Simulation of Systolic Arrays on the Connection Machine

Nariankadu D. Hemkumar
Joseph R. Cavallaro

CRPC-TR92228
August 1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

To appear in *Proceedings of the Society for Computer Simulation (SCS) 1992 International Simulation Technology Conference SimTec '92*, Houston, TX (November 1992)

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

ABSTRACT

In this paper, we present a systolic array simulator, a simulation tool for the Connection Machine (model CM2)[1], to aid the verification of algorithms for systolic arrays. Especially as recent advances have automated the design, there is a need for a verification environment to prototype these arrays. Given their characteristics, the SIMD paradigm of computation is suitable for the simulation of systolic arrays. The Connection Machine, a SIMD computer with powerful interprocessor communication capabilities, is an ideal choice. 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.

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 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 or DSP 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).

Although, systolic/wavefront array processor hardware is realized through the replication of relatively simple units, its high interconnection and synchronous communication requirements complicate realization and operation. Failure of PEs that make up the array is highly probable in large-scale implementations. Fault-tolerance is of significant importance in

[1] Unless otherwise specified, all future references to the Connection Machine pertain to model CM2.

the arrays used for real-time computations, especially when used as dedicated processors with little or no accessibility. We are currently working on extending the capabilities of the simulator to allow the simulation of dynamic array reconfiguration algorithms under transient and permanent fault conditions.

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.

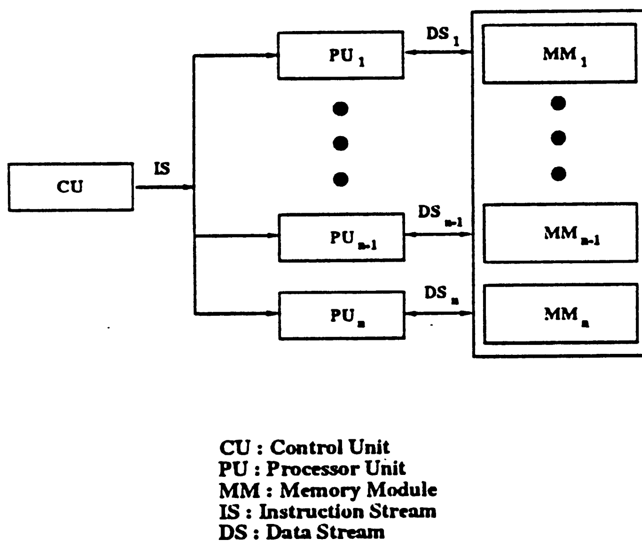

Figure 1: SIMD Architecture

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 its own 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.

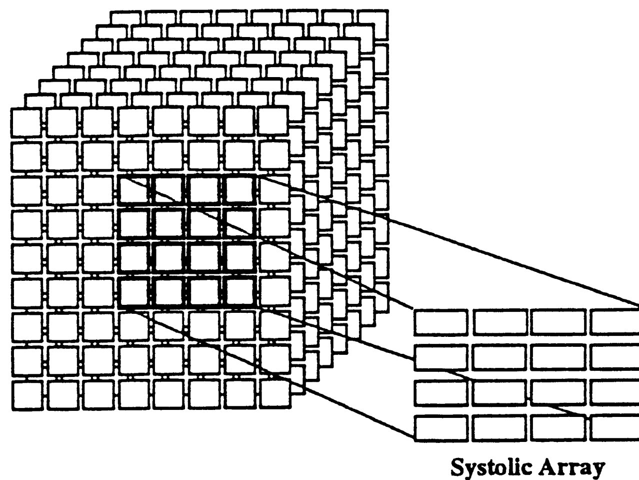

Figure 2: Mapping of a Systolic Array onto the Connection Machine

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 processor is modeled as a set of registers. The library routines in the simulation tool allow the designer to specify the array configuration and the allocation of registers and type of data stored for the PEs (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 communication is predominant. Random and global communication patterns are rare. However, both forms of communication are supported in the simulation model.

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

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

Table 1: Primitives for Simulation

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 follows 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.

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-near-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 is 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.

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 Figure 3.

The synchronous computation and communication characteristic of systolic arrays precludes hardware/software conflict resolution. A SIMD programming paradigm, where implicit instruction-level synchronization 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* (Figure 3).

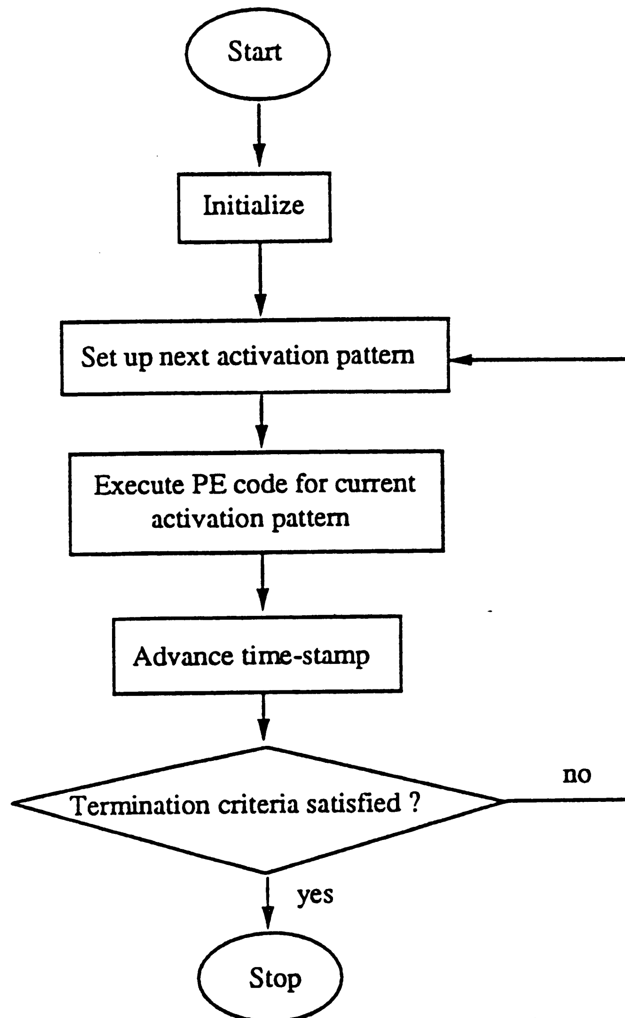

Figure 3: Systolic Array Simulation Algorithm

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 (Think-

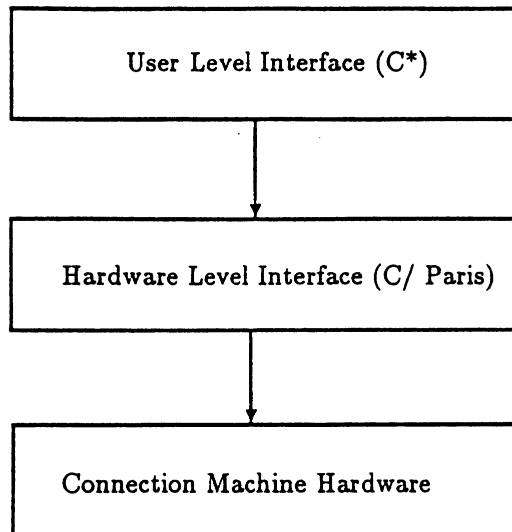

Figure 4: Organization of the Simulation Tools

ing 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 4. 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.

An Example Case Study

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). A singular value decomposition (SVD) 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 2×2 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 2×2 sub-matrices stored in them. These PEs then transmit the required parameters to the off-diagonal PEs so that they can update the 2×2 matrices stored in them to reflect to 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 *activation sequence* for the array is of length 4 and each PE is active twice every four time steps. Not only was the 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.

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.

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.

ACKNOWLEDGMENTS

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. This work was supported in part by the National Science Foundation under Research Initiation Award MIP-8909498.

References

Covington, R. G., Dwarkadas, S., Jump, J. R., Madala, S., and Sinclair, J. B. 1991, "The Efficient Simulation of Parallel Computer Systems". *International Journal in 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 Manning, L. J. 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 Moldovan, D. I. August 1985, "Parallelism Detection and Algorithm Transformation Techniques useful for VLSI Architecture Design". *J. Parallel Distributed Comput.*, 2:277–301.

Gentleman, W. M. and Kung, H. T. August 1981, "Matrix Triangularization by Systolic Arrays". *Proc. SPIE Real-Time Signal Processing IV*, 298:19–26.

Hemkumar, N. D. and Cavallaro, J. R. May 1992, "A Systolic VLSI Architecture for Complex SVD". In *Proceedings IEEE Int. Symp. on Circuits and Systems*, volume 3, pages 1061–1064, San Diego, CA.

Huang, K. H. and Abraham, J. A. 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 Lam, M. S. 1984, "Fault-Tolerant VLSI Systolic Arrays and Two-Level Pipelining". *Journal of Parallel and Distributed Computing*, 1(1):32–63.

Kung, H. T. and Leiserson, C. E. 1980. Algorithms for VLSI Processor Arrays. In *Introduction to VLSI Systems*, Mead, C. and Conway, L., editors, pages 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 Fujimoto, R. M. 1990, "Automating the Design of Systolic Arrays". *Integration - the VLSI Journal*, 9.

Rao, S. K. 1986. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD 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.