Performance Analysis of Cellular Models with Parallel Cell-DEVS

Alejandro Troccoli
Depto. de Computación, Universidad de Buenos Aires
P.B. Pabellón I. Ciudad Universitaria (1428)
Buenos Aires. Argentina.
atroccol@dc.uba.ar

Gabriel Wainer
SCE Dept., Carleton University
1125 Colonel By Drive
Ottawa, ON. K1S 5B6. Canada.
gwainer@sce.carleton.ca

Keywords: Discrete event simulation, Parallel DEVS, Parallel Cell-DEVS, distributed simulation, cellular models.

Abstract

Cell-DEVS is a formalism intended to describe cell shaped models. It defines cellular models with timing delay constructions, using simple definition of complex timing. The original specifications were recently extended to entitle parallel execution. A distributed mechanism allows the simulator to execute independently of the model specification. Here we present some implementation issues related with the definition of parallel simulators for Cell-DEVS.

INTRODUCTION

The **DEVS** formalism [1] provides a framework for the definition of hierarchical and modular models, allowing for model reuse and development time reduction. A DEVS model is seen as composed by atomic submodels than can be combined into coupled models. DEVS use a continuous time base, which allows accurate timing representation. **Cell-DEVS** [2] defines a way of describing n-dimensional cellular shaped models. Cell-DEVS can be combined with DEVS to model complex systems. A Cell-DEVS is a discrete event model using explicit delays for timing description.

The execution of complex models (such as the Cell-DEVS) usually requires a computing power that stand-alone computers do not provide, but that can be provided by parallel and distributed systems. However, the DEVS formalism posed some serialization constraints that made parallel execution inefficient. So it was revised and Parallel DEVS (P-DEVS) [3] was proposed. In [4] Parallel Cell-DEVS was introduced, conforming to the new Parallel DEVS formalism.

CD++ [5] is a toolkit developed for DEVS and Cell-DEVS modeling and simulation. The tool was modified to implement Parallel DEVS and Cell-DEVS models. To run Parallel Cell-DEVS models efficiently in a distributed environment, we implemented a modified version of the Parallel-DEVS abstract simulator [6]. The impact of the communication overhead was reduced by keeping the number of messages sent over the network to a minimum. This work analyzes the results obtained when running DEVS and Cell-DEVS models using the new parallel simulator.

BACKGROUND

P-DEVS preserves the basic modular and hierarchical structure defined for other DEVS models. The basic atomic component is defined as:

$$M = < X, S, Y, d_{int}, d_{ext}, d_{con}, l, ta >$$

X: a set of input events.
S: a set of sequential states.
Y: a set of output events.
d_{int}: $S \to S$: internal transition function.
d_{ext}: $Q \times X^b \to S$: external transition function,
X^b is a set of bags over elements in X,
$d_{ext}(s, e, \phi) = (s, e)$
d_{con}: $S \times X^b \to S$: confluent transition function.
$l : S \to Y^b$: output function.
$Ta : S \to R_{0 \to \infty}$: time advance function,
where $Q = \{ (s, e) \mid s \in S, 0 < e < ta(s)\}$
e is the elapsed time since last state transition.

Internal transitions execute at the next event time for all imminent components receiving no external events. Likewise, external events generated by these imminent trigger external transitions at receptive non-imminent (those components for which there are no internal transitions scheduled for the receiving time). However, for those components in which the internal and external transitions collide, the *confluent transition function* is employed instead of either the internal or external transition function to determine the new state [3].

Several atomic models can be put together to make a *coupled model*, which is defined by:

This work was partially supported by ANPCYT research project 11-04460 (Argentina) and NSERC (Canada).

$$DN = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} >$$

X : a set of input events.
Y : a set of output events.
D : a set of components.
for each i in D,
    M_i is a component.
for each i in $D \cup \{ self \}$, I_i is the influencees of i.
For each j in I_i,
    $Z_{i,j}$ is the i to j output translation function.

The structure is subject to the constraints that for each i in D, $M_i = < X_i, S_i, Y_i, d_{int\ i}, d_{ext\ i}, d_{con\ i}, l_i, ta_i >$ is *a P-DEVS*, I_i is a subset of $D \cup \{ self \}$, i is not in I_i , and
$Z_{self,j} : X_{self} \rightarrow X_j$
$Z_{i,self} : Y_i \rightarrow Y_{self}$
$Z_{i,j} : X_i \rightarrow Y_j$

Here *self* refers to the coupled model itself and is a device for allowing specification of external input and external output couplings.

In [2] the Cell-DEVS formalism was introduced. In traditional cellular models, every cell change occurs at the same time. Not only large amounts of compute time are required, but also the use of discrete time base pose restrictions in the precision of the model. The Timed Cell-DEVS formalism tries to solve these problems by using the DEVS paradigm to define a cell space where each cell is a DEVS atomic model. The goal is to build discrete event cell spaces, improving their definition by making the timing specification more expressive. In [4] it was extended to enable parallel execution of the models. A parallel Cell-DEVS atomic model can be formally defined as:

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

Two confluent functions have been added to the original Cell-DEVS definition: δ_{con} and τ_{con}. In addition, the external transition and output functions have been changed to handle input/output bags (X^b and Y^b) for each cell. The external transition function activates the local computation, whose result is delayed using one of both kinds of constructions: transport or inertial delays. The output function transmits the present values to other models. The confluent transition function δ_{con} is activated when there are collisions between internal and external events. It must activate the confluent local transition function τ_{con}, whose goal is to analyze the present values for the input bags and provide a unique set of input values for the cell. In this way, the cell will compute the next state by using the values chosen by the modeler.

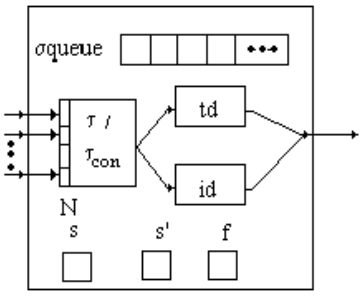


Figure 1. Informal definition of a cell [4].

Recently, a theory of quantized models was developed [7, 8]. The theory has been verified when applied to predictive quantization of arbitrary ordinary differential equation models. A curve is represented by the crossings of an equal spaced set of boundaries, separated by a *quantum* size. A *quantizer* checks for boundary crossings whenever a change in a model takes place. Only when such a crossing occurs, a new value is sent to the receiver. This operation reduces substantially the frequency of message updates, while potentially incurring into error.

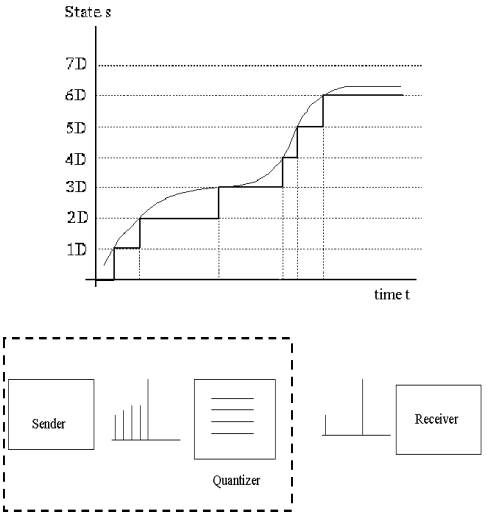


Figure 2. Quantization in DEVS models [7,8]

This theory of quantized models has been applied to Cell-DEVS. At any instant, a cell's state is within two different boundaries. When external events arrive and the local transition function computes the new cell state, the quantizer checks whether the new state falls within the same boundaries as the previous one. If it does not, then it is scheduled for output to neighbor cells.

ABSTRACT SIMULATOR FOR PARALLEL CELL-DEVS

We have built an abstract simulator for Parallel DEVS models, which entitles the execution of Parallel Cell-DEVS models. In Parallel DEVS, the simulation is executed by *Processors* that drive the simulation by exchanging messages. There are two types of *Processors*: *Simulators*, driving the simulation of atomic models, and *Coordinators*, in charge of executing coupled models and coordinating the activities of all their dependants. *Processors* are organized in a hierarchy resembling the model hierarchy.

Messages are built as pairs (type, time). There are different types of messages:

Synchronization messages:

(@, t)	*Collect message*
(, t)*	*Internal message*
($, t)	*Output Synchronization message*
(Done , t)	*Done message*

Content messages:

(q , t)	*External message*
(y, t)	*Output message*

A *Processor* is defined by describing a set of actions to be carried out upon the reception of each of these messages. A simulation cycle starts when the topmost *Coordinator* sends a *(@, t)* message. This message tells all the imminent *Processors* to execute their output functions and make the necessary translations of the resulting *(y, t)* messages to *(q, t)* messages that are sent to the model's influencees. When a *Processor* has finished sorting its outputs, it sends a *(done , t)* message to its parent *Coordinator*. After all the outputs have been processed, the topmost *Coordinator* sends a *(*, t)* message to trigger the execution of a model's transition function.

A *Simulator* receiving a *(*, t)* message will execute one of the three transition functions of its associated atomic model: d_{int}, d_{ext}, or d_{con} . If the model is imminent and has not received any external event, then d_{int} is executed. If the model is not imminent and has received external events, then d_{ext} is executed. Finally, if a model is imminent and received external events, d_{con} is executed, which will decide which of the external or internal transition function should be executed.

A *Coordinator* receiving a *(*, t)* message will forward this message to all its dependants that are either imminent or that have received external events.

CD++ [5] is a tool for executing Cell-DEVS models. This tool has been modified to conform to the Parallel Cell-DEVS formalism. The parallel simulator was designed as a layered architecture application. The topmost layer implements the Parallel DEVS abstract simulator, a middle layer carries out all required synchronization in the logical process level, and the lowest layer is in charge of communications.

The Warped kernel [9] was used as middleware. Warped provides an API for running parallel simulation. It currently supports two different kernels: a Time Warp kernel, which implements the Time Warp protocol, and a No Time kernel, which provides no synchronization. The parallel simulator has been written to support both kernels. It is currently being run with the No Time kernel because the application layer handles all synchronization.

In this environment, the models are divided into logical processes executing on different machines. Each logical process involved will host a subset of *Processors*. In particular, for Cell-DEVS models, each logical process will host the *Simulators* for a subset of cells. Under these assumptions, a *Coordinator's* children need not be executing on the same logical process. To reduce inter-process messages, coupled models will require a *Coordinator* on each logical process where a child *Processor* is running. One of these *Coordinators* will be known as a *Master Coordinator* and every other *Coordinator* will be a *Slave*. A *Master Coordinator* will handle all communication with its parent.

In Cell-DEVS models, output messages are sent every time a cell changes its state. The output is received as an external message by all neighbor cells. When a Cell-DEVS model is executed in parallel, the number of cells whose neighbors are running on a different logical process increases as more machines are added to the simulation. Since the number of output messages sent across the network to remote cells will increase as well, it is necessary to sort them efficiently. A distributed mechanism was chosen. With this mechanism, a *Slave* having outputs for a remote cell will send them directly to the corresponding *Slave*, without going through the *Master*. This same idea is applicable to Parallel DEVS models.

The distributed mechanism requires further synchronization to know when all output sorting has finished. This is achieved using the *($, t)* messages. Once a *Slave Coordinator* has sorted all outputs, it will send a *($, t)* message to all the other *Slaves*. After a *Slave* has received all *($, t)* and *(done , t)* messages, it will now be able to send a *(done, t)* message to the *Master Coordinator*. Upon receiving

a *(done, t)* from all the *Slaves*, the *Master* will know the output sorting phase has finished, and it will now be safe to send a *(*, t)* to execute the transition functions.

APPLICATION EXAMPLES

The parallel simulator has been tested with several models implemented using CD++. These models ran on a cluster of 12 workstations connected through an Ethernet hub.

The first results were obtained using an extended version of a Generator-Processor-Transducer model (GPT) [1]. The modified GPT model simulates a CPU receiving jobs and computes performance metrics (throughput and workload). It consists of a generator, a queue, a CPU and a transducer, as shown in Figure 3.

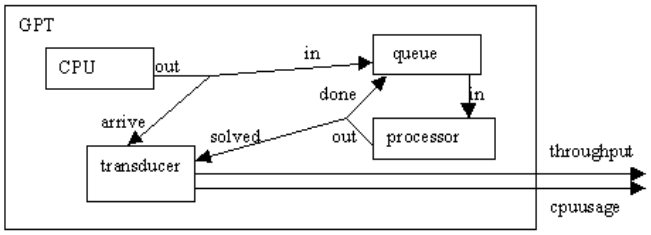


Figure 3. Coupling scheme for the GPT model

The generator outputs jobs periodically. When a new job is generated, its Id is sent to the queue and the transducer. If the queue is empty, the Id will directly be forwarded to the CPU; otherwise, it will be queued until the CPU is released. When the CPU finishes a job, it sends its Id to the transducer and the queue. If the queue has jobs waiting, it will send the next job to the CPU. The transducer will compute the turnaround time and update the throughput and CPU usage values, which it will output periodically.

The extended version of the GPT model consists of several copies of the GPT model just described with an increase in the workload of each component. Tests were executed using 12, 48 and 96 instances, running on 1 to 12 machines. The execution results are shown in Figure 4.

Figure 4 shows how the execution time changes with different partition sizes and different number of instances of the GPT. As more instances are used, a higher number of simulation objects are used and the simulation load is greater. As the load increases, so does the execution time. In addition, we can see that for the same load, as more machines are added to the simulation the execution time is reduced.

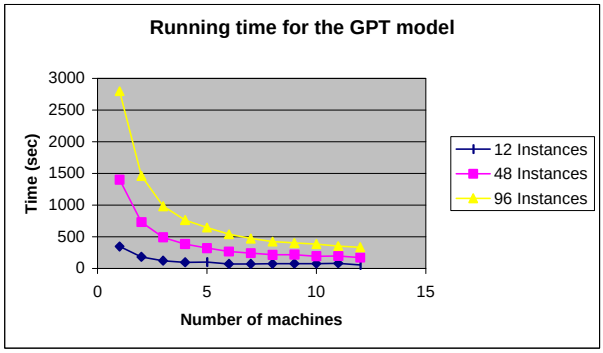


Figure 4. Execution times for the GPT model.

The performance of Parallel Cell-DEVS was tested using a heat diffusion model. In this model, a surface is represented by a 100 x 100 Cell-DEVS. Each cell contains a temperature, which is updated to the average of the values of the neighborhood whenever a new input arrives to the cell. In addition, heat and cold generators were connected to 90 different cells. Each time a generator produces an event, the temperature of a cell is updated.

Figure 5 shows part of the definition of the heat diffusion model using CD++.

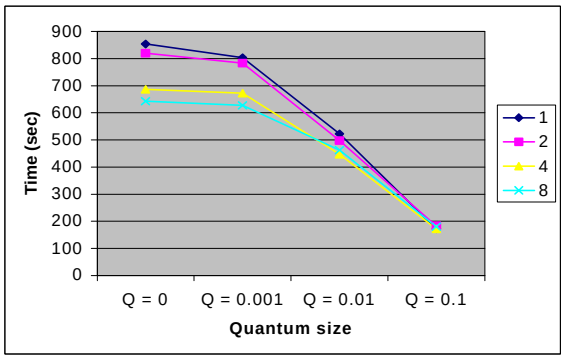
```
01   [top]
02   components : surface Heat@Generator Cold@generator
03   link : out@Heat inputHeat@surface
04   link : out@Cold inputCold@surface
05
06   [surface]
07   type : cell
08   width : 100
09   height : 100
10   delay : transport
11   defaultDelayTime  : 1000
12   border : wrapped
13   neighbors : surface(-1,-1) surface(-1,0) surface(-
1,1)
14   neighbors : surface(0,-1)  surface(0,0)  surface(0,1)
15   neighbors : surface(1,-1)  surface(1,0)  surface(1,1)
16   initialvalue : 24
17   in : inputHeat inputCold
18   link : inputHeat in@surface(25,25)
...
22   localtransition : heat-rule
23   portInTransition : in@surface(25,25)  setHeat
...

28   [heat-rule]
29   rule : { ((0,0) + (-1,-1) + (-1,0) + (-1,1) + (0,-1)
+ (0,1) + (1,-1) + (1,0) + (1,1)) / 9 } 10000 { t }
```

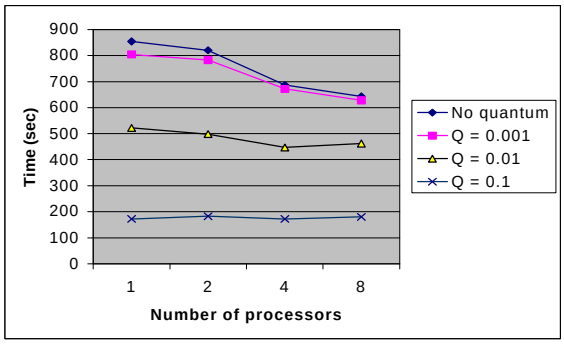
Figure 5. A heat diffusion model using CD++.

The model has three components: a surface, a heat generator and a cold generator (line 2). The output from the heat and cold generators is mapped to the inputHeat and inputCold ports of the surface (lines 3 and 4). Then, surface is defined as a wrapped cellular model of 100 x 100 cells with transport delays (lines 7 to 11). The neighborhood of a cell is set to the adjacent set of cells (lines 13 to 15). Line 18

connects the surface inputHeat port with the in port of cell (25,25). This line is repeated for all cells that will be connected to the inputHeat (not shown) and inputCold port. Line 22 sets the default transition function to heat-rule, which is defined in line 29. Finally, line 23 sets the transition function for events arriving through the *in* input port of cell (25,25) to setHeat.This model was evenly partitioned for 1, 2, 4 and 8 logical processes. The simulation was executed using a quantized simulator, whose results are shown in Figure 6.



(a)



(b)

Figure 6. Results of executing the heat diffusion model with different quantum values on 1, 2, 4 and 8 logical processes.

Figure 6(a) shows the effect of quantization on the execution time, which is reduced as the quantum size increases. This is due mainly a considerable reduction in the number of messages being sent. As discussed in [7] and [10], the use of a quantized model produces an error in the simulation results. Nevertheless, it was shown in [10] that the error magnitude for this model was bounded when a quantum size smaller than 0.1 was used. In this case, improvements in the execution time from 4 to 6 times were achieved.

Figure 6(b) shows how the execution time varies with different partition sizes. For the executions with quantum sizes 0 and 0.001, every time a machine is added to the simulation a reduction in the execution time is observed. However, this behavior is not observed for executions with quantum sizes 0.01 and 0.1. In this cases adding more machines does not necessarily produce a reduction in the execution time. The reason for this is that when a quantizer is used, the model activity is not evenly distributed among machines.

To assess how suitable a model is for parallel execution, a metric of model parallelism was developed. Basically, this metric will have a value of 1 when all processors involved share the same load (i.e. there is simultaneous execution). The value will be close to 0 if the simulation is completely executed in only one of the available processors.

One way to determine how much activity there is on each simulation cycle is to count the number of *(*, t)* messages received. If this information is obtained for each logical processor, a clear picture of how much activity is taking place can be drawn. Assuming all *(*, t)* messages take the same time to execute, then it can be determined how much time the simulation cycle will take, and for each machine, how busy it was during the simulation cycle. If all logical processes received the same number of *(*, t)* messages, then the load is evenly distributed.

Figure 7 shows the results of applying this metric to the non-quantized diffusion model, and using a quantum size of 0.1. If the load is evenly distributed, then a value of 1 is obtained. As the load distribution turns uneven, values get closer to 0.

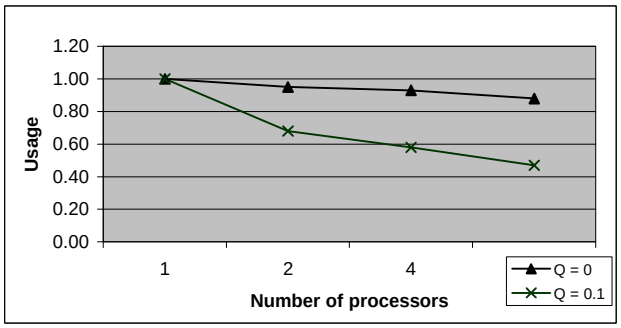


Figure 7. Parallelism for the heat diffusion model with and without quantum.

For the non-quantized version, the parallelism metric keeps very close to 1. However, there is a slight deviation from a perfect 1. This is because during the initial stages not every cell is active, and some partitions have more active cells than others do. On the contrary, when a quantizer with a quantum size of 0.1 is used, the metric's value falls below 0.6 as more machines are added. In this case, cells are not so

reactive to changes and reach a stable value quickly. Then, as changes propagate from one partition to the other, the workload shifts from processor to processor.

There is a relation between the metric's value and the performance observed. From Figures 6 and 7, it can be observed that the non-quantized version, which shows a value greater than 0.8 for the metric, experiences a performance gain when executed in parallel. The quantized version, which has a value for the metric that decreases as more processors are added to the simulation, experiences a slowdown.

CONCLUSION

We have presented performance results on the execution of a Parallel Cell-DEVS simulation algorithm. This abstract simulator entitled parallel execution of Parallel Cell-DEVS and DEVS models. The distributed mechanism allowed the simulator to execute independently of the model specification. During the development process, different simulation mechanisms were defined: some using the Warped environment, others based on the No-Time algorithm. We started using a Master/Slave coordination mechanism that was later changed to a distributed version. The abstract simulator runs originally in a centralized fashion. These results allowed us to check the feasibility of the approach, as no change was needed to any of the existing models.

We obtained substantial gains in the execution times. Quantization techniques produced even higher gains with a related the cost expressed as errors in the executed models. The introduction of a metric of parallelism in parallel DEVS models allowed us to analyze load-balancing issues that should be attacked in future works.

At present we are trying to reproduce the results here obtained in a multiprocessor architecture and a distributed architecture with a high speed switch, letting us to analyze the communication overhead involved. Different partitions of the existing models will be presented. The experimental results will be applied to a new range of models with the goal of classify them in order to obtain the highest speedups according to the application to be executed.

REFERENCES

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

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

[3] Chow, A., and Zeigler, B. 1994. "Parallel DEVS: A parallel, hierarchical, modular modeling formalism." In *Winter Simulation Conference Proceedings*. SCS, Orlando, Florida.

[4] Wainer, G. "Improved cellular models with parallel Cell-DEVS". *Transactions of the SCS.* June 2000.

[5] Rodriguez, D.; Wainer, G. 1999. "New Extensions to the CD++ tool." In *Proceedings of SCS Summer Multiconference on Computer Simulation*, Chicago, USA.

[6] Chow, A.; Kim, D.; Zeigler, B. 1994. "Abstract Simulator for the parallel DEVS formalism". *AI, Simulation, and Planning in High Autonomy Systems*, December.

[7] Zeigler, B. 1998. DEVS Theory of Quantization. DARPA Contract N6133997K-0007: ECE Dept., UA, Tucson, AZ.

[8] Zeigler, B.; Cho, H. ; Lee, J. and Sarjoughian, H. 1998. The DEVS/HLA Distributed Simulation Environment and its Support for Predictive Filtering. DARPA Contract N6133997K-0007: ECE Dept., UA, Tucson, AZ.

[9] Martin, D.; McBrayer, T.; Radhakrishnan, R.; Wilsey, P. 1997. "Time Warp Parallel Discrete Event Simulator". Technical Report. Computer Architecture Design Laboratory, University of Cincinnati.

[10] Wainer, G.; Zeigler, B. 2000. "Experimental results of Timed Cell-DEVS quantization". In *Proceedings of AIS 2000*, Tucson, AZ.

Alejandro Troccoli has received his M. Sc. (2001) from the Universidad de Buenos Aires, Argentina. He is a Teaching and Research Assistant in the same University, and a part-time consultant.

Gabriel Wainer received his M. Sc. (1993) and Ph.D. degree (1998) from the Universidad de Buenos Aires, Argentina, and Université d'Aix-Marseille III, France. He is currently Assistant Professor at the SCE Dept. of Carleton University (Ottawa, Canada). He is a member of the Board of Directors of the Society for Computer Simulation International, and a member of a group on standardization of DEVS modelling tools.