching the different algorithms within the selected class. The concept of the data parallel model is a relatively simple and natural one. When compared to an MIMD architecture, SIMD machines perform synchronization implicitly and do not suffer from the memory contention problem. This does not imply that current massively parallel SIMD machines are perfect. In fact, there are several obstacles and problems that must be overcome and solved before utilizing all the capabilities of the SIMD architecture model.

Based on the work done in this research, we list four problems with current SIMD architectures: first, the inflexibility in memory addressing; second, the inefficiency in numerical operations since reduced instruction set computer (RISC) technology is used; third, the slowness in routing data from one processor to another; and finally, the presence of a bottleneck in input and output.

Considering the first problem, current SIMD computers require that each processor access its own local memory at the same address as all the other processors. Table lookup and indirect addressing operations are usually employed in current systems to solve the problem but these operations are currently very slow and inefficient. By providing indexed and indirect addressing in future SIMD computers, it becomes possible to implement efficient table lookup operations that will result in overall improved efficiency.

Considering the second problem, almost all of the current SIMD computers use processors that can add only 1-bit numbers in a single

machine cycle. This indicates that multibit arithmetic must be performed in bit-serial fashion, resulting in a particularly long computing times for floating-point operations. To overcome this problem, future SIMD computers must use more powerful processors that might employ more than 1-bit operations.

Discussing the third problem, general routing in some current SIMD machines must be performed through combinations of near-neighbor moves. The processors in these systems must perform the same routing direction. In essence, if one processor is getting data from its north neighbor the other processors must do the same thing, that is, getting data from their north neighbors. A solution might be found in making the memory and the routing control more tightly integrated and probably running at their own clock speed. A dual ported memory can be used so that accesses by the router would proceed independently from those by the processors.

The fourth problem is concerned with the input and output to the specialized architecture that form a main bottleneck in usage of current architectures. Logical and arithmetic operations might require nanoseconds to be performed while transferring arrays from the front end to the specialized architecture can easily require milliseconds, that is, one million times as long. A solution to the problem might involve the use of a bimodal memory system. In such a system, data can be accessed on either of two ports that employ different data formatting. The memory system performs the task of implicit formatting. The processor array can access the memory from one port where it is utilized as part of the address space for the local memories of the processor elements. In the same

fashion, the front end computer can access the same memory from the other port where it utilizes a multibit layer of the bimodal memory as if it were part of the address space.

Aside from the aforementioned problems and obstacles that face current SIMD architectures, research must continue in exploring the possibilities of using optical and/or neural technologies to improve the overall performance. Both technologies are reported to have faster switching speed and emphasize massive parallelism and fine granularity.

Research must continue in the area of software development, especially that of designing compilers for such complex systems. Virtualizing compilers can be of great importance when used in such systems. Such compilers are able to automatically map parallel data structures onto processor arrays which would allow the development of machine independent programs neglecting the details of the host architecture.

CHAPTER VI

SUMMARY AND CONCLUSION

The field of digital signal processing has a wide variety of applications that has served to create a vitality that is often missing in other scientific fields of study. With the invention of digital computers and, more recently parallel architectures, the field of digital signal processing has become an increasingly significant field.

A problem of great importance has emerged: what is the best way to map a digital signal processing algorithm onto a given parallel architecture? The answer is by no means simple since there are no established principles to govern the mapping techniques. It was the ultimate objective of this research to establish some of these principles.

The main objective of this work was to explore the different techniques of mapping digital signal processing algorithms onto parallel computer architectures. It was impossible to cover all algorithms and all architectures. The algorithms were limited to those which can be characterized as one dimensional, batch, and time domain. As for the architectures, the availability of such systems was the major limitation. The goal of this research was to discover the types of computer architectures that are the best suited for digital signal processing.

In chapter I, the problem of inverse filtering was presented and selected for further investigation in this work. The inverse filtering problem was selected because it is of great importance in the field of digital signal processing and is used in many other fields such as control theory, parametric spectrum analysis, estimation theory, seismic signal processing, and speech processing. The basic idea behind inverse filtering is to determine the parameters of a model given observations of the physical process being modeled. The time series model selected was the autoregressive process. Chapter I concluded with a setup of the so called Yule-Walker or normal equation which is considered to be the basis for developing the solution of the inverse filtering problem.

In chapter II, batch inverse filtering algorithms were presented to identify their similarities and differences. Finally, one algorithm was chosen which is representative of this class of algorithms. The algorithms have a common computational structure, namely, a time shift / inner product operation. In fact, this operation is a key step in performing all digital signal processing algorithms which involve convolution or correlation. The Burg algorithm was selected for implementation on advanced parallel computer architectures.

Most previous studies of parallel signal processing have been concerned with the design of special purpose hardware for real-time signal processing using recursive algorithms. In contrast, the goal of this work was the efficient implementation of batch signal processing algorithms on general purpose parallel machines. In many signal processing applications, it is not necessary to process the data in real-time, and it is clear that even with new special purpose signal

processing chips, there will still be a need to perform signal processing on large, general purpose, main frame computers.

Chapter III discussed in detail the seven advanced computer architectures used in this research: the Denelcor HEP, a shared memory (tightly coupled) multiple-instruction stream multiple-data stream (MIMD) machine with switch network interconnect architecture; the Cray X-MP/48, a shared memory (tightly coupled) MIMD supercomputer with direct connect interconnect architecture; the Intel iPSC/2 hypercube computer, a distributed memory (loosely coupled) MIMD machine; the Alliant FX/8, a shared memory (tightly coupled) MIMD machine with a bus interconnect architecture; the NASA/Goodyear MPP, a massively parallel SIMD machine with mesh interconnect architecture; the Connection Machine model CM-2, a massively parallel SIMD machine with hypercube interconnect architecture; and the Cray-2 supercomputer, a tightly coupled MIMD machine with direct connect interconnect architecture and is the latest Cray to be produced.

It was important to show that the architectures used in this research represent a good portion of the available parallel machines. This was accomplished by using two computer classification schemes. Flynn's computer classification and a new classification scheme were presented. The new scheme is based on three essential issues: the granularity of the processing elements; the topology of the interconnections between the processing elements; and the distribution of control across the processing elements. The so called organizational space of parallel computer systems was presented with these variables as the axes. An attempt was made to place each

computer system presented in chapter II in its approximate position within the space. The results of the classification show that the architectures occupy a substantial volume of the three dimensional space making them good representatives of the available computer systems.

In chapter IV we described the parallel implementation of the Burg algorithm on the seven architectures and presented a preliminary analysis of the results. To take full advantage of the parallel nature of the Burg algorithm, the maximally parallel graph showed that we would need to have the number of processors equal to the number data points. An implementation classification was presented to facilitate the discussion of the relative performance of the machines. It was shown that class three machines, namely, the MPP and the CM-2, are clearly the best machines for implementing the Burg algorithm. Class three machines are characterized by their massive number of processors, which takes full advantage of the parallel nature of the Burg algorithm.

Chapter V examined the relative performance of all machines used and presented an overall ranking. The ranking system was based on four essential issues: the technology used, the cost of the system, the mapping suitability, and finally the execution time. The results of the ranking enforce the aforementioned preliminary analysis, namely, the MPP and the CM-2 are the best machines for implementing the Burg algorithm. Small-grain massively parallel SIMD architectures are the most suitable for the selected algorithm. Guidelines for designing future machines were included and are based mostly on the experience acquired throughout this research

and directed towards solutions that will improve the mapping of the selected signal processing algorithms.

This research addressed an important problem that is encountered when using advanced computer architectures to implement signal processing algorithms, namely, the mapping problem. The solutions presented in this manuscript can be generalized to the class of algorithms selected and will help in solving algorithms of similar computational structure. As more architectures become available, the need for such research grows so as to cover these newly developed parallel machines and establish guidelines to efficiently map signal processing algorithms.

REFERENCES

[1] A. Macovski, Medical Imaging Systems, Prentice-Hall, Englewood Cliffs, N.J., 1983.

[2] B. D. Steinberg, Principles of Aperture and Array System Design, John Wiley and Sons, N.Y., 1976.

[3] L. Rabiner and R. Schafer, Digital Processing of Speech Signals, Prentice-Hall, 1978.

[4] K. Feher, Advanced Digital Communications, Prentice-Hall, 1987.

[5] E. Robinson and S. Treitel, Geophysical Signal Analysis, Prentice-Hall, 1980.

[6] L. Ljung, System Identification: Theory for the User, Prentice-Hall, 1987.

[7] G. Goodwin and K. Sin, Adaptive Filtering Prediction and Control, Prentice-Hall, 1984.

[8] S. Kay, Modern Spectral Estimation: Theory and Practice, Prentice-Hall, 1988.

[9] L. Marple, Digital Spectral Analysis with Applications, Prentice-Hall, 1987.

[10] A. Giordano and F. Hsu, Least Square Estimation with Applications to Digital Signal Processing, John-Wiley and Sons, New York, 1985.

[11] S. Y. Kung, H. J. Whitehouse and T. Kailath (Eds.), VLSI and Modern Signal Processing, Prentice-HAll, 1985.

[12] B. A. Bowen and W. R. Brown, VLSI System Design for Digital Signal Processing, Prentice-Hall, 1982.

[13] P. Denyer and D. Renshaw, VLSI Signal Processing: A Bit Serial Approach, Addison-Wesley Publishing Company, Reading, MA, 1985.

[14] U. Schendel, Introduction to Numerical Methods for Parallel Computers, Ellis Horwood Limited, London, 1984.

[15] D. Heller, " A Survey of Parallel Algorithms in Numerical Linear Algebra," SIAM Review, No. 4, p.740, October 1978.

[16] J. Jover and T. Kailath, " A Parallel Architecture for Kalman Filter Measurement Update and Parameter Estimation," Automatica, Vol. 22, No. 1, p.43, 1986.

[17] H. M. Ahmed, " Signal Processing Algorithms and Architectures," Ph.D. Dissertation, Stanford University, June 1982.

[18] H. M. Ahmed, J. M. Delsome and M. Morf, " Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," Computer, Vol. 15, No.1, pp 65-80, January 1982.

[19] K. Hwang and F. A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill, New York, NY, 1984.

[20] K. Hwang and D. Degroot (Eds.), Parallel Processing for Supercomputers and Artificial Intelligence, McGraw-Hill, New York, NY, 1989.

[21] G. Desrochers, Principles of Parallel and Multiprocessing, McGraw-Hill, New York, NY, 1987.

[22] K. Hwang, "Advanced Parallel Processing with Supercomputer Architectures," Proceedings of the IEEE, Vol. 75, No. 10, p1348, October 1987.

[23] J. Mendel, Lessons in Digital Estimation Theory, Prentice-Hall, 1987.

[24] G. E. Box and G. M. Jenkins, Time Series Analysis Forecasting and Control, Holden Day, San Francisco, CA, 1967.

[25] O. D. Anderson, Time Series Analysis and Forecasting: The Box-Jenkins Approach, Butteworth and Co. , London, 1975.

[26] J. Makhoul, " Linear Prediction: A Tutorial Review," Proc. IEE, Vol 63, pp. 561-580, April 1975.

[27] T. Kailath, " A View of Three Decades of Linear Filtering Theory," IEEE Trans IT, Vol IT-20, No. 2, pp. 146-181, March 1974.

[28] S. Kay and S. Marple, " Spectrum Analysis- A Modern Perspective," Proceedings of the IEEE, Vol 69, No. 11, November 1981, p1380.

[29] S. Orfanidis, Optimal Signal Processing: An Introduction, MacMillan Publishing, New York, 1988.

[30] M. Hagan, H. Demuth, and P. Singgih, " Parallel Signal Processing Research on the HEP," Proceedings of the 1985 International Conference on Parallel Processing, St. Charles, ILL, pp. 599-606, August 20-23, 1985.

[31] J. Burg," Maximum Entropy Spectral Analysis," Ph.D. Dissertation, Stanford University, May 1975.

[32] G. Golub and C. Van Loan, Matrix Computations , The Johns Hopkins University Press, Baltimore, MD, 1983.

[33] B. Dickinson, " Estimation of Partial Correlation Matrices Using Cholesky Decomposition," IEEE Trans AC, Vol AC-24, No.2, April 1979.

[34] R. Yarlagadda, J. Bednar, and T. Watt, " Fast Algorithms for L_p Deconvolution," IEEE Trans ASSP, Vol ASSP-33, No. 1, pp.174-182, February 1985.

[35] J. Shroeder, " Linear Predictive Spectral Analysis Via the L_p Norm," Ph.D. Dissertation, Oklahoma State University, Stillwater, OK, 1985.

[36] C. Kriel, " L_p -Norm Estimation Techniques Applied to Multiple Emitter Location," Ph.D. Dissertation, Oklahoma State University, Stillwater, OK, 1985.

[37] J. Lansford," L_p models in Speech Coding and Markov Chains in Speech Recognition," Ph.D. Dissertation, Oklahoma State University, Stillwater, OK, 1988.

[38] J. Bednar, R. Yarlagadda, and T. Watt, " L_1 Deconvolution and its Application to Seismic Signal Processing," IEEE Trans ASSP, Vol ASSP-34, No. 6, pp.1655-1658, December 1986.

[39] R. Yarlagadda and J. Hershey, " Signal Processing, General," Encyclopedia of Physical Science and Technology Vol 12, Academic Press, pp. 626-646, 1987.

[40] R. Denoel and J.P. Solvay, " Linear Prediction of Speech with a Least Absolute Error Criterion," IEEE Trans ASSP, Vol ASSP-33, pp. 1397-1403, December 1985.

[41] G. Lipovski, " Computer Architecture," Encyclopedia of Physical Science and Technology Vol. 3, Academic Press, pp377-389, 1987.

[42] D. Skillicorn, " A Taxonomy for Computer Architectures," IEEE Computer, Vol 21, No.11, p 46, November 1988.

[43] R. Duncan, " A Survey of Parallel Computer Architectures," IEEE Computer, Vol 23, No. 2, p 5, February 1990.

[44] B. Smith, " Architecture and Applications of the HEP Multiprocessor Computer System," Real Time Signal Processing IV, Proceedings of SPIE, 1981, pp. 241-248.

[45] "HEP FORTRAN 77 Reference Manual," Denelcor Inc Report, June 1984.

[46] J. Larson, " Multitasking on CRAY XMP-2 Multiprocessor," IEEE Computer, Vol. 17, No. 7, July 1984.

[47] C. Seitz, " The Cosmic Cube," Communications of the ACM, Vol 28, No. 1, January 1985, pp. 22-33.

[48] K. Batcher, " Design of a Massively Parallel Processor," IEEE Trans Computers, Vol C-29, No. 9, p 836, 1980.

[49] N. Sammur and M. Hagan, " Parallel Implementation Considerations for a Class of Signal Processing Algorithms," Proc. 2nd Symposium on the Frontiers of Massively Parallel Computation, Fairfax, VA, Oct. 10-12, 1988.

[50] N. Sammur and M. Hagan, " Mapping Signal Processing Algorithms on the Hypercube," Proc. Fourth Conference on Hypercube Concurrent Computers and Applications, Monterey, CA, Mar. 1989.

[51] N. Sammur and M. Hagan, " Mapping Signal Processing Algorithms on Parallel Architectures," Journal of Parallel and Distributed Computing, Vol 8, No. 2, February 1990, pp. 180-185.

APPENDIXES

APPENDIX A

SEQUENTIAL BURG LISTINGS

```
      PROGRAM BURG
      DIMENSION E(20000),B(20000),A(50),A1(50),C(50),X(20000)
      OPEN(2,FILE='ARMASEQ.DAT',STATUS='OLD')

      PRINT *,'ENTER MAX'
      READ *,MAX
      PRINT *,' ENTER M'
      READ *,M
      DO 10 I=1,M
         READ(2,*)X(I)
10    CONTINUE

      DO 100 I=1,M
         E(I)=X(I)
         B(I)=X(I)
100   CONTINUE

      DO 500 N=1,MAX

            SUM1=0.
            SUM2=0.

            PRINT *,N+1,M,N+1-N,M-N
            DO 200 I=N+1,M
               SUM1=SUM1+E(I)*B(I-N)
               SUM2=SUM2+E(I)*E(I)+B(I-N)*B(I-N)
200         CONTINUE
            C(N)=-2.*SUM1/SUM2

            IF(N.LE.1) GOTO 350
            DO 300 I=1,N-1
               A1(I)=A(I)+C(N)*A(N-I)
300         CONTINUE
            DO 310 I=1,N-1
               A(I)=A1(I)
310         CONTINUE
350         A(N)=C(N)
            DO 360 I=1,N
               WRITE(10,*)I,A(I)
360         CONTINUE

            DO 400 I=N+1,M
               TEMP=E(I)+C(N)*B(I-N)
               B(I-N)=B(I-N)+C(N)*E(I)
               E(I)=TEMP
400         CONTINUE

500   CONTINUE

      STOP
      END
```

APPENDIX B

GENERATE WHITE NOISE LISTINGS

```
      PROGRAM WHITENOISE
*
*this program generates a sequence of white random noise.
*
      OPEN(9,FILE='RNDATA.DAT',STATUS='NEW')

      PRINT *,'ENTER VARIENCE,LENGTH OF SEQUENCE, AND ROOT'
      READ *,VAR,LEN,INIT

      DO 100 I=1,LEN
         CALL RANDOM(INIT,Y,VAR)
         WRITE (9,*) Y
 100  CONTINUE

      STOP
      END
*
*SUBROUTINE TO GENERATE RANDOM NOISE WITH A GIVEN VARIENCE
*
      SUBROUTINE RANDOM(INIT,Y,VAR)

      INIT=MOD(3125*INIT,65536)
      Y=INIT
      Y=Y/65536
      Y=(Y-.5)*SQRT(12.)*SQRT(VAR)

      RETURN
      END
```

APPENDIX C

SIMULATE ARMA LISTINGS

```
      PROGRAM SIMARMA
*
*this program simulates an ARMA process.
*
      DIMENSION Z(-10:50000),A(-10:50000)
      DIMENSION PRAR(10),PRMA(10)

      OPEN (9,FILE='RNDATA.DAT',STATUS='OLD')
      OPEN (10,FILE='ARMASEQ.DAT',STATUS='NEW')

      PRINT *,'ENTER ORDER OF AR,ORDER OF MA,&SEQUENCE LEN'
      READ *,ORAR,ORMA,LEN

      DO 10 I=1,ORAR
            PRINT *,'FOR AR PROCESS, ENTER COEFFICIENTS #',I
            READ *,PRAR(I)
   10 CONTINUE

      DO 20 I=1,ORMA
            PRINT *,'FOR MA PROCESS, ENTER COEFFICIENTS #',I
            READ *,PRMA(I)
   20 CONTINUE

      DO 30 I=0,LEN-1
            READ(9,*) A(I)
   30 CONTINUE

      DO 40 M=0,LEN-1
            TEMP1=0.
      DO 50 K=1,ORAR
            TEMP1=TEMP1+PRAR(K)*Z(M-K)
   50 CONTINUE
      TEMP2=A(M)
      DO 60 K=1,ORMA
            TEMP2=TEMP2+PRMA(K)*A(M-K)
   60 CONTINUE
      Z(M)=TEMP1+TEMP2
      WRITE(10,*) Z(M)
   40 CONTINUE
      STOP
      END
```

APPENDIX D

HEP LISTINGS

```
      DIMENSION X(5000),JJ(20)
      CHARACTER*64 FNAME
      LOGICAL $DONE1,$DONE2,DUMMY
      COMMON/EC/E(5000),B(5000),A(50),C(50),N,M,NPROC,
     1          $K,$DONE1,$DONE2
      COMMON/EB1/S1,S2

      T=15.
      CREATE TIMOUT(T)

C     READ IN INPUT PARAMETERS

      READ(5,25) NPROC
      READ(5,25) MAX
      READ(5,25) M
      READ(5,50) (X(I), I=1,M)
 25   FORMAT(I5)
 50   FORMAT(F10.0)

C     INITIALIZING

      DO 100 I=1,M
           E(I)=X(I)
           B(I)=X(I)
 100  CONTINUE

C     MAIN LOOP

      CALL CLOCK(ITIMO)

      DO 500 N=1,MAX

C          CALCULATE C(N)

           PURGE $K,$DONE1,$DONE2
           S1=0.0
           S2=0.0
           $K=NPROC
           IF (NPROC.EQ.1) GO TO 210
           DO 200 J=1,NPROC-1
           JJ(J)=J
                CREATE EB(JJ(J))
 200       CONTINUE
 210       CALL EB(NPROC)
           DUMMY=$DONE1
           C(N)=-2.*S1/S2

C          CALCULATE A(1),..,A(N)

           CREATE AUTO

C          UPDATE E AND B

           PURGE $K,$DONE1,$DONE2
```

```
            $K=NPROC
            IF(NPROC.EQ.1) GO TO 410
            DO 400 J=1,NPROC-1
                 JJ(J)=J
                 CREATE EBUPDAT(JJ(J))
 400        CONTINUE
 410        CALL EBUPDAT(NPROC)
            DUMMY=$DONE1
            DUMMY=$DONE2
 500   CONTINUE

C      PRINT OUT RESULTS

       CALL CLOCK(ITIM1)
       TTIME=(ITIM1-ITIM0)*1.0E-7
       WRITE(6,525) M,MAX,NPROC,TTIME
 525   FORMAT(10X,'NUMBER OF DATA POINTS =',I5/
      1     10X,'NUMBER OF REFLECTION COEFFICIENTS =',I5/
      1     10X,'NUMBER OF PROCESSORS =',I4/
      1     10X,'TOTAL TIME FOR THIS RUN =',F10.5//)
       DO 600 I=1,MAX
            WRITE(6,550) I,C(I),I,A(I)
 550
       FORMAT(10X,'C(',I3,')=',F10.5,5X,'A(',I3,')=',F10.5)
 600 CONTINUE
       STOP
       END

C      SUBROUTINE EB
C
C      THIS SUBROUTINE IS USED IN THE CALCULATION OF C(N)

       SUBROUTINE EB(J)
       COMMON/EC/E(5000),B(5000),A(50),C(50),N,M,NPROC,
      1          $K,$DONE1,$DONE2
       COMMON/EB1/S1,S2
       LOGICAL $DONE1,$DONE2

       SUM1=0.0
       SUM2=0.0
       DO 10 I=N+J,M,NPROC
            SUM1=SUM1+E(I)*B(I-N)
            SUM2=SUM2+E(I)*E(I)+B(I-N)*B(I-N)
 10    CONTINUE
       K1=$K-1
       IF(K1.EQ.0) $DONE1=.TRUE.
       S1=S1+SUM1
       S2=S2+SUM2
       $K=K1

       RETURN
       END

C      SUBROUTINE AUTO
```

```
C
C       THIS SUBROUTINE IS USED TO CALCULATE THE
C       AUTOREGRESSIVE COEFFICIENTS

        SUBROUTINE AUTO
        DIMENSION A1(50)
        COMMON/EC/E(5000),B(5000),A(50),C(50),N,M,NPROC,
       1          $K,$DONE1,$DONE2
        LOGICAL $DONE1,$DONE2
        IF(N.LE.1)GO TO 300
        DO 100 I=1,N-1
             A1(I)=A(I)+C(N)*A(N-I)
 100    CONTINUE
        DO 200 I=1,N-1
             A(I)=A1(I)
 200    CONTINUE
 300    A(N)=C(N)
        $DONE2=.TRUE.
        RETURN
        END

C       SUBROUTINE EBUPDAT(J)
C
C       THIS SUBROUTINE UPDATES THE FORWARD AND
C       BACKWARD PREDICTION ERRORS

        SUBROUTINE EBUPDAT(J)
        COMMON/EC/E(5000),B(5000),A(50),C(50),N,M,NPROC,
       1          $K,$DONE1,$DONE2
        LOGICAL $DONE1,$DONE2

        DO 100 I=N+J,M,NPROC
             TEMP=E(I)+C(N)*B(I-N)
             B(I-N)=B(I-N)+C(N)*E(I)
             E(I)=TEMP
 100    CONTINUE
        K1=$K-1
        IF(K1.EQ.0) $DONE1=.TRUE.
        $K=$K1

        RETURN
        END
```

APPENDIX E

iPSC/2 LISTINGS

```
      PROGRAM HOST
C
C The following tasks are performed:
C           1- initialize the variables used in the program
C           2- prompt user for # reflection coeffs,length of
C              input data sequence,and number of nodes.
C           3- decides on the size of the cube to be allocated
C           4- gets the cube,set the hostpid,& load the nodes
C           5- sends the user input to node zero(root node)
C           6- receives the AR pars and the time elapsed
C           7- print the results.

      INTEGER*4   APPLPID,ALLNODES,HOSTPID,NN
      INTEGER*4   KEEP,TYPEA,TYPEB,TYPEC,TYPELENX,TYPEZ
      CHARACTER*4 CUBETYPE
      CHARACTER*5 CUBENAME,SRMNAME
      REAL*4      X(30000),A(50),C(50),A1(50)
      INTEGER*4   M,MAX,NN,IY(3),IT

      OPEN(2,FILE='armaseq.dat',STATUS='OLD')

C initialiaze the program variables
      HOSTPID=1
      KEEP=0
      CUBENAME='hyper'
      ALLNODES=-1
      APPLPID=1
      TYPEA=10
      TYPEB=20
      TYPEC=30
      TYPELENX=200
      TYPEZ=80

C prompt user for input
      PRINT *,' PLEASE ENTER # REFLECTION COEFFICIENTS'
      READ *,MAX
      PRINT *,' PLEASE ENTER LENGTH OF DATA SEQUENCE'
      READ *,M
      PRINT *,' PLEASE ENTER NUMBER OF NODES'
      READ *,NN
      DO 10 I=1,M
           READ (2,*)X(I)
 10   CONTINUE
      PRINT *,X(1)
      PRINT *,X(M)

C determine the size of the cube and set CUBETYPE accordingly
      IF((NN.GE.1).AND.(NN.LE.2))    CUBETYPE='d1m1'
      IF((NN.GT.2).AND.(NN.LE.4))    CUBETYPE='d2m1'
      IF((NN.GT.4).AND.(NN.LE.8))    CUBETYPE='d3m1'
      IF((NN.GT.8).AND.(NN.LE.16))   CUBETYPE='d4m1'
      IF((NN.GT.16).AND.(NN.LE.32)) CUBETYPE='d5m1'

C get the cube, set host pid, and load the nodes
```

```
      PRINT *,'GETTING THE CUBE...'
      CALL GETCUBE(CUBENAME,CUBETYPE,SRMNAME,KEEP)
      CALL SETPID(HOSTPID)
      PRINT *,'LOADING THE CUBE...'
      CALL LOAD('node',ALLNODES,APPLPID)
      PRINT *,'LOAD SUCCESSFUL...'

C send data to node zero
      IY(1)=MAX
      IY(2)=M
      IY(3)=NN
      LENX=M*4
      CALL CSEND(TYPELENX,IY,12,0,APPLPID)
      CALL CSEND(TYPEA,X,LENX,0,APPLPID)

C receive reflection coefficients and calculate the AR
parameters
      DO 20 N=1,MAX
           CALL CRECV(TYPEB,C(N),4)
           IF (N.GT.1) THEN
           DO 15 I=1,N-1
                A1(I)=A(I)+C(N)*A(N-I)
 15        CONTINUE
           DO 18 I=1,N-1
                A(I)=A1(I)
 18        CONTINUE
           ENDIF
           A(N)=C(N)
 20   CONTINUE

C receive time elapsed and print results
      CALL CRECV(TYPEZ,IT,4)
      TIME=FLOAT(IT)/1000.
      PRINT *,'TIME ELAPSED(in sec.)=',TIME
      PRINT *,'AR COEFFICIENTS ARE:'
      DO 30 N=1,MAX
         PRINT *,N,A(N)
 30   CONTINUE

      PRINT *,'CLEARING THE CUBE...'
      CALL RELCUBE(CUBENAME)

      END



      PROGRAM NODE

      INCLUDE  '/usr/include/fcube.h'
C     1-if node zero do the following:
C          a- receive data from host
C          b- start timing
C          c- calculate the ranges of indices for data to be
```

```
C              sent to other nodes
C         d- send the appropriate data to the assigned node
C              and perform calculation on node 0 data
C         e- receive partial sums from nodes and sum them
C         f- stop timer & send the sum & time to host
C    2-if other nodes do the following:
C              a-receive data from node zero
C              b-calculate partial sums & send to node 0

      INTEGER*4
      TYPEA,TYPEB,TYPEC,TYPED,TYPEE,TYPEF,TYPELENX,TYPELXS
      INTEGER*4 HOSTPID,APPLPID,HOST
      REAL*4    XS(15000)
      INTEGER*4 M,MAX,NN,IY(3)
      INTEGER*4 LL(31),LU(31)
      INTEGER*4 LSEND(3)
      REAL*4    E(30000),B(30000),SUM(2),C(50)
      INTEGER*4 TYPES,TYPEE1,TYPER,TYPEZ
      INTEGER*4 ITS,ITF,IT


C initialize the variables
      HOSTPID=1
      TYPEA=10
      TYPEB=20
      TYPEC=30
      TYPED=40
      TYPEE=50
      TYPEF=60
      TYPEZ=80
      TYPELENX=200
      TYPELXS=210
      TYPEE1=299
      HOST=MYHOST()
      ME=MYNODE()
      APPLPID=1

      IM2=0
C if this is node zero
      IF (ME.EQ.0) THEN
C receive data from host
      CALL CRECV(TYPELENX,IY,12)
C     ITS=MCLOCK()
      LENX=IY(2)*4
      CALL CRECV(TYPEA,E,LENX)
      MAX=IY(1)
      M=IY(2)
      NN=IY(3)

C CALCULATE LOWER AND UPPER LIMITS FOR X TO BE SEND TO NODES
      NUM=M/NN
      NADD=M-NUM*NN
      DO 10 I=1,NN-1
           LL(I)=I*NUM+1
```

```
          LU(I)=LL(I)+NUM-1
          IF (I.EQ.NN-1) LU(I)=LU(I)+NADD
 10   CONTINUE

C SETUP THE ARRAYS TO BE SENT TO NODES AND SEND THEM
      DO 20 I=1,NN-1
          DO 15 J=LL(I),LU(I)
               XS(J-LL(I)+1)=E(J)
 15       CONTINUE
          LXS=(LU(I)-LL(I)+1)*4

          ND=GRAY(I)
          LSEND(1)=LXS
          LSEND(2)=MAX
          LSEND(3)=NN-1
          CALL CSEND(TYPELXS,LSEND,12,ND,APPLPID)
          CALL CSEND(TYPED,XS,LXS,ND,APPLPID)

 20   CONTINUE

C done initializing

C set E and B equal to X
      DO 31 I=1,NUM
          B(I)=E(I)
 31   CONTINUE
      ITN2=0


CCCCCCCCMAIN LOOPCCCCCC
      TYPES=2
      ITS=MCLOCK()
      DO 50 N=1,MAX

C shifting if more than one node used
      ITN1=MCLOCK()
      IF(NN.GT.1) THEN
          IHNDEX1=NUM+N
          CALL CRECV(TYPES,E(NUM+N),4)
      ELSE
          INDEX1=NUM
      ENDIF
      ITN2=MCLOCK()-ITN1+ITN2
C calculate partial sums
      SUM1=0.
      SUM2=0.
      DO 35 I=N+1,INDEX1
          SUM1=SUM1+E(I)*B(I-N)
          SUM2=SUM2+E(I)*E(I)+B(I-N)*B(I-N)
 35   CONTINUE
C receive partial sums and produce the final 2 sums
      IM1=MCLOCK()
      DO 40 I=1,NN-1
          CALL CRECV(TYPEE1,SUM,8)
```

```
            SUM1=SUM1+SUM(1)
            SUM2=SUM2+SUM(2)
 40    CONTINUE
C      IM2=MCLOCK()-IM1+IM2
C calculate reflection coef and broadcast to other nodes
       C(N)=-2.*SUM1/SUM2
       DO 45 I=1,NN-1
            CALL CSEND(TYPEF,C(N),4,GRAY(I),APPLPID)
 45    CONTINUE
       CALL CSEND(TYPEB,C(N),4,HOST,HOSTPID)
       IM2=MCLOCK()-IM1+IM2
C update forward and backward reflection coefficients
       DO 48 I=N+1,INDEX1
            TEMP=E(I)+C(N)*B(I-N)
            B(I-N)=B(I-N)+C(N)*E(I)
            E(I)=TEMP
 48    CONTINUE
 50    CONTINUE
C stop clock and send time to host
       ITF=MCLOCK()
       IT=ITF-ITS
       IM2=IM2+ITN2
       TIME=FLOAT(IM2)/1000.
       PRINT *,'TIME',TIME
       CALL CSEND(TYPEZ,IT,4,HOST,HOSTPID)


CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C if any other node but node zeroC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
       ELSE

C receive the partial aray XS
       CALL CRECV(TYPELXS,LSEND,12)
       LXS=LSEND(1)
       MAX=LSEND(2)
       NNMOD=LSEND(3)
       CALL CRECV(TYPED,XS,LXS)

C done initialization
C set E and B equal to XS
       DO 76 I=1,LXS/4
            E(I)=XS(I)
            B(I)=XS(I)
 76    CONTINUE

C assign types for send and receive
       IF(MOD(GINV(ME),2).EQ.0) THEN
            TYPES=1
            TYPER=2
       ELSE
            TYPES=2
            TYPER=1
       ENDIF
```

```
C assign some useful variables
      MEGGIN=GRAY(GINV(ME)-1)
      MEGIN=GINV(ME)

CCCCCCCMAIN LOOPCCCCCCC

      DO 51 N=1,MAX
C shift
      CALL CSEND(TYPES,E(N),4,MEGGIN,APPLPID)
C taking care of the last node to the right
      IF(MEGIN.NE.NNMOD) THEN
      INDEX=LXS/4+N
      CALL CRECV(TYPER,E(INDEX),4)
      ELSE
      INDEX=LXS/4
      ENDIF
C
C calculate partial sums
      SUM(1)=0.
      SUM(2)=0.
      DO 36 I=N+1,INDEX
            SUM(1)=SUM(1)+E(I)*B(I-N)
            SUM(2)=SUM(2)+E(I)*E(I)+B(I-N)*B(I-N)
 36   CONTINUE
C send partial sums to node 0 and receive reflection coef
      CALL CSEND(TYPEE1,SUM,8,0,APPLPID)
      CALL CRECV(TYPEF,C(N),4)
C update forward and backward prediction errors
      DO 49 I=N+1,INDEX
            TEMP=E(I)+C(N)*B(I-N)
            B(I-N)=B(I-N)+C(N)*E(I)
            E(I)=TEMP
 49   CONTINUE

 51   CONTINUE

      ENDIF

      END
```

APPENDIX F

ALLIANT FX/8 LISTINGS

```
      program burg

      real etime1,etime2,etimeoh
      real ustime1(2),ustime2(2),utimeoh,stimeoh
      dimension e(16384),b(16384),x(16834),temp(16384)
      dimension a(10),a1(10),c(10)
      open(10,file='armaseq.dat',status='old')

      print *,'enter number of reflection coefficients'
      read *,max
      print *,'enter length of data sequence'
      read *,m

      do 2 i=1,m
           read(10,*)x(i)
2     continue
      do 3 i=1,m
           e(i)=x(i)
           b(i)=x(i)
3     continue

      do 5 i=1,3
           etime1=etime(ustime1)
           etime2=etime(ustime2)
           etimeoh=etime2-etime1
           utimeoh=ustime2(1)-ustime1(1)
           stimeoh=ustime2(2)-ustime1(2)
5     continue

      etime1=etime(ustime1)
      do 100 n=1,max
           s1=0.0
           s2=0.0
           s1=sum(e(n+1:m)*b(1:m-n))
           s2=sum(e(n+1:m)**2+b(1:m-n)**2)
           c(n)=-2.0*s1/s2
           if(n.gt.1)then
                a1(1:n-1)=a(1:n-1)+c(n)*a(n-1:1)
                a(1:n-1)=a1(1:n-1)
           endif
           a(n)=c(n)
           temp(1:m-n)=e(n+1:m)+c(n)*b(1:m-n)
           b(1:m-n)=b(1:m-n)+c(n)*e(n+1:m)
           e(n+1:m)=temp(1:m-n)
100   continue
      etime2=etime(ustime2)
      print *,'total time=',etime2-etime1-etimeoh
      do 200 i=1,max
           print *,'i,a(i)',i,a(i)
200   continue
      stop
      end
```

APPENDIX G

MPP LISTINGS

```
program VAX_prog(input,output,vax_x_file);

(*Program to generate matrices on front end*)

type
     a_record=array[0..127] of real;
var
     vax_x_file: file of a_record;

function random(var init:integer):real;
begin
     init:=(3125*init) mod 65536;
     random:=((init/65536)-0.5)*sqrt(12);
end;

procedure gen_VAX_file;

type
     arr=array [0..16384} of real;
var
     ranum,z:arr;
     prar:real;
     init,i,j: integer;
     vax_x_array,vax_y_array: a_record;

begin
     open(vax_x_file,file_name:='xfile.dat');
     rewrite(vax_x_file);
     init:= 983;
     for i:=1 to 16384 do
          ranum[i]:=random(init);
     prar:=0.1;
     z[0]:=0.0;
     for i:=1 to 16384 do
          z[i]:=prar*z[i-1]+ranum[i];
     for i:=0 to 127 do
          begin
          for j:=0 to 127 do
               begin
               vax_x_array[j]:=z[j+i*128];
          end;
          write(vax_x_file,vax_x_array);
     end;
end; {of procedure}

begin {of program}
     gen_VAX_file;
end. {of program}
```

```
(*$d+*)
program burg(input,output,vax_x_file);

const
     top=8;
     listsize=32;
%include 'type.dat'
     par_array= parallel array[0..127,0..127] of real;
     stag_array=stager array[0..127,0..127] of real;
     arr = array [0..100] of real;
var
     vax_x_file; file of stag_array;
     section: unsigned;
     e,b,x,s1,s2,temp:par_array;
     c: arr;
     max,m,n:integer;
     sum1,sum2:real;

procedure pfm_init;EXTERN;
procedure pfm_close;EXTERN;
procedure pfm_start(VAR section: unsigned); EXTERN;
procedure pfm_stop(VAR section: unsigned); EXTERN;

function snake_shift(var x : par_array):par_array;
var
     r1,r2:par_array;
     row_index,col_index:par_array;
begin
     r2:=shift(x,0,1);
     r1:=rotate(x,1,1);
     where (col_index=127) do
          where (row_index < 127) do
               r2:=r1;
               snake_shift:=r2;
end; {snake_shift}

begin {burg}

(*Open VAX files for reading*)
reset(vax_x_file);

(*Read the VAX file and load data into stager*)
get(vax_x_file);

(*Move data from stager tp array*)
transfer(vax_x_file,x);
waitio;

writeln('enter max');
readln(max);
pfm_init;
e:=x;
b:=x;
```

```
(*Perform computations in array*)
for n:=1 to max
     do begin
          e:=snake_shift(e);
          s1:=e*b;
          s2:=sqr(s1);
          sum1:=sum(s1,1,2);
          sum2:=sum(s2,1,2);
          c[n]:=-2.0*sum1/sum2;
          temp:=c[n]*b+e;
          b:=c[n]*e+b;
          e:=temp;
end; {for}
pfm_close;

(*Print Results*)
for n:=1 to max do
     writeln(c[n]);
end.
```

APPENDIX H

CM-2 LISTINGS

```
      PROGRAM BURG
      DIMENSION E(16384),B(16384),TEMP(16384),X(16384)
      DIMENSION A(50),C(50)
      OPEN(2,FILE='ARMASEQ.DAT',STATUS='OLD')

      PRINT *,'ENTER MAX'
      READ *,MAX
      PRINT *,'ENTER M'
      READ *,M
      DO 10 I=1,M
           READ(2,*)X(I)
10    CONTINUE

      E=X
      B=X

      CALL CM_TIMER_CLEAR(0)
      CALL CM_TIMER_START(0)
      DO 500 N=1,MAX
           E=E(2:M-N+1)
           B=B(1:M-N)
           C(N)=-2.0*SUM(E*B)/SUM(E**2+B**2)
           IF(N.EQ.1)THEN
                A(1)=C(1)
           ELSE
                A(1:N-1)=A(1:N-1)+C(N)*A(N-1:1)
           ENDIF
           TEMP=E+C(N)*B
           B=B+C(N)*E
           E=TEMP
500   CONTINUE
      CALL CM_TIMER_STOP(0)
      CALL CM_TIMER_PRINT(0)

      DO 300 N=1,MAX
           PRINT *,'C(',N,')=',C(N)
300   CONTINUE
      STOP
      END
```

APPENDIX I

CRAY X-MP/48 LISTINGS

```
      PROGRAM BURG
      DIMENSION E(16384),B(16384),A(10),A1(10),C(10),X(16384)
      OPEN(2,FILE='ARMASEQ.DAT',STATUS='OLD')

      PRINT *,'ENTER MAX'
      READ *,MAX
      PRINT *,' ENTER M'
      READ *,M
      DO 10 I=1,M
         READ(2,*)X(I)
 10   CONTINUE

      DO 100 I=1,M
         E(I)=X(I)
         B(I)=X(I)
 100  CONTINUE

      IVAR1=ITMUCPU()
      DO 500 N=1,MAX
           SUM1=0.
           SUM2=0.
           DO 200 I=N+1,M
              SUM1=SUM1+E(I)*B(I-N)
              SUM2=SUM2+E(I)*E(I)+B(I-N)*B(I-N)
 200       CONTINUE
           C(N)=-2.*SUM1/SUM2
           IF(N.LE.1) GOTO 350
           DO 300 I=1,N-1
              A1(I)=A(I)+C(N)*A(N-I)
  300      CONTINUE
           DO 310 I=1,N-1
              A(I)=A1(I)
  310      CONTINUE
  350      A(N)=C(N)
           DO 400 I=N+1,M
              TEMP=E(I)+C(N)*B(I-N)
              B(I-N)=B(I-N)+C(N)*E(I)
              E(I)=TEMP
 400       CONTINUE
 500  CONTINUE

      IVAR2=ITMUCPU()
      ITIME=IVAR2-IVAR1-4
*TIME IN MICROSECONDS(NOTE:4 IS OVERHEAD)
      PRINT*,'TIME(usec)=',ITIME
      PRINT *,'C(1)=',C(1)
      STOP
      END
```

APPENDIX J

CRAY-2 LISTINGS

```
      PROGRAM BURG
      DIMENSION E(16384),B(16384),A(10),A1(10),C(10),X(16384)
      OPEN(2,FILE='ARMASEQ.DAT',STATUS='OLD')

      PRINT *,'ENTER MAX'
      READ *,MAX
      PRINT *,' ENTER M'
      READ *,M
      DO 10 I=1,M
         READ(2,*)X(I)
10    CONTINUE

      DO 100 I=1,M
         E(I)=X(I)
         B(I)=X(I)
100   CONTINUE

      CALL SECOND(S1)

      DO 500 N=1,MAX

            SUM1=0.
            SUM2=0.
            DO 200 I=N+1,M
               SUM1=SUM1+E(I)*B(I-N)
               SUM2=SUM2+E(I)*E(I)+B(I-N)*B(I-N)
200         CONTINUE
            C(N)=-2.*SUM1/SUM2

            IF(N.LE.1) GOTO 350
            DO 300 I=1,N-1
               A1(I)=A(I)+C(N)*A(N-I)
300         CONTINUE
            DO 310 I=1,N-1
               A(I)=A1(I)
310         CONTINUE
350         A(N)=C(N)
            DO 360 I=1,N
               PRINT *,A(I)
360         CONTINUE

            DO 400 I=N+1,M
               TEMP=E(I)+C(N)*B(I-N)
               B(I-N)=B(I-N)+C(N)*E(I)
               E(I)=TEMP
400         CONTINUE

500   CONTINUE

      CALL SECOND(S2)
      PRINT *,'TIME=',S2-S1
      PRINT *,'C(1)=',C(1)
      STOP
      END
```

VITA

Nidal M. Sammur

Candidate for the Degree of

Doctor of Philosophy

Thesis: MAPPING SIGNAL PROCESSING ALGORITHMS ON PARALLEL ARCHITECTURE

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Amman, Jordan, June 5, 1963, the son of Musa and Fatima Sammur.

Education: Graduated from Terra Santa College, Amman, Jordan in May 1980; received Bachelor of Science degree in Electrical Engineering from the University of Tulsa in May of 1984. Received Master of Science degree in Electrical Engineering from the University of Tulsa in December of 1986; completed requirements for the Doctor of Philosophy degree at Oklahoma State University in July, 1992.

Professional Experience: Graduate Teaching Assistant, The University of Tulsa, 1984 to 1986; Graduate Research Assistant, Oklahoma State University, 1986 to 1991.