PARALLEL ALGORITHMS FOR CELLULAR MODELS SIMULATION

SHAFAGH JAFER
and
GABRIEL A. WAINER
Department of Systems and Computer Engineering, Carleton University, 1125 Colonel By Drive Ottawa, Ontario, K1S 5B6

ABSTRACT

DEVS is a sound formal modeling and simulation (M&S) framework based on generic dynamic system concepts. Cell-DEVS is a formalism for cell-shaped models based on DEVS. This work presents a new simulation technique for execution of DEVS and Cell-DEVS models in parallel environments. These techniques are modifications to the original Time Warp mechanism offered by WARPED kernel. Time Warp functionalities are revised to include two new algorithms namely, Local Rollback Frequency Model (LRFM) and Global Rollback Frequency Model (GRFM). The resulting simulator is used as new simulation engine for CD++, an M&S toolkit that implements DEVS and Cell-DEVS theories. The results obtained allowed us to achieve considerable speedups due to the reductions that LRFM and GRFM protocols perform on number of rollbacks and anti-messages.

KEYWORDS: Cellular Automata, Parallel Simulator, Cell-DEVS, Optimistic Simulator.

1. Introduction

Modeling and simulation (M&S) methodologies have become crucial for implementing, designing, and analyzing a broad verity of systems. Among the existing simulation techniques, the **DEVS** (Discrete Event System Specification) formalism [1] provides a discrete-event M&S approach which allows construction of hierarchical models in a modular manner. DEVS is a sound formal framework based on generic dynamic systems concepts that allows model reuse, and reduction in development and testing time due to its hierarchical approach in constructing models. **Cell-DEVS** [2] is an extension to DEVS which integrates DEVS and cellular automata by presenting each cell as an atomic DEVS model.

Cell-DEVS introduced a novel mechanism for computation based on asynchronous cellular models with explicit timing constructions. The technique has been used to develop a wide variety of models in different fields, ranging from environmental sciences, traffic, biology and physics. When large complex models are defined, the computing power of a parallel simulator can improve execution times. Here, we present new techniques for executing DEVS and Cell-DEVS models in parallel and distributed environments based on the WARPED kernel [3], an implementation of the Time Warp protocol [4]. Our optimistic simulator, called as PCD++, is built as a new simulation engine for CD++ [5], an M&S toolkit that implements the DEVS and Cell-DEVS formalisms. Algorithms in CD++ and the WARPED kernel are redesigned based on Near Perfect State Information technique to carry out optimistic simulations using a non-hierarchical approach that reduces the communication overhead. Two new algorithms namely, Local Rollback Frequency Model (LRFM) and Global Rollback Frequency Model (GRFM) have been implemented and used by our PCD++ simulator. These two algorithms have been tested using different Cell-DEVS models. Here we present in details an evacuation model of a ship and a model of the Synapsin-Vesicle reaction in neurons. Also, a brief description of two other models namely Fire Propagation model, and Game of Life model are provided.

We have designed many Cell-DEVS models which vary in size, complexity, and functionality. As the main contribution of this work, we have implemented two new optimism control mechanisms based on NPSI protocols. These two protocols, namely LRFM and GRFM were integrated into the existing optimistic PCD++ simulator and therefore two distinct optimism controlling simulators were modeled. This led to

creating a workbench consisting of four different simulators; Conservative, Pure Optimistic, LRFM-based Optimistic, and GRFM-based Optimistic simulators. This workbench serves as simulation environment that can be used as the base in studying parallel simulations of DEVS and Cell-DEVS. On the other hand, the precise and detailed testing scenarios that we are presenting can be used along with this workbench to analyze the capability, performance, and robustness of PCD++ simulators.

2. Background

DEVS [1] is a formalism for modeling and simulation for Discrete Events Dynamic Systems that provides a framework for the definition of hierarchical models in a modular way by decomposing the real system into behavioral (atomic) and structural (coupled) components. DEVS theory provides a rigorous methodology for representing models, and it does present an abstract way of thinking about the world with independence of the simulation mechanisms, underlying hardware and middleware. A DEVS atomic model is formally defined by:

$$M = <X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta>,$$

where

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

S is the set of sequential states;

$\delta_{int}: S \rightarrow S$ is the internal state transition function;

$\delta_{ext}: Q \times X \rightarrow S$ is the external state transition function, where

$Q = \{(s,e) \mid s \in S, 0 < e < ta(s)\}$ is the total state set, e is the time elapsed since the last state transition;

$\lambda: S \rightarrow Y$ is the output function;

$ta: S \rightarrow R^{+}_{0,\infty}$ is the time advance function.

The semantics for this definition is given as follows. At any time, a DEVS coupled model is in a state $s \in S$. In the absence of external events, the model will stay in this state for the duration specified by ta(*s*). When the elapsed time e=ta(*s*), the state duration expires and the atomic model will send the output λ(s) and performs an internal transition to a new state specified by δ_{int}(s). Transitions that occur due to the expiration of ta(s) are called internal transitions. However, state transition can also happen due to arrival of an external event which will place the model into a new state specified by δ_{ext}(s,e,x); where s is the current state, e is the elapsed time, and x is the input value. The time advance function ta(s) can take any real value from 0 to ∞.

A DEVS coupled model is composed of several submodels and it is formally defined by:

$$CM = <X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, Select>,$$

where

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

D is the set of the component names, and the following requirements are imposed on the components, which must also be DEVS models:

For each $d \in D$, $M_d = (X_d, Y_d, S_d, \delta_{int}, \delta_{ext}, \lambda, ta)$ is a DEVS model.

Select: $2^D \rightarrow D$ is the tie-breaking function for imminent components.

Due to the closure property, a coupled model is regarded as a new DEVS model [1]. This property clarifies that the overall behavior of a coupled model is equivalent to a basic atomic model, and therefore allows hierarchical model construction.

Cell-DEVS [2] is an extension to DEVS which integrates DEVS and cellular automata by presenting each cell as an atomic DEVS model. Two types of timing delays can be used, namely *transport* and *inertial* [6]. When transport delay is used, the future value is added to queue sorted by output time, allowing the

previous values that were scheduled for output but have not yet been sent to be kept. On the other hand, inertial delays allow a preemptive policy at which any previous scheduled output value will be deleted and the new value will be scheduled. Cell-DEVS formalism is defined by:

$$TDC = < X, Y, I, S, \theta, N, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D >$$

where *X* is a set of external input events; *Y* is a set of external output events; *I* represents the model's modular interface; S is the set of sequential states for the cell; θ is the cell state definition; N is the set of states for the input events; d is the delay for the cell; δ_{int} is the internal transition function; δ_{ext} is the external transition function; τ is the local computation function; λ is the output function; and D is the state's duration function. The model uses N inputs to compute its next state. These inputs, which are received through the model's interface (***X***, ***Y***), activate the local computing function (**τ**). State (**s**) changes can be transmitted to other models, but only after the consumption of a delay (**d**). Two kinds of delays can be defined: *transport* delays model a variable commuting time, and *inertial* delays, which have preemptive semantics (scheduled events can be discarded). Once the cell behavior is defined, a coupled Cell-DEVS is created by putting together a number of cells interconnected by a neighborhood relationship.

By integrating atomic Cell-DEVS, coupled models can be constructed representing the cell space. A coupled Cell-DEVS model is formally defined as follows:

$$GCC = < Xlist, Ylist, I, X, Y, n, \{t_1,...,t_n\}, N, C, B, Z, select >$$

where *Xlist* is the input coupling list; *Ylist* is the output coupling list; *I* represents the definition of the model's interface; *X* is the set of external input events; *Y* is the set of external output events; *n* is the dimension of the cell space; $\{t_1,...,t_n\}$ is the number of cells in each of the dimensions; N is the neighborhood set; C is the cell space; B is the set of border cells; *Z* is the translation function; and *select* is the tie-breaking function for simultaneous events. The above formalism explains that a coupled model is composed of an array of atomic cells with given size and dimensions where each cell is connected through standard DEVS input/output ports to the cells defined in the neighborhood . Since the cell space is finite, the borders of the cells are either connected to a different neighborhood than the rest of the space, or they are "wrapped" in which they are connected to those in the opposite one using the inverse neighborhood relationship. However, border cells have a different behavior due to their particular locations, which result in a non-uniform neighborhood. A Cell-DEVS coupled model is informally presented in Fig. 1.

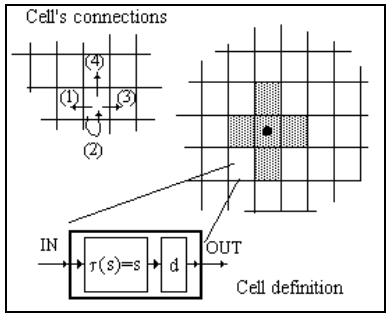


Fig. 1. Description of a Cell-DEVS atomic model [6]

CD++ [5] is a modeling tool that implements the DEVS and Cell-DEVS theories by applying the original formalisms. The toolkit includes facilities to build DEVS and Cell-DEVS models. CD++ toolkit also includes an interpreter for Cell-DEVS models [6]. The language is based on the formal specifications of

Cell-DEVS. The model specification includes the definition of the size and dimension of the cell space, the shape of the neighborhood and the type of cell's bordering. The cell's local computing function is defined using a set of rules with the form *POSTCONDITION DELAY { PRECONDITION }*. These indicate that when the *PRECONDITION* is met, the state of the cell will change to the designated *POSTCONDITION* after the duration specified by *DELAY*. If the precondition is not met, then the next rule is evaluated until a rule is satisfied or there are no more rules.

In parallel and distributed environments the entire task of simulation is divided among the processors or nodes (Logical Process - **LP**) and therefore each one of them handles a smaller chunk of the simulation while the whole process of execution takes place in parallel and as a result in a significantly reduced time. In sequential simulations, events are executed base on timestamp order; in contrast, parallel and distributed simulations require a mechanism to ensure that the result of concurrent execution is identical to that of sequential one [7]. Therefore, synchronization among LPs is needed. The most widely used strategies for event driven simulations can be classified as **Conservative** (or Pessimistic), in which causality violations are strictly avoided [8], and **Optimistic** [4], in which causality errors are fixed by the notion of *rollbacks*.

Conservative synchronization can cause deadlocks, which can be avoided by providing *lookahead* information (i.e., the smallest time stamp of the new events that a process can schedule in the future). *Null* messages are responsible to carry out the lookahead information among LPs. This way each LP, based on the lookahead information that it receives from all other LPs can derive a lower bound on the time stamp (LBTS) of the events that it will receive in future. As a result, the LP would know which event is safe to process. The biggest drawback of the conservative synchronization approach is the time wasting flow of null messages which degrade the simulation performance significantly. Optimistic techniques [4] consider that each LP has a Local Virtual Time (LVT) which advances every discrete step as events are executed on the process. Therefore, time warp processes execute their own portion of the simulation based on LP's LVT. Since every LP has its own LVT, causality errors occur when LPs send messages to each other. This way, an LP may receive a message with time stamp smaller than its current LVT. Such events are referred to as straggler events. Once a straggler event is received the process will rollback. Rollback is the operation performed upon reception of a straggler event, where the process recovers from the causality error by undoing the effects of all the events that were processed and had timestamp greater than the time stamp of the straggler event. Therefore, these messages were falsely sent to other processes and now must be cancelled. This cancellation is performed by sending anti-messages.

Optimistic approaches offer two important advantages over conservative techniques:

(i) They have a higher degree of parallelism unlike the conservative approaches where they are overly pessimistic and force the simulation to behave sequentially when it is not necessary.

(ii) Conservative approaches rely very much on application-specific information when making run-time decisions on whether it is safe to process the event or not. Optimistic mechanisms allow a simplified software development and more transparent synchronization.

3. Definition of a Parallel Simulator

PCD++ optimistic simulator implements the DEVS and Cell-DEVS formalisms in parallel and provides the framework for building and executing DEVS and Cell-DEVS models in parallel environments using the Time Warp protocol. We have modified CD++ sequential simulator to enable parallel and distributed simulations by implementing optimistic synchronization protocol [4]. PCD++ executes the simulation via several Time Warp processes [3] by exchanging time-stamped event messages. The Time Warp protocol

used by PCD++ consists of two parts: the *local control mechanism* and the *global control mechanism*. The local control mechanism which is provided in each Time Warp process is in charge of rollback operations which include: sending anti-messages, restoring the state of the LP, readjusting Local Virtual Time (**LVT**), etc. On the other hand, the global control mechanism takes care of global issues such as memory management, I/O operations, and termination detection.

We used the WARPED [14] simulation kernel, which is a configuration middleware that implements the Time Warp mechanism and a variety of optimization algorithms. Warped uses the Message Passing Interface (MPI), a standard specification of message-passing library for high-performance communications on both massively parallel machines and on workstations clusters [12]. We have used the MPICH [12] portable implementation of MPI which provides a vehicle for MPI implementation research and for developing parallel and distributed applications. CD++ simulation is driven by message passing.

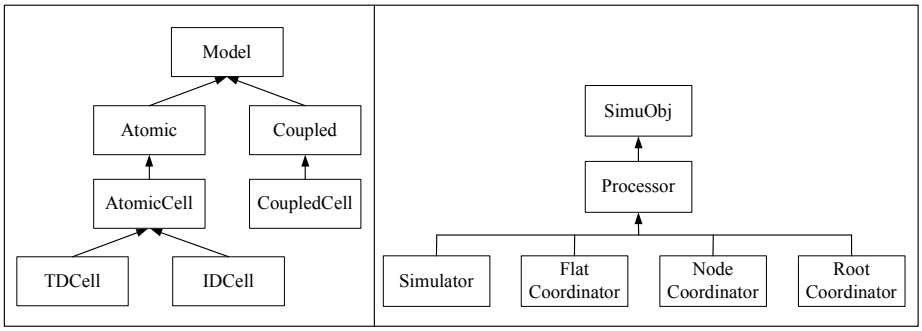


Fig. 2. Model and Processor hierarchies.

As seen on in Fig. 2, there are four types of PCD++ processors (associated to the Modeling hierarchy): *Simulator*, *Flat Coordinator (FC)*, *Node Coordinator (NC)*, and *Root Coordinator (RC)*. When DEVS and Cell-DEVS models are executed over multiple machines, a distributed processor structure is constructed in PCD++ to carry out the simulation. Lets consider the following example to see how partitioning takes place (on two machines). Fig. 3 shows a scenario with four atomic models (A1, A2, A3, and A4) where A1 and A2 are part of the coupled model C1, and C1, and the other two atomic models A3, and A4 are then part of the TOP coupled model. Since we will execute the simulation on two machines, we will allocate models by putting A1 and A2 in Machine 0, and A3 and A4 in Machine1.

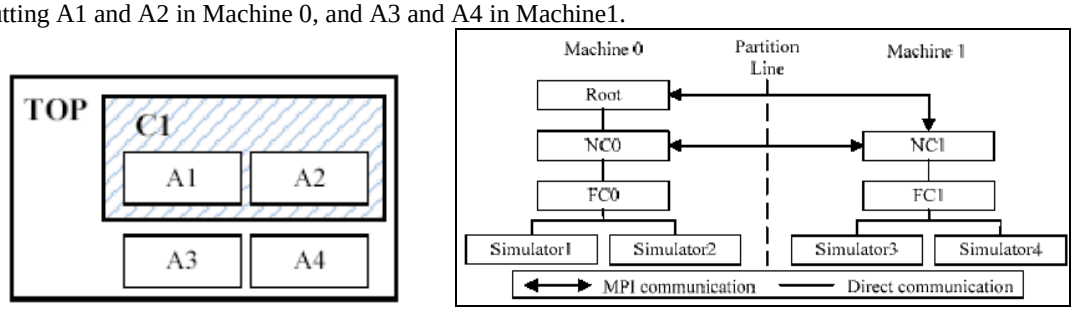


Fig. 3. Example model structure and distributed processor structure for the example

In this case, two logical processes are created LP0 and LP1 (one per machine). LPs group together the PCD++ processors on the machine they belong to. Local messages are handled by the FC, and the remote

messages are handled by the NC and then sent to the appropriate Simulator through the destination FC. The *root coordinator* is created only on machine 0. It starts the simulation and handles I/O operations. The NC on each machine is the local central controller on each LP and the end point of inter-LP communications. The FC residing between the NC and the Simulators is responsible for synchronizing the execution of its child Simulators. Finally, the Simulator is responsible for executing DEVS abstract functions defined in the atomic models. When a Simulator sends a message to another Simulator sitting on a remote machine, the message is first directed to the FC, then to the local NC through direct communication. Once the message gets to the NC, it will be forwarded to the destination NC through MPI communication. On the receiving end, the NC will then forward the message to the destination Simulator through the child FC.

There are two types of communications among LPs: *synchronous* intra-LP communications, carried out by all PCD++ processors, and *asynchronous* inter-LP communications, carried out only by NCs. Since inter-LP communications are asynchronous, the NCs require a special structure named as *NC Message Bag* to handle message passing between LPs with different LVTs. The following properties hold for *NC Message Bag*:

(i) Messages inside a *Message Bag* can have different timestamps.
(ii) The time of a *Message Bag* is equal to the minimum timestamp among the contained messages. If the *Message Bag* is empty, then its time is set to infinity.
(iii) Messages inside a *Message Bag* are processed based on their timestamp in an increasing order. Once a message is processed, it is then removed from the bag, and the bag's time is advanced to the next minimum value among the timestamps of the remaining messages. Once all the messages are processed and removed from the bag, the *Message Bag*'s time is restored back to infinity implying that the bag is empty.

In contrast, synchronous intra-LP communications are handled by the Simulators and the FC since they are local to the LP. Similar to the *NC Message Bag*, for intra-LP messages the FC holds a *message bag*. In this case, when two local Simulators (i.e. sitting on the same LP) need to communicate to each other, they send the message to the local FC, and then the message will be directed to the destination local Simulator by the FC. There is no direct communications between Simulators, even the ones sitting on the same LP. Local Simulators can only communicate with each other through their FC.

PCD++ processors exchange *content* or *control* messages. The first category includes the external message (*x*) and the output message (*y*), and the second category includes the initialization message (*I*), the collect message (@), the internal message (*), and the done message (*D*). External and output messages are used to exchange simulation data between the models. Initialization messages start the simulation, collect and internal messages trigger the output and the state transition functions respectively in the atomic DEVS models, done messages handle synchronization by carrying the model timing information. Each PCD++ processor defines its own functionality for each type of message, as follows:

Simulator: upon receiving (*I*, 0) from the parent FC, the current simulation time (t_L) and the next scheduled event (t_a) are recorded. Then the simulator initializes the variables defined in its associated atomic model, and after that, it informs its parent FC of the value of t_a by sending a *done* message stamped with time 0. When a (@, *t*) message is received, the Simulator invokes the output function (λ) of the atomic model and as a result an output message (*y*, *t*) is sent to the FC. After this, the Simulator will send (*D*, *t*) to the FC with $t_a = 0$ to indicate that it is imminent. Following the collect message, a (*,*t*) will arrive to trigger internal/external/ confluent function of the atomic model depending on the timing of the message and the status of the Simulator's message bag. A message (*x*, *t*) is simply inserted into the Simulator's message bag.

Flat Coordinator: when (*I*, 0) is received, the FC records the total number of its children and forwards the (*I*, 0) message to each child. After this, the FC waits for all its children to respond to this initialization by sending back a (*D*, 0). The FC will only pass the control over to the NC if all its children have finished their previous computation and have sent done messages as notification messages. Upon receiving a (*@*, *t*) message, the FC forwards it to all imminent Simulators and will keep a record of this for later use (to know which children need to do state transitions when (*, *t*) is received). Moreover, when (*y*, *t*) is received, the FC searches the model coupling information to find out the correct destination. The destination is either an input port on an atomic model, or an output port on the topmost coupled model. In case of receiving (*x*, *t*) message, the FC will simply insert the message into its message bag. Upon receiving (*, *t*) message, the external messages inside the FC's message bag are flushed to the local receiving Simulators. This will trigger the imminent Simulators to perform a state transition. Finally, when a (*D*, *t*) message is received from a child Simulator, the FC updates the child's t_N.

Node Coordinator: upon receiving (*I*, 0), the NC simply forwards it to the child FC. In case of receiving (*x*, *t*), NC will insert this message into the *NC Message Bag*. These external messages contain values sent from remote Simulators to local ones. When (*y*, *t*) is received the NC simply forward it the Root (it has to be sent to the environment). Reception of a (*D*, *t*) message by the NC from a child FC indicates that this is a response to a control message that was previously sent out by the NC.

Root Coordinator: this processor only handles environment-oriented output messages during the simulation. Output to the environment is done through a test file called as output file or OUT file.

Aside from the functionalities of each of the PCD++ processors, we have modified the WARPED [3] kernel in order to run simulations under different protocols. These protocols are modifications of the optimistic one that WARPED implements. The idea is to reduce the number of rollbacks by suspending the LP that has large number of rollbacks and therefore stopping it from flooding the net with anti-messages. However, the LP will still be able to receive input events and they will be inserted into the corresponding message bags. After a predefined duration, the suspend LP is released and will go on simulating. These two protocols [15], namely Local Rollback Frequency Model (LRFM) and Global Rollback Frequency Model (GRFM) are based on the "Near Perfect State Information - NPSI" protocol [16]. The NPSI protocol implements the Elastic Time mechanism. Briefly, Elastic Time is composed of two parts: (i) identifying the NPSI of the simulation, and (ii) translating the NPSI in optimism on the simulation objects.

Each part can be implemented in many ways. The main concept is to associate each LP with a *potential error* (**PE**) to control the optimism of LP_i. During the simulation run, the value of each PE is kept updated by evaluating a function called *M1* which uses state information that is received from the feedback system. Then, the function *M2* translates dynamically every update of PE_i in delays in the execution events.

3.1. Local Rollback Frequency Model

The Local Rollback Frequency Model (LRFM) protocol is only based on local information of the logical processes. That is, the simulation object within a LP will be suspended or allowed to continue simulating only based on the number of rollbacks it had. First, M1 and M2 functions must be defined:

Function M1: The error potential of a simulation object is the number of rollbacks that the object had from a time T1 until the actual time T2, having that $T2 - T1 \leq T$, where T is the interval after which the local number of rollbacks of the simulation object gets restarted back to zero.

Function M2: If the number of rollbacks for a simulation object at the interval T is greater than a specified value, then the object is suspended, adopting a conservative behavior. By suspending the

simulation object, the LP where the object resides on will still be able to receive incoming events, but the events are not processed until the simulation object is again given the permission to resume. However, if the number of rollbacks of the simulation object is less than the predefined value, then the object simulates aggressively, adopting its usual optimistic behavior (as in Time Warp).

To implement this protocol each LP has to be informed the maximum number of allowed rollbacks before suspension of the simulation object (*max_rollback*), and the duration for which the simulation object will stay suspended (*period*). The algorithm is presented in Fig. 4.

```
1. In each LP, at the beginning predefine:
   max_rollbacks and period
2. In each simulation object, at the simulation start:
   previous_time = 0
3. In each object, when the LP is scheduled to run:
   actual_time = Warped.TotalSimulationlTime ()
        if (actual_time - previous_time >= period)
                simulateNextEvent()
                previous_time = actual_time
                rollbacks = 0
        else
                if (rollbacks < max_rollbacks)
                simulateNextEvent()
                /* else, SUSPEND the simulation object */
```

Fig. 4. LRFM algorithm

From the LRFM algorithm we see the following three possible scenarios:

- The LRFM period has expired, therefore the simulation object starts a new period, its number of rollbacks gets reset to zero, and it is given the permission to continue its execution.
- The period has not yet expired. If the number of rollbacks of the simulation object is less than the allowable range (i.e. *max_rollbacks*), it continues its normal execution.
- The LRFM period has not yet expired, but the number of rollbacks has exceeded *max_rollbacks*, thus it gets suspended for the entire duration of the current LRFM period.

With the inclusion of this protocol, in every simulation cycle an object will simulate the lowest timestamp event if the number of its rollbacks in the period *T* is smaller than the maximum allowable rollbacks; if not, the object suspends executing until the new period of time *T*, after which Warped restarts the rollbacks number to zero. In order for an LP to be able to simulate objects that mustn't be delayed, we have modified the scheduler policy to choose the next object to simulate. It chooses the first object of the input event list (that is, the object with the lowest input event timestamp) only if its rollbacks count does not exceed *max_rollbacks*; else, the scheduler checks the next object of the input event list and so on, until it finds an object in condition to be simulated or until it reaches the end of the list.

3.2. Global Rollback Frequency Model

In the Global Rollback Frequency Model (GRFM) protocol each simulation object uses global information in such a way that among all the simulation objects residing on all LPs, the one with greatest number of rollbacks must be suspended for the duration of time defined at the beginning of the simulation. Therefore, at each simulation cycle all the LPs must broadcast the information regarding the rollback counts of all of their simulation objects. As in LRFM, M_1 and M_2 functions must first be defined:

Function M1: The error potential of a simulation object is the number of rollbacks that the object had minus the maximum number of rollbacks of the other simulation objects (both local and remote ones) participating in the simulation, from a time T1 until the actual time T2, having that $T2 - T1 \leq T$, where T is the interval after which the local number of rollbacks of the simulation object gets restarted.

Function M2: If the number of rollbacks for a simulation object at the interval T is greater than other number of rollbacks of the other simulation objects, then the object is suspended, adopting a conservative behavior. By suspending the simulation object, the LP where the object resides on will still be able to receive incoming events, but the events are not processed until the simulation object is again given the permission to resume. However, if the number of rollbacks of the simulation object is less than the predefined value, then the object simulates aggressively, adopting its usual optimistic behavior (as in Time Warp). The algorithm is presented in Fig. 5.

```
1. In each LP, at the beginning predefine: period
2. In each simulation object, at the beginning predefine:
      previous_time  = 0
      max_rollbacks = 0
3. In each simulation object, when the LP is scheduled to run:
    actual_time = Warped.TotalSimulationlTime ()
          if (actual_time - previous_time >= period)
                simulateNextEvent()
                previous_time  = actual_time
                rollbacks = 0
          else
                if  (rollbacks < max_rollbacks)
                simulateNextEvent()
          /* else, SUSPEND the simulation object */
4. For i from 1 until the number of LPs
          if (i is NOT this LP id)
          send to LP i the number of rollbacks of the objects of the LP id
          Subroutine that receives the number of rollbacks from other LP:
          For j from 1 until the numbers received
                If (rollbacks[j] > max_rollbacks)
                max_rollbacks = rollbacks[j]
```

Fig. 5. GRFM algorithm

As in LRFM, the GRFM algorithm yields three different scenarios:

- The GRFM period has expired, therefore the simulation object starts a new period, its number of rollbacks gets reset to zero, and it is given the permission to continue its execution.

- The GRFM period has not yet expired, if the number of rollbacks of the simulation object is less than the allowable range (i.e. *max_rollbacks*), then the simulation object continues its normal execution.

- The GRFM period has not yet expired, but the number of rollbacks has exceeded *max_rollbacks*, thus the simulation object gets suspended for the entire duration of the current GRFM period.

The main difference of GRFM and LRFM is the way *max_rollbacks* is initialized. In LRFM, the maximum allowable rollbacks are predefined by the user at run time, while in GRFM maximum allowable rollbacks is set to the largest number of rollbacks of all participating simulation objects. That is, whenever a simulation objects is scheduled to execute, it must send the number of rollbacks it had so far to all other simulation objects, both local and remote ones. As a result, at any time *max_rollbacks* is the largest number of rollbacks among all the existing simulation objects.

By implementing LRFM and GRFM protocols in our optimistic PCD++ simulator, different simulation results can be collected since the RFM *period* (and in case of LRFM the *max_rollbacks*) can be modified very easily at the beginning of the simulation. This is done by changing these values in the configuration files right before the simulation starts and therefore, there is no need to rebuild the whole simulator in order for these modifications to have effect.

4. Testing models

In this section we introduce the description of different models we used to carry out the testing of the simulation engine, including a Synapsin-Vesicle Reaction at Nerve Terminal (which represents the interaction of synapsin with vesicles at nerve terminal), Fire Spread (illustrates fire propagation in a forest), and Ship Evacuation (an emergency ship evacuation scenario). We have run a variety of tests to analyze the performance of our existing PCD++ simulators; the optimistic and the conservative as well as our LRFM and GRFM Time Warp-based protocols.

4.1. Ship Evacuation Model

The first model we used represents an evacuation scenario of a ship under emergency [9]. The rules defining the model are based on the following restrictions:

(i) Each cell representing a person on the ship, calculates its shortest path toward the exit. During initialization phase, people are placed randomly in any empty cell.

(ii) People run in their initial direction until they encounter another person or an obstacle (e.g. wall).

At the end of simulation, there should be no one left on the ship. The neighborhood of each cell consists of 10 cells (i.e. they can be walls, exit doors, people, or empty cells) as shown on Fig. 6.

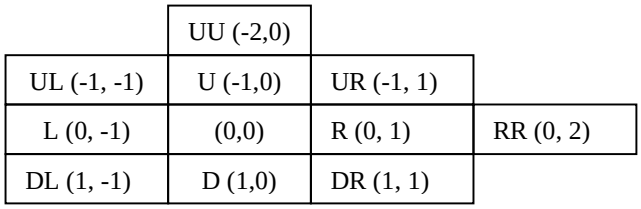


Fig. 6. Cell neighborhood

Each value on the cell space defines a distinct state, such as the type of the cell: wall, empty, exit door, a moving person. Also each type of movement is given a state value in order to identify the next position.

Table 1. State values and their description

Name	Value	Comments
N/A	0	Unknown Empty cell.
Wall	1	Represents an obstacle or a wall.

Name	Value	Comments
Exit	2	Represents an exit (e.g. stairs, door).
ED	3	Empty cell and its down (D) cell is the shortest path to the nearest exit.
ER	5	Empty cell and its right (R) cell is the shortest path to the nearest exit.
EU	7	Empty cell and its up (U) cell is the shortest path to the nearest exit.
EL	9	Empty cell and its left (L) cell is the shortest path to the nearest exit.
FD	4	Full cell (cell with person) and its down (D) cell is the shortest path to the exit.
FR	6	Full cell (cell with person) and its right (R) cell is the shortest path to the exit.
FU	8	Full cell (cell with person) and its up (U) cell is the shortest path to the exit.
FL	10	Full cell (cell with person) and its left (L) cell is the shortest path to the exit.

Based on these values, we define different rules for the movement of people in the vessel. The following rules initialize the model by calculating the shortest path. When a cell detects that one of its attached cells has changed its state to "defined", it would know that the attached cell is the shortest path.

Result	Precondition
3 or 4 (ED or FD)	(0,0) = Undefined and (1,0) is defined.
5 or 6 (ER or FR)	(0,0) = Undefined and (0,1) is defined.
7 or 8 (EU or FU)	(0,0) = Undefined and (-1,0) is defined.
9 or 10 (EL or FL)	(0,0) = Undefined and (0, -1) is defined.

The following rules define the case when a cell knows that a person will move towards it, which will occur if it is empty and it is the shortest path to at least one cell with a person occupying it.

Result	Precondition
4 → FD state	(0,0) = ED and ((0,1) = FL or (-1,0) = FD or (0,-1) = FR)
6 → FR state	(0,0) = ER and ((1,0) = FU or (-1,0) = FD or (0,-1) = FR)
8 → FU state	(0,0) = EU and ((1,0) = FU or (0,1) = FL or (0,-1) = FR)
10 → FL state	(0,0) = EL and ((1,0) = FU or (0,1) = FL or (-1,0) = FD)

The next rules define when a cell occupied with a person is attached to the exit. Then, the cell knows that a person will leave it and exit.

Result	Precondition
3→ ED state	(0,0) = FD and (1,0) is exit
5→ ER state	(0,0) = FR and (0,1) is exit
7→ EU state	(0,0) = FU and (-1,0) is exit
9→ EL state	(0,0) = FL and (0,-1) is exit

Finally, the next rules define when a cell knows that a person will leave it when it is not near an exit. A person will leave it when the cell is occupied by a person and its shortest path cell is empty. However, only one person can move to the empty cell when more than one person is trying to move to the same cell. In this case, the priority is first with the person who is in the upper cell, second the one in the right cell, third the one in the down cell, and finally the one in the left cell has the lowest priority.

Result	Precondition
3→ ED state	(0,0) = FD and down (D) cell is empty.
5→ ER state	(0,0) = FR and right cell (R) is empty and UR,RR, and DR cells have no person moving to R.
7→ EU state	(0,0) = FU and upper cell (U) is empty and UU and UR don't have a person moving to U.

Result	Precondition
9→ EL state	(0,0) = FL and left cell (L) is empty and UL doesn't have a person moving to L.

Fig. 7 shows an extract of the model's definition in CD++.

```
[top]
components : ship

[ship]
type : cell dim : (20,20) delay : transport
defaultDelayTime : 20 border : nowrapped

neighbors : (-2,0) (-1,-1) (-1,0) (-1,1) (0,-1)
neighbors : (0,0) (0,1) (0,2) (1,-1) (1,0) (1,1)
…
localtransition : ship-rule
[ship-rule]
rule : {3 + randInt(1)} 0 {(0,0)=0 and (1,0)>1 and (1,0)<11}
…
rule : 4  100 {(0,0)=3 and ((0,1)=10or(-1,0)=4 or (0,-1)=6)}
…
rule : 3 100 { (0,0) = 4 and (1,0) = 2}
…
rule : 3  100 { (0,0) = 4 and odd((1,0)) }
…
rule : {(0,0)} 100 { t }
```

Fig. 7. Definition of ship evacuation model in CD++

The ship evacuation model can be modified by adding more exit doors or changing the position of these cells. As presented in Fig. 8, four different types of cells appear on the grid: empty spaces, walls, people, and exit doors. The final result of the simulation shows no presence of people, i.e. the ship is evacuated.

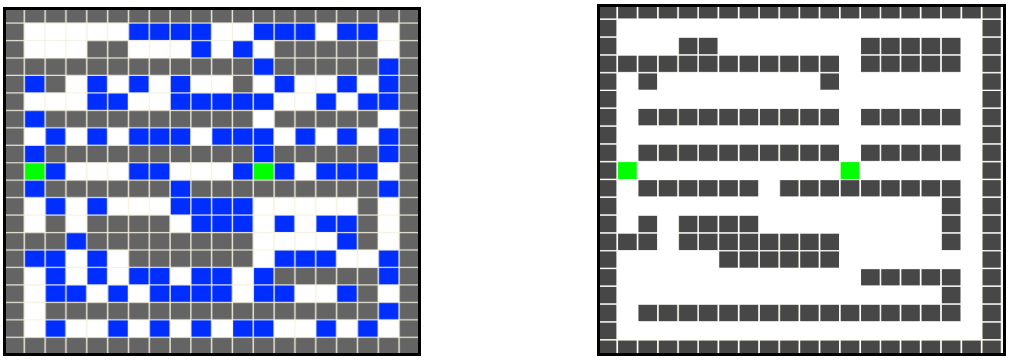


Fig. 8. Model Execution Results; initial values; final execution

4.2. Synapsin-Vesicle reaction Model

We built a model representing the reserve pool of synaptic vesicles in a presynaptic nerve terminal, predicting the number of synaptic vesicles released from the reserve pool as a function of time under the influence of action potentials at differing frequencies. Time series amounts for the components are obtained using rule-based methods (the rules defined by Cell-DEVS) [10].

Synapsin is a neuron-specific phosphoprotein that binds to small synaptic vesicles and actin filaments in a phosphorylation-dependent pattern. Microscopic models have demonstrated that synapsin inhibits

neurotransmitter release either by forming a cage around synaptic vesicles (cage model) or by anchoring them to the F-actin cytoskeleton of the nerve terminal [11].

We modeled the molecular interaction of *synapsin* (**S**) with *vesicles* (**V**) which occur inside a nerve cell. The model describes the behavior of synapsin movements until reaching a vesicle and binding to it. Once binding has occurred, depending on *offrate* V and S can again go apart and break their bindings. The *onrate* and *offrate* describe how often bindings occur or break then after. The following formula describes the nature of the reaction:

$$S + V \leftrightarrow SV$$

From the above formula, the left hand side of the equation demonstrates the binding scenario where *synapsin* and *vesicles* perform a bind at a rate specified by *onrate*, while the right hand side of the equation illustrates the bind-break scenario where an *synapsin-vesicle* at an *offrate* which is always smaller than *onrate* breaks apart and again *synapsin* and *vesicles* get released. Then, *synapsin* and *vesicles* can again perform binding and break apart then after. This equation shows an on-going process of "binding" and "breaking apart" which depends on *offrate/onrate*. The larger the *offrate* is, the more bindings get broken apart. Similarly, the larger the *onrate* is, the more V-S binds are produced. Three different scenarios are modeled: 1) V is stationary (with a fixed position on cell space), and S is mobile, 2) V is mobile and S is stationary, and 3) V and S are both mobile (leads to maximum number of total movements and therefore bindings).

The coupled Cell-DEVS model for this application is described as follows.

$$M=<I,X,Y,Xlist,Ylist,\eta, N,\{m,n\}, C, B, Z, select>$$

Xlist=Φ Ylist=Φ η=9 I=<P^X,P^y>,with P^X={Φ},P^y={Φ};

N={ (-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1)(1,0) (1,1)};
X=Y={0,1,2,11,12,13,14,21,22,23,24,31,32,33,34,41,42,43,44};

m=26; n=22; B={Φ}; C={Cij/iε[1,26], jε[1,22]}

select ={ (-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1), (1,0), (1,1) };
Z is defined by Cell-DEVS specifications.

The cell space, the value 1 was used to represent V, and the value 2 was used to represent S. The number 0 represents an empty cell for which a mobile S can occupy. To give direction to the V (although the model assumes fixed V) or S, a two digit number was used. For example, the following represent:

11	"up" moving V	21	"up" moving S
12	"right" moving V	22	"right" moving S
13	"down" moving V	23	"down" moving S
14	"left" moving V	24	"left" moving S

As we can see, Cell-DEVS provides great support for defining these models, for having independent cell states and random mobility of cells, provide an excellent environment to simulate the process of synapsin-vesicles interactions of a nerve. As mentioned earlier, the model constructed can be further extended to include the movement of both synapsin (S) and vesicles (V) as well as defining different off and on rates. Aside from V-S reactions, the model can also include *Actins, which* bind to *synapsins*. Actins can be represented as a string of cells being fixed at their cell space position. A summarized version of the model's definition in CD++ is as follows:

```
[top]
components : chemCell

[chemCell]
type : cell dim : (26,22)  delay : transport
defaultDelayTime : 100    border : wrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1) (0,0) (0,1)
neighbors :  (1,-1)  (1,0)  (1,1)
localtransition : chemCell-rule

[chemCell-rule]
rule : {round(uniform(11,14))} 100 { (0,0) = 1  }
…
rule : {round(uniform(31,34))} 100 {((0,0)=21 or (0,0)=22 or
(0,0)=23 or (0,0)=24) and(((-1,0)-10=1 or (-1,0)-10=2 or…}
…
%moving up
rule : 91 100 {(0,0)=21 and (-1,0)=0 and t}
rule : {round(uniform(21,24))} 0 {(0,0)=0 and (1,0)=91 }
…
%release 0.1 of the S (the offrate is 0.1)
rule : {round(uniform(21,24))} 100 {((0,0)=33 or (0,0)=32 or
(0,0)=31 or (0,0)=34) and random < 0.10}
…
rule : { (0, 0) } 100 { t}
```

Fig. 9. Synapsin-Vesicle Reaction model in CD++

The following rules initialize cells with 11-14 (for Vesicles) and 21-24 (for Synapsin), where bindings have not yet been performed.

```
rule : {round(uniform(11,14))} 100 { (0,0) = 1  }
rule : {round(uniform(21,24))} 100 { (0,0) = 2  }
```

Once bindings occur, cells change their values; 11-14 get replaced with 31-34, and 21-24 get replaced with 41-44. Also for Synapsins, four intermediate values 91-94 are used to represent a moving cell that has not yet being settled down. Once it settles down its value changes back to 21-24 (depending on its direction of movement) and gets ready to bind to a vesicle in its neighborhood.

```
rule : {round(uniform(31,34))} 100 {((0,0)=21 or(0,0)=22 or(0,0)=23 or (0,0)=24) and
  (((-1,0)- 10 = 1 or (-1,0)- 10 = 2 or (-1,0)- 10 = 3 or (-1,0)- 10 =4 )or
  ((1,0)- 10 = 1 or (1,0)- 10 = 2 or (1,0)- 10 = 3 or (1,0)- 10 = 4)     or
  ((0,-1)- 10 = 1 or (0,-1)- 10 = 2 or (0,-1)- 10 = 3 or (0,-1)- 10 = 4) or
  ((0,1)- 10 = 1 or (0,1)- 10 = 2 or (0,1)- 10 = 3 or (0,1)- 10 = 4 )    or
  ((-1,1)- 10 = 1 or (-1,1)- 10 = 2 or (-1,1)- 10 = 3 or (-1,1)- 10 = 4) or
  ((1,-1)- 10 = 1 or (1,-1)- 10 = 2 or (1,-1)- 10 = 3 or (1,-1)- 10 = 4) or
  ((1,1)- 10 = 1 or (1,1)- 10 = 2 or (1,1)- 10 = 3 or (1,1)- 10 = 4)     or
   ((-1,-1)- 10 = 1 or (-1,-1)- 10 = 2 or (-1,-1)- 10 = 3 or (-1,-1)- 10=4))
   and random > 0.10}
```

The above rule describes the following scenario: if there is a synapsin having the value 21, 22, 23, or 24 (a synapsin that can move up/right/down/left) and there is a vesicle in its neighboring which could be an adjacent cell or a diagonal cell, then the synapsin (red cells) will move toward this vesicle and a binding will occur soon, the value of the synapsin gets changed to 31, 32, 33, or 34 (i.e. 21 changes to 31, 22 changes to 32, 23 changes to 33, and 24 changes to 34) to represent a synapsin that is bonded to a vesicle.

```
rule : {round(uniform(41,44))} 100 {((0,0)=11 or (0,0)=12 or (0,0)=13 or (0,0)=14) and
( ((-1,0)- 30 = 1 or (-1,0)- 30 = 2 or (-1,0)- 30 = 3 or (-1,0)- 30 = 4)  or
((1,0)- 30 = 1 or (1,0)- 30 = 2 or (1,0)- 30 = 3 or (1,0)- 30 = 4)        or
((0,-1)- 30 = 1 or (0,-1)- 30 = 2 or (0,-1)- 30 = 3 or (0,-1)- 30 = 4)    or
((0,1)- 30 = 1 or (0,1)- 30 = 2 or (0,1)- 30 = 3 or (0,1)- 30 = 4 )       or
```

```
((-1,1)- 30 = 1 or (-1,1)- 30 = 2 or (-1,1)- 30 = 3 or (-1,1)- 30 = 4)  or
((1,-1)- 30 = 1 or (1,-1)- 30 = 2 or (1,-1)- 30 = 3 or (1,-1)- 30 = 4)  or
((1,1)- 30 = 1 or (1,1)- 30 = 2 or (1,1)- 30 = 3 or (1,1)- 30 = 4)      or
((-1,-1)- 30 = 1 or (-1,-1)- 30 = 2 or (-1,-1)-30 = 3 or (-1,-1)-30=4)) and random > 0.10}
```

Similarly, the above rule describes the case with a vesicle having the value 11, 12, 13, or 14 (a vesicle that can move up/right/down/left) and a synapsin in its neighborhood. Then, since the synapsin will come toward this vesicle and a binding will occur soon, the value of the vesicle gets changed to 41, 42, 43, or 44 (i.e. 11 changes to 41, 12 changes to 42, 13 changes to 43, and 14 changes to 44).

For the movement of synapsin the following four rules are implemented:

```
rule : 91 100 {(0,0)=21 and (-1,0)=0 and t}
rule : {round(uniform(21,24))} 0 {(0,0)=0 and (1,0)=91 }
rule : 00 0 {(0,0)=91}
```

step 1: checking to see if there is an empty cell so the synapsin can move into it, for example if the synapsin's direction is upward (value = 21), then at first we need to check if there is an empty cell right above it. (91 is used as an intermediate value to occupy the empty cell)

step 2: once an empty cell is found, it gets occupied by the synapsin (i.e. the cell's value changes from 0 to a random number 21-24).

step 3: the previous position of the synapsin that just moved to an empty cell gets cleared by setting the value of the cell to 0.

The same procedure is used for right, left, and down movement. The following rule is used to break the S-V bindings using an offrate=0.10. According to this, 10% of the bindings get broken and synapsins get released to be given another direction and they will move around until finding a vesicle and binding to it.

```
%release 0.1 of the S (the offrate is 0.1)
rule : {round(uniform(21,24))} 100 {((0,0)=33 or (0,0)=32 or (0,0)=31 or (0,0)=34) and
random < 0.10}
```

Fig. 10 shows the grid at the initial case where S and V have not yet interacted. Then, Fig. 11 shows how bounds are formed and the corresponding cells change their values to represent the binding.

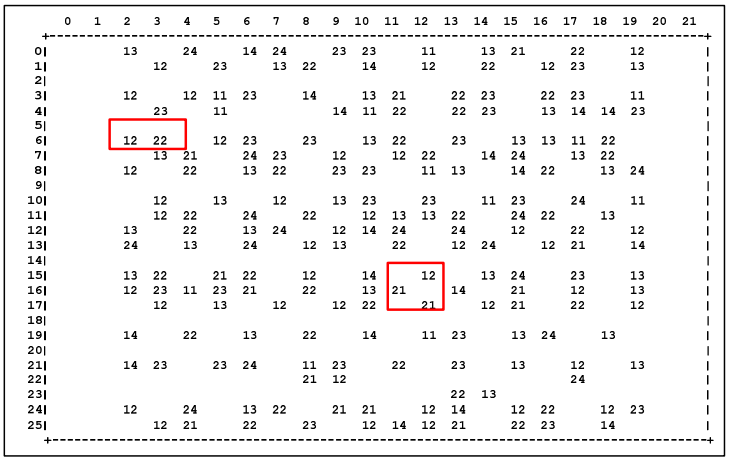


Fig. 10. V and S before binding at Time: 00:00:00:100 (bold boxes represent examples of binding structures)

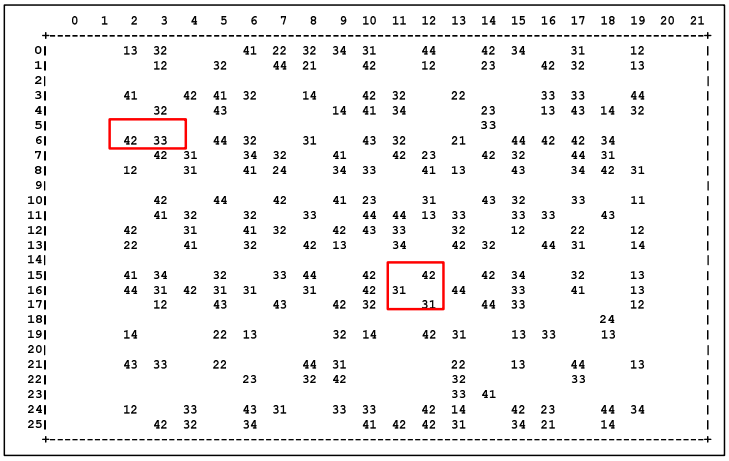
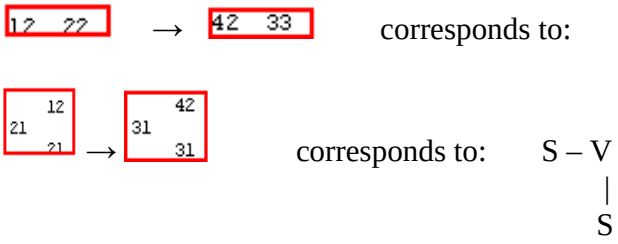


Fig. 11. V and S after binding at Time: 00:00:00:300

As illustrated above, the bold boxes show bindings between synapsin (31-34) and vesicle (41-44). The first illustration (Fig. 10) represents the initial scenario where synapsins (21-24) and vesicles (11-14) are free and have not yet performed bindings. Once synapsins move toward vesicles, the values of the corresponding cells change to 31-34 (bonded synapsins) and 41-44 (bonded vesicles). Vesicles can be surrounded by more than one synapsin, but each synapsin can bind to only one vesicle at any time. From the above figure we can see the following possible binding scenarios:



5. Experiments and Performance Analysis

The main goal of this section is to show the capability of PCD++ in terms of handling the number of nodes driving the simulation, complexity of the model, and the size of the model. We have selected different models with distinguishable functionality, complexity, and size to better judge the capability of the simulators. Our experiments were carried out on a HP PROLIANT DL Server, a cluster of 32 compute nodes (dual 3.2GHz Intel Xeon processors, 1GB PC2100 266MHz DDR RAM) running Linux WS 2.4.21 interconnected through Gigabit Ethernet and communicating over MPICH 1.2.6.

Each Cell-DEVS model consists of a number of necessary and optional files grouped together in a package. Since the simulation can be distributed among 1 to 32 nodes of the cluster, we used a partitioning mechanism implemented earlier in [17,18] which evenly divides the cell space into horizontal rectangles. Different partitioning strategies can be implemented which in return result in a significant impact on the performance of the simulation.

5.1. Performance Metrics

The *total elapsed time* value was collected from the execution environment to measure the performance of the simulators in terms of execution time. Also, the speedups with respect to changing the number of simulating nodes were calculated to show how the parallel simulation outperforms the sequential one (using only one node). The *overall speedup* for N nodes is given as follows.

$$\text{Overall Speedup} = \frac{T(1)}{T(N)}$$

Where T (1) represents the serial execution time measured on one node, and T (N) is the total execution time taken by the simulation running on N nodes. Each of the models which were presented in Chapter 5 is executed on four different simulators:

- The optimistic PCD++ simulator [18];
- The conservative PCD++ simulator [17];
- The optimistic PCD++ simulator implementing LRFM protocol; and
- The optimistic PCD++ simulator implementing GRFM protocol.

The goal is to identify the execution performance of each simulator as we increase the number of participating nodes. Due to the partitioning mechanism that is used by our optimistic and conservative simulators, we can only increase the number of nodes to a certain limit. That is, the maximum number of nodes that a model can be simulated on is equal to the number of rows of the cell gird for that particular model. For instance, if we have a model of 400 cells arranged in a 20x20 mesh, we can run the model on 1 to 20 nodes. In order to obtain stable results, for each model, simulations were run on 1 to N nodes and for each scenario five trials were collected. The execution results which will be presented in the next section reflect the average of these five trials which are within a confidence interval of 95%.

5.2. Simulation Results

In the following points we will present the simulation results of executing our models.

- Ship Evacuation Model

This model consists of 400 cells arranged in a 20x20 mesh with a total execution time of 6.4327 seconds when run on standalone CD++. Fig. 12 represents the execution time resulting from running the model with four different simulators on 1 to 8 nodes.

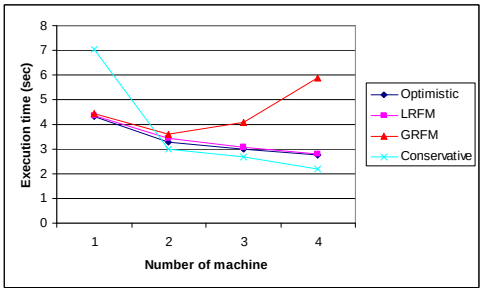


Fig. 12. Ship evacuation model execution time on 4 different simulators

From the execution time graph, we can see that the conservative simulator outperforms the other three simulators. This is due to the causality-error avoidance mechanism of this simulator which avoids rollbacks and anti-message flows. The optimistic and LRFM-based simulators produce very similar results for 2 to 6,

and 8 nodes. However, the GRFM-based simulator does not present good results. This is mainly due to the huge message-passing mechanism among the LPs who are sending messages back and forth reporting information about their rollbacks. To prove this, we can see that the GRFM-based simulator reduces the execution time when there are two computing nodes, but as the number of nodes increases, the performance degrades.

- Synapsin-Vesicle Reaction Model

This model consists of 676 cells arranged in a 26x26 mesh with a total execution time of 3.7621 seconds when run on standalone CD++. Fig. 13 represents the execution time resulting from running the model with four different simulators on 1 to 8 nodes.

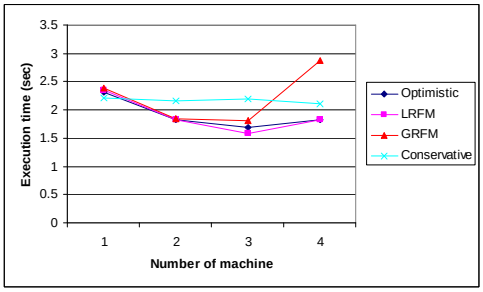


Fig. 13. Synapsin-vesicle model execution time on 4 different simulators

We can see that the optimistic and LRFM-based simulators produce very close results on 1 to 8 nodes. Also, the GRFM-based simulator has similar results for 1, 2, 3, and 5 nodes. However, it degrades the performance when 4, 6, 7, and 8 nodes are participating. On the other hand, the conservative simulator shows different behavior as the number of nodes increases. The conservative simulator improves the total execution time significantly when more than 2 nodes are available. Again, as in the previously discussed models, as the number of computing nodes increases, the GRFM-based simulator has the lowest performance among other ones. The main reason is communication overhead among the participating LPs which leads in a noticeable time added to the duration of the model execution.

- Game of Life Model

This model consists of 1200 cells arranged in a 30x40 mesh with a total execution time of 4.6723 seconds when run on standalone CD++. The Game of Life was created by mathematician John Conway in 1970 [19]. It is the best-known example of cellular automata algorithms. The standard Game of Life uses a two-dimensional grid. We will use this simple example to show the basic facilities of CD++ to define model's rules. Cells can be either on (alive) or off (dead). The key rule is known as "B3/S23": a new cell is born when it has exactly 3 neighbors; an existing cell (alive cell) survives if it has 2 or 3 neighbors. In all other cases the cell dies, either of overcrowding (with more than three live neighbors) or loneliness (with less than two). At each time step all cells update their state simultaneously. We have modeled the Game of Life using CD++, on a 20x20 cell grid (400 cells). Fig. 14 illustrates the cell gird at four different time stamps of the simulation. The first cell grid shows the initial scenario where seventeen alive cells exist. As the simulation proceeds, either new cells are born or live cells die (based on the "B3/S23" rule).

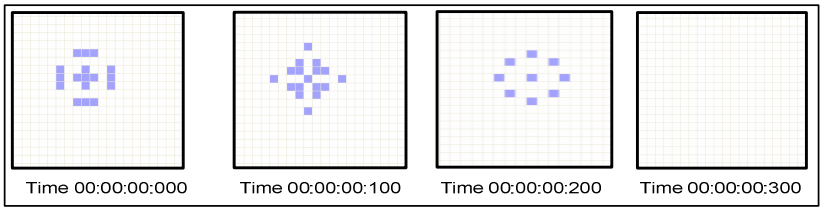


Fig. 14. Game of life model at four different time steps throughout the simulation

Fig. 15 represents the execution time resulting from running the model with four different simulators on 1 to 6 nodes. From the execution time graph, we can see that the optimistic, LRFM-based, and GRM-based simulators outperform the conservative one on 1 to 6 nodes and at the same time produce very close results. However, as the number of machines goes beyond 3, the conservative simulator starts dropping down the execution time. Among the three optimistic simulators, the GRFM-based simulator takes longer time due to its time consuming mechanism in broadcasting information about each LP's rollbacks among the participating nodes.

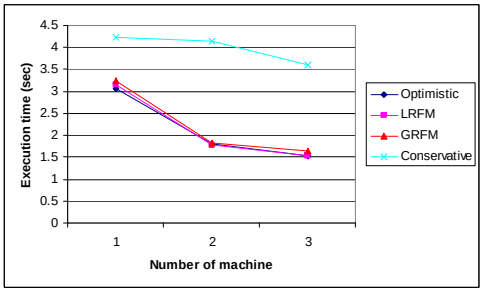


Fig. 15. Game of life model execution time on 4 different simulators

- Fire Propagation Model

This model consists of 900 cells arranged in a 30x30 mesh with a total execution time of 6.2145 seconds when run on standalone CD++. This model represents a fire propagation scenario in forest based on Rothermel's mathematical definition [20]. The model computes the ratio of spread and intensity of fire in forest based on specific environmental and vegetation conditions. Three parameter groups determine the fire spread ratio: 1) vegetation type (caloric content, mineral content and density); 2) fuel properties; 3) environmental parameters (wind speed, humidity, and field slope).

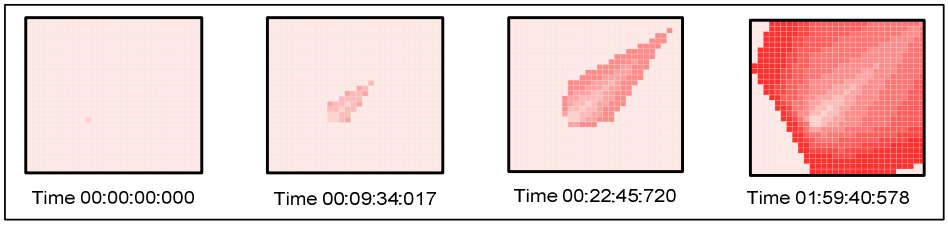


Fig. 16. Fire propagation at four different snapshots throughout the simulation

Fig. 16 illustrates snapshots of the simulation results at four different times. Initially, fire starts as fire spot (the dark cell on the grid). Then as time passes by, fire spreads to the neighboring cells in the direction of wind. Therefore, each cell, depending on its position and heat, fires its surrounding cells. As presented on

the final scenario of Fig. 16, the wind direction leads the fire from the starting point, cell (19, 10), towards southeast of the forest. Fig. 17 represents the execution time resulting from running the model with four different simulators on 1 to 8 nodes.

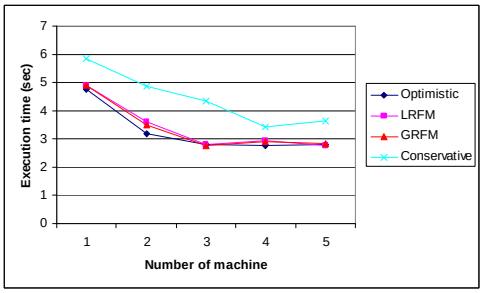


Fig. 17. Fire propagation model execution time on 4 different simulators

As seen on the graph, our parallel simulators significantly improved the execution time of the fire propagation model. The three optimistic simulators produced very similar results on 1 to 7 nodes. For this model, we can definitely remark that the optimistic simulators outperform the conservative one. For the optimistic simulators the best results were achieved on 5 nodes, while the conservative one had its lowest execution time on 4 nodes.

6. Conclusions

This work presented the parallel simulation of DEVS and Cell-DEVS models using PCD++, a parallel and distributed environment based on the Time Warp optimistic synchronization protocol. PCD++ serves as an extension to the CD++ toolkit which was developed by previous researcher [18] aiming at exploiting parallelism for the purpose of fast and efficient simulation of complex models. The concept of Parallel and Distributed Simulation was presented.

We illustrated the software architecture of the purely optimistic parallel CD++ simulator (PCD++). The layered architecture of the optimistic PCD++ simulator consists of five layers (from top to bottom): model, PCD++, Time Warp - WARPED, and the operating system, where ach layer was explained in details. A variety of optimization strategies of the Time Warp kernel were pointed out and discussed thoroughly. Some optimizations in terms of GVT calculation, dynamic memory management, and state management were mentioned.

We have analyzed the performance of our two existing parallel CD++ simulators, namely Conservative PCD++ simulator [17] and Optimistic PCD++ simulator [18]. We looked at the design and implementation of these two simulators and compared their structures as well as functionalities in parallel and distributed simulations.

The hierarchical structure of the conservative PCD++ simulator was compared against the flattened structure of the optimistic PCD++ simulator. The migration from a hierarchical structure to a flattened structure was illustrated as two major modifications; i.e. the departure from conservative-based simulator to an optimistic-based simulator, and flattening the structure of the simulator. Then it was illustrated how the optimistic PCD++ simulator deals with the communication overhead dilemma by using the flattened structure.

Aiming at improving the performance of the optimistic simulator, we modified the WARPED kernel to handle rollbacks in a more efficient way. We presented two new algorithms that we have implemented in

WARPED kernel. The *Near-perfect State Information* protocol was discussed and after that our new algorithms; Local Rollback Frequency Model (LRFM) and Global Rollback Frequency Model (GFRM) were presented. Finally, we have run a variety of tests to analyze the performance of our existing PCD++ simulators; the optimistic and the conservative as well as our LRFM and GRFM Time Warp-based protocols [21].

7. References

[1] Zeigler, B.; Kim, T.; Praehofer, H. “Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems”. Academic Press. 2000.

[2] Wainer, G.; Giambiasi, N. “Timed Cell-DEVS: modeling and simulation of cell spaces “. In “Discrete Event Modeling & Simulation: Enabling Future Technologies”, Springer-Verlag. 2001.

[3] Martin, D. E.; McBrayer, T. J.; Radhakrishnan, R.; Wilsey, P. A. “WARPED – A Time Warp Parallel Discrete Event Simulator (Documentation for version 1.0)”.

[4] Jefferson, D. “Virtual Time”. ACM Transactions on Programming Languages and Systems. 7(3):405-425. 1985.

[5] Wainer, G. “CD++: a toolkit to develop DEVS models”. Software – Practice and Experience. Vol. 32, pp. 1261-1306. 2002.

[6] Wainer, G. "Improved cellular models with Parallel Cell-DEVS". Transactions of the Society for Computer Simulation International. Vol. 17, No. 2, pp. 73-88. 2000.

[7] Fujimoto, R. M. “Parallel and Distributed Simulation Systems”. Wiley-Interscience. ISBN 0-471-18383-0. 2000.

[8] Bryant, R.E. Simulation of Packet Communication Architecture Computer Systems. Massachusetts Institute of Technology, Cambridge, MA. USA. 1977.

[9] Klüpfel, W.; Meyer-König, T.; Wahle, J.; Schreckenberg, M. “Microscopic Simulation of Evacuation Processes on Passenger Ships”, Theoretical and Practical Issues in Cellular Automata, pp. 63-71, Springer-Verlag 2001.

[10] Al-Aubidy, B.; Dias, A.; Bain. R.; Jafer, S.; Dumontier, M.; Wainer, G.; Cheetham, J. ”Advanced DEVS models with applications to biology”. *Artificial Intelligence, Simulation and Planning*. Buenos Aires, Argentina. 2007.

[11] Benfenati, F., Valtorta, F., Greengard, P.; “Computer Modelling of Synapsin 1 Binding to Synaptic Vesicles and F-actin”. Implications for Regulation of Neurotransmitter Release. 1990.

[12] Gropp, W.; Lusk, E.; Doss, N.; Skjellum, A. “A high-performance, portable implementation of the MPI message-passing interface standard”. Parallel Computing. Vol. 22, pp. 789-828. 1996.

[13] Glinsky, E. “New Techniques for Parallel Simulation of DEVS and Cell-DEVS Models in CD++”. M. A. Sc. Thesis. Carleton University. Canada. 2004.

[14] Radhakrishnan, R.; Martin, D. E.; Chetlur, M.; Rao, D. M.; Wilsey, P.A. “An Object-Oriented Time Warp Simulation Kernel”. Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE’98). Vol. LNCS 1505, pp. 13-23. Springer-Verlag. 1998.

[15] Szulsztein, E.; Wainer, G. “New Simulation Techniques in WARPED Kernel” (in Spanish). Proceedings of JAIIO, Buenos Aires, Argentina, 2000.

[16] Srinivasan, S.; Reynolds, J., “Elastic Time”, ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 2. 103-139. April 1998.

[17] A. Troccoli, G. Wainer. "Implementing Parallel Cell-DEVS". In *Proceedings of 36th IEEE/SCS Annual Simulation Symposium.* Orlando, FL. U.S.A. 2003.

[18] Q. Liu, G. Wainer. “Optimistic Simulation of DEVS and Cell-DEVS Models with PCD++”. Accepted for publication in *Simulation, Transactions of the SCS.* Accepted: May 2007.

[19] Gardner, M. " Mathematical games: The fantastic combinations of John Conway's new solitaire game "life"". *Scientific American* 223: 120 - 123. 1970.

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

[21] S. Jafer, G. Wainer. "An infrastructure for computational simulation of cellular models". In *Proceedings of Unconventional Computing 2007 Computational Problems Workshop*. Kingston, ON, Canada (2007).