Conservative DEVS - A Novel Protocol for Parallel Conservative Simulation of DEVS and Cell-DEVS Models

Shafagh Jafer, Gabriel Wainer
Dept. of Systems and Computer Engineering
Carleton University Centre of Visualization and Simulation (V-Sim)
1125 Colonel By Dr. Ottawa, ON, Canada
{sjafer,gwainer}@sce.carleton.ca

Keywords: Conservative DEVS, parallel DEVS protocol, Cell-DEVS models, DEVS models, dynamic look-ahead.

Abstract
We present a novel conservative algorithm based on the classical Chandy-Misra-Bryant (CMB) synchronization mechanism by extending DEVS abstract simulator to provide means for look-ahead computation and null-message distribution. We integrate this mechanism into the CD++ simulation toolkit, providing a purely conservative simulator for running large-scale DEVS and Cell-DEVS models. Our algorithm is implemented on a revised DEVS abstract simulator to reduce the frequency of look-ahead computation. It also replaces time information estimations with a single lookahead computation, causing reduction in the number of null-messages. The dynamic lookahead values of the proposed algorithm are extracted from the model specification and the user is not required to provide lookahead values prior to the execution. In addition, the low-cost lookahead computation feature of the algorithm provides a fast and efficient method and reduces overhead.

1. INTRODUCTION

Modeling and simulation (M&S) methodologies have become crucial for implementing, designing, and analyzing a broad variety of systems. Among the existing modeling and simulation techniques, the DEVS (Discrete Event System Specification) formalism [1] provides a discrete-event approach to construct hierarchical models. This feature of DEVS enables the modeler to easily extend or expand the model by simply creating new components or duplicating the existing ones. P-DEVS [2], allows adequate handling of simultaneous events, which is needed for efficient execution of models in parallel and distributed environments. Cell-DEVS [3] combines DEVS with Cellular Automata [4] to form an n-dimensional cell space to represent complicated discrete event spatial models. As the models become larger and more complex, the problems of limited resources within a single-processor arise. Not only the shortage of resources, but also the long execution times brought up the idea of Parallel discrete event simulation (PDES). Synchronization, as the key to parallel and distributed simulation, requires a robust mechanism to handle communication among concurrent processes. There are two major classes of synchronization: conservative approaches, which strictly avoid causality violations [5] and optimistic approaches, which allow violations and recover from them by providing rollback mechanism [6]. Conservative synchronization approach was the first synchronization algorithms proposed in the late 1970s by Bryant [7], Chandy and Misra [8]. This technique (known as the Chandy-Misra-Bryant (CMB) algorithm) prevents any occurrence of causality errors.

CD++ [9] is an M&S toolkit that implements DEVS and Cell-DEVS theories. PCD++ [10] supports optimistic parallel simulation of DEVS and Cell-DEVS models based on the Time Warp mechanism. In this article, we present a Conservative DEVS algorithm, implemented based on the classical CMB synchronization mechanism with deadlock avoidance. Our conservative algorithm extends the DEVS abstract simulator to provide means for look-ahead computation null-message distribution. Various issues related to performance, scalability, and complexity of the optimistic simulation algorithms motivated us to implement the first purely conservative simulator for Cell-DEVS, called Conservative CD++ (CCD++). CCD++ uses our Conservative DEVS synchronization algorithm.

2. RELATED WORK AND MOTIVATION

PDES synchronization techniques usually fall into two major categories: conservative and optimistic approaches. In conservative schemes, a simulation entity (called a logical process - LP) does not allow causality errors. If the LP has an unprocessed event with timestamp *t* it guarantees that no event with earlier timestamp can be received (which prevents causality errors). As long as there are unprocessed events from all other LPs, synchronization is guaranteed. However, if this condition is not met, deadlock may occur. A way to resolve deadlock is to find the model's *lookahead*, which provides the smallest timestamp of any new events the LP can schedule in the future. *Null messages* are used to share lookahead information among LPs [8].

Numerous algorithms and tools have been built, including varied parallel DEVS simulators (for instance, DEVS-C++ [11], DEVS/CORBA [12], DEVSCluster [13], DEVS/P2P [14], DEVS/RMI[15], DEVSim++[16], and P-DEVSim++ [17]). In [18] the authors presented a distributed

simulation strategy for DEVS, which combines conservative and risk-free optimistic strategies. This optimistic PCD++ is purely optimistic, and based on the Time Warp mechanism. Likewise, in [1], the authors introduced an approach to conservative parallel simulation of DEVS, which is mainly based on the classical CMB approach with deadlock avoidance and the Yaddes [19] algorithm. The principal idea behind this conservative DEVS simulator is to maintain a network of correlated earliest output time (EOT) and earliest input time (EIT) estimates, which matches the output-to-input coupling structure of the DEVS coupled model. The EOT/EIT estimates represent the time information that is distributed with null-messages. Under this scheme, the lookahead calculation is performed at each DEVS simulator (representing an atomic DEVS component) by looking at input and output ports. In fact, the conservative mechanism is implemented at the lowest level of the abstract simulator hierarchy and the coordinator sitting at the top of the hierarchy is only responsible for distributing the EIT and EOT information. Two main limitations of this algorithm are:

1. A large number of EIT and EOT computations are required to implement the algorithm at the simulator level (the overhead of the algorithm increases as the DEVS model grows, since there must be one DEVS simulator for every DEVS atomic component).
2. Large numbers of null-messages are sent among processors, since both EIT and EOT must be distributed, as opposed to sending only one type of information (i.e. only lookahead).

The need for a robust conservative simulator and the two limitations of the original conservative DEVS algorithm led us to propose a new representation of the conservative DEVS algorithm and the first purely conservative DEVS and Cell-DEVS simulator. Conservative CD++ (CCD++), was implemented at the topmost level of the DEVS abstract simulator hierarchy (i.e. the coordinator); thus, the frequency of information computation is reduced. In addition, EIT/ EOT calculations are replaced with a single lookahead computation, reducing in the number of null-messages.

CCD++ was built on top of WARPED [20], which provides services for defining different types of simulation objects on top of the Message Passing Interface (MPI). WARPED was chosen, as it allows comparing its numerous optimistic algorithms with conservative performance.

3. CONSERVATIVE SIMULATION IN CCD++

CCD++ adopts a flat architecture with four DEVS processors [21]: Simulator, Flat Coordinator (FC), Node Coordinator (NC), and Root (Figure 1). The simulation is message-driven and managed by a set of NCs running on different machines synchronizing through null-messages. CCD++ processors exchange messages carrying content or synchronization information. The former includes the *external* (x, t) and *output* messages (y, t), while the latter includes the *initialization* (I, t), *collect* (@, t), *internal* (*, t), and *done* messages (D, t).

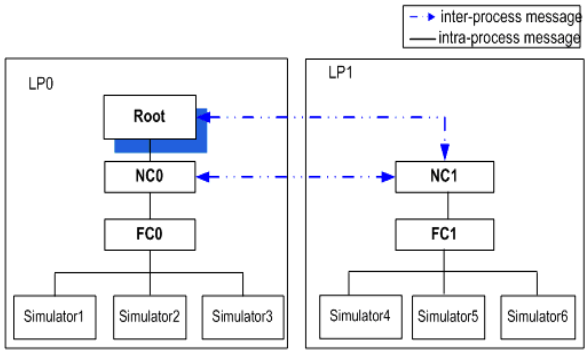

Figure 1. Distributed flat structure of CCD++

Figure 1 illustrates the Processors hierarchy. *Root* is created only on LP0 (to start/end the simulation and perform I/O operations). NC and FC are created on each LP. FC is in charge of intra-LP communications between its child Simulators. NC is the local central controller on its LP and the end of inter-LP communications. *Simulator* executes the DEVS functions defined in its atomic model.

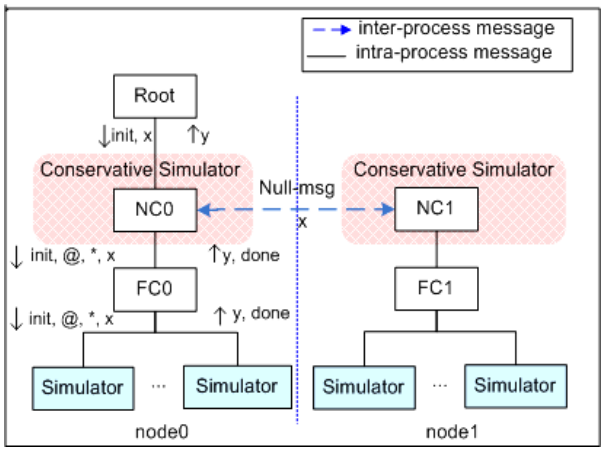

Figure 2. Structure of the parallel abstract simulator

Our conservative algorithm is mainly based on the original CMB approach with deadlock avoidance. The key focus is on how to compute lookahead and send it via null-messages and when to suspend the LP or resume. The algorithm is implemented at the NC and thus the simulators are unaware of its existence (the NC is responsible for lookahead calculation and sending null-messages). Figure 2 illustrates the proposed parallel abstract simulator.

After initialization, the execution of messages on an LP is either one of two distinct phases: The *collect* phase, and the *transition* phase. The collect starts with a *collect* message sent from the NC to the FC and ends with the following *done* message received by the NC. The transition phase

begins with the first internal message sent from the NC to the FC, and ends at the last done message received by the NC at that time. The transition phase is mandatory for each individual simulation time. The output functions in the imminent atomic models are invoked during the collect phase; the state transitions for the atomic models are performed in the transition phase (as defined in P-DEVS). Outgoing inter-LP communication happens only in the collect phases, whereas incoming inter-LP communication can occur in any phase. Since the output functions of the imminent models are invoked only in the collect phases, it is clear that at any given simulation time, all external messages going to remote NCs are sent out by the end of the collect phase. On the other hand, an external message from a remote source can arrive at the destination NC during any phase.

The NC is the actual starter for every collect and transition phase, and this is why our conservative algorithm is implemented at the NC. The algorithm is invoked every time the NC receives a *done* message from the FC, which could be in response to messages (I, t), (@, t), or (*, t) previously sent to the FC. The lookahead computation is thus:

lookahead = MIN(*timestamp of the x msg recently sent to a remote LP, time of the NC Message Bag,* t_N) (1)

where t_N is the closest state transition time given by the FC in the *done* message, and *time of the NC Message Bag* is the minimum timestamp of all unprocessed *x* messages received from other NCs. The NC then sends a null-message carrying the lookahead time to all remote NCs. The lookahead describes when other LPs should expect an external message from this LP. Thus, the sender LP is blocked waiting for other LPs to send their lookahead values. When the LP receives all remote null-messages, the LP resumes. Now the NC is responsible for driving the rest of the simulation by deciding if a transition or a collect phase has to be performed. If the NC decides to send an internal message (*, t), then normal procedure takes. However, if the NC detects that a collect phase must be issued, rest of the conservative algorithm is carried out (mainly because simulation time is advanced during collect phase and this is when the synchronization is required). First, the NC recalculates the Local Virtual Time (LVT) of the LP, which could result in advancement of the current time of the LP. The new LVT will be the minimum value among:

(i) the timestamp of the first not-yet-sent external event in the event list as referred by the event-pointer;
(ii) the timestamp of *external* message recently sent to a remote LP;
(iii) the time of the NC Message Bag;
(iv) the minimum lookahead value from remote LPs;
(v) the closest state transition time given by the FC in the *done* message.

LVT = MIN(*timestamp of the event pointed by event-pointer, timestamp of the x msg recently sent to a remote LP, time of the NCMessageBag, minimum RemoteLookahead,* t_N). (2)

After this, according to the new LVT the NC chooses, one of the following five cases must be performed:

1. If there are input messages received from outside environment to the system, the NC issues an internal phase and sends a (*, t) message to the FC.
2. If there are external messages sent to remote LPs at the recent collect phase, the NC must send its current lookahead and block the LP, because the LP receiving the external message could generate a smaller lookahead than that of its previous stage. This will cause causality violations at this LP because it had updated its LVT based on a larger lookahead and when it receives the new lookahead which happens to be smaller than before, then the new LVT turns out to be smaller than the old LVT and this is strictly forbidden by the definition of conservative synchronization mechanism.
3. If the new LVT is equal to the minimum of all lookahead values received from other LPs, the LP must wait. Therefore, the NC recalculates the lookahead, sends null-messages, and the LP is suspended.
4. If there are imminent children (Simulators) a collect phase is issued by sending a (@, t) message to the FC.
5. If there are external messages received from other LPs with receive time equal to the new LVT, the NC issues an internal phase and sends a (*, t) message to the FC.

These cases have processing priority and the NC only processes one of them every time. Special tie breaking is performed when more than one case is true which is discussed in Section 3.1.

As presented by Formula (1), our lookahead computation is a fast, efficient, and low-cost method that involves a simple comparison of existing model parameters. Compared to other existing conservative mechanisms, this reduces the overhead of the algorithm especially when the frequency of lookahead computation increases as the model size grows. Likewise, the modeler is not required to specify the lookahead of the system. Thus, the ability of the algorithm in dynamically extracting the lookahead information from the model itself stands as a remarkable point.

3.1. Revised DEVS Abstract Simulator

The abstract simulator implemented in CCD++ is based on a revised version [10] of the P-DEVS abstract simulator [2]. Herein, we present our modifications to the NC structure, which reflect the new conservative NC. The rest of the abstract simulator remains unchanged from the one presented in [10]. Figure 3 illustrates the description of the conservative NC functionalities when a (done, t) message is received from the FC.

```
1. when a (D, t) is received from the child FC
2.         t_L = t; t_N = t_L + D.ta
3.         if next-message-type = * then
4.                 send (*, t) to the child FC
5.                 next-message-type = @
6.         else
7.                 lookahead = MIN( timestamp of the x msg recently sent to a remote LP, time of the NC Message Bag, t_N )
8.                 for each NC in the RemoteNCList do
9.                         sendNullMsg(lookahead);
10.                end for each
11.                if t_N != ∞ and minimumRemoteLookahead != ∞ and NCMessageBag != Empty then
12.                        suspend this LP
13.                else       this LP is DONE
14.                end if
15.                min-time = MIN( timestamp of the event pointed by event-pointer, timestamp of the x msg recently sent to a
16.                                remote LP,time of the NC Message Bag, minimum RemoteLookahead, t_N )
17.                resetLookahead∞oArray()
18.                if min-time = ∞ then
19.                        min-time = the timestamp of the received D message
20.                        send (D, t) to this NC with D.ta = zero
21.                else
22.
23.                        if min-time = the timestamp of the event pointed by event-pointer then
24.                                for each x in the Event List with min-time do
25.                                        send (x, t) to the child FC
26.                                        move event-pointer to the next event
27.                                end for each
28.                        end if
29.                        else if  an x msg was recently sent to a remote LP then
30.                                send (D, t) to this NC with D.ta = t_N – (min-time – timestamp of the received D msg )
31.                        end if
32.                        else if min-time = minimumRemoteLookahead and minimumRemoteLookahead =  t_N then
33.                                send (D, t) to this NC with D.ta = t_N – (min-time – timestamp of the received D msg )
34.                        end if
35.                        else if min-time = t_N then
36.                                send (@, t) to the child FC
37.                                next-message-type = *
38.                        end if
39.                        else if min-time = the time of the NC Message Bag then
40.                                for each x in the NC Message Bag with min-time do
41.                                        send (x, t) to the child FC
42.                                end for each
43.                        end if
44.                        else if min-time = minimumRemoteLookahead then
45.                                send (D, t) to this NC with D.ta = t_N – (min-time – timestamp of the received D msg )
46.                        end if
47.                end if
48.        end if
49. end when
```

Figure 3. Conservative NC algorithm for done message

The NC starts by computing its lookahead according to Formula (1) (line 7). Each NC maintains a list of remote NCs of LPs that will exchange messages with this LP. This list is initialized at the beginning of the simulation and is used by the NC to send null-messages (line 7 to 9). On every LP, the NC acts as the local controller of the simulation and carries on the event execution. The NC performs checks (line 11 to 14) to see if the LP must be suspended or finished (in this case, the LP will remain idle until the rest of LPs finish; the simulation terminates when all the LPs are idle). When the LP is suspended, it will wait for the lookahead of every LP in the *RemoteN-*

CList. When all the lookahead values are received, the NC resumes (line 14). The first *Done* message received by the NC is the response to the initialization message forwarded to the FC to start the simulation on that machine. Since *next-message-type* is initialized to @, the NC follows the second half part of the algorithm (line 6 to 48). The NC first calculates min-time using Formula (2) (line 15 to 16). The resulting min-time is the next local simulation time (i.e. the LVT of this LP) to which the NC should advance. After calculation of min-time, the *LookaheadInfoArray* of the NC is reset so that it will be filled with new lookahead information (line 17). A special situation might occur (line 18 to 20) when the min-time is ∞. This case arises when the LP is done with the simulation, and there is no event to be received from other LPs (because they have all sent an ∞ lookahead value). Therefore, the LP is done with the simulation. To finish the simulation at the LP, the NC sets the min-time back to the previous value, which was the timestamp of the received done message and sends a done message to itself with D.ta equal to zero, where D.ta is the next transition time that is reported to the NC. When this done message is received at the NC (line 7) the LP will be marked as *idle* and no further event execution will take place at that LP (line 9).

However, if the condition of line 18 is not met, the four cases mentioned earlier are checked (line 29 to 46). The tie-breaking mechanism is invoked as follows:

1. Case 1 (line 29 to 31) is given the highest priority.
2. If the minimum remote lookahead value is equal to the timestamp of the closest transition at this LP (i.e. t_N), priority is given to processing the *minimumRemoteLookahead* (line 32 to 34).

The *done* messages sent from the NC to itself (line 30, 33, and 45) are for synchronization purposes only. They are recognized by looking at the sender and receiver ID of the message (in this case, both are the NC itself). The D.ta value carried by these messages is calculated as the difference between the NC's current t_N and the LVT advancement:

$D.ta = t_N - (\textit{min-time} - \textit{timestamp of the received D msg})$

where *minTime* is the new LVT, and *timestamp of the received D msg* is the previous LVT.

When the NC receives this special done message it calculates its new lookahead, sends it across, and suspends itself.

To summarize the Conservative DEVS algorithm, the key features and assumptions of the simulation process are highlighted as follows:

1. All messages originating from Simulators must go through the parent FC. Hence, there is no direct communication between Simulators (even local ones), and FCs are always aware of the timing of state changes at their child Simulators.
2. Outgoing inter-LP communication happens only in the *collect* phases, whereas incoming inter-LP communication can occur in any phase. Since the output functions of imminent models are invoked only in the *collect* phases, at any given simulation time, all *external* messages going to remote NCs are sent out by the end of the *collect* phase. On the other hand, an *external* message from a remote source can arrive at the destination NC in any phase.
3. The NC is the starter for every *collect* and *transition* phase. The NC is invoked when it receives a *done* message from the FC (in response to a (I, t), (@, t), or (*, t) previously sent to the FC).
4. On each node, the NC advances the simulation time. The NC calculates the Local Virtual Time (LVT) of the LP at the beginning of every collect phase. The local FC and the Simulators do not send messages with a timestamp different from the current LVT.
5. *Dynamic Lookahead*: lookahead computation is performed after each LVT computation; hence, it is updated and distributed among all remote LPs every time before the LP is suspended. This strategy ensures that the lookahead value of an LP represents the latest LVT update, as there is at least one lookahead computation per LVT update. The dynamic lookahead mechanism states that lookahead value is not fixed and every lookahead computation could result in a different value than the previous stage. Unlike other conservative algorithms, the modeler is not required to specify the lookahead; instead, the algorithm dynamically extracts the lookahead information from the model itself.
6. *Low-cost Lookahead Computation*: as presented by Formula (1), our lookahead computation is a fast, efficient, and low-cost method, which involves a simple comparison of two existing parameters (t_N and $t_{NCMessageBag}$). There is neither an actual computation nor a significant computation time required to calculate the lookahead. Rather, the lookahead is extracted from already computed data that existed in the simulator before the conservative algorithm was integrated with it. Compared to other existing conservative mechanisms, this benefit reduces the overhead of our algorithm outstandingly especially that the frequency of invoking lookahead computation increases as the model size grows.
7. *Deadlock Avoidance*: since null-message distribution occurs before LP suspension, deadlock is strictly avoided. NC only suspends the LP after performing a lookahead computation and propagating it to all remote LPs via null-messages. Thus, when an LP is suspended, it has already forwarded its null-messages, and if every other LP gets suspended as well, they would all resume because all required null-

messages have been already distributed among them before suspension has taken place. This property of our algorithm was borrowed form the classical CMB mechanism.

4. IMPLEMENTATION DETAILS

CCD++, developed in C++, implements the original and Parallel DEVS and Cell-DEVS formalisms. It supports both standalone and parallel conservative simulations. CCD++ uses the WARPED kernel as a middleware to provide scheduling, memory, file, event, communication and time management. The major part of our conservative algorithm was implemented at the CCD++ level. Some modifications were done at the WARPED kernel to comply with the requirements of the conservative algorithm.

4.1. Scheduling

Each node maintains an input queue (*inputQ*, using linked lists provided by WARPED). Since causality violations are not allowed, *inputQ* is based on a FIFO mechanism. On every node, both DEVS messages and null-messages are treated as basic events and inserted into this queue.

```
1  when the scheduler is invoked to return the first unprocessed element
2          if currentPos != NULL then
3                  if currentPos is a null-message then
4                          currentPos = currentPos->next
5                          return NULL
6                  end if
7                  else if currentPos is a remote x message then
8                          currentPos = currentPos->next
9                          return NULL
10                 end if
11                 else if currentPos is a suspension message then
12                         for each unprocessed element of inputQ do
13                                 if event is a null-msg then
14                                         event->checked = true
15                                         recvdNullMsg++
16                                 end if
17                         end for
18                         if recvdNullMsg = totalLPs – 1 then
19                                 return currentPos
20                         else    return NULL
21                         end if
22                 end if
23                 else    return currentPos
24         end if
25 end when
```

Figure 4. Conservative scheduler algorithm

When the scheduler is invoked, it simply returns the head element of the *inputQ* and the event is deleted after execution. Our modifications to the scheduling mechanism are for the purpose of LP suspension. Recall the NC algorithm: when the NC decides that the LP should be suspended, it sends a special *done* message to itself. When the event returned by the scheduler happens to be this message, the event is not executed until all remote null-messages are received and inserted into the LP's *inputQ*. Figure 4 shows the conservative algorithm for the scheduler. The *currentPos* variable represents the first unprocessed element of *inputQ*. When a suspension event is detected (line 11), it is not returned until all required null-messages are received. The number of null-messages that must be received by an LP in order to resume is equal to the total number of LPs minus one (line 17), since the LP does not need a null-message from itself. In addition, the *recvdNullMsg* counter is only incremented once per null-message sender (if there is more than one null-message from a remote LPi, only the first one is counted).

4.2. Resuming a Blocked LP

When the event returned by the scheduler is the suspension event (the special done message sent form a NC to itself), two actions must be performed before it can be executed. First, all null-messages counted to increment *recvdNullMsg* must be executed. Second, all unprocessed remote external messages sent from other NCs to this LP, must be executed. Once they execute, the special *done* message is returned, and is serviced by the NC (Figure 5).

```
1. when executeProcess() is invoked
2.         event = Scheduler.getEvent()
3.         if event != NULL then
4.                 if event is suspension event then
5.                         for each null-msg with checked = true
6.                                 receive (null-msg)
7.                         end for
8.                         for each unprocessed remote x msg
9.                                 receive (x)
10.                        end for
11.                    receive (suspension)
12.                end if
13.        end if
14. end when
```

Figure 5. Suspension of event execution algorithm

4.3. Null-message Handling

When a null-message is received at the NC, a null-message handling mechanism is invoked. Every NC maintains a queue named *lookaheadInfoArray* to store the lookahead value carried by the null-message.

```
1  when a null-message is received
2          for i = 1 to arraySize do
3                  if lookaheadInfoArray[i] = NULL then
4                      lookaheadInfoArray[i] = recvdLookahead
5                      break
6                  end if
7          end for
8          latestRemoteLookahead = MIN (lookaheadInfoArray)
9  end when
```

Figure 6. NC null-message handling algorithm

The size of this queue is equal to the total number of LPs minus one (in a simulation with *n* LPs, every LP communicates at most directly with $n - 1$ LPs). When the null-message is received, NC saves the lookahead content into *lookaheadInfoArray* and calculates the *minimumRemoteLookahead* as the smallest element of this queue. The *minimumRemoteLookahead* is updated every time the NC receives a null-message. Hence, it is always the minimum lookahead value of all remote LPs. This mechanism is shown in Figure 7.

4.4. Simulation Termination

The NC on each LP decides the simulation termination. We discussed the criterion under which an LP terminates and becomes *idle*. The WARPED kernel checks the status of LPs every specified period. When all participating LPs are *idle*, the simulation terminates and the rest of memory clean-ups will be taken care by the kernel. Figure 8 shows the termination algorithm modified to support conservative simulation.

```
1 when the kernel checkIdle() is invoked
2   allIdle = true
3   for each participating LP do
4     if LP inputQ has unprocessed event and is not a null-message then
5                       allIdle = false;
6     end if
7   end for
8           return allIdle
9 end when
```

Figure 9. Simulation termination algorithm

An LP is not idle if it has an unprocessed event that is not a null-message in its inputQ. The restriction that the event should not be a null-message is for the case when the LP has finished its simulation and is idle waiting for other LPs to finish. While an LP is idle, it can still receive null-messages, however, they will not be executed. This null-message is the last one that the sender is distributing before it enters the idle state. The lookahead value carried by these null-messages is ∞ and it states that the originating LP has completed the simulation (there is nothing else to do on that machine).

5. RESULTS AND DISCUSSION

This section illustrates the performance of CCD++ by discussing the experiments we conducted on a cluster of 32 compute nodes (dual 3.2 GHz Intel Xeon processors, 1 GB PC2100 266 MHz DDR RAM) running Linux WS 2.4.21 interconnected through Gigabit Ethernet and communicating over MPICH 1.2.6. The Cell-DEVS model tested in our experiments is a model for forest fire propagation [22] based on Rothermel's model [19].

We are interested in evaluating the performance of the new CCD++ simulator in terms of the execution time and speedups. For all the Cell-DEVS models, a simple partition strategy evenly divides the cell space into horizontal rectangles. The fire propagation model was tested using 6400 cells (80 x 80 cell space). Figure 10 illustrates the results for this model.

As we can see our conservative DEVS algorithm reduces the execution time of this model when more nodes are engaged in the simulation. The execution time decreases from 77.37 to 17.24 s when the number of nodes climbs from 1 to 8 achieving a speedup of 4.48. However, when the number of nodes increases beyond 6, the difference among execution times is not significant (the execution time decreases by only 3.3% from 6 to 8 nodes). This is merely because when a model, especially a small one, is partitioned onto more nodes, the increasing overhead involved in inter-LP communication and the null-messages eventually degrade the performance. Hence, a tradeoff between the benefits of a higher degree of parallelism and the associated overhead costs needs to be reached when we consider different partition strategies.

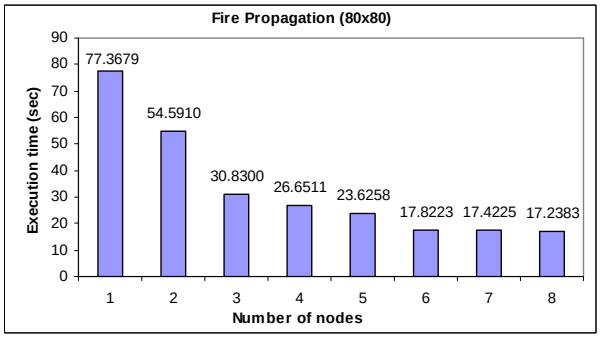

Figure 10. CCD++ performance - fire model (80x80)

6. CONCLUSION AND FUTURE WORK

We have presented a novel conservative algorithm for DEVS and Cell-DEVS models and have integrated this mechanism into the CD++ simulation toolkit. The resulting parallel simulator, called, CCD++ is based on CMB null-messages and lookahead concept and serves as the first purely conservative simulation benchmark for running large-scale DEVS and Cell-DEVS model in parallel and distributed fashion. We presented how our algorithm overcomes the limitations of the original DEVS conservative algorithm by implementing the mechanism at the top most level of the DEVS abstract simulator hierarchy (i.e. the coordinator), thus, significantly reducing lookahead computation frequency. Also, by replacing EIT and EOT calculations with a single lookahead computation the number of null-messages distributed among nodes are reduced notably.

Performance analysis has been conducted to evaluate the conservative DEVS algorithm in simulating DEVS-based models. We showed that CCD++ simulator mark-

edly improves execution times as the number of participating nodes increases. Considerable speedups were observed in our experiments, indicating the conservative simulator is well suited for simulating large and complex models. We are currently working on a thorough testing analysis by running intensive tests with larger and more complex models on both CCD++ and the purely optimistic simulator (PCD++) to provide a reference guide on whether to use a conservative simulator or an optimistic one and under which circumstances one outperforms the other.

7. REFERENCES

[1] Zeigler, B., T. Kim, and H. Praehofer. 2000. *Theory of modeling and simulation*. San Diego: Academic Press.

[2] Chow, A. C. and B. Zeigler. 1994. "Parallel DEVS: A parallel, hierarchical, modular modeling formalism". In *Proceedings of the Winter Computer Simulation Conference*, Orlando, FL.

[3] Wainer, G. *Discrete-Event Modeling and Simulation: a Practitioner's approach*. CRC Press. Taylor and Francis. 2009.

[4] Wolfram, S. 1986. *Theory and applications of cellular automata*. Advances series on complex systems, 1. World Scientific: Singapore.

[5] Fujimoto, R. M. *Parallel and distributed simulation systems*. New York: Wiley. 2000.

[6] D. R. Jefferson. 1985. "Virtual time". *ACM Trans. Program. Lang. Syst. 7*(3), pp. 404-425.

[7] Bryant, R. E. "Simulation of packet communication architecture computer systems". Massachusetts Institute of Technology. Cambridge, MA. USA. 1977.

[8] Chandy, K. M.; Misra J. "Distributed simulation: A case study in design and verification of distributed programs". *IEEE Transactions on Software Engineering*. pp.440-452. 1978.

[9] Wainer, G. 2002. CD++: A toolkit to develop DEVS models. *Software – Practice and Experience*, 32:1261-1306.

[10] Q. Liu, G. Wainer, "Parallel environment for DEVS and Cell-DEVS models". *SIMULATION* 83(6), 2007, pp.449-471.

[11] Zeigler, B.; Moon, Y.; Kim, D.; Kim, J. G. "DEVS-C++: A high performance modeling and simulation environment". The 29th Hawaii International Conference on System Sciences. 1996.

[12] Zeigler, B.; Kim, D.; Buckley, S. "Distributed supply chain simulation in a DEVS/CORBA execution environment". Proceedings of the 1999 Winter Simulation Conference. 1999.

[13] Kim, K.; Kang, W. "CORBA-based, Multi-threaded Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-hierarchical One". International Conference on Computational Science and Its Applications (ICCSA).Assisi, Italy. 2004.

[14] Cheon, S.; Seo, C.; Park, S.; Zeigler, B. "Design and implementation of distributed DEVS simulation in a peer to peer network system". Advanced Simulation Technologies Conference. Arlington, VA, USA. 2004.

[15] Zhang, M.; Zeigler, B.; Hammonds, P. "DEVS/RMI – An auto-adaptive and reconfigurable distributed simulation environment for engineering studies". DEVS Integrative M&S Symposium. Huntsville, AL, USA. 2006.

[16] T.G. Kim, S.B. Park, "The DEVS formalism: Hierarchical modular systems specification in C++". In *Proceedings of European Simulation Multiconference*. 1992.

[17] Y.R. Seong, S.H. Jung, T.G. Kim, K.H. Park, "Parallel simulation of hierarchical modular DEVS models: A modified Time Warp approach". *Internat. J. Comput. Simulation 5 (3)*, 1995, pp.263-285.

[18] Praehofer, H., Reisinger, G.: "Distributed Simulation of DEVS-based Multiformalism Models". *AIS '94, Gainesville, FL, IEEE/CS Press*, Dec. 1994, pp. 150-156.

[19] E. DeBenedictus, S Ghosh, M.-L- Yu. "A Novel Algorithm for Discrete Event Simulation". *IEEE Computer*, June 1991, pp. 21-33.

[20] Radhakrishnan, R., D. E. Martin, M. Chetlur, D. M. Rao, and P. A. Wilsey. 1998. "An object-oriented time warp simulation kernel". In *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments*, LNCS 1505, pp. 13-23.

[21] Glinsky, E. and G. Wainer. 2006. "New parallel simulation techniques of DEVS and Cell-DEVS in CD++". *Proceedings of 39th Annual Simulation Symposium*, 244–251.

[22] Ameghino, J., A. Troccoli, and G. Wainer. "Models of Complex Physical Systems Using Cell-DEVS". *The 34th IEEE/SCS Annual Simulation Symposium*. 2001.

[23] 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. 1972.