

# Simulation of Systolic Arrays on the Connection Machine

Nariankadu D. Hemkumar and Joseph R. Cavallaro  
Department of Electrical and Computer Engineering  
Rice University  
Houston, TX 77251-1892

*The use of a programming model which extends naturally from the underlying hardware, greatly eases the design and implementation of simulators, especially for those systems that resemble the hardware in the paradigm of computation. Given the characteristics of systolic arrays, SIMD computers which employ the data parallel programming model provide an ideal environment. In this paper, we present a systolic array simulator, a simulation tool written for the Connection Machine \*(model CM2), a SIMD machine with powerful interprocessor communication capabilities. Especially as recent advances have automated the design, there is a need for a verification environment to prototype systolic arrays. Primarily a simulation tool, the systolic array simulator also helps identify inefficiencies and motivates optimal design prior to implementation in either custom VLSI or DSP systems. Currently, we are updating the tool to allow the simulation of dynamic array reconfiguration algorithms under transient and permanent fault conditions. The simulator is also being ported to the CM5.*

**Key words** Systolic Arrays, Algorithm Verification, Design Optimization, Connection Machine.

\*Unless otherwise specified, all future references to the Connection Machine pertain to model CM2.

## 1. Introduction

Many definitions of systolic arrays exist in the literature (Gentleman and Kung, 1981; Ullman, 1984). In their paper on systolic arrays, Kung and Leiserson (Kung and Leiserson, 1980) define a systolic system as a "network of processors which rhythmically compute and pass data through the system." Systolic arrays as a class of pipelined array architectures display regular modular structures locally interconnected to allow a high degree of pipelining and synchronized multiprocessing capability (Kung, 1987). The primary reasons for the use of systolic arrays in special-purpose processing are simple and regular design, concurrency and communication, with balanced computation and I/O (Kung, 1982).

Due to the massive parallelism and data flow possible with locally interconnected computing networks, such as systolic and wavefront processor arrays, a large number of algorithms of practical significance in the area of signal processing and other engineering applications can be efficiently implemented. These architectures are capable of real-time solutions to a wide variety of advanced computational problems. The computations in systolic arrays are spread over the entire index set of processor elements (PEs).

Recent work has automated the design of systolic arrays (Moldovan, 1987; Rajopadhye and Fujimoto, 1990). The transformation of algorithms for parallel processing on processor arrays (Fortes and Moldovan, 1985; Rao, 1986) has further advanced the theory. In the

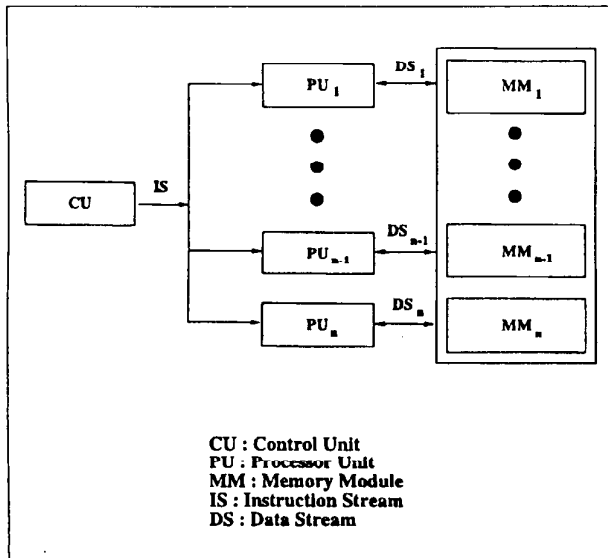


Figure 1. SIMD Architecture

previous decade, considerable research effort has been devoted to the realization of processor arrays and their optimal design. Between the realization of a systolic algorithm from a high-level problem description and its implementation using custom VLSI/WSI (Wafer Scale Integration) or DSP (Digital Signal Processing) chips, there is a need for a verification environment to prototype these arrays.

In this paper, we present a systolic array simulator written for the Connection Machine (Thinking Machines, 1990b) to allow a systolic array design to be tested for functionality. The systolic array simulator may also help in identifying performance bottlenecks and inefficiencies to motivate optimal design and implementation. Once the algorithm has been verified and optimized, a general simulation environment like the Rice Parallel Processing Testbed (Covington et al., 1991) may be employed to simulate arrays of simple instruction set processors (Dawkins, 1989).

## 2. The array model on the Connection Machine

The distinguishing features of systolic arrays map well onto the SIMD (Single Instruction Multiple Data) (Flynn, 1966) paradigm of computation. Although there are significant differences between systolic arrays and SIMD computers (Dew and Manning, 1986), the SIMD architecture (Figure 1) provides excellent hardware support for the simulation of systolic arrays. The Connection Machine, a SIMD computer with 64K processors and powerful inter-processor communication capabilities, was chosen for the simulation of these arrays.

The Connection Machine employs the data parallel model of computation. Each instruction is executed by all processors in parallel. However, each processor may be selectively activated or deactivated to allow variations in computations. Each processor on the Connection Machine has private memory. The Connection Machine communication primitives allow the transfer of data from one processor's memory to another. Parallel transfer of data in a regular manner is a very useful feature of the Connection Machine hardware.

The logical unit of simulation is the PE. Each PE is represented in hardware by a physical processor (equivalent to the PU of a generic SIMD machine) on the Connection Machine in a one-to-one mapping (Figure 2). All PEs in a systolic array are virtually identical though some systolic arrays have boundary PEs which are different. They perform similar computations with minor variations depending on their location in the index set of processors. In our simulation of the PE, each is modeled as a set of registers. The library routines in the simulation tool allow the designer to specify the array configuration, registers and the type of data stored (Table 1).

Registers may store fixed or floating point data of user defined bit precision. This is possible due to the special bit-addressable memory and bit-serial math capability of each Connection Machine processor. Interaction of PEs is through the exchange of data stored in these registers. In most systolic designs local interconnections and local communication are predominant. Random and global communication patterns are rare. However, both forms of communication are supported in the simulation model.

There are several advantages to implementing a simulator for systolic arrays on a SIMD machine. Since each PE is associated with a physical processor on the Connection Machine, the scalability is only constrained by the number of physical processors available and the performance deteriorates when array dimensions are larger than what can be directly mapped onto the hardware. On a typical SIMD machine, a PE or a cluster of PEs needs to be associated with a process. The fracturing of an array within or across processes implies that scalability is available at the cost of performance. Also, synchronized processing, which is a central characteristic of systolic arrays, necessitates cumbersome synchronization requirements in the form of barriers etc., further constraining scalability. Thus, instruction level synchronization on the hardware is critical. On the other hand, if the simulated systolic systems have significantly diverse index dependent computations, the SIMD model may severely hamper performance. Such arrays are rare and indeed antithetic to the systolic paradigm.

### 3. Using the systolic array simulator

The systolic array simulator is essentially a set of routines to assist the designer/implementor of systolic arrays to set up, simulate, examine the behavior and verify the results/correctness of the code executed by each PE. The simulator provides the user with a model of the array and the PE as described in the previous section. The principal simulation primitives available and their functions are tabulated in Table 1.

The user specifies the physical layout of the array in terms of the number of processors (PEs) and the organization of data storage within each PE. The data to be input or loaded into the array at specific times during the simulation is then initialized. Before simulation can begin, the user needs to specify the computations that occur at each PE along with the synchronous communication of data between PEs. Finally, an *activation sequence* of the PEs/systolic array is necessary.

The *setup* routines listed in Table 1 allow the user to specify the layout of the array and the PE organization. The *load reg* and *dump reg* routines permit data I/O to and from the array into and out of the host. The data for the input is determined prior to the simulation and stored in data files with a specialized naming convention which includes a timestamp corresponding to the time in simulation at which the data is utilized. The data captured by the output routines is stored in files with a naming convention similar to that used for input. The data files reside on the front-end to the Connection Machine and this interaction mimics the interaction between systolic arrays and their host processors (see following section).

The state of a processor (PE) at any time during computation is characterized by the data in its registers. A *snapshot* is the cumulative state of all the processors in the array at any time during simulation. *Snapshots* capture significant details of array activity and are extremely useful in the verification of designs and the correctness of algorithms. The *dump\_reg* routine may be used to obtain *snapshots* of the array's activity when suitably inserted at different times during the simulation.

Systolic arrays are synchronized by a global clock. Also, the times at which the different PEs become active follow a cyclic pattern that is mostly independent of the size of the array and a property of the problem being solved. An *activation sequence* is the periodic time sequence of activity/inactivity of the PEs of a systolic array. Each step in the *activation sequence* is termed an *activation pattern* and may be specified as a bitmap laid out in the shape of the array. The *activation\_seq* routine is the simulation primitive that helps specify the *activation sequence* for the array being simulated.

The computations that are performed by the PEs of a systolic array are similar with minor variations depending upon the location of the PE in the index set of processors. It is therefore possible to specify the computation of all PEs in a single subroutine with conditional branches to handle the variations. However, in view of the fact that multirate systolic arrays do exist, i.e. PE computations vary significantly with the activation pattern, it is preferable to index the computations at PEs by the corresponding step/*activation pattern* in the *activation sequence*. The *pe\_computation* simulation primitive shown in Table 1 relates *activation patterns* and PE computations.

Table 1. Primitives for Simulation

Primitive	Function
<i>setup_array(m, n)</i>	Set up an array of dimensions $m \times n$
<i>setup_regs(r, type)</i>	Set up $r$ registers of specified <i>type</i> (float, fixed etc.)
<i>load_reg(reg, type)</i>	Data input to register <i>reg</i> of specified <i>type</i>
<i>dump_reg(reg, type)</i>	Data output from register <i>reg</i> of specified <i>type</i>
<i>send_reg(type, src, dst, dir, wrap_mode)</i>	Data transfer from register <i>src</i> of specified <i>type</i> of all active PEs to register <i>dst</i> of neighboring PEs in the direction <i>dir</i> with exchange across edges determined by the <i>wrap_mode</i>
<i>recv_reg(type, src, dst, dir, wrap_mode)</i>	Data transfer to register <i>dst</i> of specified <i>type</i> of all active PEs from register <i>src</i> of neighboring PEs in the direction <i>dir</i> with exchange across edges determined by the <i>wrap_mode</i>
<i>activation_seq(i, *active[i][m][n])</i>	Loop of length $i$ of different activation patterns stored at <i>*active[i][m][n]</i>
<i>pe_computation(i)</i>	Code to be executed by active processors during activation pattern $i$

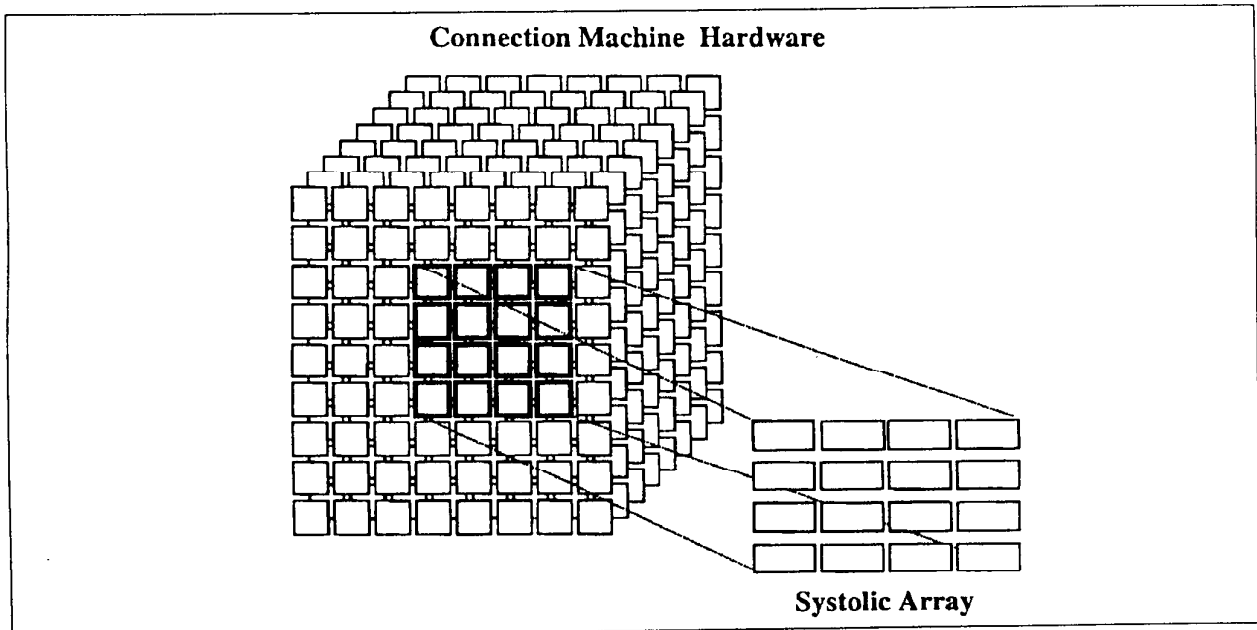


Figure 2. Mapping of a Systolic Array onto the Connection Machine

Communication between PEs is an essential characteristic of systolic arrays. The type of synchronous communication that is seen in these processor arrays is predominantly near-neighbor. The simulation primitives support an eight-way near-neighbor type of interconnection. The communication primitives *send\_reg* and *recv\_reg* provide the basic support for near-neighbor data exchange. The physical processors on the Connection Machine are placed on the nodes of a hypercube. It is therefore possible to have toroidal interconnections across the edges/boundaries of the array, if indeed desired. General communication among PEs is also possible. However, it is advisable to cast regular non-neighbor communication as combinations of near-neighbor communication steps for performance reasons. It is important to note that interconnections between PEs are modeled implicitly through the data communication specified as part of the PEs' computation.

With a view to extending the capabilities of the simulator, it has been designed with two principal operational modes during simulation. In the fault-free (FF) mode, there are no faults in the array, and normal PE and array behavior are observed. In the fault-tolerant (FT) mode, the array is modified to reflect the effect of the specific fault-reconfiguration algorithm being invoked. The modification is performed at two levels. It may be architectural to reflect physical reconfiguration, or it may be behavioral to reflect the change in PE activity under fault conditions. The primitives for the FT mode are under development.

#### 4. Simulation method and implementation

As mentioned in the previous section, the user provides the description of the array configuration and PE organization. Information on the computations at PEs and how they differ across the index set of processors is also necessary. In the current implementation, the user description of the array and PE behavior is in the form of a high-level language program aided by a library of simulation primitives. Once the array architecture and PE behavior are specified, the simulation library manages the simulation. An event-driven simulation environment simulates array behavior. The algorithm followed in the simulation of systolic arrays is shown in Table 2.

The synchronous computation and communication characteristic of systolic arrays precludes hardware/software conflict resolution. A SIMD programming paradigm, where implicit instruction-level synchroniza-

Table 2. Systolic Array Simulation Algorithm

```

begin
  while TRUE do
    for i = 1 to length(activation_sequence)
      activate_processors(activation_sequence(i))
      processor_element_computation(i)
      if termination_condition() == TRUE then exit
    end
  end
end

```

tion is available on the hardware, is therefore ideal for the simulation of systolic arrays and there is no need to enforce any kind of synchronization. The core loop in the execution of the simulation is the enforcing of the *activation pattern* followed by the execution of the PE computation corresponding to the index of the current *activation pattern* in the overall *activation sequence* (Table 2).

The programming environment of the Connection Machine supports parallel versions of several common high-level languages. The operations on the Connection Machine hardware may also be specified using Paris (PARallel Instruction Set). The Connection Machine hardware essentially operates as a co-processor to a host or front-end computer. Most of the simulation library routines are written in C/Paris (Thinking Machines, 1990a), a front-end C compiler with a Paris interface to control the Connection Machine hardware. The use of C/Paris for most of the simulation library improved code efficiency and performance due to the low level control of the Connection Machine hardware possible through Paris. The source code for the simulator can be easily ported across the different front-end architectures that can support the Connection Machine hardware.

The user code is written using C\* (Thinking Machines, 1990c), a parallel C language compiler. The syntax of C\* is quite powerful while preserving ease of notation. The C\* language provides a variety of parallel programming primitives for user specified control of computation. The organization of the various software modules that make up the simulator is shown in Figure 3.

#### 4.1 Porting the Simulator to the CM5

With the arrival of the CM5 (Thinking Machines, 1992), a MIMD machine which is designed to support C\* code written for the earlier Connection Machine models, the C/Paris interface module is being modified to allow portability. Paris is a macro interface for the microcode of the CM2 hardware and is incompatible with the CM5 architecture which derives its processing power from SPARC processors.

Porting the simulator to the CM5 considerably increases the speed of the simulator since the CM2 is bit-serial and is clocked at speeds an order of magnitude slower than the CM5. However, some of the features of the simulator that were made possible by the flexibility afforded by the bit-serial nature of the CM2 are no longer possible to accommodate without a considerable loss in performance. User-specified bit-precision in the definition of storage on the PEs is a major feature that is made infeasible by the port.

The Connection Machine models support a software abstraction called *virtual processor* (Thinking Machines,

1990b), which facilitates the mapping of a virtual array with more processors than are physically available. This is done by mapping many-to-one virtual processors onto a single physical processor and sharing the resources. This necessarily affects performance if more *virtual processors* are needed than the number of physical processors available. Often CM2 installations have more physical processors available than CM5, but this does not always translate to better performance since the CM5 is capable of computation at a much faster rate.

#### 5. Case study - complex SVD array

We have used the simulator in verifying an algorithm for a systolic array to compute the Singular Value Decomposition (SVD) of complex matrices (Hemkumar and Cavallaro, 1992). The array is based upon the Brent-Luk-VanLoan array for computing the SVD of real matrices (Brent et al., 1985). A singular value decomposition of a matrix  $M \in C^{m \times n}$  is a factorization given by

$$M = U \Sigma V^H,$$

where  $U \in C^{m \times m}$  and  $V \in C^{n \times n}$  are unitary matrices and  $\Sigma \in R^{m \times n}$  is a real non-negative "diagonal" matrix of singular values. As an example case study, it illustrates the benefits from the use of such a simulator.

The complex SVD array is a square array of processors. Each PE stores a  $2 \times 2$  sub-matrix of the problem. As an atomic step in the iterative algorithm based on the Jacobi method, the PEs on the main diagonal compute the SVD of the  $2 \times 2$  sub-matrices stored in them. These PEs then transmit the required parameters to the off-diagonal PEs so that they can update the  $2 \times 2$  matrices

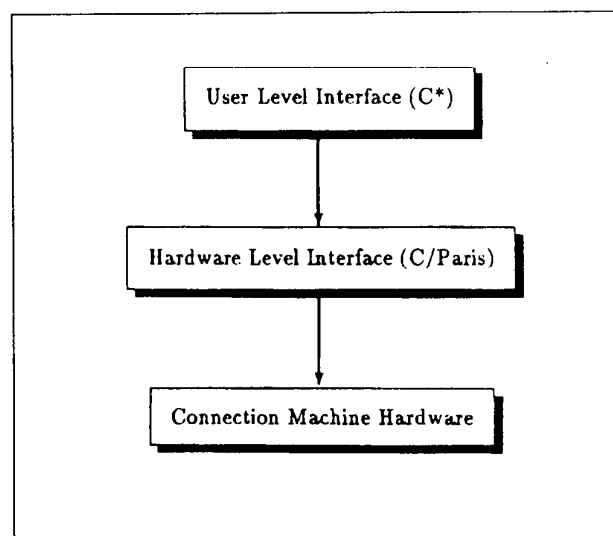


Figure 3. Organization of the Simulation Tools

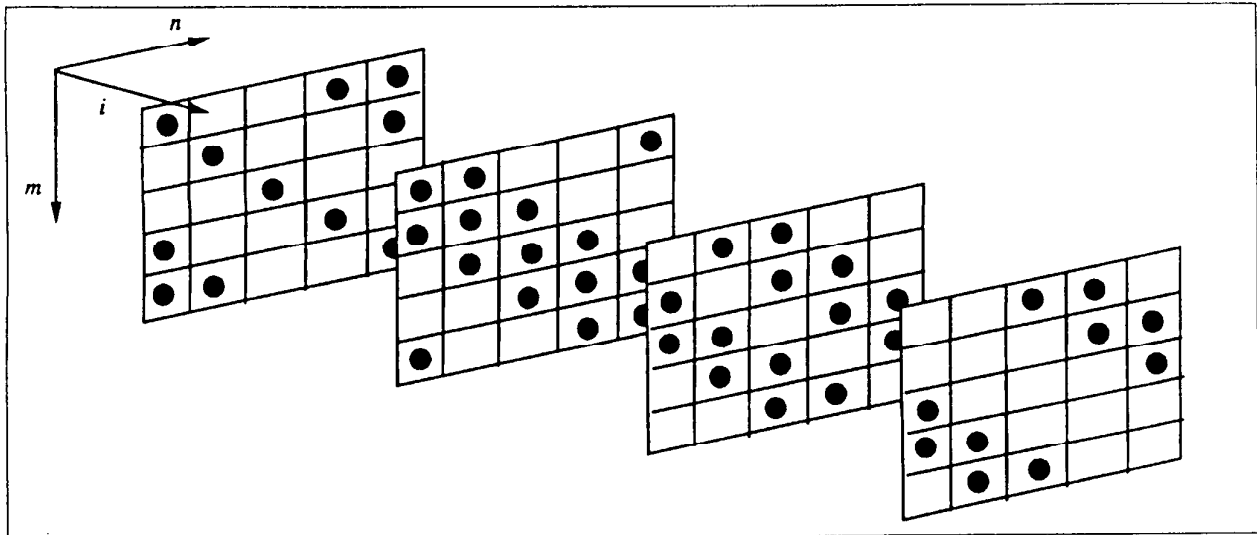


Figure 4. Complex SVD Array: Activation Sequence

stored in them to reflect changes made along the main diagonal. The algorithm also requires a complicated data exchange among the PEs between each successive computation along the main diagonal. The data exchange achieves a permutation of the elements of the matrix so that eventually, only the diagonal elements remain non-zero, thus computing the SVD. The complex SVD array algorithm is outlined in Table 3. The routines are shown only as shells to mask the complexity and to detail the essential steps. The *activation sequence* of the complex SVD array for an arbitrarily chosen dimension of  $5 \times 5$  processors ( $10 \times 10$  matrix) is shown in Figure 4, where the dots represent active processors. The sequence is of length 4 (same for arrays of any dimension) and each PE is active twice every four time steps.

Not only was the simulation useful in the verification of the data exchange algorithm, it was instrumental in validating the conjecture regarding the number of permutations needed for convergence and the comparison of the convergence behavior for real and complex data matrices. Figure 5 shows a series of snapshots of data that was captured at different times ( $t = \text{number of instantiations of activation patterns}$ ) during a simulation of the SVD array ( $16 \times 16$  processors or  $32 \times 32$  matrix). It is rendered as a surface plot where the  $x$  and  $y$  dimensions are defined by the shape of the array and the  $z$  dimension is a measure of the absolute value of the data at the processors. From the snapshots, it is readily observable that the data in the array is converging to a diagonal matrix.

Table 3. Complex SVD Array : Algorithm

```

begin
  while TRUE do
    for i = 1 to 4
      activate_processors(activation_sequence(i))
      processor_element_computation()
      {
        where m == n                               /* diagonal processor */
        compute_2x2_svd_step()
        transmit_parameters()
        exchange_data()
      }
      else                                         /* off-diagonal processor */
        receive_parameters()
        apply_2x2_transformation()
        exchange_data()
      }
      if off_norm() <= min_value then exit
    end
  end
end

```

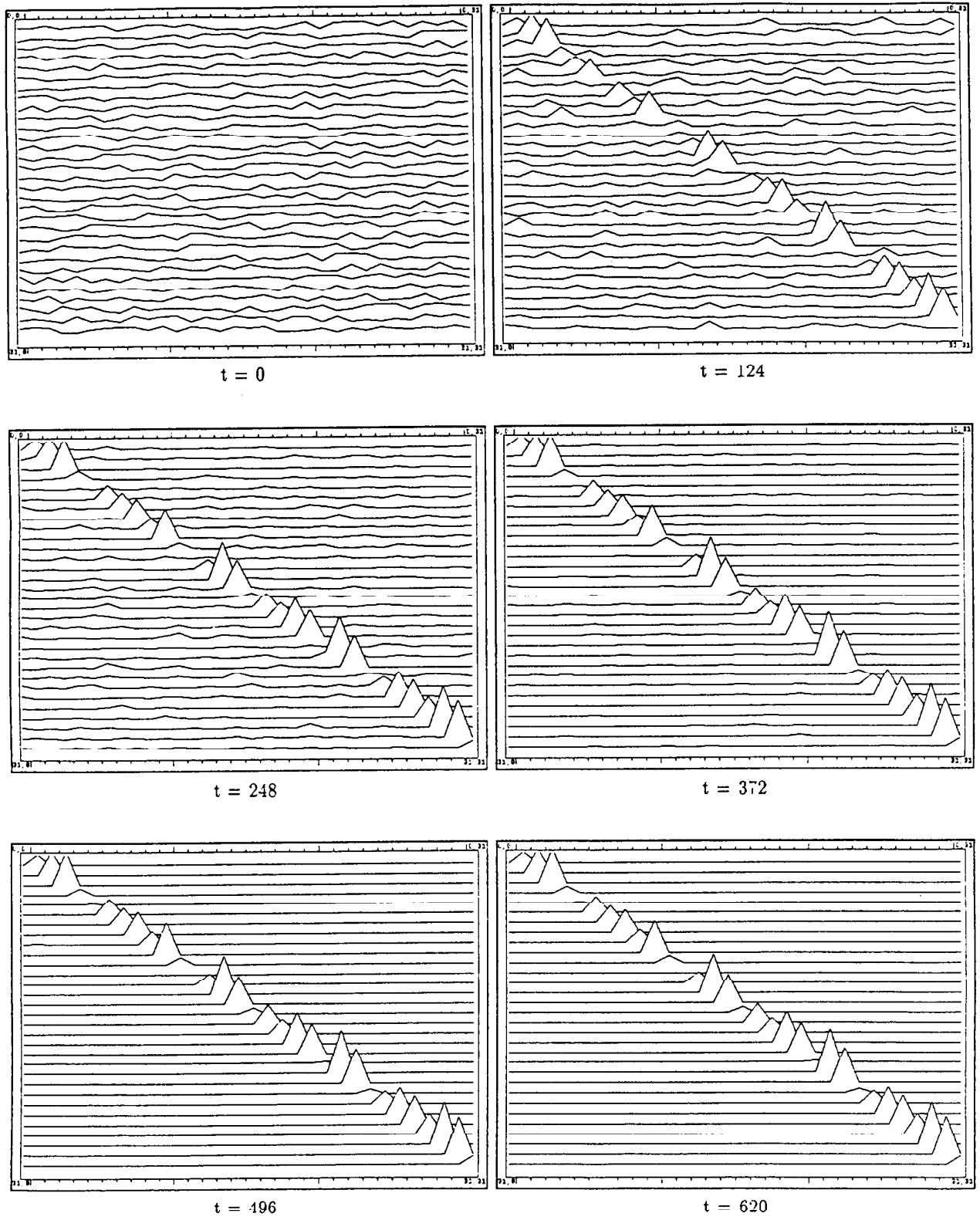


Figure 5. Snapshots of Data from SVD Array

## 6. Future Work

Fault-tolerance is of significant importance in the arrays used for real-time computations, especially when used as dedicated processors with little or no accessibility. Failure of PEs that make up the array is highly probable in large-scale implementations. The goal of any fault-tolerant hardware or software array reconfiguration scheme is to realize a logical configuration of PEs capable of meeting the algorithmic needs. Several approaches to fault-tolerance reported in the literature include: spatial redundancy (or hardware redundancy), temporal redundancy and algorithm-based fault-tolerance schemes (Huang and Abraham, 1984; Kung and Lam, 1984). We are extending the capabilities of the simulator to simulate random faults in the array and to observe the performance of dynamic fault-reconfiguration algorithms that have been designed into the PEs of a given processor array.

## 7. Conclusions

Systolic architectures and algorithms have received significant attention in the last decade. There are a variety of formal methods available for the realization of systolic algorithms from a high-level problem specification. The translation of a systolic algorithm to hardware and its implementation on custom VLSI or DSP arrays is aided by the use of a simulator for systolic arrays. In this paper, we presented a systolic array simulator. It is a set of library routines which use the Connection Machine for the efficient simulation of systolic arrays. The model of the array used in simulation maps each processor of the array to a physical processor on the Connection Machine. The simulator is a valuable tool in the verification and testing of systolic array designs. Current work on the simulator is aimed at extending its capabilities to aid in design for fault-tolerance and optimal strategies for fault-reconfiguration.

## Acknowledgements

Use of the Connection Machine at Rice was provided by the Center for Research on Parallel Computation under NSF Cooperative Agreement Number CCR-9120008 with support from the Keck Foundation and Thinking Machines Corporation. Use of the CM5 was supported by the Army Research Office contract number DAAL02-89-C-0038 with the Univ. of Minnesota Army High Performance Computing Research Center. This work was supported in part by the National Science Foundation under awards MIP-8909498 and DDM-9202639.

## References

- Brent, R. P., F.T. Luk and C.F. Van Loan, 1985. "Computation of the Singular Value Decomposition Using Mesh-Connected Processors." *Journal of VLSI and Computer Systems*, 1(3):242-270.
- Covington, R. G. S. Dwarkadas, J.R. Jump, S. Madala, and J.B. Sinclair, 1991. "The Efficient Simulation of Parallel Computer Systems." *International Journal of Computer Simulation*, 1:31-58.
- Dawkins, W. P., 1989. "Efficient Simulation of Simple Instruction Set Array Processors." Master's thesis, Rice University, Department of Electrical and Computer Engineering, Houston, TX.
- Dew, P. M. and L.J. Manning, July 1986. "Comparison of Systolic and SIMD Architectures for Computer Vision Computation". In *Proc. Inter. Workshop on Systolic Arrays*, University of Oxford.
- Flynn, M. J., 1966. "Very High Speed Computing Systems". *Proceedings of the IEEE*, vol. 54:1901-1909.
- Fortes, J. A. B. and D.I. Moldovan, August 1985. "Parallelism Detection and Algorithm Transformation Techniques useful for VLSI Architecture Design". *Journal of Parallel Distributed Comput.*, 2:277-301.
- Gentleman, W. M. and H.T. Kung, August 1981. "Matrix Triangularization by Systolic Arrays". *Proc. SPIE Real-Time Signal Processing IV*, 298:19-26.
- Hemkumar, N. D. and J.R. Cavallaro, May 1992. "A Systolic VLSI Architecture for Complex SVD". In *Proceedings IEEE Int. Symp. on Circuits and Systems*, vol. 3, 1061-1064, San Diego, CA.
- Huang, K. H. and J.A. Abraham, June 1984. "Algorithm-based Fault-tolerance for Matrix Operations". *IEEE Transactions on Computers*, vol. C-33(6):518-528.
- Kung, H. T. January 1982. "Why Systolic Architectures?" *IEEE Computer*, 15(1):37-46.
- Kung, H. T. and M.S. Lam, 1984. "Fault-Tolerant VLSI Systolic Arrays and Two-Level Pipelining". *Journal of Parallel and Distributed Computing*, 1(1):32-63.
- Kung, H. T. and C.E. Leiserson, 1980. "Algorithms for VLSI Processor Arrays." In *Introduction to VLSI Systems*, Mead, C. and Conway, L., editors, 271-292. Addison-Wesley, Reading, MA.
- Kung, S. Y. 1987. *VLSI Array Processors*. Prentice Hall, Englewood Cliffs, NJ.
- Moldovan, D. I. January 1987. "ADVIS: A Software Package for the Design of Systolic Arrays." *IEEE Transactions on Computer-Aided Design*, vol. CAD-6(1):33-40.
- Rajopadhye, S. V. and R.M. Fujimoto, 1990. "Automating the Design of Systolic Arrays." *Integration - the VLSI Journal*, 9:225-242.
- Rao, S. K. 1986. "Regular Iterative Algorithms and their Implementations on Processor Arrays." Ph.D. thesis, Stanford University.



Thinking Machines 1990a. *Connection Machine, Introduction to Programming in C/Paris*. Thinking Machines Corporation, Cambridge MA.

Thinking Machines 1990b. *Connection Machine, Model CM-2 Technical Summary*. Thinking Machines Corporation, Cambridge MA.

Thinking Machines 1990c. *Connection Machine, Programming in C\**. Thinking Machines Corporation, Cambridge MA.

Thinking Machines 1992. *Connection Machine, Model CM5 Technical Summary*. Thinking Machines Corporation, Cambridge MA.

Ullman, J. D. 1984. *Computational Aspects of VLSI*. Computer Science Press, Rockville, MD.



JOSEPH R. CAVALLARO received the B.S. degree from the University of Pennsylvania, Philadelphia, PA, in 1981, the M.S. degree from Princeton Univ. Princeton, NJ, in 1982, and the Ph.D. degree from Cornell University, Ithaca, NY, in 1988, all in electrical engineering. From 1981 to 1983, he was with AT&T Bell Laboratories, Holmdel, NJ. In 1988 he joined the faculty of Rice University,

Houston, TX, where he is an Assistant Professor of Electrical and Computer Engineering. His research interests include computer arithmetic, fault tolerance, VLSI design and microlithography, and VLSI architectures and algorithms for parallel processing and robotics.

Dr. Cavallaro is a recipient of the NSF Research Initiation Award 1989-1992 and the IBM Graduate Fellowship 1987-1988, and is a member of Tau Beta Pi and Eta Kappa Nu.



NARIANKADU D. HEMKUMAR was born in Madras, India on March 11, 1968. He received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, India, in 1989. He received the M.S. degree in 1991, also in electrical engineering, from Rice University, Houston, TX, where he is currently pursuing the Ph.D. degree.

His research interests include computer arithmetic, VLSI architectures and algorithms, and synthesis and design for fault tolerance of array processors. He is a recipient of the Government of India National Talent Scholarship and the Larsen & Toubro (India) Scholarship, and is a member of Eta Kappa Nu.