Uniform Approach for Solving Some Classical Problems on a Linear Array

David R. O'Hallaron*

Abstract

We show that the same basic recurrence can be used to solve a number of classical problems from linear algebra and graph theory, including instances of the algebraic path problem, matrix multiplication, matrix triangularization, and matrix transpose, and we give a simple mapping of the recurrence onto a unidirectional linear array. Qualitative advantages to programming linear arrays using this approach include uniformity of design, simplicity of programming, and scalability to larger problems. The major disadvantage is that the resulting algorithms are not necessarily optimal.

* An earlier version of this paper was presented at the International Conference on Parallel Processing, St. Charles, IL, August, 1989. This work was supported in part by the GE Aerospace Business Group, while the author was with the GE Research and Development Center, Schenectady, NY, 12301. The author's current address is School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 15213.

Keywords

parallel algorithm, linear array, matrix computation, linear algebra, graph theory.

1 Introduction

In this paper, we describe a technique for solving a number of classical problems on a unidirectional linear array using essentially the same parallel algorithm.

First, we demonstrate that some interesting problems from linear algebra and graph theory, including instances of the algebraic path problem, matrix multiplication, matrix triangularization, and matrix transpose, can be solved using the same recurrence. Informally, at the *kth* step in the recurrence, transformation $T^{(k)}$ is applied to square matrix $A^{(k-1)}$. The resulting matrix is then permuted by circularly shifting its columns one position towards the left and its rows one position towards the top. The permuted matrix $A^{(k)}$ becomes the input to the $(k+1)st$ step.

Next, we give a simple mapping of the recurrence onto a linear array, where the *kth* cell inputs matrix $A^{(k-1)}$ from the left, applies transformation $T^{(k)}$, permutes the resulting matrix, and sends the permuted matrix $A^{(k)}$ to the right, all on an element-by-element basis.

This approach (when applicable) offers several qualitative advantages. First, it can simplify the design process. If we are able to cast a problem into the desired recurrence, the mapping onto the linear array is straightforward and mechanical. Second, it can simplify the programming process because each cell executes the same code. Third, it is easily generalized to handle large problems on small arrays. The major disadvantage of the approach is that the resulting algorithms are not necessarily optimal.

The work builds on previous work by S.Y. Kung [1] and P.S. Tseng [2], each of whom used permutations to design parallel algorithms for instances of the algebraic path problem, Kung using a 2-D array, Tseng using a linear array. Our contribution is to show that a similar technique is useful on a larger class of problems. These problems all share the same localized and regular dependency structure that is exploited by systolic algorithms [3], wavefront algorithms [4], and automatic synthesis algorithms [5, 6, 7, 8, 9].

Section 2 describes the recurrence and gives numerous examples of its utility. Section 3 describes a simple mapping of the recurrence onto a linear array. The paper concludes with a discussion of the results.

2 Recurrence

A surprising number of classical problems from graph theory and linear algebra can be solved using the same simple recurrence. We define the recurrence and then give some examples of its usefulness.

2.1 Definitions

Given an $n \times n$ matrix $A = \{a_{ij}\}$, we want to compute the $n \times n$ matrix, $B = \{b_{ij}\}$, using a recurrence of the form:

$$\begin{aligned} A^{(0)} &= A \\ A^{(k)} &= CT^{(k)}(A^{(k-1)})C^{-1}, \quad k = 1, \ldots, m \\ B &= A^{(m)} \end{aligned} \tag{1}$$

The symbol C denotes an $n \times n$ permutation matrix,

$$C = \begin{bmatrix} e_n & e_1 & e_2 & \cdots & e_{n-1} \end{bmatrix}$$

where e_i is an $n \times 1$ column vector such that the only non-zero element is a 1 in the *ith* position. Notice that

$$C^{-1} = C^T = \begin{bmatrix} e_2 & e_3 & \cdots & e_n & e_1 \end{bmatrix}.$$

Multiplying a matrix A on the left by C and on the right by its inverse C^{-1} has the effect of circularly shifting each column of A one position towards the left and each row of A one position towards the top. For example,

$$\begin{bmatrix} 5 & 6 & 4 \\ 8 & 9 & 7 \\ 2 & 3 & 1 \end{bmatrix} = C \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} C^{-1}.$$

S.Y. Kung [1] and P. S. Tseng [2] have used a similar permutation to design parallel algorithms for instances of the algebraic path problem, Kung in the context of an optimal algorithm for 2-D arrays, and Tseng in the context of an implementation of the shortest paths problem on the Warp computer [10]. As we shall see, this type of permutation is useful for solving other kinds of problems as well.

The symbol $T^{(k)}$ denotes a transformation applied to $A^{(k-1)}$ such that

$$\hat{A}^{(k)} = \{\hat{a}_{ij}^{(k)}\} = T^{(k)}(A^{(k-1)})$$

where $\hat{A}^{(k)}$ is an $n \times n$ matrix. Let

$$S_{ij} = \{(p, q) \mid p \leq \max(i, i') \text{ and } q \leq \max(j, j')\} \tag{2}$$

be a set of *index pairs*, where for some index r,

$$r' = \begin{cases} n & \text{if } r = 1 \\ r - 1 & \text{otherwise.} \end{cases}$$

Let $(p,q)^{(k-1)}$ denote the index pair associated with $a_{pq}^{(k-1)}$. Then the only restriction on transformation $T^{(k)}$ is that each $\hat{a}_{ij}^{(k)}$ must be computable using only elements $a_{pq}^{(k-1)}$ such that $(p,q)^{(k-1)} \in S_{ij}$.

Set S_{ij} defines a restriction on transformation $T^{(k)}$ that limits the collection of elements of $A^{(k-1)}$ that can be used to compute the element $\hat{a}_{ij}^{(k)}$. As we will see in the following sections, the restriction on transformation $T^{(k)}$ defined by S_{ij} is weak enough to allow a significant number of problems from linear algebra and graph theory to be solved using a recurrence in the form of Equation (1), yet strong enough to admit a uniform parallel solution for each of these problems on a linear array.

The set S_{ij} can be interpreted visually using the graph in Figure 1, for the case where $n = 4$. Unfilled vertices in the graph correspond to index pairs of $A^{(k-1)}$. Filled vertices in the graph correspond to index pairs of $A^{(k)}$. If we let $a \xrightarrow{*} b$ denote the existence of a path from vertex a to vertex b, then we can interpret set S_{ij} as

$$S_{ij} = \left\{ (p,q)^{(k-1)} \mid (p,q)^{(k-1)} \xrightarrow{*} (i',j')^{(k)} \right\}. \tag{3}$$

For example, using either Equation (2) or (3), we find that for $n = 4$,

$$\begin{aligned} S_{11} &= \{(1,1), \ldots (1,4), (2,1), \ldots, (4,3), (4,4)\} \\ S_{12} &= \{(1,1), \ldots (4,1), (1,2), \ldots, (4,2)\} \\ S_{22} &= \{(1,1), (1,2), (2,1), (2,2)\} \end{aligned}$$

2.2 Examples

This section gives examples of some problems — in particular, instances of the algebraic path problem, matrix multiplication, matrix triangularization, and matrix transpose — that can be solved using the recurrence in Equation (1).

2.2.1 Instances of the Algebraic Path Problem

A number of problems, including the *shortest paths*, *transitive closure*, and *minimum spanning tree* problems from graph theory, and the *matrix inverse* problem from linear algebra, have been shown to be instances of the general problem known as the *algebraic path problem* (APP). Formal definitions of the APP can be found in [1, 11]. All instances of the APP can be solved using the recurrence in Equation (1). We give three examples: shortest paths, minimum spanning tree, and matrix inverse.

Shortest Paths: We are given a directed graph G with n vertices and labeled with non-negative edge weights. The length of a path p is defined to be the sum of the edge weights along p. The

problem is to determine the length of the shortest path between all pairs of vertices a and b. In Equation (1), let A be the adjacency matrix of a graph G containing $m = n$ vertices and let

$$\hat{a}_{ij}^{(k)} = \min(a_{ij}^{(k-1)}, a_{i1}^{(k-1)} + a_{1j}^{(k-1)}), \quad 1 \leq i, j \leq n.$$

Then $a_{ij}^{(n)}$ is the shortest path from vertex v_i to vertex v_j in G. This formulation is equivalent to solutions presented in [1, 11, 12]. For example, consider the graph in Figure 2 with adjacency matrix

$$A = A^{(0)} = \begin{bmatrix} 0 & 2 & 1 & \infty \\ 2 & 0 & \infty & 3 \\ 1 & \infty & 0 & 7 \\ \infty & 3 & 7 & 0 \end{bmatrix}$$

Applying Equation (1) gives

$$A^{(1)} = \begin{bmatrix} 0 & 3 & 3 & 2 \\ 3 & 0 & 7 & 1 \\ 3 & 7 & 0 & \infty \\ 2 & 1 & \infty & 0 \end{bmatrix}$$

$$A^{(2)} = \begin{bmatrix} 0 & 6 & 1 & 3 \\ 6 & 0 & 5 & 3 \\ 1 & 5 & 0 & 2 \\ 3 & 3 & 2 & 0 \end{bmatrix}$$

$$A^{(3)} = \begin{bmatrix} 0 & 5 & 3 & 6 \\ 5 & 0 & 2 & 1 \\ 3 & 2 & 0 & 3 \\ 6 & 1 & 3 & 0 \end{bmatrix}$$

$$A^{(4)} = \begin{bmatrix} 0 & 2 & 1 & 5 \\ 2 & 0 & 3 & 3 \\ 1 & 3 & 0 & 6 \\ 5 & 3 & 6 & 0 \end{bmatrix}$$

The reader can verify by inspection that $A^{(4)}$ contains the shortest paths between all pairs of vertices.

Minimum Spanning Tree: We are given an undirected graph G with n vertices, labeled with distinct non-negative edge weights. A spanning tree of G is a subgraph of G that contains n vertices and $n - 1$ edges. The size of a spanning tree is the sum of its edge weights. The problem is to find the smallest spanning tree of G. In Equation (1) let $A^{(0)} = A$ be the adjacency matrix of a graph G containing $m = n$ vertices and let

$$\hat{a}_{ij}^{(k)} = \min(a_{ij}^{(k-1)}, \ \max(a_{i1}^{(k-1)}, a_{1j}^{(k-1)})), \quad 1 \leq i, j \leq n.$$

Then by a result in [13], edge (i, j) is in the minimum spanning tree of G if and only if $a_{ij}^{(0)} = a_{ij}^{(n)}$. For example, consider the graph in Figure 2. Applying Equation (1) gives

$$A^{(4)} = \begin{bmatrix} 0 & 2 & 1 & 3 \\ 2 & 0 & 2 & 3 \\ 1 & 2 & 0 & 3 \\ 3 & 3 & 3 & 0 \end{bmatrix}$$

Comparing the elements of $A^{(4)}$ with the elements of $A^{(0)}$, the reader will find that the minimum spanning tree of the graph in Figure 2 consists of the edges (v_1, v_2), (v_1, v_3), and (v_2, v_4).

Matrix Inverse: We are given a nonsingular $n \times n$ matrix X with non–zero diagonal elements. The problem is to find (without pivoting) the unique matrix X^{-1} such that $XX^{-1} = X^{-1}X = I$. In Equation (1), let $A = I - X$. Let $m = n$ and let

$$\hat{a}_{ij}^{(k)} = \begin{cases} (1 - a_{11}^{(k-1)})^{-1} & \text{if } i = 1 \text{ and } j = 1 \\ a_{1j}^{(k-1)} \hat{a}_{11}^{(k)} & \text{if } i = 1 \text{ and } j > 1 \\ a_{i1}^{(k-1)} \hat{a}_{11}^{(k)} & \text{if } i > 1 \text{ and } j = 1 \\ a_{ij}^{(k-1)} + \hat{a}_{i1}^{(k)} a_{1j}^{(k-1)} & \text{otherwise.} \end{cases}$$

Then $X^{-1} = A^{(n)}$. Note that $\hat{a}_{11}^{(k)}$ is always defined because of the assumption of non–zero diagonal elements in X. This is equivalent to the formulation given in [14]. For example, consider the problem of finding the inverse of

$$X = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 0 & 1 & 1 \end{bmatrix}$$

We let

$$A = A^{(0)} = I - X = \begin{bmatrix} 0 & -1 & -1 \\ -1 & -1 & -3 \\ 0 & -1 & 0 \end{bmatrix}$$

Applying Equation (1) gives

$$A^{(3)} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & -1 & 2 \\ -1 & 1 & -1 \end{bmatrix}$$

The reader can verify that $A^{(3)} = X^{-1}$ by using the identity $XX^{-1} = I$.

2.2.2 Matrix Multiplication

We are given $m \times m$ matrices X and Y. The problem is to compute the $m \times m$ matrix $Z = XY$. Using a technique similar to one described in [15], let A in Equation (1) be the $n \times n$ matrix

$$A = A^{(0)} = \begin{bmatrix} 0 & Y \\ X & 0 \end{bmatrix}$$

where $n = 2m$. Let

$$\hat{a}_{ij}^{(k)} = a_{ij}^{(k-1)} + a_{i1}^{(k-1)} a_{1j}^{(k-1)}, \quad 1 \leq i, j \leq n$$

Then

$$A^{(m)} = \begin{bmatrix} Z & X \\ Y & 0 \end{bmatrix}.$$

For example, suppose that

$$X = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}$$

and

$$Y = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}$$

Then

$$A = A^{(0)} = \begin{bmatrix} 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 2 & 0 & 0 \\ -1 & 3 & 0 & 0 \end{bmatrix}$$

and applying Equation (1) gives

$$A^{(1)} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 2 & 1 & 1 \\ 3 & -2 & -1 & -1 \\ 0 & 2 & 1 & 0 \end{bmatrix}$$

$$A^{(2)} = \begin{bmatrix} 2 & 3 & 1 & 2 \\ -2 & 2 & -1 & 3 \\ 2 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

The reader can verify by inspection that

$$A^{(2)} = \begin{bmatrix} Z & X \\ Y & 0 \end{bmatrix}$$

2.2.3 Matrix Triangularization

We are given an $n \times n$ matrix A. The problem is to compute the $n \times n$ matrix product

$$B = A\Theta \tag{4}$$

where Θ is any $n \times n$ orthogonal matrix that triangularizes A. This important computation arises frequently in estimation applications such as Kalman filtering, where a factored form of a covariance matrix is updated in-place [16, 17]. A popular technique for computing B is to use orthogonal

rotations to zero the elements of the upper triangle of matrix A[18]. An orthogonal rotation is an $n \times n$ matrix of the form

$$\Theta_{ij} = \begin{bmatrix} 1 & \vdots & & \vdots & \\ \cdots & r_{ii} & \cdots & r_{ij} & \cdots \\ & \vdots & 1 & \vdots & \\ \cdots & r_{ji} & \cdots & r_{jj} & \cdots \\ & \vdots & & \vdots & 1 \end{bmatrix}$$

where

$$\begin{aligned} r_{ii} &= \cos(\theta) \\ r_{ij} &= \sin(\theta) \\ r_{ji} &= \sin(\theta) \\ r_{jj} &= -\cos(\theta) \end{aligned}$$

and where

$$\begin{aligned} \theta &= \arctan(a_{ii}/a_{ij}) \\ \cos(\theta) &= a_{ii}/(a_{ii}^2 + a_{ij}^2)^{1/2} \\ \sin(\theta) &= a_{ij}/(a_{ii}^2 + a_{ij}^2)^{1/2}. \end{aligned}$$

Postmultiplying A by Θ_{ij} zeros element a_{ij}. Since the product of two orthogonal matrices is orthogonal, Equation (4) can be computed by postmultiplying A by the appropriate orthogonal rotations:

$$B = ((\cdots((A\Theta_{12})\Theta_{13})\cdots)\Theta_{n-1,n}) \tag{5}$$

Equation (5) can be computed using the recurrence in Equation (1). Let A be the $n \times n$ matrix. Let $m = n$ and let

$$< p, q >_r \equiv \begin{bmatrix} a_{p1}^{(k-1)} & \cdots & a_{pr}^{(k-1)} \end{bmatrix} \begin{bmatrix} a_{q1}^{(k-1)} \\ \vdots \\ a_{qr}^{(k-1)} \end{bmatrix}$$

In words, $< p, q >_r$ is an inner product formed by multiplying the row vector formed from the first r elements of row p by the transpose of the row vector formed from the first r elements of row q. Also, let

$$n(i, j, k) \equiv < 1, i >_{j-1} a_{1j}^{(k-1)} - < 1, 1 >_{j-1} a_{ij}^{(k-1)}$$

and let

$$d(j) \equiv < 1, 1 >_j^{1/2} .$$

Finally, let

$$\hat{a}_{ij}^{(k)} = \begin{cases} 0 & \text{if } i = 1,\ 1 < j \leq n-k+1 \\ < 1, i >_{n-k+1} / d(n-k+1) & \text{if } 1 \leq i \leq n-k+1,\ j = 1 \\ n(i,j,k)/d(j-1)d(j) & \text{if } 1 < i,j \leq n-k+1 \\ a_{ij}^{(k-1)} & \text{otherwise.} \end{cases}$$

Then $B = A\Theta = A^{(n)}$. For example, suppose that

$$A = A^{(0)} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Applying Equation (1) gives

$$A^{(1)} = \begin{bmatrix} -1/\sqrt{2} & -1/\sqrt{2}\sqrt{3} & 2/\sqrt{3} \\ 0 & -\sqrt{2}/\sqrt{3} & 1/\sqrt{3} \\ 0 & 0 & \sqrt{3} \end{bmatrix}$$

$$A^{(2)} = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{3} & \sqrt{3}/3\sqrt{2} \\ 0 & \sqrt{3} & 0 \\ 0 & 2/\sqrt{3} & \sqrt{2}/\sqrt{3} \end{bmatrix}$$

$$A^{(3)} = \begin{bmatrix} \sqrt{3} & 0 & 0 \\ 2/\sqrt{3} & \sqrt{2}/\sqrt{3} & 0 \\ 1/\sqrt{3} & \sqrt{3}/3\sqrt{2} & 1/\sqrt{2} \end{bmatrix}$$

The reader can verify that $B = A^{(3)}$ by using the identity $BB^T = AA^T$.

2.2.4 Matrix Transpose

We are given an $n \times n$ matrix $A = \{a_{ij}\}$. The problem is to compute $B = A^T$. In Equation (1), let $m = n$. Let

$$\hat{a}_{ij}^{(k)} = \begin{cases} a_{ji}^{(k-1)} & \text{if } i = 1 \text{ and } 1 < j \leq n-k+1 \\ a_{ji}^{(k-1)} & \text{if } 1 < i \leq n-k+1 \text{ and } j = 1 \\ a_{ij}^{(k-1)} & \text{otherwise.} \end{cases}$$

Then $B = A^T = A^{(n)}$.

3 Mapping on Linear Array

In this section we give a simple approach for solving the recurrence in Equation (1) using a unidirectional linear array, and we illustrate the approach using examples from the previous section.

3.1 Machine Model

The machine model is the linear array depicted in Figure 3. The model is an informal idealization of the Warp computer [10]. Rectangles denote computational cells. Arrows denote unidirectional communications channels. Cells are numbered from left to right, starting with unity.

Cells are separately programmable and asynchronous. Each cell has its own local code and data storage. Each cell knows its number, which is contained in local variable `cellid`. Each cell runs at most one process, which is assigned to the cell at compile time. The communications channel between two cells is a bounded FIFO buffer. Data passes through the array from left to right. Cell k sends a word of data to cell $k+1$ using a statement of the form

send (`val`)

which blocks until the buffer between cells k and $k+1$ is nonfull, and then places the local variable `val` at the end of the buffer. Similarly, cell k receives a word of data from cell $k-1$ using a statement of the form

receive (`val`)

which blocks until the buffer between cells k and $k-1$ is nonempty, and then retrieves the first item from the buffer and assigns it to variable `val`.

3.2 Basic Algorithm

The mapping of matrices and transformations onto the linear array is shown in Figure 4. Cell k receives $A^{(k-1)}$ in Equation (1) from its left hand neighbor, computes the transformation

$$\hat{A}^{(k)} = T^{(k)}(A^{(k-1)}),$$

permutes $\hat{A}^{(k)}$, and sends the resulting $A^{(k)}$ in Equation (1) to its right hand neighbor. The scalar elements of $A^{(k-1)}$ are assumed to arrive at cell k in row-major order, although in general other orderings (such as column-major or diagonal-major) can be used.

Figure 5 describes the behavior of cell k. In this basic algorithm, the transformation $T^{(k)}$ in Equation (1) is simply the null transformation

$$\hat{A}^{(k)} = A^{(k-1)}.$$

By applying less trivial transformations on each cell, or equivalently, by adding some statements to the basic algorithm in Figure 5, we are able to solve any recurrence in the form of Equation (1), including all of the examples from the previous section.

For example, in Figure 6, adding Line (7a) to the basic algorithm gives a shortest paths algorithm. Notice that we take advantage of the invariance of the elements in the first row and

column of the input matrix. In Figure 7, adding lines (2a)–(2c), (5a), (7a), and (12a) to the basic algorithm gives a matrix inverse algorithm. Finally, in Figure 8, adding a number of lines to the basic algorithm gives an algorithm for matrix triangularization.

The approach we have described offers a number of practical qualitative advantages. The first advantage is uniformity of design. Once we are able to cast a problem in the form of Equation (1), the mapping onto the linear array is straightforward and mechanical. We simply take the basic algorithm in Figure 5 and add the code that implements the transformation $T^{(k)}$. The basic pattern of **send** and **receive** statements (always a tricky part of a design) does not change! The second advantage is simplicity of programming. Each cell in the linear array executes the same code; we are not forced to program each cell separately. The third advantage is that the approach generalizes easily to large problems on arrays with fewer than m cells. As a simple example, suppose that we want to compute the shortest paths for a graph with $n = 350$ vertices using an array with 10 cells. If we stream the initial adjacency matrix A through the array, the output of cell 10 is $A^{(10)}$. If we then stream $A^{(10)}$ through the array, the output of cell 10 is $A^{(20)}$. And so on. In fact, this is precisely the technique used by P. S. Tseng [2] at Carnegie Mellon for the implementation of the shortest paths problem in [10].

Assume that the transformation of each scalar element of $A^{(k-1)}$ can be applied in constant time on cell k (true for all problems discussed in this paper). Then cell k waits $O(kn)$ time for the first element of $A^{(k-1)}$ to arrive from the left, and then transforms and permutes the elements of $A^{(k-1)}$ as they arrive in $O(n^2)$ time. Thus, using m cells, the result matrix $B = A^{(m)}$ in Equation (1) is computed in $O(n^2 + mn)$ time. This points out a major quantitative disadvantage of the approach; it does not necessarily result in an optimal algorithm. For example, the parallel algorithm for matrix multiplication in [10] requires only $O(n^2)$ time on a linear array. As another example, the *sequential* algorithm for matrix transpose requires only $O(n^2)$ time.

4 Summary and Discussion

We have presented two results. First, we demonstrated that a number of classical problems from linear algebra and graph theory — in particular, instances of the algebraic path problem, matrix multiplication, matrix triangularization, and matrix transpose — can be solved using the same recurrence. Second, we gave a simple mapping of the recurrence onto a linear array.

We have reported on an interesting phenomenon, namely, that a number of interesting problems can be solved on a linear array in a similar way. Unfortunately, we do not yet understand the underlying mathematical characteristic of these problems that gives rise to this phenomenon. We identified these problems more by chance, as we developed algorithms and programs for a prototype of the Warp computer, than by any consistent and organized search. The Warp prototype supported

only unidirectional communication, with the same code executing on each of its 10 cells. In some cases, the approach presented in this paper was our only alternative. Essentially, we noticed that for many problems, the *kth* row and column of the matrix are in some way "special" during the *kth* iteration over the matrix. Further, the values of the *kth* row and column are needed to compute other elements of the matrix during the *kth* iteration. The intuition for the approach we have presented is that the permutation of the matrix in Equation (1) keeps these special rows and columns in the same position for each cell in the linear array, and the restriction on transformation $T^{(k)}$ in Equation (1) guarantees that all of the needed matrix elements have been received by a cell when it needs to pass some result on to the next cell.

The fact that a single column and row of the matrix are used to compute the other elements of the matrix is reminiscent of the rank one modification from linear algebra. Perhaps the underlying mathematical characteristic of the problems discussed in this paper can ultimately be explained in these terms.

Acknowledgements

Thanks to H. T. Kung for providing access to the Warp machine at Carnegie Mellon. Thanks also to the anonymous referees, and to Chuck Fiduccia, Bob Mattheyses, and P. S. Tseng for helpful discussions.

References

[1] S. Y. Kung, "Optimal systolic design for the transitive closure and shortest paths problems," *IEEE Transactions on Computers*, vol. C-36, pp. 603–614, May 1987.

[2] P. S. Tseng, 1987. personal communication.

[3] H. T. Kung, "Why systolic architectures?," *IEEE Computer*, pp. 37–46, Jan. 1982.

[4] S. Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. Bhaskar Rao, "Wavefront array architecture: Language, architecture, and applications," *IEEE Transactions on Computers*, vol. C-31, pp. 1054–1066, Nov. 1982.

[5] P. Quinton, "Automatic systhesis of systolic arrays from uniform recurrent equations," in *International Symposium on Computer Architecture*, pp. 208–214, IEEE, 1984.

[6] S. K. Rao, *Regular Iterative Algorithms and Their Implementations on Processor Arrays*. PhD thesis, Stanford, 1985.

[7] D. I. Moldovan and J. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Transactions on Computers*, vol. C-35, pp. 1–12, Jan. 1986.

[8] P. Lee and Z. Kedem, "Synthesizing linear array algorithms from nested for loop algorithms," *IEEE Transactions on Computers*, vol. 37, pp. 1578–1598, Dec. 1988.

[9] P. Lee and Z. Kedem, "Mapping nested loop algorithms into multidimensional systolic arrays," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 64–76, Jan. 1990.

[10] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. Webb, "The Warp computer: Architecture, implementation, and performance," *IEEE Transactions on Computers*, vol. C-36, pp. 1523–1538, Dec. 1987.

[11] G. Rote, "A systolic array algorithm for the algebraic path problem," *Computing*, vol. 34, pp. 191–219, 1985.

[12] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[13] B. M. Maggs and S. A. Plotkin, "Minimum-cost spanning tree as a path-finding problem," *Information Processing Letters*, vol. 26, pp. 291–293, Jan. 1988.

[14] Y. Robert and D. Trystram, "Systolic solution of the algebraic path problem," in *Systolic Arrays* (W. Moore, A. McCabe, and R. Urquhart, eds.), Boston: Adam Hilger, 1986.

[15] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[16] J. M. Jover and T. Kailath, "A parallel architecture for Kalman filter measurement update and parameter estimation," *Automatica*, vol. 22, no. 1, pp. 32–57, 1986.

[17] R. S. Baheti, D. R. O'Hallaron, and H. R. Itzkowitz, "Mapping extended Kalman filters onto linear arrays," 1990. to appear in IEEE Transactions on Automatic Control.

[18] G. H. Golub and C. F. V. Loan, *Matrix Computations*. Johns Hopkins University Press, 1983.

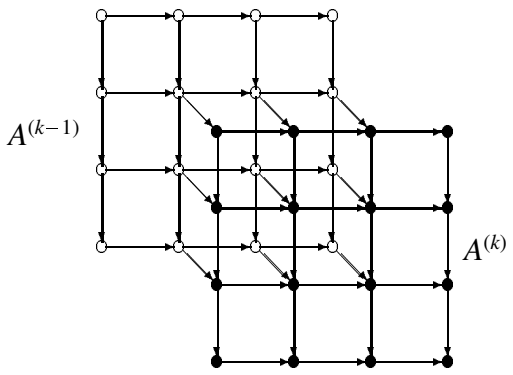

Figure 1: Visual interpretation of set S_{ij} for $n = 4$

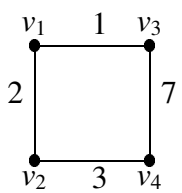

Figure 2: Graph for algebraic path problem examples, $n = 4$

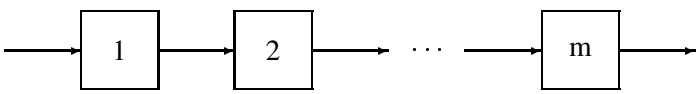

Figure 3: Linear array machine model

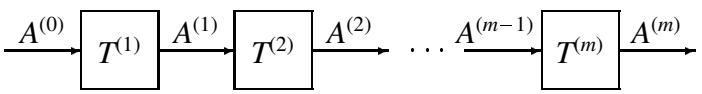

Figure 4: Mapping of the recurrence onto a linear array

```
1.    for j := 1 to n do
2.       receive(row[j]);
3.    endfor
4.    for i := 2 to n do
5.       receive(col);
6.       for j := 2 to n do
7.          receive(val);
8.          send(val);
9.       endfor
10.      send(col);
11.   endfor
12.   for j := 2 to n do
13.      send(row[j]);
14.   endfor
15.   send(row[1]);
```

Figure 5: Algorithm for cell *k* (basic algorithm)

```
1.      for j := 1 to n do
2.         receive(row[j]);
3.      endfor
4.      for i := 2 to n do
5.         receive(col);
6.         for j := 2 to n do
7.            receive(val);
7a.           val := min(val, col + row[j]);
8.            send(val);
9.         endfor
10.        send(col);
11.     endfor
12.     for j := 2 to n do
13.        send(row[j]);
14.     endfor
15.     send(row[1]);
```

Figure 6: Algorithm for cell *k* (shortest paths)

```
1.      for j := 1 to n do
2.        receive(row[j]);
2a.       if i = 1 then
2b.         row[1] := 1/(1 - row[1]);
2c.       endif
3.      endfor
4.      for i := 2 to n do
5.        receive(col);
5a.       col := col * row[1];
6.        for j := 2 to n do
7.          receive(val);
7a.         val := val + row[j]*col;
8.          send(val);
9.        endfor
10.       send(col);
11.     endfor
12.     for j := 2 to n do
12a.      row[j] := row[j]*row[1];
13.       send(row[j]);
14.     endfor
15.     send(row[1]);
```

Figure 7: Algorithm for cell *k* (matrix inverse)

```
        initially:
        ip11[0] := 0;
        last := n-cellid+1;

1.      for j := 1 to n do
2.        receive(row[j]);
2a.       if j ≤ last then
2b.         ip11[j] := ip11[j-1] + row[j]*row[j];
2c.       endif
3.      endfor
4.      for i := 2 to n do
5.        receive(col);
5a.       ip1i[1] := col*row[1];
6.        for j := 2 to n do
7.          receive(val);
7a.         if j ≤ last then
7b.           ip1i[j] := ip1i[j-1] + val*row[j];
7c.           numer := ip1i[j-1]*col - ip11[j-1]*val;
7d.           denom := sqrt(ip11[j-1])*sqrt(ip11[j]);
7e.           val := numer/denom;
7f.         endif
8.          send(val);
9.        endfor
9a.       if i ≤ last then
9b.         col := ip1i[last]/sqrt(ip11[last]);
9c.       endif
10.       send(col);
11.     endfor
12.     for j := 2 to n do
12a.    if j ≤ last then
12b.      row[j] := 0;
12c.    endif
13.       send(row[j]);
14.     endfor
14a.    row[1] := sqrt(ip11[last]);
15.     send(row[1]);
```

Figure 8: Algorithm for cell *k* (matrix triangularization)

List of Figures

1 Visual interpretation of set S_{ij} for $n = 4$ 15
2 Graph for algebraic path problem examples, $n = 4$ 16
3 Linear array machine model 17
4 Mapping of the recurrence onto a linear array 18
5 Algorithm for cell k (basic algorithm) 19
6 Algorithm for cell k (shortest paths) 20
7 Algorithm for cell k (matrix inverse) 21
8 Algorithm for cell k (matrix triangularization) 22

Mailing Address and Phone Number

Please return galley proofs to

David R. O'Hallaron
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, 15213

phone:(412)268-8199

Biography

David R. O'Hallaron received the Ph.D. degree in Computer Science from the University of Virginia, Charlottesville, in 1986.

From 1986 to 1989 he was with the GE Research and Development Center, Schenectady, NY, as a Computer Scientist. In 1989, he joined the faculty in the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, as a Systems Scientist. His main research interests are in the areas of programming models, algorithms, and applications for parallel machines.