A Parallel Algorithm to Accelerate DEVS Simulations in Shared Memory Architectures

Guillermo German Trabes, Gabriel A. Wainer, *Senior Member, IEEE*, and Veronica Gil-Costa

***Abstract*—We propose a new algorithm for the execution of Discrete Event System Specification (DEVS) simulations on parallel shared memory architectures. Our approach executes parallel discrete-event simulations by executing all tasks in the PDEVS simulation protocol in parallel. The algorithm works by distributing the computations among different cores on shared memory architectures. To show the benefits of our algorithm, we present the results of a set of experiments using a synthetic benchmark and a real-world scenario using two independent computer architectures. The results obtained show how our algorithm accelerates simulations up to eight times, improving previous approaches. In addition, we show that our approach scales when we increase the number of CPU-cores used.**

***Index Terms*—Discrete-event, parallel algorithms, simulation, shared memory.**

I. Introduction

MODELING and simulation (M&S) became an essential tool to represent problems in various disciplines and conduct scientific explorations. Although many M&S methodologies have been developed, discrete-event simulation (in which system behavior is modeled as a chronology of events in continuous time) is now widely used to study a wide variety of problems [1].

Formal M&S methodologies allow defining the models precisely, to conduct formal checks and to build simulators that are easier to verify. In particular, the Discrete Event System Specification (DEVS) formalism [2] has provided a formal theoretical framework for the development of discrete-event M&S. DEVS has been used in numerous application areas since its inception. The formal definition of DEVS offers the ability to separate the definition, implementation, and experimentation of the models. Models are defined using a formal notation and then they are simulated using algorithms that have been formally verified. This separation of concerns facilitates reuse and improved the verification of models [2].

There has been a growing demand to simulate complex DEVS models in different fields of study, leading to increasing execution times. To tackle this problem, there have been multiple attempts to achieve parallel executions of DEVS using parallel discrete-event simulation (PDES) approaches [3], [4], [5], [6]. However, in practice, these algorithms result in complex simulation architectures, and they give rise to various issues related to zero lookahead loops and correctness [4].

A different approach to accelerate DEVS simulations was proposed in [7], which allows the execution of the simulation protocol in parallel. The main idea behind this approach is to allow a simple and error-free algorithm by executing simultaneous events in parallel. In this approach, the authors identified two situations where the simulation protocol can execute in parallel: 1) when two or more components execute their output functions; and 2) when two or more components execute their state transition functions. However, this approach showed limited acceleration in real-world applications [8].

Here we define an algorithm that provides additional parallel execution opportunities. In our approach, the complete algorithm executes in parallel, obtaining the same results as in the sequential version in all scenarios. The algorithm is intended to execute in shared-memory computer architectures, which can use systems resources efficient and have low communication latency compared to distributed memory systems.

The idea is to decompose tasks and assigning them to a group of threads that accelerates the execution. After each task is completed, the threads synchronize to ensure the completion of a task before starting a new one. To test our approach, we present an experimental evaluation to show how this algorithm can accelerate the execution of DEVS models, using a benchmark designed for Cellular DEVS models [9] and a real-world case study. Our real-world case study is an epidemiological model that simulates how diseases spread among populations. The results presented show how we can accelerate up to eight times and that our algorithm scales when we increase the number of CPU-cores used. Finally, we show how our proposed approach improves performance compared with the previous approach presented in the literature.

The rest of the paper is organized as follows. Section II provides a literature review on the execution of DEVS simulations and parallel programming for shared memory computers. On Section III we present our proposed parallel algorithm. Next, on Section IV we show an experimental evaluation for our

Manuscript received 30 August 2022; revised 28 January 2023; accepted 2 March 2023. Date of publication 14 March 2023; date of current version 31 March 2023. This work was supported by NSERC Canada. Recommended for acceptance by S. Chandrasekaran. *(Corresponding author: Guillermo German Trabes.)*

Guillermo German Trabes is with the Department of Systems and Computing Engineering, Carleton University, Ottawa, ON K1S 5B6, Canada, and also with the Universidad Nacional de San Luis, San Luis D5700, Argentina (e-mail: guillermotrabes@sce.carleton.ca).

Gabriel A. Wainer is with the Department of Systems and Computer Engineering, Carleton University, Ottawa, ON K1S 5B6, Canada (e-mail: gwainer@sce.carleton.ca).

Veronica Gil-Costa is with the National University of San Luis and CCT CONICET, San Luis D5700, Argentina (e-mail: ggvcosta@gmail.com).

This article has supplementary downloadable material available at https://doi.org/10.1109/TPDS.2023.3256083, provided by the authors.

Digital Object Identifier 10.1109/TPDS.2023.3256083

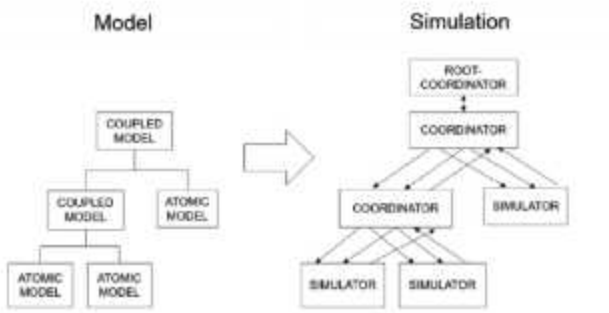

Fig. 1. PDEVS model example with its corresponding PDEVS abstract simulator.

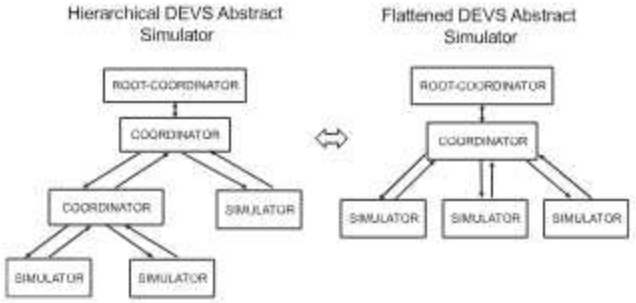

Fig. 2. Hierarchical DEVS simulator and its corresponding flattened DEVS simulator.

algorithm. Finally, on Section V we present the conclusions and future work.

II. Background

Following the ideas from classical DEVS [2], PDEVS [10] makes a clear separation between model and simulation. Users only need to follow the specifications provided by the formalism to define their models. Then, to execute simulations, there exists a general mechanism. This mechanism is based on a data structure known as the PDEVS abstract simulator [11]. Given a PDEVS model, the PDEVS abstract simulator creates a data structure to mimic its behavior and execute simulations. The structure consists of three types of components: simulators, coordinators, and root-coordinator. Each atomic model is associated with a simulator and each coupled model is associated with a coordinator. One root-coordinator is placed at the root of the structure hierarchy. We can see an example of the PDEVS abstract simulator in Fig. 1.

Once the abstract simulation structure is created, there is a mechanism to execute simulations, known as the PDEVS simulation protocol [11]. This algorithm works by exchanging several messages between the components between "Processors", which can be classified into Coordinators, Simulators, and Root-Coordinator. There are messages for initialize a model (i), execute an output function (@), execute state transitions (∗), send outputs (y), add event to a bag (q), and to schedule the next internal event (done). At the top of this hierarchy, the Root-coordinator starts a sequence of simulation steps by communicating to the top-most Coordinator. Coordinators at the different levels interchange messages with their parent Coordinator and with the Processors one level below. The messages disseminate through the tree structure levels by coordinators until reaching Simulators at the leaves of the tree. The details for this algorithm can be found in [11]. This protocol was proven to be correct, and it was widely accepted and implemented in many simulators.

One problem in the execution of the PDEVS simulation protocol above is that the hierarchical message passing across several levels on the tree might involve a large overhead, increasing execution time in the coordination of components at the different levels. To improve this hierarchical communication overhead, a flattening algorithm was proposed in the DEVS literature [12], [13]. The idea of this algorithm is to transform the hierarchical PDEVS structures into flat data structures where all intermediate coordinators are eliminated. The resulting structure has only one coordinator that holds all the information about couplings among simulators. We can see an example of this transformation in Fig. 2.

Every flat abstract PDEVS simulator has the same basic structure. One root-coordinator oversees the simulation, advances global simulation time, and determines when the simulation ends. Below the root-coordinator there is a flat coordinator. This component is responsible for the message interchange among simulators, triggers the computation of events on the simulators, and determines the minimum time for the next events on all simulators. Finally, at the leaves of the tree there are |D| components, where D is the set of atomic models. These components are responsible to execute the behavior of the atomic models defined by the modelers. They execute output, transition, and time advance functions. In addition, they reply to their parent component when they receive messages.

There are several reasons to use a flattened PDEVS abstract simulator approach. In first place, the number of messages transmitted between components is reduced by eliminating all intermediate levels in the tree hierarchy. Second, the execution of the PDEVS simulation protocol on this structure is simpler, and therefore easier to understand, analyze and implement [14], [15]. In third place, there is empirical evidence showing the benefits of applying this method and how this can accelerate simulation's executions in practical implementations [13], [16], [17], [18]. Finally, since any hierarchical PDEVS abstract simulator's structure can be transformed into an equivalent flat PDEVS abstract simulator, the improvements made on the execution of flat PDEVS simulations can be used to improve performance on any PDEVS simulation.

A. Parallel Discrete Event Simulation (PDES)

In addition to the techniques to reduce the overhead on the hierarchical structure, several techniques have been proposed to parallelize DEVS simulations. The main techniques applied were adapted from the Parallel Discrete Event Simulation (PDES) field. The reader can find the main efforts on PDES in the Proceedings of the PADS conferences between 1990 and 2019, and an introduction to this field can be found in [19].

The most important method in the PDES field consists of the concept of Logical Processes (LPs). LPs work as simulation entities, sharing no state variables, and interacting with each other through time-stamped event messages. The biggest challenge in PDES is being able to produce the same results as in a sequential execution. Synchronization between LPs is violated when one of the LPs receives an event that is older than the current clock of the receiving LP. Such a violation is known as a causality error. To deal with causality errors, different synchronization techniques have been proposed [19], [20]. The techniques for the synchronization of LPs can be classified in four categories.

The first category is called conservative synchronization [21]. These algorithms strictly avoid any occurrence of causality errors. To do so, the LPs are blocked from further processing of events until it can make sure they are safe from future events arrivals from other LPs with smaller timestamps. The basic problem for a conservative parallel simulator is how to determine if it is safe for a processor to execute events. To deal with this issue, several techniques were proposed which can be further classified into four categories [5]: methods with deadlock avoidance, deadlock detection and recovery, synchronous operation, and conservative time windows. The main drawback in this technique is that to guarantee strict causality error avoidance, very complex synchronization mechanisms must be executed.

The second category is called optimistic synchronization [21]. These algorithms are based on the detection of straggler events. The stragglers events are those whose timestamp is less than the current time of the LP. When a straggler is detected, a rollback mechanism corrects the state of the simulation by resetting it to a previous time. The most famous and widely used optimistic algorithm is called Time Warp and was applied to DEVS simulations [3], [6].

There exists a third category called approximated synchronization algorithms [22], which are a special version of the optimistic algorithms. In this technique, the occurrence of straggler events is ignored, and the LPs continue with their execution. This way, the execution time is accelerated but the precision in the results of the simulation is sacrificed. This technique was applied to DEVS simulations [23], [24]. The drawback of this approach is that the errors in the results are highly dependent on the model simulated.

Finally, there exists a more recently proposed technique called hybrid PDES [25]. Hybrid PDES dynamically switches between conservative and optimistic synchronization protocols based on the simulation runtime characteristics. The advantage of this technique is that it can take advantage of the best characteristics of both synchronization techniques. However, the disadvantage in is that methods to measure runtime variables must be deployed to determine if the synchronization protocol must be switched.

B. Parallel Events in the PDEVS Simulation Protocol

The problem with the LPs approaches is that only some conservative techniques have been proven to exactly represent the behavior of the DEVS sequential simulator [7], for example the ones presented in [26], [27]. This feature distinguishes DEVS from the many other simulation engines derived from the LP approaches, the behavior of the sequential DEVS simulator cannot be reproduced in all scenarios.

A different approach proposed in [7] enables the execution of simultaneous events in parallel. The main idea behind this approach is to execute in parallel the independent simultaneous events occurring in the PDEVS simulation protocol. The idea is to apply a parallel execution in two situations:

- When multiple output functions must be computed at the same time.
- When multiple transition functions must be computed at the same time.

Furthermore, in [7] a theoretical analysis was made with this idea, concluding that speedups are related to specific model characteristics, such as the level of activity, defined as the probability of execute a transition function on each component. This theoretical analysis concludes that this protocol can achieve an execution time 60% higher on average than the best possible parallel execution, using this simple parallel implementation. This idea was used in [8], where the authors show a practical implementation in a simulator. However, the speedup obtained was limited and did not scale when increasing the number of threads. In this work we propose to improve this idea by defining an algorithm that introduces additional parallelism.

III. Related Work

A. Cadmium Simulator

The software we use as a base implementation is the Cadmium simulator [28], which is an open-source simulator written in C++. Cadmium was designed to execute in a sequential algorithm the PDEVS simulation protocol [29] and therefore guaranteeing correct simulation results. Cadmium uses typed messages and typed ports, a time representation independent of the model implementation and it includes automated checking of some properties of the DEVS models, for early error detection. Cadmium implements the PDEVS simulation protocol and therefore guarantees correct simulation results.

As in many DEVS simulation tools, Cadmium was developed in a way that the user only defines the models and does not need to know the mechanism to execute simulations. The simulations are executed by invoking a ROOT-COORDINATOR component, which creates a SIMULATOR for each atomic model and COORDINATOR for each coupled model. SIMULATOR components manage the state of the atomic models and execute their functions, and COORDINATOR handle intercommunication and synchronization between subcomponents, which can be COORDINATORs or SIMULATORS. We can see an example of Cadmium's architecture in Fig. 3.

The main distinctive Cadmium's feature is that all message passing is performed by functions call and returns. This way, Cadmium reduces the overhead caused by message passing and synchronization between components. This design was proposed to guarantee predictable execution traces and to achieve efficient performance, with the goal of make it suitable for real-time execution.

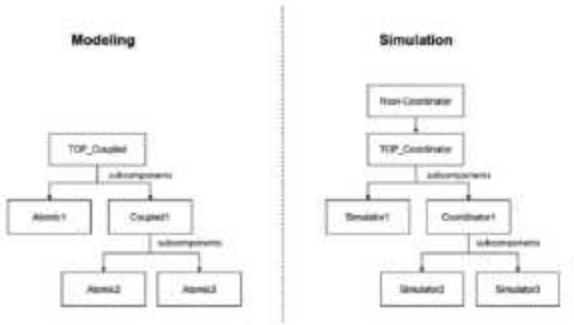

Fig. 3. Cadmium's architecture example.

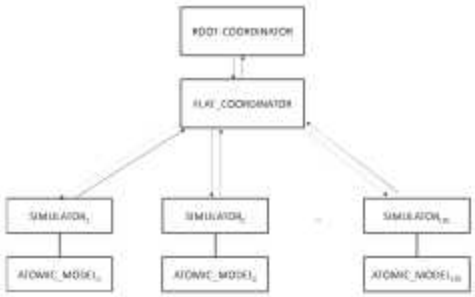

Fig. 4. Flattened PDEVS abstract simulator.

```
1: ROOT-COORDINATOR {
2:  variables: FLAT-COORDINATOR, next_time;
3:  execute_simulation(initial_time, simulation_time){
4:    FLAT-COORDINATOR.initialize(initial_time);
5:    next_time = FLAT-COORDINATOR.next_time();
6:    while(next_time<simulation_time) {
7:      FLAT-COORDINATOR.execute_output_functions(next_time);
8:      FLAT-COORDINATOR.route_messages();
9:      FLAT-COORDINATOR.execute_state_transitions(next_time);
10:     next_time = FLAT-COORDINATOR.next_time();
11:   } // end while
12: } // end execute-simulation
13: } // end ROOT-COORDINATOR
```

Fig. 5. Pseudocode for ROOT-COORDINATOR component.

In addition, Cadmium was extended to execute flattened PDEVS simulations efficiently. For this, a FLAT-COORDINATOR component was added. With this component, the simulation algorithm is simplified and there is no need to call recursive functions along the simulation structure. This algorithm is implemented in three types of components: ROOT-COORDINATOR, FLAT_COORDINATOR and SIMULATOR. We can see a graphical representation for the structure used in this algorithm in Fig. 4.

In Fig. 5 we can see a definition for ROOT-COORDINATOR. We use a FLAT-COORDINATOR component and a variable to store the next simulation time. Simulations begin with the execute_simulation function (with initial time and simulation time limit). Functions initialize and next_time only execute at the beginning of the simulation. The initialize function executes the time advance function on all SIMULATORs. Then, next_time determines the minimum of all those times.

```
1: FLAT-COORDINATOR {
2:  variables: subcomponents, internal_couplings, next_time;
3:  initialize(initial_time){
4:   for each SIMULATOR in subcomponents:
5:    SIMULATOR.initialize(initial_time);
6:   end-for
7:  }//end initialize;
8:  execute_output_functions(next_time){
9:   for each SIMULATOR in subcomponents:
10:   SIMULATOR.execute_output_function(next_time);
11:  end-for
12: }//end execute_output_functions;
13: route_messages(){
14:  for each coupling in internal_couplings:
15:   outbox = coupling.origin.get_outbox();
16:   if(coupling.origin.get_outbox() != empty)
17:    coupling.destiny.add_to_inbox(outbox);
18:   end-if
19:  end-for
20: } //end route_messages
21: execute_state_transitions(next_time){
22:  for each SIMULATOR in subcomponents:
23:   SIMULATOR.execute_state_transition(next_time);
24:  end-for
25: }//end execute_state_transitions
26: next_time(){
27:  next_time = subcomponents.first.next_time();
28:  for each SIMULATOR in subcomponents:
29:   if(SIMULATOR.next_time() < next_time)
30:    next_time = SIMULATOR.next_time();
31:   end-if
32:  end-for
33:  return next_time;
34: }//end next_time
35: }//end FLAT-COORDINATOR
```

Fig. 6. Pseudocode for FLAT-COORDINATOR component.

After these two tasks, a simulation loop starts, executing a finite sequence of simulation cycles until the time limit for the simulation is reached or there are no more next events in the simulation. Each cycle is composed by four tasks. First, we execute the output functions on the imminent SIMULATORs (i.e., those with an event scheduled on the next simulated time). Next, we call the route_messages function on FLAT-COORDINATOR, which transmits the results from the output functions computed on the previous tasks to the receiver components (using the couplings defined by the user). Following, we call the execute_state_transitions function to update the state on those components that are imminent and/or receivers. As mentioned, the imminent components are those whose time for the next event is equal to the next simulation time. The receiver components are those who received a message from an imminent component in the previous task. Finally, we call to next_time to determine the time for the next events by comparing the next times scheduled in all SIMULATORs.

The next component we need to execute this algorithm is a FLAT-COORDINATOR. We can see the pseudocode for this component in Fig. 6.

FLAT-COORDINATOR uses a set of subcomponents, which are all SIMULATORs; the internal couplings according to the model's specification; and a variable to store the next time. In addition, FLAT-COORDINATOR has five functions.

The initialize function takes the initial time for the simulation as parameter. This function calls the initialize function on each SIMULATOR subcomponent with the initial time as parameter. The execute_output_functions routine calls the

```
1: SIMULATOR {
2:  variables: inbox, outbox, last, next, model;
3:  initialize(t) {
4:   last = t;
5:   next = t + model.time_advance();
6:  } //end initialize
7:  execute_output_function(t) {
8:   if (next==t) {
9:    outbox = model.output_function();
10:   }
11:  } // end execute_output_function
12:  state_transition(t) {
13:   if (t == next) {
14:    if (inbox == empty) {
15:     model.internal_transition();
16:    } else {
17:     model.confluent_transition(t - last, inbox);
18:    }
19:    last = t;
20:    next = last + model.time_advance();
21:   } else {
22:     if (inbox != empty) {
23:      model.external_transition(t - last, inbox);
24:      last = t;
25:      next = last + model.time_advance();
26:     }
27:   }
28:   inbox= empty;
29:   outbox = empty;
30:  } // end state_transition
31:  next_time() {
32:   return next;
33:  } // end next_time
34:  get_outbox(){
35:   return outbox;
36:  } // end get_outbox
37:  add_to_inbox(message){
38:   inbox.add(message);
39:  } //end add_to_inbox
40: }//end SIMULATOR
```

Fig. 7. Pseudocode for SIMULATOR component.

execute_output function on every SIMULATOR subcomponent. The route_messages function checks the outboxes on SIMULATOR subcomponents. If the outbox is not empty, then it adds that outbox to the inbox of the receiver component. Then, execute_transition_functions calls the execute_transition function on every SIMULATOR subcomponent. Finally, the next_time function is used on each SIMULATOR subcomponent to compares the values obtained to determine the minimum of them.

To conclude this algorithm, must analyze the SIMULATOR component. In Fig. 7 we can see the pseudocode for this component.

In this case, the *inbox* variable is used to receive messages from components coupled to its input. The outbox variable is used to send messages to components coupled to its output. Then, there is a last variable used to store the time for the last event in the component as well as a next variable, which we use to store the time for the next event. Finally, there is a reference to the atomic model simulated by this component.

The initialize function here stores the time received as parameter in the last variable and then calculates the time_advance function to determine the time for its next event. The execute_output_function checks if the time for next event in the component is equal to the next time in the simulation. If this is the case, then SIMULATOR executes its output function and stores it in its outbox variable. The state_transition function performs two checks. First, to see if the component is imminent (i. e. if the time for the next event in the component is equal to the next simulation time). Then, it checks whether its inbox is empty or not. According to the results of these checks, we define which actions to execute:

- If the component is imminent and the inbox is empty, the component executes the internal transition function.
- If the component is imminent and the inbox is not empty, the component executes the confluent transition function.
- If the component is not imminent and the inbox is not empty, the component executes the external transition function.
- In any other case, the component does not execute a state transition.

To conclude, the input and output variables are reset to empty. Then, the next_time function returns the value of its next scheduled time. The get_outbox function returns the value of the outbox. Finally, the add_to_inbox function receives as a parameter a message and adds it to the inbox variable.

The algorithm in Cadmium works by the interaction among one ROOT-COORDINATOR and FLAT-COORDINATOR components; and a set of SIMULATOR components, through function calls and returns. The advantage of this algorithm is that allows to execute DEVS simulation in a simple way in shared memory architectures. The modular components are defined according to the PDEVS abstract simulator, and the algorithm follows the PDEVS simulation protocol.

B. Shared Memory Parallel Programming

Our research focuses on efficient algorithms to execute DEVS simulations on parallel shared-memory computers. To do this it is important to understand the parallel computer architectures and how we can program them to obtain high performance. Parallel shared-memory computers have become the main trend in the past decades due to the limitations in the improvement of single processor machines. These multi-processors consist of tightly coupled processors whose coordination and usage are typically controlled by a single operating system. They share memory through a shared address space [30]. The most common design for modern multi-processors is called multicore, where several processors (called cores), are placed inside a single computer chip.

To develop programs for multicore computers, the natural programming model is the thread model, where all threads have access to shared variables. From all the libraries and tools available, OpenMP [31] stands over the rest by providing a simple way to implement multi-threaded programs and is therefore the tool of choice for our research. OpenMP is a portable standard for the programming of shared memory systems [32]. The programming model of OpenMP is based on cooperating threads running simultaneously on multiple processors or cores. OpenMP provides with directives to synchronize threads, declare shared and private variables, loop parallelism, scheduling algorithms, and mechanism to assign threads to specific cores in the architectures.

```
1: PARALLEL-ROOT-COORDINATOR {
2:  variables: PARALLEL-FLAT-COORDINATOR, next_time;
3:  execute_simulation(initial_time, simulation_time) {
4:  Create threads, shared variables:
    PARALLEL-FLAT-COORDINATOR, next_time
5:   PARALLEL-FLAT-COORDINATOR.initialize(initial_time);
6:   next_time = PARALLEL-FLAT-COORDINATOR.next_time();
7:   while(next_time<simulation_time) {
8:    PARALLEL-FLAT-COORDINATOR.execute_output_functions
      (next_time);
9:    PARALLEL-FLAT-COORDINATOR.route_messages();
10:   PARALLEL-FLAT-COORDINATOR.execute_state_transitions
      (next_time);
11:   next_time = PARALLEL-FLAT-COORDINATOR.next_time();
12:  } // end while
13:  Destroy threads
14:  } // end execute-simulation
15:} // end PARALLEL-ROOT-COORDINATOR
```

Fig. 8. Pseudocode for PARALLEL-ROOT-COORDINATOR component.

IV. A Parallel Algorithm to Execute PDEVS Simulations

As discussed earlier, the main contribution of this work is to propose an algorithm to execute the PDEVS simulation protocol in parallel shared memory computers. The main advantage of shared memory computers is that the complete simulation data is available for all computation cores. By contrast, in distributed memory platforms, simulation components must be distributed on different nodes in the platform.

The algorithm we propose takes as a base the sequential algorithm for flattened PDEVS abstract simulators presented in Section III. We add parallel execution in all the tasks required to execute the PDEVS simulation protocol. The main idea is to execute each of the tasks in parallel using threads. Each thread calls and executes in parallel a subset of the functions needed to perform the tasks. After each task, the threads perform a synchronization to guarantee the complete execution of a task before starting the next one.

To define this algorithm, we propose specialized versions for the components used in the flattened PDEVS simulation algorithm in Cadmium. We propose the PARALLEL-ROOT-COORDINATOR, PARALLEL-FLAT-COORDINATOR and PARALLEL-SIMULATOR components, as an extension of the ROOT-COORDINATOR, FLAT-COORDINATOR and SIMULATOR in the sequential version, respectively.

In Fig. 8 we can see a definition for PARALLEL-ROOT-COORDINATOR. The definition is similar to the sequential version, but here execute_simulation starts by creating threads that share two variables: a PARALLEL-FLAT-COORDINATOR component, and the next_time variable. After this, the threads execute in parallel the same tasks as in the sequential version. The difference is that the threads call and execute functions in parallel. The threads call and execute the initialize and next_time functions in parallel on the PARALLEL-FLAT-COORDINATOR component. Like in the sequential version, after these two tasks, a simulation loop starts, and each cycle executes the threads in parallel four functions: execute_output_functions, route_messages, execute_state_transitions, and next_time.

```
1: PARALLEL-FLAT-COORDINATOR {
2:  variables: subcomponents, internal_couplings, outbox,
    inbox, next_time;
3:  initialize(initial_time){
4:   parallel-for each PARALLEL-SIMULATOR in subcomponents:
5:    PARALLEL-SIMULATOR.initialize(initial_time);
6:   end-parallel-for
7:   synchronization
8:  }//end init;
9:  execute_output_functions(next_time){
10:  parallel-for each PARALLEL-SIMULATOR in subcomponents:
11:   PARALLEL-SIMULATOR.execute_output_function(next_time);
12:  end-parallel-for
13:  synchronization
14: }//end execute_output_functions;
15: route_messages(){
16:  parallel-for each coupling in internal_couplings:
17:   outbox = coupling.origin.get_outbox();
18:   if(coupling.origin.get_outbox() != empty)
19:    coupling.destiny.add_to_inbox(outbox);
20:   end-if
21:  end-parallel-for
22:  synchronization
23: }
24: execute_state_transitions(next_time){
25:  parallel-for each PARALLEL-SIMULATOR in subcomponents:
26:   PARALLEL-SIMULATOR.execute_state_transition(next_time);
28:  end-parallel-for
29:  synchronization
30: }//end execute_state_transitions
31: next_time(){
32:  next_time = subcomponents.first.next_time();
33:  parallel-for-reduction each PARALLEL-SIMULATOR in
     subcomponents:
34:   if(PARALLEL-SIMULATOR.next_time() < next_time)
35:    next_time = PARALLEL-SIMULATOR.next_time();
36:   end-if
37:  end-parallel-for-reduction
38:  synchronization
39:  return next_time;
40: }//end next_time
41:}//end PARALLEL-FLAT-COORDINATOR
```

Fig. 9. Pseudocode for PARALLEL-FLAT-COORDINATOR component.

The next component we need to execute this algorithm is a PARALLEL-FLAT-COORDINATOR. We can see the pseudocode for this component in Fig. 9.

The initialize function and takes the initial time for the simulation as parameter. Like in the sequential version, this function calls the initialize function on each subcomponent. The difference is that in this version, we do this in parallel by assigning the functions to different threads. Each thread is responsible for executing the function on a subset of the PARALLEL-SIMULATORs. These functions are independent and therefore can execute in parallel. To finish this function, the threads perform a synchronization to guarantee that all of them finished before starting the next task. The execute_output_functions routine calls the execute_outputs function on every PARALLEL-SIMULATOR subcomponent in parallel. To do this, each thread is responsible for executing on a subset of the PARALLEL-SIMULATORs. Once again, these functions are independent tasks and therefore can execute in parallel. To finish this function, the threads perform a synchronization. These tasks are independent and can execute in parallel. However, a data race can occur if more than one thread is adding a message on the same input variable at the same time. We solved this issue adding a lock on the inbox of the PARALLEL-SIMULATOR components. Like in the other functions in this component, the threads perform

```
1: PARALLEL-SIMULATOR {
2:  variables: inbox, outbox, last, next, model;
3:  init(t) {
4:   last = t;
5:   next = t + model.time_advance();
6:  }
7:  execute_output_function(t) {
8:   if (next==t) {
9:    outbox = model.output_function();
10:  }
11: }
12: state_transition(t) {
13:  if (t == next) {
14:   if (inbox == empty) {
15:    model.internal_transition();
16:   } else {
17:    model.confluent_transition(t - last, inbox);
18:   }
19:   last = t;
20:   next = last + model.time_advance();
21:  } else {
22:   if (inbox != empty) {
23:    model.external_transition(t - last, inbox);
24:    last = t;
25:    next = last + model.time_advance();
26:   }
27:  }
28:  inbox= empty;
29:  outbox = empty;
30: }
31: next_time() {
32:  return next;
33: }
34: get_outbox(){
35:  return outbox;
36: }
37: add_to_inbox(message){
38:  lock(inbox);
39:  inbox.add(message);
40:  unlock(inbox);
41: }
42: }//end PARALLEL-SIMULATOR
```

Fig. 10. Pseudocode for PARALLEL-SIMULATOR component.

a synchronization before they finish this function. The execute_transition_functions routine calls the execute_transition function on every PARALLEL-SIMULATOR subcomponent in parallel. To do this. each thread is responsible for executing on a subset of the PARALLEL-SIMULATORs. Once again, these functions are independent tasks and therefore can executed in parallel. Finally, the next_time function obtains the next_time for each PARALLEL-SIMULATOR subcomponent and compares the values obtained to determine the minimum of them. This is implemented in parallel through a parallel reduction. A parallel reduction is a type of operation used to reduce the elements of an array into a single result. In this function, each thread calculates a partial minimum result on a subset of times and saves it on a private copy of the variable. Once they finish, the partial results are compared to obtain the minimum of all of them. After this, the threads perform a synchronization.

To finish the description of our parallel algorithm in Fig. 10 we can see the pseudocode for the proposed PARALLEL-SIMULATOR component.

The variables and functions in PARALLEL-SIMULATOR are the same as in the SIMULATOR component in the sequential version presented on Section II. The only difference is in the definition of the add_to_inbox function. In this function we added a lock on the inbox variable. Each thread adding a message

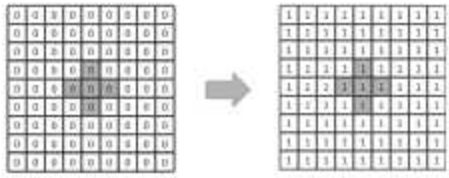

Fig. 11. Synthetic benchmark model example.

first locks the inbox, then adds the message and finally unlocks the inbox. If more than one thread is trying to add a message at the same time, they will perform this one at the time, guaranteeing a correct execution.

As we saw in the pseudocode of the proposed components, this algorithm works by the interaction among a PARALLEL-ROOT-COORDINATOR, a FLAT-COORDINATOR, and a set of PARALLEL-SIMULATOR components. This algorithm extends previous work by providing additional parallelism in the execution of the PDEVS simulation protocol. In [7] only parallelism in the execution of output and transition functions is mentioned. As we saw, here we extend this idea by executing in parallel three addition tasks: initialize components, obtain the times for next events, and route messages. Furthermore, the main advantage of this algorithm is that it allows a simple parallel execution. It does not require complex synchronization mechanism like in the methods derived from the Logical Processes approach. In addition, it allows to obtain the same results as in the sequential version in all scenarios. In the next section we present an experimental evaluation to show how this algorithm performs.

V. Experimental Evaluation

A. Experimental Setup

Our experimental setup used to perform our evaluation used two independent models; one communication intensive and the second one is computation intensive. The first model is built as a synthetic benchmark. The model uses a 2D cell space with a one-digit value as the state of each cell. Each cell uses a Von Neumann neighborhood with five neighbors, including itself. The rules the cells evaluate are simple, they start with a zero value in their state, and change their state from zero to one, and vice versa throughout the simulation, as seen in Fig. 11.

All cells change their state on every simulation cycle. As we can see, the main effort in executing simulations for this model is in communicating the state value to the neighbors. To evaluate our algorithm on several scenario sizes for this problem, we used cell spaces ranging from 10000 to 100000 cells. All scenarios were executed for 500 simulation steps. We chose this problem dimensions and number of steps to evaluate model scenarios that require large amounts of time to execute, and therefore require acceleration. In addition, we replicated all experiments 10 times and calculated the average times. All scenarios were executed for 500 simulation steps, and we replicated the results on each scenario 10 times and calculated the average times. We designed

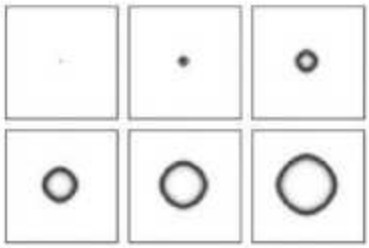

Fig. 12. SIR execution example [33].

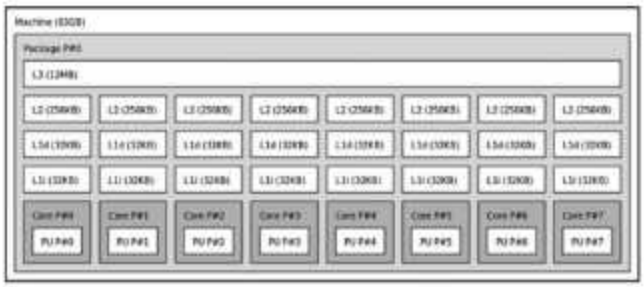

Fig. 13. Desktop computer architecture.

Fig. 14. Server computer architecture.

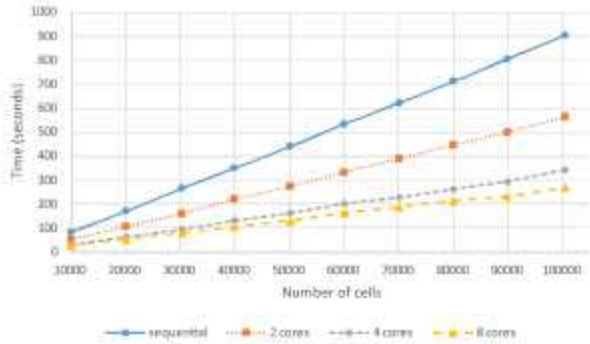

Fig. 15. Time results for the synthetic benchmark problem executed on Desktop computer.

our experiments to evaluate this problem on a range of cell space sizes and executing several simulation steps.

The second experiment uses an epidemiological model to simulate epidemic spreading. This model divides the population into three classes of individuals: those that are Susceptible, Infected, and Recovered from the disease (SIR). Each cell represents a subset of the population and the state for each cell stands for the portion of SIR individuals in the cell. Due to its implementation with the Cell-DEVS formalism, only those cells that are about to change their state will transmit messages to their neighbors, therefore not all of them will execute in every step (which is the case in the benchmark model presented in Fig. 11). In Fig. 12 we can see that the center cell is initialized with infected population, and the disease spreads from there.

All the details for this model can be found in [36]. The equations in this model must be solved in each cell (a compute intensive model). In a similar way as with the benchmark problem, to evaluate this problem over several cell space sizes, we performed experiments with scenarios ranging from 10000 to 100000 cells. All scenarios were executed for 500 simulation steps, and we replicated the results on each scenario 10 times and calculated the average times. Similar as with the benchmark problem, we designed our experiments to evaluate this problem on a range of cell space sizes and executing several simulation steps.

To evaluate our implementation, we use two modern multicore architectures: a desktop computer with an intel octa-core i7-9700 processor and 64 Gb of RAM memory, which we can see on Fig. 13; and a server equipped with two octa-core Xeon E5-2609 v4, giving a total of 16 cores, and 24 Gb of RAM memory, which we can see on Fig. 14 [34].

The simulator we use to perform our experimental evaluation is the Cadmium simulator, presented in Section III. Our parallel algorithm was programmed with the OpenMP library and was added as an extension for Cadmium.

To compare the results obtained we use two reference points. First, a naive parallel implementation. This version uses the ideas presented in [7]. It is implemented in the same way as our algorithm, but only executes in parallel two tasks: the output and state transition functions. The rest of the algorithm is executed sequentially. The second reference point we use to compare the results obtained in our algorithm is the theoretical ideal speedup limit. This is determined by the number of processors or cores in a parallel computer. It indicates the acceleration limit we can obtain using the best possible parallelization.

B. Experimental Results

1) Execution Time: In this section we show the results for the execution time obtained on the baseline sequential algorithm and on our proposed parallel algorithm. The results show the times obtained increasing the number of cells and the number of cores used on the parallel algorithm. In Fig. 15 we can see the results

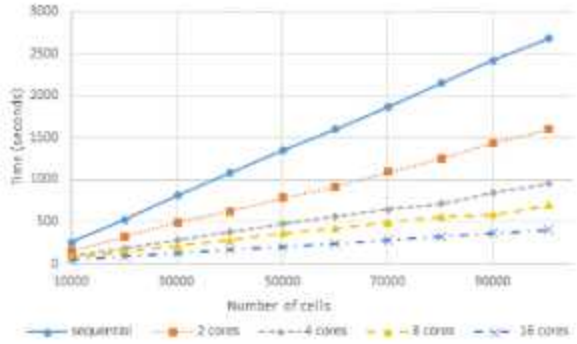

Fig. 16. Time results for the synthetic benchmark problem executed on Server computer.

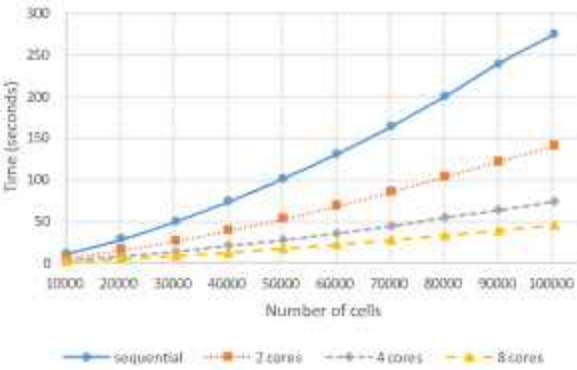

Fig. 17. Time results for the epidemiological problem executed on Desktop computer.

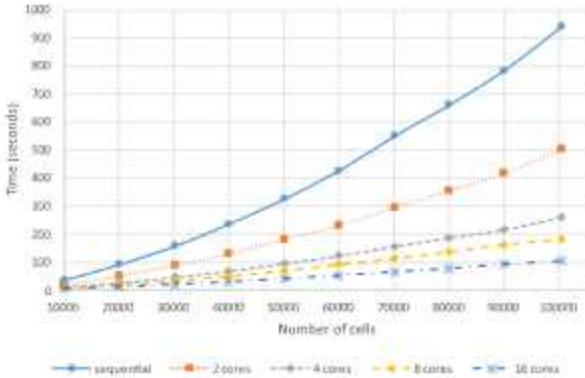

Fig. 18. Time results for the epidemiological problem executed on Server computer.

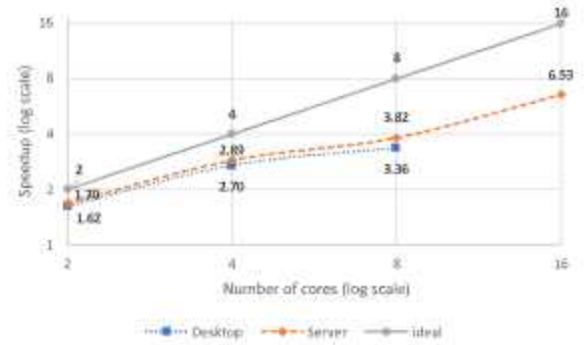

Fig. 19. Average speedup for the synthetic benchmark problem using different number of cores.

obtained for the synthetic benchmark problem executed on the Desktop computer.

As expected, the time increases when the problem dimension increases for every number of cores used. This is because there are more cells, and this requires larger computations to route messages, perform state transitions and calculate the next time on each step of the simulations. In addition, we can see that our algorithm scales, time decreases when we increase the number of cores used. We perform more computations in parallel and the best results are obtained using all cores available in the computer. Similarly, in Fig. 16 we can see the results obtained for the synthetic benchmark problem executed on the Server computer.

In this architecture we can also see, as expected, that time increases when the size of the problem increases for every number of cores used. Once again, this is because when the problem size increases there are more cells, and simulations require larger computations to communicate messages, perform state transitions and calculate the next time on each step of the simulations. In addition, we can see that this algorithm also scales in this computer, time decreases when we increase the number of cores used. This is because of the additional parallel computations the computer performs when using more cores. The best results are obtained using all sixteen cores in the architecture.

Next, we conducted the second set of experiments on the epidemiological model. In Fig. 17 we can see the times results for the sequential version and our parallel algorithm using different number of cores on the Desktop computer.

Like in the previous results presented, time increases when the problem dimension increases, and time decreases when we increase the number of cores used, as expected.

Finally, in Fig. 18 we can see the times results for the sequential version and our parallel algorithm using different number of cores on the Server computer.

Once again, like expected, the time increases when the problem dimension increases in all versions. In addition, like in the synthetic problem, we can see that the time decreases when we increase the number of cores used in the architecture. The best results are obtained using all cores available in the computer.

2) Speedup: As we saw in the previous section, time decreases when we increase the number of cores used. However, we want to analyze how much acceleration we obtain using different number of cores with our algorithm. In Fig. 19 we can see the average speedup results for the synthetic benchmark.

As we can see, the speedup increases when we increase the number of cores used in both architectures. This is because the algorithm benefits from more computing resources to execute the computations, and therefore it reduces the execution time. To compare our results, we show the ideal speedup that can be obtained for each number of cores. Even though the overhead of executing in parallel does not allow us to achieve the

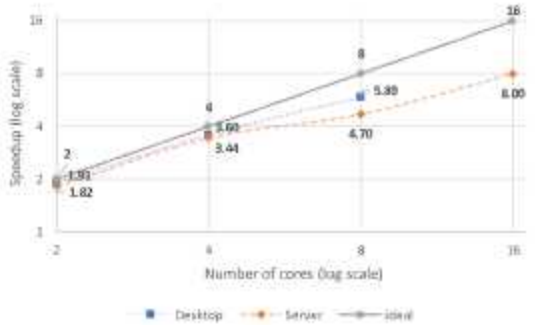

Fig. 20. Average speedup for the epidemiological problem using different number of cores.

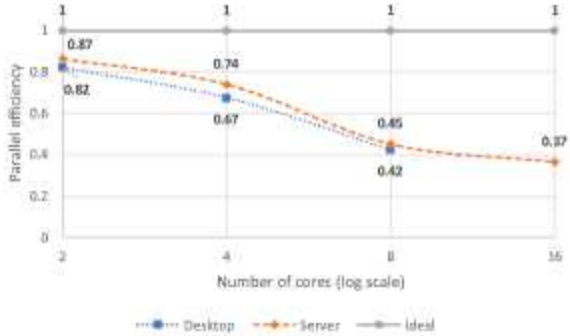

Fig. 21. Average parallel efficiency for the synthetic benchmark problem using different number of cores.

ideal speedup, our algorithm improves the performance of the sequential version by accelerating the simulations.

Next, in Fig. 20 we can see the average speedup results for the epidemiological problem.

Like on the results for the synthetic benchmark, speedup increases when we increase the number of cores. This is because our algorithm can use more cores to execute computations in parallel. To compare our results, we show the theoretical parallelization limit that can be obtained with each number of cores. In this problem we can see that the speedup with 2 and 4 cores is almost linear. However, with 8 and 16 cores is close to P/2x, where P is the number of cores used. This is due to the higher overhead in synchronization when we use more cores.

Here we can also see that even though the overhead of executing in parallel in a real application does not allow us to achieve the ideal speedup, our algorithm improves the performance of the sequential version by accelerating the simulations.

It is interesting to note that the speedup for the epidemiological problem is higher than the obtained for the synthetic benchmark. This is because the epidemiological problem is more computing intensive than the synthetic benchmark. In the synthetic benchmark, each cells performs a very simple update on the digit of its state. By contrast, in the epidemiological problem each cell must perform a more complex computation to update its state with the received state from their neighbors.

3) Parallel Efficiency: As we saw in the previous section our algorithm allows to accelerate the execution times of the sequential algorithm However, we want to evaluate as well how efficiently our algorithm uses the cores on the architectures. To achieve this, in this section we present an analysis of the parallel efficiency obtained by our algorithm. In Fig. 21 we can see the parallel efficiency results for the synthetic benchmark.

We can notice a similar behavior in both architectures, parallel efficiency decreases when we increase the number of cores. This is expected on a synchronous algorithm like the one we present in this work. Even though all tasks on the algorithm execute in parallel, the overhead of coordinate and synchronize the threads decreases the utilization of the cores. Next, in Fig. 22 we can see the parallel efficiency results for the epidemiological problem.

Like on the results obtained on the benchmark problem, both architectures follow the same pattern. As we discuss previously this is expected in a synchronous algorithm. This is a

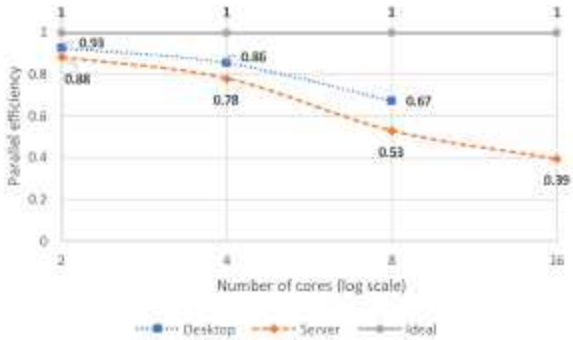

Fig. 22. Average parallel efficiency for the epidemiological problem using different number of cores.

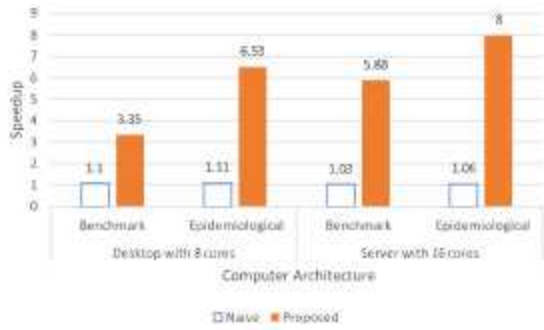

Fig. 23. Average speedup for our proposed algorithm compared with the naïve parallelization approach.

limitation for our approach, explicit synchronization becomes more expensive with more cores, and this overhead reduces the performance compared with the optimal core utilization. However, the big advantage of our approach is that it allows to accelerate execution several times, while guarantee correct results in all scenarios.

4) Comparison With Previous Approach: To conclude our experimentation, we perform a comparison for our approach with the previous ideas proposed. In Fig. 23 we can see a comparison of the average speedup obtained with our algorithm against the naïve parallelization approach.

As we can see, the average speedup for the naive paralleled approach using both architectures to execute the two problems

evaluated is smaller than the one we obtained with our algorithm. The naive approach offers very limited acceleration, between 1.03 and 1.1 speedup in our experiments. This is because there is a small benefit on the computations performed on parallel to execute the output and state transition functions, compared with the overhead created to execute in parallel in this version. By contrast, as we show in our experimental evaluation, our algorithm accelerates several times the execution for both problems in the two architectures considered. Our speedup results using all cores on the architectures range from 3.35 to 8 for the different problems executed on both architectures. This is because our algorithm executes completely in parallel and allows to also accelerate the message routing and the calculations for the next times in the simulations.

VI. Conclusion and Future Work

We presented an algorithm to execute DEVS simulations efficiently on parallel shared-memory architectures. This work extends the previous ideas in the literature by adding mechanisms to execute in parallel additional tasks in the PDEVS simulation protocol. As we mentioned, in previous works it is only considered the execution in parallel of simultaneous output and transition functions. Here, we extended this approach by parallelizing additional tasks: the initialization, the routing of messages, and the mechanism to obtain the time for the next events. This way, the algorithm we presented takes advantage of parallel execution on all tasks in the PDEVS simulation protocol. Each of these tasks are distributed among the cores in shared memory architectures and this way, we accelerate simulations. The main advantage of this algorithm is that it allows to execute in parallel in a simple way and achieves the same results as in the sequential version in all scenarios.

In addition, the experimental results we obtained show how we can accelerate the execution several times for both a synthetic benchmark and a real-world problem in two independent computers, with different hardware characteristics. We showed that our algorithm scales when we add cores to its execution. Even though parallel efficiency decreases when we increase the number of cores, our algorithm accelerates the execution several times, while guaranteeing correct results in all scenarios. Finally, we show how this idea improves the performance over the previous works in the parallelization of the PDEVS simulation protocol.

As future work, we plan to evaluate performance on architectures with more cores, as well as adding new case studies. In addition, we plan to evaluate the energy used by computer architectures when using different number of cores, and to implement this algorithm on Graphic Processing Units (GPU) to extend the algorithm's usability in a broader range of shared-memory computing platforms.

References

[1] G. S. Fishman, *Discrete-Event Simulation*. New York, NY, USA: Springer, 2001.

[2] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. San Diego, CA, USA: Academic, 2000.

[3] Q. Liu and G. Wainer, "A performance evaluation of the lightweight time warp protocol in optimistic parallel simulation of DEVS-based environmental models," in *Proc. ACM/IEEE/SCS 23rd Workshop Princ. Adv. Distrib. Simul.*, 2009, pp. 27–34.

[4] J. Nutaro, "On constructing optimistic simulation algorithms for the discrete event system specification," *ACM Trans. Model. Comput. Simul.*, vol. 19, no. 4, pp. 1–21, 2009.

[5] S. Jafer and G. Wainer, "Conservative synchronization methods for parallel DEVS and Cell-DEVS," in *Proc. Summer Comput. Simul. Conf.*, 2011, pp. 60–67.

[6] Q. Liu and G. Wainer, "Multicore acceleration of discrete event system specification systems," *Simulation*, vol. 88, pp. 801–831, 2012.

[7] B. Zeigler, "Using the parallel DEVS protocol for general robust simulation with near optimal performance," *Comput. Sci. Eng.*, vol. 19, no. 3, pp. 68–77, 2017.

[8] J. Lanuza, G. G. Trabes, and G. A. Wainer, "Parallel execution of DEVS in shared-memory multicore architectures," in *Proc. Spring Simul. Conf.*, 2020, pp. 1–11.

[9] S. Jafer, "Parallel simulation techniques for large-scale discrete-event models," Ph.D. dissertation, Dept. Systems Comput. Eng., Carleton Univ., Ottawa, Ontario, 2011.

[10] A. C. Chow and B. P. Zeigler, "Parallel DEVS: A parallel, hierarchical, modular, modeling formalism," in *Proc. Winter Simul. Conf.*, 1994, pp. 716–722.

[11] A. C. Chow, B. P. Zeigler, and D. H. Kim, "Abstract simulator for the parallel DEVS formalism," in *Proc. 5th Conf. AI Simul. Plan. High Autonomy Syst.*, 1994, pp. 157–163.

[12] K. Kim, W. Kang, B. Sagong, and H. Seo, "Efficient distributed simulation of hierarchical DEVS models: Transforming model structure into a non-hierarchical one," in *Proc. 33rd Annu. Simul. Symp.*, 2000, pp. 227–233.

[13] E. Glinsky and G. Wainer, "Performance analysis of real-time DEVS models," in *Proc. Winter Simul. Conf.*, 2002, pp. 588–594.

[14] E. Glinsky and G. Wainer, "Definition of real-time simulation in the CD++ toolkit," in *Proc. Summer Comput. Simul. Conf.*, 2002.

[15] G. G. Trabes, V. Gil-Costa, and G. A. Wainer, "Complexity analysis on flattened PDEVS simulations," in *Proc. Winter Simul. Conf.*, 2021, pp. 1–12.

[16] G. Zacharewicz, M. E.-A. Hamri, C. Frydman, and N. A. Giambiasi, "A generalized discrete event system (G-DEVS) flattened simulation structure: Application to high-level architecture (HLA) compliant simulation of workflow," *Simulation*, vol. 86, no. 3, pp. 181–197, 2010.

[17] R. Franceschini and P. A. Bisgambiglia, "Decentralized approach for efficient Simulation of DEVS models," in *Proc. IFIP Int. Conf. Adv. Prod. Manage. Syst.*, 2014, pp. 336–343.

[18] P. A. Bisgambiglia and P. Bisgambiglia, "DecPDEVS: New simulation algorithms to improve message handling in PDEVS," *Open J. Modelling Simul.*, vol. 9, no. 1, pp. 172–197, 2021.

[19] R. M. Fujimoto, *Parallel and Distribution Simulation Systems*. 1st. ed. New York, NY, USA: Wiley, 1999.

[20] R. M. Fujimoto, "Parallel discrete event simulation," *Commun. ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[21] R. M. Fujimoto et al., "Parallel discrete event simulation: The making of a field," in *Proc. Winter Simul. Conf.*, 2017, pp. 262–291.

[22] M. Marín, V. Gil-Costa, C. Bonacic, and R. Solar, "Approximate parallel simulation of web search engines," in *Proc. ACM SIGSIM Conf. Princ. Adv. Discrete Simul.*, 2013, pp. 189–200.

[23] A. Inostrosa-Psijas, V. Gil-Costa, R. Solar, and M. Marín, "Load balance strategies for DEVS approximated parallel and distributed discrete-event simulations," in *Proc. 23rd Euromicro Int. Conf. Parallel Distrib. Netw.-Based Process.*, 2015, pp. 337–340.

[24] A. Inostrosa-Psijas, V. Gil-Costa, M. Marín, and G. Wainer, "Semi-asynchronous approximate parallel DEVS simulation of web search engines," *Concurrency Computation Pract. Experience*, vol. 30, 2015, Art. no. e4149.

[25] A. Eker, Y. Arafa, A.-H. A. Badawy, N. Santhi, S. Eidenbenz, and D. Ponomarev, "Load-aware dynamic time synchronization in parallel discrete event simulation," in *Proc. ACM SIGSIM Conf. Princ. Adv. Discrete Simul.*, 2021, pp. 95–105.

[26] A. Adegoke, H. Togo, and M. K. Traoré, "A unifying framework for specifying DEVS parallel and distributed simulation architectures," *Simulation*, vol. 89, no. 11, pp. 1293–1309. 2013.

[27] B. Cardoen, S. Manhaeve, Y. Van Tendeloo, and J. Broeckhove, "A PDEVS simulator supporting multiple synchronization protocols: Implementation and performance analysis," *Simulation*, vol. 94, no. 4, pp. 281–300. 2018.

[28] L. Belloli, D. Vicino, C. Ruiz-Martin, and G. Wainer, "Building DEVS models with the cadmium tool," in *Proc. Winter Simul. Conf.*, 2019, pp. 45–59.
[29] D. Vicino, D. Niyonkuru, G. Wainer, and O. Dalle, "Sequential PDEVS architecture," in *Proc. Symp. Theory Model. Simul.: DEVS Integrative M&S Symp.*, 2015, pp. 165–172.
[30] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach*. 6th ed. Cambridge, MA, USA: Morgan Kaufmann. 2017.
[31] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998.
[32] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*. Burlington, MA, USA: Morgan Kaufmann. 2001.
[33] S. H. White, A. M. Del Rey, and G. R. Sánchez, "Modeling epidemics using cellular automata," *Appl. Math. Computation*, vol. 186, no. 1, pp. 193–202. 2007.
[34] F. Broquedis et al., "HWLOC: A generic framework for managing hardware affinities in HPC applications," in *Proc. Euromicro Int. Conf. Parallel Distrib. Netw.-Based Process.*, 2010, pp. 180–186.

Guillermo German Trabes received the degree in computer science from Universidad Nacional de San Luis (UNSL), in 2016. He is currently working toward the PhD degree in computer engineering on cotutelle between Carleton University and Universidad Nacional de San Luis. He is also a research assistant with Carleton University and Universidad Nacional de San Luis. His research interests include focus on high performance computing (HPC) and modeling and simulation (M&S).

Gabriel A. Wainer (Senior Member, IEEE) received the PhD degree (Hons.) from UBA/ Université d'Aix-Marseille III, Marseille, France, in 1998. He held visiting positions with the University of Arizona; LSIS (CNRS), Université Paul Cézanne, University of Nice, INRIA Sophia-Antipolis, Université de Bordeaux (France); UCM, UPC (Spain), University of Buenos Aires, National University of Rosario (Argentina) and others. He is currently a full professor with Carleton University, Ottawa, ON, Canada, where he is also the head with the Advanced Real-Time Simulation Laboratory, Centre for advanced Simulation and Visualization (V-Sim). He is also a fellow with the Society for Modeling and Simulation International (SCS). He is also a member of the Board of Directors of the SCS. He is also a member of the Board of Directors of the SCS. He is also a member of the editorial board of the *IEEE Computing in Science & Engineering*, *Wireless Networks* (Elsevier), and *The Journal of Defense Modeling and Simulation* (SCS). He was a recipient of various awards, including the IBM Eclipse Innovation Award, the SCS Leadership Award, the SCS Leadership Award, and various best paper awards, He also received the Carleton University's Research Achievement Awards in 2005 and 2014, the First Bernard P. Zeigler DEVS Modeling and Simulation Award, the SCS Outstanding Professional Award in 2011, the Carleton University's Mentorship Award in 2013, the SCS distinguished professional Award in 2013, the SCS Distinguished Service Award in 2015, the Nepean's Canada 150th Anniversary Medal in 2017, the ACM Recognition of Service Award in 2018, and the IEEE Outstanding Engineering Award (Ottawa Section) in 2019. He was the vice-president of several conferences and the vice-president of several publications. He is also one of the founders of the Symposium on Theory of Modeling and Simulation, SIMUTools, and the Symposium on Simulation of Architecture and Urban Design (SimAUD). He is also the Special Issues editor of *Simulation*.

Veronica Gil-Costa received the MSc and PhD degrees in computer science from the Universidad Nacional de San Luis (UNSL), Argentina, 2006 and 2009, respectively. She is a former researcher with Yahoo! Labs Santiago hosted by the University of Chile. She is currently an associate professor with the University of San Luis and researcher with the National Research Council (CONICET) of Argentina.