An Algorithm for Discrete Event Logic Simulation on Distributed Systems

S. Subramaniam
T.S. Mohan
L.M. Patnaik
Indian Institute of Science
Bangalore 560 012.

Abstract

Increase in the complexity of VLSI digital circuit design demands faster logic simulation techniques than those currently available. One of the ways of speeding up existing logic simulation algorithms is by exploiting the inherent parallelism of the sequential version. In this paper, we explore the possibility of mapping an event-driven logic simulation algorithm onto a cluster of processors interconnected by an ethernet. The set of events at any simulation time step is partitioned by the Master Task (running on the host processor) among the Worker Tasks (running on the other processors). The partitioning scheme ensures a balanced load. Each Worker Task determines the fanout elements, evaluates them independently and comes up with the new event list which is passed onto the Master Task. After receiving the event lists from all the Workers, the Master Task executes the simulation time step, computes the new event list partition for the next simulation cycle.

We have implemented this distributed logic simulation algorithm on a set of VAXstations using CT - a package for distributed programming. The paper concludes with a note on the speedup figures obtained on the ISCAS benchmark circuits.

Introduction

Logic simulators play a major role for verifying the functionality of VLSI circuit design. Existing logic simulators can be broadly classified into three types - Compiled code[1], Event driven[1] and T-Algorithm [2]. In the case of compiled code simulation, all elements are evaluated at each time step, and the zero delay modeling of elements is implied. The event-driven approach is more flexible compared to compiled code, as different delay models can be incorporated and the element evaluation is carried out only when logic value at the input changes. The T-algorithm is more efficient than event-driven approach because it reduces the number of table-lookups[2].

Most of the existing sequential techniques take considerable amount of time to carry out logic simulation of VLSI circuits. Their execution times can be reduced by exploiting the inherent parallelism of these algorithms. The methods used to speed up the simulation are:

- Tuning up the basic algorithm to be suitable for vector processing [3],
- Hardware accelerators [5] like YSE(IBM), LSM(Bell Lab), HAL(NEC), LE(ZYCAD) etc. which map parallel algorithms directly onto the hardware.

The first approach demands efficient utilization of computational power of vector processors (high vectorization ratio, long vector length)[3]. The latter approach gives a good speedup, but suffers from the disadvantage that any change in the algorithm cannot be reflected back in the hardware easily and cost to performance ratio is high, thereby making it less attractive for speeding up the simulation.

The current trends in carrying out parallel logic simulation are concentrated on mapping the simulation algorithm on the processors of the general purpose parallel machines. We can either exploit functional parallelism which is inherent in the algorithm or the data parallelism. In the former, the simulation task is partitioned and assigned to several functional units which operate in a pipeline fashion. One of the requirements is that the simulation task has to be divided into fixed number of functional units in such a way that each functional unit takes almost the same
The Sequential Algorithm

In a VLSI circuit, at any simulation time step, only a small portion of the entire circuit is active. Therefore, it is inefficient to evaluate all the elements at each simulation step. Whenever the value of the output of a change is connected to the output signal line. These elements have to be reevaluated. In other words, at each time step only those elements whose inputs have changed have to be reevaluated. This technique is called event-driven simulation. Event driven simulators supporting more complex delay models require a special type of data structure to find out the fanout of an element for evaluation and the fanouts for finding the potentially active elements. In addition, a time flow mechanism is necessary to keep track of time order maintenance of event lists.

At any time step, the event is picked from the event list and using the event data the element logic value is modified to the present value. This value is propagated to all fanouts connected to this output, and the fanout elements are pushed into the appropriate evaluation stacks. This phase is called the fanout phase. In the element evaluation phase, the element is popped from the evaluation stack, the input logic values of the element are gathered and the element is evaluated to find the new logic output value. If the evaluated value is different from the existing value, a new event is generated and is queued in the simulation time queue depending on the delay characteristic of the element. This simulation cycle is repeated for every increment in simulation time step, until the simulation period is completed.

The Parallel Logic Simulation

2.1 The Master-Worker Tasking Abstraction

We implement the parallel algorithm using the Master-Worker tasking paradigm of the CT kernel package. The CT Kernel is a distributed programming package that extends C with tasktype - a tasking abstraction. A tasktype is an abstract datatype that encapsulates a set of data objects and defines operations on them. These operations form the sequential body of an instance of the tasktype activation: called the Task. Some of the operations within the body of the task allow the pertinent task to communicate with other tasks either synchronously or asynchronously. The communication interface of a task is uniform over the multiple processors and so is its namespace. A tasktype is different from the Ada's tasktype but has some implementation similarities.

A task is explicitly created by declaring variables of the tasktype and then explicitly invoking it. On activation, a task executes sequentially the sequence of statements in the body of the tasktype definition. On completion of the execution the task terminates automatically following a well-defined termination semantics. There can be multiple instances of a
TASK master-task0

int i, j; /* Declare the workers */
tasktype worker_task worker[MAX_WORKERS]; /* Workload for each worker */
Worker-load event_set[MAX_WORKERS].total-event;
Logic_sim_state state, incr_state, temp-vals;

for (i = 0; i < MAX_WORKERS; i++)
new(worker[i]); /* Spawn the worker tasks */
initialize_master();

for (i = 0; i < MAX_NO_SIMULATION_TIME_STEPS; get_next_sim_step(ki))
{
partition_event_list(total_event, event_set);
/* Tell Worker about fanout computation */
for (j = 0; j < MAX_WORKERS; j++)
tell(worker[j], event_set[j]);
/* Hear from Workers the results */
for (j = 0; i < MAX_WORKERS; j++)
hear(worker[j], temp_vals[j]);
update_logic_sim_state(temp_vals,
state, incr_state);
/* Tell all workers - incremental state */
tell(worker, incr-state);
/* Hear from workers - evaluation results */
for (j = 0; j < MAX_WORKERS; j++)
hear(worker[j], event_set[j]);
generate_total_events(event_set, total_event);
3
print-results0;
END_TASK
}

TASK_TYPE worker_task()
{
int i, j;
Worker_load event_set, temp_set,
new_event_set;
Logic_sim_state state, incr_state
initialize_worker();
for (i = 0; i < MAX_NO_SIMULATION_TIME_STEPS;
get_next_sim_step(ki))
{ /* Get the event list from Master task */
hear(PARENT, event_set);
generate_fanout(event_set, temp-set);
tell(PARENT, temp-set);
hear(PARENT, incr_state);
update_sim_state_worker(incr_state, state);
evaluate_circuit_elements(state,
new-event-set);
tell(PARENT, new-event-set);
3
 conclude();
END-TASK-TYPE
}

Figure 3: The pseudo code for Worker Task

tasktype active at any time. Since each task is an independently executable unit, and given the bias
towards message passing architectures there cannot be any global (shared) variables between tasks. All
function and procedure invocations from the body of the task have all necessary parameters passed to it explicitly. However the reentrant code for the procedures, functions and constant declarations are shared between tasks.

In the CT kernel, we define a Master Task tasktype as one that is the first task to be initiated on the host processor and which in turn spawns a number of Worker Tasks. The Master task takes in event list generated in every cycle and communicates it to the Workers. We define the Worker Task tasktype as one that takes in the event list communicated by the Master task, does the fanout and element evaluation and after the computation of the new set of events, passes back this event list to the Master task. It then awaits the next set of events to be processed. We have a Master task and generate a number of Worker tasks which are assigned to different processors as in Fig. 1. There can be more than two Workers per processor as this increases the CPU utilization factor because while one Worker awaits the messages from the Master, the others can do the processing (but the logic simulation state is maintained on a per-processor basis). We use the ~~~~asynchronous communication statements inherited by every tasktype definition. These communication statements are also capable of broadcasting when an array of workers are specified.

2.2 The Eventlist Partitioning Algorithm

When the computational load is not evenly distributed among the Worker tasks, some of the Workers may be idle which in turn causes processor underutilization. The number of elements a Worker Task is going to evaluate depends on the number of fanouts of the element present in the event. Assuming an equal evaluation time for all types of elements, we can say that the partitioning can be carried out on the basis of number of fanouts. The partitioning method used in our implementation gives an even balance of total fanouts among Workers and is given
2.3 The Master and Worker Task Algorithm

In Figs. 2 and 3 we give the pseudo C code of the Master and Worker task definitions.

The essential aspects of this parallel algorithm is as follows: The state of the logic simulation at any step is given by the state of the logic elements (some of which may be in a delay state). We duplicate the initial state of the logic simulation on each of the processors and for each cycle of the simulation, the state of the logic simulation is updated on each processor and thus kept consistent globally. This helps avoid formulating the necessary set of logic elements into a message packet and sending it along with the partitioned eventlist to a worker task for simulation. The master task accepts the results of a simulation cycle and after updating its state of logic simulation, broadcasts it to all the worker tasks (the relevant updated portion of the state). The pseudo code expresses this succinctly.

Logic simulation is essentially divided into two phases of activity in every simulation cycle. These are: the fanout phase where the results of evaluation of a logic element is propagated to its fanout elements, and the evaluation phase, where the logic element is evaluated.

In the fanout phase, the events queued for the pertinent simulation time step is taken and its output value propagated as the input value of the corresponding fanout elements. These fanout elements are scattered across the processors so that the evaluation phase can be carried out. In the evaluation phase, those logic elements whose inputs were updated during the fanout phase are taken, functionally evaluated, and outputs tagged on to the appropriate simulation time slot after taking into account the delay characteristic. While master task maintains the simulation time queue as well as event list, the worker tasks carry out the actual fanout and the evaluation phases in a distributed manner. The master task interacts with every worker task twice in each simulation cycle.

The initialization consists of reading in the circuit description and initializing the data structures representing the state of logic simulation. This is carried out by all tasks - master and workers on a per processor basis.

As seen from the pseudo code, the master task partitions the eventlist of the pertinent simulation time step and tells all the workers their partitioned event sets. The workers carry out the fanout phase and later the master hears these results from each of the workers. The Master then updates the state of logic simulation and broadcasts the incremental state change to all the workers. The workers after Bearing the state changes and updating their logic simulation state (on a per processor basis) carries out the element evaluation corresponding to the eventlist partitioned and assigned to it. After the completion of the evaluation phase, all workers tell the output values generated to the master task who goes on to update the eventlist and initiate the simulation cycle for the next time step.

Observations:

The above method is not a very efficient solution in a distributed environment. It can be made efficient if we maintain the logic simulation state consistent across processors for every simulation cycle efficiently. The communication costs per cycle is high as compared to the computation costs and there is good scope for improvement.

In the present implementation, we optimize on the communication costs by cutting down messages which are sent only when needed. Thus during the task assignment, if the number of fanouts is less than optimal per task, then only a few of the worker tasks are chosen and sent the workload and the rest of the worker tasks directly go to the next simulation time step.

We choose two representative circuits from the ISCAS benchmarks for evaluating the performance of this algorithm. They are c432 and c499 and were chosen to be within limits of the resources available. The Table 1 gives the timing of the parallel logic simulation run for the above two circuits on a cluster of eight, VAX station 2000 workstations interconnected by LAN. Our experience with this
partitioning scheme is that the load between simulation time-steps is characterized by spurts and between spurts there is less work for most worker tasks than during the spurts. Again we find that for more than seven processors there is no decrease in the time taken which essentially implies that processors were going idle with less work and the communication costs were rising as compared to the computation costs. More refinements need to be incorporated into the event based load partitioning algorithm for perhaps overlapped and balanced load.

### 3 Conclusions

This paper discusses the mapping of an event-driven logic simulation algorithm onto a distributed system. A load balancing scheme presented ensures that the tasks are evenly loaded. Results indicate that for large circuits where the ratio of computation to communication costs is high, the performance is good. The algorithm also presents good scope for improvement. We feel that this algorithm will be useful given the general availability of networked workstations.

### Acknowledgements

The authors thank Prof. V. Rajaraman for the encouragement and members of the KBCS, CADL, MAL and SERC for their help.

### References


