TECHNIQUES TO IMPROVE THE PERFORMANCE OF LARGE-SCALE DISCRETE-EVENT SIMULATION

A Dissertation
Presented to
The Academic Faculty

by

Brian Paul Swenson

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
August 2015

TECHNIQUES TO IMPROVE THE PERFORMANCE OF LARGE-SCALE DISCRETE-EVENT SIMULATION

ACKNOWLEDGEMENTS

I would like to extend my gratitude to my advisor Dr. George Riley for his guidance and patience throughout my graduate study. Without his support I would have never been able to finish the program. I would also like to thank my dissertation committee members, Dr. Douglas M. Blough, Dr. Aaron D. Lanterman, Dr. Raheem A. Beyah, and Dr. Richard F. Fujimoto for their valuable advice and suggestions. I would also like to thank Dr. Tom Henderson for giving me many opportunities within the ns-3 community and for his support, guidance and patience.

Most importantly, I would like to thank my loving wife Jenni, and our two boys, Noah and Josh, for their unconditional love, encouragement, entertainment, understanding and support. Noah, I promise that I will have more free evenings to play Lego Marvel Universe now that this is done.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Discrete-event simulation is a commonly used technique to model changes within a complex physical systems as a series of events that occur at discrete points of time. As the complexity of the physical system being modeled increases, the simulator can reach a point where it is no longer feasible for it to run efficiently on one computing resource. A common solution is to break the physical system into multiple logical processes. When breaking a simulation over multiple computing nodes, care must be taken to ensure the results obtained are the same as would be obtained from a non-distributed simulation. This is done by ensuring that the events processed in each individual logical process are processed in chronological order. The task is complicated by the fact that the computing nodes will be exchanging timestamped messages and will often be operating at different points of simulation time. Therefore, highly efficient synchronization methods must be used. It is also important that the logical processes have a capable means to transport messages among themselves or the benefits of parallelization will be lost.

The objective of this dissertation is to design, develop, test, and evaluate techniques to improve the performance of large-scale discrete-event simulations. The techniques include improvements in messaging passing, state management, and time synchronization. Along with specific implementation improvements, we also examine techniques on how to effectively make use of resources such as shared memory and graphical processing units.

Techniques to improve the performance of Large-Scale Discrete-Event Simulation

Brian Paul Swenson

99 Pages

Directed by Professor George F. Riley

Discrete-event simulation is a commonly used technique to model changes within a complex physical systems as a series of events that occur at discrete points of time. As the complexity of the physical system being modeled increases, the simulator can reach a point where it is no longer feasible for it to run efficiently on one computing resource. A common solution is to break the physical system into multiple logical processes. When breaking a simulation over multiple computing nodes, care must be taken to ensure the results obtained are the same as would be obtained from a non-distributed simulation. This is done by ensuring that the events processed in each individual logical process are processed in chronological order. The task is complicated by the fact that the computing nodes will be exchanging timestamped messages and will often be operating at different points of simulation time. Therefore, highly efficient synchronization methods must be used. It is also important that the logical processes have a capable means to transport messages among themselves or the benefits of parallelization will be lost.

The objective of this dissertation is to design, develop, test, and evaluate techniques to improve the performance of large-scale discrete-event simulations. The techniques include improvements in messaging passing, state management, and time synchronization. Along with specific implementation improvements, we also examine techniques on how to effectively make use of resources such as shared memory and graphical processing units.

CHAPTER I

INTRODUCTION

Discrete-event simulation is a commonly used technique to model changes within a complex physical systems as a series of events that occur at discrete points of time. The events are processed in a time-stamp order and no changes to the state of the system can occur between consecutive events. The simulator is made up of a future event list, a simulation time, and the state of the physical system being modeled. When the simulator processes an event, it sets the the simulation time to the time within the timestamp of the earliest event in the future event list. Afterward the simulator updates the state of the physical system in response to the information contained within the event being processed. As an event is processed, the corresponding change in the physical state of the system may result in one or more new events to be generated in the future. The simulator will continue to run until it has completely exhausted the future-event list or until it processes an event with a stop request.

As the complexity of the physical system being modeled increases, the simulator can reach a point where it is no longer feasible for it to run efficiently on one computing resource. A common solution is to break the physical system into multiple logical processes. Each logical process is responsible for controlling a specific section of the overall system. Each logical process is made up of a local simulation time, a future-event list for the events pertaining to the section of the system it is responsible for, and the state of the system it is responsible for. The logical processes communicate state changes to one another through the use of time-stamped messages.

When breaking a simulation over multiple computing nodes, care must be taken to ensure the results obtained are the same as would be obtained from a non-distributed simulation. This is done by ensuring that the events processed in each individual logical process are processed in time-stamp order. The task is complicated by the fact that the computing nodes will be exchanging time-stamped messages and will often be operating at different points of simulation time. Therefore, highly efficient synchronization methods must be used. It is also important that the logical processes have a capable means to transport messages among themselves or the benefits of parallelization will be lost.

1.1 Contributions

The objective of this dissertation is to design, develop, test, and evaluate techniques to improve the performance of large-scale discrete-event simulations. The techniques include improvements in messaging passing, state management, and time synchronization. Along with specific implementation improvements, we also examine techniques on how to effectively make use of resources such as shared memory and graphical processing units (GPUs). The primary contributions of this work are:

- To improve the message passing performance for simulators running on multi-core systems with available shared memory, we developed a new zero-copy message passing approach specifically designed for the needs of distributed discrete-event simulators. This approach was compared to alternative available zero-copy approaches and to the traditional approach of serialization followed by a bulk memory copy. We show that our approach provides consistent performance, as would be expected of a zero-copy approach, and outperforms the traditional serialization and bulk copy approach in multiple simulation environments.

- We performed a study to determine if conservative time-synchronization techniques could be used to run large-scale distributed discrete-event simulations entirely on a GPU designed for three-dimensional gaming. Three unique and different approaches for event list management were examined and compared. Using the PHOLD benchmark we were able to process events at a rate of approximately 122 million per second on a simulation containing over 16 million logical processes. Furthermore, specific discrete-event simulation considerations are discussed in relation to GPU architecture. This work lays the ground work for future research to fully utilize the massive parallel processing abilities on GPUs for large-scale discrete-event simulation.

- We demonstrated how GPUs can be used by CPU-based discrete-event simulators for increased performance. We developed an add-on to the popular ns-3 discrete-event network simulator that allows users to utilize the BRITE topology generator to generate highly-customizable, large-scale topologies in ns-3. We then developed another GPU-based module to perform global routing between all of the nodes in the network. We show that our routing module is substantially faster than the global-routing module included in ns-3 and can even outperform the highly-efficient nix-vector routing implementation also included in ns-3. This work has lead to numerous discussions on how other ns-3 modules can utilize GPUs for processing highly-parallel tasks such as propagation loss and even topology generation.

- We examined the two conservative-time synchronization implementations provided with ns-3 for distributed discrete-event simulation. The implementations were tested in a number of different topology configurations in order to provide simulation users insight into which implementation to use for running their large-scale simulations. Furthermore, after examining both of the implementations, we were able to propose improvements to each to significantly increase their performance in certain simulation scenarios. We discuss the improvements and provide experimental results that demonstrate the improvement.

1.2 Dissertation organization

The remainder of this dissertation is organized as follows. Chapter 2 provides an overview of the background and related work in the area of distributed discrete-event simulation. Chapter 3 gives a description of our new approach to zero-copy message passing, designed specially for the demands of distributed discrete-event simulation, and compares its performance to other available methods. In chapter 4, we examine the feasibility of running large-scale simulations on GPUs as well as provide groundwork to support future research in this area. In chapter 5, we demonstrate how GPUs can be used in a supporting role to a CPU-based large-scale simulation. Here the GPU is used exclusively to quickly calculate routing information. Chapter 6 compares the conservative-time synchronization implementations in the network simulator ns-3 and provides enhancements to both that lead to significant performance increases in certain scenarios. Finally, chapter 7 concludes this dissertation.

CHAPTER II

ORIGIN AND HISTORY OF THE PROBLEM

2.1 Discrete-event simulation

Discrete-event simulation is a technique to model the operation of a physical system as a discrete sequence of events. Changes to the state of the system are made by events that are processed in time-stamp order. Between consecutive events within the simulation, no change in state occurs. The events are stored in time-stamp order in an events list. The three main pieces of the simulator are the state, the event list, and the clock, which represents the current simulation time. Prior to processing each event, the simulator updates the current simulation time to the time within the timestamp of the event. As each event is processed, new events can be scheduled in the future. Ensuring that the events are processed in chronological order is known as enforcing the causality constraint. A discrete-event simulator will continue to run until it is stopped or it runs out of events in the event list.

2.2 Parallel discrete-event simulation

As the size and the complexity of the models within the simulation grow, the simulation can reach a point where it is no longer feasible to run on one computing resource. In parallel discrete-event simulation (PDES), the standard solution is to parallelize the simulation by partitioning the model into multiple logical processes(LPs), each of which is run on a separate computing resource. Each LP has a local simulation time and an event list which will only contain the events that are to be processed on that particular LP. An LP will communicate an event impacting another LP through the transmission of a time-stamped message. It is mandatory that a distributed simulation produce the same results as a corresponding sequential simulation; therefore, causality must be enforced within the LPs. Specifically, an LP must be prevented from advancing too far ahead of other LPs, such that it receives an incoming event from another LP that contains a timestamp less than the current simulation time. To enforce the property of causality among LPs, distributed simulators make use of synchronization algorithms. These algorithms can be classified into two groups: optimistic and conservative.

2.2.1 Optimistic time-synchronization algorithms

Optimistic synchronization allows LPs to progress independently of one another for short periods of time, which can result in possible causality violations. If an LP processes an event out-of-order, i.e., it later receives an event with an earlier timestamp than its current local time from another LP, the LP must undo, or rollback, its local state to its state prior to processing the out of order event. The newly received event with the earlier timestamp can then be processed, and the LP will resume processing events in its event list, possibly reprocessing events that were rolled back if they are still valid given the update to the state of the LP by the new message.

The Time Warp mechanism [32] is the most commonly implemented optimistic time-synchronization algorithm. In the original proposal, the LP saves a copy of its entire state prior to the processing of each event. Then,if the LP needs to rollback, it discards its local state and reloads the saved state prior to the processing of the out-of-order event. In the original paper it was assumed that the state of the process was saved after the processing of every event. However, later works offered less demanding state-saving techniques [4] [74].

Regardless of the method used to save the state of the LP, over time this results in a large demand on memory resources. In order to minimize the amount of state storage for each LP, a global-virtual time (GVT) is maintained within the system. The GVT is a timestamp which contains the earliest timestamp of all of the unprocessed events in the system. Since no event with a timestamp earlier than the GVT will ever appear in the event list of a particular LP to cause a rollback, any state saved prior to the GVT can be removed. This procedure of reclaiming memory is referred to as fossil collection [24].

More recently, simulations using Time Warp have made use of reverse computation [60]. In reverse computation, the LP performs the inverse of events that have been processed out of order, effectively allowing the LP to run backwards in time to the point where it has processed an out-of-order event. However, for this process to succeed, reverse computation code must be written for each event type. Attempts have been made to create a compiler that automatically generates the reverse code for events. Perumalla [62] created source-to-source compiler for a subset of the C language that was demonstrated within the Georgia Tech Time Warp [16], an optimistic time synchronization simulator. Also work has begun on the development of a reversible C++ compiler [72]; however, frequently the generation of reverse events must be done manually due to the complexities of trying to generate reverse execution code from a language that was designed for forward only computing.

Languages that have been from designed from the start to ensure correct and efficient reversibility do exist. One such example is Janus [40], and another is R [21]. However these languages are restrictive the operations they perform, for example in Janus all variables must be global integers, and often their use is not practical for large complex systems.

2.2.2 Conservative time-synchronization algorithms

With conservative time-synchronization algorithms, each LP has to determine when an event is safe to process. An event is safe only when it can be guaranteed that the LP will not receive events from other LPs with timestamps less than the event being considered. In this way, the LP guarantees that all of its events are processed in order. These algorithms can be classified as being either *synchronous* or *asynchronous*. Synchronous algorithms require simultaneous participation from each of the LPs in the system. Asynchronous algorithms do not have the global communication requirement of synchronous algorithms instead synchronization is achieved by LP peer-to-peer communication. Conservative time synchronization algorithms must be sure to avoid deadlock, a situation where none of the LPs have a safe event to process and are waiting for updates from other LPs in order to continue.

One of the most popular conservative time synchronization algorithms is the Chandy-Misra-Bryant (CMB) algorithm. The CMB algorithm was developed independently by Chandy and Misra [12] and Bryant [9]. The CMB algorithm in an example of an asynchronous conservative time synchronization algorithm. The algorithm handles time synchronization and deadlock prevention by its use of *null messages*. The null message contains the the timestamp of the smallest unprocessed event on the sending LP plus the *lookahead* between the sending and receiving LP. The lookahead between two LPs is the minimum simulation time between the timestamp for an event on the sending LP and the timestamp on the subsequent event it generates on the receiving LP. The lookahead is closely related to the physical properties of the system being modeled. For example the lookahead between two communication nodes in a network simulator would be the speed of light delay between the two nodes. One important limitation of the CMB algorithm is that it cannot guarantee to prevent deadlocks if there are any cycles of logical processes with zero lookahead [24].

In the original algorithm proposal, after an LP processes an event, a null message is sent to each of its neighbors. Upon reception, the receiving LP knows that it can safely process the events in its event list up until the time in the null message without fear of receiving a message with an earlier timestamp from the sending LP. If the receving LP has multiple neighbors it uses the minimum null message timestamp value as its safe time. The obvious problem with this approach is that this greatly increases the amount of cross LP traffic. An alternative approach, proposed later by Mirsa [45], allows LPs to process events until no more are safe. At that point the LP requests null messages from its neighbors.

The other class of conservative time-synchronization algorithms are considered to be synchronous. To determine if an event is safe, the LP finds the lowest-bound timestamp (LBTS) on all possible events that it may receive in the future. This process is similar to how the GVT is calculated for optimistic time-synchronization algorithms. How this bound is determined is dependent on the particular algorithm used. Several algorithms for finding LBTS values [66] [42] [38] [7] exist. Another important consideration for this type of conservative simulation algorithms is how they deal with *transient messages*. Transient messages are messages that have been sent by one LP, but have not yet been processed by the receiving LP. These need to be considered because the event with the lowest time stamp may be in one of these messages.

2.2.3 Maximizing lookahead

For conservative time synchronization algorithms lookahead is an essential attribute for improved performance in parallel discrete-event simulation [22]. This value defines the amount of asynchrony among the distributed models in the system. Maximizing its value reduces the frequency of synchronization and increases the overall performance of the simulator by maximizing the amount of time each LP instance is processing internal events. A parallel simulation environment with minimal or no lookahead would essentially run as a non-parallel simulation with the additional overhead of LP communication. Due to its importance, there have been numerous studies that have examined techniques for maximizing lookahead values dynamically during simulation runtime, a technique referred to as dynamic lookahead extraction [59]. This technique has been shown to be vital in distributed wireless network simulation where the distances between communicating nodes is small, resulting in a minimal lookahead due solely to wireless propagation delay [39]. Researchers also have used information such as queuing delay to inflate the lookahead value higher than the static value assigned [76].

2.2.4 Non-CPU hardware support for parallel discrete-event simulation

There has also been work examining how hardware can be used to increase the performance of parallel discrete-event simulation. This hardware can be optimized for particular functions, allowing it to perform them faster than can a CPU. The *Rollback Chip* [25] was proposed by Fujimoto to help optimistic Time Warp simulations by reducing the overhead of state saving and rollback. It was found that the use of this chip greatly improved the simulator's performance.

On the conservative side of time synchronization, a hardware supported global synchronization unit was proposed by Lynch and Riley [41]. The chip allows individual LPs to calculate LBTS as needed, without the overhead cost of global synchronization. The chip contains three sets of registers files, one for minimum outstanding event (MOE), one for minimum outstanding message (MOM) and finally one for transient message count (TMC). Each LP in the simulation has a register in each file. Both the MOE register file and the MOM register file have N-1 comparators that can compute the minimum value in the register file in lg(N) stages. To find the smallest outstanding timestamp in the system, a `MinimumTimestamp` instruction is provided which returns the minimum of the MOM and MOE register files. It was found that the use of this unit reduced runtime by as much as 50% in tests performed with GTNetS [64].

A new area of research that has become popular as of lately is utilizing general purpose graphical processing units (GPGPU) within the realm of PDES. The first work on this was done by Perumalla [63] who used a GPGPU based discrete-event simulator to simulate a diffusion process. The work was done prior to the arrival of technologies such as CUDA and OpenCL and therefore used Brook and was executed on the DirectX 9 runtime. It was found that the GPGPU version of the simulator outperformed a similar simulator run on a CPU by almost two fold.

With the advent of GPGPU enabling technologies such as OpenCL and CUDA research into using GPGPUs for PDES has greatly increased. One example of a PDES simulator that makes extensive use of GPGPUs is Cunetsim [67]. Cunetsim [67] is a hybrid CPU-GPU network co-simulator where LPs process events using the GPU and time synchronization is done on the CPU. The authors use conservative time synchronization to ensure a casual ordering of events. The authors report substantial speedup [5] compared to non-GPGPU network simulators such as ns-3 and Singalo [17].

CHAPTER III

A NEW APPROACH TO ZERO-COPY MESSAGE PASSING

There are two main methods available to take advantage of the multiple cores on today's CPUs. The first is the thread model in which each logical process (LP) runs on its own thread within one operating system process. With multi-threaded applications, data can easily be passed between concurrently running threads using simple C type pointers. This allows the threads to communicate using messages of any size at the cost of passing a 32 or 64 bit pointer.

The alternative is the multi-process model where each logical process runs in its own individual operating system process. With multi-process applications, the process of passing messages becomes more complicated. The operating system runs each individual process in its own virtual address space, and any pointers created in a process will reference a virtual address, not a physical address. Since the mappings between virtual and physical addresses might be different for each process, pointers created by one process and passed to another might refer to a completely different physical location in memory for the receiver. Therefore, standard C type pointers can not be used as a means to pass messages between individual processes.

Both the multi-threaded and multi-process distributed simulation approaches are commonly used. In general, a distributed simulation using multiple threads will need all event handlers and the event scheduling engine to be aware of the need for multiple-access interlocking to prevent simultaneous updates and potential deadlocks. In contrast, when distributing the simulation in separate address spaces, only those portions of the simulation that send events and the portion that advances simulation time needs to be aware of the distributed execution. Further, if the original simulation package was not designed with distributed execution in mind, the multi-process approach is in general considerably easier to implement. ns-2 [48] is an example of a simulation environment that was not in fact designed originally to execute in a distributed fashion, but was later adapted to execute with multiple processes and disjointed address spaces.

A common API to use when working with multi-process simulations is the well–known Message Passing Interface (MPI) [20]. However, a significant overhead for MPI based applications is in message passing between the disjointed address spaces. Even in a tightly coupled, shared memory environment, the messages must be copied into a shared memory region. The copying of the entire message must be done because simple, C type pointers use virtual addresses which have no meaning once the pointer is passed to another process. Thus a common approach is to first serialize the message and its data and perform a memory copy of the serialized message to a shared memory location, where it then can be retrieved and deserialized by the receiving process. While this technique works and is in common use, a large processing cost can be incurred due to the amount of data being copied between processes. In the cases where these messages are large and must be passed frequently, this becomes a significant limiting factor in overall application performance.

An alternative approach is to use shared memory and smart pointers. With this method, data that needs to be shared with other processes can be created in a shared memory region that is accessible by every process. Then when a message needs to be passed, the owner can pass to the receiver a specialized smart pointer. These smart pointers allow the processes to pass only metadata for messages being exchanged, rather than the complete message. The receiving process can then access the original copy of the data stored in the shared memory using the smart pointer and normal dereferencing semantics. This greatly reduces the amount of data that needs to be passed between applications. This technique is referred to as zero-copy message passing.

This chapter presented a new implementation for zero-copy message passing in introduced [69]. This is not the first work to implement a zero-copy message passing approach. Boost Interprocess offers a smart pointer design that allows shared memory usage. Also provided by Boost is a general purpose, shared-memory allocator. However, it will be shown that the implementation provided by Boost does not perform well in memory-intensive applications, like distributed simulators. Furthermore, Boost's implementation does not scale well to a large number of LPs. In contrast, the new zero-copy approach described is designed to function on a large number of LPs in a memory demanding environment.Using this new approach, performance of PDES applications on multi-core architectures is greatly improved, allowing for longer and more detailed simulations.

3.1 Traditional message passing

Commonly, messages have been passed between individual processes by copying the entire content of the message to a sharedmemory region accessible by both the sender and the receiver. However the data to be transferred must first be formatted in a way that allows it to be meaningful to a receiver. This formatting, called serialization, or marshalling, generally copies each individual data item for all messages (and dereferencing pointers as needed) to a sequential array of bytes, which are then copied to the shared memory region.

In some cases, it is possible to marshall the data directly to the shared memory region, eliminating one of the memory copies needed. However this is rarely a trivial process because complex data objects generally make extensive use of pointers, references to other locations in memory. The data referenced from these pointers must also be copied into the buffer because, as mentioned previously, the pointer

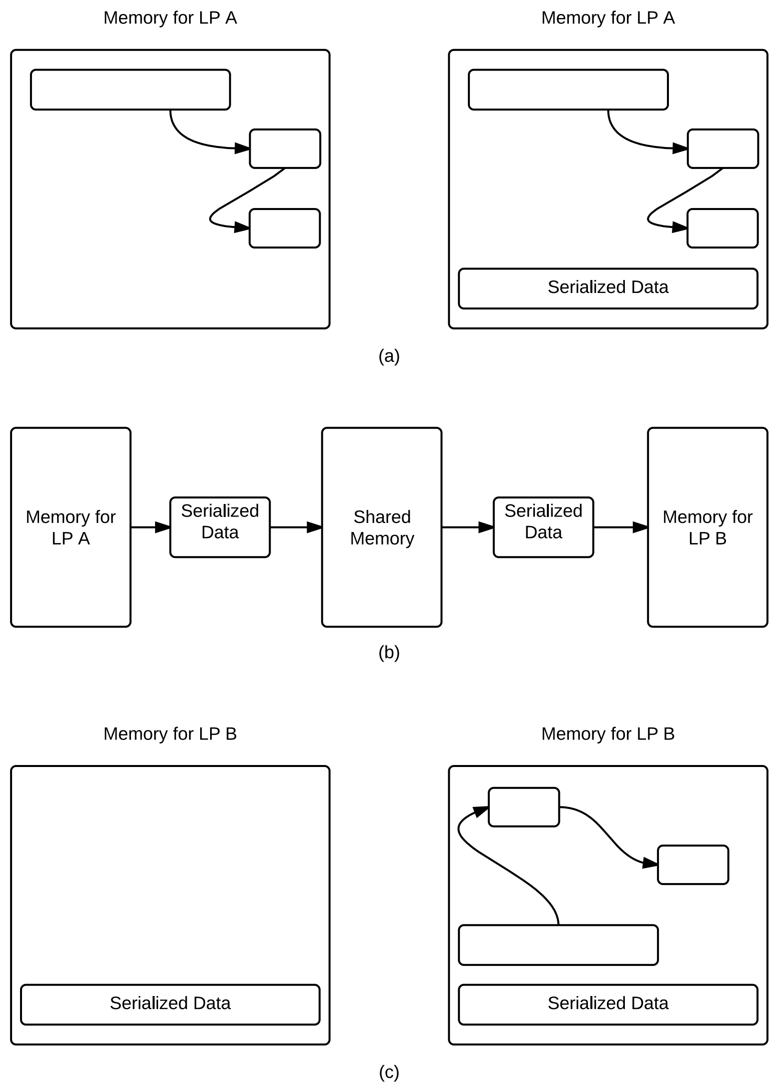


Figure 1: The transfer of a complex object between logical processes with disjoint address spaces. (a) The complex object is serialized. (b) The serialized data is first copied to shared memory and then copied into the memory space of LP B. (c) The data is finally deserialized and the object is ready to be used

will no longer be valid once the data is transferred to a different process. Thus the serialization process, in most cases, involves numerous and often recursive memory copies. Once the data to be sent has been serialized, it can then be transferred to the receiver. However, before the receiver can make use of what it has received, it must first deserialize, or unmarshall, the data. This is basically a reversal of the serialization process and therefore frequently requires multiple memory allocations to restructure complex objects back into their original form.

This process is depicted in Figure 1. While this procedure works as intended, the time to complete this marshalling and copying process is non-negligible and increases linearly with the size and complexity of the message.

3.2 Zero-copy message passing

Zero-copy message passing improves the performance of parallel simulations in a multi-process environment utilizing two main components: shared memory and smart pointers. First, rather than copying the data to shared memory, the data is created in shared memory, which can be accessed directly by the receiving process. As standard C type pointers cannot be utilized in this type of environment, zero-copy utilizes a smart pointer which is aware of the shared memory nature of the underlying data, and which has normal pointer semantics for dereferencing the pointer to access individual data items. Using the smart pointer eliminates the need for performing an expensive memory copy. Furthermore, the smart pointer can be equipped with reference counting semantics, which results in the underlying shared memory area being freed when all smart pointers pointing to the same area have gone out of scope. In this section we will describe our implementation of zero-copy and compare it to the implementation found in the Boost Interprocess C++ Library.

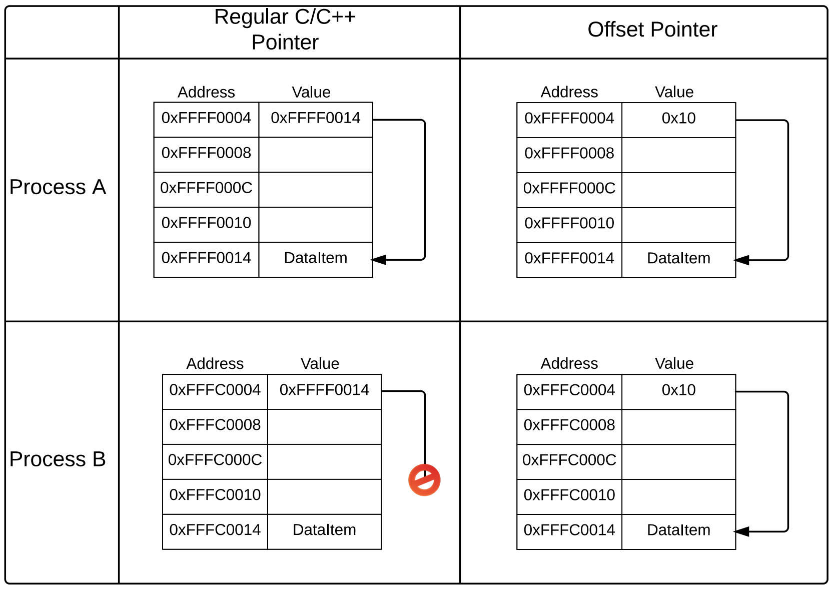


Figure 2: Difference between standard C/C++ pointer and offset pointer

3.3 Boost.Interprocess C++ library

Boost provides an extensive library for working with shared memory [26]. These classes can greatly simplify the task of working with shared memory. The following paragraphs discuss the Boost `offset pointer` and the Boost `managed memory segment` classes, which when used together can form a zero-copy implementation.

In Boost, the smart pointer that works with shared memory is the offset pointer. The offset pointer stores the distance from the offset pointers address to the object the pointer refers to. This allows objects created in shared memory to refer to each other regardless of which base address the shared memory segment is mapped into the processes address space. An example of this is shown in figure 2. Here you can see two processes have mapped a shared memory segment into their local address space. The regular pointer, create by process A, is not valid for process B since it mapped the shared memory segment to a different base address. Therefore the data to which the pointer should point is not at the address contained within the pointer. With the offset pointer, both processes can correctly address the referenced data regardless of

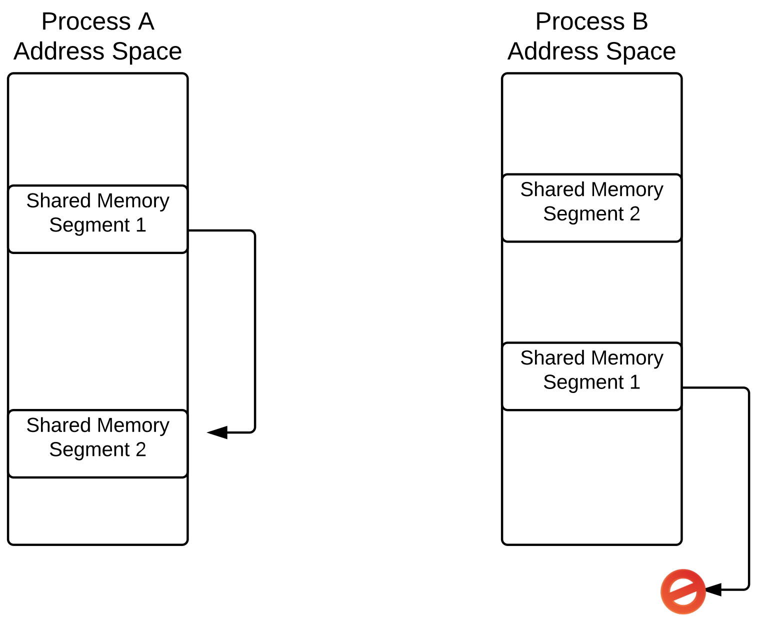


Figure 3: Offset pointer with multiple shared memory segments

where a process maps the shared memory segment. The only requirement is that the entire shared memory segment is mapped into a contiguous block of addresses.

To the programmer, the pointer functions equivalently to a normal pointer and can be coupled with a reference counting pointer to supply automatic garbage collection. The problem with this design is that for it to work correctly, all of the objects have to be stored in the same shared memory segment. There is no guarantee that the difference in base addresses of two shared memory segments mapped into the address space of a process will be the same between processes. This issue is depicted in figure 3. Therefore, in most cases, only one shared memory segment can be used for all LPs.

A pool of shared memory is obtained in Boost using managed-memory segments. Once a pool is available, objects can be created using the segments allocator. The segments provide an `allocate` method that takes as a parameter a byte size and returns a void pointer to a chunk of memory that size if it is available in the segment. The segments also provide a templated `construct` method which will create an instance of the object specified in shared memory and return a pointer to it. Managed memory segments in Boost also allow the programmer to create named shared memory objects. A string name can be given to any object created in shared memory.

This name can then be used by any process that connects to the memory segment to find the object in shared memory. While the Boost shared memory segment offers many useful features, it is unable to perform satisfactorily in a memory intensive environment, as we will show. In addition, as mentioned above, due to how Boosts smart pointer interacts with memory segments, only one such segment can typically be used in an application.

This has two major consequences. First, access to critical sections of code in the segments allocator can become a major bottleneck as the number of LPs increase. Second, if the memory in the one shared memory segment is exhausted, the application has no choice but to terminate since shared memory segments are not dynamically expandable.

3.4 Zero-copy message passing optimized for simulation

We present a new implementation for zero-copy message passing that is optimized for the demands of PDES and addresses the shortcomings of the Boost Interprocess implementation. In our implementation, each LP is given its own shared memory segment which we refer to as a shared heap. We also provide a custom offset smart pointer to interact with these shared heaps. Our smart pointer also has reference counting semantics to ensure memory is deallocated once it is no longer in use. In the following sections we describe the pieces of our implementation and demonstrate its usage.

3.4.1 Custom smart pointer

Similar to the Boost implementation, to reference items in shared memory, our zero-copy message passing technique utilizes smart pointers. These smart pointers are

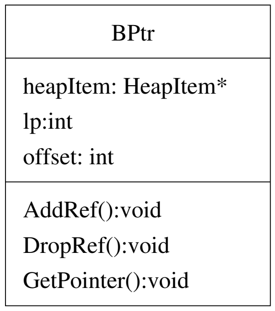


Figure 4: UML Diagram of BPtr, the offset pointer used in our implementation

necessary because, as discussed previously, standard C type pointers contain a virtual address which may be meaningless when passed between process boundaries. The smart pointers used in the zero-copy approach get around this limitation by passing the metadata necessary for the receiving process to obtain the data that it was intended to receive. The smart pointer created for zero-copy message passing technique is called `BPtr` and its UML diagram is shown in figure 4.

The smart pointer is made up of three data items. The integer LP stores the heap number of the shared heap which contains the actual data being referenced. The offset stores the integer index of the heap item in the data owner's shared heap. Details of heap items will be presented in the next section. The smart pointer also contains a heap item pointer to the heap item referenced by the heap number and offset stored in the smart pointer. This heap item pointer is updated automatically by the smart pointer whenever the smart pointer is copied so it always points to the correct heap item, even when the pointer is passed across process boundaries.

The main difference between our pointer and the Boost offset pointer is that our pointer stores the heap number along with the offset. This offers two major advantages. First it allows our pointers to point to objects in separate shared memory segments. This eliminates many of the concurrency issues experienced with the Boost managed memory segment allocator. The second advantage is that new heaps can be created during runtime if shared memory resources are exhausted. A request is made

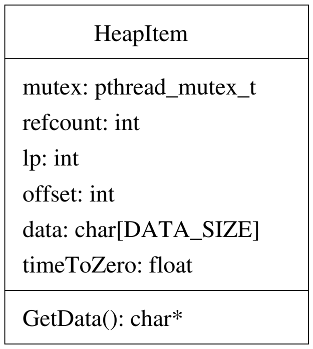


Figure 5: UML Diagram of a heap item which stores data to be shared between LPs

to the kernel for more shared memory and once it is received the heap structure can be setup and be given a unique heap number. At that point the only thing left to be done is to notify all of the processes to connect to the new heap and add it into their heap cluster.

When a copy of the smart pointer is made, only the heap number and offset data items are copied. Then the `GetPointer` function is called, which gets the heap item pointer. It does this by indexing the `HeapCluster` array using the LP to acquire a reference to the correct heap and then using the offset to address it to the correct heap item. Another feature of the smart pointer class is that it automatically handles the reference counting for the heap item. The assignment operator and copy constructor have been overloaded to atomically increase the reference count of the appropriate heap item when a new reference to the heap item is made. Similarly, the virtual destructor for any `BPtr` object will decrement the reference count appropriately.

3.4.2 Shared heaps

As previously mentioned, in our zero-copy implementation each LP is given its own shared heap for memory allocation. Each shared heap is made up of a group of heap items. It is in these heap items that the data to be shared is stored. The heap items are indexed according to the offset from the start of the heap. It is in these heap items that the data to be shared is stored. The heap items are indexed according to the offset from the start of the heap. A UML diagram of a heap item is shown in figure 5. The `Create` method returns a special templated pointer object called a `BPtr` which refers to an individual heap item.

The heap item is made up of multiple data items. It stores the number for the logical process it belongs to as well as its index number in the shared heap. The data field is where the actual data for the heap item is stored. The reference count specifies the number of smart pointers that are currently referencing the data in the heap item. This is used to determine when the heap item is no longer in use and can be recycled. This number is atomically incremented and atomically decremented whenever the heap item is copied or when it goes out of scope. Finally each heap item contains a `timeToZero` timestamp.

The `timeToZero` timestamp is used for simulators which use an optimistic time synchronization algorithm. As discussed previously, there are situations when optimistic simulators need to rollback due to the generation of a causality error. During this rollback, it may be necessary for the simulator to reacquire dynamically created objects that had previously been freed. This is true regardless of if the simulator is reverting to a previous saved state or performing reverse computation. In order to assist in this process, a heap item will not be immediately available for reuse once its reference count goes to zero. Instead the heap item will store the current simulation time for its LP in the time to zero field and will go into a dormant state, preserving itself and the data it contains. The heap item will stay in this state until the GVT of the system is greater than its stored `timeToZero` timestamp. Only after this happens will the heap item be made available for reuse. Because of this, it is necessary for the simulation to provide the shared heaps updated values for the GVT whenever they are calculated. For simulators that use a conservative time synchronization algorithm, the `timeToZero` timestamp is not used. It could either be removed or the GVT for all the heaps in the system could be set to infinity at startup. Either way, heap items that have their reference count go to zero will immediately be available for reuse. Since the concept of rollback does not exist in conservative time synchronization, there is no need to postpone garbage collection.

In our ZeroCopy implementation, the heap items are preallocated. The sizes for the heap items and the number to create can be configured by the user prior to runtime. For example, the user could setup heap items for small blocks (50 bytes), medium blocks (500 bytes), and large blocks (5,000 bytes). When an object is created, the heap will return the smallest size that the instance of the object will fit into. By preallocating the blocks, we can provide extremely efficient allocation and deallocation routines since the blocks are of a fixed size and are stored sequentially in memory. When a shared memory pool is requested from the linux kernel, the exact size must be specified and once it is allocated, it cannot be modified. Therefore we feel it makes sense to split the pool into blocks immediately, especially since the user of the simulator should have a general idea of the size of objects he/she needs to create prior to runtime.

Every logical process in the simulation has its own shared heap, which is initialized at initialization time. The heaps are created in shared memory and the permissions are set so every LP can access every other LP's shared heap as well as its own.

Once all of the logical processes finish creating their own shared heaps, all logical processes make attachments to all other shared heaps. Each LP stores pointers to each of these heaps in a global variable called `HeapCluster`. A diagram of this is shown in figure 6. Here two LPs have finished setting up their individual shared heaps and have stored pointers to both shared heaps in their `HeapCluster`, which is a global array of pointers to the heap object for all logical processes. In our implementation, each process maintains the array of virtual memory pointers to all shared memory regions in the order of the logical process number of the creating LP. This is shown in figure 7.

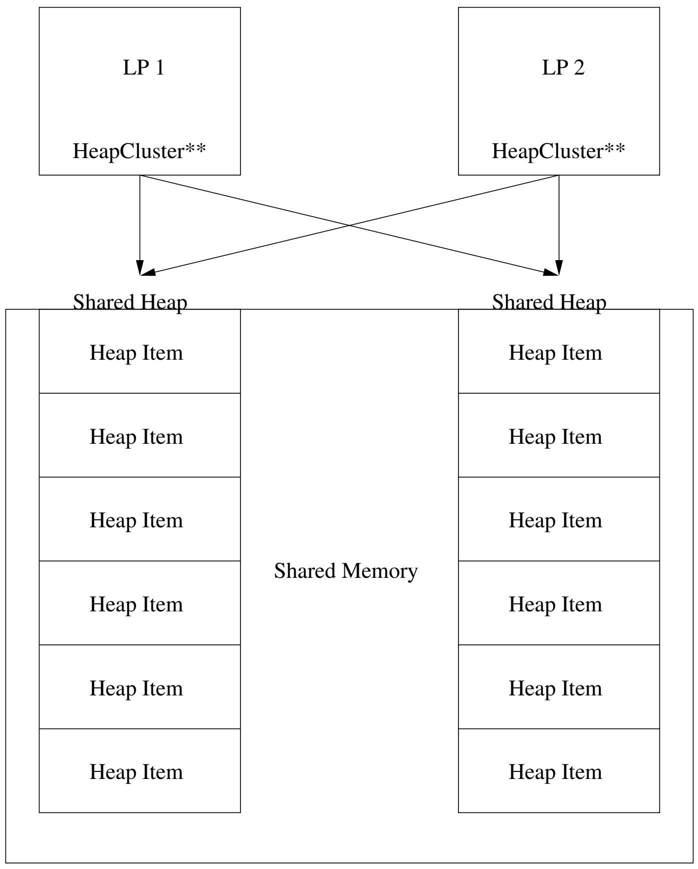


Figure 6: Layout of Shared Heaps

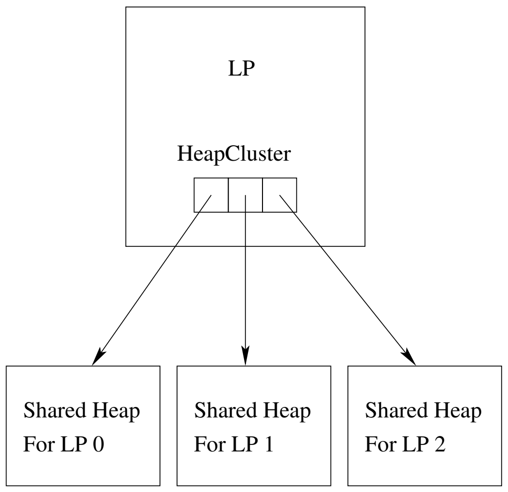


Figure 7: Ordering of pointers in HeapCluster

While both LPs have pointers to the same shared heap, the actual virtual address of the shared heap will typically be different. When a process attaches to a shared memory segment, the shared memory is mapped into the process's virtual address space and there is no guarantee that it will map to the same location for each process that attaches to it. This is why we cannot simply pass normal C/C++ pointers to data items in the shared memory across logical process address spaces.

3.4.3 Using zero-copy message passing in simulation

In our zero-copy message passing design, when a new message is created (presumably to later be passed to another process), a special constructor method, called `Create`, is used rather than the normal C++ `new` operator. The `Create` method is functionally equivalent to `new`, excepting that the memory is allocated from a pre-existing shared memory we call the shared heap, rather than from the normal memory heap used by `new`.

Once created, the smart pointer behaves syntactically the same as any other pointer. When the smart pointer is dereferenced, it uses the stored LP to connect to the shared heap of the process that created the data and then uses the stored offset to obtain the heap item. The appropriate operators have been overloaded in

Listing 3.1: Smart Pointer example

```
//Create a new packet object using the smart pointer
BPtr<Packet> packet = Create<Packet>();

//The pointer now functions syntactically the same as a regular pointer
packet->SetDestIP("192.168.1.5");
packet->SetSize("200kB");
cout << "My destination IP is: " << packet->GetDestIP() << endl;
cout << "My size is: " << packet->GetSize() << endl;

//The smart pointer class automatically handles reference counting.
//Here the reference count is increased to 2.
BPtr<Packet> anotherRef = packet;

//And now decreased back to 1
anotherRef = NULL;

//Below the reference count is decremented, which results in a zero refcount.
//The current simulator time is saved into the HeapItem's time To Zero field.
//If this isn't done explicitly, it will occur automatically when the smart pointer
    goes out of scope.
packet = NULL;
```

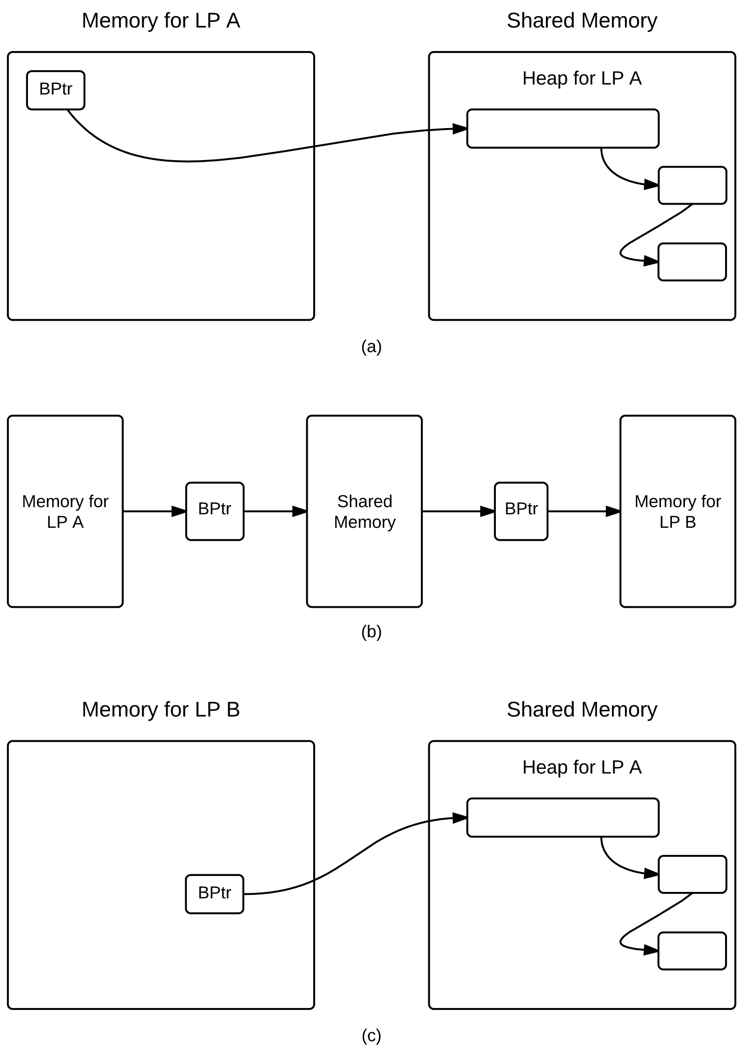


Figure 8: The transfer of a complex object between logical processes using zero-copy. (a) The object is created in LP A's shared heap and is accessed by LP A via the smart pointer. (b) The smart pointer is copied to shared memory and then copied into the memory space of LP B. (c) Using the smart pointer, LP B can directly access the object in LP A's shared heap. The object itself is never moved or copied.

the smart pointer definition to allow access to all class member functions and data in the referenced class.

Listing 3.1 shows an example of creating a Packet object for a network-simulator using the smart pointer class. Figure 8 shows a message being passed using a smart pointer. Note that the actual object referenced by the pointer is neither moved nor copied.

3.5 Evaluation

The first evaluation of our new zero-copy implementation used a custom simulator which we named SimpleSim. SimpleSim is a simple distributed discrete-event simulator that functions similarly to the PHOLD PDES benchmark [23]. In our implementation, the simulator enforces causality using a conservative lower bound time stamp (LBTS) algorithm that exchanges timestamp and message count information between LPs in a common shared memory region. Each LP starts off with one event inserted into its event queue. The timestamp for the first event is chosen randomly within the first five simulation seconds.

When handling an event, SimpleSim always creates one new message and sends it to an LP chosen randomly from a uniform distribution. It then examines the size of its event queue. If the size is less than a predefined constant value, it creates another message which is also sent to a randomly chosen LP. It is possible that the LP can choose itself. Since the size of the event lists for the LPs will grow at approximately the same rate due to the random nature in which the recipients are selected, this prevents unbounded growth in the total number of events for any individual LP.

Included in the message sent to the recipient is a timestamp, a unique ID, and a pointer to an arbitrarily sized chunk of data, which represents the data to be passed to the receiving LP. The timestamps for the new events are chosen from a uniform distribution of 1 to 5 seconds in the future. Added to the timestamp is a predefined constant lookahead value. The unique ID was used mainly for testing purposes.

When SimpleSim receives a new message, it removes the message from the queue and schedules a new event in its event list for the timestamp specified in the message. The individual simulators continue to process and create events until a predefined stop time is reached.

Three versions of SimpleSim were created. The first version is an MPI implementation that copies the entire contents of the message to the receiving process. The other two versions both use a zero-copy technique. The second version uses the Boost Interprocess library and the third version uses our zero-copy implementation.

We created this simulation because it allowed us to easily vary the size of data being passed between processes. In most cases, the message being passed to the receiving process is a complex object with multiple nested pointers. However, in this simulation, the message that is being transferred is uninitialized memory. Thus, there is no serialization step prior to the memory being copied. This also means that there is no de-serialization step. Thus, any speedup observed is due only to the lack of a bulk-memory copy.

For the first experiment with SimpleSim, the number of LPs was held constant at eight and the size of the message was scaled from 500 to 50,000 bytes. The simulator was set to run for 25,000 simulator seconds. The maximum event-list size where LPs stopped sending a second message was set to 5,000 and the lookahead value was set to five seconds. All simulation configurations were run ten times and the average result was recorded. Figure 9 shows the run time of the three approaches for a variety of message sizes. As expected, for the two zero-copy approaches, the execution time is nearly constant regardless of data size. Again this is because the data is not being copied along with the message but instead is being referenced directly from the shared heap where it was created. By comparison, the execution time of the MPI full copy approach is growing approximately linearly with the size of the data being sent. The

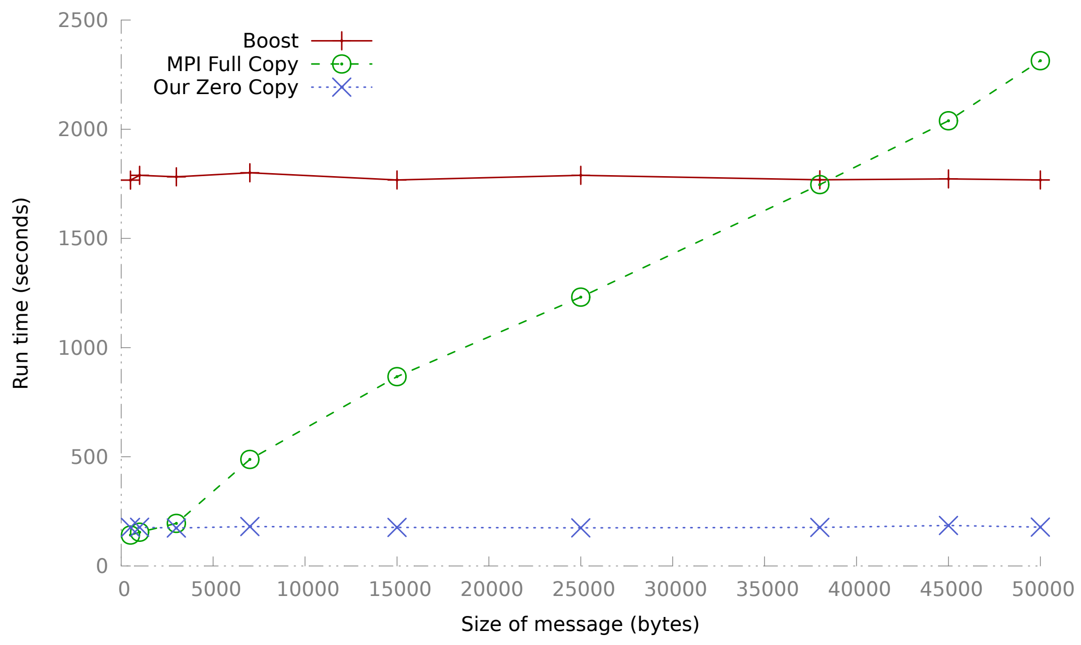


Figure 9: The runtime of SimpleSim with 8 logical processes, varying message size using MPI, Boost Interprocess and our custom zero-copy approach

results also show that our approach outperforms MPI full copy when the message is larger than approximately 3,000 bytes and outperforms Boost's implementation by almost ten times.

For the second experiment with SimpleSim, we wanted to examine how each simulator instance scaled as the number of LPs participating in the experiment was increased. The size of the data being transferred was fixed to 7,000 bytes. Again the simulator was set to run for 25,000 seconds of simulator time and the maximum event list and lookahead values were set at 5,000 and 5 respectively. Figure 10 shows the results of our experiment. The data clearly shows that the Boost implementation scaled much more poorly than either of the other two. This is presumably because, as discussed previously, the Boost implementation is limited to only one shared memory segment. However even when the Boost version is performing at its peak, our implementation outperforms it by almost six times.

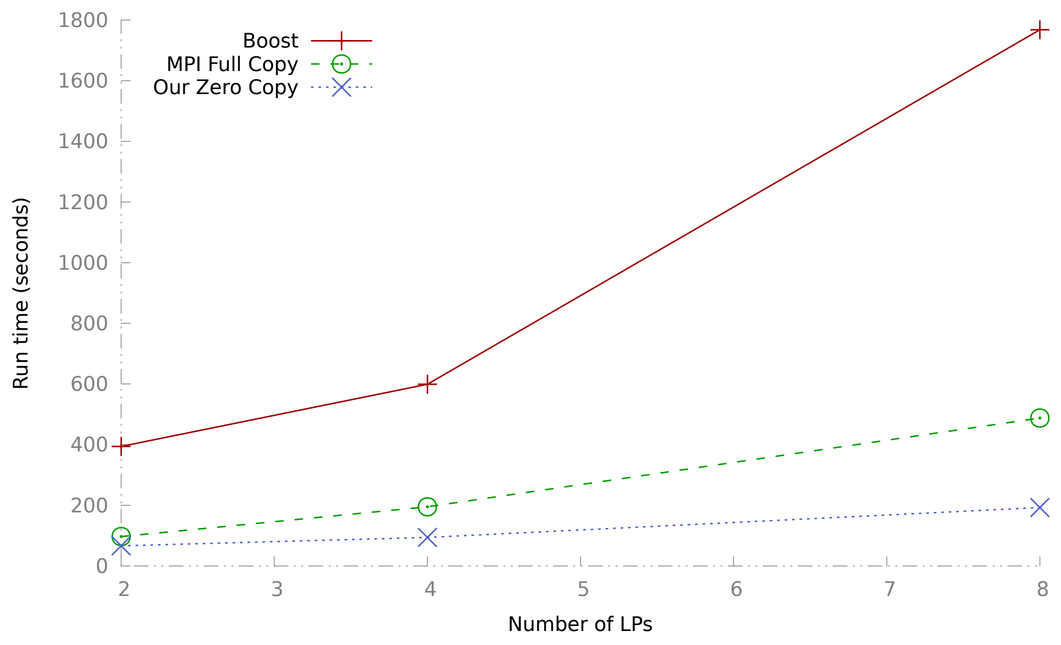


Figure 10: The runtime of SimpleSim with a message size of 7000 bytes, varying the number of logical processes using MPI, Boost Interprocess and our custom zero-copy approach. Data collected for 2, 4 and 8 LPs

3.5.0.1 GTNetS

The Georgia Tech Network Simulator (GTNetS) [64] is a full-featured network simulator for modeling large-scale topologies. GTNetS offers packet level tracing and models packets with protocol data units (PDUs) that are added and removed as the packet moves up and down the protocol stack. Similar to SimpleSim, GTNetS also uses conservative time synchronization. We chose to test our zero-copy approach on GTNetS because the messages passed between processes in GTNetS are complex packet objects that contain multiple PDU objects that must be serialized prior to transfer.

For our GTNetS experiments we created a star topology for each LP. The hubs for each of the stars was then connected to form a clique. Each star was given N-1 nodes where N was the number of LPs participating in the simulation. Each node of a star was configured to send UDP traffic to a node in a different LP. Therefore each

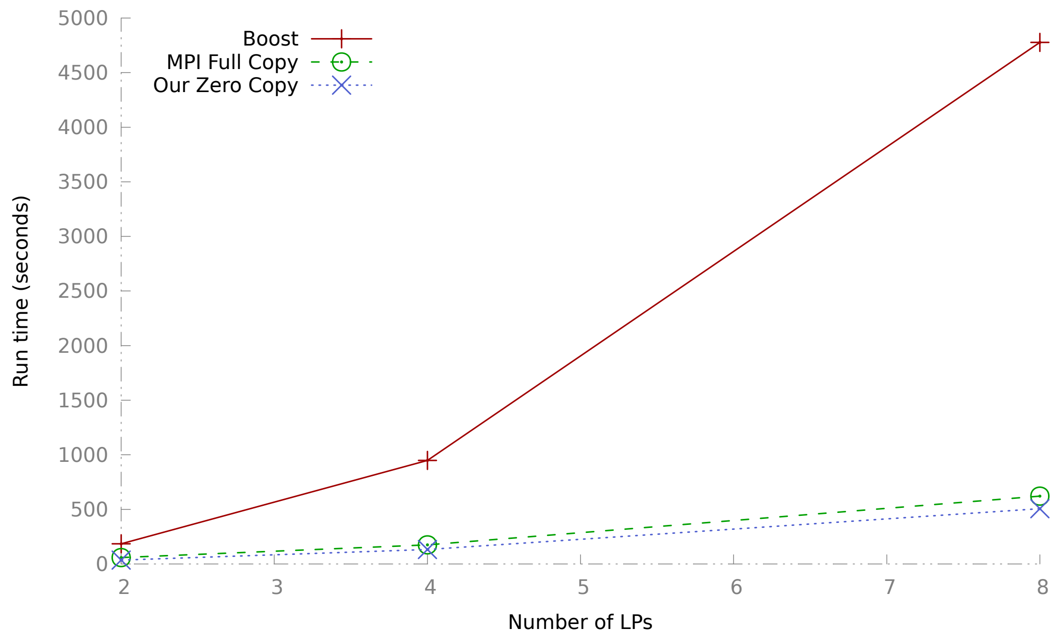


Figure 11: The runtime of GTNetS varying the number of logical processes using MPI, Boost Interprocess and our custom zero-copy approach. Data collected for 2, 4 and 8 LPs.

LP was sending UDP traffic to every other LP using one of its nodes. Each node was also given a UDP sink to receive data being sent to it. Each UDP packet sent was configured to hold 1,024 bytes of data and each sender was configured with an On/Off Application to use approximately 20% of the available bandwidth. Senders were configured to start at a random time with the first half second of simulation and the simulation was configured to run for 5,000 simulation seconds. All simulation configurations were run ten times and the average value was recorded.

Figure 11 shows the results of this experiment. Again, the Boost version of the simulator scaled worse than the other two versions. Our version of zero-copy outperforms the MPI full copy version even though the message size is less than 1,100 bytes. This is due to the fact that the MPI version has to serialize/deserialize the complex packet hierarchy before and after transferring it. This demonstrates that the effectiveness of our approach improves as the complexity of the objects being transferred increases.

3.6 Discussion

This chapter presented a new approach to zero-copy message passing on a many-core architecture was presented. The effectiveness of this approach was demonstrated using distributed discrete-event simulation. While other approaches to zero-copy message passing exist, this new approach has been shown to achieve better performance, especially in the case where many processes are attempting to allocate memory. This approach requires little additional effort on the part of the software developer when creating a new message passing application. This only significant different that affects the software programmer is the use of a special custom smart pointer object instead of normal pointers. This smart pointer has been enhanced to include reference counting semantics which will automatically free shared heap memory once all references to a given object have gone out of scope. Finally, in order to be compatable with optimistic time synchronization algorithms, this zero-copy approach allows for retaining of freed memory contents until certain GVT values have been reached, allowing for freed memory to automatically be restored in the event of a rollback.

CHAPTER IV

LARGE-SCALE DISCRETE-EVENT SIMULATION UTILIZING CUDA WITH CONSERVATIVE TIME SYNCHRONIZATION TECHNIQUES

Parallel discrete-event simulation is a technique frequently used to speed up simulations by partitioning a simulation into logical processing units which are executed on separate processors. A large amount of research has gone into techniques and optimizations on how do this most efficiently on CPUs. However, significantly less research has gone into how this could be accomplished using the massive parallel processing abilities of GPUs. This chapter analyzed and compared three different techniques for running large-scale discrete-event simulators on GPUs using conservative time synchronization techniques and the NVIDIA CUDA [54] API.

4.1 CUDA Overview

This section provides an overview of CUDA and the underlying GPU architecture. The architecture discussion will cover the NVIDIA Kepler [51] GPU architecture since it was used exclusively for the testing of the simulators presented in this work, although some details may be true of other architectures as well. Knowledge of these topics is necessary to understand the design choices of the simulator.

4.1.1 CUDA Execution Model

CUDA is a general purpose parallel computing platform and programming model that can be used to exploit the massive parallel execution abilities of NVIDIA GPUs. When using CUDA, the GPU can be viewed as a coprocessor with the ability to launch a large number of threads in parallel. Code written to execute on the GPU,

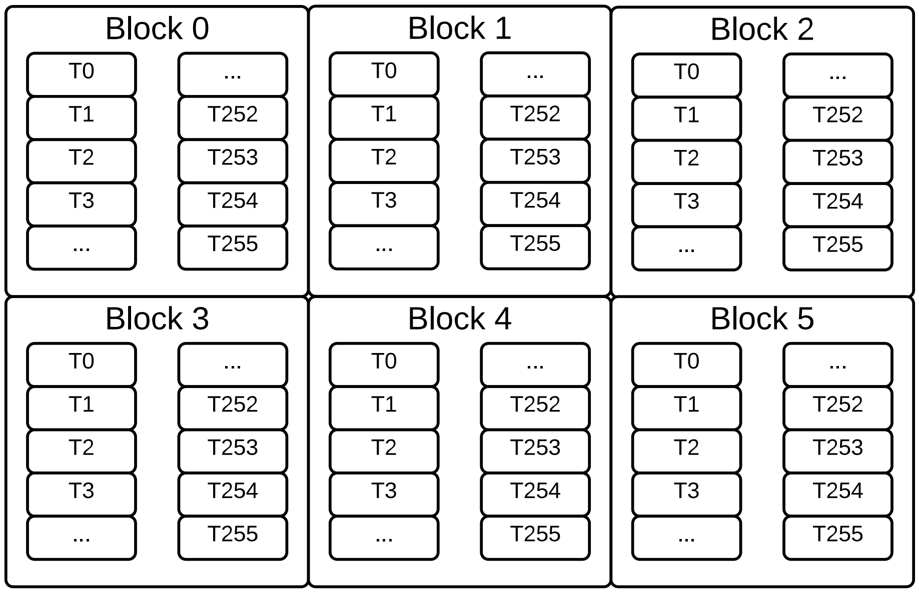


Figure 12: An example execution grid consisting of 6 blocks, each with 256 threads.

known as a kernel, is compiled to a GPU specific binary. During program execution, the compiled kernel and the data to be processed are uploaded to the GPU. After invoking a kernel, the CPU can either continue processing other tasks, including possibly launching other kernels, or wait until all kernel executions are complete.

Listing 4.1: Example grid creation

```
kernelFunction<<<6,256>>>(kernel parameters);
```

Kernel launches from the CPU are accompanied by a programmer provided `grid`. A `grid` is comprised of a `blockCount` and a `blockSize` which refers to the number of threads in each block. The block size has a maximum value of 1024 in current hardware and should always be evenly divisible by 32 based on how the threads are executed on the hardware. The value is typically used as a tuning parameter since some workloads can be executed more efficiently with more blocks of a smaller block size. In C/C++, a grid is specified within triple angle brackets following the function name as shown in listing 4.1. In this example, a grid with 6 thread blocks, each containing 256 threads, is created as shown in figure 12.

4.1.1.1 GPU Thread Execution

GPUs are comprised of an array of multi-threaded streaming multiprocessors (SM). In the Kepler architecture, each SM is made up of 192 single-precision cores and 64 double-precision cores. Each SM also contains 32 special function units and 32 load/store units. The special function units provide support for functions like sine, cosine and square root. During execution, each block of the grid is assigned to an SM and will stay with that SM until it has completed execution. Each SM is capable of holding up to 16 blocks or 2048 threads, based on whichever limit is reached first.

Thread execution within the SM occurs in groups of 32 threads, which is known as a `warp`. Based on this grouping scheme, all block sizes should be evenly divisible by 32. Otherwise, some warps will have unused threads. Threads are grouped into a warp sequentially by their thread identifier. For example, the 255 threads in block 0 from figure 12 will be split into eight warps: T0-T31, T32-T63, T64-T95, T96-T127, T128-T159, T160-T191, T192-T223, and T224-T255. The threads within the warp are executed in a pattern similar to the single instruction multiple data (SIMD) pattern in Flynn's taxonomy [19]. However, unlike standard SIMD where every thread executes every instruction, programmers using CUDA have the ability to insert divergent code paths into their CUDA kernels. NVIDIA refers to this execution pattern as single instruction multiple thread (SIMT). It is handled at the hardware level by enabling and disabling certain threads within the warp during execution. While this design provides the programmer with greater flexibility in kernel design, care must be taken to avoid potentially large performance consequences.

Listing 4.2: Example of thread divergence

```
\\obtain threadId in block
int myId = threadIdx.x;


if(myId % 2 == 0)
```

```
        doMethodA();
else
        doMethodB();
```

Listing 4.2 shows a classical example of thread divergence. In this listing, the threads with an even identifier execute `doMethodA` while the threads with an odd identifier execute `doMethodB`. Since the warp executes in a SIMD fashion and consists of threads with sequential thread identifiers, the warp must execute both `doMethodA` and `doMethodB`. During the execution of `doMethodA`, the threads with an odd thread identifier are disabled as shown in figure 13a. During the execution of `doMethodB`, the threads with an even thread identifier are disabled as shown in figure 13b. Assuming both methods are of equal size, the throughput of the kernel is effectively halved. In cases where there is a deep nesting of conditional statements, the situation becomes even worse. The warp has to sequentially step through every execution path taken by any of its threads, disabling and enabling threads as needed. Given its potential for severely limiting kernel performance, there has been a good deal of research on techniques to limit thread divergence [27] [68] [75] [15] [8].

Listing 4.3: Kernel with no divergence penalty

```
\\obtain warpId with integer division
int myWarpId = threadIdx.x / 32;

if(myWarpId % 2 == 0)
        doMethodA();
else
        doMethodB();
```

Frequently, the cost of conditional branches within a kernel can be mitigated by thoughtful algorithm selection. In listing 4.2, the intent of the kernel was to have half of the threads within a block perform one function and have half of the threads

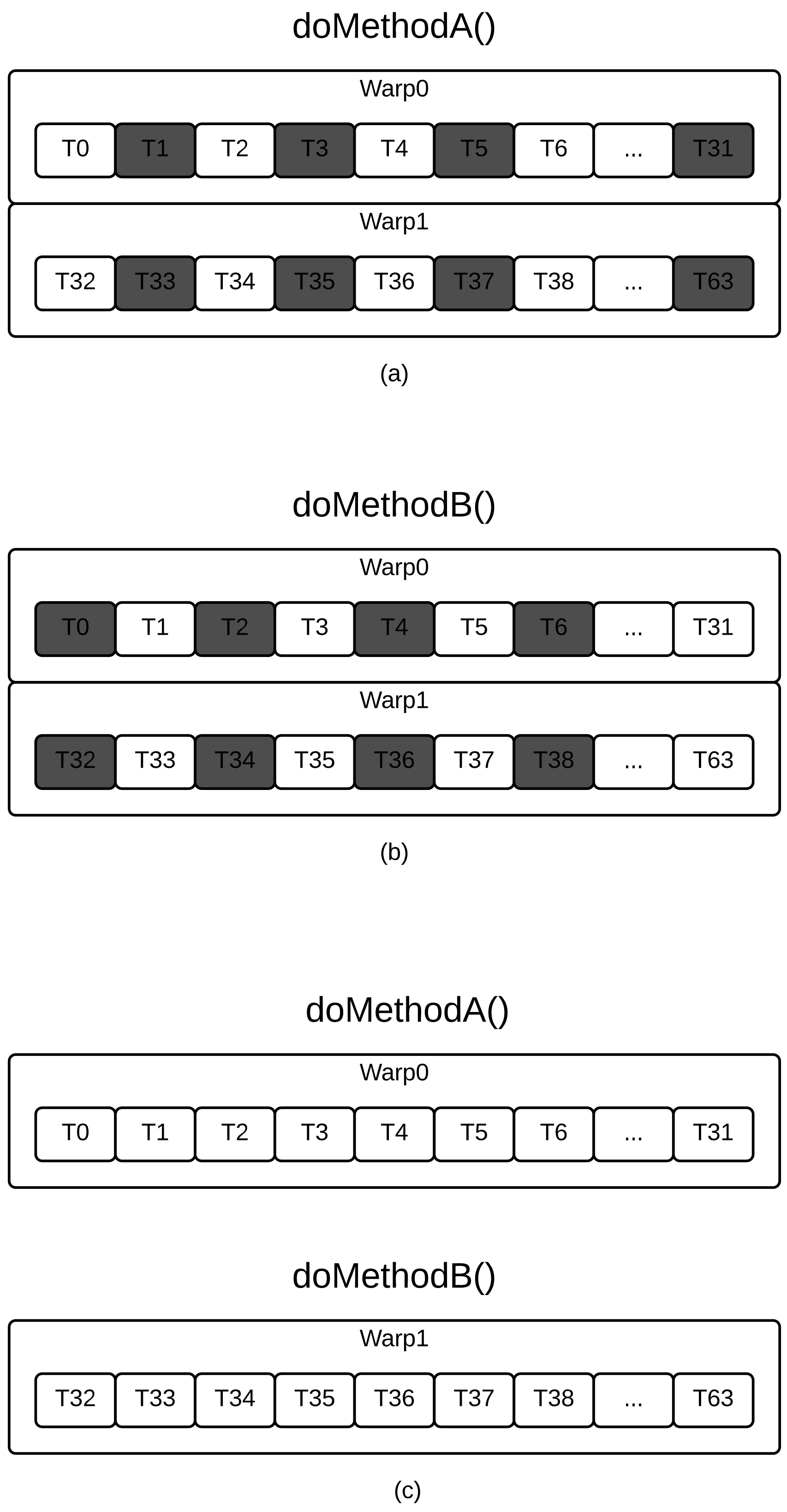

doMethodA()
Warp0
T0 T1 T2 T3 T4 T5 T6 ... T31
Warp1
T32 T33 T34 T35 T36 T37 T38 ... T63
(a)
doMethodB()
Warp0
T0 T1 T2 T3 T4 T5 T6 ... T31
Warp1
T32 T33 T34 T35 T36 T37 T38 ... T63
(b)
doMethodA()
Warp0
T0 T1 T2 T3 T4 T5 T6 ... T31
doMethodB()
Warp1
T32 T33 T34 T35 T36 T37 T38 ... T63
(c)

Figure 13: The cost of thread divergence within a warp. (a), (b) Show the result of a naive conditional statement within a warp. The darker threads are disabled. (c) Results of a modified conditional statement where there is no thread divergence within a warp

perform another. If the author was using a block size that would produce an even number of warps, the kernel shown in listing 4.3 would be a better option. In this listing, the condition is based on the identifier of the warp which can be obtained by the provided thread identifier. By making this change, all of the threads within a warp will execute only one of the functions and none of the warps will be forced to step through both, as shown in figure 13c.

The example kernels above are intentionally trivial to demonstrate the basic issue of branch divergence. The CUDA compiler, nvcc, will attempt to minimize the cost of divergent branches in areas where it is able. One area where this type of optimization would play a role in the context of discrete-event simulation would be in event handling code. A kernel that is tasked with updating the state of a large number of LPs based on the next event type of a particular LP would benefit by grouping each event type in a separate group of warps. This implementation would prevent warps from sequentially executing each and every event handler.

4.1.1.2 GPU Memory

To do anything useful, most kernels are going to have to access memory. This fact presents another area where care must be taken to avoid potentially large performance degradation. There are six different types of memory within the CUDA API: global memory, texture memory, constant memory, shared memory, local memory, and registers. The memory types vary in scope, size, and optimal usage patterns. Not all of the types will be covered in this section; however, a full description of each can be found here [56].

When a thread block has been assigned to an SM, the threads within that block are considered to be "in-flight." An "in-flight" thread has its own private copy of the non-shared variables used within the kernel. These private copies are stored in the block registers. If the registers overflow, the excess is moved to local memory. In the

Kepler architecture, each SM is supplied with 65,536 32-bit registers, and each thread is capable of addressing up to 255.

Constant memory provides fast read-only memory that can be cached at each SM. It is best used when every thread within a warp is reading from the same address because the results of each read are broadcast to each thread within the warp [13]. If every thread in a warp were to read a separate address from constant memory, the accesses would be serialized, i.e. not done in parallel. If this type of memory access pattern is needed, a better choice would be shared or global memory.

Intra-block thread communication can be achieved with the usage of shared memory. Variables declared with the `__shared__` attribute within a kernel are allocated in shared memory which resides inside the SM. Since it has faster access than global memory, it is commonly used to implement programmer controlled caches. The amount of shared memory per SM is configurable on the Kepler architecture. 64 KB per SM can be allocated between shared memory and a non-programmer controlled, on SM, L1 cache which is used only for local memory on most Kepler based GPUs. However, certain Kepler cards are configurable to allow global and local memory reads to be cached in the L1 cache [51].

Despite the name, local memory is physically stored in global memory and has the same latency when it is not cached in the L1 cache. As mentioned previously, local memory is used when there is insufficient register space for the threads of a block. Local memory will also sometimes be used for large structures and arrays declared within a kernel. The issue for arrays is that registers can not be dynamically indexed. Thus, the only type of arrays that can be stored in registers are those that are only accessed using constant indexes. The NVIDIA compiler provides options that allow a programmer to view the number of registers and the amount of local memory used by each kernel.

Global memory is the largest and highest latency memory pool available to a kernel. Large datasets that are passed from the CPU to the GPU for processing are typically, at least initially, stored into global memory. When a warp encounters a global memory read or write, the number of memory transactions needed to complete the action for all threads depends on the size of the element accessed and the address each thread is using. Global memory can be accessed by the warp in transactions of 32, 64, or 128 bytes. Optimal global memory throughput is achieved if the warp threads make coalesced memory transactions. A coalesced transaction occurs if the memory access is aligned and threads within the warp make contiguous address requests. The benefit of a coalesced memory access is that the warp can satisfy the requests from all 32 threads with one access to global memory. Given the high latency of global memory, this setup is highly desirable. If the transaction can not be coalesced, the warp will have to make multiple, serialized requests to global memory in order to satisfy the requests from the threads. See listing 4.4 for an example of a coalesced memory read. Assuming the base address of data is a multiple of 32, each warp running this kernel performs one 32-byte read. Figure 14a shows how this will look for warp 0. In Figure 14b, the threads are not reading increasing consecutive address; however, the accesses are still contiguous. Therefore, the warp will be able to perform one 32-byte read. Figure 14c represents a worst-case scenario. The threads within the warp are reading from addresses that are spread throughout global memory. The warp will need to make 32 serialized 32-byte memory accesses to global memory. Each 32-byte access only returns 4-bytes, the size of a float, of data that is actually needed. The NVIDIA CUDA Best Practices Guide [55] gives priorities to techniques that can be used to optimize kernel execution time. Not surprisingly, ensuring global memory accesses are coalesced is given their highest priority.

Listing 4.4: Example of a coalesced memory access

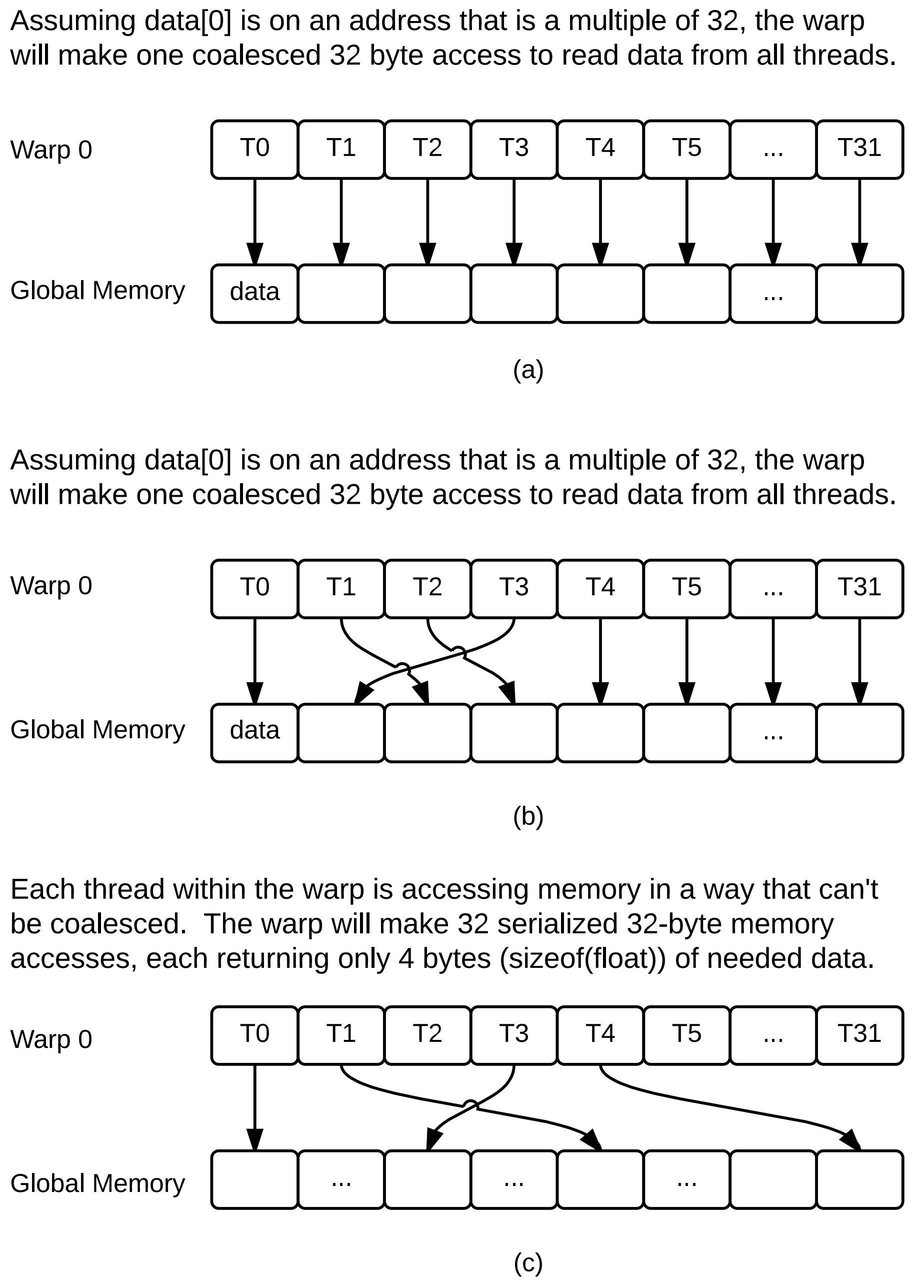
```
__global__ void myKernel(float* data)
{
       int myIdx = threadIdx.x;


       float myData = data[myIdx];
}
```

4.1.2 Kernel Programming and Libraries

When programming on a CPU, greater performance may be available if the programmer is willing to move to a lower level of hardware abstraction. For certain time critical sections of code, the best option may be inlined assembly. This option is also available within CUDA kernels via the parallel thread execution [52] (PTX) instruction set architecture. PTX assembly can be inserted inline by wrapping it within an `asm` statement. The `nvcc` compiler also can optionally output the PTX assembly for compiled kernels. This can be extremely useful during kernel optimization.

On the alternative side of the spectrum there are many highly optimized CUDA based libraries that can be used for a wide variety of parallel tasks. Two popular examples include CUB [57] and cuRAND [50] which were used within the simulators created for this work. CUB, or CUDA Unbound, is a parallel algorithm library specifically designed for CUDA. CUB offers architecture specific optimizations and provides a large number of configurable options one can use to fine tune an operation to achieve maximum performance. Another extremely popular and well supported option in this realm is Thrust [53]. Thrust is designed to appear to be similar to the C++ STL. It provides options to do all of the things that CUB can accomplish and more. Another potential benefit for Thrust is that it can be made to operate with CUDA, TBB [29] and OpenMP [6]. However, Thrust seemed to lack the tuning options available in CUB and in our testing CUB outperformed it.

The cuRAND library was used for pseudorandom number generation within the


Assuming data[0] is on an address that is a multiple of 32, the warp will make one coalesced 32 byte access to read data from all threads.

Assuming data[0] is on an address that is a multiple of 32, the warp will make one coalesced 32 byte access to read data from all threads.

Each thread within the warp is accessing memory in a way that can't be coalesced. The warp will make 32 serialized 32-byte memory accesses, each returning only 4 bytes (sizeof(float)) of needed data.

Figure 14: Memory coalescing examples

simulation. The default XORWOW [46] algorithm was used, and each LP was assigned a unique sequence number to ensure their values were not statistically correlated. The only negative with the use of this library is that the size of the random state for each LP is not a size that allows coalesced reads and writes to memory from a warp. Depending on the specific needs on the quality of random numbers needed for the simulation, this overhead could be reduced by sharing the random state among several LPs.

4.2 Simulator Overview

Three different implementations of a CUDA based parallel discrete-event simulator were created and compared for this work. Each simulator uses the same synchronous conservative LBTS algorithm to enforce time causality among the LPs. All three of the simulators utilize the CUB parallel algorithms library for common parallel algorithms such as sort and reduce. The main difference between the three simulators is how each organizes the event list between processing of events. In this section, an overview of the simulators is given, and the differences among the three simulators are explained.

For all three versions of the simulator, the state of each LP and the event list are stored in global memory on the GPU. Each LP stores its current time, the number of events it has processed, and the state of its random number generator. One event list stores all of the events for the LPs. Storing all of the events in one global event list makes calculating the lowest timestamp an efficient operation since the events are in a contiguous block that can be easily coalesced. As events are processed, the memory is reclaimed and reused. As such, the individual running the simulator must pre-allocate enough memory to handle the maximum number of events that could ever be in the simulator at any point in time. However, it is typically not overly difficult, and this type of approach is also used by other discrete-event simulators such as ROSS [11].

4.2.1 Memory Organization

As discussed in the previous section, it is highly beneficial to have warps perform coalesced global memory accesses. In order for this method of access to occur, the objects being accessed are limited to certain sizes. Listing 4.5 shows a sample layout for an event list. This type of layout is known as an array of structures.

Listing 4.5: Array of Structures

```
struct Event {
  //unique event id
  uint32_t event_id;

  //lp that should process this event
  uint32_t event_lp;

  //time the event should execute
  float   event_time;

  //random event data
  int     event_data_a;
  int     event_data_b;
};

Event event_list[256];
```

Each Event object in this example is 24 bytes. A warp attempting to read or write data into this event list will be required to perform 32 32-byte global memory accesses because the structure is not a size that can be coalesced into a single memory transaction. An alternative layout is shown in listing 4.6. This type of layout is known as a structure of arrays.

Listing 4.6: Structure of Arrays

```
struct EventList {
  uint32_t event_id[256];
  uint32_t event_lp[256];
  float    event_time[256];
  int      event_data_a[256];
  int      event_data_b[256];
};

EventList event_list;
```

By laying the data out in this manner, all memory operations performed on the event list can be coalesced because all of the data is stored in contiguous arrays of 4 bytes. The only downside to this approach is that it makes it somewhat more difficult to sort the data since parts of each event are located in five different arrays. However, since this is such a standard way to store composite data types in GPU memory, numerous approaches exist. All three of the simulators created used this approach to organize data in GPU memory.

4.2.2 Simulator Execution

At the start of execution, the CPU allocates space on the GPU for the LP state and the event list. Simulator specific constant data like the number of LPs and the simulation stop time are copied into GPU constant memory so it is quickly accessible to all threads. The CUB parallel library requires that temporary work memory used by its algorithms be allocated ahead of time. The library is used within the event processing loop so having the ability to do this is quite beneficial. Once this memory has been allocated, the CPU calls an initialization kernel which assigns initial values to the GPU memory that was allocated. The initialization kernel also creates a stop event for each LP in the simulation. Kernel calls in CUDA are asynchronous and control returns immediately to the CPU once the request for kernel invocation has been completed. Therefore, following the initial kernel invocation, the CPU pauses and waits for it to complete before proceeding into the event processing loop.

The overall structure of the event processing loop is the same for all three versions of the simulator and is shown in pseudocode in listing 4.7. At the beginning of the loop, the smallest timestamp within the simulation is determined by performing a parallel reduction on the entire event list. If this value is greater than the specified stop time, then the simulation is complete and control is passed out of the loop. If there are more events to process, the event list is processed so events that are considered safe can be executed during the final kernel which executes the safe events and updates the state within the LPs.

Listing 4.7: The main event processing loop

```
while(true)
{
  min_timestamp = reduce(event_list.event_time, min)
  if(min_timestamp >= stop_time)
      break
  //this part varies by simulator
  processed_event_list()
  process_safe_events(m_timestamp)
}
```

The difference among the three simulators occurs within the `process_event_list` method. All three versions sort at least part of the event list. The sort that was used is the `Device::Radix` sort provided within the CUB library. On more recent hardware, this sort has been shown to be able to sort approximately 2.12 billion 32-bit unsigned integers per second [44]. When sorting the event list, the simulators use a 64-bit sort instead of a 32-bit because they are sorting by the destination lp and the event time. Radix sorts do work with floating point values as long as they aren't negative. Thankfully that is never an issue when measuring time. In the next three subsections further details are provided for each simulator.

4.2.2.1 Simple Simulator

The first of the simulators is referred to as the `simple simulator` because it does the least amount of manipulation to the event list prior to event execution. The first step that it takes is to sort the event list so it is ordered by LP identifier and then time as shown in figure 15a. Once the events are sorted it then scans the event list to find the first entry by each LP which is stored as an offset as shown in figure 15b.

During `process_safe_events` each LP in run on a thread. The LP reads its offset, which can be coalesced, and then reads in the event, which most likely will not be coalesced because there will typically be more than one event per LP. The LP then determines if the event is safe to process based on the smallest timestamp and the lookahead.

4.2.2.2 Partition Simulator

The second simulator differs from `simple simulator` in that it attempts to reduce the uncoalesced memory reads and writes that occur during the event processing phase. The simulator first performs a sort on the event list. Then, instead of writing an offset, this simulator marks the first event for each LP. See figure 16b for an example. After this is complete a CUB `Device::Partition` is used to move all of the marked events to the front of the event list as shown in figure 16c. Since every LP is given an end event, every LP will have at least one event so the `process_safe_events` kernel

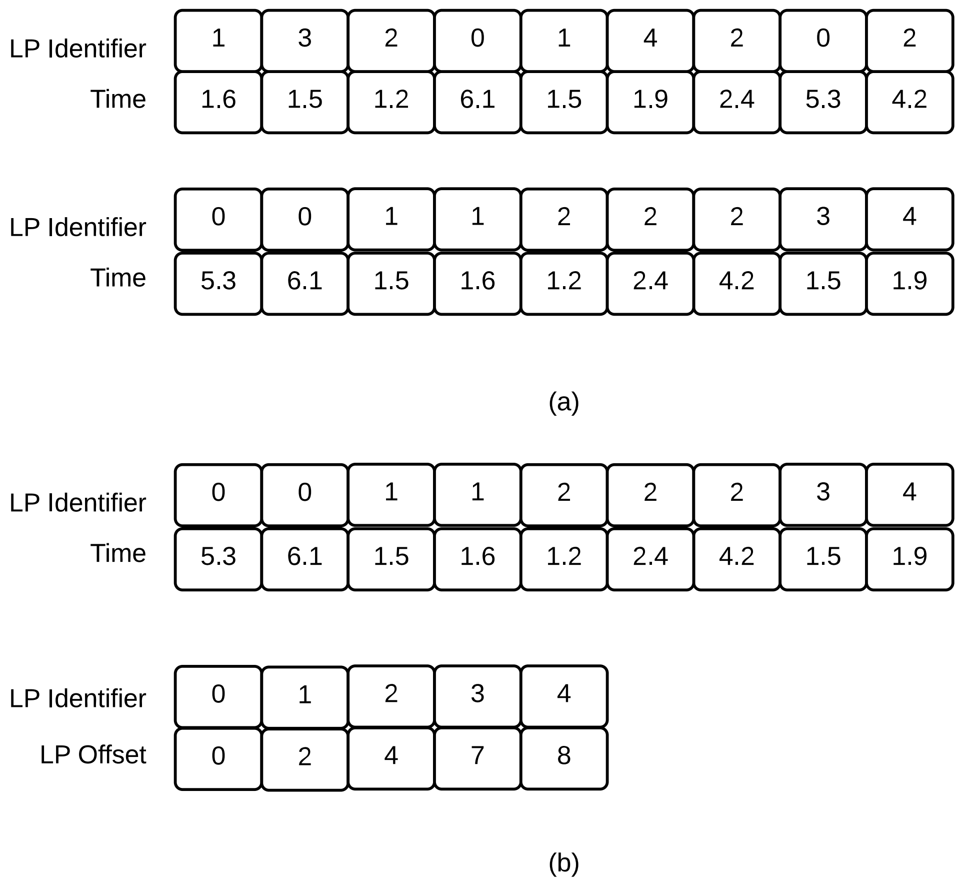


Figure 15: Simple Simulator: (a) The event list is first sorted. (b) Then an offset to the first event for each LP is calculated

will always be able to achieve coalesced read and writes for both LP and Event data.

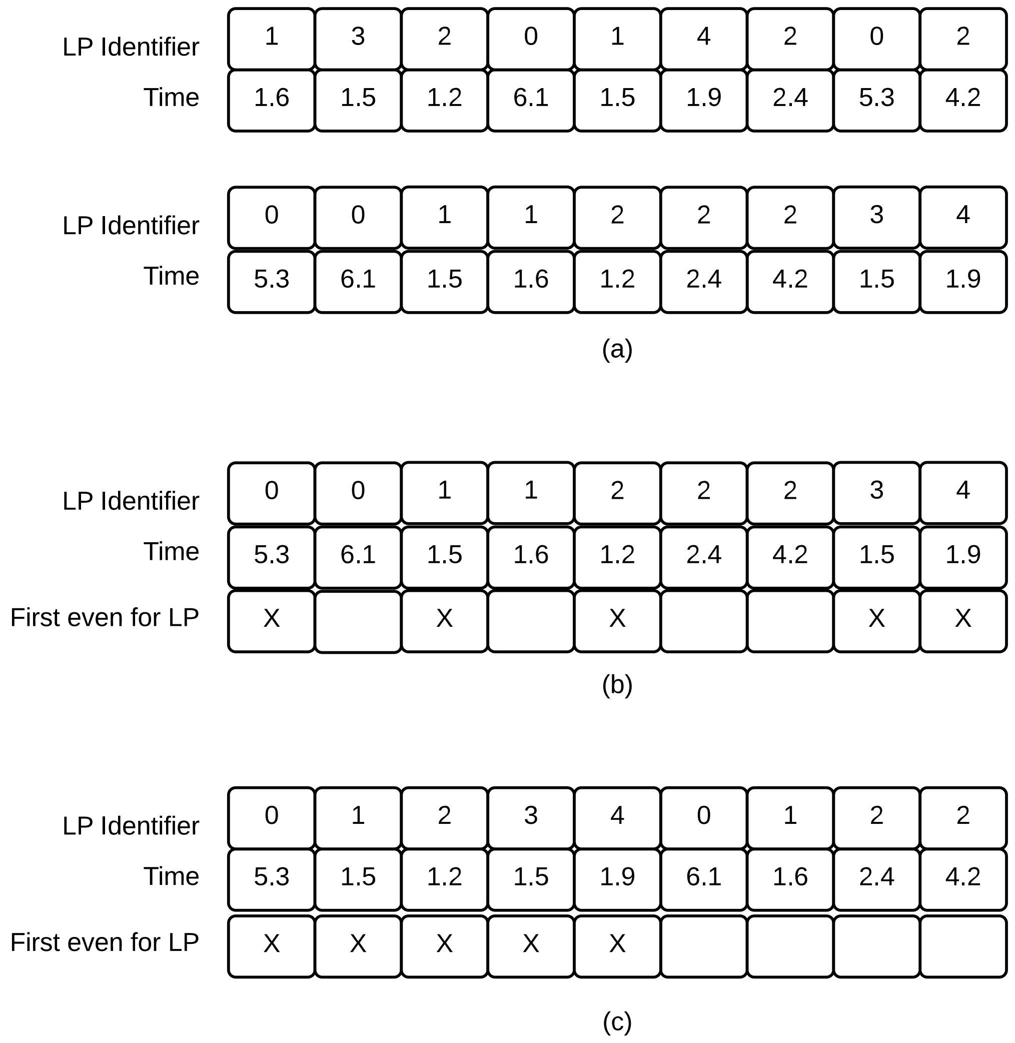
4.2.2.3 Reduce Simulator

The final simulator does the most amount of processing on the event list prior to proceeding into the event processing phase. After the lowest time stamp is calculated a kernel is invoked that examines every event in the event list and marks it if it is within the safe window for that LP. Obviously, multiple events could be marked per LP. Subsequently, another kernel partitions the event list based on the marks as was done in 16b-c. This kernel also returns N, the number of events that were marked. This is followed by radix sort which only sorts the first N events within the event list. A scan of the sorted event list is then performed to mark the first event for each LP and then the event list is partitioned again. In the end, there is one event for each LP that has a safe event at the front of the event list.

The `reduce simulator` does the same thing as the `partition simulator` with the addition of an initial partition to filter out events outside the safe window. The initial partition has two major consequences. First, if there are a significant number of events outside the safe window, subsequent manipulations of the event list are much faster since they are dealing with fewer events. Second, there are likely to be LPs without an event within the first N events of the event list because they may all be at the back of the list due to the partitioning. That was not the case with the `partition simulator` because every LP had at least an end event. Therefore the nice coalesced memory accesses to both the event list and the LP state are no longer guaranteed.

4.3 Evaluation

The PHOLD benchmark [23] was used to test the performance of the three simulators. During execution, as each LP processes an event it creates a new event. The new event will occur at some fixed amount of time in the future plus some random amount



LP Identifier	1	3	2	0	1	4	2	0	2
Time	1.6	1.5	1.2	6.1	1.5	1.9	2.4	5.3	4.2

LP Identifier	0	0	1	1	2	2	2	3	4
Time	5.3	6.1	1.5	1.6	1.2	2.4	4.2	1.5	1.9

(a)

LP Identifier	0	0	1	1	2	2	2	3	4
Time	5.3	6.1	1.5	1.6	1.2	2.4	4.2	1.5	1.9
First even for LP	X		X		X			X	X

(b)

LP Identifier	0	1	2	3	4	0	1	2	2
Time	5.3	1.5	1.2	1.5	1.9	6.1	1.6	2.4	4.2
First even for LP	X	X	X	X	X				

(c)

Figure 16: Partition Simulator: (a) The event list is first sorted. (b) The first event for each LP is marked. (c) A partition is used to pull the marked events to the front of the event list.

of padding. The fixed amount of padding is added to the random time to prevent instantaneous events. The LP will then either send the event off to some randomly selected peer or place the event in its own queue. Since each LP is given an individual pseudorandom number stream, the number of events that will be processed at each LP will be the same across all three simulators for a given seed and simulation size.

For the first set of experiments the simulation was run for 240 seconds, the lookahead was 4.0 seconds, and the local insertion rate was .9. The number of LPs participating in the simulation was varied from 1,024 to approx 16.8 million. Twenty execution runs were performed for each configuration and the run times were averaged. The experiments were run using a NVIDIA GeForce GTX TITAN Black using CUDA driver version 6.5. The GPU has 6GB of memory, 2880 CUDA Cores and a clock rate of 980 MHz.

Table 1 shows the number of events processed by each simulator. The numbers are the same for each simulator type because the same seed and sequence numbers were used for the LPs to ensure that each simulator is performing the same amount of simulation work. The totals are shown here for verification purposes. Table 2 and figure 4.3 show the number of events each of the simulators were able to process every wall clock second. The data clearly shows that the `reduce simulator` was the most efficient when the number of events in the event queue became extremely large. The ordering is reversed at the lowest number of LPs tested, the `partition simulator` was the fastest while the `reduce simulator` was the slowest.

4.4 Discussion

For comparison of performance, two other works that examined the execution rate of a PHOLD simulation are provided. Both of these solutions made use of multiple CPU cores as opposed to a single GPU. While there has been some valid criticism provided for direct CPU/GPU comparisons [36], given the relatively simple nature

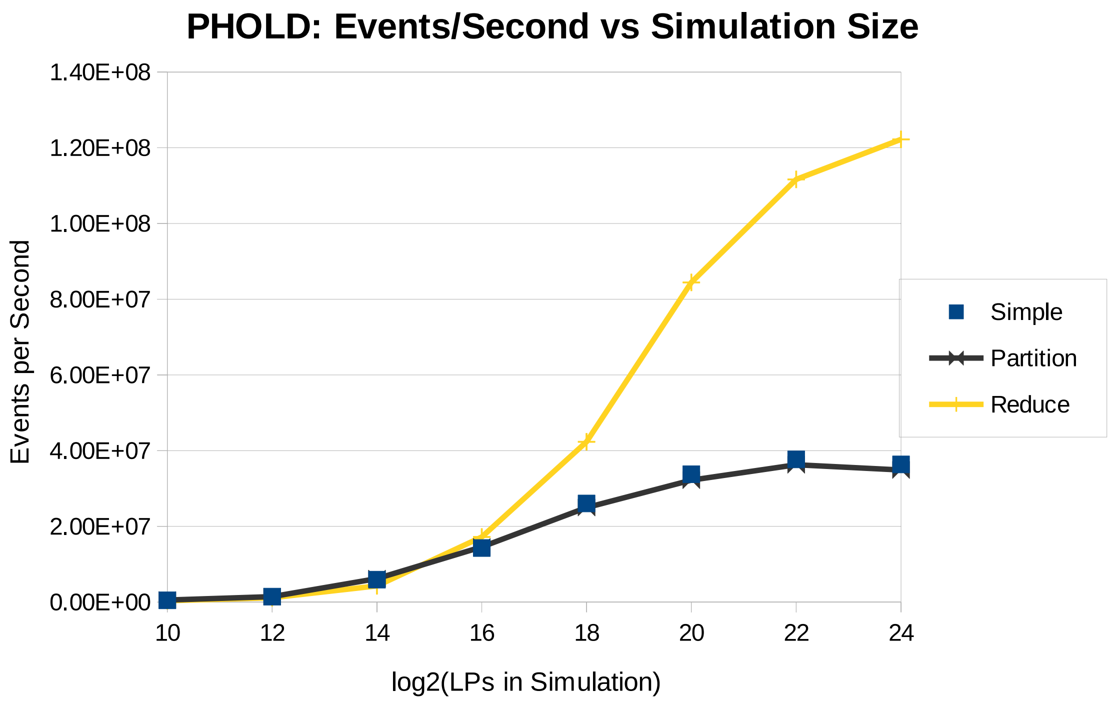


Figure 17: The rate at which each simulator instance is able to process events by simulation size. The rate is in events per second.

Table 1: PHOLD: Total number of events processed

LPs	Simple	Partition	Reduce
2^{10}	190,928	190,928	190,928
2^{12}	765,704	765,704	765,704
2^{14}	3,057,536	3,057,536	3,057,536
2^{16}	12,237,808	12,237,808	12,237,808
2^{18}	48,950,952	48,950,952	48,950,952
2^{20}	195,858,164	195,858,164	195,858,164
2^{22}	783,491,776	783,491,776	783,491,776
2^{24}	3,133,755,472	3,133,755,472	3,133,755,472

Table 2: PHOLD: Events per second

LPs	Simple	Partition	Reduce
2^{10}	451,635	530,356	352,006
2^{12}	1,423,474	1,523,474	1,132,698
2^{14}	5,928,275	6,214,504	4,277,518
2^{16}	14,270,830	14,568,819	17,187,933
2^{18}	26,037,740	24,974,976	42,330,466
2^{20}	33,768,649	32,213,513	84,421,622
2^{22}	37,697,887	36,272,767	111,646,684
2^{24}	36,367,972	34,893,635	122,218,111

of the PHOLD benchmark, few of the typical concerns are valid. Specifically there is no large amounts of data that are being passed back and forth between the CPU and GPU during simulation execution. Not measuring this transfer time is a frequent criticism of CPU/GPU performance comparisons.

Barnes, Jefferson, Carothers and LaPre [3] ran the PHOLD simulation on 1,966,080 cores using the ROSS simulator on a Blue Gene/Q Super Computer. They managed to achieve a sustained rate of 504 billion events per second. While there is a fairly large gap between their event rate and ours, there is also a fairly large price gap between a Blue Gene/Q Super Computer and the GTX graphics card that we used. Another difference between this work and ours is that they used a Time Warp optimistic synchronization algorithm. Performance of simulators using this synchronization algorithm can vary significantly depending on the frequency and severity of the rollbacks that must be handled.

Recent work by Ivey, Swenson and Riley [30] examined PHOLD performance on distributed ns-3 which, like this work, uses conservative time synchronization. ns-3 allows the simulation user to choose between using null message or a granted time window algorithm for time synchronization. Using up to 1,024 cores they reported event rates of up to 27 million events per second.

In other related GPU work, Kunz et al.[34] proposed a scheme that would use the SIMD characteristics of GPUs to perform large-scale, cost-effective parameter studies. Their method allowed users to concurrently execute multiple runs of a simulation using different parameters. This type of simulation approach fits well into the SIMD architecture of the GPU and the authors reported considerable speed-up compared to serial execution. This work differs in scope from ours since our efforts focus on using the parallel computation abilities of a GPU to run large single simulations.

Park and Fishwick[58] examined using GPUs for simulating a queuing network. In their work the future event list (FEL) is decomposed into multiple subFELs so it can be processed in parallel by multiple threads. Their approach also utilized a time approximation scheduling routine in order better group events into safe time windows. Our approach differs in that it uses no time rounding and our event list is not subdivided.

Perumalla et al.[61] used a time-stepped approach to create a large-scale GPU traffic simulation based on a queueing model. By using this type of approach, the authors did not have manage a sorted event list nor did they have to keep track of individual vehicles. The road topology was broken into nodes which contained floating-point values representing the flow of vehicles into and out of each road segment. Each time step is broken into two phases, a split phase and a gather phase. While this approach may be faster than a discrete-event simulation, it allows for oddities such as vehicles splitting into multiple pieces only to be recombined at their final destination. In their work they reported simulating up to 2 million nodes representing the road network of the state of Texas.

This chapter presented three techniques for handling large-scale discrete-event simulations on a GPU using conservative time synchronization. Using these techniques we are able to achieve an event rate of over 120 million events per second simulation approximately 16.7 million logical processes. These results indicate that GPUs can be a good fit for large-scale discrete-event simulation and further research should be performed on this subject.

CHAPTER V

SIMULATING LARGE TOPOLOGIES WITH BRITE AND CUDA DRIVEN GLOBAL ROUTING

Due to the complexities of the internet, the use of simulation tools is essential for examining the effectiveness of new protocols and the practicality of new internet applications. In many cases, in order to get a full understanding of the impact of the new protocol or application, the simulation needs to be performed on a large-scale topology with many competing sources of traffic. However, constructing large-scale topologies that exhibit the same characteristics of real networks is not a trivial task. Furthermore, once the topology is constructed, the generation of the routing tables needed to direct packets across the network can be extremely time intensive. The goal of this work is provide ns-3 users with the means to be able to generate large-scale typologies which are representative of real networks and simulate them in a time efficient manner.

This chapter presented two new models for the popular ns-3 [49] network simulator are introduced [70]. The first module is an ns-3 interface to the BRITE topology generator library. BRITE allows users to quickly create customizable, large-scale topologies for simulation. Our module takes the topology generated by BRITE and creates an identical representation in ns-3. Our module also provides helper methods which makes it easy for the user to configure the nodes in the generated topology. Futhermore, the BRITE module also works with MPI, allowing even larger networks by spreading the processing burden across multiple CPUs. The second module is a new global routing module that has been specifically designed to work with large-scale topologies. This module takes advantage of the massive parallel execution architecture of GPUs using NVIDIAs CUDA programming model. Using our new global routing module, we are able to perform global routing tasks at a rate much faster than is possible with the current global routing protocol in ns-3.

5.1 BRITE

BRITE [43], the Boston University Representative Internet Topology Generator, is a common tool for generating realistic internet topologies for simulation. BRITE is a widely used topology generator that is used in many simulation applications such as ns-2 [48], GTNetS [64], SSF [31] and OmNet++ [71]. Topology construction in BRITE is highly customizable and is controlled by user provided configuration scripts. Included in BRITE is a front-end GUI to ease the process of creating these configuration files. Using BRITE, a wide variety of different topologies can be generated including scale free (power law) networks which are commonly seen within the internet [2].

BRITE uses two main algorithms for placing nodes in generated topologies, the Waxman [73] model and the Barbasi-Albert [2] model. The Waxman model is based originally on the Erdös-Renyi [18] random graph model. The Waxman model differs from the Erdös-Renyi random graph model in that all generated nodes are placed on a plane and connectivity between nodes is based on euclidean distance. Specifically, the probability that two nodes, x and y, are connected is given by the formula:

$$P(x,y) = \alpha e^{-d/\beta L} \tag{1}$$

where d is the distance from node x to node y, L is the euclidean diameter of the network, and α and β are parameters.

The other model used by BRITE is the Barbasi-Albert model. This model can be used to create scale-free networks. This model uses incremental growth and preferential attachment to create topologies which conform with a power law. In this model, nodes are connected to the topology incrementally. As each node is added, it is more likely to connect to nodes that are highly connected. Specifically, the probability that node x, which is joining the network, is connected to node y, a node already in the network, is given by:

$$P(x, y) = \frac{d_y}{\sum_{k \in V} d_k} \tag{2}$$

where d_i is the degree of node i and V is the set of nodes already in the network. Therefore the denominator is the sum of the outgoing edges of all nodes already in the network.

There are three major types of topologies available in BRITE: Router, Autonomous System (AS), and Hierarchical which is a combination of AS and router. For the purposes of ns-3 simulation, the most useful topologies are likely to be Router and Hierarchical because both of these result in a graph of routers. Router level topologies are generated using either the Waxman model or the Barbasi-Albert model. Each model has different parameters that effect topology creation and these are specified in the configuration file. For flat router topologies, all nodes are considered to be in the same AS.

Hierarchical topologies contain two levels. First there is a top level AS topology, which can be constructed using either the AS-Waxman model or the AS-Barbasi-Albert model. These models are equivalent to the router level versions of the models. After the top level is constructed, for each node in the AS graph, a router level topology is constructed. These router level topologies can be created using a different model and different parameters than used in the AS model. BRITE then connects the different router level topologies using the original AS model as a guide. The algorithms used to make these connections were borrowed from the GT-ITM topology generator

[10]. At this point the topology is fully connected and BRITE moves on to assigning bandwidths.

Once the topology is generated, the next step BRITE performs is to assign bandwidths to the links. BRITE offers four different possible distributions for assigning bandwidths: Constant, Uniform, Exponential and Heavy-tailed. The parameters used to control these distributions, BWdist, BWmin and BWmax, are specified in the BRITE configuration file. It should be noted that BRITE treats bandwidth values as unit-less. How ns-3 handles these values will be explained in in the next section.

5.2 BRITE integration with ns-3

The ns-3 interaction with BRITE occurs through the `BriteTopologyHelper` class. The construction of the BRITE topology is controlled by a user provided configuration file. Therefore the constructor for the helper takes as a parameter a string specifying the path to a BRITE configuration file. Users can also optionally specify a BRITE seed file to be used in construction of the topology. During topology construction, BRITE uses a pseudo-random number generator and therefore seeds are needed to create different topologies with similar characteristics.

An alternative to providing a seed file is to use ns-3s uniform distribution random number generator to automatically generate the seed values required by BRITE. An instance of this class is instantiated within the topology helper. The helper class provides an accessor method called `AssignStreams` to set the stream value of the generator. If a BRITE seed file is not passed in via the constructor, the helper will use the random number generator to generate the seed values. When the BRITE library executes, it automatically generates a file named `lastSeedFile` that contains the seeds used in the generation of the last topology. Therefore, if an experimenter wanted to save the seed values in order to regenerate the same topology again, they could save a copy of this file prior to another invocation of the BRITE library.

Once an object of type `BriteTopologyHelper` has been created and configured, the next step is to actually create the new topology. To facilitate this, a method called `BuildBriteTopology` is provided by the helper class. There are two versions of this method. Both versions accept as a parameter an `InternetStackHelper` instance. This stack is installed on all nodes created in the BRITE topology.

The BRITE module has also been designed to optionally work with MPI. One version of `BuildBriteTopology` accepts an unsigned int representing the system count. Each AS in the generated topology is assigned a system number based on a modulus divide. Then, when the helper is generating the ns-3 topology, ns-3 nodes are created on the system corresponding to the system number of the AS they belong to. The system number given to an AS can be found by using the `GetSystemNumberForAs` method provided by the helper.

The `BriteTopologyHelper` uses the BRITE library to generate a topology and then uses this BRITE topology to generate an equivalent version in ns-3. Every node in the BRITE topology has a matching node in the ns-3 topology. Every link in the BRITE topology is represented by a point to point link in the ns-3 topology. When assigning bandwidths to the point to point links in the topology, the helper assumes the values provided by BRITE are specified in Mbps. Therefore, when creating a BRITE configuration script, it is important to specify the bandwidth parameters in Mbps.

Listing 5.1: Example using ns-3's new BRITE API

```
//Add required header files
#include "ns3/brite-module.h"
// Invode the BriteTopologyHelper and pass in a BRITE configuration file
// This will use BRITE to build a graph from which we can build the
// ns-3 topology
BriteTopologyHelper bth (confFile);
```

```
InternetStackHelper stack;
// The following builds the BRITE topology and then creates a ns-3
// representation. The stack helper is passed in so all newly
// created nodes can have a stack installed prior to links being
// created
bth.BuildBriteTopology (stack);

Ipv4AddressHelper address;

// A small subnet is used because all of the point to point links
// created in the topology are assigned IP addresses as a separate
// network. Therefore by using a small subnetwork we don't waste
// any of the address space
address.SetBase("10.0.0.0", "255.255.255.252");
bth.AssignIpv4Address (address);
// Iterate through all of the nodes in the BRITE generated topology
for(uint i = 0; i < bth.GetNAs(); ++i)
       for(uint j = 0; j < bth.GetNNodesForAs(i); ++j)
       {
              //perform action on all nodes in topology
              Ptr<Node> node = bth.GetNodeForAs (i, j);
       }
// Iterate through all of the leaf nodes in the BRITE generated topology
for(uint i = 0; i < bth.GetNAs(); ++i)
       for(uint j = 0; j < bth.GetNLeafNodesForAs(i); ++j)
       {
              //perform action on all leaf nodes in topology
              Ptr<Node> leaf = bth.GetLeafNodesForAs (i, j);
       }
```

In the process of topology generation, BRITE classifies nodes depending on their connectivity in the network. Router level nodes are classified into one of the following categories: Leaf, Border, Backbone, Stub and None. Typically when working with BRITE generated topologies the most useful of these are the leaf nodes. The leaf nodes can be used to attach any other ns-3 topology, including other ns-3 BRITE topologies. To provide access to the leaf nodes, the `BriteTopologyHelper` class provides a `GetLeafNodeForAs` method which accepts as parameters an AS number and a leaf node index. Access to all of the nodes in the topology is available using the `GetNodeForAs` method. See listing 5.1 for an example.

Once the ns-3 version of the topology has been generated, the next step is to assign IP addresses to the nodes in the newly created topology. This is accomplished using the method `AssignIpv4Addresses` or `AssignIpv6Addresses`, provided by the helper. When assigning IP addresses to the interfaces in the topology, each point to point link is treated as a separate network. Therefore it is important, especially for Ipv4, to set the size of the subnetwork to an appropriate size. Otherwise a large portion of the available address space will be wasted.

5.3 Floyd-Warshall and CUDA

The Floyd-Warshall algorithm is a dynamic programming algorithm that can be used to solve the all-pairs shortest path problem on a graph. The algorithm runs in $\theta(N^3)$ time where N is the number of nodes in the graph to be solved. The algorithm operates on a NxN matrix. The initial matrix is an adjacency matrix where cell(i,j) specifies the weight of the edge going from node i to node j. If nodes are not directly connected, the initial value for the cell is infinity. The algorithm then proceeds as follows:

Basic Floyd-Warshall

```
1: for k = 1 to N do
```

```
    for i = 1 to N do
        for j = 1 to N do
            if W[i][k] + W[k][j] ¡ W[i][j] then
                W[i][j] = W[i][k] + W[k][j]
            end if
        end for
    end for
end for
```

In general terms the algorithm works as following. At each step, k, the distance from i to j, for all pairs in the matrix, is checked to see if it can be improved by using vertex k as a intermediate. A full description of the algorithm and a proof of its correctness was described by Corman [14].

This algorithm is an attractive fit for CUDA because all of the comparisons along i and j for a given k are completely independent and can be performed in parallel. Furthermore, the algorithm contains no divergent paths, one if and no else, so it maps extremely well into CUDA's SIMD execution architecture.

The Floyd-Warshall algorithm provides the distance on the shortest path between all pairs. However for routing we are interested in the actual path itself. Thankfully with a slight modification, the Floyd-Warshall algorithm can provide enough information to generate the shortest path:

Floyd-Warshall with Path

```
for k = 1 to N do
    for i = 1 to N do
        for j = 1 to N do
            if W[i][k] + W[k][j] ¡ W[i][j] then
                W[i][j] = W[i][k] + W[k][j]
                next[i][j] = k
```

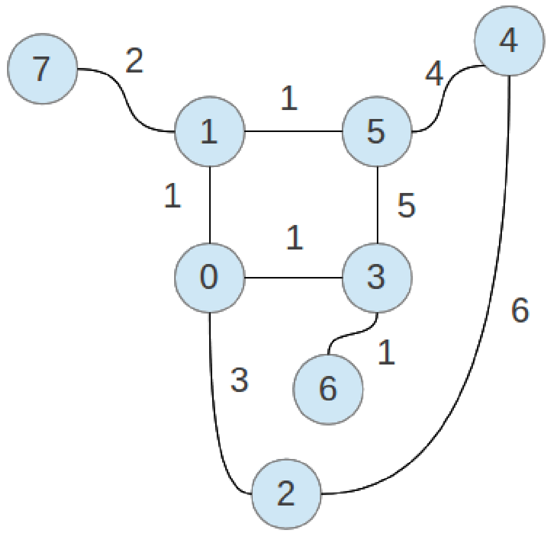
```
7:             end if
8:         end for
9:     end for
10: end for
```

Here variable next is another NxN matrix. The values for next are initialized to j for all cell(i,j). Following the completion of the algorithm, next will contain the highest index vertex, k, that i must travel through along the shortest path to j. The next matrix also contains the highest index vertex, k_1, that i must travel through along the shortest path to k and so on. Using this information, it is possible to determine the next hop for i along the shortest path to j.

Figure 18 provides an example of this process. Following the completion of the Floyd-Warshall algorithm (b), for node 5 to travel to node 6 it must travel through node 3, which is a neighbor of node 5, but it is not along the shortest path. The next matrix is examined to determine the shortest path from node 5 to node 3 and it is found that it is through node 1. The matrix is again examined to determine the shortest path from node 5 to node 1 and it is found that it is through node 1. Node 1 is now saved as the next hop on the shortest path from node 5 to node 6.

5.4 Generation of routes

The first step performed with our new routing process is to examine the ns-3 node topology and generate the information required for routing. Three main pieces of information are gathered in this process. First is an IP address to node number map. In order to reduce the amount of data that needs to be transferred to the GPU device and to speed up the route computation, we compute routes node to node as opposed to net device to net device. Because of this, we need an efficient way to translate a net devices IP address to a node number. This map is generated once, stores all of the IP address in the topology and is made available to all nodes participating in the



Initial Adjacency Graph

	0	1	2	3	4	5	6	7
0	0	1	3	1	-	-	-	-
1	1	0	-	-	-	1	-	2
2	3	-	0	-	6	-	-	-
3	1	-	-	0	-	5	1	-
4	-	-	6	-	0	4	-	-
5	-	1	-	5	4	0	-	-
6	-	-	-	1	-	-	0	-
7	-	2	-	-	-	-	-	0

Initial Next Graph

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	0	1	2	3	4	5	6	7
2	0	1	2	3	4	5	6	7
3	0	1	2	3	4	5	6	7
4	0	1	2	3	4	5	6	7
5	0	1	2	3	4	5	6	7
6	0	1	2	3	4	5	6	7
7	0	1	2	3	4	5	6	7

(a)

Shortest Path Distance

	0	1	2	3	4	5	6	7
0	0	1	3	1	6	2	2	3
1	1	0	4	2	5	1	3	2
2	3	4	0	4	6	5	5	6
3	1	2	4	0	7	3	1	4
4	6	5	6	7	0	4	8	7
5	2	1	5	3	4	0	4	3
6	2	3	5	1	8	4	0	5
7	3	2	6	4	7	3	5	0

Next Graph After FW

	0	1	2	3	4	5	6	7
0	0	1	2	3	5	1	3	1
1	0	1	0	0	5	5	3	7
2	0	0	2	0	4	1	3	1
3	0	0	0	3	5	1	6	1
4	5	5	2	5	4	5	5	5
5	1	1	1	1	4	5	3	1
6	3	3	3	3	5	3	6	3
7	1	1	1	1	5	1	3	7

(b)

Next Hop Node

	0	1	2	3	4	5	6	7
0	0	1	2	3	1	1	3	1
1	0	1	0	0	5	5	0	7
2	0	0	2	0	4	0	0	0
3	0	0	0	3	0	0	6	0
4	5	5	2	5	4	5	5	5
5	1	1	1	1	4	5	1	1
6	3	3	3	3	3	3	6	3
7	1	1	1	1	1	1	1	7

(c)

Figure 18: An example showing how FW can be used to construct shortest paths. (a) Initialization. (b) The results in the original adjacency matrix and the next matrix following the FW algorithm. (c) The final routes computed.

global routing.

The second piece of information generated in this process is the global adjacency matrix. This NxN matrix is used as the input to the Floyd-Warshall algorithm. To gather this information, we cycle through the net devices on a node and compile a list of adjacent net devices. Once this information is obtained, we fill out that nodes corresponding row in the adjacency matrix. Currently we are assigning each link with a weight of one; however, that is not a requirement. The way the kernel is programmed, it will work with any integer weight.

Finally, for each node, we create and store a partial routing table containing only its immediate neighbors. The information is stored in a map indexed by the neighbors node number and contains the interface number and the remote IP address of the adjacent device. This map allows us to quickly obtain the next hop routing information we need when it becomes time to route a packet. At this point all of the necessary information has been gathered from the ns-3 topology and control is passed to the shared library where the routes are computed.

Once control has been passed to the shared library, the first thing it does is allocate memory on the GPU for the adjacency matrix and the next matrix. Again, these are both N xN integer matrices. Next the adjacency matrix, generated from the ns-3 topology, is copied to the GPU device and the next matrix is initialized. Since the next matrix has a large number of values that must be initialized, this is done via a separate CUDA kernel program. This allows this process to occur quickly since each value can be initialized by an individual CUDA thread.

At this point, processing of the Floyd-Warshall algorithm can begin. As mentioned previously, during each step of the Floyd-Warshall algorithm all of the values in the working adjacency matrix need to be checked to see if a shorter path is found from vertex i to vertex j using vertex k as an intermediate. For a given k, all of the paths in the working matrix can be updated in parallel but synchronization needs to occur after each value for k. Since it is quite possible the number of paths in the topology will exceed the maximum number of threads per block, we cannot perform the entire algorithm with one kernel call. This is because it is not possible to synchronize between blocks within a kernel. The blocks may be scheduled in any order, in series, or in parallel. This is dependent on the GPU architecture and how the blocks are assigned to the available SMs. The CUDA API does however provide a way to synchronize between blocks outside of the kernel. The `CudaDeviceSynchronize` method halts CPU execution until all blocks submitted for execution finish.

Therefore our implementation makes a separate kernel call for each value of k and synchronizes at the CPU level after each call. There is some slight overhead for each kernel call; however, no data is being transferred between the CPU and GPU during this time so the performance impact is minimal.

Following the execution of the algorithm, the original adjacency matrix now contains the distance of the shortest path between each i, j pair and the next matrix contains the highest vertex index that i must travel through along the shortest path to j. The shortest path distance information stored in the original adjacency matrix is no longer needed for routing and is therefore discarded. The information that is needed is in the `next` matrix. However it is still not in the correct form. We are looking for next hop information and it is currently giving us a node that is potentially many hops down the line. However, as mentioned in the discussion of the Floyd-Warshall algorithm, next hop information for each pair can be obtained using the `next` matrix. This next hop information for each pair can be determined independently and therefore we use another CUDA kernel program to perform this.

Now that a next hop for each pair of vertices has been calculated, the next matrix is copied back from the GPU to the CPU and passed back from the shared library to the helper. The helper then stores it in a location that is globally accessible to all nodes participating in the global routing.

5.5 ns-3 routing implementation

Like all routing protocols in ns-3, our /tt Ipv4FwCudaRouting module derives from the /tt ns3::Ipv4RoutingProtocol base class and therefore implements `RouteOutput` and `RouteInput`. `RouteOutput` is called when a packet is provided by the transport layer and provides a route towards the destination. `RouteInput` is called when a packet arrives at a `NetDevice` and a decision must be made to either forward the packet onward or pass it up to the transport layer.

For `RouteOutput` and in the case where `RouteInput` is forwarding a packet, an instance of `ns3::Ipv4Route` must be assembled to provide the next hop information for the packet. Along with the source and destination for the packet, the information needed for this object is the output device and the next hop IP address. Our implementation obtains this information as follows. First the destination IP address is looked up in the global IP address to node number translation map to get the destination node number. Then the next hop for the destination node from the current node is looked up in the `next` matrix. Now that we have the next hop node number we use it to index the local routing table for this node to get the correct output device and remote IP address. At this point all of the information needed to route the packet has been obtained.

5.6 Evaluation

All of the experiments performed in this section were run on the Georgia Tech PACE computing cluster. The CPUs on the nodes we used are six-core AMD Opteron Processors running at 2.4 GHz. Each node has a total of 64 GB of available RAM. The GPUs used were T10 Tesla Processors with Compute Capability 1.3 and a total of 4 GB of available global memory.

For the first experiment we wanted to see how the current Ipv4 global routing implementation in ns-3 would perform with a BRITE generated topology with a

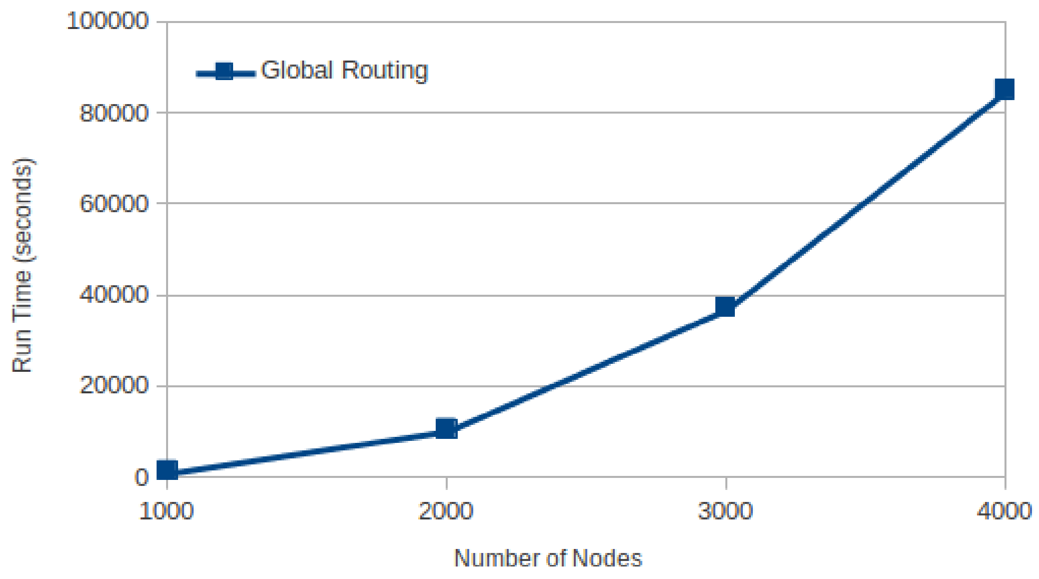


Figure 19: The run time of our test simulation varying the total number of nodes in the topology using Ipv4GlobalRoutingHelper::PopulateRoutingTables()

large number of nodes. For this experiment we used the BRITE hierarchical model with the AS-Barbasi-Albert model as the top level and used the Waxman model for the routers in each AS. We generated a total of 10 AS and the total number of nodes per AS was varied on each run. For each AS, we attached a node containing a UDP packet sink and a UDP source to the first leaf node. Each source was configured to send one packet of 100 bytes to every other sink. We did not set a stop time for the simulation, therefore it will run until the event list is exhausted. The complexity of the setup is admittedly simple, however the purpose of the experiment is to measure the time needed to generate the routing tables for the topology.

Figure 19 shows the results of this experiment. We varied the number of nodes from 1,000 to 4,000 nodes. We had originally hoped to go even larger; however, we began to run into wall clock time limits on the server on which the program was running. The results clearly show that there is a prohibitive cost when using the current global routing protocol with a large number of nodes. Even with the smallest

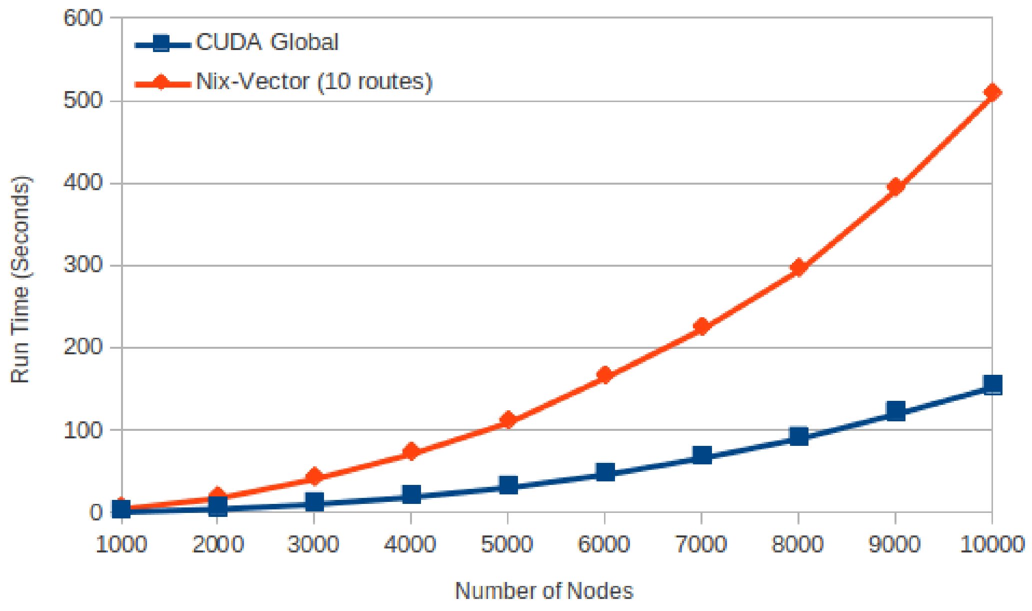


Figure 20: The run time of our experiment varying the total number of nodes in the topology using nix-vector routing and our CUDA global routing

topology we tested, 1000 nodes, the time to run a simulation which sends 90 packets is almost a 1/2 hour (1566 seconds).

For the second experiment we wanted to compare how our new CUDA global routing module compared to ns-3's Ipv4 global routing and also how it compared to nix-vector routing. nix-vector [65] routing does an on demand BFS to find the shortest path from source to sink. It then stores the steering information to guide the packet along that path within the packet itself. The node that generates the nix-vector also caches it so future transfers are done without the need for another BFS. For this experiment we used the same setup as in the previous experiment. We ran each iteration of the experiment ten times and took an average of the results. The results can be seen in figure 20.

The results show that both nix-vector routing and CUDA global routing clearly outperform the current global routing implementation for large topologies in ns-3.

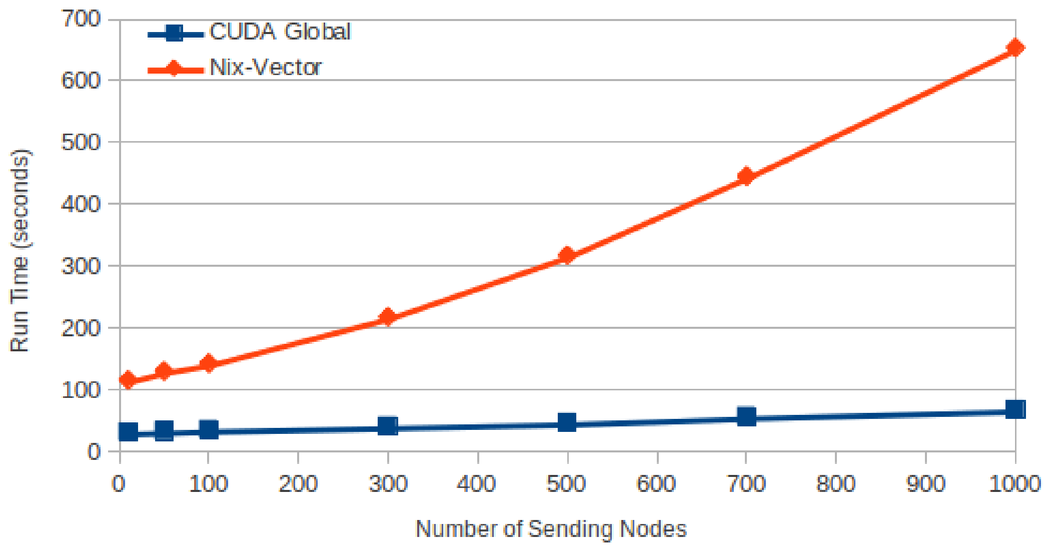


Figure 21: The run time of our experiment varying the number of TCP senders using Nix-Vector routing and our CUDA global routing

For 4000 nodes the run time for global routing was three orders of magnitude higher. Also, the results showed that our CUDA routing protocol outperformed nix-vector routing by between 3 and 3.5x for every topology size we tested. The important thing to remember here is what is being calculated in each simulation. In the nix-vector experiment, the shortest path is being calculated for ten pairs of nodes; however, in the CUDA routing experiment the shortest path is being calculated for every pair of nodes in the topology.

For the third experiment we fixed the number of nodes in the BRITE topology to 5,000. The same hierarchical model that was used in the previous two experiments was used. In addition, we then added a variable number of sending and receiving pairs to leaves in the generated topology. The minimum number of pairs we used was 10 and the maximum was 1000. Each sender was configured to send one packet via TCP to its receiver. By doing this we were able to vary the number of routes that the nix-vector protocol needed to calculate. Each configuration was run ten times and the results were averaged. The results can be seen in figure 21. As expected, for both experiments as the number of senders and receivers was increased, the run time of the experiments also increased. This is expected because the simulation has to handle more packet traffic and a greater number of nodes. However, as the results in figure 7 show, this has a much greater effect on the nix-vector protocol.

5.7 Discussion

In this chapter presented two new ns-3 modules. The first was a new BRITE module which allows ns-3 users to take advantage of the topology generation features of the BRITE topology generator. Also preseented was a new CUDA driven global routing protocol which demonstrated substantial speedup compared to the current global routing protocol in ns-3 when simulating large topologies, such as those created with BRITE. This new routing protocol also showed a significant speedup to Nix-Vector routing despite the fact it is calculating and storing many more routes. It provides a solid example of how GPUs can be used in combination with CPUs in order to achieve a faster overall simulation.

CHAPTER VI

PERFORMANCE OF CONSERVATIVE SYNCHRONIZATION METHODS FOR COMPLEX INTERCONNECTED CAMPUS NETWORKS IN NS-3

Large-scale network simulations are frequently needed to gain a full understanding of the environment being simulated. Due to limited resources on a single computing node and to reduce over all simulation run time, simulations may instead be distributed over multiple processing nodes. While this provides a powerful framework for utilizing a greater amount of computing resources, the nature of executing these simulations across multiple nodes demands that consideration by taken to ensure that the results of these distributed simulations match results that would have been produced by a single sequential simulator. This chapter examines the two conservative synchronization implementations available in the popular network simulator ns-3. ns-3 provides the option for the user to chose which of the algorithms to use for a particular simulation; however, little information is given on which algorithm is the better choice. We examine the performance of both distributed schedules in an effort to gauge specific features of an overall distributed network topology that warrant the use of one synchronization method over the other. While examining each implementation, improvements were made which resulted in better performance.

In the following sections we describe how both of these algorithms are implemented in ns-3. More general background on each algorithm can be found in the background chapter.

6.1 ns-3's granted time window implementation

The ns-3 granted time window algorithm and its accompanying MPI interface enlist a system for LPs to determine a granted time window for which they may safely process events in simulation timestamp order by achieving a global consensus among all LPs. During initializing the simulator beings by calculating the lookahead and initial granted time based on remote channel delays in the specified network topology. Distributed simulations in ns-3 currently only support remote point-to-point links between LPs, and as such, lookahead values are only derived from channels configured in this way. At the start of simulation, each LP examines the nodes within it to determine the shortest propagation delay between its nodes and any nodes in any adjacent LP. This value becomes the lookahead for that particular LP. To maintain adequate performance, LPs that do not possess inter-task links are given lookahead and initial granted values equivalent to the maximum lookahead from all LPs that do possess inter-task links in order to allow all tasks to advance in simulation time at similar rates. Once simulation execution begins, an LP will process the events in its event list until it reaches the end of its granted time window. When an LP has not more events within its granted time window, the synchronization phase will begin. The LBTS values will be gathered from all LPs using an `MPI_AllGather` call, effectively blocking individual LPs until all LPs reach this point in execution of the code. A check that the total count of completed received and completed transmitted messages are equal is executed to ensure that transient messages are not awaiting delivery by MPI. If transient messages exist in the system, each LP enterrs a phase where it stops sending traffic and completes all of its reads. Subsequently, another call to `MPI_AllGather` is made to verify that all transient messages are out of the system and gather the LBTS for the system. The minimum of these LBTS values will be summed with the LP's lookahead to determine the current granted simulation time within which an LP may process its events. Multiple calls to `MPI_AllGather` are sometimes necessary to ensure that the system can obtain an accurate snapshot and to obtain the correct LBTS values. Following this synchronization process, if the next event time lies within the granted window, the next event will be processed, and the simulation will continue.

6.2 ns-3's null message implementation

For ns-3.19, an additional distribution implementation was incorporated which provides a null message synchronization option following the Chandy/Misra/Bryant (CMB) algorithm [12] [9]. Unlike the granted time window algorithm, which is considered a synchronous algorithm due to its use of global communication, the CMB algorithm is asynchronous, only requiring peer-to-peer communication between neighboring LPs to preserve global causality.

Similarly to the granted time window implementation, an LP participating in the null message synchronization simulation begins by scanning the nodes within its topology to find external links to remote LPs. It groups all of the links to remote LPs into bundles according to which LP it connects. It then determines the minimum propagation delay value for each bundle. This value becomes the lookahead between the two LPs. Prior to the start of simulation, the LP queues a null message to the remote LP on each bundle with a timestamp of the lookahead value. The implementation also schedules a future null message to be sent to the remote LP when the sender's local simulation time reaches the minimum propagation delay to the remote LP. When this null message is sent, assuming no transfer of packets between the two LPs has occurred, the sender will continue to send null messages to the remote LP every lookahead period. When a packet is transferred between the two LPs, the next pending null message to be sent is canceled. Following the transfer of the packet, a new null message is scheduled to be sent to the remote LP after the next lookahead period.

In ns-3.19, the simulation will only stop under the null message synchronization for a specified simulation stop time. Otherwise, the simulation will continue to run, simply passing null messages between LPs, until the simulator reaches the ns-3 definition of infinity. This drawback can be a hindrance when using the ns-3 null message synchronization because a user must either specify a stop time or allow the simulation to reach infinity, a task which can be significantly time-consuming. This issue is not present in the ns-3 implementation of the granted time window algorithm as LPs will reach a consensus that the simulation is globally complete when all LPs have empty event queues.

6.3 Evaluation

The performance study demonstrated in this paper examines the runtime and memory usage of multiple instances of the well known DARPA campus network topology [47] shown in Figure 22. These are all connected in a ring topology as shown in Figure 23. Simulations are implemented using a modified version of nms-p2p-nix-distributed.cc from ns-3.19. A single campus network consists of 3 networks connected to a central network Net 0, which serves as the entry and exit point of data travelling between campus networks. Connecting each campus network at Router 0 in Net 0 involves a 2Gbps point-to-point link with a 200ms propagation delay. Within each campus network, routing nodes are connected to one another via 1Gbps point-to-point links with 5ms delays. The components labeled LAN in Figure 22 are actually 42-node local area networks (LAN) with each node individually connected to its respective router via 100Mbps point-to-point links with 1ms delays. In total, each campus network is composed of 538 nodes.

For these experiments, a modified version of the nms-p2p-nix-distributed.cc simulation script distributed with the ns-3 release has been used. The first modification implemented enabled the addition of multiple chords between members of the ring

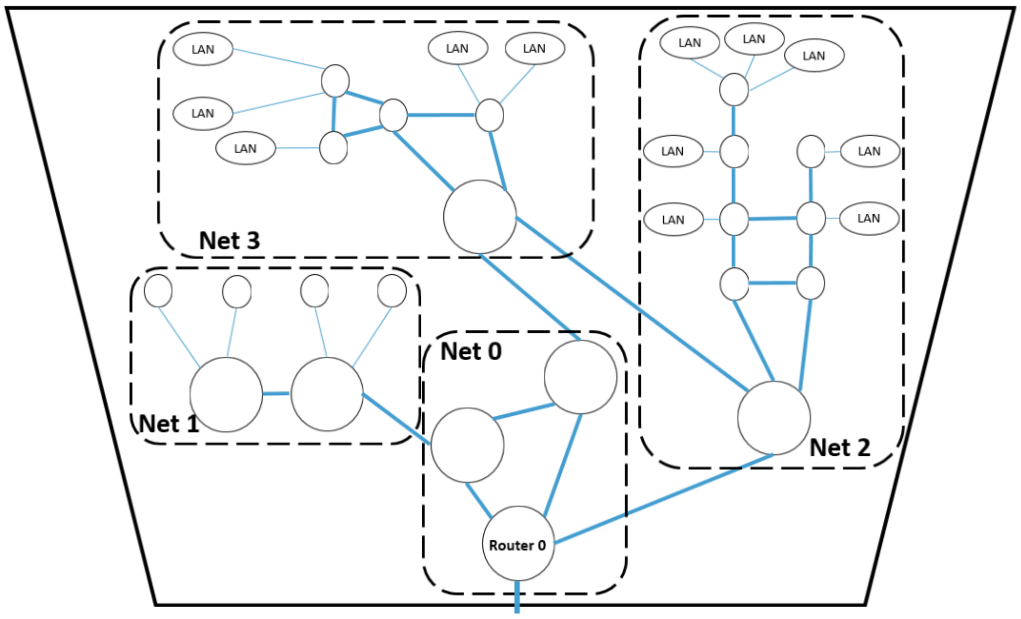


Figure 22: The DARPA campus network topology is composed of three networks which connect to Net 0, the entry and exit point for the campus network. Net 1 consists of 2 routers and four "server" nodes while Nets 2 and 3 connect a number of routers to multiple LAN networks consisting of 42 nodes each. Routers and server nodes are connected via 1Gbps point-to-point links with 5ms delays. LAN nodes connect to their associated routers with 100Mbps point-to-point links with 1ms delays. Data flowing into and out of the campus network travels a 2Gbps point-to-point channel with a 200 ms delay.

topology. A parameter named nConCon signifies the number of ”consecutive connections” that each campus network will employ to connect to multiple campus networks. In this way, nConCon equal to 1 represents the original topology and can be used as a basis for comparing results produced by increasing the nConCon value. An example showing multiple consecutive connections can be seen in Figure 23.

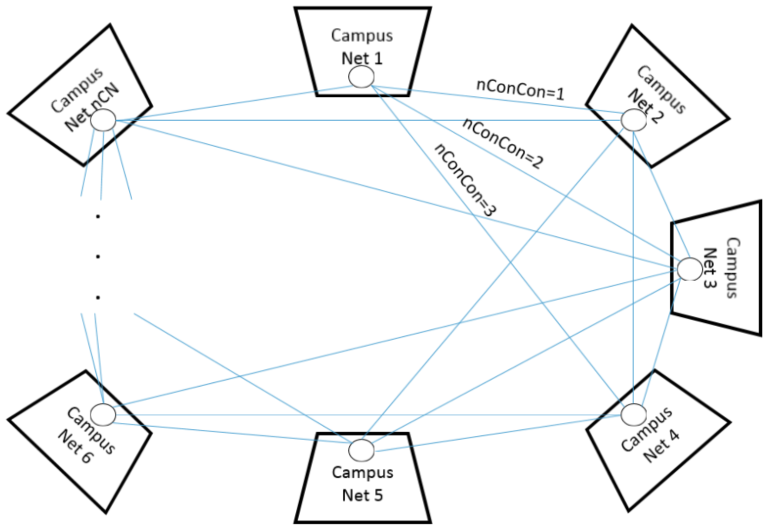


Figure 23: Multiple campus networks are connected in a ring topology with nConCon being the number of consecutive connections from one campus network to another campus network. For example, when nConCon=3, campus network 1 will connect to campus networks 2, 3, and 4, campus network 2 will connect to campus networks 3, 4, and 5, etc.

Each of the 42 nodes in each LAN in Nets 2 and 3 is given a UDP packet sink. For each packet sink added, an on-off application is installed on a randomly selected server node in Net 1. Each server node sends data to one of the sinks in the next campus on the ring. These applications oscillate between configurable periods of on time, during which data is sent, and off time, during which no data transmission occurs. Furthermore, for each additional consecutive connection for an LP, an additional on-off application is created on a server node in Net 1 and configured to send data to the campus network on the other end of the connection. The duty cycle is configured such that only one on-off application from each campus network is transmitting data out of that campus network at a time with each application being granted an equal amount of time to transmit. This setup ensures that the traffic in and out of each campus network effectively remains the same while externally presenting the distributed simulation implementations and the distributed systems on which they execute with new choices and challenges concerning their synchronization.

One other parameter is considered in this experiment which enables control of how much data is being sent remotely for each LP. When creating each on-off application, a uniformly random number is generated. If the random number is less than the specified remote percentage, the application will send data to a remote campus network. If the random number is greater than the remote percentage, the application will send data to one of its local LANs.

For routing traffic between the nodes ns-3's nix-vector routing protocol was used. nix-vector [65] routing does an on demand BFS to find the shortest path from source to sink. It then stores the steering information to guide the packet along that path within the packet itself. The node that generates the nix-vector also caches it so future transfers are done without the need for another BFS.

All of the experiments performed for this work were initially run on a machine with dual hex-core Intel Xeon E5-2620s running at 2.0 GHz. The machine was configured with 64 GBs of RAM. The experiments were all run using ns-3.19 with optimized build settings.

For the first experiment, a topology of 24 campus networks was created, for a total of 12,912 ns-3 nodes. Each on-off application was configured to send .5 MB to its chosen packet sink. For this experiment all on-off applications were configured to send data to their local LAN. In this configuration the simulation is embarrassingly parallel as there is no cross LP traffic. By performing this experiment, the cost of synchronization among the LPs can be measured. The consecutive connection parameter was varied from 1 to 23. The corresponding run time for each configuration is shown in Figure 24. As expected for both synchronization types, as the number of consecutive connections is increased, the run time is increased due to the added number of on-off applications and packet traffic. However, the results show that the granted time window application scales better as the connectivity of the topology increases. This trend is expected because the number of null messages that the CMB algorithm needs to send increases with the connectivity of the graph. For the granted time window algorithm, the number of nodes participating in the MPI_Allgather is the same for each experiment. It should be pointed out that this type of situation is ideal for the granted time window algorithm. Due to the lack of transient messages, during synchronization it only needs to perform one MPI_Allgather, a rather expensive operation, in order to obtain an accurate snapshot for LBTS calculation.

For the second experiment, a topology of 8 campus networks was created, for a total of 4,304 ns-3 nodes. The number of consecutive connections was set to one making a standard ring topology. Each on-off application was configured to send a maximum of 50 kilobytes. In this experiment, the percentage of remote traffic each LP sends was varied. For the results presented here, each configuration was run 10 times, and the average result was recorded. The results are shown in Figure 25. As shown in the graph, as the percentage of remote data increases, the performance of the granted time window decreases. Part of this decrease is likely due to the increase in frequency of transient messages during the synchronization phase causing the need for a second MPI_Allgather. The run times for CMB simulations do increase, as expected since the number of packets that need to be serialized and sent is increasing; however, the increase is much less noticeable. One benefit for the CMB algorithm in this experiment is that as the number of packets being sent to remote LPs increases, the number of null messages it needs to send to those LPs decreases.

Following the examination of the results of the second experiment, the significant increases in run time for the granted time window experiments were deemed

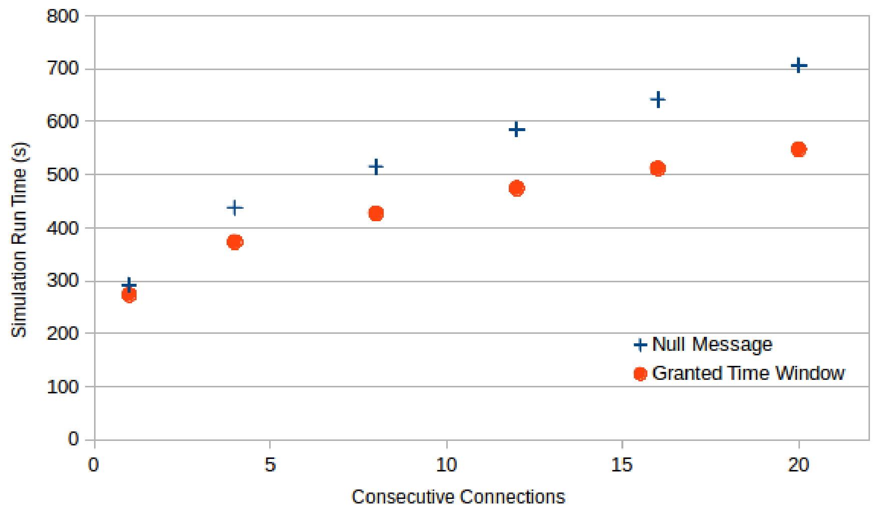


Figure 24: Simulation run time in seconds as a function of the number of consecutive connections. The total number of campus networks is 24 with each campus network residing on its own LP.

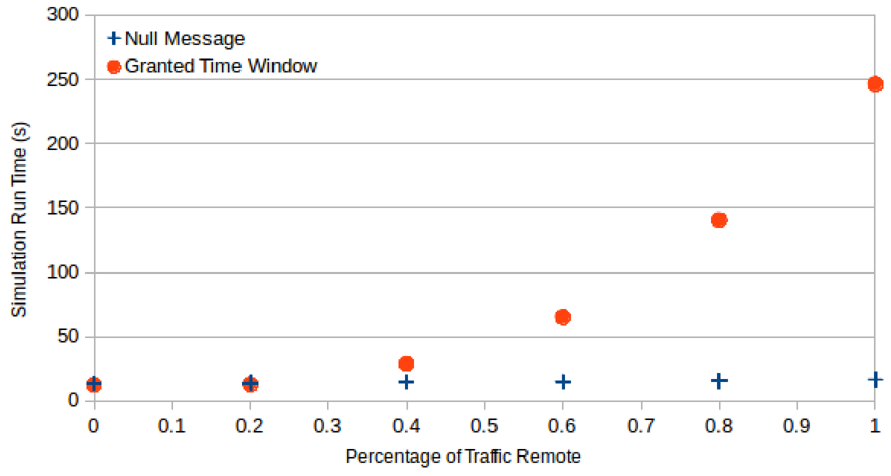


Figure 25: Simulation run time in seconds as a function of the percentage of remote traffic. The total number of campus networks is 8 with each campus network residing on its own LP and only one consecutive connection between campus networks (nConCon=1 from Figure 23).

unexpected; therefore, the experiment was repeated using a dedicated computing cluster rather than the original computing resource that employed shared memory. This computing cluster, managed by the Georgia Tech Partnership for an Advanced Computing Environment (PACE), provides dedicated cores as individual LPs. These cores are gathered for MPI job execution from a collective of six-core 2.4 GHz AMD Opteron 8431 processors. Ten time trials for each percentage of remote traffic were executed for both the null message and granted time window algorithms. As shown in Figure 26, a different picture is revealed as compared to that shown in Figure 25. The run times for the granted time window simulations were greatly improved.

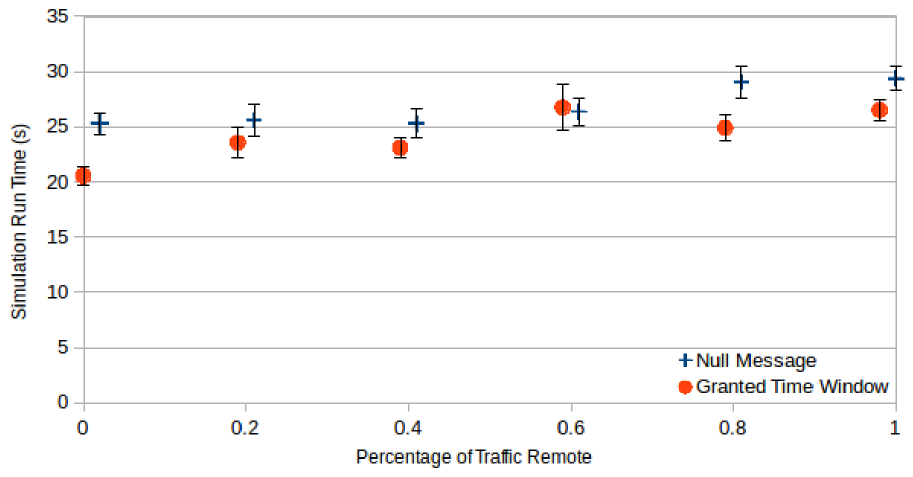


Figure 26: Simulation run time in seconds as a function of the percentage of remote traffic. This experiment utilized a cluster of computing resources rather than a shared memory resource. The total number of campus networks is 8 with each campus network residing on its own LP and only one consecutive connection between campus networks (nConCon=1 from Figure 23). Vertical bars represent the standard error of the sample mean.

The main difference between the two experiments is how MPI is utilized. For a many-core machine, as was used in the experiment depicted in Figure 25, MPI uses shared memory to pass data between the LPs. For the second cluster experiment, data sent between the LPs is passed across the network. The data obtained shows that the granted time window algorithm is performing poorly only when MPI is using shared memory. The shared memory experiment was repeated on another machine to verify that faulty operation was not inhibiting the first machine; similar results were obtained. All of these experiments were performed using OpenMPI 1.4.3.

In ns-3, the granted time window and null message algorithms check for received MPI messages at different frequencies. The null message algorithm checks for MPI received messages every time it processes an event. The granted time window implementation only checks for incoming messages once it determines it has no more safe events to process. We hypothesized that the longer run-times on the shared memory machine were due to excessive buildup of unprocessed MPI messages. For the next experiment, the granted time window algorithm was changed to check for new messages after processing each event. The same experiment as the last two was performed on the many-core shared memory machine. The time trials were again repeated 10 times across the percentages of remote traffic to better gauge the confidence of the results. The results are shown in Figure 27.

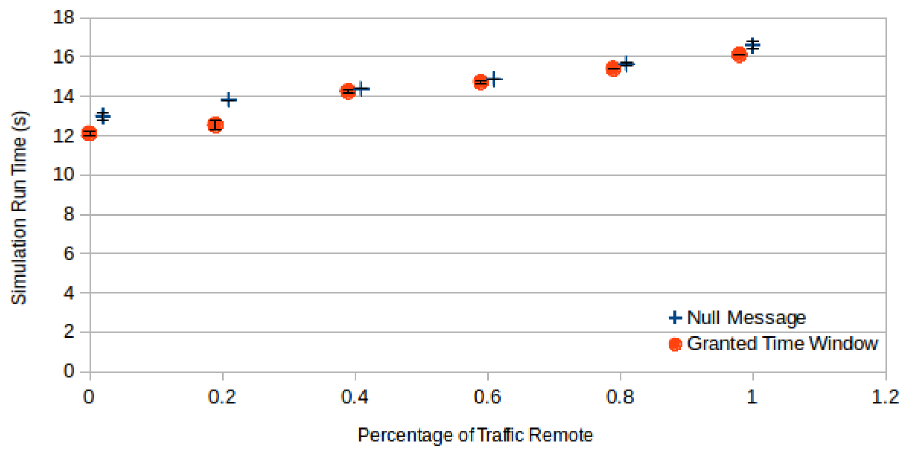


Figure 27: Simulation run time in seconds as a function of the percentage of remote traffic. This experiment utilized a shared memory resource but implemented a fix that checked for received MPI messages more frequently. The total number of campus networks is 8 with each campus network residing on its own LP and only one consecutive connection between campus networks (nConCon=1 from Figure 23). Vertical bars represent the standard error of the sample mean.

As shown by Figure 27, the results of the granted time window algorithm on a many-core shared memory computer were greatly improved by increasing the frequency with which each LP checks for received messages. These results support the hypothesis that the delay was due to excessive buildup of MPI messages. As Figure 27 shows, after the modification, the performance for both synchronization algorithms is effectively identical for this configuration. The granted time window algorithm experienced a 15.6x speedup compared to the original run-times measured on the simulations with exclusively remote traffic.

The next experiment performed examined whether increasing the frequency with which the granted time window algorithm checked for incoming messages had any negative consequences on a larger network topology run on a computer cluster. In this experiment 32 LPs were created for a total of 17,216 nodes. Each on-off application was configured to send 50 kilobytes to its selected target. All targets were in remote LPs. The number of consecutive connections was varied from 5 to 30. Again each configuration was run 10 times. The results are shown in Figure 28. The results indicate for the configurations that were tested, no performance penalty was incurred by increasing the polling frequency for new messages. In fact for all of the connection configurations, a small performance increase in the modified granted time window implementation was observed compared to the original implementation.

Upon further inspection of the null message implementation it was determined the number of null messages sent between the LPs could be reduced and the lookahead could be dynamically increased by examining other characteristics rather than just speed of light delay between the LPs. The point-to-point channels in ns-3 are FIFO; there, in the simple case where one channel connects two LPs, once an LP schedules a packet receive event from its neighbor, it doesn't need any more null messages from that neighbor until after it has executed the receive event. Another way to look at this is, if the neighboring LP always has at least one packet in its output queue destined

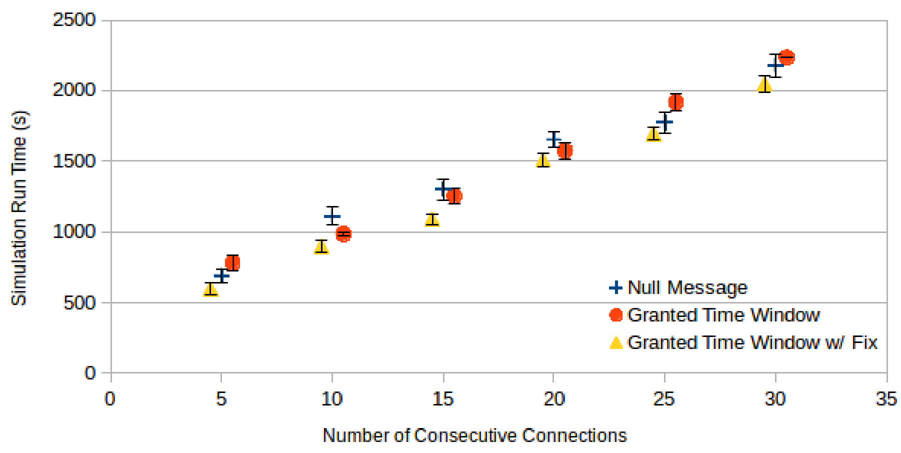


Figure 28: Simulation run time in seconds as a function of the number of consecutive connections. This experiment utilized a cluster of computing resources rather than a shared memory resource. The total number of campus networks is 32 with each campus network residing on its own LP. Vertical bars represent the standard error of the sample mean.

for the LP, the neighboring LP shouldn't need to send any null messages because it's constantly scheduling future packet receptions.

The original null message implementation in ns-3 did reschedule pending null message transmissions when sending a packet. However it rescheduled them to the sum of `currentSimulationTime` and `speedOfLightDelay`. For example, if the delay between the two LPs is 1 ms, but due to the size of the packet or the bandwidth of the device, the packet won't be fully received, last bit arrival time, for 40 ms, the implementation will send 39 unnecessary null messages.

Table 3: Original Null Message Implementation

LPs	Total Messages	Run Time (s)
8	239,104	0.172
16	478,208	0.175
32	956,416	0.175
64	1,912,832	0.184
128	3,825,664	0.185

Table 3 shows total number of null messages sent during the simulation and the wall clock runtime for a fairly simple example with a variable number of LPs. Each LP contains one node and all of the nodes are connected in a ring. Each LP sends UDP data to its neighbors on each side. Therefore all of the queues connecting LPs will always have at least one packet, assuming the LP's applications generate data faster than the rate the devices can transfer. For example, the speed of light delay between the LPs is 2 ms, the net devices send at a rate of 10kbps and the applications generate data at a rate of 1Mbps. 30 seconds of simulation time were simulated and the experiments were run on the Cab [35] computing cluster at Lawrence Livermore National Laboratory.

Table 4: Modified Null Message Implementation

LPs	Total Messages	Run Time (s)
8	32	0.097
16	64	0.101
32	128	0.098
64	256	0.100
128	512	0.102

The null message implementation was then modified to use the maximum of the channel delay and the complete packet transfer time at the time to schedule the next null message after a packet transfer. Due to the speed that the applications are sending traffic there will be a 2 ms delay before the first packets arrive in the queues. Afterward the queues on the net devices will always contain at least one packet and no more null messages should be needed. The results of this experiment are shown in table 4. Each LP in the simulation sends a null message at time 0 and at time 2 due to the rate at which application is producing data. If the application rate is increased to 5Mbps, the total number of null messages for each run is zero.

6.4 Discussion

This chapter examined and compared the two conservative time-synchronization algorithms available in ns-3.19. We performed experiments varying network connectivity and remote traffic percentage. The most interesting result was how poorly the granted time algorithm performed on the many-core machine using OpenMPI 1.4.3. Given these results, ns-3 users using this type of machine configuration would be best served by using the new null message algorithm or modifying the granted time window algorithm as we did.

The null message algorithm possesses drawbacks as well. As mentioned earlier, the algorithm is dependent on the simulation user to set an appropriate stop time. This requirement does not exist for the granted time window algorithm since it automatically exits once the simulation ends. In simulations with a fixed amount of data needing to be sent, overestimating the stop time could result in the simulator spending an excessive amount of time just passing null messages and not doing any actual simulation. Forgetting to set a stop time will result in a simulation that will run until the time variable overflows.

The changes we made to the two implementations provided significant speedup for certain scenarios. For the granted time window implementation, execution time was greatly reduced by more frequent polling by MPI for received messages. This was especially the case on the many-core machines. The change did not cause any other noticeable performance degradation. The change to the null message implementation greatly reduced the number of excess null messages that needed to be sent between LPs that frequently sending packets. A further improvement for this will be discussed in the future work section.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

This dissertation studied techniques to improve the performance of large-scale discrete-event simulations. In this final chapter, the contributions of this work are summarized and possible future work is discussed.

7.1 Contributions

Chapter 3 proposed a new approach to zero-copy message passing specifically designed for simulators running in a multi-core environment. We demonstrated the effectiveness of this approach by implementing it in two separate discrete-event simulators. The approach was compared to another zero-copy implementation made with Boost.Interprocess and also the standard approach of message serialization and bulk copy. Our approach was extremely effective compared to the other two. While the Boost.Interprocess design is zero-copy the architecture of their memory allocator has significant consequences on its ability to scale to a larger number of logical processes. While the serialization and bulk copy approach may be effective for small items with a limited amount of referential data, our approach provided consistent results for all message types including large messages with containing numerous references.

Chapter 4 demonstrated how conservative time-synchronization techniques can be used to run large-scale discrete-event simulations entirely on a graphical processing unit developed and marketed for three dimensional gaming. We examined multiple approaches for handling the future event list in the simulation and demonstrated examples where each could be effective. This work clearly shows there is a large amount of potential for graphical processing unit-based discrete-event simulators. The future work section will discuss possible future improvements.

Chapter 5 provided an example of how CPU-based discrete-event simulators can effectively utilize graphical processing units to process large parallel workloads. The work resulted in two new modules for the open-source ns-3 project. The first allows ns-3 users to easily create highly customizable large-scale network topologies that closely resemble what is typically found on the internet. The second provides a way to quickly perform static route computation for all of the nodes in the network. Our CUDA based global routing module was able to outperform the included global routing module by over three orders of magnitude for large topologies and was found to be faster than the single-source, single-destination on-demand nix-vector implementation. These additions make it much easier and less time-consuming for network researchers to perform large-scale network simulations using ns-3.

Chapter 6 compared the conservative time-synchronization implementations provided with the ns-3 network simulator. ns-3's documentation currently provides little to no instruction on when to chose one over the other. We examined both using a frequently used campus-based topology building block. For the work, we performed experiments on a large variety of topology configurations varying a large number of network parameters. While more research needs to be done provide an absolute decision criteria, we provided numerous insights to help users decide on which implementation to use for their large-scale network simulations. Furthermore, we developed improvements to both implementations that are capable of providing significant speedups under certain conditions without any noticeable performance impact in general performance.

7.2 Future work

There is a significant amount of future work that can be done on the development of CUDA based discrete-event simulators. With each new release of the CUDA API, it becomes increasingly easier to harness the full capabilities of the device. Furthermore, unlike CPU architecture which has become somewhat stagnant due to die size constants, lack of effective thermal management, and lack of any real competition among manufactures, GPU architectural advances are occurring at a steady pace. There are frequent product releases which are constantly pushing the envelope. The GPU cards are frequently being equipped with more cores that are more powerful than the previous top-of-the-line design. With the increasing availability of inexpensive hi-resolution 4K displays and predicted future advancements in display technology, this is a trend that is likely to continue.

A large amount of research could also be done on how to effectively utilize multi-GPU environments for discrete-event simulation. Newer MPI implementations including OpenMPI and MVAPICH2 have a CUDA-aware functionality [13]. Previously it was necessary to copy data from GPU device memory to CPU host memory in order to send it via MPI to a remote machine. Once on the other machine, it would then need to be transferred from CPU host memory to GPU device memory in order to be processed by the remote GPU. With a CUDA-aware MPI implementation, GPU memory can be directly passed to MPI without staging it in host memory [33]. This provides performance benefits and makes it easier for the developer. However, before a multi-GPU setup can be used in discrete-event simulation, algorithms need to be developed for remote event passing that do not result in severe thread divergence or inefficient memory usage.

The improvement to ns-3's null message implementation reduced the number of MPI messages, due to the decrease in the number of null messages being sent, and allowed the LP to provide a larger dynamic lookahead to its neighbor. However the improvement only considers the current packet on the wire being transferred. Since the ns-3 queues are FIFO, and since there is currently no way for a packet to not arrive at a remote LP once it is placed into a queue leading to that LP, the entire queue could be analyzed to provide an even larger dynamic lookahead. This could also possibly be combined with bulk sending of multiple packets to gain even more performance by further reducing MPI traffic.

REFERENCES

[1] Anderson, M., Catanzaro, B., Chong, J., Gonina, E., Keutzer, K., Lai, C.-Y., Murphy, M., Sheffield, D., Su, B.-Y., and Sundaram, N., "Considerations when evaluating microprocessor platforms," HotPar'11, (Berkeley, CA, USA), pp. 1–1, 2011.

[2] Barabasi, A.-L. and Albert, R., "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[3] Barnes, Jr., P. D., Carothers, C. D., Jefferson, D. R., and LaPre, J. M., "Warp speed: Executing time warp on 1,966,080 cores," in *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '13, (New York, NY, USA), pp. 327–336, ACM, 2013.

[4] Bauer, H. and Sporrer, C., "Reducing rollback overhead in time-warp based distributed simulation with optimized incremental state saving," in *Simulation Symposium, 1993. Proceedings., 26th Annual*, pp. 12 –20, 1993.

[5] Bilel, B. R., Navid, N., and Bouksiaa, M. S. M., "Hybrid cpu-gpu distributed framework for large scale mobile networks simulation," in *Proceedings of the 2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '12, (Washington, DC, USA), IEEE Computer Society, 2012.

[6] Board, O. A. R., "The openmp api specification for parallel programming." http://www.openmp.org.

[7] Brooks, III, E. D., "The butterfly barrier," *Int. J. Parallel Program.*, vol. 15, pp. 295–307, Oct. 1986.

[8] Brunie, N., Collange, S., and Diamos, G., "Simultaneous branch and warp interweaving for sustained gpu performance," *SIGARCH Comput. Archit. News*, vol. 40, pp. 49–60, June 2012.

[9] Bryant, R. E., "Simulation of packet communication architecture computer systems," tech. rep., Cambridge, MA, USA, 1977.

[10] Calvert, K., Doar, M., and Zegura, E., "Modeling internet topology," *Communications Magazine, IEEE*, vol. 35, pp. 160–163, June 1997.

[11] Carothers, C., Bauer, D., and Pearce, S., "Ross: a high-performance, low memory, modular time warp system," in *Parallel and Distributed Simulation, 2000. PADS 2000. Proceedings. Fourteenth Workshop on*, pp. 53–60, 2000.

[12] Chandy, K. and Misra, J., "Distributed simulation: A case study in design and verification of distributed programs," *Software Engineering, IEEE Transactions on*, vol. SE-5, pp. 440–452, Sept 1979.

[13] Cheng, J., Grossman, M., and McKercher, T., *Professional CUDA C Programming.* Wrox, 2014.

[14] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E., *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd ed., 2001.

[15] Coutinho, B., Sampaio, D., Pereira, F., and Meira, W., "Divergence analysis and optimizations," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 320–329, Oct 2011.

[16] Das, S., Fujimoto, R., Panesar, K., Allison, D., and Hybinette, M., "Gtw: a time warp system for shared memory multiprocessors," in *Simulation Conference Proceedings, 1994. Winter*, pp. 1332–1339, Dec 1994.

[17] Distributed Computing Group, *Singalo - Simulator for Network Algorithms.*

[18] Erds, P. and Rnyi, A., "On the evolution of random graphs," in *PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES*, pp. 17–61, 1960.

[19] Flynn, M., "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. C-21, pp. 948–960, Sept 1972.

[20] Forum, M. P., "Mpi: A message-passing interface standard," tech. rep., Knoxville, TN, USA, 1994.

[21] Frank, M., "The r programming language and compiler @ONLINE," 1997.

[22] Fujimoto, R. M., "Performance measurements of distributed simulation strategies," Tech. Rep. UUCS-87-026, University of Utah, 1987.

[23] Fujimoto, R. M., "Performance of time warp under synthetic workloads," tech. rep., 1990.

[24] Fujimoto, R. M., *Parallel and Distributed Simulation Systems.* Wiley, 2000.

[25] Fujimoto, R. M., Tsai, J.-J., and Gopalakrishnan, G. C., "Design and evaluation of the rollback chip: Special purpose hardware for time warp," *IEEE Trans. Comput.*, vol. 41, pp. 68–82, Jan. 1992.

[26] Gaztanaga, I., *Boost.Interprocess C++ Library.*

[27] Han, T. and Abdelrahman, T., "Reducing branch divergence in gpu programs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, p. 3, ACM, 2011.

[28] Harish, P. and Narayanan, P. J., "Accelerating large graph algorithms on the gpu using cuda," in *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, (Berlin, Heidelberg), pp. 197–208, Springer-Verlag, 2007.

[29] Intel, "Intel threading building blocks." https://www.threadingbuildingblocks.org.

[30] Ivey, J., Swenson, B., and Riley, G., "Phold performance of conservative synchronization methods for distributed simulation in ns-3," in *Proceedings of the 2015 Workshop on ns-3*, 2015.

[31] Iyigun, M., "DaSSFNet: An Extension to DaSSF for High-Performance Network Modeling," Tech. Rep. TR2001-405, Dartmouth College, Computer Science, Hanover, NH, June 2001.

[32] Jefferson, D. R., "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404–425, 1985.

[33] Kraus, J., "An introduction to cuda-aware mpi." http://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi, 2013.

[34] Kunz, G., Schemmel, D., Gross, J., and Wehrle, K., "Multi-level parallelism for time- and cost-efficient parallel discrete event simulation on gpus," in *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, PADS '12, (Washington, DC, USA), pp. 23–32, IEEE Computer Society, 2012.

[35] Laboratory, L. L. N., "Cab computing cluster." http://computation.llnl.gov/computers/cab.

[36] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P., "Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu," *SIGARCH Comput. Archit. News*, vol. 38, pp. 451–460, June 2010.

[37] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P., "Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 451–460, ACM, 2010.

[38] Lin, Y.-B. and Lazowska, E. D., "Determining the global virtual time in a distributed simulation.," in *ICPP (3)* (Yew, P.-C., ed.), pp. 201–209, Pennsylvania State University Press, 1990.

[39] Liu, J. and Nicol, D. M., "Lookahead revisited in wireless network simulations," in *Proceedings of the Sixteenth Workshop on Parallel and Distributed Simulation*, PADS '02, (Washington, DC, USA), pp. 79–88, IEEE Computer Society, 2002.

[40] Lutz, C., "Janus: a time-reversible language." *Letter to R. Landauer.* `http://www.tetsuo.jp/ref/janus.pdf`, 1986.

[41] Lynch, E. W. and Riley, G. F., "Hardware supported time synchronization in multi-core architectures," in *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS '09, (Washington, DC, USA), pp. 88–94, IEEE Computer Society, 2009.

[42] Mattern, F., "Efficient algorithms for distributed snapshots and global virtual time approximation," *J. Parallel Distrib. Comput.*, vol. 18, pp. 423–434, Aug. 1993.

[43] Medina, A., Lakhina, A., Matta, I., and Byers, J., "Brite: an approach to universal topology generation," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on*, pp. 346–353, 2001.

[44] Merrill, D. and Grimshaw, A., "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing," *Parallel Processing Letters*, vol. 21, no. 02, pp. 245–272, 2011.

[45] Misra, J., "Distributed discrete-event simulation," *ACM Comput. Surv.*, vol. 18, pp. 39–65, Mar. 1986.

[46] Nandapalan, N., Brent, R. P., Murray, L. M., and Rendell, A. P., "High-performance pseudo-random number generation on graphics processing units," in *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Volume Part I*, PPAM'11, (Berlin, Heidelberg), pp. 609–618, Springer-Verlag, 2012.

[47] Nicol D. M., "Standard baseline nms challenge topology." Accessed April 7, 2014. `http://ssfnet.org/Exchange/gallery/baseline/`, 2002.

[48] "The Network Simulator NS-2." `http://www.isi.edu/nsnam/ns/`.

[49] ns 3 Collaboration, "The ns-3 network simulator." `http://www.nsnam.org/`, 2011.

[50] NVIDIA, "curand, the nvidia cuda random number generation library." https://developer.nvidia.com/curand.

[51] NVIDIA, "Nvidia kepler gk110 architecture whitepaper"." http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf.

[52] NVIDIA, "Parallel thread execution isa version 4.2." http://docs.nvidia.com/cuda/parallel-thread-execution.

[53] NVIDIA, "Thrust." https://developer.nvidia.com/thrust.

[54] NVIDIA, "Cuda technology." http://www.nvidia.com/CUDA, 2007.

[55] NVIDIA, "Cuda c best practices guide." http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/, March 2015.

[56] NVIDIA, "Cuda c programming guide." http://docs.nvidia.com/cuda/cuda-c-programming-guide, March 2015.

[57] NVIDIA and Merrill, D., "Cub cuda unbound." http://nvlabs.github.io/cub.

[58] Park, H. and Fishwick, P. A., "A gpu-based application framework supporting fast discrete-event simulation," *Simulation*, vol. 86, pp. 613–628, Oct. 2010.

[59] Perumalla, K. S., "Parallel and distributed simulation: Traditional techniques and recent advances," in *Proceedings of the 38th Conference on Winter Simulation*, WSC '06, pp. 84–95, Winter Simulation Conference, 2006.

[60] Perumalla, K. S., *Introduction to Reversible Computing.* CRC Press, 2014.

[61] Perumalla, K. S., Aaby, B. G., Yoginath, S. B., and Seal, S. K., "Gpu-based real-time execution of vehicular mobility models in large-scale road network scenarios," in *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS '09, (Washington, DC, USA), pp. 95–103, IEEE Computer Society, 2009.

[62] Perumalla, K. S. and Fujimoto, R. M., "Source-code transformations for efficient reversibility," Tech. Rep. GIT-CC-99-21, Georgia Institute of Technology, 1999.

[63] Perumalla, K., "Discrete-event execution alternatives on general purpose graphical processing units (gpgpus)," in *Principles of Advanced and Distributed Simulation, 2006. PADS 2006. 20th Workshop on*, pp. 74–81, 2006.

[64] Riley, G. F., "The georgia tech network simulator," in *Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*, MoMeTools '03, (New York, NY, USA), pp. 5–12, ACM, 2003.

[65] Riley, G., Ammar, M., and Zegura, E., "Efficient routing using nix-vectors," in *High Performance Switching and Routing, 2001 IEEE Workshop on*, pp. 390–395, 2001.

[66] Rizvi, S., Elleithy, K., and Riasat, A., "Trees and butterflies barriers in distributed simulation system: A better approach to improve latency and the processor idle time," in *Information and Emerging Technologies, 2007. ICIET 2007. International Conference on*, pp. 1–6, July 2007.

[67] Romdhanne, B. and Navid, B. N., "Cunetsim: A new simulation framework for large scale mobile networks," in *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, SIMUTOOLS '12, (ICST, Brussels, Belgium, Belgium), pp. 217–219, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2012.

[68] Sartori, J. and Kumar, R., "Branch and data herding: Reducing control and memory divergence for error-tolerant gpu applications," *Multimedia, IEEE Transactions on*, vol. 15, pp. 279–290, Feb 2013.

[69] Swenson, B. and Riley, G., "A new approach to zero-copy message passing with reversible memory allocation in multi-core architectures," in *Principles of Advanced and Distributed Simulation (PADS), 2012 ACM/IEEE/SCS 26th Workshop on*, pp. 44–52, July 2012.

[70] Swenson, B. P. and Riley, G. F., "Simulating large topologies in ns-3 using brite and cuda driven global routing," in *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, SimuTools '13, pp. 159–166, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013.

[71] Varga, A. and Hornig, R., "An overview of the omnet++ simulation environment," in *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, (ICST, Brussels, Belgium, Belgium), pp. 60:1–60:10, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[72] Vulov, G., Hou, C., Vuduc, R., Fujimoto, R., Quinlan, D., and Jefferson, D., "The backstroke framework for source level reverse computation applied to parallel discrete event simulation," in *Simulation Conference (WSC), Proceedings of the 2011 Winter*, pp. 2960–2974, Dec 2011.

[73] Waxman, B., "Routing of multipoint connections," *Selected Areas in Communications, IEEE Journal on*, vol. 6, pp. 1617–1622, Dec 1988.

[74] West, D. and Panesar, K., "Automatic incremental state saving," in *Parallel and Distributed Simulation, 1996. Pads 96. Proceedings. Tenth Workshop on*, pp. 78 –85, 1996.

[75] Zhang, E. Z., Jiang, Y., Guo, Z., and Shen, X., "Streamlining gpu applications on the fly: Thread divergence elimination through runtime thread-data

remapping," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, (New York, NY, USA), pp. 115–126, ACM, 2010.

[76] Zhou, J., Ji, Z., Takai, M., and Bagrodia, R., "Maya: Integrating hybrid network modeling to the physical world," *ACM Trans. Model. Comput. Simul.*, vol. 14, pp. 149–169, Apr. 2004.