Improved Distributed Simulation of Sensor Networks Based on Sensor Node Sleep Time

Zhong-Yi Jin and Rajesh Gupta

Dept. of Computer Science & Eng, UCSD
{zhjin,rgupta}@cs.ucsd.edu

Abstract. Sensor network simulators are important tools for the design, implementation and evaluation of wireless sensor networks. Due to the large computational requirements necessary for simulating wireless sensor networks with high fidelity, many wireless sensor network simulators, especially the cycle accurate ones, employ distributed simulation techniques to leverage the combined resources of multiple processors or computers. However, the large overheads in synchronizing sensor nodes during distributed simulations of sensor networks result in a significant increase in simulation time. In this paper, we present a novel technique that could significantly reduce such overheads by minimizing the number of sensor node synchronizations during simulations. We implement this technique in Avrora, a widely used parallel sensor network simulator, and achieve a speedup of up to 11 times in terms of average simulation speed in our test cases. For applications that have lower duty cycles, the speedups are even greater since the performance gains are proportional to the sleep times of the sensor nodes.

1 Introduction

Simulations of wireless sensor networks (WSNs) are important to provide controlled and accessible environments for developing, debugging and evaluating WSN applications. In simulations, sensor network programs run on top of simulated sensor nodes inside simulated environments. Since the simulated entities (simulation models) are transparent to simulation users, the states and interactions of sensor network programs can be inspected and studied easily and repeatedly. In addition, the properties of the simulated entities such as the locations of sensor nodes and the inputs to the sensor nodes can be changed conveniently before or during simulations.

One of the key requirements for simulating WSNs is high simulation fidelity in terms of temporal accuracy of events and actions. High fidelity simulations usually come at the cost of increased simulation time and poor scalability because significant computational resources are required for running high fidelity simulation models. Parallel and distributed simulators [1] are developed to address these issues by leveraging the combined resources of multiple processors/cores on a same computer and on a network of computers, respectively. They can significantly improve simulation speed and scalability because sensor nodes may be

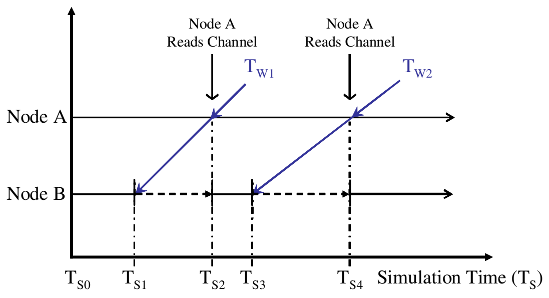

Fig. 1. The progress of simulating in parallel a wireless sensor network with two nodes that are in direct communication range of each other

simulated in parallel on different processors or computers. Since different nodes may get simulated at different speeds in this fashion, simulated nodes often need to synchronize with each other to preserve causalities and ensure correct simulation results.

Synchronizations of sensor nodes are illustrated in Figure 1 which shows the progress of simulating in parallel two sensor nodes that are within direct communication range of each other. There are two notions of time in the figure, wallclock time T_W and simulation time T_S. Wallclock time corresponds to the actual physical time while simulation time is the virtual clock time that represents the physical clock time of real sensor nodes in simulations [1]. At wallclock time T_{W1}, the simulation time of Node A (T_{SA}) is T_{S2} and the simulation time of Node B (T_{SB}) is T_{S1}. Node B is simulated slower than Node A in this case ($T_{SB} < T_{SA}$) because the thread or process that is used to simulate Node B either runs on a slower processor or receives fewer CPU cycles from an operating system (OS) task scheduler. At T_{S2}, Node A is supposed to read the wireless channel and continue its execution along different execution paths based on whether there are active wireless transmissions or not. However, despite the fact that the simulation time of Node A already reaches T_{S2} at T_{W1}, Node A probably can not advance any further because at T_{W1} Node A does not know whether Node B is going to transmit at T_{S2} or not (since $T_{SB} < T_{S2}$). There are two general approaches to handle cases like this, a conservative one, e.g., [2] and an optimistic one, e.g., [3]. With the conservative approach, Node A has to wait at T_{S2} until the simulation time of Node B reaches T_{S2}. The optimistic approach, on the other hand, would allow Node A to advance assuming there will be no transmissions from Node B at T_{S2}. However, the entire simulation state of Node A at T_{S2} has to be saved. If Node A later detects that Node B actually transmits to Node A at T_{S2}, it can correct the mistake by rolling back to the saved state and start again.

Almost all distributed WSN simulators are based on the conservative approach as it is simpler to implement and has a lower memory footprint. The conservative approach mainly involves three time-consuming steps. In the first step, the threads or processes that simulate the waiting nodes (Node A in Figure 1), have to be suspended. This would usually involve context switches to swap in other threads or processes that simulate other nodes. In the second step, the nodes that some other nodes are waiting for (Node B in Figure 1) have to notify the waiting nodes about their progresses. For example, in Figure 1, Node B must notify Node A after it advances past T_{S1} so that Node A can continue. In the last step, the waiting nodes need to be swapped back into execution once they are notified to continue. Step 1 and 3 are time-consuming as they involve context switches. Large numbers of context switches would significantly increase simulation time. Step 2 involves expensive inter-thread or inter-process communications and is generally the slowest step in distributed simulations as the notifications may have to be sent through slow networks to different computers.

Therefore, the performance gains in existing distributed WSN simulators are often compromised by the rising overheads due to inter-node synchronizations. This is reported in both Avrora [4], a cycle accurate simulator that runs over SMP (shared memory multiprocessor) computers, and DiSenS [5], a cycle accurate simulator for simulations over both SMP computers and a network of computers. In the case of Avrora, the reported time of simulating 32 nodes with 8 processors is only about 15% less than using 4 processors because of a large number of thread context switches introduced by synchronizations. In DiSenS, it is actually faster to simulate 4 nodes using 1 computer (dual-processor) than using 2 computers and simulating 2 nodes on each in most of the testing cases. This sub-linear performance in DiSenS is due to the large communication overheads in synchronizing nodes that are simulated on different computers.

In this paper, we describe a novel technique that could significantly reduce the number of synchronizations in distributed simulations of WSNs. Our technique exploits the sleep-often property of sensor network applications and uses sleep times to reduce sensor node synchronizations. It works without any prior knowledge of the sensor network applications under simulations and the number of reductions scale with both network sizes and sleep times. We demonstrate the value of our approach by its implementation in Avrora [4].

We describe our speedup technique in Section 2. Its implementation is presented in Section 3. In Section 4 we describe the results of our experiments followed by a brief overview of the related work and conclusion in Section 5 and Section 6.

2 A Novel Technique to Reduce Synchronizations in Distributed WSN Simulations

Sensor node synchronizations are required for enforcing dependencies between sensor nodes in simulations. Since the dependencies come from the interactions of sensor nodes over wireless channels, the number of required synchronizations in a distributed simulation is inversely proportional to the degree of parallelism in the WSN application under simulation [1].

To reduce the number of synchronizations, we exploit the parallelism available in WSN applications. In particular, we seek to use the information regarding

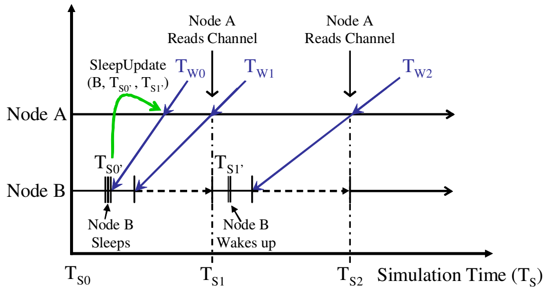

Fig. 2. The progress of simulating in parallel a wireless sensor network with two duty cycled nodes that are in direct communication range of each other

duty cycling of nodes in sensor networks to speed up simulations. Node duty cycling is common in WSN applications for power management purposes. Most WSN applications have very low duty cycles and need to carefully manage their power consumptions in order to function for an extended period of time under the constraint of limited energy supply. In other words, sensor nodes sleep most of the time and do not interact with each other frequently until certain events are detected [6]. Very little power is consumed by a sensor node in the sleep state as its wireless radio is turned off and its processor is put into a low power sleep mode.

Our speedup technique is illustrated in Figure 2 which shows the progress of simulating two duty cycled sensor nodes that are within direct communication range of each other. In the simulation, Node B enters into the sleep state at $T_{S0'}$ and wakes up at $T_{S1'}$. With existing distributed WSN simulators, Node A needs to wait for Node B at T_{S1} although Node B does not transmit anything during its sleep period. To eliminate this type of unnecessary synchronization, our technique keeps track of the time that a node enters into the sleep state and the time it wakes up. When we detect during a simulation that a node is entering into the sleep state, we immediately send both the entering time and exiting time (simulation time) in a *SleepUpdate* message to the neighboring nodes that are within direct communication range. As a result, neighboring nodes no longer need to synchronize with the sleeping node during the sleep period. For example, when we detect that Node B is entering into the sleep state at $T_{S0'}$, we immediately notify Node A that Node B will be in the sleep state from $T_{S0'}$ to $T_{S1'}$. Once Node A knows that Node B will not transmit between $T_{S0'}$ and $T_{S1'}$ and $T_{S0'} \leq T_{S1} < T_{S1'}$, it no longer needs to wait for Node B at T_{S1} and its lookahead time increases. Lookahead time is defined as the amount of simulation time that a simulated sensor node can advance freely without waiting for inputs from other simulated sensor nodes [1]. The speedup of our technique increases with the durations of sleep periods because the longer the sleep periods, the larger the lookahead time.

For the speedup technique to work, we have to be able to detect both sleep time and wakeup time in simulations. Sleep time can always be detected because a real sensor node processor has to execute some special instructions to put the node into the sleep mode. Correspondingly, sleep events, which are integral parts of any WSN simulators, are used in simulations to signal the transitions of nodes from active states to the sleep state. For example, in the case of cycle accurate sensor network simulators, sleep events are associated with the execution of specific machine instructions of the sensor node processors under simulation. However, detecting wakeup time is a challenging process as a node can be woken up by interrupts from either a timer or an external autonomous sensor such as a passive infrared sensor. Autonomous sensors are devices that can function independently without the support of a node processor and therefore may wakeup a sensor node at any time.

The wakeup time for sensor nodes that do not have autonomous sensors can always be detected. This is because the wakeup time has to be passed to a physical timer before a real sensor node is able to enter into the sleep state as the node processor can not execute any instructions during the sleep mode. For nodes equipped with autonomous sensors, if the input events to the autonomous sensors are known before a simulation starts, which is generally the case, the wakeup time can always be computed as the smaller of the time of the timers and the input events. If the input events to the autonomous sensors are not known before a node enters into the sleep mode, for example, inputs are collected in real time from real sensors [7], then the speedup technique has to be disabled on that node. However, we only need to turn off the speedup technique on those nodes receiving unpredictable input events, while other nodes can still use the technique.

3 Synchronization Algorithm and Implementation

We use Avrora [4] to evaluate the effectiveness of our approach. Avrora is a widely used cycle accurate sensor network simulator. Among all types of sensor network simulators, cycle accurate sensor network simulators [8,4,5] offer the highest level of fidelity. They provide simulation models that emulate the functions of major hardware components of a sensor node, mainly the processor. Therefore, one can run on top of them, clock cycle by clock cycle, instruction by instruction, the same binary code (images) that are executed by real sensor nodes. As a result, accurate timing and interactive behaviors of sensor network applications can be studied in details.

Avrora is written in Java and supports parallel simulations of sensor networks comprised of Mica2 Motes [9]. It allocates one thread for each simulated node and relies on the Java virtual machine (VM) to assign runnable threads to any available processors on an SMP computer. Implementing the speedup technique in Avrora mainly involves developing new code in two areas: synchronization algorithm and channel modeling. Our implementation is based on the Beta 1.6.0 code release of Avrora. Its well documented source code is publicly available.

3.1 Synchronization Algorithm

The lock-step style synchronization algorithm of Avrora is optimized for parallel simulations on SMP computers but lacks necessary features to support our

Algorithm 1. Distributed Synchronization Algorithm with Speedup Technique

```
Require: $nl := \{< nid, nclock >\}$ /*a list of neighboring node ids and their reported simulation time*/
Require: $id$ /*current node ID*/, $bytetime$ /*the amount of time to transmit one byte with a wireless radio*/
 1: $clock \Leftarrow 0$ /*current sim clock to 0*/, $lookahead \Leftarrow bytetime$, $intervalclock \Leftarrow 0$

 2: for every tuple $< nid, nclock >$ in $nl$ do
 3:     $nclock \Leftarrow 0$ /*initialize simulation time of neighboring nodes to zero before starting the simulation*/
 4: while $clock \leq$ user inputed simulation time do
 5:     $waitchannel \Leftarrow false$
 6:     execute $next\ instruction$
 7:     if the $instruction$ puts a node into the sleep state then
 8:         $exitclock = wakeuptime$, $intervalclock \Leftarrow exitclock$
 9:         send a $ClockUpdate(id, exitclock)$ message to every $nid$ node in the tuple $< nid, nclock >$ of $nl$
10:     else if the $instruction$ reads from the wireless radio then
11:         if $lookahead \geq 0$ then
12:             read the wireless radio
13:         else
14:             if $intervalclock \neq clock$ then
15:                 $intervalclock \Leftarrow clock$
16:                 send a $ClockUpdate(id, clock)$ message to every $nid$ node in the tuple $< nid, nclock >$ of $nl$
17:             $waitchannel \Leftarrow true$
18:     if $waitchannel$ is $false$ then
19:         $clock \Leftarrow clock + cyclesconsumed$ /*advance clock by the clock cycles of the executed instruction*/
20:     $updated \Leftarrow false$ /*check incoming $ClockUpdate$ messages at least once per instruction*/
21:     repeat
22:         for each received $ClockUpdate(cid, cclock)$ message do
23:             for every tuple $< nid, nclock >$ in $nl$ do
24:                 if $uid$ equals to $cid$ then
25:                     $nclock \Leftarrow cclock$
26:         $updated \Leftarrow true$
27:         $minclock = min(nclock)$ in the tuple $< nid, nclock >$ of $nl$ /*find the neighbor with the smallest clock*/
28:         $lookahead \Leftarrow (minclock - floor(clock/bytetime) * bytetime)$ /*on byte boundaries with byte-radio*/
29:         if $lookahead \geq 0$ and $waitchannel$ is $true$ then
30:             read the wireless radio /*all neighbors have advanced past the byte boundary this node is waiting on*/
31:             $clock \Leftarrow clock + cyclesconsumed$, $waitchannel \Leftarrow false$
32:     until $waitchannel$ is $false$ and $update$ is $true$
33:     if $(clock - intervalclock) \geq bytetime$ then
34:         $intervalclock \Leftarrow clock$
35:         send a $ClockUpdate(id, clock)$ message to every $nid$ node in the tuple $<$ $nid, nclock >$ of $nl$
```

speedup technique. Our distributed synchronization algorithm is shown in Algorithm 1. It is a generic distributed synchronization algorithm similar to the one in DiSenS [5] and is suitable for both parallel and distributed simulations of WSNs.

Synchronizations are only necessary between neighboring nodes that are within direct communication range of each other. The first step before applying our algorithm is to build a neighbor node list for each node according to the locations of the sensor nodes and the maximum transmission range of their wireless radios. Mobile nodes need to be included in the neighbor node list of all other nodes. Then, a time stamp is assigned to every node (node id) in the lists to keep the last reported simulation time of that node. This list, named *nl* in Algorithm 1 is the first required input to the synchronization algorithm. There are two more inputs to the algorithm. The second input *id* is used to identify the node under simulation. The nodes in *nl* are neighbors of this node. The third input *bytetime* is the amount of time to transmit one byte with a wireless radio. It is the maximum lookahead time without synchronizations. Every node starts with that lookahead time because it takes that amount of time for one byte of data to travel from the sender to the receiver. For example, if a node starts at simulation time 0 and wants to read the wireless channel at that time, it can do so because the earliest time that a byte of data can arrive is $0 + bytetime$. Similarly, after synchronizing at time T_S, all synchronized nodes can advance freely up to $T_S + bytetime$ without any additional synchronizations. However, this approach only works if the processor and radio on a real sensor node communicate by exchanging data one byte at a time (*byte-level*).

The variable *intervalclock* in Algorithm 1 is used to ensure that *ClockUpdate* messages are sent by every simulated node once every *bytetime* if the node is not already in the sleep state. These messages update neighboring nodes about the latest simulation time of the sender and ensure neighboring nodes have the right time information to make synchronization decisions according to Condition 1. The interval chosen to send the messages will affect the performance of the algorithm as nodes may have to wait if *ClockUpdate* messages are delayed. The smallest interval one can use with *byte-level* radios is *bytetime* because the actual waiting time must fall on byte boundaries. This can be seen in Algorithm 1 where the *floor* function is used to calculate the lookahead time.

Condition 1. *If a node N_i reads data sent by a node N_s over a wireless channel C_k at simulation time T_{SN_i}, then the simulation time of node N_s, T_{SN_s}, must be greater than or equal to T_{SN_i}.*

The *SleepUpdate* message described in Section 2 is replaced in Algorithm 1 with the *ClockUpdate* message because receiving nodes only need to use the wakeup time of the sender to calculate lookahead time. However, to take advantage of a special optimization described in the future work part of Section 6, *SleepUpdate* messages must be used so the time that a node enters into the sleep state is sent as well. The time that a node enters into the sleep state is detected when the *Sleep* instruction of the ATMega128L microcontroller [10] is executed. This instruction can put the microcontroller into different sleep modes based on the values set in the *MCU Control Register*. It is critical to read that register before sending any *ClockUpdate* messages as the speedup technique only works if a microcontroller is put into *Power-Save Mode*. The reason is that the only way to wake up a microcontroller from *Power-Save Mode* is through interrupts generated by timers or external autonomous sensors. Because of that, we can find the wakeup time by keeping track of the values written to the *Timer Control Register* and using the techniques described in section 2.

The computational complexity of a synchronization algorithm is determined by the total number of synchronization messages that need to be sent and the overheads in sending and processing each of the messages [11]. Our synchronization algorithm has higher computational complexity than the one in Avrora because we choose to implement it as a generic distributed algorithm. In Avrora, a global data structure is used to keep track of the simulation time of each node and therefore a node only needs to update the global data structure once for each clock update. This centralized approach is optimized for parallel simulation over SMP computers but does not support distributed simulations over a network of computers. We implement our synchronization algorithm as a truly distributed algorithm by distributing parts of the global data structure to each node. The penalty is that a node with N nodes in direct communication range has to send a total of N messages for each clock update. However, the penalty is not significant when the number of nodes within direct communication range is not big. In fact, because the synchronization algorithm of Avora works by synchronizing a node with all other nodes regardless of whether they are within communication range or not, our distributed algorithm may even perform better when nodes are sparsely distributed. If performance is an issue in the future, we could choose to optimize our implementation for parallel simulations using a centralized approach.

3.2 Channel Modeling

With our synchronization algorithm, a node can write to a wireless channel long before the packets are read by other nodes (the transmitting code is omitted in Algorithm 1 for simplicity). Because of that, we develop a new wireless channel model that uses a circular buffer to store unread wireless packets. Our channel model uses a similar method as the original channel model in Avrora to map transmitting and receiving time into time slots that are *bytetime* apart. The slot number is used to index into the circular buffer. Note that with the original channel model, a write to a channel right after synchronizations could be dropped as the write may happen before the time slots are carried forward. Our channel implementation does not drop data.

4 Evaluation

We conduct a series of experiments to evaluate the performance of our speedup technique. The experiments are conducted on an SMP server running Linux

2.6.9. It has 4 processors (Intel Xeon 2.8GHz) and 2GByte of RAM. For comparison, the same test cases are simulated using both our modified Avrora and the original Avrora. Sun's Java 1.6.0 is used to run both simulators.

To demonstrate the effectiveness of our speedup technique, we choose two programs from the CountSleepRadio example which is a part of the TinyOS 1.1 distribution [12,13]. These two programs behave exactly like the CntToRfm and RfmToLeds programs used in the experiments of the Avrora paper and serve similar purposes. The only difference is that the new programs can put sensor nodes into sleep states to save power. (The latest TinyOS 2.0 release [14] is not used for our experiments because its radio stack is not fully compatible with the radio model in the version of the Avrora that our code is based on.)

The first program we use for our experiments is CountSleepRadio. It wakes up a node periodically from the sleep state to increase the value of a counter by one and broadcast that value in a packet. Once the packet is sent, it puts the node back into the sleep state to save power. We have to modify this program for some of our experiments because the original program has an upper bound on how long a sensor node can stay in the sleep state[1]. This is unnecessary and we work around this limitation by using the *Clock* interface of the TinyOS 1.1 directly. The problem has reportedly been fixed in TinyOS 2.0. The counterpart of our CountSleepRadio program is CountReceive. It receives packets sent by CountSleepRadio and flashes different LEDs based on the values in the packets. For simplicity, we identify nodes running CountSleepRadio and CountReceive as senders and receivers respectively in the following sections.

Both CountSleepRadio and CountReceive use the default TinyOS 1.1 CC1000 CSMA (carry sense multiple access) MAC (media access control) which is based on B-MAC [15]. Before sending a packet, the CC1000 MAC first backs off for a random amount of time and then reads its transmitting channel for ongoing transmissions. It only sends the packet if the channel is clear. Otherwise, it backs off for a random amount of time before checking the channel again. As a result, a sender in our experiments reads the wireless channel at least once before each transmission.

4.1 Performance in One-Hop Networks

In this section, the performance of our speedup technique is evaluated under various sleep times and network sizes using one-hop sensor networks. One-hop sensor networks are sensor networks set up in such a way that all sensor nodes are within direct communication range of each other. It is a common form of sensor network used in actual deployments [6].

In the one-hop sensor network experiments, nodes are laid on a 10 by 10 grid 1 meter apart and their maximum transmission ranges are set to 20 meters. A fixed node is selected as a receiver and the rest as senders. The receiver listens continuously like a gateway node [6,16] and does not enter into the sleep state. The senders are duty cycled and their sleep durations are varied for different

[1] The upper bound is imposed by the timer implementation of TinyOS 1.1. The timer code sets $maxTimerInteval$ to $230ms$ and the physical timers of a real sensor node can not be set to anything larger than that using the timer API.

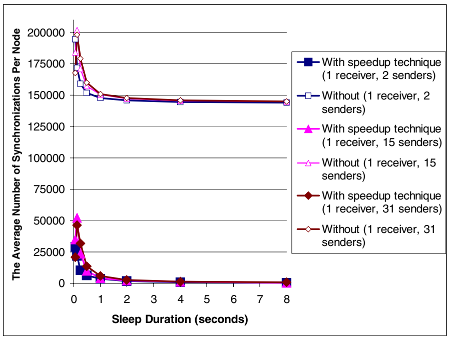

Fig. 3. Average number of synchronizations per node in one-hop networks during 60 seconds of simulation time

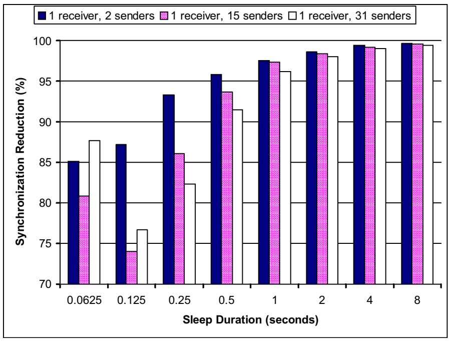

Fig. 4. Percentage reductions of the average number of synchronizations per node in one-hop networks during 60 seconds of simulation time

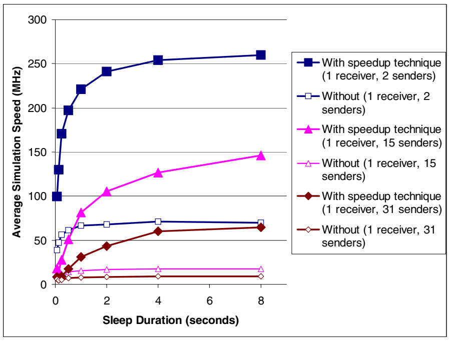

Fig. 5. Average simulation speed in one-hop networks

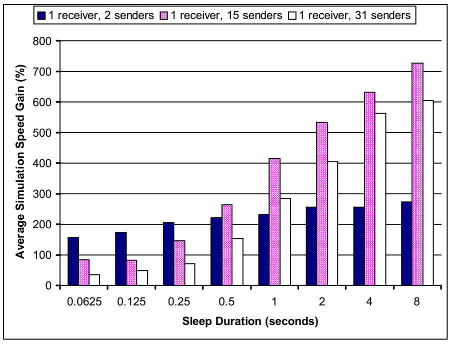

Fig. 6. Percentage increases of average simulation speed in one-hop networks

experiments. Sleep duration is how long a node stays in the sleep state before waking up. All results in this section are averages of three runs.

Figure 3 shows the average number of synchronizations per node in one-hop networks during 60 seconds of simulation time. Since all nodes are simulated for the same number of clock cycles (60 $\times$ clock frequency of ATMega128L) in all test cases, the average number of synchronizations per node is a good indicator to the performance of the speedup technique. The synchronization numbers are collected by logging code we add specifically for evaluation purposes. Figure 4 shows the percentage reductions of the average number of synchronizations per node in one-hop networks during the 60 seconds of simulation time. Figure 5 shows average simulation speed in one-hop networks. The average simulation speed V_{avg} is calculated using Equation 1. The percentage increases of average simulation speed in one-hop networks are shown in Figure 6.

$$V_{avg} = \frac{total\ number\ of\ clock\ cycles\ executed\ by\ the\ sensor\ nodes}{(execution\ time\ of\ the\ simulation) \times (number\ of\ sensor\ nodes)} \quad (1)$$

As shown in Figure 3 and Figure 4, the speedup technique significantly reduces synchronizations in all the test cases and the largest percentage reduction is more than 99%. The reduction percentages increase with sleep durations under fixed network sizes except for the 16 (1 receiver, 15 senders) and 32 (1 receiver and 31 sender) node test cases with $62.5ms$ sleep durations. The unusually high percentage reductions in those cases are results of using the CC1000 CSMA MAC protocol of TinyOS 1.1. When multiple senders in communication range transmit at the same time, the MAC protocol would sequence their transmission times using random backoffs. Since the senders will not return back to the sleep state until packets are successfully transmitted, the sleep times of the senders are sequenced as well. This effectively reduces synchronizations in simulations as the number of nodes that are active at a same time is reduced. The 3-node (1 receiver, 2 senders) test case is not affected by this because we randomly delay the starting time of each node between 0 and 1 second in all our experiments to prevent the nodes from artificially starting at the same time. When the number of nodes in a one-hop network decreases, the chance for concurrent transmissions decreases and the number of synchronizations increases. Similarly, the chance for concurrent transmissions also decreases when the sleep duration increases because the nodes in a one-hop network would transmit less frequently with larger sleep durations. As a result, the number of synchronizations increases with sleep durations in such cases. We can see this from Figure 7, a zoomed in view of Figure 3. When sleep duration doubles from $62.5ms$ to $125ms$, the average number of synchronizations per node actually increases for both of the 16 and 32 node test cases, regardless of whether the speedup technique is used or not. We can also see in the same test cases that the average number of synchronizations per node decreases with network size under fixed sleep durations. The speedup technique can further reduce synchronizations in cases like these because it increases the lookahead time of simulated nodes. The reduction from applying the speedup technique is greater for larger one-hop networks in those cases as there is more of this type of sequencing in larger one-hop networks under fixed sleep durations.

As shown in Figure 5 and Figure 6, the speedup technique significantly increases average simulation speed in all the test cases and the largest increase is more than 700%. Although the 3-node speedup test case has the highest percentage reduction of the average number synchronizations per node as shown in Figure 4, it does not have the largest average simulation speed increase in Figure 6. This is because the overhead in performing a synchronization is very low for the 3-node test cases. Context switches are generally not needed for synchronizations in those cases because there are more processors (4) than nodes/threads (3). We can also see in Figure 6 that the growth of the average simulation speed quickly flattens out for large sleep durations in original Avrora but continues after applying the speedup technique. We expect even better average simulation speed

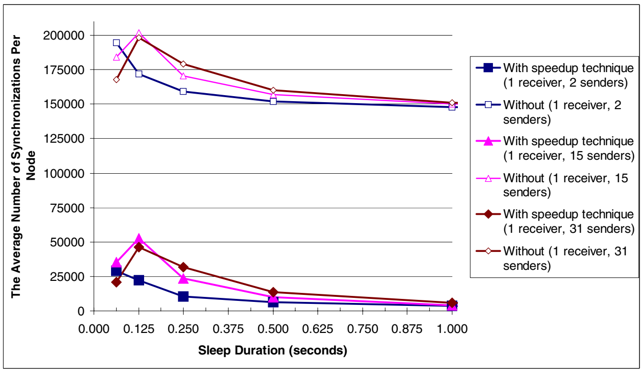

Fig. 7. Average number of synchronizations per node in one-hop networks during 60 seconds of simulation time (a zoomed in view of Figure 3)

in simulating large one-hop networks after optimizing our generic distributed synchronization algorithm for parallel simulations as discussed in Section 3.1.

The speedup technique can not completely eliminate synchronizations caused by having only limited numbers of physical processors available for simulations. We can see this from Figure 3 and Figure 7. When the sleep duration is long enough, the average number of synchronization per node with speedup is similar for all network sizes.

4.2 Performance in Multi-hop Networks

In this section, we evaluate the performance of the speedup technique using multi-hop sensor networks. Nodes are laid 20 meters apart on square grids of various sizes. Sender and receivers are positioned on the grids in such a way that nodes of the same types are not adjacent to each other. By setting a maximum transmission range of 20 meters, this setup ensures that only neighboring nodes are within direct communication range of each other. This configuration is very similar to the two dimensional topology in DiSenS [5]. Once again, only senders are duty cycled to keep the experiments simple.

Figure 8 shows the average number of synchronizations per node in multi-hop networks during 20 seconds of simulation time. The percentage reductions of the average number of synchronizations per node in multi-hop networks during the 20 seconds of simulation time is shown in Figure 9. We can see that there are significant reductions in the average number of synchronizations per node in all the test cases using the speedup technique and the reduction percentages scale with sleep durations.

Figure 10 and Figure 11 indicate that the speedup technique significantly increases average simulation speed in all multi-hop test cases. Compared to the one-hop test results in Figure 6, the speed increases scale better with network sizes in multi-hop tests. This is because our distributed synchronization

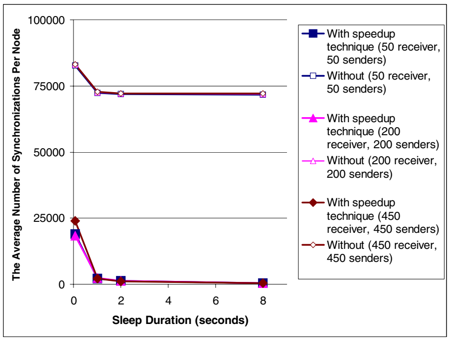

Fig. 8. Average number of synchronizations per node in multi-hop networks during 20 seconds of simulation time

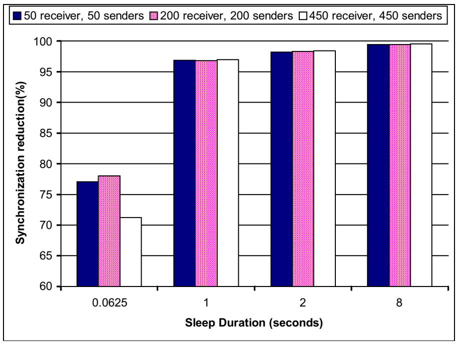

Fig. 9. Percentage reductions of the average number of synchronizations per node in multi-hop networks during 20 seconds of simulation time

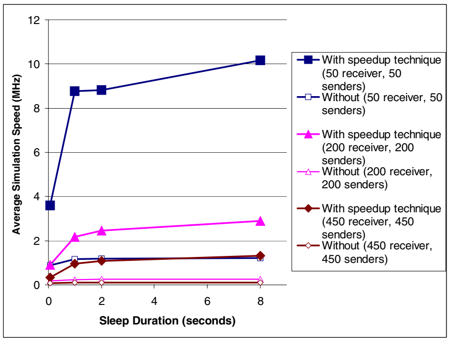

Fig. 10. Average simulation speed in multi-hop networks

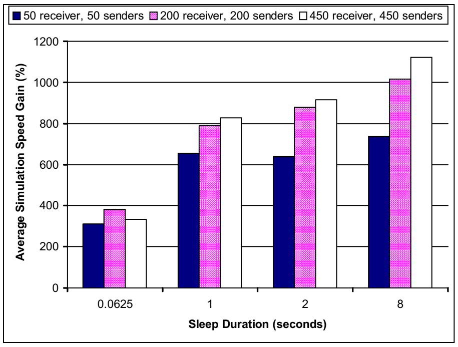

Fig. 11. Percentage increases of average simulation speed in multi-hop networks

algorithm has less overhead on sensor networks that have smaller numbers of nodes within direct communication range as described in the end of Section 3.1.

5 Related Work

There is a large body of work on improving the scalability of distributed discrete event driven simulators in general. This work can be classified into two groups, those based on conservative synchronization algorithms [2] and those based on optimistic synchronization algorithms [3]. The performance of conservative approaches is bounded by worse case scenarios. The optimistic approaches do not have this limitation but they are usually very complex for implementation and require a large amount of memory to run.

Exploiting lookahead time is a very common conservative approach to improve the scalability of distributed discrete event driven simulators [17,18]. Our approach is similar to those in the sense that we also improve scalability of distributed discrete event driven simulators by increasing lookahead time. However, our technique is fundamentally different as we use different and application specific characteristics in a different context to increase lookahead time.

6 Conclusion and Future Work

We have described a speedup technique that significantly reduces sensor node synchronizations in distributed simulations of sensor networks and consequently improves average simulation speed and scalability of distributed sensor network simulators. We implemented this technique in Avrora, a widely used parallel sensor network simulator and conducted extensive experiments. The significant performance improvements with parallel simulations suggest even greater benefits in applying our technique to distributed simulations over a network of computers because of their large overheads in sending synchronization messages across computers during simulations.

As future work, we plan to merge our implementation into the latest development branch of Avrora. This would make it possible to simulate TinyOS 2.0 based applications with our speedup technique. We also plan to support an optimization that can reduce communication overheads in distributed simulations based on the speedup technique. When a sensor node is in the sleep state, its radio is off and it will not access the wireless channel at all. In other words, when a transmitting node knows that a receiving node is in the sleep state during a simulation, it no longer needs to send packets to the receiver for the entire sleep period. If the sender and receiver are simulated on different computers, the savings in terms of communication time and network bandwidth consumptions could be significant.

References

1. Fujimoto, R.M.: Parallel and distributed simulation. In: WSC 1999: Proceedings of the 31st conference on Winter simulation, pp. 122–131. ACM Press, New York (1999)
2. Chandy, K.M., Misra, J.: Asynchronous distributed simulation via a sequence of parallel computations. Commun. ACM 24(4), 198–206 (1981)
3. Jefferson, D.R.: Virtual time. ACM Trans. Program. Lang. Syst. 7(3), 404–425 (1985)
4. Titzer, B.L., Lee, D.K., Palsberg, J.: Avrora: scalable sensor network simulation with precise timing. In: IPSN 2005: Proceedings of the 4th international symposium on Information processing in sensor networks, p. 67. IEEE Press, Piscataway (2005)
5. Wen, Y., Wolski, R., Moore, G.: Disens: scalable distributed sensor network simulation. In: PPoPP 2007: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 24–34. ACM Press, New York (2007)
6. Szewczyk, R., Polastre, J., Mainwaring, A.M., Culler, D.E.: Lessons from a sensor network expedition. In: EWSN, pp. 307–322 (2004)

7. Girod, L., Elson, J., Cerpa, A., Stathopoulos, T., Ramanathan, N., Estrin, D.: Emstar: a software environment for developing and deploying wireless sensor networks. In: Proceedings of the 2004 USENIX Technical Conference, Boston, MA (2004)
8. Polley, J., Blazakis, D., McGee, J., Rusk, D., Baras, J.: Atemu: a fine-grained sensor network simulator. In: IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, October 4-7, pp. 145–152 (2004)
9. MICA2 Datasheet, Crossbow (2008)
10. ATMega128L Datasheet, Atmel (2003)
11. Nicol, D.M.: Scalability, locality, partitioning and synchronization pdes. SIGSIM Simul. Dig. 28(1), 5–11 (1998)
12. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. SIGPLAN Not. 35(11), 93–104 (2000)
13. Tinyos 1.1.15. `http://www.tinyos.net/`
14. Levis, P., Gay, D., Handziski, V., Hauer, J.-H., Greenstein, B., Turon, M., Hui, J., Klues, K., Sharp, C., Szewczyk, R., Polastre, J., Buonadonna, P., Nachman, L., Tolle, G., Culler, D., Wolisz, A.: T2: A second generation os for embedded sensor networks, Telecommunication Networks Group, Technische Universität Berlin. Tech. Rep. TKN-05-007 (November 2005)
15. Polastre, J., Hill, J., Culler, D.: Versatile low power media access for wireless sensor networks. In: SenSys 2004: Proceedings of the 2nd international conference on Embedded networked sensor systems, pp. 95–107. ACM, New York (2004)
16. Jin, Z., Schurgers, C., Gupta, R.: An embedded platform with duty-cycled radio and processing subsystems for wireless sensor networks. In: International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS) (2007)
17. Filo, D., Ku, D.C., Micheli, G.D.: Optimizing the control-unit through the resynchronization of operations. Integr. VLSI J. 13(3), 231–258 (1992)
18. Liu, J., Nicol, D.M.: Lookahead revisited in wireless network simulations. In: PADS 2002: Proceedings of the sixteenth workshop on Parallel and distributed simulation, pp. 79–88. IEEE Computer Society, Washington, DC (2002)