Trends in systolic and cellular computation

George Miel *

Hughes Research Laboratories, 3011 Malibu Canyon Road, Malibu, CA 90265, United States

Received 16 October 1990
Revised 27 November 1990

Abstract


A profile is given of current research, as it pertains to computational mathematics, on Very Large Scale Integration (VLSI) array processors. In this type of parallel computers, the cells of the array operate in Single Instruction Multiple Data (SIMD) mode and algorithms are executed in systolic or cellular fashion. The focus of the presentation is on linear algebraic techniques. A systolization process is illustrated by matching an arbitrary gaxpy operation onto a fixed-size square array processor. Two recent systolic methods, for solving systems of linear equations, one iterative and the other direct, are described. A cellular algorithm for a fast Fourier transform, based on a new implementation on rectangular array processors of the perfect shuffle permutation, is then derived. Using annotated lists of recent references, snapshots of active research areas are given on systolic linear solvers, the singular value decomposition, artificial neural networks, and the simulated annealing algorithm.

Keywords: Parallel computational linear algebra, mesh array processors, systolic and cellular algorithms.

1. Introduction

Very Large Scale Integration (VLSI) mesh array processors have received considerable attention for applications requiring high throughput. In this type of parallel computers, the cells of the array operate in Single Instruction Multiple Data (SIMD) mode and algorithms are executed in systolic or cellular fashion. Two areas of applications for VLSI mesh arrays are in signal processing and in scientific computing. In the former case, the mesh array is used as a special-purpose processor embedded in a sensor system. In the second case, it serves as an accelerator, performing matrix-based and other "locally recursive" computations involved in large scale simulations, under the control of a conventional general-purpose computer. For certain applications, the resulting throughput is potentially that of a supercomputer at a fraction of the cost.

* Present address: Department of Mathematical Sciences (or Department of Computer Science), University of Nevada, Las Vegas, NV 89154, United States.
An array processor has two modes of operation: cellular and systolic. In the cellular mode, the input data are first loaded into the array, then processed in unison, the results are then unloaded from the array, and a similar cycle starts anew. In the systolic mode of operation, the loading, computing and unloading occur concurrently. In this mode every processor regularly pumps data in and out, each time performing a short computation, with the data rhythmically flowing through the network.

Due to the importance of linear-algebraic methods in scientific computing, matrix problems have become preferred benchmarks for high-performance computers [18,19]. A survey of parallel algorithms for dense linear-algebraic computation was recently presented in [25]. A comprehensive review of iterative methods has been presented in [67]. Both surveys concentrate on commercially available computers, consisting of a modest number of processors, with shared-memory multivector units like the CRAY-2 or distributed memory like the hypercube. Surprisingly, the surveys hardly consider massively parallel-array processors, which in fact are ideally suited for matrix computation.

Linear-algebraic operations have properties of locality, recursiveness and regularity that match well the fine-grain parallelism of array processors. When the dimension of the problem corresponds to the dimension of the array processor, a matrix operation can have very low overhead in communication and synchronization. Speiser and Whitehouse [81] have shown that computational requirements for many real-time signal processing tasks (such as adaptive filtering, beamforming, cross-ambiguity calculations, data compression, etc.) may be reduced to a common set of linear-algebraic operations. It is no surprise, therefore, that there is considerable research activity on matrix-based cellular and systolic computation.

Our goal is to portray a profile of such activity by describing recent algorithms, both systolic and cellular. The outline of the paper is as follows. Section 2 illustrates a systolization process by matching an arbitrary gaxpy operation onto a fixed-size square array processor. The next section describes two systolic methods, one iterative and the other direct, for solving systems of linear equations. Section 4 derives a cellular algorithm for a fast Fourier transform, based on a new implementation on rectangular-array processors of the perfect-shuffle permutation. Finally, Section 5 gives snapshots of active research areas, using annotated lists of recent references on systolic linear solvers, the singular value decomposition, artificial neural networks and the simulated annealing algorithm. Our notation is standard with $\mathbb{R}^n$ denoting Euclidean n-space and $\mathbb{R}^{m \times n}$ representing the space of real $m \times n$ matrices. The subscript t denotes transposition of a matrix.

For the benefit of the reader not familiar with array processors, we briefly describe an illustrative architecture, namely, a prototype systolic/cellular processor [65,71] built at Hughes Research Laboratories. As shown in Fig. 1, the system consists of a host processor, a back-end array processor coupled with a corresponding array memory, and a controller with related program memory. The host coordinates such tasks as loading data into the array processor, initiating the execution of programs, and unloading results from the array processor. The array processor consists of 256 processing cells configured as a 16 X 16 square mesh. The cells operate in SIMD mode. A masking capability allows turning off a given subset of cells during execution. Each interior cell is connected to its four nearest neighbors via two horizontal and two vertical connections. There is also a wraparound connection between the two ends of each row. Each processing cell possesses a local memory and multiple functional units for computation. Boundary cells on one side can optionally have a hardwired square root in order to accelerate certain
operations, for example, Givens rotations in the solution of linear systems [62]. The program memory is separate from the data memory. The latter is partitioned into 16 columns, each affiliated with a corresponding column of the array processor. This memory has two independent ports, one of which is used to load data into the array processor, the other to store results back into memory.

The prototype demonstrated the viability of the hardware design. Evolutionary models, currently under study, enhance miscellaneous capabilities, and increase the size of the array processor while preserving its square geometry and connectivity. A variety of algorithms have been mapped onto such architectures by Hughes researchers [55–57,61–64,70,72,73,77].

Table 1 shows comparative peak performance parameters for a CRAY supercomputer with 4 processors and two systolic array processors of different sizes.

The processors respectively have a clock rate of 105, 8, 16 and a memory cycle of 26, 16.7, 26 all measured in MHz. In both array processors, the word length is 32 bits and each cell has two multipliers, two adders, a divider, and a selector, four of which can operate concurrently. The $16 \times 16$ model uses fixed-point arithmetic. The memory Bandwidth, MIPS and MFLOPS parameters are compared to those of the CRAY-1S normalized to 1. The benchmark is the time in milliseconds to solve a $100 \times 100$ system of linear equations. The CRAY uses FORTRAN routines DGEFA and DGESL from LINPACK, with Rolled BLAS calls, compiled with the CFT.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Bandwidth</th>
<th>MIPS</th>
<th>MFLOPS</th>
<th>Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRAY X-MP-4</td>
<td>32</td>
<td>5.3</td>
<td>5.3</td>
<td>14</td>
</tr>
<tr>
<td>$16 \times 16$ array</td>
<td>0.13</td>
<td>14.6</td>
<td>~</td>
<td>31</td>
</tr>
<tr>
<td>$128 \times 128$ array</td>
<td>3.2</td>
<td>1872</td>
<td>4369</td>
<td>3</td>
</tr>
</tbody>
</table>
vectorizing compiler [18]. The two array processors use an assembly-coded version of the systolized partitioned Faddeev algorithm [62] described in the sequel.

2. Systolic gaxpy computation

Given an algorithm satisfying prerequisite properties, the process of deriving a model of a systolic array for executing the algorithm is called a mapping. Diverse methods exist for mapping algorithms onto systolic arrays. Capello an Steiglitz use geometric [9] and linear space-time transformations [10]. Kung [37] proposes a technique, which we use in part in the sequel, based on data dependence and signal flow graphs. Leiserson [47] presents a scheme for minimizing the number of delay elements. Moldovan [58] uses a formalism based on transformation of index sets and data dependence vectors. Navarro et al. [66] study partitioning of matrix operations onto fixed-size array processors. Quinton [74] produces a method for algorithms that can be expressed as a set of uniform recurrence relations. Other contributions have been made.

Although the solutions to mapping problems promote conceptual understanding of time-space issues at hand, the process by itself is only of partial use in practical situations that require the extraction of systolic algorithms for a given specific architecture. The inverse of the mapping problem, in which one starts with an array processor and wishes to get systolic algorithms for it, is called the constrained mapping or matching problem. The former term denotes the process of fitting algorithms under constraints of array topology, number of cells, global and local memory sizes, etc. A special case of the matching problem occurs when the size of the dependence graph of the algorithm is bigger than the size of the array processor. This is the so-called partitioning problem.

In this section, in order to illustrate how systolic algorithms are derived, we shall match a simple linear-algebraic operation onto a fixed-size square array processor of the type described in Section 1. Specifically, our aim is to match an affine operation

$$z = Ax + y, \quad A \in \mathbb{R}^{m \times p}, \quad x \in \mathbb{R}^p, \quad y \in \mathbb{R}^m,$$

onto a square $N \times N$ systolic array, of fixed size $N$, with square mesh connectivity, whose primitive operation in each cell is a scalar multiply-add

$$\alpha \beta + \gamma.$$

In effect, we shall specify the equivalent of a BLAS2 primitive [25, Section 3] for the targeted architecture. The affine operation (1) is also called a gaxpy operation, a name that has its origin in the LINPACK software package. One can think of the term as a mnemonic for “general $Ax$ plus $y$”. This operation, which may be viewed as an extension of the scalar operation (2), is in fact computed as a systematic combination of its scalar counterpart.

Our matching procedure proceeds in three steps.

1. The operation is mapped onto a linear array with $N$ cells when $p = N$.
2. It is then partitioned onto the same linear array when $p > N$.
3. The partitioning is applied on each row of the $N \times N$ array.

The linear array with $N$ cells in steps (1) and (2) may be viewed as one row of the square array.
2.1. Mapping

Assume that \( p = N \). An algorithm for operation (1) can be expressed as follows:

For \( i = 1, 2, \ldots, m \)

For \( j = 1, 2, \ldots, N \)

\[
x(i, j) = x(i - 1, j)
\]

\[
z(i, j) = a(i, j) \cdot x(i, j) + z(i, j - 1)
\]

End

End

The inputs are \( a(i, j) = a_{i,j} \), \( x(0, j) = x_j \), \( z(i, 0) = z_i \) and the outputs are \( z(i, N) = z_i \). The right-hand side of the fourth statement is a primitive operation. With a view of systolizing the procedure, the algorithm is put in single assignment form (each indexed variable is assigned a single value) and transmittent variables \( x(i, j) \) and \( z(i, j) \) are used to localize all data dependencies. These transmittent variables will allow us to distribute the \( x_j \) and \( z_j \) values without broadcasting.

In order to systolize the algorithm, we apply the so-called canonical mapping methodology \([37, \text{Section 3.3]}\). In this approach, three structures are successively developed:

1. a dependence graph (DG) which shows the data dependencies in the algorithm;
2. a signal flow graph (SFG), which serves as an abstract model of the processor array, obtained by operating on the DG;
3. and lastly, a systolic array design obtained by incorporating temporal considerations in the SFG.

We briefly describe each of these structures.

The DG is a directed graph whose nodes are elements of the index space

\[
\{ \bar{k} = (i, j) : 1 \leq i \leq m, 1 \leq j \leq N \}
\]

and whose edges denote data dependencies. An index point has incoming edges from all points it is directly dependent on. The DG for the affine operation with \( m = 4, N = 3 \) is shown in Fig. 2. The DG satisfies two prerequisites for the use of the canonical mapping methodology.

1. The lengths of edges are independent of the problem size \( m, N \). This property of the DG in fact formally defines a locally recursive algorithm.
2. The DG is shift invariant in the sense that for index vectors \( \bar{k}_1, \bar{k}_2, \bar{k} \) if a variable at \( \bar{k}_1 \) depends on a variable at \( \bar{k}_1 - \bar{k} \), then a variable at \( \bar{k}_2 \) depends on a variable at \( \bar{k}_2 - \bar{k} \). Inputs and outputs at the border nodes are exempted from this requirement.

A graph with these properties represents a locally repetitive and homogeneous algorithm, well-suited for systematic systolization.

An SFG is also a directed graph. Nodes represent primitive operations and each edge denotes either a dependence relation or a time delay operator, in which case that edge is labelled with a capital letter D. In the SFG, primitive operations are assumed to require no execution time. Consequently, the derivation of time-space relations associated with pipelining are postponed to the third stage of the methodology. The SFG is obtained by operating on the DG. The nodes of the index space are assigned to nodes of the SFG and the order in which these nodes are to be executed is scheduled. These two steps proceed like this:

1. A projection map \( \pi \) defined on the index space is used to assign nodes of the DG onto nodes of the SFG. For the affine operation under consideration, we take

\[
\pi(i, j) = j,
\]
namely, all the nodes along each vertical line of the DG are projected onto a single node of the SFG. See Fig. 3(a).

(2) An integer-valued map $\sigma$ defined on the index space is used to specify the order in which the nodes of the DG are to be executed. In our case,

$$\sigma(i, j) = i + j - 1.$$ 

Nodes mapped onto the same integer, which fall on a line in the index space called an *equitemporal hyperplane*, are to be executed at the same time. The set of integer values of $\sigma$ thus corresponds to a set of parallel hyperplanes in the index space. See Fig. 3(b).
The map $\sigma$, which can be written in the form $\sigma(k) = \bar{n}k^t - 1$ where $\bar{n} = (1, 1)$ is orthogonal to the hyperplanes, is sometimes called a linear schedule.

The third stage of the canonical mapping methodology is to transform the SFG into a systolic array. Whereas the SFG is always spatially localized, it may not be temporally localized because some of its edges contain no delay. Technically, a systolic array differs from an SFG only in that it has a positive number of delays on every edge. Put in equivalent terms, a systolic array may be considered as an SFG combined with retiming and pipelining. Formal techniques can be used to perform this transformation. However, as illustrated in Fig. 4, the passage from the SFG to the systolic array is straightforward for an affine operation. The result is a one-dimensional array with $N$ nodes each performing a multiply-add as a primitive operation. A cycle is the time span for the SIMD execution of a primitive operation.

2.2. Partitioning

Two schemes are commonly used to map a DG onto an SFG, namely, the locally sequential globally parallel (LSGP) method and the locally parallel globally sequential (LPGS) method [37, Section 6.3.21]. In both of these methods, the DG is partitioned into blocks. In the LPGS method, block size is chosen to match array size, the nodes within one block are processed by the array in parallel, and the blocks are thus processed one block at a time in sequential fashion. We use this method to partition large-scale affine operations onto a fixed-size linear array.

Consider again the operation (1), but where this time $p > N$. The goal is to match the procedure derived earlier onto the same one-dimensional systolic array with $N$ cells. With no loss of generality, let $p$ be a multiple of $N$, say, $p = \rho N$. (If needs be, an appropriate number of zero columns and zero elements are appended to the matrix $A$ and the vector $b$.) The DG is partitioned into $\rho$ blocks and appropriate equitemporal hyperplanes are chosen to represent the schedule onto the one-dimensional SFG. The partitioning process with $m = 4$, $N = 3$, $p = 6$ is illustrated in Fig. 5. The matrix $A$ is partitioned into $\rho = 2$ strips and a FIFO buffer is used to circulate intermediate sums back to the horizontal input pipeline.

The data flow in the general case is shown in Fig. 6. The matrix $A \in \mathbb{R}^{m \times pN}$ and the vector $x \in \mathbb{R}^{pN}$ are partitioned into $\rho$ strips $A_k \in \mathbb{R}^{m \times N}$ and $\rho$ vectors $\bar{x}_k \in \mathbb{R}^N$, respectively. The $i$th row of $A$ is $[A_{1i}, A_{2i}, \cdots, A_{mi}]$, where $A_{ki}$ is the $i$th row of $A_k$. The $i$th element of the output vector $d$ is given by

$$d_i = y_i + \sum_{k=1}^{\rho} \left( \sum_{j=1}^{N} a_{i,(k-1)N+j} \bar{x}_{(k-1)N+j} \right) = y_i + \sum_{k=1}^{\rho} t_k,$$
where \( t_k = A_{ki} \overline{x}_k \). For \( \lambda < \rho \), each intermediate sum \( y_i + \sum_{k=1}^{\lambda} t_k \) is routed back by the FIFO for horizontal pipelining. This occurs as soon as the datum \( a_{i,kN} \) (the last element of \( A_{ki} \)) has been processed by the \( N \)th cell, namely, after cycle \((k-1)m + i + N - 1\). If \( \lambda = \rho \), the sum is instead pipelined out. The \( j \)th cell is loaded with \( \overline{x}_{kj} = x_{(k-1)N+j} \) (the \( j \)th element of \( \overline{x}_k \)) between cycles \( k = (k-1)m + j - 1 \) and \( k + 1 \). It takes a total of \( m\rho + N - 1 \) cycles for the output vector \( z \) to come out.

2.3. Matching

The operation (1), where \( m = \mu N \) and \( p = \rho N \) for nonnegative integers \( \mu \) and \( \rho \), is now easily matched onto the \( N \times N \) systolic array. Partition \( A \) into \( N \) strips \( A_j \in \mathbb{R}^{\mu \times p} \) and partition \( y, z \) each into \( N \) vectors \( \overline{y}_j, \overline{z}_j \in \mathbb{R}^\mu \), as shown in Fig. 7. Each affine operation

\[
\overline{z}_j = A_j x + \overline{y}_j, \quad 1 < j < N,
\]

\[
[ z ]_{mx1} = \left[ A_1 | A_2 | \ldots | A_p \right]_{mxp} \cdot [ y ]_{mx1},
\]

Fig. 6. Partitioning for a gaxpy operation on a linear systolic array.
is partitioned as in Section 2.2 on a row of the square array. The $N$ rows operate in parallel. Each row has its own vertical pipeline and there is no data transfer between rows. The process requires $T = p + N - 1$ cycles to produce the complete output. Ignoring costs of data movement, idealized speedup is $\mu N^2 / T$ compared to serial computation.

In the special case $m = p = pN$, a method for transforming a linear systolic array into a rectangular $\mu \times N$ systolic array has recently been given [49]. However, this is not a matching procedure, since the dimensions of the resulting array are problem dependent.

### 3. Systolic linear solvers

In this section, we describe two systolic algorithms for solving systems of linear equations. The first algorithm consists of an iterative method based on the systolized gaxpy operation described earlier. The second algorithm uses a modification of the Faddeev procedure [24,62].

#### 3.1. Iterative method

Given a matrix $L = J - K \in \mathbb{R}^{n \times n}$, where $J$ is invertible, and a vector $v \in \mathbb{R}^n$, consider a system $Lu = v$. If $M = J^{-1}K$ has a spectral radius less than one, then $L$ is invertible and, for any $x_0$, the Neumann iterates

$$x_r = M x_{r-1} + w, \quad w = J^{-1}v,$$

converge to $u = L^{-1}v$. Moreover, if $\| J^{-1} \| \| K \| \leq \beta < 1$, then

$$\| u - x_r \| \leq \frac{\beta'}{1 - \beta} \| x_r - x_0 \|.$$

When the matrix $M$ is dense and large, the iteration (3) is rarely used because it is costly relative to its linear convergence.

A little-known modification due to [50], which yields quadratically convergent iterates, is advantageously implementable on square systolic arrays. The Hotelling–Lonseth iterates $y$, are
defined by the double recurrence

\[ y_0 = x_0, \quad z_0 = w, \]
\[ y_r = M^{2^{r-1}} y_{r-1} + z_{r-1}, \quad z_r = M^{2^{r-1}} z_{r-1} + z_{r-1}. \]  \( 4 \)

It turns out that \( y_r = x_{2^r} \). If \( y_0 = w \), then \( z_r = y_r \) and the iterates \( y_r \) are obtained from a single recurrence relation.

After initialization of \( M \) and \( y \), the first \( R \) iterates \( 4 \) are obtained as follows:

For \( r = 1, 2, \ldots, R \)

\[ y_r = M y_{r-1} + z_{r-1}, \]
\[ z_r = M z_{r-1} + z_{r-1}, \]
\[ M_r = M M_{r-1} \]

End

When \( r = R \), the second gaxpy and the matrix multiplication need not be done. Hence, the procedure takes \( (2R - 1)T_G + (R - 1)T_M \) cycles for arbitrary \( y_0 \) and \( RT_G + (R - 1)T_M \) cycles if \( y_0 = w \), where \( T_G, T_M \) are respectively the cycle counts for a gaxpy operation and a matrix-matrix multiplication.

Assume that the algorithm is executed on an \( N \times N \) square mesh array processor and that \( n = vN \), where \( v \) is a positive integer. From Section 2.3, we find that \( T_G = v^2 + N - 1 \). Each column of the matrix product \( MM \) can be computed as a gaxpy operation partitioned as described in Section 2.2 on a single row of the array processor, with the \( N \) rows working in unison. We thus get that \( T_M = (v^3 + 2v)N - 2v \), referring to [57] for details.

The Neumann iteration \( 3 \) for computing \( y_r \) on the same systolic array takes \( (2^k - 1)T_G \) cycles. Thus the general Hotelling–Lonseth iteration has smaller arithmetic complexity on the systolic array only if the number of iterates \( R \) satisfies

\[ 2^k > \xi R - \xi + 1, \quad \xi = \frac{T_M}{T_G}. \]

For example, on a 16 \( \times \) 16 systolic array, this is the case for a 128 \( \times \) 128 linear system if \( R \geq 10 \), whereas on a 128 \( \times \) 128 systolic array it is so for any \( R \geq 1 \).

We emphasize that the cycle counts refer only to numbers of scalar multiply-add operations executed in SIMD mode. For simplicity, we have ignored the cost of data management. In practice, however, data formatting and partitioning become increasingly costly when the dimension of the problem gets larger than the dimension of the systolic array.

### 3.2. Modified Faddeev method

Consider four matrices \( A, B, C, D \) arranged for convenience as shown in the following tableau:

\[
\begin{array}{ccc}
  n & q \\
  A & B & m \\
  -C & D & p \\
\end{array}
\]

in which the letters \( m, n, p, q \) denote dimensions of the matrices. Suppose that \( m = n \) and that
$A$ is invertible. The Faddeev algorithm [24] uses Gaussian elimination to annul the lower left-hand quadrant of the tableau. Namely, multiples of rows of the upper part $[A, B]$ are systematically added to rows of the lower part $[-C, D]$ until $-C$ is replaced by the $p \times n$ zero matrix. Assuming that there is no zero pivot, the Schur complement

$$CA^{-1}B + D$$

will then replace $D$ in the lower right-hand quadrant. Note that a matrix multiply-add can be executed by taking $A = I$ and that a gaxpy operation is obtained by further taking $q = 1$. The case with $C = I, D = O$ solves the matrix equation $AX = B$.

In contrast to the LU or QR factorization method, the Faddeev algorithm avoids the backsubstitution step used in solving an upper triangular system. This feature greatly facilitates mapping of the algorithm onto an array processor. On the other hand, partial or full pivoting, needed to avoid zero pivots and to control roundoff, is costly to implement on an array processor with nearest-neighbor connectivity. Furthermore, the direct application of orthogonal transformations to annul $-C$ results in a modification of the Schur complement.

In order to circumvent these problems, Nash and Hansen [62] proposed a modification of the Faddeev algorithm based on both orthogonal transformation and Gaussian elimination. Consider again the tableau, this time with $m > n$, and assume that $A$ has rank $n$. The modified algorithm proceeds in two stages.

**Stage 1.** The matrix $A$ is triangularized using Givens rotations to get

$$\begin{bmatrix} R & \mathbf{Q}_1 \mathbf{Q}_2 \end{bmatrix} B$$

where $Q = [Q_1, Q_2]$ is the orthogonal factor in the QR factorization of $A$.

**Stage 2.** Gaussian elimination is then applied to annul $-C$, using the diagonal elements of $R$ as pivots, to get

$$\begin{bmatrix} R & \mathbf{Q}_1 \mathbf{Q}_2 \end{bmatrix} B$$

Since $A$ is assumed to have full rank, the diagonal elements of $R$ are nonzero, and thus division by zero in the second stage is theoretically impossible. For any one column $b$ of $B$, the vector $R^{-1}Q_1 b$ is the least-squares solution to the overdetermined system $Ax = b$.

We see that the modified Faddeev algorithm consists of an orthogonal transformation of $A$ extended to $B$, followed by Gaussian elimination of $-C$ with corresponding row operations extended to $D$. Rearrangements of the tableau provide for the solution of underdetermined and generalized least-squares problems [62].
From the point of view of algorithmic mapping, the main advantage of the modified Faddeev algorithm is that both its stages can be mapped onto the same systolic array. As illustrated in Fig. 8(a), this array has trapezoidal shape and mesh connectivity. The circles and squares denote respectively boundary and internal processing cells. Cell specifications for the two stages are indicated in Figs. 8(b) and 8(c), respectively. In Stage 1, the boundary cells compute sines and cosines of Givens rotations while internal cells perform the rotations. In Stage 2, boundary cells...
compute multipliers while internal cells perform the eliminations. In the Givens rotation stage, a skewed alignment of the matrices $A$ and $B$ is pipelined vertically through the array processor, with $A$ and $B$ respectively passing through the triangular and rectangular part of the trapezoid. Simultaneously, sines and cosines generated by the boundary processors are pipelined horizontally through the entire array. Data movement during the Gaussian elimination stage is analogous, with the matrices $C$ and $D$ pipelined vertically and the multipliers pipelined horizontally.

The trapezoidal systolic array of Fig. 8(a) is readily transformed into the square structure shown in Fig. 9. These figures illustrate the case when $A$ and $B$ are both $4 \times 4$ matrices. In the square-array processor, the data flows south until it reaches the main diagonal, then it moves west to a certain subdiagonal, determined by the size of the problem, and then moves south again to exit the array. Computations occur in the horizontal phase of the flow, while the processors on the vertical path provide time delays.

Partitioning techniques exist when the problem size exceeds that of the array processor. One such technique consists of dividing the block matrices $[A, B]$ and $[-C, D]$ into vertical strips of width equal to that of the array. Each strip is further divided into blocks matching the size of the array. The processing proceeds in several steps, with each step operating on all corresponding blocks of the strips. In any one step, the boundary cells generate their elements using data in the left-most strip, and during the remainder of the step, the internal cells use these elements for rotations or eliminations on the remaining data. We refer to [62] for details.

4. Cellular fast Fourier transform

The algorithms that we have described so far are of systolic type. During their execution, the cells in the array processor regularly pump data in and out, each time performing simultaneously a short computation, with the data systematically pipelined through the network. This mode of
operation requires that the algorithms be homogeneous and locally repetitive, as exemplified by
the linear-algebraic operations that we considered.

In the cellular mode, the input data are first loaded into the array, then processed in unison
using as many steps as needed, including possible intercellular data transfers, and the results are
then unloaded from the array. There is no orderly pipelining as in the systolic mode, and
consequently, computation and data movement usually cannot be balanced for optimal concu-
rency. Hence, the cellular mode is generally not as efficient as the systolic mode. However,
procedures that cannot be systolized must then be mapped in cellular mode. Fast Fourier
transforms (FFTs), which have an inherently global data dependency that hinders local com-
unication, are among such procedures.

We illustrate the cellular mode by describing a recent FFT algorithm for rectangular array
processors [56]. The FFT is said to have constant geometry because the communication pattern
shown in its dependence graph is periodic, namely, the addressing of operands for the arithmetic
operations is the same from stage to stage. Such FFTs were originally derived by Pease [69],
using matrix factorizations to modify and parallelize the usual Cooley–Tukey procedure. For the
N-point radix 2 case, the algorithm consists of \(\log_2 N\) stages each preceded by a perfect shuffle of
the data. The most natural mapping of this algorithm is onto a linear-array architecture with \(\frac{1}{2}N\)
cells and a shuffle-exchange interconnection network [82,84].

Our strategy is to further decompose the Pease factorization in order to map a \(2MN\)s-point
radix 2 FFT onto an \(M \times N\) rectangular array processor. The result depends fundamentally on a
factorization of the perfect-shuffle permutation. The corresponding data movement is realized in
parallel as relatively small perfect shuffles inside each local memory and along each row and
column of the array processor, without requiring that the complete array itself have the
shuffle-exchange interconnection network. While matrix operations were heretofore the objects
of our algorithms, in this section we additionally use linear algebra to validate the actual
mapping of the algorithm to the architecture.

4.1. Matrix factorizations

A perfect shuffle is a permutation that transforms the 2m-vector

\[
    z = (0, 1, \ldots, m-1, m, m+1, \ldots, 2m-1)^T
\]

to the vector

\[
    S_{2m}z = (0, m, 1, m+1, \ldots, i, m+i, \ldots, m-1, 2m-1)^T. \tag{5}
\]

Components that were \(m\) apart become adjacent as a result of the perfect shuffle. For
convenience, we simply call (5) the shuffle of \(z\).

A standard result in algebra provides a constructive method for factoring any permutation as
a product of 2-cycles, e.g., [30, p.78]. Moreover, any 2-cycle is itself a product of 2-cycles
involving only adjacent components. Hence, letting \((k)\) denote the exchange of the \(k\)th and
\((k+1)\)th components, one can show that

\[
    S_{2m} = T_{m-1} \cdots T_2 T_1, \tag{6}
\]

where \(T_i\) is a product of \(i\) adjacent exchanges:

\[
    T_i = (m-i)(m-i+2) \cdots (m+i-2).
\]
The case for \( m = 4 \) is shown below:

\[
\begin{align*}
T_1 &= (3); & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
T_2 &= (2)(4); & 0 & 1 & 2 & 4 & 3 & 5 & 6 & 7 \\
T_3 &= (1)(3)(5); & 0 & 1 & 4 & 2 & 5 & 3 & 6 & 7 \\
S_8 &= 0 & 4 & 1 & 5 & 2 & 6 & 3 & 7
\end{align*}
\]

Since the \( i \) exchanges in \( T_i \) are independent of one another, they can be done in parallel. A consequence is that the factorization (6) can be implemented on an array processor with nearest-neighbor connectivity in \( m - 1 \) steps, with the \( j \)th step consisting of the parallel execution of the \( i \) exchanges in \( T_i \).

The following factorization of the shuffle permutation has a basic role in our presentation.

**Lemma 1.** \( S_{2KL} = (S_{2L} \otimes I_K)(I_L \otimes S_{2K}) \).

**Proof.** We show that the effect of the factorization on the vector \( z = (0, 1, \ldots, 2KL - 1) \) is the vector

\[(0, KL, 1, KL + 1, \ldots, j, KL + j, \ldots, KL - 1, 2KL - 1)^t.\] (7)

Partition \( z \) into \( 2K \) \( L \)-tuples,

\[z = (\tilde{z}_0, \tilde{z}_1, \ldots, \tilde{z}_{2K-1})^t,
\]

where

\[\tilde{z}_i = (iL, iL + 1, \ldots, iL + j, \ldots, (i + 1)L - 1).\]

We find that

\[(I_L \otimes S_{2K})z = (\tilde{z}_0, \tilde{z}_K, \ldots, \tilde{z}_i, \tilde{z}_{K+i}, \ldots, \tilde{z}_{K-1}, \tilde{z}_{2K-1})^t.\]

The effect of applying \( S_{2L} \otimes I_K \) on this vector is that of replacing the \( 2L \)-tuple \((\tilde{z}_i, \tilde{z}_{K+i})\) by

\[S_{2L}(\tilde{z}_i, \tilde{z}_{K+i})^t = S_{2L}(iL, \ldots, iL + j, \ldots, (i + 1)L - 1, (K + i)L, \ldots, (K + i)L + j, \ldots, (K + i + 1)L - 1)^t = (iL, iL + KL, \ldots, iL + j, iL + j + KL, \ldots, (i + 1)L - 1, (i + 1)L - 1 + KL)^t.\]

The concatenation for \( i = 0, 1, \ldots, K - 1 \) of these \( 2L \)-tuples yields the desired vector (7). \( \square \)

The factorization in the lemma is analogous to cutting a deck of \( 2KL \) cards into a row of \( 2K \) subdecks of \( L \) cards each, rearranging these subdecks according to the shuffle permutation, shuffling together adjacent pairs of subdecks, and then collecting the resulting \( K \) subdecks into a single deck.

The *discrete Fourier transform* of a complex \( P \)-vector \( z = (z_0, z_1, \ldots, z_{P-1})^t \) is \( Fz \), where \( F \) is a complex \( P \times P \) matrix whose \((\lambda, \mu)\)th element is \( \omega^{\lambda \mu} \), \( 0 \leq \lambda, \mu \leq P - 1 \), where \( \omega = e^{-i2\pi/P} \) and \( i^2 = -1 \). The *Pease factorization* \([69]\) of the matrix \( F \) can be expressed as

\[UF = (A^{(P-1)}S_p) \cdots (A^{(1)}S_p)(A^{(0)}S_p),\] (8)
The \( k \)th stage of this FFT is represented by \( A^{(k)} S_{p} \), where \( k = 0, 1, \ldots, p - 1 \) and \( p = \log_2 P \). The matrix \( U \) is the permutation matrix that represents the bit-reversed order of the output vector. Each \( 2 \times 2 \) matrix \( B(m) \), applied on a pair of adjacent data, is called a butterfly operation. The element \( \omega^m \) is called a twiddle factor. For our purposes here, we need not define the exponents \( \epsilon^{(k)}_\mu \). The usual parallel implementation of (8) is on a linear array processor with \( \frac{1}{2} P \) cells, with each cell containing a pair of adjacent data. Each stage of the FFT begins with a perfect shuffle, involving data transfers among the cells, followed by a SIMD butterfly operation.

The following theorem, which further decomposes the Pease factorization, engenders a parallel FFT algorithm for an \( M \times N \) rectangular array processor.

**Theorem 2.** Let \( P = 2MN\)s where \( M, N, s \) are powers of 2 with nonnegative integer exponents and let \( \rho = \log_2 P \). Then

\[
C_0UF = F^{(\rho - 1)} \cdots F^{(1)}F^{(0)}C_0z.
\]

where

\[
F^{(k)} = C_0A^{(k)}C_1C_2,
\]

\[
C_1 = C_0C_1C_0^t, \quad C_2 = C_0C_2C_0^t,
\]

\[
C_0 = \bigoplus_{j=0}^{M-1} S_{2Nj}, \quad \overline{C}_1 = \bigoplus_{m=0}^{M-1} (I_N \otimes S_{2x}), \quad \overline{C}_2 = I_{Ns} \otimes S_{2M}.
\]

and \( A^{(k)} \) is defined in (9).

**Proof.** Apply the lemma twice to get

\[
S_{2MNs} = (S_{2Nj} \otimes I_{M})\overline{C}_2, \quad S_{2Ns} = (S_{2N} \otimes I_{s})(I_{N} \otimes S_{2x}).
\]

Combine to get

\[
S_{p} = \bigoplus_{m=0}^{M-1} (S_{2N} \otimes I_{s})(I_{N} \otimes S_{2x})\overline{C}_2 = C_0C_1C_2.
\]

Using that \( C_1^1 = C_0^{-1} \), we find that

\[
S_{p} = C_1C_2C_0.
\]

The desired result follows. \( \square \)

**4.2. Basic algorithm**

Let \( P = 2MN\)s where \( M, N, s \) are powers of 2 with nonnegative integer exponents. Our goal is to describe a parallel FFT algorithm for a 2-dimensional \( M \times N \) array processor. We first specify needed features of the targeted architecture.

**Storage.** Each of the \( MN \) cells has a local memory with \( 2s \) locations, each able to contain a complex datum, denoted by \( R_0, R_1, \ldots, R_{2s-1} \). A complex \( P \)-vector \( z \) is stored so that pairs of
locations $R_{2i}$, $R_{2i+1}$ contain adjacent data $z_\mu$, $z_{\mu+1}$. The symbol $R_{i}^{(m,n)}$ denotes the content of the $i$th location in the $(m,n)$th cell of the array processor. The 2s-vector

$$R^{(m,n)} = \left( R_{0}^{(m,n)}, R_{1}^{(m,n)}, \ldots, R_{2s-1}^{(m,n)} \right)$$

represents the data in the local memory of the $(m,n)$th cell.

**Communication.** In addition to (15), consider the vectors

$$X_i^{(m)} = \left( R_{0}^{(m,0)}, R_{1}^{(m,0)}, \ldots, R_{2s-1}^{(m,0)} \right)$$

$$Y_i^{(n)} = \left( R_{i}^{(0,n)}, R_{i+1}^{(0,n)}, \ldots, R_{i+M-1}^{(0,n)} \right)$$

We assume that the array processor can execute the following three operations:

- rshuffle: replace $R^{(m,n)}$ by $S_{2s} R^{(m,n)}$ simultaneously for $m = 0, 1, \ldots, M - 1$ and $n = 0, 1, \ldots, N - 1$;
- xshuffle($i$): for fixed $i$, replace $X_i^{(m)}$ by $S_{2s} X_i^{(m)}$ simultaneously for $m = 0, 1, \ldots, M - 1$;
- yshuffle($i$): for fixed $i$, replace $Y_i^{(n)}$ by $S_{2s} Y_i^{(n)}$ simultaneously for $n = 0, 1, \ldots, N - 1$.

The rshuffle operation consists of $MN$ parallel shuffles of the $2s$ memory locations in each cell. This operation does not involve data transfers among cells, unlike the other two operations. The xshuffle($i$) operation consists of $M$ parallel shuffles, one in each row of the array, involving the pair of locations $R_{2i}$, $R_{2i+1}$ in each cell. The yshuffle($i$) operation consists of $N$ parallel shuffles, one in each column of the array, involving the pair of locations $R_{i}$, $R_{i+s}$ in each cell. The connectivity in these operations is illustrated in Fig. 10 for the case $M = N = s = 4$.

**Arithmetic.** For fixed $i$ and $k$, let arith($k$, $i$) denote the parallel butterfly operation that replaces $R_i = (R_{2i}, R_{2i+1})$ in each cell by $B(\lambda) R_i$, where $B(\lambda)$ is the $2 \times 2$ matrix defined in (9) and $\lambda$ is an appropriate exponent. The totality of $\frac{1}{2}P$ butterflies in the $k$th stage of the FFT is obtained by executing arith($k$, 0), arith($k$, 1), $\ldots$, arith($k$, $s - 1$).

The desired algorithm is a transliteration of the factorization (10). The correspondence between the contents of the local memories and a $P$-vector $z$ is given by the bijection

$$\Phi: z \to \{ R_k^{(m,n)} \} \equiv \mathcal{R},$$

$$\Phi = \phi(i, m, n) = 2N(i + ms) + 2n.$$
The effect of $C_0$ is to perform a shuffle on every $2N$-tuple in $z$ and the application of $\Phi$ after this permutation results in a natural ordering of the data in the local memories. Since both the input and output vectors in (10) are premultiplied by the matrix $C_0$, the mapping

$$\Psi : z \rightarrow R, \quad \Psi z = \Phi C_0 z,$$

$$R_{i}^{(m,n)} = z_{\psi}, \quad \psi(i, m, n) = (i + 2ms)N + n,$$

defines the correspondence between the $P$-vector and the local memories, respectively during the loading at the start and the unloading at the end of the FFT. During the FFT itself the correspondence is given by $\Phi$. The natural ordering induced by $\Psi$ facilitates loading and unloading and it is useful for bit-reversed sorting of the output.

It can be shown formally [56] that the operation $rshuffle$ carries out the permutation $C_1$, that the collective effect of the $s$ operations

$$yshuffle(0), yshuffle(1), \ldots, yshuffle(s - 1)$$

is that of the permutation $C_2$, and likewise, that the $s$ operations

$$xshuffle(0), xshuffle(1), \ldots, xshuffle(s - 1)$$

represent the permutation $C_0$.

Consequently the factorization (10) in Theorem 2 yields the following algorithm for executing a $P$-point FFT on the $M \times N$ array processor:

Load $\Psi z$

For $k = 0, 1, \ldots, \rho - 1$

For $i = 0, 1, \ldots, s - 1$

    $yshuffle(i)$

$rshuffle$

For $i = 0, 1, \ldots, s - 1$

    $arithmetic(k, i)$

For $i = 0, 1, \ldots, s - 1$

$xshuffle(i)$

Unload $\Psi^{-1} \overline{UFz}$

The For-$k$ loop represents the $\rho = \log_2 P$ stages of the FFT. The unload operation usually includes a bit-reversed sort to get $Fz$ in natural order. For the special cases $M = s = 1$ or $N = s = 1$, the above algorithm becomes the well-known Pease procedure for a linear array processor [69]. Thus, in its generality, the algorithm may be viewed as a partitioning of the Pease procedure for rectangular array processors.

4.3. Remarks

The communication complexity of the cellular algorithm for a $P$-point constant geometry FFT on a rectangular $M \times N$ array processor is dependent on the array's ability to perform the $rshuffle$, $xshuffle$, $yshuffle$ operations. In the ideal case, shuffle-exchange links in each local memory, row and column minimize the cost of data movement. In the case of an $N \times N$ array processor with square mesh connectivity, the shuffles become costly for large $N$. Thus, for an FFT with $P = 2N^2s$ points, using the factorization (6) of the perfect shuffle as a product of
adjacent exchanges, if we assume that North–South and East–West intercellular exchanges and internal register exchanges each have unit cost, we find that the communication complexity is
\[(2Ns - s - 1)(2 \log_2 N + \log_2 s + 1).\]
On the other hand, it is
\[6 \log_2 N + 3 \log_2 s + 3\]
in the idealized case for an \(N \times N\) array processor having maximally redundant shuffle-exchange links each with unit cost.

In addition to constant geometry FFTs, the perfect shuffle has wide applicability in parallel processing, including bitonic sorting, polynomial evaluation, matrix transposition, and linear transformations [5, 21, 82, 84]. Hence, implementations of the perfect shuffle engendered by matrix factorizations (14) for rectangular array processors, or extensions for higher-dimensional arrays, can be put to good usage for such applications. Moreover, assuming bidirectional links in the shuffle connections, we have access to the inverse perfect shuffle for the class of algorithms based on recursive doubling [36, 83]. In this case, a factorization corresponding to that in the lemma,
\[S_{2KL}^{-1} = (I_L \otimes S_{2K}^{-1})(S_{2L}^{-1} \otimes I_K),\]
serves as the basis for deriving representations of the inverse perfect shuffles on array processors.

The lemma and the theorem extend to radix \(r\) constant geometry FFTs for \(d\)-dimensional array processors [56]. The matrix factorization corresponding to an FFT is then obtained by \(d\) applications of a basic factorization of the radix \(r\) shuffle permutation extending that of the lemma for the radix 2 case. The resulting data movement involves parallel radix \(r\) shuffles in each local memory and along each of the \(d\)-dimensional axes of the array processor. The data is loaded in natural order and each cell performs a butterfly by operating on \(r\) consecutive registers \(R_i, R_{i+1}, \ldots, R_{i+r-1}\).

5. Active research areas

Due to space constraints, we are unable to describe numerous aspects of systolic and cellular computation. Our goal in this section is to provide snapshots, using annotated lists of recent references, of current research activities on the topic.

5.1. Systems of linear equations

The Gauss–Jordan method for matrix inversion, when compared to triangular matrix factorization techniques, performs poorly on serial computers. However, with no row or column interchange, it parallelizes remarkably well on array processors. Moreover, the method then embodies abstract recurrence relations solving the so-called algebraic-path problem, which encompasses as special cases, in addition to matrix inversion, the transitive closure and shortest-path problems in graph theory and regularization of languages in automata theory, see [28]. Descriptions of systolic arrays for the solution of the algebraic-path problem can be found in [48]. Such systolization illustrates nicely the symbiosis between mathematical structure and, at the conceptual level, array architecture.
Systolic arrays for LU and QR factorizations of matrices have been known since the early 1980s, see [26,46]. Partial or full pivoting, needed to avoid zero pivots and to control roundoff, is not readily amenable to parallel computation on array processors with nearest-neighbor connectivity. Gentleman and Kung [26] used so-called neighbor pivoting, in which the pivot is selected as the largest element among neighbors, to devise a systolic array for matrix triangularization. Roychowdhury and Kailath [76] have presented a systolic array for LU factorization with partial pivoting, with the property that the rows of the resulting upper triangular factor are not naturally ordered, thus necessitating considerable overhead in the determination of the solution vector. Cholesky factorization $LL'$, via the use of hyperbolic transformations, has been mapped onto a systolic array in [15].

The systolized version of the modified Faddeev method, described in Section 3.2, dates back to the mid 1980s, see [61,62]. The Faddeev algorithm was also systolized in [12] for both fixed-size and variable-size problems, and in [14] for partitioned implementation on two-dimensional arrays of transputers. Moreno and Lang [59,60] used their mapping technique, the so-called multimesh graph method, to partition the Nash–Hansen procedure onto both 1-dimensional and 2-dimensional systolic arrays. With the hypothesis that there be no data duplication, it is widely believed, though apparently no one has proved it, that optimal complexity for inversion of a dense $n \times n$ matrix on a systolic array is $5n$ inner product steps. The Nash–Hansen version of the Faddeev method and other systolic matrix inverters reach this bound. Megson [54] has recently proposed a Faddeev array that achieves matrix inversion in just $4n$ steps with $O(n^2)$ basic cells using careful duplication of some data.

Deprettere and Jainandunsing [17,31–33] have analyzed a class of systolic algorithms, called feed forward methods, for solving nonsingular systems. The solution of $Ax = b$ is obtained through a combination of an LU, LQ or $LL'$ factorization of $A$ and, if $L$ denotes the respective lower triangular factor, an updating or downdating of the Cholesky factorization of $LL' + bb'$ or $LL' - bb'$, respectively, or an LU factorization of $[L, -b']$. Like the Faddeev algorithm, the feed forward methods avoid backsubstitution and they can be generalized to compute Schur complements.

5.2. Singular-value decomposition

The singular-value decomposition (SVD) of a matrix provides the most efficient computational method for the determination of the rank of a matrix, it yields the best approximation in a least-squares sense of a high-dimensional matrix by a lower-dimensional one, and it provides the most numerically stable solution to least-squares estimation problems, see [27].

Because of such properties, there has been considerable interest on the use of SVD techniques in digital image and signal processing. For example, it has been known for a long time that the SVD is useful in image enhancement, moving target tracking, harmonic retrieval problems and image reconstruction. Refer respectively to [3,4,38,78]. Continuing research aims to find high-resolution algorithms for extracting signal and system parameters from measurements, using the SVD of appropriate data matrices as a means for robust separation of signal and noise, see [6,16].

For real-time or high-throughput applications, such as in avionics and space systems, systolic and cellular array processors provide a modern and attractive approach to computing the SVD of given matrices. Pertinent algorithms, for the SVD and related decompositions, have been
presented in [7, 8, 22, 23, 51–53, 85]. Comon and Golub [13] have recently analyzed various numerical methods for finding extreme singular values and corresponding left singular vectors when the matrix is slowly varying in time. Such algorithms are of crucial importance in signal-processing applications based on low-rank approximation of a time-dependent covariance matrix, see the references cited in [13].

5.3. Artificial neural networks

There is extensive ongoing research on neural networks and their representations on array processors. An introductory description of the subject can be found in [37, Section 4.6]. Recent texts on the topic are [20, 79, 80]. Applications in neural computing include combinatorial optimization, image and signal processing, artificial intelligence, etc. An early systolic architecture for an artificial neural network was designed in [39]. The same authors recently proposed [40] a unified architecture, consisting of a programmable ring systolic array, that provides for both the retrieving and the learning phases of a wide variety of artificial neural networks. A concise taxonomy of artificial neural networks is given in the introduction of that article.

Przytula et al. [70] have described graph-theoretic techniques for implementation of a wide class of neural networks of arbitrary size on mesh-connected SIMD arrays of fixed size. Shams and Przytula [77] used a multilayer perception network, implementable on 2-dimensional array processors, for underwater target detection. Other applications of artificial neural network in signal processing are described in references cited in [70]. A quick overview of other current activities on neural computing, e.g., on theory and algorithms, speech and image processing, and on implementation methods, can be gleaned from the proceedings [2] of a recent EURASIP workshop.

5.4. Simulated annealing algorithm

Simulated annealing, so called because of its conceptual analogy with metallurgical annealing, uses a stochastic approach to combinatorial optimization. The algorithm has diverse applications in job shop scheduling, circuit design, artificial intelligence, image restoration, etc. See, e.g., [29, 44, 45]. A general-purpose optimization routine, based on the annealing algorithm and written in Pascal, was published in [68]. A design for a simulated annealing array processor was presented in [37, Chapter 8], [41].

We make special mention of simulated annealing because the algorithm is closely related to Boltzmann machines, see [1], and because the latter topic provides a rigorous mathematical base from which to penetrate the much less rigorous, but quickly emerging, area of neural computing. A Boltzmann machine is a network of elements, each with state either 0 or 1, bidirectionally interconnected with strengths that can take arbitrary values. The aim is to maximize a consensus function $C$, defined as the sum of products of corresponding states and strengths. A transition unit is chosen for proposed changes in the machine. Such a change is accepted if either $C$ increases or if $1/(1 + e^{-\Delta C/s})$ is less than a uniformly distributed random number in $[0, 1]$. A decreasing sequence of $s$ values, called a cooling schedule, is used to bring the Boltzmann machine into near optimal configuration.

Since the appearance of papers by Kirkpatrick et al. [35] in 1982 and Cerny [11] in 1985, who initiated the research on the subject, the analytical framework behind simulated annealing has
considerably evolved. There is an extensive body of knowledge on global convergence properties of the simulated annealing algorithm. Since Boltzmann machines are analogous to simulated annealing, there are predictably corresponding results for Boltzmann machines. However, if in the definition of the Boltzmann machine one allows the choice of transition units to be implementable in parallel, asymptotic convergence for the resulting network appears to be still an open problem.

Acknowledgements

The author is grateful to colleagues at Hughes Research Laboratories, Greg Nash, Wojtek Przytula and David Schwartz, who provided much of the information contained in this paper.

References


