A High Performance Simulation Engine for Large-Scale Cellular DEVS Models

Xiaolin Hu, B. P. Zeigler
Arizona Center for Integrative Modeling and Simulation
Electrical and Computer Engineering Department
University of Arizona
Tucson, AZ 85719
{huxl, zeigler}@ece.arizona.edu

Keywords: DEVS, Cellular Space Model, High Performance Simulation, Simulation Protocol, Data Structure

Abstract

This paper presents a high performance simulation engine for large-scale cellular DEVS models. Compared to the standard *coordinator*, this simulation engine speeds up the simulation from two sources. First, it only considers the active cells during simulation. This is based on the observation that usually only a small number of cells are active (performing state changing) at any given time, even though the total number of cells may be very large. Second, it implements a data structure that allows efficient search of the active cells which can be arbitrarily faster in cellular space models where the number of cells increases but the number of active cells remains the same. Performance analysis and two examples are provided to demonstrate the speedups of this new simulation engine as compared to the *coordinator*.

1. INTRODUCTION

The Discrete Event System Specification (DEVS) formalism [1] provides a means to specify a mathematical object called a system. It has been applied to model and simulate both discrete and continuous systems (e.g., see [2], [3], [4]). The use of discrete events, rather than time steps, as a basis for simulation has been shown to reduce computation time by orders of magnitude in many applications. For example, the work presented in [5] suggests that DEVS can offer significant performance advantages for simulation of continuous phenomena characterized by spatiotemporal heterogeneity. The cellular automata paradigm defines a grid of cells using discrete variables for time, space and system states [6]. The cells are updated according with a local rule function that uses a finite set of nearby cells (called the neighborhood of the cell). Cellular DEVS models have been developed to model and simulate various phenomena such as fire spreading [7, 8], traffic control [9], flow injection [10], etc.

The cellular approach divides the underline space (one dimension or multiple dimensions) into discrete cells. To accomplish detailed modeling of spatial dynamics, a large number of cells are typically employed. While the huge number of cells provides the capability to model a system in adequate detail, it also demands for high performance simulation techniques. To achieve high performance simulation of DEVS models, various techniques have been developed. Among them the most well known one is parallel or distributed simulation [11, 12, 17] where models are simulated by multiple processors. Besides that, there is also considerable research work on the implementation of simulation environments. One of such work is recently presented in [13], where the authors enhance the implementation by applying techniques such as precomputing message destinations, and using a priority queue to sort models to achieve performance improvement for the Joint MEASURE simulation environment developed at Lockheed Martin.

Unlike the work that focuses on the structure or implementation of the simulation environments, the work presented in this paper describes a simulation engine for large-scale cellular models. This simulation engine improves simulation performance from two sources that are based on two qualities of cellular space models. First, it considers only the active cells during simulation. This is based on the observation that usually only a small number of cells are active (performing state changing) at any given time, even though the total number of cells may be very large. This approach enhances simulation performance compared with simulations that are based on cellular automata in which all cells perform computations and message exchange at every time step. The same approach has been taken by [2] and [7]. Second, this simulation engine implements a data structure that allows efficient search of the active cells which can be arbitrarily faster in cellular space models where the number of cells increases but the number of active cells remains the same. The implemented data structure takes advantage of the fact that the active cells in a cell space are typically locally clustered. This is because cells are coupled to their neighbors so the state change of one cell will first directly affect its neighbors. Based on this observation, the new data structure retains cells' spatial information and thus utilizes the localized activities of cellular space models to increase simulation performance.

The following paper describes this new simulation engine and the data structure it employs. To set the stage, we first review the standard DEVS simulation protocol as implemented by the *coordinator* in DEVSJAVA [14]. Then we propose an improved simulation engine that is based on the standard *coordinator*. With this background, we proceed to describe the new simulation engine, *oneDCoord* as implemented in DEVSJAVA, and its *minSelTree* data structure. Finally, we analyze the performance of these simulation engines and present two examples to demonstrate the performance improvement of the new simulation engine as compared to the standard *coordinator*.

2. THE STANDARD DEVS SIMULATION PROTOCOL

In a DEVS-based simulation environment such as DEVSJAVA, a coordinator is assigned to a coupled model and

simulators are assigned to the components of this coupled model. Figure 1 shows the simulation of a coupled model with three components.

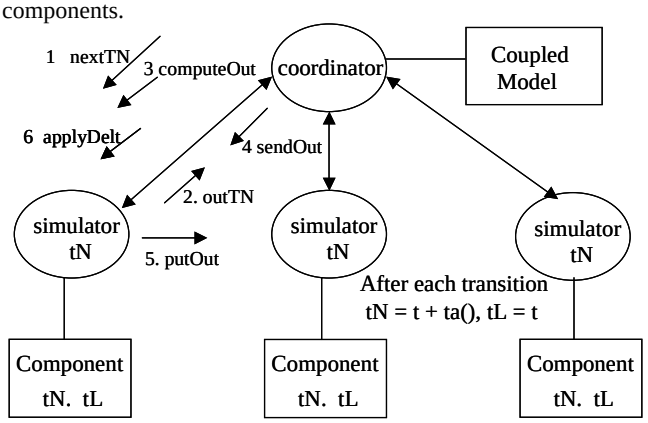

Figure 1: The Standard DEVS Simulation Protocol

The simulation of DEVS models moves forward cyclically based on the time of next event, denoted by *tN*, which is updated by component models' state transition functions. For a coupled model, the *tN* is the earliest next event time among all its sub-components. Each simulator has the *tN* of its model. In the standard DEVS simulation protocol, the coordinator is responsible for stepping simulators through the cycle of activities as shown in Figure 1. Specifically in DEVSJAVA simulation environment, the coordinator executes the following code in each simulation cycle:

```
simulators.AskAll("nextTN")
tN = compareAndFindTN();
simulators.tellAll("computeOut",tN)
simulators.tellAll("sendOut")
simulators.tellAll("ApplyDelt",tN)
```

A detailed description of how the coordinator and simulators step through a simulation cycle is given below:

1. Coordinator sends *nextTN* to request *tN* from each of the simulators.
2. All the simulators reply with their *tNs* in the *outTN* message to the coordinator. The coordinator compares these *tNs* and finds the minimum.
3. Coordinator sends to each simulator a *computeOut* message containing the global *tN* (the minimum of the *tNs*). Each simulator checks if it is imminent (its *tN* = global *tN*) and if so, computes the output message of its model. Otherwise an empty message is generated.
4. Coordinator sends to each simulator a *sendOut* message.
5. Based on the coupling specification, each simulator responses by putting its output message (if it is not empty) to the destination simulators.
6. Coordinator sends to each simulator an *applyDelt* message containing the global *tN*. Each simulator reacts to the *applyDelt* message as below:
   - If it is imminent and its input message is empty, then it invokes its model's internal transition function
   - If it is imminent and its input message is not empty, it invokes its model's confluence transition function
   - If is not imminent and its input message is not empty, it invokes its model's external transition function
   - If is not imminent and its input message is empty then nothing happens.

This standard simulation protocol follows closely with the semantic of DEVS models. Thus it is easy to be understood and implemented. However, this protocol tends to result in slow simulation speed for models that have a large number of components. This is because in every simulation cycle, all the simulators, no matter they are imminent or not, have to go through the simulation steps as described above. For the models that have only a few imminents (active cells), there exists a lot of unnecessary computation. To overcome this problem, an improved simulation engine is proposed as below. This simulation engine uses a *Heap* data structure to sort and find imminents and then only those imminents are asked to go through the simulation cycle.

3. A PROPOSED IMPROVED SIMULATION ENGINE

This proposed simulation engine implements a heap to keep track of the smallest *tNs* of its component simulators. During simulation, each active simulator removes and inserts its *tN* in the heap. So the smallest *tN* and imminents can be found from the root of the heap.

Using a heap to keep track of the *tNs* and imminents, the simulation protocol of this proposed simulation engine in each simulation cycle is shown below:

```
tN = Heap.getMin()
imminents = Heap.getImms()
imminents.tellAll("computeOut",tN)
imminents.tellAll("sendOut")
imminents = imminents.addAll(influencees)
imminents.tellAll("ApplyDelt",tN)
imminents.tellAll("updateHeap")
```

In every simulation cycle, the simulation engine fist gets the smallest *tN* and the imminents from the heap. With the smallest *tN* and imminents in hand, the simulation engine then sends out the *computeOut* and *sendOut* messages to imminents. The *sendOut* message will trigger imminents to put their output messages to their destination simulators, which are called *influences*. The *influences*, like the *imminents*, need to execute their state transition functions. Thus before *imminents.tellAll("ApplyDelt",tN)*, the coordinator adds those *influences* into *imminents* by executing *imminents = imminents.addAll(influencees)*. At the end of the iteration, the coordinator asks all *imminents* to update their newest *tNs* in the heap to prepare for the next simulation iteration.

Generally speaking, the update process where each active simulator removes and inserts its *tN* in the heap has computation complexity $O(n*log_2N)$ (*n* is the number of *imminents* in the simulation cycle; *N* is the total number of cells). For large-scale cellular models with small number of imminents ($n<<N$), this proposed simulation engine has computation complexity at the magnitude of log_2N, thus resulting in considerable performance improvement.

Based on this proposed simulation engine, a new simulation engine, the *oneDCoord* in DEVSJAVA environment, is developed. This new simulation engine not only keeps track of the imminents and asks only the imminents to go through a simulation cycle, but also implements a *minSelTree* data structure to allow efficient search of the imminents.

4. THE NEW SIMULATION ENGINE AND ITS DATA STRUCTURE

4.1 The Simulation Protocol

The simulation protocol of this new simulation engine is similar to the proposed simulation engine with heap implementation. The only difference is that a new data structure *minSelTree* is developed to replaces the heap for keeping track of *tNs* and imminents. Below is the simulation protocol of this simulation engine.

```
tN = minSelTree.getMin()
imminents = minSelTree.getImms()
imminents.tellAll("computeOut“,tN)
imminents.tellAll("sendOut")
imminents = imminents.addAll(influencees)
imminents.tellAll("ApplyDelt“,tN)
imminents.tellAll(“sendTNUp”)
```

As can be seen, in every simulation cycle, the new simulation engine first gets the smallest *tN* and *imminents* by executing *minSelTree.getMin()* and *minSelTree.getImms()* respectively. Then similar to the description above, only the *imminents* (and *influences*) go through the simulation cycle. The last step in the simulation cycle is to ask all imminents to send their newest *tNs* to the *minSelTree* so the information kept there is updated timely. This is to prepare for the next simulation iteration.

4.2 The *minSelTree* Data Structure

The *minSelTree* data structure is an essential part of this new coordinator to keep track of *tNs* and imminents. It is a complete tree, which means each leaf node of this tree data structure has the same “distance” from the root. The *minSelTree* is constructed in such a way that for each cell in the model, there is a leaf node of *minSelTree* corresponding to it. To retain a cell’s spatial information in *minSelTree*, the cell’s ID is used as a reference to assign a leaf node to the cell. As adjacent cells have adjacent IDs, their corresponding *minSelTree* nodes will sit adjacently in the *minSelTree* too. The *minSelTree* is set up during initialization of the simulation based on the total number of cells and the base of the tree. Here the base of the tree means the number of children of each internal node. It is provided by the user who runs the simulation. Different bases may make the simulation have different performance. The formula below shows the relationship among the number of cells *N*, the base *b* and the height *h* (distance from leaf to the root) of the *minSelTree*.

$$h = ceiling(\log_b N)$$

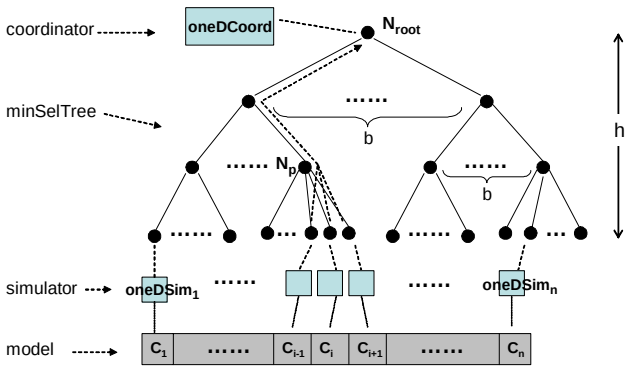

Figure 2. Model, simulator, and the *minSelTree*

Figure 2 shows the relationship among the models, simulators, and the *minSelTree* data structure. Here we assume the model to be simulated is a one-dimension cellular space model with *n* cells. These cells have IDs from *1* to *n* based on their positions in the cellular space. From the figure we can see that for each cell, there is a simulator *oneDSim* assigned to it; for each *oneDSim*, there is a leaf node of *minSelTree* assigned to it. Thus a cell, its simulator, and the *minSelTree* leaf node form one to one relationships to each other. Among them, the simulator has access to both the cell and the leaf node. Figure 2 also shows that the coordinator *oneDCoord* has access to the root node of *minSelTree*.

During simulation, a simulator *oneDSim* is responsible to drive the simulation of its cell model and to update the new *tN* to the corresponding leaf node of *minSelTree*. The leaf node then sends this *tN* up to its parent node, which compares this *tN* with the *tNs* of other children and selects the smallest ones to send up. This “*send up*” process continues until the root node is reached. Note that the information that is sent up includes not only the smallest *tN*, but also the references of those simulators which hold that *tN*. As each node selects the smallest *tN* and the imminent simulators to send up, after this recursive “send up” process ends, the root node has the global minimum of *tNs* and the references for all the imminent simulators. The coordinator *oneDCoord* can then access the root node of *minSelTree* and easily gets this information.

To make this process feasible, each node of the *minSelTree* keeps track of its children’s *tN* and imminent simulators (A leaf node keeps track of its own *tN* and itself as the imminent simulator). Specifically, each node of *minSelTree* has a variable *minEnvironment*, which basically is a table storing information in the following format: *(childName, Pair(imminents, tN))*. The *childName* is the name of a child node. The *tN* is that child node’s *tN* and the *imminents* is a set containing the references for all the simulators holding that *tN*. The *imminents* and *tN* are encapsulated into a *Pair*. With this information stored in *minEnvironment*, a node can find its *tN* (the smallest tN among its children) and the references for the simulators holding that *tN* by executing the following *whichMin()* method:

```
public Pair whichMin(){   // find the imminents and tN
   double timeGranule = .000001;
   ensembleSet imminents = new ensembleSet();
   double min = POSITIVE_INFINITY;
   while(minEnvironment.hasNext()) {
     Pair p = (Pair) minEnvironment.next(); //name, Pair
     Pair pp = (Pair)p.getValue(); //imms, tN
     double tN = pp.getValue();  // get tN
     if (Math.abs(tN - min)< timeGranule)
          imminents.addAll((ensembleSet)pp.getKey());
     else if (tN < min){
          imminents = new ensembleSet();
          min = tN;
          imminents.addAll((ensembleSet)pp.getKey());
     }
   }
   return new Pair(imminents, min);
}
```

This method compares all *tNs* stored in *minEnvironment* and selects the smallest one. In the meantime, it adds the simulators that hold the smallest *tN* into the *imminents* set. The method returns a new pair that contains the *imminents* and the smallest *tN*. Notice that a variable *timeGranule* is used when comparing and selecting the smallest *tN*. This is because the time base in DEVS is continuous rather than discrete. Thus the next event time *tN* can be any value with arbitrary precision. This *timeGranule* is used to

specify the smallest time unit in simulation. Events that happen inside the same *timeGranule* are considered happened at the same time.

The information stored in *minSelTree* need to be updated continuously when simulation proceeds. This is accomplished by executing *imminents.tellAll("sendTNUp")* at the end of each simulation iteration. This step asks all imminent simulators to update their new *tNs* to their corresponding leaf nodes. As mentioned before, the update of a new *tN* triggers a recursive "*send up*" process until the root node is reached. During this process, each node finds the smallest *tN* among its children (by executing the *whichMin()* method) and sends that information up.

As mentioned before, cells are coupled to their neighbors in cellular space models. Thus it's typical that a cell and its neighbors change their states at the same time. For example, in Figure 2, cell c_{i-1}, c_i, and c_{i+1} may all change their states at the same time and have new *tNs*. If the leaf nodes for these cells send their *tNs* up independently, each of them will trigger a "*send up*" path to the root node N_{root}. Apparently this is inefficient as these nodes actually share the same parent N_p (because the cells are adjacent in the cellular space model). Thus the "*send up*" paths behind node N_p can be bundled into one path. By bundling several "*send up*" paths into one path, the *minSelTree* nodes on that path only need to execute the *whichMin()* once instead of several times. Notice that this improvement actually shows how the new coordinator takes advantage of the fact that activities of cellular models usually happen locally. Because the spatial information of cells is retained in *minSelTree*, a cell and its neighbors' nodes will share the same parent in *minSelTree*. Thus their "*send-up*" paths can be bundled together, which results in performance improvement. The code listed below shows how a *minSelTree* node implements the *sendUp()* method.

```
public void sendUp (String nm,Pair p){
  minEnvironment.setPair(nm,p);
  receivedImmi++;
  if(receivedImmi== expectedImmi){
     receivedImmi=0; //reset the value for the next cycle
     expectedImmi =0; //reset the value for the next cycle
     whichMin = whichMin();
     if (!root) parent.sendUp(myName,whichMin);
  }
}
```

As can be seen, this *sendUp()* process is a recursive process. It continues until the root node is reached. When receiving an update from a child node with its name and the *(imminents, tN)* pair, the method first updates its *minEnvironment* variable. Then it increases *receivedImmi* and check if *receivedImmi* equals *expectedImmi*. The two variables *expectedImmi* and *receivedImmi* are used to guarantee that only one "*send up*" path is invoked even a node receives multiple calls from its children. Only when *receivedImmi* equals *expectedImmi*, which means the node has got all expected updates from its children, does the method executes the *whichMin()* to find the smallest *tN* and then call *sendUp()*to send this information up. The variable *expectedImmi*, meaning how many update a node expects from its children, is set by the *informChange()* method as shown below.

```
public void informChange(){
  if(expectedImmi ==0&&!root)
     parent.informChange();  // inform change only once
  expectedImmi ++;
}
```

Whenever a cell changes its state and has a new *tN*, its simulator calls the corresponding leaf node's *informChange()* method, which will increase the *expectedImmi* of that node. This method also makes sure that the parent's *informChange()* method will only be called once. Using the system shown in Figure 2 as an example, if cell c_{i-1}, c_i, and c_{i+1} all change their states, the *expectedImmi* of node N_p is 3; the *expectedImmi* of node N_p's parent node is 1 (assuming there are no other cells changing their states).

4.3 Building *minSelTree* for Two-dimension Cellular Space Models

The model given in Figure 2 is a one-dimension cellular space model. In a two-dimension cellular space model, cells have neighbors not only along the ***x*** dimension but also along the ***y*** dimension. This is shown in Figure 3.

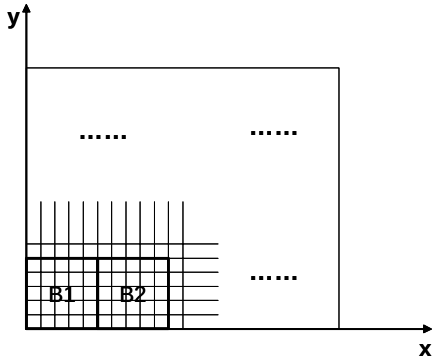

Figure 3. A two-dimension cellular space model

To construct *minSelTree* for a two-dimension cellular space model, one way is to assign consecutive IDs to all the cells row by row, thus treating the two-dimension cellular space model as a one-dimension model. However, this means cells are clustered (having nearby IDs) in only one dimension, which results in the situation that two neighbor cells may not be assigned nearby *minSelTree* nodes. To take good advantage of localized activities of cellular models, it is desirable that cells are clustered in two dimensions. This is why in our implementation, we assign cells' IDs based on blocks as shown in Figure 3. All cells inside one block belong to one parent in the *minSelTree*. The resulting *minSelTree* is a little bit different from the one discussed before. In the tree for one-dimension cellular space models, each internal node of the tree has *b* children (*b* is the base of the tree). For the new *minSelTree*, it is the same situation except that the bottom nodes have *BXs*BYs* children (*BXs* and *BYs* denote the size of a block). Notice that the previous tree is actually a special case of this new one with *BXs=base* and *BYs=1*.

5. PERFORMANCE ANALYSIS

Compared to the original coordinator with the basic simulation protocol, there are two sources of speedup of the new simulation engine. One is the keeping track of imminents so only the imminent cells will be considered in every simulation cycle. The other is the efficient smallest *tN* search which can be arbitrarily faster in cellular space models where the number of cells is very large but the number of imminent cells is small in every simulation iteration. Below we analyze the performance of these simulation engines by considering two extreme cases: only one cell is imminent in a simulation iteration, and all cells are imminent in a simulation iteration. For simplicity, we only consider the

computation complexity of finding the smallest *tN*. Let's assume there are *N* cells in the cellular space model.

For the coordinator with the basic simulation protocol, all simulators are asked to send their *tNs* to the coordinator. Then the coordinator compares and selects the minimum among these *N tNs*. Thus the computation complexity is *O(N)*, which is independent of the number of imminent cells.

For the coordinator with heap implementation, each imminent simulator removes and inserts its *tN* in the heap. Then the smallest *tN* is found by getting the value of the top of the heap. If only one cell is imminent, only one simulator needs to updates its *tN* in the heap, which takes worse case $O(log_2N)$. If all cells are imminent, all simulators need to remove and insert their *tNs* in the heap, resulting in computation complexity $O(N*log_2N)$.

For the new coordinator with *minSelTree*, the height of the tree $h = ceil(log_bN)$ (assuming the base of this tree is *b*). If only one cell is imminent in a simulation iteration, only one simulator will update its new *tN* on the leaf node of *minSelTree*. Thus only one "*send-up*" path, which has *h* nodes in the path, will be generated. Along this path, each node executes *whichMin()* method to compare and select the minimum *tN* among *b* children. This results in computation complexity $O(b*h) = O(b* log_bN)$. If all cells are imminent, all simulators will update their new *tNs* on their leaf nodes of *minSelTree*. As a result, all nodes in the *minSelTree* will be involved in the "*send up*" process. However, even a node's *sendUp()* method will be called multiple times, the *whichMin()* method will only be executed one time. Assume the total number of nodes in *minSelTree* is *T* (exclude the leaves of the tree), then the computation complexity is *O(T*b)*. We know that for a complete tree with *N* leaves, *T=(N-1)/(b-1)*. This results in computation complexity $O(T*b) = O((N-1)*b/(b-1)) = O(N)$.

The above analysis shows that when all simulators are imminents, both the standard coordinator and the coordinator with *minSelTree* has computation complexity *O(N)*, which is better than $O(N*log_2N`)$, the complexity of the proposed coordinator with heap implementation. However, when only one (or small number of) simulator is imminent, the coordinator with heap implementation has computation complexity $O(log_2N)$, which is close to the performance of the new coordinator $O(log_bN)$. To further compare the heap implementation and the *minSelTree* implementation, let's consider another example. E.g., let $N = b*b$ for a two-level *minSelTree* and let there be *b* imminent cells within one block (this means the leaf nodes of these cells share the same parent). Considering only the computation required by comparisons, the new coordinator takes *2b*. And the heap implementation takes $blog_2N = blog_2(b*b) = 2blog_2b$, while the old coordinator takes *b*b*. The utilize of localized activity is clear here: *2b* vs $2blog_2b$. The extra log_2b part is due to the full reordering that the heap implementation does for every replacement.

6. EXAMPLES AND TEST DATA

This section gives two examples to compare the simulation performance of the new coordinator and the standard one. The first example is a one-dimension cellular space model that models the phenomena of diffusion. The second example is a two-dimension cellular space model that models the phenomena of fire spreading. For each example, a brief description of the model is given and then the simulation data of the two simulators is provided to show the speedup the new simulator. More detailed description of these examples can be found in the related references.

6.1 Simulation of a Diffusion Model

Diffusion is a common problem that has been studied using simulation methods. In this example (the *diffuse2ndOrdCellSpace* example in the DEVSJAVA environment [14]), the heat diffusion phenomenon is modeled as a one-dimensional cell-space model with each cell coupled to its left and right neighbors (except the boundary cells). A cell holds its current temperature and will updates it to reach the desired temperature, which is defined as the average of the left and right cells' temperature. The speed of temperature change is defined by the difference between a cell's desired and current temperatures. For each cell, a quantum is provided so a cell will update itself and pass message to its neighbors only when the change of temperature reaches the quantum size. With this approach, each cell can calculate its time advance for the next update based on the quantum and the speed of temperature change. The initial condition of the simulation is that temperature is linearly distributed along this one dimensional cell space, with the first cell has the lowest temperature and the last cell has the highest temperature.

Table 1 shows the time (in seconds) of simulating this diffusion model with different number of cells using the standard coordinator and the new simulation engine respectively. For each simulation running, the quantum is defined as 0.1, and the number of simulation iterations is 10000. The simulations were run on a laptop with Intel Pentium 4 1.7GHZ processor, 256M memory, and Windows 2000 OS.

Table 1: Comparison of the simulation time (sec) of the two simulation engines

number of cells	50	100	500	1000
coordinator	29.592	54.448	272.712	542.721
oneDCoord (base = 6)	7.221	7.24	8.222	8.432
speedup (coordinator time/oneDCoord time)	4.098047	7.520442	33.16857	64.36444

The test data above shows how the change of the number of cells affects the execution time of *coordinator* and *oneDCoord*. It shows that as the number of cells increase, the execution time of *coordinator* linearly increases too. However, the execution time of *oneDCoord* only increases slightly. This is because in this example, the increase of cells doesn't affect the number of imminents in every simulation cycle. In fact, in this example, the number of imminents remains 1 or 2.

6.2 Simulation of a Fire Spreading Model

This example describes a dynamic forest fire spread model, which is based on the work of [8, 18]. In this model, a forest is modeled as a two-dimensional cell-space composed of individual forest cells coupled together according to their relative physical geometric locations. Each cell is modeled in the same way as that of [15, 2]. Specifically, each cell has the following six states: *unburned*, *burning*, *burned*, *unburned-wet*, *burning-wet*, and *burned-wet*. Conditions and rules are defined to govern the state transition of a cell. In the two-dimensional cell space model, each cell has eight neighbor cells *N, NE, E, SE, S, SW, W,* and *NW* except the boundary cells. Accordingly, for each cell, fixed fire spreading directions *N, NE, E, SE, S, SW, W,* and *NW* are defined. Fire spread in each cell is modeled using Rothermel's [16] stationary model, which is a one-dimension semi-empirical model. To obtain the second dimension, a propagation algorithm that uses maximum rate of spread and wind and slope factors is applied.

During simulation, the behavior of a burning cell can be influenced by external inputs from neighboring cells as well as changes in weather conditions. In addition, uncertainty is incorporated in the model by allowing certain critical parameters to be sampled from arbitrary probability distributions during the simulation run. A detailed description of this model and the initial condition of simulation can be found in [8].

Table 2 shows the time (in seconds) of simulating the fire spread model with different number of cells using the standard coordinator and the new simulation engine respectively. For each simulation running, the number of simulation iterations is 7200. The simulations were run on a laptop with Intel Pentium 4 1.7GHZ processor, 256M memory, and Windows 2000 OS.

Table 2: Comparison of the simulation time (sec) of the two simulation engines

Number of cells	30x30 (900)	34x34 (1156)	40x40 (1600)
coordinator	356.293	471.258	681.951
oneDCoord (base = 6)	19.388	23.924	36.002
speedup (coordinator time/oneDCoord time)	18.37699	19.69813	18.94203

7. CONCLUSION

The simulation data and performance analysis show that the speedup of the new simulation engine is significant as compared to the standard coordinator. This is achieved by keeping track of imminents during simulation and by implementing a data structure to allow efficient search of those imminents. As a note of caution, we mention that if a very large percentage of the models are imminent at any given instant, the timesavings would generally be much less, but this should hardly be the case in any realistic simulation.

Although this new simulation engine was initially developed to simulate cellular space models, it can be applied to other large-scale simulations such as the simulation of swarm intelligence that includes a huge number of bots, or the simulation of distributed networks that contain a large number of network nodes.

Applying DEVS to partial differential equation (PDE) simulation also uses cellular DEVS models, and these cellular models share the same characteristic such as activities are locally clustered. So this new simulation engine will also improve simulation performance for PDE simulations.

REFERENCES

[1]. Zeigler, B.P., T.G. Kim, et al.. Theory of Modeling and Simulation. New York, NY, Academic Press, 2000.

[2]. J. Ameghino, A. Tróccoli, G. Wainer. “Models of Complex Physical Systems using Cell-DEVS”, Proceedings of the Annual Simulation Symposium, Seattle, Washington, 2001.

[3]. Kofman, E.. “A Second Order Approximation for DEVS Simulation of Continuous Systems”, Simulation: Trans. of SCS, Vol 64, 2, pp. 76-89, 2002.

[4]. Alexandre Muzy, Eric Innocenti, Antoine Aiello, Jean-Franc,ois Santucci, and Gabriel Wainer. “Cell-DEVS Quantization Techniques in a Fire Spreading Application”, Winter Simulation Conference, San Diego, California, 2002

[5]. J. Nutaro, B. P. Zeigler, R. Jammalamadaka, S. Akerkar, “Discrete Event Solution of Gas Dynamics within the DEVS Framework: Exploiting Spatiotemporal Heterogeneity”, Proc. ICCS, Melbourne Australia, July 2003

[6]. S. Wolfram, Theory and Applications of Cellular Automata, World Scientific, Singapore, 1986

[7]. Muzy, A., G. Wainer, E. Innocenti, A. and Aiello, J.F. Santucci. “Comparing simulation methods for fire spreading across a fuel bed”, In Proceedings of AIS'2002, Lisbon, Portugal.

[8]. Lewis Ntaimo, Bithika Khargharia “A Cellular Devs Dynamic Fire Spread Simulation Model With An Optimized Cell Space”, Project Report, Electrical and Computer Engineering Department, University of Arizona, May, 2003

[9]. A. Davidson, G. Wainer. "Specifying truck movement in traffic models using Cell-DEVS". In Proceedings of the 33rd Anual Symposium on Computer Simulation. Washington, D.C. U.S.A. 2000

[10] A. Troccoli, J. Ameghino, F. Iñón, G. Wainer. "A flow injection model using Cell-DEVS". In Proceedings of the 35th IEEE/SCS Annual Simulation Symposium. San Diego, CA. U.S.A

[11] Wang, Y. H., and B. P. Zeigler. Extending the DEVS Formalism for Massively Parallel Simulation. Discrete Event Dynamic Systems: Theory and Applications, 3:193-218, 1992.

[12] G. A. Wainer. "Improved cellular models with parallel Cell-DEVS". In Transactions of the SCS. June 2000

[13] Steven B. Hall, Shankar M. Venkatesan, Donald B. Wood, “A Faster Implementation of DEVS in the Joint MEASURE Simulation Environment”, in Proc. of Summer Computer Simulation Conference, Montreal, July 2003

[14] DEVS-Java Reference Guide, www.acims.arizona.edu

[15] Vasconcelos, J. M, Modeling Spatial Dynamic Ecological Processes with DEVS-Scheme and Geographical Information Systems, Ph.D. Dissertation, Dept. of Renewable and Natural Resources, University of Arizona, Tucson, U.S.A., 1993

[16] Rothermel, R., “A mathematical model for predicting fire spread in wildland fuels”. Research Paper INT-115. Ogden, UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station, 1972

[17] Sunwoo Park, “Hierarchical Model Partitioning for Distributed Simulation of Hierarchical and Modular DEVS Models”, Ph.D. Dissertation, Univ. of Arizona, May 2003

[18] B. Khargharia, S. Hariri and M. Parashar, “vGrid: A Framework for Building Autonomic Applications,” *Proceedings of the Proceedings of the 1st International Workshop on Heterogeneous and Adaptive Computing, (CLADE 2003)*, Seattle, WA, USA, Computer Society Press, pp 19-26, June 2003