ISSN 1369-1961

Department of Computer Science

Technical Report No. 742

High Performance Parallel Processing Simulator

S.P.V. Barros

QUEEN MARY
AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

August 1997

ISSN 1369-1961

High Performance Parallel Processing Simulator

S. P. V. Barros[1]

ABSTRACT

This paper analyses the performance of APLS, an associative synchronous event-driven simulator. We develop further on the results of a previous phase of study[1] where in two key factors were addressed:- (a) the case for treating simulation as a SIMD problem, rather than MIMD, has been presented, and (b) the benefits of associative parallel logic simulation (APLS) based on a SIMD architecture called SAP[2] (Symbolic Associative Processor) which performs in-memory parallel processing has been shown to deliver superior performance compared with non-associative SIMD counterparts such as the DAP[3] (Distributed Array Processor) as well as other general sequential SISD architectures based on the von Neumann computer model. In our quest towards achieving very high simulation performance in a scaleable manner, we pay particular heed to the need for architectural enhancements alongside algorithm development to ensure that both aspects match the requirements of the particular applications. It is demonstrated that simulation performances in the region of 44μS per epoch (or 22,650 simulation epochs/second) are attainable, and we propose architectural enhancements to improve this performance by nearly 5 fold (i.e. in excess of 100,000 simulation epochs/second).

1. Introduction.

Developments in CAD technology in conjunction with developments in computer technology have together facilitated the realisation of very effective computer simulation tools for the design engineers. Historically, in the mechanical and the electronics CAD arena, specialist packages have been in extensive use for a considerable amount of time, the rapid uptake of which was only hindered by the lag in the development of low cost personal computing facilities until the 1980's.

In the digital CAD area, the simulation task had posed great challenges as a digital system is implicitly an active system; in that it constantly generates events that require a great deal of computational power to service efficiently. Never-the-less, the challenges were met through the development of complex algorithms and supported by the developments in high speed sequential computers.

CAD technology for digital electronics is today facing the challenge of simulating highly complex systems containing millions of logic elements (VLSI and WSI scales of integration). The deployment of CAD tools and VHDL in silicon design is no longer necessary, they are imperative if designers are to maximise the chances of producing minimum-cost, *first-time-right* designs.

Although the widespread usage of simulators in silicon design is evident, the plethora of silicon design complexities places considerable strain on commercial simulators, manifestly apparent in the form of lengthy simulation run-time, sometimes several days, and on occasions longer than the host computers MTBF.

Researchers have, for more than a decade, sought alternative solutions through the deployment of more appropriate computer architectures and parallel algorithms in the quest for better performance. Much of the work on parallel simulation has been based on the development of complex algorithms on computer

[1] *e-mail*: Silvano@dcs.qmw.ac.uk, Dept. of Comp. Sci., Queen Mary: University of London, London E1 4NS, United Kingdom

models such as Connection Machine,[6] Mini-DAP[3] and special purpose hardware accelerators.[7,8,9] Little has been reported on SIMD approaches to the problem which is the cornerstone of our research. In the next section we present a brief background of the SIMD perspective we have adopted before examining the performance of the APLS algorithm.

2. The SIMD Perspective of Simulation.

The process of logic simulation is about mimicking the behaviour of a real system accurately, in the shortest space of time. A digital system consisting of logic elements may be treated as a collection of processing elements which perform simple logic evaluations in parallel. At the lowest level of digital design, the logic elements are simple gates, each of which evaluate their own particular Boolean function (e.g. AND, OR) simultaneously.

It is easy to visualise the above process as being MIMD in nature. The process may, however, also be treated as a SIMD process in which every element functionally performs the same given, higher order, operation at each time step; namely, the *logical evaluation* of output states as a function of input states, the *scheduling* of output changes and *propagation* of new events according to the problem connection graph.

With this perception of simulation, we have demonstrated that very desirable performance characteristics are attainable using the SAP (Symbolic Associative Processor) SIMD architecture (Fig. 1). The advantages are due to two factors, *in-memory* processing and *parallel* processing.

Our present research is based on the application of the SAP as a specialised support for logic simulation (Fig. 2). The problem circuit represented as a connection graph is loaded into the SAP. Stimulus test patterns are then fed into the engine and the results returned to the host computer.

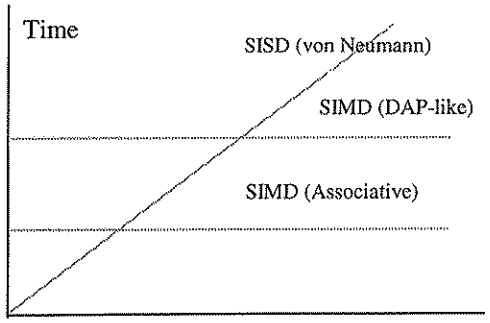

Fig. 1. - Execution times (SISD vs SIMD).

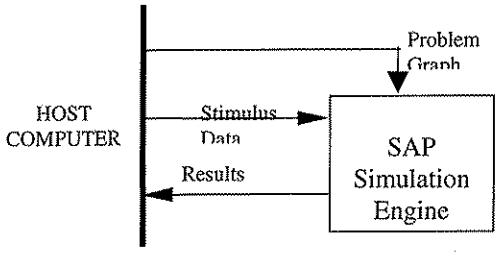

Fig. 2. - Target Simulation System.

This perspective has lead to the development of an original algorithm, called APLS, based on API's (Associative Processing Instructions). APLS operates as a synchronous SIMD process incorporating a single global clock and may be broadly categorised in the synchronous event-driven class. Special features catered for are *zero-delay initialisation, nominal delay analysis, feedback/reconvergence* and *spike detection/rejection.*

3. Analysis of the APLS Simulation Algorithm

In our previous study,[1] APLS was shown to require 1%, or less, computational time to perform *logical evaluations* compared with its sequential counterpart, making this approach particularly suitable for VLSI and WSI simulation. A speed advantage of around 50% - 67% was also shown to be attainable as a direct result of in-memory processing capabilities of the SAP in

contrast with other SIMD based architectures (depicted in Fig. 1).

Although these performance improvements in logical evaluations are highly desirable, the algorithm exhibits some significant dependencies on various architectural facets of the SAP which yield a less than optimum performance of the overall simulation algorithm. In the remainder of this paper, therefore, we analyse the constituent parallel functions of the algorithm closely to highlight some of the more significant facets and the scope for further improvement through architectural enhancements.

Fig. 3 presents the outline of an improved version of the synchronous APLS algorithm and Fig. 4, the gate descriptor cell for each logic element stored in the associative memory. During execution, the algorithm performs a general initialisation of the associative memory, followed by initialisation for simulation by reading the first set of Primary Input Events (PIE). This is followed by the main simulation loop which is executed until the simulation run is terminated.

The worst case performance of the algorithm is characterised by the following equation -- eqn. 1:-

$$T = \tau_{GINIT} + \quad \text{// Time for general initialisation}$$
$$z(\tau_{ENV} + \tau_{PZT}) + \quad \text{// time for zero-delay initialisation}$$
$$s(\tau_{ENV} + \tau_{ICT} + \tau_{DAT} + \tau_{UCV} + \tau_{PNC}$$
$$+ z(\tau_{ENV} + \tau_{PZT})) \qquad \ldots\ldots\ldots\ldots \textit{eqn 1}$$

Where:-

τ_{GINIT} = Time to perform Global Initialisation

τ_{ENV} = Time to EvaluateNewValues (outputs = f^n(inputs))

τ_{PZT} = Time to PropagateInZeroTime

τ_{ICT} = Time to InitializeCntTimers

τ_{DAT} = Time to DecrementAllActiveTimers

τ_{UCV} = Time to UpdateCurrentValues

τ_{PNC} = Time to PropagateNewCv's

The longest chain of zero delay elements in the problem graph "z" determines the number iterations around the zero-delay initialisation loop (*EvaluateNewValues* and *PropagateInZeroDelay*). The total simulation time (simulation epochs), denoted by the symbol "s", determines the number of iterations around the main simulation loop in the algorithm (Fig. 3).

```
// GENERAL INITIALISATION
Initialisation;

// INITIALISATION OF SIMULATION
Read in PIE // Primary Input Event
InitializeGlobalClock // to time of first PIE
    // Flush first PIE through any zero delay elements
WHILE (there are active zero delay elements)
BEGIN
        EvaluateNewValues;
        PropagateInZeroTime;
END

// MAIN SIMULATION LOOP
WHILE (there are awaiting events)
BEGIN
    EvaluateNewValues; // Evaluate new values ...
                       // schedule for propagation
                       // in nominal delay mode
    IF (new events are generated)  // SV != CV
        InitializeCntTimers // of new active elements;
    IF (active CNT timers exist) // marking time
    BEGIN
        DecrementAllActiveTimers;
        IF (any CNT timer fires)
        BEGIN
            UpdateCurrentValues // CV := SV
            PropagateNewCv // of relevant elements
        END
    END
    Advance Global Clock;
    Read in PIE // Any new Primary Input Event
    WHILE (there are active zero delay elements)
    BEGIN
        EvaluateNewValues;
        PropagateInZeroTime;
    END
END
```

Fig. 3. - Outline of APLS Algorithm.

TS[1]	For inter-PE Communications
CNT[3]	Event Countdown timer
Carry[1]	-- ditto -- related
DLY[3]	Element Propagation Delay
A[1], N[1]	Identifiers of Active PE's
CV[1]	Current Value at Output
SV[1]	Scheduled (New) Value
IN[2]	Two inputs per element
OPN[3]	Element Type Identifier (e.g. AND)
F[1]	Fanout Events - PE's with New Events
IG[2]	Elements with Primary Inputs

Fig. 4. - Element Descriptor Cell.

The main loop is the primary contributor to the aggregate simulation runtime. In attempting

performance improvement, therefore, we have treated the execution times due to the initialisation stages as being (relatively) negligible and focused on the functions within the main simulation *while* loop.

The execution time of interest is, thus, the time to complete a single pass of the main simulation loop as characterised by eqn. 2 :-

$$T_{main} = \tau_{ENV}+\tau_{ICT}+\tau_{DAT}+\tau_{UCV}+\tau_{PNC} \quad .. \quad eqn\ 2$$

For the purposes of further simplicity in this paper, we have chosen to assume that there will be no zero-delay elements in the problem graph being simulated, and thereby eliminate the execution time due to the term $z(\tau_{ENV}+\tau_{PZT})$ from the main simulation loop in eqn. 1. The above assumption is justifiable on the grounds that zero-delay elements are the artefacts of techniques employed in the simulation domain and do not physically exist in logic circuits. For example, a simulator that is capable of only dealing with 2-input logic elements may pre-process a 3-input AND gate, having propagation delay δ, into a 2-input gate having the same delay, which is in turn fed from another 2-input gate having zero delay, thereby mimicking the logic of the 3-input gate as depicted in Fig. 5.

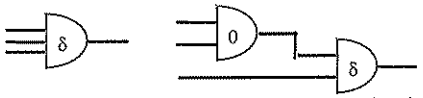

Fig. 5 - Introduction of zero-delay gates in simulation

The requirement for such pre-processing arises only if the simulator does not have sufficient provisions to deal with gates having larger numbers of inputs. The number of inputs to be catered for by a simulator is not an issue in the technical sense, but is simply a matter of debate and would be determined by empirical results. We shall, therefore, not be addressing this issue further in this paper although the impact on performance (simulation runtime) is discussed.

The potential for performance improvement through architectural enhancement is only achievable by more efficient execution of the associative functions within the APLS algorithm. The functions which provide scope for improvement (highlighted in Fig. 3) are the main contributors to the total run time and fall into two main groups; namely *propagation delay management* and *logic value management*. The characteristics of each category are now discussed to determine their contributions to overall performance.

3.1 Propagation delay management.

In contrast to most conventional synchronous event driven simulators the APLS has provisions for an independent timer/counter to be associated with each logic element in the problem graph under simulation. The scheduling of a change in value at an element output involves initialising the timer of relevant gates with the corresponding propagation delay values and initiating a count down. When the simulation epochs corresponding to the propagation delay of the element has elapsed, (i.e. the timer reaches zero) the timer fires to indicate that:-

- the corresponding element output needs to be updated,
- propagation to the next stage inputs is required, and
- new events have to be generated at the outputs of the next stage.

The main functions concerned with propagation delay management, as described, are *InitializeCntTimers* and *DecrementAllActiveTimers*.

InitializeCntTimers

The process of timer initialisation involves loading selected timers (Fig. 4 - CNT) with the designated constant propagation delay value held in the cell descriptor field (Fig. 4 - DLY). Within the SAP, although initialisation of every descriptor cell is executed in parallel, the initialisation occurs one bit at a

time starting from the LSB. This algorithm adopts a word-parallel bit-serial approach as depicted in the following pseudo-code (Fig. 6) assuming N bit delay operands:-

```
cnt := 0 // In one API, all relevant counters are reset.
FOR i = 0 to N-1 DO
BEGIN
    cnt[i] = dly[i]; // One API per bit
END
```

Fig. 6. - InitializeCntTimers Function

A single API (Associative Processing Instruction) is required to reset all *relevant* counters (*only among logic elements that generate new events*), and thereafter one API to copy each of the N bits of the DLY field into the CNT field. The bit-serial nature of the algorithm inevitably means that the execution time increases linearly with the operand size, implying that the performance of this function is architecture dependent -- eqn. 3 -- but independent of the problem circuit size :-

$$\tau_{ICT} = (N+1)\,\tau_{API} \qquad .. \qquad .. \qquad eqn\ 3$$

The significance of the dependency on the operand size is really a question of whether the absolute execution time is considered prohibitive for the size of the logic circuit being simulated. For instance, 32 bit operands for CNT and DLY will provide an ample propagation delay capacity of 2^{32} (or 4,294,967,296 time units), which corresponds to a total execution time of 12.3 μS/epoch (τ_{API} = 371 nS) irrespective of the number of active elements in the simulation. The important question is the proportion of total execution time that this function requires, given its bit serial nature (§ 4).

DecrementAllActiveTimers

Arithmetic operations in an associative processor are known to be inefficiently implemented[4] without further hardware enhancements. Incrementing and decrementing in the SAP can be performed either by adding or subtracting a unit to/from the operand value. The current specification of the SAP does not permit efficient execution of the *DecrementAllActiveTimers* function.

Again, the algorithm is in fact a word-parallel bit-serial process (Fig. 7) and benefits can be gained if the size of the problem graph is sufficiently large (e.g. VLSI design) through economy of scale.

```
FOR i = 0 to N-1 DO
BEGIN
  cnt[i] = cnt[i] + carry[i-1]; // Two API's per bit
END
```

Fig. 7. - DecrementAllActiveTimers Function

Clearly, this function is very similar to *InitializeCntTimers* in structure and is also dependant on the operand bit dimensions but with one significant difference, that each iteration around the FOR loop requires two API's, rather than one. This yields an execution time that is also linearly dependent on the operand field size -- eqn. 4:-

$$\tau_{DAT} = 2 * N * \tau_{API} \qquad .. \qquad .. \qquad eqn\ 4$$

For 32 bit CNT operands, and a nominal value for the API execution time (τ_{API}) of 371 nS, this yields an execution time of 23.7 μS irrespective of the number of active elements in the simulation, or the scale of the problem graph. Again, the important question is the proportion of total execution time that this function requires, given its bit serial nature (§ 4).

3.2. Logic value management.

This group of functions is concerned with the management of nodal logic values. When the CNT timer fires indicating that the propagation delay value has elapsed, two operations have to be performed.

Firstly, the function *UpdateCurrentValues* is required to schedule a new event by updating the current value (CV field) of that logic element using the scheduled value (SV field). Secondly, the function *PropagateNewCv* is required to propagate the new values to the inputs of the next level in the problem graph (namely to the inputs of the fan-out gates or primary outputs).

UpdateCurrentValues.

This is rather a trivial operation for the SAP and concerns the parallel execution of the statement :-

CurrentValue (CV) = ScheduledValue (CV)

using the CV and SV fields of the element descriptor cell. The task is basically a copying operation requiring only two API's; one to copy each of the two logic states (logic 0 and 1). The process is characterised by eqn. 5a and yields a constant execution time of 742 nS/epoch (τ_{API} = 371 nS).

$$\tau_{UCV} = 2 * \tau_{API} \quad .. \quad .. \quad .. \quad \textit{eqn 5a}$$

Extensions to the algorithm which deal with four state logic (0, 1, X and Z), or more, introduces a logarithmic dependency of $\log_2(S)$ over the number of simulation states S catered for by the simulator. The general equation for the performance for any number of states is shown in eqn. 5b.

$$\tau_{UCV} = (\log_2(S) + 1) * \tau_{API} \quad .. \quad \textit{eqn 5b}$$

Although dependent on the number of states, S, the approach is very efficient, requiring only one additional API if the number of states is doubled (due to the logarithmic dependency). Correlating this result with simulator technology which utilise 15 value algebra[5] for improved logic accuracy would simplify the equation to a constant coefficient of 5 as shown in eqn. 5c [$\log_2(15)+1 \triangleq 5$]. This yields a constant 1.9 µS/epoch to execute within the SAP, independent of other parameters.

$$\tau_{UCV}[15] = 5 * \tau_{API} \quad .. \quad .. \quad \textit{eqn 5c}$$

PropagateNewCv

This operation involves the transfer of the newly updated values of CV to the inputs of subsequent levels of the problem graph. Special treatment has been adopted for this prototype version of the algorithm to facilitate efficient transfer of logic values in a single API. (Signal propagation in a more scaleable manner is, however, the subject of further research - § 5). For the prototype version, execution requires 2 API's for each input and for a fan-in of F the execution time is given by eqn. 6a:-

$$\tau_{PNC} = 2 * F * \tau_{API} \quad .. \quad .. \quad \textit{eqn 6a}$$

In the case of a simulator employing S states, the equation is modified by the logarithmic function, $\log_2(S)$, as shown in eqn. 6b, which for 15 state logic reduces to the result in eqn. 6c:-

$$\tau_{PNC} = 2 * F * \log_2(S) * \tau_{API} \quad .. \quad \textit{eqn 6b}$$

For the prototype version of the APLS algorithm adopting 2-input gates and 15-state logic, this corresponds to an execution time of 5.9 µS/epoch, independent of any other parameter changes -- eqn. 6c.

$$\tau_{PNC}[15] = 8 * F * \tau_{API} \quad .. \quad .. \quad \textit{eqn 6c}$$

EvaluateNewValues

The task of logic evaluation involves using the current logic values (CV) to compute and schedule new outputs (SV). The algorithm is implemented very efficiently requiring 2 initialisation API's, together with 1 additional API for each of the following 4 logic element type (AND, NAND, OR, NOR) and an additional 2

API's for each of the following 2 types (EXOR and NXOR) requiring a constant total of 10 API's. The execution times for the NOT and BUF elements are eliminated by embodying them within the above 6 types thereby reducing execution cycles. All the primitive logic elements are catered for by the function and in a fully parallel manner yielding a constant execution time of 3.7 μS -- eqn. 7.

$\tau_{ENV} = 10 * \tau_{API}$ .. .. *eqn. 7*

(The corresponding results for multi-value logic is more complex and is the subject of additional research.)

4. Performance Analysis.

4.1 The prototype algorithm.

The complete set of equations governing the performance of the APLS algorithm is given below:-

$T_{main} = \tau_{ENV}+\tau_{ICT}+\tau_{DAT}+\tau_{UCV}+\tau_{PNC}$ .. *eqn 2*
$\tau_{ICT} = (N+1)\ \tau_{API}$ .. .. *eqn 3*
$\tau_{DAT} = 2 * N * \tau_{API}$ .. .. *eqn 4*
$\tau_{UCV} = (\log_2(S) + 1) * \tau_{API}$ .. .. *eqn 5b*
$\tau_{PNC} = 2 * F * \log_2(S) * \tau_{API}$ .. *eqn 6b*
$\tau_{ENV} = 10 * \tau_{API}$ *eqn.7*

Based on current data for the prototype algorithm, we assume 32-bit operand sizes for DLY and CNT (N=32), maximum fan-in of 5 (F=5) and 2 state logic algebra (S=2). This yields a total execution time, τ_{main}, of 44 μS for one simulation epoch (one pass of the main simulation loop), or an execution rate of 22,650 simulation epochs/second. (Alternatively, a simulation run of 100,000 simulation epochs would take 4.4 Seconds to execute.) Most importantly, the execution rate is constant for the above parameters and has no dependency on the size of the problem graph being simulated, and is consistent with the performance model depicted in Fig. 1.

4.2 Performance projections.

Decomposition of the APLS main functions execution times based on the same values of the parameters S, F, and N, indicates quite clearly that the execution time is dominated by two functions, (Fig. 8); namely, *DecrementAllActiveTimers* (τ_{DAT}) which is accountable for more than half the execution time (54%) and followed closely by *InitializeCntTimers* (τ_{ICT})which accounts for 28%.

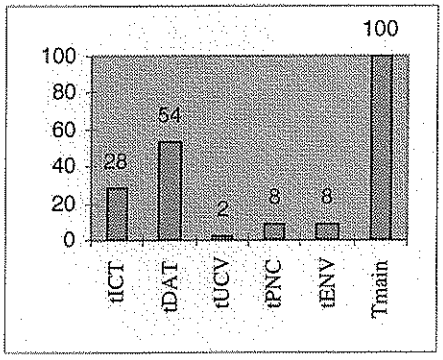

Fig. 8 - Proportion of Execution Times per epoch (%).

Although the proportions are quite contrasting the imbalance is not entirely unexpected because the two functions in question operate in a bit-serial mode. They are, therefore, considered to be less than optimum and prime candidates for improvement through implementation as special instructions in the SAP.

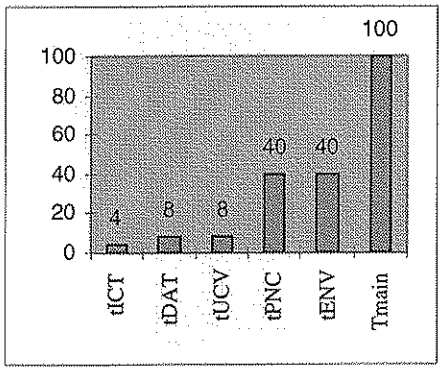

Fig. 9 - Projected Execution Times per epoch (%).

With hardware enhancement to support the new instructions, their dependency on the operand size (N) may be eliminated and considerable reduction in their execution time is achievable (Fig. 9). Furthermore, this enhancement yields a substantial reduction in the total execution time of the simulation algorithm. For the parameters specified, the total execution time (T_{main}) for each simulation epoch is 9 μS/epoch (compared with 44 previously), representing nearly a 5 fold speed up. In real terms, this corresponds to a very high simulation performance rate in excess of 100,000 simulation epochs/second.

5. Conclusions and Future Work.

We have discussed the performance of a prototype APLS simulation algorithm by examining constituent functions, and their dependencies on certain parameters. The prototype algorithm has been shown to perform at rates in the region of 44 μS/epoch (22,650 simulation epochs/second), irrespective of the size of the simulation problem (gate count).

Although the results correlated with the SIMD model shown in Fig. 1, two component functions of the algorithm (viz. *InitializeCntTimers* and *DecrementAllActiveTimers*) were found to exhibit unduly large dependencies on operand sizes used for propagation delays. This dependency is attributed mainly to their bit-serial nature of operation and it is, therefore, proposed to implement these functions in hardware within the SAP for efficient execution as a single instruction. The projected performance based on the proposed enhancement yields nearly a 5 fold aggregate speed up of the algorithm, offering a very high performance rate in excess of 100,000 simulation epochs/second.

Extensions to this work shall focus on two performance related aspects. Firstly, efficient *communication* between processing elements in a more scaleable manner than has been achieved in the prototype. Secondly, efficient *graph embedding* algorithms for the SAP, and array type architectures in general, to ensure optimum placement of the problem graph in the parallel architecture.

□

6. References

[1] Barros S. P. V., - "*Associative Parallel Logic Simulation*", Parallel Computing: State-of-the-Art and Perspectives, 1996, Elsevier Science Publishers B. V., ISBN 0 444 82490 1, pp 61 – 68.

[2] Ng Y. H., Barros S. P. V., - "*Parallel Symbolic Processing Algorithms for Text Retrieval*", Proc. International Conference on Parallel Computing '91, North-Holland, 1992 Elsevier Science Publishers B. V., ISBN 0 444 89212 5, pp 131 - 138.

[3] Lambrinoudakis C., - "*Active Memory - Computer Architecture. An introduction to the AMT DAP*", May 1989, Technical Report No. 455, Department of Computer Science, Queen Mary, University of London, UK.

[4] Barros S. P. V., - "*Arithmetic Operations with an Associative Parallel Processor*", Dept. of EE&E, Brunel University, UK, 1977.

[5] Flake P. L., Moorby P. R., Musgrave G., - "*An Algebra for Logic Strength Simulation*", Design Automation Conference, 1983, Florida USA, pp 615-618.

[6] Chung M. J., Chung Y. - "*Data Parallel Simulation using Time Warp on the Connection Machine*", 26th. Design Automation Conference, 1989, pp.99-103.

[7] Ansade Y., et al - "*Highly Parallel Logic Simulation Accelerators based upon Distributed Discrete-Event Simulation*", Proc. 1987 Oxford International Workshop on Hardware Accelerators, Adam Hilger, 1988, pp 69 - 79.

[8] Fujimoto R. M., - "*Parallel Discrete Event Simulation*", Comms. ACM, Vol. 33, No. 10, Oct. 1990, pp 30 - 53.

[9] Misra J., - "*Distributed-Event Simulation*", ACM Computing Surveys, Vol. 18, No. 1, Mar. 1986, pp 39 - 65.