Implementing Parallel Cell-DEVS

Alejandro Troccoli

Departamento de Computación. Pabellón I. Ciudad Universitaria (1428). Buenos Aires. Argentina.

Gabriel Wainer

Department of Systems and Computing Engineering, Carleton University. 1125 Colonel By Dr. K1S 5BE. Ottawa, ON. Canada

gwainer@sce.carleton.ca

Abstract

Cell-DEVS is a formalism intended to model complex physical systems as cell spaces. Cell-DEVS allow describing cellular models using timing delay constructions, allowing simple definition of complex timing. The original specificationw sere extended to permit parallel specification of these models, and an associated simulation mechanism allows their execution. Here we present some implementation issues related with the definition of parallel simulators for Cell-DEVS.

1. Introduction

The **DEVS** formalism [1] provides a framework for the construction of hierachical models in a modular manner, allowing for model reuse and reducing development time and testing. The execution of complex models requires a computing power that stand alone computers do not provide. Therefore, the original DEVS formalism was revised and the Parallel DEVS (P-DEVS) [2] formalism was proposed. P-DEVS defines a function to handle transition collisions and eliminates the use of a sequential function to resolve simultaneous events. The revision eliminates all restrictions that forced the original DEVS definition to sequential execution.

A P-DEVS is composed by atomic models that can be coupled in a hierarchical and modular fashion. A P-DEVS atomic is defined as:

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta >$$

where

X: a set of input events.

S: a set of sequential states.

Y: a set of output events.

$\delta_{int}: S \rightarrow S$: internal transition function.

$\delta_{ext}: Q \times X^b \rightarrow S$: external transition function,

X^b is a set of bags over elements in X,

$\delta_{ext}(s, e, \phi) = (s, e)$

$\delta_{con}: S \times X^b \rightarrow S$: confluent transition function.

$\lambda : S \rightarrow Y^b$: output function.

$Ta : S \rightarrow R_{0 \rightarrow \infty}$: 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 imminents trigger external transitions at receptive non-imminents (those components for which there are no internal transitions scheduled for the receiving time). However, for those components 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 [2].

A *coupled model* is defined by:

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

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 componment.

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, $M_i = < X_i, S_i, Y_i, \delta_{int\,i}, \delta_{ext\,i}, \delta_{con\,i}, \lambda_i, ta_i >$ 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 [3] the Cell-DEVS formalism was introduced. When executing cellular models, large amounts of compute time are required, and the use of a discrete time base poses 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 defined as 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 revised to eliminate all the sequential restrictions the original formalism presented. 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 executes prior to the internal transition function, transmitting the present values to other models. The δ_{int} function is in charge of keeping the values for a transport delay. The following figure shows a sketch of the contents of each cell.

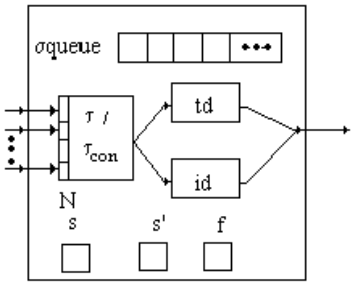

Figure 1. Cell's definition [4].

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 to 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.

The external transition function activates the local computation, whose result is delayed using one of both kinds of constructions. The output function, which executes prior to the internal transition function, is in charge to transmit the present values to other models.

In case of a collision, the confluent transition function chooses members from the bag, and updates the inputs for the cell. After, it deletes the unnecessary members of the bag. As $\sigma = 0$, an internal transition function is scheduled immediately. The modeler should define the behavior for the τ_{con} function in each cell, thus allowing the definition for this behavior under collisions.

DEVS separates the model from the actual simulation. The simulation mechanism is implemented by abstract simulators. In [5] an abstract simulator for the Parallel DEVS formalism was presented. Based on that work, we defined an abstract simulator for distributed simulation, which is the subject of this paper. In a distributed environment, there is considerable communications overhead which can not be ignored. Therefore, the abstract simulator should restrict the communications over the network to a minimum. The goal of this work is to present an abstract simulator developed to execute DEVS and Cell-DEVS models using standard tools for distributed and parallel programming. Several abstract simulators were implemented to allow parallel execution in the CD++ toolkit [9], entitling to have efficient execution of cellular models.

2. Parallel DEVS Abstract Simulators

As it was mentioned earlier, the modularity of the Parallel DEVS formalism makes it possible to separate the model from the simulation mechanism. The original abstract simulator mechanism [6] was revised to suit the Parallel DEVS formalism [2].

As in the existing definition of the abstract simulator [2], the DEVS processors will be specialized into two different simulation engines, *simulator* and *coordinator*. Basically, the role of the *simulator* is to invoke an atomic model transition and external event functions. On the other hand, a *coordinator* is attached to a coupled model and has the responsibility of translating its children' output events and of keeping the time of the next imminent/s dependants.

Every coordinator has a set of child DEVS processors. When a simulation run in distributed fashion, coordinator's children need not be executing on the same processor. If every coupled model is associated to only one coordinator, every message sent to child processors running on a different CPU will require interprocess communication. Figure 2(a) illustrates this case. It shows a coordinator sending a message to its 8 children distributed on two CPUs. Four interprocess messages are required for the four children running on processor 1.

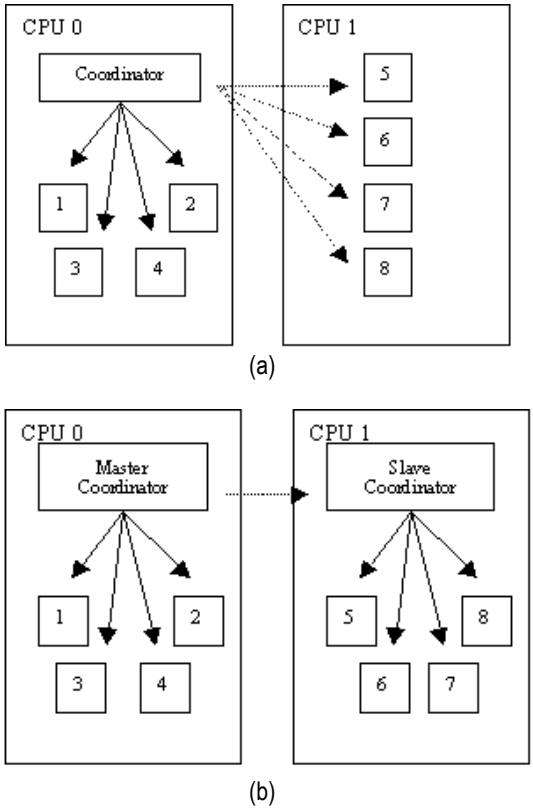

Figure 2. (a) A single coordinator sending a message to all its child processor. Dashed lines = interprocess messages. (b) A master- slave pair sending messages to all their children processors.

If the number of children processors is high (for instance, in coupled Cell-DEVS), the number of messages sent across the network will also be significant. This can be avoided if every coupled model have more than one coordinator. Figure 2(b) illustrates this case. For the same coupled model, there are two coordinators, one in processor 0 and another in processor 1. In this case, only one message is sent over the network.

For coupled models, coordinators will be required on each processor where a child processor is running. Children processors will send messages to the local coordinator, which will decide how to handle the received messages. Upon receiving a message from a child, a coordinator could forward this message to all the coordinators for the model. This would require all coordinators to know about the others. For instance, if coupled model *A* is a child of coupled model *B*, then *B´s* coordinators have to interact with *A´*s coordinators. If handled uncarefully, this communication can turn out producing the same number of interprocess messages we wanted to avoid. In such a scenario, a way of keeping the number of interprocess messages to a minimum is to have only one of the coordinators to handle all messages to the parent´s model local coordinator. This specialized coordinator will be known as a *master coordinator* and all other model coordinators will be *slaves*. The *master coordinator* for model *A* will then be the only one that can receive or send messages to *B´*s local coordinator.

With the exception of the top level DEVS processor, known as *root coordinator*, all DEVS processors will have a parent *coordinator*. To set the parent-child relationship on a distributed environment, the following rules apply,

a. for each *simulator*, the parent coordinator will be the parent's model local processor (it is guaranteed that this will exist)

b. for each *slave coordinator*, the parent coordinator will be the model's *master coordinator*.

c. For each *master coordinator*, the parent coordinator will be the parent's model local processor; just as if it were a *simulator*.

DEVS processors exchange messages which can be classified into two categories: synchronization messages and content messages. The synchronization messages are (*@* , *t*) and (*done*, *t*) and the contents messages (*y*, *t*) and (*q* , *t*). It is assumed that any two messages sent from the same source to the same destination will preserve their original ordering. The *P-DEVS* formalism states that all imminent model's output functions must be executed before any transition function. All outputs are collected and only after they have been sorted, the transition functions can be activated. These activities are co-ordinated using the synchronization messages.

We will now proceed to describe the abstract simulator mechanism for the *simulator*, *master coordinator*, *slave coordinator* and *root coordinator*.

The *simulator* attached to an atomic model has been implemented as in [2], with some minor changes:

when a (*@* , *t*) message is received
if $t = t_N$ **then**
  $y := \lambda(s)$
  send (*y* , *t*) to the parent coordinator
  send (*done*, *t*) to the parent coordinator
end if
else raise error
end when

when a (*q* , *t*) message is received
lock the *bag*
Add event *q* to the *bag*
unlock the *bag*
end when

when a (* , *t*) message is received
case $t_L \leq t < t_N$
  $e := t - t_L$
  $s := \delta_{ext}(s, e, bag)$
  empty *bag*

end case
case $t = t_N$ and *bag* is empty
$s := \delta_{int}(s)$
end case
case $t = t_N$ and *bag* not is empty
$s := \delta_{con}(s, bag)$
empty *bag*
end case
case $t > t_N$ or $t < t_L$
raise error
end case
$t_L := t$
$t_N := ta (s)$
send (*done*, t_N) to parent coordinator
end when

The implementation of a *master coordinator* is now given.

when a (@ , t) message is received from parent coordinator
if $t = t_N$ **then**
$t_L := t$
for all imminent child processors i with minimum t_N
send (@, t) to child i
cache i in the *synchronize* set
end for
wait until (*done*, t)'s have been received from all imminent processors
send (*done*, t) to parent coordinator
end if
else raise error
end when

For a *master coordinator* the set of child processors is made by the set of *slave coordinators*, the set of local child *simulators* and the set of child local *master coordinators.* A processor is local if it is executing on the same processor.

To simplify the next description it is necessary to define the function *coordinator*.

coordinator : $M \times P \rightarrow C$
where
M is a coupled model
P is a DEVS processor
S is a *coordinator (master or slave)*

coordinator (M, j) = i , where i is the *coordinator* associated to coupled M that is local to child j. The following restrictions apply for the function to be well defined:

j is a DEVS processor associated to a dependant of M

i is one of the *coordinators* associated with M

Now we can describe the behavior of a *master coordinator* upon receiving an output message. Two cases need to be distinguished:

an output message (y , t) received from a child i that is not a *slave coordinator*

an output message (y , i, t) forwarded from a *slave coordinator* that received (y , t) from a local child i.
when a (y , t) message is received from child i
for all influencees, j of child i
if j is a local processor
$q := z_{i,j}(y)$
send (q, t) to child j
cache j in the *synchronize* set
else
$s :=$ coordinator(*self*, j)
if $s \notin$ *slave-sync* set **then**
send (y, i, t) to s
cache s in the *slave-sync* set
cache s in the *synchronize* set
end if
end if
end for
if *self* $\in I_i$ (y is to be transmitted upward) **then**
$y := z_{i, self}(y)$
send (y, t) to parent coordinator
end if
clear *slave-sync* set
end when

when a (y , i, t) message is received from a slave s
cache s in the *slave-sync set* and proceed as if a (y , t) message had been received from child i
end when

Here *slave-sync* is used to avoid forwarding an output message twice to a *slave coordinator*. It is important to note that instead of forwarding a (q, t) message to a *slave coordinator*, a (y, i, t) is sent. This is done to reduce the number of messages sent across the network. A *slave coordinator* might be the parent coordinator for more than one of the influencees of i. If (q , t) messages were to be forwarded, then there will be one (q, t) message for each influencee of i. For Cell–DEVS models, this can be an important overhead. Instead, just one (y, i, t) message is sent across the network and it will be the responsibility of the *slave coordinator* to generate the appropiate (q, t) messages.

As mentioned in [2], all children ready for a transition are cached in a *synchronize* set to later distinguish active from inactive components.

when a (q , t) message is received from parent coordinator

lock the *bag*
Add event *q* to the *bag*
unlock the *bag*
end when

when a (* , *t*) message is received from parent coordinator
if $t_L \le t \le t_N$
for all $q \in bag$
for all receivers of *q*, $j \in I_{self}$
if *j* is a local processor
$q := z_{self,j}(q)$
send (*q*, *t*) to *j*
cache *j* in the synchronize set
else
s := coordinator(*self*, *j*)
if $s \notin$ *slave-sync* set **then**
send (*q* , *t*) to *s*
cache *s* in the *slave-sync* set
cache *s* in the *synchronize* set
end if
end if
end for
clear *slave-sync* set
end for
empty *bag*
for all *i* in the *synchronize* set
send (*, *t*) to *i*
end for
wait until all (*done*, t_N)'s are received
$t_L := t$
$t_N :=$ minimum of components' t_N's
clear the *synchronize* set
send (*done*, t_N) to parent coordinator
else raise an error
end when

When the output events are routed down to child processors, if the message is to be forwarded to a *slave coordinator* the *z* translation will not be applied. Instead, the original *q* message will be sent. Therefore, care must be taken not to forward a message twice to a *slave coordinator*. Here again, the *slave-sync* is used for that purpose.

The *slave coordinator* will be introduced next.
when a (@ , *t*) message is received from parent coordinator
if $t = t_N$ **then**
$t_L := t$
for all imminent child processors *i* with minimum t_N
send (@, *t*) to child *i*
cache *i* in the *synchronize* set
end for
wait until (*done*, *t*)'s have been received from all imminent processors
send (*done*, *t*) to parent coordinator
end if
else raise error
end when

As it can be noticed, there is no difference on how both *master* and *slave coordinators* handle a (@, *t*). However, the set of child processor of a *slave coordinator* is different. For a *slave coordinator* the set of child processors is made by the set of local child *simulators* and the set of local child *master coordinators*, only.

when a (*y* , *t*) message is received from child *i*
sent_to_master := false
for all influencees, *j* of child *i*
if *j* is a local processor
$q := z_{i,j}(y)$
send (*q*, *t*) to child *j*
cache *j* in the *synchronize* set
else
if not *sent_to_master*
send (*y*, *t*) to parent coordinator
sent_to_master := true
end if
end if
end for
if *self* $\in I_i$ (y is to be transmitted upward) **then**
if not *sent_to_master*
send (*y*, *t*) to parent coordinator
end if
end if
end when

when a (*y* , *i*, *t*) message is received from parent coordinator

sent_to_master := true
proceed as if a (*y* , *t*) message had been received from child *i*
end when

When an output event is received from a child *i*, the *slave coordinator* sorts the message to the influencees of *i*. If any influencee is local, the *z* function is applied a (*q* , *t*) message is sent. If there are non-local influencees, then the output event is sent to the *master coordinator*, who will then sort the message to other *slave coordinators* if necessary. Only one (*y* , *t*) message should be forwarded to the *master coordinator*.

When the *slave coordinator* receives an output event that has been forwarded by the *master coordinator* on behalf of child *i*, it will handle the event as if *i* had been local, but no (*y*, *t*) messages will be forwarded back to the *master coordinator* if there is a non-local influencee. This

is to avoid infinite loops of messages being forwarded back and forth.

when (*q*,*t*) message is received from parent coordinator
lock the *bag*
Add event *q* to the *bag*
unlock the *bag*
end when

when (*,*t*) message is received from parent coordinator
if $t_L \le t \le t_N$
for all $q \in bag$
for all receivers of *q*, $j \in I_{self}$
if *j* is a local processor
$q := z_{self,j}(q)$
send (*q*, *t*) to *j*
cache *j* in the synchronize set
else
do nothing
end if
end for
end for
empty *bag*
for all *i* in the *synchronize* set
send (*, *t*) to *i*
end for
wait until all (*done*, t_N)'s are received
$t_L := t$
$t_N :=$ minimum of components' t_N's
clear the *synchronize* set
send (*done*, t_N) to parent coordinator
else raise an error
end when

The root coordinator is a special processor that is above the topmost coordinator. It is responsible for driving the simulation and advancing the virtual simulation time. Our root coordinator can also handle external events which are stored in a sorted queue of events.

Root coordinator

load *queue* of external events and sort them by arrival time.

t := minimum of t_N of topmost coordinator and t_N of *queue*.
while $t \neq \infty$
if $t = t_N$ of *queue*
for all *q* in *queue* with time *t*
send (*q* , *t*) to topmost coordinator
end for
end if

if $t = t_N$ of topmost coordinator
send (*@*, *t*) to topmost coordinator
wait until (*done*, *t*) is received from it
end if

send (*, *t*) to topmost coordinator
wait until (*done*, *t*) is received from it

end while
raise simulation completed

This abstract simulator mechanism will be able to handle both, Parallel DEVS and Parallel Cell-DEVS models because the latter one is a specialization of the first one.

3. Parallel CD++

CD++ [7] is a modeling tool for the simulation of DEVS and Cell-DEVS models. This tool has been extended into Parallel CD++ (*PCD++*), a tool for the simulation of Parallel DEVS and Parallel Cell-DEVS models on a distributed environment.

PCD++ has been built on top of a modified version of Warped [8]. All DEVS processors have been defined as Warped objects. Warped defines a simulation API and provides a set of different simulation kernels: a sequential kernel for the execution of models in standalone mode, a TimeWarp kernel for parallel execution using optimistic synchronization mechanisms and a NoTime kernel, for parallel and standalone simulation that uses no synchronization at all. In addition, we have developed a kernel that uses pessimistic synchronization mechanisms. For the parallel kernels, Warped uses MPI for communication between CPUs. The current *PCD++* has been succesfully tested with the NoTime kernel.

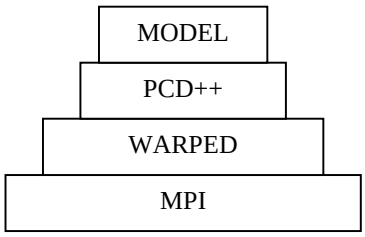

Figure 3. PCD++ layered architecture

In the abstract simulator mechanism that we presented for distributed environments, the time advance is controlled by the root coordinator. Therefore, no synchronization is required because no processor will execute an out of order event. The NoTime kernel is very well suited for this case because it provides the necessary communication primitives and avoids the overhead of TimeWarp. Figure 4 shows the Warped API.

```
class TimeWarp {
// Methods the user defines
virtual void initialize();
virtual void finalize();
virtual void executeProcess();
BasicState* allocateState();

//Simulation kernel services
void sendEvent (BasicEvent * );
BasicEvent* getEvent();
};

class BasicEvent {
int size;
Vtime sendTime;
Vtime recvTime;

int sender;
int dest;
}

class BasicState {
BasicState* copyState( BasicState*);
}
```

Figure 4. Warped API

To define new atomic models, PCD++ provides an abstract class *Atomic* that the modeler has to extend using inheritance. Coupled models, need no programming. Instead, they are defined writing a model file using a specification language PCD++ provides for that purpose. This specification language is also used for the definition of Cell-DEVS models.

```
class Atomic {
// Methods the user should def
Model& internalFunction();
Model& externalFunction (MessageBag&)
Model& outputFunction();
Model& confluentFunction();
ModelState* allocateState();

//Simulation kernel services
void sendOutput ( Port&, BasicMsgValue* );
const Vtime& lastChange();
void holdIn( state, Vtime );
};
```

Figure 5. The Atomic class

Finally, having defined the model and the set of available machines, it only remains to define how the models will be distributed. The modeler has to create a partition file that tells PCD++ which machine each atomic model should run on. This tells PCD++ where each *simulator* should be placed. The location of the *coordinators* is decided by PCD++.

4. A heat diffusion model

PCD++ has been used to simulate a heat diffusion model. A surface is represented by a 50 x 50 cellular automaton, each cell containing a temperature. In each simulation cycle, the temperature of the cell is updated to the average of the values of the neighborhood. In addition, a heat generator is connected to the cells (25, 25) and (10, 10), generating temperatures in the range [24, 40] with uniform distribution. Also, a cold generator that creates temperatures in the range [10, 15] with uniform distribution, has been connected to the cells (10, 40) and (40, 40). Both generators create values after *x* seconds, where *x* follows an exponential distribution with mean 50 seconds. When any of the generators outputs a new value, the cell to which it is connected will take that value.

The definition of the model using the language provided by the tool is showed in Figure 6. The top model and its components are defined between lines 1 and 4. Between lines 6 and 26, the model representing the surface is defined. It is composed of a cellular automata of 50x50 cells with an initial temperature of 24° C. In the lines 28 and 29 the local transition function is defined.

Lines 31 and 32 define the transition function upon receiving an external event from the heat generator, and lines 34 and 35 for transition triggered by external events coming from the cold generator. Lines 37 to 47 define the distribution parameters for the generators.

```
01  [top]
02  components : surface generatorHeat@Generator
                 generatorCold@generator
03  link : out@generatorHeat inputHeat@surface
04  link : out@generatorCold inputCold@surface
05
06  [surface]
07  type : cell
08  width : 50
09  height : 50
10  delay : transport
11  defaultDelayTime  : 100
12  border : wrapped
13  neighbors : (-1,-1) (-1,0) (-1,1)
14  neighbors : (0,-1)  (0,0)  (0,1)
15  neighbors : (1,-1)  (1,0)  (1,1)
16  initialvalue : 24
17  in : inputHeat inputCold
18  link : inputHeat in@surface(25,25)
19  link : inputHeat in@surface(10,10)
20  link : inputCold in@surface(40,40)
21  link : inputCold in@surface(10,40)
22  localtransition : heat-rule
23  portInTransition : in@surface(25,25)  setHeat
24  portInTransition : in@surface(10,10)  setHeat
25  portInTransition : in@surface(40,40)  setCold
26  portInTransition : in@surface(10,40)  setCold
27
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 }
30
31  [setHeat]
32  rule : { uniform(24,40) } 1000 { t }
33
34  [setCold]
35  rule : { uniform(-10,15) } 1000 { t }
36
37  [generatorHeat]
38  distribution : exponential
39  mean : 50
40  initial : 1
41  increment : 0
42
43  [generatorCold]
44  distribution : exponential
45  mean : 50
46  initial : 1
47  increment : 0
```

Figure 6. Definition of the heat diffusion model

The model has been simulated on a 12 PC network running Linux. Different tests were done, each with a different model partition.

```
01   0 : generatorHeat generatorCold
02   0 : surface(0,0)..(24,24)
03   1 : surface(25,0)..(49,24)
04   2 : surface(0,25)..(24,49)
05   3 : surface(25,25)..(49,49)
```

Figure 7. Model partition for 4 processors.

Figure 7 shows a model partition for running the heat diffusion model on 4 machines. There are a total of 252 simulators that have to be assigned to 4 CPUs. Line 1 defines the location for the simulators associated to the generatorHeat and generatorCold atomic models. Lines 2 to 5 set where the simulators for the cells of the surface model will be running.

In addition, there are two coupled models: the top model and the surface model. For the surface model, PCD++ will create four coordinators: a *master coordinator* running on processor 0 and three *slave coordinators*, each running in one of the CPUs 1 to 3. For the top model, there will only be one *master coordinator* on processor 0.

The results of running the simulation on 1, 2, 4 and 8 processors are shown below. For this test, the simulation was configured to use the NoTimekernel.

processors	Time (sec)
1	590
2	476
4	383
8	369

Figure 8. Simulation execution time

As it can be appreciated, there is a significant reduction in the simulation time as more processors are used. The speedups are not exponential, but they add up to the performance provided by Cell-DEVS. The following figure shows the execution time of Cell-DEVS models (ACA) against traditional Cellular Automata for this particular model.

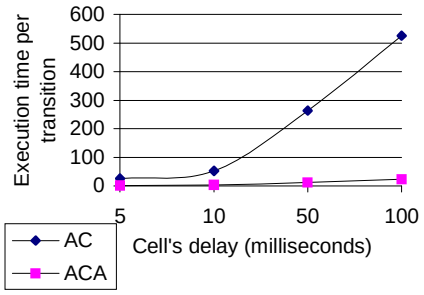

Figure 9. Simulation execution times of Cell-DEVS models

5. Conclusion

CD++ is a tool for the simulation of Parallel DEVS and Cell-DEVS models that implements this distributed abstract simulator mechanism. The tool has proven to reduce the execution time models with a high number of simultaneous events.

Distributed environments have a communications overhead that can be quite significant. The extension of the Parallel-DEVS abstract simulator here presented keeps to a minimum the number of messages sent across machines. This was possible by assigning each coupled model one *master coordinator* and zero, one or more *slave coordinators*. Messages that have to cross a processor boundary are always sent between *master* and *slave coordinators*, which then forward the received messages to their local dependants.

A new abstract simulator that will allow for out of order execution of events is being studied. For this new mechanism the Warped TimeWarp kernel will be used.

References

[1] ZEIGLER, B.; KIM, T.; PRAEHOFER, H. "Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems". Academic Press. 2000.

[2] ALEX C. CHOW; BERNARD P. ZEIGLER. Parallel DEVS: A parallel, hierarchical, modular modeling formalism. In *Winter Simulation Conference Proceedings*, Orlando, Florida, 1994. SCS.

[3] WAINER, G.; GIAMBIASI, N. "Timed Cell-DEVS: modelling and simulation of cell spaces ". In "Discrete Event Modeling & Simulation: Enabling Future Technologies", to be published by Springer-Verlag. 2001.

[4] WAINER, G. “Improved cellular models with parallel Cell-DEVS”. In *Transactions of the SCS*. June 2000.

[5] ALEX C. CHOW, DOO H. KIM; BERNARD P. ZEIGLER. "Abstract Simulator for the parallel DEVS formalism". *AI, Simulation, and Planning in High Autonomy Systems.* Dec., 1994

[6] BERNARD P. ZEIGLER. *Object Oriented Simulation with Hierarchical, Modular Models.* Academic Press, San Diego, California, 1990.

[7] RODRIGUEZ, D.; WAINER, G. "New Extensions to the CD++ tool". In *Proceedings of SCS Summer Multiconference on Computer Simulation*. 1999.

[8] MARTIN, D.; MCBRAYER, T.; RADHAKRISHNAN, R.; WILSEY, P. "TimeWarp Parallel Discrete Event Simulator". *Technical Report. Computer Architecture Design Laboratory, University of Cincinnati*. December 1997.