Georgia State University
ScholarWorks @ Georgia State University

Computer Science Theses | Department of Computer Science

5-10-2014

Parallel Discrete Event Simulation on Many Core Platforms Using Parallel Heap Event Queues

Govardhan Tanniru

Follow this and additional works at: http://scholarworks.gsu.edu/cs_theses

Recommended Citation
Tanniru, Govardhan, "Parallel Discrete Event Simulation on Many Core Platforms Using Parallel Heap Event Queues." Thesis, Georgia State University, 2014.
http://scholarworks.gsu.edu/cs_theses/76

This Thesis is brought to you for free and open access by the Department of Computer Science at ScholarWorks @ Georgia State University. It has been accepted for inclusion in Computer Science Theses by an authorized administrator of ScholarWorks @ Georgia State University. For more information, please contact scholarworks@gsu.edu.

PARALLEL DISCRETE EVENT SIMULATION ON MANY CORE PLATFORMS USING PARALLEL HEAP EVENT QUEUES

by

GOVARDHAN TANNIRU

Under the Direction of Dr. Sushil K. Prasad

ABSTRACT

Discrete Event Simulation on GPU's employing parallel heap data structure is the focus of this thesis. Two traditional algorithms, one being conservative and other being optimistic, for parallel discrete event simulation have been implemented on GPUs using CUDA. The first algorithm is the safe-window algorithm (conservative). It has produced expected performance when compared to sequential simulation. The second algorithm, known as SyncSim, is an optimistic simulation algorithm previously designed to be space efficient and reduce rollbacks. This algorithm is re-implemented on GPU platform with necessary changes on the logic simulator and the parallel heap implementation. The performance of the parallel heap when working with a logic simulator has also been validated against the results indicated in previous research paper on parallel heap without the logic simulator.

INDEX WORDS: Discrete Event Simulation, GPU, Parallel Heap, Logic Simulation

PARALLEL DISCRETE EVENT SIMULATION ON MANY CORE PLATFORMS USING PARALLEL HEAP EVENT QUEUES

by

GOVARDHAN TANNIRU

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

In the College of Arts and Sciences

Georgia State University

2014

Copyright by
Govardhan Tanniru
2014

PARALLEL DISCRETE EVENT SIMULATION ON MANY CORE PLATFORMS USING PARALLEL HEAP EVENT QUEUES

by

GOVARDHAN TANNIRU

Committee Chair: Sushil K. Prasad

Committee: Yanqing Zhang

Ying Zhu

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

May 2014

DEDICATION

I would like to dedicate my thesis to my mom, dad, sister and friends who have supported me all my life.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Dr Sushil K. Prasad, for providing me an opportunity to work on this thesis. He has been guiding me through all the obstacles encountered in my research work and has been a constant source of motivation. I would also like to address special thanks to Dr. Raj Sunderraman for his valuable suggestions.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ..... **v**

LIST OF FIGURES ..... **viii**

1. INTRODUCTION **1**

1.1 Evolution of Multi-core processors **1**

1.2 Parallel Priority Queue **2**

1.3 Parallel Simulation Algorithms **3**

2 BACKGROUND **6**

2.1 Discrete Event Simulation **6**

2.2 Parallel Discrete Event simulation **6**

2.3 Graphics Processing Units (GPUs) **8**

2.4 Parallel Heap **9**

2.5 Simglobal algorithm **10**

3 RELATED WORK **12**

4 IMPLEMENTATION **14**

4.1 Logic Simulator **15**

4.2 Concurrency Issues **17**

4.2.1 Events Distribution Algorithm on GPU threads ***17***

4.3 Safe-Window based Algorithm (Conservative Simulation) **18**

4.4 SyncSim Algorithm (Optimistic Simulation) **19**

4.4.1 Storage Component *19*

4.4.2 Parallel Heap Structure *21*

4.4.3 Algorithm details *22*

4.5 Parallel Heap **26**

5 EXPERIMENTS AND RESULTS **27**

5.1 Performance of Safe Window Algorithm **28**

5.1.1 Comparison of Safe Window algorithm with Sequential Heap *28*

5.1.2 Performance of Conservative Algorithm with respect to r value *29*

5.2 Performance of SyncSim Algorithm **30**

5.2.1 Comparison of SyncSim Algorithm with Sequential Heap *30*

5.2.2 Varying heap node size on SyncSim *31*

5.2.3 Effect of rollback count on heap node size *32*

5.3 A comparision of the algorithms **32**

6 CONCLUSION **34**

REFERENCES **35**

APPENDICES **37**

Appendix A **37**

LIST OF FIGURES

Figure 2.2.1 Parallel Execution of events 7

Figure 2.2.2 Causality Error..... 8

Figure 2.3.1 GPU Architecture 9

Figure 2.5 Message structure of Simglobal algorithm..... 11

Figure 4.1 Architecture diagram for Logic Simulation 14

Figure 4.2 Data structure for legacy simulation network. 15

Figure 4.3 New data Structures for Simulation Network 16

Figure 4.4 Structure of Message in Safe-Window Algorithm 17

Figure 4.5 Events distribution on GPU threads 18

Figure 4.6 Storage Component Model..... 20

Figure 4.7 Message structure for SyncSim Algorithm 22

Figure 4.8 Algorithm for Simulate () function..... 23

Figure 4.9 Algorithm for execute () function..... 24

Figure 4.10 Algorithm for insert () function. 25

Figure 5.1.1 Comparison of Safe Window and Sequential Heap Algorithms 28

Figure 5.1.2 Effect of r value on Conservative Algorithm. 29

Figure 5.2.1 Effect of r value on Syncsim Algorithm..... 30

Figure 5.2.2 Effect of r value on SyncSim Algorithm. 31

Figure 5.2.3 Effect of r value on Rollback count of Syncsim..... 32

Figure 5.3 Comparison of Algorithms based on think time..... 33

1. INTRODUCTION

Computer Simulation is a useful part of modeling many natural systems in Physics, Chemistry, Biology as well as Artificial systems in Very Large scale integrated circuits (VLSI), Network Traffic Simulation etc. Discrete Event Simulation models the operation of a system as a discrete sequence of events in time. Between consecutive events no change in system is assumed to occur. There are many drawbacks with Discrete Event Simulation when executed using a single computer, although with high computer power, using sequential algorithms. One of the major drawbacks is the execution time of the program which can simulate only one event at a time. There was a need to employ multiple processors to reduce the time spent on simulating a system. Parallel Discrete Event simulation is the execution of a Discrete Event Simulation program on a parallel computer using parallel algorithms. Since most of the systems that need to be simulated are characteristically sequential it becomes a difficult process to convert the existing sequential methods to parallel methods in order to exploit the available parallel processors.

1.1 Evolution of Multi-core processors

In the earlier days when the single-core computer is the option for computing, several computers were connected on a network to perform the parallel processing and relied on using very sophisticated distributed processing software. Later when the multi-core processors have become a norm in the industry, parallel simulation became much easier than before. With the advent of General Purpose Graphics Processing Units (GP GPU's) in the recent years parallel simulation has become much more interesting area to be researched upon with respect to the applicability of GPU's. Many parallel algorithms have been tested on GPU's not just for their performance analysis but also for an enhancement in the algorithm that is more suitable for

programming on GPU's using CUDA. In this thesis the IOWA[13] logic simulator previously used by Prasad (to generate a parallel programming engine attachable to sequential simulator) is considered on GPGPU Platform along with the Parallel Heap (A Parallel Priority Queue). The logic simulator had been re designed to suit some of the constraints in CUDA language. Most of the data structure for building the network had to be changed to array based structures instead of structure of structures.

1.2 Parallel Priority Queue

In Simulation of a large network, a common network event would be a message generated at a source node to be transferred to the destination node. The network events have to be processed by the simulator in the order of timestamps. A Priority Queue is an important data structure for Discrete Event Simulation as it enables the storage and extraction of highest priority events at a given time. For every cycle, the earliest network event is popped from the priority queue for processing. The processing of this event can trigger new network events which will be inserted into the priority queue for future processing. This process is repeated until a simulation condition is met for a definite amount of time so that the network is simulated and evaluated [9]. In order to process multiple network events simultaneously in a parallel simulation we need a parallel priority queue which can return a set of highest priority items in a single call. Parallel Heap data structure has been designed earlier to serve this purpose by Prasad[15,16]. Many versions of Parallel Heap have been implemented for multi core architectures in the past for various discrete simulation algorithms. In 2010, Parallel Heap has been designed for GPGPU's using CUDA. The performance of this Parallel Heap implementation when used in a logical simulator designed for GPGPU's has been evaluated in this thesis work.

1.3 Parallel Simulation Algorithms

Apart from the construction of the logic simulation network, two standard algorithms (one conservative and one optimistic) have been implemented on GPU. The first algorithm is called the **Safe-Window (or time-window)** algorithm which is a traditional conservative algorithm for Discrete Event Simulation. We impose a lower bound (LB) on the service time of each of the nodes in the network (termed as logical process, LP, described in section 2). This means that when an event is getting processed at an LP, say LP1, it will not produce any intermediate events which might cause rollback on any other LP in the network. In the first step of simulation we extract 'r' events from the priority queue which are stored in an array of events structure called **simMessages**. A set of independent events Ei is computed so that all of events in the Ei can be processed in parallel. This set is computed by checking for timestamps (ts) of the original r events if they fall in the range of Global Clock, GC and GC + LowerBound on Service Time. The timeframe <GC, GC+LB> is termed as safe-window since it has the set of messages which are 'safe' to process which will not cause any rollbacks. The events/messages which do not fall in the safe-window range are collected in a structure called **UnSimulated**. These events are sent back to the priority queue along with the newly generated events.

The second algorithm that has been implemented on GPU platform is the **SyncSim** algorithm which is essentially an optimistic algorithm in which rollbacks on the logical process states are allowed. This algorithm was originally designed by Prasad and Cao [14] for multi-core architectures. Primary features of this algorithm are:

1.) It does not depend on traditional state and message saving data structures. It employs a single backup state per LP and an earliest message component.

2) It drastically limits the number of rollbacks that will be encountered in an optimistic simulation.

Minor changes have been made to this algorithm to be suitable for GPU platform. Major change involved is the way the events are stored on the priority queue. A storage component is initialized on the GPU side to act as a global storage repository of the events. The event that exists on the priority queue has reference to its original copy of event on the storage. The storage component is logically partitioned to act as local queues based on the number of Logical Processes defined as the input to the generation of the network. During the simulation, the r messages to be simulated from the parallel heap (priority queue) are initially sorted on destination LP and then on timestamp. Threads launched by the gpu kernel act on the sorted messages so that no concurrency problems occur. Each message at a particular destination locates itself on the storage component based on the reference it holds to its original copy and then the original message is simulated. After the new events/messages are generated from the messages being simulated the insertion of these messages is delayed until the current gpu kernel being executed is completed. The concurrency problems associated with multiple threads trying to access the local queue of a Destination LP can be avoided by launching a separate gpu kernel which makes sure that only one thread is allocated per destination LP. During the simulation, a message is simulated at an LP only if it is the earliest message at that LP otherwise it is sent back to the global priority queue (GQ). If the current message getting simulated is earlier than the earliest message at that LP the message (current earliest message) with greater timestamp is sent back to the GQ. Same is the case during insertion process. A new message to be inserted at a destination LP might bump an existing earliest message at the LP and both the new and old message are sent to the GQ. For the case when the bumped message is already simulated the

state of the LP is roll backed to an earlier state and the message is sent for re-execution. Garbage collection of messages which are older than the global virtual time (GVT) of the simulation happens during execution and insertion of the messages. Syncsim is originally based on an algorithm called simglobal which maintains local queues at each LP of the simulation instead of a global queue. The details of simglobal algorithm are mentioned in section 2. Complete details of the SyncSim algorithm are mentioned in the implementation section.

The subsequent sections are organized as follows: Section 2 presents a brief description of background knowledge that is needed before the work is presented. It covers the basics of GPU technology and the essentials knowledge on Parallel Heap data structure. Section 3 presents relevant literature that has previously focused on discrete event simulation and parallel priority queues. Section 4 presents the detailed description of the time-window and syncsim algorithms and their implementation details. Experiments and Results are presented in Section 5. Conclusion and future work suggestions are presented in the final section.

2 BACKGROUND

Most of the current work involves discussion of work done in the past by various researchers in the areas of simulation and priority queue data structures. Programming on GPU's using CUDA language requires the knowledge of certain aspects specific to GPU's. Following are a series of sub-sections which give a brief background needed on various topics dealt in this thesis work.

2.1 Discrete Event Simulation

As mentioned earlier, Discrete Event Simulation models the operation of system as a discrete sequence of events in time. An event comprises a specific change in the system's state at specific point in time. Discrete event means that time advances until next event can occur. Time-steps during which nothing happened are skipped. Duration of activities determines how much time the clock advances for the simulation. Common applications of DES involve modeling procedures and processes in various industries such as manufacturing and healthcare. In the current work a logic simulator is being modeled to act as the Discrete Simulation network.

Logic Simulation is the use of simulation software to predict the behavior of digital circuits and hardware description languages. This can be used as a part of verification process in designing logical circuits in a hardware unit.

2.2 Parallel Discrete Event simulation

Large simulation problems consume enormous amounts of time when executed on sequential machines. This drawback calls for a simulation where in multiple events can be executed in parallel using a parallel computer known as Parallel Discrete Event Simulation. Although it looks simple to parallelize discrete event simulation there are many inherent problems that need to be dealt with starting with asynchronous systems. Concurrent execution of

multiple events at different points in simulated time introduces synchronization problems[1]. One of the famous problems with the Parallel Discrete event Simulation is the causality error. A causality error is said to occur when an event in the future has affected an event in the past of the simulation there by producing an incomprehensible situation. To explain the situation better consider a simulation system with logical processes LP1 and LP2. Let 2 events (say E1 and E2) be executed parallel for a particular cycle and are expected to produce new events for the outgoing connections. Let E1 be scheduled at timestamp 5 and E2 at 10.

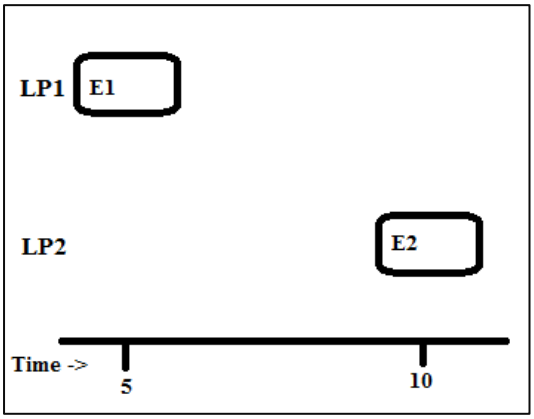

Figure 2.2.1 Parallel Execution of events

It is possible that a new event E3, might be produced at timestamp 7, for LP2 as a result of the execution of E1 at LP1.This event E3 has to be executed before executing E2 which is clearly not possible. Hence E1 and E2 cannot be executed in parallel. This situation is called as causality error. The events E1 and E2 are said to be **independent events** if the execution of both the events concurrently will not cause any causality error.

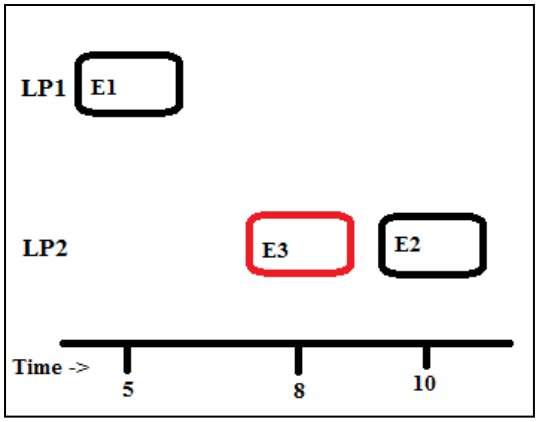

Figure 2.2.2 Causality Error

Based on this scenario of causality errors Parallel Discrete Event Simulation (PDES) has been broadly classified into 2 categories. i.e Conservative simulation and Optimistic simulation.

Conservative PDES aims to make sure that only a set of independent events (events which are safe to process) are executed in every cycle. This involves calculation independent events for every cycle of the simulation.

Optimistic PDES allow the causality errors to occur and a rollback mechanism is applied when the errors have been detected. This is termed as detection and recovery approach.[1]

2.3 Graphics Processing Units (GPUs)

In 1999 NVIDIA has released GPU chips with integrated transform, lightning, triangle setup/clipping and rendering engines capable of processing large number of polygons per second. GPU computing is the use of graphics processing units together with CPU to accelerate scientific, engineering and enterprise applications. The programs that use GPU's are usually partitioned into 2 parts, compute intensive and the rest of the code. The compute intensive part is made to run on the GPU's while the rest of the code is run on the CPU.

Compute Unified Device Architecture (CUDA) is the programming language used to program on the GPU chips. A GPU consists of a several sets of processors termed as Streaming

Multiprocessors (SM'S). All the SM's have their local memory (termed as **shared memory**) and **global memory** is used to communicate among several such SM's.

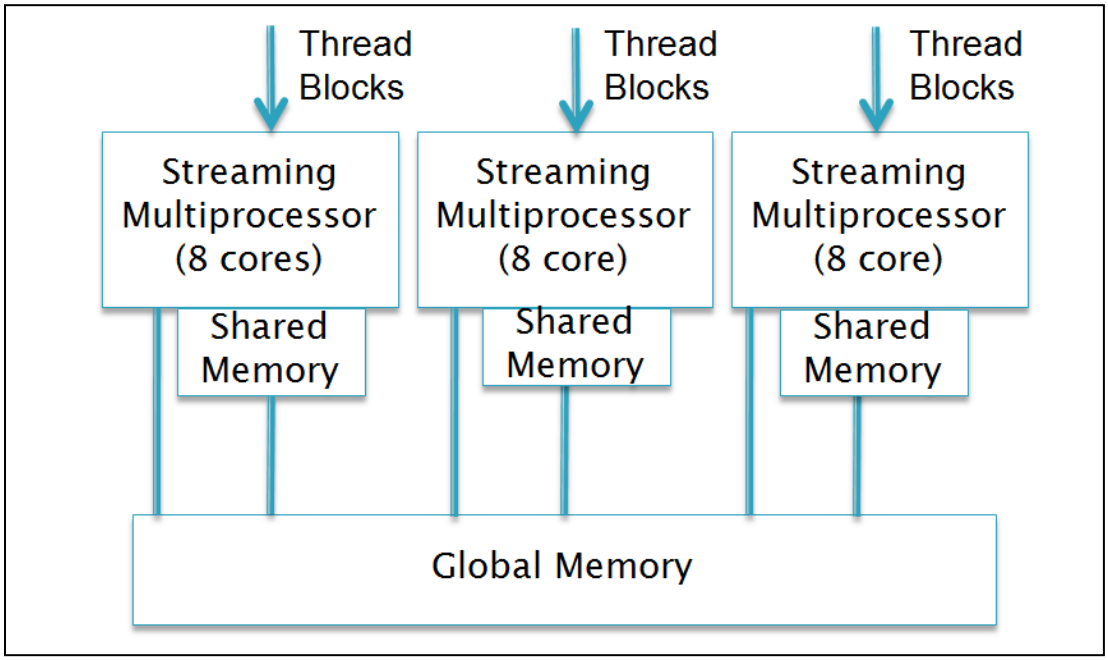

Figure 2.3.1 GPU Architecture

A general CUDA program can be organized as a set of host programs and a set of kernel programs. The host programs are executed on the CPU and the computation intensive kernel code programs are executed on GPU side. Before running the kernel code memory is allocated on the GPU and then the data structures are to be transferred onto the allocated memory (global memory) from the cpu memory.

2.4 Parallel Heap

As mentioned earlier a priority queue is a core component of discrete event simulation. A parallel heap is a priority queue from which 'r' elements can be extracted at a time where 'r' is the size of each node in the parallel heap. To be specific, a parallel heap with node capacity, r>=1 is a complete binary tree such that (i) each node contains r items (except for the last node which may contain fewer items and (ii) all r items at a node have values less than or equal to the

values of the items at its children. The parallel heap property ensures that the root node at any given time has the smallest r items (usually the highest priority items). Employing p processors, a parallel heap allows deletion of O(p) highest priority elements and insertion of O(p) new items each in O(log n) time where n is the size of parallel heap. The operations on the parallel heap are explained below.

A parallel heap of node size r keeps all the nodes sorted. During the start of insertion process the new items (sorted on their priority) to be inserted are merged with the root node and the smallest items are retained at the root node. The larger items are pushed down to the child nodes. This process is repeated at each node until the target node is reached (this is done based on the insertion path, refer [9] for additional implementation details). Similarly after a deletion of k(<=r) items from the root node we bring k items from the last node as the 'substitute' items and merge with the remaining (r-k) items at the root node. Since the heap property is disturbed at the root node we start a 'delete-update' process starting the root node. During the delete-update process we merge the current node with its two child nodes and keep the smallest 'r' items at the current node and the next smallest 'r' items at its left child if its largest item was bigger than that of the right child, else placing them at the right child. Finally the largest 'r' items are placed at the other child and a delete-update process is initiated at that child. This process is repeated until parallel heap property gets satisfied. The insert and delete-update are carried out in pipelined fashion for optimality purposes[15].

2.5 Simglobal algorithm

Simglobal is a traditional optimistic simulator [12] in which the state vector is partitioned into logical processes (LP's). Each LP is attached with queue to store its past states, input messages and output messages. Each executed message is attached with negative copies of its

outgoing messages and with the state of LP before a message is executed. When a new positive message is inserted into the channel of a destination lp, if a rollback is caused then the earliest message pointer is moved to this new message, the local clock and state are properly restored, and a copy of this message is inserted into the event queue. The actual rollback execution takes place lazily when a previous message is re-executed (termed as lazy cancellation).Please refer to [12] for complete details of the algorithm.

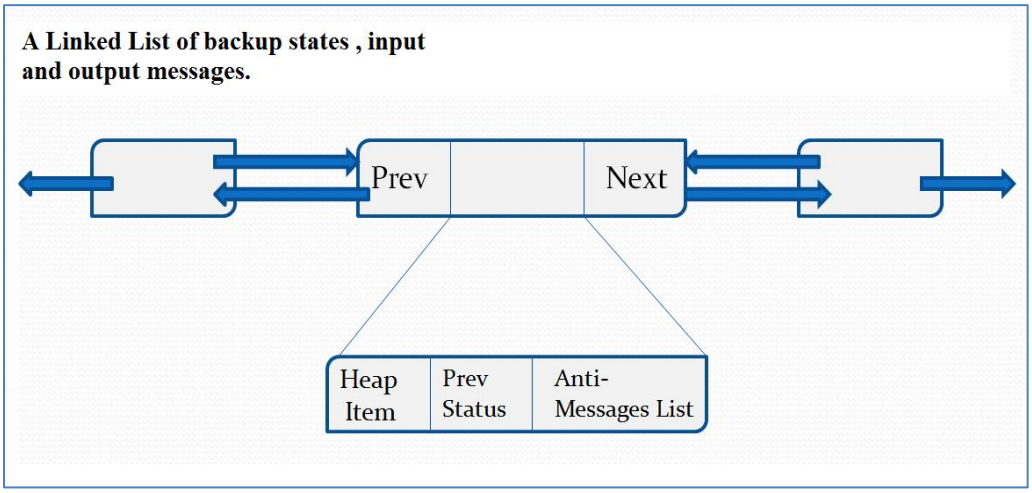

Figure 2.5 Message structure of Simglobal algorithm

3 RELATED WORK

Parallel Discrete Event Simulation (PDES) on General Purpose Graphics Processing Units (GP-GPU's) has been an interesting area of work since the evolution of GPU'S. Perumalla[2] has mentioned initial performance analysis of GPU's compared to CPU for a Diffusion Simulation which can be considered as both Time-Stepped as well as Discrete Event situation. He has also mentioned a hybrid algorithm where the minimum event time among all the update times is computed in parallel and is used as a time step to do a synchronous update of all elements in the simulation. Park and Fishwik[3] have developed a Discrete Event Simulation library based on CUDA using parallel Future Event List (FEL).Parallel operation of FEL is accomplished by dividing the FEL into sub-FEL's and assigning individual thread to each sub-FEL. They have used this library to test performance of queuing model simulations which are inherently time asynchronous. D. Chatterjee [4] has developed an event driven logic simulator on GP-GPU which is an important component of validation of digital designs. Similar works include parallelizing the verification of electronic design on GPU's by Alper Sen[5] (which is classified as Cycle Based Logic Simulation) and a Logic Simulator to verify the correctness of IC design by Bo Wang[7]. Other relevant works include a PDES scheme on GPU's[6] that enables cost and time efficient execution of large scale parameter studies and PDES on GPU's for SystemC, a modeling and Simulation language[8] . George Kunz work has dealt with both external (independent simulations) and internal parallelism (independent events).

Several mechanisms broadly classified as optimistic and conservative for Discrete Event simulation have been mentioned by fujimoto [1] in his paper in 1989. We have used the conservative algorithm for DES that is mentioned in a previous research paper by Prasad and Narsigh [10]. The foundation for conservative scheme was laid by R.E.Bryant (1977) and

independently by M.Chandy (1979) [11] . According to their works the physical systems that needs to be simulated can be partitioned into several components called logical processes (lp's).We have used a single global event queue for the purpose of load balancing in accordance with the Load balancing scheme mentioned in [12].The other foundational works for this paper are [13] where the IOWA logic simulator has been parallelized and syncsim[14] where a new algorithm syncsim has been demonstrated to be efficient just by storing one backup state.

4 IMPLEMENTATION

Implementation part is essentially composed of developing and assembling three components. The logic simulator (IOWA)[13] is the first component that was implemented specific to GPU's. The second core component was the Parallel Heap implemented earlier on GPU's and the code has been re-used with minor changes. Third component is the simulation itself along with the set of algorithms and data structures involved in the simulation process. The two basic steps involved in the simulation process are:

Simulation Phase (also referred as think phase) in which the deleted items from parallel heap are processed/simulated possibly generating new items.

Insert - Delete Phase in which the newly generated items are inserted to the parallel heap and the smallest r items are deleted for simulation in the next cycle.

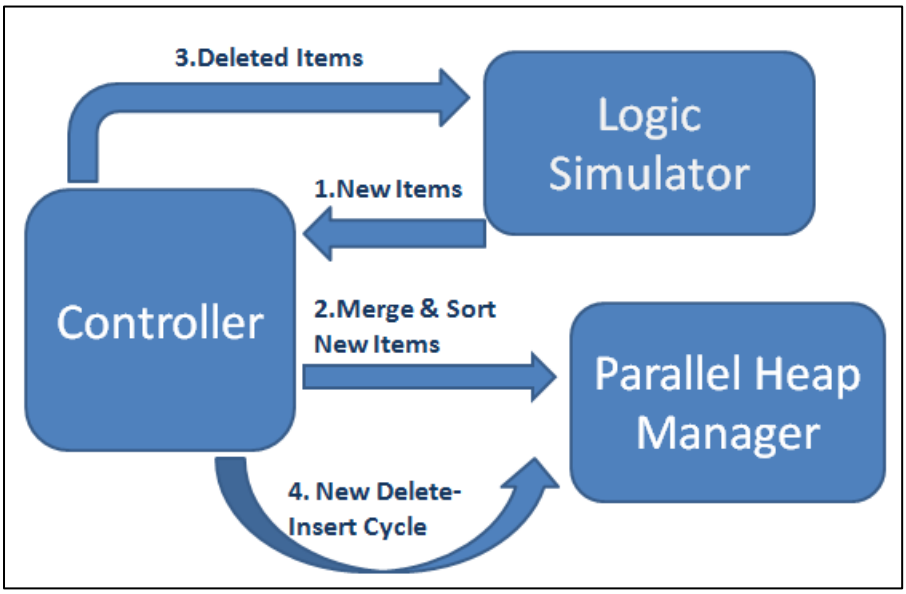

Figure 4.1 Architecture diagram for Logic Simulation

The following sections describe the basic components and other aspects of the implementation in detail.

4.1 Logic Simulator

The IOWA logic simulator of Douglas Jones is a gate-level fine-grained discrete-event simulator. It is capable of detailed gate-level simulation. The existing code and data structures in C for the formation of the logic simulation network has been transformed and re-designed to be suitable for GPU platform. One of the major requirements in the transformation was the array based data structures to represent the logical processes (LP's) and their attributes. Below is a list of attributes needed for the representation of the logic simulator which needed to be transformed.

```
/* constants representing the GATE Type */
#define AND                     0
#define OR                      1
#define XOR                     2
#define NOR                     3
#define NAND                    4

/* Structure defining a prototype for Logical Process (LP) */
typedef struct
    {
        int     service_time;           /* Service time of LP */
        int     elem_type;              /* Type of circuit element (GATE here) */
        int     status;                 /* Current output of LP (0 or 1) ?? */
        int     local_clock;            /* Local clock */
        int     in_channels_num;        /* No of input connections */
        int     out_channels_num;       /* No of output connections */
        int     InLPs[MAX_INPUTS];      /* Array of input LPs */
        int     OutLPs[MAX_OUTS];       /* Array of output LPs */
    } LP_TYPE;
```

Figure 4.2 Data structure for legacy simulation network.

The above structure represents some of the essential attributes of a logical process which has a logic gate represented by elem_type. This elem_type can hold any of the 5 values (0-4) where 0 representing AND operation, 1 representing OR, 2 representing XOR, 3 representing NOR, 4 representing NAND. Service time is the time taken by the logic gate to process the event. It can also be considered as the delayed induced on that LP before a new event is generated for its destination LPs (defined as the set of all the outgoing connections of the LP).

Array of these structures has been used in the legacy code developed in C with the number of LP's taken as input to the program. It is evident from the above figure that the LP structure is irregular and is not suitable for implementation on CUDA. Array based structures are preferred on CUDA platform. To be specific Structure of Arrays (SOA) is preferred over Array of Structures (SOA) on CUDA platform. The above structure has been changed to SOA as shown in the below diagram.

```
typedef struct
{
    int *ElementType;            /* Type of circuit element (GATE here) */
    int *ServiceTime;            /* Service times of LP */
    int *Status;                 /* Current output(status) of LP */
    int *In_channels_num;        /* No Of INPUT CHANNELS for every LP */
    int *Out_channels_num;       /* No Of OUTPUT CHANNELS for every LP */
    int *IN_Connections;         /* used to store the set of all INPUT connections for the LPs */
    int *OUT_Connections;        /* used to store the set of all OUTPUT connections for the LPs */
    int *IN_PrefixSum;           /* prefix sum of the IN_Connections */
    int *OUT_PrefixSum;          /* prefix of the OUT_Connections */
    int *outsize;                /* number of OUT connections for each LP */
    int *insize;                 /* number of IN connections for each LP */

    } NETWORK_TYPE;
```

Figure 4.3 New data Structures for Simulation Network

Each of the attributes 'ElementType', 'Service Time' etc are allocated linear memory (an array) during the initialization of the LP network. It is also necessary to mention the structure of the messages stored onto the parallel priority queue, parallel heap. It has four important components i.e item (timestamp of the message), from (Source LP number), to (Destination LP number), status (status of the message – 0 or 1). Below is a diagram showing the structure of the parallel heap message.

```
typedef struct
    {
        unsigned int item ; /* priority i.e. time-stamp of message */
        int from;           /* Source LP */
        int to;             /* Destination LP */
        int status ;        /* 0 or 1 */

    } PARHEAP;
```

Figure 4.4 Structure of Message in Safe-Window Algorithm

4.2 Concurrency Issues

When ‘r’ messages are deleted from the parallel heap it is possible that some or most of the messages may contain the same destination LP at which they should be simulated. So if while we are launching r CUDA threads on each of the messages independently, concurrent access to the status variable of an LP will induce problems. Hence there is a need for critical section in the kernel code. But critical sections are not supported well in the GPU kernel functions. One possible way to construct a critical section is using the atomic-locks available for CUDA. The use of atomic-locks is recommended in our programs especially when we are working on improving the execution time of the program.

4.2.1 Events Distribution Algorithm on GPU threads

Below is the Event/Message distribution algorithm followed to deal with the concurrency issues.

Step1: Sort the r messages based on destination LP numbers.

Step 2: Sort the messages again based on timestamp for each of the Destination LP number.

Step 3: launch a grid of ‘r’ cuda threads.

Step 4: Let thread id be tid.

For each thread, tid (except for tid=0) after picking up a message at index tid, if the Destination LP is same as the message at (tid-1) the thread will not perform any action.

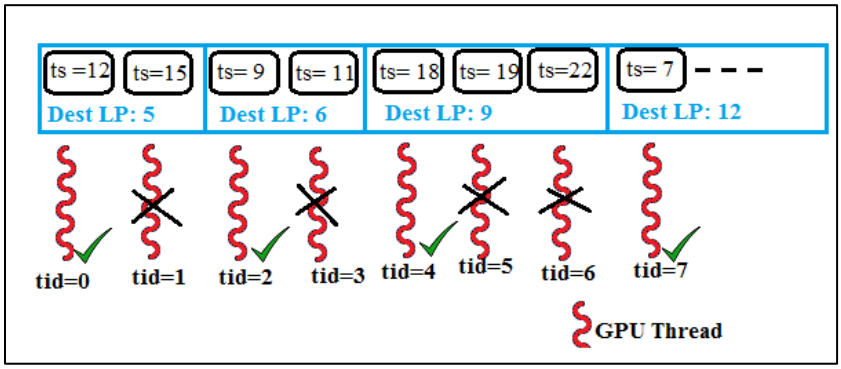

Figure 4.5 Events distribution on GPU threads

The above algorithm will make sure that only one thread is active per Destination LP and hence will solve the concurrency problems. The same algorithm has been used which insertion of newly generated messages onto the local queues of Destination LP of the Storage component (in the case of SyncSim Algorithm).

4.3 Safe-Window based Algorithm (Conservative Simulation)

As mentioned earlier safe-window algorithm is one of the conservative algorithms which focus on preventing the rollbacks instead of dealing with them later. The problem boils down to the task of finding a set of independent events/messages whenever r messages are extracted from the priority queue. Important steps in the algorithm are as follows

Step1: Get r elements from the parallel heap.

Step2: Extract the set of Independent Messages (Safe Messages) , Si, from the above set.

Step3: Store the rest of the messages in a temporary structure, UnSimulated.

Step 4: Let the Independent Message be simulated and new messages are produced which will be stored in NewMessages structure.

Step 5: Combine the messages in the UnSimulated and NewMessages and send the set to the parallel heap.

How are independent events determined?

For the safe window left boundary is the lowest time stamp among the r messages which is usually the Global Virtual Time (GVT). The right boundary is the GVT+ SERVICE_TIME_LOWER_BOUND. All the messages which fall within the range <GVT, GVT+LOWER_BOUND> are said to be safe to process concurrently and hence independent events.

4.4 SyncSim Algorithm (Optimistic Simulation)

SyncSim algorithm previously implemented by Prasad and Cao [14] was accomplished using C language. For the GPU based implementation of the same algorithm, major changes had to be done to the logic simulator and the priority queue. As mentioned in the previous section the data structures for Logic Processes have been changed to array based data structures. Apart from the array based structures additional fields have been defined on LP's of the network and also on the parallel heap structure. The parallel heap implementation needs to keep track of earliest messages at each LP and also the parent-child relationship had to be designed. An additional component named as 'Storage Component' was designed to take care of some of the above issues that arise in the parallel heap implementation for SyncSim algorithm. The following sections will provide further details about the implementation and the algorithm.

4.4.1 Storage Component

The Storage components stores all the events/messages involved in the simulation. It maintains a logical partitioning of local queue for each LP. Garbage collection is performed whenever an event gets older than the global virtual time (GVT) of the simulation. Whenever

new events are generated from the events being simulated the parent events have to remember the location of the child events that are generated. This is because the parent messages should be able to void (a state of the message indicating that it should be discarded) the children messages. Hence an index is needed for every event that exists in the storage. This index will also be used by the children events to access their parent events. One other requirement of algorithm is that only the duplicate copy of the original event will be sent to the priority queue. When the duplicate event copy comes to the logic simulator, its original copy is located first on the Storage component and then the original event is simulated. So for every event, a pointer to its original on the Storage component is required. The Storage component ideally acts as a Global queue of the events in the simulator that interferes with the flow of events between priority queue and the logic simulator. Below diagram gives a better idea about the model.

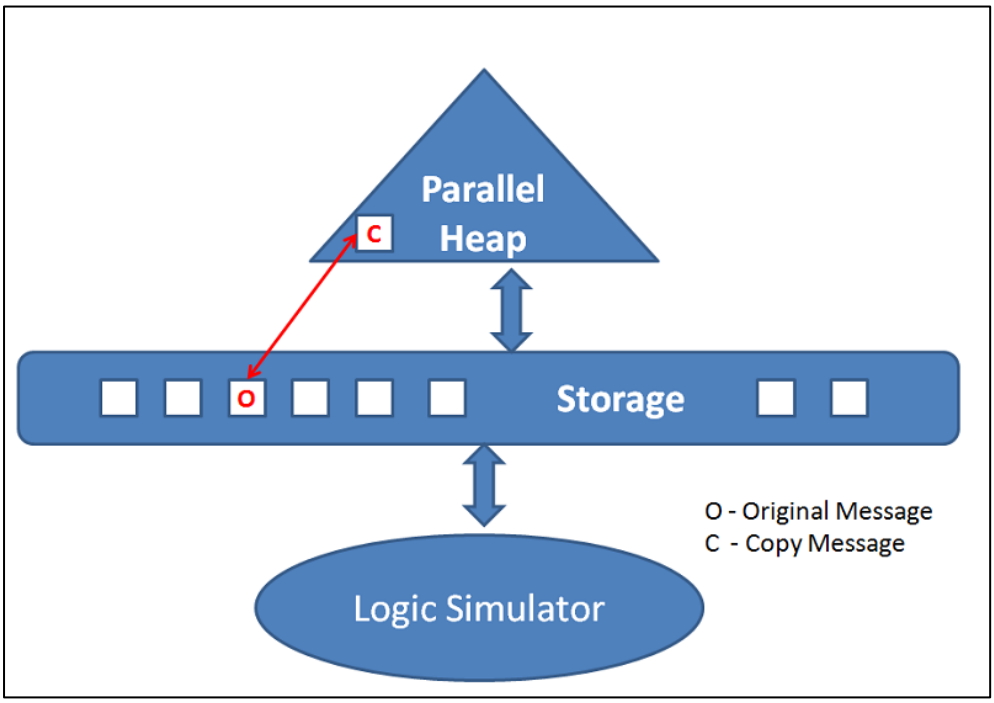

Figure 4.6 Storage Component Model

4.4.2 Parallel Heap Structure

The structure of the events is now changed to accommodate the extra fields to deal with the Storage component and the SyncSim algorithm. The other fields that are not discussed previously are ‘isvoid’ , ‘sim’ and ‘ch_offset’. The ‘isvoid’ field indicates that it has been voided by its parent and will be discarded soon according to the algorithm. It is initialized to 0 and will be set to 1 if the message is voided. The ‘sim’ field indicates that the message has previously been simulated and will be dealt with special condition in the algorithm. A value 1 indicates simulated already and 0 indicates that the message is yet to be simulated. ‘ch_offset’ is used during the insertion of the child messages into the Storage component. Initially when the messages are generated they will not have an index until they are inserted into the storage component (when is implemented as a separate routine called **insert()**). The child will carry the index of the parent in the ‘parent’ field until insert is executed. During the insert routine the ‘child1’ and ‘child2’ fields of the parent message are set appropriately by its children. In order to avoid the ambiguity that exists when a child message has to decide if it has to set the ‘child1’ or ‘child2’ of its parent the ‘ch_offset’ field is used. If ‘ch_offset’ is set to 1, the current message is the first child of its parent. If ‘ch_offset’ is set to 2, the current message is the second child of its parent and so on. Below is the complete structure of a message in SyncSim.

```
typedef struct
   {
        unsigned int item ;
        int from;
        int to;
        int status ;
        int child1;     /* pointer to the first child in storage  */
        int child2;     /* pointer to the second child in storage */
        int index;      /* pointer to itself(original copy in the Storage component) */
        int isvoid;     /* indicates if the message needs to be discarded (0 or 1) */
        int sim;        /* indicates if the message is already simulated  (0 or 1)*/

        int parent;     /* Index of the parent in the Storage */
        int ch_offset; /* indicates if it is a child1 or child2 of its PARENT */
    } PARHEAP;
```

Figure 4.7 Message structure for SyncSim Algorithm

4.4.3 Algorithm details

The SyncSim algorithm functions on the basis of few important parameters of the message. The first parameter is ‘isvoid’. If the message is voided by its parent already then it first voids its own children and then it gets discarded from the storage without any simulation. The second parameter is ‘sim’. If there exists an earliest message at the destination LP for a current message then a decision is made weather the current message has to go back to the priority queue or if it has to replace the earliest message with or without rollback. This decision is based on the timestamps of the current message and the earliest message. Whichever message has the least timestamp stays and the other message have to go back to the priority queue. A rollback might happen if the earliest message is already simulated and if it has to be replaced. The actual rollback and discarding of the child messages happens (lazy cancellation) when the message gets re-executed in the future cycle. Below is the complete algorithm.

```
Let m` be the earliest message at LP i

Simluate(message m at LP i)
 {
   if(m  voided)
   {
     void children of m ;
     discard m;
  }
  else if(m` is simulated before)
       {
          if(m` is voided)
           { - void descendants
              -discard m`
              -rollback lp
              -execute m
            }
          else if(m' <=GVT )  { discard m`  and execute m }
          else if(m < m` )      { rollback lp    and execute m
                                  m` goes back to GQ
                                }
           else m goes back to GQ
       }
   else if (m' is not simulated  yet)
   {
        if(m` voided)  { discard m` and execute m }
        else if ( m > m`)  m goes back to GQ
        else  { execute m }
   }
  else  execute m;
}
```

Figure 4.8 Algorithm for Simulate () function.

```
Execute(message m at Lpi)
 {
   step1:
      - backup state of Lpi
      - Simulate m;
      - m becomes lp's earliest(m').

   step2:
       if(m==GVT) discard m;

   step 3:
       if(m not simulated before)
            insert children of m at their dest LP's;
       else  // m simulated before
        {  if(new children of m are different from old)
             {    - void old children     //lazy cancellation
              }
            -insert children into dest LP

         }
 }
```

Figure 4.9 Algorithm for execute () function

```
Insert message(LP i)
{
   if(Lpi is empty) m becomes its earliest( m`)
   else if (m` voided)
            {
              if(m` simulated before){ void children of m`. Rollback LP}
              -m replaces m1 //  discard m` and m is the new m`
            }
    else if (m`<=GVT AND m` simulated before)
            m replaces m`
    else if (m < m` AND m` not simulated)
          m replaces m`
    else if(m < m` AND m` executed before)
          {
             Rollback LP;
             m becomes new earliest(m`)
             m` goes  into GQ.
          }
}
```

Figure 4.10 Algorithm for insert () function.

For every simulation cycle the total number of new messages that should be sent to the priority queue is the sum of messages from several parts. The first part comes from the messages which are not simulated because an earliest message exists at the destination which is yet to be simulated and has a lesser timestamp. The second part comes from the set of earliest messages which gets replaced by a new message with a lower timestamp at that LP (termed as message bump).Third part comes from the set of newly generated messages from the messages being simulated. The fourth part comes from the message bumps that happens during the insertion of the newly generated messages onto the Storage at their destination LP's local queue. For the

current implementation we made sure that the total number of newly generated messages will be less than ‘2r’ (r is the capacity of node in the parallel heap).This constraint is because the current implementation of parallel heap does not support insertion of ‘2r’ messages.

4.5 Parallel Heap

The code for priority queue, parallel heap, which has originally been implemented by Xi He[9] has been used in this research work with minor changes. The message structure has been update throughout the program to reflect the new message structure.

5 EXPERIMENTS AND RESULTS

The logic simulation performed on GPU is compared with other algorithms for performance measures. The simulation is done by varying several parameters like No of LP's, r (capacity of parallel heap node), Number of messages simulated (also with total number inserts and deletes in the parallel heap), Think time of the simulator etc. Here think time is measured as number of for-loops in the code to simulate processing time on real time environment. Think time indicates the grain size of the computation. Higher value of think time indicates courser grain and lower value of the think time indicates the fine grain computation.

```
void think (int thinktime)
 {
     int i;
    for (i=0; i<thinktime; i++);
 }
```

5.1 Performance of Safe Window Algorithm

Safe window algorithm has been analyzed for its performance with respect to varying think time and heap node size, r. In both the cases the behavior of the logic simulator and parallel heap were similar to the previous research results on parallel heap[9].

5.1.1 Comparison of Safe Window algorithm with Sequential Heap

Think time is varied starting with 2000 for loops incremented by 2000 every execution. Sequential heap execution time was growing linearly with the think time while the Safe Window algorithm has almost a constant execution time over varying think time. In other words it can be stated that the simulation speed is stable with different computation loads.

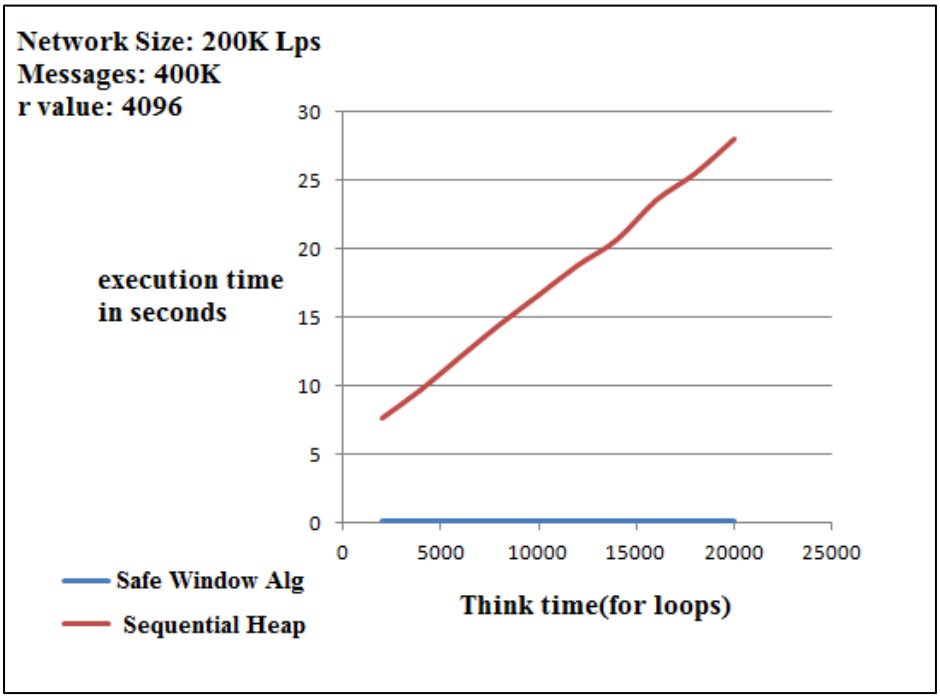

Figure 5.1.1 Comparison of Safe Window and Sequential Heap Algorithms

5.1.2 Performance of Conservative Algorithm with respect to r value

Safe Window (Conservative) algorithm has shown increase in the simulation speed with respect to increase in heap node size(r). A network of 100000 LP's has been generated and approximately 200000 messages have been processed. The stopping criteria for the simulation is based on the total number of inserts and deletes in the parallel heap.

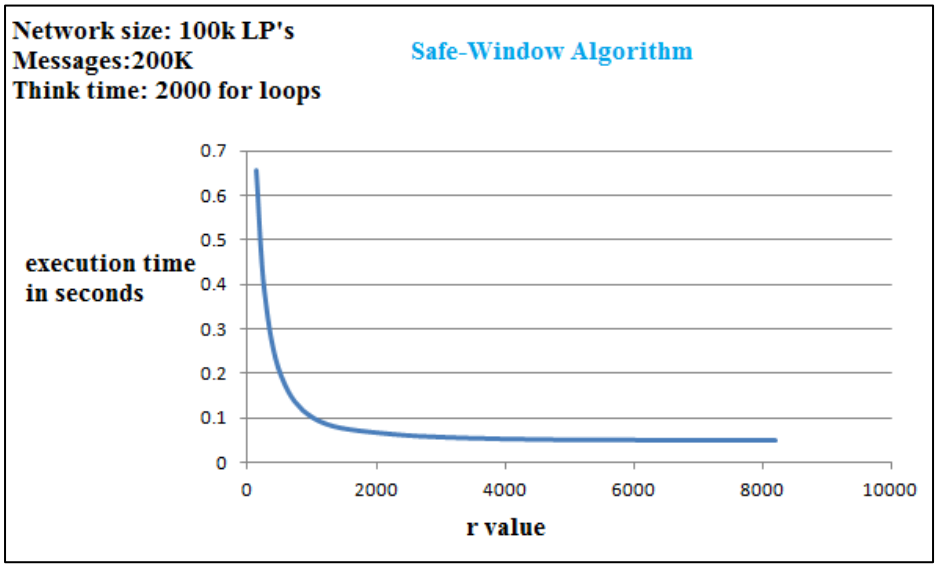

Figure 5.1.2 Effect of r value on Conservative Algorithm.

It can be inferred that the higher number of items in a heap node the better is the performance of the simulation. Logically, if the heap is processing large number of items at a time there would be a decrease in the simulation cycles required to process the items. The increase in the simulation speed with the increase in r is consistent with the previous research on parallel heap with logic simulation.

5.2 Performance of SyncSim Algorithm

SyncSim algorithm has been tested for other parameters like think time, heap node size(r) and rollback counts.

5.2.1 Comparison of SyncSim Algorithm with Sequential Heap

It has been indicated earlier that a sequential heap algorithm execution time grows linearly with time. SyncSim algorithm showed very minute fluctuations with respect to think time and was almost has constant execution time. This behavior is very similar to the behavior of the conservative algorithm where different computational loads does not affect the simulation speed.

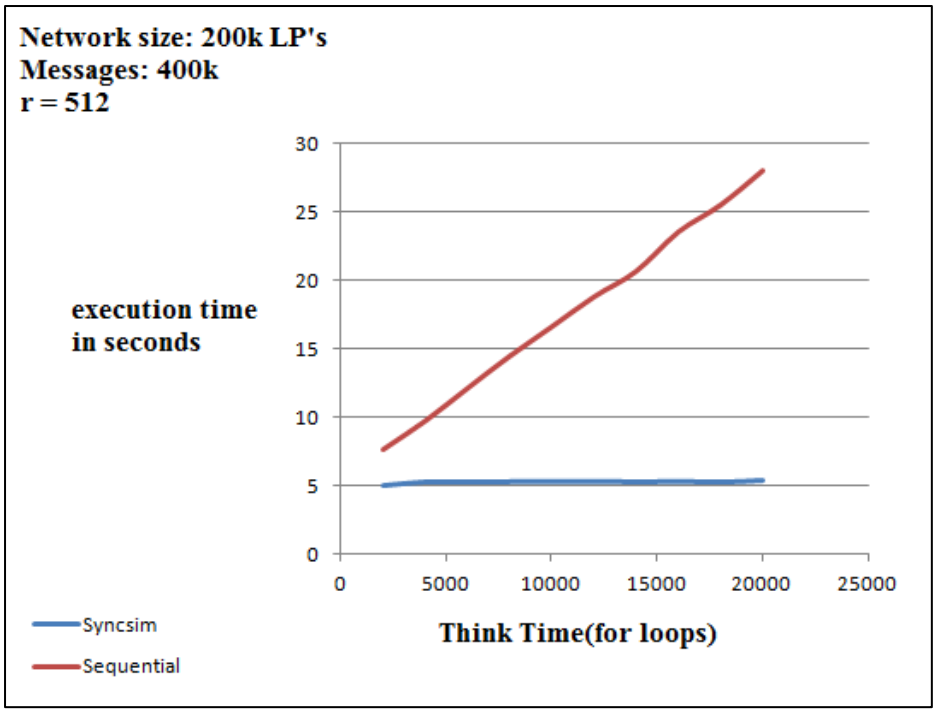

Figure 5.2.1 Effect of r value on Syncsim Algorithm.

5.2.2 Varying heap node size on SyncSim

A network of 32000 nodes has been simulated with a think time of 2000 for loops and 100000 messages processed on parallel heap. SyncSim algorithm showed decrease in simulation time with increase in heap node size (r) until a certain heap node size value. It has been observed that the simulation time fluctuates for large networks. This can be attributed to the several branching phenomenon in the syncsim algorithm which is not suitable for cuda programming on GPU's. This has to be researched further to obtain a consistent speedup with increasing r for larger network on syncsim algorithm.

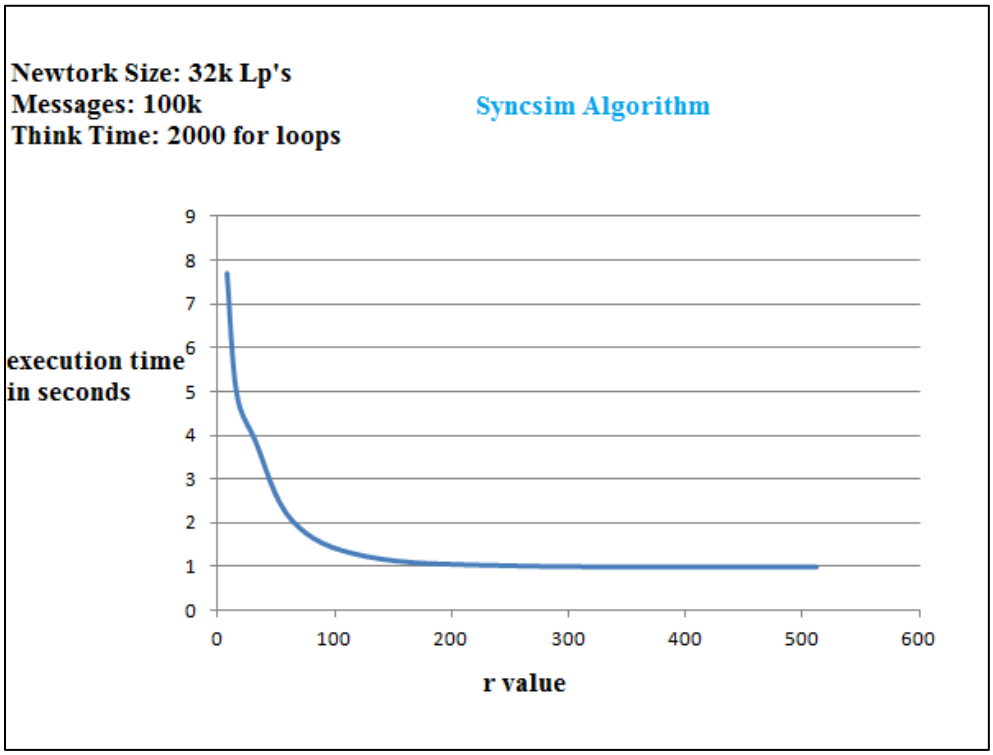

Figure 5.2.2 Effect of r value on SyncSim Algorithm.

5.2.3 Effect of rollback count on heap node size

The number of rollbacks are expected to increase with increase in the parallelism in the network. As the size of heap node is increased we can see an increase in the number of rollbacks on the network. When large number of messages are processed in a single iteration the probability of a message producing child messages at the destination nodes which cause a rollback increased. i.e the probability for causality error increases. On a network of 40000 LP's and 100000 messages simulated it can be observed that no rollbacks exists for initial values of r and they increase gradually with increase in the r value of the parallel heap.

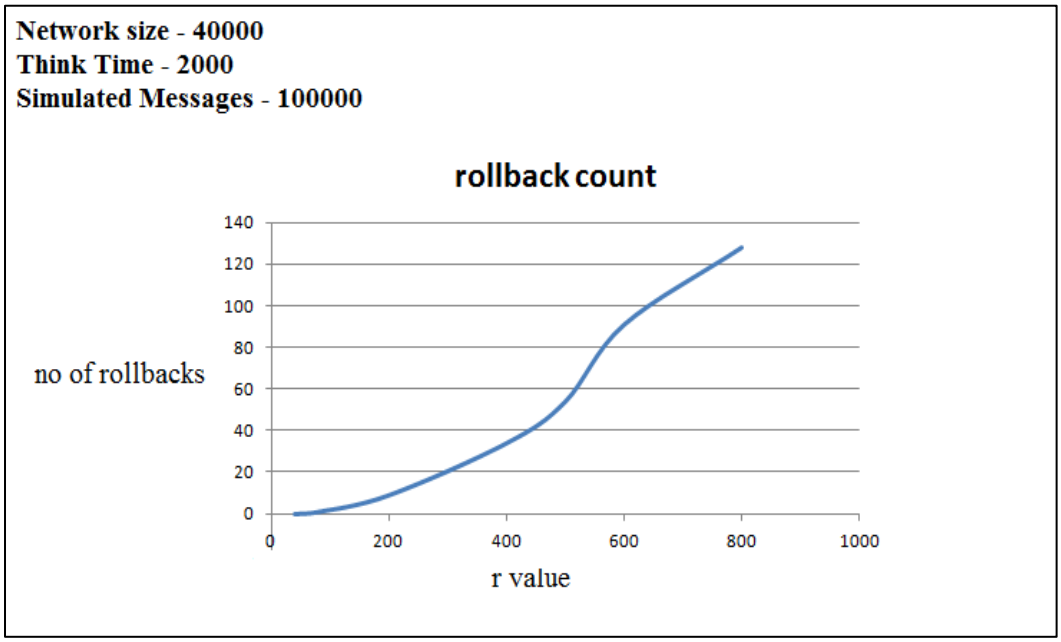

Figure 5.2.3 Effect of r value on Rollback count of Syncsim

5.3 A comparision of the algorithms

All the three algorithms: Sequential, Safe-Window (naive) and SyncSim are analyzed for performance with respect to varying think time. The network used is a 250000 LP network with 400000 messages processed on the heap for each algorithm. Think time is varied starting with

2000 for loops incremented by 2000 every execution. Sequential heap execution time is growing linearly with the think time. The other two algorithms on the gpu had almost a constant execution with the increasing think time. The Safe-window algorithm is found to be better than the other 2 algorithms.

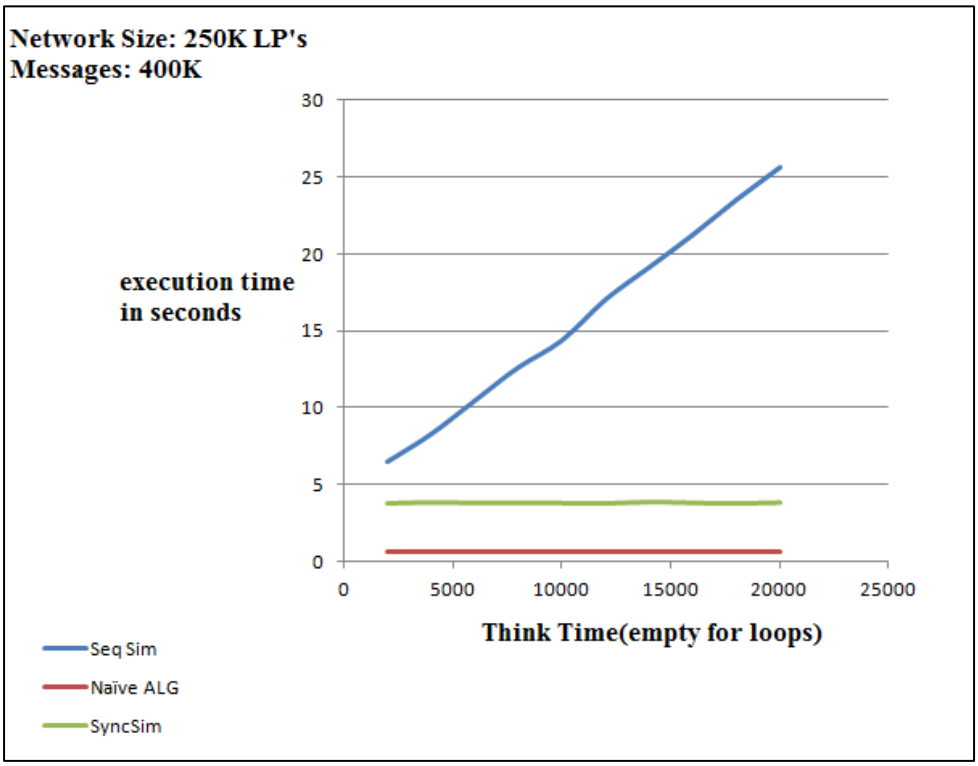

Figure 5.3 Comparison of Algorithms based on think time

6 CONCLUSION

Discrete Event Simulation on Graphics Processing Units is very challenging as it involves dynamic data structures as opposed to the array based structures that are more suitable for CUDA platforms. The implementation of storage manager for the syncsim algorithm is essentially a dynamic storage implemented specifically for CUDA platforms. The experiments conducted on the conservative and syncsym algorithms have given results very similar to the previous research on parallel heap data structures. It has been observed in the experiments that the safe window algorithm can be altered to generate a set of complete 'r' independent events in every iteration after a few initial iterations. For this to happen the service time window has to be decreased and Out degree of the network has to be increased. This feature can be used to model a new hybrid algorithm based on safe -window and an optimistic (say syncsim) algorithm to behave dynamically over the changing parameters of the network. This work can also be extended for a generic network with a dynamic in/out degree and also for agent based simulation on GPU's. Several memory hierarchies that exists on the GPU can be exploited in the algorithms and can be optimized further to get greater speed on Simulation. The storage manager used in the program can be re-used in modeling other GPU programs which require dynamic storage management.

REFERENCES

[1] R.M Fujimoto 1990. Parallel Discrete Event Simulation. Communications of the ACM.

[2] K. S. Perumalla, 2006. Discrete-event execution alternatives on general purpose graphical processing units (GPGPUs). PADS ’06:Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation, pp. 74–81, IEEE Computer Society, Washington, DC.

[3] H. Park and P. A. Fishwick 2010. A GPU-Based Application Framework Supporting Fast Discrete-Event Simulation. Simulation, vol. 86, pp.613–628.

[4] Debapriya Chatterjee, Andrew DeOrio and Valeria Bertacco. Event-Driven Gate-Level Simulation with GP-GPUs. DAC 2009, San Francisco.

[5] Alper Sen 2010 . Parallel Cycle Based Logic Simulation using Graphics Processing Units. International Symposium on Parallel and Distributed Computing (ISPDC).

[6] Georg Kunz, Daniel Schemmel 2012. Multi-level Parallelism for Time- and Cost-efficient Parallel Discrete Event Simulation on GPUs. IEEE workshop on Principles of Advanced and Distribution simulation.

[7] Bo Wang 2010. Distributed Time, Conservative Parallel Logic Simulation on GPUs. DAC '10 Calfornia.

[8] Mahesh Nanjundappa 2010. SCGPSim: A Fast SystemC Simulator on GPUs.

DAC '10 Asia and South Pacific

[9] Xi He, Dinesh Agarwal, and Sushil Prasad 2012. Design and Implementation of a Parallel Priority Queue on Many-core Architectures. High Performance Computing (HiPC) 2012.

[10] Sushil Prasad, Narsingh Deo 1991. An efficient and scalable parallel algorithm for discrete-event simulation. Proceedings of Winter simulation Conference 1991.

[11] R. E. Bryant 1977. Simulation of packet communication architecture computer systems. Tech. Rep. NIT-LCS-TR-188. Massachusetts Inst. Tech., Cambridge, MA. Chandy, K. hI., V. Holmes, and J . Misra. 1979. Distributed simulation of networks. Computer Networks 3 (Feb.): 10513.

[12] Sushil Prasad, and B. Naqib 1995. Effectiveness of Global Event Queues in Rollback Reduction and Load Balancing. Proceedings of the 9th Workshop on Parallel and Distributed Simulation, Lake Placid, NY, pp. 187-190.

[13]Sushil Prasad , Nikhil Junakar 2000. Parallelizing a Sequential Logic Simulator using an Optimistic Framework based on a Global Parallel Heap Event Queue: An Experience and Performance Report. Parallel and Distributed simulation (PADS) 2000.

[14] Sushil Prasad , zhiyong Cao 2003. syncsim: a synchronous simple optimistic simulation technique based on a global parallel heap event queue. Proceedings of Winter Simulation Conference 2003.

[15] Sushil Prasad , I. Sagar Sawant 1995. Parallel Heap: A Practical Priority Queue for Fine-to-Medium-Grained Applications on Small Multiprocessors. Proceedings of 7th IEEE Symposium on Parallel and Distributed Processing (SPDP)1995.

APPENDICES

Appendix A

Important structures from source code of syncsim algorithm

```
#define MAX_INSERTS          7     /* No of simultaneous inserts in heap  */
#define MAX_OUTS             1     /* # of out channels (except sources)  */
#define MAX_INPUTS           1     /* # of in channels */
#define MAX_SRC_OUT          1     /* # of out channels from sources      */
#define MAX_LPS              10000000
#define MAX_NEWM             2
#define MAX_SERVICE_TIME     15
#define NO_OF_GATES          5

#define TRUE          1
#define FALSE         0
#define AND           0
#define OR            1
#define XOR           2
#define NOR           3
#define NAND          4

typedef struct
  {
        unsigned int item ;     /* unsigned -- node priority i.e. time-stamp of message */
        int from;
        int to;
        int status ;   /* 0 or 1 */
        int child1;    //pointer to the child1 in storage
        int child2;  //pointer to the child2 in storage
        int index; // pointer to its original copy in the Storage component
        int isvoid; //isvoid will be 1 in case of voided by parent mesg and 0 if not
                    //this is specific to syncsim algo
         int sim; // zero if not simulated yet, else 1

        int parent; // INDEX of the parent in the Storage
       int ch_offset; //indicates weather it is a child1 or child2 of its PARENT
        } PARHEAP;

typedef struct
{
        int *ElementType;          /* Type of circuit element (GATE here) */
        int *ServiceTime;          /* Service times of LP */
        int *Status;               /* Current output(status) of LP */
        int *In_channels_num;      /* No Of INPUT CHANNELS for every LP */
        int *Out_channels_num;     /* No Of OUTPUT CHANNELS for every LP */
```

```
        int *IN_Connections;        /* used to store the set of all INPUT connections */
        int *OUT_Connections;   /*  used to store the set of all OUTPUT connections  */
        int *IN_PrefixSum;     //prefix of the IN_Connections
        int *OUT_PrefixSum;    //prefix of the OUT_Connections
        int *outsize;
        int *insize;
                //new structures for syncsim
        int *BackupState;
        int *earl; //pointers for earliest mesg at each LP
        int *localQcounts;

        int NoOfLPs;
        int NoOfSourceLPs;
        int r;
       int buffer_size;

} NETWORK_TYPE;
```

Simulate kernel of syncsim algorithm

```
__global__

void Simulate_kernel(PARHEAP *Messages, NETWORK_TYPE network_gpu ,  PARHEAP*

Storage,  PARHEAP *outmesgs,  PARHEAP *bump1,  int *unSim,  int GVT,  int *rollbackCount)

 {

        int tx=threadIdx.x;

        int bx=blockIdx.x;
        int base=bx*blockDim.x; //2
        int tid = tx+base;
        int sync_flag = 0;

        int tm, tm1;

        int r = network_gpu.r;

        if(tid >= r)  //basic condition
        {
          sync_flag =1;
          return; //return if tid is out of bounds
        }


    int destination =  Messages[tid].to;
    int earlIdx = network_gpu.earl[destination];
```

```
   //if the dest is same as prev mesg exit. second base condition
        if(  tid>0 && destination == Messages[tid-1].to )
                {
                  sync_flag = 1;
               }


 if(sync_flag ==0)
  {
         while( tid<r && destination == Messages[tid].to )
        {
             int sid  = Messages[tid].index;

                    tm1 = Storage[earlIdx].item; //earliest message time stamp
                    tm = Storage[sid].item;

           //CASE 1
            if( Storage[sid].isvoid == 1 )  // 1 indicates message voided
              {

                    //void children of m
                 void_children(sid,        Storage);          //will also make child.parent =-1

                              //discard m
        Storage[sid].item = 0; //discarding the mesg

        network_gpu.localQcounts[destination]--;           //decreasing local Q count

                      //if m is the earliest clear earl mesg pointer
         if(Storage[sid].index ==  earlIdx)
            {

network_gpu.earl[destination] =
getEarliest(Storage,network_gpu,destination);  // need to calc earl mesg here

                                        }
                    }

                    //CASE2
 else if(Storage[earlIdx].sim == 1 && Storage[earlIdx].item!=0) //if lp's earl is simulated already
       {
                            //if m' is voided
                      if(Storage[earlIdx].isvoid == 1)
                              {
                            void_children(earlIdx, Storage); //will also make child.parent =-1

                    Storage[earlIdx].item = 0;   //discarding the mesg m'
                  network_gpu.earl[destination] = 0; //clearing the earliest mesg pointer

             network_gpu.localQcounts[destination]--; //decreasing local Q count
```

```
                    //rollback to earlier state from backup
                    network_gpu.Status[destination] =network_gpu.BackupState[destination];
                    network_gpu.BackupState[destination] =-1; //empty the backup state

        //execute m
        execute(sid,tid,network_gpu,Storage,outmesgs,GVT,rollbackCount); //device method
                                        }
        else if(tm1 <= GVT)
                {

                  //garbage collecting the mesg m'
                 Storage[earlIdx].item = 0;

                 //also setting child.parents to -1
               if( Storage[earlIdx].child1 > 0) Storage[Storage[earlIdx].child1].parent = -1;
              if( Storage[earlIdx].child2 > 0) Storage[Storage[earlIdx].child2].parent = -1;

             network_gpu.earl[destination] = 0; //clearing the earliest mesg pointer
            network_gpu.localQcounts[destination]--;  //decreasing local Q count

        //execute m
execute(sid,tid,network_gpu,Storage,outmesgs,GVT,rollbackCount);

                 }
             else if(tm < tm1)
               {
                  //rollback to earlier state from backup
             network_gpu.Status[destination] = network_gpu.BackupState[destination];
             network_gpu.BackupState[destination] =-1; //empty the backup state


               //execute m
execute(sid,tid,network_gpu,Storage,outmesgs,GVT,rollbackCount); //device method

             //m' goes back to GQ
              DCopyPHItems(&bump1[tid], &Storage[earlIdx]); //destination , source

                 }
             else
               {
                                //   m goes back to GQ
                                 unSim[tid] = 1;
                 }
        }

            //CASE 3
              else if(Storage[earlIdx].sim == 0 && Storage[earlIdx].item!=0)//If lp's earl is NOT
simulated already and earl mesg does exist
                        {
                           //if m' is voided
              if(Storage[earlIdx].isvoid == 1)
```

```
                {

                        //discarding the mesg m'
                        Storage[earlIdx].item = 0;
                        network_gpu.earl[destination] = 0; //clearing the earliest mesg pointer
                        network_gpu.localQcounts[destination]--;  //decreasing local Q count

                        //also setting child.parents to -1
                if( Storage[earlIdx].child1 > 0) Storage[Storage[earlIdx].child1].parent = -1;
                if( Storage[earlIdx].child2 > 0) Storage[Storage[earlIdx].child2].parent = -1;


        //execute m
execute(sid,tid,network_gpu,Storage,outmesgs,GVT,rollbackCount);
                                        }
        else if(tm >tm1)
                {
                                //   m goes back to GQ
                                     unSim[tid] = 1;
                 }
        else
         {

                //just change the earlmesg pointer, m' will be executed later
                 network_gpu.earl[destination] = 0;

        //execute m

execute(sid,tid,network_gpu,Storage,outmesgs,GVT,rollbackCount);
          }
         }

        //CASE 4
           else    //lp's earliest mesg is empty
                        {

                          //execute m
                           execute(sid,tid,network_gpu,Storage,outmesgs,GVT,rollbackCount);
                        }

                        tid++; //this is  just a local increment
            }//while
           }

        }
```

Execute kernel of syncsim algorithm

```
__device__

void execute(int sid,  int tid,  NETWORK_TYPE network_gpu, PARHEAP* Storage, PARHEAP

*outmesgs,  int GVT, int *rollbackCount)

{
 int destination = Storage[sid].to;
 int tm = Storage[sid].item;
 int k;
 int timeofoutmesg;
 int newstatus;


 //backup dest lp staus
 network_gpu.BackupState[destination] = network_gpu.Status[destination];

 //m becomes lp's earliest message
  network_gpu.earl[destination] = sid;

 //simulate m
 switch(network_gpu.ElementType[destination])
        {
          case AND :
            network_gpu.Status[destination] &=Storage[sid].status;
            break;
          case OR  :
            network_gpu.Status[destination] |=Storage[sid].status;
            break;
          case XOR :
            network_gpu.Status[destination] ^=Storage[sid].status;
            break;
          case NOR :
            network_gpu.Status[destination] |=Storage[sid].status;
            network_gpu.Status[destination] = !network_gpu.Status[destination];
            break;
          case NAND :
            network_gpu.Status[destination] &=Storage[sid].status;
            network_gpu.Status[destination] = !network_gpu.Status[destination];
            break;
         }

      timeofoutmesg =Storage[sid].item + network_gpu.ServiceTime[destination];
       newstatus = network_gpu.Status[destination];

  if(GVT == tm)
          {
           //discard m
           Storage[sid].item = 0; // its technically garbage collected
```

```
                network_gpu.earl[destination] =
        getEarliest(Storage,network_gpu,destination); //need to calc earl mesg here


                network_gpu.localQcounts[destination]--;  //decreasing local Q count

         Storage[sid].index= -1; // to tell the childs that its parent is already garb. collected.

                 }

                int cur_source = destination;

                 //generating new messages
            if(network_gpu.Out_channels_num[cur_source]!=0)
                {
               //  printf(" \n (Generating new messages for item =%d) ",tm);

                            int newdest;
                int index = network_gpu.OUT_PrefixSum[cur_source];
                             for(k=0;k < network_gpu.Out_channels_num[cur_source];k++)
                                     {
                                       newdest = network_gpu.OUT_Connections[index++];

                                    CreateMsg_device(cur_source, newdest, timeofoutmesg, newstatus,
&outmesgs[MAX_NEWM*tid+k],Storage[sid].index,k+1); //ch_offset will be 1 for child1

                                     }
                            }

            //if the message already simulated check if the new mesgs are different
                if(Storage[sid].sim ==1)
                 {

                  int ch1_index = Storage[sid].child1;
                  int ch2_index = Storage[sid].child2;

                  //LAZY Cancellation
                   if( Storage[ch1_index].item!=0 && (Storage[ch1_index].status != newstatus) )
                     //if child1 exists and if the child has different status make it void
                    {

                  //--- This is the only race condition in the code.
                   atomicAdd(rollbackCount, 1);

                          Storage[ch1_index].isvoid =1;

                if( Storage[ch2_index].item!=0 && (Storage[ch2_index].status != newstatus) ) //if child2
exists and if the child has different status make it void
                          {
                                    Storage[ch2_index].isvoid =1;
                                }
```

```
            //we do insert of the new messages in this case
            //so we are not doing anything as we have scheduled the inserts as a separate step
                        }
              else
            {
        //we have to nullify the new messages as there is no change in the status  of the child messages

          if(network_gpu.Out_channels_num[cur_source]!=0)
                {
                              for(k=0;k < network_gpu.Out_channels_num[cur_source];k++)
                                    {
                 outmesgs[MAX_NEWM*tid+k].item =0; //discarding the mesgs generated earlier

                                    }
                            }
                        }
                }// if already sim
        else //if not simulated earlier we should just insert the new messages
                {
                  //we are not doing anything as the messages generated will be processed in the next
step for insertion.

                 Storage[sid].sim =1;
                }
```

Insert kernel of syncsim algorithm

```
        __global__

        void Insert(NETWORK_TYPE network_gpu ,PARHEAP * Storage ,PARHEAP * Messages, int
size, int * localQ_overflow, PARHEAP *bump2, int GVT)
         {
          /**
          * First Do the inserts and generate 'indexes' for self in Storage
          * Set the child components of the parent element.
          * then perform the insert algorithm in the paper
          **/

                int tx=threadIdx.x;
                int bx=blockIdx.x;
                int base=bx*blockDim.x;
                int tid = tx+base;
                int org_tid = tid;

                int sync_flag = 0;

                int tm, tm1, earl;
```

```
                if(tid >= size)  //basic condition
                 {
                  sync_flag =1;
                  return; //some of the threads will return from here. syncthreads will not work
                 }

            int destination =  Messages[tid].to;
            int earlIdx = network_gpu.earl[destination];

            //if the dest is same as prev mesg exit. second base condition
                if( tid>0 && destination == Messages[tid-1].to )
                        {
                          sync_flag = 1;
                    }

          //STEP1: Doing the inserts into the Storage
          if( sync_flag == 0 )
              {
                    int current;
                int endIndex = destination*network_gpu.buffer_size + network_gpu.buffer_size -1; //end
index of the local queue

        while( tid<size && destination == Messages[tid].to )
              {
                        tm1 = Storage[earlIdx].item; //earliest message time stamp
                        tm  = Messages[tid].item; //curr mesg time

                        current  = destination*network_gpu.buffer_size; //starting index of local queue

                        if(network_gpu.localQcounts[destination] == network_gpu.buffer_size)
                        {
                                *localQ_overflow =1;   //flag to indicate overflow of local queue
                         }

                        //else we look for an empty slot
                        while(Storage[current].item!=0 && current<= endIndex ) //until we find an
empty slot is empty
                          {
                           current++;
                          }


        //set the index component on Message(which goes to parheap) before pushing it onto Storage
                                Messages[tid].index = current;

                                int parent = Messages[tid].parent;

                       //setting the child component of its parent based on its child_offset available

                        if(Messages[tid].ch_offset == 1 && parent!=-1 ) //make sure that parent is not
already garbage collected - in which case the parent component of the childs is set to -1
```

```
                    {
                        Storage[parent].child1 = current;
                    }
            else if(Messages[tid].ch_offset == 2 && parent!=-1 )
             {
                    Storage[parent].child2 = current;
             }

        //copy the message onto Storage which becomes the Original Copy

                DCopyPHItems(&Storage[current], &Messages[tid]); //destination , source
                network_gpu.localQcounts[destination]++; //increase the local Q count


             //@@@Actual Insert algorithm @@@@//

             // if lp i is empty m becomes its  earliest //Case1
            if(network_gpu.localQcounts[destination] == 1) //localcount = 1 implies that the lp was
empty before calling the insert func.
                 {
                    //we need to assign the earliest mesg pointer in this case
                        network_gpu.earl[destination] = current;
                }

                    //m' is voided    //Case2
                else if(Storage[earlIdx].isvoid == 1 && Storage[earlIdx].item!=0)
                    {

                    if(Storage[earlIdx].sim==1)
                     {

                      //void children of m' ; Rollback lp
            void_children(earlIdx, Storage);
            network_gpu.Status[destination] = network_gpu.BackupState[destination];
             network_gpu.BackupState[destination] =-1; //empty the backup state

                     }

                    // discard m' and  make m as earl  (either way)
                     Storage[earlIdx].item = 0;
                     network_gpu.earl[destination] = current;
                     network_gpu.localQcounts[destination]--;
                    }

            //Case3: tm1<=GVT and m1 simulated before
            else if(tm1!=0 && tm1 <=GVT && Storage[earlIdx].sim ==1 ) // FYI: tm1 is same as
Storage[earlIdx].item
             {

                      //m replaces m1;
                Storage[earlIdx].item = 0;  //discarding current earliest message
```

```
                    //also setting child.parents to -1
                    if( Storage[earlIdx].child1 > 0) Storage[Storage[earlIdx].child1].parent = -1;
                    if( Storage[earlIdx].child2 > 0) Storage[Storage[earlIdx].child2].parent = -1;

            network_gpu.localQcounts[destination]--;                    //decreasing the local Q count
             network_gpu.earl[destination] = getEarliest(Storage,network_gpu,destination); //setting
the new current message

                    }

            //Case4: tm < tm1 and m1 not executed
            else if(tm1!=0 && tm<tm1 && Storage[earlIdx].sim ==0)
             {
                            //m replaces m1 ;
                            network_gpu.earl[destination] = current;
             }

            //Case5: tm < tm1 and m1' executed before
                        else if(tm1!=0 && tm < tm1 && Storage[earlIdx].sim ==1)
                            {

                                //Rollback LP
                     network_gpu.Status[destination] = network_gpu.BackupState[destination];
                    network_gpu.BackupState[destination] =-1; //empty the backup state

                            //m becomes LP's earliest
                            earl = network_gpu.earl[destination];
                             network_gpu.earl[destination] = current;

                            //m' goes back to GQ
                              DCopyPHItems( &bump2[tid], &Storage[earl]); //destination , source
                              }

                    // shud be the last step of while loop
                    tid++; //this is  just a local increment;

                   } //while

              }//sync_if

        }
```