Parallelizing a Discrete Event Simulation Application Using the Habanero-Java Multicore Library

Wei-Cheng Xiao
Department of Computer Science, Rice University
6100 Main St.,
Houston, TX, USA
garry@rice.edu

Jisheng Zhao
Department of Computer Science, Rice University
6100 Main St.,
Houston, TX, USA
jisheng.zhao@rice.edu

Vivek Sarkar
Department of Computer Science, Rice University
6100 Main St.,
Houston, TX, USA
vsarkar@rice.edu

ABSTRACT

Discrete event simulation (DES) has been widely adopted for simulating communication systems such as computer networks. As the network size and complexity of communication patterns increases, the complexity of simulation tools and the execution time of DES also increases. Parallelizing DES programs using multiple processing units reduces the overall execution time; however, unlike regular programs, data dependencies in DES are usually determined at runtime, which makes exploiting potential parallelism in the program very challenging. In this paper, we build a parallel version of a DES program written in Java using the Habanero-Java library (HJlib), which is a lightweight and programmer-friendly parallel Java 8 library. While the DES problem benefits greatly from HJlib's support for lightweight tasks and efficient parallelism based on work stealing, it also pushed the boundaries of the standard primitives available in HJlib. In particular, our study motivated the addition of fine-grained locking to the Habanero execution model, in a manner that still preserves Habanero's deadlock freedom guarantees. This extension in turn led to additional optimizations of the DES implementation relative to the original Galois implementation. Our initial results are encouraging, and point to further opportunities for exploiting parallelism in challenging applications like DES on manycore hardware platforms in the future, especially as the system being simulated increases in size and complexity.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – Parallel Programming

General Terms

Parallel programming, Java, Simulation, Performance

PMAM '15, February 7-8, 2015, San Francisco Bay Area, USA

http://dx.doi.org/10.1145/2712386.2712402

Keywords

Habanero Java, irregular program, discrete event simulation

1. INTRODUCTION

In the real world, constructing, evaluating, and analyzing communication systems, such as computer networks, electric circuits, etc, usually consumes lots of time and resources, including preparing the hardware, setting up the environment, and performing experiments. Discrete event simulation (DES) offers a cheaper and faster way to build and test such communication systems in the simulated world by leveraging the flexibility of simulation software and the availability of powerful computers. DES simulates message passing and processing in communication systems on an event-by-event basis. For each entity in a communication system, new events can be generated from the initial states of the system or upon receiving messages from other entities in the system. An event can cause state changes in some entities, which may generate new events or messages and further change the states of other entities in the system. Each event is associated with a timestamp, which is usually the logical time at which the event is generated. When the simulation is running, the simulated time proceeds as events are generated and processed, and with the constraint that events are processed in a manner that is consistent with their timestamp order.

Although DES saves time and effort in testing and analyzing communication systems compared with real-world experiments, running simulations of large and complicated systems still takes a long time. One opportunity to speed up these simulations is to use parallel computing, especially with the increased availability of multicore and manycore processors. Since multiple events might be generated and processed by different entities in the system and independent of each other, this is where parallelism could be exploited. However, it is well known that parallelization of an application like DES can be very challenging, compared to applications in other domains. Improper parallelization of the simulation may create livelock, deadlock, or starvation, or even violate the dependency of events and lead to incorrect simulation results. Unlike regular programs, data dependency in parallel DES is usually determined at runtime, which makes exploiting potential parallelism in the program more challenging. Due to the challenges mentioned above, designing a correct and efficient parallel version of a sequential DES program may be complicated and thus not programmer-friendly. In this paper, we adapted a DES program that

modeled logic circuit simulation in the Java version of the Galois system [1, 12] and parallelized the simulation program using the Habanero-Java library (HJlib) [13], which is a lightweight and programmer-friendly parallel Java 8 library implementation. HJlib allows programmers with basic Java knowledge to convert sequential programs, either regular or irregular, into parallel programs without worrying about the details of internal task or thread management. We used the *async/finish model* provided in HJlib to spawn and synchronize tasks in our implementation, in which each new task is created to process events for one logic gate at a time. The work-stealing and load balancing features inside HJlib provided efficient task scheduling and resource management, which simplified the task of creating a parallel implementation of the logic circuit simulation. While the DES problem benefits greatly from HJlib's support for lightweight tasks and efficient parallelism based on work stealing, it also pushed the boundaries of the standard primitives available in HJlib. In particular, our study motivated the addition of fine-grained locking to the Habanero execution model, in a manner that still preserves Habanero's deadlock freedom guarantees. This extension in turn led to additional optimizations of the DES implementation relative to the original Galois implementation.

Our initial results are encouraging, and point to further opportunities for exploiting parallelism in challenging applications like DES on manycore hardware platforms in the future, especially as the system being simulated increases in size and complexity.

We ran the logic circuit simulation with three different circuit and initial event settings on a POWER7 multiprocessor with up to 32 cores. Compared with Galois, our HJlib version reduced the execution time by 44.5-79.7%. We believe that HJlib could also be used to parallelize DES for communication systems with larger scale and higher complexity, such as network simulators [2], in the future.

The rest of this paper is organized as follows. In Section 2, we discuss related work in approaches to implementing DES in parallel. Section 3 presents the background on HJlib, including the work-stealing and load balancing features. We describe the detailed implementation of our HJlib version of parallel logic circuit simulation, including how we exploit potential parallelism in the program, in Section 4. Performance evaluation results for both Galois and our HJlib versions of parallel logic circuit simulation are shown in Section 5. Finally, we conclude this paper and describe our future work in Section 6.

2. RELATED WORK

2.1 Parallel discrete event simulation

There has been much research on the area of parallel discrete event simulation (PDES) for over two decades. In a DES, parallelism can be exploited in various ways [23,26], including 1) applying a parallelizing compiler to the sequential simulation implementation, 2) separating independent simulation runs on multiple processors [4], 3) running subroutine calls in the simulation on different processors, 4) maintaining a global event list and having multiple processes access and process the events in the list simultaneously [17], and 5) decomposing the simulation into multiple components in time or space domain and running the components on multiple processors at the same time [6, 21]. In this paper, we parallelize the simulation program using the last approach, dividing events in the system in the space domain, through which we may have the greatest potential of exploiting parallelism as the system being simulated increases in size and complexity.

To ensure correctness in a PDES, event processing algorithms must obey the *causality constraint*: For two events $e1$ and $e2$, if $e2$ depends on $e1$, then $e1$ must be executed before $e2$; otherwise an error may occur. As shown in [11, 21], a PDES obeys the causality constraints if and only if every logical process (LP) executes their local events in timestamp order, which is called the *local causality constraint*. Algorithms that obey the local causality constraint fall into two categories–*conservative* and *optimistic*.

The development of conservative simulation algorithms starts from the work by Chandy, Misra, and Bryant [5,6,21]. In conservative algorithms [9, 10, 25], each LP processes its local events strictly in timestamp order. Each time an LP runs, it is allowed to process only *safe events*, which are local events that have timestamps smaller than any events the LP may receive in the future; otherwise, the LP must block itself, which may cause deadlocks in the system. Deadlock avoidance using *null messages* [21] as well as deadlock detection and recovery [7] mechanisms were proposed to solve the problem of possible deadlocks. In this paper, we focus on demonstrating the efficiency and simplicity of using the high-level parallel programming primitives available in HJlib. Although we use conservative algorithms and mechanisms similar to the null messages in the simulation program, the simulation program we choose to study is simple, and LPs (tasks) in our simulation do not need to block themselves. Also, we do not need null messages in the middle of event processing but only after each node finishes processing all its local events.

On the other hand, in optimistic algorithms [3, 24, 27], an LP may process its local events out of the timestamp order, which may reduce the amount of time the LP spends on idling or blocking. This mechanism, however, could cause incorrect results. Jefferson and Sowizral [15, 16] proposed a Time Warp mechanism, in which the LP performed a *rollback* when an out-of-order events was detected, to solve this problem. When performing the rollback, the LP restores the simulation from a saved state of simulation time no later than current local clock and restarts the simulation from that state.

2.2 Galois

Galois [1, 22] is an object-based optimistic parallelization system built for efficiently running irregular applications. There are three main aspects in this system:

- library constructs called optimistic iterators for packaging optimistic parallelism as Iteration over sets;
- runtime scheme for detecting the conflicting shared data accesses and recovering from those unsafe accesses;
- assertions about methods in class libraries.

Its optimistic iterators include: **set iterator** (i.e. unordered) and **ordered-set iterator**. In both of these two iterators, the set elements are executed as activities that are running in parallel.

Galois offers a sequential, object-oriented programming model which is closed to sequential Java programming model, and Galois runtime is in charge of managing conflict detecting and recovering for those speculatively parallelized activities. Thus, users can use Galois class library to build parallel applications, and the code pattern is like the simple workset based approach. Based on the evaluation shown in [19, 22], the parallel applications written in Galois approach presented good scalability on state-of-art multicore system. However, to use Galois, users still have to learn the Galois program pattern, i.e., how to build the workset with optimistic iterators and how to judge the granularity of workset elements (i.e. the parallel activities) and adapt to the Galois model. While in the approach we presented in this paper, we used a Java-based lightweight task-parallel library, which provides language constructs that enables users to easily manipulate task creation and synchronization designs.

The Galois project has undertaken a deep analysis of available parallelism for many irregular programs including DES. Figure 1 shows the available parallelism that they observed in the DES algorithm for an input circuit of a 6x6 tree multiplier [1]. Initially the parallelism is limited because of the small number of input ports. Then the parallelism builds up due to large fanouts in the middle of the circuit. Finally, the parallelism decreases again due to the small number of output ports. This insight sets expectations for the limited speedups that we will see later in the paper, while also offering hope that larger speedups may be observed when simulating larger circuits.

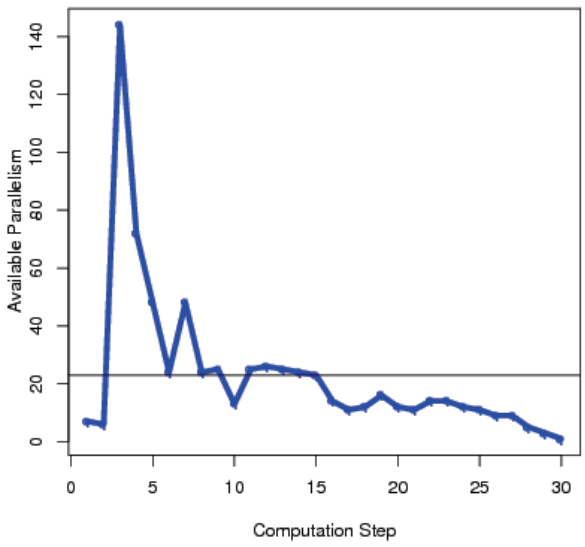


Figure 1: An example of available parallelism in DES (source: Galois website [1])

3. HABANERO JAVA LIBRARY

Habanero-Java library (HJlib) [13] is a library implementation of the pedagogic Habanero task-parallel programming model. HJlib is built using lambda expressions and can run on any Java 8 VM without any other dependencies. This library supports a series of parallel programming constructs that allow users to write task parallel Java program using different parallel programming patterns, including data parallelism, pipeline parallelism, stream parallelism, and divide-and-conquer parallelism.

3.1 Task spawning and synchronization

async: The statement "**async** (()-> <*stmt*>)" causes the parent task to create a new child task to execute <*stmt*> , *asynchronously* (i.e. before, after, or in parallel) with the remainder of the parent task (see Figure 2).

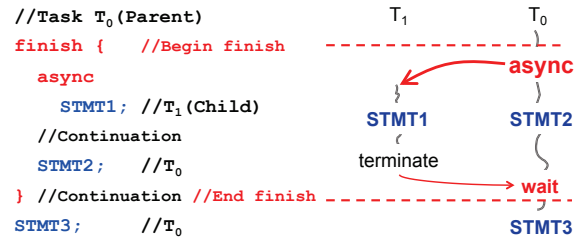


Figure 2: Async/Finish Model

finish: is a generalized join operation (i.e. task synchronization). The statement "**finish** (()-> <*stmt*>)" causes the parent task to execute <*stmt*> and then wait until all **async** tasks created within <*stmt*> have completed, including transitively spawned **async** tasks. Each dynamic instance T_A of an **async** task has an unique *Immediately Enclosing Finish* (IEF) instance F of a **finish** statement during program execution.

In an HJlib program, the user can create an unbounded number of dynamic tasks and leave the responsibility of scheduling these tasks on a fixed number of processors to the HJlib runtime work-stealing mechanisms.

3.2 Mutual exclusion

HJlib also provides an **isolated** construct that supports weak isolation, which means that any dynamic instance of an isolated statement is guaranteed to be performed in mutual exclusion with respect to all other potentially parallel dynamic instances of isolated statement. The statement "**isolated** (*var*$_1$, *var*$_2$ ... *var*$_i$, ()-> <*stmt1*>)" guarantees that each instance of <*stmt1*> will be performed in mutual exclusion with all other potentially parallel interfering instance of "**isolated** (*var*$_j$, *var*$_{j+1}$ ... *var*$_n i$, ()-> <*stmt2*>)", if the intersection between two sets *var*$_1$, *var*$_2$ ... *var*$_i i$ and *var*$_j$, *var*$_{j+1}$ ... *var*$_n$ is not empty. The statement "**isolated** (()-> <*stmt*>)" guarantees mutual exclusion between each instance of <*stmt*> with all other **isolated** statements (i.e. this implies that all objects are involved in mutual exclusion for <*stmt*>). A key semantic property for HJlib is the deadlock freedom property, which states that no HJlib program written using **async**, **finish**, and **isolated** constructs can deadlock. (This property holds with some additional HJlib constructs as well, including futures, barriers, phasers, and actors.)

Based on our experience gained with the DES application, we propose allowing the two lock APIs to be used in HJlib programs:

- TryLock(*var*): tries to acquire a runtime managed lock for the given object (i.e. *var*), and returns *true* if the acquisition is successful and *false* otherwise;

- ReleaseAllLocks(): releases all of the locks held by the current **async** task.

The implementation of these APIs is flexible. In this paper, we chose Compare-And-Swap (CAS) objects provided from Java Utility of Concurrency (JUC) as the low level implementation. For each lock object, we create a corresponding `AtomicBoolean` object with initial value *false*, and the boolean value of the object indicates whether the lock is currently held by some node or not. TryLock() calls `AtomicBoolean`'s member function `compareAndSet()` to atomically check and set the value to *true*. For the function ReleaseAllLocks(), it calls another member function `set()` to set the values of all the `AtomicBoolean` objects that are currently held by a node to *false* to release the locks. Though these lock APIs are lower-level concurrency constructs than other HJlib primitives, they are important for exploiting parallelism in the DES application. Further, these two specific APIs are guaranteed to not create a deadlock either. Livelock is a possibility, however, and we will later discuss how livelock is avoided in our implementation. Besides, we believe that the use of TryLock() and ReleaseAllLocks() APIs incurs less overhead in Java programs, compared to two-phase locking protocols.

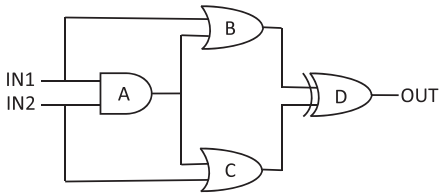
4. LOGIC CIRCUIT SIMULATION

In this paper, we choose a logic circuit simulation implementation and parallelize it using HJlib as a case study for DES applications. We obtained the sequential version of logic circuit simulation implementation from the Java version of the Galois system [1], removed the dependencies on the Galois library, restructured the code for easier and more efficient parallelization, and applied HJlib to parallelize the simulation program. Detailed scenarios and algorithms for the logic circuit simulation are presented in Section 4.1. Section 4.2 describes the sequential version of the DES simulation, and Section 4.3 presents the details of implementing DES using HJlib. Section 4.4 discusses the similarities and differences between the HJlib and Galois implementations. In Section 4.5, some parallel DES-specific optimizations are introduced.

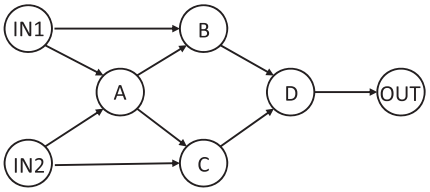
4.1 Logic circuit simulation using DES

In the simulation, a logic circuit is represented as a directed graph. Each logic gate in the circuit is a node in the graph, and the connections of the input and output ports of neighboring logic gates are represented as directed edges in the graph. A logic gate has one output port and one or two input ports, depending on the type of the gate. Beside the logic gates, each input and output of the circuit is also represented as an *input node* and *output node* respectively in the graph. In the simulation, we assume that each input port of a logic gate is connected to one and only one neighboring gate's output port, whereas the output port of a logic gate may connect to the input ports of one or more neighboring gates (fanout). An input node only has outgoing edge(s), and an output node only has an incoming edge. We also assume that there is no loop in the circuit/graph. Figure 3 shows an example of a logic circuit and its corresponding graph representation.

Every electric signal in the circuit is represented as an *event* in the simulation, and signal propagation is simulated as message passing between neighboring nodes in the graph.



(a) Logic circuit



(b) The graph representation

Figure 3: An example of a logic circuit and its graph representation

Signals generated at circuit inputs are called *initial events* in the simulation. At the beginning of a simulation, a logic circuit in its graph representation along with a list of initial events for each input node are given as the input to the simulation. During the simulation, each initial event generated by the input nodes will cause new messages and additional events to be generated, processed, and propagated through the edges in the graph to the output nodes. The simulation ends after all the events, including initial events and events generated during the simulation, are processed. Every event has a timestamp associated with it. The timestamp is the time when an event should be processed by the node which is holding that event. A delay is applied between the time the node processes an event and the time the node's neighbors receive the processing result. This simulates signal processing time of a logic gate and signal propagation time between gates in the real world. In the simulation, the signal propagation time is assumed to be constant, and for each type of logic gate, a constant processing delay is assigned in the program.

During the simulation, there might be multiple events that have been generated and queued but not yet processed. In order to correctly simulate the behavior of a logic circuit, events must be processed in their timestamp order. Processing all the events globally in the system in order is sufficient but not necessary to guarantee correctness of the simulation, and that limits the room for parallelism. Chandy and Misra [6] proposed a distributed algorithm for DES, which provides us more opportunities to parallelize the logic circuit simulation. Their simulation algorithm removes the requirement of global event controlling and allows every individual station (node) to process its local events independently and simultaneously; however, nodes must follow certain rules to guarantee the correctness of the simulation. For example, every node must process its local events in the timestamp order, and it can process events only after having received events from all its inputs. By applying their solution to the logic circuit simulation, each node maintains its own local clock and event queue. A node's local clock is defined as the minimum of the timestamps of the last event the node has received on all its input ports. All the events in the node's

Algorithm 1 Pseudocode for the sequential implementation of the logic circuit simulation

Input: G: the graph representation of the input circuit, I: the set of input nodes in graph G, $I \subset G$, and initial events of each node in I

```
1: WS ← I                                   ▷ WS is the workset.

2: for node n in WS do
3:     WS ← WS − n

4:     SIMULATE(n)

5:     for node m in n ∪ n.neighbors do
6:         if ISACTIVE(m) then
7:             WS ← WS ∪ m
8:         end if
9:     end for
10: end for
11:                                  ▷ Simulation terminates here

12: function SIMULATE(n)
13:     for event e in n.readyEvents do
14:         newEvent ← e.PROCESS()
15:         for node m in n.neighbors do
16:             m.ADDEVENT(newEvent)
17:         end for
18:         n.readyEvents ← n.readyEvents − e
19:     end for
20: end function

21: function ISACTIVE(n)
22:     n.UPDATELOCALCLOCK()
23:     n.UPDATEREADYEVENTS()
24:     return n.readyEvents ≠ φ
25: end function
```

event queue with timestamp smaller than or equal to the node's local clock are called *ready events*. Ready events are safe to be removed from the queue and processed in their timestamp order to generate new events for the neighboring nodes in the fanout. Since ready events are processed in the timestamp order, new events will also be generated and stamped in the order of time, which means, for any input port of any node in the graph, events come in the order of time. Thus, for any node, any new event coming from any input port of the node will have a timestamp greater than or equal to the timestamps of all current ready events of the node. This proves that by processing ready events only, each node is guaranteed to process all events it ever receives in the timestamp order throughout the whole simulation. Note that two ready events with the same timestamp can be processed in any order, and that does not affect the correctness of the final results.

In the simulation, due to the absence of global control, we need a mechanism for each node to know whether there will be additional events coming from the input port(s) or not. By applying Chandy and Misra's algorithm [6], after an input node sends out all its initial events, it sends a NULL message with timestamp infinity to inform all its neighboring nodes in the fanout that there will be no new events anymore. Similarly, after a normal node has received

Algorithm 2 Pseudocode for parallel logic circuit simulation using HJlib

Input: G: the graph representation of the input circuit, I: the set of input nodes in graph G, $I \subset G$, and initial events of each node in I

```
▷ SIMULATE() and ISACTIVE() are shared from Algorithm 1.

1: finish RUN()                          ▷ Simulation terminates here.

2: function RUN()
3:     for node n in I do
4:         async RUNNODE(n)
5:     end for
6: end function

7: function RUNNODE(n)
8:     for node m in n ∪ n.neighbors do
9:         if TRYLOCK(m) ≠ successful then
10:            RELEASEALLLOCKS()
11:            if m ≠ n then
12:                async RUNNODE(n)   ▷ try n again later
13:            end if
14:            return
15:        end if
16:    end for

17:    SIMULATE(n)

18:    RN ← φ
19:    for node m in n ∪ n.neighbors do
20:        if ISACTIVE(m) then
21:            RN ← RN ∪ m
22:        end if
23:    end for
24:    RELEASEALLLOCKS()
25:    for node m in RN do
26:        async RUNNODE(m)
27:    end for
28: end function
```

NULL messages from all its input ports, it sends out a NULL message via its outgoing edges. After all the output nodes receive a NULL message, the simulation terminates.

4.2 The sequential implementation

Algorithm 1 shows the pseudocode of the sequential implementation of the logic circuit simulation application. The algorithm is adapted from the DES benchmark implementation in the Galois system [12]. In the implementation, a *workset* (*WS*) is maintained to contain current *active nodes* in the system. An active node is defined as a node which has one or more ready events in its event queue. All initial events in the input nodes are ready events. Active nodes in the workset can be pulled out from the workset and run in any order. When an active node is running, its ready events are removed from the event queue and processed (by calling PROCESS()) one-by-one in their timestamp order, and newly generated events are then "sent" (through ADDEVENT()) to the event queues of the neighbors in the fanout. The call *e*.PROCESS() simulates the operation of a

logic gate upon event e and adds the signal processing and propagation delay to the timestamp of e as the timestamp of the new event *newEvent*. Since the neighboring nodes have received new events after the ready event processing, their local clocks may need to be advanced, and new local clocks may cause some existing events in their queues to become ready events. Functions UPDATELOCALCLOCK() and UPDATEREADYEVENTS() are called to update the local clock and ready events of a node respectively. If any node is found to become active in this run, the node is then added to the workset. Since active nodes in the workset are independent of each other and can be run in any order, it is possible to run different active nodes in parallel. Next, we will describe our parallel implementation using HJlib.

4.3 The parallel implementation using HJlib

We slightly modified the structure of the sequential implementation and parallelized it using HJlib. Algorithm 2 shows the pseudocode of our parallel implementation. In our implementation, we used the *async/finish model* in HJlib to spawn and synchronize tasks that are created during the simulation. When the simulation starts, a new task is created for each input node in the graph via the **async** statement. Similarly, during the simulation, every time a node is found to be active, a new task is also created in the same way for that node to be run later. Upon the creation of a task, the task is pushed into a deque and waits for future execution. When a task is ready to be executed, it is popped out from the deque it is currently located at and executed on a thread. When the task is running, the function RUNNODE() is called to process the ready events of the corresponding node. HJlib manages tasks, task deques, and threads internally, delegating each task to an available worker thread (typically one per core) through work-stealing and load balancing to minimize overheads and maximize the overall efficiency. Note that in our implementation, task deques replace the workset in the sequential implementation to the queue active nodes that are ready to run.

If multiple active nodes are running in parallel without any control, data races may exist since multiple nodes may be adding new events to the event queue of the same neighbor simultaneously. To avoid data races, we use *locks* to prevent concurrent access to the same node. Each node is associated with a lock, and each lock can be held by at most one node at the same time. When an active node is ready to run on a thread, it first tries to acquire the locks of itself and all its neighboring nodes in the fanout. If any of the trials fails, it releases all the locks it has successfully acquired and tries again later by spawning a new task on itself via the **async** statement; otherwise, it goes ahead to process the ready events in its event queue, generates new events based on the processing results, and adds the new events to its neighbors' event queues. After that, the node releases all the locks it has acquired in this run. Note that we can choose to have each node maintain a concurrent priority queue to store local events and not to use any locks in the user's implementation, which may increase the parallelism at runtime. However, this may lead to too high overhead especially when the number of events is large. This is because every time a concurrent priority queue is accessed, some locking mechanism is still required internally. Compared with the concurrent priority queue-based scenario, locking the entire queue throughout the whole run of event processing, i.e., one RUNNODE() call, saves us lots of overhead and achieves better performance.

In our implementation, all the tasks spawned via **async** during the simulation are synchronized at the end of the **finish** statement, and then the simulation terminates. In HJlib, the **finish** statement determines the scope in which all the tasks spawned via the **async** statement need to be synchronized, and it has to wait for all descendant tasks to terminate before it can proceed.

Our implementation (Algorithm 2) is guaranteed to be deadlock-free for the following reasons.

- One possible situation of deadlock is that all nodes in the system are waiting for new events and none of them can proceed to process local events, however; this has been proved to be impossible in Chandy and Misra's algorithm [6], which our implementation is based on.

- We use the async/finish model in HJlib, which is proven to be free of deadlocks [8].

- When an active node is trying to acquire locks of itself and all its neighbors in the fanout, it never blocks on any TRYLOCK() call. If it fails on any lock acquisition, it immediately releases all the locks it has successful held in the current run; otherwise, it releases all the locks after the SIMULATE() call. Therefore, no active node will be blocked when it is running, and every active node in the task deque will eventually get a chance to run.

When multiple active nodes are trying to acquire locks of the same or similar set of neighbors without a specific order, livelocks may occur, since each of them may have successfully acquired some but not all locks and then release the locks it has acquired and try again, and this process may keep repeating. To avoid livelocks, in our implementation, each node is assigned a unique node ID. When an active node is trying to acquire locks of itself and its neighbors, it acquires the locks in the ascending order of the node IDs. This guarantees that one of the active nodes which are competing on the locks of the same or similar set of neighbors simultaneously will win and go on to process its ready events, thereby guaranteeing livelock avoidance.

4.4 Comparison with Galois Approach

As mentioned in Section 2, following the Galois approach to parallelization requires the use of Galois operators and library APIs. This makes it harder to perform domain-specific optimizations on the parallelism structure, compared to performing these optimizations in an HJlib version of the application.

Algorithm 3 gives the simplified Galois representation of logic circuit simulation [1], which is also similar to Algorithm 1. The key difference with the HJlib approach lies in the Galois **for each** operator that executes each loop iteration (line 3-15) in parallel and performs the conflict detection, recovering and re-execution. In this case, the user can not check the ownership of the objects that are going to be acquired and decide if the execution should be performed or not, thus the cautious optimization [20], e.g., Algorithm 2 line 9-15, cannot be applied to this Galois implementation in a convenient manner.

Algorithm 3 Pseudocode for the Galois implementation of the logic circuit simulation

Input: G: the graph representation of the input circuit, $initEventNodes$: the graph nodes that contain initial events and need to be used for initializing work set.

```
1: WS ← new workset                          ▷ WS is the workset.
2: WS.addAll(initEventNodes)

3: foreach node n in WS do
4:     Simulate(n)

5:     for node m in n ∪ n.neighbors do
6:         if isActive(m) then
7:             WS ← WS ∪ m
8:         end if
9:     end for
10: end for
11:                                          ▷ Simulation terminates here
```

4.5 Optimizations

We optimized our logic circuit simulation implementation in a few ways. Those optimizations increased the successful rates of lock acquisition for active nodes and reduced the overall execution time and parallelism overheads. The optimizations are summarized below. We observe that these optimizations are harder to perform in user code for a Galois application compared with an HJlib application, though they can be performed by modifying the internals of the Galois library.

Note that in order to make the pseudocode as simple and clear as possible, we do not show the optimizations in Algorithm 2.

4.5.1 Separated deque for each input port

Instead of maintaining a single queue for each node, we create a queue for each input port of a node. Although this increased the number of queues in the system, it actually reduces the overhead of queue maintenance. For a node, events coming into each input port arrive in the timestamp order, as explained previously; however, events from different input ports may arrive at any order. Since every node needs to process its local events in the timestamp order, maintaining a single event queue for each node requires either 1) the event queue to be a priority queue (`java.util.PriorityQueue`), or 2) events to be sorted each time we check whether the node is *active* or not; both approaches have significant overheads. Instead, if a separated queue is maintained for each input port, we can use more lightweight array deques (`java.util.ArrayDeque`) for queue maintenance. When checking whether a node is active and finding out ready events from the deques, all we need to do is pop out events with timestamps smaller than or equal to the local clock from the heads of the deques in the order of their timestamps.

Another benefit of separating the deque is the increase of parallelism. To realize that, we maintain a lock for each input port for every node in the system, where each lock is no longer a per-node lock but a per-port lock. When a node A is trying to lock one of its neighbors B (by calling TryLock()), A does not need to lock the entire node B. Instead, A only needs to lock the input port with which B is connected to A. Thus, it allows two different nodes to successfully acquire locks of the two input ports of the same neighbor and adding new events to the two event deques of the neighbor simultaneously. This increases the successful rate of locking neighbors and therefore the overall parallelism increases. Note that an active node must acquire the locks of *all* its input ports before it can process its ready events, since its ready events may come from all its event deques. If all the input ports of the active node keep being locked throughout the current run of event processing, none of the active node's upstream neighbors can add new events to it. To remove this constraint and further increase parallelism, we create an additional queue for each node to temporarily store ready events. When an active node starts to run, it first acquires the locks of all its input ports, moves ready events from the per-port event deques to the temporary queue in their timestamp order, then releases all the locks of its input ports. When the active node is processing ready events later, it takes events out only from the temporary queue and does not touch any of its events deques, and its upstream neighbors can add new events to it at the same time.

4.5.2 Simple AtomicBoolean locks

To implement the per-port lock, we choose the lightweight `AtomicBoolean` in the Java library instead of using more complicated lock implementations, such as `ReentrantLock`. The `AtomicBoolean` implementation allows us to check and set the value of a boolean variable atomically when multiple processes or threads are trying to access the variable concurrently. We create an `AtomicBoolean` variable for each lock in our implementation, and the value of the variable indicates whether the lock is currently held by some node or not. The methods `compareAndSet()` and `set()` in the `AtomicBoolean` class provide us a lightweight and simple way to implement the functions ReleaseAllLocks() and TryLock() used in the simulation.

4.5.3 Avoiding unnecessary async statements

In Algorithm 2, we use the **async** statement to create new tasks and increase parallelism. If we do not treat task creation carefully, we may create redundant tasks or spawn tasks earlier than necessary; doing so does not affect the correctness of the simulation, but doing so can add extra overheads to the program execution. Therefore, it is important to avoid unnecessary **async** statements in the program. When an active node is trying to acquire all locks of its own input ports, if it fails on any of them, it does not need to spawn a new task for itself to try again later (line 11). This is because failing to acquire a lock of the node itself means that at least one of its upstream neighbors is holding the lock and will spawn a new task for the node after the neighbor finishes its current run of event processing. Similar to the workset implementation in the sequential version, we do not need redundant nodes in the task deques inside HJlib. Also, for each node which the active node is trying to spawn a new task for (line 18- 27), if the node has one or more locks held by others at the same time, the new task does not need to be spawn for the node (line 26 does not executed in this case). One of the nodes that are still holding a lock of the node will spawn a new task for the node later.

Circuit	Multiplier 12 bits	Kogge-Stone adder, 64 bits	Kogge-Stone adder, 128 bits
# nodes	2,731	1,306	2,973
# edges	5,100	2,289	5,303
# initial events	49	128,258	66,050
# total events	56,035,581	89,683,016	102,591,960

Table 1: Profiles of the input circuits used in the simulation

	Multiplier 12 bits	Kogge-Stone adder, 64 bits	Kogge-Stone adder, 128 bits
HJlib	31934	49004	66363
Galois (Java)	84077	134061	163643

Table 2: Minimum execution time (s) of the sequential simulation

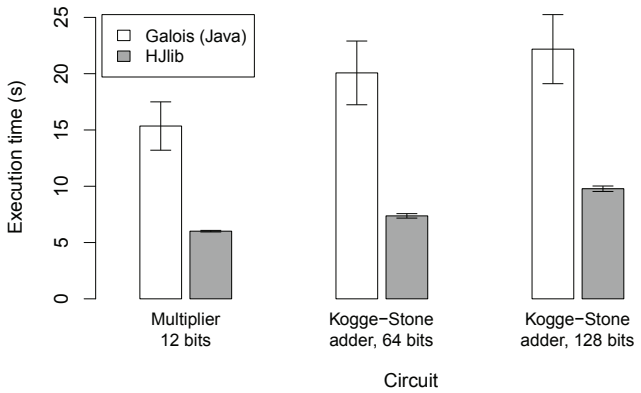


Figure 7: Average execution time of both Galois and HJlib versions of the 32-worker case

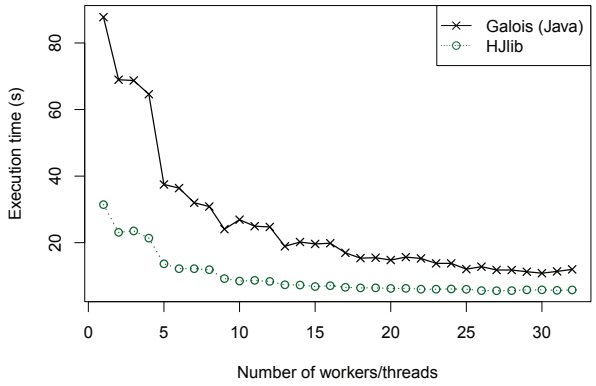
5. PERFORMANCE EVALUATION

In this section, we present experimental results for logic circuit simulation using the DES application. We compared our HJlib version of the simulation implementation with the Galois-Java version for three different input logic circuits – a 64-bit and 128-bit Kogge-Stone tree adder [18] and a 12-bit tree multiplier. Detailed information of the input circuits are shown in Table 1. The logic circuit simulation implementation of the Galois-Java version was kept unchanged from the downloaded version throughout the experiments. We ran the simulation on a single machine with 256 GB of RAM and four eight-core IBM POWER7 64-bit processors (32 cores) running at 3.8 GHz. The operating system was Linux with kernel version 2.6.32, and the Java environment was IBM J9 VM with JRE 1.8.0. The maximum Java heap size was set to 16 GB for all runs. For each input circuit and number of workers/threads, we ran the simulation for 20 times. Figures 4, 5, and 6 show the minimum execution times and corresponding speedups measured for all three input circuits. In addition to the minimum execution times, we also evaluated the average execution times and confidence intervals and showed the results for the 32-worker case in Figure 7. The Concurrent Mark Sweep collector (`-XX:+UseConcMarkSweepGC`) was chosen for garbage collection in JVM for execution time evaluation. In addition, we also ran the simulations of both Galois-Java and our versions of sequential implementation using the serial garbage collector (`-XX:+UseSerialGC`) and used the sequential execution times of the Galois-Java version as the baselines for speedup calculation. Our sequential version was implemented as Algorithm 1, whereas the sequential version of Galois-Java was implemented without using the Galois parallel runtime library. The execution times of the sequential simulations are shown in Table 2. All the execution times include time spent on garbage collection and other JVM services.

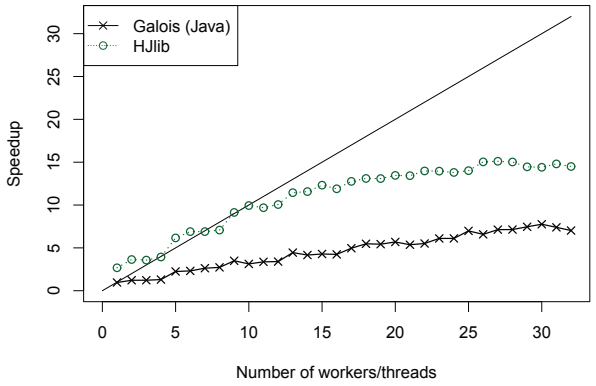
Compared with Galois, our HJlib version had shorter execution times—44.5-79.7% shorter than the execution times of Galois, and differences were more obvious under small number of workers. The reasons for the shorter execution times are twofold. First, the runtime overhead of task management inside HJlib is lower than that in the Galois system. Second, the optimizations in our logic circuit simulation implementation mentioned in Section 4.5 lead to additional savings in execution time. Avoiding the necessity of using priority queues, `java.util.PriorityQueue`, which were used in the Galois-Java version, and replacing them with `java.util.ArrayDeque`, the more lightweight array deques, in the implementation of event queues on the nodes helped reduce the execution time by nearly 50%. As the number of workers increased, the synchronization and communication overhead among different cores and physical processors in the hardware contributed more and more to the overall execution time, decreasing the parallel efficiency and the differences in execution time between the two versions. Also, the synchronization/communication overhead limited the increase of speedup when the number of workers is greater than 14. Another factor that limits the increase of parallel efficiency is the dynamics of the available parallelism, which is mentioned in Section 2.2 and Figure 1. We observe that different scalability results may be obtained for different circuits, since some circuits lend themselves to creating more events in parallel than others.

6. CONCLUSIONS AND FUTURE WORK

In this work, we parallelized a DES program that modeled logic circuit simulation using HJlib. We presented the ways we applied the task creation and synchronization functions in HJlib to the logic circuit simulation, showing the simplicity of parallelizing sequential programs. Compared with Galois, HJlib brings more programmer friendliness since users do not need to handle those performance oriented hints and annotations regarding task scheduling policies which are managed by HJlib's runtime implicitly. The work-stealing and load balancing features inside HJlib gave us efficient task scheduling and resource management, which brought good performance to our parallel implementation of the logic circuit simulation. In the performance evaluation, we compared our HJlib version of parallel logic circuit simulation with the Galois-Java version and demonstrated that our HJlib version reduced the execution time by 44.5-79.7%.

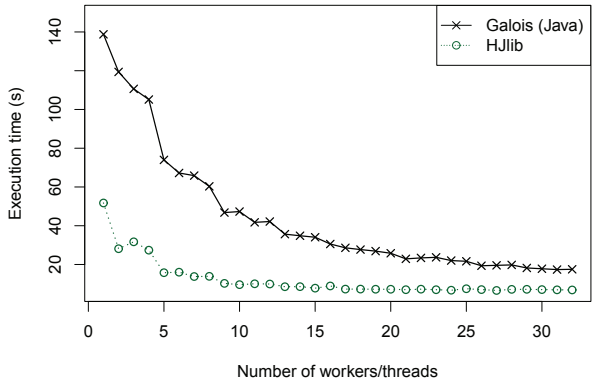


(a) Minimum execution time as a function of number of workers for Galois and HJlib versions

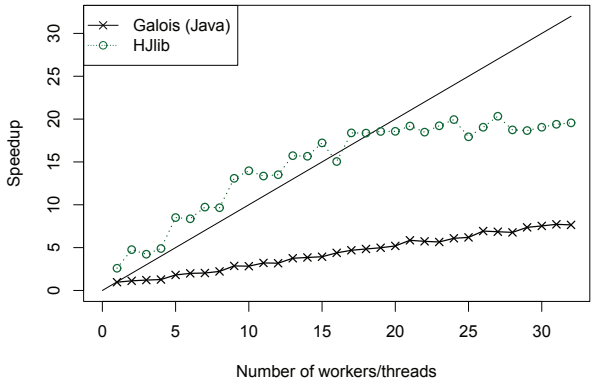


(b) Speedup of Galois and HJlib versions, relative to sequential Galois implementation

Figure 4: Performance for the 12-bit tree multiplier circuit



(a) Minimum execution time as a function of number of workers for Galois and HJlib versions



(b) Speedup of Galois and HJlib versions, relative to sequential Galois implementation

Figure 5: Performance for the 64-bit Kogge-Stone tree adder circuit

We believe that exploring larger-scale DES application, such as wireless mobile ad hoc network simulation, with Java and HJlib is an interesting approach, and we will continue on this direction in the future. In next step, we will break down and study the impact of the HJlib runtime and the optimizations introduced in Section 4.5), and investigate the generality. Another direction we are going to explore is the use of HJlib actor model [14] for parallelizing DES applications.

7. REFERENCES

[1] Galois project website. `http://iss.ices.utexas.edu/?p=projects/galois`.

[2] ns-3. `http://www.nsnam.org`.

[3] D. Bauer and E. Page. Optimistic parallel discrete event simulation of the event-based transmission line matrix method. In *Simulation Conference, 2007 Winter*, pages 676–684, Dec 2007.

[4] W. E. Biles, C. M. Daniels, and T. J. O'Donnell. Statistical considerations in simulation on a network of microcomputers. In *Proceedings of the 17th conference on Winter simulation*, pages 388–393. ACM, 1985.

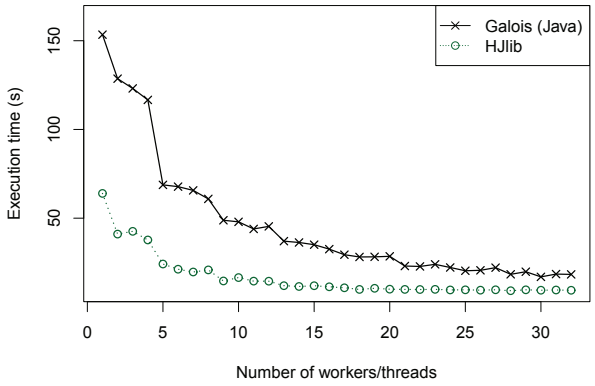
[5] R. E. Bryant. A switch-level model and simulator for mos digital systems. *Computers, IEEE Transactions on*, 100(2):160–177, 1984.

[6] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *Software Engineering, IEEE Transactions on*, (5):440–452, 1979.

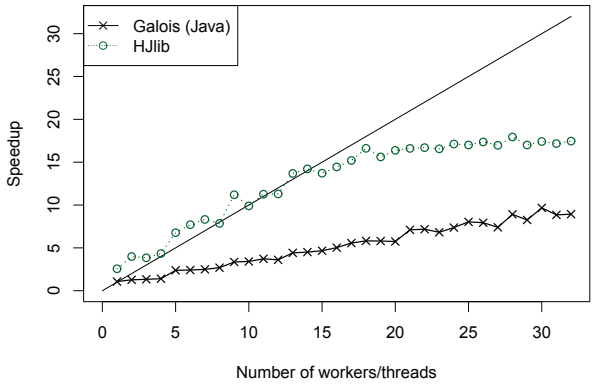
[7] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–206, 1981.

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

[9] W. Chen, X. Han, C.-W. Chang, G. Liu, and R. Domer. Out-of-order parallel discrete event



(a) Minimum execution time as a function of number of workers for Galois and HJlib versions



(b) Speedup of Galois and HJlib versions, relative to sequential Galois implementation

Figure 6: Performance for the 128-bit Kogge-Stone tree adder circuit

simulation for transaction level models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 33(12):1859–1872, Dec 2014.

[10] R. Curry, C. Kiddle, R. Simmonds, and B. Unger. Sequential performance of asynchronous conservative pdes algorithms. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, PADS '05, pages 217–226, Washington, DC, USA, 2005. IEEE Computer Society.

[11] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[12] M. A. Hassaan, M. Burtscher, and K. Pingali. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. *ACM SIGPLAN Notices*, 46(8):3–12, 2011.

[13] S. Imam and V. Sarkar. Habanero-java library: A java 8 framework for multicore programming. In *11th International Conference on the Principles and Practice of Programming on the Java Platform, PPPJ*, volume 14, 2014.

[14] S. M. Imam and V. Sarkar. Integrating task parallelism with actors. In *ACM SIGPLAN Notices*, volume 47, pages 753–772. ACM, 2012.

[15] D. Jefferson and H. Sowizral. *Fast Concurrent Simulation Using the Time Warp Mechanism: Part I, Local Control*. Rand Corporation, 1982.

[16] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.

[17] D. W. Jones. Concurrent simulation: an alternative to distributed simulation. In *Proceedings of the 18th conference on Winter simulation*, pages 417–423. ACM, 1986.

[18] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *Computers, IEEE Transactions on*, 100(8):786–793, 1973.

[19] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *ACM SIGPLAN Notices*, volume 42, pages 211–222. ACM, 2007.

[20] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *ACM Sigplan Notices*, volume 45, pages 3–14. ACM, 2010.

[21] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys (CSUR)*, 18(1):39–65, 1986.

[22] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, et al. The tao of parallelism in algorithms. *ACM SIGPLAN Notices*, 46(6):12–25, 2011.

[23] R. Righter and J. C. Walrand. Distributed simulation of discrete event systems. *Proceedings of the IEEE*, 77(1):99–113, 1989.

[24] Y. Tang, K. S. Perumalla, R. M. Fujimoto, H. Karimabadi, J. Driscoll, and Y. Omelchenko. Optimistic simulations of physical systems using reverse computation. *Simulation*, 82(1):61–73, 2006.

[25] S. Thulasidasan, S. P. Kasiviswanathan, S. Eidenbenz, and P. Romero. Explicit spatial scattering for load balancing in conservatively synchronized parallel discrete event simulations. In *Proceedings of the 2010 IEEE Workshop on Principles of Advanced and Distributed Simulation*, PADS '10, pages 150–158, Washington, DC, USA, 2010. IEEE Computer Society.

[26] V.-Y. Vee and W.-J. Hsu. Parallel discrete event simulation: A survey. Technical report, Technical report, Centre for Advanced Information Systems, Nanyang Technological University, Singapore, 1999.

[27] G. Yaun, C. D. Carothers, and S. Kalyanaraman. Large-scale tcp models using optimistic parallel simulation. In *Proceedings of the seventeenth workshop on Parallel and distributed simulation*, page 153. IEEE Computer Society, 2003.