Proceedings of the 2019 Winter Simulation Conference
N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.

A MULTITHREADED SIMULATION EXECUTIVE IN SUPPORT OF DISCRETE EVENT SIMULATIONS

Brandon Waddell
James F. Leathrum, Jr

Old Dominion University
5115 Hampton Blvd
Norfolk, VA 23529, USA

ABSTRACT

Classical Parallel Discrete Event Simulation (PDES) increases simulation performance by distributing the computational load of a simulation application across multiple computing platforms. A burden is placed on the developer to partition the simulation. This work focuses instead on identifying available parallelism within the simulation executive, thus removing the burden from the developer. Event scheduling and event execution have natural parallelism with each other; future events may be scheduled with the simulation executive while the currently executing event continues. Execution and scheduling are dealt with as multiple threads, taking advantage of multicore architectures. This paper identifies the available parallelism between event scheduling and execution, highlights points of contention between the two, provides an algorithm to take advantage of the parallelism and presents timing results demonstrating the performance benefits.

1 INTRODUCTION

The DES architecture considered in this paper consists of a reusable simulation executive and a simulation application (Pidd 2004). The simulation executive is responsible for event list management while the simulation application is responsible for defining event functionality. A standard DES is developed to run serially which inherently has longer execution times as the system grows larger, generating more pending events, thus complicating event scheduling on the simulation executive. Historically distributed simulation is utilized to increase performance by partitioning the system across many processors, with a side benefit of partitioning the pending events across the processors. With the partitioning of the system, significant modifications to the simulation application are required, leaving developers with substantial development overhead (Fujimoto 2000).

An alternative approach to partitioning the system across multiple processors is to utilize multiple cores on a single processor by creating multiple threads of execution working concurrently. This approach can be used to lower the burden placed on the simulation developer. A multi-core processor is best utilized when there is a low number of available parallelism that is identified due to the limited number of cores on a processor. Low-levels of parallelism can be identified between event scheduling and event execution within the simulation executive which creates a suitable environment to utilize a multi-core processor. With the parallelism being identified and managed in the simulation executive, an increase in performance to a DES can be achieved without modifications to the application, thus providing a "free" performance gain, albeit minimal, without identifying available performance in the application. This does not preclude the benefits of further parallelizing the application.

The goal of this research is to overlap event scheduling and execution to hide the overhead of scheduling to the greatest level possible, ideally achieving O(1) event scheduling regardless of the event list

implementation. This is accomplished by identifying two threads, event execution and event scheduling, that will aid events to continue their execution while future events are undergoing scheduling. The execution thread cannot completely eliminate the burden of scheduling events since scheduling events is initiated within the simulation application. By separating the event list into two, the execution thread can schedule events into one, hopefully small, event list. The scheduling thread will be responsible for transferring events that the execution thread schedules into the other event list. If the scheduling thread can transfer events fast enough, the execution thread will schedule into a much smaller event list than a singular event list thus lowering the scheduling overhead incurred by the execution thread.

2 BACKGROUND

2.1 Discrete Event Simulation

Discrete event simulation is a system modeling technique that describes the behavior of the system as a series of state transitions at specific moments in time. These states are transitioned through a sequence of events that occur at specific moments in time (Chen 2015). Events are executed in timestamp order therefore a DES contains an event list that holds all future events in timestamp order. There is no physical delay between event executions but rather the simulation time will instantaneously change when an event is executed.

A standard approach to DES software development is to partition the simulation into simulation application software and a reusable simulation executive as illustrated in the architecture in Figure 1 (Pidd 2004). The simulation executive is responsible for selecting the next event to execute in nondecreasing timestamp order, advancing simulation time to that of the currently executing event, and the scheduling of future events provided by the simulation application. The simulation application is responsible for executing the events which involves changing state information and scheduling future events. The simulation executive uses the control loop in Figure 2 to manage event execution (Law 2004). Pidd (2004) provides the modified loop when handling conditional events in addition to scheduled events as found in the 3-phase worldview. The algorithm for scheduling future events by placing them in the event list in timestamp order is provided in Figure 2. The event list structure being used is a sorted linked list although alternative structures exist (Vaucher 1975) such as heaps, calendar queues (Brown 1988), lazy queues (Ronngren 1991) and ladder queues (Tang 2005). The algorithms in Figure 2 is presented as the foundation upon which parallelism is identified.

2.2 Parallel Discrete Event Simulation Using Multi-Threading

In a scalar, single core processor only a single thread of execution is realizable at a moment in time. With the introduction of a multi-core processor architecture, a multi-threaded environment can provide true parallelism. A multi-threaded environment involves creating multiple threads of execution in a single program that work concurrently. A Parallel Discrete Event Simulation (PDES) can be created if a DES is partitioned into multiple threads of execution. Each thread in a program shares a common program context, shared memory spaced, allowing each thread to access the same information. Taking advantage of the common program context reduces the burden placed on the developer when compared to distributed simulation. Parallelism between processes must be taken advantage of in order to gain a noticeable performance benefit, ideally balancing the load between threads.

Multithreading has previously been applied to DES at the application level (Beckmann et al. 2010; Kunz et al. 2011; Bryan et al. 2012; Wang et al. 2014). While much is done on providing support to the application developer, the work still focuses on breaking the execution of the application onto multiple threads. An example of providing multithreading in the software library or framework supporting the application can be found in (SOFA 2019), but the emphasis there is on the graphics. This research will focus on identifying parallelism within the simulation executive with the goal of providing the developer a performance benefit free of charge, under certain simulation executive workloads.

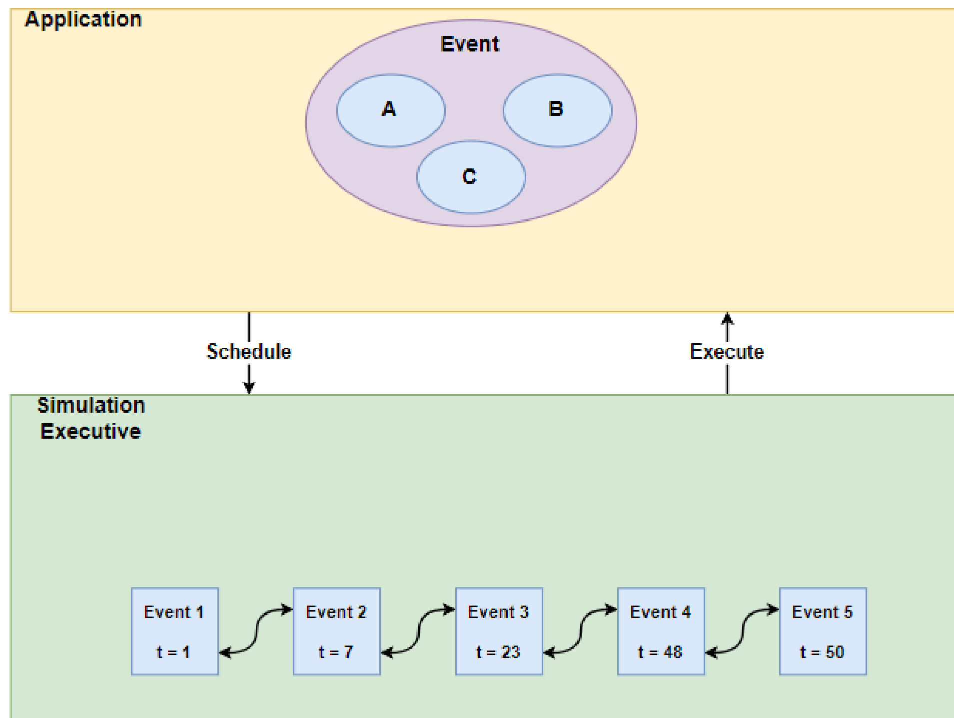

Figure 1: Application and Simulation Executive Relationship.

```
Simulation Execution
While Simulation is not done
    Get the next event
    Simulation Time = Event Time
    Execute the event
```

```
Event Scheduling
If (NewEvent.Time < First.Event.Time or List Empty)
      NewEvent = FirstEvent;
IteratorEvent = FirstEvent;
While (NewEvent.Time > IteratorEvent.Next.Time)
      IteratorEvent = IteratorEvent.Next;
IteratorEvent.Next = NewEvent;
```

Figure 2: Algorithms for Simulation Execution Algorithm and Event Scheduling.

3 MULTI-THREADED SIMULATION EXECUTIVE

3.1 Parallelism between Event Scheduling and Execution

Introducing parallelism in the simulation executive can increase the performance of a serial DES application without adding overhead to the application developer. Scheduling a future event, while initiated from the execution of an event, is independent from the continued execution of the event, suggesting that those actions can occur in parallel. Figure 3 on the left shows the relationship of executing an event selected by the simulation executive from the event list, and that event scheduling another event. On the right, the areas identified by dotted ovals represent portions identified for potential parallelism; scheduling events and executing events can be run in parallel when not in conflict for resources. Isolating the parallelism to the simulation executive allows for the simulation executive to remain reusable, with the same interface as the serial DES resulting in no intervention required by the simulation application developer.

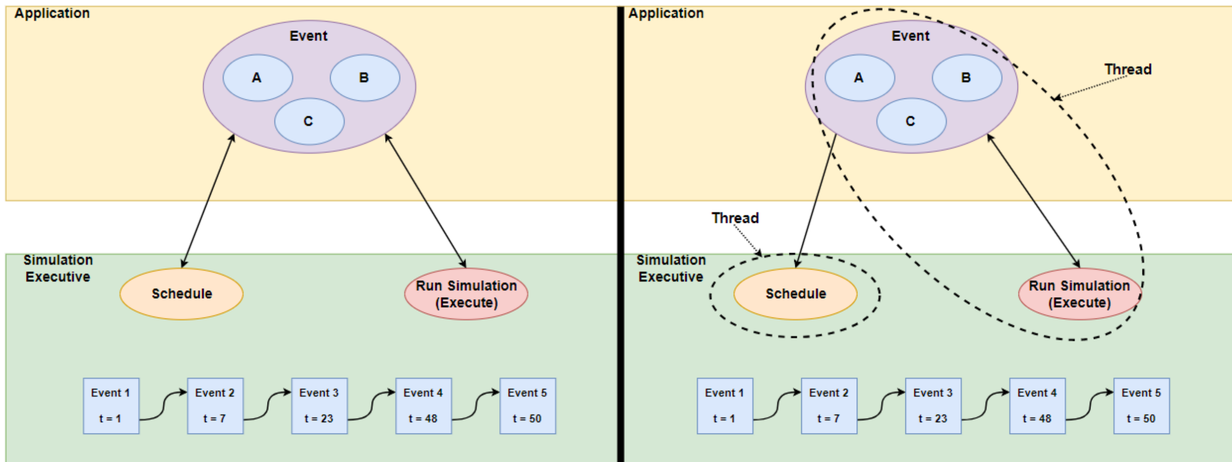

Figure 3: Application and Simulation Interactions: Serial (left) and Parallel (right).

A serial simulation executive will repetitively get the pending event with the smallest timestamp and then execute it. During the execution of an event, new events may be scheduled. This scheduling activity must finish before the execution of the current event will continue as shown on the left in Figure 4. Parallelism between event scheduling and execution allows for future events to be scheduled while the current event continues to be executed as shown on the right in Figure 4. With event scheduling and execution working in parallel, the same number of events can possibly be executed in a shorter amount of time when compared to its serial counterpart which can be deduced by comparing the left from the right in Figure 4.

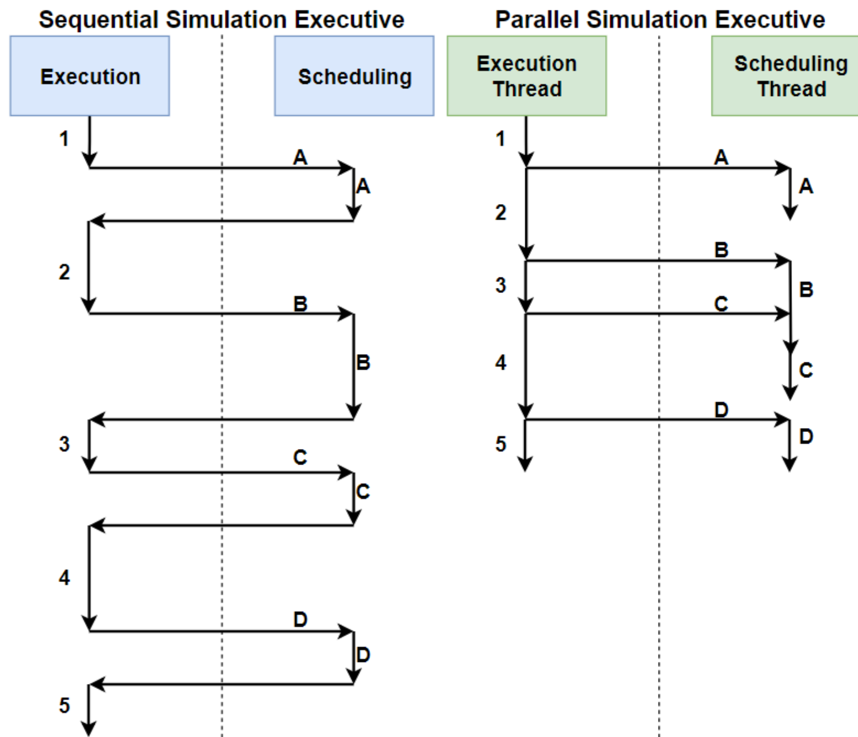

Figure 4: Serial Execution (left) vs. Parallel Execution (right).

3.2 Lessening the scheduling burden on the execution thread

A serial simulation executive forces an event to finish scheduling before continuing the execution of the current event. This means that only one event is ever being scheduled at a time, shown in Figure 4; even if an executing event were to schedule multiple events, one event would finish being scheduled before the next event would start its scheduling. When parallelizing the simulation executive there are two main concerns:

1) The execution thread cannot be fully rid of the responsibility of scheduling events. The scheduling of events occurs within the application; since the execution thread is executing events it inherently must minimally initiate the scheduling of events, providing the new event to schedule to the scheduling process.
2) The concurrency between execution and scheduling brings forth the potential that a new event wishes to be scheduled before the event currently being scheduled has finished. This is shown in Figure 4 where event B has not finished being scheduled before event C is ready to be scheduled.

To circumvent these issues, an additional event list is created, termed the scheduled event list (SEL); it holds all pending events that are to be scheduled in the main event list, now known as the execution event list (EEL), as illustrated in Figure 5. The execution thread will schedule all events into the SEL while pulling the lowest timestamp event from either event list, looking at the first event in each list to determine the smallest. This is done to avoid having execution wait until all events have been moved from the SEL to the EEL. The schedule thread's responsibility is to continually move the first event from the SEL into the EEL. On completion of scheduling an event, the scheduling thread checks the SEL to see if it has more events to schedule, else it goes to sleep, waiting for the next event to be scheduled by the execution thread on the SEL, thus waking up the scheduling thread. The goal is that the SEL will be considerably smaller than the event list in a sequential simulation, reducing the execution thread's time spent adding events into an event list. The scheduling thread attempts to transfer events from the SEL to the EEL as quickly as possible to keep the SEL as small as possible. A best case scenario is adding an event to the SEL by the execution thread will be O(1) because the schedule thread will have transferred all events from the SEL to the EEL instead of O(n) for a serial simulation executive with n pending events.

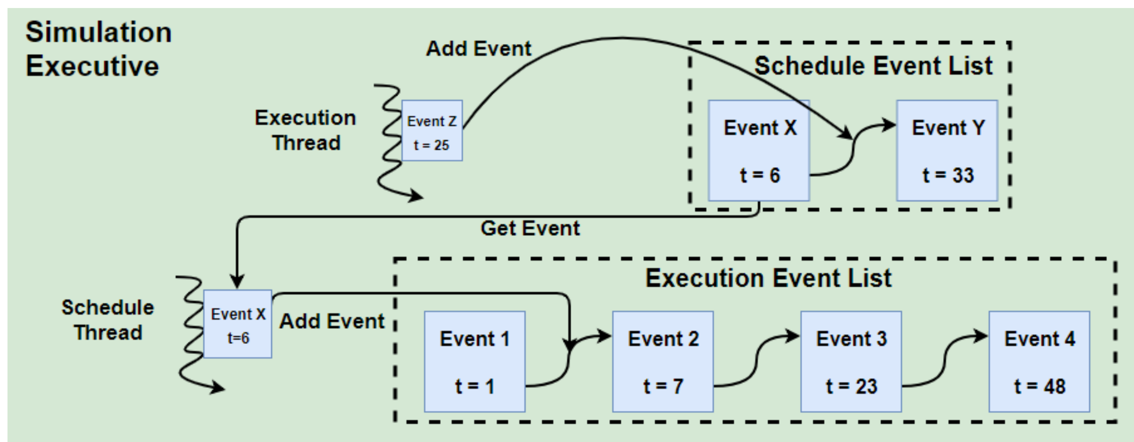

Figure 5: Schedule Event List.

3.3 Identification of Resource Contention

One concern in parallel processing is resource contention (Moseley et al. 2005; Dey et al. 2011). This occurs in multithreaded applications when multiple threads interact with the same piece of data at the same time;

multiple threads attempting to change the same data can cause major errors. The proposed approach has two threads: event scheduling and event execution that each need access to both event lists. The execution thread accesses the front of either list to get the next event, the one with the smallest timestamp, for executing and accesses the entire SEL when the application schedules a new event, searching for where to put the new event to maintain timestamp order in the SEL. The scheduling thread accesses the front of the SEL and the entire EEL when scheduling events from the SEL to the EEL, searching for where to put the event in the EEL in timestamp order. When a thread schedules an event into either list, it begins at the front of the list and works its way through the list until finding the appropriate location to insert the event, following the same algorithm utilized by the serial algorithm shown in Figure 2. The execution thread will never go beyond the first event in the EEL, similarly the schedule thread will never go beyond the first event in the SEL. Therefore, we can identify the resources in contention as the first event in both event lists as shown in Figure 6.

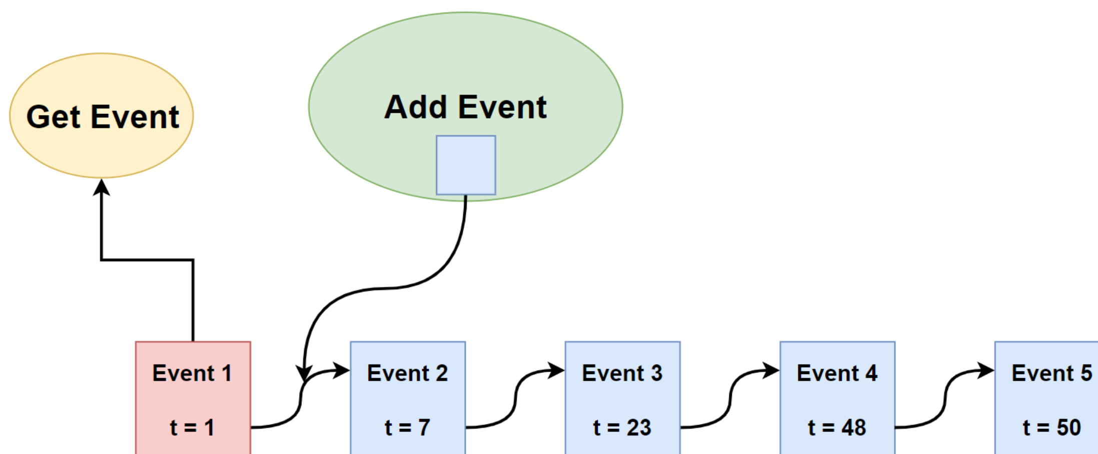

Figure 6: Resource Contention in the Event List.

3.4 Solution to Resource Contention

Mutex locking is used to solve the resource contention problem. A lock, is placed to inhibit the threads from accessing the head of either event list without permission; since the only contentious resource is at the start of the event lists, there are circumstances where a thread can obtain access to the lock while the other thread is adding an event further down the list. The lock begins unlocked and will remain in that state until one of the threads requests the lock in order to access the head of the list. Once the lock is obtained by a thread, it has full access to the head of both lists until it releases the lock. If a thread requests the lock while it is already locked, it will wait until the lock becomes available before proceeding. Both heads are conservatively locked together since the majority of actions involving the event lists start by accessing both heads.

3.4.1 Thread Algorithms

Each thread is responsible for releasing the lock as early as possible so as to not impede the progress of the other thread. There are two supporting subroutines that will request or release the lock: add event and schedule event. The add event subroutine is part of the event list's functionality and is used by both the execution and schedule threads to add an event to a specific event list. This routine will obtain the lock before starting the insertion process, once the lock is obtained it will find the appropriate place in the event list for the new event by iterating through the event list. The lock is released once the iterator passes the first event in the list since that is only identified contentious resource. The schedule event subroutine is

called by the application and thus used by the execution thread to add new events to the SEL; this subroutine will request the lock, then call the add event subroutine from the SEL. The two subroutines can be seen in Figure 7.

```
Schedule Event
Request Lock
Add Event to SEL
//Add Event will have released the lock
If (Scheduling Thread Asleep)
      Wake Scheduling Thread
```

```
Add Event
//Lock should have been requested before this call
If (Event List Empty)
      FirstEvent = NewEvent
      Release Lock
Else if (event goes at the head of the Event List)
      NewEvent.Next = StartEvent.Next
      FirstEvent = NewEvent
      Release Lock
Else if (event goes directly after first event)
      NewEvent.Next = StartEvent.Next
      StartEvent.Next = NewEvent
      Release Lock
Else
      IteratorEvent = FirstEvent.Next
      Release Lock
      While (IteratorEvent.Next != NULL)
            If (NewEvent.Time >= IteratorEvent.Time)
                  IteratorEvent = IteratorEvent.Next
      If (IteratorEvent.Next == NULL)
            IteratorEvent.Next = event
      else
            NewEvent.Next = IteratorEvent.Next
                  IteratorEvent =NewEvent
```

Figure 7: Algorithms for Schedule Event (Sim. Exec. Subroutine) and Add Event (Event List Subroutine).

The execution thread continually requests the lock, gets the smallest timestamped event from either event list, releases the lock, then executes the event. Upon executing the event a new event may need to be scheduled; the lock must then be requested again before adding the new event to the SEL. Once the event has been added, the schedule thread is notified that there is a new event in the list. The schedule thread sleeps and waits for a notification from the execution thread indicating that there are new event(s) in the SEL. The schedule thread continually requests the lock, gets the first event from the SEL, adds the event to the EEL until there are no more events in the SEL. Figure 8 shows the algorithms for each thread.

```
Execute Thread
Request Lock
While(Simulation is not done)
      Find min. timestamp (EEL, SEL)
      If(min. timestamp from EEL)
            Get event from EEL
      Else
            Get event from SEL
      Release Lock
      Simulation Time = Event Time
      Execute event
      Request Lock
Release Lock
Notify Sch. Thread that Sim. Done
```

```
Schedule Thread
While(Simulation is not done)
      Wait for schedule notification from execution thread
      Request Lock
      While(SEL has an event)
            Get event from SEL
            Add event to EEL
            //Add event releases the lock
            Request Lock
      Release Lock
```

Figure 8: Algorithms for Execution Thread and Schedule Thread.

4 RESULTS

A simple simulation is developed for testing purposes to gather execution times to assess the performance of the new threaded simulation executive compared to a base sequential implementation. The simulation application is artificially simply to place the majority of execution effort on the simulation executive as the purpose of the work is to address improving its overhead. The goal of the test simulation is to have strict control of the size of the event list, keeping it constant, to study the impact of the event list size on scheduling overhead. The test case is a fully connected graph of N nodes. Nodes randomly pass entities between each other, when an entity arrives at a node it will depart from that node after a delay time, thus on arrival, an event is scheduled for the entity to complete its delay. The delay event at a node involves randomly selecting a node from the graph as the next destination and scheduling an arrival event at that node after a travel time. A graph node consists of two events: arrive and depart; these events each do exactly one thing, schedule the other event on the appropriate node. Figure 9 illustrates the events of a delay task and how entities are passed between them. This structure ensures that there is one event for each entity in the system. So when an event is executing for an entity, there are E-1 pending events for E entities in the system. Both delay and travel times are defined as constant. This controls the placement of events in an ordered event list, creating the worst case timing for inserting events. In addition, the work involved in executing an event is kept minimal, only involving scheduling the next event, to create the worst case where scheduling dominates execution.

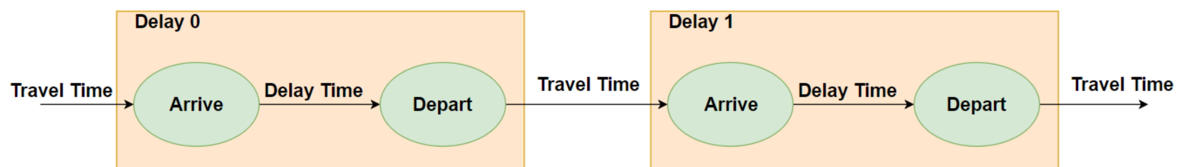

Figure 9: Delay task functionality.

Upon initialization, each node has 10 entities scheduled for arrival, this will immediately schedule 10 depart events into the SEL which then prompts the schedule thread to activate. The number of nodes in the graph is the independent variable for the experiment, this determines the number of entities in the system (E = 10N), and consequently the number of pending events at any given time.

Once all depart events have been scheduled into the SEL from initialization, the simulation runs until it reaches a steady state. This is necessary for the new threaded implementation to allow the events to distribute between the two lists. Once a steady state has been reached statistics are gathered for another 1000 simulation time units.

Modifying the threaded simulation executive's event list structure into two separate event lists causes an unreliable comparison to the traditional serial simulation executive. Not only will the benefits of the multi-threading be seen but also the benefit for splitting the event list in two. An initial comparison to a single thread/single event list produced a performance of threefold, but this does not properly evaluate the benefit of threading. To account for this, the serial simulation executive was modified to have two separate event lists as well. It will schedule events into whichever event list size is smaller and remove the smallest timestamp from either list when executing events. This ensures that each event list is half the size of a singular event list, thus saving half the scheduling time over a singular list.

4.1 Execution Time and Event Statistics Results

The most important result to capture is the execution time for the simulation. The simulation runs until reaching steady state, then the timing begins. The timing ends once the simulation advances an additional 1000 time units. The level of benefit of the threaded simulation execution varies based on the size of the

event list. The threaded simulation executive becomes beneficial between 500 and 1000 events in the system but is a deficit before that. Figure 10 shows a graph with threaded and serial execution times after reaching steady state.

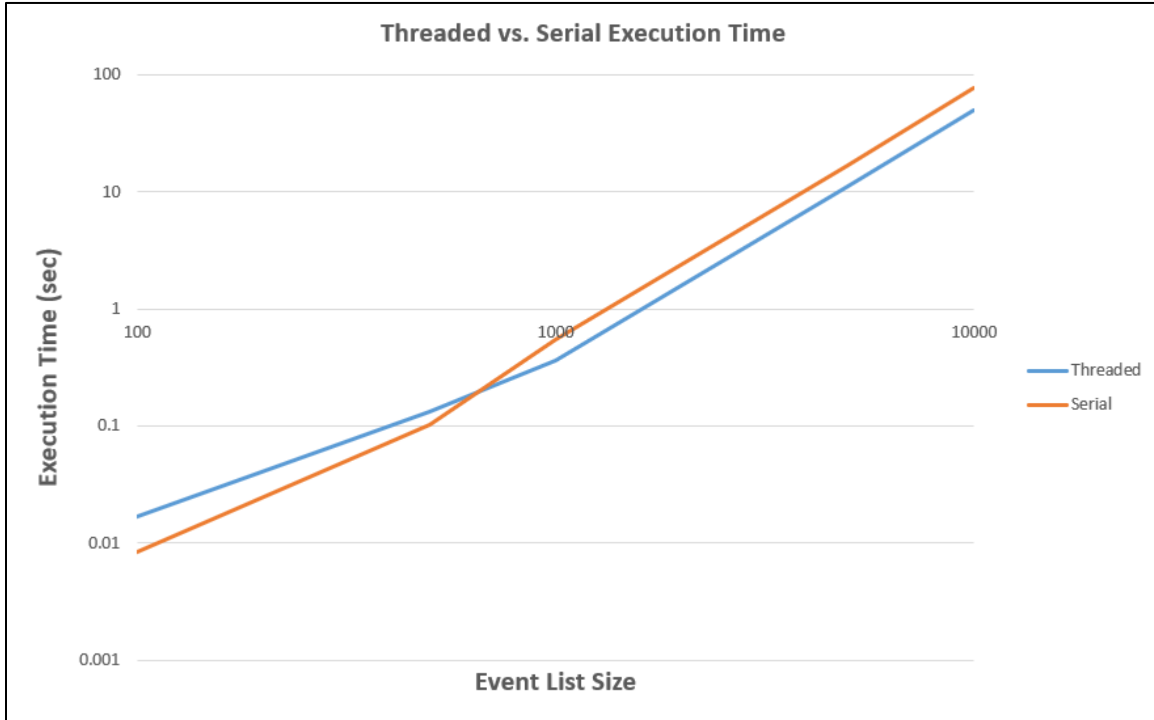

Figure 10: Execution time of simulation after reaching steady state (log-log scale).

In addition, statistics for average event list sizes were captured (independent from capturing the performance statistics). The execution time results indicate that a correlation between the total number of events in the system and execution time. The average number of events in the event lists should also be correlated to the execution time. It is expected that if there are less events in the SEL than either serial event lists then the threaded one should execute faster. The average size of each event list based on the number of events in the system can be seen in Table 1.

Table 1: Average event list sizes.

Number of events in system	**50**	**100**	**500**	**1000**	**5000**	**10000**
Threaded Average EEL Size	28.92	65.45	344.98	639.84	2935.45	6016.04
Threaded Average SEL Size	41.30	63.82	161.41	360.56	2064.54	3983.95
Serial Average Event List 1 Size	24	49	299	499	2499	4999
Serial Average Event List 2 Size	24	49	299	499	2499	4999

The average event list size is calculated when an event is added to that event list. For this example, the average number of events in the system remains constant since the execution of an event adds another event to the system. One would assume that summing the average number of events in each event list would equate to the average number of events in the system. This is the case for the sequential version since the removal and addition of events is 1:1. This is not the case for the threaded version however, there are cases where the addition of the SEL and EEL do not equate to the average number of events in the system. This is because the average event list size is calculated upon the addition of an event to that specific list and does not consider the size of the other list at that moment in time. At a single moment, the sum of the two lists

accounts for all pending events, but the sum of the averages based on size at time of insertion is skewed. In the threaded simulation executive, many events may leave an event list before a new event is added. In the cases with a small number of events in the system, the execution thread can dominate the scheduling not allowing the scheduling thread to transfer any events. For cases with larger number of events in the system, both threads may constantly be working, allowing for the addition and removal of events to be closer to 1:1. If event statistics were captured for both event lists every time an event was added to either list then the averages would sum to the average number of events in the system. This however does not capture the important information, how much work is being done to schedule an event. The benefit that the threaded version presents is correlated by amount of work that the execution thread does scheduling, therefore the important information to capture is the average size of the list when an event is being added to an event list.

4.2 Results Analysis

If the average size of the SEL is higher than the average event list size of either serial event list then it is expected that the serial simulation will run faster, giving no benefit to multithreading. This is the case for all cases except for 500 events in the system. This is most likely due to the total number of events in the system; the threaded simulation executive uses mutex locking. This requires that a thread wait until the lock is available, it is likely that 500 events in the system causes the schedule thread to have control of the mutex more frequently than the execution thread; this could cause the execution thread to be waiting for the scheduling thread to release the mutex more often.

This example is considered a worst case scenario for the threaded simulation executive because the amount of work done by an event is minimal and every event schedules another event. If the event has a heavy work load when executing, the schedule thread can transfer more events from the SEL to the EEL in the event execution time frame. This will get the execution thread's insertion time closer to O(1).

5 FUTURE WORK

5.1 Testing with Realistic Simulation Examples

The test simulation utilized in this work was designed as a worst case scenario. The next step in the research is to develop tests with realistic distributions controlling the relative placement of events in the two event lists. This should reduce the work required for scheduling, allowing the SEL to stay as small as possible. It will also allow studying the relationship between the work required by the application and the simulation executive.

5.2 Reducing SEL List Size

Further work will examine strategies to reduce the SEL size, especially in the presence of real applications. In particular, real applications may have the execution thread schedule more than one event each time executed. In this case, a third list can be maintained holding this relatively very small list. Then this new list can be merged with the SEL faster than individually inserting the events into the SEL.

5.3 Event Lists Data Structures

Both the SEL and EEL are standard ordered linked lists, an potentially inefficient but preliminary step. The time to get an event is O(1) but the time to add an event is O(n) where n is the individual list size. There may be hidden synergy between parallelism and other data structures that exist such as a heap, binary search tree, or calendar queue. The calendar queue seems the most appropriate since the current design has already broken a singular event list into two separate event lists; splitting it into more event lists may attain an larger increase in performance with the potential for a separate thread for each list in the calendar queue. It is also possible that specific data structures may lend themselves better to specific applications.

5.4 Identifying Factors to Reach O(1) Insert/Remove

The goal is for the execution thread to insert/remove events from the SEL in O(1). Altering the underlying data structure, discussed in 5.3, is one way to potentially reach this goal. But the real question is "what conditions need to be met so the SEL can reliably move all events from the SEL to the EEL?" There are two main factors, the average amount of work per event executed and the total number of events in the system. The more work done by events, the longer the scheduling thread has to transfer events. The total number of events in the system may affect the amount of time the scheduling thread takes transfer an event. The current structure using sorted linked lists can reach our goal but will likely require more work per event to achieve and may have constraints on the number of events in the system. The effects of these factors for different types of data structures will need to be investigated.

5.5 Multiple Schedule Threads

Currently there are only two threads, an execution thread and a scheduling thread. It may be possible to increase the number of scheduling threads. More than one scheduling thread transferring events from the SEL to the EEL will further increase the chances that the execution thread will have an add event time of O(1). This would increase the gap between threaded and serial on the right side of Figure 10 The main concern with this approach is the entire EEL becomes in resource contention between all the scheduling threads because the schedule threads may interfere with one another when scheduling thus a more sophisticated lock structure will need to be explored to manage the resource contention. Alternative data structures may address this concern, such as a calendar queue where a separate thread could address each calendar entry.

5.6 Adaptive Event Scheduling

The current threaded simulation executive always has both threads activated, that is that the scheduling thread will always be transferring events from the EEL to the SEL and creating lock contention with the execution thread. We have identified that this architecture is beneficial if there are more than approximately 700 events in the system. The serial simulation executive is better when there are less than 700 events in the system; the scheduling thread could become deactivated if the total events in the system become this way, giving all the work to the execution thread. This puts the threaded simulation executive into "serial" mode until it detects that there are more than 700 events back in the system. Doing this would tighten the left end of Figure 10.

6 CONCLUSION

This paper has identified natural parallelism within the simulation executive, an approach to utilize that parallelism, a solution to combat the resource contention, and results that help identify when a threaded simulation executive is beneficial. A threaded simulation executive takes advantage of the parallelism between event scheduling and event execution, allowing an event to be finished executing by one thread while the other thread handles the scheduling of new events. Doing this does eliminate the burden of scheduling from the execution thread and also raises problems where multiple events are attempting to be scheduled at the same time. The standard single event list structure was separated into two event lists, the execution event list and the schedule event list, to remedy these problems. Resource contention was identified at the first event in both event lists raising the need for a mutex to lock the start of the event lists.

An example was created to test the benefits of the threaded simulation executive. This example is a worst case scenario for the threaded simulation executive because event execution has little work, only scheduling a new event which will always be placed at the end of the SEL. Given a more sophisticated simulation where events have more work to be done, the scheduling thread may remove events from the SEL faster than the execution thread can add. This would cause the execution thread's event insertion to reach O(1) causing a substantial increase in execution speed. The threaded simulation executive shows a

benefit after about 700 events are in the system, even with the worst case scenario demonstrated in this paper. When going up to 10000 events in system, the threaded simulation executive sees an increase of 35% in execution speed. This increase in speed comes free to the application developer because the simulation executive is independent and reusable among many applications.

The work has presented a prototype multithreaded simulation executive. The results clearly justify further investigation into potential parallelism. Ideally keeping the SEL empty would result in the execution thread having O(1) interaction with the scheduling process. But practically, the goal is to keep the SEL as small as possible so that the execution thread has as little overhead as possible, focusing its computational efforts on the application itself.

REFERENCES

Beckmann, C., O. Khan, S. Parthasarathy, A. Klimkin, M. Gambhir, B. Slechta, and K. Rangan. 2010. "Multithreaded Simulation to Increase Performance Modeling *Throughput* on Large Compute Grids". In *Proceedings of the ASLOPS Workshop on Exascale Evaluation and Research Techniques (EXERT)*, March 13th-17th, Pittsburg, PA.

Brown, R. 1988. "Calendar Queues: A Fast O(1) Priority Queue Implementation For The Simulation Event Set Problem". *Communications of the ACM* 31(1):1220-1227.

Bryan, P., J. Poovey, J. Beu, and T. Conte. 2012. "Accelerating Multi-threaded Application Simulation through Barrier-Interval Time-Parallelism". In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Telecommunication Systems*, 117-126. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc. .

Chen, W. 2015. *Out-of-order Parallel Discrete Event Simulation for Electronic System-level Design.* Cham, Switzerland: Springer International Publishing.

Dey, T., W. Wang, J. Davidson, and M. Soffa. 2011. "Characterizing Multithreaded Applications Based on Shared-resource Contention". In *Proceedings of the 2011 IEEE International Symposium on Performance Analysis of Systems and Software*, 76-86. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Fujimoto, R. M. 1990. "Parallel Discrete Event Simulation". *Communications of the ACM* 33(10):30-53.

Fujimoto, R. M. 2000. *Parallel and Distributed Simulation Systems.* New York, NY: John Wiley & Sons.

Moseley T., J. Kihm, D. Connors, and D. Grunwald. 2005, "Methods for Modeling Resource Contention on Simultaneous Multithreading Processors". In *Proceedings of the 2005 International Conference on Computer Design*. 373-380. October 2nd-5th, Los Alamitos, CA. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Kunz, G., M. Stoffers, J. Gross, and K. Wehrle. 2011. "Runtime Efficient Event Scheduling in Multi-threaded Network Simulation". In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*. March 21st-25th, Barcelona Spain.

Law, A. M. and W. D. Kelton. 2004. *Simulation Modeling and Analysis*. Boston, MA: McGraw-Hill, Inc.

Pidd, M. (2004). "Simulation Worldviews – So What?" In *Proceedings of the 2004 Winter Simulation Conference*, edited by R. G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, 288-292. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Ronngren, R., J. Riboe, and R. Ayani. 1991. "Lazy Queue: An Efficient Implementation of the Pending-event Set". *ACM SIGSIM Simulation Digest* 21(3):194-204.

SOFA. 2019. SOFA: Simulation Open Framework Architecture. https://www.sofa-framework.org/applications/marketplace/multithreading-simulation-level/, accessed 11th April 2019.

Tang, W., R. Goh, and I. Thing. 2005. "Ladder Queue: An O(1) Priority Queue Structure for Large-Scale Discrete Event Simulation". *ACM Transactions on Modeling and Computer Simulation* 15(3):175-204.

Vaucher, J. G. and P. Duval. 1975. "A Comparison of Simulation Event List Algorithms". *Communications Of The ACM* 18(4):223-230.

Wang, J., D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev. 2014. "Parallel Discrete Event Simulation for Multi-Core Systems: Analysis and Optimization". *IEEE Transactions on Parallel and Distributed Systems* 25(6):1574-1584.

AUTHOR BIOGRAPHIES

BRANDON WADDELL is a Ph.D student in the Department of Computational Modeling and Simulation Engineering at Old Dominion University. He received his bachelor's degree in Modeling, Simulation, and Visualization Engineering at Old Dominion University. His research interests include parallel simulation and networking. His email address is bwadd006@odu.edu.

JAMES F. LEATHRUM JR. is an Associate Professor in the Department of Computational Modeling and Simulation Engineering at Old Dominion University. He earned his Ph. D in Electrical Engineering from Duke University. His research interests include simulation software design, distributed simulation, and simulation education. His email address is jleathru@odu.edu.