Parallel Simulation of DEVS and Cell-DEVS Models in PCD++

Gabriel A. Wainer, Qi Liu and Shafagh Jafer
Department of Systems and Computer Engineering
Carleton University Centre on advance Visualization and Simulation (V-Sim)
Carleton University . 1125 Colonel By Drive. Ottawa, ON K1S 5B6, Canada

1. Introduction

As we can see in the numerous examples found throughout this book, DEVS modeling and simulation (M&S) has become a widely used tool for tackling complex problems and supporting efficient decision-making in a broad array of domains. Nevertheless, as the system under study becomes more and more complex, the simulations tend to be time-consuming and resource-demanding. In the quest for better performance, parallel and distributed simulation technologies have received increasing interest, as these technologies allow executing simulations on a computing system using multiple processors interconnected by a communication network.

A parallel or distributed simulation typically comprises a collection of concurrent processes, each modeling a different part of the physical system and executing on a dedicated process, interact with each other by exchanging time-stamped event messages. The subtask executed by each process consists of a sequence of event computations, where each computation may modify the state of the process and/or schedule new events that need to be executed on the present process or on other processes. Unlike sequential simulations (which ensure that all events are simulated in time stamp order) parallel and distributed simulations use varied resources executing concurrently at different speeds, thus, we need to employ advanced synchronization mechanisms in order to guarantee same results obtained with sequential execution.

Such synchronization is key to parallel and distributed simulation. It ensures that each process complies with the *local causality constraint* [1], which requires that events are processed in non-decreasing time stamp order. Errors resulting from out-of-order event execution are referred to as *causality errors*. Synchronization techniques for Parallel Discrete Event Simulation (PDES) systems generally fall into two categories: *conservative approaches* that strictly avoid violating the local causality constraint, and *optimistic approaches* that allow violations to occur, but provide mechanisms to recover from them through an operation known as *rollback*. Usually, optimistic approaches can exploit higher degree of parallelism available in the simulation, whereas con-

servative 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. Optimistic algorithms can also execute more efficiently with such information, but this is not needed for correct execution, allowing more 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.

CD++ [2] is an open-source M&S environment that implements both P-DEVS and Cell-DEVS formalisms and has been used to successfully solve a variety of sophisticated problems (see, e.g., [3-5]). The CD++ environment has been ported to different platforms, including an embedded version [6], a standalone one, several parallel versions (conservative and optimistic synchronization protocols) [7-9], and a distributed version that supports Web-based simulations over the Internet [10]. In this chapter, we discuss the advanced techniques that have been developed for parallel simulation of DEVS and Cell-DEVS models in the PCD++ family of simulators. Specifically, we will cover the software architecture, parallel event execution paradigm, synchronization protocols, and performance optimizations in PCD++.

2. Parallel Simulation

The first parallel simulator, introduced in [7], was the first attempt to reduce simulation time in CD++ using parallel execution of models. It has been shown that this parallel simulator can speed up the execution of both DEVS and Cell-DEVS models in comparison to the stand-alone version [9]. This parallel simulator, presented in this section, was based on an approach exploiting the parallelism inherent to the DEVS formalism. Under that scheme, a single root coordinator acts as a global scheduler for every node participating in the simulation. Based on this structure, all events with the same timestamp are scheduled to be processed simultaneously on the available nodes. The simulator introduces two different types of coordinators; Head and Proxy to reduce inter-process communication. The simulator consists of a hierarchical structure creating a one-to-one correspondence between the model components and simulation objects.

2.1. Parallel DEVS abstract simulator

DEVS separates the model from the actual simulator engine. The abstract simulator creates a one-to-one correspondence between the model and the simulation entity as illustrated by Figure 1.

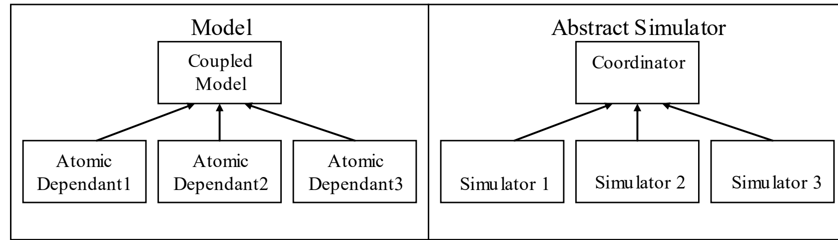

Figure 1. Correspondence between the model and the DEVS processors [11]

The simulation is carried out by DEVS processors which are of two types: simulator and coordinator. The simulator represents an atomic DEVS model, where the coordinator is paired with a coupled model. The simulator is in charge of invoking the atomic model's transition and external event function. On the other hand, the coordinator has the responsibility of translating its children's output events and estimating the time of the next imminent dependant(s). As shown in Figure 1 every coordinator has a set of child DEVS processors. When running parallel and distributed simulation, the whole model is divided among a set of logical process, each of which will execute on a different CPU. In general terms, each logical process will host one or more simulation objects. For the present discussion, those simulation objects will be DEVS processors.

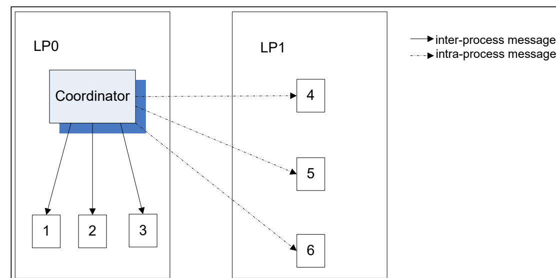

Figure 2. A single coordinator with remote and local child processes

At the beginning of the simulation, one logical process will reside on each machine (physical process). Then, each logical process will host one or more DEVS processors. This implies that not all of a coordinator's children are necessarily sitting on the same logical process. Due to the one-to-one correspondence, each coupled model is mapped to only one coordinator. A coordinator communicates with its child processors through intra-process messaging if they reside on the same logical process, and through inter-process messaging if they are sitting on remote logical processes. Figure 2 shows a scenario at which a coupled DEVS model consisting of six atomic components is simulated using this simulator. The coordinator itself and three of its child processors are on the same logical process (LP0), where the other three child processors are hosted on another logical process (LP1). When the number of remote child processors of a coordinator is

high, this design mechanism will lead to considerable overheads due to inter-process messages that are sent back and forth among the coordinator and its child processors. To overcome this issue, the concept of Head and Proxy Coordinators was introduced [7].

In the new design a coordinator is assigned with each logical process. As a result, all child processors will have a local coordinator through which they can communicate with remote child processors. The Head coordinator is responsible for synchronizing the model execution, interacting with upper level coordinators, and exchanging messages among the local and remote model components. The Proxy coordinator is responsible for message exchange among the local model components, and forwarding local components messages to the Head coordinator if it resides on another logical process. This structure organizes DEVS processor into a hierarchy which does not have a one to one correspondence with the model hierarchy. Thus, a parent-child relationship that takes into account the existence of Head and Proxy coordinators must be defined as follows [7]:

1. For each simulator, the parent coordinator will be the parent's model local processor.
2. For each Proxy coordinator, the parent coordinator will be the model's Head coordinator.
3. For each Head coordinator, the parent coordinator will be the parent's model local processor; just as if it was a simulator.

Under this design, the simulation advances as a result of exchange of messages in the form of (type, time) between the parent and child DEVS processors based on the original algorithm in [11]. Two different types of messages exist: synchronization and content messages. The Collect message (@, t) is sent from a parent DEVS processor to its imminent children to tell them to send their outputs. The Internal message (*, t) is sent from a parent DEVS processor to its imminent children to tell them to invoke their transition function (either an external, internal, or confluent). The results produced by a model can be translated into Output messages (y, t) which are exchanged among a child DEVS processor and its parent. Finally, the External messages (q, t) represent the external messages arrived from outside the system or the ones generated as a result of an output message being sent to an influence.

3. Optimistic PCD++ – A Time Warp Based Parallel Simulation Engine for CD++

Originally introduced in [12], Jefferson's Time Warp protocol is the first and most well-known optimistic synchronization. A Time Warp simulation is executed via several Logical Processes (LPs) interacting with each other by exchanging time-stamped event messages. Each LP maintains a *Local Virtual Time* (LVT) that changes in discrete steps as each event is executed on the process.

A causality error arises if an LP receives an event with time stamp less than its LVT. Such events are referred to as *straggler events*. Upon the arrival of a straggler event, the process recovers from the causality error by undoing the effects of those events executed speculatively during previous computations through the rollback operation. Due to the nature of optimistic execution, erroneous computations on an LP can spread to other processes via false messages. These false messages are cancelled during rollbacks by virtue of *anti-messages*. When an LP sends a message, an anti-message is created and kept separately. The anti-message has exactly the same format and content as the positive (original) message except in one field, a negative flag. Whenever an anti-message meets its counterpart positive message, they annihilate one another immediately, hence canceling the positive one.

The Time Warp protocol consists of two distinct pieces that are sometime called the local control and global control mechanisms [1]. The local control mechanism is provided in each LP to implement the rollback operations. To do so, an LP maintains three major data structures: an *input queue* of arrived messages, an *output queue* of negative copies of sent messages, and a *state queue* of the LP's states. The global control mechanism is concerned with such global issues as space management, I/O operations, and termination detection. It requires a distributed computation involving all of the processes in the system. The central concept of the global control mechanism is the *Global Virtual Time* (GVT), which serves as a floor for the virtual time of any future rollback that might occur. Any event occurred prior to GVT cannot be rolled back and may be safely committed. Therefore, the historical events kept in the input and output queues whose time stamp is less than the GVT value can be discarded. Similarly, all but the last saved state older than GVT can be reclaimed for each process. Furthermore, I/O operations with virtual time less than GVT can be irrevocably committed with safety. Destroying information older than GVT is done via an operation known as *fossil collection*. GVT computation and fossil collection are crucial components of the global control mechanism to reclaim memory and to commit I/O operations.

Over the years, many algorithmic and data structure based optimizations have appeared in the literature to improve the efficiency of the original Time Warp protocol (see, e.g., [13-16]). The WARPED simulation kernel [17] is a configurable middleware that implements the Time Warp protocol and a variety of optimization algorithms. It relies on the Message Passing Interface (MPI) for high-performance communications on both massively parallel machines and on workstation clusters. Although the Time Warp protocol has been discussed in a great number of studies, its

applicability to simulating DEVS models is only rarely explored in the PADS literature (but see, e.g., [18-20]). The **optimistic PCD++** engine, has been developed to allow optimistic simulation of complex and large-scale DEVS and Cell-DEVS models on top of the WARPED kernel [8, 9], as shown in Figure 3.

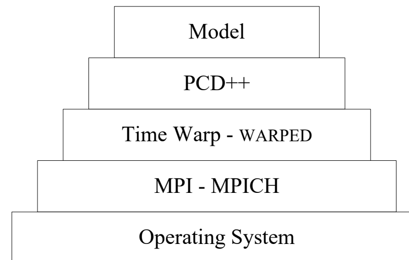

Figure 3. Layered architecture of the optimistic PCD++ simulator

3.1. Introduction to optimistic simulation in PCD++

The MPI layer and the operating system provide the communication infrastructure. The WARPED kernel offers services for creating different types of Time Warp LPs. The PCD++ simulator defines two loosely coupled frameworks: the modeling framework and the 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 LP definition in the WARPED kernel, to implement the simulation mechanisms. That is, the PCD++ processors are concrete implementations of LPs to realize the abstract DEVS simulators.

The Optimistic PCD++ employs a flattened structure consisting of four types of DEVS processors: *Simulator*, *Flat Coordinator* (*FC*), *Node Coordinator* (*NC*), and *Root* [8]. Introducing the FC and NC eliminates the need for intermediary coordinators in the DEVS processor hierarchy and, hence, minimizes communication overhead. Parallelism is achieved by partitioning the LPs onto multiple nodes. PCD++ processors exchange messages that can be classified into two categories: *content* and *control* messages. The former includes External (*x, t*) and Output message (*y, t*), which encode the data transmitted between the model components; the latter includes the Initialization (*I, t*), Collect (*@, t*), Internal (**, t*), and Done message (*D, t*), which are used to implement a high-level control flow in line with the P-DEVS formalism.

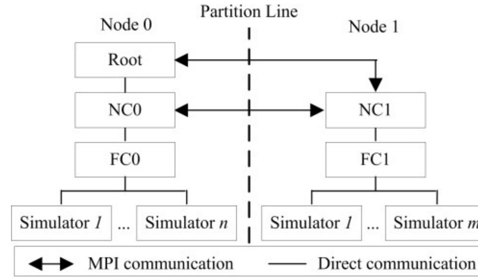

Figure 4. Optimistic PCD++ processor structure on two nodes

Figure 4 shows the PCD++ structure of the LPs involving two nodes. A single Root coordinator is created on Node0 to start the simulation and to interact with the environment. The simulation is managed by a set of NCs running asynchronously on different nodes in a decentralized manner. The FC synchronizes its child Simulators and is responsible for routing messages between the child Simulators and the parent NC using the model coupling information. Specifically, the FC keeps track of the imminent Simulators (i.e., those Simulators that have state transitions scheduled at present simulation time) and triggers state transitions at these imminent child Simulators by forwarding control messages received from the NC to them. In addition, when the FC receives a (*y, t*) from a child Simulator, it searches the model coupling information to find the ultimate destinations of the output message. 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*) to the parent NC. Otherwise, the FC translates the (*y, t*) into a (*x, t*) using the $Z_{i,j}$ translation function and directly forwards the (*x, t*) to the local receivers, which are recorded in a *synchronize set* for later state transitions. A Simulator executes the DEVS functions defined in its associated atomic model upon the arrival of control messages from the FC.

The message-processing algorithms originally proposed in [8] have been redesigned in [9] to allow for a more appropriate division of functionalities among the PCD++ processors and to address a variety of issues in parallel optimistic simulations. The major portion of the redesign effort focused on the message-processing algorithms for the NC. In the following, we summarize the main aspects of the new algorithms with an emphasis on the role of the NCs in the optimistic simulation. The NC acts as the local central controller on its hosting node and the endpoint of inter-node MPI communication. It performs a number of important operations, including:

1. Performing inter-LP communications. A structure called *NC Message Bag* is used to contain the received external messages from other remote NCs. The time of the *NC Mes-*

sage Bag is defined as the minimum time stamp 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 these external events. The current position in the *Event List* is referred by an *event-pointer* defined in the NC's state.
3. Driving the simulation on the hosting node. The NC advances the local simulation time to the minimum among: the time stamp 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*) coming from the FC.
4. Managing the flow of control messages in line with the P-DEVS formalism. The NC uses a *next-message-type* flag to keep track of the type of the control message (either @ or *) that should be sent in the next simulation cycle.

Like the FC, the NC checks the destination of each received (*y, t*) message. If the (*y, t*) is sent to the environment, the NC directly forwards it to the Root coordinator. Furthermore, the NC finds out the remote nodes on which the ultimate receiving Simulators based on the model coupling and partition information. It then translates the (*y, t*) into a (*x, t*) and sends it to the remote NC on each of those nodes. On the receiving end, the (*x, t*) will be eventually delivered to the destination Simulators located on that node. When the NC receives a (*D, t*) from the FC, the NC calculates the next simulation time (referred to as *min-time*), if the *next-message-type* is @. If the *min-time* is larger than the user-specified stop time, the NC sets a *dormant* flag and exits. Otherwise, it sends any external messages scheduled at *min-time*, to the FC. Then, it sends a control message to the FC and resets the *next-message-type* accordingly, the *next-message-type* is set to * after NC sends a (*@, t*) to the FC (in which case the output functions of the imminent Simulators will be invoked when the (*@, t*) arrives). The imminent Simulators perform internal transitions immediately after the output operations. Thus, the NC triggers the internal transitions by sending a (**, t*) to the FC in the next simulation cycle. On the other hand, if there is no imminent Simulator, the NC sends a (**, t*) whenever external messages are flushed to the FC to trigger the external transitions in the receiving Simulators.

In an optimistic simulation, some LPs may have processed all their local events while waiting for other LPs. The lagging-behind LPs may send messages to the waiting LPs and reactivate them. The *dormant* state is used by the NC where all events scheduled on the local node have been

processed. The NC exits the dormant state and reactivates the simulation on the hosting node upon the arrival of external messages from other remote NCs. In this case, the NC spontaneously flushes the received external messages with the minimum time stamp 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.

3.2. Message-passing organization

Based on the message-processing algorithms just presented, we show a sample 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-node message and a line with a stick arrowhead denotes an (asynchronous) inter-node 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.

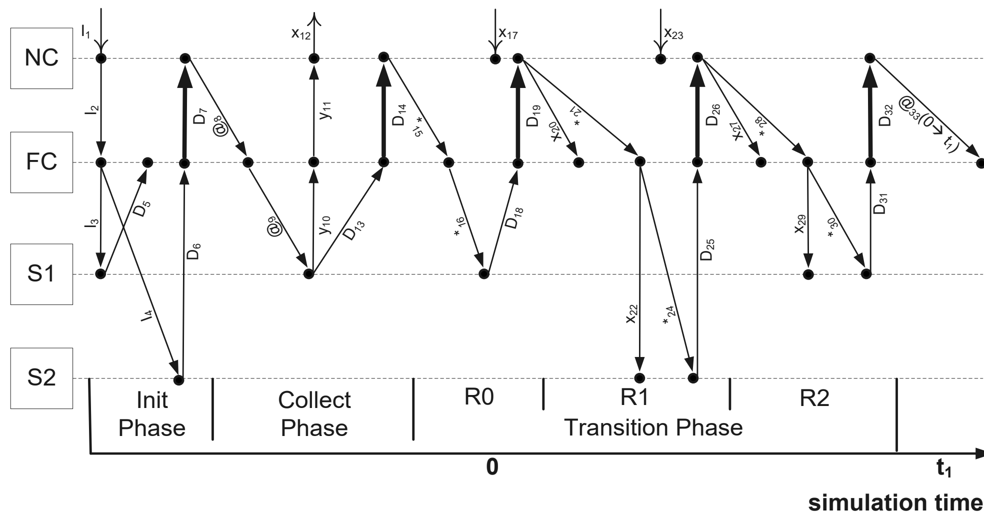

Figure 5. An example message-passing scenario on a node

Figure 5 illustrates the flow of messages on a node with four LPs: a NC, a FC, and two Simulators (S1 and S2). As we can see, the execution of messages on a node at any simulation time can be decomposed into at most three distinct phases: *initialization phase* (**I**), *collect phase* (**C**), and *transition phase* (**T**), as demarcated by the (*D, t*) message in bold black arrows received by the NC. Only one initialization phase exists at the beginning of the simulation, including the 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 range [$@_8$, D_{14}]. This phase is optional, it happens if there are im-

minent Simulators on the node 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 simulation time and may contain multiple rounds of computations (each starting with (x, t) followed by a $(*, t)$ sent from the NC to the FC and ends with a (D, t) 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. Hereinafter, we will denote a transition phase of (n+1) rounds as $\mathbf{[R_0 \ldots R_n]}$.

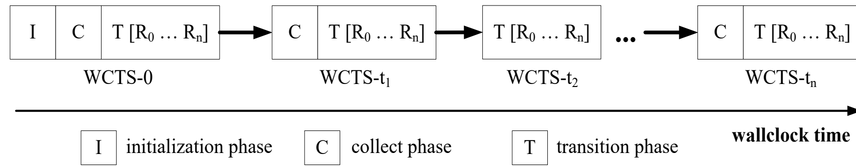

Figure 6. WCTS representation of the simulation process on a node

To have a better understanding of the simulation process on each node, a high-level abstraction, called as *wall clock time slice* (WCTS), was introduced in [9]. A WCTS at virtual time t, stands for the execution of the events scheduled at time t on all the LPs mapped on a node. Figure 6 illustrates the WCTS representation of the simulation process on a node. Several properties of the WCTS are given as follows [9].

1. The simulation on a node starts with WCTS-0, the only WCTS that has all three phases.
2. WCTS's are linked together by messages sent from NC to FC (shown as black arrows in Figure 6). At the end of each WCTS, the NC calculates the next simulation time and sends out messages that will be executed by the local LPs at this new virtual time, initiating the next WCTS on the node. The linking messages between two adjacent WCTS's have send time equal to the virtual time of the previous WCTS and receive time equal to that of the next, all other messages executed within a WCTS have the same send and receive time.
3. The completion of the simulation on a node is identified with a WCTS that sends out no linking messages (e.g. WCTS-t_n in Figure 6). The whole simulation finishes when all participating nodes have completed their portions of the simulation.
4. WCTS's are *atomic* computation units during rollback operations. That is, the events executed within a WCTS are either committed as a whole (when GVT advances beyond the

WCTS) or cancelled altogether during. In the latter case, the simulation resumes after the rollbacks from unprocessed messages sent out from the previous WCTS (with virtual time less than the current rollback time).

3.3. Cell-DEVS algorithms for optimistic execution

P-DEVS utilizes a message bag to store all simultaneous events scheduled for an atomic model so that all of them are available when the state transition at simulation time *t* is executed. This is necessary since all these simultaneous events created as (*x, t*) messages must be included in the computation to produce the correct state transition. Without explicit synchronization between the LPs, however, this requirement may not be satisfied in the optimistic simulation. Since the NC triggers state transitions speculatively based only on local information currently available on the hosting node, the state transitions at the atomic models may only involve (*x, t*) messages (those actually received by the Simulators). After the state transitions occurred at the Simulators, additional (*x, t*) messages may arrive, invalidating the results of the previous (speculative) state transitions. As shown in Figure 5, for instance, x_{23} arrives after the state transition triggered by $*_{16}$ has occurred at Simulator S1. This (*x, t*) message will be sent to the Simulator in the following round (R_2 in Figure 5) and, thus, involved in the additional state transition at S1 (x_{29} and $*_{30}$).

Since the state transitions are performed *incrementally* at the Simulators in the transition phases, the algorithms need to be adapted to this *asynchronous state transition paradigm* to obtain the same results as in a sequential simulation. A brief description of the new computation model under the asynchronous state transition paradigm is given as follows (detailed algorithms can be found in [9]).

1. Applying preemptive semantics to the state transition logic. For a transition phase with (n+1) rounds of computations, the state transitions in all but the last round (R_n) are based on incomplete information and, hence, false transitions. Only R_n has the best chance to perform the correct transition, which is the case if the WCTS is not rolled back later on in the simulation. Since the state transition in a later round involves additional external messages, it has a better chance to perform the correct computation and, thus, generate the correct results. Therefore, the state transition logic should 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, each 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 WCTS. For time 0, the previous value and state are the cell's initial value and state defined by the modeler. Except the 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 task is performed in the initialization phase.

2. Handling user-defined state variables. User-defined state variables may be involved in the evaluation of local rules defined for the cells. With the multi-round transition phase, this computation becomes more complex. During each round, a potentially different rule is evaluated and the state variables referenced in the rule are computed. Consequently, potentially wrong values are assigned to the variables and passed to the next round. The computation errors accumulate, and finally, 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 of the previous WCTS. During the following rounds at a specific simulation time, the state variables are first restored to the recorded values. Only after this restoration operation, a new computation is performed. Therefore, the cell always uses potentially correct values as the basis for a new computation.
3. Handling external events. In CD++, the *port-in* transition function (for evaluating external events received from external models) 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 overwritten by the local transition rules in later rounds. To preserve the effect of external events throughout the multi-round transition phase, an *event-flag* is set so that no further changes can be done to the cell's value in the following rounds at this time. The influence of the external event is spread out in the cell space as expected, and afterward the cell's value is again under control of its local transition rules.

3.4. Enhancements to optimistic PCD++ and the WARPED kernel

In order to carry out optimistic simulations, several other issues need to be handled. This section

covers the essential enhancements to the PCD++ and the WARPED kernel to ensure correct and efficient execution of simulations.

3.4.1. Rollbacks at virtual time 0

During rollbacks, the state of an LP 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 an LP receives a straggler message with time stamp 0, the state restoration operation will fail (negative virtual time is not found in the state queue). In optimistic PCD++, this problem is solved by explicitly synchronizing the LPs at an appropriate stage with an MPI Barrier so that no straggler message with time stamp 0 will ever be received by any LP in the simulation. The best place to implement the MPI Barrier is after the collect phase in WCTS-0, as illustrated in Figure 7.

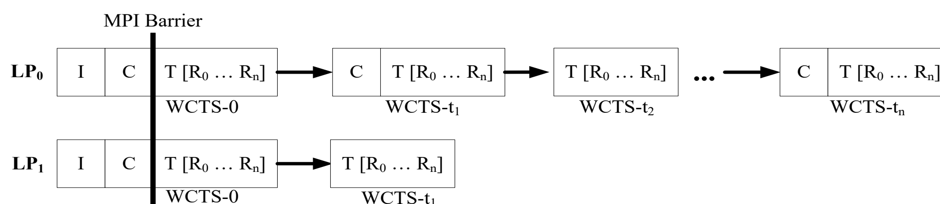

Figure 7. Using an MPI Barrier to avoid rollbacks at virtual time 0 in optimistic PCD++

The underlying assumption of this approach is that all inter-node communication happens in the collect phase. Hence, messages with time stamp 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 small when compared with the whole simulation.

3.4.2. User-controlled state-saving (UCSS) mechanism

WARPED implements the *copy state-saving* (CSS) strategy using state managers (of type *StateManager*), which save the state of an LP after executing each event. The *periodic state-saving* (PSS) strategy is realized using state managers (of type *InfreqStateManager*) that save LP's state infrequently every a number of events. Simulator developers can choose to use either type of state managers at compile time. Once selected, all the LPs will use the same type of state managers

throughout the simulation. This rigid mechanism has two 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. (2) It eliminates the possibility that different LPs may use different types of state managers to fulfill their specific needs at runtime. To overcome these shortcomings, a two-level *user-controlled state-saving* (UCSS) mechanism was introduced in [9] to provide a more flexible and efficient mechanism.

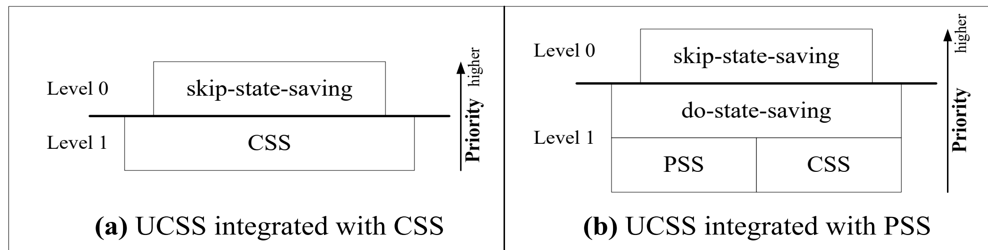

Figure 8. UCSS mechanism integrated with CSS and PSS strategies

As shown in Figure 8, a flag called *skip-state-saving* is defined in each LP. The CSS policy only takes effect when the flag is false; otherwise, no state is saved after executing the current event. The flag is reset to false so that a new state-saving decision can be made for the next event. When the PSS strategy is used, an additional flag called *do-state-saving* with a lower priority is defined, if this flag is set to true by an LP, the state manager will save the state after every event (just like CSS).

Therefore, simulator developers have the full power to choose the best possible combination of state-saving strategies at runtime.

3.4.3. Messaging anomalies

In PCD++, the NC calculates the next simulation time (*min-time*) based on the time of its *NC Message Bag*. However, external messages with time stamp less than the *min-time* may arrive after the calculation, invalidating the previous computation. 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 9.

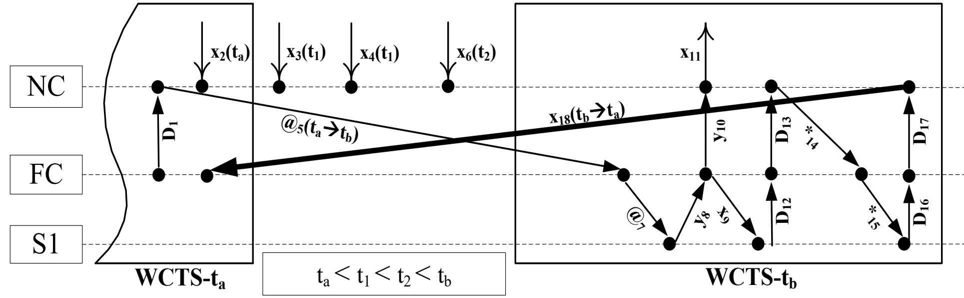

Figure 9. Example scenario of messaging anomalies

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 ($@_5$) with send time t_a and receive time t_b to the FC, initiating WCTS-t_b on the node. Meanwhile, external messages x_2, x_3, x_4, and x_6 (with time stamps 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 WARPED: First, the rollback on 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 times in the output queues, this abnormal straggler message is misplaced in the NC's output queue, resulting in causality errors later on in the simulation. Secondly, 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) *anomaly with empty NC Message Bag*: In Figure 10(a), if there are lagging external messages with time stamp t_a (e.g., $x(t_a)$) inserted into the *NC Message Bag*, the abnormal straggler message sent to the FC will have a time stamp of t_a. Hence, the LPs 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. (2) *anomaly with non-empty NC Message Bag*: In Figure 10(b), if there is no lagging external message with time stamp t_a arrived at the NC, the abnormal straggler message

will have a time stamp of t_1 ($t_1 > t_a$). Hence, the LPs 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.

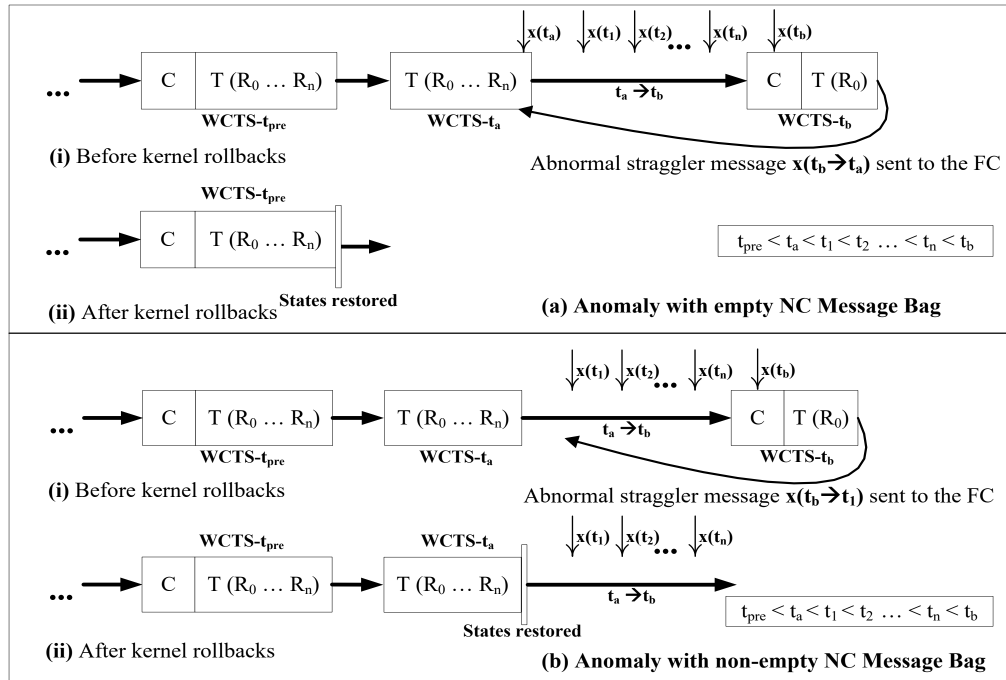

Figure 10. Two possible types of messaging anomalies in optimistic PCD++

To recover from potential messaging anomalies, the NC's message-processing performs additional cleanup operations after the completion of kernel rollbacks under abnormal situations [9]. The cleanup operations remove any remaining erroneous data generated during the anomalies from the input, output, and state queues. Only after the cleanup the control is passed to the WARPED kernel so that the simulation on the node can resume forward execution.

3.5. Optimization strategies

Two optimization strategies have been integrated with the optimistic PCD++ simulator; the *message type-based state-saving* (MTSS) which reduces the number of states saved in the simulation, and the *one log file per node* strategy to break the bottleneck caused by file I/O operations [9].

3.5.1. Message Type-based State Saving (MTSS)

During rollbacks, the state of a PCD++ processor is always restored to the *last* one saved at the end of a WCTS with virtual time strictly less than the 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 in advance, 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*). Since the Root coordinator only processes output messages, it still saves state for each event. The resultant state-saving strategy is called as *message type-based state-saving* (MTSS), a specific type of UCSS. 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 reduce the number of states saved in the simulation significantly. Consequently, the rollback overhead is reduced as fewer states need to be removed from the state queues during rollbacks. MTSS is risk-free in the sense that there is no penalty for saving fewer states, and it 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 processing (*I, t*), (*@, t*), and (*x, t*) messages. The flag is left unchanged for (**, t*) since the Simulator needs to save state for these messages.

3.5.2. One Log File per Node

The PCD++ simulator provides message logging facility for debugging, simulation monitoring and visualization purposes. However, file I/O operation is a well-known performance bottleneck in parallel simulation, especially when the files are accessed via a Network File System (NFS). This is particularly severe for Time Warp since a file queue must be maintained in the kernel for each opened file (containing uncommitted data) and all the file queues are involved in rollback operations. To remove the bottleneck the *one log file per node* strategy was implemented [9]. Only one file queue is created for the NC on each node, which is shared among all the local LPs.

The advantages of this strategy are summarized as follows.

1. The required number of file descriptors for logging purposes is upper-bounded by the number of nodes involved in the parallel simulation, rather than increasing linearly with the size of the model.
2. The simulation bootstrap time is reduced considerably due to the decrease in the number of files opened in this process.
3. The rollback operations are accelerated since only one file queue needs to be maintained in the kernel.

4. The communication overhead is reduced as well since the data concentrated in a single file queue is flushed to the file system in bigger chunks, and less frequently, over the network.

4. Hybrid Optimistic Algorithms

The Near-perfect state information (NPSI) protocols proposed by Srinivasan [21] are a new class of synchronization for parallel discrete event simulation which outperforms Time Warp, both temporally and spatially. NPSI-based protocols dynamically control the rate at which processes exploit parallelism achieving a more efficient parallel simulation. In optimistic protocols such as Time Warp, logical processes execute events aggressively assuming freedom from errors. Thus, the aggressive event execution would include risk which is the potential at which erroneous results propagate to other LPs [23]. The NPSI protocols aim at controlling both aggressiveness and risk of optimism adaptively by computing an error potential (EP). The EP of a process is defined as a function of the states of other LPs participating in the simulation. It works as an elastic force which sometimes blocks and sometimes frees the progress of the LP.

The optimism implemented by Time Warp protocol incurs three time costs: state saving, rollback, and memory management [21]. Furthermore, by restricting optimism time cost gets introduced; the lost opportunity cost is defined as the potential loss in performance when an LP is suspended while it was safe for it to continue. Thus, protocols that control optimism define the cost function as follows:

Total cost = state saving cost + rollback cost + memory management cost + lost opportunity cost.

Since the time cost of state saving can be a function of the size of the state and the frequency rate at which it is saved, the Total cost function can be rewritten by omitting the sate saving cost as:

Total cost = rollback cost + memory management cost + lost opportunity cost.

By limiting optimism the first two costs can be reduced but the lost opportunity cost would increase in return, as illustrated in Figure 11.

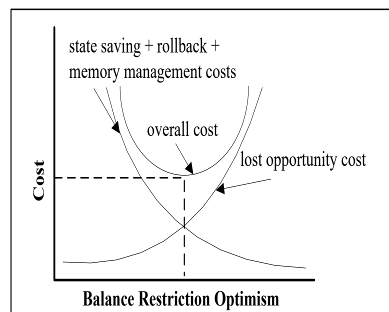

Figure 11. Tradeoff introduced by limiting optimism [21]

The best performance is attained when the controlled optimism eliminates both rollback and memory management cost and in return adds zero lost opportunity cost [21]. Optimism limiting protocols can thus achieve a good balance by precisely identifying incorrect computations and avoiding their propagation. This can be done by providing each LP with perfect state information about other LPs, but due to various latencies in computing distributed snapshots, it is impossible to obtain this information. The NPSI mechanism approximates perfect state information by using a dynamic feedback system that operates asynchronously with respect to LPs, providing them with near-perfect state information at low-cost.

NPSI controls aggressiveness and risk of LPs by dynamically computing near-perfect sate information.

In order to control the optimism of PCD++, we have modified WARPED to implement a NPSI mechanism based on the number of rollbacks. The idea is to reduce the number of rollbacks by suspending the simulation object within LP that has large number of rollbacks and therefore blocking it from flooding the net with anti-messages. However, the LP will still be able to receive input events and they will be inserted into the corresponding message bags. After a predefined duration, the suspend simulation object is released and will go on simulating. We have implemented two new protocols, namely Local Rollback Frequency Model (LRFM) and Global Rollback Frequency Model (GRFM) [22] to limit the optimism of PCD++ simulator.

The main concept is to associate each LP with an EP to control the optimism of the LP. During the simulation run, the value of each EP is kept updated by evaluating a function M1 which uses state information that is received from the feedback system. Then, a second function M2 dynamically translates every update of the EP in delays of execution events.

4.1. Local Rollback Frequency Model

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

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

Function M2: If the number of rollbacks at the interval T is greater than a specified value, then the object is suspended. The LP where the object resides on will still be able to receive incoming

events, but the events are not processed until the object is given permission to resume. However, if the number of rollbacks is less than the predefined value, the object simulates, using optimistic behavior.

To implement this protocol each LP has to be informed about two values: *max_rollback*, and *period*, where *max_rollback* is the maximum number of rollbacks allowed before, and *period* is the duration of suspension. The algorithm is presented in Figure 12.

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

Figure 12. LRFM algorithm

We can identify the following three scenarios:

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

In order for an LP to be able to simulate objects that mustn't be delayed, we have modified the scheduler policy to choose the next object to simulate. It chooses the first object of the input event list (i.e., the ones with the lowest timestamp) only if the rollbacks count does not exceed *max_rollbacks*; else, the scheduler checks the next object until it finds an object in condition to be simulated or until it reaches the end of the list.

4.2. Global Rollback Frequency Model

In the Global Rollback Frequency Model (GRFM) protocol each simulation object uses global information in such a way that among all the simulation objects residing on all LPs, the one with largest number of rollbacks must be suspended for the duration of time defined at the beginning of the simulation. Therefore, at each simulation cycle all the LPs must broadcast the information regarding the rollback counts of all of their simulation objects. As in LRFM, M1 and M2 functions must first be defined:

Function M1: The EP is the number of rollbacks that the object had minus the maximum number of rollbacks of the other simulation objects (both local and remote) participating in the simulation, from a time T1 until the current time T2, with $T2 - T1 \leq T$, where T is the interval after which the local number of rollbacks of the simulation object gets reset to zero.

Function M2: If the number of rollbacks at the interval T is greater than the number of rollbacks of the other simulation objects, then the object is suspended. The LP where the object resides on will still be able to receive incoming events, but the events are not processed until the simulation object resumes. However, if the number of rollbacks of the simulation object is less than the predefined value, then the object simulates the usual optimistic behavior.

This algorithm is presented in Figure 13. As we can see, there are three different scenarios:

1. The GRFM period has expired, therefore the simulation object starts a new period, its number of rollbacks gets reset to zero, and it is given permission to continue.
2. The GRFM period has not yet expired; if the number of rollbacks of the simulation object is less than the allowable range, the simulation object continues its normal execution.
3. The GRFM period has not yet expired, but the number of rollbacks within the simulation object has exceeded *max_rollbacks*, thus the simulation object is suspended for the entire duration of the current GRFM period.

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

Figure 13. GRFM algorithm

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

5. Lightweight Time Warp Protocol

As mentioned earlier, the operational overhead incurred in a Time Warp based optimistic simulation constitutes the primary bottleneck that may degrade system performance. Broadly speaking, Time Warp has two forms of operational overhead: *time overhead* and *space overhead*. The former consists of the CPU time required to perform the local and global mechanisms, while the latter is the result of historical data stored by each LP in its input, output, and state queues. Several technical challenges must be addressed to tackle the issues related to performance, scalability, and increased complexity of Time Warp based large-scale parallel simulation systems, including the following:

1. With a large number of LPs loaded on each available node, memory resources can quickly

be exhausted due to the excessive amount of space required for saving past events and states. Consequently, the simulator is forced to reclaim the storage space of historical data with frequent GVT computation and fossil collection, an operation that itself is an important contributor to the overall operational overhead. Advanced algorithms such as *pruneback* [24], *cancelback* [25], and *artificial rollback* [26] attempt to recover from a memory stall at the expense of additional computation and communication overhead. It is desired to have a protocol that can support large-scale optimistic simulation even when memory resources are tight, while at the same time reducing the overhead of GVT computation and fossil collection to the minimum (doing it only when absolutely necessary).

2. One potential performance hazard in large-scale optimistic simulation is that the cost of rollbacks increases dramatically simply because a massive number of LPs are involved in the rollback operation on each node. Prolonged rollbacks not only result in poor system performance, but also increase the probability of *rollback echoes* [1], an unstable situation where simulation time does not progress. Therefore, it is imperative to fashion a new approach that can dramatically reduce the rollback cost without introducing if additional runtime overhead.
3. Different implementations of the event sets in Time Warp have been the focus of research for several years (see, e.g., [27-30]). A primary motivation behind these efforts is to improve the efficiency of queue operations as the number of stored events increases in large-scale and fine-grained simulations. In addition to using advanced data structures, the simulation performance would also be improved if we could keep the event queue relatively short throughout the simulation, an alternative approach that warrants close examination.
4. Dynamic load balancing has been recognized as a critical factor in achieving optimal performance in both general-purpose parallel applications and large-scale PADS systems where the workload on different nodes and the communication patterns between them are in constant fluctuation (see, e.g., [31-34]). Algorithms for dynamic load balancing usually rely on metrics whose values are valid only for a short period. This problem is especially severe in optimistic simulations since a potentially unbounded number of uncommitted events and states associated with an LP need to be transferred to a different node before the invalidation of the metric values. Only a few studies address specifically the issue of fa-

cilitating load migration in Time Warp systems. For example, Reiher and Jefferson proposed a mechanism to split an LP into phases to reduce the amount of data that must be migrated [35]. More recently, Li and Tropper devised a method that allows for reconstructing events from the differences between adjacent states so that only the state queue needs to be transferred between processors [36]. However, this approach works only for systems with fine event granularity and small state size such as VLSI circuits. An agile load migration scheme is needed to reduce the communication overhead in Time Warp based large-scale parallel simulation systems.

5. The way how simultaneous events are handled has serious implication on both simulation correctness and performance. To ensure correct simulation results, P-DEVS introduces (partial) causal dependency among simultaneous events, requiring a control flow to enforce an orderly event execution at each virtual time. From a performance perspective, however, this expanded execution of simultaneous events could increase the overhead for state-saving, rollback, and fossil collection, an issue that has not yet been addressed by existing TW-based DEVS systems.

To address these problems in a systematic way, a novel protocol, known as Lightweight Time Warp (LTW), has been proposed for high-performance optimistic simulation of large-scale DEVS and Cell-DEVS models [37]. LTW is able to set free the majority of the LPs from the heavy machinery of the Time Warp mechanism, while the overall simulation still runs optimistically, driven by only a few fully-fledged Time Warp LPs. Although the following discussion is centered on realizing the LTW protocol in optimistic PCD++, the basic concepts also apply to a broad range of PADS systems under certain conditions and with appropriate control over the LPs.

The roles of the various LPs under LTW are as follows [37].

1. The NC is the only LP that executes based on the standard Time Warp mechanism.
2. The FC, becomes a mixed-mode LP that serves as an interface between full-fledged and lightweight LPs.
3. The Simulators are turned into lightweight LPs, free from the burdens of maintaining historical data in their input, output, and state queues.

5.1. A brief review of the simulation process in optimistic PCD++

As introduced in Section 3, on each node, the optimistic PCD++ simulator creates only one NC and FC, whereas many Simulators coexist in a typical large-scale simulation. Hence, a substantial

reduction in the operational overhead at the Simulators would result in a significant improvement in the overall system performance. Note that even though the LPs are grouped together on each node, their LVT values may differ. Thus, the key characteristics of the simulation process can be summarized as follows.

1. Simulators only communicate with their parent FC (i.e. no direct communication between the Simulators). Thus, their states can change only as the result of executing events coming from the FC. The FC has full knowledge of the timing of state changes at its child Simulators.
2. Rollbacks on a node begin either at the NC (as a result of straggler or anti-messages arrived from other remote NCs), or at the FC (in the case of messaging anomalies). In both cases, rollbacks propagate from the FC to its child Simulators on a node. Thus, the FC knows when the rollbacks will occur at the Simulators. Besides, a WCTS is an atomic computation unit for the FC and Simulators during rollbacks.
3. The advance of the simulation time on each node is controlled entirely by the NC. The FC and the Simulators do not advance their LVTs voluntarily, nor do they send messages across WCTS boundaries.

Based on these assumptions, we now discuss LTW protocol main parts: a rule-based dual-queue event scheduling mechanism, an aggregated state-saving scheme, and a lightweight rollback mechanism (while the detailed algorithms can be found in [37]).

5.2. Rule-based dual-queue event scheduling mechanism

Under Time Warp, the input queue is persistent, in the sense that the events remain in the queue until being fossil collected when the GVT advances beyond their time stamps. However, keeping this not only consumes lots of memory, but also increases the cost of queue operations. LTW solves this problem by introducing an additional volatile input queue that is used to hold temporarily the simultaneous events exchanged between the FC and its child Simulators within each phase of a WCTS. On the contrary, the persistent input queue is used only by the NC and FC to contain the events sent between them. A key observation is that, during a rollback, the incorrect events previously exchanged between the FC and its child Simulators are essentially annihilated with each other (due to the atomicity of the WCTS). Therefore, it is safe to exclude these events from the persistent queue. An example message flow between the LPs mapped on a node under this dual-queue arrangement is illustrated in Figure 14.

From the Time Warp perspective, the simulation process only involves a small fraction of the total events executed by the LPs, as shown in Figure 15. Comparing both figures, we can see that the events executed by the FC and the Simulators within each phase of a WCTS can be considered as being collapsed into a single aggregated event. Note that the linking messages between adjacent WCTS's (e.g., x_{23} and $*_{24}$) are still kept in the persistent queue, allowing the simulation to resume forward execution after rollbacks.

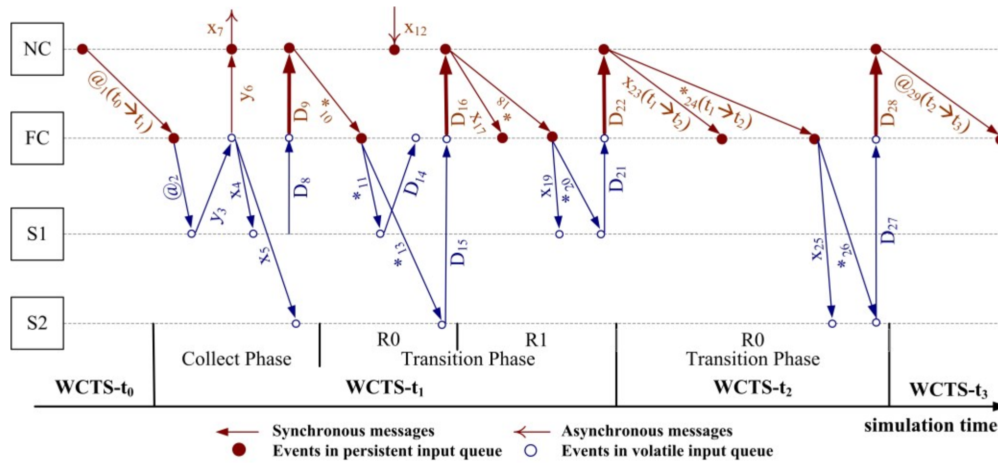

Figure 14. Message flow between the LPs using both persistent and volatile event queues

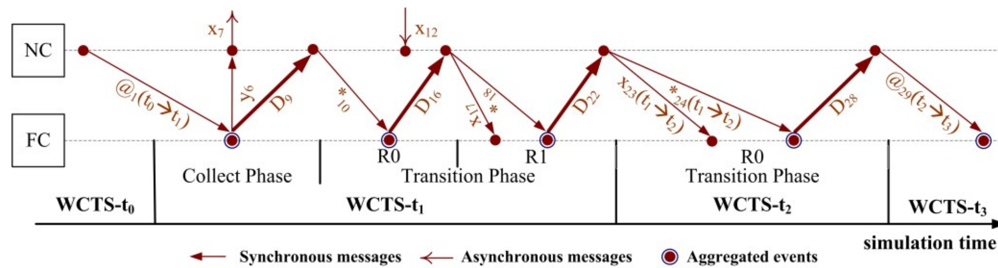

Figure 15. Message flow between the LPs from the Time Warp perspective

The volatile input queue has two appealing properties that allow us to reduce the memory footprint and the cost of queue operations significantly:

1. Events in the queue are discarded and their memory reclaimed immediately after execution, (reducing the memory footprint of the system).
2. Events in the queue always have the same time stamp. They are inserted into the queue as the simulation moves into each phase of a WCTS, and removed as the execution proceeds. At the end of each phase, the queue becomes empty. This means that a simple FIFO queue suffices, and queue operations can be performed efficiently in O(1) time.

Consequently, the persistent queue becomes shorter, allowing for more efficient operations as

well. For those events in the volatile queue, their counterpart anti-messages are no longer saved in the output queues of the sending LPs, further reducing memory consumption and speeding up the forward execution of the simulation. Similarly, message annihilations are not required to cancel these events during rollbacks, minimizing the rollback overhead and enhancing the stability and performance of the system. In addition, this also facilitates fossil collection due to the significant reduction in the number of past events and anti-messages stored in the persistent input and output queues, which, in turn, allows for more frequent GVT computation and fossil collection.

A rule-based event scheduling scheme has been proposed to schedule the execution of events from both queues [37]. This scheme is implemented on each node by a scheduler that maintains two pointers, called as *p-ptr* and *v-ptr*, to reference the next available events in the queues respectively. The persistent queue contains events sorted in Lowest Time Stamp First (LTSF) order, including unprocessed events and those processed but not yet been fossil collected. On the other hand, the volatile queue only holds simultaneous events not yet processed in the current phase of a WCTS, as illustrated in Figure 16.

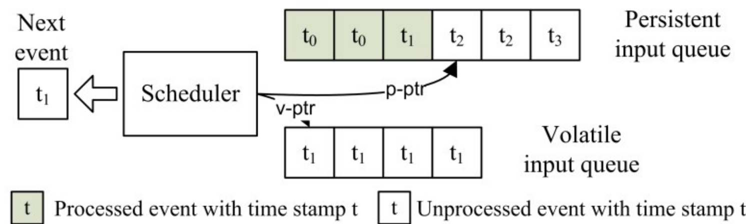

Figure 16. Rule-based Dual-queue event scheduling

The *p-ptr* pointer may need to be updated when the persistent queue is modified (event insertion and/or annihilation) to ensure that it always points to the first unprocessed event with the minimum time stamp, whereas the *v-ptr* pointer simply points to the event at the front of the volatile queue. At each event selection point, the scheduler compares the two events based on a set of predefined rules and chooses one of them as the next event to be executed in the current cycle, allowing the scheduler to adjust the priorities of the input queues dynamically on an event-by-event basis. The scheduling rules are executed in the order of appearance in the list.

1. **Idle condition.** The simulation becomes idle if the volatile queue is empty and the persistent queue does not contain events with time stamps before or at the simulation stop time. The simulation may be reactivated later upon the arrival of messages from other nodes.
2. **Simulation progress**. If the volatile queue becomes empty the scheduler selects the next

persistent event with a time stamp earlier than the simulation stop time. The NC can 1) advance simulation time on the node, or 2) resume forward execution from unprocessed persistent events after a rollback, or 3) reactivate the simulation from the idle state upon the arrival of remote messages, which are inserted into the persistent queue.

3. **Aggressive inter-node communication.** During a collect phase, the NC may send messages to remote nodes. As these are potentially straggler messages at the receiving end, a delay in their delivery could postpone rollbacks at the destination, degrades performance. Thus, the scheduler grants higher priority to the persistent events events with the same time stamp in order to process inter-node messages immediately.
4. **LTSF execution**. In all other cases, the next volatile event is selected to execute, enforcing a Least-Time-Stamp-First execution on the node.

Note that an event selected from the volatile queue is removed (it will be deleted by the receiving LP after execution), whereas an event chosen from the persistent queue is simply marked as processed and the *p-ptr* is moved to the next available event.

5.3. Aggregated state management and an optimal risk-free state-saving strategy

In a Time Warp system, each LP maintains its own state queue in order to restore to state variables during rollbacks. This approach allows for wide generality and straightforward implementation. However, it has several disadvantages. The historical states are scattered among the individual LPs, prohibiting efficient batch operations from being applied to the state queues. Also, state restorations are triggered entirely by straggler and/or anti-messages, stressing on the underlying communication infrastructure. LTW allows the Simulators to delegate the responsibility of state management to the FC. As a result, the Simulators are turned into truly lightweight LPs, totally isolated from the complex data structures required by Time Warp.

In this new state management scheme, the FC employs an aggregated state manager that maintains the state queue for the FC itself, but also those used by the child Simulators. As illustrated in Figure 17, the state queue for a Simulator uses a *dirty bit* to identify the Simulators whose states have been modified during the computation of the current WCTS. The state-saving operation is carried out only when the FC detects that the events previously sent to the Simulators have been processed, and is performed only for those Simulators with dirty bits set to true. Note that no dirty bit is associated with the state queue for the FC itself since the FC is always involved in the computation of each WCTS.

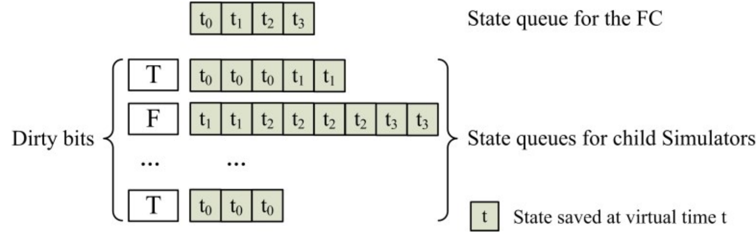

Figure 17. Aggregated state manager for the FC

With this aggregated state management scheme, we now introduce an optimal risk-free state-saving strategy. As discussed in Section 3, the optimistic PCD++ simulator utilizes a Message Type-based State-Saving (MTSS) strategy that allows to save states only after executing certain types of events. Specifically, the NC and FC save states only after processing (D, t) messages, while the Simulators save state only after executing (*, t) messages. Using the MTSS strategy, a transition phase may have multiple rounds of computation, thus, an LP could still save many states in each WCTS. During rollbacks, an optimal state-saving strategy should save only a single state at the end of each WCTS. The following strategy satisfies this condition and it is risk-free (no penalty is incurred as the result of saving fewer states).

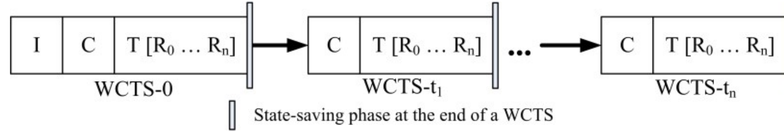

Figure 18. Introducing a state-saving phase at the end of each WCTS

As shown in Figure 18, a state-saving phase is added to the end of each WCTS. Before advancing the simulation time, the NC instructs the FC to save states for the current WCTS. At this moment, all events belonging to the current WCTS have been processed by the FC and the Simulators and, thus, the saved states will reflect the latest values of the state variables in the LPs. Only when the state-saving phase completes, can the NC send linking messages to the FC to initiate the next WCTS for the new simulation time. The state of the NC itself is saved after processing (D, t) messages, just like in the MTSS strategy.

5.4. Lightweight rollback mechanism

As the Simulators are turned into lightweight LPs, they are neither expected nor allowed to carry out rollbacks on their own in LTW. Instead, the Simulators must rely on the FC to recover from causality errors. As rollbacks always propagate from the FC to the Simulators on each node, the FC has the full knowledge of when to perform rollbacks for its child Simulators. One of the most elegant features of LTW is that the incorrect input events previously executed by the Simulators

have already been deleted in the volatile queue during forward execution. As a result, a great amount of CPU time that would otherwise be wasted on message annihilation can be saved, leading to an accelerated rollback and significant improvement in performance overall.

The rollback of the FC itself is still triggered by straggler and/or anti-messages from the NC. The lightweight rollback mechanism must restore properly the state at the Simulators. One difficulty is that Simulators execute asynchronously and may not have the same LVT. Only the states of the Simulators involved in incorrect computations need to be restored during rollbacks. To solve this problem, the FC uses an array of *latest state change time* (LCT) to keep track of the time when the Simulators modify their states. The LCT is updated whenever the FC sends a (*, t) to a Simulator. If a rollback occurs, the FC cancels speculative interactions with the NC (based on the standard Time Warp mechanism). It then invokes the scheduler to delete all volatile events with a time stamp greater than or equal to the rollback time. Finally, the FC instructs the aggregated state manager to recover the state for each Simulator whose LCT is greater than or equal to the rollback time. After the state restoration, the LCT is updated to the LVT of the recovered state.

In this way, LTW can perform rollbacks much more efficiently than the standard Time Warp mechanism due to, the reduction of message annihilations in the persistent input queue. Moreover, all the Simulators can be rolled back without sending any anti-messages, reducing overhead.

5.5. LTW implications

Though largely a local control protocol, LTW also has an impact on several aspects of the TW global control mechanism. First, fossil collection on each node is accelerated, not only because the fossil data in the persistent queues are minimized, but also because most of the states are managed in a centralized manner, allowing for efficient batch operations. Secondly, agile process migration is possible since only the state queues need to be transferred to move lightweight LPs for dynamic load balancing. The appropriate decision points for process migration would be at the end of each state-saving phase when all the volatile events have been executed (and deleted) and the states of the LPs have been saved.

Additionally, LTW can be seamlessly integrated with other TW optimizations to further improve performance. For instance, various state-saving and cancellation strategies can be applied to the TW domain directly. LTW can be considered as complementary to Local Time Warp [38]: the former is a purely optimistic approach to reducing operational cost within each local simulation space, while the latter is a locally optimistic approach to mitigating cascaded rollbacks in the

global space. It is easy to combine both approaches in a consistent way.
Though only a single LTW domain is considered here, the protocol can be readily extended to support hybrid systems that require multiple LTW domains coexisting on each node to implement domain-specific formalisms. Besides, the basic concepts derived from the LTW protocol could also apply to a wide range of TW systems through carefully choosing the level of event granularity and imposing an appropriate control over the LPs.

6. Performance Analysis of LTW

We studied the performance of PCD++ quantitatively. Our experiments were carried out on a cluster of HP PROLIANT DL servers that consists of 32 compute nodes (dual 3.2GHz Intel Xeon processors, 1GB PC2100 266MHz DDR RAM) running Linux WS 2.4.21 interconnected through Gigabit Ethernet and communicating over MPICH 1.2.6.
The performance of optimistic PCD++ was tested with Cell-DEVS models with different characteristics, including a model for forest fire propagation [39] and a 3-D watershed model representing a hydrology system [40],[43]. The fire propagation model computes the ratio of spread and intensity of fire in forest based on specific environmental and vegetation conditions. The watershed model represents the water flow and accumulation depending on the characteristics vegetation, surface waters, soil, ground water, and bedrock. The watershed model was coded as a 3-D Cell-DEVS model to simulate the accumulation of water under the presence of constant rain. Conservative PCD++ was analyzed by running the above mentioned fire model as well as another CELL-DEVS model on Synapsin-Vesicle Reaction [41]. We have modeled the reserve pool of synaptic vesicles in a presynaptic nerve terminal, predicting the number of synaptic vesicles released from the reserve pool as a function of time under the influence of action potentials at differing frequencies. Time series amounts for the components are obtained using rule-based methods [42]. Creating this model in PCD++ allows spatial description of synapsin-vesicle interactions and PCD++ makes it possible to have graphical representations which are comparable to the real scene observed in electron microscopes.
Performance metrics were collected during the simulations to gauge the performance and profile the simulation system. These metrics fall into two categories based on their intended purposes: namely performance measurement and system profiling. The former group includes three values collected to measure execution time, memory consumption, and CPU utilization. The latter group consists of a variety of values at runtime to profile the simulation system, including the number of

states saved during the simulation, the time spent on state-saving operations, and the bootstrap time for simulation initialization. We use two different speedups measuring the overall speedup reflects how much faster the simulation runs on multiple machines than it does on a single one (as perceived by the users), whereas the algorithm speedup is calculated from the actual running time (i.e., without considering the simulation bootstrap time, to assess the performance of the parallel algorithms alone).

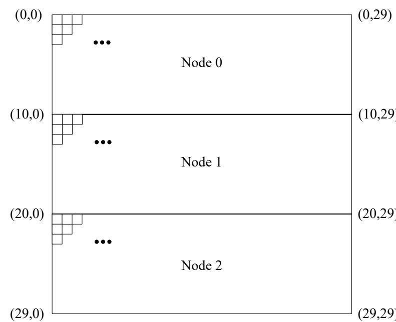

Figure 19. A simple partition strategy for Cell-DEVS models

A simple partition strategy was adopted for all the models in the following tests. It evenly divides the cell space into horizontal rectangles, as illustrated in Figure 19 for a 30×30 model partitioned over 3 nodes. Using different partition strategies could have a big impact on the performance of the simulation. Since the workload on the nodes is unpredictable and keeps changing during the simulation, it is hard, if not impossible, to predict the best partition strategy for a given model before the simulation. This problem can be alleviated by using some dynamic load-balancing techniques in the simulation algorithms.

6.1. Performance analysis of the conservative PCD++

The *Synapsin-Vesicle Reaction* model consists of 676 cells arranged in a 26x26 mesh with a total execution time of 3.7621 seconds when run on the standalone single-processor CD++. Figure 20 represents the execution time resulting from running the model with the conservative parallel simulator on 1 to 8 nodes.

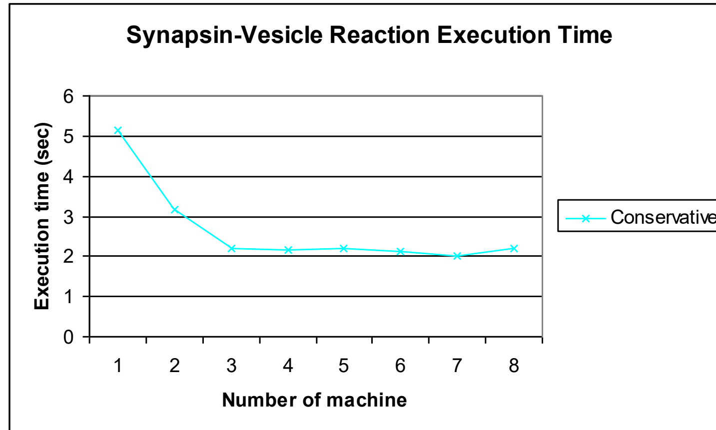

Figure 20. Synapsin-vesicle model execution time on conservative parallel simulator

As seen on the graph, the conservative simulator improves the total execution time significantly when more than 2 nodes are available. As the number of participating nodes increases, the speed up factor decreases. The main reason is communication overhead among the participating LPs which leads in a noticeable time added to the duration of the model execution.

The *Fire Propagation* model consists of 900 cells arranged in a 30x30 mesh with a total execution time of 6.2145 seconds when run on standalone CD++. Figure 21 represents the execution time resulting from running the model with the conservative simulator on 1 to 8 nodes.

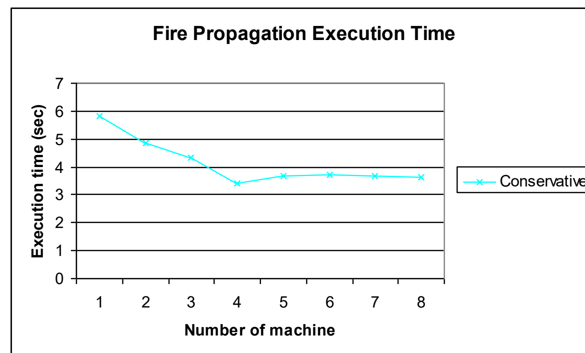

Figure 21. Fire propagation model execution time with the conservative parallel simulator

As seen on the graph, the parallel simulator improved the execution time of the model. The conservative simulator presented the best results on 4 nodes, while still all other scenarios with more than one node have lower execution results with respect to simulation on single node.

6.2. Performance analysis of optimistic PCD++

In this section, we evaluate the performance of optimistic PCD++ under the standard Time Warp protocol and the PCD++ optimization strategies discussed in Section 3.

6.2.1. Effect of the one log file per node strategy

The fire propagation model (900 cells arranged in a 30×30 mesh) was executed on 1 and 4 nodes with and without using the strategy during a period of 5 hours. The resulting execution time (T) and bootstrap time (BT) are illustrated in Figure 22, where the BT for 4 nodes is the arithmetic average of the BT values collected on these nodes.

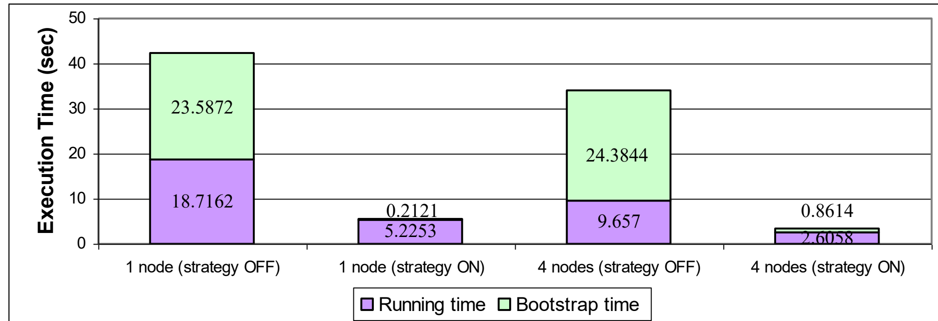

Figure 22. Execution and bootstrap time before and after one log file per node strategy on 1 and 4 nodes

As we can see BT is greater than the actual running time (when the strategy is turned off), clearly indicating that the bootstrap operation is a bottleneck. 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.

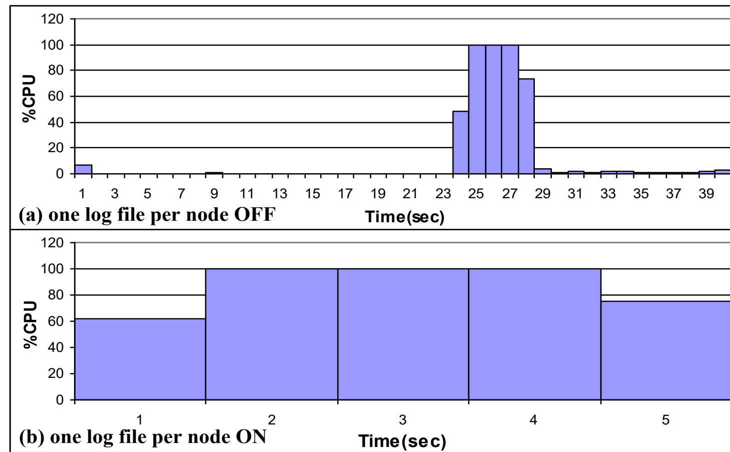

Figure 23. CPU usage before and after one log file per node strategy on 1 node

The CPU usage (%CPU) monitored in our experiments also suggests that the file I/O operation is a major barrier in the bootstrap phase. The CPU usage collected before and after applying the one log file per node strategy on a single node is shown in Figure 23. Again, when the strategy is turned off, the CPU essentially remains idle in the first 23 seconds (corresponding to the observed BT),

during which a majority of time has been dedicated to I/O operations for creating the log files at the NFS server over the network. At the end of the simulation, the logged data is flushed to the physical files, resulting in intensive I/O operations again. As expected, the CPU rests idle in the last 12 or so seconds. On the other hand, the computation is condensed when the strategy is applied to the simulator. As a result, 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.
Other observations are described below:

1. The bootstrap time tends to increase when more nodes are used to do the simulation. For example, the BT increased from 0.2121 seconds on 1 node to 0.8614 seconds on 4 nodes in our experiments. The reason is that the number of log files increases with the number of nodes, causing higher delays in communication and file I/O operations at the NFS server.
2. The bootstrap time also tends to increase somewhat along with the size of the model because of the additional operations for memory allocation and object initialization in the main memory. However, this is a relatively moderate increase when compared with the previous case.
3. Even though the bootstrap time is reduced significantly with the one log file per node strategy, it still constitutes an overhead that cannot be ignored when we measure the real effect of the parallel algorithms. In the experiments, it accounts for 3.9% and even 24.84% of the total execution time on 1 and 4 nodes respectively.

6.2.2. Effect of the message type-based state-saving strategy

The fire propagation model was used to evaluate the performance improvement derived from the MTSS strategy. Besides the standard Time Warp algorithms, the one log file per node strategy is also applied to the simulator in the following experiments.

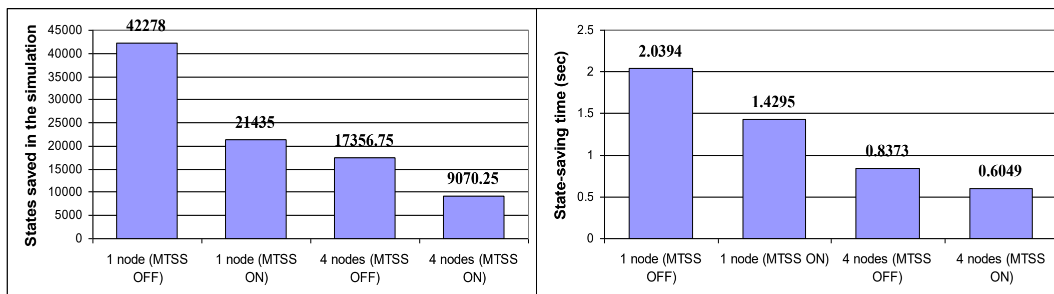

Figure 24. States saved and state-saving time before and after MTSS strategy on 1 and 4 nodes

The model was executed on 1 and 4 nodes (respectively) with and without the MTSS strategy

(respectively). The number of states saved in the simulation (SS) and the time spent on state-saving operations (ST) are shown in Figure 24. Here, the data for 4 nodes is the average of the corresponding values collected on the nodes. Owing 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%.

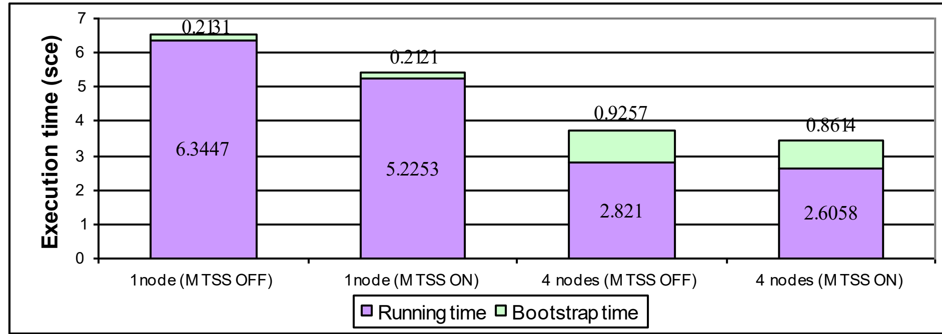

Figure 25. Running and bootstrap time before and after MTSS strategy on 1 and 4 nodes

The corresponding running time and bootstrap times are shown in Figure 25. While the bootstrap time remains nearly unchanged, the actual running time is reduced by 17.64% and 7.63% on 1 and 4 nodes respectively because fewer states are saved and potentially removed from the queues.

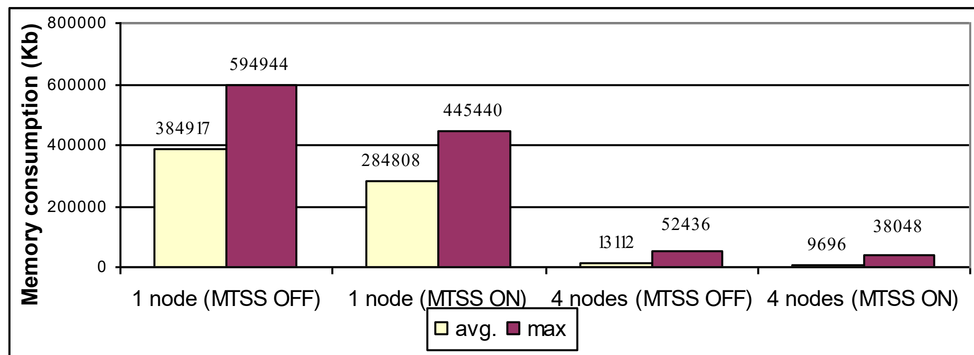

Figure 26. Average and maximum memory consumption before and after MTSS strategy

The most noticeable effect of the MTSS strategy is the decrease in memory consumption. Figure 26 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 time-weighted average was calculated using an interval of 1 second. For 4 nodes, the data was also averaged over the nodes. The average memory consumption declines by 26% in both cases, while the peak memory consumption decreases by 25.13% and 27.44% on 1 and 4 nodes respectively.

6.2.3. Performance of the optimistic PCD++

The metrics used for evaluating the performance of the optimistic PCD++ simulator included the execution time and speedup. We analyze the execution results of the Cell-DEVS models with the standard Time Warp algorithms. In addition, the one log file per node and MTSS strategies were applied to the simulator in the experiments.

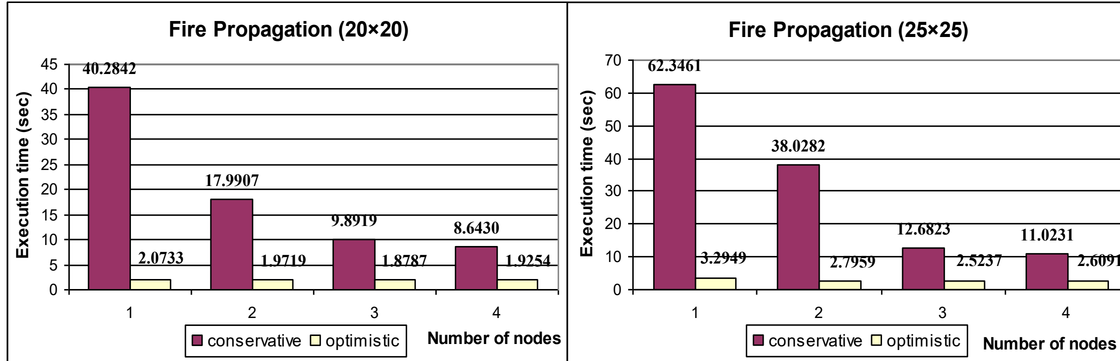

Figure 27. A comparison of the conservative and optimistic PCD++ using the fire model

We compare the performance of conservative and optimistic PCD++ using the fire propagation model executed on a set of compute nodes. Figure 27 shows the total execution time for the fire propagation simulation with different sizes, including 20×20 (400 cells), 25×25 (625 cells), 30×30 (900 cells) and 35×35 (1225 cells).

In all cases, the optimistic simulator markedly outperforms the conservative one. There are three major contributing factors:

1. Optimistic PCD++ has been optimized with the one log file per node strategy. Hence, its bootstrap time is substantially less than that of the conservative. Although the data logged during the simulations is the same for both simulators, the number of log files generated by PCD++ is only a small fraction of that created by the conservative simulator. This factor accounts for much of the difference in the execution time on a single node.
2. Time Warp avoids, for the most part, the serialization of execution that is inherent in the conservative algorithms, and hence exploit higher degree of concurrency.
3. The non-hierarchical approach adopted in optimistic PCD++ outperforms the hierarchical one of the conservative simulator. The flattened structure reduces the communication overhead and allows more efficient message exchanges between the PCD++ processors.

The total execution time and running time of the fire model executed on 1 up to 4 nodes are listed in Table 1.

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

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 to do the simulation. For example, as the model size increases from 400 to 1225 cells, the execution time ascends sharply by nearly 280% (from 2.0733 to 7.8702 seconds) on 1 node, whereas it merely rises by 98% (from 1.9254 to 3.8138 seconds) on 4 nodes. On the other hand, for a fixed model 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 seconds 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 seconds as the number of nodes rises from 3 to 4. When a model, especially a small one, is partitioned onto more nodes, the increasing overhead involved in inter-LP communication and potential rollbacks may eventually degrade performance. Hence, a trade-off between the benefits of 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.
Based on the collected execution and running time, the overall and algorithm speedups are shown in Figure 28.

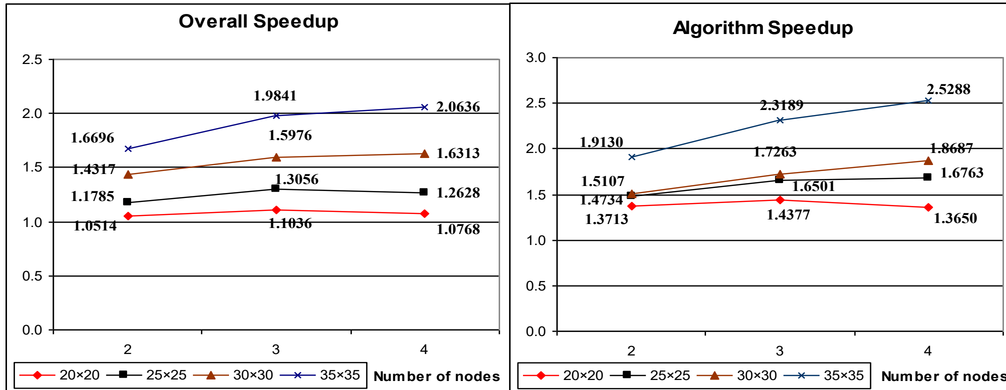

Figure 28. Overall and algorithm speedups for fire model of various sizes on a set of nodes

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, an evidence showing that the Time Warp optimistic algorithms are major contributors to the overall performance improvement.

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

A more computation-intensive 3-D watershed model of size 15×15×2 (450 cells) was tested to evaluate the performance of optimistic PCD++ for simulating complex physical system. Table 2 shows the 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 seconds respectively.

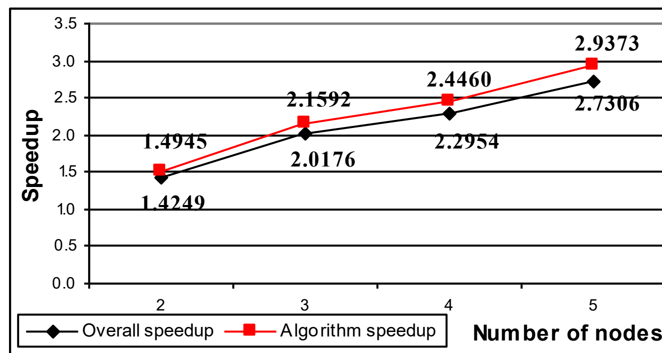

Figure 29. Overall and algorithm speedups for the 15×15×2 watershed model

The resulting speedup is illustrated in Figure 29. The best overall and algorithm speedups achieved for the 15×15×2 watershed model are 2.7306 and 2.9373 respectively, higher than those obtained with the fire model.

6.3. Effect of hybrid optimistic algorithms

To analyze the impact of the LRFM and GRFM Time Warp-based protocols on the performance of the optimistic PCD++ simulator, the same models were also run using LRFM- and GRFM-based optimistic simulators. Different models with distinguishable functionality, complexity, and size were selected for the experiments to better judge the capability of the simulators. The *Synapsin-Vesicle Reaction* model presented in Section 6.1 is presented in Figure 30 which represents the execution time resulting from running the model with four different simulators on 1 to 8 nodes.

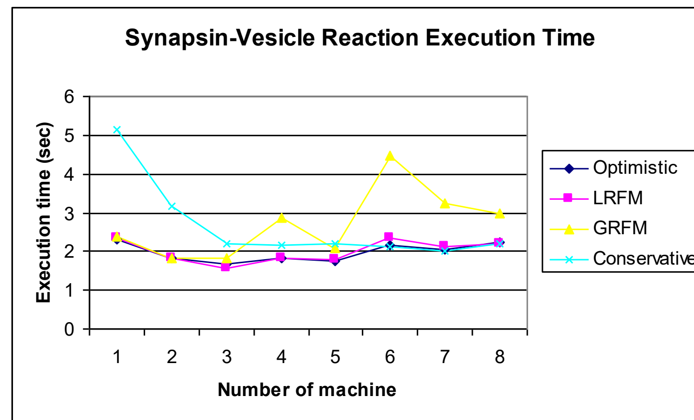

Figure 30. Synapsin-vesicle model execution time on 4 different simulators

We can see that the optimistic and LRFM-based simulators produce very close results on 1 to 8 nodes. Also, the GRFM-based simulator has similar results for 1, 2, 3, and 5 nodes. However, the performance is degraded when 4, 6, 7, and 8 nodes are participating due to the increase of number of remote messages that are sent back and forth. On the other hand, the conservative simulator shows different behavior as the number of nodes increases. As the number of computing nodes increases, the GRFM-based simulator has the lowest performance among other ones. The main reason is communication overhead among the participating LPs which leads in a noticeable time added to the duration of the model execution.

Figure 31 represents the speedups of the model execution times with respect to execution on one node for each particular simulator.

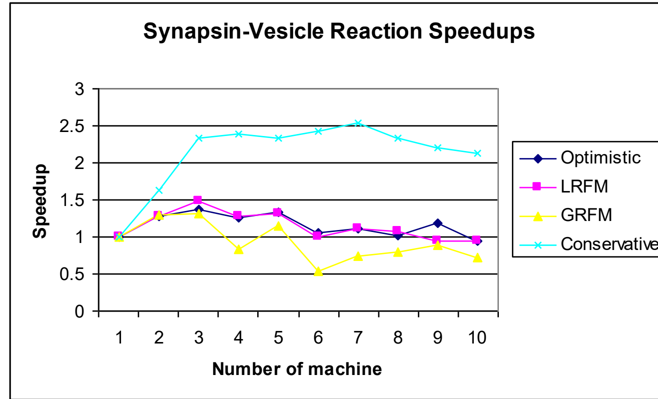

Figure 31. Synapsin-vesicle model speedups with regards to execution on one node

6.3.2. Fire Propagation Model

This model uses 900 cells arranged in a 30x30 mesh with a total execution time of 6.2145 seconds when run on standalone CD++. Figure 32 represents the execution time resulting from running the model with four different simulators on 1 to 8 nodes.

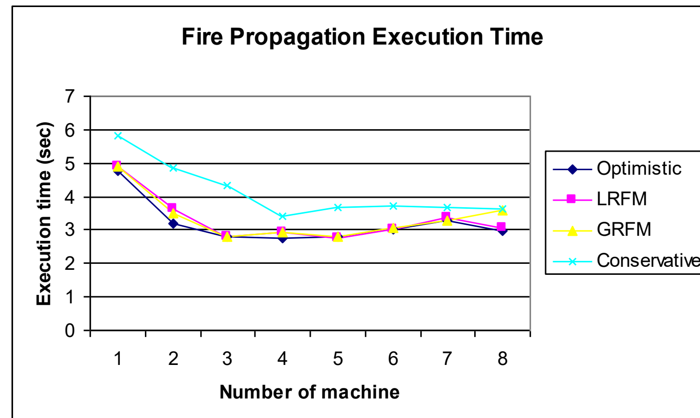

Figure 32. Fire propagation model execution time on 4 different simulators

As seen on the graph, the parallel simulator significantly improved the execution time of the simulation. The three optimistic simulators produced very similar results on 1 to 7 nodes and the optimistic simulators outperforms the conservative one. For the optimistic simulators the best results were achieved on 5 nodes, while the conservative one had its lowest execution time on 4 nodes since the number of remote messages and the suspension duration of the simulation objects were at their best on 4 nodes.

6.4. Comparing Time Warp and Lightweight Time Warp

As analyzed in Section 5, LTW can effectively improve system performance in various ways

(including reduced memory consumption, lowered operational overhead, and accelerated queue operations). In this section, we present a comparison of the optimistic PCD++ performance under the standard Time Warp and Lightweight Time Warp protocols.

6.4.1. Experiment platform and metrics

Both TW and LTW protocols have been implemented in optimistic PCD++. A stress test was carried out on a cluster of 28 HP Proliant DL140 servers (dual 3.2GHz Intel Xeon processors, 1GB 266MHz RAM with 2GB disk swap space) running on Linux WS 2.4.21 and communicating over Gigabit Ethernet using MPICH 1.2.7. Note that severe memory swapping may occur if memory usage approaches the upper limit of 1GB on a node. Table 3 lists the metrics collected in the experiments through extensive instrumentation and measurement. The experimental results for each test case were averaged over 10 independent runs to strike a balance between data reliability and testing effort. For those cases executed on multiple nodes, the results were averaged over the participating nodes to obtain a per-node evaluation. The queue lengths (i.e., PQLen and VQLen) were averaged over samples collected every 20 event insertions.

Table 3. Performance metrics

Metrics	Description
T	Total execution time of the simulation (sec)
MEM	Maximum memory consumption (MB)
PEE	Number of events executed in persistent queue
VEE	Number of events executed in volatile queue
PQLen	Average length of the persistent input queue
VQLen	Average length of the volatile input queue
SS	Total number of states saved
OPT-SK	Number of states reduced by the optimal strategy
FCT	Average time spent on a single fossil collection (ms)
PriRB	Number of primary rollbacks
SecRB	Number of secondary rollbacks
RB	Total number of rollbacks (i.e., PriRB + SecRB)
EI	Number of events imploded in persistent queue
ER	Number of events unprocessed in persistent queue

6.4.2. Environmental models

Three Cell-DEVS models with varied characteristics were validated and tested. Two of them simulate stationary wildfire propagation over 50 hours in a 2D cell space based on the Rothermel model. However, they differ in the way the spread rates are calculated. The first fire model, referred to as *Fire1*, uses predetermined rates at reduced runtime computation cost. The second fire model, referred to as *Fire2*, invokes the `fireLib` library [44] to calculate spread rates dynamically, with higher runtime computation density, based on a set of parameters such as fuel type,

moisture, wind direction and speed. The time for executing a (*, t) message at the Simulators, which reflects the computation intensity of the state transitions, was calibrated at 112 and 748 μs for *Fire1* and *Fire2* respectively.

The other model, called as *Watershed*, simulates the environmental influence on hydrological dynamics of water accumulation over 30 minutes in a 3D cell space. Though it is not as compute-intensive as Fire2 (577 μs state transition time), a larger neighborhood of 10 cells on different layers of the cell space is defined with increased communication intensity.

Unlike cellular automata models, which evaluate all the cells synchronously at discrete time steps, these Cell-DEVS models define the cell spaces as discrete-event models where each cell is an independent atomic model executed by a Simulator allowing efficient asynchronous execution without losing accuracy. These Cell-DEVS models execute a great number of simultaneous events at each virtual time, increasing the operational cost of TW simulation. In the next section, we will show that LTW is well-suited for improving simulation performance in such situations.

6.4.3. Test results and analysis

The comparative evaluation was conducted under the same configurations. Both protocols used aggressive cancellation and copy state-saving optimized with Message Type-based State Saving (MTSS). In addition, the optimal risk-free state-saving strategy introduced in Section 5 was enabled for LTW. In all test cases, message logging activities were turned off to minimize the impact of file I/O operations on system performance. Also, the corresponding test cases used the same partition scheme to divide the cell space (horizontal rectangles evenly distributed among nodes). In the following tables, a "×" mark indicates a failed test case due to memory exhaustion, while a shaded entry attributes the poor performance to severe memory swapping activities. A "–" mark stands for a case that was not tested because either the performance trend is already clear in the series, or the model cannot be divided further with the given partition scheme. The best execution time is highlighted in each series. The results (T and MEM) of a sequential simulator are also provided as a reference for evaluating the absolute performance of both protocols.

Table 4. Total execution time and maximum memory usage for Fire1

Size	Sequential	Prot.	Metric	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28
50×50	5.54 (T) 29.11 (MEM)	TW	T	×	9.08	5.87	5.26	**5.01**	5.39	5.49	5.55	5.95	–	–	–	–	–	–
			MEM	×	813.57	220.42	109.94	61.79	43.73	34.84	26.37	22.22	–	–	–	–	–	–
		LTW	T	5.78	3.61	3.02	2.98	**2.78**	3.01	3.23	3.25	3.54	–	–	–	–	–	–
			MEM	63.53	65.83	27.42	20.58	14.25	13.24	11.98	9.95	9.31	–	–	–	–	–	–
100×100	56.07 (T) 110.59 (MEM)	TW	T	×	×	2749.13	484.91	40.09	35.66	34.46	**32.35**	33.51	32.53	32.44	33.4	35.0	35.19	35.96
			MEM	×	×	2279.42	1492.31	882.82	576.61	410.19	307.79	244.6	197.97	162.92	137.77	121.47	103.03	91.75
		LTW	T	78.21	43.84	31.62	24.35	23.58	22.61	22.26	**21.62**	21.86	21.88	22.03	22.2	22.0	22.46	21.76
			MEM	405.5	373.25	271.62	160.26	110.94	82.65	66.75	55.65	48.18	43.55	38.92	36.22	34.05	32.3	29.94

150×150	260.65 (T) 242.69 (MEM)	**TW**	**T**	×	×	×	×	×	1516.48	893.43	572.83	314.03	202.71	141.46	**140.98**	142.63	142.01	143.18
			MEM	×	×	×	×	×	2309.12	1935.02	1449.83	1131.65	906.07	744.91	623.9	527.05	460.76	404.44
		LTW	**T**	1489.77	517.92	394.56	122.44	112.93	110.63	111.7	109.67	107.02	107.23	105.27	107.1	106.75	104.88	**104.74**
			MEM	1418.85	1294.08	986.62	660.31	415.01	296.96	230.4	186.68	161.7	137.22	123.85	105.07	96.8	90.88	85.09
200×200	815.43 (T) 432.13 (MEM)	**TW**	**T**	×	×	×	×	×	×	×	×	×	×	4324.31	1236.26	1065.79	881.61	**737.14**
			MEM	×	×	×	×	×	×	×	×	×	×	1848.93	1560.7	1528.73	1188.06	1058.7
		LTW	**T**	12571.7	6894.36	1425.16	920.86	646.56	350.58	334.77	331.2	333.12	326.7	327.56	327.46	**322.93**	330.03	327.24
			MEM	1679.36	1644.54	1393.66	1229.82	1145.6	805.17	582.49	431.18	393.15	291.49	244.01	209.52	235.47	186.1	188.68

Table 4 gives the resulting total execution time and maximum memory usage (T and MEM) for *Fire1* of varied sizes on different number of nodes. It is clearly shown that LTW outperforms TW counterpart in all successful cases. First, the maximum memory usage on each node is reduced by 45% up to 92%, making it possible to execute the model using a smaller number of nodes, with significantly lower simulation cost. Secondly, the total execution time is decreased by 24% up to 60% among those test cases with sufficient memory, and this improvement is achieved with a much smaller memory footprint at the same time.

Table 5. 100×100 Fire1 on 14 nodes

Metrics	TW	LTW	LTW vs. TW
PEE	96685.07	10597.71	
VEE	0	67214.07	
PQLen	24798.12	2636.95	↓ **89.37%**
VQLen	0	121.89	
SS	52819.64	22675.14	↓ **57.07%**
OPT-SK	0	18445.36	
FCT	488.14	84.15	↓ **82.76%**
PriRB	613.14	604.00	↓ **1.49%**
SecRB	11922.07	981.14	↓ **91.77%**
RB	12535.21	1585.14	↓ **87.35%**
EI	61751.93	5826.36	↓ **90.56%**
ER	48118.79	5790.93	↓ **87.97%**

To find out the reason that causes the differences, other metrics were compared. As an example, in Table 5 we present a comparison of the 100×100 *Fire1* on 14 nodes. The introduction of the volatile input queue reduces the average length of the persistent input queue by 89.37%, reducing the overhead of queue operations and memory consumption. On the other hand, the volatile queue is kept short throughout the simulation (average length of 121.89 events), despite the fact that 86.38% input events executed on each node have been turned into volatile under LTW.

Owning to the optimal risk-free state-saving strategy (which reduces the number of state-saving by 44.86% on top of MTSS), the total number of states saved in LTW is 57.07% fewer than in TW, resulting in less memory usage as well. As expected, the time for each fossil collection decreased from 488.14 ms to 84.15 ms (a reduction of 82.76%).

When comparing rollback performance, LTW shows a big advantage over TW. The number of secondary rollbacks is reduced by 91.77%, showing that rollback propagation is effectively con-

tained within the TW domain on each node. Moreover, the number of primary rollbacks reduced by 1.49%. This combined with the fact that the total number of events executed on each node (i.e., PPE + VEE) decreased by 19.52%, suggests a more stable system with less speculative computation. Consequently, the numbers of events imploded and unprocessed in the persistent queue also declined around 90%, further accelerating the rollback operations.

Table 6. Total execution time and maximum memory usage for Fire2

Size	Sequential	Prot.	Metric	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28
50×50	19.29 (T) 29.52 (MEM)	TW	T	×	20.89	13.93	12.19	10.91	**10.41**	10.8	10.64	10.84	10.55	11.31	12.51	12.76	13.39	13.44
			MEM	×	800.26	226.82	108.41	65.37	46.29	34.54	28.13	23.23	20.19	18.19	16.31	14.81	13.72	12.85
		LTW	T	20.26	14.23	10.38	9.69	9.46	8.84	9.01	8.51	8.64	8.4	**8.32**	9.28	9.34	9.51	10.27
			MEM	81.24	66.92	34.99	22.6	17.77	14.65	13.02	11.76	10.83	10.17	9.63	9.29	8.99	8.73	8.49
100×100	119.95 (T) 109.57 (MEM)	TW	T	×	×	3284.37	460.32	68.67	54.63	52.03	48.92	48.58	46.96	**46.37**	47.53	48.69	49.39	49.97
			MEM	×	×	2159.1	1319.08	658.14	576.72	411.14	310.95	240.42	198.4	163.47	149.65	112.23	99.94	83.78
		LTW	T	206.16	114.98	60.09	54.37	51.22	44.11	41.61	40.37	38.87	37.55	**35.54**	36.83	36.23	36.46	36.48
			MEM	314.37	285.18	248.32	137.73	102.24	81.63	65.57	54.35	48.91	45.62	42.6	38.42	35.75	33.84	32.03
150×150	414.25 (T) 243.71 (MEM)	TW	T	×	×	×	×	×	4448.08	2487.95	651.06	394.92	244.97	167.25	**164.79**	167.42	165.64	168.88
			MEM	×	×	×	×	×	1817.71	1375.23	1399.3	1086.72	905.96	744.91	562.55	532.14	425.91	399.51
		LTW	T	1592.43	493.61	223.65	178.2	174.63	165.84	168.66	167.14	140.67	140.21	137.0	134.3	136.11	**133.1**	134.01
			MEM	1210.37	924.16	641.79	586.92	385.41	269.62	205.4	172.18	139.44	122.16	112.93	104.22	94.47	89.39	85.79
200×200	1033.61 (T) 424.96 (MEM)	TW	T	×	×	×	×	×	×	×	×	×	12112.7	3206.02	1501.28	1202.48	900.05	**764.21**
			MEM	×	×	×	×	×	×	×	×	×	1943.55	1785.9	1618.94	1522.69	1475.58	1243.95
		LTW	T	11707.5	3363.07	1339.92	1173.69	562.68	414.52	412.92	412.89	381.1	376.58	417.44	373.11	372.6	**370.04**	371.56
			MEM	1661.95	1562.62	1267.71	1292.97	885.61	438.81	363.5	313.96	289.68	274.55	240.98	227.23	208.62	192.61	173.08

Table 7. Total execution time and maximum memory usage for Watershed

Size	Sequential	Prot.	Metric	1	2	4	6	8	10	12	14	16	18	20	22	24	26	28
15×15×2	258.27 (T) 43.99(MEM)	TW	T	×	×	2059.62	899.49	**84.97**	87.06	86.59	88.76	–	–	–	–	–	–	–
			MEM	×	×	1718.02	997.21	691.2	536.37	422.49	333.53	–	–	–	–	–	–	–
		LTW	T	262.99	171.18	112.69	100.54	**79.45**	82.27	82.08	82.59	–	–	–	–	–	–	–
			MEM	45.66	27.91	148.48	121.54	128.96	113.14	101.29	90.39	–	–	–	–	–	–	–
20×20×2	471.86 (T) 72.67 (MEM)	TW	T	×	×	×	×	2451.7	857.3	757.65	724.55	**638.97**	676.42	–	–	–	–	–
			MEM	×	×	×	×	1618.94	1180.67	967.51	778.53	643.52	535.21	–	–	–	–	–
		LTW	T	473.81	268.87	181.94	155.09	140.14	**104.77**	108.52	109.58	110.35	112.87	–	–	–	–	–
			MEM	76.02	40.04	164.35	136.36	130.82	149.81	137.24	129.85	115.87	111.99	–	–	–	–	–
25×25×2	735.39 (T) 115.48 (MEM)	TW	T	×	×	×	×	×	×	×	2002.73	1948.95	1922.21	1705.19	1597.08	**1585.6**	–	–
			MEM	×	×	×	×	×	×	×	1519.54	1434.77	1262.59	1063.03	774.38	663.21	–	–
		LTW	T	748.49	469.65	306.25	257.18	195.16	176.19	172.39	**136.18**	136.37	142.69	143.86	139.54	141.85	–	–
			MEM	119.8	70.46	164.86	128.68	131.07	132.81	132.27	153.82	141.87	128.25	114.39	113.95	103.44	–	–
30×30×2	1041.39 (T) 168.46 (MEM)	TW	T	×	×	×	×	×	×	×	×	5381.55	4475.37	3133.72	3130.89	2920.06	**2765.2**	2784.83
			MEM	×	×	×	×	×	×	×	×	2192.96	1867.83	1602.25	1388.87	1206.87	1055.31	924.49
		LTW	T	1098.11	616.28	390.68	293.33	237.82	208.26	204.82	198.27	169.12	168.45	168.01	165.54	165.64	166.55	**162.43**
			MEM	174.08	89.69	163.07	164.18	151.55	171.62	148.91	138.31	117.57	139.45	156.5	149.91	130.2	122.69	114.69

The experimental results for the *Fire2* and *Watershed* models are shown in Table 6 and Table 7 respectively. Again, LTW reduces maximum memory consumption by approximately 34% up to 92% for *Fire2* and by 73% up to 93% for *Watershed*. The reduction in memory usage is more prominent for *Watershed* largely because, with a higher number of simultaneous events exchanged between the LPs at each virtual time, a larger percentage of states are reduced with the optimal state-saving strategy.

For those cases with sufficient memory, the total execution time decreased by 13% up to 32% for *Fire2* and by 5% up to 91% for *Watershed*. A general trend reflected in the experimental results is

that the reduction in execution time and memory usage is greater for models with larger sizes, indicating an improved scalability.

Other metric values for the 100×100 *Fire2* on 20 nodes and 20×20×2 *Watershed* on 18 nodes are given in Table 8 and Table 9 respectively. As we can see, a similar pattern can be observed regarding the improvement of the metrics, suggesting that LTW is suitable for simulating models with varied computation and communication characteristics.

Table 8. 100×100 Fire2 on 20 nodes

Metrics	TW	LTW	LTW vs. TW
PEE	68346.55	11658.75	
VEE	0	56057.00	
PQLen	17533.37	2149.91	↓ **87.74%**
VQLen	0	75.31	
SS	33833.00	17565.40	↓ **48.08%**
OPT-SK	0	15591.10	
FCT	245.12	58.36	↓ **76.19%**
PriRB	769.95	740.55	↓ **3.82%**
SecRB	12794.35	2036.45	↓ **84.08%**
RB	13564.30	2777.00	↓ **79.53%**
EI	46877.55	7197.90	↓ **84.65%**
ER	29512.45	6651.60	↓ **77.46%**

Table 9. 20×20×2 Watershed on 18 nodes

Metrics	TW	LTW	LTW vs. TW
PEE	1253641.94	361457.78	
VEE	0	856256.00	
PQLen	334016.67	77790.62	↓ **76.71%**
VQLen	0	26.04	
SS	371273.33	73186.94	↓ **80.29%**
OPT-SK	0	288247.50	
FCT	61313.67	395.63	↓ **99.35%**
PriRB	173.50	159.94	↓ **7.81%**
SecRB	22816.67	2165.33	↓ **90.51%**
RB	22990.17	2325.28	↓ **89.89%**
EI	625210.33	175521.11	↓ **71.93%**
ER	569337.94	172280.33	↓ **69.74%**

In terms of absolute performance, LTW attain higher and more consistent speedup than TW. In some scenarios, the performance of a TW simulation is even worse than the sequential execution (e.g., 50×50 *Fire1* on 2 and 4 nodes; 20×20×2 *Watershed* on 14, 16, and 18 nodes) mainly due to the excessive communication and operational overhead. However, such scenarios do not arise in the LTW cases tested.

7. Summary

In this chapter we presented and analyzed the performance of our two existing parallel CD++ simulators, namely the Conservative PCD++ simulator and Optimistic PCD++ simulator with a focus on the last one. We looked at the design and implementation of these PCD++ simulators and compared their structures as well as functionalities in parallel and distributed simulations. The hierarchical structure of the conservative PCD++ simulator was compared against the flattened structure of the optimistic PCD++ simulator. We illustrated the migration from a hierarchical structure to a flattened structure as two major modifications; the departure from conservative-based simulator to an optimistic-based simulator, and flattening the structure. Then we showed how the optimistic PCD++ simulator deals with the communication overhead by a flat structure.

Aiming at improving the performance of the optimistic simulator, we modified the WARPED kernel to handle rollbacks in a more efficient way. We presented new algorithms that we have implemented in WARPED kernel: The *Near-perfect State Information* protocol and the Local Rollback Frequency Model (LRFM), and the Global Rollback Frequency Model (GFRM). We run a variety of tests to analyze the performance of our existing PCD++ simulators; the optimistic and the conservative as well as our LRFM and GRFM Time Warp-based protocols. The main goal of these tests was to show the maximum capability of the two mentioned PCD++ simulators in terms of handling the number of nodes driving the simulation, complexity of the model, and the size of the model.

A novel optimistic synchronization protocol, referred to as Lightweight Time Warp (LTW), has been proposed and its performance compared to the standard Time Warp. LTW offers a novel approach that systematically addresses several important issues of TW-based systems (especially for DEVS-based simulations that require a large number of simultaneous events to be executed at each virtual time). It allows purely optimistic simulation to be driven by only a few full-fledged TW LPs, preserving the dynamics of the TW mechanism, while at the same time, accelerating the execution in each local simulation space significantly. Both TW and LTW have been implemented in optimistic PCD++. The experimental results demonstrated that LTW outperforms TW in various aspects, including shortened execution time, reduced memory usage, lowered operational cost, and enhanced system stability and scalability. We are currently working on integrating LTW with other TW optimizations to further improve performance. By taking advantages of the LTW

protocol, we are also investigating dynamic process creation, deletion, and migration schemes to support more efficient load balancing as well as runtime structure changes in optimistic DEVS systems.

References

[1] Fujimoto, R. M. 2000. *Parallel and distributed simulation systems*. New York: Wiley.
[2] Wainer, G. 2002. CD++: A toolkit to develop DEVS models. *Software – Practice and Experience*, 32:1261-1306.
[3] Wainer, G. "Discrete-Event Modeling and Simulation: a Practitioner's approach". CRC Press. Taylor and Francis. 2009.
[4] G. Wainer. "Applying Cell-DEVS Methodology for Modeling the Environment". SIMULATION: Transactions of the Society for Modeling and Simulation International, Vol. 82, No. 10, pp. 635-660, 2006.
[5] U. Farooq, G. Wainer, and B. Balya. "DEVS Modeling of Mobile Wireless Ad Hoc Networks". Simulation Modeling Practice and Theory, Vol. *15, No.* 3, pp. 285-314, 2006.
[6] J. Yu, G. Wainer. "eCD++: an engine for executing DEVS models in embedded platforms". In Proceedings of the 2007 Summer Computer Simulation Conference. San Diego, CA. 2007.
[7] Troccoli, A. and G. Wainer. 2003. "Implementing parallel Cell-DEVS". In Proceedings of the 36th IEEE Annual Simulation Symposium (ANSS'03), pp. 273-280, Orlando, FL, USA.
[8] G. Wainer, E. Glinsky. "Advanced Parallel simulation techniques for Cell-DEVS models". Simulation News Europe. EUROSIM. Vol. 16, No. 2. September 2006. pp. 25-36.
[9] Q. Liu, G. Wainer. "Parallel Environment for DEVS and Cell-DEVS Models". Simulation, Transactions of the SCS. Vol. 83, No. 6, 449-471. 2007.
[10] G. Wainer, K. Al-Zoubi, R. Madhoun. "Distributed simulation of DEVS and Cell-DEVS models in CD++ using Web-Services". Simulation Modeling, Practice and Theory. 16 (9). pp. 1266–1292. October 2008.
[11] 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.
[12] D. R. Jefferson. 1985. "Virtual time". *ACM Trans. Program. Lang. Syst. 7*(3), pp. 404-425.
[13] J. Fleischmann, P. A. Wilsey, "Comparative analysis of periodic state saving techniques in time warp simulators". In *Proceedings of PADS'95*, Washington DC, 1995, pp. 50-58.
[14] J. S. Steinman, "Breathing time warp". *SIGSIM Simul. Dig. 23*(1), 1993, pp. 109-118.
[15] Y. B. Lin, E. D. Lazowska, "A study of time warp rollback mechanisms". *ACM Trans. Model. Comput. Simul. 1*(1), 1991, pp. 51-72.
[16] C. D. Carothers, K. S. Perumalla, R. M. Fujimoto, "Efficient optimistic parallel simulations using reverse computation". *ACM Trans. Model. Comput. Simul. 9*(3), 1999, pp. 224-253.
[17] 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.
[18] Zeigler, B. P., G. Ball, H. Cho, J. Lee, and H. Sarjoughian. 1999. "Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions". In 1999 Fall Simulation Interoperability Workshop.
[19] Nutaro, J. 2004. "Risk-free optimistic simulation of DEVS models". In Proceedings of the Advanced Simulation Technologies Conference, Arlington, VA, USA.
[20] Nutaro, J. 2008. "On Constructing Optimistic Simulation Algorithms for the Discrete Event

System Specification". Accepted for publication in the ACM Transactions on Modeling and Computer Simulation (TOMACS).
[21] Srinivasan, S. and P. F. Reynolds. 1998. "Elastic Time". ACM Transactions on Modeling and Computer Simulation (TOMACS) 8(2), pp. 103-139.
[22] Szulsztein, E.; Wainer, G. "New Simulation Techniques in WARPED Kernel" (in Spanish). Proceedings of JAIIO, Buenos Aires, Argentina, 2000.
[23] Reynolds, P. F. 1988. "A Spectrum of Options for Parallel Simulation". *Proceedings of the 20th Winter Simulation Conference,* pp. 325-332, San Diego, CA, USA.
[24] Preiss, B. R. and W. M. Loucks. 1995. "Memory Management Techniques for Time Warp on a Distributed Memory Machine. *SIGSIM Simul. Dig. 25*(1), pp. 30-39.
[25] Jefferson, D. 1990. "Virtual Time II: Storage Management in Conservative and Optimistic Systems". In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pp. 75-89, New York, NY, USA.
[26] Lin, Y.-B. and B. R. Preiss. 1991. "Optimal Memory Management for Time Warp Parallel Simulation". *ACM Trans. Model. Comput. Simul. 1*(4), pp. 283-307.
[27] Brown, R. 1988. "Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem". *Commun. ACM 31*(10), pp. 1220-1227.
[28] Rönngren, R., R. Ayani, R. M. Fujimoto, and S. R. Das. 1993. "Efficient Implementation of Event Sets in Time Warp". *SIGSIM Simul. Dig. 23*(1), pp. 101-108.
[29] Oh, S. and J. Ahn. 1999. "Dynamic Calendar Queue". In *Simulation Symposium, 1999. Proceedings. 32nd Annual*, pp. 20-25.
[30] Schof, S. 1998. "Efficient Data Structures for Time Warp Simulation Queues". *Journal of Systems Architecture 44* (6), pp. 497-517.
[31] El-Khatib, K. and C. Tropper. 1999. "On Metrics for the Dynamic Load Balancing of Optimistic Simulations". In *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences (HICSS-32).* Maui, HI, USA.
[32] Wilson, L. F. and W. Shen. 1998. "Experiments in Load Migration and Dynamic Load Balancing in SPEEDS". In *WSC'98: Proceedings of the 30th Conference on Winter Simulation*, pp. 483-490, Los Alamitos, CA, USA.
[33] Carothers, C. D. and R. M. Fujimoto. 2000. "Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms". *IEEE Transactions on Parallel and Distributed Systems 11* (3), pp. 299-317.
[34] Peschlow, P., T. Honecker, and P. Martini. 2007. "A Flexible Dynamic Partitioning Algorithm for Optimistic Distributed Simulation". In *PADS '07: Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation*, pp. 219-228, Washington, DC, USA.
[35] Reiher, P. L. and D. Jefferson. 1990. "Dynamic Load Management in the Time Warp Operating System". *Trans. Soc. Comput. Simul. Int. 7* (2), pp. 91-120.
[36] Li, L. and C. Tropper. 2004. "Event Reconstruction in Time Warp". In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation (PADS 2004),* pp. 37-44.
[37] Liu, Q. and G. Wainer. 2008. "Lightweight Time Warp – A Novel Protocol for Parallel Optimistic Simulation of Large-Scale DEVS and Cell-DEVS Models". In *Proceedings of the 12th IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2008)*, pp. 131-138, Vancouver, BC, Canada.
[38] H. Rajaei, "Local Time Warp: An implementation and performance analysis". Proceedings of PADS, 2007, pp. 163-170.

[39] Rothermel, R. 1972. "A Mathematical Model for Predicting Fire Spread in Wild-land Fuels". Research Paper INT-115. Ogden, UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station.
[40] Moon, Y., B. Zeigler, G. Ball, and D. P. Guertin. 1996. "DEVS Representation of Spatially Distributed Systems: Validity, Complexity Reduction". IEEE Transactions on Systems, Man and Cybernetics. pp. 288-296.
[41] Al-aubidy, B.; Dias, A.; Bain. R.; Jafer, S.; Dumontier, M.; Wainer, G.; Cheetham, J. "Advanced DEVS models with applications to biomedicine". Artificial Intelligence, Simulation and Planning. Buenos Aires, Argentina. AIS 2007.
[42] Jafer, S.; Wainer, G. "An Environment for Advanced Parallel Simulation of Cellular Models". Proceedings of the International Conference on Unconventional Computation'07. 2007.
[43] Ameghino, J., A. Troccoli, and G. Wainer. 2001. "Models of Complex Physical Systems Using Cell-DEVS". The 34th IEEE/SCS Annual Simulation Symposium.
[44] C. D. Bevins, "fireLib User Manual and Technical Reference". http://www.fire.org/, accessed in Dec. 2008.