New Parallel Simulation Techniques of DEVS and Cell-DEVS in CD++

Ezequiel Glinsky Gabriel Wainer
Dept. of Systems and Computer Engineering
Carleton University
4456 Mackenzie Building
1125 Colonel By Drive
Ottawa, ON. K1S 5B6. CANADA.
{eglinsky, gwainer}@sce.carleton.ca

Abstract

DEVS is a sound formal modeling and simulation (M&S) framework based on generic dynamic system concepts. Cell-DEVS is a formalism for cell-shaped models based on DEVS. This work presents a new simulation technique for execution of DEVS and Cell-DEVS models in parallel/distributed environments. The parallel simulator is based on Time Warp, and developed as a new simulation engine for CD++, a M&S toolkit that implements DEVS and Cell-DEVS theories. The technique uses a non-hierarchical approach that simplifies the structure of the simulator and reduces the communication overhead. The results obtained allowed us to achieve considerable speedups.

1. Introduction

The widespread use of M&S in different application domains is leading to execution of larger and more complex systems, which often translates into more memory and processor requirements [1]. Parallel discrete event simulation (**PDES**) studies the execution of discrete event models (i.e., those in which simulation advances by the occurrence of events that take place at discrete points in time) in parallel or distributed computers. The main concern of this community was to reduce execution time of applications by using multiple processors, and a large number of synchronization algorithms were developed [1].

Another approach considered using of the DEVS formalism [2] as the modeling framework for PDES [3,4,5]. DEVS is a sound formal framework based on generic dynamic systems concepts that supports provably correct, efficient, event-based simulation. DEVS enables the construction of models in a hierarchical, modular fashion, allowing component reuse and reducing development and testing time. **Parallel DEVS** or **P-DEVS** [6] provides means of handling simultaneous scheduled, while keeping all the major properties of classic DEVS. Since P-DEVS eliminates serialization constraints, it enables improved execution of models in parallel and distributed environments.

Cell-DEVS [7] combines Cellular Automata with DEVS theory, allowing individual cells to be defined as basic DEVS models and coupled to form complete cell spaces. **CD++** [8] is a M&S tool that implements DEVS and Cell-DEVS theories. A hierarchical, conservative parallel simulator was implemented in CD++ [9], showing improvements for both DEVS and Cell-DEVS. However, its degree of parallelism and speedups are bounded. Here, we introduce a new technique for optimistic simulation in CD++. The technique combines the Time Warp synchronization mechanism and the DEVS abstract simulators. In our approach, the hierarchy of the simulation objects is flattened to reduce the communication overheads, using a flat simulation approach that eliminates the need for intermediate coordinators [3]. Consequently, it reduces the overhead of message passing, improving the overall performance of the simulation.

2. Background

The **DEVS** formalism [2] provides a framework for the definition of hierarchical and modular models. A real system modeled with DEVS is described as a composite of behavioral (atomic) or structural (coupled) submodels. P-DEVS [6] provides a way of dealing with simultaneous events. An atomic P-DEVS model is defined as:

$$M = \langle X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$$

At any given time, an atomic model is in state s during a period defined by ta(s). When that time expires, the system outputs the value $\lambda(s)$ and then it changes to the state specified by $\delta_{int}(s)$. If one or more external events (X_M) occur before ta(s), the new state is given by the external transition function, $\delta_{ext}(s, e, X_M)$. δ_{ext} uses a bag of events, which allows multiple events to be processed simultaneously. If external and internal transitions conflict, the new state is given by δ_{con}.

Coupled models are defined as a set of basic components (atomic or coupled), which are interconnected through the model's interfaces.

$$CM = \langle X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC \rangle$$

X, is the set of input ports, Y is the set of output ports, D is a set of the component names; for each $d \in D$, M_d is a basic DEVS model; the external input couplings (EIC) defines how to connect external inputs to components; the external output couplings set (EOC) defines how to connect components to external outputs; and the internal couplings set (IC) defines how to interconnect components.

Cell-DEVS [7] is an extension that allows the specification of executable cell spaces with explicit timing delays. A parallel Cell-DEVS atomic model [10] can be formally defined as:

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

Each cell uses a set of N inputs to compute the next state. These values are received through a well-defined interface (X^b, Y^b), activating a local function (τ, τ_{con}), which uses the cell's inputs and present state (S). A delay function (d) associated with each cell, allows deferring the transmission of the results. The model advances through the activation of the internal, external, output and state's duration functions, as in other DEVS models. After the basic behavior for a cell is defined, the complete cell space will be constructed by building a coupled Cell-DEVS model:

$$GCC = \langle Xlist, Ylist, X, Y, n, \{t_1,...,t_n\}, N, C, B, Z \rangle$$

The cell space is a coupled model composed of an array of $t_1 \ldots t_n$ atomic cells (C). Each of them is connected to the cells defined by the neighborhood (N). The border (B) can be provided with a different behavior than the rest of the space. The Z function allows one to define the internal and external coupling of cells in the model, using the neighborhood definition and to the external models through Xlist and Ylist.

CD++ [8] implements P-DEVS and Cell-DEVS formalisms. The tool was built as a class hierarchy in C++, where each class corresponds to a simulation entity using the basic concepts defined in [2]. There are two basic abstract classes: *Model* and *Processor*. The former is used to represent the behavior of the atomic and coupled models, while the latter implements the simulation mechanisms. *Simulators* manage the atomic models, handling the execution of δ_{int}, δ_{ext}, δ_{con} and λ. *Coordinators* manage coupled models. The *Root Coordinator* manages global aspects (global time, starting/stopping the simulation, I/O, etc). *AtomicCell* and *CoupledCell* simulate cell spaces.

The simulation process is message-driven. Each message contains information to identify the *sender* and the *receiver*, including a *timestamp* for the message and an associated *value*. There are two main categories of messages: **synchronization** (@: Collect message; *: Internal message; done: Done message) and **content** messages (q: External message; y: Output message).

As mentioned earlier, our goal is to combine advanced DEVS simulators with PDES techniques. In PDES, the simulation is subdivided in smaller, simpler parts that run on different nodes. Warped [11] is a simulation kernel that provides an implementation of Time Warp with different optimizations, and an interface that hides most of the implementation issues. Warped is written in C++ and uses the MPI message passing standard for communication, a standard designed for high performance communication on parallel and distributed environments. Simulation objects within the same LP exchange messages using direct communication, whereas those running in different LPs use MPI communication services.

CD++ was originally developed as a stand-alone simulator, and it was redesigned to provide parallel execution of P-DEVS and Parallel Cell-DEVS. Although Parallel CD++ showed speedups, a single Root Coordinator still acts as a global scheduler for every node in the simulation. Another problem is that most DEVS simulators usually create a one-to-one correspondence between model components and simulation objects (Figure 1), increasing the communication costs of message passing.

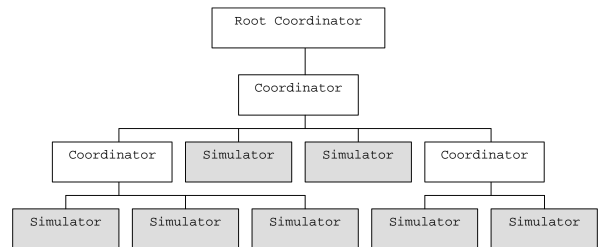

Figure 1. DEVS hierarchical simulator.

Flat simulation mechanisms, instead, reduce this overhead by simplifying the underlying simulator structure, while keeping the same model definition and preserving the separation between model and simulator.

Studies have shown that flat simulators can outperform hierarchical mechanisms.

3. Optimistic PDES of DEVS Models

The fundamental classes in CD++ can be divided in two major groups: those who inherit from the basic *model* or *processor* classes. This reflects the clear distinction between the model and its simulator. All classes inheriting from *model* remain unchanged from those defined in earlier versions of the tool. Two new classes are introduced, both inheriting from *processor*: *Flat Coordinator* (FC) and *Node Coordinator* (NC). Additionally, we modified the *Simulator* and *Root Coordinator* classes. The algorithms we defined are based on those in [12] and [10], as it has been proven that they correctly simulate P-DEVS models.

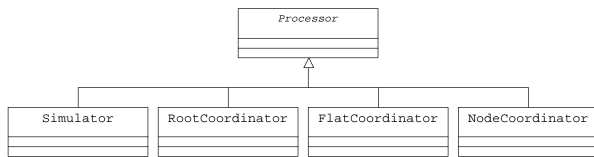

Figure 2. New processors' class hierarchy in CD++

Two processors (*coordinator* and *cell coordinator*) have been eliminated in the new hierarchy presented in Figure 2. Figure 3 presents the new structure for the model in Figure 2.

The Root Coordinator only handles I/O operations, and starts/stops the simulation. The NC is in charge of synchronization and time management for the LP. The FC is responsible for receiving, translating, and sending messages between its descendants, contained on a flat data structure for handling all the coupling information for every component. In order to run the model on a distributed environment, we need to indicate the nodes that can participate in the simulation, and how they are allocated to each processor (Figure 4). During the instantiation and registration of each Simulator object, they are associated to the corresponding LP. NCs can communicate with each other using inter-LP messaging. The Root Coordinator executes on one LP, and it forwards messages from the environment to the corresponding NC. On the other hand, when a NC processes an output that must be sent back to the environment, it is sent to the Root Coordinator.

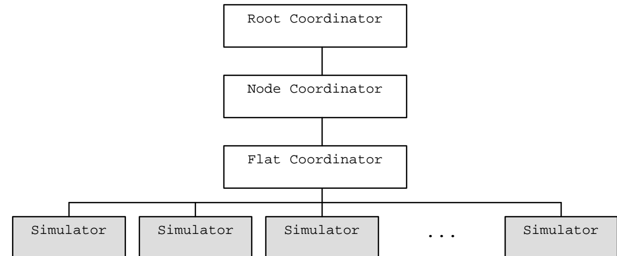

Figure 3. Processor hierarchy using a flat approach

```
0:atomic_4 atomic_5
1:atomic_1 atomic_2 atomic_3
2:atomic_6 atomic_7
```

When an output is sent from an atomic component, a_1, to another, a_2, we can identify two different cases: both Simulators for a_1 and a_2 execute on the same or on different LPs. In the first case, the FC running on that LP takes care of the situation: the source Simulator sends the message to its parent FC (which has all the information for the port mappings), and it sends the output to the corresponding Simulator. In the second case, a Simulator on LP_i has to send an output to a Simulator on LP_j. FC_i identifies that the destination Simulator is not one of its descendants, it forwards the message to its parent NC_i, which identifies the corresponding LP_j and forwards the message. The NC running on LP_j forwards the message to FC_j, which in turn sends it to the destination Simulator.

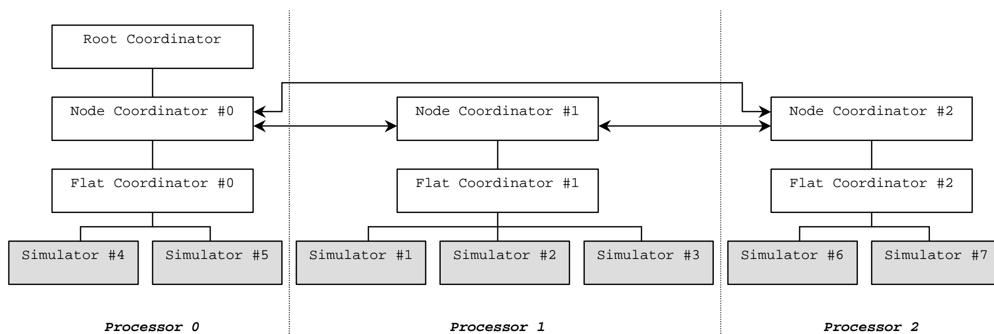

Figure 4. Model partition file for CD++

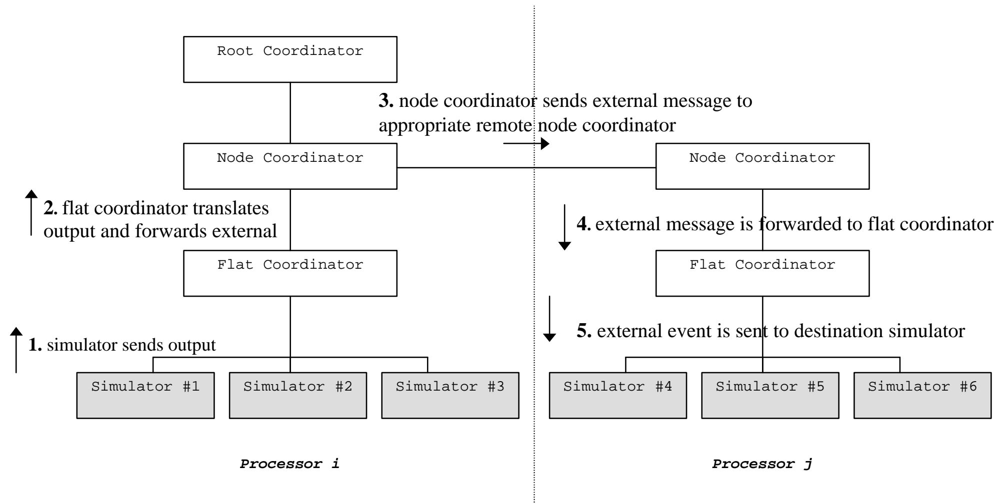

Figure 5. Sending an output to a remote simulator

Notice that inter-LP communication can lead to violations to the local causality constraint, depending on the time at local and destination LPs. More specifically, if the timestamp of the message is smaller than the local time at the destination LP, a rollback is triggered. A detailed description of the algorithms involved in each of the components can be found in [13].

5. Implementing the Abstract Simulator

We defined the previous algorithms using different services provided by Warped. On the Warped API, *TimeWarp* is a basic class defining the data and methods to allow every object to participate in a simulation. Three main methods determine the behavior of the objects: *initialize*, *finalize* and *executeProcess*. *executeProcess* runs every time a simulation object is scheduled for execution (i.e., when it has an event ready to be processed). *finalize* releases memory, collects statistics, etc. *saveState* is called automatically to save the current state of an object. In case of receiving a straggler message with a timestamp *t*, *rollback(t)* restores the state of the object and sends the necessary anti-messages. *calculateMin* reports the minimum time of the unprocessed events, and is used to compute the global virtual time. *inputGcollect*, *stateGcollect*, and *outputGcollect* take care of garbage collection in the input, state, and output queues. The state of a simulation object is defined by an instance of *BasicState*. The state of an object contains the information that can change in each simulation cycle, including pointers to input and output queues (*inputPos* and *outputPos*). Simulator objects communicate by message passing, which belong to the class *BasicEvent* or to one of its subclasses. *LogicalProcess* groups the simulation objects executing in the same processor. To create a new LP, it is necessary to specify the total number of objects in the simulation, the number of simulation objects to be handled on this LP, and the number of LPs participating in the simulation. The method *registerObject(TimeWarp)* is used to define which objects are running on this LP. The method *simulate(VTime)* starts the execution of this LP (the simulation stops when the global time is greater than the specified time). *calculateLGVT* is used to compute the local global virtual time at the end of each simulation cycle. It is calculated by a *GVTManager* as the minimum time reported by simulation objects. Figure 6 shows the new class diagram of the DEVS processors along with some of their main methods that implement the algorithms previously described.

*Processor (*inherits from Warped *TimeWarp)* provides basic functionality and data that are common to all DEVS processors in the application (methods *initialize*, *executeProcess, finalize*,etc.). It includes the definition of:

a. send methods for each type of message (e.g., *send(initMsg,dest)*, *send(doneMsg,dest)*), which use, in turn, *sendEvent* in *TimeWarp*.
b. Time management methods (e.g., *timeNext()*, *timeLast()*, *timeNext(VTime)*, etc.).
c. *initialize*, *finalize*, and debugging methods.
d. *executeProcess()*, which defines the behavior of any DEVS processor.
e. *rollbackCheck()*,called in the receive method, to check for straggler messages.
f. Basic variables (model associated to this processor, its parent, id and descriptors).

The method *processor.executeProcess()* is in charge of getting the first event in the queue of events, logging the event, and calling the corresponding receive method based on the message type.

The *receive(initMessage)* method first sends initialization messages to all of its descendants (*send(initMessage,dest)*) and waits for all *done* messages from its dependant Simulators. The *nodeCoordinator* keeps track of the number of *done* messages (*doneCount()*) and it determines the time of next change (*nextChange(VTime)*), sening this value to its parent NC (*send(doneMsg,dest)*). The *receive(initMessage)* method, in contrast, initializes the model variables, computes the next time for the next transition and sends a *done* message to its parent, which is a FC. The Simulator executes δ_{ext}, δ_{int}, δ_{con}, λ, ta.

Figure 6. Class diagram for the new DEVS processors

5. Execution Results

We carried out different performance tests to analyze the results obtained with the new algorithms. To provide uniform means for the overhead, we used the DEVStone benchmark, a synthetic model generator that automatically creates models. Its accuracy relies on the execution of a large pool of models to provide a robust test set for the study. DEVStone generates models with different size, complexity and behavior, resembling different kinds of applications [13]. DEVStone uses three different types of models with variations in their internal and external structure:

- **LI** models, with a low level of interconnections for each coupled model,
- **HI** models with a high level of input couplings, and
- **HO** models with high level of coupling and numerous outputs.

Table 1 shows the parameters we used for different tests, including model type, structure and time spent on transition functions (e.g., model E is of HI type, it is composed of 3 levels, with 6 components per level).

Table 1. Simulation parameters

Name	Type	Depth	Width	d_{int}	d_{ext}
A	LI	3	10	50 ms	50 ms
B	LI	10	3	50 ms	50 ms
C	LI	5	5	50 ms	50 ms
D	LI	10	10	50 ms	50 ms
E	HI	3	6	50 ms	50 ms
F	HI	6	3	50 ms	50 ms
G	HI	5	5	50 ms	50 ms
H	HI	6	6	50 ms	50 ms
I	HO	3	6	100 ms	0 ms
J	HO	6	3	0 ms	100 ms
K	HO	5	5	50 ms	50 ms
L	HO	6	6	50 ms	50 ms

Figure 7 shows the overhead obtained for these models executed on a single processor using the stand-alone (Original CD++), conservative (Parallel NoTime), and optimistic (Parallel TimeWarp) approaches.

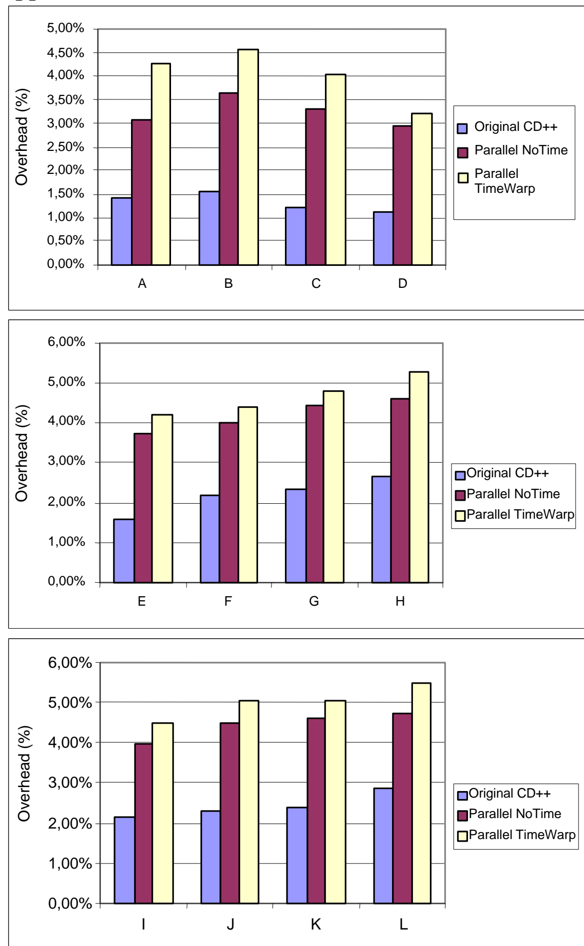

Figure 7. Overhead incurred by the abstract simulators.

The percentage of overhead of the parallel versions is below 5.5% for the most complex problems running on top of the Warped and MPICH middleware. This is a promising result, as the amount of speedup time achievable by these simulators is considerable, and having a constrained overhead in the kernel permits a better utilization of the computing resources.

We also studied the performance of our new simulator using variations of a sample Cell-DEVS model representing the execution of the 'Life' game. We executed the life game using different cell spaces: 16x16 (256 cells), 20x20 (400 cells), 25x25 (625 cells) and 30x30 (900 cells). The initial configuration of cells for each model was randomly generated. First, the models were executed on one and four processors. We used simple rectangular partitions for the distributed case (Figure 8).

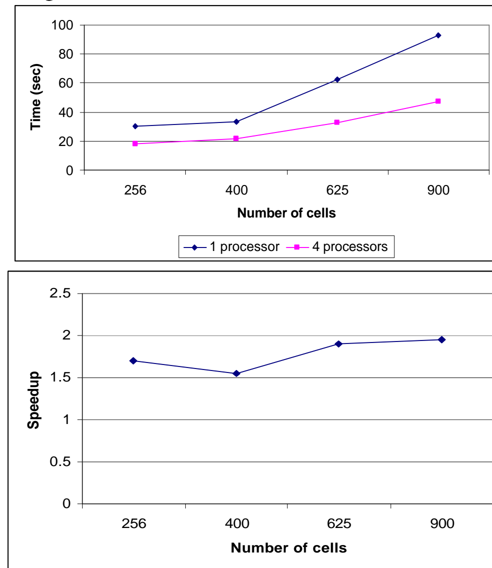

Figure 8. Execution times; Execution speedups (1 vs. 4 processors)

The distributed execution of the model outperformed the execution in a single processor. The execution time for the model running on one processor varies from 30.7 to 90.8 seconds. When running the model in parallel on 4 processors, the execution time is smaller (between 18.1 and 47.5 seconds); in some cases, the optimistic simulator allows to reduce the execution time in ~50%. It also shows that the factor of speedup falls between 1.55 and 1.95 when distributing the execution of the life model among 4 processors using this partitioning approach. In these particular cases, the speedup has been affected by the communication costs, as the tests were executed over a relatively slow network (a 10 Mbit/s hub, which limits the simultaneous transfers rate to 10 Mbits per second).

Results presented in [9] show promising results when faster networks are used.

Figure 9 shows a comparison between our parallel simulator and the previous conservative simulator for different configurations of 30x30 (*life 1-4*) and 40x40 (*life 5-8*) models using 4 processors.

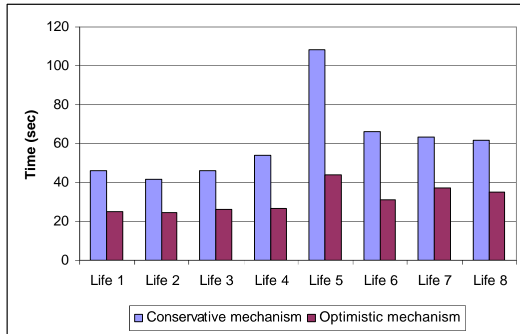

Figure 9. Execution times using optimistic and conservative simulators (4 processors)

Figure 9 shows that the optimistic simulator outperforms the conservative simulator for all configurations of 30x30 and 40x40 models. In the configuration labeled as *life 5* (a 30x30 model), most of the 900 cells are active in the first cycles of the simulation. In cases like this, we observe the largest difference in execution times. In general, the difference is a result of the performance gains obtained not only by distributing the simulation in multiple processors but also by distributing the scheduling tasks in multiple NCs.

We are interested in analyzing the performance of our simulator for larger Cell-DEVS. The following figures show the execution times and speedups for different configurations for a cell space of 50x50 (different initial values were used, shown as *life A*, *B*, *C*, and *D* in Figure 10).

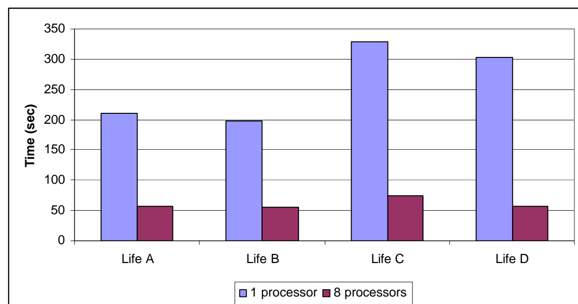

Figure 10. Execution times; Execution times (50x50 life model in 1 and 8 processors)

The execution times for these cases are significantly reduced when we distribute the simulation in 8 processors. When a 50x50 model is executed on a single processor, only one LP is created. Hence, a single instance of a FC is in charge of the 2500 Simulators participating in the simulation, and a single NC is in charge of scheduling tasks for the entire model, for instance, the time required to update the list imminent components (i.e., models that are scheduled for a transition), which is maintained by a single FC. In contrast, the distribution of this model in 8 processors allows a smaller structure associated with each LP participating in the simulation (each LP has an associated FC/NC in charge of 312 Simulators). Figure 10 shows that distributing the simulation of a large model in 8 processors allows significant execution speedups.

The following set of tests uses a sample Cell-DEVS model to study the performance of a firefly model, in which most of the cells change frequently, producing increased processor load. These rules produce changes for almost every cell at every simulation cycle. We execute models with 400 and 900 cells, using two different initial configurations for each case. Figure 11 shows that the simulation in 4 processors using the optimistic simulator achieves the best performance for all the cases. The conservative simulator distributed in 4 processors outperforms its single-processor counterpart. The optimistic simulator running on a single processor achieves almost the same performance as the conservative simulator running on 4 processors, which shows the increased communication costs of the latter alternative and the good performance achieved by our simulator. Figure 11 shows the speedup of the optimistic simulator distributed in 1 and 4 processors in relation to the conservative simulator for the 20x20 and 30x30 models.

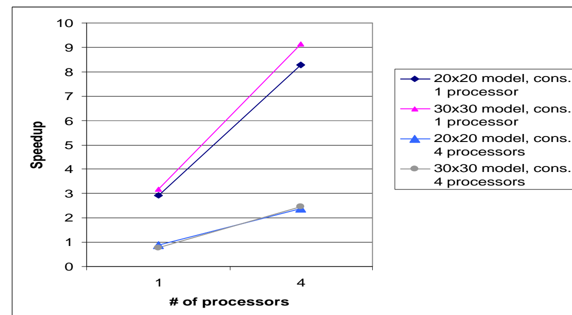

Figure 11. Speedups using conservative and optimistic simulators; Speedup (1-4 processors)

Figure 11 illustrates the speedups obtained by our simulator using 1 and 4 processors in relation with the conservative simulator. The figure shows that the execution of the optimistic simulator in 1 processor allows significant speedups (2.91 for 20x20 models, 3.17 for 30x30 models) in comparison to the conservative simulator running on a single processor. The speedup factor obtained by executing the simulation in 4 processors using the optimistic approach instead of the equivalent partitioning for the conservative approach is approximately 2.45 for 20x20 and 30x30 models. The execution of the model using our approach in 4 processors enables speedup factors of up to 9.15 in comparison to the execution in a single-processor using the pessimistic technique. Although the execution of both 20x20 and 30x30 models using the pessimistic approach in 4 processors outperforms our simulator executing in 1 processor, it is only by a relatively small fraction (the speedup factor is .82-.86).

6. Conclusions

We have introduced a new flat simulation technique for P-DEVS and Cell-DEVS based on Time Warp, a well-known optimistic synchronization protocol. Our efforts address the need for efficient, fast execution of models using parallel and distributed simulation. We propose an optimistic distributed mechanism that enables achieving higher degrees of parallelism than previous efforts, which only allowed exploiting parallelism in a limited way. Under our new approach, scheduling tasks are distributed on the LPs; each NC is in charge of the scheduling tasks for the local simulation objects. NCs advance the simulation optimistically, assuming that there will be no straggler events. In case of detecting a violation to the local causality constraint, a rollback mechanism allows recovering from it.

Using DEVStone, we compared the overhead of our new technique with the overhead of previous implementations. Although the overhead associated with synchronization tasks implemented by our simulator can be considerable, it still outperformed previous alternatives for some models in single-processor executions. This is a consequence of the flat mechanism implemented in our engine, which outweighs the increased overhead associated with its more complex implementation. More importantly, we showed that when executing different types of DEVS models, the overhead is reasonable small (2.5%-5%).

We showed that the execution times for a particular Cell-DEVS model can be reduced using distributed simulation. Different model sizes where considered, ranging from 256 to 2500 cells. The execution of the model in a distributed environment allowed achieving better performance than stand-alone execution. Using distributed environments, our simulator outperforms other alternatives and achieves considerable speedups.

7. References

[1] Fujimoto, R.M. *Parallel and Distribution Simulation Systems.* Wiley. 1999.

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

[3] Kim, K.H.; Seong, Y.R.; Kim, T.G.; Park, K.H. “Distributed Simulation of Hierarchical DEVS Models: Hierarchical Scheduling Locally and Time Warp Globally” *Trans. of the SCS.* vol. 13(3), pp. 135-154. 1996.

[4] Troccoli, A.; Wainer, G. “Implementing Parallel Cell-DEVS.” Proceedings of the Annual Simulation Symposium. Washington DC, USA. 2003.

[5] Zeigler, B.; Moon, Y.; Kim, D.; Ball, G. “The DEVS Environment for High-Performance Modeling and Simulation” *IEEE Computational Science and Engineering.* 4 (3), pp. 61 -71. 1997.

[6] Chow, A.C.; Zeigler, B.P. “P-DEVS: A parallel, hierarchical, modular modeling formalism.” Proceedings of the Winter Simulation Conference. Orlando, FL. USA. 1994.

[7] G. Wainer, N. Giambiasi. "N-Dimensional Cell-DEVS". *Discrete Events Systems: Theory and Applications,* Kluwer. Vol. 12, No. 1. January 2002. pp. 135-157.

[8] Wainer, G. “CD++: a toolkit to develop DEVS models.” *Software Practice and Experience.* (32), 1261-1306. 2002.

[9] Troccoli, A.; Wainer, G. “Performance results of parallel Cell-DEVS execution.” Proceedings of the Summer Computer Simulation Conference. Orlando, FL. USA. 2001.

[10] Wainer, G.; “Improved cellular models with parallel Cell-DEVS.” *Transactions of the SCS.* vol. 17 (2). June 2000.

[11] Martin, D.; McBrayer, T.; Wilsey, P. “WARPED: Time Warp Simulation Kernel for Analysis and Application Development.” Proceedings of the 29th Hawaii International Conference on System Sciences. 1996.

[12] Chow, A.C.; Kim, D.C.; Zeigler, B.P. “Abstract Simulator for the P-DEVS formalism.” AI, Simulation, and Planning in High Autonomy Systems. Gainesville, FL. USA. 1994.

[13] Glinsky, E.; Wainer, G. “Abstract simulation algorithms for Parallel CD++”. Technical Report SCE-05-11. Carleton University. Submitted for publication. 2005.