Parallel Environment for DEVS and Cell-DEVS Models

Qi Liu and Gabriel Wainer

SIMULATION 2007; 83; 449
DOI: 10.1177/0037549707085084

Parallel Environment for DEVS and Cell-DEVS Models

Qi Liu
Gabriel Wainer
Carleton University
Centre for Advanced Visualization and Simulation (V-Sim)
Department of Systems and Computer Engineering
3216 V-Sim Building
1125 Colonel By Drive
Ottawa, ON K1S 5B6, Canada
{liuqi, gwainer}@sce.carleton.ca

Discrete Event System Specification (DEVS) is a sound formalism to describe generic dynamic systems in a hierarchical and modular way. Cell-DEVS is a DEVS-based formalism intended to model complex physical systems as cell spaces. This work presents new techniques for executing DEVS and Cell-DEVS models in parallel and distributed environments based on the WARPED kernel, an implementation of the Time Warp protocol. The optimistic simulator PCD++, built as a new simulation engine for CD++, is a toolkit that implements the DEVS and Cell-DEVS formalisms. We redesign algorithms in CD++ to carry out optimistic simulations using a non-hierarchical approach that reduces the communication overhead. The message-passing organization is analyzed using a high-level abstraction referred to as wall clock time slice. We propose a two-level user-controlled state-saving mechanism to achieve efficient and flexible state saving at runtime. Various optimization strategies are applied to PCD++ and their effects are analyzed quantitatively, including a risk-free message type-based state-saving strategy to reduce the number of states saved during the simulation significantly, and a one log file per node strategy to break the bottleneck caused by file I/O operations. It is shown that PCD++ markedly outperforms other alternatives and considerable speedups can be achieved in parallel and distributed simulations.

Keywords: Parallel DEVS models, Cell-DEVS models, discrete event simulation, optimistic synchronization techniques

1. Introduction

Computer-based modeling and simulation (M&S) has become an important tool for analyzing and designing a broad array of complex systems where a mathematical analysis is intractable. As a sound formal M&S framework based on generic dynamic system concepts, the DEVS [1] formalism supports hierarchical and modular construction of models, allowing model reuse, reducing development and testing time. Since its first formalization, DEVS has been extended into various directions. The Parallel DEVS or P-DEVS [2] formalism is an extension that eliminates the serialization constraints which existed in the original DEVS definition, allowing increased parallelism to be exploited in parallel and distributed simulations. The Cell-DEVS [3] formalism combines Cellular Automata [4] with DEVS theory to describe n-dimensional cell spaces as discrete event models, where each cell is represented as a DEVS basic model that can be delayed using explicit timing constructions.

Parallel discrete event simulation (PDES) has received increasing interest as simulations become more time consuming and geographically distributed. Synchronization techniques for PDES systems generally fall into two categories: conservative approaches that strictly avoid violating the local causality constraint [5] and optimistic approaches that allow violations to occur, but provide mechanisms to recover from them through a process known as *rollback*. Usually, optimistic approaches can exploit a higher degree of parallelism available in the simulation,

SIMULATION, Vol. 83, Issue 6, June 2007 449–471

DOI: 10.1177/0037549707085084
Figures 9, 16–26 appear in color online: http://sim.sagepub.com

whereas conservative approaches tend to be overly pessimistic and force sequential execution when it is not necessary. Moreover, conservative approaches generally rely on application-specific information to determine which events are safe to process. While optimistic algorithms can execute more efficiently if they exploit such information, they are less reliant on the application for correct execution, allowing greater transparent synchronization and simplifying software development. On the other hand, the overhead of state saving and rollback operations incurred in optimistic simulations constitutes the primary bottleneck that may result in degradation of system performance. Jefferson's Time Warp mechanism [6] is by far the most well known optimistic synchronization protocol that uses *virtual time* to model the passage of time in the simulation. The simulation is executed via several Time Warp processes interacting with each other by exchanging time-stamped event messages. The WARPED simulation kernel [7] is a configurable middleware that implements the Time Warp mechanism and various optimizations.

CD++ [8] is an M&S toolkit that implements P-DEVS and Cell-DEVS formalisms. It currently supports both standalone and parallel conservative simulations [9]. In this work, we present new techniques for optimistic simulations in CD++ based on the Time Warp mechanism. Our optimistic simulator PCD++ is built as a new extension to the CD++ toolkit. PCD++ employs a layered architecture as introduced by Troccoli and Wainer [9]. It also adopts the flat simulation approach that eliminates the need for intermediate coordinators [10]. The algorithms for the DEVS processors are redesigned to allow optimistic simulations in parallel and distributed environments. The message-passing organization is analyzed using a high-level abstraction called *wall clock time slice* (WCTS). The algorithms for the Cell-DEVS atomic models are adapted to the asynchronous state transition paradigm. Various enhancements and optimizations are proposed and integrated into the PCD++ simulator. We show that PCD++ markedly outperforms other alternatives and considerable speedups are achievable in simulations, indicating that PCD++ is well suited for simulating large and complex models.

2. Parallel DEVS Background

The DEVS [1] formalism provides a framework for the definition of hierarchical models in a modular way. A real system modeled using DEVS can be described as a composition of behavioral (atomic) and structural (coupled) components. The P-DEVS [2] formalism eliminates the restrictions that forced the original DEVS definition to sequential execution. It is used as the theoretical foundation for our research. A P-DEVS atomic model is defined as:

$$M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle .$$

At any time, an atomic model is in some state $s \in S$. If no external event occurs, it will remain in state s for ta(s). When ta(s) expires, the atomic model outputs the value $\lambda(s)$ and changes to a new state given by $\delta_{int}(s)$. P-DEVS models employ a bag of inputs (X^b) to allow the execution of multiple concurrent events. If one or more external events occur before ta(s), the atomic model changes to a new state defined by $\delta_{ext}(s, e, X^b)$, which combines the functionality of a number of external transitions into a single one. A δ_{con} function is defined to decide the new state in cases of collision between external and internal functions.

P-DEVS coupled models are defined as a set of basic models (atomic or coupled) interconnected through the interfaces of the models. A P-DEVS coupled model is defined as:

$$DN = \langle X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC \rangle .$$

The specifications for the set of input and output events (X and Y) and couplings (EIC, EOC and IC) follow the definitions of DEVS coupled models [1]. The basic components (D and M_d) are specified by the P-DEVS atomic model definition.

The Cell-DEVS [3] formalism allows the specification of discrete event cell spaces, improving their definition by using explicit timing delays. A Parallel Cell-DEVS atomic model [11] can be formally defined as:

$$TDC = \left\langle X^b, Y^b, S, N, d, \delta_{int}, \delta_{ext}, \delta_{con}, \tau, \tau_{con}, \lambda, D \right\rangle .$$

A cell can interchange data with other neighboring cells and models outside the cell space via its interface (X^b, Y^b). The input values are used to compute the future state of the cell by evaluating the local function (τ, τ_{con}). New state values are transmitted only after the completion of the delay time given by the delay function (d). The P-DEVS transition (δ_{int}, δ_{ext}, δ_{con}) and output (λ) functions are included in each cell. A cell space consisting of multiple cells interconnected by the neighborhood relationship is defined by a coupled Cell-DEVS model:

$$GCC = \langle X_{list}, Y_{list}, X, Y, n, \{t_1, \ldots, t_n\}, N, C, B, Z \rangle .$$

The cell space (C) is a coupled model defined as an array of Cell-DEVS atomic models of fixed size ($t_1 \times \ldots \times t_n$). The neighborhood set (N) gives the relative position between the origin cell and the surrounding neighbors. The border of the cell space is specified by the border cells (B). The Z function allows definition of the coupling between cells in the model. Figure 1 illustrates the informal definition of the DEVS and Cell-DEVS formalisms.

Various DEVS-based M&S toolkits have been implemented by different researchers. The following is a brief survey on some of the existing toolkits intended for distributed simulation.

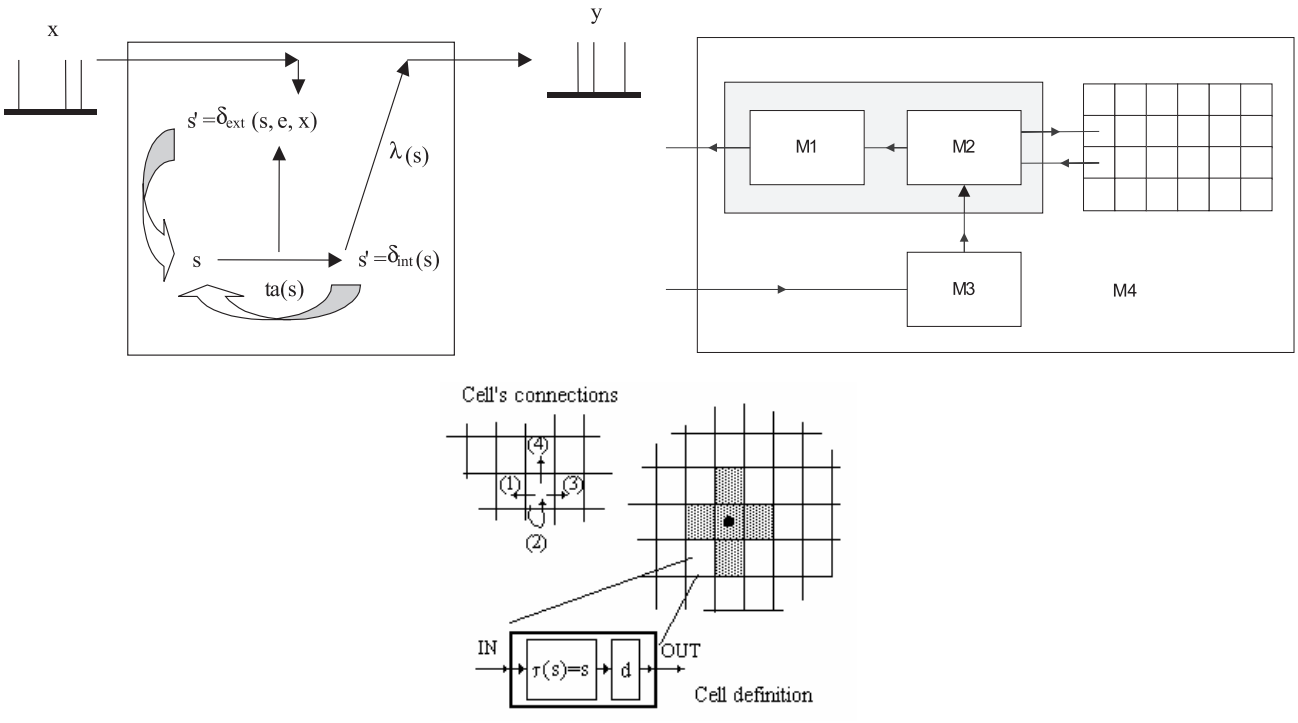

Figure 1. Informal definition of DEVS and Cell-DEVS [1, 11]

- DEVS/CORBA [12] is a runtime infrastructure on top of CORBA middleware to support distributed simulation of DEVS components. It is possible to embed DEVS/CORBA in a larger network-centric environment to provide a combination of graphical process modeling, discrete-event simulation, animation, activity-based costing and optimization functions.

- DEVS/HLA [13] is an HLA-compliant M&S environment implemented in C++ that supports high-level model construction. It greatly simplifies the underlying programming details required to establish and participate in an HLA federation.

- DEVSCluster [14] is a CORBA-based, multithreaded distributed simulator implemented in Visual C++. It transforms a hierarchical DEVS model into a non-hierarchical model to ease the synchronization of the distributed simulation.

- DEVS/Grid [15] is an M&S framework implemented using Java and Globus toolkit for Grid computing infrastructure. It constructs a fully automated simulation environment based on a set of facilities, including cost-based hierarchical model partitioning, dynamic coupling restructuring, automatic model deployment and M&S name and directory service.

- DEVS/P2P [16] is an M&S framework based on P-DEVS formalism and Peer-to-Peer message communication protocol. It uses a customized DEVS simulation protocol to achieve decentralized inter-node communication. Simulators are synchronized by themselves without involving a coordinator.

- DEVS/RMI [17] is a DEVS-based system that provides a fully dynamic and re-configurable runtime infrastructure for handling load balancing and fault tolerance in distributed simulations. It reduces the overhead associated with common middleware solutions by using the native support of Java RMI to achieve the synchronization of local and remote simulators.

However, none of them supports optimistic simulation of Cell-DEVS models in parallel and distributed environments. Nutaro presents a risk-free optimistic simulation algorithm [18] to simulate the class of systems represented in the DEVS formalism correctly. In this approach, only correct outputs with the minimum global time are sent to avoid the spread of causality errors to remote processes. This mechanism is well suited for shared memory architectures, but has limitations in distributed heterogeneous environments.

As mentioned earlier, the CD++ toolkit is extended in our research to allow optimistic simulation of complex and large-scale DEVS and Cell-DEVS models. The op-

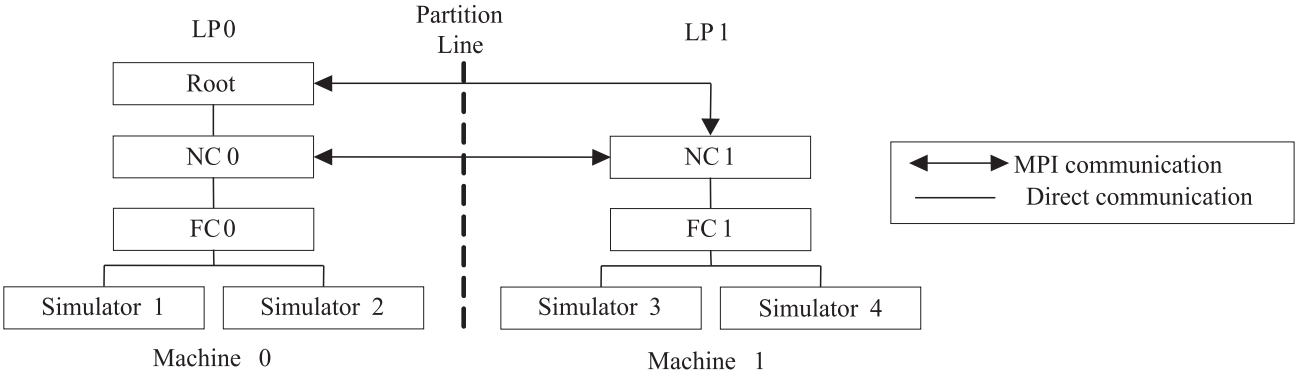

Figure 2. An example distributed processor structure involving two machines

timistic simulator PCD++ is built on top of the WARPED kernel, which provides services for defining different types of Time Warp processes (known as simulation objects) based on Jefferson's definition. Simulation objects mapped on a physical processor are grouped by an entity referred to as *logical process* (LP). The WARPED kernel relies on the Message Passing Interface (MPI) for high-performance communications on both massively parallel machines and on workstation clusters. The PCD++ simulator provides two loosely-coupled frameworks: a modeling framework and a simulation framework. The former consists of a hierarchy of classes rooted at *Model* to define the behavior of the DEVS and Cell-DEVS models; the latter defines a hierarchy of classes rooted at *Processor*, which, in turn, derives from the abstract simulation object definition in the kernel to implement the simulation mechanisms. That is, the PCD++ processors are concrete implementations of simulation objects to realize the abstract DEVS simulators.

3. Optimistic Simulation in PCD++

In the original definition of the abstract simulator [2], the DEVS processors are specialized into two different simulation engines, namely *simulators* and *coordinators*. The structure of these hierarchical DEVS processors mimics the DEVS model hierarchy. Basically, the role of a *simulator* is to invoke the state transition (internal and external) functions in an atomic model, while a *coordinator* is attached to a coupled model to translate the output events between its children and to keep track of the time of the next imminent dependents. When the CD++ toolkit is run in standalone and parallel conservative modes, a single root coordinator is associated with the top-level coupled model to manage the entire simulation. In addition, a one-to-one correspondence is established between the model components and the DEVS processors, increasing the communication overhead of message passing.

Based on previous research [10], PCD++ employs a flattened structure consisting of four types of DEVS processors: Simulator, Flat Coordinator (FC), Node Coordinator (NC), and Root. Introducing the FC and NC eliminates the need for intermediary coordinators in the DEVS processor hierarchy. Further, the Root coordinator is no longer the global scheduler in the simulation. Instead, the simulation is managed by a set of NCs running on different machines in a decentralized manner. The simulation is executed in a message-driven fashion. PCD++ processors exchange messages that can be classified into two categories: content messages and synchronization messages. The former includes the external message (x, t) and output message (y, t), while the latter includes the initialization message (I, t), collect message (@, t), internal message (*, t), and done message (D, t). Figure 2 shows an example of the distributed processor structure involving two machines.

A LP is created on each machine, grouping together the PCD++ processors mapped on that machine. A Root is created only on LP0. It starts the simulation and performs I/O operations between the simulation system and the surrounding environment. A NC and a FC are created on each LP. The FC is in charge of intra-LP communications between its child Simulators underneath. The NC is the local central controller on its hosting LP and the end point of inter-LP communications. A Simulator is responsible for executing the DEVS abstract functions defined in its associated atomic model.

The original message-processing algorithms for the PCD++ processors can be found in Glinksy and Wainer [10]. Some of the algorithms have been redesigned in our research to allow a more appropriate division of functionalities among the processors and to address a variety of issues in distributed optimistic simulations. We now present the redesigned algorithms, including the FC algorithm for (y, t), and the NC algorithms for (y, t), (x, t), and (D, t).

The FC synchronizes its child Simulators, routes messages among them, and forwards to the NC messages sent

```
1. when a (y, t) is received from a child Simulator C_i
2.     if y ultimately influences remote Simulators or the environment then
3.           send a single (y, t) to the parent NC
4.     end if
5.     for each child C_j influenced by y do
6.           x = Z_{i,j}(y)
7.           cache j in the synchronize set
8.           send (x, t) to C_j
9.     end for each
10. end when
```

Figure 3. FC algorithms for (y, t)

from its children to the environment or to other remote Simulators. The new FC algorithm for (y, t) is shown in Figure 3, where Simulators ready for a state transition are cached in *synchronize set*.

Upon the arrival of a (y, t), the FC searches the model coupling information to find its ultimate destinations. A destination is ultimate if it is an input port on an atomic model or an output port on the topmost coupled model. If the (y, t) is sent eventually to remote Simulators or to the environment, the FC simply forwards the (y, t) itself to the parent NC. Otherwise, the FC translates the (y, t) into a (x, t) using the $Z_{i,j}$ translation function and directly sends the (x, t) to the local receivers, which are recorded in the *synchronize set* for later state transitions.

The major portion of our redesign effort is reflected in the message-processing algorithms for the NC. As the local central controller, the NC performs a number of important operations.

1. Inter-LP communications: a structure called *NC Message Bag* is introduced to contain the received external messages from other remote NCs. The time of the NC Message Bag is defined as the minimum timestamp among the messages contained in it, while an empty bag has a time of infinity.

2. Handling external events from the environment: the NC uses a structure called *Event List* to hold the external events. The current position in the Event List is held by the *event-pointer*, which is defined in the NC's state.

3. Driving the simulation on the hosting LP: the NC advances the local simulation time to the minimum among the timestamp of the external event pointed by the *event-pointer*, the time of the NC Message Bag, and the closest state transition time given in the (D, t) received from the FC.

4. Managing the flow of control messages in line with the P-DEVS formalism: the NC uses *next-message-type* to keep track of the type of the control message (either @ or *) that will be sent in the next simulation cycle.

The redesigned NC algorithm for (y, t) is shown in Figure 4. If the (y, t) is sent to the environment, the NC directly forwards it to the Root. In addition, the NC determines the remote machines on which the ultimate receiving Simulators locate, based on the model coupling and partition information. The NC then translates the (y, t) into a (x, t) and sends it to the NC on each of those machines. On the receiving end, the (x, t) will eventually be delivered to the receiving Simulators on that machine.

A simplified version of the NC algorithm for (D, t) is shown in Figure 5. If the *next-message-type* has a value of @, the NC calculates the next simulation time, *min-time*, based on the three factors as discussed earlier (line 7 to 9). If the *min-time* is larger than the user-specified stop time, the NC simply sets a flag, *dormant*, and exits the algorithm. The usage of the *dormant* flag will be discussed shortly. Otherwise, the NC sends the external events (line 15) and external messages (line 21) scheduled at the *min-time*, if any, to the FC. It then sends a control message to the FC and sets the *next-message-type* accordingly (line 25 to 31). The *next-message-type* is set to * only after the NC sends out a (@, t) (line 27), in which case imminent Simulators exist on the LP and their output functions will be invoked when the (@, t) arrives. The imminent Simulators need to perform internal transitions immediately after the output operations. Therefore, the NC triggers the internal transitions by sending out a (*, t) in the next simulation cycle (line 4). On the other hand, if there is no imminent Simulator at this time, the NC sends a (*, t) whenever external messages are flushed to the FC (line 29) to trigger the external transitions in the non-imminent Simulators.

In optimistic simulations, some LPs may have processed all their local events while waiting for other LPs to complete the whole simulation. Meanwhile, the lagging-behind LPs may send messages to the waiting LPs and thereby reactivate them. To allow proper reactivation of the simulation on a LP, we define a special state called *dormant* for the NC. The NC enters into the dormant state once the computed *min-time* is greater than the stop time

```
1. when a (y, t) is received from the child FC
2.     if y ultimately sends to the environment then
3.            send (y, t) to the Root
4.     end if
5.     for each remote machine i hosting destination Simulators of y do
6.            x = Z_{i,j}(y)
7.            send (x, t) to NC_i
8.     end for each
9. end when
```

Figure 4. NC algorithms for (y, t)

```
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.            min-time = MIN( timestamp of the event pointed by event-pointer,
8.                    time of the NC Message Bag,
9.                    t_N )
10.           if min-time > stop-time then
11.                  dormant = true
12.           else
13.                  if min-time = the timestamp of the event pointed by event-
                        pointer
                        then
14.                         for each x in the Event List with min-time do
15.                                send (x, t) to the child FC
16.                                move event-pointer to the next event
17.                         end for each
18.                  end if
19.                  if min-time = the time of the NC Message Bag then
20.                         for each x in the NC Message Bag with min-time do
21.                                send (x, t) to the child FC
22.                         end for each
23.                  end if
24.                  remove all x in the NC Message Bag with min-time
25.                  if t_N = min-time then
26.                         send (@, t) to the child FC
27.                         next-message-type = *
28.                  else
29.                         send (*, t) to the child FC
30.                         next-message-type = @
31.                  end if
32.           end if
33.    end if
34. end when
```

Figure 5. Simplified NC algorithm for (D, t)

```
1. when a (x, t) is received from a remote NC
2.     insert message x to the NC Message Bag
3.     bag-time = the time of the NC Message Bag
4.     if (dormant = true) & (bag-time <= stop-time) &
5.      (all events in the input queue with timestamp = bag-time have been
         processed)
         then
6.           dormant = false
7.           t_L = bag-time; ta = 0
8.           for each x in the NC Message Bag with bag-time do
9.                 send (x, t) to the child FC
10.          end for each
11.          remove all x in the NC Message Bag with bag-time
12.          send (*, t) to the child FC
13.          next-message-type = @
14.    end if
15. end when
```

Figure 6. NC algorithms for (x, t)

(line 11 in Figure 5), indicating that all the local events on the LP have been processed. The NC exits the dormant state and reactivates the simulation on its LP upon the arrival of external messages from other remote NCs. In this case, the NC spontaneously flushes (i.e. without the presence of a (D, t) from the child FC) the received external messages with the minimum timestamp in its *NC Message Bag* to the FC. It also sends a (*, t) to the FC to trigger the appropriate state transitions at the receiving Simulators. The new NC algorithm for (x, t) is shown in Figure 6.

4. Message-Passing Organization

Based on the new message-processing algorithms presented in the previous section, we now show an example message-passing scenario using an *event precedence graph*, where a vertex (black dot) represents a message and an edge (black arrow) represents the action of sending a message with the message type placed nearby. A line with a solid arrowhead denotes a (synchronous) intra-LP message and a line with a stick arrowhead denotes an (asynchronous) inter-LP message. A lifeline (dashed line) is drawn for each PCD++ processor. The execution sequence of messages is marked by the numbers following the message type.

Figure 7 illustrates the flow of messages on a LP with four PCD++ processors: a NC, a FC and two Simulators (S1 and S2). We do not consider the potential out-of-order execution of messages since the rollback operations are performed automatically and transparently in the kernel.

From the diagram, we can see that the execution of messages at any simulation time on a LP can be decomposed into at most three distinct phases: initialization phase (**I**); collect phase (**C**); and transition phase (**T**), as demarcated by done messages (bold black arrows) received by the NC. Only one initialization phase exists at the beginning of the simulation (time 0), including messages in the range of $[I_1, D_7]$. The collect phase at simulation time t starts with a (@, t) sent from the NC to the FC and ends with the following (D, t) received by the NC. For example, the collect phase at time 0 comprises messages in the range $[@_8, D_{14}]$. This phase is optional; it happens if, and only if, there are imminent Simulators on the LP at that time. Finally, the transition phase at simulation time t begins with the first (*, t) sent from the NC to the FC and ends at the last (D, t) received by the NC at time t. In the diagram, messages in the range of $[*_{15}, D_{32}]$ belong to the transition phase at time 0. The transition phase is mandatory for each individual simulation time.

Furthermore, a transition phase may contain multiple rounds of computations; each starts with zero/one/more (x, t) followed by a (*, t) sent from the NC to the FC and ends with a (D, t) returned to the NC. In the example, the transition phase at time 0 has three rounds: R_0 with messages in range $[*_{15}, D_{19}]$, R_1 with messages in $[x_{20}, D_{26}]$, and R_2 with messages in $[x_{27}, D_{32}]$. During each round, state transitions are performed incrementally with additional external messages and/or for potentially extra Simulators. We will denote a transition phase of (n + 1) rounds as $[R_0 \ldots R_n]$.

Based on the above analysis, we now present a new abstraction that allows a higher-level understanding of the simulation process on each LP. From a computational standpoint, the sequential simulation on a LP can be viewed as a sequence of computation units, one for each group of simultaneous events, transforming the system mapped on that node according to the P-DEVS formalism. Each computation unit is performed during a span of time as measured by a physical wall clock. Such compu-

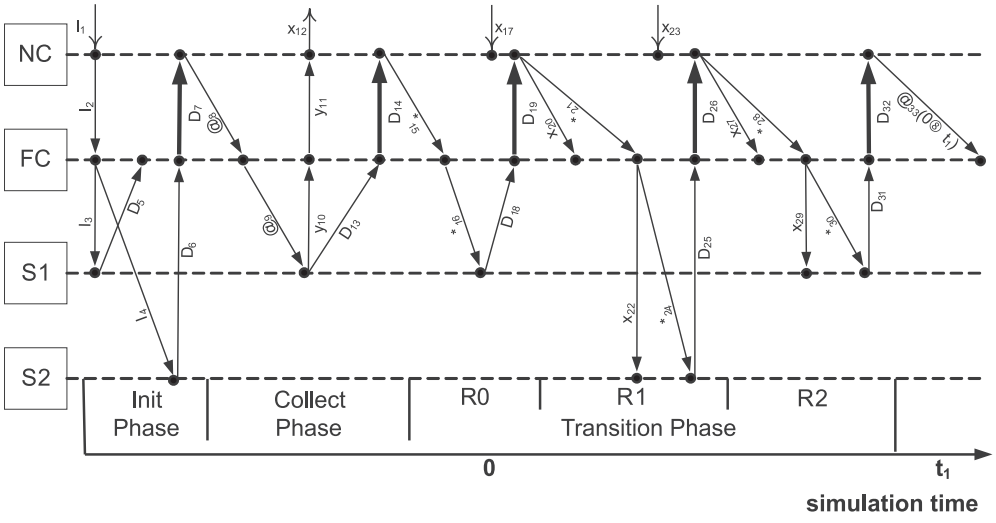

Figure 7. An example message-passing scenario on a LP

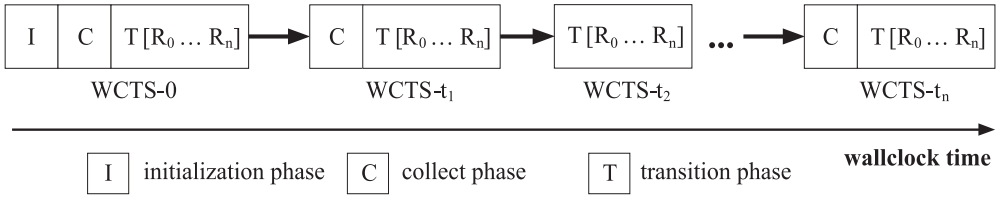

Figure 8. WCTS representation for the simulation on a LP

tation unit is referred to as wall clock time slice (WCTS). A WCTS comprising simultaneous events occurred at virtual time t is denoted as WCTS-t, and t is called the virtual time of the WCTS.

Figure 8 illustrates the sequential simulation on a LP in terms of WCTS. The simulation is viewed as a sequence of wall clock time slices linked together along the time axis. Each stands for the execution of simultaneous events at a specific simulation time on all the PCD++ processors associated with the LP according to the P-DEVS formalism. Each WCTS-t may contain one mandatory transition phase and one optional collect phase.

Several properties of the WCTS are summarized as follows.

1. The simulation on a LP starts with WCTS-0, the only WCTS with all three phases.
2. Wall clock time slices are linked together by messages sent from the NC to the FC (black arrows in Figure 8). When the NC determines the next simulation time at the end of a WCTS, it sends out messages that will be executed by the FC at the new simulation time, thus initiating the next WCTS on the LP.
3. The completion of the simulation on a LP is marked by a WCTS sending out no linking messages, e.g. WCTS-t_n in the diagram. The whole simulation finishes only when all participating LPs have completed their corresponding parts of the simulation.
4. Wall clock time slices are *atomic* computation units during rollback operations. A typical rollback scenario is shown in Figure 9.

In the diagram, the simulation on LP_i is executing in WCTS-t_n when a straggler or anti-message with timestamp t_2 arrives at the NC (action 1). Based on the kernel rollback mechanisms, the received straggler or anti-message is inserted into WCTS-t_2 (a message implosion happens in WCTS-t_2 if it is an anti-message) (action 2). The rollbacks are then propagated among the PCD++ processors, restoring their states to those saved at the end of WCTS-t_1 (action 3), and all messages in WCTS-t_2 up to WCTS-t_n are undone. After the rollbacks, the simulation

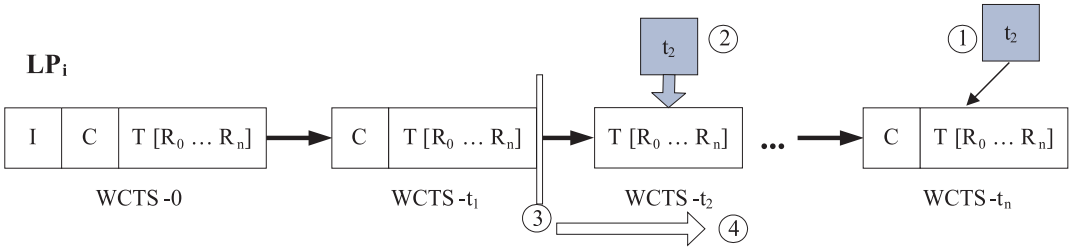

Figure 9. Typical rollback scenario shown in terms of wall clock time slices

on LP$_i$ resumes forward execution from the unprocessed linking messages between WCTS-t$_1$ and WCTS-t$_2$ (action 4). Simply put, the arrival of a straggler or anti-message modifies the WCTS to which it belongs, and the simulation resumes forward execution from the modified WCTS after the rollbacks, taking the straggler or anti-message into account.

5. Algorithms for Cell-DEVS Models

As the state transitions are performed incrementally at the Simulators in the transition phases, the algorithms for Cell-DEVS atomic models need to be adapted to this *asynchronous state transition paradigm* to obtain the same simulation results in PCD++ as in the standalone version for any given Cell-DEVS model. A brief description of the new computation model under the asynchronous state transition paradigm is as follows.

1. *Applying preemptive semantics to the state transition logic.* For a transition phase [R$_0$. . . R$_n$], the state transitions in all but the last round (R$_n$) are based on incomplete information and hence false transitions. R$_n$ has the best chance to perform the correct transition. (This is the case if no rollback happens later. Otherwise, the whole transition phase will be reprocessed after the rollbacks.) Since the state transition in a later round involves additional external messages, it has a better chance to perform the correct computation and generate the correct results. The state transition logic should therefore be implemented so that the computation of the later round preempts that of the previous round. In the end, the potentially correct results obtained in R$_n$ preempt those erroneously generated in R$_{n-1}$, and the simulation advances to the next virtual time. Both the value and state of the cell must follow this preemptive logic during the multi-round state transitions. To do so, the cell needs to record its *previous value* and *previous state* passed in from the previous virtual time at the beginning of R$_0$ for each individual simulation time. For time 0, the previous value and state are the cell's initial value and state defined by the modeler. With the exception of R$_0$ at time 0, the entry point of R$_0$ is identified by a change in the simulation time. Hence, a cell can record its previous value and state once a time change is detected at the beginning of the state transition algorithm. For time 0, this job can be done in the initialization phase.

2. *Handling user-defined state variables.* User-defined state variables may be involved in the evaluation of local rules. With the multi-round transition phase, this computation becomes much more complex. During each round, a potentially different rule is evaluated and the state variables referenced in the rule are computed. As a result, potentially wrong values are assigned to the variables and passed to the next round. The computation errors accumulate throughout the rounds and the wrong values are passed to the simulation at the next virtual time. To ensure correct computation of the state variables, a cell needs to record the values of the user-defined state variables at the beginning of each R$_0$. These recorded values are inherited from the potentially correct computation of R$_n$ at the previous simulation time. During the following rounds at a specific simulation time, the state variables are first restored to the recorded values. Only after this restoration operation can a new computation be performed. Therefore, the cell always uses the potentially correct values as the basis for a new computation.

3. *Handling external events.* In CD++, port-in transition function (for evaluating external events) is given a higher priority than the local transition rules. Under the new asynchronous state transition paradigm, the computation results of the port-in transition function may be modified by the local transition rules in later rounds. In order to preserve the effect of external events throughout the multi-round transition phase, we defined an *event-flag* in each cell. Whenever the cell's value is influenced by an external event or events at a given simulation time, this flag is set so that no further changes can

be done to the cell's value in the following rounds at this time. This flag will be reset once the preserved value has been output to other cells. In this case, the influence of the external event has spread out in the cell space as expected, and the cell's value is again under the control of its local transition rules.

Figure 10 shows the new algorithms for the initialization and external transition (δ_{ext}) functions in Cell-DEVS models with transport delay, while the output (λ) and internal transition (δ_{int}) functions are implemented as in Troccoli and Wainer [9].

When the initialization function is invoked, the *event-flag* is initialized to false. A *transient-value* is used to record the tentative value changes throughout the multi-round state transitions at a given simulation time. It is initialized to the cell's initial value. The *time-record* used to detect the entry point of R_0 is initialized to zero. The cell also records the initial value of the user-defined state variable in *state-variable-record*. Then, the cell inserts an element *<0/out* = v> into the queue so that its initial value v can be sent to all its neighbors via its output port *out* in the collect phase at time 0. Finally, function *holdIn* is invoked, preparing the cell to output its initial value once the collect phase begins.

The major modifications are made to the δ_{ext} function, which is invoked repeatedly throughout the multi-round transition phases. The cell detects the entry point of R_0 by comparing the *time-record* with the current simulation time, *current-time*. Once found, the cell's value passed in from the previous time, *previous-value*, is retrieved and recorded in *transient-value* for use in the later rounds. The *event-flag* is then reset to false in case external events have been processed during the computations of the previous time, and the current value of the user-defined state variable is recorded in *state-variable-record*. These housekeeping operations are performed only at the beginning of R_0 for each individual simulation time. The remaining logic is common for all the rounds in a transition phase. The state variable is restored to the recorded value before rule evaluations (line 21). The *event-flag* is set (line 30) if the new value is derived from external events, preventing further modification to the cell's value in the following rounds.

The preemptive transition logic is realized in line 32 to 52. There are three possible cases that can happen in each round: a new value change occurs (line 33 to 35), the value is changed back to *previous-value* (line 37 to 45), or the value is changed further from the result of the previous round (line 47 to 51). For all these cases, the *transient-value* always follows the newly generated *new-value*. Once a new value change is detected, the *new-value* is inserted into the queue and an output is scheduled. If the cell's value is changed back to *previous-value*, i.e. there is actually no value change if we consider the computation up to the current round as a whole, the cell preempts the result of the previous round by removing the previously inserted element from the queue, and reschedules output based on the present queue. On the other hand, if the cell's value is changed further, the cell preempts the previous result by replacing the element with a new one and reschedules output accordingly.

Figure 11 shows the new algorithms for the initialization and external transition (δ_{ext}) functions in Cell-DEVS models with inertial delay. Again, the output (λ) and internal transition (δ_{int}) functions are implemented as in [9].

In the initialization function, the cell's future value f is initialized to its initial value v, which is also copied in *f-record*. Notice that the cell needs to explicitly make a copy of its state, *state-record*, and the duration of the state, *delay-record*. The other operations are the same as in the initialization function for cells with transport delay.

When the δ_{ext} is invoked, the cell detects time changes and does the housekeeping operations at the beginning of R_0 for each simulation time in the manner of an atomic model with transport delay. It records the current f for reference in the following rounds of state transitions at this time. In addition, it copies the current state and the duration of that state in *state-record* and *delay-record*, respectively. The user-defined state variable and external events are handled in the same way as in transport-delay cells.

The preemption is done once a change in f is detected (line 38) and the operations are carried out in two steps: step one preempts the current state of the cell along with its duration (line 39 to 53), and step two preempts the current f of the cell (line 54). If the cell's current state is passive, the state preemption is carried out via the *holdIn* function (line 40). This operation is common for both preemption of events which occurred at different times (i.e. later events preempt earlier events, referred to as *preemption-A*) and preemption of events which occurred at the same time but in different rounds of a transition phase (i.e. events in a later round preempt those in previous rounds, referred to as *preemption-B*). If the cell's current state is active, it should be preempted differently depending on whether the preemption occurs in *preemption-A* or in *preemption-B*. The operations for *preemption-A* (line 43 to 45) are defined by the semantics of the inertial delay [3]. On the other hand, the cell's f may or may not be changed to the *f-record* during the following rounds in *preemption-B*. If it is not changed to *f-record* (line 42), the preemption logic is the same as in *preemption-A*. Otherwise, the cell's current state is recovered to the *state-record* (line 47 to 51). Notice that the duration of the state is also recovered to the *delay-record* (line 50). The recovery can only occur in the multiple rounds of a transition phase, as secured by the condition in line 46.

```
1. when the initialization function is invoked
2.     get the cell's initial value v
3.     event-flag = false
4.     transient-value = v
5.     time-record = 0
6.     state-variable-record = state-variable
7.     push <0 / out = v> into the queue
8.     holdIn(active, 0)
9. end when

10. when the δext function is invoked
11.    set the values of neighboring cells based on the current message bag
12.    time-change = false
13.    if time-record != current-time then
14.           time-record = current-time
15.           time-change = true
16.           get the previous-value from the input port
17.           transient-value = previous-value
18.           event-flag = false
19.           state-variable-record = state-variable
20.    end if
21.    state-variable = state-variable-record
22.    if there are external events in the current message bag  then
23.           new-value = port-in-function()
24.    else
25.           new-value = local-transition-function()
26.    end if
27.    output-time = current-time + delay
28.    if event-flag = false then
29.           if new-value is derived from external events  then
30.                  event-flag = true
31.           end if
32.           if (new-value != previous-value) & (transient-value = previous-
                  value)
       then
33.                  transient-value = new-value
34.                  push <output-time / out = new-value> into the queue
35.                  holdIn(active, time to the next output)
36.           else if (new-value = previous-value)&(transient-value != previous-
                     value)
       then
37.                  transient-value = new-value
38.                  if time-change = false then
39.                         remove the previous element from the queue
40.                         if queue is empty then
41.                                passivate()
42.                         else
43.                                holdIn(active, time to the next output)
44.                         end if
45.                  end if
46.           else if (new-value != previous-value) &
                      (transient-value != previous-value) &
                      (new-value != transient-value)  then
47.                  transient-value = new-value
48.                  if time-change = false then
49.                         replace the previous element in the queue
50.                         holdIn(active, time to the next output)
51.                  end if
52.           end if
53.    end if
54. end when
```

Figure 10. Algorithms for the Initialization and δ_{ext} functions in Cell-DEVS models with transport delay

```
1. when the initialization function is invoked
2.     get the cell's initial value v
3.     event-flag = false
4.     f = v
5.     f-record = f
6.     state-record = passive
7.     delay-record = infinity
8.     time-record = 0
9.     state-variable-record = state-variable
10.    holdIn(active, 0)
11. end when

12. when the δext function is invoked
13.    set the values of neighboring cells based on the current message bag
14.    time-change = false
15.    if time-record != current-time then
16.           time-record = current-time
17.           time-change = true
18.           event-flag = false
19.           f-record = f
20.           state-variable-record = state-variable
21.           state-record = current-state
22.           if current-state = active then
23.                  delay-record = tN - current-time
24.           else
25.                  delay-record = infinity
26.           end if
27.    end if
28.    state-variable = state-variable-record
29.    if there are external events in the current message bag then
30.           new-value = port-in-function()
31.    else
32.           new-value = local-transition-function()
33.    end if
34.    if event-flag = false then
35.           if new-value is derived from external events then
36.                  event-flag = true
37.           end if
38.           if new-value != f then
39.                  if current-state = passive then
40.                         holdIn(active, delay)
41.                  else
42.                         if new-value != f-record then
43.                                if tN - current-time > 0 then
44.                                       holdIn(active, delay)
45.                                end if
46.                         else if time-change = false then
47.                                if state-record = passive then
48.                                       passivate()
49.                                else
50.                                       holdIn(active, delay-record)
51.                                end if
52.                         end if
53.                  end if
54.                  f = new-value
55.           end if
56.    end if
57. end when
```

Figure 11. Algorithms for the Initialization and δ_{ext} functions in Cell-DEVS models with inertial delay

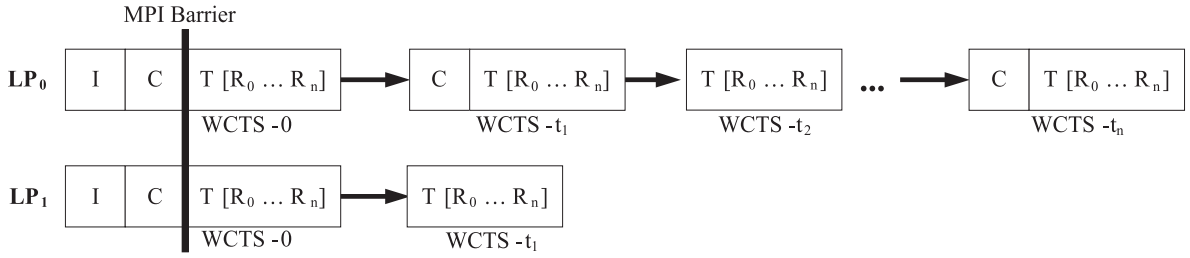

Figure 12. Using MPI Barrier to avoid rollbacks at virtual time 0 in PCD++

```
2.1. if (t = 0) & (D.ta = 0) & (next-message-type = *) then
2.2.   call kernel service function synchronizeLPs()
2.3. end if
```

Figure 13. Code snippet for handling rollbacks at time 0 in the NC algorithm for (D, t)

6. Enhancements to PCD++ and Warped Kernel Algorithms

Before the PCD++ simulator can be used to carry out optimistic simulations in parallel and distributed environments, it must be enhanced to address a variety of issues. This section covers the essential enhancements to the PCD++ and the WARPED kernel to ensure correct and efficient execution of simulations.

6.1 Rollbacks at Virtual Time 0

During rollbacks, the state of a process is restored to a previously saved copy with virtual time strictly less than the rollback time. However, the problem of handling rollbacks at virtual time 0 is left unsolved in the WARPED kernel. If a process receives a straggler with timestamp 0, the state restoration will fail since no state with negative virtual time can be found in its state queue. There are two different approaches to solving this problem. One is to save a special state that has an artificial negative virtual time at the head of each state queue, and let the process bounce back from it using the standard rollback mechanism. The other is to synchronize the processes at an appropriate stage with MPI Barriers so that no straggler message with timestamp 0 will ever be received by any process in the simulation.

The former approach is purely optimistic in the sense that no explicit synchronization is used. However, there is a performance hazard in this approach. The probability of *rollback echoes* [5] increases significantly at virtual time 0. In this case, the processes in the system are forced to restart execution from time 0 repeatedly, resulting in an unstable situation where there is no progress in simulation time as the simulation proceeds.

The second approach tries to avoid the problem altogether using explicit synchronizations. In PCD++, the best place to implement the MPI Barrier is after the collect phase in WCTS-0, as illustrated in Figure 12.

The underlying assumption in this approach is that all outgoing inter-LP communication happens only in the collect phase before the corresponding transition phase at any given virtual time. Hence, messages with timestamp 0 are sent to remote LPs only in the collect phase of WCTS-0. The LPs are synchronized by a MPI Barrier at the end of this collect phase so that these messages can be received by their destinations before the simulation time advances beyond time 0. Therefore, no straggler with timestamp 0 will be received by any LP afterwards. Once the LPs exit from the barrier, they can safely continue optimistic execution based on the standard rollback mechanism. The states saved for the events executed at virtual time zero provide the necessary cushion for later rollbacks on the processes. The cost of this approach is small, since the length of the synchronized execution is trivial when compared with the whole simulation.

The following pseudo-code snippet (Figure 13) is inserted between lines 2 and 3 in the NC algorithm for (D, t) (Figure 5) to implement this approach, where *synchronizeLPs* is a service function added to the warped kernel to realize the MPI Barrier. The end of the collect phase in WCTS-0 is detected by the NC using three conditions: (1) the current simulation time is zero; (2) the value of *sigma* (ta) in the received (D, t) is also zero; and (3) the current *next-message-type* is *.

However, this approach also precludes simulation of some DEVS models that allow the definition of closed

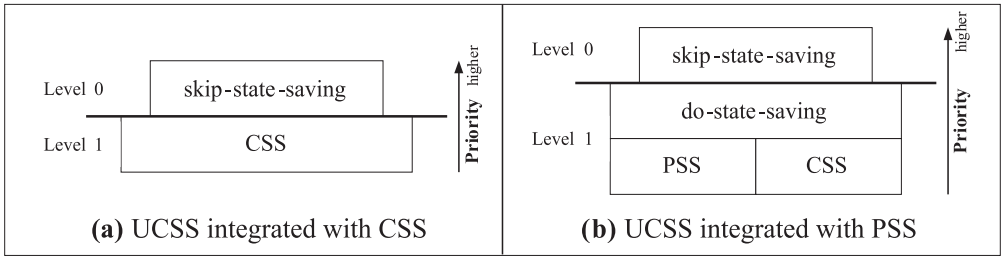

Figure 14. UCSS structure with CSS and PSS strategies

feedback loops with transient states. That is, the output port of a component is connected directly to an input port of the same component or indirectly through other components that have states with zero ta functions. In this case, a LP may still receive straggler messages with timestamp 0 that are echoed back from other LPs after the MPI Barrier, nullifying the above mechanism. Handling rollbacks at virtual time 0 and, in general, optimistic simulations for this type of DEVS models will be investigated further in our future research.

6.2 User-Controlled State-Saving (UCSS) Mechanism

In the WARPED kernel, the *copy state-saving* (CSS) strategy is implemented using state managers of type *StateManager* which saves the state of a process after executing each event. The *periodic state-saving* (PSS) strategy is realized using state managers of type *InfreqStateManager* that only saves the state of a process infrequently every a number of events. Simulator developers can choose to use either type of state managers at compile time. Once selected, all the processes will use the same type of state managers throughout the simulation. This rigid mechanism has two major disadvantages: (1) it ignores the fact that simulator developers may have the knowledge as to how to save states more efficiently to reduce the state-saving overhead; and (2) it eliminates the possibility that different processes may use different types of state managers to fulfill their specific needs at runtime. To overcome these limitations, we introduced a two-level *user-controlled state-saving* (UCSS) mechanism in the kernel so that simulator developers can utilize more flexible and efficient state-saving strategies at runtime.

A flag called *skip-state-saving* is defined in each simulation object. The kernel algorithm is modified so that the CSS policy only takes effect when the flag is false. Otherwise, no state is saved after executing the current event. Instead, the flag is reset to false so that a new state-saving decision can be made during the execution of the next event. That is, the UCSS operates on an event-by-event basis for each simulation object. When the PSS strategy is used, an additional flag called *do-state-saving* with a lower priority is defined in the *InfreqStateManager* associated with each simulation object. The state-saving algorithm is modified so that, if this flag is set to true by a simulation object, the *InfreqStateManager* saves states after every event, as does the *StateManager* under the CSS strategy. By default, both *skip-state-saving* and *do-state-saving* are false. The structure of the UCSS mechanism is shown in Figure 14.

A PCD++ processor can therefore make state-saving decisions based on application-specific criteria by setting the *skip-state-saving* flag at level zero. Further, it can dynamically switch between the CSS and PSS strategies by virtue of the *do-state-saving* flag at level 1. Thus, the UCSS mechanism virtually gives simulator developers the full power to dynamically choose the best possible combination of state-saving strategies at runtime.

6.3 Messaging Anomalies

In PCD++, the NC calculates the next simulation time (*min-time* in Figure 5) based on the time of its *NC Message Bag*. However, more lagging external messages with timestamp less than the *min-time* may arrive after the calculation, invalidating the previous computation result. In this case, the NC's speculative calculation of the *min-time* leads to messaging anomalies that cannot be recovered by the kernel rollback mechanism alone. Messaging anomalies will be detected when the control returns to the NC in the transition phase at the next (wrong) simulation time. Once found, the NC needs to perform cleanup operations to restore the simulation to the status before the previous wrong computation. An example scenario is shown in Figure 15, where the simulation on the LP involves three PCD++ processors (the Simulator is labeled as S1). The execution sequence of the messages is denoted by the numbers in the diagram. Only the final portion of WCTS-t_a is illustrated.

Suppose that when the last done message (D_1) is executed by the NC at the end of WCTS-t_a, there is no external message in its NC Message Bag and the closest state transition time carried in D_1 is t_b. Hence, the NC calculates the *min-time* as t_b, and sends a collect message ($@_5$)

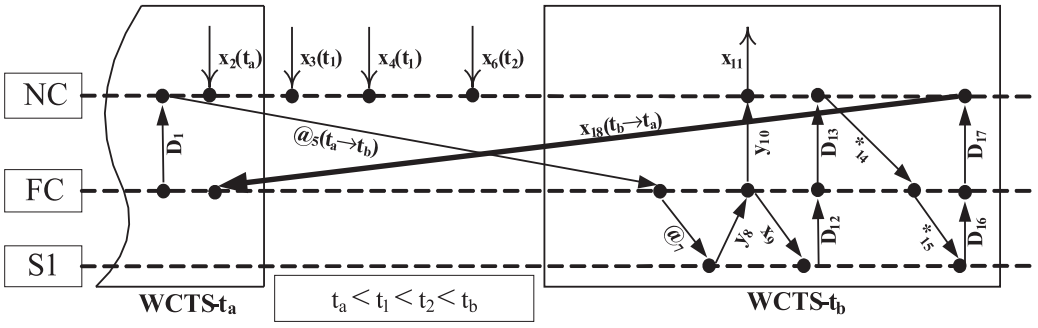

Figure 15. Example scenario of messaging anomalies

with send time t_a and receive time t_b to the FC, initiating WCTS-t_b on the LP. Meanwhile, more lagging external messages (x_2, x_3, x_4 and x_6) with timestamp less than t_b arrive at the NC, invalidating the previously computed *min-time* t_b. Thus, the linking messages between WCTS-t_a and WCTS-t_b (e.g. $@_5$) are proven to be *false messages*. During the execution of D_{17} at the end of R_0 in WCTS-t_b, the NC calculates the *min-time* again, based on its present NC Message Bag which now contains the lagging external messages. The resulting *min-time* is t_a, the timestamp of x_2. Hence, the NC sends an external message (x_{18}) with send time t_b and receive time t_a ($t_b > t_a$) to the FC. Since x_{18} is a straggler message for the FC, rollbacks propagate from the FC to the other processors immediately. Nonetheless, these rollbacks violate two assumptions made by the WARPED kernel. (1) The rollback at the FC is triggered by an *abnormal straggler message* (x_{18}) with a send time greater than its receive time. Since the events are ordered by their send time in the output queues, this abnormal straggler message is misplaced in the NC's output queue, resulting in causality errors and runtime crash later on during the simulation. (2) The rollbacks occur right in the middle of executing the done message (D_{17}) by the NC. Therefore, the rollbacks are not transparent to the NC anymore.

Messaging anomalies can be classified into two categories. (1) In Figure 16(a), if there are lagging external messages with timestamp t_a, e.g. $x(t_a)$, inserted into the NC Message Bag, the abnormal straggler message sent to the FC will have a timestamp of t_a. Hence, the processors are rolled back to the end of WCTS-t_{pre}, the WCTS before WCTS-t_a. In this case, all the lagging external messages are removed from the NC Message Bag and no erroneous data is left in the state queues. This type of messaging anomalies is referred to as *anomaly with empty NC Message Bag*. (2) In Figure 16(b), if no lagging external message with timestamp t_a has arrived at the NC, the abnormal straggler message will have a timestamp of t_1 ($t_1 > t_a$). Hence, the processors are rolled back to the end of WCTS-t_a, and the lagging external messages remain in the NC Message Bag after the kernel rollbacks. This type of messaging anomaly is referred to as *anomaly with non-empty NC Message Bag*.

Figure 17 shows the cleanup operations for anomalies with empty NC Message Bag. First, the abnormal straggler message is removed from both the NC's output queue and the FC's input queue. A new function, *removeStragglerEvent*, is defined in the kernel for removing the positive straggler from the input queue. Secondly, the state-saving operation is skipped after processing the current done message (e.g. D_{17} in Figure 15). This is achieved by virtue of the UCSS mechanism (line 4).

The simplified cleanup operations for anomalies with non-empty NC Message Bag are given in Figure 18. The abnormal straggler is removed from both the NC's output queue and the FC's input queue (line 2 to 3). The false messages are identified (line 4). The undue external events sent along with the false messages, if any, are unprocessed (line 5 to 8). The false messages are removed from the queues with the abnormal straggler (line 9 and 10). The state saved at the end of WCTS-t_a is removed from the NC's state queue to erase the incorrect data contained in that state (line 11 and 12). A flag called *breakpoint* is introduced in the kernel abstract state definition to allow tracing of the occurrence of previous messaging anomalies in future rollbacks. The NC sets the breakpoint flag to true to leave a tag in its state queue (line 14). Then, the external messages with time t_1 are resent to the FC (line 15 to 18) with the correct send and receive time. The lagging external messages with timestamp greater than t_1 are removed from the NC Message Bag and unprocessed in the input queue (line 18 to 23). These messages will be reprocessed when their virtual time comes. Finally, the NC sends a (*, t) to the FC to initiate a new WCTS-t_1 on the LP (line 24).

The enhanced NC algorithm for (D, t) combines the logic for normal execution (Figure 5) with the algorithms for handling rollbacks at time 0 (Figure 13) and both types of messaging anomalies (Figure 17 and Figure 18).

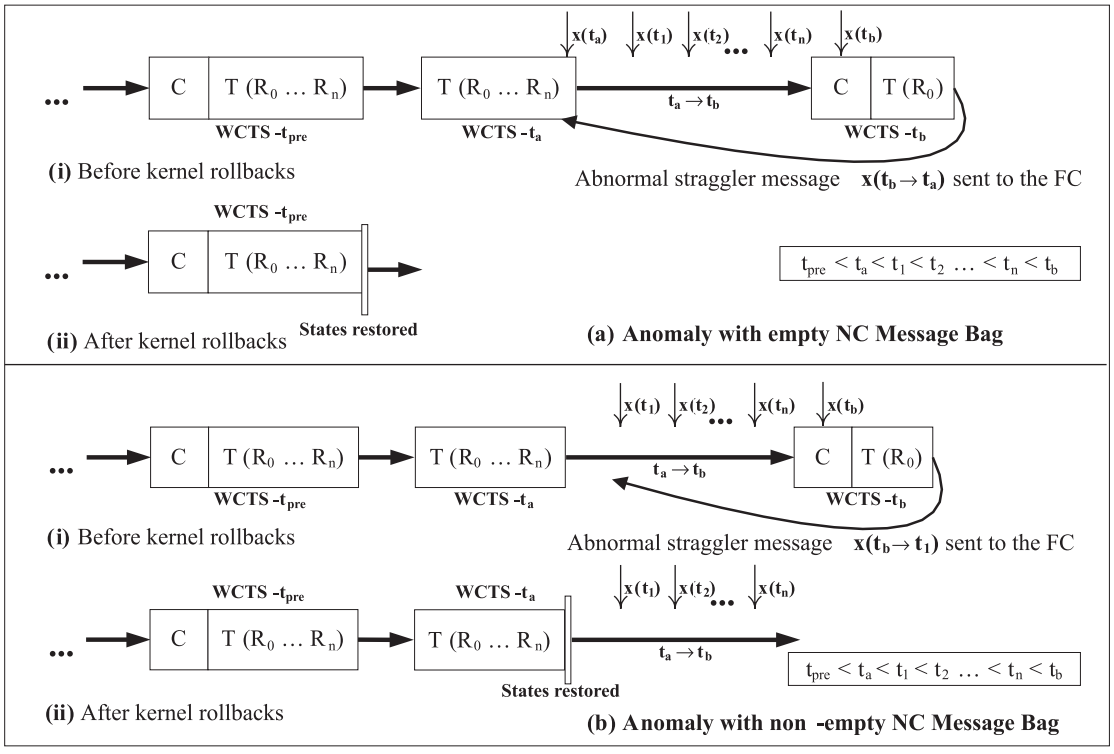

Figure 16. Two types of messaging anomalies in PCD++

```
1. when cleanup operations for anomaly with empty NC Message Bag is invoked
2.     abnormal-straggler = NC's output queue→remove(tail)
3.     FC's input queue→removeStragglerEvent(abnormal-straggler)
4.     skip-state-saving = true
5. end when
```

Figure 17. NC algorithm for handling anomaly with empty NC Message Bag

7. PCD++ Optimizations

Various optimization strategies have been integrated into the PCD++ simulator. In this section, we discuss two optimizations that are specifically implemented for PCD++, including a risk-free *message type-based state-saving* (MTSS) strategy to reduce the number of states saved during the simulation, and a one log file per node strategy to break the bottleneck caused by file I/O operations.

7.1 Message Type-based State Saving (MTSS)

During rollbacks, the state of a PCD++ processor is always restored to the *last* state saved at the end of a WCTS with virtual time strictly less than the present rollback time. Hence, it is sufficient for a processor to save its state only after processing the last event in each WCTS for rollback purposes. The state-saving operation can be safely skipped after executing all the other events. The last event in a WCTS is processed at the end of R_n in the transition phase. Although the actual number of rounds in a transition phase cannot be determined with certainty, we can at least identify the type of the messages executed at the end of the transition phases by a given processor. For the NC and FC, it must be a (D, t) and for the Simulators, it should be a (*, t). Therefore, PCD++ processors need to save states only after processing these particular types of messages. Since the Root only processes output messages, it still saves state for each event. The resultant state-saving

```
1. when cleanup operations for anomaly with non-empty NC Message Bag is invoked
2.     abnormal-straggler = NC's output queue→remove(tail)
3.     FC's input queue→removeStragglerEvent(abnormal-straggler)
4.     false-messages = FC's input queue→findFalseMessages(t_b)
5.     undue-external-events = findUndueExternalEvents(false-messages)
6.     if the size of undue-external-events > 0 then
7.           rollbackExternalEvents(undue-external-events)
8.     end if
9.     NC's output queue→remove(false-messages)
10.    FC's input queue→removeStragglerEvent(false-messages)
11.    false-state = NC's state queue→tail
12.    NC's state queue→remove(false-state)
13.    NC's current state→LVT = t_1
14.    NC's current state→breakpoint = true
15.    for each external message x with timestamp t_1 in the NC Message Bag do
16.          resend (x, t_1) to the child FC
17.    end for each
18.    remove all x in the NC Message Bag with timestamp t_1
19.    if the size of the NC Message Bag > 0 then
20.          reset the NC's LVT = t_1
21.          unprocess the external messages remained in the NC Message Bag
22.          empty the NC Message Bag
23.    end if
24.    send (*, t_1) to the child FC
25.    next-message-type = @
26. end when
```

Figure 18. NC algorithm for anomalies with non-empty NC Message Bag (simplified)

strategy is referred to as *message type-based state-saving* (MTSS), a specific type of UCSS for the PCD++ toolkit. Considering that there are a large number of messages executed in each WCTS, and that they are dominated by external and output messages, MTSS can significantly reduce the number of states saved during the simulation when compared to the original CSS strategy. Further, the rollback overhead is also reduced as fewer states need to be removed from the state queues during rollback operations. Unlike the PSS strategy, MTSS is risk-free in the sense that there is no penalty for saving fewer states.

The MTSS strategy can be easily implemented using the UCSS mechanism. A PCD++ processor simply sets the *skip-state-saving* flag to true in all but the algorithm for the required type of messages. For example, a Simulator will set the flag to true in its algorithms for (I, t), (@, t), and (x, t). This flag is left untouched with value false in its algorithm for (*, t) since the Simulator should save its state after processing such type of messages.

7.2 One Log File per Node

Previously, one log file is created for each PCD++ processor to log the received messages in a human readable format. Depending on the size of the model, this can consume many file descriptors. In addition, creating these files and transferring data to them constitute a large operational overhead, especially when the files are accessed via a Network File System (NFS) during the simulation. When considering the overhead in Time Warp optimistic simulations, the cost is prohibitive since one file queue is maintained in the kernel for each of these files and all the file queues participate in rollback operations.

To reduce the overhead of file I/O operations, a new optimization strategy referred to as *one log file per node* is implemented in the toolkit. Based on this strategy, only one log file is created for the NC on each node. The NC's file queue is shared among all the processors on that node. Messages received by the NC itself are logged directly in the NC's file queue, while the other processors on that node must first get a reference to the local NC (which can be done in constant time) and then log their received messages into the NC's file queue.

There are several advantages associated with the one log file per node strategy. (1) The required number of file descriptors for logging purposes is upper-bounded by the number of machines used in the simulation, rather than increasing linearly with the size of the model. (2) The simulation bootstrap time is reduced considerably due to the dramatic decrease in the number of files opened in this process. (3) The kernel rollback operations are accelerated since only one operation is needed to restore the single

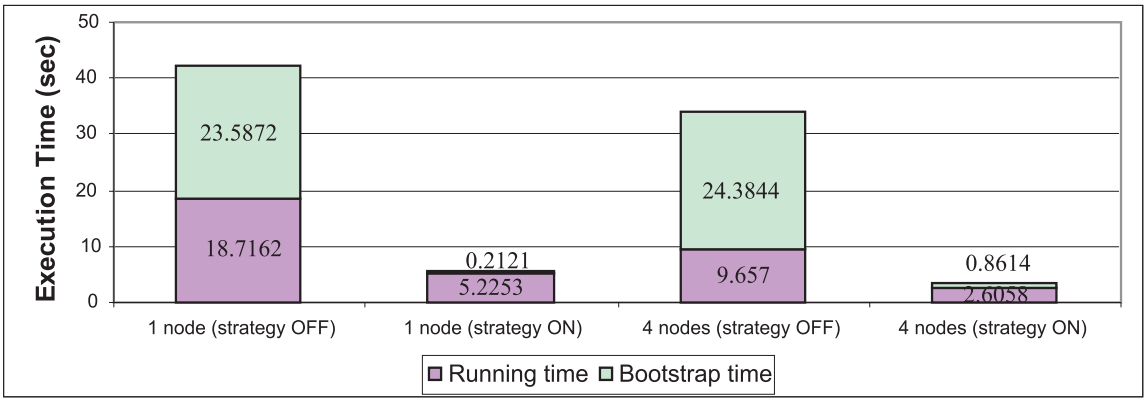

Figure 19. Execution and bootstrap time before and after one log file per node strategy on 1 and 4 nodes

file queue maintained in the kernel. (4) The communication overhead is reduced since the data concentrated in a single file queue is flushed to the physical file in bigger chunks, and less frequently, over the network.

8. Experimentation Results

Our experiments were conducted on a HP PROLIANT DL Server, 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 models tested in our experiments include a model for forest fire propagation [19] based on Rothermel's mathematical definition [20] and a 3-D watershed model representing a hydrology system originally presented by Moon et al. [21] and enhanced by Ameghino et al. [19]. The following simulation results are averaged over 10 independent runs.

We use two different speedups in our analysis. The *overall speedup* is calculated from the total execution time that reflects how much faster the simulation runs on multiple machines than it does on a single machine, as perceived by the users. The *algorithm speedup* is calculated from the actual running time (i.e. without considering the simulation bootstrap time) that is used to assess the performance gain attributed to the parallel algorithms alone.

8.1 Effect of One Log File per Node

The performance improvement derived from the one log file per node strategy is tested using the fire propagation model of 900 cells arranged in a 30 × 30 mesh. The model was executed on 1 and 4 nodes with and without using the strategy to simulate the behavior of forest fire during a period of 5 hours. Results are depicted in Figure 19.

Notice that the bootstrap time is even greater than the actual running time when the strategy is turned off. This clearly indicates that the bootstrap operation is a bottleneck in the simulation. When the strategy is turned on, the bootstrap time is reduced by 99.1% on 1 node and by 96.47% on 4 nodes. Further, the running time is decreased by 72.08% on 1 node and by 73.02% on 4 nodes due to more efficient communication, I/O and rollback operations.

The CPU usage monitored in our experiments also suggests that the file I/O operation is a major barrier in the bootstrap phase. As shown in Figure 20, the CPU is utilized much more efficiently with the one log file per node strategy. A similar pattern was observed in simulations running on multiple nodes.

8.2 Effect of MTSS

The same fire propagation model is used to test the effect of MTSS strategy. The model was executed on 1 and 4 nodes (respectively) with and without the MTSS strategy (respectively); results are depicted in Figure 21.

Due to the MTSS strategy, the number of states saved during the simulation is reduced by 49.29% and 47.74% on 1 and 4 nodes, respectively. Accordingly, the time spent on state-saving operations is decreased by 29.9% and 27.76%.

The corresponding running time and bootstrap times are shown in Figure 22. While the bootstrap time remains nearly unchanged in both cases, the actual running time is reduced by 17.64% and 7.63% on 1 and 4 nodes, respectively, because fewer states are saved in the state queues and, potentially, removed from the queues during rollbacks.

Figure 23 shows the time-weighted average and maximum memory consumption with and without the strategy for the fire propagation model on 1 and 4 nodes. The average memory consumption declines by 26% in both

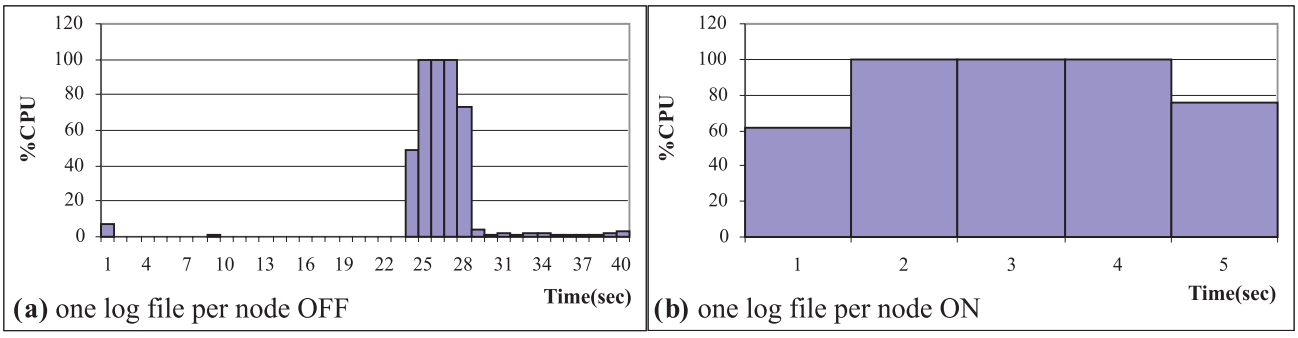

Figure 20. CPU usage before and after one log file per node strategy on 1 node

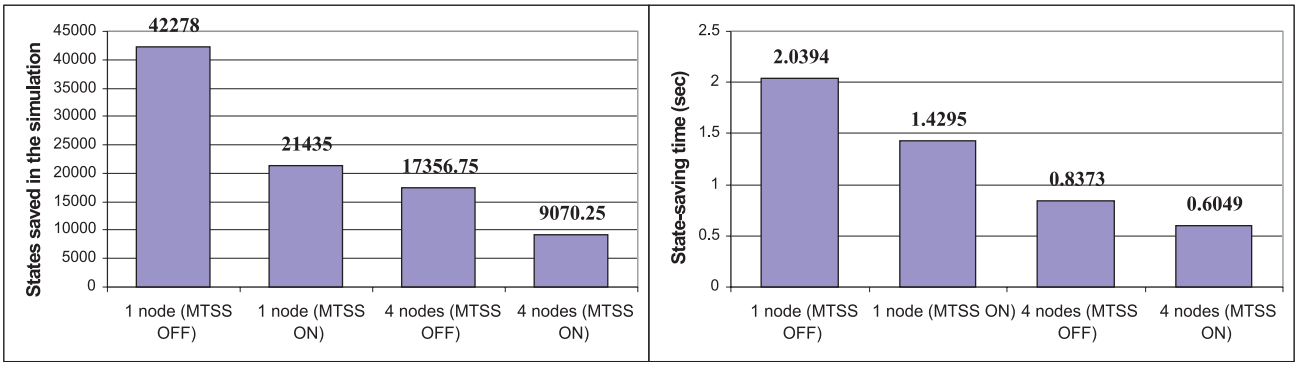

Figure 21. Number of saved states and state-saving time before and after MTSS strategy on 1 and 4 nodes

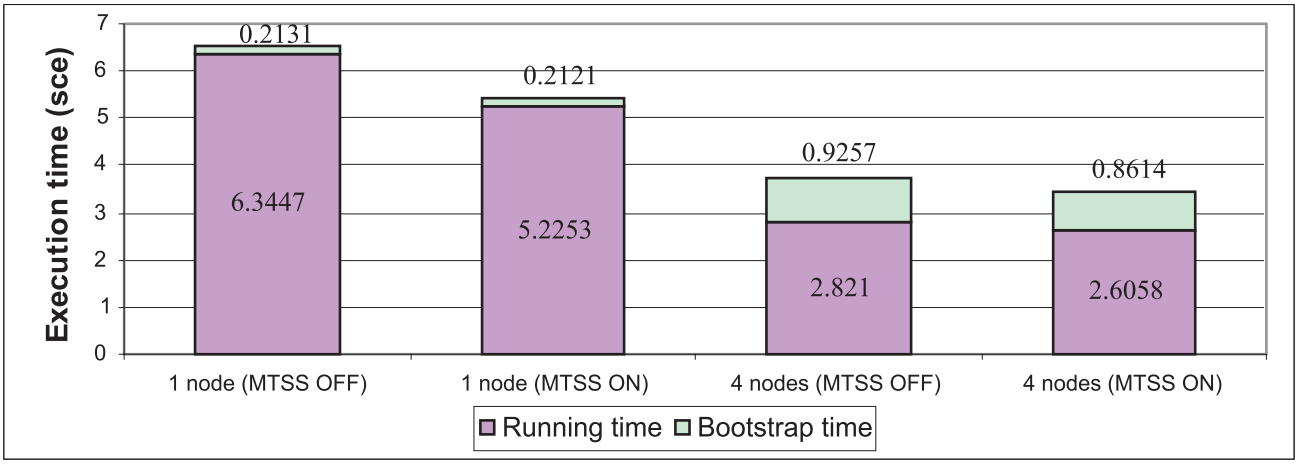

Figure 22. Running and bootstrap time before and after MTSS strategy on 1 and 4 nodes

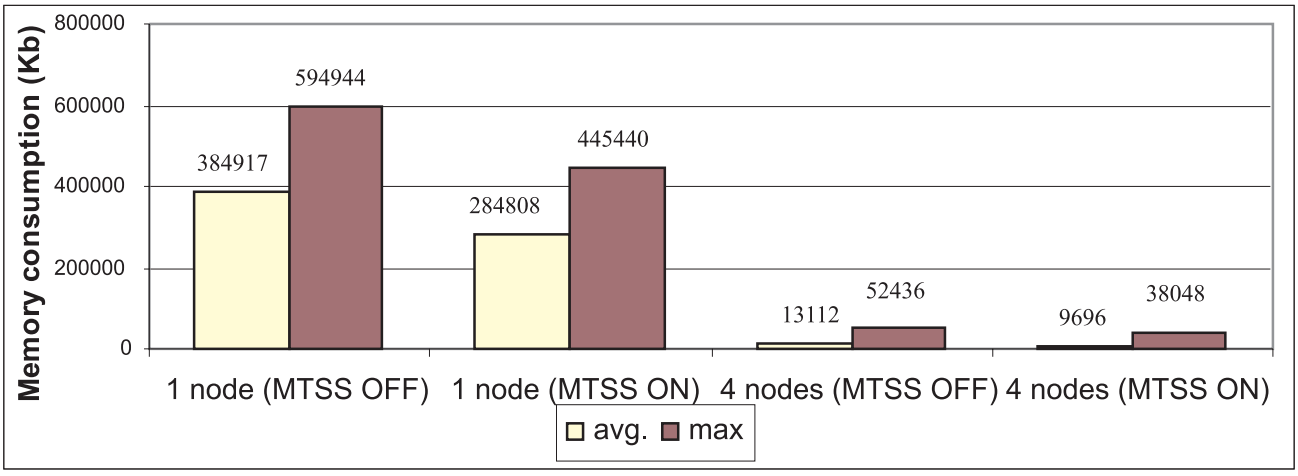

Figure 23. Average and maximum memory consumption before and after MTSS strategy

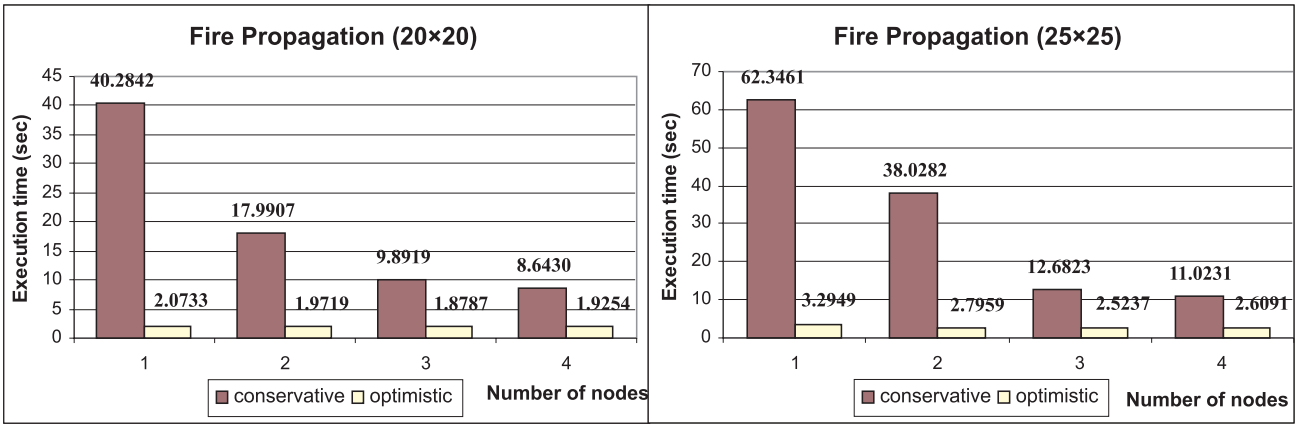

Figure 24. Comparison between optimistic and conservative simulators using the fire model

cases, while the peak memory consumption decreases by 25.13% and 27.44% on 1 and 4 nodes, respectively.

8.3 Performance of the PCD++ Toolkit

The key metrics for evaluating the performance of the PCD++ simulator are the execution time and speedup. Both the one log file per node and MTSS strategies were applied to the simulator in the following experiments. For all the Cell-DEVS models, a simple partition strategy was used that evenly divides the cell space into horizontal rectangles. First, the fire propagation model was tested using different sizes of cell spaces: 20 × 20 (400 cells), 25 × 25 (625 cells), 30 × 30 (900 cells) and 35 × 35 (1225 cells).

Figure 24 shows a comparison between the PCD++ optimistic simulator and the previous conservative simulator [9] for different model sizes on a set of compute nodes. In all cases, the optimistic simulator markedly outperforms the conservative simulator. The total execution time and running time of the fire model with different sizes, executed on 1 up to 4 nodes, is listed in Table 1.

For any given number of nodes, the execution time always increases with the size of the model. Moreover, the execution time rises less steeply when more nodes are used in the simulation. For example, as the model size increases from 400 to 1225 cells, the execution time increases sharply by nearly 280% (from 2.0733 to 7.8702 s) on 1 node, whereas it merely rises by 98% (from 1.9254 to 3.8138 s) on 4 nodes. On the other hand, for a fixed model

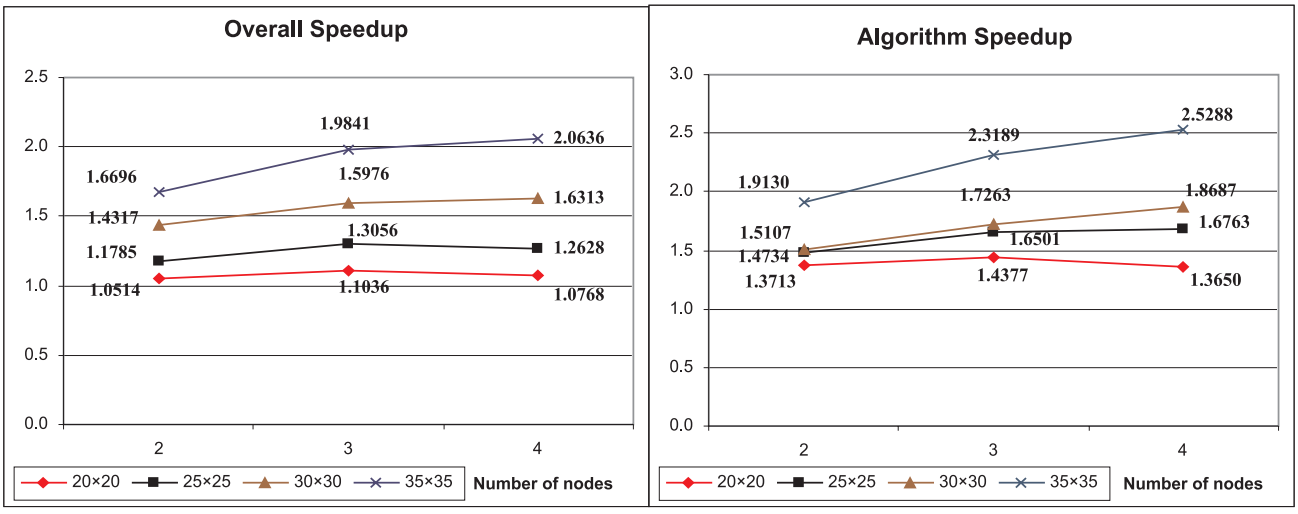

Figure 25. Overall and algorithm speedups for fire model of various sizes on a set of nodes

Table 1. Execution time and running time for fire model of various sizes on a set of nodes

Total execution time (sec)					Running time (sec)				
Number of nodes	**20×20**	**25×25**	**30×30**	**35×35**	**Number of nodes**	**20×20**	**25×25**	**30×30**	**35×35**
1	2.0733	3.2949	5.0442	7.8702	**1**	1.9515	3.1273	4.3566	7.6428
2	1.9719	2.7959	3.5232	4.7138	**2**	1.4232	2.1225	2.8838	3.9952
3	1.8787	2.5237	3.1573	3.9667	**3**	1.3574	1.8953	2.5237	3.2959
4	1.9254	2.6091	3.0922	3.8138	**4**	1.4296	1.8656	2.3314	3.0224

size, the execution time tends to (but not always) decrease when more nodes are utilized. The execution time for the 20 × 20 model decreases from 2.0733 to 1.8787 s when the number of nodes climbs from 1 to 3. However, when the number of nodes increases further, the downward trend in execution time is reversed. It increases slightly from 1.8787 to 1.9254 s as the number of nodes rises from 3 to 4. When a model, especially a small one, is partitioned onto more and more nodes, the increasing overhead involved in inter-LP communication and potential rollbacks may eventually degrade the performance. Hence, a tradeoff between the benefits of a higher degree of parallelism and the concomitant overhead costs needs to be reached when we consider different partition strategies.

From the table, we can also find that better performance can be achieved on a larger number of nodes as the model size increases. The shortest execution time is achieved on 3 nodes for the 20 × 20 and 25 × 25 models, while it is obtained on 4 nodes for the other two larger models. It is clear that we should use more nodes to simulate larger and more complex models where intensive computation is the dominant factor in determining the system performance.

Using the execution and running time, we can calculate the overall and algorithm speedups, as shown in Figure 25. As we can see, higher speedups can be obtained with larger models. In addition, the algorithm speedup is always higher than its counterpart overall speedup, evidence showing that the Time Warp optimistic algorithms are major contributors to the overall performance improvement.

A more computation-intensive 3-D watershed model of size 15 × 15 × 2 (450 cells) was tested to evaluate the performance of PCD++ for simulating models of complex physical system. Table 2 shows the resulting total execution time and running time. The best performance is achieved on 5 nodes with execution and running time of 6.1538 and 5.6743 s, respectively.

The speedups are illustrated in Figure 26. The best overall and algorithm speedups are 2.7306 and 2.9373, respectively, higher than those obtained with the fire models.

9. Conclusion

This work tackles the problem of executing DEVS and Cell-DEVS models in parallel and distributed environ-

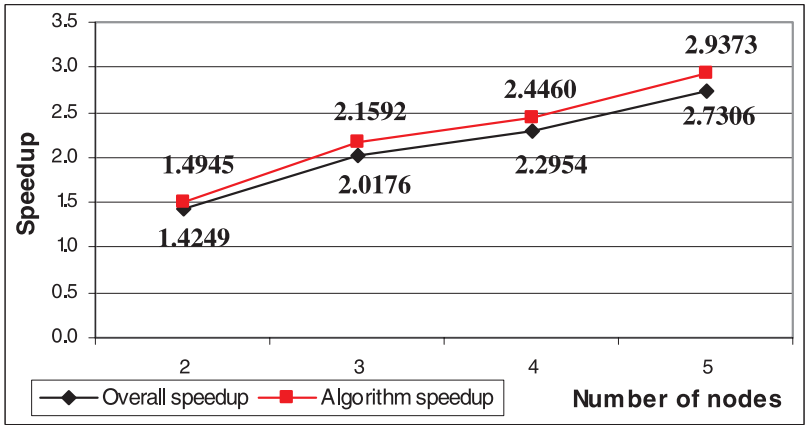

Figure 26. Overall and algorithm speedups for the 15 × 15 × 2 watershed model

Table 2. Execution time and running time for the 15 × 15 × 2 watershed model on a set of nodes

Number of nodes	1	2	3	4	5
Total execution time (sec)	16.8036	11.7930	8.3285	7.3205	6.1538
Running time (sec)	16.6668	11.1522	7.7191	6.8140	5.6743

ments based on the Time Warp synchronization protocol. A new extension to the CD++ toolkit, PCD++, was developed in our research to meet the need for faster and more efficient simulation of complex models. The algorithms for the PCD++ processors and Cell-DEVS models with transport and inertial delays were redesigned to address the need of distributed optimistic simulation. The simulation process on each LP was abstracted using the notion of WCTS, which greatly simplifies the task of analyzing the complex message exchanges between the PCD++ processors involved in the simulation. A two-level UCSS mechanism was proposed so that simulator developers can utilize more flexible and efficient state-saving techniques during the simulation. Mechanisms were provided to handle various issues in optimistic simulations such as rollbacks at virtual time 0 and messaging anomalies. Several optimization strategies were implemented in PCD++ such as the MTSS strategy and the one log file per node strategy. We showed that PCD++ simulator markedly outperforms the conservative simulator in all testing scenarios. Considerable speedups were observed in our experiments, indicating the optimistic simulator is well suited for simulating large and complex models.

10. Acknowledgments

This research has been partially supported by NSERC, CFI (Canadian Foundation for Innovation), OIT (Ontario Innnovation Fund) and Precarn.

11. References

[1] Zeigler, B., T. Kim, and H. Praehofer. 2000. *Theory of modeling and simulation: Integrating discrete event and continuous complex dynamic systems*. 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. and N. Giambiasi. 2002. N-dimensional Cell-DEVS models. *Discrete Event Dynamic Systems* **12**(2): 135–157.

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

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

[6] Jefferson, D. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems* **7**(3): 405–425.

[7] 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*, *Lecture Notes In Computer Science*, 1505: 13–23.

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

[9] Troccoli, A. and G. Wainer. 2003. Implementing parallel Cell-DEVS. In *Proceedings of the 36th IEEE/SCS Annual Simulation Symposium*, Orlando, FL.

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

[11] Wainer, G. 2000. Improved cellular models with parallel Cell-DEVS. *Transactions of the Society for Computer Simulation International* **17**(2): 73–88.

[12] Zeigler, B., D. Kim, and S. Buckley. 1999. Distributed supply chain simulation in a DEVS/CORBA execution environment. In *Pro-*

ceedings of the 1999 Winter Simulation Conference, Phoenix, AZ.

[13] Zeigler, B., and H. S. Sarjoughian. 1999. Support for hierarchical modular component-based model construction in DEVS/HLA. *Simulation Interoperability Workshop*, Orlando, FL.

[14] Kim, K., and W. Kang. 2004. 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*, Assisi, Italy.

[15] Seo, C., S. Park, B. Kim, S. Cheon, and B. Zeigler. 2004. Implementation of distributed high-performance DEVS simulation framework in the Grid computing environment. *Advanced Simulation Technologies Conference – High-Performance Computing Symposium*, Arlington, VA.

[16] Cheon, S., C. Seo, S. Park, and B. Zeigler. 2004. Design and implementation of distributed DEVS simulation in a peer to peer network system. *Advanced Simulation Technologies Conference – Design, Analysis, and Simulation of Distributed Systems Symposium*, Arlington, VA.

[17] Zhang, M., B. Zeigler, and P. Hammonds. 2006. DEVS/RMI – An auto-adaptive and reconfigurable distributed simulation environment for engineering studies. *Spring Simulation Multiconference – DEVS Integrative M&S Symposium*, Huntsville, AL.

[18] Nutaro, J. 2004. Risk-free optimistic simulation of DEVS models. *Advanced Simulation Technologies Conference – Military, Government, and Aerospace Simulation Symposium*, Arlington, VA.

[19] Ameghino, J., A. Troccoli, and G. Wainer. 2001. Models of complex physical systems using Cell-DEVS. In *Proceedings of the 34th IEEE/SCS Annual Simulation Symposium*, Seattle, WA.

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

[21] Zeigler, B. and Y. Moon. 1996. DEVS representation and aggregation of spatially distributed systems: speed-versus-error tradeoffs. *Transactions of the Society for Computer Simulation International* **13**(4): 179–189.

Qi Liu *received his B. Eng. (1993) from the Huazhong University of Science and Technology, China. He received an M.A.Sc. degree from Carleton University, Ottawa, ON, Canada (2006, Senate Medal for Outstanding Academic Achievement). He is currently a Ph.D. candidate in the Department of Systems and Computer Engineering of Carleton University. His research interests are centered around high-performance computing in simulation and advanced parallel/distributed simulation algorithms.*

Gabriel Wainer *(Senior Member, SCS) received the M.Sc. (1993) and Ph.D. degrees (1998, with highest honors) from the Universidad de Buenos Aires, Argentina, and Université d'Aix-Marseille III, France. In July 2000, he joined the Department of Systems and Computer Engineering, Carleton University (Ottawa, ON, Canada), where he is now an Associate Professor. He is the author of three books and over 140 research articles. He is Associate Editor of the Transactions of the SCS, and the International Journal of Simulation and Process Modeling. He is a chairman of the DEVS standardization study group (SISO), and a member of Carleton University Center for advanced Simulation and Visualization (V-Sim). He is Director of the Ottawa Center of The McLeod Institute of Simulation Sciences and chair of the Ottawa M&S Net. His current research interests are related to modeling methodologies and tools, parallel/distributed simulation and real-time systems.*